(1992. 5. 26)

ルールの適用順序を考慮した効率的な帰納推論手法

毛利隆夫　田中英彦
東京大学　工学部

従来の帰納推論の結果得られるルールは，正の事例だけを説明し，負の事例を許容しないものであった．ところが，帰納推論の段階で適用順序を考慮してルールを生成することにより，負の事例を説明するようなルールを使用することができる．すなわち，一部のルールは負の事例を説明しても，ルール全体としては負の事例を許容しないようなルール群を構成することができる．本研究では，負の事例を含むようなルールを導出する帰納推論手法 WINE (foil WIth NEgative rules) を提案する．WINE により得られるルールは，適用順序があらかじめ固定されているが，その分推論時間は短くて済み，ルールも簡潔なものが得られる場合がある．

AN EFFICIENT INDUCTIVE REASONING METHOD WITH FIX-ORDERED RULES

Takao Mohri　Hidehiko Tanaka
Faculty of Engineering, the University of Tokyo

The rules generated by usual inductive reasoning methods explains only positive cases, and never explains negative ones. Meanwhile, the evaluation order of rules is taken in thought in reasoning, we can generate rule groups, which has some rules that permit negative cases, but don't permit negative ones as a whole. In this research, we propose an inductive reasoning method called WINE (foil WIth NEgative rules). WINE is based on FOIL and generates fix-ordered rules. WINE can produce simple rules in short reasoning time.

1　はじめに

正／負の事例および背景知識から，それらを説明するようなルール表現を導き出す帰納推論に関しては，現在までに様々な研究が行なわれてきている ([Qui90],[Qui91]). 従来の帰納推論の結果得られるルールは，正の事例だけを説明し，負の事例を許容しないものであった．このようなルールは，どのような順序で評価しても構わない．

ところが，帰納推論の段階で適用順序を考慮してルールを生成することにより，負の事例を説明するような'rough' なルールを使用することができる．すなわち，一部のルールは負の事例を許容しても，ルール全体としては負の事例を説明しないようなルール群を構成することができる．

本研究では，負の事例を含むようなルールを導出する帰納推論手法 WINE (foil WIth NEgative rules) を提案する．WINE により得られるルールは，適用順序があらかじめ固定されているが，その分，推論時間は短くて済み，ルールも簡潔なものが得られる場合がある．

ただし，WINE によって導かれたルールは，健全性と完全性に関して従来の方法で導かれたルールとは異なる振舞いを示すため，利用法が限定されることも同時に述べる．

2　ルールの適用戦略と簡単化

帰納推論の結果として出力されるルールは,「いくつか（なるべく多く）の正の事例を説明し，負の事例は全く含まない」ことが要請される．これは,「正の事例を説明し，負の事例を説明しないようなルールの作成」という，帰納推論のそもそもの出発点である．

しかし，この条件は少し厳し過ぎるとも言える．というのは，ルールの適用順序を考慮しつつ，失敗する述語 fail を用いたり,「対応する事例」を用いたりすれば，個々のルールとしては負の事例を説明してしまうかも知れないが，ルール全体としては，どの負事例も説明しないようにすることが可能だからである．以下では，これらの方法によってルール全体を簡単にする方法について説明する．

2.1　fail 述語によるルールの簡単化

まず，fail 述語を用いたルール全体の簡単化について，概念的に説明しよう．図 1のような事例空間を考える．○印は正事例，×印は負事例，長方形は帰納推論によって生成されるルールが解を返す範囲を表している．

図 1のような事例の分布が与えられた場合，fail 述語が使えないとすると，上側の図のように小さな領域しか覆えないようなルールを多数用いて正事例全体を覆う必要がある．

ところが，fail 述語が利用でき，かつルールの評価に優先順位をつけられるのならば，下側の図のようにすることができる．まずルール R1 を評価して，負事例は負事例として取り除いておいて，あとから 'rough' なルール R2 によって正事例を説明することができる．このように，fail 述語によってルール数の少ない，簡潔なルールを作成することができる．

ここでいう'rough' とは，ルール R2 単独では正事例だけでなく負事例も説明してしまうという意味である．ルール R2 は，従来の帰納推論では許されていないルールであるが，ルールの適用順序を限定することによって利用可能になった．

