JAIST Repository
https://dspace.jaist.ac.jp/

Title	FPGAを用いたNP完全問題の全解探索法に関する研究
Author(s)	山岸，洋平
Citation	
Issue Date	2003-03
Type	Thesis or Dissertation
Text version	author
URL	http://hdl.handle.net/10119/1711
Rights	
Description	Supervisor:中野 浩嗣，情報科学研究科，修士

JAIST
JAPAN
ADVANCED INSTITUTE OF
SCIENCE AND TECHNOLOGY

Japan Advanced Institute of Science and Technology

修　士　論　文

FPGAを用いたNP完全問題の全解探索法に関する研究

北陸先端科学技術大学院大学
情報科学研究科情報処理学専攻

山岸 洋平

2003年3月

修 士 論 文

FPGAを用いたNP完全問題の全解探索法に関する研究

指導教官　中野 浩嗣 助教授

審査委員主査　中野 浩嗣 助教授
審査委員　浅野 哲夫 教授
審査委員　金子 峰雄 教授

北陸先端科学技術大学院大学
情報科学研究科情報処理学専攻

110123 山岸 洋平

提出年月: 2003 年 2 月

Copyright © 2003 by Yamagishi Youhei

概 要

本稿では、n 個のブール変数 $x_1, x_2, ..., x_n$ の論理式 (例えば $f(x_1, x_2, x_3) = ((x_1 \to x_2) \vee \neg(\neg x_1 \oplus x_3)) \wedge \neg x_2$）が与えられたときに，充足可能な真理値割当が存在するか否かを判定する問題は充足可能性問題と呼ばれる．充足可能性問題は多くの問題が帰着できる重要な問題である（この問題の集合は NP 完全と呼ばれる）．本研究では充足可能性問題をハードウェアを用いて高速に解く手法の開発をめざした．ハードウェアを用いる理由は，問題の論理式はそのままが回路化でき，任意の真理値割当について論理式の値を高速に評価できるからである．

特に，真理値割当に現れる 1 の個数を制限した場合について考えた．つまり，n 個の変数のうち 1 を割当る変数の数を k 個に制限した充足解を求める．このように変更を加えた充足可能性問題の解をハードウェアで求めるには，n ビットの中で 1 が k 個含まれる割当を列挙する特殊なカウンタが必要である．本研究では，このカウンタの様々な実現方法を考案し，論理回路が再構築可能なハードウェアである FPGA に実装し性能評価を行った．

目 次

第 1 章　NP 完全問題と全解探索法 **3**
1.1　NP 完全問題の概要 3
1.2　提案手法 4

第 2 章　カウンタアルゴリズム **8**
2.1　${}_nC_k$ カウンタ 10
2.1.1　素朴な方法 13
2.1.2　右シフト法 13
2.1.3　左シフト法 15
2.1.4　k プロセッサ法 18
2.1.5　${}_nC_k$ カウンタに関するまとめ 19
2.2　${}_nC_{[0,k]}$ カウンタ 21
2.3　${}_nC_{[k,n]}$ カウンタ 26
2.4　${}_nC_{[i,j]}$ カウンタ 27

第 3 章　カウンタアルゴリズムの実装と評価 **31**
3.1　${}_nC_k$ カウンタの実装と評価 32
3.2　${}_nC_{[0,k]}$ カウンタの実装と評価 34
3.3　${}_nC_{[i,j]}$ カウンタの実装と評価 37

第 4 章　結論 **41**

はじめに

本論文では多項式時間では解けそうにない NP 完全問題の全解列挙問題を扱う．この問題を解く方法として全解探索法に着目し, 全解探索法を高速化するハード ウェアを用いた手法を提案し，実際に FPGA という論理回路が再構築可能なハード ウェアを用いて実装し, 評価を行う．

まず，NP 完全とは言語 L が NP に属して，かつ NP の中の任意の言語が帰着できるもっとも難しい言語のクラスである．NP 完全問題とは NP 完全に属する言語に対応する実際の問題であり，幅広い分野で発見されている．例えば, ブール論理, グラフ，算術, ネットワーク設計，集合と分割，格納と取り出し, 並べ変えとスケジューリング，数理計画，代数と数論，ゲームとパズル, オートマトンと言語論理，プログラム最適化，さらに多くの分野がある．この論文では代表的にブール代数とグラフの NP 完全問題を例にとり, そこで必要となるアルゴ リズムについて考察・提案する．

ブール代数の NP 完全問題の例として，n 個のブール変数 $x_1, x_2, ..., x_n$ の論理式 (例えば $f(x_1, x_2, x_3) = ((x_1 \rightarrow x_2) \vee \neg(\neg x_1 \oplus x_3)) \wedge \neg x_2$) が与えられたときに，充足可能な真理値割当が存在するか否かを判定する問題である充足可能性問題がある．本研究では充足可能性問題をハード ウェアを用いて高速に解く手法の開発をめざした．ハード ウェアを用いる理由は，問題の論理式はそのままが回路化でき，任意の真理値割当について論理式の値を高速に評価できるからである．

特に，真理値割当に現れる 1 の個数を制限した場合について考えた．つまり，n 個の変数のうち 1 を割当る変数の数を k 個に制限した充足解を求める．このように変更を加えた充足可能性問題の解をハード ウェアで求めるには，n ビットの中で 1 が k 個含まれる割当を列挙する特殊なカウンタが必要である．以下このカウンタのことを ${}_nC_k$ カウンタとよぶ．他にも真理値割当に可能な 1 の個数を k 個以下に制限した場合に必要な ${}_nC_{[0,k]}$ カウンタ，i 個以上 j 個以下に制限した場合に必要な ${}_nC_{[i,j]}$ カウンタが必要である．本研究では，これらのカウンタの様々な実現方法を考案し，論理回路が再構築可能なハード ウェアであるＦＰＧＡに実装し性能評価を行った．

次に，NP 完全問題の代表例を取り上げ，カウンタによる真理値割当を試していく方法を FPGA を用いて実装し, 性能評価を行った．

本論文の構成は以下のとおりである．

第 1 章では，NP 完全問題と本論文が提案したいハード ウェア手法について説明し, 第 2 章では，ハード ウェア手法で必要となるカウンタアルゴ リズムの様々な実現方法を提案

し, 第 3 章では，実際に FPGA を用いた実装について説明し, 性能評価を行う．第 4 章で，この手法についての結論を述べる．

第1章　NP 完全問題と全解探索法

本章では，まず，本論文で扱う NP 完全問題の定義をおこなう．次に，NP 完全問題の性質をしめし，本論文で提案したい手法ににについて説明する．そして，その代表例をしめし，本論文での主な結果であるカウンタアルゴリズムの有用性について説明する．

1.1　NP 完全問題の概要

定義 1.1. ある問題に対応する言語 $L \in \{0,1\}^*$ が

$$\text{言語 } L := \begin{cases} L \in \mathrm{NP} \text{ かつ} \\ \text{すべての } L' \in \mathrm{NP} \text{ に対して}, L' \leq_P L \end{cases}$$

を満たすとき, その問題のことを *NP* 完全問題と呼ぶ.

