超酷算法：Levenshtein自动机

2014-12-13 伯乐在线

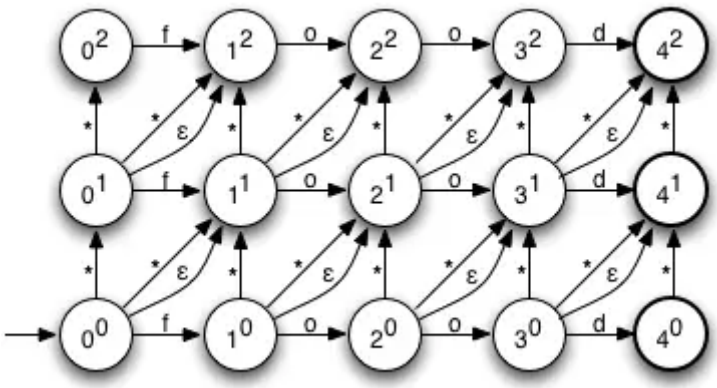
在上一期的超酷算法中，我们聊到了BK树，这是一种非常聪明的索引结构，能够在搜索过程中进行模糊匹配，它基于编辑距离（Levenshtein distance），或者任何其它服从三角不等式的度量标准。今天，我将继续介绍另一种方法，它能够在常规索引中进行模糊匹配搜索，我们将它称之为Levenshtein自动机。

简介

Levenshtein自动机背后的基本理念是：能够构建一个有限状态自动机，准确识别出和某个目标单词相距在给定编辑距离内的所有字符串集合。之后就好办了，我们可以输入任意单词，自动机能够判断这个单词到目标单词的距离是否大于我们在构建时指定的距离，并选择接收或拒绝。更进一步说，根据FSA的自然特性，这项工作可以在O(n)时间内完成，取决于测试字符串的长度。与此相比，标准动态编程距离向量算法需要消耗O(mn)时间，m和n分别是两个输入单词的长度。因此很显然，起码Levenshtein向量机提供了一种更快的方式，供我们针对一个目标单词和最大距离，检查所有的单词，这是一个不错的改进的开端。

当然，如果Levenshtein向量机只有优点，那这篇文章将会很短。我们将会谈到很多，不过我们先来看一下Levenshtein向量机究竟是何物，以及我们如何建立一个Levenshtein自动机。

构建与评价



上图展示了针对单词food的Levenshtein自动机的NFA（译者注：非确定性有限自动机），其最大编辑距离为2。你可以看到，它很普通，构建过程也非常直观。初始状态在左下部分，我们使用ne记法对状态进行命名，n是指到目前为止被处理过的特性的数量，e

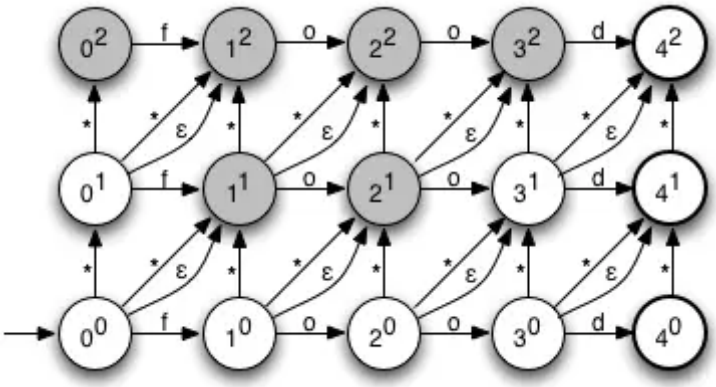
是指错误的个数。水平线表示没有被修改的特性，垂直线表示插入的值，而两条对角线则分别表示交换（标记a*）和删除。

我们来看一下如何通过一个给定的输入单词和最大编辑距离构建一个NFA，由于整个NFA类是非常标准化的，因此我就不赘述其源码了，如果你需要更多细节，请看Gist。以下是基于Python的相关方法：

```
def levenshtein_automata(term, k):
  nfa = NFA((0, 0))
  for i, c in enumerate(term):
    for e in range(k + 1):
      # Correct character
      nfa.add_transition((i, e), c, (i + 1, e))
      if e < k:
        # Deletion
        nfa.add_transition((i, e), NFA.ANY, (i, e + 1))
        # Insertion
        nfa.add_transition((i, e), NFA.EPSILON, (i + 1, e + 1))
        # Substitution
        nfa.add_transition((i, e), NFA.ANY, (i + 1, e + 1))
  for e in range(k + 1):
    if e < k:
      nfa.add_transition((len(term), e), NFA.ANY, (len(term), e + 1))
    nfa.add_final_state((len(term), e))
  return nfa
```

