計算機アーキテクチャ　127－5
設　計　自　動　化　86－5
（１９９７．　１２．　１１）

# 非明示的列挙を用いたパタンマッチングについて

松永　裕介

（株）富士通研究所

〒 211-88 川崎市中原区上小田中 4-1-1

044-754-2663

yusuke@flab.fujitsu.co.jp

[概要]　テクノロジマッピングにおけるマッチングアルゴリズムは大別するとグラフによるパタンマッチングと論理関数の一致判定を用いたブーリアンマッチングに分けられる．前者は非常に高速であるが，一つの論理関数 (セル) に対して複数のパタングラフを必要とするなどの問題点がある．

そこで今回は，同一の論理関数に対してただ一つのパタンを用いてすべての可能なマッチングを列挙する新しいパタンマッチングのアルゴリズムの提案を行なう．このアルゴリズムはパタンを非明示的に列挙してマッチングを行なうもので，マッチング対象の回路パタンに合致しないパタンは全く考慮する必要がないので，従来の実際にパタンを全て列挙してからマッチングを行なう手法に比べて効率の良い処理が行なえる．

キーワード　論理合成，テクノロジマッピング，パタンマッチング

# A Pattern Matching Algorithm Using Implicit Enumeration

Yusuke Matsunaga

Fujitsu Laboratories LTD.

1015 Kamikodanaka, Nakahara-Ku, Kasawaki 211

044-754-2663

yusuke@flab.fujitsu.co.jp

[abstract] Matching algorithms for technology mapping are categorized into the following two groups — pattern matching based on graph manupulation and Boolean matching based on functional equivalence checking. Though the former is very fast, there is a problem that the number of patterns required may blow up for a large cell library.

In this paper, we present a novel matching algorithm that requires only a pattern for a function. This algorithm is based on implicit enumeration of matching patterns and it efficiently cut off partial matchings that has no hope to success.

**Keywords**　logic synthesis, technology mapping, pattern matching

# 1 はじめに

論理合成は大きくテクノロジ独立な合成／最適化処理とテクノロジ依存の最適化処理に分けることができる．後者はテクノロジマッピングと呼ばれ，テクノロジ独立な回路 (ブーリアン・ネットワーク) から，ライブラリ中のセルを用いて面積や遅延時間などの制約条件を満たした回路をつくり出す処理を行なう．

現在用いられているテクノロジマッピング手法の多くはマッピング対象の回路を一旦，2 入力 NAND ゲートのような小さなゲートで実現して，その 2 入力 NAND ゲートによって作られた部分回路 (クラスタと呼ばれる) に対して実際のセルを割り当てる，というアプローチをとっている．そこで，テクノロジマッピングの問題は，

1. 2 入力 NAND ゲートの部分回路にマッチするセルを列挙する．
2. 対象回路をカバーする最適なセルの組合せを求める．

という二つの問題に細分化される．

本 TM ではこのうちの最初の問題—セルのマッチング問題—について取り上げる．セルのマッチングアルゴリズムは大別するとグラフによるパタンマッチング [2, 3] と論理関数の一致判定を用いたブーリアンマッチングに分けられる．前者は非常に高速であるが，一つの論理関数 (セル) に対して複数のパタングラフを必要とするなどの問題点がある．一方，後者は論理関数の一致を見るため，パタングラフを必要としないが，2 つの論理関数が入力変数の位相反転および変数順序の入れ換えによって等しくなるかを確かめる必要があるため，単純な方法では $O(2^n \cdot n!)$ 通りの位相反転／変数順序変換を試さなければならず，たとえ，論理関数を二分決定グラフ (BDD)[4] を用いて表現したとしても効率の良い処理は行えない．

筆者は以前，探索の枝刈りを工夫することによって効率的なブーリアンマッチングを行なうアルゴリズムを提案している [1]．しかし，実際のテクノロジマッピングにおいて，マッチングを行なうためには，対象回路から可能なクラスタを全て切り出して，そのクラスタに対する論理関数を計算するという前処理が必要となるため，たとえブーリアンマッチング自体が高速化されても前処理のオーバーヘッドがかかってしまうという問題点がある．

