机器学习与人工智能

Zhuoyang Liu

May 2024

1 全局搜索

1.1 问题的定义以及问题的解

1、问题描述模型——一个问题的定义包含五个部分：

1）初始状态 S_0

2）可选动作 A：给定一个状态 s，ACTION(s) 返回一组可能得动作

3）状态转移模型 T：在状态 s 下执行动作 a 之后所到达的状态用 s' =RESULT(s,a) 表示，一个状态经过一个动作后来到的下一个状态我们称之为后继状态 s'。初始状态、动作、状态转移模型构成了状态空间。状态空间构成一幅有向图。路径是从一个状态出发通过一系列动作所经过的状态序列。

4）目标状态 G

5）路径花费 C：每条路径可以有一个花费，用来度量解的好坏

2、问题的建模方式决定的问题的复杂度，如何建模比如何搜索更重要

1.2 通过搜索对问题求解

1.2.1 基本概念

1）搜索树

2）父节点 parent node，子节点 child node，叶节点 leaf node

3）开节点集,frontier, 或称 open list

子节点还没有长出来的点的集合称为开节点集

4）闭节点集,closed list, 或称 explored set

子节点已经完全长出来（完全探索过）的点的集合称为闭节点集

5）搜索策略 search strategy

关键点：分清开节点集和闭节点集！如何从开节点集中挑选下一个要搜索的节点?!

1.2.2 树搜索和图搜索

区别在于，树搜索不需要记住已经搜过的点集，图搜索需要标记哪些点集已经搜索过了

1）树搜索：

function TREE-SEARCH(*problem*) **returns** a solution, or failure

initialize the frontier using the initial state of the *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

2）图搜索：

function GRAPH-SEARCH(*problem*) **returns** a solution, or failure

initialize the frontier using the initial state of *problem*

initialize the explored set 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** then corresponding solution

add the node to the explored set

expand the chosen node, adding the resulting nodes to the frontier

only if not in the frontier or explored set

1.2.3 问题难度和搜索复杂度

1）问题的难度可以用两种方法衡量：图规模和搜索树规模

a、图规模：利用状态空间图的大小衡量问题规模：$|V|+|E|, v$ 是点数，E 是边数

b、搜索树的规模：用分支数和最浅目标状态所在衡量 2）算法复杂度

时间复杂度通常用搜索树展开的节点数目表示

空间复杂度通常用需要存储的最大节点数目来估计

1.3 无信息搜索（盲目搜索）

宽度优先，深度优先，效率比较，一致代价

1.3.1 宽度优先 (Breadth First Search，BFS)

1、复杂度

时间复杂度：$b+b^2+b^3+...+b^d=O(b^d)$，访问过的节点

空间复杂度：$O(b^d)$

2、实现

宽度优先算法用队列 **queue** 实现（先进先出）

宽度优先的图搜索：Breadth-first search on a graph

第一次遇到目标状态就可以宣布搜索结束

关键点仅在于谁先从开节点集出来，不同的出来顺序用不同的数据结构实现

3、问题

内存的需求远远大于运行的时间问题

时间问题仍然是一个大问题（指数级别的爆炸增长）

1.3.2 深度优先 (Depth-First Search, DFS)

1、实现

使用先进后出 LIFO 的栈 **stack**，存下一个待展开的节点

使用递归调用（recursive function）的方法

2、优势

· 如果是树搜索，对于一个分支数为 b ，最大深度为 m 的搜索空间, 深度优先搜索的开节点集空间是 $O(bm)$ 每个节点存了 b 个动作。而宽度优先的开节点集是 $O(bd)$，在内存一样的情况下，深度优先可以搜得更深

· 深度优先搜索因为可以搜的更深，所以是许多 AI 领域工作的基础性算法

3、宽度优先和深度优先的效率分析

搜索算法	宽度优先	深度优先
是否完备?	是	否
时间复杂度	$O(b^d)$	$O(b^m)$
空间复杂度	$O(b^d)$	$O(bm)$
最优的吗?	是	否

· b 是分支数，d 是最浅目标所在的深度

· m 是搜索树的最大深度

· 宽度优先的完备性的前提是 b 是有限的

关键：深度优先更省空间，搜的更深

1.3.3 深度受限

DFS 因为没有存储搜索过的节点，所有可能陷入死循环

限定搜索的最大深度 L，深度为 L 的节点会被视为没有后继节点的叶节点

深度限制解决了无限循环的问题

如果 $L < d$，搜索可能是不完全的。即最浅的解的深度比 L 要深。这里 d 是解所在的深度

如果我们选择 $L > d$，深度受限搜索也可能不是最优的（如果只找一个解就停止）。它的时间复杂度是 $O(b^L)$ ，空间复杂度是 $O(bL)$

深度优先搜索可以看成 $L = \infty$ 的深度受限搜索深度受限搜索可能有两种搜索不成功的情况：真的

“没有解”和由于没有搜索到足够深度而返回“无解”

1.3.4 迭代加深

不知道解在何处，迭代加深搜索算法是最推荐的使用的方法
使用 DFS 依次搜索 0,1,2,3,... 层

1.3.5 双向搜索

同时运行两个搜索程序：从前向后 + 从后向前，中间相遇
动机是 $b^{\frac{d}{2}} + b^{\frac{d}{2}}$ 远比 b^d 小
双向宽度优先的时间复杂度和空间复杂度都是 $O(b^{\frac{d}{2}})$

1.3.6 一致代价搜索 (Uniform Cost Search,UCS)

宽度优先，先展开浅层节点，遇到目标就是结束，这就是最优算法。前提是每一步的花费是一样的。如果每一步的花费不一样呢？比如这个真实的最短路问题。如图所示，从 S 市到 B 市，下面这条路走三段，花费是 278，上面那条路走两段，花费是 310。如果用宽度优先得到的结果是上面那条路显然不是最优的
于是我们有了一个一致代价搜索
1、实现

function UCS(problem) returns a solution, or failure
    node⟵a node with STATE=problem.INITIAL-STATE,PATH-COST=0
    frontier⟵a priority queue ordered by PATH-COST,with *node* as the only element
    explored⟵an empty set
    loop do
        if EMPTY?(*frontier*) **then return** failure
        node⟵POP(*frontier*)/*choose 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⟵CHILD-NODE(*problem,node,action*)
            if *child*.STATE is not in *explored* or *frontier* **then**
                frontier⟵INSERT(*child,frontier*)
            else if *child*.STATE is in *frontier* with higher PATH-COST **then**
                replace that *frontier* node with *child*

要点：
实现 UCS 搜索的核心在于使用**优先队列**!
开节点集是一个优先队列
闭节点集是一个哈希表

当发现一条更优路径时，更改开节点集的信息

优先队列按从起点到 n 的花费 $g(n)$ 排序

有可能找到两条路径

展开时没有更短的了

关键点：从开节点集出来时找到最优解，留下的花费更高

2、算法复杂度

用 C^* 表示最优解的花费, 并且假设每一步的最小花费是 ϵ

当所有步骤的花费一致时，$b^{1+\lfloor C^*/\epsilon \rfloor} = b^{d+1}$

算法在最坏的情况下的时间和空间复杂度是 $O(b^{1+\lfloor C^*/\epsilon \rfloor}$，这比 b^d 大得多

因此，一致代价搜索有可能在找到解之前展开一棵有很多小的花费的枝丫的很大的树

1.4 有信息搜索（启发式搜索）

· 有信息搜索——运用问题相关的知识，更有效的进行搜索

· 被选择展开的节点是根据某个**估值函数 evaluation function,**$f(n)$ 决定的

贪婪最佳优先搜索，A^* 搜索，启发式搜索

1.4.1 贪婪最佳（贪心）

1、理论

· 贪婪最佳搜索的实现和 UCS 是一样的，只不过用估值函数 f 取代 g 来确定优先顺序

· f 决定了搜索策略

· 最简单的情况是 $f(n) = h(n), h(n)$ 是从节点 n 到**目标节点**的最短路径的估计值

· 贪婪最佳搜索试图每次寻找离目标最近的节点，这貌似可以快速找到解，所以它使用启发式函数 $f(n) = h(n)$

对于一个具体问题，其搜索花费一般是最小的，但是有可能不是最优解，因为每一步都只选择看似距离目标最近的节点展开

1.4.2 A^* 搜索

1、理论

一致代价只看前面已经发生的实际花费 $g(n)$，不够高效；贪婪最佳只看后面未发生的估计花费 $h(n)$ ，可能找不到最优解, 把 $g(n)$ 和 $h(n)$ 结合起来:

$f(n) = g(n) + h(n) =$ 预计经过节点 n 的最短路径的花费

因为我们要找到的解的好坏是根据全程花费来的，所以优先展开 $g(n) + h(n)$ 值最小的节点是合理的

2、A^* 的最优性

这里的思路是，根据 A^* 的算法流程，当目标节点被展开时，它的花费就是真实的，就找到了最优解。那么我们就要保证开节点集的其他节点再出来时不会有更优的路径了。目前在开节点集里的点的 f 值都比解的实际花费要大。但是 $f = g + h, h$ 是估计的，可能不准

只要保证估计的未来花费 $h(n) \leq$ 真实未来花费，A^* 就有最优解

只要开节点集里的点到目标的真实花费不小于 h，就能保证这些点从开节点集出来后到达目标的真实花费不小于当下得到的解

$h(n)$ **是可采纳的 admissible heuristic，如果** h **不会超过真实花费**（（$h(n)$ 是真实花费的下限，极限情况 $h(n) = 0$ 的话就是只用 $g(n)$ 的一致代价搜索了））则 A^* 是最优的：

证明：如果存在一条通过 n' 从 n 到 Gn 的花费比 $h(n)$ 更小的路径, 就会与 $h(n)$ 是通过 n 到达 Gn 的路径的花费的下限的前提相矛盾。

关键：估值函数 $h(n)$ 是可采纳的，A^* 就是最优的

A^* 的最优性：

树搜索版本的 A^* 是最优的，如果 $h(n)$ 是可采纳的

图搜索版本是最优的，如果 $h(n)$ 是一致的

3、A^* 的完备性和高效性

· A^* 算法是完备的：当满足所有单步花费都超过一个有限的值 ϵ，并且分支数 b 有界时，如果所展开的花费 $\leq C^*$ 的节点数目是有限个时（主要说明算法会主动结束）

· A^* 是最高效的：只展开了必要的节点（除了对于 $f(n) = C^*$ 的节点），这是因为如果不展开所有 $f(n) < C^*$ 的节点，就有可能错过最优解（注：只是针对这个 f 是最高效的）

