修士学位論文

帰納学習における背景知識の利用

平成 14 年度

茨城大学大学院理工学研究科

システム工学専攻

阿部修也

概要

本論文では帰納学習における背景知識の利用方法を提案する．学習手法として背景知識を容易に組み込めるという特徴をもつ帰納論理プログラミング（Inductive Logic Programming; ILP）と信頼ネットワーク（Belief Network; BN）を用いる．それぞれの手法を多義語の曖昧性解消問題に適用し，背景知識を用いない通常の学習手法との比較を行う．

自然言語処理では個々の問題を分類問題として定式化し，帰納学習の手法を利用して，その問題を解決するというアプローチが大きな成功をおさめている．しかしこのアプローチには帰納学習で必要とされる訓練データを用意しなければならないという大きな問題がある．大量の訓練データを用意するのは難しいため，少量の訓練データからどのようにして分類規則を学習すればよいかという問題である．これは機械学習における1つの重要な課題とされている．その解決方法の1つとして背景知識の利用が提案されている．背景知識とは，訓練データには明示されない問題固有の知識であり，広く捉えれば，人間の持つ常識的知識と考えて良い．一種の知識データベースである．問題はその背景知識を，どのように学習手法に取り入れてゆくかである．

本論文では2つの解決方法を提案する．1つはILPの利用である．ILPは訓練データを述語論理の形式で表し，そこから分類規則に相当する規則（述語論理の形式では節に対応）を導出する．知識データベースは述語論理の形式によって自然に表現できるので，背景知識の利用の観点からはILPを用いた学習戦略が優れている．更にILPの背景知識では，複雑なグラフ構造を持ったものも表現できるので，近年，CMUの機械学習チームはWebページの文書分類にILPを利用している．更にいくつかの自然言語処理への応用も知られている．

もう1つはBNの利用である．BNは因果的確率関係を，ノードに条件付き確率を付した有向グラフで表し，P(原因 | 結果) すなわち結果が与えられたときの原因の事後確率をメッセージ伝播を用いて計算する．ノード間の確率は訓練データ以外のリソースからも計算が可能な場合もあり，この点で背景知識を利用することができる．またBNは利用するモデルを柔軟に設定できるという長所も有している．特に通常の確率統計的な学習手法であるNaive Bayesを自然に拡張したモデルを利用できるために，より精度の高い規則を学習できることから，近年，活発に研究されている．

ILPを用いた手法の評価では，SENSEVAL2の日本語翻訳タスクを取り上げた．日本語翻訳タスクは，Translation Memory（TM）と呼ばれる日英対訳対が与えられ，テスト文

中の該当単語を英訳する際に利用できる TM の例文番号を返すタスクである．これは英訳を語義と考えた場合の多義語の曖昧性解消問題となっており，分類問題の一種である．このタスクは訓練データを新たに作成することが困難という特異なタスクになっており，TM の例文だけ，つまり少量の訓練データからどのようにして精度の高い分類規則を獲得するかが解決の鍵であり，背景知識を用いる ILP が有効であろう．実験では，ILP の実装システムとして Muggleton による Progol，背景知識として分類語彙表を利用した．その結果を確率統計的な学習手法である決定リストと比較し，本手法の優位性を示した．

BN を用いた手法の評価では，SENSEVAL2 の日本語辞書タスクを取り上げた．日本語辞書タスクは単純な多義語の曖昧性解消問題である．訓練データがある程度提供されているために，従来の学習手法がよい成績を納めている．特に Naive Bayes を用いたシステムの成績がよい．実験では，Naive Bayes を拡張したモデルを使い BN による学習を試みた．さらにその BN に背景知識を用いる手法も試した．評価は Naive Bayes との比較により行い，BN の優位性を示した．ただし実験では背景知識を用いても精度の向上は起らなかった．

ILP は確率の扱いが考慮されていない．また BN は述語論理のような推論機能が弱い．論理と確率をどのように結び付けるかは，背景知識を用いる学習手法の今後の課題である．

Abstract

In this master's thesis, I describe the use of background knowledge to classification problems. In proposed methods, I use the Inductive Logic Programming (ILP) and the Belief Network (BN), which easily use background knowledge in learning. I apply these two methods to word sense disambiguation problems, and evaluate them by comparing with the learning methods not using background knowledge.

Many problems in natural language processing can be converted into classification problems, and be solved by an inductive learning method. This strategy has been very successful, but it has a serious problem in that an inductive learning method requires a large amount of training data. This is the problem how to learn through small training data, and is an important issue in machine learning. To overcome this problem, the use of background knowledge have been proposed. Background knowledge is domain specific knowledge, and is roughly regarded as common knowledge which men have, that is, knowledge database. The problem is how to bring it in a learning method.

In this thesis, I use two learning methods to do it. One is ILP which generates classification rules from the data described by the predicate logic form. In this case the learned rules is expressed by clauses in predicate logic. Knowledge database can be described by the predicate logic form easily, so ILP is suitable as the method using background knowledge. Furthermore ILP can handle complex data with graph structure, so there are many applications of ILP to natural language processing. For example, the machine learning team in CMU has been using ILP to classify Web documents.

Another learning method is BN which use the directed acyclic graph (DAG) to express cause and effect. Nodes of the graph have the conditional probability `P(cause|effect)`. When the effect is given, BN computes this probability by using the message-passing algorithm. In some cases, the probability can be computed through other resource than training data, so background knowledge can be used in BN. Furthermore BN can expand the model assumed in Naive Bayes which is a standard statisticall learning method, so can generate better rules than Naive Bayes. For this reason, BN has been actively researched recently.

To evaluate the ILP method, I used the Japanese Translation Task in SENSEVAL2. In this task, the Japanese-English example sentences, named by Translation Memory (TM), is provided in advance. In real test, the system find the example, which is most useful in

translating the Japanese target word in the test sentence to English, and answer the ID number of that example. This task is a kind of the word sense disambiguation problems by taking the translated word as the sense, that is a classification problem. This task has the peculiar characteristic that it is difficult to make training data newly. The key of this task is how to build the classifier through only TM, that is small training data, so ILP is available. In experiments, I used Progol developed by Muggleton as the ILP system, and bunrui-goi-hyou as background knowledge to compare the decision list method, which is a statistical learning method. As a result, I showed the ILP is superior to the decision list method in this task.

To evaluate the BN method, I used the Japanese Dictionary Task in SENSEVAL2. This task is the simple word sense disambiguation problems. Because provided training data are not so small, general statistical learning methods are available. Particularly, the Naive Bayes method works well for this task. In experiments, I expanded the model assumed in the Naive Bayes method, and learned the classifier by using BN. Furthermore, I used background knowledge. These methods were compared to the Naive Bayes method, and BN was showed to be superior to the Naive Bayes method, but the method using background knowledge could not improve the the Naive Bayes method.

ILP is difficult to handle the probability. BN does not have such powerful deduction mechanism that ILP has. In learning using background knowledge, it is an important future work how To combine the logic and the probability.

目次

1 序論 1
1.1 背景 1
1.2 本論文の構成 2

2 Inductive Logic Programming 3
2.1 分類問題 3
2.2 Progol による分類問題の解法 3
2.3 語義の多義性解消問題への応用 7
2.4 背景知識の利用 8

3 Belief Network 10
3.1 ベイズの公理 10
3.1.1 事前確率 10
3.1.2 条件付き確率 12
3.1.3 ベイズ規則とその使用法 13
3.1.4 正規化 14
3.1.5 ベイズ規則の適用：証拠の組合せ 16
3.2 信念ネットワーク 18
3.3 Belief Network による語義の多義性解消問題 20
3.3.1 Naive Bayes による語義判別 20
3.3.2 Belief Network への拡張 21
3.3.3 モデルへ素性の設定 23
3.4 多重結合ネットワーク 25

4 実験 29
4.1 SENSEVAL2 29
4.2 ILP 29
4.3 BN 33

5 考察 39
5.1 ILP 39

5.1.1 ILP と確率的手法 39
5.1.2 背景知識の悪影響 39
5.1.3 規則の優先順位について 40
5.2 BN 41

6 結論 44
6.1 ILP 44
6.2 Belief Network 44

A データ 49

B プログラム 64

図目次

1 哺乳類に共通の規則の学習 . . . 6
2 素性を Progol の入力形式にした場合 . . . 8
3 分類語彙表の一部 . . . 9
4 素性と分類語彙表を結ぶ規則 . . . 9
5 確率変数の集合 . . . 20
6 リンクまたは矢印の集合がノード対を結ぶ . . . 20
7 各ノードは親ノードがそのノードへ及ぼす影響を定量化した条件付き確率表を持つ . . . 20
8 グラフは矢印の方向にサイクルを持たない . . . 20
9 天候によって傘が必要かどうか？ . . . 21
10 天候によって傘と様々な関係 . . . 23
11 Naive Bayes 型モデル . . . 24
12 拡張モデル（Belief Network） . . . 24
13 Belief Network . . . 26
14 クラスタリング法 . . . 26
15 接合木 . . . 27
16 接合木 . . . 28
17 デフォルト規則の生成 . . . 41
18 $M \rightarrow X$ と $N \rightarrow X$ で構成されるクラスタ . . . 42
19 名詞 その 1 . . . 50
20 名詞 その 2 . . . 51
21 名詞 その 3 . . . 52
22 名詞 その 4 . . . 53
23 名詞 その 5 . . . 54
24 名詞 その 6 . . . 55
25 名詞 その 7 . . . 56
26 動詞 その 1 . . . 57
27 動詞 その 2 . . . 58
28 動詞 その 3 . . . 59
29 動詞 その 4 . . . 60

30 動詞 その5 61
31 動詞 その6 62
32 動詞 その7 63

表目次

1 哺乳類による分類問題の例 4
2 天候の確率 22
3 ある天候の元で，傘が必要な確率 22
4 ILP の実験結果 31
5 BN の実験結果（名詞） 35
6 BN の実験結果（動詞） 37
7 生成された規則数の詳細 40
b

1 序論

1.1 背景

本論文では帰納学習における背景知識の利用方法を提案する．学習手法として背景知識を容易に組み込めるという特徴をもつ帰納論理プログラミング（Inductive Logic Programming; ILP）と信念ネットワーク（Belief Network; BN）を用いる．それぞれの手法を多義語の曖昧性解消問題に適用し，背景知識を用いない通常の学習手法との比較を行う．

自然言語処理では個々の問題を分類問題として定式化し，帰納学習の手法を利用して，その問題を解決するというアプローチが大きな成功をおさめている．しかしこのアプローチには帰納学習で必要とされる訓練データを用意しなければならないという大きな問題がある．これは少量の訓練データからどのようにして分類規則を学習すればよいかという問題であり，機械学習における 1 つの重要な課題とされている．その解決方法の 1 つとして背景知識の利用が提案されている．背景知識とは，訓練データには明示されない問題固有の知識であり，広く捉えれば，人間の持つ常識的知識と考えて良い．一種の知識データベースである．問題はその背景知識を，どのように学習手法に取り入れてゆくかである．

本論文では 2 つの解決方法を提案する．1 つは ILP の利用である．ILP は訓練データを述語論理の形式で表し，そこから分類規則に相当する規則（述語論理の形式では節に対応）を導出する．知識データベースは述語論理の形式によって自然に表現できるので，背景知識の利用の観点からは ILP を用いた学習戦略が優れている．更に ILP の背景知識では，複雑なグラフ構造を持ったものも表現できるので，近年，CMU の機械学習チームは Web ページの文書分類に ILP を利用している．更にいくつかの自然言語処理への応用も知られている．

もう 1 つは BN の利用である．BN は因果的確率関係を，ノードに条件付き確率を付した有向グラフで表し，P(原因 | 結果) すなわち結果が与えられたときの原因の事後確率をメッセージ伝播を用いて計算する．ノード間の確率は訓練データ以外のリソースからも計算が可能な場合もあり，この点で背景知識を利用することができる．また BN は利用するモデルを柔軟に設定できるという長所も有している．特に通常の確率統計的な学習手法である Naive Bayes を自然に拡張したモデルを利用できるために，より精度の高い規則を学習できることから，近年，活発に研究されている [10].

ILP を用いた手法の評価では，SENSEVAL2 の日本語翻訳タスクを取り上げた．日本語

翻訳タスクは，Translation Memory（TM）と呼ばれる日英対訳対が与えられ，テスト文中の該当単語を英訳する際に利用できるTMの例文番号を返すタスクである．これは英訳を語義と考えた場合の多義語の曖昧性解消問題となっており，分類問題の一種である．このタスクは訓練データを新たに作成することが困難という特異なタスクになっており，TMの例文だけ，つまり少量の訓練データからどのようにして精度の高い分類規則を獲得するかが解決の鍵であり，背景知識を用いるILPが有効であろう．実験では，ILPの実装システムとしてMuggletonによるProgol，背景知識として分類語彙表を利用した．その結果を確率統計的な学習手法である決定リストと比較し，本手法の優位性を示した [7, 8].

BNを用いた手法の評価では，SENSEVAL2の日本語辞書タスクを取り上げた．日本語辞書タスクは単純な多義語の曖昧性解消問題である．訓練データがある程度提供されているために，従来の学習手法がよい成績を納めている．特にNaive Bayesを用いたシステムの成績がよい．実験では，Naive Bayesを拡張したモデルを使いBNによる学習を試みた．さらにそのBNに背景知識を用いる手法も試した．評価はNaive Bayesとの比較により行い，BNの優位性を示した．ただし実験では背景知識を用いても精度の向上は起らなかった [13].

ILPは確率の扱いが考慮されていない．またBNは述語論理のような推論機能が弱い．論理と確率をどのように結び付けるかは，背景知識を用いる学習手法の今後の課題である．

1.2 本論文の構成

2章では帰納論理プログラミングについて，3章では信念ネットワークについて説明した．4章ではSENSEVAL2の日本語翻訳タスクを用いて帰納論理プログラミングの実験と，日本語辞書タスクを用いて信念ネットワークの実験を行った．5章では実験結果をふまえて考察を行った．6章では結論と今後の課題を述べた．

2 Inductive Logic Programming

本論文では，帰納論理プログラミング（Inductive Logic Programming，ILP）を利用して，多義語の曖昧性解消規則の学習を試みる．

自然言語処理の個々の問題を，分類問題として定式化し，帰納学習の手法により解決するというアプローチは大きな成功をおさめている．そして，そこで利用される帰納学習手法は，主に確率統計的な手法である．しかし分類問題はILPを利用しても解くことができる．ILPは確率統計的な手法にはない特徴も有しており，問題によっては実用的な手法として利用されている[6][11]．

確率統計的な手法にはないILPの大きな特徴は，背景知識の利用である[15]．背景知識は，訓練データとして与えられた情報とは別の情報のことを指す．この背景知識をうまく利用することで，確率統計的な手法よりも精度の高い規則を学習できる可能性がある．

本論文ではSENSEVAL2の日本語TMタスク[9]における多義語の曖昧性解消問題にILPの手法を適用した．ILPの実装システムとしてはProgolを利用した．背景知識としては分類語彙表を利用した．

2.1 分類問題

本論文では，多義語の曖昧性解消問題を扱うが，この問題は分類問題として定式化することができる．分類問題について説明を行う．

物事を分類するためには，分類カテゴリが与えられていることが必要である．たとえば，動物分類では，脊椎動物，無脊椎動物などのカテゴリが与えられている．さらに，脊椎動物の細分類として，哺乳類，爬虫類，両生類，鳥類，魚類の五種類が知られている．動物分類の問題は，ある動物がたとえば，哺乳類，爬虫類，両生類，鳥類，魚類のいずれに属しているかを，その動物の特徴からのみ判断することである．我々は常識的に嘴を持っていれば鳥類であろうと考えているが，例外があり，カモノハシは嘴があるにもかかわらず鳥類ではない．本論文目的は，分類規則をその特徴から自動的に生成することにある．

2.2 Progolによる分類問題の解法

ILPは，Muggleton[2][3]，Quinlan[4]等により開発された体系である．ILPの実装では，MuggletonによるProgol[3]やQuinlanによるFoil[5]が有名である．本実験では，Progolを使用する．

Progol は以下の特徴を持つ.

- 背景知識を利用し，帰納推論に基づいて学習を行う.
- 一階述語論理に基づく，高い表現能力をもつ.

表 1: 哺乳類による分類問題の例

動物名	類	授乳	足の数	住処	恒温
犬	哺乳類	○	4	陸上	○
イルカ	哺乳類	○	0	水中	○
鮫	魚類	×	0	水中	×
鷲	鳥類	×	2	空中	○

Progol が分類問題に対する分類規則を学習できることを示すため，ある動物が哺乳類なのか，それ以外の類なのかを判別する分類問題を考える [16]. 表 1 では，授乳するという特徴が，哺乳類のみに共通する特徴である. これは，哺乳類だけが授乳し，それ以外の類（魚類，鳥類）は授乳しないということを示している. つまり，哺乳類であることを分類する規則は，「授乳」である.

Progol は，事例（動物名）とその素性（授乳，足の数，住処，産卵，恒温）から，クラス（哺乳類）を判別する規則を学習する.

Progol は，事実を述語論理で表現するため，事例も素性もクラスも述語論理で記述する. 例えば，犬に関しては次の様に記述する.

```
class('犬', yes).            % 犬は哺乳類
milk('犬', yes).             % 犬は授乳される
legs('犬', 4).               % 犬の足の数は 4 本
habitat('犬', '陸上').       % 犬の住処は陸上
homothermic('犬', yes).      % 犬は恒温動物
```

Progol の入力ファイルと Progol による実行結果を，図 1 に示す. 正例は学習対象となる事例とそのクラスを表し，背景知識は事例の素性，それ以外はバイアスとして Progol の動作を制御する. バイアスの中でも，modeh は学習対象を，modeb は学習対象の素性を Progol に伝える.

実行結果の，class(A,yes) :- milk(A,yes). は，“A が授乳するならば，クラス（類）

が哺乳類"（A は変数）ということを示している．哺乳類とそれ以外の類を分類する規則は「授乳」であることを，正しく学習することができている．

```
% %以降，行末まではコメント

% 正例のみの学習
:- set(posonly)?

% モード宣言
% #nat は，数字を表すタイプとして，予め宣言されている
% animal, yn, habitat の要素はタイプ宣言を参照
:- modeh(*,class(+animal, #yn))?
:- modeb(*,milk(+animal, #yn))?
:- modeb(*,legs(+animal, #nat))?
:- modeb(*,habitat(+animal, #habitat))?
:- modeb(*,eggs(+animal, #yn))?
:- modeb(*,homeothermic(+animal, #yn))?

% タイプ宣言
animal('犬'). animal('イルカ'). animal('鮫'). animal('鷺').
yn(yes).  yn(no).
habitat('陸上'). habitat('水中'). habitat('空中').

% 正例
class('犬', yes). class('イルカ', yes).
class('鮫', no). class('鷺', no).

% 背景知識
milk('犬', yes). legs('犬', 4). habitat('犬', '陸上').
eggs('犬', no). homeothermic('犬', yes).

milk('イルカ', yes). legs('イルカ', 0). habitat('イルカ', '水中').
eggs('イルカ', no). homeothermic('イルカ', yes).

milk('鮫', no). legs('鮫', 0). habitat('鮫', '水中').
eggs('鮫', yes). homeothermic('鮫', no).

milk('鷺', no). legs('鷺', 2). habitat('鷺', '空中').
eggs('鷺', yes). homeothermic('鷺', yes).
```

```
% Progol による学習結果は次のようになる．
class(A,yes) :- milk(A,yes).
class(A,no) :- milk(A,no).
```


図 1: 哺乳類に共通の規則の学習

2.3 語義の多義性解消問題への応用

語義の多義性解消問題は分類問題として定式化できる．Progol は分類問題を解くことができるため，語義の多義性解消問題を解くことができる．語義のクラスは Progol では正例にあたり，文章の素性が背景知識に相当する．

素性として，多義語の曖昧性解消の対象となる単語の前後にある単語を用いる．具体的には，語義となる単語の直前の単語（“e1”）と直後の単語（“e2”），それを除いた，前方最大 3 単語（“e3”），後方最大 3 単語（“e4”）を使用する．

判別する語義が「与える」の例を示す．与えられた文章は，「彼では力不足という印象を与えるかもしれない」．この文章を単語分割すると「彼」「で」「は」「力」「不足」「と」「いう」「印象」「を」「与える」「かも」「しれ」「ない」と分割される．この文章の素性は次のようになる．

- 直前の単語 e1 = ’を’
- 直後の単語 e2 = ’かも’
- 前方の単語 e3 = {’と’, ’いう’, ’印象’}
- 後方の単語 e4 = {’しれ’, ’ない’}

Progol の入力形式で表わすと図 2 のようになる．Progol では，素性は背景知識として扱われる．また，ここで文章の ID とは，文章を識別するための番号である．

```
% 正例
class('文章の ID', 'クラス名').

% 背景知識
e1('文章の ID', 'を').
e2('文章の ID', 'かも').
e3('文章の ID', 'と').
e3('文章の ID', 'いう').
e3('文章の ID', '印象').
e4('文章の ID', 'しれ').
e4('文章の ID', 'ない').
```

図 2: 素性を Progol の入力形式にした場合

2.4 背景知識の利用

例えば,「宿題」と「課題」という単語を考える. この 2 つの単語は異なる単語だが, 同じ意味を持っていると考えられる. しかし, 異なる単語が同じ意味を持つ可能性をシステムは知らない. 語義の多義性解消問題は, 単語の意味に基いてルールを生成した方がよい. そのため異なる単語でも意味が同じならば, 同じ単語として扱った方がよい.

上記の考えを背景知識として組み入れるために, 分類語彙表を利用する. 分類語彙表は, 階層構造を持つ. 階層が上がるほど, 同じ意味の単語は増加する. (図 3).

