词典及容错式检索

Dictionary and tolerant retrieval

*节选自王斌老师《现代信息检索》课程的课件

请按时提交作业

本周推荐：
https://mp.weixin.qq.com/s/sM739XOPXNVfM0dTkSnPCw

论文写作专题报告会

《论文写作小白的成长之路》

兰艳艳
中国科学院计算技术研究所研究员

《计算机视觉会议论文从投稿到接收》

施柏鑫
北京大学研究员

《谈如何写一篇合格的国际学术论文》

赵鑫
中国人民大学副教授

报告时间：
3月17日下午（周二）14:30-17:30
报告形式：
线上直播
参与方式：
扫描下方二维码，添加小智（微信号：baai03）为好友，备注"Live+姓名+单位/院校+职位/专业"，并发送口令"**live0317**"入群，获取直播间地址。

提纲

① 上一讲回顾

② 词典

③ 通配查询

④ 编辑距离

⑤ 拼写校正

提纲

① 上一讲回顾

② 词典

③ 通配查询

④ 编辑距离

⑤ 拼写校正

上一讲内容

- 文档
- 词条/词项
- 短语查询的处理(双词索引和位置索引)

文档

- 索引的基本单位
  - 与文件不是一回事，严格地说，一篇文档可能包含多个文件，也可能一个文件包含多篇文档
  - 依赖于具体应用
    - 句子级检索： 一个句子为一篇文档
    - 段落级检索： 一段文本为一篇文档
    - ……

词类(type)/词条(token)的区别

- 词条(Token) - 词或者词项在文档中出现的实例，出现多次算多个词条
- 词类(Type) - 多个词条构成的等价类(equivalence class)集合
- *In June, the dog likes to chase the cat in the barn.*
- 12 个词条, 9个词类

- 词类经过一些处理(去除停用词、归一化)之后，最后用于索引的称为为词项

词条化中考虑的问题

- 词之间的边界是什么？空格？撇号还是连接符？
- 上述边界不一定是真正的边界 (比如, 中文)
- 另外荷兰语、德语、瑞典语复合词中间没有空格 (*Lebensversicherungsgesellschaftsangestellter*)

词项归一化中的问题

- 词项实际上是一系列词条组成的等价类
- 如何定义等价类？
  - 数字 (3/20/91 vs. 20/3/91)
  - 大小写问题
  - 词干还原，Porter工具
- 形态分析: 屈折 vs. 派生
- 其他语言中词项归一化的问题
  - 比英语中形态更复杂
  - 芬兰语: 单个动词可能有12,000 个不同的形式different forms
  - 重音符号、元音变音问题（umlauts，由于一个音被另一个音词化而导致的变化，尤其是元音的变化）

位置(信息)索引

- 在无位置信息索引中，每条倒排记录只是一个docID
- 在位置信息索引中，每条倒排记录是一个docID加上一个位置信息表
- 一个查询的例子: *“to_1 be_2 or_3 not_4 to_5 be_6”*

TO, 993427:

‹ 1: ‹7, 18, 33, 72, 86, 231›;
2: ‹1, 17, 74, 222, 255›;
4: ‹8, 16, 190, 429, 433›;
5: ‹363, 367›;
7: ‹13, 23, 191›; . . . ›

BE, 178239:

‹ 1: ‹17, 25›;
4: ‹17, 191, 291, 430, 434›;
5: ‹14, 19, 101›; . . . ›

第4篇文档能够与查询匹配！

位置信息索引

- 基于位置信息索引，能够处理短语查询(phrase query)，也能处理邻近式查询(proximity query)

本讲内容

- 词典的数据结构：访问效率和支持查找的方式
- 容错式检索(Tolerant retrieval): 如果查询词项和文档词项不能精确匹配时如何处理？
  - 通配查询：包含通配符*的查询
  - 拼写校正：查询中存在错误时的处理

