**2024年全国大学生信息存储技术竞赛**

**设计文档**

**赛题名称： 磁电盘IO最优调度**

**队伍名称： 曼城足球队**

**电子邮箱： 3260797806@qq.com**

**提交日期： 2024.10.18**

填写说明

1. 所有参赛项目必须为一个基本完整的设计。设计文档旨在能够清晰准确地阐述该参赛队的参赛方案。

2. 设计文档采用A4纸撰写。除标题外，所有内容必需为宋体、小四号字、1.5倍行距。

3. 设计文档中各项目说明文字部分仅供参考，设计文档撰写完毕后，请删除所有说明文字。(本页不删除)

4. 设计文档模板里已经列的内容仅供参考，作者可以在此基础上增加内容或对文档结构进行微调。

5. 为保证网评的公平、公正，设计文档中应避免出现作者所在学校、院系和指导教师等泄露身份的信息。

# 第一章 方案设计

**1.1 问题抽象**

对问题进行整体分析可知，问题要求输入当前的磁头位置HeadInfo与一组IO请求IOUint，输出按寻址方案排序后的IO请求ID。由于IO分布在Wrap上的一定区域内，可以将其抽象为一组点，两点之间边的权值为寻址代价，即图模型。

考虑到相邻Wrap上数据排布方向相反，而数据读取必须沿排布方向，且调头操作会造成时间上的消耗，故磁头从IO请求a到IO请求b的寻址时间与从b到a的寻址时间并不相同。因此，磁头与IO分布所抽象成的图模型是有向带权图，任意两点间都有两条方向相反的边，并且这两条边权值不同。

磁头从起始位置出发，按一定顺序访问磁盘上所有IO请求所在数据段，不能重复访问，不要求回到起始位置，且要求磁头访问完毕后所用总寻址时间最小，故该问题可以简化为开环、不对称的旅行商问题（ATSP）。

**1.2 设计原理**

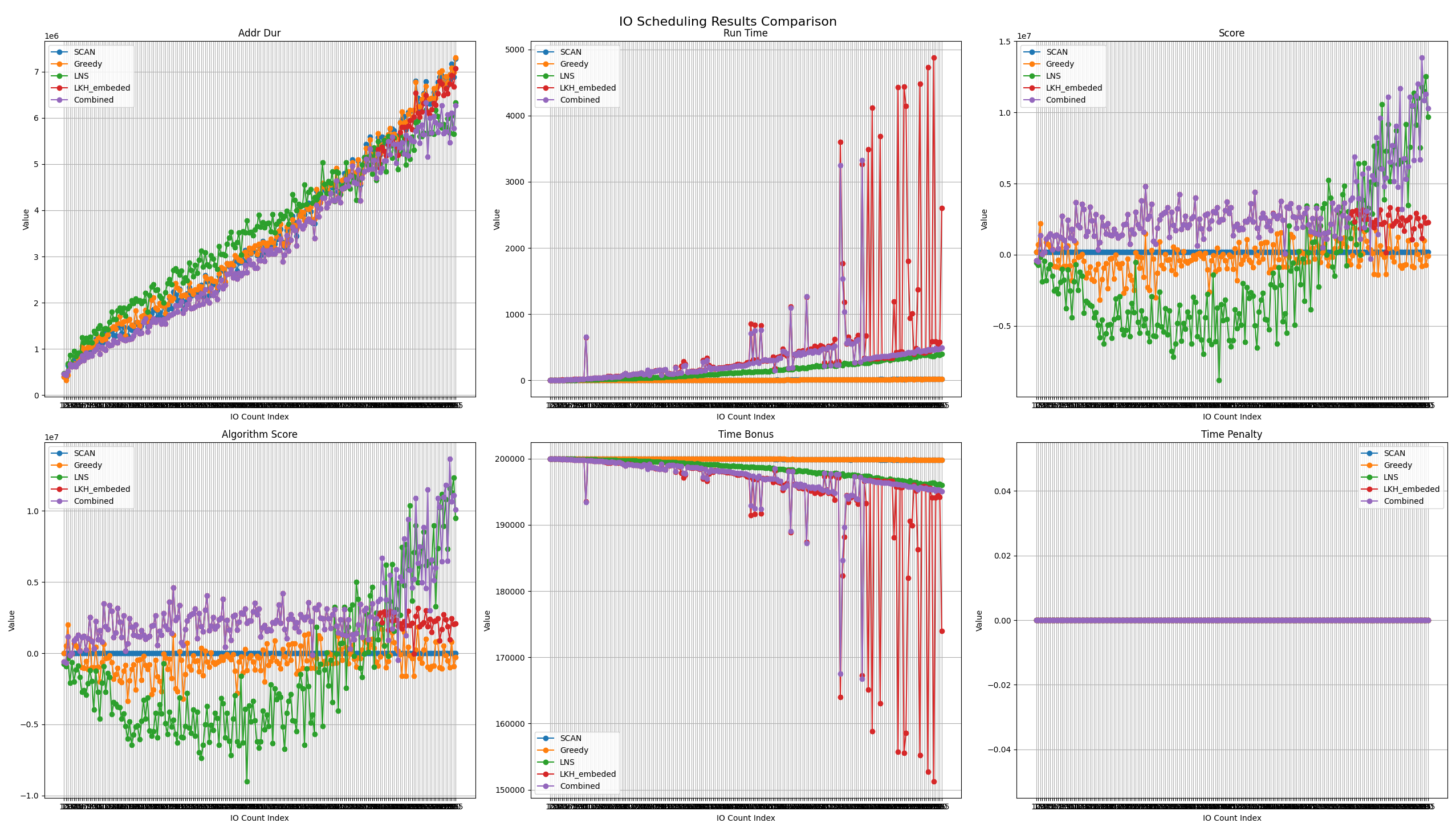
查阅资料发现，邻域搜索（Neighborhood Search）是一类十分常见的改进算法，其在每次迭代时通过搜索当前解的“邻域”以获取更优解。根据经验，邻域越大，局部最优解越好，以此获得的全局最优解也越好。由此，我们选择该算法类中的大规模邻域搜索作为基本方案。

