

MOZ 2016 全国高校密码数学挑战赛

论文名称： 极大布尔多项式方程组可满足性问题的三种解法

参赛队员： 谢鹏程 司念文

指导教师： 乔 琛

单 位： 西安交通大学

联系方式： 15291851278 9608186@qq.com

2016 年 12 月 17 日

摘 要：本文提出了关于该问题的三种解法：基于遗传算法的解法、基于混合粒子群算法的解法、基于平均分解策略和逐层优化组合方法的解法。

方法一采用遗传算法对问题进行建模。将遗传算法的种群作为方程组的解空间，个体作为方程组的一个可行解。通过对个体进行二进制编码，并选取合适的个体适应度评价函数，使得能让布尔方程组中值为 0 的方程个数越多的个体，其对应的适应度值越高。本文设计的个体适应度函数如下：

$$F(X_i) = 1 - (1 - \frac{len_i - \min len}{\max len - \min len + \varepsilon})^m$$

经过遗传算法中的选择、交叉和变异等操作，适应度高的个体优先被保留并遗传到下一代，直到算法收敛，最终得到该问题的近似最优解。采用 matlab 进行编程实现，经过多次实验，选取最佳实验结果，得到最优解为 182。

方法二采用混合粒子群算法对问题进行建模。通过改进标准的粒子群算法，设计了针对该问题的粒子更新方案，将遗传算法中交叉和变异操作引入到粒子更新中，使得粒子在不断地与个体最优和群体最优的交叉过程中不断移向最优值。同时，设计了一种迭代更新策略，在一定阈值内不断地更新粒子，增加与群体最优交叉次数，提升了粒子的全局搜索能力，算法收敛时得到问题的最优解。采用 matlab 进行编程实现，经过多次实验，选取最佳实验结果，得到最优解为 179。

方法三利用平均分解策略和逐层优化的组合方法。首先，将自变量集合依次分解为 2, 4, 6, 8 组，每组对应一个子集，每个子集自变量个数相同；然后，对每个子集依次进行求解；同时，下一组自变量寻优建立在上一次的最优结果基础上。最后，通过 C 语言编程实现，求得最优解为 172，约占所有方程总数的 2/3。

以上三种方法均成功解决该问题，并且我们对所得结果均进行了验证，提升了结果可信度。综上所述，本题我们求得的最优结果为 182。

关键词：布尔多项方程 最优化 遗传算法 混合粒子群算法 任务分解

目 录

| | |
|--------------------------------|----|
| 第一章 概 述 | 1 |
| 1.1 背景分析 | 1 |
| 1.2 现状分析 | 2 |
| 1.3 本文主要工作 | 3 |
| 第二章 基于遗传算法的解法 | 4 |
| 2.1 符号说明 | 5 |
| 2.2 模型建立 | 6 |
| 2.3 模型求解 | 9 |
| 2.3.1 求解过程 | 9 |
| 2.3.2 最优结果 | 10 |
| 第三章 基于混合粒子群算法的解法 | 13 |
| 3.1 符号说明 | 14 |
| 3.2 模型建立 | 14 |
| 3.3 模型求解 | 16 |
| 3.3.1 求解过程 | 16 |
| 3.3.2 最优结果 | 17 |
| 第四章 基于平均分解策略和逐层优化组合方法的解法 | 19 |
| 4.1 符号说明 | 19 |
| 4.2 模型建立 | 20 |
| 4.3 模型求解 | 21 |
| 4.3.1 求解过程 | 21 |
| 4.3.2 最优结果 | 28 |
| 第五章 结论 | 29 |
| 5.1 模型总结 | 29 |
| 5.2 未来工作 | 30 |
| 参考文献 | 30 |
| 附 录 | 32 |

第一章 概 述

1.1 背景分析

布尔函数是密码学中的重要概念，它在对称密码、Hash 函数和认证码等密码学领域广泛应用[1]。布尔方程组的求解，对于密码分析破译来说，具有十分重要的意义。Shannon[2]于 1949 年提出，对一个密码算法的攻击，相当于求解一个包含大量未定元的方程系统。一般来说，布尔多项式方程组（以下简称布尔方程组）拥有大量的自变量，导致其解空间巨大，搜索的范围也因此扩大，求解一个特定的布尔方程组对计算机的存储能力和运算能力要求极高。因而，在有限的存储和计算资源下，通过找到较好的布尔方程组的求解方法，可有效提升布尔方程组的求解效率，尽可能的找出较多的满足条件的解。

二元域 $GF(2)$ 定义为只含有 0, 1 两个元素的有限域。在极大布尔多项式方程组的可满足性问题（Max-PoSSo 问题）中，给定 m 个含有 n 个变量的布尔多项式：

$$\begin{aligned} f_1(x_1, x_2, \dots, x_n) \\ f_2(x_1, x_2, \dots, x_n) \\ \dots\dots\dots \\ f_m(x_1, x_2, \dots, x_n) \end{aligned} \tag{1.1}$$

试在 $GF(2)$ 中寻找一组 x_1, x_2, \dots, x_n 的赋值（即每个 x_i 的取值均在 $GF(2)$ 中），使得布尔方程 f_1, f_2, \dots, f_m 在这组变量赋值下取值为 0 的个数最多。本题中选取 256 个含有 128 个变元的布尔多项式作为对象，即令 m 的值为 256， n 的值为 128，布尔方程组的具体形式见赛题附录。

在变换 $\{0, 1\}^n \rightarrow \{0, 1\}$ 上定义的函数被称为布尔函数。布尔函数作为密码体制设计与分析中一个不可缺少的工具，一直是密码学研究的重要问题之一。布尔代数上的 n 元布尔多项式 $f(x_1, x_2, \dots, x_n)$ 的形式可以表为：

$$f(x_1, x_2, \dots, x_n) = \sum f(\alpha_1, \alpha_2, \dots, \alpha_n) x_1^{\alpha_1} x_2^{\alpha_2} \dots x_n^{\alpha_n} \tag{1.2}$$

在密码分析中的代数攻击中，攻击者首先将秘密信息（如密钥）设为变量，之后通过已知信息与秘密信息的关系建立相应的多项式方程组，最终通过求解多项式方程

组来恢复秘密信息。然而，在多项式方程组的建立过程中，一些概率假设以及通过物理手段获取信息中出现的噪声，会造成需要求解的多项式方程组的某些项(一般为常数项)出现一定的错误，造成多项式组无法同时取值为 0。此时为了恢复秘密信息，攻击者需要寻找使得方程成立个数最多的解，而该解有很大概率就是未出现错误的方程组的解。

1.2 现状分析

布尔方程组的求解本身是计算困难问题，对其高效算法的探索一直在进行中。考虑本文问题：自变量 X_0, X_1, \dots, X_{127} 共 128 个，布尔多项式方程组共 256 个方程，在给定自变量 $X_i \in \{0, 1\}, i=0, 1, \dots, 127$ 的情况下，找到能够使得值为 0 的方程个数最多的自变量组合。经过对该问题研究分析，我们给出以下几种可能的解决问题的思路。

一是采取直接遍历变量方法，通过依次变换 X_0, X_1, \dots, X_{127} 的 0, 1 值，计算每一种情形下该方程组中值为 0 的函数个数，最后统计出 0 个数最多时对应的自变量的取值，即为该问题的最优解。通过该过程可以发现，直接变量遍历方法有两个重大缺陷：一是自变量的搜索空间巨大，时间复杂度高，导致计算机的计算开销大，普通计算机无法胜任此任务。二是该方法为了寻求最优解，对解空间所有可能解依次遍历，不可避免地遇到许多无效解。而实际密码分析问题中，一般明显不可能的解是没有必要进行分析的，而且并不是所有情形下都需要寻求最优解，有些情形可能只需要找到有关问题的大部分可用信息即可达到密码分析目的。在此情况下，如何找到一种既能满足要求的相对最优解，同时在计算开销上也可接受的求解方法，对于解决实际问题具有十分重要的意义。

二是采用任务分解思想和逐层优化方法组合方法。根据思路一提供的方法，可以对具体问题分解和细化，对各子问题求解最终达到整个问题求解目的。按照该思路，首先，可以采用任务分解策略，将自变量集合依次分组，对每个子集依次进行求解。同时，在每次对单个子集寻优过程中，保存当前子集局部最优解对应的自变量取值。其次，采取逐层优化的方法，下一组变量寻优过程总是建立在上一次求得的最优化结果上，因而所求得局部最优解是逐步逼近全局最优解的，这样最终可逐层求得最后的最优解。

三是采用遗传算法等启发式优化方法。实际上，基于遍历搜索思想的算法是可以应用到 Max-PoSSo 问题当中的，而我们需要做的就是如何有效地将 Max-PoSSo 转化为其它问题，然后可直接采取相应的求解方法，来求解转化后的问题[8]。布尔多项式方程组可满足性问题，本质上可以看作在一定的解空间内，通过确定自变量对应的目标函数，寻求满足约束条件的目标函数最优值及此时对应的自变量组合。解空间即为自变量的多组可能解组成的集合，目标函数即为自变量改变时对应的方程值为 0 的个数，约束条件即为自变量 $X_i \in \{0,1\}$, $i=0,1,\dots,127$ 。这样就可以将该问题转化为优化问题，标准的组合优化问题可定义为以下形式[14]：

$$\min_{X \in D} (F(X) | G(X)) \quad (1.3)$$

其中 X 是决策变量， D 是 X 的定义域， $F(X)$ 是目标函数， $G(X)$ 是约束条件集合。在定义域 D 上，满足条件 $G(X)$ 的解是有限的，目标是求解满足 $G(X)$ 的 $F(X)$ 的最小值。目前，可解决组合优化问题的算法有许多，遗传算法作为其中具有代表性的一种算法，其对解空间的搜索效率和自学习能力都很好，对实际问题的解决不需要太多辅助信息，拥有广泛的研究和应用基础。实践证明，遗传算法对于解决组合优化中的 NP-hard 问题非常有效。此外，我们又针对 Max-PoSSo 问题设计了一种混合粒子群算法，该算法同样可以很好地解决布尔多项式方程组的求解问题。

根据以上问题分析及求解思路，本文拟采用三种问题解决方法：基于遗传算法的解法，基于混合粒子群算法的解法，基于平均分解策略和逐层优化组合方法的解法。

1.3 本文主要工作

针对 Max-PoSSo 问题，本文设计并实现了三种解决方法，每种方法都有一定的创新性，对于解决该问题具有很强的可操作性，同时也给出了具体实验过程以及求得的最优值结果。

本文的主要工作总结如下：

(1) 设计并实现了基于遗传算法的解决方法。将 Max-PoSSo 问题成功地转化为组合优化问题，采用遗传算法建立了模型，设计了个体编码方案，确定了适应度函数计算方法。

(2) 设计并实现了基于混合粒子群算法的解决方法。针对 Max-PoSSo 问题，设计了一种混合粒子群算法，该算法改进了粒子更新方案，通过采用迭代更新策略，扩大了单个粒子的全局搜索范围，增加了算法全局寻优能力。

(3) 设计并实现了基于平均分解策略和逐层优化组合方法的解法。该解法采用任务分解策略和贪心求解思想，将问题由大化小，逐层寻优，最优找到问题的最优解。

本文给出的解决方法有以下几个特点：

(1) 求解问题快速高效。将 Max-PoSSo 问题成功的转化为组合优化问题后，即可采用一套通用的解决方案，不需要太多的人工干预，也不需要额外的领域知识，模型建立完成以后即可自动求解。

