計算機アーキテクチャ 75－3
（１９８９．３．２３）

前向きプロダクションシステムのための要求駆動型マッチングアルゴリズム

藤田 聡　介弘達哉　山下雅史　阿江 忠
（広島大学工学部）

プロダクションシステム(以下PS)は、断片的知識を自然に表現できるなどの点で注目を集めている推論メカニズムの一つである。しかしながらPSの処理時間は知識ベースの大規模化に伴って著しく増大する傾向にあり、その解決が強く望まれている。また、従来から行なわれているPSの高速化に関する研究の多くでは、戦略と独立なデータ駆動型マッチング方式を前提としているが、戦略を方式の中で陽に考慮することによってその実行は一層高速化されるはずである。

本稿では戦略を考慮した高速なPS実行方式の提案を行なう。具体的には戦略の持つ方向性に着目し、逐次実行を前提とした要求駆動型のマッチングアルゴリズムを提案する。前向きPSに対して戦略を考慮した要求駆動方式を導入することにより、推論に必要なマッチングのみの実行が可能となる。なお提案するアルゴリズムは、従来方式を特にマッチングの試行回数の点で上回っており、その効果は、PSプログラムを用いたシミュレーションにより実験的に確認される。

A DEMAND-DRIVEN MATCHING ALGORITHM FOR FORWARD REASONING PRODUCTION SYSTEMS

Satoshi FUJITA, Tatsuya SUKEHIRO, Masafumi YAMASHITA and Tadashi AE

Faculty of Engineering, Hiroshima University
Saijo-cho, Higashi-Hiroshima-shi, 724 Japan

Production systems (in short, PS's) are one of powerful tools to represent declarative knowledge for problem solving. The execution, however, has much redundancy due to the data-driven search mechanism, which makes it seriously slow as the size of knowledge-base grows.

This report proposes a demand-driven matching algorithm for forward reasoning PS's. Strategies of PS programs generally have a sort of directionality for inference. The proposed algorithm is designed to achieve a fast and efficient inference on PS's by directly considering such strategies in the level of the matching mechanism. Hence, it can avoid redundant invokation of matching processes which never contribute to the inference.

The effect of the proposed algorithm for speedup of PS's will be estimated by simulation using actual PS programs.

1. まえがき

プロダクションシステム(以下PS)は、断片的知識を自然に表現できるなどの点で注目を集めている推論メカニズムの一つである。しかしながらPSの処理時間は、その実行が検索を基本とすることなどから知識ベースの大規模化に伴って著しく増大する傾向にある。このことはPSの実用上非常に大きな問題点となっている。

この問題を解決すべく、従来よりマッチングアルゴリズムの改良[1-3]や並列実行方式の検討[4-8]など、PS高速化のための様々な面からのアプローチがなされてきた。これらの基本となっているのは通常、Rete[1]に代表される推論中の状態(具体的には競合集合)を保持する型のマッチングアルゴリズムであり、これらはいわば推論の一部であるマッチングの高速化に注目したものである(すなわち高速化の工夫は戦略と独立である)。見方を変えるとこれらのアプローチは、制御の流れを規定する戦略と断片的な知識の集合である知識ベースとを分離することに意義を持つ知識表現モデルとしてのPSの流れを、強く汲んでいるものと考えられる。

ところがPSにおける推論の本質はデータベースの初期状態から最終状態までの遷移経路(path)であり、極論すれば、経路上にない分岐枝はその推論過程においてはそもそも考慮する必要のなかったものである。したがってこのような分岐枝をマッチング時に極力考慮しないようにすることで、処理量はかなり軽減されるはずである。この事実は、戦略を考慮したマッチングメカニズムが、PS上の推論を高速化する上で非常に有効な手段となり得ることを示唆している。

本稿では戦略を考慮した高速なPS実行方式の提案を行なう。具体的には、通常のデータ駆動的な高速実行法が前向き推論においてしばしば冗長な振舞いをすることから、要求駆動型のマッチング方式を提案する。このように前向きPSに対して戦略を考慮した要求駆動(後向き連鎖)の概念を新たに導入することで、推論に本質的に必要なマッチングのみを効率的に行なうことが可能となる。提案する方式の効果は、筆者らが試作したプロトタイプPS上でのベンチマークにより実験的に確認される。

