南開大學
Nankai University

计算机学院

算法导论大作业实验报告

基于倒排索引问题的算法设计

马浩祎

学号：2213559

专业：计算机科学与技术

2024 年 6 月 1 日

目录

1 摘要

Abstract

本次实验, 我针对搜索引擎应用领域的倒排索引大问题做了探索。首先我介绍了倒排索引求交问题的核心概念。然后尝试用本课程课上提到的一些高级算法如**动态规划，分治策略，贪心算法，图网络，网络流**，对搜索引擎应用领域的一些常见问题的解决做出了尝试。特别地，我对于条件成熟的倒排索引求交问题更是深入设计了**分治策略的优化算法**，成功降低了时间复杂度，并在实际数据集上给出了大量查询的性能指标，科学严谨地验证了算法的正确性和高效性，体现了理论算法和实际应用的完美结合!

关键词： 倒排索引，集合求交，算法设计，分治策略

2 问题描述

2.1 倒排索引

图 2.1: 当代搜索引擎

在现代，随着科技的迅速发展，业界出现了各式各样的搜索引擎，如图4.2。在搜索引擎里，索引是检索数据最有效率的方式，但考虑搜索引擎的如下特点:

- 搜索引擎面对的是海量数据。像 Google，百度这样大型的商业搜索引擎索引都是亿级甚至百亿级的网页数量
- 搜索引擎使用的数据操作简单。一般而言, 只需要增、删、改、查几个功能, 而且数据都有特定的格式, 可以针对这些应用设计出简单高效的应用程序。
- 搜索引擎面临大量的用户检索需求。这要求搜索引擎在检索程序的设计上要分秒必争, 尽可能的将大运算量的工作在索引建立时完成, 使检索运算尽量的少。

我们就不能去构建简单的索引了，早在 1958 年，IBM 就在一次会议上展示了一台“自动索引机器”。在当今的搜索引擎里也经常能看到它的身影，它就是——倒排索引。倒排索引又叫反向索引，它是一种逆向思维运算，是现代信息检索领域里面最有效的一种索引结构。它多使用在全文搜索下，是

一种通过映射来表示某个单词在一个文档或者一组文档中的存储位置的索引方法。在各种文档检索系统中，它是最常用的数据结构之一。

那么既然称其为倒排索引，反向索引，对应的，我们需要额外理解一下什么是正向索引。所谓正向索引，就是当用户发起查询时，搜索引擎会扫描索引库中的所有文档，找出所有包含关键词的文档，这样依次从文档中去查找是否含有关键词的方法。那很显然，这个搜索量是巨大的，算法效率低。

再来清楚的看一下两者的索引结构：

- 正向：文档-> 单词、单词、单词……
- 反向：单词-> 文档、文档、文档……

我们假设有一个数据集，包含 n 个网页或文档，现在我们想将其整理成一个可索引的数据集。我们当然可以按照正向索引，逐一遍历网页，为其内部的单词建立索引，但是正如前文所说，这样效率低下。具体地，利用倒排索引，我们可以为数据集中的每篇文档选取一个文档编号，使其范围在 [1,n] 中。其中的每一篇文档，都可以看做是一组词的序列。则对于文档中出现的任意一个词，都会有一个对应的文档序列集合，该集合通常按文档编号升序排列为一个升序列表，即称为倒排列表（Posting List）。所有词项的倒排列表组合起来就构成了整个数据集的倒排索引。这和我们的索引结构是相符的。

2.2 倒排索引求交

有了倒排索引后，搜索引擎在查询处理过程中，需要对这些索引集合进行求交操作，这个也是运算时间占比非常大的部分。假定用户提交一个 k 个词的查询，查询词分别是 $t_1, t_2, \ldots, t_k$，这 k 个关键词对应的倒排列表为 $\ell_{t_1}, \ell_{t_2}, \ldots, \ell_{t_k}$ 求交算法返回 $\cap_{1 \leq i \leq k} l\left(t_i\right)$。

首先求交会按照倒排列表的长度对列表进行升序排序，使得：

$$\left|l\left(t_1\right)\right| \leq \left|l\left(t_2\right)\right| \leq \cdots \leq \left|l\left(t_k\right)\right|$$