つまり，クラス NP に属する言語の中で最も難しい言語に対応する問題のことを NP 完全問題いう．

一般的に NP 完全問題は，多項式時間で動作する効率的なアルゴリズムは発見されることはないだろうと考えられている．その仮定がただしいなら, 近似アルゴリズムやヒューリスティックを使わず, 正しく NP 完全問題を解くには全解探索を行わなければならない．

そこで，本論文では単純に全解探索を行う方法をより高速に行うために，ハードウェアを用いた並列計算による手法を提案する．また，種々のヒューリスティックや，ランダムアルゴリズムでも，この手法が有用である可能性を示唆したい．

NP 完全問題の代表例として以下のようなものがある．

回路充足可能性問題 (CIRCUIT-SAT) And,Or,Not からなる論理回路の充足可能性の判定

充足可能性問題 (SAT) 論理式の充足可能性の判定

ハミルトン閉路問題 (HAM-CYCLE) 単純グラフ G がハミルトン的であるかの判定

頂点被覆問題 (VERTEX-COVER) 単純グラフ G が k 個以下の頂点で被覆できるかの判定

独立頂点集合問題 (INDEPENDENT-SET) 単純グラフ G において k 個以上の頂点を選んだときに独立である場合が存在するかどうかの判定.

他にも様々な分野で NP 完全問題が発見されている．詳細は [1] をみてほしい．

ここで, 注目すべきなのはグラフの NP 完全問題の一部は，頂点や辺をある制約のもとで選択して，ある条件を満たすかどうかを判定する必要性があることである．ハードウェアで全解探索をおこなって解を得る手法を考えたとき, n 個のの中からある制約のもとで組み合わせを生成する生成器が必要となる

1.2 提案手法

先ほどの，1.1 節でのハードウェアを用いて並列に計算する手法について 1.1 節で紹介した VERTEX-COVER を具体例にとって説明する．まず，VERTEX-COVER を定義する．

定義 1.2. *VERTEX-COVER* は，
単純グラフ $G(V,E)$(ただし, $|V|=n$) と自然数 $k \leqslant n$ が与えられたとき，

$$\Upsilon := \{V' \subseteq V | \forall (u,v) \in E, u \in V' \vee v \in V', |V'| \leqslant k\}$$

という集合 Υ が要素を持つか，持たないかを判定する問題である．

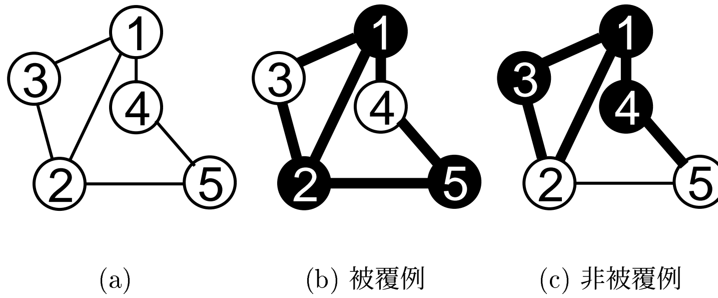
つまり，V' が一つでも存在すれば，そのグラフは k 頂点以下で被覆可能であるということが言える．具体例を図 1.1 で示す．今, 図 1.1(a) の 5 頂点単純グラフが与えられたとする．また，$k=3$ であったとする．そのとき，全解探索法では 3 個以下の頂点を選んできて定義 1.2 の条件を満たすかどうかを判定しなければならない．つまり，与えられたグラフの全ての辺について，任意に選択した頂点が端点となっているかどうかを調べなければならない．よって，辺の数を m とすると，条件を満たすかどうかの判定は $O(m)$ 時間かかる．辺の数は頂点数で抑えられるので，頂点数を n とすると，判定は $O(n^2)$ 時間かかることになる．この例では，図 1.1(b) のように頂点集合 $\{1,2,5\}$ を選んできた場合は全ての辺の端点のどちらかを覆うので成功し, 1.1(c) のように頂点集合 $\{1,3,4\}$ を選んだできた場合は辺 $(2,5)$ が覆えないので失敗することを示している．よって，図 1.1 の例の単純グラフは $k=3$ 以上のとき，常に頂点被覆可能なグラフであることが保証されているといえる．ただし，今の例の場合 2 個以下を選んでくると必ず失敗するグラフである．そこで，条件式をつぎのように一般化する．

定義 1.3. 頂点被覆とは，
単純グラフ $G(V,E)$ が与えられ，任意の頂点の選択

$$\chi(v) := \begin{cases} 1 & \text{頂点 } v \text{ が選択されているとき} \\ 0 & \text{頂点 } v \text{ が選択されないとき} \end{cases}$$

によって被覆が成功か否かの判定値 δ は

$$\delta := \bigwedge_{(u,v)\in E} \chi(u) \vee \chi(v)$$



(a)　　(b) 被覆例　　(c) 非被覆例

図 1.1: 5 頂点の単純グラフ

となる．

つまり，この問題の条件判定は驚くべきことに簡単に論理式で表せるのである．よって，条件判定値を求める計算は各辺の端点のどちらか一方でも選んだ頂点と一致するかを並列に計算することで，$O(1)$ 時間で実行できる．先ほどの図 1.1 の例で示した単純グラフに対する頂点被覆が可能かの条件判定値を決定する論理回路をハード ウェア化したものを図 1.2 でしめす.

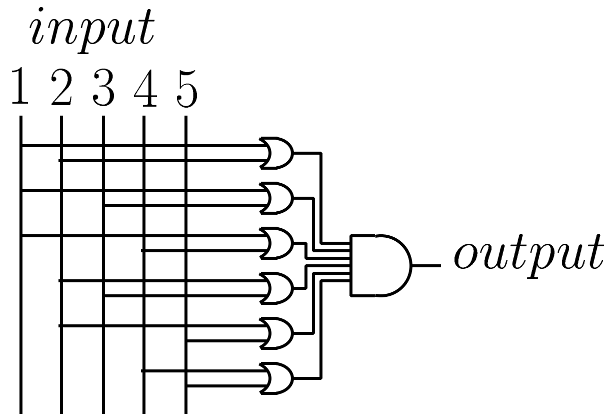


図 1.2: 図 1.1 に対応する論理回路.

また，1.1 節でしめした，独立頂点集合問題も同じ ように判定式を回路化することができる．ここでは，簡単に独立頂点集合の定義と，判定式の論理的表現をしめしておく．

定義 1.4. 独立頂点集合とは，
単純グラフ $G(V,E)$(ただし，$|V|=n$ とする）と n 以下の自然数 k が与えられたとき，

$$\Upsilon := \{V' \subseteq V | \forall (u,v) \notin E, u \in V' \wedge v \in V', |V'| \leqslant k\}$$

Υが空か，そうでないかを判定する問題である．

また，任意に選択されたV'がΥに入るかどうかの判定値δは，定義1.3の関数$\chi(v)$を用いて，

$$\delta := \bigvee_{(u,v)\in E} \chi(u) \wedge \chi(v)$$

となる．

また，1.1節でしめした，充足可能性問題は条件式自体が論理式であるため，そのまま回路化が可能である．

定義 1.5. 充足可能性 *(SAT)* とは，
n個の変数集合$\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$で構成された， 論理式

$$f(x_1, x_2, \ldots, x_n)$$

の値を真 *(1)* とする入力変数の真理値割当が存在するか否かという問題である．

このように，NP完全問題の一部は判定式や条件式を回路化することがたやすい．そこで，NP完全問題を解くハードウェア手法を図1.3にしめす．また，NP完全問題を解くヒューリスティックは条件式を頻繁に評価する．したがって，本論文では，条件式を回路化したハードウェア手法によるNP完全問題の解探索法を提案する．本論文の他にもFPGAを用いたNP完全問題を解くハードウェア手法として，グラフ彩色問題(GRAPH-CLORLING)ついてかかれた[2], ハミルトン閉路問題(HAM-CYCLE)についてかかれた[3], 充足可能性問題(SAT)についてかかれた[4], [8], [11] ,[12]などがある．また，NP完全問題ではないがグラフの問題である位相同型問題を扱った[7]がある．について FPGA を用いて

