搜索算法

姚金宇

Outline

- 产生式系统
- 剪枝
- 博弈问题

最简单or最困难的算法

- 最简单
  - 入门算法
  - 所需基本知识不多，算法描述简单，思考直白
- 最困难
  - 搜索不难，时间不够
  - 如何选择搜索方法，如何选择顺序，如何剪枝
  - 难以举一反三

搜索问题

- 搜索问题：在一个空间中寻找目标
  - 搜索什么（目标）
  - 在哪里搜索（状态空间）
- 在搜索中，如何扩展状态空间是关键性问题
- 搜索可以根据是否使用启发式信息分成
  - 盲目搜索：只能区分当前状态是否为目标状态
  - 启发式搜索：在搜索过程中加入与问题相关的启发信息，指导搜索的方向

产生式系统

- 知识单元之间存在着因果关系，或者说前提和结论，一般用**产生式**（或者称为规则）来表示。
- 把一组产生式放在一起，让它们互相配合，协同作用，一个产生式生成的结论可以供另一个产生式作为前提使用，以这种方法求得问题的解决，这就叫做**产生式系统**。

产生式系统的组成部分

- **一个综合数据库(GlobleDatabase)**
  - 用来表述问题状态或有关事实，含有所求解问题的信息，其中有些部分可以是不变的，有些部分则可能只与当前问题的解有关。
- **一组产生式规则(Set of Rules)**
  - 产生式规则表示为：if……then…
  - 一条产生式规则满足了应用的先决条件之后，就可对综合数据库进行操作，使其发生变化。
- **一个控制系统(Control System)**
  - 规定如何选择一条可应用的规则对数据库进行操作，决定了问题求解过程的推理路线。

例子：八数码问题

- 在3*3组成的九宫格棋盘上，摆有八个将牌，每一个将牌都刻有1—8中的某一个数码。
- 棋盘中留有一个空格，允许其周围的某一个将牌向空格移动，这样通过移动将牌就可以不断改变将牌的布局。
- 这种游戏求解的问题是，给定一种初始的将牌布局或结构(称初始状态)和一个目标的布局(称目标状态)，问如何移动将牌，实现从初始状态到目标状态的转变。

八数码问题：综合数据库

- 选择一种数据结构来表示将牌的布局。通常可用来表示综合数据库的数据结构有符号串、向量、集合、数组、树、表格、文件等。
- 对八数码问题，选用二维数组来表示将牌的布局很直观，因此该问题的综合数据库可以如下形式表示:
  - (S_{ij})，其中1<=i、j<=3，$S_{ij} \in \{0,1,\ldots,8\}$，且$S_{ij}$互不相等。

八数码问题：产生式规则

- 改变状态有4种走法：
  - 空格左移、空格上移、空格右移、空格下移。
- 设$S_{i,j}$记矩阵第i行第j列的数码，i0、j0记空格所在的行、列数值，即$S_{i0,j0}=0$，则

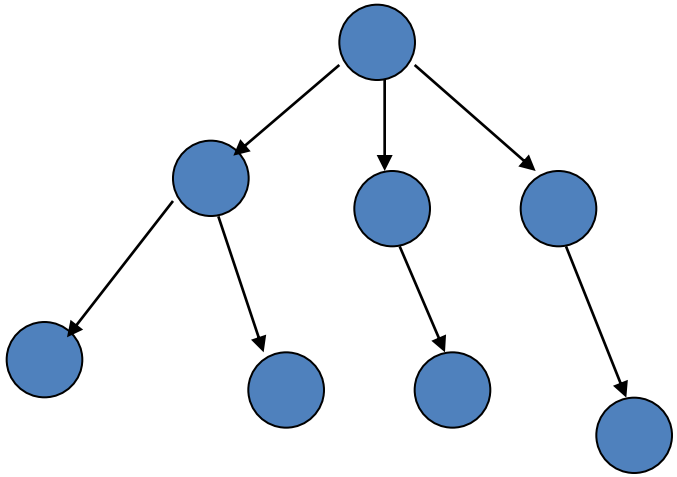
- 1.if j0-1≥1 then $S_{i0j0}:=S_{i0(j0-1)}$, $S_{i0(j0-1)}:=0$；(S_{i0j0}向左)
- 2.if i0-1≥1 then $S_{i0j0}:=S_{(i0-1)j0}$, $S_{(i0-1)j0}:=0$；(S_{i0j0}向上)
- 3.if j0+1≤3 then $S_{i0j0}:=S_{i0(j0+1)}$, $S_{i0(j0+1)}:=0$；(S_{i0j0}向右)
- 4.if i0+1≤3 then $S_{i0j0}:=S_{(i0+1)j0}$, $S_{(i0+1)j0}:=0$；(S_{i0j0}向下)

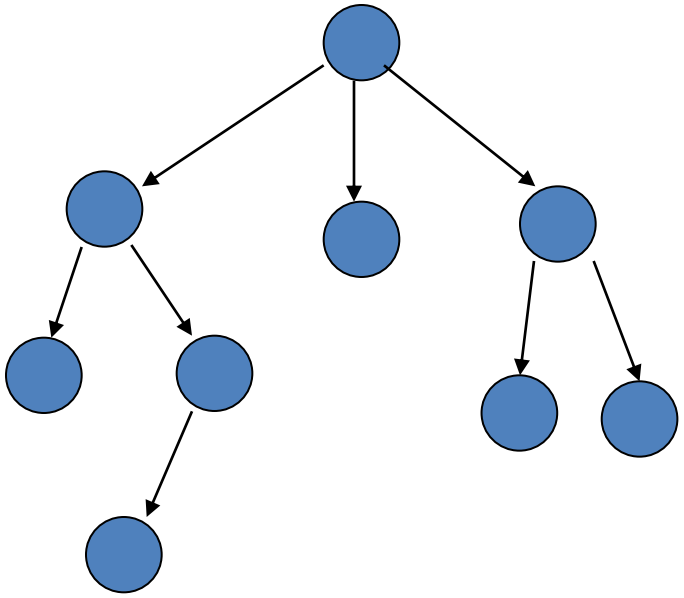
八数码问题：控制系统（搜索策略）

- 深度优先的搜索策略
- 广度优先的搜索策略
- 启发式方法
- ……

盲目搜索策略

- 宽度优先搜索（Breadth-first search）
- 深度优先搜索（Depth-first search）





宽度优先搜索

- 优点
  - 目标节点如果存在，用宽度优先搜索算法总可以找到该目标节点，而且是最小（即最短路径）的节点
- 缺点
  - 当目标节点距离初始节点较远时，会产生许多无用的节点，搜索效率低

例 Jack and Jill（POJ1729）

- Jack和Jill要从各自的家走到各自的学校，但是他们互相不喜欢对方，因此希望你找到一个行走方案，使得在行走的过程中他们之间的直线最近距离最远。
- 单位时间内每个人都可以从一个格子转移到相邻的（上下左右）四个格子之一，也可以停止不动。
- 计算距离的时候只算单位时间移动结束之后两人所在位置的直线距离，不考虑移动过程中的距离。

- 图例：
  - H: Jack的家；S:Jack的学校
  - h:Jill的家；s:Jill的学校
  - *:不可进入的区域
  - .:可进入的区域

```
..........
...H......
.**...s...
.**.......
.**.......
.**.......
.**.......
.**.......
...S..h..*
..........
```

