

データ圧縮型多次元空間インデックス VA-TREE

吉田 忠城[†] 赤間 浩樹[†] 谷口 展郎[†]
山室 雅司[†] 串間 和彦[†]

大容量のマルチメディアデータベースに必須な高速多次元空間インデックスを実現するためには、ディスク I/O とベクトル間の距離計算回数の削減が重要である。特にディスク I/O はコストが非常に高いためメモリ空間の有効利用が高速化の鍵となる。本論文では、木構造インデックスにおける中間ノードおよびリーフノードの矩形情報を圧縮してメモリ空間に保持する動的多次元空間インデックス VA-TREE について提案する。画像検索システムで用いられている色相、輝度、および形状特徴量の実データを用いて評価実験を行った結果、データを圧縮してフラットな構造で管理する VA-File に対する優位性を確認した。また、高速な木構造インデックスとされる VAM Split R-tree と比較してデータ分布の偏りが強いデータセットに対しては劣るものの、ディスク I/O がデータ分布に影響を受けない優位性を確認した。

Similarity Search Index Using Vector Approximation VA-TREE

TADASHIRO YOSHIDA,[†] HIROKI AKAMA,[†] NOBUROU TANIGUCHI,[†]
MASASHI YAMAMURO[†] and KAZUHIKO KUSHIMA[†]

This paper describes the similarity search index VA-TREE, which uses the vector approximation method to represent internal and leaf nodes in a tree structure index. Our experiment on 16-dimensional hue, 16-dimensional intensity and 24-dimensional shape data used in an actual image retrieval application shows the advantage over the flat structure index VA-File, which uses vector approximation as well. The experiment also shows that the number of accessing feature vectors on a disk in VA-TREE is not affected by data distribution as much as VA-File or VAM Split R-tree.

1. はじめに

デジタルカメラやイメージスキャナ、音声高圧縮方式 MP3 を用いた録音/再生装置の普及により、近年、画像や音楽などのいわゆるマルチメディアデータのデジタル化が急速に進んでいる。また、インターネットやイントラネットにおけるネットワーク環境の向上はこれらのマルチメディアデータの共有を可能にし、WWW (World Wide Web) などを含む広い意味でのマルチメディアデータベースが大規模化している。蓄積されたデータを有効利用するために、従来のテキストデータに加え、これらのマルチメディアデータに対して検索を行いたいという要求は必然的であるといえる。

マルチメディアデータの検索には大きく 2 つの方式がある。1 つはデータにその内容を表現するテキスト

情報(キーワード)を付与(タグ付け)し、検索時にはタグ付けされたキーワードを検索対象とするものである。テキストによる内容表現は人間の思考感覚と親和性が良く状況に応じた検索者の意図を把握する手段となりうるほか¹⁾、従来のテキスト情報を対象にした検索技術が利用可能であり多くのサービスで使用されている^{2),3)}。しかしながら、タグ付けには人の介在が必要であり、人的コストの問題や付与されるキーワードがタグ付けする人の主観により影響されるなどの問題がある。

マルチメディアデータ検索のもう 1 つの方式は、テキスト情報によるキーワードを用いるのではなく、たとえば画像であれば色や形状などメディアから直接抽出される特徴量そのものを用いて検索を行う方式^{4)~6)}である。データの内容を表現する特徴量は画像処理や音声処理により抽出され、人の介在を必要としないためキーワード方式における問題は存在しない。

後者の方式を用いる多くのシステムではベクトル空間モデルを採用している。各データから抽出される特

[†] NTT サイバースペース研究所
NTT Cyber Space Laboratories

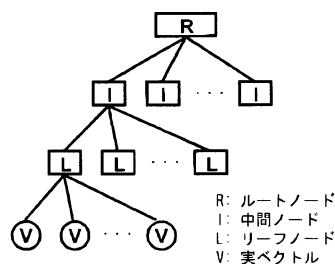


図1 木構造インデックス

Fig. 1 Tree index.

微量は多次元ベクトルとして表現され、データ間の類似尺度としてベクトル間の距離が用いられる。しかしながら、データの大容量化および特徴量を表現するベクトルの高次元化により、ベクトルデータへのアクセスコスト、およびベクトル間の距離計算コストが大きい問題がある。これらのコストは検索性能に大きな影響を与えるために、高次元大容量ベクトルデータに対して高速な検索を実現することがマルチメディアデータベースを実現するうえでの大きな課題の1つであるといえる。

この課題を解決するために様々なインデックス方式^{7)~19)}が提案されている。その多くのものはデータ空間（またはデータ集合）をある基準に従って再帰的に分割し、部分空間（または部分集合）を木構造により管理する。木構造インデックスは、図1に示すように特徴量を表現するベクトルデータである実ベクトル、実ベクトルへのポインタを格納するリーフノード、リーフノードおよび中間ノードへのポインタを格納する中間ノード、木構造の頂点に位置するルートノードより構成される。検索時には上位階層より、つねに検索キーからより近い中間ノードを優先的に掘り下げるにより、不必要な距離計算を省略（枝刈り）して検索効率を上げることが可能である¹⁸⁾。しかしながら、大容量高次元ベクトルデータに対する木構造インデックスでは、インデックス容量が大きくなるためにインデックスのすべてをメモリ上に実現することはきわめて困難であり、一部またはすべてをディスク上に配置することが必要になる。ディスクI/Oコストは非常に高いため、メモリ空間を有効に利用しディスクI/Oを抑えることにより、距離計算コストとディスクI/Oコストの双方を考慮したインデックス方式を実現することが重要である。

本論文では、(1) 実ベクトルをディスク上に配置し、(2) リーフノードおよび中間ノードの構成情報を圧縮してメモリ上に配置する、動的な木構造インデックス VA-TREE を提案する。また、実際の画像検索シ

