

苏州大学实验报告

院、系	计算机科学与技术	年级专业	2020 软件工程	姓名	高歌	学号	2030416018
课程名称	信息检索综合实践					成绩	
指导教师	贡正仙	同组实验者	无	实验日期	2023 年 4 月 12 日		

实验名称 实验 5 模糊匹配

一. 实验目的

了解模糊匹配的基本原理及其他相关知识，理解文本相关性、编辑距离等概念，学习并实现在文档列表中利用上一个实验中建立的倒排索引进行词语的模糊匹配。

二. 实验内容

编写程序，自选合适的算法，对提供的数据集 corpus.txt 展开模糊搜索。

预处理工作：

- (1) 把 corpus.txt 里的每句话当作是一个文档
- (2) 对每个文档用 jieba 分词软件进行分词生成一个新的 corpusnew.txt（保留原来的每句话后的回车换行，即新生成文件的行数要等于原来文件的行数）
- (3) 对 corpusnew.txt 新建作业 4 要求的倒排索引，不过文档编号要换成是句子行号

编写程序，自选合适的算法，在前面生成的倒排索引基础上实现模糊搜索。

输入：待搜索词

按下面的步骤测试算法正确性：

- (1) 倒排索引中存在待搜索词，直接用类似作业 4 的方法进行全词匹配查询
- (2) 如果 (1) 没有搜索到，即搜索词没有出现在倒排索引中，就要考虑使用不同的模糊策略进行改进。

输出：能够全匹配或者模糊匹配到待搜索词的句子编号

模糊策略 1

假设查询词是 A，当使用 fuzzywuzzy 库的 process.extract 方法可以获得与倒排索引中每个词的模糊查询匹配分数，将该分数记作 Q(A)；

联合阈值控制，若 $Q(A) > \delta$ 时认为查询结果不为空，返回匹配项对应的文档集；否则认为空，找不到匹配项；

联合排序控制，返回 top n 的模糊匹配，若 $n > 1$ ，按照匹配值大小排序，若 A and B 方式（B 是另一个查询词），可按照 A 和 B 的匹配均值排序；

建议同时控制 top n 和阈值。

模糊策略 2

策略 1 显然效率不高，表现用户需要等待很久才能得到结果。

所有除了前面的基于词的倒排索引之外，可以再联合建立一个基于字的倒排索引，比如存在如下情况：

- 搜索词的每一个字都不在原始文档中
- 搜索词的部分字可以在原始文档中找到
- 搜索词的每一个字都可以在原始文档中找到，但因为分词算法的原因没有出现在倒排索引中

三. 实验步骤和结果

注：代码使用 TypeScript 编写，运行时使用 ts-node。使用 Prettier 与 ESLint 作为代码格式化工具，代码风格遵从 TypeScript ESLint Recommended 标准。使用 fuzzball.js 作为 Python 库 fuzzywuzzy 的替代。

（一）实验步骤

1. 本实验逻辑稍有些复杂，这里对目录结构做一个简单解释：

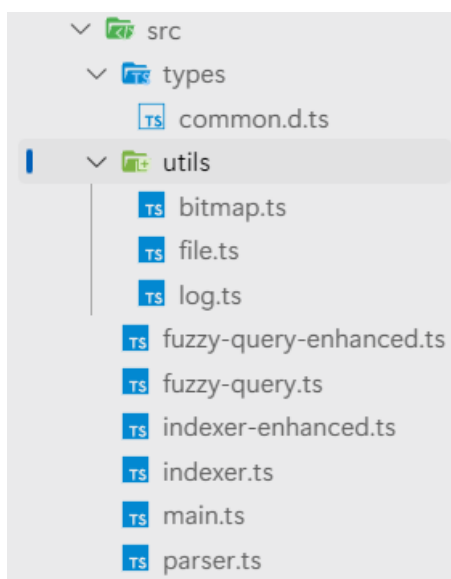


图 1 目录结构

其中，`types` 目录下包含了一些必要的类型定义；`utils` 目录下包含了一些工具函数，`bitmap.ts` 包含了与位图相关的函数，用来辅助集合操作如去重等，`file.ts` 包含了一些与文件相关的工具函数，`log.ts` 包含了一些与日志相关的工具函数；而`indexer.ts` 包含了构建与读取索引文件所需的相关函数，`indexer-enhanced.ts` 提供了一个增强版本的索引器，在之后会详细阐释；`parser.ts` 中包含了一个简单的语法分析器（lexer 与 parser），用来将布尔查询表达式转换成抽象语法树以便后续求值；`fuzzy-query.ts` 根据抽象语法树与索引字典模糊查询并输出最终结果，同样的，`fuzzy-query-enhanced` 包含了一个增强版本的模糊查询算法，它使用增强版本的索引字典，也会在之后解释。`main.ts` 则为主入口文件。

在这里，`common.d.ts`、`bitmap.ts`、`file.ts`、`parser.ts`、`indexer.ts` 与上一次实验基本一致。其

中有一些细微的改动会在之后涉及时顺带说明，这里不再详述。

而`log.ts`为了适应更复杂的日志输出功能换了一个比较复杂与晦涩的实现，以实现在输出日志时获取函数调用栈的能力，以实现更好的自定义日志。但由于日志实现并非核心功能，在这里并不给出代码实现，只简单阐释该如何使用其导出的`logged`高阶函数。

对于普通的函数，使用如下方式创建会输出日志的函数版本：

```
const loggedReadIndex = logged('Index read', readIndex);
```

该`loggedReadIndex`函数的用法与`readIndex`函数完全一致。只是运行结束后，将向终端打印`Index read in <dur> ms`，其中`<dur>`为运行该函数所用的毫秒数。

对于下面这样采用 Fluent interface 风格的函数：

```
fuzzyQuery('中国女排', {
  threshold: 60,
  limit: 5,
}).on(wordIndexMap),
```

使用这样的语法生成其日志版本：

```
const loggedFuzzyQuery = logged(
  ({ __callStack: [[, [input]]] }) => `Queried "${input}"`,
  fuzzyQuery,
  Nested.of(1),
);
```

其中，`Nested.of(1)`表示代理一层函数调用，即这里的`.on`。相比于上面`loggedReadIndex`的定义，这里将第一个参数改成了一个函数，它提取函数调用栈中的参数，以显示更好的日志。

