ダブルアレイによる高速かつコンパクトな
統計的言語モデルの実装手法

2016 年 3 月

乗松 潤矢

ダブルアレイによる高速かつコンパクトな
統計的言語モデルの実装手法

乗松 潤矢

システム情報工学研究科
筑波大学

2016 年 3 月

目次

図目次

表目次

アルゴリズム目次

第 1 章

はじめに

統計的言語モデルは、統計的機械翻訳において最も重要なコンポーネントの一つである。統計的言語モデルの性質として、学習に使うデータを増加させれば増加させるほどシステム全体の性能が向上することが知られている (Brants et al. 2007)。したがって、システム全体の性能を向上させるためには、できるだけ大きな学習データから得られた言語モデルを利用すべきである。

従来、大規模な言語モデルの推定は大量の計算資源を利用するため、大多数のユーザーには難しい問題だったが、Heafield et al. (2013) の手法により手軽に大規模な言語モデルを推定可能となった。この手法はメモリの少ない計算機でも大規模なモデルパラメータの推定を可能にする点で画期的であり、これによって一般的なユーザーでも大規模な言語モデルを利用できるようになった。

言語モデルのパラメータ推定は少ないメモリでも推定可能だが、推定された言語モデルを利用する際には大量のメモリを必要とする。統計的機械翻訳においては、言語モデルへの大量のクエリが発生するため、言語モデルを HDD などのメモリ外へ配置する方法は処理速度を著しく低下させるためである。したがって、大規模な言語モデルを利用した翻訳システムを動作させるためには大量のメモリを必要とする。

上記の理由から、一般的なユーザーにとって翻訳性能はシステムを動作させる計算機が搭載するメモリの量で制限を受ける。すなわち、ほとんどのユーザーは利用可能な言語資源を活かしきれていない。また、先行研究においては、メモリ使用量と実行速度はトレードオフの関係にあることが知られており、コンパクトなだけでなく高速にアクセスできる言語モデルの実装が求められる。

今後、計算機に搭載されるメモリ容量が増加したとしても、利用可能なコーパスの量も同時に増加していくと考えられる。したがって、メモリ使用量の問題は依然として問題となり続けると考えられる。

本研究では、高速かつコンパクトな言語モデルを実現するため、ダブルアレイ (Aoe 1989) に着目した。ダブルアレイは、Trie と呼ばれる木構造を実現する手法の一つであり、本研究で必要な高速さとコンパクトさを兼ね揃えている。もし、言語モデルにダブルアレイを適応し、ダブルアレイの高速・コンパクトという特徴を活かすことができれば、従来の言語モデル実装を凌ぐ性能を発揮できる可能性がある。

本研究では、ダブルアレイをベースとした統計的言語モデルを提案する。言語モデルに Trie を用いることは従来からなされているが、同じ言語モデルを Trie で表現するにも何通りかのバリエーションが存在する。本研究では特に、Backward Suffix Tree と Reverse

Trie に着目し、それぞれについてダブルアレイを適応する方法を提案する。単純にダブルアレイを使うだけでは、メモリ中に未使用領域が発生し、メモリ中に無駄な領域が発生する。本研究では言語モデルの性質とダブルアレイの未使用領域を組み合わせることで、コンパクトに格納する方法を提案する。

ダブルアレイを言語モデルに適応するだけでなく、チューニングの方法を 2 つ提案する。一つは単語 ID のチューニング、もう一つは Trie の葉ノードの削減法である。

単語 ID をチューニングすることで、ダブルアレイのサイズをよりコンパクトにできる。ダブルアレイで Trie を表現する際、ノード数が同じでも Trie の性質によってダブルアレイのサイズが変わることがある。そこで、よりコンパクトになりやすいよう単語 ID をチューニングする。

Trie の葉ノードを削減する方法は、言語モデルの特性上、モデルパラメータを保持しなくても良い場合と Trie の形がうまく整合することに着目し、Trie からノードを削減する手法である。ノード数の削減はモデルのメモリ使用量の削減に直結するため、よりコンパクトな言語モデルを実現した。この手法は、Reverse Trie にのみ利用可能である。

実験では、3 通りの場合において、従来の言語モデル実装と本研究の実装を比較した。一つ目の実験は、先行研究とモデルの構築時間を比較した。この実験では、提案手法のモデル構築がどの程度の時間を要するのかを確認し、また先行研究との比較を行った。二つ目は、パープレキシティ計算である。この実験では、言語モデルだけの純粋なクエリ時間を測定することを目指した。二つ目の実験は、統計的翻訳のコンポーネントとして言語モデルを利用した場合の処理時間を計測した。実際のアプリケーションに組み込んだ場合での従来手法との比較が目的である。

実験の結果、構築時間は先行研究と比べて遅いことがわかった。パープレキシティ計算においては、Reverse Trie に基づく言語モデルが先行研究の持つ速度とメモリ使用量のトレードオフを改善したことを確認した。また、翻訳実験においても Reverse Trie に基づく言語モデルが先行研究の持つ速度とメモリ使用量のトレードオフで改善していることを示した。

本論文ではまず、研究の背景を述べる (第 2章)。ここでは、統計的機械翻訳のフレームワークやそのコンポーネントとして利用される言語モデルについて概説し、さらに本研究で利用するダブルアレイについても述べる。次に、関連研究を述べる (第 3章) を述べ、本研究で取り扱う問題を既存の研究がどのように取り扱ってきたか概説する。続いて提案手法を述べた (第 4章) 後、実験を行う (第 5章)。最後に本研究を総括して結論を述べる (第 6章)。

第 2 章

研究の背景

2.1　統計的機械翻訳

統計的機械翻訳 (Brown et al. 1993) は、大量に与えられた翻訳文対を学習データとして、統計処理により翻訳ルールを獲得する手法である。統計的機械翻訳において、翻訳元の言語を元言語、翻訳先の言語を目的言語という。元言語の文を $\boldsymbol{f}$ とすると、統計的機械翻訳は以下の式で表される。

$$\hat{\boldsymbol{e}} = \arg\max_{\boldsymbol{e}} P\left(\boldsymbol{e} \mid \boldsymbol{f}\right). \tag{2.1}$$

ここで、$\boldsymbol{e}$ は翻訳途中に生成される翻訳候補の文であり、$\hat{\boldsymbol{e}}$ は最終的に翻訳結果として得られる目的言語文である。統計的機械翻訳では、元言語文の翻訳確率を最大化する目的言語文を求める。

式 2.1に対してベイズの定理を適用することにより、以下の式を得る。

$$\hat{\boldsymbol{e}} = \arg\max_{\boldsymbol{e}} P\left(\boldsymbol{e}\right) P\left(\boldsymbol{f} \mid \boldsymbol{e}\right).$$

この式で、$P\left(\boldsymbol{e}\right)$ は言語モデルと呼ばれ、言語としての自然さを確率的に評価する。$P\left(\boldsymbol{f} \mid \boldsymbol{e}\right)$ は翻訳モデルと呼ばれ、主に訳語としての正しさを確率的に評価する。言語モデルと翻訳モデルの各モデルで評価した値を掛けあわせることで、正確で自然な翻訳を実現する。

2.2　翻訳モデル

2.2.1　フレーズモデル

フレーズモデル (Koehn, Och, and Marcu 2003) は、統計的機械翻訳において最も標準的に用いられている翻訳モデルの一つである。このモデルは主に仏英翻訳など、語順が似ている言語間で高い性能を締めすることが知られている。

フレーズモデルでは、与えられた元言語文 $\boldsymbol{f}$ と目的言語文 $\boldsymbol{e}$ のペアに対して、以下の 2 つの仮定をする。

- 元言語文と目的言語文はそれぞれ、フレーズの組み合わせに分解できる。
- 元言語文のフレーズと目的言語文のフレーズは、それぞれ 1 対 1 対応する。

上記 2 つの仮定より、元言語文のフレーズ数と目的言語文のフレーズ数が一致することは自明である。元言語文と目的言語文のフレーズ分割と、フレーズ間の対応を $\boldsymbol{c}$ とすると、

フレーズモデルは以下の式で表される。

$$P(\boldsymbol{e} \mid \boldsymbol{f}) = \sum_{\boldsymbol{c}} P(\boldsymbol{c})\, P(\boldsymbol{e} \mid \boldsymbol{f}, \boldsymbol{c})$$

$P(\boldsymbol{c})$ は歪モデルと呼ぶが、ここでは簡単のために定数 ϵ とする。また、$P(\boldsymbol{e} \mid \boldsymbol{f}, \boldsymbol{c})$ は、各フレーズの対応ごとに独立であるとすると、

$$P(\boldsymbol{e} \mid \boldsymbol{f}) = \epsilon \sum_{\boldsymbol{c}} \prod_{(\bar{e},\bar{f}) \in \boldsymbol{c}} P\left(\bar{e} \mid \bar{f}\right) \tag{2.2}$$

と書ける。ただし、$\bar{e}$ と $\bar{f}$ はフレーズ同士の対応であり、これをフレーズペアと呼ぶ。

式 2.2から、フレーズベースモデルは元言語文と目的言語文のペアについて、それぞれの部分単語列を組み合わせたペアが翻訳関係にある確率を必要とする。すなわちフレーズモデルとは、元言語文 $\boldsymbol{f}$ と目的言語文 $\boldsymbol{e}$ について、それぞれの部分単語列 (フレーズ) を $\bar{f}, \bar{e}$ と置くと、任意のフレーズのペアについて $P\left(\bar{e} \mid \bar{f}\right)$ を与えるモデルである。

フレーズモデルは、各 $P\left(\bar{e} \mid \bar{f}\right)$ の値をパラメータとして持つ。翻訳の際には、元言語側フレーズ・目的言語側フレーズ・確率値の三つ組が記載されたテーブルを参照しつつ翻訳する。このテーブルをフレーズテーブルと呼ぶ。現実的には、任意の $\bar{e}, \bar{f}$ に対して確率値を持つことはできないため、ある程度それらしいペアについてのみ確率値を保持し、それ以外のペアについては確率 0 として取り扱う。

統計的機械翻訳の基本式にフレーズモデルを代入すると、

$$\hat{\boldsymbol{e}} = \arg\max_{\boldsymbol{e}} \sum_{\boldsymbol{c}} P(\boldsymbol{e}) \prod_{(\bar{e},\bar{f}) \in \boldsymbol{c}} P\left(\bar{e} \mid \bar{f}\right)$$

となる。ここで、$\sum_{\boldsymbol{c}}$ の計算は膨大な計算量を必要とするため実用的でないため、フレーズモデルでは更に近似を加え、

$$\hat{\boldsymbol{e}} = \arg\max_{\boldsymbol{e},\boldsymbol{c}} P(\boldsymbol{e}) \prod_{(\bar{e},\bar{f}) \in \boldsymbol{c}} P\left(\bar{e} \mid \bar{f}\right)$$

を求める。

2.2.2　階層フレーズモデル

階層フレーズモデル (Chiang 2007) は、フレーズモデルのフレーズを階層に拡張したモデルである。このモデルは Synchronous Context Free Grammar (SCFG) をベースとしている。階層フレーズモデルによると、元言語文 $\boldsymbol{e}$ と目的言語文 $\boldsymbol{f}$ の対は、以下の規則により確率的に生成される。

$$\begin{aligned} &< S, S > \rightarrow < SX, SX >, \\ &< S, S > \rightarrow < X, X >, \\ &< X, X > \rightarrow < \chi, \psi > . \end{aligned}$$

$< \chi, \psi >$ は階層フレーズモデルにおけるフレーズペアである。χ は元言語側のフレーズ、ψ は目的言語側のフレーズである。χ、ψ は共に各言語の単語を含む。これらの単語は SCFG における終端記号である。また、χ、ψ は非終端記号を含み、記号は X のみに限定される。

初期の階層フレーズモデルでは、翻訳時間を短縮するため以下の 2 つの制約を加える。

制約 1 非終端記号 X は元言語側で連続しない
制約 2 非終端記号 X は上限 2 つとする
制約 3 元言語側フレーズには少なくとも 1 単語の非終端記号を含む

統計的機械翻訳における翻訳モデルの確率値 $P(\boldsymbol{f} \mid \boldsymbol{e})$ は、SCFG による導出に基づいて以下の通り計算する。

$$P(\boldsymbol{f} \mid \boldsymbol{e}) \propto \sum_{\boldsymbol{c}} \prod_{<\chi,\psi>\in\boldsymbol{c}} P(<\chi,\psi>).$$

ここで、$\boldsymbol{c}$ は、与えられた言語対の可能な導出である。フレーズモデルと同様に、$\arg\max_{\boldsymbol{e}} \sum_{\boldsymbol{c}}$ の計算は現実的でないため、$\arg\max_{\boldsymbol{e},\boldsymbol{c}}$ で近似する。

2.3 言語モデル

2.3.1 $n$gram モデル

本研究の対象とする統計的言語モデルについて、その性質と特徴を述べる。統計的言語モデルは、N 単語からなる文 $\boldsymbol{w}_1^N$ の出現確率を与えるモデルである。統計的機械翻訳において最もよく利用されているのは $n$gram モデル (Jelinek 1990) である。$n$gram モデルは、$n-1$ 次マルコフモデルの一種であり、以下の式で表される。

$$P\left(\boldsymbol{w}_1^{N+1}\right) = \prod_{i=1}^{N+1} P\left(w_i \mid \boldsymbol{w}_{\max(i-n-1,0)}^{i-1}\right). \tag{2.3}$$

ここで、w_0 は文の開始記号 $< s >$、w_{N+1} は文の終端記号 $< /s >$ である。右辺の確率値 $P\left(w_i \mid \boldsymbol{w}_{\max(i-n-1,0)}^{i-1}\right)$ は、確率値を格納したテーブル α に格納しておき、必要に応じて参照する。

2.3.2 バックオフ

前節で述べたとおり、$n$gram モデルでは各 n 単語連鎖に対応する確率値をテーブル α に格納するが、n 単語連鎖のあらゆる組み合わせを持つ必要があり、これは大量のメモリを必要とする。この問題に対処するため、テーブル α に格納する単語列は学習データに出現した単語列のみに限定する。これにより、学習データに出現しなかった単語列に関しては確率値を取得できなくなるが、この場合は 1 単語短い $n-1$gram モデルの確率値で代用する。ただし、確率の公理を満たすため、$n-1$gram モデルの確率値に補正係数をかける。この手法をバックオフ (Katz 1987) と呼び、補正係数をバックオフウェイトと呼ぶ。

具体的には、以下のように表される。$j = \max(i-n-1, 1)$ とおくと、式 2.3右辺の確率値は、以下の式で表す。

$$P\left(w_i \mid \boldsymbol{w}_j^{i-1}\right) = \begin{cases} \alpha\left(\boldsymbol{w}_j^i\right) & \boldsymbol{w}_j^i\text{が学習データに現れた場合} \\ \beta\left(\boldsymbol{w}_j^{i-1}\right) P\left(w_i \mid \boldsymbol{w}_{j-1}^{i-1}\right) & \text{それ以外.} \end{cases}$$

ここで、$\alpha\left(\boldsymbol{w}_k^l\right)$ は確率値を格納したテーブル α から $\boldsymbol{w}_k^l$ から出現確率を取得する関数、$\beta\left(\boldsymbol{w}_k^l\right)$ はバックオフウェイトを格納したテーブル β から $\boldsymbol{w}_k^l$ からバックオフウェイトを取得する関数である。この式から、バックオフは再帰的に行われることがすぐにわかる。

また、$n$gram モデルにおいては、unigram (1gram) から、$(n-1)$gram までは各エントリについて、確率値とバックオフウェイトを持ち、$n$gram については書くエントリについて確率値のみを持つことがわかる。

$n$gram モデルは、学習データを増やせば増やすほど、エントリ数は増加し、それだけ多くのメモリを消費する。これは、学習データの増加に伴って、それに含まれる n 単語連鎖の種類数も増加するためである。本研究においては、エントリとなる単語列、確率値、バックオフウェイトの 3 つ組をコンパクトに格納しつつ、それらへ高速にアクセスする方法が求められる。

2.3.3　線形補間

学習データに出現しなかった単語列に対して確率値を与える方法として、バックオフの他に、線形補間モデル (Jelinek and Mercer 1980; Brown et al. 1992) がある。線形補間モデルは、確率値を格納したテーブル α に値が存在するときでも低次の言語モデルを参照する点が、バックオフと異なる。線形補間モデルにおける、各エントリの確率値を格納したテーブルを α' とし、そのテーブルの値を返す関数を $\alpha'\left(\boldsymbol{w}_j^i\right)$ とすると、線形補間モデルは、以下の式で表される。

$$P\left(w_i \mid \boldsymbol{w}_j^{i-1}\right) = \lambda\alpha'\left(\boldsymbol{w}_j^i\right) + (1-\lambda)\, P\left(w_i \mid \boldsymbol{w}_{j+1}^{i-1}\right).$$

ただし、$\boldsymbol{w}_j^i$ がテーブル α' に含まれないとき、値は 0 である。また、λ は、線形補間モデルにおけるパラメータである。

線形補間モデルはバックオフモデルに変換可能である。現在、一般的に使われているのは線形補間モデルであるが、実用的には推定結果のモデルをバックオフモデルに変換して使うのが一般的である。これは、バックオフモデルはテーブルに単語列が見つかった時点で計算を撃ち切れるが、線形補間モデルは必ず unigram まで値を探さねばならず、計算量の観点でバックオフモデルの方が優れているためである。

2.3.4　ARPA フォーマット

ARPA フォーマットは、バックオフモデルを表現するためのファイルフォーマットである。ほとんどすべての言語モデル実装は、ARPA フォーマットに対応しており、ある言語モデル実装で学習したバックオフモデルを、別の言語モデル実装から読み込むことができる。ARPA フォーマットはテキスト形式である。

このフォーマットは、以下の $n+1$ 個のセクションからなる。

- data セクション。言語モデルの各オーダーに何個のエントリが存在するかを記載する。
- 1-gram 〜 n-gram セクション。各オーダーに含まれるエントリの単語列、確率値、バックオフウェイトを記載する。

data セクションは、以下の行から始まる。

```
\data\
```

この行に続いて、1 以上、n 以下の各値 (m とする) について、

ngram m = [mgram のエントリ数]

が記載される。$m = n$ となった次の行は空行である。

$m$gram の各セクションは、以下の行から始まる。

\m–grams:

この行に続いて、$m$gram の各エントリについて、1 行ずつ以下の行が繰り返される。

[対数確率][TAB][単語列][TAB][対数バックオフウェイト]

ただし、TAB は水平タブ (ASCII 文字コードにおいては 9) を表す。また、対数の底は 10 である。すべてのエントリを記載した次の行は、空行である。また、$m = n$ のとき、対数バックオフウェイトは省略される[*1]。

$n$gram セクションの記載を終えた後、ファイルの末尾に以下の行を記載する。

\end\

2.4 デコーダ

2.4.1 フレーズモデル

フレーズモデルの翻訳は、Beam サーチ (Koehn, Och, and Marcu 2003) を用いる。翻訳時は、デコーダは仮説にフレーズペアを結合することで翻訳を進める。仮説とは、翻訳済み断片を意味し、本研究では $< \hat{f}, \hat{e} >$ と表記する。翻訳は目的言語文を前から後ろに生成する順に進める。したがって元言語側の翻訳済み断片 $\hat{f}$ は、必ずしも連続した単語列ではない。

デコーダは、元言語文が与えられるとその部分単語列 (フレーズ) を持つフレーズペアをフレーズテーブルから検索し、Lattice を作る。本研究では元言語文の部分単語列 $\boldsymbol{e}_i^j$ を持つフレーズペアの集合を、$\boldsymbol{\gamma}_i^j$ と表記する。また、Lattice 全体を $\boldsymbol{\gamma}$ と表記する。

次に、入力された元言語文の単語数よりひとつ多い数 ($N + 1$ 個) の仮説格納用スタックを用意する。i 番目 ($0 \leq i \leq N$) のスタックを本研究では s_i と表記する。i 番目のスタック s_i は元言語文のうち i 単語翻訳済みの仮説を格納するために利用する。翻訳開始前に初期仮説として $<, < s >>$ を s_0 に格納しておく。これは 0 単語翻訳済みを表す仮説である。

デコーダは、Lattice $\boldsymbol{\gamma}$ とスタック s_0^N を用いてアルゴリズム 1に示した手順で翻訳を行う。ビーム幅はデコーダを制御するパラメータの一つである。ビーム幅が大きいとデコーダの探索範囲が広がり、より翻訳確率の高い翻訳結果を得られるが、実行速度が低下する。逆に、ビーム幅が小さいと翻訳確率の低い翻訳結果となってしまうが、実行速度は速い。デコーダは各スタックに格納された仮説について、Lattice に含まれるフレーズペアと突き合わせながら翻訳を進める。