(2) 可扩展性能强。本文设计的算法不针对特定的布尔多项式方程组，在变量个数和方程个数改变时，只需要改变算法中编码位的参数，即可扩展到更多变量、更多方程组的求解当中，算法本身无需作任何调整与改变。

(3) 时间复杂度低。本文设计的三种求解方法，主要时间消耗在布尔方程组的运算上，而具体的时间消耗主要取决于算法设定的迭代次数和种群中个体数量。一般而言，迭代数次在 200~300 之间，种群规模在 200 左右即可搜索并收敛到问题的最优值，具体时间消耗在实验结果中给出了分析统计。

第二章 基于遗传算法的解法

遗传算法是一种启发式的最优化方法，它采用了达尔文优胜劣汰的进化机制。其核心思想是：一组潜在的解种群，通过自然进化与自然选择策略进行不断的优化，最终找到最优个体。遗传算法的主要内容包括：设置适应度函数，初始种群规模，然后进行选择、交叉和变异，在多次迭代之后，最后得到该问题的最优解。

遗传算法中的基本概念解释如下：

种群：是指遗传算法开始求解时，初始给定的多个解的集合。一般来讲，种群随机是随机初始化的，也可以经过简单处理给定经过优化的初始种群。

个体：是指种群中的单个元素，通常由一个用于描述其基本遗传结构的数据结构表示，例如可以用 0, 1 组成的串表示个体。

适应度：用来度量种群中各个个体对环境适应性，适应度函数值是遗传算法实现优胜劣汰的主要依据，就衡量个体好坏的量化评价指标。

遗传操作：是指作用于种群中的个体，使之不断进化产生新的种群的操作。标准遗传操作包括：选择、交叉和变异。

遗传算法是从代表解集的一个种群开始的，按照适者生存原理，逐代演化产生出越来越好的近似解。一个种群是由特定数目的经过基因编码的个体组成，每个个体的染色体带有实体特征。初代种群产生之后，在每一代，根据问题域中个体的适应度大小选择个体，并借助于自然遗传学中的遗传算子进行组合交叉和变异，产生出代表新的解集的种群。该过程如同自然界的生物进行一样，后世代种群比前代种群更加适应环境，末代种群中的最优个体经过解码，可作为原始问题的近似最优解。

2.1 符号说明

在应用遗传算法进行建模求解过程中，使用到的符号说明如表 2.1。

表 2.1 符号说明

| 符号 | 含义 |
|-------------------------------|-------------------------|
| M | 表示种群 |
| $X_i, i=1,2,\dots,N$ | 表示种群中个体 |
| $A_i, i=0,1,2,\dots,127$ | 表示个体编码的第 i 位 |
| N | 表示种群规模 |
| P_r | 选择概率 |
| P_c | 交叉概率 |
| P_m | 变异概率 |
| $F(X_i), i=1,2,\dots,N$ | 个体 X_i 的适应度值 |
| $Fitness(X_i), i=1,2,\dots,N$ | 个体 X_i 的相对适应度值 |
| len_i | 表示个体 X_i 对应方程值为 0 的个数 |
| $maxlen$ | 表示当前种群中最优个体对应的 len 值 |
| $minlen$ | 表示当前种群中最劣个体对应的 len 值 |

2.2 模型建立

根据上述对遗传算法的介绍，可以发现 Max-PoSSo 问题能够利用遗传算法进行建模求解。通过对该问题的进一步分析，结合遗传算法中概念及要素的含义，将该问题采用遗传算法建立模型如下。

种群 M: 包含布尔方程组的多个解的集合，种群规模相当于解空间中解的个数。本题采用随机初始化种群的方法，初始种群规模设定为 100，即初始种群中包含 100 个个体。

个体 X: 个体采用二进制编码，由自变量 $A_0A_1...A_{127}$ 组成的 0, 1 数字串构成，单个个体的编码序列表示方程组一个解，多个个体组成种群，经过多代进化后得到的最优个体，其编码的 0, 1 值作为方程组的近似最优解。

种群 M、个体 X 和方程组的解空间、单个解对应关系如下：

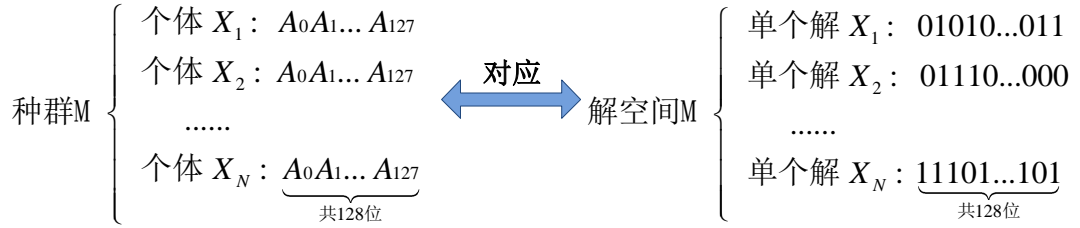


图 2.1 种群与解空间、个体与单个解对应关系

适应度函数 $F(X)$: 对于单个个体 $X_i = A_0A_1...A_{127}$ ，将其对应的自变量代值入到布尔方程组中，值为 0 的方程个数越多则该个体的适应度值越大，在遗传操作中被选中保留的概率越大。经过多次迭代保留下来的个体是适应度高的个体，对应值为 0 的方程个数越多。适应度函数具体计算方法如下：

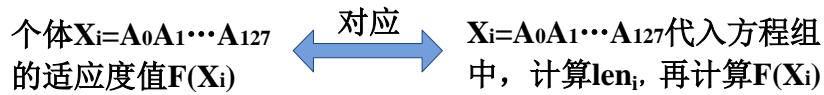


图 2.2 个体适应度值对应关系

本题中，个体 X_i 适应度函数的定义如下（经过归一化处理）：

$$F(X_i) = 1 - \left(1 - \frac{len_i - \min len}{\max len - \min len + \varepsilon}\right)^m \quad (2.1)$$

其中, len_i 表示当前种群中个体 X_i 对应的值为 0 的方程个数; $\min len$ 表示种群所有个体中, 值为 0 的方程最少个数; $\max len$ 表示种群所有个体中, 值为 0 的方程最多个数; ε 表示极小数, 用于保证分母不为 0, 本题中取为 0.0001; m 归一化加速淘汰指数, 加速遗传进化的速度。

选择操作: 按照一定概率每次从种群中选择出一定数目个体, 准备后续交叉和变异操作。实际中有多种选择策略, 本题采用最为常用采用轮盘赌选择方法。轮盘赌选择法中, 个体被选中的概率取决于该个体的相对适应度, 个体 X_i 相对适应度定义为:

$$Fitness(X_i) = \frac{F(X_i)}{\sum_{j=1}^N F(X_j)} \quad (2.2)$$

其中 $Fitness(X_i)$ 表示个体 X_i 相对适应度; $F(X_i)$ 表示个体适应度; N 表示种群规模。

轮盘赌选择法的基本思想, 是根据每个个体的选择概率 $P(X_i)$ 将一个转盘分成 n 个扇区, 其中第 i 个扇区的中心角为:

$$2\pi \frac{f(x_i)}{\sum_{j=1}^n f(x_j)} = 2\pi P(x_i) \quad (2.3)$$

如图 3-1 所示, 个体 i 对应该转盘上第 i 个扇区, 种群中所有个体对应的扇区共同构成转盘。设想在进行选择操作时, 固定指针不动, 随机转动该转盘, 转盘静止时指针指向的扇区即为被选中的个体。从统计学的角度看, 适应度高的个体对应转盘扇区面积大, 在选择操作中被选中的概率也就更大, 此即为转盘赌选择法。

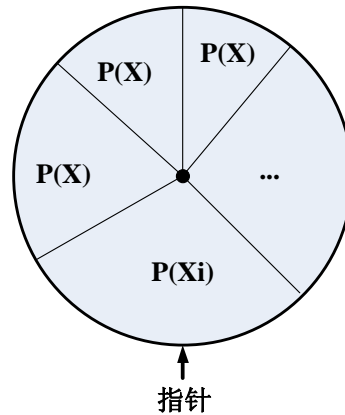


图 2.3 轮盘赌选择法

交叉操作：对选择出来的父代个体染色体的部分基因进行交配重组，从而形成新的个体。本题采用二进制单点交叉。单点交叉是在两个父代个体 X_i 和 X_j 编码串中随机设定一个交叉点，然后对该交叉点前面或后面部分基因进行交换，生成两个新的子代个体，并加入到种群中。假设种群中被选中的两个个体 X_i 和 X_j 编码如下：

$$\begin{aligned} X_i &= a_1 a_2 \dots a_k a_{k+1} \dots a_n \\ X_j &= b_1 b_2 \dots b_k b_{k+1} \dots b_n \end{aligned}$$

图 2.4 个体编码

随机选择第 K 位作为交叉点，采用对 K 位后基因进行交叉，则交叉结果为：

$$\begin{array}{ccc} X_i = a_1 a_2 \dots a_k a_{k+1} \dots a_n & \xrightarrow{\text{交叉}} & X_i = a_1 a_2 \dots a_k b_{k+1} \dots b_n \\ X_j = b_1 b_2 \dots b_k b_{k+1} \dots b_n & & X_j = b_1 b_2 \dots b_k a_{k+1} \dots a_n \end{array}$$

交叉前 交叉后

图 2.5 交叉操作

变异操作：当个体的染色体采用二进制编码时，其变异操作应对应的采用二进制变异方法。在个体编码序列上随机选择一个变异位置，将其值由 0 变为 1（或 1 变为 0），即产生新个体。变异操作可以维持种群多样性，提升算法的局部随机搜索能力。

$$\begin{aligned} A_0 A_1 \dots A_{127} &= 01010001 \dots 10101 \\ &\downarrow \text{变异} \\ (A_0 A_1 \dots A_{127})' &= 01010000 \dots 10101 \end{aligned}$$

图 2.6 变异操作

经过以上使用遗传算法对问题进行建模后，极大布尔多项式方程组可满足性问题即被转化为使用遗传算法在解空间内寻优问题，后续可以采用遗传算法的标准操作，来求得本题的最优个体，及对应的自变量值。遗传算法作为一种比较成熟的组合优化方法，其研究应用已经比较广泛，大量的可参考资料对解决本文所建立的模型提供了方便。

注：本题在实现过程中，为充分保留较好的个体，在选择操作中按照一定概率从种群中选出大量满足要求的个体，然后在选出的个体中进行两两交叉操作。每次从被淘汰的个体中保留 1 个最优个体，加入到种群中。因此，初始种群规模为 100，后期种群规模则在不断变化当中。