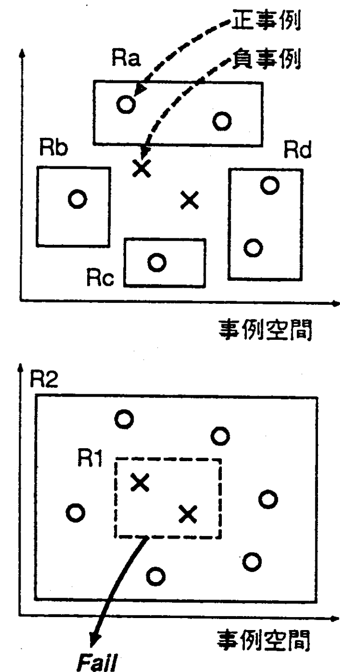

図 1: fail 述語によるルールの簡単化

2.2　対応する事例によるルールの簡単化

また，fail 述語を使わなくても,「対応する事例」があれば，ルールの適用順序を考慮することによって,

同様の簡単化が行なえる．まず次のように，帰納推論で得られるルールの lhs の述語の引数を Input 部分と Output 部分に分けて，Input 部分の変数に束縛された値を入力とし，Output 部分の変数に値を束縛して返すような質問の形式を考える．つまり，lhs の述語が $p(+In_1,\ldots,+In_N,-Out_1,\ldots,-Out_M)$ のような形をしていて，In 変数に値を束縛して，Out 変数を未束縛にしておき，ルールを用いて Out 変数の値を決定する質問を考える．この時，$(In_1,\ldots,In_N,Out_1,\ldots,Out_M)$ が事例に相当する．

このような質問の場合には,「対応する事例」があると，fail を用いる場合よりもさらに簡単なルール表現に置き替えることができる．ここでいう「対応する事例」とは，事例のうち Input 部分は同じ値を，Output 部分では異なる値を持ち，事例の正／負が異なる事例同士であると定義する．

まず，従来の方法では，図 2のように，それぞれの正事例を細かく覆う必要がある．つぎに，上で述べた fail を用いる方法では，rough なルール R2 が使えるようになるので，より簡潔なルール表現が生成できる．さらに，R1 の前に必ず R2 が評価されることを考えると，R1 は不必要で R2,R3 だけで十分になり，より簡潔なルール表現になる．

以下では，この対応する事例によるルールの簡単化について扱う．

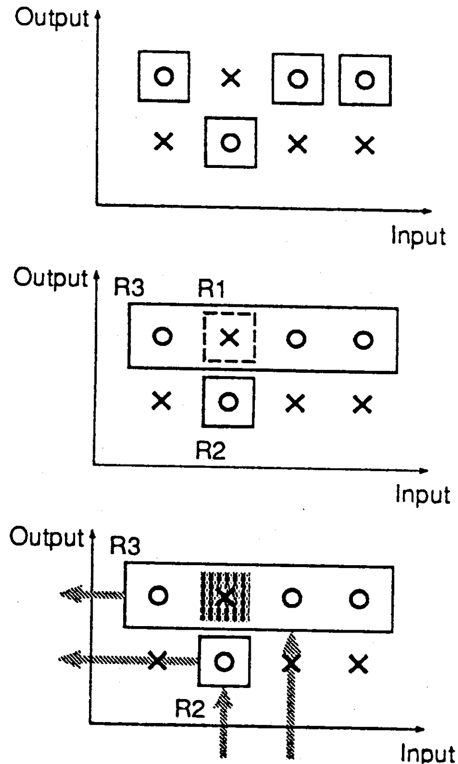

図 2: 対応する事例によるルールの簡単化

2.3 Penguin の例題でのルールの簡単化

上で述べた対応する事例によるルールの簡単化について，図 3のような,「鳥は飛ぶ，ペンギンは飛ばない」という，非単調推論でよく用いられる例で考えてみよう．

すべてのルールが，負の例を含まないことが要請された場合，帰納推論を用いると，次のようなルールが得られるだろう．

$$
\begin{aligned}
fly(X,Y) &:- \; pigeon(X), yes(Y).\\
fly(X,Y) &:- \; swallow(X), yes(Y).\\
fly(X,Y) &:- \; eagle(X), yes(Y).\\
fly(X,Y) &:- \; penguin(X), no(Y).
\end{aligned}
$$

しかし，ルールを Prolog のように，上から順に適用することにすれば，次のような簡潔な表現でも，解答になっている．

$$
\begin{aligned}
fly(X,Y) &:- \; penguin(X), no(Y). \quad \cdots \quad (1)\\
fly(X,Y) &:- \; yes(Y). \quad \cdots \quad (2)
\end{aligned}
$$

［P enguinの例題］の問題定義

正事例

fly(pigeon-1,yes).
fly(pigeon-2,yes).
fly(swallow-1,yes).
fly(swallow-2,yes).
fly(eagle-1,yes).
fly(eagle-2,yes).
fly(penguin-1,no).
fly(penguin-2,no).

