百囚徒问题模拟分析报告

> **Note**
>
> - **作者**：**王文杰**
> - **学号**：**2022141090107**

1. 问题描述

百囚徒问题是一个经典的概率谜题。问题描述如下：100名编号从1到100的囚徒，面临一个生死挑战。一个房间内有100个抽屉，每个抽屉里随机放入了一位囚徒的编号。每位囚徒可以打开最多50个抽屉，且必须在这些抽屉中找到写有自己编号的纸条。所有囚徒都必须成功找到自己的编号，他们才能被集体释放；只要有一人失败，则全体失败。囚徒们可以在挑战开始前商定一个策略，但在进入房间后不能有任何交流。

2. 算法说明

实现包括两个策略，分别是**随机选择策略**和**循环跟踪策略**，接下来进行详细阐述

2.1. 随机选择策略 (Random Strategy)

2.1.1. 核心思想

该策略最直观朴素：每位囚徒进入房间后，从所有抽屉中完全随机地选择 `k` 个进行查看。囚徒之间的选择是相互独立的。

2.1.2. 函数参数定义

函数 `simulate_random_strategy(n, k, t, seed)` 的参数如下：

- `n: int`: 囚犯与抽屉的数量。
- `k: int`: 每位囚犯允许尝试的最大次数。
- `t: int`: 独立模拟实验的总轮次。
- `seed: int | None`: 随机数生成器的种子，用于保证实验结果的可复现性。

2.1.3. 详细算法逻辑

1. **初始化**:
    - 根据可选的 `seed` 初始化一个 `numpy.random.default_rng` 实例，用于后续所有随机数生成。
    - 初始化成功计数器 `success` 为 0。
2. **主循环**:
    - 程序进入一个循环，总共执行 `t` 轮独立的模拟。
3. **单轮模拟**:
    - **布置房间**: 使用 `rng.permutation(n)` 生成一个从0到 `n-1` 的随机排列 `boxes`，代表每个抽屉中存放的囚徒编号。
    - **囚徒尝试**: 程序进入一个内层循环，按顺序模拟每位囚徒的尝试。
        - 对于编号为 `prisoner` 的囚徒，程序调用 `rng.choice(n, k, replace=False)`，从所有 `n` 个抽屉的索引中，无放回地随机抽取 `k` 个。

        - 检查 `prisoner` 的编号是否存在于这 `k` 个被选中的抽屉所对应的 `boxes` 内容中，即 `prisoner in boxes[chosen_indices]`。
        - **失败判断**: 如果 `prisoner` 不在其中，意味着该囚徒失败。本轮模拟立即宣告失败，程序通过 `break` 语句跳出囚徒循环，进入下一轮模拟。
    - **成功判断**: 如果囚徒循环正常完成（未被 `break`），则意味着所有 `n` 名囚徒都成功找到了自己的编号。`else` 从句被执行，成功计数器 `success` 加1。

```python
# 单轮模拟的核心代码
boxes = rng.permutation(n)
for prisoner in range(n):
    if prisoner not in boxes[rng.choice(n, k, replace=False)]:
        break
else:
    success += 1
```

4. **返回结果**:
    - 在 `t` 轮模拟全部结束后，函数返回总成功率 `success / t`。

2.2. 循环跟踪策略 (Cycle-Following Strategy)

2.2.1. 核心思想

这是一种经过精心设计的策略。它利用了房间内抽屉（索引）与囚徒编号（内容）之间构成的数学结构——**置换 (Permutation)**。一个置换可以被分解为若干个不相交的循环。

策略规定：每位囚徒首先打开与自己编号相同的抽屉，然后根据抽屉里的号码，去打开对应号码的抽屉，如此往复，直到找到自己的号码。这个过程本质上是在追踪囚徒本人所在的那个置换循环。集体成功的条件等价于该置换中不存在长度超过 `k` 的循环。

2.2.2. 函数参数定义

函数 `simulate_cycle_strategy(n, k, t, seed)` 的参数如下：

- `n: int`: 囚犯与抽屉的数量。
- `k: int`: 每位囚犯允许尝试的最大次数。
- `t: int`: 独立模拟实验的总轮次。
- `seed: int | None`: 随机数生成器的种子，用于保证实验结果的可复现性。

2.2.3. 详细算法逻辑

1. **初始化**:
    - 同样初始化随机数生成器 `rng` 和成功计数器 `success`。
    - 额外创建一个 `numpy` 数组 `max_cycles`，用于记录每轮模拟中发现的最长循环长度。
2. **主循环**:
    - 程序进入一个循环，总共执行 `t` 轮独立的模拟。
3. **单轮模拟**:
    - **生成置换**: 使用 `rng.permutation(n)` 生成一个随机置换 `perm`。
    - **计算最长循环**: 调用辅助函数 `_max_cycle_length(perm)` 计算出该置换中的最长循环长度 `m`。
    - **记录数据**: 将得到的 `m` 存入 `max_cycles` 数组，用于后续的统计分析和绘图。

