論　文

移動オブジェクトの更新に適した領域分割形木構造：kdm-tree

西川　嘉人[†]　辻　俊明[†]　金子　裕良[†]　阿部　茂[†]

kdm-tree: A Region Splitting Tree Structure Suitable for Frequent Update of Moving Objects

Yoshihito NISHIKAWA[†], Toshiaki TSUJI[†], Yasuyoshi KANEKO[†], and Shigeru ABE[†]

あらまし　GPS 等の位置情報取得技術と移動体通信技術の進歩に伴い，車や電車，人といった時々刻々位置の変化する物体（以下，移動オブジェクトと呼ぶ）の位置情報の継続的な入手が容易となってきた．これに伴い移動オブジェクトの運行効率の向上，セキュリティ管理，位置に基づく情報提供などの応用が検討され，移動オブジェクトに適したデータ管理構造の研究が行われている．研究は目的により大きく，(1) 最新位置管理，(2) 未来位置予測，(3) 軌跡管理，の三つに分けられる．本論文では (1) の最新位置管理を扱う．最新位置管理では，一般に 1 万以上の移動オブジェクトの最新位置を，秒や分単位で更新する必要があり，この頻繁な更新による処理コストがデータ管理構造の大きな問題となる．筆者らはこの解決のためには，(1) 領域分割形の多次元データ管理構造が適している，(2) データの削除と挿入が同時に起こるためデータ削除の負荷が小さい構造が必要である，と考えた．本論文ではこれらを備えたバランス M 分木 kdm-tree (k-d tree for moving objects) を提案する．本文では kdm-tree のアルゴリズムと特徴を述べ，他の木構造との性能比較実験を行い，kdm-tree が更新性能に優れ，メモリコストや検索性能も同等以上であることを示す．

キーワード　移動オブジェクト，最新位置管理，更新，木構造

1. まえがき

GPS 等の位置情報取得技術と無線通信技術の進歩により，車や電車，人といった時間によって位置の変化する移動オブジェクトの位置情報を継続的に取得することが容易となり，位置情報を用いた運行管理やセキュリティ管理，情報配信などのサービス (Location Based Service: LBS) が検討されている．これらの応用ではある地域に分布し，時々刻々位置を変える膨大な数の移動オブジェクトから対象オブジェクトを位置により高速検索する必要があり，移動オブジェクトに適したデータ管理構造の研究が行われている．移動オブジェクト管理の研究は大きく 3 種類に分けることができる．第 1 は最新位置情報を管理するもの [1]〜[5]，第 2 は未来位置予測を行うもの [4]〜[7]，第 3 は軌跡を蓄積管理するもの [8]〜[10] である．

[†] 埼玉大学大学院理工学研究科数理電子情報専攻，さいたま市
Mathematics, Electronics and Infomation Division, Graduate School of Science and Engineering, Saitama University, 255 Shimookubo, Sakura-ku, Saitama-shi, 338–8570 Japan

本論文では第 1 の最新位置情報管理を扱う．最新位置管理では多数の移動オブジェクトの秒や分単位の位置変化に伴って生じる膨大なデータ更新処理が問題となる．大量の多次元データを高速検索する代表的データ管理構造として R-tree [11] が広く用いられてきたが，R-tree は本来大規模 CAD や地理情報処理などの静的データの管理を目的としており，移動オブジェクトの頻繁なデータ更新には適していない．このため更新処理を高速化（処理コストを削減）するための様々な手法が提案されている．

D. Kwon ら [1] は葉ノードへの直接リンクを用いることで更新時の木構造をたどる処理を削減した．M. L. Lee らによる R-tree Bottom-Up Approach（以下，RBU-tree と略す）[2] は更新時に木構造を葉ノードから根ノードに向かってたどる手法を一般化した．X. Xiong ら [3] は削除をすぐに行わないことで木構造へのアクセスを低減した．R-tree 以外でも C. S. Jensen ら [4] は多次元を一次元に圧縮することで多次元の問題を単純化し，高速化を行った．S. Guo らの提案した Buddy*-tree [5] は領域を等分割する単純な木構造

を用いることで木の高速構築を行った．しかし R-tree を用いた手法 [1]〜[3] はデータを MBR (Minimum Bounding Rectangle) で包括管理するため，位置変化に伴う木構造変化が避けられない問題がある．また多次元を一次元に圧縮する手法 [4] や単純な等分割を行う手法 [5] には不均一なデータ分布に対して非効率な木構造が作成され，最悪メモリコストを保証できない問題がある．

本論文ではデータの頻繁な更新に適した領域分割形バランス木構造 kdm-tree (k-d tree for moving objects) を提案する．kdm-tree は k-d tree [12] を拡張した構造で，木の高さがバランスし，任意の数の子ノードをもつ M 分木である．新たなノード分割の手法を取り入れることにより，領域統合が容易となり，少ない処理量で最悪メモリコストを保証することができる．領域分割形であるため R-tree で問題となる構造変化は少なく，領域に重複もない．

2. 更新管理に適した多次元木構造

最新位置管理において移動オブジェクトは点データで表され，その分布は一様ではなく時間とともに変化する．そのため頻繁な更新を高速に処理可能で，かつ変化に対して安定した性能をもつデータ管理構造が必要である．従来より，大量の多次元データを高速検索する索引構造として k-d tree や R-tree に代表される多次元木構造が広く用いられており，移動オブジェクト管理においても多次元木構造が利用される．移動オブジェクトの更新管理の際，多次元木構造に求められる要件として以下に示す三つが考えられる．もちろん検索性能，木の構築時間など基本性能を備えた上でのことである．

(a) 管理領域が重複しない領域分割形であること

(b) バランス M 分木で最悪メモリコストが保証されること

(c) データ削除処理の負荷が小さい構造であること

ここでメモリコストとは木構造作成に必要な記憶領域を指し，最悪メモリコストの保証とは管理データ数によって使用する記憶領域の最大値が定まることである．(a)〜(c) の特徴を備えることで挿入・削除時の木構造の変化を抑えて更新を高速化するとともに，更新によって木構造が劣化するのを防ぐことができる．各要件について述べる．