その点，パタンマッチングではマッチングが成功した時にはじめて対象となるクラスタが切り出されるので，無駄なクラスタの列挙や論理関数の計算といったオーバーヘッドはかからない．そこで今回は，同一の論理関数に対して複数のパタンを必要とする，というパタンマッチングの欠点を克服した新しいパタンマッチングのアルゴリズムの提案を行なう．

# 2 従来手法

テクノロジマッピングの手法として広く用いられているものはパタンマッチングに基づくものである．この手法では，まず対象となる回路を 2 入力 NAND ゲートのような小さなゲート[1] を用いた回路へ変換する．以後，この回路は専ら単なる非巡回有向グラフ (directed acyclic graph : DAG) として扱われるため，主グラフ (subject graph) とも呼ばれる．また，割り当て対象のセルも同様に 2 入力 NAND ゲートを用いた等価回路へ変換する．このような回路はパタングラフ (pattern graph) と呼ばれる．パタンマッチングでは，主グラフの各部分にマッチするパタングラフを求める処理を行なう．その後，パタングラフに対応したセルに置き換えることで，与えられたライブラリのセルを用いた回路を得ることができる．例えば，図 1 の回路に対して，図 2 で与えられた 4 入力 NAND セルのマッチを調べると，図 3 に示したような部分回路が求められる．

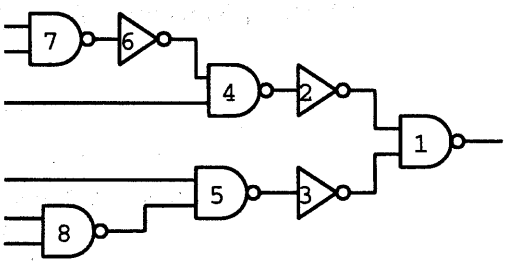



図 1: マッピング対象の回路

パタンマッチングは具体的には次のように行なわれる．まず，図 1 の主グラフの節点 1 を根とする部分グラフと図 2 のパタン 1 のマッチングを考える．双方の根の節点 1 と a はともに 2-NAND と同じタイプなのでマッチングすることが可能である．次に節点 1 のファンインの 2 つの節点を調べるとどちらの節点

[1] 2 入力 NAND を INVERTER のみで任意の組合せ論理回路を実現することができるため，2 入力 NAND は回路の最小構成要素と見なすことができる．この他，AND/OR/NOR などでも同様のことが言えるが，CMOS セルなどで最も基本的なセルということで NAND が良く用いられる．

