自己適応最適二分探索木の研究

松川 理拓[1,a)]　山内　由紀子[2]　来嶋　秀治[2]　山下　雅史[2]

概要：大規模な計算機ネットワークの普及に伴い，大量のデータを効率的に管理，利用する手法が重要となっている．二分探索木を用いることで高速な検索が可能であるが，検索クエリ列の傾向をもとに木を再構築することで検索時間を短縮できる可能性がある．本稿では，二分探索木の各節点の深さと探索頻度から定義される二分探索木の期待探索コストに着目する．まず，期待探索コストを最小化する最適二分探索木を $O(n^3)$ 時間で求める動的計画法を示す．次に，Afek 達が提案した Lazy CBTree をもとに，探索時の木の回転によって逐次的に木の再構成を行う手法を提案する．更に，計算機実験により，既によく知られている自己適応木であるスプレー木，トリープ，Lazy CBTree と提案手法の性能比較を行い，提案手法が期待探索コストを改善することを示す．

キーワード：データ構造，二分探索木，自己適応

A Study on Self-Adjusting Optimal Binary Search Trees

Abstract: As large-scale computer systems become widely used, the techniques to manage and utilize large quantities of data becomes more and more important. Though binary search trees promise fast search procedure, we can improve the search time by adjusting the tree structure to the stream of search queries. We focus on the expected search cost of a binary search tree, which is defined by the depth and the frequency of each vertex. We first show an algorithm that computes the optimal binary search tree (i.e., tree with the minimum expected search cost) in $O(n^3)$ time. Then, based on the Lazy CBTree proposed by Afek et al. , we present a self-adjusting binary search tree that updates its structure by rotating the tree after each search query. We conducted a computer experiment to show the improvement of the proposed algorithm compared with the splay tree, the treap, and the Lazy CBTree.

Keywords: Data structure, binary search tree, self-adjustment

1. はじめに

近年，インターネットの拡大や電子端末の普及により，個人の活動が膨大な量の情報を生み出し，個人単位での情報の検索も日常的となっている．情報量の大幅な増加や，流行する情報の絶え間ない変化を伴う現代においては，必要な情報を高速に検索できることが重要である．二分探索木はデータの検索に対して効率が良く，広く使用されてきたが，より高速な検索を行う工夫が必要になってきたと言える．そこで，検索パターンに高速に適応し探索時間を短縮するために，本研究では探索を行った際に自動的に平衡化する二分探索木の木構造の収束について議論する．本研究の目的は，二分探索木の期待探索コストを回転操作を通して最適化することである．期待探索コストは，節点の二分探索木での深さと探索確率によって定義され，期待探索コストを最小化する二分探索木として最適二分探索木が知られている [3]．しかし，動的計画法を用いて最適二分探索木を計算することは可能ではあるが，計算時間が $O(n^3)$ であるため大規模データには不向きである．そこで，本研究では探索時の木の回転により，逐次的に木の再構成を行う手法に注目する．既存研究として，スプレー木 [1]，Lazy CBTree[2]，トリープ [3] などの探索時に自動的に木構造を平衡化する二分探索木が知られている．本研究では，Lazy CBTree をもとに，探索コストを改善するアルゴリズムを

[1] 九州大学工学部電気情報工学科
Department of Electrical Engineering and Computer Science, Kyushu University
[2] 九州大学大学院システム情報科学研究院)
Graduate School of Information Science and Electrical Engineering, Kyushu University
[a)] matukawa@tcslab.csce.kyushu-u.ac.jp

提案する．そして，要素の出現頻度に偏りのある検索クエリ列に対する各手法の性能を計算機実験に基づいて評価し，提案手法の期待探索コストが最適二分探索木の期待探索コストに最も近づいたことを確かめた．

2. 準備

本章では，二分探索木自体の定義，探索コストの定義などを行った後，本研究で用いたスプレー木，トリープ，Lazy CBTree のデータ構造及び最適二分探索木を求めるアルゴリズムを簡潔に記す．

2.1 二分探索木

二分探索木（Binary Search Tree）は，予め与えられたキー値を持つ節点によって構成され，キー値に関して以下の条件に従う二分木である．

定義 1. 二分木が以下の条件を満たす時，その二分木は二分探索木である．

全ての節点 v について，

(1) 左の子のキー値 $\leq$ 親のキー値，かつ

(2) 親のキー値 $<$ 右の子のキー値．

以降，本稿で扱う探索木は全て，定義 1 を満たすものとする．

次に，各節点 v に探索回数 $f(v)$ が与えられた時の，二分探索木 T の探索コストを定義する．T 上での節点 v の深さを $d(v)$ と表す．深さ $d(v)$ とは，根から節点 v までの枝数と定義する．根の深さは $d(root) = 0$ である．

