RD-001

遷移先節点に着目したダブル配列構造による探索の高速化の提案

Proposal to Speed Up Double-Array Retrieval Focused on the Reachable Nodes

村山 智也 †
Tomoya Murayama

望月 久稔 †
Hisatoshi Mochizuki

1. はじめに

ネットワークの普及や情報社会の発達により，我々が扱える情報量は増加し続けている．膨大な情報を活用するためには，高速な情報検索手法が必要となる．

高速な探索手法としてダブル配列が挙げられる．ダブル配列構造は，コンパクトかつ高速にキーを探索するために考案されたデータ構造である [1] [2]．また，ダブル配列構造はトライを実装するデータ構造であるため，探索の時間計算量はトライと同じくキー長に依存する．関連研究として，矢田ら [3] はキャッシュラインに着目して，節点情報の圧縮と遷移式の変更により探索を高速化した．

ダブル配列構造によるトライにキーを登録したとき，同じキー集合であれば一意のトライに定まる．しかし，登録するキーの順序によって，配列上の節点の配置が異なる．本稿は節点の配置方法に着目して，探索を高速化するダブル配列の構造を提案する．同じキー集合を登録していても配列内の構造が異なりえるため，配置を評価する指標として遷移距離を定義して，提案する構造へ変換するアルゴリズムを示す．既存のダブル配列構造を対象として，ダブル配列構造の変換前後で探索時間を比較して評価する．

2 章では，ダブル配列の遷移式によるキーの登録と節点配置を示す．3 章では，遷移距離を定義して効率的なダブル配列の構造を提案した後，提案する構造へ既存のダブル配列構造を変換するアルゴリズムを説明する．4 章では，提案する構造に実験的な評価を与えて，5 章で総括する．

2. ダブル配列と節点の配置

本章では，ダブル配列が異なった構造をとりえることを説明する．まず，ダブル配列の遷移式を示し，続いて，キーの登録順序により節点の配置が異なることを示す．

ダブル配列構造に登録されたキーは，トライと同じく，根節点から葉節点までの遷移によって表現される [1] [2]．ダブル配列構造は 2 本の配列 Base，Check から構成され，節点間の遷移は遷移式により表現する．ある遷移元節点 r から遷移種 a による遷移先節点 t への遷移が存在するとき，式 (1)，(2) が同時に成り立つ．$s = Base[r]$ としたとき，トライ図とダブル配列構造の節点の配置との対応を図 1 に示す．本稿では，遷移でつながる 2 節点の関係を遷移関係と呼ぶ．ダブル配列は，探索するキーを遷移種の列として解釈し，式 (1)，(2) を用いて根節点から遷移を繰り返すことで探索する．

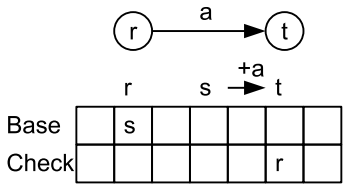


図 1: 節点間の遷移

$$Base[r] + a = t \quad (1)$$

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

キー集合 $K = \{$"$abba$","$abaa$","$abbc$","$abbba$","aaa"$\}$ を登録したトライ図と，ダブル配列による節点の配置を図 2 に示す．トライ図において，二重丸は葉節点，一重丸は葉でない節点を表し，節点の数字はダブル配列上の節点番号を表す．

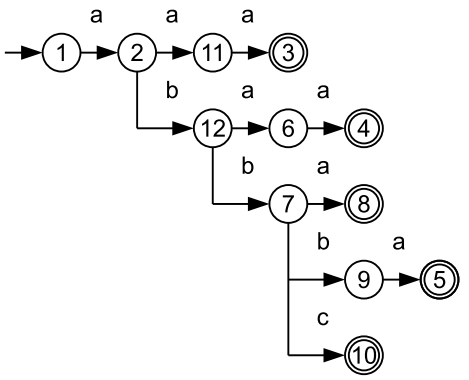
ここで，キー集合 K を並び替え，$\{$"aaa","$abaa$","$abba$","$abbba$","$abbc$"$\}$ の順序で登録する．登録したトライ図と，配列における節点の配置を図 3 に示す．図 2，図 3 はそれぞれ，同じキー集合 K を登録しているため，同じトライ図になる．しかし，キーの登録順序により，節点の配置が異なるダブル配列構造が構築される．したがって，同じキー集合を登録する際，出来上がるダブル配列の節点配置は異なりえる．

