COMPSCI 188: Introduction to AI

Course notes for UC Berkeley COMPSCI 188: Introduction to AI

Hongyi Chen

UC Berkeley

Contents

Chapter 1: Search and Constraint Satisfaction

COMPSCI 188: Introduction to AI

Included lectures

- Lecture 1: Search
- Lecture 2: Informed Search
- Lecture 3: Constraint Satisfaction Problems
- Lecture 4: Constraint Satisfaction Problems II

Lecture 1: Search

0.1 1. Different Agents

- **Reflex agents:** based on current percept (and maybe memory).
- **Planning agents:** based on (hypothesized) consequences of actions.

0.2 2. Search Problem 搜索问题

一个搜索问题必须包含以下元素：* A **state space** - The set of all possible states that are possible in your given world * A set of **actions** available in each state * A **transition** model - Outputs the next state when a specific action is taken at current state * An action **cost** - Incurred when moving from one state to another after applying an action * A **start state** - The state in which an agent exists initially * A **goal test** - A function that takes a state as input, and determines whether it is a goal state

A solution is a sequence of actions (a plan) which transforms the start state to a goal state.

0.2.1 2.1. State Space

解决问题无需所有细节

- **World State:** includes every last detail of the environment.
- **Search State:** keeps only the details needed for planning.

0.2.2 2.2. State Space Graphs

A mathematical representation of a search problem.

(We can rarely build this full graph in memory - **Too Big!!!**)

- Nodes - (abstracted) world configurations
- Arcs - successors (actions)
- The goal test is a set of goal nodes (maybe only one)

搜索问题在于寻找从初始状态到目标状态的路径。

所有可能位置和道路构成的完整地图即为状态空间图。

0.2.3 2.3. Search Tree

General Procedure

```
function Tree-Search(problem,strategy) -> returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if no candidates for expansion:
            return failure

        choose a leaf node for expansion according to startegy
        if node contains a goal state:
            return the corresponding solution
```

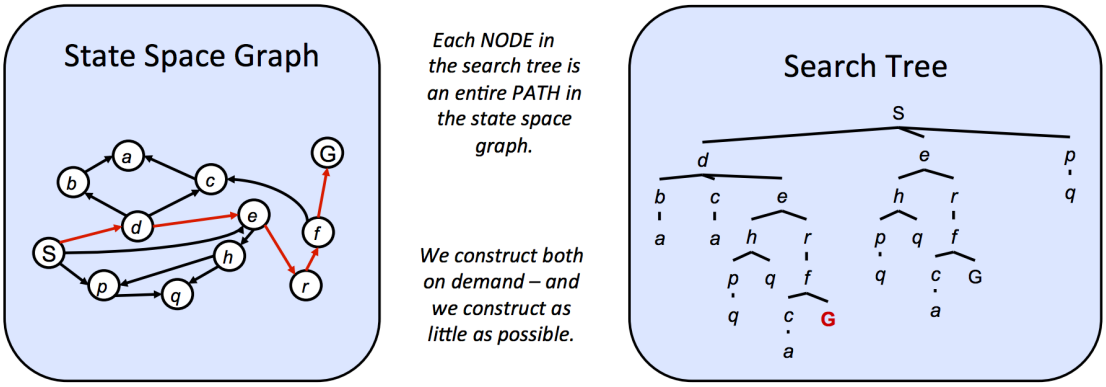


Figure 1: Graph and Tree

```
else:
    expand the node and add the resulting nodes to the search tree
```

如果这些结构体过大而无法在内存中表示怎么办?
仅储存当前状态!

Similarity
永远无法在内存中构建完整的搜索树，因为它会变得过于庞大

Each NODE in the search tree is an entire PATH in the state space graph.
搜索树中的节点不仅表示一个位置，更代表到达该位置所采取的完整动作序列。

0.3 3. Uninformed Search Methods

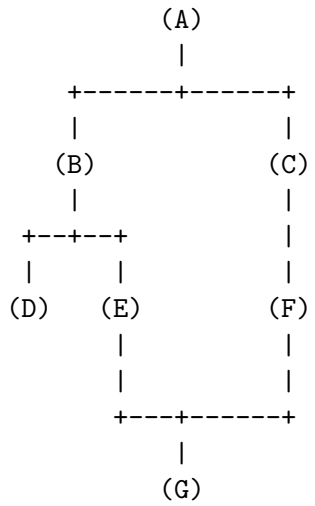
当我们在搜索树中无法确定目标状态的位置时，就不得不从**无信息搜索**范畴内的技术中选择树搜索策略。
这包含了三个常见的算法：DFS, BFS, UCS。

0.3.1 3.1. Depth-First Search (DFS)

特性	描述
思路	从起始节点选择最深的边界节点进行扩展。
边界	移除最深节点并用其子节点替换。这同时意味着子节点现在成为新的最深节点——它们的深度比之前最深节点的深度大 1。**使用 Stack 很容易能完成!**
完整性	不完全。如果存在循环，相应的搜索树深度会无限。
最优性	不具备。只是在搜索树中找到 “最左边” 的解决方案，而不考虑路径成本。
时间复杂度	$O(b^m)$, m is the maximum depth, b is the branching factor （一个节点最多有多少个子节点）.

特性	描述
空间复杂度	$O(bm)$

例：在一个图中用 DFS 找到从节点 A 到 G 的路径



逻辑：使用两个数据结构实现 * 边界 (Frontier)：用一个栈 (Stack) 来实现。栈顶永远是我们下一步要探索的节点。* 已访问 (Visited)：用一个集合 (Set) 来记录已经探索过的节点，防止走回头路或陷入死循环。

Step 0: 将起点 A 放入栈中。* Frontier (Top -> Bottom): [A]
* Visited: {} **Step1:** 探索 A
从栈顶取出 A。A 不是目标。将 A 标记为 “已访问”。找到 A 的所有邻居 (B, C)，将它们压入栈中。* Frontier: [B, C] * Visited: {A} **Step 2:** 探索 B
从栈顶取出 B。B 不是目标。将 B 标记为 “已访问”。找到 B 的邻居 (D, E)，将它们压入栈中。* Frontier: [D, E, C] * Visited: {A, B} **Step 3:** 探索 D (撞南墙) 从栈顶取出 D。D 不是目标。将 D 标记为 “已访问”。D 没有任何邻居，是个死胡同。* Frontier: [E, C] * Visited: {A, B, D} **Step 4:** 探索 E * Frontier：[G, C] * Visited: {A, B, D, E} **Step 5:** 探索 G (找到目标！) 从栈顶取出 G。G 是我们的目标！搜索成功！

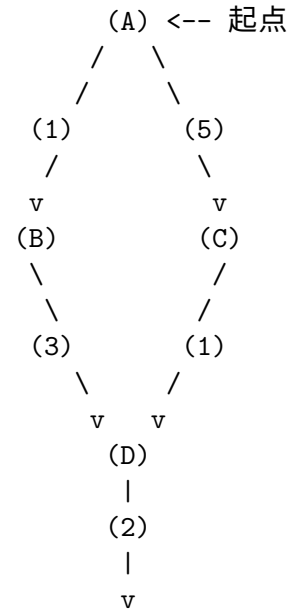
0.3.2 3.2. Breadth-First Search (BFS)

特性	描述
思路	选择距离起始节点最浅的边界节点进行扩展。
边界	先访问较浅的节点，然后再访问较深的节点，按照节点的插入顺序进行访问。**使用 Queue 很容易实现！**
完整性	完全。层数有限，必然到达最深处。

特性	描述
最优性	不具备。在确定边界上要替换的节点时根本不考虑成本。BFS 保证最优的特殊情况是所有边成本相等，因为这会将 BFS 简化为均匀成本搜索的特例。
时间复杂度	$O(b^s)$, s 为最优解的深度, b is the branching factor （一个节点最多有多少个子节点）.
空间复杂度	$O(b^s)$

0.3.3 3.3. Uniform Cost Search (UCS)

特性	描述
思路	始终从起始节点选择成本最低的边界节点进行扩展。
边界	选择基于堆的 **Priority Queue**。入队节点 v 的优先级为起点节点到 v 的路径成本，或 v 的后向成本。在移除当前最小成本路径并替换为其子节点时，会自动重新排序以维持基于路径成本的期望排序。
完整性	完全。如果目标状态存在，它必然存在一条有限长度的最短路径.
最优性	具备。(统一成本搜索采用的策略与 Dijkstra 算法相同，主要区别在于 UCS 会在找到解状态时终止，而不是找到到达所有状态的最短路径。)
时间复杂度	$O(b^{C^*/\epsilon})$, C^* 为从起点到终点的最优路径的实际成本, ϵ 为任意一步（边）的最小成本。
空间复杂度	$O(b^{C^*/\epsilon})$



(G) <-- 目标

逻辑：* 边界 (Frontier)：用一个优先队列 (Priority Queue) 实现。队列中的每个元素是 (成本, 节点)，并始终按成本从小到大排序。* 已探索 (Explored)：用一个集合 (Set) 来记录已经探索过的节点，确保我们不会重复处理。

Step 0: 初始化
将起点 A 放入优先队列，其成本为 0。* Frontier (按成本排序): [(0, A)]
* Explored: {}

Step 1: 探索 A 从优先队列中取出成本最低的节点 (0, A)。将 A 放入 "已探索"。找到 A 的邻居 B 和 C。

到 B 的成本是 $0 + 1 = 1$。将 (1, B) 放入队列。
到 C 的成本是 $0 + 5 = 5$。将 (5, C) 放入队列。* Frontier: [(1, B), (5, C)] * Explored: {A}

Step 2: 探索 B (而不是 C！) 此时队列中成本最低的是 (1, B)。取出它，并将 B 放入 "已探索"。找到 B 的邻居 D。

到 D 的成本是 (A 到 B 的成本) + (B 到 D 的成本) $= 1 + 3 = 4$。将 (4, D) 放入队列。

- Frontier: [(4, D), (5, C)] (注意 (4,D) 排在了 (5,C) 的前面)
- Explored: {A, B}

关键点：尽管 C 是 A 的直接邻居，但因为当前到 D 的已知路径成本 (4) 比到 C 的成本 (5) 更低，所以 UCS 优先选择了这条看起来更有希望的路径。

Step 3: 探索 D 从优先队列中取出成本最低的节点 (4, D)。将 D 放入 "已探索"。找到 D 的邻居 G。

到 G 的成本是 (A 到 D 的成本) + (D 到 G 的成本) $= 4 + 2 = 6$。将 (6, G) 放入队列。

- Frontier: [(5, C), (6, G)]
- Explored: {A, B, D}

Step 4: 探索 C 从优先队列中取出成本最低的节点 (5, C)。将 C 放入 "已探索"。找到 C 的邻居 D。

到 D 的新路径成本是 (A 到 C 的成本) + (C 到 D 的成本) $= 5 + 1 = 6$。

检查发现 D 已经在 "已探索" 集合里了。而且新路径成本 (6) 并不比之前到 D 的成本 (4) 更低，因此我们忽略这条路径。

- Frontier: [(6, G)]
- Explored: {A, B, D, C}

Step 5: 探索 G (找到目标！) 从优先队列中取出成本最低的节点 (6, G)。G 是我们的目标！搜索成功！找到的最低成本是 6，路径为 A -> B -> D -> G。

Lecture 2: Informed Search

这和 Uninformed Search 有什么区别? 利用与问题相关的额外信息，优先探索 "看起来" 离目标最近的节点。
不过，实际上未必会更近!