(a) 管理領域が重複しない領域分割形

木構造のデータ管理方式は k-d tree や quad tree [13] のような領域分割形と R-tree のようなデータ包括形に大別できる．領域分割形は階層的に全データ空間を分割し，それぞれの領域にデータを割当管理を行う．データ包括形はデータを囲む最小長方形 MBR で領域を管理し，MBR によって階層的な管理を行う．

領域分割形はデータ包括形に比べて分割が単純なため構築が高速で，空間的な余裕があるためオブジェクトの移動を木構造の変化なしに処理できる利点がある．また上位ノードの領域も重複はないか，あってもわずかである．領域の重複は挿入・検索・削除において不要な処理コストを増加させるのでない方がよい．

これに対しデータ包括形は MBR が移動オブジェクトの位置で決まるため移動に伴い変化しやすく，上位ノードへの伝搬も生じやすい．また上位内部ノードの領域の重複も多い．R-tree をベースにする手法はこの負荷を減らすことが主目的となっている．以上より移動オブジェクトの最新位置の更新管理には領域分割形が適しているといえよう．

(b) バランス M 分木と最悪メモリコストの保証

データ分布が不均一かつ変化する移動オブジェクト管理において検索性能を保つために木構造はバランスすることが必要である．また，最大子ノード数が 2〜4 のような少数であった場合，内部ノード分割や統合の頻度が増加して処理コストが増大する問題があるため最大子ノード数は大きい方がよい．頻繁な更新が起こる場合，葉ノード内のデータ数変化に応じて木構造を拡大・縮小させる必要があり，木構造が劣化してメモリコストが増加することから最悪メモリコストの保証が重要となる．例えば領域を等分割する quad tree や Buddy*-tree は最悪メモリコストの保証ができない．

(c) データ削除処理の負荷が小さい構造

移動オブジェクトの更新管理では大量の削除要求が生じるため，それに応じて木構造を高速に変化させる必要がある．多次元木構造は元々静的データを対象としていたため，削除アルゴリズムが十分考慮されていない．最悪メモリコストを保証しない構造は削除要求に対して十分なノード削減が行われず，最悪メモリコストを保証する構造は削除処理コストが大きい．移動オブジェクト管理においては最悪メモリコストを保証し，かつ削除処理コストの小さい手法が求められる．MBR を用いる手法は場合によっては再帰的に上位内部ノード MBR を調整して MBR を最小に保つ必要があり処理コストが大きい．領域分割形で最悪メモリコストを保証する MD-tree [14] は 2-3 木であるためデー

表 1　木構造性質の比較
Table 1　Comparison of tree structure.

	k-d tree	quad tree	K-D-B tree	MD-tree	R-tree	RBU-tree	Buddy*-tree	kdm-tree
データ管理方式	領域分割	領域等分割	領域分割	領域分割	データ包括	データ包括	領域等分割	領域分割
領域の重複の有無	無	無	無	少し有	有	有	無	無
木のバランス	×	×	○	○	○	○	○	○
最大子ノード数	2	4	M	3	M	M	M	M
最悪メモリコストの保証	○	×	×	○	○	○	×	○

タ削除時に内部ノード統合が頻繁に起こり，処理コストが大きい．

表 1 は各木構造の性質を比較したものである．k-d tree や quad tree は領域分割形であるが木構造がバランスしない．領域分割形でバランスする K-D-B tree [15] や Buddy*-tree は最悪メモリコストの保証ができない問題がある．また領域分割形でメモリ効率向上をねらった MD-tree は 2-3 木であるため挿入・削除時の処理コストが大きい．R-tree や R-tree を基盤とした RBU-tree はデータ包括形であるため更新時の木構造変化が大きく，頻繁な更新管理において改善の余地がある．

本論文で提案する kdm-tree は上記の要件をすべて満たし，最新位置の更新を高速に行うことができ，メモリコストや検索性能も他の手法と同等以上である．

3. 提案手法 kdm-tree

kdm-tree (k-d tree for moving objects) は k-d tree を最悪メモリコストを保証するバランス M 分木に拡張し，削除処理の負荷が少ない分割方式を用いている点に特徴がある．また移動オブジェクトの更新管理を高速にする葉ノードへの直接リンクも採用している．

3.1 kdm-tree の構成

図 1 に kdm-tree の領域分割とそれに対応するデータ構造（最大子ノード数 M=3 の場合）を示す．kdm-tree では図 1 (a) に示すようにデータ分布に応じて管理長方形の分割を行い，管理長方形に重複はない．図 1 (b) に示すように木構造の高さはバランスし，最大 M 個の子ノードを管理する M 分木となる．図 1 (b) 中の直接リンクによってデータを管理する葉ノードへ木構造を介さずにアクセスし，更新を高速化する．この葉ノードへの直接リンクは D. Kwon ら [1] が提案し，RBU-tree でも用いられている．

kdm-tree は葉ノードと内部ノードから構成される．両ノードを区別しないときは単にノードと呼ぶ．葉ノード L の構成要素は以下の四つである．

$$L = (parent, n_l, data[P], R)$$

ここで $parent$ は親ノードへのポインタ，n_l は L 中のデータ数，$data[P]$ は P 個のデータを格納できるスロットあるいはデータへのポインタ，P はバケット容量と呼ばれる任意数，R は L に割り当てられた管理長方形である．葉ノードは初期を除いて $minP(\leq P/2)$ 以上のデータをもつ．

内部ノード N の構成要素は以下の四つである．

$$N = (parent, n_n, child[M], R)$$

ここで $parent$ は親ノードへのポインタ，n_n は N の子ノード数，$child[M]$ は子ノードを指すポインタ M 個を格納するスロット，M は最大子ノード数，R は N の子ノードに割り当てられた管理長方形を包括する管理長方形である．内部ノードは根ノードを除いて $minM(\leq M/2)$ 以上の子ノードをもつ．

