人工智能实践
Artificial Intelligence Practice

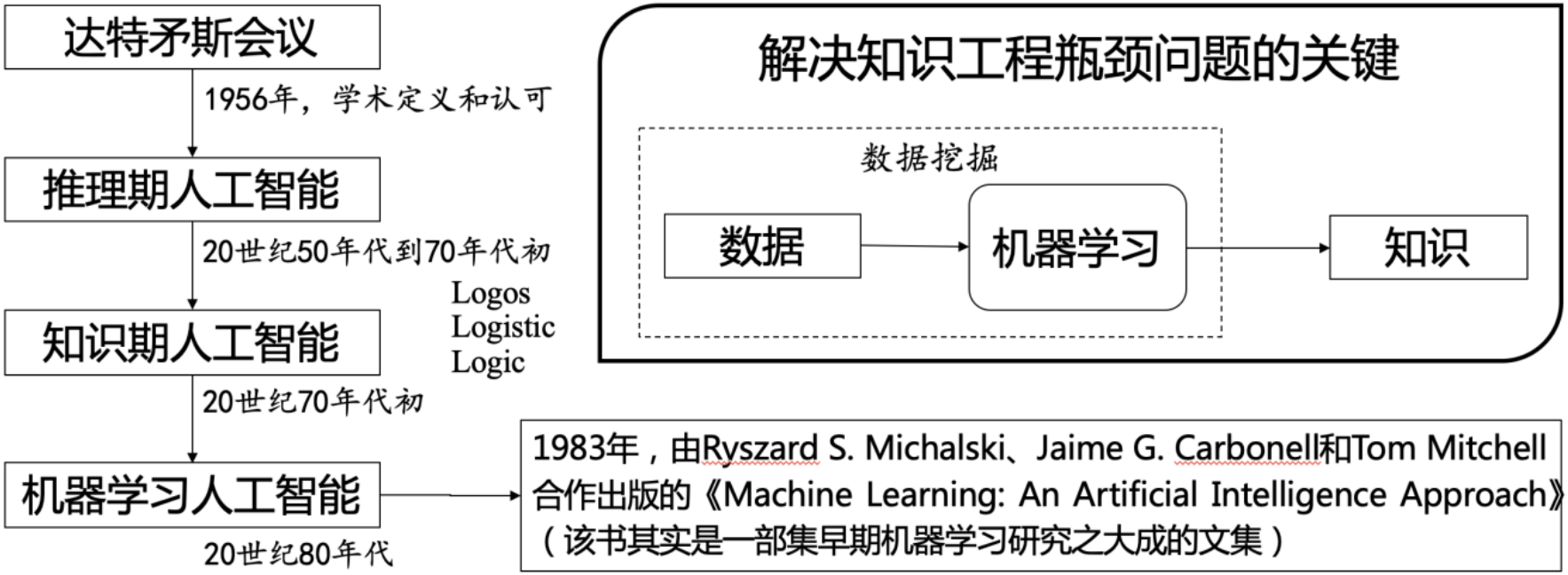
DCS3015 Autumn 2022

Chao Yu （余超）

School of Computer Science and Engineering
Sun Yat-Sen University

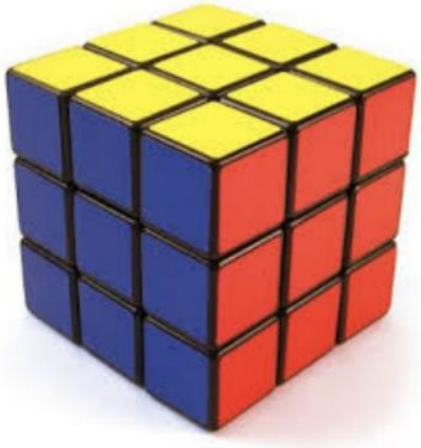
Lecture 2：搜索

AI历史



机器学习成为一个独立的学科领域，两个学派：之于应用统计、之于人工智能

搜索问题的应用



分类和搜索

- 分类器（reflex-based models）

- 搜索（state-based models）

1. 不能简单迭代分类产生动作序列，需要考虑动作对环境的影响；
2. 有时可以用reflex模型（分类器）求解搜索问题

搜索问题定义

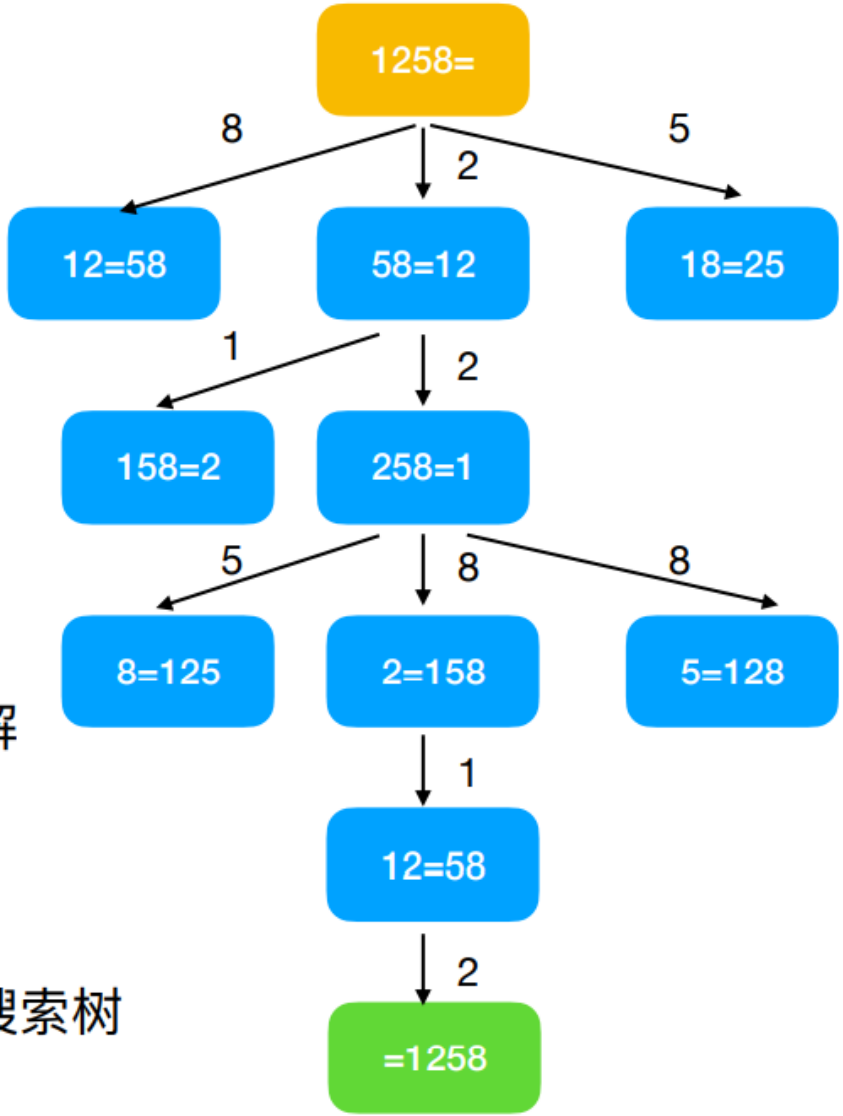
- s: 起始状态
- Actions(s): 状态s下允许的动作集合
- Cost(s, a): 动作a的代价
- Succ(s, a): 状态s执行动作a得到的状态（集合）
- IsGoal(s): 状态s是否目标状态

过桥问题

- 已知ABCD四个人，单独过桥分别需要1,2,5,8分钟
- 最多两个人同时过， 并且只有一个手电筒，每次都需要电 筒，两人过桥按慢的时间算
- 问：四个人过桥最少需要几分钟?

搜索树

- **搜索树**
  - 根节点：起始状态
  - 叶节点：目标状态
  - 每条边是一个动作
  - 边上的数字代表其代价
  - 根节点到叶节点路径代表解
  - **最短路径代表最优解**
  - 代码中不会显式构造具体搜索树



最短路

- 回溯搜索（最简单，穷举所有的路径）

```
def backtrackingSearch(s, path):
    If IsEnd(s): update minimum cost path
    For each action a ∈ Actions(s):
        Extend path with Succ(s, a) and Cost(s, a)
        Call backtrackingSearch(Succ(s, a), path)
    Return minimum cost path
```

- 搜索树：每个状态b个可能动作，解最大长度D个动作
  - Memory: **O(D)**
  - Time: **O(b^D)** [2^50 = 1125899906842624]

深度优先搜索

- **所有动作代价都为0**，或只需任意一个解

```
def backtrackingSearch(s, path):
    If IsEnd(s): Return minimum cost path
    For each action a ∈ Actions(s):
        Extend path with Succ(s, a) and Cost(s, a)
        Call backtrackingSearch(Succ(s, a), path)
    Return minimum cost path
```

- Memory: **O(D)**
- Time: **O(b^D)**

宽度优先搜索

- **所有动作代价都为常数c (c>=0)**
- 按照节点深度次序进行遍历

- 每个状态b个动作，最优解长度为d
  - Memory: **O(b^d)** 比DFS差
  - Time: **O(b^d)** 比DFS好

迭代加深的宽度优先搜索

- **所有动作代价都为常数c (c>=0)**
- DFS(n): 到达第n层后停止搜索
- 调用DFS(1), DFS(2)…
- 每个状态b个动作，最优解长度为d
  - Memory: **O(b^d)** 比DFS差
  - Time: **O(b^d)** 比DFS好

