搜索引擎 demo 版：Prototype Phoenix 设计

综述

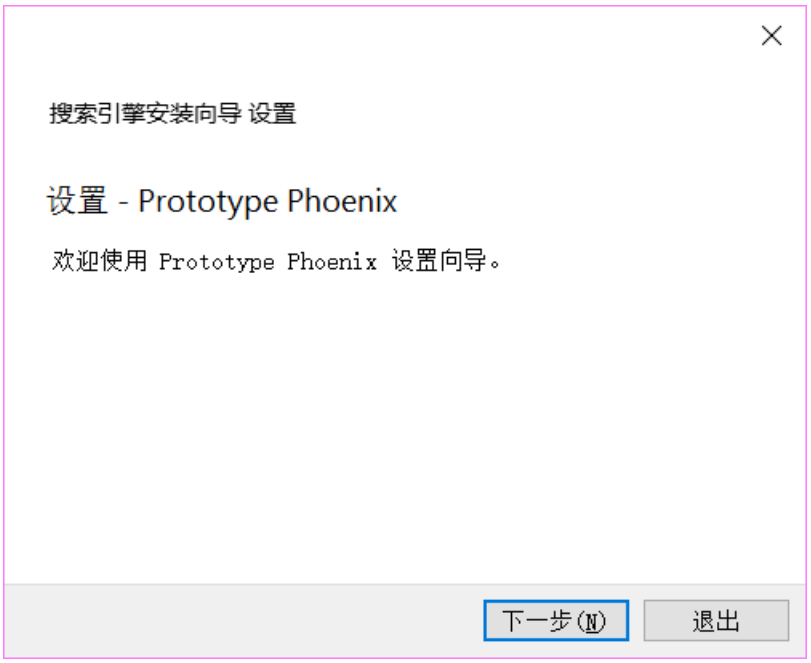
本搜索引擎 demo 版名为 Prototype Phoenix，实现了在线爬取网页、本地载入网页、对网页源代码进行处理、分词、建立倒排索引、根据 PageRank 进行排序、自动识别布尔检索等功能，界面友好，功能完善，运行速度快，鲁棒性强。

