ICOT Technical Memorandum: TM-0630

TM-0630

並列論理プログラミングにおける制約充足問題の解法

横尾真(NTT), 上田和紀

November, 1988

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03) 456-3191~5
Telex ICOT J32964

Institute for New Generation Computer Technology

並列論理プログラミングにおける制約充足問題の解法

Solving Constraint Satisfaction Problems in Concurrent Logic Programming

横尾　真(NTT情報通信処理研究所)　上田　和紀((財)新世代コンピュータ技術開発機構)

概要

並列論理型言語を用いて、制約を能動的に利用し、複数の可能性を並列に探索して、制約充足問題の全解を求める方法を示す。さらに、全解探索を目的とした制約論理型言語のプログラムを、本方法に基づく、同じ解を求める並列論理型言語のプログラムに系統的に変換する方法を示す。これらの方法により、宣言的な制約の記述から、複数の可能性を並列に探索し、制約を能動的に用いて効率的に制約充足問題を解くプログラムを得ることが可能となる。制約を受動的に用いるプログラムと、本方式によって得られたプログラムとの実行効率、並列度等に関する比較を行い、本方式のプログラムの実行効率が高いことを示す。

1. 研究の動機

本研究の目的は、AIにおける重要な問題である制約充足問題を並列論理型言語を用いて解くことである。並列論理型言語により制約充足問題を解くことの利点は次の通りである。

- 並列性を生かした効率的な実行が可能になる。
- 複数の可能性を同時に探索することにより、探索の制御を行うことが可能となる。

制約充足問題は次のように定式化される。有限領域D1, D2, ..., Dn中の値をとる変数の有限集合I={X1, X2, ..., Xn}、および制約の集合が存在するとする。制約 c(Xi1, ..., Xik)はIの要素であるk変数間の制約であり、直積Di1×...×Dikの部分集合である。これは互いに整合のとれた変数の値の組を表す。制約充足問題を解くことは、すべての制約を充足するような変数への値の割当を求めることである。

論理型言語で制約充足問題を解く場合、制約を宣言的に記述し、バックトラック機構を用いて探索を行うことにより、容易にプログラムの記述ができるが、このようなgenerate & test型のプログラムは非効率的である。効率の向上をはかるため、制約を能動的に用いるように論理型言語の機能を拡張する研究が、制約論理型言語と呼ばれる分野で盛んである[Jaffer 87], [坂井 88], [大木 88]。特に、ECRC (European Computer-Industry Research Centre)で開発された制約論理型言語CHIP[Dincbas 88]は、領域変数と呼ばれる有限領域の値域を持つ変数を導入し、forward checkingと呼ばれる手法を用いて制約を評価することにより、制約充足問題を効率的に解くことを可能にしている。制約を能動的に用いるという意味は、generate & test型のプログラムでは、制約はgenerateされた解が正しいかどうかの、事後的なテストのみに用いられるのに対し、制約をgenerateの可能性を絞るために、generateを行う前に利用するということである。

本論文ではまず、GHC[上田 85]によって代表されるcommitted-choice型の並列論理型言語上で、制約論理型言語と同様に制約を能動的に用いて制約充足問題を解く方法を示す。この方法により、並列性を生かして、かつ効率的に制約充足問題を解くことができるが、次のような理由により、記述の労力が大きい。

- 制約を、領域を狭めるための手続きとして記述しなければならない。

この問題点を解決するために、直接並列論理型言語でのプログラミングを行うのではなく、制約論理型言語を用いて記述された宣言的な制約の記述を系統的に変換して並列論理型言語のプログラムを得る方法を検討する。この方法の意義は次の通りである。

- 記述の労力が削減される。
- 制約論理型言語を、GHCの一つのユーザ記述用の言語として位置づけられる。

最後に、本方式によって得られたプログラムと、並列論理型言語上で制約を受動的に用いるプログラムとの実行効率、並列度等に関する比較を示す。

2. 制約充足問題の並列的解法

2.1 parallel generate & reduce

制約を能動的に用いて、並列に複数の可能性を探索して制約充足問題を解く手続き（以下、generate & testとの対比で、parallel generate & reduceと呼ぶ）は次のようにまとめられる。ここで、x1, X2, ..., xnは変数名、D1, D2, ..., Dnは変数値の取り得る有限領域である。

Procedure p_g_r((x1, D1), (x2, D2), ..., (xn, Dn))