提纲

① 上一讲回顾

② 词典

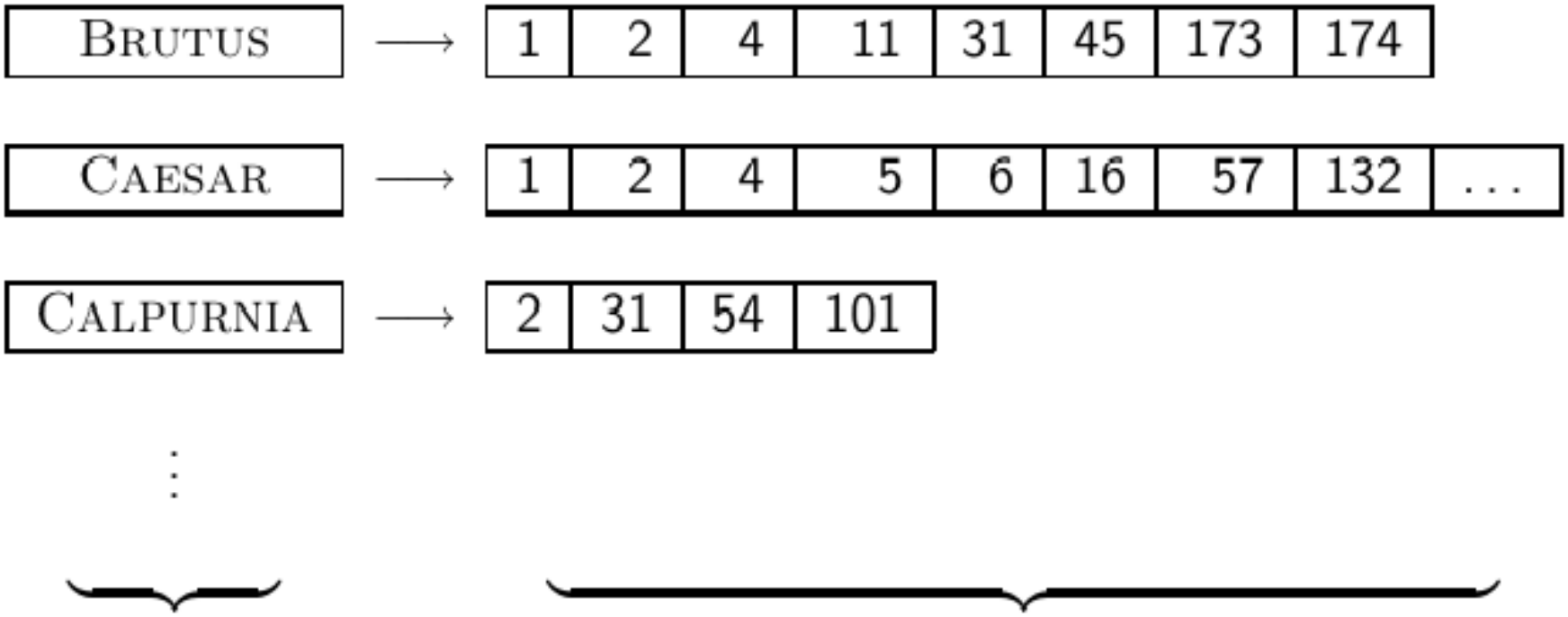
③ 通配查询

④ 编辑距离

⑤ 拼写校正

倒排索引索引

对每个词项*t*, 保存所有包含*t*的 文档列表



词典

- 词典是指存储词项词汇表的数据结构
- 词项词汇表(Term vocabulary): 指的是具体数据
- 词典(Dictionary): 指的是数据结构

采用定长数组的词典结构

- 对每个词项，需要存储:
  - 文档频率
  - 指向倒排记录表的指针
  - . . .
- 暂定每条词项的上述信息均采用定长的方式存储

- 假定所有词项的信息采用数组存储

采用定长数组的词典结构

词项	文档频率	指向倒排记录表的指针
a	656 265	→
aachen	65	→
…	…	…
zulu	221	→

空间消耗： 20字节 4字节 4字节

词项定位(即查词典)

- 输入“信息”，如何在词典中快速找到这个词?
- 很多词典应用中的基本问题。
- 以下介绍支持快速查找的词典数据结构。

用于词项定位的数据结构

- 主要有两种数据结构：哈希表和树
- 有些IR系统用哈希表，有些系统用树结构
- 采用哈希表或树的准则：
  - 词项数目是否固定或者说词项数目是否持续增长？
  - 词项的相对访问频率如何？？
  - 词项的数目有多少？

哈希表

哈希函数，输入词项，输出正整数(通常是地址)