大规模邻域搜索（Large Neighborhood Search, LNS）是一种强大的启发式搜索算法，在组合优化问题，尤其是TSP问题上表现出色。其基本思想是在邻域搜索的基础上采用大型邻域结构，以快速发现高质量的解。具体而言，LNS采用了“破坏”（Destroy）与“修复”（Repair）的策略，将问题分解为子问题，并在子问题上应用不同的搜索策略，以寻找更优解。一般而言，LNS的效果在很大程度上依赖于破坏与修复策略的选择：破坏策略需要足够大胆，以确保能够跳出局部最优解，并探索解空间中更大的区域；修复策略需要足够聪明，以有效地重建解，以寻找更优的解决方案。

初始解的质量对于LNS的性能至关重要，一个好的初始解可以加速算法收敛，而一个差的初始解会消耗更多搜索时间。经过大量探索，我们发现贪心算法运行速度较快且效果在Baseline之上，因此，我们**使用随机贪心算法生成多个解**，并从中选择最优解作为初始解。

对于破坏得到的局部解空间，我们**使用2-opt算法进行局部搜索优化**，通过反复交换路径中的两条边来寻找更短的路径。除此之外，在得到局部最优解后，我们**加入模拟退火机制以避免陷入局部最优**，尽可能地寻找全局最优解。

**1.3 作品效果展示**

****

(紫色-10至1000IO数-最终作品算法)

图表, 直方图

描述已自动生成

(紫色-1000至10000IO数-最终作品算法)

# 第二章 关键代码说明

**2.1 编程语言、编程环境**

编程使用C语言

编程环境：Ubuntu 18.04, cmake 3.27.5, make 4.1, GCC 7.5

**2.2 算法实现**

**2.2.1 LNS算法**

**1.数据结构**

主要使用了以下数据结构:

typedef struct {

    uint32\_t id;

    bool visited;

} IORequest;

这个结构用于表示IO请求,包含请求ID和是否已访问的标志。

**2.随机贪心算法**

1. 初始化:

* 将所有IO请求标记为未访问设置初始磁头位置。

1. 主循环:

* 遍历所有未访问的请求，计算每个请求到当前磁头位置的代价。选择代价最小的前3个候选请求。随机从这3个候选中选择一个，将选中的请求添加到序列中，并标记为已访问。更新当前磁头位置，重复直到所有请求都被访问。

1. 计算总代价:

* 使用calculateTotalCost函数计算生成序列的总代价。

1. 核心代码

int32\_t chosen = candidates[rand() % (remainingRequests < 3 ? remainingRequests : 3)];

        sequence[currentIndex++] = chosen;

        requests[chosen].visited = true;

        IOUint \*chosen\_io = &globalInput->ioVec.ioArray[chosen];

**3.大规模邻域搜索**

这个函数用于优化初始解:

1. 初始化:

* 设置破坏规模(destroySize)为总请求数的10%
* 设置迭代次数(LNS\_ITERATIONS)

1. 主循环(每次迭代):

* 复制当前序列到临时序列
* 随机选择要破坏的子序列(start到end)
* 使用贪心算法重建子序列:
  + 遍历子序列中的每个位置
  + 计算剩余未放置请求到当前位置的代价
  + 选择代价最小的请求放入当前位置
* 计算新序列的总代价
* 使用模拟退火策略决定是否接受新解:
  + 如果新解更好,直接接受
  + 如果新解较差,以一定概率接受(概率随温度降低而减小)
* 如果接受新解,更新当前序列和代价
* 如果新解是最优解,更新最优序列和最优代价

1. 关键代码:

double temperature = SA\_INITIAL\_TEMP \* pow(SA\_COOLING\_RATE, iter);

        if (newCost < currentCost ||

            (double)rand() / RAND\_MAX < exp((currentCost - newCost) / temperature)) {

            for (uint32\_t i = 0; i < globalInput->ioVec.len; i++) {

                sequence[i] = tempSequence[i];

            }

            currentCost = newCost;

这段代码实现了模拟退火的核心逻辑,决定是否接受新解。

**4. 主算法流程(IOScheduleAlgorithmLNS函数)**

整个算法的主要流程如下:

a) 初始化参数:

* 设置初始解数量、迭代次数、初始温度和冷却率

b) 生成多个初始解:

* 循环调用randomGreedy函数生成多个初始解
* 选择代价最小的解作为最佳初始解

c) 应用大规模邻域搜索:

* 将最佳初始解设为当前解
* 调用largeNeighborhoodSearch函数进行优化

d) 输出结果:

* 将最优序列转换为输出格式
* 设置输出参数
* 关键代码:

// 生成多个初始解并选择最好的

    for (int i = 0; i < INITIAL\_SOLUTIONS; i++) {

        randomGreedy();

        if (currentCost < bestCost) {

            bestCost = currentCost;

            for (uint32\_t j = 0; j < input->ioVec.len; j++) {

                bestSequence[j] = sequence[j];

            }

        }

    }

    // 应用大规模邻域搜索

    for (uint32\_t i = 0; i < input->ioVec.len; i++) {

        sequence[i] = bestSequence[i];

    }

    currentCost = bestCost;

    largeNeighborhoodSearch();

这段代码展示了如何生成多个初始解并应用大规模邻域搜索。

**5.代价计算函数**

这个函数计算给定序列的总代价:

a) 初始化:

* 设置总代价为0
* 设置初始磁头位置

b) 遍历序列:

* 对于序列中的每个请求:
  + 计算从当前磁头位置到该请求位置的代价
  + 将代价加到总代价上
  + 更新当前磁头位置

c) 返回总代价

总结: 算法采用了随机贪心算法生成初始解,然后使用大规模邻域搜索结合模拟退火策略进行优化。通过多次生成初始解并反复优化,最终得到一个较优的IO请求处理序列,以最小化总的寻道时间和旋转延迟。

**2.2.2 LKH算法**

由于LNS算法在小规模数据集上有其局限性，需要对其中一部分路线进行破坏再重建，对小型数据集呈现负优化。所以在小规模数据集（io请求数小于800）中使用LKH算法。