①すべての変数x1, ..., xnの値が決定されていれば解とする

②ある変数xiの値域が空集合ならば失敗とする

③ある変数xiの値域の大きさが1なら、xiの値を決定し、適用可能な制約を用いて他の変数の値域を狭める。①に戻る。

④値域の大きさが1より大きい変数xiを選択する。

xiの値域Di中の各値v1,v2,...,vmに対して、xiの値域をその値のみからなる集合、すなわち、{v1}, {v2}, ...,{vm}としたものそれぞれに対して再帰的にparallel generate & testを行う。すなわち、m個のプロセス

```
p_g_r((x1,D1),...,(xi,{v1}),...),
p_g_r((x1,D1),...,(xi,{v2}),...),
...
p_g_r((x1,D1),...,(xi,{vm}),...)
```

を起動する。

④での変数の選択にヒューリスティックスを用いることができる。なるべく領域の大きさが小さい変数から選択を行うことによって、失敗を少なくすることができるという、first-fail principleが知られている[Van Hentenryck 87]。

2.2.forward checkingを用いたparallel generate & reduce

本研究では制約の能動的な利用方法として、forward checkingという手法を用いる。forward checkingの特徴は次の通りである。

- n変数に関する制約は、制約中のn-1個の変数の値が決定された時に起動される。
- 制約を起動することにより、値が未定の変数の値域を狭める。
- 制約は決定手続き（制約中のすべての変数の値が決まった時に、制約を満たすかどうかを判断できる）であればよい。
- 決定手続きを用いて、値が未定の変数の値域から、制約を満たさない値を取り除く。
- 制約の性質から他の変数の値域を狭めるための効率的な手続きが存在する制約（equality、non-equality等）については、制約評価のメカニズムを特殊化しておくことが可能である。

制約評価の方法としてforward checkingを用いた理由は以下の通りである。

- 有限領域を対象とする点で制約充足問題に向いている。
- 扱える制約が一般的である。
- forward checkingを基本として、制約評価の機能を拡張することができる（n-1個の変数が決定する以前に制約を用いるlook-ahead ヒューリスティックス等[Dincbas 87]）。

2.1のparallel generate & reduceの手続きを記述する際に、①、②、④、⑤の手続きは、変数の選択のヒューリスティックスを固定すれば、問題に依存しない一般的な手続きを用いることができる。問題は、ある変数が決定されたときに、適用可能な制約を選択して他の変数の領域を狭めるという③の手続きを記述することである。変数の選択の順序は実行時に決定されるため、ある変数が決定されたときに、どの制約が適用可能かは実行時にしか分からない。

本論文ではつぎのような方法を取る。すなわち、ある変数が決定されたときに適用される手続きとして、その変数を含むすべての制約について、適用可能であるかのチェックを行い、適用可能であれば制約を適用して他の変数の領域を狭める、適用可能でなければ（n-1個の変数がまだ決定していない、あるいはすでにn個決定しているならば）なにもしないという手続きをあらかじめ記述しておく。

別の方法として、制約のリストを引数として持ち、実行時に適用可能な制約を選択するという方法が考えられる。本論文の方法では、n変数の制約はn箇所に記述しておくことになり、記述の労力が大きい。一方、制約のリストを引数として持つ方法は、実行可能な制約を選択する処理のオーバーヘッドが問題となる。

4queenのGHCのプログラムを付録1に示す。このプログラムは、コミットオペレータをカットオペレータに置き換えることにより、決定的なPrologプログラムとして実行でき、また、restricted AND並列Prologの処理系で実行することによっても、複数の可能性を並列に探索することができる。

3．宣言的な制約記述からの変換方法

2章で示した方法でプログラミングをする際の問題点は、次のような理由により記述の労力が大きいことである。

- committed choice型の言語はバックトラックのような探索向きの機能がないため、複数の可能性を探索する手続きを陽に記述する必要がある。
- 制約を、領域を狭めるための手続きとして記述しなければならない。

このうち、探索の手続きは問題に依存しない汎用のルーチンとして記述しておき、それを利用することで記述の労力を削減することができる。一方、制約は問題ごとに記述することが避けられない。

プログラミングを容易にするためには、宣言的な制約の記述ができることが望ましい。このため、以下、直接並列論理型言語でのプログラミングを行うのではなく、より高レベルのユーザ言語を仮定し、その言語の記述を変換して並列論理型言語のプログラムを得る方法を検討する。

