suffix tree にもとづいた n-gram の trie 構造化とその応用

一井 崇* 清田 陽司† 吉田 稔† 中川 裕志†

1 はじめに

用例や言い換えパタン，質問応答といった，文字列で表現される情報をコーパスから得るために，文書検索によって検索語句を含む文書を求め，それらを参照することでその情報要求を満たす手法が多数提案されている。しかしコーパスが大きくなるにつれ，検索結果の文書数が膨大になってしまい，すべての文書を参照することは現実的でなくなってしまった。また 1 文書の区切りも明確でないことが多く，bag-of-words モデルのような，1 文書が意味のよいまとまりになっているとの仮定が必ずしも成り立たないことも多い。

そこで，本研究では，同じ文書に現れることよりも文字や単語列として近接していることをより重要と考え，近接する n 文字 (もしくは単語，形態素などの列) の統計情報として，n-gram の trie 構造化を考える。具体的には，suffix が $n = \infty$ における n-gram と見なせることに注目し，Ukkonen[4] による suffix tree の構築アルゴリズムにもとづいて，n-gram を trie 構造化することを提案する。また，実際に n-gram trie を実装し，その性質の評価とその応用を考察する。

2 suffix tree

suffix tree とは，文字列のすべての suffix，すなわち文字列中のある位置から末尾までの部分文字列を trie 構造化したものである。図 1 に文字列 “cacao” の suffix tree を示す。

Ukkonen は，suffix tree にいくつかの拡張を施すことで，suffix tree を文字列の長さ N にたいして $O(N)$ の時間と記憶容量で構築するアルゴリズムを提案した [4]。その要点は以下の通りである。

path compression 分岐のない辺を 1 つにまとめ，辺文字列を直接記さずに先頭の 1 文字と文字列中の開始位置，終了位置を記す

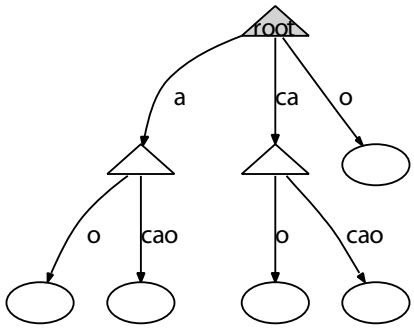

図 1 “cacao” の suffix tree

open leaf 新たに葉ノードを作成する際に，終了位置を ∞ とすることで，その後の終了位置の操作を省く

suffix link 先頭の 1 文字が少ない場所へのリンク (“xabc”→“abc”) を内部ノードに与え，構築時の作業の流れを決定する。

suffix tree はその構造から，ある文字列に後続する文字列の種類が直ちに分かるという性質を持つ。しかしながら，その記憶容量は文字列の大きさに比例するものの 20 倍から 30 倍以上にまで達し，必ずしも扱いやすいものではなかった。また元文字列中での出現位置を記すことで $O(N)$ を達成しているが，同時に元文字列への参照を頻繁に必要とする要因にもなっている。

suffix tree の容量を圧縮する手法はいくつかある [1] が，基本的には $O(N)$ を維持した上で，その比例定数の削減を目指したものである。一方，単語から始まる suffix だけを含む suffix tree を構築するアルゴリズムが提案されており [2]，これは文字列の長さ N とは独立に容量削減が可能になっている。

3 n-gram の trie 構造化

文字列にすべての文字列と異なる特殊文字 (sentinel, ‘$’ と記す) を無限に付加したとき，その suffix は $n = \infty$ における n-gram と解釈できる (図 2)。

* 東京大学 大学院学際情報学府
† 東京大学 情報基盤センター

元文字列	c a c a o $ $ $ $ ...
suffix	c a c a o $ $ $ $ ...
	a c a o $ $ $ $ ...
	c a o $ $ $ $ ...
	a o $ $ $ $ ...
	o $ $ $ $ ...
3-gram	c a c
	a c a
	c a o
	a o $
	o $ $

図 2　suffix(=∞-gram) と 3-gram

このことから，suffix tree と同様な考え方にもとづき，文字列のすべての n-gram，すなわち文字列中のある位置から長さ n の部分文字列を trie 構造化したものを考えることができる。

