NAIST-IS-DT9761025

博士論文

大規模言語コーパスの解析と検索に関する研究

山下 達雄

2002年 2月 5日

奈良先端科学技術大学院大学
情報科学研究科 情報処理学専攻

本論文は奈良先端科学技術大学院大学情報科学研究科に
博士 (工学) 授与の要件として提出した博士論文である.

論文番号： NAIST-IS-DT9761025

提出者： 山下 達雄

審査委員： 松本 裕治 教授
伊藤 実 教授
関 浩之 教授

提出日： 2002年 2月 5日

大規模言語コーパスの解析と検索に関する研究*

山下 達雄

内容梗概

本論文では、大規模言語コーパスに関して、言語情報、特に形態素情報のアノテート技術とそのアノテートされたコーパスを解析や検索に利用する技術を提案し有効性を示す。

言語情報がアノテートされた大規模なコーパス、いわゆるタグ付きコーパスは、近年の自然言語処理において、不可欠なものとなっている。例えば、大量のタグ付きコーパスから統計的な手法によりパラメタを学習し、それを用いた形態素解析器やパーザが開発されており、それらは意味解析などの深い言語処理の研究に役立っているだけでなく、表層的な言語情報を用いて最大限の効果を得ることを目的に情報検索などで利用されている。タグ付きコーパスを大量に作成していくということは、自然言語処理の基礎データの蓄積を意味し、自然言語処理研究に欠かせない重要な課題である。

本研究では、まず、アノテート対象の言語に着目し、特定の言語や同系統の数言語の解析のみを念頭に置いて開発されている現状の言語技術を打破すべく、言語非依存の形態素解析の枠組を提案し有効性を示す。そして、次にタグ付きコーパスを作成する人間に着目し、タグ付きコーパスを効率的に蓄積する支援環境 (グラフィカル・ユーザ・インターフェース) を提案し有効性を示す。タグ付きコーパス作成作業の効率化への寄与という視点に基づき、これまでのコーパスから統計的な手法で学習されたパラメタを用いる解析手法ではなく、人手作成ルールとの融合や事例ベースによる解析手法といった利用法を提案し有効性を示す。さらに、解析処理以外のでのコーパスの利用法として、人間によるタグ付きコーパス蓄積作業の際の「参照」に着目し、そのための、大規模なタグ付きコーパスからの類似文字列検索を高速、高精度で行なう手法を提案し有効性を示す。

このように、タグ付きコーパスに関して、蓄積・利用に関する技術を一連の流れとしてとらえることは、自然言語処理技術の基盤を整備することであり、その重要性ははかり知れない。

キーワード

自然言語処理, コーパス, アノテート, 形態素解析, 確率モデル, 事例ベース, 類似検索

* 奈良先端科学技術大学院大学 情報科学研究科 情報処理学専攻 博士論文, NAIST-IS-DT9761025, 2002 年 2 月 5 日.

Studies on Analysis and Retrieval of Large Scale Linguistic Corpus *

Tatsuo Yamashita

Abstract

This thesis shows basic technologies for large scale linguistic corpus. The corpus which has linguistic information tags are inevitable resources in natural language processing. This thesis focuses on three topics related to large scale linguistic corpus. (1) This thesis shows a framework of language independent morphological analysis. Various languages exist in the world, and strategies for morphological analysis differ by types of languages. This framework works for various language types. (2) This thesis shows the graphical user interface which supports an annotator to tag the linguistic information to the plain corpus. This system enables annotators to efficiently construct annotated corpus. From a view of annotation support, I propose two approaches for morphological analysis. The first approach integrates probabilities of tag sequences in corpus and hand-crafted rules. The second approach uses annotated corpus directory by using fast string search technique. These two approaches are suitable for bootstrap annotation. (3) This thesis focuses on one of annotator's activities, which is retrieval of annotated corpus. This thesis shows two types of approximate search methods for retrieving corpus.

Keywords:

Natural Language Processing, Corpus, Morphological Analysis, Stochastic Model, Example-based aproach, Approximate Search

* Doctor's Thesis, Department of Information Processing, Graduate School of Information Science, Nara Institute of Science and Technology, NAIST-IS-DT9761025, February 5, 2002.

目 次

図 目 次

第1章

序論

1.1. 研究の背景

品詞などの言語情報がアノテートされたテキスト、いわゆるタグ付きコーパスは、近年の自然言語処理において、不可欠なものとなっている。

大量のタグ付きコーパスから統計的な手法によりパラメタを学習し、それを用いる形態素解析器やパーザが開発されており、意味解析などの深い言語処理の研究に役立っているだけでなく、表層的な言語情報を用いて最大限の効果を得ることを目的に情報検索などで利用されている。タグ付きコーパスを大量に作成していくということは、自然言語処理の基礎データの蓄積を意味し、自然言語処理研究に欠かせない重要な課題である。

このタグ付きコーパスを作成するのは人間である。そのため、タグ付きコーパスを効率的に蓄積するための支援環境が必要となる。蓄積したコーパスから漸進的にパラメタ学習を行ない、タグ付け作業にフィードバックする、といった「賢い」支援が欠かせない。

また、このようなタグ付きコーパス作成作業の効率化への寄与という視点に基づき、これまでのコーパスから統計的な手法で学習されたパラメタを用いる解析手法だけではなく、人手作成ルールとの融合や事例ベースによる解析手法といった利用法も有望である。

解析処理以外のコーパスの利用法として、人間によるタグ付きコーパス作成作業の際の「参照」が挙げられる。そのためには、大規模なタグ付きコーパスから、類似した文等を高速かつ高精度で検索できる必要がある。

このように、タグ付きコーパスに関して、蓄積・利用に関する技術を一連の流れとしてとらえることは、自然言語処理技術の基盤を整備することであり、その重要性ははかり知れない。

1.2. 研究の目的

本論文では、品詞タグ付きコーパスの蓄積から利用までを一連の流れとしてとらえ、蓄積・利用に関する手法を提案し、その有用性を示す。

1.2.1 言語に依存しない形態素解析処理

メジャーな言語のタグ付きコーパスは近年大量に蓄積されつつあるが、新たな言語を対象にタグ付きコーパスの蓄積を行なうことを考えると、特定の言語や同系統の数言語の解析のみを念頭に置いて開発されている現状の言語技術では不十分である。

構文解析や意味解析などの高度な言語処理では、対象言語の違いはほとんど意識しないくても良いため、既存の枠組で対応可能である。しかし、入力文を意味のある塊に分

割するトークナイズや形態素解析といった低レベルの言語処理では、対象言語の記述の多様性が共通の枠組の構築を妨げていた。

そこで、本論文では、特定の言語に依存した部分を見つけ出し、そこをできるかぎり小さくし、他の部分を共通化した言語処理の枠組を提案する。

1.2.2 品詞タグ付きコーパスの蓄積とそれを利用した解析

タグ付きコーパスを作成するのは人間である。これまでのタグ付きコーパス蓄積プロジェクトではエディタ等の単純なインターフェースでの作業が主であった。そこで、作業の効率化、作成されたコーパスの質の一貫性などを考慮した作業環境が必要になる。本論文では、直観的に理解しやすいグラフィカル・ユーザ・インターフェースによるタグ付きコーパス作成支援システムを提案する。さらに、作業の効率化、質の一貫性を保つため、それまでの作業で蓄積したコーパスを利用した事例ベース学習機能を実装し、その有用性を確認する。

一般に、蓄積されたタグ付きコーパスは、統計的手法でパラメタ学習され解析処理の質の向上に利用される。しかし、ある程度の精度を出すには大量のタグ付きコーパスが必要となり、それらが整備されていない未開拓な分野には不向きである。一方、ルールベースの解析器は比較的簡単に実装でき、手作業による調整もしやすいが、多くの人的労力が必要となるのが問題である。そこで、タグ付きコーパスと人手で作成されたルールを融合し、蓄積途中の少量のコーパスでも十分に利用できる枠組を提案する。

また、統計的手法でパラメタ学習する以外にも、解析処理の質の向上に寄与する利用法がある。それはタグ付きコーパスをそのまま事例データとして利用する方法である。タグ付きコーパスを直接部分文字列検索し、マッチした部分文字列とそれに付与されたタグを元に形態素解析を行なう手法を提案する。

1.2.3 Suffix Array によるコーパスの類似検索

タグ付きコーパス作成作業の際には、過去のタグ付与基準などを参照するため、これまで蓄積してきたタグ付きコーパスを検索する必要がある。この際には、全く同じ文を検索するだけではなく、部分的に類似しているものも検索したい。

このような類似検索技術によるコーパス検索は、作業者にとってはタグ付与の際の基準を明らかにできるので作業効率の向上につながる。また、このようにして蓄積されたタグ付きコーパスは、同じ文脈の文字列に対しては同じタグが付与されるという、タグ付与の一貫性が保たれ、質の向上にもつながる。

そこで、Suffix Array と呼ばれる文字列検索のためのデータ構造を用いた二種類の類

似検索手法を提案する。一つは動的計画法と組み合わせたもので、検索キーと数文字異なるだけの文字列など、見つけたいものがある程度分かっている場合に向いている。もう一つは、転置インデックスと組み合わせたもので、全体は似ていなくてもどこかの部分が似ているようなものを検索する場合に向いている。

1.3. 論文の構成

図 1.1 に本論文で扱う範囲を示す。

2 章では、言語に依存しない形態素解析処理について説明する。まず、特定の言語や同系統の数言語の解析のみを念頭に置いて開発されている現状の言語技術の問題点を明らかにし、それに基づき特定の言語に依存しない処理の枠組について述べる。図 1.1 の「形態素解析器」「解析用パラメタ」の部分に相当する。

3 章では、品詞タグ付きコーパスの蓄積とそれを利用した解析の技術について説明する。まず、品詞タグ付きコーパス作成支援システム (GUI) と作業効率化のための学習機能について述べる。そして、蓄積されたタグ付きコーパスを形態素解析に利用する二つの方法 (人手作成ルールとの統合、コーパスの直接利用) について述べる。図 1.1 の「類似検索エンジン」以外の全部分に相当する。

4 章では、タグ付きコーパス作成作業の際の参照に有用である類似検索技術について述べる。図 1.1 の「タグ付きコーパス」「Indexer」「検索用データ」「類似検索エンジン」の部分に相当する。

5 章では、結論と今後の課題について述べる。

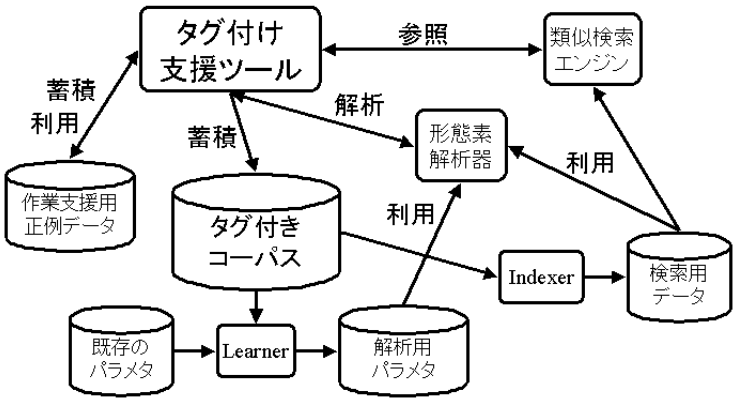


図 1.1 全体像

第2章

言語に依存しない形態素解析処理の枠組み

本章では、これまでの形態素解析処理に対して、特定の言語に依存した部分を見つけ出し、そこをできるかぎり小さくし、他の部分を共通化した言語非依存の枠組を提案する。また、その枠組による形態素解析システムの開発について述べる。

2.1. はじめに

形態素解析処理とは文を形態素という文字列単位に分割し品詞情報を付与する処理である。すでに成熟している技術であるが、解析精度や速度の向上のために様々な手法を試みる余地はあり、そのための技術的な拡張要求もある。他の自然言語処理処理技術と比べ形態素解析技術は実用に近い位置にあり、それゆえ、形態素解析システムに対する現場からの使い勝手の向上のための要求が多い。

その要求の一つに、多言語対応がある。インターネット上で様々な言語のテキストが行き交う現代において、特定の言語に依存しない、多種多様な言語を視野に入れた自然言語処理が必要とされている。しかし、これまでの形態素解析システムは、特定の言語、または、同系統の数言語の解析のみを念頭に置いて開発されている。本章の目的の一つは、特定の言語に依存しない形態素解析の枠組の構築である。本章では、形態素解析処理の言語に依存した部分を考察し、その部分をできるかぎり共通化した枠組を提案する。

形態素解析は自然言語処理における基本的なコンポーネントであるが、ミクロな視点から見れば形態素解析処理自体も複数のコンポーネントからなりたっている。本研究では、完成した単一のシステムとして提供するだけではなく、システムを構成しているコンポーネント単位で利用できるように設計・実装を行った。コンポーネント化により、変更箇所を最小限におさえることができ、機能拡張が容易になる。また、言語非依存化などの調整や個々のコンポーネントの評価が行いやすくなる。

2.2 節では、形態素解析処理の言語に依存した部分をできるかぎり共通化した言語非依存の枠組について解説する。2.3 節では、形態素解析システムの主要な内部処理のコンポーネント化を行い、それを基に形態素解析ツールキットの実装を行った。個別のコンポーネントについての言語非依存性と汎用性を考察し、実装の方針について解説する。

2.2. 言語に依存しないトークン認識と辞書検索

特定の言語に依存しない一般的な形態素解析システムを考えてみる。まず、最初に行う処理は、システムに与えられた解析対象文を**トークン**と呼ばれる文字列単位に分割する処理である。この処理を **トークン認識** (Tokenization) と呼ぶ。トークンとは、英語では単語や数字や記号などに該当するが、厳密な定義は無い。トークンを発展させた概念が本論文で提案する**形態素片** (Morpho-fragment) である。2.2.3 節で導入する。**語** (Lexeme) は一個以上の連続するトークンから構成されるもので用意された**辞書のエントリとして存在するもの**と定義する。語は**形態素**と呼ばれることもあるが、言語学的な形態素の定義とは異なるので本章では別名を与えることにした。次に行う処理は、認識されたトークンの列に対して形態素辞書の検索を行い、語を認識する処理である。この処理を本論

文では単に**辞書検索**と呼ぶ。

トークン認識と辞書検索は解析対象言語のコンピュータ上での表記の特性によって処理方法が異なる。言語によって異なる場合もあるし、同じ言語でも清書法によって異なる場合もある。本論文では「書かれた言語」を対象に形態素解析することを前提としているので、「言語」とは表記、清書法をも含む意味に捉えることにする。本論文では、言語を表記の特性によって以下の二つに分類する。

- わかち書きされる言語
- わかち書きされない言語

わかち書きされる言語の例として英語があげられる。英語では語は**空白文字** (whitespace) や**記号文字** (panctuation mark) によって区切られていると考えられることが多く、トークン認識は単純明解な処理とみなされあまり重要視されなかった。しかし、いくつかの問題がある。これについては 2.2.2 節で説明する。わかち書きされない言語の例として日本語や中国語などがあげられる。単語の境界が視覚的にはっきりしていないという表記上の特性をもっている。ゆえに、トークン認識は重要かつ困難な処理である。

トークンの処理はわかち書きされる言語とわかち書きされない言語ではまったく異なるとみなされてきた。英語などのわかち書きされる言語では明白な境界が単語の両側にあるが、日本語や中国語などわかち書きされない言語では明白な単語境界を表すものが必ずしもあるとは言えない。わかち書きされる言語とされない言語の違いは、言語の特性というよりも表記の特性によるものであり、清書法の方針による分類と言える [1]。

本章ではわかち書きされる言語とされない言語のトークン認識方法の違いに着目し、どちらの表記法にも適応できる統一的な枠組を提案する。この統一的な枠組により、システムの最小限の変更 (もしくは変更不要) とデータの入れ換えだけで様々な言語を同じシステムで解析できる。また、複数の言語が混じった文章を解析することもできる。

2.2.1 わかち書きされる言語とされない言語の処理の違い

わかち書きされない言語では、トークン認識は全ての文字をトークンとして認識すれば良い。理論的には文中の全ての部分トークン列 (この場合、部分文字列) を語の候補として考慮する必要がある。また、語辞書により同じ字面の候補に複数の品詞候補が与えられることがある。そのため、区切りの曖昧性と品詞付与の曖昧性の二種類の曖昧性が生じる。

わかち書きされる言語では、トークンは一意に決定され、かつ、1 トークン = 1 語という単純化を行うことが多く、その場合は品詞付与の曖昧性だけが生じる。しかし、実

文	彼は休暇で帰省している。 (He is home for the holidays.)
文字	| 彼 | は | 休 | 暇 | で | 帰 | 省 | し | て | い | る | 。
語 (Lexemes)	| 彼 | は | 休暇 | で | 帰省 | して | いる | 。
形態素片	| 彼 | は | 休 | 暇 | で | 帰 | 省 | し | て | い | る | 。

図 2.1 日本語文の構成

際はわかち書きされる言語でも、1 トークン ＝ 1 語とみなすことが困難な場合がある。これについては 2.2.2 節で説明する。

このような、わかち書きされる言語とされない言語のトークン認識処理の違いは、辞書検索の方法に影響を与える。

わかち書きされる言語では語の両側に必ず空白などの明白な区切りを持つ (このように空白で区切られている語を Graphic Word と呼ぶ [2])。ゆえに、辞書検索ではシステムはわかち書きされた文字列が語辞書に存在するかを問い合わせるだけでよい。もし存在するなら品詞等の情報を辞書から得る。

一方、わかち書きされない言語では文は明白な語の区切りを持っていないため (図 2.1)、辞書検索ではシステムは全ての部分トークン列 (部分文字列) が語辞書に存在するかそれぞれ問い合わせる必要がある。一般にわかち書きされない言語の辞書検索は、文中のある位置から始まる全ての語を辞書から一括で取り出す**共通接頭辞検索** (common prefix search) と呼ばれる手法が用いられる (4.3.3 節を参照)。一般に common prefix search を効率的に行うためにトライ (TRIE) というデータ構造を用いる。図 2.2 に日本語トライ辞書の一部を示す。トライ構造は一回の問い合わせで文中のある位置からはじまる全ての語を返すことが可能なので、効率的な辞書検索ができる。例えば、「海老名へ行く」という文字列を図 2.2 のトライで検索すれば、枝を一回たどるだけで、「海 (名詞)」「海老 (名詞)」「海老名 (固有名詞)」という語が見つかる。文中の全ての語を探す単純な方法は、文頭から一文字ずつ文字位置をずらしながら各位置で common prefix search を行うというものである。しかし、Aho と Corasick により提案された AC 法 [3] を用いれば、入力文を一回スキャンするだけで入力文に含まれる全ての語候補を取り出すことができ、トライによる方法と比べ検索速度は格段に向上する。Maruyama[4] はこの AC 法を用いて日本語形態素解析の辞書検索の高速化を行っている。だが、辞書のデータ格納領域が大きくなるという欠点がある。ここでは前者のトライによる方法を用いて以降の解説を行う。

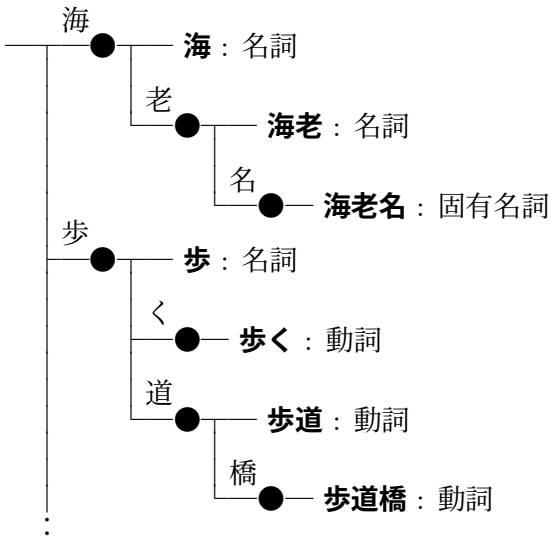


図 2.2 日本語トライ辞書

2.2.2 1 トークン = 1 語 の問題

わかち書きされる言語でも、単語は常に明白な単語境界文字列で区切られているわけではない。明白な単語境界を前提とした単純なトークン認識手法には限界がある。そこで、本章では、わかち書きされる言語をわかち書きされない言語と同じ方法で解析する方法を提案する。英語を例に、問題点とその解決のための方針を述べる。

図 2.3 に文字、Graphic Word、語の関係を示しておく。

単語内区切り曖昧性の問題 — 一つのトークンに複数の語が含まれている

語の構成要素に記号文字が含まれている場合、区切りの曖昧性が生じてしまう。

文	Dr. Lee and John’s son go to the McDonald’s in New York.
文字	|D|r|.| |L|e|e| |a|n|d| |J|o|h|n|’|s| |s|o|n| |g|o| |t|o| |t|h|e| |M|c|D|o|n|a|l|d|’|s| |i|n| |N|e|w| |Y|o|r|k|.|
Graphic Words	|Dr|.| |Lee| |and| |John’s| |son| |go| |to| |the| |McDonald’s| |in| |New| |York|.|
語 (Lexemes)	|Dr|.| |Lee| |and| |John|’s| |son| |go| |to| |the| |McDonald’s| |in| |New York|.|
形態素片	|Dr|.| |Lee| |and| |John|’|s| |son| |go| |to| |the| |McDonald|’|s| |in| |New| |York|.|

図 2.3 英語文の構成

“Mr.” や “Inc.” のように、ピリオドがトークンの末尾にある場合、それが文末記号なのか省略記号なのかという曖昧性がある。これは、わかち書きされる言語の「文の認識」という大きな課題であり、Palmer ら [5] によって研究されている。

アポストロフィがトークンの途中に含まれている場合も曖昧性が生じる。所有を表す “ ’s” が辞書にあれば解析文中の “John’s” を “John” + “ ’s” という二つの語として認識したい。しかし、“McDonald’s”(固有名詞) が辞書にあれば解析文中の “McDonald’s” を “McDonald” + “ ’s” ではなく一つの語としても解析したい。それゆえ、区切りの曖昧性を生じることになる。

ハイフンの場合は、ハイフンでつながれた文字列が辞書にあればそれを候補としたいし (例： “data-base”)、無ければハイフンを無視したい (例： “55-year-old”)。

これらの問題は単純なパターンマッチでは対処できない。そこで、わかち書きされない言語での方法を適用する。記号文字を含むトークン全てを記号文字で分割し (例： “McDonald’s.” → “McDonald” + “ ’ ” + “s” + “.”)、分割されたトークン列に対して common prefix search で語辞書を検索する。記号文字を含めた形で語が辞書に登録されていればそれも候補になる。

複合語問題 — 複数のトークンが一語になる

語の構成要素に空白文字が含まれている場合について考える。

空白文字などの明白な境界を利用した単純なトークン認識では、複合語固有名詞 “South Shields” のような空白文字を含む語が扱えない。このような場合 “South”(固有名詞) と “Shields”(固有名詞) を辞書に登録して解析することが考えられる。しかし、“South”(形容詞) + “Shields”(名詞複数形) と認識されてしまう危険も高くなる。“South Shields”(固有名詞) という空白文字を含んだ語を辞書に登録できれば、この種の誤りは減るであろう。

Penn Treebank[6] では、“New York” のような空白文字を含む固有名詞は “New” と “York” に分割され、それぞれに「固有名詞」という品詞タグが与えられている。そのため、このコーパスから得られる語も空白文字を含まない分割されたものになってしまう。このように空白文字を含んだまま単独の語として扱うべきものをわざわざ分割してしまうと、形態素解析処理における曖昧性の増加の要因になる。

この問題を解決するために、わかち書きされない言語での方法を適用する。これは、Webster ら [7] により最初に提示されたアイディアである。彼らはわかち書きされる語のイディオムや定型表現などを扱うため、空白文字を含む語の辞書登録を可能にし、トークン認識時にそれらを Table-look-up matching という方法で検索をする。ここでは語辞書を common prefix search で検索するという方法により、空白文字を含む語を扱う。

これらの問題を同時に解決するためには、「記号文字、及び、空白文字で区切られた文字列単位をベースに語辞書を common prefix search 」すれば良いという結論に達した。そのためには、この文字列単位を、言語非依存性と処理効率を考慮しきちんと定義する必要がある。これについては、次節で詳しく解説する。

2.2.3 形態素片

形態素片の導入

2.2.2 節で述べたように、わかち書きされる言語は単語境界が明白であると考えられてきたにもかかわらず、単語内区切り曖昧性、複合語の問題がある。

このような問題を解決する素朴な方法として、わかち書きされる言語をわかち書きされない言語と同じとみなし、「わかち書きされない言語を解析する方法」で解析するという方法が考えられる。英語を例に考えると、“They’ve␣gone␣to␣school␣together.” という文の全てのスペース (␣) を削除して、“They’vegonetoschooltogether.” という文を作り、これをわかち書きされない言語を解析する方法で解析すればよい [8]。しかし、このような方法は “They/’ve/gone/to/school/to/get/her/.” のような余計な曖昧性を含む結果を生んでしまう。

このような影響を調べるために実験を行った。Penn Treebank[6] の 128 万形態素 から学習されたパラメタ (品詞 trigram による状態遷移表と出現確率が付与された単語辞書) を用いた HMM ベースの形態素解析システム MOZ(2.3.4 節を参照) で、以下の 3 点

- わかち書き (segmentation) の精度
- 品詞付与 (part-of-speech tagging) の精度
- 解析速度

について、以下の 3 つの条件で測定を行なった。

- LXS: ベースラインの精度を出すために理想的な状態で解析を行なう。区切りの曖昧性が生じないように、全ての語が完全に認識できるようにした文を解析する。

  例: “`It␣’s␣Mr.␣Lee␣’s␣pen␣.`”

- NSP: 空白を全て消して、わかち書きしない言語を解析する方法で解析する。つまり全ての部分文字列が理論上、辞書検索の対象となる。

  例: “`It’sMr.Lee’spen.`”

- NOR: 空白を消さずに、わかち書きしない言語を解析する方法で解析する。NSP と同様に全ての部分文字列が理論上、辞書検索の対象となる。

  例: “It’s␣Mr.␣Lee’s␣pen.”