· A^* 的实际效果取决于启发式函数的好坏

· 对于大多数问题来说，在目标等高线之内的状态数目相对于解路径的长度来说依旧是指数关系。

1.4.3 启发式函数

1、启发式函数之间的比较

当对于两个启发式函数 $h_1(n), h_2(n)$，总有 $h_2(n) \geq h_1(n)$，我们说 h_2 支配 h_1

支配意味着有效，因为用 h_2 永远不会比 h_1 展开更多的节点（除了 $f(n) = C^*$）

证明：用 A^* 算法所有 $f(n) < C^*$ 的节点都会被展开，且 $f = g + h$，即 $h(n) < C^* - g(n)$ 一定会被展开，对于任何节点 $h_2(n) \geq h_1(n)$，所有被 h_2 展开的节点都会被 h_1 展开，h_1 还会展开更多的点

所以，一般来说拥有更大的启发式函数更好，前提是计算启发式函数值不会浪费更多的时间

2、启发式函数的设计方法

· 松弛法获得启发式函数

· 减少对动作的限制可以得到原问题地松弛问题

· 松弛问题的状态图是原问题状态图的超图，因为可行动作增加了（比如说走地图都可以直线飞行了），在原来的图里增加了新的边

· 原问题的最优解也是松弛问题的解。松弛问题会有更优的解，如果加上的边提供更短的路径。所以松弛问题的最优解是对原问题的一个可采纳的启发式函数

· 通过松弛问题产生可采纳的启发式（松了意味着 $h(n)$ 肯定小于真实的未来花费）

· 一个数字可以从 A 到 B，如果:

条件 1: A 与 B 垂直或水平相邻 B

条件 2: B 是空的

· 可以产生三个松弛问题，通过去掉任何一个约束条件或者两个同时去掉:

(a) 一个数字可以从 A 到 B 如果 A 和 B 相邻（没条件约束 2）

(b) 一个数字可以从 A 到 B 如果 B 是空的（没条件约束 1）

(c) 数字可以从 A 直接到 B（没任何条件约束）

· 一个最佳的启发式函数很难获得

· 如果能得到一组启发式函数 $h_1(n), ..., h_m(n)$，但是其中没有一个占统治地位我们该如何选呢？事实上不需要选，全部都用上就行:

$h(n) = max\{h_1(n), ..., h_m(n)\}$

· 都算出来取最大的。因为每个都是采纳的，所以最大的也是可采纳的

· 同时，这个启发式函数支配所有其他的

2 搜索中的局部优化算法

2.1 概要

2.1.1 为什么有局部优化算法

在搜索问题中，为了速度，我们往往可以放弃找到全部解，只找一个可行解;

例如，在 n 皇后问题中，我们不关心路径，只要最终结果，因此进行局部调整：先把八个皇后随便摆放到棋盘的八列上，然后哪个皇后处在被攻击的位置，就调整哪个皇后，直到八个皇后互不攻击;

· 由于全局搜索要记住搜索路径，必定会受到内存的限制，不适合解决超大规模问题。在实际中，很多问题并不需要记住得到解的路径，只需要得到解。这一节我们不再关心得到解的路径;

· 盲目（无信息）搜索遍历整个动作序列空间，A^* 试图使用问题相关启发式函数加快搜索。这一节我们将启发式函数改成状态估值函数，并充分利用它来寻找解;

2.1.2 重新建模

问题描述模型：

1、初始状态 S_0-八个皇后在棋盘上，每列有一个皇后

2、动作–移动任意列上的皇后到同列其他行（8*7）

3、状态转移–移动后棋盘的样子（邻居状态，56 个）（**状态之间不再以父子相称，而是邻里相称**）

4、目标状态 Goal–皇后不互相攻击

5、状态估值函数 h-可能互相攻击的皇后对的数目，可以是直接的，也可以是间接的。表明当前状态的好坏。

2.1.3 状态估值函数

状态估值函数，描述一个状态好还是不好，输入当前状态，输出一个评估值

整个问题的求解就是根据状态估值函数不断从当前状态移动到估值更低/高的邻居状态，直到到达目标状态为止的过程

基本思路：

1、从一个初始状态出发，向更好的邻居状态移动；

2、如果邻居都不如当前状态的评估值高，我们就来到了一个局部极值点（局部最优解）。如果全局只有一个极值点，我们就找到了要找的目标状态。

2.1.4 解空间的形状

在局部优化中，最重要的一点是，解空间的形状是非常复杂的，有以下几个关键概念：

1、局部极值：在一个局部范围内的最优解，就像深度学习中的梯度下降一样，但是不是全局最优解；

2、全局最值：整个解空间中的最优解，也是我们期望得到的解；

3、平台：在平台上，无论做任何动作（1 步），状态估值函数的值都不会发生改变，即邻居的估值和自身一样；

4、肩状平台：一般平台上，往两边进行多步后状态估值函数会变差，也就是说达到了一种“局部平台最优”，而肩状平台则是往两边进行多步后有一侧会变差，一侧会变好；

一个局部优化算法是完备的和最优的的含义是：

· 完备的：如果目标存在则总能找到；

· 最优的：算法能找到全局最优解；

2.2 爬山法

2.2.1 最陡下降爬山法

每次当前状态移动到**相邻节点中最好的一个**（则最陡的、目标评估值最优）；

算法会在一个山峰（四周的点都比它低）处停下;

算法并不存储一棵搜索树，所以数据结构**只存储当前节点和一个估值函数**;

注意：爬山法存在正负两个版本，爬到山峰还是山谷，取决于状态估值函数 h 是越大越好还是越小越好而已。代码里面只要改一个正负号;

2.2.2 问题：陷入局部最优

每次算法到了一个局部极值点就僵住不动了

以八皇后问题为例，从一个随机生成的八皇后状态，用最陡爬山法，86% 僵住，能解决 14% 的问题;

找到解的平均花费是 4 步，而僵住的平均花费是 3 步，这对于 $8^8 = 17$ million 个状态来说还不赖。

2.2.3 平移

利用随机平移：随机选一个 h=1 的状态、允许皇后跳到随机的 h=1 的新状态

但是也可能死循环（来回移动），设置一个上限，比如 100 次、若都跳不出去就放弃了

也就是说、让状态连续在状态估值函数平台上随机左右平移 100 次出解率 14% –> 94%，出解率大

幅提升

但是找到解平均 21 步，找不到解平均 64 步，加入随机平移后，找到/找不到解的代价都增加了很多；

随机平移只能跳出肩状平台，不能跳出平台，一旦某次平移走到平台边界，就能脱离平台，继续往全局极值移动；

2.2.4 随机爬山法-Stochastic hill climbing

顾名思义，即在更优的邻居点里随机选一个

· 每次从更优的点中随机选一个（之前的最陡爬山法是选择最优点）

· 这种方法一般比之前的最陡爬山法收敛的慢。但是在某些空间里，它往往可以找到更优的解。

· 不一定完全随机，选择的概率可以根据陡峭程度设定，可加速收敛，类似于可变化的学习率；

我们对最陡爬山法和随机爬山法进行一个比较：

最陡爬山法：生成估值表，获得所有邻居的估值，选择最优的那个邻居;

随机爬山法：生成估值表，获得所有邻居的估值，随机或根据概率选择最优的那个邻居;

发现共同的问题是，如果邻居很多，生成估值表会很慢！

2.2.5 第一选择爬山法-First-choice hill climbing

第一选择爬山法不生产估值表

· 随机找一个邻居，计算它的估值，若它的估值比当前状态要好，则立马跳到那个邻居上；若这个邻居估值没当前的好，则随机再选择一个邻居做相同的操作;

· 好处是不需要生成估值表来获得所有邻居的估值了，当一个状态有很多后继时 (比如几千个) 这不失为一个好办法;

但是, **到目前为止我们描述的爬山法是不完备的—当解存在时它有时会找不到解因为可能陷在局部极值**

2.2.6 随机重启爬山法-Random-restart hill climbing

找不到解再从随机位置（初始状态）开始

“If at first you don’ t succeed, try, try again.” 如果失败了就再试一次;

一系列的爬山算法，每次随机产生一个初始状态;

它是近乎完备的，因为总会有一个初始状态会找到解;

爬山法是否成功取决于状态空间的形状。如果没有很多局部极值和平台，随机重启爬山法可以很快找到一个好的解;

随机重启爬山法是非常高效的：

一次就找到最优解成功率 p-> 期望的重启次数是 $\frac{1}{p}$

对于八皇后问题：

不允许平移：$p \approx 0.14$

平均重启次数：7 次以内找到一个解（6 次失败，1 次成功）

平均步数 = 成功的步数 + 失败的步数，根据最陡下降爬山法中的数据，约为 $4 + 3 * 6 = 22$ 次

允许平移：$p \approx 0.94$

平均重启次数：1.06 次

总步数：根据平移法中的数据，约为 $(1 * 21) + (0.06/0.94) * 64 \approx 25$ 次

对于八皇后问题, 随机重启爬山法是非常高效的, 即使是 300 万皇后问题，该解法也能在一分钟内出解。

2.2.7 总结

1、最陡爬山法：获得所有邻居的估值，跳到最优的邻居。最贪心的方法，很容易进入局部极值点

2、随机平移：可以帮助算法跳出肩状平台，但对一般平台无效

3、随机爬山法：获得所有邻居的估值，跳到更好的邻居（不是最优）。往往比最陡爬山法有更优解，但收敛速度变慢

4、第一选择爬山法：不需要获得所有邻居的估值，减少计算量

5、随机重启爬山法：一个暴力找最优解的方法。若找不到最优解、就随机一个新的初始化状态再试试

2.3 模拟退火算法-Simulated annealing

之前我们已经讨论了两种方法：爬山法和纯粹的随机游走算法（不使用 $h(n)$ 的搜索算法）

1) 爬山法的问题在于，只向更优邻居点移动，不会向状态估值更差的邻居移动 -陷入局部极值- 不完备的；

2) 纯粹的随机游走算法的问题在于，等概率地向任何一个邻居移动，是完备的，但却是非常低效的。

2.3.1 启发

· 在材料加工领域，“淬火”是金属或玻璃到一个很高的温度、然后瞬间放到水或油中快速降温，以提高硬度，但材料往往会很脆。(找局部最优解)

· “退火”是指“淬火”后重新加热金属或玻璃到某个温度、然后慢慢冷却，使之没那么脆，且硬度还在期望的性能上。(温度慢慢降低、找全局最优解)

