2020年电子科技大学数学建模校内赛

承　诺　书

我们仔细阅读了大学生数学建模竞赛的竞赛规则.

我们完全明白，在竞赛开始后参赛队员不能以任何方式（包括电话、电子邮件、网上咨询等）与队外的任何人（包括教师）研究、讨论与赛题有关的问题。

我们知道，抄袭别人的成果是违反竞赛规则的，如果引用别人的成果或其他公开的资料（包括网上查到的资料），必须按照规定的参考文献的表述方式在正文引用处和参考文献中明确列出。

我们郑重承诺，严格遵守竞赛规则，以保证竞赛的公正、公平性。如有违反竞赛规则的行为，我们将受到严肃处理。

我们所选题号是（A题 或者 B题）：B题

我们的题目是：基于哈希技术和自动机等理论的关键词匹配算法

参赛年级是（一年级，二年级以上，研究生）：一年级

所属学院（请填写完整的全名，可填多个）：计算机科学与工程学院（网络空间安全学院）、信息与软件工程学院、经济与管理学院

参赛队员姓名、学号：1. 张镕麒　2019081300126
2. 巫玮　2019091613029
3. 唐浩　2019150801005

指导教师或指导教师组负责人：无

是否愿意参加国内赛（是，否）：是

日期：2020年 5 月 17 日

报名队号（请查阅《2020校内赛报名队信息-0514》后填写）： F058

2020 电子科技大学大学生数学建模竞赛

编 号 专 用 页

报名队号（请查阅《2020 校内赛报名队信息-514》后填写）： F058

评阅记录

评阅人				
评分				
备注				

基于哈希技术和自动机等理论的关键词匹配算法

摘要

随着人工服务价格水平的提高，眼下普通民众对于自助维修保养系统的需求日渐增加，但是由于用户对需求配件的描述方式和厂家（商家）对零部件分类方式不统一，系统无法正确匹配到真正的用户需求，导致自助维修保养系统难以大规模普及。本文针对以上问题提出了四种可以部分解决该问题的关键词匹配算法，从而可以推进自助维修保养系统的普及，给人们的生活提供便利。

针对问题一，我们通过观察分析附件二的内容，根据用户在描述询问时的用词和顺序特点，抽象概括出用户描述询问所用的关键词特点的结构模型。

针对问题二，我们通过观察分析附件一的内容，根据厂家零件手册分类零部件时的顺序、从属关系和命名规则，抽象概括出厂家零件手册中各个零部件的结构模型。

针对问题三，本文先是根据问题一和问题二的结果建立了建立用户描述零部件关键词与厂家对零部件分类的对应关系模型。其次通过提出用于两种不同的实际情况的匹配算法——精确匹配和模糊匹配，建立了进而建立用户搜索关键词与零部件唯一编号之间的关系模型，其中精确匹配的效率更高，而模糊匹配的应用范围更广。对于精确匹配，我们给出了哈希匹配算法原理及流程，再建立哈希碰撞分析模型，给出了平均意义下对于一定大小的哈希值域发生哈希冲突的概率，并且分析概率函数，量化地给出了哈希算法性能的理想程度。考虑到厂家数据库零件种类可能只增不减，我们在这种情况下进一步提出了空间复杂度更优的基于哈希树数据结构的精确匹配算法。

然后，我们再进行基于 Aho-Corasic 自动机和 WM 算法的多模式匹配算法，提出了模糊输入集这个概念，在具体应用中，需要统计用户错误输入与其正确意向之间的关系作为模糊输入集的元素，并给其赋予关联性权重，根据权重排序，使得该算法可以在用户的错误输入下匹配到正确的关键词。本文通过设计一种最简单的模糊输入集来验证该算法的可行性与高效性。

最后在算法的验证阶段，我们设计了零件手册数据生成算法，来生成所有可能的零件手册数据，从而来更加强有力地验证前面提出的匹配算法的正确性和普适性。

针对问题四，我们根据上述建立的模型和构建的匹配算法，结合现在的智能大背景，对用户使用自助维修保养系统提出格式化描述方式的建议，也对商家在建造自助维修保养系统时关心的成本及效率问题做出了分析，完成本题的非技术报告。

关键词：哈希函数 Aho-Corasic 自动机 WM 算法 模糊匹配

目录

1 问题重述

现阶段，随着我国的飞速发展，国民的生活水平也随之提高，因此，有更多的人有能力购买汽车。然而，想要长期使用汽车则必需维修和保养。考虑到人工成本，越来越多的人选择自己进行对汽车的日常保养和简单维修。可是，在使用搜索引擎的时候往往会发生找不到需要的零部件情况。由于厂家、卖家对零部件的描述于用户不同，导致零部件不能匹配。如何使得根据用户的搜索内容描述准确找到拥有厂家唯一代码的零部件是一项难题。

本次 2020 年数学建模校内赛的题目就是要求建立刻画用户关键词特点和厂家零件手册的分类特点的分析模型，从而建立实现对用户对零部件的关键词描述和零部件分类匹配的对应关系优化模型，进而建立搜索关键词与零部件编号之间的关系模型，并且根据这个优化模型的结果、结论及建议完成一份非技术报告。

2 问题分析

针对问题一，通过我们对参考附件二所给的例子分析，用户正如题目所给的零部件品牌、车型、配件名顺序描述所需零部件，除此之外我们发现用户的描述可能会存在与零件手册不匹配的现象。我们选择以有序三元组的方式形式化描述用户的描述内容，由此来找到关键词的拓扑顺序。

针对问题二，根据附件一的信息，我们将整个目录抽象成树形结构，借此明确零件手册名称间的顺序关系，发现厂家分类零件的本质。再分析树状图的分类情况，同样通过有序四元组的形式来描述单个零件的位置信息。

针对问题三，我们对提问进行具体化，即提供搜索（匹配）算法，对于给定的用户输入，给出符合用户需求的的零件件号或者部件件号以供自助维修系统识别。用户的描述方式如果对应多个不同零件，要求找到所有满足条件的零件。基于问题一的用户描述特点、问题二的厂家分类特点，我们提出了基于双哈希表的精确匹配算法和基于 AC 自动机模糊匹配算法。通过分析两种算法的优势和缺陷，进一步提出了基于哈希树的精确匹配算法和 WM 模糊匹配算法，在最后分别对两类算法做了分析比较。

3 模型假设

1. 总成与零件的地位相等，即讲总成视为单一的不可分割的零件。

2. 用户的输入及零件手册中的名称及编号皆由英文字符和数字构成。

3. 在哈希过程中不会出现两个哈希函数同时碰撞的情况。

4 名词解释和符号说明

- 图：表示物件与物件之间的关系的数学对象，其基本元素是点和边。
- 树：任意两个顶点间有且只有一条路径的图。
- 模式串：需要进行匹配的短字符串。
- 文本串：可能包含模式串的长字符串。
- 子串：给定字符串的某一个连续部分

- 字符串匹配：寻找模式串在文本串中出现的位置的操作。
- 拓扑顺序：有向图各节点按照先后指向顺序形成的次序

表 1 模型符号说明表

符号定义	符号说明
x	汽车品牌名
y	汽车车型号
Z	配件名
a	器件名
b	部件名
c	部件件号
d	零件名
e	零件件号
D	前缀树的最大深度
k	需要哈希的数据规模
n	哈希映射到的空间大小
u	哈希碰撞的次数
Σ	字符集
S	模式串
M	文本串
S_i	模式串子串
l_i	Trie 树上的节点
$base$	哈希函数所选的基
mod	哈希函数所选的模数
m	模式串集合中，字符串长度最小的模式串的长度
B	Shitf 表中的索引，字符块长度
h	当前扫描过程中长度为 B 的模式串子串
C	前缀长度
P	模式串集合
k	模式串的数目

5 模型的建立与求解

5.1 用户搜索零部件时描述的分析模型

观察分析附录二《零配件查询实例》后，我们发现用户搜索零部件时所用关键词是具有鲜明的拓扑顺序的链状结构。即公司代号+产品代号+配件名称的形式。其中配件名称可为器件+部件（如小松 D155_5 推土铲油缸），可为部件+零件或总成（如 75-9 三一斗杆阀芯, PC56 推土铲总成），也可直接为部件（如 LG833 变扭器，日立 ZAX130-5A 下车架）。可以发现用户描述的配件顺序并不是简单地按照器件+部件+零件的标准形式给出。并且除了公司代号+产品代号外，用户