2.3 模型求解

2.3.1 求解过程

根据上面建立的遗传算法模型，按照遗传算法的一般求解过程，本文设计了如下的算法：

Step1: 选择编码策略。采用二进制编码策略，按照 0, 1 数字串的形式，对单个个体进行编码，初始种群中的个体编码由随机生成。

Step2: 定义遗传策略。种群规模为 M ，确定选择概率 P_r ，交叉概率 P_c 以及变异概率 P_m 等参数。

Step3: 设定迭代次数 $t=0$ ，按照种群规模随机产生 M 个初始个体，初始化种群。

Step4: 定义适应度函数 $F(X)$ 的计算方法。

Step5: 计算种群中每个个体的适应度值 $F(X_i)$ 。

Step6: 运用选择算子，从种群中选择出个体。

Step7: 按照交叉概率 P_c ，对选出的个体进行交叉操作。

Step8: 按照变异概率 P_m ，对个体进行变异运算。

Step9: 判断达到迭代次数，若未达到迭代次数， $t=t+1$ ，并返回（5）；若达到迭代次数，则保留当前种群中最优个体，作为问题的近似最优解并结束程序。

该算法的流程图如图 2.7 所示：

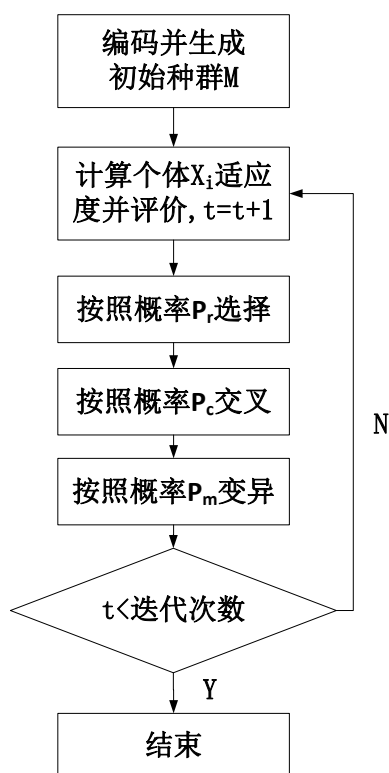


图 2.7 算法流程图

按照以上算法对模型进行求解，具体结果见下一小节。

2.3.2 最优结果

本题采用遗传算法进行模型建立和求解，在 matlab2012b 上进行编程实现，设定初始种群规模 100，个体编码长度 128，设定交叉概率为 0.6，变异概率为 0.1，迭代次数为 200 次。由于初始种群的随机性，导致每次程序运行得到的近似最优解都不相同，其中某次运行结果如图 2.8 和图 2.9 所示（两次不同的实验结果截图）：

```

迭代第198次
当前种群规模: 232
当前种群满足条件方程个数最多为: maxlen=158
本次迭代最优个体为:
1 0 1 0 0 1 1 1 1 1 1 1 1 0 1 0 1 0 0 0 1 1 0 0 1 1 0 0 0 1 0 1 0 0 1 1 1 1 1 1 0 1 0 1 1 1 0 0 0 1 0
迭代第199次
当前种群规模: 233
当前种群满足条件方程个数最多为: maxlen=158
本次迭代最优个体为:
1 0 1 0 0 1 1 1 1 1 1 1 1 0 1 0 1 0 0 0 1 1 0 0 1 1 0 0 0 1 0 1 0 0 1 1 1 1 1 1 0 1 0 1 1 1 0 0 0 1 0
迭代第200次
当前种群规模: 234
当前种群满足条件方程个数最多为: maxlen=158
本次迭代最优个体为:
1 0 1 0 0 1 1 1 1 1 1 1 1 0 1 0 1 0 0 0 1 1 0 0 1 1 0 0 0 1 0 1 0 0 1 1 1 1 1 1 0 1 0 1 1 1 0 0 0 1 0

最优值为: 158
最优个体为: 1 0 1 0 0 1 1 1 1 1 1 1 1 0 1 0 1 0 0 0 1 1 0 0 1 1 0 0 0 1 0 1 0 0 1 1 1 1 1 1 0 1 0 1 1 1
  
```

图 2.8 遗传算法运行结果示意图

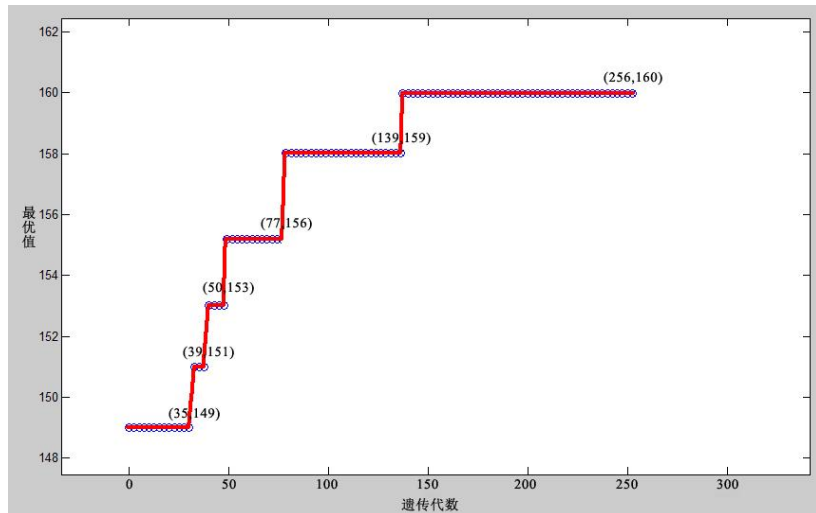


图 2.9 遗传算法运行结果示意图

分析上图容易得出，遗传算法运行结果的最优值，随着遗传代数的不断增加（X轴），在整体趋势上是不断呈上升状态的（Y轴）。也就是说，在遗传进化策略的作用下，各代种群中个体的适应度值越来越高，最优值不断的上升。上图中的最优结果最终收敛于点（256，160），表示遗传算法结束于第256代，此时种群中最优个体对应的 $\max len=160$ ，即该个体能够使方程组中值为0的方程个数为160个。

| Lines where the most time was spent | | | | | |
|-------------------------------------|--|-------|------------|--------|-------------|
| Line Number | Code | Calls | Total Time | % Time | Time Plot |
| 78 | <code>len_l(i,1)=myLength(popm(i,:))...</code> | 603 | 9.318 s | 39.3% | <div></div> |
| 135 | <code>len(i,1)=myLength(popm_sel(i,:...</code> | 606 | 9.301 s | 39.2% | <div></div> |
| 50 | <code>len(i,1)=myLength(popm(i,:)):%...</code> | 200 | 4.728 s | 20.0% | <div></div> |
| 194 | <code>axis([0 C_old 0 200]);</code> | 3 | 0.112 s | 0.5% | |
| 189 | <code>hold on</code> | 3 | 0.044 s | 0.2% | |
| All other lines | | | 0.196 s | 0.8% | <div></div> |
| Totals | | | 23.698 s | 100% | |

图 2.10 主程序时间消耗

| Lines where the most time was spent | | | | | |
|-------------------------------------|--|-------|------------|--------|-------------|
| Line Number | Code | Calls | Total Time | % Time | Time Plot |
| 146 | <code>char(112)= ((x111+x101+x99+x78...</code> | 51609 | 10.045 s | 1.2% | <div></div> |
| 194 | <code>char(160)= ((x113+x112+x111+x1...</code> | 51609 | 9.886 s | 1.2% | <div></div> |
| 247 | <code>char(213)= ((x115+x114+x113+x1...</code> | 51609 | 9.869 s | 1.2% | <div></div> |
| 170 | <code>char(136)= ((x112+x111+x110+x1...</code> | 51609 | 9.808 s | 1.2% | <div></div> |
| 74 | <code>char(40)= ((x108+x107+x106+x97...</code> | 51609 | 9.707 s | 1.2% | <div></div> |
| All other lines | | | 759.533 s | 93.9% | <div></div> |
| Totals | | | 808.847 s | 100% | |

图 2.11 单个布尔多项式时间消耗

遗传算法运行的时间消耗情况如图 2.10 和图 2.11 所示, 由图可知, 该算法的主要运行时间花费在布尔多项式方程组的计算上, 同时, 第 146、194、247 等几个方程在所有方程中计算时间最长, 可定义为复杂方程。这种方程对破解明文信息造成的消耗较大, 表明其在破译中的重要性。

由于遗传算法的整体搜索策略和优化方法在计算上是不依赖于梯度信息和其它辅助知识的,而只需要确定搜索方向的目标函数和相应的适应度函数,所以遗传算法对许多问题进行建模和求解,有一套通用的解决框架。但是遗传算法的“早熟现象”,在初期缺乏有效启发信息的情况下局部收敛速度较慢,给出的最优结果通常是局部最优结果,并不是全局最优的。为了克服该问题,我们在实现中经过多次微调参数,重复进行实验,取得了部分中间结果,其中一些较为优化的结果如下:

(1) maxlen=149

0111111000110001100101000101100001000000000010110100110011110101
1111010001100011101110100101101100000110111010100101111010101001

(2) maxlen=154

10100111111101011001100110001010111111010011000100000111100110
0101111111000101100110101111010010110111101111010011111010010101

(3) maxlen=167

1000100101000100100111100100010000100010010010110110101100110010
011110011011011111011000111000101001101101010001100111100011010

(4) maxlen=170

0001000000010101101010110100000011010000111100100111010100100011
0001001000101110111101000110000110011110110111011110010111001000

(5) maxlen=175

[illegible]

(6) maxlen=182

[illegible]

在上述多次运行实验所得到的结果中，我们取得的最优值为 **182**，对应的个体编码为：

0001000000001010110101011010000001101000011110010011101010010001
1000100100010111011110100011000011001111011011101110010111001000

我们对结果进行了验证,将该个体对应的自变量的值代入到方程组当中,得到值为 0 的方程个数为 182,即 182 为满足条件的正确结果。

第三章 基于混合粒子群算法的解法

粒子群算法 (particle swarm optimization, PSO) 作为一种进化计算方法, 起源于人们对鸟类捕食行为的研究, 其基本思想是通过群体中个体之间的协作与信息共享来寻找最优解。粒子群算法在模拟鸟类群体捕食的社会行为时, 认为个体与个体之间的交互可以有效引导群体离搜索的食物越来越近, 最简单有效的方法就是搜索当前离食物最近的鸟的周围区域。

粒子群算法中的基本概念解释如下:

群体 X : 由多个粒子组成的集合 $X = (X_1, X_2, \dots, X_N)$, 其中每个粒子被抽象成没有质量和体积的点, 初始状态下, 群体中的所有个体是由随机初始化产生的。

粒子: 群体中的单个个体, 没有质量和体积的微粒, 用一个 D 维向量 $X_i = (x_{i1}, x_{i2}, \dots, x_{iD}), i = 1, 2, \dots, N$ 表示, 每一个粒子都代表问题的一个潜在的解。

粒子速度: 用来表示粒子当前飞行速度的 N 维矢量 $V_i = (V_{i1}, V_{i2}, \dots, V_{iD}), i = 1, 2, \dots, N$, 其中每一维都有一个具体数值。粒子速度决定了粒子移动的方向和距离, 速度按照指定的更新公式进行更新, 主要受到自身及群体中其它粒子经验的影响。

粒子位置: 用来表示粒子当前位置的 N 维矢量 $P_i = (P_{i1}, P_{i2}, \dots, P_{iD}), i = 1, 2, \dots, N$, 其中每一维都有一个具体数值。单个粒子总在不断地搜索最佳位置, 以便离目标最近, 这是粒子群算法的基础。

粒子适应度值: 每个粒子都有一个由目标函数决定的适应度值, 该适应度值决定了当前粒子位置的优劣, 是对粒子群体进行更新的主要依据。

粒子群算法首先在可行解空间中初始化一群粒子, 每个粒子都代表对应问题的一个潜在的最优解。每个粒子都有三个重要特征: 位置、速度和适应度值。其中, 适应度值的大小决定了该粒子的优劣, 位置和速度由一定的公式更新:

$$V_{id}^{k+1} = w \times V_{id}^k + c_1 \times rand_1 \times (P_{id}^k - X_{id}^k) + c_2 \times rand_2 \times (P_{gd}^k - X_{id}^k) \quad (3.1)$$

$$X_{id}^{k+1} = X_{id}^k + V_{id}^{k+1} \quad (3.2)$$

上式中， w 是惯性权重， c_1 和 c_2 是加速度因子， r_1 和 r_2 是分布于[0,1]区间的随机数， k 表示当前迭代次数， P_{id} 表示个体最佳值， P_{gd} 当前群体最佳值。粒子速度的更新参考了以前的经验、当前自身的认知以及群体的认知，反映了粒子间的协同合作和信息共享。粒子在解空间中运动，通过跟踪个体极值和群体极值，来更新个体位置，粒子每次更新都计算一次适应度值，通过比较新粒子的适应度值和个体极值、群体极值的适应度值，来更新个体极值和群体极值。