Jack and Jill：综合数据库&产生式规则

- 将当前Jack和Jill某一时刻各自所在的格子对作为状态。
- 预处理
  - 将状态按照格子对的直线距离从大到小排序。

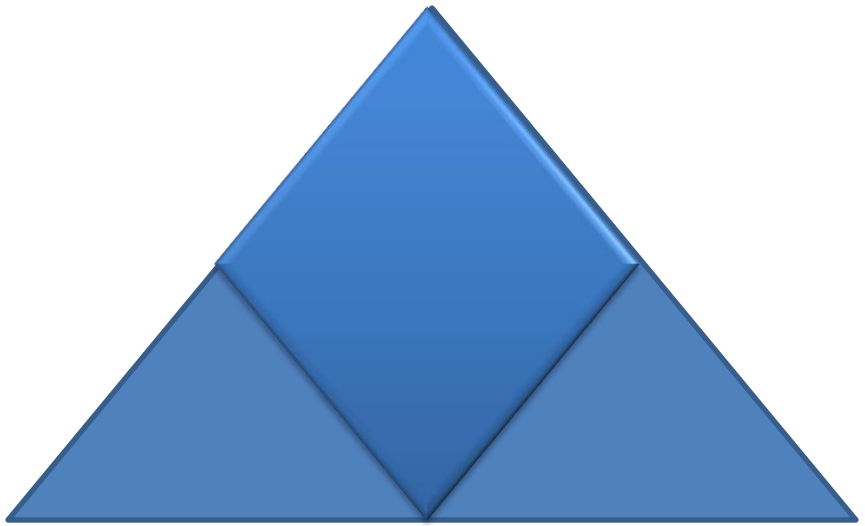
- 产生式规则: (格子对➔格子对)的转移

Jack and Jill：控制系统

- 从大到小枚举最近距离k，维护一个可搜索状态集合T，T中包含所有格子对距离大于等于k的状态。
- 按照格子对的距离从大到小将综合数据库中的状态添加入集合T。
- 对于当前的可搜索状态T，采用宽度优先搜索，判断是否能从初始状态到达目标状态。
- 问题：
  - 为什么采用宽度优先搜索？
  - K的初始值？
  - 当距离k减小时，是否要重新搜索？

双向宽度优先搜索

- 从初始结点和目标结点分别做广度优先搜索，每次检查两边是否重合。
- Bonus: 每次扩展结点总是选择结点比较少的那边进行下次搜索，并不是机械的两边交替。



Jack and Jill：双向宽搜

- 从(H, h)和(S, s)分别作为搜索的初始节点，同时进行搜索。
- 如果两个搜索集合有交集，则搜索结束。

深度优先搜索

- 深度优先搜索也称为回溯搜索
  - 回溯：能进则进，进不了则换，换不成则退
- 优点
  - 空间需求少，深度优先搜索的存储器要求是深度约束的线性函数
- 问题
  - 可能搜索到错误的路径上，在无限空间中可能陷入无限的搜索
  - 最初搜索到的结果不一定是最优的

例：SHIPS（ POJ1138）

- 一个类似海战棋的游戏。在一个n*m的海面上，摆放着7搜船（船不会重叠）。船的形状如下图所示（可以旋转，不能翻转）。你的目标是猜出海面上那些“小块”被船占据，并“翻开”那些小块。
- xx xx xx x x x
  xx xx xx xxx xxx xxx xxxx
- 现在你已经翻开了一部分小块，得到了一些结果。现在问你，是否可以在只错一次的情况下，完成任务（即翻开所有船）
- -- Ships

例：SHIPS

- 一个输入样例：

```
10 10
.x..x.....
oooooxoooo
oxooxxx...
xxooooooo..
xoooxooo..
ooxxxxoo..
ooooooxxox
oooooooxox
ooooooooxx
oooooooooo
```

例：SHIPS

- 该题有什么特点？
- 那么用什么搜索方法？
- 用什么搜索顺序？
- 如何处理“可以错一次”

例：SHIPS

- 两个过程：
  - 猜测过程
  - 判断是否唯一覆盖
- 猜测过程：
  - 选择一个还未揭开的单元格
  - 假设是’o’，并且唯一覆盖
  - 假设时’x’，并且错一次可以唯一覆盖（递归）
- 判断是否唯一覆盖
  - 如果出现第二个可行解则false

迭代加深搜索

- 算法思路
  - 总体上按照深度优先算法方法进行
  - 对搜索深度需要给出一个深度限制dm，当深度达到了dm的时候，如果还没有找到解答，就停止对该分支的搜索，换到另外一个分支继续进行搜索。
  - dm从小到大依次增大（因此称为迭代加深）
- 迭代加深搜索是最优的，也是完备的
- 重复搜索深度较小的节点，是否浪费？

例：旋转游戏(POJ2286)

在如下图的棋盘中，摆放着8个1，8个2和8个3，每一步你可以沿着A、B、C、D、E、F、G、H任意一个方向移动该字母所指的长块移出边界的小块会放在最后一个（就好象魔方一样）。如图，最左边的期盼沿着A移动一次（我们不妨称之为操作A），就会变成中间的棋盘布局，再进行操作C，就会变成右边的棋盘布局。

你需要设法使的最中间的8个格子的数字相同，问最少需要多少步。如何移动？

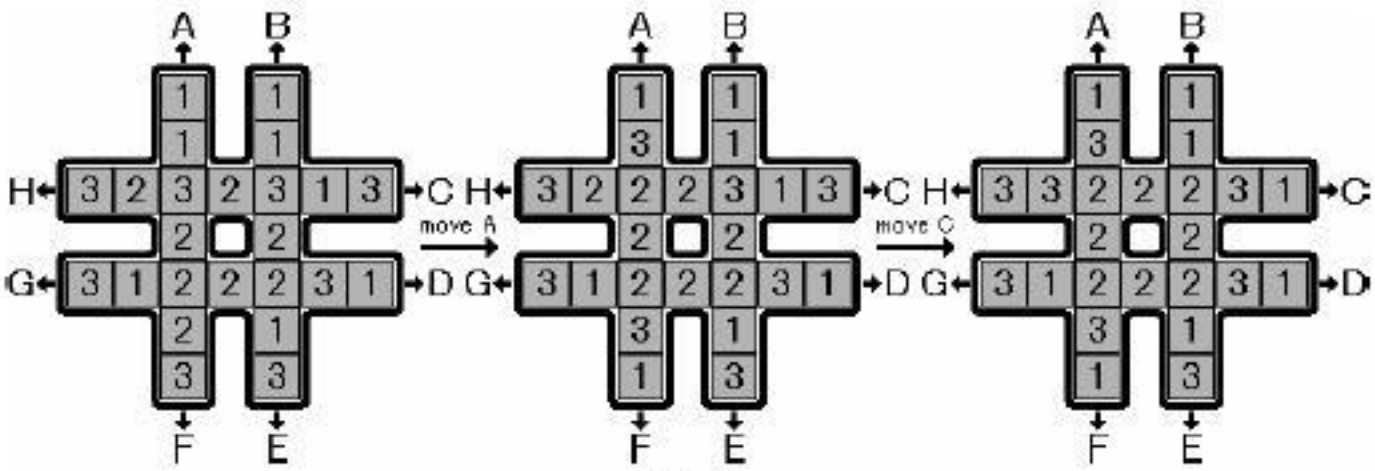


Fig.1

例：旋转游戏(POJ2286)

