

启发式搜索

搜索算法：贪婪最佳优先搜索

A*算法

对抗搜索

最小最大搜索(Minimax Search):

例子：井字棋

alpha-beta 剪枝 (alpha-beta pruning)

什么时候可以剪枝：

蒙特卡洛树搜索

单一状态蒙特卡洛规划：多臂赌博机(multi-armed bandits)

西瓜书上方法

ϵ -贪心

Softmax 算法

上限置信区间策略(Upper Confidence Bound Strategies, UCB)

蒙特卡洛树搜索(Monte-Carlo Tree Search)

例子：围棋蒙特卡洛树搜索

选择：

扩展：

模拟，反向传播：

伪代码

逻辑与推理

命题逻辑

推理规则

谓词逻辑

知识图谱推理

FOIL (First Order Inductive Learner)

强化学习

马尔可夫奖励过程 (Markov Reward Process)

马尔可夫决策过程 (Markov Decision Process)：引入动作

贝尔曼方程 (Bellman Equation)

策略优化与策略评估

策略优化

通过迭代计算贝尔曼方程进行策略评估

Q-Learning

Meta Learning

定义

Learning Algorithm Set

Learning Algorithm 的评估

MAML: Model-Agnostic Meta-Learning for Fast Adaptation of Deep Network

Few-Shot Learning

Embedding Learning 方法

Siamese Neural Network 孪生神经网络

Match Network 匹配网络

Prototypical Networks 原型网络/Relation Networks 关系网络

北大：人工智能原理

Problem Solving Agents

Related Terms

Example Problems

Vacuum-cleaner world

8-puzzle

8-queens

(书P66) 玩具问题

(书P70) 通过一些约束处理冗余路径

Searching for Solutions

无信息搜索 (Uninformed Search Strategies)

BFS

一致代价搜索 (Uniform cost Search)

(书P75) 与BFS的不同

DFS

深度受限搜索 (Depth limited Search)

迭代加深搜索 (Iterative Deepening Search)

双向搜索 (Bidirectional Search)

有信息/启发式搜索 (Informed Search Strategies)
最佳优先搜索 (Best-first Search)
 贪婪搜索
 A* 搜索
 迭代加深 A* 搜索 (Iterative Deepening A* Search)
 启发式函数 (Heuristic Functions)
 8-puzzle 的启发式函数
超越经典搜索
 爬山法 (Hill Climbing)
 例子: n 皇后问题
 爬山法弱点
 局部束搜索 (Local Beam Search)
 例子: 旅行商问题 TSP
 禁忌搜索 (Tabu Search)
 例子: 最小生成树问题
StanFord CS221: 人工智能原理与技术
P5 5. Lecture 5 - Search 1 - Dynamic Program, Uniform Cost Search
state-based model
树搜索: 以农夫过河为例
例子: 运输问题
 DP 加路径约束
Uniform Cost Search
P6 6.Lecture 6 Search 2 - A
UCS最优的证明
Learning: 再次以运输问题为例
A*
 理解启发函数, 为什么能更好找到目标?
 启发函数特性: Consistency
 启发函数特性: Admissibility
 如何寻找到一个好的启发函数?

- 启发式搜索:

- 给定搜索目标, 设计启发函数, 保证搜索目标的最优化求解.
- 对抗搜索: 搜索解决方案的过程中有一个对手, 阻止解决方案获得最大收益.
- 蒙特卡洛树搜索: 采样为基础.

问题求解是从海量信息源里面, 依靠约束条件和额外信息.

- 人工智能中的搜索 以寻找最短路径为例:

- 形式化的描述: 状态, 动作, 状态转移, 路径, 目标测试.
- 状态: 从原问题转化出的问题描述. 例如, 在最短路径问题中, 城市可作为状态. 将原问题对应的状态称为初始状态.
 原问题转换的描述, 比如最短路这里的状态就是 城市. 向目标城市(目标状态)搜索最短路径.
- 动作: 从当前时刻所处状态转移到下一时刻所处状态所进行操作. 一般而言这些操作都是离散的.
- 状态转移: 对某一时刻对应状态进行某一种操作后, 所能够到达状态.
 动作, 状态转移就是 从一个城市转移到另一个城市.
- 路径: 一系列状态的集合. 一个状态序列. 该状态序列被一系列操作所连接.
- 目标测试: 评估当前状态是否为所求解的目标状态.

启发式搜索

(有信息搜索), 有额外的信息给出, 比如最短路中就有额外的地图.

- 辅助信息：所求解问题之外、与所求解问题相关的特定信息或知识。比如最短路问题中就是：

Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

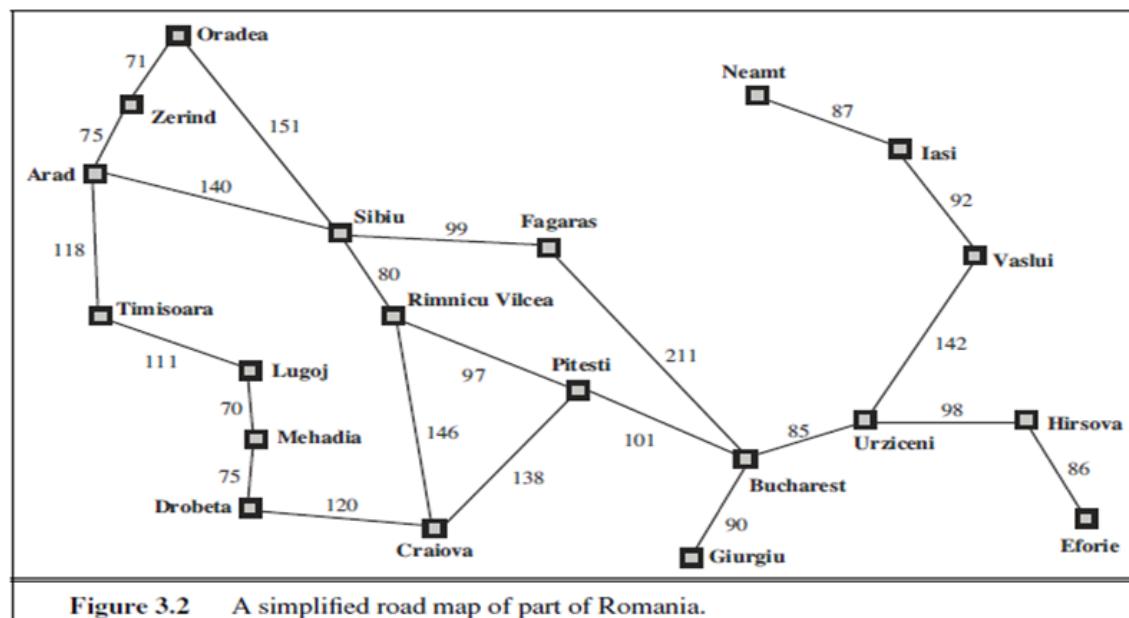
Figure 3.22 Values of h_{SLD} —straight-line distances to Bucharest.

辅助信息：任意一个城市与Bucharest之间的直线距离

- 评价函数：根据评价函数来选择下一个结点，最短路问题中即评价选择下一个结点的优劣。
- 启发函数：计算从节点 n 到目标节点之间所形成路径的最小代价值。这里将两点之间的直线距离作为启发函数。

例子：

搜索算法：贪婪最佳优先搜索

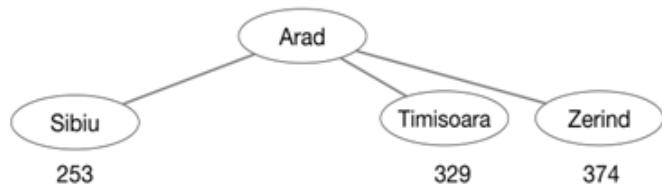


评价函数 $f(n) = \text{启发函数 } h(n)$, 选择与源结点有最短路径的结点。

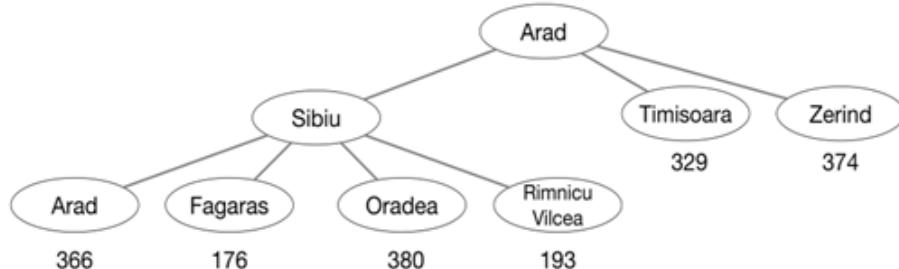
(a) 初始状态



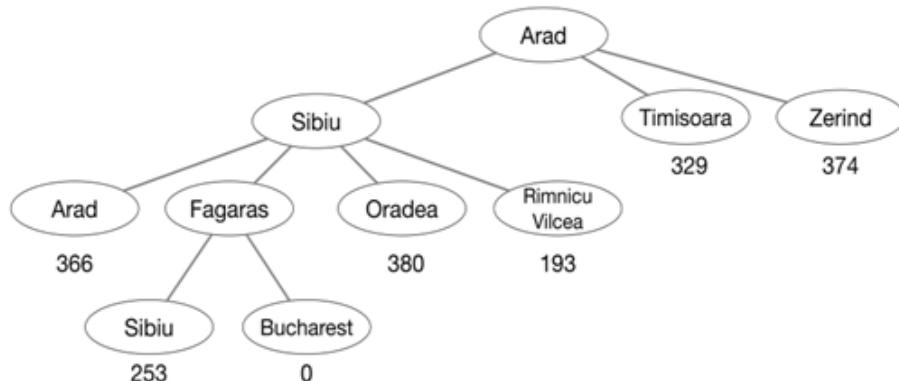
(b) 扩展 Arad 之后



(c) 扩展 Sibiu 之后



(d) 扩展 Fagaras 之后



1. 第一步，扩展 Arad，评价函数告诉我们 Sibiu 是最优的。
2. 接下来扩展上一层选择的结点。

其实上面的过程和 Dijkstra 算法的原理有一点不一样，这里的贪婪是只有上一层选择的那个结点的拓展(fringe)，但Dijkstra是所有已经选择的结点的拓展(fringe)。所以：

- 贪婪最佳优先搜索不是最优的。经过Sibiu到Fagaras到Bucharest的路径比经过Rimnicu Vilcea到Pitesti到Bucharest的路径要长32公里。
- 启发函数代价最小化这一目标会对错误的起点比较敏感。考虑从Iasi到Fagaras的问题，由启发式建议须先扩展Neamt，因为其离Fagaras最近，但是这是一条存在死循环路径。(这就是有别于 Dijkstra 的地方)

不足之处：

- 贪婪最佳优先搜索也是不完备的。所谓不完备即它可能沿着一条无限的路径走下去而不回来做其他的选择尝试，因此无法找到最佳路径这一答案。
- 在最坏的情况下，贪婪最佳优先搜索的时间复杂度和空间复杂度都是 $O(b^m)$ ，其中 b 是节点的分支因子数目、 m 是搜索空间的最大深度。

原因在启发函数不够好。因此提出：

A*算法

定义新的 评价函数： $f(n) = g(n) + h(n)$

$$\begin{array}{lcl} \text{评估函数} & = & \text{当前最小开销代价} + \text{后续最小开销代价} \\ f(n) & = & g(n) + h(n) \end{array}$$

- $g(n)$ 表示从起始节点到节点n的开销代价值。

- $h(n)$ 表示从节点n到目标节点路径中所估算的最小开销代价值。

搜索算法：A*算法

$$f(n) = g(n) + h(n)$$

评估函数 当前最小开销代价 后续最小开销代价

为了保证A*算法是最优 (optimal)，需要启发函数 $h(n)$ 是可容的(admissible heuristic)和一致的(consistency，或者也称单调性，即 monotonicity)。

最优	不存在另外一个解法能得到比A*算法所求得解法具有更小开销代价。
可容(admissible)	专门针对启发函数而言，即启发函数不会过高估计(over-estimate)从节点n到目标结点之间的实际开销代价（即小于等于实际开销）。如可将两点之间的直线距离作为启发函数，从而保证其可容。
一致性 (单调性)	假设节点n的后续节点是n'，则从n到目标节点之间的开销代价一定小于从n到n'的开销再加上从n'到目标节点之间的开销，即 $h(n) \leq c(n, a, n') + h(n')$ 。这里n'是n经过行动a所抵达的后续节点， $c(n, a, n')$ 指n'和n之间的开销代价。

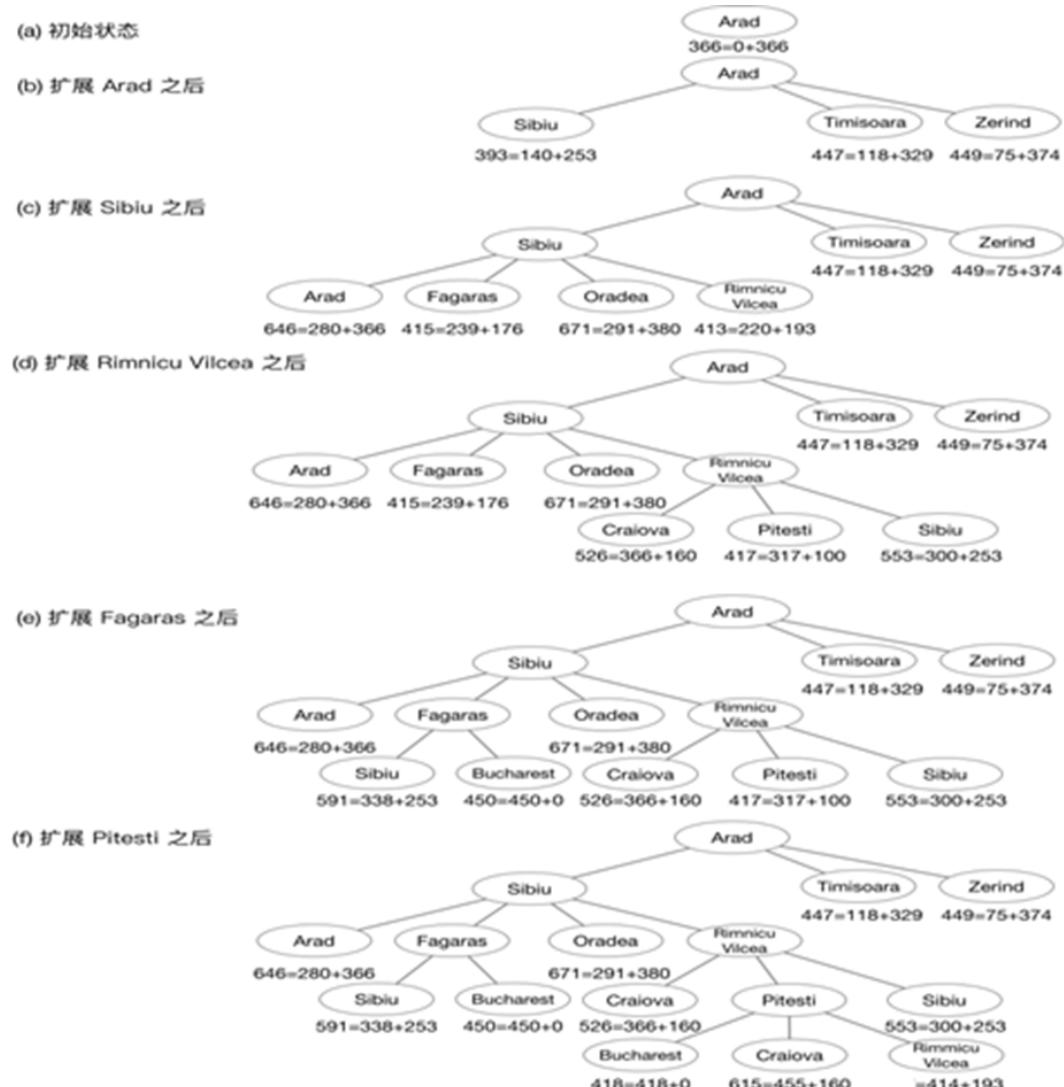
In computer science, a heuristic function is said to be admissible if it is no more than the lowest-cost path to the goal. In other words, a heuristic is admissible if it never overestimates the cost of reaching the goal.

An admissible heuristic is also known as an optimistic heuristic.

注意上面对 可容性和一致性在最短路问题上的描述。

可容：不会过高估计结点 n 到目标结点之间的实际开销代价。

一致：后续最小开销代价是单调的。



Arad → Sibiu: 评估函数显示 $140 + 253$, 其中140是实际距离, 253是直线距离.

在评估函数计算选择下, Sibiu被选择.

此时A*算法寻找到了最短路.

总结:

A*算法保持最优的条件: 启发函数具有可容性(**admissible**)和一致性(**consistency**)。

- 将直线距离作为启发函数 $h(n)$, 则启发函数一定是可容的, 因为其不会高估开销代价。
- $g(n)$ 是从起始节点到节点 n 的实际代价开销, 且 $f(n) = g(n) + h(n)$, 因此 $f(n)$ 不会高估经过节点 n 路径的实际开销。
- $h(n) \leq c(n, a, n') + h(n')$ 构成了三角不等式。这里节点 n 、节点 n' 和目标结点 G_n 之间组成了一个三角形。如果存在一条经过节点 n' , 从节点 n 到目标结点 G_n 的路径, 其代价开销小于 $h(n)$, 则破坏了 $h(n)$ 是从节点 n 到目标结点 G_n 所形成的具有最小开销代价的路径这一定义。

$$\begin{array}{ccc} f(n) & = & g(n) + h(n) \\ \text{评估函数} & \text{当前最小开销代价} & \text{后续最小开销代价} \end{array}$$

- Tree-search的A*算法中, 如果启发函数 $h(n)$ 是可容的, 则A*算法是最优的和完备的; 在Graph-search的A*算法中, 如果启发函数 $h(n)$ 是一致的, A*算法是最优的。
- 如果函数满足一致性条件, 则一定满足可容条件; 反之不然。
- 直线最短距离函数既是可容的, 也是一致的。

如果 $h(n)$ 是一致的(单调的), 那么评估函数 $f(n)$ 一定是非递减的(**non-decreasing**), 简单证明如下: (抓住 当前代价 $g(n') = g(n) + c(n, a, n')$)

证明: 假设节点 n' 是节点 n 的后续节点, 则有 $g(n') = g(n) + c(n, a, n')$ (a 是从节点 n 到节点 n' 的一个行动), 存在:

$$f(n') = g(n') + h(n') = g(n) + c(n, a, n') + h(n') \geq g(n) + h(n) = f(n)$$

A*算法每一步选择的结点一定是在后续最优路径上的结点, 证明(反证法)如下:

- 如果A*算法将节点 n 选择作为具有最小代价开销的路径中一个节点, 则 n 一定是最优路径中的一个节点。即最先被选中扩展的节点在最优路径中。

证明: 反证法。假设上述结论不成立。则存在一个未被访问的节点 n' 位于从起始节点到节点 n 的最佳路径上。根据非递减性质, 存在 $f(n) \geq f(n')$, 则 n' 应该已经被访问过了(expanded)。因此, 无论什么时候, 一旦一个节点被访问到, 它一定位于从起始节点到它自己之间的最佳路径上。

不是很理解上面的证明, 为什么比 $f(n)$ 小的一定会被访问到?

对抗搜索

智能体之间通过竞争实现相反的利益.

当前仅讨论 确定的、全局可观察的、竞争对手轮流行动、零和游戏 (zero-sum) 下的对抗搜索.

最小最大搜索(Minimax Search):

两人对决游戏 (MAX and MIN, MAX先走) 可如下形式化描述, 从而将其转换为对抗搜索问题

初始状态 S_0	游戏所处于的初始状态
玩家 $PLAYER(s)$	在当前状态 s 下, 该由哪个玩家采取行动
行动 $ACTIONS(s)$	在当前状态 s 下所采取的可能移动
状态转移模型 $RESULT(s, a)$	在当前状态 s 下采取行动 a 后得到的结果
终局状态检测 $TERMINAL - TEST(s)$	检测游戏在状态 s 是否结束
终局得分 $UTILITY(s, p)$	在终局状态 s 时, 玩家 p 的得分。

例子: 井字棋

minmax算法:

- 给定游戏搜索树(所有的可能情况 解空间).
- 通过每个节点的minmax值来决定最优策略.
- MAX(这个玩家)希望最大化minmax值, MIN希望最小化.

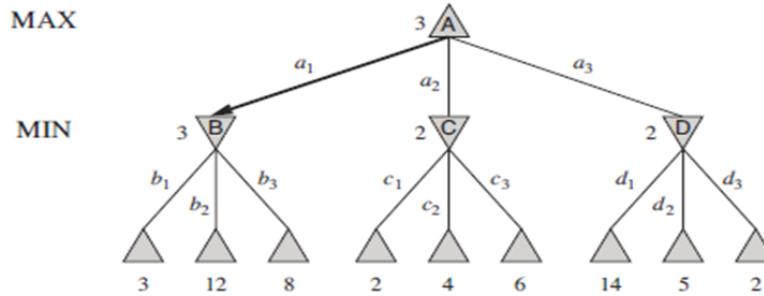


Figure 5.2 A two-ply game tree. The \triangle nodes are “MAX nodes,” in which it is MAX’s turn to move, and the ∇ nodes are “MIN nodes.” The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX’s best move at the root is a_1 , because it leads to the state with the highest minimax value, and MIN’s best reply is b_1 , because it leads to the state with the lowest minimax value.

上图, MAX在 A 这个结点决定接下来走BCD哪一个, 如果到B则MIN会选择3这个结点, 同理MIN到CD会选择2的结点, $3 > 2$, 所以MAX在A一定会选择走B这个结点.

最后MAX选择 a_1 .

- 上述 MAX 选择 $\max(\min(\text{后续}))$ 的形式化表述:

$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s) & \text{if } \text{TERMINAL-TEST}(s) \\ \max_{a \in \text{Actions}(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{if } \text{PLAYER}(s) = \text{MAX} \\ \min_{a \in \text{Actions}(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{if } \text{PLAYER}(s) = \text{MIN} \end{cases}$$

这里的假设是对手一定是选择对自己有利的方法.

```

function MINIMAX-DECISION(state) returns an action
  return  $\arg \max_{a \in \text{ACTIONS}(s)} \text{MIN-VALUE}(\text{RESULT}(s, a))$ 

function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
   $v \leftarrow -\infty$ 
  for each a in ACTIONS(state) do
     $v \leftarrow \text{MAX}(v, \text{MIN-VALUE}(\text{RESULT}(s, a)))$ 
  return v

function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
   $v \leftarrow \infty$ 
  for each a in ACTIONS(state) do
     $v \leftarrow \text{MIN}(v, \text{MAX-VALUE}(\text{RESULT}(s, a)))$ 
  return v

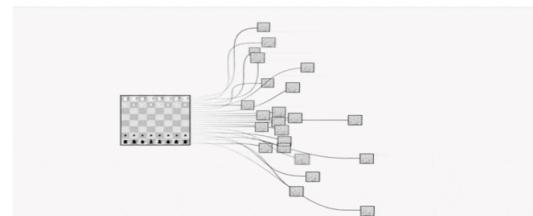
```

Figure 5.3 An algorithm for calculating minimax decisions. It returns the action corresponding to the best possible move, that is, the move that leads to the outcome with the best utility, under the assumption that the opponent plays to minimize utility. The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state. The notation $\arg \max_{a \in S} f(a)$ computes the element *a* of set *S* that has the maximum value of *f(a)*.