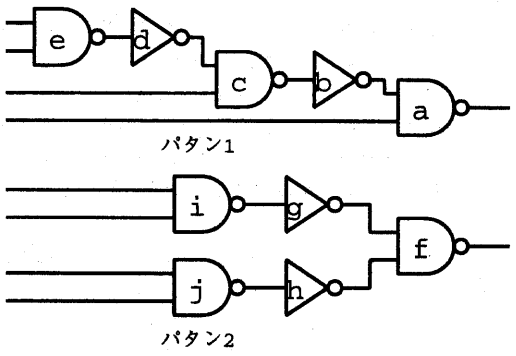



図 2: 4 入力 NAND セルに対するパタングラフ

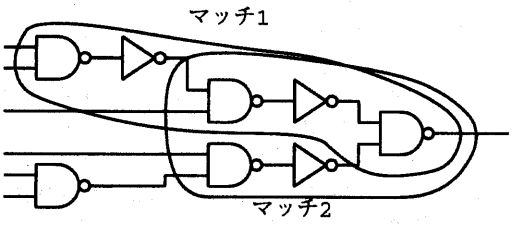



図 3: マッチングの結果

2, 節点 3 とも INVERTER となっている. 一方, 節点 a のファンインは節点 b は INVERTER, 他方は入力となっている. この場合, 節点 2, 節点 3 のどちらを節点 b とマッチさせるかで 2 通りが考えられる. ここでは, 節点 2 と節点 b をマッチさせて, 節点 1 から節点 3 へ至る枝とパタン 1 の入力をマッチさせる組合せを最初に試す. この時点でマッチが未確定な部分は節点 4 を根とする部分グラフと節点 c を根とする部分グラフである. この後, 同様の処理を続けることによって, (節点 4, 節点 c), (節点 6, 節点 d), (節点 7, 節点 e) というマッチ (図 3, マッチ 1) を得ることができる. 次に, 残っているもう一つの組合せを試す. 節点 1 から節点 2 へ至る枝をパタン 1 の入力のマッチは成功する. また, 節点 3 と節点 b, 節点 5 と節点 c のマッチも成功するが, 節点 5 のファンインは一方が入力, 他方が 2-NAND であるに対して, 節点 c のファンインは一方が入力, 他方が INVERTER なのでどのような組合せでもマッチさせることができない. そこで, 節点 3 と節点 b をマッチさせることができないことが分かる.

パタン 2 を用いたマッチングは, 考慮すべき組合せはただ 1 通り[2] で, (節点 1, 節点 f), (節点 2, 節点 g), (節点 3, 節点 h), (節点 4, 節点 i), (節点 5, 節点 j) というマッチ (図 3, マッチ 2) を得ることができる.

このようにパタンマッチングでは, グラフ構造のみに基づいてマッチングを行なうので非常に高速に処理することができる. 上に述べたような naive なアルゴリズムでは各節点のファンインをマッチさせる度に組合せの数が 2 倍に増えるため, 最悪の計算量はパタンのサイズには比例しないが, 実際のセルライブラリに対応するパタンの場合には対称なファンインが多く含まれているため, 実効的にはパタンのサイズに比例した手間でマッチングを行なえると言える.

ただし, 一つのセルに対してそれと機能的に等価なパタングラフは唯一ではないため, 複数のパタングラフを考慮する必要がある. 例えば, $n$ 入力 NAND セルを 2 入力 NAND セル (およびインバータ) で作ることを考えると, 出力側の最終段はもちろん 2 入力 NAND セルなので, その 2 つの入力にそれぞれ, $i$ 入力 AND, $n-i$ 入力 AND のパタンがつながった形から構成される. そこで, $n$ 入力 NAND セルを表すパタンの総数を $P(n)$ で表すことにすると, $P(n)$ は以下のような式で再帰的に定義される.

$$P(n) = \begin{cases} 1 & n = 0, 1 \\ P(1) \cdot P(n-1) & \\ \quad + P(2) \cdot P(n-2) & \\ \quad + P(3) \cdot P(n-3) + \cdots & n \geq 2 \end{cases} \quad (1)$$

で与えられる[3]. $n = 2 \ldots 20$ までの $P(n)$ は表 1 のようになる. このように 10 入力を越えるとパタン数が莫大なものとなる. また, AND-OR-INVERTER のような複合セルの場合はさらに組合せ的にパタン数が増えるので, 実用的なセルラリブラリのセルに対してその全てのパタンを列挙することは非効率的となる.

## 3 非明示的列挙によるパタンマッチングアルゴリズム

前節で述べたようなパタンマッチングの問題点は, パタングラフを 2 入力の節点に分解するところに根本的な原因があると考えられる. 主グラフの構成要

[2] 正確には節点 2 と節点 g をマッチさせる組合せと, 節点 2 と節点 h をマッチさせる組合せの 2 通りが存在するが, これらは対称であり区別できないので 1 通りと見なす.

[3] ただし, $n$ が偶数の場合には対称性を考慮して少しパタン数が減る. 具体的には $i = n-i$ の時のパタン数は $P(i) \times P(n-i)$ ではなく, $\frac{P(i) \times (P(i)+1)}{2}$ となる

| 入力数 | $P(n)$ | 入力数 | $P(n)$ |
|---|---|---|---|
| 2 | 1 | 11 | 207 |
| 3 | 1 | 12 | 451 |
| 4 | 2 | 13 | 983 |
| 5 | 3 | 14 | 2179 |
| 6 | 6 | 15 | 4850 |
| 7 | 11 | 16 | 10905 |
| 8 | 23 | 17 | 24631 |
| 9 | 46 | 18 | 56011 |
| 10 | 98 | 19 | 127912 |