对于其他名称的描述更加口语化（如力仕德 360 大转盘），使得仅凭字符串匹配的方式无法找到真正的零件编号。用户搜索零部件时所用关键词的结构模型图如下：

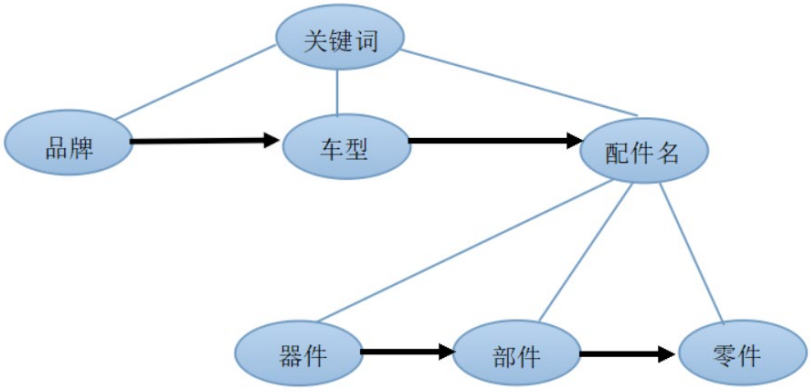

图 1 用户搜索零部件时所用关键词的结构模型图

我们用有序三元组为 $A=\left(x,y,Z\left(b,d\right)\right)$ 表示用户单一的一个需求，其中 x 为品牌名，y 为车型，Z 为配件名，a 为器件名，b 为部件名，d 为零件名。当用户的描述中缺少器件、部件、零件的某一项时，将对应的值复制为0。这样即可使用该三元组代表所有类型的用户描述。

5.2 厂家零件手册对零部件分类特点的分析模型

观察分析附件一《$PC56-7$_零件目录》后，我们发现厂家零件手册的零件分类方式是典型的树状结构目录，其中器件是一级目录，部件是二级目录，零件和总成是三级目录。每一种部件有对应的部件号，每一种零件和总成有对应的零件号。厂家零件手册的分类模式的结构模型图如下：

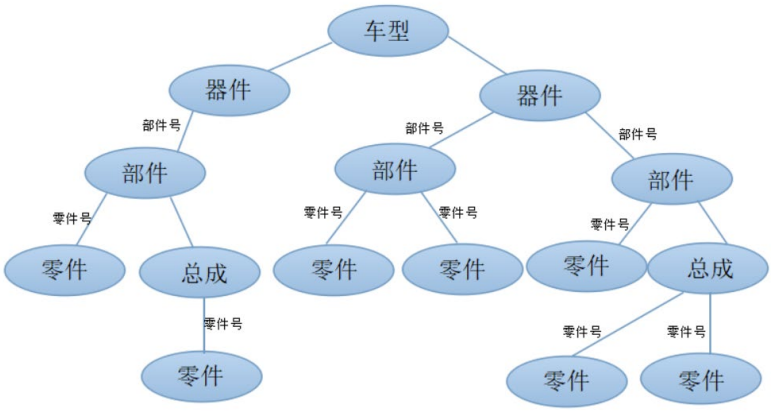

图 2 厂家零件手册分类树形图

在观察零件手册的同时，我们还发现了总成作为的三级目录下，仍然存在由零件和总成组成的四级目录和由零件组成的五级目录。本文中我们将三级目录下的总成近似视作不可拆分的零件，即总成与零件的地位等价。故此时的目录即可视为由器件——部件——零件构成的最多三级目录的满多叉树。对于更多级数的

目录的处理方式与三级目录相同，本文仅讨论后者。

同样的，对于零件手册的单个零件，我们可以用有序二元组$\left(y,Z\left(a,b,c,d,e\right)\right)$描述其地址索引信息，其中$y$为型号，$Z$为配件集合，$a$为器件，$b$为部件，$c$为部件号，$d$为零件名，$e$为零件号。

5.3 问题三

我们根据问题一、问题二的分析，发现用户与厂家在表述同一种零部件时，其核心问题在于两者的表述不同。于是我们考虑通过改变顺序的方式来转换某个描述方式，再进行匹配。我们首先建立了用户描述关键词与厂家零件手册分类的对应关系模型。然后，我们一共建立了两种匹配算法，使得用户在输入之后能得到需要的零件件号或者部件件号。先通过哈希函数分析，我们提出了双哈希字符串匹配算法，并分析了哈希表和哈希树的性能及功能，以用于精准匹配。再基于多模式串匹配的 Aho-Corasic 自动机原理以及 WM 算法原理，建立了功能更强、实践效果更好的模糊匹配算法。

5.3.1 用户关键词与厂家分类的对应关系模型

我们根据问题一与问题二的分析，发现用户在描述零部件时使用的关键词与厂家零件手册分类有指向关系，由于拓扑顺序的相似性，本文仅考虑对单个厂家的单个零件手册（即附件一），多个厂家的多个零件手册情况不再赘述，于是建立一个如下图论模型描述两者之间联系：

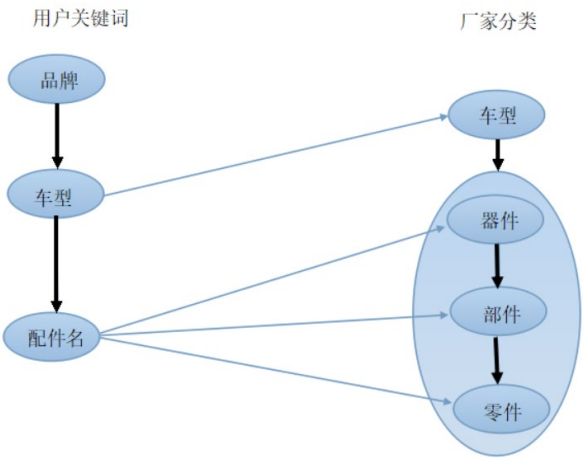

图 3 用户关键词与厂家分类关系图

5.3.2 基于哈希匹配的双哈希字符串精确匹配算法

在最简单的情况下，我们根据附件二提出假设 1，即用户的输入与厂家零件手册中某个配件的路径名完全一致。借此，我们引入哈希技术的相关概念。

哈希技术（又名散列技术）是把任意长度的输入通过散列算法变换成固定长度的输出，该输出就是散列值。这种转换是一种信息发生不可逆损失的压缩映射，也就是，散列值的空间通常远小于输入的空间，不同的输入可能会散列成相同的输出，所以不可能从散列值来确定唯一的输入值。简单的说就是一种将任意长度

的消息压缩到某一固定长度的消息摘要的函数。哈希函数的作用如下图所示。

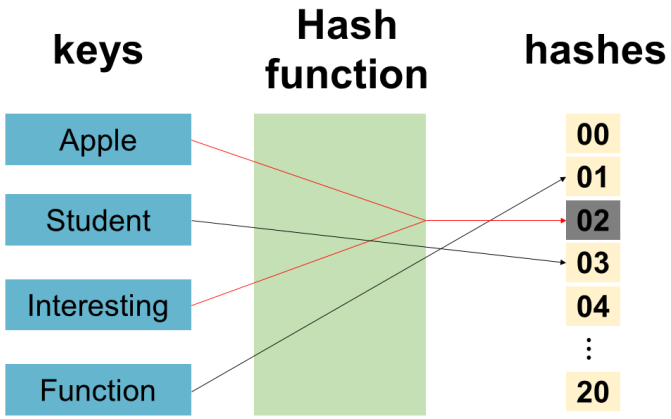

图 4 哈希函数的示范图

本节中将利用哈希技术相同输入会得到相同哈希值，不同输入几乎无法得到相同哈希值的特点，建立用户询问描述和数据库中零件路径信息的对应关系模型。

5.3.2.1 哈希算法的原理与流程

由于仅考虑对于单本零件手册组成的数据库的查询问题，我们将零件手册中的每一种零件的路径，即器件+部件+零件的形式作为哈希函数的输入值，预处理出所有零件的哈希值存储到哈希表中。

