北京大学

总复习 – 搜索+强化学习

Solving Problems by Searching

内容主要来自：Artificial Intelligence A Modern Approach Stuart Russel Peter Norvig Chapter 3

主讲人：李文新 2023年春

全局搜索

1. 问题的定义及问题的解
   - 问题描述模型
2. 通过搜索对问题求解
   - 搜索树、搜索算法框架、搜索算法的性能评价
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
4. 有信息搜索（启发式搜索）
   - 贪婪最佳优先搜索、A* 搜索、启发式函数

全局搜索

1. 问题的定义及问题的解

- 问题描述模型

关键点一：把问题用一个统一的模型表示清楚，就能够用通用的搜索方法来求解

关键点二：建模很重要

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。
3. 状态转移模型。
4. 目标状态。
5. 路径花费。

一个问题的解 是从初始状态出发到达目标状态的一个动作序列。解的质量可以用路径的花费来度量。最优解是所有解中花费最小的一个。

内容提要

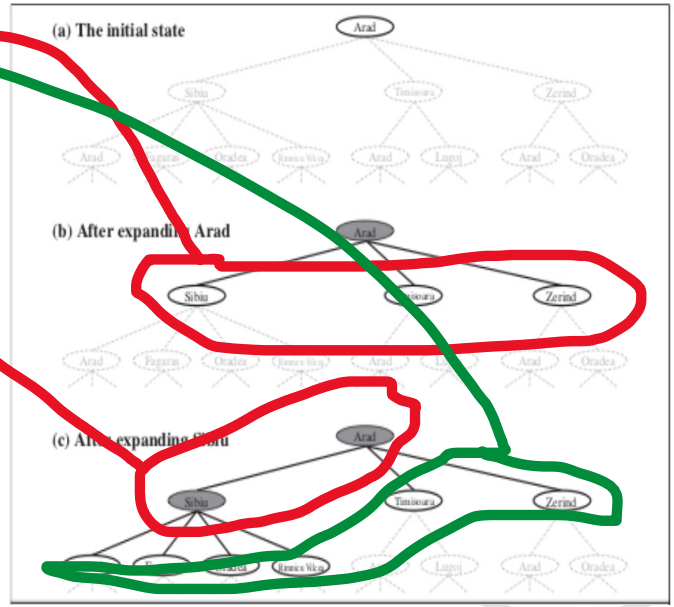
2. 通过搜索对问题求解

- 搜索树、搜索算法框架、搜索算法的性能评价

- 关键点三：分清开节点集和闭节点集
- 关键点四：分清树搜索和图搜索
- 关键点五：搜索算法的效率评价

关键点三：分清开节点集和闭节点集

- 搜索树
- 父节点 parent node 、 子节点 child nodes 、 叶节点leaf node
- 开节点集， frontier，或称 open list
- 闭节点集 closed list ， 或称 explored set
- 搜索策略 search strategy



关键点四：分清树搜索和图搜索

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

树搜索

---

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** the 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

图搜索

关键点五：搜索算法的效率评价

搜索算法的评价：

- 完备性 Completeness: 如果存在解，该算法是否一定会找到解?
- 最优性 Optimality: 该算法是否能够找到最优解?
- 时间复杂度 Time complexity: 算法找到解需要花长时间?
- 空间复杂度 Space complexity: 算法需要多少内存用于搜索?

内容提要

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

4. 有信息搜索（启发式搜索）

- 贪婪最佳优先搜索、A* 搜索、启发式函数

每种算法

出开节点集的顺序

时间复杂度和空间复杂度

实现的技术手段（队列、栈、递归）

关键点六：不同策略的区别仅在于谁先从开节点集出来

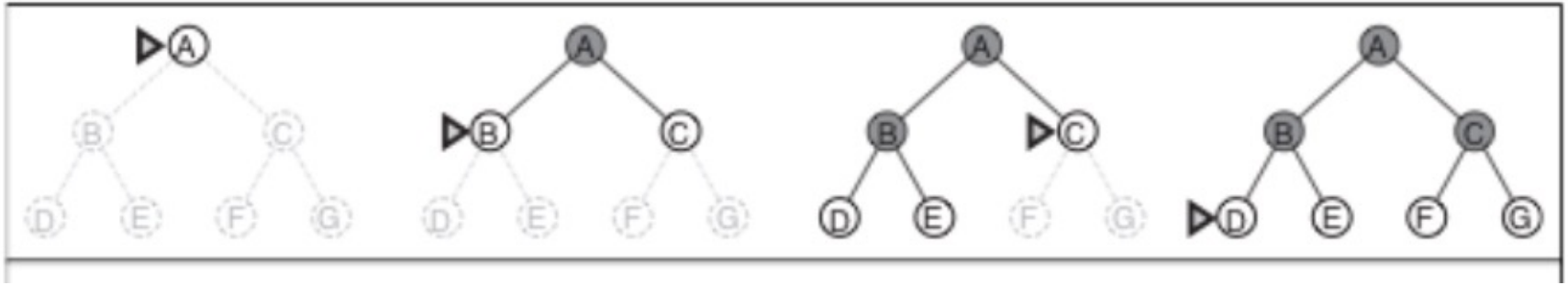
不同的出来顺序使用不同的数据结构实现

关键点七：深度优先更省空间搜的更深

关键点八：从开节点集出来时找到最优解，留下的花费更高

关键点九：h(n)是可采纳的（树搜搜）或者是一致的（图搜索），A*就是最优的

1 宽度优先



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

这里我们可以用一个神奇的数据结构（队列）来实现宽度优先算法

空间复杂度：$O(b^d)$ 树搜索：开节点集大小　　图搜索：所有访问过的节点

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

Figure 3.13 Time and memory requirements for breadth-first search. The numbers shown assume branching factor $b = 10$; 1 million nodes/second; 1000 bytes/node.

