

# 搬运工问题的启示

重庆外语学校 刘汝佳

【关键字】搬运工问题，人工智能搜索，IDA\*

【摘要】本文讨论了一个有趣又富有挑战性的问题：搬运工问题。文章从状态空间搜索的基本知识开始讨论，根据搬运工问题的特点选择了 IDA\* 算法，并做了初步改进。本文的主要部分讨论了让程序智能化的几个方法 – 下界估计的改进，死锁判断，合适的任务分解与合并，模式搜索已经随机化实验，最后粗略的介绍了一些前面没有提到的想法，并做了总结。

## 目录

前言

正文

一 状态空间搜索基本知识

二 搬运工问题及其特点

三 用 IDA\* 算法解搬运工问题 – 实现与改进

四 如何使程序智能化

五 模拟人的预测能力 – 下界估计

六 模拟人的判断能力 – 死锁

七 模拟人的安排能力 – 任务分解与合并

八 模拟人的学习能力 – 模式搜索

九 给程序注入活力 – 随机化实验

十 另一些成功的和失败的想法

十一 总结

附录

A.游戏 (Xsokoban for Linux 和 SokoMind for Windows)

B.测试关卡 (标准 90 关, 儿童 61 关, 文曲星 170 关)

C.Rolling Stone 源程序

D.我的程序 S4 - Srbga's Super Sokoban Solver

E.论文配套幻灯片

F.参考资料

# 前言

“搬运工”是一个十分流行的单人智力游戏，玩家的任务是在一个仓库中操纵一个搬运工人将  $N$  个相同的箱子推到  $N$  个相同的目的地。不清楚规则不要紧，玩一玩附件里的 SokoMind 就知道了。我在里面加上了标准的测试关卡 90 关，幼儿关卡 61 关和文曲星 170 关，在以后的介绍中，我们就用这些关来测试我们的程序，因此我建议你自己先试一试，看看你能过多少关：)

因为本文内容不算少，在这里我先给大家提供一个小小的阅读建议。初学的朋友或者不知道 IDA\*算法的朋友一定要从第一章开始阅读并弄懂，因为它是全文的基础。第二章很好懂，大家只需要做一个了解，知道搬运工问题的难点在哪里，为什么我们选择了 IDA\*算法。急于看程序的朋友可以先看看第三章，我们的第一个版本 S4-Baby 诞生了，接着是一系列的改进措施，喜欢人工智能的朋友不妨认真看看，也许会找到灵感哦！最后是总结，和第一章一样的重要，不要错过了。

我假定本文的读者已经对相关知识有一定了解，所以一般不给出很严格的定义或者论证，只是粗略的提一下，语言尽量做到通俗易懂。但由于我的水平实在有限，错误之处一定不少，恳请大家批评指正：)

欢迎大家和我联系。Email:[liurujia@163.net](mailto:liurujia@163.net) , OICQ:2575127。

# 一 状态空间搜索基本知识

## 1. 状态空间(state space)

对于一个实际的问题，我们可以把它进行一定的抽象。通俗的说，状态(state)是对问题在某一时刻的进展情况的数学描述，状态转移(state-transition)就是问题从一种状态转移到另一种（或几种）状态的操作。如果只有一个智能体(Agent)可以实施这种状态转移，则我们的目的是单一的，也就是从确定的起始状态(start state)经过一系列状态转移而到达一个（或多个）目标状态(goal state)。

如果不止一个智能体可以操纵状态转移(例如下棋)，那么它们可能会朝不同的，甚至是对立的目标进行状态转移。这样的题目不在本文讨论范围之内。

我们知道，搜索的过程实际是在遍历一个隐式图，它的结点是所有的状态，有向边对应于状态转移。一个可行解就是一条从起始结点出发到目标状态集中任意一个结点的路径。这个图称为状态空间(state space)，这样的搜索就是状态空间搜索(Single-Agent Search)

## 2. 盲目搜索(Uninformed Search)

盲目搜索主要包括以下几种：

### 纯随机搜索(Random Generation and Random Walk)

听起来比较“傻”，但是当深度很大，可行解比较多，解的深度又不重要的时候还是有用的，而且改进后的随机搜索可以对付解分布比较有规律（相对密集或平均，或按黄金分割比例分布等）的题目。一个典型的例子是：你在慌乱中找东西的时候，往往都是进行随机搜索。

### 广度优先搜索(BFS)和深度优先搜索(DFS)

大家都很熟悉它们的时间效率，空间效率和特点了吧。广度优先搜索的例子是你的眼镜掉在地上以后，你趴在地板上找：)- 你总是先摸最接近你的地方，如果没有，在摸远一点的地方...深度优先搜索的典型例子是走迷宫。它们还有逆向和双向的搜索方式，但是不再本文讨论范围之内。

### 重复式搜索

这些搜索通过对搜索树扩展式做一些限制，用逐步放宽条件的方式进行重复搜索。这些方法包括：

### 重复式深度优先(Iterative Deepening)

限制搜索树的最大深度  $D_{max}$ ，然后进行搜索。如果没有解就加大  $D_{max}$  再搜索。虽然这样进行了很多重复工作，但是因为搜索的工作量与深度成指数关系，因此上一次（重复的）工作量比起当前的搜索量来是比较小的。这种方法适合搜

索树总的来说又宽又深，但是可行解却不是很深的题目（一般的深度优先可能陷入很深的又没有解的地方，广度优先的话空间又不够）

**重复式广度优先(Iterative Broadening)**

它限制的是从一个结点扩展出来的子节点的最大值  $B_{max}$ ,但是因为优点不是很明显，应用并不多，研究得也比较少。

**柱型搜索(Beam Search)**

它限制的是每层搜索树节点总数的最大值  $W_{max}$ 。显然这样搜索树大小与深度成正比，但是可能错过很接近起点的解，而增加  $W_{max}$  的时候保留哪些节点， $W_{max}$  增加多少是当前正在研究的问题。

**3.启发式搜索(Informed Search)**

我们觉得一些问题很有“想头”，主要是因为启发信息比较多，思考起来容易入手，但是却不容易找到解。我们不愿意手工一个一个盲目的试验，同样也不愿意我们的程序机械的搜索。也就是说，我们希望尽可能的挖掘题目自身的特点，让搜索智能化。下面介绍的启发式搜索就是这样的一种智能化搜索方法。

在刚才的那些算法中，我们没有利用状态本身的信息，只是利用了状态转移来进行搜索。事实上，我们自己在解决问题的时候常常会估计状态离目标到底有多接近，进而对多种方案进行选择。把这种方法用到搜索中来，我们可以用一个状态的估价函数来估计它到目标状态的距离。这个估价函数是和问题息息相关的，体现了一定的智能。为了以后叙述方便，我们先介绍一些记号：

|          |   |
|----------|---|
| S        | 问题的任何一种状态   |
| $H^*(s)$ | s 到目标的实际（最短）距离 – 可惜事先不知道:)  |
| H(s)     | s 的启发函数 – s 到目标距离的下界，也就是 $h(s) \leq h^*(s)$ ,如果 h 函数对任意状态 $s_1$ 和 $s_2$ ,还满足 $h(s_1) \leq h(s_2) + c(s_1, s_2)$ （其中 $c(s_1, s_2)$ 代表状态 $s_1$ 转移到 $s_2$ 的代价）,也就是状态转移时，下界 h 的减少值最多等于状态转移的实际代价，我们说 h 函数是相容 (consistent)的。(其实就是要求 h 不能减少得太快)              |
| G(s)     | 到达 s 状态之前的代价，一般就采用 s 在搜索树中的深度。  |
| F(s)     | s 的估价函数，也就是到达目标的总代价的估计。直观上，应该有 $f(s) = g(s) + h(s)$ ，即已经付出的和将要付出的代价之和。如果 g 是相容的，对于 $s_1$ 和它的后辈节点，有 $h(s_1) \leq h(s_2) + c(s_1, s_2)$ 两边同时加上 $g(s_1)$ ,有 $h(s_1) + g(s_1) \leq h(s_2) + g(s_1) + c(s_1, s_2)$ ,也就是 $f(s_1) \leq f(s_2)$ 。因此 f 函数单调递增。 |