*1. 言語モデル実装によっては、$m < n$ の場合でもまれに省略される場合があるが、実用上はバックオフウェイトが 1 であるとして扱って問題ない

アルゴリズム 1: Lattice とスタックから翻訳する手順 (フレーズモデル)

Input: Lattice $\boldsymbol{\gamma}$
Input: スタック $\boldsymbol{s}_0^N$
Input: ビーム幅 b
Result: 翻訳結果の目的言語文 $\bar{\boldsymbol{e}}$
begin
  for $i \leftarrow 0$ *to* $N-1$ *step* 1 **do**
    $s \leftarrow s_i$ の翻訳確率上位 b 件
    foreach *hypothesis* $\in s$ **do**
      foreach $< j, k >\in$ *hypothesis* の元言語文における翻訳済み区間を除いた連続する区間 **do**
        foreach $< \bar{f}, \bar{e} >\in \boldsymbol{\gamma}_j^k$ **do**
          newhypo $\leftarrow$ *hypothesis* と $< \bar{f}, \bar{e} >$ を結合してできた新しい仮説
          $l \leftarrow$ *newhypo* の元言語文における翻訳済み単語数
          s_l に *newhypo* を追加
  return s_N の翻訳確率最大となる仮説

アルゴリズム 1においては、1 単語ずつ順に翻訳を進める。すなわち、$s_0, s_1, s_2, \cdots$ の順にスタックに格納された仮説を処理する。前述のとおり、スタックは翻訳済み単語数が同じ部分翻訳結果 (仮説と呼ぶ) に分けて格納する。これは 0 単語翻訳済み仮説群 $\rightarrow$1 単語翻訳済み群 $\rightarrow$2 単語翻訳済み群 $\rightarrow \cdots$ の順で仮説を処理していくことを意味する。

i 番目の仮説を処理するとき、s_i に含まれる仮説を順に取り出し、取り出した仮説は i 単語翻訳済みであるため、$N-i$ 単語の未翻訳部分が残っている。未翻訳部分に適用可能なフレーズペアの集合は、Lattice を参照することで得られる。これにより、取り出した仮説と適用可能なフレーズペア ($< \bar{f}, \bar{e} >$ とする) を組み合わせて、新しい仮説を作ることができる。新しく出来た仮説は、$s_{i+|\bar{f}|}$ に格納する。ここで $|\bar{f}|$ は $\bar{f}$ の単語数であるとする。

仮説とフレーズペアを結合する際は、仮説に含まれる翻訳済み目的言語文の後ろに、フレーズペアの目的言語フレーズを接合する。したがって、翻訳の過程において目的言語文は文の先頭から後方に向かって生成される。

仮説とフレーズペアを接合して新しい仮説を作る際には翻訳モデルと言語モデルの翻訳確率を計算する。翻訳モデルについては、フレーズペアの出現確率 (対数確率) がモデルパラメータとして参照可能なので、その値を結合元の仮説のスコアに加算する。言語モデルについては目的言語側フレーズ $\bar{e}$ の各単語について出現確率 (対数確率) を求め、仮説のスコアにすべて加算する。$\bar{e}$ の出現確率は、以下の 2 パターンを考慮する必要がある。

- 先頭 $n-1$ 単語の出現確率については、仮説の末尾 $n-1$ 単語を参照しつつ $n$gram モデルの値を求める。これらについては各単語以前に $n-1$ 単語が確定しないと確率値も確定しないためである。
- n 単語目以降の出現確率については、各単語の出現確率の合計を事前計算しておき結合時には単にその値を加算するのみとする。これらの単語については、フレーズ

ペアを読み込む時点で確率を確定できるためである。

2.4.2 階層フレーズモデル

階層フレーズモデル (Chiang 2007) に基づく翻訳は、CKY アルゴリズムに基づいて翻訳する。すなわち、任意の元言語部分単語列の数だけスタックを用意し、目的言語文の部分単語列 f_i^j を翻訳済みの仮説を X_i^j に格納する。仮説は元言語文の同じ部分単語列を翻訳済みのものを同じスタックに入れる。翻訳仮説をボトムアップで組み上げながら翻訳を進める。

アルゴリズム 2に、翻訳手順を示す。このアルゴリズムでは、元言語文 $\boldsymbol{f}$ に対する Lattice $\boldsymbol{\gamma}$ は生成済みであるとしている。フレーズペアの場合と同様、元言語側単語列 f_i^j に対応するフレーズペア集合は $\boldsymbol{\gamma}_i^j$ と表記する。ここで、Lattice に含まれるフレーズペアは非終端記号を含みうることに留意されたい。例えば、3 単語からなる元言語文 $f_1 f_2 f_3$ に対して、$\boldsymbol{\gamma}_1^3$ には元言語側フレーズが $f_1 X f_3$ となるようなフレーズペアを含みうる。

まず、非終端記号 X の導出について述べる。フレーズペアに含まれる非終端記号 X は、元言語文において範囲を特定できる。この範囲を k, l とすると、$\boldsymbol{f}_k^l$ に対する翻訳仮説はスタック X_k^l に格納する。ボトムアップに翻訳を進めるとき、スタック X_k^l は既に翻訳済みであるため、この仮説を使って新しい範囲 $\boldsymbol{f}_i^j$ の翻訳結果を生成できる。

非終端記号 X についての導出が完了すると、次に非終端記号 S の導出を行う[*2]。S に関する導出ルールより、S に関するスタックは左から右へ処理される。

階層フレーズモデルでは、目的言語側の文が左右両方向に生成される。例えば、フレーズペアの目的言語側フレーズ $e_1 X e_2$ と、翻訳仮説 $\bar{e}$ を組み合わせて新しい仮説を作るとき、新しい仮説は $e_1 \bar{e} e_2$ となり、元の仮説の前後に 1 単語ずつ付与される。

2.4.3 Right State

State (Li and Khudanpur 2008) は、言語モデルの評価プロセスをより簡潔に表現するための概念である。State には Right State と Left State の 2 種類がある。本説では Right State について述べる。

統計的機械翻訳では、言語モデルを評価するとき入力文を前から後ろに向かって一単語ずつ評価する (式 2.3)。このとき、確率 $P\left(w_i \mid \boldsymbol{w}_j^{i-1}\right)$ の $\boldsymbol{w}_j^{i-1}$ を Right State と呼ぶ。すなわち、文 $\boldsymbol{w}_1^N$ の出現確率 $P\left(\boldsymbol{w}_1^{N+1}\right)$ は、

$$
\begin{aligned}
P\left(\boldsymbol{w}_1^{N+1}\right) &= \prod_{i=1}^{N+1} P\left(w_i \mid \boldsymbol{w}_j^{i-1}\right) \\
&= \prod_{i=1}^{N+1} P\left(w_i \mid state\left(\boldsymbol{w}_j^{i-1}\right)\right)
\end{aligned}
$$

と置き換えできる。ただし、$state\left(\boldsymbol{w}_j^{i-1}\right) = \boldsymbol{w}_j^{i-1}$ とする。

条件付き確率の条件部分を関数化することで、バックオフ計算を効率化できる。$\boldsymbol{w}_j^i$ がテーブル α に含まれず、かつ $\boldsymbol{w}_j^{i-1}$ がテーブル β に含まれないとき、バックオフモデル

*2. 反復の順番を工夫することでこれらは同時に行うこともできる

アルゴリズム 2: Lattice とスタックから翻訳する手順 (階層フレーズモデル)

Input: Lattice $\boldsymbol{\gamma}$
Input: X の導出結果を格納するスタック $\boldsymbol{X}$
Input: S の導出結果を格納するスタック $\boldsymbol{S}$
Input: ビーム幅 b
Result: 翻訳結果の目的言語文 $\bar{\boldsymbol{e}}$
begin
  for $i \leftarrow 0$ *to* $N+1$ *step* 1 **do**
    for $j \leftarrow i$ *to* $N+1$ *step* 1 **do**
      foreach $< \bar{f}, \bar{e} > \in \boldsymbol{\gamma}_i^j$ **do**
        $X_1 \leftarrow \bar{f}$ に含まれる 1 つ目の非終端記号
        $X_2 \leftarrow \bar{f}$ に含まれる 2 つ目の非終端記号
        $k_1, l_1 \leftarrow X_1$ の元言語側の範囲
        $k_2, l_2 \leftarrow X_2$ の元言語側の範囲
        for $hypothesis_1 \leftarrow X_{k_1}^{l_1}$ **do**
          for $hypothesis_2 \leftarrow X_{k_2}^{l_2}$ **do**
            $hypothesis \leftarrow < \bar{f}, \bar{e} >, hypothesis_1, hypothesis_2$ から作成した仮説
            X_i^j に $hypothesis$ を追加
    X_i^j を翻訳確率上位 b 件に絞り込む
  for $i \leftarrow 0$ *to* $N+1$ *step* 1 **do**
    // 導出ルール $< S, S > \rightarrow < X, X >$ を利用
    S_i に X_0^i の内容をコピーする。
    for $j \leftarrow 1$ *to* i *step* 1 **do**
      foreach $hypothesis_1 \in S_{j-1}$ **do**
        foreach $hypothesis_2 \in X_j^i$ **do**
          // 導出ルール $< S, S > \rightarrow < SX, SX >$ を利用
          $hypothesis \leftarrow hypothesis_1, hypothesis_2$ を組合せて作成した仮説
          S_i に $hypothesis$ を追加
  return S_{N+1} のうち、翻訳確率最大となる仮説

により 1 単語トランケートされて、$P\left(w_i \mid \boldsymbol{w}_j^{i-1}\right) = P\left(w_i \mid \boldsymbol{w}_{j+1}^{i-1}\right)$ となる。これを一般化するには、関数 *state* を、以下のように再定義する。

$$state\left(\boldsymbol{w}_j^{i-1}\right) = \boldsymbol{w}_{\hat{k}}^{i-1},$$
$$where\ \hat{k} = \min\left(\left\{k \mid j \leq k \leq i-1 \wedge I\left(\boldsymbol{w}_k^{i-1} \in \alpha\right)\right\}\right).$$

ただし、I は指示関数である。この式は、単語列 $\boldsymbol{w}_j^{i-1}$ が与えられたとき $\boldsymbol{w}_j^{i-1}, \boldsymbol{w}_{j+1}^{i-1}, \cdots \boldsymbol{w}_j^j$ の中で言語モデルに含まれる最長の単語列を求める式である。

State を使うと、言語モデルの確率評価手順がより簡潔に書ける。まず、言語モデルを

アルゴリズム 3: State を用いた文の出現確率を評価する手順

Input: 文 $\boldsymbol{w}_1^N$
Result: $P\left(w_1^{N+1}\right)$
begin
  $p \leftarrow 1$
  $state \leftarrow \{< s >\}$
  for $i \leftarrow 1$ *to* $N+1$ *skip* 1 **do**
    $pw, state \leftarrow LM(w_i, state)$
    $p \leftarrow p \times pw$
  return p

評価する関数 LM を、state と目的単語を引数に取る関数とする。さらに、返り値として単語列の確率だけでなく、一単語ずらした $state$ の値も返すようにする。すなわち、

$$LM\left(w_i, state\left(\boldsymbol{w}_j^{i-1}\right)\right) = \left\{P\left(w_i \mid state\left(\boldsymbol{w}_j^{i-1}\right)\right), state\left(\boldsymbol{w}_j^{i}\right)\right\}$$

となる。ここで、$state\left(\boldsymbol{w}_j^i\right)$ は、$i-j+1 > n$ のとき $state\left(\boldsymbol{w}_j^i\right) = state\left(\boldsymbol{w}_{j+1}^i\right)$ が成立する。言語モデルの値を評価する関数は、バックオフ計算の副産物として $state\left(\boldsymbol{w}_j^i\right)$ の解を得られることから、これらの値を同時に返すことは効率が良い。

言語モデルの値を評価する際は、前から順に 1 単語ずつ評価していくことから、得られた state の値を次の単語の確率評価時に引数として与えることで自動的に不要な単語がトランケートされる。アルゴリズム 3に、文 $\boldsymbol{w}_1^N$ の確率評価の手順を示す。

フレーズモデルに基づく統計的機械翻訳では、翻訳仮説は前から後ろに向かって生成される。各仮説は仮説の最後の単語を評価した視点での State を保持する。フレーズペアを結合して新たな仮説を生成するときは生成元の仮説が持つ Right State を用いて、新たに結合された単語列の確率値を評価する。

2.4.4 Left State

階層フレーズモデルにおいては、仮説は前後両方の方向に生成される。したがって、ある翻訳仮説に対して先頭 $n-1$ 単語の各出現確率を仮説の生成に伴って修正する必要がある。従って、仮説の目的言語側単語列先頭 $n-1$ 単語は、文頭が確定しないかぎり確率値も確定しない。ある仮説の目的言語側単語列について、先頭 m 単語 $(1 \leq m < n)$ が $\boldsymbol{v}_1^m$ であるとする。このとき、m 番目の単語を含む先頭 $n-1$ 単語の出現確率は確定できない。$n$gram モデルは n 単語の単語列に対して確率を与えるモデルであるためである。

階層フレーズモデルに基づくデコーダが、先頭 m 単語目の確率値をどのように処理するかを述べる。まず、デコーダは v_m の出現確率を暫定的に $P\left(v_m \mid \boldsymbol{v}_1^{m-1}\right)$ と置いて処理を進める。デコーダがこの仮説に対して適当なフレーズペアを結合することで新たな仮説を生成する際、仮説の前方に単語列 $\boldsymbol{u}_1^{n-m}$ を結合すると、単語 v_m の前方に $n-1$ 単語存在するため出現確率が確定できる。したがって、デコーダは仮説の生成時に、先頭 $n-1$ 単語について確率値を更新する。

ここで、仮説の 1 単語目が未知語だった場合を考える。このとき、仮説の先頭 $n-1$ 単語の確率値はいかなるフレーズペアの結合によっても更新できない。未知語を含む単語列

は言語モデルのエントリとして含まれないためである。このとき、デコーダは先頭 $n-1$ 単語の確率値を更新する必要はなく、言語モデルの呼び出し自体を省略できる。

この処理を効率的に行うため、Left State (Li and Khudanpur 2008; Heafield et al. 2011) を導入する。Left State は、翻訳仮説の先頭 $n-1$ 単語のうち、確率値が確定していない単語列を記憶するためのものである。Right State と同様、各仮説は Left State を保持し、ここに含まれる単語列についてデコーダは確率値を更新する。

2.4.5　State 最適化

Right State、Left State 共に、短ければ短いほど処理を効率化できる。したがって、言語モデルの State 処理は可能な限り短い単語列を返すべきである。最適化の方法は様々なものが提案されているが、本研究においてもっとも重要な拡張性 (Li and Khudanpur 2008; Heafield et al. 2011) について述べる。

Right State 場合、単語列 $\boldsymbol{w}_j^i$ がテーブル α に含まれている場合でも、$\boldsymbol{w}_j^i w$ となるような単語 w がモデル中に存在しないことがある。このとき、単語列 $\boldsymbol{w}_j^i$ は「右に伸びない」という。

単語列 $\boldsymbol{w}_j^i$ が右に伸びないとき、次回のクエリは以下のように必ずバックオフされる。

$$P\left(w \mid \boldsymbol{w}_j^i\right) = P\left(w \mid \boldsymbol{w}_{j+1}^i\right) \beta\left(\boldsymbol{w}_j^i\right)$$

ここで、$\beta\left(\boldsymbol{w}_j^i\right) = 1$ のとき、$P\left(w \mid \boldsymbol{w}_j^i\right) = P\left(w \mid \boldsymbol{w}_{j+1}^i\right)$ となり w_j をトランケートした場合と結果が一致する。

したがって、単語列 $\boldsymbol{w}_j^i$ が右に伸び、かつそのバックオフウェイトが 1 のとき、クエリ結果の State から w_j を除いても結果が一致する。この場合、w_j はクエリ結果の State から除く (Heafield 2011)。

同様に、Left State も任意の $m$gram が左に伸びるかどうかが問題となる (Heafield et al. 2011)。Left State は、階層フレーズモデルに基づく翻訳で用いられるが、階層フレーズモデルに基づく翻訳の際、仮説は左右いずれにも単語を追加される可能性がある。仮説の先頭 $n-1$ 単語の各単語については、それ以前に単語が付与されると単語の出現確率を更新する必要がある。$n$gram モデルにおいては、単語の直前 $n-1$ 単語がその単語の出現確率に影響するためである。

階層フレーズモデルに基づく翻訳において、仮説の先頭 m 単語 ($m < n-1$) からなる単語列 $\boldsymbol{e}_1^m$ が左に伸びないとき、e_m の出現確率は確定できる。これは、仮説の左側にいかなる単語が付与されてもそれらの単語に確率値が影響されないこと[*3]が保証されるからである。さらに、カットオフ[*4]値を単調増加に設定している場合、$\boldsymbol{e}_1^m$ が左に伸びないとき $\boldsymbol{e}_1^{m+1}$ も左に伸びない。したがって、m 単語目以降の単語列を確定できるため、言語モデルの呼び出し回数を減らすことができる。

*3. 厳密にはバックオフウェイトを除いた確率値のみが確定する。仮説の左側に単語が付与された場合のバックオフウェイトの乗算は後から小さい計算量で対処できるが詳細は省略する。

*4. 言語モデルのサイズをコンパクトにするためによく使われる手法の一つである。カットオフは、出現回数の少ない単語列をモデルから削ることでモデルサイズを減らすことができる。削除の基準とする出現回数 (設定した数値に呼称はないが、本研究ではカットオフ値と呼ぶ) の設定は、オーダーごとに行う。一般には、オーダーの増加に伴って単調増加するようカットオフ値を設定する。

2.4.6 リコンバイン

統計的機械翻訳のデコーダは、より翻訳確率の高い仮説を求めて様々な仮説を生成するが、確実にスコアが低いと判明した仮説についてはその仮説を捨てる。これは枝切りの一種であるが、統計的機械翻訳においてはリコンバインと呼ばれる。

リコンバインにおいては、2 つの仮説がその先同じように伸び、仮説のスコアが全く同じ差分となるとき、その時点で低いスコアの仮説を捨てる。同じ元言語単語を翻訳した 2 つの仮説が、同じ state を持つとき、それぞれの仮説に同じフレーズペアで新しい仮説を生成し続けると、そこから先の確率評価は全く同じ計算となる。したがって、その時点でスコアの低い仮説は捨てて良い。

この手法は、前述の state 最適化と合わせて利用する。State の最適化によって、単語列の長さが短ければ、他の翻訳仮説と state が一致しやすいため、state を利用しない場合と比べて多くの仮説を捨てられる。Beam search では余計な仮説を捨てれば捨てるほど実質的に探索空間を広げることとなり、よりスコアの高い翻訳結果を得られる。

2.5 言語モデルのデータ構造

2.5.1 Hash Table

Hash Table は、ハッシュ関数を用いたデータ構造の一つで、言語モデルを実装するときに使われる。Hash Table では、格納すべき $m$gram $\boldsymbol{v}_1^m$ を、適当なハッシュ関数 $h\left(\boldsymbol{v}_1^m\right)$ の値に基づいて格納する。

Hash Table の構築法は以下の通りである。長さ l の配列 α を用意する。α には、エントリの見出し ($m$gram)、確率値、バックオフウェイトを格納する。l は格納したいエントリ数よりも大きい必要がある。ハッシュ関数 $h\left(\boldsymbol{v}_1^m\right)$ を用意する。ハッシュ関数の値は、$0 \leq l < l$ を満たす整数である必要がある。あるエントリ $\boldsymbol{v}_1^m$ に対して、$k = h\left(\boldsymbol{v}_1^m\right)$ とすると、$\alpha\left[k\right]$ がエントリ格納先の候補となる。ハッシュ関数は異なる 2 つのエントリに対して、同じハッシュ値を与えることがあるため、$\alpha\left[k\right]$ が既に利用されていないか確認する必要がある。該当箇所が使用済みの場合、未使用位置が得られるまで k を 1 ずつ増加させる。得られた未使用位置に対して、エントリの見出し、確率値、バックオフウェイトを格納する。

Hash Table から確率値、バックオフウェイトを取り出す方法は以下の通りである。入力の $m$gram $\boldsymbol{v}_1^m$ から、ハッシュ値を計算する。$k = h\left(\boldsymbol{v}_1^m\right)$ として、$\alpha\left[k\right]$ の見出しが、$\boldsymbol{v}_1^m$ と一致するか調べる。一致する場合、$\alpha\left[k\right]$ に格納された確率値、バックオフウェイトを返す。一致しない場合、一致するまで k を増加させ、一致した場所の確率値、バックオフウェイトを返す。ただし、k を増加させている間に未使用位置が見つかった場合、$\boldsymbol{v}_1^m$ はテーブルに存在しないと判定する。