このような異なるダブル配列の構造における探索の遷移過程を示す．まず，図 2 のダブル配列に対し，キー"abba"を探索する際の遷移過程を示す．節点 1 から遷移種'a' により節点 2 に遷移する．以降，遷移種'b' により節点 2 から節点 12 へ，遷移種'b' により節点 12 から節点 7 へ，遷移種'a' により節点 7 から節点 8 へと遷移する．したがって，遷移過程は $1 \to 2 \to 12 \to 7 \to 8$ となる．

次に，図 3 のダブル配列に対しても同様に，キー"abba"を探索する際の遷移過程を示す．節点 1 から遷移種'a' により節点 2 に遷移する．以降，遷移種'b' により節点 2 から節点 6 へ，遷移種'b' により節点 6 から節点 9 へ，遷移種'a' により節点 9 から節点 10 へと遷移する．したがって，遷移過程は $1 \to 2 \to 6 \to 9 \to 10$ となる．

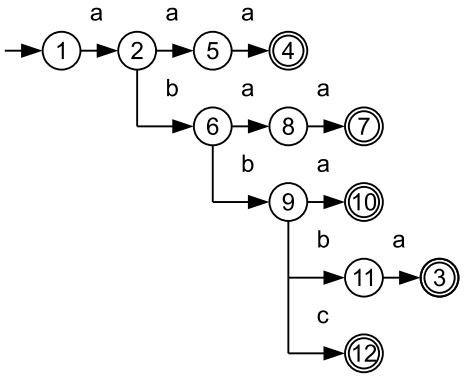
以上より，同一のキー集合を登録したトライであるが，節点配置が異なる 2 つのダブル配列（図 2，図 3）において，同じキーを探索する際，それぞれ遷移の過程が異なることが分かる．

†大阪教育大学, Osaka Kyoiku University



	1	2	3	4	5	6	7	8	9	10	11	12
Base	1	10	-1	-1	-1	3	7	-1	4	-1	2	5
Check		1	11	6	9	3	3	7	7	7	2	2

図 2: キー集合 K を表現するトライ図とダブル配列 1



	1	2	3	4	5	6	7	8	9	10	11	12
Base	1	4	-1	-1	3	7	-1	6	9	-1	2	-1
Check		1	11	5	2	2	8	6	6	9	9	9

図 3: キー集合 K を表現するトライ図とダブル配列 2

ダブル配列にキーを登録する際，節点の遷移先情報が配列上で衝突する可能性がある．節点を移動することで衝突を解決する [1] [2] が，キーの登録順により移動先が異なる．そのため衝突が発生することでダブル配列の構造は異なりえる．

3. 遷移距離を縮小した構造

本稿はダブル配列の構造に着目し，構造を定量的に扱うための指標として，3.1 節で遷移距離を定義する．この遷移距離と探索時間には相関があると考え，遷移距離の縮小により探索時間を短縮するダブル配列の構造を提案する．3.2 節に，提案する構造へ既存のダブル配列構造を変換するアルゴリズムを示す．

3.1. 遷移距離

本章では，キーの探索時間を短縮するため，以下の 2 点に着目した構造を提案する．

1. 遷移関係にある節点
2. 比較的分岐の多い節点の遷移先節点

図 2，図 3 のトライは形が同じであるため，道長は等しいが，対応するダブル配列の構造が異なり，探索の遷移過程は異なる．例えば，図 2 のダブル配列上で 11 番の節点位置に，図 3 では 5 番の節点が位置する．そこで，同じトライ構造でも配置の異なるダブル配列を区別するために，探索における遷移過程に基づいた指標を考える．