高レベルのユーザ言語の記述方法を4queenを例にとって説明する（図3.1）。付録1のGHCプログラムと比較して大幅に記述量が少なく、読みやすくなっている。プログラムは領域変数の宣言、制約の記述、generateの順序の指定からなる。領域の宣言はdomainという述語により行われる。

```
domain predicate（set1,....,setn）
```

という宣言は、predicateが領域変数のリストn本を引

```
domain four_queens([1,2,3,4]).
four_queens([X1,X2,X3,X4]) :-
   check([X1,X2,X3,X4])//
   generate_left_to_right([X1,X2,X3,X4]).

check(X) :-
   pair(X,Xres),
   allsafe(Xres).
allsafe([]).
allsafe([F|T]) :-
   allnoattack(F,T),
   allsafe(T).
allnoattack(X,[]).
allnoattack([Nb1,Var1],[[Nb2,Var2]|Z]) :-
   Var1=\=Var2,
   Var1-\=Var2+Nb1-Nb2,
   Var1=\=Var2+Nb2-Nb1,
   allnoattack([Nb1,Var1],Z).

pair(X,Y) :- pair_aux(X,Y,1).
pair_aux([],[],N).
pair_aux([X|Y],[[N,X]|Yres],N) :-
   N1 is N+1,
   pair_aux(Y,Yres,N1).
```

図3.1. ユーザ記述言語による4queenのプログラム

```
check([x1,x2,x3,x4]) :-
   x1 =\= x2,x1 =\= x2 + 1 - 2,x1 =\= x2 + 2 - 1,
   x1 =\= x3,x1 =\= x3 + 1 - 3,x1 =\= x3 + 3 - 1,
   x1 =\= x4,x1 =\= x4 + 1 - 4,x1 =\= x4 + 4 - 1,
   x2 =\= x3,x2 =\= x3 + 2 - 3,x2 =\= x3 + 3 - 2,
   x2 =\= x4,x2 =\= x4 + 2 - 4,x2 =\= x4 + 4 - 2,
   x3 =\= x4,x3 =\= x4 + 3 - 4,x3 =\= x4 + 4 - 3.
```

図3.2. 部分評価された制約の記述

```
domain four_queens([1,2,3,4]).
four_queens([X1,X2,X3,X4]) :-
   check([X1,X2,X3,X4]),
   labeling([X1,X2,X3,X4]).

labeling([]).
labeling([X|Y]) :-
   indomain(X),
   labeling(Y).
```

図3.3. CHIPによる4queenのプログラム

数とし、各リスト中の領域変数の値域がset1,...,set nであることを示す。

述語定義の本体は、

頭部 :- 制約の記述... //
generateの順序の指定

という形式で記述される。頭部の引数は領域変数のリストである。generateの順序の指定は、generate_first_fail,generate_left_to_right等の、あらかじめ指定されたいくつかの方法から選択するものとする。制約の記述方法は通常のPrologのプログラムとほぼ同様である。=\=というnon-equalityを表す組み込みの制約が存在するとする。制約は領域を持つ変数と通常の論理変数を引数とする。

このユーザ記述言語を変換して、並列論理型言語のプログラムを得る際、領域変数の個数と値域はdomainによる宣言と述語定義の頭部で与えられており、generateの順序も指定されている。探索の手続きは汎用のルーチンを用いればよいため、宣言的な制約の記述を、変数の領域を変更する手続きに変換することができればよい。この変換を次の手順で行う。

①宣言的な制約の記述を部分評価することにより、組み込みの制約の述言を得る。

②組み込みの制約の記述を領域変更手続きに変換する。

③領域変更手続きを組み合わせて、各変数が決定された時に他の変数の領域を変更する手続きを構成する。

制約の記述を部分評価した結果を図3.2に示す。ただし領域変数は変数名に対応するアトムに束縛されている。

②のステップで、組み込みの制約に関して、その制約を用いて領域を変更する効率の良い手続きが与えられていれば、それを利用する。non-equality、equality等については効率の良い手続きが与えられているものとする。効率の良い手続きが与えられておらず、制約が決定手続きとしてしか定義されていない場合は、1つ1つ試してみるという手続きを生成する。ユーザが新しい組み込みの制約を定義することも可能である。

