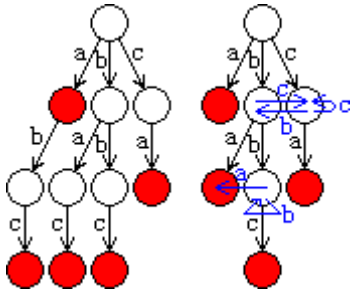
Trie 图的构建、活用与改进

Maigo 2006.1.14

我们知道 **trie 树**（也叫**字母树**）这种数据结构。它是词典的一种存储方式。词典中的每一个单词在 trie 树中表现为一条从根结点出发的路径，路径中边上的字母连起来就形成对应的单词。图 1 就是一棵 trie 树，其中含有 a,abc,bac,bbc,ca 五个单词。

利用 trie 树可以对词典中的单词进行一些适合用树这种数据结构进行的操作，如求两个单词的公共前缀长度（在树中表现为求两个单词对应结点的最近公共祖先）。其实，如果把 trie 树加以改造，多连一些边，形成的 **trie 图**在解决多模式串匹配问题上会发挥奇效。



左：图 1，一棵含有五个单词的 trie 树。红色表示单词终止的位置。

右：图 2，由图 1 的 trie 树改造成的 trie 图。红色表示危险结点，白色表示真安全结点，蓝色表示新加的边。为简单起见，危险结点以下的结点及与之关联的边没有画出。

一、Trie 图的构建

我们通过一个例题来探究 trie 图的构建方法。

【例 1】不良单词探测器

【题目描述】给出一个词典，其中的单词为不良单词。单词均为小写字母。再给出一段文本，文本的每一行也由小写字母构成。判断文本中是否含有任何不良单词。例如，若 rob 是不良单词，那么文本 problem 含有不良单词。

【输入】第一行为一个整数 n，表示不良单词的个数。接下来 n 行是词典。下面一行为一个整数 m，表示文本的行数。接下来 m 行是文本。

【输出】如果文本包含不良单词，输出一行“Yes”，否则输出一行“No”。

【样例输入】

```
1
rob
1
internetproblemsolvingcontest
```

【样例输出】

```
Yes
```

【备注】因本题只是用来讨论 trie 图的构建方法，故未给出数据范围。

【分析】判断文本是否包含不良单词可以一行一行地判断。而判断长为 L 的一行文本 s 是否含有不良单词可以这样进行：让 i 从 1 变化到 L，依次判断 s 的前 i 个字符构成的字符串是否以不良单词结尾。

然而，我们希望在判断 s 的前 k 个字符时，能够利用前 k-1 个字符的结果，即这两个状态间可以方便地进行转移。注意到 trie 树中的边正如一个个“方向标”，因此我们有了一个美好的设想：从根结点出发，沿着标有 s[1]的边走一步，再沿标有 s[2]的边走一步，一直这样走下去！

现在有了一个问题：如果从当前走到的结点出发，没有需要走的边，该怎么办？只要“创造”一条这样的边即可。那么这条边应该指向哪个结点呢？如果同样“创造”一个结点，那是毫无意义的。解决这个问题，要从我们“沿边走”的动机谈起。

我们之所以“沿边走”，是因为我们把结点看成了状态，把边看成了状态间转移的途径。要确定新加的边应连到哪个结点，就需要找我们想走到但去不存在的那个结点与已有的哪个结点是等价的。那么“等价”的标准是什么呢？我们先来解决另一个问题：

定义 trie 树中从根结点到某个结点的路径上的边上的字符连起来形成的字符串为这个结点的**路径字符串**。如果一个结点的路径字符串以不良单词结尾，那么称这个结点为**危险结点**，否则称之为**安全结点**。那么如何判断某个结点是否危险呢？