设计者：张帅，杨一江，乔裕哲

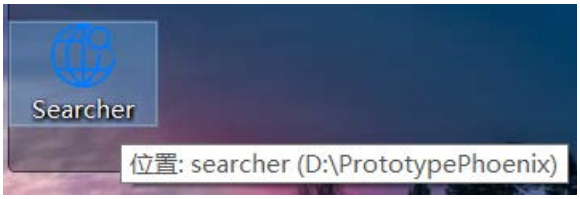
使用概述

我们实现了程序的打包，制作了安装包，便于程序的发布和安装；

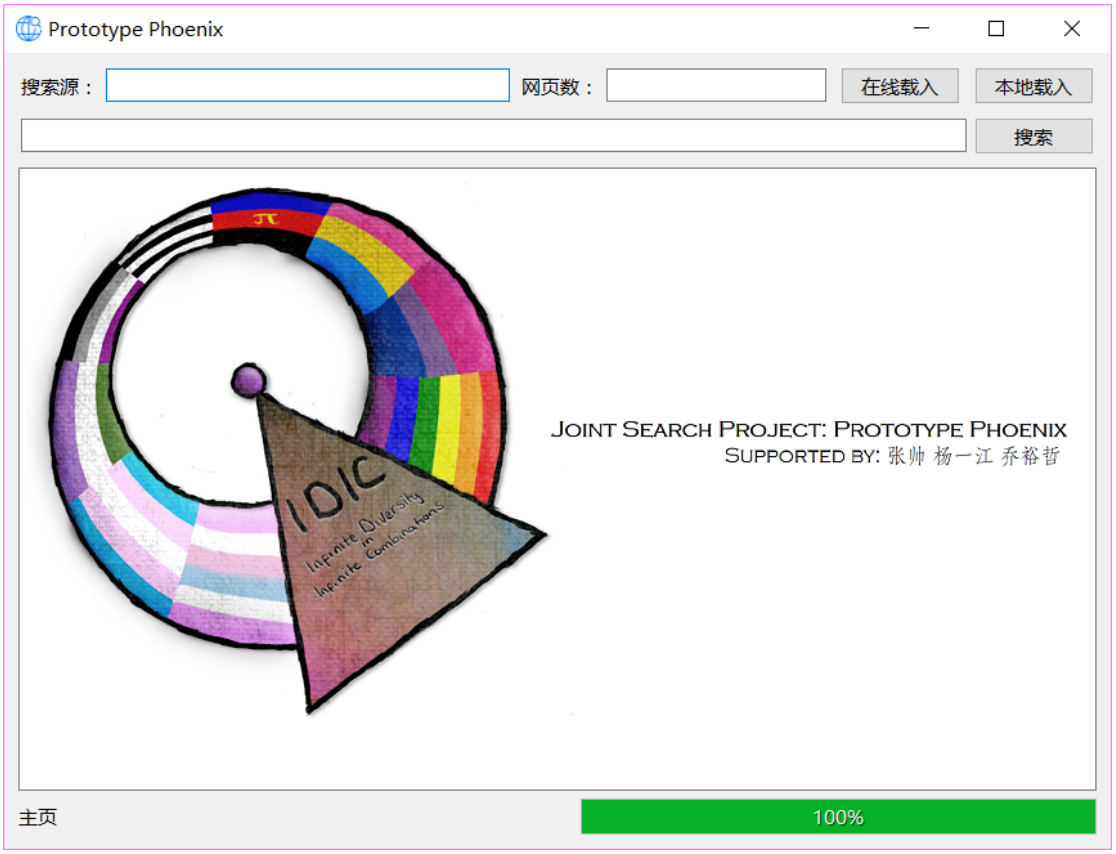
双击开始安装



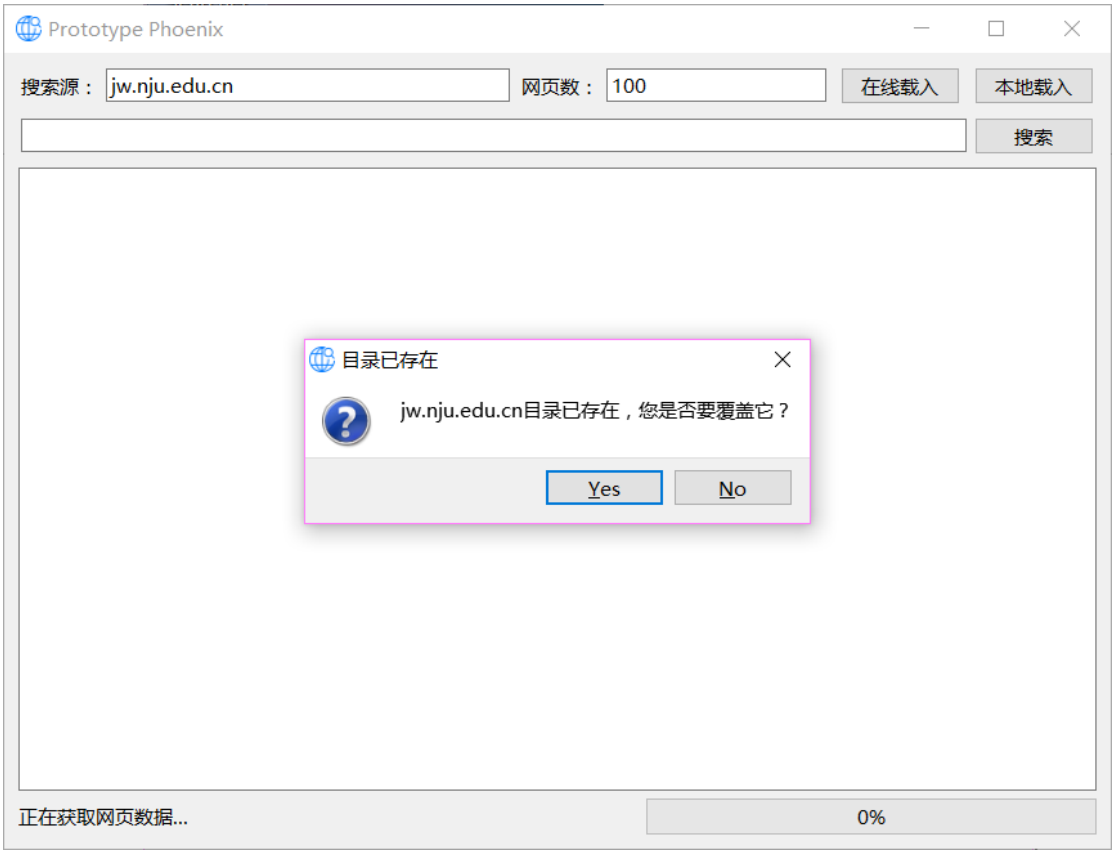