· 乒乓球 A ping-pong ball. 如果我们摇晃一个表面，可以将球摇出局部极值点。这里摇晃的力度需要保证球能被摇出局部极值点但不会被摇出全局最优点。

因此，我们的解决方案是，**爬山法 + 随机游走算法 = 高效又完备的算法——模拟退火**

2.3.2 思想

· 在搜索的初期以一个较大概率允许向下走

· 这个概率和这一步“坏”的程度成指数关系;

· 随着时间的推移这个概率会变小，这个概率会随着“温度”的下降而下降;

我们设法让“温度”足够慢地下降，算法会最终以接近 1 的概率找到全局最优解。

假设估值越大越好：

1) 用一个“温度”值来控制**往差方向搜索的概率**，允许跳到差的邻居。温度越高，则概率越大

2) **搜索刚开始的时候**温度高，跳到差邻居的概率大，以此来跳出局部极值

3) 温度随着搜索时间的推进而减少，跳到差邻居的概率越来越小、接近于 0，以此找到全局极值

2.3.3 实现

伪代码

function SIMULATED-ANNEALING(*problem,schedule*) **returns** a solution state

inputs: *problem*, a problem

schedule, a mapping from time to ”temperate”

current⟵MAKE-NODE(*problem*.INITIAL-STATE)

for t=1 to ∞ do

$T \longleftarrow$ *schedule(t)*(时间到温度的映射函数)

if $T = 0$ **then return** current

next⟵a randomly selected successor of *current*

$\triangle E \longleftarrow$ *next*.VALUE-*current*.VALUE

if $\triangle E > 0$ **then** *current*⟵ *next*

else *current*⟵*next* only with probability $e^{\triangle E/T}$

假设估值越大越好，

$\triangle E$ 越小，概率越小，则邻居越坏跳过去概率越小

T 越小，概率越小，则随着搜索时间增加、概率越小

T 与 $\triangle E$ 决定了以多大的概率跳过这个点

其中存在超参数的选择问题：

1) 刚开始温度（步幅）应该多大?

2) 以什么样的速率下降温度?

3) 温度降到什么程度可以报告解?

4) 需要具体问题具体分析;

2.4 局部束搜索-Local beam search

对于全局搜索，展开整个搜索树，占用内存太大，分支太多

而爬山法和模拟退火只用一个节点搜索 - 矫枉过正

我们会问：如何用更多内存加快搜索?

2.4.1 思想

在同一时刻保留 k 个状态。

1）它从 k 个随机生成的初始状态开始。

2）在每一步, 所有 k 个状态各自生成后继，共有 k*b 个后继。

3）如果 k*b 个后继中有一个是全局最优点，则算法结束。否则从全部 k*b 个后继中找出 k 个最好的后继成为当前状态的 k 个状态，算法继续。

重点：这 k 个搜索是相互通讯的，一个状态若找到好几个好后继，会告诉其他人来一起往下搜索

2.4.2 比较

乍一看，k 状态局部束搜索类似于同时运行 k 个随机重启爬山算法。事实上，这两个算法非常不同。

随机重启爬山法：每个搜索是彼此独立的，不通讯的

局部束搜索：在上述第 3 步中选出最优的 k 个后继作为新状态

2.4.3 问题

基础版本的局部束搜索, 会很快向状态空间中的一个小的局部聚集，k 个状态之间缺少多样性，使得在找更优解方面跟爬山法差不多，都容易到局部极值点，这个问题和最陡下降爬山法是一样的

缓解方案：随机束搜索-stochastic beam search

类似于随机爬山法, 减轻进入局部点的问题。随机束搜索不再选择 k 个最好的后继，而是用一个函数计算出一个与后继的优劣程度相关的量，并以此为概率选择 k 个后继。随机束搜索有些像“自然选择法则”，适者生存概率高，不适者死亡概率高;

2.5 遗传算法-Genetic algorithms

局部束搜索的问题：很快集中到一小部分节点，缺少多样性，比单纯爬山改进有限

2.5.1 思想

是随机束算法的一个变种;

后继节点不再是由单个状态产生了，而是由两个父状态结合产生的;

类似于自然法则的从无性繁殖到有性繁殖，即扩大了后代的多样性;

以八皇后问题为例：

用行号表示:**327**52411 + 247**48552** = **32748552**

给出了前两个父代“交配”产生第一个子代的情况。非加粗的部分在“交配”中丢失，加粗的部分被保留下来。

像随机束搜索一样，遗传算法结合了爬山法和随机探索，并且在并行的搜索线程中交换信息。

算法的主要好处来自交换子串。

从数学上来讲，如果父代的编码是随机的，交叉并不会带来好处。

直觉上好处来自于交叉可以把具有好的功能的块组合在一起来产生更好的下一代。

例如，将前三个皇后分别在 2, 4, 6 行互相不攻击的皇后作为一个好的块保留下来与其他不同的块进行组合也许会得到更好的状态。

2.5.2 实现

伪代码:

function GENETIC-ALGORITHM(*population*（群体，k 个初始状态）,FITNESS-FN) **returns** an individual（个体，也就是解）

inputs: *population*,a set of individuals

FITNESS-FN（适应/估值函数）,a function that measures the fitness of an individual

repeat

new_population⟵ empty set

for i=1 **to** SIZE(*population*)（群体个数，从 1 到 k）**do**

x ⟵ RANDOM-SELECTION(*population*,FITNESS-FN)（根据估值求概率，选一个个体 x）

y ⟵ RANDOM-SELECTION(*population*,FITNESS-FN)（根据估值求概率，选一个个体 y）

child ⟵REPRODUCE(x,y)（**交配**）

if (small random probability) **then** *child*⟵ MUTATE(*child*)（子代以很小的概率发生变异）

add *child* to *new_population*

population⟵*new_population*

until some individuals is fit enough,or enough time has elapsed（循环直到估值函数发现最优解）

return the best indivuduals in *population*,according to FITNESS-FN（最后在 k 个状态中，选择最好的返回）

function REPRODUCE(x,y) **returns** an individual

inputs: x,y,parent individuals

n ⟵LENGTH(x);c ⟵random number from 1 to n（根据个体的长度，随机选择一个 c 作为分割点）

return APPEND(SUBSTRING(x,1,c),SUBSTRING(y,c+1,n))（1-c 取 x 里的，c+1-n 取 y 里的）（交叉配对，合并出子代）

2.5.3 schema-模式

遗传算法的理论用模式 schema 来解释它的工作原理。

模式的意思是某个字符串可以有一些位置空置着。

例如，模式 246***** 表示所有前三个皇后位置为 2, 4, 6 的串。

与模式相匹配的具体的串称为模式的一个实例，例如 24613578 。(类似于类与派生类)

使用 schema 主要的目的是如果具有该模式的个体比平均的要好，则通过遗传该模式，后代中好的个体数目会变多。具有较好模式的数目会增加;

当模式很好地表达了解中的有特征意义的部分时，遗传算法会表现得很好;

这就表明如果想要遗传算法有好的表现，状态的良好表达非常重要。

在实际应用中，遗传算法在优化问题上有着很广泛的影响。例如集成电路布线和作业调度问题。目前还不清楚，遗传算法的魅力是源自她良好的效果还是源自她起源于进化论的神奇传说。究竟何种情况下遗传算法会取得好的效果还有待于进一步研究。

2.6 连续空间中的局部搜索

前面的例子用的都是离散空间的问题，如何用搜索的方法解决连续空间的问题呢?（输出是浮点数）

2.6.1 机场问题

假设我们要在地图上新建三个机场, 我们希望地图上每个城市与离它最近的机场的距离的平方和最小，状态空间定义为机场的坐标:$(x_1, y_1), (x_2, y_2), (x_3, y_3)$, 这是一个六维空间; 我们也称为状态由六个浮点数变量所定义;

对于任意的给定状态，目标函数 $f(x_1, y_1, x_2, y_2, x_3, y_3)$ 是很容易计算的。

令 C_i 表示离机场 i 最近的城市集合（一个机场对应一个城市集合，每个城市找最近的机场作为它所在的集合），则有

$$f(x_1, y_1, x_2, y_2, x_3, y_3) = \sum_{i=1}^{3} \sum_{c \in C} (x_i - x_c)^2 + (y_i - y_c)^2$$

一种避免连续问题的方法是离散化。例如，我们将地图坐标离散化，每次可以只以一个很小的量 $\pm\delta$ 移动一个机场的 x 或者 y, δ 越小则精度越高;

对于 6 个变量,x 或 y 每次变化是增加或减少 δ，则当前状态有 12 个可能的后继邻居。于是我们就可以使用前面介绍的任何局部搜索的算法;

· 初始状态 $S_0 - (x_1, y_1, x_2, y_2, x_3, y_3)$ 任意位置

· 动作–任何一个 x 或者 y，$\pm\delta$（12 个合法动作）

· 状态转移–采取动作后的位置（邻居状态，12 个）

· 目标状态 Goal–$f(x_1, y_1, x_2, y_2, x_3, y_3)$ 最小的 x, y 取值

· 状态估值函数 $h - f(x_1, y_1, x_2, y_2, x_3, y_3)$

方法一：经验梯度的值

左右各取一个点 $x + \delta$,$x - \delta$,

比较 $f(x), f(x + \delta), f(x - \delta)$ 的值;

向好的方向移动;

方法二：梯度下降（gradient descent）/上升（ascend）

前提是函数 f 可求导，不需要求 $f(x \pm \delta)$ 都可以知道方向

目标是最大化则上升 $x + \alpha f'(x)$，或最小化则下降 $x - \alpha f'(x)$, 导数绝对值越大则越陡，更新幅度越大

α 控制基础的幅度，称为学习率

若估值函数不可微–则使用经验梯度的值法

若能计算局部梯度，我们可以用梯度法来实现最陡爬山法，依据下面的公式修改当前状态:

$$x \longleftarrow x + \alpha f'(x)$$

这里 α 是一个小的常数，通常称为学习率或步长。“α 是一个小的常数” 引发了一大类调整 α 的方法;
基本的问题是, 如果 α 太小, 需要太多的步骤、更新慢; 如果 α 太大, 搜索可能会越过极值点。
线性搜索（line search）技术试图沿着当前对的梯度方向延展：动态修改 α 和在离散空间一样，在连续空间里，局部搜索算法的最大难题仍是局部极值和平台。
随机重启和模拟退火经常是有效的。

2.6.2 总结

对于多维的情况，例如我们的机场问题有

$$f(x_1, y_1, x_2, y_2, x_3, y_3) = \sum_{i=1}^{3} \sum_{c \in C} (x_i - x_c)^2 + (y_i - y_c)^2$$