表 1: $n$ 入力 NAND セルを表すパタン数

素を 2 入力 NAND のような小さなセルに限定して粒度を細かくすることは，さまざまな回路の実現構成を探索するために必要であるが，パタングラフまで同様な細かいセルで構成する必要性はない．歴史的に，従来手法で用いているマッチングアルゴリズムが，同じタイプのグラフに対してのみ適用可能なものであったため，主グラフと同じタイプということでパタングラフも 2 入力 NAND セルを用いて表されてきたのである．つまり，マッチングアルゴリズムが適用可能であるならばパタングラフのタイプに制限はない．

そこで，本節ではパタングラフとして入力数に制限を持たない一般の DAG を用いたパタンマッチングアルゴリズムを提案する．このアルゴリズムは，いわば非明示的に複数の (2 入力 NAND に分解された) パタングラフに対するマッチングを同時に行なうもので，計算量，使用記憶容量ともに従来のパタンマッチングアルゴリズムよりも効率のよいものである．

### 3.1 $n$ 入力 NAND とのマッチング

ここでは，まず，$n$ 入力 NAND の節点と 2 入力 NAND のグラフとのマッチングについて説明する．一般のパタングラフはこの $n$ 入力 NAND と INVERTER によって構成されるので，$n$ 入力 NAND 節点のマッチングは全てのマッチングの基本となっている．

前述の様に，$n$ 入力 NAND を表す 2 入力 NAND のパタンは $P(n)$ 通り存在する．2 入力 NAND で構成された主グラフに対して $n$ 入力 NAND セルとのマッチングを単純に列挙するなら，この $P(n)$ 通りのパタンを列挙し，その各々のパタンと与えられた主グラフとの (明示的な) マッチングを行なう必要がある．もしも $n$ 入力 NAND にマッチする部分グラフが存在しない場合には $P(n)$ 通りのマッチングはすべて失敗するし，たとえマッチが存在する場合でも $P(n)-1$ 通りのマッチングは失敗してしまう．つまり，もともと主グラフの部分グラフとして現れないパタンを列挙してマッチングを試すことは全くの無駄となっている．

この $P(n)$ 通りのパタンが互いに規則性や類似性を持たない場合には，このように各々のパタンを列挙してマッチングを試す以外に方法はないと考えられるが，$n$ 入力 NAND のパタンの場合には非常に単純で明解な規則性が存在する．それは，"$n$ 個の入力を持つ木でかつ，根から葉に至る (出力側から入力側へ至る) いかなる経路上にも INVERTER と 2-NAND が交互に現れる．" というものである．実は，$n$ 入力 NAND のマッチングを行なうにはこの規則だけを知っていれば良く，具体的なパタンの列挙は必要ない．この規則を用いたマッチングアルゴリズムは次のようになる (図 4)．

このアルゴリズムを用いて主グラフ上の節点 $r$ を根とするマッチを見つけるには，最初に境界の節点集合を表すスタック $F$ に主グラフの節点 $r$ を push して，FindNandMatch($F, n$) を呼べばよい．各再帰ステップ中で，境界スタックの先頭の節点 $s$ が取り出され，まずその節点を入力にマッチさせて残りのマッチを試し，次に $s$ を内部節点としてさらにファンイン側へ境界を広げてから残りのマッチを試す．このような処理によって節点 $r$ を根とする部分木でかつ，全ての内部節点が INVERTER，2-NAND の順で現われるものを列挙することができる．そこで，そのようなもののうち，入力数が $n$ に等しいものを記録することで，$n$ 入力 NAND にマッチする部分グラフを列挙することができる．

この探索を高速化するために 2 種類の枝刈り手法が考えられる．一つは，現在の探索状態から作られる部分グラフの入力数の下限が $n$ を越えたらそれ以上の探索を行なわないというもので，入力にバインドしている節点の数 + 境界スタックに積まれた節点の数がその下限となる．もう一つは，あらかじめ主グラフの各節点に対して，その節点を根としてマッチする NAND(もしくは AND[4] ) パタンの入力の最大値を記録しておき，その値が $n$ を越えないことが分かった時点で探索をやめるというものである．節点 $t$ の 2 つのファンインを $u$, $v$ として，$u$, $v$ それぞれにマッチする NAND(AND) パタンの最大値をそれぞれ $N(u)$,