なお、テストデータは学習に用いたのと同じデータを使用した。

図 2.4 に実験結果を示す。スペースを削除した場合は区切りの曖昧性が発生するため、recall と presicion で評価した。

NSP の結果を細かく見てみると、“a way”, “a head”, “any more”, “work force” のような複数の語の連続をそれぞれ “away”, “ahead”, “anymore”, “workforce” のように一つの語として認識してしまう傾向にある。recall よりも precision が高いのはこのためである。また、“a tour”, “a ton”, “Alaskan or” を “at our”, “at on”, “Alaska nor” のように認識してしまう誤りもある程度見られた。前者のような区切りの曖昧性は conjunctive ambiguity、後者のような区切りの曖昧性は disjunctive ambiguity と呼ばれる [7][9]。conjunctive ambiguity による区切り誤りは 11267 個、disjunctive ambiguity による区切り誤りは 223 個あった。区切り曖昧性は精度以外にも性能に影響を与える。文の全ての位置から検索ができるので検索回数が増え、それにともない候補となる語も増えるため、解析時間が増大してしまう。実験では NSP の解析時間は、LXS の約 4 倍を要した。これは重大な問題である。また、NOR は NSP と比べ、余計な部分文字列の辞書検索が減り、解析候補が減るため、若干精度は高くなる。しかし、それでもまだ余計な部分文字列は多いため、やはり、解析時間は LXS の約 4 倍を要した。

より精度の高い効率的な解析を行うためには、わかち書きの情報を活かし、余計な曖昧性をできるかぎり排除できる単位を定義すべきである。また、特定の言語に依存しないように考慮する必要がある。

本章では、わかち書きされる言語のこのような問題を解決するために、効率的かつ洗練された方法を提案する。それは、文中での「辞書検索を始めて良い位置・終えて良い位置」を言語ごとに明確に定義し、それを元に common prefix search で辞書検索するという方法である。これは、わかち書きされない言語で採用されている方法を一般化したものである。わかち書きされない言語では「文字」の境界が辞書検索を始めて良い位置・終えて良い位置となる。このような「辞書検索を始めて良い位置・終えて良い位置」に囲まれた文字列を**形態素片** [10] と呼ぶ。この形態素片を各言語ごとに定義すれば、本節冒頭の英語の例のような非論理的な曖昧性を含むことなく、わかち書きされる言語とされない言語を統一的に扱える。形態素片は、言語非依存性や処理の効率を考慮してより厳密に定義されたトークンと言える。

ある言語の形態素片の集合は、その言語の辞書中の全ての語をより短い語に再帰的に分割して得られる文字列[1] の集合である。ただし、分割位置はその言語での「辞書検索

[1] 例えば、 “data-base” は “data”, “-”, “base” に、「海老名」は「海」「老」「名」に分割され

	Segmentation (Recall / Precision)	POS Tagging (Recall / Precision)	Analysis Time (Ratio)
LXS	100	96.98	1.0
NSP	99.52 / 99.67	96.52 / 96.69	4.3
NOR	99.87 / 99.91	96.84 / 96.88	4.2
MF	99.88 / 99.93	96.85 / 96.91	1.4

図 2.4 英語における区切り曖昧性を調べる実験

英語	I’m in New York.	→	[I][’][m]␣[in]␣[New]␣[York][.]
日本語	学校へ行きましょう。	→	[学][校][へ][行][き][ま][し][ょ][う][。]

図 2.5 形態素片認識の例

を始めて良い位置・終えて良い位置」でなければならない。また、**デリミタ**と呼ばれる文字列集合は除く。デリミタは文中で語の境界を表す空白文字などの文字列で、語の最初と最後には現れないものと定義する。以降、“␣” と表記する。英語ではアルファベットのみが連続する文字列、及び、全ての記号文字は形態素片であると定義できる。それゆえ、単語中の記号文字で分断される各文字列も形態素片である。例えば、英語文字列 “they’re” は “they”, “ ’ ”, “re” の 3 つの形態素片から成る。当然、複合語を構成する各単語も形態素片である。辞書に “New␣York” や “New” というエントリがあれば、“New”, “York” はそれぞれ形態素片であるが、“␣” はデリミタなので形態素片にはならない。また、“␣York” といった文字列は定義により語にならない。日本語、中国語などのわかち書きされない言語では全ての文字が形態素片であると定義できる。図 2.5 に文から形態素片を認識した例を示す。認識された形態素片は角括弧で囲って表されている。図 2.1、図 2.3 にも形態素片の例を示した。

る。

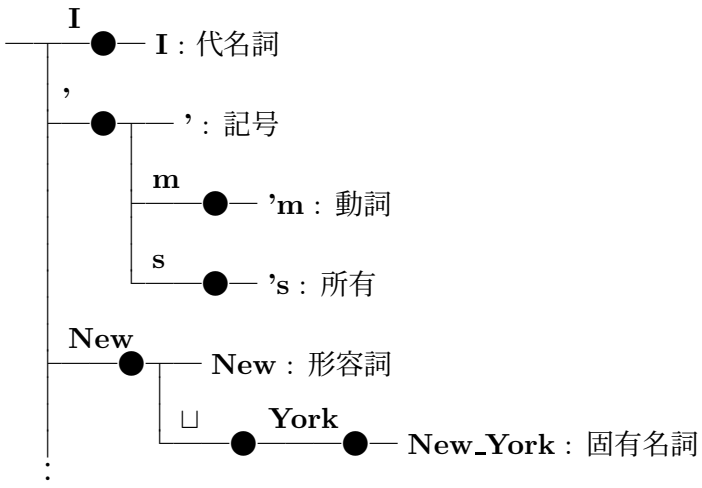


図 2.6 英語トライ辞書

形態素片を用いた方法では、わかち書きされない言語と同様にわかち書きされる言語の語辞書はトライに格納する。形態素片が枝のラベルになる。図 2.6 に形態素片ベースの英語トライ辞書を示す。図 2.5 の英語の例の [New] の位置から図 2.6 のトライを検索すれば、一回たどるだけで “New(形容詞)”, “New␣York”(固有名詞) という二つの語が見つかる。トライ辞書構築時と形態素片認識処理時には、連続する二つ以上のデリミタは一つのデリミタと見なして処理を行う。デリミタの連続には、特に言語的な意味は無いと仮定している。

形態素片認識の方法

形態素片は辞書を引き始める位置と引き終える位置を明確にし、わかち書きされる言語でも、効率的な common prefix search を可能にする概念である。しかし、ある言語の形態素片の集合を過不足なく定義することは難しい。そこで、「ユーザが簡単に定義できる必要最小限の情報」のみを用いた疑似的な形態素片の定義法を提案する。

わかち書きされる言語である英語を例に考えてみると「デリミタと記号文字で区切られる文字 (アルファベット) の連続」と「記号文字」の二種類が形態素片となり、形態素片認識にはデリミタと記号文字を定義する必要があることが分かる。わかち書きされない言語である日本語や中国語を例に考えると、各「文字」が形態素片になり、「文字」を定義すれば良いことが分かる。これらの考察により、次の 3 種類の情報を用いれば形態素片認識処理ができることが分かる。

1. 文字の定義、及び、全ての文字が形態素片になりうるかどうかの区別

   わかち書きされない言語では全ての文字が形態素片になりうる。つまり、これは、わかち書きされる言語かされない言語かを区別する情報である。

2. デリミタ辞書

   形態素片の境界として働き、それ自体は独立した形態素片にはならない文字列の辞書。語の開始と終了位置にはデリミタは現れない。

3. 形態素片辞書

   形態素片となる特殊な文字・文字列の辞書。記号文字のように形態素片の境界として働き、それ自体も形態素片として扱われる文字列などを格納する。

全ての文字が形態素片となるわかち書きされない言語では、(2),(3) は不要の場合が多い。

これらの情報の英語 (Penn Treebank[6] のフォーマットに準拠) での定義例を示す。

1. 1 文字 ≠ 1 形態素片 (わかち書きされる言語)
2. デリミタ辞書：空白文字 (␣)
3. 形態素片辞書：[.][,][:][;][’][-]···[$][%]···[n’t]

Penn Treebank では、 “don’t” などの縮約形は “do” と “n’t” に分割されタグ付与されているので、形態素片辞書に [n’t] が必要になる。

日本語での定義例を示す。

1. 1 文字 = 1 形態素片 (わかち書きされない言語)
2. デリミタ辞書：空白文字 (␣)
3. 形態素片辞書：なし

日本語はわかち書きされない言語であるが、デリミタを定義しておくと、わかち書きした文の解析もできる。わかち書きした日本語文は区切り曖昧性が減少する。

韓国語の通常の清書法では、句単位でわかち書きする。これは、日本語の文節に相当する単位である。わかち書きをする位置は、新国語表記法 [11] によって定められているが、必ずしも完全に守られているわけではない [12]。一般に、韓国語は日本語のようなわかち書きされない言語に分類できると言われている。ゆえに、形態素片の定義例は上記の日本語のものをそのまま用いることができる。しかし、わかち書きの境界の前後の品詞の分布には偏りがある。平野ら [13][12] は、境界内部では品詞 bigram を用い、境界

を越えての連接には品詞 trigram (境界も品詞の一つ) を用いることにより、わかち書き境界という情報をうまくとりこんでいる。わかち書き境界がスペース (␣) で表されるとすれば、この場合の韓国語での定義例は次のようになる。

1. 1 文字 = 1 形態素片 (わかち書きされない言語)
2. デリミタ辞書：なし
3. 形態素片辞書：空白文字 (␣)

もちろん、語辞書にスペース (␣) を登録し「わかち書き境界」などといった品詞を持たせておく必要がある。

ドイツ語はわかち書きされる言語であるが、複合名詞は区切りの曖昧性を持っている。例えば、*Staubecken* は、*Stau-becken* と区切れば「貯水池」、*Staub-ecken* では「ゴミ捨て場」という意味になる [14]。このような区切り曖昧性を扱うためには、わかち書きされない言語として処理を行えば良い。定義例をあげる。

1. 1 文字 = 1 形態素片 (わかち書きされない言語)
2. デリミタ辞書：空白文字 (␣)
3. 形態素片辞書：なし

形態素片認識アルゴリズムは、文の先頭から末尾まで 1 バイトずつずらしながら、形態素片・デリミタを探して行くという単純なものである。しかし、この方法による形態素片認識結果を用いれば全ての語を認識でき、実用上の問題は無い。

形態素片の効果を調べるためにさきほど同じシステムで実験を行なった。条件は、NOR とほぼ同じであり、違いは英語の形態素片定義を用いる点である[2] 。結果 (図 2.4 の “MF” の行) から、形態素片を用いることより、 NOR, NSP と比べ解析速度が著しく向上しているのが分かる。これは形態素片により、語の開始終了位置が明確になり、余計な部分文字列が解析候補になることが減ったからである。この影響で、精度も NOR, NSP と比べ上がっている。これらの実験により、形態素片の有効性が示された。

2.3. コンポーネント化と実装

本節では、形態素解析システム内部の様々な処理をそれぞれコンポーネント化した設計・実装について述べる。コンポーネント化は、形態素解析以外の用途への利用、特殊

[2] 例: “It/’/s/␣/Mr/./␣/Lee/’/s/␣/pen/.”

な機能の追加、言語に特化した処理の追加などに必須である。本節では、これらの目的を念頭に置いた設計・実装の方針について解説する。

本章で考える言語非依存の形態素解析処理の流れを次に示す。

1. 入力された解析対象文を形態素片列として認識し、辞書検索を簡単にする。
2. 形態素片列に対し語辞書検索を行い品詞候補、及び、語自体のコストを与える。
3. 語を区切り・品詞の曖昧性を保持したままトレリス (trellis) データ構造に格納する。同時に状態遷移の情報もチェックし格納する。
4. トレリスから最適解 (語の列) を選択する。
5. 結果を出力する。

このような処理の流れに基づき各処理をコンポーネントに分割した設計を行い、形態素解析ツールキット LimaTK を実装した [15][16][17]。LimaTK は図 2.7 に示すようなコンポーネントから成り立っている。全てのコンポーネントは独立しておりインターフェース等の仕様に従えば自作のコンポーネントと置き換えが可能である。

これらのコンポーネントのうちで言語依存性の高いものは、形態素片認識、辞書検索、未定義語処理である。形態素片認識と辞書検索は形態素片の導入により、言語依存部分がほぼ解消されたと言える。形態素片認識の実装については、2.2 節で説明した。辞書検索の実装については 2.3.1 節、最適解選択の実装については 2.3.2 節、未定義語処理の実装については 2.3.3 節で述べる。2.3.4 節では LimaTK を用いて実装した形態素解析システム MOZ について述べる。

2.3.1 辞書検索

辞書検索コンポーネントは形態素片列として認識された文から可能性のある語全てを辞書から獲得する。これらの処理の詳細については、2.2 節で既に述べた。現在の辞書検索コンポーネントの実装について述べる。

語辞書のデータ構造であるトライを単純に実装すると大きなデータ領域が必要になる。そこで現在は、データ格納領域が小さく済む 2 種類の方法で実装している。suffix array[18] を使うものと、パトリシア木 [19] を使うものである。パトリシア木はデータ消費量が若干大きいが高速であり、suffix array は若干低速であるがデータ消費量が小さいという特徴がある。suffix array による実装は高速文字列検索ライブラリ SUFARY([20], 4.1.2 節) を使用している。

語としてどのようなものを辞書に入れておくかということは言語や用途に依存した問題である。例えば、英語で動詞イディオム “look up” (“looking up”, “looked up” など

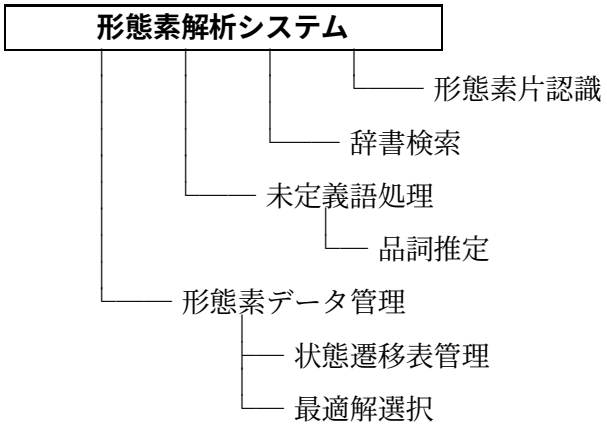


図 2.7 LimaTK の構成

も) を語として辞書登録したいとする。すると、以下の例の (1) では、登録された語が辞書検索の結果得られるが、(2) では分割されているのでイディオムとして認識されない。

1. I looked up the answer.

2. I looked the answer up.

Webster ら [7] はこのような連続した文字列で表現できないイディオム、定型表現などを形態素解析処理の段階で扱うために、辞書検索とパージングの知識・処理を融合するという枠組を提案している。コンポーネント化設計により、文字列として連続していない語の認識処理も、他のコンポーネントに影響を与えないように辞書検索コンポーネントなどの内部で実装できる。しかし、このような言語の構造に関わる処理は形態素解析より後の高次の処理で扱うべきであると考える。これはここでの目標が、形態素解析システムの単純化・効率化・言語非依存性を目指すことにあるためである。

2.3.2 最適解選択処理

最適解選択処理は HMM による方法を採用した。最適解選択に必要な HMM パラメタはある程度の量の品詞タグ付きコーパスがあれば得られるので、特定の言語の解析が容易に始められるという利点がある。具体的には、品詞タグ付きコーパスから、語と品詞 N-gram をカウントし、シンボル出力確率 (品詞別単語出現確率) と状態遷移確率 (品詞間、または、状態と品詞間の遷移確率) を計算し、動的計画法の一つであるビタビ・アルゴリズムで出現確率最大の解を求める。実際の実装は、積演算より和演算の方が効率

的に処理できるという理由から [21] パラメタ (確率値) の逆数の対数に適当な係数をかけた整数値 (**コスト**) を用いている。コストの和演算で最適な解を選択する方法は**コスト最小法**とも呼ばれており [1][22]、JUMAN[23]、茶筌 [24] といった日本語形態素形態素解析システムなどで採用されている。つまり、和演算による実装は、これらのシステムで長年用いられてきた、人手によって調整されたコスト体系 (単語コスト、接続コストなど) も利用できるという柔軟性を持っている。

特定の言語のために形態素解析を行うためにユーザが必要なものは、形態素片認識を行うための情報と、語と接続表 (HMM パラメタ) だけである。HMM パラメタは、十分な大きさの品詞タグ付きコーパスとユーザの望む統計モデル (bigram, trigram, variable memory model[25] など) に基づいた学習プログラムがあれば得られる。

形態素情報管理コンポーネントは、前述の方法により最尤解選択を行う。これは形態素解析における解選択の一般的な実装方法である [1][22]。文頭から文末へ向かって、一語ずつトレリス (ラティス) 構造に格納してゆき、そのときにその語までの部分解析のコストを求める。最適解は、文末から文頭へ向かって、最適な部分解析のコストを持つノードを順次辿れば得られる。格納の際に必要になる、状態遷移 (接続) にかかるコストと遷移先状態は、状態遷移表管理コンポーネントから得る。状態遷移表管理コンポーネントは、現在の状態と次の品詞をキーに状態遷移のコストと遷移先状態を返すという単純な仕事をする。

2.3.3 未定義語処理

未定義語処理コンポーネントは辞書に登録されていない語に対して品詞推定を行う。未定義語の品詞推定は統計的な方法と人手による規則などのヒューリスティックを用いる方法がある。

統計的な方法は、未定義語が全ての品詞を持つと仮定し、トレリスでの曖昧性解消処理の段階で品詞 N-gram の統計値により最適な品詞を自動的に選ぶという方法である [21]。これは、言語に依存しない実装が可能である。しかし、この方法ではデータ格納領域が増大してしまい、処理効率が悪い。そこで、未定義語が全ての品詞を持つのではなく、あらかじめ「未定義語が推定されうる品詞」の集合を限定する方法が考えられる。例えば「この言語の未定義語は『名詞』か『固有名詞』である」と定義すれば、曖昧性解消処理で未定義語の品詞はどちらかに選ばれる。この方法は完全な推定とは言えないがデータ格納領域の増大を押えることができる現実的な方法であり、茶筌 [24] で採用されている。実用性と性能のバランスの良さから、LimaTK の未定義語処理コンポーネントの標準の機能として採用した。

ヒューリスティックによる方法は、例えば、英語ならば、「文中で大文字で始まるなら

固有名詞」「-tion で終われば名詞」といった規則を用いて品詞を推定する方法である。これは言語に依存する方法なので、言語ごとに処理系を実装する必要がある。LimaTK ではこのようなルールを埋め込むためには、統計的手法による未定義語処理コンポーネントを修正するか、まったく新しく作り直す必要があるが、作り直す場合でもインターフェース規約を守れば他の処理に影響を与えずに実装できる。

未定義語の長さ、すなわち、未定義語がいくつの形態素片で構成されるかを決定する処理も難しい。理論的にはある位置から始まる全ての長さの部分形態素片列が未定義語の候補になる可能性がある。しかし、これでは候補が増大してしまい処理効率に問題がある。日本語のように字種にバリエーションのある言語は、連続する漢字列・カタカナ列・記号列などを一まとめにするといった字種によるまとめ処理により未定義語の候補を限定できる [22]。このような字種による未定義語候補の決定処理は JUMAN[23] や 茶筌 [24] の様な日本語形態素解析システムに採用されている。単純なまとめ処理ではなく、字種による語の長さの分布の違いに着目して未定義語処理を行うという研究もある [26]。わかち書きされる言語ではこれまでこの問題は起こらなかった。しかし、本研究では形態素片という概念を導入したため、わかち書きされる言語でも問題になるようになった。未定義語に複合語は無いと仮定すれば、ある位置から始まり次のデリミタまで間の全ての部分形態素片列を未定義語の候補とすれば良い。この仮定は正しいものではないが、実用上はさほど問題なく現実的である。

そもそも未定義語処理は言語依存性の高い処理であり、品詞推定精度の高い共通の枠組を構築するのは困難である。より精度の高い処理を求めるユーザはやはりプログラムの調整を行う必要がある。ゆえに、各言語共通に利用できる最低限の機能と調整の行いやすい枠組で実装を行った。言語非依存性とユーザの利便性と処理効率のバランスを考慮した実装と言える。実装方針をあげておく。

- 未定義語の品詞推定 : あらかじめ「未定義語が推定されうる品詞」の集合を定義し、最適解選択処理にまかせる
- 未定義語の長さの決定 :「形態素片」「字種によるまとまり」「デリミタに挟まれた領域」を未定義語を構成する単位に選択でき、未定義語を構成する「最大単位数」も指定できる

2.3.4 ツールキットによる形態素解析システムの実装

LimaTK を用いて、簡単な多言語対応形態素解析システム MOZ を作成した。日本語、英語、中国語、韓国語など様々な言語を実装した。図 2.8 に解析結果の例を示す。各行はそれぞれ一つの形態素を表し、各列は左から、「見出し文字列」「各形態素の持つコスト」

「品詞コード」「状態コード」「解析には用いないその他の情報 (角括弧で囲まれている)」[3] となっている。使用した言語データを次に示す。

日本語

そんな感じがします。

```
そんな 3075 226 3648 [Y:ソンナ BF:そんな P:連体詞 Pr:100/3864]
感じ 6960 198 1585 [Y:カンジ BF:感じ P:名詞-一般 Pr:37/144546]
が 1306 59 744 [Y:ガ BF:が P:助詞-格助詞-一般 Pr:17509/82739]
し 0 117 2688 [Y:シ BF:する P:動詞-自立/サ変・スル/連用形 Pr:10638/10638]
ます 3048 73 1666 [Y:マス BF:ます P:助動詞/特殊・マス/基本形 Pr:813/30431]
。 1 11 2072 [PP:記号-句点 Y:。 BF:。 P:記号-句点 Pr:27418/27452]
```

英語

```
What is a word?

What 2208 20 1835 [P:WP Pr:218/3156]
is 946 62 1373 [P:VBZ Pr:8789/27619]
a 1207 28 1384 [P:DT Pr:25820/111243]
word 6629 85 117 [P:NN Pr:59/179722]
? 3771 54 202 [P:. Pr:556/53362]
```

中国語

人力基盤構築不能促成.

```
人力 5929 28 28 [P:Na Pr:185/372140]
基盤 8286 28 28 [P:Na Pr:9/372140]
構築 6816 70 70 [P:VC Pr:17/106692]
不能 3989 78 78 [P:D Pr:1003/167440]
促成 8473 32 32 [P:VH Pr:2/105135]
.  2007 31 31 [P:PERIODCATEGORY Pr:6046/79413]
```

図 2.8 形態素解析例

[3] 出力を人間にも読みやすくするための情報であり、解析時には一切使用されない。形態素辞書のこの項目に記述されている文字列をそのまま表示する。フォーマットは任意であり、後処理の目的に合わせ、形態素辞書を作成する段階で自由に変更できる。英語、中国語の例では、品詞名 (P) と形態素の出現確率 (Pr) が、日本語の例では、読み (Y)、基本形 (BF)、品詞名 (P)、形態素の出現確率 (Pr) がこの項目に含まれている。形態素の出現確率は「各形態素の持つコスト」を算出する基になった値である (Pr : この形態素のコーパスでの出現回数/品詞のコーパスでの出現回数)。

日本語 RWCP の品詞タグ付きコーパス [27](約 92 万形態素) から品詞 trigram モデルでパラメタ学習を行い、さらに茶筌 [24] の語辞書エントリを追加した。解析精度は、同様の方法で作成された解析用データを用いた茶筌のものと同等で、インサイドデータで Recall, Precision とも 97% 程度である。

英語 Penn Treebank[6] の品詞タグ付きコーパス (約 128 万形態素) から品詞 trigram モデルで学習を行い、電子化テキスト版 Oxford Advanced Learner's Dictionary[28] の辞書エントリを追加した。語幹 (stem) 情報も同じく Oxford Advanced Learner's Dictionary から補完した。解析精度はインサイドデータで 97%、アウトサイドデータで 95% 程度である。

中国語 台湾の中央研究院の品詞タグ付きコーパス [29] (約 210 万形態素) から品詞 bigram モデルで学習を行った。解析精度は Recall, Precision ともインサイドデータで 95%、アウトサイドデータで 91% 程度である。

未定義語品詞推定のチューニング、高次の N-gram の利用、スムージングなどより高度な統計的手法を用いれば、それに応じて精度は向上する。しかし、本研究の目的は精度の向上ではないので、これ以上は追求しない。

2.4. 関連研究

Webster ら [7] はわかち書きされる言語 (英語) のイディオムや定型表現が扱えるように、英語のデリミタで区切られた単位をわかち書きされない言語 (中国語を想定している) の文字に対応させわかち書きされない言語での解析方法を適用している。しかし、"They've" のような単語内区切り曖昧性の問題 (2.2.3 節) には言及していない。曖昧性の解消については、確率を使った方法が適用できると言及されているのみである。

わかち書きされる言語における境界の曖昧性解消について、Mills はわかち書きが不明瞭な古代英語の解析の際に、わかち書きされない言語で用いる形態素解析手法を単純に適用している [8]。これは、2.2.3 節で説明した、文の全てのスペース (␣) が削除された英文をわかち書きされない言語の解析手法で解析する方法である。しかし、この研究での解決法は、目的から分かるように、言語非依存ではない。また、単純なマッチング、ヒューリスティックによる力まかせな方法に基づいている。

言語非依存な形態素解析の枠組として、Kimmo の二段階形態素論 (two-level morphology)[30] が知られている。簡単に言うと、オートマトンを用いてトークン認識と品詞付与を同時に行う方法である。

本章では、わかち書きされる言語で考えられる分割の可能性、及び、わかち書きされる言語とされない言語との整合性も考慮し、形態素片という概念を導入した。形態素片

はこれまで漠然と定義されていたトークンに替わる概念である。これにより言語非依存の枠組を構築した。このアイディアは極めて単純ではあるが、効率的なトライ辞書検索の実現とトークン認識における非論理的な曖昧性の排除を達成した。

本研究の成果物である形態素解析ツールキット LimaTK の柔軟性を示す例として文節まとめあげと形態素解析の融合についての研究 [31] をあげる。英語を念頭に考えると、これは品詞付与と名詞句 (BaseNP) 認識を HMM により同時に行うという方法である。スペース (␣) を「名詞句始まり」「名詞句終わり」「名詞句の間 (終わりであり始まりである)」「名詞句途中」「名詞句の外部」のタグ (品詞) を持つ語として扱い、名詞句区切り情報と品詞情報の付与されたコーパスから HMM パラメタを学習し、それを用いて最適なものを選ぶという手法である。この手法は、形態素片を以下の情報で定義すれば MOZ で簡単に実装できる。

1. 1 文字 ≠ 1 形態素片 (わかち書きされる言語)
2. デリミタ辞書：なし
3. 形態素片辞書：スペース (␣)

このように、LimaTK は様々な言語を解析できるだけでなく、形態素解析以外の用途にも適用できる汎用的な HMM パーザとして柔軟に利用できる。

2.5. おわりに

本章では、言語に依存しない形態素解析の枠組の提案と、形態素解析の内部処理のコンポーネント化によるツールキットの設計・実装を行った。従来は、わかち書きするか否かという言語の特徴により大きく処理が異なる形態素解析処理を、形態素片という辞書検索単位を定義したことにより、言語非依存の共通の枠組で行えるようになった。また、形態素解析の内部処理のコンポーネント化により、言語非依存化のみならず様々な改良や他の言語処理への適用が行いやすくなった。