f(信息)=18, f(数据)=19,
f(挖掘)=19

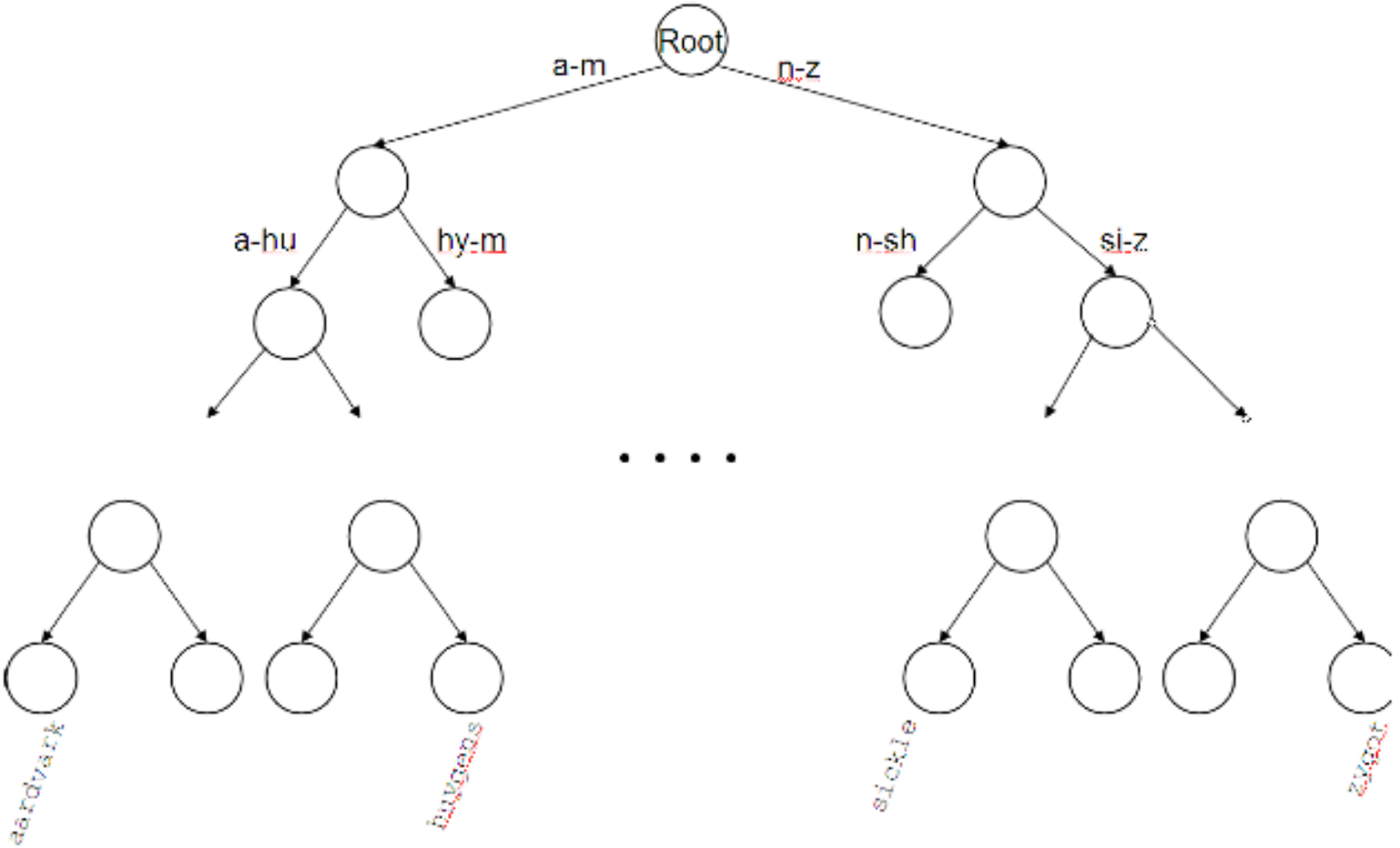
哈希表

- 每个词项通过哈希函数映射成一个整数
- 尽可能避免冲突
- 查询处理时： 对查询词项进行哈希，如果有冲突，则解决冲突，最后在定长数组中定位
- 优点: 在哈希表中的定位速度快于树中的定位速度
  - 查询时间是常数
- 缺点：
  - 没办法处理词项的微小变形 (*resume* vs. *résumé*)
  - 不支持前缀搜索 (比如所有以*automat*开头的词项)
  - 如果词汇表不断增大，需要定期对所有词项重新哈希