安装之后会自动在开始菜单和桌面创建快捷方式



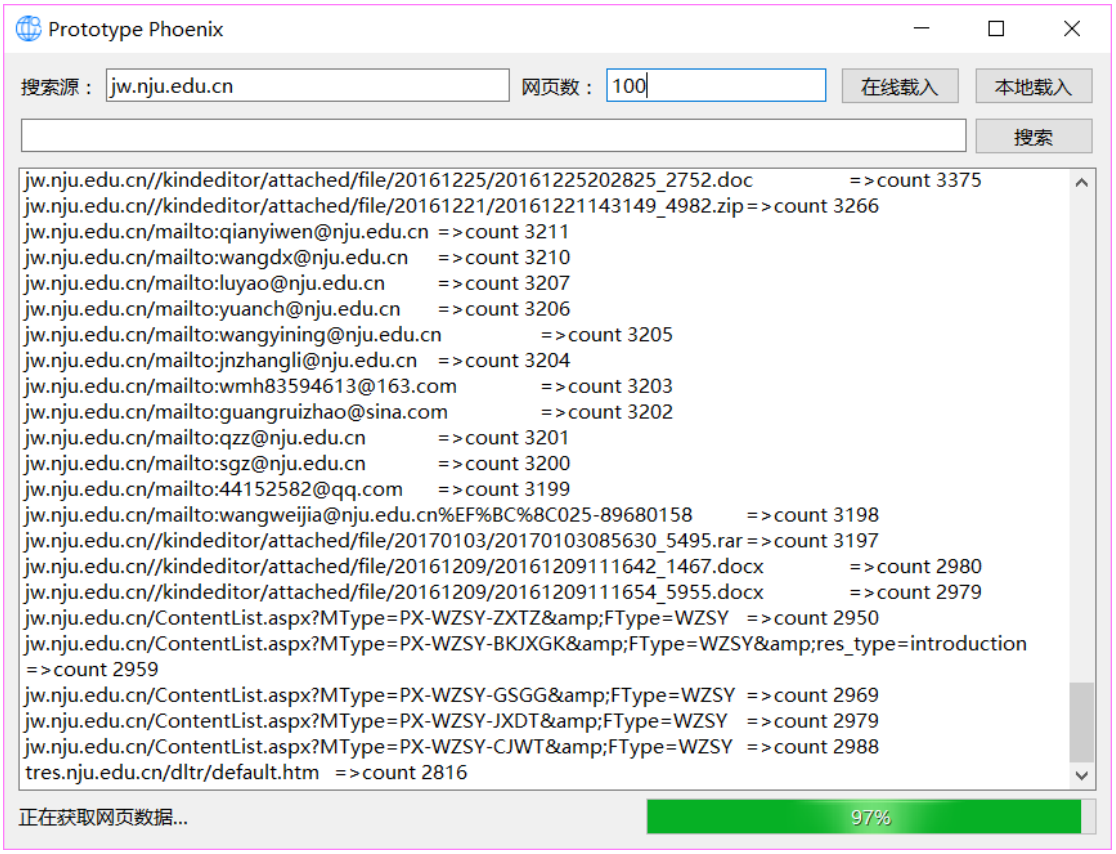
双击启动后，显示开始界面。



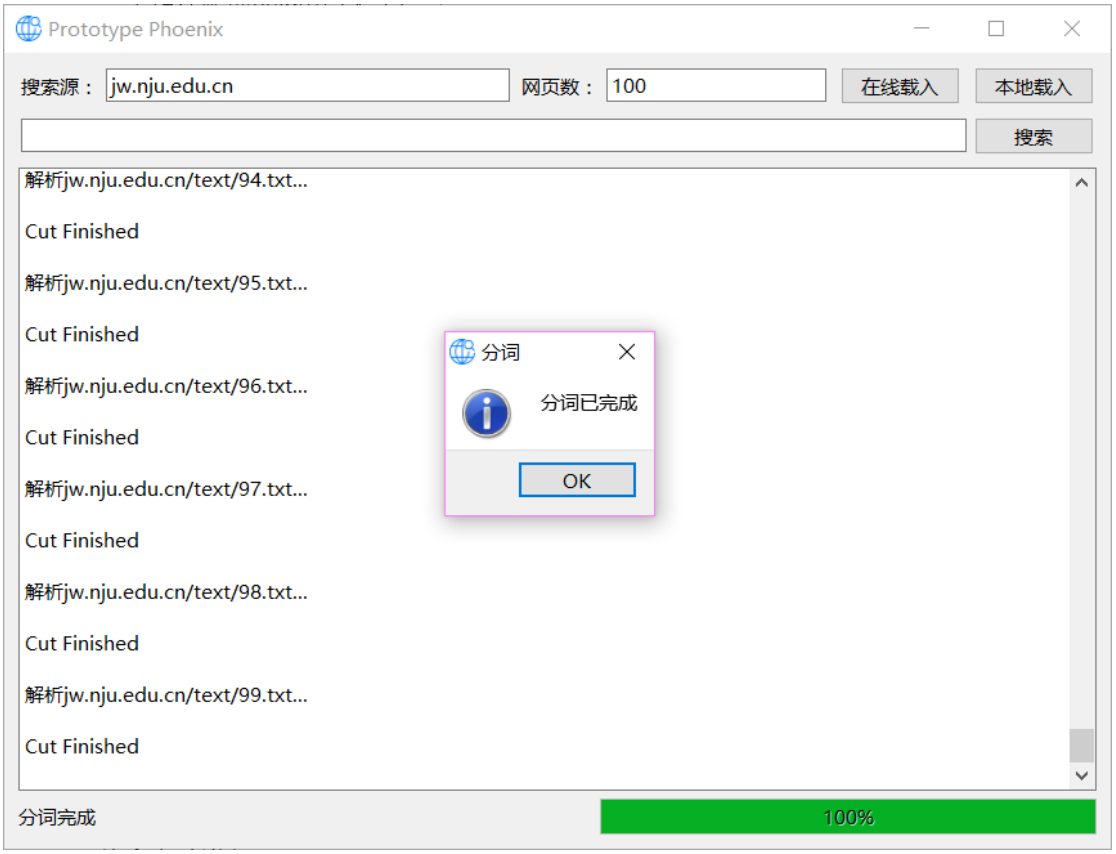
用户可以在“搜索源”中填写爬取网页的起始地址，“网页数”中填写搜索的网页数，直接按 ENTER 或点击“在线载入”便可自动开始在线载入，若目录已存在则会弹出提示信息