- **成功判断**: 检查 `m` 是否小于或等于 `k`。如果是，则本轮模拟成功，`success` 计数器加1。

```
# 单轮模拟的核心代码
perm = rng.permutation(n)
m = _max_cycle_length(perm)
if m <= k:
    success += 1
```

4. **返回结果**:
    - 在 `t` 轮模拟结束后，函数返回两个值：总成功率 `success / t` 和记录了所有轮次最长循环长度的 `max_cycles` 数组。

2.2.4. 辅助函数 `_max_cycle_length`

此函数是循环策略的核心，用于寻找一个置换中的最长循环。

1. **初始化**:
    - 获取置换长度 `n`。
    - 创建一个长度为 `n` 的布尔数组 `visited`，所有元素初始化为 `False`，用于标记一个元素是否已被访问（即是否已属于某个被追踪过的循环）。
    - 初始化最长循环记录 `longest` 为0。
2. **遍历与追踪**:
    - 遍历所有元素索引 `start`（从0到 `n-1`）。
    - 对于每个 `start`，首先检查 `visited[start]`。如果为 `True`，说明它所属的循环已被计算过，直接跳过。
    - 如果为 `False`，则从 `start` 开始追踪一个新循环：
        - 初始化当前循环长度 `length` 为0，设置当前位置 `current` 为 `start`。
        - 进入一个 `while` 循环，条件是 `not visited[current]`。
        - 在循环内部：将 `visited[current]` 置为 `True`；将 `current` 更新为 `perm[current]`（即沿着置换前进一格）；`length` 加1。
        - 当 `while` 循环结束时，说明已回到该循环的起点，一个完整的循环已被追踪完毕。
    - 将当前计算出的 `length` 与 `longest` 进行比较，并更新 `longest` 为两者中的较大值。

```
# 循环追踪的核心代码
if not visited[start]:
    current = start
    length = 0
    while not visited[current]:
        visited[current] = True
        current = perm[current]
        length += 1
    longest = max(longest, length)
```

3. 实验结果与分析

3.1. 成功率对比

脚本不仅可以自己设置 `N` 和 `K` 的值进行模拟，还提供了不同的 `N` 和 `K` 值进行模拟的结果，展示如下：

N	K	循环策略成功率 (%)	随机策略成功率 (%)
50	25	~31.4	0.00
100	50	~31.1	0.00
150	75	~30.9	0.00
200	100	~31.3	0.00

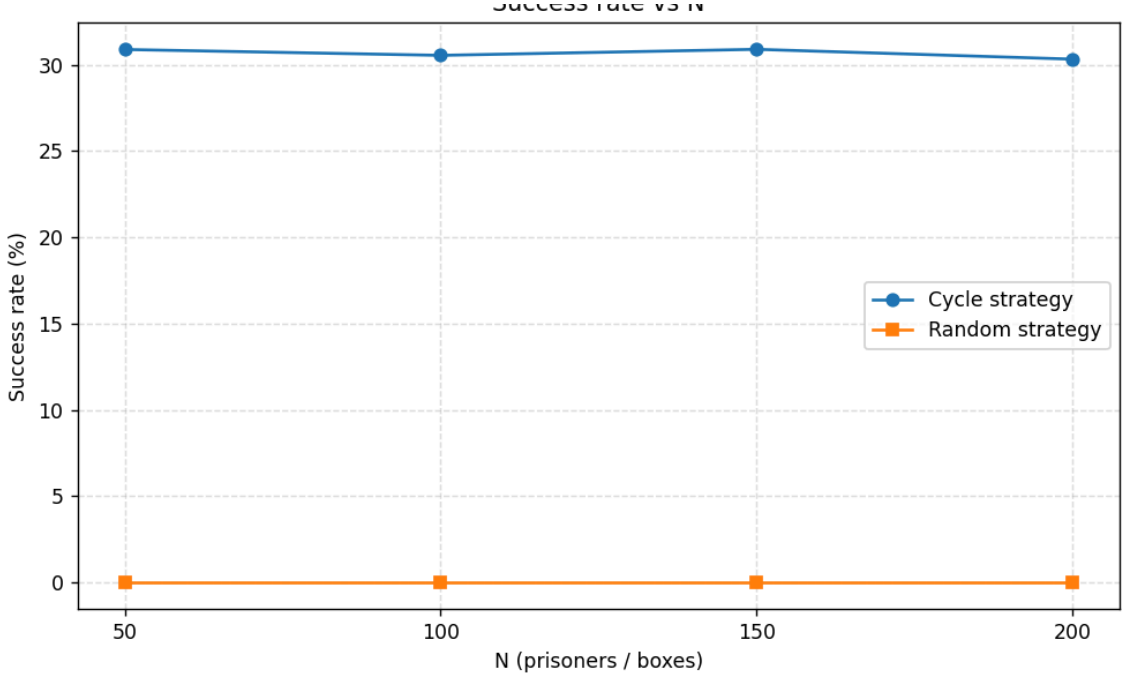
从表格数据可以清晰地看到：

- **随机策略**的成功率在所有规模下均为0.00%。这与理论预期相符，即其成功概率小到在10,000次模拟中几乎不可能出现一次成功。
- **循环策略**的成功率稳定在31%左右，并且不随囚徒总数N的增加而显著下降。这有力地证明了该策略的有效性和鲁棒性。