負事例

fly(pigeon-1,no).
fly(pigeon-2,no).
fly(swallow-1,no).
fly(swallow-2,no).
fly(eagle-1,no).
fly(eagle-2,no).
fly(penguin-1,yes).
fly(penguin-2,yes).

背景知識

pigeon(pigeon-1).
pigeon(pigeon-2).
swallow(swallow-1).
swallow(swallow-2).
eagle(eagle-1).
eagle(eagle-2).
penguin(penguin-1).
penguin(penguin-2).
yes(yes).
no(no).

図 3: 非単調推論で良く用いられる例題

ここで，(2) のルールは，負の事例 (fly(penguin-1,yes), fly(penguin-2,yes)) を満たしてしまっている．しかし，ルールの適用順序を「上から順番に適用する」方法だけに限定すれば，このルール群からは，fly(penguin-1,yes) という答が返ってくることはない．それは，例えば fly(penguin-1,X) (X は変数) という問い合わせがなされた場合，必ず (1) のルールによって fly(penguin-1,no) という解が得られ，(2) のルールが評価されることがないからである．

3 WINE のアルゴリズム

WINE(foil WIthNEgative rules) は，上の「対応する事例」によるルールの簡単化の考え方に基づき，負の事例を含むルールを導出する帰納推論手法である．

WINE では，基本的には FOIL[Qui90, Qui91] のアルゴリズムを用いている．FOIL は有名な帰納推論プログラムの一つで，情報量に基づくヒューリスティクスを用いて，仮説空間をトップダウンに探索してルールを求めていく．FOIL では負の事例を許容しなくなるまで節に述語を追加していく．一方，WINE では，負の事例が事例集合の一定割合を下回れば，その段階で仮説の選択を打ち切り，そこまでのルールを答として一旦出力する．その後で，その負事例に対応する正事例を導くためのルールを作成して，先に評価するような順番にそれを並べるのである．

FOIL のアルゴリズムを図 4 に，WINE のアルゴリズムを図 5 に示す．WINE では，負事例を許容してしまったルールと，それを補うために対応する正事例を説明するルールとが，ひとつのまとまりをなしている．以下ではこれを「ルール群」と呼んでいる．WINE では，ルール群単位でルールの導出を行なっていく．main 関数のループにおいて FOIL と異なるのは，FOIL では「ルールの作成」となっているところが，WINE では「ルール群の作成」になっている点である．

関数「ルール群の作成」では，一旦生成したルールが負事例を許容した場合には，それに対応する正事例がすべて説明し尽くされるまで，自分自身を再帰的に呼び出している．

関数「仮説がルールとしての条件を満たしている」の中の定数 α が，含んでいい負事例の割合の限界を定めている．α が 1 に近付けば，ほとんど負事例を許容しなくなり，WINE は FOIL のアルゴリズムと同じになる．逆に α が 0 に近付けば，簡単に負事例を許容することになり，WINE アルゴリズムが起動されやすくなるが，許容してしまう負事例の数が多いルールは，かえってそれを補うためのルールが多数必要になり，好ましくない．なお，定数 α は 0.7 と定めているが，この値は経験的に決めたものである．

```
main(){
    事例集合を作成する.
    while (事例集合に正事例が残されている){
        ルールの作成 (事例集合).
        そのルール群で説明される正事例を,
            事例集合から取り除く.
    }
}

ルールの作成 (事例集合){
    while (not(仮説がルールの条件を満たしている)){
        ルールに新しいリテラルを加える.
        事例集合をつくりなおす.
    }
}

仮説がルールとしての条件を満たしている (){
    if(事例集合に, 負事例が含まれていない){
        return TRUE;
    } else {
        return FALSE;
    }
}
```