本論文で用いているユーザ記述言語による制約の記述は、CHIPなどの制約論理型言語における記述と同一であり、この変換方法により、制限付きの制約論理型言語のプログラムを並列論理型言語のプログラムに変換することができる。図3.3にCHIPによる4queenのプログラムを示す。checkの定義は図3.1のプログラムと同一であり、CHIPではgenerateの順序の指定がindomainという非決定的に領域変数の値を定める述語を用いてプログラムで記述されている点のみが異なる。

制約論理型言語のプログラムが変換可能であるためには次の条件を満たすことが必要である。

- 領域変数およびその領域が指定されていること。
- generateの順序が指定されていること。
- 制約の記述を部分評価することにより組み込みの制約の述言に変換されること。
- 部分評価によって得られた組み込みの制約中の論理変数がすべて基底となっていること。

制約充足問題を解くプログラムのほとんどはこれらの条件を満たしていると考えられる。これらの条件を満たさないのは、例えば盤面の大きさnを引数としてnqueenを計算するプログラム等である。

本論文で示したユーザ記述言語のコンパイルという方法とは別のアプローチとして、並列実行の機能と制約の能動的な適用機能を持つ制約並列論理型言語を設計し、そのインタプリタをGHC等の並列論理型言語で記述するという方法が考えられる。このアプローチにより、前述のようなプログラムの制限はなくなるが、多環境性が必要となるため処理系が複雑になるという問題点がある。

4. 評価

いくつかの例題に関して、3章の方法で得られたプログラムの評価を行う。比較の対象として[上田 85].

[上田 86]の方式によるプログラムと、[奥村 87]の方式によるプログラムを選ぶ。これらは制約を受動的に利用し、複数の可能性を並列に探索する方式である。前者を継続方式と呼び、後者をレイヤードストリーム方式と呼ぶ。

継続方式のプログラムは、制限付きのPrologプログラムを並列論理型言語のプログラムに変換することによって得られる。この方式では本来OR関係にある節をANDゴールで呼び出し、継続（Continuation）と呼ばれる概念を用いて単一環境で全解収集を行う。レイヤードストリーム方式は、並列論理型言語で探索型のプログラムを直接記述するために考案されたものであり、解を表現するために、並列実行に適したレイヤードストリームと呼ばれる階層化された再帰的なデータ構造を用いる。この方式の特徴は、複数の可能性のgenerateに相当する処理を、すべての変数に関して並列に行い、並列に制約のチェックを行う点にある。

比較は２つの観点から行う。最初に、実行時間、リダクション数、サイクル数、サスペンション数、並列度を比較する。実行時間はGHCのプログラムのコミットオペレータをカットオペレータに置き換えることによってPrologの処理系で実行可能な形式としたプログラムで評価する。その他のパラメータはGHCの処理系のシミュレータで測定されるものである。パラメータの意味を表4.1に示す。次に、制約充足問題を探索問題とみなし、探索を行った探索木のノードの個数の比較を行う。実行時間、リダクション数等の測定結果は、領域の表現、制約を評価する手続き等のインプリメンテーション技術に依存するが、探索木のノードの個数はインプリメントに依存しない。

4.1.例題の説明

例題は8queen、five houses puzzle、send+more=moneyの３種類である。

8queenのプログラムは３章のユーザ記述言語のプログラムを変換したものであり、継続方式、レイヤードストリーム方式のプログラムは[奥村 87]のものと同一である。

five houses puzzleとは次のような問題である。５人の異なった国籍の人が同じ通りに建てられた、異なった色に塗られた５つの家に住んでおり、職業、ペット、好きな飲物が異なる。イギリス人は赤い家に住む等の事実が知られている。これらの事実を満たすように、国籍、色等の割当を行う。問題の詳細については文献[Van Hentenryck 87]を参照されたい。図4.1にユーザ記述言語によるプログラムを示す。各国籍、色等に対応する25の変数に、家の番号を示す1から5の変数を割り当てる。組み込みの制約としてequalityを表す=:=と、隣同士であるという関係を表すためのplus_or_minusが存在する。継続方式によるプログラムは、述語memberを用いて非決定的に変数の値を定めてテストを行うPrologプログラムを[上田 85]の方法に従って変換したものである。レイヤードストリーム方式によるプログラムは、[1,2,3,4,5]の順列を生成することにより、変数5つの値の割当をまとめて行っている。

send+more=moneyは有名な覆面算の問題である。s,e,n,d,m,o,r,yの８つの変数に0から9までの数字を割り当てる。図4.2にユーザ言語による記述を示す。継続方式によるプログラムは、付録2のプログラムを文献[上田 85]の方式に従って変換したものであり、各桁について候補の生成を順次行って制約のチェックを行うというものである。このプログラムでは足し算の表を参照する述語addc99があり、この述語が４通りの異なるモードで呼ばれる。この述語に関しては人手による変換および最適化を行っている。レイヤードストリーム方式によるプログラムは、足し算の表の参照は行わず、各桁の数字の候補の生成を並列に行っている。