- 深搜还是广搜？
  - 各自的优劣
- 深搜的策略：
  - 迭代加深
- 广搜的策略：
  - 状态表示

启发式搜索

- 所谓启发式搜索，就在于当前搜索结点往下选择下一步结点时，可以通过一个启发函数来进行选择，选择代价最少的结点作为下一步搜索结点而跳转其上（遇到有一个以上代价最少的结点，不妨选距离当前搜索点最近一次展开的搜索点进行下一步搜索）。一个经过仔细设计的启发函数，往往在很快的时间内就可得到一个搜索问题的最优解，对于NP问题，亦可在多项式时间内得到一个较优解。

启发式估价函数

- f(n) = g(n) + h(n)
- 其中f(n) 是节点n的估价函数，g(n)实在状态空间中从初始节点到n节点的实际代价，h(n)是从n到目标节点最佳路径的估计代价。在这里主要是h(n)体现了搜索的启发信息，因为g(n)是已知的。

若干启发式搜索算法

- 局部择优搜索法
  - 在搜索的过程中选取“最佳节点”后舍弃其他的兄弟节点、父亲节点，继而继续搜索
- 最好优先搜索法
  - 在每一步的估价中都把当前的节点和以前的节点的估价值比较得到一个“最佳的节点”，继而进行搜索
- A*算法
  - 如果一个估价函数可以找出最短的路径，我们称之为可采纳性。
  - A*算法是一个可采纳的最好优先算法。

A*算法

- 评估函数：f*(n)=g*(n)+h*(n)
  - 其中
    - g*(n)为起始节点到节点n的最短路径代价
    - h*(n)为n到目标节点的最短路径代价
  - 则f*(n)是起始节点通过节点n到达目标节点的代价。
- 估价函数：f(n)=g(n)+h(n)
  - g(n)是g*(n)的估计，一般有g(n)>=g*(n)，why?
  - h(n)是h*(n)的估计

A*算法的条件

- 一种具有f(n)=g(n)+h(n)策略的启发式算法能成为A*算法的充分条件是:
  - 1）搜索树上存在着从起始点到终了点的最优路径。
  - 2）问题域是有限的。
  - 3）所有结点的子结点的搜索代价值>0。
  - 4）h(n)<=h*(n) 。
- 当此四个条件都满足时，一个具有f(n)=g(n)+h(n)策略的最好优先启发式算法能成为A*算法，并一定能找到最优解。

A*算法实现框架

- 初始化起始点，将其加入未搜索过点的优先队列
- 当队列非空且未达到终了点时

  获取队列头结点，并寻找其所有孩子结点；若孩子结点未在已搜索和待搜索队列中，计算其F值，并将其有序插入队列中；若在待搜索队列中且F值较小，则替换之。
- 输出搜索结果

  从已搜索队列中可逆推搜索过程

A*的证明

IDA*: 迭代加深的A*算法

- IDA*算法如下:

```
Procedure  IDA*
Begin
   初始化当前的深度限制 c = 1;
   把初始节点压入栈; 并假定 c'= ∝;
   While 栈不空do
      Begin
        弹出栈顶元素n
              If  n= goal, then结束, 返回n以及从初始节点到n的路径。
              Else  do
          Begin
              For n 的每个子节点n'
                  If   f ( n' ) ≤ c, then 把n'压入栈
                  Else c'=min( c', f ( n' ) )
              End for
            End
   End While
   If 栈为空并且c'= ∝, then 停止并退出;
   If 栈为空并且c'≠ ∝, then c = c', 并返回2。
End
```

IDA*与A*

- IDA*与A*算法相比，主要的优点是对于内存的需求
  - A*算法需要指数级数量的存储空间，因为没有深度方面的限制
  - 而IDA*算法只有当节点n的所有子节点n’的f(n’)小于限制值c时才扩展它，这样就可以节省大量的内存。
- 另外，IDA*不需要对扩展出的节点按启发值排序。

搜索的优化

- 状态空间的优化
- 减少搜索过程中的计算量
- 搜索顺序的选择
- 记忆化搜索
- 最优化剪枝：如果当前部分搜索方案比目前求得的最优方案还要差，则剪枝。
- 可行性剪枝：如果可以判断继续按当前方向搜索下去，必定无解或者达不到最优解，则剪枝。

- 最重要的优化：随机应变！！！

例 聪明的打字员（POJ1184）

- 阿兰是某机密部门的打字员，她现在接到一个任务：需要在一天之内输入几百个长度固定为6的密码。当然，她希望输入的过程中敲击键盘的总次数越少越好。
- 不幸的是，出于保密的需要，该部门用于输入密码的键盘是特殊设计的，键盘上没有数字键，而只有以下六个键：Swap0, Swap1, Up, Down, Left, Right，为了说明这6个键的作用，我们先定义录入区的6个位置的编号，从左至右依次为1，2，3，4，5，6。

例 聪明的打字员

- **Swap0：**按Swap0，光标位置不变，将光标所在位置的数字与录入区的1号位置的数字（左起第一个数字）交换。如果光标已经处在录入区的1号位置，则按Swap0键之后，录入区的数字不变
- **Swap1：**按Swap1，光标位置不变，将光标所在位置的数字与录入区的6号位置的数字（左起第六个数字）交换。如果光标已经处在录入区的6号位置，则按Swap1键之后，录入区的数字不变
- **Up：**按Up，光标位置不变，将光标所在位置的数字加1（除非该数字是9）。例如，如果光标所在位置的数字为2，按Up之后，该处的数字变为3；如果该处数字为9，则按Up之后，数字不变
- **Down：**按Down，光标位置不变，将光标所在位置的数字减1（除非该数字是0），如果该处数字为0，则按Down之后，数字不变，光标位置也不变
- **Left：**按Left，光标左移一个位置，如果光标已经在录入区的1号位置（左起第一个位置）上，则光标不动
- **Right：**按Right，光标右移一个位置，如果光标已经在录入区的6号位置（左起第六个位置）上，则光标不动。

例 聪明的打字员

- 当然，为了使这样的键盘发挥作用，每次录入密码之前，录入区总会随机出现一个长度为6的初始密码，而且光标固定出现在1号位置上。当巧妙地使用上述六个特殊键之后，可以得到目标密码，这时光标允许停在任何一个位置。
- 现在，阿兰需要你的帮助，编写一个程序，求出录入一个密码需要的最少的击键次数。

例：聪明的打字员

- 考虑到密码总数有10^6个，再加上光标位置，总的状态数目有$6*10^6$个。如此庞大的状态数是解题的瓶颈。
  - 把移动光标的操作Left ，Right，Swap0，Swap1和改变数值的操作Up，Down分离开
  - 先只进行移动光标的操作，记录得到的数的排列以及光标曾经到过的位置，状态数目$6!*2^6=46080$
  - 对上一步得到的每一个状态判断通过改变光标经过的位置上的数字能否得到目标状态。