探索における遷移の過程は，2 節点間の遷移まで細分化できる．ここで，最小の遷移過程を表す指標として，遷移距離を式 (3) として定義する．

$$\text{遷移距離} = |\,\text{遷移先節点番号} - \text{遷移元節点番号}\,| \quad (3)$$

1 キーあたりの探索における遷移距離は，根から葉に至る遷移距離の和で表し，ダブル配列の探索性能は，登録された全キーの遷移距離の総和で評価する．これを式 (4) に定義する．

$$\text{遷移距離総和} = \Sigma\,\text{遷移距離} \quad (4)$$

例として，図 2 において節点 2 と節点 12 の遷移距離は 10 であり，全キーの遷移距離総和は 95 である．

本稿は，遷移距離の総和と探索時間の間に相関があると考える．そのため，着目点 1 について，遷移関係にある節点同士が近くに配置されていれば，離れて配置されている場合と比べて，探索が高速になると考えられる．ダブル配列の遷移式 (1) より，遷移先節点の配置は遷移元節点の Base 値に依存する．そこで，遷移距離を縮小するためには，遷移元節点の Base 値を，自身の節点番号に近い値にする必要がある．

また，ダブル配列構造はキー集合を木構造で表現するため，分岐が多い節点は，分岐が少ない節点に比べて，遷移の先につながるキーの数が多いと考えられる．よって，着目点 2 について，分岐が多い節点の遷移距離を縮小することで，比較的多くのキーの遷移距離を縮小できると考えられる．

以上を踏まえて，3.2 節では，遷移距離を縮小した構造への変換アルゴリズムを示す．

3.2. 遷移距離を縮小した構造への変換

既存のダブル配列から節点の遷移関係を取得して，提案する構造へ変換するアルゴリズムを示す．はじめにデータ構造を説明し，次に手続きを説明する．遷移距離を縮小するために，2 つの着目点に対し，以下のアプローチをとる．

1. 遷移関係にある節点を近くに配置する．
2. 比較的分岐が多い節点の遷移先節点を優先的に配置する．

1 つ目について，節点は番号の小さいものから詰めて配置するため，連続で配置する節点は近くに配置しやすい．そこで，スタックにより組の配置順を管理することで，直前に配置した節点の遷移先を，直後に配

手続き：Convert

(C-1) 旧ダブル配列構造の根節点と，新ダブル配列構造の根節点を組 rootTuple にする．
(C-2) 組 rootTuple を Sieve に渡す．
(C-3)Fetch によりスタックから組 tuple を取り出せる間，以下の処理を繰り返す．
　(C-3-1) 組 tuple を Locate に渡す．

図 4: 手続き：Convert

置しやすくする．そのため，遷移関係にある節点同士をなるべく近い位置に配置することで，遷移距離の縮小が期待される．

2 つ目について，分岐が比較的多い節点は，遷移先に比較的多くのキーが登録されている．この節点をハブ節点とし，式 (5) として定義する．

$$\text{ハブ節点} = \{node \,|\, node \text{の遷移先節点数} \geq \theta\} \quad (5)$$

ハブ節点の分岐度の閾値を θ とする．ハブ節点を配置後，その遷移先節点を小さい番号の節点に優先的に配置することで，探索全体における遷移距離の総和の縮小が期待される．ここで，配置する節点の間で優先順位を管理するために，スタックを 2 種類用意する．ハブ節点は HubStack に，それ以外の節点は ShaftStack に保存する．

提案手法は，変換前のダブル配列構造の節点番号と，変換後のダブル配列構造の節点番号を組にして管理する．本章では，節点 r_1, r_2 の組を組 $\{r_1, r_2\}$ と表記し，変換前の節点 r_1 を旧節点，変換後の節点 r_2 を新節点とよぶ．この組を用いて，既存のダブル配列構造を変換し，新しいダブル配列構造を構築する．そのため，配置の順を管理するスタックは，節点の組を要素にもつ．