然后求交操作返回 k 个倒排列表的公共元素 $\cap_{1 \leq i \leq k} l\left(t_i\right)$。

举个例子：比如查询 “2024 Spring ”，搜索引擎会在索引中找到 “2024”，“Spring” 对应的倒排列表，假定为:

$$l(2024) = (13, 14, 15, 16, 17)$$

$$l(Spring) = (4, 8, 11, 13, 14, 16)$$

这是排好序的倒排列表，下面求交获得：

$$l(2024) \cap l(Spring) = (13, 14, 16)$$

但是这里只给出了输入和输出。如何进行求交操作并未提出，而如何设计出高效的求交算法正是我研究的问题关键。其实这个问题如果在没有进行索引压缩时，它的本质就是多个有序数组的求交集问题。

3 实验相关背景

3.1 实验环境

Windows 平台下，硬件：CPU：Intel(R) Core(TM)i9-14900HX,24 核，基准速度 2.2GHz，内存容量 16GB，Cache L1: 2.1MB;L2: 32.0MB;L3: 36.0MB。软件：编程 IDE 为 Visual Studio，其内置

编译器，这里不再介绍。

3.2 数据准备

本问题研究所采用的数据集是截取自 GOV2 数据集的子集，它包含 1000 条查询，见同目录 **ExpQuery** 文件；及 1756 条倒排索引，见同目录 **ExpIndex** 二进制文件。具体格式如下：

- ExpIndex 是二进制倒排索引文件，所有数据均为四字节无符号整数（小端）。格式为：[数组 1] 长度，[数组 1]，[数组 2] 长度，[数组 2]....
- ExpQuery 是文本文件，文件内每一行为一条查询记录；行中的每个数字对应索引文件的数组下标（term 编号）。

值得说明，对于第一个文件，索引下标是从 0 开始的。为了让我们的研究更加清晰，我进行了初步的数据分析，见 **QueryInfo** 文件，以第 1 个查询为例，它包含此查询包含的索引个数以及对应索引的长度。文本格式为

- 第 1 条查询信息:
- 第 1 个列表 ID: 1116 长度: 268
- 第 2 个列表 ID: 580 长度: 30000
- 第 3 个列表 ID: 1613 长度: 30000

我还给出了 **Array MaxValue** 数据预分析文件，它包含每个索引列表的 ID、最大值和长度。经过观察，这些列表包含的 DocID（也就是长度）最多 30000 个，最大的 DocID 达 25205174。如何对这样大规模的数据集进行有效的算法设计确实是一个挑战！也因此没有给出每个列表的具体数据的文件，但是在代码中，读取的所有索引数据均存在 arrays 二维向量内，读者可以添加输出代码，根据自身需要输出查看即可。

3.3 性能测试方法

本问题研究的性能测试主要以时间为主，单位均为秒，时间测试方法如下：

```
LARGE_INTEGER freq, start, end0;
QueryPerformanceFrequency(&freq);
...Initial...
QueryPerformanceCounter(&start);
...Function...
QueryPerformanceCounter(&end0);
double elapsedSeconds = static_cast<double>(end0.QuadPart - start.QuadPart) / freq.QuadPart;
```

同时我也在同目录下提前给出我前 100 条查询实验的用时表格 **Time** 文件。后面 900 条可以作为全新的输入样例供读者检验。

3.4 实验正确性

由于算法优化的前提是保证结果的正确性，所以我在问题研究中的相关实验都保证结果是正确的。我首先给出了正确结果，这个的生成是由 **stl 库 set_intersection 函数生成的标准结果**，并保存在同目录 **Result** 文件下。读者可以自行测试相关源文件验证结果，这里提供一种验证方法：**新建 VS 项目，然后将目标源文件和上述的两个 ExpQuery，ExpIndex 文件导入到项目下即可运行！**这是很简便的。

4 算法设计

关于本问题的研究，首先介绍我设计的测试框架，这是针对所有本实验讨论的算法的，后面的算法代码我也只给出算法部分，而不再介绍框架部分。因为将测试和算法抽离开来是有益于我们的独立算法设计的。

