Lecture3 无信息搜索

1. 基于目标的 agent

它也叫做 **problem solving agents** 或者 **planning agents**

- agent 朝着一个**目标**努力
- agent 考虑**行为对未来状态的影响**，这意味着它们的工作是识别导致目标的操作或一系列操作
- 形式化为通过可能的解进行**搜索**

无信息搜索与有信息搜索

- 无信息搜索：不知道目标在哪里，随便搜索
- 有信息搜索：知道目标在哪里，规划

搜索问题的解决方案

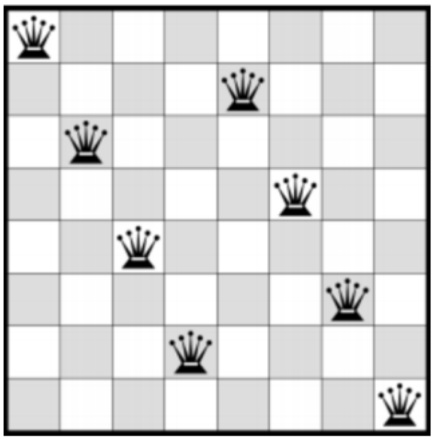
- 通过以下方式定义问题
  - 目标公式化
  - 问题公式化
- 解决问题的过程分为两个阶段
  - 搜索：对几种可能性的思考或“离线”探索
  - 执行找到的解决方案

问题公式化

- **初始状态 Initial state**：agent 开始的状态
- **状态 States**：从初始状态开始，任何行为序列都可以到达所有状态
- **动作 Actions**：对 agent 可用的可能动作，在一个状态 s 下，Actions(s) 返回可以在该状态中可以执行的所有动作
- **转换模型 Transition model**：每个动作所做的描述 Results(s, a)
- **目标检测 Goal test**：确定一个给定的状态是否是目标状态
- **路径开销 Path cost**：为每条路径分配数值成本的函数

2. 典型的搜索问题

八皇后问题



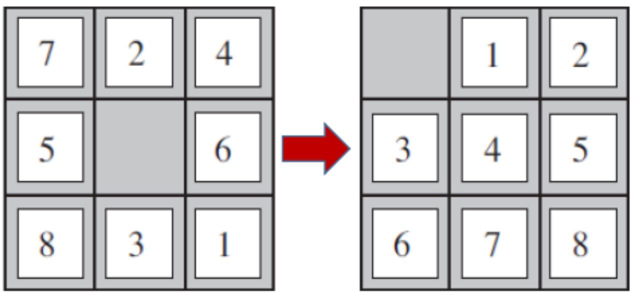
在一个国际象棋棋盘上找到一种布局，放置8个皇后，使得没有皇后攻击任何其他水平，垂直或对角（同一列、同一行、同一对角线上只有一个皇后）

- 初始状态：没有皇后在棋盘上
- 状态：棋盘上 8 张皇后的所有安排
- 动作：将一个皇后添加到任何空的格子里
- 转换模型：更新棋盘
- 目标检测：是否满足八皇后问题的布局
- 路径开销：八皇后问题的开销小，不考虑