ステムで用いられている色相、輝度、および形状特徴量を用いて、距離計算回数と実ベクトルアクセス回数の観点から評価実験を行った結果について述べる。VA-TREEは実ベクトルを圧縮してフラットな構造で管理する VA-File と比較して、各次元軸を4ビットに圧縮するとき、データ分布に偏りの強い色相特徴量では84.5%の距離計算回数の削減と99.4%の実ベクトルアクセス回数の削減が可能となるなど、いずれの特徴量についても優位性を確認した。一方、VAM Split R-tree との比較では、VAM Split R-tree が静的インデックスであることにより、1回のディスクアクセスで複数の実ベクトルをアクセス可能であることを考慮すると、色相特徴量と輝度特徴量では VAM Split R-tree に優位性があるが、データ分布の偏りが弱い形状特徴量においては4.3倍の距離計算回数を必要とするものの、ディスクアクセス回数では62.7%の削減が可能であり VA-TREE の優位性を確認した。また、検索性能に大きな影響を与える実ベクトルアクセス回数において、実験に用いた各特徴量間で VA-File では4.4倍、また VAM Split R-tree では45.1倍の開きがあるのに対し、VA-TREE では2.7倍の開きにとどまっており、データ分布の偏りによる影響を受けない優位性を確認した。

以下、2章において距離計算コストを削減するために有効な木構造インデックス方式と、ディスクI/Oコストを削減するために有効なベクトル圧縮方式における関連研究を述べる。3章において提案する VA-TREE の構造、構築方法および探索方法について述べる。4章において評価実験結果と他のインデックス方式との比較結果について述べ、5章において本論文をまとめる。

2. 多次元空間インデックス

2.1 木構造インデックス

木構造インデックスは、R-tree⁷⁾や k-d-tree⁸⁾に代表されるような低次元空間において有効性を示すインデックス手法をベースにしているものが多く、基本的に分割時において任意の1軸について垂直分割していく方式であり、分割方法（分割軸の選択と分割位置）および中間ノードの構成情報（部分空間の形状）に違いがある。

SS-tree⁹⁾は、分散が最大となる軸において、分割時に分割対象空間のデータを再投入し分割後の分散の総和が最大になる位置で分割する動的インデックスである。部分空間を多次元矩形ではなく多次元球で管理するところに特徴がある。多次元球は多次元矩形に比べ

より少ない情報量で表現することが可能であり、ディスクページのインプリメントにおいてディスクページあたりの格納ノード数を増やすことができるほか、検索キーとノード間の1回の距離計算コストをより低く抑えることが可能である。しかしながら、SS-treeでは中間ノードが管理する部分領域のオーバーラップが大きく、距離計算回数を増加させている問題がある。SR-tree¹⁰⁾は、部分空間を多次元矩形と多次元球の両方を用いて管理し、検索キーとの距離を両者のより大きい方を採用することにより効果的な枝刈りを行い検索性能を向上させている。

高次元ベクトル空間における木構造インデックスでは中間ノードのオーバーラップが検索効率に大きな影響を与えている。X-tree¹¹⁾は、オーバーラップしている中間ノードをまとめて管理する“supernode”を用いることによりこの問題を回避している。この最適化のためにインデックス構築時間が長い短所があるが、高速な検索性能を示している。VAMSplit R-tree¹²⁾は、最大分散値を持つ軸においてほぼデータを2分する位置で分割する方式であり、部分空間を多次元矩形により管理する。全体空間を再帰的に分割することによりオーバーラップを回避している。VAMSplit R-treeはメモリベースのインデックスであり、また、データの動的な追加には対応していないがインデックスの構築時間が短く、高次元空間における高速な検索方式としてその有効性が報告されている。

これらの木構造インデックスは、特にデータ分布に偏りがあるデータについて有効性を示しており、実際のアプリケーションにおける特徴量ベクトルデータには分布に偏りがある場合が多く適用範囲は広いと考えられる。しかしながら、データの大規模化に加え特徴量の高次元化によりインデックス容量が大きくなることから、インデックスのすべてをメモリ上に実現することはきわめて困難であり一部またはすべてをディスク上に配置することが必要になる。したがって、検索性能がディスクI/Oコストに大きく影響されてしまう問題がある。

2.2 ベクトル圧縮を用いたインデックス

ベクトルデータが一樣に分布している高次元空間では、木構造インデックスが効果的に機能しないという報告がある^{10),13),14)}。これは検索時において枝刈りが効かずに多くの中間/リーフノードおよび実ベクトルとの距離計算を行う必要があることを意味しているが、特に中間ノードの構成情報がディスクに配置されるディスクベースのインデックスではディスクアクセスがランダムになることから大きな問題となる。

VA-File^{13),14)}は、木構造インデックスのように階層的な構造を用いてデータを管理するのではなく、全体空間の各次元軸を等分割してメッシュ状の多次元矩形(セル)を構築し、1階層のフラットな構造で管理するディスクベースのインデックスである(3.1節参照)。VA-Fileでは、実ベクトルを含む各セルをビット列で表現(ベクトル近似)し、これを実ベクトルの圧縮表現としてファイル(VAファイル)に格納する。VA-Fileでの検索はベクトル近似されたビット列に対する全件走査を基本としており、ベクトル近似によりディスクページあたりの格納効率を上げるとともに、ディスクへのアクセスをシーケンシャルにすることによりディスクI/Oコストを抑えている。また、近似データによる枝刈りによりランダムアクセスとなる実ベクトルアクセス回数を抑えることにより、一様分布である高次元データにおいてX-treeに対する優位性を示している。しかしながら、全件走査であるために距離計算コストがデータ数に比例してしまう問題がある。

部分空間符号化法¹⁵⁾は、実ベクトルデータを用いて構築されるR-treeなどの木構造インデックス(Real Part)の各最小包囲領域を、VA-Fileと同様にメッシュ状に分割したセルを表現するビット列を用いて近似(仮想包囲領域)し、仮想包囲領域から構成される木構造(Virtual Part)を構築する手法である。検索時には、実ベクトルとVirtual Partが使用される。部分空間符号化法では、SR-treeなどのディスクベースの木構造インデックスにおいて、ノードの矩形情報を圧縮することによりディスクページあたりの格納効率を上げ、ディスクアクセス回数の大幅な削減を実現している。

ベクトル圧縮法はディスクベースのインデックスにおいて有効性を示しているが、ディスクI/Oコストは非常に高いため、メモリ空間を併用利用するインデックスにおいてもインデックスのできるだけ多くの部分をメモリ上に構築するうえで重要な手法であるといえる。

3. VA-TREE

本論文で提案するデータ圧縮型動的インデックスVA-TREEは、VA-Fileのセル構築法(空間分割法)およびベクトル近似法を階層構造に拡張することにより、中間ノードおよびリーフノードの矩形情報の容量を削減しメモリ上に配置し、実ベクトルをディスク上に配置する。検索時には、メモリ上に配置された中間ノード/リーフノードを用いた枝刈りにより実ベクトルのアクセス回数を抑えてディスクI/Oを削減するとともに、木構造である特性を活かして特に偏りの強

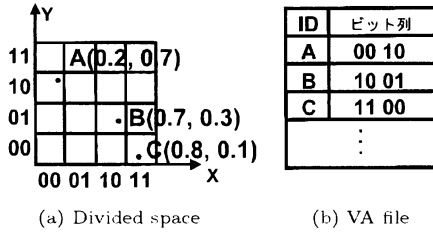


図2 VA-Fileの構造
Fig. 2 Structure of VA-File.

