演绎法数独生成和基于深度优先搜索的 Kakuro 求解分级模型

1752161 姚鸿增, 1753309 毕晓栋, 1753763 陈旭阳

Team tsp

2018 年 03 月 31 日

目录

1 背景介绍 **1**

1.1 问题重述 1

1.2 解题途径 1

2 模型假设 **2**

3 符号介绍 **3**

4 模型的建立与求解 **3**

4.1 Kakuro 求解算法 3

4.1.1 概述 3

4.1.2 基本定义 3

4.1.3 交集剪枝法 4

4.1.4 重复性判定剪枝 5

4.1.5 最优化剪枝 6

4.1.6 归谬法剪枝 7

4.1.7 贪心算法 7

4.1.8 算法复杂度 8

4.2 Kakuro 难度划分 8

4.2.1 对运算量的思考 8

4.2.2 对空格数和约束数量的思考 8

4.2.3 基于深度优先搜索树和归谬法次数的难度评估系统 . . 8

4.2.3.1 深度优先搜索树的路径复杂度 8

4.2.3.2 归谬法的次数 10

4.2.3.3 难度函数 10

4.2.3.4 系统稳定性实验 10

4.2.3.5 模型的难度分级标准 10

4.2.4 模型检验 11

4.3 生成 Kakuro 12

4.3.1 概述 12

4.3.2 实例 12
4.4 蜂巢数独求解 15
4.4.1 算法概述 15
4.4.2 基本策略集 15
4.4.3 可行性剪枝 16
4.4.3.1 行内可行性 16
4.4.3.2 行间可行性 16
4.4.4 最优化剪枝 16
4.4.5 算法检验与效果测试 16

5 结论 **18**

附录 **20**

摘要

本文主要建立了 Kakuro 求解算法模型, Kakuro 难度评估模型, 蜂巢数独求解模型, 并且还给出了生成不同难度和具有唯一解的 Kakuro 的方法.

在求解 Kakuro 数独和求解蜂巢数独的模型中, 本文使用深度优先搜索的算法, 采用了回溯的思想, 并采用人们所常用的几个数独技巧作为剪枝方法, 其中包括了可行性剪枝, 最优化剪枝和归谬法剪枝, 实现了两种不同数独的快速求解. 在蜂巢数独求解模型中, 本文将一整行当作一个状态, 从而只用较少的递归次数就能实现数独的求解. 在建立难度评估系统模型中, 本文基于深度优先搜索树的性质, 搜索树上根节点到目标节点的唯一路径的分支总数来刻画数独的难度, 得出了难度评估公式, 并给出了分级标准, 实现了对 Kakuro 数独不同难度的划分. 对于 Kakuro 的生成的方法, 本文使用演绎法, 先确定数独中黑格子的位置, 再一步步地添加约束条件, 在保证数独的唯一解的原则下, 由局部到整体地生成数独.

本文的模型能够在极短的时间内对两种数独进行求解, 在所有搜集到的两种数独之中, Kakuro 都能在 1s 以内计算出结果, 蜂巢数独能在 0.1s 以内完成求解. 本文建立的难度评估系统, 在实例分析中, 能够对不同难度的 Kakuro 数独实现较为准确的分级, 且具有稳定性, 可以不受搜索的随机性的影响. 本文建立的数独生成模型能够生成不同难度具有唯一解的数独, 并且难度可以在生成之前指定.

关键词: 深度优先搜索, 回溯法, 可行性剪枝, 最优化剪枝, 演绎法, 唯一解生成, Kakuro, 蜂巢数独.

1 背景介绍

1.1 问题重述

数独 (Sudoku) 是一种运用纸笔进行验算的智力游戏 [1], 由十八世纪著名数学家欧拉所发明, 旨在通过数字的推理来锻炼人们的思维, 容易上手, 易于沉迷. 而 Kakuro 作为普通数独的推广, 比 Sudoku 更加难玩, 除了涉及逻辑推理, 更要大家计算加数 [2]. 它与 Sudoku 玩法相近, 但趣味更丰富, 挑战性更大.

Kakuro 的主要规则有:

- 在空格中填入数字 1-9, 数字 0 不能出现.
- 带斜线的方格, 斜线上方的数字等于该方格右面对应的一组水平空格的数字之和; 斜线下方的数字, 等于该方格下面对应一组垂直空格里的数字之和.
- 同一个数字在每组水平 (垂直) 的空格里只能出现一次.

在本文中，我们要完成以下任务:

- 讨论并找出一种求解 Kakuro 数独的算法模型, 并且对算法的复杂度进行讨论.
- 给出一种 Kakuro 级别划分系统, 给出具体的划分方式和划分公式, 并且对该级别划分系统进行实例检验.
- 给出一种生成不同难度且具有唯一解的数独的方法.
- 利用 Kakuro 数独的求解模型, 对蜂巢数独进行求解并进行实例检验.

1.2 解题途径

写程序解 Kakuro 之前, 我们先分析了人解 Kakuro 时可能用到的技巧.

1. **数集缩小**: 通过所给的和来确定每行每列的所填情况和候选数集，初步将原先 1—9 的范围缩小.
2. **交叉影线**: 查找交叉的一行一列, 比较它们的候选数集, 则交叉的白格填入的数字一定是它们候选数集交集的元素. 如果元素唯一, 则交叉白格的数字确定. 我们首先要找的便是交集唯一的行与列, 进而确定交叉的白格.

3. **组合参考**: 在使用交叉影线的技巧确定完所有可确定的白格后, 更新每行每列的候选数集 (一般可将能填情况缩小为 1-2 种), 之后进行深一步的考虑. 对于每行 (列), 考虑它候选数集中的最大最小数, 并与交叉的列 (行) 进行比对, 可缩小交叉白格的候选数集或限制所填情况, 进而确定该行唯一的填写情况 (此时序列不确定, 但数的集合确定).

4. **假设推理**: 当白格中填写的数字只有 2 种可能时, 可假设填写其中的一个, 再结合其他行列进行推理判断, 如该数字是否与其他行列的数字重复, 有没有导致其他行列无法达到所给之和等, 进而确定白格中的数字.

在求解 Kakuro 过程中, 我们使用深度优先搜索算法, 并采用了上述几种人们常用的技巧作为剪枝方法. 在求解蜂巢数独的过程中, 我们同样在算法中体现了人们经常使用的技巧. 在建立难度评估系统模型中, 我们基于程序采用技巧时的不确定性以及采用归谬法的次数, 来计算难度评估公式. 对于 Kakuro 的生成的方法, 本文使用演绎法, 运用上述的解题技巧进行逆向推理, 从而生成给定难度的具有唯一解的数独.

2 模型假设

- 本文中讨论的 Kakuro 都是以 8×10 作为大小, 且最上一行和最左一列均为不能填数的格子.
- 本文中的 Kakuro 没有只有一个元素的一组数字.
- 网上搜寻的 Kakuro 符合要求.

3 符号介绍

符号	意义
$\log(x)$	x 的自然对数
n-in-m	和为 n, 组内数字个数为 m 的一种约束
$S(x,y)$	见定义4.1.2
$S_r(x,y)$	见定义4.1.3
$S_c(x,y)$	见定义4.1.3
O	算法时间复杂度上限
W	深度优先搜索树的路径复杂度
R	难度函数
$P(a_1,a_2,\ldots,a_n)$	$\{(a_{b_1},a_{b_2},\ldots,a_{b_n}) \mid \bigcup_{i=1}^{n} b_i = \{1,2,\ldots,n\}\}$
	$a_1,a_2,\ldots,a_n$ 所有排列构成的集合
白格子	允许填入数字的格子
黑格子	不能填入数字的格子

4 模型的建立与求解

4.1 Kakuro 求解算法

4.1.1 概述

首先, 看到问题我们有两种思路, 一种是采用深度优先搜索 (Depth-first search), 另一种是采用广度优先搜索 (Breadth-first search). 如果采用广度优先搜索, 能够避免程序陷入死角, 时间复杂度低, 能更快的实现搜索, 但是空间复杂度高. 而深度优先搜索算法则有较小的空间复杂度. 经过权衡比较, 因为广度优先搜索算法的空间复杂度预估过高, 我们采用了深度优先搜索算法的算法. 可是如果是裸的深度优先搜索算法, 时间复杂度高达 $\Theta(n!)$, 这绝对不是一个能够承受的复杂度, 因此, 我们就要考虑利用人们做 Kakuro 的技巧进行剪枝.