2.5.2 Trie

Trie (Fredkin 1960) は、Hash Table と並んで言語モデルの実装に良く用いられるデータ構造である。Trie は、木構造の一種であり、ノード間遷移を単語で行うという特徴を持

アルゴリズム 4: Hash Table から $m$gram を検索する手順

Input: $m$gram $(\boldsymbol{v}_1^m)$
Result: $m$gram を格納した Hash Table の位置
begin
  $k \leftarrow h(\boldsymbol{v}_1^m)$
  while $\alpha[k]$ にエントリが格納されている場合 **do**
    if $\alpha[k] = \boldsymbol{v}_1^m$ **then**
      return k
    else
      $k \leftarrow k + 1$

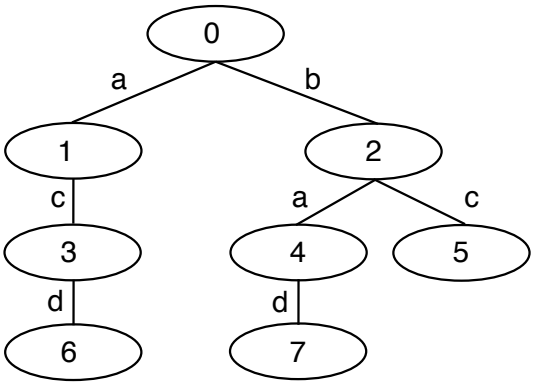

図2.1　Trie の例。ノード間を単語で遷移する木構造の一種である。

つ。図 2.1に、Trie の例を示す。丸が木構造のノードに相当し、ノード間をつなぐ線の横に付与されている文字が遷移に利用する単語である。丸の中の数字は便宜上与えたノード番号である。例えば、3 単語からなる単語列 $b\ a\ d$ を Trie 上で探索する際は、ノード 0 を起点として、ノード 2、ノード 4、ノード 7 の順にノードを辿る。

Trie には、データの格納方法について幾つか種類がある。本研究では Backward Suffix Tree (Stolcke 2002) と Reverse Trie の 2 種類を利用するため、それらについて順に述べる。

Backward Suffix Tree は、言語モデルの各エントリを 2 段階に分けて格納する。まず、言語モデルに含まれる各エントリの履歴単語列のみで Trie を作成する。履歴単語は逆順で格納する。次に、目的単語を表すノードに付随する表に格納する。Backward Suffix Tree の例を図 2.2に示す。この例では、Trigram “you eat soup” にアクセスするには、“eat” → “you” とノードをたどり、<2> に到達する。次に、<2> の中から目的単語の “soup” を探す。Backward Suffix Tree をたどる為の擬似コードを、アルゴリズム 5に示す。

Reverse Trie は言語モデルの各エントリに現れる単語を逆順に格納する。Reverse Trie の例を図 2.3に示す。Trigram “you eat soup” にアクセスするには、すべての単語を逆に辿る。すなわち、“soup” → “eat” → “you” の順にノードを辿る。Reverse Trie をたどる為の擬似コードを、アルゴリズム 6に示す。

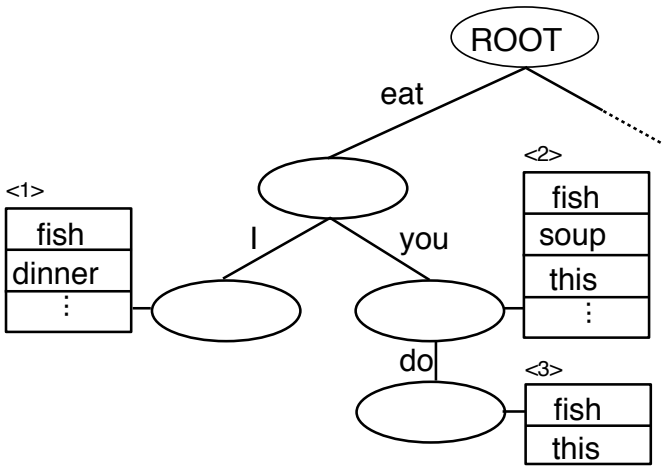

図2.2 Backward Suffix Tree の例。言語モデルに含まれる各エントリを目的単語と履歴単語列に分解し、履歴単語列を逆順で格納する。

アルゴリズム 5: Backward Suffix Tree から $m$gram を検索する手順

Input: $m$gram $(\boldsymbol{v}_1^m)$
Result: $m$gram を表す Backward Suffix Tree のスロット
begin
  node ← *ROOT*
  for $i \leftarrow m-1$ ***to*** 1 ***step*** -1 **do**
    node ← *node* から単語 v_i で遷移した先の子ノード
    if *node* が見つからなかった場合 **then**
      return *Not Found*
  slot ← *node* に接続する単語リストのうち、v_m を格納したスロット
  if *slot* が見つからなかった場合 **then**
    return *Not Found*
  else
    return *slot*

Reverse Trie は、バックオフ計算と相性が悪い。$m$gram $\boldsymbol{w}_j^i$ に対するクエリの際にバックオフが発生したとすると、以下の計算が必要となる。

$$P\left(w_i \mid \boldsymbol{w}_j^{i-1}\right) = P\left(w_i \mid \boldsymbol{w}_{j+1}^{i-1}\right) \beta\left(\boldsymbol{w}_j^{i-1}\right).$$

$P\left(w_i \mid \boldsymbol{w}_j^{i-1}\right)$ を求めるためには、Trie を $ROOT \to w_i \to w_{i-1} \to \cdots \to w_2 \to w_1$ の順に辿る必要があるが、w_1 が見つからなかった場合の確率値は一単語短い w_2 のノードを参照する[*5]。すなわちバックオフ計算において、確率値のパラメータはクエリされた単語列の探索と同時に得ることができる。

*5. もし w_2 が見つからなかった場合は w_3、というように最終的に見つかったノードの確率値を参照すれば良い

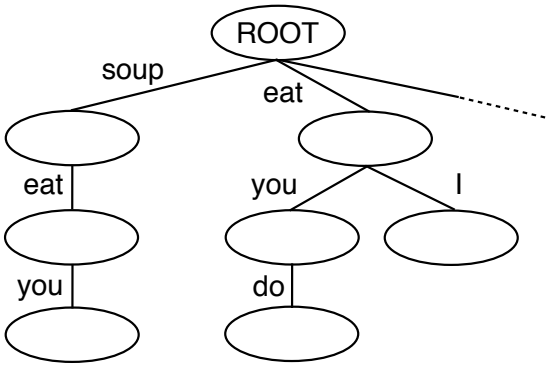

図2.3　Reverse Trie の例。言語モデルに含まれる各エントリを逆順で格納する。

アルゴリズム 6: Reverse Trie から $m$gram を探索する手順

Input: $m$gram $(\boldsymbol{v}_1^m)$
Result: $m$gram を表すノード
begin
　$node \leftarrow ROOT$
　for $i \leftarrow m$ ***to*** 1 ***step*** -1 **do**
　　$node \leftarrow node$ から単語 v_i で遷移した先の子ノード
　　if $node$ が見つからなかった場合 **then**
　　　return *Not Found*
　return $node$

しかし、バックオフウェイトは $\beta\left(\boldsymbol{w}_j^{i-1}\right)$ が必要であり、この値を得るためには Trie を $ROOT \rightarrow w_{i-1} \rightarrow \cdots \rightarrow w_{j+1} \rightarrow w_j$ の順に辿る必要がある。これは、バックオフウェイトは改めて ROOT ノードからたどり直す必要があることを意味し、クエリ速度の低下を招く。

Heafield (2011) は、Reverse Trie の速度を改善するために Right State を拡張した。前述のとおり、Right State は連続したクエリに置いてひとつ前のクエリから次のクエリを効率化するためのものである。Right State は、クエリで求めたい条件付き確率の条件部分に必要な単語を格納していたものだが、Heafield はこれにバックオフウェイトも持つように修正した。連続クエリにおいて、直前のクエリは $P\left(w_{i-1} \mid \boldsymbol{w}_{j-1}^{i-2}\right)$ を求めているはずなので、$ROOT \rightarrow w_{i-1} \rightarrow \cdots \rightarrow v_{j+1} \rightarrow v_j$ の順に Trie を辿っているはずである。これはバックオフウェイトを拾うために辿り直す Trie の辿り方と同一である。したがって、前回クエリ時に Trie を辿る際、同時にバックオフウェイトも参照して State として保存すれば良い。すなわち、拡張 State は以下のように表される。

$$state_{extended}\left(\boldsymbol{w}_j^{i-1}\right) = \boldsymbol{w}_{\hat{k}}^{i-1}, \left\{\beta\left(\boldsymbol{w}_{\hat{k}}^{i-1}\right), \beta\left(\boldsymbol{w}_{\hat{k}+1}^{i-1}\right), \cdots, \beta\left(\boldsymbol{w}_{i-1}^{i-1}\right)\right\},$$

ここで、$\hat{k}$ は元々の State 定義時に定義されたのと同じ、モデルに含まれる単語列のうち最も長い単語列となる位置を表す。

State を拡張することでクエリ時に Trie を辿り直す必要がなくなる。バックオフの際に

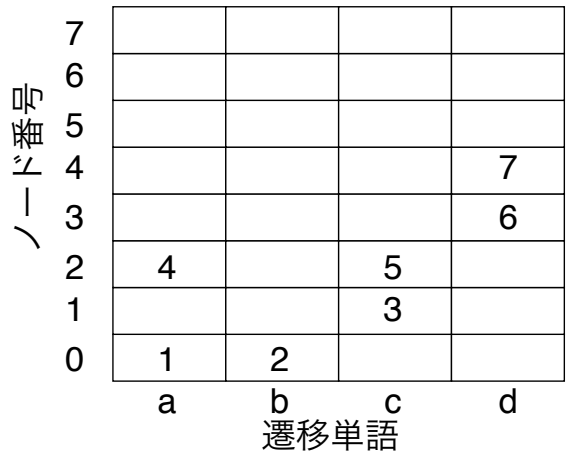

図2.4 疎行列による Trie の表現例。ノード番号と遷移単語のクロスする位置に遷移先ノード番号が記入されている。

は引数に与えられた Right State からバックオフウェイトを取得し利用する。

2.6 ダブルアレイ

2.6.1 ダブルアレイのデータ構造

本研究で利用するダブルアレイ (Aoe 1989) について、その概要を述べる。

Trie の最も単純な実装として、疎行列を使う方法がある。この方法では行列の行を使って Trie のノードを表し、列を使って遷移可能な単語を表す。Trie の各ノードを行列の行に対応させるため、各ノードに対してノード番号を付与する。同様に、単語を列で表現するため、各単語に対して単語 ID を付与する。

図 2.1を疎行列で表現した例を図 2.4に示す。この図では、ノード番号と遷移単語のクロスする位置に遷移先ノード番号を記入している。例えば、ノード 0 は単語 a と b で子ノードへ遷移可能でありその子ノードのノード番号はそれぞれ 1 と 2 である。それゆえ、最下行は a と b の位置に 1、2 と値が記入される。この方式は、ノード遷移が高速であるというメリットがあるが、単に 2 次元配列として格納してしまうと大量のメモリを消費するという欠点がある。

ダブルアレイは、前述の単純な実装のノード遷移が高速であるというメリットを保ちつつ、メモリ使用量を削減する手法である。基本的な考え方としては、前述の疎行列の各行を、縦方向に重ならないようにずらし、一本の配列に潰す。本研究では得られた配列を *merged* と呼ぶ。ただし、一本の配列に潰してしまうと、以下の 2 つの情報が失われることになり、潰した配列から元の疎行列を復元することはできない。

- 潰した配列の数字が元々どの行にあったのか
- 潰した配列の数字はどれだけずらされてこの位置に来たのか

元の疎行列を復元可能にするため、これらの情報を格納する 2 つの配列を用意する。ひとつは元のノード番号を格納する配列、もう一つはずらし幅を格納する配列である。ここで

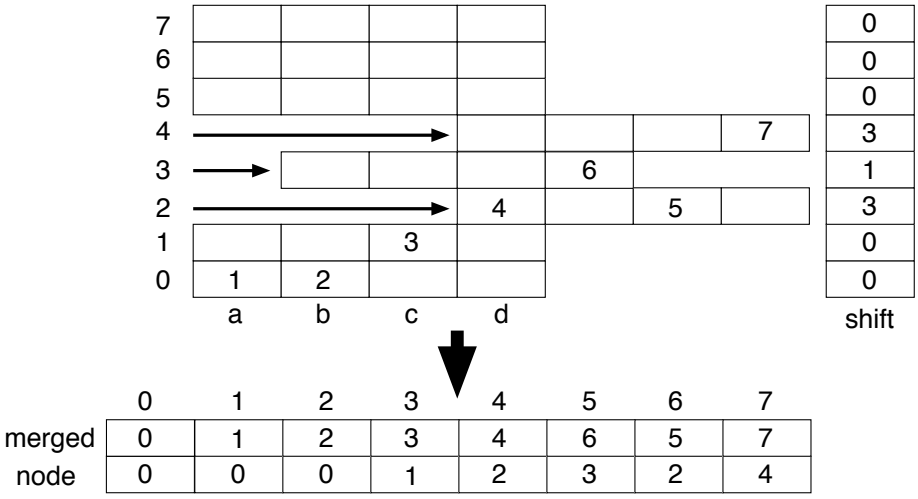

図2.5　ダブルアレイの基本的な考え方。疎行列の各行を縦方向に重ならないようにずらし、縦方向に潰す。元の情報が失われないよう、ノード番号とずらし幅をメモする。

はそれぞれ *node*, *shift* と呼ぶ。また、便宜的に、*merged*, *node* の先頭に 0 を追加する。以上で、図 2.5に示したような構造となる。

配列 *merged*, *node* の先頭に 0 を付与することで、このデータ構造上でノードを辿るのが簡単になる。元々、疎行列をマージしたことで無駄な領域は減ったが、同一ノードの遷移先を表す記号がデータ構造上に散逸した。例えば図 2.5において、ノード 2 に関する情報は配列の 4 番目と 6 番目に出現している。ノード 2 の親ノードはノード 0 だが、ノード 2 に遷移するためには *merged* 配列の 2 番目から遷移することになる。これは、*merged*[2] であることからわかる。同様にノード 0 は配列の 1 番目と 2 番目に現れるが、ノード 0 はルートノードなので親ノードを持たない。配列 *merged*, *node* の先頭に 0 を持つ要素を追加することで、便宜的に配列の 1 番目と 2 番目の親ノードとして、配列の 0 番目が存在するように解釈できる。

このデータ構造上でノードを辿る手順をアルゴリズム 7に示す。ここで、関数 *WordId* は、単語 v の単語 ID を返す関数である。更に *merged*[*context*] は merged 配列の *context* 番目の値を参照する操作であり、その他の配列についても同様である。配列 *merged* から得たノード番号を用いて *shift* からずらし幅を得、単語 ID と合わせて遷移先を特定する。遷移が成功したかどうかは遷移先の *node* 値と遷移元の *merged* 値が一致するかで調べられる。

ダブルアレイは、上記の手法を更に簡略化したものである。この手法では 3 つの配列を使うが、ダブルアレイでは 2 つの配列で同じ機能を実現する。

遷移の際には遷移元と遷移先が一致するか調べるが、この確認をノード番号を使わずに行う。配列 *node* は遷移元ノード番号を格納していたが、これは遷移元ノードを表す配列位置 (インデックス) で良い。*merged* はノード ID がそれぞれ 1 回しか出てこないためである。*node* を配列位置に置き換えた配列を *CHECK* と呼ぶ。

次に、*merged* と *shift* を統合する。遷移の際には必ず *merged* で得た値を元に *shift* を参照する。*CHECK* の導入により *merged* 単体で参照されることがなくなったため、各

アルゴリズム 7: *merged*, *node*, *shift* から Trie を辿る手順

Input: Trie を辿る順に並べた単語列 v_1^m
Result: 単語列を表すノード ID
begin
  $context \leftarrow 0$
  for $i \leftarrow 1$ ***to*** m ***step*** 1 **do**
    $j \leftarrow shift[merged[context]] + WordId(v_i)$
    if $node[j] = merged[context]$ **then**
      $context \leftarrow j$
    else
      return *Not Found*
  return *context*

	0	1	2	3	4	5	6	7
BASE	0	0	3	1	3	0	0	0
CHECK	0	0	0	1	2	3	2	4

図2.6 ダブルアレイの例。配列 BASE と CHECK により Trie を表現する。

アルゴリズム 8: ダブルアレイ上で Trie を辿る手順

Input: Trie を辿る順に並べた単語列 v_1^m
Result: 単語列を表すノード ID
begin
  $context \leftarrow 0$
  for $i \leftarrow 1$ ***to*** m ***step*** 1 **do**
    $j \leftarrow BASE[context] + WordId(v_i)$
    if $CHECK[j] = context$ **then**
      $context \leftarrow j$
    else
      return *Not Found*
  return *context*

merged の値を *shift* 参照済みの値に置き換えることができる。置き換えた配列を *BASE* と呼ぶ。

この操作により、得られたデータ構造をダブルアレイと呼ぶ。これは元の疎行列の情報をすべて持ったデータ構造である。

ダブルアレイ上でノードを辿る手順を、アルゴリズム 8に示す。

$$n = BASE[n_k] + WordId(v).$$

前述の例と同じ、$b\ a\ d$ を例にとって、ダブルアレイのノード遷移例を示す。まず、単語 ID

は $a \to 1$、$b \to 2$、$c \to 3$、$d \to 4$ である。これは、図 2.4で単語をおいた順に由来する。まず、インデックス 0 から単語 b で遷移する。$BASE[0] + WordId(b) = 2$ かつ $CHECK[2] = 0$ なので遷移に成功する。次に a で遷移する。$BASE[2] + WordId(a) = 4$ かつ $CHECK[4] = 2$ なので、遷移に成功する。最後に、d で遷移する。$BASE[4] + WordId(d) = 7$ かつ $CHECK[7] = 4$ なので、遷移に成功し、単語列 $b\ a\ d$ が Trie に含まれていることがわかる。

Trie は木構造であるため、長い単語列を格納するためにその単語列に至るための部分単語列が Trie に格納されている必要がある。一般的に、単語列の集合に長い単語列が Trie に含まれているからと言ってその部分単語列がもとの集合に含まれているとは限らない。そのため、Trie の各ノードが元の単語列集合に含まれていたものなのかを表示する必要がある。

エンドマーカーは、ダブルアレイの各ノードが元の単語列集合に含まれていたものなのか示すための特殊なノードである。ダブルアレイでは、Trie の各ノードの内、元のエントリ集合に含まれるノードにエンドマーカーノードを子ノードとして付与し、エンドマーカーノードへは特殊な単語 $< \# >$ で遷移可能にしておく。各ノードは $< \# >$ での遷移に成功した場合、そのノードが表す単語列が元の単語列集合に属しているものとして取り扱う。

統計的言語モデルでは、モデルに含まれる任意の $m$gram に対して部分単語列がモデルに含まれるという性質がある。そのためエンドマーカーを直接取り扱う必要はない。本研究ではエンドマーカーを別の用途に転用することで効率的なデータの格納を実現する。

2.6.2　ダブルアレイの構築

ダブルアレイの構築は大きな計算量を要する問題である。ダブルアレイのサイズが大きくてよければ、疎行列の各行を横に並べるだけで良いため、高速に構築できるが、これでは疎行列表現と同様膨大な未使用領域が発生してしまう。ここでは、コンパクトなダブルアレイを得るためのヒューリスティック手法を述べる[*6]。

ダブルアレイを構築する際には、Trie の上位ノード (ルートノードに近いノード) から順に挿入する。ここで、「ダブルアレイにノードを挿入する」とはダブルアレイ中の対応するスロットの BASE 値を決めることである。挿入の際には、子ノードの一覧を用い、すべての子ノードが構築時点で未使用位置に配置されるように決める。

ダブルアレイにノードを挿入する最も単純な方法は、BASE 値の値を 1 から順に 1 ずつ増加させ、すべての子ノードが、その時点で未使用となる位置に対応付けられるまで適切な BASE 値を探す。図 2.7に図 2.1のノード 2 を挿入する様子を示す。ノード 2 に対応する位置 BASE 値 x は、ノード 4,5 の位置を決める。したがって、ノード 4,5 が他のノードと衝突しないよう x の値を決める。

ダブルアレイを構築するのは計算量を要するタスクである。計算量を削減する手法はいくつか提案されている (中村康正 and 望月久稔 2006; 矢田晋 et al. 2009; 重越秀美, 蔵満琢麻, and 望月久稔 2009) が、本研究においては、双方向リストを利用する方法 (中村康正 and 望月久稔 2006) を利用する。この方法は、ダブルアレイの未使用領域を双方向リ

