队伍编号	MC2305018
题号	A

基于 QUBO 模型的信用评分卡组合优化

摘 要

本文基于银行信用评分卡的组合优化背景与题中提供的信用卡典型数据，基于 QUBO 模型研究了信用评分卡组合优化的问题。首先，针对不同问题分别构建以最大化最终收入为目标的组合优化模型。其次，根据不同目标函数形式，通过换元降幂等数学技巧进行变换，转化为 QUBO 形式，并运用量子退火算法求解出使得最终收入最大时的信用评分卡组合。最后，本文运用简单随机抽样法、贪心算法、遗传算法对量子退火算法的结果进行对比检验分析，凸显量子退火算法优越性并总结不同算法优劣及适用场景。

针对问题一：首先，将问题转化为从 1000 张具有唯一阈值的信用评分卡中选择一张的形式。其次，构建以最大化最终收入为目标的组合优化模型，并将其转化为 QUBO 形式，并运用量子退火算法进行求解。计算得出，**当选择第 49 张信用评分卡的第 1 个阈值时，最终收入最大，为 61172.0，对应通过率和坏账率分别为 0.82 和 0.005。**最后，通过简单随机抽样进行结果检验，确保所求信用评分卡阈值组合可以使最终总收入最大。

针对问题二：首先，构建以最大化最终收入为目标的组合优化模型，由于目标函数最高次幂大于 2，运用两次换元降幂的技巧对模型进行处理，最终将模型转化为 QUBO 形式，并运用量子退火算法进行求解。计算得出，**最佳信用评分卡组合为：第一张信用评分卡的第 8 个阈值，第二张信用评分卡的第 1 个阈值，第三张信用评分卡的第 2 个阈值，最大收入为 27914.82。**最后，通过贪心算法对结果进行检验，结果表明量子退火算法结果优于贪心算法，确保了 QUBO 模型结果可信度。

针对问题三：首先，构建以最大化最终收入为目标的组合优化模型，由于目标函数最高次幂大于 2，运用两次换元降幂的技巧对模型进行处理，最终将模型转化为 QUBO 形式，由于问题复杂度过高，本文借助 Kaggle 平台云服务，运用量子退火算法进行求解。计算得出，**最佳信用评分卡组合为：第 8 张信用评分卡的第 2 个阈值，第 33 张信用评分卡的第 6 个阈值，第 49 张信用评分卡的第 3 个阈值，最大收入为 43880.97。**最后，通过贪心算法和遗传算法对结果进行检验，结果表明量子退火算法效果最佳、遗传算法次之、贪心算法较差，有力确保了 QUBO 模型结果可信度。另外本文还总结了三种算法的优劣及使用场景。

关键词：信用评分卡组合优化、QUBO 模型、量子退火算法、贪心算法、遗传算法

目录

1. 问题重述

1.1 问题背景

当前，全球科技创新进入空前密集活跃期，量子科技作为新一轮科技革命和产业变革的前沿领域得到快速发展。2020 年 10 月，习近平总书记在中央政治局第二十四次集体学习中强调，要深刻认识推进量子科技发展重大意义，加强量子科技发展战略谋划和系统布局。相较于现有经典二进制算法，量子计算具有更强大的计算能力，能够更快地解决一些问题。例如，在金融投资领域中，确定投资组合相当于等式约束下的二次规划问题，常见形式有在固定预期收益下选择最佳投资组合，或者在固定风险下确定预期收益最大的投资组合等[1]。其中，QUBO（Quadratic Unconstrained Binary Optimization）模型是解决该类问题的常见方法，该模型能够运行在量子计算机硬件上，通过量子计算机进行毫秒级的加速求解，能够大幅缩短模拟和优化的时间，并有效提高模型准确性，在实际金融投资决策中具有重要意义。

目前，关于量子计算应用于投资组合问题中的研究相对不多。投资组合优化问题属于 NP（Non-deterministic Polynomial）难题[2]，量子优化算法能够有效解决该问题。量子优化算法的核心是绝热量子计算，而量子退火则是实现绝热量子计算的物理过程。实际上，量子退火过程尚难满足绝热量子计算所需条件，因此量子退火是绝热计算的一种近似实现[3]。Rosenberg 等(2016)在 D-Wave 量子退火机上求解投资组合问题，结果表明具有较高成功率[4]。Venturelli 等(2018)基于现代投资组合理论原则，基于实际金融统计数据生成投资组合优化问题参数化样本并在 D-Wave2000Q 量子退火机上进行模拟，研究了均值—方差投资组合优化问题[5]。Egger 等(2020)从凸优化、组合优化、混合二进制优化问题等方面总结了相关量子优化算法，并给出了在投资管理、投资组合优化、拍卖等领域的应用实例[6]。

综上，已有相关学者将量子计算应用于优化领域，但具体落脚在金融投资领域的研究还相对较少，求解金融投资领域的量子算法仍存在较大拓展空间，这就为本文研究提供了契机。

1.2 问题提出

本研究立足金融投资领域中常见问题——信用评分卡组合优化，遵循固定利息收益率下预期收益最大化的原则，聚焦解决组合优化问题的关键方法——QUBO 模型，并应用量子计算的思想，主要解决以下 3 个问题：

问题一：从附件 1 中的 100 个信用评分卡中找出 1 张及其对应阈值，使最终收入最多，对该问题进行建模，并将该模型转为 QUBO 形式来求解。

问题二：基于题中示例给出的信用评分卡 1、信用评分卡 2、信用评分卡 3 这三种规则，分别设置对应阈值使最终收入最多，对该问题进行建模，并将模型转为 QUBO 形式来求解。

问题三：从附件 1 中的 100 个信用评分卡中任选取 3 种信用评分卡，分别设置对应阈值使最终收入最多，对该问题进行建模，并将模型转为 QUBO 形式来求解。

2. 问题分析

本题可以看作一个二次无约束二值优化问题，问题的关键点在于将问题的公式转化为 QUBO（Quadratic Unconstrained Binary Optimization）形式来求解，即目标函数应当是一个最高次为二次，决策变量为 0-1 型变量的多项式公式。问题一、问题二、问题三依次要求从 100 个信用评分卡中找出特定数量的信用评分卡并确定其阈值，使得最终总收入最大。本文通过问题转化，根据问题信息设置决策变量等方式构建 QUBO 公式，运用量子退火算法等对问题进行求解检验。

2.1 问题一分析

对于问题一，题目要求从 100 个信用评分卡中找出 1 张及其对应阈值，使得最终总收入最多。首先，将问题等价转化为从 1000 张信用评分卡（每一张信用评分卡只有一个阈值）中找出一张信用卡。其次，针对转化后的问题建立 QUBO 等式，并运用量子退火算法进行求解。最后，针对求解出的结果运用简单随机抽样进行对比检验。

2.2 问题二分析

对于问题二，要求从题中所给信用评分卡 1、信用评分卡 2、信用评分卡 3 这三种规则，分别设置对应阈值，使得最终总收入最多。首先对问题进行数学刻画，将问题二中每一张信用卡的某一个阈值是否会被采样用二元决策变量x_{li}表示，其中l表示所选择的信用卡序号，$l \in [1,3]$，其中i表示对应信用卡所选择的阈值，$i \in [1,10]$。其次，由于目标函数中最高次幂大于 2，对目标函数进行两次换元降幂处理后建立 QUBO 等式，并运用量子退火算法进行求解。最后，针对求解出的结果运用贪心算法进行对比分析。

2.3 问题三分析

对于问题三，题目要求从 100 张信用评分卡中分别选择三张信用评分卡及其对应阈值，使得最终总收入最多。首先对问题进行数学刻画，将问题三中每一张信用卡的某一个阈值是否会被采样用二元决策变量x_{li}表示，其中l表示所选择的信用卡序号，$l \in [1,100]$，其中i表示对应信用卡所选择的阈值，$i \in [1,10]$。其次，由于目标函数中最高次

幂仍大于 2，对目标函数进行两次换元处理后建立 QUBO 等式，并运用量子退火算法进行求解。最后，针对求解出的结果运用贪心算法、遗传算法进行对比分析。

综上，本文问题技术路线如下图所示：

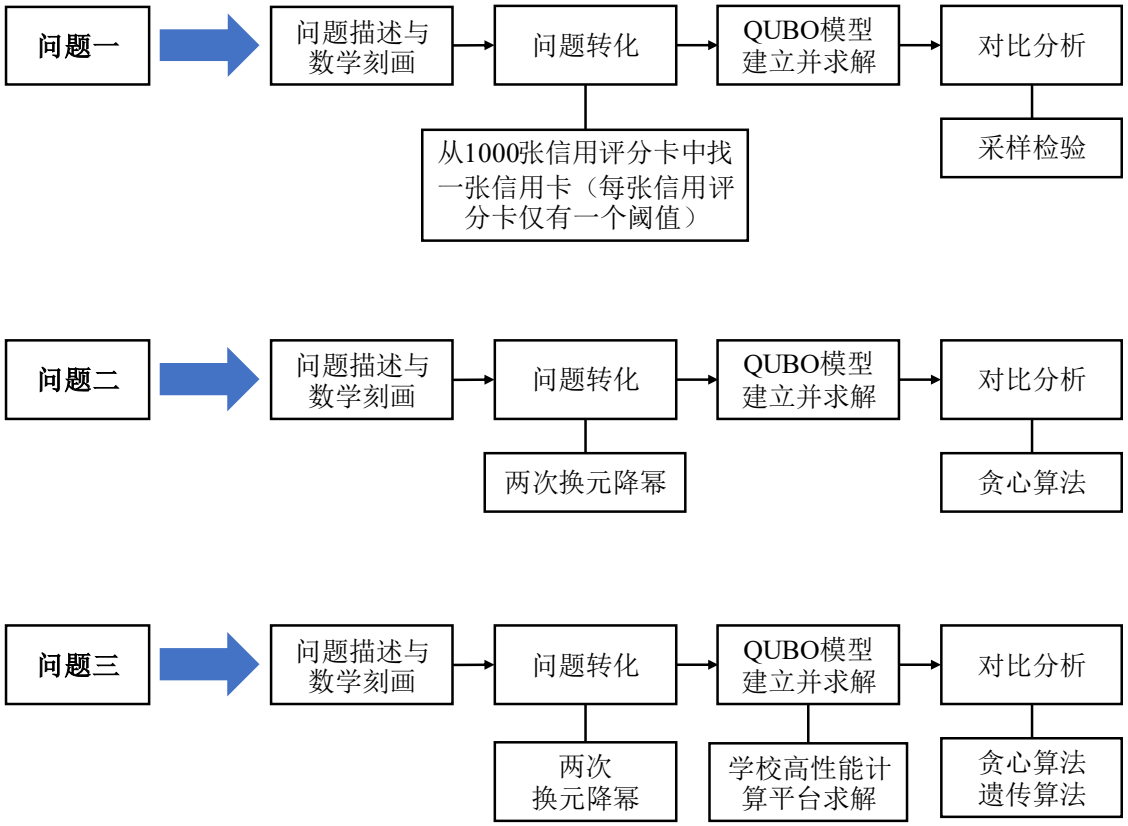

图 1 技术路线图

3. 模型假设

（1）赛题提供的数据来源于权威的原始数据，并且这些数据都是真实可靠的。

（2）每个问题之间的信息相互独立，没有必然的联系。

（3）每个问题的最终收入、贷款利息收入、坏账损失的计算方式依据赛题信息进行计算，不考虑赛题之外的信息，不考虑特殊情况的发生。

（4）每一个问题的贷款资金为 1000000 元，银行贷款利息收入率为 8%。

4. 模型建立与求解

4.1 问题一

4.1.1 信用评分卡组合优化问题

（1）组合优化问题

组合优化（Combinatorial Optimization）问题是在一个有限的对象集中找出最优对象的一类问题。该类问题的特征是可行解的集是离散或者可以简化到离散的，目标是找到最优解，例如背包问题、旅行商问题和生产调度问题等。组合优化中的算法包括贪心算法、动态规划、分支界定、线性规划等等。在实际应用中，通常需要根据具体问题的特点选择合适的算法，以求得最优解或近似最优解。该类模型包括目标函数、约束条件和定义域，具体结构如图 2 所示。

$$\begin{aligned} \text{目标函数} &\rightarrow \quad \min f(x) \\ \text{约束条件} &\rightarrow \quad s.t.\ g(x) \geq 0 \\ \text{定义域} &\rightarrow \quad x \in D \end{aligned}$$