定義 2. 二分探索木 T が N 個の節点で構成される時，

$$COST(T) = \sum_{v \in N} f(v) \times d(v)$$

を，その二分探索木の探索コストとする．

本研究では，二分探索木に含まれないキー値の探索は考慮しない．

2.2 スプレー木

スプレー木（Splay Tree）は Tarjan 達 [1] によって提案された自己適応木であり，検索クエリ列中に頻繁に出現する節点の探索時間の短縮を目的としている．スプレー木は，探索の際に**スプレー操作**と呼ばれる，発見した節点を木の回転操作により根の位置まで上昇させる操作を行う．スプレー木を構成する節点にはキー値の他に特別な情報を記憶させる必要はなく，自己適応木の中では比較的簡単に実装できる．

以下，スプレー操作の説明を行う．探索する節点を x，x の親を y，y の親を z とし，x の根からの深さを短縮する回転操作を y のスプレー木中での位置により場合分けをして説明する．木の回転操作を実行しても，木構造は定義 1 の二分探索木の性質を保っていることに注意していただきたい．

(1) zig 回転

y が根であり，かつ，x が y の左子供である時は，枝 (y, x) に対して一回の右回転を行う．

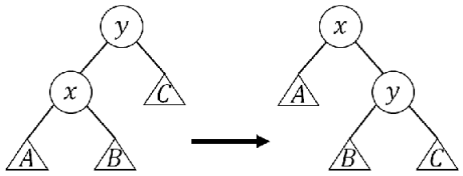

図 1 節点 x における zig 回転

(2) zig-zig 回転

y が根ではなく，y が親 z の左子供かつ x が y の左子供である時は，枝 (z, y)，枝 (y, x) に対して順番に右回転を行う．

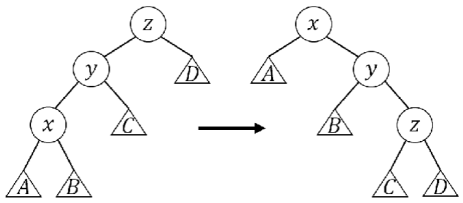

図 2 節点 x における zig-zig 回転

(3) zig-zag 回転

y が根ではなく，y が親 z の左子供かつ x が y の右子供である時は，枝 (y, x) に対して左回転を行った後，できた枝 (z, x) に対して右回転を行う．

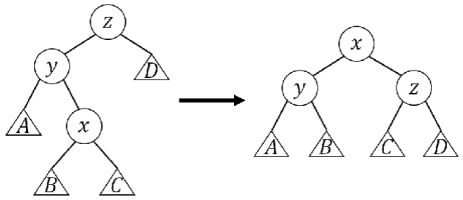

図 3 節点 x における zig-zag 回転

左右対称な木構造については説明を省略する．目的の節点を発見後，上記の回転操作を再帰的に適用することで探索した節点 x を根の位置まで上昇させる．

2.3 トリープ

トリープ（Treap）は，ツリー＋ヒープの造語であり，各節点がキー値の他に優先度を記憶することで，木の平衡化を図った自己調整木である [3]．トリープとは，キー値に関して二分探索木の性質を保ち，優先度に関してヒープ木の性質を持つ二分探索木である．

定義 3. 二分木が以下の条件を満たす時，その二分木はトリープである．

(1) 左の子のキー値 $\leq$ 親のキー値

(2) 親のキー値 < 右の子のキー値
(3) 親の優先度 ≥ 子の優先度

本研究では，各節点の優先度をその節点が探索された回数に置き換え，子の探索回数が親の探索回数を上回らないトリープを構築した．

本研究では新たに，木の回転によるトリープの維持手法を提案する．探索操作を行う際に子の探索回数が親の探索回数を上回った場合，図 4 に示す zig 回転を，探索された節点が親の探索回数を超えなくなる，または根となるまで実行することでトリープの条件を維持することができる．

定理 1. 探索操作により子の探索回数が親の探索回数を上回った場合，探索された節点が回転後に親となる節点の探索回数以下となる，または探索された節点が根となるまで *zig* 回転を繰り返し行うことで，トリープを維持できる．

