バックトラッキング機構を持った プロダクション・システム言語

6J-4

広瀬　紳一
日本アイ・ビー・エム株式会社　東京基礎研究所

1．はじめに

前向きプロダクション・システムでは、各時点で実行可能なルールの中から、競合解消によってひとつのルールを選び、それを適用する、という方法で実行が進んでいく。したがって、同程度に見込みのありそうなルールが複数個あっても、それらの各々について適用を行った結果を調べることはできない。ある時点での選択が望ましいものであるかどうかは、実行が何ステップか先に進んでからでないと判断できないことも多いが、そのような場合には、一般に、細かい場合分けと、それに対応した複雑な条件部を持ったルールが多数必要になり、プロダクション・システムによる記述が非常に困難になる。

このような問題点を解決するための手法として、ＡＴＭＳ［1］が提案されている。ＡＴＭＳを利用した場合の利点として、実行中のある時点でいくつかの選択肢が存在する場合に、各々の選択をおこなった結果を別々の『世界』として管理することができ、さらに、それらを比較して最も良いものを選べることがあげられる。しかし、各世界で共通な情報の共有が可能であるとはいえ、処理系はすべての世界の情報を保持する必要があり、実行に必要とされる記憶領域が増大するという問題がある。

これに対して、単純なバックトラッキングによって、選択肢を順次試行していくという方法をとった場合には、複数の世界を比較することはできないが、処理系は、ひとつの世界の情報だけを持っていればよく、効率的な実行が可能になる。本報告では、このような考え方に基づき、プロダクション・システム言語ＰＯＰＳ２［2］へのバックトラッキング機構の組み込みの方法について検討し、また、実現された機構を用いたプログラミング例を紹介する。

2．バックトラッキング機構

ＰＯＰＳ２では、ルールの条件部と作業記憶とのマッチングを調べるのに、ＲＥＴＥアルゴリズム［3］を用いており、実行サイクルごとに適用可能な全てのルールのインスタンシエーションが競合集合として得られる。バックトラッキング機構とは、複数のインスタンシエーションが存在するある時点において、ひとつのルールを選択して推論を進めていったが、それが後にあまり好ましくなかったと判明した時に、実行をこの時点までさかのぼって、その他のルールを試してみることを可能にする仕組みのことである。この機構を利用することによって、ある時点で同程度にもっともらしい複数の選択肢から、無理にひとつだけを選びだす必要がなくなる。

ＰＯＰＳ２のルールの動作部は、Prolog ゴールであり、その実行には、成功、失敗の二通りが考えられる。従来のＰＯＰＳ２では、動作部が失敗した場合には、エラーとみなして、推論が停止するという規約であった。今回は、このような場合にプロダクション・システム側でバックトラッキングを行うようにした。このことによって、Prolog プログラムの宣言的な面を、より生かした知識表現が可能になることが期待される。

3．実現方法の検討

ＲＥＴＥアルゴリズムでは、作業記憶の変化は、ＲＥＴＥネットワークにトークンとして与えられ、ネットワークの各ノードの持つ、部分マッチングの情報を更新する。トークンには、事実が加わったことを示す正タイプのものと、削除されたことを示す負タイプのものがある。これらのトークンによるネットワークの内部状態の変化は可逆的である。すなわち、ある正トークンによって生じた変化は、同じ事実に対する負トークンによって完全に元に戻すことができるし、逆も同様である。したがって、ネットワークの状態を元に戻すには、作業記憶の変化の履歴としてネットワークに流したトークンの情報を記録しておき、バックトラックする時に、逆のトークンを流してやればよい。