显然根结点是安全结点。对于一个非根结点，它是危险结点的**充要条件**是：它的路径字符串本身就是一个不良单词，或者它的路径字符串的后缀（一个字符串去掉第一个字符后剩下的部分叫做它的**后缀**）对应的结点（一个**字符串对应的结点**是指在 trie 图中从根出发，依次沿该字符串的每个字符走一步所达到的结点）是危险结点。如果称一个结点的路径字符串的后缀对应的结点为它的**后缀结点**，那么如何求任一结点的后缀结点呢？

根结点的后缀结点是它本身。处于 trie 树第二层的结点的后缀结点也是根结点。对于再往下的某个结点，设它的路径字符串的最后一个字符为 c，那么这个结点的后缀为从它在 trie 树中父结点的后缀结点出发，沿标有 c 的边走一步后到达的结点。（下文中称从 x 结点出发，沿标有字符 c 的边走一步到达的结点为 **x 的 c 孩子**）

那么，如果它的父结点的后缀结点 w 没有 c 孩子怎么办呢？到此，我们看到两个问题已经合而为一了。我们假设 w 有这样一个 c 孩子（记作 x），并且从 x 出发又繁衍出无数的子子孙孙。我们来判断以 x 为根的子树中的结点的危险性。显然 x 本身的路径字符串不是不良单词，且它的子孙的路径字符串也不是不良单词。因此以 x 为根的子树中任一结点 y 的危险性与 y 的后缀结点的危险性相同（回忆一下一个非根结点是危险结点的充要条件）。这也就是说，以 x 为根的子树与以 x 的后缀结点为根的子树是一模一样的。因此，我们把需要新建的从 w 指向 x 的边直接指向 x 的后缀结点，即 w 结点的后缀结点的 c 孩子即可。简言之，由于 x 本身的路径字符串既不是不良单词，又不是某个不良单词开头的一部分，所以它的首字母便没有用了！在这种情况下，x 结点就等价于它的后缀结点。

由此我们可以把 trie 树改造成一个有向图：

按层次遍历 trie 树，求出每个结点的危险性和后缀结点，并补齐由它出发的边。危险性与后缀结点的求法在【分析】部分第 6、7 自然段已有说明；若当前结点为根结点，则补充的边应指向本身，否则若 x 没有 c 孩子，则新建一条这样的边，指向 x 的后缀结点的 c 孩子。

处理某个结点的过程中需要用到深度比它小的结点的后缀结点及各个孩子。由于我们按层次遍历 trie 树，这些信息都已求得。

这样由 trie 树改造成的有向图就叫做 **trie 图**。图 2 就是由图 1 的 trie 树改造成的 trie 图。我们美好的设想终于变成了现实。由根结点出发，按照文本中的字符一步步走下去。若走到一个危险结点，则发现了一个不良单词；若一直没走到危险结点，则文本不含不良单词。

本题的算法还可稍加优化。把安全结点分为两类：如果在 trie 树中由根结点到某个安全结点的路径上没有危险结点，那么称这个安全结点为**真安全结点**，否则称之为**假安全结点**。由于新建的边的终点的深度不会大于起点的深度，因此要到达一个假安全结点，必须经过一

个危险结点。而在本题中，一旦到达一个危险结点，程序就会停止，因此假安全结点是没有用的，也就是说，在本题 trie 图的构建过程中，若发现一个危险结点，那么它及它的子孙的属性都不必计算了。

如果用 L_1、L_2 分别表示不良单词和文本的总长度，用 a 表示字符集中字符的个数，那么 trie 图的时间复杂度为 $O(L_1a+L_2)$，空间复杂度为 $O(L_1a)$。

二、Trie 图的活用

在上面的例题中，我们在 trie 图中记录了每个结点的危险性、后缀结点，并通过按层次遍历得到了图中结点的一个 BFS 序。其实，trie 图中可以记录的信息不止这些；得到的 BFS 序也并不是毫无用处。