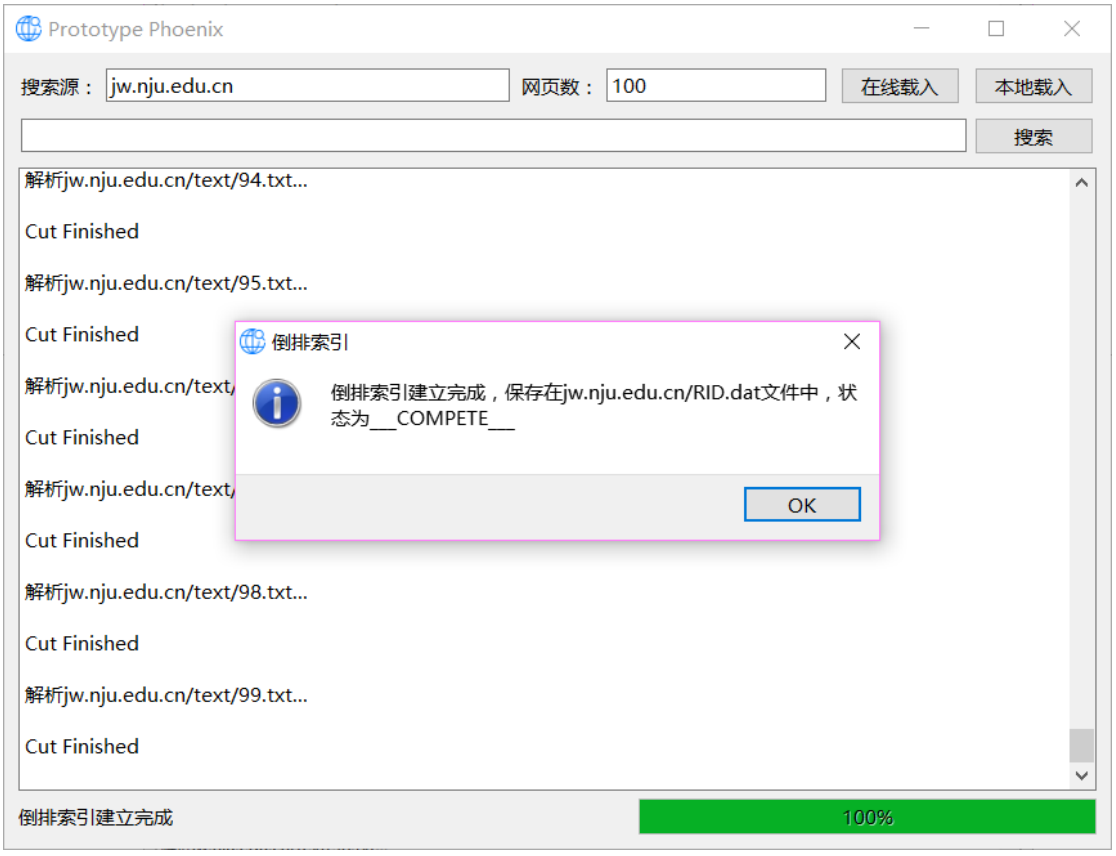


选择“Yes”，爬虫便可以开始工作，右下角进度条显示爬取进度，浏览器区域显示正在爬取的网页等信息。



等待须臾，便可看到已爬取完成，后续对网页的处理（包括分词和倒排索引）将会自动进行。





然后便可以开始搜索了。输入"伯罗奔尼撒战争 毛泽东思想"并按 ENTER 或

点击"搜索"，搜索引擎将自动识别出这是普通搜索，结果中将把同时出现这两词的网页排在最前面，只出现一词或只出现了一个词的一部分的网页将放在后面。

可以看到搜索结果中，含有关键词的网页部分被提取出来，并且用红色将关键词作了标记，并且标题为超链接，点击标题可自动调用系统浏览器并打开网页。

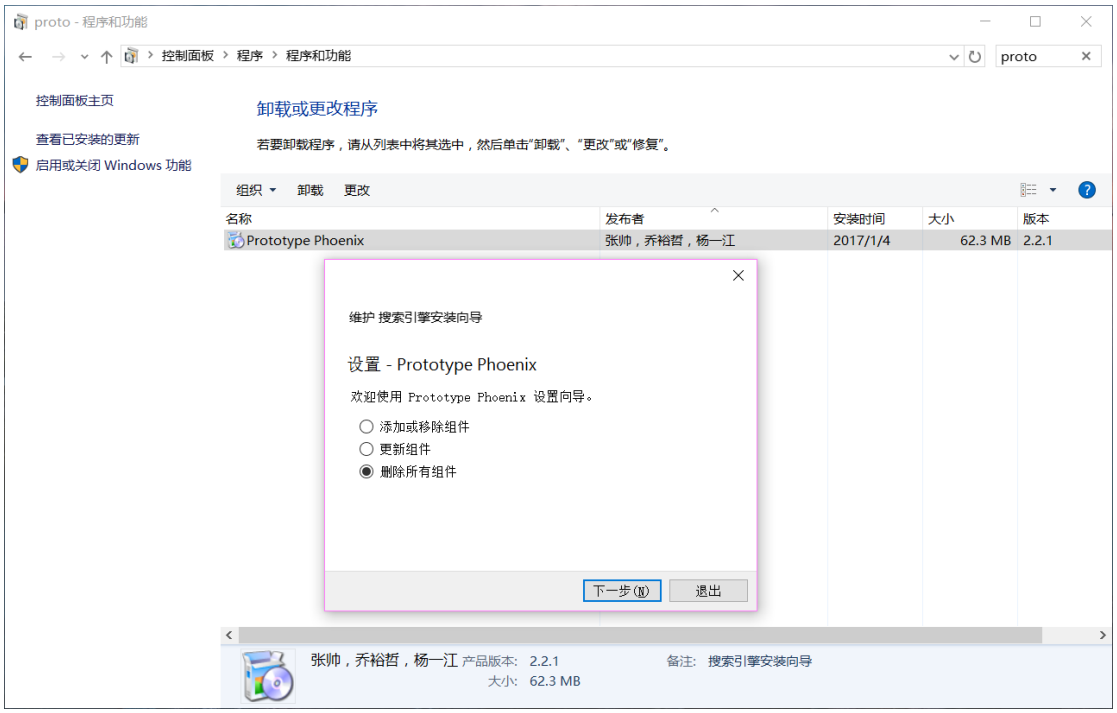
若想进行布尔检索，只需输入布尔检索所需的关键词即可，如'*'代表 AND，'+'代表 OR，'-'代表 NOT，这里以"教务-信箱"为例。

可以看到搜索结果中，既含"教务"又含"信箱"的网页已经从含"教务"的集合中被去掉。

另外，若本起始地址已经加载完成，则可以选择"本地载入"，载入完成之后可以进行同样的搜索。

本搜索引擎鲁棒性良好。1、在用户输入非法时均会给出提示；2、在运行爬虫或倒排索引这些相对较慢的服务时使用了子线程，这样主线程不会因为等待时间过长而"未响应"；3、使用了垂直、水平布局器，窗口各组件会随着窗口大小的变化而变化，简洁美观。

使用完成之后便可以在"控制面板>添加/删除程序"进行卸载。



这个搜索引擎是我们三个人从 12 月 4 日开始到 1 月 4 日，历经整整一个月的时间，呕心沥血之作。其中在很多细节方面做了大量的优化，花费了大量的精力，源文件总数达 28 项，不计安装包生成脚本和界面设计，总代码达 2800 余行。前后发布版本从 V1.0.0 直至 V2.2.1 最终版，发布最终版时已是 2017 年 1 月 4 日中午 14:00，激动心情难以言表，我们也希望助教、老师也能让我们为自己的不懈努力而感到骄傲。