[4] その節点が INVERTER の場合には AND にマッチする．

```
FindNandMatch(Stack of Nodes F, int n) {
    if (n = 0) {
        /* マッチが求まった. */
        現在のバインドを記録する;
        return;
    }
    主グラフの節点 s ← F.Pop;
    /* まず s を境界 (入力) にする. */
    s を n 番目の入力にバインドする;
    FindNandMatch(F, n − 1);
    s のバインドをはずす;
    if (s が INVERTER ではない) {
        goto end;
    }
    t ← s のファンイン
    if (t が NAND ではない) {
        goto end;
    }
    u, v ← t のファンイン;
    /* s を内部節点とするマッチを探す. */
    F.Push(u);
    F.Push(v);
    FindNandMatch(F, n);
    F.Pop;
    F.Pop;
 end:
    F.Push(s);
}
```

図 4: $n$ 入力 NAND に対するマッチングアルゴリズム

$N(v)$ とすると，$t$ にマッチする NAND(AND) パタンの最大値 $N(t)$ は，

$$N(t) = N(u) + N(v) \tag{2}$$

で計算できる．

## 3.2 一般のパタングラフに対するマッチングアルゴリズム

前節で提案したマッチングアルゴリズムを核として，一般のパタングラフと主グラフのマッチングを行なうアルゴリズムを考えることができる．まず，マッチングの前準備としてパタングラフを根から深さ優先順序でたどり，先行順に枝 (ファンアウト先の節点とファンイン元の節点のペア) を記録しておく．以後これを枝リストと呼ぶ．この枝リストを用いてマッチングを行なうには，枝リストから一つずつ枝をとりだし，枝の両端の節点に対して主グラフ上の節点を矛盾なくバインドさせてゆけばよい．枝リストは根から深さ優先探索でたどった時の先行順なので，必ずファンアウト先の節点には対応する主グラフ上の節点がバインドしており，その節点のファンインの節点のみを対象に探索すれば良い．以上のアルゴリズムの骨子を図 5 に示す．これは説明のため，なんの効率化も行なわない単純なアルゴリズムとなっている．次節でいくつかの高速化手法を説明する．

```
FindMatch(edge-list E) {
    if (E is empty) {
        /* マッチが求まった. */
        現在のバインド情報を記録する;
        return;
    }
    edge e ← E の先頭;
    p_root ← e のファンアウト先;
    p_leaf ← e のファンイン元;
    s_root ← p_root に対応する主グラフ上の節点;
    FindNandMatch を用いて s_root を根とする
    マッチ M を列挙する;
    for each match m ∈ M {
        for each leaf s_leaf in m {
            p_leaf と s_leaf をバインドする;
            FindMatch(E − e);
        }
    }
}
```

図 5: マッチングアルゴリズム

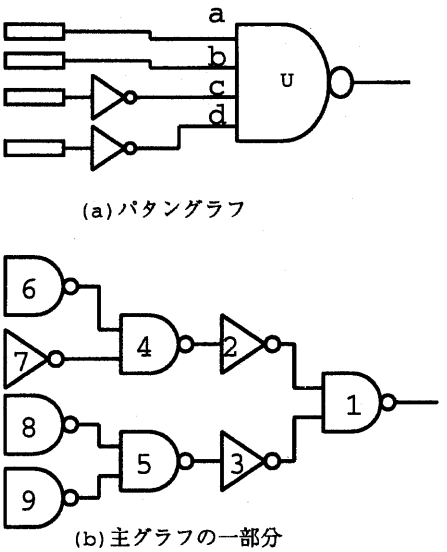
アルゴリズム **FindMatch** は前述の用に枝リストから一つずつ枝を取り出して，そのファンイン側の節点を主グラフ上の節点と結び付けてゆく．もしもパタングラフも 2 入力 NAND で構成されている場合には，**FindNandMatch** の返すマッチは高々1 つとなり，**FindMatch** は従来のパタンマッチングアルゴリズムと等価となる．一般には一つの節点を根として複数のマッチが考えられるので (図 3)，その各々について探索を行なう必要がある．また，一つのマッチについてもどの葉 (パタングラフ上の入力) に対応付けるかによって異なる場合があるので，それぞれの場合を試している．もしも主グラフおよびパタングラフが純粋な木である場合には，$p\_leaf$ と $s\_leaf$ の