4.1.2 基本定义

定义 4.1.1. 一个白格子的**策略**指经过一定的逻辑推理之后, 可能填入该格子的数.

定义 4.1.2. 对于一个坐标在 (x, y) 的白格子, 其策略的集合为该白格子的**策略集**, 记为 $S(x, y)$, 不引起混淆的情况下, 可简记为 S.

定义 4.1.3. 对于一个坐标在 (x, y) 的白格子, 它被同一行 (列) 的黑格子约束, 则将仅受到此约束时可能填入的数的集合, 称为该白格子的**行 (列) 约束策略集**, 记为 $S_r(x, y)$ $(S_c(x, y))$, 不引起混淆的情况下, 可简记为 S_r (S_c).

为了减小白格子策略集中元素的个数, 从而减小解出白格子的时间复杂度, 我们规定了如下剪枝技巧.

4.1.3 交集剪枝法

根据在第1页的技巧2, 人解 Kakuro 的时候会取行约束策略集和列约束策略集的交集, 以减少白格子策略集的元素.

交集剪枝: 设白格子 (x, y) 分别受到两个黑格子的行约束和列约束, 则

$$S(x, y) = S_r \cap S_c.$$

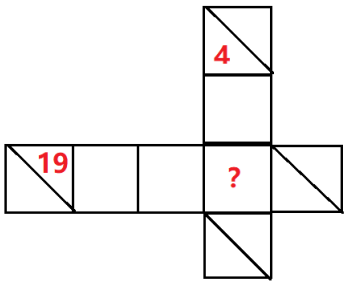

图 1: 交集剪枝的例子 -题目

交集剪枝的例子 如图1, 求问号处白格子的策略集。因为 19-in-3 可分为

$$\begin{aligned} 19 &= 2+8+9 \\ &= 3+7+9 \\ &= 4+6+9 \\ &= 4+7+8 \\ &= 5+6+8, \end{aligned}$$

并且 4-in-2 可分为

$$4 = 1 + 3,$$

因此

$$S_r = \{2, 3, 4, 5, 6, 7, 8, 9\},$$

$$S_c = \{1, 3\}.$$

根据定义4.1.3, 可知 $S = S_r \cap S_c = \{3\}$, 即问号处白格子只能填入 3.

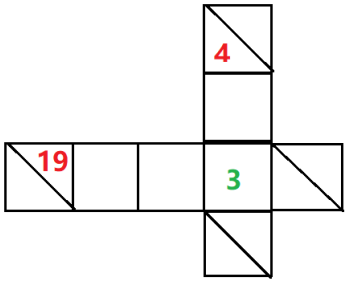

图 2: 交集剪枝的例子 -答案

通过这个例子, 我们可以看出, 运用交集剪枝可以使每个白格子策略集元素个数大大减少, 甚至能够唯一确定, 这将为深度优先搜索的可行性奠定基础. 并且任一 n-in-m 情况的行 (列) 约束策略集是我们能够在程序运行之前通过回溯算法预处理出的, 且时间复杂度较低.

4.1.4 重复性判定剪枝

重复性判定剪枝: 根据在同一个约束下数字不能重复的规则, 通过判断是否重复, 来再次减少策略集元素的个数.

重复性判定剪枝的例子 如上图, 根据交集剪枝, 我们已经知道了问号处白格子的策略集 $S = \{1, 3\}$, 但因为同样的行约束中已经填入了 1, 因此 $S = \{1, 3\} - \{1\} = \{3\}$, 即问号中只能填入 3. 程序中, 我们建立了一个 bool 类型的数组, 来判断在 (x, y) 处的白格子填入 n 是否会造成重复.

通过重复性判定剪枝, 我们可以再次减少策略集元素的个数. 此方法是一个非常有效的剪枝.

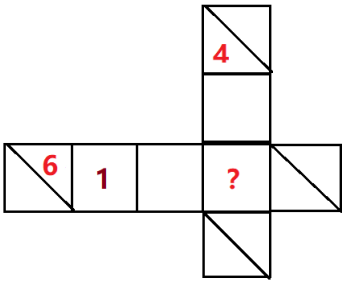

图 3: 重复性判定剪枝的例子

4.1.5 最优化剪枝

在一个约束条件下, 有若干组可能的加数组合. 如果当约束内有一个或多个数被确定, 有可能会导致某种组合不再可能, 从而可能导致有的数不再可能被填入, 因此我们可以将这些数 (不仅仅是已经确定被填入的数) 从策略集中删去. 在具体的算法实现中, 我们定义了如下方法. **最优化剪枝**: 参见在第2页的3, 当我们在某个白格子的策略集中挑选数时, 如果我们所选的数, 使得同一个约束下的别的白格子即使取最优值, 仍然不能满足该约束, 那么我们就舍去这种方案.

图 4: 最优化剪枝的例子

最优化剪枝的例子 我们知道 20-in-3 可分为

$$\begin{aligned} 20 &= 3+8+9 \\ &= 4+7+9 \\ &= 5+7+8, \end{aligned}$$

因为 3 已经填入, 所以问号处白格子的策略集 $S = \{4,5,7,8,9\}$. 如果在问号处填入 4, 5 或 7, 即使空白的白格子中填入 9, 都有这三个数的和小于 20, 无法满足约束条件, 因此问号处白格子的策略集 $S = \{4,5,7,8,9\} - \{4,5,7\} = \{8,9\}$. 策略集中的元素进一步减少.

4.1.6 归谬法剪枝

根据在第2页的技巧4, 如果人无法或难以直接推理出结果, 会采用归谬法. 因此在算法中我们也体现了这一点. **归谬法剪枝**: 当一个白格子的策略集中有不止一个元素, 我们假设填入一个数, 如果经过之后的逻辑推理得出矛盾, 那个可以从策略集中删去这个数.

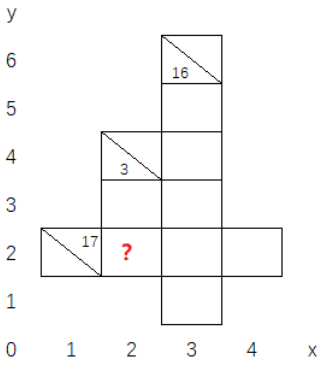

图 5: 归谬法剪枝的例子

归谬法剪枝的例子 这是题目给出样例的局部, 我们从中截取了此例所需的部分, 并定义了坐标. 当我们试图推理问号 (2,2) 处的值时, 易知 $S(2,2)=\{1,2\}$. 如果我们填入 1, 则在问号右边的两个白格子处, 只能一个填入 7, 一个填入 9, 即 $S(3,2)=S(4,2)=\{7,9\}$. 但是 (3,2) 的格子受到了 (3,6) 黑格子的 16-in-5 的列约束, 而 16-in-5 仅可分为

$$16=1+2+3+4+6,$$

即 $S_c(3,2)=\{1,2,3,4,6\}$. 此时发现 $S(3,2)\cap S_c(3,2)=\emptyset$, 推出谬误.

归谬法剪枝是一种十分有效, 对解题关键且难度较大的技巧. 因此, 我们将把程序解 Kakuro 数独时使用归谬法剪枝的次数作为难度分级的重要依据之一.

