日本語文書を対象とした
n-gram 索引による高速検索手法の研究

小 川 泰 嗣

目 次

図目次

表 目 次

第1章　序論

1.1　文書検索の重要性と高速化への要求

人類の発展とともに、世の中の情報量は増大してきた。コンピュータの発明以降、情報の増大はいっそう加速され、現代は情報化社会と呼ばれるに至った。情報化社会においては、膨大な情報の中かから必要な情報を的確に見つけ出すことが基本的な技能の一つとなっている。膨大な情報の中から必要な情報を的確に見つけ出すことが広い意味での情報検索(information retrieval)である[Sal83b, Tok99]。これに対し、工学における情報検索はユーザによって明確に表現された問合せに対して適合する情報をコンピュータを用いて見つけ出す技術を指す[Yam98a]。

文書検索は、テキスト文書を検索対象とする情報検索である。本研究がテキストを対象としたのは、コンピュータの黎明期から画像・音声・動画等へと情報の表現媒体（メディア）の多様化が進んだ今日に至るまで、テキストが知識を表現する上で最も主要なメディアという地位にあるからである。

文書検索のニーズは最近一層の高まりを見せており、その要因は図1.1のようにまとめることができる。

- 電子化文書の増大

  文書は電子化（コンピュータ上で適切にコード化）によって多様な形態での利用が可能になり、その価値は著しく高まる。コンピュータおよび文書作成ソフトウェアの普及に支えられ、現在では新規に作成される文書は最初から電子的に作成されるようになった。また、OCR（光学的文字読み取り）技術の発展により紙文書の電子化が容易になり、過去の文書の電子化も進んでいる。電子化文書の増大に伴い所望の文書を見つけることはより一層困難となり、検索ニーズを高めている。

- 文書共有化の進展

  コンピュータ同士をネットワークで接続することにより文書は共有され、利用範囲が拡大する。最近のネットワーク技術や利用コストの低価格化に基づくインターネット・イントラネットの発展が文書の共有化を促進している。文書共有化が進展により、共有化された文書を再利用するために検索に対する必要性が増大している。

- 検索ユーザの拡大

  業務効率化のためコンピュータはオフィスに急速に普及した。また、最近ではコンピュータの低価格化・インターネットの普及に伴い家庭への浸透も著しい。従来は限られた専門家だけが行っていた文書検索であるが、今日ではコンピュータの普及とともに拡大した利用者が業務・趣味等の様々な目的のために利用するようになった。

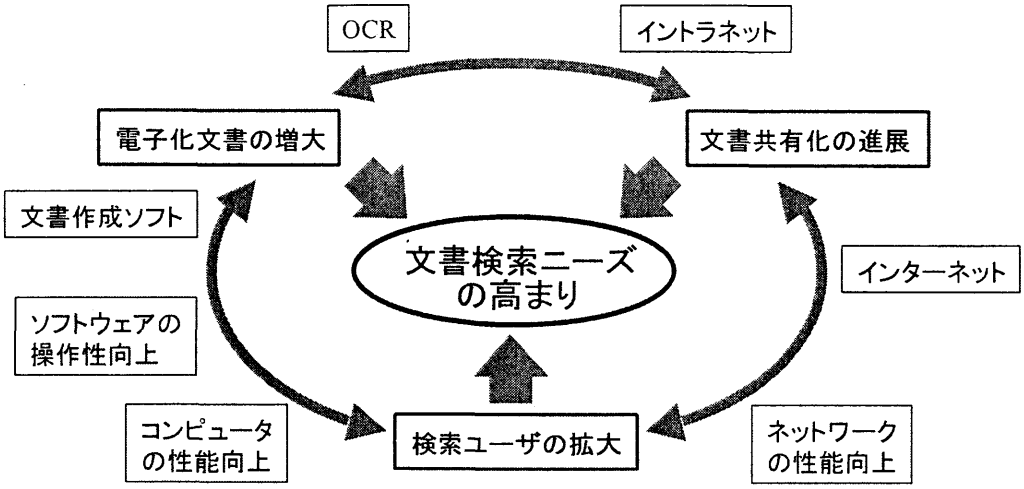


図 1.1: 文書検索ニーズの高まり

ここで述べたような文書を取り巻く状況は検索高速化の要求へと連ながるものである。すなわち、文書の増大、共有化の進展は検索対象を大規模化させるため、膨大な検索対象に対しても快適に応答するべく検索高速化が必要となる。検索ユーザの拡大からは検索システムのスループット向上が求められ、検索速度向上が課題になる。また、拡大したユーザの多様な要求に答えるにはランキング検索等の高度な検索機能を提供しなければならず、複雑化した検索処理に対応できるだけの処理速度が要求されることとなる。

検索高速化への要求は次のような事例からも確認することができる。文書検索の研究コミュニティにおいては、米国の標準化機関 NIST (National Institute of Standards and Technology) 主催の 検索システムの相互性能比較を狙いとした評価会である TREC (Text Retrieval Conference)[1] が大きな役割を果たしている。1992 年に開催された第 1 回 TREC では 2GB という当時としては非常に大量な文書を対象に取り上げ、それ以降も 100GB の Web 文書を対象とする課題を扱うことで理論的な研究と実用化の橋渡しを果たしてきた。一方、商用システム提供者側に大きな影響を与えたのは 90 年代後半から爆発的に発展している WWW (World Wide Web) である。例えば、代表的なインターネットのサーチエンジンである Google では、2002 年 5 月の時点で検索対象ページ数が 20 億、1 日当りの処理検索数が 1.5 億に達している[2]。

1.2 文書検索の基本概念

1.2.1 文書検索モデル

文書検索の概念的なモデルを図 1.2 に示す。検索対象である文書集合に対し、ユーザは探したい文書を表現した検索要求（information need）をモデルに合わせた（システムが受理可能な）問合せ（query; 以下、検索条件とも書く）の形式で与える。文書集合中の各文書と検索条件は適切な内部表現に変換された上で比較照合され、検索条件に合致する文書群が検

[1] URL は http://trec.nist.gov/
[2] URL は http://www.google.com/

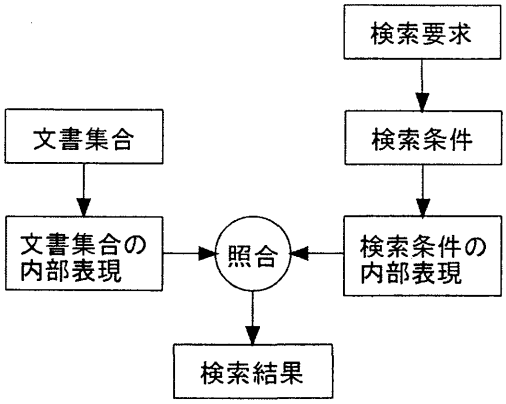


図 1.2: 文書検索のモデル

索結果としてユーザに返される。

内部表現と照合方法に応じて様々な文書検索モデルがこれまでに提案されてきているが、検索モデルは以下の 2 つに大別することができる [Bae99, Fra92, Sal83b]。

- ブーリアン検索

  検索対象文書を検索要求を満足するかしないかの 2 値で判断し、検索条件を満足する文書だけを検索するモデルである。完全一致（exact match）モデルと呼ぶこともある。

  ユーザの検索要求を文書を単一の検索文字列（キーワード）で記述するのは一般に困難である。そこで、ブーリアン検索では、AND、OR 等の論理演算子を導入し、検索文字列を論理演算子で組み合わせたブール式を検索条件とする。さらに、複数の検索文字列の文書内の出現位置を指定する近傍演算子を用いることもある。近傍演算子の例としては、複数の検索文字列が同一文、同一段落などの文書構造上の同じ要素に出現することを指定するものや、相対距離が与えられた文字数あるいは単語数以内であることを指定するものがある。

- ランキング検索

  検索対象文書ごとに検索要求を満足する程度（スコア）を求め、検索対象を順序つけるモデルである。最適照合（best match）モデルと呼ぶこともある。

  ランキング検索はスコア計算方式によって特徴付けられ、これまでに非常に多くの方式が提案されている。代表的なものとしては、文書・検索条件を単語等を軸とするベクトルで表現し、その内積をスコアとするベクトル空間モデル [Sal75]、検索文字列が適合する文書に現れる確率等に基づいてスコアを計算する確率モデル [Rob77] がある。

  ブーリアン検索と同様に演算子で構造化した検索条件を用いることもあるが、自然言語文を検索条件として使用することが一般的である。自然言語文からはシステムによって適切な検索文字列が選択され、それに基づいてスコア計算が行われる。

2 つのモデルは排他的なものではなく、目的に応じて適切なモデルを使い分けることが重要である。例えば、特許検索では漏れのない検索を行う必要があるため、ブーリアン検索がよく使用されるのに対し、インターネットのサーチエンジンでは検索対象の件数が膨大である

ことからランキング検索が一般的である。現実の文書検索システムにおいても、両モデルをサポートするものが存在する。

1.2.2 索引形式

検索モデル・目的に応じて様々な文書検索システムの実装方法がある。最も単純な実装方式は検索時に検索文字列と各文書と文字列照合する逐次検索である。しかし、逐次検索は検索時間が対象規模に比例するため検索対象が大規模である場合には現実的ではなく、内部表現を検索処理向きに配置した索引を用いる必要がある。索引における基本的な内部表現の要素を索引単位と呼ぶ。文書検索の研究は欧米での歴史が長いが、そこでは索引単位として単語が中心的に用いられてきた [Sal83b]。

文書検索で用いられる代表的な索引形式には以下の２つがある [Fra92, Wit94]。

- 転置ファイル

  索引単位ごとに出現した文書識別子等の情報をまとめて記録するもの。例えば、索引単位を単語とした場合、検索条件で与えられた単語に対応する文書識別子を簡単に得ることができるので、検索は非常に高速である。

  文書情報として文書識別子のみを記録する文書レベルの転置ファイルと、文書内の出現位置なども記録する索引単位レベルの転置ファイルがある。文書レベルの転置ファイルは構造が簡単で小型なのに対し、索引単位レベルは複雑でサイズも大きくなるがランキング検索・近接演算等の高度な検索も高速処理できるという利点がある。一方、文書サイズに対し 50 ～ 300%の大きさになるという報告 [Has81] もあり、ファイルサイズが大きいことが問題である。

- シグネチャファイル

  文書ごとにその内容を縮約した情報を記録するもの。索引単位ごとにシグネチャ（signature）と呼ばれるビット列を計算し、それを文書（あるいは文書内の固定長のブロック）でビット ORしたもの（文書シグネチャ）を内部表現とする。検索時には、文書そのものを照合するのではなく、文書シグネチャと検索文字列から文書と同様にして求めたシグネチャを比較することで検索結果を求める。ただし、シグネチャは文書を縮退した表現であるため、本来は検索されるべきでない文書（false drop）が誤って検索されることがある。このような過剰検索を除去するため、シグネチャファイルで検索された文書を検索文字列で文字列照合する必要がある。この意味で、シグネチャファイルを用いた検索は転置ファイルと逐次検索の中間的な方法と言える。

  ファイルサイズ等は転置ファイルよりも小型にすることも可能であるが、文字列照合が必要なので検索速度では転置ファイルに劣る。シグネチャ計算のパラメータ調整によりファイルサイズを制御可能であるが、ファイルサイズと検索速度は相反する関係にあるため、パラメータ調整は難しい。また、ランキング検索等の高度な検索機能をサポートできない点が大きな問題である。

転置ファイル・シグネチャファイルの特徴をまとめたものが表 1.1 である（この表を含む性能比較の表では、○が性能が優れていること、×は性能が劣ること、△は両者の中間であ

表 1.1: 転置ファイルとシグネチャファイルの比較

	転置ファイル	シグネチャファイル
索引サイズ	○	○
登録速度	△	○
検索速度	○	×
近傍演算対応	○	×
ランキング対応	○	×
パラメータ調整	○（不要）	×（必要）

ることを示す)。以前は、シグネチャファイルの方がパラメータ調整により小型化できること、ブーリアン検索が実用システムの主要な検索モデルであったことから、シグネチャファイルも広く使用されていた。その後、転置ファイルの圧縮技術が向上して処理速度を落とすことなくサイズを小型化可能になったこと、ランキング検索が広まったこと等から、最近の文書検索システムでは転置ファイルが一般的となった [Wit94, Zob94]。

1.3 文書検索の言語依存性

文書検索が対象としているものは自然言語で記述されたテキストである。文法・語彙等の特性は言語ごとに異なるため、言語ごとに相応しい文書検索モデル・実装方法も異なるという言語依存性がある [Mya96]。

文書検索の内部表現あるいは索引単位を考える上では言語の構成要素を意識する必要がある。言語の構成要素は、小さいものから順に文字、単語[3]、句、文となる。索引単位として採用し得るものは適度に小さな単位が望ましいことから、文字、単語のいずれかが一般的に用いられる。文字に関しては、文書の識別性が低く、意味の構成単位より小さいことから、効率性・有効性の両面で単語よりも劣るため、隣接する n 個の文字組である文字 n-gram[4]を索引単位として使用することが多い [Kit98, Yam98b]。以下では、特に問題のない限り文字 n-gram を単に n-gram と書くこととする。

単語および n-gram の特徴は、文書検索の観点から、以下のようにまとめられる。

- 単語

  単語とは意味を担う最小単位である。したがって、索引単位を単語とすることで、検索を意味に沿ったものにしやすい。しかし、意味を正しく扱うためには、構文関係・文脈などは単語を超えたレベルの高度な言語処理が必要であり、単語単位とすることによって意味に基づく検索が実現できるわけではない。一方、言語によっては単語の捉え方が複数あり、そのうちのいずれの捉え方が検索に最適であるかが必ずしも明らかではこと、文中の単語を常に正確かつ高速に切り出すのは必ずしも簡単ではないこと等が問題となる。

[3]言語学では意味を担う最小単位を形態素（morpheme）と呼び、単語と区別している。しかし、文書検索においては単語と形態素の区別を厳密に行わずに同等のものとして扱うことが多く、本論文でも区別を行わない。

[4]$n = 1$ の文字 n-gram は文字そのものであり、文字は文字 n-gram の特殊な例として扱うことができる。

- n-gram

  隣接する n 個の文字組のことである。1 個以上の文字から構成される点では単語と共通する面があるが、単語が文法的な役割と意味を持つのに対して n-gram は文法・意味などによって決定されるものではない点で単語とは大きく異なる。

  文字 n-gram による検索は表記に基づく表層的なものである。しかし、実際には表記だけで単語を特定できることも多く、単語の場合に近いレベルの意味的な検索は達成可能である。

索引単位を単語とする索引方式は単語索引、文字 n-gram とするものは**n-gram 索引**と呼ばれる。なお、索引単位は 1.2.2 節の索引方式とは独立に選択可能である。すなわち、単語索引・n-gram 索引とも、転置ファイル・シグネチャファイルのいずれの索引形式によって実現可能である。

索引単位の選択時に考慮すべき言語の特性には以下のものがある。

- 正書法

  文章の書き方を定めた規則を正書法（orthography）と呼ぶ [Kit02]。例えば、英語において、単語と単語の間に空白を入れて分かち書きすること、固有名詞を大文字で書き始めることといった規則が英語の正書法に含まれる。テキストからの単語切り出しが容易に実現できるかには正書法が大きく影響する。単語の切り出しが容易であれば単語を索引単位に使用することが可能だが、困難であったり、抽出精度に問題がある場合には n-gram の方がよいと考えられる。

- 表記の揺れ

  同一単語に対する表記が複数存在する現象を表記の揺れと呼ぶ。例えば、英語において「色」を "color" あるいは "colour" と表記したり、日本語において「電子計算機」を「コンピューター」、「コンピュータ」、「コンピユウタ」等と表記するのが表記の揺れである。表記の揺れに関係なく検索できるようにするためには、揺れの関係にある複数の表記を正規化（統一）する機能が必要である。英語の異表記正規化には単語レベルの情報が必要であることが多く、単語索引との相性がよい。一方、日本語におけるカタカナ表記の揺れであれば文字列変換処理によって統一できるので、索引単位を n-gram としても差し支えない。

- 屈折

  単語の数（単数か複数か等）・格（主語か目的語か等）・時制（現在か過去か等）等の文法的性質に伴う語形変化を屈折と言い、屈折が豊富な言語を屈折言語（inflectional language）と呼ぶ。例えば、英語において、名詞の末尾に 's' を付けると複数形、動詞に末尾に '(e)d' を付けると過去形を表すのが屈折である。これに対し、助詞・助動詞等の付属語によって単語の文法的性質を表す言語を膠着言語（agglutinative language）と呼ぶ。屈折言語では、単語の文法的性質の違いに関係なく検索できるようにするためには語形変化した単語の原形や語形変化に影響されない語幹（stem）を求める語幹処理（stemming）が重要となる。語幹処理には単語が切り出されていることが前提になるため、語幹処理が必要な場合には索引単位は単語とするのが自然である。

- 造語力

  複合語の生成され易さを造語力と呼ぶ。例えば、日本語であれば「データの通信」に対し「データ通信」という複合語、ドイツ語あれば "Datenkommunikation" という複合語があるのに対し、英語では "data communication" のように「データ」「通信」を表す単語から成る名詞句として書く。文書検索において、複合語を同等な句とでも照合させるには単語索引であれば複合語を構成単語に分割する必要があるのに対し、n-gram 索引であれば部分文字列照合によって複合語分割を代替することが可能である。複合語解析には基本語彙やそれらの出現確率、単語の連接確率などのデータが整備されている必要があるため、データ整備のコストがかかる。さらに、データが整備されていても、常に正確に複合語解析ができるとは限らない。したがって、造語力の高い言語に対しては n-gram の方が向いている。

英語・フランス語・ドイツ語等の欧州言語は屈折言語で単語を分かち書きするという共通の性質を持っている [Tok98a]。ただし、造語力に関しては、英語・フランス語等の多くの言語が低いのに対し、ドイツ語等の一部の言語は高いという違いがある。一方、日本語・中国語・韓国語等の東アジアの言語は膠着言語であり、単語を分かち書きせず、造語力も高い等の特徴を共通に持っている [Tok98b]。

このような言語の特徴から、欧州言語に対しては単語索引が用いられるのが一般的であり、n-gram 索引の適用例 [Cav95] は限定的である。欧州言語のなかでは造語力の高いドイツ語に対しては n-gram 索引の適用例も見ることができるようになる [Hed01]。一方、東アジア言語に対しては n-gram 索引が単語索引と同等に用いられている [Che97, Kwo97, Lee96, Raj97, Wil97]。

1.4 日本語文書検索における索引方式

前節では言語依存性の観点から索引方式を考察したが、本節では単語索引・n-gram 索引の日本語への適用について詳細に検討する。

1.4.1 単語索引

単語索引は索引単位を単語とする。したがって、検索文字列が単語として出現している文書がヒットと判定され、検索結果を構成する。

日本語は単語を分かち書きしないので、単語索引を実現するには単語切り出しに形態素解析の利用が不可欠である。検索対象文書を形態素解析するのはもちろんのこと、検索文字列自体が複合語であることも考えられるのでユーザが指定した検索文字列についても形態素解析が必要である。検索文字列が複合語で形態素解析によって複数の単語に分割された場合、文書中に同様の単語の並びが出現しているかを検査することで複合語の検索文字列が出現している文書の特定が可能である。

日本語の形態素解析は、切り出し単位の大きさから複合語レベルと（複合語の）構成語レベルの２つに大別される。複合語レベルであれば、数百語程度の機能語辞書と比較的少数のルールによって形態素解析可能である [Kam95]。一方、構成語レベルを実現するためには少なくとも数万語規模の大規模辞書が不可欠である[5]。処理速度や辞書管理などの点からは複

合語レベルのほうが望ましいが、複合語と同等の概念・内容を表現する方法は多数存在するので、文書検索の有効性を高めるためには構成語レベルの形態素解析が不可欠と考えられる。実際、複合語・構成語レベルの形態素解析に基づく性能比較結果によれば、構成語レベルの方が優れていることを実験的に示した研究もある [Oga96c]。

構成語レベルの形態素解析を文書検索に適用するには以下のような問題点がある。

- 数万から数十万語規模の単語辞書が必要である。形態素解析系にはあらかじめ大規模辞書が添付されているが、固有名詞や学術用語などを中心に新しい単語がつぎつぎに生み出されるので、単語辞書の拡充は避けられない。しかし、辞書拡充は人手 — 言語的な知識と対象分野に関する知識を兼ね備えた専門家 — で行う必要があるため、システムの運用コストが増大する。

- 単語辞書を拡充しても、索引に登録済みの文書は拡充前の辞書に基づいて形態素解析が行われている。したがって、登録済み文書についても辞書拡充の効果を反映するには、索引を作り直しが必要である。

- 形態素解析の処理速度が向上しているとは言え、形態素解析が登録時間増大の原因となることは避けられない。少量の文書登録時には影響は小さいが、大量文書を扱う索引の初期作成や再作成時には大きな問題となる。