表 1 启发式搜索用到的符号

**贪心搜索(Best-First Search)**

象广度优先搜索一样用一个队列储存待扩展，但是按照 h 函数值从小到大排

序(其实就是优先队列)。显然由于  $h$  估计的不精确性, 贪心搜索不能保证得到的第一个解最优, 而且可能很久都找不到一个解。

## A\*算法

和贪心搜索很类似, 不过是按照  $f$  函数值进行排序。但是这样会多出一个问题: 新生成的状态可能已经遇到过了的。为什么会这样呢? 由于贪心搜索是按照  $h$  函数值排序, 而  $h$  只与状态有关, 因此不会出现重复, 而  $f$  值不仅状态有关, 还与状态转移到  $s$  的方式有关, 因此可能出现同一个状态有不同的  $f$  值。解决方式也很简单, 如果新状态  $s_1$  与已经遇到的状态  $s_2$  相同, 保留  $f$  值比较小的一个就可以了。(如果  $s_2$  是待扩展结点, 是有可能出现  $f(s_2) > f(s_1)$  的情况的, 只有已扩展结点才保证  $f$  值递增)。A\*算法保证得到最优解, 但是所用的空间是很大的, 难以适应我们的搬运工问题。

## IDA\*算法

既然 A\*算法存在空间问题, 那么我们能不能借用深度优先搜索的空间优势, 用重复式搜索的方式来缓解危机呢? 经过研究, Korf 于 1985 年提出了一个 Iterative Deepening A\*(IDA\*)算法, 比较好的解决了这一问题。一开始, 我们把深度最大值  $D_{max}$  设为起始结点的  $h$  值, 开始进行深度优先搜索, 忽略所有  $f$  值大于  $D_{max}$  的结点, 减少了很多搜索量。如果没有解, 再加大  $D_{max}$  的值, 直到找到一个解。容易证明这个解一定是最优的。由于改成了深度优先的方式, 与 A\*比较起来, IDA\*更加实用:

1. 不需要判重, 不需要排序, 只用栈就可以了。操作简单。
2. 空间需求大大减少, 与搜索树大小成对数关系。

## 其他的启发式搜索

这些方法包括深度优先+最优剪枝式的 A\*, 双向 A\*, 但是由于很不成熟或者用处并不大, 这里就不介绍了。A\*算法有一个加权的形式, 由于在搬运工问题中效果不明显, 这里从略。

## 二 搬运工问题及其特点

在对状态空间搜索算法有一定了解之后，我们来看看我们的搬运工问题。究竟用什么方法比较好呢？让我们先来看看该问题的特点。

### 1. 搬运工问题

我们在前面已经介绍过搬运工问题，这里我只是想提一些和解题有关的注意事项。首先，我们考虑的搬运工问题的地图规模最大是 20\*20，这已经可以满足大部分关卡了。为了以后讨论方便，我们把地图加以编号。从左往右各列称为 A, B, C..., 而从上往下各行叫 a, b, c...。而由于不推箱子时的走路并不重要，我们在记录解的时候忽略了人的位置和移动，只记录箱子的移动。人的动作很容易根据箱子的动作推出来。下面是包含解答的标准关卡第一关。



*He-Ge, Hd-Hc-Hd, Fe-Ff, Fh-Gh-Hh-Ih-Jh-Kh-Lh-Mh-Nh-Oh-Ph-Qh-Rh-Rg,  
Ff-Fg-Fh-Gh-Hh-Ih-Jh-Kh-Lh-Mh-Nh-Oh-Ph-Qh-Qi-Ri,  
Fc-Fd-Fe-Ff-Fg-Fh-Gh-Hh-Ih-Jh-Kh-Lh-Mh-Nh-Oh-Ph-Qh-Qg,  
Ge-Fe-Ff-Fg-Fh-Gh-Hh-Ih-Jh-Kh-Lh-Mh-Nh-Oh-Ph-Qh-Rh,  
Hd-He-Ge-Fe-Ff-Fg-Fh-Gh-Hh-Ih-Jh-Kh-Lh-Mh-Nh-Oh-Ph-Pi-Qi,  
Ch-Dh-Eh-Fh-Gh-Hh-Ih-Jh-Kh-Lh-Mh-Nh-Oh-Ph-Qh*

呵呵，怎么样，第一关都要那么多步啊...以后的各关，可是越来越难。

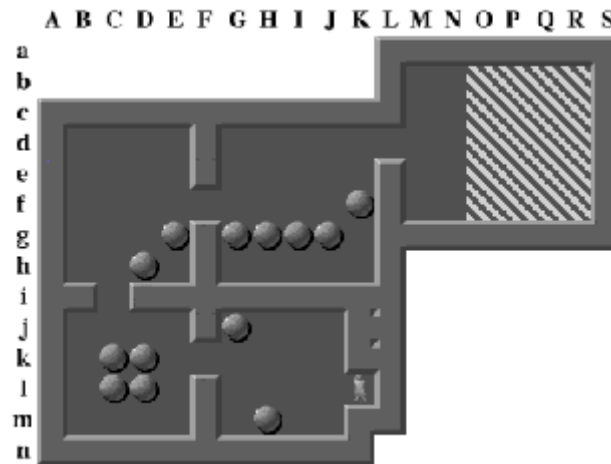
### 2. 搬运工问题的特点

我在前言里吹了这么半天，我想你即使以前没有玩，现在也已经玩过了吧：)。有什么感觉呢？是不是变化太多了，不好把握？不仅人不好把握，连编程也变得困难了很多。我们不妨拿它与经典的 8 数码问题作一个比较。

#### 1. 死锁！

初学者很快就会学到什么是死锁 – 一旦他（她）把一个箱子推到角上。显然，这样的布局再继续玩下去是没戏了，不管以后怎么推都不可能把这个箱子推离那个角。不少玩家都总结了不少死锁的经验，但是要比较系统的解决这个问题

并不是一件容易的事。我们将用整整一章（其实也不长啦）的篇幅来分析这个问题。



典型的死锁。想一想，为什么:) 我们再看一下 8 数码问题。它没有死锁，因为每一步都是可逆的。在这一点上，搬运工问题要令人头疼得多了。容易看出，这样的状态空间不是无向图，而是有向图。

## 2. 状态空间。

8 数码问题每次最多有 4 中移动方法，最多的步数也只有几十步。而搬运工问题呢？困难一点的关卡可以是一步有 100 多种选择，整个解答包括 600 多次推箱子动作。分支因子和解答树深度都这么大，状态空间自然就非同小可了。

## 3. 下界估计

在启发式搜索中，我们需要计算  $h$  值，也就是需要对下界进行估计。8 数码问题有很多不错的下界函数（如“离家”距离和），但是搬运工问题又怎么样呢？我们不能直接计算“离家”距离，因为谁的家是哪儿都不清楚。很自然，我们可以做一个二分图的最佳匹配，但是这个下界怎么样呢？

