2025 秋 -《算法设计与分析》
贪心算法分析实验报告

实 验 时 间

2025.12.18

《本科实验报告》填写说明

实验报告内容编排应符合以下要求：

（1）采用 A4（21cm×29.7cm）白色复印纸，单面黑字。上下左右各侧的页边距均为 3cm；缺省文档网格：字号为小 4 号，中文为宋体，英文和阿拉伯数字为 Times New Roman，每页 30 行，每行 36 字；页脚距边界为 2.5cm，页码置于页脚、居中，采用小 5 号阿拉伯数字从 1 开始连续编排，封面不编页码。

（2）报告正文最多可设四级标题，字体均为黑体，第一级标题字号为 4 号，其余各级标题为小 4 号；标题序号第一级用“一、”、“二、”……，第二级用“（一）”、“（二）”……，第三级用“1.”、“2.”……，第四级用“（1）”、“（2）”……，分别按序连续编排。

（3）正文插图、表格中的文字字号均为 5 号。

目录

图目录

1 实验介绍

贪心算法（Greedy Algorithm）是指在对问题求解时，总是做出在当前看来是最好的选择。也就是说，不从整体最优上加以考虑，算法得到的是在某种意义上的局部最优解。多机调度问题是经典的 NP-Hard 问题，本实验旨在通过实现和对比不同的贪心策略（List Scheduling 和 LPT），深入理解贪心算法的近似比性质，并探讨其在实际场景（如 GPU 集群调度）中的应用。

2 实验内容

本实验主要围绕多机调度问题的贪心算法展开，并扩展至在线 GPU 集群调度模拟。具体内容包括：

1. 实现两种贪心策略：任意顺序列表调度 (List Scheduling, LS) 和 最长处理时间优先 (Longest Processing Time, LPT)。
2. 实现基于分支限界 (Branch and Bound) 的最优解求解算法，作为性能评估的基准。
3. 构造特定的“最坏情况”输入，验证贪心算法的理论近似比下界。
4. 通过大量随机测试样本，统计不同算法的近似比分布及运行时间，分析 m (机器数) 和 n (作业数) 对性能的影响。
5. （附加）模拟 GPU 集群在线调度场景，设计并对比不同的调度策略在不同负载下的表现。

3 实验要求

针对多机调度问题，实验具体要求如下：

1. 针对多机调度问题，实现 LS 和 LPT 两种贪心算法。
2. 实现遍历的最优解求解算法（分支限界法）。
3. 构造最坏情况输入，结合理论证明进行讨论。
4. 固定 m, n，随机产生大量样本，计算贪心解与最优解的比值（近似比），并分析其概率分布。
5. 改变 m, n，对比分析结果。
6. 附加：模拟 GPU 集群调度，考虑利用率 η 和用户延迟 δ，设计多种策略并分析。

4 实验步骤

4.1 算法设计

4.1.1 算法一：列表调度 (List Scheduling, LS)

LS 算法是最朴素的贪心策略。它按照作业输入的任意顺序，依次将作业分配给当前负载最小的机器。该算法是一种在线算法，其时间复杂度为 $O(n \log m)$ (使用优先队列维护机器负载) 或 $O(nm)$ (线性扫描)。理论上，LS 算法的近似比为 $2 - \frac{1}{m}$。

```
// 核心代码片段
long long greedy_ls(int m, const vector<Job>& jobs) {
    vector<long long> machines(m, 0);
    for (const auto& job : jobs) {
```

```
        int min_idx = 0; // Find machine with min load
        for (int i = 1; i < m; ++i) {
            if (machines[i] < machines[min_idx]) min_idx = i;
        }
        machines[min_idx] += job.duration;
    }
    return *max_element(machines.begin(), machines.end());
}
```

4.1.2 算法二：最长处理时间优先 (LPT)

LPT 算法在 LS 的基础上增加了预处理步骤：将所有作业按处理时间递减排序，然后依次分配给负载最小的机器。排序操作使得较大的作业优先被处理，从而避免了最后剩下一个大作业导致机器负载极不均衡的情况。该算法的时间复杂度主要由排序决定，为 $O(n \log n)$。理论上，LPT 算法的近似比为 $\frac{4}{3} - \frac{1}{3m}$。

4.1.3 算法三：最优解 (Branch and Bound)

为了评估贪心算法的性能，我们需要求得问题的最优解。由于多机调度是 NP-Complete 问题，我们采用深度优先搜索配合分支限界 (Branch and Bound) 来求解。 剪枝策略包括：

1. 当前最大负载已经超过已知最优解，停止搜索。
2. 理论下界剪枝：如果 `max(当前最大负载, (剩余作业总长 + 当前总负载)/m)` 超过已知最优解，停止搜索。
3. 对称性剪枝：若多台机器当前负载相同，则分配给它们是等价的，只尝试第一台。