図 4: FOIL のアルゴリズム

4 WINE の動作

では，先ほどの penguin の例題を用いて，このアルゴリズムを詳しく見ていくことにしよう．

まず，fly(X,Y) という形のルールを求めるために帰納推論プログラムが起動される．最初に，次のような事例集合が作成される (表 1)．

次に，この事例集合を満たすようなルール群が作成される．候補となるリテラルは次の 7 種類作成され，それぞれについて，その仮説によって満たされる正事例／負事例，Gain が計算される．(尚，表 2 中の事例数の欄は，「そのリテラルで説明される事例の数／そこまでに残っている事例の数」を意味する)．

ここで，Gain が最も大きいリテラル yes(Y) が選択される．この時点での仮説は，

$$fly(X,Y) :- yes(Y).$$

となる．

```
main(){
    事例集合を作成する.
    while (事例集合に正事例が残されている){
        ルール群の作成 (事例集合).
        そのルール群で説明される正事例を,
            事例集合から取り除く.
    }
}
ルール群の作成 (事例集合){
    while (not(仮説がルールの条件を満たしている)){
        ルールに新しいリテラルを加える.
        事例集合をつくりなおす.
    }
    if ((ルールは負事例を許容しない) &&
        (以前許容した負事例に対応する正事例は
            すべて説明した)){
        return <ルール >;
    }else {
        新しい事例集合=次の事例集合の作成 (事例集合).
        ルール群=ルール群の作成 (新しい事例集合).
        return <ルール群, ルール >;
    }
}
仮説がルールとしての条件を満たしている (){
    if( (事例集合中, 正事例の占める割合が
            定数α以上である) &&
        (含まれている負事例のなかに,
            以前のルールで含まれた正事例に
            対応する事例が含まれていない)){
        return TRUE;
    } else {
        return FALSE;
    }
}
```

図 5: WINE のアルゴリズム

表 1: 事例集合 (その 1)

正事例:(X,Y)={
(pigeon-1,yes),(pigeon-2,yes),
(swallow-1,yes),(swallow-2,yes),
(eagle-1,yes),(eagle-2,yes),
(penguin-1,no),(penguin-2,no) }
負事例:(X,Y)={
(pigeon-1,no),(pigeon-2,no),
(swallow-1,no),(swallow-2,no),
(eagle-1,no),(eagle-2,no),
(penguin-1,yes),(penguin-2,yes) }

表 2: 正／負事例数, Gain 値

リテラル	正事例数	負事例数	Gain
pigeon(X),	2 / 8	2 / 8	0
swallow(X),	2 / 8	2 / 8	0
eagle(X),	2 / 8	2 / 8	0
penguin(X),	2 / 8	2 / 8	0
yes(Y),	6 / 8	2 / 8	3.51
no(Y),	2 / 8	6 / 8	-2.0

その次に, この仮説がルールとしての条件を満たしているかどうかを調べる. ルールとしての条件は次の 2 種類があった.

1. 事例集合中, 正事例の占める割合が定数 α 以上である.
2. 含まれている負事例のなかに, 以前のルールで含まれた正事例に対応するものが含まれていない.

ここで, 定数 α は 0.7 であるとしている. この例では, 事例集合 (正事例=6 個, 負事例=2 個) での負事例の割合は, $6/(6+2) = 0.75 > 0.7$ であるので, この条件 1 は満たされている. また, 条件 2 は, 許容した負の事例を補完するためのルールを導く際に必要な部分であるので, 今回は関係ない. よって条件がすべて満たされるので, 仮説 $fly(X,Y) : -yes(Y)$がルールとして認められる.

しかし, このルールは負の事例を許容しているので, 負の事例に対応する正の事例を導くようなルールを導いて, このルールより先に適用するようにしておく必要がある.

次の段階のために, 表 3のように新たに事例集合が作り直され, ルール作成が続行される

新しい事例集合の作り方を, まず正事例に関して述べる. 仮説によって説明された正事例は, 事例集合から取り除かれる. 残った正事例のうち「以前のルールで許容した負事例に対応する正事例」には, ＊印がつけられる. 今回の例ではすべての正事例に＊印がつけられているが, 必ずそうなるというわけではない. ＊印の正事例を満たすルールは, 負事例を許容したルールよりも前に位置していなくてはならない. また, ＊印のついた正事例がなくなれば, ルール群の作成が終了する.

