社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2009－ＦＩ－94 (9)
2009－ＤＤ－70 (9)
2009／3／25

ダブル配列を用いた文書検索用キーワード提示の高速化

三上崇志　　相川勇之　　川又武典
三菱電機株式会社　情報技術総合研究所

カーナビや携帯電話などの電子機器の高機能化に伴い，製品の操作説明書あるいは施設名や住所などを電子化して機器上で検索・閲覧するニーズが高まっている．そのためキーボードがなく文字入力が困難な機器上でも，簡単に文書を検索できるインターフェースが求められている．そこで本稿では，組込機器上でもユーザがストレスなく利用できるキーワードの自動提示インターフェースの実現を目的として，組込機器上でも高速に動作可能な前方一致検索方式を提案する．本提案方式では，ダブル配列法に最小・最大子ノード配列を導入することで高速に前方一致単語を列挙できるようにした．組込機器上で約 64 万件の辞書を用いた検索速度評価を行い，読み仮名 1 文字に前方一致する単語の検索において従来のダブル配列法よりも 2.5～6.8 倍高速となる結果を得た．

Speed-up Technique for Keyword Suggestion in Document Retrieval using Double Array

Takashi Mikami　　Takeyuki Aikawa　　Takenori Kawamata
Information Technology R&D Center, Mitsubishi Electric Corporation

Owing to increasing functions of the equipments such as car navigation systems and mobile phones, demands for retrieval and browsing in the multifunctional equipments are certainly growing, especially for retrieval of huge data such as electronic operating manuals, facilities names and addresses. It is required that the user interface facilitates retrieval of words from those huge data in the keyboard-less equipments whose character input is difficult. This report proposes a prefix search method that can be operated at high speed in the embedded equipments, in order to achieve the automatic keyword sugesstion interface without users' stress. We introduced the array of minimum and maximum child nodes into the double-array structures to enumerate the words that match the input character strings forward at high speed. Our proposed method achieved 2.5 to 6.8 times faster than a conventional double-array structure in the retrieval speed evaluation for enumerating the words in 640,000 dictionary entries, that match forward one Japanese syllabary character input in the embedded system.

1. はじめに

携帯電話やカーナビなどの電子機器の高機能化が進み，製品の操作説明書あるいは施設名や住所などを電子化して機器上で検索・閲覧するニーズが高まっている．そこで我々は，キーボードがなく文字入力が困難な機器上でも，簡単な操作で操作説明書や施設名などの検索を可能とするため，読みを数文字入力するだけで検索用のキーワード候補を自動提示する「入力サジェスト機能」を開発している[1][2].

入力サジェスト機能ではキーワード候補の表記文字列（および付加情報）とその読み仮名を格納した辞書をあらかじめ用意し，ユーザが入力中の読み文字列に前方一致する辞書エントリを検索してキーワード候補一覧として出力する．ユーザがストレスなく入力サジェスト機能を利用するためには，1 文字入力される度に行う読み仮名の前方一致検索を，高速に実行しなくてはならない．しかし，例えばカーナビに搭載される施設名や住所などは 1000 万件を越えるのが一般的であり，2 分探索などの単純な方式では実用的な検索速度が得られない．また，一般に組込機器は主記憶の容量が小さく，1 つのプロセスにあまり多くを割り当てることはできないため，検索用の索引をすべて主記憶に配置して高速化を図るといったこともできない．

高速な辞書検索の方式にはハッシュ法，2 分探索木，トライ法などがある．この中でもトライ法[3]はキーの検索時間が辞書の登録数に依存せずに高速であることや，検索文字列の前方部分文字列が辞書に登録されているかどうかを高速に検出できることなどを理由に，自然言語処理分野において広く利用されている．トライ法の実装方式の一つとして提案されているダブル配列法[4]は，コンパクトなデータサイズで高速な検索を実現する優れたデータ構造である．しかし，従来のダブル配列法では高速に前方一致検索を行えるようにはなっておらず，入力サジェストの目的には利用できない．本稿では，ダブル配列法の改良により，組込機器環境においても動作可能な前方一致検索方式を提案する．

以降，2章でダブル配列法の概要と入力サジェスト機能のための要求について述べる．3章で関連研究について述べ，4章で提案する方式を説明する．5章で実験による提案方式の検証を行い，6章でまとめと今後の課題を述べる．