**1.算法简介**

LKH算法在经典的Lin-Kernighan算法基础上引入了多种优化，使其在处理大规模TSP问题时表现更加优异。其核心思路包括以下几点：

候选边集（Candidate Set）：在每次k-opt操作中，算法不对所有边进行操作，而是先生成一个候选边集，该集合包含可能对优化有贡献的边。这些边通常是基于一些启发式规则（如最近邻或贪心规则）选择的，从而减少了搜索的复杂度。

动态调整k值：经典的Lin-Kernighan算法需要设定k的值，即删除的边数。在LKH中，k的值不是固定的，而是动态调整的。算法从较小的k开始，如果未能找到更优解，则增加k的值，从而允许算法更灵活地跳出局部最优解。

随机化和多次运行：为了避免陷入局部最优解，LKH通常会多次运行，使用不同的初始解来探索更大的解空间。初始解可以通过贪心算法、最近邻算法等方法生成。

使用有效的启发式规则：LKH结合了多种启发式方法来生成候选解集。例如，LKH使用Delaunay三角剖分和邻近边（nearest neighbors）等几何启发式方法来选择可能有助于优化的候选边。

有效的终止条件：LKH不仅通过局部搜索改进解，还引入了多种终止条件，例如基于解的收敛性、当前温度变化等，来有效地控制算法的运行时间。

2.算法流程

初始化：生成一个初始解，这通常使用简单的贪心算法或最近邻算法。

生成候选边集：基于启发式规则，为每个节点生成一个候选边集，存储该节点可能连接的若干条边。

k-opt局部搜索：动态选择k值，删除k条边，并重新连接路径。如果新的路径更短，则更新当前解。

多次迭代和重新启动：通过多次迭代和重新启动，算法会从不同的初始解开始，不断寻找更优的全局解。每次迭代结束时，保留找到的最优解。

终止：当经过多次迭代后没有明显的改进时，算法停止，返回最优解。

**3.关键代码**

do {

    while ((t1 = RemoveFirstActive())) {

        for (X2 = 1; X2 <= 2; X2++) {

            t2 = X2 == 1 ? PRED(t1) : SUCt1;

            G0 = C(t1, t2);

            do

                t2 = Swaps == 0 ? BestMove(t1, t2, &G0, &Gain) :

                    BestSubsequentMove(t1, t2, &G0, &Gain);

            while (t2);

            if (PenaltyGain > 0 || Gain > 0) {

                Cost -= Gain / Precision;

                StoreTour();

                Activate(t1);

                break;

            }

        }

    }

} while (PenaltyGain > 0 || Gain > 0);

外层循环通过RemoveFirstActive移除当前活跃的节点t1。对t1的前驱和后继节点尝试交换边。G0：存储当前边t1 -> t2的成本。BestMove：尝试寻找最优边交换路径，更新增益Gain。如果找到了改进路径，即增益Gain或罚金收益PenaltyGain大于0，则更新路径的成本，并标记节点为激活状态以便进一步优化。

if (HashSearch(HTable, Hash, Cost))

    goto End\_LinKernighan;

HashInsert(HTable, Hash, Cost);

哈希表查找：检查当前路径是否已经在哈希表中存在，如果存在，说明该路径已经处理过，跳出循环。哈希插入：如果当前路径是新的，则将其插入哈希表。

if (Gain23Used && ((Gain = Gain23()) > 0 || PenaltyGain > 0)) {

    Cost -= Gain / Precision;

    StoreTour();

}

非顺序4/5-opt：该部分使用更复杂的移动方式（如4-opt或5-opt）来寻找潜在的改进路径，允许更大范围的节点交换。找到改进时，同样更新总成本Cost并保存路径。