调用栈`__callStack`的类型为`Array<[string | number | symbol, unknown[]]>`，其中的每个子数组有两个元素，第一个元素为调用的函数名，第二个元素为参数列表，如上面的`fuzzyQuery('中国女排', ...)`其`__callStack`应当为`[['fuzzyQuery', ['中国女排', { threshold: 60, limit: 5 }]], ['on', [wordIndexMap]]]`。

在这个例子中，若将上面对`fuzzyQuery('中国女排', ...)`的调用改为`loggedFuzzyQuery('中国女排', ...)`，将在终端输出`Queried “中国女排” in <dur> ms`，其中`<dur>`为运行该函数所用的毫秒数。

上面该日志函数使用方式的描述稍显晦涩，但并非本作业的重点。如不理解也无关系。

接下来将一一解释剩余文件中的代码逻辑。

2. 在`indexer.ts`中，为了适应本次作业的索引生成，对`generateIndex`函数稍作修改。相比于上一次作业的改动已经用不同颜色的背景标出。

```
/**
 * The strategy to generate the index.
 * @default IndexStrategy.BY_WORD
 */
export enum IndexStrategy {
  BY_WORD,
  BY_CHARACTER,
}

/**
 * Generate the index from corpus file and save it to another file.
```

```

* @param corpusPathname The path to the corpus file.
* @param targetPathname The path to the corpus file that is already
segmented.
* @param indexPathname The path to the index file.
* @param indexStrategy The index strategy.
* @returns The index map.
*/
export const generateIndex = async (
  corpusPathname: string,
  targetPathname: string,
  indexPathname: string,
  indexStrategy: IndexStrategy = IndexStrategy.BY_WORD,
): Promise<IndexMap> => {
  const lines = (await fs.readFile(corpusPathname, 'utf-8')).split('\n');

  const wordOccurs = new Map<string, Bitmap>();
  let targetFileContent = '';

  for (const [index, line] of lines.entries()) {
    const sentenceID = index + 1;
    const words =
      indexStrategy === IndexStrategy.BY_CHARACTER
        ? line.split('')
        : jieba.cut(line);

    targetFileContent += `${words.join(' ')}\n`;

    // Keep only Chinese words
    for (const word of words.filter((word) =>
      /^[u4e00-\u9fa5]+$/.test(word),
    )) {
      if (!wordOccurs.has(word)) {
        wordOccurs.set(word, createBitmap(lines.length));
      }
      (wordOccurs.get(word) as Bitmap).set(sentenceID);
    }
  }

  // Transform to a map of word -> [sentenceIDs]
  const indexMap = new Map(
    Array.from(wordOccurs.entries()).map(([word, bitmap]) => [
      word,
      R.range(0, lines.length + 1).filter((sentenceID) =>
        bitmap.isSet(sentenceID),
      ),
    ]),
  );

```

```

    },
  );

  // Write to target file
  targetFileContent = targetFileContent.slice(0, -1); // Remove the last
  newline
  await fs.writeFile(targetPathname, targetFileContent, 'utf-8');

  // Write to file
  await saveIndex(indexMap).toFile(indexPathname);

  return indexMap;
};

```

可以看到，首先由于本次作业要求输出 corpusnew.txt，加入了`targetPathname`参数，表示要输出的 corpusnew.txt 的文件路径，并且加入了一些相关逻辑。除此之外，为了处理按词索引和按字索引两种索引方式，引入了`indexStrategy`参数。按词索引时，使用 jieba 分词，按字索引时，简单将文档按字拆分即可。

该文件中的`readIndex`与`saveIndex`函数并未改变。

3. 与上一次作业类似，在`fuzzy-query.ts`中定义函数`evaluateAstFuzzy`。该函数对解析用户查询字符串后产生的 AST 根据索引字典进行求值，得到查询结果。

```

/**
 * The result of a fuzzy query.
 */
interface FuzzyQueryResult {
  sentenceID: number;
  scores: {
    total: number;
    [term: string]: number;
  };
}

const evaluateAstFuzzy = (
  ast: ASTNode,
  indexMap: IndexMap,
  byCharacter: boolean,
  allSentenceIDs: number[] = getAllSentenceIDs(indexMap),
): FuzzyQueryResult[] => {
  switch (ast.type) {
    case 'Term': {
      const term = ast.value;

      if (byCharacter)

```

```

        return allSentenceIDs.map((sentenceID) =>
calculateScoreByCharacter(sentenceID, term).on(indexMap));

        const fuzzyTerms = generateFuzzyTerms(term).on(indexMap);
        return allSentenceIDs.map((sentenceID) =>
calculateScoreByFuzzy(sentenceID, term).on(fuzzyTerms));
    }
    case 'And': {
        const left = evaluateAstFuzzy(ast.left, indexMap, byCharacter,
allSentenceIDs);
        const right = evaluateAstFuzzy(ast.right, indexMap, byCharacter,
allSentenceIDs);
        return mergeScores(left, right, MergeScoreStrategy.AND);
    }
    case 'Or': {
        const left = evaluateAstFuzzy(ast.left, indexMap, byCharacter,
allSentenceIDs);
        const right = evaluateAstFuzzy(ast.right, indexMap, byCharacter,
allSentenceIDs);
        return mergeScores(left, right, MergeScoreStrategy.OR);
    }
    case 'Not': {
        const operand = evaluateAstFuzzy(ast.operand, indexMap, byCharacter,
allSentenceIDs);
        return reverseScores(operand);
    }
}
};

```

可以观察到，与上一次实验不同的是，这一次不再输出唯一的匹配结果，而是输出一个数组，包含了每个句子（文档）的匹配分数。在这里，`scores` 中包含了对每一个词语的查询分数，其中的 `total` 为每个词语查询分数的加权得分（并非简单平均，视查询操作符如 AND/OR/NOT 等会有不同的计算方式，下面会详述）。

如`中国男排 and 奥运会`就会得到类似于`{ total: 83.5, '中国男排': 67, '奥运会': 100 }`这样的分数，表示这个句子对于“中国男排”的匹配得分是 67，对“奥运会”的匹配得分是 100，由于使用了“and”操作符，因此将它们平均得到 83.5 作为最终得分。

可以看到，上面的函数定义中使用了若干辅助函数。首先查看一下`mergeScores`和`reverseScores`这两个辅助函数的定义，它们负责将复杂查询的结果综合起来：

```

/**
 * The strategy to merge the scores of two fuzzy query results.
 */
enum MergeScoreStrategy {
    AND,
    OR,
}

