人 工 知 能 72－3
（1990. 9. 12）

暗記学習のための記憶管理方式について

小倉宏明*, 畝見達夫**
長岡技術科学大学
*機械システム工学課程, **計画・経営系

機械学習あるいは知識獲得の手法のうち類推, 事例に基づく推論, 記憶に基づく推論などでは, 経験をそのまま記憶し実行時に利用する. これらを実用的な規模の問題に応用するには, 大量の経験データを扱うための効率の良い計算アルゴリズムの開発が不可欠となる. ここでは, それらの特殊形と考えられる“有限の記憶容量のもとでの暗記学習”について, 要素の重み付けに基づく忘却, および, 類似データ検索のアルゴリズムを提案し, その効率について, 理論, 実測の両面から考察を加える. ここでは, データが実数の場合について 2 種類, 実数の 2 次元ベクトルの場合について 1 種類を提案する. 計算時間は $O(1)$ あるいは $O(\log N)$ である.

ON MEMORY MANAGEMENT ALGORITHMS FOR ROTE LEARNING BY MACHINE

Hiroaki OGURA
ogura@voscc.nagaokaut.ac.jp
Course of Mechanical Systems Engineering

and

Tatsuo UNEMI
unemi@voscc.nagaokaut.ac.jp
Department of Planning and Management Science

Nagaoka University of Technology
1603-1 Kamitomioka-cho, Nagaoka, Niigata 940-21, JAPAN

In analogy, case-based reasoning and memory-based reasoning, the learner memorize its experience without large modification, and recall it to use in future performance. To apply these methods to practical scale of problems, we need some efficient algorithm to process large amount of data. In this paper, we mention “rote learning under ristricted capacity of memory,” and propose algorithms for forgetting by weightening and for retrieval of similar data. We mention two cases, where the domain of data is real number and is two dimensional vector of real numbers. Each of the time complexity of these algorithms is $O(1)$ or $O(\log N)$.

1 はじめに

最近,機械学習あるいは知識獲得の分野では,帰納学習や演繹学習と共に,類推[1],事例に基づく推論[2],記憶に基づく推論[3]などが,研究テーマあるいは応用として多く取り上げられている.前者は,入力された例題の集合をもとに,ある意味で抽象化された表現形式を生成し,後の実行に役立てるのに対し,後者では,ほとんど加工せずにそのまま記憶された経験を実行時に利用するものである.後者では必然的に大量の経験データを扱うための計算アルゴリズムの開発が実用的な応用に際して重要な問題となる.

ここでは,それらの特殊形と考えられる暗記学習について,実装上,問題となる計算課題を解くための効率の良いアルゴリズムを提案し,その効率について,理論,実測の両面から考察を加える.暗記学習では,外部からの入力とシステム側からの出力を網羅的に記憶しておき,次の出力を決定する,あるいは問題解決を行なう際に,現在の問題状況と類似の過去の経験を参考にする.ただし,記憶容量は有限であるためすべての経験を記憶する訳にはいかず,なんらかの指標によって過去の不要と思われる記憶を忘却し,新たな経験の記憶領域として割り当てる必要がある.ここでは,各記憶要素に実数の重みを割り当て,記憶容量を越えた場合には,もっとも重みの低い要素を忘却することとする.また,記憶された経験の中から現状に似かよったデータを検索するという厄介な問題もある.何が似ているかという問題は,何を似ていると考えるのが目的に適合しているか.という視点から考えなければならず,類似度自身の定義が問題領域の性質に依存することになる.すなわち,類似データ検索のアルゴリズムについて一般的に論ずることは困難である.しかし,問題領域で対象となるデータの型によって,ある程度の一般的方法論を展開することは可能である.ここでは,個々の記憶要素に含まれるデータが,1. 実数の場合,および 2. 実数の2次元ベクトルの場合について考える.知識工学や,論理を基礎とする人工知能の一部の分野では記号構造が対象領域となるが,ここでは,それらについては扱わない.

以下では,まず問題を明確化し,ついで各データの形式について,実数に関しては2つのアルゴリズム,2次元ベクトルについて1つのアルゴリズムを提案し考察を加える.

2 問題

制限された記憶容量のもとでの暗記学習では,つぎのような問題を解くアルゴリズムが必要となる.

1. 重みを付けた要素を記憶する.このとき,既に記憶に余裕が無い場合は重みの最も小さい要素を記憶から削除する.
2. 一部の記憶要素の重みを変更する.一度に変更される要素の数は,記憶容量に比べて非常に少ないものとする.
3. 与えられたデータと類似度の高い数個の記憶要素に対して,高い順に,ある手続きを適用する.適用の対象となる要素の個数は,その手続きによって決まるが,その数は記憶容量に比べて非常に少ないものとする.

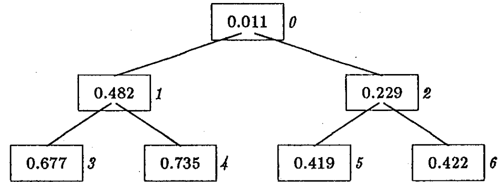


図 1: 記憶容量が7のときの,重み付けに基づく忘却のための2分木構造の例.矩形内の数字は重みを,矩形右側の数字は配列の添え字 (index) を表わす.