对抗搜索：minimax算法

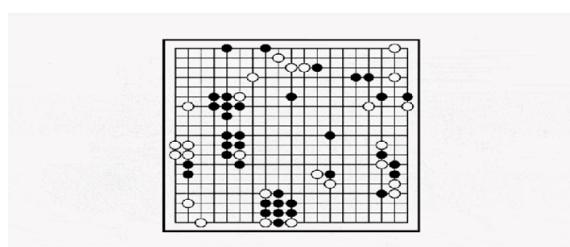
优点:

- 算法是一种简单有效的对抗搜索手段
- 在对手也“尽力而为”前提下，算法可返回最优结果



缺点:

- 如果搜索树极大，则无法在有效时间内返回结果



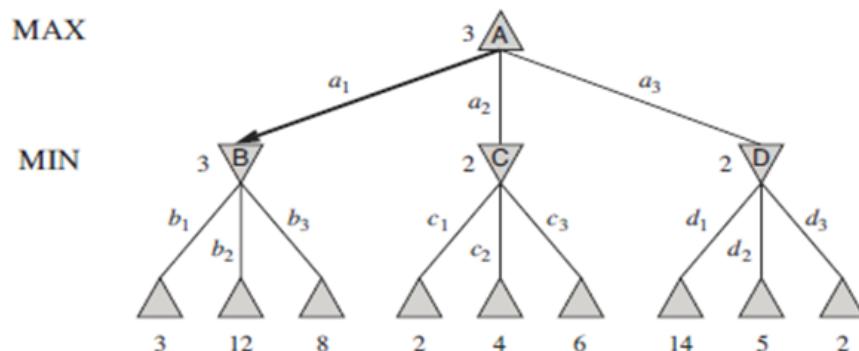
改善:

- 使用alpha-beta pruning算法来减少搜索节点
- 对节点进行采样、而非逐一搜索 (i.e., MCTS)

枚举当前局面之后每一种下法，然后计算每个后续局面的赢棋概率，选择概率最高的后续局面

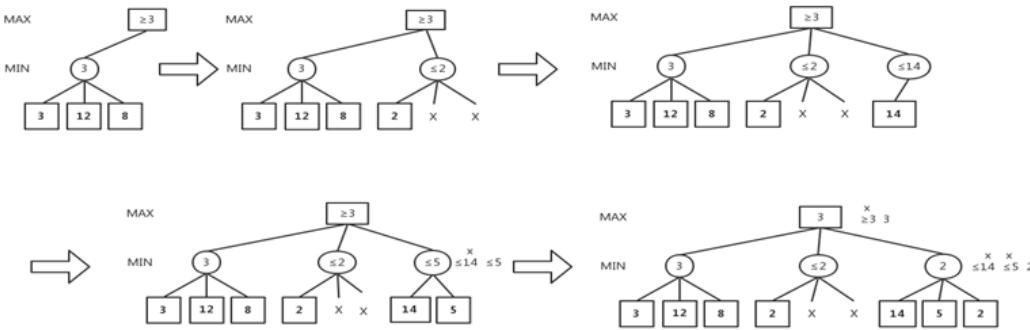
所以提出了：

alpha-beta 剪枝 (alpha-beta pruning)



MAX 在A，搜索搜完了B，发现能给出3，在搜索C的后续时候遇到了2，就不会在搜索4 6结点了。

剪枝剪去不影响最后结果的 搜索分支，如上的 c_2, c_3 :



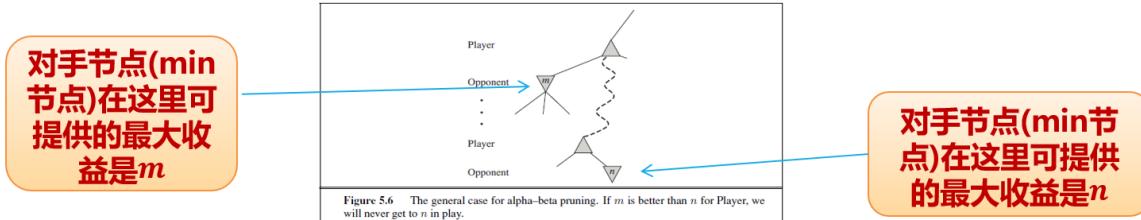
在很大的树中可以快速减除掉很多不影响结果的分支.

什么时候可以剪枝:

注意蓝字:

对抗搜索：如何利用Alpha-Beta 剪枝

Alpha值(α)	玩家MAX（根节点）目前得到的最高收益
	假设 n 是MIN节点，如果 n 的一个后续节点可提供的收益小于 α ，则 n 及其后续节点可被剪枝
Beta值(β)	玩家MIN目前给对手的最小收益
	假设 n 是MAX节点，如果 n 的一个后续节点可获得收益大于 β ，则 n 及其后续节点可被剪枝
α 和 β 的值初始化分别设置为 $-\infty$ 和 ∞	



在图中 $m > n$ ，因此 n 右边节点及后续节点就被剪枝掉了

- α 为可能解法的最大上界，请注意这是MIN结点，就是MIN玩家在这个结点可以选很多结果，MIN选了一个最小的。

请注意是 MAX玩家 在这一层的搜索获得收益目前已经可以有 α 了，所以接下来小于 α 的都被剪枝。

还有一种表达：MAX(根节点) 目前能得到的 最高收益 .

- β 为可能解法的最小下界 .

MIN 目前给对手的最小收益 .

变化总体过程： α 从负无穷大($-\infty$)逐渐增加、 β 从正无穷大(∞)逐渐减少。

每个节点同时有 α 和 β ，如果一个节点中 $\alpha > \beta$ ，则该节点的后续节点可剪枝，如何理解：

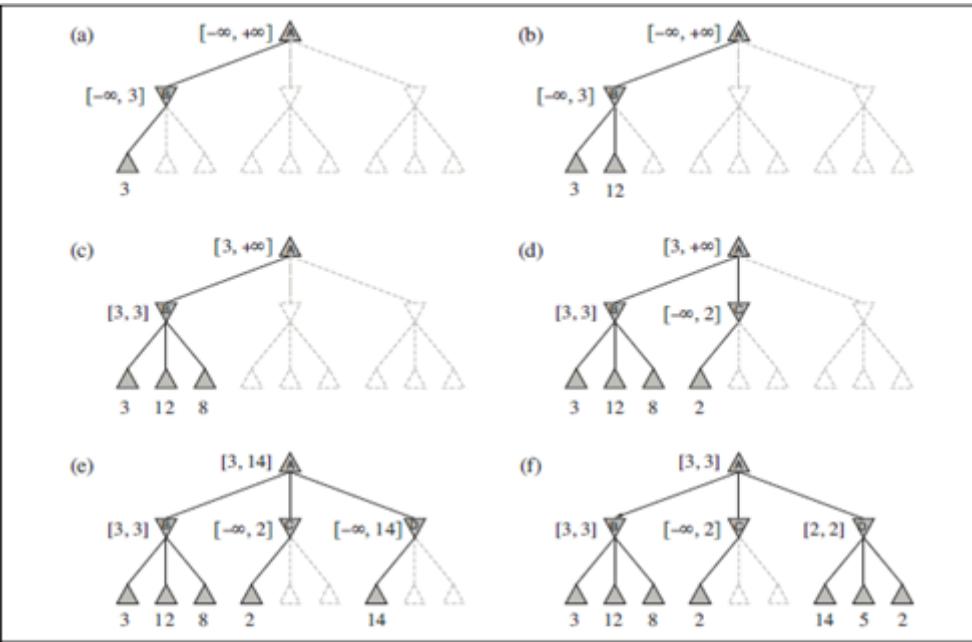
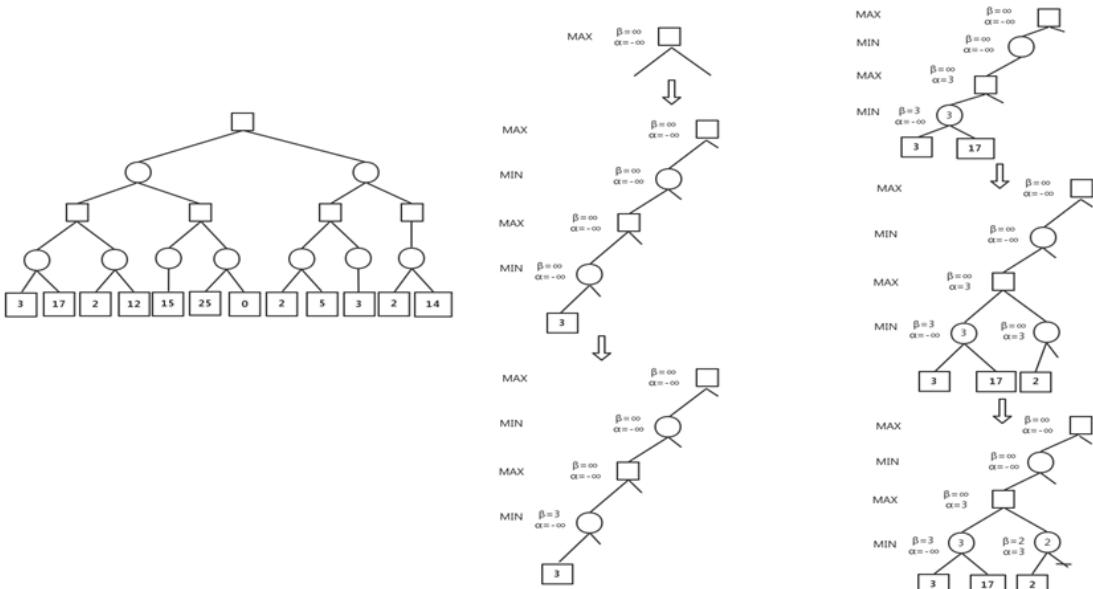


Figure 5.5 Stages in the calculation of the optimal decision for the game tree in Figure 5.2. At each point, we show the range of possible values for each node. (a) The first leaf below B has the value 3. Hence, B , which is a MIN node, has a value of *at most* 3. (b) The second leaf below B has a value of 12; MIN would avoid this move, so the value of B is still at most 3. (c) The third leaf below B has a value of 8; we have seen all B 's successor states, so the value of B is exactly 3. Now, we can infer that the value of the root is *at least* 3, because MAX has a choice worth 3 at the root. (d) The first leaf below C has the value 2. Hence, C , which is a MIN node, has a value of *at most* 2. But we know that B is worth 3, so MAX would never choose C . Therefore, there is no point in looking at the other successor states of C . This is an example of alpha–beta pruning. (e) The first leaf below D has the value 14, so D is worth *at most* 14. This is still higher than MAX's best alternative (i.e., 3), so we need to keep exploring D 's successor states. Notice also that we now have bounds on all of the successors of the root, so the root's value is also at most 14. (f) The second successor of D is worth 5, so again we need to keep exploring. The third successor is worth 2, so now D is worth **exactly 2**. MAX's decision at the root is to move to B , giving a value of 3.

- 上图(a)是剪出了一个 α : 3

注意下图 右下角部分：如果一个节点中 $\alpha > \beta$ ，到2那个结点 β 就被更新了。



注意上图，分 MAX 结点 和 MIN 结点，分情况，感觉上另一个是 $+\infty$ ，但是条件到了就会更新到有限的值了，比如右下角的2结点。

```

function ALPHA-BETA-SEARCH(state) returns an action
  v  $\leftarrow$  MAX-VALUE(state,  $-\infty$ ,  $+\infty$ )
  return the action in ACTIONS(state) with value v

```

```

function MAX-VALUE(state,  $\alpha$ ,  $\beta$ ) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v  $\leftarrow -\infty$ 
  for each a in ACTIONS(state) do
    v  $\leftarrow \text{MAX}(v, \text{MIN-VALUE}(\text{RESULT}(s, a), \alpha, \beta))$ 
    if v  $\geq \beta$  then return v
     $\alpha \leftarrow \text{MAX}(\alpha, v)$ 
  return v

```

```

function MIN-VALUE(state,  $\alpha$ ,  $\beta$ ) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v  $\leftarrow +\infty$ 
  for each a in ACTIONS(state) do
    v  $\leftarrow \text{MIN}(v, \text{MAX-VALUE}(\text{RESULT}(s, a), \alpha, \beta))$ 
    if v  $\leq \alpha$  then return v
     $\beta \leftarrow \text{MIN}(\beta, v)$ 
  return v

```

Figure 5.7 The alpha–beta search algorithm. Notice that these routines are the same as the MINIMAX functions in Figure 5.3, except for the two lines in each of MIN-VALUE and MAX-VALUE that maintain α and β (and the bookkeeping to pass these parameters along).

蒙特卡洛树搜索

单一状态蒙特卡洛规划: 多臂赌博机(multi-armed bandits)

游戏大厅有多个赌博机，摇动某个赌博机的手臂 会吐出或者不吐硬币，问在 n 次摇动赌博机之后，接下来将摇动哪一个。

- 单一状态：只有一个玩家。
- k 种行动：摇动 k 个赌博机其中一个的手臂。

序列决策问题，这种问题需要在利用(**exploitation**)和探索(**exploration**)之间保持平衡。

形式化描述：

- 定义悔值函数：

如果有 k 个赌博机，这 k 个赌博机产生的操作序列为 $X_{i,1}, X_{i,2}, \dots$ ($i = 1, \dots, K$)。在时刻 $t = 1, 2, \dots$ ，选择第 I_t 个赌博机后，可得到奖赏 $X_{I_t,t}$ ，则在 n 次操作 I_1, \dots, I_n 后，可如下定义悔值函数：

$$R_n = \max_{i=1,\dots,k} \sum_{t=1}^n X_{i,t} - \sum_{t=1}^n X_{I_t,t}$$

假设有一个上帝知道当前摇动的最优赌博机是哪一个，将摇动这个得到的奖赏（也是玩家可以得到的最大奖赏）减 玩家选择的那个，就是 悔值。

越小越好。

- 一个良好的多臂赌博机操作的策略是在不同人进行了多次玩法后，能够让悔值函数的方差最小。每个人玩完体验都差不多。

西瓜书上方法

ϵ -贪心

见西瓜书 P374.

- 增量更新方式 更新 $Q(k)$, 和最后平均是一样的, 但是这样只用存两个变量.
- 以 ϵ 的概率探索: 随机选取. 否则选 $Q(k)$ 最大的.
- ϵ 可随尝试次数逐渐减小.

Softmax 算法

- 基于当前已知的摇臂平均奖赏来对探索和利用进行折中.
- 式(16.4)中 τ 的大小决定 利用/探索 的折中.

以上做法有局限, 没有考虑到马尔可夫决策过程的结构.

上限置信区间策略(Upper Confidence Bound Strategies, UCB)

在探索-利用 (exploration-exploitation) 之间取得平衡。

- 在UCB方法中, 使 $X_{i,T_i(t-1)}$ 来记录第 i 个赌博机在过去 $t-1$ 时刻内的平均奖赏, 则在第 t 时刻, 选择使如下具有最佳上限置区间的赌博机:

$$I_t = \max_{i \in \{1, \dots, k\}} \{\bar{X}_{i,T_i(t-1)} + c_{t-1,T_i(t-1)}\}$$

其中 $c_{t,s}$ 取值定义如下:

$$c_{t,s} = \sqrt{\frac{2\ln t}{s}}$$

$T_i(t) = \sum_{s=1}^t \prod(I_s = i)$ 为在过去时刻 (初始时刻到 t 时刻) 过程中选择第 i 个赌博机的次数总和。

不仅考虑平均奖赏, 还要考虑探索(那些比较少被摇动但是有潜能的).

- s : 在UCB中就是次数.

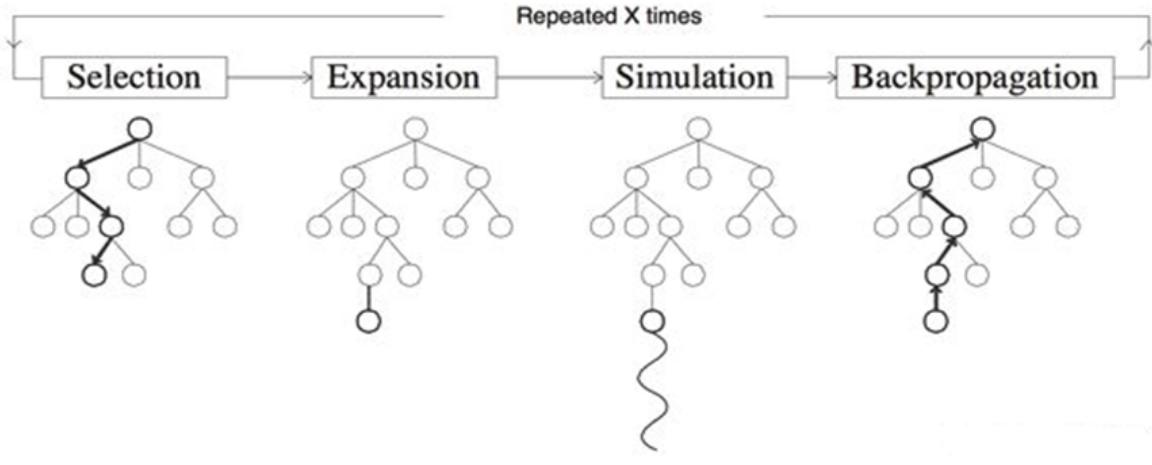
也就是说, 在第 t 时刻, UCB算法一般会选择具有如下最大值的第 j 个赌博机:

$$UCB = \bar{X}_j + \sqrt{\frac{2\ln n}{n_j}} \text{ 或者 } UCB = \bar{X}_j + C \times \sqrt{\frac{2\ln n}{n_j}}$$

\bar{X}_j 是第 j 个赌博机在过去时间内所获得的平均奖赏值, n_j 是在过去时间内拉动第 j 个赌博机臂膀的总次数, n 是过去时间内拉动所有赌博机臂膀的总次数。 C 是一个平衡因子, 其决定着在选择时偏重探索还是利用。

UCB算法最开始先每个赌博机都拉动一次, 接下来就选UCB最大那个的了.

蒙特卡洛树搜索(Monte-Carlo Tree Search)



四个步骤：

- 选择(selection):

选择最具“潜力”的后续节点，直到一个叶子结点 L .

同样在利用和探索之间取得平衡，可以使用UCB算法选择.

- 扩展(expansion):

访问到了叶子结点，但叶子结点 L 不是终止结点，随机创建其后的一个未被访问节点，选择该节点作为后续子节点 C .

- 模拟(simulation):

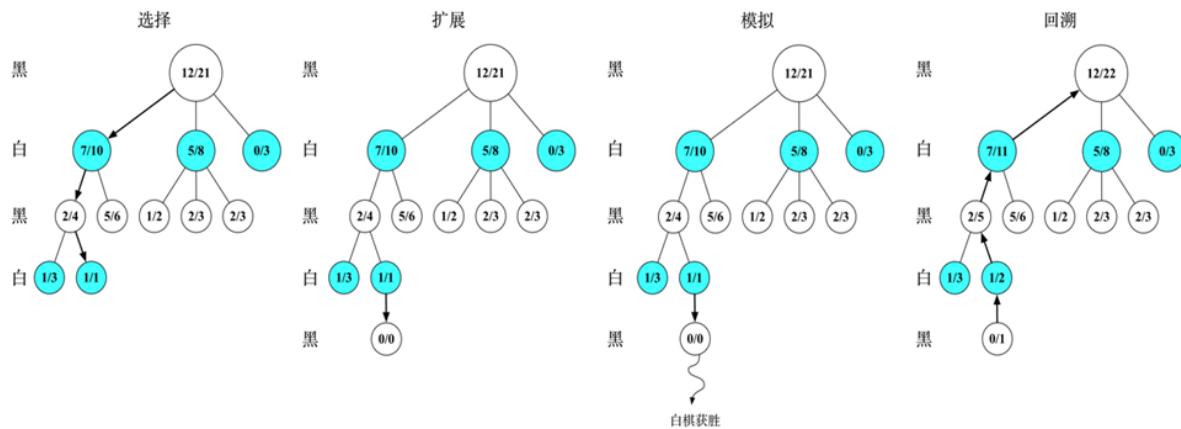
从扩展结点 C 出发，进行模拟仿真对抗，直到博弈结束.

- 反向传播(Back-Propagation):

记录模拟博弈的结果(胜/败)，向上反向传播，回溯更新.

例子：围棋蒙特卡洛树搜索

这些结点记录的信息都是模拟后反向传播的.



- 以围棋为例，假设根节点是执黑棋方。

- 图中每一个节点都代表一个局面，每一个局面记录两个值A/B:

- A: 该局面被访问中黑棋胜利次数。对于黑棋表示己方胜利次数，对于白棋表示己方失败次数（对方胜利次数）；
- B: 该局面被访问的总次数。

选择:

选择后续结点，现在每一层的后续结点就像多臂赌博机：由UCB1公式来计算，取一个值最大的：

- 左1: 7/10对应的局面奖赏值为 $\frac{7}{10} + \sqrt{\frac{\log(21)}{10}} = 1.252$
- 左2, 5/8对应的局面奖赏值为 $\frac{5}{8} + \sqrt{\frac{\log(21)}{8}} = 1.243$
 - 左3, 0/3对应的局面评估分数为 $\frac{0}{3} + \sqrt{\frac{\log(21)}{3}} = 1.007$
- 由此可见，黑棋会选择局面7/10进行行棋。

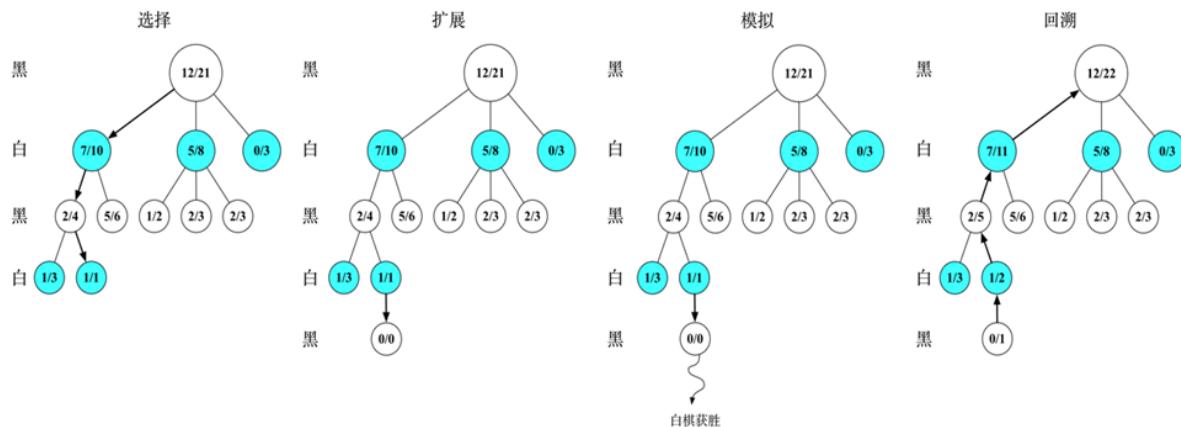
比如7/10结点，说明上一层结点模拟时往这里走了10次（就像是多臂赌博机 玩家之前玩的历史记录中，选择这个选了10次）。

在节点7/10，由白棋行棋，评估该节点下的两个后续结点（此时A记录的是白棋失败的次数，所以第一项为1-A/B）：

- 左1, 2/4对应的局面奖赏为 $1 - \frac{2}{4} + \sqrt{\frac{\log(10)}{4}} = 1.26$
- 左2, 5/6对应的局面奖赏为 $1 - \frac{5}{6} + \sqrt{\frac{\log(10)}{6}} = 0.786$

由此可见，白棋会选择局面2/4进行行棋。

扩展:



到1/1之后扩展。初始化为0/0，接着在该节点下进行模拟。

假设经过一次仿真行棋后，最终白棋获胜。

模拟, 反向传播:

更新该仿真路径上每个节点的A/B值，该新节点的A/B值被更新为0/1，并向上回溯到该仿真路径上新节点的所有父辈节点，即所有父辈节点的A不变（因为该轮仿真是白棋赢了），B值加1。