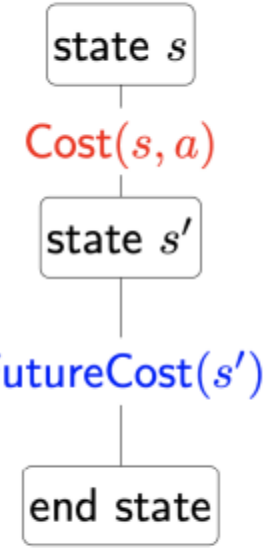
树搜索

- **指数时间复杂度**
- 利用迭代加深减少使用空间

算法	动作代价	空间	时间
回溯搜索	any	O(D)	O(b^D)
深度优先搜索	0	O(D)	O(b^D)
宽度优先搜索	非负常数	O(b^d)	O(b^d)
迭代加深的宽度优先搜索	非负常数	O(d)	O(b^d)

动态规划

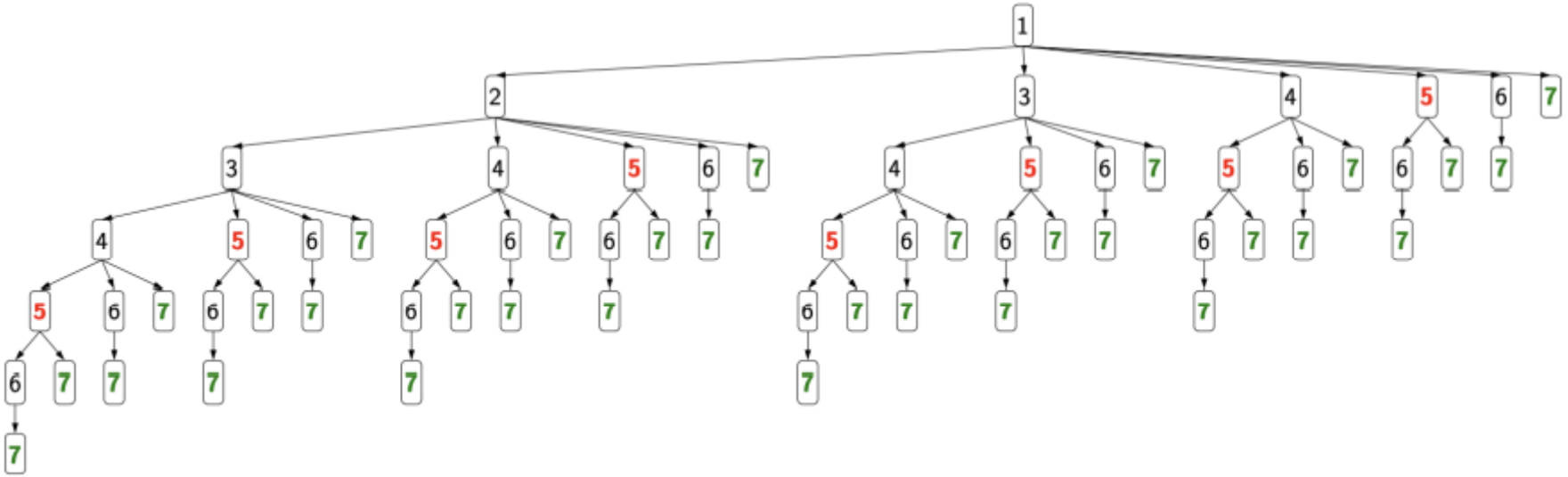
- 从状态s到目标状态的最短路径代价



$$\text{FutureCost}(s) = \begin{cases} 0 & \text{if IsEnd}(s) \\ \min_{a \in \text{Actions}(s)} [\text{Cost}(s,a) + \text{FutureCost}(\text{Succ}(s,a))] & \text{otherwise} \end{cases}$$

动态规划

- shortest path from city 1 to city 7
- Observation: future costs only depend on current city



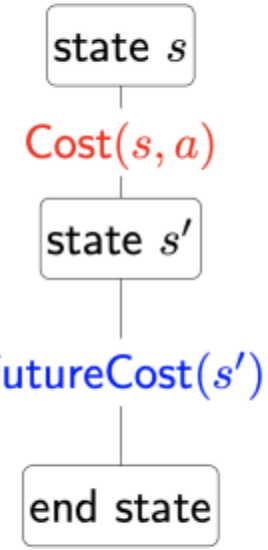
动态规划

- State: ~~past sequence of actions~~ current city
- Exponential saving in time and space!

```
def DynamicProgramming(s):
    If already computed for s, return cached answer.
    If IsEnd(s): return solution
    For each action a ∈ Actions(s): ...
```

动态规划

- 状态空间图存在环怎么办



$$\text{FutureCost}(s) = \begin{cases} 0 & \text{if IsEnd}(s) \\ \min_{a \in \text{Actions}(s)} [\text{Cost}(s,a) + \text{FutureCost}(\text{Succ}(s,a))] & \text{otherwise} \end{cases}$$

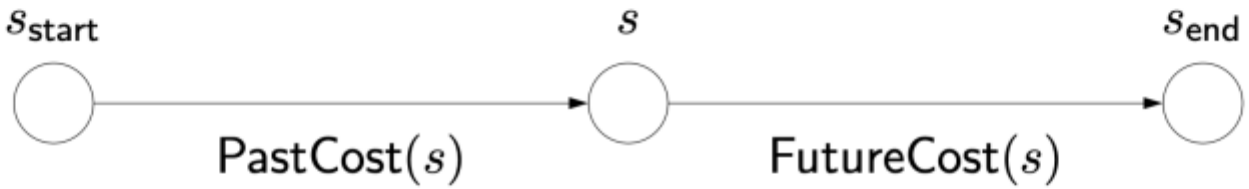
一致代价搜索（Uniform-cost Search,UCS）

- Algorithm: uniform cost search [Dijkstra, 1956]

Add s_{start} to **frontier** (priority queue)
Repeat until frontier is empty:
    Remove s with smallest priority p from frontier
    If IsEnd(s): return solution
    Add s to **explored**
    For each action $a \in$ Actions(s):
        Get successor $s' \leftarrow$ Succ(s, a)
        If s' already in explored: continue
        Update **frontier** with s' and priority $p +$ Cost(s, a)

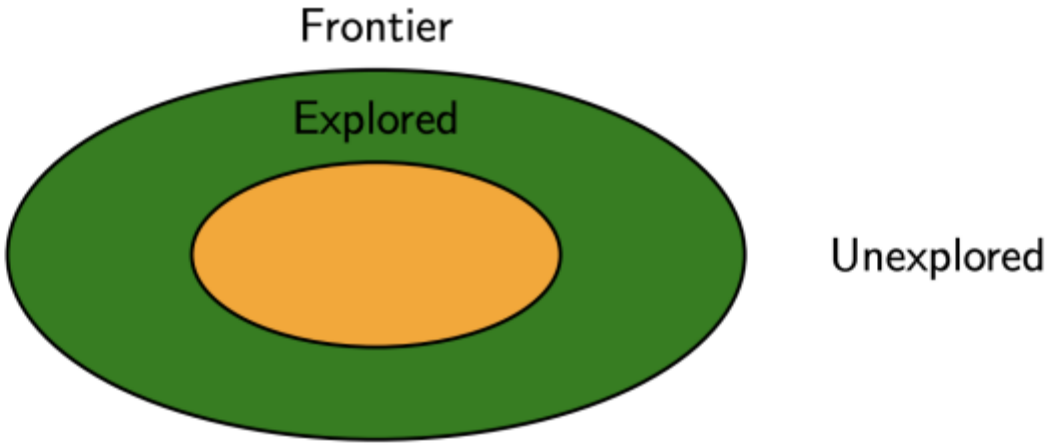
一致代价搜索（Uniform-cost Search,UCS）

- UCS explore states in order of PastCost(s)： the minimum cost to s



- When a state s is popped from the frontier and moved to explored, its priority is PastCost(s)

一致代价搜索（Uniform-cost Search,UCS）



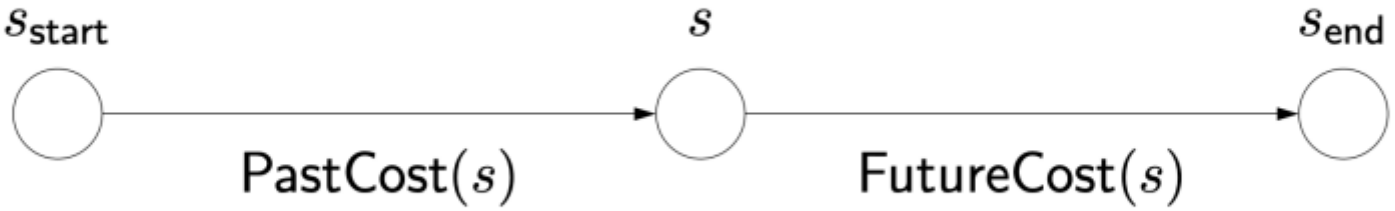
- Explored: states we've found the optimal path to
- Frontier: states we've seen, still figuring out how to get there cheaply
- Unexplored: states we haven't seen

一致代价搜索（Uniform-cost Search,UCS）

算法	是否允许有环	动作代价	时间空间复杂度
动态规划	是	any	O(N)
UCS/Dijkstra	是	非负	O(nlogn)

启发式搜索

- UCS: explore states in order of PastCost(s)