本文选取的哈希方式为除留余数法。这种方法可以近似地认为可以将一个字符串输入随机地映射到区间$[0, mod-1]$内。即对字符串的每一位字符 i，更新哈希函数值如下：

$$Hash[i]=\left(\left(Hash[i-1]*base\right)+char[i]\right)\%mod \tag{1}$$

又等于：

$$Hash[i]=s[1]base^{i}+s[2]base^{i-1}+s[3]base^{i-2}+\cdots+s[i] \tag{2}$$

这样有：

$$Hash(l,r)=Hash[r]-Hash[l-1]base^{r-l+1} \tag{3}$$

这样就可以做到匹配所有零件地址索引中的子串，即处理形如器件+部件或部件+零件这种用户询问描述和零件地址索引无法一一对应的情况。

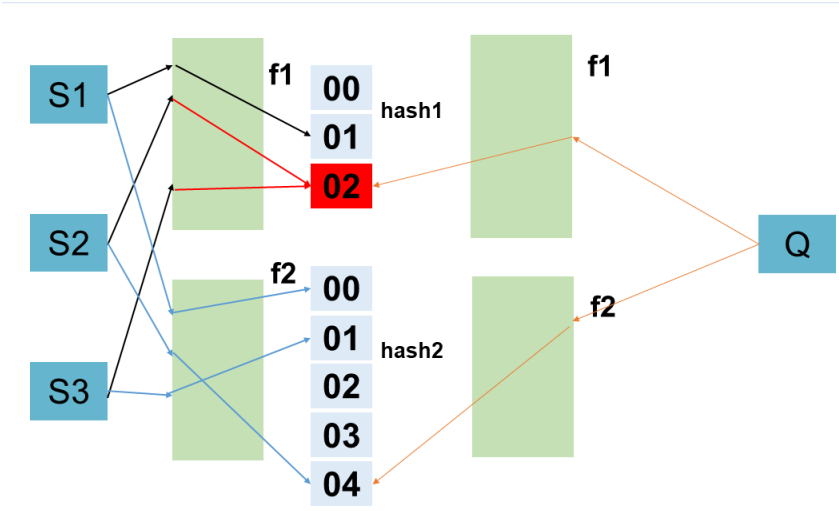

图 5 hash 冲突的情况

本文 hash 采用链地址法发处理冲突，即对 hash 值相同的不同对象添加到 hash 桶的链表上。

为了解决单哈希冲突无法判断答案的情况，我们设计了双哈希算法，当不同的两个零件路径名的 hash1 值相同时，比较 hash2 函数是否相同。通过合理的选取$(base1, mod1)$和$(base2, mod2)$质数对，相比于零件手册零件规模数，我们可以认为 hash2 函数不可能发生冲突。这样通过直接寻址法和比较，我们可以在 O（链长度+hash 处理时间）的时间内找到所有满足条件的配件。具体算法如表所示：

表 2 双哈希精准匹配算法

1. 将所有零件手册上零件的所有可能的地址索引用 hash1，hash2 函数映射为两个哈希值。
2. 将零件的件号和哈希值信息存储到下标为 hash1 的哈希表处，若该位置存在其他零件信息，将本零件信息存入到该位置的队列末尾。
3. 对于每一个用户描述询问字符串，将其格式化后分别计算 hash1 和 hash2 函数的哈希值，直接寻址在哈希表中找到下标为 hash1 的队列，依次与队中元素比较 hash2 的值。
4. 若相等，则表示已找到用户需求的配件。否则，该零件不是用户需求的配件。

5. 3. 2. 2 基于哈希碰撞次数的分析模型

由于要达到匹配两者的目的，通过搜索与分析，我们通过参考哈希函数来计算数据的冲突问题[1]。

为了提高数据访问速度，我们使用了哈希得到散列地址，但由于提前已经设置哈希值域，根据抽屉原理，随之会带来哈希碰撞的问题，即出现不同关键字对应于同一地址的情况。为评估该情况发生的概率，我们假设哈希算法将数据均匀映射到映射空间，并且当冲突后，数据不会放入新的空间，即该数据被遗弃。

根据我们的假设条件，我们如果将 k 个厂家零件手册的数据哈希到大小为 n 的哈希表中，那么我们可以将该模型等价为把 k 个球依次随机放入 n 个盒子，放球时盒子里有球的情况。其中每个球代表为数据，每个盒子代表地址。

由于良好的随机性，每个盒子被放入的概率为$\frac{1}{n}$，因此每个盒子不被放入球概率为$1-\frac{1}{n}$，所以放入 k 个球后，一个盒子里无球的概率就为$\left(1-\frac{1}{n}\right)^k$。根据二项分布期望公式，我们得到放入 k 个球后，空盒数期望为如下：

$$C_1 = n\left(1-\frac{1}{n}\right)^k \tag{4}$$

因此，有球的盒子数目为如下：

$$C_2 = n - n\left(1-\frac{1}{n}\right)^k \tag{5}$$

于是在放入 k 个球过程中，冲突次数即哈希碰撞次数为如下：

$$u(k,n) = k - (n - n(1-\frac{1}{n})^k) \tag{6}$$

5.3.2.3 哈希碰撞敏感性分析模型

通过基于哈希碰撞次数的分析模型我们得到哈希碰撞次数函数关系式(6)，考虑到哈希数据量较大，即 k 值较大，所以有以下近似：

$$u(k,n) = k - \left(n - n\left(1-\frac{1}{n}\right)^{\left(-n\left(-\frac{k}{n}\right)\right)}\right) = k - \left(n - ne^{-\frac{k}{n}}\right) \tag{7}$$

于是我们得到哈希碰撞次数函数关系式$u(k,n)$，再进行敏感性分析，于是又如下公式和模型，得到全微分方程：

$$\frac{\partial u}{\partial k} = 1 - e^{-\frac{k}{n}}, \frac{\partial u}{\partial n} = -1 + \left(1+\frac{k}{n}\right)e^{-\frac{k}{n}} \tag{8}$$

$$du = \frac{\partial u}{\partial k}dk + \frac{\partial u}{\partial n}dn \tag{9}$$

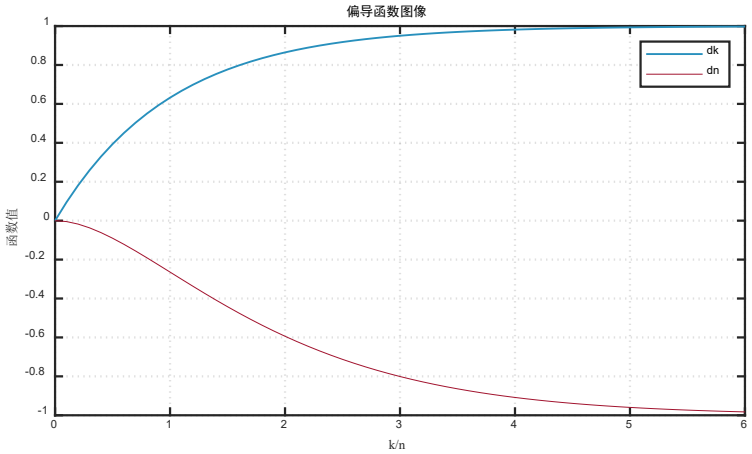

图 6 敏感性分析图

我们通过图 2 可以看出，当$0<\frac{k}{n}<5$，即$k<5n$时， 两者绝对值大小均处于上升状态，且均小于 1，这意味着，当哈希数据量大于哈希表大小五倍时，哈希碰撞次数随数据量变化不大，较为稳定。

同时，我们发现在$\frac{k}{n}>5$时，dk，dn的大小几乎稳定于 1 或-1 附近，这说明了在哈希数据量大于哈希表大小六倍时，其碰撞次数几乎与数据量和哈希表比值线性增长，其增长速度较快，不太稳定。

由此可见，当大规模尝试数据，会极大增大哈希碰撞概率。因此指导我们应尽量控制尝试数据规模，避免超出哈希表大小太多。

5.3.2.4 生日攻击的分析模型

生日攻击是一种密码学攻击手段[2]，所利用的是概率论中生日问题的数学原理。即有 30 人的班级约有 70%的概率存在两个人生日日期相同，此攻击依赖于在随机攻击中的高碰撞概率和固定置换次数（鸽巢原理）。

我们把哈希碰撞平均尝试次数X，定义为哈希碰撞的阈值。再进行考察不发生冲突的概率，即考察当k个球放入n个盒子里，一次也没有发生冲突的概率。显然，这种情况下球的个数不大于盒子的个数。