# 第三章 测试及分析

3.1 环境配置

华为云ECS鲲鹏通用计算增强型kc2，920 2vCPU、4GB内存

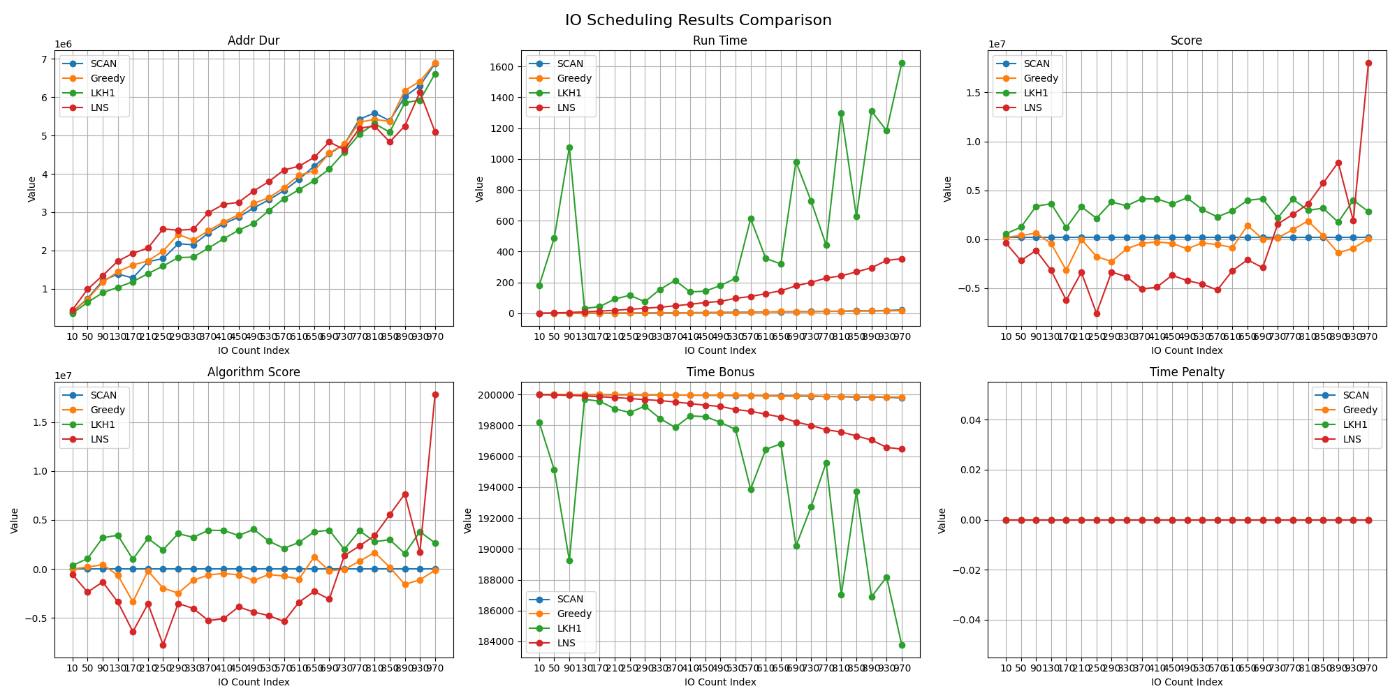
3.2 测试内容

测试主要检验的算法有：LKH、贪心算法、LNS、SCAN、先入先出五种算法

调用自行实现的评分系统（score\_system.py）进行每种算法的检测，此评分系统已经根据算法时间、算法占用空间实现了加分、罚分机制。检测的数据集主要有以下特征：io数量的范围是[10,10000]、分布在磁带的前100%、随机分布。根据检测的寻道时延与得分情况绘制可视化图表便于得出实验结论，对于不同区间的io\_count采用不同的算法。