```

```

/**
 * Merge the scores of two fuzzy query results.
 * @param left The left fuzzy query result.
 * @param right The right fuzzy query result.
 * @param strategy The strategy to merge the scores.
 * @returns
 */
const mergeScores = (
  left: FuzzyQueryResult[],
  right: FuzzyQueryResult[],
  strategy: MergeScoreStrategy,
): FuzzyQueryResult[] => {
  const mergeScore = (
    left: FuzzyQueryResult,
    right: FuzzyQueryResult,
    strategy: MergeScoreStrategy,
  ) => ({
    sentenceID: left.sentenceID,
    scores: {
      ...left.scores,
      ...right.scores,
      total:
        strategy === MergeScoreStrategy.AND
          ? (left.scores.total + right.scores.total) / 2
          : Math.max(left.scores.total, right.scores.total),
    },
  });

  return left.map((leftResult) => {
    const rightResult = right.find(
      (r) => r.sentenceID === leftResult.sentenceID,
    );

    if (rightResult === undefined) return leftResult;

    return mergeScore(leftResult, rightResult, strategy);
  });
};

/**
 * Reverse the scores of the fuzzy query results.
 * @param results The fuzzy query results.
 * @returns
 */

```

```
const reverseScores = (results: FuzzyQueryResult[]): FuzzyQueryResult[]
=> {
  const reverseScore = (result: FuzzyQueryResult) => ({
    ...result,
    scores: {
      ...result.scores,
      total: 100 - result.scores.total,
    },
  });

  return results.map(reverseScore);
};
```

上面的代码看起来比较复杂，实际思路其实非常简单：

- 对于 AND 操作符，将左右分数平均，即相加除以二
- 对于 OR 操作符，取左右分数中最高的一个
- 对于 NOT 操作符，使用 100 减去分数（分数的范围是 0-100）

为了有一个更直观的理解，在这里提前展示一下搜索“中国男排 or (奥运会 and not 人民)”得到的前几个结果：

```
Queried "中国男排 or (奥运会 and not 人民)" in 76ms.
[
  {
    sentenceID: 266,
    scores: { total: 100, '中国男排': 67, '奥运会': 100, '人民': 0 }
  },
  {
    sentenceID: 262,
    scores: { total: 83.5, '中国男排': 67, '奥运会': 67, '人民': 0 }
  },
  {
    sentenceID: 263,
    scores: { total: 83.5, '中国男排': 67, '奥运会': 67, '人民': 0 }
  },
  {
    sentenceID: 268,
```

可以看到，对于第一个结果，“奥运会”的得分是 100，“人民”的得分是 0，则“奥运会 and not 人民”的得分为 $(100 + (100 - 0)) / 2$ 为 50，而“中国男排”的得分是 100，则最终得分为 $\max(100, 50)$ 得到 50。剩余的几项同理。

然后，展示上面`evaluateAstFuzzy`用到的、包含了本作业核心算法的两个函数`calculateScoreByCharacter`和`calculateScoreByFuzzy`。它们分别实现了按字的模糊匹配与调用 fuzzball.js 的模糊匹配。

首先展示调用 fuzzball.js 的模糊匹配算法实现：

```
type FuzzyTerm = [string, [number, number[]]];
```



```

const generateFuzzyTerms = (term: string) => ({
  on: (indexMap: IndexMap): FuzzyTerm[] =>
    fuzz
      .extract(term, [...indexMap.keys()])
      .map(([t, score]) => [t, [score, indexMap.get(t) ?? []]]),
});

/**
 * Calculate the score of a sentence with a specific term by the fuzzy score.
 * @param sentenceID
 * @param term
 * @returns
 */
const calculateScoreByFuzzy = (sentenceID: number, term: string) => ({
  on: (fuzzyTerms: FuzzyTerm[]): FuzzyQueryResult => {
    const score =
      fuzzyTerms.find(([, [, sentenceIDs]]) =>
        sentenceIDs.includes(sentenceID),
      )?.[1][0] ?? 0;

    return {
      sentenceID,
      scores: {
        total: score,
        [term]: score,
      },
    };
  },
});

```

先解释一下`generateFuzzyTerms`函数。它首先调用`fuzz.extract`，得到所有索引字典中存在的单词（`indexMap.keys()`）与搜索词`term`的匹配分数。然后，对于每一个[单词，分数]元组，将其转换为[单词，[分数，出现该单词的文档编号数组]]这样的形式，以供后续处理。

在`calculateScoreByFuzzy`函数中，输入文档编号与需要计算分数的搜索词，然后在刚刚计算出的`fuzzyTerms`中，搜索第一个包含了该文档编号的单词所对应的分数（`fuzz.extract`输出的结果按分数从高到低排序，因此这里得到的分数应当是可能的分数中最高的），若没有搜索到，则分数为0。

这里的描述稍微有些抽象，考虑这样一个例子：现在要搜索“中国男排”，而索引字典中包含了“中国女排”、“中国”和“排球”三个词，那么`fuzz.extract`的结果如下：

```
> const indexMap = new Map([
... ['中国女排', [1, 2]],
... ['中国', [1, 2, 3, 4]],
... ['排球', [3, 5, 7]]
... ])
undefined
> fuzz.extract('中国男排', [...indexMap.keys()])
[[ '中国女排', 75, 0 ], [ '中国', 67, 1 ], [ '排球', 33, 2 ] ]
```

在该 indexMap 上运行`generateFuzzyTerms`的结果如下：

```
> generateFuzzyTerms('中国男排').on(indexMap)
[
  [ '中国女排', [ 75, [Array] ] ],
  [ '中国', [ 67, [Array] ] ],
  [ '排球', [ 33, [Array] ] ]
]
> generateFuzzyTerms('中国男排').on(indexMap)[0]
[ '中国女排', [ 75, [ 1, 2 ] ] ]
> generateFuzzyTerms('中国男排').on(indexMap)[1]
[ '中国', [ 67, [ 1, 2, 3, 4 ] ] ]
> generateFuzzyTerms('中国男排').on(indexMap)[2]
[ '排球', [ 33, [ 3, 5, 7 ] ] ]
```

然后，在文档 1 上搜索“中国男排”，就得到如下的结果：

```
> const fuzzyTerms = generateFuzzyTerms('中国男排').on(indexMap)
undefined
> calculateScoreByFuzzy(1, '中国男排').on(fuzzyTerms)
{ sentenceID: 1, scores: { total: 75, '中国男排': 75 } }
```