しかし，実際にVERTEX-COVER問題などを解くには任意の組合わせを生成する必要がある．そこで，本論文ではn個の中からある制約のもとでの組合わせを{0,1}ベクトルで生成するカウンタアルゴリズムを提案する．

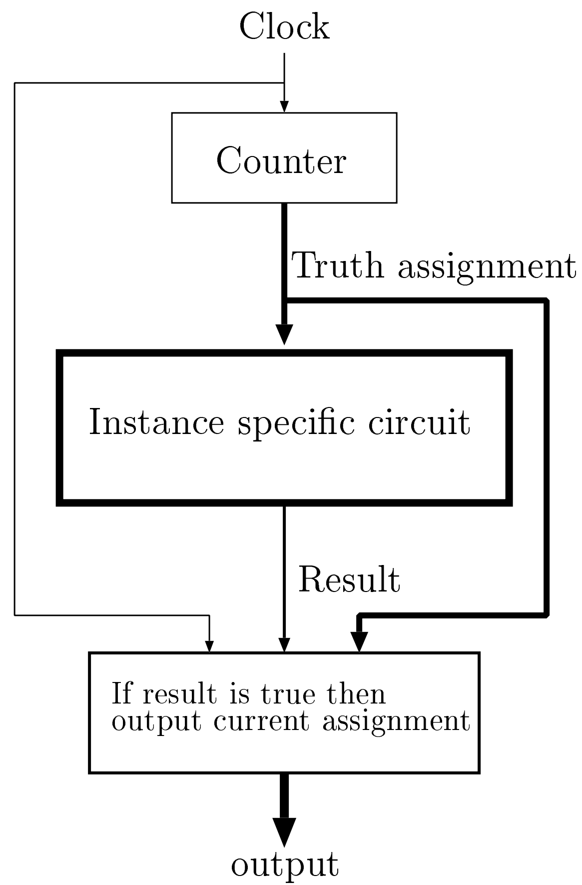


図 1.3: NP 完全問題を解くハードウェア手法の概要

第2章　カウンタアルゴリズム

ここでしめすカウンタアルゴリズムは任意の自然数 n と，0 から n までの自然数の部分集合 Φ が与えられたとき, つかえる 1 の数が $^{\forall}k(\in \Phi)$ 個に限定された長さ n の $\{0,1\}$ ベクトル v を全て列挙する.

定義 2.1. カウンタアルゴリズムを以下のように定義する.

入力

$$n \in N, \quad \Phi \subseteq \{0,1,2,...,n\}$$

出力

$$\bigcup_{k\in\Phi}\{v \mid v \in \{0,1\}^n, \text{かつ } v \text{ に現れる 1 の数が } k \text{ 個})\}$$

定義 2.2. n 個の中から,k 個を選ぶ組合わせの数を $\binom{n}{k}$ と表す．また，組合わせの数は次のようにも表すことができる.

$$\binom{n}{k} = \frac{n!}{(n-k)!k!}$$

定義 2.3. 自然数 n と，*0* から n までの自然数の部分集合 Φ が与えられたとき，定義 *2.1* で出力されるベクトルの集合を $\mathcal{V}_{n,\Phi}$ とする．また，出力されるベクトルの数を $|\mathcal{V}_{n,\Phi}|$ とし，次のような方程式で表す.

$$|\mathcal{V}_{n,\Phi}| := \sum_{^{\forall}i\in\Phi}\binom{n}{i}$$

定義 2.4. $\{0,1\}^n$ ベクトル v の i 番目の要素は，$v[i]$ と表す．$\{0,1\}^n$ ベクトル v をリストとして表すと，

$$v[n-1], v[n-2], \ldots, v[1], v[0]$$

となる.

定義 2.5. カウンタアルゴリズムの中には $\mathcal{V}_{n,\Phi}$ とは関係ないベクトルを出力しながら，動作するものもあるが, カウンタアルゴリズム自体がそのことを認識しながら，外部にそのことを伝えながら動作する状態にあるとき，その状態を冗長状態とし，そのときに, 現れる，ベクトルのことを冗長ベクトルとする.

例えば，2 進カウンタは制約集合 Φ を n 以下の自然数の集合として，辞書式順に $\mathcal{V}_{n,\leqslant n}$ を生成するカウンタと定義できる．また，そのときの列挙されるベクトルの総数は，

$$|\mathcal{V}_{n,\leqslant n}| = \sum_{0\leqslant i\leqslant n} \binom{n}{i} = \sum_{0\leqslant i\leqslant n} \binom{n}{i} 1^i 1^{n-i} = (1+1)^n = 2^n$$

となる．2 進カウンタを用いて逐一 1 のかずを数えながら，Φ の値との関係をとりながら，冗長状態をとりながら，$\mathcal{V}_{n,\Phi}$ を出力する方法もあるが，当然 2^n かかり，必要のない冗長ベクトルも多くつくられてしまう．

具体的なカウンタアルゴリズムを提案する前に，カウンタアルゴリズムの回路化はいくつか注意すべき点を以下に挙げる．

1. クロック同期式で動作させる
2. 高速化するためにクロック毎に行う計算とそうでないものに分けることが望ましい
3. クロック毎に行なわない計算回路が，クロック毎に行う計算回路よりも遅延時間が長い場合，クロック毎に行う計算回路の遅延時間より短くなるように分割することが望ましい

具体的には，カウンタアルゴリズムのなかで，冗長ベクトルを計算するような場合 2 を適用し, 回路の高速化を図る．

2.1 ${}_nC_k$ カウンタ

定義 2.6. ${}_nC_k$ カウンタは Φ が n 以下の自然数 k のみに限定された，辞書式順に

$$\mathcal{V}_{n,k}$$

を生成するカウンタである．

まず，例として $\mathcal{V}_{6,3}$ を表 2.1 にしめす．

$00\underline{01}11$	$0101\underline{01}$	$100\underline{01}1$	$1\underline{01}\mathbf{1}00$
$001\underline{01}1$	$01\underline{01}\mathbf{1}0$	$1001\underline{01}$	$1100\underline{01}$
$0011\underline{01}$	$0110\underline{01}$	$10\underline{01}\mathbf{1}0$	$110\underline{01}0$
$0\underline{01}\mathbf{11}0$	$011\underline{01}0$	$1010\underline{01}$	$11\underline{01}00$
$010\underline{01}1$	$\underline{01}\mathbf{11}00$	$101\underline{01}0$	111000

表 2.1: $V_{6,3}$ 左上から上から下へ，左から右へと辞書式順に並んでいる．

この例により，辞書式順に i 番目と $i+1$ 番目のベクトルの関係を，i 番目の状態から $i+1$ 番目の状態への遷移と考え, 例を観察したのちその原理を推測すると,

- (**初期状態**）は右端に k 個の 1 のかたまりがある
- 下線のついた最右の 01(**検索 rm01**）が次の状態では 10 に変わる (**ステップ 1**）
- 下線のついた最右の 01 より右側にある 1 のかたまりがある場合 (**判定 to2**）はその 1 のかたまりは右端に移動する (**ステップ 2**）
- (**終了状態**）は左端に k 個の 1 のかたまりがある