表4.1.測定されるパラメータの意味

リダクション数	実行が終了するまでに行われたcommit operationの数
サスペンション数	各サイクルでcommitできなかったゴールの総数
サイクル数	ゴールのうちcommitできるものをすべてcommitするという処理を1サイクルとし、実行が終了までの処理の繰り返しの数
並列度	リダクション数をサイクル数で割ったもの。各サイクル平均いくつのリダクションが起こったかの目安

```
domain five({1,2,3,4,5}).
five([N1,N2,N3,N4,N5,C1,C2,C3,C4,C5,P1,P2,P3,P4,P5,
      A1,A2,A3,A4,A5,D1,D2,D3,D4,D5]) :-
   N1=:=C2,N2=:=A1,N3=:=P1,N4=:=D3,N5=:=1,D5=:=3,
   P3=:=D1,C1=:=D4,P5=:=A4,P2=:=C3,C1=:=C5+1,
   plus_or_minus([A3,P4],1),plus_or_minus([A5,P2],1),
   plus_or_minus([N5,C4],1),
   alldifferent([N1,N2,N3,N4,N5]),
   alldifferent([C1,C2,C3,C4,C5]),
   alldifferent([P1,P2,P3,P4,P5]),
   alldifferent([A1,A2,A3,A4,A5])//
   alldifferent([D1,D2,D3,D4,D5]),
   generate_left_to_right([N1,N2,N3,N4,N5,C1,C2,C3,C4,C5,
      P1,P2,P3,P4,P5,A1,A2,A3,A4,A5,D1,D2,D3,D4,D5]).

alldifferent([]).
alldifferent([X|Y]) :-
   outof(X,Y),
   alldifferent(Y).
outof(X,[F|T]) :-
   X=:=F,outof(X,T).
```

図4.1.ユーザ記述言語によるfive houses puzzleのプログラム

```
domain sendmory({0,1,2,3,4,5,6,7,8,9},{0,1}).
sendmory([S,E,N,D,M,O,R,Y],[R1,R2,R3,R4]) :-
   alldifferent([S,E,N,D,M,O,R,Y]),
   S=\=0,M=\=0,R1=:=M,
   R2+S+M=:=O+10*R1,
   R3+E+O=:=N+10*R2,
   R4+N+R=:=10*R3,
   D+E=:=Y+10*R4//
   generate_first_fail([S,E,N,D,M,O,R,Y,R1,R2,R3,R4]).
```

図4.2.ユーザ記述言語によるsend+more=moneyのプログラム

4.2.評価結果

実行時間、リダクション数等の測定結果を表4.2に示す。実行時間はSun3/260上のSICStus Prologで測定した結果である。この測定結果から、次のようなことが示される。

- 本方式はすべての例題に関して、実行時間、リダクション数とも最も少ない。
- 8queenは他の２方式との差が小さい。この理由は、8queenの制約はnon-equalityのみで、non-equalityのforward checkによる効果がequalityよりも小さいためである。
- レイヤードストリーム方式は、five houses puzzle, send+more=moneyの性能が悪い。この理由は部分解の生成を並列に行っているためである。探索木のノードの個数の評価で詳しく説明する。
- send+more=moneyでは継続方式で本方式に近い効率が得られている。この理由は、足し算の表を参照することにより、候補の生成を行う手続き中に制約が組み込まれており、制約の能動的な利用に近くなっているためである。
- 本方式は他の２方式と比較してサイクル数が多く、並列度が小さい。これは、制約を能動的に用いて探索空間を狭めていることによる当然の結果である。しかしながら、制約充足問題の並列度は、変数の数、領域の大きさが増加すれば組合せ的に増加するため、並列計算機上での実行を考えた場合、現実的な問題に対してはプロセッサ数よりも並列度が大きくなることは明白である。

