時間情報を含んだ図形データの木構造による管理

3C－1

寺岡照彦　丸山稔　中村泰明　西田正吾

三菱電機(株)　中央研究所

1. はじめに

点・線・領域といった図形データの木構造による管理手法が提案されているが、いずれも時間情報を含んでいるデータを対象としてはいなかった [1]。本稿では「時間情報を含んだ図形データ」としてパターン情報と発生・消滅時間等の時間情報をもつデータを扱い、従来の木構造による管理手法を拡張して時間情報を含んだ検索にも対応できる手法を提案する。なお以下では簡単のため、2次元の点情報について説明する。

2. 従来の木構造による管理

2.1 多次元データの管理

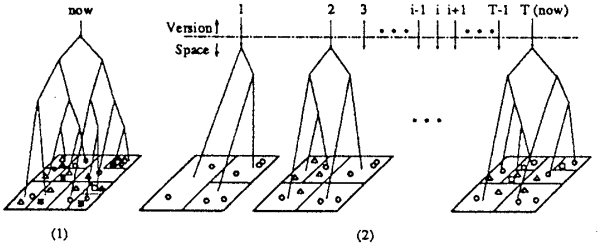

図1:　従来の管理木による時間情報を含んだ管理

多次元データの管理手法として、k-d木やBD木等の領域管理木がよく知られている [1]。これらによって時間情報(バージョン)を含んだ図形データの管理を考えると、通常は、メモリーコスト、更新コストや検索コストの間にトレードオフが生じる。例えば次のような方法が考えられる(図1)。

(1) メモリー重視型…現在までに登録された全データについて領域管理木を生成する。

(2) 単一バージョン検索重視型…更新毎に新しい領域管理木を生成し、それらをすべて保持する。

(1)では、データの重複が起こらないのでメモリーコストが低いが、データの有効・無効に関わらず全バージョンを一つの木で管理しているために、単一バージョンのデータ検索効率は非常に悪い。(2)では、単一バージョンに対する各種検索は非常に効率がよいが、全バージョンに対する検索は非常に効率が悪く、またデータが重複して記憶されメモリーコストが非常に高いという欠点がある。更新時のコストも非常に高い。したがって、(1)、(2)のような構造ではなく、(a) メモリーコストおよび更新時のコストが(1)と同等、(b) 単一バージョン(特に current バージョン)に対する各種検索の効率が(2)と同等、といった性質をもち、かつ(c)「ある期間にある領域に存在したデータは?」のような時間情報を含んだ検索が可能なデータ構造が望まれる。

A New Data Structure for Spatio-Temporal Data
Teruhiko Teraoka, Minoru Maruyama, Yasuaki Nakamura and Shogo Nishida
Mitsubishi Electric Corp., Central Research Lab.
8-1-1 Tsukaguchi-honmachi,Amagasaki,Hyogo,Japan

2.2 時間情報を含んだ1次元データの管理

時間情報を含んだ1次元データを管理し、過去のバージョンにアクセスできる構造として、Persistent search tree が提案されている [2]。更新の前後で木の構造は一部を除いて同じため、(2)のようにすべての構造を保持するのではなく、変更部分だけを新たに保持すれば十分である。Persistent search tree の構成法の一つであるノードコピー法 [2] では、更新の前後で変化のあったノードだけを新たに保持し、それ以外は過去の木と共有する。但し、各ノードは左および右部分木へのポインタ以外に余分なポインタを持ち、ポインタが変更されるときはこの余分なポインタを使用し、余分なポインタも使用済みのときのみ新たにノードを生成する。これにより、新しいノードの生成が必要最小限に押えられる。

3. 時間情報を含んだ図形データの管理

本稿では、前記のトレードオフを解消し、時間情報を含んだ図形データを管理するために、k-d木やBD木等の領域管理木に Persistent search tree のノードコピー法を適用し、さらに次のような拡張を行なった。

i) 従来の Persistent search tree が対応していなかったバケット型(ページ)管理を可能にする。

ii) 最も頻度の高い current バージョンにおける検索を重視して、一つの葉に有効なデータと無効なデータが同時に記憶されるのをさけるため、葉はその時点で有効なデータのみを記憶し、削除されたデータは別に用意したブロックに記憶する。

iii) バージョンに関する検索に対応するために、各ノードにバージョンの情報を持たせる。

各ノードは次のような属性を持つ。

内部ノード：　$\langle v, R, l_c, r_c, (ins, del), v_p, p_c, lr \rangle$

v はノード生成時のバージョン、R は管理領域、l_c, r_c は左右の子ノードへのポインタ、ins は下位ノードが格納するデータの中で最近挿入されたデータのバージョン、del は最近削除されたデータのバージョン、v_p はポインタ変化時直前のバージョン、p_c は変化前の子ノードへのポインタ(余分なポインタ)、lr はそれが右か左かの識別子を示す。

葉：　$\langle v, D[B], N, p_d, v_d, (ins, del) \rangle$

v は葉生成時のバージョン、$D[B]$ はデータを格納する配列(B はデータ容量)、N は格納しているデータ数、p_d は削除データの格納ブロックのアドレス(ポインタ)、

v_d は削除データ用ブロック生成時のバージョン、ins は最近挿入されたデータのバージョン、del は最近削除されたデータのバージョンを示す。p_d で示されるブロックは削除されたデータを配列 $d[B]$ に格納する。各データは、⟨ id 番号, 実体, 挿入時のバージョン, 削除時のバージョン, 修正・コピー後 (前) のデータが保存されているブロックのアドレス (ポインタ) ⟩ を要素としてもつ。ノードの l_c, r_c ポインタ、および葉の中のデータ $D[B]$ は、更新後、すなわちその時点での current バージョンでは必ず有効であるために、current における空間検索ではバージョンの属性チェックは一切必要ない。これは current においては (2) と同等の検索効率を持つことを意味する。提案手法における更新の基本操作を図2に示す。ここで、* は NULL、括弧内は (ins, del) を表す。