ということがいえる．

原理よりアルゴリズムは次のようになる．

---

input n, k
initialize $v_c = 0^{n-k}1^k$ # 初期ベクトル生成
output v_c
while $find_{01}(v_c)$: #01 がなくなるまで繰り返す
  $p = \mathrm{index}_{rightmost01}(v_c)$ # 最右の位置 (検索 rm01）
  $v_{tmpl} = v_c \wedge 1^{n-p+1}0^{p-1}$
  $v_{tmpr} = v_c \wedge 0^{n-p+1}1^{p-1}$
  $v_{tmpl} = v_{tmpl} \oplus 0^{n-p-1}110^{p-1}$ # ステップ 1
  if $v_c[p-1] == 1$ **and** $v_c[0] == 0$: # 判定 to2
    while $v_{tmpr}[0] == 1$: # ステップ 2
      $v_{tmpr} = v_{tmpr} \gg 1$
  end if
  $v_c = v_{tmpl} \vee v_{tmpr}$
  output v_c

つぎに，このアルゴリズムでの現在の状態から次の状態を導くときにかかる計算の複雑さについて考察する．

初期状態 状態の遷移には関係なくはじめに用意するものとする．

検索 rm01 ベクトルの長さを n とするとを $O(\log n)$ 時間で実行できる．理由は，まず，現在のベクトルに存在する 01 の場所を $n-1$ プロセッサ用いて並列に検出し, つぎに, ベクトルを半分に分け，右側に 01 が存在すれば，そちらの位置を出力し, なければ左側を検索するということを再帰的に行うことでできる．(参考資料 [10])

ステップ 1 検索 rm01 で見つけた 01 を 10 に変えるだけなので，定数時間で実行できる．

判定 to2 ステップ 2 を行うかどうかの判定で, 検索 rm01 での最右の 01 の位置がわかれば，その位置がベクトルの右端ではなく，かつその位置のすぐ右隣に 1 があるかどうかを判断するだけで，それより右側の 1 のかたまりをベクトルの右端に移動させなければならないかどうかを判定することができるので定数時間で実行できる．その理由は，最右の 01 の右隣に 1 が存在せず，それより右に 1 が存在するとすると，その 1 のかたまりの先頭が 01 となり，最右の 01 であるという事実と矛盾するからである．

ステップ 2 逐次的な方法として右端に到達するまで 1 ビットづつシフトしていくが考えられるが，この方法では高々$n-k-1$ の時間を必要とする．最悪の場合は最右の 01 がベクトルの中の位置で右から (右端を 0 番目として）$n-2$ 番目にあるときで, そのとき，1 のかたまり $k-1$ 個を右端に移動させるため，$n-k-1$ 回シフトを行わなければならない．ステップ 2 が $\mathcal{V}_{n,k}$ の全列挙の間に何度起るかの解析は後述で行う．

終了状態 判定は検索 rm01 で最右の 01 が見つからなかったときである．理由は，左端に k 個 1 が詰まっているから，01 が存在しないからである．

よって，このアルゴリズムの次状態に対する計算時間はステップ 2 を逐次で行うと,$O(n)$ となる．

定義 2.7. $\{0,1\}^*$ ベクトル v の最も右にある *01* の *1* の位置を $v[rm01]$ と表す．また，最も右にある *01* の *1* の位置にしか 1 がないベクトルを v_{rm01} と表す．

定義 2.8. $\{0,1\}^*$ ベクトル v の中にある *1* の数を $\sharp 1(v)$ と表す．

${}_nC_k$ カウンタが $\mathcal{V}_{n,k}$ を全て出力するまでに，ステップ 2 が何回現れるかを解析する．ステップ 2 は次のようなベクトルで起る．

$$\mathcal{V} := \{v | v \in (0+1)^{\alpha} 011^{\beta} 0^{\gamma} | \alpha+\beta+\gamma = n-2, 0 \leqslant \alpha, 1 \leqslant \beta \leqslant k-1, 1 \leqslant \gamma, \sharp 1(v) = k\}$$

$\mathcal{V}'$ の大きさはステップ 2 が終了した時点でのベクトルが

$$\mathcal{V}' := \{v'|v' \in (0+1)^{\alpha'}001^{\beta'} \mid \alpha' + \beta' = n-2,\ 0 \leqslant \alpha',\ 1 \leqslant \beta' \leqslant k-1\ ,\ \sharp 1(v') = k\}$$

なので，その大きさは $|\mathcal{V}| = |\mathcal{V}'|$ となり，

$$\begin{aligned}
|\mathcal{V}'| &= \sum_{1 \leqslant i \leqslant (k-1)} \binom{n-i-2}{k-i} \\
&= \binom{n-k}{1} + \binom{n-k-2}{2} + \cdots + \binom{n-2}{k-1} \\
&= \sum_{j \leqslant (k-1)} \binom{(n-k-2)+j}{j} - 1
\end{aligned}$$

となる．文献 [5]p.174,Table 174 より $\sum_{p \leqslant q} \binom{r+p}{p} = \binom{r+q+1}{q}$ なので，

$$|\mathcal{V}'| = \binom{n-2}{k-1} - 1$$

となる．

${}_nC_k$ カウンタが $\mathcal{V}_{n,k}$ を全て出力するまでにステップ 2 を行う比率は，

$$\begin{aligned}
\frac{\binom{n-2}{k-1} - 1}{\binom{n}{k}} =& \frac{\binom{n-2}{k-1}}{\frac{n}{k}\binom{n-1}{k-1}} - \frac{1}{\binom{n}{k}} \\
=& \frac{k}{n}\frac{\binom{n-1}{k-1} - \binom{n-2}{k-2}}{\binom{n-1}{k-1}} - \frac{1}{\binom{n}{k}} \\
=& \frac{k(n-k)}{n(n-1)} - \frac{1}{\binom{n}{k}} \\
\approx& \frac{k(n-k)}{n(n-1)}
\end{aligned}$$

となる．よって，ステップ 2 が起る確率は $\frac{k(n-k)}{n(n-1)}$ である．
また，$0 \leqslant k' \leqslant \frac{n}{2}$ とすると，

$$|\mathcal{V}_{n,k}| = \binom{n}{k} = \binom{n}{n-k} = |\mathcal{V}_{n,n-k}|$$

なので，

$$\mathcal{V}_{n,n-k} = \{v|\overline{v} \in \mathcal{V}_{n,k}\}$$

となる．つまり，$\frac{n}{2}$ を超えるときは入力に $n-k$ を与え, 出力のベクトルに Not 演算を行うことで，$\mathcal{V}_{n,n-k}$ を得ることができるので, カウンタの一般性を失わない．この場合, 任意の状態でステップ 2 が起る確率は高々$\frac{1}{4}$ である．

次に，このアルゴリズムをハードウェアで実装するためのハードウェア的アルゴリズムの提案と，ステップ 2 の様々な改良法をしめす．様々な方法を試した理由は，ステップ２は状態遷移で毎回起るわけでなく，その部分を高速化するために回路規模が大きくなると, 回路の深さがより深くなり, 結果として全体の動作速度も遅くなるのである．この事実は様々な方法と，評価実験からしめす．尚, 回路化するにあたって，n は固定されているとする．