*6. この手法を提案した論文を見つけることができなかったが、ダブルアレイ実装の際にはよく用いられる手法の一つである。

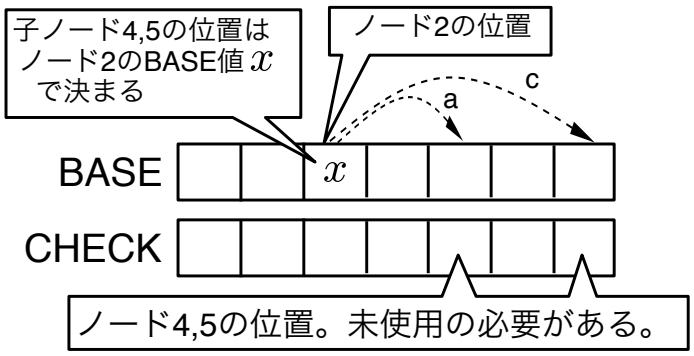

図2.7 ダブルアレイにノードを挿入する例。図 2.1のノード 2 を挿入する場合、子ノード 4, 5 の位置が未使用である必要がある。子ノード 4,5 が共に未使用となる位置を探すため x の値を 1 から順に増加させて探す。

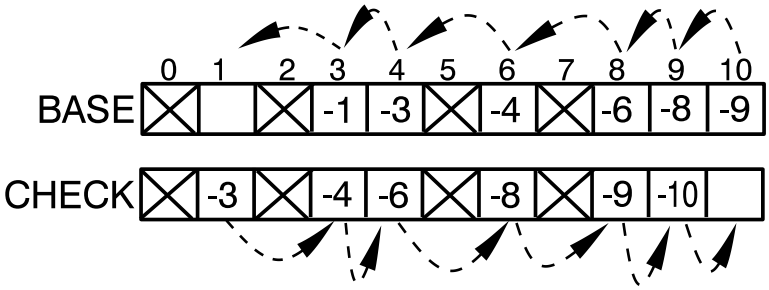

図2.8 双方向リストによる未使用位置の結合。未使用位置が双方向リストで結合されることにより、挿入したいノードの子ノード群の少なくとも 1 つを未使用位置に配置されるよう BASE 値を決定できる。

ストとして利用し、BASE 値の探索を効率化する。単純な方法では BASE 値は 1 から順に 1 ずつ増加させて、子ノードを配置可能な値を探したが、この方法を使うと、1 より大きな数で増加できる。

最も基本的なアイディアは、子ノードの内少なくともひとつは必ず未使用位置に配置されるよう BASE 値を増加させることで、BASE 値の探索範囲を減らす。これを実行するため、ダブルアレイの未使用領域置の BASE、CHECK を利用して双方向リストで結合する。ダブルアレイの未使用領域に関しては、未使用であることがわかれば任意の値を格納できる。そこで、各位置の BASE 側にひとつ前の未使用領域の位置、CHECK 側にひとつ後ろの未使用領域の位置を格納する (図 2.8)。ただし、ノードが配置されていないことを示すため、負の数で格納する。この双方向リストを使うことで、挿入したいノードの子ノード群のうち少なくとも 1 つが未使用位置に配置されるよう BASE 値を決定できる。従来法では 1 から順に増加させ子ノード群の位置を確認したが、この手法では増加幅を空き要素の間隔分だけ増加させることができるので、BASE 値の決定を高速にできる。ダブルアレイへのノード挿入手順をアルゴリズム 9に示す。BASE 値を増加させる際に $base \leftarrow pointer - smallest$ で、子ノードの衝突が確実な BASE 値候補をスキップすることで高速化する。

アルゴリズム 9: 双方向リストによるダブルアレイへのノード挿入手順

Input: $\boldsymbol{u}_1^m$: 挿入したいノードの表す $m$gram の単語 ID 列。ただし Trie をたどる順に並べる

Input: $\boldsymbol{c}$: 挿入したいノードから遷移可能な子ノードの単語 ID 集合

begin

- $node \leftarrow 0$ // 挿入したいノードのダブルアレイ上での位置を得る
- **for** $i \leftarrow 1$ ***to*** m **do**
  - $node \leftarrow BASE[node] + u_i$
- $smallest \leftarrow \min(\boldsymbol{c})$
- $pointer \leftarrow$ 未使用位置の内最小のインデックス
- $base \leftarrow pointer - smallest$
- **while** $base < 0$ // $base$ が 0 以下になるのを避ける **do**
  - $pointer \leftarrow -CHECK[pointer]$
  - $base \leftarrow pointer - smallest$
- **while** $True$ // $base$ の探索 **do**
  - $success \leftarrow True$
  - **foreach** $id \in \boldsymbol{c}$ **do**
    - **if** $CHECK[base + id] >= 0$ // 未使用位置は負の数 **then**
      - $pointer \leftarrow -CHECK[pointer]$
      - $base \leftarrow pointer - smallest$
      - $success \leftarrow False$
      - break
  - **if** $success$ **then**
    - break
- $BASE[node] \leftarrow base$
- **foreach** $id \in \boldsymbol{c}$ **do**
  - $child \leftarrow base + id$
  - **if** $child$ より前に未使用要素が存在しない // 双方向リストを更新 **then**
    - $previous \leftarrow BASE[child]$
    - $next \leftarrow CHECK[child]$
    - $CHECK[previous] = next$
    - $BASE[next] = previous$
  - $CHECK[child] = node$

2.6.3 ダブルアレイへの効率的なデータ格納

Trie は $m$gram と共に関連する値を格納することがある。ダブルアレイは Trie の実装手法の一つであり、ダブルアレイでも $m$gram と共に関連する値を格納する必要がある。

ダブルアレイに $m$gram だけでなく関連する値も格納する最も単純な方法は、BASE、CHECK の各配列だけでなく値格納用の配列 VALUE をもう一本追加することである。ダブルアレイでは、BASE、CHECK の各配列のインデックスを同じくする要素が Trie のノードを表すが、各ノードに対応する $VALUE$ 配列の同じ位置に、ノードの表す $m$gram に対応する値を格納する。

VALUE 配列に値を格納する方法は、シンプルだが未使用領域が発生する。Trie の葉に位置するノードは、子ノードを持たないため BASE 値を持たない。したがって、葉ノードの BASE 値格納位置が未使用のままメモリを消費する。

工藤によるダブルアレイ実装である Darts (Double Array Trie System)[*7]は VALUE 配列を使わず値を格納する。この方法では、エンドマーカーノードの BASE 値格納位置に値を格納する。エンドマーカーは常に葉ノードとなるため、必ず BASE 値格納場所が未使用のまま残っている。この領域に値を格納することで無駄な領域を発生させずデータ構造をよりコンパクトにすることに成功している。

工藤の手法は、値を BASE 配列に格納するため格納可能な値は BASE 配列の 1 要素分の大きさが上限となる。一般的に BASE 配列は 32bit 配列を利用するため、格納可能な値は 32bit までである。

*7. http://chasen.org/~taku/software/darts/

第 3 章

関連研究

3.1　初期の言語モデル実装

CMU Toolkit (Clarkson and Rosenfeld 1997) は、言語モデルを実装したツールキットの一つである。CMU Toolkit は、言語モデルの辞書構造として Trie を利用する。

CMU Toolkit では、Sorted Array と呼ばれる方法で Trie を表現する。まず、Trie の各ノードは、子ノードへ遷移可能な単語 ID を格納した配列を持つ。次に、この配列を同じオーダーごとに結合する。この結合によって、n 個の配列ができる。さらに、各オーダーに同じ長さの配列をもう一つ用意し、各単語の遷移先を格納する。Sorted Array による Trie 表現の例を図 3.1に示す。左側の Trie を Sorted Array で表現すると右側のような形になる。同じオーダーのノードをオーダーごとに 1 本の配列に格納し、子ノードの遷移先をポインタで示す。

Sorted Array 上で、Trie のノード遷移を行うためには二分探索が必要となる。二分探索は $O(\log M)$ の時間が必要となり、速度は遅い。

Sorted Array は、後続の言語モデル実装でも頻繁に利用されるデータ構造である。CMU Toolkit では単語を表す ID に 16bit 変数を利用しているため、2^16 種類の単語しか表現できず、大規模な言語モデルには適用できない。

SRILM (Stolcke 2002) は、Backward Suffix Tree による言語モデル実装である。Backward Suffix Tree の特性を活かして、高速なバックオフ計算が行えるのが特徴である。Backward Suffix Tree では、単語 ID として 32bit 非負整数型を使っているため比較的大

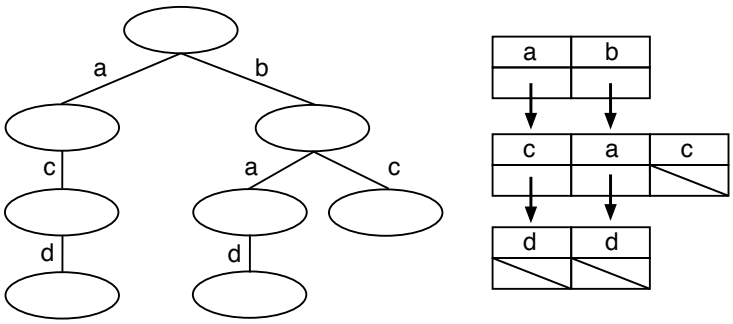

図3.1　Sorted Array による Trie 表現の例。オーダーごとに配列を用意し、子ノードへの遷移先を遷移可能単語と共にポインタで示す。

きな言語モデルに適用できる。ただし、ノード遷移のために 64bit のポインタ変数を使用しているためメモリ消費が大きく、大規模な言語モデルをロードすると容易にメモリ不足に陥る。

3.2　高速かつコンパクトな言語モデルの実装

統計的機械翻訳においては、言語モデルのデータを増加させると翻訳性能も改善することが知られている (Brants et al. 2007)。翻訳性能を確保するため大量の言語リソースを投入すると、言語モデルのエントリ数もそれにしたがって上昇する。エントリ数の上昇はメモリを圧迫するだけでなく、実行速度も低下させる。増え続ける言語モデルのエントリ数に対応するため、できるだけコンパクトにモデルを格納する手法が様々に提案されている。

Talbot and Osborne (2007) は、Bloom Filter を使った言語モデルを提案した。Bloom Filter は、ある集合の中にエントリが含まれているかどうかを調べることができるデータ構造である。Bloom Filter 単体では、言語モデルのパラメータを格納することはできない。この問題を解決するため、この手法ではエントリと出現回数をペアにしてモデルに格納する。具体的には、任意の $m$gram $\boldsymbol{v}_1^m$ について、$< \boldsymbol{v}_1^m, 1 >, \cdots, < \boldsymbol{v}_1^m, c >$ というペアを作りこのペアをモデルに格納する。c は $m$gram の出現回数である。クエリの際は、$m$gram $\boldsymbol{v}_1^m$ の出現回数がわからないため、1 から順に数を増加させてエントリ集合から見つからなくなるまでクエリを繰り返す。確率値は得られた出現回数を元にその場で計算する。

Guthrie and Hepple (2010) は、最小完全ハッシュを用いた言語モデル実装である。最小完全ハッシュ (Belazzougui, Botelho, and Dietzfelbinger 2009) とは、与えられたエントリ集合に対して、衝突なくかつ最小の範囲にハッシュする関数のことである。この手法では、ハッシュ値で示される位置にモデルパラメータとエントリのフィンガープリントを格納する。モデルパラメータは Simple Dense Coding (Fredriksson and Nikitin 2007) によって圧縮した状態で格納する。圧縮率を高めるため、モデルパラメータとして学習データにおける出現回数を格納する。

これまで述べた手法はすべて *lossy* な言語モデルと呼ばれる手法である。モデルのエントリ情報をフィンガープリントといった情報で代替することで、まれに誤った結果を返す。*Lossy* な言語モデルは、コンパクトさのために精度を犠牲にする。

Pauls and Klein (2011) は、Sorted Array を採用した言語モデル実装である。可変長配列と、ブロック圧縮の機能を持ち、実行速度よりもコンパクトさが重要なときに利用できる。

Watanabe, Tsukada, and Isozaki (2009) は、LOUDS (Jacobson 1989) に基づく言語モデルを提案した。LOUDS は簡潔データ構造による Trie の実装方法で、ランダムアクセス性を保ちつつ情報量下限まで圧縮できることが知られている。LOUDS による言語モデルはコンパクトだが、実行速度については調査されていない。

Sorensen and Allauzen (2011) は、LOUDS に基づく言語モデルの別の実装方法である。この手法は言語モデルを Trie ではなく $n-1$ 次マルコフモデルの遷移グラフで表し、このグラフを LOUDS で表現する。

Heafield (2011) は、2 種類の手法を実装した言語モデル実装である。ひとつは Hash

table を利用する方法。もうひとつは、Trie を利用する方法である。この実装は KenLM と呼ばれ、近年最も利用されている。

Hash Table を利用する方法 (KenLM `probing` と呼ぶ) は、従来から広く用いられている Hash Table と Linear probing 法をそのまま言語モデルの格納方法として適用した手法である。ほぼ定数時間でデータにアクセスできる点で優れているが、Table にあらかじめ隙間を開けておく必要がある。これはメモリ消費量を増大させるという欠点がある。

Trie を利用する方法 (KenLM `trie` と呼ぶ) は、Trie と呼ばれる木構造の一種をベースとした手法である。Trie は、木構造のノード遷移を単語で行うことに特徴がある。特に、従来手法では、Sorted Array を用いてノード遷移を行う。Sorted Array はモデルデータをコンパクトに保存することができるが、ノード間遷移が $O(\log \log M)$ であり、実行速度は KenLM `probing` に比べて遅い。

KenLM `probing` は速度において、KenLM `trie` に優れるが、メモリ使用量においては、KenLM `trie` の方が優っている。KenLM `probing` と KenLM `trie` はトレードオフの関係にあるといえる。

従来手法においては、潤沢にメモリが使える環境であれば KenLM `probing` を、そうでない環境では KenLM `trie` を利用することが推奨される。特に、統計的機械翻訳の場合、KenLM `probing` ではなく KenLM `trie` を用いて利用可能メモリ量上限に達するまで学習データを追加するということがしばしば行われる。これは、Brants らによる実験で、言語モデルの学習データを増加させることが翻訳性能を改善することが示されているためである (Brants et al. 2007)。このとき、KenLM `trie` は KenLM `probing` より遅いため、実行速度の面でユーザーは妥協を迫られる。もし、KenLM `trie` とほぼ同じメモリ使用量で、より高速にクエリする方法が存在するなら、ユーザーはそのような妥協をする必要はない。

第 4 章

ダブルアレイの言語モデルへの適応

4.1 概要

本研究では、ダブルアレイを言語モデルに適応する。言語モデルの内部構造として従来から Trie がよく使われるが、Trie の実装には Sorted Array などの他の方式が用いられてきた。ダブルアレイは、高速かつコンパクトな Trie の実装方法として知られており、もし言語モデルの実装にダブルアレイを適用できれば、ダブルアレイの持つ高速・コンパクトという特徴を言語モデルでも享受できる。

言語モデルの内部構造で使われている 2 種類の Trie について、ダブルアレイを適応する方法を提案する。ひとつは Backward Suffix Tree、もう一つは Reverse Trie である。

本章では、それぞれ言語モデルに適応した方法について述べた上で (4.2節、4.3節)、それらについて State 最適化方法を述べる (4.4節)。次に、ダブルアレイのサイズを小さくするためのチューニング手法について述べる (4.5節)。最後に、大規模な言語モデルを構築するために必要な分割手法について述べる (4.7節)。

4.2 Backward Suffix Tree に基づくダブルアレイ言語モデル

Backward Suffix Tree は、通常の Trie と異なり各ノードに目的単語を格納した表が紐付いている。この表はダブルアレイでは表現不可能であるため、ダブルアレイで表現可能な形に変換する。DALM$_{bst}$ では、これを実現するためにエンドマーカーノードを工夫する。

通常のダブルアレイでは、Trie の各ノードについて、それぞれ該当する単語列が実際にモデルに含まれるかのフラグとして、エンドマーカーと呼ばれるノードを各エントリに付加する。Backward Suffix Tree をダブルアレイで表現するために、各エントリの履歴単語列の直後にエンドマーカーノードを挿入する。それらのエンドマーカーノードから、各ノードに対応する目的単語のノード群を付加する。すなわち、従来の Backward Suffix Tree に対して、各ノードにエンドマーカーノードを付与し、更にその下に目的単語ノードを付与する。これによって Backward Suffix Tree は一つの Trie で表現可能となる。図 4.1に、図 2.2と同じ内容を表したダブルアレイで表現可能な Trie を示す。“you eat soup” にアクセスするには、“eat” → “you” → < # > → “soup” と辿る。Backward Suffix Tree が一つの Trie として表現可能となったことで、このデータ構造はダブルアレイを使って表現可能である。

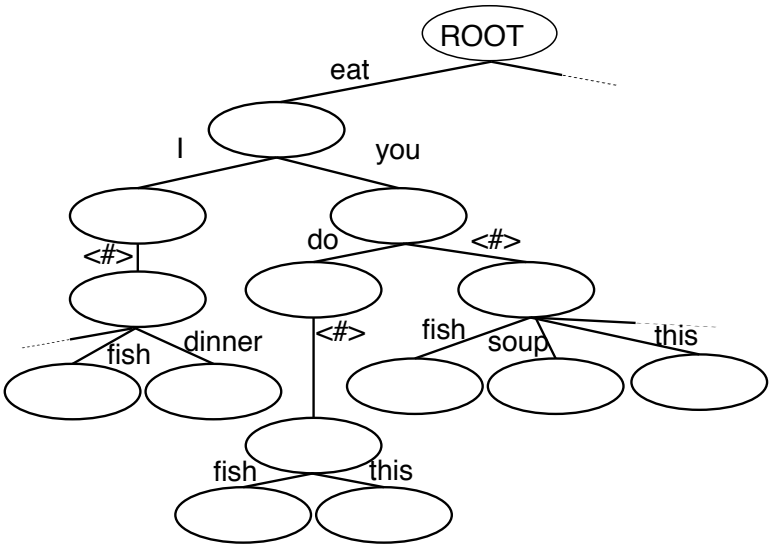

図4.1　Backward Suffix Tree をダブルアレイで表現可能な形に変換する。履歴単語ノードの後にエンドマーカー $< \# >$ を配置し、その直後に目的単語ノードを配置する。

次に、モデルパラメータの格納方法について述べる。言語モデルでは、1 回のクエリに対して 2 つの値 (確率値及びバックオフウェイト) を参照する。確率値は 31bit で表現でき、バックオフウェイトは 32bit で表現できるため、保存すべきデータの総量は 63bit である。従来のダブルアレイは 1 回のクエリに対して 32bit までの値を参照する前提で設計されており、63bit の値を参照できるように構造を修正する必要がある。

ダブルアレイ言語モデルでは、ダブルアレイの未使用領域に着目し、確率値とバックオフウェイトをダブルアレイの内部に組み込んだ。前述のとおり、1 個のエントリに対して 32bit 以下の値を効率よく扱う手法は、従来までにすでに知られている。従来手法では、エントリのエンドマーカーが格納されている位置の BASE 配列部分が未使用であることに着目し、エンドマーカーの BASE 配列部分に値を格納する[*1]。

まず、確率値の格納について述べる。本研究では、エンドマーカーノードの子ノードとして目的単語ノードが存在するため、エンドマーカーノードの BASE 配列位置に値を格納することはできない。ただし、従来手法と同様に目的単語ノードの BASE 配列位置が未使用である。そこで、本研究ではこの未使用領域に対応するエントリの確率値を格納する。

次に、バックオフウェイトの格納について述べる。言語モデルに含まれるエントリには、単語列 $\boldsymbol{v}_1^m$ がモデルに含まれるとき、それ以下の長さの単語列もモデルに含まれるという性質がある[*2]。したがって、本研究で利用する Trie はすべてのノードがエンドマーカーを持つ。すなわち、ダブルアレイに必要な CHECK 配列による遷移チェックはエンドマーカーノードに対しては必要ない。そこで、エンドマーカーの CHECK 配列側にバックオフウェイトを格納する。誤遷移を防止するため、CHECK 配列側にモデルパラメータを格納する際、それらは負の数である必要がある。バックオフウェイトは一部正の数になりう

∗1. 工藤による Darts(Double Array Trie System, `http://chasen.org/~taku/software/darts/`) でこの手法が実装されている。

∗2. この性質はカットオフ値を低次から高次に単調増加させた場合に限る。ただし、先行研究の KenLM ではカットオフ値を単調増加させない場合にエラーとなるように実装されており、事実上この制約は保証されているとみなして良い。

