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：把心放在肚子里

电子邮箱：2389287408@qq.com

提交日期：2025/1/1

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。

前言

（只针对决赛的评分规则来做描述）

背景

赛题磁电盘 IO 最优调度要求通过采用 IO 调度技术，对一批 IO 数据访问序列进行最优化排序，目的是降低寻道时延迟与访问磨损。从这可以看出该问题是一个典型的 OTSP 调度问题，与 TSP 调度不同的点在于不要求起点与终点相连接，但是 TSP 调度的解法仍然适用与 OTSP 问题。该问题是一个 NP 难问题，是可以求出一个解析解。常用的做法有数学规划中的分支定界法，运筹学中的指派问题结合子回路消除以及动态规划中的最短哈密尔顿路径问题。但是随着数据规模的增大，上述解法的时间复杂度无法用多项式来衡量。所以结合本题的数据范围我就摒弃了上述的解法，采用另一种常用的近似求解算法。常用的贪婪构造型算法有最近邻点策略，最近邻插入策略，最短链接策略，便宜算法等，启发式构造算法有 MST 启发式，K-opt，Christofides 启发式等。除以上类型的构造算法外，还有专门正对磁带调度的 Scan 算法。并且针对 TSP 问题这类组合优化问题，在构造解成功后，会利用元启发搜索算法来对调度结果进行调整直到无法提高或者达到题目的时间限制。

第一章 方案设计

思路概括

我在实际解题过程中，选择一些构造型算法并且结合搜索算法来得到一个较为优质的解。从最终的评分函数来看由 5 部分组成，分别是调度算法加分，调度用时加分，调度超时罚分，空间超限罚分，调度错误罚分。我进一步对得分函数进行简化得到如图 1 所示。

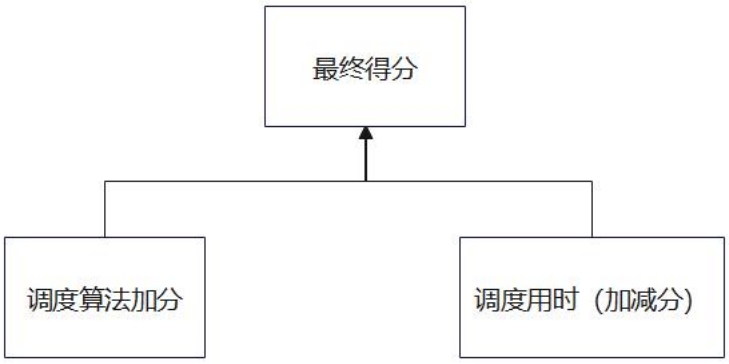


图 1

明确我的得分函数后，我的大致做法是通过多组构造解的组合，选择最好的一种作为启发搜索的初始解，根据数据集的规模以及单位时间内结果的提升速度来选择是否结束启发，从而得到最后的较为优质的可行解。

总体架构

通过以上的思路概括可以明确我是将构造解和启发算法进行结合来得到最终结果，所以该部分我会分别阐述我如何设计构造解以及如何设计启发式算法。具体的我所设计的构造型解法如表 1 所示。

表 1

调度算法名称	简单解释	文档中的表示符号
BruteForce	枚举所有可能的方案	BF
正序一遍调度算法	基线算法	BS
逆序一遍调度算法	从右往左调度的基线算法	RBS
ScanLR	从指针当前位置的从左到右的循环扫描	SLR
ScanRL	从指针当前位置的从右到左的循环扫描	SRL
NearestNeighbor	最近邻算法，每次选择里当前最近的 IO	NN
NearestInsertion	最近邻插入算法，根据某些策略选择 IO 插入到已构造序列中	IN
ShortestLink	最短链接法，借助 AGNES 从底向上的聚类思想	SL
混合构造算法（我通过多次试验得到的创新组合算法）		
NN&IN	组合最近邻和最近邻插入	NN&IN
IN&SL	组合最近邻插入和最短链接	IN&SL

函数声明展示：

```
// 初始解函数
int32_t IOScheduleAlgorithm(const InputParam *input, OutputParam *output);
uint32_t IOScheduleAlgorithmScan(const InputParam *input, OutputParam *output);
uint32_t IOScheduleAlgorithmScanRightToLeft(const InputParam *input, OutputParam *output);
uint32_t IOScheduleAlgorithmNearest(const InputParam *input, OutputParam *output);
uint32_t IOScheduleAlgorithmBaseline(const InputParam *input, OutputParam *output, bool flag);
uint32_t IOScheduleAlgorithShortestLink(const InputParam *input, OutputParam *output);
uint32_t IOScheduleAlgorithmNearestInsert(const InputParam *input, OutputParam *output);
uint32_t IOScheduleAlgorithmNIMixSL(const InputParam *input, OutputParam *output);
int32_t AlgorithmRun(const InputParam *input, OutputParam *output);
/*
* 随机种子
```

具体的我所设计的启发算法的算子如表 2 所示。

表 2

领域算子名称	解释
NeighborsSwap	随机交换两个 IO
Neighbors2Opt	颠倒一段调度序列
NeighborsInsert	随机选择一个 IO 插入到序列中的某个位置
NeighborsInsertPart	按照某种策略选择一段序列插入到随机的位置中
neighborsDisturbance1	扰动算子 1，防止过早陷入局部解
neighborsDisturbance2	扰动算子 2，防止过早陷入局部解

在实际的调用过程中，我对算子做了进一步改进，目的是为了加快迭代的次数以及减少无效的迭代。

函数声明展示：

```
/*
* 领域算子
*/
void neighborsSwap(uint16_t *oldSeries, uint16_t *newSeries, int length);
void neighbors2Opt(uint16_t *oldSeries, uint16_t *newSeries, int length);
void neighborsInsert(uint16_t *oldSeries, uint16_t *newSeries, int length);
void neighborsInsertPart(uint16_t *oldSeries, uint16_t *newSeries, int length);
void neighborsDisturbance1(uint16_t *oldSeries, uint16_t *newSeries, int length);
void neighborsDisturbance2(uint16_t *oldSeries, uint16_t *newSeries, int length);
```

具体的我所设计的启发算法如表 3 所示。

表 3

启发算法	解释
SA	模拟退火
HillClimb	爬山算法

GVNS	爬山算法改进-变领域搜索

函数声明展示：

```
/*
* 模拟退火以及变种
*/
void SA1(const InputParam *input, OutputParam *output);
void SA2(const InputParam *input, OutputParam *output);
void SA3(const InputParam *input, OutputParam *output);

/*
* 爬山算法以及变种(在数据量大的时候使用)
*/
void GVNS(const InputParam *input, OutputParam *output, int length);
void hillClimbSwap(const InputParam *input, OutputParam *output);
void hillClimbInsert(const InputParam *input, OutputParam *output);
/*
* 禁忌搜索
*/
void TS(const InputParam *input, OutputParam *output, int length, int op);
bool inTabuList(uint16_t* series, int tabuLen);
void pushTabuList(uint16_t* arr, int tabuLen);
/*
* 领域算子
```