【例 2】字谜（题目来源：SPOJ WPUZZLES）

【题目描述】给定一个 L 行 C 列的、由大写字母构成的矩阵，以及 W 个单词。每个单词可在矩阵中的任何位置朝着任何方向出现，且仅出现一次。编程找出每个单词的首字母在矩阵中的位置，以及单词的朝向。

【输入（标准输入）】第一行为一个整数 T，表示数据的组数。下面有 T 组数据。每组数据中：

第 1 行为三个不超过 1000 的整数 L、C、W。

下面 L 行，每行 C 个大写字母，表示矩阵。

下面 W 行，每行一个单词。

【输出（标准输出）】对每组数据，输出 W 行，每行为两个整数和一个字母，之间用一个空格隔开。第 i 行的两个整数表示第 i 个单词首字母的行号和列号（从上至下依次为第 0 至 L-1 行，从左往右依次为第 0 至 C-1 列）；字母表示单词的朝向，对应关系如下：

字母	A	B	C	D	E	F	G	H
朝向	上	右上	右	右下	下	左下	左	左上

相邻两组数据的输出之间用一个空行隔开。

【样例输入】

```
1
4 5 4
MAIGO
QKRPT
AREMO
WERTY
AKI
MAIGO
ARM
ARMY
```

【样例输出】

```
2 0 B
0 0 C
0 1 D
0 1 D
```

【分析】本题中多模式匹配的模型是显而易见的。由于要求的是每个单词首字母的位置，我

们在建 trie 图时，把每个单词都反过来，如单词 MAIGO 变成 OGIAM。对每个方向的每一串字母进行一次多模式匹配，就可以找到所有的单词了。

在本题的 trie 图中，仅仅记录每个结点的危险性是不够的，还要记下每个结点的危险性是由哪个单词引起的。我们定义危险结点 x 的**危险源**：若 x 的路径字符串本身就是不良单词，那么它的危险源就是该单词；否则 x 的危险源就是它后缀结点的危险源。每个结点的危险源可以在后缀结点的过程中求出。

那么，是不是每走到一个危险结点，便记下危险源的位置及朝向就可以了呢？不是的。比如在样例中，当我们沿着左上方向扫描(3,4)-(0,1)这个字符串（YMRA），到达字母 A 时，由于该结点的危险源是 YMRA，我们便记下了 ARMY 的位置和朝向。但同时我们就把单词 ARM 漏掉了。正确的做法是，当遇到一个危险结点时，记下它的危险源的位置和朝向，同时继续检查危险源对应结点的后缀结点，直到到达一个安全结点为止。

【注】本题的字符集虽然只有 26 个字母，但 trie 图中结点的数目可能达到 1,000,000，内存复杂度太高。这个问题留待第三部分解决。

通过《字谜》一题我们学会了如何在 trie 图中记下更多的信息。下面简述一下对 BFS 序的应用。

在做多模式匹配时，有时仅仅找到不良单词是不够的，还要统计出每个单词出现的次数。进行这项工作时，我们并不需要判断每个结点的危险性，而只需累加经过每个结点的次数。但这时有单词对应的结点的访问次数并不就是这个单词出现的次数，比如在图 2 中，单词 a 出现时光标完全可能在结点 ba 上。因此我们**自底向上**地把每个结点的访问次数加到它的后缀结点上。这样处理之后，有单词对应的结点的访问次数就代表这个单词出现的次数了。

上面介绍了 trie 图的一些初步活用。但如果仅仅用 trie 图来做多模式匹配，那就太大材小用了。下面再通过几个例题来说明 trie 图的更灵活的应用。

从例 1 可以看到，危险结点在图中往往是一些障碍，在许多用到 trie 图的问题中，有用的结点只有真安全结点。我们把 trie 图中的真安全结点以及它们之间的边构成的子图叫做**安全图**。

【例 3】病毒（题目来源：POI #7）

【题目描述】已知某些特定的 01 串是病毒的特征代码。如果一个 01 串不含有任何病毒特征代码，则称它为一段安全代码。给定病毒特征库，判断是否存在无限长的安全代码。

