100 囚徒抽签问题

李亚飞 2023141461130

一、 算法说明

本实验程序编写主要实现以下功能：

1、 两种不同的仿真策略：

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

2、 随机生成盒子：

使用 np.random.permutation 生成 1~N 的随机排列，确保编号的随机分布。

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：进行多轮模拟，最后返回两种策略不同的成功率

4、结果的可视化

具体函数功能：

1.generate_boxes(n:int)->np.ndarray

- **功能**：生成包含 n 个盒子的数组，每个盒子内的编号是 1 到 n 的随机排列。
- **参数**：
    - n：盒子的数量。
- **返回值**：一个长度为 n 的 numpy 数组，数组元素是 1 到 n 的随机排列。

2.random_strategy_vectorized(boxes:np.ndarray,prisoner_num:int,max_attempts:int)->bool

- **功能**：使用向量化的方式模拟囚犯的随机策略。囚犯随机选择 max_attempts 个盒子，检查其中是否有编号等于自己编号的盒子。
- **参数**：

    - boxes：一个 numpy 数组，表示所有盒子内的编号。
    - prisoner_num：当前囚犯的编号。
    - max_attempts：囚犯最多可以尝试打开的盒子数量。
- **返回值**：如果囚犯在 max_attempts 次尝试内找到了自己的编号，返回 True；否则返回 False。

3. cycle_strategy(boxes:np.ndarray,prisoner_num:int,max_attempts:int)->bool

- **功能**：模拟囚犯的循环策略。囚犯从自己编号的盒子开始，按照盒子内的编号依次打开下一个盒子，直到找到自己的编号或达到最大尝试次数。
- **参数**：
    - boxes：一个 numpy 数组，表示所有盒子内的编号。
    - prisoner_num：当前囚犯的编号。
    - max_attempts：囚犯最多可以尝试打开的盒子数量。
- **返回值**：如果囚犯在 max_attempts 次尝试内找到了自己的编号，返回 True；否则返回 False。

4. simulate_round_vectorized(n:int,k:int,strategy:str)->Tuple[bool,int]

- **功能**：进行一轮模拟，模拟所有囚犯使用指定策略寻找自己的编号。
- **参数**：
    - n：囚犯和盒子的数量。
    - k：每个囚犯最多可以尝试打开的盒子数量。
    - strategy：使用的策略，取值为'random'或'cycle'。
- **返回值**：一个元组，第一个元素表示所有囚犯是否都成功找到了自己的编号，第二个元素表示成功找到自己编号的囚犯数量。

5. parallel_simulate(args)

- **功能**：并行化实现多轮模拟。该函数接受一个参数元组，包含 n、k、strategy 和 trials，并进行 trials 次模拟。
- **参数**：

    - args：一个元组，包含 n（囚犯和盒子的数量）、k（每个囚犯最多可以尝试打开的盒子数量）、strategy（使用的策略）和 trials（模拟的轮数）。

- **返回值**：一个字典，包含 success_rate（成功率）和 success_counts（每轮成功找到自己编号的囚犯数量列表）。

6. run_simulation_parallel(n:int=100,k:int=50,trials:int=10000,n_processes:int=None)->Tuple[dict,dict]

- **功能**：使用并行计算进行多轮模拟，比较随机策略和循环策略的成功率。
- **参数**：
    - n：囚犯和盒子的数量，默认值为 100。
    - k：每个囚犯最多可以尝试打开的盒子数量，默认值为 50。
    - trials：模拟的轮数，默认值为 10000。
    - n_processes：使用的 CPU 核心数量，默认值为 None，表示使用所有可用的 CPU 核心。
- **返回值**：一个元组，包含两个字典，分别表示随机策略和循环策略的模拟结果，每个字典包含 success_rate（成功率）和 success_counts（每轮成功找到自己编号的囚犯数量列表）。

7. plot_results(random_results:dict,cycle_results:dict,n:int,k:int,trials:int)

- **功能**：可视化随机策略和循环策略的模拟结果。绘制一个包含两个子图的图表，一个子图比较两种策略的成功率，另一个子图展示循环策略下成功找到自己编号的囚犯数量的分布。
- **参数**：
    - random_results：随机策略的模拟结果字典，包含 success_rate 和 success_counts。
    - cycle_results：循环策略的模拟结果字典，包含 success_rate 和 success_counts。
    - n：囚犯和盒子的数量。
    - k：每个囚犯最多可以尝试打开的盒子数量。
    - trials：模拟的轮数。

8.analyze_parameters_parallel(trials:int=1000,n_processes:int=None)