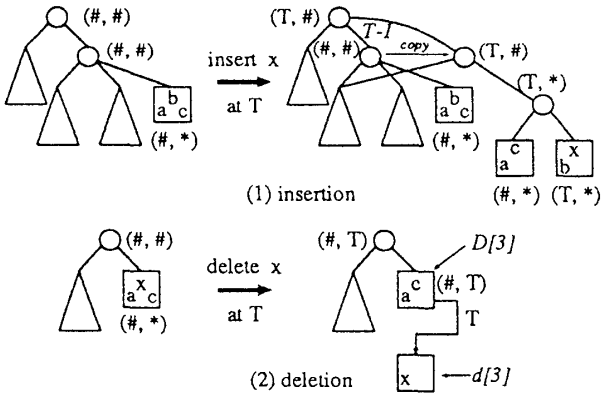

図 2: データ更新時の基本操作 ($B = 3$)

また、「バージョン $[i, j]$ に領域 r に存在するデータの検索」は以下のようにすればよい。

● $[i, j]$ と重なる期間を管理するすべての根 $root_k$ について、順に以下の操作を行なう。

1. $I = root_k$ とする。
2. I が葉でないとき、
   $v > j$ あるいは I が調査済みならば return。
   R と r が交差しなければ return。
   それ以外のとき、
   $v_p \geq j$ ならば p_c と、lr とは反対側の子ノードを I として 2 以下を再帰的に適用。
   $i \leq v_p < j$ ならば、p_c, l_c, r_c で示される子ノードを I として 2 以下を再帰的に適用。
   $v_p < i$ ならば、l_c, r_c で示される子ノードを I として 2 以下を再帰的に適用。
3. I が葉のとき、
   $v > j$ あるいは I が調査済みならば return。
   それ以外のとき、
   I に格納されているデータについて、$[i, j]$ 内に存在し、かつ r と交差するものを求める。
   さらに、p_d が指すブロック内のデータについても調べる。return。

図3に提案法と方法 (1), (2) により、同一のデータ更新を行なったときの例を示す (BD 木を使用)。提案手法は、上記のような時空間情報による検索が可能であり、ノード数 (メモリーコストに相当) は (2) ほど多くなく (上の例では (1)15 個、(2)44 個に対して、23 個)、単一バージョンのデータ検索効率は、全てのバージョンにおける構造が残っているために (1) ほど悪くない。更新時は、高々、データ挿入・削除の際のパス上のノードがコピーされるだけで (2) ほど悪くない。

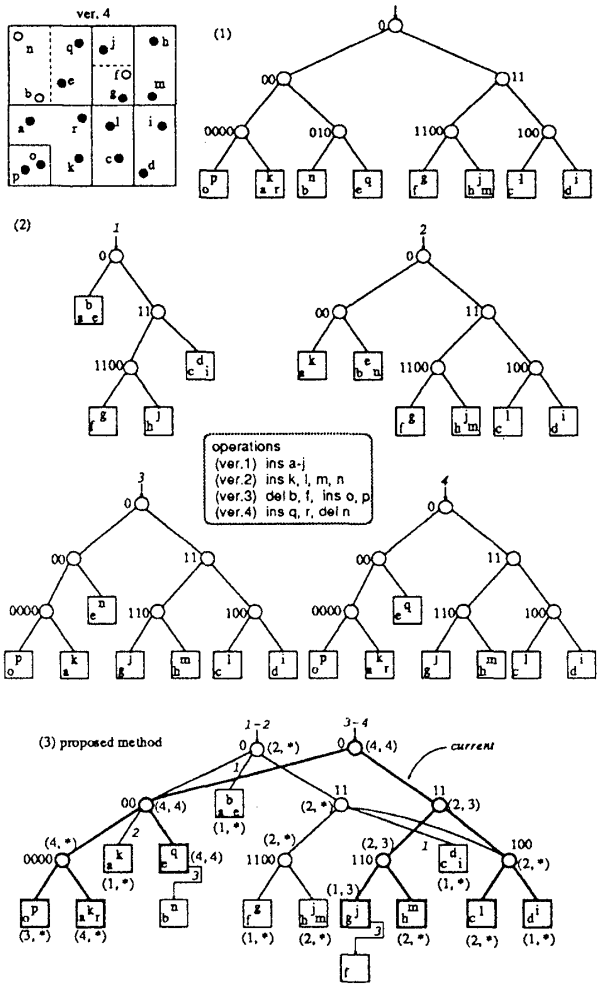

図 3: 提案法と方法 (1), (2) の比較 ($B = 3$)

4. おわりに

本稿では、時間情報を含んだ図形データに対して、「ある期間にある領域に存在したデータは?」といったような時間情報を含んだ検索にも対応できる管理手法を提案した。今後は本手法を地図・図面管理システムなどに応用する予定である。

参考文献

[1] 坂内正夫, 大沢裕; 画像データベース, 昭晃堂 (1987)

[2] Sarnak, N. and Tarjan, R. E.; "Planar point location using persistent search trees", *CACM*, vol. 29, pp.669-679 (1986)