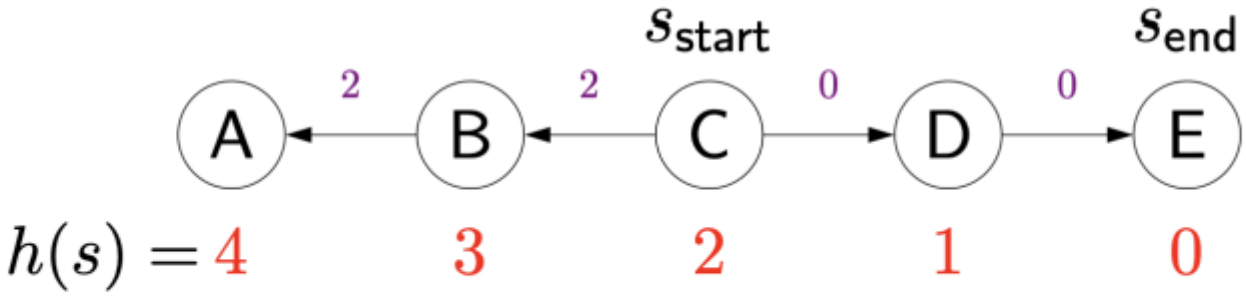
- 理想: explore in order of PastCost(s) + FutureCost(s)
- A*: explore in order of PastCost(s) + h(s)
  - **A heuristic h(s) is any estimate of FutureCost(s).**

A*搜索

- Run uniform cost search with **modified edge costs**:

$$\text{Cost}'(s,a) = \text{Cost}(s,a) + h(\text{Succ}(s,a)) - h(s)$$

- Intuition: add a penalty for how much action a takes us away from the end state



$$\text{Cost}'(C,B) = \text{Cost}(C,B) + h(B) - h(C) = 1 + (3-2) = 2$$

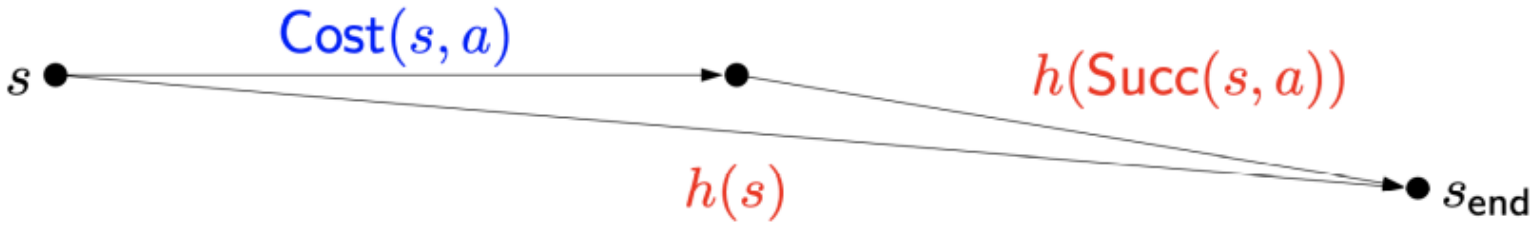
一致性（Consistency）

Definition: consistency

A heuristic h is **consistent** if

- $\text{Cost}'(s,a) = \text{Cost}(s,a) + h(\text{Succ}(s,a)) - h(s) \geq 0$
- $h(s_{\text{end}}) = 0$.

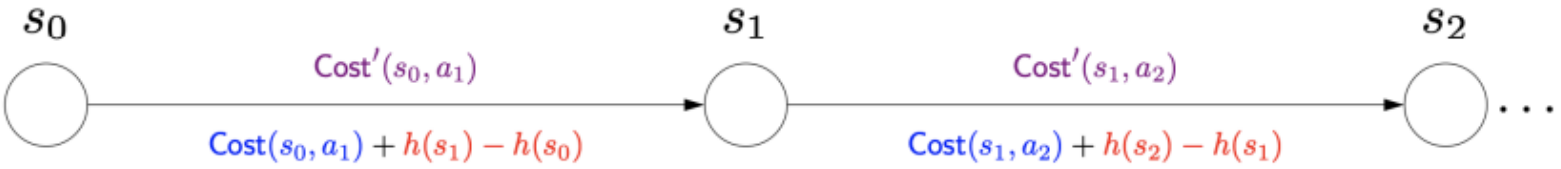
Condition 1: needed for UCS to work (triangle inequality).



Condition 2: $\text{FutureCost}(s_{\text{end}}) = 0$ so match it.

正确性（Correctness）

- **If h is consistent, A* returns the minimum cost path.**



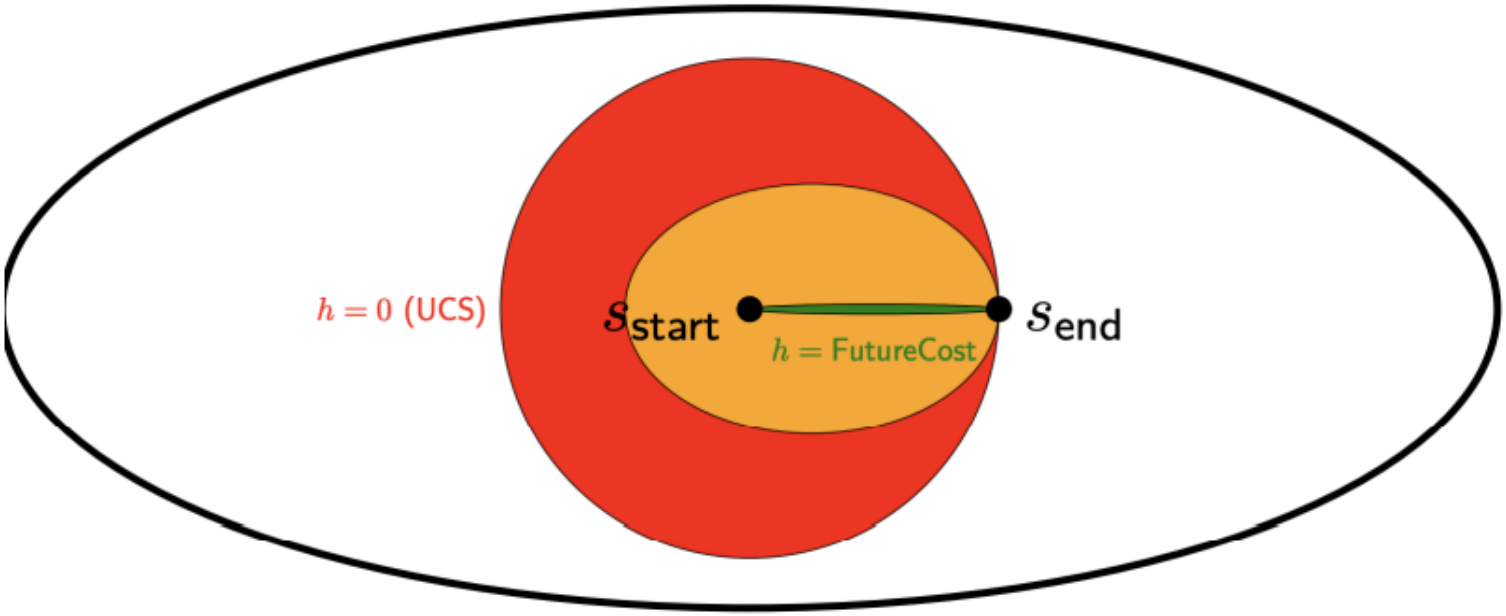
- Key identity:

$$\underbrace{\sum_{i=1}^{L} \text{Cost}'(s_{i-1}, a_i)}_{\text{modified path cost}} = \underbrace{\sum_{i=1}^{L} \text{Cost}(s_{i-1}, a_i)}_{\text{original path cost}} + \underbrace{h(s_L) - h(s_0)}_{\text{constant}}$$

有效性（Efficiency）

- A* explores all states s satisfying :
  PastCost(s) ≤ PastCost(t) – h(s)

- the larger h(s), the better



高级搜索

搜索算法在内存中保留一条或多条路径并且记录哪些是已经探索过的，哪些是还没有探索过的。当找到目标时，到达目标的路径同时也构成了这个问题的一个解。在许多问题中，问题的解与到达目标的路径是无关的。例如，在八皇后问题中，重要的是最终皇后的布局，而不是加入皇后的次序。

这一类问题：

集成电路设计；工厂场地布局作业车间调度；自动程序设计电信网络优化；车辆寻径文件夹管理

高级搜索

局部搜索算法从单独的一个当前状态出发，通常只移动到与之相邻的状态。典型情况下，搜索的路径是不保留的。

- 优点：

（1）它们只用很少的内存

（2）它们通常能在很大状态空间中找到合理的解

许多最优化问题不适合于“标准的”搜索模型。如，自然界提供了一个目标函数——繁殖适应性——达尔文的进化论可以被视为优化的尝试，但是这个问题没有“目标测试”和“路径耗散”。

高级搜索

- 地形图既有“位置”（用状态定义），又有“高度”（由启发式耗散函数或目标函数的值定义）。
- 如果高度对应于耗散，那么目标是找到最低谷——即一个全局最小值；
- 如果高度对应于目标函数，那么目标是找到最高峰——即一个全局最大值。
- 局部搜索算法就象对地形图的探索，如果存在解，那么完备的局部搜索算法总能找到解；最优的局部搜索算法总能找到全局最小值 / 最大值。