这应该很容易实现，基本上我们用了一种最直接了当的方式构建前图中表示的变换，同时也指出了最终正确的状态集。状态标签是元组，而不是字符串，这与我们前面的描述是一致的。

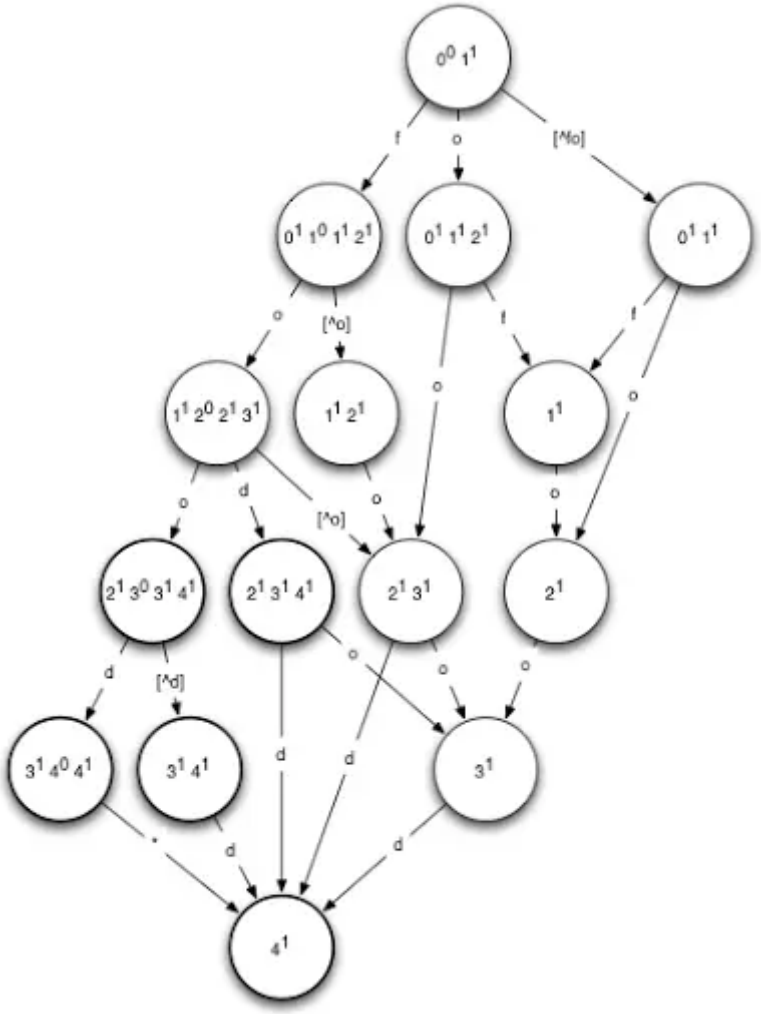
由于这是一个NFA，可以有多个活跃状态，它们表示目前被处理过的字符串的可能解释。举个例子，考虑一下，在处理字符f和x之后的活跃状态：



这表明，在前两个字符f和x一致的情况下，会存在若干可能的变化：一次替换，如fxod；一次插入，如fxood；两次插入，如fxfood；或者一次交换和一次删除，如fxd。同时，这也会引入了一些冗余的情况，如一次删除和一次插入，结果也是fxod。随着越来越多的字符被处理，其中一些可能性会慢慢消失，而另一些可能性会逐渐产生。如果，在处理完整个单词的所有字符后，在当前状态集中存在一个接收状态（bolded state），那么就表明

存在一种方式，能够将通过两次或更少次的变换，将输入单词转化为目标单词，那么我们就可以将该单词视为是有效的。

实际上，要直接评价一个NFA，从计算的角度来讲是极其昂贵的，因为会存在多个活跃状态和epsilon变换（不需要输入符号的变换），所以通常的做法是首先使用powerset构建法将NFA转换为DFA（译者注：确定性有限自动机）。使用这个算法能够构建出一个DFA，使每一个状态都对应原来NFA中的一个活跃状态集。在这里我们不会涉及powerset的细节，因为这有点扯远了。以下是一个例子，展示了在一个容差下，单词food的NFA所对应的DFA：



记住，我们是在一个容差下描述DFA的，因为要找出完全匹配我们提到的NFA所对应的DFA实在是太复杂了！以上DFA能准确接收与单词food相距一个或更少编辑距离的单词集。试试看，选择任意一个单词，通过DFA跟踪它的路径，如果你最终能到达一个接收状态，则这个单词是有效的。