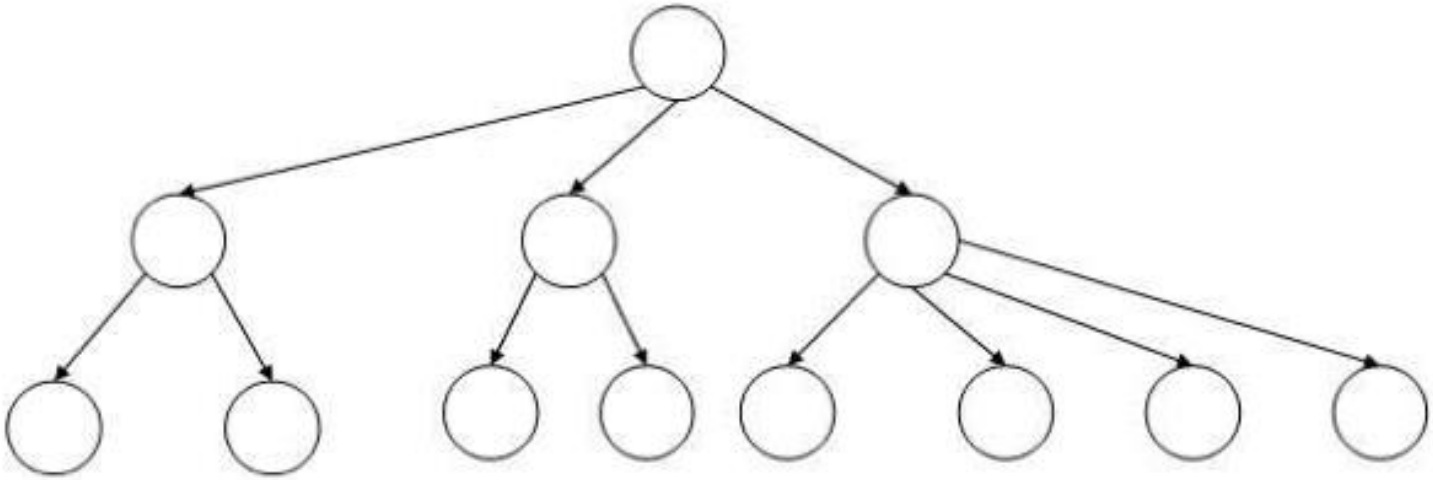
树

- 树可以支持前缀查找
- 最简单的树结构：二叉树
- 搜索速度略低于哈希表方式： $O(\log M)$, 其中 M 是词汇表大小，即所有词项的数目
- $O(\log M)$ 仅仅对平衡树成立
- 使二叉树重新保持平衡开销很大
- B-树 能够减轻上述问题
- B-树定义：每个内部节点的子节点数目在 $[a, b]$之间，其中 a, b 为合适的正整数, e.g., [2, 4].

二叉树



B-树



提纲

① 上一讲回顾

② 词典

③ 通配查询

④ 编辑距离

⑤ 拼写校正

通配查询的处理

- mon*: 找出所有包含以 *mon*开头的词项的文档
- 如果采用B-树词典结构，那么实现起来非常容易，只需要返回区间mon ≤ t < moo上的词项t
- *mon: 找出所有包含以*mon*结尾的词项的文档
  - 将所有的词项倒转过来，然后基于它们建一棵附加的树
  - 返回区间nom ≤ t < non上的词项t
- 也就说，通过上述数据结构，可能得到满足通配查询的一系列词项，然后返回任一词项的文档

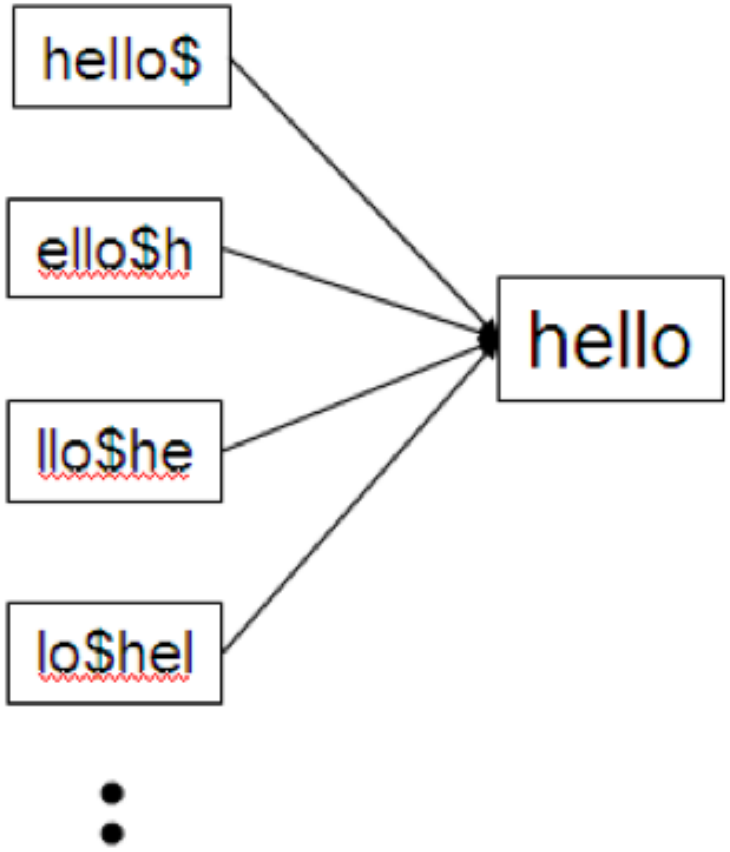
词项中间的 *号处理

- 例子: m*nchen
- 在B-树中分别查找满足m*和 *nchen的词项集合，然后求交集
- 这种做法开销很大
- 另外一种方法: 轮排(permuterm) 索引
- 基本思想：将每个通配查询旋转，使*出现在末尾
- 将每个每个旋转后的结果存放在词典中，即B-树中