使用蒙特卡洛树搜索的原因

- Monte-Carlo Tree Search (MCTS): 蒙特卡洛树搜索基于采样来得到结果、而非穷尽式枚举（虽然在枚举过程中也可剪掉若干不影响结果的分支）。

基于采样的。

伪代码

蒙特卡洛树搜索算法(UCT)

```
function UCTSEARCH( $s_0$ )
    create root node  $v_0$  with state  $s_0$ 
    while within computational budget do
         $v_l \leftarrow \text{TREEPOLICY}(v_0)$ 
         $\Delta \leftarrow \text{DEFAULTPOLICY}(s(v_l))$ 
        BACKUP( $v_l, \Delta$ )
    return  $a(\text{BESTCHILD}(v_0, 0))$ 

function TREEPOLICY( $v$ )
    while  $v$  is nonterminal do
        if  $v$  not fully expanded then
            return EXPAND( $v$ )
        else
             $v \leftarrow \text{BESTCHILD}(v, C_p)$ 
    return  $v$ 

function DEFAULTPOLICY( $s$ )
    while  $s$  is non-terminal do
        choose  $a \in A(s)$  uniformly at random
         $s \leftarrow f(s, a)$ 
    return reward for state  $s$ 
```

S	状态集
$A(s)$	在状态 s 能够采取的有效行动的集合
$s(v)$	节点 v 所代表的状态
$a(v)$	所采取的行动导致到达节点 v
$f : S \times A \rightarrow S$	状态转移函数
$N(v)$	节点 v 被访问的次数
$Q(V)$	节点 v 所获得的奖赏值
$\Delta(v, p)$	玩家 p 选择节点 v 所得到的奖赏值

v_0

```
function BACKUP( $v, \Delta$ )
    while  $v$  is not null do
         $N(v) \leftarrow N(v) + 1$ 
         $Q(v) \leftarrow Q(v) + \Delta(v, p)$ 
         $v \leftarrow \text{parent of } v$ 
```

```
function EXPAND( $v$ )
    choose  $a \in \text{untried actions from } A(s(v))$ 
    add a new child  $v'$  to  $v$ 
    with  $s(v') = f(s(v), a)$ 
    and  $a(v') = a$ 
    return  $v'$ 
```

```
function BESTCHILD( $v, c$ )
    return  $\arg \max_{v' \in \text{children of } v} \frac{Q(v')}{N(v')} + c \sqrt{\frac{2 \ln N(v)}{N(v')}}$ 
```

ppt里面有树搜索过程图解。

10 单选 (2分) 下面对minimax搜索、alpha-beta剪枝搜索和蒙特卡洛树搜索的描述中，哪一句描述是不正确的（ ）

得分/总分

- A. alpha-beta剪枝搜索和蒙特卡洛树搜索都是非穷举式搜索 X0.00/2.00
- B. 对于一个规模较小的游戏树，alpha-beta剪枝搜索和minimax搜索的结果会不同
- C. 三种搜索算法中，只有蒙特卡洛树搜索是采样搜索
- D. minimax是穷举式搜索

正确答案: B 你错选为A

- A. 从当前节点出发来选择后续节点
- B. 判断搜索算法的时间复杂度
- C. 计算从当前节点到目标节点之间的最小代价值
- D. 判断搜索算法的空间复杂度

✓2.00/2.00

正确答案: C 你选对了

逻辑与推理

命题逻辑

- 命题逻辑(**proposition logic**)是应用一套形式化规则对以符号表示的描述性陈述进行推理的系统。
- 在命题逻辑中, 一个或真或假的描述性陈述被称为原子命题, 对原子命题的内部结构不做任何解析。
- 若干原子命题可通过逻辑运算符来构成复合命题。

可通过命题联结词(connectives) 对已有命题进行组合, 得到新命题。这些通过命题联结词得到的命题被称为复合命题(compound proposition)。假设存在命题 p 和 q , 下面介绍五种主要的命题联结词:

命题连接符号	表示形式	意义
与(and)	$p \wedge q$	命题合取(conjunction), 即“ p 且 q ”
或(or)	$p \vee q$	命题析取(disjunction), 即“ p 或 q ”
非(not)	$\neg p$	命题否定(negation), 即“非 p ”
条件(conditional)	$p \rightarrow q$	命题蕴含(implication), 即“如果 p 则 q ”
双向条件(bi-conditional)	$p \leftrightarrow q$	命题双向蕴含(bi-implication), 即“ p 当且仅当 q ”

逻辑等价: 给定命题 p 和命题 q , 如果 p 和 q 在所有情况下都具有同样真假结果, 那么 p 和 q 在逻辑上等价, 一般用 \equiv 来表示, 即 $p \equiv q$ 。

推理规则

命题逻辑中的推理规则

假言推理 (Modus Ponens)	$\frac{\alpha \Rightarrow \beta, \alpha}{\beta}$
与消解 (And-Elimination)	$\frac{\alpha_1 \wedge \alpha_2 \wedge \dots \wedge \alpha_n}{\alpha_i (1 \leq i \leq n)}$
与导入 (And-Introduction)	$\frac{\alpha_1, \alpha_2, \dots, \alpha_n}{\alpha_1 \wedge \alpha_2 \wedge \dots \wedge \alpha_n}$

双重否定 (Double-Negation Elimination)	$\frac{\neg\neg\alpha}{\alpha}$
单项消解或单项归 结 (Unit Resolution)	$\frac{\alpha \vee \beta, \neg\beta}{\alpha}$
消解或归结 (Resolution)	$\frac{\alpha \vee \beta, \neg\beta \vee \gamma}{\alpha \vee \gamma}, \frac{\alpha_1 \vee \alpha_2 \vee \dots \vee \alpha_m, \neg\beta}{\alpha_1 \vee \alpha_2 \vee \dots \vee \alpha_{k-1} \vee \alpha_{k+1} \vee \dots \vee \alpha_m} (\neg\alpha_k = \neg\beta)$

具体可看 <数理逻辑> 课程.

谓词逻辑

谓词逻辑

- 在谓词逻辑中，将原子命题进一步细化，分解出个体、谓词和量词，来表达个体与总体的内在联系和数量关系，这就是谓词逻辑研究内容。
- 谓词逻辑中三个核心概念：
 - 个体、谓词（predicate）和量词（quantifier）

■ 谓词逻辑的推理例子

证明过程:

已知:

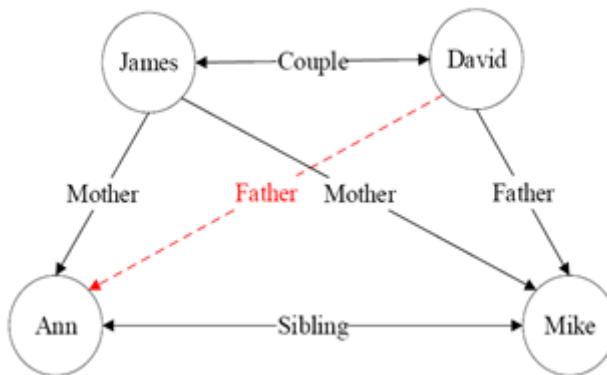
- $(\forall x)(F(x) \rightarrow (G(x) \wedge H(x)))$
- $(\exists x)(F(x) \wedge P(x))$

试证明: $(\exists x)(P(x) \wedge H(x))$

1. $(\forall x)(F(x) \rightarrow (G(x) \wedge H(x)))$ (已知)
2. $(\exists x)(F(x) \wedge P(x))$ (已知)
3. $F(a) \wedge P(a)$ (2的EI)
4. $F(a) \rightarrow (G(a) \wedge H(a))$ (1的UI)
5. $F(a)$ (由3知)
6. $G(a) \wedge H(a)$ (4和5的假言推理)
7. $P(a)$ (由3知)
8. $H(a)$ (由6知)
9. $P(a) \wedge H(a)$ (7和8的合取)
10. $(\exists x)(P(x) \wedge H(x))$ (9的EG)

知识图谱推理

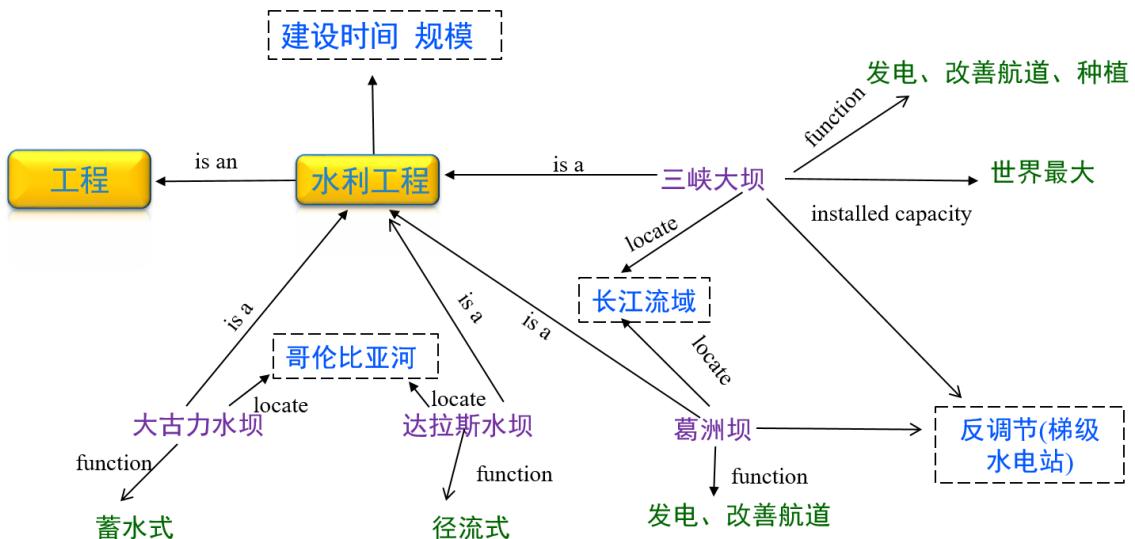
知识图谱可视为包含多种关系的图。



在图中，每个节点是一个实体（如人名、地名、事件和活动等），任意两个节点之间的边表示这两个节点之间存在的关系。

一般而言，可将知识图谱中任意两个相连节点及其连接边表示成一个三元组 (*triplet*)，即 (*left_node, relation, right_node*)，例: (*David, Father, Mike*)。

知识图谱的构成：以水利工程为例



□ 概念之间层次化关系(ontology):

□ 如: 工程 → 水利工程

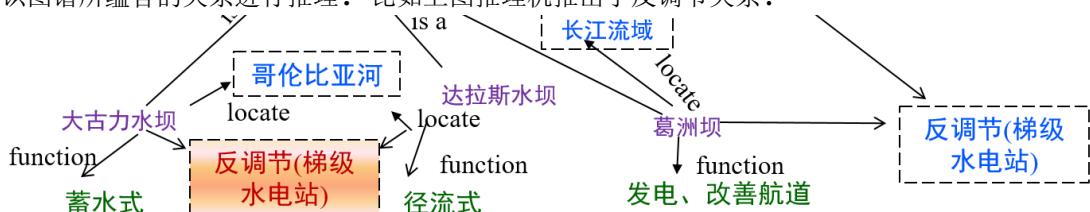
□ 与 Wordnet 等早期本体知识构建不同, 现有方法多在传统分类法 (Taxonomy) 中结合大众分类(Folksonomy)和机器学习来构建语义网络分类体系。

□ 概念对应的例子或实体(instance/entity)

□ 如三峡大坝和葛洲坝等属于水利工程这一概念。

□ 一般通过分类识别等手段实现。

对知识图谱所蕴含的关系进行推理。比如上图推理机推出了反调节关系:



知识图谱的构成：以水利工程为例

□ 概念或实体的属性:

□ 属性是对概念或实体内涵的描述, 如水利工程具有建设时间和规模等属性、三峡大坝具有发电功能等属性。

□ 概念或实体之间的关系:

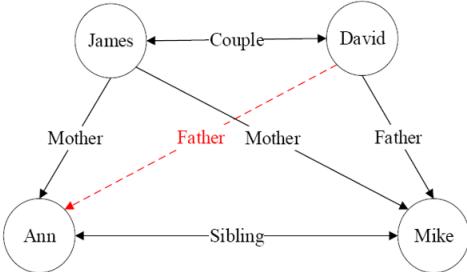
□ 如三峡大坝和葛洲坝之间具有“梯级调节”关系。

□ 概念或实体的属性描述和关系表达一般通过三元组来表示:

□ $(entity, relation, entity)$ 或 $(subject, predicate, object)$

□ 学习概念或实体属性描述及其关联关系是丰富知识图谱的关键!

可利用一阶谓词来表达刻画知识图谱中节点之间存在的关系，从而为基于知识图谱的推理创造了条件。



一个简单的家庭关系知识图谱

问题：如何从知识图谱中推理得到

$\text{father}(\text{David}, \text{Ann})$



$$(\forall x)(\forall y)(\forall z)(\text{Mother}(z, y) \wedge \text{Couple}(x, z) \rightarrow \text{Father}(x, y))$$

如果能够学习得到这条规则，该有多好？
(从具体例子中学习，这是归纳推理的范畴)

知识图谱推理：归纳学习

归纳逻辑程序设计 (inductive logic programming, ILP) 算法

归纳逻辑程序设计 (ILP) 是机器学习和逻辑程序设计交叉领域的研究内容。

ILP 使用一阶谓词逻辑进行知识表示，通过修改和扩充逻辑表达式对现有知识归纳，完成推理任务。

作为ILP的代表性方法，FOIL (First Order Inductive Learner) 通过序贯覆盖实现规则推理。

FOIL (First Order Inductive Learner)

利用已有的正例 负例 背景知识样例。

反例条件：只能在已知两个实体的关系 且确定其关系与目标谓词相悖时，才能将这两个实体用于构建目标谓词的反例，而不能在不知两个实体是否满足目标谓词前提下将它们来构造目标谓词的反例。

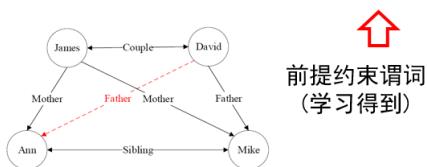
需要从知识图谱中的特定样例出发，来推理出泛化的推理规则。

具体来说：从一般到特殊：对目标谓词或前提约束谓词中的变量赋予具体值，直到所构成的推理规则不覆盖任何反例。

如将 $(\forall x)(\forall y)(\forall z)(\text{Mother}(z, y) \wedge \text{Couple}(x, z) \rightarrow \text{Father}(x, y))$ 这一推理规则所包含的目标谓词 $\text{Father}(x, y)$ 中 x 和 y 分别赋值为 David 和 Ann ，进而进行推理。

通过信息增益值来判断 哪些谓词暂时不能作为目标谓词的约束谓词。

$$(\forall x)(\forall y)(\forall z)(\text{Mother}(z, y) \wedge \text{Couple}(x, z) \rightarrow \boxed{\text{Father}(x, y)})$$



前提约束谓词
(学习得到)

目标谓词
(已知)

- $\text{Mother}(\cdot, \cdot)$
- $\text{Sibling}(\cdot, \cdot)$
- $\text{Couple}(\cdot, \cdot)$

依次将谓词加入到推理规则中作为前提约束谓词，并计算所得到新推理规则的FOIL增益值。
基于计算所得FOIL增益值来选择最佳前提约束谓词。

选择最大的信息增益的前提约束谓词来加入推理规则。

总结：

推理手段： *positive examples + negative examples + background knowledge examples* \Rightarrow hypothesis

背景知识 样例 集合	Sibling(Ann, Mike) Couple(David, James) Mother(James, Ann) Mother(James, Mike)	目标谓词 训练样例 集合	Father(David, Mike) \neg Father(David, James) \neg Father(James, Ann) \neg Father(James, Mike) \neg Father(Ann, Mike)	给定目标谓词，FOIL算法从实例（正例、反例、背景样例）出发，不断测试所得到推理规则是否还包含反例，一旦不包含负例，则学习结束，展示了“ 归纳学习 ”能力。
------------------	---	--------------------	---	---

PPT里有详细 每一步谓词的过程，看得不是很懂。

FOIL (First Order Inductive Learner) 算法

FOIL算法	
输入：	目标谓词 P , P 的训练样例（正例集合 E^+ 和反例集合 E^- ），其他背景知识
输出：	推导得到目标谓词 P 的推理规则
1	将目标谓词作为所学习推理规则的结论
2	将其他谓词逐一作为前提约束谓词加入推理规则，计算所得到推理规则的 FOIL 信息增益值，选取最优前提约束谓词以生成新推理规则，并将训练样例集合中与该推理规则不符的样例去掉
3	重复2过程，直到所得到的推理规则不覆盖任意反例

强化学习

- 以逻辑推理为核心的符号主义人工智能。
- 以数学建模为核心的机器学习。
- 以环境交互为核心的强化学习。

在与环境交互之中进行学习，得到最大化收益。

智能体：agent

	有监督学习	无监督学习	强化学习
学习依据	基于监督信息	基于对数据结构的假设	基于评价 (evaluative)
数据来源	一次性给定	一次性给定	在交互中产生 (interactive)
决策过程	单步 (one-shot)	无	序列 (sequential)
学习目标	样本到语义标签的映射	同一类数据的分布模式	选择能够获取最大收益的状态到动作的映射

强化学习的特点

- **基于评估：** 强化学习利用环境评估当前策略，以此为依据进行优化
- **交互性：** 强化学习的数据在与环境的交互中产生
- **序列决策过程：** 智能主体在与环境的交互中需要作出一系列的决策，这些决策往往是前后关联的

注：现实中常见的强化学习问题往往还具有奖励滞后，基于采样的评估等特点

序列学习方法，将强化学习看成离散马尔可夫过程。

- 马尔可夫链 (Markov Chain)：满足马尔可夫性 (Markov Property) 的离散随机过程，也被称为离散马尔可夫过程。 $t+1$ 时刻状态仅与 t 时刻状态相关。

$$Pr(X_{t+1} = x_{t+1}|X_0 = x_0, X_1 = x_1, \dots, X_t = x_t) = Pr(X_{t+1} = x_{t+1}|X_t = x_t)$$

$$Pr(X_{t+1} = x_{t+1}|X_t = x_t, X_{t-1} = x_{t-1})$$

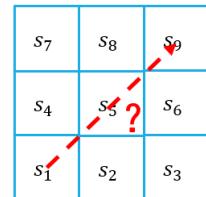
例子：

强化学习示例

(序列优化) 问题：

- 在下图网格中，假设有一个机器人位于 s_1 ，其每一步只能向上或向右移动一格，跃出方格会被惩罚（且游戏停止）
- 如何使用强化学习找到一种策略，使机器人从 s_1 到达 s_9 ？

刻画解该问题的因素



智能主体	迷宫机器人
环境	3×3 方格
状态	机器人当前时刻所处方格
动作	每次移动一个方格
奖励	到达 s_9 时给予奖励；越界时给予惩罚

状态转移概率 $Pr(S_{(t+1)}|S_t)$ 满足马尔可夫性。

- 从第 t 步状态转移到第 $t+1$ 步状态给予奖励机制。引入与环境交互，衡量决策序列的优劣。

马尔可夫奖励过程 (Markov Reward Process)

马尔可夫奖励过程 (Markov Reward Process)

问题：给定两个因为状态转移而产生的奖励序列 $(1,1,0,0)$ 和 $(0,0,1,1)$ ，哪个奖励序列更好？

为了比较不同的奖励序列，定义反馈 (return)，用来反映累加奖励：

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots$$

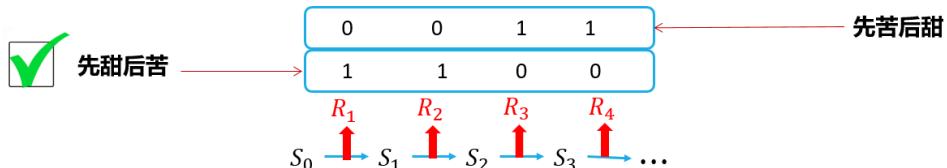
其中折扣系数 (discount factor) $\gamma \in [0, 1]$

假设 $\gamma = 0.99$

$$(1,1,0,0): G_0 = 1 + 0.99 \times 1 + 0.99^2 \times 0 + 0.99^3 \times 0 = 1.99$$

$$(0,0,1,1): G_0 = 0 + 0.99 \times 0 + 0.99^2 \times 1 + 0.99^3 \times 1 = 1.9504$$

反馈值反映了某个时刻后所得到累加奖励，当衰退系数小于1时，越是遥远的未来对累加反馈的贡献越少



马尔可夫奖励过程 (Markov Reward Process)

使用离散马尔可夫过程描述机器人移动问题

- 随机变量序列 $\{S_t\}_{t=0,1,2,\dots}$: S_t 表示机器人第 t 步的位置，每个随机变量 S_t 的取值范围为 $S = \{s_1, s_2, \dots, s_9, s_d\}$
- 状态转移概率: $Pr(S_{t+1}|S_t)$ 满足马尔可夫性
- 定义奖励函数 $R(S_t, S_{t+1})$: 从 S_t 到 S_{t+1} 所获得奖励，其取值如图中所示
- 定义衰退系数: $\gamma \in [0, 1]$

综合以上信息，可用 $MRP = \{S, Pr, R, \gamma\}$ 来刻画马尔科夫奖励过程

0	0	1
0	0	0
0	0	0

这个模型不能体现机器人能动性，仍然缺乏

与环境进行交互的手段

$R_{(t+1)}$ 为 $R(S_t, S_{(t+1)})$ 的简写，即 t 到 $t + 1$ 的奖励。

马尔可夫决策过程 (Markov Decision Process) : 引入动作

马尔可夫决策过程是强化学习基本求解框架，智能体与环境交互过程中可自主决定所采取的动作，不同动作会对环境产生不同影响。

- 定义智能主体能够采取的动作集合为 A
- 由于不同的动作对环境造成的影响不同，因此状态转移概率定义为 $Pr(S_{t+1}|S_t, a_t)$ ，其中 $a_t \in A$ 为第 t 步采取的动作
- 奖励可能受动作的影响，因此修改奖励函数为 $R(S_t, a_t, S_{t+1})$

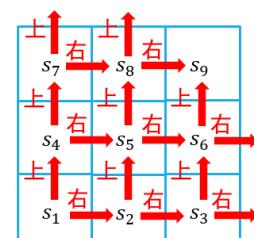
重新定义了状态转移概率(引入 a_t)，状态转移可以是确定的(概率为1)。

马尔可夫决策过程 (Markov Decision Process)

使用离散马尔可夫过程描述机器人移动问题

- 随机变量序列 $\{S_t\}_{t=0,1,2,\dots}$: S_t 表示机器人第 t 步所在位置（即状态），每个随机变量 S_t 的取值范围为 $S = \{s_1, s_2, \dots, s_9, s_d\}$
- 动作集合: $A = \{\text{上}, \text{右}\}$
- 状态转移概率 $Pr(S_{t+1}|S_t, a_t)$: 满足马尔可夫性，其中 $a_t \in A$ 。状态转移如图所示。
- 奖励函数: $R(S_t, a_t, S_{t+1})$
- 衰退系数: $\gamma \in [0, 1]$

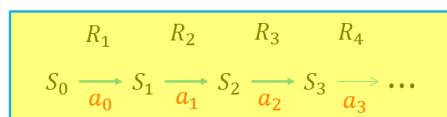
综合以上信息，可通过 $MDP = \{S, A, Pr, R, \gamma\}$ 来刻画马尔科夫决策过程



在每一个状态执行一个动作，进入下一个状态，得到了一定的奖励。

马尔可夫决策过程 (Markov Decision Process)