以下2.では従来の研究の概説を行なう。3.では今回試作したプロトタイプPSの概要について示す。4.ではこのシステム上での要求駆動型マッチングのPS高速化に関する効果に関する検討を行なう。最後に5.では今後の課題等について言及する。

2. 従来の研究

2.1. 基本実行モデル

PSのモデルとして現在最も一般的なものは次のようなモデルである。(以下これをOPS型モデルと呼ぶ。) このモデルは基本構成要素としてルールベース，データベースおよび推論エンジンを持ち、以下の三つのフェーズからなるサイクルの繰り返しによって逐次的に計算(すなわち前向き推論)を行なう[1]。ここでルールベース中の各ルールは条件部と行動部から成っており、それぞれ『現在のデータベースにおいてそのルールの条件部が満たされるならば、そのルールの行動部の実行によってデータベースの更新を行なってもよい』ことを表している。また各ルールの条件部は複数の条件要素の連接であり、すべての条件要素が満たされたとき(すなわちその条件要素に適合するデータがデータベース中に存在し、それらが条件要素間の束縛変数制約を満足するとき)そのルールの条件部が満たされていると呼ぶ。

[OPS型モデルにおける計算]((1)〜(3)を繰り返す)

(1) データベースとルールベースを参照することで、条件部の満たされているルールを全て検出する。(マッチングフェーズ)

(2) 条件部の満たされているルール集合の中から、戦略に従って適当なものを一つ選択する。選択すべきルールがなければ停止する。(競合解消フェーズ)

(3) 選択されたルールの行動部を実行しデータベースの更新を行なう。(実行フェーズ) □

ルールの表現法としてはいくつかの方式が考えられるが、通常は実用性を考慮して変数の使用を許している。この場合マッチングフェーズで生成されるのは、有限サイズのデータベースによって具現化(instantiate)された有限サイズのルールインスタンス集合である。以下この集合を競合集合(conflict set)と呼び、ルールインスタンスを単にインスタンスなどと呼ぶ。また推論エンジンに用意されている競合解消戦略は、推論実行中固定されている。その選択基

準としては通常、条件要素数などのルールの構造的な複雑さやインスタンスの条件部を構成する各データの生成時刻などが多く用いられる[9]。具体的なPSプログラムの例を図1に示す。この例(例1と呼ぶ)では、ガールフレンドとデートをする際のお金と時間に関する条件が、ルール形式で記述してある。

```
// rule ( a part of rulebase ) //
(defrule '(rule1
  (girl ^name=$s ^age=$t)
  (condition ^time=$u ^money=$v)
  (course
   ^name=$w ^time(= $x<=$u) ^money(= $y<=$v))
-->
  (modify 2 '
    (condition ^time(- $z $w) ^money(- $v $y))))

// data ( a part of database ) //
(girl ^name Seiko    ^age 22)
(girl ^name Akina    ^age 21)
(girl ^name Marina   ^age 20)
(course ^name drive      ^time 3   ^money 3000)
(course ^name shopping   ^time 4   ^money 1000)
(course ^name movie      ^time 2   ^money 2500)
(condition ^time 4   ^money 3000)
```

図1　PSプログラムの一部 (例1)

図中で、条件要素の先頭(例えば、condition)は条件要素名であり、記号^の後はそれぞれ条件要素のフィールド名(例えばtime)とその値(例えば$u)を表している。また記号$に続くフィールド値は変数である。

2.2. 状態保持型マッチングアルゴリズム

Forgyによって提案されたReteアルゴリズム[1]は、OPS型モデルのマッチングフェーズを効率的に実行するためのデータ駆動型マッチングアルゴリズムである。その高速化に関する工夫の本質は、