表4.2. 例題の実行結果の比較

(a) 8queen

方式	Time	Red.	Sus.	Cyc.	Para.
本方式	1780	17533	2106	141	124.35
layered stream	2719	19418	2470	38	511.00
継続方式	3179	48543	0	133	364.98

(b) five houses puzzle

方式	Time	Red.	Sus.	Cyc.	Para.
本方式	740	9357	291	644	14.53
layered stream	38680	122948	23182	292	421.05
継続方式	5319	53944	0	207	260.60

(c) send+more=money

方式	Time	Red.	Sus.	Cyc.	Para.
本方式	2139	19659	3215	542	36.27
layered stream	33540	375690	83015	54	6957.22
継続方式	2480	47630	7042	116	410.60

表4.3に8queen、five houses puzzleの探索木の各レベルのノードの個数を示す。探索木のレベルとは、探索木の展開を行った回数である。レベルnのノードの個数は、8queenであれば制約を満たすn個のqueenの置き方に対応し、失敗となったノードは含まれない。send+more=moneyの評価は、３つの方式がそれぞれ異なる探索木の展開方法を用いているため割愛する。レイヤードストリーム方式によるfive houses puzzleは、５つの変数の値をまとめて決定しているため、探索木の展開は５回となっている。この結果から導かれることを示す。

表4.3. 探索木のノードの個数の比較

(a) 8queen

レベル	本方式	継続方式	layered
1	8	8	64
2	42	42	294
3	140	140	840
4	324	344	1720
5	362	568	2272
6	284	550	1650
7	182	312	624
8	92	92	92
total	1434	2056	7556

(b) five houses puzzle

レベル	本方式	継続方式	layred
1	1	5	
2	1	20	
3	1	60	
4	4	120	
5	3	24	408
6	9	120	
7	9	96	
8	18	288	
9	18	108	
10	14	12	6060
11	14	12	
12	42	12	
13	6	36	
14	6	72	
15	6	72	8424
16	5	72	
17	5	288	
18	15	300	
19	3	96	
20	3	15	6639
21	6	15	
22	5	60	
23	1	21	
24	1	6	
25	1	1	1
total	199	1931	21532

・本方式は探索したノード数が最も少ない。
・five houses puzzleの方が8queenよりも探索空間の削減の効果が大きい。
・レイヤードストリーム方式のfive houses puzzleは探索したノード数が特に多い。この理由は、レイヤードストリーム方式は部分解の生成を並列に行うためである。five houses puzzleは国籍（Nation）、色（Color）、職業（Profession）、動物（Animal）、飲物（Drink）に対応するリストの順列を生成し、その組合せを解の候補（部分解）としている。レイヤードストリーム方式では図4.3に示すような組合せの部分解を生成する。図中の数字は、その上の線で示されるような組合せの部分解の個数を示す。継続方式では、部分解の生成は逐次的であるため、例えば色－職業－動物といった組合せの部分解を生成することはない。この色－職業－動物の組合せの部分解のほとんどは国籍との制約から最終的な解にはならないものであり、結果的に無駄な処理をしている。

以上の結果をまとめる。
・forward checkilgにより、探索空間の削減が行われ、プログラムの実行効率が高くなる。
・探索空間の削減の効果は、five houses puzzleのように制約が複雑で不規則な問題で特に大きい。
・制約が不規則な問題に対して、レイヤードストリーム方式のように部分解の生成を並列に行うことは、無駄な探索を多く行う可能性がある。

その他、プログラムの性質として、本方式、継続方式によるプログラムでは、generateされたプロセスは

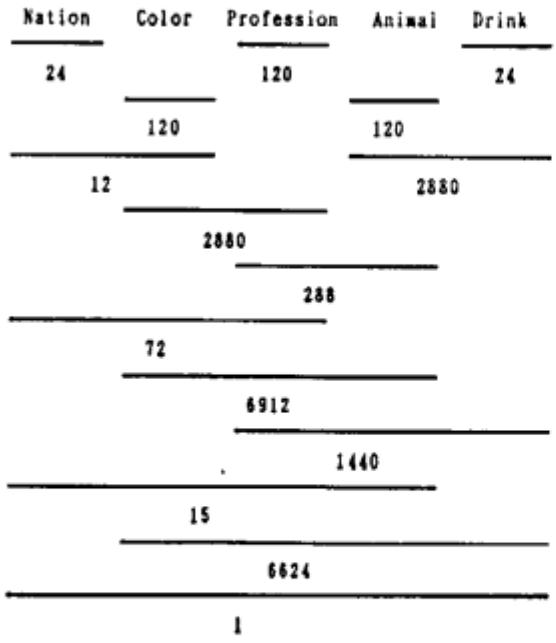

図4.3. five houses puzzle（レイヤードストリーム方式）の探索木のノードの分布

解の収集に用いる差分リスト以外の共有変数を持たない。一方レイヤードストリーム方式では多くのプロセスがストリームによって通信を行う必要がある。レイヤードストリーム方式のプログラムを並列マシン上で効率的に実行するためには通信のオーバーヘッドを少なくする工夫が必要である。

