100_prisoner 实验报告

肖俊浩 四川大学软件学院

一、算法说明

本实验程序编写主要实现以下函数模块：

1、盒子生成：

使用 np.random.permutation 生成 1~N 的随机排列，确保每个编号唯一且随机分布。

2、仿真不同策略

a.随机策略：为每个囚犯随机选择 K 个盒子进行检查。

b.循环策略：基于排列的循环特性，从囚犯自己的编号开始查找，直到找到目标或达到尝试次数上限

3、模拟框架

a.单轮模拟：支持两种策略，返回本轮实验的成功状态和成功人数。

b.多轮模拟：多轮模拟，最后返回两种策略不同的成功率

4、可视化结果

```
import random
import matplotlib.pyplot as plt

def generate_boxes(n):
    #生成 n 个盒子，每个盒子中放入一个不重复的囚犯编号（1 到 n）
    boxes = list(range(1, n + 1))
    random.shuffle(boxes)
    return boxes


def random_strategy(boxes, prisoner_id, max_attempts):
    #仿真随机策略：囚犯随机打开 K 个盒子，看是否能找到自己的编号
```

```
    attempts = 0
    opened_boxes = set()

    while attempts < max_attempts:
        box_to_open = random.randint(0, len(boxes) - 1)  # 0-based index
        if box_to_open in opened_boxes:
            continue  # 不重复打开同一个盒子

        if boxes[box_to_open] == prisoner_id:
            return True

        opened_boxes.add(box_to_open)
        attempts += 1

    return False


def cycle_strategy(boxes, prisoner_id, max_attempts):
    #仿真循环策略：从自己编号对应的盒子开始查找，直到找到自己或达到最大尝试次数
    current_box = prisoner_id - 1  # 盒子索引是0-based
    attempts = 0

    while attempts < max_attempts:
        if boxes[current_box] == prisoner_id:
            return True

        current_box = boxes[current_box] - 1  # 下一个盒子索引
        attempts += 1

    return False


def simulate_one_round(n, k, strategy_func):
    #模拟一轮实验：所有囚犯使用指定策略尝试找到自己的编号    boxes = generate_boxes(n)
    success_count = 0

    for prisoner_id in range(1, n + 1):
        if strategy_func(boxes, prisoner_id, k):
            success_count += 1

    all_success = (success_count == n)
    return all_success, success_count


```

```python
def run_simulation(n=100, k=50, trials=1000):
    #运行多轮实验，统计两种策略的成功率
    strategies = {
        'random': random_strategy,
        'cycle': cycle_strategy
    }

    results = {}

    for name, strategy_func in strategies.items():
        print(f"正在运行 {name} 策略...")
        success_count = 0
        success_distribution = []

        for trial in range(trials):
            all_success, count = simulate_one_round(n, k, strategy_func)
            if all_success:
                success_count += 1
            success_distribution.append(count)

        success_rate = success_count / trials
        results[name] = {
            'success_rate': success_rate,
            'success_distribution': success_distribution
        }

    return results


def plot_results(results, n, k, trials):
    #可视化结果：成功率对比和成功人数分布
    fig, axes = plt.subplots(1, 2, figsize=(14, 6))

    # 成功率对比图
    strategy_names = ['随机策略', '循环策略']
    rates = [results['random']['success_rate'], results['cycle']['success_rate']]

    axes[0].bar(strategy_names, rates, color=['#ff9999', '#66b3ff'])
    axes[0].set_title(f'成功率对比 (N={n}, K={k}, 实验次数={trials})')
    axes[0].set_ylabel('成功率')
    for i, v in enumerate(rates):
        axes[0].text(i, v, f'{v:.2%}', ha='center', va='bottom')

    # 成功人数分布图（仅显示循环策略）
```

```
    cycle_success_counts = results['cycle']['success_distribution']
    axes[1].hist(cycle_success_counts, bins=range(n+2), edgecolor='black', alpha=0.7)
    axes[1].set_title('循环策略成功人数分布')
    axes[1].set_xlabel('成功人数')
    axes[1].set_ylabel('频数')

    plt.tight_layout()
    plt.show()


if __name__ == '__main__':
    # 设置随机种子以确保结果可复现
    random.seed(42)

    # 参数设置
    N = 100   # 囚犯数量
    K = 50    # 每人最多尝试次数
    T = 1000  # 实验次数

    # 运行模拟
    results = run_simulation(N, K, T)

    # 可视化结果
    plot_results(results, N, K, T)