いデータに対して距離計算回数の削減を実現する。

3.1 VA-Fileの構造

VA-Fileは、多次元ベクトル空間の各次元軸を軸ごとに異なる分割数により等分割しメッシュ状のセルを構築し、各セルを表現するビット列によりベクトルデータを近似表現する。

d 次元のベクトルデータ p_i ($1 \leq i \leq N$: N はデータ総数)の次元軸 j 成分を b_j ($1 \leq j \leq d$) ビットで表現する総ビット長 b ($b = \sum_{j=1}^d b_j$) を用いて表現するとき、 b_j は式 (1) により決定される。

$$b_j = \left\lfloor \frac{b}{d} \right\rfloor + \begin{cases} 1 & \text{if } j \leq (b \bmod d) \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

ベクトルデータ p_i が次元軸 j ($1 \leq j \leq d$) において値域 $[0, w_j]$ に存在し、 p_i の次元軸 j における要素値を p_{ij} とするとき、 p_{ij} の次元軸 j のビット列表現は w_j を 2^{b_j} 分割したときの区間番号 m_{ij} ($0 \leq m_{ij} \leq 2^{b_j} - 1$) と等価になる。

$$m_{ij} = \left\lfloor \frac{p_{ij}}{w_j / 2^{b_j}} \right\rfloor - \begin{cases} 1 & \text{if } p_{ij} = w_j \\ 0 & \text{otherwise} \end{cases} \quad (2)$$

この m_{ij} を全次元軸についてビット列として表現し、さらにそれらのビット列を連結したものが p_i のベクトル近似表現となる。つまり、 p_i のベクトル近似表現は、メッシュ状に分割したセルを column-wise にオーダリングし、 p_i を含むセルの位置を b ビットを用いて2値表現したものに等しい。

図2は2次元空間におけるベクトルデータを4ビット ($b = 4$) を用いて近似表現した例である。たとえば、ベクトルデータ A、B、Cが、各次元軸の値域が $[0, 1]$ であるベクトル空間に存在し (図2(a))、各次元軸を2ビット ($b_j = 2$: $1 \leq j \leq 2$)、すなわち各軸を4分割にしてベクトル近似するとき、データ A は X 軸については区間0に存在するためにビット列は“00”、Y 軸については区間3に存在するためにビット列は“10”となり、データ A を示すビット列は“0010”となる。VA-Fileでは、以上のようにベクトル近似された実ベ

クトルのビット列をファイル (VA ファイル) に格納し (図2(b))、このビット列に対して全件走査を行う。

3.2 VA-TREEの構造

VA-TREEの各階層におけるベクトル圧縮表現はVA-Fileのベクトル近似手法を基本としており、式 (1) を用いて各次元軸のビット長を決定する。さらに、VA-TREEでは、ベクトル近似 (以降、VA-TREEではベクトル圧縮と呼ぶ) されたビット列が示すセルを木構造インデックスの MBR (Minimum Bounding Rectangle) と見なし、任意の階層において同じビット列表現を持つ実ベクトルが閾値 (δ) 個以上存在するとき、すなわち、同じセルに δ 個以上の実ベクトルが存在するときに、そのセルを新たな全体空間と見なして同様に繰り返し分割することにより木構造へ拡張する。

また、各階層におけるビット列の表現は、以下の理由により、上位ノードが示す空間を全体空間と見なしセルを構築したときの上位ノード内における相対位置表現とする。

(1) 構造の簡易化

上位ノードが示す空間内における相対位置を示す表現とすることで任意の階層におけるセルを固定長ビットにより表現することが可能となり記憶構造を簡易化できる。なお、相対位置を用いる表現方式はBD木¹⁹⁾においても領域式として提案されているが、領域式では階層に対応した可変長ビットを用いており構造が複雑になってしまっている。

(2) 必要ビット数の削減

相対位置表現を用いることで、より上位のノードにおいて、絶対位置表現に比べてセルを表現するために必要なビット数を削減することが可能となる。

(3) スケーラビリティのある動的追加

絶対位置表現は、データの総数および偏りに応じて軸分割数を動的に変更する必要があるため木構造全体への波及が大きい。上位ノード内の相対位置表現とすることで、分割が局所的になりスケーラビリティが向上する。

したがって、ベクトルデータ p_i が次元軸 j ($1 \leq j \leq d$) において値域 $[0, w_j]$ に存在するとき、任意の階層 l における p_i の次元軸 j における要素値 p_{ij} が存在する区間番号 m_{ij}^l は式 (3) で求められる。

$$m_{ij}^l = \left\lfloor \frac{p_{ij} - CS_j^{l-1}}{w_j^l} \right\rfloor - \begin{cases} 1 & \text{if } p_{ij} = w_j^0 \\ 0 & \text{otherwise} \end{cases} \quad (3)$$

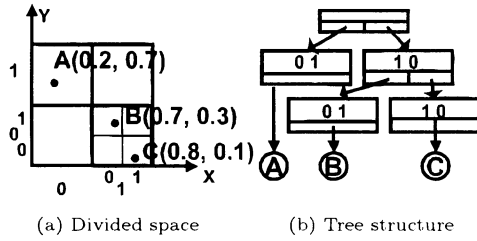


図3 VA-TREEの構造
Fig. 3 Structure of VA-TREE.