- 马尔可夫决策过程 $MDP = \{S, A, Pr, R, \gamma\}$ 是刻画强化学习中环境的标准形式
- 马尔可夫决策过程可用如下序列来表示：



马尔科夫过程中产生的状态序列为轨迹(trjectory)，可如下表示

$$(S_0, a_0, R_1, S_1, a_1, R_2, \dots, S_T)$$

- 轨迹长度可以是无限的，也可以有终止状态 S_T 。有终止状态的问题叫做分段的（即存在回合的）(episodic)，否则叫做持续的(continuing)

分段问题中，一个从初始状态到终止状态的完整轨迹称为一个片段或回合(episode)。

如围棋对弈中一个胜败对局为一个回合。

马尔可夫决策过程 (Markov Decision Process) 中的策略学习

马尔可夫决策过程 $MDP = \{S, A, Pr, R, \gamma\}$ 对环境进行了描述，那么智能主体如何与环境交互而完成任务？需要进行策略学习

对环境中各种因素的说明

已知的： S, A, R, γ

不一定已知的： Pr

观察到的： $(S_0, a_0, R_1, S_1, a_1, R_2, \dots, S_T)$

策略函数：

- 策略函数 $\pi: S \times A \mapsto [0, 1]$ ，其中 $\pi(s, a)$ 的值表示在状态 s 下采取动作 a 的概率。
- 策略函数的输出可以是确定的，即给定 s 情况下，只有一个动作 a 使得概率 $\pi(s, a)$ 取值为 1。
对于确定的策略，记为 $a = \pi(s)$ 。

马尔可夫决策过程 (Markov Decision Process) 中的策略学习

如何进行策略学习：一个好的策略是在当前状态下采取了一个行动后，该行动能够在未来收到最大化的反馈：

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots$$

为了对策略函数 π 进行评估，定义

- **价值函数 (Value Function)** $V: S \mapsto \mathbb{R}$ ，其中 $V_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s]$ ，即在第 t 步状态为 s 时，按照策略 π 行动后在未来所获得反馈值的期望
- **动作-价值函数 (Action-Value Function)** $q: S \times A \mapsto \mathbb{R}$ ，其中 $q_\pi(s, a) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a]$ 表示在第 t 步状态为 s 时，按照策略 π 采取动作 a 后，在未来所获得反馈值的期望

由马尔可夫性，未来的状态和奖励只与当前状态相关，与 t 无关。因此 t 取任意值该等式均成立，如“逢山开路，遇水搭桥”。

这样，策略学习转换为如下优化问题：

寻找一个最优策略 π^* ，对任意 $s \in S$ 使得 $V_{\pi^*}(s)$ 值最大



具体的在策略 π 下价值函数 和 动作-价值函数的计算：

价值函数与动作-价值函数的关系：对策略进行评估

$$\begin{aligned} V_\pi(s) &= \mathbb{E}_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots | S_t = s] \\ &= \mathbb{E}_{a \sim \pi(s, \cdot)} [\mathbb{E}_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots | S_t = s, A_t = a]] \end{aligned}$$

$$= \sum_{a \in A} \pi(s, a) q_\pi(s, a)$$

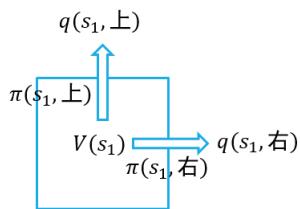
$$\begin{aligned} q_\pi(s, a) &= \mathbb{E}_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots | S_t = s, A_t = a] \\ &= \mathbb{E}_{s' \sim Pr(\cdot | s, a)} [R(s, a, s') + \gamma \mathbb{E}_\pi[R_{t+2} + \gamma R_{t+3} + \dots | S_{t+1} = s']] \end{aligned}$$

$$= \sum_{s' \in S} Pr(s' | s, a) [R(s, a, s') + \gamma V_\pi(s')]$$

具体的例子：

$$V_\pi(s) = \sum_{a \in A} \pi(s, a) q_\pi(s, a)$$

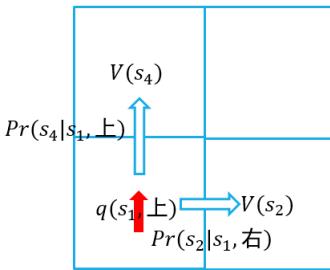
$$V_\pi(s_1) = \pi(s_1, \text{上}) q_\pi(s_1, \text{上}) + \pi(s_1, \text{右}) q_\pi(s_1, \text{右})$$



不同动作下的反馈累加

$$q_\pi(s, a) = \sum_{s' \in S} Pr(s' | s, a) [R(s, a, s') + \gamma V_\pi(s')]$$

$$q_\pi(s_1, \text{上}) = Pr(s_4 | s_1, \text{上}) [R(s_1, \text{上}, s_4) + \gamma V_\pi(s_4)]$$



动作确定时状态转移后的反馈结果

- 马尔科夫性质：

$$P(S_{t+1} | S_t) = P(S_{t+1} | S_t, S_{t-1}, \dots, S_1)$$

- S : 状态的集合，比如每一种棋盘(上面有棋子).
- A : 动作集合，比如移动棋子.
- P : 状态转移矩阵. 是一个概率分布(因为这是判断对方做的动作，所以是有概率的，做某个动作在某个状态下的影响是一定的)，在状态 s 采取动作 a ，转移到 \rightarrow 状态 s' 的概率：

$$P_{ss'}^a = P\{S_{t+1} = s' | S_t = s, A_t = a\}$$

- R : 当前状态采取动作转移到下一个状态的奖励，reward 是随机变量.

reward function(期望)：

$$R_s^a = E[R_{t+1} | S_t = s, A_t = a]$$

- γ : 对以后的 reward 进行衰减.

- π : 策略，是一个随机变量(有分布)：

$$\pi(a | s) = P\{A_t = a | S_t = s\}$$

- state-value function 西瓜书377页 (16.5).

注意这里依据马尔可夫性质，对接下来所有可能的状态的奖赏 以某种方式求和.

表示 指定"状态"上/指定"状态-动作"上的累积奖赏.

请注意这两个函数中的 r 是对状态转移的概率以及奖赏（可能转移到的所有概率）全概率相乘之后的结果。

因为

$$\pi(x, a) = P(\text{action} = a | \text{state} = x)$$

表示在状态 x 下选择动作 a 的概率，又因为动作事件之间两两互斥且和为动作空间，由全概率展开公式

$$P(A) = \sum_{i=1}^{\infty} P(B_i)P(A | B_i)$$

可得

$$\begin{aligned} & \mathbb{E}_\pi \left[\frac{1}{T} r_1 + \frac{T-1}{T} \frac{1}{T-1} \sum_{t=2}^T r_t \mid x_0 = x \right] \\ &= \sum_{a \in A} \pi(x, a) \sum_{x' \in X} P_{x \rightarrow x'}^a \left(\frac{1}{T} R_{x \rightarrow x'}^a + \frac{T-1}{T} \mathbb{E}_\pi \left[\frac{1}{T-1} \sum_{t=1}^{T-1} r_t \mid x_0 = x' \right] \right) \end{aligned}$$

其中

$$r_1 = \pi(x, a) P_{x \rightarrow x'}^a R_{x \rightarrow x'}^a$$

最后一个等式用到了递归形式。

这就类似 DP.

累积奖赏是在迭代计算并不断减小的，只需设置一个阈值跳出即可。

- 理想的策略应能最大化累积奖赏。
- state-action value function (16.5) 西瓜书P380:**
最优Bellman等式。同上仅需把求和代换成取max。仍然后递推形式。

贝尔曼方程 (Bellman Equation)

刻画了价值函数和行动-价值函数自身以及两者相互之间的递推关系。

$$V_\pi(s) = \sum_{a \in A} \pi(s, a) q_\pi(s, a) \quad q_\pi(s, a) = \sum_{s' \in S} Pr(s'|s, a) [R(s, a, s') + \gamma V_\pi(s')]$$

将右式带入左式，得到价值函数的贝尔曼方程

$$V_\pi(s) = \sum_{a \in A} \pi(s, a) \sum_{s' \in S} Pr(s'|s, a) [R(s, a, s') + \gamma V_\pi(s')]$$

将左式带入右式，得到行动-价值函数的贝尔曼方程

$$q_\pi(s, a) = \sum_{s' \in S} Pr(s'|s, a) [R(s, a, s') + \gamma \sum_{a' \in A} \pi(s', a') q_\pi(s', a')]$$

将利用贝尔曼方程进行策略评估，进而进行策略优化

$$\begin{aligned} V_\pi(s) &= \mathbb{E}_\pi [R_{t+1} + \gamma V_\pi(S_{t+1}) | S_t = s] \\ q_\pi(s, a) &= \mathbb{E}_\pi [R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}) | S_t = s, A_t = a] \end{aligned}$$

策略优化与策略评估

寻找一个最优策略 π^* 在任意状态 s 都是反馈的最大期望。基于反馈期望调整策略函数。

策略优化

这里的方法为：

- 基于价值 (Value-based) 的方法
对价值函数进行建模和估计，以此为依据制订策略

• 策略优化定理：

对于确定的策略 π 和 π' ，如果对于任意状态 $s \in S$ ：

$$q_{\pi}(s, \pi'(s)) \geq q_{\pi}(s, \pi(s))$$

那么对于任意状态 $s \in S$ ，有：

$$V_{\pi'}(s) \geq V_{\pi}(s)$$

因此给定当前策略 π 、价值函数 V_{π} 和行动-价值函数 q_{π} 时，可如下构造新的策略 π' ，只要 π' 满足如下条件：

$$\pi'(s) = \operatorname{argmax}_a q_{\pi}(s, a) \quad (\text{对于任意 } s \in S)$$

π' 便是对 π 的一个改进

收获的最大的反馈期望，作为在状态 s 所做的行为。

具体例子：

第一部分：策略优化 (Policy Improvement)

给定当前策略 π 、价值函数 V_{π} 和行动-价值函数 q_{π} 时，可如下构造新的策略 π' ， π' 要满足如下条件：

$$\pi'(s) = \operatorname{argmax}_a q_{\pi}(s, a) \quad (\text{对于任意 } s \in S)$$

$$q_{\pi}(s, a) = \sum_{s' \in S} Pr(s'|s, a) [R(s, a, s') + \gamma V_{\pi}(s')]$$

假设当前价值函数在右图中给出，策略用箭头表示，对状态 s_1 而言：

注意这个问题里状态转移是确定的

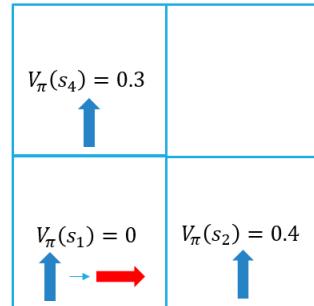
$$q_{\pi}(s_1, \text{上}) = 0 + 0.99 \times 0.3 = 0.297$$

$$q_{\pi}(s_1, \text{右}) = 0 + 0.99 \times 0.4 = 0.396$$

$$R(s_1, \text{右}, s_2) \quad \gamma \quad V_{\pi}(s_2)$$

因此根据策略优化原则 $\pi'(s) = \operatorname{argmax}_a q_{\pi}(s, a)$

更新状态 s_1 策略 $\pi'(s_1) = \operatorname{argmax}_a q_{\pi}(s, a) = \text{右}$
如上，计算了 q_{π} ，然后依据策略优化原则取最大的。



通过迭代计算贝尔曼方程进行策略评估

基于动态规划的价值函数更新：使用迭代的方法求解贝尔曼方程组

初始化 V_{π} 函数
循环
枚举 $s \in S$
 $V_{\pi}(s) \leftarrow \sum_{a \in A} \pi(s, a) \sum_{s' \in S} Pr(s'|s, a) [R(s, a, s') + \gamma V_{\pi}(s')]$
直到 V_{π} 收敛

更新 $V_{\pi}(s_1)$ 的值：

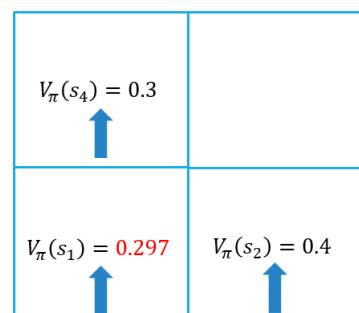
$$q_{\pi}(s_1, \text{上}) = 1 \times (0 + 0.99 \times 0.3) + 0 \times (0 + 0.99 \times 0.4)$$

$$+ \dots = 0.297$$

$$V_{\pi}(s_1) = 1 \times q_{\pi}(s_1, \text{上}) + 0 \times q_{\pi}(s_1, \text{右}) = 0.297$$

动态规划法的缺点：1) 智能主体需要事先知道状态转移概率；2)

无法处理状态集合大小无限的情况



画加粗红线的地方：

- $1 \times (0 + 0.99 \times 0.3)$

表示： s_1 向上移动到 s_4 的概率 (是 1) \times (该动作从环境收获的反馈 (是 0) + 衰减系数 (0.99) \times 在 s_4 状态智能体收获的反馈期望 (是 0.3))

上图最后就显示了在状态 s_1 的价值函数得到了更新.

基于蒙特卡洛采样的价值函数更新

选择不同的起始状态，按照当前策略 π 采样若干轨迹，记它们的集合为 D
枚举 $s \in S$

计算 D 中 s 每次出现时对应的反馈 G_1, G_2, \dots, G_k

$$V_\pi(s) \leftarrow \frac{1}{k} \sum_{i=1}^k G_i$$

假设按照当前策略可样得到以下两条轨迹

$$(s_1, s_4, s_7, s_8, s_9)$$

$$(s_1, s_2, s_3, s_d)$$

s_1 对应的反馈值分别为

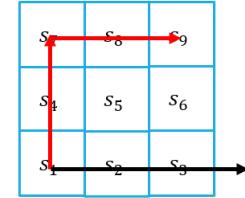
$$0 + \gamma \times 0 + \dots + \gamma^3 \times 1 = 0.970$$

$$0 + \gamma \times 0 + \gamma^2 \times (-1) = -0.980$$

因此估计

$$V(s_1) = \frac{1}{2}(0.970 - 0.980) = -0.005$$

如果是确定的策略，
每个起点只会产生
一种轨迹



上图： s_d 为越出了 3×3 方格的状态，此时终止且没有完成任务。

上图采样，每一次采样都需要 达到终止状态。得到了两条轨迹，综合考虑他们的反馈值。

蒙特卡洛采样法的缺点：

- 状态集合比较大时，一个状态在轨迹可能非常稀疏，不利于估计期望
- 在实际问题中，最终反馈需要在终止状态才能知晓，导致反馈周期较长

基于时序差分 (Temporal Difference) 的价值函数更新

初始化 V_π 函数

循环

 初始化 s 为初始状态

 循环

$$a \sim \pi(s, \cdot)$$

 执行动作 a ，观察奖励 R 和下一个状态 s'

$$\text{更新 } V_\pi(s) \leftarrow V_\pi(s) + \alpha [R(s, a, s') + \gamma V_\pi(s') - V_\pi(s)]$$

$$s \leftarrow s'$$

 直到 s 是终止状态

 直到 V_π 收敛

- 根据贝尔曼方程 $V_\pi(s) = \mathbb{E}_{a \sim \pi(s, \cdot), s' \sim p_{r(\cdot|s, a)}}[R(s, a, s') + \gamma V_\pi(s')]$

- 利用蒙特卡洛采样的思想，通过采样 a 和 s' 来估计期望

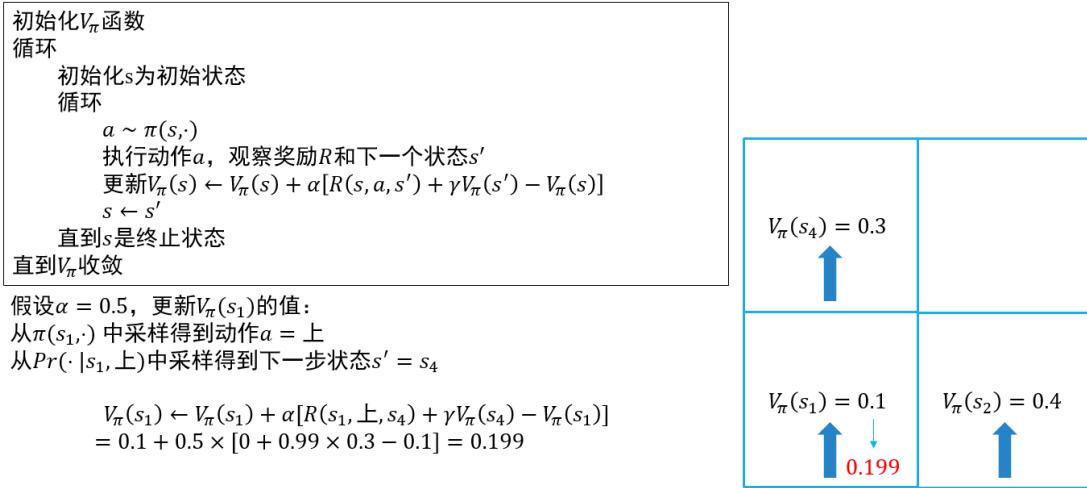
- $R(s, a, s') + \gamma V_\pi(s')$ 是对 $V_\pi(s)$ 的一个估计值

- 部分更新 $V_\pi(s)$ 的值： $V_\pi(s) \leftarrow (1 - \alpha)V_\pi(s) + \alpha[R(s, a, s') + \gamma V_\pi(s')]$

$$\frac{\text{过去的}}{\text{价值函数值}} \quad \frac{\text{学习得到的}}{\text{价值函数值}}$$

基于时序差分：首先从某一个状态出发，起始状态，从该状态任意采样得到一个动作，执行该动作，观察下一个状态反馈期望的大小。

基于时序差分 (Temporal Difference) 的价值函数更新



根据贝尔曼方程更新.

Q-Learning

Q函数就是之前定义的动作价值函数的大小.

选择一个行为, 能得到最大的反馈期望. 动作 a 是策略优化的思想得到的. 环境能够给予状态 s 到 s' 多大的奖励. 当前状态从 s 变换到 s' .

AlphaGo 是先监督学习, 学的差不多了开始强化学习. 两个bot一直对抗, 对抗结果给 reward.

强化学习最常用的应用就是打游戏. 这里机器看到的就是像人一样看到图像, 对每个图像的反应都是它自己学出来的.

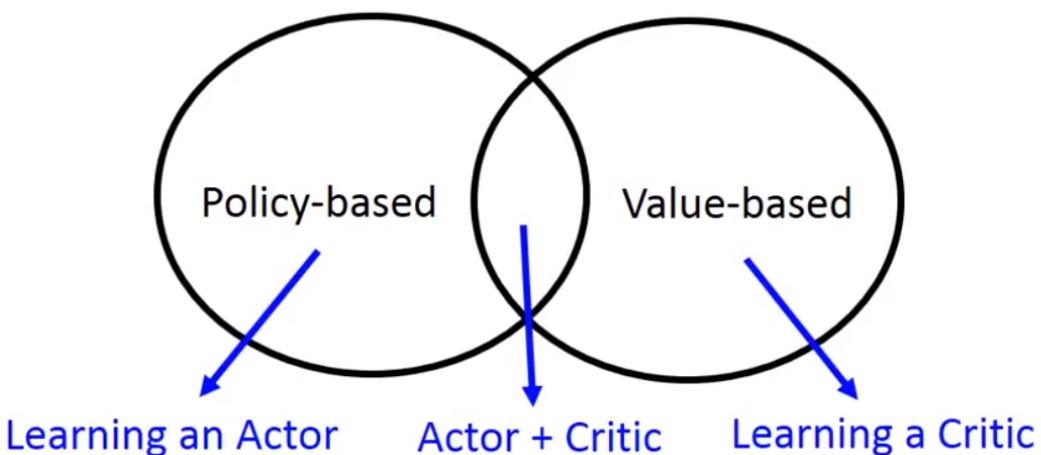
一轮游戏就是一个episode, 强化学习就是要最大化每一个episode的奖赏.

- Reward delay: 强化学习会遇到, 就是短期的牺牲可以带来长远的回报.
- 还需要探索那些未做的动作.

两类: Policy-based(学习到一个带动作的), Value-based(学习到一个评价(批判)的).

Outline

Alpha Go: policy-based + value-based + model-based



Asynchronous Advantage Actor-Critic (A3C)

Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, Koray Kavukcuoglu, "Asynchronous Methods for Deep Reinforcement Learning", ICML, 2016

Created with EverCam.
<http://www.camdemmy.com>

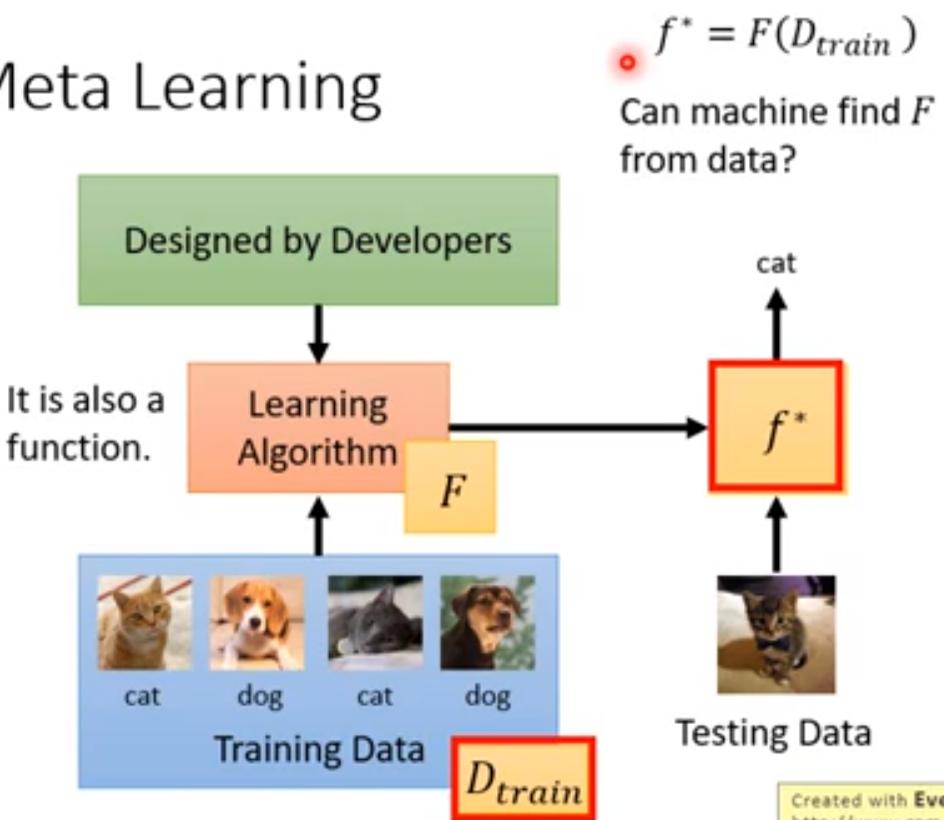
Meta Learning

定义

学习如何去学习，learn to learn.