完整的程序运行流程如图 2 所示。

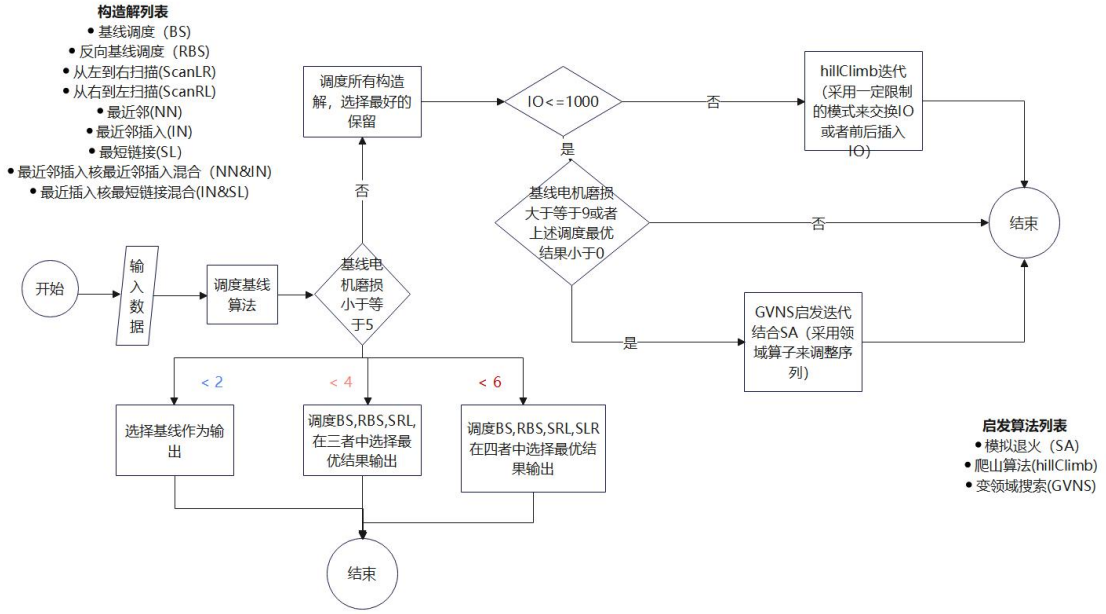


图 2

上述的流程图描述了程序完整的运行过程，在每个模块中的关键细节设计并没有体现，在后续的详细设计过程中我会具体阐述。

得分函数设计

考虑到决赛评分涉及到寻道时间，电机磨损，磁带磨损三个因素的影响，并且 IO 的存在备份归档（backup)和性能型场景(hdd)，在我多次试验过后，发现使用 hdd 的权重作为最后的得分函数效果最佳，因为性能型场景在一定程度上覆盖了备份归档，并且在寻道时间为较大权重时，会隐式的加大电机磨损的权重。综上因素，我确定 hdd 类型的权重来设计我最终的得分函数。公式如下所示。

$$scores = 0.5 * (b_t - m_t)/b_t + 0.3 * (bb_w - mb_w)/bb_w + 0.2 * (bm_w - mm_w)/bm_w$$

函数定义如下

```
double getHddScore(const InputParam *input, const OutputParam *output) {

    AccessTime accessTime = {0};
    TotalAccessTime(input, output, &accessTime);

      /* 带体磨损 */
    TapeBeltSegWearInfo segWearInfo = {0};
    double tapeBeltWear = TotalTapeBeltWearTimes(input, output, &segWearInfo);

    /* 电机磨损 */
    double tapeMotorWear = TotalMotorWearTimes(input, output);


    return hdd[0] * (baselineAD - 1.0 * accessTime.addressDuration - 1.0 * accessTime.readDuration) / baselineAD
    + hdd[1] * (baselineBelt - 1.0 * tapeBeltWear) / baselineBelt +
    hdd[2] * (baselineMotor - 1.0 * tapeMotorWear) / baselineMotor;
}
```

构造型解法的设计

在该部分我会具体描述我的所有构造解的详细工作流程。

BruteForce（暴力枚举）

枚举所有的序列排列，在初赛的时候我使用这种暴力+一些剪枝手段可以保证在 io 数量为 10 的时候求出最优解，并且程序运行的时间不超过 1 秒，但是在 io 数量大于 10 的时候，程序运行时间大大增加，在决赛的时候，我通过基线电机磨损的分析可以很好的在 O(m*n*n)的复杂度下求出最优解，其中 m 表示我的初始解个数，n 表示 io 数量。所以在决赛过程中我摒弃了该算法。

正序一遍调度算法

基线调度算法，将所有从左到右的 io 和从右到左的 io 分开来统计，并且按照起点的先后顺序排序（若是从左到右，则按照起点从小到大排序，若是从右到左，则按照起点从大到小排序），这部分算法的实现是后续启发搜索的基础，因为我需要得到基线的寻道时间，磁带磨损，电机磨损来填充得分函数的空缺部分，当基线调度算法不准确的时候会很大影响结果。

逆序一遍调度算法

与基线调度算法的区别在与是先调用从右到左的 io,再调用从左到右的 io。这部分设计是为了弥补暴力的耗时多，在基线电机磨损小于等于 5 的时候尽可能覆盖最优解，在通过数次实验过后发现，确实会存在最优解落在该调度算法上的可能。

ScanLR(从左到右循环扫描）

根据磁头的初始位置，依次从左到右调度能够顺延调度的 io，该部分的设计也是为了弥补暴力的耗时多，保证获得优质解的同时获取最大的算法调度用时加分。

ScanRL(从右到左循环扫描）

根据磁头的初始位置，依次从右到左调度能够顺延调度的 io，该部分的设计也是为了弥补暴力的耗时多，保证获得优质解的同时获取最大的算法调度用时加分。

上述的调度算法是为了保证在很短的时间内可以得到一个至少不会比基线差的结果，从而保证了整个程序运行的下界。

NearestNeighbor(最近邻）

每次的 io 选择都是离当前磁头最近的，通过多次实验发现用这种单独做法来调度所有的 io，效果在 io 分布较为分散的时候效果较好，但是当数据量大于等于 100 以后这种做法做法很差，所以我在实际的使用过程中是将该算法混合其他构造算法来使用。

NearestInsertion(最近邻插入）

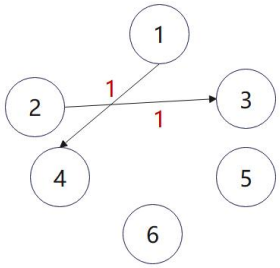
随机选择三个相互较近的 io 作为初始序列，每次在未调度的集合中按照某种策略选择一个 io 找到使得序列长度增加最小的位置插入，直到所有 io 都被调度则结束构

造，上述的某种策略可以分为最远插入，最近插入，最节省插入，随机插入。我在调用的时候等概率在这四种选择中做出选择。但是这种做法的时间复杂度较高，在实际过程中也是结合其他构造算法来使用。