第3章

品詞タグ付きコーパスの蓄積とそれを利用した解析

本章では、品詞タグ付きコーパスの蓄積とそれを利用した解析に焦点をあて、コーパス作成作業者が実際に使う直観的に理解しやすいグラフィカル・ユーザ・インターフェースによるタグ付きコーパス作成支援システムを提案する。そこで作業者に提示するタグを付与する形態素解析の手法として、タグ付きコーパスと人手で作成されたルールを融合し蓄積途中の少量のコーパスでも十分に利用できる手法と、タグ付きコーパスを直接部分文字列検索しマッチした部分文字列とそれに付与されたタグを元に形態素解析を行なう手法を提案する。

3.1. 品詞タグ付きコーパス作成支援環境

3.1.1 はじめに

近年、日本語テキストの電子化が急激な勢いで進んでいるが、大部分が加工されていない生のテキストデータである。自然言語処理に限らず様々な分野で基礎データとして品詞タグ付きコーパスの需要は高く、迅速に整備していく必要がある。

しかし現在の形態素解析技術は完全ではないので、精度の高い品詞タグ付きコーパスを作るには、最終的には人手による修正が不可欠である。そのためにも効率の良い作業支援システムが必要となる。

今までに、品詞タグ付きコーパスの作成支援に用いられてきたシステムは、テキストエディタベースの簡易的なものが多い。例えば、 Penn Treebank[32] 作成時に用いられたものは GNU Emacs ベースのマウス操作によるユーザインターフェースである。これは一通りの形態素解析結果を表示し、品詞の誤りがあれば作業者がその単語をマウスで選び正しい品詞を打ち込み修正するいう単純なものである。

しかし、このようなシステムは、単語の品詞の修正のみが目的であり、日本語のようにわかち書きのされていない言語には不可欠の、単語境界の修正が容易ではない。また品詞情報の修正という点から見ても到底使いやすいとは言えない。

3.1.2 品詞タグ付きコーパスの作成支援システム

そのような理由から、使いやすさに重点を置いた、品詞タグ付きコーパスの作成支援システムを開発した。

初期のシステムである、Vi JUMAN[33] は、形態素解析システム JUMAN[23] による形態素解析結果を視覚化し品詞タグ付きコーパスの作成支援を行うシステムである。図 3.1 に実行画面を示す。その後、奈良先端科学技術大学院大学で、JUMAN から派生した形態素解析システム 茶筌 [24] が開発されたため、それに対応した支援システム Vi Cha を開発した。ここでは、これらの品詞タグ付け支援システムを総称して単に「支援システム」と呼ぶことにする。支援システムの特徴を以下に挙げる。

- 形態素解析システムの解析結果をグラフ状に図示することにより曖昧性が視覚的に理解できる。
- グラフ状に図示された解析結果からマウス操作のみで、任意にパスを選ぶことができる。さらに文の一部だけ制約を緩めて再び解析する機能も実装されており、最初の解析結果で得られなかった形態素を出現させることもできる。

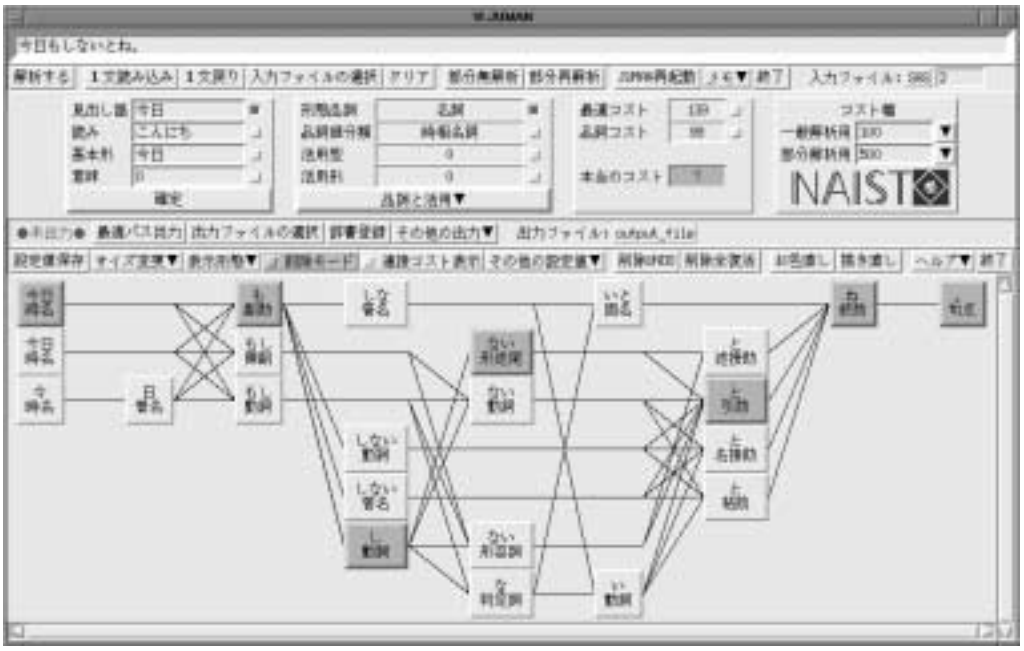


図 3.1 Vi JUMAN の実行画面

- メニュー選択方式により、形態素解析システムの文法体系に基づき矛盾なく品詞情報を選択することができる (図 3.2)。これにより容易に新単語の登録・品詞や活用の修正ができる。
- 前記の方法により品詞情報を修正した単語をボタン一つで形態素解析システムのユーザ辞書へ登録でき、次の解析から反映させることができる。

機能の詳細については文献 [33] を参照されたい。

3.1.3 学習機能

目的

このような GUI ベースの品詞タグ付きコーパス作成支援システムとして利用する際の不満として、一度行ったパス修正を別の文についても繰り返し行わなくてはならないという点が挙げられる。例えば、「…によると、」の解析結果「…-に (格助詞) -よる **(時相名詞)** -と (格助詞) -、」を「…-に (格助詞) -よる **(動詞)** -と (格助詞) -、」に修正しても、次にまた「…

図 3.2 形態素解析システムの文法体系に沿った品詞情報の選択

によると、」を含む文を解析すると「…-に (格助詞) -よる **(時相名詞)** -と (格助詞) -、」となってしまい再度の修正が必要となる。これは作業者にかなりの心理的負担を与える。

新聞を解析していくと、十数文に一回という頻度でこのような事態が生じる。これに対処するために、 修正したパスを事例データとして蓄積しておき、それらを用い形態素解析システムの解析結果を自動的に修正にするという処理を行う。

処理の流れ

支援システムでの、品詞タグ付きコーパス作成作業の流れは以下の通りである。

1. 一文を形態素解析システムで解析する。
2. 形態素解析システムの結果がグラフで表示され、コスト計算による最適パスが強調される (図 3.1)。
3. 人間が束図をマウスでクリックし、パスを修正する。
4. 修正したパスを品詞タグ付きコーパスのデータとして出力する。

提案する学習機能では、まず (4) の段階で、形態素解析システムが選択した最適パスと作業者が修正したパスの差分をとることによって、修正した部分 (形態素列) を取り出し事例データ (正例) として蓄積していく。そして (1) の段階の直後に、事例データをも

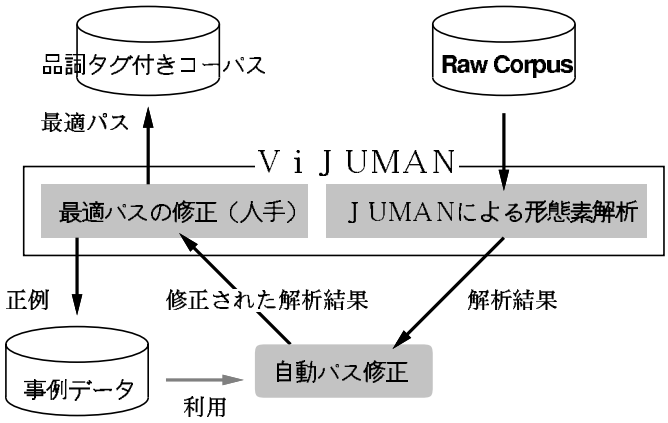


図 3.3 処理の流れ

とに解析結果を修正する。これにより一度人手で修正すれば、以降、同じ事例が現れたときに最適パスが自動的に修正されるので、人手による作業が軽減される。図 3.3 に全体の処理の流れを示す。

実装では、事例データの蓄積・検索にパトリシア木を用いている。パトリシア木とは、ビット単位のトライ中で枝が 1 本しかない無駄なノードをまとめて圧縮した効率の良いデータ構造である (図 3.4)。詳しくは、文献 [34][35] を参照されたい。

正例の蓄積

最適パスと修正したパスの差分をとることにより得られる正例を事例データとして蓄積する。

例えば、「先日、彼と食べた。」という文の場合、図 3.5 のような手順で正例を得る。そして、正例部分の表層文字列をキー、正例を固定長ビットでコード化したものを検索される内容としてパトリシア木に入れていく。

最適パスの自動修正

形態素解析システムの出力結果中の最適パスを、用例を用いて自動修正を行う。

解析した文に正例と同じ文字列が存在し、形態素解析システムの結果のグラフ中に正例と同じ形態素列がある場合、その形態素列を最適パスに含めるよう形態素解析結果を修正する。解析した文に正例と同じ文字列が存在するが正例と完全にマッチする形態素

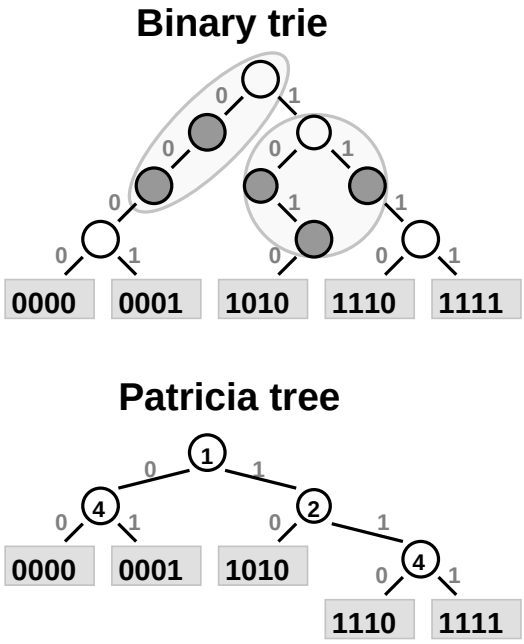


図 3.4 パトリシア木

ノード上の数字は何ビット目をチェックするかを表す。

列がない場合、後述の条件にあえば、その正例の形態素を作り、最適パスに含めるよう形態素解析結果を修正する。

パトリシア木からの正例の探索

形態素解析結果のグラフから正例と同じ形態素列を探す第一段階である。全ての正例に対して単純にマッチングを行うと計算量がかかりすぎるため、まず、ありそうな正例を絞り込むという処理を行う。

解析した文の先頭からポイントを次の単語区切りの場所までずらしながら、そのポイントから始まる文字列でパトリシア木を探索する (図 3.6)。

以下に述べる方法により、解析した文のそのポイントから始まる全ての正例候補をパトリシア木から引き出すことができる。

1. コード列をビット列とみなし、パトリシア木を探索する。

2. 途中に敷居ビット (この場合、16 の倍数) をチェックするノードがあれば、そこの左 (0) 側のノードのデータを取り出し、検索ビット列と矛盾が無いかチェックする。矛盾があれば終了する。
3. いきどまりならば、データを取り出し、検索ビット列と矛盾が無いかチェックする。

図 3.7 に例を示す。

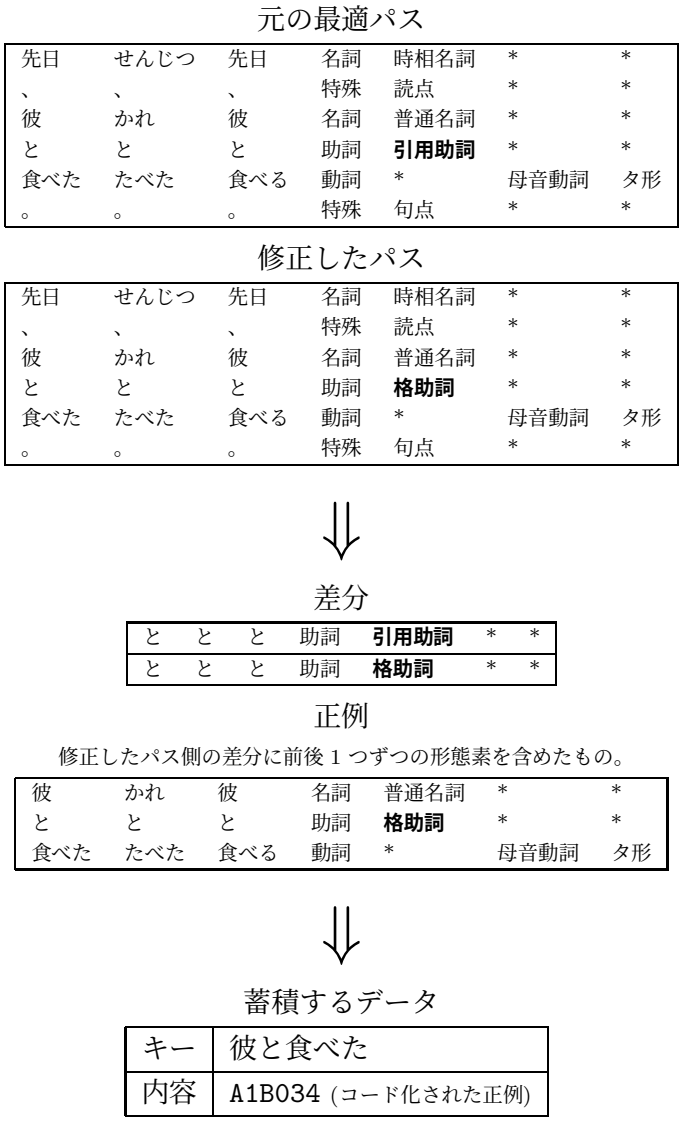


図 3.5 正例の蓄積

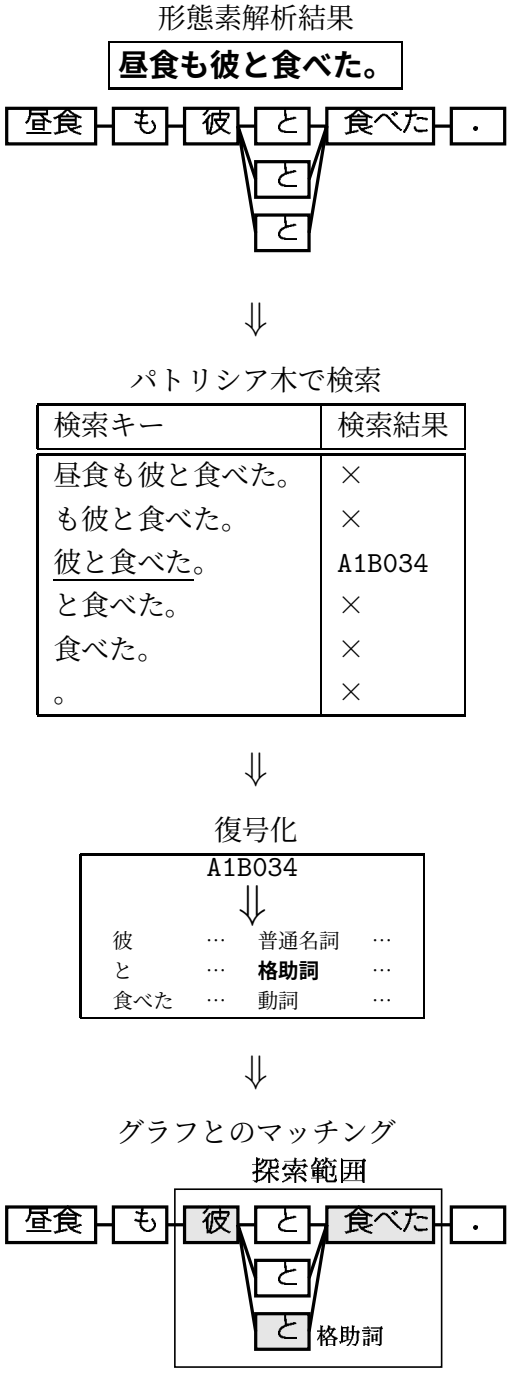


図 3.6 正例のマッチング

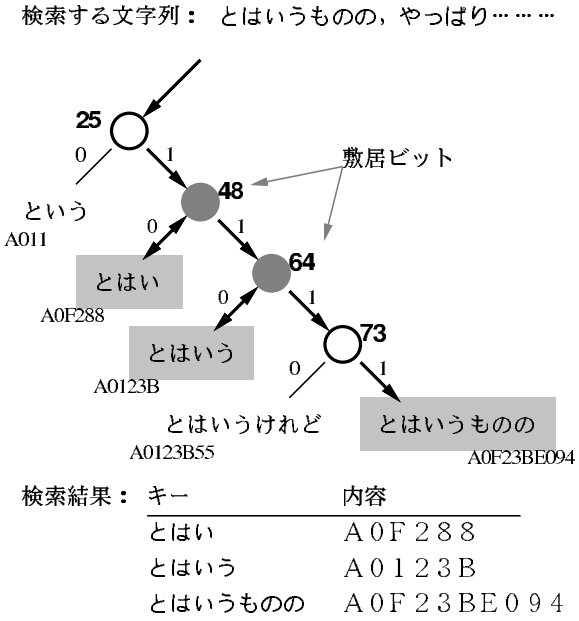


図 3.7 パトリシア木による正例の検索

形態素解析結果からの正例の探索とコスト修正

第二段階として、パトリシア木から引き出した正例候補とマッチする形態素を形態素解析結果のグラフから捜し出す。

さきほどの文字列マッチングでマッチした範囲 (何文字目から何文字目か) から、グラフの中での探索範囲を計算し、その範囲内で形態素のマッチングを行う (図 3.6)。マッチングに成功した場合、それらの形態素同士の連接コストや品詞コストを 0 にする。また、全ての正例候補とのマッチングに失敗した場合、(最長文字列の) 正例の両端の形態素が探索範囲の両端に存在すれば、新たにその正例と同じ形態素を作り、それらの連接コストや品詞コストを 0 にする。この処理により、最適パスに必ず正例の形態素が含まれることになる。

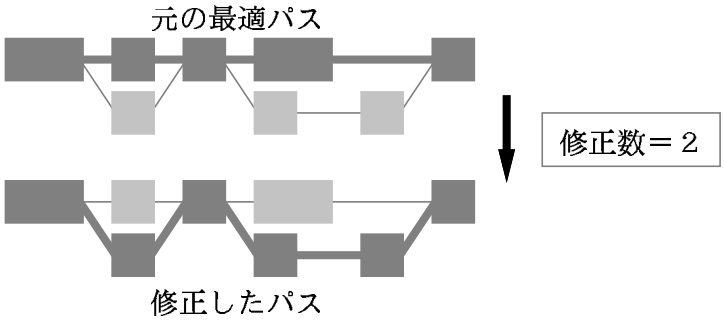


図 3.8 修正数の数え方

実験

実験方法

新聞コーパスのテキストデータと品詞付きデータを用い、品詞タグ付け作業のシミュレーションを行い、パスの修正数をカウントした。修正数とは、図 3.8 に示す通り、元の最適パスと修正後のパスの差分の数である。シミュレーションには支援システムのプログラムから GUI の処理を削ったものを用いている。

実験結果と考察

表 3.9 に実験結果を示す。再修正数とは一度修正したパスとまったく同じパスを再度修正する回数である。

どちらの実験でも自動パス修正の回数に比べ学習による修正数の減少分の方が少なくなっている。これは、自動パス修正が修正数減少に寄与しない、つまり、自動パス修正の結果、別のパス修正が必要となることがあるからである。

例えば、図 3.10 に示すように、「自由化などの」の解析結果から「化-など **(名詞性名詞接尾辞)** -の」を学習すると、「文化などの」の解析結果を自動パス修正したとき「文-化-など-の」が最適になってしまい、結局「文化-など-の」に修正しなければならない。

また、解析結果の同じ部分に影響する複数の正例が蓄積されると、自動パス修正が意味をなさなくなるという問題がある。これを「**ゆれ**」と呼ぶことにする。

例えば、「太郎とみた」の「と」は、「私の勘では真犯人は**太郎とみた**。」と「ＴＶを**太郎とみた**。」では違う。前者では「太郎-と **(引用助詞)** -みた」、後者では「太郎-と **(格助詞)** -みた」となる。このような形態素列がどちらも正例として蓄積されると、自動パス修正

は機能しなくなる。

しかし、各形態素の表層レベルまで考慮して正例を蓄積していくので、「ゆれ」は非常に頻度の低い現象と思われる。実際、この実験に先立って行われた試験実験では、全ての「ゆれ」はタグ付け間違いによるものであった。本研究の実験は、そのタグ付け間違いを全て正したものを用いて行っている。

実験 1：日経新聞 600 文

	学習なし	学習あり
修正数	1009	959
再修正数	51	0

自動パス修正の回数	65
修正数の減少分	50

蓄積された正例	959
利用された正例	38

実験 2：日経新聞 300 文

	学習なし	学習あり
修正数	535	520
再修正数	16	0

自動パス修正の回数	20
修正数の減少分	15

蓄積された正例	520
利用された正例	16

図 3.9 実験結果

実験 1 と実験 2 は記事データ及びタグ付け作業者が異なる。

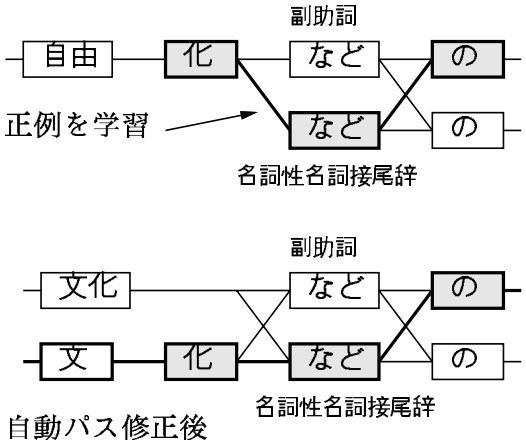


図 3.10 自動パス修正が修正数減少に寄与しない例

正例データの利用

今回の実験では蓄積された正例のほとんど (約 96%) は利用されていない。これは副作用を避けるために表層レベルまで見て蓄積しているためで、無駄が多くなってしまうのは避けられない。利用された正例のうち頻度の高いものを表 3.11 に挙げる。

利用頻度にかかわらず正例データを整理し抽象化することによって、有用なデータとすることができる。一つの方法は、先頭と末尾以外の形態素が共通である正例のそれぞれの先頭と末尾の形態素を品詞レベルで抽象化するというものである。例えば、「みる-と **(述語接続助詞)** -、」「する-と **(述語接続助詞)** -、」「買う-と **(述語接続助詞)** -、」などがあった場合、「**動詞/基本形** -と **(述語接続助詞)** -、」と整理できる。

に	よる **(動詞)**	と **(述語接続助詞)**	、
(文頭)	一方 **(接続詞)**		、
みる	と **(述語接続助詞)**		、
する	と **(述語接続助詞)**		、
した	こと **(名詞)**	に **(格助詞)**	ついて
した	もの **(名詞)**	で **(判定詞)**	、
	⋮		

図 3.11 使用頻度の高い正例

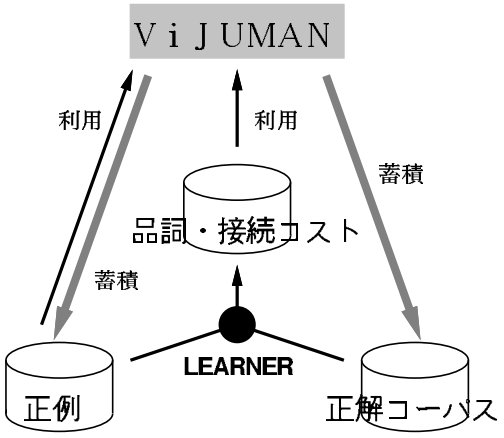


図 3.12 正例と正解コーパスからの品詞・連接コストの学習

これらのデータは形態素解析システムが最適パスの選択をたまたま間違えたために得られたものであり、統計的に普遍的な価値は無いが、形態素解析システムでの解析の弱点を表すデータとして利用できる。

例えば、今までの連接規則に加え形態素列を連語として登録・利用できる JUMAN[23] や茶筌 [24] で、大量の整理された正例を連語として用いることによって、形態素解析精度が向上する。

また、正例の蓄積だけでなく、ある程度品詞タグ付きコーパスが蓄積されたら bigram 統計をとり品詞・連接コストを更新する処理も並行して行うつもりである (文献 [36] の 2.3.1 を参照)。その際に、整理された正例も bigram 統計のデータとして利用することで、その時点での解析の弱点を早く克服できるように品詞・連接コストが更新されていくと期待できる。図 3.12 に概観を示す。

3.1.4 おわりに

本研究では、品詞タグ付きコーパス作成時の作業負担を事例データの学習により軽減できることを示した。また、実験過程から、今回の手法は作業負担の軽減だけでなく、品詞タグ付きコーパスの一貫性チェックにも利用できることが明らかになった。

3.2. コスト最小法と確率モデルの統合による形態素解析

3.2.1 はじめに

形態素解析は、構文解析・意味解析などの高度な自然言語処理の基盤として重要な技術である。

日本語形態素解析において、今までに研究されてきた最適解の選択手法は大きく次の二つの流れに分けれられる。

一つは、**人手により作成された制約や優先規則を用いた方法**である。これは、人間がさまざまな言語現象をおおまかにとらえて、規則などの形に抽象化したものと言える。記述量や理解の容易さといった面において効率の良いものであるが、記述者の考慮外の現象がとらえきれていないことが多い。そのような例外的な現象を規則として追加・修正することによって、解析精度を向上させていくのが一般的である。しかし、それにつれて、規則は次第に複雑になり、保守・管理が人間の手には負えなくなってしまう。つまり、一つの規則を修正することによって他に与える影響が予測不能になり、全体的な精度をあげていくのが難しくなる。しかし、これらの規則は今までの経験の蓄積であり、有効な資源と言える。このような方法に基づくシステムとして、Breakfast[37]・茶筌 [24]・JTAG[38]・JUMAN[23] などが知られている。