宽度优先算法的问题（作业）

1. 在宽度优先搜索中，内存的需求问题要比运行时间问题更严重。
2. 时间问题仍旧是个大问题。

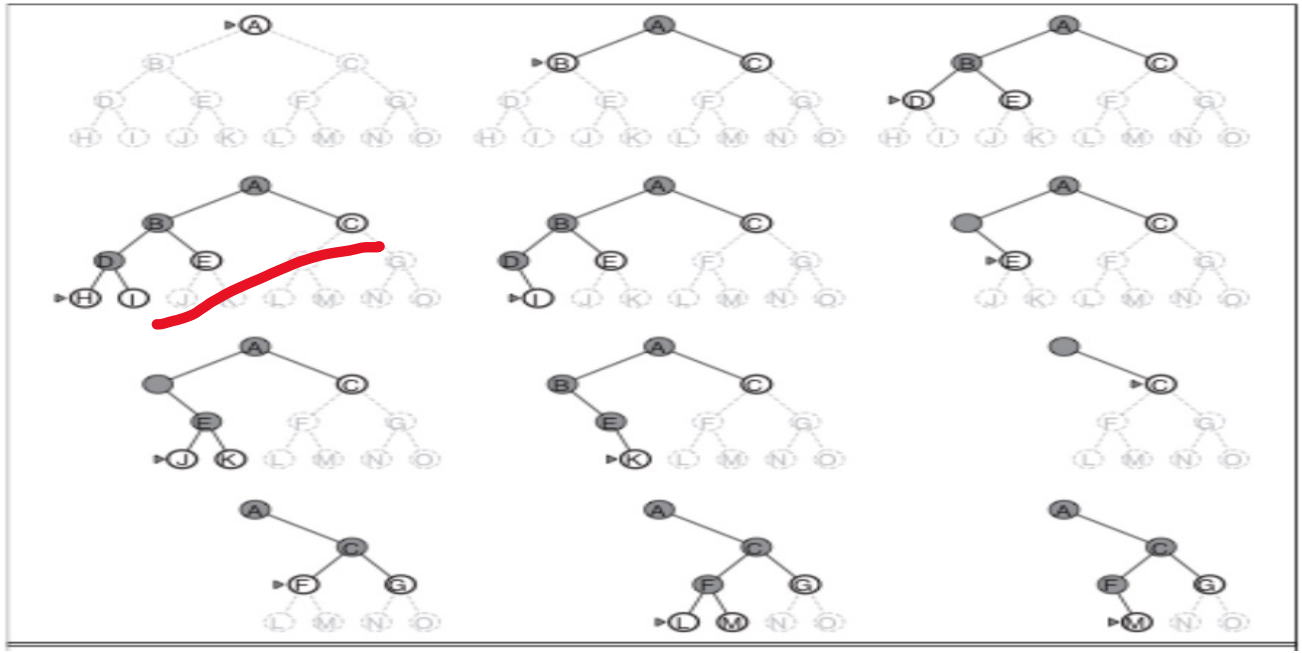
1 宽度优先 伪代码

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
    node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    frontier ← a FIFO queue with node as the only element
    explored ← an empty set
    loop do
        if EMPTY?(frontier) then return failure
        node ← POP(frontier)  /* chooses the shallowest node in frontier */
        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
                if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
                frontier ← INSERT(child, frontier)
```

关键点六：不同策略的区别仅在于谁先从开节点集出来
不同的出来顺序使用不同的数据结构实现

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

2 深度优先



右边这一串
不展开，但
是得记住，
一会儿回来
还得进去，
用栈存储
回来的时候
用倒序

深度优先搜索算法的实现：

1）使用先进后出LIFO的栈，存下一个待展开的点。

2）使用 （recursive function） 递归调用 的方法。

宽度优先和深度优先的效率分析

Criterion	Breadth-First	Depth-First	
Complete?	Yes[a]	No	完备性
Time	$O(b^d)$	$O(b^m)$	时间复杂度
Space	$O(b^d)$	$O(bm)$	空间复杂度
Optimal?	Yes[c]	No	最优性

关键点七：深度优先更省空间
搜的更深

- b 是分支数; d 时最浅的目标所在的深度;
- m 是搜索树的最大深度;
- a 是完备的如果b 是有限的;

一致代价