4.1 测试框架

下面我给出附有清晰注释的测试框架代码，相关解释在其中均可以找到，以便读者迅速掌握我们的测试逻辑，以便更好的进行验证。

```
#include <bits/stdc++.h>
#include <Windows.h>
using namespace std;
//保存索引
vector<vector<uint32_t>> arrays;
//结果保存
set<uint32_t> res;

int main() {
    //索引文件读入并保存
    ifstream file("ExpIndex", ios::binary);
    if (!file) {
        cerr << "Failed to open the file." << endl;
        return 1;
    }

    uint32_t arrayLength;
    while (file.read(reinterpret_cast<char*>(&arrayLength), sizeof(arrayLength))) {
        std::vector<uint32_t> array(arrayLength);
        file.read(reinterpret_cast<char*>(array.data()), arrayLength * sizeof(uint32_t));
        arrays.push_back(array);
    }
    file.close();
    //查询文件打开，准备读取
```

```
25      ifstream queryFile("ExpQuery");
26      if (!queryFile) {
27          cerr << "Failed to open the query file." << endl;
28          return 1;
29      }
30      //计时变量
31      LARGE_INTEGER freq, start, end0;
32      QueryPerformanceFrequency(&freq);
33
34      string line;
35      int queryCount = 1;
36      int targetQuery;
37
38      // 让用户输入查询编号
39      cout << " 请输入查询编号 (1-1000 之间的整数执行单条查询，其他整数执行所有查询，非整数向下取整)
40      cin >> targetQuery;
41      //处理到指定查询的逻辑
42      while (getline(queryFile, line)) {
43          if (targetQuery >= 1 && targetQuery <= 1000 && queryCount != targetQuery) {
44              queryCount++;
45              continue;
46          }
47          //指定查询的处理
48          uint32_t k = 0;
49          stringstream ss(line);
50          //保存这一行查询需要的索引下标
51          vector<uint32_t> queryIndices;
52          uint32_t row_idx;
53          bool flag = true;
54          while (ss >> row_idx) {
55              queryIndices.push_back(row_idx);
56              k++;
57          }
58          //开始计时
59          QueryPerformanceCounter(&start);
60
61              //索引求交的算法设计
62
63          QueryPerformanceCounter(&end0);
64
65          double elapsedSeconds = static_cast<double>(end0.QuadPart - start.QuadPart) / freq.QuadPart;
66          //输出查询耗时和查询结果
```

```
67          cout << "Query " << queryCount << ": " << elapsedSeconds << " seconds" << endl;
68          cout << "Result size: " << res.size() << endl;
69          //处理到指定查询的逻辑，指定的情况下直接跳出即可
70          if (targetQuery >= 1 && targetQuery <= 1000) {
71              break;
72          }
73          res.clear();
74          Flag.clear();
75          queryCount++;
76      }
77      queryFile.close();
78      return 0;
79  }
```

简单总结一下，我设计的测试框架有着如下特点：

- 计时准确。计时机制完全计时算法的执行时间，忽略其他操作，这是因为基于搜索引擎的实际问题，一些存储排序的操作均可以离线进行。算法引起的时间影响均在用户输入条目后搜索引擎进行求交操作部分。因此计时部分可以准确反映算法的设计性能。
- 输入安全。保证输入的查询编号全覆盖，防止恶意输入导致程序异常。
- 较好的独立性。可以方便地修改算法部分而保证框架的稳定性。
- 验证便捷。可以输出查询结果，验证实际结果是否和先前提到的 Result 文件结果相同。

4.2 经典算法设计

4.2.1 算法实现

这个问题经典的算法是按表求交算法，设计思想是：先使用两个表进行求交，得到中间结果再和第三条表求交，依次类推直到求交结束。这样求交的好处是，每一轮求交之后的结果都将变少，因此在接下来的求交中，计算量也将更少。下面我们看伪代码：

Algorithm 1 表求交串行算法

Input: $\boldsymbol{l}(\boldsymbol{t}_1), \boldsymbol{l}(\boldsymbol{t}_2), \ldots, \boldsymbol{l}(\boldsymbol{t}_k)$, Sorted by $|\boldsymbol{l}(\boldsymbol{t}_1)| \leq |\boldsymbol{l}(\boldsymbol{t}_2)| \leq \cdots \leq |\boldsymbol{l}(\boldsymbol{t}_k)|$