もう一つは、**品詞タグ付きコーパスから学習された確率パラメタを用いた方法**であり、人手による手間・解析精度などの問題をある程度解決できる [39]。しかし、ある程度の精度を出すには大量の品詞タグ付きコーパスが必要となり、それらが整備されていない未開拓な分野には不向きである。

本節では、未開拓な分野で高い解析精度を得るための手法として、少量の品詞タグ付きコーパスから学習された確率パラメタと有用な言語資源である人手により作成された制約や優先規則の統合による形態素解析を提案する。実際には、コスト最小法に基づく日本語形態素解析システム 茶筌 で用いられる人手により与えられたコストと、品詞 bi-gram モデルに基づき品詞タグ付きコーパスから学習された確率パラメータを統合し新たなコストを作成する。この手法を用いることで、品詞タグ付きコーパスの作成の際のブートストラップを効率良く行うことができる。

3.2.2 最適解の選択手法

本節では、**人手により作成された制約や優先規則を用いた方法**のうち、現在流布している日本語形態素解析システムの多くが採用している人手により与えられたコストを用いたコスト最小法に着目する。また、**品詞タグ付きコーパスから学習された確率パラメタを用いた方法**のうち、品詞 bi-gram モデルに基づく統計的学習により得られた確率パラメタを利用する方法に着目する。

どちらの方法も、コスト・確率パラメタへの値の与え方が異なるだけで、基本的には、同じアルゴリズム (ヴィテルビ・アルゴリズム) で、最適解の選択を行うことができる。

ここでは、コスト最小法及び品詞 bi-gram モデル[1] に基づく最適解の選択法について説明し、この二つの手法の関係について述べる。

コスト最小法

分割された各形態素間の接続と形態素そのものにコストを与えて、その合計が最小の解を優先するという手法である。

分割された各形態素間の接続に与えられるコストを**連接コスト**、形態素そのものに与えられるコストを**形態素コスト**と呼ぶことにする。

コスト最小法を用いた日本語形態素解析システムでは、これらのコストを人手により与えることが多い。

例えば、図 3.13 のように「名詞と判定詞の連接コストは 30」「名詞『さかな』の形態素コストは 100」というようにコストを設定する。この場合の最適解は、文頭から文末までの連接・形態素コストの合計が最小 (240) である「さかな:名詞、だ:判定詞、よ:助詞」となる。

実際には形態素一個一個にコストを与えるのは大変なので、品詞ごとに設定することが多い。

品詞 bi-gram モデル

ある品詞の次にある品詞が現れる確率 $p(t_i \mid t_{i-1})$、ある品詞のときにある形態素が現れる確率 $p(w_i \mid t_i)$ を用いて、これらの確率の積が最大になるパスを優先する手法である。確率の積 P は以下のような式で表される。w_i は形態素、t_i は品詞を表す。

[1] 実際には品詞 trigram モデルが用いられることもある (文献 [22] に詳しい) が、人手で trigram ルールを作成することは非現実的であり、本研究の目的である人手によるルールとの統合には適さないので採用しなかった。

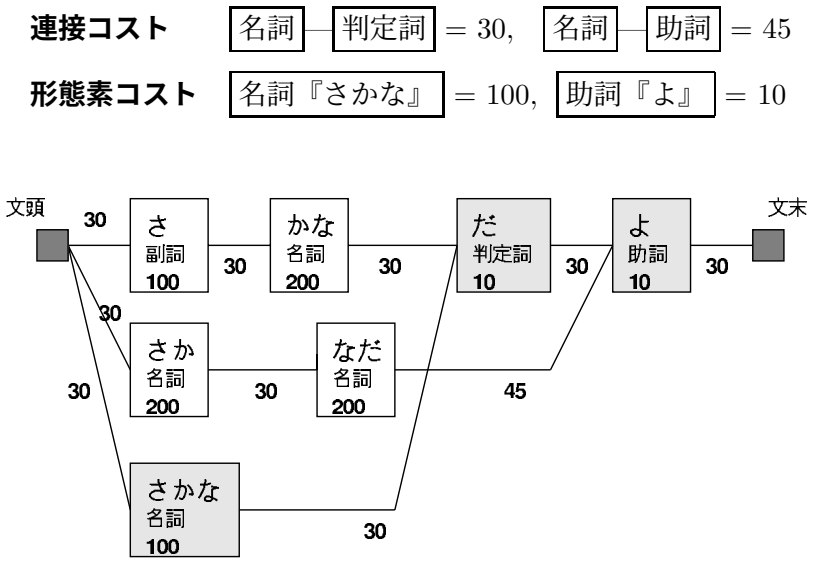


図 3.13 コスト最小法

$$P(w_1, \cdots, w_n) \cong \prod_{i=1}^{n} p(w_i \mid t_i) p(t_i \mid t_{i-1})$$

図 3.14 に例を挙げる。名詞の次に判定詞が現れる条件付き確率は p(判定詞 | 名詞) = 0.03、名詞というカテゴリー内で『さかな』という語が現れる確率は p(さかな | 名詞) = 0.05 となっている。この場合の最適解は、文頭から文末までの確率の積が最大 (4.5×10^{-7}) になる「さかな:名詞、だ:判定詞、よ:助詞」となる。

このような条件付き確率 (確率パラメタ) は、品詞タグ付きコーパスがあれば最尤推定によって簡単に求めることができる。例えば、ある品詞タグ付きコーパス中で、名詞が 100 回現れて、その名詞の直後に判定詞が 30 回現れたとすると

$$p(\text{判定詞} \mid \text{名詞}) = \frac{30}{100} = 0.3$$

と推定でき、また、名詞 100 回のうち「さかな」が 5 回現れたとすると

$$p(\text{さかな} \mid \text{名詞}) = \frac{5}{100} = 0.05$$

と推定できる。

p(判定詞 | 名詞) = 0.3, p(助詞 | 名詞) = 0.02
p(さかな | 名詞) = 0.05, p(だ | 判定詞) = 1

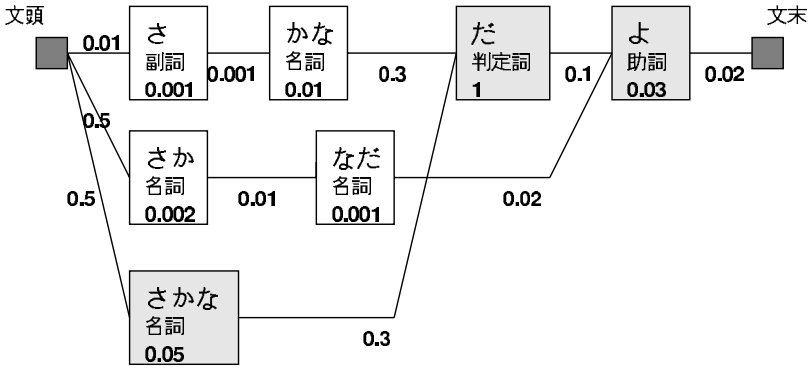


図 3.14 品詞 bi-gram モデル

二つの手法の関係

コスト最小法における**コストの和**は品詞 bi-gram モデルにおける**確率の積** P の逆数の対数を取ったものとみなすことができる [22]。

$$\begin{aligned}\log\frac{1}{P} &= -\log\Big(\prod_{i=1}^{n} p(w_i \mid t_i)p(t_i \mid t_{i-1})\Big) \\ &= \sum_{i=1}^{n}\Big(-\log p(w_i \mid t_i)\Big) \\ &\quad + \sum_{i=1}^{n}\Big(-\log p(t_i \mid t_{i-1})\Big)\end{aligned}$$

$-\log p(w_i \mid t_i)$ がコスト最小法における形態素コスト、$-\log p(t_i \mid t_{i-1})$ が連接コストに対応する。

これにより人手によるコスト体系と品詞タグ付きコーパスからの学習による確率パラメタを同じ土俵で扱うことが可能であることが分かる。つまり、確率パラメタをコストに変換するか、コストを確率パラメタに変換すれば良い。

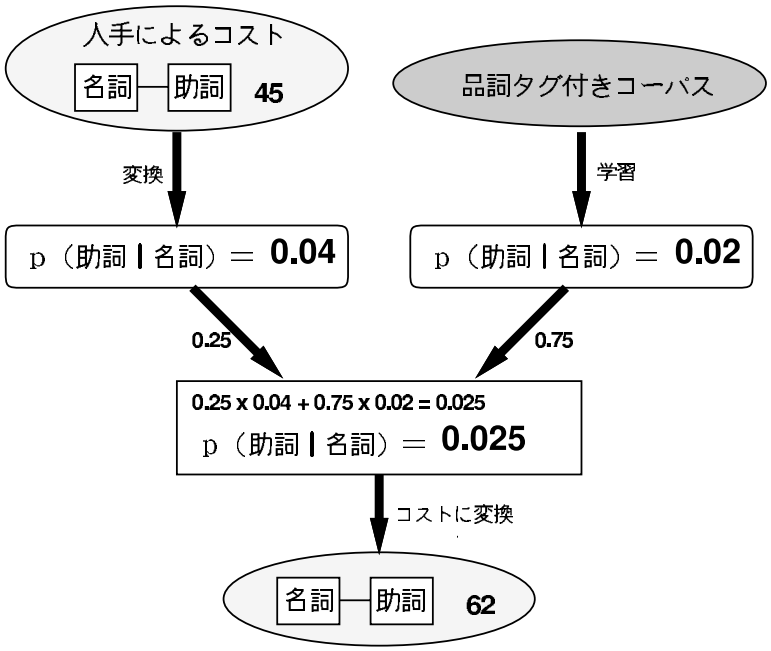


図 3.15 統合処理の流れ

3.2.3 コストと確率パラメタの統合

人手によるコスト体系とコーパスから学習された確率パラメタをある比率で混ぜ合わせ新たな値をつくり出し、それを最適解の選択に用いる。

どのような方法で統合すれば最適なのかは不明であるが、ここでは確率値の線形和という一番単純な方法を採用する (3.2.3 節)。そのために両者を確率として扱うことにする。それには人手によるコストを確率パラメタに変換することを考えなければならない (3.2.3 節)。

処理の流れを図 3.15 に示す。まず人手によるコストを確率パラメタへ変換する。次に二つの確率パラメタをある比率で統合する。そして最後にその新たな値をコストに変換する。

コストから確率パラメタへの変換

3.2.2 節の議論により、確率パラメタの逆数の対数を取ったものがコスト最小法におけるコストに対応する。そこで、指数関数を用いてコストを確率パラメタに変換すること

を考える[2] 。

コストは人間による恣意的な値であり、修正しやすいように比較的大きい値 (100 など) が用いられていることが多い。このような値を直接確率値に変換しても、極端に小さくなってしまい、確率値として直感に合わない。

そこで、まずコストのスケール変換を行う。最大のコストが最小の確率になるように係数を求めて、それを用いてスケール変換を行う。この係数を**確率化係数** α_p と呼ぶことにする。確率化係数の最適値については 3.2.4 節で述べる。

変換は式 (3.1) を用いて行う。人手によるコスト C_h を確率化係数 α_P を用いてスケール変換を行ったコストを指数関数で確率パラメタ P_h に変換する。

$$P_h \quad = \quad \exp(-\alpha_p \times C_h) \tag{3.1}$$

二つの確率パラメタの統合

人手によるコストを変換した確率パラメタ P_h と、コーパスからの学習による確率パラメタ P_c を以下の式を用いて足し合わせ、新たなパラメタ P_{new} を作り出す。λ は混ぜ合わせの比率で**統合比率**と呼ぶ。

$$P_{new} = \lambda P_h + (1 - \lambda) P_c \tag{3.2}$$

図 3.15 では、人手によるコスト 45 (名詞と助詞の連接コスト) を前節の方法で確率値 $0.04(= P_h)$ に変換し、それを品詞タグ付きコーパスから得た確率パラメタ $0.02(= P_c)$ と、1:3 の比率 ($\lambda = 0.25$) で足し合わせている。P_{new} は、0.025 となる。

統合比率の最適値については 3.2.4 節で述べる。

コストへの変換

実装にあたっては、コスト最小法に基づく形態素解析システム 茶筌 [24] を使用しているため、確率パラメタをコストに変換する必要がある。確率パラメタの逆数の対数を取り、形態素解析システムに合わせてスケール変換を行う。

茶筌 ではコストの範囲は、1〜255 であるので、図 3.15 では、それに合わせて $-\log P_{new}$ をスケール変換している。

[2] コストから変換されたものは、厳密には確率値とは言えないが、ここでは、確率値として扱うことにする。

3.2.4 実験

形態素解析システム 茶筌 により実験を行なう。

統合に用いる人手によるコストは、茶筌 Version 1.0 に付随の定義ファイルに記されているものを利用する[3] 。

実験に用いた品詞タグ付きコーパスは、日経新聞 CD-ROM 94 年版の記事 1000 文 (約 30000 形態素) と ATR 経路探索課題コーパス 30 対話 (約 87000 形態素) で、共に当研究室において人手により品詞タグが付与された。これらは「書き言葉」「話し言葉」という異なる分野を代表するデータとみなせる。

日経新聞では 10 fold、対話コーパスでは 30 fold の cross validation を行った。

また、統合手法との比較のために、人手によるコストのみでの解析、品詞タグ付きコーパスからの学習結果のみを用いた解析も行った。

確率化係数及び統合比率は、経験上の最適値を用いた。学習に用いるデータの量によるが、確率化係数は 3〜11 、統合比率は 0.1〜0.001 の範囲内で良い精度が得られる。

図 3.16 と図 3.17 に実験結果を示す。図中の “rule” は人手によるコストのみでの解析、“probablistic” は品詞タグ付きコーパスからの学習結果のみでの解析、“integration” は本節で提案した統合手法による解析を表す。横軸は、日経新聞では学習に用いた文の数 (一文約 30 形態素) を、ATR 経路探索課題コーパスでは学習に用いた対話数 (一対話約 3000 形態素) を表す。縦軸は、解析精度を表す。解析結果が正しいか否かの判断には「形態素の持つ全ての情報が等しい」という基準を採用した。これは、単語分割、読み、品詞情報の全てが一致していなければ、誤りとみなすという、大変厳しい基準である。

この実験により、統合手法は、人手作成コストによる手法や、同量の品詞タグ付きコーパスから学習された確率パラメタのみを利用する手法といった既存の手法と比べ、学習に用いる品詞タグ付きコーパスが小規模の場合において、解析精度が優位であることが示された。小規模な品詞タグ付きコーパスであっても、形態素解析の精度向上に有効に利用できることにより、3.2.5 節で述べるようなブートストラップを効率良く行うことができる。

3.2.5 おわりに

適用分野を限定し形態素解析を行う場合、その分野に大量の品詞タグ付きコーパスが存在すれば、統計的学習により確率パラメタを獲得し、高い解析精度を得ることができ

[3] 連接コストは連接規則ファイルに記されているもの、形態素コストは形態素辞書に記されている形態素ごとの重みとリソースファイル (.chasenrc) に記されている品詞ごとの重みを掛け合わせたものを用いる。実装上の問題点については、文献 [40] を参照のこと。

る[4] 。

しかし未開拓な分野ではそのような大量のコーパスの入手は不可能であり、自らその分野の品詞タグ付きコーパスを作成しなければならない。形態素解析の精度が低いと品詞タグ付け作業は非常に困難になる。

ここで提案した手法を用いれば、そのような未開拓な分野においても小規模の品詞タグ付きコーパスを作成しさえすれば既存の手法に比べ高精度の解析が可能である。これにより、その分野の品詞タグ付きコーパスの作成が容易になる。つまり、ある程度の量の品詞タグ付きコーパスが整備される度に、そのコーパスを対象として統合手法を用い新たなコストを作成し、それを用いて形態素解析精度を徐々に向上させていくことができ、作業効率の向上につながる。

[4] 分野が異なると解析精度が低くなることは、新聞・対話の二つのデータで確認した。分野依存に関する最近の研究として文献 [41] を挙げておく。

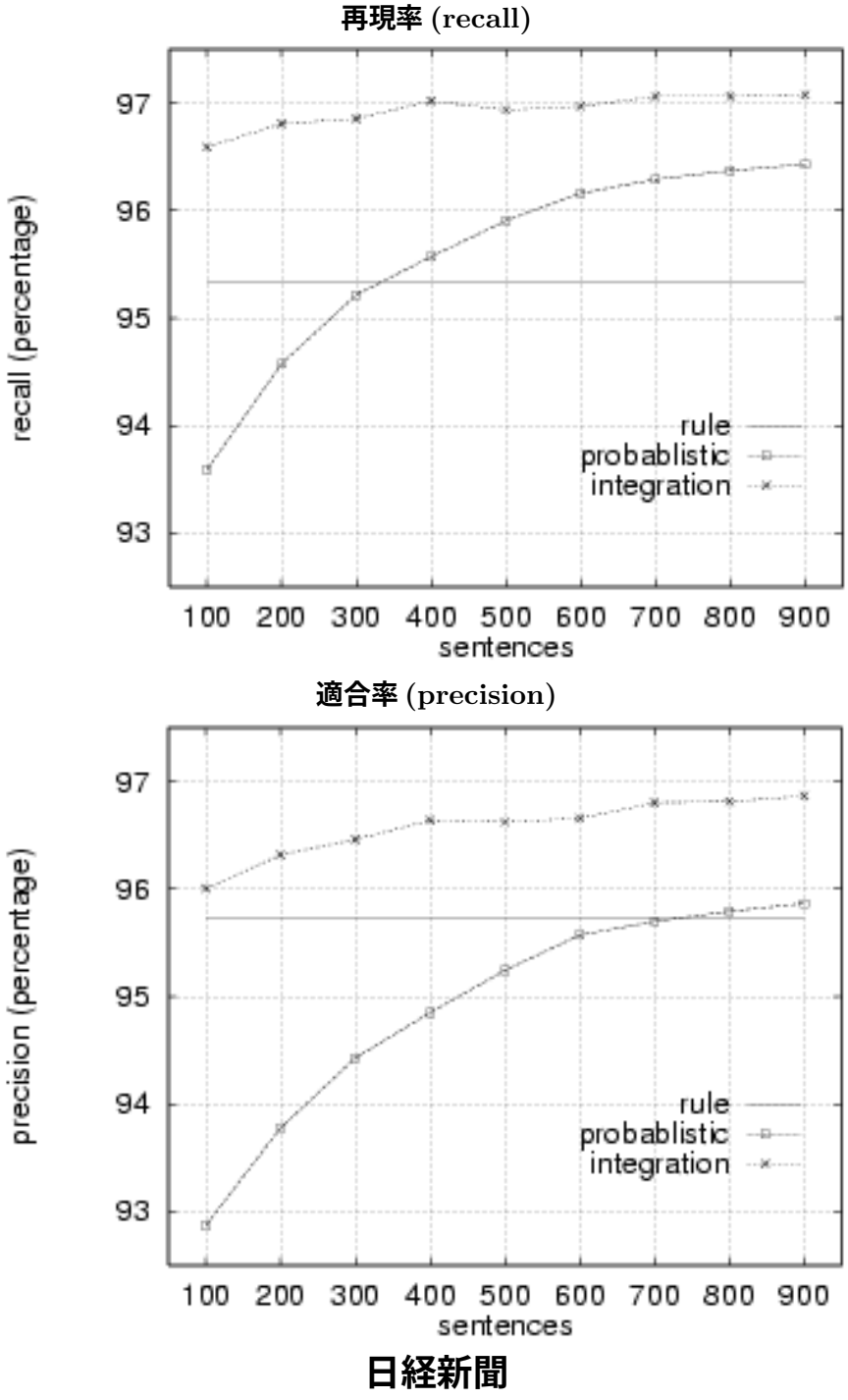

再現率 (recall)
recall (percentage)
97
96
95
94
93
100 200 300 400 500 600 700 800 900
sentences
rule
probablistic
integration
適合率 (precision)
precision (percentage)
97
96
95
94
93
100 200 300 400 500 600 700 800 900
sentences
rule
probablistic
integration
日経新聞

図 3.16 実験結果 1

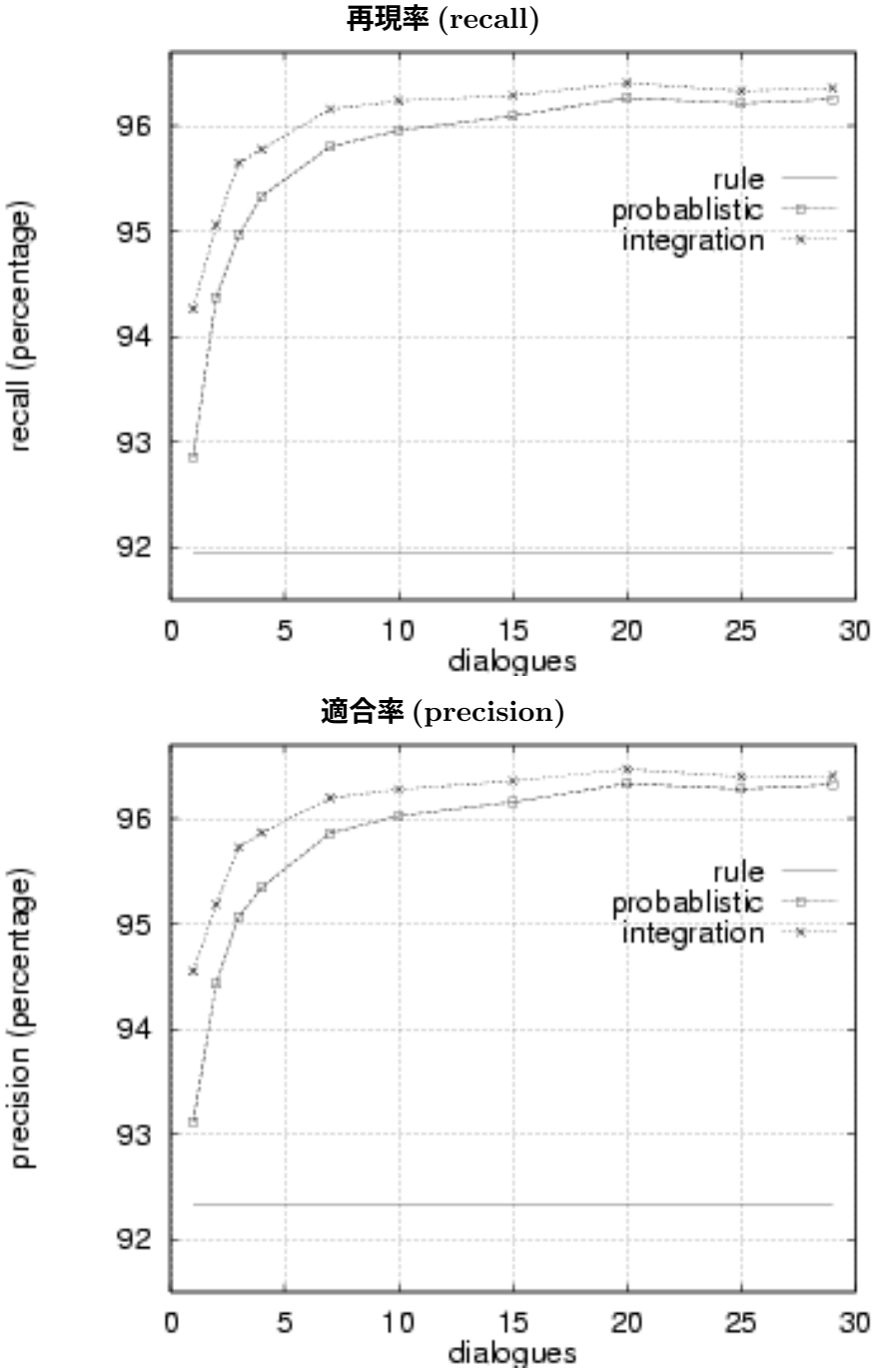

再現率 (recall)
recall (percentage)
96
95
94
93
92
0
5
10
15
20
25
30
dialogues
rule
probablistic
integration
適合率 (precision)
precision (percentage)
96
95
94
93
92
0
5
10
15
20
25
30
dialogues
rule
probablistic
integration
ATR 経路探索課題コーパス

図 3.17 実験結果 2

3.3. 品詞タグ付きコーパスを直接利用した形態素解析

3.3.1 はじめに

茶筌 [24] をはじめとする最近の形態素解析システムは、コーパスから統計的に語の出現確率や連接確率を推定しそれを用いるという手法が一般的である。しかし、このような確率によるデータの抽象化手法には、例外的な現象に対応できないなどの弊害がある。これに対処するために、近年、今まで切捨ててきた言語現象を連語登録 [42] や可変長規則 [43] といった実際のデータに近い形で補助的に利用する研究が行なわれている。

一方、実際のデータを補助的に用いるのであれば、始めからデータを加工せずにそのまま使用して解析を行なうという方法が考えられる。この方法は、記憶装置、演算装置、高速文字列検索 [20] などの技術の進歩を考えれば、現在の段階で十分実用可能であると言える。本節では、この考えに基づき、品詞タグ付きコーパスをそのままデータとして用いる形態素解析手法を提案する。

3.3.2 処理の概要

ここで提案する手法では、まず、解析したい文の部分文字列をキーとして品詞タグ付きコーパスを検索し、見つかったコーパス片[5] に付与されているタグをそのまま利用することで、形態素解析を行なう。簡単な例を用いて処理の概念を説明する。以下のような 3 文からなる品詞タグ付きコーパスがあるとする。

(1) 昨日 [名詞] 、[記号] 太郎 [名詞] が [助詞] 走った [動詞] 。[記号]
(2) 花子 [名詞] は [助詞] ラーメン [名詞] を [助詞] 食べた [動詞] 。[記号]
(3) 彼 [名詞] は [助詞] 走った [動詞] 。[記号]

解析したい文「昨日、花子は走った。」に対して、「昨日、」という部分は (1) の「昨日 [名詞] 、[記号]」、「花子は」という部分は (2) の「花子 [名詞] は [助詞]」、「は走った。」という部分は (3) の「は [助詞] 走った [動詞] 。[記号]」を利用してタグ付けを行なえば、図 3.18 のような形態素解析結果を得ることができる。

実際には、コーパス片の重なり部分に形態素の曖昧性が生じることがあるので、曖昧性解消の基準を設けておく必要がある。これについては後で詳しく述べる。ここで、この手法の利点をいくつかあげてみる。

- 頻度の低い言語現象も扱うことができる。

[5] コーパス中の部分文字列をこう呼ぶことにする。

<table>
<tr><td>文</td><td colspan="2">昨</td><td>日</td><td>、</td><td colspan="2">花</td><td>子</td><td>は</td><td>走</td><td>っ</td><td>た</td><td>。</td></tr>
<tr><td rowspan="3">コーパス片</td><td colspan="3">名詞</td><td>記号</td><td colspan="2"></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td colspan="3">名詞</td><td>助詞</td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td>助詞</td><td colspan="4">動詞</td><td>記号</td></tr>
<tr><td>結果</td><td colspan="3">名詞</td><td>記号</td><td colspan="2">名詞</td><td>助詞</td><td colspan="4">動詞</td><td>記号</td></tr>
</table>