```

二、算法优化

优化的点主要在于三个：

1、向量化模拟随机策略：

使用 NumPy 数组代替 Python 原生列表，提升数据运算速度；

在随机策略中一次性生成所有尝试，减少循环开销；

利用 NumPy 内置函数（如 np.any()）加速判断操作。

```
def random_strategy_vectorized(boxes: np.ndarray, prisoner_num: int, max_attempts: int)
-> bool:
    n = len(boxes)
    attempts = np.random.choice(n, max_attempts, replace=False)
    return np.any(boxes[attempts] == prisoner_num)
```

```
5.
```

2、利用批次化

在随机策略中为所有囚犯一次性生成随机尝试路径。

3、并行化加速计算

使用 multiprocessing.Pool 实现跨 CPU 核心并行；

将总实验次数平均分配给各 CPU 核心；

程序支持动态调整进程数。

```
1. def run_simulation_parallel(n: int = 100, k: int = 50, trials: int = 10000, n_processes:
int = None) -> Tuple[dict, dict]:
2. #并行化多轮模拟
3.     if n_processes is None:
4.         n_processes = mp.cpu_count()  # 使用所有可用的 CPU 核心
5.
6.     # 将总实验次数分配给每个进程
7.     trials_per_process = trials // n_processes
8.     remaining_trials = trials % n_processes
9.
10.    # 准备并行处理参数
11.    random_args = [(n, k, 'random', trials_per_process + (1 if i < remaining_trials else
0))
12.                for i in range(n_processes)]
13.    cycle_args = [(n, k, 'cycle', trials_per_process + (1 if i < remaining_trials else
0))
14.                for i in range(n_processes)]
15.
16.    # 创建进程池并执行并行计算
17.    with mp.Pool(processes=n_processes) as pool:
18.        random_results_list = pool.map(parallel_simulate, random_args)
19.        cycle_results_list = pool.map(parallel_simulate, cycle_args)
20.
21.    # 合并结果
22.    random_results = {'success_rate': 0, 'success_counts': []}
23.    cycle_results = {'success_rate': 0, 'success_counts': []}
24.
25.    for result in random_results_list:
26.        random_results['success_rate'] += result['success_rate'] *
(len(result['success_counts']) / trials)
27.        random_results['success_counts'].extend(result['success_counts'])
```

```
28.
29.     for result in cycle_results_list:
30.         cycle_results['success_rate'] += result['success_rate'] *
(len(result['success_counts']) / trials)
31.         cycle_results['success_counts'].extend(result['success_counts'])
32.
33.     return random_results, cycle_results
34.
```