图 2 组合优化模型

贪心算法和遗传算法都是常用的优化算法，它们在求解最优化问题方面具有独特优势。贪心算法的核心思想是通过每一步贪心选择来逐步简化问题规模，在许多情况下能够得到足够好的解，同时具有高效性和简洁性的优点。贪心算法通常适用于那些具有“贪心选择性质”的问题，即通过选择当前看起来最好的选项来获得全局最优解。而遗传算法则是基于生物进化原理的优化算法，通过对候选解进行选择、交叉和变异等操作，模拟自然选择和遗传机制，逐步优化解空间中的个体，直到达到预设的终止条件。遗传算法具有并行性强、全局寻优能力强等特点，尤其适用于复杂问题和多目标优化问题。在求解实际问题时，常常需要结合贪心算法和遗传算法等多种优化算法进行综合运用，以达到更好的优化效果。

为凸显 QUBO 模型优势，本文另采用贪心算法、遗传算法进行对比分析。第二问决策变量较少，选择贪心算法和 QUBO 模型分别求解并对比分析；第三问决策变量较多，选择贪心算法、遗传算法和 QUBO 模型分别求解并对比分析。

（2）信用评分卡组合优化问题

在银行信用卡或相关贷款等业务中，对客户授信前，需先通过相关审核规则评定客户信用等级，评定通过后的客户方可获得信用或贷款资格。实际上，规则审核过程是经过一重或多重组合规则后对客户进行打分，这些规则被称为信用评分卡，每个信用评分卡又分为多种阈值设置（但只有一个阈值生效），因此不同信用评分卡在不同阈值下对应不同通过率和坏账率。由于实际中的复杂性，常常需要采用选择多个不同信用评分卡

进行组合来实现最佳的风险控制策略。

下面，通过一个简单算例来介绍信用评分卡组合优化中的关键点和计算逻辑，以题中所给的 3 个信用评分卡为例，假设信用评分卡 1 的阈值设置为 9，信用评分卡 2 的阈值设置为 8，信用评分卡 3 的阈值设置为 7，具体见表 1。

表 1 题中信用评分卡示例（算例设置阈值均已加粗显示）

信用评分卡 1			信用评分卡 2			信用评分卡 3		
阈值	通过率	坏账率	阈值	通过率	坏账率	阈值	通过率	坏账率
1	5%	0.50%	1	5%	0.50%	1	5%	0.50%
2	10%	1.00%	2	10%	1.00%	2	10%	1.00%
3	25%	1.50%	3	25%	1.50%	3	20%	1.70%
4	30%	2.00%	4	30%	2.00%	4	33%	2.00%
5	40%	2.50%	5	45%	2.50%	5	40%	2.70%
6	50%	3.00%	6	50%	2.70%	6	52%	3.00%
7	60%	3.50%	7	65%	3.50%	**7**	**62%**	**3.70%**
8	70%	4.00%	**8**	**70%**	**4.00%**	8	73%	4.00%
9	**80%**	**4.50%**	9	82%	4.70%	9	82%	4.70%
10	93%	5.00%	10	90%	5.00%	10	95%	5.00%

①总通过率：三种信用评分卡对应通过率乘积，即：

$$0.8\times0.7\times0.62=0.3472$$

②总坏账率：三种信用评分卡对应坏账率平均值，即：

$$1/3\times（0.045+0.04+0.037）=0.0407$$

③贷款利息收入：假设贷款资金为 1000000 元，贷款利息收入率为 8%，即：

$$\text{贷款利息收入} = \text{贷款资金} \times \text{利息收入率} \times \text{总通过率} \times \left(1 - \text{总坏账率}\right) \quad (1)$$

$$=1000000\times0.08\times（0.8\times0.7\times0.62）\times（1-1/3\times（0.045+0.04+0.037））=26645.5168$$

④坏账损失：由坏账带来，即：

$$\text{坏账损失} = \text{贷款资金} \times \text{总通过率} \times \text{总坏账率} \quad (2)$$

$$=1000000\times（0.8\times0.7\times0.62）\times1/3\times（0.045+0.04+0.037）=14119.46667$$

⑤最终收入：贷款利息收入与坏账损失的差值，即：

$$\text{最终收入} = \text{贷款利息收入} - \text{坏账损失} \quad (3)$$

$$=26645.5168-14119.46667=12526.05013$$

4.1.2 QUBO 模型及量子计算

（1）QUBO 模型

QUBO 模型是指二次无约束二值优化（Quadratic Unconstrained Binary Optimization）模型，它是一种用于解决组合优化问题的数学模型。在 QUBO 模型中，需要将问题转化为一个决策变量为二值变量，目标函数是一个二次函数形式优化模型，如式(4)所示。

$$y = x^T Q x \quad (4)$$

其中，Q为 QUBO 矩阵，x为二进制变量组成的向量，每个变量取值均为{0,1}，QUBO 目标为找到使得y最小或最大的x，Q 矩阵的形式有 2 种：

①对称形式

$$q'_{ij} = \frac{(q_{ij} + q_{ji})}{2} \tag{5}$$

②上三角形式

$$\begin{aligned} q'_{ij} &= (q_{ij} + q_{ji}), i \le j \\ q'_{ij} &= 0, i > j \end{aligned} \tag{6}$$

当添加约束条件后：

$$\begin{aligned} &\min f(x) \\ &s.t.\ Ax = b \\ &f'(x) = f(x) + P(Ax - b)^T (Ax - b) \end{aligned} \tag{7}$$

其中，P 表示正标量惩罚项。但随着问题规模的增加，利用传统算法求解组合优化问题，求解时间将会大大增加，但利用 QUBO 模型可以运行在量子计算机硬件上，通过量子计算机进行毫秒级的加速求解，能够高效求解组合优化问题。

（2）量子计算

量子计算是当今的前沿研究领域，是计算机科学和量子物理学之间双向交流的成果。一方面，计算机科学使用量子力学来定义新的计算模型，另一方面，信息和复杂性理论的语言使人们能够更清晰地理解量子力学的某些方面。下面，本文简要介绍一种由量子系统行为所建议的经典启发式算法——量子退火算法。

在模拟退火中，给定优化问题的解空间由依赖于温度的随机游动访问，成本函数表明解空间的势能分布，热跃迁避免探索陷入局部最小值。然后，适当安排降温（退火）可以稳定围绕潜在分布的最小值行走，向较差解移动的概率趋于零，从而保证找到或接近全局最优解。与模拟退火中依赖于温度的热跃迁不同，量子退火中使用量子跳跃来探索解空间，这些跃迁由量子隧穿效应驱动，能够更快地穿过局域极值点旁的势垒，比传统热力学探索局部极小值更有效[7]，具体区别如图 3 所示。

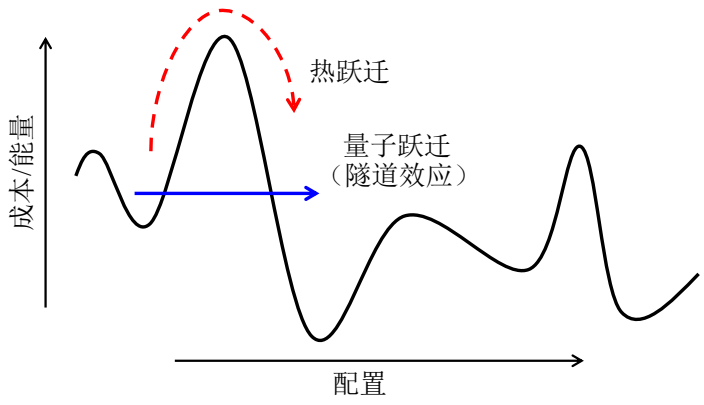

图 3 模拟退火中允许探索解决方案的热跃迁在量子退火中被量子跃迁所取代

量子跃迁由式(8)形式的哈密顿量的基态（最小能量状态）相关联的随机过程q_v的行为所暗示：

$$H_v = -\frac{v^2}{2}\frac{\partial^2}{\partial x^2} + V(x) \tag{8}$$

其中，势函数V编码要最小化的成本函数。

对于任何固定的v，一旦给定H_v的基态ψ_v，就可以通过基态变换构建随机过程q_v：设$\psi_v \in L^2(R, dx)$为哈密顿量(8)的基态。在对势函数V的相当一般的假设下，基态ψ_v可以严格取为正值，进行变换：

$$U{:}\psi \to U\psi = \frac{\psi}{\psi_v} \tag{9}$$

或进行基态变换，从$L^2(R, dx)$到$L^2(R, \psi_v^2 dx)$定义明确且单一。在这种变换下，H_v的形式为：

$$H_v = UH_vU^{-1} = -vL_v + E_v \tag{10}$$

其中L_v见式(11)，L_v在实线上具有扩散过程q_v生成器的形式，其漂移量见式(12)：

$$L_v = \frac{1}{2}v\frac{d^2}{dx^2} + b_v \tag{11}$$

$$b_v(x) = \frac{1}{2}\frac{d}{dx}\ln\left(\psi_v^2(x)\right) \tag{12}$$

稳态基态过程q_v样本路径的行为以围绕稳定配置的长时间逗留为特征，即$V(x)$的最小值被罕见的大波动打断，这些波动将q_v从一个最小值带到另一个，因此允许q_v从局部最小值“隧道化”到$V(x)$的全局最小值（如图 3 所示）。q_v的扩散行为由(8)中的拉普拉斯项决定，即动能由参数v控制。