変換アルゴリズムの手続きを説明する．手続き Convert を図 4 に示す．スタックには初期状態として，旧ダブル配列構造の根節点と，新ダブル配列構造の根節点の組を追加しておく（C-1,2）．Convert では，配置待ちスタックから組 tuple を取り出せる限り，組 tuple の新節点の遷移先節点を配置する（C-3,C-3-1）．新規に節点を配置するたび，その節点を新節点として含む組が，適切なスタックに追加される．以降，スタックが空になるまで節点の配置を繰り返すことで，ダブル配列構造を変換する．

手続き Sieve を図 5 に示す．ハブ節点は，多くのキーの遷移距離に影響するため，優先的に配置する HubStack に追加する．

手続き Fetch を図 6 に示す．Fetch は HubStack が空でない限り，HubStack から組を取り出す．Fetch を使って組を取り出すことで，ハブ節点の組を優先的に取り出す．

手続き Locate を図 7 に示す．Locate では，組 tuple の新節点に Base を設定することで，遷移先節点を配置する（L-1-1,L-1-2）．遷移先節点を配置する際，番

手続き：Sieve　引数：組 tuple

(S-1) 組 tuple の旧節点の遷移先節点数が θ 以上であるとき，
　(S-1-1)HubStack に組を PUSH する．
(S-2) そうでないとき，
　(S-2-1)ShaftStack に組を PUSH する．

図 5: 手続き：Sieve

手続き：Fetch

(F-1)HubStack が空でなければ，
　(F-1-1)HubStack から組を POP して返す．
(F-2)HubStack が空で，かつ ShaftStack が空でなければ，
　(F-2-1)ShaftStack から組を POP して返す．
(F-3)HubStack も ShaftStack も空であれば，
　(F-3-1) スタックが空であることを示す FALSE を返す．

図 6: 手続き：Fetch

号の小さい空き領域を使うように配置する．旧節点が葉節点であれば，新節点の Base には葉節点であることを示す Base 値を設定する（L-2-1）．

手続き Associate（図 8）は，節点の組 tuple と tuple の旧節点がもつ遷移パターン pat から，配置する新節点と対応する旧節点とを組として関連付ける．遷移パターンは式 (6) で表される遷移種集合を指す．

$$\text{遷移パターン} = \{a \,|\, Check[Base[r] + a] = r\} \quad (6)$$

変換後のダブル配列構造に配置する新節点は対応する旧節点を持つ．そこで Associate はそれらの節点を組とし，Sieve により適切なスタックに追加する（A-1-5）．また，Associate は新節点の遷移元節点と遷移をつなぐ（A-1-4）．

以上の手続きにより，3.1 節の着目点に基づいて遷移距離を縮小する例を示す．3.1 節の着目点 1 は例 1 を，着目点 2 は例 2 を使って説明する．以下の例では，$\theta = 3$ とし，遷移種は $a = 1, b = 2, c = 3$ とする．

[例 1] 遷移関係にある節点間の遷移距離の縮小

キー集合 K を登録したダブル配列構造（図 2）を変換する過程を示す．根節点の組 {1,1} を Sieve に渡す（C-1,C-2）．スタックから組 {1,1} を取り出し（C-3），Locate に渡す（C-3-1）．旧節点 1 の遷移パターンは {'a'} なので（L-1），空き領域の先頭である 2 に遷移先節点を配置するよう，新節点 1 の Base に 1 を代入する（L-1-1）．組 {1,1} を Associate に渡し（L-1-2），遷移先節点の組 {2,2} を Sieve に渡す（A-1-5）．旧節点 2 はハブ節点でないため（S-2），組 {2,2} は ShaftStack に追加する（S-2-1）．

手続き：Locate　引数：組 tuple

(L-1) 組 tuple の旧節点の遷移先節点数が 1 以上であれば
　(L-1-1) 旧節点の遷移パターンを満たす Base 値を，新節点の Base に代入する．
　(L-1-2) 組 tuple を Associate に渡す．
(L-2) そうでなければ
　(L-2-1) 旧節点の Base 値を，新節点の Base に代入する．