暴力搜索时间复杂度

$$64 \times 63 \times 62 \times \ldots \times 57 = 1.8 \times 10^{14}$$

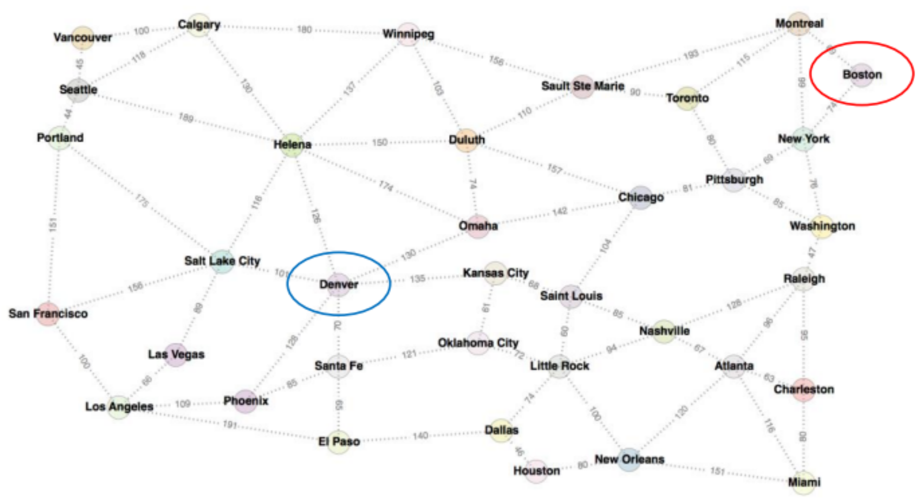
8-拼图问题



每次可以将某个拼图块移动到空的格子处，保证目标为右图的效果

- 初始状态：任何可能的一个状态
- 状态：在 3 x 3 网格上 8 个方片的任何配置
- 动作：将一个方片向上 / 下 / 左 / 右进行移动
- 转换模型：给定一个状态和一个动作，返回结果状态
- 目标检测：是否是按照右上图的顺序完成
- 路径开销：移动方片的次数

寻路问题



在地图寻路的典型例子中，我们需要使用链接或转换从一个位置到另一个位置。应用的例子包括网站上的驾驶方向工具，车内系统等

- 初始状态：在 Boston
- 状态：在任何一个属于寻路范围内的城市
- 动作：移动到一个邻接城市
- 转换：新的城市
- 目标检测：到达 Denver
- 路径开销：路径的长度（以 km 计）

真实生活中的搜索问题

旅行商问题 TSP

找到每个城市游览一次的最短路线

VLSI 的布局

将上百万个元件和连接放置在一个芯片上，以最小化面积，缩短延迟，它们不会重叠，并为布线留出空间

机器人导航

为没有特定路线或连接的机器人寻找路线的特殊情况，机器人在二维或三维空间中导航，其中状态空间和行动空间可能是无限的

自动装配

找到一个装配零件的顺序，这通常是一个困难和昂贵的几何搜索

蛋白质设计

找到一个氨基酸序列，它将折叠成一个具有正确属性的 3D 蛋白质，以治疗某些疾病

2. 搜索的概念

概念定义

- **状态空间 State space**：物理配置
- **搜索空间 Search space**：用搜索树或可能解的图表示的一种抽象结构
- **搜索树 Search tree**：模拟动作的顺序
    - 根：初始状态
    - 分支：动作
    - 节点：动作的结果，一个节点有父节点、子节点、路径、深度、代价，关联状态等
- **扩展 Expand**：给定一个节点，创建所有子节点的函数

搜索空间 Search space

- 搜索空间被分成 3 个区域
    - **已经访问过的： Explored** / Closed List / Visited Set
    - **边界： Frontier** / Ready List / Open List 已经被加入搜索队列去的
    - **未访问：Unexplored**

搜索的本质是将节点从第 3 个区域移动到第 1 个区域，且最核心的策略是来判定这样移动时候所遵循的顺序（Frontier 中的顺序）

图搜索伪代码

```
function GRAPH-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE:

    initialize frontier with initialState
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.remove()
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in forntier ∪ explored:
                frontier.add(neighbour)

    return FAILURE
```

搜索策略

策略，定义为节点扩张的顺序，可以从以下几个维度来考量

- **完成性 Completeness**：如果它存在，算法总能找到解决方案吗？
- **时间复杂度 Time Complexity**： 节点生成 / 扩张所开销的时间
- **空间复杂度 Space Complexity**：在内存中最大的节点数量
- **最佳性 Optimality**：算法总能找到最小的开销吗？