- 形態素解析の誤りが検索漏れ・過剰検索を起こし、検索精度を低下させる。

1.4.2 n-gram 索引

n-gram 索引は索引単位を n-gram とする。検索文字列が文字列として現れている文書がヒットと判定され、検索結果を構成する。n-gram 索引では単語という言語的な構成単位を考慮しないため、同一検索文字列に対する検索結果であっても単語索引のものとは一般には一致しない。

n-gram では単語境界を考慮する必要が無く、形態素解析などの言語的なツールは不要である。単語境界を考慮していないといっても、表意文字である漢字に関しては n-gram 索引でも実質的には単語索引に近い機能を果たすことができる。一方、外来語の表記に用いられるカタカナ・アルファベットは表音文字であり、n-gram 索引と単語索引の差は大きい。

n-gram 索引では、n の値によって性能が大きく影響されるので、その値の選択が重要である [Kit98, Mic96]。一般的には文字の異なり数が少ない程文字そのものの文書の識別性が低いため、n を大きくするする必要がある。日本語では、日常用いられる文字の異なり数は約 2000 文字程度であり、n は 1, 2 等が採用されることが多い。

n-gram 索引には以下の問題点がある。

- n-gram 索引では、2.2 節に詳しく説明するように文字列の長さによって検索処理方法が異なる。検索文字列が索引単位よりも長い場合（文字数が n を超える場合）には、検索文字列中の n-gram の全てが存在し、かつ連続した位置に出現していることを確認しなければならない。この位置検査によって検索時間が増大することが問題となる。

[5] 文字ごとの単語の先頭あるいは末尾になる頻度・確率によって構成語への分割を実現する研究もある [Kag96, Nak95, Oga97b, Oga99b]。しかし、解析精度、統計情報の収集方法などの問題が残っている。

表 1.2: 単語索引と n-gram 索引の比較

	単語索引	n-gram 索引
索引サイズ	○	×
登録速度	△	○
検索速度	○	×
検索精度	△	○
検索漏れ	×（あり）	○（なし）
過剰検索	△（多少あり）	×（あり）
辞書管理	×（必要）	○（不要）

- 検索文字列を単語としてではなく文字列として捉えて検索を行なうため、過剰検索が起こりやすい。例えば、「帯電 (electrification)」という検索語に対して「携帯電話」が現れている文書も検索されてしまう。その結果、検索精度が低下する。
- 登録文書から抽出される索引単位数は単語よりも n-gram の方が多いため、索引サイズは一般に大きい。

1.4.3 単語索引と n-gram 索引の比較

単語索引と n-gram 索引の比較結果を表 1.2 にまとめた。索引サイズ・検索速度の点からは単語索引が優れているが、登録速度・辞書管理不要という点からは n-gram 索引が望ましい。検索精度について見ると、n-gram 索引は検索漏れが無いものの検索ノイズがあり、単語索引は過剰ノイズは少ないものの検索漏れが起こるというように相反する特性を持っている。ただし、ランキング検索によって的確な文書をすくい上げることが可能ということを考慮すると、検索漏れよりも検索ノイズの方が問題は小さい。したがって、トータルでは n-gram 索引の方が検索精度では若干有利と考えられる。

1.5 検索高速化に関する従来研究

日本語文書を対象とした検索高速化に関する従来研究は n-gram 索引を対象としたものに集中している。表 1.2 に示したように、n-gram 索引には検索速度が遅いという問題があるため、その高速化が研究されるのは当然のことといえる。もう一つの理由としては、単語索引についてはいったん形態素解析により単語が識別されてしまえば言語による差異は小さく、欧米言語を対象として研究されてきた単語索引の高速化手法をそのまま日本語に適用できることがあげられる。以上の理由から、日本語に特化した単語索引の高速化の研究が少ないのだと考えられる。

n-gram 索引の形式としては、転置ファイル・シグネチャファイルのいずれもが高速化の研究対象となっている。シグネチャファイル形式の n-gram 索引は文字成分表と呼ばれ、90 年代前半に集中的に研究開発が進んだ [Fuj94, Fur94, Hat92, Iwa93, Miy90]。筆者も文字成分表の検索高速化に関する研究 [Oga95b, Oga95a, Oga96a, Oga96b] を実施したが、その内

容は付録 A で紹介する。

しかし、表 1.1 に示したように転置ファイルの方が検索速度・機能の高さで優れており、90 年代後半以降は転置ファイル形式の n-gram 索引が研究開発の中心になっている [Aka96b, Jon98, Kaw96, Mat97a, Oga98, Sug96]。したがって、以下では転置ファイルを対象としたものを取り上げることとする。なお、転置ファイルには索引単位の文書内出現位置を記録する索引単位レベルと記録しない文書レベルの 2 種類があるが、n-gram 索引では複数の n-gram を含む検索文字列の処理のために索引単位レベルが用いられる。

検索高速化に関する従来研究を整理すると、以下のように分類することができる。

1.5.1 n-gram 抽出法

検索速度は n に依存しているため、n を適切に調整することにより検索は高速化可能である。以下のような n-gram 抽出の調整に基づく高速化方法が提案されている。

- n の組み合せ

  2.5 節で説明するように検索文字列が n より短いか長いかによって性能に与える影響が反対なので、単一の n によってあらゆる場合の検索を高速化することはできない。この問題に対する最も単純な解決策は複数の n に対する n-gram 索引を組み合わせることである [Aka96a, Sug96]。日本語の文字数の多さを考慮すると、n より短い検索処理を発生させないことが重要となる。その一方、組み合わせる索引数に応じて索引サイズは増大するので、それほど多くの n を組み合わせることは非現実的である。実際には、$n = 1, 2$ あるいはせいぜい $n = 1, 2, 3$ の組み合わせに限定される [Kit98, Mic96]。

- 文字種適応

  複数の n を組み合わせる方法は単純ではあるが、索引サイズが大きくなるという問題点がある。これに対するものとして、文字種に応じて n を調整する方法（以下、文字種適応型 n-gram 索引）がある [Kaw96, Yok97]。日本語にはひらがな・カタカナ・漢字等の複数の文字種があり、それぞれに特性が異なる [Iwa93, Oga96b]。すなわち、ひらがなは助詞・助動詞など機能語にもちいられることが多いため検索文字列として使用されることは少ない、カタカナは外来語に用いられるためカタカナ語は長い、漢字は漢語に用いられるためカタカナ語と比較して単語は短い等の違いがある。異なる文字種から構成される単語は少ないが、漢字とひらがなの連続は頻繁に見られるというような特徴もある。

  こうした特性に基づいて、文字種適応型では同じ文字種の連続部分ではその文字種に応じて n を調整するとともに、異なる文字種の連続部は前後の文字種に応じて n-gram（$n > 1$）を抽出するか決定する。文字種適応型は索引サイズをそれ程大きくすることなく検索を高速化できる。

- n の動的調整

  上に示した 2 つの方法は、索引作成時に n-gram の抽出法（n の値）を決定する静的な方法である。しかし、追加・削除のある動的な状況では、検索対象の n-gram の頻度

分布と抽出法の決定に用いた頻度分布にはずれが生じ、決定した n-gram 抽出法が最適であるとは限らないという問題がある。

この点を鑑みて考案されたのは、n-gram 抽出法を動的に決定する方法である [Sug96]。この方法では、初期状態では n を 1 としておき、ある文字の出現頻度が閾値以上になれば、その文字から始まる $n > 1$ の n-gram を抽出対象に追加していく。例えば、文書の登録により「画」の出現が増えて閾値より大きくなったら、それ以降は「画」で始まる「画像」「画面」等の bi-gram も索引単位として抽出する。

この方法であれば、索引作成時に n を決定するわけではないので、複数の n に対する索引を使用する方法と比較して、索引サイズを小さくしたまま検索時間を高速化することが可能である。ただし、登録において長い n-gram を登録すべき状況になった場合、登録済みの文書に関するそれら n-gram の出現情報を索引に追加しなければならず、登録時間の増大を招くことになる。[6]

1.5.2 検索処理法

検索処理自体の工夫によって検索処理を高速化する方法には以下のものがある。

- 文書頻度の昇順による処理

  単語索引における AND 演算子処理の高速化手法として、索引単位の文書頻度の少ないものから順に処理を行うというものがある [Har90, Wit94]。これは、文書頻度の少ない索引単位同士は同じ文書に出現する可能性が少ないので、検索結果集合を素早く絞り込めるということに基づいている。

  このアイデアは n-gram 索引の検索処理にそのまま適用可能である。すなわち、検索文字列中の n-gram が全て出現する文書を特定する処理は AND 演算子処理そのものであり、このアイデアが適用できる。さらに、このアイデアは複数の n-gram が検索文字列を構成しているかを判定する位置検査にも応用可能であり、n-gram の出現位置を突き合わせる際にはその文書における出現回数の昇順に処理することで検索を高速化できる [Kik92]。

- 長い検索文字列処理における n-gram の省略

  長い検索文字列の検索処理で必要とされる n-gram を見直し、省略できるものは使用しないという提案である [Mat97b, Kaw96]。基本となるアイデアは、$n > 1$ の場合、隣接する 2 つの n-gram に跨がった位置にある n-gram の存在は、隣接する n-gram によって保証されるので、検索に用いる必要がないというものである。例えば、「携帯電話」から抽出される bi-gram 「携帯」「帯電」「電話」のうちの中央の「帯電」は省略可能である。使用する n-gram を削減することで、ファイルアクセスも減少し、検索の高速化が期待できる。

- n-gram 単位のスコア計算

[6] 追加する n-gram の情報をバックグラウンドで追加することとすれば、見た目の登録時間を短くすることは可能である。しかし、排他制御のために必ずしも十分な速度が得られるとは限らない、システムの構造は複雑になる等の問題がある。

ランキング検索では検索文字列の頻度情報に基づいて文書スコアを計算するが、そのためには位置検査が必要である。特に、ある文書における検索文字列の出現回数を求めるためには検索文字列の文書中の全ての出現位置を求める必要があり、ブーリアン検索時よりも処理コストが増大する。この問題を回避するため、検索文字列を構成するn-gram単位に文書スコアを計算する方式が提案されている[Aka97, Oga97b, Oga97c]。この方式であれば位置検査が全く不要となるため、検索速度の向上が期待できる。

1.5.3 転置ファイルの物理編成法

転置ファイルは索引小型化のために圧縮するのが一般的である（詳細は2.4.2節参照）。圧縮は索引小型化だけでなく、検索の高速化にも貢献する。すなわち、圧縮したことによりファイルから読み出したデータを伸長する必要が生じるものの、読み出すデータ量自体は減少する。現在の計算機環境ではCPUと比較してディスクアクセスははるかに遅いため、アクセス減少の効果の方が大きく、検索速度が向上する。

しかし、CPUが高速であるといっても伸長処理は検索を遅くする要因となるため、圧縮法の改良[Lin93, Mof96a, Wit94]とともに圧縮データを格納する物理編成法も高速化には重要である。以下では、物理編成法に関する従来の提案をまとめる。

- 位置情報の圧縮長の記録

  2.2節で詳しく説明するように、検索処理では文書内出現位置は常に必要となるわけではない。したがって、単純に文書ID・文書内頻度・文書内出現位置を圧縮して格納した場合、検索処理には不要な文書内出現位置であっても次の文書IDを得るために伸長しなければない。この問題に対しては、各文書に対応する文書内出現位置の圧縮長を記録すればよい[Mat97a]。この場合、文書内出現位置が不要な場合には圧縮長だけ伸長位置を進めることで次の文書の情報に進むことができ、無駄な伸長処理を省略できる。

- 圧縮情報の構造化

  最も単純な転置リストでは図2.13に示すように、文書ID・文書内頻度・文書内出現位置が連続して圧縮・格納される。しかし、検索アルゴリズムにあわせて転置リストを圧縮する際に適切に構造化すると、不要な伸長処理やディスクアクセスを無くすことが可能となる。これまで2つの方法が主に単語索引向けに提案されている。

  最初の方法は、文書ID・文書内頻度をブロック化するというものである[Anh98, Mof94b, Mof95]。単純な圧縮では、文書IDは前の値との差分をとっているため、途中の値を得るためにも先頭から順次伸長する必要がある。一方、ブロック化しそのブロックの先頭の文書IDは前の値との差分を取らないこととすると、所望の文書IDが出現しているかを検索する際、ブロックの先頭の文書IDを使ってその文書が含まれるブロックを判断した上で、ブロック内の値をその先頭から順に伸長すれば良くなり、検索を高速化できる。

  もう一つの方法は、転置リストを文書ID・文書内頻度を記録するディスク上での物理的な領域を文書内出現位置の領域と分離するというものである[Bro95b]。これは、単語索引においてランキング検索を行う場合、通常は文書内出現位置は不要であること

に基づいている。すなわち、文書内出現位置を異なる領域に記録することで、検索時にはその領域へのディスクアクセスと伸長処理を省略でき、検索を高速化できる。

1.6 研究の目的と概要

本論文では、n-gram 索引の検索処理の高速化について研究を行う。検索速度・機能の高さから索引形式は転置ファイルに限定する。

高速化の基本原理は、n-gram 索引の検索処理において発生する位置検査処理を可能な限り省略するというものである。従来の検索高速化の研究との関連を整理すると、従来の高速化が n-gram 抽出法に集中していたのに対し、本研究では検索処理法・転置ファイルの物理編成法の観点から検索の高速化することに特徴がある。

位置検査省略に着目したのは、日本語では造語力の高さから長い複合語がいくらでも生成されるため、n-gram 抽出法を工夫するだけでは n-gram 索引における検索速度低下の原因である位置検査を完全に排除することが不可能だからである。実際、日本語に n-gram 索引を適用する場合には索引サイズなどの点から n は 2 であることが一般的であり、多くてもせいぜい 3 であるのに対し、検索文字列の平均の長さはそれよりも大きく、カタカナやアルファベットで記述される外来語は長さが 10 文字以上となることも珍しくない。たとえ n-gram 抽出法を工夫することで実効的な n を大きくしても、長い検索文字列の処理が不要になることはあり得ない。

なお、短い検索文字列は長い検索文字列よりも n による影響が大きい。しかし、複数の n の組み合わせ、あるいは動的に n を変更する方法により、短い検索文字列としての処理を不要とすることは容易であるため、本論文では短い検索文字列の処理高速化は特に取り上げない。

本研究では検索処理法の改良により位置検査省略を実現し、さらに改良処理法向きの物理編成法を提案する。以下、それぞれの概要を説明する。

- 検索処理法の改良

  位置検査省略というアイデアは 1.5.2 節で紹介した長い検索文字列処理における n-gram の省略と基本的には同じである。しかし、従来の位置検査の省略による高速化が、単一検索文字列の場合のみを極めて限定的に扱っていたのに対し、本研究では以下の 3 点について拡張する。

  - 省略する n-gram を動的に選択するように拡張する

    従来は静的に省略する n-gram を選択していた。これに対し本研究では、省略可能な n-gram の組合せが複数になる場合に、処理コストが最小になるものを検索文字列・索引の状況に応じて動的に選択することで処理を効率化する。また、位置検査すべき文書を特定する場合と位置検査する場合の処理特性の違いに着目して、n-gram の選択方法をフェーズに応じて独立に調整することで一層の高速化をはかる。

  - AND・OR・ANDNOT の論理演算子に拡張する

    従来は単一検索文字列の処理のみを高速化の対象にしており、論理演算子処理を対象とする研究はなかった。これに対し本研究では、論理演算子の処理アルゴリ

ズムを複数の検索文字列の論理関係を考慮することで不要な位置検査を行なわないように改良する。AND, OR, ANDNOT の 3 種類の論理演算子について、この考えに基づく改良アルゴリズムを提示し、演算子が入れ子になった複合条件にもこの拡張が適用できることを示す。

- ランキング検索に拡張する
  ランキング検索の文書スコア計算には文書頻度・文書内頻度の 2 種類の頻度情報が必要であるが、両頻度を求めるたびごとに位置検査が発生するため検索コストが増大する。従来研究として n-gram の頻度情報に基づいてスコア計算するスコア合成法が提案されているが、検索精度の低下という問題があった。これに対し、検索精度を低下させることがないように検索文字列の頻度情報に基づいてスコア計算しつつ、頻度情報を求める際に必要な位置検査を可能な限り削減する 2 つの高速化手法を提案する。1 つ目の順序入れ替え法は、検索文字列が出現する文書を特定すると同時に文書内頻度も求めることで、従来必要であった文書頻度を単独で求める処理を省略する。もう 1 つの頻度推定法は、検索文字列を構成する n-gram の頻度情報から文書頻度および文書内頻度を近似的に求めることで位置検査を省略する。これら 2 つの高速化手法は組み合わせ可能であり相乗効果が得られることも示す。

- 転置ファイルの物理編成法の改良
  単語索引におけるランキング検索向きの物理編成法が従来検討されてきており、そうした従来手法は n-gram 索引に対してもある程度は有効である。しかし、n-gram 索引では位置検査が頻繁に行われるので、位置検査省略という高速化の方針と整合を取る必要性がある。本研究では、位置情報の圧縮長の記録と圧縮情報の構造化という手法を組み合わせによる、高速検索手法に適した物理編成法を提案する。

1.7 本論文の構成

次の 2 章では転置ファイル形式の n-gram 索引の基本概念について説明する。まず、登録、ブーリアン検索、ランキング検索といった n-gram 索引の基本動作について述べ、つぎに転置ファイルの基本構成と圧縮方法を紹介する。

3 章ではブーリアン検索処理の高速化について述べる。位置検査省略という基本原理を説明した後、単一文字列に対して位置検査省略を達成する複数の手法を提案する。さらに、単一文字列に対する手法を AND、OR、ANDNOT という論理演算子に適用する方法を提案する。4 章ではブーリアン検索高速化手法を評価する。評価には新聞記事 8 年分、約 85 万件、約 750MB を使用する。

5 章ではランキング検索処理の高速化について述べる。既存のランキング検索手法の問題点を説明した後、位置検査省略のための 2 つの手法を提案する。6 章ではランキング検索高速化手法を評価する。ランキング検索の評価では検索精度・検索速度の両面からの評価が不可欠であり、評価にはランキング検索用のテストコレクションである NTCIR-1（論文要旨約 33 万件、約 270MB）を使用する。さらに、単語索引との性能比較も行う。

7 章では高速化を実現するための転置ファイルの物理編成法ついて述べる。位置検査省略に基づく高速化手法に適応した物理編成法の簡単な評価も行う。8 章では提案手法を組み込

んだシステムについて説明し、社内特許システムのデータを用いて他の検索システムと性能比較した結果を報告する。

最後の９章で結論を述べる。

第2章　n-gram索引の基本概念

この章では、n-gram索引の基本概念を説明する。n-gram索引への文書の登録処理を概説した後、ブーリアン検索およびランキング検索について述べる。さらに、n-gram索引を実現するファイル形式である転置ファイルについても説明する。

2.1　登録処理

文書登録時には対象文書テキストの先頭から順に全てのn-gramを抽出する。$n > 1$の場合、隣り合うn-gramには重なりが生じるが、重なるものがあっても全てを抽出する。テキストの末尾部分ではn文字未満のn-gramも抽出する[Yok97]。これは、文書の末尾部分にあるn文字未満の単語を効率的に検索するためで、このように末尾部分を特別扱いすることを以下では末尾処理と呼ぶ。

以下、対象文書テキストTのi文字目からj文字目までの部分文字列をT_j^iと書くこととする。テキストがm $(m \geq n)$文字であるとき抽出されるのは、長さnのn-gramが$\{T_{i+n-1}^i\}(1 \leq i \leq m-n+1)$の$m-n+1$個と、末尾処理用の長さ$l$ $(1 \leq l < n)$の$\{T_m^{m-1+l}\}$の$n-1$個、合計m個である。テキスト長が$m(< n)$の場合には末尾処理だけが行われ、抽出されるn-gram数はやはりm個となる。

抽出したn-gramはその出現位置（先頭からの文字数とする）とともに、検索のために転置ファイル（詳しくは2.4節で述べる）に記録される。例えば、bi-gram（$n = 2$のn-gramをbi-gramと呼ぶ）索引においては、「携帯した携帯電話で電話した」というテキストは図2.1のように処理される。まず、テキストからn-gramが出現位置と組にして抽出する。ここで、最後に「た」を1文字で抽出しているのが末尾処理である。次に、抽出した組をn-gramごとに出現位置をまとめ、表記順に並べ直す（この処理を**転置 (inversion)** と呼ぶ）。その際、文書を識別するための文書ID、およびn-gramのその文書における出現回数も記録する。文書IDは文書を識別するための整数で、登録順に単調増加する値が付与されるものとする。図2.1では文書IDを1としている。

2.2　ブーリアン検索処理

ブーリアン検索では、検索対象文書中の検索文字列を含む文書を探し出す。n-gram索引では検索文字列の長さ（以下m文字とする）に応じて処理方法が異なるので、それぞれの場合について説明する。

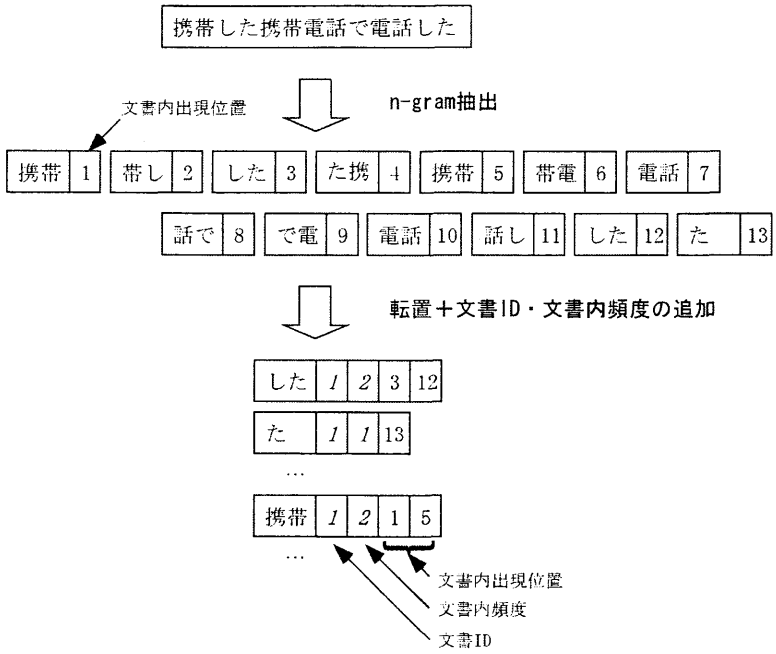


図 2.1: 登録処理の概要

```
void GramNode::booleanRetrieve(BooleanResult& result)
{
  int docId = 1;
  // 次の該当文書 ID の決定
  while ((docId = findNext(docId)) != maxId) {
    // 該当文書を結果に追加
    result.push(docId);
    ++docId;
  }
  // これ以上該当文書がないので、検索終了
}
```