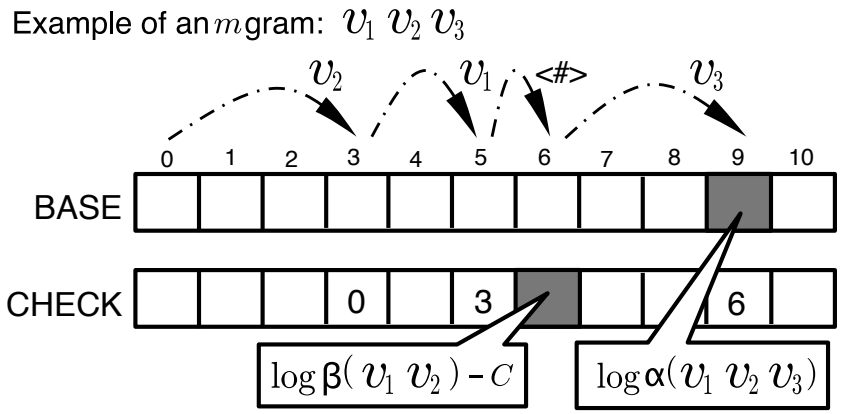

図4.2 Backward Suffix Tree に基づくダブルアレイ言語モデルの例。履歴単語を逆順に並べ、エンドマーカーを挟んで目的単語を配置する。C は、格納する値が負の数であることを保証するための定数である。

るため、モデルに含まれるバックオフウェイトの最大値より大きい数で全体のバックオフウェイトから減算しておくことで、負の数となることを保証する。ダブルアレイ言語モデルの例を図 4.2に示す。

DALM$_{bst}$ から、モデルパラメータを取り出す手順をアルゴリズム 10に示す。取り出されたバックオフウェイトは引数に与えた単語列よりも 1 単語短くなっていることに注意が必要である。

4.3 Reverse Trie に基づくダブルアレイ言語モデル

次に、Reverse Trie に基づくダブルアレイ言語モデル (DALM$_{rev}$ と呼ぶ) を提案する。DALM$_{rev}$ では、言語モデルのエントリ集合を Reverse Trie で表し、各ノードの下にエンドマーカーを配置する。DALM$_{bst}$ と同様に、言語モデルの全ノードにエンドマーカーが付与されるため、各ノードの表す単語列が言語モデルに含まれているか確認する必要はない。したがって、本研究ではエンドマーカーノードをモデルパラメータの格納に転用する。

最上位オーダーの単語列を表すノードについては、それに付加されるエンドマーカーは必ず兄弟ノードを持たない。ダブルアレイにおいて、兄弟ノードを持たないノード (シングルノードと呼ぶ) はダブルアレイ上の任意の場所に配置することができ、ダブルアレイの充填率を上昇させることが知られている (大野将樹 et al. 2003)。エンドマーカーを持たない場合のダブルアレイと比較すると、エンドマーカーを持つダブルアレイはエンドマーカーを持たない場合にできていた隙間にこれらのシングルノードを挿入することができる。したがって、エンドマーカーを使わない場合に無駄になっていた領域をパラメータ格納に使うことができるため、メモリ空間をより効率的に利用することができる。

DALM$_{rev}$ では、CHECK 配列側に対数確率、BASE 配列側に対数バックオフウェイトを格納する。CHECK 側に正の数を格納してしまうと、Trie の遷移を誤る可能性があることからこの組み合わせで格納する必要がある。ダブルアレイの CHECK 値は、遷移元のインデックスを意味しており、遷移元のインデックスと区別がつかなくなるためであ

アルゴリズム 10: DALM$_{bst}$ からモデルパラメータを取り出す手順

Input: $\boldsymbol{v}_1^m$: m-gram.
Output: $\log \alpha\left(\boldsymbol{v}_1^{\boldsymbol{m}}\right), \log \beta\left(\boldsymbol{v}_1^{m-1}\right)$
begin
  $node \leftarrow 0$
  for $i \leftarrow m-1$ ***to*** 1 ***step*** -1 **do**
    $next \leftarrow BASE[node] + WORDID(v_i)$
    if $CHECK[next] = node$ **then**
      $node \leftarrow next$
    else
      return *Not Found*
  $node \leftarrow BASE[node] + WORDID(< \# >)$
  $bow \leftarrow$ reinterpret $CHECK[node]$ as a floating-point number
  $bow \leftarrow bow + C$
  $next \leftarrow BASE[node] + WORDID(v_m)$
  if $CHECK[next] = node$ **then**
    $prob \leftarrow$ reinterpret $BASE[next]$ as a floating-point number
    return $prob, bow$
  else
    return *Not Found*

る。対数確率は必ず負の数となるため、このように格納することでエンドマーカーノードの CHECK 値は必ず負の数になり、この問題を解消できる。Reverse Trie に基づく言語モデルの例を図 4.3に示す。言語モデルの各エントリは、単語列の逆順に登録し、最後にエンドマーカーを付与する。

アルゴリズム 11にここまで述べたデータ構造から値を取り出す手順を示す。DALM$_{bst}$ と異なり、引数の単語列に対応するバックオフウェイトを返すことができる。

4.4　State 最適化の対応

2.4.5節で述べた通り、言語モデルには State の最適化が求められる。最適化に必要な情報は 2 つある。一つは各エントリが右に伸びるか否かであり、もう一つは各エントリが左に伸びるか否かである。従って、各エントリに対して、確率値、バックオフウェイトに加え、もう 2 ビットの情報を付加する必要がある。

DALM$_{bst}$ の拡張性情報格納方法について述べる。右拡張性は、ヒストリ単語列を表すスロットの BASE 値に格納する。ダブルアレイの BASE 値は必ず正の数という制約があるため、各スロットの符号ビットが未使用となっているため、各ノードの BASE 値の符号に右拡張性の情報を格納した。第 2.4.5節で述べたとおり、右拡張性を考慮するのは対数バックオフウェイトが 0 であるときだけである。この情報は常にバックオフウェイトと共に扱われるため、バックオフウェイトと関連の深いこの位置に格納した。左拡張性は確

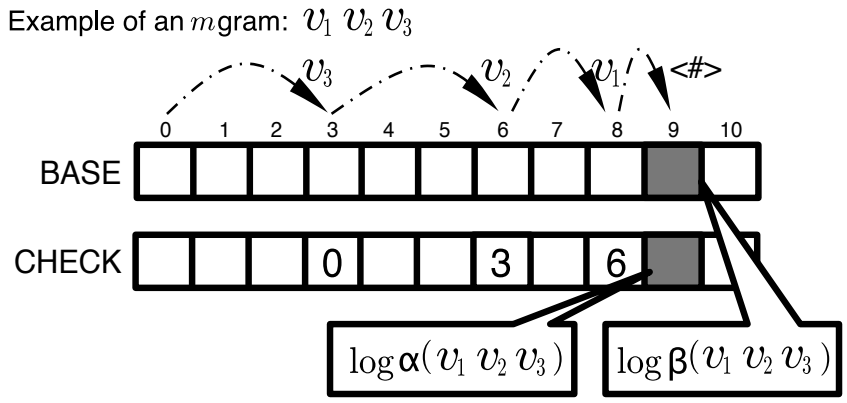

図4.3 Reverse Trie に基づくダブルアレイ言語モデルの例。エントリの単語列を逆順に格納する。単語列の最後にエンドマーカーを付与し、BASE 側に対数バックオフウェイト、CHECK 側に対数確率を格納する。

アルゴリズム 11: DALM$_{rev}$ からモデルパラメータを取り出す手順

Input: $\boldsymbol{v}_1^m$: m-gram.
Output: $\log \alpha\left(\boldsymbol{v}_1^{\boldsymbol{m}}\right), \log \beta\left(\boldsymbol{v}_1^m\right)$
begin
  $node \leftarrow 0$
  for $i \leftarrow m$ ***to*** 1 ***step*** -1 **do**
    $next \leftarrow BASE[node] + WORDID(v_i)$
    if $CHECK[next] = node$ **then**
      $node \leftarrow next$
    else
      return *Not Found*
  $node \leftarrow BASE[node] + WORDID(< \# >)$
  $prob \leftarrow$ reinterpret $CHECK[node]$ as a floating-point number
  $bow \leftarrow$ reinterpret $BASE[node]$ as a floating-point number
  return $prob, bow$

率値を格納するスロットの符号ビットを利用する。対数確率は必ず負の数であるため、符号ビットが未使用である。ここに、その確率値を持つエントリの左拡張性を格納する。対数確率値の符号ビットを利用するのは、KenLM `probing` と同一である。図 4.4に右拡張性情報の格納方法を示す。

次に、DALM$_{rev}$ の拡張性格納方法について述べる。DALM$_{rev}$ では、右拡張性、左拡張性の各情報を、バックオフウェイトと共に格納する。表 4.1に拡張性情報の格納方法を示す。対数バックオフウェイトは、正の数、負の数両方になりうる。そのため、単純な方法ではこのスロットの符号ビットは利用できない。そこで、ここに格納する値が必ず負になるように、あらかじめ対数バックオフウェイトから一定数 C を減算しておく。こうすることで対数バックオフウェイトを格納するスロットに 1 ビットの余裕ができるため、こ

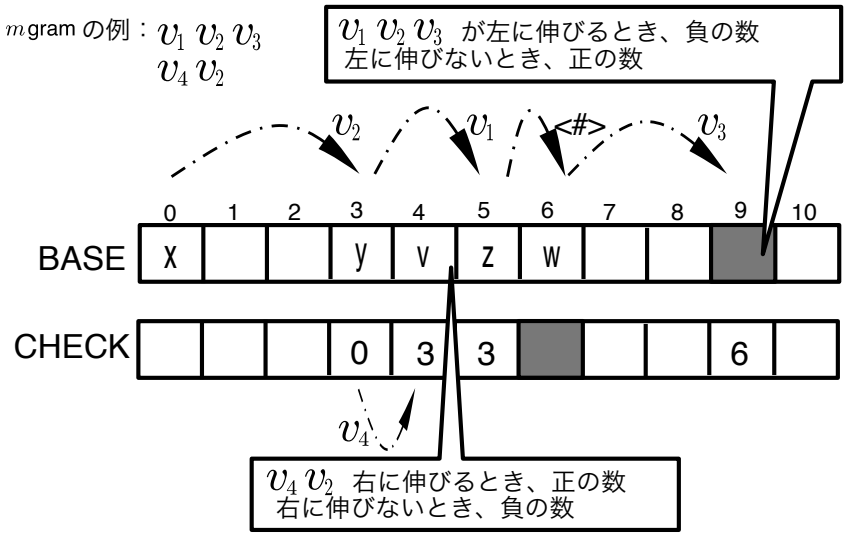

図4.4　DALM$_{bst}$ における拡張性情報の格納。右拡張性の情報を、ヒストリ単語列を表すスロットの BASE 値に格納する。BASE 値は必ず正の数であるという制約があるため、符号ビットが空いている。左拡張性の情報は、各エントリの確率値と共に保存する。対数確率は必ず負の数であるため、符号ビットが空いている。確率値の符号ビットを利用するのは、KenLM `probing` と同じである。

表4.1　DALM$_{rev}$ における拡張性情報の格納。DALM$_{rev}$ では、左右拡張性の情報をバックオフウェイトと共に格納する。対数バックオフウェイトは、正の数にも負の数にもなり得るため、負の数を保証するために一定数を減算する。空いた符号ビットに左拡張性の有無を格納する。右拡張性を利用するのは対数バックオフウェイトが 0 のときのみであることから、右に伸びないときは無限大を格納する。

		左に伸びる	
		Yes	No
右に伸びる	Yes	負の数	正の数
	No	$-\infty$	$+\infty$

こに左拡張性の情報を格納する。DALM$_{bst}$ と同様、右拡張性については、対数バックオフウェイトが 0 になる時だけ判定すれば良いため、0 の代わりに ∞ を格納して右に伸びないことを記録する。

4.5　チューニング

4.5.1　Leaf Compression

DALM$_{rev}$ のみで利用可能なチューニング手法について述べる。言語モデルにおいて、最上位オーダーのエントリはバックオフウェイトを持たないため、この性質を利用してメモリ使用量を削減する。

単純に Trie の形を Reverse Trie に変えただけでは、言語モデルのサイズはほぼ変わらない。ただし、最上位オーダーについては、バックオフウェイトを持たず、確率値のみ格

納できれば良い。この場合、格納すべき値は 1 つだけになるため、従来手法が適用可能である。本研究では Reduced double-array trie (Yoshinaga and Kitsuregawa 2014) の手法を適用した。

言語モデルにおける最上位オーダーには、エンドマーカー以外の兄弟ノードが存在しないという性質がある。そこで、該当の $n$gram については、エンドマーカーを削除し、その手前ノードの BASE 配列側に対応する確率値を格納する。これにより Trie のノード数が減少し、使用メモリ量を削減することができる。

厳密には、Leaf Compression が適用できるのはエントリが以下の条件を満たす時である。

- エントリがバックオフウェイトを持たない (対数バックオフウェイトが 0 である)
- エントリが左に伸びない
- エントリが右に伸びない

エントリがバックオフウェイトを持つとき、エントリに対して持つべき値が 2 つ (合計 63bit) になる。2 つの値を BASE 側に格納することはできないため、Leaf Compression に該当しない。

エントリが左に伸びるとき、Reverse Trie では該当エントリを表すノードにエンドマーカー以外の子ノードが存在することを意味する。これはダブルアレイにおいては、BASE 値を消費してしまうパターンであるため、Leaf Compression に該当しない。

第 2.4.5節で述べたとおり、エントリが右に伸びず、かつ対数バックオフウェイトが 0 のとき、クエリ時返す State をトランケートする必要がある。したがって、すべてのエントリはこの条件をみたすかどうか判定できるようにしておく必要がある。本研究では、Leaf Compression されたものはこの条件を満たすものとして扱い、この条件を満たさないものは Leaf Compression の対象から外すことで判別できるようにした。

次に、クエリ時の Leaf Compression 対応について述べる。クエリの際、入力された $m$gram の各単語について逆順にノードを辿る。各ノードから子ノードへの遷移処理を行う前に、遷移元の BASE 値の符号を確認する。符号が負の数のとき、Leaf Compression されていると認識し、遷移元の BASE 値は確率値であるとして取り扱う。また、上記の Leaf Compression 適用の条件より、該当ノードが指し示す単語列は右にも左にも伸びず、またバックオフウェイトも持たないとして処理を進めれば良い。Leaf Compression されたノードにあたった時は、該当ノードからのさらなるノード遷移は不可能なので、探索もその時点で打ち切る。

アルゴリズム 13に、これまで述べた手法に基づいて、モデルパラメータを取り出す手順を示す。アルゴリズム内で利用している関数 get_endmarker はアルゴリズム 12に示す。Leaf Compression が適用されているノードの場合、処理が大幅に簡略化される。Leaf Compression が適用されないノードは、State 最適化のためやや複雑な条件分岐を必要とする。

4.5.2 単語 ID チューニング

ダブルアレイを構築する際、単語 ID をチューニングすることでダブルアレイのサイズをコンパクトにすることができる。Code Mapping Methods (Liu et al. 2011) を利用し

アルゴリズム 12: ノードに紐付いたエンドマーカーの位置を取得する手順 (get_endmarker, DALM$_{bst}$, DALM$_{rev}$ 共通)

Input: Trie のノードを表すダブルアレイ配列上のインデックス c
Output: 引数に与えられたノードに付随するエンドマーカーの位置
begin
    // すべてのノードがエンドマーカーを持つため、遷移に CHECK 値の確認は不要
    return $BASE[c] + 1$

アルゴリズム 13: DALM$_{rev}$ においてモデルパラメータを取得する手順 (get_params)

Input: ノードのダブルアレイ上の位置 c
Output: 確率値 *prob*
Output: バックオフウェイト *bow*
Output: 左拡張性の有無 *left* (true のとき左に伸びる)
Output: 右拡張性の有無 *right* (true のとき右に伸びる)
begin
    if $BASE[c] < 0$ **then**
        // Leaf Compression
        prob ← $BASE[c]$ を浮動小数点数として解釈した値
        bow ← 0
        left ← false
        right ← false
    else
        endmarker ← get_endmarker (c)
        prob ← $CHECK[endmarker]$ を浮動小数点数として解釈した値
        bow ← $BASE[endmarker]$ を浮動小数点数として解釈した値
        left ← $bow < 0$
        if $bow = +\infty$ または $bow = -\infty$ **then**
            right ← false
            bow ← 0
        else
            right ← true
            bow ← $-\text{abs}\,(bow) + C$
    return *prob*, *bow*, *left*, *right*

てチューニングする。ダブルアレイ構築は、2.6節で述べた通り、疎行列表現で表されたTrieを1行ずつずらしながら構築を進めるが、この時のずらし幅を全体的に小さくすることがこの手法の目的である。

ダブルアレイの単語IDをチューニングしない場合、Trieを疎行列表現すると、遷移の存在を示す数字が行列全体にまばらに分布する。この疎行列の各行を、列方向に値が重ならないようにずらすと、ある程度ずらしただけではずらした先にノード遷移が存在する可能性が高く、結果としてずらし幅が大きくなる。疎行列のずらし幅が大きくなると、最終的に構築されるダブルアレイが大きくなることに直結する。

もし、少ないずらし幅で衝突の可能性を減らすことができれば、結果として完成したダブルアレイのサイズが小さくなることが期待できる。本研究においては、これを実現するため単語IDをUnigram確率の降順に付与する。単語IDをUnigram確率順にすることで、疎行列表現におけるノードの存在位置が前方による事になる。したがって、ノードを挿入したいときでも、少ないずらし幅で衝突可能性を大きく減らすことができる。この手法により、モデルサイズのコンパクト化が期待できる。

4.6 単語の出現確率を求める手順

4.6.1 DALM$_{bst}$

DALM$_{bst}$ におけるStateと目的単語から出現確率を求める手順をアルゴリズム16に示す。ただし、アルゴリズム16内で利用している各関数は、それぞれアルゴリズム14、15に定義する。

DALM$_{bst}$ では、クエリ時にTrieを2度辿る必要がある。クエリ時にState最適化を実行する必要があるが、右拡張性の情報と確率値とが別のところに格納されているためである。Trieを2度辿る速度低下を軽減するため、DALM$_{bst}$ では各単語列を表すダブルアレイ配列上のインデックスを単語列とともにStateに格納する。バックオフウェイトはStateに格納しない。これにより、クエリ時の目的単語ノードを探すコストを抑えた。

Backward Suffix Treeは単発クエリに強いが、Stateの適用には課題が残る。仮にStateの最適化を断念すると、機械翻訳においてはリコンバインの効率が悪化し、翻訳精度が下がる。本研究では、翻訳精度を維持することを優先した。

4.6.2 DALM$_{rev}$

DALM$_{rev}$ において、Stateと目的単語から確率値を求める手順をアルゴリズム17に示す。与えられた単語列を逆順に辿りながら、それぞれの時点での確率値及びバックオフウェイトを取得しつつ計算を進める。Reverse Trieでは確率値計算の過程と同時にStateも生成できていることがこのアルゴリズムからもわかる。

4.7 ダブルアレイの分割構築

ダブルアレイの構築は多くの計算を必要とすることが知られている。中村康正 and 望月久稔 (2006) によると、ノードを一つ挿入するのに要する時間は、ダブルアレイの空き要素数をU、言語モデルのUnigram単語集合を$|V|$とすると、$O(U|V|)$である。ダブル

アルゴリズム 14: DALM$_{bst}$ においてノードに紐付いた確率値を取得する手順 (get_prob)

Input: 単語 v
Input: エンドマーカーの位置 *endmarker*
Output: 確率値 *prob*
begin
  $next \leftarrow BASE[endmarker] + v$
  if $endmarker = CHECK[next]$ // 単語 v での遷移が成功
  then
    $prob \leftarrow BASE[next]$ を浮動小数点数として解釈した値
    return *prob*
  else
    return *Not Found*

アルゴリズム 15: DALM$_{bst}$ において、ノードに紐付いたバックオフウェイトを取得する手順 (get_bow)

Input: エンドマーカーのダブルアレイ配列上のインデックス *endmarker*
Output: バックオフウェイト *bow*
begin
  $bow \leftarrow CHECK[endmarker]$ を浮動小数点数として解釈した値
  // 誤遷移防止のため、バックオフウェイトは定数を減算して必ず負の数にして格納される。読み出す際は、同じ定数を加算して元の値に戻す必要がある。
  $bow \leftarrow bow + C$
  return *bow*

アレイの空き要素数が挿入回数に比例すると仮定すると、ノード数 M 回の挿入にかかる時間は、$\sum_{i=1}^{M} O\left(i|V|\right) = O\left(M^2|V|\right)$ となる。