$$\text{ただし, } w_j^l = w_j^{l-1}/2^{b_j}$$

$$CS_j^{l-1} = w_j^{l-1} * m_{ij}^{l-1}$$

図3は図2と同じ2次元空間におけるベクトルデータを2ビット ($b = 2$) を用いて圧縮表現するときのVA-TREEの構造を示している。ベクトルデータA, B, Cの各次元軸を1ビット ($b_j = 1; 1 \leq j \leq 2$) でベクトル圧縮するとき、第1階層におけるベクトル圧縮されたビット列はそれぞれA(01), B(10), C(10)である。 $\delta = 2$ とするとB, Cは同一のセルに存在し同じビット列表現になるため分割が生じる。このとき、第2階層においてはX軸が値域[0.5,1], Y軸が[0.0,0.5)であるセルを全体空間と見なして、B, Cの各軸の値を補正し第1階層と同様にベクトル圧縮を行う。この結果、第2階層におけるB, Cのベクトル圧縮表現はB(01), C(10)となる。以上のように構築される木構造を模式的に表したものが図3(b)である。

なお、上位ノード内の相対位置を用いるという各次元軸におけるビット列生成法の考え方は部分空間符号化法¹⁵⁾と同じであるが、部分空間符号化法では任意の階層におけるMBRを上位ノード内における2つのセルの位置情報を用いて表現するのに対し、VA-TREEのすべてのノードは必ず1つのセル自身の位置情報として表現される。また、VA-TREEでは次節に示すように木構造の作成の際にReal Partを必要とせずに直接木構造を作成することが可能である。

3.3 構築処理

VA-TREEは動的インデックスであるために、全データに対するインデックスへの挿入処理がインデックスの構築処理になる。図4にVA-TREEの任意の階層における中間ノードへの挿入アルゴリズムを示す。

VA-TREEの最も特徴的な点は、各次元軸が軸単位にすべての階層において同数に等分割されること、およびノードの矩形を表現するビット列に上位ノード内の相対位置表現を用いることである。このため、式(1)および式(3)を用いてベクトルの圧縮表現を求める処理に必要な任意の階層におけるノード矩形

```

Procedure StoreVectorToVNode( vaNode, insertVector, level )
1:  if level > deepestLevel then
2:    MakeDivisionInfo( TOTALWIDTHs, level, TOTALBITS);
3:  end;
4:  vaVector = MakeBitVector( insertVector, level );
5:  SearchInVNode( vaNode, vaVector );
6:  if NOT found then
7:    appendVectorToVNode(vaNode, vaVector, level );
8:  else
9:    if foundVAVector.type = VANODE then
10:     StoreVectorToVNode( childVNode, insertVector, level+1);
11:    else
12:     childVNode = MakeChildVNode(vaNode, level+1);
13:     StoreVectorToVNode( childVNode, insertVector, level+1);
14:    end;
15:  end;
16: end.

```

図4 挿入アルゴリズム

Fig. 4 Insert algorithm.

の任意の軸の矩形辺長は、次元数、各次元軸の全体空間における値域幅 (TOTALWIDTHs), 階層の深さ (level), およびベクトル圧縮表現のための総ビット長 (TOTALBITS) により決定される。したがって、ビット列生成に必要な部分的な計算結果 (たとえば、矩形辺長や開始位置) をあらかじめ求めておくこと (ステップ2) や、それらの計算結果を同じ値域を持つ複数の次元軸で共有することが可能であり、ベクトル圧縮を用いることによる計算のオーバーヘッドを抑えることができる。

ステップ4において挿入階層 level における挿入ベクトルデータ insertVector を圧縮表現しビット列 vaVector を生成する。

各階層における圧縮表現は分割領域 (たとえば、 k 次元空間が各次元軸2分割されるとすると 2^k) の数分がありうることになるが、データ分布の偏りが強いとき、より下位階層ではすべての分割領域が使用される確率は低い。このため、VA-TREEの各ノードではデータが存在する部分領域の圧縮表現のみを管理している。ステップ5では、ステップ4で求められたビット列 vaVector がノード vaNode に存在するかを判定する。存在しない場合には、vaNodeに追加して終了する (ステップ7)。

一方、ノード vaNode にビット列 vaVector がすでに存在する場合には、そのノードタイプを判定する。このとき、ノードタイプが VANODE であるときには、さらに下位にノード (中間ノード) を持つことを意味し、LEAF であるときには実ベクトルをポイントすることを意味する (リーフノード)。ノードタイプが VANODE のときには下位ノードに挿入するために、StoreVectorToVNode() を再帰的に実行する (ステップ10)。また、ノードタイプが LEAF であるときには、新しく下位の中間ノード childVNode を作成し、すでに登録されていた実ベクトルを作成した

```

Procedure SearchVATree( vaTreeRoot, queryVector, returnNum )
1:  InsertCandidateNodeList( vaTreeRoot );
2:  while ( CandidateNodeList not empty ) do
3:    item = popCandidateNodeList();
4:    if item.type = VANODE then
5:      childItemList = getChildItemList( item );
6:      for each childItem in childItemList do
7:        childItem.MBR = rebuildMBR( childItem );
8:        childItem.nearestDistance
          = calcNearestDistance( queryVector, childItem.MBR );
9:        InsertCandidateNodeList( childItem );
10:     end;
11:   else
12:     if ResultListKthNearestDistance > item.nearestDistance then
13:       item.vector = readFromDisk( item );
14:       item.nearestDistance
          = calcRealNearestDistance( queryVector, item.vector );
15:       if ResultListKthNearestDistance > item.nearestDistance then
16:         insertResultList( item );
17:         alterResultListNearestKthNearestDistance();
18:       end;
19:     end;
20:   end;
21: end;
22: end.