バインドは常に成功するが，DAG の場合には矛盾を生じる場合があるため，新たなバインドを行なう際にはチェックが必要となる．

### 3.3 対称性を利用したマッチングの高速化

例えばパタングラフ上の節点 $p$ のファンインが全て等価な部分グラフである場合には，アルゴリズム **FindMatch** 中において，一つの *s_leaf* に複数の *p_leaf* のバインドを試すことは無駄であり，どれか一つの *p_leaf* を試せば良い．パタングラフも 2 入力 NAND のみで構成されている場合のマッチングでは，各節点が高々2 つしかファンインを持たないため，対称か否かの 2 種類の場合のみを考えれば良く，対称の場合には 1 通りの組合せを試せば良い，という非常に単純なアルゴリズムで対称性を考慮できたが，パタングラフの節点が任意の入力数のファンインを持つ場合には，そのファンイン全てが対称の場合と全てが対称でない場合以外にもさまざまな場合が考えられるので，その対応は容易ではない．このような処理を行なうために，パタングラフの各節点に対して，そのファンインの枝を等価なグループに分け，同一グループ内で優先順位を付けて保持させることとした．また，固定的な枝リストを用いて枝を取り出すのではなく，各節点を訪れる度に，その時点で最も優先順位の高いファンイン (各等価グループに対して一つずつ存在する) を動的に選ぶようにすることで対称性を考慮した枝刈りを行なっている．

図 6 の例を用いて説明を行なう．パタングラフ (同図 a) の節点 $u$ が 4 つのファンインを持ち，そのうち $a$ と $b$，$c$ と $d$ がそれぞれ等価とする．そこで，$a \prec b$，$c \prec d$ という優先順位を持たせておく．この節点は主グラフ (同図 b) 中の部分グラフ $\{1,2,3,4,5\}$ にマッチするので，$a,b,c,d$ の 4 本の枝が，$6,7,8,9$ へ至る枝とバインドされることになり，$4! = 24$ 通りの組合せが存在する．事実，図 5 のアルゴリズムでは全てのマッチを列挙することになってしまう．しかし，実際には $(a,1),(b,2)$ という割り当てと $(a,2),(b,1)$ という割り当ては全く等価であり，どちらか一つを試せばよい．このような対称性を考慮するために，パタングラフの枝を一つ選んでからそれに主グラフの枝を対応させるというやり方をやめ，主グラフの枝を一つ選んでから，対応するパタングラフの枝のうち優先順位の高いものを選ぶ，というアルゴリズムを用いる．つまり，節点 $u$ と $\{1,2,3,4,5\}$ がマッチすることが分かったら，まず，主グラフの枝 (例えば 4–6) を選ぶ．これに対して，パタングラフの枝のうち，まだ未選択でかつ優先順位の高いものを選ぶ (ここでは $a$ と $c$)．また，例えば，枝 4–7 と $a$ がバインドされていたとすると，パタングラフの枝としては $b$ と $c$ が選ばれる．つまり，$b$ や $d$ はそれぞれ $a$ や $c$ よりも先には選択されない，という規則を設けることによって，無駄な探索を避けることができる．この例では，わずか 6 通りの探索を試すだけですべての場合を尽くすことができる．



(a) パタングラフ

(b) 主グラフの一部分

図 6: 対称性のあるパタン

### 3.4 非明示的列挙を用いたマッチングの高速化

図 5 のアルゴリズムでは，**FindMatch** 中で主グラフの節点 *s_root* に対して，パタングラフの節点 *p_root* とのマッチを全て求め ($=\mathcal{M}$)，そのマッチの入力を一つ一つバインドして残りのマッチングを行っているように説明してきたが，この処理は効率的ではない．例えば図 7 の例で，(a) のパタングラフを (b) の主グラフにマッチさせることを考える．