三、实验结果分析

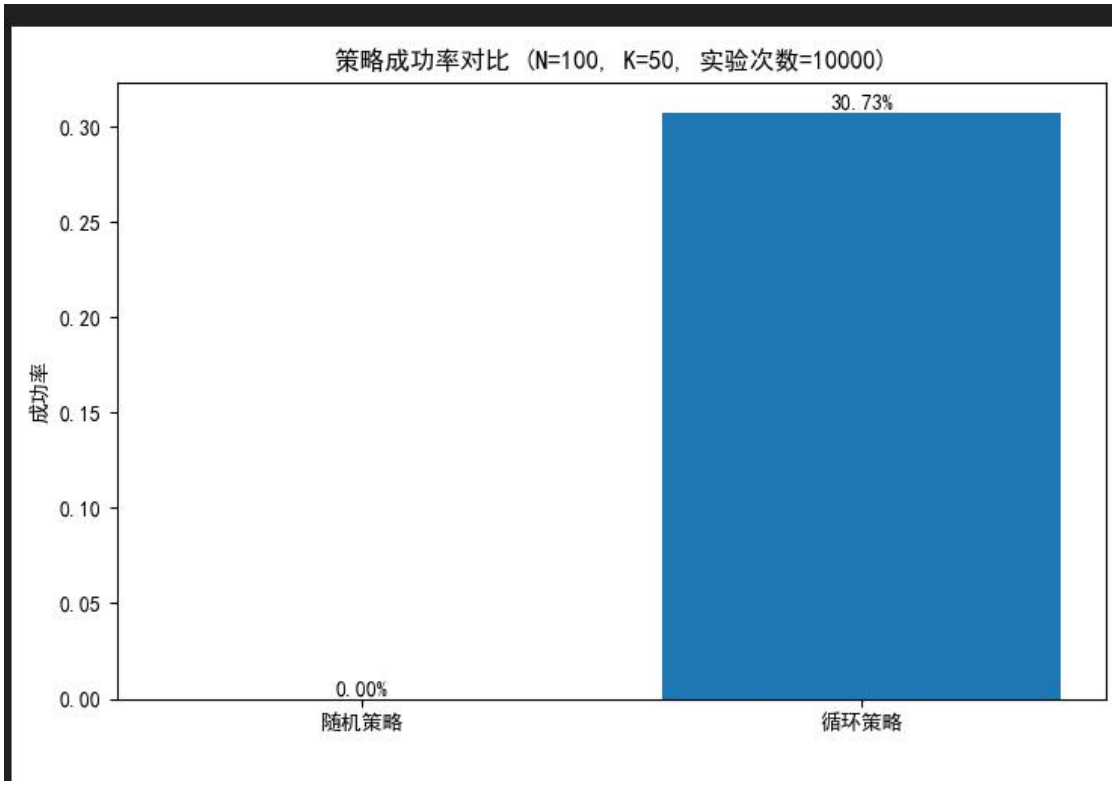

图 1 策略成功率对比

参数：N=100，K=50，实验次数=10000

成功率对比：随机策略的成功率为 0.00% 循环策略的成功率为 30.73%

从数学期望上看，随机策略：对于每个囚犯，找到自己编号的概率为：

$$\frac{K}{N} = \frac{50}{100} = 0.5。$$

所有囚犯都成功的概率为，$(0.5)^{100} \approx 8.9 \times 10^{-31}$，几乎为 0，符合实验结果。