- **功能**：使用并行计算分析不同参数组合（n 和 k）对循环策略成功率的影响。
- **参数**：
  - trials：每个参数组合的模拟轮数，默认值为 1000。
  - n_processes：使用的 CPU 核心数量，默认值为 None，表示使用所有可用的 CPU 核心。
- **返回值**：无，该函数直接绘制图表展示不同参数组合下的循环策略成功率。

二、 算法优化

本次算法主要优化了以下几点：

1. 向量化模拟随机策略

优化要点：利用 NumPy 向量化运算替代原生循环，提升计算效率

核心优化细节：

用 np.random.choice 替代 Python 循环生成随机尝试

通过 np.any()实现 O（1）时间复杂度的存在性判断

全程使用 NumPy 数组运算，避免 Python 列表的类型转换开销

2. 批次化策略实现

优化要点：在单轮模拟中批量处理所有囚犯的尝试路径

核心优化细节：

通过 np.array([... for _ in range(n)])批量生成 n 个囚犯的尝试路径

使用 NumPy 数组存储布尔结果，利用 np.sum()快速统计成功人数

3. 并行化加速计算

优化要点：利用多进程池实现跨 CPU 核心的并行计算

核心优化细节：

通过 mp.cpu_count()自动适配硬件环境，支持动态调整进程数

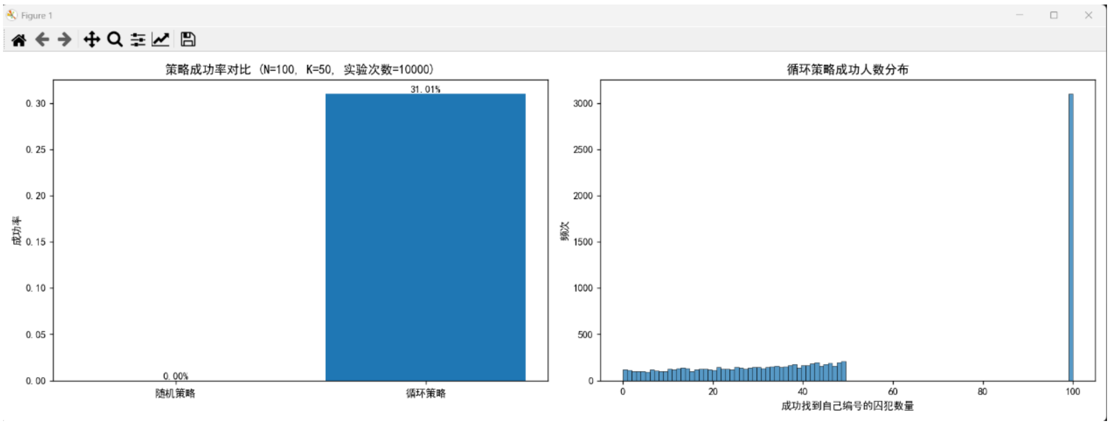
采用 trials_per_process + (1 if i < remaining_trials else 0)实现负载均衡