4.1.7 贪心算法

为了加快搜索速度, 我们采用贪心的思想进行搜索. **贪心算法**是每次选取格子的时候选取策略集内元素最少的格子进行搜索. 这是一种能够保证正确且有效的贪心算法, 而且符合人类解数独的思维, 这个贪心将为我们用程序来衡量数独难度中关键所在.

4.1.8 算法复杂度

因为使用了大量的剪枝, 只能分析出算法的时间复杂度上限为 $O(n!)$, 不过事实上, 实际时间复杂度远小于时间复杂度上限. 以题目中样例为例, 见在第20页的程序运行结果截图, 程序共递归次数仅 1544 次, 所耗时间 0.1 秒之内.

经过大量的测试, 能够搜集到的数独中, 此程序都能在几千次搜索内完成, 运行时间均在 0.1 秒以内, 能实现数独的快速解决. 可见, 本算法是一种高效的解决方案. 程序由 c++ 语言编写, 附在第??页中.

4.2 Kakuro 难度划分

4.2.1 对运算量的思考

因为程序是按照人类的思维进行编写的, 所以其运算量能够在一定程度上反应数独的难度. 但是, 通过测试我们发现, 运算量有很大的随机性. 比如说同一个数独, 从 1 到 9 枚举和从 9 到 1，运算量差距很大. 对于题目给出的样例, 如果逆序枚举策略, 运算只需要 91 次, 远远小于正序枚举策略的 1544 次.

可见, 通过运算量来评估数组难度是不合理的甚至是荒谬的. 因此, 我们需要建立一种新型的评测机制.

4.2.2 对空格数和约束数量的思考

经过实践发现, 即使空格数目相同, 约束数目相同, 且布局相似的数独, 难度仍有天壤之别. 因此空格数目和约束数目无法作为分级的依据.

4.2.3 基于深度优先搜索树和归谬法次数的难度评估系统

4.2.3.1 深度优先搜索树的路径复杂度

图6 (见下页)是深度优先搜索树, 每个节点表示一个数独的状态, 节点的分支数表示某次策略所选的格子的策略集的元素数目.

定义 4.2.1. 路径复杂度:

$$W = 1^{s_1} \times 2^{s_2} \times 3^{s_3},$$

其中

- s_1: 深度优先搜索树初始节点到目标节点的路径上出度为一的节点的数量

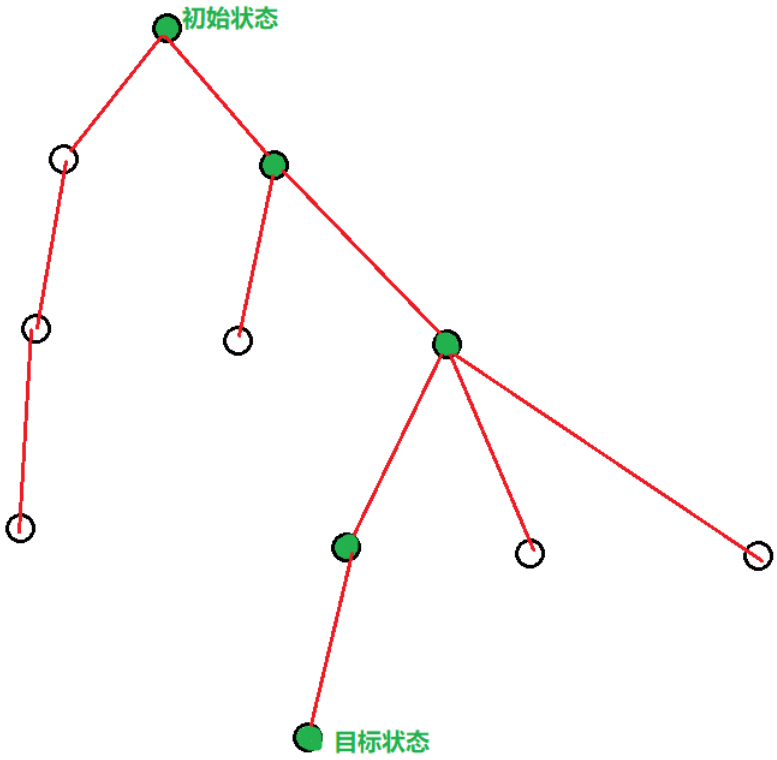

图 6: 深度优先搜索树

- s_2: 深度优先搜索树初始节点到目标节点的路径上出度为二的节点的数量
- s_3: 深度优先搜索树初始节点到目标节点的路径上出度为三的节点的数量

图6中, 路径上的点已经用绿色标识, 其中 $s1 = 2, s2 = 1, s3 = 1$, 则 $W = 2 * 3 = 6$.

可见, W 反应了从初始状态到目标状态中, 所选择策略的不确定性的评估. 当使用前文常用的三个剪枝后, 如果策略还不能唯一确定, 就说明产生了分歧, 而 W 正是一个评测这种分歧多少的量. 注意, W 和总运算量有所不同, 总运算量会受到枚举顺序的影响 (正序枚举策略集和倒序枚举策略集的元素). 但是 W 却不受到此影响, 这是因为树上初始节点和目标节点的路径是唯一的, 因此 s_1, s_2, s_3 不受到枚举顺序的影响, W 也是唯一确定的.

4.2.3.2 归谬法的次数

为了叙述方便, 记程序中归谬法剪枝使用的次数为 c.

在第7页已经说过归谬法是一种难度较大的剪枝技巧. 因此, 程序采用这种技巧的数目也能在一定程度上反映数独的难度. 但是, 需要注意的是, 归谬法使用的次数和总运算次数有一定的联系, 具有一定的随机性, 所以所占的权重不能太大.

4.2.3.3 难度函数

通过思考, c 能够反映从初始状态到目标状态的所使用的难度高的技巧的数目, W 能反应采用了技巧之后的路径复杂度, 则二者的乘积 $W \times c$ 就能客观的反映数独的难度.

定义 4.2.2. 难度函数:

$$R(s_1, s_2, s_3, c) = \log(1^{s_1} \times 2^{s_2} \times 3^{s_3} \times c),$$

注 4.2.1: 因为 W 是指数级别增长的, 所以我们取了自然对数作为难度.

根据此定义, 我们算出了题目给出样例的难度系数 $R = 7.82$, 具体数据在表1 (见下页)

4.2.3.4 系统稳定性实验

改变枚举顺序进行测试, 发现正序和逆序枚举时, 所得的难度系数均为 7.82. 可见, 系统稳定性得到保障, 不受随机因素的影响.

4.2.3.5 模型的难度分级标准

通过实践和调整, 我们划分了标准:

$$\text{难度} = \begin{cases} \text{简单} & \text{R<=3} \\ \text{中等} & \text{3<R<=6} \\ \text{困难} & \text{6<R<=10} \\ \text{地狱} & \text{R>10} \end{cases}$$

表 1: 题目样例与网上找到的 Kakuro 的难度系数

数独原难度	s_1	s_2	s_3	c	R	难度分级
题目案例	39	6	0	39	7.82	困难
简单 [3]	12	0	0	7	1.95	简单
简单 [4, ID: 205183]	43	2	0	31	4.82	中等
中等 [4, ID: 205226]	40	1	0	28	4.03	中等
自己生成 (较简单)	47	1	0	49	4.58	中等
困难 [4, ID: 205191]	42	4	1	41	7.58	困难
困难 [4, ID: 205190]	34	9	0	24	9.42	困难
自己生成 (超困难)	39	11	1	42	12.46	地狱

4.2.4 模型检验

分级系统标准划分后, 我们又在网站上生成了不同难度的 Kakuro. 数据在表1, 原数独和程序的运行结果附在在第21页中. 可以发现, 我们的难度模型是比较可靠和准确的.