2. ダブル配列法と入力サジェスト機能のための要求

2.1節でダブル配列法の基本となるトライ構造について説明し，2.2節でダブル配列法について説明する．2.3節で入力サジェスト機能のための要求を述べる．

2.1. トライ構造

トライ構造は，辞書データを木構造形式で格納したデータ構造である．単語集合{“aaa”, “abc”, “abcd”, “abfgh”, “afghi”}に対して構築したトライ構造の模式図を図 1に示す．トライ構造では，辞書の 1 文字に応じて遷移する検索途中の状態をノードとして表現する．ノード間のエッジは辞書データの単語を構成する各文字に対応する．図 1では便宜的に連番のノード番号を振ってある．また，“#”を文字列の終端を表す記号として用いている．文字列検索時はルートノード(ノード番号 1)から開始して，検索文字列と 1 文字ずつ比較することで木構造上のノード間を遷移する．例えば辞書に“abcd”が登録されているかどうかを検索する場合，図 1のノードを 1，2，6，7，9，10，の順に辿っていけば“abcd”が登録されていることがわかり，検索成功となる．また，“abce”を検索する場合は，1，2，6，7 とノードを辿った後，“e”で遷移できるノードが存在しないため，“abce”は辞書に登録されていないということがわかる．トライ構造はノードを遷移する度に終端文字で遷移できるかどうかを確認することで，検索文字列の前方部分文字列に一致する登録単語をすべて発見でき，日本語形態素解析などで利用される．

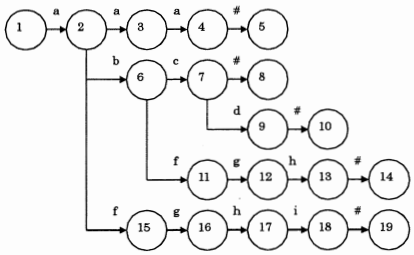


図 1　トライ構造

2.2. ダブル配列によるトライ構造

ダブル配列法はトライ構造の実装方式の一つであり，トライ構造を *Base* 配列と *Check* 配列という 2 つの配列で表現したものである．図 1と同等のトライ構造を表現したダブル配列を図 2(a)に示す．図 2(a)における *State* は，*Base* 配列および *Check* 配列のインデックスを示す．ダブル配列の各要素はトライ構造の 1 ノードに対応するため，State はノード番号とみなすことができる．*State* をノード番号とみなして，図 2(a)のダブル配列を図 1と同様の木構造で表現したものを図 2(b)に示す．

ダブル配列によるトライ構造の遷移は *Base* 配列と *Check* 配列の値を参照しながら行う．ダブル配列法では，辞書データ中に出現する各文字に対する内部コードを設定する．以下の説明では，文字集合を S_c ={“#”, “a”, “b”, …, “z”}，文字集合と 1 : 1 に対応する内部コード集合を S_{code} ={1, 2, 3, …, 27}とし，S_c から S_{code} への写像を *Code*()で表す．例えば *Code*(“b”)=3 となる．ダブル配列法による文字列検索では，まず初期 *State* を 1（ルートノード）とし，検索文字列と 1 文字ずつ照合しながら *State* を更新していくことでノードの遷移を行う．現在の *State* を *s*，照合する文字を *c* としたとき，遷移先の *State* となる *t* を下記の式(1)に従って求める．

$$t = Base[s] + Code(c) \quad (1)$$

このとき

$$Check[t] = s \quad (2)$$

であれば遷移は成功となる．以降は *s* に *t* の値を代入し，*c* には検索文字列の次の文字を代入して，上記の遷移を繰り返す．検索文字列の終端まで遷移し，最後に終端記号“#”で遷移できれば検索文字列が辞書に登録されていることがわかる．途中で式(2)が成立しない場合，遷移は失敗し，辞書には検索文字列が登録されていないことがわかる．