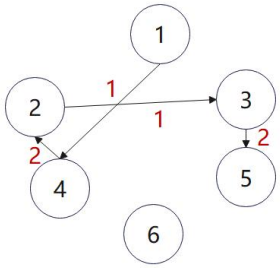
图 2 借助 AGENS 层次聚类来描述最短链接 AGENS 是一种自底向上聚合策略的层次聚类算法，先奖数据集中的每一个样本看作一个初始聚类，然后在算法运行的每一步找出距离最近的两个聚类簇进行合并，该过程不断重复，直到达到预设的聚类簇的个数，我在实际使用过程中将最终簇的个数设置为 1 即可。假设现在有些独立为调度的 io，一次最短链接的做法是，寻找离每个 io 最近的 io,按照得到的距离进行排序，若当前块在当次链接过程中未被拼接并且能够拼接的块在当次也未被其他块拼接，则进行拼接形成新的块。直到剩下一个块为止结束构造。在实际的过程中我发现在 io 数量大于等于 2000 的时候得到的效果几乎超过了我以上的所有构造解，所以考虑到算法调度用时加分的影响，我在 io 数量大于等于 2000 的时候使用的都是基于最短链接混合其他构造解这样的模式。以下我用图 3 近一步介绍该算法运行过程。

	1	2	3	4	5	6
1	∞	3	3	1	4	5
2	3	∞	1	2	5	2
3	4	7	∞	1	2	6
4	2	2	1	∞	4	2
5	3	3	2	1	∞	8
6	9	3	1	2	4	∞

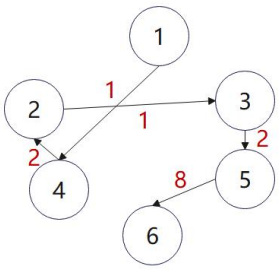
距离矩阵



(1)



(2)



(3)

最短链接得到结果:
1->4->2->3->5>6
tsp路径长度为14

图 3

IN&SL(最近邻插入和最短链接混合）

经过多次实验的测试结果发现，最短链接法在 io 合并的后期会出发极其不合理的情况出现，导致合并的效果急剧变差，事实上理论分析后也能发现前期合并的块几乎是一个 top5 甚至到 top10 之内的情况，但是在合并后期会远离这个范围，所以我尝试

用最近邻插入进行改善，发现有显著效果。同样基于图 3 的例子尝试用最近邻插入来改进后的结果如图 4 所示。

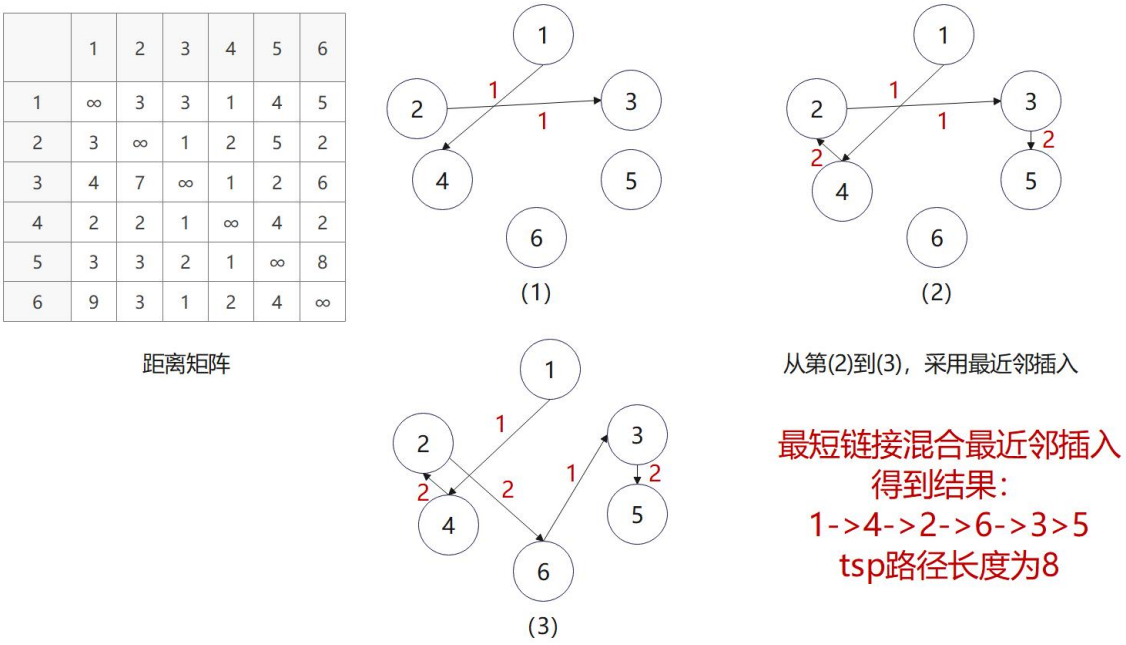


图 4

启发式搜索的设计

在该部分我会具体描述我的所有构造解的详细工作流程。

领域算子的构造

在具体的实现过程我实现了很多领域算子，但是我发现大部分算子在启发过程中得到的效果并不是很好，只有成块插入的领域算子有较好的效果，所以在实际的运行过程中我以该算子作为启发的主要算子，其余算子的用途包括不限于用来防止陷入局部最优解作扰动作用。

NeighborsInsertPart 的实现

起初我实现的插入算子，在进行一次调整后会计算对整个序列的寻道时间，电机磨损以及磁带磨损会进行重新计算，这样会浪费大量的时间开销，所以在后续我调整了插入算子，并且在一次调整后只对局部变化的部分进行重新计算，这样使得迭代的次数大大增加，并且启发的效果也提高了不少，以下是我实现插入算子的代码。

```
inline void neighborsInsertPartChange(int *insertPos, int *insertL, int *insertR, int
length) {
    // 选择需要移动的部分
    int l = nextN(length);
    int r = nextN(length);
    while (l == r || abs(r - l + 1) >= length || abs(l - r + 1) >= length) {
        r = nextN(length);
    }
    if (l > r) {
        int temp = l;
        l = r;
        r = temp;
    }
    // 选择插入的位置随机选择
    int flag = 0, _insertPos = -1;
    while (_insertPos == -1) {
        int randomPos = -1;
        if (flag == 0 && l >= 0) {
            if (l > 0) {
                randomPos = nextNN(0, l);
            }
            flag = 1;
        } else if (flag == 1 && r < length){
            if (r < length - 1) {
                randomPos = nextNN(r + 1, length);
            }
            flag = 0;
        }

        if (randomPos != -1) {
            _insertPos = randomPos;
        }
        // printf("%d% d %d\n", insertPos, l, r);
    }
    // 得到插入的位置以及待插入的序列
    (*insertPos) = _insertPos, (*insertL) = l, (*insertR) = r;
}
```