Output: $\cap_{1 \leq i \leq k} l(t_i)$

function FUNCTION($\boldsymbol{l}(\boldsymbol{t}_1), \boldsymbol{l}(\boldsymbol{t}_2), \ldots, \boldsymbol{l}(\boldsymbol{t}_k)$)
  $S \leftarrow \boldsymbol{l}(\boldsymbol{t}_1)$
  for $i = 2 \rightarrow k$ **do**
    for *each element* $\boldsymbol{e} \in \boldsymbol{S}$ **do**
      $found = find(e, \boldsymbol{l}_i)$
      if $found = false$ **then**
        Delete e *from* S
      end if
    end for

end for
end function

核心算法代码如下：

```
//循环每一个表
for (auto idx : queryIndices) {
    if (flag) {//第一个列表
        for (auto elem : arrays[idx]) {
            res.insert(elem);
            Flag.push_back(true);
        }
        flag = false;
    }
    else {//其他列表
        vector<uint32_t> unfound;
        //遍历结果集，未出现则删掉
        for (auto elem : res) {
            bool flag0 = false;
            for (size_t i = 0; i < arrays[idx].size(); i++) {
                if (elem == arrays[idx][i]) {
                    flag0 = true;
                    break;
                }
            }
            if (!flag0) {
                unfound.push_back(elem);
            }
        }
        for (size_t i = 0; i < unfound.size(); i++) {
            res.erase(unfound[i]);
        }
    }
}
```

我针对上述代码作出算法层面的分析：

4.2.2 时间复杂度分析

假设：

- k 是 `queryIndices` 的长度，即要处理的数组个数。
- n 是每个数组的平均长度。

在第一次遍历时（即 `flag` 为 `true` 时），将第一个数组的所有元素插入 `res` 集合。此时的复杂度是：

- 插入操作：$O(1)$（均摊时间复杂度）

因此，总时间复杂度为：$O(n)$。在后续遍历中，每个元素需要检查 `res` 中的每个元素是否在当前数组中。

- 对于每个元素 $elem$ 在 `res` 中的检查复杂度是 $O(n)$（线性扫描当前数组）。
- 因此，对于 `res` 中的每个元素，检查其是否存在于当前数组的复杂度是：$O(n) \times O(n) = O(n^2)$。
- 因此，对于 $k-1$ 个数组，求交集的总时间复杂度是：$O((k-1) \times n^2)$。

综合上述分析，总时间复杂度为：

$$O(n) + O((k-1) \times n^2) = O(k \times n^2)$$

4.2.3 空间复杂度分析

- `res` 集合的空间复杂度为 $O(n)$，因为最多存储所有元素。
- `unfound` 向量的空间复杂度也是 $O(n)$，因为在最坏情况下，需要存储所有未找到的元素。
- 其他变量（如 `flag` 和 `Flag`）的空间复杂度是常数级别 $O(1)$。

综合来看，总空间复杂度为：

$$O(n)$$

4.3 课程学习——分治算法

看到这个 $O(k \times n^2)$ 的时候，我想到算法导论课上提到的，这么敏感的时间复杂度，好像可以试试分治策略！而且观察问题本身，也很适合分治算法的优化。

4.3.1 分治优化分析

分治算法是一种通过将问题分解为更小的子问题，分别解决这些子问题，然后将它们的结果合并来解决原问题的方法。在这个多列表求交算法中，我们需要计算多个列表的交集。该问题可以自然地分解为计算两个列表的交集，然后将结果与下一个列表求交，如此进行下去。每一步都可以看作是求两个集合的交集，而这个双表求交的过程很适合分治策略，下面我给出这个子问题的结构性分析：

- 有序性：假设输入的两个列表 a 和 b 都是有序的，这使得我们可以在子问题中高效地进行二分搜索和合并操作。有序性使得我们在分割问题时可以利用中间元素将列表分为近似相等的两部分，从而保持子问题的平衡。
- 可分性：分治法要求问题可以分解成更小的子问题，并且这些子问题的解可以合并成原问题的解。交集问题可以通过将列表分成两个子列表来解决，每个子列表的交集可以递归地求解，最后将结果合并。
- 减少比较次数：通过分治法，每次可以有效地排除掉一半的元素，从而减少不必要的比较次数。这是因为在有序列表中，如果我们知道某个元素不在交集中，那么我们可以立即排除掉剩余的一半元素。

通过上述分析，可以看出该双表，多列表求交算法适合分治优化。其问题结构特点都体现了分治优化的潜力。我可以采用分治策略，可以将原问题分解为更小的子问题，分别求解并合并，提升算法的效率。

4.3.2 算法实现

但是这个分治化的过程充满挑战，分治的具体实现也比较困难，但还好列表已经事先有序了，这为我的算法设计提供了便利。下面我们就来尝试进行分治的算法优化！先给出求交集算法的伪代码：