図 7: 手続き：Locate

手続き：Associate　引数：節点の組 tuple，tuple の旧節点の遷移パターン pat

(A-1)pat に含まれる全ての遷移種 a について，
　(A-1-1) 組 tuple の旧節点の a による遷移先節点を，組 tTuple の旧節点とする．
　(A-1-2) 組 tuple の新節点の a による遷移先節点を，組 tTuple の新節点とする．
　(A-1-3) 新ダブル配列構造に，組 tTuple の新節点を配置する．
　(A-1-4) 組 tTuple の新節点の Check に，組 tuple の新節点番号を代入する．
　(A-1-5) 組 tTuple を Sieve に渡す．

図 8: 手続き：Associate

スタックから組 {2,2} を取り出し（C-3），Locate に渡す（C-3-1）．旧節点 2 の遷移パターンは {'a','b'} なので（L-1），空き領域の先頭である 3，4 に遷移先節点を配置するよう，新節点 2 の Base に 2 を代入する（L-1-1）．組 {2,2} を Associate に渡し（L-1-2），遷移先節点の組 {11,3}，{12,4} を Sieve に渡す（A-1-5）．旧節点 11,12 はそれぞれハブ節点でないため（S-2），組 {11,3}，{12,4} は ShaftStack に追加する（S-2-1）．

スタックから組 {11,3} を取り出し（C-3），Locate に渡す（C-3-1）．旧節点 11 の遷移パターンは {'a'} なので（L-1），空き領域の先頭である 5 に遷移先節点を配置するよう，新節点 3 の Base に 4 を代入する（L-1-1）．組 {11,3} を Associate に渡し（L-1-2），遷移先節点の組 {3,5} を Sieve に渡す（A-1-5）．旧節点 3 はハブ節点でないため（S-2），組 {3,5} は ShaftStack に追加する（S-2-1）．組 {11,3} を取り出し，遷移先節点の組を追加したときのスタックの様子を図 9 に示す．

スタックから組 {3,5} を取り出し（C-3），Locate に渡す（C-3-1）．旧節点 3 は遷移先を持たないので（L-2），旧節点 3 の Base 値を，新節点 5 の Base に代入する（L-2-1）．

変換の前後で，キー"aaa"を探索する時の遷移距離の和が 18 から 4 に減少した．これは，キー"aaa"を構成する遷移過程 1 → 2 → 11 → 3 が，1 → 2 → 3 → 5 に変換されたためである．遷移距離総和も，28 から 6 に減少した．以上より，提案手法は，遷移関係にある節点同士の遷移距離を縮小している．　[**例終**]

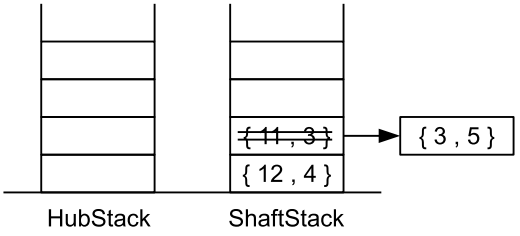
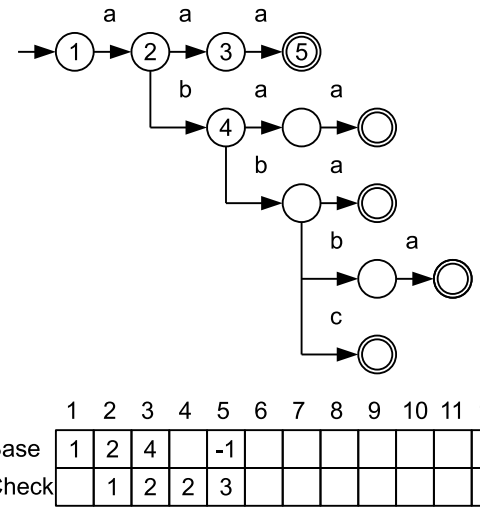