パタングラフの根の節点 (4 入力 NAND) に対するマッチのうち，枝 4–6 を入力とするものは図中で示したマッチ 1 のマッチ 2 の 2 つである．パタングラフの枝 $a,b,c,d$ はどれも等価なので，例えば $a$ を 4–6 にバインドしてみると，$a$ のファンイン元の節点は NAND であるのに対して節点 6 は INVERTER であるので

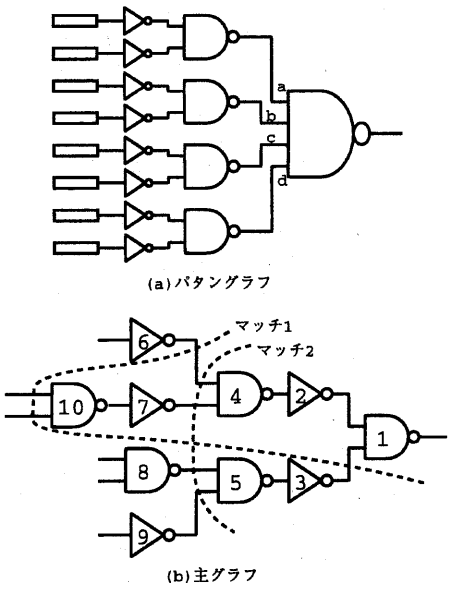



図 7: マッチングの例

マッチしないことがわかる．つまり，マッチ 1，マッチ 2 に対しては残りの入力に対するバインドを試すまでもなくマッチングが成功しないことがわかる．さらに，このマッチングが不成功に終る原因は枝 $a$ と 4–6 がマッチしないということのみにあるので，実はマッチ 1 やマッチ 2 といった 4 入力 NAND に対する完全なマッチングを求める必要すらないことがわかる．**FindNandMatch** によって完全なマッチングを求めてから入力側の節点のマッチングを行なうのではなく，**FindNandMatch** 中で一つのバインディングを求めるごとに入力側の節点のマッチングを試すようにすれば，完全なマッチングを列挙することなく望みのない探索を枝刈りすることが可能となる．直観的には図 4 のアルゴリズム **FindNandMatch** において一つバインドするごとに **FindMatch** を (**FindNandMatch** の代わりに) 呼ぶようにすれば良い．ただし，そのようにすると **FindNandMatch** が処理中の節点が次々と変わってゆくので，バックトラックのために節点を入れるスタックが必要となる．また，境界スタック $F$ は各パタングラフの節点ごとに必要となる．以下では節点 $p$ の境界スタックを $F_p$ で表す．このように改良されたマッチングアルゴリズムを図 8 に示す．

この非明示的列挙手法は，従来の 2 入力 NAND に分解してからパタンマッチングを行なう手法に比べて決定的に異なる点であり，計算時間，使用メモリ量のどちらにとっても効率の良い処理を行なうことが可能となっている．

## 4 実験結果

上記のアルゴリズムの有効性を確認するために実験を行なった．セルライブラリとしては論理合成にベンチマークとして広く用いられている lib2(`lib2.genlib`) と富士通のセルライブラリ cg61 を用いた．セルライブラリの諸元を表 2 に示す．

| ライブラリ | セル数 | 関数の数 | NPN 同値類の数 |
|---|---|---|---|
| lib2 | 29 | 28 | 14 |
| cg61 | 712 | 271 | 27 |