図 3.18 形態素解析例

- 解析失敗の原因 (解析の基になるコーパスのタグ付け間違い) がすぐに分かるので品詞タグ付きコーパスの保守作業に有用。
- 解析結果をコーパスに追加すればすぐに反映される。
- 同じ文がコーパスにあれば解析精度はほぼ 100%になる。
- 頑健な解析が可能。スペルチェッカーなどに応用可能。

また、自然言語処理を応用した高度な検索のために、今後、意味や構造などのタグのついた日本語テキストが蓄積されていくと期待できる [44]。この手法では、そのようなデータを検索だけでなく解析にもそのまま利用できる。例えば、タグ付きデータを大量に保有しているサイトでは、そのデータに対しての検索質問文などの解析を効率的に行なうことができる。

このようなタグ付きデータをそのまま利用した方法として ATR の ν (NUU)-Talk 音声合成システム [45] が知られている。ν-Talk では、入力音声記号列に対して、蓄えられたタグ付き音声データからできるだけ使用環境に近いスペクトルを選ぶという方針により音声合成を行なっている。

3.3.3 システムの詳細

本節では、システム構築にあたって以下の目標を設けた。

- 言語知識を用いない：特定の言語に依存せず、あらゆるタグ付きデータに利用できる汎用システムを目指す。

位置	文	タグ	意味
0	花	(M	「名詞」を構成する最初の文字
2	子	)M	「名詞」を構成する最後の文字
4	は	|J	「助詞」を構成する唯一の文字
6	走	(V	「動詞」を構成する最初の文字
8	っ	-V	「動詞」を構成する途中の文字
10	た	)V	「動詞」を構成する最後の文字
11	。	|T	「記号」を構成する唯一の文字

図 3.19 タグの意味

- 統計情報を用いない：統計情報などの外部データがなくとも動く身軽なシステムを目指す。

とりあえず、汎用システムの構築を最優先とし、形態素解析精度を上げるためのテクニック (言語知識・統計情報の付加など) は今回は用いない。システム構築には、プログラング言語 Perl を主に用い、検索部分は C を使用した。

高速全文検索のための品詞タグ付きコーパスのフォーマット

本手法では、まず、品詞タグ付きコーパスを検索してマッチした部分文字列に付与されている品詞列を得る処理を行なう。これを高速で行なうために、各文字に対して品詞を表すコードを割り当てる方法を用いた。

一般に品詞タグ付きコーパスは 3.3.2 節の例で示したように形態素ごとに分割されているフォーマットが多い。このようなフォーマットでは形態素をまたぐ文字列の検索には複雑な文字列処理が必要となる。そこで、以下のように、品詞タグ付きコーパスを文とタグに分けて、文字位置で対応が取れるようなフォーマットを利用することにした。

(1) 花子は走った。 (M)M|J(V-V)V|T
(2) 彼は食べた。 M|J(V-V)V|T

各文字に対してその文字が含まれる形態素の品詞とその文字の形態素内での位置を表すタグが付与される。図 3.19 に「花子は走った。」という文とそれに付与されているタグの意味を示す。

このフォーマットにより部分文字列を高速に検索することができ、その文字列に対応する品詞タグも素早く容易に取り出すことができる。例えば、図 3.19 の例 (「花子は走った。

(M)M|J(V-V)V|T」) を品詞タグ付きコーパスとみなし、「は走った」という文字列にどのような品詞が振られている例があるのか検索することを考える。「は走った」は「花子は走った。」という文字列の 4 文字目から 12 文字目なので、品詞タグ列「(M)M|J(V-V)V|T」の 4 文字目から 12 文字目の文字列「|J(V-V)V」が、対応する。よって、「は」は助詞、「走った」は動詞ということが容易に分かる。

解析対象文の部分文字列の検索

ここでは実際に品詞タグ付きコーパスを検索する方法について説明する。検索部には Suffix Array を用いて高速全文検索を行なうライブラリ SUFARY([20], 4.1.2 節) を使用している。検索は文頭から文末に向かって行ない、各位置から品詞タグ付きコーパスとの common prfix (4.3.3 節を参照) を探し、それを検索結果として後の処理に利用する。ただし、他の common prefix に含まれるような文字列は削除される。図 3.20 に検索の例を示す (×は削除された検索結果を表す)。

今回は、検索結果に部分的にしか一致しない形態素が混じっていた場合は無視することにする。例えば、検索結果「は走っ」では「走っ」が「走った [動詞]」の一部でしかなく、半端な検索結果として削除される。

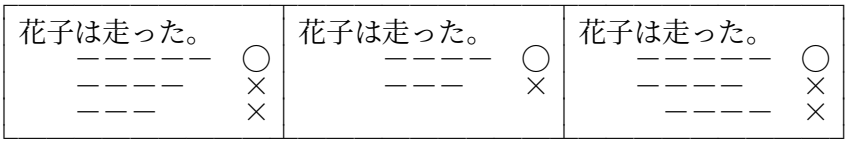


図 3.20 検索例

最適解の選択

前節の処理により選ばれた検索結果をコーパス片と呼ぶ。これに含まれる形態素列を基に形態素解析結果としての解を選択する処理について説明する。選択のための優先条件として以下のものを考慮した。

1. 他の形態素と連接しない形態素は優先度が低い。
2. コーパス片が重なっている部分の共通の形態素は優先度が高い。
3. 長いコーパス片に属する形態素は優先度が高い。

4. 長い形態素の優先度は高い。

これらをコストなどの体系にまとめるのは困難なので、優先条件の適用の順序を決めてそれに従って選択処理を行なうことにした。

他の形態素と連接しない形態素の削除 直前・直後に接続できる形態素がなく、かつ、他の対抗する形態素よりも文字列長が短い形態素を浮き形態素と呼ぶことにする。最適解に浮き形態素が選択されると未定義語を出力する必要がでてきてしまうので、検索結果のコーパス片に含まれている浮き形態素を全て削除することにする。例えば、図 3.21 では「です [助動]」が浮き形態素である。これを最適解に選択すると、「む」を未定義語にしなければならない。それよりも「で [助詞] すむ [動詞]」を選択した方が現実的である。結局、浮き形態素は最適解としてふさわしくないと思われるので、早い段階で削除する。

<table>
<tr><td>文</td><td>で</td><td>も</td><td>、</td><td>し</td><td>な</td><td>い</td><td>で</td><td>す</td><td>む</td></tr>
<tr><td rowspan="5">コーパス片</td><td colspan="2">接続</td><td>記号</td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>助詞</td><td>助詞</td><td>記号</td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td>動詞</td><td colspan="2">助動</td><td>助詞</td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td colspan="2">助動</td><td colspan="2">助動</td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td>助動</td><td colspan="2">動詞</td></tr>
</table>

図 3.21 他の形態素と連接しない形態素

曖昧性の解消 曖昧性の解消は、検索結果に含まれる全ての形態素の共通の区切りに着目し、それらで区切られた部分ごとに行なう。上の例では、**でも, 、, し, ない, で, すむ** の 7 つの部分に分割される (「です [助動]」は浮き形態素として削除されてるので考慮しない)。共通の区切りによって分割された部分を共通区切り領域と呼ぶことにする。処理の手順としては、まず検索結果を共通区切り領域に分割し、それから各領域ごとに最適形態素の選択を行なう。共通区切り領域は図 3.22 に示す 3 つのタイプに分類できる。

Type 1 の場合は、曖昧性がないので、そのままそのタグが最適解として決定される。 Type 2, 3 の場合は、図 3.23 の手順に従って曖昧性の解消を行なった。

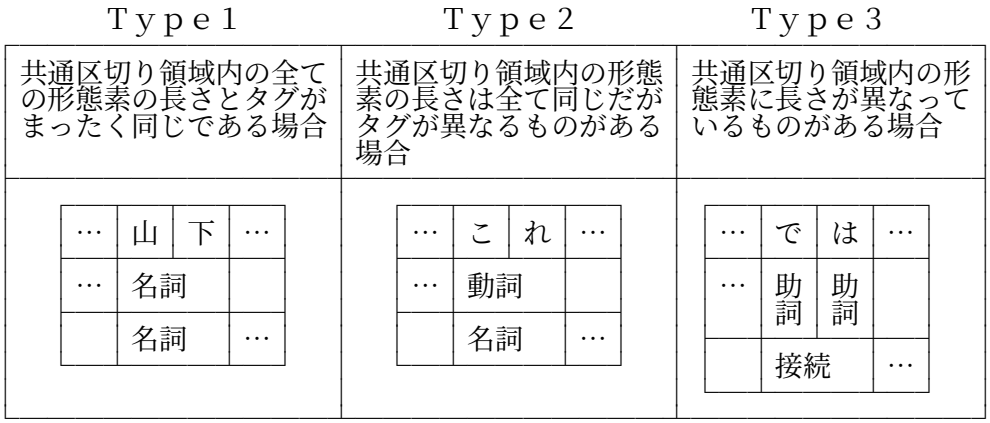


図 3.22 共通区切り領域のタイプ

3.3.4 実験と考察

本手法と統計的手法とを比較するための実験を行なった。品詞タグ付きコーパスをそのまま用いたシステム (BTG) と、そのコーパスから統計的にパラメタ (bigram) を学習しそれを用いた形態素解析システム茶筌 [24] とで、アウトサイドデータを解析し精度を測定した。

実験には毎日新聞 95 年版の品詞タグ (IPA 品詞体系) 付きコーパス [27] を使用した。1 万文 (約 26 万形態素)、2 万文 (約 52 万形態素)、3 万文 (約 78 万形態素) を学習/検索用コーパスとし、これらとは別の 2000 文 (形態素数約 5 万 3 千) を評価用コーパスとした。茶筌の辞書項目は学習コーパスから抽出した形態素のみを使用している。解析結果を評価用コーパスと形態素ごとに比較して、品詞の一番荒い分類 (名詞、動詞、助詞といったレベル) が異なるものを誤りとみなした。評価結果を表に示す。

	茶筌 適合率/再現率	BTG 適合率/再現率
1 万文	87.1% / 92.3%	88.1% / 88.9%
2 万文	88.7% / 92.8%	90.4% / 91.8%
3 万文	90.1% / 93.5%	91.7% / 93.2%

両手法の解析結果を比較すると、誤りを生じる場所に違いが多い。解析精度はほぼ同じであることから、これは、それぞれの手法の特徴が解析に反映されているということを意味する。

本手法の欠点として、活用などの言語情報を利用しないので、活用語の解析精度が落ちるという点があげられる。茶筌ではプログラム内部で活用処理をしているので、この

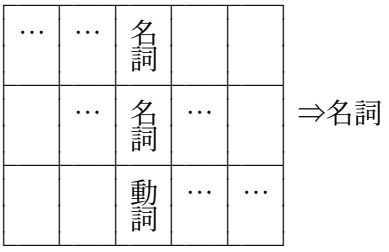
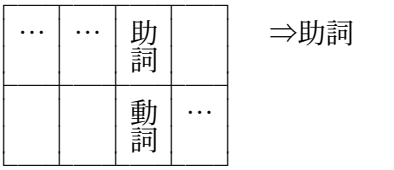
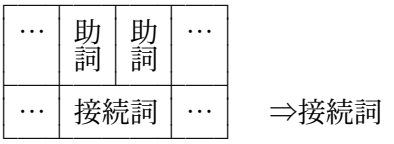
1	共通の形態素があればそれを選択する。	
2	それで決まらなければ、各形態素の属するコーパス片の文字列長を比較し、それが最長のコーパス片に属する形態素を選択する。	
3	それでも決まらなければ、各形態素の長さを比較し、最長の形態素を選択する。	
4	それでも決まらなければ、適当に選ぶ。	

図 3.23 最適解選択手順

部分の精度の差は歴然としている。これは品詞体系との相性 (活用語をどう扱うか) の問題といえる。また、本手法では、検索用コーパスの量が少ないと、短いコーパス片に現れる形態素を採用してしまうことが多い。これにより本手法の特徴である文脈 (前後の文字列) が考慮できず当然精度は落ちる。優先条件が活かされず、曖昧性の解消も不正確になる。これはスパースネスの問題である。

一方、本手法の特徴が活かされている部分も少なくない。例えば、「障害はなくなりつつある。」という文を茶筌で解析した結果は以下のようになる。

障害 [名詞] は [助詞] なく [助動詞] なり [動詞] つつ [助詞] ある [動詞] 。[記号]

評価用コーパスでは「なくなり [動詞]」が正解とされているので、「なく [助動詞] なり [動詞]」の部分は誤りである。この誤りは、助動詞「ない」と動詞「なる」の出現確率 (つまり出現頻度) が動詞「なくなる」に比べ非常に高いことが原因であった。一方、BTG では同じ文を誤らずに解析できる。これは、検索用コーパスに、

今 [名詞] は [助詞] なくなり [動詞] つつ [助詞] ある [動詞] 。[記号]

という文が存在し、「はなくなりつつある。」という部分がそのまま流用できるからである。

これらのことにより、本手法で十分な精度が得られない部分に、統計情報などを用いた従来の形態素解析手法を補助的に用いることにより、より高い解析精度を得ることが可能だと思われる。つまり、実際のデータに答えがあるのならばそのまま利用してしまえばいいのであり、それでうまくいかない部分は従来の手法を利用するということである。

また、現在入手できる日本語品詞タグ付きコーパスには、一貫性の欠如に起因する誤りが多い。例えば、同じ「大阪駅」という文字列に対する品詞タグが「大阪 [名詞] 駅 [名詞]」であったり「大阪駅 [名詞]」であったりする。本手法は、品詞タグ付きコーパスの全文検索という処理を行なうので、この種のタグ付け誤りを効率的に発見することができ、品詞タグ付きコーパス修正支援ツールとしての用途が期待できる。

3.3.5 おわりに

今後の展開として、品詞タグ付きコーパスを直接利用する手法の統計的手法との統合による解析精度の向上、および、品詞タグ付きコーパス修正支援ツールとしての利用を目指し研究を進めたい。また、形態素解析以外の自然言語処理への適用も考えていきたい。

第4章

Suffix Array によるコーパスの類似検索

本章では、タグ付きコーパス作成作業の際に過去のタグ付与基準などを参照することを目的として、Suffix Array と呼ばれる文字列検索のためのデータ構造を用いた二種類の類似検索手法を提案する。一つは動的計画法と組み合わせたもので、検索キーと数文字異なるだけの文字列など、見つけたいものがある程度分かっている場合に向いている。もう一つは、転置インデックスと組み合わせたもので、全体は似ていなくてもどこかの部分が似ているようなものを検索する場合に向いている。

4.1. Suffix Array

本節では、本論文で多用している Suffix Array という検索用データ構造について解説する。

Suffix Array とは高速な文字列検索を可能にするデータ構造である。UNIX の grep コマンドのような「テキストに対するあらゆる部分文字列の検索」を高速[1] に行なうことができる。ただし、あらかじめ検索用インデックス (= Suffix Array) を作成しておく必要がある。

Suffix Array の特徴を簡単にまとめておく。

- どんな部分文字列でも検索可能。日本語テキストへのインデクシングで、形態素解析などの単語分割処理が必要無い。
- しくみが単純なので実装が簡単。
- 検索時に必ず元テキストが必要。WWW サーチエンジンには不向き。サイト内検索ならば問題無い。
- Suffix Array のサイズはインデックスポイントの数、つまりファイルサイズに比例する。

4.1.1 Suffix Array の仕組み

Suffix Array のしくみについて説明する前に、まず suffix について説明する。suffix とは検索対象となるテキスト中のある位置から始まりテキスト末尾までの範囲の文字列である。どの suffix も開始位置が特定されれば一意に決まる。この開始位置をインデックスポイント (index point) と呼ぶ [46]。例えば、テキスト「さくさくさくら」に対し、suffix「さくら」 は元テキストの 5 文字目から始まるのでインデックスポイントは 5 となる (図 4.1)。

このインデックスポイントの配列を、それぞれに対応する suffix の辞書順に従ってソートしたものが Suffix Array である。例えば、図 4.1 のインデックスポイントの配列「1 2 3 4 5 6 7」を対応する suffix でソートすると、「2 4 6 1 3 5 7」となる (図 4.2)。この配列が Suffix Array である。文献 [47] に非常にすっきりした Suffix Array の作成プログラム (sufsort.c) が掲載されているので図 4.3 にあげておく。

[1] 二分探索を用いるので、計算量は $O(\log(n))$: n = テキストサイズ。grep のようなテキスト全体を走査する方法では $O(n)$ かかる。巨大なファイルの検索において、その検索速度の差が顕著になる。

Text	さ	く	さ	く	さ	く	ら
Index point	1	2	3	4	5	6	7

図 4.1 インデックスポイント

文字列の検索には二分探索 (binary search) を用いる。図 4.4 に検索キー「くさくさ」で検索する例をあげる。二分探索は中心の要素と検索キーを比較して検索範囲を狭めていく探索手法である。まず、Suffix Array の中心 (4 番目) であるインデックスポイント 1 に対応する suffix「さくさくさくら」と検索キー「くさくさ」を比較する (phase 1)。辞書順で考えると「くさくさ」が小さいので、Suffix Array の中心より前半分 (1 番目から 3 番目) に検索範囲が絞られる。次に、その前半分の中心であるインデックスポイント 4 の suffix「くさくら」と検索キーを比較し (phase 2)、また検索範囲を絞る。このようにして、インデックスポイント 2 が最終的な検索結果となる (phase 3)。

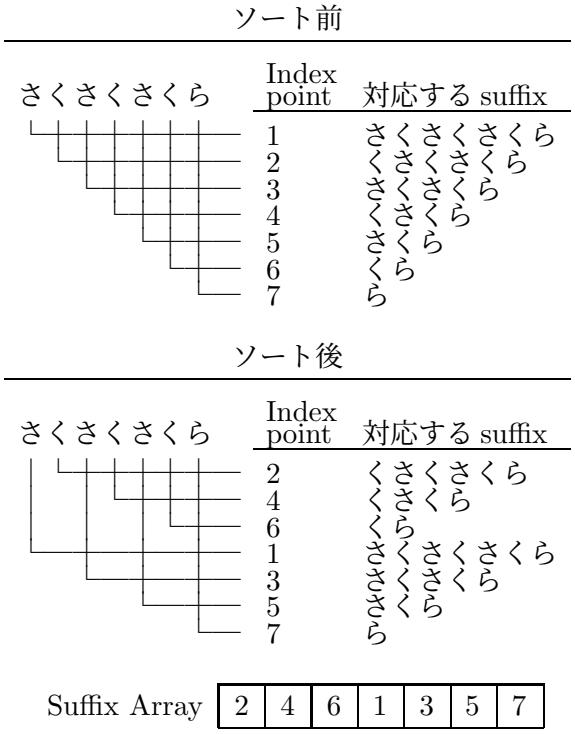


図 4.2 Suffix Array の作り方

```
#include <fcntl.h>
#include <malloc.h>
#include <stdio.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <sys/types.h>
/* usage: sufsort1 text > text.suf */

char *text;

suffix_compare(int *a, int *b)
{
  return strcmp(text + *a, text + *b);
}

main(int ac, char **av)
{
  struct stat stat_buf;
  int N, i, *suf;
  FILE *fd = fopen(av[1], "r");
  fstat(fileno(fd), &stat_buf);
  N = stat_buf.st_size;
  text = (char *)malloc(N+1);
  fread(text, sizeof(char), N, fd);
  text[N] = 0; /* pad with null */

  suf = (int *)malloc(N * sizeof(int));
  for(i=0;i<N;i++) suf[i] = i;

  qsort(suf, N, sizeof(int), suffix_compare);
  fwrite(suf, N, sizeof(int), stdout);
}
```

図 4.3 Suffix Array 作成プログラム

このように、Suffix Array は非常に単純な方法ということもあり、1970 年代にはすでに使われている [48]。1990 年になり Manber [18] により Suffix Array と命名された。Suffix Array についての教科書的な文献として [46] をあげておく。巨大な Suffix Array の作成方法、圧縮、高速化、正規表現、転置インデックスとの比較など、関連する話題が言及されている。また、日本語による簡単な解説として [49] がある。

Phase	Suffix array	対応する suffix
3 →	2	**くさくさ**くら
2 ⟶	4	くさくら
	6	くら
1 ⟶⟶	1	さくさくさくら
	3	さくさくら
	5	さくら
	7	ら

図 4.4 Suffix Array による文字列検索

4.1.2 Suffix Array による文字列検索ライブラリ SUFARY

Suffix Array を研究用に容易に利用できるように SUFARY を開発した。SUFARY は Suffix Array の作成・それを用いた検索を行なうライブラリで、フリーソフトである [20]。本論文では、2.3 節、3.3 節、4.2 節、4.3 節で使用している。

4.1.3 Suffix Array による疑似トライ

Suffix Array による疑似トライの実現方法について説明する。トライ構造の欠点であるデータ領域の非効率性を解消するための方法として有効である。従来のトライの実装に比べ必要なデータ領域は数分の 1 に圧縮できる。

まず、例として、テキスト BABAC に対する Suffix Array (図 4.5) を用いる。説明のために Suffix Array の各インデックスポイントの指す suffix を縦書きに表示した。ここで、ABAC, AC の先頭の A、BABAC, BAC の先頭の B と 2 文字目の A をそれぞれまとめて一つのノードとみなせば図 4.6 と同等なトライ (Suffix Tree) となることに気付くであろう。

つまり、先頭から X 文字目までが共通な suffix を指すインデックスポイントの Suffix Array での範囲をトライ上の深さ X のノードとして扱うことで疑似的なトライ構造が実現できる。この疑似トライにおいて、検索文字列に従ってノードを辿るという操作は、検索文字列の先頭文字から始めて一文字ずつ増やしながら (つまり、徐々に検索範囲を狭めながら) Suffix Array を検索するという操作になる。検索の度に徐々に検索範囲が狭められていくので検索効率は良い。

図 4.7 に疑似トライのアルゴリズムを示す。

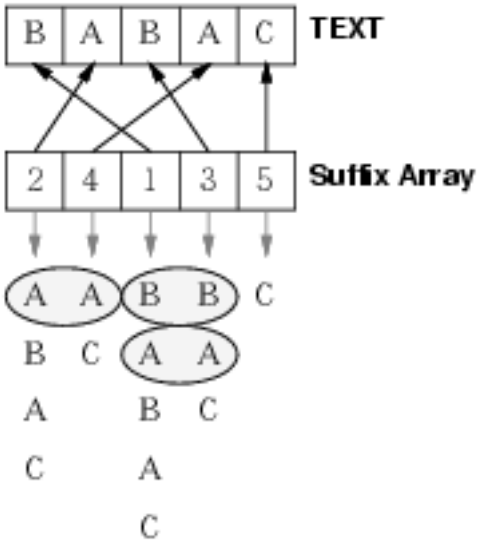
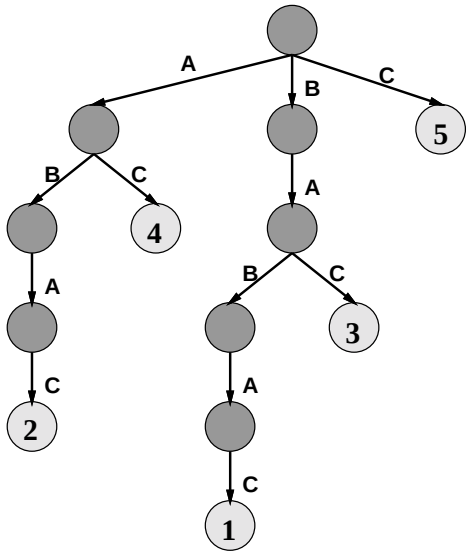


図 4.5 Suffix Array による疑似トライ



テキスト	B	A	B	A	C
インデックスポイント	1	2	3	4	5

図 4.6 Suffix Tree

- $t_1, ..., t_M$: 検索キーを構成する文字列
- M: 検索キーを構成する文字数
- $s_1, ..., s_N$: Suffix Array
- N: Suffix Array のサイズ

1. $l = 1, r = N$
2. for $(i = 1; i <= M; i++)$
   - Suffix Array の一部 (最初は Suffix Array 全体) $s_l, ..., s_r$ から、文字列 $t_1, ..., t_i$ を二分探索する。
   - 検索成功ならば、
     - 結果を l', r' に格納。($s_{l'}, ..., s_{r'}$ のそれぞれのインデックスポイントが指す文字列の先頭 i 文字は $t_1, ..., t_i$ 一致する。)
     - { トライの各ノードで行なう処理をここで行なう }
     - $l = l', r = r'$
   - 検索失敗なら i のループを抜ける

図 4.7 Suffix Array による疑似トライのアルゴリズム

4.2. 動的計画法と Suffix Array による類似検索

4.2.1 はじめに

曖昧性を許す検索、つまり、入力データ (検索キーワード) と検索対象データの間の類似性を用いた検索は様々な分野において有用な技術である。例えば、用例に基づく機械翻訳 [50] [22] や翻訳支援 [51] [52] における類似用例検索、校正システムや OCR 読み取りにおけるスペル誤りの訂正 [53]、新聞などの文書データに対する類似検索を用いた情報検索 [54] など、自然言語処理の分野において用途は広い。また、ゲノムサイエンスの分野においても、塩基・アミノ酸配列からの特定の部位の検索[2] やアライメントなどで用いられている [55][56]。

本節では、このような類似検索のうち、フルテキスト類似用例検索の高速化の手法について論じる。4.2.2 節では、ここで扱うフルテキスト類似検索のモデルについて説明する。フルテキスト類似検索と辞書類似検索の違いについても明確にする。4.2.3 節では、これまでに研究されてきた類似検索の手法について説明する。一つは DP マッチングによる方法で、スペルチェッカーや塩基配列・アミノ酸配列の類似検索によく用いられている。もう一つはトライによる方法で、DP マッチングに比べ計算量において優れている。4.2.4 節では、トライによる方法の問題点を克服し、Suffix Array を用いたフルテキスト類似検索を実現する方法を提案する。

4.2.2 フルテキスト類似検索のモデル

本節で対象とするフルテキスト類似検索のモデルを以下のように定義する。

- 検索キーワードに対し置換/挿入/削除の 3 つの基本操作 (図 4.8) が用意されている。各基本操作にはあらかじめペナルティが設定できる (図 4.9)。
- 3 つの基本操作を用いて、ペナルティの合計が最小になるように、検索キーワードと検索対象テキストの部分文字列を並べる。
- ペナルティの合計がある値 (限度) 以下のものだけを検索結果として出力する。