```

図5 探索アルゴリズム
Fig. 5 Search algorithm.

childVANodeに挿入し（ステップ12）、挿入ベクトルデータ insert Vector についても childVANodeに挿入する（ステップ13）。なお、図4では、説明を簡略化するために複数の実ベクトルを含むリーフノードの処理や同一ベクトルの管理を省略している。

3.4 探索処理

探索処理では、検索キーベクトルと各ノードとの距離を基に、下位方向に探索範囲を掘り下げる対象ノードに優先度を持たせることにより枝刈りを行う¹⁸⁾。文献13)では、近似ベクトルで枝刈りを行った後にまとめて実ベクトルにアクセスする手法と、近似ベクトルでの枝刈り中に実ベクトルにアクセスする手法を示しており、前者の手法が有効であると結論づけている。近似ベクトルは木構造インデックスにおける多次元矩形と考えることができるため、前者の方式では検索キーから k 件目の検索結果を保持するセルへの最遠距離より近い最近距離を持つセルについても候補として残し、最後にまとめて実ベクトルにアクセスすることになる。しかしながら、VA-TREEはリーフノードが管理する部分領域である多次元矩形面積がデータ分布の影響を受けるために近似されたノードのみでは十分に絞り込みが行えないため後者の手法を用いる。

図5に探索アルゴリズムを示す。探索処理では、検索キーからの距離により優先度をつけて探索範囲を絞り込む候補ノードを管理するリストである CandidateNodeList、および最終結果である実ベクトルを管理するリストである ResultList を用いる。

ステップ1において、ルートノード vaTreeRoot を候補ノードリスト CandidateNodeList に設定する。ステップ3では、CandidateNodeList より先頭のノード、つまり候補ノードリストの中で検索キーより最も近い距離にあるノードを抽出し item に設定する。

候補ノードリスト CandidateNodeList には中間ノードまたはリーフノードが格納されている。ステップ3において抽出したノード item のタイプが中間ノード (VANODE) である場合には、中間ノード item が管理する複数の下位ノード childItemList を抽出し（ステップ5）、抽出したすべての下位ノードについて、領域矩形 childItem.MBR を復元し（ステップ7）、検索キー queryVector と復元された領域矩形との距離 childItem.nearestDistance を計算して（ステップ8）、候補ノードリスト CandidateNodeList へ挿入する（ステップ9）。

一方、ステップ3において抽出したノード item のタイプがリーフノード (LEAF) である場合には、すでに計算されている検索キーと領域矩形との距離 item.nearestDistance が最終結果リスト ResultList の k 番目 (k は検索結果の返却数) の距離 ResultList-KthNearestDistance より小さいときのみ、ディスクに格納されている実ベクトル item.vector を読み出し（ステップ13）、検索キー queryVector と読み出した実ベクトル item.vector の実距離 item.nearestDistance を計算する（ステップ14）。その後、ステップ14で求められた検索キーとの実距離 item.nearestDistance を、再度、最終結果リストの k 番目の距離 ResultList-KthNearestDistance と比較を行い、距離が小さいときのみ最終結果リスト ResultList に挿入し（ステップ16）、最終結果リストの k 番目の距離 ResultList-KthNearestDistance を更新する（ステップ17）。

4. 評価

VA-TREE を SUN Ultra60 (UltraSPARC-II 450M) 上に実装し、 k 近傍検索実験 (k を20に設定)を行った。実験に用いたデータは、風景などを収めた日常写真のオブジェクト画像⁴⁾から抽出される16次元の色相 (Hue) 特徴量、16次元の輝度 (Intensity) 特徴量、24次元の形状 (Shape) 特徴量の実データである。本章では、VA-TREEにおけるデータ総量、およびベクトル圧縮表現の総ビット長の違いに対する特性を確認することを目的として、構築処理についてはインデックス容量および構築時間の観点から、また探索処理についてはノードおよび実ベクトルとの距離計算回数、ディスク上に存在する実ベクトルのアクセス回数の観点から色相特徴量を用いて評価実験を行った結果について述べる。また、色相特徴量、輝度特徴量、形状特徴量のそれぞれの特徴量を用いてインデッ

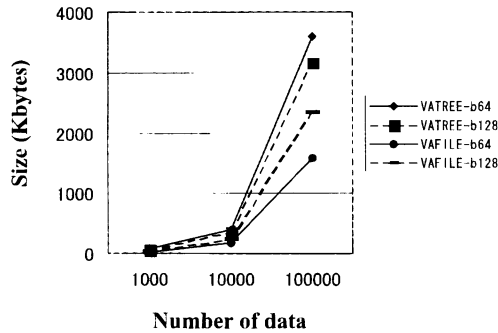


図 6 インデックス容量

Fig. 6 Index size.

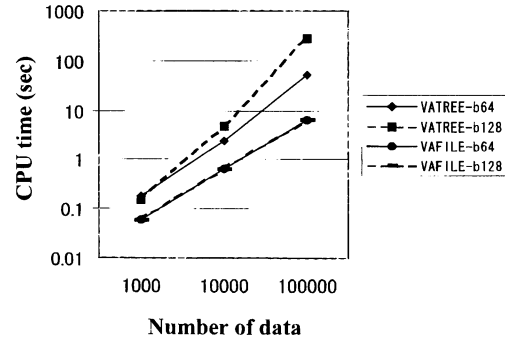


図 7 インデックス構築時間

Fig. 7 Construct time.

クス容量、距離計算回数および実ベクトルへのアクセス回数の観点から VA-File、VAMSplit R-tree と比較した結果についても示す。なお、インデックス容量は UNIX の top コマンドを用いて計測している。top コマンドの結果が示すメモリ使用量には、インデックスのために使用するデータ領域のほかにプログラムの実行モジュールが使用する領域が含まれている。このため、インデックス容量として、データ挿入後のメモリ使用量からデータ挿入前のメモリ使用量を差し引いた値を用いている。

4.1 構築処理の評価

4.1.1 インデックス容量と構築時間

VA-TREE ではインデックスの実ベクトルを除く部分をメモリ上に配置するため、メモリ上の容量が重要な評価指標となる。色相特徴量を対象にデータ総数の変化におけるインデックス容量の変化を図 6 に、データ全件を挿入するのに要した CPU 時間を図 7 に示す。リーフノード内の最大データ個数を 2 に固定し、総ビット長を 64、128 とする 2 つのベクトル圧縮を用いて測定を行った。図 6、図 7 において suffix の b64、b128 はベクトルの圧縮表現に用いた総ビット長がそれぞれ 64/128 ビットであることを示す。また、参考値として VA-File における結果についても示す (prefix が VAFILE)。なお、VA-File についても実ベクトルをディスク上に配置し近似ベクトルを格納する VA ファイルをメモリ上に配置する構成で実装を行い測定を行っている。

インデックス容量は、図 6 に示すように、VA-File も含めていずれのパラメータにおいてもデータ総数の増加に対して急激な増加を示しているが、2 つの原因が考えられる。