言語モデルのサイズが増大すると、ダブルアレイの構築は大変難しくなる。この問題を緩和するため、本研究ではダブルアレイを分割した。ダブルアレイを分割することで、一つ一つに含まれる Trie に含まれるノード数が減るため、構築時間が短くなる。d 分割によって、1 つの Trie の大きさが $\frac{1}{d}$ になったとすると、構築時間は $\frac{1}{d^2} \times d = \frac{1}{d}$ になることが期待できる。また、Trie を分割することで、それらを平行して構築することができるため、さらなる速度改善が期待できる。

DALM では、ダブルアレイの BASE 配列を 4Bytes の配列として定義する。それらの最左ビットは左右の拡張性情報等で利用するため、1 つのダブルアレイに格納可能なノード数は $2^{31} - 1$ 個が上限となる。分割ダブルアレイはこの制約も緩和する。すなわち、分割された各 Trie に含まれるノード数は、元の Trie のノード数より小さくなるため、より大きな言語モデルを構築できる。

分割には、Trie の 1 段目のノードの単語 ID を分割数で割った剰余を用いる。ただし、DALM$_{bst}$ の Unigram 単語に関しては、それらの単語 ID を分割数で割った剰余で分け

アルゴリズム 16: DALM$_{bst}$ における単語列の出現確率を求める手順

Input: 目的単語 v_m
Input: State に含まれる単語列 $\boldsymbol{v}_1^{m-1}$
Input: State の各単語列を表す Trie ノードの位置 $\boldsymbol{c}_1^{m-1}$
Result: 対数確率 $\log P\left(v_m \mid \boldsymbol{v}_1^{m-1}\right)$
Result: 新しい State の単語列 $\boldsymbol{u}_1^{m'}$
Result: 新しい State の各単語列を表す Trie ノードの位置 $\boldsymbol{d}_1^{m'}$
begin
  // State の情報を元に、確率値を計算
  $prob \leftarrow 0$
  for $i \leftarrow 1$ *to* $m-1$ **do**
    $endmarker \leftarrow \text{get\_endmarker}\left(c_i\right)$
    $p \leftarrow \text{get\_prob}\left(v_m, endmarker\right)$
    if 確率値が見つかった場合 **then**
      $prob \leftarrow p$
      break
    else
      $prob \leftarrow prob + \text{get\_bow}\left(endmarker\right)$
  // 次回クエリのために新しい State を作る
  $context \leftarrow 0$
  $m' \leftarrow 0$
  for $i \leftarrow m$ *to* 1 *step* -1 **do**
    $next \leftarrow BASE[context] + v_i$
    if $CHECK[next] = context$ **then**
      $context \leftarrow next$
      $endmarker \leftarrow \text{get\_endmarker}\left(context\right)$
      $d_{m-i+1} \leftarrow context$
      $u_{m-i+1} \leftarrow v_i$
      if $\boldsymbol{v}_i^m$ が右に伸びるとき // State 最適化
      then
        $m' \leftarrow m - i + 1$
    else
      break
  $\boldsymbol{d}_1^{m'}$ を逆順にする
  $\boldsymbol{u}_1^{m'}$ を逆順にする
  return $prob$, $\boldsymbol{d}_1^{m'}$, $\boldsymbol{u}_1^{m'}$

アルゴリズム 17: DALM$_{rev}$ における単語列の出現確率を求める手順

Input: 目的単語 v_m
Input: State に含まれる単語列 $\boldsymbol{v}_1^{m-1}$
Input: $\boldsymbol{v}_1^{m-1} \sim \boldsymbol{v}_{m-1}^{m-1}$ のそれぞれに対応するバックオフウェイト $\boldsymbol{b}_1^{m-1}$
Result: 対数確率 $\log P\left(v_m \mid \boldsymbol{v}_1^{m-1}\right)$
Result: 新しい State の単語列 $\boldsymbol{u}_1^{m'}$
Result: $\boldsymbol{u}_1^{m'} \sim \boldsymbol{u}_{m'}^{m'}$ のそれぞれに対応するバックオフウェイト $\boldsymbol{a}_1^{m'}$

```
begin
    prob ← 0
    context ← 0
    for i ← m to 1 step −1 do
        next ← BASE[context] + v_i
        if check[next] = context then
            context ← next
            prob, a_{m−i+1}, left, right ← get_params(context)
            u_{m−i+1} ← v_i
            if v_i^m が右に伸びるとき                          // State 最適化
             then
                m′ ← m − i + 1
        else
            for j ← i to 1 step −1 do
                prob ← prob + b_j                              // b_m = 0 とする
            break
    a_1^{m′} を逆順にする
    u_1^{m′} を逆順にする
    return prob, a_1^{m′}, a_1^{m′}
```

る。この方法は、短い計算時間で各エントリの格納先ダブルアレイを求めることができる点で優れているが、Trie の 1 段目の単語が超頻出単語の場合、その下に続く Trie が巨大なものになりうる点で欠点を抱えている。これは分割後の各 Trie のノード数にばらつきを与えるため、幾つかの Trie だけ構築が遅くなることや、ダブルアレイに格納可能なノード数上限に到達しやすくなるなどの問題を引き起こす。この観点から、ダブルアレイの分割法はまだ改善の余地があり、今後の研究が必要である。

第 5 章

実験

5.1　実験の概要

本研究で提案した手法の有効性を締めするため、実験を行う。実験は以下の 3 つから構成される。

- 構築実験 DALM の構築時間が、データサイズと分割数によってどう影響されるか調べる。また、先行研究での手法との構築時間を比較する。
- パープレキシティ測定実験クエリ速度とモデルサイズの関係を調べる。この実験は、言語モデルだけの純粋な性能を見ることが目的である。
- 翻訳実験翻訳速度とメモリ使用量の関係を調べる。この実験は、言語モデルが使われる実環境においてどのような性能を示すか見ることが目的である。

2 番目と 3 番目の実験は似ているが、2 番目のパープレキシティ測定実験は人間の記述したテキストを言語モデルの入力とするのに対し、3 番目の翻訳実験は翻訳システムが生成した非文を含むテキストを言語モデルの入力とするという違いがある。したがって翻訳システムにおいてはバックオフが多く発生すると考えられ、独立した実験を行う意義がある。

5.2　実験環境

実験においては、NTCIR10 特許翻訳タスク (Goto et al. 2013) を用いる。実験に用いた言語モデルは日本語データを用いて学習した。

DALM のスケーラビリティを確認するため、4 つの言語モデルを用いる。これらのモデルは KenLM (Heafield 2011) により学習されたものであり、今後は小さいものから順に Tiny, Small, Mid, Large と呼ぶ。これらのコーパスとモデルの仕様を表 5.1に示す。NTCIR10 特許翻訳タスクのデータは複数年に渡る特許文を含んでおり、本研究においてこのデータをコーパスとして用いる際は、半年単位で区切って利用した。

パープレキシティ測定実験においては、コーパスの 2005 年のデータをテストセットとした。テストセットは、4,765,362 文からなり、総単語数は 261,341,268、単語の種類数は 720,800 である。

翻訳実験においては、NTCIR10 で用意された対訳ペアを学習データ・テストデータとして用いた。翻訳モデルは英日翻訳のモデルを用意し、階層フレーズモデル (Chiang 2007)

表5.1 コーパスとモデルの仕様

	Tiny	Small	Mid	Large
データの出典	2004 年 後半	2004 年	2002 年 から 2004 年	1999 年 から 2004 年
ディスク上の ファイルサイズ (ARPA 形式)	12 GB	22 GB	51 GB	85 GB
総単語数	1,274,687,871	2,596,089,299	7,098,400,326	12,966,811,114

	Tiny	Small	Mid	Large
1gram 数	1,275,423	2,001,505	4,255,055	6,408,076
2gram 数	13,400,024	20,509,170	39,663,125	58,477,327
3gram 数	43,058,742	71,389,459	147,400,329	226,567,483
4gram 数	96,977,172	174,447,270	400,258,370	657,349,181
5gram 数	136,732,280	259,325,582	640,413,474	1,100,976,293
Total	291,443,641	527,672,986	1,231,990,353	2,049,778,360

表5.2 翻訳実験で用いたコーパスの仕様

データの種類	文数	単語総数	単語種類数
学習 (英語)	3,186,284	108,093,438	279,575
学習 (日本語)	3,186,284	118,287,534	260,372
チューニング (英語)	2,000	68,507	5,032
チューニング (日本語)	2,000	74,529	4,850
テスト (英語)	2,300	80,066	6,403
テスト (日本語)	2,300	88,317	6,120

を学習した。表 5.2にデータの仕様を示す。翻訳モデルのチューニングでは、KenLM を用いてチューニングした。言語モデルは 4 種類あるため、4 種類のチューニングを行い、それぞれのチューニング結果をそれぞれの実験で用いる。翻訳実験には Moses decoder (Koehn et al. 2007) を用いた。Moses decoder にはテストデータに出現しないフレーズを除去することでフレーズテーブルを削減する機能があるが、実際のアプリケーションでのパフォーマンスを見るためこの機能は利用していない。

先行研究として比較するのは KenLM (Heafield 2011) である。KenLM には `probing` 法と `trie` 法の 2 種類が実装されているため、両方比較する。また、KenLM `probing` にはパラメータ p があり、この値でハッシュテーブルにどの程度の隙間を作るかをコントロールできる。これはモデルサイズに直結するため、本実験では $p = 1.2, 1.5, 2.0$ の 3 パターンを比較する。

すべての実験は Linux サーバー上で行う。CPU は Intel® Xeon® X5672 3.20 GHz で、8 コア持つ。OS から認識されているメモリサイズは 141GB である。

構築時間測定実験においては、各実験の前に 1 度サーバーを再起動し、初回起動時の時

間を測定する。その他の実験では、測定プログラムを 2 回続けて起動し、2 度目の速度を正式結果として採用した。DALM は C++ で実装した。単語の文字列表現から ID への変換は darts-clone ライブラリ[*1]を利用した。

5.3 構築実験

DALM の構築時間を測定・比較する実験を行った。この実験の目的は以下の通りである。

- KenLM のモデル構築速度と比較する
- 各手法の構築時間を比較する
- 分割数が構築時間に与える影響を調べる
- 分割数がモデルサイズに与える影響を調べる
- 2 種類のチューニング法がモデルサイズに与える影響を調べる

構築実験においては、CPU 時間と実時間を測定した。DALM の構築は並列で行うが CPU 時間はこれらの合計時間を示す。また、CPU 時間にはディスク I/O 時間は含まれない。実時間は構築の際に利用者が実際に待つ時間であり、並列化による効率化やディスク I/O のオーバーヘッド時間などを含む。

KenLM でのモデル構築には、Moses decoder release 3.0 に含まれる KenLM 構築プログラムを利用した。KenLM のモデル構築には `mmap` と `after` という 2 つのオプションがあるが、これらについては両方測定した。`mmap` オプションははじめにディスク上にモデルファイルを確保し、mmap の機能を使いながらモデルを構築する。`after` オプションは、メモリ上ですべてモデルを構築してからファイルに出力する。DALM のモデルの構築においては、最大 8 スレッドを利用した。これはサーバーのコア数が 8 個だからである。単語の文字列表現から ID への変換は、構築初期の前処理部分で変換される。

はじめに、モデルサイズ別に DALM と KenLM の比較を行う。図 5.1と表 5.3に構築時間の結果を示す。KenLM のモデル構築において、`after` の方が良い性能を示したので、結果は `after` のみを掲載した。また、`probing` 法の p は 1.5 を用いた。DALM の分割数は 16 である。これらの結果から、DALM の構築時間は指数関数的ではないが、モデルサイズが大きくなるにつれ増大している。モデルサイズによって構築時間の増大は免れないが、実時間は CPU 時間より短くなった。これは Trie の分割手法によって並列化が可能になった事による。同じモデルに対する KenLM の構築時間は DALM の構築より速い。これはダブルアレイの構築において BASE 値を探索する時間に計算量を多く必要とする事による。

次に、Trie の分割数が構築時間・充填率にどう影響されるかを調べた。第 4.7節では、計算量の見積もりに以下の仮定を置いた。

- 未使用スロット数は挿入されたノードの数に比例する
- 分割によりできたそれぞれの Trie の大きさは、それぞれ等しい

*1. https://code.google.com/p/darts-clone/

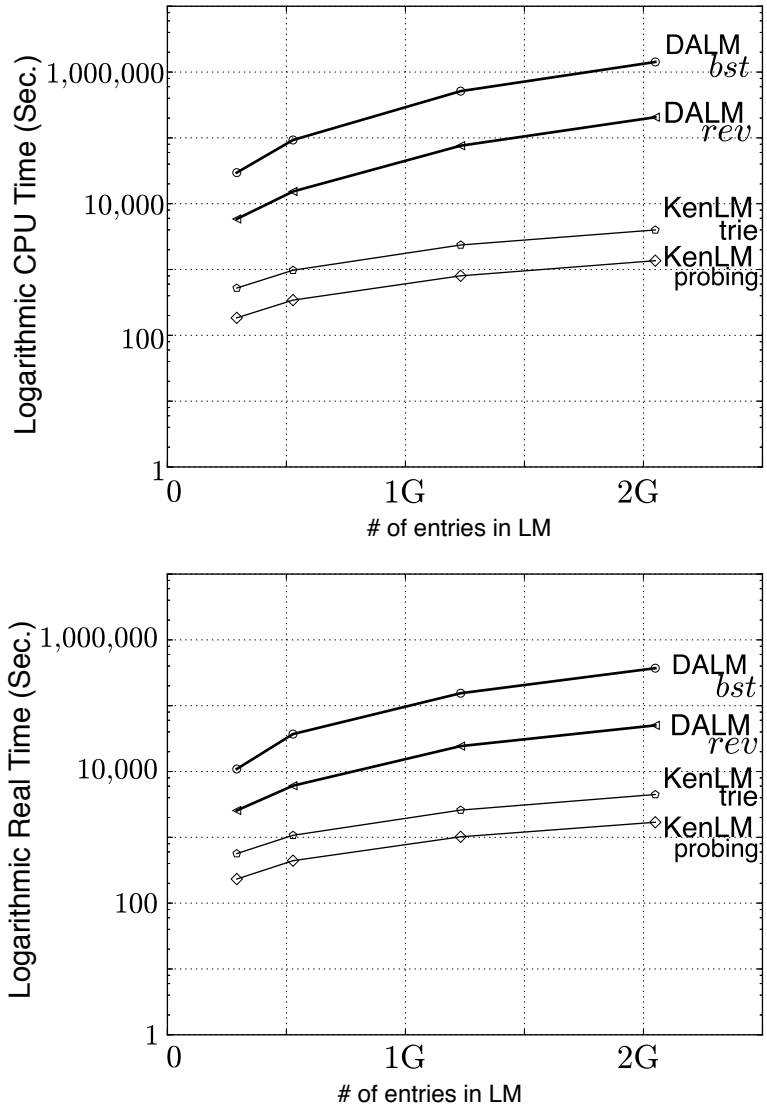

図5.1　モデルサイズ別の構築時間比較。DALM の分割数は 16 に固定した。KenLM `probing` においては $p = 1.5$ である。上部に CPU 時間、下部に実時間を示す。

現実には、Trie の分割は未使用スロット数に影響を与えるだけでなく、分割された Trie はそれぞれ異なる大きさとなる。この実験では現実の言語モデルに対して Trie の分割がどのような影響を与えるか確認する。図 5.2に充填率、図 5.3に DALM の構築時間を示す。また、表 5.4に各手法の分割数、構築時間、充填率を示す。

分割数を増加させると充填率は下がった。DALM$_{bst}$ については構築時間の高速化で得られるメリットを考えると、充填率はあまり問題にならないと考える。DALM$_{rev}$ については、DALM$_{bst}$ に比べると充填率の下がり方が小さい。これは DALM$_{rev}$ のノード数が DALM$_{bst}$ より小さいためである。ダブルアレイの構築時間は挿入ノード数に影響されるが、ノード数が少ないと構築は高速化される。構築が高速であるということは BASE 値がより早い段階で見つかっているということであり、これはダブルアレイの配列サイズの減少に直結する。CPU 時間と実時間を比較すると、並列処理がうまく機能していること

表5.3 モデルサイズ別の構築時間比較。DALM の分割数は 16 に固定した。KenLM `probing` においては $p = 1.5$ である。

手法	言語モデル	CPU 時間 (Sec.)	実時間 (Sec.)
DALM$_{bst}$	Large	1,423,898	371,415
DALM$_{rev}$		205,924	50,234
KenLM `probing`		1,362	1,694
KenLM `trie`		3,991	4,472
DALM$_{bst}$	Mid	512,994	154,699
DALM$_{rev}$		76,070	24,314
KenLM `probing`		802	1,019
KenLM `trie`		2.358	2,592
DALM$_{bst}$	Small	92,893	37,064
DALM$_{rev}$		15,334	6,089
KenLM `probing`		343	442
KenLM `trie`		975	1,074
DALM$_{bst}$	Tiny	29,650	10,911
DALM$_{rev}$		5,858	2,558
KenLM `probing`		184	232
KenLM `trie`		522	566

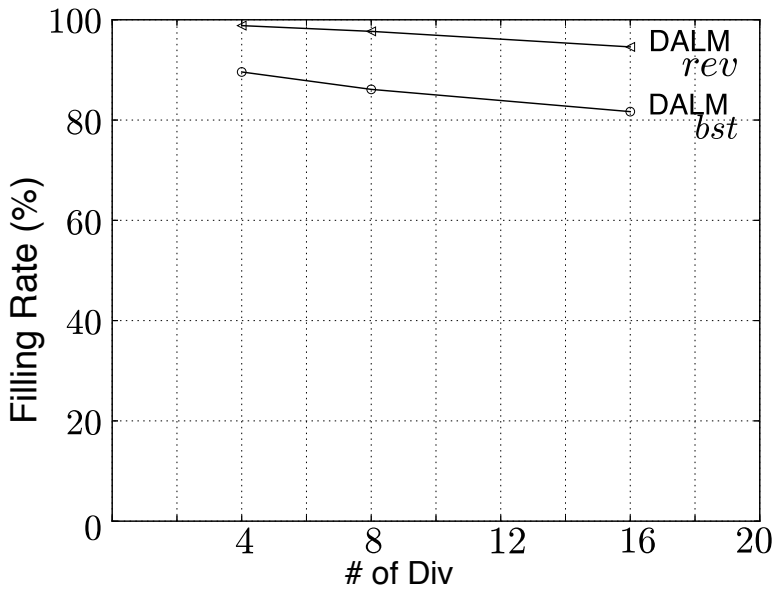

図5.2 分割数による手法ごとの充填率 (Tiny モデル)

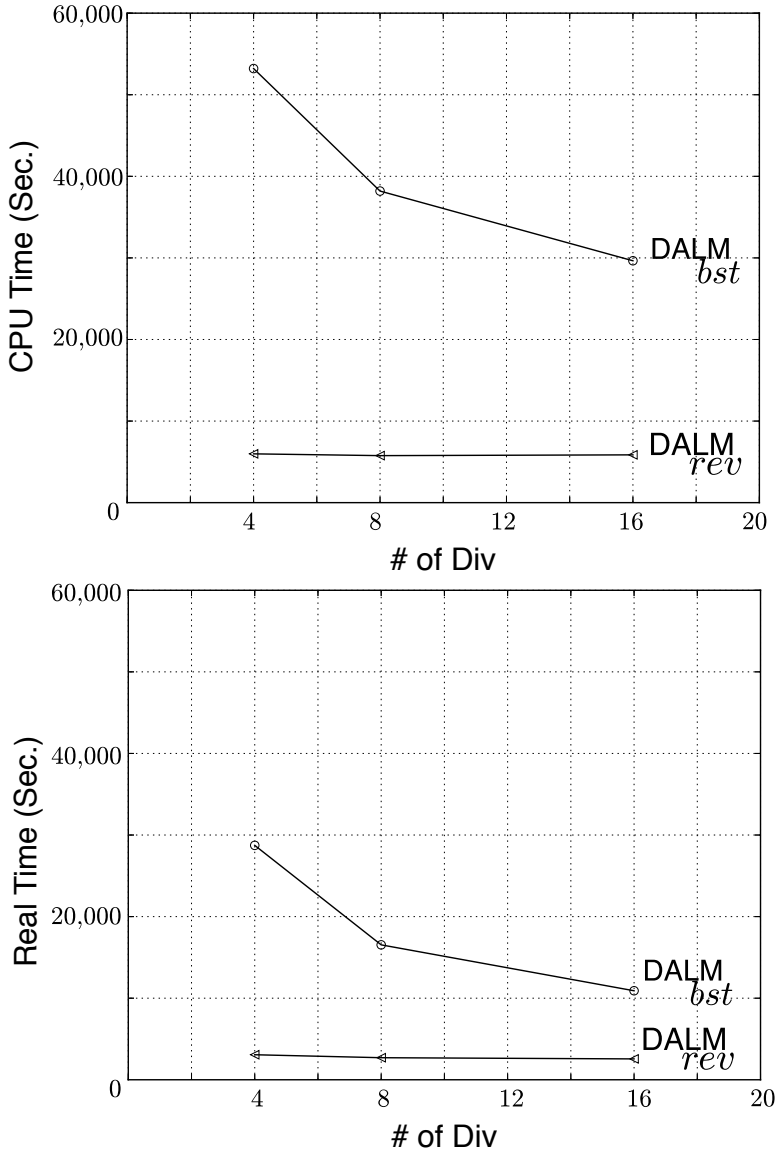

図5.3　分割数による手法ごとの構築時間 (Tiny モデル)。上部に CPU 時間、下部に実時間を示す。

表5.4　分割数による手法ごとの構築時間と充填率 (Tiny モデル)