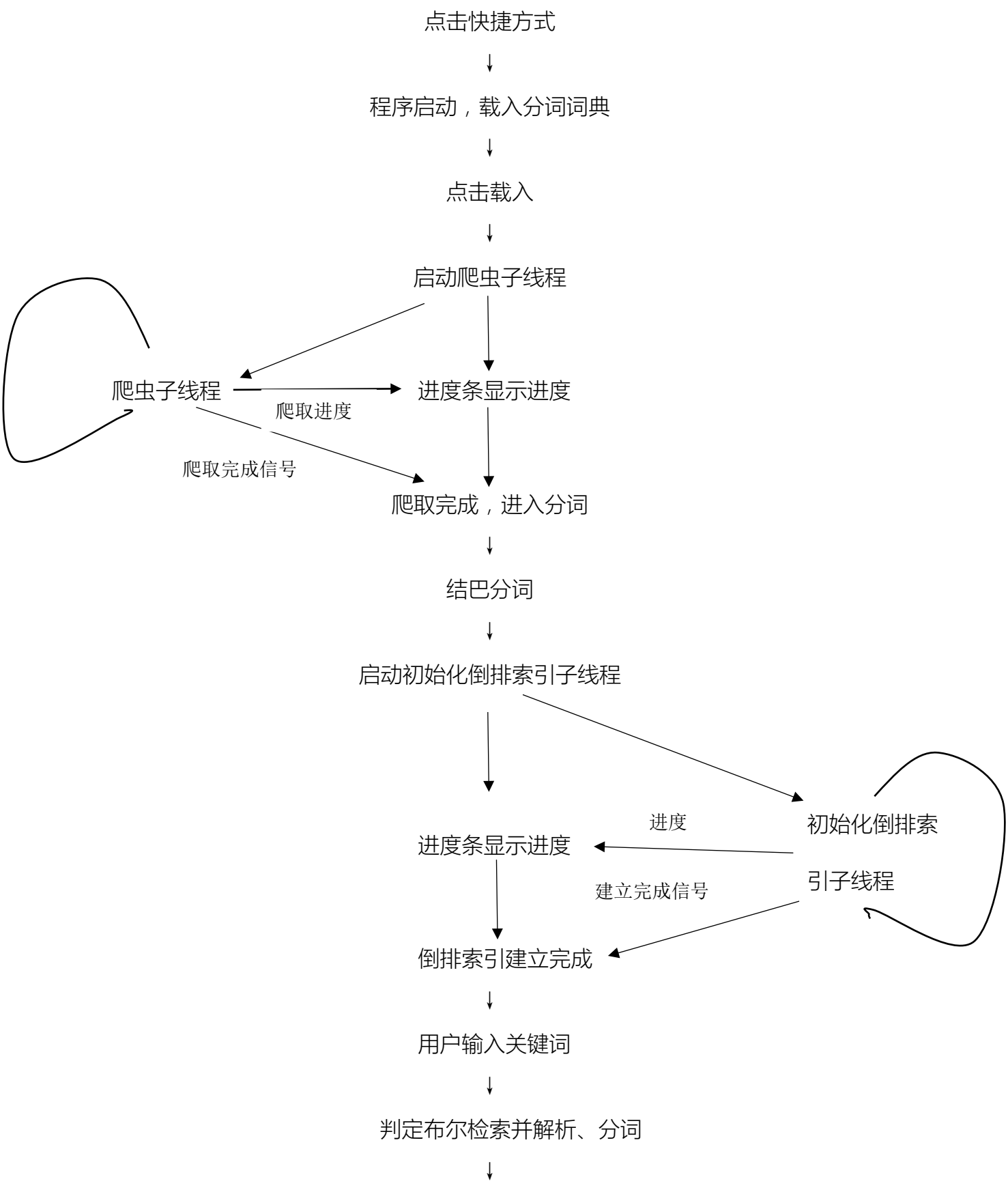
设计思路

总体框架、分词及界面，设计者：张帅

总体框架

总体框架为：以主界面为主体，较慢的爬虫和初始化倒排索引为子线程，较快的分词和搜索直接在主线程中调用，使用信号-槽机制实现界面、爬虫、建立

倒排索引、搜索之间的通信。爬虫为自写，分词使用结巴分词。



交付倒排索引搜索

↓

将结果网页文本转为 HTML 格式

↓

显示在浏览器上

↓

搜索完成

多线程机制

考虑到爬虫在爬取页面的过程中会出现等待时间较长的情况、倒排索引写文件较慢的情况，使用了多线程机制。

界面为主线程，爬取页面和初始化倒排索引为子线程。当要使用爬虫或初始化倒排索引时，主线程会调用子线程的 start()函数，让子线程开始运行，而主线程仍可保持接收信号的状态。子线程与主线程通过信号-槽机制进行通信，子线程任务进度将会以信号的形式发给主线程，从而在主线程的进度条上显示出来；当子线程运行结束时，将会给主线程发送任务完成的信号。

子线程分别名为 CrawlerThread（爬虫子线程）和 InitRidThread（初始化倒排索引子线程），其类定义函数分别在 crawlerthread.h 和 initridthread.h 中，实现函数分别在 crawlerthread.cpp 和 initridthread.cpp 中。为了方便分工且让工程代码有更好的风格，所以爬虫和初始化倒排索引的代码均与这两个子线程对应，而不直接与主线程进行通信。

同时，为了方便代码阅读，将两个子线程的槽函数分别放在为 widget_crawlslots.cpp 和 widget_initridslots.cpp 两个文件中。

使用的类成员：

CrawlerThread *crawlThread;//爬虫子线程

InitRidThread *initRidThread;//初始化倒排索引子线程

主线程槽函数：

void slotCrawlQuestion(const QString &title, const QString &content, int &ret);//子线程需要阻塞式弹框信号，实现主线程与子线程同步<来自爬虫>

void slotCrawlWarning(const QString &title, const QString &content);//子线程阻塞式警告信号<来自爬虫>

void slotShowProgress(int p);//子线程进度信号

void slotCrawlComplete(int pageCount);//子线程完成信号<来自爬虫>

void slotCrawlBrowserAppend(const QString &str);//浏览器内容添加信号<来自爬虫>

void slotInitRidComplete(const QString &filename, int state);//初始化倒排索引完成信号<来自初始化倒排索引>