4.1.3 信用评分卡组合优化问题 QUBO 模型构建及量子退火算法求解

（1）QUBO 模型构建

在 100 个信用评分卡中找出 1 张及其对应阈值，每一张卡有 10 个阈值，使最终收入最多。可以转化为 1000 个信用卡中找出一张具有明确阈值的信用卡，使其最终收入最多。每一张信用卡是否采用通过二元值x_i表示，其中$i \in [1,1000]$。每一张信用卡i的通过率为A_i，坏账率为B_i，则问题一预期最终收入为：

$$\begin{aligned} &80000 \cdot \prod_{i=1}^{1000} A_i^{x_i} \cdot \left(1 - \sum_{i=1}^{1000}(B_i \cdot x_i)\right) - 1000000 \cdot \prod_{i=1}^{1000} A_i^{x_i} \cdot \sum_{i=1}^{1000}(B_i \cdot x_i) \\ &= \prod_{i=1}^{1000} A_i^{x_i} \cdot \left(80000 - 1080000 \cdot \sum_{i=1}^{1000}(B_i \cdot x_i)\right) \end{aligned} \tag{13}$$

因为明确只能使用一张确定阈值的信用卡，所以原式可以转化为：

$$\sum_{i=1}^{1000}(A_i \cdot x_i) \cdot \left(80000 - 1080000 \cdot \sum_{i=1}^{1000}(B_i \cdot x_i)\right) \tag{14}$$

$$= -1080000 \cdot \sum_{i=1}^{1000}(B_i \cdot x_i) \cdot \sum_{i=1}^{1000}(A_i \cdot x_i) + 80000 \cdot \sum_{i=1}^{1000}(A_i \cdot x_i)$$

对式(14)取负后转化为求最小，则目标函数为：

$$\begin{aligned} &-\left(-1080000 \cdot \sum_{i=1}^{1000}(B_i \cdot x_i) \cdot \sum_{i=1}^{1000}(A_i \cdot x_i) + 80000 \cdot \sum_{i=1}^{1000}(A_i \cdot x_i)\right) \\ &= 1080000 \cdot \sum_{i=1}^{1000}(B_i \cdot x_i) \cdot \sum_{i=1}^{1000}(A_i \cdot x_i) - 80000 \cdot \sum_{i=1}^{1000}(A_i \cdot x_i) \end{aligned} \tag{15}$$

约束条件为：

$$\sum_{i=1}^{1000} x_i = 1 \tag{16}$$

约束条件等价为：

$$\left(1 - \sum_{i=1}^{1000} x_i\right)^2 = 0 \tag{17}$$

转化为 QUBO 形式为：

$$\begin{cases} \boldsymbol{QUBO}: \min 1080000 \cdot \sum_{i=1}^{1000}(B_i \cdot x_i) \cdot \sum_{i=1}^{1000}(A_i \cdot x_i) - 80000 \cdot \sum_{i=1}^{1000}(A_i \cdot x_i) \\ s.t. \left(1 - \sum_{i=1}^{1000} x_i\right)^2 = 0 \end{cases} \tag{18}$$

通过 Python 的 Pyqubo 程序得出[8]：当选择第 49 张信用卡的第 1 个阈值时，收入最大，最大收入为 61172.0，对应通过率和坏账率分别为 0.82 和 0.005。

4.1.4 量子退火算法与简单随机抽样对比分析

为检验运用量子退火算法求解出的信用评分卡阈值策略，问题一基于简单随机抽样(Simple random sampling)对结果进行检验。对比简单随机抽样与量子退火算法获得的信用卡策略，结果如下表所示：

表 2 基于简单随机抽样的结果检验

对比方法	信用卡阈值选择策略	最终收入
量子退火算法	第 49 张信用卡的第 1 个阈值	61172.0
简单随机抽样	第 49 张信用卡的第 3 个阈值	58820.0
	第 11 张信用卡的第 1 个阈值	54330.0
	第 23 张信用卡的第 8 个阈值	10764.0
	第 94 张信用卡的第 6 个阈值	28751.2

通过表 2 可知，运用量子退火算法求解的信用评分卡阈值选择策略最佳，可以得知：针对问题一，选择第 49 张信用卡的第 1 个阈值时，收入最大，最大收入为 61172.0。

4.2 问题二

4.2.1 信用评分卡组合优化问题 QUBO 模型构建及量子退火算法求解

在三张具有 10 个阈值的信用评分卡中分别找出一个阈值，确保最终收入最多。假设每一张卡是否选择对应的阈值为二元值x，三张卡的二元值分别为x_{1i}、x_{2j}和x_{3z}，i、j、$z \in [1,10]$。第一张信用卡x_{1i}的通过率为A_{1i}，坏账率为B_{1i}。第二张信用卡x_{2j}的通过率为A_{2j}，坏账率为B_{2j}。第三张信用卡x_{3z}的通过率为A_{3z}，坏账率为B_{3z}。则问题二预期最终收入为：

$$\begin{aligned}
&80000 \cdot \left(\sum_{i=1}^{10} A_{1i}x_{1i} \cdot \sum_{j=1}^{10} A_{2j}x_{2j} \cdot \sum_{z=1}^{10} A_{3z}x_{3z}\right) \cdot \left(1 - \frac{1}{3}\left(\sum_{i=1}^{10} B_{1i}x_{1i} + \sum_{j=1}^{10} B_{2j}x_{2j} + \sum_{z=1}^{10} B_{3z}x_{3z}\right)\right) \\
&\quad - 1000000 \cdot \left(\sum_{i=1}^{10} A_{1i}x_{1i} \cdot \sum_{j=1}^{10} A_{2j}x_{2j} \cdot \sum_{z=1}^{10} A_{3z}x_{3z}\right) \\
&\quad \cdot \frac{1}{3}\left(\sum_{i=1}^{10} B_{1i}x_{1i} + \sum_{j=1}^{10} B_{2j}x_{2j} + \sum_{z=1}^{10} B_{3z}x_{3z}\right) \\
&= 80000 \cdot \sum_{i=1}^{10}\sum_{j=1}^{10}\sum_{z=1}^{10}\left(A_{1i}x_{1i} \cdot A_{2j}x_{2j} \cdot A_{3z}x_{3z}\right) \cdot \left(1 - \frac{1}{3}\sum_{i=1}^{10}\sum_{j=1}^{10}\sum_{z=1}^{10}\left(B_{1i}x_{1i} + B_{2j}x_{2j} + B_{3z}x_{3z}\right)\right) \\
&\quad - 1000000 \cdot \sum_{i=1}^{10}\sum_{j=1}^{10}\sum_{z=1}^{10}\left(A_{1i}x_{1i} \cdot A_{2j}x_{2j} \cdot A_{3z}x_{3z}\right) \\
&\quad \cdot \frac{1}{3}\left(\sum_{i=1}^{10} B_{1i}x_{1i} + \sum_{j=1}^{10} B_{2j}x_{2j} + \sum_{z=1}^{10} B_{3z}x_{3z}\right) \\
&= \sum_{i=1}^{10}\sum_{j=1}^{10}\sum_{z=1}^{10}\left(A_{1i}x_{1i} \cdot A_{2j}x_{2j} \cdot A_{3z}x_{3z}\right) \cdot \left(80000 - 360000\left(\sum_{i=1}^{10} B_{1i}x_{1i} + \sum_{j=1}^{10} B_{2j}x_{2j} + \sum_{z=1}^{10} B_{3z}x_{3z}\right)\right)
\end{aligned} \tag{19}$$

原始化简为：

$$\begin{aligned}
&80000\sum_{i=1}^{10}\sum_{j=1}^{10}\sum_{z=1}^{10}\left(A_{1i}x_{1i} \cdot A_{2j}x_{2j} \cdot A_{3z}x_{3z}\right) \\
&-360000\sum_{i=1}^{10}\sum_{j=1}^{10}\sum_{z=1}^{10}\left(A_{1i}x_{1i} \cdot A_{2j}x_{2j} \cdot A_{3z}x_{3z}\right) \cdot \left(\sum_{i=1}^{10} B_{1i}x_{1i} + \sum_{j=1}^{10} B_{2j}x_{2j} + \sum_{z=1}^{10} B_{3z}x_{3z}\right)
\end{aligned} \tag{20}$$

因为上式最高次幂大于 2，本文借鉴 Glover 等(2019)的约简技术思想，对高次幂进行换元处理，使其满足 QUBO 形式转化[9]。令$x_{1i} \cdot x_{2j} = y_{ij}$、$y_{ij} \cdot x_{3z} = t_{ijz}$，则原式有：

$$80000\sum_{i=1}^{10}\sum_{j=1}^{10}\sum_{z=1}^{10}\left(A_{1i}A_{2j}A_{3z}t_{ijz}\right)$$
$$-360000\sum_{i=1}^{10}\sum_{j=1}^{10}\sum_{z=1}^{10}\left(A_{1i}A_{2j}A_{3z}t_{ijz}\right)\cdot\left(\sum_{i=1}^{10}B_{1i}x_{1i}+\sum_{j=1}^{10}B_{2j}x_{2j}+\sum_{z=1}^{10}B_{3z}x_{3z}\right) \tag{21}$$

根据 Glover(2019)等的降幂思想，需要对目标函数中添加惩罚[9]，因此目标函数为：

$$-80000\sum_{i=1}^{10}\sum_{j=1}^{10}\sum_{z=1}^{10}\left(A_{1i}A_{2j}A_{3z}t_{ijz}\right)$$
$$+360000\sum_{i=1}^{10}\sum_{j=1}^{10}\sum_{z=1}^{10}\left(A_{1i}A_{2j}A_{3z}t_{ijz}\right)\cdot\left(\sum_{i=1}^{10}B_{1i}x_{1i}+\sum_{j=1}^{10}B_{2j}x_{2j}+\sum_{z=1}^{10}B_{3z}x_{3z}\right) \tag{22}$$
$$+\sum_{i=1}^{10}\sum_{j=1}^{10}\alpha_{ij}\left(x_{1i}\cdot x_{2j}-2y_{ij}\left(x_{1i}+x_{2j}\right)+3y_{ij}\right)+\sum_{i=1}^{10}\sum_{j=1}^{10}\sum_{z=1}^{10}\beta_{ijz}\left(y_{ij}\cdot x_{3z}-2t_{ijz}\left(y_{ij}+x_{3z}\right)+3t_{ijz}\right)$$

其中$\boldsymbol{\alpha}$和$\boldsymbol{\beta}$为惩罚因子向量。

针对约束条件，对于每张卡，对应的 10 个阈值之间存在一个约束条件，只能选择其中一个阈值，因此有：

$$\sum_{i=1}^{10}x_{1i}=1,\qquad\sum_{j=1}^{10}x_{2j}=1,\qquad\sum_{z=1}^{10}x_{3z}=1$$

该约束条件可以用一组二次项来表示：

$$\left(1-\sum_{i=1}^{10}x_{1i}\right)^2+\left(1-\sum_{j=1}^{10}x_{2j}\right)^2+\left(1-\sum_{z=1}^{10}x_{3z}\right)^2=0 \tag{23}$$