図 2.2: 検索文字列長が n に等しい場合のブーリアン検索処理

2.2.1 検索文字列長が n に等しい場合

$m = n$ の場合に相当する。検索文字列と等しい n-gram が索引に登録されていれば、その n-gram が出現していた文書が検索結果となる。この場合、検索処理において n-gram の文書内出現位置を使用する必要はない。検索文字列と等しい n-gram が索引に登録されていなければ、該当文書なしが検索結果となる。

処理アルゴリズムを C++言語風に記述したものが図 2.2 である。`GramNode` は n-gram と等しい長さの検索文字列を表現するクラスである（`GramNode` を含めてこれ以降の説明に使用されるクラスは付録 B で説明する）。このアルゴリズムにおいて呼び出される`findNext` は引数で与えられた文書 ID 以上で最小である該当文書を返す関数であり、このクラスのメン

```
void OrNode::booleanRetrieve(BooleanResult& result)
{
  // 最初の子ノードについて検索結果を求める
  child[0].booleanRetrieve(result);
  // 2番目以降の子ノードを処理する
  for (int i = 1; i < child.size(); ++i) {
    int docId = 1;
    while (1) {
      // 該当文書の文書 ID の決定
      docId = child[i].findNext(docId);
      if (docId == maxId) {
        // これ以上候補文書がないので、検索終了
        break;
      }
      if (result.isFound(docId) == false) {
        // これまでの検索結果に含まれていないので、検索結果に追加
        result.push(docId);
      }
      ++docId;
    }
  }
}
```

図 2.3: 検索文字列長が n より小さい場合のブーリアン検索処理

バーである転置リスト `invertedList` によってその機能が提供される。転置リストとは、詳しくは 2.4 節で説明するが、転置ファイルにおいて各 n-gram の出現情報を管理するものである。`BooleanResult` は検索結果を格納するもので、文書 ID の表現に用いられる `int` の配列（ここでは、STL（Standard Template Library）[Tsu01] のベクトルを用いた `vector<int>` のサブクラス）である。また、`maxId` は登録済みの全ての文書 ID より大きな値であり、転置リストの `findNext` 関数は該当文書がなくなった場合にこの値を返す。

2.2.2 検索文字列長が n より小さい場合

$m < n$ の場合に相当する。このときは、先頭 m 文字が検索文字列に一致する n-gram のいずれかを含む文書が検索文字列を含む文書となる[1]。したがって、そのような n-gram の全てを OR 演算子で結合したものを検索条件として、検索を行なう。例えば、検索文字列が「話」であれば #or(話, 話あ, $\cdots$, 話遥) という条件で検索する。なお、$m = n$ の場合と同様、n-gram の文書内出現位置を使用する必要はない。

文字の異なり数を C と書くと、先頭 m 文字が一致する n-gram は $(C^{n-m+1} - 1)/(C - 1)$ 種類ある。実際には登録文書のいずれにも現れていない n-gram があるのでこの値以下となるが、展開される n-gram 数は $O(C^{n-m})$ になると考えられる。

[1] 先頭の m 文字とできるのは登録時に文書末尾については n 文字未満の n-gram を抽出する末尾処理を行っているからである。末尾処理を行わなくとも、中間あるいは末尾に検索文字列を含む n-gram で展開すれば正しい結果を得ることは可能である。しかし、展開数が多くなること、先頭が一致する n-gram は B 木等の探索構造により簡単に集められるのに対し中間・末尾が一致するものを集めるのは面倒であることが問題となる。

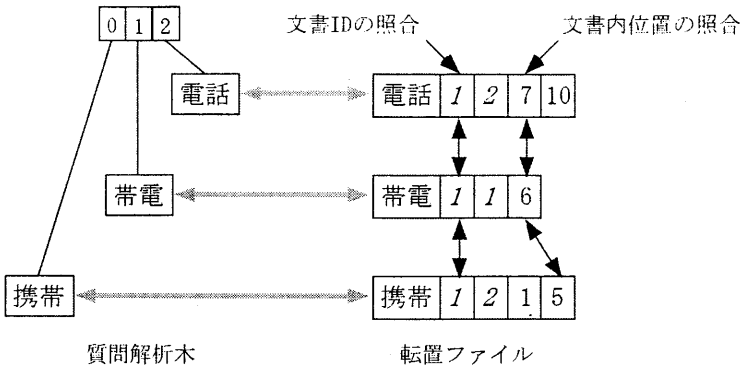


図 2.4: 検索文字列長が n より大きい場合の検索処理の概要

処理アルゴリズムは OR 演算子と同じである。短い検索文字列は`ShortTermNode`クラスによって処理されるが、付録 B にあるように`ShortTermNode`は`OrNode`のサブクラスであって、ブーリアン検索では`OrNode`のブーリアン検索関数がそのまま使用される。`OrNode`のブーリアン検索については 3.4.2 節で詳しく説明するが、図 2.3 にブーリアン検索関数を示す。まず、先頭のノードに対し検索を実施して先頭ノードの検索結果を得、それ以降のノードに対しては、文書 ID 順に該当文書を得てそれがそれ以前の検索結果に含まれていない場合には検索結果に追加するという処理を、順次実行する。ここで `child` は OR で結合される n-gram に対応するノード（前節で示した`GramNode`）のベクタ、`isFound` 関数は`docId`が検索結果に含まれていれば真（`true`）を返す関数である。

2.2.3 検索文字列長が n より大きい場合

$m > n$ の場合に相当する。この場合には、検文字列語から長さ n の n-gram のみを抽出する。登録時とは異なり末尾部分の n 未満の n-gram は抽出する必要はないので、抽出される n-gram 数は $m-n+1$ 個となる。例えば、「携帯電話」からは「携帯」「帯電」「電話」を抽出する。

検索文字列を含む文書は抽出される全ての n-gram を必ず含んでいる。しかし、全ての n-gram を含む文書であっても、それらが無秩序に分散して出現しているのでは検索文字列を含むことにならない。例えば、「携帯式電話機の帯電」を含む文書は「携帯電話」から抽出される n-gram を全て含んでいるが、「携帯電話」自体は含んでいない。このような過剰検索（検索ノイズ）を取り除くため、全ての n-gram を含む文書において n-gram が連続した位置に出現し、検索文字列を構成しているかを検査する必要がある。抽出した n-gram の出現位置を $\{i_1, i_2, \cdots, i_{m-n+1}\}$ とした場合、$i_1 = i_2 - 1 = \cdots = i_m - m + n$ の関係を満たしていることを検査し、検索文字列の存在が確認できる。複数の n-gram の出現位置の組み合わせが存在する場合、上記関係を満たすものが 1 つでも存在することが確認できれば良い。

検索の概要（図 2.1 の索引に対し「携帯電話」を検索する様子）を図 2.4 に示す。「携帯電話」を含む文書は、この検索文字列から抽出される「携帯」「帯電」「電話」が全て出現しており、さらに「携帯」に対し「帯電」は 1 文字、「電話」は 2 文字離れた位置に出現しているものである。文書内での位置検査のためには各 n-gram の相対位置を記憶しておく必要があり、図 2.4 の左側に示した検索処理のための内部構造（以下、解析木と呼ぶ）においては

```
void LongTermNode::booleanRetrieve(BooleanResult& result)
{
  int docId = 1;
  while (1) {
    // 次の候補文書の文書 ID の決定
    docId = findNextCandidate(docId);
    if (docId == maxId) {
      // これ以上候補文書がないので、検索終了
      break;
    }
    // 候補文書の検査
    if (checkCandidate(docId) == true) {
      // 検査結果が真ならば結果に追加
      result.push(docId);
    }
    ++docId;
  }
}
```

図 2.5: 検索文字列長が n より大きい場合のブーリアン検索処理

検索文字列に対応するノード（図左上）が記憶している（「携帯」に対しては先頭であることがわかるように値 0 を割り当てている）。この例では、ID=1 の文書には全ての n-gram が出現しているので候補文書と判断され、さらに位置検査が行われる。位置検査の結果、「携帯」の 2 番目の出現位置 5、「帯電」の出現位置 6、「電話」の 1 番目の出現位置 7 の組み合わせは $5 = 6 - 1 = 7 - 2$ という出現位置の制約条件を満たしており、この文書には「携帯電話」が出現していると判断できる。

以上の議論から、処理手順は検索文字列を構成する n-gram を全て含む文書を特定する候補文書決定と、候補文書において n-gram が連続した位置に出現しているかを確認する位置検査の 2 つのステップから構成可能である [Kik92]。ただし、実際には、候補絞込みで全ての候補文書を特定した後、各候補文書において位置照合を行うのではなく、候補文書が特定されるごとにその候補文書に関する位置照合を行うという手順をとる[2]。

C++風に記述すると図 2.5 のようになる。ここで、`findNextCandidate` は候補文書決定、`checkCandidate` は位置検査を行う関数である。候補文書決定は抽出 n-gram の AND 検索に相当し、付録 B にあるように `LongTermNode` は `AndNode` のサブクラスであるため、図 2.6 のように `AndNode` の `findNext`（詳細は 3.4.1 節）を呼び出すだけでよい。一方、位置検査関数 `checkCandidate` は図 2.7 の通りであり、検索文字列の出現位置検索関数 `findNextLocation`（図 2.8）を呼び出して有効な出現位置が返るかを検査する。`child` は検索文字列から抽出される n-gram ノードのベクタである。`LongTermNode::findNextLocation` から呼び出される `findNextLocation` は `GramNode` クラスのものであり、第 1 引数の文書 ID で第 2 引数で与えられた値以上で最小である出現位置を返す関数であり、転置リストによって実現される。

[2] このように複数の索引単位を文書 ID の順に処理する手順を文書順（document-order）と呼ぶ [Mof94a]。これに対し、検索文字列が n より短い場合のように、複数の索引単位を逐次的に処理する手順は索引単位順（term-order）と呼ぶ [Tur95, Anh98]。

```
int LongTermNode::findNextCandidate(int currentId)
{
  return AndNode::findNext(currentId);
}
```

図 2.6: 候補文書の決定アルゴリズム（findNextCandidate 関数）

```
bool LongTermNode::checkCandidate(int currentId)
{
  if (findNextLocation(currentId, 1) == maxLoc) {
    return false;
  }
  return true;
}
```

図 2.7: 候補文書の検査アルゴリズム（checkCandidate 関数）

```
int LongTermNode::findNextLocation(int currentId, int currentLoc)
{
retry:
  // 先頭の子ノードの候補出現位置を見つける
  int nextLoc
    = child[0].findNextLocation(currentId, currentLoc + dist[0]);
  if (nextLoc == maxLoc) {
    // 該当する出現位置がないので終了
    return maxLoc;
  }
  // 残りの子ノードについて検査する
  for (int i = 1; i < child.size(); ++i) {
    nextLoc
      = child[i].findNextLocation(currentId, currentLoc + dist[i]);
    if (nextLoc == maxLoc) {
      // 該当する出現位置がないので終了
      return maxLoc;
    }
    if (nextLoc > currentLoc + dist[i]) {
      // 該当する出現位置がない
      // つぎの出現位置について繰り返し処理をする
      currentLoc = nextLoc - dist[i];
      goto retry;
    }
  }
  return currentLoc;
}
```

図 2.8: 検索文字列の出現位置の検索アルゴリズム（findNextLocation 関数）

なお、図 2.8 の`dist`が n-gram 間の相対距離を記録するベクタである。`child`において子ノードが検索文字列での出現順にソートされていれば、`dist[i]=i`なので`dist`は不要である。しかし、検索高速化のために`child`は文書頻度（その n-gram を含む文書数）の昇順にソートするので、`dist`が必要となる。

2.3 ランキング検索処理

2.3.1 ランキング検索の概要

ランキング検索では、全ての検索対象文書について検索条件に対する適切さをあらわす文書スコアを計算し、文書を順序付ける。ランキング検索では、検索条件は自然言語文で与えられ、そのなかから抽出される適切な単語を用いて検索が実施される。すなわち、抽出された単語のいずれかを含む文書が検索され、文書ごとのスコアは単語の頻度情報に基づいて検索されるのが一般的である。ベクトル空間モデル [Sal75]・確率モデル [Rob77] 等の多くのモデルに共通して用いる頻度情報には以下のものがある [Umi88]。

- 文書頻度（document frequency）

  単語 t を含む文書数。以下 f_t と書く。

- 文書内頻度（in-document frequency）

  文書 d における単語 t の出現回数。以下 $f_{d,t}$ と書く。

- 検索要求内頻度（in-query frequency）

  検索要求文 q における単語 t の出現回数。以下 $f_{q,t}$ と書く。

文書のスコアは、弁別性（文書を識別する能力）と代表性（文書の内容を表現する能力）を考慮して計算される [Sal83b]。弁別性は、文書頻度が小さいほど大きいと考えられるので、文書頻度の逆数として計算される。一方、代表性は文書内頻度・検索要求内頻度に関して単調増加する項として計算される。

例えば、代表的な確率モデルの 1 つである Okapi モデルでは、文書 d における検索文字列 t のスコア $score(d,t)$ を以下の式で計算する [Rob94]。

$$score(d,t) = \left(k_t + \log\frac{N}{f_t}\right) \cdot \frac{f_{d,t}}{k_d + f_{d,t}} \cdot \frac{f_{q,t}}{k_q + f_{q,t}} \tag{2.1}$$

ここで、N は全文書数、k_t, k_d, d_q は各頻度情報の影響を調整するパラメータである。(2.1) 式では、最初の項が弁別性、2、3 番目の項が代表性に相当する。文書 d における検索要求文 q のスコア $score(d,q)$ は、検索要求文中の検索文字列のスコアの合計として計算される。

$$score(d,q) = \sum_{t \in q} score(d,t) \tag{2.2}$$

同一モデルを採用しても、単語索引と n-gram 索引ではランキング検索結果に差ができることに注意が必要である。これは、ある検索語（文字列）に対する頻度が単語索引と n-gram 索引とでは異なるからである。単語索引の場合、登録文書における単語の出現が記録されており、文書頻度・文書内頻度は単語としての出現に関する値である。一方、n-gram 索引

```
void GramNode::rankingRetrieve(RankingResult& result)
{
  int df = getDF();
  int docId = 1;
  // 次の該当文書 ID の決定
  while ((docId = findNext(docId)) != maxId) {
    // 該当文書を結果に追加
    Entry entry;
    entry.docId = docId;
    entry.score = calculate(df, getTF(docId));
    result.push(entry);
    ++docId;
  }
}
```

図 2.9: 検索文字列長が n に等しい場合のランキング検索処理

では、単語であるか否かとは関係なく、n-gram の出現が記録されるため、文書頻度・文書内頻度は文字列としての出現に関する値となる。したがって、同一の検索語（文字列）に対する頻度情報は単語索引と n-gram 索引では異なる。例えば、「携帯電話」を含む文書には、「帯電」という文字列は出現しているが、これは単語として出現しているわけではないため、頻度差の原因になる。ただし、テストコレクションを利用した従来の実験では、検索方法が単語単位・文字列単位であるかが検索精度に有意な差を与えないことが示されている [Jon98, Oga97b, Oga99b]。したがって、以下では文字列単位のスコア計算を行うものとする。

頻度のカウント方法の違いを無視しても、n-gram 索引のランキング検索は複雑なものとなる。単語索引を用いた場合、索引中に記録されている単語ごとの頻度情報をそのままスコア計算に使用できるので、検索処理は単純かつ高速である。一方、n-gram 索引では、検索文字列の長さが n と異なる場合、複数の n-gram を用いて文書頻度・文書内頻度を計算しなければないからである [Aka97, Fuk97, Mat97a]。

以下、ランキング検索方法を検索文字列の長さに応じて説明する。

2.3.2 検索文字列長が n に等しい場合

検索文字列が n-gram である場合、出現頻度は索引に記録されている頻度情報をそのまま使用することが可能であり、検索処理は単語索引におけるランキングと全く同じである。

処理アルゴリズムを図 2.9 に示す。`RankingResult` はランキング検索結果を格納するもので、`docId` と `score` の 2 つのメンバからなる構造体 `Entry` のベクタである。ブーリアン検索の処理アルゴリズム（図 2.2）と似ているが、はじめに文書頻度 `df` を求めておく点、該当文書が見つかった際に文書内頻度を求めた上でスコア計算する点が異なっている。文書頻度・文書内頻度は転置リストクラスが提供する `getDF`, `getTF` 関数よって得ているが、これは転置ファイルに記録されている値を読み出すだけで、簡単に実装できる。

なお、検索結果をスコア順にソートするのは呼び出し側が行なうこととして、図 2.9 には

```
void TermNode::rankingRetrieve(RankingResult& result)
{
  BooleanResult bresult;
  // ブーリアン検索する
  booleanRetrieve(bresult);
  // 文書頻度を求める
  int df = bresult.size();
  // 文書ごとの処理
  for (int i = 0; i < df; ++i) {
    Entry entry;
    entry.docId = bresult[i];
    // 文書内頻度を求める
    int tf = getTF(entry.docId);
    // スコア計算し、結果に追加
    entry.score = calculate(df, tf);
    result.push(entry);
  }
}
```

図 2.10: 検索文字列が n と異なる場合のランキング検索処理

含めていない。

2.3.3 検索文字列長が n より小さい場合

検索文字列が n と異なる場合、その検索文字列に関する文書頻度・文書内頻度は索引に記録されていないので、ブーリアン検索の手法を応用することで頻度を計算しなければならない。文書頻度は検索文字列を含む文書数であるので、検索文字列のブーリアン検索した結果件数として求められ、文書内頻度は対象文書における検索文字列の出現回数であるので、対象文書における検索文字列の全ての出現位置を数え上げることで求められる。

ただし、検索文字列が n より小さい場合、展開される n-gram において、ある出現位置が複数の n-gram によって重複して記録されることはない。したがって、展開した n-gram の文書内頻度の合計が検索文字列の文書頻度となり、検索文字列の出現位置をすべて求める必要はない。

検索アルゴリズム（次節の検索文字列長が n より大きい場合と共通）を図 2.10 に示す。まず、ブーリアン検索を行い、その結果件数を文書頻度`df`としてスコア計算用に記憶する。さらにブーリアン検索された文書ごとに文書内頻度を求めてスコアを計算する。文書内頻度を求める`getTF`関数は図 2.11 の通りである。

2.3.4 検索文字列長が n より大きい場合

この場合も索引から頻度情報を直接得ることができないという点で検索文字列が n より小さい場合と同様である。処理アルゴリズムも図 2.10 をそのまま適用できる。しかし、文書内頻度は n より小さい場合のように、n-gram の文書内頻度の合計として求めることはで

```
int ShortTermNode::getTF(int currentId)
{
  int tf = 0;
  for (int i = 0; i < child.size(); ++i) {
    tf += child[i].getTf(currentId);
  }
  return tf;
}
```

図 2.11: 検索文字列が n より小さい場合の文書内頻度の計算アルゴリズム

```
int LongTermNode::getTF(int currentId)
{
  int tf = 0;
  int loc = 1;
  while (1) {
    // 次の検索文字列の出現位置を求める
    loc = findNextLocation(currentId, loc);
    if (loc == maxLoc) {
      // 検索文字列が出現していないので終了
      break;
    }
    ++tf;
    ++loc;
  }
  return tf;
}
```

図 2.12: 検索文字列が n より大きい場合の文書内頻度の計算アルゴリズム

きず、図 2.12 のように検索文字列の出現回数を数え上げる必要がある [Fuk97, Kik92]。なお、図 2.12 中の`findNextLocation`関数は図 2.8 で定義したものである。

2.3.5 自然文検索の処理方法

ランキング検索では、ユーザの検索要求はわれわれが日常使用する自然言語で記述された文あるいは文章とすることが多い [Fra92, Sal83b, Wit94]。以下では自然言語で検索要求が与えらる検索を自然文検索と呼ぶ。

自然文検索では検索要求文をそのまま検索文字列として扱うことも可能ではあるが、検索要求文を含む文書だけを検索することとなるため、ユーザの要求を満足させることはできない。英語など単語索引を前提とする場合、検索要求文から検索にふさわしい単語を選択し、それらのいずれかを含む文書を特定した上でランキングする。日本語を対象とする場合、単語単位の方法を実現するには検索要求文を単語に分割しなければならず、単語単位の索引を行う場合と同様に形態素解析が必要となる。形態素解析後は、名詞類を選択した上で、不要

語辞書により検索に不適格な単語を排除した結果を検索語とする。

n-gram 索引に対しては、単語索引の場合と同様に適切な単語を選択する方式と、検索要求文から検索向きの n-gram を選択する方法の 2 種類がある [Kan98]。両者の特徴は以下のようにまとめられる。