循环策略：根据数学理论，当 K ≥ N/2 时，循环策略的成功率大约为$\frac{1}{e} \approx 36.8\%$，接近实验结果 30.73%。

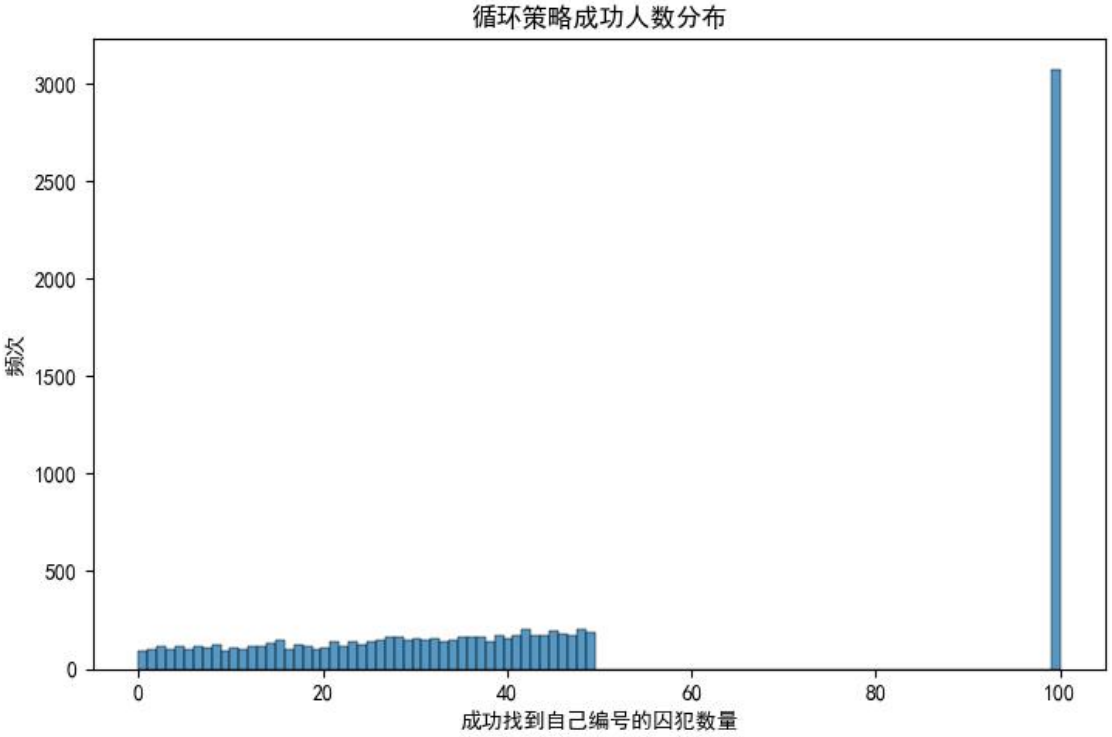

图 2 循环策略成功人数分布

实验结果显示，大多数情况下，要么大部分囚犯失败（频次较低），要么全部成功（频次较高）。这符合循环策略的特点：要么全胜，要么全败。

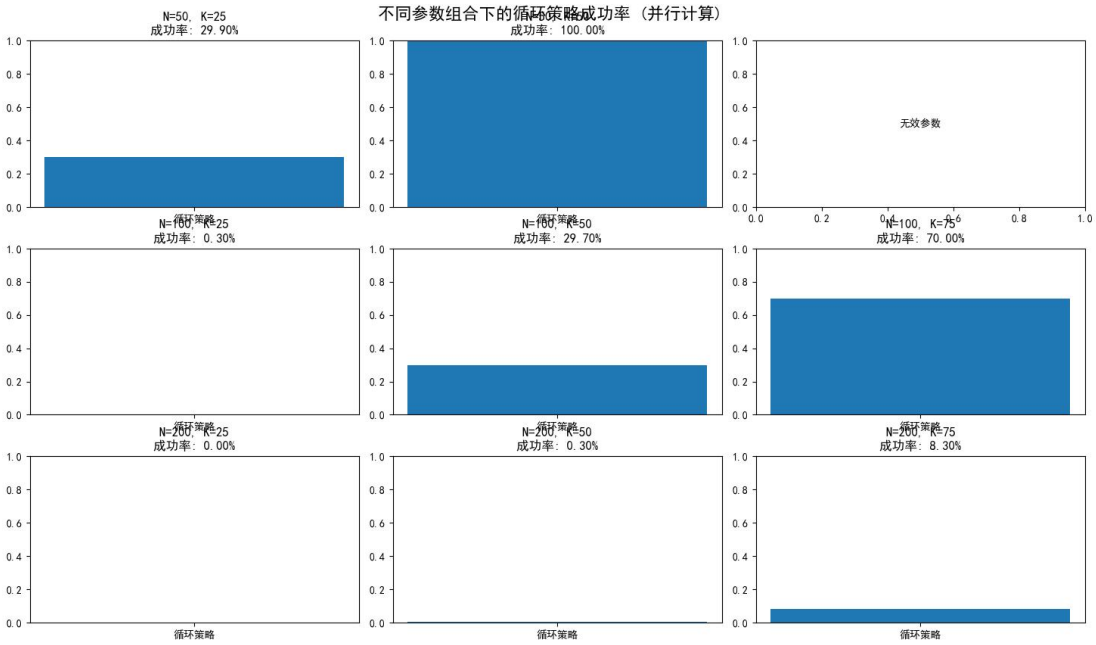

图 3 不同参数对循环策略成功率影响

显然，K 越接近于 N/2，以至于越接近于 N，成功率越高，在 N/2 时约为 36.8%，当显著小于 N/2 成功率极低，实验结果符合理论。