轮排索引

- 对于词项hello: 将 *hello$, ello$h, llohe, lohel,* 和 *o$hell* 加入到 B-树中，其中 $ 是一个特殊符号
- 即在词项前面再加一层索引

轮排结果 → 词项的映射示意图



轮排索引

- 对于hello，已经存储了 *hello$, ello$h, llohe, lohel,* 和*o$hell*
- 查询
  - 对于 X, 查询 X$
  - 对于 X*, 查询 X*$
  - 对于 *X, 查询 X$*
  - 对于 *X*, 查询 X*
  - 对于 X*Y, 查询 Y$X*
  - 例子: 假定通配查询为 hel*o, 那么相当于要查询o$hel*
- 轮排索引称为轮排树更恰当
- 但是轮排索引已经使用非常普遍

使用轮排索引的查找过程

- 将查询进行旋转，将通配符旋转到右部
- 同以往一样查找B-树
- 问题：相对于通常的B-树，轮排树的空间要大4倍以上 (经验值)

k-gram 索引

- 比轮排索引空间开销要小
- 枚举一个词项中所有连读的k个字符构成的k-gram 。
- 2-gram称为二元组(bigram)
- 例子: from *April is the cruelest month* we get the bigrams: *$a ap pr ri il l$ $i is s$ $t th he e$ $c cr ru ue el le es st t$ $m mo on nt h$*
- 同前面一样，$ 是一个特殊字符
- 构建一个倒排索引，此时词典部分是所有的2-gram，倒排记录表部分是包含某个2-gram的所有词项
- 相当于对词项再构建一个倒排索引(二级索引)

3-gram(trigram)索引的例子

k-gram (bigram, trigram, . . .) 索引

- 需要注意的是，这里有两个倒排索引
- 词典-文档的倒排索引基于词项返回文档
- 而*k*-gram索引用于查找词项，基于查询包含的k-gram查找词项

利用2-gram索引处理通配符查询

- 查询mon* 可以先执行布尔查询: $m AND mo AND on
- 该布尔查询会返回所有以前缀*mon*开始的词项 . . .
- . . . 当然也可能返回许多伪正例，比如MOON.
- 因此，必须要做后续的过滤处理
- 余下的词项将在词项-文档倒排索引中查找文档
- *k*-gram索引 vs. 轮排索引
  - *k*-gram索引的空间消耗小
  - 轮排索引不需要进行后过滤

Google对通配符查询的支持极其有限的原因

- 问题 1: 一条通配符查询往往相当于执行非常多的布尔查询
- 对于 [gen* universit*]: geneva university OR geneva université OR genève university OR genève université OR general universities OR . . .
- 开销非常大

- 问题 2: 用户不愿意敲击更多的键盘
- 如果允许[pyth* theo*]代替 [pythagoras' theorem]的话，用户会倾向于使用前者
- 这样会大大加重搜索引擎的负担
- Google Suggest是一种减轻用户输入负担的好方法

提纲

① 上一讲回顾

② 词典

③ 通配查询

④ **编辑距离**

⑤ 拼写校正

拼写校正

- 两个主要用途
  - 纠正待索引文档
  - 纠正用户的查询
- 两种拼写校正的方法
- 词独立(Isolated word)法
  - 只检查每个单词本身的拼写错误
  - 如果某个单词拼写错误后变成另外一个单词，则无法查出，e.g., *an asteroid that fell form the sky*
- 上下文敏感(Context-sensitive)法
  - 纠错时要考虑周围的单词
  - 能纠正上例中的错误 *form/from*

关于文档校正

- 本课当中我们不关心文档的拼写校正问题 (e.g., MS Word)
- 在IR领域, 我们主要对OCR处理后的文档进行拼写校正处理. (OCR = optical character recognition，光学字符识别)
- IR领域的一般做法是：不改变文档

查询校正

- 第一种方法: 词独立(isolated word)法
- 假设1: 对需要纠错的词存在一系列“正确单词形式”
- 假设2: 需要提供存在错误拼写的单词和正确单词之间的距离计算方式
- 简单的拼写校正算法: 返回与错误单词具有最小距离的”正确”单词
- 例子: i*nformaton* → *information*
- 可以将词汇表中所有的单词都作为候选的“正确”单词
- 这种方式为什么有问题?