### a. 准确性

对于  $A^*$  及其变种来说，下界与实际代价越接近，一般来说算法效率就越高。我们这个最佳匹配只是“理想情况”，但是事实上，在很多情况下箱子相互制约，不得已离开目标路线来为其他箱子腾位置的事情是非常普遍的。例如我们的标准关卡第 50 关，有的箱子需要从目标格子穿过并离开它来为其它箱子让路。我们的下界函数返回值是 100，但是目前的最好结果是 370。多么大的差别！

### b. 效率

由于下界函数是一个调用非常频繁的函数，其效率不容忽视。最佳匹配的时间渐进复杂度大约是  $O(N^3)$ ，比 8 数码的下界函数不知大了多少...我们将会在后面给出一些改进方法，但是其本质不会改变。

## 3. 如何解决搬运工问题

已经有人证明了搬运工问题是 NP-Hard，看来我们还是考虑搜索吧。回想一



下上一节提到过的状态空间搜索，用哪一种比较好呢？

既然是智力游戏，可用的启发式信息是非常丰富了，我们不仅是要用，而且要用得尽量充分，所以应该用启发式搜索。而前面已经提到了，搬运工问题的状态空间是非常大的，A\*是没有办法了，因此我们选择了IDA\*算法：实现简单，空间需求也少。

既然搬运工问题这么难，为什么有那么多人解决了相当数量的关卡呢（标准的90N年以前就被人们模透了）。因为人聪明嘛。他们会预测，会安排，会学习，有直觉的帮助，还有一定的冒险精神。他们（也包括我啦，呵呵）常用的是些“高层次”的解题策略，既有效，又灵活。（Srbga:想学吗？Readers:当然想!!）可惜这些策略不是那么简单易学，也不是很有规律的。在后面的章节中，我将尽力模仿人的思维方式给我们的程序加入尽量多的智能。

# 三 用 IDA\*算法解搬运工问题

## 实现与改进

在上一节中，我们知道了 IDA\*算法是我们解决搬运工问题的核心算法。在这一节里，我们将用 IDA\*算法来做一个解决搬运工问题的程序 – 虽然是最初版本(我们称做 S4-Baby)，但是不要小看它哦！

### 1 IDA\*算法框架

由前所述，IDA\*算法是基于重复式深度优先的 A\*算法，忽略所有  $f$  值大于深度限制的结点。那么，我们不难写出 IDA\*算法框架的伪代码

伪代码 1 - IDA\*算法框架

```
procedure IDA_STAR(StartState)
begin
  PathLimit := H( StartState ) - 1;
  Success := False;
  repeat
    inc(PathLimit);
    StartState.g:= 0;
    Push(OpenStack,StartState);
    repeat
      CurrentState:=Pop(OpenStack);
      If Solution(CurrentState) then
        Success = True
      Else if PathLimit >= CurrentState.g + H(CurrentState) then
        Foreach Child(CurrentState) do
          Push(OpenStack, Child(CurrentState));
    until Success or empty(OpenStack);
  until Success or ResourceLimitsReached;
end;
```

这只是一个很粗略的框架，什么事情都不能做。不过我想大家可能比较急于试验一下 IDA\*的威力，因此我们不妨就做一个最基本的程序。

### 2. 第一个程序

要从框架做一个程序需要填充一些东西。在这里我们就展开一些讨论。

#### 输入输出文件格式

输入文件是一个文本文件，它由  $N$  行构成，每行是一些字符。各种字符的含义是：

|       |    |
|-------|----|
| SPACE | 空地 |
|-------|----|

|    |           |
|----|-----------|
| .  | 目标格子      |
| \$ | 箱子        |
| *  | 目标格子中的箱子  |
| @  | 搬运工       |
| +  | 目标格子中的搬运工 |
| #  | 墙         |

表 2 输入文件格式

这种格式和 Xsokoban, SokoMind 和 Rolling Stone 的格式是一致的，因此会比较方便一些。

输出文件第一行是推箱子的次数 M，以下 M 行，每行的格式是：x y direction，代表把第 x 行第 y 列的箱子往 direction 的方向推一步。Direction 可以是 left,right,up,down 之中的一个， $1 \leq x, y \leq 20$

## 数据结构

由于是最初的版本，我们不必考虑这么多：只需要可行，编程方便就可以了，暂时不管它的效率和其他东西。优化是以后的事。

我们定义新的数据类型 BitString, MazeType, MoveType, StateType 和 IDAType。请大家看附录中的程序，不难猜出它们的含义和用途。唯一需要说明的 BitString 类型。记录状态时，我们把地图看成一个的大数，一个格子是一个 bit。那么所有箱子构成一个 BitString，检查某一个是否有箱子（或者目标，墙）时只需要检测对应位置上的 bit 是否为 1。这样虽然会浪费一些空间，但是判断会比较快，操作也比较简单。

我们把 x,y 坐标合并成一个“position”变量。其中  $Position = (x-1) * width + (y-1)$ 。我们用常量数组 DeltaPos:array[0..3] 表示上，下，左，右的 Position 增量。

## 算法

为了简单起见，我们连最佳匹配也不做了，用所有箱子离最近目标的距离和作为下界函数。不过，这里的“距离”是指推的次数，计算的时候（MinPush 函数），只要忽略其它所有箱子，然后用一次 BFS 就可以了。

## 效果

嘿嘿，这个效果嘛，不说你也知道的，就是标准关一个也过不了啦。不过为了说明我的程序是正确的，你可以试验一下幼儿关卡(共 61 关)嘛！

什么！第一关都就没有动静了...55555，生成了 18 万个结点???不过很多关都很快就过了的。我们用 1,000,000 个结点为上限(在我的 Celeron 300A 上要运行十多分钟),得到以下的测试结果：

| No. | 步数 | 结点数    | No. | 步数 | 结点数   | No. | 步数  | 结点数   |
|-----|----|--------|-----|----|-------|-----|-----|-------|
| 1   | 15 | 186476 | 21  | 8  | 102   | 41  | 11  | 145   |
| 2   | 6  | 24     | 22  | 7  | 110   | 42  | 10  | 118   |
| 3   | 5  | 14     | 23  | 10 | 192   | 43  | 12  | 223   |
| 4   | 6  | 24     | 24  | 10 | 432   | 44  | 8   | 63    |
| 5   | 9  | 31     | 25  | 4  | 23    | 45  | 12  | 138   |
| 6   | 5  | 8      | 26  | 11 | 846   | 46  | 14  | 178   |
| 7   | 6  | 35     | 27  | 3  | 18    | 47  | 8   | 296   |
| 8   | 11 | 39     | 28  | 9  | 38    | 48  | 8   | 156   |
| 9   | 4  | 12     | 29  | 10 | 142   | 49  | 5   | 60    |
| 10  | 5  | 14     | 30  | 8  | 641   | 50  | 11  | 14451 |
| 11  | 5  | 13     | 31  | 7  | 192   | 51  | N/A | >1M   |
| 12  | 4  | 19     | 32  | 3  | 12    | 52  | N/A | >1M   |
| 13  | 4  | 14     | 33  | 11 | 51    | 53  | 8   | 470   |
| 14  | 6  | 20     | 34  | 11 | 332   | 54  | 16  | 24270 |
| 15  | 6  | 57     | 35  | 16 | 11118 | 55  | N/A | >1M   |
| 16  | 12 | 3947   | 36  | 10 | 242   | 56  | 14  | 3318  |
| 17  | 6  | 63     | 37  | 9  | 1171  | 57  | N/A | >1M   |
| 18  | 11 | 5108   | 38  | 11 | 556   | 58  | N/A | >1M   |
| 19  | 10 | 467    | 39  | 10 | 72    | 59  | 11  | 328   |
| 20  | 10 | 1681   | 40  | 9  | 203   | 60  | N/A | >1M   |
|     |    |        |     |    |       | 61  | N/A | >1M   |