这个示例应当能比较直观地展示`calculateScoreByFuzzy`的工作方式。

接下来，展示`calculateScoreByCharacter`函数的定义。

```
/**
 * Calculate the score of a sentence with a specific term by the number
 * of characters that match the term.
 * @param sentenceID
 * @param term
 * @returns
 */
const calculateScoreByCharacter = (sentenceID: number, term: string) => {
  ({
    on: (indexMap: IndexMap): FuzzyQueryResult => {
      const characters = term.split('');

      const toScore = (count: number, total: number) => (count / total) *
100;
```

```

const isContained = (character: string) =>
  indexMap.get(character)?.includes(sentenceID) ?? false;

const score = toScore(
  characters.filter(isContained).length,
  characters.length,
);

return {
  sentenceID,
  scores: {
    total: score,
    [term]: score,
  },
};
},
});
});

```

相比上面使用 `fuzzball.js` 实现的按词的模糊匹配，这个按字的模糊匹配算法非常简单：它只是简单地检查搜索词中每个字是否出现在文档中，然后将出现次数除以搜索词字数乘以 100 作为得分。如文档中出现了“中”“国”“女”三个字，搜索“中国女排”的得分就为 75。

显然，该算法的实现过于简单，因此下面包含了一个基于字的更复杂的实现。

4. 为了使基于字的模糊匹配更加精准，加入对字之间距离的考虑。这一新算法如下：

1. 初始化一个匹配分数变量（`matchScore`）为 0。
2. 初始化一个当前处理汉字索引（`currentCharIndex`）为 0。
3. 遍历词语中的每个汉字，对于每个汉字：
 - a. 从索引字典中获取该汉字在目标文档中的位置列表。
 - b. 如果位置列表为空，表示该汉字在文档中不存在，给匹配分数一个较小的惩罚值。
 - c. 如果位置列表不为空，遍历位置列表：
 - i. 对于每个位置，计算该位置与上一个汉字的位置之间的距离。如果这是词语中的第一个汉字，简单将匹配分数加 1，直接对下一个汉字进行处理。
 - ii. 使用距离的倒数作为的分数，取两个位置列表中得到的最大的距离分数，并将其累加到匹配分数上。
 - iii. 更新当前处理汉字索引。

在处理完词语中的所有汉字后，返回匹配分数。

可以看到，该算法需要获取汉字在文档中的位置。尽管在搜索时直接读取文档内容是一个可行的方式，但效率太低，因此这里需要建立一个更复杂的索引字典，以包含每个汉字的位置信息。

为了集中于当前这个算法，先不考虑索引算法的实现。假设在这里我们已经实现了索引算法，可以得到一个“增强版”的索引字典，它的类型定义如下：

```

/**
 * An enhanced version of index map, where the key is still the term,
 * but the value is a map where the key is the sentence ID

```

```

    * and the value is the list of indexes (start by 0) of the term in the
    sentence.
    *
    * @example
    * ```typescript
    * const indexMap: EnhancedIndexMap = new Map([
    *   ['中', {
    *     1: [0, 12], // The term '中' appears twice in the sentence with ID
    *     1, at index 0 and 12.
    *   }],
    * ]);
    * ```
    */
    export type EnhancedIndexMap = Map<string, Record<number, number[]>>;

```

该字典的键仍为汉字，但值不再是一个包含文档 ID 的列表，而是另一个嵌套的字典，其键为文档 ID，值为该汉字在该文档中出现的位置列表。

根据上面描述的算法，在`fuzzy-query-enhanced.ts`中，我们这么实现计算搜索词分数的算法`calculateScore`：

```

/**
 * Calculate the score of a sentence with a specific term by the number
 * of characters that match the term.
 * @param sentenceID
 * @param term
 * @returns
 */
const calculateScore = (sentenceID: number, term: string) => ({
  on: (indexMap: EnhancedIndexMap): FuzzyQueryResult => {
    const PENALTY = 0.5;

    const characters = term.split('');

    let matchScore = 0;
    let lastPositions: number[] = [];

    for (const character of characters) {
      const positionsMap = indexMap.get(character);

      if (!positionsMap || !positionsMap[sentenceID]) {
        // Character not found in the sentence
        // Penalize the score
        if (matchScore > 0) matchScore -= PENALTY;
      } else {
        const positions = positionsMap[sentenceID];
        let maxDistanceScore = 0;

```

```

    for (const position of positions) {
      if (lastPositions.length === 0) maxDistanceScore = 1;
      for (const lastPosition of lastPositions) {
        const distance = Math.abs(position - lastPosition);
        // Use the inverse of the distance as the score
        const distanceScore = 1 / distance;

        if (distanceScore > maxDistanceScore)
          maxDistanceScore = distanceScore;
      }
    }

    matchScore += maxDistanceScore;
    lastPositions = positions;
  }
}

const toScore = (count: number, total: number) => (count / total) *
100;
const score = toScore(matchScore, characters.length);

return {
  sentenceID,
  scores: {
    total: score,
    [term]: score,
  },
};
},
});

```

该实现基本上就是对上面算法描述的代码翻译。在这里，将惩罚值设为 0.5，并使用距离倒数作为分数。这样，就可以使搜索结果倾向于搜索词中每个汉字尽可能集中的结果。对于很大的文档，可能在许多文档中，搜索词中的每个字都出现了，因此简单基于字数的来计算分数显然并不合理。在新算法中，考虑了汉字之间的距离，这将得到更加精准的结果。

除`calculateScore`的实现与`fuzzy-query.ts`中不同外，`fuzzy-query-enhanced.ts`中其他函数的实现基本一致，只是有一些函数因为索引字典的变化而需要稍作调整，这里就不再赘述了。

5. 上面的新模糊匹配需要新的索引算法实现，在这里给出相关的几个函数，它们包含在`indexer-enhanced.ts`中：

```

/**
 * Save the index to a file.
 * @param indexMap The index map.
 * @returns
 *