```
function UNIFORM-COST-SEARCH(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)  /* 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 ← 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
```

- 从起点到n的花费g(n)
- 开节点集 - 优先队列
- 闭节点集 - 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

关键点八：从开节点集出来时找到最优解，留下的花费更高

贪婪最佳和A* 搜索

贪婪最佳和A*使用与一致代价相同的搜索框架，他们出开节点集的顺序都取决于f(n) ，我们定义f(n) =g(n) + h(n) ；

对于一致代价， f(n) = g(n)

对于贪婪最佳， f(n) = h(n)

对于A*, f(n) = g(n) + h(n)

- A* 具有如下性质：
  - 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。

- 极端情况下（满足可采纳或者一致性），
  - 最好即贪心，
  - 最坏（全0）即一致代价。

关键点九：h(n)是可采纳的或者是一致的，A*就是最优的

局部搜索

- 局部搜索概述
  - 问题、思想、挑战
- 局部搜索算法
  - 爬山法
  - 模拟退火算法
  - 局部束搜索
  - 遗传算法
  - 连续空间中的局部搜索

为什么要局部优化算法?

全局搜索方法从初始状态出发，遍历整个动作序列空间，寻找目标状态。

- 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。

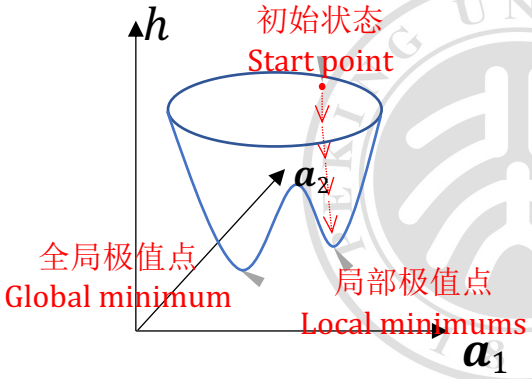
状态估值函数和问题求解过程

- 状态估值函数
  - 描述一个状态是不是好的，
  - 输入是当前状态，
  - 输出是一个评估值。
- 整个问题的求解过程就是根据状态估值函数不断从当前状态移动到估值更低（或更高）的邻居状态，直到到达目标状态的过程。

局部搜索基本思想

- 基本思路

  - 从一个初始状态出发，向更好的邻居状态移动，
  - 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。

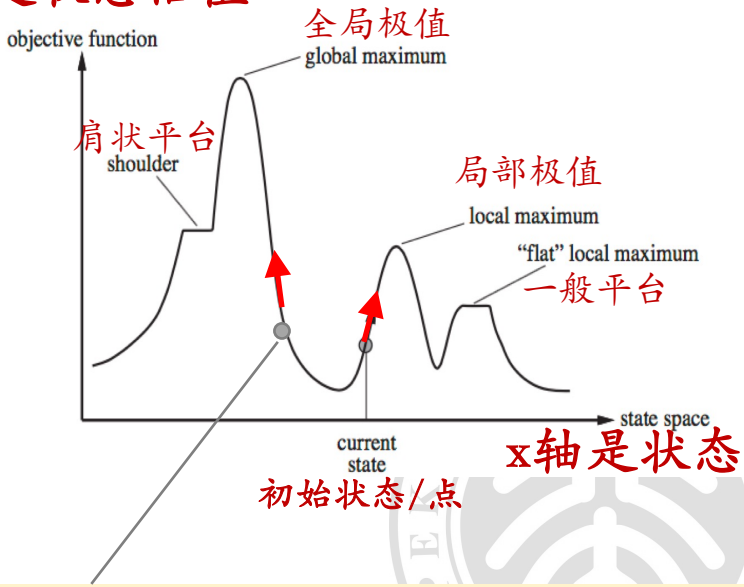


解空间的形状

以状态估值越大越好为例：

- 右图给出了一个一维状态空间的例子。假设状态是一个连续变化的值，用X轴表示，Y轴是每个状态下估值函数的评估值，在X取值范围内的评估值如图所示。
- 在任意一个当前状态下，可以左右移动。
- 图中有几个概念要知道：局部极值、全局最值、一般平台、肩状平台。

y轴是状态估值



如果初始状态在这，就能爬到全局最优值
初始点点位置影响了能不能到全局最优的极值点

局部搜索的挑战和效率分析

- 解空间形状带来的问题
  - 如何跳出局部极值点而奔向全局最优点?
  - 进入平台时如何解决?
- 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。

局部搜索算法

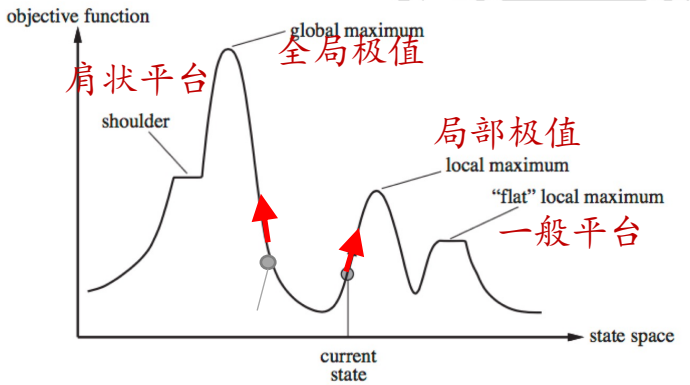
- 爬山法
- 模拟退火算法
- 局部束搜索
- 遗传算法
- 连续空间中的局部搜索
- 八皇后问题不同的建模和求解方式比较

爬山法小结

重点页

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



局部优化算法

- 爬山法
- 模拟退火算法
- 局部束搜索
- 遗传算法
- 连续空间中的局部搜索
- 八皇后问题不同的建模和求解方式比较

爬山法的问题

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。

纯粹的随机游走算法（不使用h(n)的搜索算法）

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。