- 単語を選択する場合

  単語索引の場合と同様に検索にふさわしい単語を選択する。検索要求に関係する概念を表す単語を単位に検索することができるので、検索精度の点では有利と考えられる。しかし、検索用に選択される単語が n-gram と一致するとは限らないため、前述の n と異なる長さの検索を行う必要があり、処理効率の点では不利である。この方式では索引単位と検索単位が異なっているため、ハイブリッド方式と呼ばれることもある。

- n-gram を選択する場合

  検索要求文を分割して得られる n-gram を OR で結合した条件でランキング検索する。その際、不要語辞書と同様な不要 n-gram 辞書を用意し、検索に不適格と思われる n-gram を除去することもある。索引単位と検索単位が一致するので処理は高速である。しかし、品詞等の言語的な情報を用いることができないこと、カタカナ語等の長い単語が複数の n-gram に分割されること等から検索精度の点では不利と考えられる。

ハイブリッド方式は、形態素解析に伴う問題を回避するために研究されてきた n-gram 索引に対し、検索要求文からの単語選択のために形態素解析を組み合わせているため自己矛盾に思われるかもしれない。しかし、登録と検索の性質の違いから、検索処理において使用する分には形態素解析が大きな問題とはならない。

- 速度低下

  検索要求文は量が少ないため処理の絶対時間が小さく、検索時間全体に対する割合も低い。したがって、形態素解析による速度低下は問題とならない。

- 解析誤り

  n-gram 索引であれば、自然文の解析誤りでも文字列上の一致により何らかの検索結果が得られるので、単語索引＋単語検索のように解析誤りが検索漏れに直結しない。さらに、検索時であればユーザが対話的に誤りを修正することも可能である。したがって、解析誤りも索引時のように問題にはならない。

- 辞書更新

  検索はその場限りのものであるので、辞書更新時にも過去に遡って索引付けを行なう必要はない。

以上の理由から、ハイブリッド方式は広く研究されている [Kan98, Oga97a]。

2.4 転置ファイル

2.4.1 ファイル構造

転置ファイル (inverted file) は、索引単位ごとに索引単位の出現情報を記録するものである [Fra92, Sal83b, Wit94]。

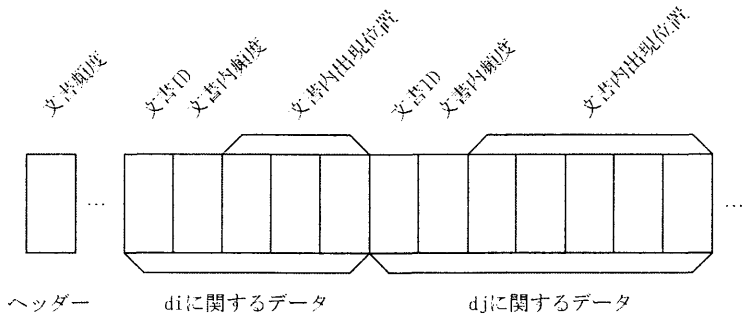


図 2.13: 転置リストの構造

索引単位ごとの出現情報を管理する構造を転置リスト(inverted list) と呼ぶ。転置リストは、出現情報として通常以下のものを記録する[3]。

- 文書頻度

  索引単位が出現した文書数。

- 文書 ID

  索引単位が出現した文書の識別子（文書 ID)。

- 文書内頻度

  索引単位が出現した文書におけるその索引単位の出現回数。

- 文書内出現位置

  索引単位が出現した文書におけるその索引単位の出現位置。文書ごとに文書内頻度に等しい個数だけ記録する。もっとも基本的な出現位置は文書の先頭からの文字数であるが、先頭からの文数などの他の情報も出現位置として記録することもある。

文書頻度は転置リストにつき 1 つだけ記録するが、文書 ID・文書内頻度・文書内出現位置は 3 組（以下転置リストエントリ）を構成し、索引単位が出現する文書ごとに記録する。転置リストの構造を図 2.13 に示す。

転置ファイルの論理的な構成は図 2.14 の通りであり、索引文字列に対応する転置リストを特定するための語彙ファイル（dictionary file）と、転置リストの集合体である位置ファイル（posting file）から構成される。

2.4.2 圧縮

単純に n-gram の出現文書ごとの出現位置を記録すると索引ファイルは著しく大きくなる。2.1 節で述べたように文書から抽出される n-gram の数はその文書に文字数に等しい。したがって、文書内出現位置を 4 バイトで記録すると、そのデータサイズだけでも全文書の文字数 ×4 バイトの大きさになる。ファイルサイズの増大はディスクスペースを大量消費するだ

[3]必要に応じてここにあげた以外の情報を記録することもある。

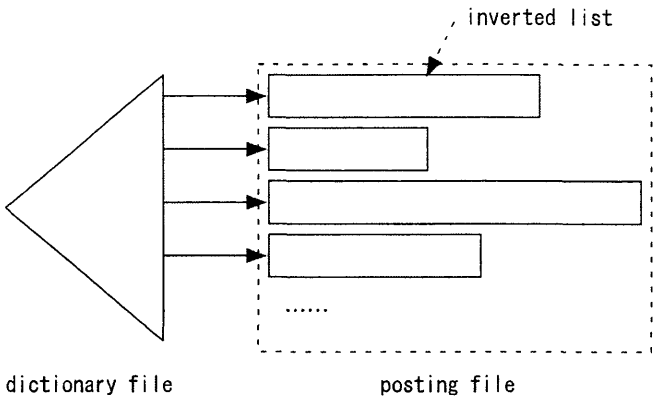


図 2.14: 転置ファイルの構造

けでなく、登録・検索時のディスクアクセスが増大して処理速度を低下させる点からも好ましくない。そこで、索引の小型化のために圧縮の導入が必須となる。

転置ファイルの圧縮では、アクセス単位である転置リストごとに圧縮しなければならない。通常、転置ファイルを構成する各データは以下のように圧縮される [Wit94, Zob92]。

- 文書 ID

  前回の出現との差分[4]を可変長符号化により圧縮する

- 出現回数

  そのままの値を可変長符号化により圧縮する

- 出現位置

  文書ごとに、前回の出現位置との差分を可変長符号化により圧縮する

符合化方式にはさまざまなものがあるが、索引に文書が追加することを考慮して、静的な圧縮方式を使用することが多い。圧縮アルゴリズムとしては、以下のようなものがある。

- unary 符号 [Wit94]

  ビット長で値を表現する。値 -1 個の '0'（接頭部; prefix）とそれに続く '1'（区切り子; delimiter）で構成される。

- γ 符号 [Eli75]

  接頭部、'1' である区切り子、接頭部と同じ長さの接尾部（suffix）で値を表現する。値 x に対し、接頭部は $\lfloor \log_2 x \rfloor$ 個の '0'、接尾部は $x - 2^{\lfloor \log_2 x \rfloor}$ となる。

- Exponential Golomb 符号 [Gol66]

  γ 符号を拡張したものであり、接尾部は接頭部+k の長さとなる。

これらの符合化により、1 ～ 10 までの値がどのように表現されるかを示したのが図 2.15 である。なお、この表においてカンマ「,」は接頭部、区切り子、接尾部を識別しやすいように書いたもので、実際に記録するものではない。

[4]文書 ID・出現位置の初期値を 1 とすれば、最初の文書 ID・出現位置に対しては仮想的な前の値として 0 を考えればよい。

値	unary	γ	Exponential Golomb ($k=2$)
1	,1	,1,	,1,00
2	0,1	0,1,0	,1,01
3	00,1	0,1,1	,1,10
4	000,1	00,1,00	,1,11
5	0000,1	00,1,01	0,1,000
6	00000,1	00,1,10	0,1,001
7	000000,1	00,1,11	0,1,010
8	0000000,1	000,1,000	0,1,011
9	00000000,1	000,1,001	0,1,100
10	000000000,1	000,1,010	0,1,101

図 2.15: さまざまな符合化手法による圧縮結果

文書 ID 差分、頻度、出現位置差分によって値の分布が異なるため、それぞれの分布に応じた符号化手法を用いる必要がある [Wit94]。さらに、文書の長さ、索引単位の選択方法（n-gram 索引であれば n の値）等にも左右される。

2.5 nと性能との関係

n-gram 索引では索引単位である n-gram の長さ n によって性能が大きく左右される。索引サイズ・登録速度・検索速度に対する n への影響をまとめると以下のようになる。

- 索引サイズ

  2.1 節に示したように、抽出する n-gram 数は n に依存せず、文書の長さに一致する。すなわち、索引に記録する出現位置情報自体は n に依存しない。しかし、索引には出現位置情報を n-gram と対応して記録する必要があるが、n-gram の異なり数は n に応じて大きくなる。文字の異なり数を C とした場合、n-gram の異なり数は C^n となる[5]。これら全てが登録文書に出現するわけではなく、索引に記録される n-gram の異なり数と索引サイズが比例するものではないが、索引サイズは n に応じて大きくなる。

- 登録速度

  登録処理は、文書からの n-gram 抽出と抽出結果のファイル書き込みから構成されるが、処理時間に影響するのは主にファイル書き込みである。索引（転置ファイル）では、n-gram ごとに出現情報をまとめて記録するので、登録文書に関する n-gram の出現情報の書き込みは抽出 n-gram の異なり数に応じて増大する。したがって、索引サイズ同様、登録時間は n に応じて増大する。

- 検索速度

[5]末尾処理を行うことにより n 以下の長さのすべての n-gram が抽出される可能性が生じるため、正確には $(C^{n+1}-1)/(C-1)$ である。

表 2.1: n-gram 索引に対する n の影響

	小 ←	n	→ 大
索引サイズ	○	—	×
登録速度	○	—	×
検索速度	△	—	×
n 文字未満	○	$O(C^{n-m})$	×
n 文字超	×	$O(m-n)$	○

検索処理は、検索文字列に関連する n-gram の読み出しとそれらの伸長（および付き合わせ）から構成されるが、処理時間に主に影響するのはファイル読み出しである。ファイル読み出しは処理すべき n-gram 数に依存し、その数は 2.2 節・2.3 節で示したように検索文字列長に応じて、検索文字列が n より短い場合には $O(C^{n-m})$ 個、長い場合には $O(m-n)$ 個となる。すなわち、検索語が長いか短いかによって n は正反対な影響を持っているが、オーダーを考慮すると n は比較的小さな値が望ましい。

前述の関係を表にしたものが表 2.1 である（m は検索文字列長）。索引サイズ・登録速度については n に対するオーダーを求めるのが困難であるので、空欄としてある。

実際に適用する n を決めるには、システムにおいて索引サイズ・登録速度・検索速度のいずれを重視するか、対象文書や検索語の分布によって最適な値が変わる。しかし、本研究が対象とする日本語においては文字の異なり数が大きいこと[6]、検索語の平均の長さが 4 文字程度であること等から通常 n は 1 ～ 3 の範囲であり、特に 2 とされることが最も多い [Kit98, Mic96]。

[6]通常使用される JIS X0208 であれば異なり数は約 7000 である。

第3章　ブーリアン検索処理の高速化

この章では、位置検査の省略に基づくブーリアン検索処理の高速化を提案する。まず、単一検索文字列の場合の高速化方法を述べ、つぎにその方法を AND・OR・ANDNOT の各論理演算子に展開する。最後に、AND・OR 演算子が混在した条件についても考察する。

3.1　従来の高速化手法とその問題点

n-gram 索引を用いた長い検索文字列の処理においては、検索文字列に含まれる n-gram をいかに用いるかが検索速度に影響する。これまでにも、使用する n-gram を最小限に抑えることで検索を高速化する手法が提案されている。本節では、その内容を説明し、問題点を明らかにする。

3.1.1　パスに基づく高速化とパスの単純選択法

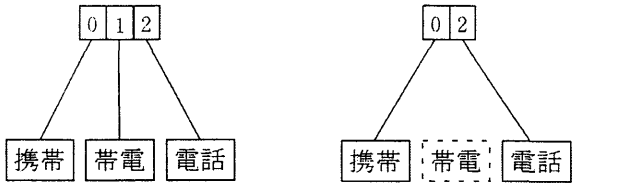
1.5.2 節でふれたように、長い検索文字列の処理では検索文字列から抽出される $m-n+1$ 個の n-gram の全てを使用する必要はない [Kaw96, Mat97b]。$n>1$ の場合、検索文字列から抽出される n-gram の隣り合うものには重なりがあり、隣接する 2 つの n-gram に跨がった位置にある n-gram の存在は、隣接する n-gram によって保証されるからである。例えば、「携帯電話」から抽出される bi-gram には「携帯」「帯電」「電話」の 3 つがあるが、「携帯」と「電話」が隣接していれば（2 文字離れて出現していれば）、「帯電」が「携帯」に対し 1 文字離れて出現していることがわかるので、「帯電」を検索処理に使用する必要はない。一方、検索文字列が出現していることを確認するには、検索文字列を構成する全ての文字が含まれるように n-gram を選ばなければならない。すなわち、検索処理では、検索文字列を被覆し、かつお互いの重複ができるだけ少ない n-gram を使用すれば十分といえる。以下、このような n-gram の組をパスと呼ぶ。

図 3.1 に「携帯電話」に対する解析木を示す。図 3.1 の (a) は検索文字列から抽出される n-gram を全て使用した解析木と比較して、パスに基づく (b) の解析木では「帯電」が使用されなくなっている。

パスを構成する n-gram 数は、検索文字列の長さ m に対して $\lceil m/n \rceil$ 個となる。パスを構成する n-gram のみを使用する方式は、処理すべき n-gram が少ないので、検索時間の短縮が期待できる。

最も単純なパスの選択方法は、先頭から重複しないように n-gram を選択し、そのようにして選択した n-gram だけで検索文字列を被覆できない場合には、最後尾の n-gram も追加的に選択するというものである [Aka96a, Kik92]。以下、この方法を単純選択法と呼ぶ。

単純選択法によって選択される n-gram を例示する。検索文字列が「携帯電話」であれば、「携帯」「帯電」「電話」の 3 つの n-gram の先頭から順に、重複がないように、「携帯」「電



(a) 全てのn-gramを用いた場合　(b) パスのn-gramを用いた場合

図 3.1: 「携帯電話」に対する解析木

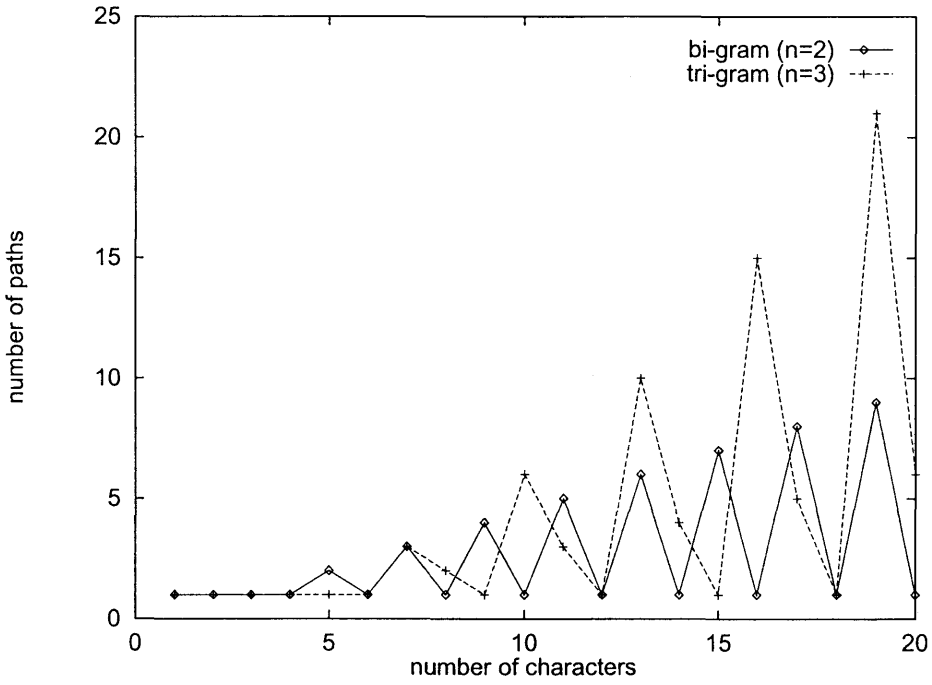


図 3.2: パスの個数

話」の 2 つを選択する。また、検索文字列が「自動車電話」であれば、「自動」「動車」「車電」「電話」の先頭から重複がないように「自動」「車電」を選択し、さらに最後尾の「電話」も選択する。

3.1.2 単純選択法の問題点

前述の単純選択法には問題がある。検索文字列が長い場合には、パスが複数個存在することがあり、処理コストに差が生じる。例えば、「自動車電話」に対するパスには、「自動」「車電」「電話」と「自動」「動車」「電話」の 2 種類がある。しかし、単純選択法はパスごとの検索コストの相違を考慮せず常に先頭から重複の内容に選択しているので、最適なパスが選ばれるとは限らない。つまり、必ずしも検索時間が最短となる最適な組み合わせが選択されないという問題があった。

検索文字列の長さが m であるときのパスの個数は、n-gram の長さ n との関係で定まる。詳細は付録 C に示すが、$m = (\alpha + 1)n - x, (\alpha > 1, 0 < x < n)$ の場合のパスの個数は

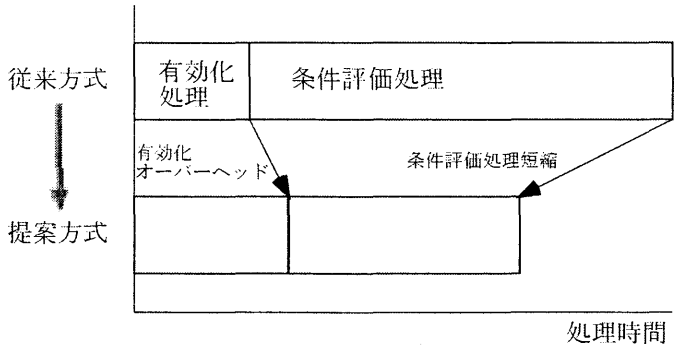


図 3.3: 検索高速化の摸式図

$\{\prod_{s=0}^{x-1}(\alpha+s)\}/x!$ となる。図 3.2 に $n=2,3$ について、1～20文字の検索文字列に対するパスの個数をプロットしたものである。この図からわかるように、検索文字列が長いほどパスの個数は増え、単純選択法では最適なパスが選ばれにくくなることを意味している。

もうひとつの問題点は、論理演算子の高速化である。パスに基づく高速化は単一検索文字列の高速化手法であり、論理演算子を含む条件の検索時間も個々の検索文字列の処理が高速化されれば短縮できる。しかし、論理演算子の処理手順自体に関しては、長い検索文字列の処理に伴う位置検査に対する配慮が全くなされていないため、個々の検索文字列の高速化以上の性能改善が見られないという問題があった。

3.2 本研究における高速化の考え方

本研究では、索引単位の n-gram よりも長い検索文字列を対象に高速化を試みる。これは、1.6 節でも触れたが、実用的な n の範囲では長い検索文字列が使用されることが多いため、長い検索文字列の高速化が平均的な検索性能向上には必要不可欠だからである。

単一検索文字列に対しては、次の 2 つの方針に基づいて検索処理を高速化する。

- 長い検索文字列の処理において省略できる n-gram は一意に定まるとは限らず、複数の組合せが考えられる。複数の組合せがある場合には、組合せごとの検索コストを比較して、最適なものを選択することで、検索の高速化をはかる。
- 検索処理は候補文書特定と位置検査という 2 種類の処理がある。省略 n-gram の選択にあたっては、それぞれの処理の特性を考慮して適切な組合せを独立に選択し、検索の高速化をはかる。

検索処理全体は n-gram を選択するための検索条件の有効化処理と実際に検索条件を満たす文書を決定する条件評価処理に分けられる[1]。上記方針によれば、適切な n-gram の選択がオーバヘッドとなるため有効化処理に要する時間（コスト）は従来よりも増大するが、条件評価処理の時間は大幅に削減できるので、検索時間全体も短縮できる。この様子を摸式的に表したものが図 3.3 である。

論理演算子に対しては、次の方針をとる。

[1] 長い検索文字列に対しては、条件評価処理がさらに候補文書特定と位置検査に分けられる。

- 論理演算子によって結合される複数の検索文字列の関係を考慮すると、検索文字列ごとの位置検査を省略可能な場合がある。そのような位置検査を実際に省略することで検索の高速化をはかる。

ここで、各論理演算子の位置検査を省略可能な場合とは以下に示す通りである。

- AND 演算子の場合、演算子で結合された全ての検索文字列について候補文書となる文書においてのみ、各検索文字列の位置検査が必要である。それ以外の個々の検索文字列の候補文書については位置検査を省略可能である。

- OR 演算子の場合、演算子で結合されたいずれかの検索文字列について位置検査を行い、検索結果に含まれること確定した文書について、残りの検索文字列の位置検査は省略可能である。

- ANDNOT 演算子の場合、第2オペランドの検索文字列が出現すると確定している文書について、第1オペランドの検索文字列の位置検査は省略可能である。あるいは、第1オペランドの検索文字列が出現しないと確定している文書について、第2オペランドの検索文字列の位置検査は省略可能である。

以下、3.3 節では単一検索文字列処理の高速化、3.4 節では論理演算子処理の高速化について詳しく説明する。

3.3 単一検索文字列処理の高速化

3.3.1 最小頻度法

複数のパスが存在する際に静的にパスを選択するという単純選択法の問題に対しては、複数のパスの中から処理コストが最小となるものを動的に選択すればよい。長さ m が $m \neq \alpha n$, (ただし $m > 2n$) である検索文字列に対してはパスが複数個存在するので、検索コストを考慮してパスを選択することで検索時間を短縮できる。