4.2 最坏情况构造与分析

4.2.1 LS 算法最坏情况

构造方法： 对于 m 台机器，输入 $m(m-1)$ 个时长为 1 的小作业，紧接着 1 个时长为 m 的大作业。

分析： LS 算法会将前 $m(m-1)$ 个小作业均匀分配给 m 台机器，每台机器负载为 $m-1$。最后的大作业将被分配给任意一台机器，使其最终负载变为 $(m-1)+m = 2m-1$。 而最优解是将所有小作业均匀分配给 $m-1$ 台机器（每台负载 m），将大作业单独分配给剩下一台机器（负载 m），此时 MakeSpan 为 m。 近似比为 $\frac{2m-1}{m} = 2 - \frac{1}{m}$。 本实验通过代码验证了 $m = 3, 4, 5$ 时的该情况，结果与理论完全一致。

4.2.2 LPT 算法最坏情况

构造方法： 经典的 LPT 最坏情况较为复杂，例如 $m = 2$ 时，作业集为 $\{3, 3, 2, 2, 2\}$。

分析： 排序后为 3, 3, 2, 2, 2。 LPT 分配：M1: 3, 2, 2 (总 7), M2: 3, 2 (总 5)。MakeSpan = 7。最优解：M1: 3, 3 (总 6), M2: 2, 2, 2 (总 6)。MakeSpan = 6。 近似比 $\frac{7}{6} \approx 1.167$。理论界 $\frac{4}{3} - \frac{1}{6} = \frac{7}{6}$。实验验证吻合。