4.3 生成 Kakuro

4.3.1 概述

考虑到之前在 Section 4.1完成了求解 Kakuro 的程序, 并且能判断有多少组解, 因此做了随机生成先黑格子, 再往白格子里随机填数, 最后以此确定每个黑格子约束的尝试. 但无论怎么优化, 随机生成的数独都不尽人意, 解的数量十分不稳定, 有时只有 10 组解, 有时却有 62008 组解. 而且就算能随机生成唯一解的数独, 生成的数独一般也会比较简单, 因为数独越难, 意味着给的约束条件越宽松, 使得生成唯一解数独的可能性越小. 因此, 我们决定用**演绎法**人工生成数独.

演绎法是指: 从一个无任何约束的只有白格子和黑格子的空数独开始, 通过逻辑推理确定黑格子的约束, 使得任意给定一个白格子, 都可以经过推理确定它的取值. 运用演绎法时所用逻辑的深度难度, 可以决定该数独的难度, 并且生成数独的难度的上限一般不会很低, 取决于生成者的逻辑水平.

运用演绎法时有一定技巧. 为了减少数的组合, 可以经常使用范围内最大的和或是最小的和作为约束, 还可以运用别的一些创造性手段来消除模棱两可的格子. 一般来说, 生成是由一组交叉的较大的约束和较小的约束开始, 比如 38-in-6 和 10-in-4, 它们交集的元素唯一, 因此它们的交叉点数字就确定了, 只能是 3. 但为了增加难度, 下文给出的例子中, 我们将使用更高难度的逻辑来开始演绎.

4.3.2 实例

在这部分, 我们将会用到许多较难的逻辑, 以体现出我们有能力生成高难度的 Kakuro. 如果想要生成简单的数独, 只需要用简单的逻辑即可.

首先, 我们随机生成了黑格子 (图7), 生成的时候避免了长度为 1 的约束, 并且避免了大量的长度为 2 的约束挤在一起, 以提高难度.

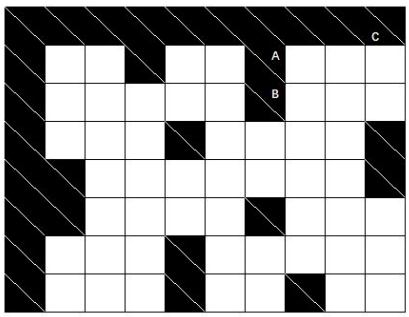

图 7: 生成 Kakuro 过程 1

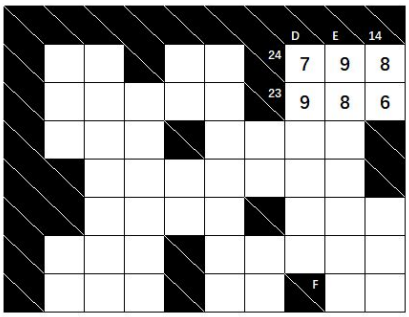

图 8: 生成 Kakuro 过程 2

观察数独右上角, A 和 B 都是长度为 3 的约束,C 是长度为 2 的约束. 我们知道, 只有一种可能分解的长度为 3 的约束有 3, 4, 23 和 24. 我们选择 24 和 23 分别作为 A 和 B 和行约束, 发现分别从两个约束中取值, 和为 14 的情况只有一种, 就是 24 中的 8 加上 23 中的 6, 于是我们选择 14 作为 C 的列约束, 此时右上角的 6 个元素已经被确定 (见图8).

观察 E 所在列, 与 F 所在行有交叉, 并且已经填入了 8 和 9. 而 F 约束的长度为 2, 当 F 的取值为 14, 15, 16 或 17 时都能唯一确定 F 约束下的数字, 我们选取了 16 作为 F 的取值. 再看 D 和 E 的约束, 长度分别为 6 和 7. 不妨使其组合唯一来防止刚开始生成的时候就有过多的可能性. 此时 D 约束的可能取值为 38 和 39, E 约束的可能取值为 41 和 42. 我们取 D 约束为 38, E 约束为 42, 并在 D 列和 E 列记下每个白格子策略集中的元素 (见图9).

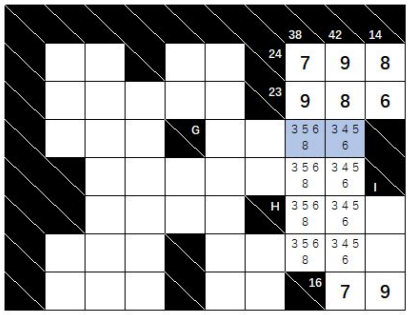

图 9: 生成 Kakuro 过程 3

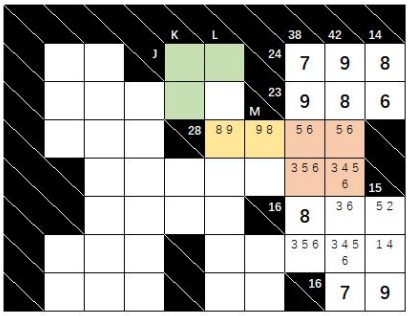

图 10: 生成 Kakuro 过程 4

观察 G 的行约束, 如果值为 29, 那么可以推出左边的蓝色格子值 8, 右边的蓝色格子值为 5, 这样显得太简单了. 于是我们放宽了 G 的行约束, 取值为 28. 此时, 两个蓝色格子中都不能填入 3, 而且如果右边的蓝色格子填入了 4, 那么左边的蓝色格子只能填入 8. 根据这一层推理, 我们要设法使左边的蓝色格子不能填入 8, 这样还能同时缩小右边蓝色格子的策略集. 观察到行约束 H 右边的两个格子, 如果左边的那个格子不填入 8, 那么这两个格子的和最大仅为 11. 据此, 我们在 H 取比较大的约束, 在 I 取比较小的约束, 分别为 16 和 15, 使得如果 H 右边的第一个格子不填入 8, 那么 H 与 I 交叉的那个格子无法取到足够小, 使得 I 列的和不超出 I 的约束. 因此, 此时 H 行右边的一个格子的值为 8. 更新各格子的策略集 (见图10).

注意, 虽然 16 和 15 在 3 个数中的和不算大也不算小, 但是在这种情况下, 它们却共同约束了 H 行左边的格子只能填入 8, 这实在是让人惊叹.

这时我们发现, 因为上方 2 个红色格子中只可能是分别填入 5 和 6, 因此 2 个

黄色格子中只可能分别填入 8 和 9. 为了确定 2 个黄色格子的取值, 我们从 J 和 K 这两个长度为 2 的约束入手. 如果想让 L 下方的绿色格子取到 9, 那么我们需要让 J 取偏大的值, 让 K 取偏小的值, 类似于之前对 H 和 I 的处理, 取 J 为 14, 取 K 为 6, 那么 3 个绿色格子和 2 个黄色格子的取值都能确定. 这时候, 4 个红色格子都能取 5,6 两个值, 为了不让它们导致非唯一解, 在 M 下方第二个格子我们选择填入 6, 即列约束 M 取值为 15. 至于 L, 随意选了一个组合不过多且满足行内 8 与 9 的数 37(组合为 3 种). 更新各格子的策略集 (见图11).