上述代码的目的是得到插入的位置以及待插入的序列的开始位置以及结束位置。

```
inline void updateInsertPartChange(uint16_t *oldSeries, uint16_t *newSeries, int
insertPos, int insertL, int insertR, int length) {
    int l = insertL, r = insertR;
    // l - r 插入到 pos 位置
    if (insertPos < l) {
        // 向前插入
        for (int i = 0; i < insertPos; ++ i) newSeries[i] = oldSeries[i];
        for (int i = insertPos, len = 0; len < r - l + 1; ++ len, ++ i) {
            newSeries[i] = oldSeries[l + len];
        }
        // insertPos ---- l - 1
        for (int i = insertPos + r - l + 1, len = 0; len < l - 1 - insertPos + 1; ++
len, ++ i) {
            newSeries[i] = oldSeries[insertPos + len];
        }
        // r + 1 - length - 1
        for (int i = r + 1; i < length; ++ i) {
            newSeries[i] = oldSeries[i];
        }
    } else {
        // for (int i = 0; i < length; ++ i) {
        //     newSeries[i] = oldSeries[i];
        // }

        for (int i = 0; i <= l - 1; ++ i) newSeries[i] = oldSeries[i];
        for (int i = l, len = 0; len < insertPos - (r + 1) + 1; ++ len, ++ i) {
            newSeries[i] = oldSeries[r + 1 + len];
        }
        for (int i = l + insertPos - (r + 1) + 1, len = 0; len < r - l + 1; ++ len,
++ i) {
            newSeries[i] = oldSeries[l + len];
        }
        for (int i = insertPos + 1; i < length; ++ i) {
            newSeries[i] = oldSeries[i];
        }
    }
}
```

上述代码用来实现执行真实的插入操作。

爬山算法的实现

考虑到 io 的数据范围在 10 到 10000，所以当 io 数量大于等于 1000 的时候，只需要使用爬山的策略就可以在规定的时间范围内得到比较好的结果，爬山的过程展示如图 5 所示。

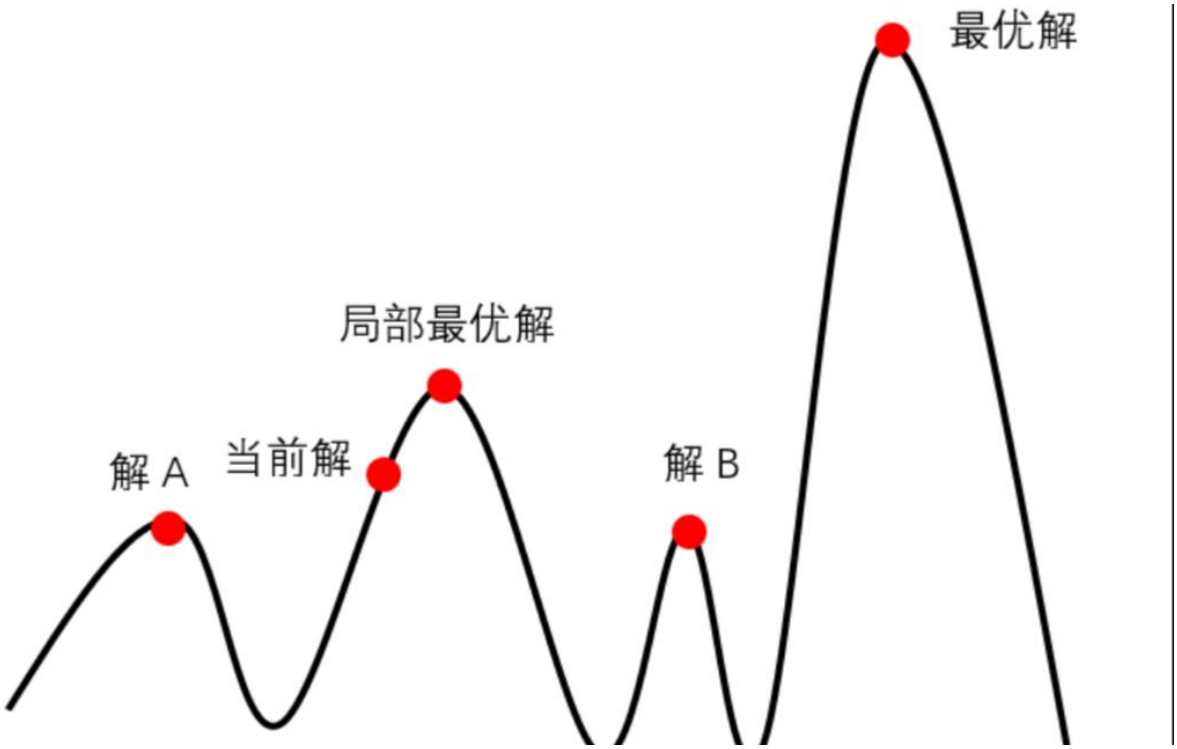


图 5

为什么我在 io 数量大于等于 1000 的时候选择使用爬山而并不是使用模拟退火的原因在于当初始解落在山中的某个点的时候，往能够看到的山顶进行爬坡在规定的时间内也不一定能够爬到，并且经过多次实验过后，发现当 io 数量较大的时候确实采用爬山算法有较好的效果。在具体的实现过程中我以及时间作为是否停止迭代的标准，也考虑到调度用时加分的影响后加入了逻辑或的条件，大致描述为当在某段时间跨度内没有得到有效的提高（我在实现的时候是用变化率与某个阈值来进行比较，当小于阈值后则结束爬山）。这里用到的算子就是上述的插入算子。具体的条件判断如图 6 所示。

```
gettimeofday(&end, NULL);  // 记录结束时间
long tseconds, ttuseconds;
long seconds, useconds;    // 秒数和微秒数

tseconds = end.tv_sec - e1.tv_sec;
ttuseconds = end.tv_usec - e1.tv_usec;

long mseconds = (tseconds)*1000 + ttuseconds / 1000;

seconds = end.tv_sec - start.tv_sec;
useconds = end.tv_usec - start.tv_usec;
uint32_t costTime = ((seconds)*1000000 + useconds) / 1000000.0;
if (costTime > 18 || mseconds >= stuck) break;
```

图 6

变领域搜索算法的实现

当 io 数量小于 1000 的时候我发现单纯的用爬山算法可能会过早陷入局部的最优解，所以我尝试用多种算子来作扰动算子，并且采取模拟退火的策略允许扰动结果比当前最好的结果差，并且以一定概率接受。具体的扰动算子的选择以及下降算子的选择如图 7 所示。

```
// 扰动
switch(k) {
    case 0: neighborsSwap(oldSeries, shakeSeries, length);break;
    case 1: neighborsDisturbance1(oldSeries, shakeSeries, length);break;
    case 2: neighborsDisturbance2(oldSeries, shakeSeries, length);break;
}

while (l < lmax) {
    double curResult = -1;
    for (int i = 0; i < neigNumber; i ++ ) {
        switch(l) {
            case 0: neighbors2Opt(shakeSeries, newSeries, length);break;
            case 1:  neighborsInsert(shakeSeries, newSeries, length);break;
            case 2: neighborsInsertPart(shakeSeries, newSeries, length);break;
        }

        // 改变output
```