0.4 1. Heuristics

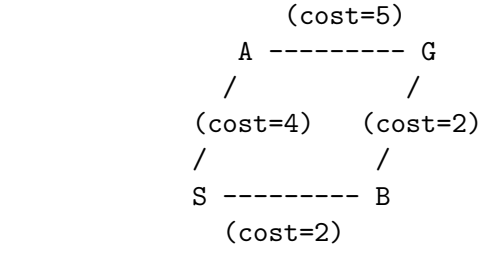
A heuristic is a function that estimates how close a state is to a goal.

启发式算法通常是针对宽松问题（即移除了原始问题的一些约束）的解决方案

0.5 2. Greedy Search 贪婪搜索

特性	描述
思路	选择具有最低启发式值的边界节点进行扩展。
边界	采用 **Priority Queue**。区别在于，贪婪搜索不是使用后向成本（从起点走到当前节点 n 所已经花费的实际成本）来分配优先级，而是使用启发式值形式的预估前向成本（从当前节点 n 走到终点所预估的剩余成本）。
完整性	不完全。在不同场景下的表现通常相当难以预测。
最优性	不具备。在不同场景下的表现通常相当难以预测。

对比 UCS 和 Greedy Search
从城市 S (起点) 前往城市 G (终点)。途中有两个中转城市 A 和 B。



给定 Heuristics （贪婪搜索需要，UCS 不需要）
我们用 "到终点 G 的直线距离" 作为启发式函数 h(n) 的值。

$$h(A) = 1$$
$$h(B) = 4$$
$$h(S) = 5$$
$$h(G) = 0$$

特性	**UCS (只关心后向成本)**	**Greedy Search (只关心启发值)**
核心关注点	路径已花费的实际成本 $g(n)$	到达目标的估计成本 $h(n)$
决策过程	UCS 发现去 B 的成本比去 A 的成本更低，选择扩展 B；随后发现 G 是终点，搜索结束。	贪婪搜索发现 A 的启发值远低于 B 的启发值，选择扩展 A；随后发现 G 是终点，搜索结束。
找到的路径	$S \to B \to G$	$S \to A \to G$
最终路径总成本	4	9
路径结果	最优路径	非最优路径

0.6 3. A* Search

特性	描述
思路	选择具有最低估计总扩展成本的边界节点
边界	采用 **Priority Queue**。A* 将 UCS 使用的总后向成本与贪婪搜索使用的估计前向成本（Heuristics 值）相加，从而有效地得出从起点到目标的估计总成本.
完整性 & 最优性	在给定合适的启发式算法（我们稍后会介绍）的情况下，A* 搜索既完备又最优。

0.7 4. Admissibility 可接纳行（寻找好的 Heuristics）

Recall：首先重新表述 UCS、贪婪搜索和 A* 中用于确定优先级队列顺序的方法。$g(n)$ - Total backwards cost computed by UCS. $h(n)$ - The heuristic value function, or estimated forward cost, used by greedy search. $f(n)$ - Estimated total cost, used by A* search. $f(n) = g(n) + h(n)$

但不是所有 Heuristics 都能达到完备和最优。这引出了 Admissibility 可接纳性的概念：The condition required for optimality when using A* tree search is known as **admissibility**.

启发式算法估计的值不能高估实际成本。

$$\forall n, 0 \leq h(n) \leq h^*(n)$$

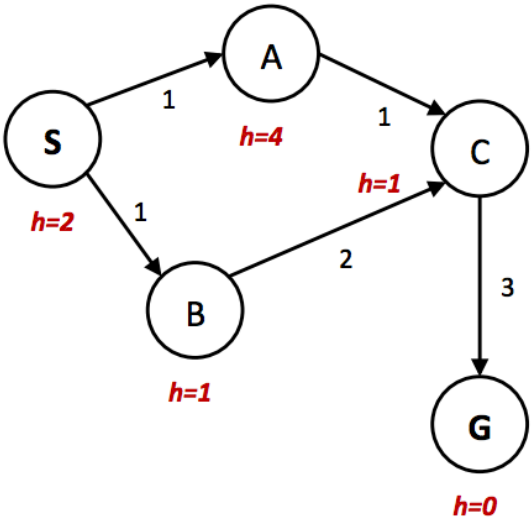
Theorem: 如果启发式函数 h 满足可接纳性约束，则对该搜索问题使用带有 h 的 A* 树搜索将产生最优解。

可以在此处查看证明

0.8 5. Graph Search

Intention: 树搜索的问题：在某些情况下，它可能永远找不到解，在状态空间图中无限循环搜索

Graph Search 在使用您选择的搜索方法时，维护一个 “已到达” 的扩展节点集合。然后，确保每个节点在扩展前尚未包含在集合中，如果不在，则在扩展后将其添加到集合中。带有这种附加优化的树搜索称为图搜索。



Intention: 图搜索依然存在问题,举一个例子如下。

由于节点 A 的启发式值远大于节点 B 的启发式值，因此节点 C 首先沿着第二条次优路径扩展为节点 B 的子节点。然后，它被放入 “已到达” 集。因此当 A* 图搜索将其作为 A 的子节点进行访问时，它无法重新扩展它，因此它永远找不到最优解。

→ 不仅需要检查 A* 是否已经访问过某个节点，还需要检查是否找到了一条更便宜的路径!

0.9 6. Dominance 优势度（比较 Heuristics 的好坏）

怎样更好? → 更精确地估计从任何给定状态到目标的距离! 如果启发式 a 优于启发式 b ，那么对于状态空间图中的每个节点，a 的估计目标距离都大于 b 的估计目标距离。

$$\forall n : h_a(n) \geq h_b(n)$$

通常的做法是，针对任何给定的搜索问题生成多个可采纳启发式算法，并计算它们输出值的最大值.

0.10 7. Local Search 本地搜索

与之前的不同：之前我们关心目标状态以及到达它的最佳路径，接下来我们只关心目标状态在哪里。

思路：算法局部地向目标值更高的状态移动，直到达到最大值（最好是全局最大值）

这部分包含以下算法：hill-climbing, simulated annealing, local beam search, and genetic algorithms.

!!! 待补充!!!

Lecture 3: Constraint Satisfaction Problems

与搜索问题不同，CSP 是一种识别问题，即我们只需识别某个状态是否为目标状态，而无需考虑如何达到该目标。

约束满足问题是 NP-hard 的，大致意味着不存在已知算法可以在多项式时间内找到此类问题的解。

CSP 由三个因素定义：* Variables - CSPs possess a set of N variables $X_1, \ldots, X_N$ that can each take on a single value from some defined set of values.

- Domain - A set $\{x_1, \ldots, x_d\}$ representing all possible values that a CSP variable can take on.
- Constraints - Constraints define restrictions on the values of variables, potentially with regard to other variables.

典型的例子是 N 皇后问题。**变量** - X_{ij}，其中 $0 \leq i, j < N$。每个 X_{ij} 代表 $N \times N$ 棋盘上的一个网格位置。

域 - $\{0, 1\}$。每个 X_{ij} 可以取值 0 或 1，这是一个布尔值，表示在棋盘位置 (i, j) 是否存在一个皇后。

约束 - * $\forall i, j, k\ (X_{ij}, X_{ik}) \in \{(0,0), (0,1), (1,0)\}$。任何两个皇后不能在同一行。* $\forall i, j, k\ (X_{ij}, X_{kj}) \in \{(0,0), (0,1), (1,0)\}$。任何两个皇后不能在同一列。* $\forall i, j, k\ (X_{ij}, X_{i+k,j+k}) \in \{(0,0), (0,1), (1,0)\}$。任何两个皇后不能在同一主对角线或副对角线上。* $\forall i, j, k\ (X_{ij}, X_{i+k,j-k}) \in \{(0,0), (0,1), (1,0)\}$。同上。* $\sum_{i,j} X_{ij} = N$。满足棋盘上恰好有 N 个皇后的要求。

0.11 1. Constraints Graph 约束图

约束满足问题通常表示为约束图，其中节点表示变量，边表示变量之间的约束。

约束包含以下几种类型：- Unary Constraints 一元约束：涉及 CSP 中的单个变量。不会在约束图中表示，而是在必要时用于修剪其约束变量的域。- Binary Constraints 二元约束：二元约束涉及两个变量。它们在约束图中以传统的图边形式表示。- Higher-order Constraints 高阶约束：涉及三个或更多变量的约束也可以用 CSP 图中的边表示，它们只是看起来有点不寻常。

0.12 2. Backtracking Algorithm (Solving CSP)

使用回溯搜索解决——本质上是优化后的 DFS。改进基于以下原则：- 固定变量的顺序，并按此顺序选择变量的值。（因为赋值是可交换的，所以有效）- 为变量选择值时，仅选择与之前赋值不冲突的值。如果不存在这样的值，则回溯并返回到上一个变量，并更改其值。

```
function BACKTRACKING-SEARCH(csp) returns solution/failure
  return RECURSIVE-BACKTRACKING({}, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) returns soln/failure
  if assignment is complete then return assignment
  var $\leftarrow$ SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
    if value is consistent with assignment given CONSTRAINTS[csp] then
      add {var = value} to assignment
      result $\leftarrow$ RECURSIVE-BACKTRACKING(assignment, csp)
      if result $\neq$ failure then return result
      remove {var = value} from assignment
```