1, 2 は制限された記憶容量と重み付けに関するタスクで,3 は類似知識検索に関するものである.重みは実数である.

個々のデータが 1. 実数,および 2. 実数の2次元ベクトルの場合について考える.類似度はともにユークリッド距離に反比例するものとする.すなわち,実数の場合,2つのデータ x と y の間の距離は,

$$\text{距離}: \mathcal{D}(x,y) = |x-y| \tag{1}$$

であり,2次元ベクトルの場合,(x_1,x_2) と (y_1,y_2) の距離は,

$$\text{距離}: \mathcal{D}((x_1,x_2),(y_1,y_2)) = \sqrt{(x_1-y_1)^2+(x_2-y_2)^2} \tag{2}$$

となる.距離が小さいものほど類似度が高いということになる.

3 重み付けに基づく忘却

ここで提案するアルゴリズムは,いわゆるヒープソートの応用である.すなわち,記憶を2分木構造とし,最も重みの軽い要素が絶えず根の位置にくるように管理する.もちろん,記憶容量にまだ余裕がある場合はその必要はない.これにより,記憶が溢れたときに新たな要素を最も重みの軽い要素と置き換えることが容易になる.記憶容量が7のときの2分木構造の例を図1に示す.この図にあるとおり,バランスした2分木を用い,木の分枝点および葉に対応するそれぞれの節点に要素を割り当てる.このとき,各分枝点において親節点の要素の重みが,子節点の重みより小さくなるようにする.兄弟節点の重みの大小関係は任意である.具体的には1次元配列を用い,index が0,すなわち先頭の配列要素を根の節点とし,index が N の要素の子節点を,index が $2N+1$ と $2N+2$ のものとする.

与えられた1つの要素を記憶するアルゴリズムはつぎのとおり.

初期化

1) 変数 LastIndex に記憶の大きさ $|M|$ を代入する.

2) Heap を $|M|$個の要素をもつ配列とする.

HeapInsert(DATA) : DATA を挿入する.

1) LastIndex の値が正のとき,すなわち記憶容量に溢れが生じないときは,Heap[LastIndex] に DATA を格納し,LasiIndex の値を1減らす.

2) 記憶容量に溢れが生じたとき, 最も重みの小さい Heap[0] と DATA の重みを比較する.

3) もし, DATA の重みの方が大きければ DATA を Heap[0] に格納する. そうでなければアルゴリズムを終了する.

4) DATA を格納した場合は HeapCheckDown へ進む.

HeapCheckDown(INDEX) : Heap のデータの並びを下側からチェックする.

1) INDEX を親とする親と子の3つの節からなる部分木を考える. このとき子節のインデックスはそれぞれ, INDEX×2 + 1, INDEX×2 + 2 となる.

2) もし部分木を作成できない場合は, アルゴリズムを終了する.

3) 子節どうしについて重みを比較し, 重みの小さい方の節と親節とでもう一度重みを比較する.

4) もし親節の重みの方が大きければ, その子節と親節を交換する. そうでなければアルゴリズムを終了する.

5) 交換した子節のインデックスを新たな INDEX として, 再び 1) に戻る.

以上

重みの変更操作については, 重みが以前の値よりも増加した場合には上述の HeapCheckDown を実行し, 減少した場合には以下に述べる HeapCheckUp を実行し節点を移動する.

HeapCheckUp(INDEX) : Heap のデータの並びを上側からチェックする.

1) INDEX の節の親節のインデックスを MOTHER とする. 即ち, MOTHER のインデックスは (1−INDEX)/2 である.

2) もし MOTHER の節が存在し, 且つ INDEX の節の値が MOTHER の値よりも小さければ, INDEX の節と MOTHER の節を交換する. そうでなければ, アルゴリズムを終了する.

3) MOTHER を新たなインデックスとして、再び 1) に戻る.

以上

詳しいプログラムについては付録Aを参照されたい.

4 類似データの検索

2節で述べた, データ間の距離に基づく類似データ検索のアルゴリズムについて解説する.

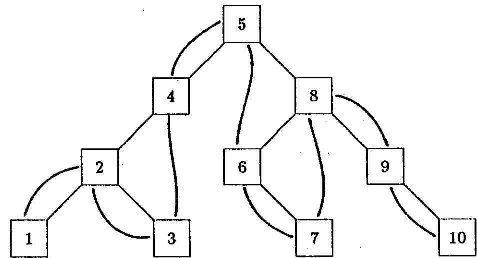


図 2: データが実数の場合の類似データ検索のための2分木構造の例. データが, 例えば 5, 4, 8, 6, 9, 7, 2, 10, 3, 1 の順に与えられたとき

4.1 データが実数の場合

各データ間の距離は式 (1) で定義されるものを用いる. 以下のアルゴリズムは, いわゆる2分ソートの応用である. 各データを2分木構造の各節点に割り当て, 記憶されているデータの順序を保持する. 図2 に2分木構造の例を示す. 前述の「重み付けに基づく忘却」の場合とは異なり, 木は必ずしもバランスしない. また, 配列に格納するのではなく, 各節点を, left, right, lower, upper と名付けられた4つのポインタを含む構造体とし, ポインタによる参照関係によって木構造を構成する. left, right は, それぞれ左側および右側の子節点を, lower, upper は, データの順序が, それぞれ1つ前のもの, および1つ後のものを含む節点を指す. たとえば図2の根にあたる節点 [5] に含まれるポインタは, left = [4], right = [8], lower = [4], upper = [6] である. アルゴリズムはつぎのとおり.