使用词汇表的几种其他方式

- 采用标准词典 (韦伯词典, 牛津词典等等)
- 采用领域词典 (面向特定领域的IR系统)
- 采用文档集上的词项词汇表，但是每个词项均带有权重

单词间距离的计算

- 以下将介绍几种计算方法
- 编辑距离(Edit distance或者Levenshtein distance)
- 带权重的编辑距离
- *k*-gram 重叠率

编辑距离

- 两个字符串s_1和s_2编辑距离是指从 s_1 转换成s_2所需要的最短的基本操作数目
- Levenshtein距离: 采用的基本操作是插入(insert)、删除(delete)和替换(replace)
- Levenshtein距离 *dog-do*: 1
- Levenshtein距离 *cat-cart*: 1
- Levenshtein距离 *cat-cut*: 1
- Levenshtein距离 *cat-act*: 2
- Damerau-Levenshtein距离 *cat-act*: 1
- Damerau-Levenshtein还包括两个字符之间的交换（transposition）操作

Vladimir Iosifovich Levenshtein

- 俄罗斯科学家(1935-)
- 研究信息论、纠错理论
- 毕业于莫斯科国立大学
- 1965年提出Levenshtein距离
- 2006年获得IEEE Richard W. Hamming Medal

Edit Distance

- The minimum edit distance between two strings is the minimum number of editing operations
  - ✓ Insertion
  - ✓ Deletion
  - ✓ Substitution
- Needed to transform one into the other

Uses of Edit Distance

✓ Spell correction

The user typed "graffe"

Which is closest?

- graf
- graft
- grail
- giraffe

✓ Computational Biology

- Align two sequences of nucleotides

```
AGGCTATCACCTGACCTCCAGGCCGATGCCC
TAGCTATCACGACCGCGGTCGATTTGCCCGAC
```

- Resulting alignment:

```
-AGGCTATCACCTGACCTCCAGGCCGA--TGCCC---
TAG-CTATCAC--GACCGC--GGTCGATTTGCCCGAC
```

✓ Also for Machine Translation, Information Extraction, Speech Recognition

Uses of Edit Distance

- Evaluating Machine Translation and speech recognition

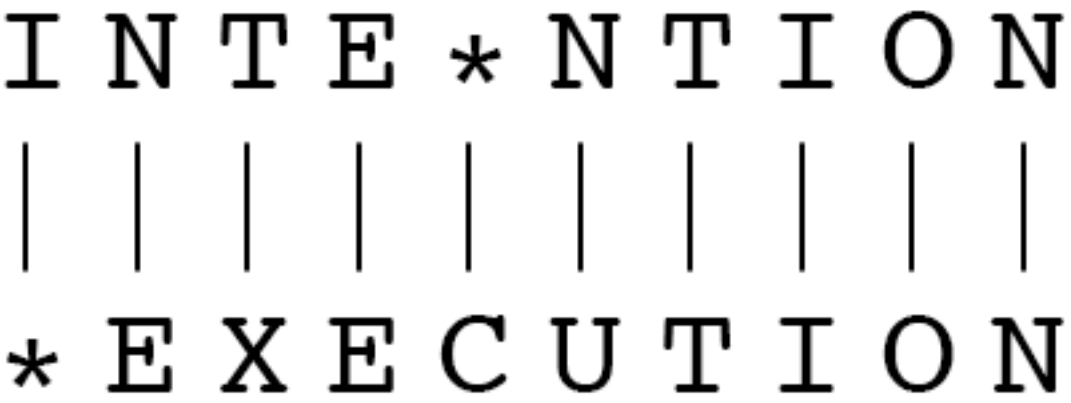
  Spokesman confirms senior government adviser was shot

  Spokesman said the senior adviser was shot dead

- Named Entity Extraction and Entity Coreference
  - IBM Inc. announced today
  - IBM profits
  - Stanford President John Hennessy announced yesterday
  - for Stanford University President John Hennessy

Minimum Edit Distance

- Two strings and their **alignment**:



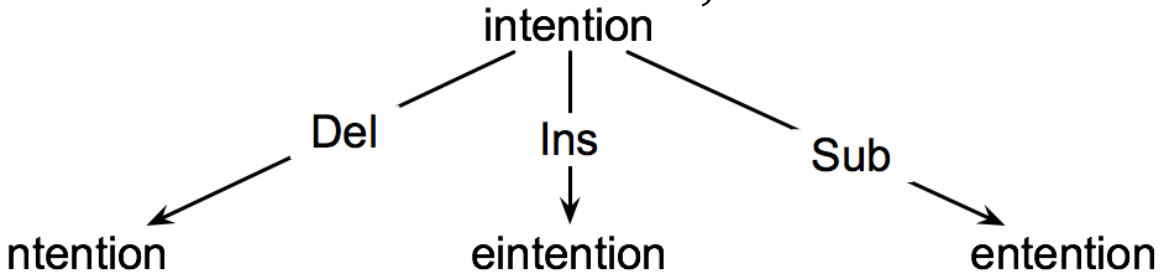
Minimum Edit Distance

```
I N T E * N T I O N
| | | | | | | | | |
* E X E C U T I O N
d s s   i s
```

- ✓ If each operation has cost of 1
  - Distance between these is 5
- ✓ If substitutions cost 2
  - Distance between them is 8

How to find the Min Edit Distance?