証明. 探索した節点を x，x の親を p，x の右の子を c，y の親を g，各節点 v のカウンターの値を $v.selfCnt$ とする．x を発見した時，$x.selfCnt$ に 1 加算する．この加算により $x.selfCnt > p.selfCnt$ となった場合，枝 $(x,\ p)$ に対して一回の zig 回転を行うと，親が変更される節点は x，p，c の 3 点のみである（図 4）．探索前は $p.selfCnt = x.selfCnt \geq c.selfCnt$ であったことから，回転後に $p.selfCnt \geq c.selfCnt$ が成り立つ．また，$x.selfCnt > p.selfCnt$ であるのは自明である．よって，回転後にトリープの条件が成り立つかどうかの判定は $x.selfCnt$ と $g.selfCnt$ を比較するだけで行える．枝 $(x,\ g)$ 間でトリープの条件を満たしていなければ再び枝 $(x,\ g)$ に対して zig 回転を行えばよい．この操作を，(i) 枝 $(x,\ g)$ 間でトリープの条件を満たす，または，(ii)x が根となるまで繰り返すことで，探索の際に常にトリープの形を維持することができる． □

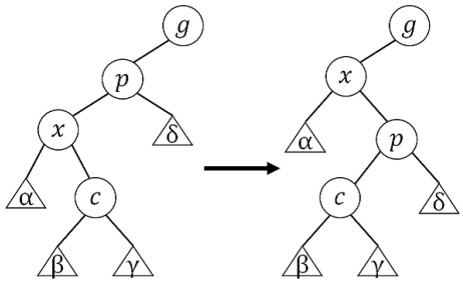

図 4 節点 x における zig 回転

2.4 Lazy CBTree

Lazy CBTree（Lazy Counting Based Tree）とは，2012 年に Afek 達 [2] の提案した自己適応木である．Lazy CBTree は二分探索木であり，各節点 v は，自身が探索された回数を記憶している．この回数を，$v.selfCnt$ と表す．また各節点 v は，その左部分木に含まれる全ての節点の $selfCnt$ の総和 $v.leftCnt$ 及び右部分木に含まれる全ての節点の $selfCnt$ の総和 $v.rightCnt$ を記憶する．

Lazy CBTree では，探索操作において目的の節点 x を見つけた後，x に対して次の条件を評価し，対応する回転操作を行う．以下では，節点 p を x の親，節点 c を x の右の子とする．

二重回転 (Double rotation) を行う条件

$$x.rightCnt \geq p.selfCnt + p.rightCnt$$

単一回転 (Single rotation) を行う条件

$$x.selfCnt + x.leftCnt > p.selfCnt + p.rightCnt$$

x が二重回転の条件を満たせば，x に対して図 5 の回転操作を施す．二重回転の条件を満たさず，単一回転の条件を満たす場合は x に対して図 6 の回転操作を施す．単一回転の条件も満たさないならば，回転操作を行わない．スプレー木やトリープとは異なり，回転操作が行われるのは一度のみである．二重回転及び単一回転の条件を満たす際に対応する回転を行うことで，探索コストを減少させる．

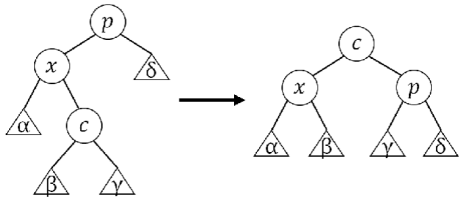

図 5 節点 x における二重回転

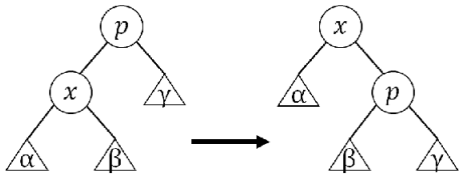

図 6 節点 x における単一回転

二重回転，単一回転を施した木について，以下の補題が示せる．

補題 2. Lazy CBTree T において，根ではない任意の節点 x が二重回転の条件を満たす時，x に対して二重回転を行って得られた木 T' について，以下が成立する：

$$COST(T') \leq COST(T).$$

補題 3. Lazy CBTree T において，根ではない任意の節点 x が単一回転の条件を満たす時，x に対して単一回転を行って得られた木 T' について，以下が成立する：

$$COST(T') < COST(T).$$

x から根までの各節点のカウンター値の更新は回転操作の後に行う（遅延更新）．Lazy CBTree では，読み込んだ検索クエリ列中の出現頻度の低い節点が根から遠い位置に格納される．総探索回数が少ない時点で検索クエリ列中の出現頻度が低い節点が現れた際，出現頻度が高い節点の探

索回数を上回ることがある．目的の節点が見つかった直後にカウンター値を更新すると，本来探索木中での根からの位置が深い節点が二重回転，単一回転の条件を満たしてしまい，一時的にその節点の探索木中での位置が上昇することがある．Afek 達はこのことを考慮して，遅延更新を導入したものと思われる．

2.5 最適二分探索木

最適二分探索木（Optimal Binary Search Tree）とは，探索確率が付加されたキー集合に対して，探索コストの期待値を最小化する二分探索木である [3].