2.1.1 素朴な方法

まず，回路規模や動作速度にこだわらない単純な方法を示す．素朴な方法では判定 to2 によりステップ 2 に移らなければならないベクトル

$$\mathcal{V} := \{v|v \in (0+1)^{\alpha}011^{\beta}0^{\gamma}|\alpha+\beta+\gamma = n-2, 0 \leqslant \alpha, 1 \leqslant \beta \leqslant k-1, 1 \leqslant \gamma, \sharp 1(v) = k\}$$

を検索 rm01 より右に存在するベクトル (つまり，$1^{\beta}0^{\gamma}$) を別レジスタに保存し，ステップ 2 のアルゴリズムを実行する．今, 別レジスタに保存されたベクトルを $0^{n-\beta-\gamma}1^{\beta}0^{\gamma}$ とすると，素朴な方法では一番右のレジスタが 1 になるまで右方向に 1 ビットシフトを繰り返す．つまり，γ 回 1 ビットシフトを繰り返す．これを n 段の深さの回路を用いて一度に行う．

回路の概要を図 2.1 にしめす．n ビットのベクトル (ただし，判定 rm01 で検出されるの最も左側の位置が左端の右隣であるので, 左端の 2 ビットは常に 0 となる．) この回路では上から順に右に 1 ビットシフトを行い右端が 1 になったときの，ベクトルを出力する．この回路の深さは明らかに $O(n)$ 段になり，回路規模は $O(n^2)$ 必要となる．そこで，この回路を基準にステップ 2 の改良法を次にしめす．

2.1.2 右シフト法

素朴な方法では，アルゴリズム全体を回路に埋め込んだ．しかし，ここで改良した右シフト法では１つのシフターを用いて，$O(n)$ 時間かけて右端まで移動させる方法である．この方法の利点として，素朴な方法は同期回路で実装すると，ステップ２の実行に毎回 $O(n)$ の深さにかかる時間を要したのに対して, １つのシフターしか用いないことで回路全体の同期速度に影響を与えないことである．右シフト法の回路の概要を図 2.2 にしめす．素朴な方法と同じように γ 回実行することで，右端に１のかたまりを移動させることができる．

このアルゴリズムで，${}_nC_k$ カウンタが $\mathcal{V}_{n,k}$ を全て出力するまでに，ステップ 2 にかかる時間を解析する．ステップ 2 の解析で行ったようにステップ 2 実行後のベクトル (つまり, 冗長ベクトル)

$$\mathcal{V}' = \{v'|v' \in (0+1)^{\alpha'}01^{\beta} \,|\, \alpha'+\beta = n-1,\, 0 \leqslant \alpha',\, 1 \leqslant \beta \leqslant k-1\ ,\, \sharp 1(v') = k\}$$

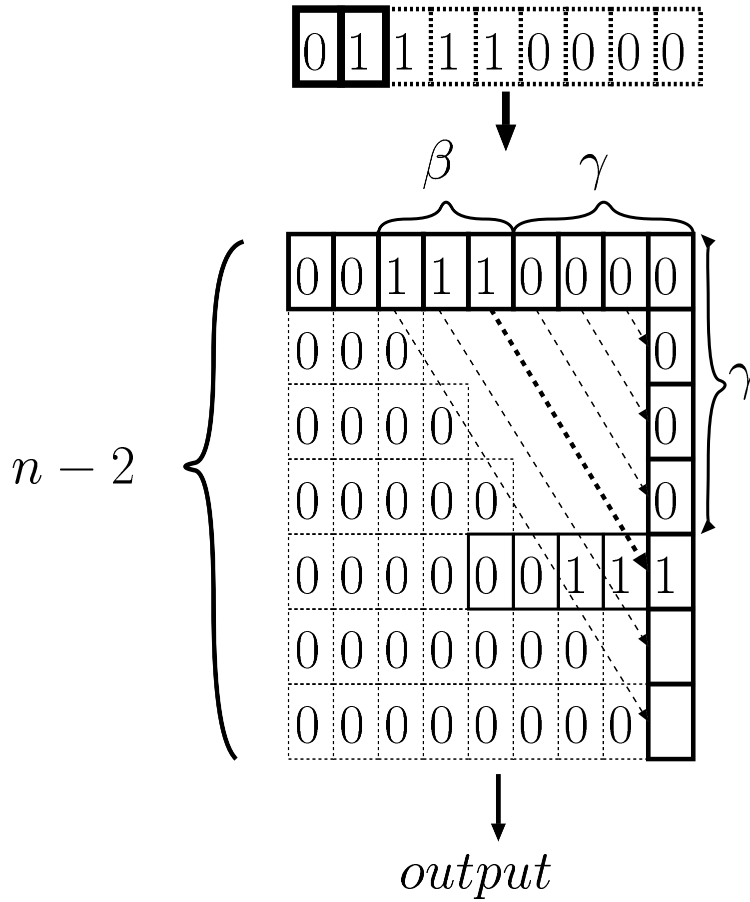


図 2.1: 素朴な方法の概念図

について考えれば，各々のベクトルで γ 回かかることは明らかだ．しかし，もう少し良く考えてみると,β 個連なった 1 が右にシフトしていく仮定で現れるベクトルは

$$\mathcal{V}^* := \{v^* | v^* \in (0+1)^{\alpha^*} 0 | \alpha^* = n-1, \sharp 1(v^*) = k\}$$

であることが推測される．しかし，例外として，

$$\{0^{\beta^*} 1^k 0^{\gamma^*} 0 | \beta^* + \gamma^* = n-k-1\}$$

が現れないので, ステップ 2 で右シフトに現れるベクトルのの数 $|\mathcal{V}^r|$ は

$$|\mathcal{V}^r| := \binom{n-1}{k} - \binom{n-k-1}{1}$$

と定義することができる．

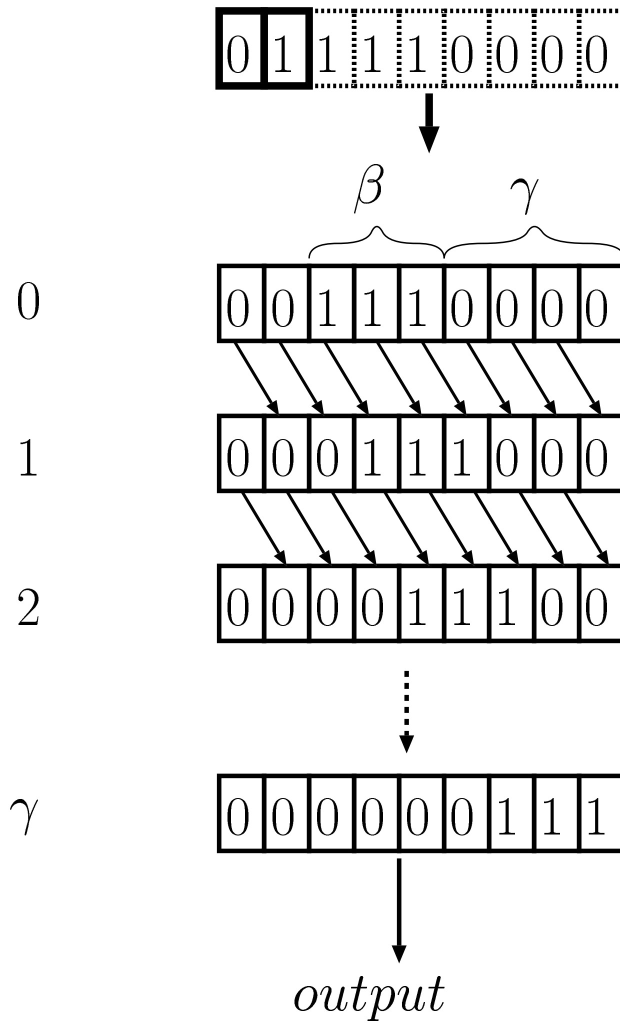


図 2.2: 右シフト法の回路の概念図

2.1.3 左シフト法

次に，右シフト法の改良を行った左シフト法を紹介する．右シフト法ではステップ 2 に現れるベクトル

$$\mathcal{V} := \{v|v \in (0+1)^{\alpha}011^{\beta}0^{\gamma}|\alpha+\beta+\gamma = n-2, 0 \leqslant \alpha, 1 \leqslant \beta \leqslant k-1, 1 \leqslant \gamma, \sharp 1(v) = k\}$$