- Searching for a path (sequence of edits) from the start string to the final string:
  - **Initial state**: the word we're transforming
  - **Operators**: insert, delete, substitute
  - **Goal state**: the word we're trying to get to
  - **Path cost**: the number of edits, what we want to minimize



Defining Min Edit Distance

- For two strings
  - X of length *n*
  - Y of length *m*
- We define D(*i*,*j*) as the edit distance between X[1..*i*] and Y[1..*j*]
  - i.e., the first *i* characters of X and the first *j* characters of Y
- The edit distance between X and Y is thus D(*n*,*m*)

Defining Min Edit Distance (Levenshtein)

- Initialization

  $D(i,0) = i$

  $D(0,j) = j$

- Recurrence Relation:

  For each $i = 1 \ldots N$

  For each $j = 1 \ldots M$

$$D(i,j) = \min \begin{cases} D(i-1,j) + 1 \\ D(i,j-1) + 1 \\ D(i-1,j-1) + \begin{cases} 2; \text{ if } X(i) \neq Y(j) \\ 0; \text{ if } X(i) = Y(j) \end{cases} \end{cases}$$

- Termination:

  D(N,M) is distance

Example

```
I N T E * N T I O N
| | | | | | | | | |
* E X E C U T I O N
d s s   i s
```

- ✓ If each operation has cost of 1
  - Distance between these is 5
- ✓ If substitutions cost 2
  - Distance between them is 8

The Edit Distance Table

N	9									
O	8									
I	7									
T	6									
N	5									
E	4									
T	3									
N	2									
I	1									
#	**0**	1	2	3	4	5	6	7	8	9
	#	E	X	E	C	U	T	I	O	N

N	9									
O	8									
I	7									
T	6									
N	5									
E	4									
T	3									
N	2									
I	1									
#	0	1	2	3	4	5	6	7	8	9
	#	E	X	E	C	U	T	I	O	N

$$D(i,j) = \min \begin{cases} D(i-1,j) + 1 \\ D(i,j-1) + 1 \\ D(i-1,j-1) + \begin{cases} 2; & \text{if } S_1(i) \neq S_2(j) \\ 0; & \text{if } S_1(i) = S_2(j) \end{cases} \end{cases}$$

$$D(i,j) = \min \begin{cases} D(i-1,j) + 1 \\ D(i,j-1) + 1 \\ D(i-1,j-1) + \begin{cases} 2; & \text{if } S_1(i) \neq S_2(j) \\ 0; & \text{if } S_1(i) = S_2(j) \end{cases} \end{cases}$$

N	9	8	9	10	11	12	11	10	9	**8**
O	8	7	8	9	10	11	10	9	8	9
I	7	6	7	8	9	10	9	8	9	10
T	6	5	6	7	8	9	8	9	10	11
N	5	4	5	6	7	8	9	10	11	10
E	4	3	4	5	6	7	8	9	10	9
T	3	4	5	6	7	8	7	8	9	8
N	2	3	4	5	6	7	8	7	8	7
I	1	2	3	4	5	6	7	6	7	8
#	**0**	1	2	3	4	5	6	7	8	9
	#	E	X	E	C	U	T	I	O	N

动态规划算法(Cormen et al.)

- 最优子结构: 最优的问题解决方案中包括子解决方案，及子问题的最优解决方案（两点最短路径问题最典型的解法就是动态规划算法）

- 重叠的子解决方案Overlapping subsolutions: 子解决方案中有重叠，如果采用暴力计算方法(穷举法)，子解决方案将会被反复计算，从而使得计算开销很大.

- 编辑距离计算中的子问题: 两个前缀之间的编辑距离计算

带权重的编辑距离

- 思路： 对不同的字符进行操作时权重不同
- 希望能更敏锐地捕捉到键盘输入的错误, e.g., *m* 更可能被输成 *n* 而不是 *q*

- 因此，将 *m* 替换为 *n* 的编辑距离将高于替换为*q*的距离
- 也就是输入的操作代价矩阵是一个带权重的矩阵
- 对上述动态规划算法进行修改便可以处理权重计算

利用编辑距离进行拼写校正

- 给定查询词，穷举词汇表中和该查询的编辑距离(或带权重的编辑聚类)低于某个预定值的所有单词
- 求上述结果和给定的某个“正确”词表之间的交集
- 将交集结果推荐给用户
- 代价很大，实际当中往往通过启发式策略提高查找效率(如：保证两者之间具有较长公共子串)