(a) データベースのクラスタリングによって、データベースアクセスを局所化すること

(b) 競合集合の保存により、データベースの変化分に対するマッチングのみで次サイクルでの競合集合が得られること

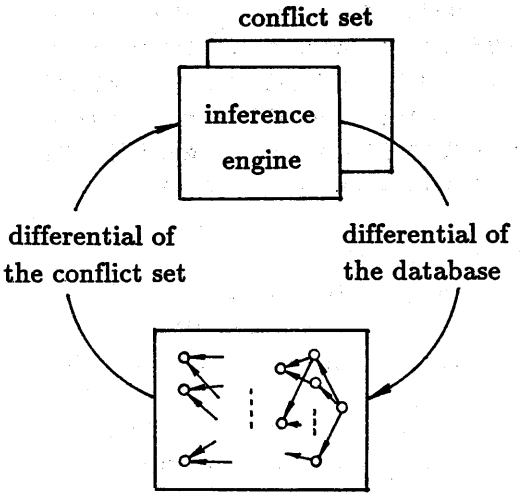


図2　状態保持型マッチング

(c) 部分的にマッチングに成功した不完全なインスタンス情報の保存によるマッチング重複の回避

の三点に集約される。状態保持型アルゴリズムのイメージを図2に示す。ここで(b)の工夫は、例えば例1のプログラムの実行のある時点で新しいデータ(girl ^name Yoko ^age 21)がデータベースに追加されたとしても、それ以前に存在する他の3人に関するルールインスタンスが保持される(条件部が満たされ続ける)ということに対応している。

Reteの後で提案されたいくつかの改良型アルゴリズム[2,3]においても、上記(a)〜(c)の各点はほぼ踏襲されている。またデータベースサイズが小さくその更新が頻繁に行なわれるプログラムでは競合集合を保持しない方が実行がより高速であるとの報告もあるが[11]、そのような場合にはRete適用のいかんにかかわらず十分な処理速度は得られるはずである。すなわち一般に競合集合全体を保持しておく状態保持型マッチングアルゴリズムは、条件部の満たされた全てのインスタンスを検出する処理に関しては有効な方式であるといえるであろう。

しかしながら多くの場合、必ずしも全てのインスタンスを生成しなくても競合解消フェーズで選択されるべきインスタンスを検出することが可能である。また通常の(全探索的でない)PSプログラムでは、せ

っかく生成されたインスタンスが別のインスタンスの実行に伴って、実行されることなく消去されてしまうことが多い。例えば例1において、できるだけ若い子とのなるべく安い費用でのデートを選択するような戦略を仮定してみる。この戦略を用いた場合、例1の状況で選択されるインスタンスは

```
(defrule '(rule1
  (girl ^name Marina ^age 20)
  (condition ^time 4 ^money 3000)
  (course ^name shopping ^time 4 ^money 1000)
-->
  (modify 2 '
      (condition ^time 0 ^money 2000)))
```

であり(これをインスタンス1と呼ぶ)、このインスタンスはある種のソーティングを併用することで(他のインスタンスを生成することなく)生成可能である。またインスタンス1を実行することによって、例1では実行可能であった全てのインスタンスの条件部は満たされなくなる。すなわちこの戦略に従った実行においては、インスタンス1以外のインスタンスを(少なくともこの時点では)生成する必要がない。

本稿では以上の考えに基づき、戦略を考慮した効率的な要求駆動型マッチングアルゴリズムを提案する。次節ではある具体的な戦略を一つ固定した後、その戦略に対応したマッチングアルゴリズムを示す。ここでのアイデアは一般の戦略に対しても拡張することができる。

3. 試作システム

3.1. 競合解消戦略

以下では簡単のため、まず特定の戦略を固定してアルゴリズムの説明を行なう。その後で、このアルゴリズムの一般化に関する議論を行なっていく。ここで対象とする戦略は、図3のような手続きで表現される。この手続きでは競合集合Sを入力として用いているが、現実にはSを具体的に構成する必要はない。なおここでは、S中の第i番目($1 \leq i \leq |S|$)のインスタンスをr[i]で表している。

図3で関数t(i,j)の値は、インスタンスr[i]の『第j番目の条件要素に対応するデータ』に固有のタイムタグを表している[1]。データのタイムタグはそ

```
// CONFLICT RESOLUTION STRATEGY //

procedure CRS
input     : set of instances S={ r[i] }.
output    : index of the selected instance.

begin
    S':=S;              // initialization //

    // filter 1 --- status of instances //
    for all  i≤|S|
        if ( f(i)=1 )  S':=S'-{ r[i] }  fi
    rof;

    // filter 2 --- order of data //
    j :=1;       // left most condition element //
    while ( (|S'|>1) & (j≤MAX²) ) do
        for all  i   s.t., r[i] ∈ S'
            t :=max t(k,j),  where  r[k] ∈ S';
            if ( t(i,j)<t )  S':=S'-{ r[i] }  fi;
        rof;
        j :=j+1
    od;

    // filter 3 --- order of rule //
    for all  i   s.t., r[i] ∈ S'
        n :=min n(k),  where  r[k] ∈ S';
        if ( n(i)>n )  S':=S'-{ r[i] } fi
    rof;
    output i   s.t., r[i] ∈ S'
end
```