界面设计

界面采用 QT 库，使用了 lineEdit、pushButton、progressBar、label、textBroswer 等基础界面工具，后期为了布局美观，加入了 Vertical Layout、Horizontal Layout、Vertical Spacer、Horizontal Spacer 等布局工具。

lineEdit：用于输入搜索源、最大网页数、搜索目标等；

pushButton：用于发送信号执行任务，如开始载入、搜索等；

label：提供提示信息；

textBroswer：显示爬取进度、分词进度和搜索结果，其中使用了 HTML 语言

对关键词进行了标记，对结果标题嵌入超链接，使得点击超链接可以调用系统浏览器访问网站，关键词以红色出现。

Vertical Layout、Horizontal Layout、Vertical Spacer、Horizontal Spacer：用于将布局规范化，使得各组件可以跟随窗口变化而变化。

结果的标红及超链接使用 HTML 语言，打开相应网页的文件之后，搜索关键词，提取出关键词前后 20 个字，并且在关键词两端加上相应的 HTML 标记使其变为红色；标题则通过爬虫的接口函数 get_url 获取编号对应的网址，并且为其加上 HTML 标签，成为超链接。

使用的函数有：

QString str_to_html(const QString& str, const vector<QString>& keywords, int fileNum, int type);//将普通字符串转为 HTML 格式

void show_result(const vector<string> &targetWords, int fileNum);//显示单个网页的结果

void show_results(int *resultPages, int szResultPages, vector<string> targetWords);//显示所有结果网页

调用关系为 :show_results 遍历每一个网页 ,在每一个网页调用 show_result 函数，show_result 函数遍历一个文件内容，对内容调用 str_to_html 函数。

分词

使用结巴分词， jieba 为 Widget 的一个私有成员，程序启动时初始化，加快了分词速度。

主要使用 jieba 的 CutForSearch 成员函数。

页面载入完成之后将自动对 text 目录下的文本文档进行分词，统计各网页词频，保存在 text-index 目录下供倒排索引使用；

用户启动搜索时将对关键词进行分词并交付倒排索引相应的接口函数

int RI_itsc_symdif(std::string *request, int szRequest,int *&intersection, int &szIntersection, int *&symDifference, int &szSymdifference);

和

int RI_exclude(std::string base, std::string *exclude, int szExclude, int *&result, int &szResult);

使用。

使用的函数：

void divide_text();//对爬取的网页进行分词，结果在 text-index 目录下

倒排索引系统，设计者：杨一江

倒排索引系统整合了收集信息，读写文件，响应请求的功能，每个功能根据不同的使用情况改变数据结构的存储和处理方式，适应不同的需求。

收集信息：

问题描述：

根据建立好的正排索引，快速建立初步的倒排索引，并保持一定的模块独立性，便于维护。简单的数学表达式如下

$$\text{Index} \rightarrow \text{Primary Reverse Index}$$

数据结构设计：

红黑树、链表

使用红黑树进行数据的组织，采用线性表虽然存储简单但是会极大的增加搜寻的时间长度，而采用 AVL 树则会造成维护上的极大困难，所以采用折中的办法，进行初步倒排索引的建立。

每个红黑树的节点除了用来连接上下文关系的部分，其中的数据部分主要分为两个部分，字符串和倒排索引的指针。由于添加操作比较多，所以每个倒排索引的元素指向的索引节采用链表的结构。每个链表记录提及该词条的网站的代号。

算法设计：

红黑树查找、插入排序

在正排索引文件夹中依次读取索引文件，顺序的读完所有的词条，并在读取的同时建立起红黑树。类似于树的查找，在当前红黑树中根据字符串寻找该词条，如果该词条不在红黑树当中，则在红黑树中添加这一节点，并记录该文件所对应的网站标号，在堆区申请链表的空间，并维护好该节点。如果发现这个词条在当前红黑树当中，则在该节点的对应链表中插入该节点，插入的时候同时做好排序的工作，采用插入排序的方法，遍历到合适的位置。

复杂度分析：

红黑树建立的时间复杂度：

$$O(n\,log_2\,n)$$

红黑树单个查找的时间复杂度：

$$O(log_2\,n)$$

插入排序的时间复杂度：

$$O(w^2)$$

总体时间复杂度：

$$O\left(n\,log_2\,n + n(logn + w^2)\right) = O(n\,log_2\,n + nw^2)$$，其中$n \gg w$

根据经验和实际的情况，每个红黑树的元素大小当超过 4000 个时，处理速度和程序的稳定性将会极大的下降。处理这种问题有两种方法，第一种减少每次读取文件的个数，分批读取，写成不同的页数；第二种控制红黑树的整体大小，同样写成不同的页数。经过测试，第一种方法在遇到特别的情况（词条数增长急剧不平均）会影响整体的性能；第二种方法在退出的时候会影响排列的稳定性。综合考虑，采取第二种方法来处理这样的情况。

读写文件：

问题描述：

整合初步的倒排索引，加速对索引的查找，建立紧致的倒排索引，并负责存储文件的管理。简单的数学表达式如下

$$\text{Primary Reverse Index} \rightarrow \text{Compact Reverse Index}$$

数据结构设计：文件存取，散列表，线性表

读写文件是通过经过处理过的初步倒排索引，建立起以文件为基础的倒排索引库，于是，文件的生成格式以及读取方式也是这个模块的核心。

生成的文件结构上参考了 ELF 文件在处理不定长度的时候的处理方式，将整个文件分为页、节、单元三个层次。其中，页对应着上一个模块关于红黑树大小控制而生成的多个文件，他们之间是相互并列的，每次都是读取每一页的文件内容。

而每一页都分为数据库头、词条节、索引表节、名称节以及索引头五个部分，每个部分都有着不同的功能。节和单元详细的描述如下

Storage formation:

The RID file is written in binary way. The structure is referred to ELF file, including Database Heading, Entry Section, List Section, Name Section and Index Heading.

Database Heading contains the basic information of the RID file. Starting with "RID", it is easy to distinguish it from other files. The streampos and some constants concerning the standard, which includes maxFile size and so on.

Index Heading tells the contemporary information of the RID file, such as maxWeb_num and Hash strategy, and some constants surrounding entry units and list units.