3.1 符号说明

在应用粒子群算法进行建模求解过程中，使用到的符号说明如下。

表 3.1 符号说明

| X | 表示群体 |
|-------------------------------|----------------|
| $X_i, i=1,2,\dots,N$ | 表示群体中单个粒子 |
| $A_i, i=0,1,2,\dots,127$ | 表示个体编码的第 i 位 |
| N | 表示群体规模 |
| $Fitness(X_i), i=1,2,\dots,N$ | 粒子 X_i 的适应度值 |
| P_{id} | 表示个体最佳值 |
| P_{gd} | 表示当前群体最佳值 |
| maxiter | 表示迭代更新阈值 |

3.2 模型建立

相比于遗传算法的复杂，粒子群算法的优势在于过程简单、容易实现，并且没有像遗传算法那样多的参数（如选择、交叉和变异概率）需要调整。同时，粒子在和自身以及其它粒子的信息共享中，使得粒子拥有扩展搜索空间的能力，具有较快的收敛速度。标准的粒子群算法算法流程如下；

- (1) 初始化微粒，包括初始化其位置和速度；
- (2) 根据适应度函数计算每个粒子的适应度值；

- (3) 比较每个微粒，将其适应度值与其自身经过的最好位置作比较，如果较好，则将其作为当前的最好位置；
- (4) 比较每个微粒，将其适应度值与其群体经过的最好位置作比较，如果较好，则将其作为当前的群体的最好位置；
- (5) 未达到结束条件是地，循环（2）、（3）步调整粒子的速度和位置；达到迭代终止条件时，选择当前群体中搜索到的最优粒子的位置及适应度值，作为问题的最终解。

然而，标准的粒子群算法中，粒子速度和位置的更新会产生非整数的情形，并且位置和速度的更新上下限不容易控制。虽然针对组合优化问题也有对应的离散形式的粒子群算法提出，但实际上，粒子速度和位置的更新方式依然未改变，只是在位置的更新中依据一定概率对位置进行简单的取整操作，其实质并未改变。

为了更好的切合本题，为布尔方程建立合适的粒子群算法模型，我们提出一种基于混合粒子群算法的解决方法。与标准的粒子群算法相比，改进的混合粒子群算法具有以下两方面的创新之处：

- 改进了粒子位置和速度的更新方式。针对本题，我们提出将遗传算法中交叉和变异两个操作引入到粒子群算法中，利用交叉和变异操作对粒子位置和速度进行更新，这样避免了标准粒子更新公式将产生小数的问题。
- 设计了一种迭代更新策略。通过确定一个迭代更新阈值，在阈值内不断的迭代更新直到有适应度值高于当前粒子的新粒子产生，本次更新才会结束。这样使得算法在一次进化更新中有了更强的搜索能力，更易找到最优解。

采用改进的粒子群算法对极大布尔多项式方程组进行模型如下：

群体 M：包含布尔方程组的多个解的集合，群体规模相当于解空间中解的个数。本题采用随机初始化种群的方法，初始群体规模设定为 100，即初始群体中包含 100 个个体。

粒子 I：单个粒子采用二进制编码，由自变量 $A_0A_1...A_{127}$ 组成的 0, 1 数字串构成，单个个体的编码序列表示方程组一个解，多个粒子个体组成群体，经过多代进化后得到的最优个体，其编码的 0, 1 值作为方程组的近似最优解。

适应度值：当前粒子的优势的评价指标，是决定粒子是否采用更新操作的主要依据。

交叉和变异操作: 交叉和变异操作用于产生新粒子, 通过计算新粒子的适应度值, 并与当前粒子自身最优值和全局最优值相比较, 决定是否更新粒子。

3.3 模型求解

3.3.1 求解过程

采用改进的混合粒子群算法对问题进行建模后, 具体解题过程按照如下算法进行:

Step1: 选择粒子编码策略。单个粒子采用二进制编码方案, 按照 0, 1 序列串形式编码。

Step2: 设定适应度值计算方法。适应度值是评价粒子好坏的主要指标, 本题中选用单个粒子对应的值为 0 的方程个数作为其适应度值。为 0 的个数越多, 其适应度值越高。

Step3: 更新粒子。采取个体与个体、个体与群体之间的交叉操作, 以及个体自身的变异操作对粒子进行更新, 同时根据适应度值决定当前是否接受当前粒子更新操作。

Step4: 个体最优交叉。将当前粒子个体与其经历过的自身最优进行交叉, 与个体最优的交叉是粒子对自身经验的学习和利用。在迭代更新中, 多次交叉直到产生优于个体最优的粒子, 当前交叉结果方被接受。

Step5: 与群体最优交叉。将当前粒子个体与当前群体最优进行交叉, 与群体最优的交叉是粒子与群体之间其它个体信息共享机制的体现, 有利于粒子朝着更优化的方向进化。在迭代更新中, 多次交叉直到道理优于当前群体最优的粒子, 当前交叉结果方被接受。

Step6: 粒子变异。变异操作是借鉴遗传算法中的变异操作, 是保持群体多样性的一种有效方式。

Step7: 判断达到迭代次数, 若未达到迭代次数, $t=t+1$, 并返回 (2); 若达到迭代次数, 则保留当前群体中最优粒子个体, 作为问题的最优解并结束程序。

该算法的流程图如图 3.1 所示:

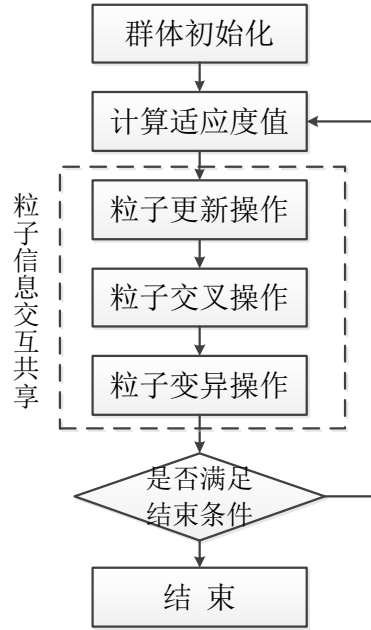


图 3.1 混合粒子群算法流程图

按照以上算法对模型进行求解，具体结果见下一小节。

3.3.2 最优结果

本题采用改进的混合粒子群算法进行模型建立和求解，在 matlab2012b 上进行编程实现，设定初始种群规模 100，个体编码长度 128，迭代次数为 200 次。由于初始种群的随机性，导致每次程序运行得到的近似最优解都不相同，其中某次运行结果如图 3.2 所示。

```

第47次迭代中...
本次迭代最大适应度值: 158
本次迭代最佳粒子: 0101101010011100100010111110110010001101010110101001110010001011
1110110000110011010000110111100110000101000111110010100000010110
第48次迭代中...
本次迭代最大适应度值: 158
本次迭代最佳粒子: 0101101010011100100010111110110010001101010110101001110010001011
1110110000110011010000110111100110000101000111110010100000010110
第49次迭代中...
本次迭代最大适应度值: 158
本次迭代最佳粒子: 0101101010011100100010111110110010001101010110101001110010001011
1110110000110011010000110111100110000101000111110010100000010110
第50次迭代中...
本次迭代最大适应度值: 158
本次迭代最佳粒子: 0101101010011100100010111110110010001101010110101001110010001011
1110110000110011010000110111100110000101000111110010100000010110
fx >>

```

图 3.2 算法运行过程

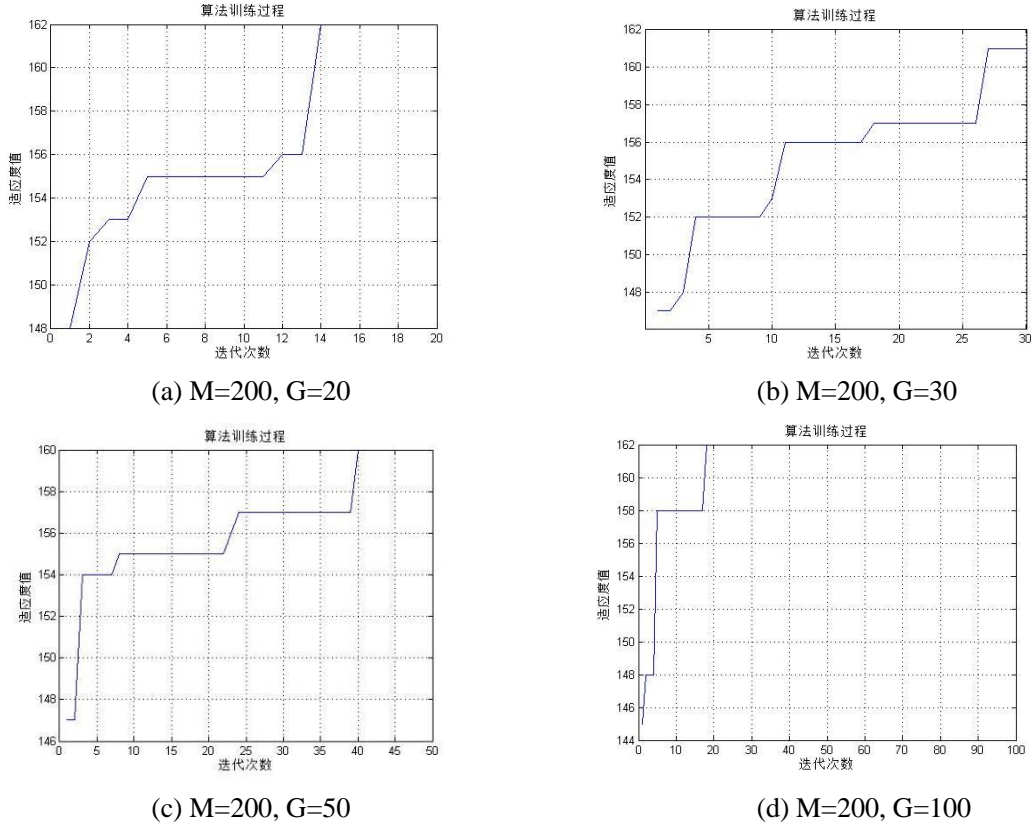


图 3.3 算法运行结果对比 (M 表示群体数量, G 表示迭代次数)

相比于遗传算法, 混合粒子群算法参数较少, 并且收敛较快, 因此可以在更少的进化代数之内快速收敛到最优解。图 3.3 所示是群体数量为 200 时, 进化代数分别设置为 20 代、30 代、50 代和 100 代时, 混合粒子群中算法的寻优路线图。从图 3.3 (d) 中可以看出, 大约进化到 50 代以后, 算法已经收敛, 其后最优值已经没有变化。该算法运行过程中, 同样找出了时间消耗较多的几个布尔多项式方程。经过多次运行, 不断地调整群体规模和进化代数, 我们得到的最优结果为 170, 同时纪录了中间一些相对较优的结果, 如下:

(1) fitness=154

```
1010100001101111001110000010110100110111000100111011101110100011
111111110001100001111100010111001111110001010111101011010010110
```