Algorithm 2 求两个数组的交集

Input: 两个有序数组 a 和 b

Output: 返回两个数组的交集

```
function INTERSECT(a, b)
    result ← empty list
    it_a ← a.begin()
    it_b ← b.begin()
    while it_a ≠ a.end() and it_b ≠ b.end() do
        if *it_a < *it_b then
            ++it_a
        else if *it_b < *it_a then
            ++it_b
        else
            result.push_back(*it_a)
            ++it_a
            ++it_b
        end if
    end while
    return result
end function
```

再给出递归分治的伪代码：

Algorithm 3 分治求交集

Input: 两个有序数组 a 和 b

Output: 返回两个数组的交集

```
function DIVIDEANDCONQUERINTERSECTION(a, b)
    if a.size() < 10 or b.size() < 10 then return INTERSECT(a, b)
    end if
    mid_a ← a.size()/2
    lower_b ← lower_bound(b.begin(), b.end(), a[mid_a])
    upper_b ← upper_bound(b.begin(), b.end(), a[mid_a])
    left_a ← vector(a.begin(), a.begin() + mid_a)
    right_a ← vector(a.begin() + mid_a + 1, a.end())
    left_b ← vector(b.begin(), lower_b)
    right_b ← vector(upper_b, b.end())
```

$left\_intersect \leftarrow$ DivideAndConquerIntersection($left\_a, left\_b$)
$right\_intersect \leftarrow$ DivideAndConquerIntersection($right\_a, right\_b$)
$mid\_intersect \leftarrow$ vector()
if $lower\_b \neq b$.end() **and** $*lower\_b == a[mid\_a]$ **then**
    $mid\_intersect$.push_back($a[mid\_a]$)
end if
$left\_intersect$.insert($left\_intersect$.end(), $mid\_intersect$.begin(), $mid\_intersect$.end())
$left\_intersect$.insert($left\_intersect$.end(), $right\_intersect$.begin(), $right\_intersect$.end())
return $left\_intersect$
end function

然后，我们继续看一下核心代码的实现，以明晰主函数内是如何进行调用处理的，详细的算法解释也以注释形式给出：

```
// 定义两个有序数组的交集函数
vector<uint32_t> intersect(const vector<uint32_t>& a, const vector<uint32_t>& b) {
    vector<uint32_t> result; // 用于存储交集结果
    auto it_a = a.begin(); // 迭代器指向 a 数组的起始位置
    auto it_b = b.begin(); // 迭代器指向 b 数组的起始位置

    // 遍历两个数组直到其中一个数组结束
    while (it_a != a.end() && it_b != b.end()) {
        if (*it_a < *it_b) {
            ++it_a; // 如果 a 中的当前元素小于 b 中的当前元素，则移动 a 的迭代器
        }
        else if (*it_b < *it_a) {
            ++it_b; // 如果 b 中的当前元素小于 a 中的当前元素，则移动 b 的迭代器
        }
        else {
            result.push_back(*it_a); // 如果 a 和 b 中的当前元素相等，则将该元素添加到结果中
            ++it_a; // 移动 a 的迭代器
            ++it_b; // 移动 b 的迭代器
        }
    }
    return result; // 返回交集结果
}

// 分治求交集函数
vector<uint32_t> divideAndConquerIntersection(const vector<uint32_t>& a,\
const vector<uint32_t>& b) {
    // 当子列表小于 10 时，直接求交集
    if (a.size() < 10 || b.size() < 10) {
        return intersect(a, b);
```

```
30      }
31
32      // 找到 a 的中间元素
33      int mid_a = a.size() / 2;
34
35      // 在 b 中找到与 a 的中间元素相等的元素范围
36      auto lower_b = lower_bound(b.begin(), b.end(), a[mid_a]);
37      auto upper_b = upper_bound(b.begin(), b.end(), a[mid_a]);
38
39      // 将 a 分成左半部分和右半部分
40      vector<uint32_t> left_a(a.begin(), a.begin() + mid_a);
41      vector<uint32_t> right_a(a.begin() + mid_a + 1, a.end());
42
43      // 将 b 分成小于 a 中间元素的左半部分和大于 a 中间元素的右半部分
44      vector<uint32_t> left_b(b.begin(), lower_b);
45      vector<uint32_t> right_b(upper_b, b.end());
46
47      // 递归求左半部分的交集
48      vector<uint32_t> left_intersect = divideAndConquerIntersection(left_a, left_b);
49
50      // 递归求右半部分的交集
51      vector<uint32_t> right_intersect = divideAndConquerIntersection(right_a, right_b);
52
53      // 创建一个向量存储 a 中间元素与 b 中间元素相等的部分，中间元素的巧合情况是极容易被忽略的!
54      vector<uint32_t> mid_intersect;
55      if (lower_b != b.end() && *lower_b == a[mid_a]) {
56          mid_intersect.push_back(a[mid_a]);
57      }
58
59      // 将左半部分的交集、中间元素、右半部分的交集按顺序合并到一起
60      left_intersect.insert(left_intersect.end(), mid_intersect.begin(), mid_intersect.end());
61      left_intersect.insert(left_intersect.end(), right_intersect.begin(), right_intersect.end());
62
63      // 返回合并后的结果
64      return left_intersect;
65  }
66
67  //主函数框架内的代码句
68  if (queryIndices.size() > 1) {
69      //多列表遍历，逐一合并
70      vector<uint32_t> result = arrays[queryIndices[0]];
71      for (size_t i = 1; i < queryIndices.size(); ++i) {
```

```
72            result = divideAndConquerIntersection(result, arrays[queryIndices[i]]);
73        }
74        res.insert(result.begin(), result.end());
75    }
76    else if (queryIndices.size() == 1) {
77        //单个列表直接赋值即可，本问题数据集无此情况
78        res.insert(arrays[queryIndices[0]].begin(), arrays[queryIndices[0]].end());
79    }
80
```