例えば、図 4.9 では、ギャップ (挿入/削除) ペナルティは 2、B → C または C → B の置換ペナルティは 2、同じ文字同士の置換ペナルティは 0、他の置換ペナルティは 1 となっている。すると、ペナルティの合計は、図 4.8 の置換の例では 1、挿入/削除の例で

[2] ホモロジー検索と呼ばれている。

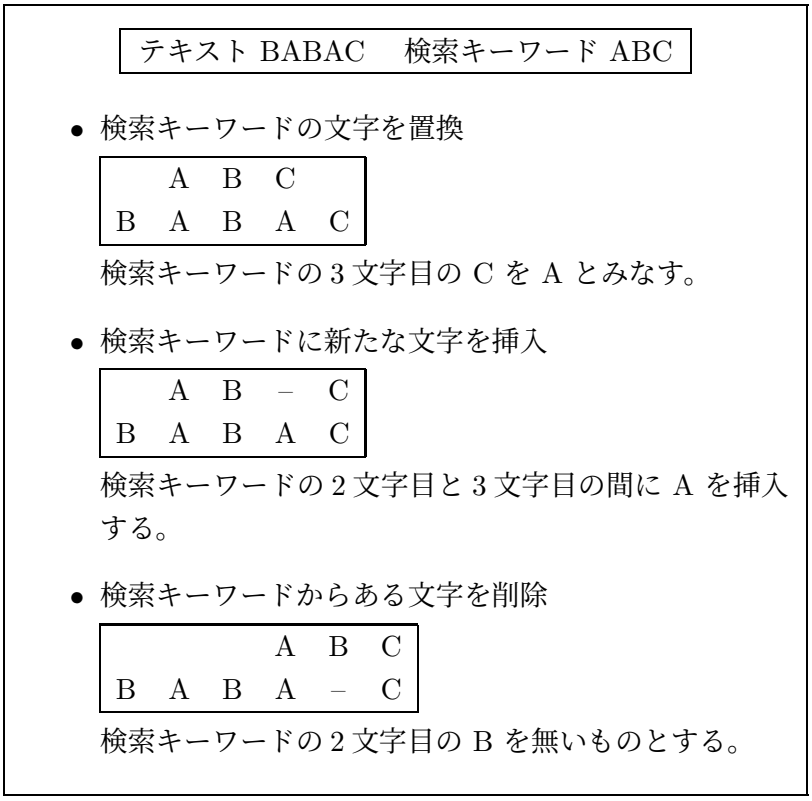


図 4.8 基本操作

はそれぞれ 2 となる。もしペナルティの合計の限度が 1 とするならば、置換の例 (ABA) だけが検索キーワード ABC の類似検索結果として出力される。

ここでフルテキスト類似検索と辞書類似検索の違いを明確にしておく。

フルテキスト類似検索 検索対象テキストの中から検索キーワードと類似した部分文字列を探す。

辞書類似検索 すべての検索対象の文字列 (見出し語) と検索キーワードを両方が同じ長さになるように並べてみて、ある類似性を充たす文字列を探す。

例えば図 4.8 の例では、両方の文字列の長さをそろえるために下図のように検索キーワードの両側にギャップを入れる必要がある。図 4.9 に従うとペナルティ合計は 4 になる。

–	A	B	C	–
B	A	B	A	C

検索対象のデータ量が同じ場合、当然フルテキスト検索の方が検索量が多くなる。用例ベース機械翻訳などの従来の類似用例の検索は辞書類似検索である [50]。塩基・アミ

ギャップペナルティ	2			
置換ペナルティ	A	B	C	
	0	1	1	A
		0	2	B
			0	C

図 4.9 ペナルティの例

ノ酸配列のホモロジー検索はフルテキスト類似検索である

4.2.3 類似検索の手法

これまで研究されてきた類似検索の手法について説明する。

DP マッチング

n を検索対象テキストの長さ、m を検索キーワードの長さとする。テキスト $t_1, t_2, \cdots, t_n$ から検索キーワード $a_1, a_2, \cdots, a_m$ を類似検索することを考える。これまでに $a_1, a_2, \cdots, a_{i-1} (i \leq m)$ と $t_1, t_2, \cdots, t_{j-1} (j \leq n)$ の比較が終わっていたとして、そこまでのペナルティの合計が $P_{i-1,j-1}$ で表されているとする。この先比較を続ける際、可能性として、一致、不一致を問わず a_i と t_j を並べるか (置換)、挿入/削除を行うかしかない。従って、ペナルティの合計 $P_{i,j}$ を求めるには、この中でペナルティ最小の操作を選べば良い。これを式で表すと以下のようになる。s は置換ペナルティ、g はギャップ (挿入/削除) ペナルティを表す。

$$
\begin{aligned}
P_{i,j} = \quad & \min(P_{i-1,j-1} + s(a_i, t_j), && \#\text{置換} \\
& P_{i-1,j} + g, && \#\text{削除} \\
& P_{i,j-1} + g) && \#\text{挿入}
\end{aligned}
$$

例として、図 4.10 に、図 4.9 のペナルティ定義を用いてテキスト BABAC を検索キーワード ABC で検索する例を示す。格子上の数字は $P_{i,j}$ を表す。斜め線上の数字は置換ペナルティ、格子上以外の縦・横線上の数字は挿入/削除ペナルティを表している。矢印

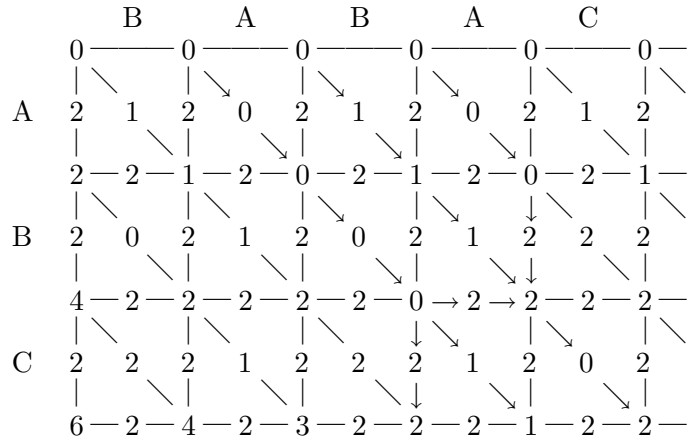


図 4.10 DP マッチング

線はペナルティの合計が 2 以下の結果 (AB, ABA, ABAC, BAC, AC) を表している。

しかし、DP マッチングは検索対象テキストを頭から走査していくので、計算量は $O(mn)$ となり、検索対象が大きい場合、時間がかかりすぎる。そのため、計算量を低減するための手法が盛んに研究されている [3][57]。また、実用面を重視して、事前に検索対象を絞り込み DP マッチングの計算回数を低減するという手法も研究もされている。畑中ら [53] は文字列に対応付けられた整数値により類似度 (ハッシュ距離) 計算を行い検索対象を絞り込んでいる。ゲノムサイエンスの分野で、塩基配列やアミノ酸配列を類似検索するプログラムとしてよく知られている FASTA では、固定長文字列での転置インデックスを用いて検索対象を絞り込んでいる [55]。

トライを用いる方法

Oflazer[58] はコード化した構文木を検索するためにトライ (TRIE) を用いた類似検索を行っている。これは Error-tolerant Recognition アルゴリズム [59] をトライに適用したものである。深さ優先でトライを探索し、ノードを辿る度に現在位置までの文字 (コード) 列と検索キーワードとの距離 (cut-off distance) を DP マッチングで計算し、それがある限度を越えたらその方向の探索を打ち切るという方法である。

図 4.11 に Error-tolerant Recognition アルゴリズムを示す[3] 。このアルゴリズムは 4.2.2 節で説明した辞書類似検索を扱うものである。m は検索キーワードの長さ、t はペナルティの合計の限度、q_i はトライ上のノード、q_0 はトライのルートノードを表す。*cutdist*

[3] 説明のために文献 [58] のものに少し変更を加えてある。

1. $push((空文字列, q_0))$
2. スタックが空になるまで以下の処理を繰り返す。
   (a) $pop((Y', qi))$
   (b) q_i から辿れる全てのノードと、そのときに通る枝のラベル V に対して以下の処理を行う。
      i. $Y = concat(Y', V)$　# 文字をつなげる
      ii. $k = length(Y)$　# 文字列の長さ
      iii. $cutdist(m, k) \leq t$ ならば $push(Y, q_j)$
      # 限度を越えていたら切り捨てる。
      iv. $P_{m,k} \leq t$ かつ q_j が**終端ノード**ならば検索結果として Y を出力する。

図 4.11 Error-tolerant Recognition アルゴリズム

は cut-off distance で、以下のように定義される[4] 。

$$\begin{aligned} cutdist(m,k) &= \min_{l \leq i \leq u} P_{i,k} \\ l &= \max(1, k - \lfloor t/M \rfloor) \\ u &= \min(m, k + \lceil t/M \rceil) \end{aligned}$$

M はギャップのペナルティを表す。

ここで、Error-tolerant Recognition アルゴリズムを用いた簡単な例を図 4.12 に示す。左側は AAA, AB, BBA という 3 つの文字列から構成されるトライで、これから文字列 AAB をペナルティの合計が 1 以下という条件 ($t = 1$) で類似検索する過程を右側に示した。ギャップペナルティと置換ペナルティ (A → B,B → A) はどちらも 1 とする。灰色のボックスは cut-off distance を表す。BBA の方は、[B]→[B] とトライを辿るとこの時点で cut-off distance が 1 を越えてしまうので探索は打ち切られる。AAB,AB は探索は打ち切られることなく、葉までたどり着き、$P_{m,k}$(右下隅の数字) が $t(= 1)$ 以下なので検索結果として出力される。

[4] cut-off distance の l と u の定義は文献 [58] と文献 [59] とで異なっている。ここでは後者の定義を採用した。

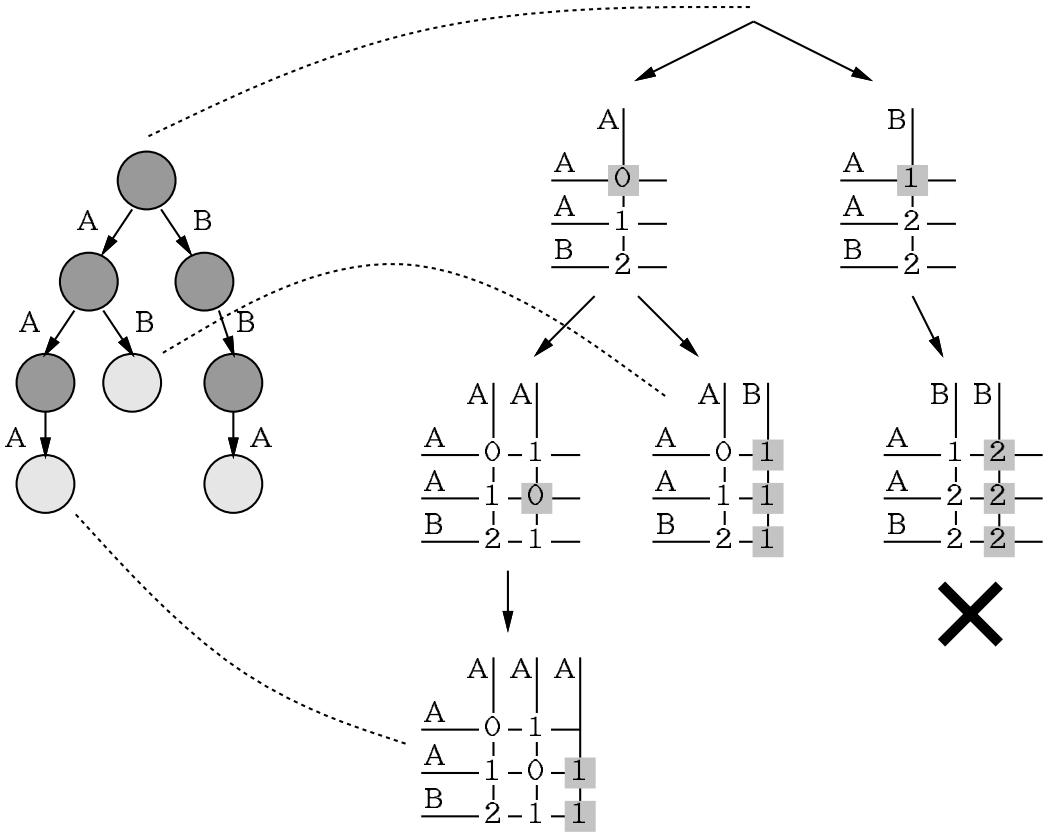


図 4.12 Error-tolerant Recognition の例

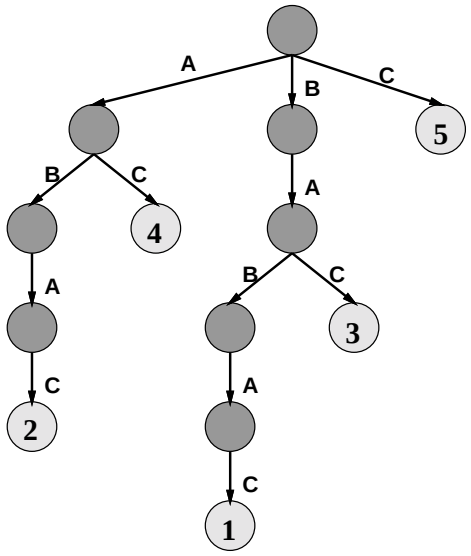
4.2.4 提案する手法

ここでは前節で説明した Error-tolerant Recognition アルゴリズムを拡張してフルテキスト類似検索を実現する方法を説明する。

ポイントは、以下の 2 点である。

- 辞書類似検索に特化している Error-tolerant Recognition アルゴリズムをフルテキスト類似検索に適用する
- トライ構造の欠点であるデータ領域の非効率性を解消するために、Suffix Array[18] を用いた疑似トライを利用する

これらについて説明し、最後に、定性的な特性を調べるため簡単な実験を行う。



テキスト	B	A	B	A	C
インデックスポイント	1	2	3	4	5

図 4.13 Suffix Tree (図 4.6 再載)

Error-tolerant Recognition アルゴリズムのフルテキスト類似検索への適用

ここでは、Error-tolerant Recognition アルゴリズムのフルテキスト類似検索への適用について説明する。

フルテキスト検索のために、図 4.12 のような辞書的なトライではなく、図 4.13 に示すような Suffix Tree というフルテキスト検索ためのトライを用いる。suffix とはテキスト中のある位置からテキストの終了までを含む文字列であり (4.1.1 節を参照)、Suffix Tree とはテキストの全ての suffix を対象としたトライである。図 4.13 は、テキスト BABAC の全ての suffix (BABAC, ABAC, BAC, AC, C) に対してトライを作成したものである。終端ノード上の数字はインデックスポイント (その suffix がテキストの何文字目から始まるか表す) である。

フルテキスト類似検索では検索キーワードと検索結果文字列の長さをそろえる必要はないので、Error-tolerant Recognition アルゴリズムで Suffix Tree を扱うことを考えた場合、アルゴリズムの検索結果出力条件「終端ノードか否かの判定」(図 4.11: 2. (b) iv.) は必要ない。例として、図 4.14 を用いて説明する。図 4.14 で Error-tolerant Recognition アルゴリズムを用い、Suffix Tree に対してペナルティの合計が 1 以下という条件で文字列 BA を類似検索してみる。ギャップペナルティと置換ペナルティ (A → B,B → A) はどちらも 1 とする。トライを [B]→[A]→[B] と辿った時点で BA(ペナルティ合計 0), B, BAA(ペ

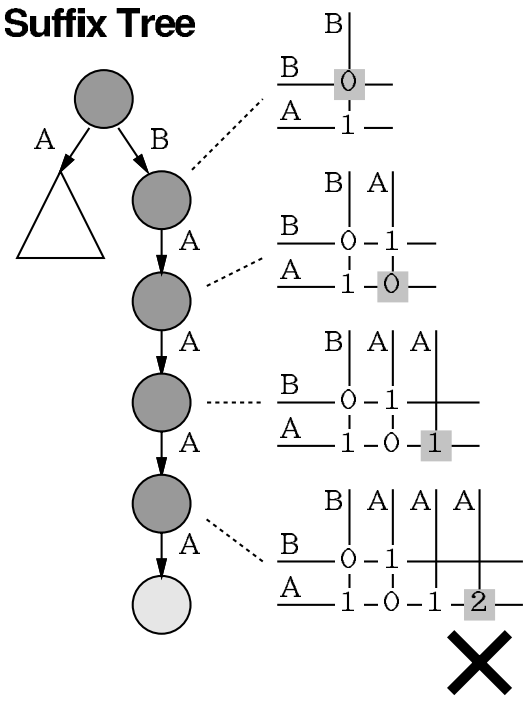


図 4.14 Error-tolerant Recognition アルゴリズムの不都合

ナルティ合計 1) を検索結果として出力して欲しいのだが、Error-tolerant Recognition アルゴリズムでは終端ノードまで行かないと検索結果を出力してくれない (図 4.11: 2. (b) iv.)。結局さらに辿ることになり、ペナルティ合計が 1 を越え、途中で打ち切られてしまう。

そこで、単純にアルゴリズム (図 4.11) の 2. (b) iv. から「終端ノードか否かの判定」を取り除き以下のように書き換える。

$P_{m,k} \leq t$ ならば検索結果として Y を出力する。

これにより終端ノードのあるなしに関わらずペナルティ合計がある限度以内の検索結果を出力でき、Suffix Tree に対しても使用できるアルゴリズムになる。

Suffix Array による疑似トライの実現

しかし、Suffix Tree の問題点としてデータ領域の非効率性が挙げられる。これは木構造のための付属情報が原因となっている。この問題を Suffix Array を用いた疑似トライ

により解決した。詳しくは 4.1.3 節を参照されたい。

実験

これまでに説明した Error-tolerant Recognition の Suffix Tree への適用と疑似トライによりフルテキスト類似検索システムを実現した。

システムの動作確認と定性的な特性を調べるために簡単な実験を行った。

大腸菌の塩基配列データ (約 4.7M バイト)[5] に対して、検索キーワードの文字数、ペナルティ合計の限度 (threshold) をいろいろ変えて検索時間を測定した。キーワードはランダムに作成されたものを用いた。これを 10 回行い、検索時間の平均をとった。図 4.15 に結果を示す。検索時間 (time) の単位は秒である。検索時間には検索結果の表示時間は含まれていない。使用したマシンは Sun Ultra 1(Solaris 2.5.1) である。実験 (1) は、ギャップペナルティが 1、異なる文字同士の置換ペナルティが 1 のときの、実験 (2) では、ギャップペナルティが 2、置換ペナルティが 1 のときの検索時間を示した。

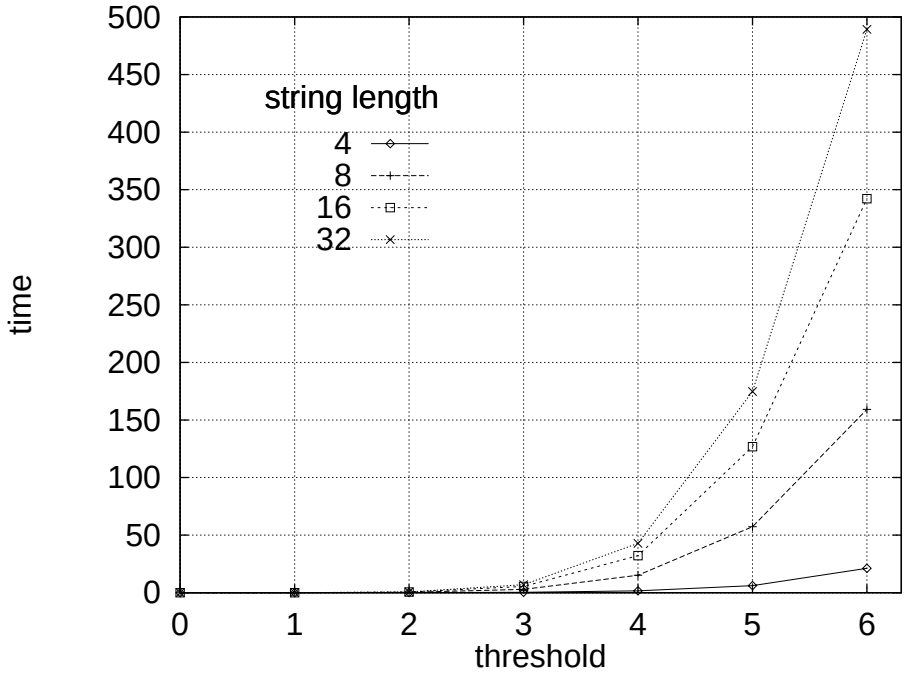
ペナルティ合計の限度を増やすに連れて検索時間が指数的に増えていくことが分かる。また、ギャップペナルティの設定により検索時間にかなりの差が見られる。これは、ギャップペナルティを低く設定するとギャップ操作が頻繁に起こりトライの探索範囲が極端に広がるからである。今回のデータでは字種が 4 つ (A,C,G,T) であったが、字種が多ければ (当然、ペナルティの設定に依存するが) 置換操作による探索範囲の増大が見込まれる。

実験結果から、検索対象に異なりが多い場合にはあまり適さないが、スペルミス、語尾、助詞などの小量の異なりを許容するような用途には十分利用できることが分かる。

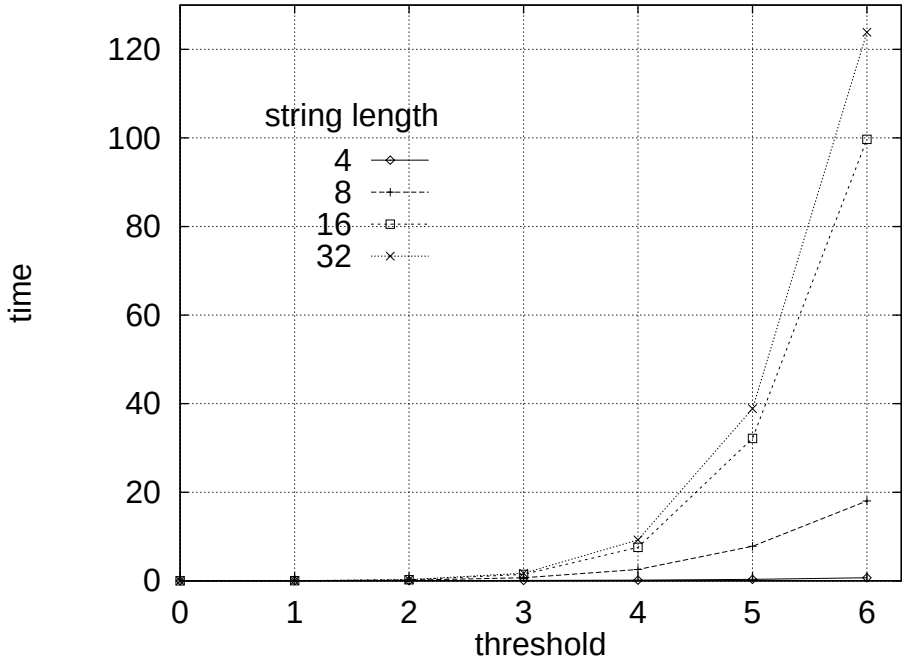
4.2.5 おわりに

データ領域の問題で実現が困難であったフルテキスト類似検索を Suffix Array を用いることにより実現できた。本節で作成したシステムは、様々な分野で用いることのできるような汎用的なシステムになっている。SUFARY のホームページ [20] にて公開していく予定である。

[5] A, C, G, T の 4 種類の文字から構成されているテキストデータである。以下の URL から入手できる。
http://bsw3.aist-nara.ac.jp/GTC/mori/



実験 (1)：ギャップペナルティ $= 1$, 置換ペナルティ $= 1$



実験 (2)：ギャップペナルティ $= 2$, 置換ペナルティ $= 1$

図 4.15 実験結果

4.3. 転置インデックス法と Suffix Array による類似検索

4.3.1 はじめに

現在注目している文と類似した用例 (文) を検索するという機能は、本論文で主張しているタグ付きコーパス作成支援だけでなく、用例に基づく自然言語処理システム全般において非常に重要である。本節では、類似度の高い用例をユーザが利用して何らかの作業を行うという前提に基づき、データベースから類似用例を検索する機能について論じる。例えば、翻訳作業支援、作文支援などのシステムでは、このような類似用例検索機能はシステムの核となる。翻訳作業支援では、ユーザが翻訳したい文と類似した文をシステムが探し出しその対訳文を得ることができれば、ユーザはそれを修正することにより、少ない作業量で一貫性のある翻訳を行うことができる。また、作文支援では、システムが入力した文の類似例文を検索して結果をリストとして提示することにより、ユーザは適切な表現を選択し、自然な文を作成することができる。

類似用例検索の速度や精度 (検索結果の妥当性) を向上させることは、これらのシステム全体の性能の向上に直接結び付くことであり、重要な課題である。

本節では、検索用データ構造に Suffix Array[18] を用いて、「より長い部分文字列がマッチした文はより類似度が高くなる」という人間の直感に合った類似用例検索手法を提案する。従来のテキスト走査や転置インデックスによる方法では、速度と精度のトレードオフがある。テキスト走査による方法は、全ての用例を一つ一つ照合していくため検索漏れがなく、検索の精度は高いが、速度は遅い。一方、転置インデックスによる方法は、あらかじめ用意したインデックスを用いるため検索の速度は前者と比べ著しく速いが、検索漏れの可能性が高く精度は前者ほどではない。提案する手法は、速度と精度のバランスが良いく、実装が容易で、インデックスサイズが小さい実用的な手法である。

4.3.2 節では、これまでに行われてきた類似用例検索と本手法との関係について述べる。4.3.3 節では本手法の詳細を、4.3.4 節では実験による評価について述べる。

4.3.2 これまでの研究

本研究の立場を明確にするため、従来の類似用例の研究における以下の論点について整理する。

- 使用する言語情報
- 基本ヒューリスティック

- 検索手法

使用する言語情報

テキストの類似度を定義する方法は、表層情報 (文字・単語) に基づく方法、構造に基づく方法に大きく分けられる。表層情報に基づく方法は表層の文字列、一次元での言語的な情報 (単語の品詞や意味)、構造に基づく方法は構文木・フレーズなどの構造の一致・類似を重視する。

構造に基づく方法は、実例型機械翻訳では多く用いられている [50]。言語的に理想的な類似用例を検索することが可能であるが、以下のような問題がある。

- 構文解析処理に時間がかかる (データ作成時、または、検索時)
- 検索精度が構文解析処理の精度に依存する

一方、文字・単語などの表層情報に基づく方法は、構造レベルで類似している用例などには対処できないが、構文解析処理などの複雑な処理に依存しないため容易に実装できるという利点がある。