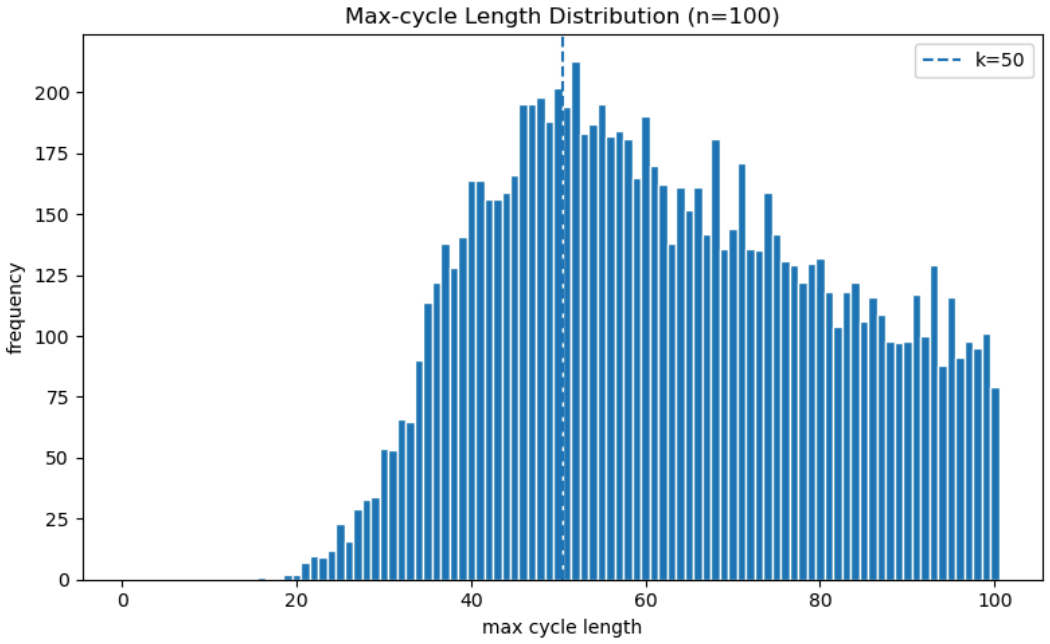
3.2. 结果可视化

脚本能够生成两种类型的图表，进一步揭示了策略的内在特性。

- **成功率对N的变化曲线**：该图（由`batch_simulation`生成）将不同N值下的两种策略成功率绘制成折线图。从图中可以更直观地看到，循环策略的成功率曲线几乎是一条水平线，稳定在30%以上；而随机策略的成功率曲线则一直紧贴x轴，几乎为零。



- **最大循环长度分布直方图**：该图（由`plot_cycle_distribution`生成）展示了在大量随机置换中，最长循环长度的分布情况。对于N=100的模拟，我们会看到分布的大部分集中在50以下，只有一小部分情况的最长循环超过50。



4. 代码结构与优化思路

4.1. 算法层面

脚本中`simulate_random_strategy`函数的实现存在显著的性能瓶颈。在每一轮模拟中，它为`n`个囚徒中的每一位都调用一次`rng.choice(n, k, replace=False)`来模拟选择抽屉。这意味着对于一次`t`轮的模拟，`rng.choice`被调用了`t * n`次。当`n`和`t`很大时，这会非常耗时。

- **优化方案**:

  将“每个囚犯是否找到自己的编号”的判断改为向量化操作：

```
success = all(prisoner in boxes[choices[prisoner]] for prisoner in range(n))
```

4.2. 并行计算

当前模拟的核心瓶颈在于`for i in range(t)`这个主循环。每一轮模拟都是完全独立的。可以利用Python的`multiprocessing`库将`t`轮模拟任务分配到多个CPU核心上同时执行，从而成倍地提升计算速度。

- **优化方案**:

  1. 将单轮模拟的逻辑封装成一个独立的函数，如`run_single_cycle_trial(seed)`。该函数接收一个随机种子，执行一轮模拟并返回结果。
  2. 在主模拟函数中，创建一个`multiprocessing.Pool`对象。
  3. 使用`pool.map()`方法，将`run_single_cycle_trial`函数应用到一个包含`t`个不同随机种子的列表上。
  4. 最后，汇总所有并行任务返回的结果，计算最终的成功率。

通过这种方式，如果计算机有`m`个核心，理论上可以将模拟时间缩短到原来的`1/m`左右，这对于需要进行数百万甚至更多轮次的高精度模拟至关重要。