表 3: 事例集合 (その 2)

正事例:(X,Y)={
(penguin-1,no),(penguin-2,no) }
負事例:(X,Y)={
#(pigeon-1,no),#(pigeon-2,no),
#(swallow-1,no),#(swallow-2,no),
#(eagle-1,no),#(eagle-2,no),
(penguin-1,yes),(penguin-2,yes) }

次に負事例の事例集合は，すべての負事例が前回の事例集合から引き継がれる．但し，一部の負事例は特別扱いされる．#印のついた事例は,「以前のルールで説明した正事例に対する負事例」である．これらの負事例を許容するルールは，作成してはならない．というのは，対応する正事例を説明するルールが，そのルールよりも後に評価されてしまうので，負事例の方が先に解答されてしまうからである．

さて，このように作成された事例集合に対して，同様に仮説の探索が行なわれる．今回の正／負事例数と Gain 値を表4に示す．

表 4: 正／負事例数，Gain 値 (その 2 の 1)

リテラル	正事例数	負事例数	Gain
pigeon(X),	0 / 8	2 / 8	0
swallow(X),	0 / 8	2 / 8	0
eagle(X),	0 / 8	2 / 8	0
penguin(X),	2 / 8	2 / 8	0
yes(Y),	0 / 8	0 / 8	0
no(Y),	2 / 8	6 / 8	-2.0

Gain が 0 の候補が 5 つあるが，そのなかで正事例数が 0 でない penguin(X) が選択される．(実際には，他の 4 つの候補は，正事例数が 0 であることが分かった時点で，候補から除外されるので，penguin(X) しか候補に残らない)．

この時点での仮説は

$$fly(X,Y) :- penguin(X),$$

である．

つぎに 2 つめのリテラルを同様に探索する (表 5)．

表 5: 正／負事例数，Gain 値 (その 2 の 2)

リテラル	正事例数	負事例数	Gain
pigeon(X),	0 / 2	2 / 2	0
swallow(X),	0 / 2	2 / 2	0
eagle(X),	0 / 2	2 / 2	0
penguin(X),	2 / 2	2 / 2	0
yes(Y),	0 / 2	0 / 2	0
no(Y),	2 / 2	0 / 2	1.1

ここでは，最も Gain 値の大きい no(Y) が選択される．この時点での仮説：

$$fly(X,Y) :- penguin(X), no(Y).$$

は終了条件を満たしているので，ルールの作成はここで終了する．また，ここで＊印のついている正事例はすべて説明されたため，ルール群の作成も同時に終了する．

そして，ここで最初の事例集合の中の，すべての正事例が説明されたので，以上で帰納推論全体が終了する．結果として得られたルールは，得られたルールを逆順に並べたものであるので，

$$fly(X,Y) :- penguin(X), no(Y).$$
$$fly(X,Y) :- yes(Y).$$

となる．

5 WINE と FOIL の定性的な比較

ここでは，WINE と FOIL とを定性的に比較してみる．比較の時のパラメータになるのは，次のような項目である．

- Query の方法
  生成されたルールに対して質問を行なう時には 2 通りの方法がある．

  Query of True/False 質問の時の述語の変数にはすべて値が束縛されていて，真偽の結果を返す．

  Query of Value 質問の時の述語には未束縛な変数があり，真偽の結果が返るとともに，変数に値が束縛される．

- 健全性・完全性
  ここでいう健全性とは，ルール全体としてみたときに，Query of True/False の場合には，

1. 正事例と同じ質問をされたら，必ず真を返す．(True as True)
2. 負事例と同じ質問をされたら，必ず偽を返す．(False as False)

Query of Value の場合には，

1. 真になったときに返す事例には，負事例は含まれない．(True as True)
2. 偽になったときに返す事例には，正事例は含まれない．(False as False)

の両方が満たされることを意味する．一方，完全性とは，バックトラックなどを用いて繰り返し問い合わせることによって，与えたすべての正事例を，答として返すことができる性質を意味するものとする．

5.1 健全性・完全性の比較

WINE と FOIL の健全性・完全性についてまとめると表6, 7のようになる．

表 6: FOIL と WINE の健全性

	Query of T/F		Query of Value	
	T as T	F as F	T as T	F as F
FOIL	OK	OK	OK	OK
WINE	OK	*no*	OK	OK

