システムＬＳＩ設計技術　93－7
（1999. 11. 27）

複数の制約条件を考慮したテクノロジマッピングのアルゴリズムについて

松永　裕介

（株）富士通研究所

〒211-8588 川崎市中原区上小田中 4-1-1

044-754-2663

yusuke@flab.fujitsu.co.jp

あらまし　テクノロジマッピングのアルゴリズムとしては Keutzer の提案した tree-covering アルゴリズムに基づくものが広く用いられている．本稿では，tree-covering アルゴリズムの拡張である，遅延制約下での面積最小解を求めるアルゴリズムについての考察を行い，従来手法よりも効率のよいアルゴリズムを提案する．また，複数の遅延制約を考慮したアルゴリズムへの拡張に付いても述べる．

キーワード　論理合成，テクノロジマッピング，遅延制約，最適化

On A Technology Mapping Algorithm Considering Multiple Constraints

Yusuke Matsunaga

Fujitsu Laboratories LTD.

1015 Kamikodanaka, Nakahara-Ku, Kasawaki 211

044-754-2663

yusuke@flab.fujitsu.co.jp

Abstract　Tree-covering algorithm, which was proposed by Keutzer, is widely used as a basic algorithm for various technology mapping algorithm. This paper describes on technology mapping algorithm for minimizing area under delay constraints that is an extension of tree-covering algorithm, and proposes a novel efficient algorithm for that problem. An extension for handling multiple delay constraints is also discussed.

key words　logic synthesis, technology mapping, delay constraint, optimization

1 はじめに

論理的な仕様から回路のネットリストを生成する論理合成の処理は多くの場合，2つのフェーズ — テクノロジ独立な合成・最適化とテクノロジマッピング (テクノロジ依存最適化) — から構成される．前段のテクノロジ独立な合成・最適化処理では，テクノロジに依存した細かな遅延時間や面積などは考慮せずに，主に論理的な冗長性を取り除くことで初期回路を生成する．後段のテクノロジマッピングでは与えられた初期回路から回路面積や遅延時間を制約条件や目的関数として考慮して最適化を行う．本稿では，このテクノロジマッピングのアルゴリズムについての考察及び提案を行う．

テクノロジマッピングのアルゴリズムとしては Keutzer の提案した tree-covering に基づくものが広く知られている [3]．このアルゴリズムは動的計画法 (dynamic programming) を用いて，部分回路に対する最適解を記録して行くことによって，回路全体の最適解を求めるというものである．Keutzer の提案したアルゴリズムは与えられた初期回路に対する面積最小解のマッピングを求めるものであるが，その後，拡張として遅延最小解や与えられた遅延制約のもとでの面積最小解を求めるアルゴリズムが提案されている [2, 4, 5]．もしもゲートの遅延時間がゲートの駆動する負荷に依存しない遅延モデルを仮定した場合には，遅延最小のマッピングは面積最小のマッピングとほぼ同様に求めることができる．遅延制約のもとでの面積最小マッピングの場合，回路全体に対する遅延制約は一意に決められているが，部分回路に対する遅延制約は他の部分との兼ね合いで決まるものであり，各部分回路に対する最適解を求めるときには定まっていない．そこで，各部分回路に対して複数の最適解を記録しておく必要がある．Touati の提案した手法 [4] は，各部分回路における遅延制約の範囲を，その部分回路における面積最小解と遅延最小解から求め，その範囲をあらかじめ設定した N 個のスロットに区切り，各スロット (遅延制約) ごとの面積最小解を記録するというものである．このアルゴリズムではスロット数によって解の精度と計算時間のトレードオフを取ることができるが，実際には事前に適切なスロット数を予測することは難しい．一方，Chaudhary と Pedram は各部分回路において他のいずれの解よりも面積または遅延のどちらか一方が優れている解 (極小解) をすべて記録することで最適解を求めるアルゴリズムを提案している [5]．Chaudhary と Pedram のアルゴリズムは完全であるが[1]，Touati のアルゴリズムに比べて多くの計算量を必要とする．特に，極小解の数が増えるとその傾向は顕著となる．