一般的机器学习，有一个Learning Algorithm(我们设计的)，从一堆训练集中学习参数，遇到测试集可以做出判断。Meta Learning不只是学到了特征等，还学到了如何学习，这样就会学得更快。

Meta Learning



Created with EverCam.
<http://www.camdemmy.com>

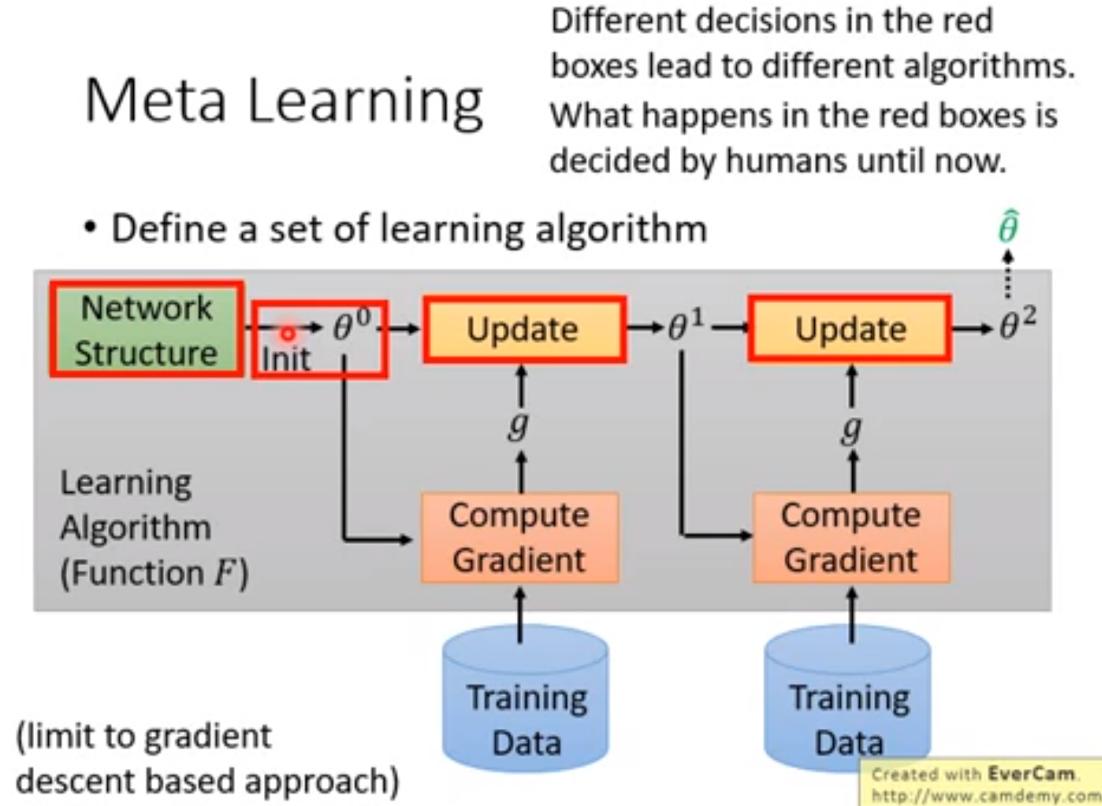
Meta Learning 就是找到Learning Algorithm F , 这个 F 的能力是可以找到最优预测函数 f^* .

$$F(D_{train}) = f^*$$

Meta Learning 就是找到最好的 Learning Algorithm, 同样也可以有loss.

Learning Algorithm Set

- F 的例子：根据梯度下降 网络找到最优的 $\hat{\theta}$ 这个过程：



不同的网络结构，不同的 θ^0 ，都可以看做不同的Learning Algorithm. 红框里面的需要 Meta Learning.

上面这些构成了Learning Algorithm Set

Learning Algorithm 的评估

F 在第 i 个Task上训练出了一个预测函数 f^i ，测试集上的loss为 l^i ，那么 F 的loss为 (N 个任务)：

$$L(F) = \sum_{i=1}^N l^i$$

训练Task是很多的. 训练 Task 和 测试 Task.

MAML: Model-Agnostic Meta-Learning for Fast Adaptation of Deep Network

以前的Initialization是从一个分布中采样，现在就要学习一个Initialization.

MAML

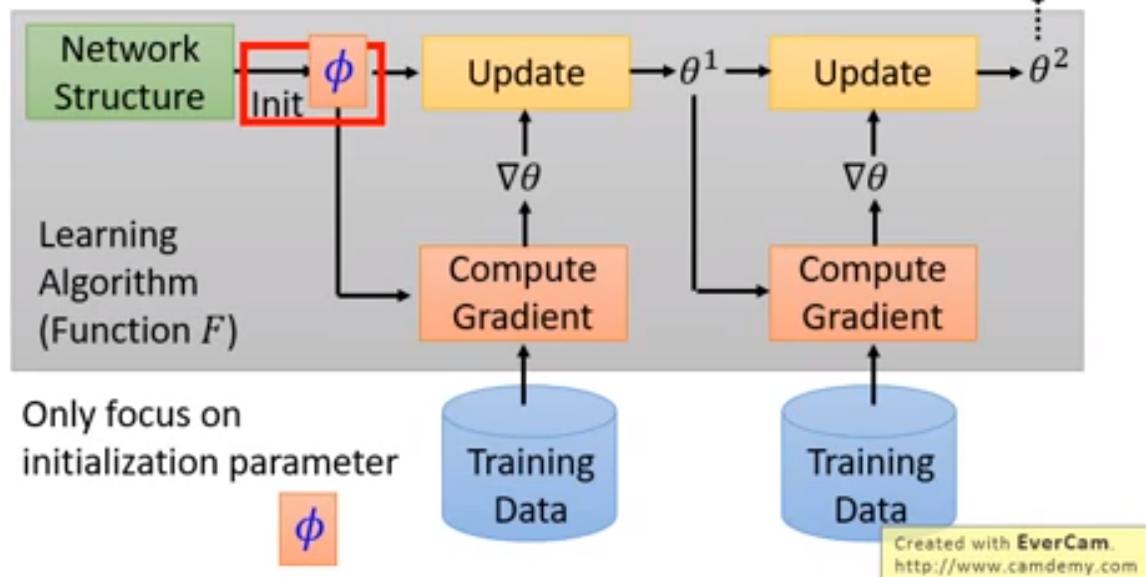
$\hat{\theta}^n$: model learned from task n

Loss Function:

$$L(\phi) = \sum_{n=1}^N l^n(\hat{\theta}^n)$$

$\hat{\theta}^n$ depends on ϕ

$l^n(\hat{\theta}^n)$: loss of task n on the testing set of task n



各变量解释看上图. 每一个小任务都是训练的过程.

- 如何优化 Gradient Descent:

$$X_{ij}^* = \begin{cases} X_{ij} + \epsilon \cdot \text{sign}(\nabla_{X_{ij}} \mathcal{L}), & X_{ij} \in \text{range} \\ X_{ij} & \text{others} \end{cases}$$

ϕ 训练到 $\hat{\theta}$

- 与 Model Pre-training 的区别:

Model Pre-training 损失函数的自变量是超参数(这里就是 ϕ), 但是MAML的损失函数自变量是 经过 F 学习得到的 f 的最优参数在test set上的损失.

Model Pre-training 关注的是即刻的 ϕ 给进去loss如何, 而 MAML 关注的是经过训练后怎么样.

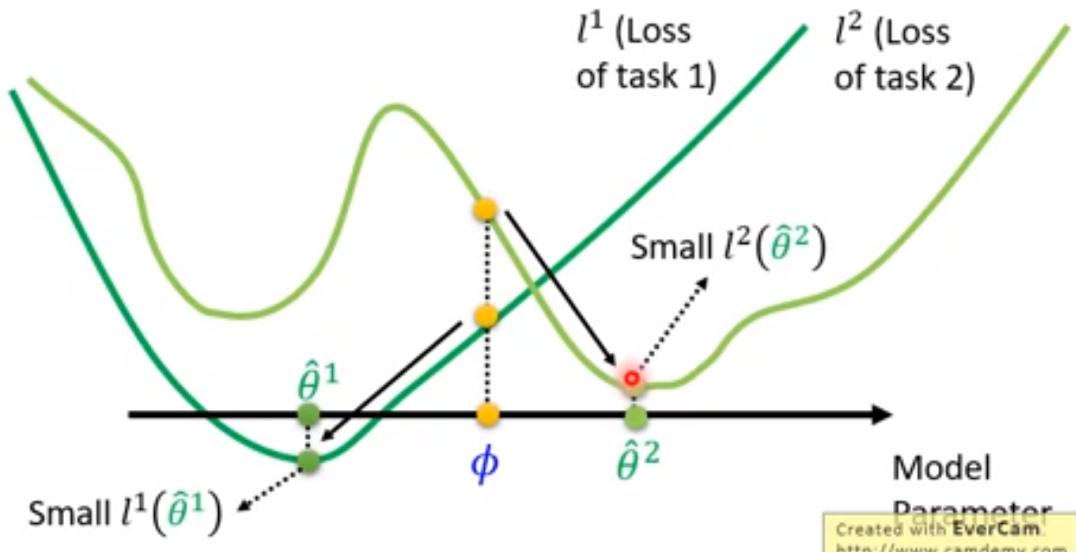
MAML 并不在意Initialization: ϕ 在 training Task(set) 上的表现, 而是经过 ϕ 训练出来的 $f^*/\hat{\theta}$ 的表现. | 即对于学习器的loss(上面有): $l^i(\phi)$ 可能不是很好, 但是经过这个 ϕ 训练过后的 $\hat{\theta}$ 也就是 $l^i(\hat{\theta})$ 可能很好. 不同Task的 $\hat{\theta}$ 当然不同:

MAML

$$L(\phi) = \sum_{n=1}^N l^n(\hat{\theta}^n)$$

我們不在意 ϕ 在 training task 上表現如何

我們在意用 ϕ 訓練出來的 $\hat{\theta}^n$ 表現如何



- 训练细节：

在训练的时候希望 Gradient Update 一次就到最优，但是测试时可以Update多次.

Few-Shot Learning 就希望只Update一次.

- Toy Example:

有一个 y , 一个Task就是从 y 采样出不同的 x , 通过这些采样的来估测 y 的形式.

$$y = a \sin(x + b)$$

不同的Task: 选不同的 a, b 即可.

- Model Pre-training 的结果：

Toy Example

Model Pre-training

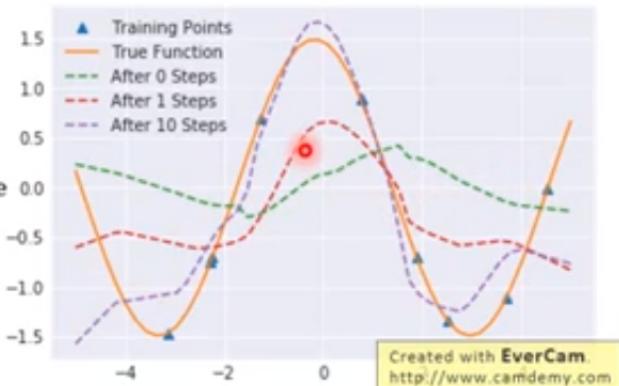


因为拿 ϕ 要在所有 y 上都表现好，所有 y 综合就是上图.

◦

Source of images
<https://towardsdatascience.com/paper-repro-deep-metalearning-using-maml-and-reptile-fd1df1cc81b0>

MAML



Few-Shot Learning

Training Set 是 Support Set, Test Set 是 Query Set.

- * N-ways K-shot classification:
在 Training 和 Test Tasks 里，有 N 个类别，每个类别有 K 个用例.
- 一般从每类样本中抽取 $k + 1$ 个实例，其中 k 个实例是训练集，还有一个是测试集.

Why few-shot learning?

- 尽量像人一样看过一些就会学到很多.
- 从少量样本中学习.
- 减少数据收集和计算的消耗.

相关领域：

- 弱监督学习：标签不完整不准确有噪声的，使用无标签数据引入额外信息.
- 迁移学习：先验知识由原任务迁移到目标任务.
- Meta-Learning：学习到跨任务的知识，利用新任务上的特有信息解决问题.

有一种说法：训练集里面没有Test样本的类别，为此给Support Set. 本质的学到一个函数处理相似度：

Basic Idea

- Learn a similarity function: $\text{sim}(\mathbf{x}, \mathbf{x}')$.
- Ideally, $\text{sim}(\mathbf{x}_1, \mathbf{x}_2) = 1$, $\text{sim}(\mathbf{x}_1, \mathbf{x}_3) = 0$, and $\text{sim}(\mathbf{x}_2, \mathbf{x}_3) = 0$.

Bulldog



\mathbf{x}_1

Bulldog



\mathbf{x}_2

Fox



\mathbf{x}_3

- Training Set 很大，是让Model知道图片之间的区别 哪些很像.
- 而Support Set每个类对应的样本很少，让Model知道每个是什么，来了一个Test Set，一张张和Support Set对比，像的就是那个类了.

数据集：

Omniglot

- 50 different alphabets. (Every alphabet has many characters.)
- 1,623 unique characters (i.e., classes).
- Each character was written by 20 different people (i.e., each class has 20 samples.)
- The samples are 105×105 images.
- Training set:
 - 30 alphabets, 964 characters (classes), and 19,280 samples.
- Test set:
 - 20 alphabets, 659 characters (classes), and 13,180 samples.

定义假设空间中的一个假设 h , 期望

Given a hypothesis h , we want to minimize its **expected risk** R :

$$R(h) = \int \ell(h(x), y) dp(x, y) = \mathbb{E}[\ell(h(x), y)]$$

As $p(x, y)$ is unknown , we can compute **empirical risk** by sampling:

$$R_I(h) = \frac{1}{I} \sum_{i=1}^I \ell(h(x_i), y_i)$$

- $\hat{h} = \arg \min_h R(h)$ be the function that minimizes the expected risk;
- $h^* = \arg \min_{h \in \mathcal{H}} R(h)$ be the function in \mathcal{H} that minimizes the expected risk;
- $h_I = \arg \min_{h \in \mathcal{H}} R_I(h)$ be the function in \mathcal{H} that minimizes the empirical risk.

$$\mathbb{E} [R(h_I) - R(\hat{h})] = \underbrace{\mathbb{E} [R(h^*) - R(\hat{h})]}_{\mathcal{E}_{\text{app}}(\mathcal{H})} + \underbrace{\mathbb{E} [R(h_I) - R(h^*)]}_{\mathcal{E}_{\text{est}}(\mathcal{H}, I)}$$

- 第一项为近似误差：衡量假设空间 \mathcal{H} 的假设与最优假设之间的接近程度。
- 第二项为估计误差：衡量在假设空间中用**empirical risk**代替**expected risk**的效果。

数据量大时，采样方式计算得到的 h_I 可能和 h^* 非常接近，但是数据小 \mathcal{E}_{est} 就可能差很多。

减小该估计误差可以从以下3个角度考虑：

- 数据；提供 D_{train} ；
- 模型，决定了假设空间 \mathcal{H} ，可以使用先验知识限制 \mathcal{H} 的复杂度；
- 算法，在 \mathcal{H} 内基于 D_{train} 寻找最优的 H_I 。

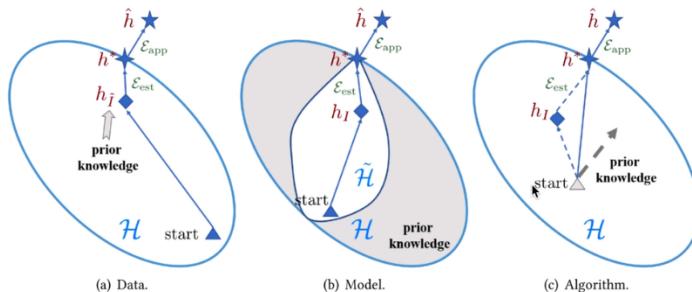


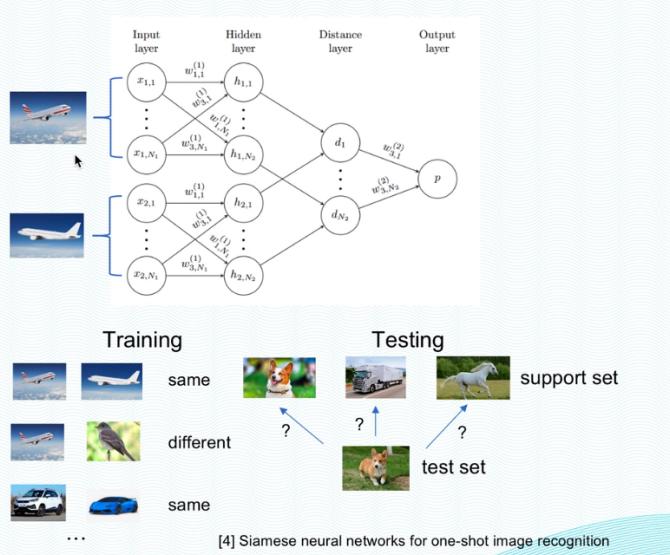
Fig. 2. Different perspectives on how FSL methods solve the few-shot problem.

Embedding Learning 方法

Siamese Neural Network 李生神经网络

Siamese Neural Networks(李生网络)[4]

- 通过监督学习的方式，基于双路神经网络自动发现可以在新样本上泛化的特征；
- 训练时，通过组合的方式构造不同的成对样本，输入网络训练。在上层网络通过计算样本对的距离判断是否属于同一个类，并生成对应的概率分布；
- 测试时，该李生网络处理测试样本和支撑集之间的每一个样本对，最终预测结果为支撑集上预测概率最高的类别。



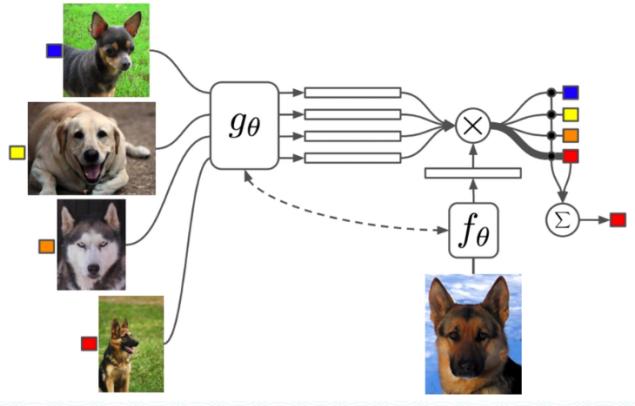
[4] Siamese neural networks for one-shot image recognition

参数是共享的。

Match Network 匹配网络

Match Network(匹配网络)[5]

- 给定支撑集 $S = \{(x_i, y_i)\}_{i=1}^k$ 和待测试样本 \hat{x} ，其预测输出为 $\hat{y} = \sum_{i=1}^k a(\hat{x}, x_i) y_i$ ；
- $a(\hat{x}, x_i) = e^{c(f(\hat{x}), g(x_i))} / \sum_{j=1}^k e^{c(f(\hat{x}), g(x_j))}$ ，实质就是对cos距离的softmax；
- 设计支撑集中样本的embedding为 $g(x_i, S)$ ，基于双向LSTM来学习，每个支撑训练样本的embedding是其他训练样本的函数。
- 设计测试样本的embedding为 $f(\hat{x}, S) = \text{attLSTM}(f'(\hat{x}), g(S), K)$ 基于attention-LSTM使得每个测试样本的embedding是支撑集embedding的函数。

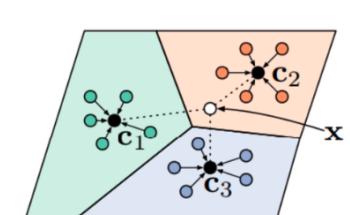


[5] Matching Networks for One Shot Learning

Prototypical Networks 原型网络/Relation Networks 关系网络

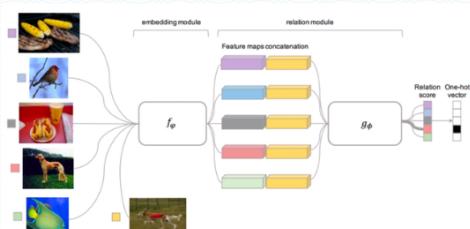
Prototypical Networks(原型网络)[6]

Idea: 每个类别都存在其原型表达，该表达就是支撑集在embedding空间的均值。问题就由分类变成了在embedding空间中的最近邻。



Relation Networks(关系网络)[7]