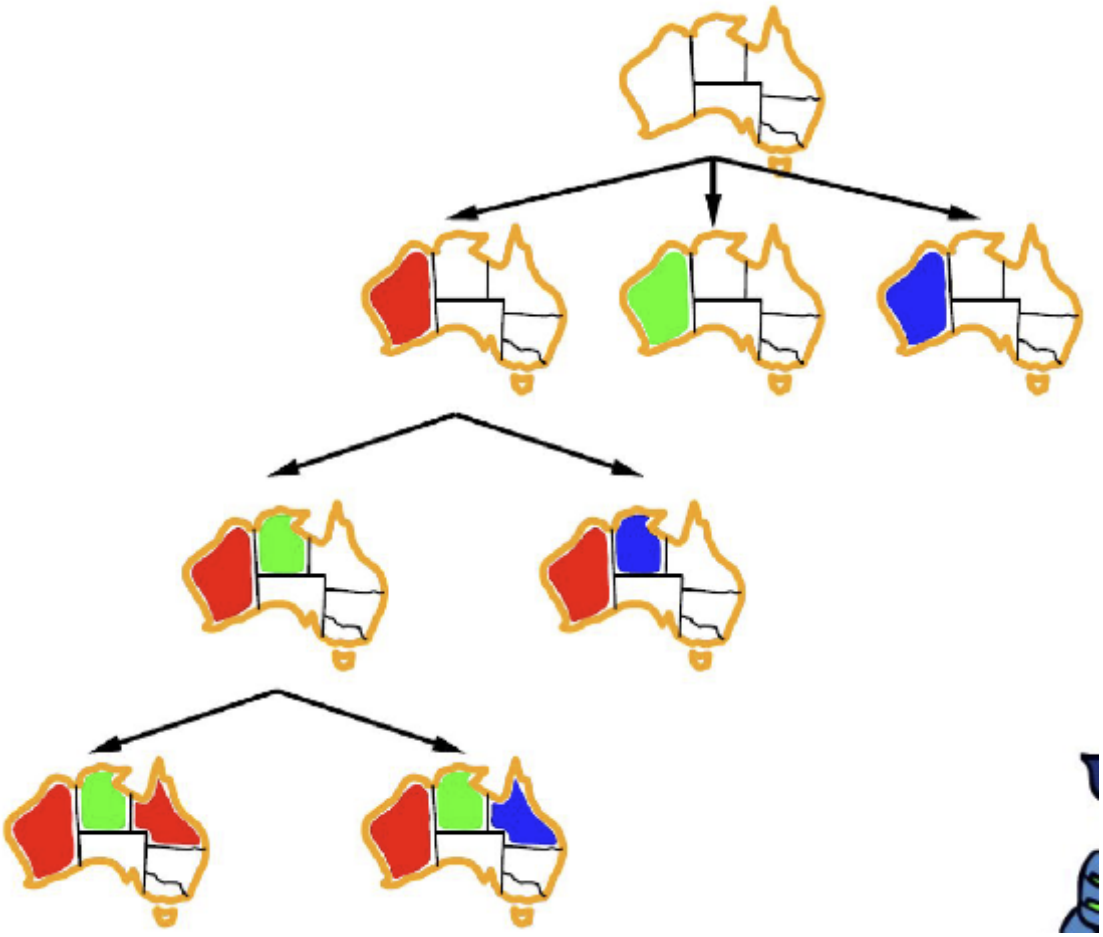


Figure 2: Backtracking example

```
  return failure
```

0.13 3. Improving Backtracking

0.13.1 3.1. Filitering 过滤

Filtering 解决了 `Can we detect inevitable failure early?` 的问题

Forward Checking

检查未分配变量与当前分配变量的直接冲突！（但其他冲突他就无能为力了）

Arc Consistency

- Important: If X loses a value, neighbors of X need to be rechecked!
- Arc consistency detects failure earlier than forward checking
- Can be run as a preprocessor or after each assignment
- What's the downside of enforcing arc consistency?
- Remember: **Delete from the tail!**

```
function AC-3(csp) returns the CSP, possibly with reduced domains
  inputs: csp, a binary CSP with variables {X_1, X_2, ..., X_n}
```

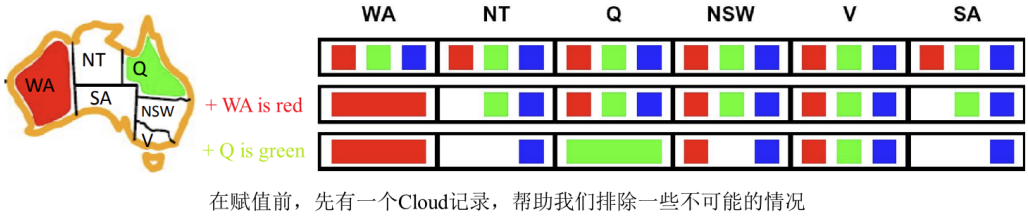


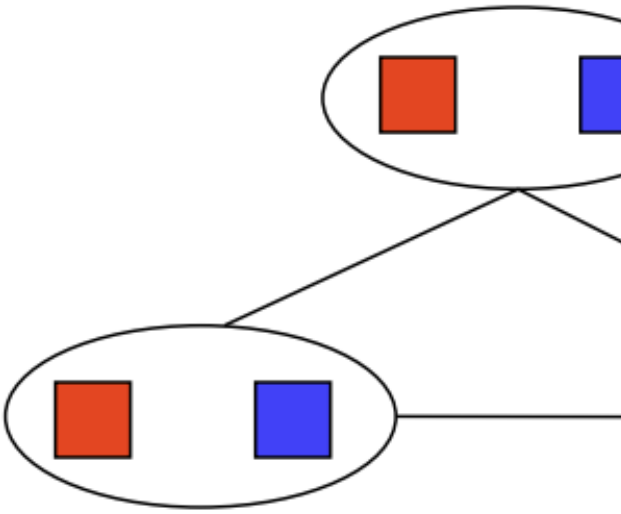
Figure 3: Filtering

```
  local variables: queue, a queue of arcs, initially all the arcs in csp

  while queue is not empty do
    (X_i, X_j) <- REMOVE-FIRST(queue)
    if REMOVE-INCONSISTENT-VALUES(X_i, X_j) then
      for each X_k in NEIGHBORS[X_i] do
        add (X_k, X_i) to queue

function REMOVE-INCONSISTENT-VALUES(X_i, X_j) returns true iff succeeds
  removed <- false
  for each x in DOMAIN[X_i] do
    if no value y in DOMAIN[X_j] allows (x, y) to satisfy the constraint X_i <-> X_j
    then delete x from DOMAIN[X_i]; removed <- true
  return removed
```

Runtime: $O(n^2d^3)$, can be reduced to $O(n^2d^2)$ BUT detecting all possible futrue problems is NP-hard.



Limitation of Arc Consistancy See an example below:
Arc Consistancy satisfied!! BUT no solution!!!

0.13.2 3.2. Ordering 排序

在求解 CSP 时，我们会对所涉及的变量和值进行一些排序。在实践中，使用两个广泛的原则“动态” 计算下一个变量及其对应的值通常更有效，这两个原则是：最小剩余值和最小约束值：- **Minimum Remaining Value 最小剩余值 (MRV)** 选择未赋值且剩余有效值最少的变量（即约束最严格的变量）。

- **Least Constraining Value 最小约束值 (LCV)** 选择那个对其他（邻居）变量约束最少的值，即选择一个能为未来的选择保留最多可能性的值。这需要额外的计算（例如，对每个值重新运行弧一致性/前向检查或其他过滤方法以找到 LCV），但仍然可以根据使用情况提高速度。

这很有意思！对于 variable 我们选择做最严格的，但对于 value 我们选择最宽松的。

Lecture 4: Constraint Satisfaction Problems II

0.14 Recall

1. K-Consistency

Increasing degrees of consistency

- **1-Consistency (Node Consistency)**: Each single node's domain has a value which meets that node's unary constraints.
- **2-Consistency (Arc Consistency)**: For each pair of nodes, any consistent assignment to one can be extended to the other.
- **K-Consistency**: For each k nodes, any consistent assignment to $k-1$ can be extended to the k^{th} node.

Higher k more expensive to compute (but makes filtering better: tradeoff!).

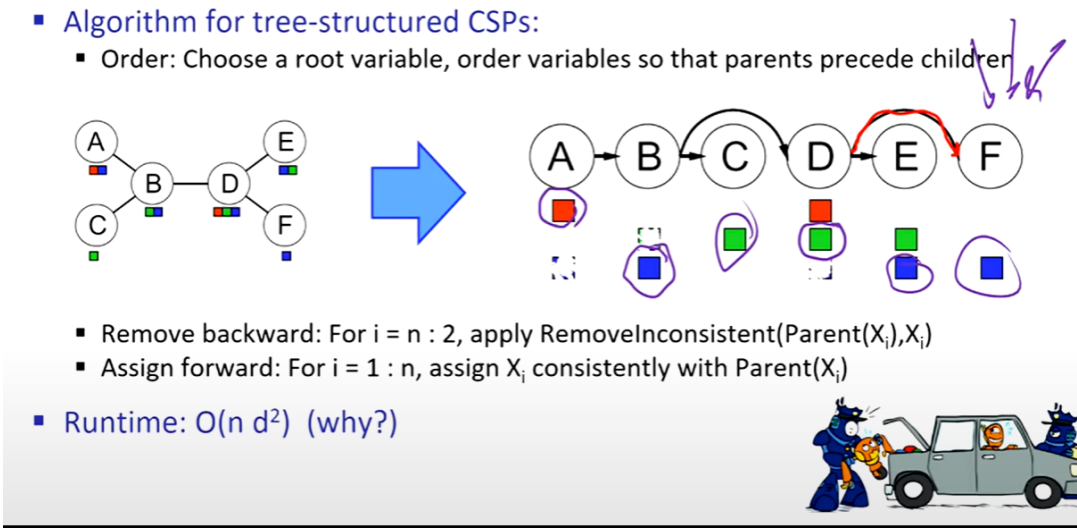
2. Strong K-Consistency * Strong k-consistency 的区别: also k-1, k-2, … 1 consistent.

- **Claim**: strong n-consistency means we can solve without backtracking! Problem: NP-hard Problem, so it's extremely hard to build a Strong K-Consistency!!!
- Lots of middle ground between arc consistency and n-consistency! (e.g. k=3, called path consistency)

0.15 1. Structure: Improvement to solving CSP (continue)

Theorem: if the constraint graph has no loops, the CSP can be solved in $O(n \cdot d^2)$ time. 对于一般的 CSP 问题 worst-case time is $O(d^n)$.