我不会把power构建的源码贴在这里，同样的，如果你感兴趣，可以在GIST里找到。

我们暂时回到执行效率的问题上来，你可能想知道Levenshtein DFA构建的效率怎么样。我们可以在O(kn)时间内构建NFA，k是指编辑距离，n是指目标单词的长度。将其变换为DFA的最坏情况需要O(2^n)时间，所以极端情况下会需要O(2^kn)运行时间！不过情况并没有那么糟糕，有两个原因：首先，Levenshtein自动机并不会充斥着2^n这种最坏情况的DFA构建；其次，一些智慧的计算机科学家已经提出了一些算法，能够在O(n)时间内直接构建出DFA，甚至还有人[SCHULZ2002FAST]完全避开了DFA构建，使用了一种基于表格的评价方法！

索引

既然我们已经证实可以构建一个Levenshtein自动机，并演示了其工作原理，下面我们来看一看如何使用这项技术高效地模糊匹配搜索索引。第一个观点，同时也是很多论文[SCHULZ2002FAST] [MIHOV2004FAST]所采用的方法，就是去观测一本字典，即你所要搜索的记录集，它自身可以被视为是一个DFA。事实上，他们经常被存储为一种字典树或有向非循环字图，这两种结构都可以被视为是DFA的特例。假设字典和标准（Levenshtein自动机）都表示为DFA，之后我们就可以高效地通过这两个DFA，准确地在字典中找到符合标准的单词集，过程非常简单，如下：

```
def intersect(dfa1, dfa2):
  stack = [("", dfa1.start_state, dfa2.start_state)]
  while stack:
    s, state1, state2 = stack.pop()
    for edge in set(dfa1.edges(state1)).intersect(dfa2.edges(state2)):
      state1 = dfa1.next(state1, edge)
      state2 = dfa2.next(state2, edge)
      if state1 and state2:
        s = s + edge
        stack.append((s, state1, state2))
        if dfa1.is_final(state1) and dfa2.is_final(state2):
          yield s
```

好了，我们按照两个DFA共有的边界同时进行遍历，并记录遍历的路径轨迹。只要两个DFA处于最终状态，单词在输出集内，我们就将其输出。

如果你的索引是以DFA（或字典树，或有向非循环字图）的形式存储的话，这非常完美，但遗憾的是许多索引并不是：如果在内存中，它们很可能位于一个排序列表中；如果在磁盘上，它们很可能位于BTree或类似结构中。有没有办法可以让我们修改方案适应这些排序索引，继而继续提供一种速度极快的方法？事实证明是有的。

这里的关键点在于，根据我们目前以DFA表示的标准，我们可以，对于一个不匹配的输入字符串，找到下一个（按字母排序）匹配的字符串。凭直觉来说，这相当容易：我们基于DFA去评估输入字符串，直到我们无法进一步处理为止，比如说没有针对下一个字符的有效变换，之后，我们可以反复遵照字母排序的最小标签的边界，直到到达终态。在这里我

们应用了两个特殊事件：首先，在第一次变换中，我们需要遵照按字母排序的最小标签，同时这些标签要大于在准备步骤中没有有效变换的特性。第二，如果我们达到了一个状态而其没有有效的外边界，那么我们要回溯到之前的状态，并重试。这差不多是解决迷宫问题的一种“循墙”算法，应用在DFA上。

以此举例，参照food(1)的DFA，我们来思考一下输入单词foogle。我们可以有效处理前4个单词，留下状态3141，这里唯一的外边界是d，下一个字符是l，因此我们可以向前回溯一步，到21303141，现在下一个字符是g，有一个外边界f，所以我们接收这个边界，留下接收状态（事实上，和之前的状态是一样的，只不过路径不同），输出单词为fooh，这是在DFA中按字母排序在foogle之后的单词。

以下是python代码，展示了在DFA类上的一个方法。和前面一样，我不会写出整个DFA的样板代码，它们都在这里。

```
def intersect(dfa1, dfa2):
  stack = [("", dfa1.start_state, dfa2.start_state)]
  while stack:
    s, state1, state2 = stack.pop()
    for edge in set(dfa1.edges(state1)).intersect(dfa2.edges(state2)):
      state1 = dfa1.next(state1, edge)
      state2 = dfa2.next(state2, edge)
      if state1 and state2:
        s = s + edge
        stack.append((s, state1, state2))
        if dfa1.is_final(state1) and dfa2.is_final(state2):
          yield s
```

在这个方法的第一部分，我们以常见的方式评价DFA，记录下访问过的状态，这些状态包括它们的路径以及我们尝试寻找遵循它们的边界。之后，假设没有找到一个准确的匹配项，那么就进行一次回溯，尝试去寻找一个可以到达接收状态的最小变换集。关于这个方法的一般性说明，请继续阅读……