爬山法搜索

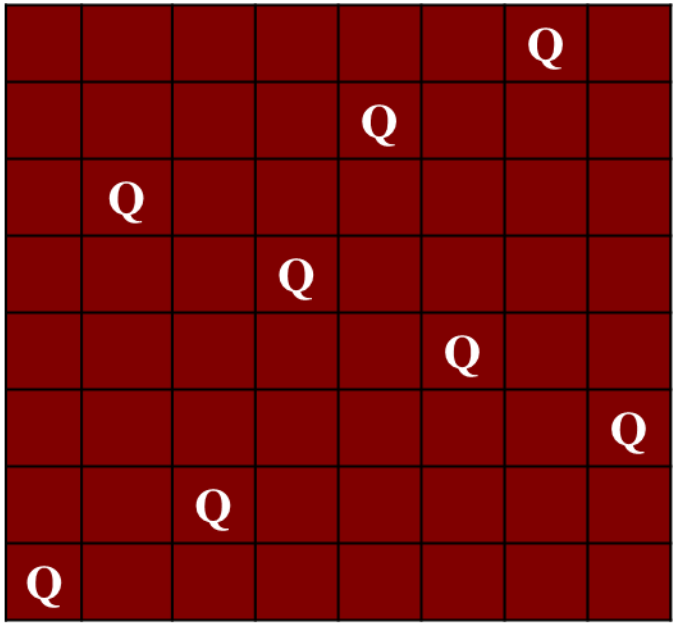
爬山法搜索——局部搜索

登高——一直向值增加的方向持续移动，将会在到达一个“峰顶”时终止，并且在相邻状态中没有比它更高的值。这个算法不维护搜索树，因此当前节点的数据结构只需要记录当前状态和它的目标函数值。

爬山法不会预测与当前状态不直接相邻的那些状态的值。

爬山法搜索

18	12	14	13	13	12	14	14
14	16	13	15	12	14	12	16
14	12	18	13	15	12	14	14
15	14	14	Q	13	16	13	16
Q	14	17	15	Q	14	16	16
17	Q	16	18	15	Q	15	Q
18	14	Q	15	15	14	Q	16
14	14	13	17	12	14	12	18



- 经常遇到局部极大值、山脊、高原等问题。
- 从一个随机生成的八皇后问题的状态开始，最陡上升的爬山法86%的情况下会被卡住，只有14%的问题实例能求解。
- 针对爬山法的不足，有许多变化的形式：随机爬山法、首选爬山法

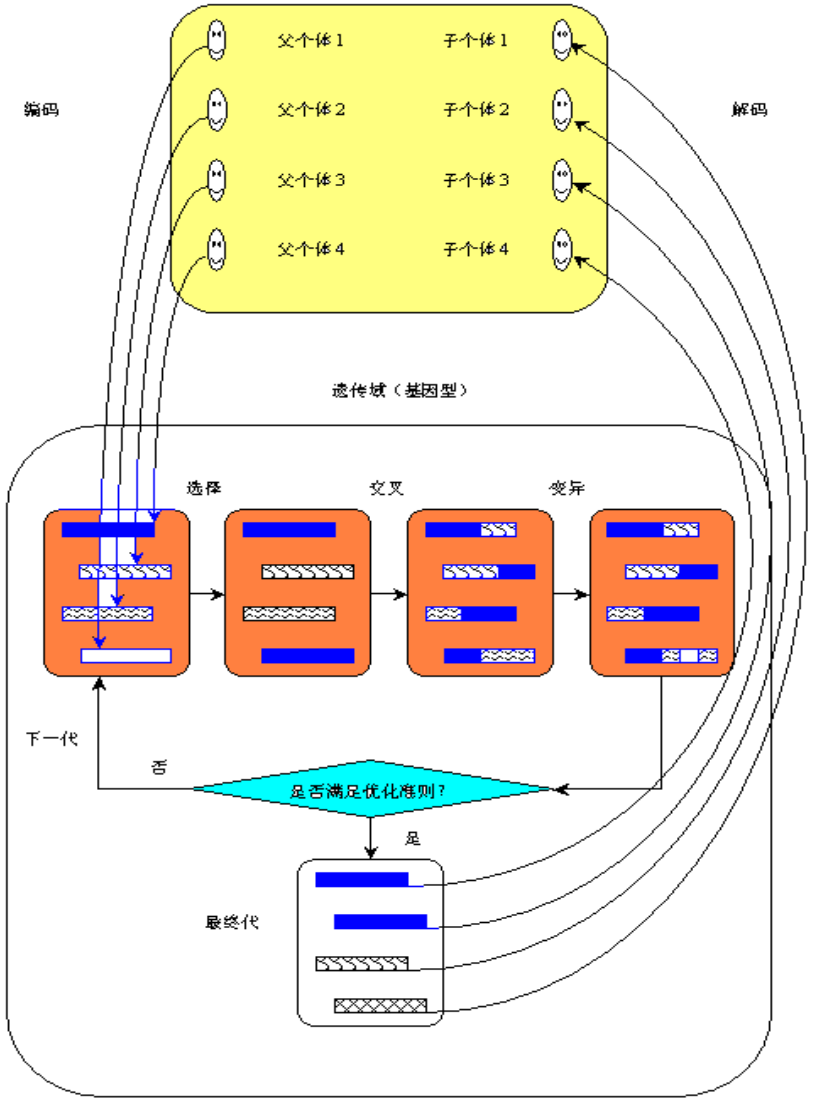
模拟退火搜索

- ＳＡ算法引入一个温度参数，当搜索到较差的邻近解时，利用温度参数和目标函数值之差共同确定一个概率参数，利用此概率参数决定是否接受较差的邻近解，概率参数P的定义为

$$P = \mathrm{e}^{-\frac{\delta}{t}}$$

- 其中，δ为邻近解与当前解的目标函数之差，t为温度参数，该参数对应于固体退火过程中的温度，随着搜索过程的不断推进而不断减小，直至算法达到终止条件。
- 在温度下降足够慢时，算法找到全局最优解的概率接近１。

遗传算法



- 开始时，种群随机地初始化，并计算每个个体的适应度函数，初始代产生了。
- 如果不满足优化准则，开始产生新一代的计算。
- 为了产生下一代，按照适应度选择个体，父代进行基因重组（交叉）而产生子代。
- 所有的子代按一定概率变异。然后子代的适应度又被重新计算，子代被插入到种群中将父代取而代之，构成新的一代。
- 循环，直到满足优化准则。

遗传算法

遗传算法不同于传统的搜索和优化方法。

1. 自组织、自适应和自学习性（智能性）。
2. 遗传算法的本质并行性。遗传算法按并行方式搜索一个种群数目的点，而不是单点。
3. 遗传算法不需要求导或其他辅助知识，而只需要影响搜索方向的目标函数和相应的适应度函数。
4. 遗传算法强调概率转换规则，而不是确定的转换规则。
5. 遗传算法可以更加直接地应用（函数优化、组合优化、生产调度问题、自动控制、机器人智能控制、图像处理和模式识别等　）。
6. 遗传算法对给定问题，可以产生许多的潜在解，最终选择可以由使用者确定（在某些特殊情况下，如多目标优化问题不止一个解存在，有一组pareto最优解。这种遗传算法对于确认可替代解集而言是特别合适的）。

自然启发式优化算法

在新型最优化算法中，有一类算法的基础是群体智能（swarm intelligence, SI）。

SI的设计思想是：群体（蚂蚁和蜜蜂等）中的有机个体，根据区域信息、行为主体（agent）之间的交流及其自身环境做出的决定，是群体智慧（collective intelligence）或社会智慧（social intelligence）的起源。

自然启发式优化算法

初始化种群和迭代计数
评估初始种群并找到最优解
while（未满足停止条件）
　修改局部或全局现有的种群，生成新解
　评估新解
　if 新解更好，then接纳新解，更新种群
　if 新解不好，then依照某些概率性准则接纳新解
　　更新迭代计数
结束
输出过程结果
【典型自然启发算法的伪代码】

遗传算法

算法类型

1. 蚁群优化算法（Ant colony optimization）
2. 蜂群优化算法(Bee colony optimization)
3. 蝙蝠算法(Bat algorithm)
4. 布谷鸟搜索算法（Cuckoo search）
5. 粒子群优化算法（Particle swarm optimization）
6. 萤火虫算法（Firefly algorithm)
7. 花朵授粉算法（Flower pollination algorithm）