下面我同样针对分治策略的算法给出相应分析；

4.3.3 时间复杂度分析

假设每个数组的大小为 n，我们来分析 `divideAndConquerIntersection` 和 `intersect` 函数的时间复杂度。

intersect 函数的时间复杂度是 $O(n)$，因为它遍历两个列表并寻找公共元素。

divideAndConquerIntersection 函数将两个有序列表进行分治处理。每次递归调用会将两个列表分为两半，这类似于合并排序的过程。因此，每一层递归的时间复杂度为 $O(n)$，递归深度为 $O(\log n)$。

因此，总的时间复杂度是：

$$O(n \log n)$$

但是，由于我们在主函数中需要对多个列表进行交集操作，我们来看整体的时间复杂度：

- 对第一个列表和第二个列表求交集：$O(n \log n)$
- 对结果与第三个列表求交集：$O(n \log n)$
- 对结果与第四个列表求交集：$O(n \log n)$
- ...

假设我们有 k 个列表，那么总的时间复杂度为：

$$O(k \times n \log n)$$

4.3.4 空间复杂度分析

intersect 函数的空间复杂度是 $O(n)$，因为结果数组的大小最多是 n。

divideAndConquerIntersection 函数递归调用深度为 $O(\log n)$，但每次调用只使用常数级别的额外空间（不考虑返回值）。返回值占用的空间为 $O(n)$。

所以 divideAndConquerIntersection 函数的空间复杂度是：

$$O(n)$$

在主函数中，我们使用了一个结果集合 `res`，这个集合的空间复杂度为 $O(n)$。

整体的空间复杂度为：

$$O(n)$$

4.4 性能比较

在保证结果相同的情况下，这里我给出部分查询测试性能的结果，如表1，这里提供查询 ID 以保证数据真实性 (单位为秒)：

查询 ID	列表平均长度	经典算法	分治
76	872	0.0031056	0.0007836
37	937	0.0049116	0.0004626
59	1668	0.0200710	0.0011471
52	5813	0.0018554	0.0016409
18	7143	0.3133930	0.0029136
9	9962	0.8994020	0.0047349
70	13534	0.7322500	0.0055221
80	15012	0.0103185	0.0045334
34	15333	0.2312300	0.0042937
20	15538	0.3776250	0.0042052