定義 4. 節点 $k_i(1 \leq i \leq n)$ の探索回数を $f(k_i)$ とし，これらの n 個の節点によって構成される二分探索木を T とする．節点 k_i の T における深さ $d(k_i)$ を用いて二分探索木 T の期待探索コスト $E(T)$ を，

$$E(T) = \sum_{i=1}^{n} f(k_i) \times d(k_i)$$

と表したとき，

$$T^* = argminE(T)$$

を最適二分探索木と呼ぶ.

本研究では，二分探索木を構成する各節点の探索される確率を各節点が探索された回数に置き換え，定義 4 の最適二分探索木の期待探索コストの計算を行う．動的計画法を用いることで，節点数が n 個の二分探索木は $O(n^3)$ 時間で最小な期待探索コストを計算することができる．以下では文献 [3] の第 15 章を参考とした．

一般性を失うことなく，節点をキー値の昇順に整列してあると仮定できる．最適二分探索木に含まれる部分木は，i 番目から j 番目 $(1 \leq i \leq j \leq n)$ までの連続した節点によって構成されるので，節点を昇順に整列することで最適二分探索木の部分構造を考え易くなる．ここで，k_i から k_j までの節点により作られる部分木の期待探索コストを，

$$COST(i,j) = \sum_{l=i}^{j} f(k_l) \times d(k_l)$$

と表す.

最適二分探索木 T^* が部分木 T' を含む時，T' に含まれる頂点で作られる全ての二分探索木の期待探索コストは最小であり部分構造最適性を満たしていることをを示す．T' の期待探索コストが最小でないと仮定すると，部分木 T' よりも小さい期待探索コストを持つ部分木 T'' が存在することになり，T^* に含まれる部分木 T' を部分木 T'' に置き換えることで T^* よりも小さい期待探索コストを持つ二分探索木が作れることになる．これは，T^* が最適であることに矛盾する．よって，部分木の期待探索コストは最小であり，最適二分探索木に含まれる部分木は部分構造最適性を満たしている.

部分構造最適性より，$COST(i,j)$ が最小となる時の根を $k_r(i \leq r \leq j)$ とし，$W(i,j) = \sum_{l=i}^{j} f(k_l)$ とすると，

$$\begin{aligned} COST(i,j) = \min_{i \leq r \leq j} \{&COST(i,r-1) + W(i,r-1) \\ &+ COST(r+1,j) + W(r+1,j)\} \end{aligned}$$

ただし，$i = r$ では，

$$COST(i,r-1) = 0, W(i,r-1) = 0.$$

$r = j$ では,

$$COST(r+1,j) = 0, W(r+1,j) = 0. \tag{1}$$

と表される漸化式が得られる.

以上より，動的計画法を用いることで $O(n^3)$ 時間で最適二分探索木とその期待探索コストを計算可能である.

3. 提案手法

3.1 提案アルゴリズム

Lazy CBTree の探索コストの減少を目指す改良アルゴリズムを提案する．提案アルゴリズムの目的は，回転によって探索コストを既存の Lazy CBTree よりも減少させ，最適二分探索木のコストに近づけることである.

Lazy CBTree は，$x.rightCnt \geq p.selfCnt + p.rightCnt$ を満たす時に二重回転を行い，二重回転により二分探索木の探索コストを減少できることを補題 2 で証明した．しかし，厳密には探索された節点 x の子である c の $selfCnt$ を左辺に加えた，

$$x.rightCnt + c.selfCnt > p.selfCnt + p.rightCnt$$

を満たす時に二重回転を行うことで二分探索木の探索コストを減らすことができる．また，Lazy CBTree では二重回転の条件判定を行った後に単一回転の条件判定を行っており，二重回転が実行された場合は単一回転を行わない．だが，実際には二重回転の条件と単一回転の条件を同時に満たす場合があり，その際に，

$$x.selfCnt + x.leftCnt > x.rightCnt + c.selfCnt$$

を満たしていれば二重回転を行わず，単一回転を行う方が探索コストを減少できる．以上の Lazy CBTree の問題点を踏まえ，それを改善する提案手法の条件式を記す．なお，以降は提案手法を Detailed Lazy CBTree と呼ぶ.