没有解决的几关是：51,52,55,57,,58,60,61

比较困难的几关是 1,16,18,20,26,30,35,37,38,50,53,54,56

下面，我们来看看“困难关卡”的下界估计的情况，看看“偷懒”付出的代价。

| 关卡 | 最优步数 | 初始深度 | 结点总数   | 顶层结点数 |
|----|------|------|--------|-------|
| 1  | 15   | 11   | 186476 | 7416  |
| 16 | 12   | 7    | 3947   | 844   |
| 18 | 11   | 10   | 5108   | 49    |
| 20 | 10   | 6    | 1681   | 42    |
| 26 | 11   | 5    | 846    | 394   |
| 30 | 8    | 6    | 641    | 200   |
| 35 | 16   | 3    | 11118  | 3464  |
| 37 | 9    | 4    | 1171   | 493   |
| 38 | 11   | 5    | 556    | 250   |
| 50 | 11   | 6    | 14451  | 51    |

|    |    |   |       |      |
|----|----|---|-------|------|
| 53 | 8  | 5 | 470   | 48   |
| 54 | 16 | 9 | 24270 | 2562 |
| 56 | 14 | 4 | 3318  | 460  |

由此可见，下界估计对于搜索树的大小是很有关系的。看看第 18,20,35,50,54,56 关吧。顶层结点多么少！如果一开始就从这一层搜索不就...看来我们真的需要用最佳匹配算法了。

### 3. 试验最佳匹配算法的威力

好，下面我们来使用最佳匹配算法。最佳匹配算法可以用网络流来实现，但是这里我们采用修改顶标算法，我是抄的书上的程序（偷个懒嘛，呵呵）。现在，程序改叫 Baby2 了^\_^

下面是刚才的“难题”的测试情况。

| 关卡 | 实际步数 | 初始深度 | Baby-1 结点总数 | Baby-2 结点总数 |
|----|------|------|-------------|-------------|
| 1  | 15   | 15   | 186476      | 60          |
| 16 | 12   | 10   | 3947        | 304         |
| 18 | 11   | 11   | 5108        | 46          |
| 20 | 10   | 8    | 1681        | 76          |
| 26 | 11   | 5    | 846         | 552         |
| 30 | 8    | 8    | 641         | 153         |
| 35 | 16   | 4    | 11118       | 6504        |
| 37 | 9    | 5    | 1171        | 438         |
| 38 | 11   | 5    | 556         | 546         |
| 50 | 11   | 7    | 14451       | 98          |
| 53 | 8    | 8    | 470         | 37          |
| 54 | 16   | 12   | 24270       | 273         |
| 56 | 14   | 4    | 3318        | 2225        |

哇！有的比刚才的顶层结点还要少！当然了，下界估计好了，当前层的深度剪枝也更准确了嘛。

另外，现在我们来看看文曲星的前 12 关，第 1, 2, 4, 6, 8, 9, 11 关已经可以在 50000 个结点之内出解。

| 关卡 | 实际步数 | 初始深度 | 结点总数 | 顶层结点数 |
|----|------|------|------|-------|
| 1  | 31   | 31   | 75   | 75    |
| 2  | 11   | 11   | 142  | 142   |

|    |    |    |       |      |
|----|----|----|-------|------|
| 4  | 26 | 18 | 33923 | 159  |
| 6  | 16 | 16 | 47    | 47   |
| 8  | 27 | 21 | 239   | 213  |
| 9  | 12 | 6  | 4806  | 2778 |
| 11 | 14 | 14 | 73    | 73   |

那么下一步干什么呢？打印出每个状态来分析，我们不难发现大量重复结点。所以下一个问题自然就是：怎么避免重复呢？

#### 4. 试验 HASH 表的威力

判重嘛,当然就需要 HASH 表了。不过一个很棘手的问题是:如何表示状态?在结点扩展中,我们用比特流的方式定义了箱子的状态,但是在这里我们需要的是合适的数组的下标。这种表示法不爽吧。所以在构造 HASH 表的时候我们就用箱子的坐标来表示状态,也就是  $N$  元组  $(p[1], p[2], p[3] \dots p[n])$ 。至于散列函数嘛,我们根据 HASH 表的项数来考虑。这里,如果箱子最多 100 个,我们就用 10000 项试试看。一种方案是把所有的坐标加起来,但是这样做冲突很多!因为一个箱子 A 右移一格,另一个箱子 B 左移一格,散列函数值不变。考虑到必须使冲突变少,函数又不宜太复杂,我们采用坐标的加权和来作为散列函数值,也就是  $\text{Sum}\{k * \text{Position}[k]\}$ ,当然,最后要对 10000 取余数,其实这个函数也不好,不过我比较懒了,以后再改进吧。至于冲突处理吗,为了简单起见,我们用开链法 设立链表来保存所有元素。值得注意的是,箱子坐标相同而人的坐标不能互通的状态是不同的,应该一起保存。下面是刚才那些关的测试结果:

| 关卡     | 实际步数 | 初始深度 | Baby2 结点数 | Baby3 结点数 |
|--------|------|------|-----------|-----------|
| Kid 1  | 15   | 15   | 60        | 52        |
| Kid 16 | 12   | 10   | 304       | 189       |
| Kid 18 | 11   | 11   | 46        | 41        |
| Kid 20 | 10   | 8    | 76        | 67        |
| Kid 26 | 11   | 5    | 552       | 192       |
| Kid 30 | 8    | 8    | 153       | 145       |
| Kid 35 | 16   | 4    | 6504      | 704       |
| Kid 37 | 9    | 5    | 438       | 136       |
| Kid 38 | 11   | 5    | 546       | 152       |
| Kid 50 | 11   | 7    | 98        | 96        |



|        |    |    |       |       |
|--------|----|----|-------|-------|
| Kid 53 | 8  | 8  | 37    | 24    |
| Kid 54 | 16 | 12 | 273   | 258   |
| Kid 56 | 14 | 4  | 2225  | 1518  |
| Wqx 1  | 31 | 31 | 75    | 75    |
| Wqx 2  | 11 | 11 | 142   | 107   |
| Wqx 4  | 26 | 18 | 33923 | 33916 |
| Wqx 6  | 16 | 16 | 47    | 46    |
| Wqx 8  | 27 | 21 | 239   | 226   |
| Wqx 9  | 12 | 6  | 4806  | 968   |
| Wqx 11 | 14 | 14 | 73    | 67    |

新完成的关卡有：

| 关卡     | 实际步数 | 初始深度 | 结点总数  | 顶层结点数 |
|--------|------|------|-------|-------|
| Kid 51 | 13   | 13   | 629   | 629   |
| Kid 52 | 18   | 18   | 39841 | 39841 |
| Kid 55 | 14   | 4    | 4886  | 1919  |
| Kid 60 | 15   | 9    | 6916  | 916   |

需要注意的是，在保护模式下运行 kid57 的时候出现的 heap overflow，说明完全保存结点没有必要(费空间，费时间)，也不大可能。那么，我们应该怎样

做呢？我们知道，评价一个 HASH 表的优劣，一般是从两个方面：查表成功的频率和查表成功以后节省的工作。因此，我们可以设置两个 Hash 表，一个保存最近的结点（查到的可能性比较大）和深度大的结点（一旦找到，节省很多工作）。这样做不会增加多少结点数，但是是程序效率有所提高，求解能力（空间承受能力）也有较大改善。但是为了方便，我们的程序暂时只使用第一个表。