の右端に存在する 0 の数 γ についてシフトさせていたが，左シフト法では,β についてシフトさせる．左シフト法では検索 rm01 で見つけた位置を利用して移動させる 1 のかたまりを左に巡回シフトさせる．具体的には，まず，ステップ 2 に移行しなければならない $v \in \mathcal{V}|$ の検索 rm01 で見つかった位置 (つまり，その位置にしか 1 がないベクトル）を右に 1 ビットシフトさせる．そして，別レジスタにとっておいた 1 のかたまりと”$\wedge$” 演算を

行い 1 があれば左シフトをおこない，右はしに 1 を付け足すことを繰り返す．繰り返すかどうかの判定 δ は検索 rm01 で求めた v_{rm01} を用いて

$$\delta = \bigvee_{0<i<n} v[i] \wedge v_{rm01}[i-1]$$

と表すことができる．

回路の詳細は，図 2.3 にしめす.

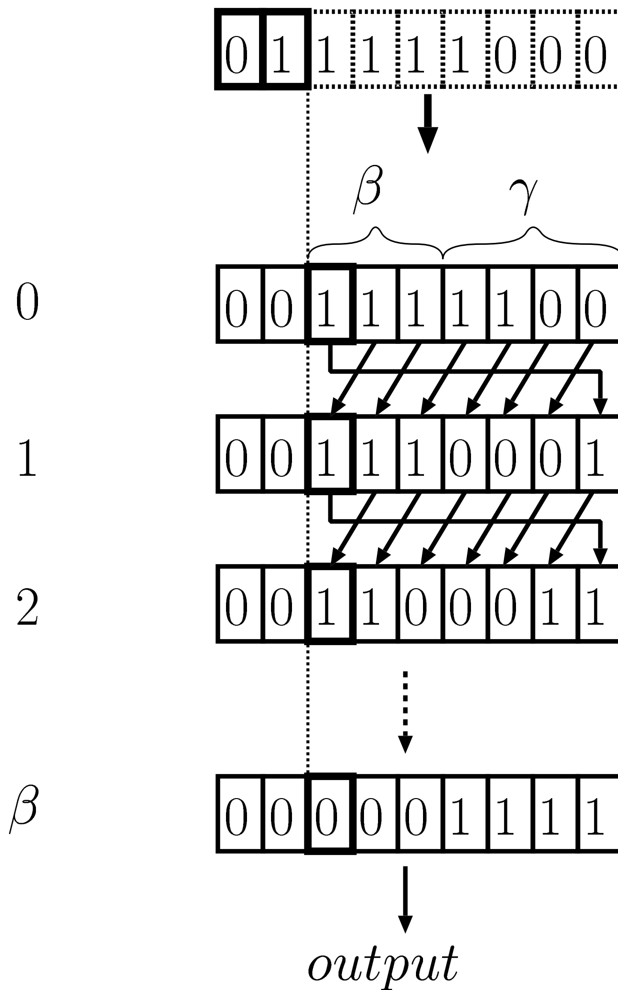


図 2.3: 左シフト法の回路の概念図

左シフト法を用いた ${}_nC_k$ カウンタが $\mathcal{V}_{n,k}$ を全て出力するまでに，ステップ 2 にかかる時間を解析する．ステップ 2 実行後に現れるベクトル

$$\mathcal{V}' = \{v' | v' \in (0+1)^{\alpha'} 01^{\beta} \mid \alpha' + \beta = n-1,\ 0 \leqslant \alpha',\ 1 \leqslant \beta \leqslant k-1\ ,\ \sharp 1(v') = k\}$$

より，場合の数は

$$\sum_{1\leqslant i\leqslant k-1}\binom{n-i-2}{k-i}i$$

となる．これを解析する前に補題 2.1 をしめす．

定理 2.1.

$$\binom{n}{k}=\sum_{0\leqslant i\leqslant k}\left\{\binom{n-i-2}{k-i}(i-1)\right\}$$

Proof.

$$\begin{aligned}\binom{n}{k}&=\binom{n-1}{k}+\binom{n-1}{k-1}\\&=\binom{n-1}{k}+\binom{n-2}{k-1}+\cdots+\binom{n-k}{1}+\binom{n-k-1}{0}\end{aligned}$$

それぞれの項に同じことをおこなう．

$$\begin{aligned}&=\binom{n-2}{k}+\binom{n-3}{k-1}+\binom{n-4}{k-2}+\cdots+\binom{n-k-1}{1}k+\binom{n-k-1}{0}+\\&\quad\binom{n-3}{k-1}+\binom{n-4}{k-2}+\cdots+\binom{n-k-1}{1}k+\binom{n-k-1}{0}+\\&\quad\vdots\\&\quad+\binom{n-k-1}{1}k+\binom{n-k-1}{0}\\&\quad+\binom{n-k-1}{0}\\&=\binom{n-2}{k}1+\binom{n-3}{k-1}2+\cdots+\binom{n-k-1}{1}k+\binom{n-k-1}{0}(k+1)\end{aligned}$$

$\binom{n-k-1}{0}=\binom{n-k-2}{0}$ なので，

$$=\sum_{0\leqslant i\leqslant k}\left\{\binom{n-i-2}{k-i}(i+1)\right\}$$

□

よって，補題 2.1 より，

$$\begin{aligned}\sum_{1\leqslant i\leqslant k-1}\left\{\binom{n-i-2}{k-i}i\right\}&=\sum_{0\leqslant i\leqslant k-1}\left\{\binom{(n-1)-i-2}{(k-1)-i}(i+1)\right\}-\binom{n-k-2}{0}k\\&=\binom{n-1}{k-1}-k\end{aligned}$$

となる．

2.1.4 k プロセッサ法

k プロセッサ法では今まで行ってきたステップ 2 の計算の改良というアプローチとは全く別の方法を行う．まず，この方法の原理を説明するため今まで用いてきたベクトルを再定義する．

定義 2.9. ${}_nC_k$ カウンタは

$$\mathcal{V}_{n,k} := \{v | v \in (0+1)^n \ and \ \sharp 1(v) = k\}$$

を生成するカウンタである．

定理 2.2. 最初に右に *1* が k 個詰まった状態で，

$$0^{n-k} 1_{k-1} 1_{k-2} \dots 1_2 1_1 1_0$$

のように右肩に順番をつけると，${}_nC_k$ カウンタアルゴリズムの実行中に順番が入れ替わることはない．

Proof. 操作は，一番右の 01 を 10 に変えることと，1 のかたまりを右はしに移動させるだけなので順番が変わることはないので，明らか．(ただし，左シフト法の冗長ベクトルでは一時的に入れ替わることがあるが, 冗長ベクトル以外ではこの性質が満たされているので, 一般性を失わない． □