表 1: 性能结果表

为了直观展示随规模的对比效果，我给出对比折线图，如图4.2

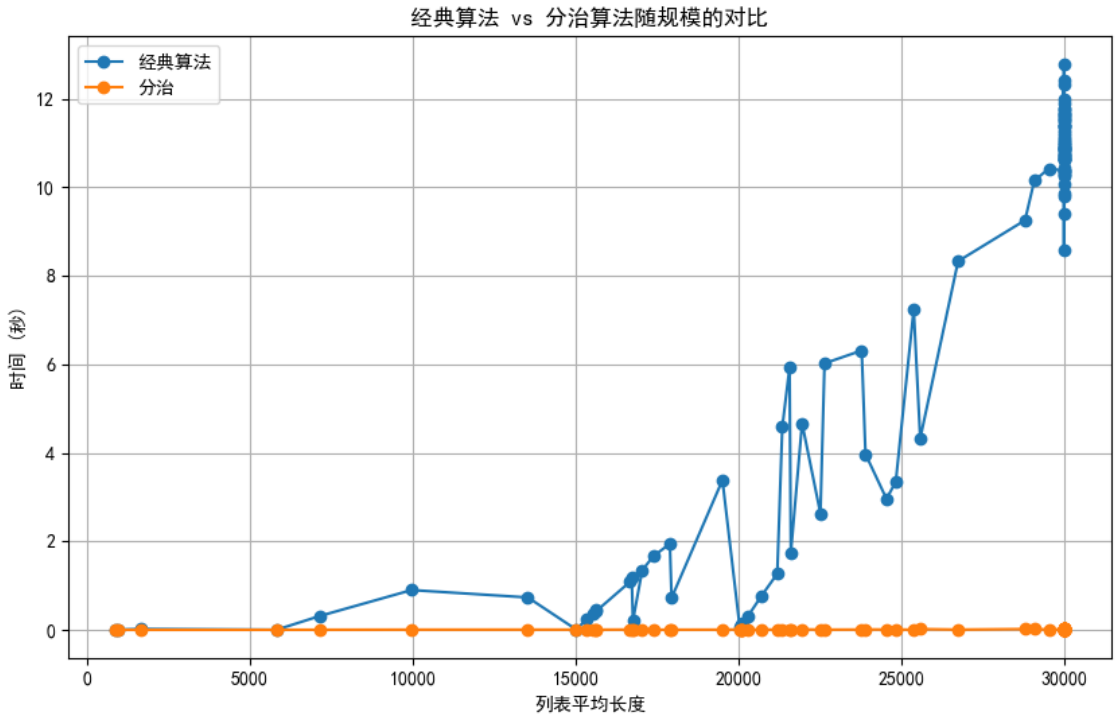

图 4.2: 性能折线图

我们可以很清楚的看到，我成功地应用分治策略进行算法的设计和编程并取得了性能的巨大提升！尤其是在规模大的时候，这验证了我们的复杂度分析。在成功应用分治策略后，我深刻体会到了算法设计中的重要原则：复杂问题可以通过拆分为更小的子问题来高效解决。在本问题中是这样体现的：多列表转化为双列表形式逐一求交，双列表再分治处理！这不仅仅是算法中的一个技巧，更是一种解决问题的思维方式。通过将一个庞大而复杂的问题分解为若干个相对简单的部分，再逐一攻克，每一步

都变得更加明确和易于管理。最终，不仅达到了性能上的巨大提升，还让我更加自信地面对今后的编程挑战。

5 其他问题

针对搜索引擎的大应用范围下，还有很多问题可以应用本课程学到的算法设计方法。本节简要给出一些调研问题，和与课程结合的可能算法思路（这些问题由于一些原因限制，此处无法进行实验验证，在未来条件成熟后可能会进行相关实践），这里涉及分治算法的便不再阐述，因为我们已经对其进行了深入分析和实验证明。

5.1 倒排索引的优化压缩

问题描述：在保证查询速度的前提下，压缩倒排索引以节省存储空间。可以应用贪心算法和动态规划找到最佳的压缩策略，例如最优分词策略或最优编码方案。

可能的算法设计：

- 贪心算法：针对文档频率高的词汇采用较短的编码，而频率低的词汇采用较长的编码。
- 动态规划：通过构建状态转移方程，找到最优的压缩方式，使得压缩后的索引在存储和查询速度之间达到平衡。