5．まとめと今後の課題

制約論理型言語を用いて、forward checkingにより制約を能動的に利用し、複数の可能性を並列に探索して、制約充足問題の全解を探索する方法を示した。さらに、制約論理型言語のプログラムを変換し、並列論理型言語のプログラムを得る方法を示した。これらの方法により、宣言的な制約の記述から、複数の可能性を並列に探索し、制約を能動的に用いて効率的に制約充足問題を解くことプログラムを得ることが可能になった。いくつかの例題について評価を行った結果、本方式で得られたプログラムは、探索空間を削減することにより、制約を受動的に用いるプログラムよりも実行効率が高いことが示された。

今後の課題として、制約評価の機能を強化することが挙げられる。ユーザ記述言語による制約の記述をそのままもちいるのではなく、[坂井 88]、[大木 88]等の制約評価の機能を用いて、等式、不等式の簡約化を行い、簡約化された式を用いてforward checkingを行うことについて検討を行っている。

謝辞

本研究の機会を与えて下さったICOT研究所の淵所長、NTT情報通信処理研究所知識処理研究部の村上部長に感謝致します。熱心に御討論頂いたICOT研究所の古川次長、長谷川第一研究室長に感謝致します。また、比較評価用のプログラムを提供して頂いた奥村研究員をはじめとするICOT第一研究室の皆様に感謝致します。

参考文献

[上田 85] 上田：全解探索論理プログラムの決定的論理プログラムへの変換、日本ソフトウェア科学会第２回大会論文集（1985）
[上田 86] 上田：全解探索論理プログラムの決定的論理プログラムへの変換（II）、日本ソフトウェア科学会第３回論文集（1986）
[大木 88] 大木、沢本、坂根、藤井：Sup-Inf法に基づいた制約論理プログラミング言語、日本ソフトウェア科学会第５回論文集（1988）
[奥村 87] 奥村、松本：レイヤードストリームを用いた並列プログラミング、 Proc. of The Logic Programming Conference (1987)
[坂井 88] 坂井、相場、：CAL:制約論理プログラミングの理論と実例、電子情報通信学会、SS87-28（1988）
[Dincbus 87] Dincbas, M., Simonis, H. and Van

Hentenryck, P. : Extending Equation Solving and Constraint Handling in Logic Programming, Internal Report IR-LP-2203, European Computer-Industry Research Centre (1987)

[Dincbas 88] Dincbas, M., Simonis, H. and Van Hentenryck, P. : Solving a Cutting-Stock Problem in Constraint Logic Programming, Int. Conf. on Logic Programming (1988)

[Jaffer 87] Jaffer, J., and Michaylov, S. : Methodology and Implementation of a CLP System Int. Conf. on Logic Programming (1987)

[Ueda 85] Ueda, K. : Guarded Horn Clauses, ICOT Tech. Report TR-103 (1985)

[Van Hentenryck 87] Van Hentenryck, P. and Dincbas, M.: Forward Checking in Logic Programming, Int. Conf. on Logic Programming (1987)

