中山大學
SUN YAT-SEN UNIVERSITY

人工智能

笔记整理

姓名：刘斯宇
学号：17341110

目录

☒ 第一章--人工智能概述

1 人工智能概述

人工智能简介

人工智能是研究如何将人的智能转化为机器智能，或者是用机器来模拟或实现人的智能。

人工智能字面上的意义是智能的人工制品。

2 样例学习

2.1 K 近邻分类

KNN

KNN 最近邻算法来解决分类问题的主要思路就是在训练过程中将多个已经标记好了的样例的 Label 标记下来，然后在预测一个新的样本 T 的分类的时候，分别算出该样本与之前训练集中的所有样本的距离，找出最近的 K 个样例，将这 K 个样例中出现次数最多的 Label 作为该新样本的 Label。

KNN 算法认为所有属性都具有同等重要的地位。

2.2 决策树

决策树

分类决策树模型是对实例进行分类的树形结构。由结点和有向边组成。其中内部结点表示一个属性，叶子节点表示分类的一个类，有向边是属性的分类依据。

2.2.1 ID3 算法

1. 计算数据集 D 的经验熵 $H(D)$

$$H(D) = -\sum_{k=1}^{K} \frac{|C_k|}{|D|} \log_2 \frac{|C_k|}{|D|}$$

2. 计算特征 A 对数据集 D 的经验条件熵 $H(D|A)$

$$H(D|A) = \sum_{i=1}^{n} \frac{|D_i|}{|D|} H(D_i) = -\sum_{i=1}^{n} \frac{|D_i|}{D|} \sum_{k=1}^{K} \frac{|D_{ik}|}{|D_i|} \log_2 \frac{|D_{ik}|}{|D_i|}$$

3. 计算信息增益

$$g(D, A) = H(D) - H(D|A)$$

4. 选择信息增益最大的特征作为决策点

2.2.2 C4.5 算法

1. 计算特征 A 对数据集 D 的信息增益

$$g(D,A) = H(D) - H(D|A)$$

2. 计算数据集 D 关于特征 A 的值的熵 $H_A(D)$

$$H_A(D) = -\sum_{j=1}^{v} \frac{|D_j|}{|D|} \times \log\left(\frac{|D_j|}{|D|}\right)$$

3. 计算信息增益 $g_R(D,A)$

$$g_R(D,A) = \frac{g(D,A)}{H(D)}$$

4. 选择信息增益增益率最大的特征作为决策点

2.2.3 CART 算法

1. 计算特征 A 的条件下，数据集 D 的 $GINI$ 系数

$$\text{gini(D,A)} = \sum_{j=1}^{v} p\left(A_j\right) \times \text{gini}\left(D_j|A = A_j\right)$$

$$\text{gini}\left(D_j|A = A_j\right) = \sum_{i=1}^{n} p_i\left(1 - p_i\right) = 1 - \sum_{i=1}^{n} p_i^2$$

2. 选择 $GINI$ 系数最小的特征作为决策点

2.3 逻辑回归

为什么要引入逻辑回归模型？

逻辑回归模型是由线性回归发展而来的，但是线性回归有他致命的缺点，比如我们用线性回归来做二分类的问题的时候，$y = w_0 + \sum_{j=1}^{d} w_j x_j + u = \tilde{W}^T \tilde{X}$, 基于这个模型预测的 y 值，也就是样本属于某个类的概率会超过 0-1 的范围，所以我们就引入了逻辑回归。

2.4 PLA

PLA 的全称是 Percetron Learning Algorithm，又称感知机器学习。PLA 用于解决的是对于二维或者高维的线性可分问题的分类，最终将问题分为两类——是或者不是。感知机 (Perceptrons) 也是一种人工神经网络，是一种最简单形式的前馈式人工神经网络，是一种二元线性分类器。

2.4.1 算法实现

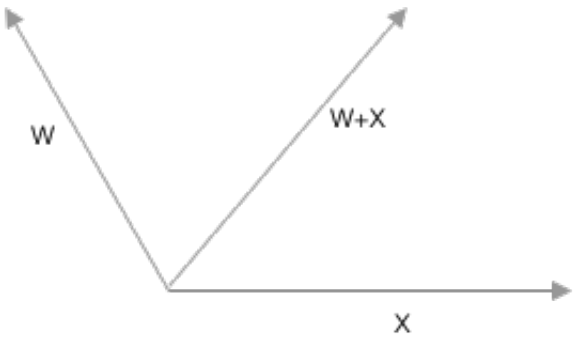
1. 先初始化 $W_{(0)}$, 然后根据 D 来修正 W
2. 遍历每一个数据 $(X_{i(t)}, Y_{i(t)})$, 直到找到 $\text{sign}\left(\tilde{\mathbf{W}}_{(t)}^{\mathrm{T}}\tilde{\mathbf{X}}_{i(t)}\right) \neq \mathcal{Y}_{i(t)}$
3. 如果找到错误的话，按照下面的方法进行 W 的修正

$$\tilde{\mathbf{w}}_{(t+1)} \leftarrow \tilde{\mathbf{w}}_{(t)} + y_{i(t)}\tilde{\mathbf{x}}_{i(t)}$$