例えば図 2(a)のダブル配列から“abc”が登録されているかどうかを検索する場合，次のようにノードを遷移する．まず *State* が 1 の状態から開始し，“abc”の最初の文字“a”で遷移する．式(1)より *s*=1，*c*=“a”として *t* を求めると，

$$t = Base[1] + Code("a") = 1 + 2 = 3$$

となる．

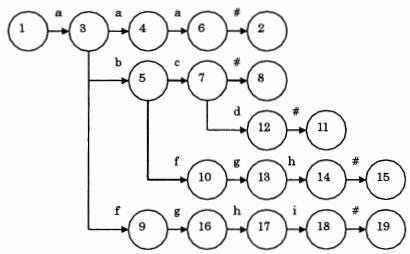
$$Check[t] = Check[3] = 1 = s$$

なので遷移は成功する．*s*=3，*c*=“b”として次の *t* を求め，以降同様に遷移すると *State* を 1，3，5，7 と辿ることができる．最後は終端記号”#”で *State*=8 に遷移する．終端記号の次は遷移がないため *Base*[8]=-1 としている．また各遷移において“#”での遷移が可能かどうか調べることで，検索文字列の前方部分文字列に一致する単語が辞書に登録されているかを知ることができる．上記ではアルファベットにより例示したが，日本語文字列の場合でも 1 バイト単位に区切って処理すれば，同様に扱うことができる．

このようにダブル配列はトライ構造を配列で表現したもので，辞書のエントリ数によらず計算量が $O(1)$ で検索できる．しかし，例えば「先頭が“a”で始まる単語をすべて列挙する」といった前方一致検索を行いたい場合，“a”の後ろに続く遷移可能な文字を知る術がないため，しらみつぶし的に探索する必要があり，高速な処理はできない．

State	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Base	1	-1	2	4	3	1	7	-1	8	5	-1	10	5	14	-1	8	8	17	-1
Check	0	6	1	3	3	4	5	7	3	5	12	7	10	13	14	9	16	17	18

(a)ダブル配列



(b)ダブル配列のトライ構造図示

図 2　ダブル配列

2.3.　入力サジェスト機能の実装における要求

我々は，組込機器上の文書検索用キーワードの提示に用いる大規模なキーワードを高速に検索するためのデータ構造としてダブル配列を用いる．目的とする入力サジェスト機能を実現するためには，次のような要求を満たす必要がある．

要求 1：　高速前方一致検索の実現
　1 文字や 2 文字の入力に対し，前方一致する登録単語を高速に列挙できる．
要求 2：　限られた表示領域での候補提示
　単語には重要度を示すスコアを事前に付与しておき，スコアの上位のみを出力できる．
要求 3：　省メモリ動作
　組込機器などの 1 次記憶容量が非常に限られている環境で高速かつ省メモリで動作する．

3.　関連研究

ダブル配列の改良に関する研究は単語の動的な追加・削除の効率化[5][6]を目的とするものが主である．主記憶上での高速化についてはキャッシュの効率化[7]によるものがあるが，組込機器上で動作させるためには，2 次記憶にダブル配列を格納することが必須となる．

2 次記憶への格納を考慮した研究としては分割ダブル配列法[8]がある．分割ダブル配列法ではダブル配列を所定のサイズでブロック化し，ブロック単位で主記憶に読み上げながら遷移を行う．検索時のブロックの転送回数（＝ディスク読み込み回数）ができるだけ少なくなるようにうまく分割することで，非常に少ない転送回数で検索できることが報告されている．また，ダブル配列を分割することにより *Base* 配列，*Check* 配列の要素のサイズを小さくすることができ，ダブル配列全体のサイズを分割しない場合の約 50%にできることも報告されている．

しかし，いずれの研究においても前方一致検索については考慮されていない．入力サジェストにおけるキーワード提示のためには 2 次記憶を用いた前方一致検索が必要であり，本研究の目的とは異なる．

4.　提案方式

この章ではダブル配列を改良し，高速に前方一致検索可能な辞書検索方式を提案する．以降，2.3節で述べた要求 1～3 に対する本提案方式の解決方法を述べる．

4.1.　最小・最大子ノード配列と単語リストによる前方一致検索

要求 1.に関して，“最小・最大子ノード配列”と“単語リスト”をダブル配列に導入することを提案する．最小・最大子ノード配列は，ダブル配列上の任意のノードから辞書順で最小となる単語の子ノードと最大となる単語の子ノードへの遷移を高速に処理できるようにするための情報である．単語リストは前方一致検索の結果を 2 次記憶から連続的に取得するための情報である．図 3(a)は最小・最大子ノード配列を追加したダブル配列の例である．*Child_First* 配列と *Child_Last* 配列が最小・最大子ノード配列である．*Child_First* 配列には各 *State* から遷移できる子ノードのうち，辞書順で最小となる単語に遷移するための内部コードを格納しておき，*Child_Last* 配列には辞書順で最大となる単語に遷移するための内部コードを格納しておく．一方，辞書に登録されている単語の全リストを辞書順にソートした上で，単語リストとして別に保存しておく．図 3(b)は単語リストの例である．また，ダブル配列上で単語の終端に対応する *State* の *Base* 配列の値を図 2(a)では-1 としていたが，図 3(a)では対応する単語の単語リスト上でのアドレス番号を負の値として格納している．