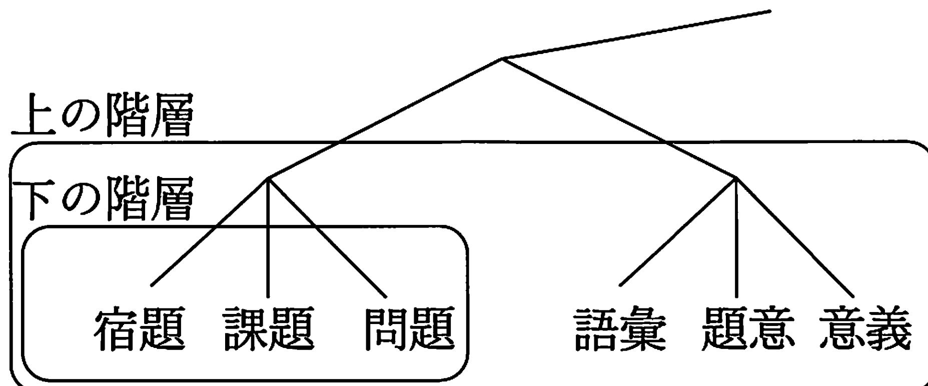


図 3: 分類語彙表の一部

一番下の階層を考えると，「課題」「宿題」「問題」は，同じ意味のグループである．「語彙」「題意」「意義」は，別の意味のグループである．同じ意味のグループに属する単語は，同じ意味を持つ．1 つ上の階層を考えると，「課題」「宿題」「問題」「語彙」「題意」「意義」という単語は同じ意味になる．

規則（図 4）は，素性（e1，e2，e3，e4 の単語）と，分類語彙表に含まれる単語と，その単語と同じ意味を持つ単語を結び付ける．つまり，Progol は同じ意味の単語を同じ単語として扱うようになる．述語 b は，分類語彙表の単語とその意味を表わす数字の組を表す．述語 e1　　e4 は，文章の素性を表す．述語 e1_w　　e4_w は，述語 e1　　e4 の代りとして，文章の素性を表す．また，規則の記述を簡略化するために，素性が分類語彙表に存在しない場合は，あらかじめ素性を分類語彙表に登録する．

```
e1_w(ID, Word1) :- e1(ID, Word2),
                   b(Word2, Number), b(Word1, Number).
e2_w(ID, Word1) :- e2(ID, Word2),
                   b(Word2, Number), b(Word1, Number).
e3_w(ID, Word1) :- e3(ID, Word2),
                   b(Word2, Number), b(Word1, Number).
e4_w(ID, Word1) :- e4(ID, Word2),
                   b(Word2, Number), b(Word1, Number).
```

図 4: 素性と分類語彙表を結ぶ規則

3 Belief Network

本論文では信念ネットワーク（Belief Network; BN）を用いた語義判別規則の学習を試みる．

自然言語の個々の問題を分類問題として定式化し，帰納学習の手法により解決するというアプローチは大きな成功を納めている．そこで利用される帰納学習手法は主に確率統計的な手法である．特に Naive Bayes は単純なモデルであるにも関わらず，比較的よい規則を学習できるために広く利用されている [14]. Naive Bayes は変数間の独立性を仮定しているために計算が容易であるという長所をもつが，その反面変数間の関係を記述することができず，ある種の情報を落しているとも考えられる．一方，Belief Network は確率推論技術の一種であり，利用するモデルを柔軟に設定できることから，近年，活発に研究が行われている [10]. そこで本論文では，多義語の曖昧性解消問題を題材に，Naive Bayes のモデルに変数間の関係を追加したモデルを設定し，Belief Network による学習を試みる．

また Belief Network は学習の訓練データの中に背景知識を容易に組み込めるという特徴も有している．これは Naive Bayes などの確率統計的な手法にはない大きな長所である．本論文では背景知識を導入した場合の実験も行う．

実験では SENSEVAL2 の日本語辞書タスクを用いる．Naive Bayes との正解率の比較により本手法を評価する．

3.1 ベイズの公理

3.1.1 事前確率

命題 A が真である無条件（unconditional）確率あるいは事前確率（prior probability）を表すために $P(A)$ という記法を用いる．例えば，*Cavity* をある特定の患者が虫歯を持っているという命題としよう．

$$P(Cavity) = 0.1 \tag{1}$$

という表現は，何の情報もない場合に，エージェントが，患者が虫歯をもっていうという事象に対して 0.1（または 10%）という確率を付与するということを意味する．$P(A)$ という記法は何の情報もない場合にのみ用いられることを覚えておくことは重要である．つまり，新しい情報 B がわかったときには，$P(A)$ の代わりに B があるときの A の条件付き

確率を推論する必要がある．条件付き確率は次の節で説明する．

確率的表明の中で使われている命題は，$P(A)$ の中の A のように命題記号で表すことができる．命題は，ランダム変数（random variable）を用いた等式としても表わすことができる．例えば，*Weather* というランダム変数を使えば，以下のように確率を表すことができる．

$$P(Weather = Sunny) = 0.7 \tag{2}$$
$$P(Weather = Rain) = 0.2 \tag{3}$$
$$P(Weather = Cloudy) = 0.08 \tag{4}$$
$$P(Weather = Snow) = 0.02 \tag{5}$$
$$\tag{6}$$

各ランダム変数 X は，定義域（domain）$< x_1, ..., x_n >$ を持つ．これは，その変数が取ることのできる可能な値のことである．我々は通常離散値の集合を扱う．我々は，命題記号を $< true, false >$ という定義域を持つランダム変数とみなすこともできる．したがって，$P(Cavity)$ という式は $P(Cavity = true)$ の略記法と見ることができる．同様に，$P(\neg Cavity)$ は $P(Cavity = false)$ の略である．通常，A，B などの文字をブールランダム変数として用い，X，Y などの文字を二つ以上の複数の値を取りうる変数として用いる．

ときどき，ランダム変数のすべての値の確率をひとまとめにして議論したいときがある．天気の例で言えば，$P(Weather)$ のような式を用いて天気の個々の状態の確率の値のベクタを表す．例えば，上記の値を用いれば，

$$P(Weather) =< 0.7, 0.2, 0.08, 0.02 > \tag{7}$$

と書くことができる．この表明は，ランダム変数 *Weather* に対する確率分布（probability distribution）を定義している．

また，$P(Weather, Cavity)$ のような表現を用いて複数のランダム変数の値のすべての組合せの確率を表す．例えば，$P(Weather, Cavity)$ は 4×2 の大きさの確率の表を表す．等式がたくさんあるとき，この表記がそれを簡略化するのに役立つことが後でわかる．

論理結合子を用いてより複雑な文を作り出し，それに対する確率を付与することができる．例えば，

$$P(Cavity \wedge \neg Insured) = 0.06 \tag{8}$$

は，患者が虫歯で何の保険も掛けていない確率が 6％ であることを意味する．

3.1.2 条件付き確率

エージェントがドメイン中の命題に関係する証拠を得ると，事前確率はもはや適用できなく，その代わり，我々は，条件付き（conditional）あるいは事後（posterior）確率を用いる．記法としては，$P(A|B)$ を用いる．これは，「知っていることが B のみであるときの A の確率」を表わす．例えば，

$$P(Cavity|Toothache) = 0.8 \tag{9}$$

は患者が歯痛であることが観察され，かつ，それ以外の情報がない場合に，患者が虫歯を持つ確率が 0.8 であることを表している．ここで注意しなければならないことは，$P(A|B)$ は知っていることが B のみでないときは，もはや使うことができないということである．B だけでなく C を知った場合は，条件付き確率 $P(A|)$ の特別な場合であると考えられる．すなわち，何の証拠がない場合の条件付き確率を表していると考えることができる．

$P()$ 記法は，条件付き確率にも適用できる．$P(X = x_i|Y = y_i)$ が X，Y の可能な値 x_i，y_i に対して与えられたときに，$P(X|Y)$ は，2 次元の表となる．また，条件付き確率は，以下のように無条件確率で表すことができる．関係式

$$P(A|B) = \frac{P(A \wedge B)}{P(B)} \tag{10}$$

は，$P(B) > 0$ のときには必ず成り立つ．この関係式は，以下のように表すこともでき，積の公式（product rule）と呼ばれている．

$$P(A \wedge B) = P(A|B)P(B) \tag{11}$$

積の公式は以下のように覚えると簡単かもしれない．「A と B が真となるためには，B が真であるとともに B が真であるときに A が真でなければならない」．さらに，上の式を以下のように書くこともできる．

$$P(A \wedge B) = P(B|A)P(A) \tag{12}$$

場合によっては，連言の事前確率を考えるほうが簡単になることがあるが，ほとんどの場合には，確率推論の対象は条件付き確率である．

省略．

3.1.3 ベイズ規則とその使用法

$$P(A \wedge B) = P(A|B)P(B) \quad (13)$$

$$P(A \wedge B) = P(B|A)P(A) \quad (14)$$

両辺の右辺が等しいのでそれを等号で結んで両辺を $P(A)$ で割れば

$$P(B|A) = \frac{P(A|B)P(B)}{P(A)} \quad (15)$$

を得る．この関係式は，ベイズ規則（Bayes' rule）（またはベイズの法則またはベイズの定理）として知られている．この単純な関係式は，確率推論を行なう現在のすべての AI システムの基礎となっている．多値変数のためのより一般的な場合は，P 記法を用いて以下のように表すことができる．

$$P(Y|X) = \frac{P(X|Y)P(Y)}{P(X)} \quad (16)$$

上は，前の記法と同様，表どうしの対応する要素ごとの関係式の集合を表している．また，ある証拠 E が存在したときの条件付き確率を表す一般的な式は以下のようになる．

$$P(Y|X,E) = \frac{P(X|Y,E)P(Y|E)}{P(X|E)} \quad (17)$$

ベイズ規則の適用：簡単な場合

ベイズ規則の簡単な例

一見するとベイズ規則はあまり役に立つとは思えない．なぜなら，一つの条件付き確率を計算するのに条件付き確率一つと事前確率二つを必要とするからである．

しかし，実際上は，これら必要な三つの確率の値をうまく見積もることができて，なおかつそれらから計算される条件付き確率がしばしば求めたいものになっているため，ベイズ規則は役に立つ．医療診断のような分野では，因果関係の条件付き確率がわかっていて，診断の確率を導きたいことがよくある．ある医者は脳膜炎が肩こりを 50％ の確率で生じさせること知っているとしよう．また，患者が脳膜炎である事前確率は 1/50,000 であり，患者が肩こりになっている事前確率が 1/20 であることもその医者が知っていると

しよう．S を患者が肩こりになっているという命題とし，M を患者が脳膜炎であるという命題を表すとすれば，以下のような計算ができる．

$$
\begin{aligned}
P(S|M) &= 0.5 \\
P(M) &= 1/50000 \\
P(S) &= 1/20 \\
P(M|S) &= \frac{P(S|M)P(M)}{P(S)} = \frac{0.5 \times 1/50000}{1/20} = 0.0002
\end{aligned}
$$

すなわち，肩こりとなっている患者 5000 人のうち 1 人だけが脳膜炎であるとおそれがあるということである．ここで，気をつけなければならないのは，たとえ脳膜炎のときに肩こりが確率 0.5 で起るとしても，肩こりになっている患者が脳膜炎である確率は小さいままである，ということである．これは，肩こりの事前確率が脳膜炎の事前確率よりもたいへん高いためである．

上の議論からすぐ生じる疑問は，なぜ条件付き確率が一方向からしかわからないのかということである．上の脳膜炎の場合では，医者が上で述べた確率の情報だけではなく，肩こりをもつ患者 5000 人のうち一人が脳膜炎であるということも知っているだろうから，ベイズ規則が必要でないのではないかという疑問が生じるだろう．残念ながら，診断知識自体は，因果知識よりも状況の変化に敏感であることが多いためあまり役に立たない．例えば，脳膜炎の突然の流行により，脳膜炎の事前確率 $P(M)$ が増えたとする．流行以前に，患者の統計的な観察から $P(M|S)$ を得ていた医者は，どのようにその値を変更すればよいかまったくわからないが，$P(M|S)$ を他の三つの値から得ていた医者は，$P(M|S)$ が $P(M)$ に比例して増加することがわかる．より重要なことは，因果関係 $P(S|M)$ は，流行によって影響されないことである．なぜならば，$P(S|M)$ は，脳膜炎の性質を表しているからである．この種の直接的な因果関係やモデルに基づく知識を使うことによって，確率システムが実際に役に立つために必要な頑健性が得られる．

3.1.4 正規化

患者が肩こりになたときの脳膜炎の確率を計算する以下の式について再び考える

$$
P(M|X) = \frac{P(S|M)P(M)}{P(S)} \tag{18}
$$

ここで，肩こりになっている患者がムチ打ち症である確率 $P(W|S)$ も知りたいとしよう

$$P(W|S) = \frac{P(S|W)P(W)}{P(S)} \tag{19}$$

この二つの関係式をよく見ると，肩こりの事前確率 $P(S)$ を調べなくても，肩こりの場合に，脳膜炎であるときのムチ打ち症であるときの相対的尤度（relative likelihood）を計算できることがわかる．$P(S|W) = 0.8$ かつ $P(W) = 1/1000$ とすれば，

$$\frac{P(M|S)}{P(W|S)} = \frac{P(S|M)P(M)}{P(S|W)P(W)} = \frac{0.5 \times 1/50000}{0.8 \times 1/1000} = \frac{1}{80} \tag{20}$$

となり，肩こりの場合に，ムチ打ち症である方が腱鞘炎であるよりも 80 倍も起こりやすいことがわかる．

相対的尤度は，場合によっては，意思決定に十分なこともあるかもしれないが，上の例のように，二つの病気が異なる治療処置を必要とし，その処置に対して効用が劇的に違う場合には，合理的意思決定のための正確な値を必要とする．しかし，すべての可能性を考えることで，「兆候」の事前確率を調べなくてもよいことがある．例えば，M と $\neg M$ に対する関係式は，以下のように書ける

$$P(M|S) = \frac{P(S|M)P(M)}{P(S)} \tag{21}$$

$$P(\neg M|S) = \frac{P(S|\neg M)P(\neg M)}{P(S)} \tag{22}$$

二つの関係式の両辺の和を取り，$P(M|S) + P(\neg M|S) = 1$ より，

$$P(S) = P(S|M)P(M) + P(S|\neg M)P(\neg M) \tag{23}$$

となる．$P(M|S)$ に関するベイズ規則の中の $P(S)$ を上の右辺で置き換えれば，

$$P(M|S) = \frac{P(S|M)P(M)}{P(S|M)P(M) + P(S|\neg M)P(\neg M)} \tag{24}$$

となる．この過程は，正規化（normalization）と呼ばれている．なぜなら，$1/P(S)$ を正規化定数として扱い，条件付き確率の総和が 1 になるようにしているからである．したがって，$P(S|\neg M)$ を調べることにより，$P(S)$ を調べることなく，ベイズ規則から正確な確率を得ることができる．より一般的な多値の場合には，以下のようなベイズ規則となる

$$P(Y|X) = \alpha P(X|Y)P(Y) \tag{25}$$

ここで，α は，$P(Y|X)$ の要素の和が 1 になるための正規化定数である．正規化を用いる通常の方法は，正規化されていない値を計算したあと，それらの総和が 1 になるようにその値を変更するものである．

3.1.5 ベイズ規則の適用：証拠の組合せ

虫歯に関係する以下の条件付き確率がわかっているとしよう

$$P(Cavity|Toothache) = 0.8 \tag{26}$$
$$P(Cavity|Catch) = 0.95 \tag{27}$$
$$\tag{28}$$

これらは，ベイズ規則を使って計算されたであろう．ここで，歯医者がさぐり針で患者の痛い歯を調べて，さぐり針がその歯でひっかかった（catch）としよう．もし，すべての結合確率を知っていたら，$P(Cavity|Toothache \wedge Catch)$ を計算するのはたやすいが，それができないとする．すると，$P(Cavity|Toothache \wedge Catch)$ を計算するための別な方法は，ベイズ規則を用いて上の問題を以下のように変換して考えることである

$$P(Cavity|Toothache \wedge Cavity) = \frac{P(Toothache \wedge Cavity|Cavity)P(Cavity)}{P(Toothache \wedge Cavity)} \tag{29}$$

これがうまくいくためには，*Cavity* のときの $Toothache \wedge Catch$ の条件付き確率を知る必要がある．n 個の変数に対して，*Cavity* であるときの条件付き確率を見積もることは，可能かもしれないが，n^2 個の変数のペアの連言に対する条件付き確率を考えることは，気の滅入る仕事である．さらに悪いことには，診断は上記のようなペアの連言ではなく，はるかに多数の変数の連言に依存するかもしれないということである．これは，診断を完全にするために指数オーダの値を必要とすることを意味し，結合確率を用いた場合と同じことになってしまう．この困難によって，研究者の中には，確率論をあきらめて，証拠の組合せに関する近似手法を研究するものも現れた．そのような近似手法では，答えが誤っているおそれがあるが，より少ない値で何らかの答えを出すことができる．

しかし，多くのドメインでは，以下のようにベイズ規則を使うことによって，結果を得るために必要な確率値の数を減らすことができる．その第 1 ステップは，複数の証拠を組

み合わせる過程を違った観点から見ることである．ベイズ更新（Bayesian updating）のプロセスは，証拠を一つずつ組み合わせて，所望の変数の確率を順に更新するものである．例えば，*Toothache* から始まる関係式は以下となる．（以下では，ベイズ規則を更新プロセスに合わせて書いていく）

$$P(Cavity|Toothache) = P(Cavity)\frac{P(Toothache|Cavity)}{P(Toothache)} \tag{30}$$

Catch が観察されると，*Toothache* を既にわかっている証拠として扱って，ベイズ規則を適用する．

$$P(Cavity|Toothache \wedge Catch) \tag{31}$$

$$= P(Cavity|Toothache)\frac{P(Catch|Toothache \wedge Cavity)}{P(Catch|Toothache)} \tag{32}$$

$$= P(Cavity)\frac{P(Toothache|Cavity)}{P(Toothache)}\frac{P(Catch|Toothache \wedge Cavity)}{P(Catch|Toothache)} \tag{33}$$

ゆえにベイズ更新では，新しい証拠が得られると，確率を知りたい変数の信念は，その信念と新しい証拠に依存する因子との積に変更される．また，この過程が我々の期待どおりに証拠が得られる順番に依存しない．

しかし，まだ問題が解決されたわけではない．なぜなら，上記の積の因子は，新しい証拠に依存しているだけでなく既に得られた証拠にも依存しているからである．因子 $P(Catch|Toothache \wedge Cavity)$ の値を見つけることは，$P(Toothache \wedge Catch|Cavity)$ の値を見つけることよりも必ずしも簡単とは言えない．この式を簡単にするためには，本質的な仮定をいれる必要がある．ここで，虫歯の例において鍵となることは，虫歯であることが，歯が痛いことと，さぐり針がその歯で引っかかることの直接の原因となっていることである．もし患者が虫歯であることがわかれば，さぐり針が引っかかることの確率が歯痛のあるなしに関係することはないし，逆に，さぐり針で虫歯を触るか触らないかによって歯痛の確率が変わることもないであろう．数学的には，上記の以下のように書ける．

$$P(Catch|Cavity \wedge Toothache) = P(Catch|Cavity) \tag{34}$$

$$P(Toothache|Cavity \wedge Catch) = P(Toothache|Cavity) \tag{35}$$

この関係式は，*Cavity* であるときの *Toothache* と *Catch* に関する条件付き独立性（conditional independence）を表している．条件付き独立性があれば，更新の関係式を以下のように簡単にできる

$$P(Cavity|Toothache \wedge Catch) \tag{36}$$

$$= P(Cavity)\frac{P(Toothache|Cavity)}{P(Toothache)}\frac{P(Catch|Cavity)}{P(Catch|Toothache)} \tag{37}$$

まだ，$P(Catch|Toothache)$ という項が残っているため，すべての兆候のペアの連言（またはそれ以上の連言）を考えないといけないように見えるが，実際には，この項はなくなってしまう．分母の積は，$P(Catch|Toothache)P(Toothache)$ すなわち $P(Toothache \wedge Catch)$ であることに注意してほしい．この項は，$P(Toothache|\neg Cavity)$ と $P(Catch|\neg Cavity)$ の値を調べることで，先ほどの正規化により消すことができる．結果として，単一の証拠の場合に帰着できるため，原因の事前確率とその影響に関する条件付き確率のみを考えればよいことになる．

多値変数の場合でも条件付き独立を用いることができる．Z が与えられたときに X と Y が条件付き独立であるとは，

$$P(X|Y,Z) = P(X|Z) \tag{38}$$

のごとである．ここで，上の式は，おのおのの条件付き独立の表明の集合を表している．ブール変数でのベイズ規則の簡単化に対応する多値変数での簡単化は

$$P(Z|X,Y) = \alpha P(Z)P(X|Z)P(Y|Z) \tag{39}$$

と表すことができる．ここで，α は，$P(Z|X,Y)$ の要素の総和が 1 になるようにする正規化定数である．

ここで覚えておかなければならないことは，ベイズ更新の簡単化は，条件付き独立の関係が成立しているときのみに使えるということである．したがって，条件付き独立の情報は確率システムが効果的に働くための決定的な要素である．

3.2 信念ネットワーク

BN（Belief Network）はノード間の信頼度に基いた有向グラフで表現され、自由にモデルを選択できることから NB（Naive Bayes）で表現できないモデルを用いることができる。また、NB と同様に背景知識を用いることもできる。さらに、不確実性が存在する状況下でも整合性のとれる推論で知られているため、語義の多義性解消問題のような不確実

性が存在する問題への適用に期待がもてる。本研究ではNBでは扱えないモデルを使いさらに背景知識を利用することで、BNの利点を生かしこれを語義の多義性解消問題に応用させた。

ベイジアンネットでは，因果的確率関係を，ノードに条件付き確率を付した有向グラフで表現し，P（原因—結果）すなわち結果が与えられたときの原因の事後確率をメッセージ伝搬と呼ばれるアルゴリズムを使い，効率良く計算する．

確率計算は不確定性を扱う基本中の基本技術として広く用いられているが，大量の計算パワーを必要とするので（一般に確率計算はNP困難である）計算機パワーの乏しい時代には実用性が乏しかった．

1986年単結合ベイジアンネットワーク（エッジの向きを無視したとき，グラフがループをもたない）に適応できる信念伝搬（belief propagation）アルゴリズムがPearlにより導入され，事後確率の計算がグラフのノード数に比例する時間で計算できるようになったことにより状況が変わった．このアルゴリズムはノード間の$\lambda\pi$と呼ばれるメッセージを並列非同期的により取りすることにより確率を計算する．さらに，グラフがループをもつ複結合ベイジアンネットでも，ノードをまとめたクラスタ単位としてメッセージ伝搬を能率良く行う接合木（junction tree）アルゴリズムが1988年にLauritzenとSpiegelhalterにより提案された．

変数間の依存関係を表現するために信念ネットワーク（belief network）と呼ぶデータ構造を使用し，結合確率分布の簡明な表現を与える．信念ネットワークは次の性質を有するグラフである

1. 確率変数の集合がネットワークのノードを形成する（図5).
2. リンクまたは矢印の集合がノード対を結ぶ．ノードXからノードYへの矢印の直接的な意味は，XがYに直接的影響を及ぼすということである（図6).
3. 各ノードは親ノードがそのノードへ及ぼす影響を定量化した条件付き確率表を持つ．ノードの親とは，そのノードを矢印によって指すすべてのノードである（図7).
4. グラフは矢印の方向にサイクルを持たない．したがって，有向非循環グラフ（DAG）である（図8).