4. 直到没有错误，返回最后的 W

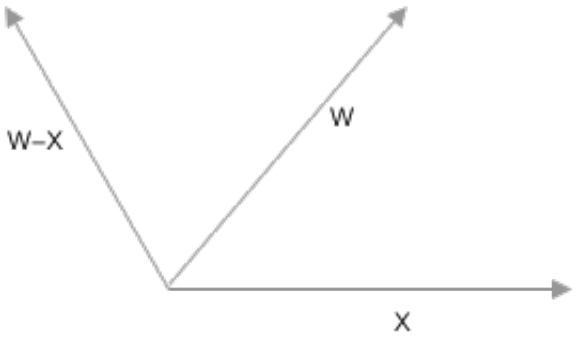
2.4.2 修正算法的正确性

1. 正样例被预测为负的情况下



如果所示，在正样例被预测为负的情况下，说明两个向量的内积为负，需要调整向量使得二者的值为正,$W + X$ 会更靠近 X, 从而使得该样例的错误率降低。

2. 负样例被预测为正的情况下



如果所示，在负样例被预测为正的情况下，说明两个向量的内积为正，需要调整向量使得二者的值为负,$W - X$ 会更靠近负, 从而使得该样例的错误率程度。

2.5 神经网络模型

人工神经网络的组成部分：

- 这些信号代表来自环境的数据或其他神经元的激活
- 一组实值权重。这些权重的值代表连接强度.
- 一个激活层。神经元的激活层由加权输入的总和确定
- 一个阈值函数 f. 该函数通过确定激活是低于还是高于阈值来计算最终输出.

2.5.1 神经网络模型

如果我们使用 $f(x) = \frac{1}{1+e^{-x}}$ 当作激活函数的话，那么我们来详细解释神经网络的前向传播和后向传播。

- 给定隐藏层或输出层的单元 j, 单位 j 的净输入 I_j 为

$$I_j = \sum_i w_{ij} O_i + \theta_j$$

w_{ij} 是从上-层单元 i 到单元 j 的连接权重;O_i 是上一层单元 i 的输出;θ_j 单元的偏置。

- 给定单元 j 的输入 I_j, 则单位 j 的输出 O_j 的公式为:

$$O_j = \frac{1}{1 + e^{-I_j}}$$

- 对于输出层中的单元 k，误差 Err_k 由下式计算:

$$Err_k = O_k(1 - O_k)(T_k - O_k)$$

T_k 为真实值。

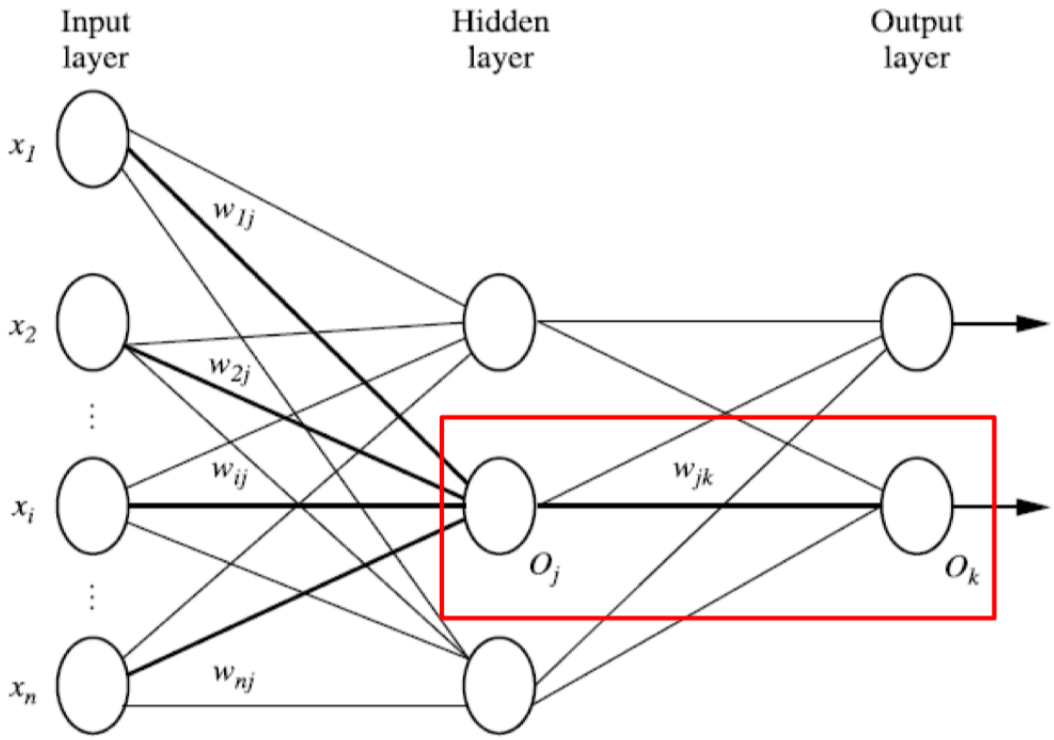
- 隐藏层单元 j 的误差为

$$\mathrm{Err}_j = O_j \left(1 - O_j\right) \sum_k \mathrm{Err}_k \, w_{jk}$$

- 权重跟新公式为

$$w_{jk} = w_{jk} + \eta \, \mathrm{Err}_k \, O_j$$

$$\theta_k = \theta_k + \eta \, \mathrm{Err}_k$$



详细的推导过程见 https://www.zybuluo.com/hanbingtao/note/476663

3 搜索

3.1 一致代价搜索

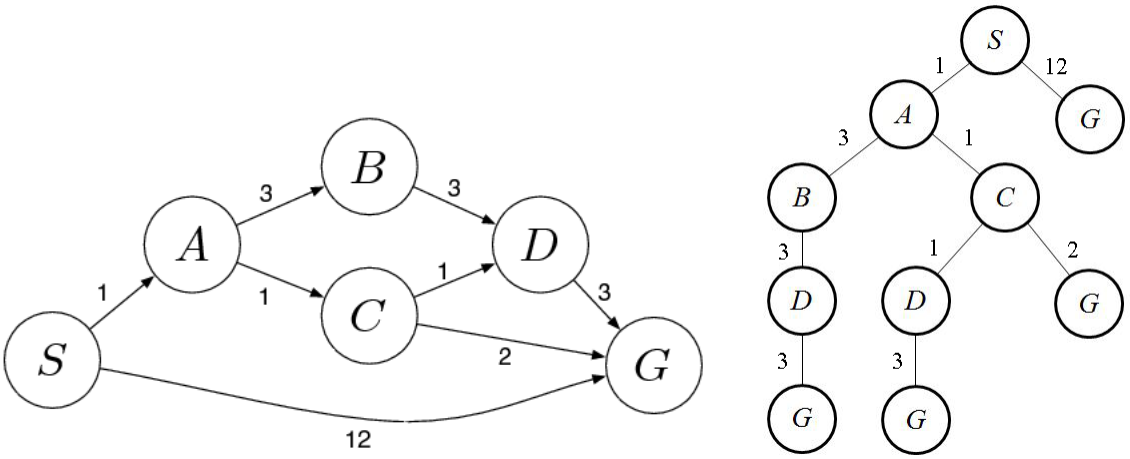
3.1.1 算法原理

一致代价搜索方式制定了一个路径消耗函数 g(n), 该函数描述根节点到 n 的路径消耗 (假设每个边上有相关的消耗值), 每次都扩展当前 g(n) 最小的边缘节点。

3.1.2 算法伪代码

```
function Uniform-cost-search():return a solution or a failure
  node <- a node with state = problem.initial.state ,path-cost = 0
  frontier <- a priority queue ordered by path-cost with node as
     element
  explored <- an empty set
  loop do
    if EMPTY(frontier) return failure
    node <- POP(frontier)
    if problem.SATISFY(node.state)
      return SOLUTION(node.state)
    add node.state to explored
    for each action in problem.ACTION(node.state) do
      child <- CHILD-STATE(node.state, action)
      if child.state nor in explored or frontier
        frontier <- INSERT(child.state)
```

3.1.3 例子



Initialization: { [S , 0] }

Iteration1: { [S->A , 1] , [S->G , 12] }

Iteration2: { [S->A->C , 2] , [S->A->B , 4] , [S->G , 12] }

Iteration3: { [S->A->C->D , 3] , [S->A->B , 4] , [S->A->C->G , 4] , [S->G , 12] }

Iteration4: { [S->A->B , 4] , [S->A->C->G , 4] , [S->A->C->D->G , 6] , [S->G , 12] }

Iteration5: { [S->A->C->G , 4] , [S->A->C->D->G , 6] , [S->A->B->D , 7] , [S->G , 12] }

Iteration6 gives the final output as S->A->C->G.