本稿では，Chaudhary と Pedram のアルゴリズムに対して計算量的に優位な改良アルゴリズムを提案し，さらに複数の遅延時間制約を考慮するための拡張について述べる．以下，2章で tree-covering とその拡張アルゴリズムについて説明し，続く3章で Chaudhary と Pederam のアルゴリズムの改良アルゴリズムについて述べる．4章で複数の遅延時間制約を考慮するための拡張および，効率化手法について述べる．最後に5章で実験結果を示し考察を行う．

[1] 彼らの論文のタイトルが Near Optimal となっているのは負荷に対する影響を近似しているためと思われる．

2 tree-covering アルゴリズムとその派生アルゴリズム

2.1 tree-covering アルゴリズム

本稿ではテクノロジマッピング問題を以下のように定義する．

- 入力:
  - テクノロジ独立な組み合わせ回路 (ブーリアンネットワーク:Boolean network と呼ばれる)．
  - セルライブラリ：セル (論理ゲート) の集合
- 出力:
  - セルを割り当てられたネットリスト

tree-covering アルゴリズムは Keutzer が提案したテクノロジマッピングのアルゴリズムで，以下のような処理から構成される [3]．

1. 初期回路を木状回路 (tree) に分割する．
2. 各木状回路を 2 入力 NAND ゲートと NOT ゲートを用いてマッピングする．
3. 入力側から回路をたどり，各ノードを実現するもっともコストの低い実現を求める．
4. 出力側から回路をたどり，セルを選択して行く．

図 1(a) に木状に分割された初期回路を示す．この回路を 2 入力 NAND と NOT を用いてマッピングした例が同図 (b) である．任意の論理関数は NAND と NOT のみを用いて構成することが可能であり，任意の入力数の NAND は 2 入力 NAND と NOT の組み合わせで実現可能であるので，このようなマッピングは必ず存在する．この例では出力側の 3 入力 AND が，C と D を入力とした NAND+NOT に残りの入力との NAND

をとる形でマッピング (分解) されているが，これ以外にも分解の仕方が考えられる．一般に，このような 2 入力 NAND への分解方法は最終的なマッピングの解に影響を与えるが，ここではその問題については触れない．2 入力 NAND と NOT に分解された回路を**サブジェクトグラフ (subject graph)** と呼ぶ．

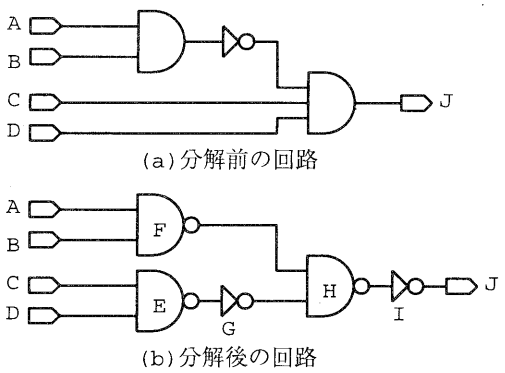

図 1: サブジェクトグラフの生成

一方，マッピングに用いられるセルも 2 入力 NAND と NOT を用いた形で表現される．図 2 にその例を示す．こちらは**パタングラフ (pattern graph)** と呼ばれる．サブジェクトグラフの場合と異なり，パタングラフの場合は可能なすべての分解を列挙する．つまり，一つのセルに対して複数のパタンが対応する場合がある．

テクノロジマッピングの問題とは，パタングラフを組み合わせてサブジェクトグラフと同型のグラフを作る (もしくはサブジェクトグラフを被覆する) 問題と見なすことができる．セルの面積をコストとした場合には，サブジェクトグラフを葉から根の方向にたどることで最適な被覆を求めることができる．これは，各ノード (2 入力 NAND と NOT) に対する最適解が，その入力側 (葉の側) のノードの最適解を用いて計算可能であることに起因する．もしもセルの遅延がその出力の負荷によらずに固定値であると仮定した場合には最大遅延をコストにしても同様に最適解を求めることができる．図 2 のセルを用いて図 1(b) に対する遅延最小のマッピングを行った例を図 3 に示す．図 2 においてセル名の下の $a \rightarrow f : 1$ の様な表記は入力 a から出力 f までの遅延が 1 であることを示す．図 3 の各ノードにはそのノードを実現可能なセル名とその場合の遅延時間が記されている．例えばノード H の場合，F と G を入力とした 2 入力 NAND を用いて実現した場合には出力の遅延は 8 となるが，F,C,D を入力とした 3 入力 NAND