每一维都是彼此独立的，可以分别求方向，以获得多维的方向
目标函数的梯度向量 ∇f，给出了优化方向和陡峭度

$$\nabla f = (\frac{\partial f}{x_1}, \frac{\partial f}{y_1}, \frac{\partial f}{x_2}, \frac{\partial f}{y_2}, \frac{\partial f}{x_3}, \frac{\partial f}{y_3})$$

2.7 最小化局部约束

伪代码:

function MIN-CONFLICTS(*csp,max_steps*) **returns** a solution or failure

inputs:*csp*,a constraint satisfaction problem（约束满足问题，作为找到解的条件）

max_steps,the number of steps allowed before giving up（最大步数，作为放弃搜索的条件）

current⟵an initial complete assignment for *csp*（随机初始化状态）

for *i*=1 to *max_steps* **do**

if *current* is a solution for *csp* **then return** *current*（若当前状态是解，则返回）

var⟵a randomly chosen conflicted variable from *csp*.VARIABLES（若当前状态不是解，则随机找一个存在冲突的变量）

value⟵ the value *v* for *var* that minimizes CONFLICTS(*var,v,current,csp*)（将这个变量更新到让冲突最小化的值上）

set *var*=*value* in *current*

return *failure*（如果找不到解、或者最大步数到了，则放弃）

3 约束满足问题-Constraint Satisfaction Problems

全局搜索和局部搜索把状态当作一个整体，约束满足问题拆开状态求解

3.1 概念

一个约束满足问题有三个组成部分,X, D 和 C:

X 是一组变量, $\{x_1, x_2, ..., x_n\}$

D 是一组值域, $\{D_1, D_2, ..., D_n\}$, 每个变量一个值域（取值范围）

C 是一组约束，用于指定变量取值之间的约束关系；

每个值域 D_i 包含一组对于变量 X_i 可行的取值 $\{v_1, v_2, ..., v_n\}$，每一个约束条件 C_i 包含一对数值(scope,rel),scope 是约束条件中涉及的变量,rel 定义了变量的可行取值之间的关系。

3.2 CSP 问题的解

在 CSP 问题中，全部变量的一组赋值 $\{X_1 = v_1, X_2 = v_2, ...\}$，如果不违背任何一个约束条件，则称为这个 CSP 问题的一个**可行解**，是完全不违背约束条件的完美解，可能存在多个

如果我们给定一个解的评价方式，则我们还可以寻找一个**最优解（注：最优解不一定是可行解，但是是可以找到最好的解了!）**

在实际求解问题中，我们可能会要寻找:

一个可行解

全部可行解

一个最优解

3.3 把问题建模成约束满足问题 CSP 的原因

3.3.1 原因一

表示很宽泛的类别的问题；设计通用 CSP 求解器，使用自动优化技术迅速排除大量的无关搜索空间，比基于状态空间的搜索算法更快。

3.3.2 原因二

知道为什么某个赋值不是解。发现一个部分赋值不是解时，知道是哪个变量的赋值违背了哪个约束，并可以立即放弃对当下路径后续探索。

许多使用传统搜索方法不能出解的问题，建模成 CSP 问题可以很快出解。

3.4 CSP 模型的可变因素 - 变量的类型

CSP 问题是用 X、D、C 三元组表示的，X 是变量，D 是取值范围。

3.4.1 变量 X

它的类型可以是离散的，也可以是连续的；

3.4.2 值域 D

它可以是有限的，也可以是无限的；

离散类型变量的取值也可以是无限的，例如整数，或者是字符串。

连续值域的约束满足问题的例子。例如，哈勃望远镜（the Hubble Space Telescope）观测时间需要非常精确的；观测开始和结束的时间，是在连续值域上取值的变量，并且需要遵循一系列天体、先决条件和电力的限制。连续值域的约束满足问题在现实世界里是很常见的，也是被广泛研究的。

3.4.3 约束条件 C

用 C 所涉及到的变量个数来对它分类:

一元约束（unary constraint）, 对单个变量取值的约束。

二元约束（binary constraint）涉及到两个变量。

三元约束，例如，变量 Y 的取值在变量 X 和 Z 之间, Between(X, Y, Z)

全局约束（global constraint）含有任意多个变量。最常见的全局约束是 Alldiff, 其含义是所有变量的取值必须互不相同。

用 C 是否是强制性的来分类:

绝对约束，违背这种约束就意味着排除了成为解的可能性。

优先约束，用来指定哪些解是优先选择的，可用来求最优解。例如，R 教授可能会更倾向于上午。那么如果安排 R 教授下午 2 点上课依旧是可行的，但不是最优选择。优先约束（Preference constraints）一般会实现成给每个变量的赋值增加一个花费（costs）, 例如，为 R 教授安排下午的课花费为 2，而安排上午的课花费为 1。这样，有优先约束的 CSP 问题称为约束优化问题（constraint optimization problem）, 简称 COP。

3.5 约束传播：CSP 问题的推理

在普通的状态空间搜索中，我们能做的事情只有搜索。
在 CSP 问题中，除了搜索还可以有一个选择就是推理（我们也称之为约束传播）:
使用约束减小一个变量的可能取值, 由此还可以减少其他变量的可能取值。约束传播可以和搜索相结合，也可以先用约束做预处理，再搜索。
有时预处理就可以解决全部问题，不用再搜索了。

3.5.1 点一致

变量的所有取值都满足一元约束。可以通过运行点一致性算法去除所有的一元约束

3.5.2 边一致

变量的所有取值都满足二元约束。一个网络是边一致的，如果每个变量相对于其他任意变量都是边一致的。

当面对一个问题，除了用搜索，我们还可以用推理，来去除一些不合理的解。

3.6 实际中的约束满足问题

· 寄存器分配问题：将大量的程序变量合理分配给少量寄存器，以提高程序执行效率。

· 旅行商问题：给定城市列表和城市间距离，找到一条回路访问每座城市恰好一次并返回起点，使得移动路线最短。

· 绿色物流规划问题：根据供需关系、运输能力等，合理规划物流活动中的运输、存储、配送等环节，降低物流成本，同时通过降低能源消耗和废物排放来减少对环境的污染。

· 地下水流速估计问题：给定地下水道网络结构，水流入口和出口等信息，估计每段区间水流的流速。

· 基因组映射问题：给定一组基因片段及它们之间的距离约束，要求在一段 DNA 序列中定位这些基因片段。

3.7 约束满足问题的通用求解方法

教材中给出的方法:

通用约束求解器

描述语言（例如：MiniZink）

实现一个具有动态优化策略的大而全的通用求解器

每次把问题送进去直接求解

林舒博士提出的方法:

自动生成求解程序（PDL2C）

描述语言（PDL）

通过静态分析问题性质，并基于这些性质自动生成包含了剪枝优化和动态规划的问题求解程序

3.7.1 PDL2C 实现流程

1、通过词法分析和语法分析创建标识符表;

2、选取独立变量

识别变量之间的依赖关系;

将所有变量按照取值范围从小到大排序;

采用启发式搜索的方式找到最小的独立变量集;

3、根据独立变量的类型情况，生成基于循环的搜索程序框架或基于递归的搜索程序框架;

4、进行搜索剪枝优化和动态规划优化 (此处为难点，单列于第二个创新点);

5、生成 C 代码。

OJ 上的题都可以作为训练 PDL 的数据集!

4 对抗搜索

对抗游戏的决策问题

零和游戏: 我赚的就是你输的，内卷!

非零和游戏: 各怀心腹，每个人只管最大化自己的利益，损人未必利己，有时候还要合作，多个人从第三方赚收益

例：不围棋，换手五子棋，斗地主，国标麻将，坦克大战，吃豆人

4.1 游戏的特征和分类

4.1.1 人数

单人游戏、双人游戏、多人（3 人以上）游戏

4.1.2 对抗性（收益总和是否为零）

零和游戏、非零和游戏

4.1.3 随机性（动作引发的后果）

确定性游戏、非确定性游戏

4.1.4 状态可见性

完全信息游戏、非完全信息游戏

4.1.5 同步性

同步游戏, 异步游戏

4.1.6 环境的可变性

环境信息不变游戏, 环境信息变化游戏

4.1.7 环境物理规律的可知性

物理规律已知游戏, 物理规律未知游戏

	人数	对抗性	随机性	环境信息	状态可见性	同步性	物理规律
八数码	单人	N/A	确定性	不变	信息完全	N/A	已知
扫雷	单人	N/A	确定性	变化	信息不完全		已知
多臂老虎机	单人	N/A	确定性	不变/变化	信息完全	N/A	未知
围棋	双人	零和	确定性	不变	信息完全	异步	已知
坦克大战 2s	双人	零和	确定性	变化	信息不完全	同步	已知
双陆棋	双人	零和	不确定性	不变	信息完全	异步	已知
石头剪刀布	双人	零和	确定性	不变	信息完全	同步	已知
囚徒困境	双人	非零和	确定性	不变	信息完全	同步	已知
斗地主	多人	N/A	确定性	变化	信息不完全	异步	已知
国标麻将	多人	N/A	确定性	变化	信息不完全	异步	已知

4.2 双人零和完全信息游戏的决策

我们首先考虑双人零和游戏。玩家 MAX 先走，然后轮换直到游戏结束。游戏结束时，赢家得到奖赏，输家得到惩罚。

4.2.1 问题定义

一个游戏可以用下面的元素定义成一个搜索问题：
1)S_0: 初始状态, 描述游戏开始时的状态。
2)PLAYER(s): 定义在一个局面下，轮到哪个玩家选择动作。（比之前的搜索算法多的部分）
3)ACTIONS(s): 返回在某个状态下的合法动作集合。
4)RESULT(s,a): 状态转移模型（transition model）, 一个动作执行后到达哪个状态。
5)TERMINAL-TEST(s): 游戏结束返回 true 否则返回 false，游戏结束时的状态称为终止状态。
6)UTILITY(s, p): 效用函数 (目标函数或者支付函数), 游戏结束时玩家 p 的得分。零和游戏是指所有玩家得分的和为零（完全信息）

4.2.2 问题

难点在于这棵搜索树可能很大! 例如，国际象棋平均每步有 35 个选择，一局对战每个选手要下 50 步左右，两个人就是 100 步。所以搜索树有 35^{100} 或 10^{154} 个节点。
围棋平均每步有 250 个选择，一局对战两个选手要下 150 步左右。所以搜索树有 250^{150} 或 10^{360} 个节点。

前面讲到的搜索算法无法遍历和存储这样一棵搜索树，这就使得这个问题变得很难
然而，游戏的乐趣就在于难解。就像现实世界一样，当无法计算最优动作时，游戏依然要做某种决策。游戏 AI 的研究产生出一些有趣的方法来解决如何有效利用时间进行搜索的问题。