因此，针对问题二，转化为 QUBO 形式为：

$$\begin{cases}\boldsymbol{QUBO}:\ \min -80000\sum_{i=1}^{10}\sum_{j=1}^{10}\sum_{z=1}^{10}\left(A_{1i}A_{2j}A_{3z}t_{ijz}\right)\\ +360000\sum_{i=1}^{10}\sum_{j=1}^{10}\sum_{z=1}^{10}\left(A_{1i}A_{2j}A_{3z}t_{ijz}\right)\cdot\left(\sum_{i=1}^{10}B_{1i}x_{1i}+\sum_{j=1}^{10}B_{2j}x_{2j}+\sum_{z=1}^{10}B_{3z}x_{3z}\right)\\ +\sum_{i=1}^{10}\sum_{j=1}^{10}\alpha_{ij}\left(x_{1i}\cdot x_{2j}-2y_{ij}\left(x_{1i}+x_{2j}\right)+3y_{ij}\right)\\ +\sum_{i=1}^{10}\sum_{j=1}^{10}\sum_{z=1}^{10}\beta_{ijz}\left(y_{ij}\cdot x_{3z}-2t_{ijz}\left(y_{ij}+x_{3z}\right)+3t_{ijz}\right)\\ s.t.\ \left(1-\sum_{i=1}^{10}x_{1i}\right)^2+\left(1-\sum_{j=1}^{10}x_{2j}\right)^2+\left(1-\sum_{z=1}^{10}x_{3z}\right)^2=0\end{cases} \tag{24}$$

通过 Python 的 Pyqubo 程序得出最佳信用评分卡组合为[8]：第一张信用评分卡的第 8 个阈值，第二张信用评分卡的第 1 个阈值，第三张信用评分卡的第 2 个阈值，最大收入为 27914.820479999995。

4.2.2 量子退火算法与贪心算法对比分析

利用贪心算法进行求解时，认定使最终收入越大时，应该使通过率尽可能大，而坏账率尽可能小。所以算法将会从通过率最高的等级开始往下搜索，并且设定坏账率的最高限定值，直到找到第一个坏账率不超过限定值的阈值。在实际问题中，贪心算法通常会要求实验人员给定一些关键指标的限定值或限制条件。本题中选择的阈值不仅要使得总收入最大，还需要保证每个信用评分卡的坏账率不超过限定值。求解大规模问题时，贪心算法往往并不保证能得到全局最优解，但通常能在较短时间内得到一个足够好的解。根据样本中坏账率基本分布，分别将坏账率限定值定为 0、0.02、0.05 并观察运行结果差异，具体结果见表 3。

表 3 不同限定坏账率下贪心算法运行结果

限定坏账率	信用评分卡编号	阈值选择	最终收入
0	1	1	26037.96
	2	1	
	3	1	
0.02	1	3	26562.33
	2	1	
	3	2	
0.05	1	9	19964.62
	2	3	
	3	5	

根据结果可以看出，由于贪心算法在一开始就默认信用评分卡选定阈值必须倾向于高通过率，即使对坏账率不加要求也无法找到全局最优解。而对于限定坏账率的选择也并非越小（大）越好，解决实际问题时由于限定值选定的自主性，会使得最终的结果存在差异。

综上，贪心算法运行时间快，但寻优效果不如 QUBO 模型运用量子退火算法。

4.3 问题三

4.3.1 信用评分卡组合优化问题 QUBO 模型构建及量子退火算法求解

在 100 张具有 10 个阈值的信用评分卡中分别找出三张信用评分卡的三个阈值，使得最终收入最多。假设每一张卡是否选择对应的阈值为二元值x，三张卡的二元值分别为x_{oi}、x_{pj}和x_{qz}，其中o、p、q表示信用卡的序号，o、p、$q \in [1,100]$，且o、p、q各不相同。i、j、z表示每一张信用卡的阈值序号，i、j、$z \in [1,10]$。第一张信用评分卡x_{oi}的通过率为A_{oi}，坏账率为B_{oi}。第二张信用评分卡x_{pj}的通过率为A_{pj}，坏账率为B_{pj}。第三

张信用评分卡x_{qz}的通过率为A_{qz}，坏账率为B_{qz}。则问题三预期最终收入为：

$$
\begin{aligned}
&80000\left(\sum_{o=1}^{100}\sum_{i=1}^{10}A_{1i}x_{1i}\sum_{p=o+1}^{100}\sum_{j=1}^{10}A_{pj}x_{pj}\sum_{q=p+1}^{100}\sum_{z=1}^{10}A_{qz}x_{qz}\right)\\
&\quad\cdot\left(1-\frac{1}{3}\left(\sum_{o=1}^{100}\sum_{i=1}^{10}B_{oi}x_{1i}+\sum_{p=o+1}^{100}\sum_{j=1}^{10}B_{pj}x_{pj}+\sum_{q=p+1}^{100}\sum_{z=1}^{10}B_{qz}x_{qz}\right)\right)\\
&\quad-1000000\left(\sum_{i=1}^{10}A_{1i}x_{1i}\sum_{j=1}^{10}A_{2j}x_{2j}\sum_{z=1}^{10}A_{3z}x_{3z}\right)\\
&\quad\cdot\frac{1}{3}\left(\sum_{o=1}^{100}\sum_{i=1}^{10}B_{oi}x_{1i}+\sum_{p=o+1}^{100}\sum_{j=1}^{10}B_{pj}x_{pj}+\sum_{q=p+1}^{100}\sum_{z=1}^{10}B_{qz}x_{qz}\right)\\
&=80000\sum_{o=1}^{100}\sum_{p=o+1}^{100}\sum_{q=p+1}^{100}\sum_{i=1}^{10}\sum_{j=1}^{10}\sum_{z=1}^{10}\left(A_{oi}x_{oi}A_{pj}x_{pj}A_{qz}x_{qz}\right)\\
&\quad\cdot\left(1-\frac{1}{3}\sum_{o=1}^{100}\sum_{p=o+1}^{100}\sum_{q=p+1}^{100}\sum_{i=1}^{10}\sum_{j=1}^{10}\sum_{z=1}^{10}\left(B_{oi}x_{oi}+B_{pj}x_{pj}+B_{qz}x_{qz}\right)\right)\\
&\quad-1000000\sum_{o=1}^{100}\sum_{p=o+1}^{100}\sum_{q=p+1}^{100}\sum_{i=1}^{10}\sum_{j=1}^{10}\sum_{z=1}^{10}\left(A_{oi}x_{oi}A_{pj}x_{pj}A_{qz}x_{qz}\right)\\
&\quad\cdot\frac{1}{3}\sum_{o=1}^{100}\sum_{p=o+1}^{100}\sum_{q=p+1}^{100}\sum_{i=1}^{10}\sum_{j=1}^{10}\sum_{z=1}^{10}\left(B_{oi}x_{oi}+B_{pj}x_{pj}+B_{qz}x_{qz}\right)\\
&=\sum_{o=1}^{100}\sum_{p=o+1}^{100}\sum_{q=p+1}^{100}\sum_{i=1}^{10}\sum_{j=1}^{10}\sum_{z=1}^{10}\left(A_{oi}x_{oi}A_{pj}x_{pj}A_{qz}x_{qz}\right)\\
&\quad\cdot\left(80000-360000\sum_{o=1}^{100}\sum_{p=o+1}^{100}\sum_{q=p+1}^{100}\sum_{i=1}^{10}\sum_{j=1}^{10}\sum_{z=1}^{10}\left(B_{oi}x_{oi}+B_{pj}x_{pj}+B_{qz}x_{qz}\right)\right)
\end{aligned}
\tag{25}
$$

原始化简为：

$$
\begin{aligned}
&80000\sum_{o=1}^{100}\sum_{p=o+1}^{100}\sum_{q=p+1}^{100}\sum_{i=1}^{10}\sum_{j=1}^{10}\sum_{z=1}^{10}\left(A_{oi}x_{oi}A_{pj}x_{pj}A_{qz}x_{qz}\right)\\
&\quad-360000\sum_{o=1}^{100}\sum_{p=o+1}^{100}\sum_{q=p+1}^{100}\sum_{i=1}^{10}\sum_{j=1}^{10}\sum_{z=1}^{10}\left(B_{oi}x_{oi}+B_{pj}x_{pj}+B_{qz}x_{qz}\right)\\
&\quad\cdot\sum_{o=1}^{100}\sum_{p=o+1}^{100}\sum_{q=p+1}^{100}\sum_{i=1}^{10}\sum_{j=1}^{10}\sum_{z=1}^{10}\left(A_{oi}x_{oi}A_{pj}x_{pj}A_{qz}x_{qz}\right)
\end{aligned}
\tag{26}
$$

因为上式的最高次幂大于 2，借鉴 Glover(2019)等人的约简技术思想，对高次幂进

行换元处理，使其满足 QUBO 形式的转化[9]。令$x_{oi}\cdot x_{pj}=y_{opij}$、$y_{opij}\cdot x_{qz}=t_{opqijz}$，则原式有：

$$80000\sum_{o=1}^{100}\sum_{p=o+1}^{100}\sum_{q=p+1}^{100}\sum_{i=1}^{10}\sum_{j=1}^{10}\sum_{z=1}^{10}\left(A_{oi}A_{pj}A_{qz}t_{opqijz}\right)$$
$$-360000\sum_{o=1}^{100}\sum_{p=o+1}^{100}\sum_{q=p+1}^{100}\sum_{i=1}^{10}\sum_{j=1}^{10}\sum_{z=1}^{10}\left(A_{oi}A_{pj}A_{qz}t_{opqijz}\right) \tag{27}$$
$$\cdot\sum_{o=1}^{100}\sum_{p=o+1}^{100}\sum_{q=p+1}^{100}\sum_{i=1}^{10}\sum_{j=1}^{10}\sum_{z=1}^{10}\left(B_{oi}x_{oi}+B_{pj}x_{pj}+B_{qz}x_{qz}\right)$$

根据 Glover(2019)等的降幂思想，需要对目标函数中添加惩罚[9]，因此目标函数为：

$$-80000\sum_{o=1}^{100}\sum_{p=o+1}^{100}\sum_{q=p+1}^{100}\sum_{i=1}^{10}\sum_{j=1}^{10}\sum_{z=1}^{10}\left(A_{oi}A_{pj}A_{qz}t_{opqijz}\right)$$
$$+360000\sum_{o=1}^{100}\sum_{p=o+1}^{100}\sum_{q=p+1}^{100}\sum_{i=1}^{10}\sum_{j=1}^{10}\sum_{z=1}^{10}\left(A_{oi}A_{pj}A_{qz}t_{opqijz}\right)$$
$$\cdot\sum_{o=1}^{100}\sum_{p=o+1}^{100}\sum_{q=p+1}^{100}\sum_{i=1}^{10}\sum_{j=1}^{10}\sum_{z=1}^{10}\left(B_{oi}x_{oi}+B_{pj}x_{pj}+B_{qz}x_{qz}\right) \tag{28}$$
$$+\sum_{o=1}^{100}\sum_{p=o+1}^{100}\sum_{i=1}^{10}\sum_{j=1}^{10}\alpha_{opij}\left(x_{oi}x_{pj}-2y_{opij}\left(x_{oi}+x_{pj}\right)+3y_{opij}\right)$$
$$+\sum_{o=1}^{100}\sum_{p=o+1}^{100}\sum_{q=p+1}^{100}\sum_{i=1}^{10}\sum_{j=1}^{10}\sum_{z=1}^{10}\beta_{opqijz}\left(y_{opij}x_{qz}-2t_{opqijz}\left(y_{opij}+x_{qz}\right)+3t_{opqijz}\right)$$