当放入第一个球后其他球只能放入其他盒子，因此放入 k 个球不发生冲突的概率为：

$$p=\frac{n}{n}\bullet\frac{n-1}{n}\bullet\frac{n-2}{n}\cdots\frac{n-(k-1)}{n}=\frac{n!}{(k-2)!n^k} \tag{10}$$

同样，我们考虑到哈希数据量较大，即 k 值较大，同时，对于哈希映射：$D->R$，$d=|D|$，$r=|R|$，其定义域维数远大于值域维数：

近似得：

$$P(X>k)=\left(1-\frac{1}{n}\right)\left(1-\frac{2}{n}\right)\cdots\left(1-\frac{k-1}{n}\right)$$

$$=\prod_{i=1}^{k-1}\left(1-\frac{i}{n}\right)\approx\prod_{i=1}^{k-1}e^{-\frac{i}{n}}=e^{-\frac{k(k-1)}{2n}} \tag{11}$$

因此阈值 X 的期望为：

$$E(X)=\sum_{k=1}^{d}E(X)=\sum_{k=1}^{d}k\bullet p(X=k)==\sum_{k=1}^{d}k\bullet P(X>k-1)-\sum_{k=1}^{\infty}k\bullet P(X>k)$$

$$=\sum_{k=0}^{d-1}(k+1)\bullet P(X>k)-\sum_{k=1}^{d}k\bullet P(X>k)$$

$$\approx\sum_{k=0}^{d}P(X>k)\approx\sum_{k=0}^{\infty}e^{-\frac{k^2}{2n}}\approx\int_0^{\infty}e^{-\frac{x^2}{2n}}dx=\sqrt{\frac{\pi n}{2}} \tag{12}$$

则对于生日攻击来说，其平均尝试次数为$\sqrt{\frac{\pi n}{2}}$。

同时，这种情况下，考察碰撞次数的问题[3]：

已知每次选择 $p=1/n$，服从二项分布，则得到如下概率分布函数：

$$f(i)=C_k^i p^i(1-p)^{k-i} \tag{13}$$

假设$k=10000,n=2^{14}$, 得到如下概率分布图：

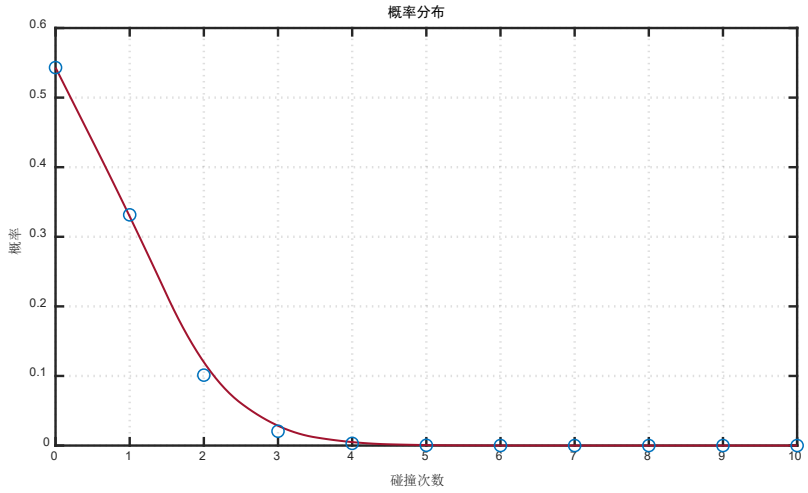

图 7 生日攻击分析图

因此可以看出，为了解决冲突，我们构建的链表有大概 54.3%为空链，有 33.2%有一个元素，有 10.1%有两个元素……这是理想情况下，可以以此数据检验我们哈希算法设计的理想程度。

5.3.2.5 算法改进

哈希表由于为了避免冲突需要提前分配大量内存空间，而且随着厂家零件数目的增加，哈希表的快速查找性能会逐步下降，要想给新的零件分配空间只能扩大哈希映射值域，重新对数据库中的所有零件名重新计算哈希值，可能会带来不小的麻烦。

在计算机科学中，哈希树是一种持久性数据结构[4]，可用于实现集合和映射。哈希树可以广泛应用于那些需要对大容量数据进行快速匹配操作的地方。哈希树不需要额外的平衡和防止退化的操作，效率十分理想。下图为哈希树的结构模型。

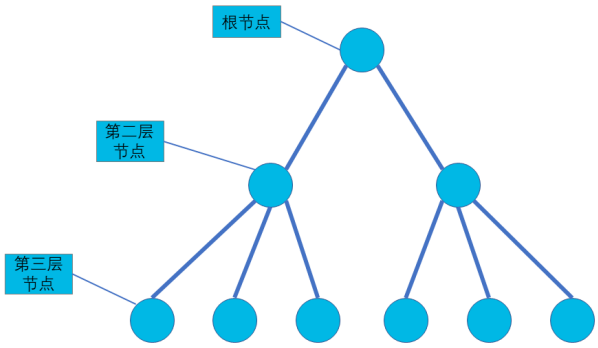

图 8 哈希树的结构模型

哈希树的插入算法流程：

表 3 哈希树插入算法流程

1. 求出前 50 位的质数表（从小到大）prime[50]，每 i 层对应的质数数就是 prime[i-1]（以 0 为起始下标，根节点为第 0 层）；
2. 如果当前的位置没有被占领，我们改变这个节点的 key 和 value 然后标记为 true；
3. 如果当前的节点被占领了，我们就对这一层的 prime[level]取模, 如果这个节点不存在，我们就往下建一个节点然后到下一层去比较；
4. 比较两个数不同的标准是特征编码是否相同（长度和内容）。

哈希树的查找操作流程：

表 4 哈希树查找算法流程

1. 如果当前节点没有被占领，执行第五步；
2. 如果当前节点已经被占领，比较 key 的值；
3. 如果关键值相同，返回 value;
4. 如果不等，执行第五步
5. 计算 index=key%prime[level];
6. 如果 nodes[index]=NULL, 返回查找失败；
7. 如果 nodes[index]为一个已经存在的子节点，我们递归下去往下找，重复第一步操作。

哈希树可以通过分辨质数的方式进行查找并且动态存储。“分辨”就是指这些连续的整数不可能有完全相同的余数序列。n 个不同的质数可以“分辨”的连续整数的个数和他们的乘积相等。从 2 起的连续质数，连续 20 个质数就可以分辨大约 5e26 个数，这个值域范围已经远远超过一个厂家所拥有的零件数了。

从哈希树的结构来说，每层节点的子节点个数为连续的质数。子节点可以随时创建，但无法删除。因此哈希树的结构是动态的，也不像某些哈希算法那样需要长时间的初始化过程，也没有必要为不存在的关键字提前分配空间。

5.3.3 基于 Aho-Corasick 自动机的字符串模糊匹配算法

通过分析附件二中用户询问描述实例，我们发现在绝大多数情况下用户的描述往往并不能与厂家零件手册中的零件路径名称一一对应，哈希精确算法无法很好的处理这种情况，基于这种需求，我们进一步设计了基于 Aho-Corasick 自动机的字符串模糊匹配算法，该算法该算法可以通过构造用户可能输入的模糊范围集 T，并给 T 中所有元素分配权重进行排序，根据统计规律找到最可能的正确输入进行字符串匹配。本节中将利用多模式匹配算法可以高效充分在文本串中充分匹配每一个模式串的特性，将用户的错误输入与用户的真正需求之间建立对应关系模型。

5.3.3.1 前缀树 trie 字符串匹配算法原理

Aho-Corasick 自动机的原理建立在前缀树 trie 上。Trie 是一种树型数据结构，用于存储字符串，典型应用是用于统计和排序大量的字符串（但不仅限于字符串），所以经常被搜索引擎系统用于文本词频统计。

定义[5]：令 $S=\{s_1,s_2,s_3.....,s_n\}$ 是一个定义在字符集合∑上的字符串集合。S 的一个 Trie 结构是一棵 m 分支树 $(|\Sigma|=m)$，每条边存储∑中的一个字符，每一个具有权值的节点 l_i（图中用深色节点表示）对应一个字符串 s_i，并且从根到节点 l_i 的通路上的结点字符就是字符串 s_i，

Trie 的结构特性如下：

① Trie 是有序树,Trie 所表示的字符串互不相同;