表 7: FOIL と WINE の完全性

	Ouery of T/F	Query of Value
FOIL	—	OK
WINE	—	OK*

表6の *no* は，WINE で Query of T/F を負事例に関して行なった場合，負事例を許容しているルールによって，それが正事例であるとする誤った答が返される可能性があることを示している．これは，対応する事例を返すように作られたルールが，負事例の質問を素通りさせてしまい，rough なルールが評価されて，真だという答が返ってしまう可能性があるからである．

また，WINE で Query of Value を行なった場合には，学習に使ったすべての正事例が返されることは保証されるので，その意味で完全性は成り立つといえる．しかし，その答の中には負事例として与えたものも混じってしまう可能性がある (表 7の*)．もちろん，最初に得られる解が負事例でないことは保証されている．

また，WINE では，負事例を許容するルールを出力するので，Query of Value を行なった時，そのままでは，1 回目の解が負事例でないことは保証されるものの，2 回目以降の解が負事例でないことは保証できなくなりそうに思える．しかし，このことは，Prolog で用いられてる カット・オペレータによって，回避することができる．

つまり，ルール群単位で新たに別の述語名を用意し，そのルール群に属するルールは lhs にその述語名を使用し，最後にカット・オペレータを付けておく．そうすると，その述語名によって解がひとつ得られた場合には，カットによって二度とそのルール群が使用されなくなり，負事例が解として答えられることはなくなるため，健全性が得られるのである．ただし，この場合は，多くの正事例を説明するルールが，カット・オペレータによって評価されないことがあり得るために，同じ質問で多数の解を得ようとすると，うまくいかないかもしれない．

以上のような性質を見た時に，WINE は Query of Value 型の質問で，小数の解しか必要としていない場合に有効であることが予測される．

5.2 述語の否定を用いる方法との比較

さて，負の例を許容するようなルールを作成することにより，簡潔なルールを導くことができることを示したが，同様なことは，述語の否定を用いることでも実現できる．先ほどの例では，つぎのようなルールを導けばいいのである．(ここでは，not() で述語の否定を意味することとする)．

$$fly(X,Y) \ :- \ not(penguin(X)), yes(Y).$$
$$fly(X,Y) \ :- \ penguin(X), no(Y).$$

このように述語の否定を用いる方法も，簡潔なルールを推論するための，有効な方法の一つである．ただし，簡潔さはWINE の方が勝っている．WINE では負の事例を許容してもいいために，否定された述語を省略できる場合があるからである．例えばこの例では，$not(penguin(X))$ の分だけ，WINE の導き出すルールの方が簡潔である．

また，負の例を導き出すのは，'not-predicate' という

predicate があるものと仮定して，帰納推論を進めればいいが，すべての predicate に関して，not-predicate という述語を作成してしまうと，特に not つきの述語が determinate litaral として認識された場合，仮説空間の探索の幅が広くなって，計算コストが大きくなってしまうおそれがある．WINE では，述語の否定を用いないので，計算コストが増大するおそれはない．

さらに，述語の否定は，必ずしも論理的な否定を意味しない ([中島 83], [LS86]). 例えば Prolog では，

$$p(1).\ \ p(2).\ \ q(1).$$

というファクトが宣言されている状態で，

$$?-p(x), not(q(X)).$$

という質問をすれば，$X = 2$ が解答として得られるのに対して，

$$?-not(q(X)), p(X).$$

という質問の場合には，$not(q(X))$ で失敗してしまう．つまり，未束縛な変数を含む述語を否定してしまうと，論理学の否定と異なる解釈が行なわれる．したがって，否定された述語を含んだルールを導出する際には，否定される述語に，未束縛の変数が含まれないように，注意する必要がある．

6 実験結果

6.1 リスト処理問題での FOIL と WINE の比較

ここでは，帰納推論の例題として良く用いられているリスト処理問題を例にとって，FOIL と WINE の性能比較を行なう．ただし，例題 insert と penguin は，今回の実験のために作成したものである．quick sort, member, combination は [Qui90] で使われているものをそのまま用いた．