4.3 极大极小搜索 MINIMAX（最基础）

最初的思路：模仿人类玩游戏（下棋）的方式：
“如果我是黑方，看到这个局面应该下在哪里?”
“如果下在这里，对手会如何应对?”
“如果对手这样应对，我应该怎么下才好?”
1949 年，Claude Shannon 香农提出了电脑如何下国际象棋的构想，但未实现:
1) 遍历所有可能发生的局面，并找出最佳的方案;
2) 这种方案假设对手和你一样聪明，他总是最小化你的收益，而你自己总是最大化自己的收益;
3) 你采取的方案是相对稳妥的方案。
最重要的假设：“我” 和对手都是最聪明的，即这是两人零和游戏，所以对手总是想最小化你的收益

4.3.1 模型

· 状态：局面
· 动作：在该局面下该走棋的玩家的合法动作
· 状态转移：一步过后的所有可能局面
· 极大极小搜索：在决策树上的游历，假设敌人和自己一样聪明
极大极小搜索对未来的游戏 AI 有着极其重大的意义，是人类制造游戏 AI 最为直白的形式
井字棋的搜索树相对较小, 只有少于 $9! = 362880$ 个终局结点。国际象棋有超过 10^{40} 个终局结点。

4.3.2 搜索树

Max 层选最大值
Min 层选最小值
每个节点的值从最下方的叶节点往上传导
强调极大极小搜索的假设: 我们要相信敌人的聪明的、不会犯错的、计算值的最后肯定是选下面最小值，不会把大的值往上传播给我的。

4.3.3 伪代码

function GET-VALUE(node)
    if *node* is a leaf node **then**

4.3.4 效率分析

上面的极大极小搜索算法采用了深度优先搜索算法；
假设搜索树的深度是 m 每个结点有 b 个合法操作。则极大极小算法的时间复杂度是 $O(b^m)$。如果每次访问一个结点把它的儿子全部展开，空间复杂度是 $O(bm)$；如果每次只展开一个儿子是 $O(m)$
对于一个真实的游戏，这种方法会很慢，显然是不实用的, 但是这种方法是对游戏算法进行数学分析和进一步研究实用算法的基础;

棋种	状态复杂度	博弈树复杂度
西洋跳棋	10^{21}	10^{31}
黑白棋	10^{28}	10^{58}
五子棋	10^{105}	10^{70}

1956 年，IBM 的 Arthur Samuel 制作出了计算机上运行的相对成熟的西洋跳棋 AI，能够轻松打败新手玩家
他的 AI 仍然是基于极大极小搜索的，但是却增加了学习能力，从而避开了利用人类有限经验无法完全指导 AI 的难题。Samuel 提出的“学习”包括两种方式：
死记硬背的学习（Rote-learning），也就是对每步的估值评分是从之前极大极小搜索的计算结果中直接拿到的。
一般化（Generalization）学习，比如让估值函数的公式有着不同的参数，而参数可以进行修改，从而不断缩小计算的估值和实际局面评价的差距。
这在当时的机器条件下，带来了 AI 水平的突破性进展。后来的深蓝、甚至今日的 AlphaGo，都可以视为上述第二种算法的演化版本。

4.4 $\alpha-\beta$ 剪枝

MINIMAX 方法在问题：问题规模增大时树的规模太大

4.4.1 背景

五十年代后期，卡耐基梅隆大学与 RAND 公司的 Allen Newell、Herbert Simon 与 Cliff Shaw 组成小组，合作研究国际象棋的 AI;
提出了创造性的 $\alpha-\beta$ 剪枝，大大缩小了 AI 的计算量，将有限的计算资源用在了更加有意义的棋步上。从此，$\alpha-\beta$ 剪枝与极大极小搜索一起成为了国际象棋 AI，以及其他各种 AI 的标准做法;
当我们使用 $\alpha-\beta$ 剪枝算法进行搜索时，它返回和极大极小搜索一样的结果，只是搜索树上那些不会影响最后搜索结果的部分被忽略掉了而不再搜索。也可以说，$\alpha-\beta$ 剪枝算法是极大极小搜索算法的一个简化版本;
Alpha 最小值，Beta 最大值，搜索过程不断缩小范围。

当上方前继节点确定了搜索范围以后，下方后继节点如果不在那个范围内，就立刻被剪枝掉。

4.4.2 实现

算法框架和 MINIMAX 算法一样；只是在两个函数中分别加上了维护和传递 α 和 β 的值的语句
几个要点：
因为 MAX 节点，所以有下界 α
因为 MIN 节点，所以有上界 β
出界就要剪枝
在界内就会更改 α/β，减小界 V 通过向上传播影响父辈的界
伪代码：
function ALPHA-BETA-SEARCH(*state*) **returns** an action
$v \longleftarrow$MAX-VALUE(*state*,$-\infty, +\infty$)
return the *action* in ACTIONS(*state*) with value v
function MAX-VALUE(*state*,α, β) **returns** a *utility value*
if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
$v \longleftarrow -\infty$

4.5 不完美的实时决策

$\alpha - \beta$ 剪枝方法可以剪掉不影响解的节点，加快搜索速度。但当问题规模增大时，仍旧不能解决搜索树太大的问题
劳德 · 香农在文章 Programming a Computer for Playing Chess (1950) 里提出了用启发式函数取代直接截断搜索，有效的将非叶子节点转化成叶子节点。这个方法以如下方法修改 minimax 或 $\alpha - \beta$ 算法。
1）用 cutoff test 取代 terminal test 来决定何时调用 EVAL
2）用可以估计状态的收益的启发式函数的值替 utility 函数
用启发式函数估计状态收益、以代替真实状态收益、减少探索

4.5.1 思路

具体描述如下：
· 首先, 在终局状态必须给出状态估值函数和真正的得分函数；
· 其次, 计算不能用时太多! (关键点在于要搜的更快！)
· 第三, 非终局，状态估值函数和最终的胜负有**很强的正相关性**；

4.5.2 状态估值函数

启发式状态估值函数不一定能够确切地知道谁输谁赢，但是可以返回一个能够反映胜率的值。
例如，我们的经验告诉我们当两个兵对一个兵时，有 72% 的概率会赢（+1); 20% 概率会输 (0)，

8% 概率是平局 ($\frac{1}{2}$)。
一个有价值的状态估值函数可以是类似于:$(0.72 * (+1)) + (0.20 * 0) + (0.08 * \frac{1}{2}) = 0.76$
· 数学一点儿的说法是，这类状态估值函数称作线性加权函数，因为它可以表示为:

$$\text{EVAL}(s) = w_1 f_1(s) + w_2 f_2(s) + ... + w_n f_n(s) = \sum_{i=1}^{n} w_i f_i(s)$$

· 把所有特征的取值加起来看上去是有道理的，但是这里有一个很强的假设: **每个特征的贡献是彼此独立的**;
· 一个特征在其他状态特征不同的时候，价值不同。例如，指定象为 3 分忽略了一个事实就是终局阶段象有更大的价值;
· 还有就是有些情况下无法线性估值，即**场面上的估值不是平滑变化的**，单一的函数对于不同的局面没有办法很好的估计;
· 现在股票及象棋程序以及其他游戏也使用特征的非线性组合；例如，两个象的得分比一个象的得分略高一点儿，一个象在终局时得分会更高些。(也就是说，当走过的步数这个特征变大时或者说棋盘上剩子数目这个特征变小时);
· 特征 $f_i(s)$ 和权重 w_i 不是游戏规则的一部分！他们来自人类千百年来下棋的经验;
· 当我们没有这类经验可用时，估值函数的权重 w 可以用机器学习的方法算出来;
· 事实上机器学习的方法表明 1 个象的价值和 3 个兵的价值相当;

4.5.3 对剪枝的修改

有了状态估值函数，下一步就是修改 ALPHA-BETA-SEARCH 算法，使得它能够在搜索即将截断时调用启发式函数 EVAL;
我们代替 TERMINAL-TEST 函数的地方如下:
if CUTOFF-TEST(state,depth) then return EVAL(state)
我们还要做些标记使得每次递归调用可以增加当前搜索的深度值。最直接的方式是给搜索一个确定的深度限制，使得 CUTOFF-TEST(state, depth) 函数在搜索深度大于 d 时返回;
搜索深度 d 的设置要保证在规定的深度内能够给出一步的走法;
更鲁棒的做法是采用**迭代加深**的搜索方法。当深度限制到了，算法返回搜索到的最深一层所返回的值;
迭代加深的搜索方法也可以用来帮助对一个状态下的动作进行排序;

4.5.4 小结

总结一下，不完美的实时决策，这里**实时决策**指的是我们要在规定的时间内给出决策结果，比如我们下棋，是有一个走子时限的。由于时间有限，我们可能搜不到终局，不知道准确的胜负情况，所以我们说是不完美的实时决策;
解决这个问题可以用启发式状态估值函数来估计的状态价值，然后限定一个搜索的深度，到了一定的深度就截断。如果不好确定深度就用迭代加深的方法。在时间允许范围内迭代加深，时间到了就

停止搜索。

没有搜到终局不知道胜负怎么办呢?

1、我们可以根据经验定义一个估值函数，搜到终局就用真的得分，没搜到就返回**状态估值函数**的值。

2、估值函数既可以是人类经验，也可以用机器学习的方法得到。

3、估值函数的好坏直接影响了搜索算法给出的**决策的好坏**。

4、估值函数还可以用来**对子结点排序**，好的节点排在前面，可以提升 $\alpha - \beta$ 剪枝的效率。

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

不完美的实时决策中截断搜索和估值函数是一种解决搜索树太大的方法，但是估值函数哪里来呢?