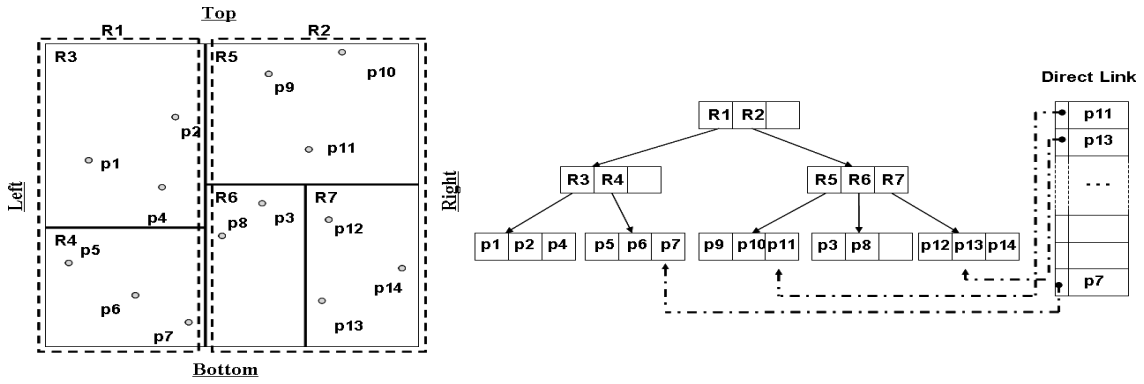
葉ノード，内部ノードの $parent$ は更新時に葉ノードから根ノードへたどるときに必要となる．$minP$ と $minM$ の値は小さいほど更新が高速になるが，葉ノードや内部ノードが増加するため，速度とメモリコストのトレードオフの関係にある．

ここで直接リンクは移動オブジェクトの ID をキーとし，そのオブジェクトが存在する葉ノードへのポインタを保持するハッシュテーブルである．

3.2 Critical Line による管理長方形分割

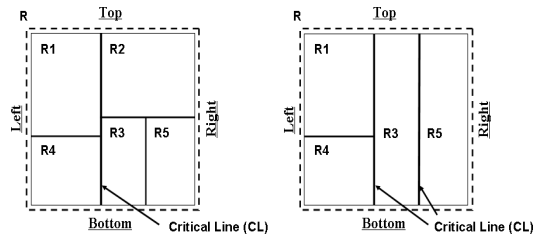
M 分木では内部ノードの分割時，その子ノードの分配方法が重要となる．例えば K-D-B tree は強制的な分割線により分配し，R-tree は面積和が小さくなるよう分配する．本研究では内部ノードの管理長方形を二分割する（Top と Bottom 若しくは Left と Right を結ぶ）線分を Critical Line (CL) と呼び，kdm-tree のノード分割に利用する．kdm-tree の子ノード管理長方形は親ノード管理長方形の繰返し分割により作られるため，すべての内部ノードにはその管理長方形を二分割する線分 (CL) が必ず一つ以上存在する．

図 2 に kdm-tree の管理長方形の例を示す．R は親



(a) Spatial index of kdm-tree (b) kdm-tree's structure

図 1 kdm-tree の領域分割と索引構造
Fig. 1 kdm-tree's region splitting and index structure.



(a) Example 1. of kdm-tree rectangle (b) Example 2. of kdm-tree rectangle

図 2 kdm-tree の管理長方形例
Fig. 2 Eaxmple of kdm-tree management rectangle.

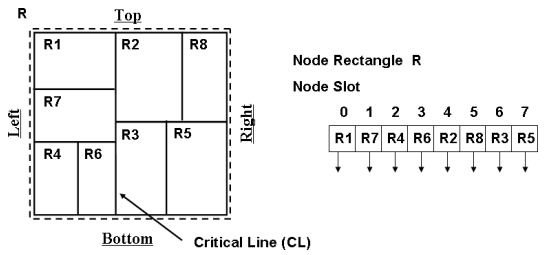


図 3 内部ノードスロット内の子ノード並び順序と長方形位置の関係
Fig. 3 Relationship between nodeslot and rectangle.

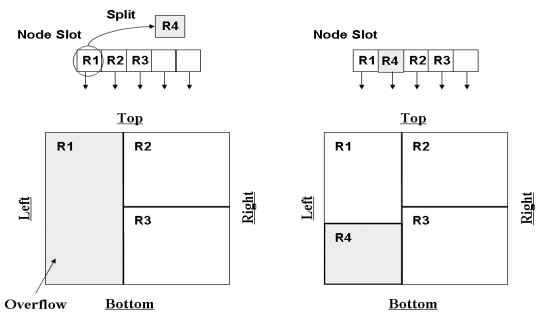
ノードの管理長方形，$R1$〜$R5$ はその子ノードの管理長方形である．図 2 (a) の R に対する CL は $R1$ と $R4$ の Right 辺で作られ，子ノードを二つのグループ $\{R1, R4\}$ と $\{R2, R3, R5\}$ に分ける．

図 2 (b) は図 2 (a) において R2 が削除されたときの領域分割であり，CL が 2 本存在する．内部ノードが複数の CL をもつ場合，子ノード数をよりバランスよく分ける CL を選び，後述するノード分割アルゴリズムに用いる．図 2 (b) の R に対する CL は，$R3$ の Left 辺と Right 辺にあたる線分となる．図 2 (b) の例で R を分割する場合，$R3$ の Left 辺を CL に用いると $\{R1, R4\}$ と $\{R3, R5\}$ に分けられ，$R3$ の Right 辺を CL に用いると $\{R1, R4, R3\}$ と $\{R5\}$ にグループ分けされるため，分配バランスの良い $R3$ の Left 辺を分割時に用いる．

内部ノードにおける CL の取得を容易にするために，kdm-tree は内部ノードスロット内の子ノードに順序をもたせる．kdm-tree の内部ノードスロット内で隣り合う管理長方形は，Top > Bottom, Left > Right の優先順位を常にもつ．それによりスロット内で隣り合う管理長方形は，一方の Bottom 辺と他方の Top 辺若しくは一方の Left 辺と他方の Right 辺が一つの線分を共有する．図 3 に内部ノードスロット内の子ノード順序と，その管理長方形 $R1$〜$R8$ の関係を示す．内部ノードスロットを 0 から順にたどり，スロット [i] と $[i+1]$ の共有する線分を調べることで CL を得る．図 3 の例ではスロット [3] と [4]，つまり $R6$ と $R2$ が共有する線分が CL となり，スロット [0]〜[3] とスロット [4]〜[7] で二つのグループに簡単に分けることができる．

3.3 データの挿入と領域分割

データ挿入時に用いられるアルゴリズムは以下の五つから構成される．木をたどり挿入対象葉ノードを決定する Insert，葉ノードを分割する Split Leaf，内部ノードを分割する Split Node，Split Node において CL を取得する Get Critical Line，内部ノードの不均



(a) Leaf Split occurred (b) After Leaf Split

図 4 Split Leaf の例
Fig. 4 Example of Split Leaf.