```

```

* @example
* ```typescript
* await saveIndex(new Map([
*   ['中', { 1: [0, 14], 3: [15, 23] }],
*   ['国', { 2: [1, 7], 3: [6, 9], 4: [15, 18] }],
* ])).toFile('index.txt');
* ```
*/
const saveEnhancedIndex = (indexMap: EnhancedIndexMap) => ({
  toFile: async (indexPath: string) => {
    await fs.writeFile(
      indexPath,
      R.sortBy(R.prop(0), Array.from(indexMap.entries()))
        .map(
          ([word, occurrence]) =>
            `${word}\t${Object.keys(occurrence).length}\t${Object.entri
es(
          occurrence,
        )
        .flatMap(
          ([sentenceID, positions]) =>
            `${sentenceID}:${positions}`,
        )
        .join(' ')}`,
        )
        .join('\n'),
      'utf-8',
    );
  },
});

/**
 * Generate the index from corpus file and save it to another file.
 * @param corpusPathname The path to the corpus file.
 * @param indexPathname The path to the index file.
 * @returns The index map.
 */
export const generateEnhancedIndexByCharacter = async (
  corpusPathname: string,
  indexPathname: string,
): Promise<EnhancedIndexMap> => {
  const lines = (await fs.readFile(corpusPathname, 'utf-8')).split('\n');

  const indexMap = new Map<string, Record<number, number[]>>();

```

```

for (const [index, line] of lines.entries()) {
  const sentenceID = index + 1;
  const characters = line
    .split('')
    // Keep only Chinese words
    .filter((word) => /^[\\u4e00-\\u9fa5]+$/.test(word));

  for (const [position, character] of characters.entries()) {
    if (!indexMap.has(character)) indexMap.set(character, {});
    if (!(indexMap.get(character) as Record<number,
number[]>)[sentenceID])
      (indexMap.get(character) as Record<number, number[]>)[sentenceID]
= [];
    (indexMap.get(character) as Record<number,
number[]>)[sentenceID].push(
      position,
    );
  }
}

// Write to file
await saveEnhancedIndex(indexMap).ToFile(indexPathname);

return indexMap;
};

/**
 * Read the index from a file.
 * @param pathname The path to the index file.
 * @returns The index map.
 */
export const readEnhancedIndex = async (
  pathname: string,
): Promise<EnhancedIndexMap> =>
  (await fs.readFile(pathname, 'utf-8'))
    .split('\\n')
    .map((line) => line.split('\\t'))
    .map(
      ([word, , sentenceIDAndPositions]) =>
        [
          word,
          Object.fromEntries(
            sentenceIDAndPositions.split('
').map((sentenceIDAndPosition) => {

```

```

        const [sentenceID, positions] =
sentenceIDAndPosition.split(
            ':',
            2,
        );
        return [sentenceID, positions.split(',').map(Number)];
    }},
    ),
] as const,
)
.reduce((acc, [word, sentenceIDAndPositions]) => {
    acc.set(word, sentenceIDAndPositions);
    return acc;
}, new Map());

```

由于索引算法并非本次作业的重点，这里就不再详述这几个函数的原理了。并且，它们的原理也比较简单，只是看上去繁琐而已。

6. 最后在`main.ts`中编写主函数。该函数首先判断三个索引文件（使用 jieba 分词的索引文件、基于字的索引文件、增强版的基于字的索引文件）是否存在，如不存在，先构建索引并保存；若存在，则直接读取索引文件。

在读取索引文件完成后，首先提示用户选择模糊搜索算法。选择结束后，循环等待用户输入布尔查询字符串，然后在索引字典中查询对应的 docID。

```

import { enhancedFuzzyQueryByCharacter } from
'./fuzzy-query-enhanced.js';
import { fuzzyQuery } from './fuzzy-query.js';
import {
    generateEnhancedIndexByCharacter,
    readEnhancedIndex,
} from './indexer-enhanced.js';
import { IndexStrategy, generateIndex, readIndex } from './indexer.js';
import { fileExists } from './utils/file.js';
import { Nested, logged } from './utils/log.js';

const CORPUS_PATHNAME = './data/corpus.txt';
const TARGET_PATHNAME = './data/corpusnew.txt';
const WORD_INDEX_PATHNAME = './dict.index';
const CHARACTER_INDEX_PATHNAME = './dict.index.char';
const CHARACTER_ENHANCED_INDEX_PATHNAME = './dict.index.char.enhanced';

const loggedReadIndex = logged('Index read', readIndex);
const loggedGenerateIndex = logged('Index generated', generateIndex);
const loggedFuzzyQuery = logged(
    ({ __callStack: [[, [input]]] }) => `Queried "${input}"`,
    fuzzyQuery,

```



```

    Nested.of(1),
  );

const loggedReadEnhancedIndex = logged(
  'Enhanced character index read',
  readEnhancedIndex,
);

const loggedGenerateEnhancedIndexByCharacter = logged(
  'Enhanced character index generated',
  generateEnhancedIndexByCharacter,
);

const loggedEnhancedFuzzyQueryByCharacter = logged(
  ({ __callStack: [[, [input]]] }) => `Queried "${input}"`,
  enhancedFuzzyQueryByCharacter,
  Nested.of(1),
);

const main = async () => {
  const wordIndexMap = (await fileExists(WORD_INDEX_PATHNAME))
    ? await loggedReadIndex(WORD_INDEX_PATHNAME)
    : await loggedGenerateIndex(
        CORPUS_PATHNAME,
        TARGET_PATHNAME,
        WORD_INDEX_PATHNAME,
      );

  const characterIndexMap = (await fileExists(CharacterIndex.PATHNAME))
    ? await loggedReadIndex(CharacterIndex.PATHNAME)
    : await loggedGenerateIndex(
        CORPUS_PATHNAME,
        TARGET_PATHNAME,
        CharacterIndex.PATHNAME,
        IndexStrategy.BY_CHARACTER,
      );

  const characterEnhancedIndexMap = (await fileExists(
    CharacterEnhancedIndex.PATHNAME,
  ))
    ? await loggedReadEnhancedIndex(CharacterEnhancedIndex.PATHNAME)
    : await loggedGenerateEnhancedIndexByCharacter(
        CORPUS_PATHNAME,
        CharacterEnhancedIndex.PATHNAME,
      );

  process.stdout.write(