特别地，时间和空间复杂性是通过以下方面来衡量的

- b：搜索树的最大分支个数（每个状态的可能采取动作数量）
- d：解的深度
- m：状态空间的最大深度（可能是 ∞）（有时也记为 D）

搜索的分类

搜索又分为

- 无信息搜索 Uninformed Search
- 有信息搜索 Informed Search

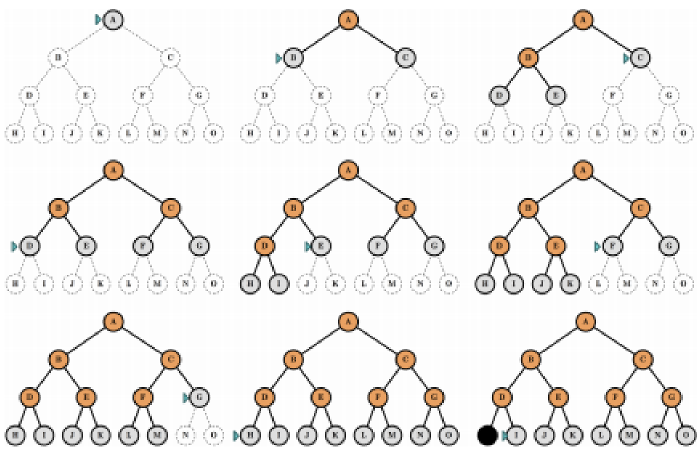
3. 无信息搜索 Uninformed Search

没有对目标的相关信息

策略

- BFS Breadth-first search ： 扩张最浅的节点
- DFS Depth-first search：扩张最深的节点
- DLS Depth-limited search：深度优先，但是有深度限制
- IDS Iterative-deepening search：DLS，但是有增长限制
- UCS：扩张最小成本节点

BFS



扩张最浅的节点

伪代码

```
function BFS(initialState, goalTest)
    returns SUCCESS or FAILURE:

    frontier = Queue.new(initialState) # new
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.dequeue() # new
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in forntier ∪ explored:
                frontier.enqueue(neighbour) # new

    return FAILURE
```

评价指标

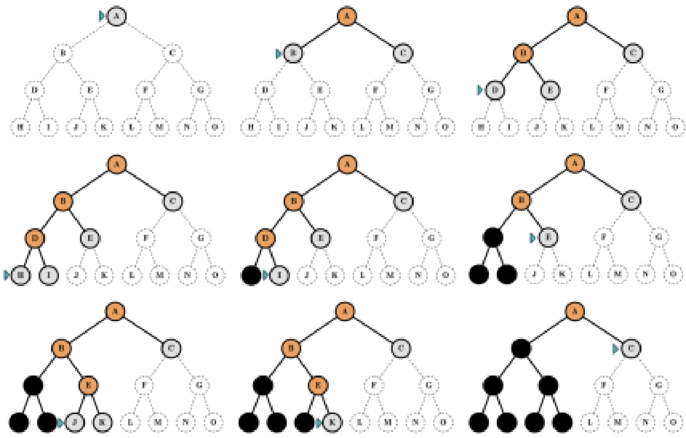
- **完成性 Completeness**：如果 b 是有限的，那么它是可完成的
- **时间复杂度 Time Complexity**：$1+b+b^2+b^3+...+b^d=O(b^d)$
- **空间复杂度 Space Complexity**：$O(b^d)$
- **最佳性 Optimality**：如果开销每一步为 1，那么它是最优的
- **实现**：frontier 使用 Queue

效率差

内存需求 + 指数时间复杂度是 BFS 的最大障碍

Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

DFS



扩张最深的节点

伪代码

```
function DFS(initialState, goalTest)
    returns SUCCESS or FAILURE:

    frontier = Stack.new(initialState) # new
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.pop() # new
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in forntier ∪ explored:
                frontier.push(neighbour) # new

    return FAILURE
```

评价指标

- **完成性 Completeness**：如果 d 是有限的，那么它是可完成的
- **时间复杂度 Time Complexity**：$1+b+b^2+b^3+\ldots+b^m=O(b^m)$
  - 如果 $m>>d$ 那么算法是较差的
  - 但如果解存在的数量很多，可能会比 BFS 快得多
- **空间复杂度 Space Complexity**：$O(bm)$
- **最佳性 Optimality**：不是最优的
- **实现**：frontier 使用 Stack