【输入（文件 wir.in）】第一行为一个整数 n，表示病毒特征代码的条数。下面 n 行，每行一段病毒特征代码。所有代码长度之和不超过 30000。

【输出（文件 wir.out）】若存在无限长的安全代码，输出一行“TAK”，否则输出一行“NIE”。

【样例输入】

```
3
01
11
00000
```

【样例输出】

```
NIE
```

【分析】“无限长”的安全代码是什么意思呢？就是说从根结点出发，在安全图中可以走无限步。“无限步”又是什么意思呢？就是说安全图中有环。因此我们建立一个 trie 图并对其安全图进行拓扑排序，若成功，则安全图无环，输出“NIE”，否则输出“TAK”。

【例 4】Censored!（题目来源：Ural 1158）

【题目描述】已知一个由 n(1<=n<=50)个字符组成的字符集及 p(0<=p<=10)个不良单词（长度均不超过 10），求长度为 m(1<=m<=50)且不含不良单词的字符串的数目。

【输入（标准输入）】第一行为三个整数 n,m,p。第二行为 n 个字符，表示字符集。下面 p 行，每行一个不良单词。

【输出（标准输出）】一个整数，表示长度为 m 且不含不良单词的字符串的数目。

【样例输入】

```
3 3 3
QWE
QQ
WEE
Q
```

【样例输出】

```
7
```

【分析】求长度为 m 且不含不良单词的字符串的数目，就是求在安全图中从根结点出发走 m 步有多少种走法。用 count[step,x]表示从根结点出发走 step 步到结点 x 的走法数，则容易写出下面的伪代码：

```
fillchar(count,sizeof(count),0);
count[0,根]:=1;
for step:=1 to m do
for 安全图中每条边(i,j) do
  inc(count[step,j],count[step-1,i]);
ans:=0;
for 安全图中每个结点 x do
  inc(ans,count[m,x]);
```

显然，本题还需要用高精度。

我们看到，trie 图的安全图上还是大有文章可做的。Trie 图（或其安全图）作为一个有向图，它具有一般有向图具有的性质，因此在它上面可以进行拓扑排序。同样，它的有向性也可以成为动态规划划分阶段的依据。

三、Trie 图的改进

Trie 图的空间复杂度是比较高的，当 trie 图中结点个数较多或字符集较大时，内存根本无法承受。下面探讨这个问题的解决方法。

【例 5】不良单词过滤器（题目来源：Ural 1269）

【题目描述】给出一个词典，其中的单词为不良单词。再给出一段文本，文本的每一行可能包含除 chr(0),chr(10),chr(13)外的任何字符。若文本中有不良单词，找出文本中不良单词第一次出现的位置，若没有，输出一行“Passed”。

【输入（标准输入）】第一行为一个整数 n(1<=n<=10000)，表示不良单词的个数。接下来 n 行是词典。词典的大小不超过 100KB，每个不良单词的长度不超过 10000。下面一行为一个整数 m，表示文本的行数。接下来 m 行是文本。文本的大小不超过 900KB。

【输出（标准输出）】若文本中有不良单词，输出一行两个整数，表示不良单词第一次出现的行和列，用一个空格隔开。若文本中无不良单词，输入一行“Passed”。

【样例输入】

```
2
rob
Problem
1
Internet Problem Solving Contest
```

【样例输出】

```
1 10
```

【注意】样例中“第一次出现”的不良单词是 Problem 而不是 rob，虽然 rob 比 Problem 先结束。

【时间限制】1s。

【内存限制】5000KB。

【分析】乍一看，这道题与例 1 不是一模一样的吗？其实不然。与例 1 相比，这道题的字符集大得多，如果直接建 trie 图，从每个结点出发要建 253 条边，而结点数最多为 100000，严重超内存。试想一下，如果要做一个汉字的多模式匹配系统，岂不是要从每个结点出发建几千几万条边呢？所以，为解决此问题，trie 图的改进势在必行。