図3　競合解消戦略の例

のデータの生成された順番を表しており、データの生成された順に1から昇順に各データに対して付記される(関数tは手続き中のフィルタ2で使用)。またルールには固有(distinct)の優先順位が静的に付けられているものとする。すなわちルール集合は全順序集合であり、具体的にはこの関係は、条件要素数やルールの記述順序などによって決定される。フィルタ3で用いられる関数n(i)の値は、インスタンスr[i]に対応するルールの優先順位を表している(ここでは値が少ないほど優先度が高いとする)。さらにこの戦略では同一インスタンスの繰返し実行を避ける

[1] ここでは便宜上タイムタグによってデータ間の全順序関係を付けているが、戦略の決定性さえ保証されるならば、任意のフィールド値によるデータ間の順序付けも可能である。

[2] *MAX*は、各ルールの条件要素数の最大値を表す。またここでは、t(i,j)に対応する条件要素が存在しないときその値は0であるとする。

ため、各インスタンスがすでに実行されたものであるかどうかの判定を行なっている。フィルタ1で用いられているf(i)は、インスタンスr[i]がすでに実行されているときは1、まだ実行されてないときは0を返す関数である。

この戦略は決定的(deterministic)な振舞いをする。なお図3の戦略は、若干の差異はあるものの、通常よく用いられるMEAやLEX[10]などの戦略と同様の働きをする。

3.2. データ構造

図4に本アルゴリズムで用いられる基本データ構造を示す(図は一つのルールに関して描かれている)。このような探索木上で部分的に評価の完了した『ルールとデータ組の組』(これを部分マッチとよぶ) および実行済み(すなわち評価済み)インスタンスを記憶しておくことにより[3]、図3のフィルタ1の判定を容易に実現することができる。ここで各部分マッチは根ノードからのパス(path)に対応している。

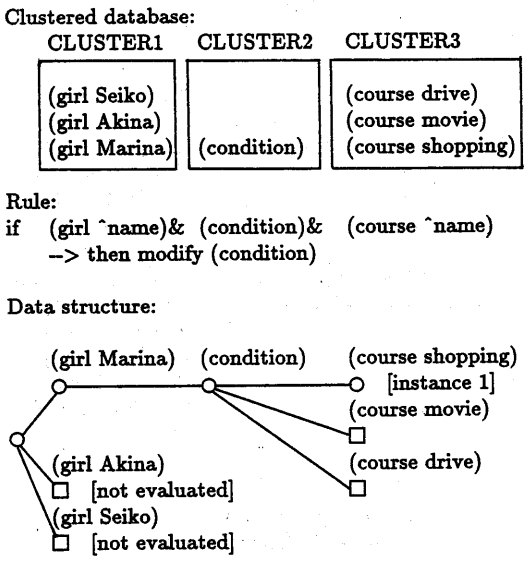


図4　要求駆動型マッチングの基本データ構造

探索木上の各ノードは一つのデータに対応しており、それぞれの葉ノードには、そのノードに対応する部分マッチが未評価であるか否かを示すフラグが用意されている。図4では、未評価ノードが□で、評価済みノードが○でそれぞれ示されている。また木の深さは各々そのルールの条件要素に対応しているが、Reteと同様、各条件要素に関するデータの候補はデータの型や条件要素中の定数記述によってあらかじめ絞り込まれているものとする[1]。この候補集合を以下ではクラスタと呼ぶ。また各条件要素に関するクラスタは、データ間の動的な順序関係に関する戦略の要請に従って、評価に先立ってソートしておくことができる[4]。

また本アルゴリズムでは、その時刻における『各ルールの生成しうる最も選択される可能性の高いデータ組』を記録しておくためのテーブルが、別に用意されている。その使用法等は以下で述べる。

3.3. マッチングアルゴリズム

次にアルゴリズムの動作について説明する。アルゴリズムはルールの実行に伴うデータベースの更新(フェーズ1)とデータベース更新後の最大インスタンスの検出(フェーズ2)の2フェーズから成っており、これはOPS型モデルにおける認知-実行サイクルに一致する。

[フェーズ1](データベースの更新)