初期化

1) node を weight, value, left, right, lower, upper からなる構造体とする.

2) Root にはデータが何も格納されていないことを示す, NIL を代入する.

Insert(WEIGHT, DATA) : 重みが WEIGHT の値 DATA を挿入する.

1) 引数 WEIGHT, DATA をそれぞれ weight, value に代入した NODE を作る.

2) HeapInsert により, この NODE の weight を heap に格納する.

3) もし, NODE の weight が heap に格納され, 且つ heap に溢れが生じたならば, 溢れた node:OLD を Discard により削除する. また, heap 格納されなかったときは, アルゴリズムを終了する.

4) Root からスタートする. NODE = Root である.

5) Root が NIL, 即ちデータが何も格納されていない場合は, Root が格納場所である.

6) NODE の値 value と DATA の値を比較する. DATA の方が小さければ NODE の左手の子節に進む. そうでなければ, 右手の子節に進む.

7) 進むべき節が NIL となるまで 6) を繰り返す.

8) NODE と親節との位置関係を調べる.

9) もし, NODE が親節の右手にある場合, NODE の upper の示す節点は NODE の親節の upper の示す節点となり, 親節の upper は NODE に変わる. 同時に, NODE の lower は親節である.

10) NODE が親節の左手にある場合は, NODE の lower の示す節点が NODE の親節の lower の示す節点となり, 親節の lower は NODE に変わる. 同時に, NODE の upper は親節である.

Discard(NODE) :節点 NODE を削除する.

1) NODE の右手の子節を RIGHT, 左手の子節を LEFT とする.

2) RIGHT が存在する場合, RIGHT を根とする部分木を考える. RIGHT が左手に子節を持つならば, その部分木の最も左に位置する葉を NODE に移動する. そうでない場合は, RIGHT を NODE に移動する.

3) RIGHT が存在しない場合, LEFT を NODE に移動する.

Search(DATA,FUNC) :DATA に最も近い数を持つ節点から順に検索する.

1) Root からスタートする. NODE = Root である.

2) Root が NIL ならば, アルゴリズムを終了する.

3) NODE の値 value と DATA を比較する. DATA が value より小さければ, NODE の左手の子節に進む. そうでなければ, 右手の子節に進む.

4) 進むべき節点が NIL となるまで 3) を繰り返す.

5) NODE の value と親節の value について, DATA との差をそれぞれとる. 差の小さい方の値を持つ節点が DATA に最も近い節点である. 関数 FUNC でこの節点を処理する.

6) 以後, NODE を中心として 2 つの節点の lower, upper の示す節点を放射状にたどり, DATA との差の小さい順に同様の作業を繰り返す.

7) 進むべき節点が共に NIL となるか, あるいは関数 FUNC での処理が行なわれなくなった場合はアルゴリズムを終了する.

以上

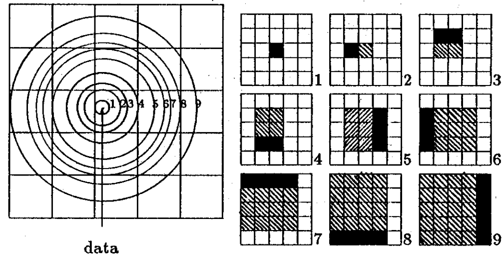


図 3: データが実数の 2 次元ベクトルの場合の類似のデータ検索のためのデータ領域分割の例. 数字は繰り返しのステップ数を表わす. 左図は, ソートの対象となるデータの範囲を, 右図は, 対象となるデータを取り込む cell を表わす.

詳しいプログラムについては付録Bを参照されたい.

一般的にいうと, このアルゴリズムは, データ間に全順序関係が成り立ち,

$$x > y > z \text{ ならば } \mathcal{D}(x,z) > \mathcal{D}(x,y) \quad (3)$$

という規則が成り立つならば, 実数以外の場合にも適用可能である.

4.2 データが実数の多次元ベクトルの場合

つぎに, データが実数の多次元ベクトルの場合について考える. 以下のアルゴリズムはハッシュテーブルの考え方に一部, 類似点がある. すなわち, データの領域を等間隔に区切り区画に対応する n次元配列を用意し, 各要素に対応する区画 (cell) に属するデータをリストにして記録する. 類似データの検索は, 与えられたデータが属する cell から出発し, 必要な個数のデータが得られるまで, 順に隣接する cell に記録されたデータを試す. 以下では簡単のために 2 次元の場合についてアルゴリズムを説明する.

アルゴリズムはつぎのとおり.

初期化

1) node を weight, value, distance からなる構造体とする.

2) 2 次元平面で, x 方向, y 方向ともに区間 $[D_{\min}, D_{\max}]$ で限定された正方形領域を考える.

3) この正方形領域の縦・横を MESH 個に等分し, MESH^2個の cell をつくる.