我们看到，在本题中，算法的瓶颈在于从每个结点出发的边数。那么我们自然会想到：一定要存储所有的边吗？答案是否定的。Trie 树中的边自然是要存储的（用左孩子右兄弟表示法），但新建的边则不必存储。如果不存储新建的边，那么如何实现状态间的转移呢？我们用一个函数 child(x,c)来获得结点 x 的 c 孩子。函数内部的程序其实完全是按照加边的原则编写的：如果 x 本来就有 c 孩子，那么就返回这个孩子；如果 x 没有 c 孩子，根据加边的原则，函数应该返回 x 的后缀结点的 c 孩子，也就是令 x 为它的后缀结点，重新执行函数。如果 x 变成了根结点仍然没有 c 孩子，同样根据加边的原则，函数的返回值就应该是根结点本身。

经过这样的处理，算法的空间复杂度由 $O(L_1a)$降到了 $O(L_1)$，对于本题来说是足够低的了。但是，由于 child 函数的执行时间的不确定性，我们对算法的时间复杂度产生了疑问。其实，算法的时间复杂度为 $O(L_1+L_2)$，数量级并没有受到影响，只是增加了一点常数系数。为什么呢？显然，在调用 child(x,c)的时候，只有当 x 没有 c 孩子，需要重复执行 child 函数时运行时间才会增加。我们分别讨论增加的这点时间对建图过程和文本检查过程所需时间的影响：

- ✧ **建图过程**：由于我们并不存储一个结点的所有孩子指针，所以建图的过程其实就是求每个结点的后缀结点的过程。若 b 是 trie 树中 a 结点的一个孩子，那么 b 的后缀结点的深度至多比 a 的后缀结点大 1。如果把 trie 树中某条路径上的结点的后缀结点的深度排成一个数列，那么相邻两项中，后一项减一项的差一定小于等于 1。当后一项减前一项的差小于 1 时，child 函数就会被重复执行。但是，child 函数回溯的次数不会超过 trie 树的深度，所以建图过程的时间复杂度为 $O(L_1)$。
- ✧ **文本检查过程**：把这个过程看成是一个光标在安全图中漫游的过程。因为光标如果往下走，它只能走 1 步，所以若把光标经过的位置的深度也排成一个数列，这个数列与上一段提到的数列具有相同的性质：增长是缓慢的。同理，文本检查过程的时间复杂度为 $O(L_2)$。

综上所述，改进后的 trie 图的时间复杂度为 $O(L_1+L_2)$。无论从时间复杂度上看还是从空间复杂度上看，改进后的算法都明显优于改进前。第二部分中的《字谜》一题到此也得到了

圆满解决。其实，改进后的算法已经就是用于多模式串匹配的改进 KMP 算法了。

对于什么样的题目需要用改进的 trie 图，在此作一下总结：

- ✧ **纯粹的多模式匹配问题**：当题目中的字符集大小有限且较小时，不必用改进的 trie 图，因为内存够用，若进行改进，增加的常数系数可能反而大于字符集的大小 a。如果字符集较大甚至无限（汉字的多模式匹配系统的字符集几乎可以认为是无限的），就必须使用改进的 trie 图。
- ✧ **用到图中每一条边的题目**（如第二部分中提到的拓扑排序、动态规划等）：一般用未改进的 trie 图。在内存十分紧张的情况下，可以采用改进的 trie 图，用时间换空间，但这样做时间复杂度仍为 $O(L_1a+L_2)$，且常数系数较未改进时大。

附：

《病毒》一题的程序： 

《字谜》一题的程序可以在 http://purety.jp/akisame/oi/SPOJ.rar 下载。

《Censored!》和《不良单词过滤器》的程序可以在 http://purety.jp/akisame/oi/URAL.rar 下载。