(1) x における二重回転を行う条件式を，(2) 式に変更する.

$$\begin{aligned} &x.rightCnt + c.selfCnt \\ &\quad > p.selfCnt + p.rightCnt \end{aligned} \tag{2}$$

(2) 節点 x が条件式 (2) を満たす際に，以下の式を満たしていれば二重回転を行い，満たしていなければ単一回転を行う.

$$\begin{aligned} &x.selfCnt + x.leftCnt \\ &\quad \leq x.rightCnt + c.selfCnt \end{aligned} \tag{3}$$

Detailed Lazy CBTree は，Lazy CBTree を踏まえた上で考案しており，基本的な構造は Lazy CBTree と同じである．条件式 (2) 及び条件式 (3) について考察を行う．なお，補題 4，補題 5 において $selfCnt, leftCnt, rightCnt$ をそれぞれ $scnt, lcnt, rcnt$ と省略して表記する．

補題 4. Lazy CBTree T において，根ではない任意の節点 x が条件式 (2) を満たす時，x に対して二重回転を行って得られた木 T' について，以下が成立する：

$$COST(T') < COST(T)$$

証明． 補題 2 式より，二重回転を行う前後で，

$$\begin{aligned} &COST(T) - COST(T') \\ =\ &x.rcnt + c.selfCnt - p.selfCnt - p.rcnt > 0 \end{aligned}$$

が成り立つことが示されている．よって，条件式 (2) に従って二重回転を行うことで探索木のコストは小さくなる． □

観察 1. Lazy CBTree の節点 x における二重回転が，

$$x.rightCnt \geq p.selfCnt + p.rightCnt$$

で行われるのに対し，Detailed Lazy CBTree での節点 x における二重回転の条件式は，

$$x.rightCnt + c.selfCnt \geq p.selfCnt + p.rightCnt$$

であり，x の子である c の探索回数も考慮した条件式となっている．$c.selfCnt \geq 0$ でなので，Detailed Lazy CBTree では，二重回転を行う時の条件が緩和されると考えられる．よって，Detailed Lazy CBTree は，探索コストを減少させる機会が増えると期待される．

補題 5. Lazy CBTree T において，根ではない任意の節点 x が条件式 (3) を満たす時，x に対して二重回転を行って得られた木 T'' と，単一回転を行って得られた木 T' について，以下が成立する：

$$COST(T'') \geq COST(T')$$

証明． 図 5，図 6 より，二重回転を行った場合と単一回転を行った場合において根からの深さが変化するのは節点 x, c 及び部分木 α, β, γ のみである．二重回転を行った後の各節点，各部分木の根からの深さを基準に考えると，

$$\begin{aligned} &COST(T'') - COST(T') \\ =\ &x.scnt \times (d(x) - (d(x) - 1)) \\ &+c.scnt \times (d(c) - (d(c) + 2)) \\ &+ \sum_{w \in \alpha} w.scnt \times (d(w) - (d(w) - 1)) \\ &+ \sum_{w \in \beta} w.scnt \times (d(w) - (d(w) + 1)) \\ &+ \sum_{w \in \gamma} w.scnt \times (d(w) - (d(w) + 1)) \\ =\ &x.scnt - 2 \times c.scnt + \sum_{w \in \alpha} w.scnt \\ &+ \sum_{w \in \beta} w.scnt + \sum_{w \in \gamma} w.scnt \end{aligned} \tag{4}$$

また，回転前の $x.lcnt$，及び $x.rcnt$ の値より，

$$\begin{aligned} x.lcnt &= \sum_{w \in \alpha} w.scnt \\ x.rcnt &= c.scnt + \sum_{w \in \beta} w.scnt + \sum_{w \in \gamma} w.scnt \end{aligned}$$

これらを (4) 式に代入して，

$$\begin{aligned} &COST(T'') - COST(T') \\ =\ &x.scnt + x.lcnt - c.scnt - x.rcnt \end{aligned}$$