5.1.1 贪心算法

设有一组词汇 $W = \{w_1, w_2, \ldots, w_n\}$，每个词汇的频率为 f_i。我们希望为每个词汇分配一个编码，使得总编码长度最短。

- 对词汇按频率 f_i 从高到低排序。
- 为频率最高的词汇分配最短的编码。
- 思路很像我们课上提到的 Huffman 编码！

5.1.2 动态规划

设状态 $dp[i]$ 表示前 i 个词汇的最优压缩长度。我们可以设计这样的转移方程：

$$dp[i] = \min_{k=1}^{i}(dp[k-1] + \text{cost}(k, i))$$

其中，$\text{cost}(k, i)$ 表示第 k 到第 i 个词汇的压缩成本。这样经过回溯可以求出最优的压缩方案。

5.2 倒排索引的合并与更新

问题描述：在动态数据环境中高效地合并和更新倒排索引。可以使用动态规划来优化合并策略，或者使用网络流算法来优化数据传输和合并过程。

可能的算法设计：

- 动态规划：确定多个部分索引合并的最优顺序。
- 网络流算法：建模为最小成本最大流问题，以优化合并和更新过程中的数据传输。

5.2.1 动态规划

设 $dp[i]$ 表示合并前 i 个部分索引的最小成本。我们可以尝试设计转移方程：

$$dp[i] = \min_{k=1}^{i-1}(dp[k] + \text{merge\_cost}(k, i))$$

其中，$\text{merge\_cost}(k, i)$ 表示从第 k 个部分索引合并到第 i 个部分索引的成本。

5.2.2 网络流算法

基于我的理解，我们或许可以将部分索引作为节点，数据传输成本作为边权重，构建一个有向图。使用最小成本最大流算法求解最优数据传输方案，这里不再赘述。

5.3 排名算法的优化

问题描述：在倒排索引的基础上优化搜索结果的排名。可以使用贪心算法或动态规划来优化排名函数，使其能够高效计算相关性得分。

可能的算法设计：

- 贪心算法：根据词频和文档评分，逐步选择相关性最高的文档进行排名。
- 动态规划：建立状态转移方程，优化每一步选择的相关性得分。

5.3.1 贪心算法

设有文档集合 $D = \{d_1, d_2, \ldots, d_n\}$，每个文档的相关性得分为 r_i。我们希望选择得分最高的文档进行排名。

- 对文档按相关性得分 r_i 从高到低排序。
- 依次选择得分最高的文档进行排名。

5.3.2 动态规划

设 $dp[i]$ 表示前 i 个文档的最优排名得分。我们可以设计转移方程为：

$$dp[i] = \max_{k=1}^{i-1}(dp[k] + \text{score}(k, i))$$

其中，$\text{score}(k, i)$ 表示第 k 到第 i 个文档的相关性得分。这样可以给出较优的搜索结果！

5.4 构建支持多媒体（如图像、视频）搜索的倒排索引

问题描述：在多媒体数据上构建倒排索引，优化多媒体特征的索引和匹配。可以使用图算法和动态规划来优化索引和匹配过程。

可能的算法设计：

- 图算法：将多媒体特征表示为图结构，利用图匹配算法优化搜索。
- 动态规划：优化多媒体特征的匹配过程，找到最优的匹配方案。

5.4.1 图算法

将每个多媒体特征表示为图的节点，特征之间的相似度作为边权重。使用图匹配算法（如 Hungarian 算法）优化特征匹配。

5.4.2 动态规划

设 $dp[i][j]$ 表示第 i 个查询特征与第 j 个多媒体特征匹配的最优得分。设计转移方程为：

$$dp[i][j] = \max(dp[i-1][j-1] + \text{similarity}(i,j), dp[i-1][j], dp[i][j-1])$$

其中，$\text{similarity}(i,j)$ 表示第 i 个查询特征与第 j 个多媒体特征的相似度。这其实和上一节很相似，不过是可以支持更多的查询特征来提供最优匹配。

6 总结

本次报告为算法导论课程的期末大作业实验报告，我针对搜索引擎应用领域的倒排索引大问题做了许多探索。我深入理解了倒排索引求交问题的核心概念，并尝试用本课程课上提到的一些高级算法如**动态规划，分治策略，贪心算法，图网络，网络流**，对搜索引擎应用领域的一些常见问题的解决做出了尝试。对于条件成熟的倒排索引求交问题更是深入设计了**分治策略的优化算法**，成功降低了时间复杂度，并在实际数据集上给出了大量查询的性能指标，科学严谨地验证了算法的**正确性和高效性**！这也正是算法应该具有的性质。

此外，这次成功的经验也让我意识到，不断学习和实践新的算法和编程方法是多么重要。算法设计不仅仅是写代码，更是对问题进行深度思考和优化的过程。通过分治策略，我不仅提升了代码的执行效率，还加深了对算法原理的理解。这种成就感和满足感，激励我继续探索和应用更多先进的算法策略，不断提高自己的编程水平和解决问题的能力。

最后，感谢老师本学期的讲授，您课上趣味的故事引入和严谨的算法论证，以及科学的复杂度分析都给我留下了深刻印象，这增强了我的问题结构分析和算法分析的能力。祝本课程组的所有老师工作顺利！