其它这类算法还有很多很多，如引力（gravitational）、和声（harmony）、狼群（wolf)搜索算法，以及以生物地理学和免疫系统为基础的最优化算法。

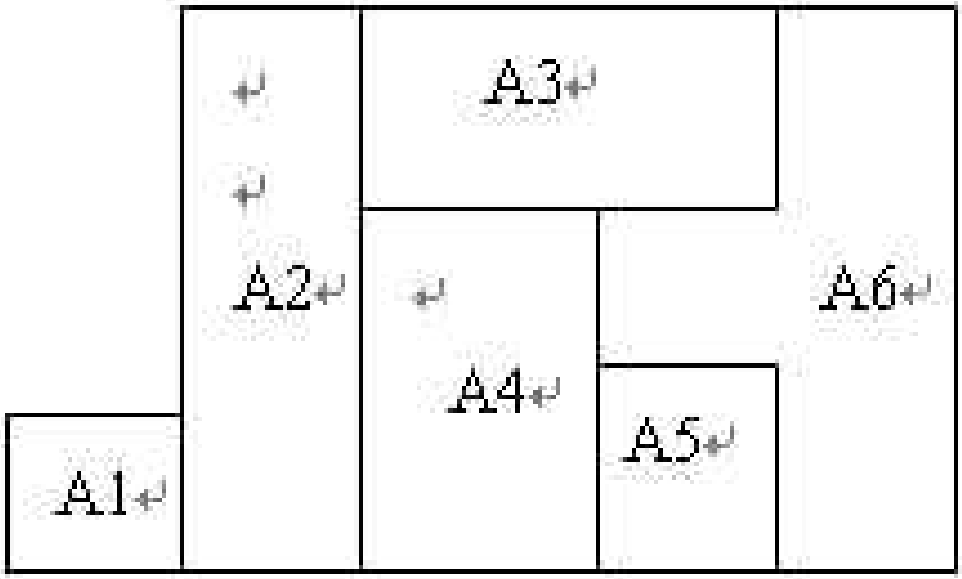
约束满足搜索

- 约束满足问题（CSP）就是为一组变量寻找满足约束的赋值。

- 如，N-皇后问题就是一个约束满足问题。这里的问题就是为N个变量赋值，每个变量的值表示每行上皇后的问题，值域均为[1，N]，约束就是N个皇后谁也“吃”不到谁。

约束满足搜索

例 地图着色问题：对于下图所示的地图，从{红（R），绿（G），黄（Y）}中选择一种颜色赋予图中的国家，使得相邻的国家具有不同的色彩。



假设的地图

约束满足搜索

定义 一个约束满足问题表述为一个三元组（V, D, C），

V———n个变量的集合V={v1, ..., vn}，

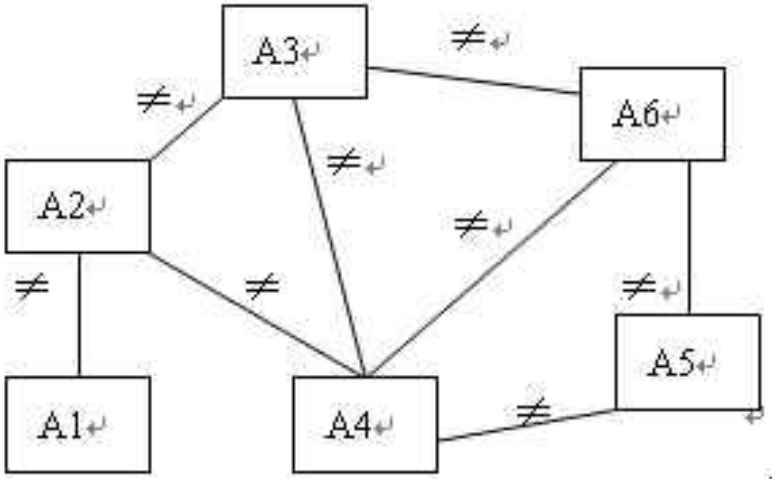
D———变量vi （i=1, 2, ..., n)相应的取值集合 D={D1, ..., Dn}，

C———约束的有限集合，其中每个约束对若干变量同时可取的值做出限制。

问题的解是对所有变量，满足所有约束的赋值。

约束满足搜索

- 用下图来表示相邻关系，其中结点表示国家，连线表示结点之间的邻接关系，约束用≠表示。
- 首先，对于任意一个国家，赋予任意一个色彩。假定首先A1赋予R。用Ai≠R表示Ai不能被赋予R色彩。这样，可以有下面的步骤：



➢ 约束网络 （≠表示连线两端的国家不能使用同一种色彩）

约束满足搜索

Step1：{A1←R, A2≠R}

Step2：{A1←R, (A2←G, A3≠G, A4≠G)}

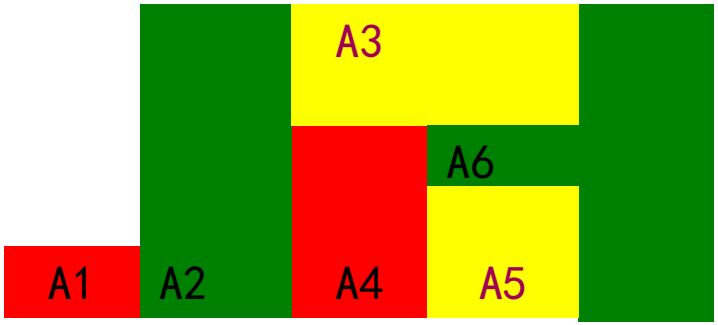
Step3：{A1←R, A2←G, (A3←Y, A4≠Y, A6≠Y), A4≠G}

Step4：{A1←R, A2←G, A3←Y, (A4←R, A5≠R, A6≠R), A6≠Y}

Step5：{A1←R, A2←G, A3←Y, A4←R, (A5←Y, A6≠Y), A6≠R}

Step6：{A1←R, A2←G, A3←Y, A4←R, A5←Y, A6←G}

- 在步骤2中，用A2←G代替步骤1中的A2≠R，并增加一些约束，用括号表示。这里步骤6得到一个可能的解答。



蒙特卡洛树搜索
Monte Carlo Tree Search (MCTS)

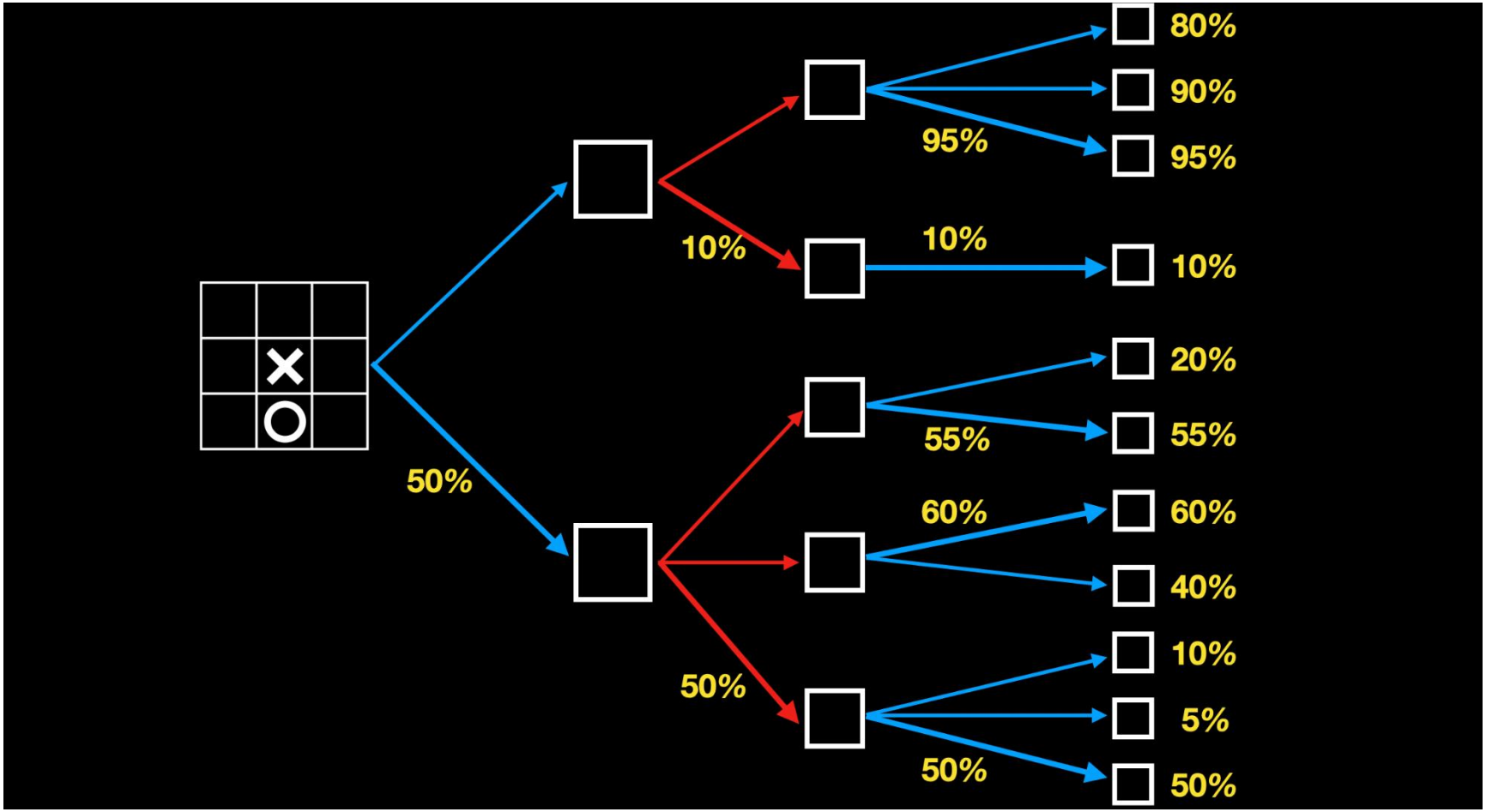
MCTS

- Introduction
  - 围棋 (GO): 状态空间复杂度10^170，机器很难在如此大的空间中学习获胜的策略。很长一段时间以来，学术界都认为围棋是人工智能的“圣杯”。
  - 2016年，DeepMind 设计的 Alpha Go 击败李世石和柯洁。Alpha Go主要包含四个核心技术：
    - ✓ 蒙特卡洛树搜索（**Monte Carlo Tree Search**）
    - ✓ 残差网络 （**Residual Convolutional Neural Networks**）
    - ✓ 强化学习 （**Reinforcement Learning**）
    - ✓ 监督学习 （**Supervised Learning**）