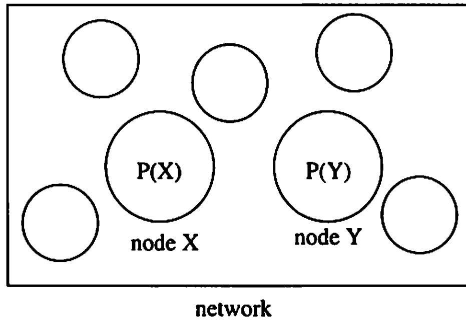


図 5: 確率変数の集合

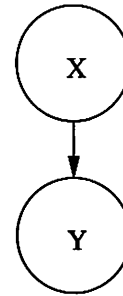


図 6: リンクまたは矢印の集合がノード対を結ぶ

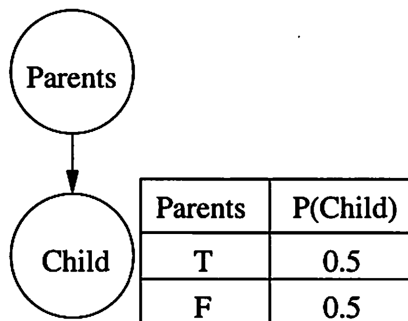


図 7: 各ノードは親ノードがそのノードへ及ぼす影響を定量化した条件付き確率表を持つ

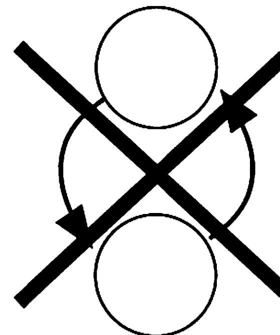


図 8: グラフは矢印の方向にサイクルを持たない

3.3 Belief Network による語義の多義性解消問題

3.3.1 Naive Bayes による語義判別

ある事例 x が素性のリストとして，以下のように表現されたとする．

$$x = (f_1, f_2, ..., f_n) \tag{40}$$

x の分類先のクラスの集合を $C = \{c_1, c_2, ..., c_m\}$ と置く．分類問題は $P(c|x)$ の分布を推定することで解決できる．実際に，x のクラス c_x は以下の式で求まる．

$$c_x = \arg\max_{c \in C} P(c|x) \tag{41}$$

ベイズの定理を用いると，

$$P(c|x) = \frac{P(c)P(x|c)}{P(x)}$$

なので，結局，以下が成立する．

$$c_x = \arg\max_{c \in C} P(c)P(x|c)$$

ここで，$P(c)$ は比較的簡単に推定できる．問題は，$P(x|c)$ の推定だが，これは現実的に難しい．Naive Bayes のモデルは，素性間の独立性を仮定して，以下の変形により $P(x|c)$ の推定を行う．

$$P(x|c) = \prod_{i=1}^{n} P(f_i|c) \tag{42}$$

3.3.2 Belief Network への拡張

Naive Bayes は素性間の独立性を仮定している．確率モデルの観点でみると素性は確率変数に対応しており，確率変数間の独立性を仮定してると見なせる．しかし現実には確率変数間には何らかの依存関係がある場合が多く，より自然なモデルを扱うためには Naive Bayes は適していない．そこでここでは素性間の依存関係を考慮したモデルを扱える Belief Network を用いることにする．

Belief Network は以下の条件を満たす，有向非循環グラフ（Directed Acyclic Graph; DAG）として表される．

- 確率変数の集合がネットワークのノードを形成する．
- リンクまたは矢印の集合がノード対を結び，ノード間の因果的確率関係を表す．
- 各ノードは親ノードがそのノードへ及ぼす影響を定量化した条件付き確率表を持つ．
- グラフは矢印の方向にサイクルを持たない．

図 9: 天候によって傘が必要かどうか？

例を示す．図 9 は天候によって傘が必要かどうか判断されることを表している．雨ならば傘が必要で，晴天ならば傘は必要ない．しかし，目的地までの距離が近い場合や地下鉄

で移動できる場合は雨が降っていても傘は必要ない．紫外線に気を配る人は晴天でも傘が必要になる．曇りの日は傘が必要かどうかわからない．図 9 はこのような天候と傘に関する条件付き確率の関係を表すモデルとして捉えられる．そして天候の確率は表 2，ある天候のとき傘が必要な条件付き確率は表 3 の形にまとめられる．

表 2: 天候の確率

天候	P(天候)
晴天	0.3
曇り	0.5
雨天	0.2

表 3: ある天候の元で，傘が必要な確率

天候	傘が必要か？	P(傘 | 天候)
晴天	必要	0.1
	必要なし	0.8
曇り	必要	0.5
	必要なし	0.5
雨天	必要	0.9
	必要なし	0.1

しかし，現実はもう少し複雑であり，季節や天気予報というファクターが天候や傘に影響を与えている．例えば，季節によって雨が多い降ることもある．また，夏に日傘をさす人は多いが，日差しが強くても冬はあまり日傘をささない．さらに，実際の天候によって天気予報は変化し，天気予報によって傘が必要かどうか判断する人は多い．このような因果関係を先のモデルで表した場合，図 10 のようになる．

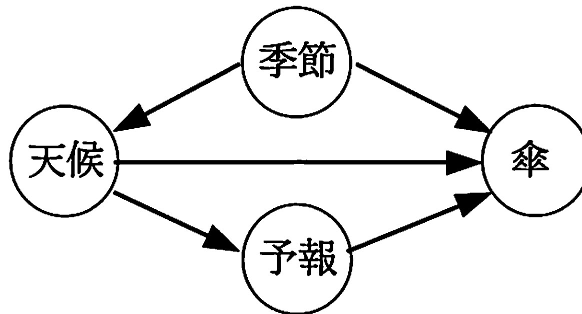


図 10: 天候によって傘と様々な関係

図 10 のモデルは図 9 の拡張になっており，より現実世界を反映しているが，図 1 と比べて条件付き確率を計算するのが困難になっている．Belief Network は図 10 のようなモデルに対して，効率よく条件付き確率を計算できる学習手法である．

3.3.3 モデルへ素性の設定

語義判別の手がかりとなる属性として以下のものを設定した．

e1	**直前の単語**
e2	**直後の単語**
e3	**前方の内容語２つまで**
e4	**後方の内容語２つまで**

例えば，語義判別対象の単語を「出す」として，以下の文を考える（形態素解析され各単語は原型に戻されているとする）．

短い/コメント/を/出す/に/とどまる/た/。

この場合，「出す」の直前，直後の単語は「を」と「に」なので，**‘e1=を’**，**‘e2=に’** となる．次に，「出す」の前方の内容語は「短い」と「コメント」なので，**‘e3=短い’**，**‘e3=コメント’** の２つが作られる．またここでは句読点も内容語に設定しているので，「出す」の後方の内容語は「とどまる」「。」となり，**‘e4=とどまる’**，**‘e4=。’** が作られる．

結果として，上記の例文に対しては以下の６つの素性が得られる．

e1=を，e2=に，e3=短い，e3=コメント，

e4=とどまる， e4=。

図 11 は，素性間の独立性を仮定した Naive Bayes モデルである．直前の単語（e1），直後の単語（e2），前方の単語（e3），後方の単語（e4）それぞれの間にリンクがなく，単語間の関係は考慮されていない．

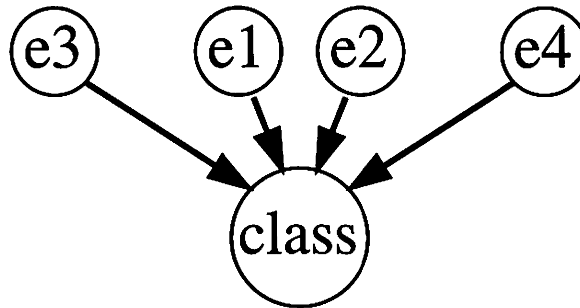


図 11: Naive Bayes 型モデル

図 12 は，前方の単語（e3）から直前の単語（e1）への影響，後方の単語（e4）から直後の単語（e2）への影響を追加したモデルである．

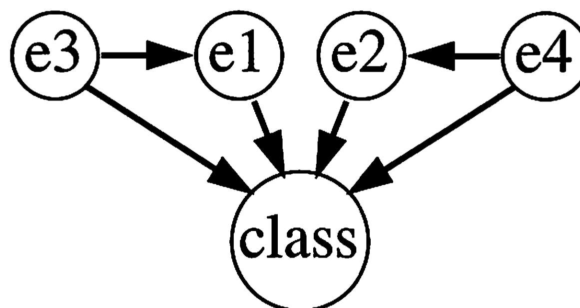


図 12: 拡張モデル（Belief Network）

図 11 では素性同士の関係が独立であるが，前方の単語と直前の単語がなんの関係も持たないとは考え難い．そこで，前方の単語から直前の単語への関係を追加したのが図 12 である（同様に後方の単語から直後の単語への関係も追加した）．

本論文では，背景知識としてラベルなしデータを用いる．ラベルなしデータとは素性の分類が示されていないデータある．しかし，素性とその分類から学習を行う帰納学習の際に用いるデータは，素性の分類が与えられているラベル付きデータである．ところが，ラ

ベル付けデータを作成するのは難しく，十分な量を用意することができない．一方で，背景知識として用いるラベルなしデータは，ラベル付けする必要がないことから簡単に作成することができる．そのため，ラベルなしデータを用いて素性間の関係を学習することで，高い正解率を期待できる．

本論文ではラベルなしデータから単語同士の関係を学習することで，前方の単語から直前の単語（e3 → e1）と，後方の単語から直後の単語（e4 → e2）の条件付き確率をより正確に求める．

3.4 多重結合ネットワーク

BN はネットワーク構造によって，単結合ネットワーク（singly connected network, poly tree）と複結合ネットワーク（multiply connected network）に分類される．

- 単結合ネットワークは，2 ノード間の経路が最大 1 本で構成されるネットワークの事をいう．
- 複結合ネットワークは，2 ノード間の経路が 2 本以上になりうるネットワークの事をいう．

本論文で扱うモデル（図 12）は複結合ネットワークである．例えば，e3 → class の経路は，e3 → class と e3 → e1 → class の 2 本ある．

ネットワークの構造によって BN の解法アルゴリズムが異る．単結合ネットワークの場合は，一般的な BN の解法である Pearl のアルゴリズムを用いることができる．しかし，複結合ネットワークの場合それを使うことができない．本論文で検討した複結合ネットワークの解法を挙げる．

1 つは，クラスタリング（clustering）法である．この解法は，単結合ネットワークとして不都合なノードを併合することにより，多重結合ネットワークを確率的に等価な（しかしトポロジ的には異なる）多重木に変形する．そのため，正確な解法であるが，最悪の場合ネットワークの大きさは指数的に増大する．図 14 は，複結合ネットワークである図 13 をクラスタリング法を用いて，単結合ネットワークに変換した．

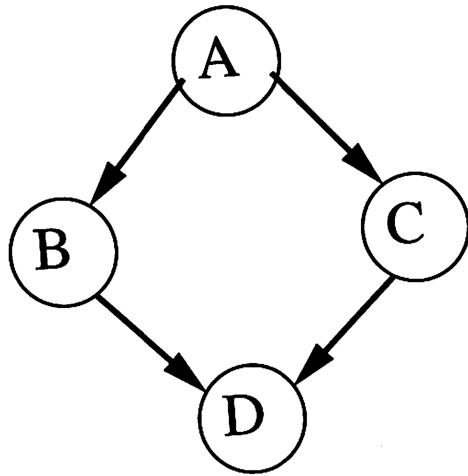


図 13: Belief Network

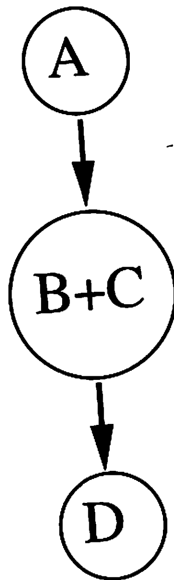


図 14: クラスタリング法

もう 1 つ，接合木（Junction Tree, Join Tree）を用いる解法もある．接合木は木構造の一種であると同時に，Belief Network から接合木を構築して，それを用いて確率の計算を行うこともいう．接合木は変数間の条件付き独立性をできるだけ保ち，かつ任意の周辺分布の計算に便利な形になっている．そのため接合木を用いた確率計算は厳密に正確ではないが，近似的に正しい計算を素早く行うことできる．図 13 から構築した接合木が，図 15

である．

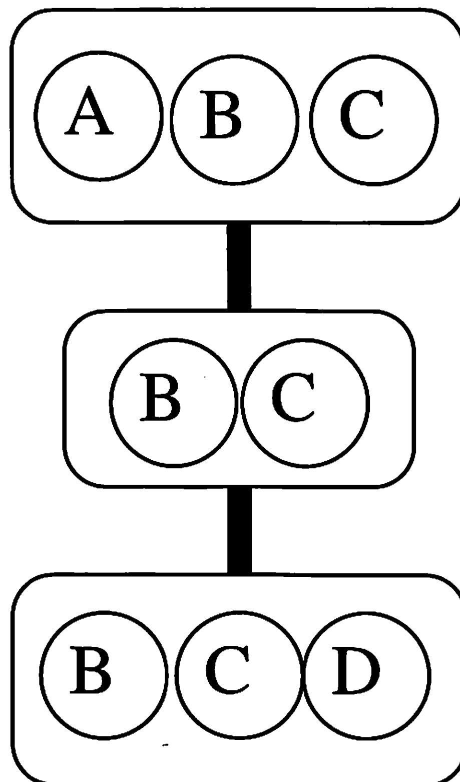


図 15: 接合木

接合木の構造に注目すると，接合木は 3 つの変数を一つのクラスタ（cluster, clique）として扱い，クラスタの集合が木構造を成している．計算の際は，クラスタ単位で条件付き確率から，証拠をもとに事後確率を求める．その後，メッセージ伝播により木構造全体に事後確率を伝える．証拠とは変数に与えられる値のことで，変数から変数への条件付き確率に証拠が与えられると，変数から変数への事後確率が決定する．図 15 は，図 16 と書き直せる．

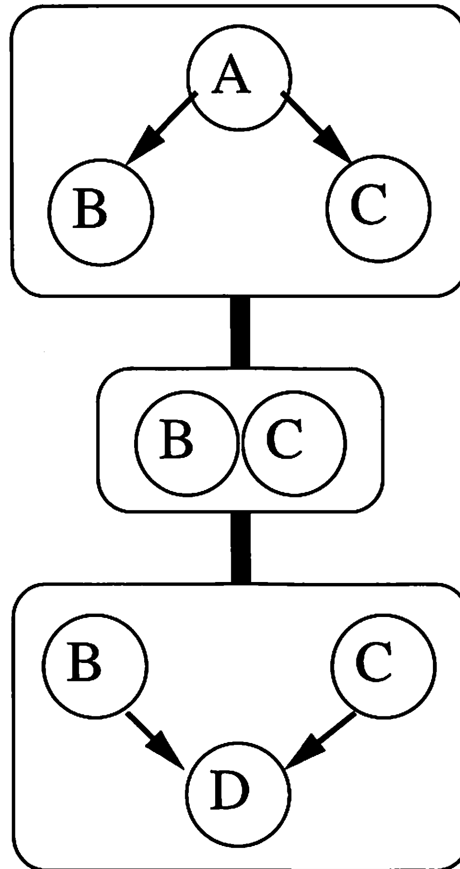


図 16: 接合木

クラスタリング法と接合木を用いた解法を比較した結果，本論文では接合木を用いた解法を使用した [1]．その理由は，クラスタリング法の最悪のケース*1にある．クラスタリング法では，最悪の場合ネットワークの大きさが指数的に増加する．本論文で扱う問題では変数に単語が相当するため，クラスタリング法の最悪のケースに該当し，計算を行うことが不可能になる．そのため，本論文では接合木を用いた．

*1 ノード B とノード C の取り得る値が多い場合，ノード B とノード C の全ての組み合わせは膨大な数になる．

4 実験

本論文では大きく分けて2つの実験を行った．SENSEVAL2の日本語翻訳タスクを用いたILPを用いた手法の評価と，SENSEVAL2の日本語辞書タスクを用いたBNを用いた手法の評価である．どちらの実験も背景知識を用いた評価を行う．

ILPを用いた手法の評価では，SENSEVAL2の日本語翻訳タスクを取り上げた．日本語翻訳タスクは，Translation Memory（TM）と呼ばれる日英対訳対が与えられ，テスト文中の該当単語を英訳する際に利用できるTMの例文番号を返すタスクである．これは英訳を語義と考えた場合の多義語の曖昧性解消問題となっており，分類問題の一種である．このタスクは訓練データを新たに作成することが困難という特異なタスクになっており，TMの例文だけ，つまり少量の訓練データからどのようにして精度の高い分類規則を獲得するかが解決の鍵であり，背景知識を用いるILPが有効であると考える．実験では，ILPの実装システムとしてMuggletonによるProgol，背景知識として分類語彙表を利用した．

BNを用いた手法の評価では，SENSEVAL2の日本語辞書タスクを取り上げた．日本語辞書タスクは単純な多義語の曖昧性解消問題である．訓練データがある程度提供されているために，従来の学習手法がよい成績を納めている．特にNaive Bayesを用いたシステムの成績がよい．実験では，Naive Bayesを拡張したモデルを使いBNによる学習を試みた．さらにそのBNに背景知識を用いる手法も試した．

4.1 SENSEVAL2

SENSEVAL2[9]は語義判別のコンテスト形式の国際会議であり，2001年7月にACL-01に併設されて開催された．SENSEVAL2の日本語タスクには辞書タスクと翻訳タスクの2つのタスクが設定されている．辞書タスクでは語の意味の区別（曖昧性）を国語辞典によって定義し，翻訳タスクではこれを訳語選択によって定義した．

4.2 ILP

本実験ではSENSEVAL2の翻訳タスクを扱う．翻訳タスクではTMと呼ばれる日英対訳データが予め与えられる．そして対象単語を含むテスト文が出題され，その対象単語を英訳する際に利用できるTMの例文番号を解答として返すタスクである．例文番号をクラスと考えれば，翻訳タスクは分類問題として定式化できる．またTMの例文が正例と背景知識の基となる．

TM の例文を，JUMAN（形態素解析）を用いて単語分割する．単語分割が，失敗または適切ではない部分は，手作業で修正する．ここで，単語の一部を素性（e1，e2，e3，e4）として抽出する．それとは別に，例文番号をクラスと考え，例文ごとにクラス付けを行う．例文番号，クラス，素性を述語に変換し，Progol の入力ファイルを生成する．入力ファイルを Progol で処理し規則を生成した．

テストは 40 単語が対象である．各単語ごとに 30 問，計 1200 問のテストとなる．それらに対して，Progol により生成された規則を使い，語義判別を行った．

実験結果，分類語彙表を使用しない場合の正解率は 48.67 %．分類語彙表を使用すると正解率は 54.00 % に向上した．分類語彙表を使用しない場合の正解率は，SENSEVAL2 において TM だけを用いて学習したシステムと，ほぼ同等の正解率である [9]．分類語彙表を使用すると正解率は 5.33 % 増加し，同等以上の正解率を得られた．背景知識が有効に利用され，より優れた規則が生成されたと言える．

Progol と確率的手法の 1 つである決定リストを直接比較してみる．論文 [12] では TM の例文だけを訓練データとして決定リストを作成し，54.00 % の精度を出している．ただしそこでは，例文をグループ化して，クラスの数を減らしている．実際に TM タスクでは例文番号をクラスとして扱うよりも，例文をグループ化して，同じグループの例文には同一のクラスを与えた方が精度が高くなる．

ここでも論文 [12] で利用したクラスと同一のクラスを用いて，Progol により学習させた結果，60.50 % の結果を得た．ただし，ここでは分類語彙表を用いていない．同一クラスで学習した場合でも，確率的手法に比べて高い正解率となった．これは，ILP が背景知識を用いずとも，十分に優れた規則を生成することができるためである．

表 4: ILP の実験結果

	分類語彙表を用いない (%)	分類語彙表を用いる (%)
ataeru	16.67	16.67
baai	3.33	3.33
chikaku	33.33	50.00
chushin	36.67	36.67
deru	33.33	33.33
egaku	33.33	33.33
hakaru	60.00	56.67
hana	66.67	70.00
hantai	80.00	93.33
ima	6.67	86.67
imi	40.00	56.67
ippan	33.33	53.33
ippou	63.33	53.33
iu	3.33	3.33
jidai	70.00	73.33
jigyou	50.00	50.00
kaku_n	80.00	73.33
kaku_v	96.67	96.67
kau	46.67	83.33
kiku	50.00	50.00

表 4: ILP の実験結果

	分類語彙表を用いない（%）	分類語彙表を用いる（%）
kiroku	26.67	26.67
koeru	96.67	86.67
kokunai	100.00	100.00
kotoba	70.00	93.33
mae	16.67	0.00
mamoru	3.33	3.33
matsu	86.67	86.67
miseru	73.33	93.33
mitomeru	23.33	23.33
mondai	53.33	53.33
motomeru	86.67	86.67
motsu	33.33	86.67
mune	23.33	26.67
noru	30.00	26.67
shimin	86.67	96.67
sugata	20.00	13.33
tsukau	70.00	66.67
tsukuru	63.33	36.67
tsutaeru	40.00	36.67
ukeru	40.00	43.33

4.3 BN

本手法の有効性を確認するために，SENSEVAL2 の日本語辞書タスクの課題に対して語義判別を試みた．

日本語辞書タスクは，単語の語義を岩波国語辞典の語義立てによって定義し，語の意味的曖昧性解消（Word Sense Disambiguation; WSD）の正確さを競うタスクである．テキストは毎日新聞の 1994 年の新聞記事を用いた．語義を決定する評価単語の数は 100 である．評価単語のそれぞれについて 100 語ずつ語義を決めるため，評価単語の数は 10,000 である．語義判別の対象単語と正解データを含んだ訓練データが予め与えられる．そして対象単語を含むテスト文が出題され，その際対象単語の語義を表す番号を返すタスクである．

SENSEVAL2 の日本語辞書タスクは，単純な語義判別問題である．対象単語は名詞 50 単語，動詞 50 単語の計 100 単語である．ラベル付きの訓練データは 1 単語平均して名詞は 177.4 事例，動詞は 172.7 事例用意されている．またテストデータは各単語に対して 100 問のテストデータが用意されている．つまり，名詞に対しては計 5000 問，動詞に対しても計 5000 問のテストが行える．ただし，

SENSEVAL2 の日本語辞書タスクは，単純な語義判別問題である．対象単語は名詞 50 単語，動詞 50 単語の計 100 単語である．これら 100 単語は語義の頻度分布のエントロピーを考慮して選定されており，語義判別が容易なものから困難なものまでバランス良く選定されている．ラベル付きの訓練データは 1 単語平均して名詞は 177.4 事例，動詞は 172.7 事例用意されている．またテストデータは各単語に対して 100 問のテストデータが用意されている．つまり，名詞に対しては計 5000 問，動詞に対しても計 5000 問のテストが行える．ただし，ラベルなし訓練データは SENSEVAL2 から提供されていない．