Tree Structured CSPs



步骤：1. 首先，在约束图中为 CSP 选择一个任意节点作为树的根（哪一个并不重要，因为基本图论告诉我们树的任何节点都可以作为根）。2. 将树中所有无向边转换为指向远离根节点的有向边。然后对生成的有向无环图进行排序，使所有边都指向右方。3. 执行弧一致性的**反向**传递：从 $i = n$ 向下迭代到 $i = 2$ ，对所有弧 $Parent(X_i) \to X_i$ 强制执行弧一致性。4. 最后，执行**前向**赋值。从 X_1 开始，到 X_n ，为每个 X_i 赋值，使其与其父节点的值一致。

由于我们在所有这些弧上都强制了弧一致性，因此无论我们为任何节点选择什么值，我们都知道其子节点至少会有一个一致的值。

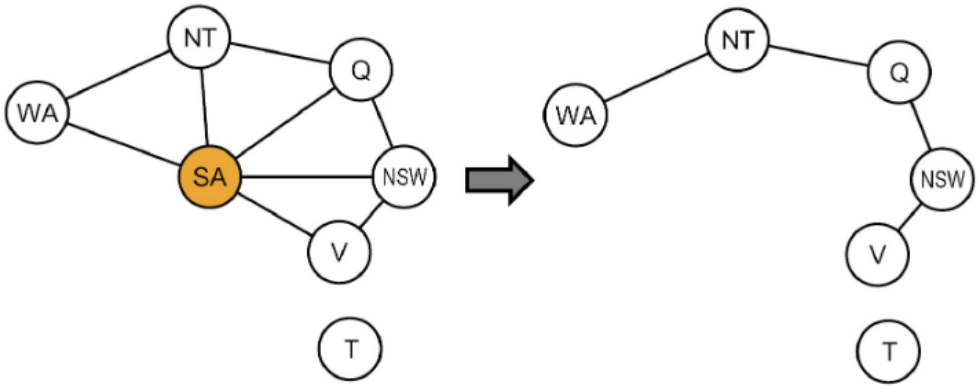


Figure 4: Nearly Tree example

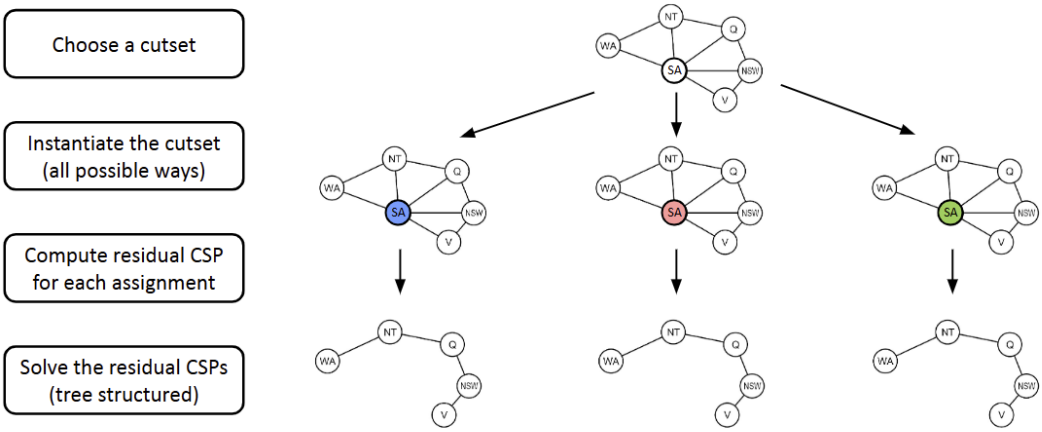


Figure 5: cutset conditioning

0.16 2. Improving Structure

Tree is not that common! -> Nearly Tree-Structured CSPs

- **Conditioning (条件化)**: 实例化一个变量，修剪其邻居的域。
- **Cutset Conditioning (割集条件化)**: 实例化（通过所有方式）一组变量，使得剩余的约束图成为一棵树。
- **割集大小 c 的运行时复杂度**为 $O((d^c)(n-c)d^2)$，当 c 很小时，运行速度非常快。
-

0.17 3. Local Search 局部搜索

Backtrcking search is not the only algorithm that exists for solving CSP.

Another widely used algorithm is **Local Search**.

局部搜索通过迭代改进来工作——首先对值进行随机赋值，然后迭代地选择一个随机冲突变量，并将其值重新赋给违反约束最少的变量，直到不再存在约束违反。(min-conflicts heuristic)

Local Search 不完整也不最优。

Critical ratio 临界比率：这个比率附近，成本会极其昂贵。

$$R = \frac{\text{number of constraints}}{\text{number of cariables}}$$

三种算法（供了解）：hill-climbing, simulated annealing, and genetic algorithms

Chapter 2: Game Trees, MDPs, and Reinforcement Learning

COMPSCI 188: Introduction to AI

Included lectures

- Lecture 5: Game Tree / Adversarial Search
- Lecture 6: Game Tree II
- Lecture 7&8: Markov Decision Processes
- Lecture 9&10: Reinforcement Learning

Lecture 5: Game Tree / Adversarial Search

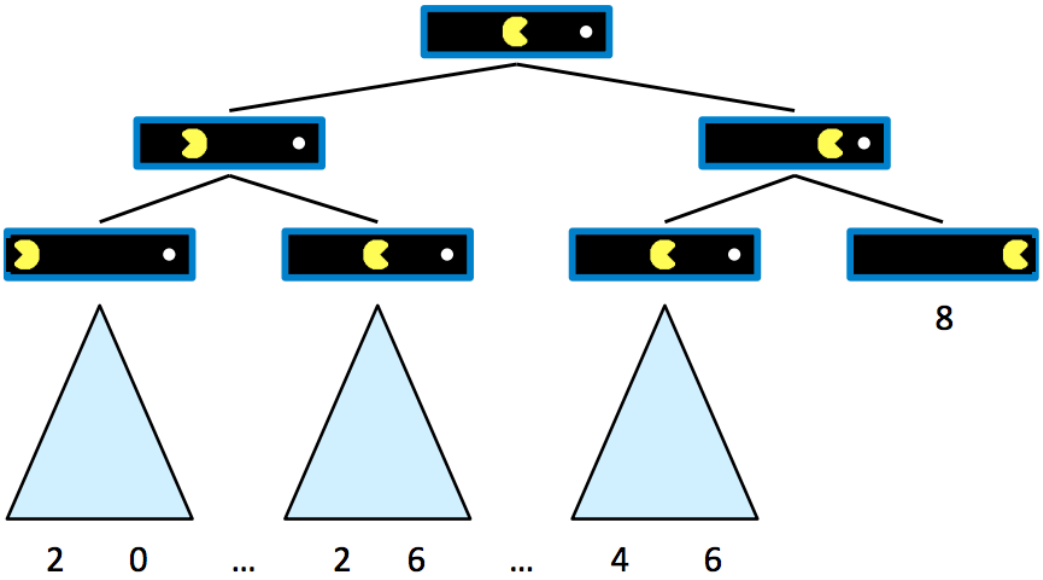
0.18 1. Deterministic Games

Many possible formalizations, one is:

- **Initial state**: `s_0`
- **Players**: `Players(s)` denote whose turn it is
- **Actions**: `Actions(s)` available actions for the player
- **Transition model**: `Result(s, a)`
- **Terminal test (True/False)**: `Terminal-test(s)`
- **Terminal values**: `Utility(s, player)`

0.19 2. Settings for Zero-Sum Game

假设吃豆人初始得分为 10 分，每走一步扣 1 分，直到吃掉小球为止。此时游戏到达终止状态并



结束。

状态值定义为代理在该状态下能够实现的最佳结果（Utility），非终端状态的值定义为其子状态值的最大值。

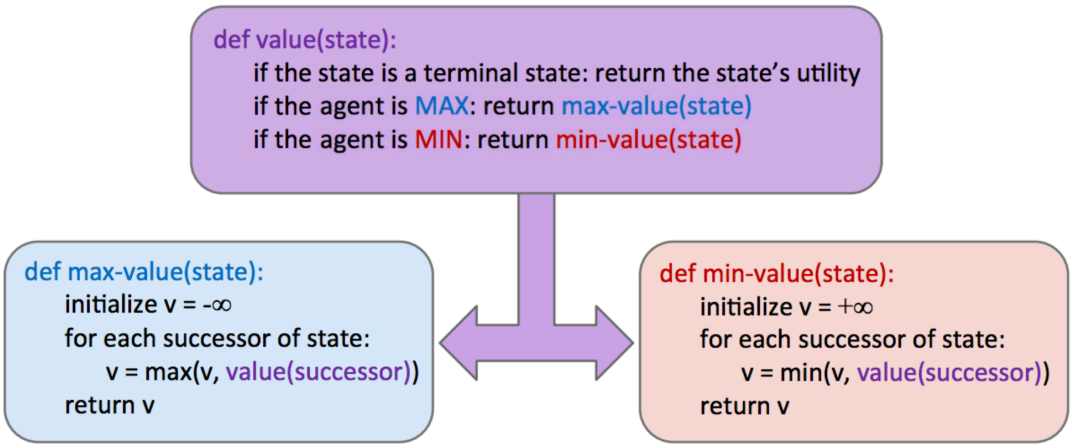
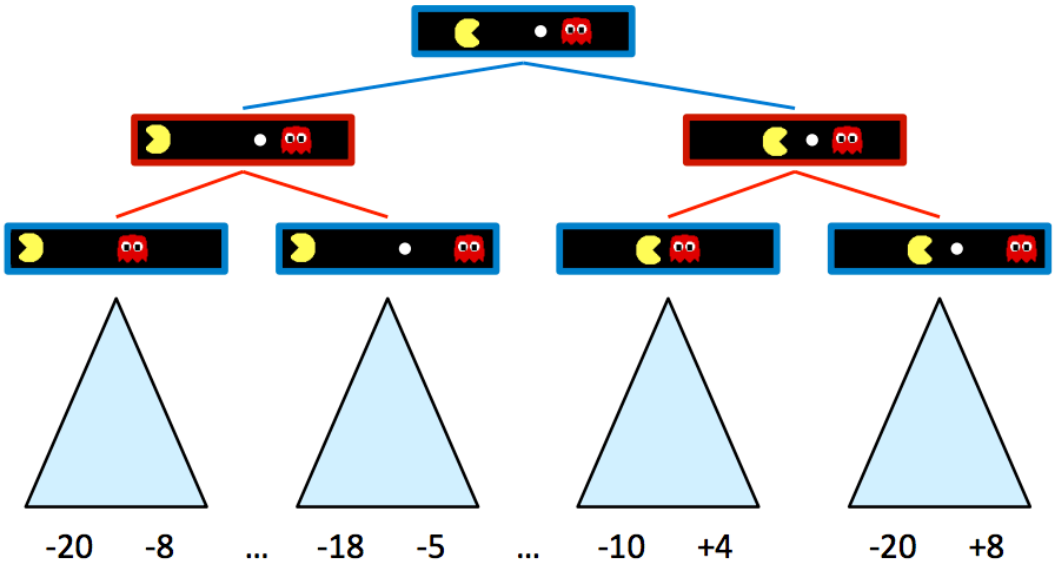