このように構成したダブル配列および単語リストから前方一致検索を行うには次のようにする．まず検索文字列の先頭から 1 文字ずつ，式(1)，式(2)にしたがってダブル配列上で遷移する．検索文字列の終端まで達したときの *State* を t とすると，t 以降は最小・最大子ノード配列に格納された内部コードを検索文字列の続きとみなして遷移し，辞書順で最小となる単語の W_f と最大となる単語の W_l を探索する．つまり，W_f を探索するには次の式(3)に従って遷移を行い，

$$t = Base[s] + Child\_First[s] \quad (3)$$

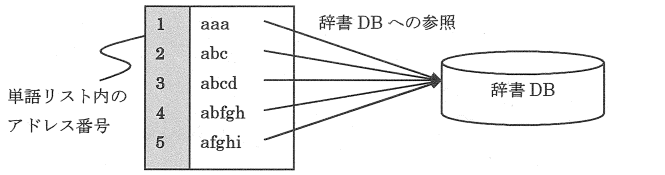
W_l を探索するには次の式(4)に従う．

$$t = Base[s] + Child\_Last[s] \quad (4)$$

遷移先の *Base* 配列の値 $Base[t]$ が負になるまでそれぞれの遷移を行い，負になったとき $Base[t]$ の値を単語リスト中のアドレス番号として取得する．単語リストは辞書順でソートされているため，単語リスト中で W_f と W_l の間に位置する単語はすべて検索文字列に前方一致する単語となり，単語リストを順次読み込むことで前方一致検索の結果を得られる．*Child_First* 配列と *Child_Last* 配列の各要素は内部コードを格納出来ればよいので，これらの配列のデータサイズはダブル配列全体のデータサイズと比較して大きくはならない．

State	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Base	1	-1	2	4	3	1	7	-2	8	5	-3	10	5	14	-4	8	8	17	-5
Check	0	6	1	3	3	4	5	7	3	5	12	7	10	13	14	9	16	17	18
Child_First	2	0	2	2	4	1	1	0	7	8	0	6	9	1	0	9	10	1	0
Child_Last	2	0	7	2	7	1	5	0	7	8	0	6	9	1	0	9	10	1	0

(a)拡張されたダブル配列



(b)単語リスト　　(c)詳細な辞書データベース

図 3　前方一致対応ダブル配列の構成

以下に前方一致検索の動作例を示す．

（例）のダブル配列と単語リストから“ab”に前方一致する単語をすべて取得する．

(1) *State*=1 から“a”（*Code*(“a”)=2）により遷移し，*Base*[1]=1 なので *State*=*Base*[1]+*Code*(“a”)=3 となる．*Check*[3]=1 なので遷移は成功する．

(2) *State*=3 から“b”（*Code*(“b”)=3）により遷移し，*Base*[3]=2 なので *State*=5 となる．*Check*[5]=3 なので遷移は成功する．

(3) 入力は終了したので，最小単語 W_f を探索するため *Child_First*[5]を次の入力の内部コードとみなし遷移する．*Base*[5]=3，*Child_First*[5]=4 なので式(3)から *State*=7 となる．

(4) *State*=7 から *Child_First*[7]を入力の内部コードとみなし遷移する．*Base*[7]=7，*Child_First*[7]=1 なので *State*=8 に遷移する．

(5) *Base*[8]=-2<0 なので単語の終端であり，これが W_f にアクセスするための単語リストのアドレス番号となる．2 を *First_Index* として記憶しておく．

(6) *State*=5 から前方一致する最大単語 W_l を探索するため *Child_Last*[5]を次の入力の内部コードとみなし遷移する．*Base*[5]=3，*Child_Last*[5]=7 なので式(4)から *State*=10 に遷移する．以降同様に *Child_Last* の値を入力の内部コードとみなし，State=15 に遷移する．

(7) *Base*[15]=-4<0 なので単語の終端であり，これが W_l にアクセスするための単語リストのアドレス番号となる．4 を *Last_Index* として記憶する．

(8) 単語リストの *First_Index* から *Last_Index* までを順次取得する．図 3(b)を参照すると，アドレス番号が 2 から 4 の単語集合{“abc”, “abcd”, “abfgh”}を得ることができる．