同时我们还需要一个工具函数find_next_edge，找出一个状态中按字母排序比指定输入大的最小外边界：

```
def find_next_edge(self, s, x):
    if x is None:
      x = u''
    else:
      x = unichr(ord(x) + 1)
    state_transitions = self.transitions.get(s, {})
    if x in state_transitions or s in self.defaults:
      return x
    labels = sorted(state_transitions.keys())
    pos = bisect.bisect_left(labels, x)
    if pos < len(labels):
      return labels[pos]
    return None
```

经过一些预处理，这可以更高效，打个比方，我们可以对每个字符和第一个大于它的外边界建立一个映射关系，而不是在茫茫大海中进行二进制检索。再强调一次，我会把这些优化工作作为练习题留给读者。

既然我们已经找到了这一过程，那么我们就可以最终描述如何使用这一过程进行索引搜索，算法出人意料的简单：

1. 取得索引中的第一个元素，或者，比索引任意有效字符串更小的一个字符串，将其称之为“当前”字符串。
2. 将“当前”字符串传入我们之前谈到的DFA算法，得到“下一个”字符串。
3. 如果“下一个”字符串和“当前”字符串相等，那么你已经找到了一个匹配，将其输出，再从索引中获取下一个元素作为“当前”元素，重复步骤2。
4. 如果“下一个”字符串和“当前”字符串不相等，那么在你的索引中搜索大于等于“下一个”字符串的第一个字符串，将其作为“当前”元素，重复步骤2。

以下是用Python实现这一过程的代码：

```
def find_all_matches(word, k, lookup_func):
  """Uses lookup_func to find all words within levenshtein distance k of word.

  Args:
    word: The word to look up
    k: Maximum edit distance
    lookup_func: A single argument function that returns the first word in the
      database that is greater than or equal to the input argument.
  Yields:
    Every matching word within levenshtein distance k from the database.
  """
  lev = levenshtein_automata(word, k).to_dfa()
  match = lev.next_valid_string(u'')
  while match:
    next = lookup_func(match)
    if not next:
      return
    if match == next:
      yield match
      next = next + u''
    match = lev.next_valid_string(next)
```

理解这一算法的一种方式是将Levenshtein DFA和索引都视为排序列表，那么以上过程就类似于App引擎中的“拉链合并连接”策略。我们重复地在一侧查找字符串，再跳转到另一侧的合适位置，等等。结果是，我们省去了大量不匹配的索引实体，以及大量不匹配的Levenshtein字符串，节省了枚举它们的工作量。这些描述表明，这一过程有潜力避免去评估所有的索引实体，或所有的候选Levenshtein字符串。

补充说明一下，所有的DFA针对任意字符串都可以找到按字母排序的最小后继，这句话是错误的。比如说，考虑一下DFA中字符串a的后继，识别模式为a+b，答案是没有这样的后继，它必须由无限多的a字符跟随单个b字符构成！不过我们可以基于以上过程做一些简单的修改，比如返回一个字符串，确保它是DFA可以识别的下一个字符串的一个前缀，这能满足我们的需求。由于Levenshtein DFA总是有限的，因此我们总是会得到一个有限长度的后继（当然，除了最后一个字符串），我们把这样的扩展留给读者作为练习题。使用这种方法，会产生一些很有意思的应用程序，比如索引化正则表达式搜索。

测试

首先，我们理论联系实际，定义一个简单的Matcher类，其中实现了一个lookup_func方法，它会被find_all_matches方法调用：

```
class Matcher(object):
  def __init__(self, l):
    self.l = l
    self.probes = 0

  def __call__(self, w):
    self.probes += 1
    pos = bisect.bisect_left(self.l, w)
    if pos < len(self.l):
      return self.l[pos]
    else:
      return None
```

记住，在此我们实现一个可调用的类的唯一理由是：我们想要从程序中提取一些信息，比如探针的个数。通常来说，一个常规或嵌套函数已经足够完美，现在，我们需要一个简单的数据集，让我们加载web2字典：

```
>>> words = [x.strip().lower().decode('utf-8') for x in open('/usr/share/dict/web2')]
>>> words.sort()
>>> len(words)
234936
```

我们也可以使用几个子集测试随着数据规模的变化，会发生什么：

```
>>> words10 = [x for x in words if random.random() <= 0.1]
>>> words100 = [x for x in words if random.random() <= 0.01]
```