1 つは動的インデックスのためのオーバーヘッドである。VA-TREE は動的インデックスであること、およ

び 3.3 節で述べたように、ノード内データの充填率を高くすることを目的としてノード内のデータが増えるたびに領域の拡張を行うが、この管理のオーバーヘッドが影響している (VA-File についても動的追加に対応した実装を行っている)。

もう 1 つは、VA-TREE における木の構築アルゴリズム自体の問題である。インデックスの容量はノード数に大きく依存する。VA-TREE における分割法は 2^k 空間分割法に類似しているが、 2^k 空間分割法ではデータ分布の偏りに対して木のバランスが悪くなりノード数が増えることが知られている。たとえば VATREE-b64 では、データ総数 1000、10000、100000 に対してノード数がそれぞれ 63、885、8183 とデータ量の増加に対してほぼ比例増加しておりインデックス容量を増加させる原因になっている。

VA-TREE の構築時間については、緩やかではあるが指数増加を示している。これは、任意のノードへのデータ挿入時に、圧縮されたビット列表現がそのノードにすでに登録されているかを判定する必要があるためである (図 4 ステップ 5)。特に VATREE-b128 では、1 階層における分割セル数が増えるが、これは VA-TREE では枝数が増えうることを意味している。したがって、登録有無判定のためのビット列比較回数が増加する。この問題については、現状の実装ではシーケンシャルに比較を行っているが、2-3 木などを採用することによる削減が期待できる。

4.1.2 圧縮効果

表 1 にデータ総数 10 万件の色相、輝度、形状の 3 つの特徴量データについて、各次元軸を 4 ビットで表現するときのメモリ上のインデックス容量の比較を示す。表 1 における各値は上段が実測値であり下段が VA-TREE の容量を 100 としたときの相対値である。VA-TREE (no compression) はノードの表現を圧縮

表 1 インデックス容量の比較
Table 1 Comparison of index size.

	Upper: Kbytes Lower: Ratio to VA-TREE(%)		
	Hue	Intensity	Shape
VA-File	1584 43.9	1584 47.7	1968 98.8
VA-TREE	3608 100.0	3320 100.0	1992 100.0
VA-TREE (no compression)	16045 444.7	17083 514.5	20002 1004.1

せずに対角に位置する2つのベクトル値で表現したときの値である。

VA-TREE は非圧縮の状態に比較して色相特徴量において 22.5%、輝度特徴量において 19.4%、形状特徴量において 10.0%の容量にとどまっており、圧縮が有効であることを示している。なお、形状特徴量において VA-TREE の容量は VA-File とほとんど等しい。これは、各軸を 4 ビットを用いて表現すると空間が 1 回の分割で 2^{48} 個に分割されるが、形状特徴量におけるデータ分布の偏りが弱く、VA-TREE の木構造が深くならずに VA-File とほとんど同じ形状になるためである。

4.2 探索処理の評価

4.2.1 距離計算回数と実ベクトルアクセス回数

図 8 に 4.1.1 項の条件で構築した VA-TREE および VA-File における探索処理での距離計算回数を、図 9 に実ベクトルへのアクセス回数を示す。測定結果は挿入データからランダムに選択した 100 個の検索キーによる検索結果の平均である。なお、距離計算回数には実ベクトルとの距離計算回数を含んでいる。

図 8 に示すように、VA-File では全件走査のために距離計算回数がデータ総数に対して比例増加するのに対し、VA-TREE では計算回数の増加が抑えられている。また、実ベクトルへのアクセス回数についても VA-TREE はデータ総数の増加にほとんど影響を受けておらず有効性を示している（図 9）。

データ総数 10 万件における VATREE-b64 と VATREE-b128 を比較すると、距離計算回数は VATREE-b128 が VATREE-b64 より約 3 倍多いが、逆に実ベクトルのアクセス回数は VATREE-b64 が VATREE-b128 より約 4 倍多い。これは、ベクトル圧縮した中間ノードおよびリーフノードにおいてできるだけ距離計算を行い枝刈りを行うことでディスクアクセス回数を抑える VA-TREE の性質が出ているが、VATREE-b128 の方がより小さな多次元矩形面積を持つリーフノードを構成できるために枝刈りが有

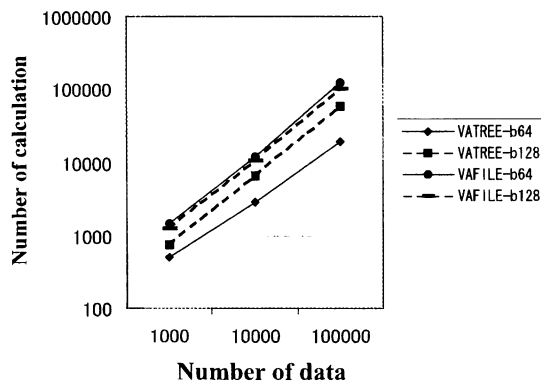


図 8 距離計算回数

Fig. 8 Number of distance calculation.

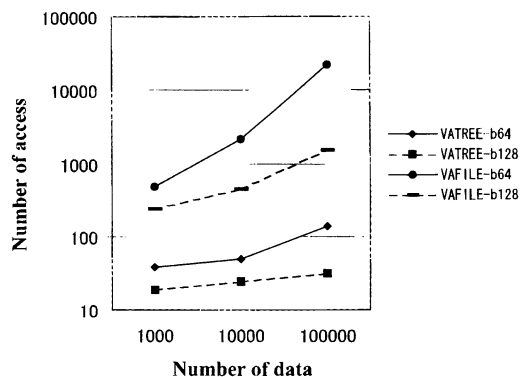


図 9 実ベクトルアクセス回数

Fig. 9 Number of accessing feature vectors.