つぎに処理コストの求め方を考える。検索処理においては、索引要素の出現が独立であると仮定すれば、処理コストは検索文字列から抽出される索引単位の文書頻度の和にほぼ比例する [Kik92, Tur95]。パスの選択においては、パスの相対的なコスト比較ができれば十分なので、n-gram の文書頻度の和を処理コストの推定値として用いることとする。この方法によれば、文書頻度の和が最小となるパスを選択することになるので、以下では、このようにして求められるパスを最小頻度パス、この手法を最小頻度法と呼ぶ。

「自動車電話」に対する最小頻度法による解析木を、各 n-gram に対する文書頻度（図中では df）とともに図 3.4 に示す。ここでは、中間にくる「動車」「車電」のどちらを選択されるかが文書頻度によって決定される。(a) では df(動車) > df(車電) なので「車電」、(b) では df(動車) < df(車電) なので「動車」が選択されている。なお、選択された n-gram の処理順序は、単語索引に対する AND 演算子の場合と同様、文書頻度の小さい順とする。図 3.4 においては、(a)(c)(d) は全て同じ n-gram が選択されているが、それらの文書頻度の違いにより処理順序が異なることを示している。

続いて、最小頻度パスの求め方を考える。これ以降、m 文字の検索文字列を Q_m^1 と表すこととする。

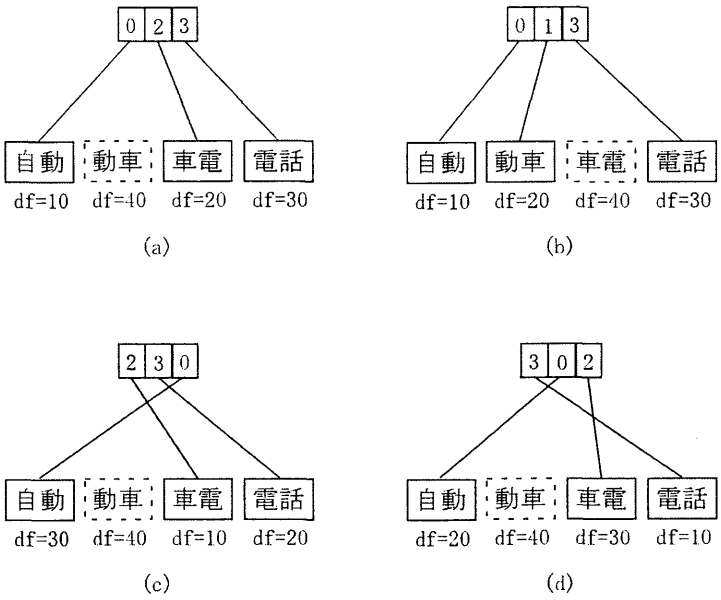


図 3.4: 最小頻度法による解析木

まず $m = \alpha n, (\alpha > 1)$ の場合を考える。検索文字列は重なりのない $\alpha(= m/n)$ 個の n-gram に分割できるので、パスは $(Q_n^1, Q_{2n}^{n+1}, \cdots, Q_{\alpha n}^{(\alpha-1)n+1})$ の一意に定まる。検索文字列の文書頻度の和の最小値（以下、最小頻度和）を $f_{min}(X)$ とすると、$f_{min}(X)$ は以下の式によって計算できる。

$$f_{min}(Q_{\alpha n}^1) = \sum_{s=1}^{\alpha} f(Q_{sn}^{(s-1)n+1}) \tag{3.1}$$

ここで、$f(X)$ は文字列 X の文書頻度を表す。

つぎに $m = (\alpha+1)n - 1, (\alpha > 1)$ の場合を考える[2]。最小頻度パスおよび最小頻度和は再帰的に求めることができる。検索文字列において、末尾部分を被覆するために末尾の n-gram $Q_{(\alpha+1)n-1}^{\alpha n}$ を選択しなければならず、残りの部分を被覆する先頭からの部分文字列には $Q_{\alpha n-1}^1, \cdots, Q_{(\alpha+1)n-2}^1$ の n 通りがある。ただし、先頭からの部分文字列のうち $Q_{\alpha n}^1$ より長いものはそれ自体のパスの構成 n-gram 数がもとの検索文字列と変わらないため、分割後の n-gram の組み合わせはパスとはならず、$Q_{\alpha n-1}^1, Q_{\alpha n}^1$ の 2 つだけが有効である。したがって、検索文字列が最終的にパスとなる、末尾 n-gram を含むような検索文字列の分割方法は $(Q_{\alpha n}^1, Q_{(\alpha+1)n-1}^{\alpha n})$ と $(Q_{\alpha n-1}^1, Q_{(\alpha+1)n-1}^{\alpha n})$ の 2 通りである。最小頻度和に関しては以下の関係式が成り立つ。

$$\begin{aligned} f_{min}(Q_{(\alpha+1)n-1}^1) = min(&f_{min}(Q_{\alpha n}^1) + f(Q_{(\alpha+1)n-1}^{\alpha n}), \\ &f_{min}(Q_{\alpha n-1}^1) + f(Q_{(\alpha+1)n-1}^{\alpha n})) \end{aligned} \tag{3.2}$$

この式の右辺第 1 式の第 1 項には式 (3.1) が適用でき、右辺第 2 式の第 1 項は $(\alpha - 1) > 1$ であれば再帰的に式 (3.2) を適用することができる。したがって、最小頻度パス・最小頻度和は式 (3.1)・式 (3.2) により簡単に求まることがわかる。これら式変形を繰り返すことで、以下の式が得られる。

[2] $\alpha = 1$ の場合には、上述のようには検索文字列を 2 つに分割できないため、パスは 1 つしか存在しない。

$$f_{min}(Q^1_{(\alpha+1)n-1}) = min(f(Q^1_n) + \cdots + f(Q^{(\alpha-1)n+1}_{\alpha n}) + f(Q^{\alpha n}_{(\alpha+1)n-1}), \quad (3.3)$$
$$\cdots,$$
$$f(Q^1_n) + f(Q^n_{2n-1}) + \cdots + f(Q^{\alpha n}_{(\alpha+1)n-1}))$$

右辺の各式はパスに対応しており、$\alpha+1$ 個の要素から構成される。各式において $\alpha+1$ 個の要素のうち隣り合う 1 組だけに重なりがあるので、式の個数（＝パスの個数）は α となる。右辺において最小値となるものが最小頻度パスであり、検索処理に使用する。

最後に最も一般的な $m = (\alpha+1)n - x, (\alpha > 1, 0 < x < n)$ の場合を考える。このとき、末尾 n-gram を除いた先頭からの部分文字列の選び方は $Q^1_{\alpha n}, \cdots, Q^1_{\alpha n-x}$ の $x+1$ 通りとなる。したがって、式 (3.2) は以下のように一般化できる。

$$f_{min}(Q^1_{(\alpha+1)n-x}) = min(f_{min}(Q^1_{\alpha n}) + f(Q^{\alpha n-x+1}_{(\alpha+1)n-x}), \quad (3.4)$$
$$\cdots,$$
$$f_{min}(Q^1_{\alpha n-x}) + f(Q^{\alpha n-x+1}_{(\alpha+1)n-x}))$$

この式 (3.4) と式 (3.2) から最小頻度パス・最小頻度和は求めることができる。先ほどと同様の式変形により以下の式が得られる。

$$f_{min}(Q^1_{(\alpha+1)n-x}) = min(f(Q^1_n) + \cdots + f(Q^{(\alpha-1)n+1}_{\alpha n}) + f(Q^{\alpha n-x+1}_{(\alpha+1)n-x}), \quad (3.5)$$
$$\cdots,$$
$$f(Q^1_n) + f(Q^{n-x+1}_{2n-x}) + \cdots + f(Q^{\alpha n-x+1}_{(\alpha+1)n-x}))$$

ここで、右辺の各式は $\alpha+1$ 個の要素から構成され、式の個数は $\{\prod_{s=0}^{x-1}(\alpha+s)\}/x!$ である（この式の導出は付録 C に示す）。

3.3.2 全 n-gram 法

条件評価ステップは候補文書特定と位置検査の 2 段階で構成されるが、前述の最小頻度法を含めた従来方式ではこれら両段階とも同じ n-gram 群を使用する。このことが、最小頻度法が全ての n-gram を使用する基本方式よりも必ずしも効率的にならないという問題を引き起こす。すなわち、最小頻度法では最小限の n-gram を用いているので、ある文書における位置検査のコストは削減されるが、候補文書数が基本方式よりも多くなってしまうため、検索全体のコストを減少できるとは限らないのである。なお、これはパスに基づく単純選択法にも共通な問題である。ただし、最小頻度法は文書頻度の和が最小となるパスを選択しているので候補文書数は必ず単純選択法の場合より小さいので、最小頻度法の方がこの問題の影響も小さい。

この問題を解決するには、候補文書特定と位置検査で使用する n-gram 群は一致させる必要はないことに着目し、位置検査で使用するのは最小頻度パスとしたまま、候補文書特定に使用する n-gram を増やせばよい。その結果、文書ごとの位置検査のコストを抑えたまま候補文書数を減らすことが可能で、検索全体のコストも削減できる。最も単純には、検索文字

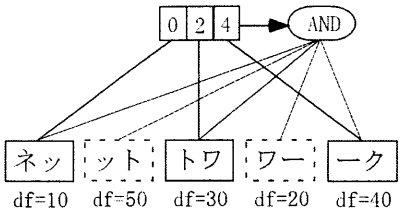


図 3.5: 全 n-gram 法による解析木

```
void LongTermNode::booleanRetrieve(BooleanResult& result)
{
  int docId = 1;
  while (1) {
    // 次の候補文書の文書 ID の決定
    docId = findNextCandidate(docId);
    if (docId == maxId) {
      // これ以上候補文書がないので、検索終了
      break;
    }
    // 候補文書の検査
    if (checkCandidate(docId) == true) {
      // 検査結果が真ならば結果に追加
      result.push(docId);
    }
    ++docId;
  }
}
```

図 3.6: 長い検索文字列の拡張アルゴリズム

列から抽出される全ての n-gram を使用すればよく、この方法 — 候補文書特定に検索文字列から抽出される全 n-gram、位置検査に最小頻度パスを用いる — を**全 n-gram 法**と呼ぶ。

全 n-gram 法では、候補文書特定と位置検査に異なる n-gram 群を用いるので、解析木の形もそれに合わせて修正する必要がある。すなわち、検索文字列から抽出される全 n-gram を結合させた AND 演算子を候補文書評価用のノードとして用意する。図 3.5 に検索文字列「ネットワーク」に対する全 n-gram 法の解析木を示す。中央上部の位置検査用ノードには最小頻度パスに属する「ネッ」「トワ」「ーク」が結合され、右側上部の候補文書評価用ノードには位置検査には使用しない「ット」「ワー」も結合される。

解析木の変更と同様、検索アルゴリズムも修正する必要がある。まず、`LongTermNode` に候補文書評価用ノードである`candidateNode` というメンバーを追加する。アルゴリズムは図 3.6 のように修正し、候補文書の特定には`findNextCandidate` 関数を用いることとする。`findNextCandidate` 関数は図 3.7 のように評価用ノード（`candidateNode`）に対して`findNext` 関数を呼び出す。前述のように候補文書評価用ノードは n-gram を集めた AND ノードであり、AND ノードの`findNext` 関数が呼び出される。AND ノードでは位置検査が起こり得ないので、3.4.1 節の図 3.10 のようになる。

```
int LongTermNode::findNextCandidate(int currentDid)
{
  return candidateNode->findNext(currentId);
}
```

図 3.7: `findNextCandidate` 関数

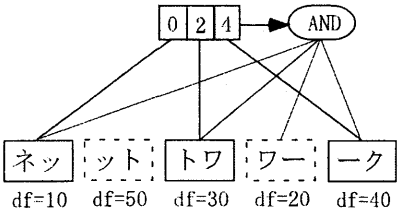


図 3.8: 選択 n-gram 法による解析木

3.3.3 選択 n-gram 法

全 n-gram 法では、候補文書特定に検索文字列から抽出される全 n-gram を用いる。しかし、候補文書特定のために使用する n-gram を増やすことにより、それら n-gram に対する転置リストの読み出し（ディスクアクセス）および伸長処理が増加するので、それらの増加分よりも候補文書数減少に伴うコスト削減が上回らないと検索は高速化できない。全 n-gram 法のように全ての n-gram を追加すると、登録済みの大多数の文書に出現するような候補文書を絞り込むのに有効でない n-gram が含まれることが多くなり、かえって検索時間が増大することもある [Oga98, Oga99a]。

この問題を解決するには、絞り込みに有効な n-gram のみを候補文書決定に追加すればよい。そこで、本研究では、前後に位置する位置検査に用いる n-gram よりも文書頻度の小さいものを絞り込みに有効と判断し、候補特定に用いることとした。この方法を**選択 n-gram 法**と呼ぶ。

図 3.8 に選択 n-gram 法の解析木を示す。図 3.5 の全 n-gram 法とは異なり、位置検査に使用しない「ット」「ワー」の中から隣接する n-gram よりも文書頻度の小さい「ワー」のみを候補文書評価用ノードに結合している。これは、「ット」の文書頻度はその前後にある「ネッ」「トワ」の文書頻度よりも大きいので除外されるのに対し、「ワー」の文書頻度はその前後にある「トワ」「ーク」の両者の文書頻度のよりも小さいからである。

なお、解析木自体の構造は全 n-gram 法と同じであるので、検索アルゴリズムは図 3.6 をそのまま使用することができる。

3.4 論理演算子処理の高速化

前節では単一検索文字列の検索処理の高速化を検討したが、実際の検索場面では、複数の検索文字列を AND, OR, ANDNOT 等の演算子で組み合わせた検索条件（以下、複合条件）が使用されることが多く、単一検索文字列だけでは十分ではない。しかし、筆者の知る

```
void AndNode::booleanRetrieve(BooleanResult& result)
{
  int docId = 1;
  while (1) {
    // 該当文書の文書 ID の決定
    docId = findNext(docId);
    if (docId == maxId) {
      // これ以上候補文書がないので、検索終了
      break;
    }
    result.push(docId);
    ++docId;
  }
}
```

図 3.9: AND 演算子の基本アルゴリズム

限り n-gram 索引における複合条件の高速化を扱った研究はない。もちろん、単一の検索文字列が高速化されれば複合条件も高速化されるが [Mat97b]、複合条件に合わせた最適化ができれば一層の高速化が達成できるので、論理演算子の処理自体を高速化する意義は大きい [Oga99c]。

以下、AND、OR、ANDNOT の 3 種類の論理演算処理の高速化および論理演算子が入れ子になった複合条件の扱いについて検討する。

3.4.1 AND 演算子

AND 演算子の基本処理方法には、全ての検索文字列が出現する文書を文書 ID の小さい順に見つける文書順（document-order）と検索文字列順にその文字列に対する検索結果を求め、前の検索文字列までの結果との AND を取る索引単位順（term-order）がある。両者を比較すると、中間結果を保持しておく必要のない文書順が効率的であることが知られている [Fra92, Wit94]。文書順による基本アルゴリズムを図 3.9 に示す。ここで`findNext`は`currentId`以上で最小の文書 ID を持つ全ての検索文字列が出現する文書を検索する関数である。この関数は図 3.10 に示す通り、先頭ノードについて該当文書（そのノードに対応する検索文字列を含む文書）を見つけ、その文書が残りのノードについても該当文書であるかを検査することで、AND としての該当文書を見つけ出す。図 3.10 において`child`は AND で結合される検索文字列に対応するノードのベクタであり、効率化のために`child`は文書頻度の小さい順にソートしておく。

基本アルゴリズムを n-gram 索引にそのまま適用することも可能である。しかし、AND のオペランドに長い検索文字列が含まれ、位置検査が生じる場合、基本アルゴリズムには以下のような問題がある。AND 演算子では、全ての検索文字列を含む文書が検索結果になるので、検索文書は少なくとも全ての検索文字列について候補文書でなければならない。したがって、全ての検索文字列について候補文書となっていない文書について位置検査を行う必要はない。しかし、基本アルゴリズムでは、`findNext`関数において検索文字列を含む文書

```
int AndNode::findNext(int currentId)
{
retry:
  // 先頭の子ノードの候補文書を見つける
  int nextId = child[0].findNext(currentId);
  if (nextId == maxId) {
    // 該当するIDがない。先頭でなければもう候補文書はない
    return maxId;
  }
  // 残りの子ノードについて検査する
  for (int i = 1; i < child.size(); ++i) {
    if (child[i].check(nextId) == false) {
      // 該当するIDがない。つぎのIDで繰り返し処理をする
      currentId = nextId + 1;
      goto retry;
    }
  }
  return currentId;
}
```

図 3.10: AND 演算子の`findNext`関数

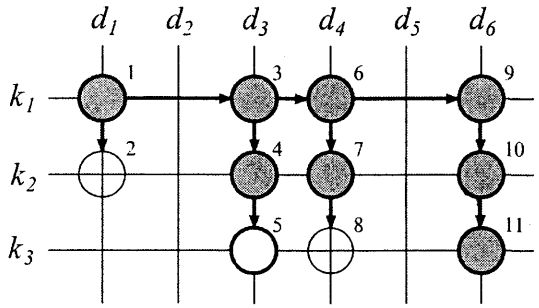


図 3.11: AND 演算子の基本アルゴリズムの動作例

を検索しているので、その途中で見つかる候補文書において、それが位置検査を行うべきものであるかにかかわりなく常に位置検査が行なわれる。すなわち、本来行わなくても良い位置検査を行うことがあるため、検索時間が増大することになる。

3 つの検索文字列から成る AND 演算子の基本アルゴリズムの動作例の図 3.11 を使って、この問題を説明する。図 3.11 で、d_i は文書、k_j は検索文字列、(d_i, k_j) の交点の白丸は d_i が k_j の候補文書であること、網掛けの丸は d_i が k_j の該当文書（実際に k_j を含む文書）であること、透明の丸（線が内部に見えるもの）は d_i が k_j の候補文書ですらないがアルゴリズムによって候補文書であるかが検査されることを示す。この例では、d_6 が検索結果となる。丸の右上の数字が処理順序を示しており、1 によって d_1 が k_1 を含むことが検索されるが、2 によって d_1 が k_2 を含まないことが確認されたので次の文書に進み、3 によって d_3 が k_1 を含むことが検索されるというように処理は進む。太線になっている丸（1,3,4,5,6,7,9,10,11）は処理中に位置検査が行われる文書・検索文字列の組み合わせを示している。しかし、d_1, d_4

```
void AndNode::booleanRetrieve(BooleanResult& result)
{
  int docId = 1;
  while (1) {
    // 候補文書の文書 ID の決定
    docId = findNextCandidate(docId);
    if (docId == maxId) {
      // これ以上候補文書がないので、検索終了
      break;
    }
    // 候補文書の検査
    if (checkCandidate(docId) == true) {
      // 検査結果が真ならば結果に追加
      result.push(docId);
    }
    ++docId;
  }
}
```

図 3.12: AND 演算子の拡張アルゴリズム

は全ての検索文字列について候補文書となっているわけではない（2,8 は透明である）ので、それら文書における残りのノードの位置にあたる 1,6,7 では位置検査が不要である。

不要な位置検査を排除するには、全ての検索文字列について候補文書である文書について位置検査を行うようにすればよい。すなわち、AND としての候補文書を検索する処理とその候補文書が AND としての該当文書かを検査する処理に分ければよい。改良したアルゴリズムは図 3.12 のようになる。このアルゴリズムで使用される`findNextCandidate` 関数・`checkCandidate` 関数を図 3.13・図 3.14 に示す。`findNextCandidate` 関数は`findNext` 関数と同様であるが、候補文書を検索するので、先頭ノードに対しては`findNextCandidate` 関数、残りのノードに対しては`isCandidate` 関数を呼び出す。もう一方の`checkCandidate` 関数は子ノードに対し`checkCandidate` 関数を呼び出している。なお、子ノードである`LongTermNode` クラスの`findNextCandidate` 関数・`checkCandidate` 関数は図 3.7・図 2.7 に示した通りであり、`isCandidate` 関数は図 3.15 のように n-gram を集めた AND ノードである候補文書評価用ノードに対して`check` 関数を呼び出せばよい。

改良アルゴリズムの動作例は図 3.16 の通りである。全ての検索文字列について候補文書となる文書でのみ位置検査を行うので、図 3.11 において不要な位置検査と指摘した 1,6,7 について位置検査が行われないことがわかる。全ての検索文字列について候補文書となる d_3, d_6 においては、候補文書かの検査と位置検査の 2 回のループが回るので、k_3 から k_1 に戻る矢印がついていることも相違点である。

なお、図 3.13 の`findNextCandidate` 関数では、各検索文字列ノードにある候補文書決定用ノードを間接的に呼ぶ出すことになるので非効率である。AND ノードにも候補文書決定用ノードを用意し、そこに各検索文字列ノードの候補文書決定用ノードの下に来る n-gram ノードを集めれば、検索処理を効率化できる。その様子を示したのが図 3.17 であり、(a) の状態から (b) のような候補文書決定用ノード（AND ノードの右にある AND ノード）が作

```
int AndNode::findNextCandidate(int currentDid)
{
retry:
  // 先頭の子ノードの候補文書を見つける
  int nextId = child[0].findNextCandidate(currentId);
  if (nextId == maxId) {
    // 該当するIDがない。先頭でなければもう候補文書はない
    return maxId;
  }
  // 残りの子ノードについて候補文書か検査する
  for (int i = 1; i < child.size(); ++i) {
    if (child[i].isCandidate(nextId) == false) {
      // 該当するIDがない。つぎのIDで繰り返し処理をする
      currentId = nextId + 1;
      goto retry;
    }
  }
  return nextId;
}
```