② Trie 最多有 n 个叶子结点

③ Trie 的深度 D 为 S 中最长字符串的长度;T 的结点个数是 O(n)(n 为总的字符串长度)

若有 $S=\{'is','her','him','he','his'\}$，则 S 的 Trie 如下图所示：

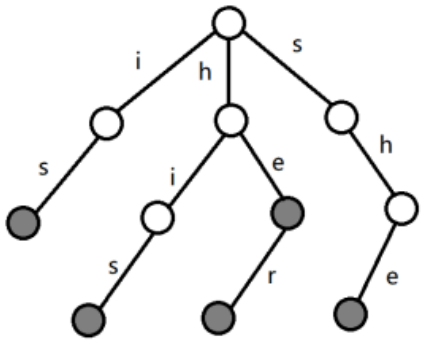

图 9 前缀树的结构示意图

这时，如果我们匹配文本串 $M='\#hihis\#'$ 时，我们需要枚举 M 的每一个起始位置 i 开始，提取 M 的子串 $M_i='M(i)M(i+1)...M(i+D-1)'$，再每次从根节点出发，依次按照 M'的顺序在 trie 中递归查找，最终在 i=3 处提取到了子串 $M_3='his'$，在 i=4 处提取到了子串 $M_4='is\#'$。具体的算法实现请见附录。

5.3.3.2. Aho-Corasick 自动机字符串匹配算法原理

trie 的多模式串匹配需要枚举 M 的所有子串在 trie 上搜索，其中产生了大量的不必要运算。Aho-Corasick 自动机（简称 AC 自动机），该算法在 1975 年产生于贝尔实验室，是著名的多模匹配算法。AC 自动机借助了克努特—莫里斯—普拉特操作（简称 KMP 算法）的思想，同样地在枚举文本串 M 的子串 Mi 且匹配失败时通过建立失配指针 fail 来在 trie 树上进行跳转，从而不必重新逐位地重新枚举子串 Mi+1，大大地减少了计算量。AC 自动机的结构图如下图所示：

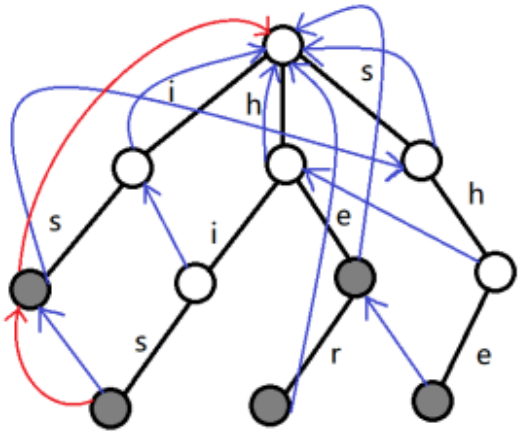

图 10 fail 树和部分 last 指针的结构示意图

fail 指针的定义：最长的当前字符串的后缀在 Trie 上可以查找到的末尾节点。

fail 树的定义：有 trie 树上所有节点的 fail 指针构成的根与 trie 树的根相同的

树。（图中用蓝色箭头表示了 fail 指针）

last 指针的定义：当前节点在 fail 树上的最近的是单词节点的父节点。（图中用红色箭头表示了部分 last 指针）

5.3.3.3 模糊匹配原理

在匹配之前，我们首先进行对两个词的相关性的判断，参考基于用户行为的搜索关键字的权重分析[6]，计算两个词之间关联词公共部分的比例，其比例越高，相关性取值越高，两者之间的相关性越强。我们通过数据信息收集，再采用 FP-Growth 算法[4]进行关联计算。最后，通过权重关联的方式，我们不但可以在匹配第一个关键词后进一步缩小范围，便于节省时间，还可以在用户表述的不是特别准确时，关联出可能的词，加大匹配到正确结果的可能性。

但是限于题目中缺少用户输入大规模数据信息，我们无法统计出一个真正具有实践意义的模糊输入集。取而代之的是，我们模拟了一种最简单的模糊输入集：在用户其他字符输入正确的情况下，仅打错一个字符的字符串构成的集合。由于用户输入字符串长度较小，该集合的大小仍在可接受范围之内，规模为 O(模式串长度^2*字符集大小)。

5.3.3.4 基于 Aho-Corasick 自动机的字符串模糊匹配算法流程

求解 fail 指针的算法：

表 5 求解 fail 指针的算法流程

1.将根节点加入队列。
2.当队列中存在元素时，取队首元素，否则结束算法。
3.队首元素沿着父节点的 fail 指针走，直到走到一个节点满足它的子节点中也有与队首节点相同的字符。把当前节点的 fail 指针指向那个与队首元素相同字符的子节点。
4.将队首元素的所有子节点入队。
5.队首出队，重复步骤 2。

基于 ac 自动机查询文本串 M 的算法：

表 6 ac 自动机查询文本串 M 的算法流程

1.当前字符匹配，表示从当前节点沿着树边有一条路径可以到达目标字符，此时只需沿该路径走向下一个节点继续匹配即可，目标字符串指针移向下个字符继续匹配；
2.当前字符不匹配，则去当前节点失败指针所指向的字符继续匹配，匹配过程

随着指针指向根节点结束。
3.根据情况重复步骤 1 或步骤 2，直到模式串走到结尾为止。

5.3.3.5 算法改进

在实际应用中，零件数目是会不断增加的，随着模式串的增加，ac 自动机的内存消耗巨大，并且需要很长的初始化时间。WM 算法能在模式串满足在一定条件的限制内，解决上述问题，可以证明 WM 算法期望的运行时间小于线性时间[7]。

WM 算法需要预处理三种数据，分别为移动表(SHIFT table)、哈希表(HASH table)、前缀表(PREFIX table). 下图展示了文本串 M='dcbacabcde'，模式串集合 $P=\{'abcde','bcbde','abcabe'\}$，其中模式串集合中，字符串长度最小的模式串的长度 m 为 5，字符块长度 B 为 2，模式串的数目 k 为 3，前缀长度 C=2，预处理后得到的移动表和前缀表如下所示：

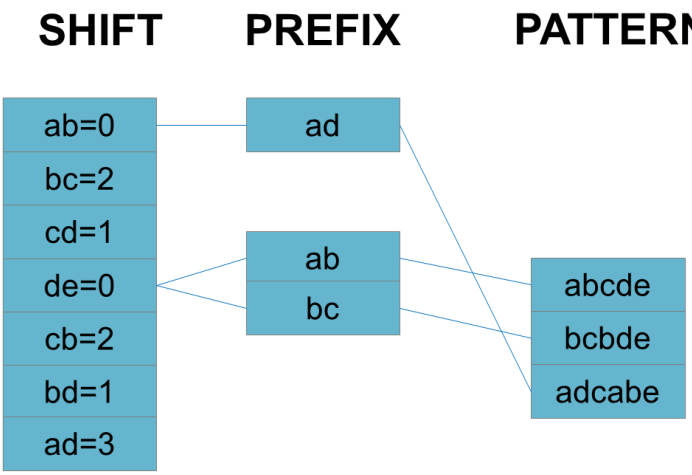

图 11 WM 算法的移动表和前缀表结构示意图

shift 表：用于记录文本串向右移动的长度，即一张跳转表。

hash 表：hash 表记录了所有模式串后缀（长度为 B）与模式串本身的映射关系。当 $shift[h]=0$ 时，B 与对应模式串 P 的映射关系，但是存在一对多的映射，因为模式串集合中存在相同后缀的模式串，所以 hash 表的 value 应该是一个链表的形式，存储多个模式串

prefix 表：prefix 记录了所有模式串前缀（长度为 B）与模式串本身的映射关系。同 hash 表一样，B 与对应模式串 P 的映射关系存在一对多，所以 prefix 表的 value 也是一个链表的形式，存储多个模式串。

WM（Wu-Manber）算法的匹配过程：

表 7 WM 算法匹配流程

1.当 B 个字符构成的子串 h 在模式串集合中没有匹配，即 $shift[h]<0$，则跳转的距离是： $m-B+1$。
2. 当 B 个字符构成的子串 h 在模式串集合中有匹配且非后缀，即 $shift[h]>0$，则跳转的距离是： $shift[h]$。
3. 当 B 个字符构成的子串 h 在模式串集合中且是后缀，即 shift[h]=0，则查 hash 和 prefix 表确定匹配到了哪个模式串。