一方、競合集合に対しては、別の手続きが必要となる。なぜならば、ＰＯＰＳ２も含めて、通常のプロダクション・システムの実行においては、一度発火したルールは、競合集合から取り除かれるという、競合解消の規約があるために、競合集合は、厳密な意味での、作業記憶とルールの条件部とのマッチングの結果にはなっていないからである。そのため、競合集合については、別途そのコピーを保持するという処理をおこなう必要がある。このとき、すべてのルールについて、その実行ごとに競合集合のコピーをとるのは処理系の負担が大きいため、あとで戻ってきたい時点を、何らかの方法で指定するようにした方が良いと考えられる。

以上のことを考慮するとバックトラッキングの実現の手法の一つとして、次のようなものが考えられる。

- 実行のある時点でいくつかの選択が可能であり、一意に決定できない時には、とりあえず選ばれたルールの実行時に競合集合のコピーを作り、また、それ以降の作業記憶の変化の履歴をとるようにする。これは、ルールの動作部で、create_world述語を実行することによっておこなわれる。
- ルールの実行部が失敗した場合には、直前のcreate_worldが実行された時点まで実行をさかのぼる。これは、作業記憶の変化とそれに伴うネットワークの状態の変化を履歴情報を用いて打ち消すことによって実現できる。ただし、競合集合については、保存しておいたコピーとさしかえる。

この方法においては、処理系が保持すべき情報は、ある時点での競合集合のコピーと、それ以降におこった作業記憶の変化のみであるため、必要とされる記憶領域は、比較的わずかですむ。しかし、作業記憶の変化については、バックトラックする時に、通常の実行と同様にネットワークにトークンを流して、ルールの条件部とのマッチングをおこなう必要がある。特に、ＰＯＰＳ２の条件部での Prolog 呼び出しの結果については、コピーを保存したときと同じであるとは限らないので、ネットワーク内の部分マッチングの情報と、競合集合内の情報が一貫しない結果になるおそれがある。

この問題に対処するため、次のような方法も考えられる。すなわち、従来の正、負のトークンにくわえて、ある事実を含む部分マッチングの情報を非活性にするトークンと、非活性の情

A Production System Language with Backtracking Mechanism

Shin-ichi HIROSE
IBM Research, Tokyo Research Laboratory

報を再び活性化するためのトークンを導入するという方法である。この方法をとった場合には、再度のパターン・マッチングは、不要となるので、競合集合や、 Prolog 呼び出しも含めて、完全に状態を復元できるという利点がある。しかし、一般には、大量の非活性の情報をＲＥＴＥネットワーク内に保持するため、必要とされる記憶領域が増大する。

今回の処理系の試作では、実現の簡単さと、記憶領域の問題を考慮して、前者の方法を採用している。

４．プログラミング例

『農夫のジレンマ』をバックトラッキング機構を利用して解くためのルールを図１に示す。ここでは、問題の状態は、ひとつの事実で表現されており、初期状態では、農夫たちはすべて左岸にいる。ルール cross0 は、農夫が単独で、また cross1 から cross3 までは、それぞれ、農夫が何かと一緒に対岸へ渡るという試行をおこなうものである。これらのルールは、他の選択の可能性を残すために、実行部の先頭で create_world 述語を実行する。また、ルール check1 は、問題の持つ拘束条件であり、山羊が、他のものと一緒にいるときには、農夫もそこにいなければならないことを表現している。そして、この拘束条件が満足されないと実行部は失敗し、バックトラッキングがおきて、別の可能性の探索がおこなわれることになる。ルール check2 は、探索がループするのを防ぐためのものである。

図2は、4-Queen の問題を解くルールを示している。ここで、ルール put0 は、最初の女王を置くためのものであり、put は、それ以降の女王のためのものである。ルール check は、二つの女王が互いを取る位置にないという拘束条件を表現している。このルールのセットは、66回の発火の後に、解のひとつを見つけるが、ここで、ルール put の優先度を、abs(P-Q) にすると、はじめの回を見つけるまでの発火の回数を57回に減らすことができる。これは、直前の女王から、なるべく離れた位置に次の女王を置いた方が良いというヒューリスティクスを導入した最良優先探索にあたる。