仮定より，

$$x.scnt + x.lcnt \leq x.rcnt + c.scnt$$

よって，

$$COST(T'') - COST(T') \leq 0$$

以上より，$x.scnt + x.lcnt \leq x.rcnt + c.scnt$ を満たす時に，二重回転を行って得られた木 T'' と単一回転を行って得られた木 T' について，

$$COST(T') < COST(T'')$$

が成り立つことが示された． □

4. 実験

各既存手法及び Detailed Lazy CBTree について，検索クエリ列に対する適応性を計算機実験により計測した．

4.1 実験条件

各実験における共通の条件を，以下のように設定する．

- 節点数 1000 個．ただし，各節点には事前に 1 から 1000 までのキー値が与えられている．
- 探索操作の前に，全ての節点をランダムに発生させた人気度順に挿入する．
- 探索操作は失敗しないものとする．
- 探索操作で用いた検索クエリ列は，Zipf 分布と呼ばれる分布を基に作成した．Zipf 分布とは，要素の総数を N とした時，人気度 k の要素の出現確率が以下の式を満たす確率分布である．

$$p(k; s, N) = \frac{1/k^s}{\sum_{n=1}^{N} 1/n^s}$$

各実験では，Zipf 分布のパラメータ s を $s = 0, 1, 2, 4$ とし，各 s の値についてランダムに発生させた検索クエリ列を 100 通り用意し，その平均値を示す．

4.2 適応性の測定

本研究の目的である検索クエリ列への適応性を調べた．

4.2.1 回転回数の収束

回転回数の収束について図 7 から図 10 にまとめた．縦軸は回転操作の通算の回数を，横軸は探索操作の回数を表している．$s = 0$ の検索クエリ列におけるグラフは，回転回数が多くなりすぎたため割愛した．各適応木毎にグラフの縦軸の値が異なることに注意していただきたい．