Idea: 之前的一些工作将重心放在学习一个好的数据表示上，然后使用已有的度量方法来进行分类。如果距离度量也可以通过训练得到，就可以得到泛化能力更好的模型。



[6] Prototypical Networks for Few-shot Learning

[7] Learning to Compare: Relation Network for Few-Shot Learning

原型网络：每一类样本embedding的均值被认为是原型表达。测试样本离哪个更近。

关系网络：学习到一个好的 embedding 网络。去和其他图片embedding之后的结果输入到度量网络，最后看和那个相似。

北大: 人工智能原理

北大的表述和AIMA几乎一致。

Problem Solving Agents

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
    persistent: seq, an action sequence, initially empty
                state, some description of the current world state
                goal, a goal, initially null
                problem, a problem formulation
                action, the most recent action, initially none
    state  $\leftarrow$  UPDATE-STATE(state, percept)
    if seq is empty then
        goal  $\leftarrow$  FORMULATE-GOAL(state)
        problem  $\leftarrow$  FORMULATE-PROBLEM(state, goal)
        seq  $\leftarrow$  SEARCH(problem)
        if seq = failure then return a null action
    action  $\leftarrow$  FIRST(seq)
    seq  $\leftarrow$  REST(seq)
    return action
```

注意persistent。

- 首先根据状态来更新。
- 将第一个动作抽出，并更具其他动作和状态定义问题，根据新定义的问题搜索新的序列。

Related Terms

- 状态空间：问题的状态空间可以形式化地定义为：初始状态、动作和转换模型。
- 图：状态空间形成一个图，其中节点表示状态、链接表示动作。
- 路径：状态空间的一条路径是由一系列动作连接的一个状态序列。

- 初始状态：`In(Arad)`
- 动作：`{Go(zerind), Go(sibiu), Go(Timisoara)}`
- 转化模型：`RESULT(In(Arad), Go(zerind)) = In(zerind)`, `RESULT(s, a) =` 返回在s状态下动作a作用后的状态。
- 目标测试：确定给定的状态是不是目标状态，去Bucharest，所以只有一个元素：
`{In(Bucharest)}`
- 路径代价：每条路径所分配的数值代价，状态s下执行动作a到达s'的代价：
`c(s, a, s')`

抽象：如果我们能够把任何抽象解扩展成为更细节的世界中的解，这种抽象就是有效的。如果执行解中的每个行动比原始问题中的容易，那么这种抽象就是有用的。

选择一个好的问题抽象，包括在保持有效抽象的前提下去除尽可能多的细节，和确保抽象后的行动容易完成。

Example Problems

Vacuum-cleaner world

Example 1: Vacuum-cleaner world 真空吸尘器世界

States 状态

- Agent is in one of two locations, each may contain dirt or not.

智能体在两个地点中的一个，每个也许有灰尘或者没有。

- Possible states, 2 locations: $2 \times 2^2 = 8$ ($n \times 2^n$).

可能的状态，2个地点： $2 \times 2^2 = 8$ ($n \times 2^n$)。

1) Initial state 初始状态

Any state can be as the initial state.

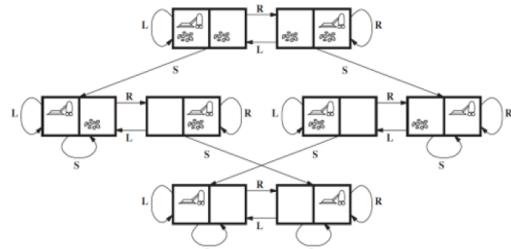
任何状态都可以作为初始状态。

2) Actions 动作

Each state has just three actions:

Left, Right, and Suck.

每个状态仅有三个动作：左移，右移，以及吸尘。



有两个房间，每个房间有两个格子，格子里面有灰尘或者无灰尘，一共有 2×2^2 个状态。上面有状态转移图。

- 转换模型：其中的动作应该合法(在左边不能左移了)并有效果(在没有灰尘的格子不能吸尘)。
- 目标检测：是否所有的区域内都干净。
- 路径代价：等于路径的步数(代价和)。

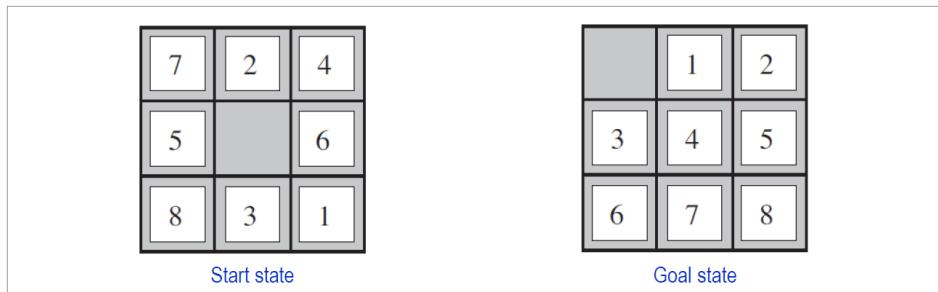
8-puzzle

八数码难题：

Example 2: 8-puzzle 8数码难题

8-puzzle: 3×3 board with 8 numbered tiles and a blank space.

8数码难题： 3×3 棋盘上有8个数字棋子和一个空格。



A tile adjacent to the blank space can slide into the space. The object is to reach a specified goal state.

与空格相邻的滑块可以移向该空格，目的是达到一个指定的目标状态。

注意这里各个量的定义：

States 状态

- Each of 8 numbered tiles in one of the 9 squares, and blank in the last square.
8个数字滑块每个占据一个方格，而空格则位于最后一个方格。

7	2	4
5		6
8	3	1

Start state

1) Initial state 初始状态

- Any state can be the initial state. 任意一个状态都可以成为初始状态。

2) Actions 动作

- Simplest formulation defines the actions as movements of the blank space: Left, Right, Up, or Down.
最简单的形式化是将动作定义为空格的移动：左、右、上、下。
- Different subsets are depending on where the blank is.
不同的子集依赖于空格的位置。

3) Transition model 转换模型

Given a state and action, this returns the resulting state. E.g., if we apply *Left* to the start state, the resulting state has the 5 and the blank switched.

给定状态和动作，其返回结果状态。例如，如果我们对初始状态施加左移动作，由此产生的状态则使5与空格互换。

4) Goal test 目标测试

Checks whether the state matches the goal configuration.
即检查状态是否与目标布局相符。

7	2	4
5	←	6
8	3	1

Transition

5) Path cost 路经代价

The number of steps in the path (each step costs 1).

等于路径的步数（每一步的代价）。

动作的最简单形式化定义是 空格的移动。

滑块难题家族， NP-complete.

8-queens

不能同一行/列/斜线。

形式化：

Two main kinds of formulation:

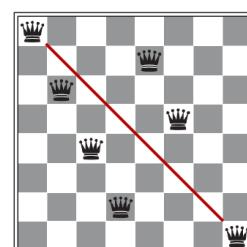
两种主要类型的形式化方法：

- **Incremental formulation:** starts with an empty state, then each action adds a queen to change the state.

增量形式化：从空状态开始，然后每次添加一个皇后改变其状态。

- **Complete-state formulation:** starts with all 8 queens on the board, and moves them around.

全态形式化：初始时8个皇后都放在棋盘上，然后再将她们移开。



A queen attacks another one in the same diagonal.

增量形式化：

- 初始状态：从全空状态开始，每次添加一个皇后。
- 动作：添加一个皇后到任意空格。
- 转换模型：将一个皇后添加到指定空格，返回该棋局(状态)。
- 目标测试：8个皇后都在棋盘上且没有相互攻击。
- 路径代价：总步数。

注意这种形式化下的状态空间大小/状态个数。

书上66页有加约束条件的状态定义方法，且大大降低了状态空间大小。

(书P66) 玩具问题

4 → 任意数字，只阶乘 平方根 取整三个操作。

状态空间无限大。

(书P70) 通过一些约束处理冗余路径

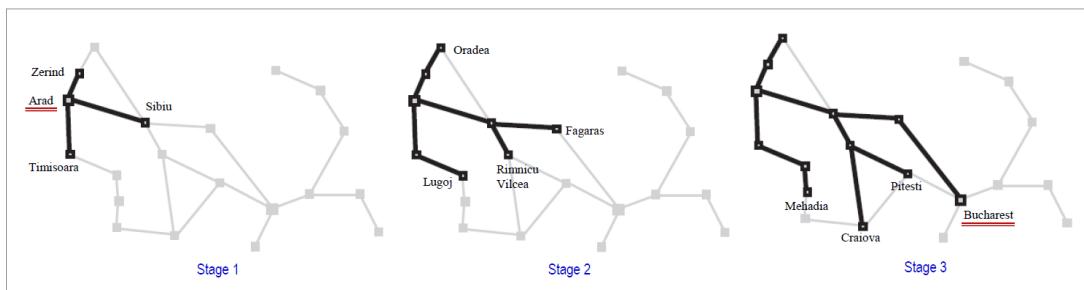
否则会造成状态太多，死循环的问题。

从初始状态出发至任一未被探索的状态都需要经过边缘中的点。边缘：有待扩展的叶子结点。

Searching for Solutions

图搜索的最短路径问题。

通过图搜索在该罗马尼亚地图上生成一系列搜索路径。



Each path has been extended at each stage by one step. Notice that at 3rd stage, the northernmost city (Oradea) has become a dead end.

每个路径在每个阶段通过每一步加以扩展。注意在第3阶段，最北部城市(Oradea)已成为死胡同。

A General Tree-search Algorithm 一种通用的树搜索算法

```
function TREE-SEARCH(problem) returns a solution, or failure
    initialize the frontier using the initial state of problem
    loop do
        if the frontier is empty then return failure
        choose a leaf node and remove it from the frontier
        if the node contains a goal state then return the corresponding solution
        expand the chosen node, adding the resulting nodes to the frontier
```

The *frontier* (also known as *open list*): a data structure, to store the set of all leaf nodes.

该 *frontier* (亦称 *open list*)：一种数据结构，用于存储所有的叶节点。

The process of expanding nodes on the *frontier* continues until either a solution is found or there are no more states to expand.

在 *frontier* 上扩展节点的过程持续进行，直到找到一个解、或没有其它状态可扩展。

树扩展，在每层被选择的结点上扩展。

改进：

```
function GRAPH-SEARCH (problem) returns a solution, or failure
    initialize the frontier using the initial state of problem
    initialize the explored to be empty
    loop do
        if the frontier is empty then return failure
        choose a leaf node and remove it from the frontier
        if the node contains a goal state then return the corresponding solution
        add the node to the explored
        expand the chosen node, adding the resulting nodes to the frontier
        only if not in the frontier or explored
```

The *explored* (aka *closed list*) is an data structure to remember every expanded node.

该*explored* (亦称*closed list*)：一种数据结构，用于记忆每个扩展节点。

The nodes in the *explored* or the *frontier* can be discarded.

*explored*或*frontier*中的节点可以被丢弃。

加了一个*explored*，记忆每个扩展结点，不要重复了。

无信息搜索 (Uninformed Search Strategies)

无信息搜索(盲目搜索)：该术语（无信息、盲目的）意味着 该搜索策略没有超出问题定义提供的状态之外的附加信息。能做的就是生成后继结点 并且从区分一个目标状态 或一个非目标状态。

这类搜索策略以节点扩展的顺序区分：有 深度优先搜索，宽度优先搜索，一致代价搜索。

如何评估/评价无信息搜索：

- 完备性：是否总能找到一个存在的解？
- 时间复杂性：花费多长时间找到这个解？
- 空间复杂性：需要多少内存？
- 最优性：是否总能找到最优的解？
- 时间复杂性和空间复杂性用如下术语来度量：
 - b -- maximum branching factor of the search tree.
搜索树的最大分支因子。
 - d -- depth of the shallowest solution.
最浅解的深度。
 - m -- maximum depth of the search tree.
搜索树的最大深度。

d 最浅解，就像有很多个解，最浅的那个。

BFS

```

function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node  $\leftarrow$  a node with STATE = problem.INITIAL-STATE
  PATH-TEST = 0
  frontier  $\leftarrow$  a FIFO queue with node as the only element
  explored  $\leftarrow$  an empty set
  loop do
    if EMPTY ? (frontier) then return failure
    node  $\leftarrow$  POP(frontier)      /* chooses the shallowest node in frontier */
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child  $\leftarrow$  CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
        frontier  $\leftarrow$  INSERT(child, frontier)

```

Properties of Breadth-first Search 宽度优先搜索的性质

Time complexity 时间复杂性 $b + b^2 + b^3 + \dots + b^d = O(b^d)$

Space complexity 空间复杂性 $O(b^d)$

一致代价搜索 (Uniform cost Search)

与BFS几乎一致，用队列实现：路径代价排序，低的优先。

```

function UNIFORM-FIRST-SEARCH(problem) returns a solution, or failure
  node  $\leftarrow$  a node with STATE = problem.INITIAL-STATE, PATH-TEST = 0
  frontier  $\leftarrow$  a priority queue ordered by PATH-COST, with node as the only element
  explored  $\leftarrow$  an empty set
  loop do
    if EMPTY ? (frontier) then return failure
    node  $\leftarrow$  POP(frontier)      /* chooses the lowest-cost node in frontier */
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child  $\leftarrow$  CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        frontier  $\leftarrow$  INSERT(child, frontier)
      else if child.STATE is in frontier with higher PATH-COST then
        replace that frontier node with child

```

注意将其孩子加入队列(*frontier*)时，如果有一个已经在队列里面，并且有更高的Cost，则需要替换。

Properties of Uniform-cost Search 一致代价搜索的特性

Time complexity $O(b^{1 + \lfloor C^*/\epsilon \rfloor})$

时间复杂性

Space complexity $O(b^{1 + \lfloor C^*/\epsilon \rfloor})$

空间复杂性

where

■ b -- the branching factor

分支因子

■ C^* -- the cost of the optimal solution

最优解的代价

■ ϵ -- every action costs at least

至少每个动作的代价

(书P75) 与BFS的不同

- 目标检测 应用于结点被选择扩展时，而不是在结点生成的时候进行。因为第一个生成的目标结点可能在次优路径上。
- 如果边缘中的结点有更好的路径到达该结点，那么会引入一个测试。

结合CS221，要理解在哪儿/为什么UCS和BFS类似，和Dijkstra几乎一致。

DFS

利用栈。

Properties of Depth-first Search 深度优先搜索的特性

Time complexity $O(b^m)$

时间复杂性

Space complexity $O(bm)$

空间复杂性

where

■ b -- the branching factor

分支因子

■ m -- the maximum depth of any node

任一节点的最大深度

深度受限搜索 (Depth limited Search)

如果状态空间无限，DFS会失败。所以深度受限搜索预设了一个深度，在深度外的结点视为没有后继。

但是 解有很多个，最浅为 d ，受限深度 $l < d$ 则出现额外的不完备性；反之如果 $l > d$ ，不能保证解的完备性。

```

function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)
function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  if limit = 0 then return cutoff /* no solution */
  cutoff_occurred ?  $\leftarrow$  false
  for each action in problem.ACTIONS(node.STATE) do
    child  $\leftarrow$  CHILD-NODE(problem, node, action)
    result  $\leftarrow$  RECURSIVE-DLS(child, problem, limit - 1)
    if result = cutoff then cutoff_occurred ?  $\leftarrow$  true
    else if result  $\neq$  failure then return result
  if cutoff_occurred ? then return cutoff /* no solution */
  else return failure

```

迭代加深搜索 (Iterative Deepening Search)

将深度优先和宽度优先的优势相结合，逐步增加深度限制反复运行直到找到目标。

以深度优先搜索相同的顺序访问搜索树的节点，但先访问节点的累积顺序实际是宽度优先。

```

function ITERATIVE-DEEPENING-SEARCH (problem) returns a solution, or failure
  for depth = 0 to  $\infty$  do
    result  $\leftarrow$  DEPTH-LIMITED-SEARCH(problem, depth)
    if result  $\neq$  cutoff then return result

```

调用之前的受限深度优先，一层一层设置受限深度(所以累积顺序是宽度优先).

双向搜索 (Bidirectional Search)

它同时进行两个搜索：一个是从初始状态向前搜索，而另一个则从目标向后搜索。当两者在中间相遇时停止。一个从终点开始，一个从源点开始。

该方法可以通过一种剩余距离的启发式估计来导向。

有信息/启发式搜索 (Informed Search Strategies)

这类策略采用超出问题本身定义的、问题特有的知识，因此能够找到比无信息搜索更有效的解。

- 评价函数： $f(n)$ ，用于选择一个结点进行扩展.
- 启发函数： $h(n)$ ， f 的一个组成部分 .

最佳优先搜索 (Best-first Search)

树搜索，图搜索。基于一个评价函数 $f(n)$.

用 $f(n)$ 替代一致代价搜索中的队列。

启发函数 $h(n)$: 从结点 n 到目标状态的最低路径估计代价。

贪婪搜索

Greedy Search 贪婪搜索

Search Strategy 搜索策略

- Try to expand the node that is closest to the goal.

试图扩展最接近目标的节点。

Evaluation function 评价函数

$$f(n) = h(n)$$

- It evaluates nodes by using just the heuristic function.

它仅使用启发式函数对节点进行评价。

- $h(n)$ -- estimated cost from n to the closest goal.

$h(n)$ -- 从 n 到最接近目标的估计代价。

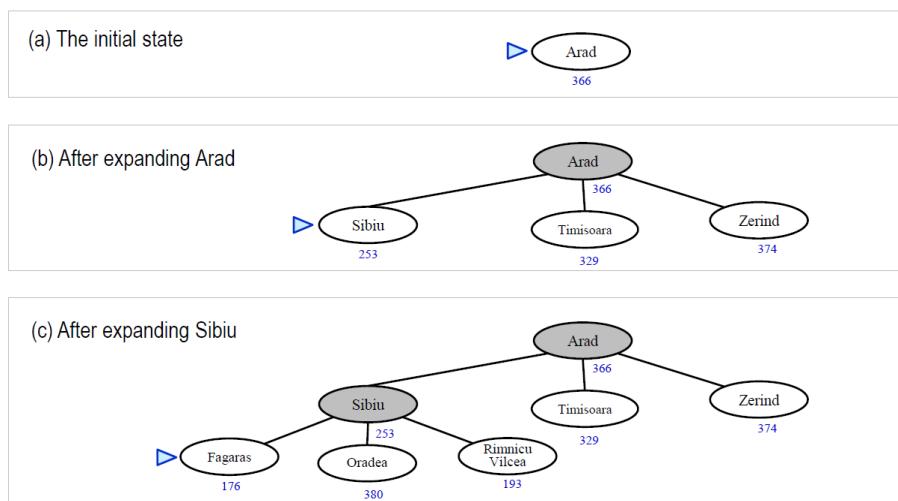
Why call "greedy" 为什么称为“贪婪”

- at each step it tries to get as close to the goal as it can.

每一步它都试图得到能够最接近目标的节点。

与浙大讲课一致。

Example: from Arad to Bucharest 举例：从Arad到Bucharest



h_{SLD} Values

Arad	366
Bucharest	0
Craiova	160
Drobeta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Properties of Greedy Tree Search 贪婪树搜索的特性

Worst-case time: $O(b^m)$

最差情况下的时间

Space complexity: $O(b^m)$

空间复杂性

where

- b -- the branching factor

分支因子

- m -- the maximum depth of the search space

搜索空间的最大深度

并非最优。

A* 搜索

最优的。

A* Search A*搜索

□ Search Strategy 搜索策略

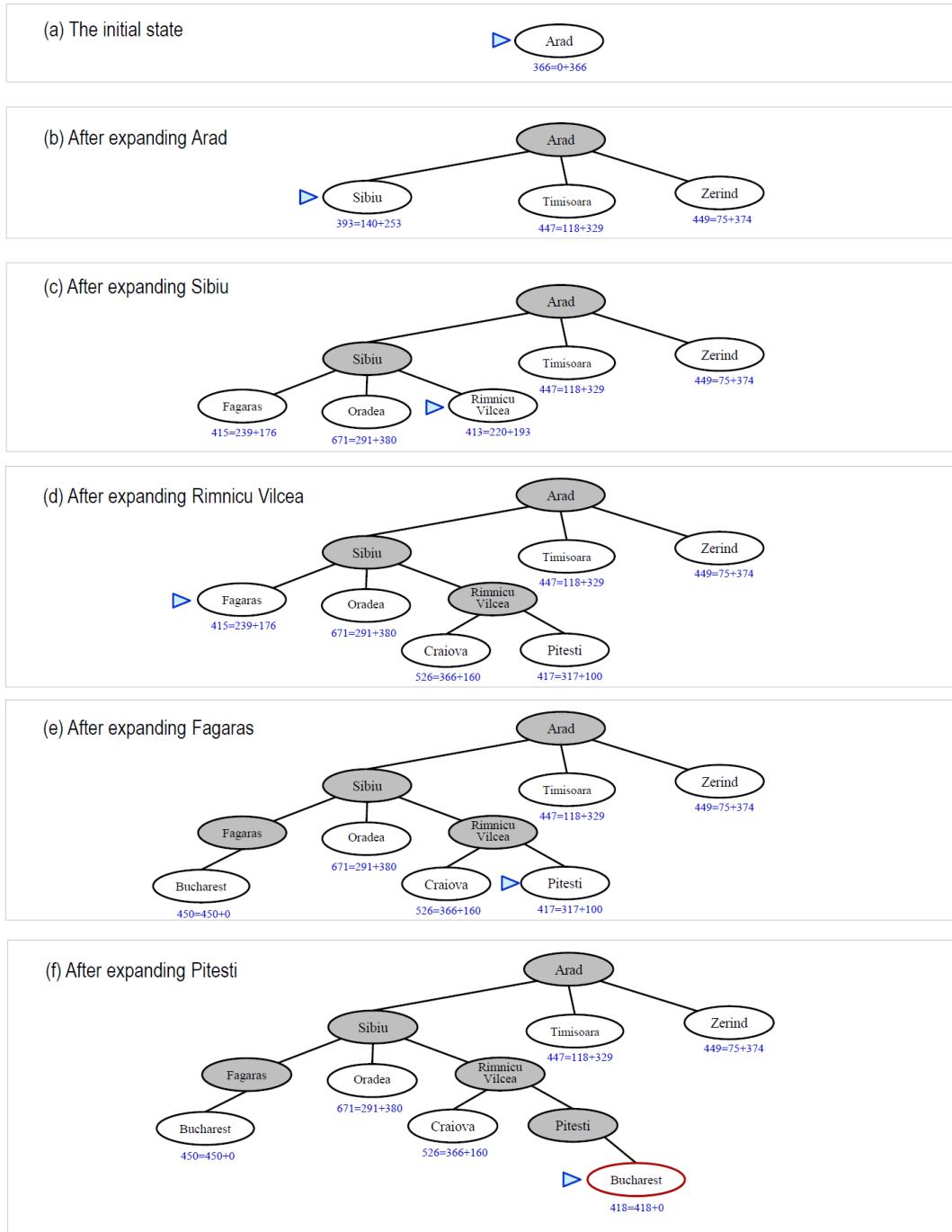
- avoid expanding expensive paths, minimizing the total estimated solution cost.
避免扩展代价高的路径，使总的估计求解代价最小化。

□ Evaluation function 评价函数

- $g(n)$ -- cost to reach the node $f(n) = g(n) + h(n)$
到达该节点的代价

- $h(n)$ -- estimated cost to get from the node to the goal
从该节点到目标的估计代价

Example: Form Arad to Bucharest 举例：从Arad到Bucharest



迭代加深 A* 搜索 (Iterative Deepening A* Search)

迭代加深搜索 (Iterative Deepening Search) 的变种。

它是深度优先的，内存使用低于 A*，但是不同于标准的迭代加深搜索，它集中于探索最有希望的节点，因此不会去搜索树任何处的同样深度。

Comparing Iterative Deepening Search 迭代加深搜索之比较

Iterative deepening depth-first search

- uses search depth as the cutoff for each iteration.

迭代加深深度优先搜索：使用搜索深度作为每次迭代的截止值。

Iterative Deepening A* Search

- uses the more informative evaluation function, i.e.

迭代加深A*搜索：使用信息更丰富的评价函数，即

$$f(n) = g(n) + h(n)$$

where

- $g(n)$ -- cost to reach the node
到达该节点的代价。
- $h(n)$ -- estimated cost to get from the node to the goal
该节点到目标的估计代价

启发式函数 (Heuristic Functions)

8-puzzle 的启发式函数

两种启发式函数，注意联想最短路问题中的启发式函数(要保证一个最优的界)：

- h_1 ：错位数码的个数。
- h_2 ：每个数码到目标位置的曼哈顿距离。

Search Cost 搜索代价

Search Cost (nodes generated)

d (depth)	Iterative Deepening Search	$A^*(h_1)$	$A^*(h_2)$
2	10	6	6
4	112	13	12
6	680	20	18
8	6384	39	25
10	47127	93	39
12	3644035	227	73
14	-	539	113
16	-	1301	211
18	-	3056	363
20	-	7276	676
22	-	18094	1219
24	-	39135	1641

- If $h_2(n) \geq h_1(n)$ for all n , then h_2 dominates h_1 and is better for search.

若对于所有的 n , $h_2(n) \geq h_1(n)$, 则 h_2 优于 h_1 , 因而 h_2 更适合搜索。

超越经典搜索

爬山法 (Hill Climbing)

是一种属于局部搜索家族的数学优化方法。贪婪搜索算法。

是一种迭代算法：开始时选择问题的一个任意解，然后递增地修改该解的一个元素，若得到一个更好的解，则将其修改作为新的解；重复直到无法找到进一步的改善。

不保持一棵搜索树。

```

function HILL-CLIMBING(problem) returns a state that is a local maximum
  persistent: current, a node
    neighbor, a node
  current  $\leftarrow$  MAKE-NODE(problem.INITIAL-STATE)
  loop do
    neighbor  $\leftarrow$  a successor of current
    if neighbor.VALUE  $\leq$  current.VALUE then return current.STATE
    current  $\leftarrow$  neighbor

```

当前每一步都用具有最高值的邻居替换，直到到达一个峰值。

例子: n 皇后问题

局部搜索通常采用完整状态形式化。

n 个皇后放在棋盘上，每次移动一个皇后来减少冲突。直到最后他们都没有冲突。

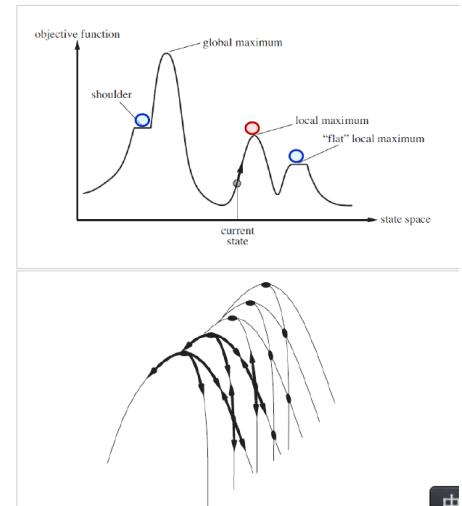
爬山法弱点

Weaknesses of Hill-Climbing 爬山法的弱点

- It often gets stuck for the three reasons:

它在如下三种情况下经常被困：

- Local maxima 局部最大值
higher than its neighbors but lower than global maximum.
高于相邻节点但低于全局最大值。
- Plateaux 高原
can be a flat local maximum, or a shoulder.
可能是一个平坦的局部最大值，或山肩。
- Ridges 山岭
result in a sequence of local maxima that is very difficult to navigate.
结果是一系列局部最大值，非常难爬行。



局部束搜索 (Local Beam Search)

- It begins with k randomly generated states.
- At each step, all the successors of all k states are generated.
- If any one is a goal, the algorithm halts, else it selects the k best successors from the complete list, and repeats.

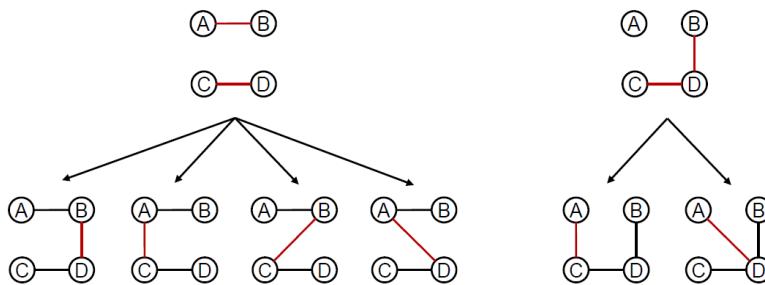
- 初始 k 个随机状态。
- 每次迭代中 k 个状态的所有后继状态生成，如果有目标则返回，否则在这些后继中取出 k 个最优的，继续迭代。

有用的信息能够在并行搜索线程间传递。

例子: 旅行商问题 TSP

哈密顿。最开始先随机两个状态：

保持 k 个状态而不仅仅为1。从 k 个随机生成的状态开始。本例中 $k=2$ 。



Generate all successors of all the k states. None of these is a goal state so we continue.
生成所有 k 个状态的全部后继节点。这些后继节点中没有目标状态，故继续下一步。

上面这样已经找到目标了。

变型方法 - 随机束搜索：选择后继节点的概率是其值的递增函数，来随机地选择 k 个后继节点。

禁忌搜索 (Tabu Search)

meta heuristic algorithm, used for solving combinatorial optimization problems.

它使用一种局部搜索或邻域搜索过程，从一个潜在的解 x 到改进的相邻解 x' 之间反复移动，直到满足某些停止条件。

The memory structure to determine the solutions is called tabu list.

用于确定解的数据结构被称为禁忌表。

Three Strategies of Tabu Search 禁忌搜索的三种策略

Forbidding strategy 禁止策略

control what enters the tabu list.

控制何物进入该禁忌表。

Freeing strategy 释放策略

control what exits the tabu list and when.

控制何物以及何时退出该禁忌表。

Short-term strategy 短期策略

manage interplay between the forbidding strategy and freeing strategy to select trial solutions.

管理禁止策略和释放策略之间的相互作用来选择试验解。