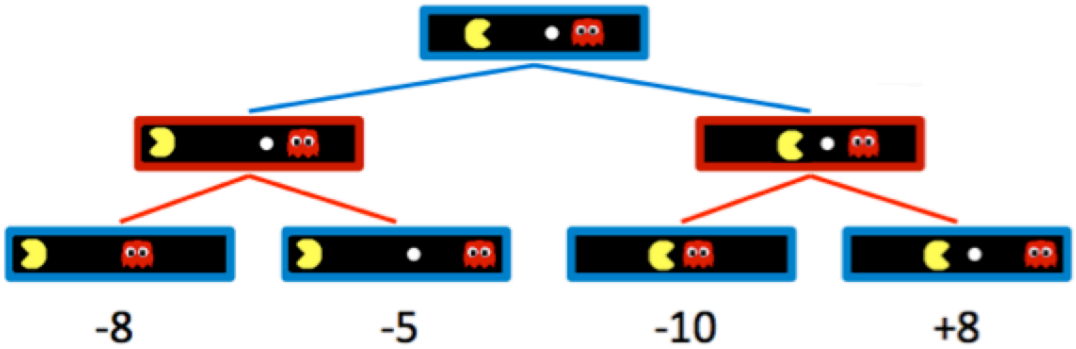
Figure 6: Minimax



拓展到两个 agent：

0.20 3. Minimax

基于一个激励假设：我们面对的对手会采取最优行为，并且总是会做出对我们最不利的举动。分析



一个简单的例子：

$\forall$agent-controlled states, $V(s) = \max_{s' \in successors(s)} V(s')$

$\forall$opponent-controlled states, $V(s) = \min_{s' \in successors(s)} V(s')$

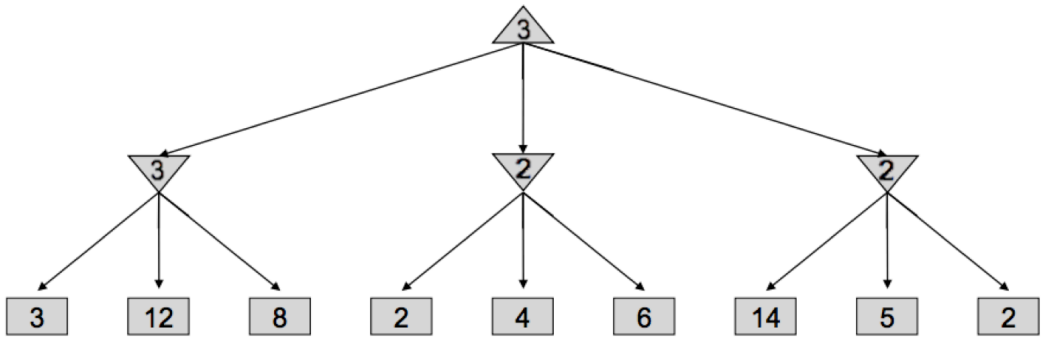


Figure 7: alphabeta

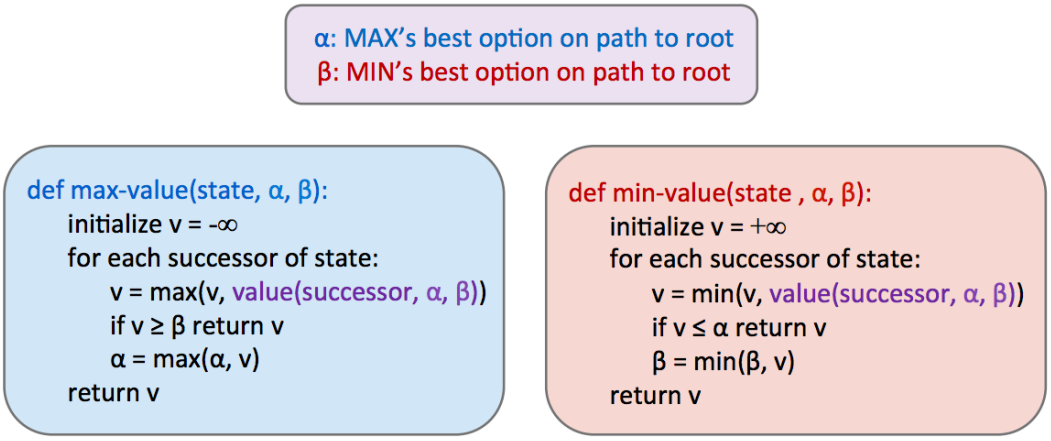


Figure 8: alpha beta code

Lecture 6: Game Tree II

0.21 1. Minimax Improving

0.21.1 1.1 Alpha-Beta Pruning

Intuion: Minimax 看着不错，但他与 DFS 很类似——时间复杂度 $O(b^m)$——运行时间太长！

优化方法：$\alpha - \beta$ pruning。

只要我们访问了中间最小值值为 2 的子节点，就不再需要查看中间最小值的其他子节点了！为什么呢？因为我们已经看到中间最小值的一个子节点的值为 2，所以我们知道，无论其他子节点的值是什么，中间最小值的值最多为 2。既然这一点已经确定，让我们再进一步思考——根节点的最大值节点正在左边最小值的值 3 和 ≤ 2 之间做出选择，它肯定会优先选择左边最小值返回的 3，而不是中间最小值返回的值，而不管其剩余子节点的值是什么。

Alpha (α): MAX 玩家的 "最低承诺"（在已探索过的路径中，MAX 玩家至少能获得的分数。）
Beta (β): MIN 玩家的 "最高限制"

$$\text{if } \alpha \geq \beta \rightarrow \textit{pruning}$$

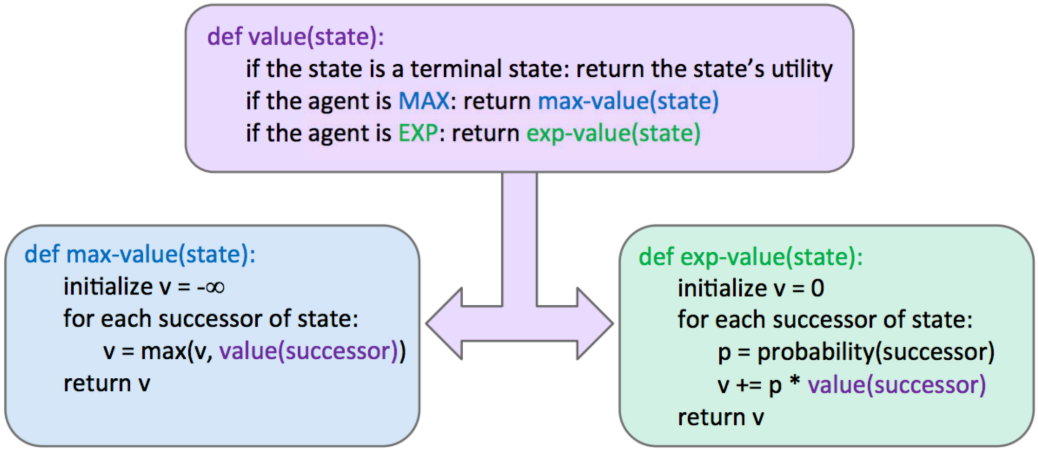


Figure 9: expectimax code

0.21.2 1.2. Evaluation Function

depth-limited minimax：设置深度限制 → 最后一层的状态值怎么得到？**Heuristics!** In this case we call it **evaluation function!**

$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s)$

0.22 2. Expectimax

由于极小极大算法认为它正在对最优对手做出响应，因此在无法保证对智能体动作做出最优响应的情况下，它往往会过于悲观。

这种随机性可以通过极小极大值（minimax）的泛化（expectimax）来表示。expectimax 将机会节点引入博弈树，它不像最小化器节点那样考虑最坏情况，而是考虑平均情况。

节点值规则如下：

$$\forall \text{agent-controlled states}, V(s) = \max_{s' \in successors(s)} V(s')$$

$$\forall \text{chance states}, V(s) = \sum_{s' \in successors(s)} p(s'|s) V(s')$$

$$\forall \text{terminal states}, V(s) = \text{known}$$

在上面的公式中，$p(s'|s)$ 指的是给定的不确定性动作导致从状态 s 移动到 s' 的概率，或者是对于选择导致从状态 s 移动到 s' 的动作的概率，具体取决于游戏的具体情况和所考虑的游戏树。

从这个定义中，我们可以看出 **minimax** 只是 **expectimax** 的一个特例。

0.23 3. Other Game

不同的智能体在博弈中可能承担着不同的任务，而这些任务并不直接涉及彼此之间的严格竞争。

这类博弈可以用以多智能体效用为特征的树来构建——表示为元组，元组中的不同值对应于不同智能体的独特效用。然后，每个智能体尝试在其控制的每个节点上最大化自身的效用。

Lecture 7&8: Markov Decision Processes

0.24 1. Non-deterministic Search

An MDP is defined by: * **A set of states** $s \in S$ * **A set of actions** $a \in A$ * **A transition function** $T(s, a, s')$ 引入了非确定性动作的可能性 * Probability that a from s leads to s', i.e., $P(s'|s, a)$ * Also called the **model** or the **dynamics** * **A reward function** $R(s, a, s')$

- **A start state**
- **Maybe a terminal state**

Discount factor γ Sooner rewards probably have higher utility than later rewards!

$$\gamma \in [0, 1]$$

Markovianess: 无记忆性，如果我们知道当前状态，那么了解过去并不能给我们提供更多关于未来的信息。

0.25 2. Bellman Optimality Equation

首先我们要为状态定价："当前" 状态的价值，是由 "未来" 状态的价值来定义的。

$$\text{Bellman equation: } V^*(s) = \max_a Q^*(s, a)$$

含义：一个状态 s 的最优价值等于从这个状态出发，所能采取的所有动作 a 中，能产生的那个最优的动作价值 $(Q^*(s, a))$。

$$Q^*(s, a) = \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V^*(s')]$$

0.26 3. Value Iteration

如何实际计算这些最优值？我们需要 time limited values!

对于时间限制为 k 个时间步的状态 s ，时间限制值表示为 $V_k(s)$ ，它表示在所考虑的马尔可夫决策过程终止于 k 个时间步的情况下，从 s 可获得的最大预期效用。

等效地，这也是在 MDP 的搜索树上运行深度为 k 的 expectimax 函数所返回的结果。

运行方式如下：

1. $\forall s \in S$，初始化 $V_0(s) = 0$。这是很直观的，因为设置一个 0 步的时限意味着在终止前不能采取任何行动，因此也无法获得任何奖励。
2. 重复以下更新规则直到收敛：

$$\forall s \in S, V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V_k(s')]$$

在价值迭代的第 k 次迭代中，我们使用每个状态的时限为 k 的值来生成时限为 $(k + 1)$ 的值。本质上，我们使用已计算的子问题解（所有的 $V_k(s)$）来迭代地构建更大子问题的解（所有的 $V_{k+1}(s)$）；这就是价值迭代成为**动态规划算法**的原因。

Policy Extraction

原理：如果你处于状态 s ，你应该采取动作 a ，以获得最大的预期效用。

$$\forall s \in \mathcal{S}, \pi^*(s) = \operatorname*{argmax}_a Q^*(s, a) = \operatorname*{argmax}_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V^*(s')]$$

Q-Value Iteration

Q 值迭代是一种计算时间受限的 Q 值的动态规划算法——只能玩 N 步（比如 10 步），游戏必定结束。目标是在这有限的 N 步内，让总收益最大化。

$$Q_{k+1}(s,a) \leftarrow \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma \max_{a'} Q_k(s',a')]$$