効に機能しているためと考えられる。しかしながら、VATREE-b128 では、1 階層における分割数が多く枝数が多くなり距離計算回数の増加をもたらしている。このように VA-TREE ではディスク I/O と距離計算回数がトレードオフの関係にあるため、ベクトル圧縮の表現ビット数を変更することにより最適化を行うことが可能である。

4.2.2 他手法との比較

データ総数 10 万件の色相、輝度、形状の 3 つの特徴量データについて、VA-TREE、VA-File、VAM Split R-tree の比較を行った。表 2 に距離計算回数の比較結果を、また表 3 に実ベクトルアクセス回数の比較結果を示す。表 2、表 3 の各値は上段が実測値であり下段が VA-TREE を 100 としたときの相対値である。なお、VA-TREE および VA-File は各次元軸について 4 ビットを用いるベクトル圧縮とし、VAM Split R-tree においてはノードサイズを 20 とした[☆]。また、VA-TREE

[☆] VAM Split R-tree はメモリベースの実装であるが、実ベクトルがディスクにあるものとして比較を行った。

表 2 距離計算回数の比較

Table 2 Comaprison of number of distance calculation.

Upper: Number of times Lower: Ratio to VA-TREE(%)			
	Hue	Intensity	Shape
VA-File	121688 646.2	124735 418.1	104919 104.6
VA-TREE	18829 100.0	29832 100.0	100261 100.0
VA-TREE(L20)	18817 99.9	29815 99.9	100261 100.0
VAMSplit R-tree	629 3.3	2355 7.9	23352 23.3

表 3 実ベクトルアクセス回数の比較

Table 3 Comaprison of number of accessing feature vectors.

Upper: Number of times Lower: Ratio to VA-TREE(%)			
	Hue	Intensity	Shape
VA-File	21668 15594.2	24735 7453.8	4919 1303.3
VA-TREE	139 100.0	332 100.0	377 100.0
VA-TREE(L20)	214 153.7	844 254.3	377 100.0
VAMSplit R-tree	449 323.3	1473 443.9	20229 5359.6

は 4.1.1 項同様、リーフノード内の最大データ個数を 2 としているが、木の形状はベクトル圧縮表現のビット数のほかにリーフノード内の最大データ個数にも影響を受ける²⁰⁾ために、VAM Split R-tree との適正な比較のためリーフノード内の最大データ個数を 20 にしたものについても測定を行った [VA-TREE (L20)].

表 2 に示すように、VAM Split R-tree は他の手法に比べて極端に距離計算回数が少ない。特に色相特徴量については、VA-TREE の 3.3% の計算量しか要していない。これは、色相特徴量におけるデータ分布に偏りが強く平衡木である VAM Split R-tree が有効に機能していることを示している。また、VA-TREE および VA-File では距離計算時に圧縮ビット列からセルへのノードの復元操作が必要であり、実際の計算量の差はさらに大きいものとなる。

VA-TREE と VA-File の比較では、いずれの特徴量データにおいても VA-TREE の方が距離計算回数が少ないが、特に色相特徴量においては VA-TREE は VA-File の 15.5% の距離計算回数にとどまっており、VA-File を階層構造化することによる距離計算量削減効果が確認できる。

一方、実ベクトルアクセス回数はいずれの特徴量データにおいても VA-TREE が他の手法より少ない。VA-File では、データ分布の偏りが強い色相特徴量と

比較的偏りが弱い形状特徴量とを比較すると、形状特徴量におけるアクセス回数が極端に少なく、実験結果は VA-File が一様分布データに有効な手法であることを裏付けている。しかしながら、形状特徴量においてさえも VA-TREE に比較して 10 倍以上のアクセス回数を要しており VA-TREE の優位性を示している。

VAM Split R-tree では形状特徴量でのアクセス回数が色相特徴量や輝度特徴量に比較して極端に多く、実験結果より木構造インデックスが偏りの強いデータセットに有効であるということが確認できる。VAM Split R-tree では偏りの強い色相特徴量においてさえも VA-TREE より 3 倍以上の実ベクトルアクセス回数を要しており VA-TREE の優位性を示している。ただし、VAM Split R-tree は静的インデックスであり、ディスク上の実ベクトルの配置を最適化できることを考慮しなければならない。つまり、VA-File や VA-TREE の実ベクトルアクセスがディスクへのランダムアクセスになるのに対し、VAM Split R-tree では 1 回のページアクセスにより 1 つのリーフノードに含まれる複数の実ベクトルを読み込むことができるため、ディスクアクセス回数としてはさらに少なくなる。ディスクアクセス回数は、ディスク上における実ベクトルの配置法やディスクページの容量および実ベクトルの容量に依存するが、一般的には 1 つのリーフノード配下のすべての実ベクトルは 1 つのディスクページに配置されるために、1 回のページアクセスによりリーフノード配下のすべての実ベクトルを読み込むことが可能である。このため、表 3 における VAM Split R-tree のディスクアクセス回数は、実測値をノードサイズ（本実験においては 20）で割った値と考えることができる。したがって、色相、輝度、形状の各特徴量におけるディスクアクセス回数は、それぞれ 23 回、74 回、1012 回となり、色相特徴量や輝度特徴量では VAM Split R-tree が VA-File や VA-TREE に比較して有効な方式であるといえる。しかしながら、形状特徴量においてはこの点を考慮しても、なお VA-TREE は、リーフノード内の最大データ個数が 20 のときも含め、VAM Split R-tree の 37.3% のディスクアクセス回数にとどまっており偏りの弱いデータセットについては VA-TREE が優位であることを示している。これは VAM Split R-tree の空間分割方法が任意の 1 軸における分割であり、最下位のリーフノードを考えると、高次元ベクトル空間においては分割されない軸が残ってしまい、距離の離れたベクトルを同じリーフノードで管理しているためであると考ええる。

また、実ベクトルアクセス回数において、特徴量の

違いにより VA-File では 4.4 倍、また VAM Split R-tree では 45.1 倍の開きがあるのに対し、VA-TREE では 2.7 倍の開きにとどまっている。実ベクトルアクセスはコストの高いディスク I/O につながることから検索性能に大きな影響を与える。VA-TREE では他の手法に比較してデータ分布の影響を受けずに低く抑えられており、その有効性を示している。

5. おわりに

大容量のマルチメディアデータベースに必須な高速多次元空間インデックスを実現するためには、ディスク I/O とベクトル距離計算回数の双方を考慮した削減が重要である。

本論文では、実ベクトルをディスク上に配置し、木構造の中間ノードおよびリーフノードの矩形情報を圧縮してメモリ上に配置する動的多次元空間インデックス VA-TREE について提案した。また、VA-TREE を実装し、画像検索システムで用いられている色相、輝度、および形状特徴量の実データを用いて距離計算回数とディスクへのランダムアクセスとなる実ベクトルアクセス回数の観点から評価実験を行い提案手法の有効性を確認した。