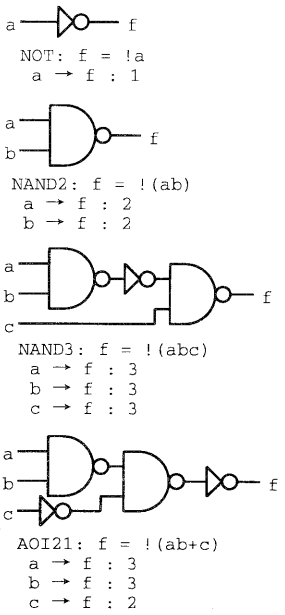

図 2: パタングラフの例

を用いて実現した場合には遅延は 9 となることを示している[2]．以降の処理ではノード H に対する最適解は 2 入力 NAND でその遅延値が 8 であることのみを記録しておけば良い．このように，tree-covering アルゴリズムではサブジェクトグラフのノードに比例した手間でマッピング結果を求めることができる．

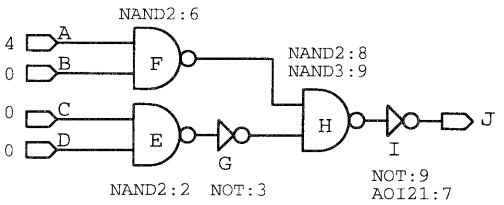

図 3: マッピングの例

[2] この例ではセル名のみでマッピングを一意に示すことができているが一般にはセル名＋入力ノードのリストが必要となる

2.2 遅延制約のもとでの面積最適マッピングアルゴリズム

テクノロジマッピングに対するもっとも現実的な要件は，回路の最大遅延をある一定値以内に抑えた上での面積最小解を求めるというものであろう．もしも，サブジェクトグラフの各ノードに対して遅延制約を前もって与えることができれば，その遅延制約のもとで面積最小解を求めることは tree-covering アルゴリズムを用いて行える．しかし，実際には遅延制約はそのノードの出力側の回路の実現に依存するため，入力側から出力側へ処理を行う tree-covering アルゴリズムでは前もって与えておくことはできない．Touati は各ノードが考慮すべき遅延制約の範囲として，そのノードにおける遅延最小解の遅延値と面積最小解の遅延値の間のみを考慮すれば良いことを指摘している．まず，遅延最小解の遅延値が下限であることは自明である．面積最小解の遅延値を上限とした理由は，これよりも遅い解がどのような遅延制約下でも面積最小解としては選ばれないことによる．Touati はこの範囲を K 分割し，各々の分割された範囲における面積最小解を記録することによって近似的に遅延制約下での面積最小解を求めるアルゴリズムを提案した [4]．分割数 K が定数であればこのアルゴリズムの計算複雑度はオリジナルの tree-covering と同じであるが，約 K 倍の計算量が必要となる．一方，解の精度を上げるためには K の値を大きくする必要があり，計算時間との兼ね合いで適切な K を定める必要がある．