3.2 双向搜索

3.2.1 算法原理

双向广度优先搜索算法是对广度优先算法的一种扩展。广度优先算法从起始节点以广度优先的顺序不断扩展，直到遇到目的节点；而双向广度优先算法从两个方向以广度优先的顺序同时扩展，一个是从起始节点开始扩展，另一个是从目的节点扩展，直到一个扩展队列中出现另外一个队列中已经扩展的节点，也就相当于两个扩展方向出现了交点，那么可以认为我们找到了一条路径。

3.2.2 算法伪代码

```
初始状态和目标状态都知道，求初始状态到目标状态的最短距离；
利用两个队列，初始化时初始状态在1号队列里，目标状态在2号队列里，
并且记录这两个状态的层次都为0，然后分别执行如下操作：

```

a.若1号队列已空，则结束搜索，否则从1号队列逐个弹出层次为K(K >= 0)的状态；

i. 如果该状态在2号队列扩展状态时已经扩展到过，
那么最短距离为两个队列扩展状态的层次加和，结束搜索；

ii. 否则和BFS一样扩展状态，放入1号队列，
直到队列首元素的层次为K+1时执行b；

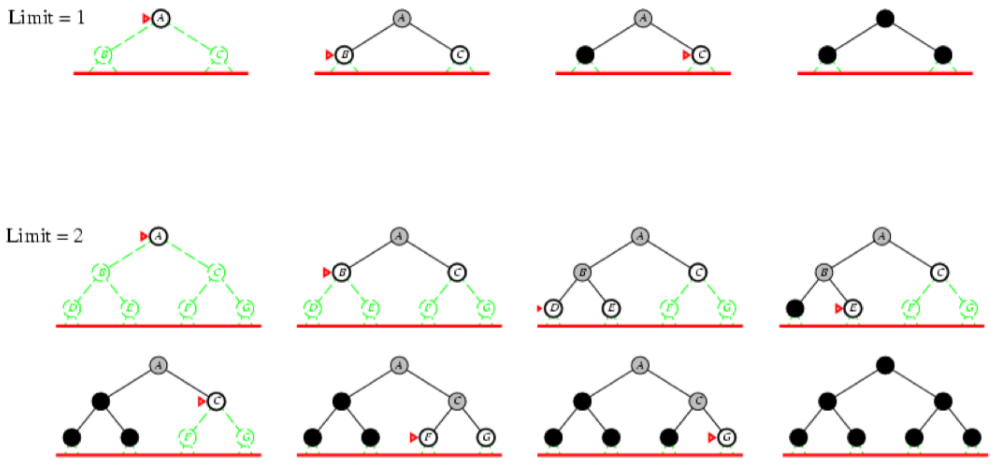
b.若2号队列已空，则结束搜索，否则从2号队列逐个弹出层次为K(K >= 0)的状态；

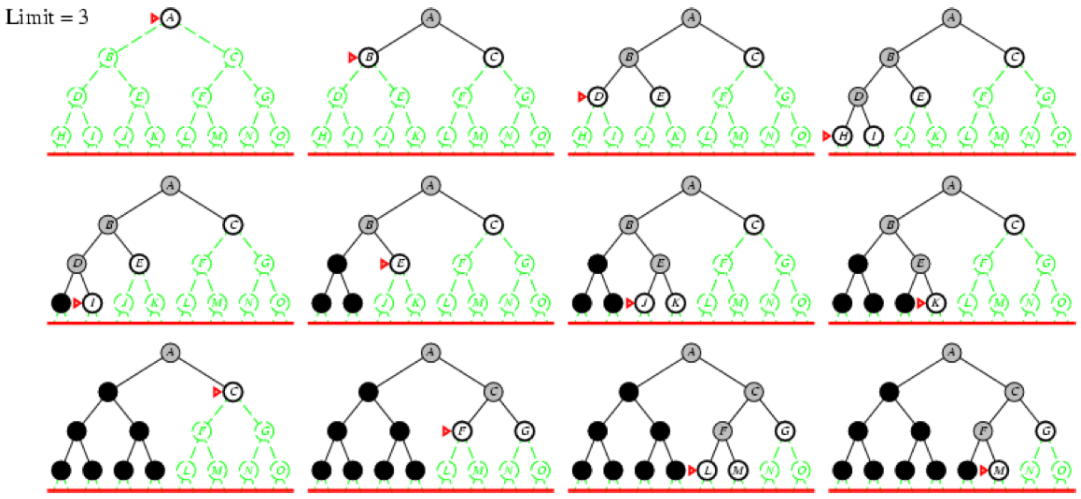
i. 如果该状态在1号队列扩展状态时已经扩展到过，
那么最短距离为两个队列扩展状态的层次加和，结束搜索；

ii. 否则和BFS一样扩展状态，放入2号队列，
直到队列首元素的层次为K+1时执行a；

3.3 迭代加深搜索

3.3.1 算法原理





3.3.2 算法伪代码

```
def IDA_Star(STATE startState):
    maxDepth = 0
    while true:
        if( DFS(startState, 0, maxDepth) ):
            return
        maxDepth = maxDepth + 1
```