图 7

启发提分的关键设计

启发的过程是围绕得分函数进行迭代，所以第一个关键设计是要保证得分函数的准确表达，而且在得分函数的表达式中需要实现基线算法，在第一次追测后几乎可以确定我自己实现的基线算法准确，至此解决了得分函数表达式中所有需要的变量。第二个关键设计就是在有限的时间内尽可能多的迭代并且保证一定的迭代高效性。经过我的多次实验后发现调用官方给的计算磁带磨损的接口来计算的话会有很大的时间开销，导致在有限的时间内只能迭代到10^4的数量级，所以我考虑等效替换掉磁带磨损的计算，也是在多次实验下来找到替换的办法，并且也经过测试得到的最终分数与实际通过官方接口计算的分数的差值在十分位甚至更小。在替换之后我的迭代可以到10^7的数量级，使得整体的分数有了很大提高。具体的设计如下所示。

```
inline uint32_t getBeltWear(HeadInfo *info1, HeadInfo *info2) {
    int direct1 = (*info1).wrap % 2 == 0 ? 1 : -1;
    int direct2 = (*info2).wrap % 2 == 0 ? 1 : -1;


    int extra = ((*info1).status == 1 ? addRoute : SE0);
    if ((*info1).status == 1) {
        // 1 : 正向; 2 : 反向
        int lpos1 = (*info1).lpos, lpos2 = (*info2).lpos;
        if (direct1 == direct2) {
            if (direct1 == 1) {
                if (lpos2 >= lpos1) {
                    return abs(lpos1 - lpos2) + ((*info1).wrap != (*info2).wrap ? 1 : 0);
                } else {
                    if (lpos1 <= PdMaxPos && lpos2 >= SameDircetTerval) {
                        return abs(lpos1 - lpos2) + addRoute;
                    }
                    else if (lpos2 >= SameDircetTerval) {
                        return abs(lpos1 - lpos2) + addRoute - (lpos1 - PdMaxPos) * 2;
                    } else if (lpos1 <= PdMaxPos){
                        return abs(lpos1 - lpos2) + addRoute - (SameDircetTerval - lpos2) *
2;
                    }else {
                        return abs(lpos1 - lpos2) + (MAX_LPOS - lpos1) * 2 + 2 * (lpos2);
                    }
                }
            } else {
                if (lpos2 <= lpos1) {
                    return abs(lpos1 - lpos2) + ((*info1).wrap != (*info2).wrap ? 1 : 0);
                } else {
```

```
                    if (lpos1 >= rdMinPos && lpos2 <= MAX_LPOS -  SameDircetTerval) {
                        return abs(lpos1 - lpos2) + addRoute;
                    } else if (lpos2 <= MAX_LPOS -  SameDircetTerval) {
                        return abs(lpos1 - lpos2) + addRoute - (rdMinPos - lpos1) * 2;
                    } else if (lpos1 >= rdMinPos) {
                        return abs(lpos1 - lpos2) + addRoute - (lpos2 - (MAX_LPOS -
SameDircetTerval)) * 2;
                    }else {
                        return abs(lpos1 - lpos2) + (lpos1) * 2 + 2 * (MAX_LPOS - lpos2);
                    }
                }
            }
        } else {
            if (direct1 == 1) {
                if (lpos1 >= lpos2) {
                    int sub = lpos1 - lpos2;
                    if (lpos1 <= PdMaxPos) {
                        if (sub > turnInterval) {
                            return sub + turnRoute;
                        } else {
                            return sub + turnRoute + (turnInterval - sub) * 2;
                        }
                    } else {
                        if (sub > turnInterval) {
                            return sub + turnRoute - (lpos1 - PdMaxPos) * 2;
                        } else {
                            return sub + turnRoute - (lpos1 - PdMaxPos) * 2 + (turnInterval
- sub) * 2;
                        }
                    }
                } else {
                    int sub = lpos2 - lpos1;
                    if (lpos1 < PdMaxPos) {
                        if (lpos2 >= MAX_LPOS - rturnInterval) {
                            return sub + rturnRoute - (lpos2 + rturnInterval - MAX_LPOS) * 2;
                        } else {
                            return sub + rturnRoute;
                        }
                    }
                    if (lpos1 <= MAX_LPOS - rturnInterval && lpos2 <= MAX_LPOS - rturnInterval)
{
                        return abs(lpos1 - lpos2) + rturnRoute;
                    }
                }
```

```
            } else {
                if (lpos1 <= lpos2) {
                    int sub = lpos2 - lpos1;
                    if (lpos1 >= rdMinPos) {
                        if (sub > turnInterval) {
                            return sub + turnRoute;
                        } else {
                            return sub + turnRoute + (turnInterval - sub) * 2;
                        }
                    } else {
                        if (sub > turnInterval) {
                            return sub + turnRoute - (rdMinPos - lpos1) * 2;
                        } else {
                            return sub + turnRoute - (rdMinPos - lpos1) * 2 + (turnInterval
- sub) * 2;
                        }
                    }
                }else {
                    int sub = lpos1 - lpos2;
                    if (lpos1 > rdMinPos) {
                        if (lpos2 < rturnInterval) {
                            return sub + rturnRoute - (rturnInterval - lpos2) * 2;
                        } else {
                             return sub + rturnRoute;
                        }
                    }
                    if (lpos1 >= rturnInterval && lpos2 >= rturnInterval) {
                        return abs(lpos1 - lpos2) + rturnRoute;
                    }
                }
            }
        }
    } else {
        // 1 : 正向; 2 : 反向
        int lpos1 = (*info1).lpos, lpos2 = (*info2).lpos;
        if (direct1 == direct2) {
            if (direct1 == 1) {
                if (lpos2 >= lpos1) {
                    if ((lpos2 - lpos1) >= addRoute) {
                        return abs(lpos1 - lpos2) + ((*info1).wrap != (*info2).wrap ? 1 : 0);
                    }
                } else {
                    if (lpos1 <= PdMaxPos) {
                        return abs(lpos1 - lpos2) + extra;
```

```
                    }
                }
            } else {
                if (lpos2 <= lpos1) {
                    if ( (lpos1 - lpos2) >= addRoute) {
                        return abs(lpos1 - lpos2) + ((*info1).wrap != (*info2).wrap ? 1 : 0);
                    }
                } else {
                    if (lpos1 >= rdMinPos) {
                        return abs(lpos1 - lpos2) + extra;
                    }
                }
            }
        }
    }
    return BeltWearTimes(info1, info2, &segWearInfo);
}
```