図 3.13: AND 演算子の`findNextCandidate`関数

```
bool AndNode::checkCandidate(int currentId)
{
  for (int i = 0; i < child.size(); ++i) {
    if (child[i].checkCandidate(currentId) == false) {
      // 該当するIDがない。ひとつでもなければもう該当文書ではない
      return false;
    }
  }
  return true;
}
```

図 3.14: AND 演算子の`checkCandidate`関数

```
bool LongTermNode::isCandidate(int documentId)
{
  return candidateNode->check(documentId);
}
```

図 3.15: `LongTermNode`の`isCandidate`関数

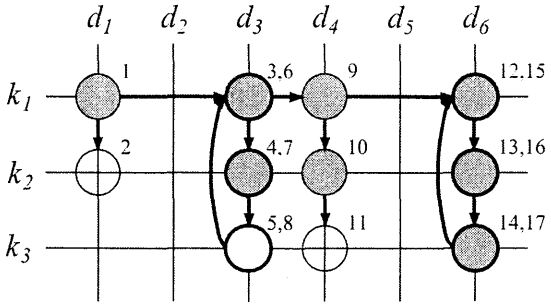
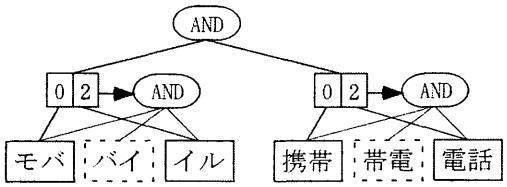
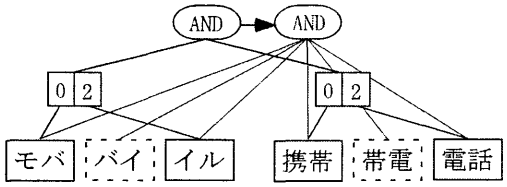


図 3.16: AND 演算子の改良アルゴリズムの動作例



(a) AND演算子に候補文書決定用ノードがない場合



(b) AND演算子に候補文書決定用ノードがある場合

図 3.17: AND 演算子の候補文書決定用ノードを含む解析木

成される。この方法は、重複する n-gram がある場合にその n-gram への呼び出しを単一化できること、AND ノードの候補文書決定用ノードにおいて文書頻度の少ない順に n-gram ノードをソートすることでより大域的な処理順序の最適化が行われることからも効率的と言える。この場合の`findNextCandidate`関数は長い検索文字列のもの（図 3.7）と同じく、`candidateNode`に対し`findNext`関数を呼び出せばよい。

最後に検索中に必要な記憶領域についても検討しておく。この方法によれば、従来は検索文字列にあった候補文書決定用ノードは不要となり、かわりに AND ノードに候補文書決定用ノードが増える。しかし、検索文字列の個数は 2 以上であるので、必要なデータ量は若干ではあるが削減される。

3.4.2 OR 演算子

OR 演算子では、AND 演算子とは異なり、文書順よりも検索単位順の方が効率的である。これは、OR 演算子を文書順で実現するためには、ある時点での全ての検索文字列に該当する文書のなかで最小の文書 ID を持つ文書を見つけなければならない。しかし、最小の文書を見つけるには、各検索文字列に対する最小文書 ID をヒープ構造などを用いて常に管理し

```
void OrNode::booleanRetrieve(BooleanResult& result)
{
  // 最初の子ノードについて検索結果を求める
  child[0].booleanRetrieve(result);
  // ２番目以降の子ノードを処理する
  for (int i = 1; i < child.size(); ++i) {
    int docId = 1;
    while (1) {
      // 該当文書の文書 ID の決定
      docId = child[i].findNext(docId);
      if (docId == maxId) {
        // これ以上候補文書がないので、検索終了
        break;
      }
      if (result.isFound(docId) == false) {
        // これまでの検索結果に含まれていないので、検索結果に追加
        result.push(docId);
      }
      ++docId;
    }
  }
}
```

図 3.18: OR 演算子の基本アルゴリズム

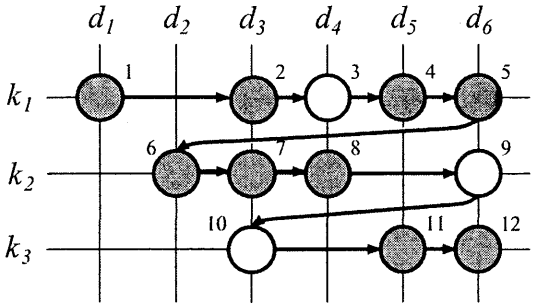


図 3.19: OR 演算子の基本アルゴリズムの動作例

ておく必要があるが、その処理の負荷が大きいからである [Anh98]。一方、索引単位順であれば、処理途中では常に 1 つの検索文字列だけが処理対象となるので、こうした問題は発生しない。

索引単位順によるアルゴリズムを図 3.18 に示す。まず、先頭のノードに対し検索を実施して、先頭ノードの検索結果を得る。それ以降のノードに対しては、文書 ID 順に該当文書を得、それがそれ以前の検索結果に含まれていない場合には検索結果に追加するという処理を、順次実行する。

しかし、基本アルゴリズムを n-gram 索引に適用した場合、AND 演算子の場合と同様に位置検査が無駄になるという問題が生じる。OR 条件では、どれか 1 つの検索文字列を含む

```
void OrNode::booleanRetrieve(BooleanResult& result)
{
  // 最初の子ノードについて検索結果を求める
  child[0].booleanRetrieve(result);
  // 2番目以降の子ノードを処理する
  for (int i = 1; i < child.size(); ++i) {
    int docId = 1;
    while (1) {
      // 該当文書の文書IDの決定
      docId = child[i].findNextCandidate(docId);
      if (nextDid == maxDid) {
        // これ以上候補文書がないので、検索終了
        break;
      }
      if (result.isFound(docId) == false &&
          child[i].checkCandidate(docId) == true) {
        // これまでの検索結果に含まれていないので、検索結果に追加
        result.push(docId);
      }
      ++docId;
    }
  }
}
```

図 3.20: OR 演算子の拡張アルゴリズム

文書は検索結果に含まれるので、すでに検索結果に含まれることが確定した文書について別の検索文字列が含まれるかを調べる必要はない。ところが基本アルゴリズムでは、2番目以降のノードに対してもfindNextを呼び出しているので、それ以前の検索文字列が出現している文書に対しても位置検査が発生し、検索時間を増大させる。

この問題を図 3.19 を用いて説明する。丸の右あるいは左上の数字が処理順序を示しており、まず k_1 について、1によって d_1 に含まれること、2によって d_3 に含まれることを調べながら処理が進み、5に到達したら次の k_2 に移動する。基本アルゴリズムでは、全ての丸が太線となっており、既に処理済みの検索文字列での結果に関わりなく位置検査が行われる。しかし、実際には2において d_3 が正解に含まれることが決まっているので、7,10において位置検査を行う必要はない。同様に、9,11,12も位置検査が不要であることがわかる。

不要な位置検査を排除するには、2番目以降のノードの処理において、まず候補文書を検索し、その文書が検索結果に含まれると判断されていない場合に限って位置検査を行うように改めればよい。改良アルゴリズム（図 3.20）では、2番目以降のノードに対するfindNext関数呼び出しを、位置検査を伴わないfindNextCandidate関数に改め、その候補文書がその時点までの検索結果に含まれていない場合のみcheckCandidate関数により位置検査を行う。この修正によって無駄な位置検査を排除でき、検索処理が高速化される。

改良アルゴリズムの動作例を図 3.21 に示す。図 3.19 において不要な位置検査と指摘した7,9,10,11,12が太線で囲まれておらず、位置検査が行われない。

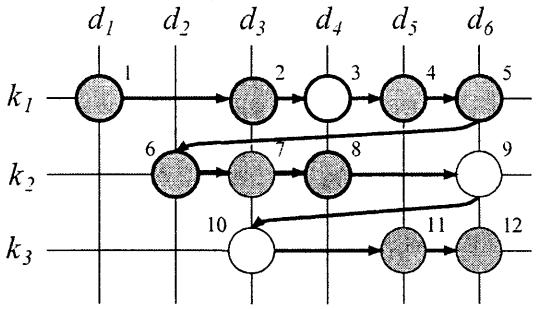


図 3.21: OR 演算子の拡張アルゴリズムの動作例

```
void AndnotNode::booleanRetrieve(BooleanResult& result)
{
  int docId0 = 1, docId1 = 0;
  while (1) {
    // 1番目の子ノードの該当文書を探す
    docId0 = child[0].findNext(docId0);
    if (docId0 == maxId) {
      // これ以上候補文書がないので、検索終了
      break;
    } else if (docId0 > docId1) {
      // 2番目の子ノードの該当文書を探す
      docId1 = child[1].findNext(docId0);
    }
    if (docId0 < docId1) {
      // 2番目の子ノードが追い越したので、1番目の子ノードの
      // 該当文書は ANDNOT としても該当文書
      result.push(docId0);
    }
    ++docId0;
  }
}
```

図 3.22: ANDNOT 演算子の基本アルゴリズム

3.4.3 ANDNOT 演算子

ANDNOT 演算子は、第2子ノードが補集合を求める NOT 演算子である AND 演算子と捉えることも可能であり、処理手順としては AND と同様に文書順が効率的である。ただし、第2子ノードの検索結果の補集合を求めてから AND を取るのは非効率であり、第1子ノード（検索文字列1）の該当文書の文書 ID 以上で最小の文書 ID を持つ第2子ノード（検索文字列2）の該当文書を検索しながら検索集合を決定するほうがよい。第1、2子ノードの該当文書の文書 ID を docId1, docId2 とすると、docId1=docId2 であれば docId1 は ANDNOT の該当文書ではなく、docId1<docId2 であれば docId1 は該当文書である。この考えに基づく ANDNOT 演算子の基本アルゴリズムは図 3.22 のようになる。

ANDNOT 演算子でも、基本アルゴリズムをそのまま用いると無駄な位置検査が行われて

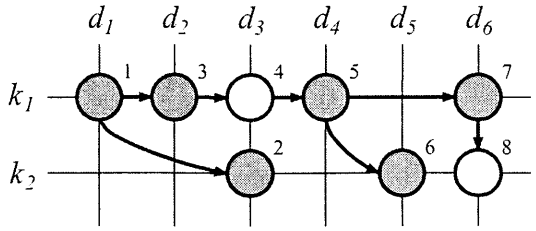


図 3.23: ANDNOT 演算子の基本アルゴリズムの動作例

しまう。すなわち、ANDNOT の意味から考え、検索文字列２が出現すると確定している文書について検索文字列１の位置検査は省略可能であるし、逆に検索文字列１が出現しないと確定している文書について検索文字列２の位置検査は省略可能である。ところが、基本アルゴリズムでは、検索文字列１、２に対して常に位置検査を行っているため、位置検査が無駄になり、検索時間を増大させる。

この問題を図 3.23 を用いて説明する。これまでとは若干処理順序が異なっており、1 で検索文字列１の最初の検索を行った後は、2 のように検索文字列２の検索を行うことになる。位置検査が無駄になっているのは、検索文字列２が出現しているのに検索文字列１の位置検査を行っている 4、および検索文字列１が出現しないのに検索文字列２の位置検査を行っている 6 である。

この問題を解決するには、`findNext` するところを候補文書を検索する`findNextCandidate` 関数に置き換えればよい。その代わり、検索結果かの判断は複雑になり、検索文字列１、２の候補文書 ID である docId0, docId1 に対して、docId0<docId1 の場合には検索文字列２、docId0=docId1 の場合には検索文字列１、２ともに位置検査した上で検索結果となるかを判断する必要がある。

拡張アルゴリズムによれば、動作例（図 3.25）のように 4,6 での位置照合が行われなくなる。

3.4.4 複合条件の扱い

前節までで提案した各論理演算子の拡張アルゴリズムは複数の論理演算子が入れ子になった複合条件においてもそのまま適用可能である。そして、複合条件においても各論理演算子が拡張アルゴリズムを採用すれば、検索文字列の位置検査を削減できるため、検索を高速化できる。以下では、拡張アルゴリズムを複合条件に適用する際の注意点に触れておく。

図 3.12・図 3.20・図 3.24 等の各論理演算子の拡張アルゴリズムにおいて、各子ノード（`child[i]`）が検索文字列に対応したクラスでなければならないという制約があるわけではない。ただし、AND・OR・ANDNOT 演算子を実現するクラスが`findNextCandidate`・`checkCandidate` の両関数を実現している必要がある。AND 演算子については 3.4.1 節で、図 3.13・図 3.14 の擬似コードを示してある。OR 演算子の疑似コードは図 3.26、図 3.27、ANDNOT 演算子の疑似コードは図 3.26、図 3.27 の通りである。

いずれの関数も、子ノードに対して再帰的に同じ関数を呼び出している。最終的には、検索文字列に対応するノードの`findNextCandidate` 関数、`checkCandidate` 関数が呼び出されるが、前者は図 3.7、後者は図 2.7 ですでに示した通りである。

```
void AndnotNode::booleanRetrieve(BooleanResult& result)
{
  int docId0 = 1, docId1 = 0;
  while (1) {
    // 1番目の子ノードの該当文書の候補を探す
    docId0 = child[0].findNextCandidate(docId0);
    if (docId0 == maxId) {
      // これ以上候補文書がないので、検索終了
      break;
    } else if (docId0 > docId1) {
      // 2番目の子ノードの該当文書の候補を探す
      docId1 = child[1].findNextCandidate(docId0);
    }
    if (docId0 < docId1)  {
      // 2番目の子ノードが追い越したので、1番目の子ノードの
      // 該当文書は ANDNOT としても該当文書
      if (child[0].checkCandidate(docId0) == true) {
        // docId0 が本当に出現していれば該当文書
        result.push(docId0);
      }
    } else {
      // 2番目の子ノードの候補文書と1番目の子ノードの
      // 該当文書が同じ
      if (child[1].checkCandidate(docId1) == false &&
          child[0].checkCandidate(docId0) == true) {
        // docId0 が本当に出現し、docId1 が本当に出現していなければ
        // 該当文書
        result.push(docId0);
      }
    }
    ++docId0;
  }
}
```

図 3.24: ANDNOT 演算子の拡張アルゴリズム

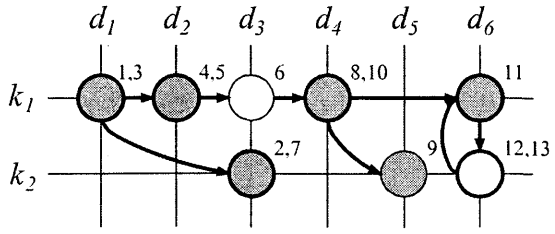


図 3.25: ANDNOT 演算子の拡張アルゴリズムの動作例

```
int OrNode::findNextCandidate(int currentDid)
{
  int nextId = maxId;
  for (int i = 0; i < child.size(); ++i) {
    // 先頭の子ノードの候補文書を見つける
    int tmpId = child[0].findNextCandidate(currentId);
    if (tmpId == currentId) {
      // currentIdと同じ文書IDが返れば、それが答え
      return tmpId;
    }
    if (tmpId < nextId) {
      // これまでのnextId以下の文書IDが返れば、それが答えの可能性を持つ
      nextId = tmpId;
    }
  }
  return nextId;
}
```

図 3.26: OR 演算子の`findNextCandidate` 関数

```
bool OrNode::checkCandidate(int currentId)
{
  for (int i = 0; i < child.size(); ++i) {
    if (child[i].checkCandidate(currentId) == true) {
      // 該当するIDがある。ひとつでもあれば該当文書である
      return true;
    }
  }
  return false;
}
```

図 3.27: OR 演算子の`checkCandidate` 関数

```
int AndnotNode::findNextCandidate(int currentDid)
{
  // 第1子ノードの候補文書をANDNOTの候補文書とすればよい
  return child[0].findNextCandidate(currentId);
}
```

図 3.28: ANDNOT 演算子の`findNextCandidate`関数

```
bool AndnotNode::checkCandidate(int currentId)
{
  if (child[0].checkCandidate(currentId) == true &&
      child[1].checkCandidate(currentId) == false) {
    return true;
  }
  return false;
}
```

図 3.29: ANDNOT 演算子の`checkCandidate`関数

第4章　ブーリアン検索高速化手法の評価

前章で提案したブーリアン検索の高速化手法の有効性を新聞記事8年分約750MBを用いて評価を行う。

4.1　評価データ

4.1.1　検索対象

検索対象には新聞記事8年分（毎日新聞CD-ROMデータ版1991～1998年）を使用した。このCD-ROMでは、記事ごとに見出し・掲載日・紙面・キーワード等の書誌項目も付与されているが、本実験では見出し・本文のみを使用した。文書数は855824件、文書サイズは748 MB（1件当たり930 B）であった。

4.1.2　単一検索文字列の検索条件

検索文字列の長さによる影響を調べるため、4文字から10文字までの7通り、文字長ごとに20個、合計140個の検索文字列を用意した。これらは、社内で運用している情報検索システムの検索ログから検索対象に少なくとも1件は該当文書がある文字列を選択したものである。

なお、文字列長を4文字からとしたのは、前章で提案した高速化手法が有効なのは検索文字列に複数のパスが存在する場合であり、bi-gram索引に対しては4文字以上が相当するからである。

4.1.3　論理演算子を含む検索条件

本研究が対象とするAND, OR, ANDNOTの3つの論理演算子とAND, ORを組み合わせた複合条件に対し、以下の検索条件を用意した。

- AND演算子

  3文字以上の文字列を少なくとも1つ含む2文字以上の文字列をAND演算子で結合したもの[1]。検索文字列数が2,3,4,5のもの各10個、合計40個。

  例：#and(複写機, 消費電力, 発熱, 低減)

[1] 3文字以上の文字列を少なくとも1つ含むとしたのは、論理演算子の高速化が有効なのは位置検査を含む長い文字列が検索条件に含まれる場合だからである。ただし、論理演算子の高速化には単一検索文字列の高速化が前提となるわけではないので、単一検索文字列の評価のように文字列長を4以上とする必要はなく、3文字以

- OR 演算子

  3 文字以上の文字列を少なくとも 1 つ含む 2 文字以上の文字列を OR 演算子で結合したもの。検索文字列数が 2,3,4,5 のもの各 10 個、合計 40 個。

  例： #or(内線, 構内回線, P B X)

- ANDNOT 演算子

  3 文字以上の文字列を少なくと 1 つ含む 2 文字以上の文字列を ANDNOT 演算子で結合したもの 30 個。

  例： #andnot(液晶, ディスプレイ)

- 複合条件

  3 文字以上の文字列を少なくとも 1 つ含む文字列を AND/OR 演算子で結合したもの 30 個。ただし、まず OR 演算子で結合し、つぎに OR 演算子あるいは文字列を AND 演算子で結合した形式。

  例： #and(#or(感光体, ドラム),#or(冷却, 風))

選択方法は単一文字列の場合と同じく社内システムの検索ログから選択した。

4.2 評価方法

性能評価のため、検索対象文書に関する bi-gram（$n = 2$）索引を作成した。使用したマシンは、Sun Microsystems 社のワークステーション Ultra 10 （CPU: Ultra SAPRC II 300MHz, メインメモリ 128 MB）で、OS は Solaris 2.5.1 である。索引は外付けローカルディスク（18GB, 7200rpm, Ultra-SCSI 接続）に置いた。索引作成に要した時間は 14.3 時間で、索引サイズは 1574 MB （もとテキストの 2.1 倍）であった。

検索時間を以下の 3 つの場合について測定した。

- COLD：

  システム起動直後のデータがキャッシュされていない状態での検索時間。OS レベルのキャッシュも無効となるように、索引のあるファイルシステムを測定前にマウントし直した直後に検索を行った場合の検索時間である。

- WARM：

  有効化フェーズに必要なデータがキャッシュされた状態での検索時間。有効化フェーズは検索文字列から文書頻度に基づいて検索に使用する n-gram を決定するフェーズであり、文書頻度がキャッシュされた状態での検索時間である。運用状態では n-gram ごとの文書頻度はキャッシュされるようになると考えられるので、この時間は運用状態での検索時間に近い値と考えられる。

  本実験では、COLD の測定において条件評価フェーズの処理時間を測定した。

上でよい。また、2 文字以上の文字列を組み合わせた場合に限定したのは、1 文字の検索文字列は bi-gram 索引においては OR で展開されるため、AND/OR を含む複合条件になってしまうからである。

- HOT：

  検索に必要なデータ全てをキャッシュした状態での検索時間。COLD の測定後に同一の検索条件を再度検索した場合の検索時間を測定する。

検索条件ごとに処理時間を 3 回測定し、その平均値をその検索条件の検索時間とする。さらに、全検索条件に関する検索時間の平均値を最終的な検索時間とした。なお、検索時間は全て秒単位である。

検索時間を決定する要因を確認するため、以下の数値も測定した。

- 検索処理全体でのページアクセス数：

  検索処理全体でアクセスしたページ数[2]。

- 条件評価フェーズでのページアクセス数：

  実際に検索結果文書を決定する条件評価フェーズにおいてアクセスしたページ数。

- 位置検査数：

  検索文字列ごとに位置検査した回数の合計。単一検索文字列の場合には候補文書数と一致し、論理演算子を含む場合には候補文書におけるその文書を候補文書とする検索文字列数の合計に一致する。