## 5. 结点扩展顺序的优化

在这一节中，我们的最后一个改进是优化结点扩展的顺序，不是想修剪搜索树，而是希望早一点得到解。具体的改进方法是这样的：

1. 优先推刚刚推过的箱子

2. 然后试所有的能够减少下界的方案，减少得越多越先试。如果减少得一样多，就先推离目标最近的。

3. 最后试其他的，也象 2 一样按顺序考虑。

可以预料，这样处理以后，“比较容易”先找到解，但是因为下界估计不准所花费的代价是无法减小的（也就是说只能减少顶层结点数）。不过作为 IDA\* 的标准改进方法之一，我们有必要把它加入我们的程序中试试。

（需要注意的是，我们使用的是栈，应该把比较差的方案先压栈）

实际测试结果，1 的效果比较好，2 和 3 的效果不佳，甚至产生了更多的结点。可能主要是我们的下界估计不准确，而 2 和 3 用到了下界函数的缘故。这一个版本 Baby-4 中，我们屏蔽了第 2，3 项措施。

好了，写了四个 Baby 版程序，想不想比较一下呢？不过我只对几个困难一点的数据感兴趣。

| 关卡     | 实际步数 | Baby-1   | Baby-2   | Baby-3 | Baby-4 |
|--------|------|----------|----------|--------|--------|
| Kid 1  | 11   | 186476   | 60       | 52     | 38     |
| Kid 16 | 7    | 3947     | 304      | 189    | 149    |
| Kid 18 | 10   | 5108     | 46       | 41     | 31     |
| Kid 35 | 16   | 11118    | 6504     | 704    | 462    |
| Kid 50 | 11   | 14451    | 98       | 96     | 152    |
| Kid 51 | 13   | Too many | Too many | 629    | 54     |
| Kid 52 | 18   | Too many | Too many | 39841  | 97     |
| Kid 54 | 16   | 24270    | 273      | 258    | 140    |
| Kid 55 | 14   | Too many | Too many | 4886   | 3390   |
| Kid 56 | 14   | 3318     | 2225     | 1518   | 1069   |
| Kid 60 | 15   | Too many | Too many | 6916   | 5022   |
| Wqx 4  | 26   | 97855    | 33923    | 33916  | 24251  |
| Wqx 9  | 12   | 116927   | 4806     | 968    | 350    |

从上表可以看出，我们的优化总的来说是有效的，而且直观的看，那些改进不明显的很多是因为下界估计比较差，这一点我们以后会继续讨论。不管怎样，这 61 关“幼儿关”过了 58 关倒是挺不错的，至少可以说明我们程序的 Baby

版已经具有普通儿童的“智力”了^\_^。不过这只是个开头，好戏还在后头！

## 6. Baby-4 源程序

程序 S4BABY4.PAS 在附件中，这里只是加了少量的注释。大家可以试试它的效果，但是没有必要看得太仔细，因为在以后的章节中，我会改动很多东西，甚至连 IDA\* 主程序框架都会变得不一样。

常量定义：

**const**

**{Version}**

**VerStr='S4 - SRbGa Super Sokoban Solver (Baby Version 4)';**

**Author='Written by Liu Rujia(SrbGa), 2001.2, Chongqing, China';**

**{Files}**

**InFile='soko.in';**

**OutFile='soko.out';**

**{Charactors}**

**Char\_Soko='@';**

**Char\_SokoInTarget='+';**

**Char\_Box='\$';**

**Char\_BoxInTarget='\*';**

**Char\_Target='.';**

**Char\_Wall='#';**

**Char\_Empty=' ';**

**{Dimentions}**

**Maxx=21;**

**Maxy=21;**

**MaxBox=50;**

**{Directions}**

**Up=0;**

**Down=1;**

**Left=2;**

**Right=3;**

**DirectionWords:array[0..3] of string=('UP','DOWN','LEFT','RIGHT');**

**{Movement}**

**MaxPosition:integer=Maxx\*Maxy;**

**Opposite:array[0..3] of integer=(1,0,3,2);**

**DeltaPos:array[0..3] of integer=(-Maxy,Maxy,-1,1);**

我们把 x,y 坐标合成一个值 position, 其中  $position=(x-1)*maxy+(y-1)$ 。这里用类型常量是因为以后会根据地图的尺寸改变 MaxPosition 的值。Opposite 就是相反方向例如 Opposite[UP]:=DOWN;DeltaPos 也是会重新设定的。我们在进行移动的时候只需要用: NewPos:=OldPos+DeltaPos[Direction]就可以了, 很方便。

**{IDA Related}**

**MaxNode=1000000;**

**MaxDepth=100;**

**MaxStack=150;**

**DispNode=1000;**

每生成多少个结点报告一次。

**{HashTable}**

**MaxHashEntry=10000;**

**HashMask=10000;**

**MaxSubEntry=100;**



**{BitString}**

**BitMask=array[0..7] of byte=(1,2,4,8,16,32,64,128);**

**Infinite=Maxint;**

**类型定义:**

**type**

**PositionType=integer;**

**BitString=array[0..Maxx\*Maxy div 8-1] of byte;**

整个地图就是一个 BitString。第 position 位为1当且仅当 position 位置有东西（如箱子，目标，墙）。

**MapType=array[1..Maxx] of string[Maxy];**

**BiGraph=array[1..MaxBox,1..MaxBox] of integer;**

**MazeType=**

**record**

**X,Y:integer;**

```

    Map:MapType;

    GoalPosition:array[1..MaxBox] of integer;

    BoxCount:integer;

    Goals:BitString;

    Walls:BitString;

end;

```

尺寸, 原始数据 (用来显示状态的), 目标的 BitString, 箱子总数, 目标位置 (BitString 和位置数组都用是为了加快速度) 和 Walls 的 BitString。

```

MoveType=

record

    Position:integer;

    Direction:0..3;

end;

```

Direction 是箱子被推向的方向。

```

StateType=

record

    Boxes:BitString;

```

**ManPosition:PositionType;**

**MoveCount:integer;**

**Move:array[1..MaxDepth] of MoveType;**

**g,h:integer;**

**end;**

**IDAType=**

**record**

**TopLevelNodeCount:longint;**

**NodeCount:longint;**

**StartState:StateType;**

**PathLimit:integer;**

**Top:integer;**

**Stack:array[1..MaxStack] of StateType;**

**end;**

**Top 是栈顶指针。**

**PHashTableEntry=^HashTableEntry;**

**HashTableEntry=**

**record**

**Next:PHashTableEntry;**

**State:StateType;**

**end;**

**PHashTableType=^HashTableType;**

**HashTableType=**

**record**

**FirstEntry:array[0..MaxHashEntry] of PHashTableEntry;**

**Count:array[0..MaxHashEntry] of byte;**

**end;**

这些是 Hash 表相关类型。我们采用的是拉链法，这样可以利用指针申请到堆空间，结合保护模式使用，效果更好。

**var**

**HashTable:PHashTableType;**

**SokoMaze:MazeType;**

**IDA:IDAType;**

```
procedure SetBit(var BS:BitString; p:integer);
```

```
begin
```

```
    BS[p div 8]:=BS[p div 8] or BitMask[p mod 8];
```

```
end;
```

```
procedure ClearBit(var BS:BitString; p:integer);
```

```
begin
```

```
    BS[p div 8]:=BS[p div 8] xor BitMask[p mod 8];
```

```
end;
```

```
function GetBit(var BS:BitString; p:integer):byte;
```

```
begin
```

```
    if BS[p div 8] and BitMask[p mod 8]>0 then GetBit:=1 else GetBit:=0;
```

```
end;
```