6 算法检验和比较

6.1 零部件手册数据的生成

我们根据问题 2 厂家零部件手册的分类特点模型设计算法生成一棵拥有固定高度的树，并且这棵树具有节点越深，该层节点个数越多的特点。由此我们枚举所有的叶节点（即零件），概率随层数递增地在当前树中新建一个树权，同时，不同深度的节点具有从属关系，当父节点是新子树的第一个节点时当前节点一定属于新的子树，并给他赋予新的名称。通过以上思路，我们可以设计算法来生成具有零部件手册特点的数据并用以检验。

运行的部分结果如表所示：

表 8 生成的零部件手册的部分数据

HbwYdZSU+ahdmbKmf+42509eZG+aXOXiprq+96897323
HbwYdZSU+ahdmbKmf+42509eZG+aXOXiprq+Q78392N5
HbwYdZSU+ahdmbKmf+3f6R319Z+wgkMpXcf+3109L878
HbwYdZSU+zwIdXWTt+DP0Z7363+hMrVPFdj+2028F371
KyijFwqV+ByRtlnpt+G36d0911+HOZEsWTU+053X1o92
KyijFwqV+ByRtlnpt+G36d0911+uulIslDr+207682J4

其中，1,2,4 项分别为器件，部件，零件名。第三和第五项分别为部件号和零件号。

6.2 hash 表和 hash 树

对于 6.1 所生成的数据库，两者都做到了精确匹配，给出了相同的输出，结果如下表：

表 9 双哈希精确匹配算法匹配结果

输入	输出
HbwYdZSU+ahdmbKmf+aXOXiprq	部件号：42509eZG 零件号：96897323

	零件号：Q78392N5
HbwYdZSU+ahdmbKmf+wgkMpXcf	部件号：3f6R319Z 零件号：31O9L878

在近似意义下，哈希表和哈希树的预处理时间都是 O(字符串长度*零件总个数)，匹配时间为 O(字符串长度)，哈希表的空间复杂度 O(哈希映射的空间大小)，哈希树的空间复杂度为 O(零件总个数)。

6.3 AC 自动机和 WM 算法

对于 6.1 所生成的数据库，在第一个字符产生错误的情况下，两者都做到了模糊匹配，给出了相同的输出，结果如表 4 所示。

表 10 AC 自动机和 WM 算法模糊匹配结果

输入	输出
HbwYdZSU+ahdmbKmf+aXOXiprq	部件号：42509eZG 零件号：96897323 部件号：42509eZG 零件号：Q78392N5
WbwYdZSU+ahdmbKmf+aXOXiprq	部件号：42509eZG 零件号：96897323 部件号：42509eZG 零件号：Q78392N5

预处理性能测试字符串长度为 10-20，测试字符串条数 10000 条时[8]：

表 11 AC 算法和 WM 算法的预处理所需时间空间比较

初始化空间	AC 算法加载条数	WM 算法加载条数	AC 内存	WM 内存	AC 加载耗时	WM 加载耗时
10240	4847	10000	222m	71m	40.01s	0.17s
9000	2394	9000	124m	71m	11.54s	0.13s
5000	2394	5000	111m	68m	11.14s	0.14s
2000	2000	2000	97m	67m	8.10s	0.14s

初始化空间为 10240，加载字符串长度 6-30，匹配次数 100000 次时：

表 12 AC 算法和 WM 算法的预处理与匹配时间比较

待匹配串长度	加载规则条数	AC 加载时间	WM 加载时间	AC 总匹配耗时	WM 总匹配耗时
100	100	1.3s	0.1s	1.0s	0.58s
100	200	1.3s	0.11s	1.1s	0.52s
100	400	1.6s	0.13s	1.2s	0.53s
100	800	3.0s	0.13s	1.3s	0.65s
100	1600	8.1s	0.13s	1.5s	0.78s
100	3200	27s	0.13s	1.6s	0.91s
100	6400	时间过长	0.17s		1.2s

基于以上的测试比较，我们可以分析出 AC 算法和 WM 算法的性能差异。

表 13 AC 算法和 WM 算法的性能比较结果

算法	内存占用	预处理时间	匹配效率	模式串限制
AC 自动机	比较大	大 $O(N*L*L)$	高 模式串内容无影响	无
WM 算法	小	小 $O(N*m)$	随机意义下高 大量模式串前缀相同时效率低	最短长度不能小于 2，且长度最好是相差不大

7 模型评价

对于问题一与问题二，我们使用图像说明了用户和厂家在描述和分类时的两种结构模型，简洁扼要地展现了其中的拓扑顺序。但是，由于现实中用户的描述方式是多种多样的，我们的结构模型不能表示一切的用户询问描述[9]。同时零件手册目录的层数不止三层，但是我们只考虑了三层的基本情况。

在精确匹配方面，哈希技术的简单可行性使得通过多台终端并行运算。哈希是一种以空间换时间的经典算法。哈希树在保持时间复杂度只比哈希表多了近乎常数倍的前提下，进一步显著优化了空间的开销。由于不涉及元素删除的操作，哈希树在此情况下是更优的选择。但在数据库涉及多次添加及删除的情况下，哈希树的表现会劣于哈希表。

在模糊匹配方面，我们在构造模糊输入集时枚举了多种模糊情况，借此尽可能地接近模糊匹配的效果，但是，我们在匹配的模型中只模拟了一类最简单的模糊集合，即单个字符输错的情况，现实情况需要统计整个网络的大数据来对正确输入和模糊输入之间分配权重，通过相似度权重排序从而建立一个更具体、更复杂的模糊集合，再进行模糊匹配。

在多模式串匹配方面，一方面可以通过 AC 自动机和 WM 算法对多组用户的输入字符串处理匹配，同时也可以在精确匹配失败的情况下通过构造模糊输入集进行模糊匹配。WM 算法在随机输入的情况下的性能要明显优于 AC 自动机，但是由于 WM 算法对模式串特性的依赖，使用时需要根据现实中用户的询问情况进行实践测试。

此外，在分析用户描述零部件的关键词与厂家零件手册对零部件的分类的关系时，我们还应分析其与卖家分类的关系（如下图所示），不过由于其处理方法和上文提出的算法完全相同，本文不再赘述。

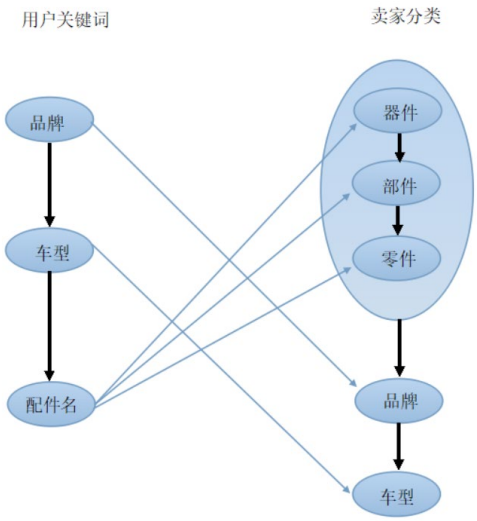

图 12 用户关键词与卖家分类关系图

8 参考文献

[1]阮一峰.哈希碰撞与生日攻击[J/OL].
http://www.ruanyifeng.com/blog/2018/09/hash-collision-and-birthday-attack.html.2020-5-15.
[2]魏福成.哈希冲突的概率[J/OL].
https://blog.csdn.net/u012926924/article/details/50717407.2020-5-15.
[3]暮夏.哈希表之数学原理[J/OL].
https://www.cnblogs.com/niniwzw/archive/2010/05/06/1728822.html.2020-5-16.
[4] 亓国涛,王颖,刘云,包智妍.基于用户行为的搜索关键字的权重分析[J].电脑编程技巧与维护.中国信息产业商会.18-19,25.2016.
[5]刘竹松,何喆.基于 Merkle 哈希树的云存储加密数据去重复研究[J].计算机工程与应用.54(5):85-90,121.2018.
[6]郑丽英.数据结构 Trie 及其应用[J].现代计算机(专业版).中山大学出版社:20-22.2004.
[7]IvanB.G.Liu.浅谈 WM 算法[J/OL].
https://www.cnblogs.com/ladawn/p/9281509.html.2020-5-16.
[8]njuzhoubing.多模匹配-AC 与 WM 算法实测[J/OL].
https://www.cnblogs.com/njuzhoubing/p/4298769.html.2020-5-16.
[9] 丁宏飞.个性化电子商务系统中用户兴趣的研究[D].广州暨南大学.2008.