単語リストには *First_Index* から *Last_Index* まで連続的にアクセスすればよいため，ハードディスク上に配置されていても高速に処理することができる．説明を簡単にするため上記では単語リスト自体に単語文字列を格納する形式で示したが，実際には図 3(b)に示すように，各単語エントリの詳細情報や付加的情報を持つ辞書データベース図 3(c)への参照を持つことで，後述するスコアによるソート処理やデータの追加更新などへの柔軟な対応が可能となる．

4.2.　単語リストのブロック化による高速化

要求 2 に対しては，単語リストにスコアを持たせておくことで，上位抽出処理を容易にできる．例えばスコア上位 3 件を取得する場合，単語リストの *First_Index* から最初の 3 件を順次取得し出力候補とする．以降は単語リストを順次読み込み，出力候補中の最小スコアよりも大きい単語の場合は出力候補の最小スコアのものと置換し，小さい単語の場合は破棄すればよい．*Last_Index* まで読み込んだ後，出力候補の 3 件のみをソートして最終出力とする．

さらに単語リストを複数のブロックに分け，ブロック内での最大スコアを別途保存しておくことにより，より効率的な抽出が可能である．出力候補の最小スコアよりもブロック内の最大スコアが小さければそのブロックには出力候補に追加するべき単語が存在しないため，ブロックごと読み飛ばせばよい．図 4にブロック化した単語リストの例を示す．例えば“a”に前方一致する単語を検索した結果，図 4 の単語リストのアドレス番号 1～9 が前方一致単語として見つかったとする．スコアの上位 3 件を抽出する場合，まずブロック A から 3 件取得し出力候補とする．この時点での最小スコアは 55 となる．次にブロック B のブロック内最大スコアを参照すると出力候補の最小スコアより小さいためブロック B は読み飛ばし，ブロック C を参照する．ブロック C のブロック内最大スコアは 60 なので出力候補に

なる単語が含まれている可能性がある．ブロック C 内の単語を順次読み込み，出力候補の最小スコアと比較して大きいスコアを持つ単語を出力候補の最小スコアのものと置換する．例の場合は“abcd”を出力候補から削除して“akdef”を追加する．

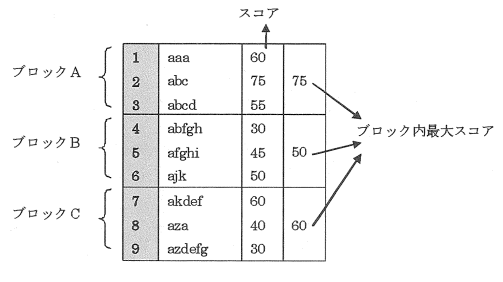


図 4　ブロック化した単語リスト

4.3.　組込機器対応

要求 3 に関してはダブル配列を 2 次記憶に保存し，部分的に 1 次記憶に読み込みながら値を参照することにより解決する．ただし部分的に読み込むダブル配列のバッファサイズを小さくしすぎると，1 回の検索中に何回も 2 次記憶へのディスクアクセスが発生するため全体のパフォーマンスが落ち，バッファサイズを大きくしすぎても遷移に使わない無駄な領域まで読み込んでしまい，オーバーヘッドとなってパフォーマンスが落ちる．そのため稼働環境ごとに適切なバッファサイズを設定する必要がある．

本稿における実験において，多い場合は 10 回以上のディスク読み込みが 1 回の検索中に発生しており，ディスク読み込みがボトルネックとなっていると思われる．そのため分割ダブル配列法を本提案方式に適用することにより，さらなる高速化が期待できるが，この点については本稿では将来の課題とする．

5.　実験

5.1.　実験環境

検索対象として以下の 4 つの辞書データを用い，読み仮名に対してダブル配列を作成した．

人名辞書　：日本人名
機関名辞書：日本の機関・組織名
全国住所　：日本全国の住所
総合　　：人名辞書，機関名辞書，全国住所の連結

それぞれの辞書の単語数，平均単語長，最大単語長を表 1 に示す．読み仮名は半角カナとし，1 バイト単位に区切ってダブル配列を構築する．ダブル配列の構築は PC 上で行い，検索時間の測定実験は当社組込機器製品（CPU 400MHz）上で行った．