0.27 4. Policy Iteration

流程：1. 定义一个**初始策略**。这个策略可以是任意的（但是如果初始策略更接近最优策略，收敛会更快）。2. 重复以下步骤直到收敛：* 通过 **策略评估**（policy evaluation）来评估当前策略。对于一个策略 π，策略评估意味着计算所有状态 s 的 $V^{\pi}(s)$，其中 $V^{\pi}(s)$ 是指在状态 s 开始并遵循 π 时所期望的效用：

$$V^{\pi}(s) = \sum_{s'} T(s,\pi(s),s')[R(s,\pi(s),s') + \gamma V^{\pi}(s')]$$

由于我们为每个状态固定了一个单一的行动，我们不再需要 max 运算符。或者，我们也可以使用以下更新规则，像价值迭代（value iteration）一样，迭代计算 $V^{\pi_i}(s)$ 直到收敛：

$$V^{\pi_i}_{k+1}(s) \leftarrow \sum_{s'} T(s,\pi_i(s),s')[R(s,\pi_i(s),s') + \gamma V^{\pi_i}_{k}(s')]$$

然而，这第二种方法在实践中通常较慢。* 一旦我们评估了当前策略，就使用**策略改进**（policy improvement）来生成一个更好的策略。策略改进使用对策略评估产生的状态价值（values of states）进行策略提取，从而生成这个新的、改进后的策略：

$$\pi_{i+1}(s) = \operatorname*{argmax}_{a} \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^{\pi_i}(s')]$$

如果 $\pi_{i+1} = \pi_i$，则算法已经收敛，我们可以得出结论：$\pi_{i+1} = \pi_i = \pi^*$

Lecture 9&10: Reinforcement Learning

上一节课：马尔可夫决策过程，以及用诸如 Value Iteration 和 Policy Iteration 等技术来求解该过程，以计算状态的最优值并提取最优策略。

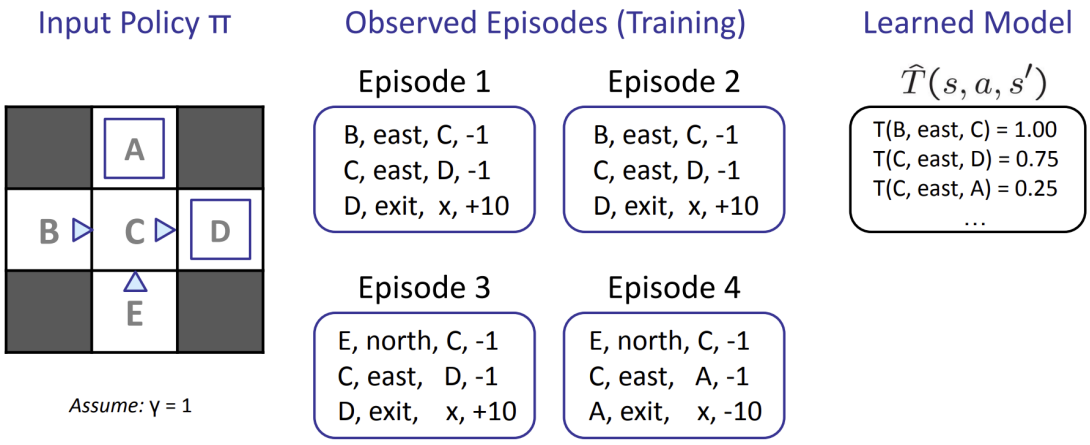
求解马尔可夫决策过程是 **离线规划 (offline planning)** 的一个例子，其中 Agent 对转换函数和奖励函数拥有充分的了解，它们预先计算最优动作所需的所有信息，而无需实际采取任何行动。

Next: **在线规划**在在线规划过程中，代理对现实世界中的奖励或转换没有任何先验知识（仍然以马尔可夫决策过程 (MDP) 的形式表示）。在在线规划中，代理必须尝试探索，在此期间它会执行操作并接收反馈，反馈的形式包括它到达的后继状态及其获得的相应奖励。代理会利用这些反馈，通过一个 RL 来估计最优策略，然后再利用这个估计的策略进行开发或奖励最大化。

Setup - 仍然假设是一个马尔可夫决策过程 (MDP)：- 一个状态集合 $s \in \mathcal{S}$ - 一个行动集合（每个状态）$\mathcal{A}$ - 一个模型 $T(s,a,s')$ - 一个奖励函数 $R(s,a,s')$ - 仍然在寻找一个策略 $\pi(s)$

- 不知道 T 或 R

0.28 1. Model-Based Learning



根据大数定律，随着我们通过让代理经历更多场景收集到越来越多的样本，我们的 $\hat{T}$ 和 $\hat{R}$ 模型将会不断改进，其中 $\hat{T}$ 会收敛到 T，而 $\hat{R}$ 则会随着我们发现新的 (s,a,s') 组而获取之前未发现的奖励知识。在合适的时候可以结束代理的训练，通过使用当前的 $\hat{T}$ 和 $\hat{R}$ 模型运行价值或策略迭代来生成策略 $\pi_{exploit}$，并使用 $\pi_{exploit}$ 进行利用，让代理遍历 MDP，采取行动寻求奖励最大化。

0.29 2. Model-Free Learning

Big Picture * Passive reinforcement learning: Direct evaluation and temporal difference learning * 在被动强化学习中，智能体被赋予一个策略，并在经历事件的过程中学习该策略下状态的值，这正是当 T 和 R 已知时，MDP 的策略评估所做的事情。* Active reinforcement learning: Q-learning * 在此过程中，学习智能体可以使用收到的反馈在学习过程中迭代更新其策略，直到经过充分探索后最终确定最优策略。

Regret: 一开始就在环境中采取最优行动所积累的总奖励与通过运行学习算法所积累的总奖励之间的差异

0.29.1 2.1. Direct Evaluation (Passive RL)

回顾第一轮，我们可以看到从状态 D 到终止，我们获得的总奖励为 10，从状态 C 获得的总奖励为 $(-1)+10=9$，从状态 B 获得的总奖励为 $(-1)+(-1)+10=8$。完成此过程可得出每个状态下各轮的总奖励及其估计值，如下所示：

s	Total Reward	Times Visited	$V^{\pi}(s)$
A	-10	1	-10
B	16	2	8
C	16	4	4
D	30	3	10
E	-4	2	-2

计算得到 $V^{\pi}(E)=-2$ 和 $V^{\pi}(B)=8$，但理论上 B 和 E 在 π 下应该具有相同的值。

这是因为我们的代理处于状态 C 的 4 次中，它转换到 D 并获得了 3 次 10 的奖励，转换到 A 并获得了 1 次 -10 的奖励。纯属偶然的是，当它唯一一次获得 -10 奖励时，它的初始状态是 E，这严重扭曲了 E 的估值。

经过足够多的回合后，B 和 E 的值将收敛到真实值，但这种情况会导致该过程耗时超过我们的预期。可以通过选择使用我们的第二种被动强化学习算法——时间差分学习来缓解这个问题。

0.29.2 2.2. Temporal Difference Learning (Passive RL)

TD Learning 采用从每一次经验中学习的理念，而不是像 Direct Evaluation 那样简单地跟踪总奖励和状态访问次数，并在最后进行学习。

Sample of V(s):

$$\text{sample}=R(s,\pi(s),s')+\gamma V^{\pi}(s')$$

Update to V(s):

$$V^{\pi}(s)\leftarrow(1-\alpha)V^{\pi}(s)+(\alpha)\text{sample}$$

其中 learning rate $\alpha\in[0,1]$，一般从 1 慢慢缩小到 0.

0.29.3 2.3. Q-Learning (Active RL)

TD Learning 或 Direct Learning 通常与一些基于模型的学习结合使用，以获取 T 和 R 的估计值，从而有效地更新学习代理所遵循的策略。Q Learning 提出直接学习状态的 Q 值，无需了解任何值、转换函数或奖励函数。因此，Q 学习完全无需模型（off-policy learning）。

Q 学习使用以下更新规则来执行所谓的 Q 值迭代：

$$Q_{k+1}(s,a)\leftarrow\sum_{s'}T(s,a,s')[R(s,a,s')+\gamma\max_{a'}Q_k(s',a')]$$

- Learn $Q(s,a)$ values as you go - Receive a sample (s,a,s',r) - Consider your old estimate: $Q(s,a)$ - Consider your new sample estimate:

$$\text{sample}=R(s,a,s')+\gamma\max_{a'}Q(s',a')$$

- Incorporate the new estimate into a running average:

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + (\alpha)[\text{sample}]$$

0.29.4 2.4. Approximate Q-Learning

Q-Learning 占用内存太大，我们可以用特征向量简化：

$$V(s) = w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s)$$

$$Q(s,a) = w_1 f_1(s,a) + w_2 f_2(s,a) + \cdots + w_n f_n(s,a)$$

$$\text{difference} = \left[r + \gamma \max_{a'} Q(s',a')\right] - Q(s,a)$$

$$Q(s,a) \leftarrow Q(s,a) + \alpha[\text{difference}]$$

$$w_i \leftarrow w_i + \alpha[\text{difference}] f_i(s,a)$$

0.30 3. Exploration vs. Exploitation / 探索 vs. 利用

- Exploration - 通过尝试不同的行为来得到一个最佳的策略，得到最大奖励的策略。
- Exploitation - 不去尝试新的东西，就采取已知的可以得到很大奖励的行为。

0.30.1 3.1. ϵ Greedy

遵循 ϵ Greedy 的代理定义了某个概率 $0 \leq \epsilon \leq 1$，并且会以概率 ϵ 随机行动和探索。

如果为 ϵ 选择了一个较大的值，那么即使在学习了最优策略之后，代理仍然会以随机的方式行事。同样，为 ϵ 选择了一个较小的值意味着代理将很少进行探索，从而导致 Q 学习（或任何其他选定的学习算法）非常缓慢地学习最优策略。

为了解决这个问题，必须手动调整 ϵ 并随着时间的推移降低才能看到结果。

0.30.2 3.2. Exploration Function

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha \cdot [R(s,a,s') + \gamma \max_{a'} f(s',a')]$$

其中 f 表示探索函数。设计探索函数有一定的灵活性，但通常的选择是使用：

$$f(s,a) = Q(s,a) + \frac{k}{N(s,a)}$$

Chapter 3: Probabilistic Models and Machine Learning

COMPSCI 188: Introduction to AI

Included lectures

- Lecture 11-14: Probability and Bayes Nets
- Lecture 15&16: Bayes Net Inference & Sampling
- Lecture 17: Decision Networks and VPI
- Lecture 18: HMMs
- Lecture 19: Particle Filters
- Lecture 20: Naive Bayes

Lecture 11-14: Probability and Bayes Nets

0.31 1. Probability Recap

- **Conditional probability**
$$P(x|y) = \frac{P(x,y)}{P(y)}$$
- **Product rule**
$$P(x,y) = P(x|y)P(y)$$
- **Chain rule**
$$P(X_1, X_2, \ldots, X_n) = P(X_1)P(X_2|X_1)P(X_3|X_1, X_2)\ldots$$
$$= \prod_{i=1}^{n} P(X_i|X_1, \ldots, X_{i-1})$$
- **X, Y independent if and only if:**
$$\forall x, y : P(x,y) = P(x)P(y)$$
- **X and Y are conditionally independent given Z if and only if:**
$$\forall x, y, z : P(x,y|z) = P(x|z)P(y|z) \qquad X \perp Y|Z$$

0.32 2. Bayesian Network Representation

E.g. 考虑一个模型，其中我们有如下五个二进制随机变量：

- **B:** Burglary occurs.
- **A:** Alarm goes off.
- **E:** Earthquake occurs.
- **J:** John calls.
- **M:** Mary calls.