3.1　n-gram trie の定義

文字列に十分長い sentinel を付加し，そのすべての n-gram(長さ n の部分列) を trie 構造化したものを，n-gram trie と呼ぶ。suffix tree と同様に，以下の拡張を施す。

辺に長さを記す　分岐のない辺を 1 つにまとめ, 辺文字列を直接記さずに先頭の 1 文字とその長さを記す

葉ノードの長さの決定　新たに葉ノードを作成する際に，根からその葉ノードまでの経路に対応する文字列長が n となるような長さとし，その後の終了位置の操作を省く

suffix link の拡張　先頭の 1 文字が少ない場所へのリンク (“xabc”→“abc”) を内部ノードのみならず葉ノードにも与える

頻度情報　各葉ノードは，それが現す n-gram の頻度情報を持つ

図 3 に, 文字列 “cacao“ の $n = 2, 6$ での n-gram trie を示す。楕円で示された葉ノード中の数はその頻度である。$n = 6$ は suffix tree と同様な構造になっているのに対し，$n = 2$ は “c” から始まる葉ノードが頻度 2 となっている。“c, 2” と記された辺の葉ノードからのリンクを辿ることで，この辺の 2 文字目である “a” が “a, 1” と記された辺の 1 文字目として得られる。

特に注意する点として，n-gram trie は一般に用いられる suffix tree と異なり，n-gram の出現位置情報を持たない。これは，n-gram trie はテキストの統計情報であって，コーパスとなるテキストを n-gram にいったん分解し，それを trie 構造化したものだと考えているからである。

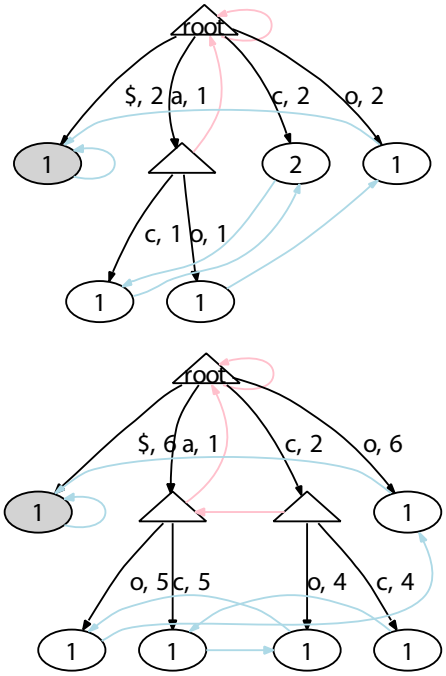

図 3　“cacao” の 2-gram trie, 6-gram trie

辺に記された文字は “(1 文字目の文字，長さ)”

3.2　辺文字列の復元

内部ノードだけでなく葉ノードにも suffix link を適用したことにより，テキストを参照せずに辺文字列を 1 文字目とその長さから復元することができる。ある葉ノードへの辺文字列を得たいときは，その葉ノードから suffix link を順次辿って行き，先頭の 1 文字を連結してゆけばよい。

3.3　n-gram trie の構築アルゴリズム

Ukkonen の suffix tree 構築アルゴリズムと同様な用語を定義する。追加しようとしている文字列が tree 内にすでに存在する場合 (nest している場合), その位置を active suffix とよぶ。このとき，文字列 $S = a_1 a_2 \dots a_i$ に対応する n-gram trie から $S = a_1 a_2 \dots a_i a_{i+1}$ に対応する n-gram trie への拡張操作 update は，アルゴリズム 1 のようになる。

4　実験・評価

固定された n に対し，suffix tree と同様に，入力テキストの大きさに対して線形な時間で n-gram trie が構築された。$n = 5$ では，50Mbyte 程度のテキストに対し，文字単位で 80sec，形態素単位では，分かち書き

Algorithm 1 update

```
次の 1 文字 a_{i+1} を取得し，Queue に追加
while Queue の長さ > 0 do
    if Queue の文字列が nest している then
        if Queue の長さ = n then
            対応する葉ノードの頻度を 1 増やす
            Queue の先頭を削る
            suffix link を辿り，次の active suffix に進む
        end if
        break
    else
        新たに葉ノードを作る
        必要に応じて suffix link を張る
        Queue の先頭を削る
        suffix link を辿り，次の active suffix に進む
    end if
end while
```