蒙特卡洛方法（Monte Carlo method）是统计模拟方法，是一种思想的统称，不是一个严格意义上的算法

基本思想：通过大量采样获得概率估计（依据概率论中的大数定律）

4.6.1 概念

蒙特卡洛树搜索（MCTS, Monte Carlo Tree Search）是一种通过在决策空间中随机采样，并根据结果构建搜索树来在给定域中寻找最优决策的方法。MCTS 这一概念最早在论文中出现是 2006 年 Remy Coulom 的论文 Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search

特点：基于统计的方法，更多的计算能力可以带来更好的性能表现

优点：可以在拥有很少甚至没有领域知识的情况下使用

4.6.2 纯随机蒙特卡洛方法

纯随机蒙特卡洛方法：

· 有 N 个儿子，为了选择估值最好的儿子，对每个儿子下面敌我双方都用随机动作往下搜索直到终局，每个儿子搜索 M 次，求平均值来作为该儿子的胜率。

· 然后跳到胜率估值最好的儿子。

缺点：

1、没有充分利用内存

2、不够准确（除非 M 很大）

4.6.3 方法

改进：与贪心树搜索结合

1 ~ 5 步：最开始每个节点值都不知道，所以都展开、每个儿子随机搜索 M 次，求估值

6 步：根据贪心路径展开最优节点下未展开的一个儿子，并更新路径上节点的估值

7 步：根据贪心路径展开最优节点下未展开的一个儿子，并更新路径上节点的估值

8 步：上一步中，第二个节点的估值被更新了，现在比第五个节点小了，所以这次展开第五个节点下未展开的儿子，并更新路径上节点的估值

蒙特卡洛搜索树以增量不对称的方式构建

每次迭代按照一定的策略**选择（select）**一个节点（如贪心树搜索），然后从所选择的节点进行**模拟（simulate）**并根据结果更新搜索树（如随机法）

用贪心树搜索算法找一个要搜索节点，然后对该节点用纯随机法估值并更新它轨迹上节点的估值；

然后根据更新过的估值、再用贪心树搜索算法找一个节点继续展开，以此反复；

基本的 MCTS 就是迭代地构建一棵贪心搜索树，直到一定的限定预设条件达到才停止迭代；

常见的限定预设条件包括时间、内存或迭代次数等。

每次迭代更新估值包含四个步骤：

选择 select、扩张 expand、模拟 simulate 和反向传播 backup

搜索得越久、估值越准，但需要限定一下时间/步数

1、**选择（Selection）**：从根节点开始，递归地使用子节点树策略不断向树的下方延伸，直到一个非终止且未访问的子节点。

2、**扩张（Expansion）**：根据可以选择的动作，添加一个（或多个）子节点来展开树。

树策略（tree policy）是 MCTS 选择节点策略的统称

树策略需要平衡探索和开发的关系：

探索（exploration）：查看还没有充分了解的区域

利用（exploitation）：使用已被探索且预期效果最好的区域

3、**模拟（simulation）**：开始于被选择的节点，基于一定的**默认策略（default policy）**进行选择动作、状态转换，直到达到终止状态。蒙特卡洛搜索树根据模拟结果进行更新。

4、**反向传播（backup）**：包括添加与被选择节点相对应动作的子节点，以及其祖先节点的统计更新。

树策略（Tree Policy）：在选择和扩张阶段，从已有的搜索树中选择并创建叶节点的选择机制

默认策略（Default Policy）：在模拟阶段，从一个非终止状态不断进行游戏并得到一个价值评估的选择机制

4.6.4 UCT 算法

伪代码：

function MctsSearch(s_0)

  create root node v_0 **with state** s_0

  while within computational budget do（时间限制或步数限制）

    $v_1 \longleftarrow$ **TreePolicy**(v_0)

    $\triangle \longleftarrow$ **DefaultPolicy**($s(v_1)$)（实际终局的得分）// Backup($v_1, \triangle$)（反向传播）

  return BestChild(v_0)（返回最佳）

function TreePolicy(v)

  while v is nonterminal **do**

    if v not fully expanded **then**

      return Expand(v)

else

$v \longleftarrow \text{BestChild}(v, c_p)$

return v

function Expand(v)

choose $a \in$ untired 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 $argmax_{v' \in childrenofv} \frac{Q(v')}{N(v')} + c\sqrt{\frac{2lnN(v)}{N(v')}}$

上式分为利用和探索两部分

利用 (exploitation):$N(v')$ 是这个儿子节点被访问过的次数，$Q(v')$ 是这个儿子节点的估值，Q/N 就是这个儿子的胜率

探索 (exploration): 这个儿子这个儿子和父亲访问次数的比率，儿子访问得少则表示这个儿子被探索得不够，让它的值大一些

function DefaultPolicy(s)

while s **is non-terminal do**（随机法：如果不是叶节点/终局，就敌我双方随机动作，走到终局的到真实得分）

choose $a \in A(s)$ **uniformly at random**

$s \longleftarrow f(s, a)$

return reward for state s

function Backup($v, \triangle$)

while v is not null **do**

$N(v) \longleftarrow N(v) + 1$（访次数 +1）

$Q(v) \longleftarrow Q(v) + \triangle(v, p)$（更新估值分数，p 是玩家）

$v \longleftarrow$ parent of v

4.7 AlphaGo 围棋介绍

蒙特卡洛树搜索是 AlphaGo 的主框架，AlphaGo 中用到的估值函数是通过监督学习和强化学习方法得到的，这些方法在课程后续内容中讲授

4.7.1 简介

1、实现的难度

每局大约有 b^d 个可能情况

b：每步可能的动作数目，d：整局平均的步数

国际象棋：$b \approx 35, d \approx 80$

围棋：$b \approx 250, d \approx 150$

围棋的局面的可能情况数目极大!

5 强化学习基本思想

5.1 生物学基础

强化理论：斯金纳箱

· 通过奖励和惩罚的方式可以改变智能体的行为方式

· 随机奖励可以使智能体上瘾

5.2 问题模型和求解过程

5.2.1 问题模型、求解过程

· 环境（Environment）-智能体与之交互的对象，描述了问题模型

1、初始状态 S_0(state)

2、当前玩家 C(current player(s))

3、动作 A(action): 智能体在某个状态下的合法动作集合

4、状态转移 P(transition):$P(S_{t+1}|S_t, A_t)$ 用以表示环境

衡量一个环境的复杂程度：某个状态下，智能体采取某个动作后，转移到下一状态的状态转移模型。可能到达的所有状态构成了**状态空间（state space)**，所有状态下可行动作，构成**动作空间(action space)**；

5、终止状态 S_T(terminate state)

6、奖励 R(reward):$R_t \longleftarrow S_t, A_t$: 某个状态下，智能体采取某个动作后得到的分数

· 智能体（Agent）-学习者和决策者（问题的解）

· 策略函数 $\pi : A_t \longleftarrow \pi(S_t)$ 用以表示智能体

1、状态 s 到动作 A 的映射关系，给出了智能体在状态 S 下如何选择动作 A 的决策方法

2、注意：策略 π 是全局性的，任何状态下都要能给出动作选择（智能体和世界打交道的整体模型）

· 目标（问题的解）

寻找**最优策略** π，使得从初始状态 S_0 到终止状态 S_T 的累积收益 $G(gain) = \sum_{i=1}^{T} R_i$ 最大

以井字棋的问题模型为例：

假设：敌人用的是确定性的策略（**未必最优**）

基本思想：在和敌人的对弈中逐步找到一个好的对敌策略

1、初始状态 S_0：空棋盘

2、当前玩家 C：轮到下子的一方，（也可以把对手建模在环境里，每次状态转移返回的状态是对手已经落子后的状态，这样游戏就是单人游戏）

3、动作 A: 落子到当前为空的位置

4、状态转移 P: 落子之后的棋盘状态

5、终止状态 S_T: 棋盘满或一方获胜

6、奖励 R: 终止状态，胜者 $+1$，负者-1. 战平双方均为 0，其他状态为 0

井字棋问题的解：

· 策略函数 π:

使用状态估值表，每个状态一个入口，记录从该状态出发下到终局的胜率，根据状态估值表选择动作

学习策略 π_1: 大概率选择估值最高的下一个状态，小概率随机选择一个动作（探索）

目标策略 π^*: 每次选择通往估值最高的下一个状态（贪心）

· 目标（问题的解）：

最优策略 π^*，使得智能体从初始状态 S_0 下到最终的效率/胜率最大

1、第一步：建立状态估值表（值函数表）

· 对于井字棋而言，因为状态数少（状态空间小），可用表格存下估值

· 每个状态一个估值，估值表示这个状态到最终的胜率

· 整个表是值函数

初值：（根据游戏规则）

· 三个 X 连成一线的状态，价值为 1，因为我们已经赢了。

· 三个 O 连成一线的状态，价值为 0，因为我们已经输了。

· 其他状态的值都为 0.5，表示有 50% 的概率能赢

2、第二步，和对手玩很多次

· 利用：大概率贪心选价值最大的地方下；

· 探索：偶尔随机地选择以便探索之前没有探索过的地方；

· 利用和探索要平衡；

· 值函数表决定了我们的策略，改进值函数表就改进了策略;

3、第三步，边下边修改状态的值，使得它更接近真实的胜率

$$V(S_t) \longleftarrow V(S_t) + \alpha[V(S_{t+1}) - V(S_t)]$$

初值：只有终局的价值是正确的，中间局面的价值都是估计值 0.5；

· 过程中：状态价值从后面向前传导；

· 分析：假设我们一直在一条路径上反复走，每走到终点一次，终局价值至少向上传导一步，走多了终将把这个终局的输赢带到最上面的初始节点，于是我们在初始节点就会知道最后的输赢；

· α: 是一个小的正的分数，称为步长参数，或者学习率;

· 通过学习 $V(s)$ 可以得到策略最优策略 π^*;

小结：

1、与上节课 minimax 相比，不再假设对手使用最优策略；

2、将对手建模在环境里；每次采取动作后面临的状态都是对手执行完它的动作后的新状态；（也可以建模成多智能体博弈问题，有一个对手决策模型，轮到对手落子时让对手模型决策）

3、用值函数表存储状态估值/值函数表 $V(s)$

4、通过不断对弈更新值函数表

5、根据值函数表、可以得到贪心选最优动作 π^*

5.2.2 问题模型的泛化

环境：状态转移模型 P 和奖励 R

· 状态转移不一定是确定性的，可以按**概率状态转移**

· P: 状态转移函数 $< S, A, S- > \longrightarrow R^+ >, P(s,a,s') = Pr[s'|s,a], s, a$ 是当前状态和动作，s' 是下一状态;

· 对于任意 s, a，有 $\sum_{s'} P(s,a,s') = 1$

· 奖励也不一定是确定性的，可以是一个**概率奖励**

· R: 奖励函数 $< S, A, R^+ > \longrightarrow R^+, R(s,a,r) = Pr[r|s,a], s, a$ 是当前状态和动作，r 是奖励

· 对于任意 s, a，$\sum_r R(s,a,r) = 1$

智能体：策略 π 和累积收益 G

策略 π 给出的动作选择可以是确定的，也可以是一个概率分布

· π: 策略函数 $< S, A > \longleftarrow R^+$，$\pi$ 描述状态 s 下采取动作 a 的概率，$\pi(s,a) = Pr[a|s], s$ 是当前状态,a 是当前状态下的可选动作