这一部分实现的功能就是来替换原有的计算磁带磨损的接口，我通过不断实验尝试，得出了以下存在的 io 调度的模式，如图 8 所示。

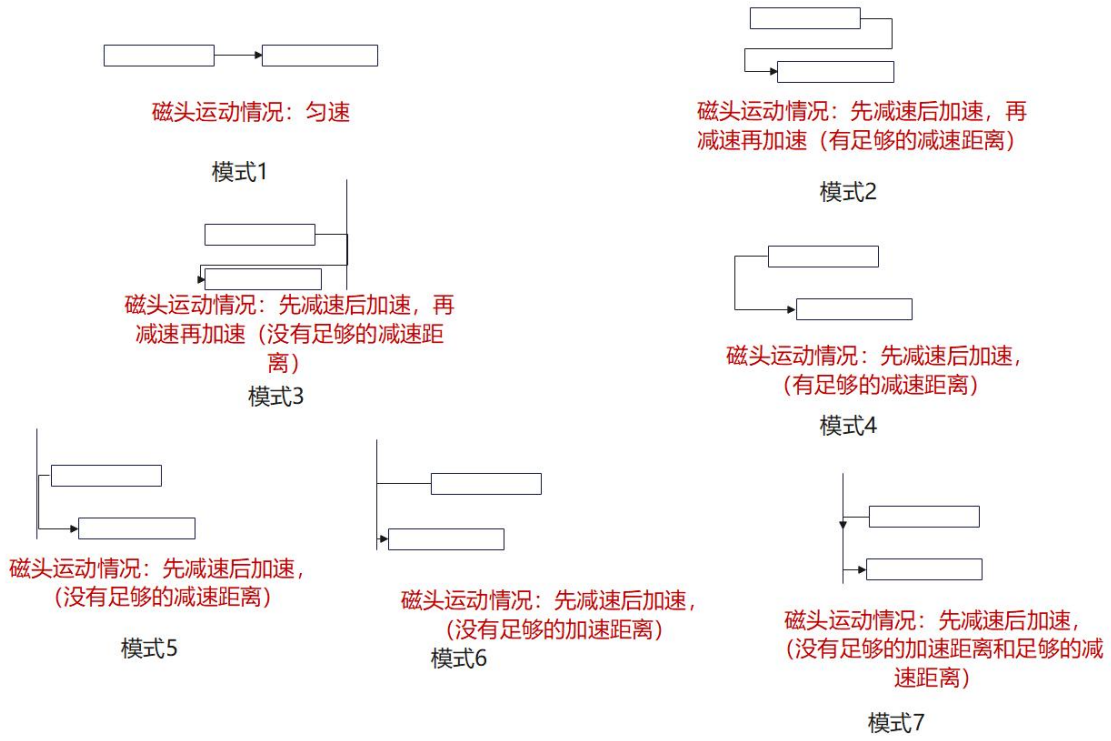


图 8

大数据集下的领域交换策略

在数据量超过 1000 的时候使用插入算子的方式提分几乎与算法调度加分持平，所以为了使得迭代更加有效果，设计出如下模式，让同方向的 io 并且起点相差距离在一定范围内的 io 对加入待交换集合中，让同方向并且终点与起点相差距离在一定范围内的 io 对加入到待插入集合中。然后遍历交换集合，尝试进行交换，并且遍历插入集合，尝试进行插入，若有得分有提高则接受。