9 附件

9.1 非技术报告

致需求零部件的用户及供给商家的非技术报告

时代高速发展，智能产品进入千家万户，普及得越来越广泛，人们在富足生活的同时，少不了拥有汽车。而其中，汽车的保养修缮问题逐渐凸显。传统的去4S 或者汽修店的因为成本提高的问题被人们逐渐抛弃，时代的发展与百姓的需求驱使着自助维修保养店的出现，可是由于用户对于需求零件的描述与厂家或商家对零件的分类有着很大的区别，所以自助系统无法识别用户需求这一技术性问题大大地阻碍了自助维修保养店的发展。

对此，我们小组进行了如下步骤的工作，先分析两者的关系，再根据不同的需求设计了多种优于朴素搜索的算法，可供您在遇到此类问题的时候，作为参考。

根据我们的建立的结构模型，我们发现用户和厂家在描述方面最大的不同是表述顺序以及描述时名词的使用。用户在时候描述的时候过于的口语化和生活化，这使得用户在搜索时很难找到需要的零部件。但是，通过分析用户给出的描述特点，我们也找到了令人欣喜的发现，我们发现“的”的作用是分隔关键词，而数字则表示该处的输入字符为汽车型号……这些有着特殊意义的字符可以帮助自助系统更快更准确的对用户描述提炼关键词。

因此我们对用户的建议是：尽量使用分隔符来准确的表达自己的需求，如PC+56+推土铲总成。这样对系统辨识您的需求有很大帮助。此外我们还建议您在描述的时候用词要准确科学，尽量符合专业的描述。

提取关键词之后，我们设计了两大类匹配算法来实现用户输入与零件数据库的匹配过程。其中第一类是利用了哈希技术的精确匹配算法，它可以做到在保证用户输入合法化的前提下，以最快的速度和极高的准确率来找到匹配的零件。并且具有实现简单，系统易维护等特点，并且在匹配时可调用多组终端并行计算，能充分地利用了闲置资源。

而成本方面，需要以系统内存的大小和匹配速度及匹配准确率之间进行权衡，若想过于追求的速度和准确率的话，需要近乎 4 倍及以上地投入成本来获得双倍效果。在使用方面需要仔细权衡。此外，当数据库中的零件种类只增不减时，可以使用哈希树这种数据结构进一步减少内存开销所带来的的成本。

第二类是利用了 ac 自动机理论和 WM 算法的模糊匹配算法。这种方法虽然比前面提到的精确匹配算法要慢，但是应用面更广。首先，商家需要一个成熟的模糊输入集，这可以通过购买各类搜索引擎的云服务来获取其日志，里面记载了很多用户的错误输入与用户实际的需求标答。找到这个模糊输入集之后，利用我们提出的多模式串匹配算法我们可以根据错误的用户输入来找到最接近的用户真实需求。

成本方面，购买搜索引擎的日志的品质决定了搜索速度的快慢，需要商家在两者之间作出权衡。此外，我们提供了两种不同原理的实现方法，在实际使用中

需要根据模糊输入集的特性来实际测验两种算法运行的效率的高低，从而选择最高效的方式来降低成本并获得更好的效果。

9.2 零件手册数据生成算法(c++编写)

```
#include<bits/stdc++.h>
using namespace std;
const int maxn=1e1;
const int len=9;
const int p1=100;
const int p2=15;
const int p3=15;
const int p4=2;
const int p5=1;
const int p6=4;
const int step=5;
int a[step+10]={p1,p2,p3,p4,p5};
char data[maxn+10][(len+2)*step];
char randlower(){return rand()%(26)+'a';}
char randupper(){return rand()%(26)+'A';}
char randnumber(){return rand()%10+'0';}
void randword(int x,int y)
{
    for(int i=1;i<len;i++)
    {
        if((y==step-1||y==step-3)&&rand()%p6)data[x][y*len+i]=randnumber();
        else if(rand()&1)
        {
            data[x][y*len+i]=randlower();
        }
        else
        {
            data[x][y*len+i]=randupper();
        }
    }
```

```
    if(y!=step-1)data[x][(y+1)*len]='+';
}
void copydata(int x,int y,int j)
{
    for(int i=1;i<=len;i++)data[x][j*len+i]=data[y][j*len+i];
}
void makedata(int x,int dep,bool state)
{
    if(dep==step)return;
    if(dep==1)
    {
        if(state)randword(x,dep),randword(x,dep+1),makedata(x,dep+2,1);
        else copydata(x,x-1,dep),copydata(x+1,x,dep),makedata(x,dep+1,0);
    }
    if(state)randword(x,dep),makedata(x,dep+1,1);
    else if((rand()%(a[dep]))==0)randword(x,dep),makedata(x,dep+1,1);
    else copydata(x,x-1,dep),makedata(x,dep+1,0);
}
int main()
{
    freopen("data.txt","w",stdout);
    srand(time(NULL));
    makedata(1,0,1);
    for(int i=2;i<=maxn;i++)makedata(i,0,(rand()%a[0])==0);
    for(int i=1;i<=maxn;i++)printf("%s\n",data[i]+1);
}
```

9.3 双哈希精确匹配算法(c++编写)

```
#include<bits/stdc++.h>
using namespace std;
typedef unsigned long ull;
const ull base=131;
const ull mod1=19260817;
```

```
const ull mod2=19660813;
const int maxn=3e3+10;
const int layer=10;
const int len=20;
struct data
{
	ull x,y,xx,yy,p[layer];
	char id1[len],id2[len],pos[len*3];
}a[maxn];
map<int,vector<int> >mp1,mp2;
char s[len*layer];
int n=1,l,p[len],cnt;
ull hash1(char c[])
{
	int len=strlen(c+1);
	ull ans=0;
	for (int i=1;i<=len;i++)
		ans=(ans*base+(ull)c[i])%mod1;
	return ans;
}
ull hash2(char c[])
{
	int len=strlen(c+1);
	ull ans=0;
	for (int i=1;i<=len;i++)
		ans=(ans*base+(ull)c[i])%mod2;
	return ans;
}
bool comp(data a,data b)
{
	return a.x<b.x;
}
main()
{
```

```
FILE *f=fopen("data.txt","r");
while(fscanf(f,"%s",s+1)!=EOF)
{
    int i=1,cnt=0;
    while(s[i])
    {
        if(s[i]=='+')a[n].p[++cnt]=i;
        i++;
    }
    for(i=a[n].p[2]+1;i<a[n].p[3];i++)
    {
        a[n].id1[i-a[n].p[2]]=s[i];
    }
    for(i=1;i<a[n].p[2];i++)
    {
        a[n].pos[i]=s[i];
    }
    int hh1=hash1(a[n].pos+1),hh2=hash2(a[n].pos+1);
    if(!mp2[hh1].empty())
    {
        bool flag=0;
        for(vector<int>::iterator it=mp2[hh1].begin();it!=mp2[hh1].end();it++)
        {
            int tmp=*it;
            if(hh2==a[tmp].yy)
            {
                flag=1;
                break;
            }
        }
        if(!flag)
        {
            mp2[hh1].push_back(n);
            a[n].xx=hash1(a[n].pos+1);
```

```
                a[n].yy=hash2(a[n].pos+1);
            }
        }
        else
        {
            mp2[hh1].push_back(n);
            a[n].xx=hash1(a[n].pos+1);
            a[n].yy=hash2(a[n].pos+1);
        }
        for(i=a[n].p[3];i<a[n].p[4];i++)
        {
            int v=a[n].p[2]+i-a[n].p[3];
            a[n].pos[v]=s[i];
        }
        for(i=a[n].p[4]+1;s[i];i++)
        {
            a[n].id2[i-a[n].p[4]]=s[i];
        }
        a[n].x=hash1(a[n].pos+1);
        a[n].y=hash2(a[n].pos+1);
         mp1[a[n].x].push_back(n);
         n++;
    }
    fclose(f);
    scanf("%s",s+1);
    l=strlen(s+1);
    for(int i=1;i<=l;i++)if(s[i]=='+')p[++cnt]=i;
    if(cnt==1)
    {
        int h1=hash1(s+1),h2=hash2(s+1),tmp;
        if(!mp2[h1].empty())
        {
            for(vector<int>::iterator it=mp2[h1].begin();it!=mp2[h1].end();it++)
            {
```

```
                    tmp=*it;
                    if(a[tmp].yy==h2)
                    {
                        cout<<"部件号："<<a[tmp].id1+1<<endl;
                    }
                }
            }
        }
        if(cnt==2)
        {
            int h1=hash1(s+1),h2=hash2(s+1),tmp;
            if(!mp1[h1].empty())
            {
                bool flag=0;
                for(vector<int>::iterator it=mp1[h1].begin();it!=mp1[h1].end();it++)
                {
                    tmp=*it;
                    if(a[tmp].y==h2)
                    {
                        if(!flag)cout<<"部件号："<<a[tmp].id1+1<<endl,flag=1;
                        cout<<"零件号："<<a[tmp].id2+1<<endl;
                    }
                }
            }
        }
    }
```