ラベルなし訓練データとして通常のテキストが使えそうだが，実際は制限がある．これは通常のテキストが使えそうだが，実際は制限がある．それはラベル付きの訓練データを作成した際に用いた単語辞書や品詞分類を合わせる必要があるからである．そのためここでは RWC テキストデータベース第 2 版に納められた毎日新聞 95 年度版の 1 年分の記事を利用して，ラベルなし訓練データを収集した．このデータはラベル付き訓練データのもとになったデータであり，同一の形態素解析システムを用いて形態素解析されている．収集できたラベルなし訓練データの数は 1 単語平均して名詞は 7585.5 事例，動詞は 6571.9 事例である．

Naive Bayes モデルの正解率は，名詞 75.85 %，動詞 76.77 %．Belief Network モデルの

正解率は，名詞 76.00 %，動詞 76.85 %．また Belief Network モデルに背景知識を適用させたときの正解率は，名詞 75.74 %，動詞 76.77 % であった．

表 5: BN の実験結果（名詞）

	Naive Bayes(%)	Belief Network(%)	Belief Network(use unlabeled)(%)
aida	82	82	85
atama	66	67	61
ippan	89.3332	89.3332	89.6666
ippou	82	83	84
ima	88	87	86
imi	53	55	55
utagai	100	100	100
otoko	92	92	92
kaihatsu	59	59	59
kaku_n	74	76.5	74
kankei	87	87	86
kimochi	65	65	65
kiroku	62	62	63
gijutsu	92	92	93
genzai	97	97	98
koushou	100	100	100
kokunai	51	50	53
kotoba	47	48	47
kodomo	67	68	64
gogo	90	91	81.5
shijo	70	70	70
shimin	60	61	60
shakai	82	82	82
shonen	93	93	92
jikan	53	53	53
jigyou	66	66	65

表 5: BN の実験結果（名詞）

	Naive Bayes(%)	Belief Network(%)	Belief Network(use unlabeled)(%)
jidai	74	72	73
jibun	92	92	92
joho	76	75	75
sugata	52	53	51.3333
seishin	69	70	69
taishou	92	92	92
daihyou	84	84	86.5
chikaku	78	80	80
chihou	58	58	58
chushin	97	97	97
te	49	51	51
teido	99	99	99
denwa	84	85	84
doujitsu	70	69	68
hana	99	99	98
hantai	98	98	98
baai	86	86	86
mae	86	85	87
minkan	100	100	100
musume	86	83	85
mune	58.5	58.5	59.25
me	12	12	14
mono	31	31	30
mondai	95	95	96
合計	75.85	76.00	75.74

表 6: BN の実験結果（動詞）

	Naive Bayes(%)	Belief Network(%)	Belief Network(use unlabeled)(%)
ataeru	64	64	61
iu	93	93	93
ukeru	53	52	53
uttaeru	83	81	80
umareru	65	65	65
egaku	56	55	57
omou	90	90	90
kau	86	86	86
kakaru	58	59	59
kaku_v	64	66	66
kawaru	92	92	92
kangaeru	99	99	99
kiku	63	62	62
kimaru	95	95	95
kimeru	92	93	91
kuru	83	83	83
kuwaeru	88	88	87
koeru	74	75	75
shiru	95	95	95
susumu	47	47	47
susumeru	97	97	97
dasu	32	32	34
chigau	99	99	99
tsukau	98	98	96
tsukuru	59	61	61

表 6: BN の実験結果（動詞）

	Naive Bayes(%)	Belief Network(%)	Belief Network(use unlabeled)(%)
tsutaeru	77	77	74
dekiru	79	79	79
deru	56	56	58
tou	70	70	67
toru	32	33	34
nerau	97	97	97
nokosu	80	80	80
noru	60	60	62
hairu	36	38	38
hakaru	92	92	93
hanasu	100	100	100
hiraku	85	86	87
fukumu	93	93	95
matsu	52	52	52
matomeru	77	77	76
mamoru	67.5	67.5	67.5
miseru	96	96	95
mitomeru	89	89	89
miru	73	72	73
mukaeru	90	90	90
motsu	52	52	49
motomeru	85	85	85
yomu	89	89	89
yoru	97	97	97
wakaru	89	88	89
合計	76.77	76.85	76.77

5 考察

5.1 ILP

5.1.1 ILPと確率的手法

Progolと決定リストによる結果を比較した．結果はProgolの方が優れていた．しかし，確率的手法には決定リストよりも優れた結果を出すアルゴリズムが存在する．そのため，ILPを使った手法が明かに優れているとは言えない．ただし今回の実験のように訓練データが少ない場合には，訓練データにない情報，つまり背景知識の利用方法の優劣が重要になってくる．そのような問題にはILPが有利になると思われる．

5.1.2 背景知識の悪影響

論文 [12] と同一のクラスを用いて，今度は分類語彙表を使用したところ，正解率が60.50％から58.92％へと1.58％悪くなった．背景知識を用いることで，生成される規則が悪くなっているためである．しかし，大きく正解率が下っているわけではない．

この実験で利用したクラスは，ほぼ最適にグルーピングした結果得られたものであることと，分類語彙表を使用する前の高い正解率を考慮すると，分類語彙表の使用前でも，十分に優れた規則が生成されていたと考えられる．そのため，分類語彙表がノイズとなり，規則生成に悪い影響を与えたと考えられる．

具体的には，背景知識を用いた場合，通常規則の数が減少していた．表7に生成された規則の詳細を示す．*2

*2 過度に専門化された規則は規則が限定的でありすぎるために事例の分類を行うことができない規則，デフォルト規則は全ての事例を分類することができる規則のことをいう．また，通常の規則は，全体の規則から，過度に専門化された規則とデフォルト規則を除いた規則である．

表 7: 生成された規則数の詳細

	背景知識あり	背景知識なし
生成された規則全体の数	466	478
過度に専門化された規則の数	22	104
デフォルト規則の数	99	199
通常の規則の数	345	175

背景知識を用いた場合と用いていない場合とでは，生成された全体の規則数はそれほど変化していない．しかし，過度に専門化された規則と，デフォルトルールの数が増加している．そのため，通常の規則の数が減少している．事例の判断に用いることができる規則が減少していると解釈することができる[*3]．また，後述するようにデフォルト規則には問題があり，デフォルト規則の数が増加していることが正解率に良い影響を与えるわけでない．

通常，背景知識は有益な情報である．しかし，システムが背景知識を有効に扱えない場合，背景知識はノイズとなり，正解率に悪影響を与える．背景知識を有効に利用し，悪影響を最小限に防ぐことが今後の課題である．

5.1.3 規則の優先順位について

Progol が生成する規則にはいくつか特徴がある．

- デフォルト規則は 1 つあればよいにもかかわらず，複数生成されることがある．
- デフォルト規則が生成されないことがある．
- 規則は入力ファイルで述語が記述された順番に上から生成され，生成された順番のまま出力される．そのため，生成された規則には優先順位がない．

全ての事例に適用することができる規則をデフォルト規則と言い，その他のどの規則にも適用されない事例に適用される．

通常，規則に優先順位が付けらているか，優先順位で並べられていた方がよい．しかし，Progol には規則の優先順位がないため，どの規則が優先されるべき規則なのかがわか

[*3] 通常の規則の数が減少しているということは，より抽象化されたルールが生成されているように思えるが，全体の規則の数は変化していないのでそうは言えない．

らない．特に，デフォルト規則は 1 つしか用いられないにもかかわらず，デフォルト規則が複数生成される．そのため，どのデフォルト規則を用いるべきかが大きな問題になる．(図 17)

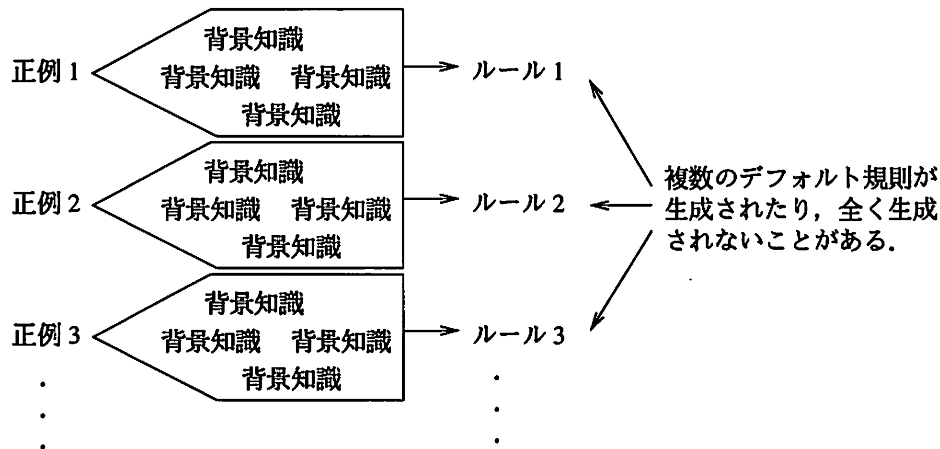


図 17: デフォルト規則の生成

上記の Progol が生成するデフォルト規則の問題に対処するためには，別の手法と組み合わせることが必要になると考えられる．その方法の一つとして確率的な手法が提案されている．生成された規則が確率を含むようになれば，優先度の問題が解決される．優先度の問題が解決されれば，デフォルト複数の規則が生成されても問題はない．また，確率を用いてデフォルト規則を生成するようにすれば，デフォルト規則が生成されないという問題を解決することができる．

今後，確率的手法を取り入れた PRISM， SLP といった実装システムを用いることで，これらの問題に対応したい [16].

5.2 BN

Naive Bayes モデルの正解率は，名詞 75.85 %，動詞 76.77 %．Belief Network モデルの正解率は，名詞 76.00 %，動詞 76.85 %．名詞，動詞共に Naive Bayes モデルを適用させた場合に比べ Belief Network モデルを適用させた方が正解率が高かった．Naive Bayes モデルに，素性同士の関係を追加することでより適切なモデルを構築することができたと考えられる．

しかし Belief Network モデルに背景知識を適用させたときの正解率は，名詞 75.74 %，

動詞 76.77 % であった．背景知識を利用した場合，動詞は Naive Bayes モデルと同じ正解率にまで下り，名詞は Naive Bayes モデルよりも正解率が下った．背景知識を利用した場合は期待した効果が得られなかった．

背景知識が上手く利用されなかった原因を考察する．

未知の値を証拠として与えられたときの条件付き確率値の値，つまり事後確率を適切に設定することができなったため，背景知識を上手く利用することができなかったと考えられる．

本実験では Belief Network を扱う手法として，接合木（Junction Tree，Join Tree）を用いた [1]．本実験で用いたモデルは，変数から変数への経路が複数あるため通常の手法で計算することができない．そこで，Belief Network から接合木という木構造を構築して，その木構造を利用して計算する手法を用いた．

接合木は 3 つの変数を一つのクラスタ（cluster, clique）として扱い，クラスタの集合が木構造を成している．計算の際は，クラスタ単位で条件付き確率から，証拠をもとに事後確率を求める．その後，メッセージ伝播により木構造全体に事後確率を伝える．証拠とは変数に与えられる値のことで，変数から変数への条件付き確率に証拠が与えられると，変数から変数への事後確率が決定する．

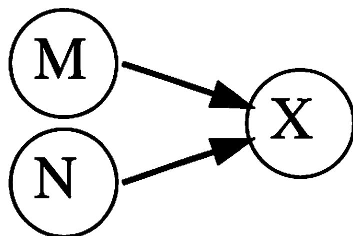


図 18: $M \rightarrow X$ と $N \rightarrow X$ で構成されるクラスタ

接合木ではクラスタ単位で事後確率を求めるが，図 18 のクラスタにおいて変数 X に証拠 E_X，M に証拠 E_M，N に証拠 E_N が与えられて全ての証拠が満されたとき，$M(E_M) \rightarrow X(E_X)$ の事後確率が決定するが，$N(E_N) \rightarrow X(E_X)$ の事後確率がわからない場合がある[*4]．この場合，クラスタの事後確率は 0.0 % になり，$M(E_M) \rightarrow X(E_X)$ の条件付き確率は無視されたことになる[*5]．これは，このクラスタにおいて証拠 E_X，E_M，E_N が与

[*4] 本実験のアルゴリズムは基本的に文献 [1] の通りであるが，本実験と異なり，文献 [1] では変数の中身を True または False であるとしているため，本実験で問題となった現象が文献 [1] では起らない．

[*5] $N(E_N) \rightarrow X(E_X)$ の証拠が満されない（N の証拠が満されない）ように振舞う方法も試みたが本実験の結

えられなかった場合と等しい結果である[6]．これは証拠を与えられずに結果を導きだすことと等しい．

この現象は背景知識が与えられたとき頻繁に起きている．Naive Bayes モデルの場合は，名詞 91 回，動詞 138 回．拡張モデルの場合は，名詞 107 回，動詞 147 回．拡張モデルに背景知識を用いた場合は，名詞 166 回，動詞 174 回．拡張モデルと背景知識を用いた場合を比較すると，名詞は発生回数が 59 回増えて，動詞は発生回数が 27 回増えている．

背景知識を用いたときにこの問題が頻発する理由は次の通りである．背景知識を用いると変数が知っている値を増やすことができる．そのため，変数が証拠を知っている回数は増え，図 18 における変数 M, N, X が証拠が満す回数は増える．しかし，$M(E_M) \rightarrow X(E_X)$ の事後確率が存在し，$N(E_N) \rightarrow X(E_X)$ の事後確率が存在する回数はあまり増加していない．このため，前述の現象が発生する回数が増え，結果として証拠が生かされない回数が増える．つまり，背景知識を用いることで，証拠が生かされない回数が増え，その影響で正解率が下った．

果よりも更に悪い結果であった．

[6] 本実験では，条件付き確率が 0.0 % になったクラスタの条件付き確率が他のクラスタに伝播しないように対策を行っている

6 結論

6.1 ILP

本論文ではILPの実装システムとしてProgolを利用し，SENSEVAL2の日本語TMタスクにおける語義の多義性解消問題にILPの手法を適用した．背景知識としては分類語彙表を利用した．実験の結果，ILPが確率統計的な手法と同等以上の精度の規則を学習できた．また分類語彙表を用いることでより精度の高い規則を学習できた．

しかし，背景知識を有効に利用することができない場合もあった．今回は条件によって背景知識が悪影響をおよぼしたが，今後はさまざま条件で背景知識を有効に利用できる方法を探りたい．また，確率的手法を取り入れたPRISM，SLPといった実装システムを用いることで，規則生成の優先順位に関する問題に対応したい．

6.2 Belief Network

本論文ではBelief Networkを用いた語義判別規則の学習を試みた．

実験で利用した語義判別問題は，SENSEVAL2の日本語辞書タスクであった．Naive Bayesモデルを適用させた場合に比べBelief Networkモデルを適用させた場合に正解率を高めることができた．しかし，Belief Networkモデルに背景知識を適用させたところ，動詞はNaive Bayesモデルと同じ正解率にまで下り，名詞はNaive Bayesモデルよりも正解率が下った．背景知識を利用した場合は期待した効果が得られなかった．

背景知識が上手く利用されなかった原因として，未知の値を証拠として与えられたとき，事後確率を適切に設定することができなったためだと考えた．一般には未知の値を証拠として与えられた場合，事後確率に小さな確率を設定する．しかし，適切な小さな確率を決定することは難しい．本実験ではこの小さな確率を0.0％としているが，実験の結果から推測するとこの値は不適切であった．だが，本実験で用いた0.0％という値よりも最適な値が存在する保証もない[*7]．また，事前に最適な小さな確率の求めることは難しいと考えられる．

仮に小さな確率を0.001％とした場合，既知の値にもかかわらず未知の値の確率である0.001％よりも小さな確率を取る場合もありうる．この場合，小さな確率と定めた0.001％は小さな確率としては間違っていると考えられる．そこで，小さな確率が既知の

[*7] 小さな確率を0.001％として実験を行っているが，0.0％の場合よりも低い正解率結果だった．

確率の中で一番小さな確率とする．ここで注意しなければいけないことは，ここで言う確率とは条件付き確率のことである．つまり，未知の値とは，事前条件が未知の場合と事後条件が未知の場合とその両方の値が未知の場合を表している．今回の実験結果から確実な結論を出すことはできないが，本実験の反省としていかに証拠を生かすかということが導きだされたことを考慮すると，事前条件または事後条件が未知の値の場合，小さな確率として同じ値を割り当てることは正しくないと言える．

小さな確率を用いない方法を考えると，変数が取り得る値を分類し，分類にたいして確率を設定する方法が考えられる．これに対して，未知の値が与えられた場合，その値の分類を求めることで確率を求めることができるため，未知の値に確率を与えるという困難な作業を行わなくてもよい．また，本論文では，語義の多義語の曖昧性解消問題を分類問題に定式化することで，BN を用いるた．つまり BN は分類問題を解くことできるので，変数の分類にも BN を用いることができる．本論文では素性からクラスを求めたが，BN のアルゴリズムを考えればある素性から別の素性の値を求めることもできる．つまり，同じネットワークを用いて変数の値を決定することができる[*8]．

今後は，この問題を含めて背景知識を有効に活用することが課題である．

[*8] しかし，この方法は変数の値を知るには変数の分類を行う必要があり，再帰的な関係になる．しかし，他手法との組み合わせで解決できる問題である．

謝辞

本研究の遂行及び論文の作成において多大な御助言及び御指導を賜りました新納浩幸教官に深い感謝の意を表します．

またその他様々な助言を頂いた主査の提泰行教官に深い感謝の意を表します．

研究室でお世話になった高橋篤史氏（修士），紺野憲一氏（学部），山村一起氏（学部），藤枝雅一氏（学部），その他研究の機会を与えれくれた人々に深く感謝致します．

参考文献

[1] Cecil Huang, Adnan Darwiche.: Inference in Belief Networks: A Procedural Guide, *International Journal of Approximate Reasoning*, 11:1-158 (1994).

[2] Muggleton, S.: Inductive logic programming, *New Generation Computing*, 8, pp.295-318 (1991).

[3] Muggleton, S.: Inverse Entailment and Progol, *New Generation Computing*, 13, pp.245-286 (1995).

[4] Quinlan, J. R.: Learning Logical Definitions from Relations, *Machine Learning*, 5, pp.239-266 (1990).

[5] Quinlan, J. R. and Camereon-Jones, R. M.: Induction of Logic Programs: FOIL and Related Systems, *New Generation Computing*, Vol.13, pp.287-312 (1995).

[6] Srinivasan, A., Muggleton, S. H., King, R. D. and Sternberg, M. J. E.(1994) Mutagenesis: ILP experiments in a non-determinate biological domain. In Proceedings of the Fourth International Workshop on Inductive Logic Programming.

[7] 阿部修也，新納浩幸：Belief Network を用いた語義判別規則の学習，言語処理学会第9回年次大会，to appear (2003).

[8] 阿部修也，新納浩幸：帰納論理プログラミングを用いた語義判別規則の学習，言語処理学会第8回年次大会，pp.667 − 670 (2002)

[9] 黒橋禎夫，白井清昭：SENSEVAL-2 日本語タスク，信学技報，Vol101, No.351, pp1-8 (2001).

[10] 佐藤泰介，櫻井彰人：特集「ベイジアンネット」，人工知能学会誌，17 巻，5 号，pp538-565 (2002).

[11] 嶋津恵子，古川康一：データベースからの知識発見システム DM-Amp – 設計と実装とエキスパートシステム開発への応用，人工知能学会誌，Vol.15, No.4, 論文特集：「発見科学」，pp.629-637 (2000).

[12] 新納浩幸：SENSEVAL-2 日本語翻訳タスクに向けて作成した語義判別学習システム Ibaraki，信学技報，Vol101, No.351, pp25-30 (2001).

[13] 新納浩幸，阿部修也：日本語翻訳タスクへの帰納論理プログラミングの適用，言語処理学会，Vol.10, No.3, to appear (2003).

[14] 新納浩幸，佐々木稔：EM アルゴリズムの最適ループ回数の予測を用いた語義判別規則の教師なし学習，情報処理学会自然言語処理研究会，151-8, pp.51-58 (2002).

[15] 古川康一：帰納論理プログラミング ー チュートリアルー，人工知能学会誌， Vol.12, No.5, 小特集：「帰納論理プログラミング」，pp655-664 (1997).

[16] 古川康一，尾崎知伸，植野研：帰納論理プログラミング，共立出版 (2001).

[17] 古川康一監訳：エージェントアプローチ人工知能, 共立出版 (1997).