Chaudhary と Pedram はこれとは異なるアルゴリズムを提案している．彼らの手法の特徴は，各ノードにおいて最適解になりうる可能性のある解はすべて記録しておく，というものである．あるノード n に対してセル g がマッチし[3]，そのセルが k 個の入力を持っているとする．この k 個の入力に対応するノード n_i が各々 N_i 個の解を持っているとすると，n の解としては，$\prod_{i=1}^{k} N_i$ 個の解の組合わせが考える必要がある．しかし，最大遅延時間は個々の入力の遅延時間の最大値によって決まるため，高々 $\sum_{i=1}^{k} N_i$ 種類しか存在しない．もしも同じ遅延時間を持つ解が複数存在したら，その中でもっとも面積の小さなもののみが考慮の対象となるので，ここで考えなければならない解の数は最悪で $\sum_{i=1}^{k} N_i$ となる．彼らの論文中ではノード n における解の集合を delay curve と呼んでいる．図 4 にノード n における解の集合を求めるアルゴリズム compute_delay_curve を示す．これは文献 [5] のコードを参考に書き直したものであるが，負荷容量に対する補正は除いてある．また，以下のように一部本文と矛盾する箇所は推測して修正を施した．文献 [5] 中では各マッチ g ごとに delay curve を求め，最後にその delay curve をマージすることで n に対する delay curve を求めると書いてあるが，アルゴリズムのコード中にはその記述がなかったので修正している．効率の面からも一つの delay curve に一つづつ追加するよりも各マッチごとに delay curve を求めてからマージするほうがよいと思われる．

```
function compute_delay_curve(n) {
1:  D = φ
2:  foreach n のマッチ g {
3:      D_g = φ
4:      foreach g の入力 i {
5:          foreach i の解 p_i {
6:              d = p_i の遅延 + i からの伝播遅延
7:              f = true
8:              foreach g の入力 j ≠ i {
9:                  d' = p_j の遅延 + j からの伝播遅延
                    が d よりも小さく，面積が最小
                    のものを求める．
10:                 if 存在した
11:                     それを p_j^* とする．
12:                 else
13:                     f = false
                }
14:             if f == true
15:                 (g, p_i, p_j^*) を解として
                    D_g に追加する．
            }
        }
16:     D_g を遅延値に従ってソートする
17:     D_g から他の解よりも面積，遅延が劣る解
        を削除する．
18:     D_g を D にマージする．
    }
}
```

図 4: compute_delay_curve アルゴリズム

アルゴリズム自体は単純である．前述のように 4,5 行目のループで $\sum_{i=1}^{k} N_i$ 個の遅延値を列挙し，その遅延制約を満たす解の中の面積最小解を 9 行目で求めている．これをソートして，極小解となっていないものを削除することで最終的な delay curve を求めている．

[3] 以降，同様の意味でノード n に対するマッチ g という表記を用いる．

計算複雑度に関しては文献 [5] 中に一つのマッチに付き $O(N^2\log(N)k\log(N_m))$ (ただし $N=\sum_{i=1}^{k}N_i$, $N_m = \max_{i=1}^{k}N_i$) という記述があるが，これは $O(Nk\log(N_m)+N\log(N))$ の誤りではないかと思われる．4,5 行目のループの回数が $O(N)$ であり，その中で k 個の入力に対して面積最小解を求める手間が $O(\sum_{i=1}^{k}log(N_i))=O(k\log(N_m)$ である．また，16 行目のソートが $O(N\log(N))$ となる．17 行目の極小解以外の解の削除は $O(N)$ で行える．18 行目のマージは 2 つの delay curve の要素数をそれぞれ N_1, N_2 とすれば $O(N_1+N_2)$ で実行できる．

3 delay curve を求める改良アルゴリズム

3.1 基本アルゴリズム

ここでは一つのマッチに対する delay curve を $O(N'k)$ で求める改良アルゴリズムを提案する．ただし，N' は結果として得られる delay curve の要素数であり，常に $N'<N=\sum_{i=1}^{k}N_i$ である．

まず，各入力 i に対する解の順列 (delay curve)D_i は面積の昇順にソートされているものとする．delay curve 中に含まれる解は他の解に比べて面積と遅延時間のどちらかはまさっているはずなので，面積の昇順にソートされた場合には遅延時間は降順に並んでいることになる．この状態で遅延制約を考慮しない面積最小解は各々の入力から先頭の要素を選びだすことで求めることができる．計算複雑度は明らかに $O(k)$(k は入力数) となる．提案する改良アルゴリズムのポイントは $\{D_1,\ldots,D_k\}$ から遅延制約に違反している要素を削除した部分順列 $\{D'_1,\ldots,D'_k\}$ を作り，そこでの面積最小解を $O(k)$ で求めるところにある．