表 1　検索対象データ：ただし読み仮名でユニーク化

辞書名	単語数	平均単語長	最大単語長
人名	418716	7.8	37
住所	122220	21.2	90
機関名	100556	13.2	71
総合	640945	11.2	90

5.2.　完全一致検索（予備実験）

本実験環境におけるダブル配列法の性能を把握し，以降の実験において適切なバッファサイズを設定できるようにするため，5.1節で示した辞書のそれぞれについて文字列の完全一致検索を行い，バッファサイズごとに検索時間を測定した．バッファサイズは 2KB，8KB，16KB，32KB，64KB，128KB とした．本実験環境では 1 回の検索に時間がかかり，辞書の全エントリで検索実験を行うのが困難であったため，検索文字列は対応する辞書から無作為に 10000 個抽出したものを用いた．それぞれの辞書における平均の検索時間を図 5，最大の検索時間（最も時間がかかったときの検索時間）を図 6に示す．全件による検索ではないため正確な性能を示せているわけではないが，1000 個ずつ抽出して同様の実験を行った場合もほぼ同じ結果が得られたため，基本性能を示すためには十分であると考える．

いずれの辞書の場合でも，バッファサイズが 8KB ないし 16KB のときが平均検索時間，最大検索時間ともに小さくなる傾向となった．これは，バッファサイズが小さい（2KB）ときはディスクアクセス回数が多くなりすぎてパフォーマンスが落ち，バッファサイズが大きいときは（32KB～128KB）無駄に読み込みすぎてオーバーヘッドとなっているためと思われる．

以上の結果から今回の実験環境ではバッファサイズは 8KB ないし 16KB が適切であると考えられる．最も単語数の多い「総合」辞書における平均検索時間はバッファサイズが 8KB のときに最小値 179ms となるため，以降の実験では 8KB をバッファサイズとして採用した．

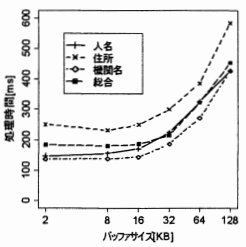


図 5　平均検索時間

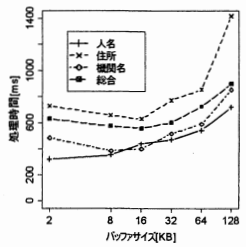


図 6　最大検索時間

5.3. 前方一致検索

本節では読み 1 文字（「ア」～「ン」，80 種）および 2 文字（「アア」～「ンン」，6400 種）を検索文字列としたときに，前方一致する単語を検索するのにかかる時間を測定する．ただし半角カナで検索するため，濁点・半濁点を持つ場合は 2 バイトで 1 文字とした．バッファサイズは5.2節の結果から 8KB とした．

提案方式と比較するため，従来のダブル配列法にもとづく次のアルゴリズム（以降，探索方式と呼ぶ）についても時間を測定した．

（探索方式）

(1) 検索文字列の終端まで式(1)，式(2)に従ってダブル配列上を遷移する．

(2) 辞書順の最小単語 W_f を探索するため，内部コードの小さい方から順にしらみつぶしに式(1)に従って遷移する．探索範囲は 256 通り（1 バイト）とする．式(2)が成立した場合，それが終端文字"#"であれば W_f の探索を終了する．終端文字以外であれば，遷移後の *State* から同様の処理を再帰的に行う．

(3) 辞書順の最大単語 W_l を探索するため，上記(2)と同様の処理を再帰的に行う．ただし，式(1)は内部コードの大きい方から適用する．

提案方式と探索方式について，W_f と W_l を発見するまでの時間を測定した結果を表 2，表 3に示す．探索方式に比べ提案方式は，1 文字の平均の検索速度が 2.5～6.8 倍，2 文字の平均の検索速度が 1.8～4.3 倍となった．提案方式において，いずれの辞書の場合も 1 文字の平均検索時間は 100ms 前後となっており，前方一致検索かつ 2 次記憶利用であっても検索時間が辞書の単語数によらないというダブル配列法の特徴は保たれていることがわかる．検索時間が大きくなるのは，前方一致する単語のうち辞書順で最小あるいは最大の単語の読み仮名が長い場合で，*Child_First* 配列あるいは *Child_Last* 配列を辿る際に時間がかかっている場合である．住所辞書において，最大検索時間が 614ms と他の辞書の最大検索時間の 2 倍程度になっているのは，住所辞書に登録されている単語の平均長さが他の辞書の 2 倍程度になっており，*Child_First* 配列あるいは *Child_Last* 配列を辿ってダブル配列上を遷移する際にディスク読み込みが多く発生するためである．