- 伸長処理数：

  索引に圧縮されて記録されている要素（文書 ID 差分、文書内頻度、出現位置圧縮長、出現位置差分）を伸長した回数。ここで、出現位置圧縮長とは文書内出現位置の圧縮したビット長のことであり、7.2.3 節・7.4.1 節において詳しく説明する。

検索条件ごとの測定値を全検索条件について平均したものを測定結果とする。

4.3 単一検索文字列の評価結果

4.3.1 高速化の効果

単一検索文字列条件については、従来方式である単純選択法と 3.3 節で提案した 3 つの高速化手法— 最小頻度法、全 n-gram 法、選択 n-gram 法 — を比較した。

全 140 個の検索条件に対する評価結果を表 4.1 に示す。この表で、各測定値の下の行は、ベースラインである単純選択法に対する増減率を表している。なお、全検索条件に対する結果件数の平均は 644 件であった。

全ての高速化手法において、検索時間は単純選択法よりも短かくなっており、提案手法の全てが有効であることが確認できる。しかし、検索時間を測定した COLD/WARM/HOT の 3 つの状況の結果を比べると、以下のように 3 つの提案手法の特性の違いが明らかになる。

- 最小頻度法と全 n-gram 法を比較すると、COLD, WARM では前者が高速だが、HOT では後者が高速である。

[2] ここでのページは n-gram 索引を実現する転置ファイル（詳細は 7 章参照）が管理する論理的なページのことであり、OS が管理する物理的なページではない。なお、ページサイズは 4KB である。

表 4.1: 高速化手法の比較

		単純選択法	最小頻度法	全 n-gram 法	選択 n-gram 法
検索時間	COLD	1.026	0.965	0.988	0.911
		—	−5.9%	−3.7%	−11.2%
	WARM	0.823	0.726	0.751	0.671
		—	−11.8%	−8.7%	−18.4%
	HOT	0.146	0.117	0.089	0.083
		—	−19.9%	−39.0%	−43.2%
ページ数	全体	71.6	66.2	69.5	63.4
		—	−7.5%	−2.9%	−11.4%
	評価	62.9	53.1	56.4	50.3
		—	−15.6%	−10.4%	−20.0%
位置検査数		1621	1398	740	850
		—	−13.8%	−54.3%	−47.6%
伸長処理数		180337	139654	99124	94784
		—	−22.6%	−45.0%	−47.4%

- 選択 n-gram 法はすべての状況において最小頻度法・全 n-gram 法を上回っている。

このような違いの原因は、次節において詳細に検討する。

検索時間とページ数あるいは伸長処理数の関係をグラフ化したものが図 4.1 である。この図の X 軸では、`MinFreq`が最小頻度法、`AllNgram`が全 n-gram 法、`SelNgram`が選択 n-gram 法を表している。また、`COLD/WARM/HOT Time`が COLD/WARM/HOT の検索時間、`Total Page Num`が全体のページ数、`Eval Page Num`が条件評価フェーズのページ数、`Decode Num`が伸長処理数を表している。この図から、COLD および WARM の検索時間は検索処理全体および条件評価フェーズのページ数と、HOT の検索時間は伸長処理数と強い相関があることが確認できる。なお、位置検査数も HOT の検索時間を左右する原因のひとつであると考えられる。しかし、全 n-gram 法と選択 n-gram 法を比較したとき、位置検査数は候補文書特定に用いる n-gram 数の多い前者のほうが少ないのに対し、伸長処理数は後者のほうが少なく、HOT 検索時間との相関は伸長処理数の方が明らかに高いことから、伸長処理数が HOT 検索時間の主要な決定要因と言える。

4.3.2 処理内容の分析

本研究の高速化手法は、3.1 節で述べたように、適切な n-gram を選択するために検索条件の有効化処理に時間をかけるが、位置照合削減により条件評価処理をそれ以上に短縮し、全体の時間も短縮するというものである。表 4.1 により検索処理全体の時間短縮は確認できたので、処理内容に応じた詳細な分析を行う。

以下では、検索処理を機能分類と処理フェーズの 2 つの軸をから分析する。機能分類としては、n-gram 分割処理・n-gram 選択処理・ファイルアクセス・伸長処理等があるが、代表

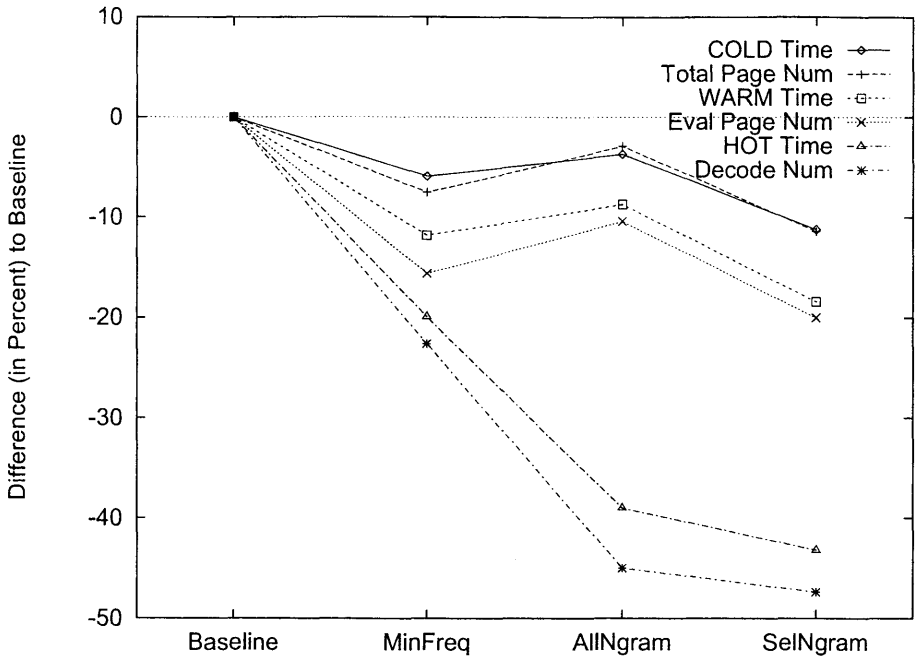


図 4.1: 高速化手法の比較

的ないくつかの検索文字列の処理プロファイリングからファイルアクセス・伸長処理以外の割合は非常に小さいことがわかったので、ファイルアクセス・伸長・その他の３つに分類した。一方、処理フェーズは有効化と条件評価の２つから構成されており、その詳細は以下の通りである。

- 有効化フェーズ

  与えられた検索文字列から適切な n-gram を選択し、対応する転置リストを用意して後続の評価処理を実施できる状態にする。具体的には、検索文字列を n-gram に分割し、n-gram に対応する転置リストのヘッダーをファイルから読み出し、必要であれば n-gram の出現頻度に基づいて n-gram の選択を行う。

  転置リストのヘッダーの読み出しに比べて n-gram 分割・n-gram 選択など CPU 処理の割合は極めて小さく、以下の分析ではその他としてまとめることとする。

- 条件評価フェーズ

  転置リストの出現情報を用いて検索文字列を含む文書を特定する。具体的には、必要となった転置リストの出現情報はファイルから読み出し、伸長する。

  ここでも処理の大部分はファイルアクセスであるが、伸長処理も無視できない時間を占めている。

表 4.2 は COLD の検索時間の分析結果である。この表から以下のことがわかる。

- 有効化のページアクセス数の増加が有効化時間を増加させている

  この表から有効化において時間増加の原因はファイルアクセスであることがわかる。表 4.1 の結果から、有効化中のアクセスページ数は単純選択法が 8.7 ページ、最小頻度

表 4.2: 高速化手法の比較

		単純選択法	最小頻度法	全 n-gram 法	選択 n-gram 法
有効化	アクセス	0.201	0.236	0.234	0.237
		—	+17.4%	+16.4%	+17.9%
	その他	0.002	0.003	0.003	0.003
		—	+50.0%	+50.0%	+50.0%
	小計	0.203	0.239	0.237	0.240
		—	+17.7%	+16.7%	+18.2%
条件評価	アクセス	0.677	0.609	0.662	0.588
		—	−10.0%	−2.2%	−13.1%
	伸長	0.135	0.107	0.079	0.073
		—	−20.7%	−41.5%	−45.9%
	その他	0.011	0.010	0.010	0.010
		—	−9.1%	−9.1%	−9.1%
	小計	0.823	0.726	0.751	0.671
		—	−11.8%	−8.7%	−18.5%

法・全 n-gram 法・選択 n-gram 法が 13.1 ページであり、50.6%も増加している。ページ数の増加ほどは処理時間が増大していないのは、今回の実装ではページサイズが 4K バイトであり、同一検索文字列中の n-gram が近隣のページに配置されている場合にはディスクレベルでは 1 回のアクセスで済む場合があること等が原因として考えられる。

- 条件評価では、ページアクセス数／伸長回数ともに減少している

  条件評価ではページアクセスの占める割合が高くて 80%程度で、残りのほとんどが伸長処理である。高速化手法によりページアクセス数／伸長回数ともに減少しており、条件評価時間も短縮されている。特に伸長処理の減少率が大きく、ディスクアクセスのない HOT において高速化の効果が大きいことの理由になっている。

- 有効化よりも条件評価の全体に占める割合が高く、トータル時間は減少している

  有効化の増加割合と条件評価の削減割合はほぼ同じである。しかし、全体に占める割合は、ベースラインである単純選択法において前者が約 25%、後者が 75%であるため、トータルの検索時間は減少している。

4.3.3 検索文字列長の影響の分析

検索文字列長ごとの各手法の検索時間とベースラインに対する増減比率を図 4.3 〜図 4.7 に示す。この図において、`Baseline` はベースライン、`MinFreq` は最小頻度法、`AllNgram` は全 n-gram 法、`SelNgram` は選択 n-gram 法の結果を示す。また、文字列長ごとの検索結果件数は文字列長順に 2289, 1244, 500, 379, 33, 52, 17 件であった。

以下、これらのグラフから手法ごとの特性を確認する。

単純選択法

ベースラインとなる手法であり、以下の傾向が見られる。

- 検索文字列長が偶数の場合に検索時間が短い。

  奇数とその次の偶数ではパスを構成する n-gram 数は一致するが、文字列長が大きいほうが検索結果件数が少なく、ファイルアクセス・CPU 処理（位置検査・伸長処理等）ともに小さいためと考えられる。

- 偶数・奇数ごとに比較した場合、COLD, WARM では検索文字列長に応じて検索時間が増大し、HOT では減少している。

  検索文字列長に応じて使用 n-gram 数は増大し、それに伴いファイルアクセスも増大するため、ファイルアクセスが検索時間の主要因となる COLD, WARM では検索時間が増大する。一方、HOT の場合、検索結果件数の減少に伴い CPU 処理も少なくなるため、検索時間が減少する。

最小頻度法

平均検索時間（表 4.1）で比べると、COLD/WARM/HOT 全ての場合でベースラインである単純選択法より高速であった。単純選択法と比較した場合、以下の傾向が見られる。

- COLD の場合、奇数では検索時間が短縮するが、偶数では増大する。

  3.3 節で述べたように、最小頻度法が有効なのは最小頻度パスが複数個存在する場合であり、bi-gram 索引においては 5 以上の奇数の場合に相当する。逆に偶数では、単純選択法によって求まるパスと最小頻度パスが一致するため、条件評価の処理時間には差がないが、最小頻度パスを求める処理の分だけ最小頻度法の方がコストがかかる。以上の理由から、奇数では検索時間が短縮するが、偶数では増大する。

- WARM, HOT の場合、偶数の検索時間がほぼ同じとなる。

  最小頻度パスを求めるコストは検索文字列中の全ての n-gram の文書頻度を得るためのディスクアクセスとそれに基づくパスの選択に大別できるが、両者を比べるとディスクアクセスが大半を占める。しかし、WARM では有効化に要するファイルアクセスがキャッシュにより 0 となるため、条件評価の処理時間を比較することになる。偶数の場合、前述のようにベースラインと最小頻度法が用いるパスが同じであるので、検索時間もほぼ同じとなる。なお、奇数における検索時間の短縮割合は WARM, HOT と大きくなるが、キャッシュされるデータが増えるに従って、CPU 処理の減少分が検索時間に直接的に現れるようになるからである。

全 n-gram 法

平均検索時間（表 4.1）では、ベースラインよりは常に高速であり、最小頻度法と比べると COLD/WARM では若干増大するものの HOT では短縮していた。検索文字列長に着目して最小頻度法と比較した場合、以下の傾向が見られる。

- 6 以外の偶数では検索時間が短縮している。

  全 n-gram 法では、候補文書特定用の n-gram を増やしたことにより候補文書数が減少し、検索時間も短縮される。ただし、今回の実験では文字列長が 6 の場合に最小頻度法よりも検索時間が増大した。これは、3.3.3 節で触れたように、候補文書絞り込みに有効でない n-gram が多い場合には全 n-gram 法は最小頻度法よりも検索コストが増大することがあるが、それに該当する検索文字列が長さ 6 の場合にたまたま多かったことが原因と思われる。なお、ベースラインと比較すると、長さ 6 の COLD/WARM 以外では常に検索時間は短い。

- 奇数では検索時間が増大している。

  最小頻度法よりも検索時間が増大している原因は、候補文書絞り込みが有効に作用しなかったことが考えられる。ここで問題なのは、偶数では長さ 6 だけの場合だけであったのに対し、奇数では常に増大している点である。この理由としては、奇数の場合、位置検査用のパス選択の時点で文書頻度の少ない n-gram が選ばれているため、候補文書特定に追加する n-gram が候補文書絞り込みに有効でないことが多くなるためと推察される[3]。

選択 n-gram 法

選択 n-gram 法は平均検索時間（表 4.1）では最速であったが、検索文字列長ごとに比較してもほとんどの場合に最も優れた方法となっている。

- 偶数の場合、全 n-gram 法よりも検索時間がさらに短い。

  偶数の場合、これまで見てきた中では全 n-gram 法が最速であるが、選択 n-gram 法はそれよりさらに高速である。特に COLD/WARM において検索時間が短くなっている。この理由としては、候補文書特定に使用する n-gram を文書頻度に基づいて選択することにより、候補文書特定の効果はほぼ同等のまま、使用 n-gram 数を減らすことでディスクアクセスを減らすことができたからと考えられる。

- 奇数の場合、最小頻度法と検索時間はほぼ同じである。

  奇数の場合、これまで見てきた中では最小頻度法が最速であり、選択 n-gram 法はそれと同等であることがわかる。全 n-gram 法のところで触れたように、奇数に関しては最小頻度法により的確なパスが選択されており、最小頻度パスが候補文書特定に関してもほぼ最適なものとなっているため、このような結果が得られている。ただし、最小頻度パスであっても候補文書特定に n-gram を追加したほうが有効な場合もあることが、HOT において選択 n-gram 法が優っていることから確認できる。

[3] 最小頻度法では、全 n-gram 法で追加する n-gram の文書頻度がパスに使用されているものよりも必ず大きくなるわけではない。なぜなら、検索文字列全体に対する文書頻度の和が最小のものを選ぶため、個々の n-gram に注目した場合には、前後の最小頻度パスを構成する n-gram よりも文書頻度が小さくなり得るからである。