ルールの実行に引き続いて、以下の一連の処理が行なわれる。

(1) 変更データに関係するクラスタの検出。

(2) 検出された全クラスタの内容更新、およびクラスタ内データのソート。

(3) 更新されたクラスタに対応する全ての探索木の変更(図5参照)。ここでデータを新たに付加した場合は、付加ノードに未評価であることを示すフラグを付記しておく。 □

以上の一連の処理は、Reteアルゴリズムにおけるルールの実行からαメモリの更新までの処理と、一部を除いてほぼ一致する。ここで(2)のソーティングは、

[3] この木を完全に展開したものがReteネットワーク上の全情報である。また各ルールに対応する木の併合(merge)は、Reteの場合と同様可能である。

[4] したがってクラスタのサイズが十分大きい場合には、ヒープ(heap)[12]によるクラスタの実現が有効である。

一般にはすでにソートしてあるクラスタの適当な位置への変更データの挿入である(ただし前述の戦略では、単なるスタック操作のみでよい)。また(3)の処理はそれまで探索木上で展開された部分についてのみ行なえばよいので、Reteなどに比べるとその分処理量は軽減されることになる。

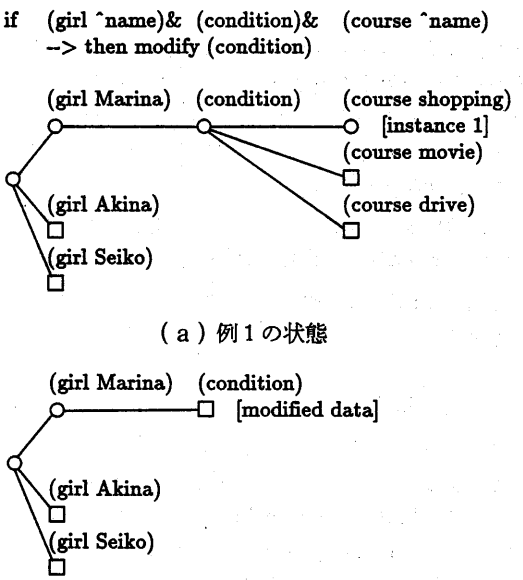
[フェーズ2](インスタンスの検出)

戦略によって選択すべきインスタンスの検出は、先に導入されたテーブルを用いて行なわれる。具体的には、以下の二つの処理がインスタンスが一つ得られるまで繰り返される。

(1) 各ルールについてその時点でインスタンスとして選択される可能性の最も高いデータの組を一つ検出し、そのデータ組をテーブルに登録する。

(2) テーブル中でさらに最も可能性の高い組に注目し、その組合せの束縛変数に関する評価を行なう。評価の成功したところまでの履歴は先のデータ構造中に記憶される。そのデータ組でインスタンスを生成することに失敗すれば、ステップ(1)に戻る。なおこの評価は、そのデータ組中の未評価フラグの立っているデータの位置から開始される。未評価フラグの立っているデータの評価を行なったときは、評価と同時にそのノードのフラグを取り下げる。評価に成功すれば、そこからさらに深さ1ほど木を展開する(全てのデータについて)。展開された各ノードには未評価フラグを立てておく(図5)。□

以上の動作を例1について行なった様子を図5に示す。同様の処理をReteのようなデータ駆動的なアルゴリズムで実行したときの振舞いを比較のため図6に示す。この例からもわかるように、データの変更が多く起こるようなプログラムの実行では、一般に要求駆動型アルゴリズムの方がデータ駆動型に比べ効率の点で有利であると考えられる。またこの方式を前述の戦略以外の一般の戦略に対して適用するには、図4のデータ構造上である種のヒューリスティック関数による可能性の予測を行ないながら条件部の評価を進めていくようにすればよい。この処理では、分枝限定法[13]が効果的に利用できる。ただしこの方式の効果は、想定する戦略がどのような方向性を持っているかに強く依存すると考えられる。



(a) 例1の状態

(b) インスタンス1の実行後

図5 例1の実行(要求駆動型マッチング)