其中$\boldsymbol{\alpha}$和$\boldsymbol{\beta}$为惩罚因子向量。

针对约束条件，对于每张卡，对应的 10 个阈值之间存在一个约束条件：只能选择其中一个阈值，因此有

$$\sum_{o=1}^{100}\sum_{i=1}^{10}x_{oi}=1,\qquad\sum_{p=o+1}^{100}\sum_{j=1}^{10}x_{pj}=1,\qquad\sum_{q=p+1}^{100}\sum_{z=1}^{10}x_{qz}=1$$

这个约束条件可以用一组二次项来表示：

$$\left(1-\sum_{o=1}^{100}\sum_{i=1}^{10}x_{oi}\right)^2+\left(1-\sum_{p=o+1}^{100}\sum_{j=1}^{10}x_{pj}\right)^2+\left(1-\sum_{q=p+1}^{100}\sum_{z=1}^{10}x_{qz}\right)^2=0 \tag{29}$$

因此，针对问题三，转化为 QUBO 形式为：

$$
\begin{cases}
\boldsymbol{QUBO}:\ \min -80000\sum_{o=1}^{100}\sum_{p=o+1}^{100}\sum_{q=p+1}^{100}\sum_{i=1}^{10}\sum_{j=1}^{10}\sum_{z=1}^{10}\left(A_{oi}A_{pj}A_{qz}t_{opqijz}\right) \\
\quad +360000\sum_{o=1}^{100}\sum_{p=o+1}^{100}\sum_{q=p+1}^{100}\sum_{i=1}^{10}\sum_{j=1}^{10}\sum_{z=1}^{10}\left(A_{oi}A_{pj}A_{qz}t_{opqijz}\right) \\
\quad \cdot\sum_{o=1}^{100}\sum_{p=o+1}^{100}\sum_{q=p+1}^{100}\sum_{i=1}^{10}\sum_{j=1}^{10}\sum_{z=1}^{10}\left(B_{oi}x_{oi}+B_{pj}x_{pj}+B_{qz}x_{qz}\right) \\
\quad +\sum_{o=1}^{100}\sum_{p=o+1}^{100}\sum_{i=1}^{10}\sum_{j=1}^{10}\alpha_{opij}\left(x_{oi}x_{pj}-2y_{opij}\left(x_{oi}+x_{pj}\right)+3y_{opij}\right) \\
+\sum_{o=1}^{100}\sum_{p=o+1}^{100}\sum_{q=p+1}^{100}\sum_{i=1}^{10}\sum_{j=1}^{10}\sum_{z=1}^{10}\beta_{opqijz}\left(y_{opij}x_{qz}-2t_{opqijz}\left(y_{opij}+x_{qz}\right)+3t_{opqijz}\right) \\
s.t.\ \left(1-\sum_{o=1}^{100}\sum_{i=1}^{10}x_{oi}\right)^2+\left(1-\sum_{p=o+1}^{100}\sum_{j=1}^{10}x_{pj}\right)^2+\left(1-\sum_{q=p+1}^{100}\sum_{z=1}^{10}x_{qz}\right)^2=0
\end{cases}
\tag{30}
$$

由于第三问的运算复杂度过大，本文借助学校高性能 Kaggle 云服务器，通过调用 Python 的 Pyqubo 包[8]，运用量子退火算法得出最终收入排名前十的结果，其中最佳信用评分卡组合为：第 8 张信用评分卡的第 2 个阈值，第 33 张信用评分卡的第 6 个阈值，第 49 张信用评分卡的第 3 个阈值，最大收入为 43880.97，详见表 4 和图 4。另外，Kaggle 提供了基于 CPU 的工作环境，用户可以在这个环境中运行和处理数据，进行建模和训练等操作。

表 4 问题三 QUBO 模型运行结果前 10 名（因后面图片展示便利，故采用逆序）

排名	银行卡及等级选择	最终收入
10	8(等级 2), 33(等级 4), 49(等级 3)	43434.86
9	8(等级 2), 19(等级 6), 33(等级 6)	43493.05
8	8(等级 2), 33(等级 6), 49(等级 1)	43540.19
7	8(等级 4), 33(等级 6), 49(等级 1)	43541.41
6	8(等级 3), 33(等级 6), 49(等级 2)	43582.71
5	8(等级 3), 33(等级 6), 49(等级 3)	43619.79
4	8(等级 4), 33(等级 6), 49(等级 2)	43822.31
3	8(等级 2), 33(等级 6), 49(等级 2)	43826.63
2	8(等级 4), 33(等级 6), 49(等级 3)	43853.78
1	8(等级 2), 33(等级 6), 49(等级 3)	43880.97

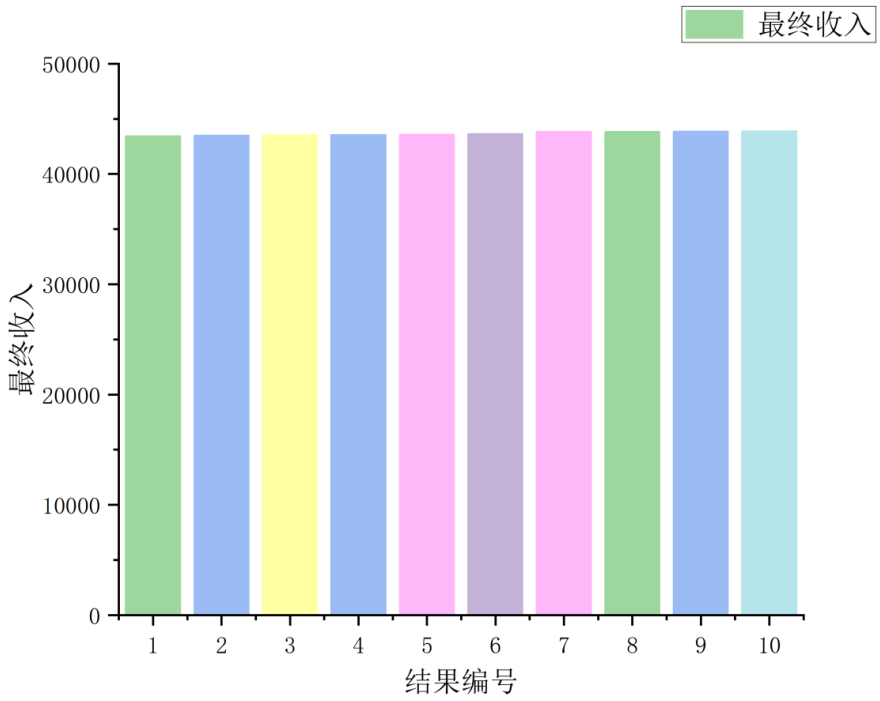

图 4 问题三 QUBO 模型运行结果前 10 名

4.3.2 量子退火算法、贪心算法与遗传算法对比分析

（1）贪心算法求解

针对问题三，贪心算法思路是：对于每组信用评分卡的组合，先遍历所有阈值组合，记录下可以获得最小总坏账率的那个阈值组合。然后，再在最优信用评分卡阈值组合下计算总通过率、总坏账率和最终收入。这种方法虽然仍需要枚举所有可能性，但只需要枚举每组卡的组合一次，而不是再对每组卡的组合进行一次循环，这使得时间复杂度从$O(n^7)$降低到$O(n^4)$，从而具有更高效率。

根据贪心算法求得最终收入排名前 10 的结果见表 5 和图 5，可见贪心算法只有一定概率能够取得全局最优解，并且其排名前十的结果相对于实际结果而言还存在较大差异。贪心算法的简单规则在处理多决策变量、大规模数据的问题时暴露出较大缺点。

表 5 问题三贪心算法运行结果前 10 名（因后面图片展示便利，故采用逆序）

排名	信用评分卡及阈值选择	最终收入
10	49(等级 1), 68(等级 1), 73(等级 1)	38773.27
9	33(等级 1), 49(等级 1), 83(等级 1)	38854.09
8	49(等级 1), 68(等级 1), 83(等级 1)	38854.09
7	41(等级 1), 49(等级 1), 73(等级 1)	39061.87
6	49(等级 1), 63(等级 1), 73(等级 1)	39085.27
5	41(等级 1), 49(等级 1), 83(等级 1)	39141.31
4	49(等级 1), 63(等级 1), 83(等级 1)	39150.29
3	4(等级 1), 49(等级 1), 73(等级 1)	39244.76
2	4(等级 1), 49(等级 1), 83(等级 1)	39336.23
1	49(等级 1), 73(等级 1), 83(等级 1)	40026.36

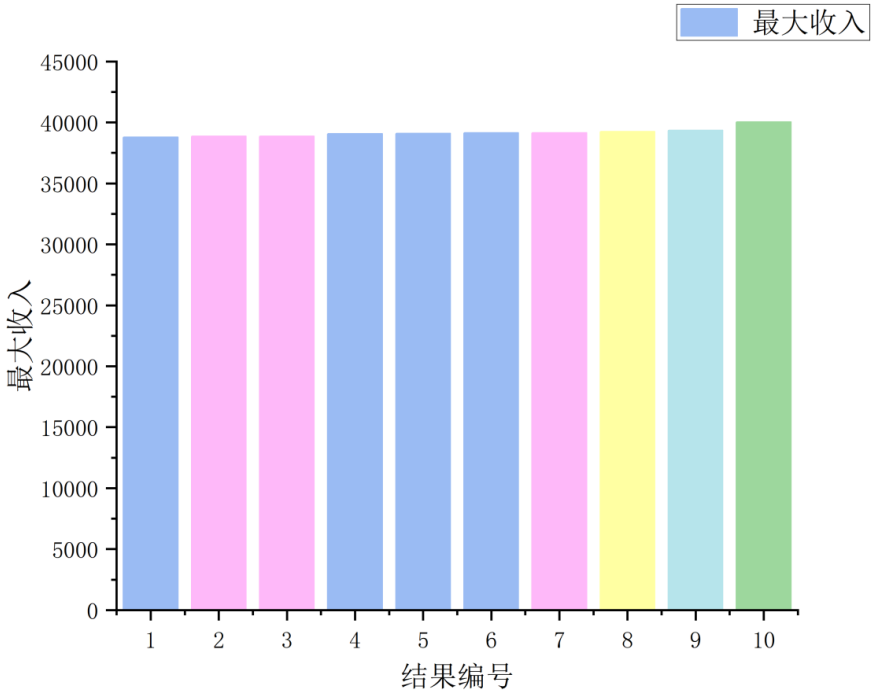

图 5 问题三贪心算法运行结果前 10 名

（2）遗传算法求解

针对问题三，通过遗传算法求解时步骤大致如下：

①确定染色体编码方式：使用一个长度为 33（3 张卡片，每张卡片有 10 种等级）的二进制串作为染色体编码。例如，001101100010111010001101101011001 表示选择第 27、58、53 号卡片且这三张卡片的等级分别为 1、3、9。