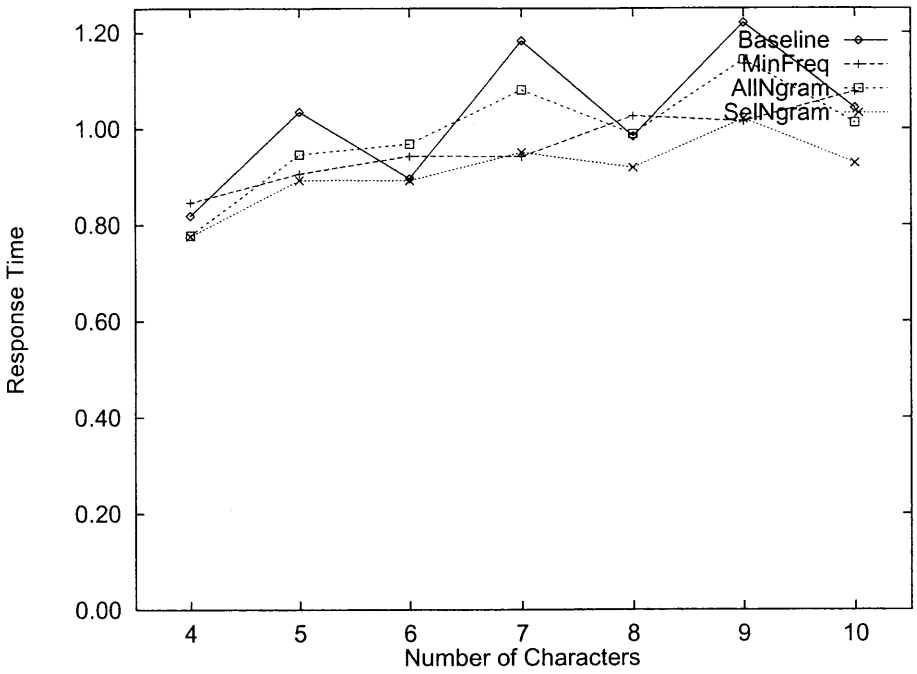


図 4.2: 検索文字数と検索時間 (COLD) の関係

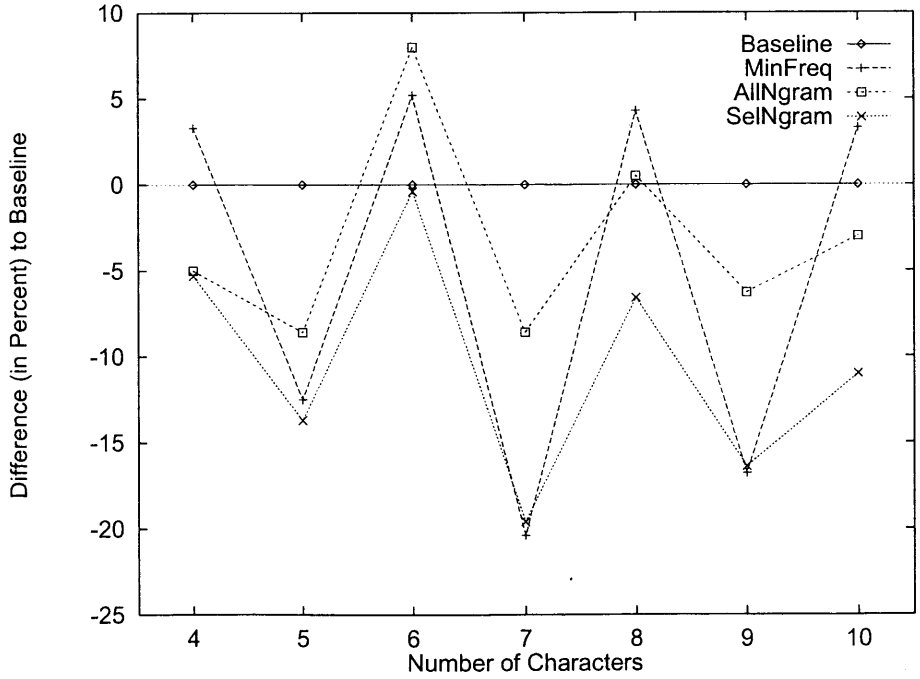


図 4.3: 検索文字数と検索時間増減率 (COLD) の関係

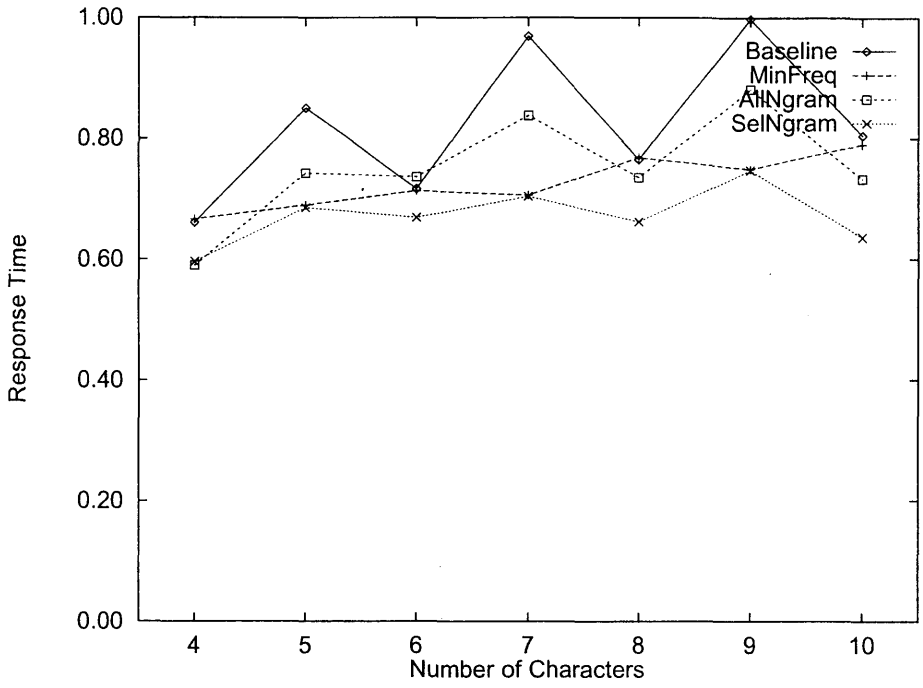


図 4.4: 検索文字数と検索時間 (WARM) の関係

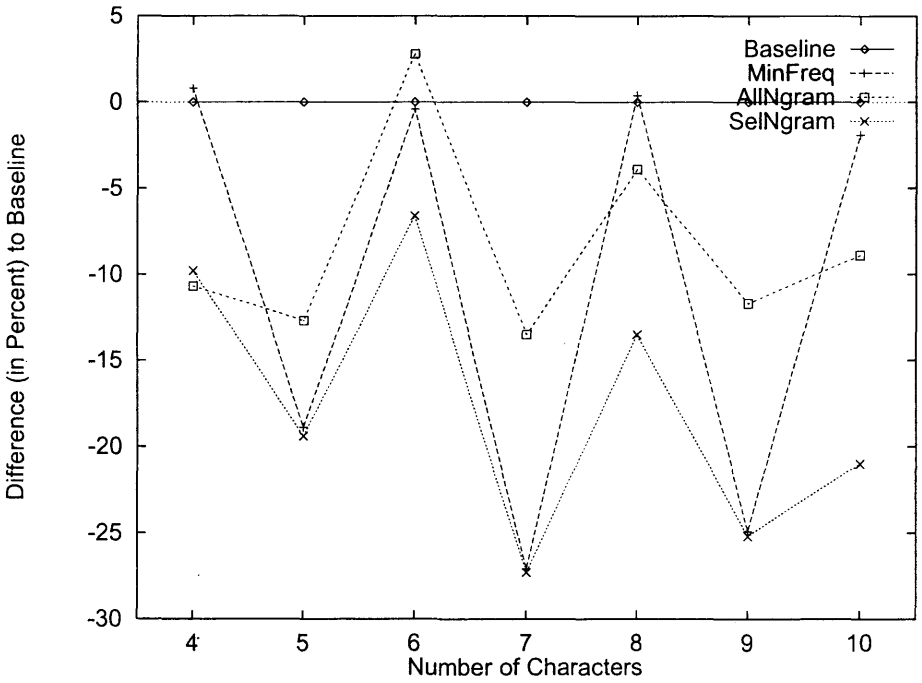


図 4.5: 検索文字数と検索時間増減率 (WARM) の関係

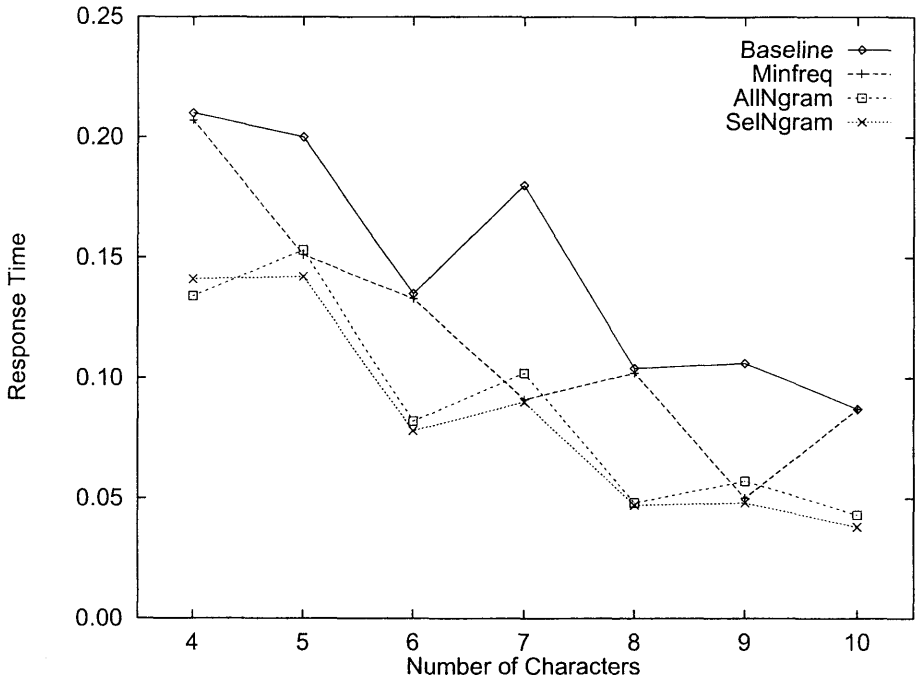


図 4.6: 検索文字数と検索時間 (HOT) の関係

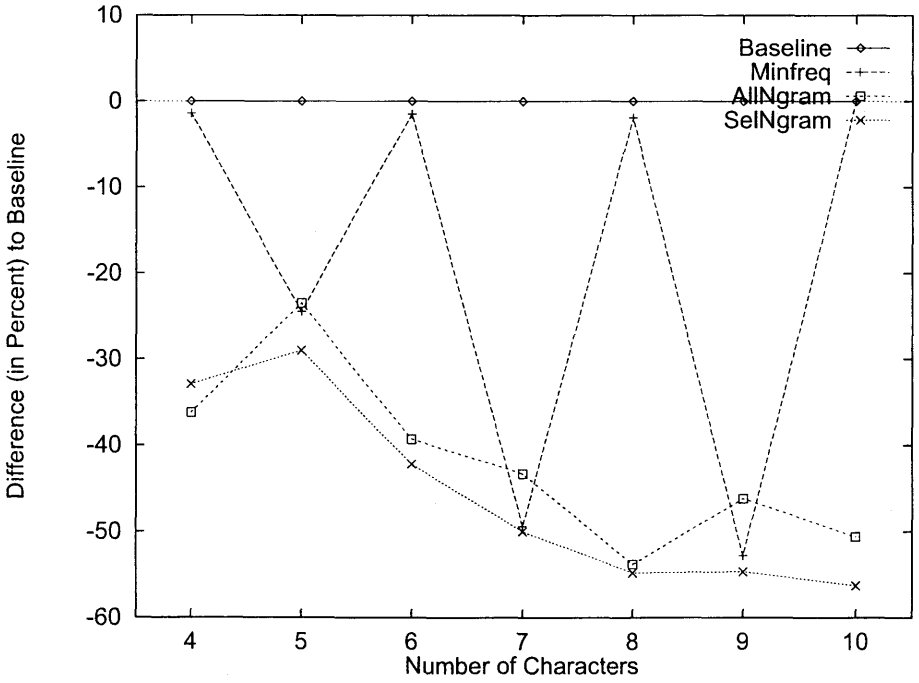


図 4.7: 検索文字数と検索時間増減率 (HOT) の関係

4.4 論理演算子の評価結果

前節の結果から、単一文字列検索の高速化手法としては選択 n-gram 法が最も優れていることがわかった。したがって、論理条件の高速化については、単一文字列の処理方法として従来手法である単純選択法と選択 n-gram 法の２つを用いた場合のみを評価することとした。

4.4.1 AND 演算子

AND 演算子の高速化手法の比較結果を表 4.3 に示す。この表では、検索文字列の処理方法ごとに演算子処理の基本アルゴリズム（表中では基本）と拡張アルゴリズム（表中では拡張）を比較している。拡張欄の括弧内の数字は基本に対する拡張の増減率を、各測定値の下の行は単純選択法に対する選択 n-gram 法の増減率（ただし、この行の最右欄の括弧内の数字は単純選択法の基本に対する選択 n-gram 法の拡張の増減率）を表している。

この表から、COLD/WARM/HOT いずれの場合も検索時間は減少しており、高速化の効果が確認できる。その傾向を詳しく分析するとつぎのようになる。

- 基本アルゴリズムに関して単純選択法と選択 n-gram 法を比較すると COLD/WARM/HOT でそれぞれ 1.0, 5.0, 15.5% 短縮されており、単一検索文字列の高速化により AND 演算子を含む検索条件処理も高速化されることが確認できる。

  ただし、単一検索文字列の場合（11.2, 18.4, 43.2%の短縮）と比較すると、短縮の割合が小さくなっている。これは、AND で結合される検索文字列の中には n に等しいものも含まれており、それらの検索文字列には高速化手法が効かないためと考えられる。

- 基本と拡張の両アルゴリズムを比較すると、単純選択法・選択 n-gram 法いずれの場合でも検索時間は十数%（COLD）から 50%以上（HOT）と大幅に短縮しており、拡張アルゴリズムの有効性がわかる。

- 拡張による検索時間短縮の割合を見ると、選択 n-gram 法の方が若干ではあるが効果が大きい。

- 単純選択法の基本と選択 n-gram 法の拡張を比較すると検索時間はさらに短縮されており、両者の組み合わせにより相乗効果が得られることがわかる。

  拡張アルゴリズムでは全ての検索文字列の候補文書である場合のみ位置検査を行うため、検索文字列ごとの候補文書数が少ないほど高速化の効果も大きくなる。選択 n-gram 法は単純選択法よりも候補文書数を少ないので、選択 n-gram 法のほうが拡張アルゴリズムが有効に作用すると考えられる。

- 拡張アルゴリズムによる増減率は伸長処理数よりも位置検査数の方が大きい。

  伸長処理は n に等しいものも含めた全ての検索文字列に関する処理であり、拡張アルゴリズムにより削減されるのは長い検索文字列処理に相当する部分だけである。一方、位置検査数はそもそも長い検索文字列のみに付随する処理なので、拡張アルゴリズムの効果が直接あらわれる。それゆえ、位置検査数の減少の割合が大きいと考えられる。

表 4.3: AND 演算子処理における高速化手法の比較

文字列処理	単純選択法			選択 n-gram 法		
演算子処理	基本	拡張		基本	拡張	
検索時間 | COLD	1.405	1.180	(−16.0%)	1.391	1.160	(−16.6%)
	—	—		−1.0%	−1.7%	(−17.4%)
| WARM	1.016	0.792	(−22.0%)	0.965	0.725	(−24.9%)
	—	—		−5.0%	−8.5%	(−28.6%)
| HOT	0.155	0.063	(−59.4%)	0.131	0.044	(−66.4%)
	—	—		−15.5%	−30.2%	(−71.6%)
ページ数 | 全体	107.2	81.8	(−23.7%)	102.7	78.9	(−23.2%)
	—	—		−4.2%	−3.5%	(−26.4%)
| 評価	90.2	64.8	(−28.2%)	81.4	57.5	(−29.4%)
	—	—		−9.8%	−11.3%	(−36.3%)
位置検査数	2379	427	(−82.1%)	2001	260	(−87.0%)
	—	—		−15.9%	−39.1%	(−89.1%)
伸長処理数	179022	117248	(−34.5%)	143853	71511	(−50.3%)
	—	—		−19.6%	−39.0%	(−60.0%)

4.4.2 OR 演算子

OR 演算子の高速化手法の比較結果を表 4.4 に示す。ここでも検索時間は短縮したが、その傾向は AND 演算子の場合とは異なる面もある。

- 基本アルゴリズムでは、選択 n-gram 法の方が単純選択法より検索時間は短く、単一検索文字列の高速化により OR 演算子を含む検索条件処理も高速化されている。

  検索時間の短縮率は COLD/WARM/HOT で 2.5, 9.1, 25.9% であり、nに等しい長さの検索文字列が含まれているため単一検索文字列の場合よりも小さいことでも AND 演算子の場合と同じである。

- 拡張アルゴリズムによって検索時間は短縮されているが、最も検索時間が短縮される選択 n-gram 法の HOT の場合であっても短縮率は 4.4%と小さく、拡張アルゴリズムの効果は AND 演算子と比べて限定的である。

  このような差異が生じたのは、位置検査を省略できる状況が AND と OR で異なるからである。AND 演算子では、全検索文字列について候補文書と判断された文書以外では、各検索文字列の候補文書において位置検査を省略可能なので、実際に省略されることが多い。一方 OR 演算子では、各検索文字列の候補文書がすでに処理した検索文字列のいずれかで該当文書と判断された場合においてのみ位置検査を省略できるが、これに該当するのは複数の検索文字列が同じ文書に出現している場合に限られる。実際、位置検査数の削減率を選択 n-gram 法で比較すると、AND 演算子の 87.0%に対し OR 演算子は 20.7%と小さい。

表 4.4: OR 演算子処理における高速化手法の比較

文字列処理	単純選択法			選択 n-gram 法		
演算子処理	基本	拡張		基本	拡張	
検索時間 | COLD	1.385	1.373	(−0.9%)	1.350	1.332	(−1.3%)
	—	—		−2.5%	−3.0%	(−3.8%)
検索時間 | WARM	0.977	0.967	(−1.0%)	0.888	0.880	(−0.9%)
	—	—		−9.1%	−9.0%	(−9.9%)
検索時間 | HOT	0.243	0.236	(−2.9%)	0.180	0.172	(−4.4%)
	—	—		−25.9%	−27.1%	(−29.2%)
ページ数 | 全体	84.0	83.6	(−0.5%)	85.9	85.1	(−0.9%)
	—	—		+2.2%	+1.8%	(+1.3%)
ページ数 | 評価	68.1	67.7	(−0.6%)	65.3	64.5	(−1.2%)
	—	—		−4.1%	−4.8%	(−5.3%)
位置検査数	5217	3880	(−25.6%)	3228	2513	(−22.1%)
	—	—		−38.1%	−35.2%	(−51.8%)
伸長処理数	372234	272606	(−26.8%)	252157	199981	(−20.7%)
	—	—		−32.2%	−26.6%	(−46.3%)

なお、位置検査数よりも HOT 検索時間の削減率が小さくなっているのは、OR の場合には処理中の検索文字列の候補文書が中間検索結果に含まれているかを検索して確認する必要があるためと考えられる。現在の実装では検索結果を文書 ID の配列で表現しているため、この確認のために中間結果に対する走査が発生する。中間結果の走査はメモリ上の処理であるが、ディスクアクセスのない HOT では無視できない割合となっている。ビット列表現に改めれば走査は不要となるので、さらなる高速化が可能となる。

4.4.3 ANDNOT 演算子

ANDNOT 演算子の高速化手法の比較結果を表 4.5 に示す。ここでも、位置検査回数は減少し、検索時間も短縮されており、拡張アルゴリズムの有効性が確認できた。短縮効果の傾向は、AND 演算子の場合に近い。これは 3.4.3 節で述べたように ANDNOT 演算子は AND 演算子の一種と捉えることができ、位置検査を省略できる場合が多いからである。ただし、ANDNOT はオペランド数が 2 に限定されるので、高速化の効果は AND の場合より小さくなっていると考えられる。

4.4.4 複合条件

最後に複合条件の測定結果を表 4.6 に示す。ここでも、位置検査数は減少し、検索時間も短縮されており、拡張アルゴリズムの有効性が確認できた。ただし、位置検査数および

表 4.5: ANDNOT 演算子処理における高速化手法の比較

文字列処理	単純選択法			選択 n-gram 法		
演算子処理	基本	拡張		基本	拡張	
| COLD	1.195	1.151	(−3.7%)	1.177	1.131	(−3.9%)
検 |	—	—		−1.5%	−1.7%	(−5.4%)
索 | WARM	0.875	0.826	(−5.6%)	0.842	0.764	(−9.3%)
時 |	—	—		−3.8%	−7.5%	(−12.7%)
間 | HOT	0.277	0.235	(−15.2%)	0.254	0.205	(−19.3%)
|	—	—		−8.8%	−12.8%	(−26.0%)
ペ | 全体	69.2	65.6	(−5.2%)	69.2	65.1	(−5.9%)
ー |	—	—		±0.0%	−0.8%	(−5.9%)
ジ | 評価	58.5	54.9	(−6.2%)	55.7	51.6	(−7.4%)
数 |	—	—		−4.8%	−6.0%	(−11.8%)
位置検査数	6887	4452	(−35.4%)	6202	3906	(−37.0%)
	—	—		−9.9%	−12.3%	(−43.3%)
伸長処理数	238823	210726	(−11.8%)	211951	183397	(−13.5%)
	—	—		−11.3%	−13.0%	(−23.2%)

表 4.6: 複合条件における高速化手法の比較

文字列処理	単純選択法			選択 n-gram 法		
演算子処理	基本	拡張		基本	拡張	
| COLD	2.224	2.183	(−1.8%)	2.238	2.217	(−0.9%)
検 |	—	—		+0.6%	+1.6%	(−0.3%)
索 | WARM	1.809	1.786	(−1.3%)	1.788	1.761	(−1.5%)
時 |	—	—		−1.1%	−1.4%	(−2.6%)
間 | HOT	0.695	0.668	(−3.9%)	0.652	0.626	(−4.0%)
|	—	—		−6.2%	−6.3%	(−9.9%)
ペ | 全体	156.9	143.3	(−8.7%)	159.8	143.8	(−10.0%)
ー |	—	—		+1.8%	+0.3%	(−8.3%)
ジ | 評価	126.4	112.7	(−10.8%)	124.8	108.9	(−12.7%)
数 |	—	—		−1.3%	−3.4%	(−13.8%)
位置検査数	35805	7403	(−79.3%)	31180	6184	(−80.2%)
	—	—		−12.9%	−16.5%	(−82.7%)
伸長処理数	4633078	1283032	(−72.3%)	4602153	1219819	(−73.5%)
	—	—		−0.7%	−4.9%	(−73.6%)

ページアクセスの減少に比べて検索時間の短縮の割合は小さい。これは実装上の問題と考えられる。すなわち、評価で用いた検索条件ではORがANDの子ノードとなっているので、`OrNode::findNextCandidate`関数が呼び出される。この関数は図3.26のアルゴリズムに沿って実装されており、この関数が呼び出されるたびに単純にforループを回して全ての子ノードの`findNextCandidate`関数を再帰的に呼び出しているため非効率である。つまり、位置検査数（および伸長処理数）の削減分をこの関数が無駄にしてしまい、検索時間を効果的に短縮できなかったと考えられる。この問題を解決するには、子ノードをその時点の候補文書IDに小さい順にヒープで管理し、必要な場合のみ子ノードに対して`findNextCandidate`関数を呼び出すように修正すればよい[Wit94]。

4.5 従来の高速化手法との関係

これまでの評価結果により、本研究で提案した位置検査省略が検索処理高速化に有効であることが確認できた。本節では、提案手法とn-gram索引に対する従来の高速化手法との関係について考察する。従来の高速化手法には、1.5節で述べたように、n-gram抽出法、検索処理法、転置ファイルの物理編成法の3種類がある。本研究は検索処理法に関する改良提案であるので、以下ではn-gram抽出法と物理編成法と提案手法との関係について考察する。

4.5.1 n-gram抽出法に関する検討

n-gram抽出法に関する従来手法について整理する。従来手法（詳細は1.5.1節）には以下のものがある。

- nの組み合わせ

  複数のn-gram索引を組み合わせることで、1文字を含めた様々な文字数の検索文字列に対して高速検索を実現する。ただし，この方法には索引ファイルが大きくなるという問題がある。

- 文字種適応

  日本語における文字種の使われ方に応じて文字種ごとに抽出するnを調整し、索引ファイルをあまり大きくすることなく平均検索速度を向上させる。

- nの動的調整

  抽出するn-gramの長さを、登録済みの文書における出現頻度に応じて動的に調整し、索引ファイルをあまり大きくすることなく平均検索速度を向上させる。

以上の説明からわかるように、程度の違いはあるものの、n-gram抽出法の改良では高速化と引き替えに索引サイズが増大する。これに対し、提案手法は検索処理の改良であるため、索引サイズに影響することなく検索を高速化できる点で大きく異なっている。その一方で、提案手法はn-gram抽出法の改良と矛盾するものではなく、組み合わせ可能である。ただし、以下のように高速化効果に対しては影響が予想される。

- nの組み合わせに関しては、提案手法は基本的な場合と全く同様に適用可能である。nの組み合わせを行っても、その中の最大のnよりも長い検索文字列は最大のnに対応するn-gram索引によって処理されるからである。

- 文字種適応に関しては、検索文字列の構成文字種によって分けて考える。

  検索文字列が単一文字種から構成されていれば、本高速化手法をそのまま適用可能である。ただし、カタカナのように平均単語長の長いものに対してはnを大きくするのが一般的であり、パスが複数個存在するような、高速化手法が有効な長さ（$2n$以上）に達しない場合が多くなり、高速化の効果は若干小さくなると考えられる。

  検索文字列が複数の文字種から構成されている場合、検索文字列は同一文字種の連続部分に分割され、処理される。例えば、検索文字列が「デジタル画像処理」であれば「デジタル」「画像処理」に分割される。同一文字種部分の処理は基本的な場合と同じなので、部分ごとに本手法を適用することが可能である。ただし、同一文字種部分の長さは検索語よりも短くなるので、本手法の効果は小さくなる。

- nの動的調整の場合も文字種適応と同様に考えられる。すなわち、検索文字列において出現頻度の高い部分文字列は長いn-gramとして抽出されるので、パスが複数個存在するような場合が少なくなり、高速化の効果は小さくなる。

4.5.2 物理編成法に関する検討

物理編成法に関する従来手法について整理する。従来手法（詳細は1.5.3節）には以下のものがある。

- 位置情報の圧縮長の記録

  文書内出現位置が不要な場合にそれらを伸長しなくて済むように、位置情報の圧縮長を記録しておく。

- 圧縮情報の構造化

  転置リスト内部をブロック化し、ブロックレベルの探索とブロック内部の伸長を伴う検索の2段階処理で文書IDの伸長を減らす。また、文書IDとそれ以外の情報を異なるページなどの割り当てる。

いずれの方法も、文書IDのみを用いる検索処理と出現位置情報を用いる検索処理の特性の違いに着目し、検索を高速化するものである。提案手法は候補文書特定と位置検査という2つのフェーズから構成されており、従来の物理編成法の改良の狙いと一致している。したがって、改良した物理編成法を採用することで、提案手法の高速化効果は増大すると考えられる。実際には、本章の評価実験で使用したn-gram索引は提案手法向きの物理編成法（詳しくは7章参照）を採用しているので、位置情報の圧縮長の記録等の物理編成法の改良を全く実施しない場合の提案手法の効果は前節までの結果よりも小さいと予想される。