付録１．forward checkingによる4queenのGHCプログラム

```
q4(A) :- true |
   four_queen([[1,2,3,4],[1,2,3,4],
               [1,2,3,4],[1,2,3,4]],A,[]).

four_queen(L,S0,S1) :- true |
   check_list(L,0,S0,S1).

reduce(1,V,[Dx1,Dx2,Dx3,Dx4],D,Ln,Dn) :- true |
   Dx1n := V,                    reduce_sub(V,1,Dx2,Dx2n),
   reduce_sub(V,2,Dx3,Dx3n),     reduce_sub(V,3,Dx4,Dx4n),
   Ln=[Dx1n,Dx2n,Dx3n,Dx4n],     Dn := D+1.
reduce(2,V,[Dx1,Dx2,Dx3,Dx4],D,Ln,Dn) :- true |
   reduce_sub(V,1,Dx1,Dx1n),     Dx2n := V,
   reduce_sub(V,1,Dx3,Dx3n),     reduce_sub(V,2,Dx4,Dx4n),
   Ln=[Dx1n,Dx2n,Dx3n,Dx4n],     Dn := D+1.
reduce(3,V,[Dx1,Dx2,Dx3,Dx4],D,Ln,Dn) :- true |
   reduce_sub(V,2,Dx1,Dx1n),     reduce_sub(V,1,Dx2,Dx2n),
   Dx3n := V,                    reduce_sub(V,1,Dx4,Dx4n),
   Ln=[Dx1n,Dx2n,Dx3n,Dx4n],     Dn := D+1.
reduce(4,V,[Dx1,Dx2,Dx3,Dx4],D,Ln,Dn) :- true |
   reduce_sub(V,3,Dx1,Dx1n),     reduce_sub(V,2,Dx2,Dx2n),
   reduce_sub(V,1,Dx3,Dx3n),     Dx4n := V,
   Ln=[Dx1n,Dx2n,Dx3n,Dx4n],     Dn := D+1.
reduce_sub(V,D,Dx,Dxn) :- integer(Dx) | Dx=Dxn.
reduce_sub(V,D,[],Dxn) :- true | Dxn=[].
reduce_sub(V,D,Dx,Dxn) :- nointeger(Dx), Dx\=[] |
   V1:=V-D, V2:=V+D,
   delete(V,Dx,Dx1), delete(V1,Dx1,Dx2),
   delete(V2,Dx2,Dxn).

check_list(L,4,S0,S1) :- true | S0=[L|S1].
check_list(L,N,S0,S1) :- N<4 | check_list_0(L,L,N,S0,S1).
check_list_0([[]|_],_,_,S0,S1) :- true | S0=S1.
check_list_0([A|L1],L,  N,S0,S1) :- A\=[] |
   check_list_0(L1,L,N,S0,S1).
check_list_0([],    L,  N,S0,S1) :- true |
   check_list_1(L,L,1,N,S0,S1).
check_list_1([[D]|_],L,X,N,S0,S1) :- true |
   reduce(X,D,L,N,Ln,Nn), check_list(Ln,Nn,S0,S1).
check_list_1([D|L1],L,X,N,S0,S1) :- D\=[_] |
   X1 := X+1, check_list_1(L1,L,X1,N,S0,S1).
check_list_1([],    L,_,N,S0,S1) :- true |
   check_list_n(L,L,1,N,S0,S1).
check_list_n([D|_],L,X,N,S0,S1) :- nointeger(D) |
   recurse(X,D,L,N,S0,S1).
check_list_n([D|L1],L,X,N,S0,S1) :- integer(D) |
   X1 := X+1, check_list_n(L1,L,X1,N,S0,S1).

recurse(X,[],L,_,S0,S1) :- true | S0=S1.
recurse(X,[H|T],L,N,S0,S2) :- true |
   reduce(X,H,L,N,Lnn,Nn), check_list(Lnn,Nn,S0,S1),
   recurse(X,T,L,N,S1,S2).

delete(X,[X|Rest],R) :- true | R=Rest.
delete(X,[A|Rest],R) :- X>A, X<5 |
   delete(X,Rest,R1), R=[A|R1].
delete(X,[],R) :- true | R=[].
delete(X,[A|Rest],R) :- X<A  R=[A|Rest].
delete(X,[A|Rest],R) :- X>4| R=[A|Rest].
```

付録２．send+more=moneyの変換前プログラム（継続方式）

```
sendmory(S,E,N,D,1,O,R,Y) :-
   addc99(D,E,0,C1,Y),
   D=\=1, E=\=D, E=\=1, E=\=Y, D=\=Y, 1=\=Y,
   addc99(N,R,C1,C2,E),
   E=\=N, N=\=D, N=\=1, N=\=Y,
   E=\=R, N=\=R, D=\=R, 1=\=R, R=\=Y,
   addc99(E,O,C2,C3,N),
   E=\=O, N=\=O, D=\=O, 1=\=O, O=\=Y, O=\=R,
   addc99(S,1,C3,1,O),
   S=\=0, S=\=E, S=\=N, S=\=D, S=\=1, S=\=O,
   S=\=R, S=\=Y.
```