付録A　データ

表5と表6のデータをグラフ化したものを添付する．

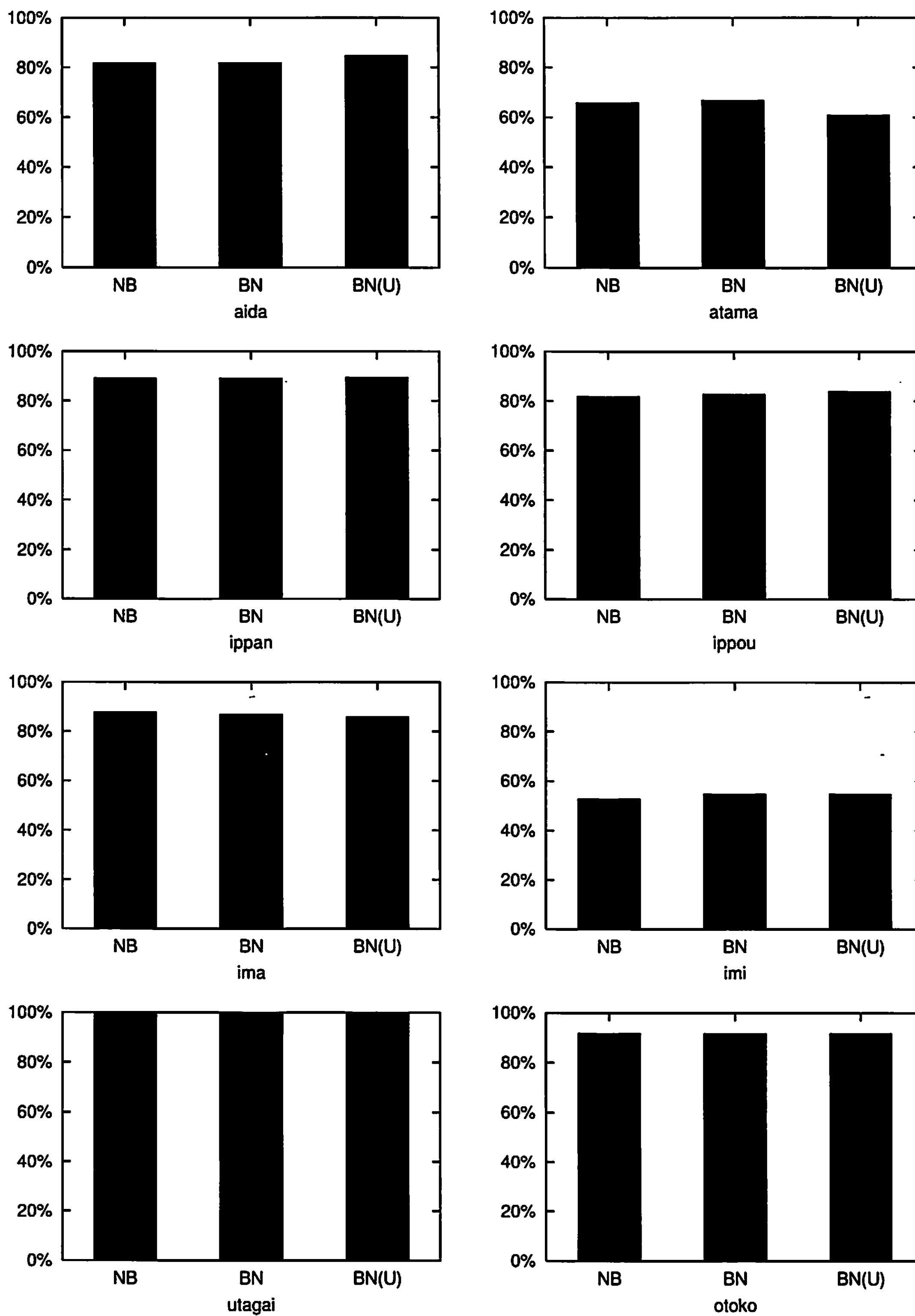


図 19: 名詞 その 1

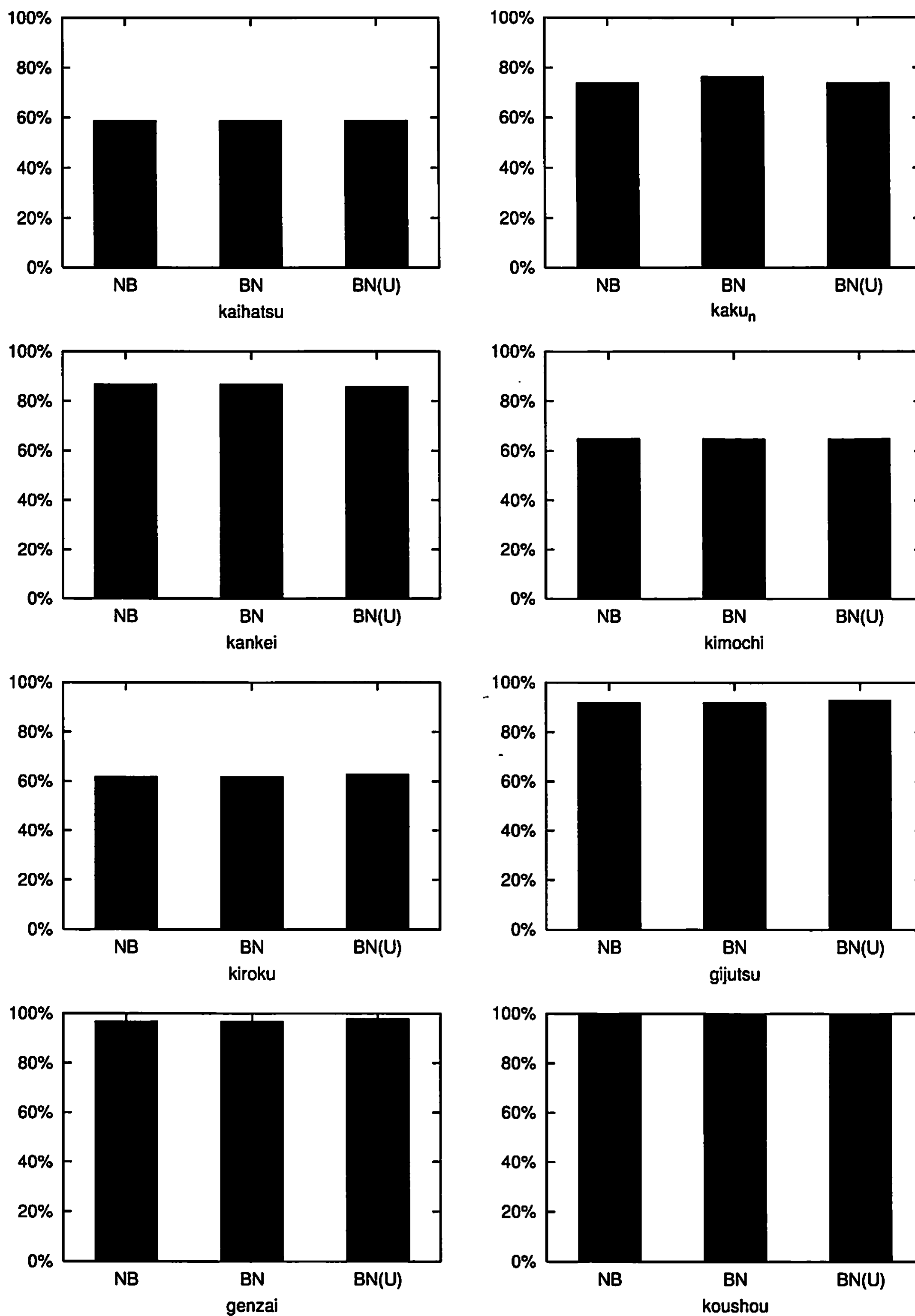


図 20: 名詞 その 2

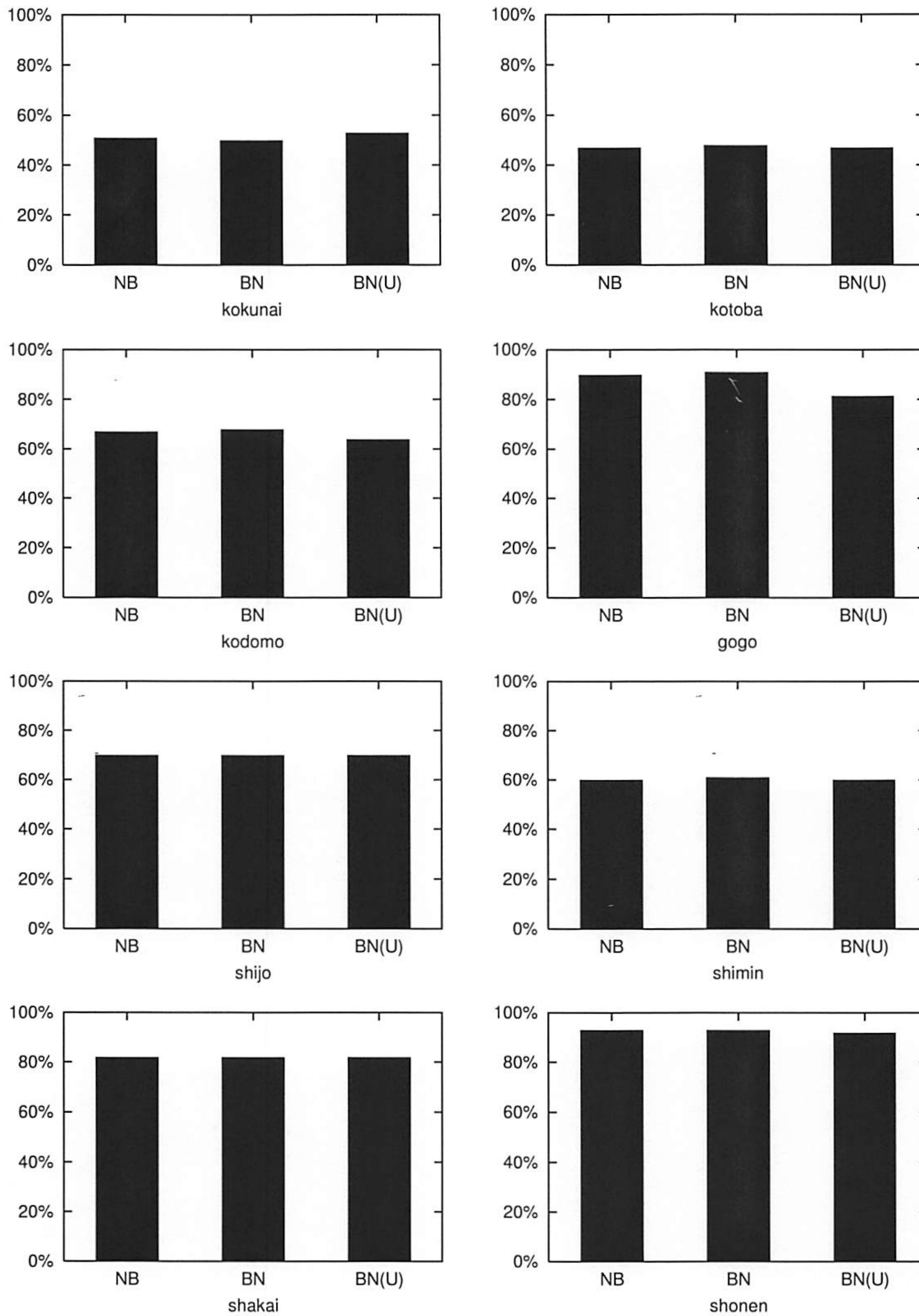


図 21: 名詞 その 3

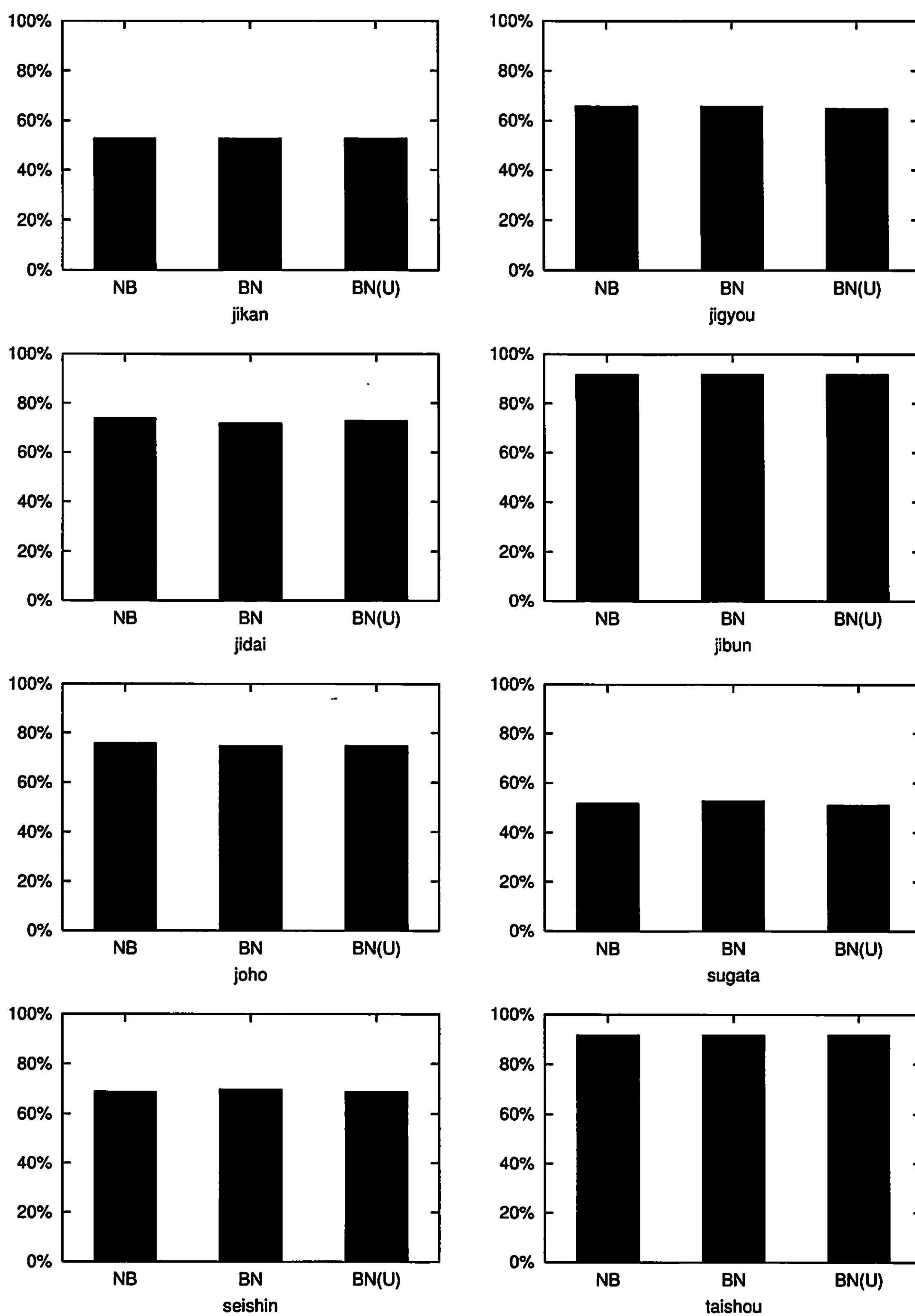


図 22: 名詞 その 4

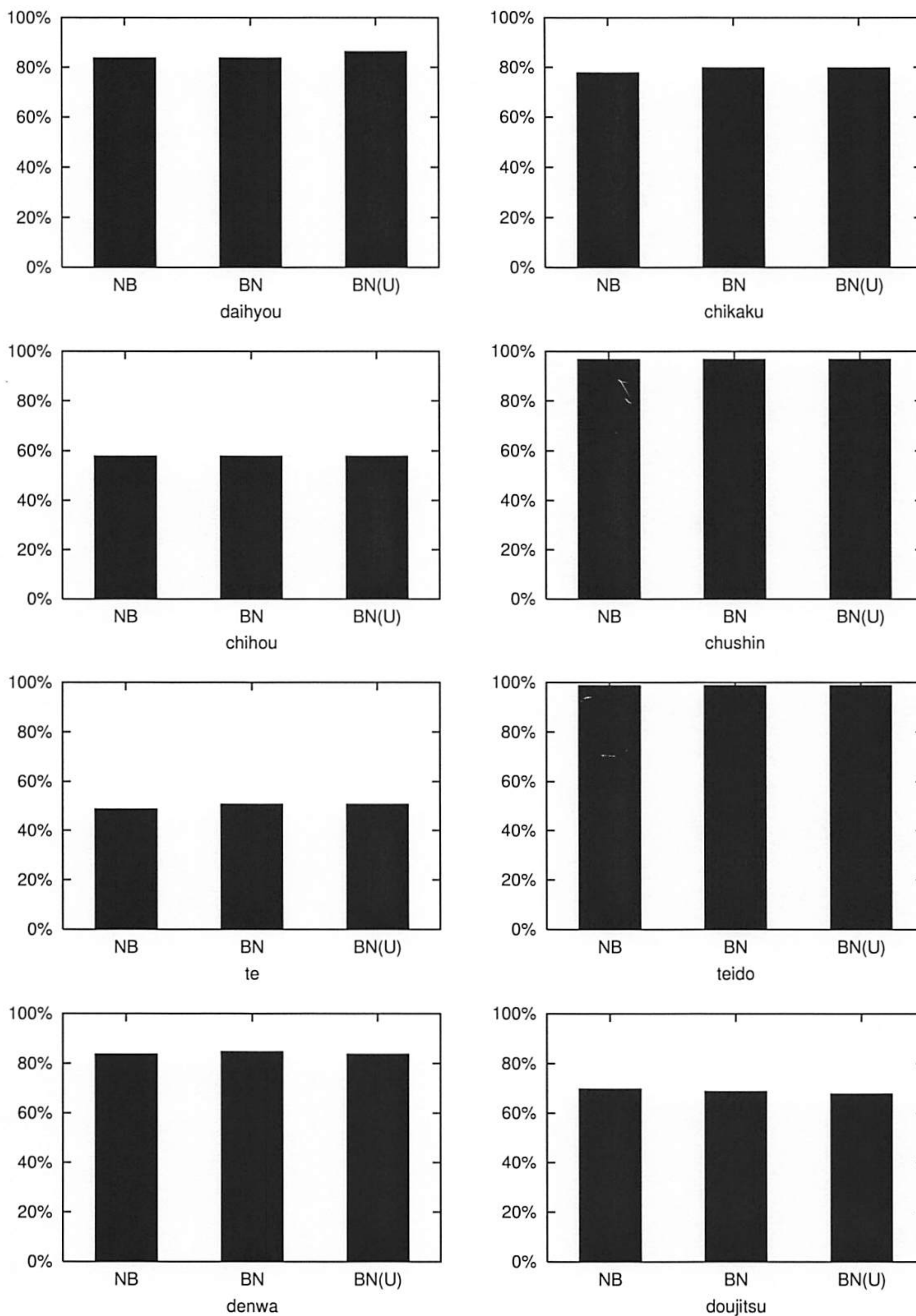
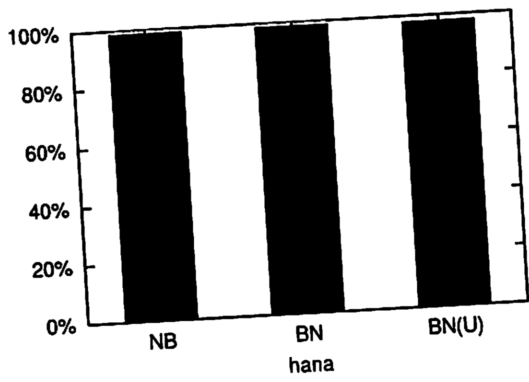
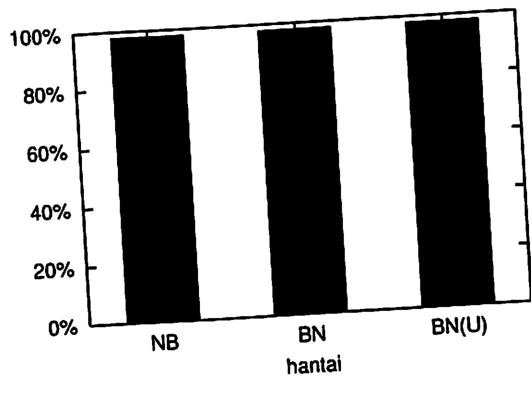
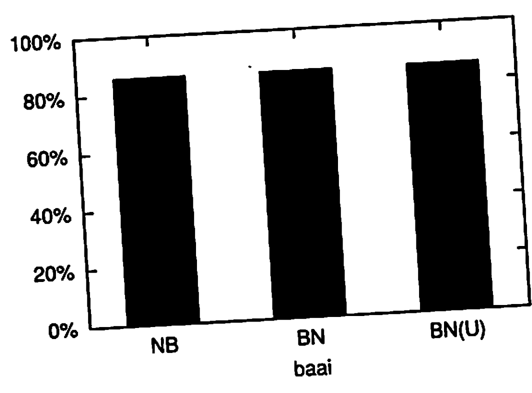
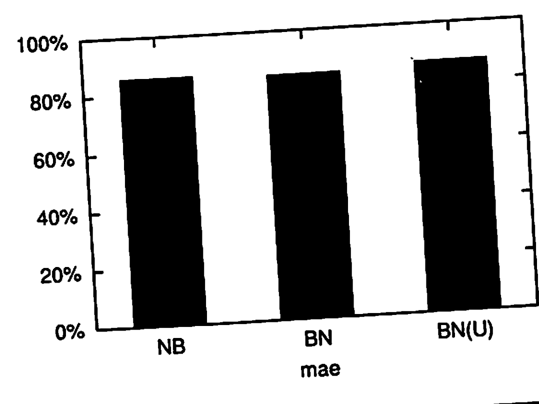


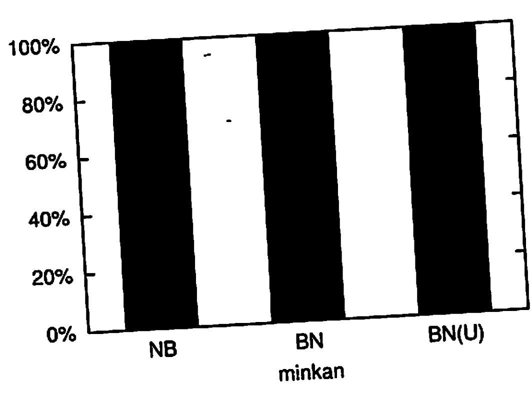
図 23: 名詞 その 5

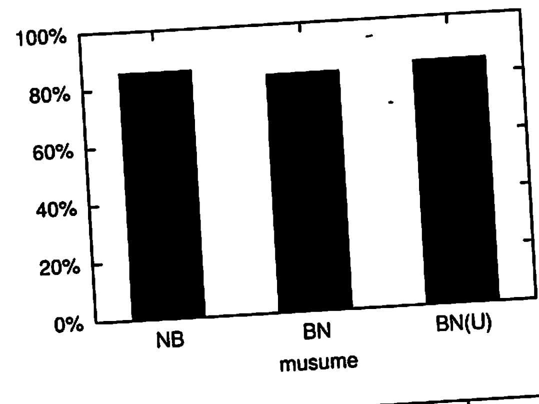


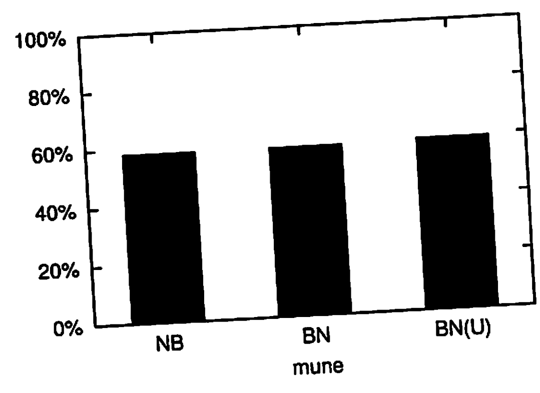












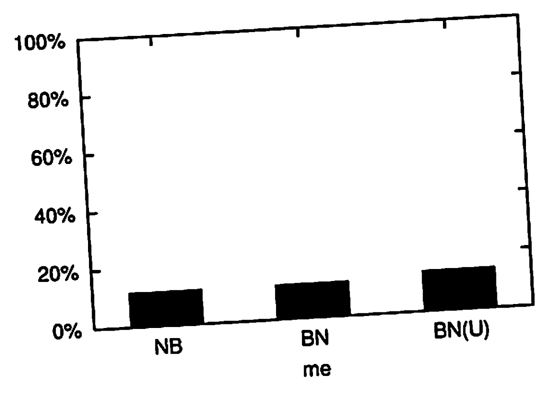
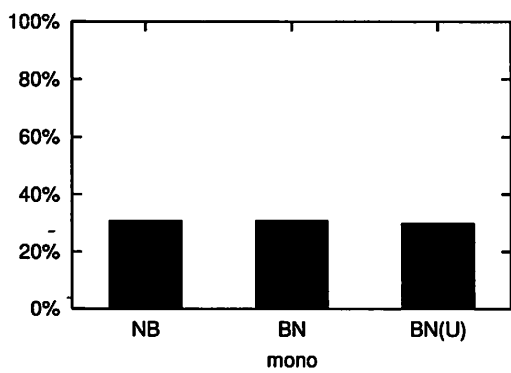