表 2: セルライブラリの諸元

パタンマッチングにしろブーリアンマッチングにしろ同一の NPN 同値類に属する関数 (パタン) に対するマッチングは同時に考慮することができるので，マッチングは NPN 同値類の代表関数 (任意に選ばれた関数) に対してのみ行なわれる．パタン数および 2 入力ノードに展開されたパタン数を表 3 に示す．

| ライブラリ | パタン数 | 2 入力パタン数 |
|---|---|---|
| lib2 | 15 | 18 |
| cg61 | 29 | 407 |

表 3: パタン数

lib2 の場合には 2 入力ノードを用いたパタン数がそれほど多くないが，cg61 の場合には多入力ノードを用いたパタン数の約 14 倍となっており，2 入力ノードのパタンへ展開することのオーバーヘッドが大きいことを示している．次に，これらのパタンを用いてベンチマーク回路に対してマッチングを求める実験を行なった．結果を表 4 に示す．

使用計算機は SUN-4/10，時間の単位は秒である．表中 n-pat が提案手法によるマッチング，2-pat が 2 入力ノードのパタンを用いた従来手法のマッチングを表している．表 3 のパタン数から予想されるように lib2 の場合には従来手法との差はあまり見られず，かえってアルゴリズムが複雑な分処理時間がかかっている場合がある．もっともその差は無視できるほど

| 回路 | 計算時間 | | | |
|---|---|---|---|---|
| | lib2 | | cg61 | |
| | n-pat | 2-pat | n-pat | 2-pat |
| 9symml | 0.20 | 0.23 | 0.33 | 2.57 |
| rot | 0.46 | 0.43 | 0.61 | 4.75 |
| C6288 | 1.35 | 1.01 | 2.02 | 13.91 |
| C7552 | 1.94 | 1.75 | 2.77 | 24.77 |
| des | 3.49 | 3.22 | 9.40 | 153.83 |

表 4: マッチングの実験

小さい．cg61 の場合は従来手法との処理時間の差は顕著である．des 以外の回路の場合，従来手法との計算時間の差が 6〜7 倍であるのに対して最も規模の大きな des のみその差が 16 倍になっている．

このように，実際のセルライブラリに対しては，従来提案されていた 2 入力ノードを用いたパタンに展開して，パタンマッチングを行なうという手法が明らかに非効率的であることがわかる．と同時に提案手法は効率的に処理しており，実用的であると言える．

## 参考文献


[1] Y. Matsunaga, "A New Algorithm for Boolean Matching Utilizing Structural Information", *IEICE Trans. Inf. & Syst.*, E78-D, No. 3, pp. 219–223, March 1995.

[2] K. Keutzer, "DAGON:Technology Binding and Local Optimization by DAG Matching" In *Proceedings of 24th DAC*, pp. 341–347, June 1987.

[3] E. Detjens and G. Gannot and R.L. Rudell and A. Sangiovanni-Vincentelli and A.R. Wang, "Technology Mapping in MIS" In *Proceedings of ICCAD*, pp. 116–119, November 1987.

[4] R.E. Bryant, "Graph-based algorithms for boolean function manipulation", *IEEE Transactions on Computer*, C-35(12), 1986.


```
FindMatch2(Stack of Nodes N) {
    パタングラフの節点 p ← TopOf(N);
    if (p の枝は全てバインドされている) {
        if (N が空) {
            /* マッチが求まった. */
            現在のバインドを記録する;
        } else {
            N.Pop;
            FindMatch2(N);
            N.Push(p);
        }
        return;
    }
    主グラフの節点 s ← F_p.Pop;
    /* まず s を境界 (入力) にする. */
    for each p の等価グループの代表枝 e {
        p_leaf ← e のファンイン元;
        if (p_leaf と s はマッチしない) continue;
        p_leaf と s をバインドする;
        if (p_leaf は内部節点) {
            N.Push(p_leaf);
            s のファンインを F_p_leaf に積む;
            FindMatch2(N);
            F_p_leaf を元に戻す;
            N.Pop;
        } else {
            FindMatch2(N);
        }
        p_leaf と s のバインドを解く;
    }
    /* s を内部節点にする. */
    if (s が INVERTER ではない) {
        goto end;
    }
    t ← s のファンイン
    if (t が NAND ではない) {
        goto end;
    }
    t のファンインを F_p に積む;
    FindMatch2(N);
    F_p を元に戻す;
end:
    F_p.Push(s);
}
```

図 8: 改良されたマッチングアルゴリズム