Insert(WEIGHT, DATA) :重みが weight の値 data を挿入する.

1) 引数 WEIGHT, DATA をそれぞれ weight, value に代入した NODE をつくり, HeapInsert により, この NODE の weight を heap に格納する.

2) もし, NODE の weight が heap に格納され, 且つ heap に溢れが生じたならば, 溢れた node:OLD を Discard により削除

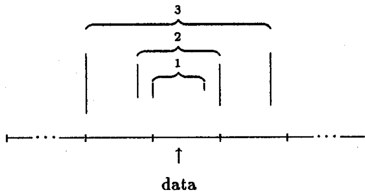


図 4: データが実数の場合の類似のデータ検索のためのデータ領域分割の例. 数字は繰り返しのステップ数を表わす.

する. また, heap に格納されなかった時は, アルゴリズムを終了する.

3) DATA が正方形領域のどの cell に属するかを, CellPosition より求め, その位置の cell に NODE を格納する.

Discard(NODE) ：節点 NODE を削除する.

1) NODE の値 value が正方形領域のどの cell に属しているかを, CellPosition より求める.

2) その位置の cell の中から NODE を削除する.

Search(DATA, FN) :DATA に最も近い数を持つ節点から順に検索する.

1) DATA が正方形領域のどの cell に属するかを, CellPosition より求める.

2) その位置の cell に存在する全ての節点について, DATA との距離を Distance より求め, distance に代入する. list:NODES はこの distance のリストである.

3) DATA と cell の 4 本の境界線との距離を求め, その最も短い距離を半径とする円を作成する.

4) その円の中に存在する全ての節点について distance の値を調べ, 最小の値を持つ節点が DATA に最も近い節点である. 関数 FN でこの節点を処理し, 以後, distance の値の小さい順に同様の作業を繰り返す.

5) 3) で作成した円内の節点を全て調べ終えた場合は, その円と接している cell の境界線を消去して cell の領域を拡張する. 拡張した cell について, 3) からのアルゴリズムを繰り返す. 拡張ができない場合, および, 関数 FN での処理が行なわれなくなった場合はアルゴリズムを終了する.

以上

詳しいプログラムについては付録Cを参照されたい.

1次元の場合, つまり前節で対象としたデータの場合には, 領域の分割および探索の順序は図4のようになる.

5 計算量についての考察

以下では, これまでに解説してきた各アルゴリズムの計算量について考察する.

5.1 重みに基づく忘却アルゴリズムの計算量

必要な記憶容量は, ポインタを要素とする大きさ $|M|$ の配列 Heap および, 1つの大域変数 LastIndex のみであるから, 空間的なコストは $|M|$ に比例すると考えて良い.

つぎに1個のデータを記憶するときの時間コストについて考える. 記憶容量の半分以下の個数までは Heap[LastIndex] に格納するだけであるから時間コストは一定, すなわち $O(1)$ である. 記憶容量の半分から 3/4 までは, 必ず HeapCheckDown の中で1回だけ重みの比較が行なわれる. 3/4 から 7/8 までは1回あるいは2回の比較が行なわれることになる. 重みの分布が一様であると仮定すると, 平均で1.5回の比較がなされると考えてよい. 同様に 7/8 から 15/16 まで, 15/16 から 31/32 まで等について考えると, 一般に, $(2^{n-1}-1)/2^{n-1}$ から $(2^n-1)/2^n$ までのデータを記憶するのに必要な比較の回数は平均で,

$$\frac{|M|}{2^n}\times\frac{n}{2} = \frac{|M|\times n}{2^{n+1}} \tag{4}$$

となる. よって記憶に溢れが生ずるまでの計算時間は, 1個のデータの格納に必要な計算時間を c_1, 1回の比較に必要な計算時間を c_2 とすると,

$$|M|\times\left(c_1+c_2\times\sum_{n=2}^{\lceil\log_2|M|\rceil}\frac{n}{2^{n+1}}\right) \tag{5}$$

となる. 記憶容量に溢れが生じた後では, 新たなに記憶すべきデータの重みがその時点で記憶されているデータの重みの値の範囲内で一様の確率で与えられると仮定すると, 1個のデータを記憶する際に必要な比較の回数は平均で $\lceil\log_2|M|\rceil/2$ となる. すなわち, $|M|$ に比べて十分大きな数のデータを記憶する場合の1個のデータの記憶に必要な計算時間 T_h は,

$$T_h = c_1+c_2\times\frac{\lceil\log_2|M|\rceil}{2} \tag{6}$$

つまり $O(\log|M|)$ である.

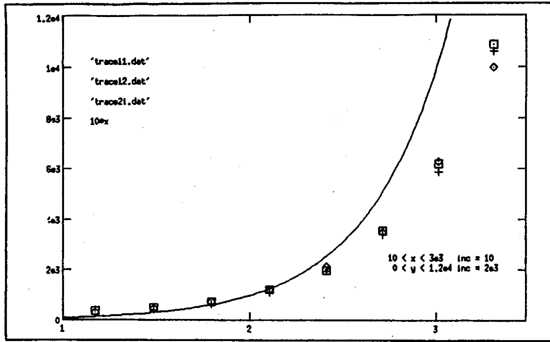
重みの変更に関する計算時間も記憶の場合とまったく同様に考えることができ, 記憶容量に溢れが生じた後での1回の重みの変更に必要な計算時間コストは $O(\log|M|)$ になる.