9.4 哈希树的插入和查找算法(c++编写)

```
void insert(HashNode entry int level, int key,int value)
{
    if(this.occupied==false)
    {
        this->key=key;
```

```
            this->value=value;
            this->occupied=true;
            return;
        }
        int index=key%prime[level];
        if(node[index]==NULL)
        {
            nodes[index]=new hashNode();
        }
        level+=1;
        insert(nodes[index],level,key,value);
    }
    int find(HashNode entry,int level,int key/*,int value*/)
    {
        if(this->occupied==true)
        {
            if(this->key==key)
                return this->value
        }
        int index=key%prime[level++];
        //level+=1;
        if(nodes[index]==NULL)
            return fail;
        return find(nodes[index],level,key);
    }
```

9.5 基于 trie 的字符串匹配算法（c++编写）

```
#include<bits/stdc++.h>
using namespace std;
const int maxm=1e5+10;
const int maxn=1e4+10;
char str[100];
struct trie
```

```
{
    trie *ch[30];
    int val;
    trie()
    {
        memset(ch,NULL,sizeof(ch));
        val=0;
    }
}root;
int n,m,len;
void insert()
{
    trie* u=&root;
    len=strlen(str);
    for(int i=0;i<len;i++)
    {
        if(u->ch[str[i]-'a']==NULL)
            u->ch[str[i]-'a']=new trie;
        u=u->ch[str[i]-'a'];
    }
    if(u->val==1)
        return;
    u->val=1;
    return;
}
void find()
{
    trie* u=&root;
    len=strlen(str);
    for(int i=0;i<len;i++)
    {
        if(u->ch[str[i]-'a']==NULL)
        {
            printf("WRONG\n");
```

```
                return ;
            }
            u=u->ch[str[i]-'a'];
        }
        if(u->val==0){printf("WRONG!\n");}
        else if(u->val>1){printf("REPEAT\n");}
        else printf("OK\n"),u->val++;
        return;
    }
    int main()
    {
        scanf("%d",&n);
        for(int i=1;i<=n;i++)
        {
            scanf("%s",str);
            insert();
        }
        scanf("%d",&m);
        for(int i=1;i<=m;i++)
        {
            scanf("%s",str);
            find();
        }
    }
```

9.6 ac 自动机模糊匹配算法（c++编写）

```
#include<bits/stdc++.h>
using namespace std;
const int maxn=3e4+10;
const int lens=20;
const int layer=10;
char str[lens*layer+maxn];
int n=1,len,tot;
struct node
{
```

```
    node *ch[26],*nxt,*last;
    vector<int> ed;
    node():ed(0),nxt(NULL),last(NULL){memset(ch,0,sizeof(ch));}
}root,*u;
struct data
{
     int p[layer];
     char id1[lens],id2[lens],pos[lens*layer];
}a[maxn];
void insert(char *s)
{
    len=strlen(s);
    u=&root;
    for(int i=0;i<len;i++)
    {
        if(!(s[i]<='z'&&s[i]>='a'))s[i]+='a'-'A';
        if(u->ch[s[i]-'a']==NULL)
            u->ch[s[i]-'a']=new node();
        u=u->ch[s[i]-'a'];
    }
    u->ed.push_back(n);
}
void build()
{
    u=&root;
    queue<node*>q;
    while(!q.empty())q.pop();
    u->nxt=u;
    for(int i=0;i<26;i++)
    {
        if(u->ch[i]==NULL)
            u->ch[i]=u;
        else
            u->ch[i]->nxt=u,q.push(u->ch[i]);
```

```
        }
        while(!q.empty())
        {
            u=q.front();
            q.pop();
            for(int i=0;i<26;i++)
            {
                if(u->ch[i]==NULL)
                    u->ch[i]=u->nxt->ch[i];
                else
                {
                    u->ch[i]->nxt=u->nxt->ch[i];

u->ch[i]->last=u->ch[i]->nxt->ed.size()?u->ch[i]->nxt:u->ch[i]->nxt->last;
                    q.push(u->ch[i]);
                }
            }
        }
    }
    void query(char *s)
    {
        int len=strlen(s);
        u=&root;
        for(int i=0;i<len;i++)
        {
            u=u->ch[s[i]-'a'];
            for(node* j=u->ed.size()?u:u->last;j;j=j->last)
            {
                for(vector<int>::iterator it=j->ed.begin();it!=j->ed.end();it++)
                {
                    int tmp=*it;
                    printf("部件号：%s\n",a[tmp].id1+1);
                    printf("零件号：%s\n",a[tmp].id2+1);
                }
```

```
            }
        }
    }
    void vague(char *s,int len)
    {
        int cnt=1;
        for(int i=0;i<len;i++)
        {
            for(int j=0;j<26;j++)
            {
                if(s[i]==j+'a')continue;
                for(int o=0;o<len;o++)
                {
                    s[cnt*len+o]=s[o];
                }
                s[cnt*len+i]=j+'a';
                cnt++;
            }
        }
    }
    int main()
    {
        FILE *f=fopen("data.txt","r");
        while(fscanf(f,"%s",str+1)!=EOF)
        {
            int i=1,cnt=0;
            while(str[i])
            {
                if(str[i]=='+')a[n].p[++cnt]=i;
                i++;
            }
            for(i=a[n].p[2]+1;i<a[n].p[3];i++)
            {
                a[n].id1[i-a[n].p[2]]=str[i];
```

```
        }
        for(i=1;i<a[n].p[1];i++)
        {
            a[n].pos[i]=str[i];
        }
        for(i=a[n].p[1]+1;i<a[n].p[2];i++)
        {
            a[n].pos[i-1]=str[i];
        }
        for(i=a[n].p[3]+1;i<a[n].p[4];i++)
        {
            int v=a[n].p[2]+i-a[n].p[3]-2;
            a[n].pos[v]=str[i];
        }
        for(i=a[n].p[4]+1;str[i];i++)
        {
            a[n].id2[i-a[n].p[4]]=str[i];
        }
         insert(a[n].pos+1);
         n++;
}
fclose(f);
build();
while(scanf("%s",str+1)!=EOF)
{
        int l=strlen(str+1),p[layer]={0},cnt=0;
        for(int i=1;i<=l;i++)if(str[i]=='+')p[++cnt]=i;
        p[++cnt]=l+1;
        for(int i=1;i<=cnt;i++)
        {
            for(int j=p[i]+1;j<p[i+1];j++)
            {
                str[j-i]=str[j];
            }
```

```
            }
            for(int i=1;str[i];i++)str[i]=str[i]>='a'&&str[i]<='z'?str[i]:str[i]-'A'+'a';
            l-=cnt-1;
            str[l+1]=0;
            puts(str+1);
            vague(str+1,l);
            query(str+1);
        }
    }
```

9.7 WM 算法核心算法实现（c++编写）

```
while(text<=textend)
{
    h=(*text<<Hbits)+(*(text-1));
    if(Long) h=(h<<Hbits)+(*(text-2));
    shift=SHIFT[h];
    if(shift==0)
    {
        text_prefix=(*(text-m+1)<<8)+*(text-m+2);
        p=HASH[h];
        p_end=HASH[h+1];
        while(p++<p_end)
        {
            if(text_prefix!=PREFIX[p])continue;
            px=PAT_POINT[p];
            qx=text-m+1;
            while(*(px++)==*(qx++));
            if(*(px-1)==0))ans++;
            shift=1;
        }
    }
    text+=shift;
}
```