手法	分割数	CPU 時間 (Sec.)	実時間 (Sec.)	充填率
DALM_{bst}	4	53,209	28,733	89.6%
	8	38,193	16,538	86.1%
	16	29,650	10,911	81.7%
DALM_{rev}	4	5,982	3,068	98.8%
	8	5,764	2,704	97.7%
	16	5,858	2,558	94.6%

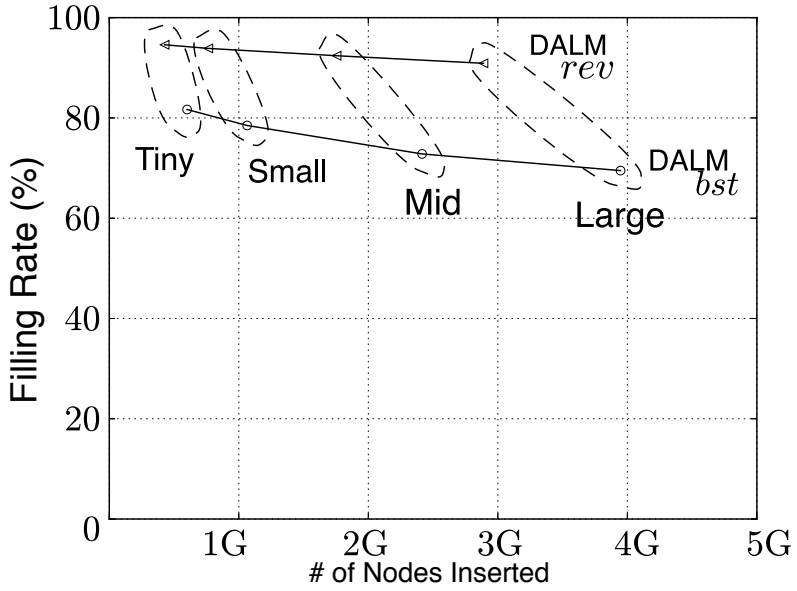

図5.4 挿入ノード数と充填率の関係 (分割数 = 16)

がわかる。DALM$_{bst}$ については分割数が構築時間に影響しているが、DALM$_{rev}$ については顕著な違いは見られなかった。DALM$_{bst}$ については、構築を高速化させたいときは分割数を増加させれば良い。DALM$_{rev}$ の構築に要する CPU 時間が分割数でほとんど変わらなかったのは、分割によるオーバーヘッドの増加と、計算量の現象が釣り合ったとかんがえられる。DALM$_{rev}$ の構築に要する実時間もほとんど分割に影響されないのは、分割された複数の Trie のうち最も遅かった Trie の構築時間が全体の構築時間を決めるためである。Trie の分割が DALM$_{rev}$ で無意味ではないことを示すため、分割数 1 (分割しない) で学習したところ、CPU 時間で 5,734 秒、実時間で 5,916 秒だった。このことから、分割による高速化は確かに機能していることがわかる。

図 5.4と表 5.5に挿入ノード数と充填率の関係を示す。Trie の分割数は 16 である。各モデル共に 16 分割された Trie から 16 個のダブルアレイができるが、ここでは使われたスロット数の合計を、16 個の配列サイズの合計で割った値を示す。この結果から、DALM は挿入ノード数が増加するとダブルアレイの充填率が下がっている事がわかる。下がり方の勢いは DALM$_{rev}$ の方が緩やかである。

全体を通して、構築時間と充填率の関係において DALM$_{rev}$ が DALM$_{bst}$ より良い性能を示した。しかし、先行研究の KenLM はより高速に構築できた。この点においては今後の研究が必要である。

最後に、提案手法のチューニング法が、モデルの与える影響を調べる。実験には、Tiny モデルを用いた。各種法を適用した場合としなかった場合について、ダブルアレイの総配列サイズと充填率を表 5.6に示す。DALM$_{bst}$ については Leaf Compression は適用できないため、単語 ID チューニング法のみとの比較となる。DALM$_{bst}$ は、初期状態で充填率が 74.1% だが、単語 ID をチューニングすることで、充填率が 96.5% まで改善した。DALM$_{rev}$ は、初期状態で充填率 99.8% と極めて高い数値を示しており、チューニングによって充填率を上昇させる余地がない。Leaf Compression は充填率を下げる傾向に

表5.5 挿入ノード数と充填率の関係 (分割数 = 16)

手法	挿入ノード数	充填率
DALM$_{rev}$	427,179,224	94.6%
	764,235,623	93.9%
	1,755,021,878	92.4%
	2,889,010,118	90.9%
DALM$_{bst}$	600,866,395	81.7%
	1,064,367,826	78.5%
	2,415,144,143	72.8%
	3,947,382,526	69.5%

表5.6 チューニング法の効果

手法	チューニング法	分割数	総配列サイズ	充填率
DALM$_{bst}$	なし	1	810,942,983	74.1%
	単語 ID	1	622,839,292	96.5%
DALM$_{rev}$	なし	1	584,146,253	99.8%
	単語 ID	1	584,145,600	99.8%
	Leaf Compression	1	428,433,284	99.7%
	両方	1	428,433,367	99.7%
DALM$_{rev}$	なし	16	608,607,407	96.8%
	単語 ID	16	603,121,638	96.6%
	Leaf Compression	16	492,410,836	86.8%
	両方	16	451,619,232	94.6%

あるが、ノード数が減っているため、ダブルアレイの配列数は大幅に小さくなっている。DALM$_{rev}$ については初期状態の充填率がやや低い状態にどうなるのかを確認するため、16 分割の場合も確認した。分割によって、Leaf Compression した場合の充填率が 86.8% まで下落したが、そこに単語 ID チューニングを加える事で充填率が 94.6% まで戻すことができた。これらのことから、単語 ID チューニングは、充填率が低い時に効果を発揮することがわかる。特に Leaf Compression は充填率を下げる傾向にあるため、単語 ID チューニングは効果的に作用することがわかった。

5.4 パープレキシティ計算時間

パープレキシティ計算における提案手法の性能を測定した。純粋なクエリスピードを測定するため、単語の文字列表現から ID への変換は処理時間に含まれない。実行時間から起動時間を除くため、単にモデルやデータをロードして終了する時間を測定し、対応する実験の実行時間から減算した。実験に使った KenLM のバージョンは Moses decoder (Koehn et al. 2007) release 3.0 に付属している KenLM と合わせた。

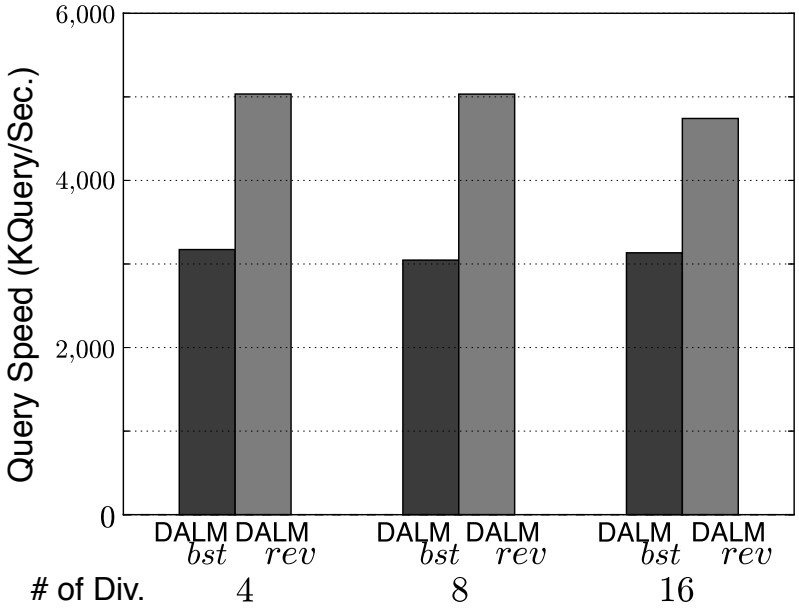

図5.5 分割数によるクエリ速度の変化 (Tiny モデル)

表5.7 分割数によるクエリ速度の変化 (Tiny モデル)

分割数	DALM$_{bst}$ (Queries/Sec.)	DALM$_{rev}$ (Queries/Sec.)
4	3,172,430	5,033,892
8	3,046,476	5,032,750
16	3,133,837	4,741,321

はじめに、分割数がクエリ速度に与える影響を測定した。 図 5.5と表 5.7に、分割数ごとのクエリ速度を示す。分割数の増加に伴い穏やかに速度が低下していることがわかる。これは、分割数の増加が充填率を下げていることから、モデルのキャッシュヒット率が下がっているためである。

次に、先行研究との速度を比較した。図 5.6と表 5.8に先行研究と DALM のメモリ使用量、クエリ速度の比較結果を示す。比較には Tiny モデルと Large モデルを用い、分割数は 16 で実験を行った。

KenLM については、モデルの読み込み方式に `read` を指定した。これは、モデルファイルを一度メモリ中にすべて格納する方式である。KenLM は Linux の mmap を利用したオプションが存在し、これについても実験したが、mmap より read の方が良かったため read の結果のみ示す。

KenLM `trie` と KenLM `probing` の間にはクエリ速度とモデルサイズの間にトレードオフの関係がある。DALM$_{bst}$ のモデルサイズは DALM$_{rev}$ より大きく、KenLM `probing` ($p = 1.5$) とほぼ同じサイズである。DALM$_{bst}$ より DALM$_{rev}$ の方が小さくなったのは、最上位オーダーについてノード数の差があることが考えられる。これは、DALM$_{rev}$ の方が DALM$_{bst}$ よりも最上位オーダーの格納効率が良いためである。すなわち、DALM$_{rev}$ については Leaf compression の効果によりサイズが小さくなっている。

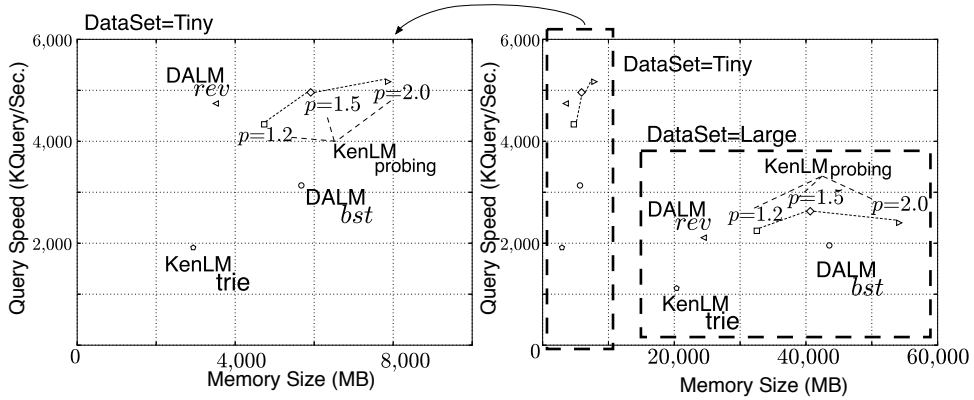

図5.6 言語モデル実装間におけるメモリ使用量、クエリ速度の比較

表5.8 言語モデル実装間におけるメモリ使用量、クエリ速度の比較

	Tiny モデル		Large モデル	
種類	メモリ使用量 (MB)	クエリ速度 (Queries/Sec.)	メモリ使用量 (MB)	クエリ速度 (Queries/Sec.)
DALM_{bst}	5,675	3,133,837	43,552	1,957,098
DALM_{rev}	3,508	4,741,321	24,467	2,108,727
KenLM `probing` ($p = 1.2$)	4,736	4,332,714	32,534	2,243,827
KenLM `probing` ($p = 1.5$)	5,912	4,959,217	40,649	2,627,461
KenLM `probing` ($p = 2.0$)	7,872	5,168,825	54,176	2,400,579
KenLM `trie`	2,938	1,914,340	20,351	1,117,138

クエリ速度については、DALM_{rev} と KenLM `probing` がほぼ同程度、または DALM_{rev} の方が僅かに遅い程度の性能を示した。DALM_{rev} のモデルサイズは、KenLM `trie` とほぼ同等か DALM_{rev} の方が僅かに大きい。DALM_{bst} は KenLM との比較において及ばなかった。

最後に、各実装におけるデータ量に対するスケーラビリティを調べた。この実験は、モデルのメモリ使用量が増加するに連れてどれだけクエリ速度が落ちるのかを見るのが目的である。 図 5.7と表 5.9にメモリ使用量とクエリ速度が、モデルサイズの増加に伴いどう変化したかを示す。いずれの言語モデル実装も、モデルサイズが増加するに伴って実行速度は低下する。DALM の速度低下の様子は KenLM とほぼ同じであることがわかる。DALM_{rev} の速度は KenLM `probing` と KenLM `trie` の中間であるが、KenLM `probing` で Mid モデルを格納するのに使ったのとほぼ同じ領域で Large モデルを格納することができる。

全体を通して、DALM_{rev} は DALM_{bst} より高速に動作した。これは、DALM_{bst} は DALM_{rev} と比べて遷移回数が多いことが原因である。すなわち、DALM_{rev} は Leaf Compression によりノード数が削減されており、それゆえ遷移回数が減少している。加えて、DALM_{rev} は遷移回数が減少しているため、キャッシュヒット率も改善している。さ

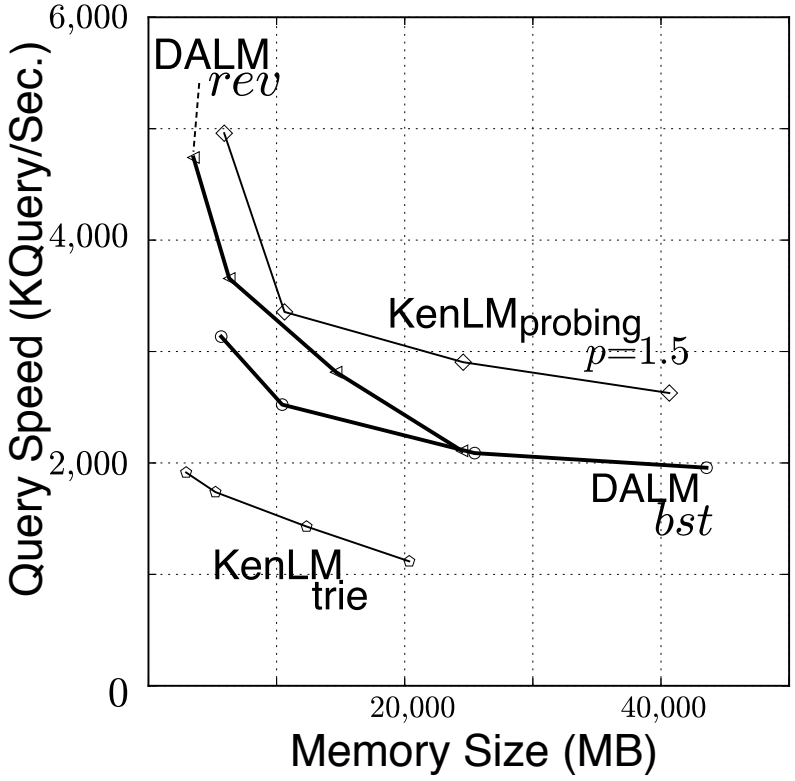

図5.7 モデルサイズによるクエリ速度の違い

表5.9 モデルサイズによるクエリ速度の違い (KenLM `probing` については $p = 1.5$)

	DALM$_{bst}$		DALM$_{rev}$	
モデル	メモリ使用量 (MB)	クエリ速度 (Queries/Sec.)	メモリ使用量 (MB)	クエリ速度 (Queries/Sec.)
Tiny	5,675	3,133,837	3,508	4,741,321
Small	10,437	2,523,510	6,303	3,655,612
Mid	25,458	2,088,421	14,644	2,814,811
Large	43,552	1,957,098	24,467	2,108,727
	KenLM `probing`		KenLM `trie`	
モデル	メモリ使用量 (MB)	クエリ速度 (Queries/Sec.)	メモリ使用量 (MB)	クエリ速度 (Queries/Sec.)
Tiny	5,912	4,959,217	2,938	1,914,340
Small	10,619	3,355,314	5,228	1,738,361
Mid	24,563	2,904,238	12,336	1,429,321
Large	40,649	2,627,461	20,351	1,117,138

らに、DALM$_{bst}$ では言語モデルの state をうまくハンドリングできない問題がある。すなわち、確率値とバックオフウェイトが異なる場所に格納されているため、これらの値を別途集める必要があるため、処理速度が低下している。

この実験で、提案手法の DALM は KenLM とほぼ同じスケーラビリティをもち、メモリ使用量とクエリ速度のバランスは DALM$_{rev}$ が最も良いことがわかった。

5.5 翻訳時間

提案手法の統計的機械翻訳システムに与えるインパクトを調べるため、実験を行った。実験に使ったデコーダは Moses decoder (Koehn et al. 2007) release 3.0 である。このバージョンの Moses decoder は DALM$_{bst}$ を言語モデルの一つとして使うことができるが、本実験ではいくつかの不具合を修正した上で実験した。実験結果の性能は、単語の文字列表現から ID へ変換する時間を一部だけ含んでいる。Moses decoder は、翻訳モデルの単語 ID と言語モデルの単語 ID をそれぞれ別に持つが、起動時にその対応表を 1 次元配列で作成する。すなわち、翻訳時には翻訳モデルの単語 ID から言語モデルの単語 ID を対応表から引く。ただし、この処理時間はごく僅かである。また、統計的機械翻訳システムは起動時に各種モデルをロードする。この処理は時間がかかるが、今回の実験ではこの時間を除外した。除外にあたっては、単にモデルをロードして終了する時間を測定し、その時間を減算した。この実験で用いた DALM の Trie 分割数は 16 である。

まず、言語モデル間でのメモリ使用量と翻訳速度について性能比較を行った。図 5.8と表 5.10にメモリ使用量と翻訳速度の比較結果を示す。メモリ使用量と翻訳時間の観点から、DALM$_{rev}$ は KenLM `probing` とほぼ同じ速度でより省メモリだった。KenLM `trie` との比較では、DALM$_{rev}$ の方がわずかにメモリ使用量が大きいが、より高速である。これらの結果から DALM はダブルアレイの高速・コンパクトという性能をよく引き出せていることがわかる。

次に、翻訳速度と翻訳の質のトレードオフについて調べた。もし、言語モデルの高速化により翻訳速度が改善するなら、デコーダの探索範囲をその速くなった分だけ広げることで、よりよい翻訳結果が得られることを期待できる。Moses decoder には、効率的に仮説空間を探索するために cube pruning (Chiang 2007) という手法が実装されているが、探索の度合いを pop limit というパラメータで調整できる。言い換えるならば、pop limit は翻訳速度と翻訳性能のトレードオフを調整するパラメータである。DALM$_{bst}$ は KenLM `probing` ($p = 1.5$) とほぼ同じ速度となった。

表 5.11に pop limit を変化させた時の翻訳時間の変化とそれぞれ対応するメモリ使用量を示す。なお、翻訳結果はほぼ同一である。この実験では、pop limit として 2, 20, 200, 400, 600 を用いた。実験結果から、KenLM `trie` と DALM の差は pop limit が大きくなるにつれて大きくなった。また、各 pop limit について、DALM$_{rev}$ は KenLM `probing` と同一またはより速い結果となった。DALM$_{bst}$ は KenLM `probing` とほぼ同一の速度である。