等な分割を是正する Adjust Node である．それぞれのアルゴリズムを示す．

［Insert アルゴリズム］ 管理対象データ d を kdm-tree に挿入する．

I1 根ノードから木をたどり，d を含める管理長方形をもつ葉ノード L を見つける．
I2 もし $L.n_l < P$ ならば，L に d を挿入，終了．
I3 もし $L.n_l = P$ ならば，L に対して Split Leaf を呼ぶ．

［Split Leaf アルゴリズム］ 葉ノード L_i に $P+1$ 個目のデータが挿入されたとき，L_i の分割を行う．ここで L_i の親ノードを N_p，$N_p.child[i] = L_i$ とする．

SL1 $L_i.R$ の長軸方向（$L_i.R$ の縦・横辺の長辺方向）に保有データ $P+1$ 個をソートする．
SL2 $L_i.R$ をそれぞれ $P/2$ 若しくは $P/2+1$ のデータをもつように $L_i.R$ の長軸と垂直方向に分割線を引くことで二つの管理長方形に分割する．
SL3 二つの長方形の空間的位置をもとに，L_i に Top または $Left$ に位置する管理長方形と対応するデータを割り当て，新たな葉ノード L_{new} に $Bottom$ または $Right$ に位置する管理長方形と対応するデータを割り当てる．
SL4 もし $N_p.n_n < M$ ならば L_{new} を $N_p.child[i+1]$（L_i の一つ後ろのスロット）に挿入，$N_p.child[i+1]$ 以降は一つずつ後ろにシフトする．終了．
SL5 もし $N_p.n_n = M$ ならば Split Node を呼ぶ．

図 4 は Split Leaf の例である．R1 を管理する葉ノードに Split Leaf が適用されたとき，R1 を R1 と R4 に分ける．葉ノード内データを長軸（ここでは縦軸）方向にソートし，R1 と R4 内のデータ数がそれぞれ $P/2$ 若しくは $P/2+1$ となるように分割線を決定する．R4 と対応データを新たな葉ノードに割り当て，親ノードのスロット内において R1 の隣に挿入する．

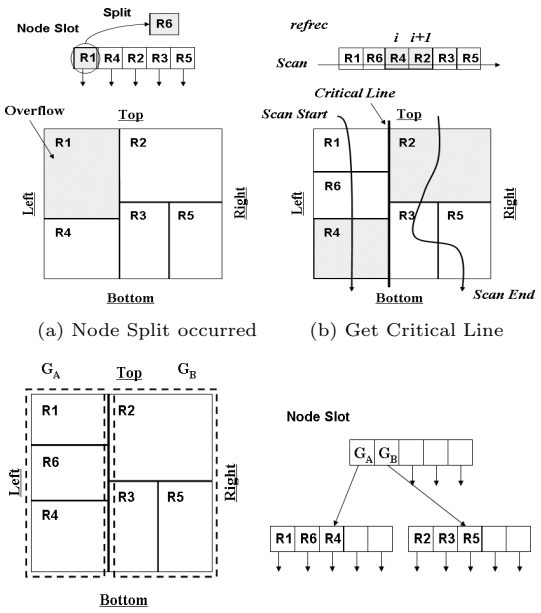
［Split Node アルゴリズム］ 内部ノード N_i に $M+1$ 個目の子ノードが挿入されたとき，N_i の分割を行う．N_i の親ノードを N_p，$N_p.child[i] = N_i$ とする．

SN1 $M+1$ 個の子ノード管理長方形を $refrec[M+1]$ とし Get Critical Line を呼ぶ．新たな子ノードの $refrec$ におけるスロット位置は子ノード順序に従う．
SN2 CL によって $N_i.R$ を二つの管理長方形に分割，子ノード群を二つのグループに分ける．
SN3 もしどちらかのグループの子ノード数が $minM$ 以下ならば，$M+1$ 個の子ノード群に対して Adjust Node を適用，終了．
SN4 二つの管理長方形の空間的位置をもとに，N_i に Top または $Left$ に位置する管理長方形と対応する子ノードを割り当て，新たなノード N_{new} に Bottom 若しくは Right に位置する管理長方形と対応する子ノードを割り当てる．
SN5 もし $N_p.n_n < M$ ならば N_{new} を $N_p.child[i+1]$（N_i の一つ後ろのスロット）に挿入，$N_p.child[i+1]$ 以降は一つずつ後ろにシフトする．終了．
SN6 もし $N_p.n_n = M$ ならば Split Node を呼ぶ．

［Get Critical Line アルゴリズム］ $M+1$ 個の子ノード管理長方形 $refrec[M+1]$ から親ノード N_p の管理長方形 $N_p.R$ を二分割する線分 CL を取得する．

GC1 $refrec$ を走査，もし $refrec[i]$ と $refrec[i+1]$ の共有する線分が R の 1 辺と長さが等しい場合，CL_i として保持する．
GC2 CL_i のうち，i が最も $M/2$ に近い CL_i を CL として返す．

図 5 に $M=5$ における Split Node とそのときの Get Critical Line の例を示す．R1 が R1 と R6 に分割されたとき，親ノードスロットに空きがなければ親ノードに Split Node を適用する．まず R6 を含めた $M+1$ 個の子ノードに対して Get Critical Line を適用する．CL は内部ノードスロット内で隣り合う二つの管理長方形が共有する線分で，かつ線分の長さが親ノードの管理長方形の 1 辺と等しい線分である．図 5 (b) 中の一時的なスロット $refrec$ 内において条件を満たす線分は管理長方形 R4 と R2 が共有する線分で，R1・R6・R4 の Right 辺からなる線分が CL となる．取得した CL を用いて長方形を二つのグルー



(a) Node Split occurred (b) Get Critical Line

(c) After Node Split

図 5 Split Node の例 1 (M=5)
Fig. 5 Exmaple of Split Node. (M=5)