図 24: 名詞 その 6



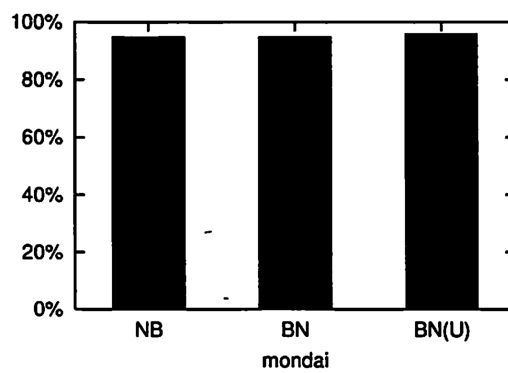


図 25: 名詞 その 7

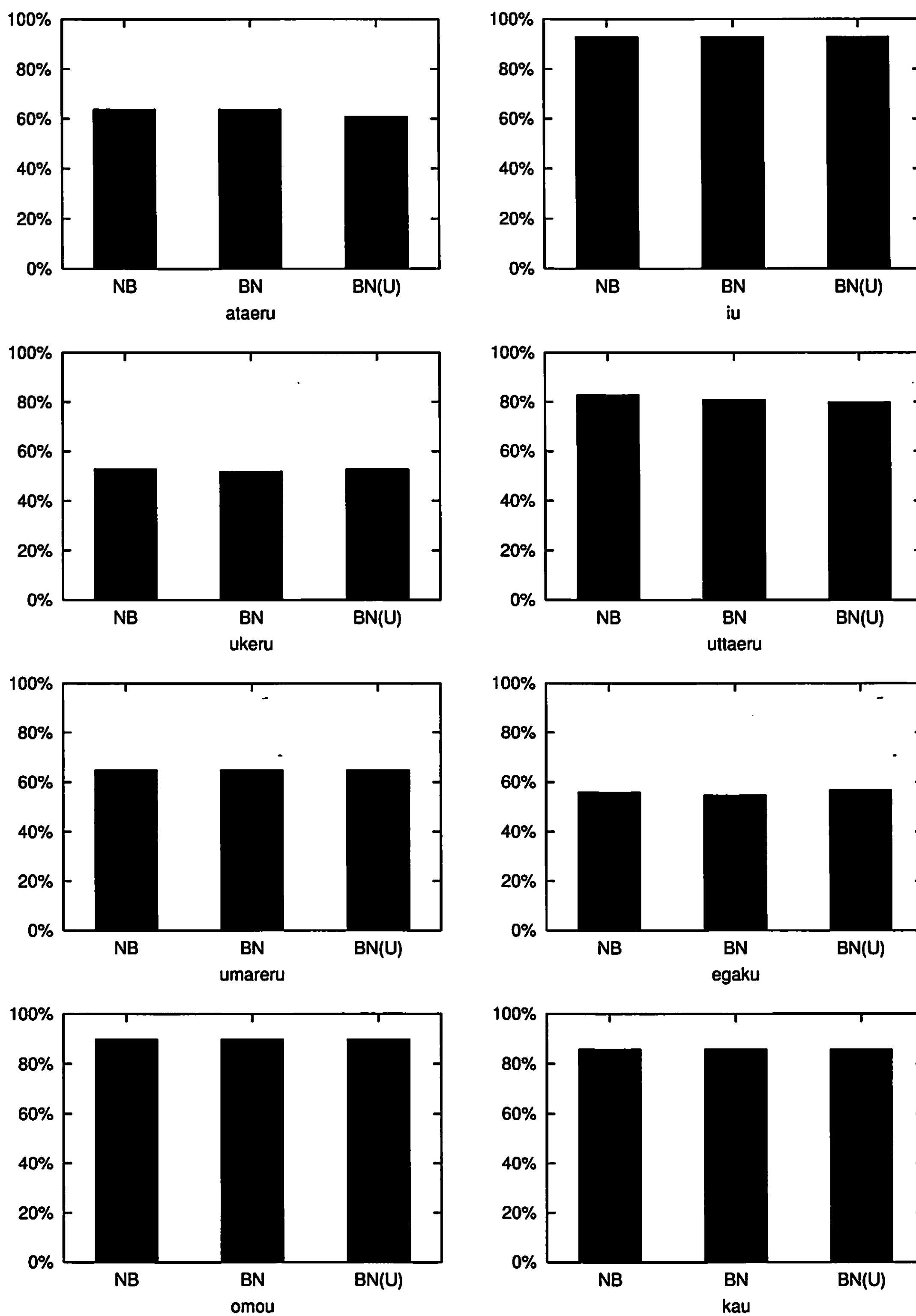
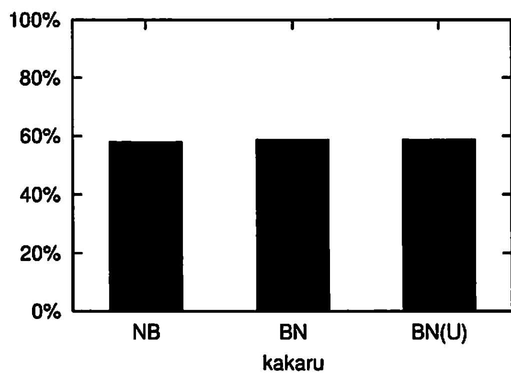
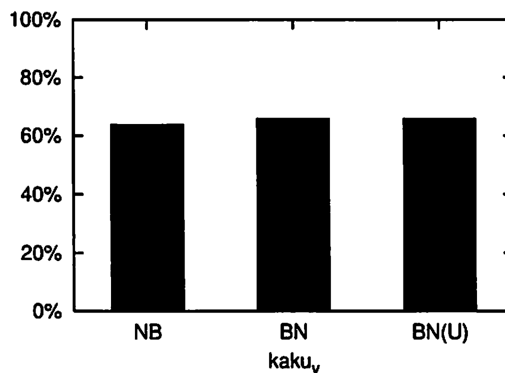
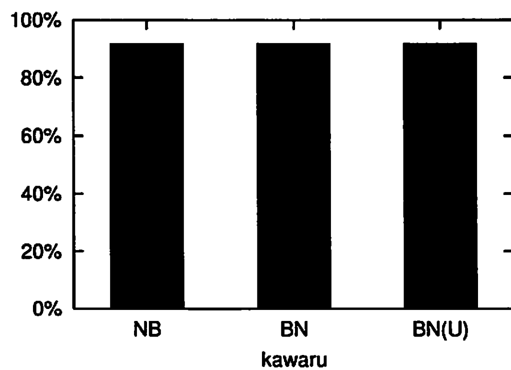
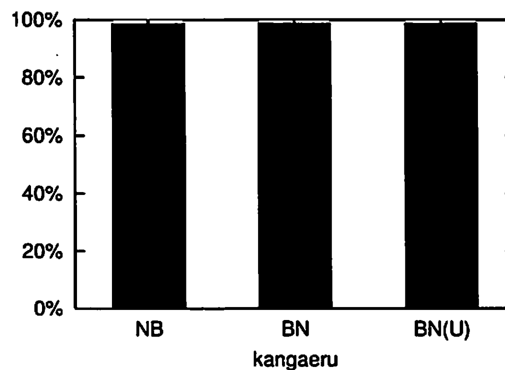


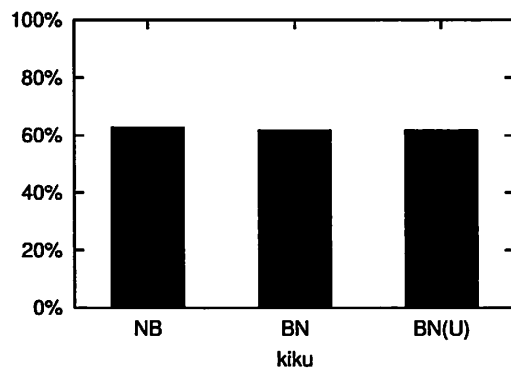
図 26: 動詞 その 1

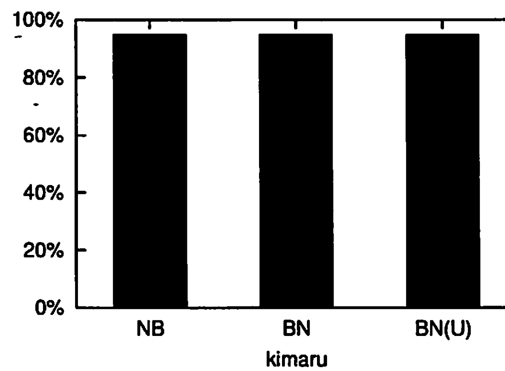


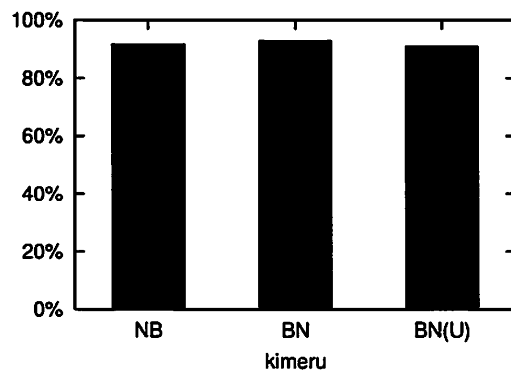












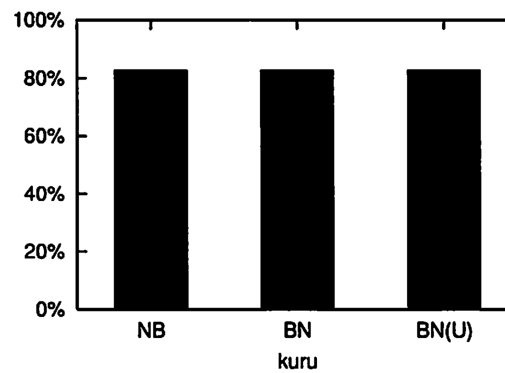
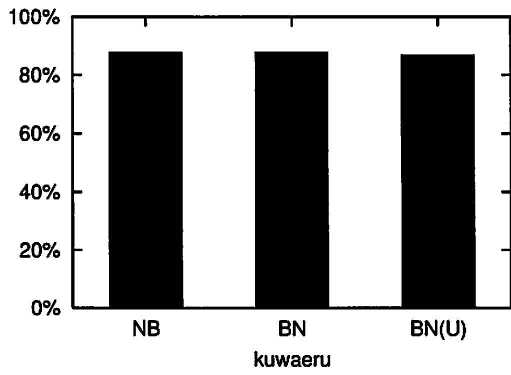
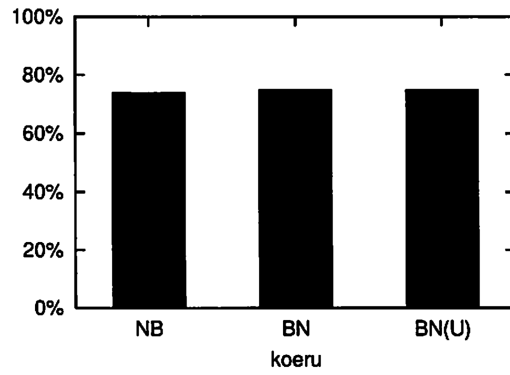
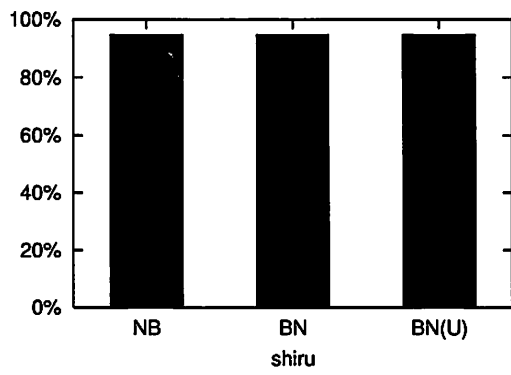
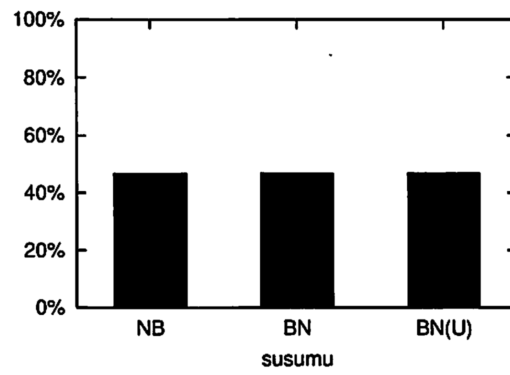


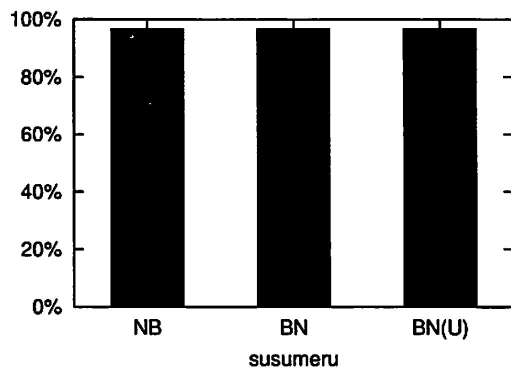
図 27: 動詞 その 2

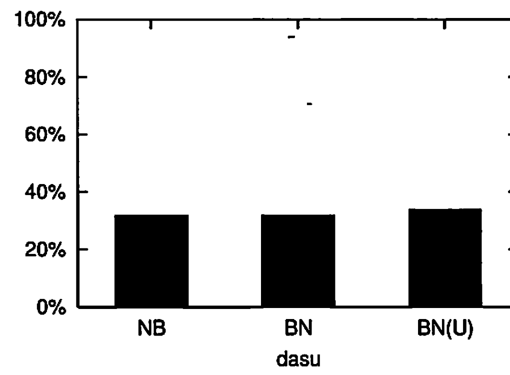


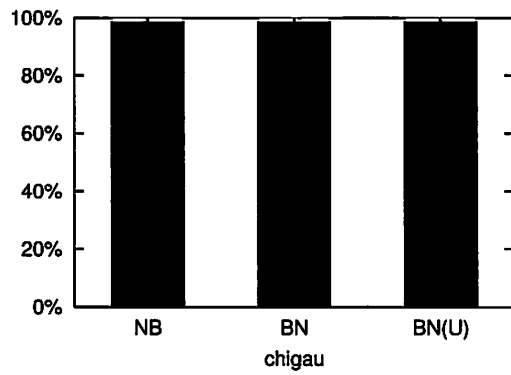












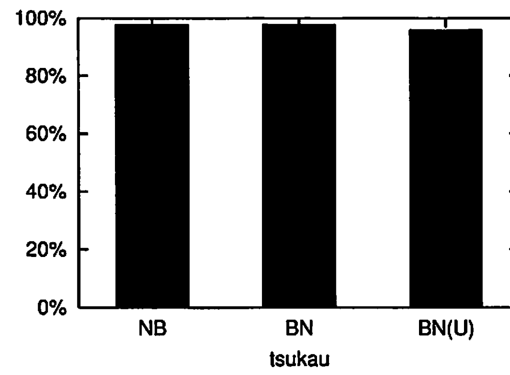
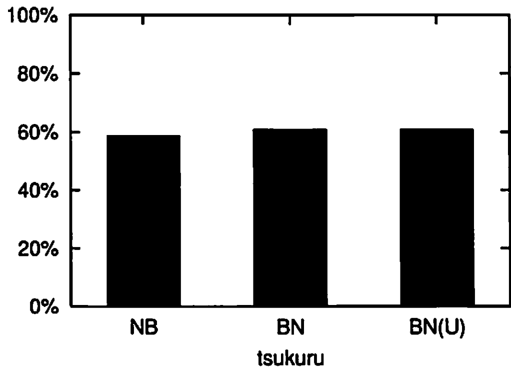
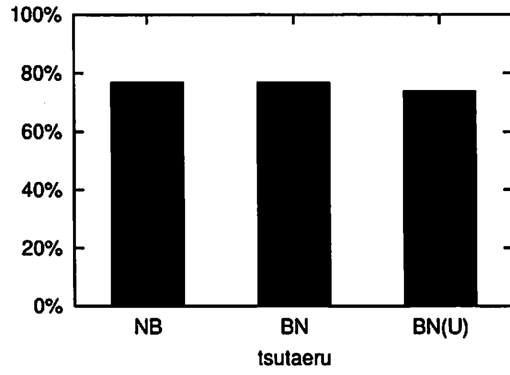
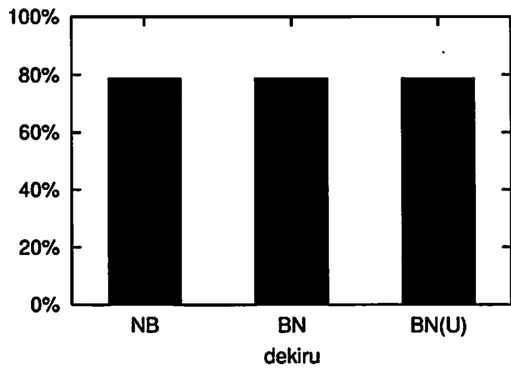
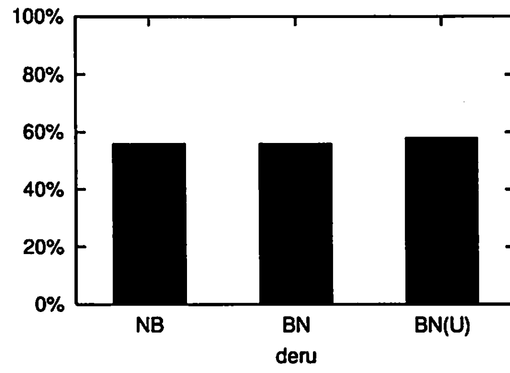