例：聪明的打字员

- 例：
- 如果通过k次可达到排列356124(其中红色表示光标达到过的位置）
- 最后的密码如果是371194，则通过这种方案需要一共操作k+(7-5)+(6-1)+(9-2)=14+k步

- 最后通过所有排列方案求得的步数中取一个最小值，即为答案。

例：SHIPS的优化

- 将旋转、覆盖的判断预处理。

例：Knight’s Problem

- 有一个骑士，在一个无穷大的棋盘上。骑士一开始位于坐标（fx，fy），目的地坐标是（tx，ty）。骑士拥有m种行走规则。一个行走规则用（mx，my）表示，说明骑士如果当前处于坐标（x，y），那么使用这种行走规则后下一步会到达坐标（x+mx，y+my）。问骑士从起点到目标点最少需要多少步？
- 数据范围：
- -10<=mx,my<=10
- -5000<=fx,fy,tx,ty<=5000

例：Knight’s Problem

- 经典广搜
  - 以起点坐标为起始点，使用广度优先搜索。
  - 每次枚举行走规则，并将新走到的点加入到广搜队列中。
  - 使用hash表来判重，对于已经搜到的点不再重复搜索。
  - 一旦搜索到目标节点，那么当前所需步数就是最少步数，退出搜索，输出。

- 经典广搜的限制。

例：Knight’s Problem

- 猜想1：
  - 总有一种最优方案，使得骑士整个行走路径在以起点和终点为对角的矩形内或者离该矩形边界不超过一次行走的最大距离（对于本题来说，一次行走的最大距离在x方向和y方向上都不超过10）。

- 复杂度O(5010×2×5010×2)

例：Knight’s Problem

- 猜想2：
  - 存在一条最优路径，该路径离上的任何一点，距离起点到终点所连线段的距离都不超过3K（K为一次行走的最大距离）

- 复杂度约O（5000×10×10）

例： POJ1011 木棒问题

- 问题描述：

乔治拿来一组等长的棍子，将它们随机地裁断（截断后的小段称为木棒），使得每一节木棒的长度都不超过50个长度单位。然后他又想把这些木棒恢复到为裁截前的状态，但忘记了棍子的初始长度。请你设计一个程序，帮助乔治计算棍子的可能最小长度。每一节木棒的长度都用大于零的整数表示

输入数据

由多个案例组成，每个案例包括两行。第一行是一个不超过64的整数，表示裁截之后共有多少节木棒。第二行是经过裁截后，所得到的各节木棒的长度。在最后一个案例之后，是零。

输出要求

为每个案例，分别输出木棒的可能最小长度，每个案例占一行。

输入样例

9

5 2 1 5 2 1 5 2 1

4

1 2 3 4

0

输出样例

6

5

解题思路

- 初始状态：有N节木棒
- 最终状态：这N节木棒恰好被拼接成若干根等长的棍子(裁前的东西称为棍子）
- 枚举什么？

  枚举所有有可能的棍子长度。从最长的那根木棒的长度一直枚举到木棒长度总和的一半，对每个假设的棍子长度试试看能否拼齐所有棍子

- 在拼接过程中，要给用过的木棒做上标记，以免重复使用

- 拼好前i根棍子，结果发现第i+1根拼不成了，那么就要推翻第i根的拼法，重拼第i根.....直至有可能推翻第1根棍子的拼法

- 搜索题，首先要解决一个问题：按什么顺序搜索？

- 把木棒按长度排序。每次选木棒的时候都尽量先选长的。为什么？

因为短木棒比较容易用来填补空缺。一根长木棒，当然比总和相同的几根短木棒要优先使用

- 搜索题，还要解决一个问题：如何剪枝（就本题而言，即尽可能快地发现一根拼好的棍子需要被拆掉，以及尽量少做结果不能成功的尝试。

- 首先要看出搜索空间里有哪些状态，状态之间如何转换

还原过程

- 假设木棒的原始长度为L，按照长度L进行还原。经过N步恰好还原出f(L)根棍子。
  - 每一步从剩余木棒中取一根进行拼接
- 每一步的结果：状态(A B C D)，其中B+C恒等于N, D≤L
  - 已还原了A根棍子
  - 已经使用了B根木棒
  - 还剩余C根木棒
  - 第A+1根棍子还缺少的长度为D
- 从状态(A B C D)出发，可演化出C个状态
  - 初始状态(0　0　N　L)
  - 目标状态(f(L) N　0　　0)
  - 终态(A B C D), D<0 是终态是不能再往下走的

  此终止态也可以说是D小于所有剩余木棒长度的状态再往下走一步得到的。
- 任给一个初始状态(0 0 N L)，判断是否可以演化到目标状态(f(L) N　0　　0)
- {(f(L) N　0　　0)}中找到L的最小值

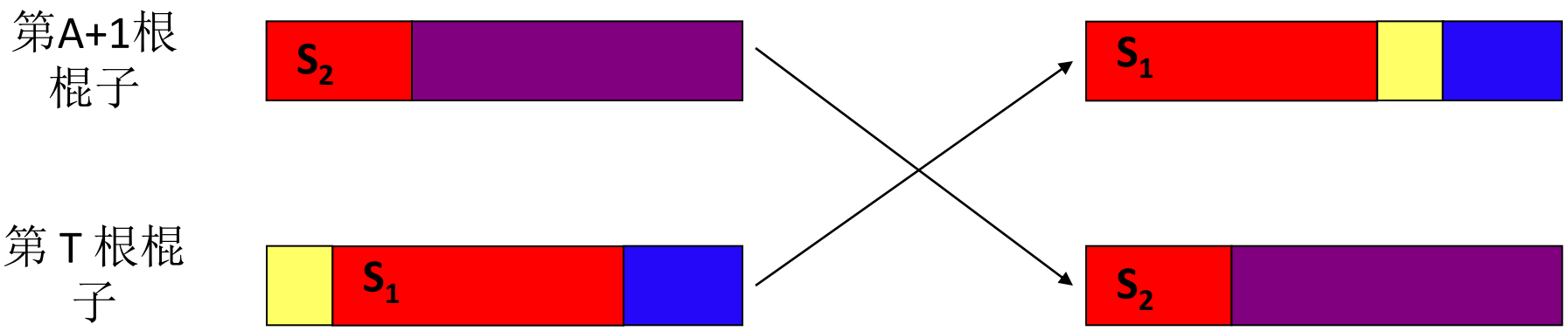
状态演化规则

- 令: minL表示输入木棒的最大长度,maxL是输入木棒的长度之和
  - 初始状态(0　0　N　L)中, L∈[minL maxL]且L可整除 maxL
- 令: (A B C D)上当前状态, S是当前被拼接的木棒, (A B C D)⊕S表示下一个状态
- 令x是S的长度, 则(A B C D)⊕S有四种可能
  - D=x且C=1:目标状态(A+1　N　0　0)
  - D<x: 终态(A　B+1 C-1 D-x)
  - D>x且C>1: (A　　B+1 C-1 D-x)
  - D=x且C>1: (A+1 B+1 C-1 L), 拼接出一根完整的棍子