· 对于任意 s,$\sum_a \pi(s,a) = 1$

· 折扣因子 $\gamma : (0 \leq \gamma \leq 1)$，**描述未来收益的重要程度**

· 累积收益 $G = R_1 + \gamma R_2 + \gamma^2 R_3 + ... = \sum_{i=1}^{T} \gamma^{i-1} R_i (T$ 可以是有限的也可以是无限的)

· 上面这个形式的好处取最大的 γ_i，全部替换为 γ_i 后可以使用等比数列求和公式

· 描述对于某个 $s_1, a_1, s_2, a_2, s_3, a_3, ...$ 状态动作序列的累积收益

5.2.3 策略的评估和最优策略

策略 π 的好坏, 用状态价值 V_π 来评估:

· 状态价值 $V_\pi(s)$ 表示从 s 出发执行策略 π 能获得的累积收益;

· 结束状态（如果有）的价值，总是零

$$v_\pi(s) = E_\pi[G_t|S_t = s] = E_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s], \forall s \in S$$

显然从同一个状态 S 出发，V_π 越大，π 越好；使得 V 最大的 π 就是最优策略，记作 π^*，执行 π^* 得到的价值，就是最优价值，记作 V^*

状态价值 V_π 和动作价值 Q_π

状态价值 $V_\pi(s)$ 表示从 s 出发执行策略 π 能获得的累积收益

$$v_\pi(s) = E_\pi[G_t|S_t = s] = E_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s], \forall s \in S$$

动作价值 $Q_\pi(s,a)$ 表示从 s 出发并做动作 a，之后执行策略 π 能获得的累积收益，**有些时候计算动作价值更方便**

$$Q_\pi(s,a) = E_\pi[G_t|S_t = s, A_t = a] = E_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s, A_t = a]$$

问题 1：用 Q_π 表示 v_π:$v_\pi(s) = \sum_{a\in A(s)} \pi(a|s)Q_\pi(s,a)$

问题 2：用 v_π 表示 Q_π：$Q_\pi(s,a) = \sum_{s\in S}\sum_{r\in R} p(s',r|s,a)[r + \gamma v_\pi(s')]$

其中 s' 为下一时刻的状态，r 为一步时的及时奖励

5.2.4 强化学习的任务

得到最优 V^* 或者 Q^* 就能得到最优策略 π^*

· 算出来

  · 使用各种方法探索出 V^* 或者 Q^*

· 存起来

  · 状态多时，查找表**保存**所有状态的价值不现实

  · 用**带参数的函数**来保存 $V_\pi(s)$ 和 $Q_\pi(s,a)$(参数数目小于状态数)

    · 学习过程中我们会调整参数，使之更符合观察到的实际收益。

    · 学习效果取决于带参数的近似函数的好坏（**非常重要的问题**）

· **后面也会讲到直接用函数模拟策略 $\pi(a|s)$ 的方法**

智能体寻找最优策略的路径:

1、智能体使用策略 π_0（开始可能是随机的）与环境交互，产生经验（Experience）

2、智能体根据经验改进 π_0 得到 π_1，再用 π_1 与环境交互以期获得更大的 G

3、如此往复，直到得到最（更）好的 π

5.3 寻找最优策略的几种思路

5.3.1 多臂老虎机

· k 个可行动作（选择第 i 号老虎机）

· 每次选择一个动作，获得一个数字化的奖励

· 每个动作的奖励服从一个确定的正态分布

· 目标是最大化一段时间的总收益的期望, 例如 1000 个金币的总收益;

5.3.2 计算动作价值 $q(s,a)$

没有摇过的有一个缺省值：0

时间 t 时选的动作为 A_t，对应的奖励为 R_t

动作 a 的价值 $Q^*(a) \approx E[R_t|A_t = a]$

根据大数定理，无限次后 $Q_t(a)$ 收敛至 $Q^*(a)$

计算动作价值的一般形式：

$NewEstimate \longleftarrow OldEstimate + StepSize[Target \longleftarrow OldEstimate]$

一个简明的伪代码：

Initialize,for $a = 1$ to k:

$Q(a) \longleftarrow 0$

$N(a) \longleftarrow 0$

Loop forever:

A=$\begin{cases} R \longleftarrow bandit(A) \end{cases}$

$N(A) \longleftarrow N(A) + 1$

$Q(A) \longleftarrow Q(A) + \frac{1}{N(A)}[R - Q(A)]$

6 马尔科夫决策过程和动态规划

6.1 马尔科夫决策过程-Markov decision process(MDP)

6.1.1 $MDP < S, A, P, R, \gamma >$

马尔科夫决策过程：

1、状态集合：S

2、动作集合：A

3、状态转移函数 $P :< S, A, S > \longrightarrow R^+, P(s, a, s') = Pr[s'|s, a], s$ 和 a 是当前状态和动作，s' 是下一状态

4、奖励函数 $R :< S, A, R^+ > \longrightarrow R^+, R(s, a, r) = Pr[r|s, a], s$ 和 a 是当前状态和动作，r 是奖励

6.1.2 马尔科夫性

马尔科夫性：在当前状态 S 下，状态转移模型 P，奖励函数 R 都与 S 之前的状态和动作无关

有限马尔科夫过程是有限的状态 S，动作 A 和奖励 R 的集合

强调几个重要的概念：

1、时间步 t 以来的累计收益值：

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

2、状态价值函数 $v_\pi(s)$：

$$V_\pi(s) = E_\pi[G_t|S_t = s] = E_\pi[R_{t+1} + \gamma R_{t+2} + ...|S_t = s]$$

3、动作价值函数 $Q_\pi(s,a)$：

$$Q_\pi(s,a) = E_\pi[G_t|S_t = s, A_t = a] = E_\pi[R_{t+1} + \gamma R_{t+2} + ...|S_t = s, A_t = a]$$

我们引入一个情景：假设由状态 S_1 可以分别通过动作 a_1, a_2 得到状态 s_2, s_3，那么当存在马尔科夫性时，计算 $V_\pi(s_1)$ 只需要向下看一层，也就是说

$$V_\pi(s_1) = \sum_a \pi(a|s_1) \sum_{s',r} p(s', r|s_1, a)[r + \gamma V_\pi(s')]$$

$V_\pi(s_2)$ 和 $V_\pi(s_3)$ 与 s_1 没有直接关系

6.1.3 贝尔曼方程

1、Bellman 期望方程

$$V_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma V_\pi(s')]$$
$$Q_\pi(s,a) = \sum_{s',r} p(s', r|s, a)[r + \gamma V_\pi(s')], \forall s \in S, a \in A(s), s' \in S^+$$
$$V_\pi(s) = \sum_a \pi(a|s) Q_\pi(s,a)$$

马尔科夫性保证了上述的递推方程成立

2、状态价值和最优策略

定义：$V^*(s) = max_\pi V_\pi(s)$(所有可能的策略下，对当状态 S 下最优的状态价值)

状态价值（等价于最优状态价值）就是从状态 S 出发所能获得的最大累积收益

最优策略 π^* 是从状态 S 出发执行该策略可以获得状态价值的策略

状态价值只有一个，最优策略可以有多个纯策略或者最优纯策略的任意组合

马尔科夫性意味着，如果一个 π 是 S_1 的最优策略，它一定也是 S_2 和 S_3 的最优策略

3、动作价值和最优策略

定义：$Q^*(s,a) = max_\pi Q_\pi(s,a)$(所有可能的策略下，对 S 和 A 下最优的动作价值)

动作价值（等价于最优动作价值）就是从状态 S 出发，采取动作 a，所能获得的最大累积收益

最优策略是从状态 S 出发，采取动作 a，之后执行该策略可以获得动作价值的策略

动作价值只有一个，最优策略可以有多个纯策略或者最优纯策略的任意组合

马尔科夫性意味着，S_2 和 S_3 的最优策略，就是 S_1, a_2 的最优策略，也是 S_1, a_1 的最优策略

4、Bellman 最优方程

$$V^*(s) = max_\pi V_\pi(s)$$

$$Q^*(s,a) = max_\pi Q_\pi(s,a)$$

$$V^*(s) = max_a \sum_{s',r} p(s',r|s,a)[r + \gamma V^*(s')] = max_a Q^*(s,a)$$

$$Q^*(s,a) = \sum_{s',r} p(s',r|s,a)[r + \gamma max_a Q^*(s',a')], \forall s \in S, a \in A(s), s' \in S^+$$

最优价值只有一个，最优策略可以有多个纯策略或者最优纯策略的任意组合

6.2 P,R 已知，用动态规划方法求解最优策略

理查德 · 贝尔曼 (Richard Bellman): 美国数学家

首先提出“维数灾难”：维度增加时，问题规模呈指数级增长

Bellman 方程：用于最优控制理论。解决多阶段决策过程 (multi-stage decision process)，是动态规划的数学基础

动态规划 (Dynamic Programming)：最早用于最优控制理论中，研究问题的优化方法。之后被抽象简化放入算法框架，幸运地成为了算法课程考察的一个知识点

Bellman 论文里对动态规划的说明 (1954):

动态规划理论一开始是为了解决**多阶段决策过程 (multi-stage decision process)** 中的数学问题：我们有一个物理系统，它的**状态 (state)** 被描述为一组状态变量。**决策 (decisions)** 等价于状态变量的**转移 (transformations)**。决策与状态转移是互相独立的，在前一段决策的结果的基础上实施后面的决策，整个过程的目标是最大化最终状态参数的相关函数。

考虑一个中间状态, 做决策时，我们并不关心如何从初状态到中间状态的过程，之后的决策都只从这个中间状态开始。这就叫做**无后效性（马尔可夫性产生的）**

最优原则 (principle of optimality)：

一个最优策略有这样的性质——无论初始状态和最初的几个决策是什么，剩余决策一定构成一个与之前决策产生的状态相关的一个最优策略。

形式化描述：一个策略 $\pi(a|s)$ 能达到状态 s 下的最优值 $v_\pi(s) = v_*(s)$，当且仅当：对于从状态 s 出发可达到的任意状态 s'，策略 π 能达到新状态 s' 的最优值 $v_\pi(s') = v_*(s')$

6.2.1 策略迭代 (Policy Iteration)-通过策略改进得到最优策略

1、策略估值 (Policy Evaluation)

定义：对于任意策略 π，计算在此策略下的状态值函数 V_π

(1) 策略估值中 Bellman 方程的赋值形式

$$V_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)[r + \gamma V_\pi(s')] \quad (1)$$

(2) 迭代更新规则：v_π 是这个更新规则的不动点

$$v_{k+1}(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_k(s')] \quad (2)$$

k 代表阶段，一个阶段里所有状态都有一个估值

注：(1) 式中的 v_π 下标表示是对策略 π 的准确稳定状态值函数

(2) 式中的 v_k 表示迭代策略估值过程中，第 k 次迭代得到的值函数，依据的策略是同一个 π ，此时值函数可能不准确

2、迭代策略估值 (iterative policy evaluation)

完全回溯 (full backup):

依据被评估策略 π 所有可能的一步转移，用 $v_k(s)$ 依次计算 $v_{k+1}(s), \forall s \in S$

迭代停止条件：

测试 $max_{s \in s}|v_{k+1}(s) - v_k(s)|$，当这个量足够小时停止

迭代内循环计算：

同步迭代 (Synchronous iteration):

已有 $v_k(s_1), v_k(s_2), ..., v_k(s_n)$

用以上值依次计算 $v_{k+1}(s_1), v_{k+1}(s_2), ..., v_{k+1}(s_n)$

需要双数组分别存储旧值和新值

异步迭代：

用 $v_k(s_1), v_k(s_2), ..., v_k(s_n)$ 计算 $v_{k+1}(s_n)$

用 $v_{k+1}(s_1), v_k(s_2), ..., v_k(s_n)$ 计算 $v_{k+1}(s_2)$

用 $v_{k+1}(s_1), v_{k+1}(s_2), ..., v_k(s_n)$ 计算 $v_{k+1}(s_3)$

...