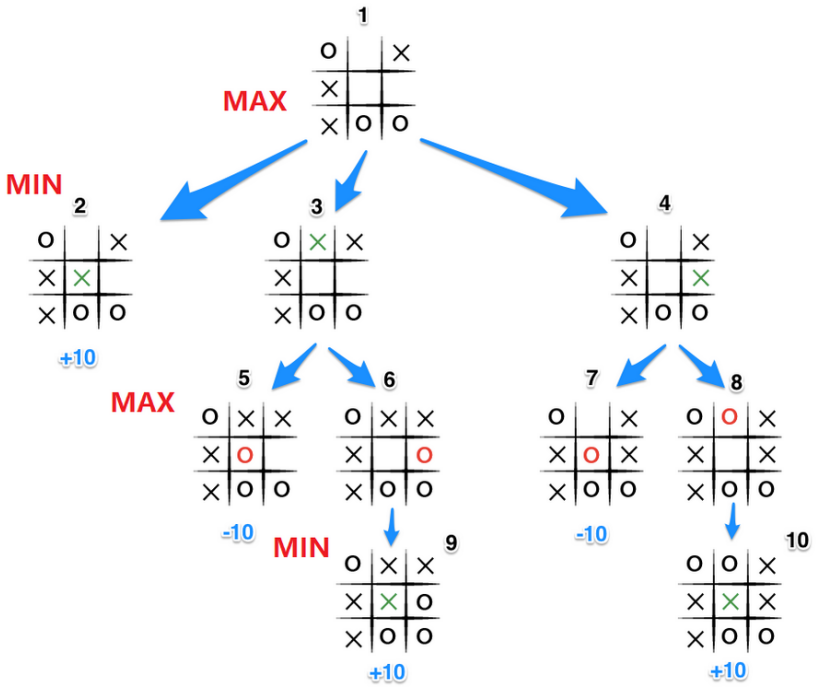
MCTS

- Introduction
  - 一般人下棋时的思维——不是在脑海里面把所有可能列出来，而是根据「棋感」在脑海里大致筛选出了几种「最可能」的走法，然后再想走了这几种走法之后对手「最可能」的走法，然后再想自己接下来「最可能」的走法。这其实就是 MCTS 算法的设计思路
  - 简单来说， MCTS是一种博弈树，其主要目标就是：在给定的状态（棋盘）中选择胜率最高的下一步动作（即最优落子位置）。
  - 以井字棋为例：
    - 树的顶部，即根节点，表征了井字棋的初始状态，即空白棋盘。
    - 任何从一个节点向另一个节点的转换被称为action。
    - 从根节点到一个叶节点的路径表示一个博弈过程。

- Introduction
  - 以井字棋为例，在某状态下的博弈树：



- Introduction
  - 常见的博弈树包括极大极小（minimax）算法和alpha-beta剪枝算法，在国际象棋、井字棋以及五指棋上表现很好。
  - Minimax：
    - 假设两个玩家A和B均采用最优策略，逐层交替执行，且获得收益相互对立。
    - 在博弈树内部节点的值，MAX节点（A）的每一步扩展要是收益最大，MIN节点（B）的扩展要是收益最小



  - 但这些博弈树算法在围棋上表现很差

Introduction

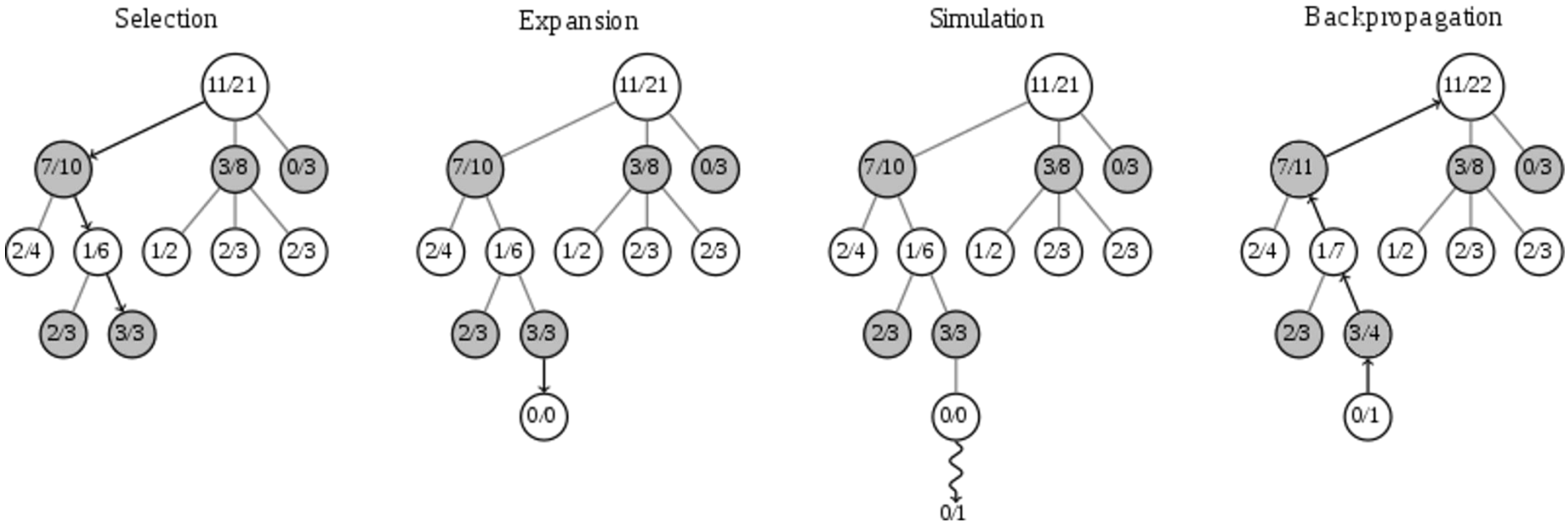
- 棋局评判能力要求更高。棋局的评判一般使用估值函数来评估，国际象棋的棋局局面特征比较明显，最容易想到的是可以给每个棋子和位置设置不同的分值，如果棋子之间的保护关系等特征，对局面的评价就已经很靠谱了。而对于围棋上述方法基本不起任何作用。

- 计算能力要求更高。国际象棋的棋盘大小为 64，围棋的大小为 361。由于棋盘大小的不同，每走一步国际象棋和围棋的计算量的要求是不一样的，围棋明显要求更高。另外一个可以说明计算能力要求不同的指标是搜索空间，在该指标上两者也存在指数级的差异，国际象棋大约是 10^50，而围棋大约是 10^170。 而宇宙中的原子总数总共大约也才10^80，因此围棋的搜索空间绝对算是天文数字， 已经不能用千千万来形容了。

- MCTS -- Basic Concept
  - MCTS 主要包含四个过程
    - 选择（Selection）：从根节点开始，选择一个“最值得探索的子结点”，一般使用UCB选择score最高的节点，直到来到一个叶结点。
    - 扩展（Expansion）：将这个叶节点的一个或多个可行的状态添加为该叶节点的子结点。
    - 模拟（Simulation）：对可行状态节点进行模拟对局，统计胜负结果。
    - 回溯（Backpropagation）：将模拟结果回溯到MCTS节点中。

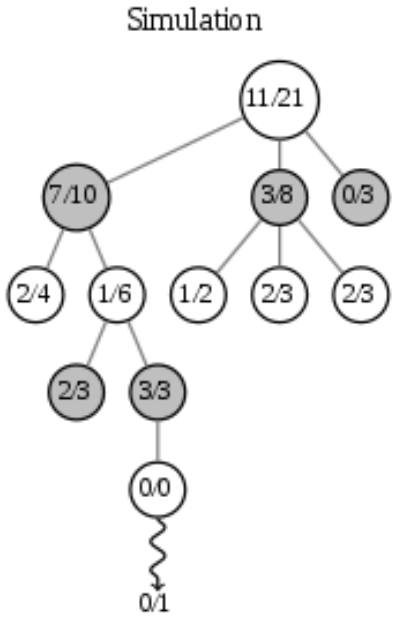
MCTS

- MCTS -- Basic Concept
  - MCTS 实例

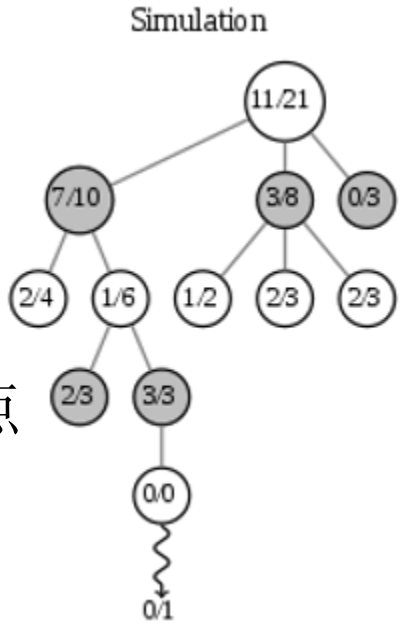


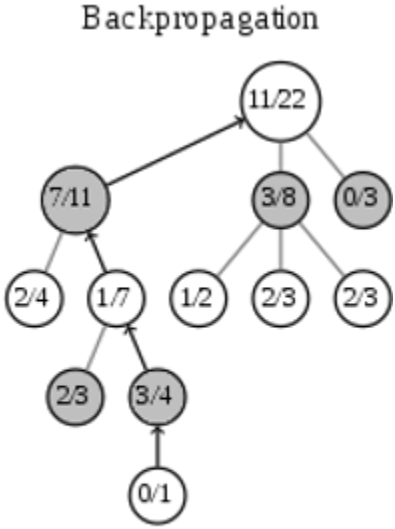
MCTS -- Basic Concept

- 模拟（Simulation）
  - 对可行状态节点进行模拟对局，统计胜负结果
  - 假设每个节点代表不同的博弈局面，其中用两个值表示从当前局面开始模拟到局面结束的博弈结果，分别为模拟次数和赢的次数。
  - 例如右图中的根节点的数值表示：
    - 从根节点出发共模拟21次
    - 赢了11次
    - 记为11/21