そこで，k 個の 1 に対して各々別々のプロセッサを割当て，次の状態を別々に計算する法がこの k プロセッサ法である．具体的には検索 rm01 で見つかったベクトル v_{rm01} とその右隣から右端まで 1 でマスクされたベクトル v_{rmask} を用いて行う．プロセッサ P_i は次のように動作する．

initial $\mathcal{I}_i = 0^{n-i-1}10^i$

input v_{rm01}, v_{rmask}

  if $\left(\bigvee_{j \leqslant (n-1)} v_{rm01}[j] \wedge v_i[j] \right) == 1$ **then**

    $v_i = v_i << 1$;

  else if $\left(\bigvee_{j \leqslant (n-1)} v_{rmask}[j] \wedge v_i[j] \right) == 1$ **then**

    $v_i = \mathcal{I}_i$;

  else

    $v_i = v_i$;

  endif

output v_i

各プロセッサから，次のように現在ベクトル $v_{current}$ を決定する．

$$v_{current} := \bigvee_{0 \leqslant i \leqslant k-1} v_i$$

k プロセッサ法の回路の概念図を図 2.4 にしめす．

2.1.5 $_nC_k$ カウンタに関するまとめ

提案した $_nC_k$ カウンタの各アルゴリズムについてまとめる．まず，回路として実装したときの回路規模と $\mathcal{V}_{n,k}$ の全列挙にかかるクロックサイクル数に関して分けると以下のようになる．

- 冗長ベクトルを全く出力せず，回路規模が大きいアルゴリズム
  - 素朴な方法
  - k プロセッサ法
- 冗長ベクトルを出力するが，回路規模は小さいアルゴリズム
  - 右シフト法
  - 左シフト法

関連研究としてソフトウェアの実装についてかかれた [9] がある．

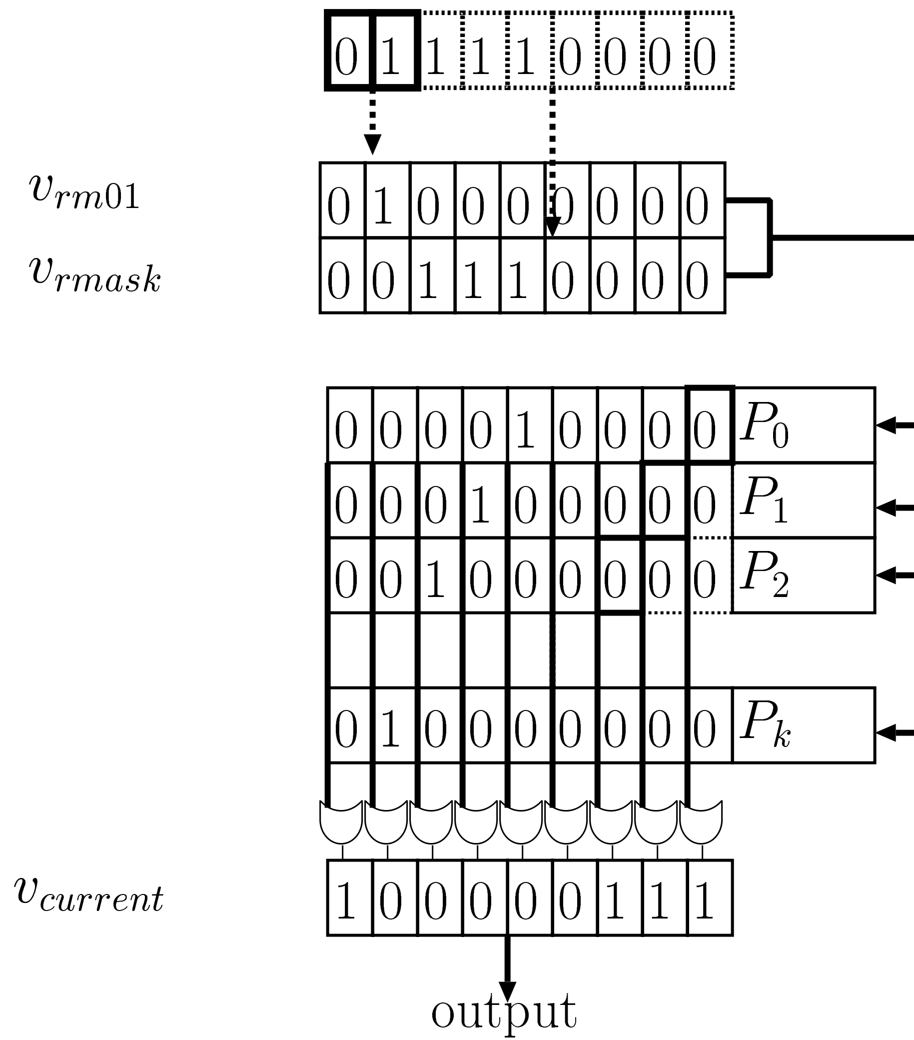


図 2.4: k プロセッサ法の回路概要

2.2 ${}_nC_{[0,k]}$ カウンタ

定義 2.10. ${}_nC_{[0,k]}$ カウンタは Φ を n 以下の自然数 k 以下の集合とした，辞書式順に

$$\mathcal{V}_{n,[0,k]}$$

を生成するカウンタである．

まず, 例として $\mathcal{V}_{6,\leqslant 3}$ を表 2.2 に示す.

```
000000     000111     001110 <-  010110 <-  100010     101010 <-
000001     001000     010000     011000     100011     101100 <-
000010     001001     010001     011001     100100     110000
000011     001010     010010     011010 <-  100101     110001
000100     001011     010011     011100 <-  100110 <-  110010 <-
000101     001100     010100     100000     101001     110100 <-
000110     001101     010101     100001     101000     111000
```

表 2.2: $n=6, k=3$ のときの ${}_nC_{[0,k]}$ カウンタの列挙例

そこで，この例からわかったことは,

初期状態 ベクトル 0^n，つまり 1 がない状態

ステップ 1 最下位ビットに 1 を足したときに，そのベクトルの中の 1 の数が k 個より多い場合はステップ 2 を行い，そうでなければそのベクトルを新しいベクトルとして出力する

ステップ 2 現在のベクトルの最も右にある 1 に 1 を足す

終了状態 ベクトル 1^k0^{n-k} つまり，01 が見つからない状態

ということである.

アルゴリズムを以下にしめす.

```
input n, k
initialize v_c = 0^n # 初期設定
output v_c
while find01(v_c): # 終了状態になるまで繰り返す
  v_tmp = v_c + 0^{n-1}1; # ステップ 1
  if count(v_tmp) > k: # 1 の数の数え上げ
    # ステップ 2 ここから
    p = index_rightmost1(v_c);
    v_tmp = 0^{n-p-1}10^{p-1};
    v_tmp = v_tmp + v_c;
    # ここまで
  end if
  v_c = v_tmp;
  output v_c
```

基本的には ${}_nC_k$ カウンタと類似のステップを踏む.

つぎに，このアルゴリズムでの現在のベクトルから次のベクトルをつくるまでの計算の複雑さについて考察する.

ステップ 1 桁上がり先読み加算器を用いれば $O(\log n)$ で実行できる

1 の数の数え上げ ベクトルを半分にして左に存在する 1 の数と右に存在する 1 の数を足す計算を再帰的に行えば求まるので, $O(\log n)$ で実行できる

ステップ 2 ${}_nC_k$ カウンタで示したように再帰的に右端の 1 を求めることができるので, $O(\log n)$ で実行できる

したがって，このアルゴリズムでベクトルの更新にかかる時間は $O(\log n)$ である.