図 9: Sieve({3,5}) 後のスタック



	1	2	3	4	5	6	7	8	9	10	11	12
Base	1	2	4		-1							
Check		1	2	2	3							

図 10: 例 1 変換後

[例 2] ハブ節点から遷移先節点への遷移距離の縮小

スタックから組 {12,4} を取り出し（C-3），Locate に渡す（C-3-1）．旧節点 12 の遷移パターンは {'a','b'} なので（L-1），空き領域の先頭である 6，7 に遷移先節点を配置するよう，新節点 4 の Base に 5 を代入する（L-1-1）．組 {12,4} を Associate に渡し（L-1-2），遷移先節点の組 {6,6}，{7,7} を Sieve に渡す（A-1-5）．旧節点 6 はハブ節点でないため（S-2），組 {6,6} は ShaftStack に追加する（S-2-1）．旧節点 7 はハブ節点なので（S-1），組 {7,7} は HubStack に追加する（S-1-1）．

HubStack から組 {7,7} を取り出し（C-3），Locate に渡す（C-3-1）．旧節点 7 の遷移パターンは {'a','b','c'} なので（L-1），空き領域の先頭である 8，9，10 に遷移先節点を配置するよう，新節点 7 の Base に 7 を代入する（L-1-1）．組 {7,7} を Associate に渡し（L-1-2），遷移先節点の組 {8,8}，{9,9}，{10,10} を Sieve に渡す（A-1-5）．旧節点 8，9，10 はそれぞれハブ節点でないため（S-2），組 {8,8}，{9,9}，{10,10} は ShaftStack に追加する（S-2-1）．組 {7,7} を取り出し，遷移先節点の組を追加したときのスタックの様子を図 11 に示す．

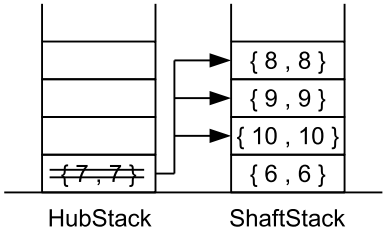


図 11: Locate({7,7}) 後のスタック

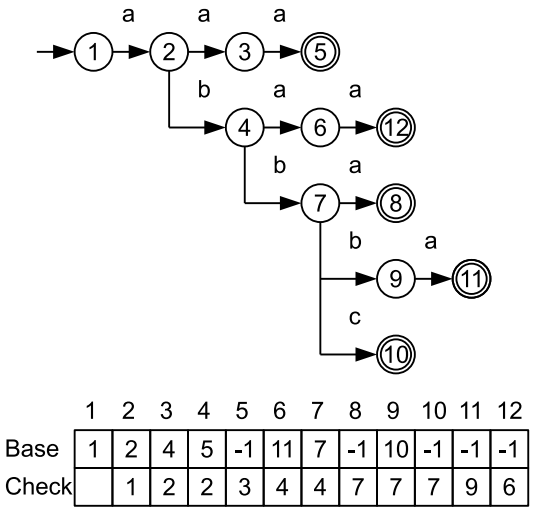


図 12: 例 2 変換後

以降同様に，組 {8,8}，{9,9}，{10,10}，{5,11}，{6,6}，{4,12} を配置する．組 {4,12} を配置した時，スタックが空となり，配置は終了する（C-3）．

構築されたダブル配列構造を図 12 に示す．接頭辞”abb”から始まる全てのキーを探索する際，変換の前後で，遷移距離の和は 58 から 26 に減少した．これは，ハブ節点の遷移先節点を優先的に配置することで，その先につながる多くのキーの遷移距離を縮小したためである．以上より，提案手法は，ハブ節点と遷移先節点の間の遷移距離を縮小している．変換により，遷移距離の総和は 95 から 41 に縮小された．したがって，提案手法はキー集合全体の遷移距離を縮小している．[**例終**]

4. 評価

提案手法を用いて変換したダブル配列構造を評価する．はじめに，ハブ節点の閾値 θ を決定して，変換による探索時間の短縮と遷移距離の縮小を評価する．最後に，探索時間と遷移距離との関係を考察する．変換する際，節点を高速に配置するために中村らの手法 [4] を，遷移先節点を高速に取得するために重越らの手法 [5] を使用する．実験マシンは，Intel Celeron G550 2.60GHz(L2 キャッシュ：512KB，L3 キャッシュ：2MB)，Fedora 14 32bit を使用する．実験では，Wikipedia [6] のページタイトルのうち，英小文字で構成されるものをキー集合として使用する．