推論時間，ルールの数，リテラル数を表8に示す．表中の全リテラル数は，導いた各ルールのリテラル数 (LHS を含む) の合計を表している．また,「WINEed Rule」は，WINE により導かれたルールを示している.「なし」と書かれている場合は，WINE が起動されず，完全に FOIL アルゴリズムとして動作した場合である.「3 → 2」と書かれている場合は，まず負の事例を含むルールとして 3 番目のルールが作成されて，その後で，許容してしまった負の事例に対応する正の事例を説明するルールとして，2 番目のルールが作成されたことを示している．

これらの例からは，まず，WINE は起動されない場合，つまり，負の事例を許容するようなルールが作成されない場合が数多くあることがわかる．WINE が起動されない場合には，得られるルールは，全く FOIL と同じになってしまうが，オーバーヘッドのために，推論時間は WINE の方が少しだけ長くなっている．

また，penguin の例題に関しては，not を用いた場合との比較も行なってみた (表 9). この例では，FOIL では not を用いることによって推論時間，ルール数，総リテラル数ともに良い結果が得られている．しかし，WINE の場合は，not のリテラルを考慮する分，推論時間が余計にかかっているだけで，かえって逆効果になっていることがわかる．

6.2 障害物回避ゲームでの FOIL と WINE の比較

もう一つの例題として，図 6のような「障害物回避ゲーム」を用いた．このゲームは盤面の右下隅にいるプレーヤーが，左上隅にいるターゲットのところまで，途中の障害物を避けながら，なるべく少ない回数で移動するという，簡単な迷路探索型のゲームである．

今回は,「ある盤面 Board で，ある位置 Position にいる時に，次に進むべき方向 Direction」を事例として用意した．目標への最短経路である方向を正事例，それ以外の方向を負事例だとした (この事例を最適事例と呼ぶことにする). 7,8,9 の場合の正事例／負事例の様子を示す．ある場所から白い菱形で示されている方向が正事例，黒の菱形が負事例を表している．

この最適事例を用いて，FOIL アルゴリズムと WINE アルゴリズムとの比較を行なった．比較は，帰納推論に要した時間，得られたルールの数，そのルールに用いられていたリテラルの総数に関して行なった (表 10). この表からわかるように，WINE の方が，FOIL よりも，推論時間では 2 分の 1 程度，ルール数，リテラル数でも良い結果が得られている．したがって，障害物回避ゲームに対しては，WINE は FOIL よりも有効な手法だといえる．

7 結論および今後の課題

本研究では，FOIL を改良した帰納推論手法 WINE を提案した．WINE で導かれるルールは，適用順序が固定化されている反面，一部負の事例を許容することが許されており，そのため効率の良いルール表現ができる場合がある．実験によって，良い場合には 2 倍程度の推論速

表 8: リスト処理問題の学習結果

例題	推論時間 (秒) F/W [a]	ルール数 F/W	総リテラル数 F/W	WINEed Rule
quick sort	93 / 95	2 / 2	10/ 10	なし
member	0.8 / 0.85	2 / 2	6 / 6	なし
insert	48 / 49	3 / 3	15 / 13	3 → 2
combination	316 / 317	2 / 2	10 / 10	なし
penguin	0.56 / 0.27	4 / 2	12 / 5	2 → 1

[a]'F/W' は'FOIL での値/WINE での値' を表す

表 9: ペンギンの例題での not を用いた場合との比較

	推論時間 (秒) F/W [a]	ルール数 F/W	総リテラル数 F/W	WINEed Rule
not を用いる	0.36 / 0.34	2 / 2	6 / 5	2 → 1
not を用いない	0.56 / 0.27	4 / 2	12 / 5	2 → 1