这里，我们看到了实践结果：

```
>>> m = Matcher(words)
>>> list(automata.find_all_matches('nice', 1, m))
[u'anice', u'bice', u'dice', u'fice', u'ice', u'mice', u'nace', u'nice',
u'niche', u'nick', u'nide', u'niece', u'nife', u'nile', u'nine', u'niue',
u'pice', u'rice', u'sice', u'tice', u'unice', u'vice', u'wice']
>>> len(_)
23
>>> m.probes
142
```

大赞啊！在拥有235000个单词的字典中找到了针对nice的23个模拟匹配，需要142个探针。注意，如果我们假设一个字母表包含26个字母，那么会有4+26*4+26*5=238个字符串在一个Levenshtein距离内是有效的，因此与详尽的测试相比，我们做出了合理的节省。考虑到有更大的字母表，更长的字符串，或更大的编辑距离，这种节省的效果应该会更明显。如果我们使用不同种类的输入去测试，看一下探针的个数随着单词长度和字典大小的变化情况，可能会更受启发：

String length	Max strings	Small dict	Med dict	Full dict
1	79	47 (59%)	54 (68%)	81 (100%)
2	132	81 (61%)	103 (78%)	129 (97%)
3	185	94 (50%)	120 (64%)	147 (79%)
4	238	94 (39%)	123 (51%)	155 (65%)
5	291	94 (32%)	124 (43%)	161 (55%)

在这个表中，” max strings”表示与输入字符串在编辑距离内的字符串总数；small，med，full dict表示所有三种字典（包含web2字典的1%，10%和100%）所需要的探针个数。所有对应的行，至少在10个字符以内，都需要与第五行差不多的探针个数。我们采用的输入字符串的例子是由单词’abracadabra’的前缀构成的。

我们可以立即看出一些端倪：

1. 对于很短的字符串和很大的字典，探针的个数并没有低很多，即使低一些，和有效字符串的最大个数相比也是小巫见大巫，所以这并没有节省什么。
2. 随着字符串越来越长，探针的个数的增长出人意料的比预期结果慢，结果就是对于10个字符，我们仅仅需要探测821中的161个（大约20%）可能结果。对于一般的单词长度（在web2字典中，97%的单词至少有5个字符长），与朴素的检查每个字符串变化相比，我们已经节省了可观的代价。

3. 虽然样本字典的大小以不同的数量级区分，但是探针的个数增长却不太明显，这是一项令人鼓舞的证据，它表明该方法可以很好的扩展到非常大的索引数量级上。

我们再来看一下根据不同的编辑距离阈值，情况会有何变化，你同样能得到一些启发。下面是相同的表格，最大编辑距离为2：

String length	Max strings	Small dict	Med dict	Full dict
1	2054	413 (20%)	843 (41%)	1531 (75%)
2	10428	486 (5%)	1226 (12%)	2600 (25%)
3	24420	644 (3%)	1643 (7%)	3229 (13%)
4	44030	646 (1.5%)	1676 (4%)	3366 (8%)
5	69258	648 (0.9%)	1676 (2%)	3377 (5%)

前途一片光明：在编辑距离为2的情况下，虽然我们被迫需要加入很多探针，但是与候选字符串的数量相比，仍然是很小的代价。对于一个长度为5、编辑距离为2的单词，需要使用3377个探针，但是比起做69258次（对每一个匹配字符串）或做234936次（对字典里的每个单词），这显然少得多了！

我们来做一个快速比较，对于一个长度为5的字符串，编辑距离为1（与上面的例子一样），一个标准的BK树实现，基于相同的字典，需要检查5858个节点，同时，相同的情况下，我们把编辑距离改为2，则需要检查58928个节点！应当承认，如果结构合理的话，这些节点中很多都应处于相同的磁盘页，但是依然存在惊人的查找数量级的差异。

最后一点：我们在这篇文章中参考的第二篇论文，[MIHOV2004FAST]描述了一个非常棒的结构：一个广义的Levenshtein自动机。这是一种DFA，它能在线性时间内判断，任意一组单词对互相之间的距离是否小于给定的编辑距离。改造一下我们前面的方案，使其能适应这种自动机，这也是我们留给读者的练习。

这篇文章是涉及的完整的源代码都可以在这里找到。

原文出处：blog.notdot.net
译文作者：伯乐在线 - Justin Wu

//////////////////////////

1. 在这个信息爆炸的时代，人们已然被大量、快速并且简短的信息所包围。然而，我们相信：过多“快餐”式的阅读只会令人“虚胖”，缺乏实质的内涵。伯乐在线博客团队正试图以我们微薄的力量，把优秀的原创/译文分享给读者，为“快餐”添加一些“营养”元素。欢迎关注微信号【 jobbole 】

2. 点击“阅读原文”，查看本文的网页版。

Read more