本節では後者の表層情報に基づく方法を採用する。これは、言語情報を用いることによる複雑さを避け、類似度計算を必要とするより多くの用途で柔軟に利用できるようにするためである。また、他の手法との比較を容易にするというねらいもある。

本節では、マッチングの単位として用いる文字・単語を「トークン」と呼ぶ。何をトークンと定義するかは、対象とする言語や検索の目的などに依存する。例えば、日本語の場合、単純に文字をトークンと定義したり、形態素解析処理後の形態素をトークンと定義することが多い。英語は単語の区切りが認識しやすいので、文字ではなく単語 (あるいはその原形) をトークンと定義することが多い。

Baldwin ら [60] は日英翻訳メモリに対して、分かち書きによる検索への影響を調べた結果、日本語では形態素ベースよりも文字ベースでトークンを定義する方が良いと結論づけている。しかし、本節では、アルゴリズム的な評価に重点を置くので、「トークンをどう定義するのが良いのか」といった言語処理に直接関わる議論には立ち入らない。

基本ヒューリスティック

文字列同士の類似度を定義するための基本ヒューリスティックとして以下のものを採用する。

> 二つの文字列で、一致する文字が多く、かつ、連続して一致する文字が多いほど類似度が高い。

我々は、検索キー文に対して、連続して一致する文字列が多く含まれている文を、より似ている文と考える傾向がある。例えば以下に示す例 [61] を考える。

A = 問題を解決する
B = 彼はきのう 問題を解決 した
B' = 問題 の 解 法 を 決定 する

A と B の一致する文字数は 5、A と B' は 6 であり、単に一致する文字数で類似度を考えると、B' の方がより A に似ているということになり、我々の直観に反する。連続して一致する文字列をどのように類似度に反映させるかということが、類似用例検索では重要になる。本手法では、部分文字列の類似度は、その部分文字列の長さ (連続したトークン数) の約 2 乗で与えており、連続を重視している。

また、もう一つの重要なヒューリスティックとして、一致する文字列の順序制約がある。例えば部分文字列の出現順序制約を考慮した以下の例では、A と B の一致する文字数は 3、A と B' は 4 であり、B' の方がより A に似ているということになり、我々の直観に反する。

A = 昨日は最悪だった。
B = 最悪だ ね、昨日。　(最悪だ ね、昨日。)
B' = 今 日は最 高の天気 だ ね。

順序制約を考慮しなければ、B の括弧内の例のように、一致する文字列は 5 となり、B の方が B' より A に似ているということになり、直観に合う。このような、順序制約を考慮しない類似用例検索に関しては、池田ら [62] の研究がある。

順序制約を考慮すべきか否かは、先程の「トークンをどう定義するのが良いのか」という問題と同様、対象とする言語や検索の目的などに依存する。類似用例検索における順序制約に関しては佐藤 [61]、田中 [63]、市原ら [64]、Baldwin ら [60] 等で議論されている。

Baldwin らは日英翻訳メモリに対する実験から、順序制約を考慮する方法が優れていると結論づけている。Baldwin らは、翻訳対象文 (日文) で翻訳メモリを検索して得られる翻訳用例文 (英文) と、あらかじめ用意しておいた意図された翻訳 (英文) との編集距離に基づいて評価している。

もちろん翻訳対象文と類似した文が存在すれば有用であるが、ほとんどの場合は一部だけしか類似していない文が多い。この場合、ユーザは翻訳用例の一部分だけを利用する。このような観点からは、翻訳対象文に含まれる、フレーズ等のある程度長い部分文字列を単位として検索する必要がある。この長い部分文字列は翻訳メモリ内の一つの文内では、順序制約にしばられている必要はない。このような理由から、本節では順序制約を考慮しない方法を重視する。

しかし、後で述べる実験では、順序制約を考慮する場合としない場合のでの効果を調べるため、両方で精度評価を行っている。

レコード番号	テキスト
1	いつのまにか
2	いつのことか
3	彼のことか

(a) テキストデータベースの例

部分列	出現するレコード番号
いつの	1, 2
ことか	2, 3
つのこ	2
つのま	1
...	...

(b) 長さ 3 の部分列インデックス

図 4.16 部分列インデックス

検索手法

これまで述べてきたような、表層の文字列に基づき類似用例を検索する手法として、動的計画法などのテキスト走査による方法、部分列インデックス等の転置インデックスを用いる方法などが知られている。

テキスト走査による方法は、レコード数 N に対して $O(N)$ の計算量がかかってしまう。このような方法は、類似度が最大となるレコードを見つけるために、データベース中の全てのレコードに対して一つ一つ照合する必要があり、検索速度に限界がある。テキスト走査による方法については、文献 [65]、文献 [66] を参照されたい。4.2.3 節で説明した DP マッチングもこのテキスト走査による方法である。

一方、転置インデックスを用いる方法は高速である。転置インデックスを用いた方法として、佐藤 [61][22] によるものを紹介する。佐藤は、部分列インデックスを用いた最適照合検索法を提案している。部分列インデックスとは、テキストデータベース中の長さ k のあらゆる部分列に対して、その部分列が現れるレコード番号を与えた表である。図 4.16 に k が 3 の場合の例 [61] を示す。

検索は、検索キー文中の全ての k 文字の部分列で部分列インデックスを検索し、各レコードに対応するスコア (類似度に相当) に特定の値を足し込んでいき、そのスコアでレコードをソートすることにより実現する。部分列の長さ k はあらかじめ決めておく必要がある。佐藤は $k = 2, k = 3$ の二種類の部分列インデックスを用いる CTM2 という手法を提案している。

本節では、テキスト走査に近い高い精度と、転置インデックスの検索の速さのバランスのとれた手法を提案する。

提案手法はテキスト走査による方法ではなく、転置インデックスによる方法に基づいている。しかし、部分列インデックスによる方法とは異なり、Suffix Array[18] というデータ構造を用いることで全ての長さ k の部分列を同時に扱うことができる。そのため、「連

続して一致する部分列」をより重要視していると言え、より人間の直観に合った類似用例検索ができるという利点がある。また、対象言語や状況に応じて部分列の長さをあらかじめ決めてる手間がいらず、適用が容易であるという利点もある。

4.3.3 提案手法

ここでは、本節で提案する類似用例検索手法について解説する。

本手法で用いる基礎となるデータ構造 Suffix Array は 4.1 節で解説した。ここでは、まず、本手法の説明に必要となる概念「common prefix search」について解説し、実際のアルゴリズムについて「処理の流れ」で詳細を、「例による説明」で例を用いて解説し、「計算量」ではこのアルゴリズムの計算量について述べる。

common prefix search とは

本手法を説明する際に必要となる概念である common prefix search について説明する。common prefix とは、検索キーの prefix になっている検索対象文中の部分文字列である。本節では common prefix search とはこの common prefix を探す処理を指す。

提案手法では、類似度計算のため、全ての common prefix を得る必要がある。例えば、図 4.4 の Suffix Array でインデクシングされたデータに対して、検索キー「さくさくさ」で common prefix search を行うとする。この結果、得られた全ての common prefix は「さ (1,3,5)」「さく (1,3,5)」「さくさ (1,3)」「さくさく (1,3)」「さくさくさ (1)」となる (括弧内の数字はインデックスポイント)。この処理は、Suffix Array を擬似的なトライ構造 (4.1.3 節を参照) とみなし、1 トークン (例では、一文字＝1 トークン) ずつ進めながら検索を行うことにより実現している。

処理の流れ

本節では、類似用例検索処理のアルゴリズムを解説する。

ここで提案する類似用例検索手法では、複数のレコードからなるデータベースを仮定する。データベースの各レコードには ID (レコード番号) と類似用例検索の対象となる用例 (文) が含まれている。

本手法での類似度計算は、検索キーと個別の文との最長の common prefix を求め、その長さ (トークン数) の約 2 乗の値をその文を含むレコードの類似度に足し合わせていることに相当する。これにより、より長い部分文字列を含む文により高い類似度を与えることができる。

図 4.17 にこのアルゴリズムを示す。アルゴリズムでの「スコア」は、類似度に相当する。j の for ループ内が common prefix search に相当する[6] 。

1 トークンのみの exact match は、ヒット数が多く、処理に時間がかかるため無視している。図 4.17 の for 文の条件内の $j = i + 1$ がトークン長 1 を無視する処理に相当する。これは unigram の出現情報を無視していることになる。

なお、検索結果であるレコードの ID の集合 S は、重複した ID を持たない。これは、キー側での部分トークン列を複数含む文 (レコード) のスコアが不当に高くなるのを防ぐためである。例えば、以下に示す極端な例では、A と B よりも、A と B' の方がスコアが高くなってしまう。

A = 問題は解けたぞ
B = この 問題 だけ 解け ない
B' = この 問題 を 問題 視するのが 問題 である

ID 集合での重複を防ぐことでこの問題を回避できる。類似度の定義における重複については文献 [61] に詳しい。

例による説明

英語を例にこのアルゴリズムを説明する。もちろん、本手法は、英語以外の言語でも適用可能である。ここでは「単語 (+ 後に続くスペース)」をトークンと定義する。

まず、検索キーの先頭単語の位置から、common prefix search を行う[7] 。次に示す図は、検索キー "This is the pen which I love." で検索した結果を示している。

`This is the pen which I love.`

Common Prefix		Common Prefix を含むレコードの ID
`This`	長さ: 1	7, 32, 80, 121, 201, 224
`This is`	2	32, 80, 121, 201
`This is the`	3	32, 121
`This is the pen`	4	32
`This is the pen which`	5	×

[6] 実際は、疑似トライによる方法 (4.1.3 節) で common prefix search を行なっているが、ここでは分かりやすくするため単純なアルゴリズムで common prefix search を記述してある。

[7] 前述のアルゴリズムでは、common prefix search とスコアの加算を同時に行っているが、ここでは説明を簡単にするため実際のアルゴリズムと変えている。しかし、最終的な検索結果は変わらない。

- N: データベースに含まれるレコードの数
- $t_1, ..., t_M$: 検索キーを構成するトークン列
- M: 検索キーを構成するトークンの数
- $score_k$: ID k のレコードのスコア $(1 \leq k \leq N)$

1. 類似スコアの計算:

   for $(i = 1; i <= M; i++)$

   - for $(j = i + 1; j <= M; j++)$
     - トークン列 $t_i, ..., t_j$ をキーにデータベースを検索 (exact match) する。
     - 結果をレコード ID の集合 S に格納。
     - S が空でなければ、S の要素であるそれぞれの ID についてそのレコードのスコアを加算する。
       - foreach $k \subset S$ { $score_k += j - i + 2$ }
     - S が空ならば、j のループから抜ける。

2. ランキング:

   ID を $score_{ID}$ の降順でソートする。上位に来るものがより類似した文となる。

図 4.17 提案手法のアルゴリズム

そして、各 common prefix が含まれるレコードに、common prefix の長さ (トークン数) をスコアとして足し込む (長さ 1 のものは無視する)。文中の下線はマッチした common prefix を表す。

ID	文	スコア
32	This is the pen.	$4 + 3 + 2 = \mathbf{9}$
121	This is the man.	$3 + 2 = \mathbf{5}$
80	This is my bag.	$2 = \mathbf{2}$
201	This is a book.	$2 = \mathbf{2}$

次に、検索キーの検索開始位置を 1 トークンずらして、また common prefix search を行う。

`This is the pen which I love.`

Common Prefix	Common Prefix を含むレコードの ID
`is`	7, 32, 80, 121, 201, 250, 401
`is the`	32, 121, 250, 303
`is the pen`	32, 250
`is the pen which`	250
`is the pen which I`	×

そして、先程と同じように、各 common prefix が含まれるレコードに、common prefix の長さをスコアとして足し込む。スコア中の下線の数値は、先程までのスコアを表す。

ID	文	スコア
250	It is the pen which he loved.	$\underline{0} + 4 + 3 + 2 = \mathbf{9}$
32	This is the pen.	$\underline{9} + 3 + 2 = \mathbf{14}$
121	This is the man.	$\underline{5} + 2 = \mathbf{7}$

この処理を検索キーの末尾まで繰り返す。最後に、各レコードのスコアでランキングする。上位のものを類似用例検索結果とする。

計算量

4.1 節で述べた通り、Suffix Array では二分探索で文字列検索を行う。ゆえに、Suffix Array でトライ検索を行う場合は、N をデータベースのサイズ、M を検索キーの文字列長とすると、計算量は $O(M \log N)$ となる。本手法では、1 トークンずつずらしながらトライ検索を行うので、最悪の場合の計算量は $O(M^2 \log N)$ になる。

テキスト走査による検索の計算量は $O(N)$ だが、大量のデータから類似用例検索を行うという現実的な問題において、この計算量の違いは大きい。また、部分列インデックスによる検索の計算量はデータベースのサイズに対して理論的には定数 ($O(1)$) であるが、実用上、提案手法との検索時間の差は問題のない範囲である。

手法のまとめ

本手法は、転置インデックスに基づく手法の一つであるが、部分列インデックスによる方法と異なり、インデックスに用いる文字列の長さの制限がないため、「より長い部分文字列がマッチした文はより類似度が高くなる」という人間の直感に合った検索を行なうことができる。また、部分列インデックスによる方法では、検索対象となる言語や用途などにより最適なインデックス文字列長を考える必要があるが、本手法では文字列の長さの制限がないため最適なインデックス文字列長について考慮する必要がなく、実装が容易であり、汎用性も高い。

4.3.4 実験

提案手法を評価するために実験を行った。検索に要する時間と、検索の精度の調査、及び、インデックスサイズの比較を行った。実験により、本手法の以下の特徴を示す。

速度 テキスト走査による手法と比べ高速な検索が可能なこと

精度 それにもかかわらずテキスト走査による結果との検索結果の一致率が高いこと

精度 部分列インデックスによる結果よりも、単語・フレーズの出現順序を考慮しない場合の一致率が高いこと

サイズ 部分列インデックスによる方法と比べインデックスのサイズがほとんど変わらないこと

準備

実験に用いるデータ、ツール、及び、実験の手順について説明する。

データ

検索対象データとして、日経新聞 CD-ROM1996 年版 [67] からヘッドラインのみを抽出したファイル (約 17 万行、9.1M バイト) を用いた。以下にファイルの一部を示す。一行一文というフォーマットになっている。

> ラグビー大学選手権――同大・大河原主将、準決勝の明大戦雪辱へ気合十分。
> 「強い木造住宅」模索続く――「阪神大震災」級の実験（にゅーすスケッチ）
> ＮＹ円大幅下落、１１２円台後半（海外外為）

住之江、中学の窓ガラス１６０枚割られる。
橋本内閣新閣僚の横顔――文相奥田幹生氏、商工族、各派にもパイプ。

このデータを対象に、提案手法と既存手法との比較を行う。

提案手法に基づき類似用例検索を行うプログラム

提案手法に基づき類似用例検索を行うプログラム rbk は、Linux 上で C++ で実装した。Suffix Arrays は、SUFARY([20], 4.1.2 節) を使用している。common prefix search は 4.1.3 節の疑似トライを利用している。

テキスト走査による類似検索のアルゴリズム

提案する手法の比較対象として使用するテキスト走査による類似検索ツールについて解説する。

テキスト走査による類似度ヒューリスティックの 1 つに、語順による制約を行うか否かがある。この実験では、語順による制約を行う場合と、行わない場合、両方で評価を行う。

語順制約を考慮する場合

順序制約を考慮する類似検索ツール simgrep は、CTM1[68] で採用されている類似度を用い、ダイナミックプログラミング法により最適な照合結果を取りだすツールである。

CTM1 の類似度 sim1 は、二つの文字 (トークン) 列 $A = a_1a_2...a_A$, $B = b_1b_2...b_B$ に対して、以下のように定義される [61]。

$$\mathrm{sim1}(A, B) = s_{l_A, l_B} \tag{4.1}$$

$$s_{i,j} = \begin{cases} 0 & \text{if } i = 0 \vee j = 0 \\ \max \begin{pmatrix} s_{i-1,j-1} + \min(cm_{i,j}, W) \\ s_{i-1,j} \\ s_{i,j-1} \end{pmatrix} & \text{otherwise} \end{cases} \tag{4.2}$$

$$cm_{i,j} = \begin{cases} 0 & \text{if } i = 0 \vee j = 0 \vee m_{i,j} = 0 \\ cm_{i-1,j-1} + m_{i,j} & \text{otherwise} \end{cases} \tag{4.3}$$

	天	気	は	昨	日	は	最	悪	だ	っ	た
今	・	・	・	・	・	・	・	・	・	・	・
日	・	・	・	・	1	・	・	・	・	・	・
は	・	・	1	・	・	1	・	・	・	・	・
最	・	・	・	・	・	・	1	・	・	・	・
高	・	・	・	・	・	・	・	・	・	・	・
の	・	・	・	・	・	・	・	・	・	・	・
天	1	・	・	・	・	・	・	・	・	・	・
気	・	1	・	・	・	・	・	・	・	・	・

$m_{i,j}$

	天	気	は	昨	日	は	最	悪	だ	っ	た
今	・	・	・	・	・	・	・	・	・	・	・
日	・	・	・	・	1	・	・	・	・	・	・
は	・	・	1	・	・	2	・	・	・	・	・
最	・	・	・	・	・	・	3	・	・	・	・
高	・	・	・	・	・	・	・	・	・	・	・
の	・	・	・	・	・	・	・	・	・	・	・
天	1	・	・	・	・	・	・	・	・	・	・
気	・	2	・	・	・	・	・	・	・	・	・

$cm_{i,j}$

図 4.18 sim1 の計算例

$$m_{i,j} = \begin{cases} 1 & \text{if } a_i = b_j \\ 0 & \text{otherwise} \end{cases} \tag{4.4}$$

参考に、図 4.18 に $m_{i,j}$ と $cm_{i,j}$ 計算例を示す。

順序制約を考慮しない場合

順序制約を考慮しない類似検索ツール eqm は、制約充足問題を探策により解決する手法を用いて類似用例検索を行う。

順序による制約を行わない類似用例検索に関しては、池田ら [62] が列島 (Archipelago) アルゴリズムを提案している。列島アルゴリズムでは、一対の単語対応を「島」、島が右下方向に連続して並んだものを「列島」と呼ぶ。図 4.19 の例では、黒丸 (●) が島、連なる 3 つの島 (「最高の天気」に対応) の対応が列島である。

列島のスコアは以下のように計算する。

1. 二つの連続する島を一つにまとめ列島にする。列島のスコアは、二つの島のスコア ($m_{i,j}$ に相当) を足して二乗したもの。
2. 二つの連続する列島をさらに結合。スコアはやはり二つの列島のスコアを足して二乗したもの。この列島の結合を繰りかえす。

これらのスコアの合計が最大になるように島を選択していく。

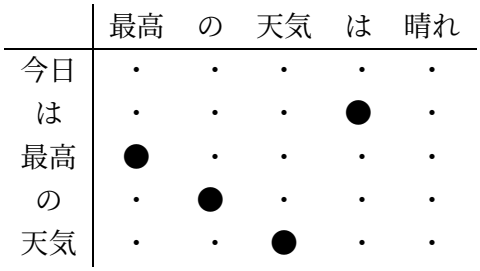


図 4.19 島と列島

eqm では、列島 $A = a_1a_2...a_N$ のスコア $score_A$ を以下の式で計算している。

$$score_A = \sum_{i=1}^{N} i^2 \tag{4.5}$$

合計スコアが同じマッチング結果が複数あった場合は、順序制約を満たしているものを優先する。また、最高スコアを持つマッチング結果を求めるためにバックトラックを行う必要があるため、大規模なファイルに対しては検索速度は非常に遅くなる。

検索時間の測定

検索速度を評価するための手順は以下の通りである。

1. 対象データ (4.3.4 節) から 100 文 (ヘッドライン) をランダムに選ぶ。
2. それぞれの文を検索キーとして、全文に対して類似用例検索を行う[8] 。
3. それぞれの検索結果を類似度順に 10 位まで取り出し表示する。

以下にこのタスクの出力フォーマットを示す。各文をキーとした類似検索の結果を上位 10 個出力し、各ランキング間は、“---” で区切る。

> 長野県内の株式売買高、８月前月比１６％減。
> 宇都宮財務事務所まとめ、栃木県内の株式売買高８月前月比１２％減。
> 栃木県内の株式売買高、７月は前月比１５％減。
> 長野県内６月、株式売買高、前月比２６％減。
> 栃木県内の株式売買高、２月前月比２６％増。

[8] タスクの性質上、ランキング結果の一位は、検索キーとして用いた文そのものになる。

手法	実行時間 (秒)
提案手法 rbk (max = ∞)	**43**
部分列インデックス (擬似 CTM2) rbk (max = 3)	**36** (参考)
テキスト走査、順序制約あり simgrep	**659**
テキスト走査、順序制約なし eqm	**4203** (参考)

図 4.20 検索に要した時間

長野県内の株式売買高、１月は２．９％プラス。
北陸３県、８月の株式売買高、株数で前月比１９．３％減。
北陸３県、７月の株式売買高、前月比１０．８％減少。
北陸３県、６月の株式売買高、株数で前月比２５％減。
栃木県内、代金は６．６％増――９月株式売買高、前月比５．６％減少。

軍事連絡事務所、各派、設置で合意――ボスニア・ヘルツェゴビナ。
ユーゴ、ボスニア・ヘルツェゴビナ、セルビア人勢力への制裁解除。
ボスニア・ヘルツェゴビナの民族抗争の歴史、第２次大戦で亀裂深まる。
ボスニア・ヘルツェゴビナ、国連難民事務所、物資輸送を再開。
ボスニア・ヘルツェゴビナのセルビア人勢力、イスラム側との交渉中断（ダイジェスト
)
ほころび？クリントン外交――ボスニア・中東和平・ロシアの改革支持。
ＥＵ、対ロシア、外交・安保で対話強化――ボスニアにらむ。
サラエボ、亀裂深く――ボスニア和平の焦点に、セルビア人、自治が“ 命綱 ”。
カラジッチ氏、公職すべて辞任、米特使会見――ボスニア選挙へ前進。
軍の大規模移動、２年禁止――ボスニア各派、信頼醸成措置に合意。

このタスクを実行するのに要した時間を図4.20 に示す。実行環境は、PC (FMV DESKPOWER) で、CPU は AMD-Athlon 700MHz、メモリ は 512M、OS は Linux である。それぞれ、2 回計測し、2 回目のものを結果とした。

rbk は提案手法に基づき類似用例検索を行うプログラムである。部分マッチに用いる部分文字列の最大トークン長 (max) を様々に変えて検索できるように実装した。例えば、max = 5 の場合、長さ 2, 3, 4, 5 の部分文字 (トークン) 列のみを検索に用いることになる。

提案手法は、長さ制限なしの部分文字列で検索を行うので、max = ∞ である。

部分列インデックスによる方法との比較は、高速化チューニングされた適切なツールが入手できなかったため、この最大トークン長 max を指定する機能を用いて行った。部分列インデックスによる方法である、佐藤 [61] の手法 CTM2 では、長さ 2 と 3 の部分文字列を組み合わせて用いるので、max = 3 のときとほぼ同等である[9] 。そこで、max = 3 のときを擬似 CTM2 としてその検索時間と提案手法の検索時間とを比較してみた。検索時間は提案手法 (max = ∞) が、擬似 CTM2 のおよそ 1.2 倍 (43 : 36) かかるが、これは実用上問題のない範囲である。この Suffix Array を用いた擬似 CTM2 (部分列インデックス手法) の実装は最適とは言えず、もっと高速な実装が可能であるはずである。ゆえにここでは参考記録としておく。しかし、たとえ最適化したとしても、速度が何倍にもなるとは考えにくく、結局、検索時間の差は実用上問題のない範囲に収まると考える。

高速化チューニングされたテキスト走査による検索ツール simgrep と比較して、提案手法の検索速度はおよそ 15 倍となっており、実用上十分な性能と言える。なお、rbk は完全な高速化のためのチューニングがされていないため、実行時間は今後の開発状況により、さらに大きく開く可能性がある。

eqm は、高速化のためのチューニングを行っていないため、極端に遅くなっているため参考記録としているが、それでも比較すると提案手法の検索速度は約 100 倍となっている。

検索精度の測定

4.3.4 節で説明したテキスト走査による類似用例検索ランキング結果を正解データとみなし、提案手法での類似用例検索ランキング結果と比較し、精度を評価する。

検索精度を評価するための手順は以下の通りである。

1. 対象データ (4.3.4 節) から 100 文 (ヘッドライン) をランダムに選ぶ。
2. それぞれの文を検索キーとして、全文に対して、提案手法に基づき類似用例検索を行うプログラム rbk で類似用例検索を行う。
3. それぞれの検索結果を類似度順に上位 5 位まで取り出す。[10]

[9] スコアの加算値が弱冠異なるが、検索時間には影響しない。

[10] タスクの性質上、ランキング結果の一位は、検索キーとして用いた文そのものになる。また、

	転置インデックス (max = 3)	提案手法 (max = ∞)
順序制約あり	**82.6**	76
順序制約なし	78.6	**93**

図 4.21 検索精度

4. 正解データに検索結果の中の文がいくつ含まれているか (順位は考慮しない) を基準に一致率を計算し、それを検索精度とする。

例えば、検索結果として得られた 5 文中、正解データ (5 文) のどれかと同じ文が 3 文ある場合、一致率は 60% となる。

また、検索時間の測定のときと同様に、部分列インデックスによる方法との比較のために、rbk での部分マッチに用いる部分文字列の最大トークン長 (max) を∞ (提案手法)、3(擬似 CTM2) にして類似用例検索を行った。

図 4.21 にその精度を示す。100 文での rbk での検索結果と正解データとの一致率の平均をとったものである。「順序制約あり」は simgrep (順序制約を考慮した場合) の出力結果との一致率、「順序制約なし」は eqm (順序制約を考慮しない場合) の出力結果との一致率である。前者は、提案手法の精度 (max = ∞) は高くなく、CTM2 での方法 (max = 3) が勝っている。後者では、提案手法の精度が勝っている。

これらの結果から、提案手法は、順序制約ありの場合よりも順序制約なしの場合に、より適していることが分かる。

順序制約ありの場合は、長さ 2 と 3 の部分文字列を組み合わせて用いる、CTM2 の方法が最も適しており、長い部分文字列の使用は逆効果である。順序制約なしの場合では、提案手法の長い部分文字列の使用が非常に有効である。このような、順序制約ありとなしでの、使用する部分文字列の長さ (max) と一致率の関係は、長い部分文字列の出現順が入れ換わっている文の扱いに起因する。以下に示すのは、「１０月神奈川県内倒産、負債額１兆円超す――帝国データバンクまとめ。」を検索キーとして検索した結果ランキング (1 位〜3 位) である。それぞれの 2 位の文は、「県内倒産、負債額」「帝国データ」「まとめ」という部分文字列の出現順の入れ換わりに影響されている[11] 。

```
--- simgrep
1: １０月神奈川県内倒産、負債額１兆円超す――帝国データバンクまとめ。
```