Entry Section stores the information of reverse index each entry contains. It is laid out as a hash list, which means the size of this section is fixed. Each entry unit tells the validity, tag, name offset, list offset and list size. We introduce a F bit to accelerate the finding procedure.

List Section stores the real information of reverse index, which is laid out as a linear list without fixed size. Entry unit will tell the start node and the size of list.

Name Section stores the strings that each entry features. The formation is relatively arbitary. The size is not fixed and each string ends with '\0', which

means search of entry name only requires one parameter: offset in Name Section.

其中相关的代码如下：

```
struct RID64_Ih {
    struct {
        uint32_t constMax_node; //
        uint32_t constMax_entry; //
        uint32_t constSize_node; //
        uint32_t constSize_entry; //
    };
    uint32_t currentNum_entry; //
    uint32_t currentNum_web; //
    uint8_t Hash; // type of Hash,
    uint8_t ABAND; // decide whe
    struct {
        uint16_t r1;
        uint16_t r2;
        uint16_t r3;
    }; // reserved room
};
```

Figure 4 Index Heading

```
struct RID64_Dh {
    char declaration[4];   // store str
    uint32_t version;   // RID versio
    uint32_t LastMod_Time;   // last
    uint32_t index_off;   // .index h
    uint32_t entry_off;// .entry sect
    uint32_t name_off;   // .name s
    uint32_t list_off;// .list section c
    uint32_t reserve;   // reserve
    struct {
        uint16_t constSize_Dh; // con
        uint16_t constSize_Ih;   // con
        uint8_t constMax_RID_idx;   /
        uint8_t constMax_entry_idx;
        uint8_t constMax_name_idx;
        uint8_t constMax_list_idx;   /
    };
};
```

Figure 3 Database Heading

```
struct RID64_En {
    struct {
        uint16_t Valid : 1;  // entry
        uint16_t Fort : 1;   // used
        uint16_t tag : 13;// used fo
        uint16_t Type : 1;   // defin
                    // 0 denotes th
                    // 1 denotes th
    };
    uint16_t name_off;   // lowe
    uint16_t init_off;   // lower 1
    uint8_t num_node;   // lowe
    struct {
        uint8_t num_node_H : 2;
        uint8_t name_off_H : 3;//
        uint8_t init_off_H : 3;// hig
    };
};
```

Figure 2 Entry unit

```
struct RID64_Nd {
    uint32_t web;
    uint32_t hit;
};
```

Figure 1 List unit

算法设计：

移位折叠法，二次探查法

本部分采用的哈希函数是移位折叠法，外加平方使得散列表尽量的散布稀疏，其简单的数学表达式如下

$$\mathrm{x} = (shift\_folding(\mathrm{string}))^2$$

由于文件内的寻址空间的限制，整个词条节所能容纳的词条数为 8096，所以整个节采用的是闭散列的存储方式。这种方式将会遇到散列冲突的问题。在这边，也是同样为了尽量的使散列表稀疏，采用二次探查法作为散列冲突的响应策略。根据上文所描述的词条结构，将会依次对有效位 V，是否正中 F，标记 tag 以及字符串进行比较，其算法说明如下

Reading strategy:

The streampos of each section is recorded in Database Heading, which means you should read it first.

If a file is freshly created, you can ignore Index Heading since you already own.

Otherwise, you should read contemporary information form Index Heading.

When searching for the entry, you should check V bit first.

First probe:

V = 0, end searching and report "NO FOUND"

otherwise, check F

F = 0, quadratic probing

F = 1, check tag

tag distincted, quadratic probing

otherwise, check Name

Name distincted, quadratic probing

otherwise, end searching and report "FOUND"

Quadratic probe:

(H ± i^2), i++

V = 0, end searching and report "NOT FOUND"

otherwise, check F

F = 1, continue

otherwise, check tag

tag distincted, continue

otherwise, check Name

Name distincted, continue

otherwise, end searching and report "FOUND"

空间复杂度分析

由于只有索引表节和名称节是随着文件中存储的词条数有关 ,因此每一页的空间复杂度可以用以下的表达式说明

$$O(\mathrm{Dh} + \mathrm{Ih} + \mathrm{Entry} * 8096 + \mathrm{n} * \mathrm{nameLength} + \mathrm{n} * \mathrm{listSize} * \mathrm{nodeSize})$$

也就是：

$$O(\mathrm{n} * (\mathrm{nameLength} + \mathrm{listSize} * \mathrm{nodeSize}))$$

根据经验，可以了解到

$$\text{nameLength} \ll \text{listSize} * \text{nodeSize}$$

也就是简化成

$$O(n * (\text{listSize} * \text{nodeSize}))$$

由于存在分页的机制，所以该表达式只是一个页的复杂度分析，其中 n 为页内索引大小。而根据分页的情况，实际上存在$\left\lfloor \frac{N}{4000} \right\rfloor$页，n = 4000。也就是说整个文件的空间复杂度如下

$$O(N * (\text{listSize} * \text{nodeSize}))$$

由于文件本身设计的容量为 8096，而其真正的使用量不超过 4000，所以不存在单个页面超过自身寻址大小的上限 4GB 的情况，因而该公式不会存在上界。实际上，根据经验，每一满页文件的大小都大约在 400KB~500KB 之间。

响应请求：

问题描述：

根据用户的需求，基于紧凑的倒排索引，返回相应逻辑的结果。其中包括求交、补、对称差。为了以后复杂逻辑的建立，因而在这个模块，对索引的组织形式将采取位向量的数据结构，对数据进行交并补算术。