図5は記憶容量が 15, 31, 63, 127, 255, 511, 1023, 2047 の8種類についてその容量の8倍のデータの挿入を繰り返し行なったときの計算時間の実測値である. 繰り返しは挿入するデータ数の総和が65536となる回数である.

5.2 データが実数のときの類似データ検索アルゴリズムの計算量

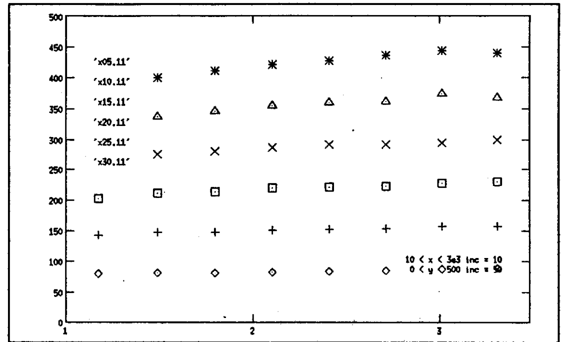
必要な記憶容量は, 4つのポインタを含む構造体が高々 $|M|$個と1つの大域変数 Root のみであるから, 空間的なコストは $|M|$ に比例すると考えて良い.

時間コストは2分木探索の場合と同様となる. すなわち, 2分木の高さが最悪の場合で $|M|$, 最良の場合では $\lceil\log_2|M|\rceil$, 平均



縦軸：時間 [sec]　横軸：記憶容量 (log)
◇ 1 次元 2 分木構造　＋ 1 次元領域分割
□ 2 次元領域分割

図 5: データインサートの計算コスト.



縦軸：時間 [sec]　横軸：記憶容量 (log)
◇ 検索データ数 5 個　＋ 検索データ数 10 個
□ 検索データ数 15 個　× 検索データ数 20 個
△ 検索データ数 25 個　＊検索データ数 30 個

図 6: 2 分木構造による類似データ検索の計算コスト.

でも約 $1.39 \times \lceil \log_2 |M| \rceil$ ([4] p. 81 など) となる. データの挿入では, 記憶容量に溢れが生じた後では, 挿入と同時に削除の動作が起こるため, その計算時間も考慮しなければならない. 忘却のアルゴリズムによって削除すべきデータ含む構造体自身を直ちに得ることができるが, 削除の操作には 2 分木の親節点を見つける必要があるため, 結局 2 分木探索をすることになる. 削除のために木の高さに比例した計算時間が必要となるが, これは計算量としては挿入の場合と同じであるから, 全体の計算量としても $O(\log |M|)$ となる.

実際にはさらに前節の忘却のための計算時間が加算される. $|M|$ に比べて十分大きな数のデータを処理する場合には, 1 個あたりのデータ挿入に必要な平均計算量は $O(\log |M|)$ となる.

類似データ探索では, 最も近いデータを見つけ出すために木の高さに比例した回数の比較が必要となるため必要な平均計算量はやはり $O(\log |M|)$ となる. ただし, 最も近いデータを発見した後は, 探索するデータの個数に比例した計算時間で済む. 図6は記憶容量が 15, 31, 63, 127, 255, 611, 1023, 2047 の 8 種類について 5, 10, 15, 20, 25, 30 個 (記憶容量が 15 のときのみ 15 個まで) の類似データの検索を 10000 回行なったときの計算時間の実測値である.

5.3 データが実数のn次元ベクトルのときの類似データ検索アルゴリズムの計算量

記憶としては, データ領域を区分する n次元配列と個々の配列要素に対応するデータを記憶するためのリストが必要となる. リストの長さの総和は高々 $|M|$ である. 配列の大きさは任意であるが, 大きさは計算量に影響する. ここでは, 単純化のために配列の大きさは全ての次元で等しく, 全体で K^n であると仮定する. すなわち空間的なコストは $O(K^n + |M|)$ となる.

挿入のための計算時間は, 記憶容量に溢れが生じるまでは一定であるが, 溢れが生じた後では, 削除のためにデータを格納したリストの長さに比例した時間が必要となる. リストの長さの平均は $|M|/K^n$ であるから計算量も $O(|M|/K^n)$ となる.

類似データ探索では, 処理の対象となる cell の拡張に伴って多くの計算時間を要することとなる. 1 ステップの拡張で取り込まれる cell 数はステップ数の n乗に比例し, 各ステップで新たに取り込まれたデータのソートが行なわれる. cell 1 個あたりの平均データ数を q とすると, i番目のステップにおける計算量は $O(qi^n \log qi^n)$, 必要なデータ数の検索が終了するまでに Jステップかかったとすると,

$$\sum_{i=1}^{J} qi^n \log qi^n \tag{7}$$

に比例する計算時間を要することになる.