多次元空間インデックスにおいては、データ量、次元数、およびデータ分布が重要な意味を持つ。今後さらに、高次元データや異なるデータ分布、1000 万オーダーの大量データを用いて提案手法の検証を行う。特にデータの高次元化については重要な問題であると考えられる。データが高次元化すると 1 階層における分割セル数が多くなるため、データ分布の偏りにかかわらず構造が VA-File に近づき距離計算回数の削減が期待できなくなるためである。この問題に対し、データ分布の偏りを利用した軸分割数の決定を検討していく予定である。

謝辞 有益なご助言、コメントをいただきました査読者の方々に深く感謝いたします。

参考文献

- 1) 宮川, 清木, 宮原, 北川: 画像データベースを対象とした意味的連想検索の高速化アルゴリズム, 情報処理学会論文誌: データベース, Vol.40, No.SIG1 (TOD5), pp.1-10 (2000).
- 2) <http://www.altavista.com/>
- 3) <http://www.ditto.com/>
- 4) 串間, 赤間, 紺谷, 山室: 色や形状等の表層的特徴量にもとづく画像内容検索技術, 情報処理学会論文誌: データベース, Vol.40, No.SIG3 (TOD1), pp.171-184 (1999).
- 5) Flickner, M., et al.: Query by Image and Video Content: The QBIC System, *IEEE Computer*, pp.23-32 (1995).
- 6) 西原, 小杉, 紺谷, 山室: 時間正規化を用いたハミング検索システム, 情報処理学会情報科学, Vol.30, No.6, pp.27-32 (1999).
- 7) Guttman, A.: R-trees: A Dynamic Index structure for Spatial Searching, *Proc. ACM SIGMOD '84*, pp.47-57 (1984).
- 8) Bentley, J.L.: Multidimensional Binary Search Trees Used for Associative Searching, *Comm. ACM*, Vol.18, No.9, pp.509-517 (1975).
- 9) White, D.A. and Jain, R.: Similarity Indexing with SS-tree, *Proc. 12th Int. Conf. on Data engineering*, pp.516-523 (1996).
- 10) Katayama, N. and Satoh, S.: The SR-tree: An Index Structure for High-Dimensional Nearest Neighbor Queries, *Proc. 1997 ACM SIGMOD*, pp.369-380 (1997).
- 11) Berchtold, S., Keim, D.A. and Kriegel, H.P.: The X-tree: An Index Structure for High-Dimensional Data, *Proc. 22nd VLDB Conf.*, pp.28-39 (1996).
- 12) White, D.A. and Jain, R.: Similarity Indexing: Algorithms and Performance, *Proc. SPIE: Storage Retrieval for Image and Video Database IV*, Vol.2670, pp.62-75 (1996).
- 13) Weber, R. and Blott, S.: An Approximation-Based Data Structure for Similarity Search, Technical Report 24 of ESPRIT project HERMES, No.9141 (1997).
- 14) Weber, R., Schek, H.J. and Blott, S.: A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces, *Proc. 24th VLDB Conf.*, pp.194-205 (1998).
- 15) 櫻井, 吉川, 植村, 児島: 多次元空間における類似探索手法の提案, 情報処理学会技術報告データベースシステム 119-18, pp.103-108 (1999).
- 16) 岩崎: 類似画像検索を実現する距離空間インデックスの実装および評価, 情報処理学会論文誌: データベース, Vol.40, No.SIG3 (TOD1), pp.24-33 (1999).
- 17) Curtis, K., Nakagawa, J., Taniguchi, N. and Yamamuro, M.: Similarity Indexing in High Dimensional Image Space, 情報処理学会技術報告, DPS-82-18, pp.99-104 (1997).
- 18) Roussopoulos, N., Kelly, S. and Vincent, F.: Nearest Neighbor Queries, *Proc. ACM SIGMOD '95*, pp.71-79 (1995).
- 19) 坂内, 大沢: 画像データベース, pp.75-90, 昭晃堂 (1987).
- 20) 吉田, 小西, 赤間, 山室: データ圧縮型インデックス VA-TREE の検討, 情報処理学会技術報告,

DBS-120, pp.29-36 (2000).

(平成 12 年 3 月 20 日受付)

(平成 12 年 6 月 27 日採録)

(担当編集委員 有川 正俊)



吉田 忠城

1989 年宇都宮大学工学部情報工学科卒業。1991 年同大学院工学研究科情報工学専攻修士課程修了。同年日本電信電話(株)入社。メッセージングシステムの研究開発等を経て、

現在はマルチメディア情報検索の研究開発に従事。電子情報通信学会、ACM 各会員。



赤間 浩樹 (正会員)

1988 年東海大学理学部情報数理学科卒業。1990 年同大学院理学研究科数学専攻修士課程修了。同年日本電信電話(株)入社。以来、インテリジェント・ネットワーク向け DBMS

の開発、ニュース・オン・デマンドの研究開発等を経て、現在はマルチメディア情報検索の研究開発に従事。人工知能学会、日本ソフトウェア科学会、ACM 各会員。



谷口 展郎 (正会員)

1992 年東京大学工学部機械工学科卒業。1994 年同大学院工学系研究科機械工学専攻修士課程修了。同年日本電信電話(株)入社。現在、マルチメディア情報の検索および流通に

関する研究開発に従事。



山室 雅司

1985 年早稲田大学理工学部数学科卒業。1987 年同大学院数学専攻修士課程修了。1990 年コロンビア大学大学院電気工学専攻修士課程修了。1999 年博士(工学)早稲田大

学。1987 年日本電信電話(株)入社。入社以来、ネットワークオペレーション情報モデル化・ビジュアル化、データベース設計法の研究に従事。現在、マルチメディア情報検索の研究開発に従事。1994 年電子情報通信学会学術奨励賞受賞。電子情報通信学会、日本ソフトウェア科学会、IEEE-CS 各会員。



申間 和彦 (正会員)

1980 年京都大学工学部電子工学科卒業。同年日本電信電話公社(現 NTT)入社。知識処理用プログラミング環境の研究、大規模クライアントサーバシステムの実用化等を経て、

現在はマルチメディアデータベースの研究開発に従事。