第二章 关键代码说明

在第一章我以方案+代码的形式进行展示，所以在这一章我只对我的主函数进行说明。

```
inline int32_t AlgorithmRun(const InputParam *input, OutputParam *output)
{
    gettimeofday(&start, NULL);
    // 运行基线算法
    saveBaseline(input, output);
    double fBaseline = 0.0;
    double bestScore = fBaseline;

    int *answer = (int *)malloc((output->len) * sizeof(int));
    // 保存结果
    for (int i = 0; i < output->len; ++ i) answer[i] = output->sequence[i];
    ioCount = input->ioVec.len;
    int32_t ret;
    if (ioCount <= 10) {
        // Bruteforce(input, output);
        IOScheduleAlgorithmBaseline(input, output, false);
        double rBaseline = getBackupScore(input, output);
        if (rBaseline > bestScore) {
            bestScore = rBaseline;
            for (int i = 0; i < output->len; ++ i) answer[i] = output->sequence[i];
        }
        IOScheduleAlgorithmScan(input, output);
        double ScanLR = getBackupScore(input, output);
        if (ScanLR > bestScore) {
            bestScore = ScanLR;
            for (int i = 0; i < output->len; ++ i) answer[i] = output->sequence[i];
        }
        IOScheduleAlgorithmScanRightToLeft(input, output);
        double ScanRL = getBackupScore(input, output);
        if (ScanRL > bestScore) {
```

```
            bestScore = ScanRL;
            for (int i = 0; i < output->len; ++ i) answer[i] = output->sequence[i];
        }
        #ifdef DEBUG
            printf("正向基线:%lf\n", fBaseline);
            printf("反向基线: %lf\n", rBaseline);
            printf("从左到右扫描: %lf\n", ScanLR);
            printf("从右到左扫描: %lf\n", ScanRL);
        #endif

    } else if (ioCount <= 2000){
        IOScheduleAlgorithmBaseline(input, output, false);
        double rBaselineScore = getBackupScore(input, output);
        double rBaselineScorex = getHddScore(input, output);
        if (rBaselineScore > bestScore) {
            bestScore = rBaselineScore;
            for (int i = 0; i < output->len; ++ i) answer[i] = output->sequence[i];
        }
        // printf("sssssssssss\n");
        IOScheduleAlgorithmScan(input, output);
        // printf("sssssssssss\n");
        double scanLRScore = getBackupScore(input, output);
        double scanLRScorex = getHddScore(input, output);
        if (scanLRScore > bestScore) {
            bestScore = scanLRScore;
            for (int i = 0; i < output->len; ++ i) answer[i] = output->sequence[i];
        }
        IOScheduleAlgorithmScanRightToLeft(input, output);
        double scanRLScore = getBackupScore(input, output);
        double scanRLScorex = getHddScore(input, output);
        if (scanRLScore > bestScore) {
            bestScore = scanRLScore;
            for (int i = 0; i < output->len; ++ i) answer[i] = output->sequence[i];
        }
        tanxindiaodu(input, output);
        double nearestScore = getBackupScore(input, output);
        double nearestScorex = getHddScore(input, output);
```

```
        if (nearestScore > bestScore) {
            bestScore = nearestScore;
            for (int i = 0; i < output->len; ++ i) answer[i] = output->sequence[i];
        }
        #ifdef DEBUG
            printf("********backup**********\n");
```

```
            printf("rbaselineScore: %lf\n", rBaselineScore);
            printf("scanLRScore: %lf\n", scanLRScore);
            printf("scanRLScore: %lf\n", scanRLScore);
            printf("nearestScore: %lf\n", nearestScore);
            printf("********hdd**********\n");
            printf("rbaselineScore: %lf\n", rBaselineScorex);
            printf("scanLRScore: %lf\n", scanLRScorex);
            printf("scanRLScore: %lf\n", scanRLScorex);
            printf("nearestScore: %lf\n", nearestScorex);
        #endif
    } else {
        tanxindiaodu(input, output);
        double nearestScore = getBackupScore(input, output);
        double nearestScorex = getHddScore(input, output);
        if (nearestScore > bestScore) {
            bestScore = nearestScore;
            for (int i = 0; i < output->len; ++ i) answer[i] = output->sequence[i];
        }
    }
    for (int i = 0; i < output->len; ++ i) output->sequence[i] = answer[i];
    /**************************************************/
    if (ioCount > 10 && ioCount <= 1000 && (baselineMotor >= 9 || bestScore == 0)) {
        // 由于系统提供的磁带磨损计算开销太大，经过我的实践发现，在符合一些情况下，加数的时间是固定
的
        info1.status = 1, info1.lpos = 30000, info1.wrap = 0;
        info2.status = 1, info2.lpos = 20000, info2.wrap = 0;
        // 二分正常加速需要的最小时间间隔
        addRoute = BeltWearTimes(&info1, &info2, &segWearInfo) - 10000;
        int l = MAX_LPOS - addRoute, r = MAX_LPOS;
        // 寻找第一个不满足的位置
        while (l < r) {
            int mid = (l + r) / 2;
            if (check1(mid)) {
                l = mid + 1;
            } else {
                r = mid;
            }
        }
        PdMaxPos = l - 1;
        rdMinPos = 0 + (MAX_LPOS - PdMaxPos);
        // printf("XXXXXXXXXXXXXX%d\n", PdMaxPos);
        /************************************************************/
        info1.status = 1, info1.lpos = 20000, info1.wrap = 0;
        info2.status = 1, info2.lpos = 10000, info2.wrap = 0;
```

```
        // 二分正常加速需要的最小时间间隔()
        l = 0, r = l + addRoute;
        // 寻找第一个不满足的位置
        while (l < r) {
            int mid = (l + r) / 2;
            if (check1_1(mid)) {
                r = mid;
            } else {
                l = mid + 1;
            }
        }
        SameDircetTerval = l + 100;
        /************************掉头的最小时间间隔***********************************/
        info1.status = 1, info1.lpos = 30000, info1.wrap = 0;
        info2.status = 1, info2.lpos = 20000, info2.wrap = 1;
        turnRoute = BeltWearTimes(&info1, &info2, &segWearInfo) - 10000;
        l = 20000, r = 30000;
        while (l < r) {
            int mid = (l + r) / 2;
            if (check2(mid)) {
                l = mid + 1;
            } else {
                r = mid;
            }
        }
        turnInterval = (30000 - l) + 100;
        // printf("sssssssssssssssssssss%d\n", turnInterval);
        /*****************************************************************************/
        info1.status = 1, info1.lpos = 20000, info1.wrap = 0;
        info2.status = 1, info2.lpos = 21000, info2.wrap = 1;
        rturnRoute = BeltWearTimes(&info1, &info2, &segWearInfo) - 1000;
        l = MAX_LPOS - 10000, r = MAX_LPOS;
        while (l < r) {
            int mid = (l + r) / 2;
            if (check3(mid)) {
                l = mid + 1;
            } else {
                r = mid;
            }
        }
        rturnInterval = (MAX_LPOS - l) + 10;
        // 计算 SE 和 SE0
        info1.status = 1, info1.lpos = 20000, info1.wrap = 0;
        info2.status = 1, info2.lpos = 10000, info2.wrap = 2;
```

```
        SE = abs(10000 - BeltWearTimes(&info1, &info2, &segWearInfo));
        // printf("SE : %d\n", SE);
        info1.status = 0, info1.lpos = 20000, info1.wrap = 0;
        SE0 = abs(10000 - BeltWearTimes(&info1, &info2, &segWearInfo));
        // printf("SE0 : %d\n", SE0);
        struct timeval1 e1;
        gettimeofday(&e1, NULL);  // 记录结束时间
        // 统计两个 io 之间(寻道时间, 磁带磨损, 电机磨损)
        int canUseHillClimb = 1;
        for (int i = 0; i < output->len; ++ i) {
            for (int j = 0; j < output->len; ++ j) {
                if (i == j) continue;
                info1.status = 1, info1.lpos = input->ioVec.ioArray[i].endLpos, info1.wrap
= input->ioVec.ioArray[i].wrap;
                info2.status = 1, info2.lpos = input->ioVec.ioArray[j].startLpos, info2.wrap
= input->ioVec.ioArray[j].wrap;
                // 寻道时间
                threeCost[i + 1][j + 1][0] = SeekTimeCalculate(&info1, &info2);
                // 带体磨损
                // // TapeBeltSegWearInfo segWearInfo = {0};
                threeCost[i + 1][j + 1][1] = getBeltWear(&info1, &info2);

                threeCost[i + 1][j + 1][2]=  MotorWearTimes(&info1, &info2);
            }
            gettimeofday(&end, NULL);  // 记录结束时间
            long seconds, useconds;    // 秒数和微秒数
            seconds = end.tv_sec - e1.tv_sec;
            useconds = end.tv_usec - e1.tv_usec;
            uint32_t costTime = ((seconds)*1000000 + useconds) / 1000000.0;
            if (costTime > 15) {
                canUseHillClimb = 0;
                break;
            }
        }
        // 计算磁头初始位置到每个 io 的开销
        for (int i = 0; i < output->len; ++ i) {
                info1.status = 0, info1.lpos = input->headInfo.lpos, info1.wrap =
input->headInfo.wrap;
                info2.status = 1, info2.lpos = input->ioVec.ioArray[i].startLpos, info2.wrap
= input->ioVec.ioArray[i].wrap;
                headToFirst[i + 1][0] = SeekTimeCalculate(&info1, &info2);
                headToFirst[i + 1][1] = getBeltWear(&info1, &info2);
                headToFirst[i + 1][2] = MotorWearTimes(&info1, &info2);
        }
```

```
        gettimeofday(&end, NULL);  // 记录结束时间
        long seconds, useconds;    // 秒数和微秒数
        seconds = end.tv_sec - e1.tv_sec;
        useconds = end.tv_usec - e1.tv_usec;
        uint32_t costTime = ((seconds)*1000000 + useconds) / 1000000.0;
        printf("预处理的时间开销:%d\n", costTime);
        if (canUseHillClimb == 1) {
            hillClimbInsert(input, output);
        }

    }
    #ifdef DEBUG1
        double backup = getBackupScore(input, output);
        double hdd = getHddScore(input, output);
        printf("bakcup: %lf\n", backup);
        printf("hdd: %lf\n", hdd);
    #endif

    free(answer);
    return RETURN_OK;
}
```

主函数的大致运行情况为先调度一遍基线算法，将基线的指标保存，再根据 io 的数量进行分类调度，当 io 数量小于等于 10 时，运行一些基础构造解（由电机磨损规律得到），当 io 数量小于等于 1000 时，由基线电机磨损的情况分类调度。当 io 数量大于 1000，调度所有构造解。

得到初始构建解后，根据 io 的数量选择不同的启发函数进行迭代，不满足迭代条件则结束。

第三章 测试及分析

我的环境配置：**测试环境:ubuntu18.04,2vcpu,2GB内存,x86_64架构**

（未使用x86架构的专门库，用了math.h在cmake文件中已经加入）

关键指标：

io数量为10

只需要在基础构造解中进行选择即可

```
正向基线:0.000000
反向基线: -0.154824
从左到右扫描: 0.050043
从右到左扫描: 0.122186
bakcup: 0.122186
hdd: 0.114091

Key Metrics:
        ioCount:                        10
        algorithmRunningDuration:       44.140 (ms)
        memoryUse:                      0 (KB)
        addressingDuration:             370657 (ms)
        readDuration:                   53622 (ms)
        tapeBeltWear:                   1116199
        tapeMotorWear:                  5
        errorIOCount:                   0

指标写入文件 ./metrics.txt
基线的情况
寻道时间 : 432622.000000, 带体磨损 : 1341441.000000, 电机磨损 : 5.000000
实际调度的情况
寻道时间 : 370657.000000, 带体磨损 : 1116199.000000, 电机磨损 : 5.000000
```