- MCTS -- Basic Concept
  - 回溯（ Backpropagation ）
    - 将模拟得到的胜负结果回溯更新父节点的胜负情况。
    - 类似于递归的回溯，从根节点开始，一直到“模拟”的子节点路径上的所有节点都要更新。
    - 如右图：
      - 对“3/3”节点模拟
      - 结果是“0/1”
      - 回溯至路径上所有节点





- MCTS -- Basic Concept
  - 选择（Selection）
    - 把树中所有结点分为三类：
      - 未完全展开：被至少评估过一次，但是子结点没有被全部访问
      - 完全展开：子结点全部被访问
      - 终止节点：在这个节点博弈已经结束
    - 对于节点v存在多个子结点，那怎么选择子节点v_i作为下一个访问节点了？
      - 不能通过节点的胜率判断。主要是因为，在开始阶段，存在一些子最优节点具有一定的胜率，而最优节点还没有被访问到。
      - 提出使用能够平衡探索利用的UCT作为评估函数，选择UCT最大的作为子结点。

- MCTS -- Basic Concept
  - 选择（Selection）
    - 对于节点v，每个子节点v_i的得分通过UCB计算

$$\mathbb{UCB}(v_i, v) = \frac{Q(v_i)}{N(v_i)} + c\sqrt{\frac{\log(N(v))}{N(v_i)}}$$

其中，$\mathrm{Q}(v_i)$是子节点赢的次数，$\mathrm{N}(v_i)$是子节点的模拟次数，C是一个常数，$\mathrm{N}(v)$ 是节点v 的模拟次数。

- UCB能够解决探索利用问题。$\frac{\mathrm{Q}(v_i)}{N(v_i)}$表示子结点的胜率，用于“利用”。$\frac{\log(N(v))}{N(v_i)}$用于“探索”，当$v_i$的访问次数增加，则UCB分数会减小。C是一个温度系数，用于控制探索程度。

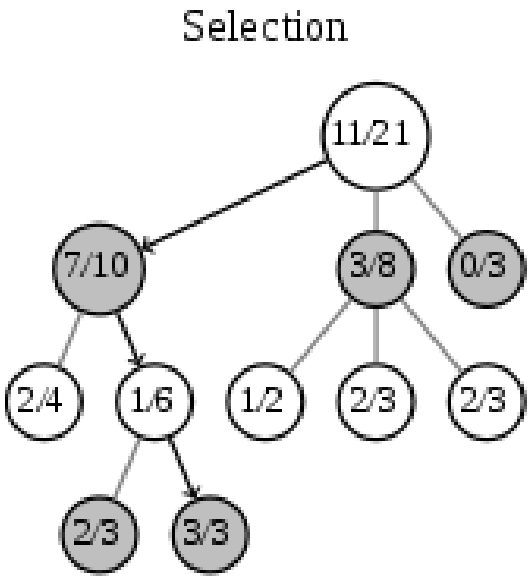
- MCTS -- Basic Concept
  - 选择（Selection）
    - 整个过程（从根节点出发）：
      1. 判断当前节点是否是终止局面，如果是则结束当前树搜索。
      2. 如果不是，则判断当前节点是否存在从未访问子节点。
      3. 如果存在从未访问过的子节点，则访问该子节点，并执行“扩展（Expansion）”。
      4. 如果所有节点都曾被访问过，利用UCT计算所有子结点得分，选择UCB最大的子结点。

- MCTS -- Basic Concept
  - 选择（Selection）
    - 例如右图根节点(C=10)：

$$socre\left(7/10,v\right)=\frac{7}{10}+c\sqrt{\frac{\log(21)}{10}}\approx 6.2$$

$$socre\left(5/8,v\right)=\frac{5}{8}+c\sqrt{\frac{\log(21)}{8}}\approx 6.8$$

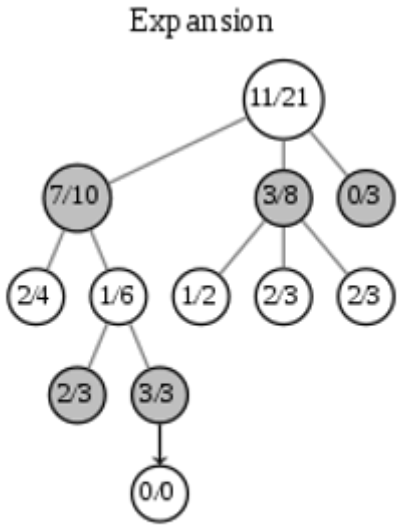
$$socre\left(0/3,v\right)=\frac{0}{3}+c\sqrt{\frac{\log(21)}{3}}\approx 10$$



则选择“0/3”这个节点

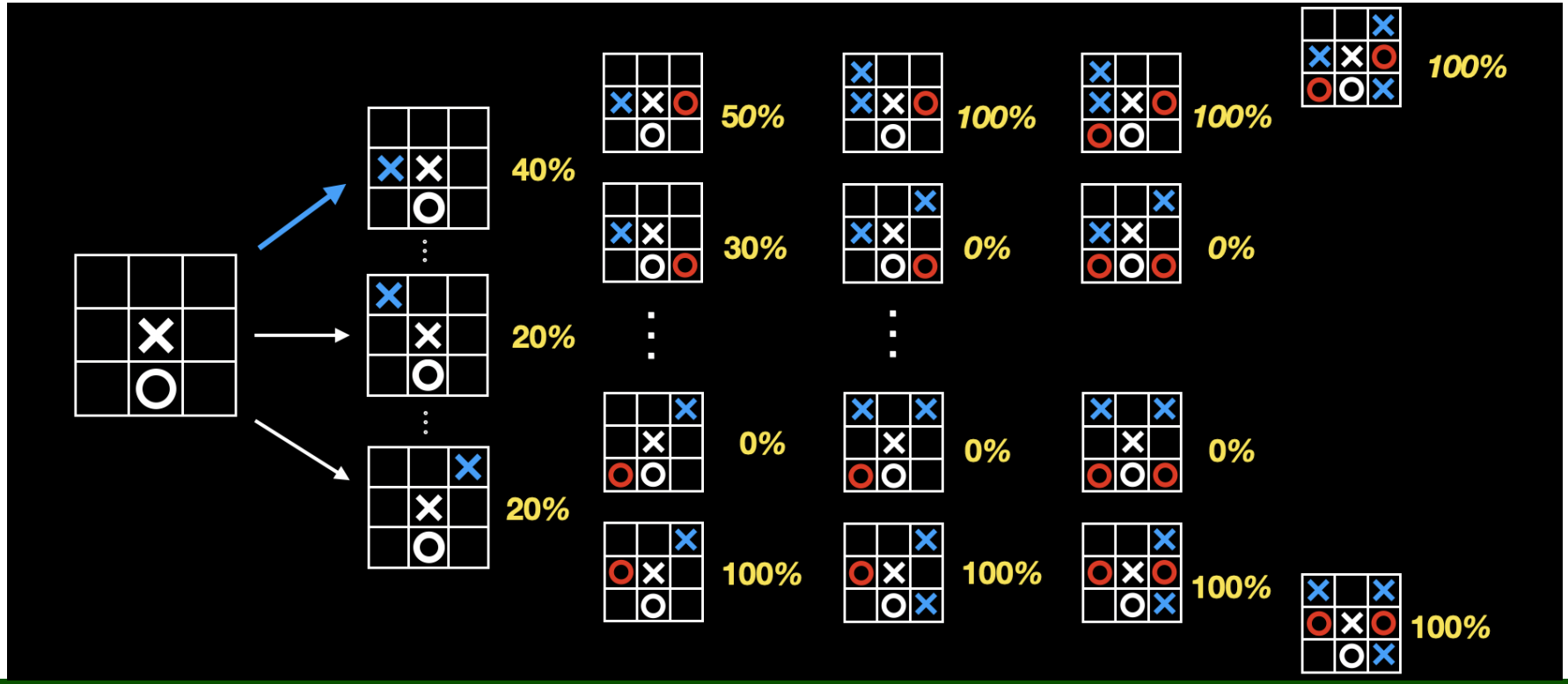
- 例如，“3/3”为叶节点，如果这个节点是终止节点，则当前树搜索结束。否则，选择一个子结点执行“扩展”。

- MCTS -- Basic Concept
  - 扩展（Expansion）
    - 将这个叶节点的一个或多个可行的状态添加为该叶节点的子结点
    - 在选择阶段结束后，如果最后选择的是未访问的节点，则在这个节点执行“扩展”。即在树中新建一个节点，并加上胜负信息“0/0”，然后执行“模拟”阶段。
    - 例如右图
      - “3/3”节点的所有子结点都没扩展
      - 任选一个未扩展子结点
      - 为子结点加上胜负信息“0/0”
      - 执行“模拟”阶段