2 文字の検索において平均検索時間が 1 文字の場合よりも極端に小さくなっているのは，検索結果が 0 件となる検索も合わせて平均しているためである．入力サジェストにおいては，検索結果がないということ自体も重要な情報となる．

表 2 提案方式の検索時間：単位は ms

対象	1 文字		2 文字	
	平均	最大	平均	最大
人名	97	195	16	233
住所	106	614	8	562
機関名	107	278	13	200
総合	117	283	22	266

表 3 探索方式の検索時間：単位は ms

対象	1 文字		2 文字	
	平均	最大	平均	最大
人名	601	6592	69	11063
住所	724	13979	14	19025
機関名	269	6814	38	15509
総合	673	9524	94	11991

5.4. スコア上位抽出

本節では検索文字列を読み仮名 1 文字および 2 文字とし，前方一致するエントリのうちスコア上位の 5 件，10 件，20 件を抽出するのに必要な時間を測定した．辞書は最も単語数の多い総合辞書を用いた．4.2節で述べた単語リストのブロック化について，1 ブロック当りの最適な単語数は検討の余地があるが，今回の実験では 100 個とした．表 4にそれぞれの抽出数における平均検索時間と最大検索時間を示す．

平均検索時間は 190ms～196ms，最大検索時間は 355ms～383ms となっており，スコア上位抽出処理を含めても組込機器上で実用に耐えうる性能を示した．

参考までに同環境において，フリーの軽量データベースエンジンである SQLite[9] を用いて同様の結果を得るのに必要な時間を測定したところ，検索 1 回あたり約 260 秒程度を要した．これはスコアによる全件ソートを行うことにより，インデックスが有効に機能しないためと思われ，前方一致検索とスコアによる上位抽出を両立する本提案方式の有効性を確認できる．

表 4 スコア上位抽出を含めた検索時間：単位は ms

対象	1 文字		2 文字	
	平均	最大	平均	最大
上位 5	190	368	28	347
上位 10	196	383	29	374
上位 20	195	355	28	374

6. おわりに

本稿では，最小・最大子ノード配列と単語リストを利用した，組込機器上でも高速に前方一致検索可能な辞書検索方式を提案した．人名辞書，住所辞書，機関名辞書，およびそれらの連結である総合辞書を用いて組込機器上で検索実験を行い，その有効性を確認した．今回の実験では分割ダブル配列法を適用していないが，これを適用することによりディスクアクセス回数を減少させられると考えられ，さらに高速化を期待できる．

参考文献

[1] 相川勇之, 三上崇志, 平野敬, 岡田康裕: 文書の論理構造を用いたブートストラップ手法による重要語句の抽出, 2008 年電子情報通信学会総合大会, D-5-6(2008).

[2] 三上崇志, 相川勇之, 平野敬, 岡田康裕: 確率伝播法を用いた文書検索用キーワードの自動抽出, 情報処理学会研究報告. 自然言語処理研究会報告, Vol.2008, No.33, pp.1-6(2008).

[3] 青江順一: キー検索技法—トライ法とその応用, 情報処理学会誌, Vol.34, No.2, pp.244-251(1993).

[4] 青江順一: ダブル配列による高速ディジタル検索アルゴリズム, 電子情報通信学会論文誌, Vol.J71-D, No4, pp.1592-1600(1988).

[5] 森田和宏, 泓田正雄, 大野将樹, 青江順一: ダブル配列における動的更新の効率化アルゴリズム, 情報処理学会論文誌, Vol.42, No.9, pp.2229-2238(2001).

[6] 大野将樹, 森田和宏, 泓田正雄, 青江順一: ダブル配列におけるキー削除の効率化手法, 情報処理学会論文誌, Vol.44, No.5, pp.1311-1320(2003).

[7] 矢田晋, 森田和宏, 泓田正雄, 平石亘, 青江順一: ダブル配列におけるキャッシュの効率化, FIT2006, pp.71-72(2006).

[8] 泓田正雄, 柏木雄一郎, 樫地真確, 森田和宏, 青江順一: 2 次記憶上のダブル配列の効率的検索法, 電子情報通信学会論文誌, Vol.J85-D-I, No.11, pp.1028-1037(2002).

[9] SQLite: http://www.sqlite.org/