局部优化算法

- 爬山法
- 模拟退火算法
- 局部束搜索
- 遗传算法
- 连续空间中的局部搜索
- 八皇后问题不同的建模和求解方式比较

上节课的搜索

展开整个搜索树，占用内存太大，分支太多

本节课的爬山法和模拟退火

只用一个节点搜索 – 矫枉过正

问：如何用更多内存加快搜索？

局部优化算法

- 爬山法
- 模拟退火算法
- 局部束搜索
- 遗传算法
- 连续空间中的局部搜索
- 八皇后问题不同的建模和求解方式比较

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

遗传算法

- **遗传算法** Genetic algorithms
  - 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。

局部束搜索，随机束搜索
类似单细胞生物
后代多样性小

遗传算法
类似高级生物
后代多样性大

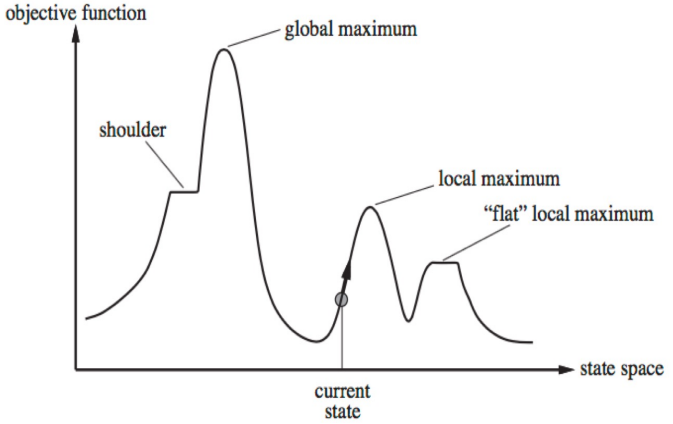
局部优化算法

- 爬山法
- 模拟退火算法
- 局部束搜索
- 遗传算法
- 连续空间中的局部搜索
- 八皇后问题不同的建模和求解方式比较

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？

连续空间中的局部搜索

- 也可以直接使用随机爬山和模拟退火算法。
- 先以一维的情况为例
- 前提
  - 我们有一个状态估值函数
  - 有两个方向可以调整 x 的取值



- 方法一：经验梯度的值
  - 左右各取一个点 x+δ，x-δ，
  - 比较 f(x), f(x+ δ), f(x- δ) 的值
  - 向好的方向移动

方法二：梯度下降，利用函数 f 的性质，判断移动方向

计算 f(x) 在当前点的导数 f’(x)，

目标是最大化则 x+δ f’(x)，最小化则 x − δ f’(x)

有些情况估值函数可能不可微 - 使用经验梯度的值

总结：局部优化算法

- 全局搜索的问题（以A*为例）
  - 指数空间代价
  - 路径长度也是个限制
- 有些问题无需知道搜索路径，只需找到好的解
- 局部优化
  - 模型：初始状态，动作，状态转移（到邻居），目标状态，状态估值函数
  - 思想：只保留一个当前状态，每次向好邻居移动
  - 问题：局部极值和平台
  - 解决方案：模拟退火、随机重启、
  - 拓展（使用更多空间）：二代种群 - 局部束、遗传算法
  - 连续空间：
    - 离散化、梯度下降
- 归根结底：建模还是最重要的

对抗搜索

- 本节课要解决的问题：对抗游戏的决策问题
- 极大极小搜索 MINIMAX
- αβ剪枝
- 不完美的实时决策
- 蒙特卡洛树搜索

双人游戏的零和与非零和

- 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
  - 两个人从第三方赚收益

双人游戏的问题模型

- 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。

本讲针对双人零和完全信息游戏

- 一个游戏可以用下面的元素定义成一个搜索问题：

① S0: 初始状态, 描述游戏开始时的状态。

比之前的搜索方法，多了一个PLAYER元素

② PLAYER(s): 定义在一个局面下，轮到哪个玩家选择动作。

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。

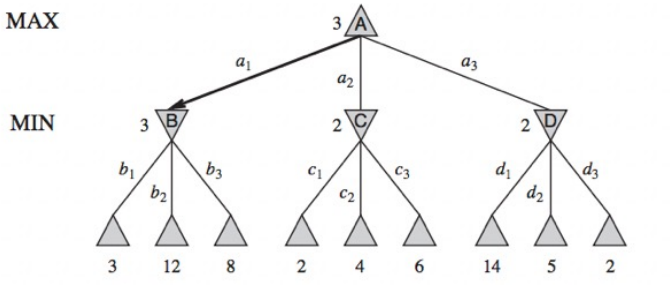
⑥ UTILITY (s, p): 效用函数(目标函数 或者 支付函数), 游戏结束时玩家p 的得分。零和游戏是指所有玩家得分的和为零。 完全信息

内容提要

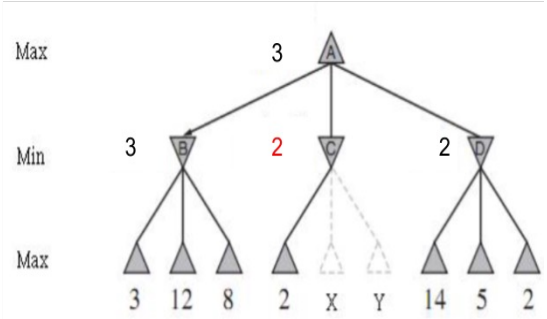
- 本节课要解决的问题：对抗游戏的决策问题
- 极大极小搜索 MINIMAX（最基础）
- αβ剪枝
- 不完美的实时决策
- 蒙特卡洛树搜索
- AlphaGo 围棋介绍

基于MINIMAX算法的对抗搜索

重点页



1. MINIMAX，我一层敌一层的搜索树，找最优解



2. αβ剪枝－无损提升效率，找最优解