②初始化种群：由于染色体长度固定为 30，可以随机生成一些长度为 30 的二进制串作为初始种群。

③确定适应度函数：适应度函数用于评估每个染色体的优劣程度。针对问题三，可以使用最终收入作为适应度函数。对于每个染色体，可以将其转化为对应的卡片组合和等级，然后根据收入公式计算该组合对应的最终收入作为适应度值。

④选择操作：为了保留好的基因，需要执行选择操作。可选用轮盘赌或其他方法对种群中的个体进行选择，选择过程中适应度高的个体有更高被选择概率。

⑤交叉操作：交叉操作用于产生新的染色体，可选用单点、多点或均匀交叉等方法。

⑥变异操作：变异操作可以增加种群的多样性，可选用随机翻转、插入、删除等方法，对某些个体进行变异。

⑦终止条件：在达到一定迭代次数或者找到满足要求的染色体后，算法可以终止。最终输出收敛结果，即最大收入以及对应的卡片组合和等级。

⑧设定初始种群数为 200、迭代 1000 次后得到图 6 结果。得到最终的最优解为选择第 8 张银行卡的等级 3、第 49 张卡的等级 2、第 83 张卡的等级 2，最终收入为 42684.55。

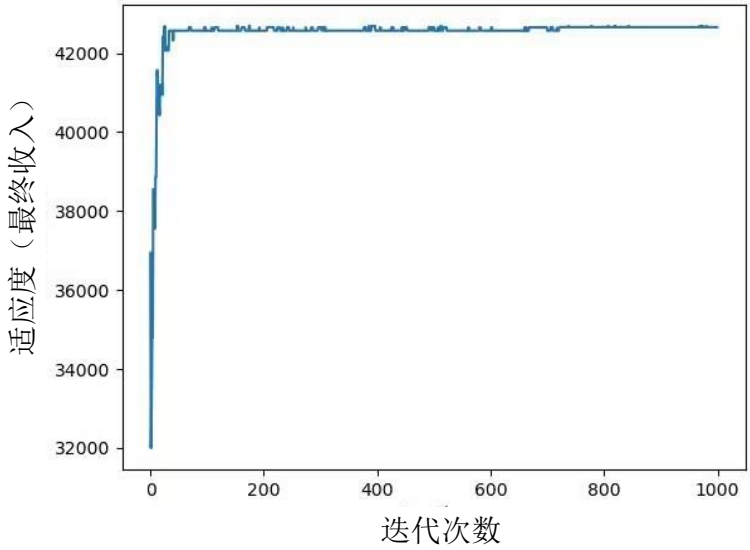

图 6 问题三遗传算法运行结果

（3）对比分析

综合以上结论和理论介绍，针对问题三，QUBO 模型运用量子退火算法优化效果最佳，遗传算法次之，贪心算法较差。以下总结贪心算法、遗传算法和 QUBO 模型运用量子退火算法优缺点：

贪心算法是一种基于局部最优策略的优化算法，它在每一个阶段都选择当前最优方案，然后继续求解下一个子问题，直到达到整体最优解。优点在于其时间复杂度通常较低，但由于只考虑当前最优解，而没有考虑长远影响，因此可能会产生局部最优解，并不能保证得到全局最优解。

遗传算法是一种模仿自然进化过程的智能算法，它通过选择、交叉和变异等操作来生成新的解，并逐步优化这些解。遗传算法优点在于对于复杂问题具有较强的全局搜索能力，能够找到比传统算法更优的解。可以在多维度的参数空间中进行搜索，适用于大规模、高维的问题。但是对于大规模问题其运算时间可能较长，参数的选择对算法的效果影响很大，需要经过调试和实验来确定。

QUBO 模型最大优点在于充分利用量子计算优势，具有高计算速度和高精度结果，可以通过增加量子比特数来提高求解精度和效率。但其缺点也十分明显，目前量子计算机仍处于实验阶段，硬件成本和技术门槛较高。其对于大规模问题求解仍然存在挑战，需要更多研究和探索。

表 6 QUBO 模型、遗传算法、贪心算法对比分析

	QUBO 模型	遗传算法	贪心算法
优点	高计算速度、高精度	有较强全局搜索能力	时间复杂度通常较低
缺点	技术仍需发展	大规模问题效果有限	可能产生局部最优解
问题三优化效果	QUBO 模型>遗传算法>贪心算法		

5. 模型评价与改进

5.1 模型优点

本文通过对不同问题特点和复杂度进行分析，采用换元等技巧构建 QUBO 模型利用量子退火算法求解并另用不同算法进行求解，对比不同算法面对不同问题复杂度时的性能表现并凸显 QUBO 模型利用量子退火算法的优势。

针对问题一，由于决策变量仅有一个，使用简单随机采样和 QUBO 模型运用量子退火算法进行结论对比，发现 QUBO 模型运用量子退火算法更加准确和高效。

针对问题二，问题复杂度有所提升，使用贪心算法和 QUBO 模型运用量子退火算法进行对比。这样可以更好地理解不同算法在不同情境下优缺点，及如何选择最合适算法来解决问题。

针对问题三，问题复杂度进一步增加，需要对信用评分卡和阈值等级分别作出决策，需要大量算力和时间。因此，本文使用贪心算法、遗传算法和 QUBO 模型运用量子退火算法进行对比，以找到最优解决方案。通过分析不同算法表现，可以深入探究其适用范围、优点和不足之处，并为后续研究提供参考。

综上，在问题复杂度逐步增加过程中使用不同算法和 QUBO 模型运用量子退火算法进行对比，可以更好了解这些方法的特点和应用场景。该递进式求解过程有助于更好理解不同算法在不同场景下的优劣性，以便未来在实际应用中选择合适的算法。

5.2 模型不足

第一，QUBO 模型的矩阵还可以进一步优化，以减小算力。QUBO 模型的矩阵可以通过系数化和特殊结构化等方法来进一步优化，但本文仅对矩阵做降幂处理，而没有提升其算法性能。第二，采用遗传算法时，设定的初始种群和迭代次数还有待调节，其中对于选择和变异的方法也可以进一步优化。第三，采用贪心算法时，也只是简单的定了第一步的筛选条件，也就是对银行的通过率的高低进行比对，但是在更大的数据规模下，应该去设定特定的条件以提升其答案的准确度。

5.3 模型改进

第一，可以采用更多新算法来对该问题进行求解，以对比不同算法在结论和性能上的差异，包括度下降算法、粒子群算法等。第二，单一数学模型可能无法完全描述一个复杂实际问题，因此也可以尝试将多个模型进行融合和集成以达到更好的结果。例如，可以利用模糊逻辑、神经网络等方法将不同模型融合起来，以获得更好预测精度和效果。

参考文献

[1] 汪勇，孟香君，沈维萍．量子计算在经济与金融领域中的应用[J]．经济学动态，2023(1):18-32.

[2] Emms D, Wilson R C, Hancock E R. Graph matching using the interference of discrete-time quantum walks[J]. Image and Vision Computing, 2009, 27(7):934-949.

[3] Farhi E, Goldstone J, Gutmann S, et al. Quantum Computation by Adiabatic Evolution[J]. Physics, 2000.

[4] Rosenberg G, Haghnegahdar P, Goddard P, et al. Solving the Optimal Trading Trajectory Problem Using a Quantum Annealer[J]. Selected Topics in Signal Processing, IEEE Journal of, 2016, 10(6):1053-1060.

[5] Venturelli D, Kondratyev A. Reverse Quantum Annealing Approach to Portfolio Optimization Problems[J]. Papers, 2019(1): 17-30.

[6] Egger D J, Gambella C, Marecek J, et al. Quantum Computing for Finance: State of the Art and Future Prospects[J]. IEEE Transactions on Quantum Engineering, 2020: 3101724.

[7] Falco D D, Tamascelli D. An introduction to quantum annealing [C]. Italian Conference on Theoretical Computer Science. EDP Sciences, 2011(45): 99-116.

[8] Zaman M, Tanahashi K, Tanaka S. PyQUBO: Python library for mapping combinatorial optimization problems to QUBO form[J]. IEEE Transactions on Computers, 2021, 71(4): 838-850.

[9] Glover F, Kochenberger G, Du Y. Quantum Bridge Analytics I: a tutorial on formulating and using QUBO models[J]. 4OR quarterly journal of the Belgian, French and Italian Operations Research Societies, 2019(5): 68-81.

附录：代码环境与代码清单

代码环境
操作系统：Windows 11 编程语言：Python 编辑器：Pycharm 代码详见：Code 文件夹

代码清单 1：问题 1——量子退火算法

```python
# coding=gb
import pandas as pd
import neal
from pyqubo import Binary, Constraint, Placeholder, Array, OneHotEncInteger
def load_data_100():
    list1 = []
    list2 = []
    for i in range(0, 200):
        if i % 2 == 0:
            list1.append(i)
            list2.append(i)
    pass_rate_list = []
    bad_rate_list = []
    namelist = []
    df = pd.read_csv("data_100.csv")
    for index, col in df.iteritems():
        namelist.append(index)
    for a in list1:
        rows = df[namelist[a]].tolist()
        pass_rate_list.extend(rows)
    for b in list2:
        rows = df[namelist[b]].tolist()
        bad_rate_list.extend(rows)
    print("信用卡的通过率为：", pass_rate_list)
    print("信用卡的坏账率为：", bad_rate_list)
    return pass_rate_list, bad_rate_list
pass_rate_list, bad_rate_list = load_data_100()

# 加载信用卡的通过率和坏账率，把 100 个信用卡转化为 1000 个信用卡
pass_weights = pass_rate_list
values = bad_rate_list

# 创建决策变量 Xi，一共有 n 个决策变量，n=1000
n = len(values)
```

```
items = Array.create('item', shape=n, vartype="BINARY")

# 构建通过率和坏账率两部分的因式
pass_weight = sum(pass_weights[i] * items[i] for i in range(n))
bad_weight = sum(values[i] * items[i] for i in range(n))
constraint_weight = 1 - sum(items[i] for i in range(n))

# 定义惩罚函数
lmd2 = Placeholder("lmd2")
Ha = Constraint((constraint_weight) ** 2, "weight_constraint")
Hb = -1080000 * bad_weight * pass_weight + 80000 * pass_weight
H = -Hb + lmd2 * Ha
model = H.compile()
sampler = neal.SimulatedAnnealingSampler()
feasible_sols = []

# 设置惩罚函数，惩罚函数尽量设置大
feed_dict = {"lmd2": 1000000000000000000}
qubo, offset = model.to_qubo(feed_dict=feed_dict)
bqm = model.to_bqm(feed_dict=feed_dict)
bqm.normalize()
sampleset = sampler.sample(bqm, num_reads=1000, sweeps=1000, beta_range=(1.0, 50.0))
# 采样的集,每一次采样的结果函数值都为 energy，然后存在 dec_sample 中
dec_samples = model.decode_sampleset(sampleset, feed_dict=feed_dict)
# 选择所有采样值中最小的那一个，即为最优值
best = min(dec_samples, key=lambda x: x.energy)

# 判断每一次迭代的结果是否满足约束条件
if not best.constraints(only_broken=True):
    feasible_sols.append(best)

# 获得最小值
best_feasible = min(feasible_sols, key=lambda x: x.energy)
print(f"selection = {[best_feasible.sample[f'item[{i}]'] for i in range(n)]}")
print(f"sum of the values = {-best_feasible.energy}")
```