解の順列 (delay curve)D_i から遅延制約 d を満たすものだけを取り出した部分順列を $S(D_i,d)$ と表すこととする[4]．各 D_i は遅延に関しては降順にソートされているので，2 つの遅延制約 d_1 と d_2 の間に $d_1<d_2$ の関係があった場合には，$S(D_i,d_1)$ は常に $S(D_i,d_2)$ の部分列となっている．もしくは，$S(D_i,d_2)$ の先頭からから 0 個以上の要素を取り除いたものが $S(D_i,d_1)$ となっている．つまり，制約条件のない D_i からはじめて，もっとも大きな遅延値を持つ要素 (=先頭の要素) を削除した順列を新たな D_i にする，という処理を繰り返すことですべての遅延制約に対応する解の部分順列を列挙することができる．

[4] 正確には D_i は対象のマッチの入力の解なので，入力における遅延時間に当該のマッチの伝播遅延時間を加えたものが考慮すべき遅延時間となる．具体的には図 4 の 6 行目の処理を行って d を計算し，それと遅延制約を比較する．

```
function compute_delay_curve2(n) {
1:  D = φ
2:  foreach n のマッチ g {
3:      D_g = φ
4:      while (1) {
5:          foreach g の入力 i
6:              p_i = D_i の先頭の要素
7:          (g, p_i) を解として D_g の末尾に追加する．
8:          foreach g の入力 i
9:              if i からの遅延が最大遅延 {
10:                 D_i から p_i を削除
11:                 if D_i が空
12:                     goto end
                }
        }
13: end:
14:     D_g を D にマージする．
    }
}
```

図 5: 改良 compute_delay_curve アルゴリズム

以上をまとめたアルゴリズムを図 5 に示す．前述のように各 D_i 中で遅延時間が最大のものは先頭の要素なので D_i をリストで実装すれば削除は定数時間で行える．計算複雑度に関するポイントとしては 2 点ある．1 点目はこのアルゴリズムで生成された D_g はすでにソート済の極小解の順列となっているということである．そのため，図 4 の 16,17 行目のような処理は必要ない．2 点目は 11 行目で一つでも空の D_i ができた時点で処理を終えることである．このことにより 4 行目のループは正確に $|D_g|$ 回だけ回ることになる．ループ中では各入力ごとの処理が定数時間で行われるだけなので $O(k)$ となり，一つのマッチに対する delay curve D_g を求めるのに必要な計算複雑度は $O(|D_g|\times k)$ となる．$|D_g|\le\sum_{i=1}^{k}|D_i|$ であり，このアルゴリズムは常に compute_delay_curve よりも効率よく delay curve の計算を行える．

3.2 高速化手法

一つのマッチに対する delay curve を求める限り，前述の改良 compute_delay_curve アルゴリズムは効率的

でありこれ以上改善の余地はないと思われるが，複数のマッチに対する delay curve をマージしてノード n に対する delay curve を求める，という点に関しては改善の余地がある．各 D_g の要素は D_g の中では極小解であることが保証されているが，すでに他のマッチから作られた delay curve D に対しても同様に極小解になるとは限らない．そこで，D の内容を考慮して，最終的な極小解になる可能性のない要素を除外する手続き prune_delay_curve を図5のアルゴリズムの2行目のあとに追加して，$\{D_1,\ldots,D_k\}$ を縮小することを考える．この手続きは以下のようになる．

1. $\{D_1,\ldots,D_k\}$ の各先頭要素を選びだし，マッチ g の入力の解として選んだときの面積 A_0 を計算する．
2. 現在の delay curve D の要素の中で面積が A_0 よりも大きく，かつ，遅延時間が最小のものを求める．その遅延時間を T_0 とする．
3. $\{D_1,\ldots,D_k\}$ の中で，遅延時間 (g による伝播遅延を足したもの) が T_0 よりも大きなものを削除する．
4. $\{D_1,\ldots,D_k\}$ が変化していたら (削除された要素があったら) 1 に戻る．