の処理時間を含めて 220sec 程度で構築が完了した。

また，n を変化させて n-gram trie の構築速度と node 数の変化を調べた。毎日新聞コーパスの文字単位 (`mainichi/char`), 形態素単位 (`mainichi/token`), およびロイターコーパスの文字単位 (`reuter/char`)，単語単位 (`reuter/token`) に対する結果を図 4, 5 に示した。$n = 5$ で node 数が suffix tree のおよそ 60 ～ 80% 程度となり，その記憶容量を削減できることが分かった。また，$n = 10$ で速度，node 数ともに変化が安定し，$n = 30$ で suffix tree($n = \infty$) とほぼ同等となることが分かった。これは，10 文字 (または 10 単語) 以上共通する列がほとんど現れなかったことを示している。

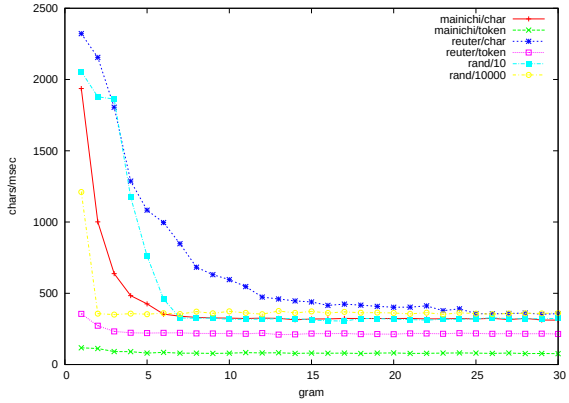

図 4　n を変化させたときの構築速度

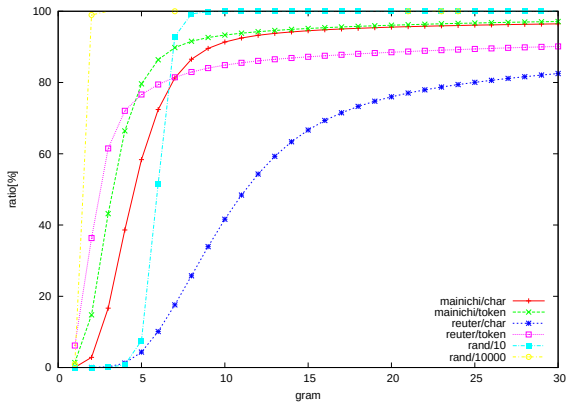

図 5　n を変化させたときの node 数 (suffix tree での node 数を 100% とする)

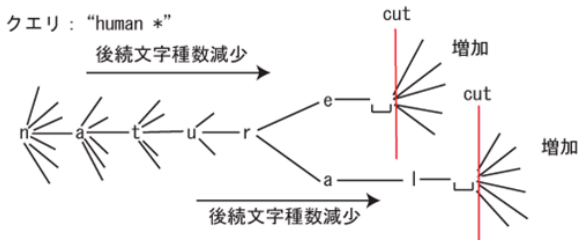

図 6　Kiwi アルゴリズムで用例を切り出す様子

5　情報検索への応用

trie 構造に対して適用可能な応用例として，用例検索システム Kiwi[3] のアルゴリズムをもとに，簡単な検索システムを実装した。Kiwi アルゴリズムでは，検索語句から始まる部分木に対し，後続文字種類数が減少から増加に転ずるところを用例の候補として切り出す (図 6)。コーパスとなるテキストの n-gram trie を構築しておけば，trie 中を探索し子ノード数が増大する部分を切り出すことで，コーパスから用例を検索することができる。

5.1　文字単位の n-gram trie からの用例検索

実験結果から suffix tree と比べて node 数が 60% 程度となった $n = 5$ を用い，毎日新聞コーパス 50Mbyte に対して文字単位で n-gram trie を構築し，Kiwi アルゴリズムを適用する用例検索システムを実装した。検索結果数が数百件程度の場合，検索にかかる時間は 100msec 以下であり，非常に高速に動作した。しかしながら，5 文字以上の用例を検索できない，という問題が発生した。