3.4 A^* 搜索算法

3.4.1 算法原理

A* 算法通过下面这个函数来计算每个节点的优先级：

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

- f(n) 是节点 n 的综合优先级。当我们选择下一个要遍历的节点时，我们总会选取综合优先级最高（值最小）的节点。
- g(n) 是节点 n 距离起点的代价。

- h(n) 是节点 n 距离终点的预计代价，这也就是 A* 算法的启发函数。关于启发函数我们在下面详细讲解。

A* 算法在运算过程中，每次从优先队列中选取 f(n) 值最小（优先级最高）的节点作为下一个待遍历的节点。另外，A* 算法使用两个集合来表示待遍历的节点，与已经遍历过的节点，这通常称之为 open_set 和 close_set。

3.4.2 算法伪代码

```
* 初始化open_set和close_set;
* 将起点加入open_set中，并设置优先级为0（优先级最高）;
* 如果open_set不为空，则从open_set中选取优先级最高的节点n:
    * 如果节点n为终点，则:
        * 从终点开始逐步追踪parent节点，一直达到起点;
        * 返回找到的结果路径，算法结束;
    * 如果节点n不是终点，则:
        * 将节点n从open_set中删除，并加入close_set中;
        * 遍历节点n所有的邻近节点:
            * 如果邻近节点m在close_set中，则:
                * 如果此时的g(n)比之前的小，更新g(n)
                * 否则，跳过
            * 如果邻近节点m也不在open_set中，则:
                * 设置节点m的parent为节点n
                * 计算节点m的优先级
                * 将节点m加入open_set中
```

3.5 IDA^* 搜索算法

3.5.1 算法原理

IDA* 的基本思路是：首先将初始状态结点的 H 值设为阈值 maxH，然后进行深度优先搜索，搜索过程中忽略所有 H 值大于 maxH 的结点；如果没有找到解，则加大阈值 maxH，再重复上述搜索，直到找到一个解。带启发式的有限制的深度优先搜索，本质是在启发式限制下以不同的深度进行 dfs。在稀疏的有向图中深度优先效果往往优于广度优先，所以会好于 A* 算法，然而如果是棋盘类稠密问题，应该是 A* 更占优。由于不再采用动态规划的方法，内存占用少。

3.5.2 算法伪代码

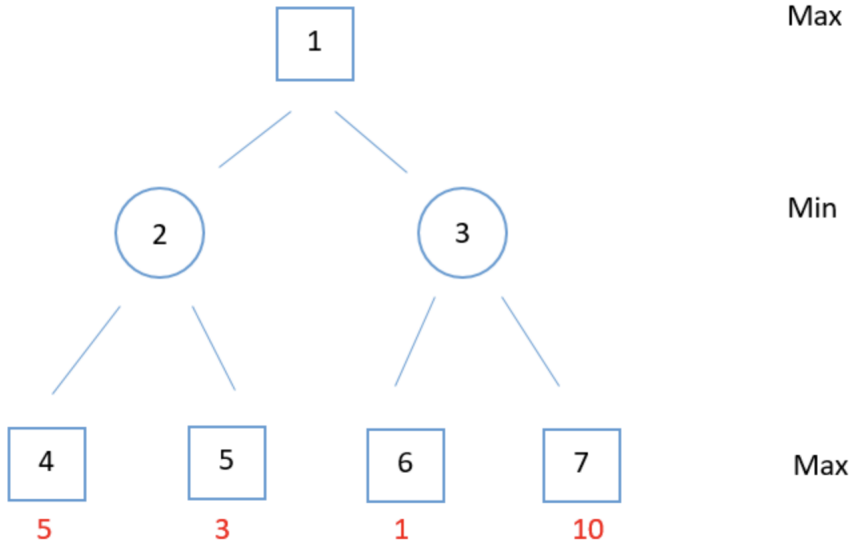
```
  function IDA_search(node, g, bound):
  // node当前节点
  // g当前节点的路径消耗值
  // bound当前搜索的一个界限值
  f = g + h(node)
  if f > bound:
      return f
  if node == B:
      return FOUND
  min = r
  for succ in successors(node) do:
      t = search(succ, g + cost(node, succ), bound)
  if t == FOUND:
      return FOUND
  if t < min:
      min = t
  return min

bound = h(A)
while(True):
  t = IDA_search(root, 0, bound)
  if t == FOUND:
      return bound
  if t =  :
      return NOT_FOUND
  bound := t
```

3.6 博弈树搜索

3.6.1 极大极小值搜索

极大极小值搜索过程的本质是回溯，以深度优先方式遍历多叉树。以棋类游戏为例，对于一个局面，一方有多种可能的落子法，对于它的任何一种落子情况，对方也都有多种落子，如此轮流行动。以多叉树形式表示，树的每一个节点表示一种可能的盘面，子节点表示所有可行方案，树的每一层代表一方，双方在树中交替出现。为了获取最后的胜利，一方要在所有可选项中做出能将其优势最大化的决策，另一方则选择令对手优势最小化的方法。极大极小值搜索过程中，假设每一步对方也会按照它的最佳方案行动；我们将博弈树中的层分成 Max 层和 Min 层，为了保证自己的收益最大化，Max 层节点会始终选择值最大的子节点来更新自己的值，Min 层则相反，始终选择值最小的子节点；因此这个算法被称为 Minimax。