必要なステップ数 J および q は, cell の個数 K^n と記憶容量 $|M|$ および検索すべきデータの個数 N によってきまり,

$$q = \frac{|M|}{K^n} \tag{8}$$

$$J \propto \sqrt[n]{N \times \frac{K^n}{|M|}} \tag{9}$$

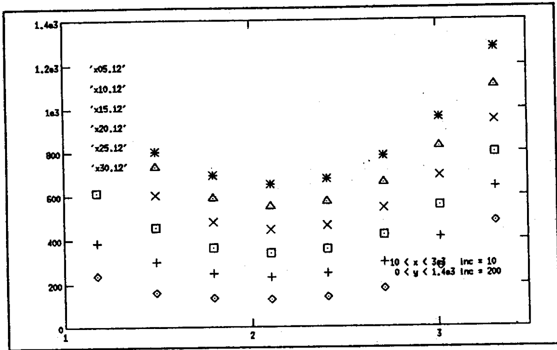
と考えてよい. よって J が十分大きい場合の計算量は $O(N \log N)$ となる. J が小さい場合, つまり N が q に比べてそれほど大きくない場合は $O(q \log q)$ と考えてよい. すなわち, 記憶容量に比べてあまり多くの cell を設けると, J が小さくなるかわりに q が大きくなるため, かえって多くの計算時間が必要となる.

図7および図8は, それぞれ 1 次元および 2 次元の場合の計算時間の実測値である. パラメータの設定条件は先の図6と同じである.

図9は, Kが 10, 14, 17, 20, 22, 24, 26, 28, 30, 31 の領域分割について, 記憶容量が 1023 のもとで、8192 個のデータのインサートを 4 回行ない, その結果のデータに対して 10 個の類似データ検索を 100 回行なったもの. グラフに示す時間は類似データ検索に要した計算コストである.

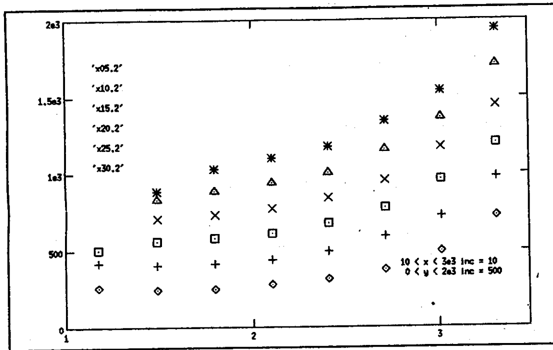
6 おわりに

ここでは, データが 1. 実数の場合, および 2. 実数の 2 次元ベクトルの場合について考えたが, より広い範囲の問題領域を扱



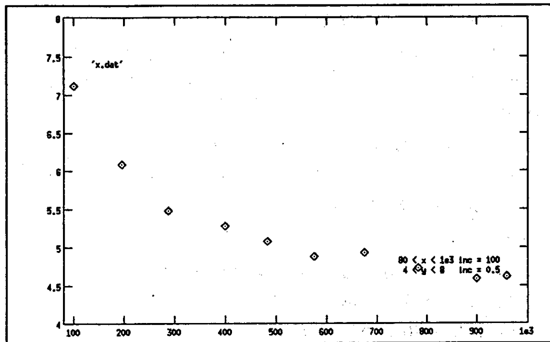
縦軸：時間 [sec]　　横軸：記憶容量 (log)
◇検索データ数 5 個　＋検索データ数 10 個
□検索データ数 15 個　×検索データ数 20 個
△検索データ数 25 個　＊検索データ数 30 個

図 7: 1 次元領域分割のよる類似データ検索の計算コスト.



縦軸：時間 [sec]　　横軸：記憶容量 (log)
◇検索データ数 5 個　＋検索データ数 10 個
□検索データ数 15 個　×検索データ数 20 個
△検索データ数 25 個　＊検索データ数 30 個

図 8: 2 次元領域分割による類似データ検索の計算コスト.



縦軸：時間 [sec]　横軸：cell の個数 (K^2)

図 9: 2 次元領域分割における分割密度の変化に伴う計算コストの変化.

おうとするならば，離散的な値を要素とする多次元ベクトルや，記号構造についても検討する必要があろう．記号については，文献データベースの情報検索の分野での成果が参考になるものと思われる．

また，ここで提案したアルゴリズムは暗記学習の実装のためだけではなく，OR など他のいくつかの分野にも応用が可能と思われる．

参考文献

[1] Carbonell, J. G. (1986). Derivational Analogy: A Theory of Reconstructive Problem Solving and Acquisition. In Michalski, R. S., Carbonell, J. G. & Mitchell, T. M. (Eds.) *Machine Learning: An artificial intelligence approach* (Vol. 2). Los Altos, CA: Morgan Kaufmann.

[2] DARPA (1988). Proceedings of a Workshop on Case-Based Reasoning.

[3] Stanfill, C. & Waltz, D. (1986). Toward Memory-based Reasoning. *Comm. of ACM*, Vol. 29, No. 12, pp. 1213–1228.

[4] 石畑 清 (1989). アルゴリズムとデータ構造. 岩波書店.