5.2 文字単位と単語単位との組み合わせ

n-gram を構成する単位には文字と単語 (形態素) の 2 通りが考えられ，その n-gram trie について，それぞれ次のような長所・短所があった。

文字単位 検索語句の自由度は高いが，記憶容量が大きくなる。

単語単位 n が小さくても長い文字列を表現でき，文字単位と比べ相対的に記憶容量が小さくなる。また検索結果は単語単位に整ったものとなる。ただし，検索語句は単語単位に限られる。

そこで，比較的小さな n に対する文字単位の n-gram trie T_1 と，必要な長さを満たす単語単位の n-gram trie T_2 を用意して，T_1 での検索結果それぞれを T_2 に対して検索し，結果をマージすることを考えた。これは，Kiwi アルゴリズムの性質から，文字単位での検索結果が検索語句を単語単位に拡張したものとして得られると期待できるからである。同時に，複数回の検索が自動的に繰り返されるため，その実用性は 1 回の検索が非常に高速であることに強く依存する。

毎日新聞コーパスに対し，文字単位は $n = 4$ で，単語 (形態素) 単位は $n = 5$ で検索システムを実装し，検索語「国」での検索結果について $n = 30$ における文字単位と単語単位でのシステムと比較した (表 1，2)。

表 1　文字・形態素の組み合わせで用意した検索システム

種類	文字単位	形態素単位	組み合わせ
gram 数	30	30	4, 5
消費メモリ	829Mbytes	562Mbytes	785Mbytes
構築時間	41,626msec	92,659msec	88,393msec
検索時間	89msec	8msec	118msec
検索結果	3,333 件	121 件	460 件

表 2　「国」での検索結果例

文字単位	形態素単位	組み合わせ
‘国民の’	‘国首脳会議）’	‘国民の’
‘国家公安委’	‘国首脳会議（’	‘国家公安委員’
‘国家公安委員’	‘国への’	‘国家公安’
‘国から’	‘国機構（ＯＰＥＣ）’	‘国会議員’
‘国会議員’	‘国と阪神高速道路’	‘国家公安委員会’
‘国政府’	‘国と国の関係」と’	‘国民に’
‘国家公安委員会’	‘国の中で’	‘国会の’
‘国民に’	‘国の関係」と’	‘国首脳’
‘国会の’	‘国と公団は’	‘国家公安委員長’
‘国首脳会議 ’	‘国別対抗戦’	‘国首脳会議）’

組み合わせによるシステムは文字・形態素単位の 2 つの n-gram trie を用意する必要があるものの，$n = 30$ で文字単位の n-gram trie を用いるよりも若干消費メモリ量は少なくて済んだ。さらに，検索結果に以下の特徴が見られた。

文字単位との比較 ‘国政府’ のような，単語の途中から始まっていると考えられる候補が含まれない。

形態素単位との比較 ‘国民’，‘国家’ といった，‘国’ から始まる単語が検索結果に含まれて入る。

6 おわりに

suffix を ∞-gram と解釈できることから，Ukkonen の suffix tree 構築アルゴリズムを元に，有限の n にたいして n-gram trie を構築する手順を示した。n を調整することで，suffix tree に比べて 60 ～ 80% 程度に node 数を削減することができた。また，出現位置情報を持たず，テキストを参照することなしに構築・検索が可能であることを示した。すなわち，テキストから独立した，近接文字列の統計情報として，n-gram trie を提案した。

さらにその応用例として，Kiwi アルゴリズムによる用例検索システムを実装した。n を任意に調整できること，および検索時間が高速であることを活かし，文字単位と単語単位でのシステムを組み合わせることで，両者の長所を活かした検索システムが実装できることを示した。

参考文献

[1] Arne Andersson and Stefan Nilsson. Efficient implementation of suffix trees. *Softw. Pract. Exper.*, Vol. 25, No. 2, pp. 129–141, 1995.

[2] Andersson, Larsson, and Swanson. Suffix trees on words. In *CPM: 7th Symposium on Combinatorial Pattern Matching*, 1996.

[3] Kumiko Tanaka-Ishii and Hiroshi Nakagawa. A multilingual usage consultation tool based on internet searching: more than a search engine, less than QA. In *WWW Conference*, pp. 363–371, 2005.

[4] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, Vol. 14, No. 3, pp. 249–260, 1995.