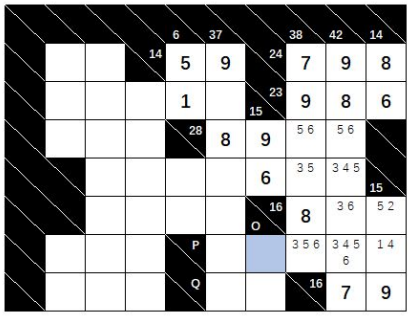

图 11: 生成 Kakuro 过程 5

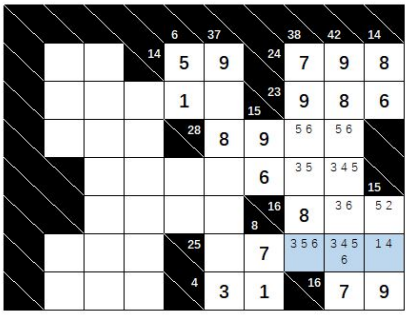

图 12: 生成 Kakuro 过程 6

到现在为止, 依旧没有足够的信息来确定右边那一个区域的白格子. 最后的切入口在中下方 P, Q 和 O 三个约束那里.

经试验, 如果蓝色格子填入 5, 那么蓝色格子右边三个的数的和至多达到 12, 并非 $6+5+4=15$, 再由于蓝色格子的左边已经不能填入 8 或者 9, 所以此时 5 个格子的和不超过 24. 因此我们取定行约束 $P=25$, 这样蓝色格子就不能填入 5 了. P 确定下来之后, 需要确定 O 和 Q 的取值. 它们的取值需要满足下列要求:

- 如果无视约束 P, 那么蓝色格子填入 5 不会引起矛盾.
- 在 P 有约束的情况下, 蓝色格子的取值尽可能唯一.

经过反复斟酌, 我们取了 $O=8$, $Q=4$, 此时数独情况如图12.

图中三个蓝格子, 取值范围为 $\{1,3,4,5,6\}$. 我们在 25-in-5 的 12 种组合中寻找至少包含 $\{1,3,4,5,6\}$ 中三个元素, 且包含 7 的组合, 发现了 4 组, 分别为 $\{1,3,5,7,9\}, \{1,3,6,7,8\}, \{1,4,5,7,8\}, \{3,4,5,6,7\}$, 而若要剔除 7 和 $\{1,3,4,5,6\}$ 中的三个元素后, 满足不与列中的 8 和 9 重复的, 就只有 $\{3,4,5,6,7\}$ 一种组合了. 因此, 最右边的蓝格子中只能填入 4, 进而顺藤摸瓜, 推出整个右边区域的白格子 (见图13). 检查一下, 39-in-7 的那一列已经填入了 3, 6, 8, 9, 依旧是有解的.

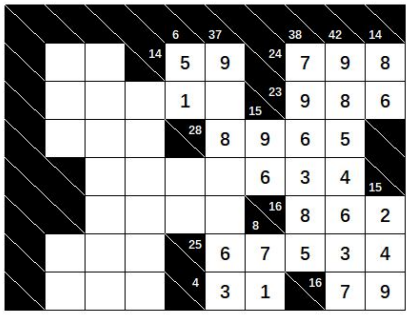

图 13: 生成 Kakuro 过程 7

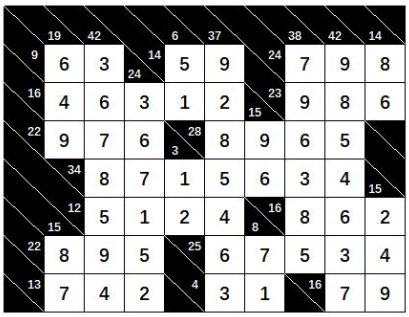

图 14: 生成 Kakuro 结果

至此, 我们已经给出了如何从空数独开始, 从小到大, 一点一点生成数独的主要方法, 之后的具体生成过程就不再列出了. 在此给出最后的生成结果 (见图14) 以及难度评级 (见表1).

除此之外, 我们还生成了一个较简单的数独, 生成结果和难度评级分别能在附录中和前文找到.

4.4 蜂巢数独求解

蜂巢数独求解和 Kakuro 求解问题类似, 我们将采用回溯的思想. 但是此问题和 Kakuro 问题有所不同, Kakuro 问题, 我们采用的深度搜索树的概念比较深刻, 而蜂巢数独问题十分类似信息学的一个经典问题 -八皇后 [5], 其所用的回溯思想更加明显, 因此我们的算法将基于八皇后的回溯思想进行.

下面许多概念可从之前 Kakuro 求解算法中类比过来, 这里不再一一赘述.

4.4.1 算法概述

首先对于第 i 行的元素在该行的策略集中所有可能的策略进行枚举, 判定策略满足约束条件之后, 转到计算第 $i+1$ 的状态. 如果 $i+1$ 行的所有策略均不能满足约束, 则回退, 继续枚举第 i 行其它可能的策略. 这样循环下去, 直到所有的格子都被填充完毕, 算法结束.

4.4.2 基本策略集

对于蜂巢数独的第 i 行, 记其长度为 l_i, l_i 为确定值. 定义第 i 行的**基本策略集** S_{l_i}: 所有连续的且元素数目为 l_i 的所有排列的集合. 例如, 当 $i = 1$ or 9, $l_i = 5$

时, 其基本策略集

$$S_{l_i} = S_5 = P(1,2,3,4,5) \cup P(2,3,4,5,6) \cup P(3,4,5,6,7) \cup P(4,5,6,7,8) \cup P(5,6,7,8,9).$$

4.4.3 可行性剪枝

4.4.3.1 行内可行性

如果在第 i 行有些格子已经由题目给出, 那么我们在枚举 S_{L_i} 策略集内排列的时候, 需要将每种排列与第 i 行题目给出的元素进行比对, 只有完全匹配的时候, 才能将该排列当作一种策略.

4.4.3.2 行间可行性

当我们枚举第 i 行策略集的内部排列时, 如果该排列与第 $i-1$ 行在左撇列或者右撇列有数字重复, 则不满足蜂巢数独的规则, 那么该排列不能当作可行的策略.

程序中, 我们建立了 bool 数组 Vis_Left[s][k](Vis_Right[s][k]), 其中 s 为一个格子所在的左撇列 (右撇列) 的序号, k 为该格子在行排列中的位数. 只有两个标记数组均给出 false, 才表示第 k 位可行, 只有一个行排列的每个位数均可行, 此排列才能作为该行的策略.

4.4.4 最优化剪枝

因为蜂巢数独需要保证数字的连续, 所以如果我们填入的一个数, 与其所在左撇列 (右撇列) 一个已经确定的数的差值大于或等于该列的长度, 那么该左撇列 (右撇列) 的连续性肯定会被破坏. 于是此种排列不能作为一种有效的策略, 应该舍去.

以题干为例子 (见图15 (见下页)), 当搜索到第五行时 (实际上前四行已经填满但图中未标识), 第五行第一个格子可以填写 {1,3,4,6,7,8,9}，但是填写 7, 8 or 9 时, 有

$$|9-2| \geq |8-2| \geq |7-2| \geq 5,$$

因此, 首位数字是 7, 8 或 9 的排列可以被剪枝.

4.4.5 算法检验与效果测试

因为采用了大量的剪枝, 所以程序有很好的效率. 运行样例的两组数据, 得到以下结果: 求解时间在 0.1 秒左右, 搜索次数的数量级也只在几十次. 可见, 本算法

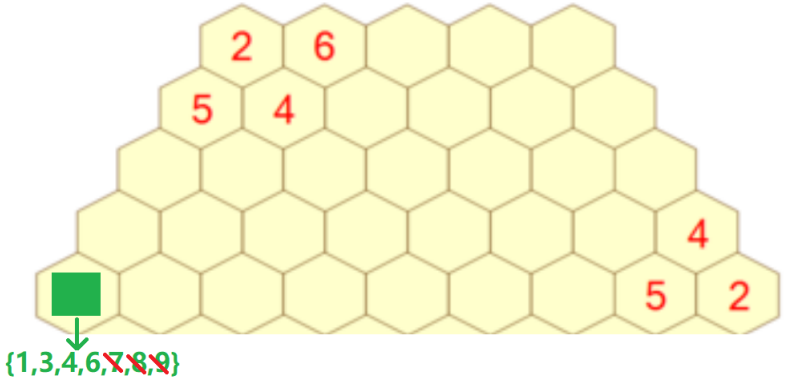

图 15: 蜂巢数独最优化剪枝的例子

表 2: 样例蜂巢数独程序运行结果

组号	递归次数	运行时间
第一组	39	109ms
第二组	52	78ms

是一种高效的解决方案. 实现算法将附在第??页, 程序运行结果截图将附在第28页.