ステップ1で求める A_0 は現在の $\{D_1,\ldots,D_k\}$ から作られる面積の最小値である．delay curve D の中で A_0 を越えない最小の遅延時間が T_0 であるとすると，少なくとも遅延時間が T_0 よりも遅い解はその面積がなんであろうと (A_0 よりは大きいので)D の中の極小解を構成する要素とはなり得ない．そこで，そのような要素を取り除く．この削除によって A_0 が変わる可能性があるので変化があるかぎり処理を繰り返す．この処理によって D の極小解に関係のない要素を削除することができる．もしも $D_i=\phi$ となるような入力ができたらそれ以降の処理も必要なくなるので大きく効率化に貢献する．ただし，通常は減らした要素数に対して高々線形の削減効果しかない．一方，この prune_delay_curve は最悪の場合，$\sum_i^k |D_i|$ 回ループを回る可能性があり，ループ中で T_0 の検索のために二分探索を用いても $O(\log(|D|))$ を必要とする．計算複雑度としては効果的でない場合もある．

4 複数の制約条件を考慮したアルゴリズム

文献 [1] で述べたように，一つの組み合わせ回路に対して複数の遅延制約が課せられる場合がある．また，回路の遅延を立ち上がりと立ち下がりで分けて考えた場合には通常の遅延時間制約も2つの遅延制約の組み合わせと見なすことができる．本章では複数の遅延制約下での面積最小解を求めるアルゴリズムについて述べる．

図5のアルゴリズムを，2つの遅延制約を考慮するように単純に拡張すると図6のようになる．

大まかな構造は変わらないが，10行目以降の処理が大きく変わっている．残念ながら2つの独立な制約がある場合には任意の制約間に順序関係が定義できるわけではない．例えば，遅延1が d_1 以下で，遅延2が d_2 以下という制約を (d_1,d_2) と表すものとすると，$(10,5)$ は $(15,7)$ よりも厳しい制約だと言えるが，$(10,5)$ と $(5,10)$ の間にはどちらがより厳しい制約かという順序関係は存在しない．つまり，これらの制約によって delay curve の集合 $\{D_1,\ldots,D_k\}$ を制限した部分列集合の間に包含関係は成り立たない．そのため，改良 compute_dleay_curve アルゴリズムのように単純に極小解を列挙することができない．

見掛け上，6行目の while ループは1重のように見えるが実際には遅延2の制約を減少させるループと遅延1の制約を減少させる2重のループとなっている．内側の遅延2の制約を減少させる時には $\{D_1,\ldots,D_k\}$ を直接用いずにこれをコピーした $\{E_1,\ldots,E_k\}$ を用いる．このようにすることによって，遅延1と遅延2の2つの制約の組に対する面積最小解をすべて列挙することが可能となる．このような構造により6行目のループの回る回数は $O(N^2)$ となる．ただし，$N=\sum_{i=1}^k |D_i|$.

しかし，単一の遅延制約の場合と異なり，このアルゴリズムでは重複して同一の解が生成されることがあり，また，前述の理由からソートもされていない．さらに，2つの delay curve のマージが線形時間では行えないなどの問題があるため，計算効率的にはさまざまな問題点を抱えている．

5 実験結果

まず，compute_delay_curve アルゴリズムと改良 compute_delay_curve アルゴリズムのパフォーマンスの比較を行った．これらのアルゴリズムを内部に組み込んだテクノロジマッパを用いて MCNC'89 のベンチマークデータのマッピングを行った．ライブラリは富士通のゲートアレイ用セルライブラリを用いている．表1にその際の CPU を時間を示す．使用計算機は 500MHz Pentium-III(FreeBSD-3.3 メモリ 128MB) である．

表中，コラム 'D' は遅延最小解を求めるのにかかった時間を示している．コラム 'AD1' はこの遅延最小解を制約時間とした時の面積最小解を提案

表 1: compute_delay_curve アルゴリズムの評価

	D	AD1	AD2	AD3
C432	4.89	177.02	294.10	435.20
C499	3.96	57.60	65.93	70.88
C880	6.71	2428.66	2933.06	6766.21
C1908	7.65	172.29	273.26	434.99
C2670	14.85	1111.59	2123.32	4427.15
C3540	31.52	992.18	1427.68	2166.45
C5315	29.40	276.60	416.64	536.22
C6288	67.39	684.67	730.92	767.52
C7552	38.54	416.43	610.42	769.56
des	115.20	2881.72	4584.05	6581.06
rot	11.35	124.88	149.44	165.96
9symml	4.07	81.71	135.87	221.23
apex6	11.77	110.84	141.88	158.04