プ G_A と G_B に分け，図 5 (c) のようにノード分割を行う．

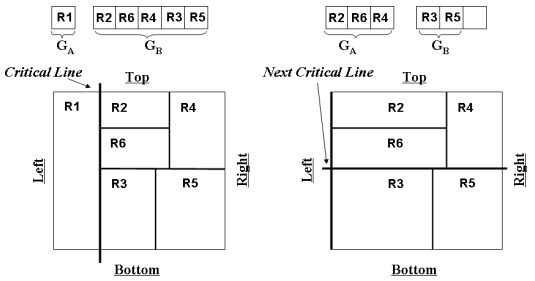
［Adjust Node アルゴリズム］ $M+1$ 個の子ノード群が CL により二つのグループ G_A と G_B に分けられ，どちらか一方（ここでは G_A）の子ノード数が $minM$ 以下の場合，子ノードの管理長方形を調整する．

AN1 G_A 内子ノードをたどりデータを回収，G_A 以降の部分木を削除する．

AN2 G_B 内子ノードの管理長方形で CL と接する辺を G_A の管理長方形を包括するよう拡大する．

AN3 N から回収したデータを再挿入する（この挿入によって再び分割が生じる可能性がある）．

図 6 は Adjust Node の例である ($M=5$)．Adjust Node は取得した CL によって分けられたグループのうち，どちらか一方の子ノード数が $minM$ 以下だった場合に呼び出される（ここでは $minM$=1）．図 6 (a) において G_A は子ノードを一つしかもたないため，図 6 (b) のように R1 と隣接する G_B 内の長方形 R2, R6, R3 を拡大することで領域を補う．次に G_A 以下に存在するデータを G_B に対して再挿入する．このとき，場合によっては再帰的にノード分割が生じるが，図 6 は再挿入後に R2, R6, R3 に対応するノードが分割されないときの例である．



(a) Before Adjust Node (Inefficient Critical Line) (b) After Adjust Node

図 6 Adjust Node の例
Fig. 6 Example of Adjust Node.

3.4 データの削除と領域統合

データ削除時に用いられるアルゴリズムは以下の三つである．木をたどり削除対象データを削除する Delete，葉ノードの統合を行う Leaf Integration，内部ノードの統合を行う Node Integration．それぞれのアルゴリズムを示す．

［Delete アルゴリズム］ 削除対象データ d を木構造から削除する．

D1 根ノードから木をたどり，d を含める管理長方形をもつ葉ノード L を見つけ，L から d を削除．

D2 もし $L.n_l > minP$ ならば，終了．

D3 もし $L.n_l \leq minP$ ならば，L に対して Leaf Integration を呼ぶ．

［Leaf Integration アルゴリズム］ 葉ノード L_i の保有データ数が $minP$ 以下のとき，隣接する管理長方形を拡大することで L_i の削除・統合を行う．L_i の親ノードを N_p，$N_p.child[i] = L_i$ とする．

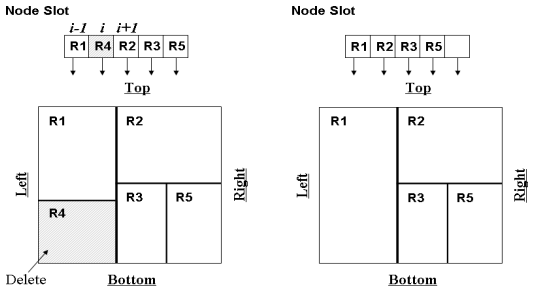
LI1 もし $N_p.child[i-1]$ または $N_p.child[i+1]$ の管理長方形が L_i と 1 辺を完全共有するとき，L_i の管理長方形を含むように拡大して L_i 内データを対象葉ノードに再挿入する．L_i を削除，LI3 へ．

LI2 $N_p.child$ の管理長方形群を走査，L_i の管理長方形と隣接し，かつ隣接辺の長さの和が L_i の 1 辺と等しくなる複数管理長方形群を取得し，L_i の管理長方形を含むように拡大，L_i のデータを N_p に再挿入する．L_i を削除．

LI3 もし $N_p.n_n > minM$ ならば，終了．

LI4 もし $N_p.n_n \leq minM$ ならば，N_p に対して Node Integration を呼ぶ．

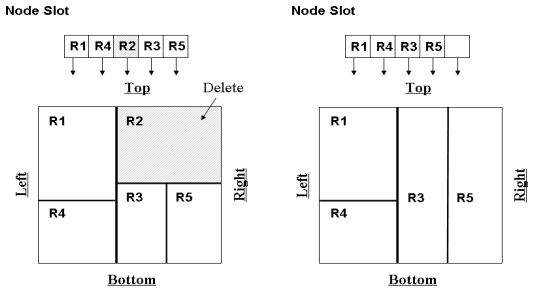
図 7 は削除管理長方形 R4 に 1 辺を完全に共有する



(a) Before integration (b) After integration

図 7 削除時における領域統合の例 1

Fig. 7 Example 1. region integration in delete.



(a) Before integration (b) After integration

図 8 削除時における領域統合の例 2

Fig. 8 Example 2. region integration in delete.

R1 が存在した場合の例である．R1 の管理長方形を拡大し，R4 の領域を補うことで統合を行う．この処理はステップ LI1 における管理長方形の統合に対応する．

図 8 は削除管理長方形 R2 の 1 辺を複数長方形の辺が構成している場合の例である．R3 と R5 の管理長方形を拡大することで R2 の領域を補う．この処理はステップ LI2 における管理長方形の統合に対応する．

［Node Integration アルゴリズム］ 内部ノード N_i のもつ子ノード数が $minM$ 以下のとき，N_i の削除・統合を行う．ここで N_i の親ノードを N_p，$N_p.child[i] = N_i$ とする．

NI1 もし $N_p.child[i-1]$ または $N_p.child[i+1]$ の管理長方形が N_i と 1 辺を完全共有し，かつ N_i との子ノード数の和が M 以下のとき，一方が他方の管理長方形を含むように拡大し，子ノードポインタを移して内部ノードを統合する．不要となった内部ノードを削除，NI4 へ．

NI2 もし $N_p.child[i-1]$ または $N_p.child[i+1]$ の管理長方形が N_i と 1 辺を完全共有し，かつ N_i との子ノード数の和が M より大きいとき，N_i との共有辺を N_i の管理長方形を含むように拡大，N_i を N_p から削除し，N_i 以降のデータを N_p から再挿入する．NI4 へ．