提高搜索效率

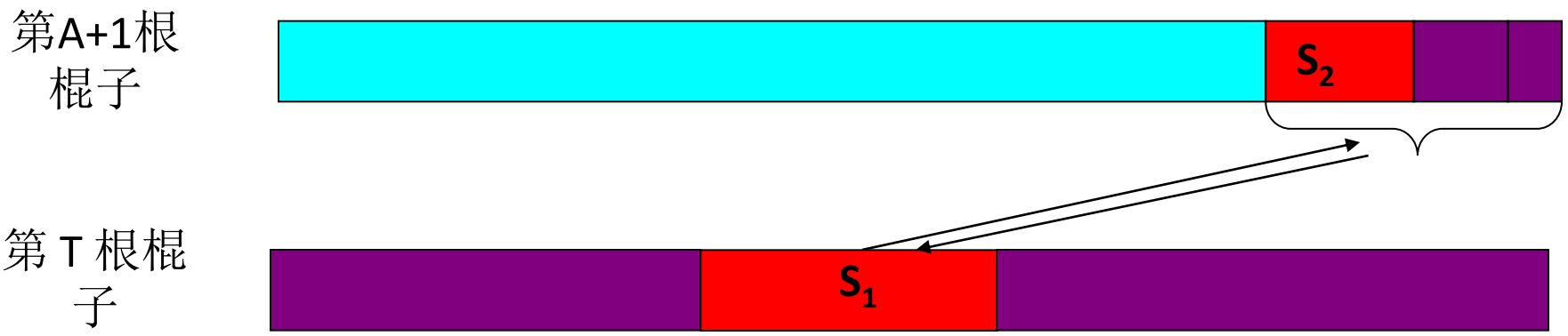
- S_1和S_2分别是状态(A B C D)时未被拼接的木棒
  - X_1 表示S_1的长度
  - X_2 表示S_2的长度
  - (A_1 B+1 C-1 D_1) 表示(A B C D)⊕S_1
  - (A_2 B+1 C-1 D_2) 表示(A B C D)⊕S_2
- 若X_1=X_2, 则(A_1 B+1 C-1 D_1)与(A_2 B+1 C-1 D_2)相同
  - 将木棒按照长度分类, 有利于状态**判重**
- 若(A_1 B+1 C-1 D_1)不能演化出目标状态，此时:
  - 若D=L，则 (A_2 B+1 C-1 D_2)也不能演化出目标状态, 因此, (A B C D)不能演化出目标状态, **需要剪枝**
  - 若D=X_1: 则(A_2 B+1 C-1 D_2)不能演化出目标状态,因此, (A B C D)不能演化出目标状态, **需要剪枝**

D=L时(A_2 B+1 C-1 D_2)不能演化出目标状态



- A= A_2
- 假设(A B+1 C-1 D_2)可演化出目标状态.
  - $S_2 \oplus S_{2,1} \oplus \ldots \oplus S_{2,X}$被还原成第A+1根棍子
  - $S_{1,1} \oplus \ldots \oplus S_{1,K-1} \oplus S_1 \oplus S_{1,K} \oplus \ldots \oplus S_{1,Y}$被还原成第T根棍子
- 则, (A B C D)到目标状态的演化路径可作如下变换,
  - $S_1 \oplus S_{1,1} \oplus \ldots \oplus S_{1,Y}$可还原第A+1根棍子
  - $S_2 \oplus S_{2,1} \oplus \ldots \oplus S_{2,X}$可还原第T根棍子
- 即(A B C L)$\oplus S_1$能演化出目标状态

$D+X_1=L$时$(A_2\ B+1\ C-1\ D_2)$不能演化出目标状态



- 假设$(A_2\ B+1\ C-1\ D_2)$可演化出目标状态.
  - $S_2 \oplus S_{2,1} \oplus \ldots \oplus S_{2,X}$的长度为$X_1$
  - $S_{1,1} \oplus \ldots \oplus S_{1,K-1} \oplus S_1 \oplus S_{1,K} \oplus \ldots \oplus S_{1,Y}$被还原成第T根棍子
- 则, (A B C D)到目标状态的演化路径可作如下变换,
  - (A B C D)$\oplus S_1$ 可还原第A+1根棍子
  - $S_{1,1} \oplus \ldots \oplus S_{1,K-1} \oplus S_2 \oplus S_{2,1} \oplus \ldots \oplus S_{2,X} \oplus S_{1,K} \oplus \ldots \oplus S_{1,Y}$可还原第T根棍子
- 即(A B C D)$\oplus S_1$能演化出目标状态

规划拼接顺序

- S_1、S_2和S_3分别是状态(A B C D)时未被拼接的木棒
  - X_1 表示S_1的长度
  - X_2 表示S_2的长度
  - X_3表示S_3的长度
  - (A_1 B+1 C-1 D_1) 表示(A B C D)$\oplus S_1$
- 若$X_1 = X_2 + X_3$
  - 当(A B C D)$\oplus S_2 \oplus S_3$可演化出目标状态时，(A B C D)$\oplus S_1$也可演化出目标状态
  - 当(A B C D)$\oplus S_1$能演化出目标状态时，(A B C D)$\oplus S_2 \oplus S_3$可能演化不出目标状态
- 在状态(A B C D)下，从未拼接木棒中，优先选择长度不超过D的长木棒，有利于**减少所演化的状态总数**。

剪枝1：

每次开始拼第i根棍子的时候，必定选剩下的木棒里最长的一根，作为该棍子的第一根木棒。对此决不后悔。

即:

就算由于以后的拼接失败，需要重新调整第i根棍子的拚法，也不会考虑替换第i根棍子中的第一根木棒（换了也没用）。如果在此情况下怎么都无法成功，那么就要推翻第i-1根棍子的拚法。如果不存在第i-1根棍子，那么就推翻本次假设的棍子长度，尝试下一个长度