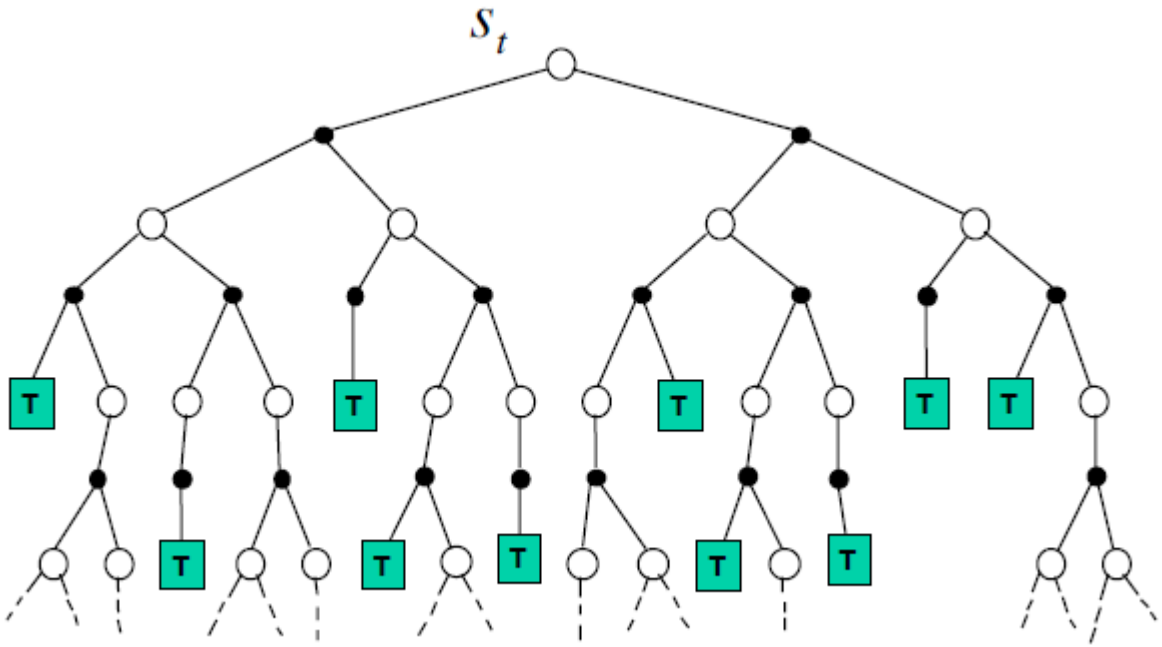


MCTS

- MCTS -- Basic Concept
  - 一个程序执行多少次MCTS才是最优的？
    - 对于简单环境，例如井字棋，可以让蒙特卡洛树把所有棋面都至少模拟一遍（模拟次数越多，树中的胜负情况越准确），然后选择最优的下一个动作。



- MCTS -- Basic Concept
  - 一个程序执行多少次MCTS才是最优的？
    - 对于复杂环境，例如围棋，让蒙特卡洛树在能容忍范围内尽可能多的模拟，然后选择当前树中最优的下一个动作。



MCTS

- MCTS -- AlphaGo
  - 相比于basic MCTS，AlphaGo做了如下改进：
    - 在选择阶段，如果子节点没被访问过，Q和N初始阶段都是0，因此只能随机选择子节点，而AlphaGo根据强化学习的policy选择子节点。
    - 在模拟阶段，AlphaGo学习一个状态转移函数，能够预测下一时刻的状态，从而可以执行rollout（在AlphaGo中将一次模拟过程称为一次rollout）过程。
    - 通过利用强化学习的Q函数，MCTS使用一个新的评估函数UCB评估子结点的score：

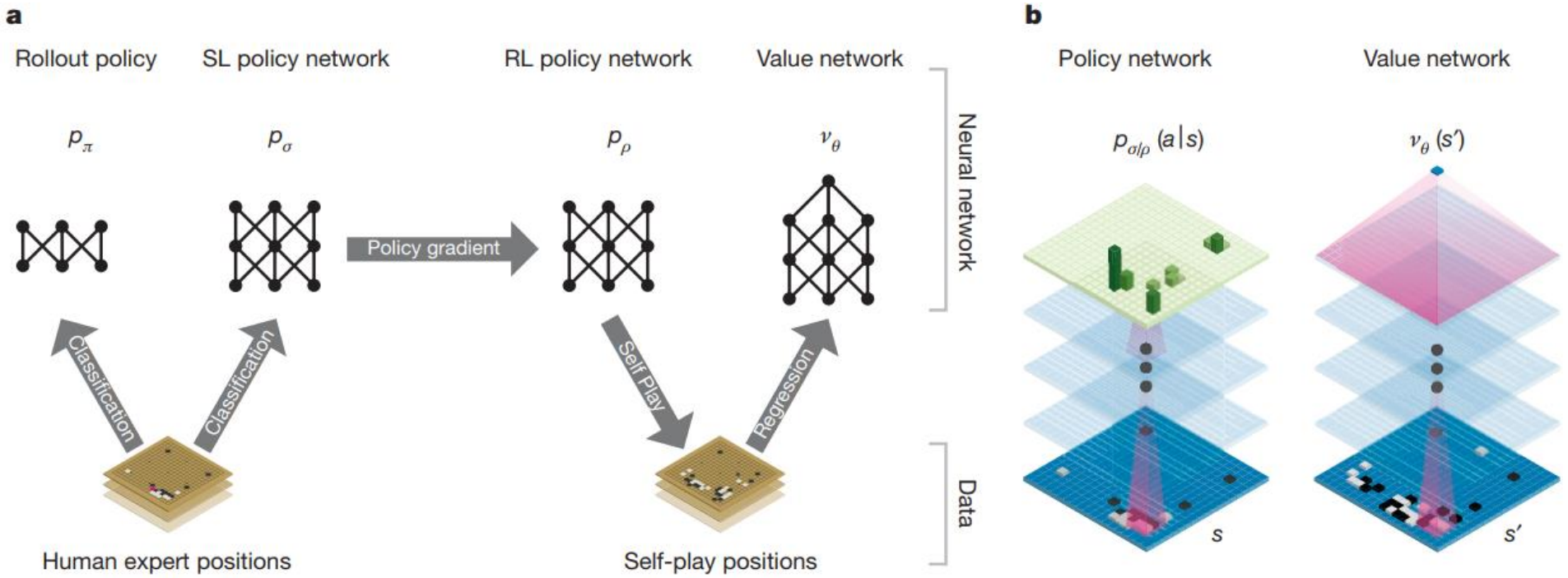
$$\mathrm{UCB} = \mathrm{Q}(s,a) + U(s,a) \quad U(s,a) \sim c * P(s,a) * \frac{\sqrt{\Sigma_b N(s,b)}}{(1+N(s,a))}$$

其中P(s,a)是转移概率，当前节点转移到每个子结点的概率

- MCTS -- AlphaGo
  - AlphaGo的整体过程：
    - 监督学习：利用人类专家的经验学习两个网络，rollout policy 和 SL policy
      - SL policy P_σ，即通过当前局面s预测人类专家的动作
      - Rollout policy P_π，即预测强化学习的状态转移函数，即在局面s，执行动作a后，转移到一个新局面s′
    - 强化学习分为两部分：策略学习和值函数学习
      - 策略学习 P_ρ：监督学习的策略网络仅能模仿给定人类经验的下棋策略，不具有较好的泛化性，将监督学习的策略网络用强化学习通过self-play的方式继续学习。（强化学习和监督学习共享策略网络）
      - 值函数学习 V_θ：在策略学习过程中同时也用强化学习学习一个值函数。

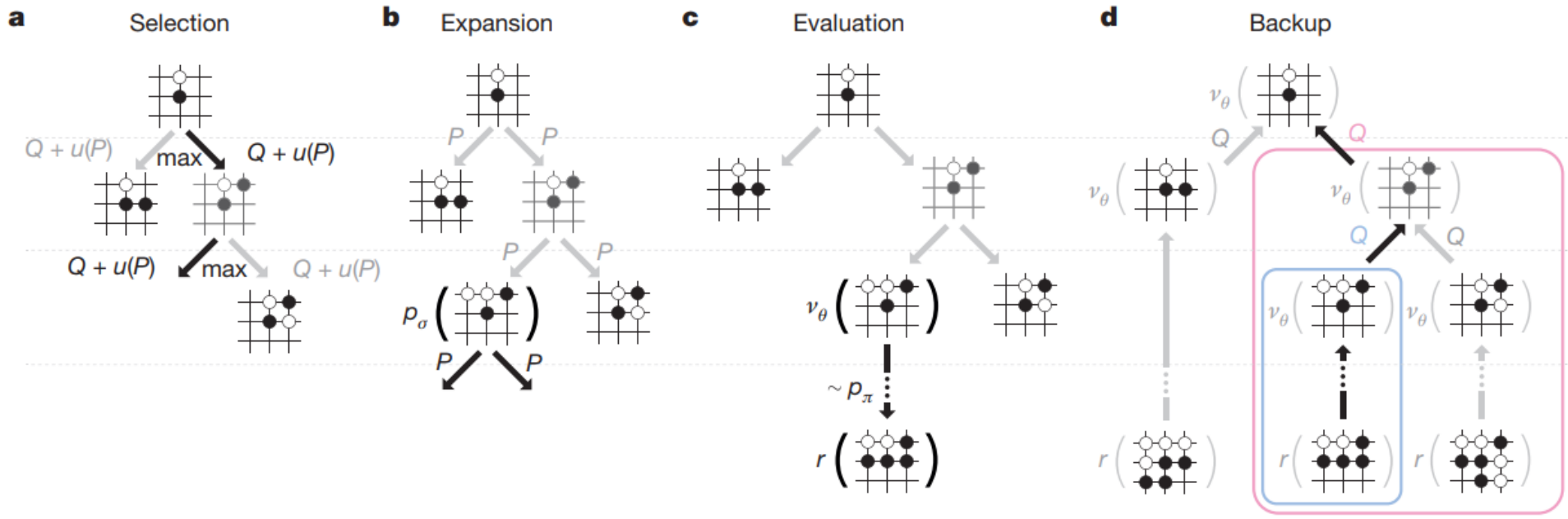
MCTS

- MCTS -- AlphaGo
  - AlphaGo的整体过程：



MCTS

- MCTS -- AlphaGo
  - AlphaGo的整体过程：
    - 当强化学习模型训练后，并不用强化学习直接决策，而是用强化学习的策略网络和值网络执行MCTS，最后将MCTS的结果作为最终决策。



Thank You ！