NI3 $N_p.child$ の管理長方形群を走査，N_i の管理長方形と隣接し，かつ隣接辺の長さの和が N_i の 1 辺と等しくなる複数管理長方形群を取得し，N_i の管理長方形を含むように拡大，N_i 以降のデータを N_p に再挿入する．N_i 以降の部分木を削除．NI4 へ．

NI4 もし $N_p.n_n > minM$ ならば，終了．

NI5 もし $N_p.n_n \leq minM$ ならば，N_p に対して Node Integration を呼ぶ．

Node Integration における管理長方形の統合は Leaf Integration と同様である（図 7，図 8 参照）．

3.5 移動オブジェクトデータの更新

オブジェクトが移動するごとに Update アルゴリズムを実行する．

［Update アルゴリズム］ オブジェクト O_i が d_{iold} から新位置 d_{inew} へ移動したとき，木構造からを d_{iold} を削除して d_{inew} の挿入を行う．

UD1 直接リンクをたどり，d_{iold} を管理する葉ノード L へ移動する．

UD2 もし d_{inew} が L の管理長方形内であれば，d_{iold} に d_{inew} を上書き，終了．

UD3 もし d_{inew} が L の管理長方形外であれば，L から d_{iold} を削除し，$L.n_l \leq minP$ ならば，Leaf Integration を適用する．再帰的に親ノードをたどって d_{inew} を挿入すべき管理長方形を探索し，対象内部ノードから Insert を適用する．

4. 計算機実験

kdm-tree の更新・メモリコスト・検索の性能評価のためにデータ包括形の R-tree Bottom Up Approach (RBU-tree) [2]，領域分割形で最悪メモリコストを保証しない Buddy*-tree [5]，領域分割形で最悪メモリコストを保証する MD-tree [14] との比較実験を行う．表 2 にシミュレーション条件を示す．これらのアルゴリズムを C++で記述し，計算機として Pentium®4 2.8 GHz，512 MByte RAM を用いて Windows XP Home Edition 上で実行する．またそれぞれの木構造は主記憶上に作成する．

移動オブジェクト数 N は 10 万〜100 万とし，10 万 × 10 万 の二次元空間に分布させる．更新回数は 1000 回とし，更新中にデータ数に増減はないも

表 2　シミュレーション条件
Table 2　Simulation condition.

CPU	Pentium®4 2.80 GHz
Main memory	512 MB RAM
The number of objects N	100 k, 200 k, 500 k, 1 M
Entire area (2 dimension)	$0 \leq X \leq 100$ k, $0 \leq Y \leq 100$ k
The number of update	1000
Distribution	Random, Gaussian, Skewed
Ratio of search area(%)	0.01, 0.02, 0.05, 0.1, 0.2, 0.5, 1
Max. moving distance V_m	50,100,200,500,1000
Max. node number M	50
Min. node number $minM$	$M/3$
Bucket capacity P	M
Min. Bucket capacity $minP$	$minM$
RBU-tree's node splitting	Quadratic-Cost algorithm
RBU-tree's Parameter ε	$V_m/10$

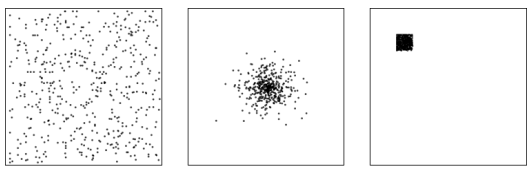
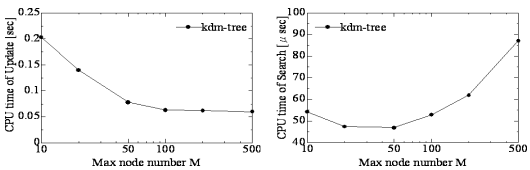


図 9　データ分布モデル
Fig. 9　Model of data distribution.



(a) CPU time of Update　(b) CPU time of Search

図 10　M による性能変化 ($N = 10$ 万，検索範囲 1%)
Fig. 10　Effect of M. ($N = 100$ k, Search Range 1%)

のとする．更新 1 回につきすべての移動オブジェクトの位置情報が更新される．図 9 に初期状態のデータ分布モデルを示す．データ分布は (a) 一様分布 (Random)，(b) 正規分布 (Gaussian)，(c) 全空間面積の 1%の領域面積に全データが集中する偏在分布 (Skewed) の 3 通りで，その後の更新によって移動オブジェクトはランダム方向に最大移動距離 $V_m(50 \leq V_m \leq 1000)$ で加速度を考慮して移動する．検索は正方形範囲検索とし，全空間面積の 0.01%〜1% の領域面積に行う．

kdm-tree のバケット容量 P に関し，他の木構造と条件を等しくするために P=M，$minP$=$minM$ とした．図 10 に M による kdm-tree の性能変化を示す．図 10 (a) に示すように M を大きくとると更新性能が向上するが，図 10 (b) に示すように検索性能は劣化するトレードオフの関係がある．これは他の木構造においても同様である．ここでは双方のバランスの良い M=50 とした．また Adjust Node が繰り返されるのを避けるため $minM$=$M/3$ とした．

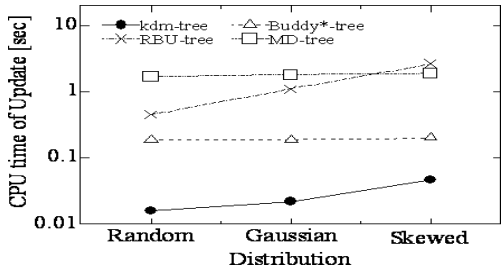


図 11　データ分布と更新時間 ($N = 10$ 万，$V_m = 50$)
Fig. 11　Update time with distribution. ($N = 100$ k, $V_m = 50$)

RBU-tree のノード分割には Quadratic-Cost アルゴリズムを用いた．RBU-tree における最大子ノード数 M と最小子ノード数 m はそれぞれ kdm-tree における M と $minM$ に等しい．RBU-tree の MBR 拡大に用いるパラメータ ε は ε=$V_m/10$ とした．