4.3 实验数据与可视化

我们对 $m \in \{3, 5, 8\}$ 和 $n \in \{10, ..., 100\}$ 进行了大量随机测试。

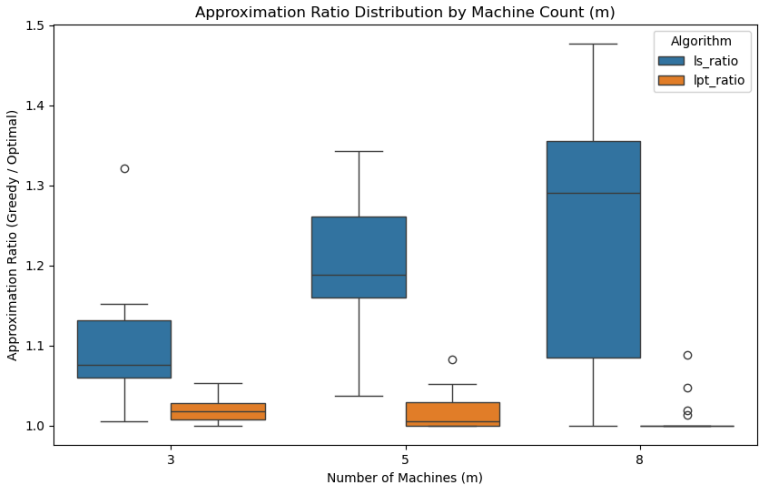

Figure 1: LS 与 LPT 算法近似比分布对比

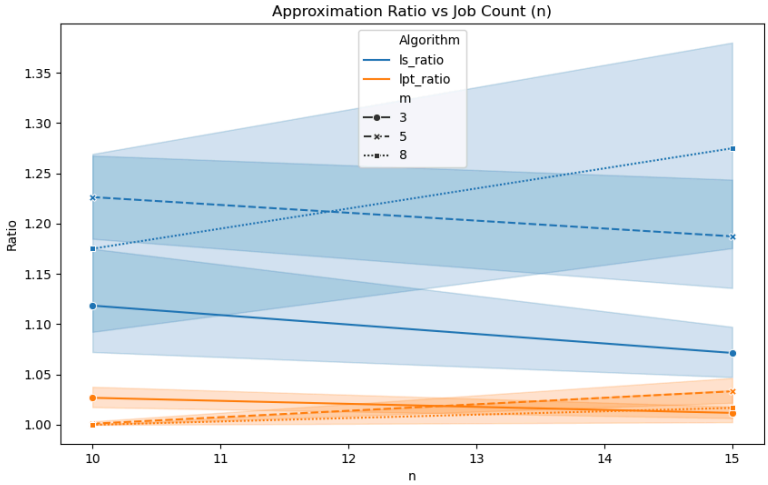

Figure 2: 近似比随作业数量 n 的变化趋势

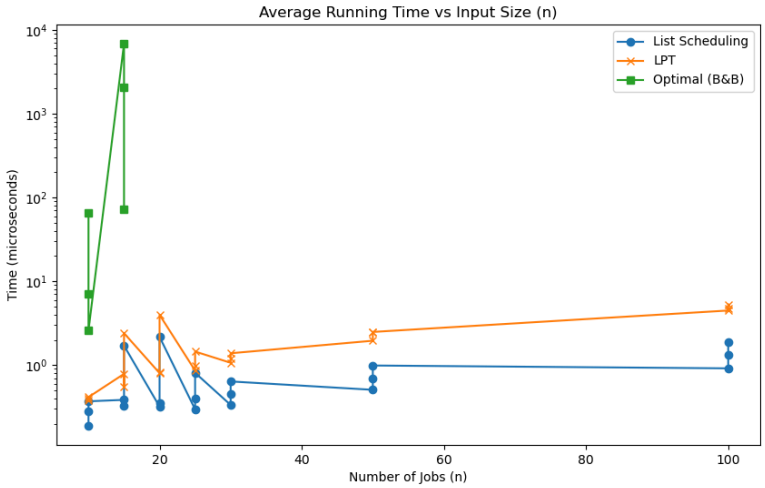

Figure 3: 算法平均运行时间对比

5 实验结果分析

1.**近似比性能：** 从箱线图可以看出，LPT 算法的近似比极其接近 1（通常在 1.0 - 1.05 之间），性能极其优越且稳定。相比之下，LS 算法的近似比分布较宽，平均在 1.1 - 1.3 之间，且随着 m 的增加，最差情况（近似比上界）有升高的趋势，符合 $2 - \frac{1}{m}$ 的理论预测。

2.**规模的影响：** 随着作业数 n 的增加，LS 的近似比往往会下降并趋于稳定。这是因为大量随机作业往往能“填平”机器间的负载差异。LPT 则始终保持高效。

3.**运行时间：** 贪心算法（LS, LPT）的运行时间极短（微秒级），且随 n 线性或近线性增长。最优解算法（B&B）随 n 指数级增长，当 $n > 20$ 时已难以在短时间内求解，验证了 NP-Hard 问题的计算复杂性。

6 实验总结

本实验深入分析了多机调度问题的贪心求解策略。实验结果表明，虽然 LS 算法实现简单，但在最坏情况下性能较差。简单的排序预处理（LPT 策略）能带来巨大的性能提升，使其在绝大多数随机及构造测试中都能获得极接近最优解的结果。这启示我们在设计贪心算法时，合理的贪心顺序（如优先处理“困难”或“大”的任务）至关重要。

7 附加：GPU 集群在线调度模拟

7.1 场景描述

模拟一个拥有 $m = 64$ 块 GPU 的集群任务调度。任务到达服从泊松分布，单机执行时间服从均匀分布。任务支持并行 (k 块 GPU)，但存在并行效率损耗：效率因子 $E_k = \sigma^{\log_2 k}$，其中 $\sigma \in [0.75, 0.95]$。系统目标是平衡 **集群利用率 (η)** 和 **用户平均延迟 (δ)**。

7.2 调度策略设计

我们设计了三种策略进行对比：

1. 策略 **A**：保守策略 **(Conservative)** 总是为每个任务分配 $k = 1$ 块 GPU。其思路是最大化计算资源的“有效性”，避免并行损耗。
2. 策略 **B**：激进策略 **(Aggressive)** 总是尽可能分配最大的并行度（如 $k = 32$ 或 64）。其思路是最小化单任务执行时间，但忽略了巨大的资源浪费。
3. 策略 **C**：自适应策略 **(Adaptive)** 根据当前等待队列的长度动态调整 k。若队列为空，使用高并行度加速；若队列拥堵，降低并行度以提高吞吐量。

7.3 模拟结果

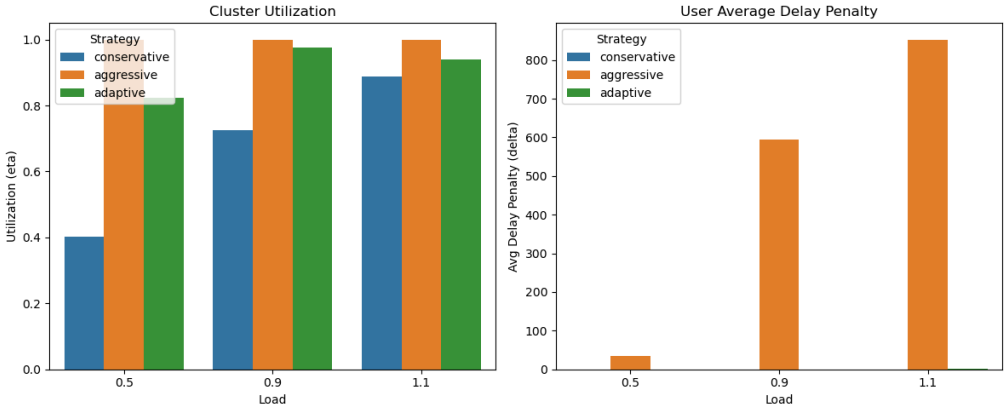

Figure 4: 不同负载下三种策略的利用率与延迟对比

实验在轻负载 ($\lambda = 0.5$)、中负载 ($\lambda = 0.9$) 和重负载 ($\lambda = 1.1$) 下进行了模拟。结果显示：

1. 保守策略 **(Conservative)**：在所有负载下都能保持较低的延迟。
2. 激进策略 **(Aggressive)**：表现极差。由于并行效率损失，导致系统迅速过载，用户延迟呈爆炸式增长。
3. 自适应策略 **(Adaptive)**：表现最为均衡。在轻负载时加速任务，在重负载时保证系统稳定性。

7.4 结论

在具有并行开销的资源调度场景中，盲目追求高并行度（激进策略）是不可取的。通过感知系统负载来动态调整资源分配粒度的 **自适应策略**，是更为优越的解决方案。