```

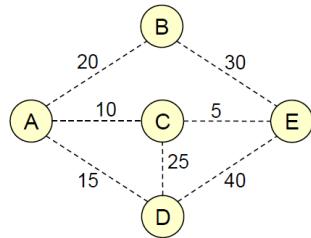
function TABU-SEARCH ( $s'$ ) return a best candidate
   $s_{Best} \leftarrow s \leftarrow s'$ 
   $tabuList \leftarrow$  null list
  while (not STOPPING-CONDITION())
     $candidateList \leftarrow$  null list
     $bestCandidate \leftarrow$  null
    for ( $sCandidate$  in  $sNeighborhood$ )
      if ((not  $tabuList$ .CONTAINS( $sCandidate$ ))
        and (FITNESS( $sCandidate$ ) > FITNESS( $bestCandidate$ )))
        then  $bestCandidate \leftarrow sCandidate$ 
     $s \leftarrow bestCandidate$ 
    if (FITNESS( $bestCandidate$ ) > FITNESS( $s_{Best}$ )) then  $s_{Best} \leftarrow bestCandidate$ 
     $tabuList.PUSH(bestCandidate)$ 
    if ( $tabuList.SIZE > maxTabuSize$ ) then  $tabuList.REMOVE-FIRST()$ 
  return  $s_{Best}$ 

```

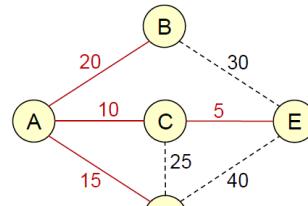
例子: 最小生成树问题

MST生成是加了一些额外的约束条件：

□ Objective 目标



Connects all nodes with minimum cost
用最小代价连接所有节点



An optimal solution without constraints
一个无约束的最优解

Constraints 1: Link AD can be included only if link DE also is included. (Penalty:100)
约束1：仅当包含连接DE时，才可以包含连接AD。（处罚：100）

Constraints 2: At most one of the three links (AD, CD, and AB) can be included. (Penalty: 100 if selected two of the three, 200 if selected all three.)
约束2：至多可以包含三个连接 (AD, CD和AB) 中的一个。（处罚：若选择了三个中的两个则处罚100，选择了全部三个则罚200）

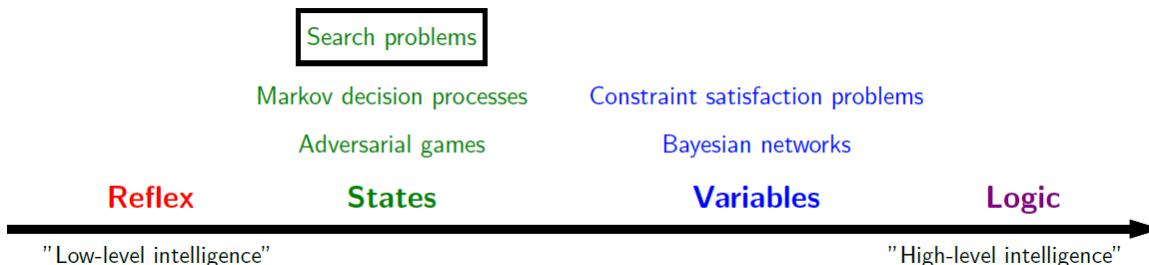
这个看得不是很懂，PPT (4.2.3)--课件3: TabuSearch.pdf 里有MST禁忌搜索的每一步。

StanFord CS221: 人工智能原理与技术

P5 5. Lecture 5 - Search 1 - Dynamic Program, Uniform Cost Search

state-based model

农夫过河，还有白菜 山羊 狼，船只能容纳两个。问有多少种方案？重点在如何解决。



- Model/Inference/Learning

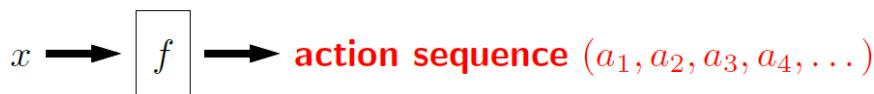
这种基于状态的机器翻译也可以有一个应用：
翻译加一个单词就是Action，目标状态就是流利准确。

与 Reflex-based 模型的对比:

Classifier (reflex-based models):



Search problem (state-based models):



Key: need to consider future consequences of an action!

输出是一系列动作.

三种算法用于推断搜索问题: Tree Search, DP, Uniform Cost Search.

树搜索: 以农夫过河为例

- 可能的动作:



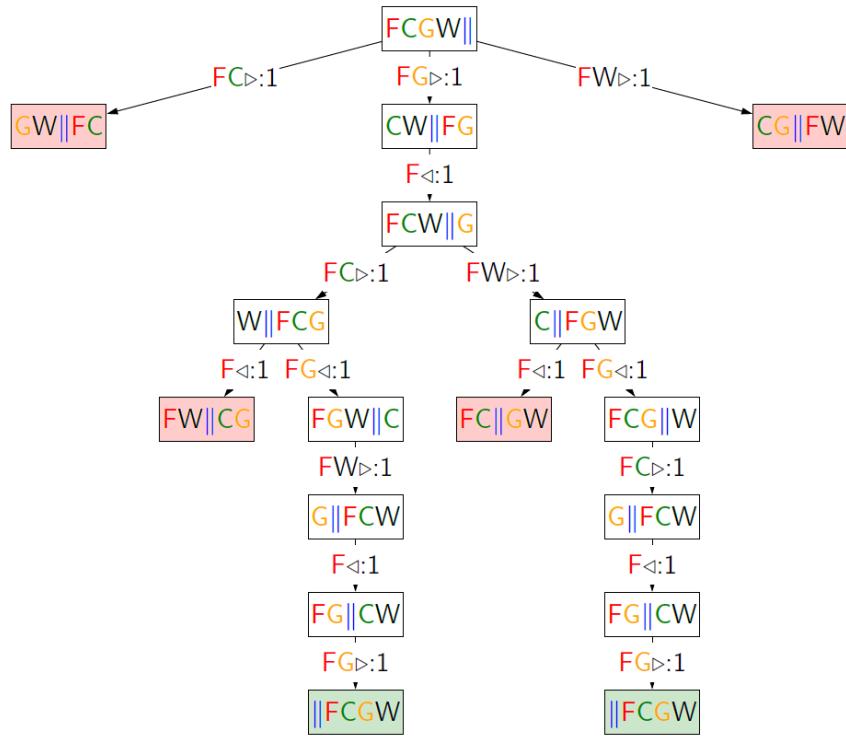
Farmer Cabbage Goat Wolf

Actions:

$F \triangleright$	$F \triangleleft$
$FC \triangleright$	$FC \triangleleft$
$FG \triangleright$	$FG \triangleleft$
$FW \triangleright$	$FW \triangleleft$

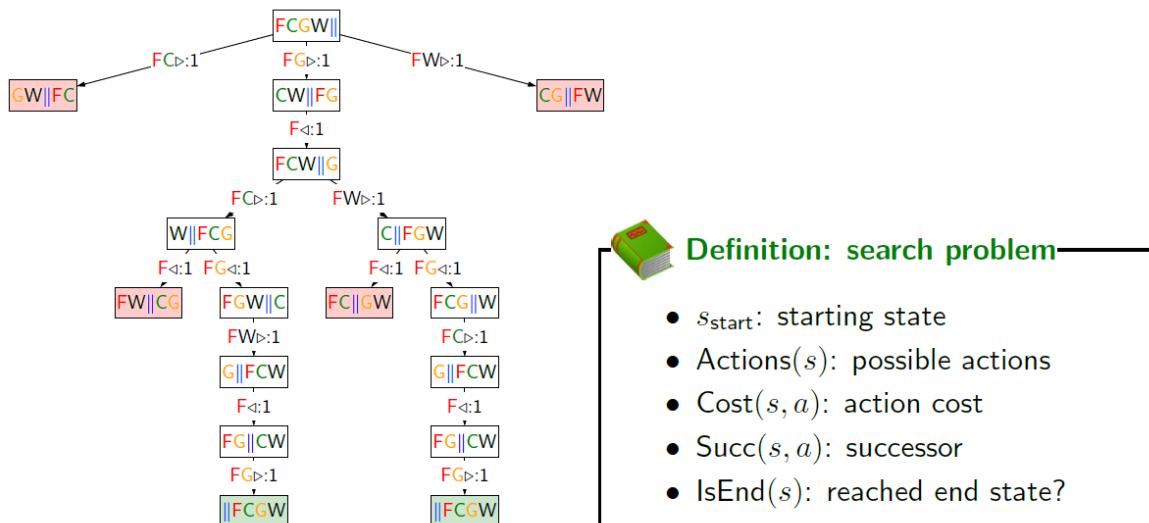
Approach: build a **search tree** ("what if?")

- 构建状态搜索树:



红色的就是不允许的点。会被吃。

Search problem



例子: 运输问题

从2到3，可以走路花费1分钟；从2到6可以坐车花费两分钟；如此构成的运输问题，从a到b运输消耗。

从回溯搜索开始，这是一种可以尝试所有路径的最简单算法。算法递归地调用当前状态和通往该状态的路径。回溯搜索执行搜索树的深度优先遍历。

搜索树的深度可能没有一个有限的上界。在这种情况下，有两种选择：

- 我们可以简单地限制最大深度并在某个点之后放弃
- 我们可以不允许访问同一状态
- 这里的DFS是假设每条边的成本都是0的，因为只要找到了目的地就OK，找到一个可行解就OK，并不在乎离源点的远近。

Idea: Backtracking search + stop when find the first end state.

If b actions per state, maximum depth is D actions:

- Space: still $O(D)$
- Time: still $O(b^D)$ worst case, but could be much better if solutions are easy to find

BFS:

Idea: explore all nodes in order of increasing depth.

Legend: b actions per state, solution has d actions

- Space: now $O(b^d)$ (a lot worse!)
- Time: $O(b^d)$ (better, depends on d , not D)

DFS_ID:

DFS with iterative deepening



Assumption: constant action costs

Assume action costs $\text{Cost}(s, a) = c$ for some $c \geq 0$.

Idea:

- Modify DFS to stop at a maximum depth.
- Call DFS for maximum depths 1, 2, ...

DFS on d asks: is there a solution with d actions?

Legend: b actions per state, solution size d

- Space: $O(d)$ (saved!)
- Time: $O(b^d)$ (same as BFS)

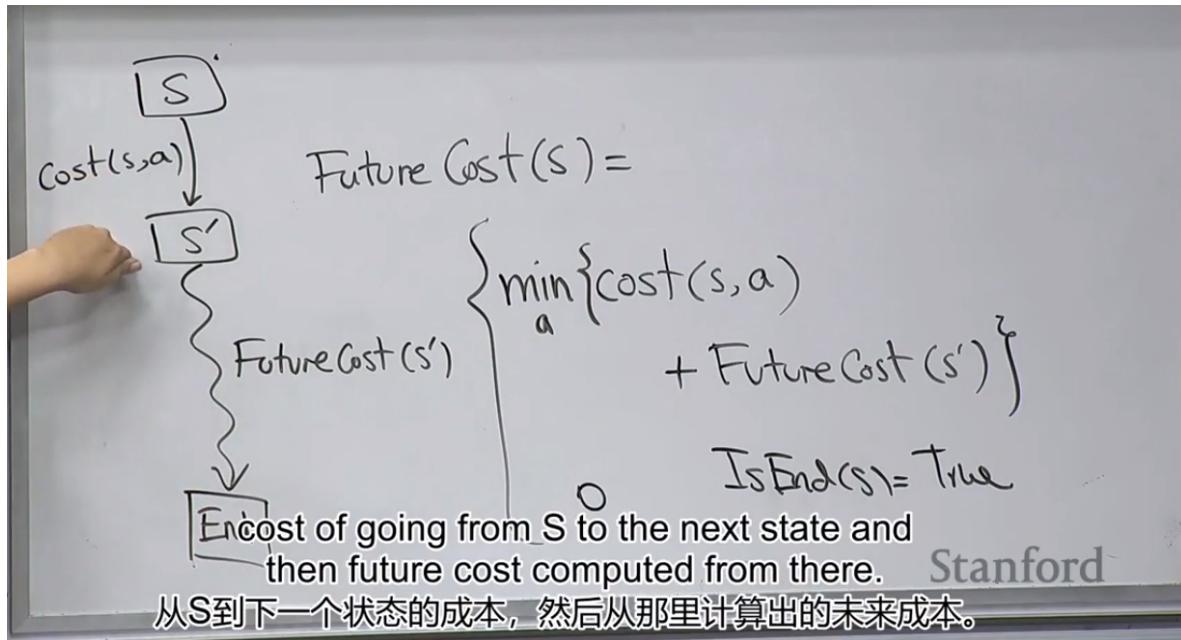
设定一个DFS最大深度，然后就在这个深度上找，一层一层这样，就像拉着狗在一定长度范围找，找不到松开/放长一点再找。

Legend: b actions/state, solution depth d , maximum depth D

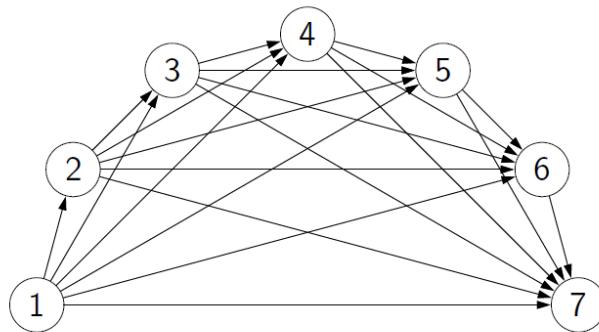
Algorithm	Action costs	Space	Time
Backtracking	any	$O(D)$	$O(b^D)$
DFS	zero	$O(D)$	$O(b^D)$
BFS	constant ≥ 0	$O(b^d)$	$O(b^d)$
DFS-ID	constant ≥ 0	$O(d)$	$O(b^d)$

- Always exponential time
- Avoid exponential space with DFS-ID

DP:



State: past sequence of actions current city



Exponential saving in time and space!

仅当前状态决定下一状态. 这也是一些开销节省的原因.

每个状态计算的值被记录下来, 在一个状态被计算的时候需要保证前序所有状态都已计算完成. 所以无环.

DP 加路径约束

比如不能连续访问三个奇数号码的城市.

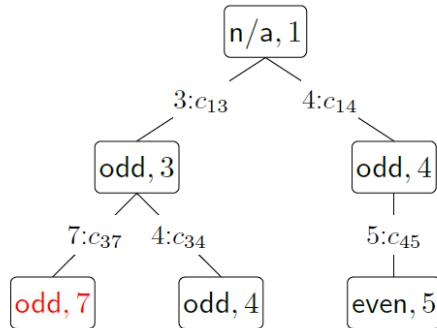


Example: route finding

Find the minimum cost path from city 1 to city n , only moving forward. It costs c_{ij} to go from i to j .

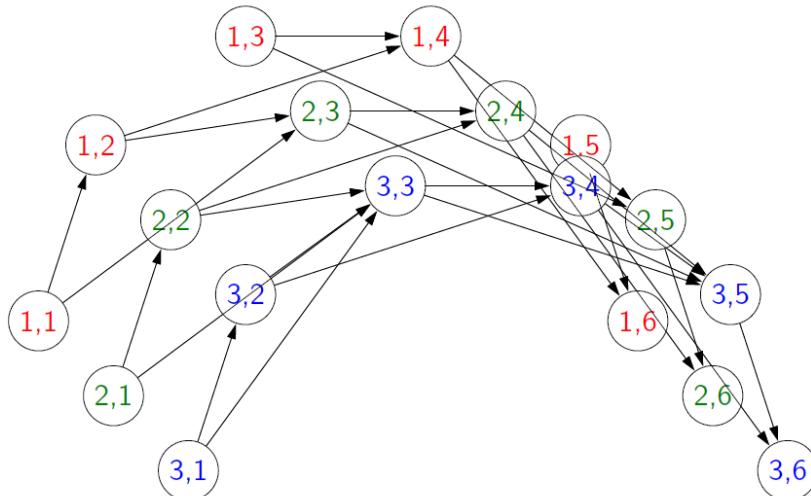
Constraint: Can't visit three odd cities in a row.

State: (whether previous city was odd, current city)



比如要求通过3个奇数号码的城市. 那么就需要把当前路径通过了几个(还剩几个)奇数号码城市记录下来, 当做一个状态, 随着一起更新.

State: (min(number of odd cities visited, 3), current city)



Uniform Cost Search

很像 Dijkstra.



Key idea: state ordering

UCS enumerates states in order of increasing past cost.



Assumption: non-negativity

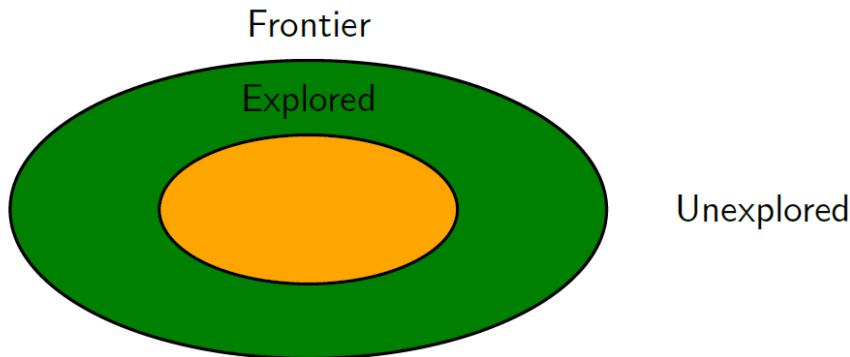
All action costs are non-negative: $\text{Cost}(s, a) \geq 0$.

思想：根据过去代价的增加顺序来计算过去的代价。

为了提高效率，假设所有的行动成本都是无影响。UCS在逻辑上就是Dijkstra算法，两者实际上是等价的。有一个重要的实现差异：UCS以搜索问题作为输入，它隐式地定义了一个巨大的甚至无限的图，而Dijkstra算法以一个完全具体的图作为输入。

另一个区别是Dijkstra找到从一开始状态到其他节点最短路径，而UCS是显式地寻找最短路径结束状态。

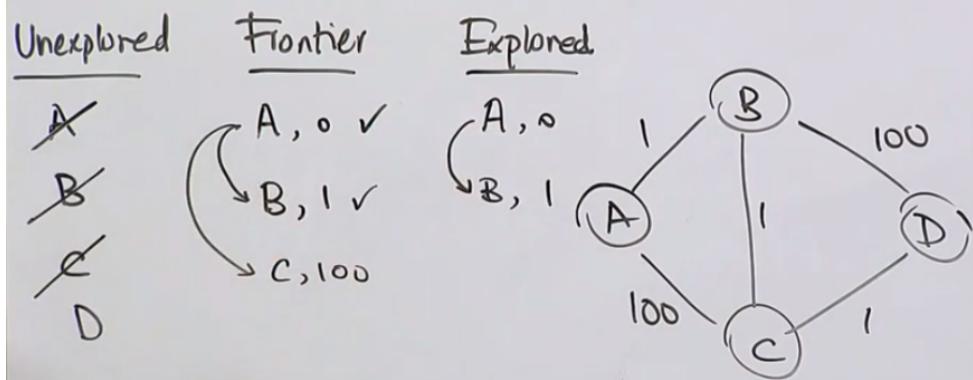
High-level strategy



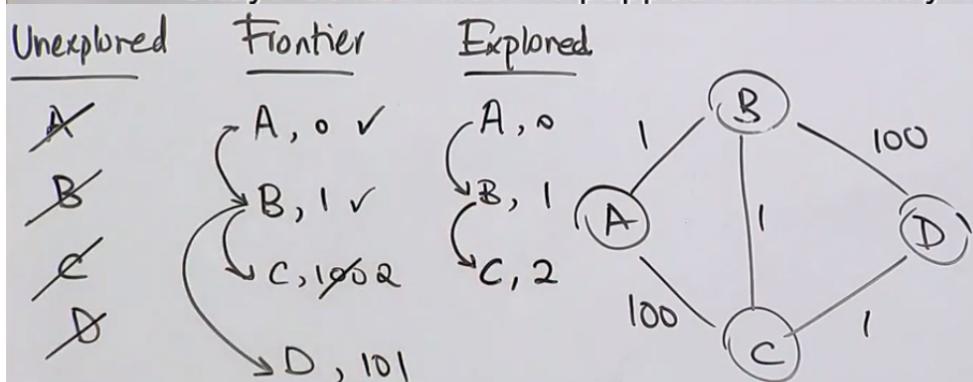
- **Explored:** states we've found the optimal path to
- **Frontier:** states we've seen, still figuring out how to get there cheaply
- **Unexplored:** states we haven't seen

跟踪三个set: **Explored**(知道到达这里的最佳状态/路径), **Frontier**(知道可以到达, 但不懂是不是最佳的), **Unexplored**.

过程：

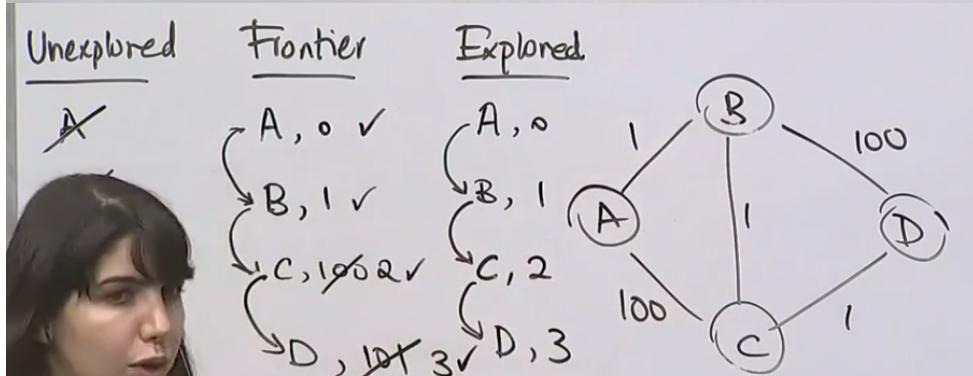


Okay? So now that I've popped off B from my frontier, Stanford



So we're done with C. And then,

Stanford



So the way to get from A to D is- is by taking this route, and it costs 1.

P6 6.Lecture 6 Search 2 - A

上堂课的一点尾巴：在路径探索问题中， $c_{ij} > 0$ ，计算从一个点到另一个点的路径(可能有多个解)：

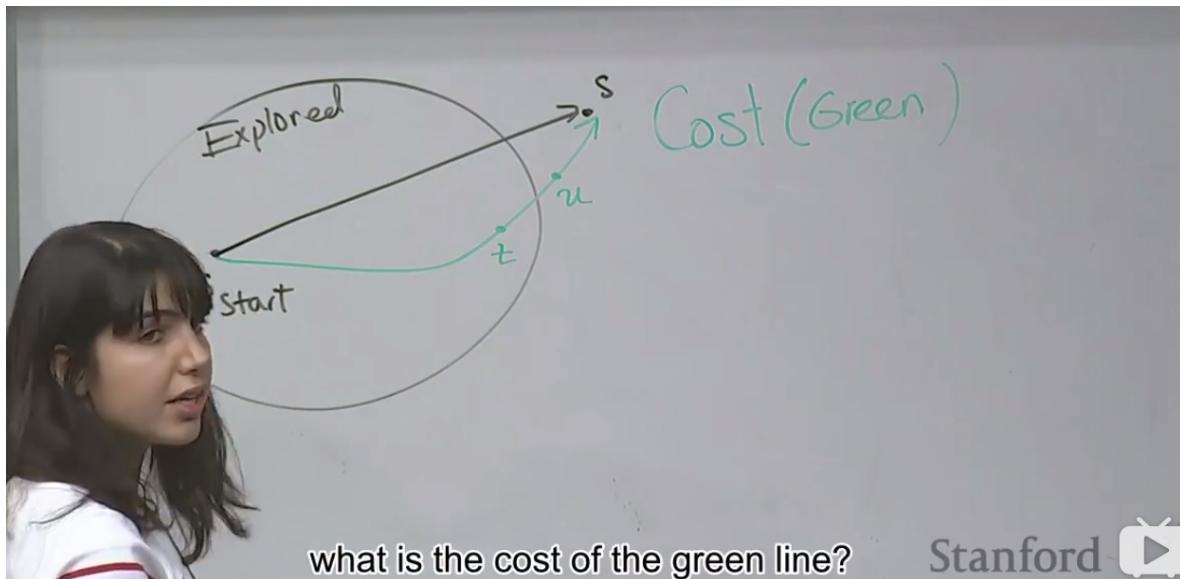
- 不能使用dfs，因为dfs只是找到一个可行解。
- 不能使用bfs，因为bfs希望的是每条边长度一致。
- 可以使用DP，记录状态的时候同时记录在哪个城市和当前行进方向。