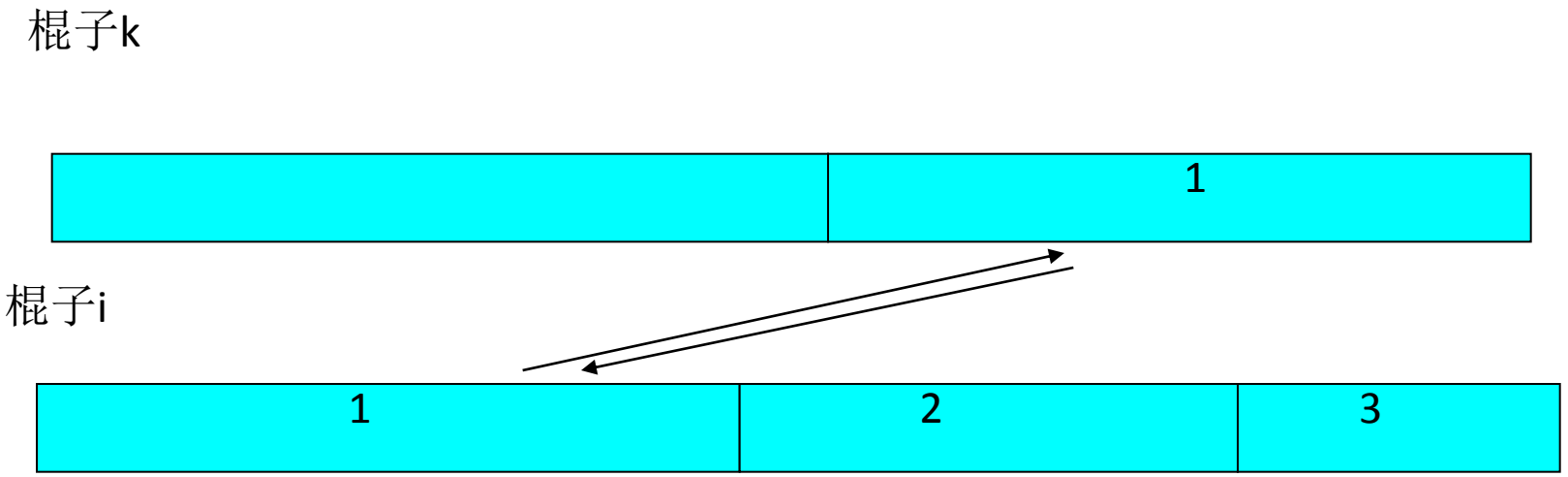
棍子I 如下拼法导致最后不能成功

可以考虑把2,3换掉重拼棍子i,但是把2，3都去掉后，换1是没有意义的

剪枝1：

为什么替换第i根棍子的第一根木棒是没用的？

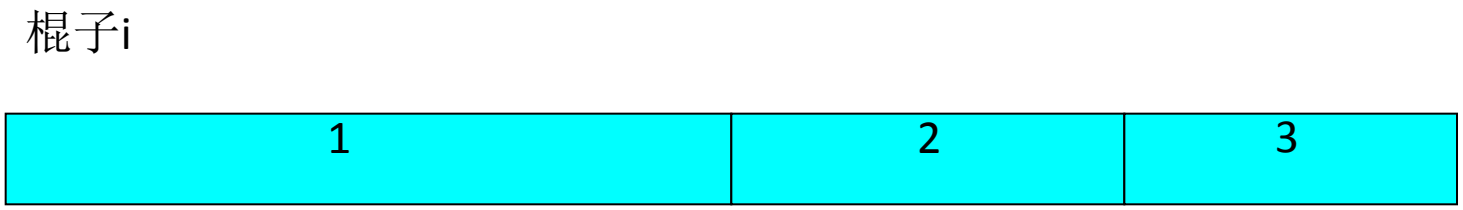
因为假设替换后能全部拼成功，那么这被换下来的第一根木棒，必然会出现在以后拼好的某根棍子k中。那么我们原先拼第i根棍子时,就可以用和棍子k同样的构成法来拼，照这种构成法拼好第i根棍子，继续下去最终也应该能够全部拼成功。



剪枝2：

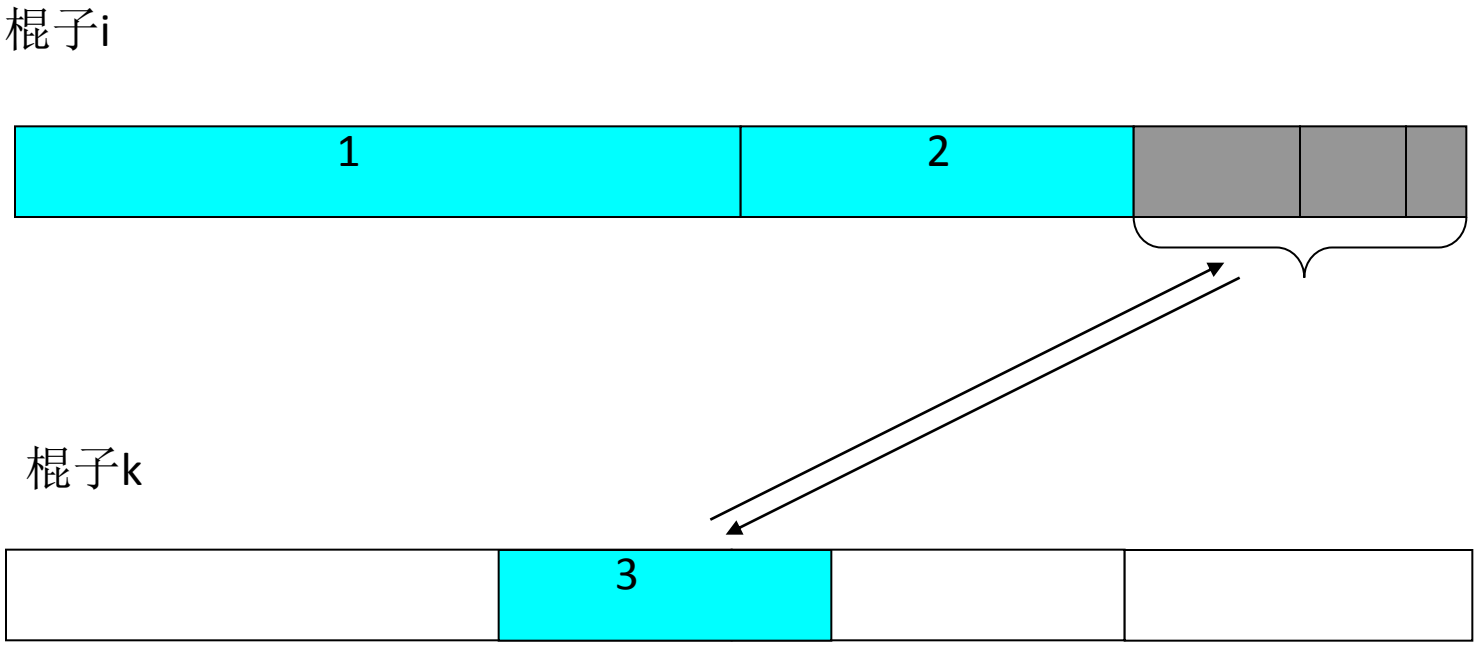
不要希望通过仅仅替换已拼好棍子的最后一根木棒就能够改变失败的局面。

假设由于后续拼接无法成功，导致准备拆除的某根棍子如下：



将 3 拆掉，留下的空用其他短木棒来填，是徒劳的

剪枝2：



假设替换3后最终能够成功，那么3必然出现在后面的某个棍子k里。将棍子k中的3和棍子i中用来替换3的几根木棒对调，结果当然一样是成功的。这就和i原来的拚法会导致不成功矛盾

剪枝3：

长度为L的棍子

已拼接部分　　未拼接部分

长度为L_1的未拼接木棒，共有N_1根

长度为L_2的未拼接木棒，共有N_2根

长度为L_3的未拼接木棒，共有N_3根

- 如果某次拼接选择长度为L_1的木棒，导致最终失败，则在同一位置尝试下一根木棒时，要跳过所有长度为L_1的木棒。

bool Dfs(int nUnusedSticks, int nLeft) ;

表示: 当前有nUnusedSticks根未用木棒，而且当前正在拼的那根棍子比假定的棍子长度短了nLeft, 那么在这种情况下能全部否拼成功。

Dfs的基本递推关系：

```
bool  Dfs(int nUnusedSticks, int nLeft) {
    …..
    找一根长度不超过nLeft的木棒(假设长为len，拼在
    当前棍子上，然后
    return Dfs(nUnusedSticks – 1,nLeft – len )       ;
}
```