这些是位操作，设置，清除和得到一个 BitString 的某一项。

```
procedure Init;
```

```
var
```

```
    Lines:MapType;
```

```
procedure ReadInputFile;

var

    f:text;

    s:string;

begin

    SokoMaze.X:=0;

    SokoMaze.Y:=0;

    SokoMaze.BoxCount:=0;

    assign(f,infile);

    reset(f);

    while not eof(f) do

        begin

            readln(f,s);

            if length(s)>SokoMaze.Y then

                SokoMaze.Y:=length(s);

            inc(SokoMaze.X);

            Lines[SokoMaze.X]:=s;

        end;

    end;
```

**close(f);**

**end;**

**procedure AdjustData;**

**var**

**i,j:integer;**

**begin**

**for i:=1 to SokoMaze.X do**

**while length(Lines[i])<SokoMaze.Y do**

**Lines[i]:=Lines[i]+' ';**

**SokoMaze.Map:=Lines;**

**for i:=1 to SokoMaze.X do**

**for j:=1 to SokoMaze.Y do**

**if SokoMaze.Map[i,j] in [Char\_BoxInTarget,Char\_SokoInTarget,Char\_Tar**

**get] then**

**SokoMaze.Map[i,j]:=Char\_Target**

**else if SokoMaze.Map[i,j]<>Char\_Wall then**

**SokoMaze.Map[i,j]:=Char\_Empty;**

调整 Map 数组，把箱子和搬运工去掉。

```
for i:=1 to SokoMaze.X do
```

```
  for j:=1 to SokoMaze.Y do
```

```
    if Lines[i,j] in [Char_Target,Char_BoxInTarget,Char_SokoInTarget] then
```

```
      begin
```

```
        inc(SokoMaze.BoxCount);
```

```
        SokoMaze.GoalPosition[SokoMaze.BoxCount]:= (i-1)*SokoMaze.Y+j-1;
```

```
      end;
```

统计 Goal 的个数和 GoalPosition。

```
DeltaPos[Up]:= -SokoMaze.Y;
```

```
DeltaPos[Down]:= SokoMaze.Y;
```

```
MaxPosition:= SokoMaze.X*SokoMaze.Y;
```

根据地图尺寸调整 DeltaPos 和 MaxPosition

```
end;
```

```
procedure ConstructMaze;
```

```
var
```

```
  i,j:integer;
```



**begin**

**fillchar(SokoMaze.Goals,sizeof(SokoMaze.Goals),0);**

**fillchar(SokoMaze.Walls,sizeof(SokoMaze.Walls),0);**

**for i:=1 to SokoMaze.X do**

**for j:=1 to SokoMaze.Y do**

**case Lines[i,j] of**

**Char\_SokoInTarget, Char\_BoxInTarget, Char\_Target:**

**SetBit(SokoMaze.Goals,(i-1)\*SokoMaze.Y+j-1);**

**Char\_Wall:**

**SetBit(SokoMaze.Walls,(i-1)\*SokoMaze.Y+j-1);**

**end;**

**end;**

**procedure InitIDA;**

**var**

**i,j:integer;**

**StartState:StateType;**

**begin**

**IDA.NodeCount:=0;**

**IDA.TopLevelNodeCount:=0;**

**fillchar(StartState,sizeof(StartState),0);**

**for i:=1 to SokoMaze.X do**

**for j:=1 to SokoMaze.Y do**

**case Lines[i,j] of**

**Char\_Soko, Char\_SokoInTarget:**

**StartState.ManPosition:=(i-1)\*SokoMaze.Y+j-1;**

**Char\_Box, Char\_BoxInTarget:**

**SetBit(StartState.Boxes,(i-1)\*SokoMaze.Y+j-1);**

**end;**

**StartState.g:=0;**

**IDA.StartState:=StartState;**

**new(HashTable);**

**for i:=1 to MaxHashEntry do**

**begin**

**HashTable^.FirstEntry[i]:=nil;**

**HashTable^.Count[i]:=0;**

**end;**

**end;**

**begin**

**ReadInputFile;**

**AdjustData;**

**ConstructMaze;**

**InitIDA;**

**end;**

**procedure PrintState(State:StateType);**

**var**

**i,x,y:integer;**

**Map:MapType;**

**begin**

**Map:=SokoMaze.Map;**

**x:=State.ManPosition div SokoMaze.Y+1;**

**y:=State.ManPosition mod SokoMaze.Y+1;**

```

if Map[x,y]=Char_Target then

    Map[x,y]:=Char_SokoInTarget

else

    Map[x,y]:=Char_Soko;

for i:=1 to MaxPosition do

    if GetBit(State.Boxes,i)>0 then

        begin

            x:=i div SokoMaze.Y+1;

            y:=i mod SokoMaze.Y+1;

            if Map[x,y]=Char_Target then

                Map[x,y]:=Char_BoxInTarget

            else

                Map[x,y]:=Char_Box;

            end;

        for i:=1 to SokoMaze.X do

            Writeln(Map[i]);

        end;

```

```
function Solution(State:StateType):boolean;
```

```
var
```

```
    i:integer;
```

```
begin
```

```
    Solution:=false;
```

```
    for i:=1 to MaxPosition do
```

```
        if (GetBit(State.Boxes,i)>0) and (GetBit(SokoMaze.Goals,i)=0) then
```

```
            exit;
```

```
    Solution:=true;
```

```
end;
```

```
function CanReach(State:StateType; Position:integer):boolean;
```

用 BFS 判断在状态 State 中，搬运工是否可以到达 Position

```
var
```

```
    Direction:integer;
```

```
    Pos,NewPos:integer;
```

```
    Get,Put:integer;
```

```
    Queue:array[0..Maxx*Maxy] of integer;
```

```
    Reached:Array[0..Maxx*Maxy] of boolean;
```

**begin**

**fillchar(Reached,sizeof(Reached),0);**

**Pos:=State.ManPosition;**

**Get:=0; Put:=1;**

**Queue[0]:=Pos;**

**Reached[Pos]:=true;**

**CanReach:=true;**

**while Get<>Put do**

**begin**

**Pos:=Queue[Get];**

**inc(Get);**

**if Pos=Position then**

**exit;**

**for Direction:=0 to 3 do**

**begin**

**NewPos:=Pos+DeltaPos[Direction];**

**if Reached[NewPos] then continue;**

**if GetBit(State.Boxes,NewPos)>0 then continue;**

**if GetBit(SokoMaze.Walls,NewPos)>0 then continue;**

**Reached[NewPos]:=true;**

**Queue[Put]:=NewPos;**

**inc(Put);**

**end;**

**end;**

**CanReach:=false;**

**end;**

**function MinPush(BoxPosition,GoalPosition:integer):integer;**

在没有其他箱子的情况下，从 BoxPosition 推到 GoalPosition 至少要多少步。

**var**

**i:integer;**

**Direction:integer;**

**Pos,NewPos,ManPos:integer;**

**Get,Put:integer;**

**Queue:array[0..Maxx\*Maxy] of integer;**

**Distance:Array[0..Maxx\*Maxy] of integer;**

```

begin

    for i:=0 to Maxx*Maxy do

        Distance[i]:=Infinite;

    end for;

    Pos:=BoxPosition;

    Get:=0; Put:=1;

    Queue[0]:=Pos;

    Distance[Pos]:=0;

    while Get<>Put do

        begin

            Pos:=Queue[Get];

            inc(Get);

            if Pos=GoalPosition then

                begin

                    MinPush:=Distance[Pos];

                    exit;

                end;

            for Direction:=0 to 3 do