```
function ALPHA-BETA-SEARCH(state) returns an action
  v ← MAX-VALUE(state, −∞, +∞)
  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 ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v

function MIN-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

H-MINIMAX(s,d) =

$$\begin{cases} \text{EVAL}(s) & \text{if CUTOFF-TEST}(s,d) \\ \max_{a \in Actions(s)} \text{H-MINIMAX}(\text{RESULT}(s,a), d+1) & \text{if PLAYER}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{H-MINIMAX}(\text{RESULT}(s,a), d+1) & \text{if PLAYER}(s) = \text{MIN}. \end{cases}$$

3. 不完美的实时决策、时间受限时的截断搜索 – 启发式估值替代实际收益

一、要点：

1. 搜索树规模大
2. α β，好节点靠前好
3. 依旧搜不到底，截断，估值

二、问题是：

1. 没有好的启发式函数怎么办？

极大极小算法的效率分析

- 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。

内容提要

- 本节课要解决的问题：对抗游戏的决策问题
- 极大极小搜索 MINIMAX
- αβ剪枝
- 不完美的实时决策
- 蒙特卡洛树搜索
- AlphaGo 围棋介绍

MINIMAX方法在问题：
问题规模增大时树的规模太大
使用αβ剪枝剪掉无用搜索

alpha－beta 剪枝 搜索顺序

- 好的动作选择在最前面，就能进行更多的剪枝。
- 在最好的情况下我们只需要检查 $O(b^{m/2})$ 个节点就能找到最优解。这比 minimax的 $O(b^m)$ 个节点要好多了。这意味着平均每个状态有 Sqrt(b) 而不是b 个选择 — 对于国际象棋, 每步有 6 个选择而不是 35个。
- 从另一个角度看 alpha－beta 算法可以在相同时间内比minimax 算法搜索深一倍的节点。
- 如果不是最好的节点排在最前面，而是随机排布的，对于一个较小的b，需要检查的节点数大致是 $O(b^{3m/4})$ 。
- 对于国际象棋来说，一个简单易行的节点排序算法（例如，先吃，再攻，前进，后退）就可以取得接近2倍于最好情况的效果，即 $O(b^{m/2})$ 。

内容提要

- 本节课要解决的问题：对抗游戏的决策问题
- 极大极小搜索 MINIMAX
- αβ剪枝
- 不完美的实时决策
- 蒙特卡洛树搜索
- AlphaGo 围棋介绍

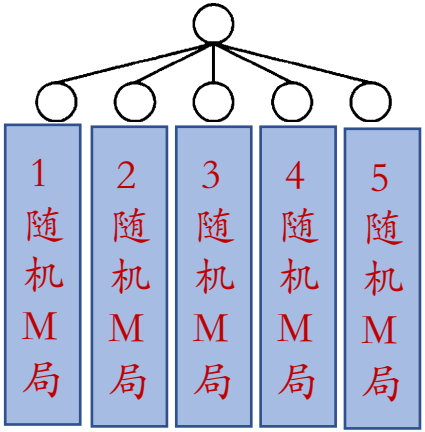
α β 剪枝方法
可以剪掉不影响解的节点，加快搜索速度。但当问题规模增大时，仍旧不能解决搜索树太大的问题

内容提要

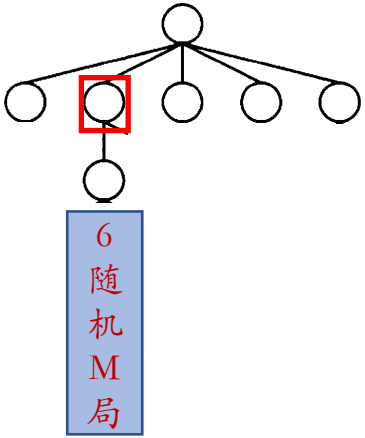
- 本节课要解决的问题：对抗游戏的决策问题
- 极大极小搜索 MINIMAX
- αβ剪枝
- 不完美的实时决策
- 蒙特卡洛树搜索 Monte Carlo
- AlphaGo 围棋介绍