代码清单 2：问题 2——量子退火算法

```
# coding=gbk
import math
import numpy as np
import pandas as pd
import neal
from pyqubo import Binary, Constraint, Placeholder, Array, OneHotEncInteger


# 转化为数据，将提供的坏账率和通过率由一维数组转化为二维数组
def reload_data(inputlist):
    outputlist = [[0 for x in range(10)] for y in range(3)]
    for i in range(3):
        for j in range(10):
            outputlist[i][j] = inputlist[i * 10 + j]
    return outputlist
def load_data_100_2():
    list1 = []
    list2 = []
    for i in range(0, 6):
        if i % 2 == 0:
            list1.append(i)
        else:
            list2.append(i)
    pass_rate_list = []
    bad_rate_list = []
    namelist = []
    df = pd.read_csv("data_100.csv")
    for index, col in df.iteritems():
        namelist.append(index)
    for a in list1:
        rows = df[namelist[a]].tolist()
        pass_rate_list.extend(rows)
    for b in list2:
        rows = df[namelist[b]].tolist()
        bad_rate_list.extend(rows)
    print("信用卡的通过率为：", pass_rate_list)
    print("信用卡的坏账率为：", bad_rate_list)
    return pass_rate_list, bad_rate_list
pass_rate_list, bad_rate_list = load_data_100_2()

# 加载信用卡的通过率和坏账率，把 100 个信用卡转化为 1000 个信用卡
pass_weights = reload_data(pass_rate_list)
values = reload_data(bad_rate_list)
```

```
43.
44. # 创建决策变量 Xi，一共有 n 个决策变量，n=shape=100*10
45. items_X1 = Array.create('items_X1', shape=10, vartype="BINARY")
46. items_X2 = Array.create('items_X2', shape=10, vartype="BINARY")
47. items_X3 = Array.create('items_X3', shape=10, vartype="BINARY")
48. items_Y = Array.create('items_Y', shape=(10, 10), vartype="BINARY")
49. items_T = Array.create('items_T', shape=(10, 10, 10), vartype="BINARY")
50.
51. # 构建通过率和坏账率两部分的因式
52. pass_weight = sum(pass_weights[0][i]*pass_weights[1][j]*pass_weights[2][z]*
    items_T[i][j][z] for i in range(10) for j in range(10) for z in range(10))
53. bad_weight = sum(values[0][i] * items_X1[i] for i in range(10))+ sum(values
    [1][j] * items_X2[j] for j in range(10)) + sum(values[2][z] * items_X3[z] f
    or z in range(10))
54.
55. # 构建约束条件
56. constraint_weight1 = 1 - sum(items_X1[i] for i in range(10))
57. constraint_weight2 = 1 - sum(items_X2[i] for i in range(10))
58. constraint_weight3 = 1 - sum(items_X3[i] for i in range(10))
59.
60. # 定义惩罚因子
61. alpha = 1000000000000000000
62. beta = 1000000000000000000
63. constraint_weight_alpha = alpha * sum(
64.     items_X1[i] * items_X2[j] - 2 * items_Y[i][j] * (items_X1[i] + items_X2
    [j]) + 3 *items_Y[i][j] for i in range(10) for j in range(10))
65. constraint_weight3_beta = beta * sum(
66.     items_X3[z] * items_Y[i][j] - 2 * items_T[i][j][z] * (items_X3[z] + ite
    ms_Y[i][j]) + 3 *items_T[i][j][z] for i in range(10) for j in range(10) for
     z in range(10))
67.
68. # 定义惩罚函数
69. lmd1 = Placeholder("lmd1")
70. lmd2 = Placeholder("lmd2")
71. lmd3 = Placeholder("lmd3")
72. Halpha = Constraint(constraint_weight_alpha , "weight_constraint_alpha")
73. Hbeta = Constraint(constraint_weight3_beta, "weight_constraint_beta")
74. Ha = 360000 * bad_weight * pass_weight - 80000 * pass_weight
75. H1 = Constraint(constraint_weight1 ** 2, "weight_constraint1")
76. H2 = Constraint(constraint_weight2 ** 2, "weight_constraint2")
77. H3 = Constraint(constraint_weight3 ** 2, "weight_constraint3")
78. H = Ha + lmd1 * H1 + lmd2 * H2 + lmd3 * H3+ Halpha + Hbeta
79. model = H.compile()
80. sampler = neal.SimulatedAnnealingSampler()
```

```
feasible_sols = []

# 设置惩罚函数，惩罚函数尽量设置大
feed_dict = {"lmd1": 1000000000000000000, "lmd2": 1000000000000000000, "lmd3": 1000000000000000000}
qubo, offset = model.to_qubo(feed_dict=feed_dict)
bqm = model.to_bqm(feed_dict=feed_dict)
bqm.normalize()
sampleset = sampler.sample(bqm, num_reads=1000, sweeps=1000, beta_range=(1.0, 50.0))

# 采样的集,每一次采样的结果函数值都为 energy，然后存在 dec_sample 中
dec_samples = model.decode_sampleset(sampleset, feed_dict=feed_dict)

# 选择所有采样值中最小的那一个，即为最优值
best = min(dec_samples, key=lambda x: x.energy)

# 判断每一次迭代的结果是否满足约束条件
if not best.constraints(only_broken=True):
    feasible_sols.append(best)

  # 获得最小值
  best_feasible = min(feasible_sols, key=lambda x: x.energy)
  print(f"selection = {[best_feasible.sample[f'items_X1[{i}]'] for i  in range(10)]}")
  print(f"selection = {[best_feasible.sample[f'items_X2[{i}]'] for i  in range(10)]}")
  print(f"selection = {[best_feasible.sample[f'items_X3[{i}]'] for i  in range(10)]}")
  print(f"selection = {[best_feasible.sample[f'items_Y[{i}][{j}]'] for i in range(10) for j  in range(10)]}")
  print(f"selection = {[best_feasible.sample[f'items_T[{i}][{j}][{z}]'] for i  in range(10) for j  in range(10) for z  in range(10)]}")
  print(f"sum of the values = {-best_feasible.energy}")
```

代码清单 3：问题 2——贪心算法

```python
import numpy as np
# 假设有 3 个等级对应表，每个表有 10 个等级
num_tables = 3
num_levels = 10

# 定义不同等级对应的通过率和坏账率
p = np.array([[0.76, 0.77, 0.78, 0.8, 0.82, 0.84, 0.87, 0.93, 0.94, 0.96],
              [0.72, 0.73, 0.76, 0.77, 0.79, 0.82, 0.86, 0.87, 0.91, 0.92],

              [0.8, 0.82, 0.83, 0.86, 0.89, 0.92, 0.93, 0.94, 0.97, 0.98]])
q = np.array([[0.013, 0.015, 0.017, 0.024, 0.026, 0.028, 0.03, 0.036, 0.043
, 0.052],
              [0.032, 0.038, 0.05, 0.053, 0.065, 0.074, 0.076, 0.079, 0.08,
 0.084],
              [0.012, 0.013, 0.024, 0.04, 0.042, 0.057, 0.068, 0.069, 0.07,
 0.073]])

# 定义计算收入的函数
def calculate_revenue(levels):
    A = np.prod([p[i][levels[i]] for i in range(num_tables)])
    B = np.mean([q[i][levels[i]] for i in range(num_tables)])
    return 80000 * A - 1080000 * A * B

# 初始化当前最优解
best_levels = [0] * num_tables
best_revenue = calculate_revenue(best_levels)

# 遍历所有可能的等级组合，找到收入最大的那一组
for i in range(num_tables):
    max_p_index = np.argmax(p[i])
    for j in range(max_p_index, -1, -1):
        if q[i][j] <= 0.05:
            best_levels[i] = j
            break

# 计算收入
best_revenue = calculate_revenue(best_levels)

# 输出结果
print(f"选择的卡片等级为：{best_levels}")
print(f"总收入为：{best_revenue:.2f}")
```