(2) fitness=157

```
0001100110100001101111001110001000010000100000000100111000001100
010100100011100101000000101100000011000001011010111110110101011
```

(3) fitness=158

```
0101101010011100100010111110110010001101010110101001110010001011
1110110000110011010000110111100110000101000111110010100000010110
```

(4) fitness=164

```
111011111100000011010000111010101111111101110110111000001101111
0000101000110001111110111101010001101110000001001000110011001011
```


(5) fitness=170

100110110011100011011100000000101001110000010010110111110101110
0100011101010110010010001101101001000001011101111101111011001111

(6) fitness=179

1010111010001001110010000110000000010000000010000000000000000000
00

在上述多次运行实验所得到的结果中，我们取得的最优值为 **179**，对应的个体编码为：

1010111010001001110010000110000000010000000010000000000000000000
00

我们对结果进行了验证，将该个体对应的自变量的值代入到方程组当中，得到值为 0 的方程个数为 179，即 179 为满足条件的正确结果。

第四章 基于平均分解策略和逐层优化组合方法的解法

4.1 符号说明

首先定义符号，在本文中我们用该方法来表示任一 n 变量布尔多项式 $L(A_0, A_1, A_2, \dots, A_{n-1})$ ，其中 $A_0, A_1, A_2, \dots, A_{n-1}$ 为布尔变量。

当 $A_0 = 1$ 而其余变量为任意时，记： $L(1, A_1, A_2, \dots, A_{n-1}) = L_{A_0}$

当 $A_0 = 0$ 而其余变量为任意时，记： $L(0, A_1, A_2, \dots, A_{n-1}) = L_{\bar{A}_0}$

类似地，记

$$\begin{aligned} L(1, 1, A_2, \dots, A_{n-1}) &= L_{A_0 A_1} \\ L(1, 0, A_2, \dots, A_{n-1}) &= L_{A_0 \bar{A}_1} \\ L(0, 1, A_2, \dots, A_{n-1}) &= L_{\bar{A}_0 A_1} \\ L(0, 0, A_2, \dots, A_{n-1}) &= L_{\bar{A}_0 \bar{A}_1} \end{aligned} \quad (4.1)$$

其余以此类推。

(三级) 子集合： $\Pi_1, \Pi_2, \Pi_3, \Pi_4, \Pi_5, \Pi_6, \Pi_7, \Pi_8$

子集合的特征值： $\lambda_1, \lambda_2, \lambda_3, \lambda_4, \lambda_5, \lambda_6, \lambda_7, \lambda_8$

布尔方程的可满足个数： $\text{numl} (l=1, 2, 3, \dots, 11)$

可满足个数的优化值： $\text{Betternumk} (k=1, 2, 3, \dots, 9)$

4.2 模型建立

由于该布尔方程组所涉及自变量数目庞大，首先采用平均分解策略，将自变量平均分为 2 组，每组自变量个数为 64，然后固定其中一组自变量值（比如全为 0），对另外一组自变量进行求解。经过平均分组后的自变量，在每组求解时相当于自变量个数减半，相当于将原任务分解为两个子任务分别进行。依次这样进行下去，直到将自变量平均分为 8 组，达到一个分组粒度相对较小，计算复杂度可接受的程度，此时再逐层优化策略进行各组分别寻优。

在初步计算时，将每组内部自变量的值统一赋值为 0 或 1，搜索在该种情形下所有可能的局部最优解，并纪录局部最优解对应的自变量的取值。

然后，依次固定其它 7 组内自变量的取值，对剩下的 1 组自变量（共 16 个）的取值进行遍历，找出该情形下对应的局部最优解，并纪录对应的该组自变量的取值。

这样，该布尔方程组的求解问题，即转化为对各个小部分变量的遍历问题，并且时间复杂度并不是很高。此外，在每次求得各组自变量的局部最优解后，都将其作为下一组自变量遍历过程中其它组自变量值固定下来，因此，每次搜索中得到的局部最优解将是越来越优化的，逐步接近最优解的。

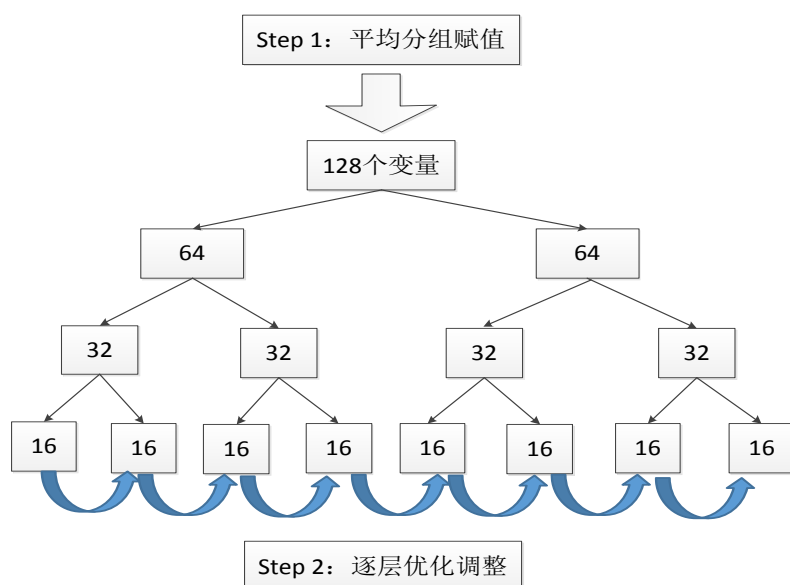


图 4.1 模型示意图

4.3 模型求解

4.3.1 求解过程

在本题的求解过程中，我们主要用了两个步骤来实现对最优解的寻找：一、逐层分组赋值；二、按序局部调整。

第一步：逐层分组赋值并最终确定分组级数

首先我们将数量庞大的 128 个变量进行分组，考虑到计算机的空间和时间运行的合理性，我们采用逐层（三级）的递推分组法：

将这个由 128 个变量组成的变量集合按“顺序临近原则”在数量上均分为 2 个“一级子集”(每个子集的元素个数为 64)，即第一个子集元素为 A_0, A_1, \dots, A_{63} ；第二个子集元素为 $A_{64}, A_{65}, \dots, A_{127}$ 在接下来的赋值中，我们保证在同一子集中的变量取值相同，即位于同一变量子集中的 64 个变量取值同 0 或同 1。对以 2 个“一级子集”为单位的 128 个变量进行遍历赋值，这时共有 $2^2 = 4$ 种赋值选择，记录下在这些赋值选择中布尔方程组中方程等于 0 成立的最大个数 num1。

将前面得到的两个“一级变量子集”分别再按上述分组法各均分为 2 个“二级子集”(共 $2 \times 2 = 4$ 个“二级子集”，每个子集的元素个数为 32)，即第一个子集元素为 A_0, A_1, \dots, A_{31} ；第二个子集元素为 $A_{32}, A_{33}, \dots, A_{63}$ ；.....；第四个子集元素为 $A_{96}, A_{97}, \dots, A_{127}$ 。接下来，我们在赋值时仍保证在同一子集中的变量取值相同，即位于同一变量子集中的 32 个变量取值同 0 或同 1。对以 4 个“二级子集”为单位的 128 个变量进行遍历赋值，这时共有 $2^4 = 16$ 种赋值选择，记录下在这些赋值选择中布尔方程组中方程等于 0 成立的最大个数 num2。将 num2 与 num1 进行大小比较，保留较大者。

依此类推，将前面得到的四个“二级变量子集”分别再按上述分组法各均分为 2 个“三级子集”，此时我们得到如下的 8 个“三级子集”（其中每个子集有 16 个初始变量元素），将这 8 个子集分别记为 $\Pi_1, \Pi_2, \Pi_3, \Pi_4, \Pi_5, \Pi_6, \Pi_7, \Pi_8$ ：

- ① $A_0, A_1, A_2, A_3, A_4, A_5, A_6, A_7, A_8, A_9, A_{10}, A_{11}, A_{12}, A_{13}, A_{14}, A_{15}$
- ② $A_{16}, A_{17}, A_{18}, A_{19}, A_{20}, A_{21}, A_{22}, A_{23}, A_{24}, A_{25}, A_{26}, A_{27}, A_{28}, A_{29}, A_{30}, A_{31}$
- ③ $A_{32}, A_{33}, A_{34}, A_{35}, A_{36}, A_{37}, A_{38}, A_{39}, A_{40}, A_{41}, A_{42}, A_{43}, A_{44}, A_{45}, A_{46}, A_{47}$

.....

⑧ $A_{112}, A_{113}, A_{114}, A_{115}, A_{116}, A_{117}, A_{118}, A_{119}, A_{120}, A_{121}, A_{122}, A_{123}, A_{124}, A_{125}, A_{126}, A_{127}$

在接下来的赋值中，我们保证位于同一变量子集中的 16 个变量取值同 0 或同 1。

我们称某子集 Π_i 中各变量元素所取的值为 Π_i 的特征值 λ_i ($i=1,2,\dots,8$)。对这 8 个特征值 λ_i ($i=1,2,\dots,8$) 进行遍历赋值，这时共有 $2^8 = 256$ 种赋值选择，记录下在这些赋值选择中布尔方程组中方程等于 0 成立的最大个数 num3。将 num3 与前面所保留的较大者进行比较，并保留最终的最大者记作 Betternum1。

通过 C 语言程序运算（程序见附件），我们得到如下的结果：

```
C:\Users\lenovo\Desktop\密码竞赛chengxu\1.exe
133 123 129 124 115 131 118 128 127 118
120 125 115 140 116 121 137 108 121 136
131 126 123 130 117 129 139 123 143 129
123 115 120 132 147 121 139 136 113 122
118 134 134 122 120 128 124 138 112 134
145 129 125 139 110 118 132 119 119 120
114 117 139 122 139 100 129 134 130 120
144 130 123 123 122 134 128 116 125 123
133 146 108 135 132 125 141 144 126 137
136 121 115 130 137 128 146 144 130 120
135 128 138 141 126 141 132 121 123 132
135 120 129 143 131 137 133 123 141 127
127 123 122 130 130 124 117 121 132 129
125 136 137 133 116 131 128 124 138 127
114 143 129 133 135 117 116 139 139 138
120 118 127 130 129 129 134 116 135 132
119 124 140 122 121 129 129 136 132 130
129 124 129 134 117 137 119 131 133 122
128 115 134 140 128 129 133 127 132 138
134 115 142 117 116 132
值为0的方程个数: max_num = 147
对应自变量值: 0 1 0 1 1 1 1 0
-----
Process exited after 0.5441 seconds with return value 0
请按任意键继续. . .
```

图 4.2 程序运行结果

由此可知: Betternum1=147

此时 λ_i ($i=1,2,\dots,8$) 的取值情况如下：

表 4.1 赋值初表

| λ_1 | λ_2 | λ_3 | λ_4 | λ_5 | λ_6 | λ_7 | λ_8 |
|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|
| 0 | 1 | 0 | 1 | 1 | 1 | 1 | 0 |

此时该赋值条件下的方程组可表述为：

$$L\bar{\lambda}_1\lambda_2\bar{\lambda}_3\lambda_4\lambda_5\lambda_6\lambda_7\bar{\lambda}_8$$

第二步：按顺序对各子集中变量值进行局部调整得到更优解

我们固定 $\lambda_2, \lambda_3, \dots, \lambda_8$ 的值如赋值初表所示,对第一个子集 Π_1 中的 16 个变量 $A_0, A_1, A_2, \dots, A_{15}$ 再次进行遍历赋值，这样共有 2^{16} 种遍历情况，记录下在这些赋值选

择中布尔方程组中方程等于 0 成立的最大个数 num4。将 num4 与 Betternum1 进行比较，并保留较大者记作 Betternum2。

通过 C 语言程序运算（程序见附件），我们得到如下的结果：

```

C:\Users\lenovo\Desktop\密码竞赛chengxu\2123456789.exe
值为0的方程个数: max_num = 157, 对应自变量值x0~x15: 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 1, 1
值为0的方程个数: max_num = 157, 对应自变量值x0~x15: 0, 0, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 1, 1
值为0的方程个数: max_num = 157, 对应自变量值x0~x15: 1, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 1, 1
-----
Process exited after 51.07 seconds with return value 0
请按任意键继续. . .