５．おわりに

プロダクション・システム言語ＰＯＰＳ２へのバックトラッキング機構を導入について検討し、また、そのうちの一手法を用いて、実際にＰＯＰＳ２に組み込んだシステムを試作した。前節で示したものは、極めて単純な例であったが、一般に、複数の世界を比較しなくても、各々の試行の結果の良否が判定できるような問題も多いと考えられる。そのような場合には、今回試作したバックトラッキングの機構は有用であると思われる。

今後の課題としては、（非）活性化のトークンの利用の検討も含めた、より効率的で適用範囲の広いバックトラッキング実現手法の開発があげられる。また、失敗の原因が、はるか以前の選択にある場合に、単純に直前に選択をおこなった時点に戻るのではなく、実際に問題を引き起こした選択をおこなった時点まで一気に実行をさかのぼるための機能のサポートも必要であろう。

参考文献

[1] de Kleer,J., An Assumption-based TMS, Artificial Intelligence, Vol.28, pp.127-162, 1986.

[2] 広瀬, プロダクション・システム記述言語ＰＯＰＳ２, 情報処理学会研究報告, 86-PL-8, 1986.

[3] Forgy,C.L., RETE: A Fast Algorithm for the Many Pattern/Many Object Pattern Matching Problem, Artificial Intelligence, Vol.19, pp.17-37, 1982.


```
literalize state(t,farmer,fox,goat,cabbage).

fact state(t=0,farmer=left,fox=left,goat=left,cabbage=left).

.revside(left,right).
.revside(right,left).

rule cross0 is
    S=state(t=T,farmer=X)
 =>
    create_world() &
    revside(X,Y) &
    T1 := T + 1 &
    cmake(S, *, {t := T1, farmer:=Y}).

rule cross1 is
    S=state(t=T,farmer=X,fox=X)
 =>
    create_world() &
    revside(X,Y) &
    T1 := T + 1 &
    cmake(S, *, {t := T1, farmer:=Y, fox:=Y}).

rule cross2 is
    S=state(t=T,farmer=X,goat=X)
 =>
    create_world() &
    revside(X,Y) &
    T1 := T + 1 &
    cmake(S, *, {t := T1, farmer:=Y, goat:=Y}).

rule cross3 is
    S=state(t=T,farmer=X,cabbage=X)
 =>
    create_world() &
    revside(X,Y) &
    T1 := T + 1 &
    cmake(S, *, {t := T1, farmer:=Y, cabbage:=Y}).

rule ok:1000 is
    state(farmer=right,fox=right,goat=right,cabbage=right)
 =>
    halt().

rule check1:1000 is
    state(farmer=X,fox=Y,goat=Z,cabbage=W)
 =>
    Z=Y -> X=Z &
    Z=W -> X=Z .

rule check2:1000 is
    state(t=T1,farmer=X,fox=Y,goat=Z,cabbage=W) &
    state(t=T2:(T2=/T1),farmer=X,fox=Y,goat=Z,cabbage=W)
 =>
    fail().
```

図１．『農夫のジレンマ』のルール

```
fact queen(1,0).
fact queen(2,0).
fact queen(3,0).
fact queen(4,0).

fact candidate(1).
fact candidate(2).
fact candidate(3).
fact candidate(4).

rule put0: P is
    QU=queen(1,0) &
       candidate(P)
  =>
       create_world() &
       modify(QU, {2 := P}).

rule put: P is
    QU=queen(N,0) &
       queen(M:(M := N - 1), Q:(Q =/ 0)) &
       candidate(P:(P =/ Q))
  =>
       create_world() &
       modify(QU, {2 := P}).

rule check:100 is
    queen(N,P&~0) &
    queen(M:(M>N),Q&~0)
  =>
    P <> Q &
    N+P <> M+Q &
    N+Q <> M+P
```

図２．『 4-Queen 』のルール