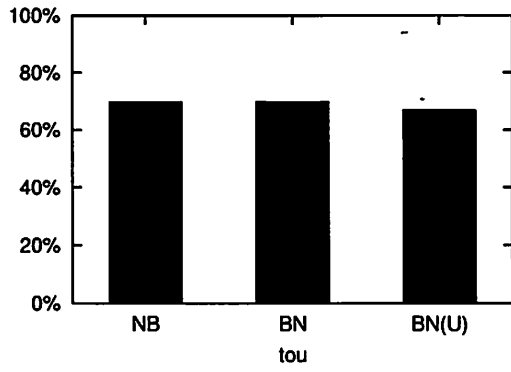
図 28: 動詞 その 3

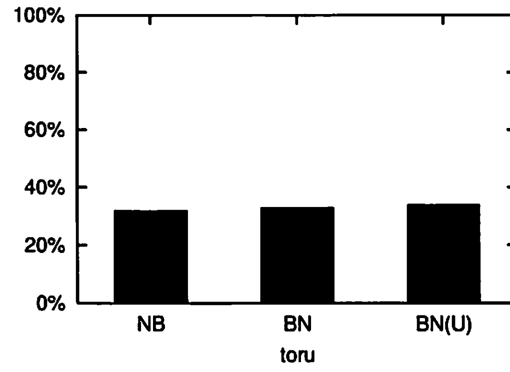


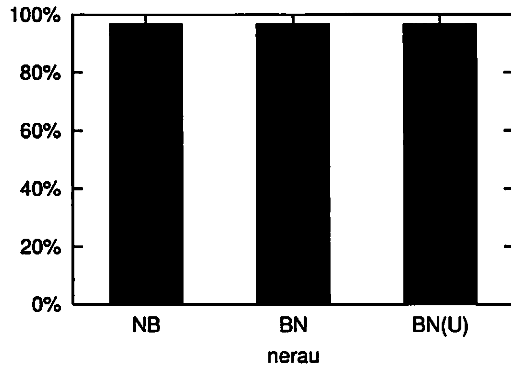












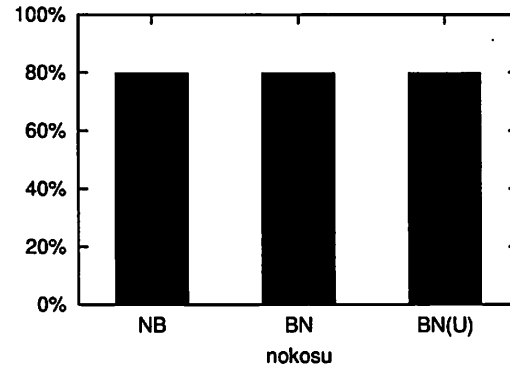


図 29: 動詞 その 4

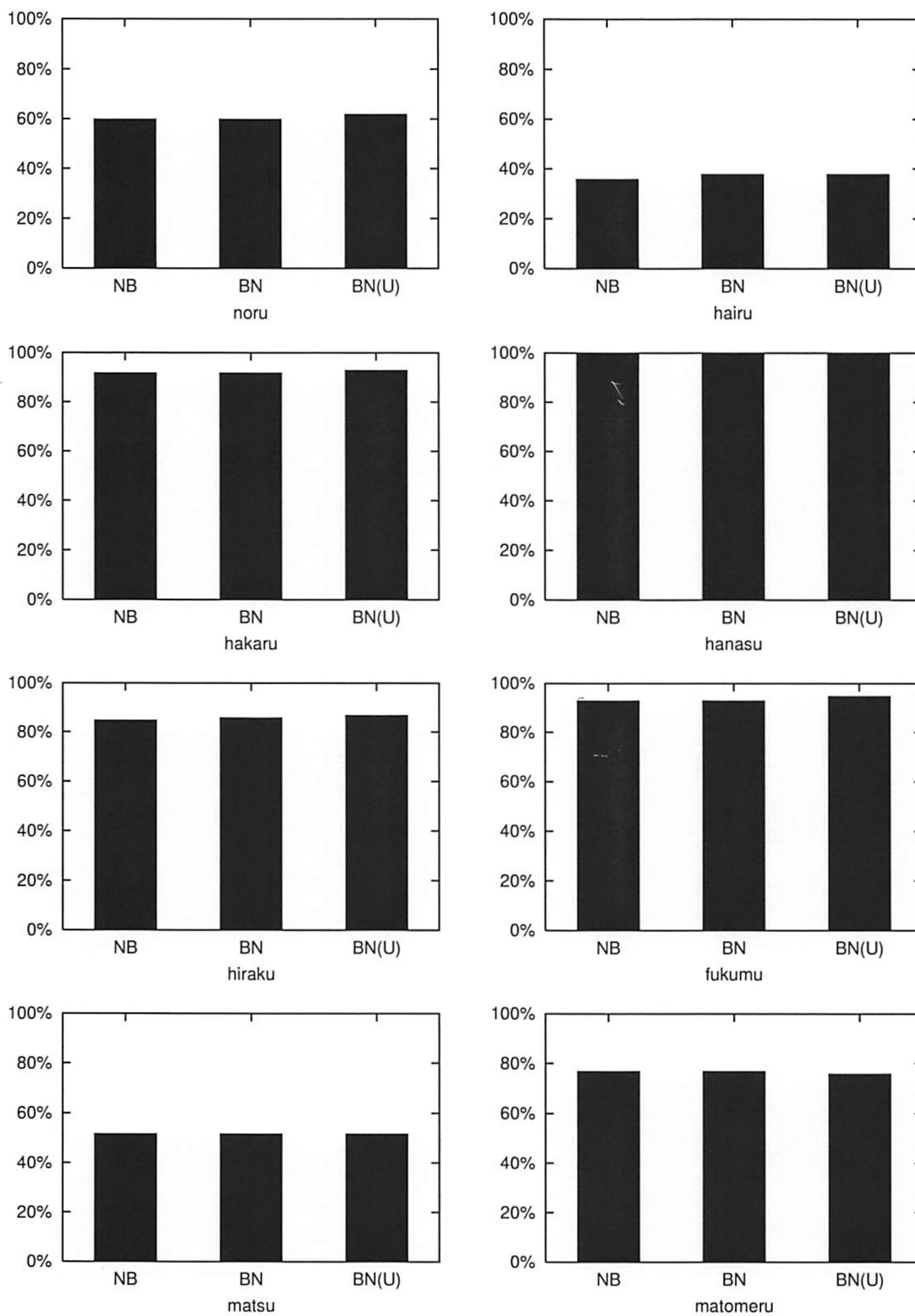
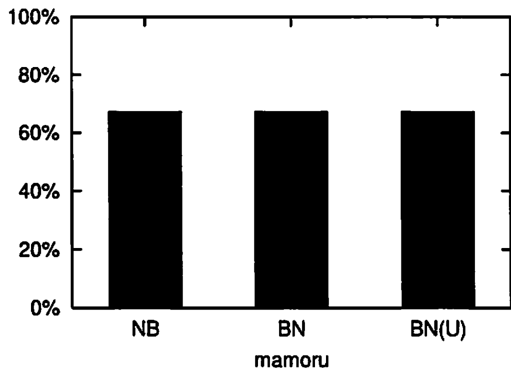
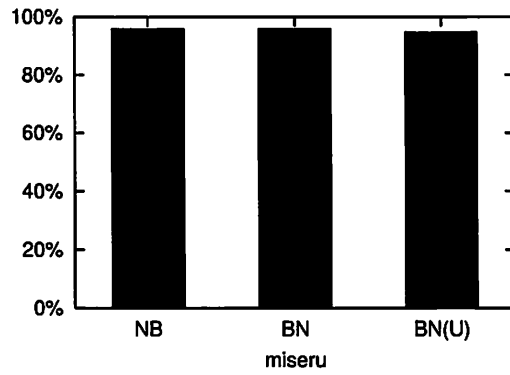
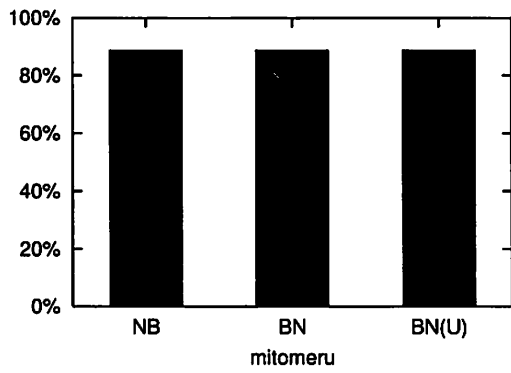
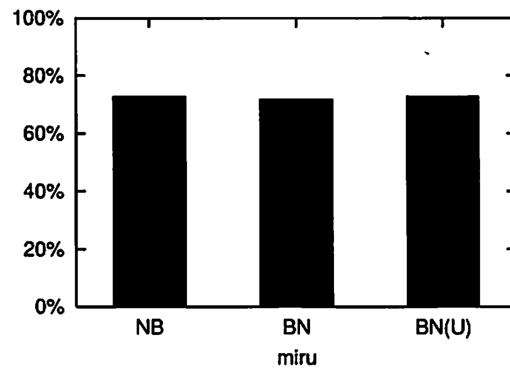


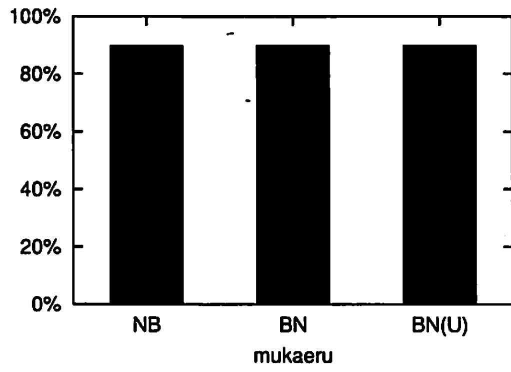
図 30: 動詞 その 5

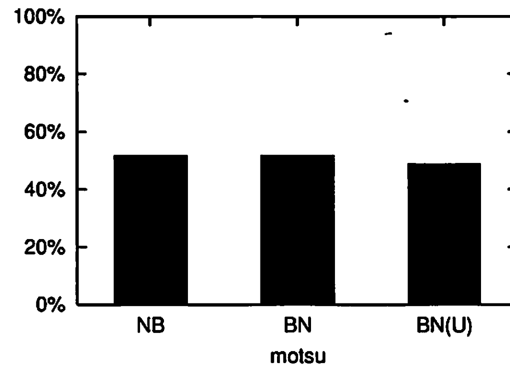


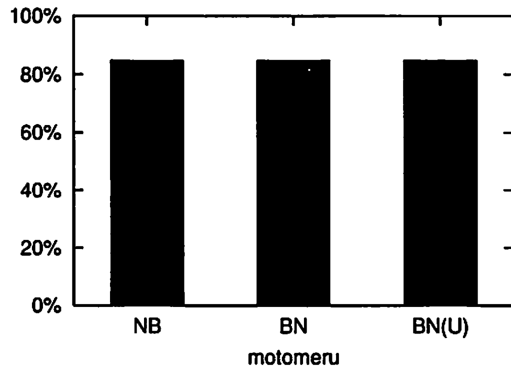












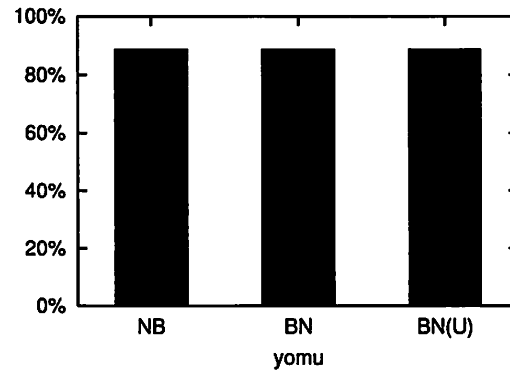
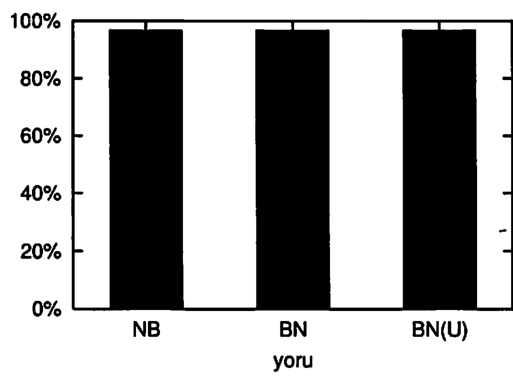