Dfs的终止条件之一：

```
bool  Dfs(int nUnusedSticks,
    int nLeft ) {

        if( nUnusedSticks == 0 &&  nLeft == 0)
                return true;
}
```

```
#include <iostream>
#include <memory.h>
#include <stdlib.h>
#include <vector>
#include <algorithm>
using namespace std;
int T, S;
int L;
vector<int> anLength;
int anUsed[65];//是否用过的标记
int i,j,k;
int Dfs(int nUnusedSticks, int nLeft);
```

```
int main()
{
        while(1) {
                cin >> S;
                if( S == 0 )
                        break;
                int nTotalLen = 0;
                anLength.clear();
                for( int i = 0; i < S; i ++ ) {
                        int n;
                        cin >> n;
                        anLength.push_back(n);
                        nTotalLen += anLength[i];
                }
         sort(anLength.begin(),anLength.end(),
                greater<int>());
```

```
            for( L = anLength[0]; L <= nTotalLen / 2; L ++ ) {
                        if( nTotalLen % L)
                                    continue;
                        memset( anUsed, 0,sizeof(anUsed));
                        if( Dfs( S,L)) {
                                    cout << L << endl;
                                    break;
                        }
            }
           if( L > nTotalLen / 2 )
                        cout << nTotalLen << endl;
    } // while
    return 0;
}
```

```
int Dfs( int nUnusedSticks, int nLeft)
// nLeft表示当前正在拼的棍子和 L 比还缺的长度
{
        if( nUnusedSticks == 0 && nLeft == 0 )
                return true;
        if( nLeft == 0 ) //一根刚刚拼完
                nLeft = L;  //开始拼新的一根
        for( int i = 0;i < S;i ++) {
                if( !anUsed[i] && anLength[i] <= nLeft) {
                        if( i > 0 ) {
                                if( anUsed[i-1] == false
                                   && anLength[i] == anLength[i-1])
                                        continue; //剪枝3
                        }
                        anUsed[i] = 1;
```

上一根同样长度的木棒为什么还没用？必然是因为刚刚用过，并且发现用了后不行，才将其 anUsed标志置回0

```
                        if ( Dfs( nUnusedSticks - 1,
                                 nLeft - anLength[i]))
                                return true;
                        else {
                                anUsed[i] = 0;//说明本次不能用第i根
                                              //第i根以后还有用

                                if( anLength[i] == nLeft || nLeft == L)
                                        return false; //剪枝2、1
                        }
                }
        }
        return false;
}
```

剪枝 4：

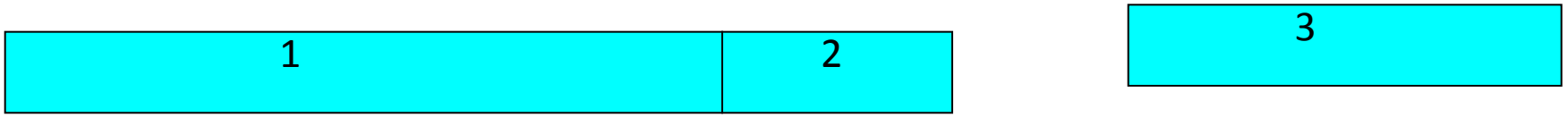
拼每一根棍子的时候，应该确保已经拼好的部分，长度是从长到短排列的，即拼的过程中要排除类似下面这种情况：

未完成的棍子i

木棒3 比木棒2长，这种情况的出现是一种浪费。因为要是这样往下能成功，那么2, 3 对调的拚法肯定也能成功。由于取木棒是从长到短的，所以能走到这一步，就意味着当初将3放在2的位置时，是不成功的

剪枝 4：

排除办法：每次找一根木棒的时候，只要这不是一根棍子的第一条木棒，就不应该从下标为0的木棒开始找，而应该从刚刚(最近）接上去的那条木棒的下一条开始找。这样，就不会往2后面接更长的3了



为此，要设置一个全局变量 nLastStickNo ，记住最近拼上去的那条木棒的下标。

```
int Dfs( int nUnusedSticks, int nLeft)
// nLeft表示当前正在拼的棍子和 L 比还缺的长度
{
        if( nUnusedSticks == 0 && nLeft == 0 )
                return true;
        if( nLeft == 0 ) //一根刚刚拼完
                nLeft = L;  //开始拼新的一根
        int nStartNo = 0;
        if( nLeft != L ) //剪枝4
                nStartNo = nLastStickNo + 1;
        for( int i = nStartNo;i < S;i ++) {
                if( !anUsed[i] && anLength[i] <= nLeft) {
                        if( i > 0 ) {
                                if( anUsed[i-1] == false
                                   && anLength[i] == anLength[i-1])
                                        continue; //剪枝3
                        }
                        anUsed[i] = 1;  nLastStickNo = i;
```

```
                        if ( Dfs( nUnusedSticks - 1,
                                 nLeft - anLength[i]))
                                return true;
                        else {
                                anUsed[i] = 0;//说明本次不能用第i根
                                              //第i根以后还有用

                                if( anLength[i] == nLeft || nLeft == L)
                                        return false;//剪枝2、1
                        }
                }
        }
        return false;
}
```

解决搜索的关键点

- 敢于下手！

例：摧毁车站（08年北京Regional）

- 给定你一个n个点，m条边的有向图，边的长度都是1。问你最少需要删掉多少个点（删掉一个点会将与其相连的边同时删除），才能使得1号点到n号点之间不存在长度小于等于k的路径。不允许删除1号或者n号点。
- 输入n，m，k和图的连边情况，n<=50,m<=4000,k<=1000

例：摧毁车站

- 这是搜索问题吗？
- 从1到n的最短路如果<k，则必须至少删掉一个点。
- 如何搜索
  - 每次枚举一条最短路，并且枚举删除的点。
  - 深度优先搜索。
- 这样搜索可行吗？
  - 当最短路太长时
  - 当最短路太多时

例：Air Conditioning Machinery (2008 World Finals)

- 用不超过六个如下Figure1的立方块连成管道（如Figure 2所示），使得某个立方体的入口和出口连通。（如Figure 3所示）
- 立方体边长不超过20

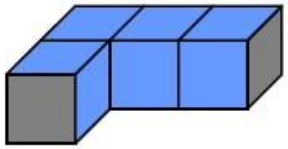


Figure 1

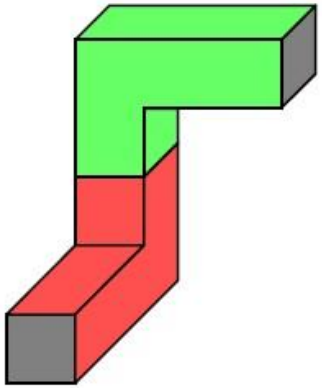


Figure 2

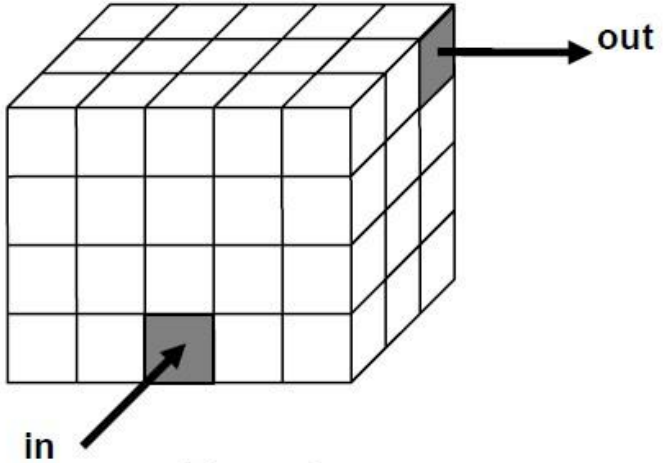


Figure 3

例:Air Conditioning Machinery

- 为什么可以搜索：
  - 只有至多6块，且所使用的是同一个立方块
  - 立方体边长不超过20
- 如何搜索
  - 维护状态：当前管道出口坐标和方向（x, y, z, dr)
  - 其中dr= +x, +y, +z, -x, -y, -z
  - 深度优先搜索
  - 立方块摆放方式预处理
  - 记忆化状态