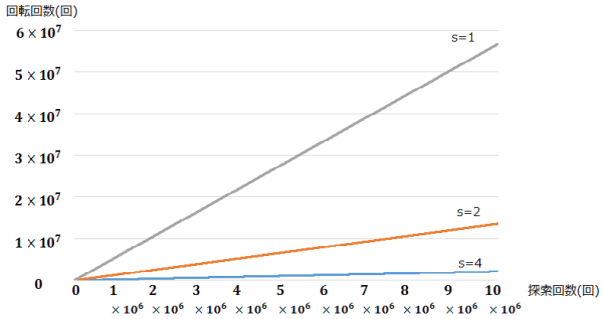

図 7 スプレー木における回転回数の収束

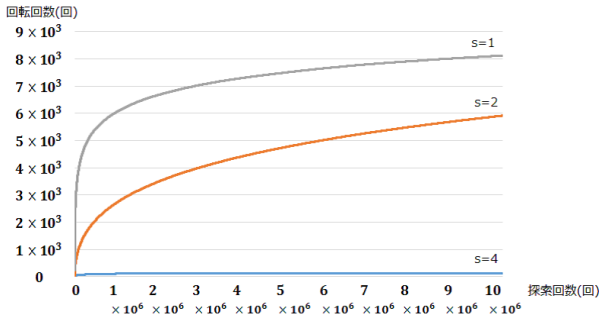

図 8 トリープにおける回転回数の収束

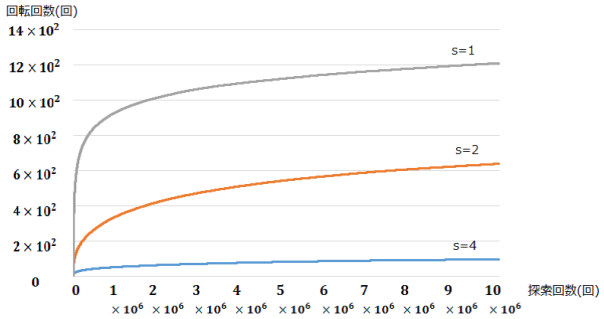

図 9 Lazy CBTree における回転回数の収束

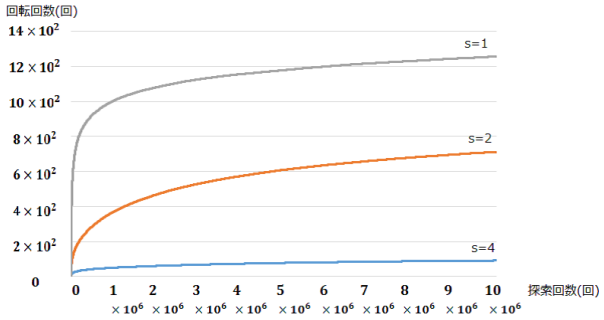

図 10 Detailed Lazy CBTree における回転回数の収束

4.2.2 期待探索コスト

予備実験により，各探索木に共通して探索回数が 200 万回に達した時点で回転操作を行う回数がおおよそ収束している（探索操作一回当たりの回転操作の回数が一定となる）ことを確認した．これらは全てのパラメータ s に関しても同じであった．そこで，探索操作を 200 万回行った後の期待探索コストを抽出し，これらの期待探索コストを同じ検索クエリ列を読み込んだ際の最適二分探索木の期待探索コストと比較した．表 2 には，最適二分探索木における期待探索コストに対する各適応木の期待探索コストを，近似比として示している．また，探索操作を行った際に，探索したい節点までの枝の本数が実際に減少しているのか否かを確かめるために，探索を行う度にその時に辿った枝の本数を計測した．表 3 に，探索回数が 200 万回に達した時点での，探索一回当たりの平均路長を示した．

表 2 各探索木における探索一回当たりの平均路長

		スプレー木	トリープ	Lazy CBTree	Detailed Lazy CBTree
s	0	11.70	10.91	8.14	8.12
	1	7.85	5.72	4.97	4.90
	2	1.62	0.94	0.91	0.90
	4	0.22	0.095	0.096	0.096

4.2.3 考察

表 2 より，期待探索コストが最適二分探索木の値に最も近かったのは Detailed Lazy CBTree であった．Lazy CBTree と Detailed Lazy CBTree の回転回数が収束するタイミングはほぼ同じであったので，より良い期待探索コストを示した Detailed Lazy CBTree の方が検索クエリ列への適応性が高いといえる．

また，表 3 より，Lazy CBTree と Detailed Lazy CBTree は $s = 0, 1$ においてスプレー木やトリープよりも平均路長を 1 割から 2 割程度減少させているので，次節で計測する探索時間の短縮が期待できる．パラメータの値が $s = 2, 4$ となると，トリープの平均路長は Lazy CBTree，及び Detailed Lazy CBTree のものと同程度であるので，スプレー木を除く各探索木の探索時間はどれも大差ないものと予想される．これは，検索クエリ列中に出現する要素の人気度に依る偏りが激しくなると，カウンター値に基づいて木構造の変形を行う各適応木の根の付近が，どれも最適二分探索木のものと近い構造になるからだと考えられる．

4.3 探索時間の測定

4.3.1 探索時間

本実験を行った実験環境を表 4 に，探索時間の測定結果を表 5 に示す．

4.3.2 オーバーヘッドの考察

s を変化させた 4 種類の検索クエリ列に対し，Detailed

表 1　各二分探索木の期待探索コストの実測値と近似比

		最適二分探索木		スプレー木		トリープ		Lazy CBTree		Detailed Lazy CBTree	
		実測値	近似比	実測値	近似比	実測値	近似比	実測値	近似比	実測値	近似比
s	0	15965811	1.000	23229170	1.455	21830490	1.367	16300340	1.020	16270250	1.019
	1	9468435	1.000	15553548	1.643	11443093	1.209	9950807	1.051	9816054	1.037
	2	1793310	1.000	3184862	1.776	1871517	1.044	1374250	1.019	1361924	1.010
	4	198681	1.000	422999	2.129	199351	1.003	199112	1.002	199086	1.002

表 3　実験環境

OS	Windows7 Enterprise
CPU	Intel Core 2 Duo 2.80GHz
メモリ	4.00GB
使用言語	C 言語
時間の計測法	C 言語の clock 関数

表 4　各探索木における探索時間 (秒)

		スプレー木	トリープ	Lazy CBTree	Detailed Lazy CBTree
s	0	188.059	127.971	144.954	145.158
	1	160.587	117.997	131.416	131.851
	2	115.847	108.031	110.356	110.698
	4	104.039	103.476	102.995	103.163

Lazy CBTree の探索時間は全ての場合において Lazy CBTree よりも僅かに遅くなった．Detailed Lazy CBTree の回転操作の条件判定では，探索された節点の子も参照しなくてはならない．Detailed Lazy CBTree では探索された節点が子を持っているか否かを if 文によりチェックするため，Lazy CBTree よりも一度の探索につき一回多く if 文のチェックが行われている．これがオーバーヘッドの原因になったと考えられる．