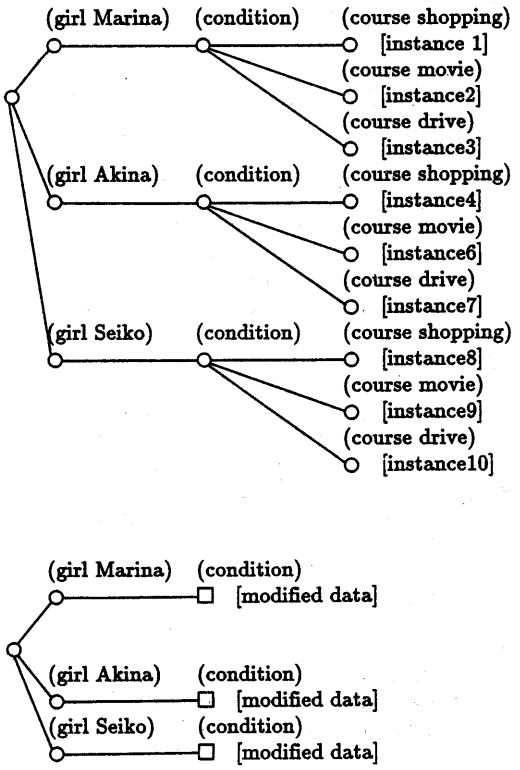


図6 図5の処理のデータ駆動型による実行

4. 評価

提案する要求駆動型アルゴリズムのマッチング回数軽減に関する効果を評価するため、プロトタイプPSの試作を行なった。システムは KCL/SUN[14] (Kyoto Common Lisp)で記述されており、ソースプログラムサイズは約３０Kbyteである。また現在は試作段階であるため、システム上には図３の戦略のみがインプリメントされている。評価はいくつかのベンチマークプログラムの実行における、 Reteアルゴリズム[1]とのマッチング処理回数の比較により行なった。

表１に評価結果を示す。Reteに対するマッチング回数比(α)はいずれの場合も向上しており、要求駆動メカニズムの効果が顕著に現われていることがわかる。ただし問題(A)では、ルール記述そのものがデータ駆動型マッチングのメカニズムを意識した(無駄な動作の少ない)ものであるため、マッチング回数比(α)はそれほど向上していない。

また表には、推論中新たに実行可能となったインスタンス数と実際に実行されたインスタンス数の比(β)が付記されている。表から、要求駆動の効果(α)が比(β)に依存していることがわかる。すなわち一般に(β)が１に漸近するような(たとえば全探索的な)実行では、要求駆動の効果は薄くなる傾向にあると思われる。しかしながら本アルゴリズムでは戦略で用いられるデータ間の順序関係を考慮することでマッチングの失敗回数もさらに極小化できるため、したがって例えばルール中の条件要素数が多く束縛変数の評価に失敗する可能性が高い場合には、表１に示したマッチング回数比(α)はさらに向上するはずである。また本アルゴリズムでは実際に照合を行なうまでにある程度のオーバヘッドを伴うことが予測されたが、今回の実験に関してはこれはほとんど問題とはならなかった。これは今回採用した戦略が、最近変更されたデータに関連するルールを優先的に選択する深さ優先探索(depth-first search)を基本としており、さらにそのことが、実行メカニズムに対してうまく反映されていたためであると考えられる。

最後に、提案するアルゴリズムの最悪計算量を把握するため、任意の戦略を採用した場合のオーバヘッドを含めた手数の最悪値を算出した。結果を以下に示す。以下ｒはルール数、ｎは各ルールに関する未評価ノード数を示す。簡単のためｎはすべてのルールで常に一定とし、クラスタの併合は行なわないとする。またｍは各クラスタ内のデータ数(すべてのクラスタで常に一定と仮定する)を表している。なお探索木とテーブルは、いずれもヒープ[12]によって実現されているものとする。

本アルゴリズムにおけるマッチング回数は、最悪の場合Reteのそれと一致し、その手数は各基本操作

表１　要求駆動型マッチングアルゴリズムの効果

	照合回数			向上比[倍]	
	クラスタへの分類	クラスタ間のチェック		(α) (2)/(1)	(β)
		要求駆動型(1)	Rete(2)		
問題(A) (ルール数２０)	１００	４１	６３	1.5	1.7
問題(B) (ルール数４)	４１	５５	１１２	2.0	1.8
問題(C) (ルール数２)	４４	３１	１２２	3.9	1.9

問題(A) 猿とバナナの問題　　(クラスタ数２０)
問題(B) 宣教師と狼の問題　　(クラスタ数９)
問題(C) ハノイの塔［ディスク３］(クラスタ数４)