5 结论

优点:

- 本文建立的 Kakuro 数独和蜂巢数独求解模型, 对任务 1 和任务 4 实现了完美的解决, 在所搜集的到不同难度的数独中, Kakuro 数独都能在 1 秒内实现求解, 而蜂巢数独的求解时间甚至达到了 0.1 秒以内.
- 本文建立的难度分级系统, 基于深度优先搜索树和归谬法剪枝次数, 实现了对不同难度的准确分级, 且分级结果是稳定的, 不受搜索策略枚举顺序的影响.
- 本文使用演绎法生成 Kakuro, 可以保证生成的 Kakuro 具有唯一解, 并且能预先指定生成 Kakuro 的难度.

需要改进之处

- 本文的难度分级系统, 最后具体的分类界线画的比较随意, 没有进行过大量数据的测试和修正, 导致可能最后会被分进简单难度的数独很少.
- 本文中的演绎法生成 Kakuro, 生成效率比较低下, 生成困难的数独可能要花好几个小时的时间. 而且生成难度较大数独的时候, 有时也需要碰运气, 如果添加了许多约束但最后发现了矛盾, 就会浪费很多时间.

参考文献

[1] 佚名. 数独 (逻辑游戏)_ 百度百科. https://baike.baidu.com/item/%E6%95%B0%E7%8B%AC/74847. 引用时间: 2018/3/31.

[2] 佚名. Kakuro_ 百度百科. https://baike.baidu.com/item/Kakuro. 引用时间: 2018/3/31.

[3] Dascalu Vald. Kakuro online. www.kakuros.com/?s5x5. 引用时间: 2018/3/30(因网站不提供数独随机因子, 无法给出所引数独的具体链接).

[4] 佚名. Kakuro generator. https://www.kakuro-online.com/generator. 引用时间: 2018/3/31.

[5] 佚名. 八皇后问题 _ 百度百科. https://baike.baidu.com/item/%E5%85%AB%E7%9A%87%E5%90%8E%E9%97%AE%E9%A2%98. 引用时间: 2018/3/30.

附录

程序运行结果

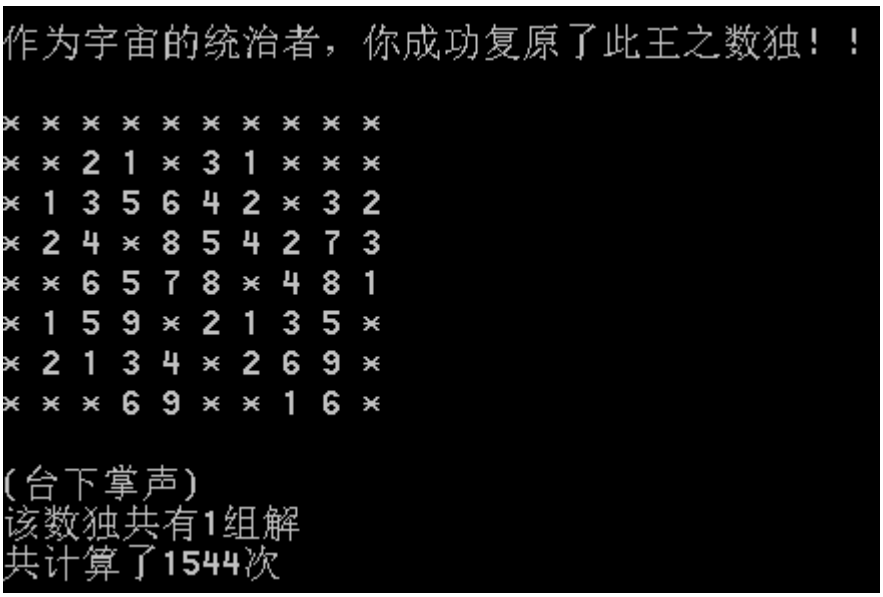

图 16: 样例 Kakuro 的求解结果

```
作为宇宙的统治者，你成功复原了此王之数独！！

* * * * * * * * * *
* * 2 1 * 3 1 * * *
* 1 3 5 6 4 2 * 3 2
* 2 4 * 8 5 4 2 7 3
* * 6 5 7 8 * 4 8 1
* 1 5 9 * 2 1 3 5 *
* 2 1 3 4 * 2 6 9 *
* * * 6 9 * * 1 6 *

(台下掌声)

使用归谬策略的次数: 39
搜索分支数(1,2,3): 39 6 0
难度系数: 7.82244
该数独共有1组解
共计算了91次

--------------------------------
Process exited after 1.343 seconds with return value 0
```