效率差

指数时间复杂度是 DFS 的最大障碍

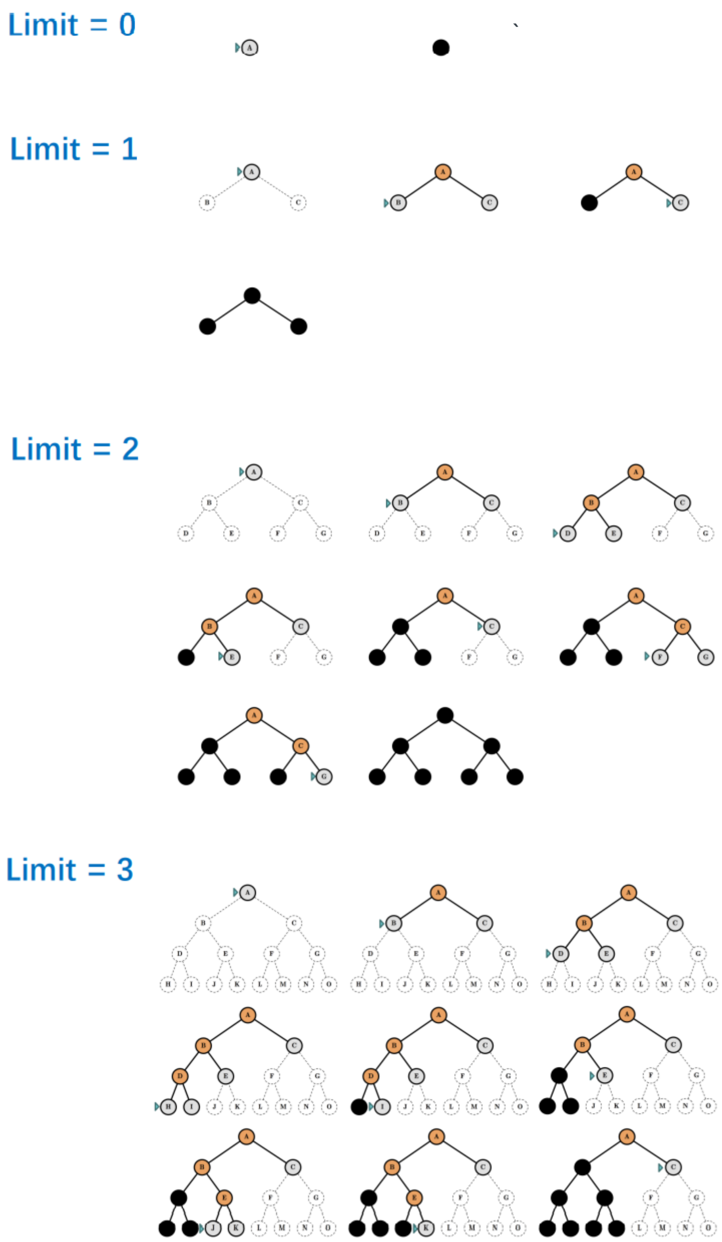
- 不过空间复杂度更小，在 Depth 等于 16 的时候，只需要 156 KB 即可

DLS

最多搜索深度为 L 的 DFS

选择一些深度限制来探索 DFS

IDS



结合了 BFS 和 DFS 的优点

想法：迭代地增加搜索深度限制 L，直到增长深度的限制 D

如果找到一个解，或者 DLS 返回一个失败（没有解），算法将停止

UCS

搜索图中的边可能有权重

BFS 会找到代价可能很高的最短路径

我们要开销最小的而不是深度最浅的解

修改 BFS：按开销不按深度排序 → 展开路径开销最小的节点$g(n)$

伪代码

```
function UCS(initialState, goalTest)
    returns SUCCESS or FAILURE:

    frontier = Heap.new(initialState) # new
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.deleteMin() # new
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in forntier ∪ explored:
                frontier.insert(neighbour) # new
            else if neighbour in frontier:
                # 更新代价函数 g(n)
                # g(n) 可以是从初始节点到目前节点的距离
                frontier.decreaseKey(neighbour) # new

    return FAILURE
```

评价指标

- **完成性 Completeness**：如果开销是有限的，那么它是可完成的
- **时间复杂度 Time Complexity**
  - 假设最优解的开销是 C^*
  - 每一次行动开销最少是 ϵ
  - 有效的深度粗略的记录为 $\frac{C^*}{\epsilon}$
  - 时间复杂度 $O(b^{\frac{C^*}{\epsilon}})$
- **空间复杂度 Space Complexity**：$O(bm)$
- **最佳性 Optimality**：是最优的
- **实现**：frontier 使用 Heap / Priority Queue