上图中，有两类节点，分别代表博弈双方，

方块，Max 层节点，代表我方回合，他始终选择子节点中的最大值来更新自己的值；圆圈，Min 层节点，代表对方回合，他始终选择子节点中的最小值来更新自己的值；现在节点 1 有两种选择，它应该做出怎样的选择，就是 Minimax 算法要解决的问题。节点 1 处于 Max 层，它要选择 2、3 中评价值最高的那一个；节点 2 处于 Min 层，它取值为所有子节点中最小的值，也就是节点 5，所以节点 2 的值为 3；同理，节点 3 的值为 1；节点 1 的值为 3；所以节点 1 应该选择第一种下法；虽然第三层四个节点中，评价值最大的是节点 7，但是如果节点 1 选择第二种方案，到达节点 3，到对方轮次，此时决定权已经到对方手中，为了使利益最大化，对方会选择到节点 6；搜索的目标是选择一处收益最高的位置落子，上面的例子里，我们认为叶子结点的值是已知的，而实际上对于五子棋游戏，我们还不知道节点的值，为此，我们需要设定一个评判标准给节点打分，这就是后

面我们会提到的评价函数。

3.6.2 alpha-beta 剪枝

由于博弈树中，很多结点其实是“没有价值”的，所以无论是否遍历这些结点的数据，对最终的结果都毫无影响，进而我们选择 Alpha-Beta 剪枝剪掉那些没用的分支，当然，你也可以计算，如果不剪枝，那么一个简单的五子棋 AI 算法将会耗费多大的电脑资源，在此不做赘述，

剪枝算法的基本依据是：棋手不会做出对自己不利的选择。依据这个前提，如果一个节点明显是不利于自己的节点，那么就可以直接剪掉这个节点。前面说到，AI 会在 MAX 层分数最大的结点，而玩家会在 MIN 层选择最小结点，那么便可分析如下：在 MAX 层，假设当前层已经搜索到一个最大值 X，如果发现下一个节点的下一层 MIN 会产生一个比 X 还小的值，那么就直接剪掉此节点。简单来说，就是 AI 发现这一步是对玩家更有利的，所以不会走这一步。在 MIN 层，假设当前层已经搜索到一个最小值 Y，如果发现下一个节点的下一层 MIN 层会产生一个比 Y 还大的值，那么就直接剪掉此节点。其实 MAX 和 MIN 层道理相似。

3.6.3 估价函数

评估函数（也称评价函数）是博弈类游戏中 AI 的重要一部分，它决定了 AI 的决策步骤，主要任务是评估节点的重要程度，通俗的讲，评估函数为 AI 评估目前的对局形式，也为 AI 模拟每一步决策后的对局形式，并决策出于 AI 最有利、于对方最不利的一步。

```
AlphaBeta(n,Player,alpha,beta) //return Utility of state
  If n is TERMINAL
    return V(n) //Return terminal states utility
  ChildList =n.Successors(Player)
  If Player == MAX
    for c in ChildList
      alpha = max(alpha, AlphaBeta(c,MIN,alpha,beta))
      If beta <= alpha
        break
        return alpha
      Else //Player == MIN
        for c in ChildList
        beta = min(beta, AlphaBeta(c,MAX,alpha,beta))
```

```
        If beta <= alpha
          break
          return beta
```

4 约束满足问题

4.1 Backtracking Search

```
BT(Level)
  If all variables assigned
    PRINT Value of each Variable
    RETURN or EXIT (RETURN for more solutions) (EXIT for only one
       solution)
  V := PickUnassignedVariable()
  Assigned[V] := TRUE
  for d := each member of Domain(V) (the domain values of V)
    Value[V] := d
    ConstraintsOK = TRUE
    for each constraint C such that
      a) V is a variable of C and
      b) all other variables of C are assigned:
        ;(rarely the case initially high in the search tree)
      IF C is not satisfied by the set of current assignments:
        ConstraintsOK = FALSE
    If ConstraintsOk == TRUE:
      BT(Level+1)
  Assigned[V] := FALSE //UNDO as we have tried all of V' s values
  return
```