图 17: 样例 Kakuro 的难度分析结果

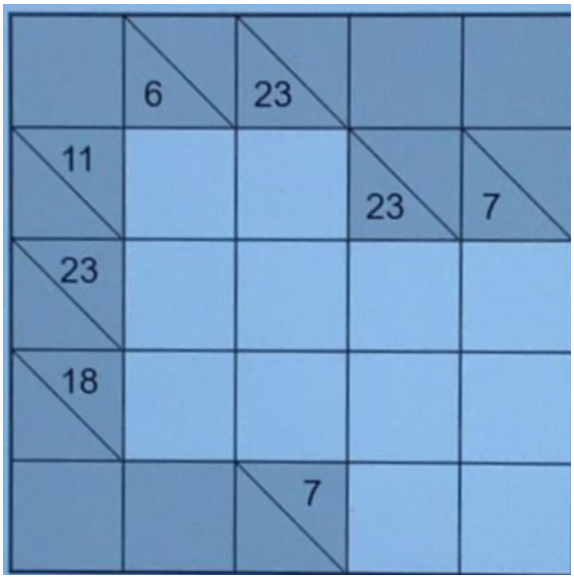

图 18: 第一个简单难度 Kakuro

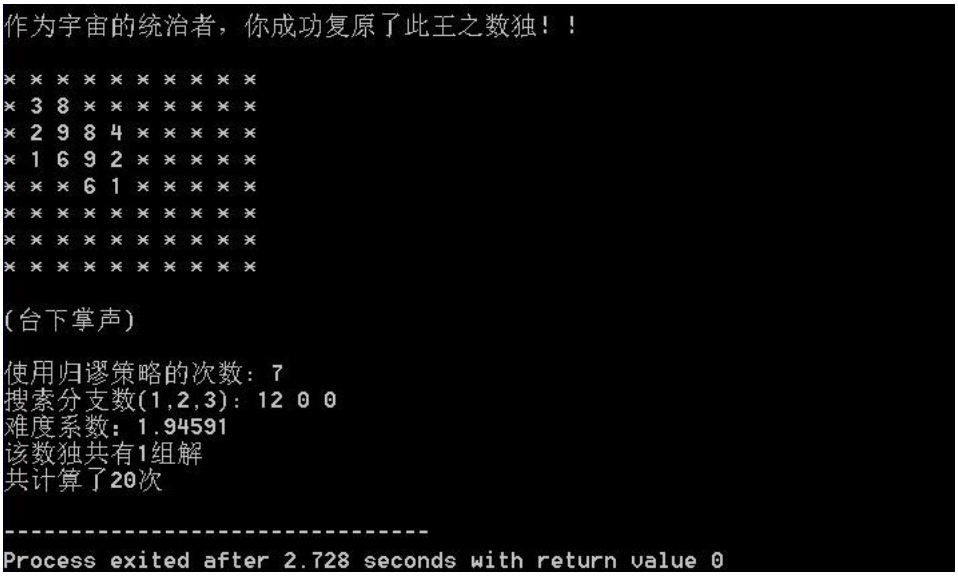

图 19: 第一个简单难度 Kakuro 的难度分析结果

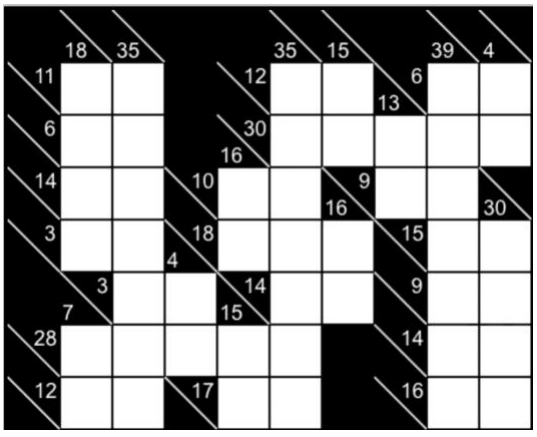

图 20: 第二个简单难度 Kakuro

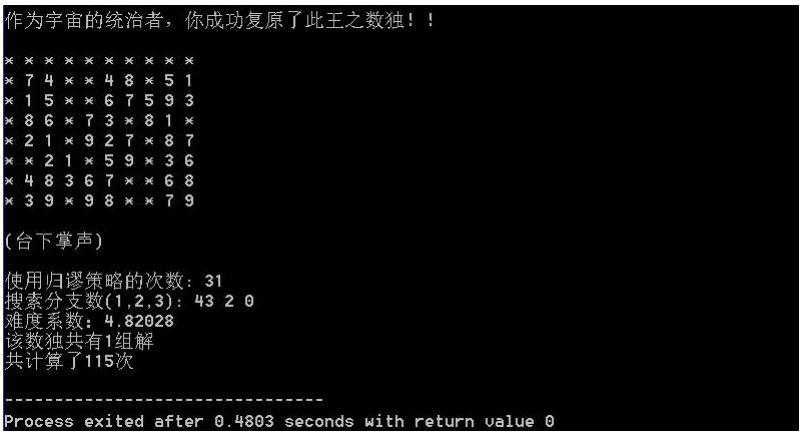

图 21: 第二个简单难度 Kakuro 的难度分析结果

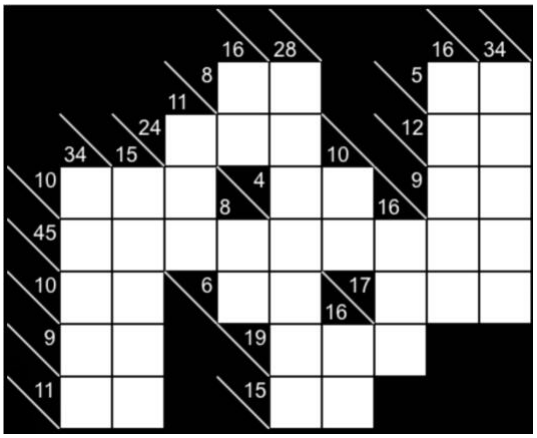

图 22: 中等难度 Kakuro

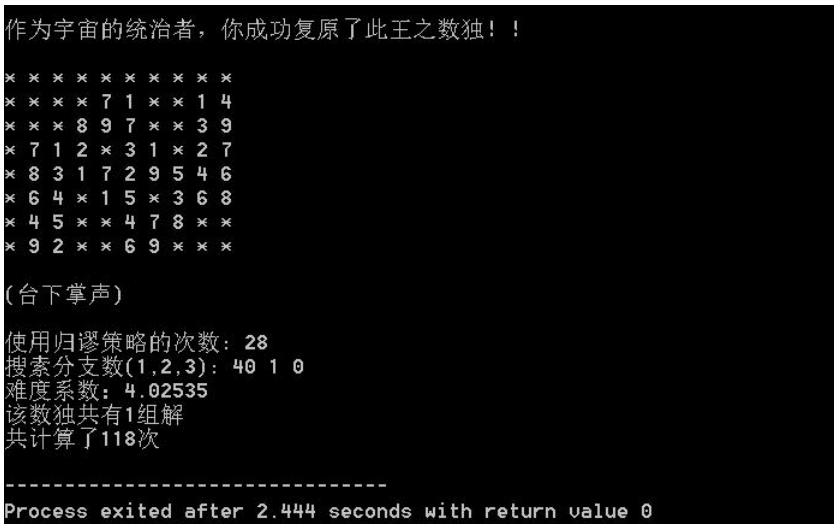

图 23: 中等难度 Kakuro 的难度分析结果

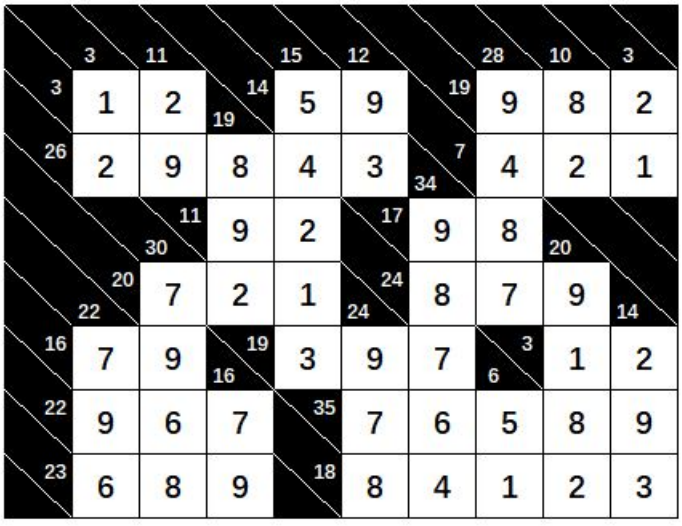

图 24: 自己生成较简单的 Kakuro

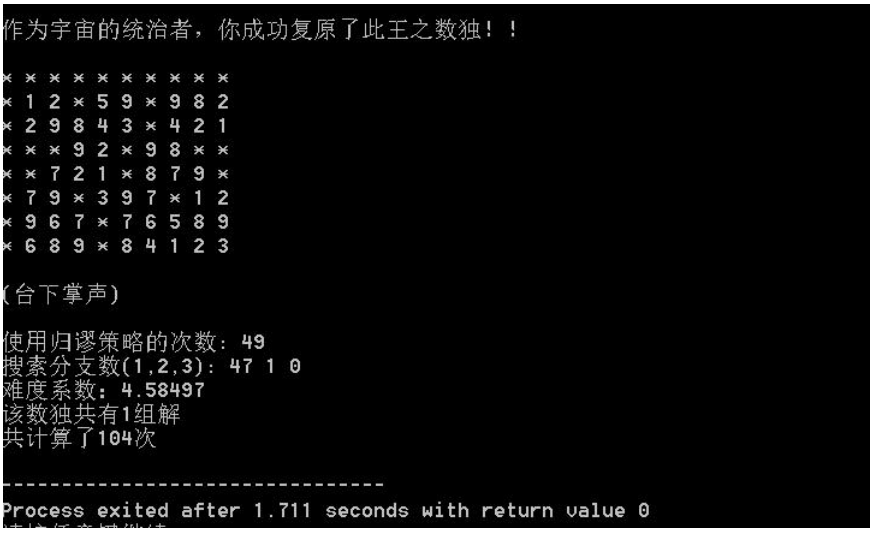