只需单数组原位更新，可以更快地收敛。之后会用到类似思想

3、迭代策略估值伪代码

Input π,the policy to be evaluated

Initialize an array $V(s) = 0$,for all $s \in S^+$

Repeat

$\triangle \longleftarrow 0$

For each $s \in S$:

$v \longleftarrow V(s)$

$V(s) \longleftarrow \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$

$\triangle \longleftarrow max(\triangle, |v - V(s)|)$

until $\triangle < \theta$(a small positive number)

Output $V \approx v_\pi$

4、v_π 的存在性和唯一性保障条件

v_π 的存在性和唯一性保障条件:$\gamma < 1$

如果是无限步，$\gamma < 1$ 可以想象成远处传递上来的值为 0 或者依据策略 π，所有状态最终能保证终止**如果是有限步比照 Bellman-Ford 算法**

迭代策略估值:

在上述存在性和唯一性保障条件下

可以确保序列 $\{v_k\}$ 在 $k \to \infty$ 时能收敛，k 是迭代次数，且收敛于 v_π

也即在迭代足够多次之后，能得到一个稳定的关于策略 π 的值函数 v_π。证明留待后面的值迭代收敛性证明

计算依据的是同一个策略 π

5、策略提升 (Policy Improvement)

定义：在原策略基础上，根据原策略计算得到的值函数，贪心地选择动作使得新策略的估值优于原策略，或者一样好

可以根据状态估值计算动作估值:

$$q_\pi(s,a) = \sum_{s',r} p(s',r|s,a)[r + \gamma v_\pi(s')]$$

贪心选择动作产生新策略:

$$\pi'(s) = argmax_a q_\pi(s,a) = argmax_a \sum_{s',r} p(s',r|s,a)[r + \gamma v_\pi(s')]$$

总结:

在某个状态 s 下，找到一个比 π 好的 q，在所有遇到状态 s 时，用 q 替代原来的 π，得到新的 π'

因为其他状态的动作都保持不变，对于所有状态，$V_{\pi'}(s)$ 一定比 $V_\pi(s)$ 要好，策略得到提升

进而考虑在所有 s 下，贪心选择 q 最好的动作，完成一次更有成效的策略提升

如果有多于一个并列最好的 q，可以在这些 q 里随机选

策略提升后，状态价值对于新的策略估计的就不准确了，需要重新估计……

6、策略迭代 (Policy Iteration)

定义：交替进行迭代策略估值和策略提升，在有限步之后找到最优策略与最优值函数

以上交替进行会得到策略序列π，由于之前已经证明策略提升中新策略一定优于旧策略，或者一样好，故策略序列单调更优

$$\pi_0 \longrightarrow v_{\pi_0} \longrightarrow \pi_1 \longrightarrow v_{\pi_1} \longrightarrow \pi_2 \longrightarrow ... \longrightarrow \pi_* \longrightarrow v_*$$

有限 MDP 只有有限种策略，在进行有限步迭代后，一定能收敛得到最优策略与最优值函数

在有限 MDP 问题中，S，A，R 的取值都有有限个

7、策略迭代算法伪代码:

1.Initialization

$V(s) \in R$ and $\pi(s) \in A(s)$ arbitraily for all $s \in S$

2.Policy Evaluation

Repeat

$\triangle \longleftarrow 0$

For each $s \in S$

$v \longleftarrow V(s)$

$V(s) \longleftarrow \sum_{s',r} p(s', r|s, \pi(s))[r + \gamma V(s')]$

$\triangle \longleftarrow max(\triangle, |v \longleftarrow V(s))$

until $\triangle < \theta$(a small positive number)

3.Policy Improvement

$policy - stable \longleftarrow true$

For each $s \in S$

$old - action \longleftarrow \pi(s)$

$\pi(s) \longleftarrow argmax_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$

If $old - action \neq \pi(s)$,then $policy - stable \longleftarrow false$

If $policy - stable$,then stop and return $V \approx v_*$ and $\pi \approx \pi_*$;else go to 2

8、策略提升结束条件:

假设新策略 π' 和旧策略 π 一样好，则有 $v_\pi = v_{\pi'}$

与 Bellman 最优方程相同，因此 $v_{\pi'}$ 一定是当前最优状态值函数 v^* 则 π' 和 π 一定都是最优策略

6.2.2 值迭代 (Value Iteration)-通过求最优状态价值得到最优策略

1、策略迭代的问题:

策略估值耗费时间

可以提前截断估值

2、一种极端的矫正方式:

只做一次估值、不需要估值收敛就开始策略提升, 这就是值迭代方法

$$v_{k+1}(s) = max_a E[R_{t+1} + \gamma v_k(S_{t+1})|S_t = s, A_t = a] = max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_k(s')]$$

所以除了策略迭代，还可以从另一个角度看值迭代

既然改进策略就是贪心的选值最大的，那不如就直接求 v 的最大估值，求到最优值，再贪心

值迭代是另一种计算最优策略的方法.

基本思想是:

定义一个状态的价值是从它出发所能达到的最优价值

计算所有状态所能达到的最优值,

贪心地选择能获得最优价值的动作–最优动作

3、值迭代算法的收敛性分析:

我们说值迭代算法最终会收敛到贝尔曼最优方程的唯一解，为什么呢?

这里引入一些数学思想:

一个基本概念是 contraction 压缩，简单讲一个压缩变化是一个带有一个参数的函数，当我们给它两个不同的输入时，它会输出两个距离更近的数值（常数倍于原距离）。
例如“除以 2”是一个压缩变换, 因为当我们把两个数都除以 2，他们间的距离也是原来的一半。注意，“除以 2”这个函数有一个**不动点**, 就是 0, 这个点除以 2 还是它自己。
从这个例子中，我们可以得到关于压缩变化的两个重要性质：

压缩变化只有一个不动点; 如果有两个，那么输入这两个不动点，输出的值的距离不会改变，所以它不是压缩变换；

当函数作用于任何参数，它的值应该离不动点更近了 (因为不动点的值不变), 所以反复使用压缩变换，会最终到达不动点；

现在我们来把贝尔曼更新（Bellman update）看做操作 B，并把它反复用于每个状态
令 V_i 表示在第 i 次迭代后，所有状态的估值构成的向量。贝尔曼更新可以写作:

$$V_{i+1} \longleftarrow BV_i$$

接下来我们需要一种方法来度量两个向量之间的距离
我们使用 max norm, 它将向量的“长度”定义为向量中最大的元素的绝对值:

$$||V|| = max_s|V(s)|$$

在这种定义下, 两个向量的距离,$||V_{i+1}-V_i||$, 是向量中对应元素差值最大的那个
令 V_{i+1} 和 V_i 是一次迭代前后状态的估值，我们的结论是每次迭代的状态估值变化量都缩小至少 γ 倍
贝尔曼更新是一个系数为 γ 的值向量空间的压缩变换
所以，当 $\gamma < 1$ 时，基于压缩变换的性质，值迭代总能收敛到贝尔曼方程的唯一解
我们也用压缩变换的性质分析收敛到一个解的速率。
我们把 $||BV_{i+1}-BV_i|| \leq \gamma||V_{i+1}-V_i||$ 中的 V_i 换成真正的价值 V，此时 $BV = V$
那么我们会得到如下不等式

$$||BV_i-V|| \leq \gamma||V_i-V||$$

如果我们把 $||V_i-V||$ 看成估计 V_i 的误差, 我们会发现每次迭代，误差至少会减小至原来的 γ 倍，这说明策略估值迭代是指数收敛的

6.2.3 广义策略迭代

1、异步动态规划 (Asynchronous Dynamic Programming):
收敛条件:
$0 \leq \gamma < 1$
则当给定所有状态在序列s_k 中出现无穷次，能保证异步收敛到 v_*
优点：使得计算和实时交互的混合更加简单

在运行迭代动态规划算法的同时令一个 agent 也在 MDP 模型中与环境交互

agent 的经历（experience）能用来决定动态规划算法优先回溯哪些状态，比如状态集内和 agent 最相关的一部分状态

来自动态规划算法的最新的值和策略信息能指导 agent 的决策

2、广义策略迭代 (Generalized Policy Iteration) 定义：策略估值和策略提升交互的一般思想，几乎所有强化学习方法都能用广义策略迭代（Generalized Policy Iteration ）GPI 描述

不同粒度表现:

策略迭代中为迭代策略估值，进行同步计算得到稳定的值函数

值迭代中为一轮策略估值，进行同步计算得到不一定稳定的值函数

异步动态规划选择性地计算，进行异步计算

GPI 可以被看作拮抗与协同

拮抗：贪心选择动作产生新策略，会使得值函数发生改变

协同：重新计算值函数使得值与策略相一致

拮抗与协同交替进行从而得到稳定的解决方案：最优策略和最优值函数

6.3 P,R 未知，用采样方法逼近最优策略

6.3.1 Bootstrap

最后，我们可以注意到，计算一个状态的估值是依据它的后继状态的估值的，我们称之为 bootstrap

许多强化学习的方法使用 bootstrap 方法

动态规划方法假设环境已知（P，R 已知）

蒙特卡洛学习方法假设环境未知（P，R 未知），也不使用 bootstrap

时序差分学习方法假设环境未知（P，R 未知），但是使用 bootstrap

环境已知和 bootstrap 是彼此独立的，当然也可以对他们进行有趣组合