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？

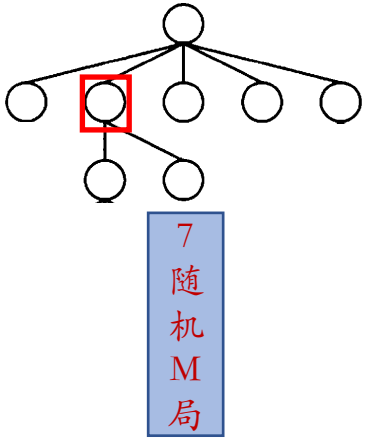
蒙特卡洛树搜索 MCTS 概述



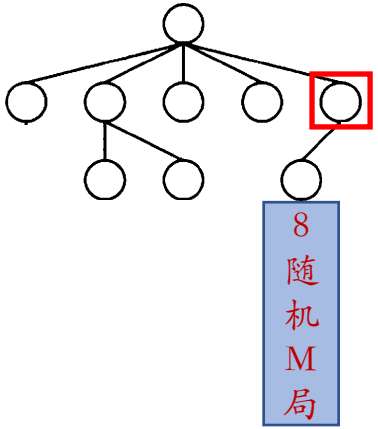
1～5步：最开始每个节点值都不知道，所以都展开、每个儿子随机搜索M次，求估值



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

估值越来越准确

[2] Browne C B, Powley E, Whitehouse D, et al. A Survey of Monte Carlo Tree Search Methods. 2012. IEEE Transactions on Computational Intelligence & Ai in Games. University of London, et al.

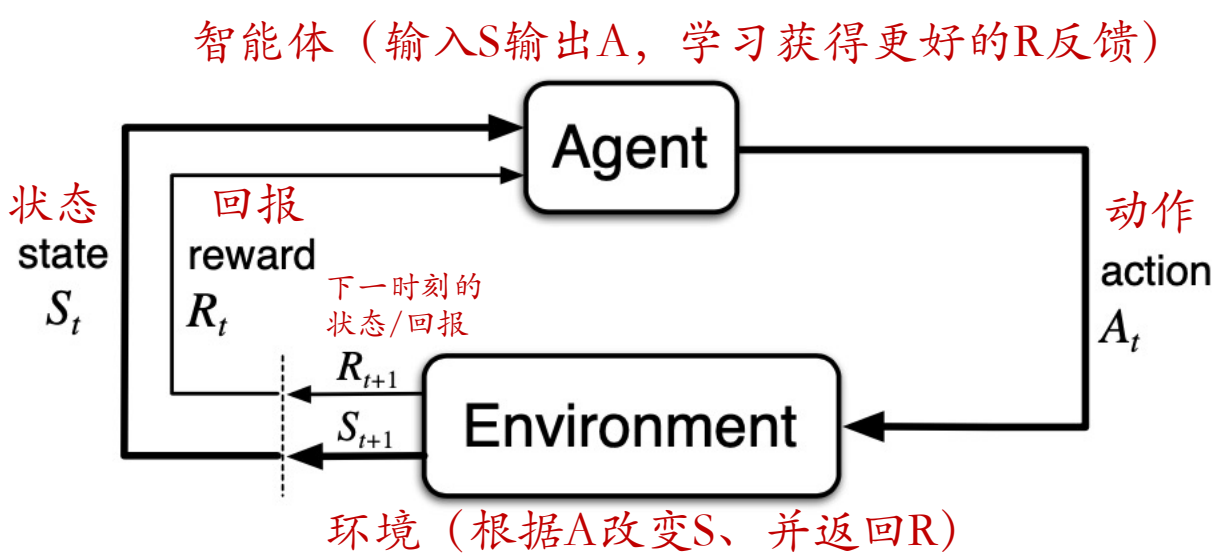
强化学习

1. 马尔科夫决策过程 Markov decision process (MDP)
2. Bellman方程： Bellman期望方程、 Bellman最优方程
3. P，R已知，用动态规划方法求解最优策略π*

   值迭代、策略迭代
4. P，R未知，用采样方法逼近最优策略π*

   蒙特卡洛学习、时序差分学习

强化学习的模型



- 环境（Environment）- 智能体与之交互的对象，描述了问题模型
- 智能体（Agent） - 学习者和决策者

马尔科夫决策过程(MDP)

- 马尔科夫决策过程 Markov decision process (MDP)，当前状态的决策只取决于当前状态，与如何来得到当前状态无关 – 马尔科夫性
- 马尔科夫决策过程的数学定义
  - M = <S, A, P, R, γ>
    - S: 状态集合
    - A: 动作集合
    - P: 状态转移函数 <S, A, S> -> $\mathcal{R}^+$，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
    - R: 奖赏函数 <S, A, $\mathcal{R}^+$ > -> $\mathcal{R}^+$，R(s, a, r) = Pr[r | s, a]，s和a是当前状态和动作，r是奖赏
    - γ: 折扣因子
- 策略的定义
  - π:策略函数 <S, A> -> $\mathcal{R}^+$， π描述的是状态s下采取动作a的概率分布
- 智能体目标
  - 寻找最优的策略π，使得从初始状态s_1到终结状态s_n的期望累积收益$\mathrm{E}[\sum_{i=1}^{n}\gamma^i r_i \,|M, \pi]$最大