```

图 4.3 程序运行结果

由此可知: $\text{Betternum2}=157$

此时第一个子集 Π_1 中的 16 个变量 $A_0, A_1, A_2, \dots, A_{15}$ 的取值情况如下（有三种）：

表 4.2 赋值表一

| A_0 | A_1 | A_2 | A_3 | A_4 | A_5 | A_6 | A_7 | A_8 | A_9 | A_{10} | A_{11} | A_{12} | A_{13} | A_{14} | A_{15} |
|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|----------|----------|----------|----------|----------|----------|
| 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |
| 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |

我们固定 $\lambda_3, \lambda_4, \dots, \lambda_8$ 的值如赋值初表所示，并固定 Π_1 中的 16 个变量 $A_0, A_1, A_2, \dots, A_{15}$ 的取值如赋值表一所示，对第二个子集 Π_2 中的 16 个变量 $A_{16}, A_{17}, A_{18}, \dots, A_{31}$ 再次进行遍历赋值，这样共有 2^{16} 种遍历情况，记录下在这些赋值选择中布尔方程组中方程等于 0 成立的最大个数 num5。将 num5 与 Betternum2 进行比较，并保留较大者记作 Betternum3。

通过 C 语言程序运算（程序见附件），我们得到如下的结果：

```

C:\Users\lenovo\Desktop\密码竞赛chengxu\3-3.exe
值为0的方程个数: max_num = 167, 对应自变量的值x16~x31: 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0, 0, 1
*
-----
Process exited after 56.56 seconds with return value 0
请按任意键继续. . .

```

图 4.4 程序运行结果

由此可知: $\text{Betternum3}=167$

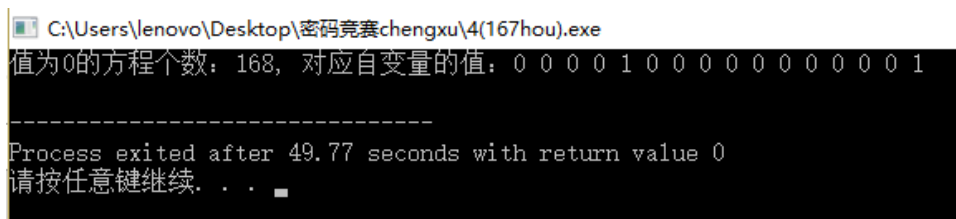
此时第二个子集 Π_2 中的 16 个变量 $A_{16}, A_{17}, A_{18}, \dots, A_{31}$ 的取值情况如下：

表 4.3 赋值表二

| A_{16} | A_{17} | A_{18} | A_{19} | A_{20} | A_{21} | A_{22} | A_{23} |
|----------|----------|----------|----------|----------|----------|----------|----------|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| A_{24} | A_{25} | A_{26} | A_{27} | A_{28} | A_{29} | A_{30} | A_{31} |
| 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 |

我们固定 $\lambda_4, \lambda_5, \dots, \lambda_8$ 的值如赋值初表所示, 并固定 Π_1 中的 16 个变量 $A_0, A_1, A_2, \dots, A_{15}$ 的取值如赋值表一, Π_2 中的 16 个变量 $A_{16}, A_{17}, A_{18}, \dots, A_{31}$ 的取值如赋值表二, 对第三个子集 Π_3 中的 16 个变量 $A_{32}, A_{33}, A_{34}, \dots, A_{47}$ 再次进行遍历赋值, 这样共有 2^{16} 种遍历情况, 记录下在这些赋值选择中布尔方程组中方程等于 0 成立的最大个数 num6。将 num6 与 Betternum3 进行比较, 并保留较大者记作 Betternum4。

通过 C 语言程序运算 (程序见附件), 我们得到如下的结果:



```

C:\Users\lenovo\Desktop\密码竞赛chengxu\4(167hou).exe
值为0的方程个数: 168, 对应自变量的值: 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1
-----
Process exited after 49.77 seconds with return value 0
请按任意键继续. . .

```

图 4.5 程序运行结果

由此可知:

Betternum4=168

此时第三个子集 Π_3 中的 16 个变量 $A_{32}, A_{33}, A_{34}, \dots, A_{47}$ 的取值情况如下:

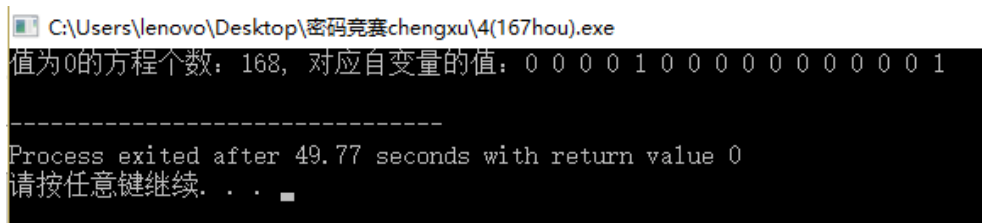
表 4.4 赋值表三

| A_{32} | A_{33} | A_{34} | A_{35} | A_{36} | A_{37} | A_{38} | A_{39} |
|----------|----------|----------|----------|----------|----------|----------|----------|
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| A_{40} | A_{41} | A_{42} | A_{43} | A_{44} | A_{45} | A_{46} | A_{47} |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

我们固定 $\lambda_5, \lambda_6, \dots, \lambda_8$ 的值如赋值初表所示, 并固定 Π_1 中的 16 个变量 $A_0, A_1, A_2, \dots, A_{15}$ 的取值如赋值表一, Π_2 中的 16 个变量 $A_{16}, A_{17}, A_{18}, \dots, A_{31}$ 的取值如赋值表二, Π_3 中的 16 个变量 $A_{32}, A_{33}, A_{34}, \dots, A_{47}$ 的取值如赋值表三, 对第四个子集 Π_4 中的 16 个变量 $A_{48}, A_{49}, A_{50}, \dots, A_{63}$ 再次进行遍历赋值, 这样共有 2^{16} 种遍历情

况,记录下在这些赋值选择中布尔方程组中方程等于 0 成立的最大个数 num7。将 num7 与 Betternum4 进行比较,并保留较大者记作 Betternum5。

通过 C 语言程序运算(程序见附件),我们得到如下的结果:



```

C:\Users\lenovo\Desktop\密码竞赛chengxu\4(167hou).exe
值为0的方程个数: 168, 对应自变量的值: 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1
-----
Process exited after 49.77 seconds with return value 0
请按任意键继续. . .

```

图 4.6 程序运行结果

由此可知: Betternum5=168

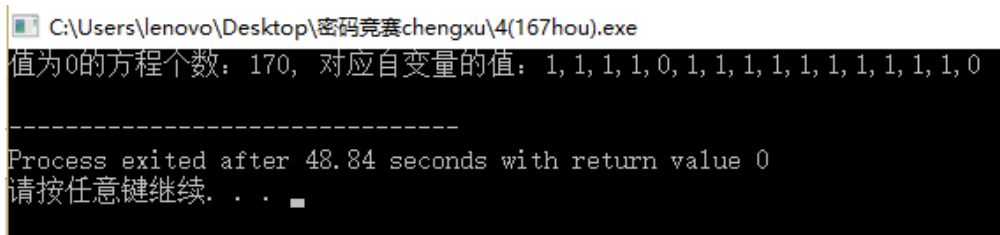
此时第四个子集 Π_4 中的 16 个变量 $A_{48}, A_{49}, A_{50}, \dots, A_{63}$ 的取值情况如下:

表 4.5 赋值表四

| A_{48} | A_{49} | A_{50} | A_{51} | A_{52} | A_{53} | A_{54} | A_{55} |
|----------|----------|----------|----------|----------|----------|----------|----------|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| A_{56} | A_{57} | A_{58} | A_{59} | A_{60} | A_{61} | A_{62} | A_{63} |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

我们固定 $\lambda_6, \lambda_7, \lambda_8$ 的值如赋值初表所示,并固定 Π_1 的 16 个变量 $A_0, A_1, A_2, \dots, A_{15}$ 的取值如赋值表一, Π_2 的 16 个变量 $A_{16}, A_{17}, A_{18}, \dots, A_{31}$ 的取值如赋值表二, Π_3 的 16 个变量 $A_{32}, A_{33}, A_{34}, \dots, A_{47}$ 的取值如赋值表三, Π_4 的 16 个变量 $A_{48}, A_{49}, A_{50}, \dots, A_{63}$ 的取值如赋值表四, 对第五个子集 Π_5 中的 16 个变量 $A_{64}, A_{65}, A_{66}, \dots, A_{79}$ 再次进行遍历赋值, 这样共有 2^{16} 种遍历情况, 记录下在这些赋值选择中布尔方程组中方程等于 0 成立的最大个数 num8。将 num8 与 Betternum5 进行比较, 并保留较大者记作 Betternum6。

通过 C 语言程序运算(程序见附件),我们得到如下的结果:



```

C:\Users\lenovo\Desktop\密码竞赛chengxu\4(167hou).exe
值为0的方程个数: 170, 对应自变量的值: 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0
-----
Process exited after 48.84 seconds with return value 0
请按任意键继续. . .

```

图 4.7 程序运行结果

由此可知: Betternum6=170

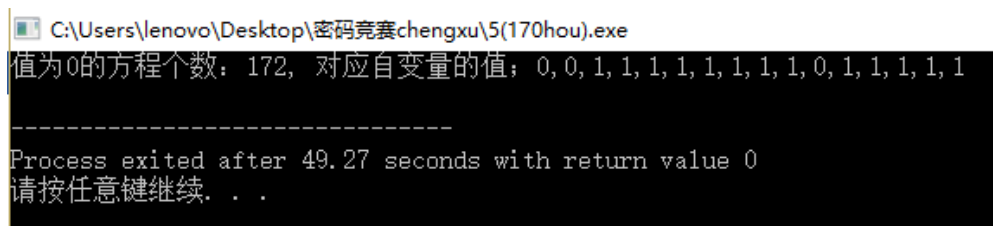
此时第五个子集 Π_5 中的 16 个变量 $A_{64}, A_{65}, A_{66}, \dots, A_{79}$ 的取值情况如下:

表 4.6 赋值表五

| A_{64} | A_{65} | A_{66} | A_{67} | A_{68} | A_{69} | A_{70} | A_{71} |
|----------|----------|----------|----------|----------|----------|----------|----------|
| 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 |
| A_{72} | A_{73} | A_{74} | A_{75} | A_{76} | A_{77} | A_{78} | A_{79} |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |

我们固定 λ_7, λ_8 的值如赋值初表所示,并固定 Π_1 中的 16 个变量 $A_0, A_1, A_2, \dots, A_{15}$ 的取值如赋值表一, Π_2 中的 16 个变量 $A_{16}, A_{17}, A_{18}, \dots, A_{31}$ 的取值如赋值表二, Π_3 中的 16 个变量 $A_{32}, A_{33}, A_{34}, \dots, A_{47}$ 的取值如赋值表三, Π_4 中的 16 个变量 $A_{48}, A_{49}, A_{50}, \dots, A_{63}$ 的取值如赋值表四, Π_5 中的 16 个变量 $A_{64}, A_{65}, A_{66}, \dots, A_{79}$ 的取值如赋值表五, 对第六个子集 Π_6 中的 16 个变量 $A_{80}, A_{81}, A_{82}, \dots, A_{95}$ 再次进行遍历赋值, 这样共有 2^{16} 种遍历情况, 记录下在这些赋值选择中布尔方程组中方程等于 0 成立的最大个数 num9。将 num9 与 Betternum6 进行比较, 并保留较大者记作 Betternum7。

通过 C 语言程序运算 (程序见附件), 我们得到如下的结果:



```

C:\Users\lenovo\Desktop\密码竞赛chengxu\5(170hou).exe
值为0的方程个数: 172, 对应自变量的值: 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1
-----
Process exited after 49.27 seconds with return value 0
请按任意键继续. . .