A 重み付けに基づく忘却のアルゴリズム

```
LastIndex ← SIZE;
Heap ← MakeVector(SIZE);
define function HeapInsert(HEAP, DATA)
Returns stored position and old-content if exists.
  local variable X;
  if (LastIndex > 0)
    { LastIndex ← LastIndex − 1;
      Heap[LastIndex] ← DATA;
      return (HeapCheckDown(LastIndex), NIL); }
  else if (Heap[0].weight < DATA.weight)
         { X ← Heap[0];
           Heap[0] ← DATA;
           return (HeapCheckDown(0), X); }
  else return (NIL, NIL);
end define
define function HeapCheckDown(INDEX)
Returns stored position.
  local variable L, X;
  L ← INDEX ×2 + 1;
  if (L < SIZE)
    { if (L+1 < SIZE & Heap[L+1].weight < Heap[L].weight)
        L ← L+1;
      if (Heap[L].weight < Heap[INDEX].weight)
        { Heap[L] ↔ Heap[INDEX];
          return HeapCheckDown(L); }
      else return INDEX; }
  else return INDEX;
end define
define function HeapCheckUp(INDEX)
  local variable integer:MOTHER;
  MOTHER ← ⌊(INDEX − 1)/2⌋;
  if (MOTHER ≥ LastIndex)
    and (Heap[INDEX].weight < Heap[MOTHER].weight)
    { Heap[INDEX] ↔ Heap[MOTHER];
      return HeapCheckUp(MOTHER); }
  else return INDEX;
end define
```

B データが実数の場合の類似データ検索アルゴリズム

```
Root ← NIL;
define structure node
  { weight, value, left, right, lower, upper }
end define
define function Insert(WEIGHT, DATA)
  local valiable node:NODE, integer:INDE, node:OLD;
  NODE ← MakeNode(weight = WEIGHT, value = DATA);
  (INDEX, OLD) ← HeapInsert(NODE);
  if (OLD ≠ NIL) Discard(OLD);
  if (INDEX ≠ NIL)
    Insert1(DATA, addressOf(Root), NIL, NIL, NODE);
  return INDEX; end define
define procedure Insert1(DATA, address:PLACE,
                         char:FLAG, node:NODE, NEW)
  local variable node:N;
  N ← valueOf(PLACE);
  if (N = NIL)
    { valueOf(PLACE) ← NEW;
      if (FLAG = "left")
        { NEW.lower ← NODE.lower;
          NEW.upper ← NODE;
          NODE.lower ← NEW;
          if (NEW.lower ≠ NIL) NEW.lower.upper ← NEW; }
      else if (FLAG = "right")
        { NEW.upper ← NODE.upper;
          NEW.lower ← NODE;
          NODE.upper ← NEW;
          if (NEW.upper ≠ NIL) NEW.upper.lower ← NEW; } }
  else if (DATA < N.value)
      Insert1(DATA, addressOf(N.left), "left", N, NEW);
  else Insert1(DATA, addressOf(N.right), "right", N, NEW);
end define
define procedure Discard(node:NODE)
  local variable node:RIGHT, UPPER;
  RIGHT ← NODE.right;
  UPPER ← NODE.upper;
  if (RIGHT ≠ NIL)
    { if (RIGHT ≠ UPPER)
        { Delete1(RIGHT, UPPER);
          NODE.right ← RIGHT; }
      UPPER.left ← NODE.left;
      Delete2(UPPER, NODE); }
  else Delete2(NODE.left, NODE);
  if (NODE.upper ≠ NIL) NODE.upper.lower ← NODE.lower;
  if (NODE.lower ≠ NIL) NODE.lower.upper ← NODE.upper;
end define
define procedure Delete1(node:ANC, node:NODE)
  if (ANC.left = NODE) ANC.left ← NODE.right;
  else Delete1(ANC.left, NODE);
end define
define procedure Delete2(node:NEW, node:OLD)
  if (OLD = Root) Root ← NEW;
  else Delete3(Root, NEW, OLD);
end define
define procedure Delete3(node:ANC, node:NEW, node:OLD)
  local variable node:NODE;
```

```
  if (OLD.value < ANC.value)
    { NODE ← ANC.left;
      if (NODE = OLD) ANC.left ← NEW;
      else Delete3(NODE, NEW, OLD); }
  else
    { NODE ← ANC.right;
      if (NODE = OLD) ANC.right ← NEW;
      else Delete3(NODE, NEW, OLD); }
end define

define procedure Search(real:DATA, function:FUNC);
  Function ← FUNC;
  if (Root ≠ NIL) Search2(DATA, Search1(DATA, Root));
end define

define function Search1(real:DATA, node:NODE)
  if (DATA < NODE.value)
    { if (NODE.left ≠ NIL) return Search1(DATA, NODE.left);
      else return (NODE.lower, NODE); }
  else if (NODE.right ≠ NIL) return Search1(DATA, NODE.right);
  else return (NODE, NODE.upper);
end define

define procedure Search2(real:DATA, node:LOWER, node:UPPER)
  if      (LOWER = NIL) Search3(UPPER, "upper");
  else if (UPPER = NIL) Search3(LOWER, "lower");
  else if ((DATA - LOWER.value) < (UPPER.value - DATA))
        { if (apply(Function, LOWER) = NIL)
            Search2(DATA, LOWER.lower, UPPER); }
  else if (apply(Function, UPPER) = NIL)
        Search2(DATA, LOWER, UPPER.upper);
end define

define procedure Search3(node:NODE, FLAG)
  if (NODE ≠ NIL) and (apply(Function, NODE) = NIL)
    { if (FLAG = "upper") Search3(NODE.upper, FLAG);
      else if (FLAG = "lower") Search3(NODE.lower, FLAG); }
end define
```