- 可以使用Uniform Cost Search. 毕竟它和Dijkstra的区别其实只是Dijkstra试图探索所有状态，但是UCS是探索部分状态是一个搜索问题。



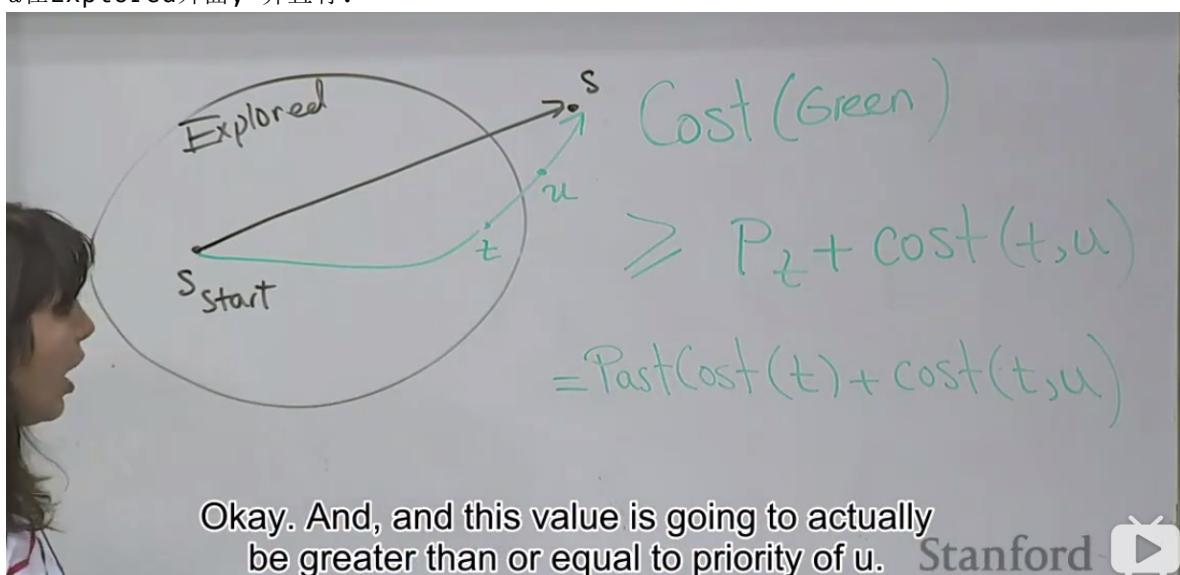
Key idea: state

A **state** is a summary of all the past actions sufficient to choose future actions **optimally**.

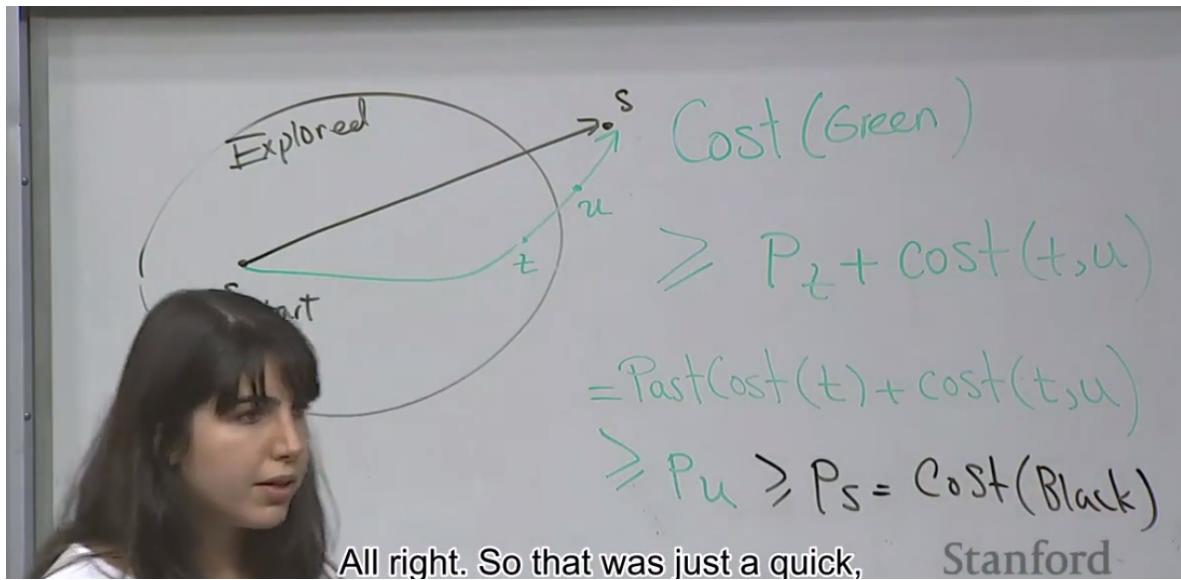
UCS最优的证明



如果源点到 s 还有一条路径 并且其上有两个状态 t, u , u, s 在Frontier内, 绿色路径上存在一个状态 u 在Explored外面, 并且有:



其中 $\text{PastCost}(t)$ 是当前算法更新过程上的 $\text{start} \rightarrow s$.



以上 P_u 为结点 u 的优先级。

N total states, n of which are closer than end state

Algorithm	Cycles?	Action costs	Time/space
DP	no	any	$O(N)$
UCS	yes	≥ 0	$O(n \log n)$

Note: UCS potentially explores fewer states, but requires more overhead to maintain the priority queue

Note: assume number of actions per state is constant (independent of n and N)

Learning: 再次以运输问题为例

Transportation example

Start state: 1

Walk action: from s to $s + 1$ (cost: 1)

Tram action: from s to $2s$ (cost: 2)

End state: n

↓ search algorithm

walk walk tram tram tram walk tram tram
(minimum cost path)

但是现在还有一种任务是：给定一个最优动作序列，求当前环境下动作的代价。这就是Learning：

Forward problem (search):

$$\text{Cost}(s, a) \longrightarrow (a_1, \dots, a_k)$$

Inverse problem (learning):

$$(a_1, \dots, a_k) \longrightarrow \text{Cost}(s, a)$$

input s : search prob. w/o costs
output y : sol'n path

y : walk, walk, walk
 y' : walk, tram

Okay? So now what we wanna do is you want to update our forward

好的？所以现在我们想做的是您想更新我们的

- 先随机初始化 Cost 的值，然后看Learning的过程如何更新。
- 绿色就是其作出的预测，是因为 $3 + 2 < 3 + 3 + 3$ ，所以会选择 walk, tram 的路径。
- 根据绿色 y' 和正确的动作序列 y 做对比，并更新：

$w[a_1] = w[\text{walk}] = 3 \xrightarrow{\text{green}} 2 \xrightarrow{\text{green}} 1 \xrightarrow{\text{green}} 0 \xrightarrow{\text{green}} 1$
 $w[a_2] = w[\text{tram}] = 2 \xrightarrow{\text{green}} 3$

input s : search prob. w/o costs
output y : sol'n path
 y : walk, walk, walk
 y' : walk, walk, tram

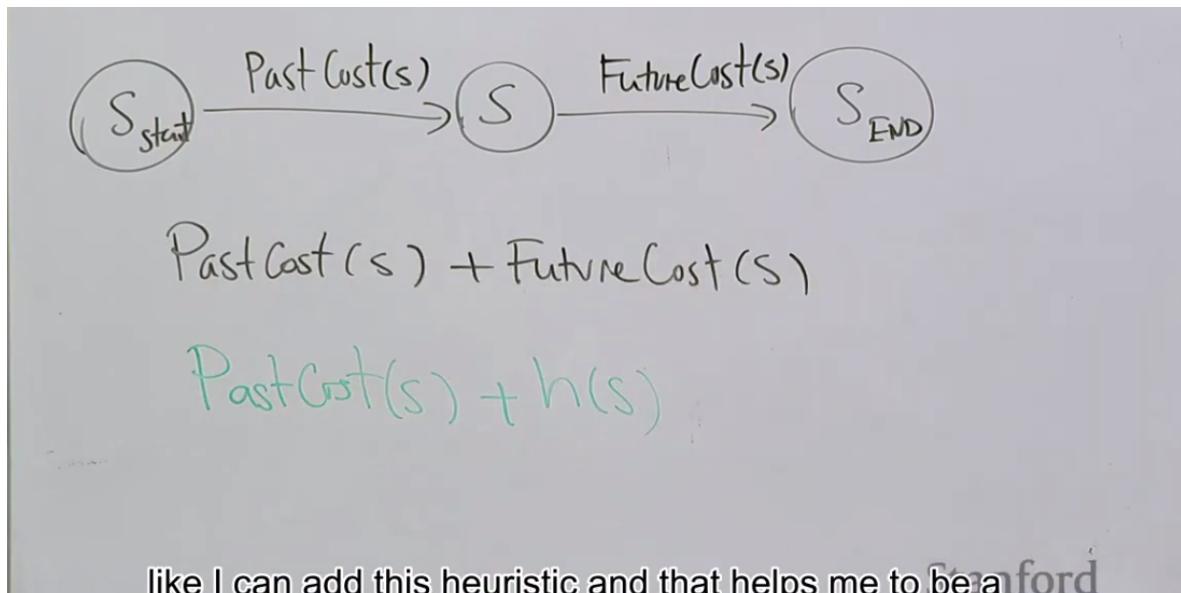
The weight of walk just became 1 and then the weight of tram just became 3.

步行的重量变为1，然后电车的重量变为3。

当然这是一个非常简化的版本。权重仅依赖动作。

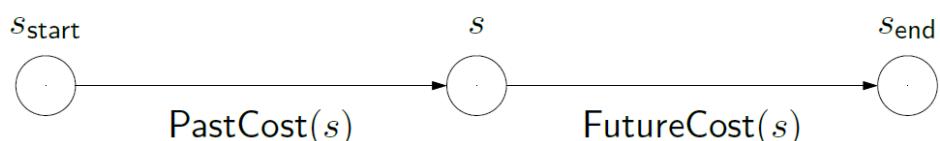
A*

理解启发函数，为什么能更好找到目标？



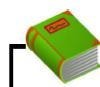
现在绿色的做了一点改变，引入了启发式函数：对未来成本的估计。

UCS: explore states in order of PastCost(s)



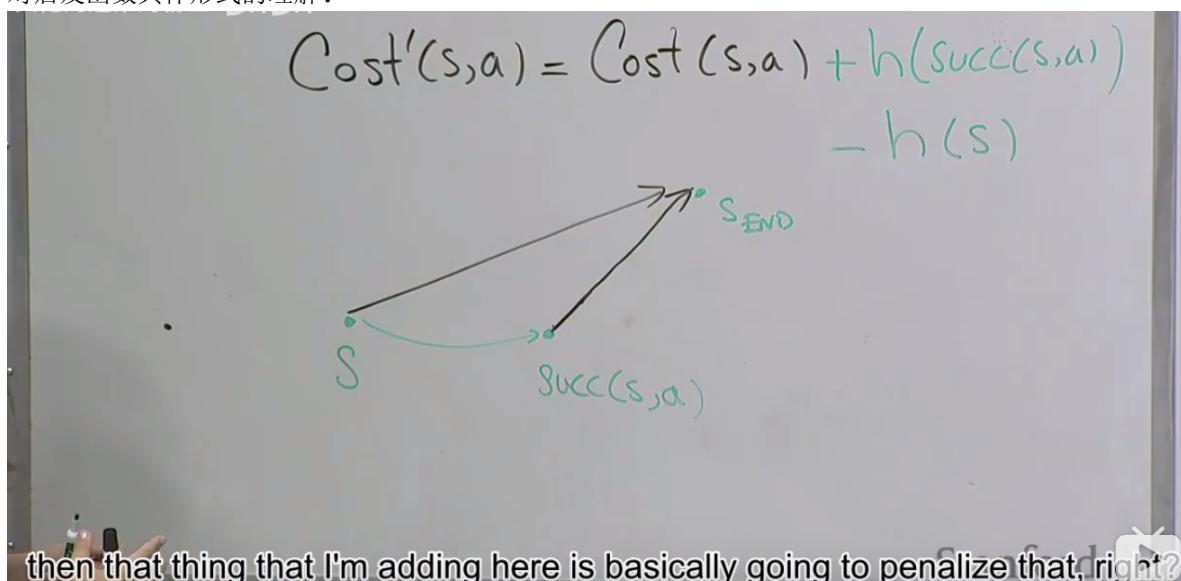
Ideal: explore in order of PastCost(s) + FutureCost(s)

A*: explore in order of PastCost(s) + $h(s)$

 **Definition: Heuristic function**

A heuristic $h(s)$ is any estimate of FutureCost(s).

对启发函数具体形式的理解：



then that thing that I'm adding here is basically going to penalize that, right?

$h(B) - h(C) \iff (\text{succ}(S, a) \rightarrow S_{\text{END}}) - (S \rightarrow S_{\text{END}})$, 请注意这里直观的理解就是: 如果 $\text{succ}(S, a)$ 后继, 接下来的那个结点, 没有往 S_{END} 那个方向引导, 那么很自然地就通过 $h(B) - h(C)$ 给一个惩罚.



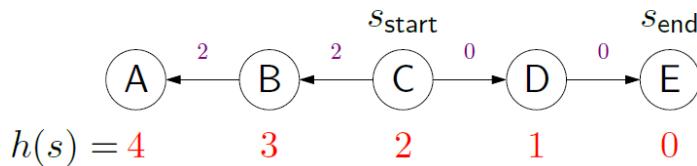
Algorithm: A* search [Hart/Nilsson/Raphael, 1968]

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

Example:



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

如上初始Cost是每两个结点之间距离为1. $h(s)$ 启发函数是到E的直线距离, 需要求的是 $C \rightarrow E$.

根据刚才说的惩罚, $\text{Cost}'(C, B)$ 被更新了(如上2 2 0 0的距离).

在更新的Cost基础上进行UCS, 显然更加容易到达终点E.

启发函数特性: Consistency

在最短路问题中, 启发式函数所应具有的特性: **consistency**, 需要满足:

- 三角不等式.
- 更新后的Cost应该是非负的
- 目标结点的 h 函数值为0.

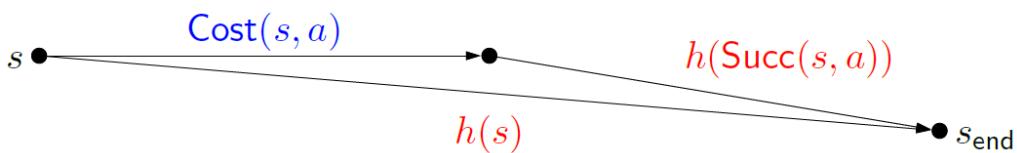


Definition: consistency

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.

Consistency \Rightarrow Correctness:

定理：满足了上面的性质，A*将是正确的

- A* is correct
if h is consistent

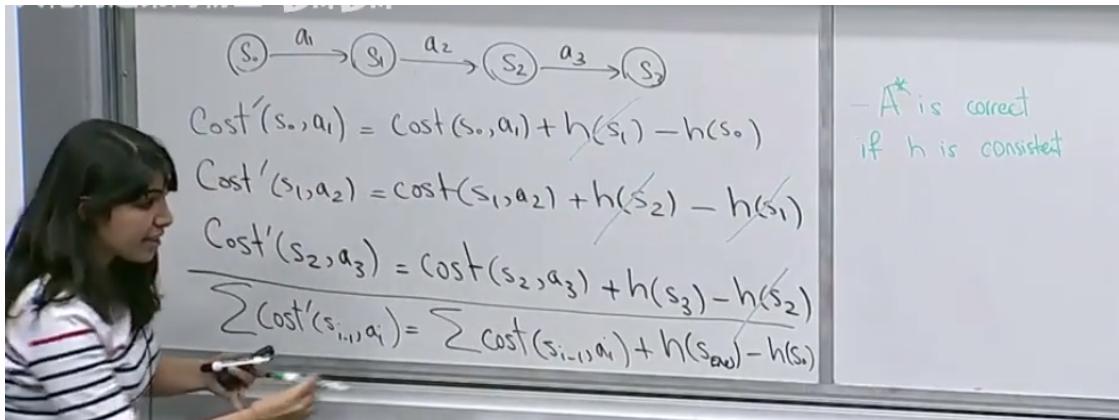


Proposition: correctness

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

- 上述定理一个较水的证明：

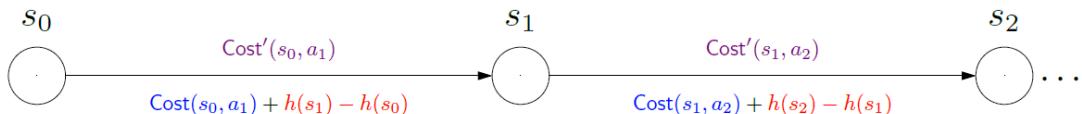
将所有更新后的Cost'累加。



所以更新后的Cost'累加起来就是 原来所有Cost累加后减去一个常数($h(S_0)$)

所以优化原问题(原Cost)上，等价于更新后Cost'上的。因此UCS将返回最佳方案。

- Consider any path $[s_0, a_1, s_1, \dots, a_L, s_L]$:



- Key identity:

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

- Therefore, A* (finding the minimum cost path using modified costs) solves the original problem (even though edge costs are all different!)

注意到UCS的性质：

Theorem: efficiency of UCS

UCS explores all states s satisfying
 $\text{PastCost}(s) \leq \text{PastCost}(s_{\text{end}})$

最小路径树的那个性质嘛。

所以对于A*有：

Theorem: efficiency of A*

A* explores all states s satisfying
 $\text{PastCost}(s) \leq \text{PastCost}(s_{\text{end}}) - h(s)$

Interpretation: the larger $h(s)$, the better

Proof: A* explores all s such that

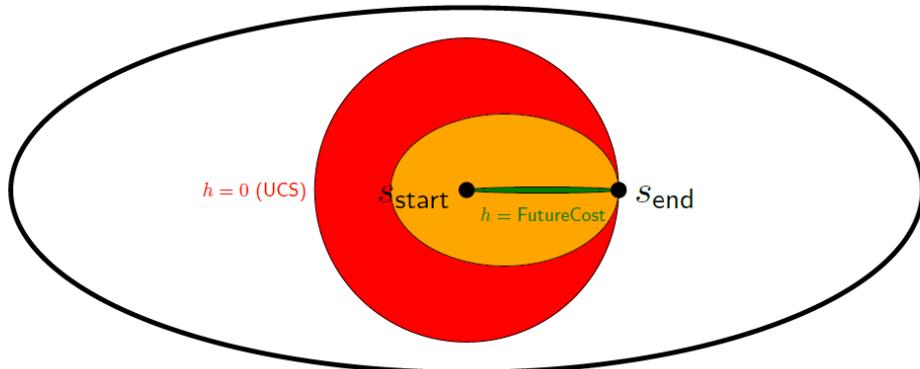
$$\text{PastCost}(s) + h(s)$$

\leq

$$\text{PastCost}(s_{\text{end}})$$

很明显看到，不等号右边变小，搜索缩小范围，启发的意义。

直接考虑FutureCost(直线距离)的搜索如下绿色，A*更想橙色区域的搜索，红色是UCS：

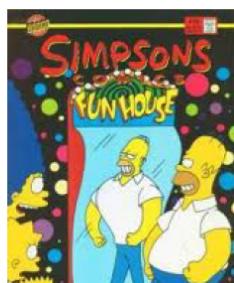


- If $h(s) = 0$, then A* is same as UCS.
- If $h(s) = \text{FutureCost}(s)$, then A* only explores nodes on a minimum cost path.
- Usually $h(s)$ is somewhere in between.

所以A*的关键是扭曲。扭曲距离：

Key idea: distortion

A* distorts edge costs to favor end states.



启发函数特性: Admissibility

A*还有一个特性就是 Admissibility，启发函数值应该是FutureCost的估计：

Admissibility



Definition: admissibility

A heuristic $h(s)$ is admissible if
$$h(s) \leq \text{FutureCost}(s)$$

Intuition: admissible heuristics are optimistic



Theorem: consistency implies admissibility

If a heuristic $h(s)$ is **consistent**, then $h(s)$ is **admissible**.

Proof: use induction on FutureCost(s)

如何寻找到一个好的启发函数?

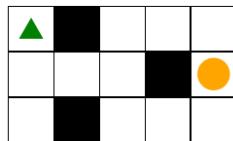
Relaxation, 例子:

Closed form solution

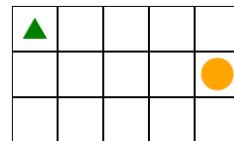


Example: knock down walls

Goal: move from triangle to circle



Hard



Easy

Heuristic:

$$h(s) = \text{ManhattanDistance}(s, (2, 5))$$

e.g., $h((1, 1)) = 5$

需要的三角形到圆形的路径，启发函数应为啥？ Relaxation：原本需要不能碰到墙，现在估计的是未来最优的界限，直接先放松到把这些墙拿走。然后用曼哈顿距离。

这样做可能是小于FutureCost的，就像上面Admissibility性质所说的那样，这样是最好的最乐观的一个估计。这是一个近似值。