```

图 4.8 程序运行结果

由此可知:

Betternum7=172

此时第六个子集 Π_6 中的 16 个变量 $A_{80}, A_{81}, A_{82}, \dots, A_{95}$ 的取值情况如下:

表 4.7 赋值表六

| A_{80} | A_{81} | A_{82} | A_{83} | A_{84} | A_{85} | A_{86} | A_{87} |
|----------|----------|----------|----------|----------|----------|----------|----------|
| 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| A_{88} | A_{89} | A_{90} | A_{91} | A_{92} | A_{93} | A_{94} | A_{95} |
| 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 |

我们固定 λ_8 的值如赋值初表所示,并固定 Π_1 中的 16 个变量 $A_0, A_1, A_2, \dots, A_{15}$ 的取值如赋值表一, Π_2 中的 16 个变量 $A_{16}, A_{17}, A_{18}, \dots, A_{31}$ 的取值如赋值表二, Π_3 中

的 16 个变量 $A_{32}, A_{33}, A_{34}, \dots, A_{47}$ 的取值如赋值表三， Π_4 中的 16 个变量 $A_{48}, A_{49}, A_{50}, \dots, A_{63}$ 的取值如赋值表四， Π_5 中的 16 个变量 $A_{64}, A_{65}, A_{66}, \dots, A_{79}$ 的取值如赋值表五， Π_6 中的 16 个变量 $A_{80}, A_{81}, A_{82}, \dots, A_{95}$ 的取值如赋值表六，对第七个子集 Π_7 中的 16 个变量 $A_{96}, A_{97}, A_{98}, \dots, A_{111}$ 再次进行遍历赋值，这样共有 2^{16} 种遍历情况，记录下在这些赋值选择中布尔方程组中方程等于 0 成立的最大个数 num10。将 num10 与 Betternum7 进行比较，并保留较大者记作 Betternum8。

通过 C 语言程序运算（程序见附件），我们得到如下的结果：

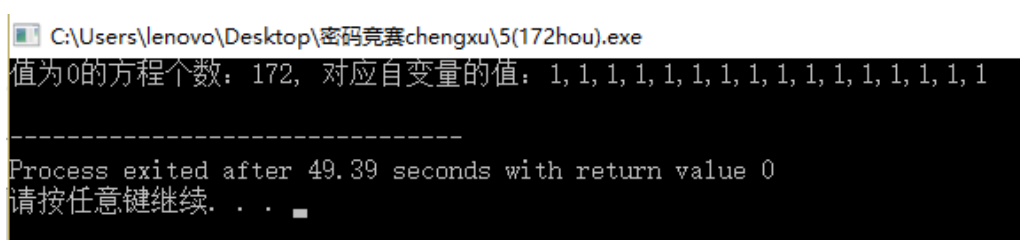


图 4.9 程序运行结果

由此可知：

Betternum8=172

此时第七个子集 Π_7 中的 16 个变量 $A_{96}, A_{97}, A_{98}, \dots, A_{111}$ 的取值情况如下：

表 4.8 赋值表七

| A_{96} | A_{97} | A_{98} | A_{99} | A_{100} | A_{101} | A_{102} | A_{103} |
|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| A_{104} | A_{105} | A_{106} | A_{107} | A_{108} | A_{109} | A_{110} | A_{111} |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

我们固定 Π_1 中的 16 个变量 $A_0, A_1, A_2, \dots, A_{15}$ 的取值如赋值表一， Π_2 中的 16 个变量 $A_{16}, A_{17}, A_{18}, \dots, A_{31}$ 的取值如赋值表二， Π_3 中的 16 个变量 $A_{32}, A_{33}, A_{34}, \dots, A_{47}$ 的取值如赋值表三， Π_4 中的 16 个变量 $A_{48}, A_{49}, A_{50}, \dots, A_{63}$ 的取值如赋值表四， Π_5 中的 16 个变量 $A_{64}, A_{65}, A_{66}, \dots, A_{79}$ 的取值如赋值表五， Π_6 中的 16 个变量 $A_{80}, A_{81}, A_{82}, \dots, A_{95}$ 的取值如赋值表六， Π_7 中的 16 个变量 $A_{96}, A_{97}, A_{98}, \dots, A_{111}$ 的取值如赋值表七，对第八个子集 Π_8 中的 16 个变量 $A_{112}, A_{113}, A_{114}, \dots, A_{127}$ 再次进行遍历赋值，这样共有 2^{16} 种遍历情况，记录在这些赋值选择中布尔方程组中方程等于 0 成立的最大个数 num11。将 num11 与 Betternum8 进行比较，保留较大者记作 Betternum9。

11111011111111111000111111101111111111111111111111110010000000000100

我们对结果进行了验证，将最优值 172 对应的自变量的值代入到方程组当中，得到值为 0 的方程个数为 172，即 172 为满足条件的正确结果。

注：本文在使用“平均分组赋值、逐层优化调整”的方法时，采用了三级双分组的方式，即我们最终将由 128 个变量组成的集合按顺序（脚标顺序）划分成了 8 组三级子集合先来分组赋值，再进行局部调整。事实上，运用相同的思路方法，我们还可以进行：

（1）顺序四级分组，即将整个自变量集合按脚标顺序，划分成 16 组四级子集合，然后分组赋值，再进行局部调整。

（2）运用其他分组方式，例如使用“模 m 剩余类分组法”（ m 为常数， $m|128$ ），即“同余分组法”；“奇偶分组法”；“逆序分组法”；“乱序分组法”等进行初步分组，再进行后续调整优化。

考虑到上述优化方法与本文所介绍的方法虽最终结论可能有小幅度偏差，但思路相同，遂不予赘述。

第五章 结论

5.1 模型总结

本文采用了三种方法思路，从不同角度研究极大布尔多项式方程组的可满足性问题，并给出了三种不同的解法。这三种解法所得结果相近，值为 0 的方程个数占有方程数目的 $2/3$ ，基本满足了密码分析破译和数据还原的需求。

方法一利用遗传算法对 Max-PoSSo 进行建模，将遗传算法的种群作为方程组的解空间，二进制编码的个体作为方程组的一个可行解，通过设置选取合适的适应度函数，使得能使布尔方程组中值为 0 的方程个数越多的个体，其适应度值越高。经过遗传算法中的选择、交叉和变异等操作，使得适应度高的个体优先被保留下来，最终得到该问题的近似最优解。

方法二利用改进的粒子群算法对 Max-PoSSo 进行建模,该方法在群体,单个粒子编码方案以及适应度函数设计上与遗传算法相同。不同的是,在混合粒子群算法中单个粒子通过不断的与个体最优、群体最优交换信息,使得该粒子不断的向最优解的方向移动。通过设计的粒子更新方案和迭代更新策略,使得该算法的全局搜索能力进一步增强,最终收敛到问题的最优解。

方法三利用分解策略和逐层优化方法,将极大布尔方程组的可满足性问题分成各组逐步优化任务来完成。考虑到实际问题的解决效率及运算开销,通过采用平均分解策略,将自变量集合依次分组,对每个子集依次进行求解。同时,在每次对单个子集寻优过程中,保存当前子集局部最优解对应的自变量取值。在此基础上,采取逐层优化的方法,下一组变量寻优过程总是建立在上一次求得的最优化结果上,因而所求得局部最优解是逐步逼近全局最优解的。

经过程序实现,该三种方法所得结果相近,表明了这两种解法的正确性。并且我们对所得结果均代入原问题中进行了验证,发现值为 0 的方程个数与之对应,进一步提升了结果的可信度。

5.2 未来工作

对于方法一和方法二,由于遗传算法初始解空间采用随机化策略,使得算法容易陷入局部最优。一种改进的思路是对初始种群经过优化后再使用,比如利用方法三中求得的相对较优结果作为初始种群,在此基础上进行遗传操作,所得结果会有所提升。

对于方法三,本文提出的方法对解决实际问题有一定的意义,但是该方法仍然有一定的缺陷,在实际操作中,自变量取值仍然有较大的搜索空间,后期将对各组自变量的搜索策略进行改进,减少搜索分支,进一步提升该方法的效率。

参考文献

- [1] 抗代数攻击布尔函数的构造与分析, 陈华谨。
- [2] C.E. Shannon. Communication theory of secrecy systems [J]. Bell Technical Journal, 1949, 28(4): 656-715.

- [3] 张建民, 沈胜宇, 李思昆. 最小布尔不可满足子式的求解算法[J]. 电子学报, 2009, 37(5):993-999.
- [4] 帅训波, 马书南. 应用布尔遗传算子求解 N 皇后问题[J]. 计算机工程与应用, 2011, 47(16):49-51.
- [5] 李云强, 孙怀波, 王爱兰. 布尔函数和伪布尔函数多项式表示的快速实现算法[J]. 计算机工程与应用, 2007, 43(1):50-52.
- [6] 王培根. 布尔函数与布尔多项式[J]. 首都师范大学学报(自然科学版), 2006, 27(5):15-18.
- [7] 李昕, 林东岱, 徐琳. 一种布尔多项式的高效计算机表示[J]. 计算机研究与发展, 2012, 49(12):2568-2574.
- [8] Albrecht, M.R. and Cid, C.: Cold Boot Key Recovery by Solving Polynomial Systems with Noise. ACNS 2011: 57-72.
- [9] Huang, Z. and Lin, D.: A New Method for Solving Polynomial Systems with Noise over F_2 and Its Applications in Cold Boot Key Recovery, Selected Areas in Cryptography, LNCS 7707, pp 16-33, 2013.
- [10] Bard G V, Courtois N T, Jefferson C. Efficient methods for conversion and solution of sparse systems of low-degree multivariate polynomials over $GF(2)$ via SAT-solvers. 2007.
- [11] Gao X S, Huang Z. Characteristic set algorithms for equation solving in finite fields[J]. Journal of Symbolic Computation, 2012, 47(6): 655-679.
- [12] 梁科, 夏定纯. Matlab 环境下的遗传算法程序设计及优化问题求解[J]. 电脑知识与技术:学术交流, 2007, 1(4):1049-1051.
- [13] 史峰. MATLAB 智能算法 30 个案例分析[M]. 北京航空航天大学出版社, 2011.
- [14] 杨天奇. 人工智能及其应用[M]. 暨南大学出版社, 2014.