```

```

    'Select a mode (0 for term, 1 for character, 2 for character-enhanced):
    ',
  );
  let mode: 'term' | 'character' | 'character-enhanced';
  process.stdin.on('data', (data) => {
    if (mode === undefined) {
      if (data.toString().trim() === '0') mode = 'term';
      else if (data.toString().trim() === '1') mode = 'character';
      else if (data.toString().trim() === '2') mode =
'character-enhanced';
      else throw new Error('Invalid mode');

      console.log();
      process.stdout.write('Query: ');
      return;
    }

    if (mode === 'term')
      console.log(
        loggedFuzzyQuery(data.toString().trim(), {
          threshold: 0,
          limit: 5,
          mode,
        }).on(wordIndexMap),
      );
    else if (mode === 'character')
      console.log(
        loggedFuzzyQuery(data.toString().trim(), {
          threshold: 0,
          limit: 5,
          mode,
        }).on(characterIndexMap),
      );
    else
      console.log(
        loggedEnhancedFuzzyQueryByCharacter(data.toString().trim(), {
          threshold: 0,
          limit: 5,
        }).on(characterEnhancedIndexMap),
      );

    console.log();

    process.stdout.write('Query: ');
  });

```

```
};  
  
await main();
```

（二）实验结果

运行`npm run dev`，使用 ts-node 执行`./src/main.ts`。

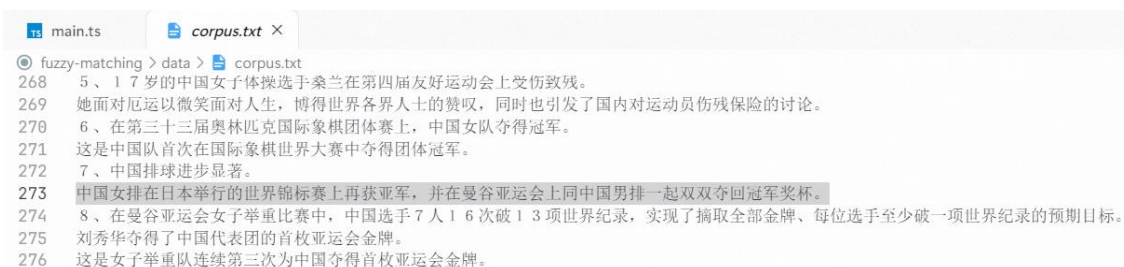
首先测试第一种搜索算法，即使用 **fuzzball.js** 进行模糊匹配的效果。

首先搜索“中国女排”：

```
Select a mode (0 for term, 1 for character, 2 for character-enhanced): 0  
  
Query: 中国女排  
Queried "中国女排" in 16ms.  
[  
  { sentenceID: 273, scores: { total: 100, '中国女排': 100 } },  
  { sentenceID: 270, scores: { total: 75, '中国女排': 75 } },  
  { sentenceID: 1, scores: { total: 67, '中国女排': 67 } },  
  { sentenceID: 3, scores: { total: 67, '中国女排': 67 } },  
  { sentenceID: 4, scores: { total: 67, '中国女排': 67 } }  
]
```

可以看到，由于 jieba 分词中出现了“中国女排”这个词，而第 273 句包含了这个词，因此其匹配分数最高。这里列出了前 5 个结果。

观察 corpus.txt，可以看到第 273 句确实完整包含了“中国女排”：



```
main.ts corpus.txt X  
fuzzy-matching > data > corpus.txt  
268 5、17 岁的中国女子体操选手桑兰在第四届友好运动会上受伤致残。  
269 她面对厄运以微笑面对人生，博得世界各界人士的赞叹，同时也引发了国内对运动员伤残保险的讨论。  
270 6、在第三十三届奥林匹克国际象棋团体赛上，中国女队夺得冠军。  
271 这是中国队首次在国际象棋世界大赛中夺得团体冠军。  
272 7、中国排球进步显著。  
273 中国女排在日本举行的世界锦标赛上再获亚军，并在曼谷亚运会上同中国男排一起双双夺回冠军奖杯。  
274 8、在曼谷亚运会女子举重比赛中，中国选手 7 人 16 次破 13 项世界纪录，实现了摘取全部金牌、每位选手至少破一项世界纪录的预期目标。  
275 刘秀华夺得了中国代表团的首枚亚运会金牌。  
276 这是女子举重队连续第三次为中国夺得首枚亚运会金牌。
```

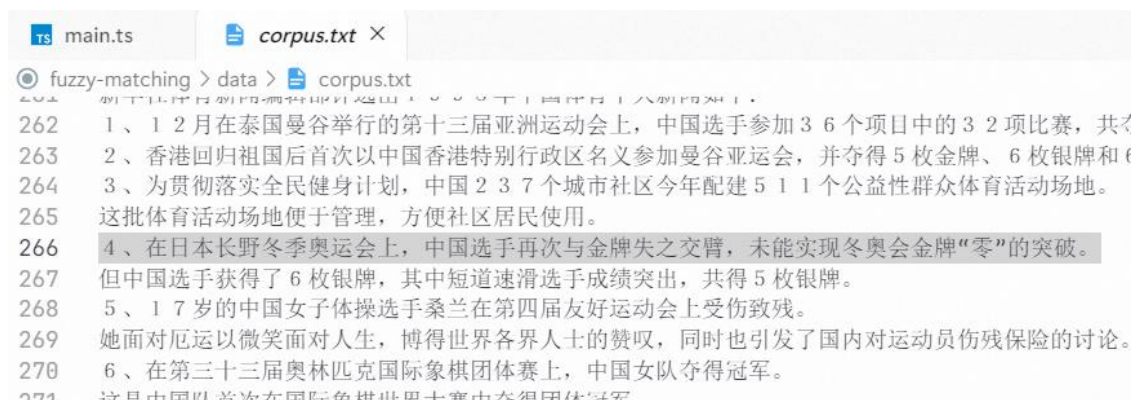
然而，对于“中国男排”，由于 jieba 分词算法的实现关系，词典中并未包含这个词语，因此尽管第 273 句也完整包含了“中国男排”，但匹配分数不是 100：

```
Query: 中国男排  
Queried "中国男排" in 45ms.  
[  
  { sentenceID: 273, scores: { total: 75, '中国男排': 75 } },  
  { sentenceID: 1, scores: { total: 67, '中国男排': 67 } },  
  { sentenceID: 3, scores: { total: 67, '中国男排': 67 } },  
  { sentenceID: 4, scores: { total: 67, '中国男排': 67 } },  
  { sentenceID: 5, scores: { total: 67, '中国男排': 67 } }  
]
```