几个概念

- G_t是时间步t以来的累积收益值

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

- 状态价值函数 $V_\pi(s)$

$$V_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s] = \mathbb{E}_\pi[R_{t+1} + \gamma R_{t+2} + \cdots | S_t = s]$$

- 动作价值函数 $Q_\pi(s, a)$

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

Formulas referenced from *Sutton, Richard S., and Andrew G. Barto. Reinforcement learning: An introduction. P44,46*

Bellman方程

- Bellman期望方程

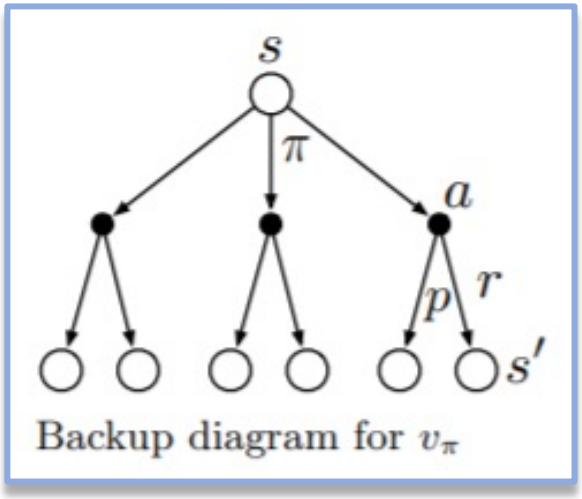
$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)\,[r + \gamma v_\pi(s')]$$

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

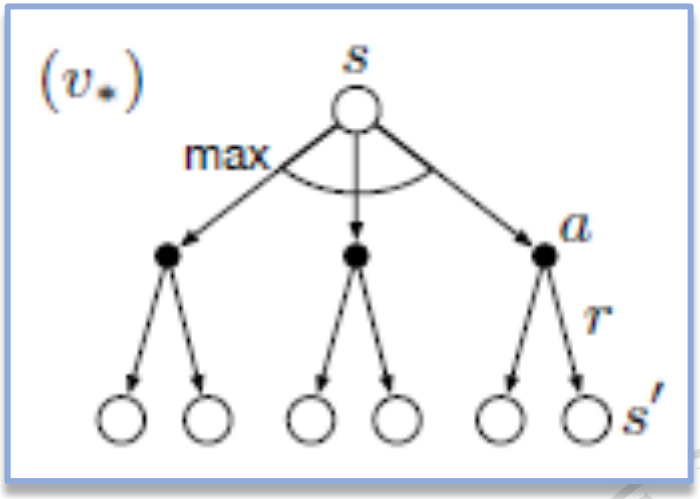
- Bellman最优方程

$$v_*(s) = \max_\pi v_\pi(s)$$
$$q_*(s,a) = \max_\pi q_\pi(s,a)$$
$$\Rightarrow$$
$$v_*(s) = \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a]$$
$$= \max_a \sum_{s',r} p(s',r|s,a)[r + \gamma v_*(s')]$$
$$q_*(s,a) = \mathbb{E}\left[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \,\middle|\, S_t = s, A_t = a\right]$$
$$= \sum_{s',r} p(s',r|s,a)[r + \gamma \max_{a'} q_*(s',a')]$$
$$\forall s \in S, a \in A(s), s' \in S^+$$

Bellman期望方程和最优方程的回溯图



- Bellman期望方程
- 策略迭代方法
  - 策略估值
  - 策略提升
  - 策略迭代



- Bellman最优方程
- 值迭代

Figure referenced from *Sutton, Richard S., and Andrew G. Barto. Reinforcement learning: An introduction. P47,51*

内容提要

1. 马尔科夫决策过程 Markov decision process (MDP)
2. Bellman方程： Bellman期望方程、 Bellman最优方程
3. P，R已知，用动态规划方法求解最优策略π*

   值迭代、策略迭代
4. P，R未知，用采样方法逼近最优策略π*

   蒙特卡洛学习、时序差分学习

值迭代（Value Iteration）

Value Iteration, for estimating $\pi \approx \pi_*$

Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop:
| $\Delta \leftarrow 0$
| Loop for each $s \in \mathcal{S}$:
| $v \leftarrow V(s)$
| $V(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a)\left[r + \gamma V(s')\right]$
| $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

Output a deterministic policy, $\pi \approx \pi_*$, such that
$\pi(s) = \arg\max_a \sum_{s',r} p(s', r|s, a)\left[r + \gamma V(s')\right]$

策略迭代

Policy iteration (using iterative policy evaluation)

1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Repeat
   $\Delta \leftarrow 0$
   For each $s \in \mathcal{S}$:
   $v \leftarrow V(s)$
   $V(s) \leftarrow \sum_{s',r} p(s', r|s, \pi(s))\left[r + \gamma V(s')\right]$
   $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number)

3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
   old-action $\leftarrow \pi(s)$
   $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r|s, a)\left[r + \gamma V(s')\right]$
   If *old-action* $\neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   If *policy-stable*, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2

Figure referenced from *Sutton, Richard S., and Andrew G. Barto. Reinforcement learning: An introduction. p83*