附 录

方法一：遗传算法 matlab 程序（部分）

```
clear;
clc;
%%%%%%%%%%%%%输入参数%%%%%%%%%
N=128;          %%编码的位数
M=100;          %%种群中个体的个数
C=200;          %%迭代次数
C_old=C;
m=2;            %%适应值归一化淘汰加速指数
Pc=0.6;         %%交叉概率
Pmutation=0.05; %%变异概率

%生成初始群体
popm_0=rand(M,N);
popm=round(popm_0);

%%随机选择一个种群
R=popm(1,:);
%%初始化种群及其适应函数
fitness=zeros(M,1);%生成 M*1 全 0 矩阵
len=zeros(M,1);%生成 100*1 的矩阵，用于暂存适应度函数值
for i=1:M
    len(i,1)=myLength(popm(i,:));%mylength 个体距离计算函数
end
maxlen=max(len);%返回 len 向量中的最大值
minlen=min(len);%返回 len 向量中的最小值
fitness=fit(len,m,maxlen,minlen);%fit 函数用于计算适应度函数值
rr=find(len==maxlen);%找到 len 中最小值的位置
R=popm(rr(1,1),:);% r(1,1)实际上就是 rr 的值，是一个标量
fprintf('随机生成的最初种群中，最优个体为：\n')
for i=1:N
    fprintf('%d ',R(i));%打印
end
fprintf('\n\n');
fitness=fitness/sum(fitness);%100 个个体对应 100 人随机序列相当于对 fitness 归一化到 0~1 之间
distance_min=zeros(C+1,1); %%各次迭代的最大的种群的距离
while C>0
    fprintf('迭代第%d 次\n',C_old+1-C);%C 表示迭代次数

    %%选择操作
```

```

nn=0;
for i=1:size(popm,1)%从 1 到种群个数
len_1(i,1)=myLength(popm(i,:));
jc=rand*0.05;
for j=1:size(popm,1)
    if fitness(j,1)>=jc
        nn=nn+1;
        popm_sel(nn,:)=popm(j,:);%popm_sel 用于存放被选中的个体
        break;
    end
end
end

%%每次选择都保存最优的种群
popm_sel=popm_sel(1:nn,:);
[len_m len_index]=max(len_1);
popm_sel=[popm_sel;popm(len_index,:)];
nnper=randperm(nn);
A=popm_sel(nnper(1),:);
B=popm_sel(nnper(2),:);
for i=1:nn*Pc
    [A,B]=cross(A,B);
    popm_sel(nnper(1),:)=A;
    popm_sel(nnper(2),:)=B;
end

%%变异操作
for i=1:nn %种群中的每个个体都要对其进行随机变异操作
    pick=rand;%产生随机小数
    while pick==0
        pick=rand;
    end
    if pick<=Pmutation
        popm_sel(i,:)=Mutation(popm_sel(i,:));%进行变异，再放回
    end
end

%%求适应度函数
NN=size(popm_sel,1);%NN 为再生个体群的个数
len=zeros(NN,1);
for i=1:NN
    len(i,1)=myLength(popm_sel(i,:));
end
maxlen=max(len);

```

```

minlen=min(len);
distance_min(C+1,1)=maxlen;
fitness=fit(len,m,maxlen,minlen);
rr=find(len==maxlen);

fprintf('当前种群规模: %d\n',size(popm_sel,1));
fprintf('当前种群满足条件方程个数最多为: maxlen=%d\n',maxlen);
R=popm_sel(rr(1,1),:);%R 为该种群中最优个体
fprintf('本次迭代最优个体为: \n')
for i=1:N
    fprintf('%d ',R(i));%打印最优个体
end
fprintf('\n');
popm=[];%种群置空
popm=popm_sel;%新一代种群放回原种群
C=C-1;
end

fprintf('\n\n 最优值为: %d',maxlen);
fprintf('\n 最优个体为: ');
for i=1:N
    fprintf('%d ',R(i));
end
fprintf('\n');

```

方法二：混合粒子群算法 matlab 程序（部分）

```

%% 参数设置
Nbit=128; %个体编码位数
nMax=20; %进化次数
indiNumber=200; %个体数目
individual=zeros(indiNumber,Nbit);
%^初始化粒子位置
for i=1:indiNumber
    individual(i,:)=round(rand(1,Nbit));
end

%% 计算种群适应度
%indiFit=fitness(individual,cityCoor,cityDist);
for i=1:indiNumber
    indiFit(i)=myLength(individual(i,:));
end

[value,index]=max(indiFit);
tourPbest=individual; %当前个体最优

```



```

tourGbest=individual(index,:); %当前全局最优
%recordPbest=inf*ones(1,indiNumber); %个体最优记录
recordPbest=zeros(1,indiNumber); %个体最优记录
recordGbest=indiFit(index); %群体最优记录
xnew1=individual; %保存一下当前种群

%% 循环寻找最优值
L_best=zeros(1,nMax); %保存最佳适应度
for N=1:nMax
    %计算适应度值
    %indiFit=fitness(individual,cityCoor,cityDist);
    fprintf('第%d 次迭代中...\n',N); %打印迭代次数

    for i=1:indiNumber
        indiFit(i)=myLength(individual(i,:));
    end

    %更新当前最优和历史最优
    for i=1:indiNumber
        if indiFit(i)>recordPbest(i)
            recordPbest(i)=indiFit(i);
            tourPbest(i,:)=individual(i,:);
        end
        if indiFit(i)>recordGbest
            recordGbest=indiFit(i);
            tourGbest=individual(i,:);
        end
    end
end

[value,index]=max(recordPbest);
recordGbest(N)=recordPbest(index);

%% 交叉操作
for i=1:indiNumber
    % 与个体最优进行交叉
    c1=unidrnd(Nbit-1); %产生交叉位
    c2=unidrnd(Nbit-1); %产生交叉位
    while c1==c2
        c1=round(rand*(Nbit-2))+1;
        c2=round(rand*(Nbit-2))+1;
    end
    chb1=min(c1,c2);
    chb2=max(c1,c2);
    cros=tourPbest(i,chb1:chb2);

```

```

ncros=size(cros,2);          %取列数

%插入交叉区域
xnew1(i,Nbit-ncros+1:Nbit)=cros;
%新路径长度变短则接受
dist=0;
dist=myLength(xnew1(i,:));
%dist=dist+cityDist(xnew1(i,1),xnew1(i,n));
if indiFit(i)<dist
    individual(i,:)=xnew1(i,:);
end

% 与全体最优进行交叉
c1=round(rand*(Nbit-2))+1; %产生交叉位
c2=round(rand*(Nbit-2))+1; %产生交叉位
while c1==c2
    c1=round(rand*(Nbit-2))+1;
    c2=round(rand*(Nbit-2))+1;
end
chb1=min(c1,c2);
chb2=max(c1,c2);
cros=tourGbest(chb1:chb2);
ncros=size(cros,2);
%插入交叉区域
xnew1(i,Nbit-ncros+1:Nbit)=cros;
%新路径长度变短则接受
dist=0;
dist=myLength(xnew1(i,:));
%dist=dist+cityDist(xnew1(i,1),xnew1(i,n));
if indiFit(i)<dist
    individual(i,:)=xnew1(i,:);
end

%% 变异操作
c1=round(rand*(Nbit-1))+1; %产生变异位
c2=round(rand*(Nbit-1))+1; %产生变异位
while c1==c2
    c1=round(rand*(Nbit-2))+1;
    c2=round(rand*(Nbit-2))+1;
end
temp=xnew1(i,c1);          %采取交换操作
xnew1(i,c1)=xnew1(i,c2);
xnew1(i,c2)=temp;

```

```

        %新路径长度变短则接受
        dist=0;
        dist=myLength(xnew1(i,:));
        if indiFit(i)<dist
            individual(i,:)=xnew1(i,:);
        end
    end

    [value,index]=max(indiFit);
    L_best(N)=indiFit(index);
    tourGbest=individual(index,:);

    fprintf('本次迭代最大适应度值: %d\n',L_best(N));
    fprintf('本次迭代最佳粒子: ');
    flg=0;
    for i=1:Nbit
        if(flg==64)
            fprintf('\n');
        end
        flg=flg+1;
        fprintf('%d',tourGbest(i));
    end
    fprintf('\n');

end

```

方法三：分为 8 组时使用的 C 程序（部分）

```

#include "stdio.h"
int number(int y1,int y2,int y3,int y4,int y5,int y6,int y7,int y8)
{
    int x0, x1, x2, x3, x4, x5, x6, x7, x8, x9, x10, x11, x12, x13, x14, x15, x16, x17, x18, x19, x20, x21,
    x22, x23, x24, x25, x26, x27, x28, x29, x30, x31, x32, x33, x34, x35, x36, x37, x38, x39, x40, x41, x42,
    x43, x44, x45, x46, x47, x48, x49, x50, x51, x52, x53, x54, x55, x56, x57, x58, x59, x60, x61, x62, x63,
    x64, x65, x66, x67, x68, x69, x70, x71, x72, x73, x74, x75, x76, x77, x78, x79, x80, x81, x82, x83, x84,
    x85, x86, x87, x88, x89, x90, x91, x92, x93, x94, x95, x96, x97, x98, x99, x100, x101, x102, x103,
    x104, x105, x106, x107, x108, x109, x110, x111, x112, x113, x114, x115, x116, x117, x118, x119, x120,
    x121, x122, x123, x124, x125, x126, x127;

    x0=x1=x2=x3=x4=x5=x6=x7=x8=x9=x10=x11=x12=x13=x14=x15=y1;
    x16=x17=x18=x19=x20=x21=x22=x23=x24=x25=x26=x27=x28=x29=x30=x31=y2;
    x32=x33=x34=x35=x36=x37=x38=x39=x40=x41=x42=x43=x44=x45=x46=x47=y3;
    x48=x49=x50=x51=x52=x53=x54=x55=x56=x57=x58=x59=x60=x61=x62=x63=y4;
    x64=x65=x66=x67=x68=x69=x70=x71=x72=x73=x74=x75=x76=x77=x78=x79=y5;
    x80=x81=x82=x83=x84=x85=x86=x87=x88=x89=x90=x91=x92=x93=x94=x95=y6;

```

```

x96=x97=x98=x99=x100=x101=x102=x103=x104=x105=x106=x107=x108=x109=x110=x111=y7;
x112=x113=x114=x115=x116=x117=x118=x119=x120=x121=x122=x123=x124=x125=x126=x127
=y8;
int str[257];
for(int i=1;i<=256;i++)
    str[i]=0;
    .....(省略处为布尔方程组)
int num=0;
static int max_num=0;
for(int k=1;k<=256;k++)
    {
        if(str[k]==0||str[k]%2==0)
            num++;
    }
    printf("%d    ",num);
    if(num>max_num)
        max_num=num;
return max_num;
}
int main()
{
    int y1,y2,y3,y4,y5,y6,y7,y8;
    y1=y2=y3=y4=y5=y6=y7=y8=0;
    int max_num=0;
    for(y1=0;y1<=1;y1++)
        for(y2=0;y2<=1;y2++)
            for(y3=0;y3<=1;y3++)
                for(y4=0;y4<=1;y4++)
                    for(y5=0;y5<=1;y5++)
                        for(y6=0;y6<=1;y6++)
                            for(y7=0;y7<=1;y7++)
                                for(y8=0;y8<=1;y8++)
                                    {
                                        max_num = number(y1,y2,y3,y4,y5,y6,y7,y8);
                                        static int flag = 0;
                                        if(max_num==147&&flag==0)
                                        {
                                            flag=1;
                                        }
                                    }
    printf("\n 值为 0 的方程个数: max_num = %d\n",max_num);
    return 0;
}

```