贝叶斯网络图的结构编码了不同节点之间的条件独立关系。

贝叶斯网络节点之间的边并不意味着这些节点之间存在特定的因果关系，也不意味着变量之间必然相互依赖。它只是意味着节点之间可能存在某种关系。

0.33 3. D-Separation

在给定所有父节点的情况下，节点有条件地独立于图中所有祖先节点。(可以这样理解：对于网络中的任何一个节点，一旦我们知道了它的所有 "直接上级"（父节点）的状态，那么它的所有 "间接上级" 或 "更早的祖先"（祖先节点）是什么状态，就变得无关紧要了。)

0.33.1 3.1. Casual Chains 因果链

以上一个由三个节点组成的因果链。它表达了 X、Y 和 Z 的联合分布，如下所示：

$$P(x,y,z) = P(z|y)P(y|x)P(x)$$

我们可以做出这样的陈述：$X \perp\!\!\!\perp Z|Y$。回想一下，这种条件独立性意味着：

假设发生入室盗窃或地震时，警报会响起，并且 Mary 和 John 听到警报后会打电话。我们可以用下图来表示这些依赖关系。

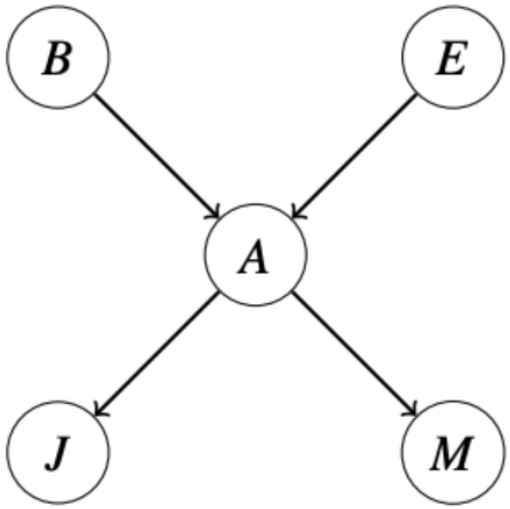


Figure 10: Bayes Net Example

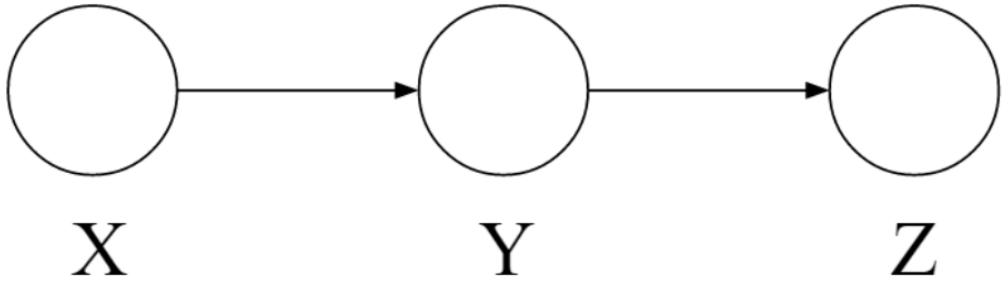


Figure 1: Causal Chain with no observations.

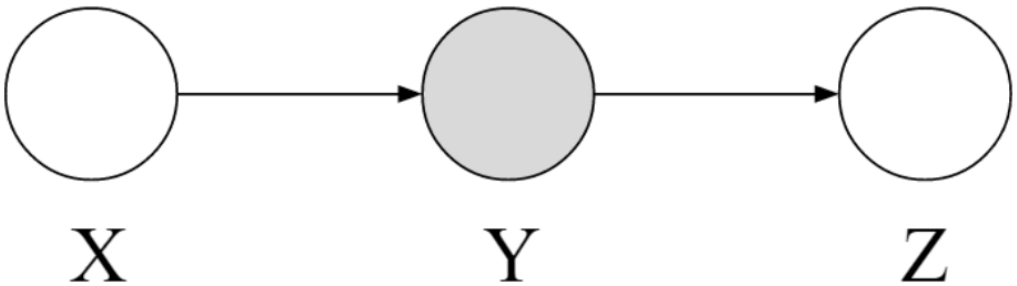


Figure 2: Causal Chain with Y observed.

Figure 11: casual chains

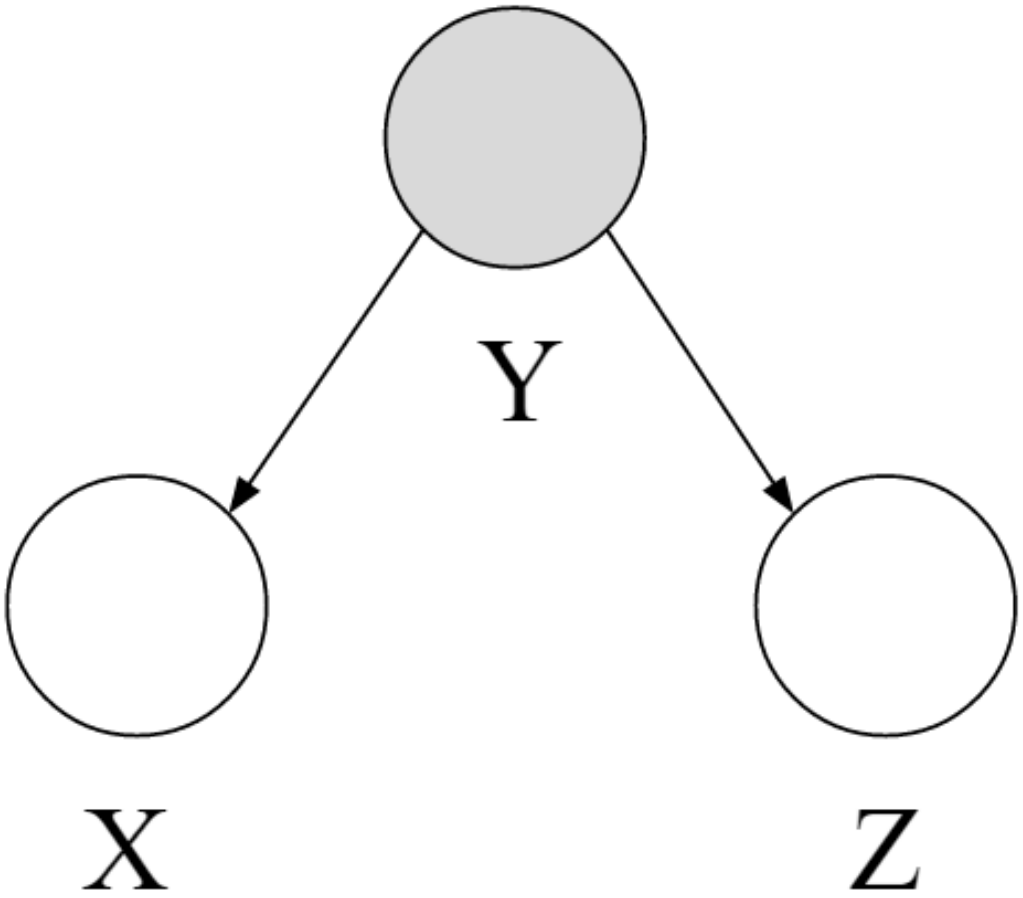


Figure 12: Common Cause

$$P(X|Z,Y) = P(X|Y)$$

可以证明这个说法如下：

$$P(X|Z,y) = \frac{P(X,Z,y)}{P(Z,y)} = \frac{P(Z|y)P(y|X)P(X)}{\sum_x P(x,y,Z)} = \frac{P(Z|y)P(y|X)P(X)}{P(Z|y)\sum_x P(y|x)P(x)} = \frac{P(y|X)P(X)}{\sum_x P(y|x)P(x)} = P(X|y)$$

0.33.2 3.2. Common Cause 共同原因

以上是一个由三个节点组成的因果链。它表达了 X、Y 和 Z 的联合分布，如下所示：

$$P(x,y,z) = P(z|y)P(y|x)P(x)$$

但是，我们可以做出这样的陈述：$X \perp\!\!\!\perp Z|Y$。回想一下，这种条件独立性意味着：

$$P(X|Z,Y) = P(X|Y)$$

我们可以证明这个说法如下：

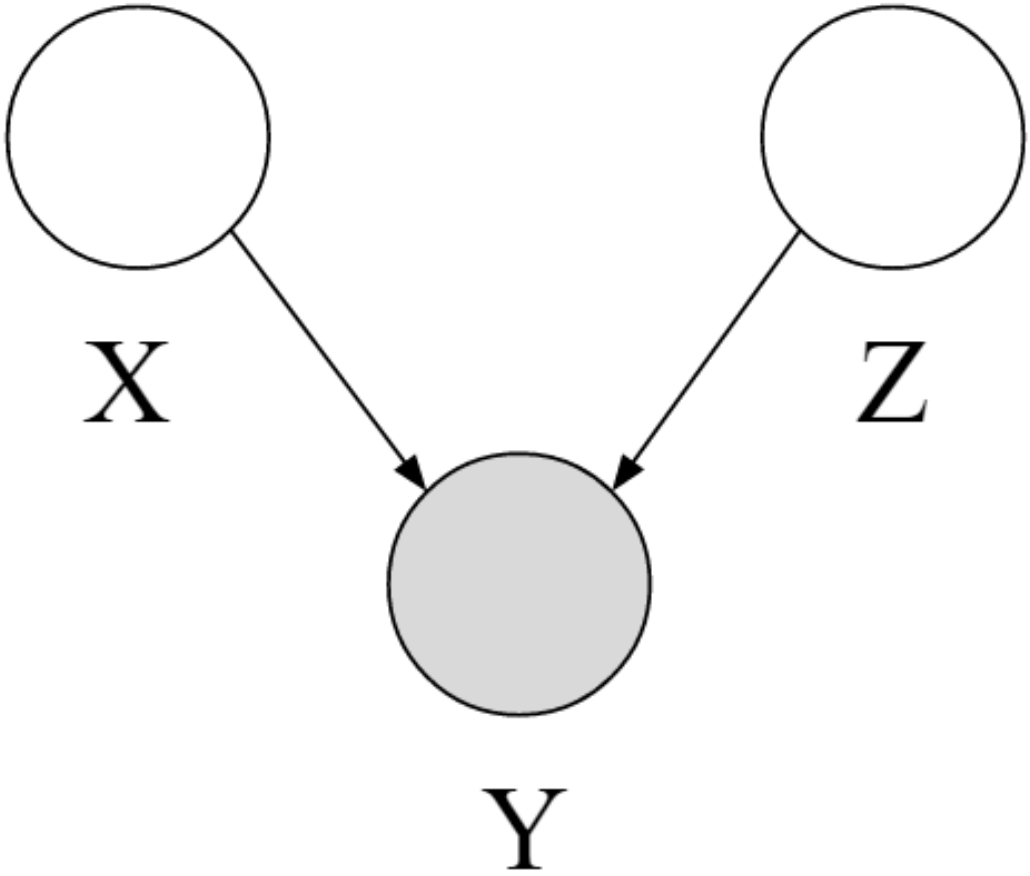


Figure 13: Common Effect

$$P(X|Z,y) = \frac{P(X,Z,y)}{P(Z,y)} = \frac{P(Z|y)P(y|X)P(X)}{\sum_x P(x,y,Z)} = \frac{P(Z|y)P(y|X)P(X)}{P(Z|y)\sum_x P(y|x)P(x)} = \frac{P(y|X)P(X)}{\sum_x P(y|x)P(x)} = P(X|y)$$

0.33.3 3.3. Common Effect 共同结果

0.34 待补充!!!