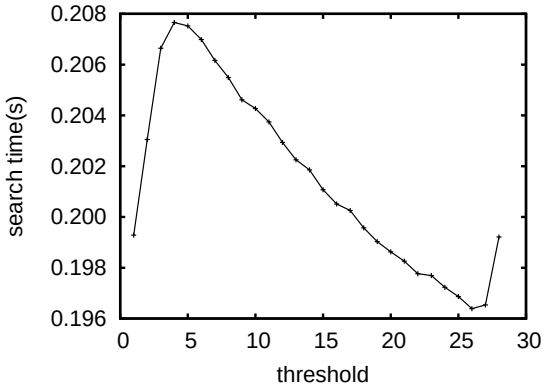


図 13: θ を変動させた際の探索時間

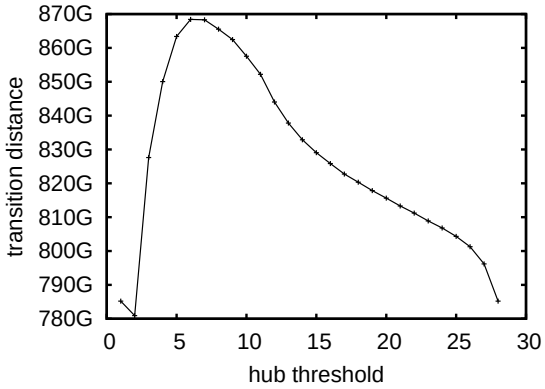


図 14: θ を変動させた際の遷移距離総和

提案手法は閾値 θ によりハブ節点を決定するため，θ は探索時間と遷移距離に影響すると考える．θ を変動して登録した 100 万語のキー集合を，全て探索する際の探索時間と遷移距離をそれぞれ図 13，図 14 に示す．図 13 において，θ が 26 の場合に探索時間が最短となった．図 14 において，おおむね遷移距離と探索時間の増減が対応していることが分かる．しかし，θ が 2，28 の場合，遷移距離と探索時間の増減は対応していない．遷移距離は探索時間におおむね影響するが，多少の変化は必ずしも影響しないと考えられる．以降の実験では，探索時間を最も短縮した θ=26 を採用する．

提案手法による変換の前後で探索性能を評価する．10 万語から 100 万語までのキー集合を登録して，全キーを探索する．探索時間を図 15 に示す．図 15 より，変換前の探索時間に比べて，変換後の探索時間は約 16〜17 %短縮されている．また，キー集合の大きさに関わらず，探索時間の短縮率はほぼ一定である．遷移距離の計測結果を図 16 に示す．図 16 から，変換前の遷移距離に比べて，変換後の遷移距離は約 82〜84 %の縮小が見られる．これは，遷移関係にある節点間の遷移距離と，ハブ節点と遷移先節点の遷移距離が提案手法により縮小されたためと考えられる．また，探索時間と