[a]'F/W' は'FOIL での値/WINE での値' を表す

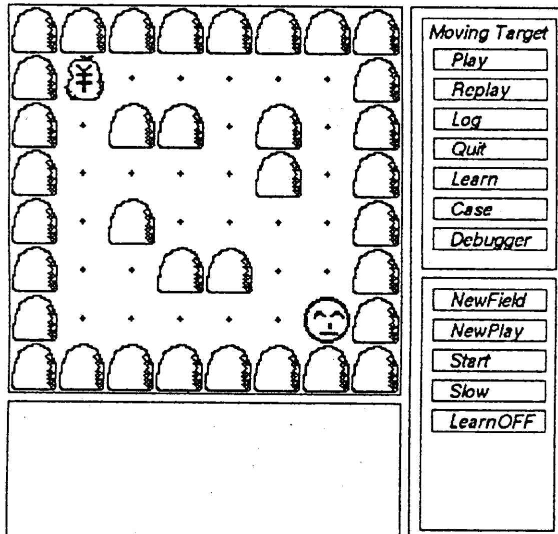

図 6: 障害物回避ゲーム

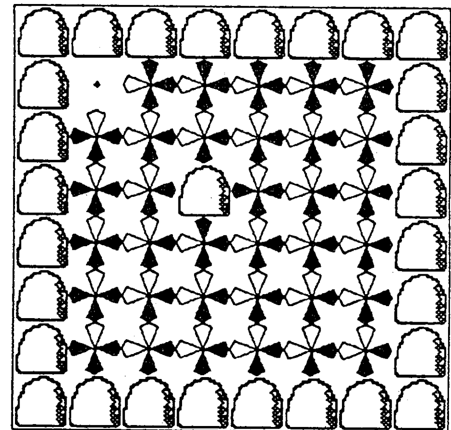

図 7: 盤面 easy と最適事例

表 10: 障害物回避ゲームでの学習結果

盤面	事例数 正 , 負	推論時間 (秒) F/W [a]	ルール数 F/W	総リテラル数 F/W	WINEed
easy	56 , 80	31 / 18	2 / 2	8 / 6	2 → 1
depth-1	50 , 78	481 / 216	7 / 6	57 / 42	6 →…→ 1
depth-2	44 , 76	11020 / 5790	13 / 10	158 / 127	7 →…→ 1
比率の平均		2.0 : 1	1.2 : 1	1.3: 1	

[a]'F/W' は'FOIL での値/WINE での値' を表す

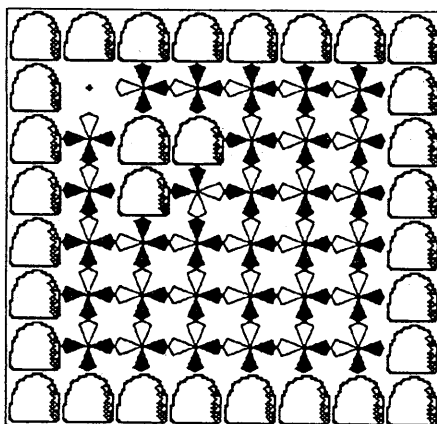

図 8: 盤面 depth-1 と最適事例

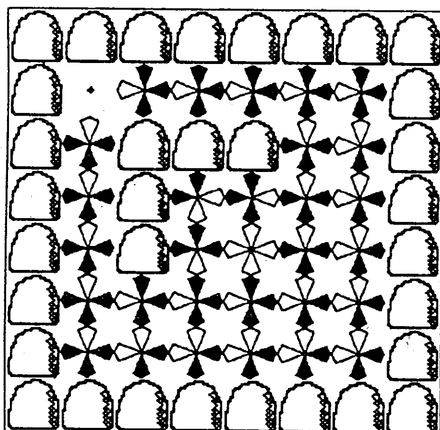

図 9: 盤面 depth-2 と最適事例

度の向上，ルール数，リテラル数の減少が達成できることが示された．しかし，WINE の特徴が活かされる例題は多くなく，健全性，完全性や質問の方法に関して，従来の方法にいくつかの制約が加わる．したがって，今後はこれらの制約に見合うような WINE の用途を探す必要があるだろう．

謝辞

東京大学電子工学科知能工学寄付講座の小野諭氏からは，研究全般に渡り多大な御指導を頂きました．また東京工業大学工学部情報工学科の沼尾正行氏からは，カット・オペレータの使い方に関するコメントを頂きました．この場をお借りしてお礼を申し上げます．

参考文献

[LS86] Ehud Shapiro Leon Sterling. *The Art of Prolog*. The MIT Press, 1986.

[Qui90] J.R. Quinlan. Learning logical definitions from relations. *Machine Learning*, Vol. 5, pp. 239–266, 1990.

[Qui91] J.R. Quinlan. Determinate literals in inductive logic programming. In *Proc. of 12th International Joint Conference on Artificial Intelligence*, pp. 746–750, 1991.

[中島 83] 中島秀之. *Prolog*. 産業図書, 1983.

[毛利 92] 毛利隆夫. 行動履歴からの事例の抽出と帰納推論による学習. 東京大学工学系研究科修士論文, 1992.