また，探索コスト，回転操作の回数，辿った枝数の面で Lazy CBTree や Detailed Lazy CBTree に劣っていたトリープであったが，探索時間に関しては $s = 0, 1, 2$ の検索クエリ列において他の適応木よりも高速であった．特に $s = 0, 1$ の検索クエリ列に対しては大きな差が現れている．Lazy CBTree と Detailed Lazy CBTree がトリープの探索時間よりも大きく遅れた原因について考察する．Lazy CBTree と Detailed Lazy CBTree では探索操作において目的の節点 x を探索した際に，x から根までの全ての節点の $selfCnt, leftCnt, rightCnt$ のいずれかを更新しなければならない．つまり，探索の際に辿った枝を，カウンター値の更新の為に再び辿り返す必要がある．表 3 より，$s = 4$ の検索クエリ列における探索の際に辿った枝数は Lazy CBTree で 0.096 本，トリープで 0.095 本であり，Lazy CBTree での枝数を二倍としてもその差は 0.097 本分である．しかし，$s = 0$ の場合は探索の際に辿った枝数が Lazy CBTree で 8.15 本，トリープで 10.91 本であり，Lazy CBTree での枝数を二倍にして差をとると 5.39 本となる．辿った枝の本数は探索操作一回当たりの平均値であるので，Lazy CBTree で 200 万回探索をした場合にはトリープでの探索に比べて $5.39 \times 2000000 = 10780000$（本）も多く枝を辿ることになる．このことが探索時間の遅延に繋がると予想し，次の実験を行った．

本実験は，探索操作が必ず成功するという条件で行っていた．そこで，Lazy CBTree 及び Detailed Lazy CBTree の探索アルゴリズムを，目的の節点を探索して枝を辿る際に，同時に各節点のカウンター値を更新するように変更して探索時間を計測した．トリープの探索時間は表 5 のデータを用いることとし，探索時間の比較を表 6 にて行う．

この結果より，Lazy CBTree 及び Detailed Lazy CBTree において探索時間の遅れが生じた原因が，各節点のカウンター値の更新によるものであると結論した．

表 5　カウンター値の更新を省いた場合の探索時間 (秒)

		トリープ	Lazy CBTree	Detailed Lazy CBTree
s	0	127.971	131.354	127.252
	1	117.997	123.146	120.322
	2	108.031	109.669	109.296
	4	103.476	103.242	103.505

5. おわりに

5.1 まとめ

本研究で扱ったスプレー木，トリープ，Lazy CBTree，Detailed Lazy CBTree について，期待探索コストを最適二分探索木のものと比較することで与えられた検索クエリ列への適応性を調べた．Lazy CBTree は，回転回数がおおよそ収束した時点での期待探索コストを，検索クエリ列中の要素の出現頻度の偏りによらず，最適二分探索木の値に大幅に近づけることができた．また，Detailed Lazy CBTree は Lazy CBTree で得られた期待探索コストをさらに減少させることに成功した．トリープの期待探索コストは検索列中の要素の出現頻度の偏りが大きい場合は最適二分探索木に近づくことが分かったが，偏りが小さい場合や一様な出現確率である場合には最適二分探索木の期待探索コストに対する近似比が大きくなった．スプレー木に関しては回転回数が収束せず，期待探索コストも最適二分探索木の値に近づかないため，最適二分探索木に自己適応するとは言い難い．

5.2 今後の課題

Lazy CBTree 及び Detailed Lazy CBTree の期待探索コストの面での適応が見られた一方で，検索クエリ列中の要素の出現頻度の偏りが小さい場合の探索時間については，比較的期待探索コストの大きいトリープの探索時間よりも悪化した．探索アルゴリズムに複数のカウンター値を導入することで最適二分探索木への自己適応をさせることは可能であったが，複数のカウンター値の更新のために生じるオーバーヘッドにより，操作時間は遅延したと考えられる．今後は，検索クエリ列への適応性を保ちつつ，各節点が記憶するカウンター値の更新回数を減少させることが課題である．

参考文献

[1] D.D. Sleator, and R.E. Tarjan, Self-Adjusting Binary Search Trees, Journal of the Association for Computing Machinery, Vol.32, No.3, pp.652–686, 1985.

[2] Y. Afek, H. Kaplan, B. Korenfeld, A. Morrison, and R.E. Tarjan, CBTree: A Practical Concurrent Self-Adjusting Search Tree, In Proc. of DISC 2012, pp.1–15, 2012.

[3] T.H. コルメン, C.E. ライザーソン, R.L. リベスト, C. シュタイン (共著), 浅野哲夫, 岩野和生, 梅尾博司, 山下雅史, 和田幸一 (共訳), アルゴリズムイントロダクション第 3 版, 近代科学社, 2013, pp.330–337.