提纲

① 上一讲回顾

② 词典

③ 通配查询

④ 编辑距离

⑤ 拼写校正

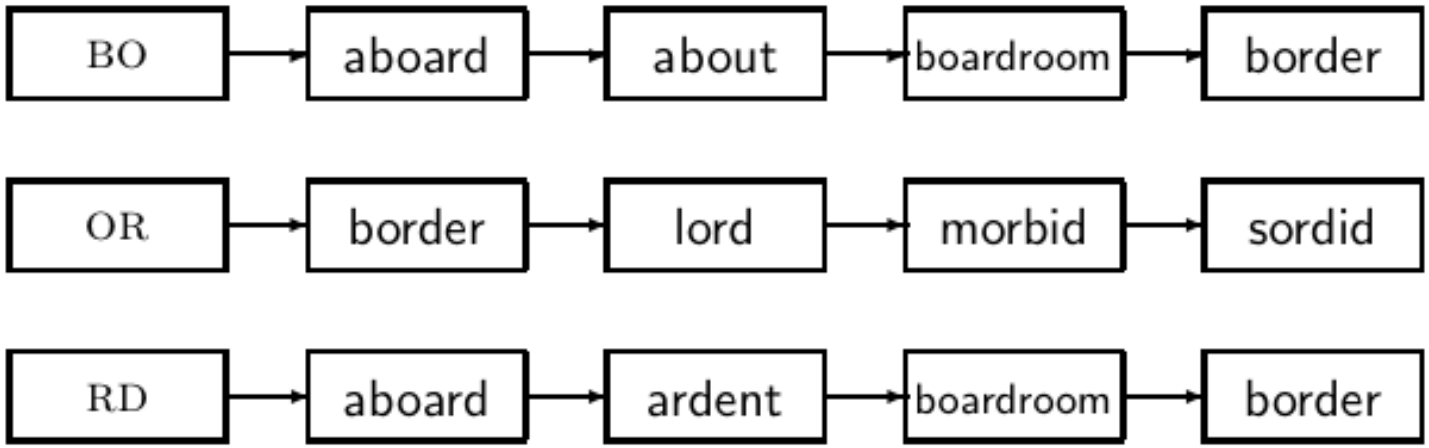
拼写校正

- 刚才已经介绍如何利用编辑距离进行词独立方式下的拼写校正
- 另一种方法： *k*-gram索引
- 上下文敏感的拼写校正
- 拼写校正中的一般问题

基于*k*-gram索引的拼写校正

- 列举查询词项中的所有*k*-gram
  - 例子：采用2-gram索引, 错误拼写的单词为bordroom
  - 2-gram: *bo, or, rd, dr, ro, oo, om*
- 利用 *k*-gram索引返回和能够匹配很多查询*k*-gram的正确单词
- 匹配程度(数目或者指标)上可以事先设定阈值
- E.g., 比如最多只有 3 个*k*-gram不同

2-gram索引示意图


BO
aboard
about
boardroom
border
OR
border
lord
morbid
sordid
RD
aboard
ardent
boardroom
border

上下文敏感的拼写校正

- 例子: *an asteroid that fell form the sky*
- 如何对*form*纠错？
- 一种方法: 基于命中数(hit-based)的拼写校正
  - 对于每个查询词项返回 相近的“正确” 词项
  - *flew form munich: flea ->flew, from -> form, munch ->munich*
  - 组合所有可能
  - 搜索 *“flea form munich”*
  - 搜索 *“flew from munich”*
  - 搜索 *“flew form munch”*
  - 正确查询 *“flew from munich”* 会有最高的结果命中数

上下文敏感的拼写校正

- 刚才提到的基于命中数的算法效率不高
- 一种更高效的做法是： 从查询库(比如历史查询)中搜索而不是从文档库中搜索

拼写校正中的一般问题

- 用户交互界面问题
  - 全自动 vs. 推荐式校正方法(Did you mean…?)
  - 推荐式校正方法通常只给出一个建议
  - 如果有多个可能的正确拼写怎么办？
  - 平衡: 交互界面的简洁性 vs. 强大性
- 开销问题
  - 拼写校正的开销很大
  - 避免对所有查询都运行拼写校正模块
  - 只对返回结果很少的查询运行拼写校正模块
  - 猜测: 主流搜索引擎的拼写校正模块非常高效，有能力对每个查询进行拼写校正

本讲小结

- 词典的数据结构：访问效率和支持查找的方式
  - 哈希表 vs. 树结构
- 容错式检索(Tolerant retrieval):
  - 通配查询：包含通配符*的查询
    - 轮排索引 vs. k-gram索引
  - 拼写校正：
    - 编辑距离 vs. k-gram相似度
    - 词独立校正法 vs. 上下文敏感校正法