```



**begin**

**NewPos:=Pos+DeltaPos[Direction];**

**ManPos:=Pos+DeltaPos[Opposite[Direction]];**

**人应该站在后面**

**if Distance[NewPos]<Infinite then continue;**

**if GetBit(SokoMaze.Walls,NewPos)>0 then continue;**

**推不动**

**if GetBit(SokoMaze.Walls,ManPos)>0 then continue;**

**人没有站的地方**

**Distance[NewPos]:=Distance[Pos]+1;**

**Queue[Put]:=NewPos;**

**inc(Put);**

**end;**

**end;**

**MinPush:=Infinite;**

**end;**

**procedure DoMove(State:StateType; Position,Direction:integer; var NewState:StateT**

**ype);**

**var**

**NewPos:integer;**

**begin**

**NewState:=State;**

**NewPos:=Position+DeltaPos[Direction];**

**NewState.ManPosition:=Position;**

**SetBit(NewState.Boxes,NewPos);**

**ClearBit(NewState.Boxes,Position);**

**end;**

**function MinMatch(BoxCount:integer;Gr:BiGraph):integer;**

这个是标准算法，抄的书上的程序，不用看了。

**var**

**VeryBig:integer;**

**TempGr:BiGraph;**

**L:array[1..MaxBox\*2] of integer;**

**SetX,SetY,MatchedX,MatchedY:Set of 1..MaxBox;**

**procedure MaxMatch(n,m:integer);**

```

function Path(x:integer):boolean;

var

    i,j:integer;

begin

    Path:=false;

    for i:=1 to m do

        if not (i in SetY)and(Gr[x,i]<>0) then

            begin

                SetY:=SetY+[i];

                if not (i in MatchedY) then

                    begin

                        Gr[x,i]:=-Gr[x,i];

                        MatchedY:=MatchedY+[i];

                        Path:=true;

                        exit;

                    end;

                j:=1;

                while (j<=m)and not (j in SetX) and (Gr[j,i]>=0) do inc(j);

                if j<=m then

```

**begin**

**SetX:=SetX+[j];**

**if Path(j) then**

**begin**

**Gr[x,i]:=-Gr[x,i];**

**Gr[j,i]:=-Gr[j,i];**

**Path:=true;**

**exit;**

**end;**

**end;**

**end;**

**end;**

**var**

**u,i,j,al:integer;**

**begin**

**Fillchar(L,sizeof(L),0);**

**TempGr:=Gr;**

**for i:=1 to n do**

**for j:=1 to m do**

**if  $L[i] < Gr[i,j]$  then**

**$L[i] := Gr[i,j];$**

**u:=1; MatchedX:=[]; MatchedY:=[];**

**for i:=1 to n do**

**for j:=1 to m do**

**if  $L[i] + L[n+j] = TempGr[i,j]$  then**

**$Gr[i,j] := 1$**

**else**

**$Gr[i,j] := 0;$**

**while  $u \leq n$  do**

**begin**

**SetX:=[u]; SetY:=[];**

**if not (u in MatchedX) then**

**begin**

**if not Path(u) then**

**begin**

**al:=Infinite;**

```

for i:=1 to n do

    for j:=1 to m do

        if (i in SetX) and not (j in SetY) and (L[i]+L[n+j]-TempGr[i,j]<al)

then

        al:=L[i]+L[n+j]-TempGr[i,j];

    for i:=1 to n do if i in SetX then L[i]:=L[i]-al;

    for i:=1 to m do if i in SetY then l[n+i]:=l[n+i]+al;

    for i:=1 to n do

        for j:=1 to m do

            if l[i]+l[n+j]=TempGr[i,j] then

                Gr[i,j]:=1

            else

                Gr[i,j]:=0;

        MatchedX:=[]; MatchedY:=[];

        for i:=1 to n+m do

            if l[i]<-1000 then

                exit;

        end

    else

```

```

        MatchedX:=MatchedX+[u];

        u:=0;

    end;

    inc(u);

end;

end;

var

    i,j:integer;

    Tot:integer;

begin

    VeryBig:=0;

    for i:=1 to BoxCount do

        for j:=1 to BoxCount do

            if (Gr[i,j]<Infinite)and(Gr[i,j]>VeryBig) then

                VeryBig:=Gr[i,j];

            inc(VeryBig);

        end;

    end;

    for i:=1 to BoxCount do

```

**for j:=1 to BoxCount do**

**if Gr[i,j]<Infinite then**

**Gr[i,j]:=VeryBig-Gr[i,j]**

**else**

**Gr[i,j]:=0;**

**这些语句是进行补集转化。**

**MaxMatch(BoxCount,BoxCount);**

**Tot:=0;**

**for i:=1 to BoxCount do**

**begin**

**for j:=1 to BoxCount do**

**if Gr[i,j]<0 then**

**begin**

**Tot:=Tot+VeryBig-TempGr[i,j];**

**break;**

**end;**

**if Gr[i,j]>=0 then**

**begin**

**MinMatch:=Infinite;**



**exit;**

**end;**

**end;**

**MinMatch:=Tot;**

**end;**

**function CalcHeuristicFunction(State:StateType):integer;**

**计算启发函数值**

**var**

**H,Min:integer;**

**i,j,p,Count,BoxCount,Cost:integer;**

**BoxPos:array[1..MaxBox] of integer;**

**Distance:BiGraph;**

**begin**

**p:=0;**

**for i:=1 to MaxPosition do**

**if GetBit(State.Boxes,i)>0 then**

**begin**

**inc(p);**

```

        BoxPos[p]:=i;

    end;

for i:=1 to p do

    for j:=1 to p do

        Distance[i,j]:=MinPush(BoxPos[i],SokoMaze.GoalPosition[j]);

    BoxCount:=SokoMaze.BoxCount;

    H:=0;

    for i:=1 to BoxCount do

    begin

        Count:=0;

        for j:=1 to BoxCount do

            if Distance[i,j]<Infinite then

                inc(Count);

            if Count=0 then

                有一个箱子推不到任何目的地

            begin

                CalcHeuristicFunction:=Infinite;

            exit;