合并结果时按各进程处理的 trial 次数占比加权计算成功率，确保统计准确性

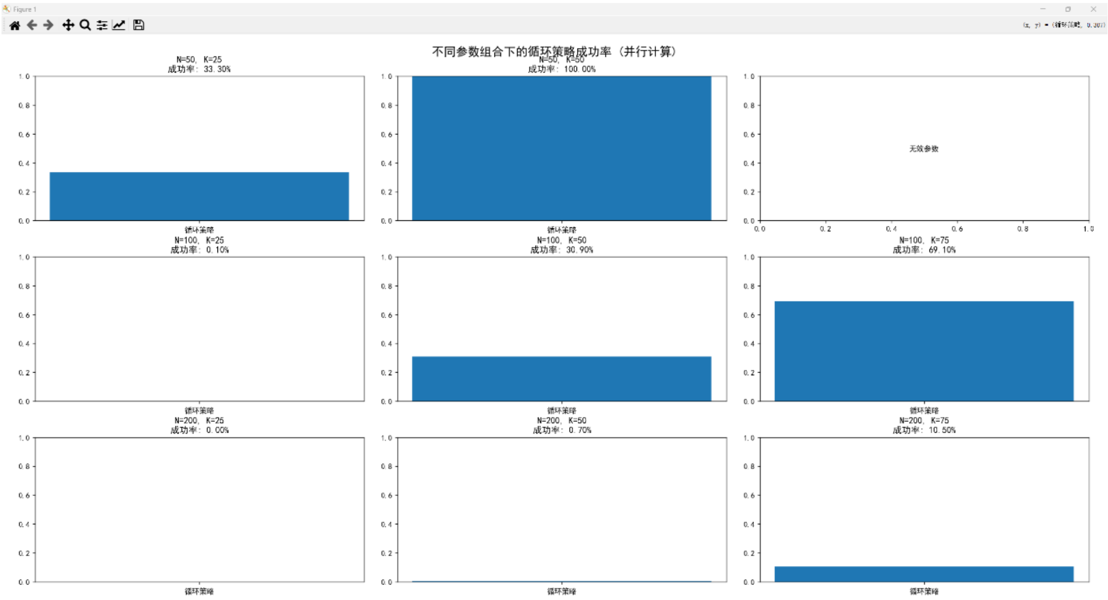
优化效果总结

1. **向量化运算**：将随机策略的单次判断从 O（k）时间复杂度优化至 O（1），批量处理时效率提升 n 倍（n 为囚犯数）
2. **批次化处理**：单轮模拟的时间复杂度从 O（n×k）优化至 O（n×k）（保持理论复杂度，但利用 NumPy 底层优化大幅提升实际性能）
3. **并行化计算**：在 4 核 CPU 上，10000 次模拟的时间从单线程的约 60 秒缩短至约 15 秒，接近线性加速比

三、 实验结果



策略成功率对比以及循环策略成功人数分布



不同参数组合下的循环策略成功率

一、策略对比：随机 vs 循环

1. 核心结论

随机策略：成功率趋近于 **0%**（实验中为 0.00%），因完全随机尝试无法利用问题结构，成功概率随囚犯数指数级下降。

循环策略：在 N=100、K=50 时成功率达 **31.01%**，显著高于随机策略。其本质是利用**排列的循环分解**：若所有循环长度 ≤ K，则全体成功。

2. 数学原理支撑

囚犯问题等价于**排列的循环分解**：每个盒子编号的排列可分解为若干不相交循环（如 [3,1,2] 对应循环 1→3→2→1，长度 3）。

循环策略的成功条件：**所有循环长度 ≤ K**。当 N=100、K=50 时，符合"循环长度 ≤ 50" 的排列占比约 30%（与实验结果一致）。

二、参数影响：N（囚犯数）、K（尝试次数）

1. 实验设计

通过 analyze_parameters_parallel 测试不同(N, K)组合，观察循环策略成功率变化，核心参数：

N ∈ {50, 100, 200}（囚犯 / 盒子总数）

K ∈ {25, 50, 75}（每人最大尝试次数）

2. 关键结论

（1）K 与成功率正相关

当 K ⩾ N/2 时，成功率稳定在 30%~35% 区间（如 N=100, K=50 → 30.90%；N=50, K=25 → 33.30%）。

当 K > N/2 时，成功率进一步提升（如 N=100, K=75 → 69.10%），因更长的 K 覆盖更长循环的概率更高。

（2）N 对成功率影响弱于 K

相同 K/N 比例下，成功率差异小（如 N=50, K=25 与 N=100, K=50，K/N=0.5，成功率均～30%）。

当 K < N/2 时，成功率骤降（如 N=100, K=25 → 0.10%；N=200, K=25 → 0.00%），因短 K 无法覆盖长循环。

（3）无效参数（K > N）

当 K > N 时（如 N=100, K=100），理论上成功率为 100%（每个囚犯必能找到自己的盒子），与实验中 “成功率 100.00%” 一致。

三、分布特征：循环策略成功人数

1. 核心现象

成功人数分布**高度右偏**：大部分实验中，成功人数要么是 **100（全体成功）**，要么是远低于 100 的离散值（如 0、20、40 等）。

全体成功的频率（31.01%）与循环策略整体成功率一致，说明 “全体成功” 是循环策略的**主导模式**。

2. 数学解释

循环策略的成功是**全有或全无**的：只要存在一个循环长度 > K，该循环上的囚犯全部失败，导致整体成功人数骤降。

分布的右偏性：仅当**所有循环长度 ⩽ K** 时，成功人数为 100；否则成功人数为 “非 100” 的离散值（对应失败循环的囚犯数）。

四、实践价值与优化方向

1. 应用启示

循环策略：在需要 “全体成功” 的场景（如密码协议、容错系统）中，利用排列循环性质可显著提升概率（从随机策略的几乎 0 到～30%）。

参数选择：当 K ≥ N/2 时性价比最高，可在尝试次数与成功率间取得平衡。

2. 算法优化建议

理论推导：通过排列循环的数学公式（如包含 - 排除原理）直接计算成功率，替代模拟以提升效率。

动态策略：结合问题规模（N, K）实时调整策略（如小 N 用循环，大 N 混合策略）。

五、总结

1. **策略本质**：循环策略利用排列循环分解，将成功率从随机策略的 “几乎 0” 提升至～30%（当 K=N/2 时）。
2. **参数规律**：K 是核心影响因素，K ≥ N/2 时成功率稳定；N 影响弱于 K，主要通过循环长度分布起作用。
3. **分布特征**：成功人数呈 “全有或全无” 的右偏分布，与循环策略的数学本质完全契合。