内容提要

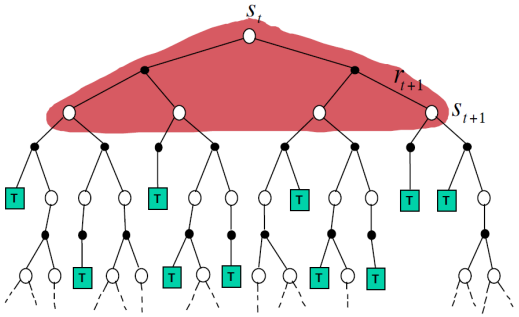
1. 马尔科夫决策过程 Markov decision process (MDP)
2. Bellman方程： Bellman期望方程、 Bellman最优方程
3. P，R已知，用动态规划方法求解最优策略π*

   值迭代、策略迭代
4. P，R未知，用采样方法逼近最优策略π*

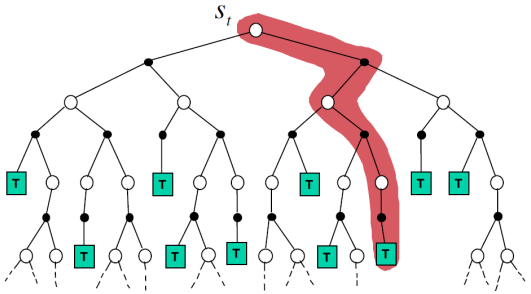
   蒙特卡洛学习（MC）、时序差分学习（TD）

学习方法：蒙特卡洛（MC）和时序差分（TD）

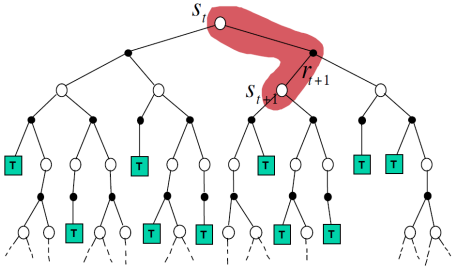
动态规划更新形式



蒙特卡洛更新形式



时序差分更新形式



三种方法价值更新方式比较

学习算法	Bootstrap	Sampling	V_π更新
动态规划	✓	✘	$V_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma V_\pi(s')]$
蒙特卡洛	✘	✓	$V_\pi(s) \leftarrow V_\pi(s) + \alpha[G_t - V_\pi(s)]$
时序差分	✓	✓	$V_\pi(s) \leftarrow V_\pi(s) + \alpha[R_{t+1} + \gamma V_\pi(s') - V_\pi(s)]$

应用ε-贪心策略的蒙特卡洛（MC）学习算法

On-policy first-visit MC control (for ε-soft policies)

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
$Q(s,a) \leftarrow$ arbitrary
$Returns(s,a) \leftarrow$ empty list
$\pi(a|s) \leftarrow$ an arbitrary ε-soft policy

Repeat forever:
(a) Generate an episode using π
(b) For each pair s, a appearing in the episode:
$G \leftarrow$ return following the first occurrence of s, a
Append G to $Returns(s,a)$
$Q(s,a) \leftarrow \text{average}(Returns(s,a))$
(c) For each s in the episode:
$A^* \leftarrow \arg\max_a Q(s,a)$
For all $a \in \mathcal{A}(s)$:

$$\pi(a|s) \leftarrow \begin{cases} 1-\varepsilon+\varepsilon/|\mathcal{A}(s)| & \text{if } a = A^* \\ \varepsilon/|\mathcal{A}(s)| & \text{if } a \neq A^* \end{cases}$$

[1] RS Sutton, AG Barto. Reinforcement learning: an introduction. MIT Press. 2016. University College London. P109

TD - SARSA: State–action–reward–state–action

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\left[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\right]$$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\epsilon > 0$

Initialize $Q(s, a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrary, and $Q(terminalstate, .) = 0$

Repeat for each episode:

Initialize s

Choose $\boldsymbol{a}$ from $\boldsymbol{s}$ using policy derived from $\boldsymbol{Q}$ $(\boldsymbol{\epsilon - greery})$

Repeat for each step of episode:

Take action a, observe R, s'

Choose $\boldsymbol{a'}$ from $\boldsymbol{s'}$ using policy derived from $\boldsymbol{Q}$ $(\boldsymbol{\epsilon - greery})$

$Q(s, a) \leftarrow Q(s, a) + \alpha[R + \gamma Q(s', a') - Q(s, a)]$

$s \leftarrow s'$; $a \leftarrow a'$

Until s is terminal;

TD - Q Learning

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right]$$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\epsilon > 0$

Initialize $Q(s, a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrary, and $Q(terminalstate, .) = 0$

Repeat for each episode:

Initialize s

Repeat for each step of episode:

Choose $\boldsymbol{a}$ from $\boldsymbol{s}$ using policy derived from $\boldsymbol{Q}$ $(\boldsymbol{\epsilon - greery})$

Take action a, observe R, s'

$$\boldsymbol{Q(s, a) \leftarrow Q(s, a) + \alpha\left[R + \gamma \max_{a \in \mathcal{A}(s')} Q(s', a) - Q(s, a)\right]}$$

$s \leftarrow s'$

Until s is terminal;