C　データが実数の２次元ベクトルの場合の類似データ検索アルゴリズム

```
Mesh ← 10;
Cells ← MakeArray(Mesh, Mesh);
Unit ← (D_min - D_max) / Mesh;
```

D_{min} is the minimum value of data, and
D_{max} is the maximum value of data.

```
define structure node
  { weight, value, distance }
end define

define function CellPosition(DATA)
  local variable integer:X;
  X ← ⌊ (DATA - D_min) / Unit × Mesh ⌋;
  return min(Mesh-1, max(0, X));
end define

define function Insert(WEIGHT, DATA)
  local variable node:NODE, integer:INDEX,
                 node:OLD, integer:X, Y;
  NODE ← MakeNode(weight = WEIGHT, value = DATA);
  (INDEX, OLD) ← HeapInsert(NODE);
  if (OLD ≠ NIL) Discard(OLD);
  if (INDEX ≠ NIL)
    { X ← CellPosition(DATA_1);
      Y ← CellPosition(DATA_2);
      Insert NODE into Cells[X, Y]; }
  return index;
end define

define procedure Discard(node:NODE)
  local variable integer:I, J;
  I ← CellPosition(NODE.value_1);
  J ← CellPosition(NODE.value_2);
  Remove NODE from Cells[I, J];
end define


define structure boundary
  { axis, index, distance, delta, limit }
end define

define procedure Search(DATA, FN)
  local variable integer:I, J, AXIS, DELTA, INDEX,
                 list:NODES, QUEUE, real:DISTANCE,
                 boundary:B;
  I ← CellPosition(DATA_1);
  J ← CellPosition(DATA_2);
  NODES ← GetNodes(DATA, I, J);
  QUEUE ← NIL;
  for all ⟨AXIS, DELTA, INDEX⟩
     in ⟨⟨1, -1, I⟩, ⟨1, 1, I⟩, ⟨2, -1, J⟩, ⟨2, 1, J⟩⟩:
     Insert CreateBoundary(DATA, AXIS, DELTA, INDEX)
       into QUEUE;
  QUEUE ← sort(QUEUE, λ(X,Y).(X.distance < Y.distance));
  DISTANCE ← first(QUEUE).distance;
  for all B in QUEUE:
     if (B.index = B.limit) Remove B from QUEUE;
LOOP:
  NODES ← Search1(DATA, FN, DISTANCE, NODES);
  if (NODES = .TRUE.) or (QUEUE = NIL) then return;
  B ← first(QUEUE);
  B.distance ← B.distance + Unit;
  B.index ← B.index + B.delta;
```

```
    if (B.index = B.limit) QUEUE ← rest(QUEUE);
    else if (rest(QUEUE) ≠ NIL)
          QUEUE ← nconc(rest(QUEUE), list(B));
    if (QUEUE = NIL) DISTANCE ← B.distance;
    else             DISTANCE ← first(QUEUE).distance;
    NODES ← nconc(NODES, GatherNodes(DATA, B, QUEUE));
    go to LOOP;
end define

define function Search1(DATA, FN, real:MAX, list:NODES)
    local variable node:N, list:L, REST;
    L ← NIL;
    REST ← NIL;
    for all N in NODES:
        { if (N.distance < MAX) L ← cons(N, L);
          else REST ← cons(N, REST); }
    for all N in sort(L, NearData):
        if (apply(FN, N) ≠ NIL) return .TRUE.;
    return REST;
end define

define predicate NearData(node:X, node:Y)
    X.distance < Y.distance;
end define

define function CreateBoundary(DATA, AXIS,
                               DELTA, INDEX)
    local variable integer:I, LIMIT;
    if (DELTA > 0) { I = INDEX + 1; LIMIT = Mesh − 1; }
    else           { I = INDEX;     LIMIT = 0; }
    return MakeBoundary(axis = AXIS, index = INDEX,
                        distance = |DATA_AXIS − Unit × I|,
                        delta = DELTA, limit = LIMIT);
end define

define function GetNodes(DATA, X, Y)
    local variable node:N, list:L;
    L ← copy(Cells[X, Y]);
    for all N in L: N.distance ← 𝒟(DATA, N.value);
    return L;
end define

define function GatherNodes(DATA, boundary:B, list:REST)
B is the nearest boundary, and REST is the list of rest boundaries.
    local variable integer:FROM, TO, I, boundary:X, list:L;
    FROM = 0; TO = Mesh − 1;
    for all X in REST:
        if (B.axis ≠ X.axis)
          { if (X.delta < 0) FROM ← X.index;
            else             TO ← X.index; }
    L ← NIL;
    if (B.axis = 1)
      for I = FROM to TO:
          L ← append(GetNodes(DATA, INDEX, I), L);
    else
      for I = FROM to TO:
          L ← append(GetNodes(DATA, I, INDEX), L);
    return L;
end define
```