図 31: 動詞 その 6



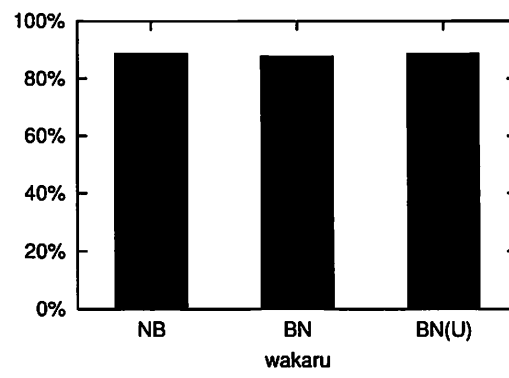


図 32: 動詞 その 7

付録B　プログラム

接合木への変換と，メッセージ伝播を扱うプログラムを提示する．

```
'''Belief Netowrk
'''
from __future__ import generators, division
import logging
log = logging.getLogger("beliefnetwork")

import psyco
psyco.jit()
from psyco.classes import *

class TableError(Exception):
    '''Not Found Table'''
    pass

# テーブルに置いて、
# 掛け算や割り算のとき、相手の値がない場合に使う値
TABLE_DEFAULT_RIGHT_VALUE = 0

# テーブルが0のとき、カットするか
IS_CUT = False

def DIV0(x, y):
    try:
        return x / y
    except ZeroDivisionError:
        return 0

def DIV0_0_0_EQ1(x, y):
    try:
        if x == y == 0:
            return 1
        else:
            return x / y
    except ZeroDivisionError:
            return 0
```

```
class Manager:
    def __init__(self, cpt=None, links=None):
        if cpt is not None:                                                        40
            self.set_cpt(cpt)
        if links is not None:
            self.set_links(links)

    def set_cpt(self, cpt):
        self.cpt = CPT(cpt)

    def set_links(self, links):
        self.links = links
                                                                                   50
    def build(self):
        self.network = Network(self.cpt, self.links)
        self.network.build_network()
        self.jtree = self.network.build_jtree()

    def use_topnode(self):
        self.jtree.use_topnode = True

    def use_every_normalizing(self):
        self.jtree.use_every_normalizing = True                                    60

    def set_table_default_right_value(self, value):
        TABLE_DEFAULT_RIGHT_VALUE = value

    def set_div_0_0_equal_1(self):
        DIV0 = DIV0_0_0_EQ1

    def set_cut(self):
        IS_CUT = True
                                                                                   70
    def setvalues(self, hash):
        self.network.setvalues(hash)

    def compute(self):
        return self.network.compute()

Model = Manager

class CPT:
    ''' 確率表                                                                     80
```

```
'''
def __init__(self, cpt):
    self._cpt = cpt
    if isinstance(cpt, dict):
        self.init_hash(cpt)

def init_hash(self, hash):
    ''' 初期化、主に値を揃えることを行う
    '''
    self._cp = {}                                                    90
    prob_value_table = {}
    for key, values in hash.iteritems():
        if isinstance(key, tuple):
            if len(key) == 2:
                # k が tuple で事前確率になっているとき
                (cond, prob) = key # P(prob|cond) = values
                if not isinstance(cond, tuple):
                    cond = (cond,)
                key = cond + (prob,) # ((a, b), c) -> (a, b, c)
            else:                                                    100
                # k が tuple だけど、事前確率になっていないとき
                prob = key[0]
                cond = ()
        else:
            # k が tuple でないとき、つまり事前確率になっていないとき
            prob = key              # P(prob) = values
            cond = ()
            key = (prob,)

        # prob に対して、そこに含まれる値の種 α を採取                   110
        if not prob in prob_value_table:
            prob_value_table[prob] = {}
        for value in values.iterkeys():
            if isinstance(value, tuple):
                if len(value) == 2:
                    (cond_value, prob_value) = value
                else:
                    prob_value = value[0]
            else:
                prob_value = value                                   120
            prob_value_table[prob][prob_value] = None

        # cpt を構築
```

```
            self._cp[key] = PT(values, list(key))
            #print prob, cond, key
        # node(prob) に含まれる値の種 α が weight になる
        self._weight_table = {}
        for prob, value in prob_value_table.iteritems():
            self._weight_table[prob] = len(value)
                                                                                130
    def get_cptable(self, ids):
        try:
            return self._cp[ids]
        except KeyError:
            raise TableError, ids

    def get_weight(self, id):
        '''weight を"s"s る
        '''
        return self._weight_table[id]                                           140
    weight = get_weight


class PT:
    ''' 個々のノードの確率表
    '''
    def __init__(self, table={}, ids=[]):
        self._table = table
        self._ids = ids
                                                                                150
    def __len__(self):
        return len(self._table)

    def is_table_old(self):
        '''0 or 1 以外をテーブルとする'''
        return self._table != {}

    def is_table_new(self):
        return self._ids != []
                                                                                160
    is_table = is_table_new

    def is_zero(self):
        return self._ids != [] and self._table == {}

    def get_table(self):
```

```
        return self._table

    def get_ids(self):
        return self._ids                                                    170

    def copy_table(self):
        if self._table == {}:
            return {}
        else:
            return self._table.copy()

    def copy_ids(self):
        return self._ids[:]
                                                                            180
    def copy(self):
        return PT(self.copy_table(), self.copy_ids())

    def __add__(self, other):
        if not self.is_table():
            if not other.is_table():
                return PT()
            else:
                return PT(other.copy_table(), other.copy_ids())
        else:                                                               190
            if not other.is_table():
                return PT(self.copy_table(), self.copy_ids())
        return self.merge(other, lambda x,y=0.001: x+y)

    def __div__(self, other):
        div0 = DIV0

        if not self.is_table():
            if not other.is_table():
                # 1 / 1                                                     200
                return PT()
            else:
                # 1 / Table
                table = {}
                for k,v in other.get_table().iteritems():
                    table[k] = div0(1, v)
                return PT(table, other.copy_ids())
        else:
            if not other.is_table():
```

```
            # Table / 1                                              210
            return PT(self.copy_table(), self.copy_ids())
        else:
            # Table / Table
            return self.merge(other, div0)

__truediv__ = __div__

def __mul__(self, other):
    if not self.is_table():
        if not other.is_table():                                     220
            # 1 * 1
            return PT()
        else:
            # 1 * Table
            return PT(other.copy_table(), other.copy_ids())
    else:
        if not other.is_table():
            # Table * 1
            return PT(self.copy_table(), self.copy_ids())
        else:                                                        230
            # Table * Table
            return self.merge(other, lambda x,y: x*y)

def merge(self, other, func):
    def large_small(self, other):
        if len(self.get_ids()) >= len(other.get_ids()):
            large = self
            small = other
        else:
            large = other                                            240
            small = self
        return (large, small)

    def check_common(large, small):
        common = []
        uncommon = []
        sid = small.get_ids()
        lid = large.get_ids()
        for i in sid:
            if i in lid:                                             250
                common.append(i)
            else:
```

```
            uncommon.append(i)
    return (common, uncommon)

def include_merge(t1, t2, i1, i2):
    table = {}
    # selfから otherへの変換リスト
    # self=[a, b, c], other=[c, a] -> numlist=[2, 0]
    numlist = map(lambda x: i1.index(x), i2)
    for k,v in t1.iteritems():
        k2 = tuple(map(lambda x: k[x], numlist))
        # ノードが ON,OFF なら関係ない問題のはず
        if t2.has_key(k2):
            table[k] = func(v, t2[k2])
        else:
            table[k] = func(v, TABLE_DEFAULT_RIGHT_VALUE)
    ids = i1[:]
    return (table, ids)

def diff_merge(t1, t2, i1, i2):
    table = {}
    ids = []
    for k1,v1 in t1.iteritems():
        for k2,v2 in t2.iteritems():
            table[k1 + k2] = func(v1, v2)
    ids = i1 + i2
    return (table, ids)

def complex_merge(t1, t2, i1, i2):
    table = {}
    ids = []
    # small の並び順で、small と large に含まれている要素の、large の番号
    nlist1 = map(lambda x: i1.index(x), filter(lambda x: x in i1, i2))
    # small の並び順で、small と large に含まれている要素の、large の番号
    nlist2 = map(lambda x: i2.index(x), filter(lambda x: x in i1, i2))
    # large の並び順で、small に含まれていない large の番号
    dlist1 = map(lambda x: i1.index(x), filter(lambda x: x not in i2, i1))
    # 必要なキーを抽出
    ct1 = []
    for k,v in t1.iteritems():
        kc = tuple(map(lambda x: k[x], nlist1)) # key1 と key2 の共通部分 (key2 の並び)
        kd = tuple(map(lambda x: k[x], dlist1)) # key1 だけの部分、これに key2 を加えれば共通部分が重複しない
        ct1.append((k, kc, kd, v))
    ct2 = []
```

```
        for k,v in t2.iteritems():
            kc = tuple(map(lambda x: k[x], nlist2)) # key1 と key2 の共通部分 (key2 の並び)
            ct2.append((k, kc, v))
        # 実際の処理
        for k1,kc1,kd1,v1 in ct1:                                                        300
            for k2,kc2,v2 in ct2:
                if kc1 == kc2:
                    table[kd1 + k2] = func(v1, v2)
        ids = map(lambda x: i1[x], dlist1) + i2
        return (table, ids)

    (large, small) = large_small(self, other)
    (common, uncommon) = check_common(large, small)
    '''
    if __debug__:                                                                        310
        print 'debug:'
        print 'large:', large
        print 'small:', small
        print 'common:', common
        print 'uncommon:', uncommon
        print
    '''
    (i1, i2) = (large.get_ids(), small.get_ids())
    (t1, t2) = (large.get_table(), small.get_table())
    table = {}                      # new table                                          320
    ids = []                        # new ids
    if uncommon == []:
        #print 'include'
        # small が large に完全に含まれている場合の処理
        (table, ids) = include_merge(t1, t2, i1, i2)
    elif common == []:
        #print 'diff'
        # small と large に共通点がない場合の処理
        (table, ids) = diff_merge(t1, t2, i1, i2)
    else:                                                                                330
        #print 'complex'
        # small が large 完全に含まれていない場合の処理
        (table, ids) = complex_merge(t1, t2, i1, i2)
    return PT(table, ids)

def get_sum(self, id):
    """ 確率の組からなるテーブルのある値について合計する
    """
```

```
        # 統合する id が含まれていなければなにもしない
        if id not in self.get_ids():                                                    340
            return self
        ids = self.get_ids()
        idnum = ids.index(id)            # 目的の id は ids の何番目か
        nlist = filter(lambda x: x != idnum, xrange(len(ids))) # 目的の id は削除するので、それ以外の id の番号
        new_ids = map(lambda x: ids[x], nlist) # 目的の id を削除した ids
        table = {}
        for k,v in self.get_table().iteritems():
            new_k = tuple(map(lambda x: k[x], nlist))
            if table.has_key(new_k):
                table[new_k] += v                                                       350
            else:
                table[new_k] = v
        return PT(table, new_ids)

    def get_sum_list(self, idlist):
        ids = self.get_ids()
        # 余計な id を切り捨てる
        idlist = filter(lambda x: x in ids, idlist)
        # 統合する id がないならなにもしない
        if len(idlist) == 0: return PT(self.copy_table(), self.copy_ids())              360
        # 統合対象になっていない id の番号 (ids での番号)
        alive_idnum = filter(lambda x: not ids[x] in idlist, xrange(len(ids)))
        # 新 id
        new_ids = map(lambda x: ids[x], alive_idnum)
        table = {}
        for k,v in self.get_table().iteritems():
            new_k = tuple(map(lambda x: k[x], alive_idnum))
            if table.has_key(new_k):
                table[new_k] += v
            else:                                                                       370
                table[new_k] = v
        return PT(table, new_ids)

    def live(self, potantial):
        ''' ポテンシャルの実装
        '''
        ids = self.get_ids()
        # 使用するポテンシャルに対応する ids の番号
        pnum = filter(lambda x: ids[x] in potantial, range(len(ids)))
        # 使用するポテンシャルがなければ終了                                              380
        #if pnum == [ ]: return PT()    # table = 1
```

```
        if pnum == []: return PT(self.copy_table(), self.copy_ids())# table はそのまま
        # 削除する Key を特定
        del_keys = {}
        for k in self.get_table().iterkeys():
            for n in pnum:
                if isinstance(potantial[ids[n]], list):
                    if k[n] not in potantial[ids[n]]:
                        del_keys[k] = None
                        break                                                   390
                else:
                    if k[n] != potantial[ids[n]]:
                        del_keys[k] = None
                        break
        # 削除する Key を実際に削除
        table = self.copy_table()
        for key in del_keys:
            del table[key]
        if table != {}:
            return PT(table, self.copy_ids())                                   400
        else:
            return PT()

    def alive(self, potantial):
        ''' ポテンシャルの実装
        '''
        def itercomb(list):
            for a in list[0]:
                for b in list[1]:
                    yield (a, b)                                                410
        def comb(list):
            newlist = []
            for a in list[0]:
                for b in list[1]:
                    if isinstance(a, tuple):
                        newlist.append(a + (b,))
                    else:
                        newlist.append((a, b))
            return newlist
                                                                                420
        ids = self.get_ids()
        # 使用するポテンシャルに対応する ids の番号
        pnum = filter(lambda x: ids[x] in potantial, range(len(ids)))
        # 使用するポテンシャルがなければ終了
```

```
if pnum == []: return PT(self.copy_table(), self.copy_ids())# table はそのまま

# 全てのノードに対して evidence が設定されている
# 処理を省略できる
if len(pnum) == len(ids):
    table = self.get_table()                                               430
    new_table = {}
    evidence = map(lambda x: potantial[ids[x]], pnum)
    if len(evidence) == 1:
        for e in evidence:
            try:
                new_table[e] = table[e]
            except KeyError:
                pass
    elif len(evidence) == 2:
        for e in itercomb(evidence):                                       440
            try:
                new_table[e] = table[e]
            except KeyError:
                pass
    else:
        for e in reduce(lambda x,y: comb((x, y)), list):
            try:
                new_table[e] = table[e]
            except KeyError:
                pass                                                       450
    if new_table != {}:
        return PT(new_table, self.copy_ids())
    else:
        return PT()


# 削除する Key を特定
del_keys = {}
for k in self.get_table().iterkeys():
    for n in pnum:                                                         460
        if isinstance(potantial[ids[n]], list):
            if k[n] not in potantial[ids[n]]:
                del_keys[k] = None
                break
        else:
            if k[n] != potantial[ids[n]]:
                del_keys[k] = None
```

```
                    break
        # 削除する Key を実際に削除
        table = self.copy_table()
        for key in del_keys:
            del table[key]
        if table != {}:
            return PT(table, self.copy_ids())
        else:
            return PT()

    def normalizing(self):
        '''normalizing(破壊的α作)'''
        table = self.get_table()
        sum = 0
        for v in table.itervalues():
            sum += v
        if sum != 0:
            for k,v in table.iteritems():
                table[k] = v / sum

    def __repr__(self):
        string = '{'
        for k,v in self.get_table().iteritems():
            string += '('
            for label in k:
                string += label + ', '
            string = string[:-2] + '): ' + str(v)
        string += '}'

        return string + str(self._ids)


class Network:
    def __init__(self, cpt, links):
        self.cpt = cpt
        self.links = links
        self.nodes = NodeSet()

    def get(self, id):
        if self.nodes.has_id(id):
            return self.nodes.get_node(id)
        else:
            return Node(id, self.cpt.get_weight(id))
```

```
def delete_node(self, node):
    self.nodes.delete(node)
    for onode in self.nodes:
        onode.disconnect(node)

def build_network(self):
    for (n1, n2) in self.links:
        node1 = self.get(n1)
        node2 = self.get(n2)                                            520
        self.nodes.add(node1)
        self.nodes.add(node2)
        self.connect(node1, node2)

def connect(self, n1, n2):
    n1.connect(n2)

def get_clusters(self):
    self.moralization()
    cliques = self.triangulation()                                      530
    return cliques

def clear(self):
    for (n1, n2) in self.links:
        n1.clear()
        n2.clear()

def reset(self):
    # node のリンクがでたら"s になっているので一旦 clear
    self.clear()                                                        540
    # 新たにリンクを生成
    self.build_network()

def rebuild_clustered_nodes_links(self, nodes):
    # リンクを消去
    for n in nodes.itervalues():
        n.clear()
    # リンクを生成
    for (n1, n2) in self.links:
        nodes[n1].connect(nodes[n2])                                    550

def build_jtree(self):
    cliques = self.get_clusters()
```

```
        cliques.check()
        sepset = cliques.getSepset()
        self.rebuild_clustered_nodes_links(cliques.get_all_nodes())
        #(cliques, sepset) = self.get_clusters_emulate()
        self.tree = JTree(cliques.data, sepset.list(), self.cpt)
        return self.tree
                                                                        560
    def setvalues(self, hash):
        self.tree.setvalues(hash)

    def compute(self):
        return self.tree.compute()

    def moralization(self):
        '''moralization'''
        connections = []
        for node in self.nodes:                                         570
            for (a, b) in node.parents.itercombi():
                connections.append((a, b))
        for (a, b) in connections:
            self.connect(a, b)

    def triangulation(self):
        """三角化
        1, グラフ G を G' にコピー (
        2, G' から頂点 V を選ぶ
        3, V と その隣のノードを全部接続する                            580
        エッジを追加してクラスターを生成する。
        4, G' で接続した線分のうち、G にない線分を G に追加する
        5, V を G' から削除する
        6, 2 に"s る (ノードが全部なくなるまで)

        グラフをコピーする部分はこの関数では省略する。元のグラフを今後使用
        することはなく、破壊しても問題ではない。

        この処理の後で必要なのは、生成されるクラスターである。
                                                                        590
        """

        clusterslist = Cliques()

        def sort(nodes):
            sorted = []
```

```
    (parents, children) = ({}, {})
    for (parent, child) in self.links:
        parents[parent] = None
        children[child] = None                                              600
    (a, b, c) = ([], [], [])
    # 親のいないノード
    for parent in parents:
        if not parent in children:
            try:
                a.append(self.nodes[parent])
            except KeyError:
                pass
    # 子のいないノード
    for child in children:                                                  610
        if not child in parents:
            try:
                b.append(self.nodes[child])
            except KeyError:
                pass
    # その他のノード
    for node in nodes:
        if not node in parents and not node in children:
            try:
                c.append(self.nodes[node])                                  620
            except KeyError:
                pass
    a.extend(b)
    a.extend(c)
    return a

def main(node, addededges, clusters):
    #print 'addededges: %d, clusters: %d' % (len(addededges), len(clusters))
    # リンクの追加
    for c in addededges:                                                    630
        (a, b) = c.list()
        self.connect(a, b)
    # クラスターをリストに追加
    clusterslist.addlist(clusters)
    #for n in self.nodes:
    #    if n.links.has(node):
    #        node.isUpdated = True
    # ノードの削除
    self.delete_node(node)
```

```
                                                                         640
    def find_next_node(nodes):
        list = []
        for node in nodes:
            (addededges, clusters) = node.get_update_cluster()
            #if not hasattr(node, 'isUpdated') or node.isUpdated:
            #    (addededges, clusters) = node.get_update_cluster()
            #    node.isUpdated = False
            # 加えられたノードの数
            addednumber = len(addededges)
            # 重量の計算                                                  650
            weight = 0
            for n in clusters:
                weight += n.weight()
            # リストに登録
            list.append((addednumber, weight, node))
            #node.cache = (addednumber, weight)
        return list

    while(len(self.nodes) != 0):
        #print 'resent list length: %d' % len(self.nodes)                 660
        #print 'clusters list number: %d' % len(clusterslist.data)
        # リンクが2つ以上ないノードを選ぶのは良くない
        nodes = filter(lambda x: len(x.links) > 1, self.nodes)
        if nodes == []:
            nodes = self.nodes
        #nodes = sort(nodes)
        # 削除対象ノードを選別する
        list = find_next_node(nodes)
        if list != None:
            node = min(list)[-1]                                          670
            (ae, cl) = node.get_update_cluster()
            main(node, ae, cl)

    return clusterslist

def __repr__(self):
    list = self.nodes.list()
    list.sort(lambda a,b: cmp(a.id, b.id))
    str = ' '
    for i in list:                                                        680
        str += ' %2s' % i.id
    str += '\n'
```

```
        for i in list:
            str += '%2s' % i.id
            for j in list:
                str += ' '
                if i.parents.has(j):
                    str += '%2s' % 'P'
                elif i.children.has(j):
                    str += '%2s' % 'C'                          690
                else:
                    str += '%2s' % 'X'
            str += '\n'
        return str


class JTree:
    def __init__(self, all_cluster, data, cpt):
        '''
        data: (node1, node2) というように sepset の組がある            700
        '''
        self.use_topnode = False
        self.use_every_normalizing = False
        self._cpt = cpt
        self._clusters = {}
        self._sepset = {}
        self._nodes = {}
        self._neighborhood = {}

        # 全 sepset のリスト                                          710
        for (a, b) in data:
            self._clusters[a] = a
            self._clusters[b] = b
            sepset = a & b
            self._sepset[sepset] = sepset

        # sepset がない場合 (cluster が一つの場合)
        if self._clusters == {}:
            for cluster in all_cluster.itervalues():
                self._clusters[cluster] = cluster               720

        # 全ノードのリスト
        for cluster in self._clusters:
            for node in cluster:
                self._nodes[node] = node
```

```
        # あるノードの隣のノード
        for cluster in self._clusters:
            self._neighborhood[cluster] = {}
            for (a, b) in data:                                          730
                if cluster == a:
                    self._neighborhood[cluster][b] = b
                elif cluster == b:
                    self._neighborhood[cluster][a] = a

    def itersepset(self):
        return self._sepset.itervalues()

    def choose_arbitrary_cluster(self):
        # pop arbitrary item(pair)                                       740
        (cluster_index, cluster_real) = self._clusters.popitem()
        # push item(pair)
        self._clusters[cluster_index] = cluster_real
        # cluster_index == cluster_real だけどね
        return cluster_real

    def iterclusters(self):
        return self._clusters.itervalues()

    def iternodes(self):                                                 750
        return self._nodes.itervalues()

    def print_status(self):
        return None
        def dprint(set):
            print '%s -> %s' % (set, set.pt)
        for set in self._clusters:
            print 'cluster:',
            dprint(set)
        for set in self._sepset:                                         760
            print 'sepset:',
            dprint(set)

    def initialization(self):
        ''' 初期化 '''
        self.initialization_one()
        self.initialization_set()
```

```
def initialization_one(self):
    ''' 初期化: table=1'''                                    770
    # クラスタの初期設定 table = 1
    for set in self.iterclusters():
        set.pt = PT()

    # sepset の初期設定 table = 1
    for set in self.itersepset():
        set.pt = PT()

def initialization_set(self):
    ''' 初期化: cluster の値を設定'''                          780
    def addnode(cluster, ids):
        table = self._cpt.get_cptable(ids)
        ptable = table.alive(self.potantial) # potantial の処理
        cluster.pt = ptable * cluster.pt

    # クラスタに実際の値を割り当てる
    for cluster in self.iterclusters():
        for node in cluster:
            for pnode in cluster:
                if node.parents.has(pnode):                   790
                    addnode(cluster, (pnode.id, node.id))

            # トップノードを使用
            if self.use_topnode == True:
                # cluster がまだテーブルをもっていなければ
                if not cluster.pt.is_table():
                    try:
                        addnode(cluster, (node.id,))
                    except TableError:
                        pass                                  800

    #for cluster in self.iterclusters():
    #    print cluster.pt.get_ids()


def backup(self):
    for i in self.iterclusters():
        i.bpt = i.pt.copy()
    for i in self.itersepset():
        i.bpt = i.pt.copy()                                   810
```

```
def restore(self):
    for i in self.iterclusters():
        i.pt = i.bpt.copy()
    for i in self.itersepset():
        i.pt = i.bpt.copy()

def global_propagation(self):
    clusters = self._clusters
    sepsetlist = self._sepset                                                      820
    def message(src, dist):
        '''message passing from src to dist.
        [src] --(message)--> [dist]
        '''
        # sepset を得る (nodeset のクラスの実装に問題があるようだ)
        try:
            sepset = sepsetlist[src & dist]
        except KeyError:
            sepset = sepsetlist[dist & src]
                                                                                   830
        # 古い sepset を保存
        old_sepset_pt = sepset.pt

        if IS_CUT and sepset.pt.is_zero():
            return

        # sepset を更新
        # src.pt.get_sum_table(src - sepset) をやりたい
        sepset.pt = src.pt.get_sum_list(map(lambda x: x.id, (src - sepset)))
        # 確率値が小さくなりすぎて 0.0 になるのを防ぎたい                              840
        if self.use_every_normalizing:
            sepset.pt.normalizing()

        # 確率を伝搬
        dist.pt = dist.pt * (sepset.pt / old_sepset_pt)
        # 確率値が小さくなりすぎて 0.0 になるのを防ぎたい
        if self.use_every_normalizing:
            dist.pt.normalizing()

    def collect_evidence(prev, cur, mark):                                        850
        mark[cur] = True
        for next in self._neighborhood[cur]:
            if next not in mark:
                collect_evidence(cur, next, mark)
```

```
            if prev != None:
                message(cur, prev)

    def distribute_evidence(cur, mark):
        mark[cur] = True
        for next in self._neighborhood[cur]:                              860
            if next not in mark:
                message(cur, next)
                distribute_evidence(next, mark)

    # 適当に cluster を一個選ぶ
    arbitrary_cluster = self.choose_arbitrary_cluster()
    collect_evidence(None, arbitrary_cluster, {})
    distribute_evidence(arbitrary_cluster, {})

def merginalization(self):                                                870
    clusters = self._clusters
    pts = {}
    self.pt = pts
    for cluster in self.iterclusters():
        for node in cluster:
            # merge
            pt = cluster.pt.get_sum_list(map(lambda x: x.id, filter(lambda x: x != node, cluster)))
            #pt = cluster.pt
            #for n in cluster:
            #    if node != n:                                            880
            #        pt = pt.get_sum(n.id)
            # normalizing

            # normalizing しない。
            #pt = pt.normalizing()
            id = tuple(pt.get_ids())
            if not pts.has_key(id):
                pts[id] = []
            pts[id].append(pt)
    '''                                                                   890
    if __debug__:
        for k,v in pts.iteritems():
            print k
            for i in v:
                print v
            print
    '''
```

```
def merginalization2(self):
    def normalizing(result_list):                                                    900
        sum = {}
        sumnum = 0
        # 複数の確率が示されたらそれを合計
        for list in result_list:
            for k,v in list.get_table().iteritems():
                if not sum.has_key(k):
                    sum[k] = 0
                sum[k] += v
                sumnum += v
        # 合計が 1.0 になるように調"s                                                    910
        if sumnum != 0:
            for k,v in sum.iteritems():
                sum[k] = v / sumnum
        return sum

    clusters = self._clusters
    pts = {}
    for cluster in self.iterclusters():
        for node in cluster:
            # merge                                                                  920
            pt = cluster.pt.get_sum_list(map(lambda x: x.id, filter(lambda x: x != node, cluster)))
            id = tuple(pt.get_ids())
            if not pts.has_key(id):
                pts[id] = []
            pts[id].append(pt)
    for key, values in pts.iteritems():
        pts[key] = normalizing(values)

    self.pt = pts
                                                                                     930
def compute_potantial(self):
    for cluster in self.iterclusters():
        cluster.pt = cluster.pt.live(self.potantial)

def setvalues(self, hash):
    self.potantial = hash

def compute(self):
    self.initialization()
    #self.compute_potantial()                                                        940
```

```
        self.global_propagation()
        #self.merginalization()
        self.merginalization2()
        return self.pt

    def print_clusters(self):
        ''' デバッグ用'''
        for cluster in self.iterclusters():
            print cluster.pt
                                                                        950

#class NodeSetList:
class Cliques:
    def __init__(self, list=None):
        self.data = {}
        if list is not None:
            self.addlist(list)

    def check(self):
        ''' リストに cluster を追加する前にチェックが必要。                960

        3 個の cluster は OK。でも、3 個に満たない cluster はチェックが必要。
        まず、その cluster(node の数が 3 個に満たない) が他の cluster に含ま
        れていないかどうか。含まれていればそのクラスターは必要ない。含
        まれていなければ、その cluster は必要な cluster なのでリストに追加
        する。
        '''
        # node が 3 個の cluster と 2 個の cluster と 1 個の cluster に分ける
        three = []
        twe = []                                                        970
        one = []
        for cluster in self.data.itervalues():
            length = len(cluster)
            if length == 3:
                three.append(cluster)
            elif length == 2:
                twe.append(cluster)
            elif length == 1:
                one.append(cluster)
        # いらないリスト                                                 980
        dellist = []
        # 2 個 cluster の内、3 個 cluster に含まれていないもの
        for c2 in twe:
```

```
        for c3 in three:
            if (c2 & c3) == c2:
                dellist.append(c2)
    # 1 個の cluster について
    for c1 in one:
        for c3 in three:
            if (c1 & c3) == c1:                                             990
                dellist.append(c1)
        for c2 in twe:
            if (c1 & c2) == c1:
                dellist.append(c1)
    # もとのリストから、いらないリストを削除
    for c in dellist:
        self.delete(c)

def add(self, cluster):
    self.data[tuple(cluster.idslist())] = cluster                           1000

def delete(self, cluster):
    try:
        del self.data[tuple(cluster.idslist())]
    except KeyError:
        pass

def addlist(self, clusters):
    for cluster in clusters:
        self.add(cluster)                                                   1010

def getSepset(self):
    sepset = Sepsets()
    list = self.list()
    for i in range(len(list)):
        for j in range(i+1, len(list)):
            sepset.add(list[i], list[j])
    sepset.make(len(self.data))        # make の中でソートしています
    return sepset
                                                                            1020
def get_all_nodes(self):
    hash = {}
    for c in self.data.itervalues():
        for n in c:
            hash[n.id] = n
    return hash
```

```
    def __iter__(self):
        return self.data.itervalues()
                                                                                          1030
    def list(self):
        return self.data.values()


class Sepsets:
    '''clique と clique のセットのリスト
    '''
    def __init__(self):
        self._list = []
                                                                                          1040
    def add(self, a, b):
        mess = self.mess(a, b)
        if 0 < mess:
            self._list.append((a, b))

    def make(self, N):
        self.sort()
        self.cut(N)

    def print_debug(self):                                                                1050
        for i in range(len(self._list)):
            a = self._list[i][0]
            b = self._list[i][1]
            mess = self.mess(a, b)
            cost = self.cost(a, b)
            print i, mess, cost, self._list[i]

    def sort(self):
        '''mess が最大になるように並べる。同じ数の場合は、cost が小さくなるようにする
        '''                                                                               1060
        for i in range(len(self._list)):
            item = self._list[i]
            self._list[i] = (self.mess(item[0], item[1])*-1, self.cost(item[0], item[1]), item)
        self._list.sort()
        for i in range(len(self._list)):
            self._list[i] = self._list[i][-1]

    def cut(self, N):
        '''N 個の clique と N-1 個の sepset
```

```
        '''                                                    1070
        self._list = self._list[:N-1]

    def extract_max(self):
        return self._list.pop()

    def mess(self, a, b):
        '''a と b の重なりの数
        '''
        return len(a & b)
                                                               1080
    def cost(self, a, b):
        '''a と b の weight の合計
        '''
        return a.weight() + b.weight()

    def __iter__(self):
        return iter(self._list)

    def __repr__(self):
        return repr(self._list)                                1090

    def list(self):
        return self._list[:]


class NodeSet:
    ''' ノード達を管理

    順序なしの集合で、値に重なりがない。
                                                               1100
    a & b(a * b) で共通部分を抽出する
    a == b は順序に関係なく、要素が等しいか比較
    '''
    def __init__(self, list=None):
        self.data = {}
        self._data = self.data
        if list != None:
            self.addlist(list)

    def addlist(self, list):                                   1110
        for n in list:
            self.add(n)
```

```
def add(self, node):
    self.data[node.id] = node

def delete(self, node):
    try:
        del self.data[node.id]
    except KeyError:                                        1120
        pass

def has_id(self, id):
    return self.data.has_key(id)

def has(self, node):
    return self.has_node(node)

def has_node(self, node):
    return self.has_id(node.id)                             1130

def __iter__(self):
    return self.iternodes()

def iterids(self):
    return self.data.iterkyes()

def iternodes(self):
    return self.data.itervalues()
                                                            1140
def list(self):
    return self.nodeslist()

def idslist(self):
    return self.data.keys()

def nodeslist(self):
    return self.data.values()

def get_node(self, id):                                     1150
    return self.data[id]

def __len__(self):
    return len(self.data)
```

```
def __eq__(self, other):
    if isinstance(other, self.__class__):
        return self.data == other.data
    else:
        return False                                                        1160

def __ne__(self, other):
    if isinstance(other, self.__class__):
        return a.data != b.data
    else:
        return True

def __call__(self):
    return self.__iter__
                                                                            1170
def __and__(self, other):
    if len(self) >= len(other):
        long, short = self, other
    else:
        short, long = self, other
    return self.__class__(filter(long.has, short))

def __sub__(self, other):
    ans = NodeSet()
    for n in self:                                                          1180
        if not other.has(n):
            ans.add(n)
    return ans

def __hash__(self):
    '''hashから返されるデータの順番が常に一緒（同じ実行"s程において）であることを"s程している
    '''
    return hash(tuple(self.idslist()))

def itercombi(self):                                                        1190
    ''' 全ての組み合わせを返す
    '''
    list = self.nodeslist()
    for i in range(len(list)):
        for j in range(i+1, len(list)):
            yield (list[i], list[j])

def __repr__(self):
```

```
        return str(self.list())
                                                                        1200
    def minweight(self):
        list = []
        for i in self.iternodes():
            list.append((i.weight(), i))
        p = min(list)
        return p[-1]

    def minweight2(self):
        list = []
        for i in self.iternodes():                                      1210
            list.append((i.weight(), i))
        p = min(list)
        list.remove(p)
        q = min(list)
        return (p[-1], q[-1])

    def sorted_list_by_weight(self):
        list = []
        for i in self.iternodes:
            list.append(i.weight, i)                                    1220
        list.sort()
        return list

    def weight(self):
        sum = 0
        for n in self.iternodes():
            sum += n.weight()
        return sum

    def __getitem__(self, id):                                          1230
        '''return node from id'''
        return self._data[id]


class Node:
    def __init__(self, id, weight):
        self.id = id
        self._weight = weight
        self.links = NodeSet()
        self.parents = NodeSet()                                        1240
        self.children = NodeSet()
```

```
def clear(self):
    self.links = NodeSet()
    self.parents = NodeSet()
    self.children = NodeSet()

def connect(self, other):
    self.links.add(other)
    other.links.add(self)                                                   1250
    self.children.add(other)
    other.parents.add(self)

def disconnect(self, other):
    if self.links.has(other):
        self.links.delete(other)
    if other.links.has(self):
        other.links.delete(self)
    if self.children.has(other):
        self.children.delete(other)                                         1260
    if other.parents.has(self):
        other.parents.delete(self)
    # 以下の切断は、方向性関係なし
    if self.parents.has(other):
        self.parents.delete(other)
    if other.children.has(self):
        other.children.delete(self)

def __hash__(self):
    return hash(self.id)                                                    1270

def weight(self):
    return self._weight

def __repr__(self):
    #return 'node(%s)' % self.id
    return '"%s"' % self.id

def get_update_cluster(self):
    addededges = []                                                         1280
    clusters = []
    if len(self.links) > 1:          # リンクの数が2つ以上のとき
        (addededges, clusters) = self.update_cluster_normal()
    elif len(self.links) == 1:       # リンクの数が1つのとき
```

```
            clusters = [NodeSet((self, self.links.list()[0]))]
        else:                               # リンクの数が 0 のとき
            clusters = [NodeSet((self,))]
        return (addededges, clusters)

    def update_cluster_normal(self):                                             1290
        ''' 頂点を引数に取り、追加されるエッジと生成されるクラスターを返す

        頂点に隣接する頂点、つまりリンク先の頂点を見付ける。
        リンク先の頂点同士がリンクされていれば、それはクラスター。
        リンクされていなれば、リンクで結んで (エッジを追加) クラスターを生成する。

        なるべく、追加するエッジの数は減らし、クラスターの重みを減らす。余っ
        た隣接頂点同士を結べばエッジの数もクラスターの数も減る。つまり、追
        加するエッジの数は (余った隣接頂点の数 / 2) を繰り上げした数になる。
                                                                                 1300
        '''
        clusters = []
        addededges = []
        clustered_node = NodeSet()
        rests = []
        for (a, b) in self.links.itercombi():
            # クラスターかどうか、
            if a.links.has(b):
                # クラスターリストに追加する
                clusters.append(NodeSet((a, b, self)))                           1310
                # クラスターを構成しているノード
                clustered_node.add(a)
                clustered_node.add(b)
        for n in self.links:
            # クラスターを構成しているノード以外は、
            if not clustered_node.has(n):
                # 余りノード
                rests.append(n)

        # 余りノードの数が奇数ならば特別な処理が必要                               1320
        # 1, より少ない数のエッジを追加
        # 2, クラスターの重みが最小になるようにする。
        # 条件 1,2 を満すた”s に、以下のような処理をおこなう
        # 余りノードの数が偶数ならば、ノード同士を接続する。
        # しかし、奇数ならば一つノードが余るので、最小の重みのノードと
        # 最大の重みのノードを結ぶ。最小の重みのノードが 2 回ほかのノー
        # ドと接続され、最大の重みのノードが一度だけ他のノードと結ばれ
```

```
# る。

if (len(rests) % 2) == 1:                                          1330
    rests.sort(lambda x,y: cmp(x.weight(), y.weight()))
    #a = self.links.minweight()
    (a1, a2) = self.links.minweight2()
    b = rests.pop(-1)
    if a1 == b:
        a = a2
    else:
        a = a1
    addededges.append(NodeSet((a, b)))
    clusters.append(NodeSet((self, a, b)))                         1340
while(rests != []):
    a = rests.pop()
    b = rests.pop()
    addededges.append(NodeSet((a, b)))
    clusters.append(NodeSet((self, a, b)))

return (addededges, clusters)
```