然后，测试布尔查询“中国男排 and 奥运会”，得到下面的结果：

```
Query: 中国男排 and 奥运会
Queried "中国男排 and 奥运会" in 45ms.
[
  { sentenceID: 266, scores: { total: 83.5, '中国男排': 67, '奥运会': 100 } },
  { sentenceID: 273, scores: { total: 71, '中国男排': 75, '奥运会': 67 } },
  { sentenceID: 262, scores: { total: 67, '中国男排': 67, '奥运会': 67 } },
  { sentenceID: 263, scores: { total: 67, '中国男排': 67, '奥运会': 67 } },
  { sentenceID: 268, scores: { total: 67, '中国男排': 67, '奥运会': 67 } }
]
```

这次，由于考虑了“奥运会”这个词，匹配度最高的不再是第 273 句，而是第 266 句：



下面尝试一个比较复杂的布尔查询表达式“中国男排 or (奥运会 and not 人民)”：

```
Query: 中国男排 or (奥运会 and not 人民)
Queried "中国男排 or (奥运会 and not 人民)" in 52ms.
[
  {
    sentenceID: 266,
    scores: { total: 100, '中国男排': 67, '奥运会': 100, '人民': 0 }
  },
  {
    sentenceID: 262,
    scores: { total: 83.5, '中国男排': 67, '奥运会': 67, '人民': 0 }
  },
  {
    sentenceID: 263,

```

可以看到，排在前面的几个结果都不包含“人民”，这符合我们的预期。

然后测试第二种算法，即简单按字匹配的效果。

同样，先测试“中国女排”。可以看到效果符合预期，也是第 273 句排在最前：

```
Select a mode (0 for term, 1 for character, 2 for character-enhanced): 1
```

```
Query: 中国女排
```

```
Queried "中国女排" in 10ms.
```

```
[
  { sentenceID: 273, scores: { total: 100, '中国女排': 100 } },
  { sentenceID: 268, scores: { total: 75, '中国女排': 75 } },
  { sentenceID: 270, scores: { total: 75, '中国女排': 75 } },
  { sentenceID: 272, scores: { total: 75, '中国女排': 75 } },
  { sentenceID: 274, scores: { total: 75, '中国女排': 75 } }
]
```

然后搜索“中国男排”，可以看到第 273 句仍排在最前。并且，与上一种搜索算法不同，其匹配得分是 100，不受分词算法的影响。

```
Query: 中国男排
```

```
Queried "中国男排" in 7ms.
```

```
[
  { sentenceID: 273, scores: { total: 100, '中国男排': 100 } },
  { sentenceID: 272, scores: { total: 75, '中国男排': 75 } },
  { sentenceID: 277, scores: { total: 75, '中国男排': 75 } },
  { sentenceID: 278, scores: { total: 75, '中国男排': 75 } },
  { sentenceID: 1, scores: { total: 50, '中国男排': 50 } }
]
```

然后搜索“中国男排 and 奥运会”，可以看到熟悉的第 273 句与第 266 句同样排在前面：

```
Query: 中国男排 and 奥运会
```

```
Queried "中国男排 and 奥运会" in 17ms.
```

```
[
  {
    sentenceID: 273,
    scores: { total: 83.33333333333333, '中国男排': 100, '奥运会': 66.66666666666666 }
  },
  { sentenceID: 266, scores: { total: 75, '中国男排': 50, '奥运会': 100 } },
  {
    sentenceID: 278,
    scores: { total: 70.83333333333333, '中国男排': 75, '奥运会': 66.66666666666666 }
  },
  {
    sentenceID: 197,
    scores: { total: 58.33333333333333, '中国男排': 50, '奥运会': 66.66666666666666 }
  },
  {
    sentenceID: 262,
    scores: { total: 58.33333333333333, '中国男排': 50, '奥运会': 66.66666666666666 }
  }
]
```

但是，如果观察其他结果，如第 278 句，就会发现其中一些结果看起来与搜索词好像没有很大关联，但仍旧出现在了靠前的搜索结果中：

```
main.ts corpus.txt X
fuzzy-matching > data > corpus.txt
272 7、中国排球进步显著。
273 中国女排在日本举行的世界锦标赛上再获亚军，并在曼谷亚运会上同中国男排一起双双夺回冠军奖杯。
274 8、在曼谷亚运会女子举重比赛中，中国选手7人16次破13项世界纪录，实现了摘取全部金牌、每位选手：
275 刘秀华夺得了中国代表团的首枚亚运会金牌。
276 这是女子举重队连续第三次为中国夺得首枚亚运会金牌。
277 9、中国男子乒乓球受到严峻挑战。
278 在汕头举行的男子世界杯赛上，3名中国男单选手无一进入决赛；在亚运会乒乓球赛中，丢掉了男单冠军。
279 10、9月在南京举行的全国场地自行车系列赛总决赛对参赛选手进行了血样检测。
280 这是中国第一次在全国性大赛中实行血检，标志着中国进一步加大了反兴奋剂斗争的力度。
```

这意味着，这种基于简单按字统计的算法存在缺陷。可以想象，若文档很大，对于某个搜索词，可能有很多文档中都能找到该词中的每个汉字，但是这不意味着这些文档的匹配度都很高。

然后，测试改进版的、引入了距离算法及惩罚值的按字匹配算法，观察它是否较前一种简单按字匹配算法更加精确：

首先，仍旧测试“中国女排”。

Select a mode (0 for term, 1 for character, 2 for character-enhanced): 2

Query: 中国女排

Queried "中国女排" in 16ms.

```
[
  { sentenceID: 273, scores: { total: 100, '中国女排': 100 } },
  { sentenceID: 268, scores: { total: 62.5, '中国女排': 62.5 } },
  { sentenceID: 270, scores: { total: 62.5, '中国女排': 62.5 } },
  { sentenceID: 272, scores: { total: 62.5, '中国女排': 62.5 } },
  { sentenceID: 274, scores: { total: 40.625, '中国女排': 40.625 } }
]
```

可以看到，第273句仍以100的得分排在首位。此外，观察到由于引入了惩罚值，其他结果的得分均有所降低。在这里观察一下第268句的内容：

```
main.ts corpus.txt X
fuzzy-matching > data > corpus.txt
262 1、12月在泰国曼谷举行的第十三届亚洲运动会上，中国选手参加36个项目中的32项比赛，共夺
263 2、香港回归祖国后首次以中国香港特别行政区名义参加曼谷亚运会，并夺得5枚金牌、6枚银牌和6
264 3、为贯彻落实全民健身计划，中国237个城市社区今年配建511个公益性群众体育活动场地。
265 这批体育活动场地便于管理，方便社区居民使用。
266 4、在日本长野冬季奥运会上，中国选手再次与金牌失之交臂，未能实现冬奥会金牌“零”的突破。
267 但中国选手获得了6枚银牌，其中短道速滑选手成绩突出，共得5枚银牌。
268 5、17岁的中国女子体操选手桑兰在第四届友好运动会上受伤致残。
269 她面对厄运以微笑面对人生，博得社会各界人士的赞叹，同时也引发了国内对运动员伤残保险的讨论。
```