3.3 结果分析

io\_count范围在[10,970]，测试结果如下：



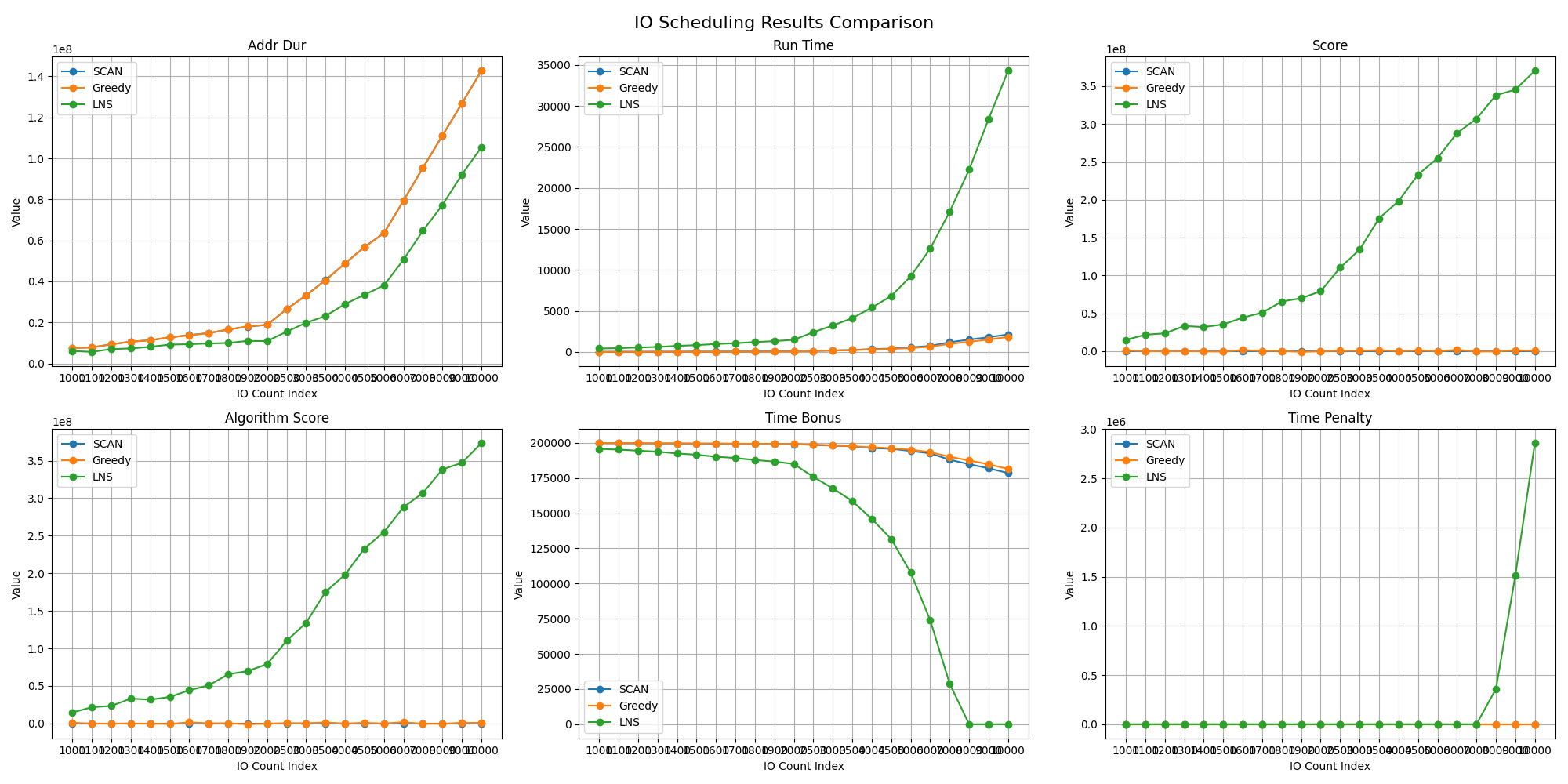
可见，LNS算法在[0,900]范围的数据集优化能力较差，寻道时延较长，比不上基线算法SCAN，呈负优化趋势。总体比不上LKH和贪心算法，LKH算法在此范围的得分占优，所以本题在[0,900]范围采用LKH算法。[900,970]采用LNS算法。

io\_count范围在[1000,2000]，测试结果如下



可见此范围内LNS算法寻道时延明显比其他算法更低，相应的得分也比其他算法更高。故此范围内使用LNS算法最佳。

io\_count范围在[1000,10000]，测试结果如下：



此范围内基线算法SCAN的曲线已与贪心算法趋于重合，而LNS算法寻道时延和得分都全面领先。

综上所述，对于磁电盘调度问题，io\_count范围在[10,900]采用LKH算法，[900,10000]采用LNS算法。