4.1 更新性能

図 11 に各データ分布に対する更新時間を示す．ここで更新時間とは，1000 回の更新の 1 回当りの平均時間である．kdm-tree は RBU-tree に比べ，一様分布では約 1/25 の更新時間ですみ，他の分布では更に少ない時間で更新が可能となる．データ包括形である RBU-tree はオブジェクトが移動するたびに，MBR を調節する必要が生じるため演算量が大きい．特に正規分布・偏在分布では，MBR 変化の頻度が多く，更新時間が増加する．kdm-tree でも正規分布・偏在分布で領域の統合が頻繁に起こるが，管理長方形の統合や分割の処理量が比較的少ないため更新時間の増加は小さい．

図 11 において kdm-tree は Buddy*-tree に比べ，偏在分布では約 1/4 の更新時間で済み，他の分布では更に少ない時間で更新が可能となる．これは直接リンクを用いており，更新時にたどるノード数が削減されるためである．Buddy*-tree は常に根ノードから挿入と削除を行うため，データ分布の影響は少ない．

図 11 において kdm-tree は MD-tree に比べ，偏在分布では約 1/35 の更新時間で済み，他の分布では更に少ない時間で更新が可能となる．MD-tree は高いメモリ効率が特徴だが，一方で最大子ノード数が 3 であるため，データ位置変化に応じて上位ノードの分割や統合が頻繁に生じ，処理コストが大きい．

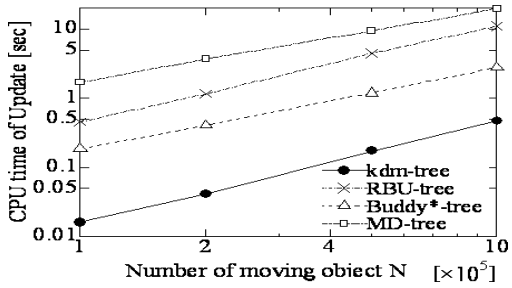


図 12　データ数と更新時間（$V_m = 50$，一様分布）
Fig. 12　Update time with data number.（$V_m = 50$, Random)

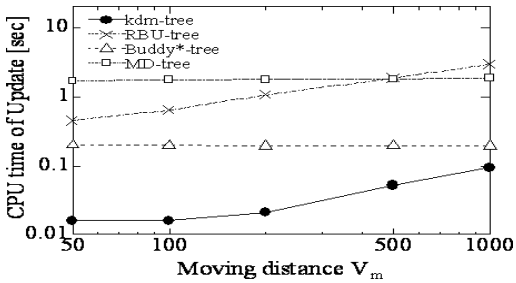


図 13　移動距離と更新時間（$N = 10$ 万，一様分布）
Fig. 13　Update time with distance.（$N = 100$ k, Random)

図 12 にデータ数 N に対する更新時間を示す．データ数 N によらず kdm-tree が最も高速で，$N = 100$ 万のときで RBU-tree に対して約 1/23，Buddy*-tree に対して約 1/6，MD-tree に対して約 1/42 の更新時間で済む．いずれの木構造もデータ数に比例した更新時間となることが分かる．

図 13 に最大移動距離 V_m に対する更新時間を示す．移動距離が 1000 のとき，オブジェクトは全体領域の 1 辺に対して 1%移動する．$V_m = 1000$ のとき，kdm-tree の更新時間は RBU-tree に対して約 1/30，Buddy*-tree に対して約 1/2，MD-tree に対して約 1/20 で済む．V_m が大きくなると，新たなデータ位置が直接リンク先の葉ノード管理長方形の外となる確率が大きくなるため，kdm-tree と RBU-tree の更新時間が増加する．Buddy*-tree と MD-tree はノードの分割・統合の処理が小さく，かつ根ノードから削除と挿入を行うため V_m の影響は小さい．MD-tree において V_m が大きくなるとノードの分割・統合の頻度が増加し，$V_m = 50$ と $V_m = 1000$ のときを比較すると約 10%更新時間が増加する．

以上のように kdm-tree の更新時間は最悪メモリコストを保証する RBU-tree や MD-tree に比べて格段に短縮され，Buddy*-tree と比べて約 1/2 以下で済む．

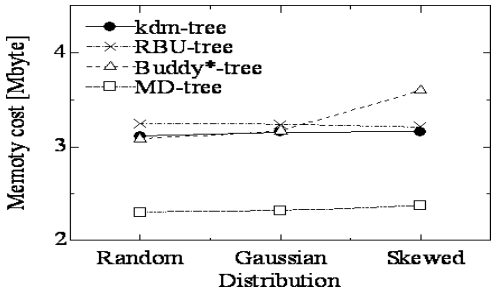


図 14　メモリコスト（$N = 10$ 万，$V_m = 50$, 1000 回更新後）
Fig. 14　Memory cost.（$N = 100$ k, $V_m = 50$, After 1000 update)

4.2　メモリコスト

図 14 に各データ分布において 1000 回更新した後のメモリコストを示す．kdm-tree と RBU-tree は直接リンク（RBU-tree はノード数に比例する更新補助情報を含む）のメモリコスト 0.4 MByte を含んでいる．kdm-tree は RBU-tree に対して約 2〜4%メモリコストが少なく済む．これは葉ノード分割時，RBU-tree はデータ数が m 以上となるように分割するのに対し，kdm-tree はデータ数を $P/2$ 若しくは $P/2+1$ に分割するために生じた差と考えられる．

Buddy*-tree は最悪メモリコストを保証できないため，更新によってメモリコストが増加する問題がある．Buddy*-tree のメモリコストは，一様分布では直接リンクをもつ kdm-tree と同程度のメモリコストとなり，偏在分布では kdm-tree に比べて約 12%メモリコストが増加する．

MD-tree の葉ノードはバケット容量の 2/3 以上データをもつことを保証しているためメモリ効率が高い．kdm-tree は MD-tree に比べて更新時間は 20 倍以上となるが，メモリコストは約 1.4 倍に増加する．

4.3　検索性能

図 15 に 1000 回更新後の検索時間を示す．ただし，検索時間は 100 回の範囲検索をランダム位置に発生させたときの平均時間である．kdm-tree と Buddy*-tree の検索性能はほぼ同等である．kdm-tree は RBU-tree に対して，検索範囲（面積）が全体領域の 1%のとき約 1/3 の検索時間で済み，検索範囲が全体領域の 0.01%のとき約 1/35 の検索時間で済む．RBU-tree は更新時に葉ノード MBR を拡大して管理するため，繰