数据结构设计：

位向量

基于位向量的逻辑运算简便，便于复杂逻辑的建立，并且位向量的大小实际上远小于词条的数量，所以采用位向量便于逻辑的建立。

算法设计：

遍历

对位向量组采取遍历，从而得到需求的逻辑。

时间复杂度：

与位向量的长度相关，而位向量的长度就是网站的个数，又由于需要进行逻辑的运算，所以对每个需求的词条都要进行相同的遍历，所以时间复杂度可以表达为

$$O(\text{requestSize} * w)$$

爬虫，pagerank，布尔检索，简单模板栈的实现，设计者：乔裕哲

爬虫模块：爬取网页数据，保存在磁盘中。

爬虫大致流程：

用户输入入口网站和 MAXFILECOUNT

↓

入口网站入队

↓

如果队首网站未在 visitedurls 找到，爬取队首网页的 html 代码，否则出队

↓

分析并找到所有链接，加入 queue 中，(parsehtml 函数)

↓

保存网页数据，同时将网址存到 website 数组中

↓

将网页加入 visitedurls，即访问过的网站

↓

出队，计数变量 filecount+=1

↓

队列为空或 filecount==MAXFILECOUNT 时结束

数据结构设计：

```
int MAXFILECOUNT;//最多爬取的网页数
static set <string> visitedurls;//记录已经访问过的网站
static queue <string> urls;//要访问的网站，用队列实现广度优先
static string filepath = "html\\";
static string textpath = "text\\";
static string urlpath = "url\\";
static int filecount = 0;//记录创建的文件个数
static pair <string, string> pair_result;//用于存储当前访问网站的主机名和资源名
string *website;//编号为i的网站的网址
```

最终保存数据的结果：

根据入口网址创建一个文件夹，并在里面创建 html,url,text 三个文件夹，分别用于存储网页的 html 代码,网页的链接，和网页的中文。每个文件夹下都有 0.txt 1.txt 2.txt...，三个文件夹下对应同名的文本存储同一网站的信息。 完成

后给出一个 website 数组，存储每个爬到的链接，并且 website[i]与 i.txt 对应。

pagerank 模块：建立邻接矩阵，计算 pagerank 值

建立邻接矩阵和记录入度出度流程：init_matrix()函数

从 html 文件夹下读取一个文本文件 A

↓

分析 A 中的 html，找到链接 B

↓

顺序查找 website 数组找到对应链接 B 的数组下标

↓

链接 B 的 in_degree 加一，A 的 out_degree 加一，并修改邻接矩阵

↓

重复直到 html 中的链接都被找到了

↓

分析下一个文件，循环计数变量+1

↓

循环计数变量==filecount 时，结束

计算 pagerank：pagerank()函数

所有网页的 pagerank 初始值为 1

↓

用一个循环将当前 pagerank 值存入 last_pr 数组中

↓

用一个两重循环遍历邻接矩阵 matrix，若 matrix[i][j] = true，则 pr[j]+=last_pr[i]/out_degree[i];

↓

重复直至循环了一百次

相关函数：

```
void sort_by_in_degree(int arr[], int size);/* 直接选择排序,复杂度 O(n²) */
void sort_by_pagerank(int arr[], int size);/* 同上 */
static void init_matrix();//建立矩阵,两层循环，复杂度 O(n*(max{n,m}))，//m为平均每个网页中所含链接数
static void pagerank();//计算 pagerank 值，两重循环，复杂度 O(n²)
```

数据结构设计：

```
static bool **matrix;//邻接矩阵
static double *pr;//网址的pagerank值
static double *last_pr;
static int *in_degree;//记录网址的入度
static int *out_degree;//记录网址的出度
```

布尔检索模块：对用户输入的布尔表达式进行解析，并搜索，最后计算出搜索结果

布尔检索大致流程：

用户输入一个表达式

↓

遍历这个字符数组，若遇到+-*()，则增加一个新的操作符 token，否则以贪婪的方式匹配一个操作数（带中文和空格的一句话，跳过其他 ascii 字符）（analysis_expr 函数）

↓

利用栈将这个 token 数组转化为后缀表达式，转换方法参考教材(to_post_order 函数)

↓

遍历这个 token 数组，对其中为操作数的 token，进行分词和搜索，搜索得到一个 int 数组，也保存在该 token 中（该部分由张帅完成）

↓

利用栈，对后缀表达式进行计算(calc_post_order_expr 函数)

↓

返回最终的计算结果

数据结构：

```
enum OpType{ OP, LB, RB, AND, OR, NOT, END };//枚举型，表示每个toke的类型
```

```
static int isp[7] = { -1, 1, 8, 5, 3, 7, 0 };
static int icp[7] = { -1, 8, 1, 4, 2, 6, 0 };//对isp和icp稍作修改，保证NOT(表达式中的减号)优先级最高
struct Token{
    OpType type;//该token的类型
    std::string str;//该token存储的字符串，用于分词和搜索
    int size;//该token对应的arr数组的大小
    int *arr;//对token分词和搜索之后得到的结果存储在这个动态数组中
};

Token post_order_token[1024];/*这个是最终的后缀表达式*/
int nr_post_token = 0;/*这个是后缀表达式的 token 个数*/

static Token token[1024];//这个是解析后得到的中缀表达式token
static int nr_token = 0;//中缀表达式的token个数
```

函数划分：

```
static Token calc_and(Token lop, Token rop);/*计算与(两个数组中相同部分)*/
static Token calc_or(Token lop, Token rop);/*计算或(两个数组中所有非重复元素)*/
static Token calc_not(Token lop, Token rop);/*计算非(lop 中有且 rop 中没有的元素*/

int calc_post_order_expr(int *&result);/*用来计算后缀表达式，利用栈实现计算，只需对后缀表达式的token数组遍历一次即可，复杂度O(n)*/
static void to_post_order();/*中缀转后缀,其实是对中缀表达式的token数组遍历一次，时间复杂度O(n)*/
bool analysis_expr(const std::string& expr);/*这个是最终的调用函数，调
```

用之后自动解析并转换成后缀表达式(在该函数内调用了 to_post_order 函数)，如果是一个复杂表达式（至少有一个操作符的表达式），则返回 true，否则（只是一句简单的话）返回 false，其实是对 expr 这个字符数组进行一次遍历，时间复杂度 O(n)*/