io数量为50

选择最好构造解+启发算法迭代

```
********backup**********
rbaselineScore: 0.117485
scanLRScore: 0.046629
scanRLScore: 0.078805
nearestScore: -0.662921
********hdd**********
rbaselineScore: 0.103597
scanLRScore: 0.041934
scanRLScore: 0.076940
nearestScore: -0.675272
预处理的时间开销:0
bestScore: 0.103625
bestScore: 0.116510
bestScore: 0.151501
bestScore: 0.169657
bestScore: 0.177201
迭代次数: 405482
bakcup: 0.198682
hdd: 0.177201

Key Metrics:
        ioCount:                        50
        algorithmRunningDuration:       3292.673 (ms)
        memoryUse:                      0 (KB)
        addressingDuration:             661227 (ms)
        readDuration:                   54700 (ms)
        tapeBeltWear:                   1359961
        tapeMotorWear:                  9
        errorIOCount:                   0

指标写入文件 ./metrics.txt
基线的情况
寻道时间 : 819686.000000, 带体磨损 : 1911752.000000, 电机磨损 : 9.000000
实际调度的情况
寻道时间 : 661227.000000, 带体磨损 : 1359961.000000, 电机磨损 : 9.000000
```

io数量为100

选择最好构造解+启发算法迭代，可以看到在构造解效果不好的时候也有很大提高

```
********backup**********
rbaselineScore: 0.039853
scanLRScore: -0.122271
scanRLScore: -0.179452
nearestScore: 0.065206
********hdd**********
rbaselineScore: 0.032007
scanLRScore: -0.017758
scanRLScore: -0.062596
nearestScore: 0.093648
预处理的时间开销:0
迭代次数: 985802
bakcup: 0.219975
hdd: 0.217972

Key Metrics:
        ioCount:                        100
        algorithmRunningDuration:       7704.440 (ms)
        memoryUse:                      0 (KB)
        addressingDuration:             1043802 (ms)
        readDuration:                   544671 (ms)
        tapeBeltWear:                   1971288
        tapeMotorWear:                  79
        errorIOCount:                   0

指标写入文件 ./metrics.txt
基线的情况
寻道时间 : 1326896.000000, 带体磨损 : 2350336.000000, 电机磨损 : 149.000000
实际调度的情况
寻道时间 : 1043802.000000, 带体磨损 : 1971288.000000, 电机磨损 : 79.000000
```

io数量为500

选择最好构造解+启发算法迭代，可以看到在构造解的基础也能提高10个点

```
********backup**********
rbaselineScore: -0.035065
scanLRScore: 0.038190
scanRLScore: 0.052311
nearestScore: 0.096134
********hdd**********
rbaselineScore: -0.025496
scanLRScore: 0.157147
scanRLScore: 0.167595
nearestScore: 0.185948
预处理的时间开销:0
迭代次数: 1568098
bakcup: 0.234226
hdd: 0.278018

Key Metrics:
        ioCount:                        500
        algorithmRunningDuration:       12873.051 (ms)
        memoryUse:                      0 (KB)
        addressingDuration:             2106332 (ms)
        readDuration:                   410336 (ms)
        tapeBeltWear:                   2729581
        tapeMotorWear:                  93
        errorIOCount:                   0

指标写入文件 ./metrics.txt
基线的情况
寻道时间 : 2866064.000000, 带体磨损 : 2765310.000000, 电机磨损 : 445.000000
实际调度的情况
寻道时间 : 2106332.000000, 带体磨损 : 2729581.000000, 电机磨损 : 93.000000
```

io数量为1000

因为构造解效果也很不错，所以启发的时间很快就结束了

```
********backup**********
rbaselineScore: -0.008979
scanLRScore: 0.451077
scanRLScore: 0.451700
nearestScore: 0.571357
********hdd**********
rbaselineScore: -0.006443
scanLRScore: 0.518305
scanRLScore: 0.518654
nearestScore: 0.601915
预处理的时间开销:0
迭代次数: 252006
bakcup: 0.571367
hdd: 0.601932

Key Metrics:
        ioCount:                        1000
        algorithmRunningDuration:       3249.628 (ms)
        memoryUse:                      0 (KB)
        addressingDuration:             4041958 (ms)
        readDuration:                   5459552 (ms)
        tapeBeltWear:                   9682826
        tapeMotorWear:                  247
        errorIOCount:                   0

指标写入文件 ./metrics.txt
基线的情况
寻道时间 : 16930454.000000, 带体磨损 : 16775807.000000, 电机磨损 : 3881.000000
实际调度的情况
寻道时间 : 4041958.000000, 带体磨损 : 9682826.000000, 电机磨损 : 247.000000
```

当io数量大于等于2000，几乎以构造解为我的最后结果。

io数量为2000

```
********backup**********
rbaselineScore: 0.017999
scanLRScore: 0.631254
scanRLScore: 0.631610
nearestScore: 0.696045
********hdd**********
rbaselineScore: 0.012598
scanLRScore: 0.689716
scanRLScore: 0.690029
nearestScore: 0.733229
bakcup: 0.696045
hdd: 0.733229

Key Metrics:
        ioCount:                        2000
        algorithmRunningDuration:       493.202 (ms)
        memoryUse:                      0 (KB)
        addressingDuration:             4406293 (ms)
        readDuration:                   2187074 (ms)
        tapeBeltWear:                   7199519
        tapeMotorWear:                  157
        errorIOCount:                   0

指标写入文件 ./metrics.txt
基线的情况
寻道时间 : 23448329.000000, 带体磨损 : 16247319.000000, 电机磨损 : 5997.000000
实际调度的情况
寻道时间 : 4406293.000000, 带体磨损 : 7199519.000000, 电机磨损 : 157.000000
```

io数量为5000

```
bakcup: 0.812796
hdd: 0.834927

Key Metrics:
        ioCount:                        5000
        algorithmRunningDuration:       490.896 (ms)
        memoryUse:                      0 (KB)
        addressingDuration:             6634620 (ms)
        readDuration:                   4118901 (ms)
        tapeBeltWear:                   10920961
        tapeMotorWear:                  235
        errorIOCount:                   0

指标写入文件 ./metrics.txt
基线的情况
寻道时间 : 62575752.000000, 带体磨损 : 40166852.000000, 电机磨损 : 16269.000000
实际调度的情况
寻道时间 : 6634620.000000, 带体磨损 : 10920961.000000, 电机磨损 : 235.000000
```

io数量为10000

```
bakcup: 0.862370
hdd: 0.878588

Key Metrics:
        ioCount:                        10000
        algorithmRunningDuration:       1113.805 (ms)
        memoryUse:                      0 (KB)
        addressingDuration:             9414630 (ms)
        readDuration:                   8403567 (ms)
        tapeBeltWear:                   17985706
        tapeMotorWear:                  319
        errorIOCount:                   0

指标写入文件 ./metrics.txt
基线的情况
寻道时间 : 141090101.000000, 带体磨损 : 89801875.000000, 电机磨损 : 36841.000000
实际调度的情况
寻道时间 : 9414630.000000, 带体磨损 : 17985706.000000, 电机磨损 : 319.000000
```