した改良 compute_delay_curve アルゴリズムで求めたときの CPU 時間 (単位は秒) を表している．ここでは prune_delay_curve を用いている．'AD2' は prune_delay_curve を用いずに提案した改良 compute_delay_curve アルゴリズムでマッピングを行ったときの CPU 時間を表している．最後の 'AD3' はオリジナルの compute_delay_curve アルゴリズムで求めた時間である．どちらもアルゴリズムを用いてもマッピング結果は同一のものが得られている．

実際のテクノロジマッピング処理では回路は木状に分割されず，また，負荷の影響を考慮した処理が行われるなど，純粋に tree-covering アルゴリズムの性能だけが CPU 時間に現れているわけではないが，提案した改良 compute_delay_curve アルゴリズムが計算時間の上で優位であることは明らかであり，ほとんどの場合にはオリジナルの compute_delay_curve アルゴリズムを用いた場合の半分以下の時間で処理を行っている．また，場合によっては prune_delay_curve ヒューリスティックが計算時間の削減に大きく役立つ場合があり，すべての例に対して計算時間の増加は見られなかった．一方，単純な遅延最適解を求める処理と比較してみると数十倍～数百倍も時間がかかっており，delay curve を計算するアルゴリズムをそのまま適用することが問題があることを示している．実際，これらの実験において delay curve の要素数は数百個になる場合が見られた．文献 [5] でも述べられているように，遅延や面積がある一定の範囲内に収まっている要素をまとめることによって，delay curve の要素数を減らすヒューリスティックが考えられる．

複数の制約条件を考慮したテクノロジマッピングに関しては，実装が十分でないため本格的な実験が行えていない．アルゴリズムの計算複雑度から考えて近似なしで厳密に計算することはあまり実用的とは思えないので，考慮すべき要素数を減らすヒューリスティックと合わせて実用的な適用方法を検討する必要があると思われる．

参考文献

[1] 松永 多苗子, 松永 裕介, 田宮 豊, "複数クロックを考慮したタイミング制約生成手法", デザインガイア'99

[2] R.L. Rudell, "Logic Synthesis for VLSI Design", PhD thesis, UCB/ERL M89/49, 1989.

[3] K. Keutzer, "Dagon: technology binding and local optimization by dag matching", In Proceedings of the 24th Design Automation Conference, pp. 341-347, Jun. 1987.

[4] H.J. Touati, "Performance-Oriented Technology Mapping", PhD thesis, UCB/ERL M90/109, November 1990.

[5] K. Chaudhary and M. Pedram, "A Near Optimal Algorithm for Technology Mapping Minimizing Area Under Delay Constraints", In Proceedings of the 29th Design Automation Conference, pp.492-498, Jun. 1992.

```
function compute_delay_curve3(n) {
1:  D = φ
2:  foreach n のマッチ g {
3:      D_g = φ
4:      foreach g の入力 i
5:          E_i = D_i
6:      while (1) {
7:          foreach g の入力 i
8:              p_i = E_i の先頭の要素
9:          (g, p_i) を解として D_g の末尾に追加する.
10:         foreach g の入力 i
11:             if i からの遅延 2 が最大遅延 {
12:                 E_i から p_i を削除
13:                 if E_i が空
14:                     goto end1
                    }
15:         continue
16:     end1:
17:         foreach g の入力 i {
18:             if i からの遅延 1 が最大遅延 {
19:                 D_i から p_i を削除
20:                 if D_i が空
21:                     goto end2
22:             E_i = D_i
            }
        }
23: end2:
24:     D_g から極小でない要素を削除する.
25:     D_g を D にマージする.
    }
}
```

図 6: 2 つの遅延制約用の compute_delay_curve アルゴリズム