に対してそれぞれオーバヘッド分だけ増加する。したがってReteアルゴリズムの、フェーズ1のステップ(2)までに対応する部分に要する手数をc1、それ以降に対応する部分に要する手数をc2とすると、本アルゴリズムの実行に要する総手数は、最悪で

$$O(c1 \times \log m + c2 \times (\log n + \log r))$$

となる[5]。また一般にルールあたりの条件要素数はnによらず一定であるから、$\log n = O(\log m)$と見なすことができる。よって総手数の最悪値のオーダは、$O((c1+c2) \times \log m + c2 \times \log r))$となり、これはほぼReteの$O(\log m)$倍となっている。しかしながら通常は、各基本処理に要するオーバヘッドは最悪値よりもはるかに少ないと思われる。また多くの場合採用される戦略に応じた処理の効率化が可能であることから、提案するアルゴリズムは一般に、オーバヘッドを考慮してもなお有効であると考えられる。

5. むすび

本稿では、実行メカニズムレベルで戦略を陽に考慮することによってプロダクションシステムの処理時間が大幅に短縮可能であることを示した。具体的には戦略の持つ方向性に着目し、戦略を用いた要求駆動型マッチングアルゴリズムの提案と評価を行なった。

プロダクションシステムの計算モデルとしての理論的な枠組みは現時点では非常に曖昧であり、その基本モデルはあらゆる戦略を容認し得るものである。したがって本稿の手法が現存の全てのプロダクションシステムに対して効果的に適用できるとは限らない。(例えば選択基準が動的な環境に依存する戦略に対しては、要求駆動方式の適用は困難である。)しかしながら本稿の結果は、採用される戦略の性質を生かしたプロダクションシステムの高速化が可能であることを主張しており、このことはルールベースシステムの実行を今後さらに高速化していくための一つの有用な示唆であると思われる。

今後本稿のアプローチをより一般化するため、プロダクションシステムの計算モデルとしての枠組みについても検討していく予定である。

[5] 前半ではクラスタ(サイズm)の変更、後半ではテーブル(サイズr)と未評価ノード集合(サイズn)の変更と最小値検索がそれぞれ行なわれるため。

[参考文献]

[1] Forgy, C.L.,: Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem, *Artif. Intell.*,Vol.19, pp.17-37 (1982).

[2] Schor, M.I., et al.: Advances in Rete Pattern Matching, *Proc. AAAI-86*,pp.226-232 (1986).

[3] 荒屋他: プロダクションシステムにおける効率的パタン照合のための連想Reteネットワーク表現, 情処学論, Vol.29, No.8, pp.741-748 (1988).

[4] Stolfo, S.J.,: Five Parallel Algorithms for Production System Execution on the DADO Machine, *Proc. AAAI-84*,pp.300-307 (1984).

[5] Gupta, A.: *Parallelism in Production Systems*, Pitman Publishing (1987).

[6] Miyazaki, J., et al.,: A Shared Memory Architecture for MANJI Production System Machine, *Database Machines and Knowledge Base Machines* (ed. M.Kitsuregawa and H.Tanaka), pp.517-531, Kluwer Academic Publishers (1988).

[7] Gupta, A., et al.,: Results Parallel Implementation of OPS5 on the Encore Multiprocessor, CMU-CS-87-146 (1987).

[8] Stolfo, S.J. and Miranker, D.P.: The DADO Production System Machine, *JPDC*,3, pp.269-296 (1986).

[9] McDermott, J. and Forgy, C.L.: Production System Conflict Resolution Strategies, *Pattern-Directed Inference Systems*, pp.177-199, Academic Press (1978).

[10] Forgy, C.L.: OPS5 User's Manual, CMU-CS-81-135 (1981).

[11] Nuutila, E., et al.: XC - A Language for Embedded Rule Based Systems, *ACM SIGPLAN NOTICES*, Vol.22, No.9, pp.23-32 (Sept. 1987).

[12] Tarjan, R.E.: Data Structures and Network Algorithms, *Society for Industrial and Applied Mathematics* (1983).

[13] 茨木俊秀: 組合せ最適化〜分枝限定法を中心として, 産業図書 (1983).

[14] Yuasa, T. and Hagiya, M.: Kyoto Common Lisp Report, *Research Institute for Mathematical Sciences*, Kyoto University (1985).