例 植物大战僵尸-(2009宁波Reginal)

- 你有一些钱，可以用来购买僵尸。你的任务是在尽量过更多的关的前提下，得到的金钱越多越好。每一关你的任务就是购买僵尸来摧毁植物。植物生长在一块5*5的草坪上（如图）。你可以将购买到的僵尸立刻放在某一行上（也可以一次放多个），然后该僵尸会自动去摧毁该行的植物（蜘蛛除外）。每当你完全摧毁一行的植物后，你会得到p1元钱的奖金。你能成功地摧毁一行的条件是，你在该行放置的僵尸的总攻击力（攻击力之和）大于该行植物的总防御力（防御力之和），否则，你在该行放置的僵尸会全部死掉且不会摧毁该行任意一颗植物。
- 初始时有p3元钱

例 植物大战僵尸

植物名称	防御力	特殊作用
向日葵	0	摧毁之后可以得到p2元钱
豌豆	100	无
磁铁	0	使得放置位置的行以及相邻两行的足球运动员僵尸的攻击力150。
莴苣	0	放置位置的格子和周围8格自动抵档蜘蛛

例 植物大战僵尸

僵尸名称	攻击力	价钱	特殊作用
小僵尸	40	50	无
大僵尸	70	75	无
足球运动员	200	100	无
蜘蛛	0	125	能够摧毁任意一颗植物，如果那颗植物不是莴笋且不与莴笋相邻

例 植物大战僵尸

- 从大局上来看，我们可以采用贪心法来解决此题，关与关之间没有联系。
- 无论如何，不应在某行放置一堆总攻击力小于或等于该行植物的总攻击力的僵尸。
- 每次的行动可以分成下面两种。一、在某行放置僵尸，消灭该行所有植物。二、放置蜘蛛，消灭某个植物。
- 如果facer想在某行放置僵尸，那么他肯定是使用最少的金钱来达到消灭该行所有植物这个目的。

例 植物大战僵尸

- 对于要在一行怎样放置僵尸才能使用的钱数最少这个问题，可以通过预处理来完成。因为，影响僵尸摆放方案的只有2点：一是这一行的植物总攻击力，二是该行以及相邻行是否有磁铁。考虑到一行最多5个植物，因此植物总攻击力就只有0,100,200,300,400,500这六种可能，考虑有无磁铁的情况，一共也就12种可能。只需要在预处理的时候求出这12种情况的应对方案（最少需要花费的钱数），就不需要搜索的时候再来求了。

例 植物大战僵尸

- 蜘蛛的作用
  - 消灭莴苣：不允许
  - 消灭磁铁：能消灭几个磁铁？
  - 消灭向日葵：有意义吗？
    - 现在你手上只有125的钱了，消灭一个向日葵能拿10000的钱，然而125不够消灭任何一行——这种情况是有可能发生的。而这时，不得不用一个蜘蛛先拿到钱，才能继续下去。
  - 消灭豌豆：没有意义的。
    - 一个蜘蛛125，换成一个小僵尸+一个大僵尸，同样是125

例：植物大战僵尸

- 最优化剪枝
  - 即在搜索的过程中用全局变量记录到目前为止已经找到的最优解的钱数，每采取一步行动后都判断一下，如果此时已经花掉的钱数，加上消灭剩下的植物所需的最少钱数

搜索的应用：博弈问题

- 博弈是一类非常有意思的问题。
- 我们这里说的博弈是二人博弈，二人零和、全信息、非偶然的博弈。博弈双方的利益是完全对立的。
- 如井字棋、象棋、围棋等。

局面的胜负

- N局面——先手必胜局面
  - winning for the Next player
- P局面——后手必胜局面
  - winning for the Previous player

- 定义：
  - 每一个最终局面都是P局面
  - 对于一个局面，若至少有一种操作使它变成一个P局面，则它是一个N局面
  - 对于一个局面，无论如何操作都必然变成一个N局面，则它是一个P局面

估价函数

- 对于局面N，定义估价函数f(N)
- 当f(N)越大，代表当前局面先手获胜的可能性越大。

- 一般来说，f(N)=1, 0, -1
- f(N)=1 表示先手胜
- f(N)=0 表示平局
- f(N)=-1 表示先手败

估价函数

- 假设Alice和Bob进行博弈，Alice是先手。
- 若局势p对越Alice有利，则f(p)越大（+）；对Bob越有利，则f(p)越小（-）。

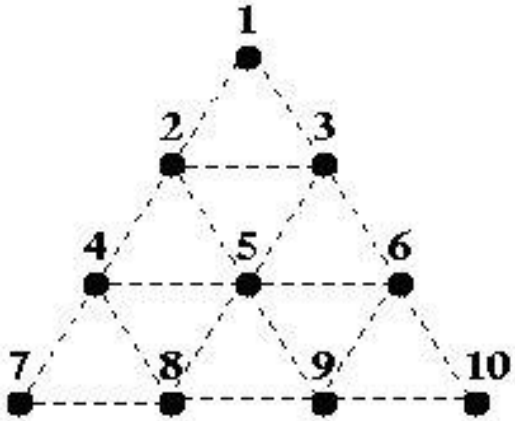
Alfa-Beta剪枝

- Alice每次选择估价函数最大的子节点进行扩展，Bob则选择估价函数最小的。

- 对于某个Alice先手的状态p，若它当前的最优值（最大值），大于等于它父状态（Bob先手状态）的当前值（最小值），则p不需要继续搜索。
- 对于某个Bob先手的状态p，若它当前的最优值（最小值），小于等于它父状态（Alice先手状态）的当前值（最大值），则p不需要继续搜索。

例：三角形大战(POJ1085)

- 两个游戏者轮流填充左边的虚线三角形。每次只能填充一条短边。
- 若某游戏者填充一条短边之后组成了一个小三角形，则该游戏者“拥有”这个三角形，并且可以继续填充。
- 当所有边都被填充之后，拥有三角形数目多的游戏者获胜。



例：三角形大战

- 博弈的思路
  - alfa-beta剪枝
  - 估价函数：A拥有的三角形数 – B拥有的三角形数
- 记忆化的思路
  - 记忆的状态: (p, own, edge)
    - p: 当前先手
    - own: 当前先手拥有的三角形数
    - edge: 当前边填充情况（二进制数）
  - 如何减少需要记忆的状态？

THE END

Thank you for your attention!