可以看到，这句话确实与“中国女排”不怎么相关，因此得到这样的分数是比较合理的。

然后，测试“中国男排”的结果：


```
Query: 中国男排
Queried "中国男排" in 32ms.
[
  { sentenceID: 273, scores: { total: 100, '中国男排': 100 } },
  { sentenceID: 272, scores: { total: 62.5, '中国男排': 62.5 } },
  { sentenceID: 277, scores: { total: 62.5, '中国男排': 62.5 } },
  { sentenceID: 278, scores: { total: 62.5, '中国男排': 62.5 } },
  { sentenceID: 1, scores: { total: 25, '中国男排': 25 } }
]
```

同样的，由于第 273 句完整包含了这个词语，因此得分为 100。而剩余的搜索结果，由于惩罚值的存在，得分都不很高。

然后搜索“中国男排 and 奥运会”，观察搜索结果：

```
Query: 中国男排 and 奥运会
Queried "中国男排 and 奥运会" in 28ms.
[
  {
    sentenceID: 273,
    scores: { total: 83.33333333333333, '中国男排': 100, '奥运会': 66.66666666666666 }
  },
  {
    sentenceID: 278,
    scores: {
      total: 64.58333333333333,
      '中国男排': 62.5,
      '奥运会': 66.66666666666666
    }
  },
  { sentenceID: 266, scores: { total: 62.5, '中国男排': 25, '奥运会': 100 } },
  {
    sentenceID: 262,
    scores: { total: 45.83333333333333, '中国男排': 25, '奥运会': 66.66666666666666 }
  },
]
```

可以看到第 273 句仍旧排在最前面。尽管它没有包含“奥运会”，但是它包含了“亚运会”这个词语，并且完整包含了“中国男排”。与第一种算法得到的第 266 句排在最前的结果，这一结果似乎更为合理。

```
main.ts corpus.txt X
fuzzy-matching > data > corpus.txt
272 7、中国排球进步显著。
273 中国女排在日本举行的世界锦标赛上再获亚军，并在曼谷亚运会上同中国男排一起双双夺回冠军奖杯。
274 8、在曼谷亚运会女子举重比赛中，中国选手7人16次破13项世界纪录，实现了摘取全部金牌、每位选手至
275 刘秀华夺得了中国代表团的首枚亚运会金牌。
276 这是女子举重队连续第三次为中国夺得首枚亚运会金牌。
277 9、中国男子乒乓球受到严峻挑战。
278 在汕头举行的男子世界杯赛上，3名中国男单选手无一进入决赛；在亚运会乒乓球赛中，丢掉了男单冠军。
279 10、9月在南京举行的全国场地自行车系列赛总决赛对参赛选手进行了血检检测。
280 这是中国第一次在全国性大赛中实行血检，标志着中国进一步加大了反兴奋剂斗争的力度。
```

同样的，第 278 句作为匹配度第二高的搜索结果，包含了“中国男单”和“亚运会”，排在第二看起来也是比较合理的。而第 266 句则排在了第三。

因此可以总结出，与前面两种算法相比较，第三种引入了距离与惩罚值的按字模糊匹配得到的结果更加合理。并且，通过终端打印的日志也可以发现，该算法的效率介于第一种使用 fuzzball.js 的匹配和第二种简单按字匹配之间，其时间效率也是比较合理的。

四. 实验总结

1. 本次实验使用了 Node.js 上的模糊匹配库 fuzzball.js 及 jieba 分词实现了第一种模糊匹配算法。与本实验实现的另外两种算法相比,该算法的一大缺点是依赖分词算法的实现,若分词算法实现不佳,很难获得精确的结果。并且,使用已有的模糊匹配库效率不高,对于大文档可能造成一定的性能问题。但是如果分词算法实现合理,这种算法是比较稳定与可靠的。

2. 第二种模糊匹配算法实现,即基于字的简单统计算法速度很快,并且不依赖分词算法实现,只需要使用按字的倒排索引。并且相比于第一种算法,建立的索引文件也比较小,在本次实验所给的数据中,使用 jieba 分词的索引文件有 4895 行,而按字的索引文件只有 1821 行。但是,这种方法得到的结果不够精确,特别是对于较大的文档,可能对于某个搜索词,有很多文档都包含了该搜索词中的每一个字,但是显然不是这所有的文档都具有同样的匹配度。

3. 在第二种算法的基础上,通过建立一个更复杂的、包含了每个字在文档中位置的索引,引入惩罚值及汉字距离,实现了第三种算法。当搜索词中的某个汉字没有在索引字典中找到时,在最终得分上减去某个惩罚值,并按照汉字距离的倒数作为得分,以使搜索词中汉字聚集得尽可能密集的文档得到更高的分数。该算法的效率介于第一种和第二种算法之间,速度较快,并且具有良好的准确性,甚至在许多情况下表现得比第一种算法更好。

不过,这种算法也有需要建立一个较大索引字典的缺点,因其需要保存每一个字在每一个文档中出现的位置。在本次实验中,该算法索引字典的大小超过了前两种算法近一倍。可以想象,若文档字数很多,其索引字典的大小会变得相当大,造成较大的存储空间占用,并且也会一定程度上影响算法效率。

4. 本实验延续了上一次实验为布尔查询表达式构建的语法解析器,并且实现了查询结果的合并。具体来说,对于 AND 操作符,将左右得分平均;对于 OR 操作符,取左右得分中较大的一个;对于 NOT 操作符,使用 100 减去得分(得分控制在 0-100)。

5. 通过本次实验,观察到相比精确搜索,模糊搜索可以有效改善以下情况中精确搜索造成的漏检:

- 拼写错误
- 同近义词
- 变体
- 部分匹配
- 顺序不敏感

即使分词结果非常完美,模糊搜索也仍有很大的存在价值。较好的分词算法在上面仅部分解决了部分匹配情况的一些问题,而对于拼写错误、顺序不敏感情况等则效果不佳。

5. 通过本次实验,了解并实践了模糊匹配算法的实现与原理,为后面的实验做了铺垫。