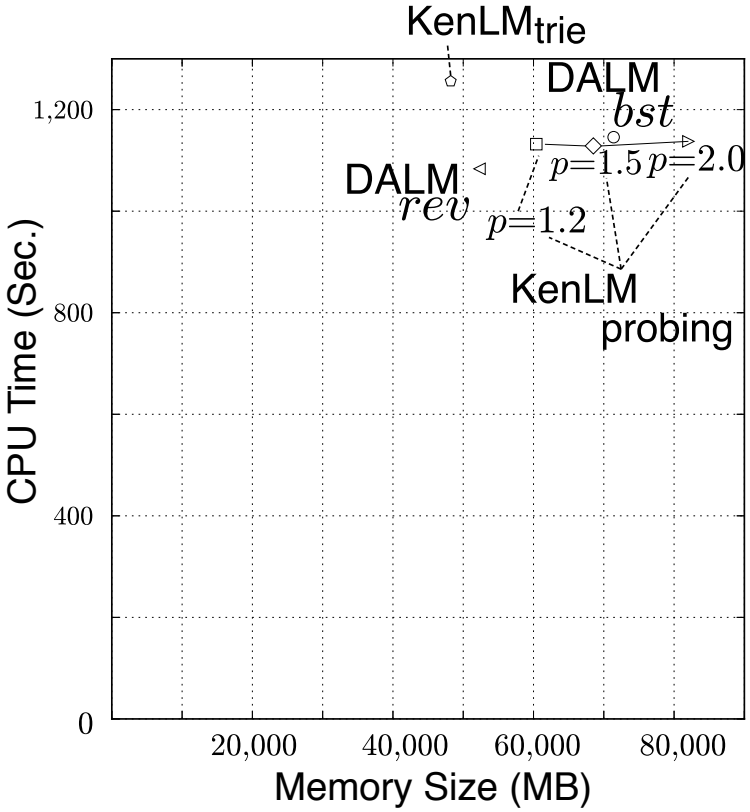

図5.8 メモリ使用量と CPU 時間の関係 (Dataset = Large, pop limit = 200)

表5.10 メモリ使用量と CPU 時間の関係 (Dataset = Large, pop limit = 200)

種類	メモリ使用量 (MB)	CPU 時間 (Sec.)
DALM_{bst}	71,400	1,146
DALM_{rev}	52,315	1,083
KenLM `probing` ($p = 1.2$)	60,396	1,132
KenLM `probing` ($p = 1.5$)	68,512	1,129
KenLM `probing` ($p = 2.0$)	82,038	1,138
KenLM `trie`	48,214	1,256

表5.11 Pop limit と翻訳時間 (CPU 時間) の関係 (Dataset = Large, KenLM `probing` においては $p = 1.5$)

PopLimit	手法	CPU 時間 (Sec.)	
メモリ使用量 (MB)			
2	DALM$_{bst}$	266	71,144
	DALM$_{rev}$	148	52,059
	KenLM `probing`	229	68,256
	KenLM `trie`	189	47,958
20	DALM$_{bst}$	288	71,144
	DALM$_{rev}$	234	52,059
	KenLM `probing`	300	68,256
	KenLM `trie`	293	47,958
200	DALM$_{bst}$	1,146	71,400
	DALM$_{rev}$	1,083	52,315
	KenLM `probing`	1,129	68,512
	KenLM `trie`	1,256	48,214
400	DALM$_{bst}$	2,224	71,784
	DALM$_{rev}$	2,088	52,699
	KenLM `probing`	2,148	68,896
	KenLM `trie`	2,409	48,598
600	DALM$_{bst}$	3,210	72,040
	DALM$_{rev}$	3,145	52,955
	KenLM `probing`	3,179	69,152
	KenLM `trie`	3,597	48,854

第 6 章

おわりに

本研究では、高速かつコンパクトな言語モデルのデータ構造を提案した。言語モデルのパラメータを検索するためのデータ構造として、本研究ではダブルアレイを用いた。ダブルアレイは高速かつコンパクトであることで知られている Trie の実装方法である。

ダブルアレイを言語モデルの実装に使う際は、ダブルアレイ上の無駄な領域を削減するためダブルアレイの隙間にモデルパラメータを埋め込んだ。これによりパラメータ格納用の別領域をメモリに確保することなく、モデルのエントリとパラメータの両方をダブルアレイ内に保持することができるようになった。

本研究では、言語モデルの実装で従来から利用されていた 2 種類の Trie をダブルアレイに適応する方法を提案した。一つは Backward Suffix Tree、もう一つは Reverse Trie である。

Backward Suffix Trie は、単発クエリにおいてその真価を発揮する手法であり、提案手法でもそれに則る形でパラメータを配置した。ただし、統計的機械翻訳においては言語モデル State と相性が悪く、クエリ速度が低下した。

Reverse Trie は、文の先頭から後方に向かって順にクエリするときに効率的なデータ構造である。この方法は統計的機械翻訳において言語モデルの State 表現と相性がよく、提案手法でも高速な翻訳を実現した。

本研究では、ダブルアレイを言語モデルの実装でより効率的に利用するため、ダブルアレイの構造のチューニングを行った。一つは、単語 ID のチューニング、もう一つは葉ノードの圧縮である。単語 ID をチューニングすることで、Trie の性質を改善させ、よりコンパクトな言語モデルを実現した。また、葉ノードを圧縮することで、ダブルアレイ内のノード数を減少させ、更にコンパクトな言語モデルとなった。

実験においては先行研究の KenLM と比較した。KenLM には KenLM `probing` と KenLM `trie` の 2 通りの手法が実装されている。KenLM `probing` は高速だがメモリ使用量が大きい、KenLM `trie` は低速だがメモリ使用量が小さいというトレードオフの関係にある。

パープレキシティ測定実験の結果、Backward Suffix Tree に基づく提案手法は KenLM `probing` とほぼ同等のメモリ使用量となったが速度で及ばなかった。Reverse Trie に基づく提案手法は、KenLM `probing` とほぼ同じクエリ速度で KenLM `trie` より僅かに大きいメモリ使用量を実現した。パープレキシティ測定実験の結果、Reverse Trie に基づく提案手法は、KenLM のもつトレードオフの関係を改善できた。

さらに、現実のユースケースにおいての性能改善を調べるため、翻訳実験も行った。実

験の結果、Backward Suffix Tree に基づく提案手法は、速度で KenLM `probing` に迫ったもののメモリ使用量の大きさにより KenLM に及ばなかった。DALM$_{rev}$ は KenLM `trie` より僅かに大きなメモリ使用量で、KenLM `probing` と同程度の翻訳速度を達成し現実のユースケースでも KenLM のトレードオフを改善していることを確認した。

最後に今後の課題を述べる。1 つ目は構築速度の改善である。ダブルアレイの構築速度は低速であることが知られているが、本研究での構築実験においてもそれが課題となった。ダブルアレイ言語モデルの構築時間は KenLM に比べて遅く、今後の研究が必要である。実用上、一度システムを構築してしまえば頻繁にモデルを差し替えることは少ないと考えるが、企業・組織内での検証用途では検証サイクルにかかる時間に影響を与えるため、改善が必要である。2 つ目は、ダブルアレイに格納可能なノード数の上限を解消することである。現在、ダブルアレイの配列には 32bit の変数を使っており、これが原因となって 1 つのダブルアレイに $2^{31} - 1$ 以上のノードを格納することができない。3 つ目は、量子化の問題である。量子化は言語モデルのパラメータを離散化することで性能を出来るだけ損なわずに、言語モデルをよりコンパクトにする手法である。本研究ではダブルアレイに元々存在した隙間にパラメータを格納したため、モデルパラメータを離散化してもモデルサイズは小さくならない。これらの点については今後のさらなる研究が望まれる。

付録 A SRILM の推定した言語モデルの取り扱い

SRILM の推定した言語モデルの問題点

SRILM (Stolcke 2002) は、言語モデルの推定・クエリができるツールキットである。本研究では KenLM を先行研究としたが、KenLM 以前には統計的機械翻訳でもよく使われていた。SRILM のモデル推定機能は、KenLM のモデル推定機能より細かい設定が可能であるため、今日でも利用者が存在する。

一般的に、カットオフを利用するときは高次のエントリに含まれる単語列の部分単語列が低次のエントリ群に含まれるように設定する。すなわち、任意の $m(1 < m \leq n)$ について、$m$gram に含まれる単語列 $\boldsymbol{v}_1^m$ の部分単語列 $\boldsymbol{v}_1^{m-1}$ と $\boldsymbol{v}_2^m$ が $m-1$gram のエントリの中に含まれるようにする。この条件をみたすようにカットオフするには、$m-1$gram のカットオフ値より $m$gram のカットオフ値が大きくなるまたは同じになるように設定する。

SRILM のモデル推定機能によって Interpolated Modified Kneser-Ney Discount を用いてモデル推定すると、出力された ARPA ファイルはカットオフの一貫性を壊すことがある。Interpolated Modified Kneaser-Ney Discount は言語モデルのモデル推定手法の一つである。これは、SRILM がカットオフ処理をする際に、出現回数の代わりに Interpolated Modified Kneser-Ney Discount の処理中で使う別の値を使ってカットオフするためである[*1]。そのため、カットオフ値を単調増加になるよう設定してもオーダー間の整合性が崩れることがある。

SRILM の推定したモデルから Reverse Trie を作る方法

SRILM の出力した ARPA ファイルから Reverse Trie を構築すると、一部の必要なパラメータが ARPA ファイルに記載されていないことがある。すなわち、ある $m$gram $\boldsymbol{v}_1^m$ が ARPA ファイルに記載されている場合でも、$\boldsymbol{v}_2^m$ が記載されていないことが起こりうる。

$\boldsymbol{v}_1^m$ は、ルートノードから $v_m \to v_{m-1} \to \cdots \to v_2 \to v_1$ と辿った位置に格納するが、そのためには $\boldsymbol{v}_2^m$ のノードが Trie の中に必要となる。$\boldsymbol{v}_2^m$ を Trie に挿入するには、確率値とバックオフウェイトを用意する必要がある。

確率値 $P\left(v_m \mid \boldsymbol{v}_2^{m-1}\right)$ は、バックオフ計算をした結果の値を格納する。すなわち、Trie

*1. 詳細は`http://www.speech.sri.com/projects/srilm/manpages/ngram-discount.7.html` を参照

の構築時に以下の式を計算し挿入する。

$$P\left(v_m \mid \boldsymbol{v}_2^{m-1}\right) = P\left(v_m \mid \boldsymbol{v}_3^{m-1}\right) \beta\left(\boldsymbol{v}_2^{m-1}\right).$$

また、バックオフウェイトは 1 (対数バックオフウェイトにおいては 0) でよい。この手法は KenLM (Heafield 2011) で実装されている。

謝辞

筑波大学大学院システム情報工学研究科の山本幹雄教授には在学中、指導教員として多くのアドバイスを頂いたのみならず、公私に渡り多くのチャレンジを後押ししていただきました。心から深く感謝申し上げます。

副査を快くお引き受け頂いた、筑波大学大学院システム情報工学研究科の久野誉人教授、宇津呂武仁教授、山田武志准教授、乾孝司准教授に感謝します。論文審査において、大変有益かつ親身なご助言をいただきました。

筑波大学大学院システム情報工学研究科の滝沢穂高准教授、津川翔助教を始め、知能情報研究室の皆様には日頃から大変お世話になりました。感謝申し上げます。同研究室卒業生の安原誠君と田中透君に感謝します。このお二人とのディスカッションは多くの示唆を与えてくれました。同研究室卒業生の豊橋技術科学大学吉田光男助教に感謝します。吉田光男助教には在学中、論文のアドバイスを何度もいただきました。

TALLIP の匿名査読者の皆様に感謝します。査読におきましては、論文のテキストだけでなく手法に至るまで多くのご指摘・ご助言を頂き、研究の質が飛躍的に向上しました。

実験には、NTCIR10 特許翻訳タスクのデータを利用させていただきました。利用許諾につきまして、心より感謝申し上げます。

ここまで育て、支えていただきました両親に感謝します。いつも祈ってくれてありがとうございます。父には生前、大変心配をかけました。母には急に博士課程に進学すると伝え、不安にさせてしまったと思います。それでも応援してくれてありがとうございます。

博士課程在学期間を支えてくれた最愛の妻、乗松 (赤星) 友香さんに感謝します。あなたの祈り支え、そして研究への理解がなければ、博士論文を完成させることはできませんでした。いつも一緒にいてくれてありがとうございます。これからもよろしくお願いします。

最後に、私の主に感謝します。私が博士後期課程に進学するときの心境は、かつてのエレミヤの叫び声に似ていたように記憶しています。“主よ、あなたがわたしを欺かれたので、わたしはその欺きに従いました。あなたはわたしよりも強いので、わたしを説き伏せられたのです。” (口語訳聖書エレミヤ書 20:7) 在学中、私はこの叫びに支えられてきたと言っても過言ではありません。最近、聖書から新たな言葉を頂きました。感謝しつつ受け止めたいと思います。

“主はあなたをエジプトの地、奴隷の家から導き出し、あなたを導いて、あの大きな恐ろしい荒野、すなわち火のへびや、さそりがいて、水のない、かわいた地を通り、あなたのために堅い岩から水を出し、先祖たちも知らなかったマナを荒野であなたに食べさせられた。それはあなたを苦しめ、あなたを試みて、ついにはあなたをさいわいにするためであった。”(口語訳聖書申命記 8:14-16)

2016 年 3 月乗松潤矢

参考文献

Aoe, Jun-ichi. 1989. “An Efficient Digital Search Algorithm by Using a Double-Array Structure.” *IEEE Transactions on Software Engineering* 15 (9): 1066–1077.

Belazzougui, Djamal, Fabiano C Botelho, and Martin Dietzfelbinger. 2009. “Hash, Displace, and Compress.” In *Proceedings of the 17th European Symposium on Algorithms,* 682–693.

Brants, Thorsten, Ashok C Popat, Peng Xu, Franz J Och, and Jeffrey Dean. 2007. “Large Language Models in Machine Translation.” In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning,* 858–867. Association for Computational Linguistics.

Brown, Peter F., Vincent J. Della Pietra, Robert L. Mercer, Stephen A. Della Pietra, and Jennifer C. Lai. 1992. “An Estimate of an Upper Bound for the Entropy of English.” *Computational Linguistics* 18 (1): 31–40.

Brown, Peter F., Vincent J. Della Pietra, Stephen A. Della Pietra, and Robert L. Mercer. 1993. “The Mathematics of Statistical Machine Translation: Parameter Estimation.” *Computational Linguistics* 19 (2): 263–311. ISSN: 0891-2017.

Chiang, David. 2007. “Hierarchical Phrase-Based Translation.” *Computational Linguistics* 33 (2): 201–228.

Clarkson, Philip, and Donald Rosenfeld. 1997. “Statistical language modeling using the CMU-Cambridge toolkit.” In *Eurospeech,* 2707–2710.

Fredkin, Edward. 1960. “Trie memory.” *Communications of the ACM* 3 (9): 490–499.

Fredriksson, Kimmo, and Fedor Nikitin. 2007. “Simple Compression Code Supporting Random Access and Fast String Matching.” In *Proceedings of the 6th International Conference on Experimental Algorithms,* 203–216. Workshop on Experimental Algorithms.

Goto, Isao, Ka Pa Chow, Bin Lu, Eiichiro Sumita, and Benjamin K. Tsou. 2013. “Overview of the Patent Machine Translation Task at the NTCIR-10 Workshop.” In *10th NTCIR Conference,* 260–286.

Guthrie, David, and Mark Hepple. 2010. “Storing the Web in Memory: Space Efficient Language Models with Constant Time Retrieval.” In *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing,* 262–272. Association for Computational Linguistics.

Heafield, Kenneth. 2011. “KenLM : Faster and Smaller Language Model Queries.” In *Proceedings of the Sixth Workshop on Statistical Machine Translation,* 187–197. 2009. Association for Computational Linguistics.

Heafield, Kenneth, Hieu Hoang, Philipp Koehn, Tetsuo Kiso, and Marcello Federico. 2011. “Left Language Model State for Syntactic Machine Translation.” In *Proceedings of the International Workshop on Spoken Language Translation,* 183–190.

Heafield, Kenneth, Ivan Pouzyrevsky, Jonathan H Clark, and Philipp Koehn. 2013. “Scalable Modified Kneser-Ney Language Model Estimation.” In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics,* 690–696.

Jacobson, Guy. 1989. “Space-efficient static trees and graphs.” In *30th Annual Symposium on Foundations of Computer Science,* 549–554. IEEE.

Jelinek, Frederick. 1990. “Readings in Speech Recognition.” Chap. Self-Organized Language Modeling for Speech Recognition, 450–506. Morgan Kaufmann Publishers Inc.

Jelinek, Frederick, and Robert L. Mercer. 1980. “Interpolated Estimation of Markov Source Parameters from Sparse Data.” In *In Proceedings of the Workshop on Pattern Recognition in Practice,* 381–397. Amsterdam, The Netherlands: North-Holland, May.

Katz, Slava M. 1987. “Estimation of probabilities from sparse data for the language model component of a speech recognizer.” *IEEE Transactions on Acoustics, Speech, and Signal Processing* 35 (3): 400–401.

Koehn, Philipp, Hieu Hoang, Alexandra Birch, Chris Callison-Burch, Marcello Federico, Nicola Bertoldi, Brooke Cowan, et al. 2007. “Moses: Open Source Toolkit for Statistical Machine Translation.” In *Proceedings of the 45th Annual Meeting of the Association for Computational Linguistics Companion Volume Proceedings of the Demo and Poster Sessions,* 177–180. Association for Computational Linguistics.

Koehn, Philipp, FJ Och, and Daniel Marcu. 2003. “Statistical phrase-based translation.” In *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies,* 48–54.

Li, Zhifei, and Sanjeev Khudanpur. 2008. “A Scalable Decoder for Parsing-Based Machine Translation with Equivalent Language Model State Maintenance.” In *Proceedings of the ACL-08:˜HLT Second Workshop on Syntax and Structure in Statistical Translation (SSST-2),* 10–18. Association for Computational Linguistics.

Liu, Huidan, Minghua Nuo, Longlong Ma, Jian Wu, and Yeping He. 2011. “Compression Methods by Code Mapping and Code Dividing for Chinese Dictionary Stored in a Double-Array Trie.” In *Proceedings of 5th International Joint Conference on Natural Language Processing,* 1189–1197. Asian Federation of Natural Language Processing.

Pauls, Adam, and Dan Klein. 2011. “Faster and Smaller N-Gram Language Models.” In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies,* 258–267. Association for Computational Linguistics.

Sorensen, Jeffrey, and Cyril Allauzen. 2011. “Unary Data Structures for Language Models.” *INTERSPEECH:* 2–5.

Stolcke, A. 2002. “SRILM-an Extensible Language Modeling Toolkit.” *Seventh International Conference on Spoken Language Processing.*

Talbot, David, and Miles Osborne. 2007. “Smoothed Bloom Filter Language Models: Tera-Scale LMs on the Cheap.” In *In Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning,* 468–476.

Watanabe, Taro, Hajime Tsukada, and Hideki Isozaki. 2009. “A Succinct N-gram Language Model.” In *Proceedings of the ACL-IJCNLP 2009 Conference Short Papers,* 341–344. Association for Computational Linguistics.

Yoshinaga, Naoki, and Masaru Kitsuregawa. 2014. “A Self-adaptive Classifier for Efficient Text-stream Processing.” In *Proceedings of COLING 2014, the 25th International Conference on Computational Linguistics: Technical Papers,* 1091–1102. Dublin City University / Association for Computational Linguistics.

中村康正 and 望月久稔. 2006. “圧縮ディジタル探索木における辞書情報更新の高速化手法.” 情報処理学会論文誌データベース（*TOD*) 47, no. 13 (September): 16–27.

大野将樹, 森田和宏, 泓田正雄, and 青江順一. 2003. “ダブル配列におけるキー削除の効率化手法,” 2003:1–6. 23. 一般社団法人情報処理学会, March.

矢田晋, 田村雅浩, 森田和宏, 泓田正雄, and 青江順一. 2009. “ダブル配列による動的辞書の構成と評価,” 71:263–264. March.

重越秀美, 蔵満琢麻, and 望月久稔. 2009. “ダブル配列の遷移集合管理による追加・削除処理の高速化 (データベース, 査読付き論文),” 8:1–6. 2. FIT(電子情報通信学会・情報処理学会) 運営委員会, August.

本論文に関連する論文の一覧

本論文の主要部分は以下の論文で公表済み・公表予定である。

公表済み論文

1. Makoto Yasuhara, Toru Tanaka, Jun-ya Norimatsu, and Mikio Yamamoto. 2013. “An Efficient Language Model Using Double-Array Structures.” In Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing, 222–232. Association for Computational Linguistics.

採録決定論文

1. Jun-ya Norimatsu, Makoto Yasuhara, Toru Tanaka, and Mikio Yamamoto. 2016. “A Fast and Compact Language Model Implementation Using Double-array Structures.” To appear in the Transactions on Asian and Low-Resource Language Information Processing (TALLIP).