代码清单 4：问题 3——量子退火算法

```
# coding=gbk
import math
import numpy as np
import pandas as pd
import neal
from pyqubo import Binary, Constraint, Placeholder,Array, OneHotEncInteger


#转化为数据，将提供的坏账率和通过率由一维数组转化为二维数组
def reload_data(inputlist):
    outputlist= [[0 for i in range(10)] for j in range(100)]
    print(outputlist)
    for i in range(100):
        for j in range(10):
            outputlist[i][j]=inputlist[i*10+j]
    return outputlist
def load_data_100():
    list1=[]
    list2=[]
    for i in range(0,200):
        if i%2==0:
            list1.append(i)
        else:
            list2.append(i)
    pass_rate_list=[]
    bad_rate_list=[]
    namelist=[]
    df=pd.read_csv("data_100.csv")
    for index, col in df.iteritems():
        namelist.append(index)
    for a in list1:
                rows = df[namelist[a]].tolist()
                pass_rate_list.extend(rows)
    for b in list2:
                rows = df[namelist[b]].tolist()
                bad_rate_list.extend(rows)
    print("信用卡的通过率为：",pass_rate_list)
    print("信用卡的坏账率为：",bad_rate_list)
    return pass_rate_list,bad_rate_list
pass_rate_list,bad_rate_list=load_data_100()

#加载信用卡的通过率和坏账率，把 100 个信用卡转化为 1000 个信用卡
```

```
pass_weights = reload_data(pass_rate_list)
values = reload_data(bad_rate_list)

#创建决策变量 Xi，一共有 n 个决策变量，n=shape=100*10
items_X1 = Array.create('items_X1', shape=(100, 10), vartype="BINARY")
items_X2 = Array.create('items_X2', shape=(99, 10), vartype="BINARY")
items_X3 = Array.create('items_X3', shape=(98, 10), vartype="BINARY")
items_Y = Array.create('items_Y', shape=(100,99,10,10), vartype="BINARY")
items_T = Array.create('items_T', shape=(100,99,98,10,10,10), vartype="BINARY")

#构建通过率和坏账率两部分的因式
pass_weight = sum(pass_weights[o][i]*pass_weights[p][j]*pass_weights[q][z]* items_T[o][p][q][i][j][z] for o in range(100) for p in range(o+1,100) for q in range(p+1,100) for i in range(10) for j in range(10)  for z in range(10))
bad_weight = sum((values[o][i] * items_X1[o][i]+values[p][j] * items_X2[p][j]+values[q][z] * items_X3[q][z]) for o in range(100) for p in range(o+1,100) for q in range(p+1,100) for i in range(10) for j in range(10)  for z in range(10))

#构建约束条件
constraint_weight1 = 1-sum(items_X1[i][j] for i in range(100) for j in range(10))
constraint_weight2 = 1-sum(items_X2[i][j] for i in range(99) for j in range(10))
constraint_weight3 = 1-sum(items_X3[i][j] for i in range(98) for j in range(10))

#定义惩罚因子
alpha=1000000000000000000
beta=1000000000000000000
constraint_weight_alpha = alpha*sum(items_X1[o][i]*items_X2[p][j]-2*items_Y[o][p][i][j]*(items_X1[o][i]+items_X2[p][j])+3*items_Y[o][p][i][j] for o in range(100) for p in range(o+1,100) for i in range(10) for j in range(10))
constraint_weight3_beta = beta*sum(items_X3[q][z]*items_Y[o][p][i][j]-2*items_T[o][p][q][i][j][z]*(items_X3[q][z]+items_Y[o][p][i][j])+3*items_T[o][p][q][i][j][z] for o in range(100) for p in range(o+1,100) for q in range(p+1,100) for i in range(10) for j in range(10)  for z in range(10))

#定义惩罚函数
lmd1 = Placeholder("lmd1")
lmd2 = Placeholder("lmd2")
```

```
lmd3 = Placeholder("lmd3")
Ha = -360000*bad_weight*pass_weight+80000*pass_weight
Halpha=Constraint(constraint_weight_alpha,"weight_constraint_alpha")
Hbeta=Constraint(constraint_weight3_beta,"weight_constraint_beta")
H1 = Constraint((constraint_weight1)**2,"weight_constraint1")
H2 = Constraint((constraint_weight2)**2,"weight_constraint2")
H3 = Constraint((constraint_weight3)**2,"weight_constraint3")
H = -Ha+lmd1*H1+lmd2*H2+lmd3*H3+Halpha+Hbeta
model = H.compile()
sampler = neal.SimulatedAnnealingSampler()
feasible_sols = []

#设置惩罚函数，惩罚函数尽量设置大
feed_dict = {"lmd1":1000000000000000000,"lmd2":1000000000000000000,"lmd3":1000000000000000000}
qubo, offset = model.to_qubo(feed_dict=feed_dict)
bqm = model.to_bqm(feed_dict=feed_dict)
bqm.normalize()
sampleset = sampler.sample(bqm, num_reads=1000,sweeps=1000, beta_range=(1.0, 50.0))

#采样的集,每一次采样的结果函数值都为 energy，然后存在 dec_sample 中
dec_samples = model.decode_sampleset(sampleset,feed_dict=feed_dict)

#选择所有采样值中最小的那一个，即为最优值
best = min(dec_samples, key=lambda x: x.energy)
print(best)

#判断每一次迭代的结果是否满足约束条件
if not best.constraints(only_broken=True):
            feasible_sols.append(best)

  #获得最小值
  best_feasible = min(feasible_sols, key=lambda x: x.energy)

  print(f"selection = {[best_feasible.sample[f'items_X1[{i}][{j}]'] for i in range(100) for j in range(10)]}")
  print(f"selection = {[best_feasible.sample[f'items_X2[{i}][{j}]'] for i in range(100) for j in range(10)]}")
  print(f"selection = {[best_feasible.sample[f'items_X3[{i}][{j}]'] for i in range(100) for j in range(10)]}")
  print(f"sum of the values = {-best_feasible.energy}")
```

代码清单 5：问题 3——贪心算法

```
import pandas as pd
import matplotlib.pyplot as plt

# 读取数据集
df = pd.read_csv('/kaggle/input/threemodel/Three.csv', index_col=None)

# 初始化变量
max_incomes = []
card_idxs = []
levelss = []

# 遍历所有可能的三张卡组合和等级
for i in range(1, 101):
    for j in range(i + 1, 101):
        for k in range(j + 1, 101):
            # 记录最优等级组合
            best_levels = [0, 0, 0]
            # 最小总坏账率
            min_bad_rate = float('inf')
            # 遍历每种等级组合
            for x in range(1, 11):
                for y in range(1, 11):
                    for z in range(1, 11):
                        # 计算总通过率
                        total_pass_rate = df.loc[x - 1, f"t_{i}"] * df.loc[y - 1, f"t_{j}"] * df.loc[z - 1, f"t_{k}"]
                        # 计算总坏账率
                        total_bad_rate = (df.loc[x - 1, f"h_{i}"] + df.loc[y - 1, f"h_{j}"] + df.loc[z - 1, f"h_{k}"]) / 3
                        # 计算最终收入
                        income = 80000 * total_pass_rate - 1080000 * total_pass_rate * total_bad_rate

                        # 更新最小总坏账率和最优等级组合
                        if total_bad_rate < min_bad_rate:
                            min_bad_rate = total_bad_rate
                            best_levels = [x, y, z]

            # 在最优等级组合下计算总通过率、总坏账率和最终收入
            total_pass_rate = df.loc[best_levels[0] - 1, f"t_{i}"] * df.loc[best_levels[1] - 1, f"t_{j}"] * df.loc[best_levels[2] - 1, f"t_{k}"]
```

```
            total_bad_rate = (df.loc[best_levels[0] - 1, f"h_{i}"] + df.loc[best_levels[1] - 1, f"h_{j}"] + df.loc[best_levels[2] - 1, f"h_{k}"]) / 3

            income = 80000 * total_pass_rate - 1080000 * total_pass_rate * total_bad_rate

            # 更新前十名最优选择的列表
            if len(max_incomes) < 10 or income > min(max_incomes):
                max_incomes.append(income)
                card_idxs.append([i, j, k])
                levelss.append(best_levels)

                # 如果列表长度超过了 10，就去掉最小值
                if len(max_incomes) > 10:
                    min_index = max_incomes.index(min(max_incomes))
                    del max_incomes[min_index], card_idxs[min_index], levelss[min_index]

# 输出前十名最优选择的结果
for rank, (max_income, card_idx, levels) in enumerate(zip(max_incomes, card_idxs, levelss), 1):
    print(f"第 {rank} 名选择的卡为{card_idx[0]}(等级{levels[0]}), {card_idx[1]}(等级{levels[1]}), {card_idx[2]}(等级{levels[2]})")
    print(f"最大收入为{max_income:.2f}")
    print("=" * 30)

# 绘制前十名最优选择的收入折线图
plt.plot(range(1, 11), max_incomes)
plt.xlabel('排名')
plt.ylabel('最大收入')
plt.title('前十名最优选择的收入')
plt.show()
```

代码清单 6：问题 3——遗传算法

```
import pandas as pd
import random
import matplotlib.pyplot as plt

# 读取数据集
df = pd.read_csv('/kaggle/input/threemodel/Three.csv', index_col=None)

# 遗传算法参数
POP_SIZE = 200  # 种群大小
GENE_SIZE = 31  # 染色体长度
CROSS_RATE = 0.8  # 交叉概率
MUTATE_RATE = 0.05  # 变异概率
N_GENERATIONS = 500  # 迭代次数


# 计算总通过率和总坏账率
def cal_rate(cards, levels):
    pass_rate = 1
    bad_rate = 0
    for i in range(3):
        card_id = cards[i]
        level_id = levels[i]
        if card_id == 0 or card_id > 100:
            card_id = 100
        if level_id >= 10:
            level_id = 9
        pass_rate *= df.loc[level_id][f"t_{card_id}"]
        bad_rate += df.loc[level_id][f"h_{card_id}"]
    bad_rate /= 3
    return pass_rate, bad_rate


# 计算最终收入
def cal_income(pass_rate, bad_rate):
    return 80000 * pass_rate - 1080000 * pass_rate * bad_rate


# 初始化种群
pop = ["".join([str(random.randint(0, 1)) for _ in range(GENE_SIZE)]) for _
    in range(POP_SIZE)]
print(len(pop))
# 存储每一代最大的适应度值
max_fitness_history = []
```

```
max_pop_fitness = {}
max_pop = []
# 迭代
for generation in range(N_GENERATIONS):
    # 计算适应度
    fits = []
    probs = []
    for chromosome in pop:
        card_idx = [int(chromosome[i:i + 7], 2) for i in range(0, GENE_SIZE, 11)]
        levels = [int(chromosome[i:i + 4], 2) for i in range(7, GENE_SIZE, 11)]
        pass_rate, bad_rate = cal_rate(card_idx, levels)
        income = cal_income(pass_rate, bad_rate)
        fits.append(income)
        if income not in max_pop_fitness.keys():
            max_pop_fitness[income] = chromosome

    # 存储当前代最大的适应度值
    max_fitness_history.append(max(fits))
    max_pop.append(max_pop_fitness[max(fits)])
    # 选择
    min_fit = min(fits)
    fits = [fit - min_fit + 1e-4 for fit in fits]  # 调整适应度，防止负数
    fits_sum = sum(fits)
    probs = [fit / fits_sum for fit in fits]
    new_pop = []
    for _ in range(POP_SIZE // 2):
        idx1, idx2 = random.choices(range(POP_SIZE), weights=probs, k=2)
        chromosome1, chromosome2 = pop[idx1], pop[idx2]
        if random.random() < CROSS_RATE:
            cross_idx = random.randint(0, GENE_SIZE)
            new_chromosome1 = chromosome1[:cross_idx] + chromosome2[cross_idx:]
            new_chromosome2 = chromosome2[:cross_idx] + chromosome1[cross_idx:]
            new_pop.extend([new_chromosome1, new_chromosome2])
        else:
            new_pop.extend([chromosome1, chromosome2])
    #     print(len(new_pop))
    # 变异
    for i in range(len(new_pop)):
        chromosome = new_pop[i]
        if random.random() < MUTATE_RATE:
```

```
            mutate_idx = random.randint(0, GENE_SIZE - 1)
            new_gene = "1" if chromosome[mutate_idx] == "0" else "0"
            new_chromosome = chromosome[:mutate_idx] + new_gene + chromosome[mutate_idx + 1:]
            new_pop[i] = new_chromosome
    # 更新种群
    pop = new_pop
# 绘制适应度函数值随迭代次数变化的曲线
plt.plot(range(N_GENERATIONS), max_fitness_history)
plt.title("Fitness Value vs. Iteration")
plt.xlabel("Iteration")
plt.ylabel("Fitness Value")
plt.show()

# 找到最佳染色体
max_income = float('-inf')
card_idx = []
levels = []
  for chromosome in max_pop:
      cur_card_idx = [int(chromosome[i:i + 7], 2) for i in range(0, GENE_SIZE, 11)]
      cur_levels = [int(chromosome[i:i + 4], 2) for i in range(7, GENE_SIZE, 11)]
      cur_pass_rate, cur_bad_rate = cal_rate(cur_card_idx, cur_levels)
      cur_income = cal_income(cur_pass_rate, cur_bad_rate)
      if cur_income > max_income:
          max_income = cur_income
          card_idx = cur_card_idx
          levels = cur_levels

  # 输出结果
  print(
      f"选择的卡为{card_idx[0]}(等级{levels[0]}), {card_idx[1]}(等级{levels[1]}), {card_idx[2]}(等级{levels[2]})，最大收入为{max_income:.2f}元。")
```