検索時間測定実験とは異なり、上位 5 位を対象にしている。

[11] この例の場合は、上位 5 個以内なので、本節での精度評価には影響はない

2: ９６年山梨県内倒産、負債額５３．２%増３４６億円、帝国データまとめ。
3: 山梨県内、３月倒産は１０件、負債額４３%減――帝国データバンク調べ。
--- 提案手法
1: １０月神奈川県内倒産、負債額１兆円超す――帝国データバンクまとめ。
2: 帝国データ７月まとめ、神奈川県内倒産、負債額６割減――小規模・不況型多く。
3: 山梨県内、３月倒産は１０件、負債額４３%減――帝国データバンク調べ。

この実験では、順序制約ありの場合では、提案手法はCTM2 の方法と比べ精度は低いが、提案手法とテキスト走査を部分的に組み合わせることで、提案手法の精度を向上させることが可能である。このことを確認するため、rbk で検索結果を上位 20 件に絞り込み、それに対して simgrep で再度ランキングを行い上位 5 位までを取り出し、前述の評価手順で検索精度を測定した。その結果、精度は、転置インデックス (max = 3) も提案手法 (max = ∞) もほぼ同じ 88% となった。

これらのことから、提案手法は、順序制約なしの場合には従来の手法より高精度であり、かつ、順序制約ありの場合にも十分対応可能であるということが言える。

インデックスサイズ

インデックスファイルの大きさは、佐藤 [61] の方法 (CTM2) では、k が 1 のときは元データベースのインデックス対象部分のサイズの約 1/3、k が 2 のときは約 1/2、k が 3 のときは約 1 倍となっている。本手法では、9.1M バイトの元データに対してインデックスは約 19M バイトであった。これは元データの約 2 倍であり、佐藤の方法と比べても特に問題のないサイズと言える。

提案手法では、インデックスサイズが使用する部分列の長さに依存しないという Suffix Array の性質により、最適と思われる部分列の長さをあらかじめ決めておく必要がないという利点がある。

4.3.5 おわりに

提案手法は、「より長い部分文字列がマッチした文はより類似度が高くなる」という人間の直感に合った検索を行う。つまり、提案手法は、検索キーに対して、細切れになった短い部分文字列が多くマッチする用例よりも、長い連続した部分文字列が少量マッチする用例の方を重視する傾向があるということである。この特徴により、順序制約を考慮しない場合、部分列インデックスによる方法よりも精度が高くなる。

提案手法は、高速文字列検索のためのデータ構造である Suffix Array を用いているため、テキスト走査による方法よりも検索が速く、インデックスサイズも元データベースファイルサイズの 2 倍ほどであり、許容範囲内である。また、Suffix Array は可変長の部分文字列を高速に検索でき、しかも、可変長部分文字列検索のための特別なインデックスを必要としない。つまり、文字列検索のためのインデックス (= Suffix Array) さえあれば、全ての部分文字列が検索可能となる。同様のことを部分列インデックス法で行おうとすると、作成するインデックスの数が膨大になり実用的でない。そのため、部分列インデックス法では、対象データに応じて部分列の長さをあらかじめ決めておく必要がある。提案手法では、そのような手間 (準備) はいらず、様々な言語・状況に容易に適用できる。これらの点から、提案手法は、前述の「より長い部分文字列がマッチした文はより類似度が高くなる」という基準による類似用例検索には最適な方法である。

単語対応、フレーズ対応等の処理は、提案手法と比べ、テキスト走査による方法の方が人手による調整が容易であるが、計算量の問題で検索速度が遅く、テキスト走査のみで類似用例検索を行うのは現実的ではない。順序制約を考慮する場合は、部分列インデックスによる方法で絞り込みを行い、その結果に対してテキスト走査を行うという方法が一般に行われている。しかし、順序制約を考慮しない場合は、提案手法により絞り込みを行う方法が、検索速度・精度を考慮すれば最適な方法であると言える。

第5章

結論

本論文では、大規模言語コーパス、特にタグ付きコーパスに関して、蓄積・利用 (解析・検索) に関する技術を一連の流れとしてとらえ、個別の技術について研究・開発を行ない、その有効性を示した。

まず 2 章では、言語に依存しない形態素解析処理について解説した。特定の言語や同系統の数言語の解析のみを念頭に置いて開発されている現状の言語技術の問題点を明らかにし、それに基づき特定の言語に依存しない処理の枠組について述べた。

その提案する枠組のベースとなるものが形態素片という概念である。形態素片は、わかち書きされる言語で考えられる分割の可能性、及び、わかち書きされる言語とされない言語との整合性も考慮したもので、これまで漠然と定義されていたトークンに替わる概念である。このアイディアは極めて単純ではあるが、今まで実現されなかった以下のことを可能にした。

- 言語に依存しない辞書検索の実現

  言語によって (つまり、文字ベース、単語ベースかによって) 検索方針が異なった辞書検索部を、形態素片を利用することに共通のモジュールとして実装できるようにした。

- トークン認識における非論理的な曖昧性の排除

  形態素片を利用することにより、無駄な辞書検索を省くことができるようにした (形態素解析の速度が 4 倍になることを実験により証明した) 。

メジャーな言語のタグ付きコーパスは近年大量に蓄積されつつあるが、新たな言語を対象にタグ付きコーパスの蓄積を行なうことを考えると、特定の言語や同系統の数言語の解析のみを念頭に置いて開発されている現状の言語技術では不十分である。しかし、この技術により、世界中のあらゆる言語の形態素タグ付与が可能になり、タグ付きコーパスの蓄積という視点から見ると大きな進歩と言える。

次に 3 章では、品詞タグ付きコーパスの蓄積とそれを利用した解析の技術について説明した。

タグ付きコーパスを作成するのは人間である。これまでのタグ付きコーパスの蓄積プロジェクトではエディタ等の単純なインターフェースでの作業が主であった。そこで、作業の効率化、作成されたコーパスの質の一貫性などを考慮した作業環境を開発し、さらに、作業の効率化、質の一貫性を保つため、それまでの作業で蓄積したコーパスを利用した事例ベース学習機能を実装した。学習機能のシミュレートを行ない効果を測定した

結果、作業の効率化へつながるとの結論が得られた。この機能は、効率化による作業者の負担軽減だけでなく、タグ付与の一貫性を保たせることもできる。品詞タグ付きコーパスの質の維持は重要な問題であり、この点でもこの機能は、タグ付きコーパスの蓄積に大いに貢献する。

また、タグ付きコーパス作成の効率化という観点から、これまでの一般的なタグ付きコーパスの解析処理での利用法 (蓄積されたタグ付きコーパスを、統計的手法でパラメタ学習し解析処理で使用する方法) とは異なる方法を提案した。

一つは、タグ付きコーパスと人手で作成されたルールの融合である。一般的なパラメタ学習では、ある程度の精度を出すには大量のタグ付きコーパスが必要となり、それらが整備されていない未開拓な分野には不向きである。一方、ルールベースの解析器は比較的簡単に実装でき、手作業による調整もしやすいが、多くの人的労力が必要となるのが問題である。そこで、タグ付きコーパスと人手で作成されたルールを融合し、蓄積途中の少量のコーパスでも十分に利用できる手法を提案した。実際には、人手で作成されたルールに付与されているコスト値を指数関数を用いて疑似的な確率パラメタに変換し、それをタグ付きコーパスから学習された確率パラメタと最適な比率で足し合わせることで、両者の統合を行なう。実験により、小量のタグ付きコーパスでもこの手法により、精度の向上に貢献できることを示した。これにより、大量のタグ付きコーパスの入手が困難な状況でも、漸進的に解析精度を向上させることが可能になり、タグ付きコーパスの蓄積作業に有効な手法と言える。

もう一つは、タグ付きコーパスをそのまま事例データとして利用する方法である。タグ付きコーパスを直接部分文字列検索し、マッチした部分文字列とそれに付与されたタグを元に形態素解析を行なう手法を提案した。競合時の候補選択の手法に大いに発展の余地があるが、現段階でも統計的な手法と比べ精度に遜色がないことを実験により示した。この手法は、パラメタ学習を行なう手法とはあきらかに一線を画すものである。この手法が持つ以下のような特徴により、パラメタ学習による手法を補完するものであると言える。

- 頻度の低い言語現象を扱うことができる
- 解析を失敗している場合、解析の基となるコーパスのタグ付与誤りが容易に分かり、コーパスの保守作業に有用である
- 作業者によりタグ付与された文がコーパスに追加されれば、即座に解析に利用できる
- 同じ文がコーパスに存在すれば解析精度は 100% になる

さらに 4 章では、タグ付きコーパス作成作業の際の参照に有用である類似検索技術について述べた。タグ付きコーパス作成作業の際には、過去のタグ付与基準などを参照するため、これまで蓄積してきたタグ付きコーパスを検索する必要がある。この際には、全く同じ文だけではなく、部分的に類似している文を検索できるのが望ましい。そこで、Suffix Array と呼ばれる文字列検索のためのデータ構造を用いた二種類の類似検索手法を提案した。

一つは動的計画法と組み合わせたもので、検索キーと数文字異なるだけの文字列を漏れなく探すという手法である。これは、見つけたいものがある程度分かっている場合に有効である。ここでは、動的計画法とトライを組み合わせた既存の方法を Suffix Array を用い、全文類似検索に拡張した。実験により、検索対象に異なりが多い場合にはあまり適さないが、スペルミス、語尾、助詞などの異なりを許容するような異なりの少ない類似検索では、効果を発揮することを示した。

もう一つは、転置インデックスと組み合わせたもので、異なり文字が多い場合、つまり、全体は似ていなくてもどこかの部分が似ているものを探すという手法である。これは、長い文に対する類似検索に向いている。より長い部分文字列が多くマッチした文はより類似度が高いという尺度に基づき、人間の直観に合った類似検索を行なう。実験により、テキスト走査による検索結果 (ベースライン) との比較において、既存の手法より優れていることを示した。

このような類似検索技術によるコーパス検索は、作業者にとってはタグ付与の際の基準を明らかにできるので作業効率の向上につながる。また、このようにして作業者が過去のタグ付与の例を参照して蓄積したタグ付きコーパスは、同じ文脈の文字列に対しては同じタグが付与されるという、タグ付与の一貫性が保たれ、質の向上にもつながる。このような点から、類似検索技術はタグ付きコーパスの蓄積作業に必要不可欠な技術と言える。

以上のように、本論文では、タグ付きコーパスの蓄積という観点から、タグ付きコーパスの蓄積・利用 (解析・検索) に関する技術について研究を行ない、個々の技術が全体の部分として有効なものであることを示した。

今後の課題として、実用化が挙げられる。実際の大規模なコーパス作成プロジェクトでの、このような統合的な技術を用いたものは、まだ無い。実際のプロジェクトでの利用を促進し、検証を行ない、より良い作業環境の構築を目指したい。

謝 辞

主指導教官である松本裕治教授には、研究内容など多くの点で有益な御意見、御助言を頂きました。先生の熱心な御指導がなければ、本研究を進めることはできなかったと思います。ここに心から深く感謝致します。

これまで一緒に研究してきた松本研究室の皆さん、および、松本研究室 OB の皆さんには、公私に渡り、とてもお世話になりました。研究会や勉強会のみならず普段の何気ない会話からも、さまざまな知識を得ることができました。

最後に、御意見、御質問を頂いた全ての方々に感謝致します。

参考文献

[1] 永田昌明. 単語と辞書, 岩波講座 言語の科学, 第 3 巻, 2. 形態素解析, pp. 53–92. 岩波書店, 12 1997.

[2] Henry Kučera and W.Nelson Francis. *Computational Analysis of Present-Day American English*. Brown University Press, 1967.

[3] Alfred V. Aho. *Handbook of Theoretical Computer Science*, Vol. A: Algorithms and Complexity, chapter 5. Algorithms for Finding Patterns in String, pp. 255–300. The MIT Press, 1990.

[4] Hiroshi Maruyama. Backtracking-Free Dictionary Access Method for Japanese Morphological Analysis. In *COLING-94: Proceedings of the 15th International Conference on Computational Linguistics*, pp. 208–213, 1994.

[5] David D. Palmer and Marti A. Hearst. Adaptive Multilingual Sentence Boundary Disambiguation. *Computational Linguistics*, Vol. 23, No. 2, pp. 241–267, June 1997.

[6] Beatrice Santorini. *Part-of-Speech Tagging Guidelines for the Penn Treebank Project (3rd Revision, 2nd printing)*, June 1990.

[7] Jonathan J. Webster and Chunyu Kit. Tokenization as the Initial Phase in NLP. In *COLING-92: Proceedings of the 14th International Conference on Computational Linguistics*, Vol. 4, pp. 1106–1110, August 1992.

[8] Jon Mills. Lexicon Based Critical Tokenization: An Algorithm. In *Euralex'98*, pp. 213–220, August 1998.

[9] Jin Guo. Critical Tokenization and its Properties. *Computational Linguistics*, Vol. 23, No. 4, pp. 569–596, December 1997.

[10] Tatsuo Yamashita and Yuji Matsumoto. Language Independent Morphological Analysis. In *Proceedings of the 6th Conference on Applied Natural Language Processing*, pp. 232–328, 2000.

[11] 全泳杓. 新国語表記法. 統一出版社, 1996.

[12] 平野善隆. 用言の活用を考慮した韓国語品詞体系の提案とそれを用いた韓国語形態素解析. Master's thesis, 奈良先端科学技術大学院大学, February 1997.

[13] Yoshitaka Hirano and Yuji Matsumoto. A Proposal of Korean Conjugation System and its Application to Morphological Analysis. In *PACLIC 11: Language, Information and Computation *Selected Papers from the 11th Pacific Asia Conference on Language, Information and Computation*, pp. 229–236, December 1996.

[14] Wolfgang Lezius, Reinhard Rapp, and Manfred Wettler. A Freely Available Morphological Analyzer, Disambiguator and Context Sensitive Lemmatizer for German. In *COLING-ACL '98*, Vol. 2, pp. 743–748, August 1998.

[15] 山下達雄. MOZ と LimaTK の説明書. NAIST Computational Linguistics Laboratory, <http://cl.aist-nara.ac.jp/~tatuo-y/ma/>, August 1999.

[16] 山下達雄, 松本裕治. 言語に依存しない形態素解析ツールキットの開発. 情報処理学会研究会報告 98-NL-129, pp. 17–22, November 1998.

[17] 山下達雄. 形態素解析システムの機能分割と再利用を目指して.「言語資源の共有と再利用」シンポジウム, <http://www.etl.go.jp/etl/nl/sympo99/>, February 1999.

[18] Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. In *1st ACM-SIAM Symposium on Discrete Algorithms*, pp. 319–327, 1990.

[19] R. Donald Morrison. Patricia — practical algorithm to retrieve information coded in alphanumeric. In *JACM 15*, pp. 514–534, 1968.

[20] 山下達雄. SUFARY ガイド 1999-06-29 版. NAIST Computational Linguistics Laboratory, <http://cl.aist-nara.ac.jp/lab/nlt/ss/>, June 1999.

[21] Christopher D. Manning and Hinrich Schütze. *Foundations of Statistical Natural Language Processing*. The MIT Press, 1999.

[22] 長尾真. 自然言語処理, 岩波講座ソフトウェア科学, 第 15 巻. 岩波書店, 1996.

[23] 黒橋禎夫, 長尾真. 日本語形態素解析システム JUMAN version 3.5, 1998.

[24] 松本裕治, 北内啓, 山下達雄, 平野善隆. 日本語形態素解析システム『茶筌』version 2.0 使用説明書. NAIST Technical Report NAIST-IS-TR99008, <http://cl.aist-nara.ac.jp/lab/nlt/chasen.html>, April 1999.

[25] Masahiko Haruno and Yuji Matsumoto. Mistake-Driven Mixture of Hierarchical Tag Context Trees. In *ACL-EACL '97 Proceedings: 35th Annual*

Meeting of the Association for Computational Linguistics and 8th Conference of the European Chapter of the Association for Computational Linguistics, pp. 230–237, July 1997.

[26] Masaaki Nagata. A Part of Speech Estimation Method for Japanese Unknown Words using a Statistical Model of Morphology and Context. In *37th Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference*, pp. 277–284, June 1999.

[27] 新情報処理開発機構. テキストデータベース報告書 平成８年度, 1997.

[28] Roger Mitton. *A Description of A Computer-Usable Dictionary File Based on The Oxford Advanced Learner's Dictionary of Current English*, 1992.

[29] Academia Sinica Institute of Information Science. 中央研究院平衡語料庫的内容興説明, 1995. in Chinese.

[30] Kimmo Koskenniemi. Two-Level for Morphological Analysis. In *IJCAI '83*, pp. 683–685, 1983.

[31] 浅原正幸, 松本裕治. 文節まとめあげと形態素解析の融合. 言語処理学会第５回年次大会予稿集, March 1999.

[32] Mitchell P. Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. Building a Large Annotated Corpus of English:The Penn Treebank. *Computational Linguistics*, Vol. 19, No. 2, pp. 313–330, 1993.

[33] 山下達雄, 松本裕治. 形態素解析視覚化システム ViJUMAN 使用説明書 version 1.0. NAIST Technical Report NAIST-IS-TR96005, February 1996.

[34] 菊田昌弘. 用語解説:パトリシアツリー (patricia tree). 人工知能学会誌, Vol. 11, No. 2, pp. 337–339, March 1996.

[35] Robert Sedgewick. アルゴリズム (Algorithms) 原書第２版 第２巻 探索・文字列・計算幾何. 近代科学社, 1992.

[36] 竹内孔一. 隠れマルコフモデルによる日本語形態素解析システムのパラメータ推定. Master's thesis, 奈良先端科学技術大学院大学, 1995.

[37] 颯々野学, 斎藤由香梨, 松井くにお. アプリケーションのための日本語形態素解析システム. 言語処理学会第３回年次大会予稿集, pp. 441–444, 1997.

[38] 渕武志, 松岡浩司, 高木伸一郎. 保守性を考慮した日本語形態素解析システム. 情報処理学会研究会報告 97-NL-117, pp. 59–66, January 1997.

[39] 北研二, 中村哲, 永田昌明. 音声言語処理 —コーパスに基づくアプローチ—. 森北出版, 1996.

[40] 山下達雄. 規則と確率モデルの統合による形態素解析. Master's thesis, 奈良先端科学技術大学院大学, 1997.

[41] Satoshi Sekine. The Domain Dependence of Parsing. In *Proceedings of the 5th Conference on Applied Natural Language Processing*, pp. 96–102, 1997.

[42] 山地治, 黒橋禎夫, 長尾眞. 連語登録による形態素解析システム JUMAN の精度向上. 言語処理学会第 2 回年次大会予稿集, pp. 73–76, 1996.

[43] 北内啓, 山下達雄, 松本裕治. 日本語形態素解析システムへの可変長連接規則の実装. 言語処理学会第 3 回年次大会予稿集, pp. 437–440, 1997.

[44] 橋田浩一, 杉村領一, 柏岡秀紀, 内山将夫, Christoph J.Neumann. 大域文章修飾: 標準タグによる言語データの大規模な構造化と再利用. 言語処理学会第 3 回年次大会予稿集, pp. 135–138, 1997.

[45] ATR 国際電気通信基礎技術研究所（編）. 自動翻訳電話. ATR 先端テクノロジーシリーズ. オーム社, 1994.

[46] Ricardo Baeza-Yates and Berthier Rieiro-Neto. *Modern Information Retrieval*. ACM Press/Addison Wesley, 1999.

[47] Kenneth W. Church. You shall know a word by the company it keeps. In *NLPRS'95: Natural Language Proceeding Pacific Rim Symposium*, pp. 22–34, 1995.

[48] Jon Bentley. *Programming Pearls*. Addison-Wesley, second edition, 2000.

[49] 山下達雄. 用語解説: Suffix Array. 人工知能学会誌, Vol. 15, No. 6, p. 1142, November 2000.

[50] 佐藤理史. アナロジーによる機械翻訳. 共立出版株式会社, 1997.

[51] 青山典生, 兵藤安昭, 浅井泰博, 応江黔, 池田尚志. 形態素解析と意味コード化に基づく翻訳支援のための類似例文検索システム. 電子情報通信学会技術報告 NLC93-68, pp. 39–45, 1994.

[52] 隅田英一郎, 堤豊. 翻訳支援のための類似用例の実用的検索法. 電子情報通信学会論文誌, Vol. J74-D-II, No. 10, pp. 1437–1447, 1991.

[53] 畑中念, 遠藤裕英. 日本語 OCR 文における英字・カタカナのスペル誤り訂正法. 情報処理学会論文誌, Vol. 38, No. 7, pp. 1317–1327, 1997.

[54] 美馬秀樹, 隅田英一郎, 飯田仁. 類似検索を用いた情報検索システム. 言語処理学会第 2 回年次大会予稿集, pp. 113–116, 1996.

[55] 中村春木, 中井謙太. バイオテクノロジーのためのコンピュータ入門. コロナ社, 1995.

[56] 美宅成樹, 金久實（編）. ヒトゲノム計画と知識情報処理. 培風館, 1995.

[57] Gad M. Landau. Fast Parallel and Serial Approximate String Matching. *Journal of Algorithms*, Vol. 10, pp. 157–169, 1989.

[58] Kemal Oflazer. Error-tolerant Tree Matching. In *COLING-96: Proceedings of The 16th International Conference on Computational Linguistics*, pp. 860–864, 1996.

[59] Kemal Oflazer. Error-tolerant Finite-state Recognition with Applications to Morphological Analysis and Spelling Correction. *Computational Linguistics*, Vol. 22, No. 1, pp. 73–89, 1996.

[60] Timothy Baldwin and Hozumi Tanaka. Balancing up Efficiency and Accuracy in Translation Retrieval. 自然言語処理, Vol. 8, No. 2, pp. 19–37, 2001.

[61] 佐藤理史. 用例検索による日英翻訳支援システム CTM2 —部分列インデックスを用いた最適照合検索—. Technical report, 北陸先端科学技術大学院大学, 1993.

[62] 池田成宏, 内野一, 古瀬蔵. 用例利用型翻訳のための類似用例検索手法. 言語処理学会第 6 回年次大会予稿集, pp. 43–46, March 2000.

[63] 田中英輝. 長い日本語表現の高速類似検索手法. 情報処理学会研究会報告 97-NL-121, pp. 69–74, September 1997.

[64] 市原創, 池原悟, 村上仁一. 要素の順序関係から見た類似文最適照合検索. 言語処理学会第 5 回年次大会予稿集, pp. 521–524, March 1999.

[65] 疋田輝雄. C で書くアルゴリズム. Computer Today ライブラリ 17. サイエンス社, 1995.

[66] Graham A. Stephen. *String Searching Algorithms*. Lecture Notes Series On Computing - Vol.3. World Scientific, 1994.

[67] 日本経済新聞社. 日本経済新聞 CD-ROM 1996 年版. 日本経済新聞社, 1996.

[68] Satoshi Sato. CTM: An Example-Based Translation Aid System. In *COLING-92: Proceedings of the 14th International Conference on Computational Linguistics*, Vol. 4, pp. 1259–1263, 1992.

研究業績

学術論文

1. 山下 達雄, 松本 裕治. 言語に依存しない形態素解析処理の枠組. **自然言語処理**, 7(3):39–56, 2000.

国際会議

1. Tatsuo Yamashita and Yuji Matsumoto. Language Independent Morphological Analysis. In *Proceedings of the 6th Conference on Applied Natural Language Processing*, pp. 232–328, 2000.
2. Yuji Matsumoto and Tatsuo Yamashita. Using Machine Learning Methods to Improve Quality of Tagged Corpora and Learning Models. In *LREC2000 - Second European Conference on Language Resources and Evaluation*, 2000.
3. Tatsuo Yamashita, Masakazu Fujio, and Yuji Matsumoto. Language Independent Tools for Natural Language Processing. In *Proceedings of the Eighteenth International Conference on Computer Processing of Oritental Languages*, pp. 237–240, March 1999.

研究会・大会発表

1. 山下 達雄. 形態素解析システムの機能分割と再利用を目指して. In **「言語資源の共有と再利用」シンポジウム**, <http://www.etl.go.jp/etl/nl/sympo99/>, February 1999.
2. 山下 達雄, 松本 裕治. 言語に依存しない形態素解析ツールキットの開発. In **情報処理学会研究会報告** *98-NL-129*, pages 17–22, November 1998.
3. 松本 裕治, 武田 浩一, 永田 昌明, 宇津呂 武仁, 田代 敏久, 山下 達雄, 林 良彦, 渡辺 日出雄, 竹澤 寿幸. SLP・NL 合同セッション「ここまでできるぞ音声/言語処理技術」. In **情報処理学会研究会報告** *98-NL-125*, pages 1–8, May 1998.

4. 山下 達雄, 松本 裕治. 品詞タグ付きコーパスを直接利用した形態素解析. In **言語処理学会第四回年次大会予稿集**, pages 524–527, 1998.
5. 山下 達雄, 松本 裕治. Suffix array を用いたフルテキスト類似用例検索. In **情報処理学会研究報告** *97-NL-121*, pages 83–91, September 1997.
6. 北内 啓, 山下 達雄, 松本 裕治. 日本語形態素解析システムへの可変長連接規則の実装. In **言語処理学会第** *3* **回年次大会予稿集**, pages 437–440, 1997.
7. 山下 達雄, 松本 裕治. 形態素解析結果の視覚化システム vijuman とその学習機能. In **情報処理学会研究会報告** *96-NL-115*, pages 29–34, September 1996.

その他

1. 山下達雄. 規則と確率モデルの統合による形態素解析. Master's thesis, 奈良先端科学技術大学院大学, 1997.
2. 山下 達雄. 用語解説: Suffix Array. **人工知能学会誌**, 15(6):1142, November 2000.
3. 松本 裕治, 北内 啓, 山下 達雄, 平野 善隆. **日本語形態素解析システム『茶筌』** *version 2.0* **使用説明書**. NAIST Technical Report NAIST-IS-TR99008, <http://cl.aist-nara.ac.jp/lab/nlt/chasen.html>, April 1999.
4. 山下 達雄, 松本 裕治. **形態素解析視覚化システム** *ViJUMAN* **使用説明書** *version 1.0*. NAIST Technical Report NAIST-IS-TR96005, February 1996.