```

**end;**

**end;**

**H:=MinMatch(BoxCount, Distance);**

**CalcHeuristicFunction:=H;**

**end;**

**function HashFunction(State:StateType):integer;**

**var**

**i,h,p:integer;**

**begin**

**h:=0;**

**p:=0;**

**for i:=1 to MaxPosition do**

**if GetBit(State.Boxes,i)>0 then**

**begin**

**inc(p);**

**h:=(h+p\*i) mod HashMask;**

**你可以自己换一个**

**end;**

**HashFunction:=h;**

**end;**

**function SameState(S1,S2:StateType):boolean;**

**var**

**i:integer;**

**begin**

**SameState:=false;**

**for i:=1 to MaxPosition do**

**if GetBit(S1.Boxes,i)<>GetBit(S2.Boxes,i) then**

**exit;**

**if not CanReach(S1,S2.ManPosition) then**

**注意只要两个状态人的位置是相通的就应该算同一个状态**

**exit;**

**SameState:=true;**

**end;**

**function Prior(State:StateType;M1,M2:MoveType):boolean;**

**var**

**NewPos:integer;**

**Inertia1,Inertia2:boolean;**

**S1,S2:StateType;**

**H1,H2:integer;**

**begin**

**Prior:=false;**

**if State.MoveCount>0 then**

**begin**

**NewPos:=State.Move[State.MoveCount].Position+**

**DeltaPos[State.Move[State.MoveCount].Direction];**

**if NewPos=M1.Position then Inertia1:=true else Inertia1:=false;**

**连续推同一个箱子的动作优先**

**if NewPos=M2.Position then Inertia2:=true else Inertia2:=false;**

**if Inertia1 and not Inertia2 then begin Prior:=true; exit; end;**

**if Inertia2 and not Inertia1 then begin Prior:=false; exit; end;**

**end;**

**end;**

**procedure IDA\_Star;**

**var**

**Sucess:boolean;**

**CurrentState:StateType;**

**H:integer;**

**f:Text;**

**procedure IDA\_Push(State:StateType);**

**begin**

**if IDA.Top=MaxStack then**

**Exit;**

**inc(IDA.Top);**

**IDA.Stack[IDA.Top]:=State;**

**end;**

**procedure IDA\_Pop(var State:StateType);**

**begin**

**State:=IDA.Stack[IDA.Top];**

**dec(IDA.Top);**

**end;**

**function IDA\_Empty:boolean;**

**begin**

**IDA\_Empty:=(IDA.Top=0);**

**end;**

上面的是栈操作

**procedure IDA\_AddToHashTable(State:StateType);**

**var**

**h:integer;**

**p:PHashTableEntry;**

**begin**

**h:=HashFunction(State);**

**if HashTable^.Count[h]<MaxSubEntry then**

**begin**

**new(p);**

**p^.State:=State;**

**p^.Next:=HashTable^.FirstEntry[h];**

```

    HashTable^.FirstEntry[h]:=p;

    inc(HashTable^.Count[h]);

end

else begin

    p:=HashTable^.FirstEntry[h];

    while p^.Next^.Next<>nil do

        p:=p^.Next;

        p^.Next^.State:=State;

        p^.Next^.Next:=HashTable^.FirstEntry[h];

        HashTable^.FirstEntry[h]:=p^.Next;

        p^.Next:=nil;

    end;

end;

function IDA_InHashTable(State:StateType):boolean;

var

    h:integer;

    p:PHashTableEntry;

begin

```



```

h:=HashFunction(State);

p:=HashTable^.FirstEntry[h];

IDA_InHashTable:=true;

while p<>nil do

begin

    if SameState(p^.State,State) then

begin

        if p^.State.g>State.g then

begin

            p^.State.g:=State.g;

            IDA_InHashTable:=false;

```

如果找到的表项深度要大些，并不代表这一次深度小点的也无解。本来应该动态更新下界的，这里作为没有找到处理，后面的章节会改进这个地方的。

```

        end;

        exit;

    end;

    p:=p^.Next;

end;

IDA_InHashTable:=false;

```

**end;**

这是 Hash 表的操作。

**procedure IDA\_AddNode(State:StateType);**

**begin**

**IDA\_Push(State);**

**inc(IDA.NodeCount);**

**if IDA.NodeCount mod DispNode=0 then**

**Writeln('NodeCount=',IDA.NodeCount);**

**inc(IDA.TopLevelNodeCount);**

**IDA\_AddToHashTable(State);**

**end;**

**procedure IDA\_Expand(State:StateType);**

**var**

**MoveCount:integer;**

**MoveList:array[1..Maxx\*Maxy\*4] of MoveType;**

**t:MoveType;**

**i,j,Direction:integer;**

```

    NewBoxPos, NewManPos:integer;

    NewState:StateType;

begin

    MoveCount:=0;

    for i:=1 to MaxPosition do

        if GetBit(State.Boxes,i)>0 then

            for Direction:=0 to 3 do

                begin

                    NewBoxPos:=i+DeltaPos[Direction];

                    NewManPos:=i+DeltaPos[Opposite[Direction]];

                    if GetBit(State.Boxes,NewBoxPos)>0 then continue;

                    if GetBit(SokoMaze.Walls,NewBoxPos)>0 then continue;

                    if GetBit(State.Boxes,NewManPos)>0 then continue;

                    if GetBit(SokoMaze.Walls,NewManPos)>0 then continue;

                    if CanReach(State,NewManPos) then

                        begin

                            DoMove(State,i,Direction,NewState);

                            if CalcHeuristicFunction(NewState)=Infinite then continue;

                            if CalcHeuristicFunction(NewState)+State.g>=IDA.PathLimit then con

```

**tinue;**

**IDA\*算法的核心：深度限制**

**if IDA\_InHashTable(NewState) then continue;**

**inc(MoveCount);**

**MoveList[MoveCount].Position:=i;**

**MoveList[MoveCount].Direction:=Direction;**

**end;**

**end;**

**for i:=1 to MoveCount do**

**for j:=i+1 to MoveCount do**

**if Prior(State,MoveList[i],MoveList[j]) then**

**调整推法次序**

**begin**

**t:=MoveList[j];**

**MoveList[j]:=MoveList[i];**

**MoveList[i]:=t;**

**end;**

**for i:=1 to MoveCount do**

**依次考虑所有移动方案**

**begin**

**DoMove(State,MoveList[i].Position,MoveList[i].Direction,NewState);**

**inc(NewState.MoveCount);**

**NewState.Move[NewState.MoveCount].Position:=MoveList[i].Position;**

**NewState.Move[NewState.MoveCount].Direction:=MoveList[i].Direction;**

**NewState.g:=State.g+1;**

**IDA\_AddNode(NewState);**

**end;**

**end;**

**procedure IDA\_Answer(State:StateType);**

**var**

**i:integer;**

**x,y:integer;**

**begin**

**Writeln(f,'Solution Found in ', State.MoveCount,' Pushes');**

**for i:=1 to State.Movecount do**

**begin**

**x:=State.Move[i].Position div SokoMaze.Y+1;**

**y:=State.Move[i].Position mod SokoMaze.Y+1;**

**Writeln(f, x, ' ', y, ' ', DirectionWords[State.Move[i].Direction]);**

**end;**

**end;**

**begin**

**Writeln(VerStr);**

**Writeln(Author);**

**IDA.PathLimit:=CalcHeuristicFunction(IDA.StartState)-1;**

**Sucess:=false;**

**repeat**

**inc(IDA.PathLimit);**

**Writeln('Pathlimit=',IDA.PathLimit);**

**IDA.TopLevelNodeCount:=0;**

**IDA.Top:=0;**

**IDA.StartState.g:=0;**

```

IDA_Push(IDA.StartState);

repeat

    IDA_Pop(CurrentState);

    H:=CalcHeuristicFunction(CurrentState);

    if H=Infinite then continue;

    if Solution(CurrentState) then

        Sucess:=true

    else if IDA.PathLimit>=CurrentState.g+H then

        IDA_Expand(CurrentState);

until Sucess or IDA_Empty or (IDA.NodeCount>MaxNode);

Writeln('PathLimit ',IDA.PathLimit,' Finished. NodeCount=',IDA.NodeCount);

until Sucess or (IDA.PathLimit>=MaxDepth) or (IDA.NodeCount>MaxNode);


Assign(f,outfile);

ReWrite(f);

Writeln(f,VerStr);

Writeln(f,Author);

Writeln(f);

```

**if not Sucess then**

**Writeln(f,'Cannot find a solution.')**

**else**

**IDA\_Answer(CurrentState);**

**Writeln('Node Count:',IDA.NodeCount);**

**Writeln;**

**close(f);**

**end;**

**begin**

**Init;**

**IDA\_Star;**

**end.**