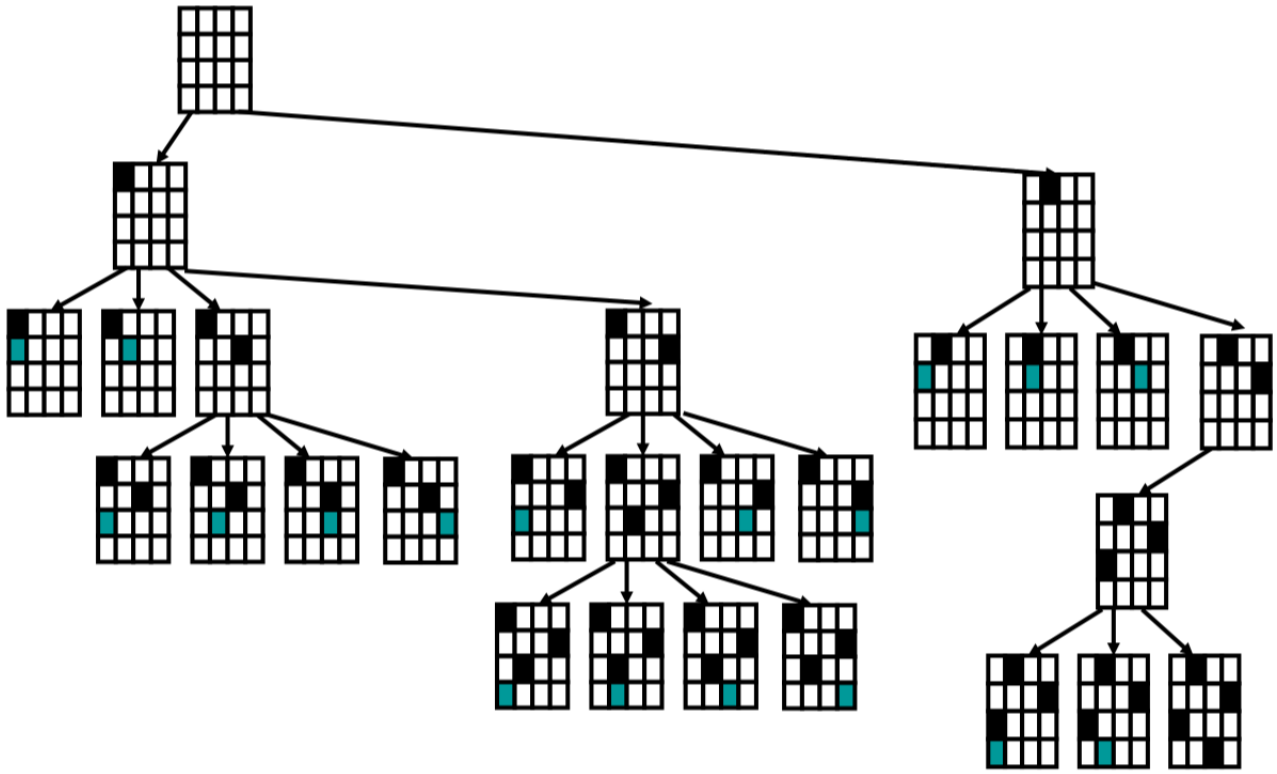


图 1: Four-Queen sloved by Backtracking Search

4.2 Forward Checking

```
FCCheck(C,x)
  // C is a constraint with all its variables already
  // assigned, except for variable x.
  for d := each member of CurDom[x]
    IF making x = d together with previous assignments to variables
       in scope C falsifies C
    THEN remove d from CurDom[x]
  IF CurDom[x] = {} then return DWO (Domain Wipe Out)
  ELSE return ok

FC(Level) /*Forward Checking Algorithm */
```

```
If all variables are assigned PRINT Value of each Variable RETURN
   or EXIT (RETURN for more solutions) (EXIT for only one solution
   )
V := PickAnUnassignedVariable() Assigned[V] := TRUE
for d := each member of CurDom(V)
  Value[V] := d
  DWOoccured:= False
  for each constraint C over V such that a) C has only one
     unassigned variable X in its scope
    if(FCCheck(C,X) == DWO) /* X domain becomes empty*/
      DWOoccurred:= True
      break /* stop checking constraints */
  if(not DWOoccured) /*all constraints were ok*/
    FC(Level+1) RestoreAllValuesPrunedByFCCheck()
  Assigned[V] := FALSE //undo since we have tried all of V' s
     values
  return;
```