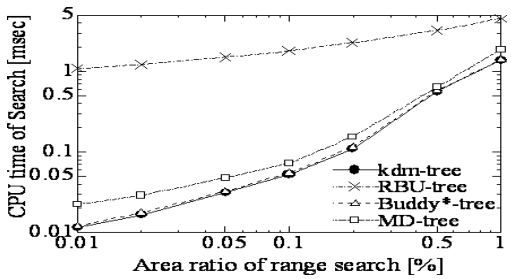


図 15 検索時間（N=10 万，一様分布，1000 回更新後）
Fig. 15 CPU time of Search. (N=100 k, Random, After 1000 update)

り返す更新で領域の重複が増加して検索性能が劣化する．R*-tree [16] の分割アルゴリズムを用いることで，領域の重複は少なくなるが更新速度は低下する．MD-tree は他の領域分割形の手法に比べて検索時間が増加する．これは MD-tree が上位ノードにおいて領域の重複を許しているためである．

5. むすび

本論文では移動オブジェクトの更新管理に適した索引構造に必要な要件として，領域分割形であること，削除処理の高速化，最悪メモリコストの保証を挙げ，これらを満たす木構造 kdm-tree を提案した．

kdm-tree 独特の CL を用いるノード分割は，分割時の処理コストは多少増加するが，削除処理を大幅に高速化するとともに最悪メモリコストの保証を可能にした．

この領域分割形の新アルゴリズムの採用により，計算機実験では kdm-tree の更新速度は RBU-tree に比べ約 25 倍，Buddy*-tree に比べ約 6 倍高速になった．またメモリコストや検索性能もこれらと同等以上であった．

文献

[1] D. Kwon, S. Lee, and S. Lee, “Indexing the current positions of moving objects using lazy update r-tree,” Third International Conference on Mobile Data Management, p.113, Singapore, Jan. 2002

[2] M.L. Lee, W. Hsu, C.S. Jensen, B. Cui, and K.L. Teo, “Supporting frequent updates in R-trees: A bottom-up approach,” Proc. VLDB, 2003.

[3] X. Xiong and W.G. Aref, “R-trees with update Memos,” Proc. International Conference on Data Engineering, vol.22, 2006.

[4] C.S. Jensen, D. Lin, and B.C. Ooi, “Query and update efficient B^+-tree based indexing of moving objects,” Proc. VLDB, 2004.

[5] S. Guo, Z. Huang, H.V. Jagadish, B.C. Ooi, and Z. Zhang, “Relaxed space bounding for moving objects: A case for the buddy tree,” SIGMOD Record, vol.35, no.4, pp.24–29, 2006.

[6] S. Saltenis, C.S. Jensen, S.T. Leutenegger, and M.A. Lopez, “Indexing the positions of continuously moving objects,” ACM Special Interest Group on Management of Data, pp.331–342, 2000.

[7] Y. tao, D. Papadias, and J. Sun, “The TPR*-tree: An optimized spatio-temporal access method for predictive queries,” Proc. VLDB, 2003.

[8] V.T. Almeida and R.H. Gutting, “Indexing the trajectories of moving objects in networks,” Technical Report 309, Fernuniversitat Hagen, Fachbereich Informatik, 2004.

[9] K. Raptopoulou, M. Vassilakopoulos, and Y. Manolopoulos, “Towards quadtree-based moving objects databases,” Proc. East-European Conference on Advances in Database Systems and Information Systems (ADBIS) 2004.

[10] J. Ni and C.V. Ravishankar, “Indexing spatio-temporal trajectories with efficient polynomial approximations,” IEEE Trans. Knowl. Data Eng., vol.19, no.5, pp.663–678, 2007.

[11] A. Guttman, “R-tree: A dynamic index structure for spatial searching,” Proc. ACM Special Interest Group on Management of Data, pp.49–59, 1984.

[12] J.L. Bentley, “Multidimensional binary search trees used for associative searching,” Commun. ACM, vol.18, no.9, pp.509–517, 1975.

[13] R.A. Finkel and J.L. Bentley, “Quad trees: A data structure for retrieval on composite keys,” Acta Inform., vol.4, no.1, pp.1–9, 1974.

[14] Y. Nakamura, S. Abe, Y. Ohsawa and M. Sakauchi, “A balanced hierarchical data structure for multidimensional data with highly efficient dynamic characteristics,” IEEE Trans. Knowl. Data Eng., vol.5, no.4, pp.682–694, 1993.

[15] J.T. Robinson, “The K-D-B tree: a search structure for large multidimensional dynamic indexes,” Proc. ACM SIGMOD International Conference on Management of Data, 10-18, 1981

[16] N. Beckman, H-P Kringel, R. Schneider, and B. Seeger, “The R*-tree: An efficient and robust access method for points and rectangles,” ACM Special Interest Group On Management of Data, 1990.

（平成 19 年 12 月 17 日受付，20 年 6 月 9 日再受付）

西川　嘉人　（学生員）

2006 埼玉大・工・電気電子システム卒．現在，同大大学院修士課程在学中．多次元データ管理構造の研究に従事．日本データベース学会会員．

辻　　俊明

2006 慶應義塾大学大学院理工学研究科総合デザイン工学専攻後期博士課程了．同年 4 月東京理科大学工学部第一部機械工学科助手．現在，埼玉大学工学部電気電子システム工学科助教．博士（工学）．主としてロボティクス，モーションコントロールに関する研究に従事．

金子　裕良

1987 埼玉大・工・電気卒．1989 同大大学院修士課程了．同年新日本製鐵（株）入社．1990 年 4 月埼玉大学工学部助手．1995 年 2 月同大学総合情報処理センター講師．2000 年 4 月工学部講師．電気機器の制御及び産業ロボットの知的情報処理・制御の研究に従事．工博．電気学会，計測自動制御学会各会員．

阿部　　茂　（正員）

1971 東大・工・電子卒．1976 同大大学院博士課程了．工博．同年三菱電機（株）入社．中央研究所，産業システム研究所で電力系統，オブジェクト指向，多次元データ管理構造とその応用システムの研究開発に従事．1997 同社稲沢製作所エレベーター開発部長．2001 ビルシステム事業本部技師長．2004 年 4 月埼玉大学工学部教授，1985 電気学会論文賞受賞．電気学会，IEEE，情報処理学会各会員．