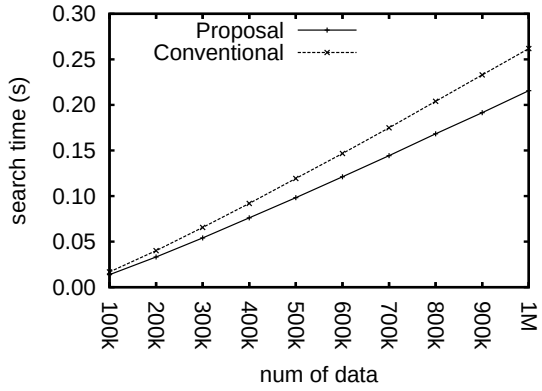


図 15: 変換前後の探索時間

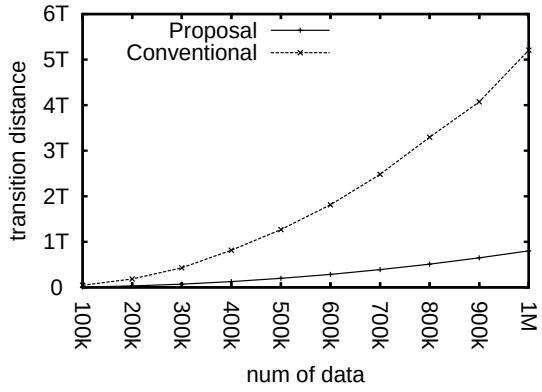


図 16: 変換前後の遷移距離総和

同じく，キー集合の大きさに関わらず，遷移距離の縮小率はほぼ一定である．

ここで，遷移距離と探索時間の相関関係について確認する．10 万語から 100 万語までのキー集合をそれぞれ 7 種類ずつ使用して，ダブル配列にキー集合を登録後，全キーを探索する．このときの遷移距離と探索時間の散布図を図 17 に示す．計測結果から，遷移距離と探索時間との相関係数は約 0.98 となり，正の相関を示した．

提案手法はダブル配列構造の遷移距離を縮小することで，節点の配置をメモリ上で近付けた．これにより遷移先情報がキャッシュに乗りやすくなり，探索の高速化につながったと考えられる．また，遷移情報を多く含むハブ節点同士をメモリ上で固めて配置した．これにより参照を局所化して，ハブ節点をキャッシュに保持しやすくする．ハブ節点は頻繁に使用するため，探索時間の短縮につながると考えられる．しかし，ハブ節点が多すぎるとキャッシュに乗りきらなくなるため，θ が小さい場合は探索時間が延長したと考えられる．根に近く，分岐が多い節点をハブ節点とすることで，より効率的にキャッシュを利用できる．したがって，遷移距離の縮小とハブ節点の優先配置とにより，探索時間の短縮が期待できる．

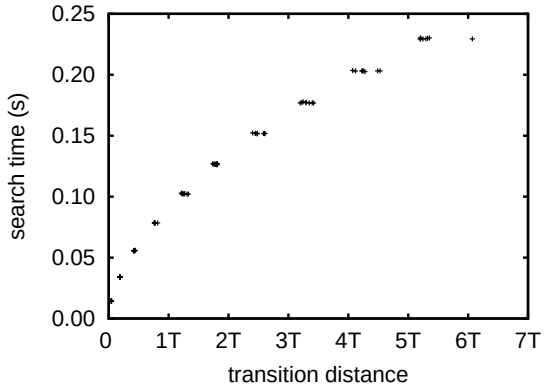


図 17: 遷移距離と探索時間の散布図

5. おわりに

本稿では，探索における効率的なダブル配列構造として，遷移関係にある節点とハブ節点の遷移先節点に着目した構造を提案した．その構造を定量的に評価する指標として，遷移距離と定義した．提案に基づいた構造への変換により，遷移距離を約 82〜84 %縮小し，探索時間を約 16〜17 %短縮した．また，遷移距離と探索時間との相関係数は約 0.98 となり，正の相関を示した．

今後の課題として，新規にダブル配列を構築する手法の設計と，節点をスタックに割り振る閾値 θ の決定方法が課題に挙げられる．

参考文献

[1] Jun-ichi Aoe：An Efficient Digital Search Algorithm by Using a Double-Array Structure, IEEE Transactions on Software Engineering, Vol.SE-15, No.9, pp.1066-1077, 1989.

[2] 青江順一，森本勝士：ダブル配列法によるトライ検索の実現法，情報処理学会研究報告自然言語処理，NL-085，pp.9-16，1991.

[3] 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，第 5 回情報科学技術フォーラム (FIT2006)，D-046，pp.71-72，2006.

[4] 中村康正，望月久稔：圧縮ディジタル探索木における辞書情報更新の高速化手法，情報処理学会:データベース，Vol.47，No.SIG 13 (TOD 31)，pp.16-27，2006.

[5] 重越秀美，蔵満琢麻，望月久稔，ダブル配列の遷移集合管理による追加・削除処理の高速化，第 8 回情報科学技術フォーラム (FIT2009)，RD-001，pp.1-6，2009.

[6] Wikipedia，https://ja.wikipedia.org/wiki/メインページ（参照 2014-04-09）.