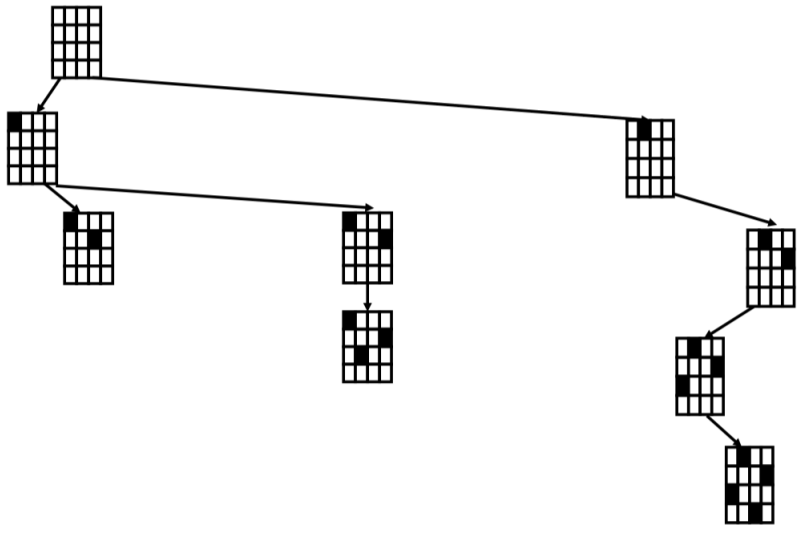


图 2: Four-Queen sloved by Forward Checking

4.3 GAC

```
GAC(Level) /*Maintain GAC Algorithm */
  If all variables are assigned
    PRINT Value of each Variable RETURN or EXIT (RETURN for more
       solutions) (EXIT for only one solution)
  V := PickAnUnassignedVariable() Assigned[V] := TRUE
  for d := each member of CurDom(V)
    Value[V] := d
    Prune all values of V   d from CurDom[V]
    for each constraint C whose scope contains V
      Put C on GACQueue
    if(GAC_Enforce() != DWO)
      GAC(Level+1) /*all constraints were ok*/
         RestoreAllValuesPrunedFromCurDoms()
  Assigned[V] := FALSE
  return;

GAC_Enforce()
  // GAC-Queue contains all constraints one of whose variables has
  // had its domain reduced. At the root of the search tree
  // first we run GAC_Enforce with all constraints on GAC-Queue
  while GACQueue not empty
    C = GACQueue.extract()
    for V := each member of scope(C)
      for d := CurDom[V]
        Find an assignment A for all other variables in scope(C)
           such that C(A U V=d) = True

        if A not found
          CurDom[V] = CurDom[V]  -  d if CurDom[V] =
            empty GACQueue
            return DWO //return immediately
          else
```

```
30          push all constraints C'  such that V \in scope(C' ) and  C
                '  \notin GACQueue on to GACQueue
31  return TRUE //while loop exited without DWO
```

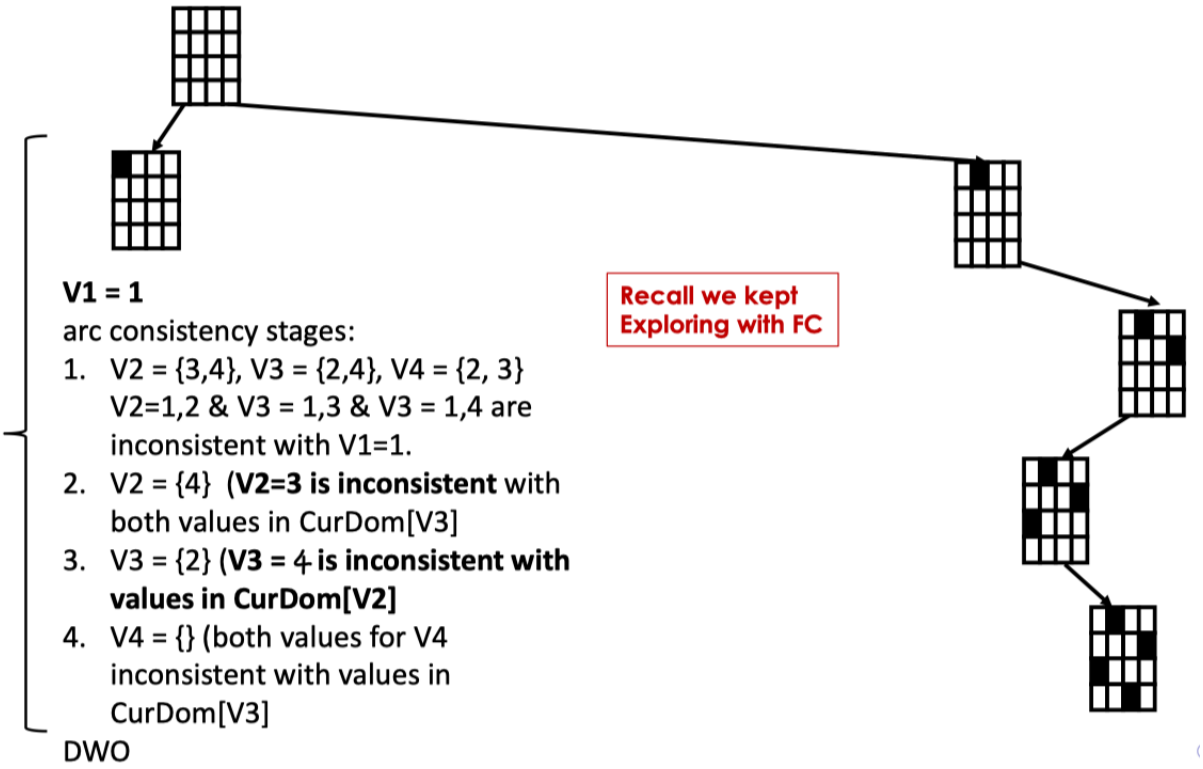


图 3: Four-Queens sloved by GAC