Lecture 15&16: Bayes Net Inference & Sampling

0.35 Recall: Bayes' Net Representation

- 有向无环图：Directed, Acyclic Graph, DAG
- 为每个节点指定一个 Conditional Probability Table, CPT

0.36 1. Inference 推断

0.37 Inference by Enumeration 枚举法

0.38 Variable Elimination 变量消除

Idea: 逐个消除隐藏变量

Procedure * Joining factors * Marginalization

0.39 2. Sampling

Approximate Inference by Sampling → faster!

0.40 Prior Sampling 先验采样

目标：得到联合概率 $P(x_1, x_2, \ldots, x_n)$

Generating samples:

$$S_{PS}(x_1...x_n) = \prod_{i=1}^{n} P(x_i|Parents(X_i))$$

这个乘积恰好等于 BN 所编码的联合概率分布 $P(x_1, ..., x_n)$ → 一致性。

通过计算某个事件 $x_1...x_n$ 出现的次数 $N_{PS}(x_1...x_n)$，除以总样本数 N，来估计这个事件的概率 $\hat{P}(x_1...x_n)$。

$$\lim_{N\to\infty} \hat{P}(x_1...x_n) = \lim_{N\to\infty} \frac{N_{PS}(x_1...x_n)}{N}$$

0.41 Rejection Sampling 拒绝采样

目标：得到条件概率 $P(Q|E = e)$

假设我们想计算以下条件概率（即查询）：

$$P(C = \text{True}|R = \text{True})$$

假设我们一共进行了 $N = 1000$ 次先验采样，其中只有 $N_{\text{Accept}} = 300$ 个样本接受（即 $R = \text{True}$）。

我们只关注这 300 个接受样本：在这 300 个样本中，我们统计 $C = \text{True}$ 出现了 $N_{C=\text{True}} = 200$ 次。估计结果：

$$\hat{P}(C = \text{True}|R = \text{True}) = \frac{N_{C=\text{True}}}{N_{\text{Accept}}} = \frac{200}{300} \approx 0.67$$

0.42 Likelihood Weighting 似然加权采样

目标：$P(C|R = \text{True})$

只对非证据变量 $\mathbf{z}$ 进行先验采样：

$$S_{WS}(\mathbf{z}, \mathbf{e}) = \prod_{i=1}^{l} P(z_i|\text{Parents}(Z_i))$$

用于补偿未对证据变量进行采样的权重：

$$w(\mathbf{z}, \mathbf{e}) = \prod_{i=1}^{m} P(e_i|\text{Parents}(E_i))$$

二者结合：

$$S_{WS}(\mathbf{z}, \mathbf{e}) \cdot w(\mathbf{z}, \mathbf{e}) = \left(\prod_{i=1}^{l} P(z_i|\text{Parents}(z_i))\right)\left(\prod_{i=1}^{m} P(e_i|\text{Parents}(e_i))\right)$$

0.43 Gibbs Sampling

目标：解决 LW 的单向性问题（只能从父节点向子节点采样）

吉布斯采样只对**非证据变量**进行操作。- 初始化：随机为所有非证据变量 $\mathbf{Z} = \{Z_1, Z_2, \ldots, Z_k\}$ 赋予一个初始值。- 迭代（生成样本）：重复以下步骤 N 次：循环遍历所有非证据变量 $Z_i \in \mathbf{Z}$。- 局部条件采样：对于选定的变量 Z_i，从它的条件概率分布中采样一个新的值 z_i'。这个条件是给定所有其他变量当前状态和所有证据的分布。

$$Z_i^{\text{new}} \sim P(Z_i|Z_1, \ldots, Z_{i-1}, Z_{i+1}, \ldots, Z_k, \mathbf{E} = \mathbf{e})$$

- 更新状态：用新的值 z_i' 替换旧值 z_i。

Lecture 17: Decision Networks and VPI

0.44 1. Decision Network

MEU: Choose the action which max(expected utility)

0.45 2. Value of Information (VPI)

- 核心思想：计算获取新证据（信息）的价值。
- 定义：信息价值是由于先揭示证据 E' 再行动，相比于现在就行动，所带来的 MEU 的预期增益。

$$VPI(E'|e) = MEU(e, E') - MEU(e)$$

0.46 2.1. Properties

- Nonnegative
- Nonadditive
- Order Independent

0.47 2.2. Imperfect Information

0.48 3. POMDPs

- MDPs have: $S, A, T(s, a, s'), R(s, a, s')$
- POMDPs add: Observations O, Observation function $O(s, o)$

Lecture 18: HMMs

0.49 1. Markov Models

- 通常需要对一系列观测值进行推理（如 $X_1, X_2, ..., X_n$）→ Need to introduce time (or space) into our models.

0.50 1.1. Definitions and Properties

Transition: $P(X_t|X_{t-1})$

- Markovness: 给定当前状态 (X_t)，过去和未来是独立的。每个时间步仅依赖于前一个时间步。
- 平稳性假设 (Stationarity)：转移概率在所有时间都是相同的。

0.51 1.2. Mini-Forward Algorithm

0.52 1.3. Stationary Distributions

- 马尔科夫链最终收敛到的分布。
- 对于大多数链，最终分布独立于初始分布，且初始分布的影响会越来越小。

$$P_\infty(X) = P_{\infty+1}(X) = \sum_x P(X|x)P_\infty(x)$$

e.g. weather

$$\begin{aligned} P_\infty(sun) &= P(sun|sun)P_\infty(sun) + P(sun|rain)P_\infty(rain) \\ P_\infty(rain) &= P(rain|sun)P_\infty(sun) + P(rain|rain)P_\infty(rain) \\ P_\infty(sun) &= 0.9P_\infty(sun) + 0.3P_\infty(rain) \\ P_\infty(rain) &= 0.1P_\infty(sun) + 0.7P_\infty(rain) \\ P_\infty(sun) &= 3P_\infty(rain) \\ P_\infty(rain) &= 1/3P_\infty(sun) \\ \text{Also: } P_\infty(sun) + P_\infty(rain) = 1 &\implies P_\infty(sun) = 3/4 \\ P_\infty(rain) &= 1/4 \end{aligned}$$

0.53 2. Hidden Markov Models, HMMs

- 标准的马尔科夫链对于需要根据观测值更新信念的智能体不够实用
- HMMs：状态 X 是隐藏的，观测到输出 E

0.54 2.1. Definition

- 初始分布 $P(X_1)$
- 转移 $P(X_t|X_{t-1})$
- 观测 $P(E_t|X_t)$

0.55 2.2. Forward Algorithm

- 解决滤波/监测（Filtering / Monitoring）任务
- 前向算法的本质是在线信念更新（Online Belief Updates），它在每个时间步同时结合了两个主要操作：时间流逝（预测）证据更新（观测）

步骤 1：时间流逝（预测）（积累不确定性）在获取新的证据 e_t 之前，首先需要根据马尔科夫转移模型，将上一步的信念 $P(X_{t-1}|e_{1:t-1})$ "推" (pushed) 过转移模型，以预测当前时刻 t 的状态分布 $P(X_t|e_{1:t-1})$。

$$\mathbf{P}(x_t|e_{1:t-1}) = \sum_{x_{t-1}} P(x_{t-1}|e_{1:t-1}) \cdot P(x_t|x_{t-1})$$

步骤二：证据更新（观测）（重新加权，减少不确定性）

$$\mathbf{P}(x_t|e_{1:t}) \propto_X P(x_t|e_{1:t-1}) \cdot P(e_t|x_t)$$

Lecture 19: Particle Filters

0.56 Recap

- Markov Models and Hidden Markov Models 的结构
- Inference
  - Exact Inference: Filtering

0.57 1. Approximate Solution: Particle Filtering

当状态空间 $|X|$ 过大或连续时（如机器人位置的连续坐标），精确推理方法（如前向算法）不可行。粒子滤波提供了一个近似解决方案。

0.58 1.1. 原理

用一个包含 N 个**粒子（样本）**的列表来近似表示分布 $P(X)$。- 每一步的计算时间与 N 成线性关系。- 更多的粒子意味着更高的准确性。

0.59 1.2. 三步骤

a. 时间流逝 Elapse Time 每个粒子 x 根据转移模型 $P(X'|x)$ 采样其下一个位置 x'

$$x' = \text{sample}(P(X'|x))$$

b. 权重 Weight 观测值 e 是固定的。每个粒子 x 根据其与证据的一致性被赋予一个权重 $w(x) = P(e|x)$。（此时所有样本都被降权（downweight），它们的权重总和不再是 1。）

c. 重采样 Resample 从当前的加权样本分布中带替换地抽取 N 次，创建一组新的非加权粒子。权重高的粒子更有可能被选中。

等同于在**归一化**。

Lecture 20: Naive Bayes

0.60 1. ML

之前的部分更关注如何使用已有的模型做出最优的决策。

机器学习是关于如何从数据或经验中获取模型。学习内容：（获取模型的过程）包括：- 学习参数（例如，学习概率）。- 学习结构（例如，贝叶斯网络图）。- 学习隐藏概念（例如，聚类、神经网络）。

0.61 2. Classification 分类任务

核心目标：给定输入 x，预测其标签（类别）y。

0.62 2.1. Model-based Classification

建立一个模型（例如贝叶斯网络），将输出标签和输入特征都视为随机变量。