图 25: 自己生成较简单的 Kakuro 的难度分析结果

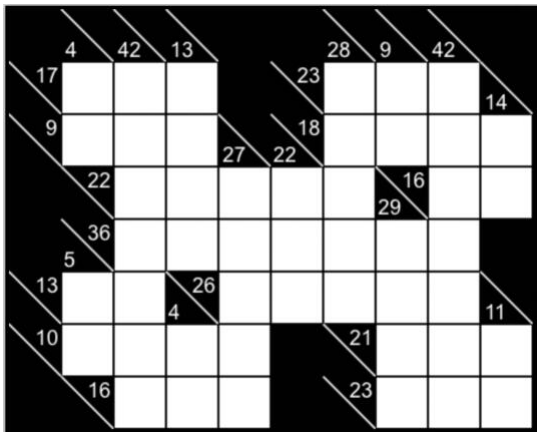

图 26: 第一个困难难度 Kakuro

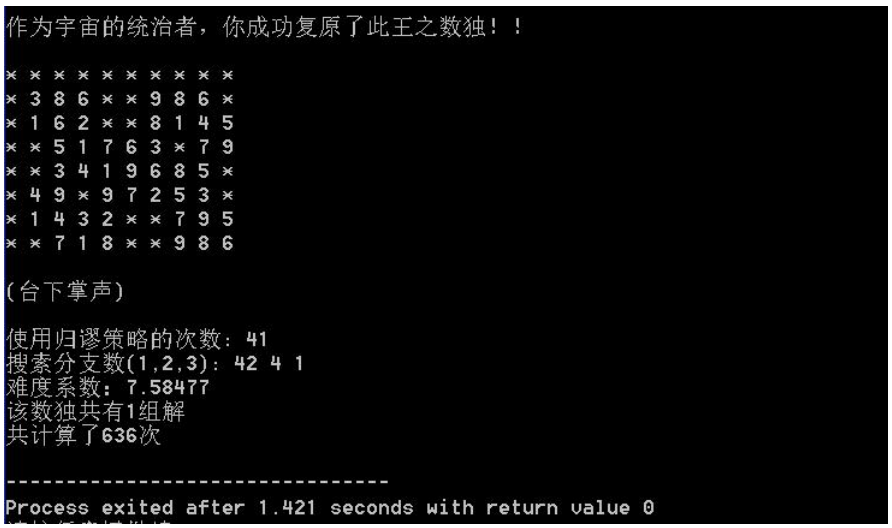

图 27: 第一个困难难度 Kakuro 的难度分析结果

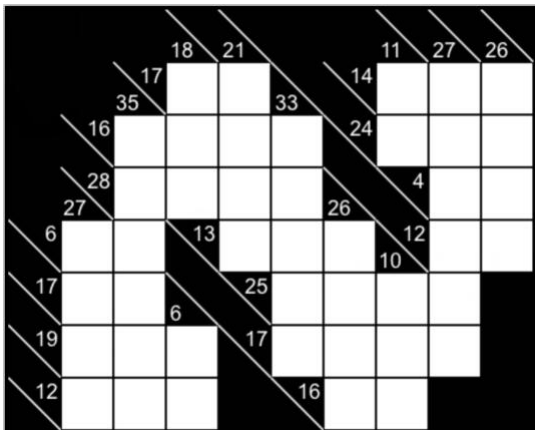

图 28: 第二个困难难度 Kakuro

图 29: 第二个困难难度 Kakuro 的难度分析结果

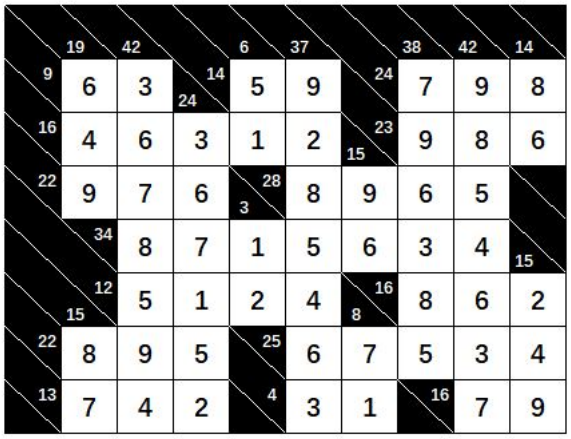

图 30: 自己生成地狱难度 Kakuro

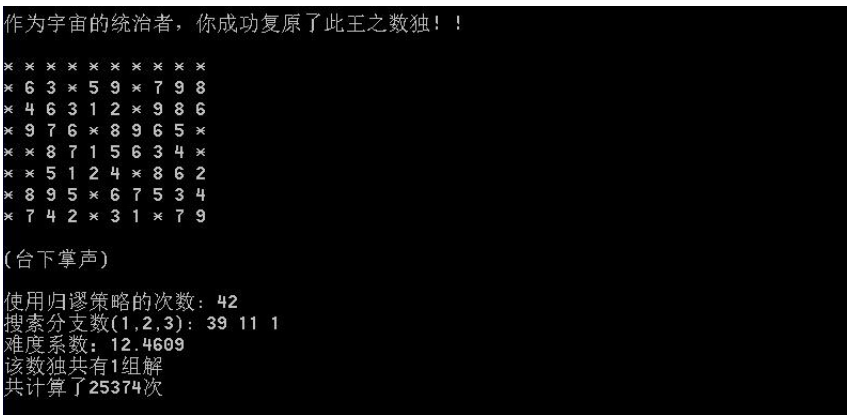

图 31: 自己生成地狱难度 Kakuro 的难度分析结果

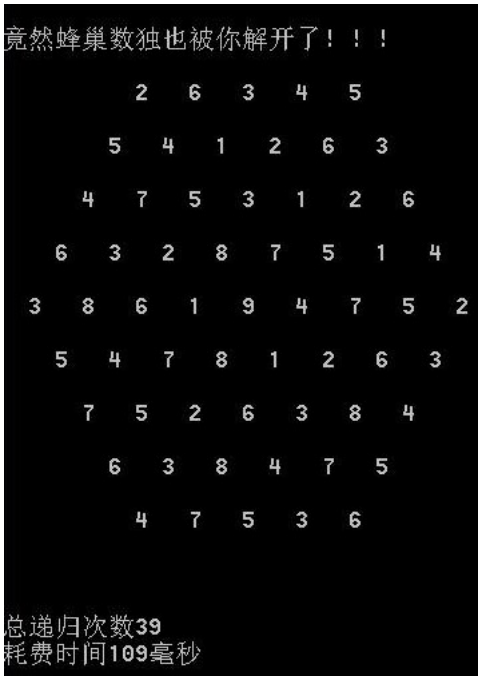

图 32: 蜂巢数独样例 1 的求解结果

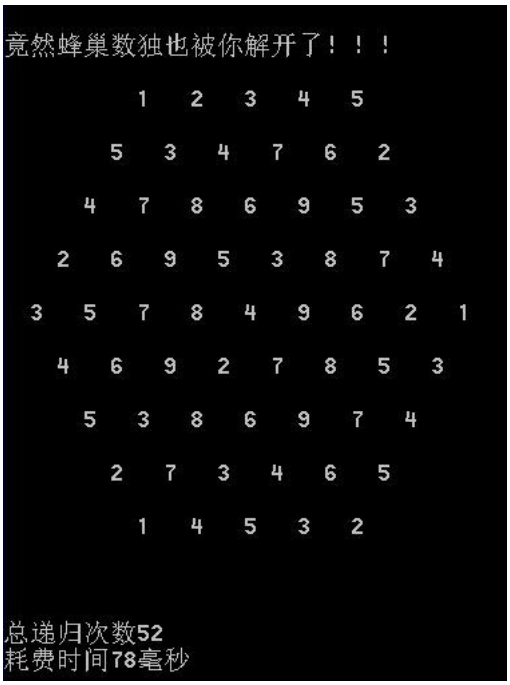

图 33: 蜂巢数独样例 2 的求解结果