$nC[0,k]$ カウンタが $\mathcal{V}_{n,[0,k]}$ を全て出力するまでに，ステップ 2 が何回現れるかを解析する．ステップ 2 は次のようなベクトルで起る

$$\mathcal{V} = \{v | v \in (0+1)^{n-1}0, \sharp(v) = k\} - \{1^k 0^{n-k}\}$$

ゆえに，

$$|\mathcal{V}| = \binom{n-1}{k} - 1$$

となる.

表 2.2 の例では

$$\binom{6-1}{3} - 1 = 9$$

となり，矢印で示したステップ 2 の数と一致する.

${}_nC_{[0,k]}$ カウンタは ${}_nC_k$ カウンタを用いても実装することが可能である．${}_nC_k$ カウンタを 0 から k まで動作させることで ${}_nC_{[0,k]}$ カウンタを実現できる．そこで，${}_nC_{[0,k]}$ カウンタと ${}_nC_k$ カウンタを用いた場合を比較する．

${}_nC_k$ カウンタを用いる場合のアルゴリズムを以下にしめす．

---

input n, k
for $i = 0$ **to** k:
  initialize $v_c = 0^{n-i}1^i$ # 初期設定
  output v_c
  while $find01(v_c)$: # 終了状態になるまで繰り返す
    $p = index_{rightmost1}(v_c)$;
    $v_{tmp} = 0^{n-p-1}10^{p-1}$;
    $c = count(v_{tmp})$;
    if $c < k$ # ステップ 2 の処理
      $v_{tmp} = v_{tmp} + 0^{n-c}1^c$;
    $v_c = v_{tmp}$;
    output v_c

各々のアルゴリズムが $\mathcal{V}_{n,[0,k]}$ の全列挙にかかるクロックサイクル数は以下のようになる．

- ${}_nC_{[0,k]}$ カウンタ

$$\sum_{0 \leqslant i \leqslant k} \binom{n}{i} + \binom{n-1}{k} - 1 \tag{2.1}$$

- ${}_nC_k$ カウンタを用いた場合

$$\sum_{0 \leqslant i \leqslant k} \binom{n}{i} + \sum_{2 \leqslant i \leqslant k} \left(\binom{n-2}{i-1} - 1 \right) \tag{2.2}$$

これらを比べるために以下の定理を証明する．

定理 2.3. *2* 以上の自然数 n と n 以下の自然数 k が与えられたとき，以下の不等式式が成り立つ．

$$\sum_{0 \leqslant i \leqslant k} \binom{n}{i} [0,k] \binom{n}{k} \tag{2.3}$$

Proof. 帰納法により証明する．$n = 0$，$k = 0$ のとき明らかに成り立つ．$n = p$, $k = q$ のとき，

$$\sum_{0 \leqslant i \leqslant q} \binom{p}{i} \leqslant q \binom{p}{q}$$

が成り立つと仮定すると, $n = p + 1$, $k = q + 1$ のとき,

$$
\begin{aligned}
\sum_{0 \leqslant i \leqslant q+1} \binom{p+1}{i} &= \sum_{0 \leqslant i \leqslant q} \binom{p}{i} + \binom{p}{q+1} \\
&\leqslant q \binom{p}{q} + \binom{p}{q+1} \\
&= (q-1)\binom{p}{q} + \left(\binom{p}{q} + \binom{p}{q+1} \right) \\
&= (q-1)\binom{p}{q} + \binom{p+1}{q+1} \\
&= (q-1)\frac{q+1}{p+1}\binom{p+1}{q+1} + \binom{p+1}{q+1} \\
&= \left(\frac{q^2-1}{p+1} + 1 \right) \binom{p+1}{q+1}
\end{aligned}
$$

定義より $q \leqslant p$ なので，$\frac{q^2-1}{p+1} \leqslant q$ よって，

$$
\sum_{0 \leqslant i \leqslant q+1} \binom{p+1}{i} \leqslant (q+1)\binom{p+1}{q+1}
$$

したがって，

$$
\sum_{0 \leqslant i \leqslant k} \binom{n}{i} [0, k] \binom{n}{k}
$$

が成り立つ. □

式 2.1, 2.2 を比較するため，2.2 から 2.1 を引く.

$$
\sum_{0 \leqslant i \leqslant k} \binom{n}{i} + \sum_{2 \leqslant i \leqslant k} \left(\binom{n-2}{i-1} - 1 \right) - \sum_{0 \leqslant i \leqslant k} \binom{n}{i} + \binom{n-1}{k} - 1
$$

$$
\begin{aligned}
&= \sum_{2 \leqslant i \leqslant k} \binom{n-2}{i-1} - k - \binom{n-1}{k} - 1 \\
&= \sum_{0 \leqslant i \leqslant k-1} \binom{n-2}{i} - \binom{n-1}{k} - k - 2
\end{aligned}
$$

定理 2.3 より，

$$
\begin{aligned}
&\leqslant (k-1)\binom{n-2}{k-1} - \binom{n-1}{k} - k - 2 \\
&= (k-1)\binom{n-2}{k-1} - \frac{n-1}{k}\binom{n-2}{k-1} - k - 2 \\
&= \frac{k(k-1)-n+1}{k}\binom{n-2}{k-1} - k - 2
\end{aligned}
$$

この式は少なくとも k が $\sqrt{n}$ より大きな値でなければ正にはならない．よって，つぎのことがいえる．

系 2.1. ${}_nC_{[0,k]}$ カウンタは変数 k が少なくとも $\sqrt{n}$ 以下では，${}_nC_k$ カウンタで代用した方が高速である．

2.3 ${}_nC_{[k,n]}$ カウンタ

定義 2.11. ${}_nC_{[0,k]}$ カウンタとは，自然数 n と n 以下の自然数 k が与えられたときに，

$$\Phi = \{k, k+1, \ldots, n\}$$

である．

$$\mathcal{V}_{n,\Phi} = \mathcal{V}_{n,[k,n]}$$

を生成するカウンタである．

${}_nC_{[0,k]}$ カウンタの原理の原理は ${}_nC_{[k,n]}$ カウンタの原理と全く同じである．つまり，${}_nC_{[k,n]}$ カウンタに入力として $n-k$ を用いて，出力を反転させることで実現できるのである．

そこで，アルゴリズムは次のようになる

```
input n, k
initialize v_c = 0^n # 初期設定
output  v_c
while find01(v_c): # 終了状態になるまで繰り返す
  v_tmp = v_c + 0^{n-1}1; # ステップ 1
  if count(v_tmp) > n - k: # 1 の数の数え上げ
    # ステップ 2 ここから
    p = index_{rightmost1}(v_c);
    v_tmp = 0^{n-p-1}10^{p-1};
    v_tmp = v_tmp + v_c;
    # ここまで
  end if
  v_c = v_tmp;
  output  v_c
```

${}_nC_{[k,n]}$ カウンタのアルゴリズムの複雑さと全く同じなので，$O(\log n)$ で動作する．

2.4 $_nC_{[i,j]}$ カウンタ

定義 2.12. $_nC_{[i,j]}$ カウンタとは，自然数 n と n 以下の自然数 j と j 以下の自然数 i が与えられたときに，

$$\Phi = \{i, i+1, \ldots, j\}$$

である．

$$\mathcal{V}_{n,\Phi} = \mathcal{V}_{n,[i,j]}$$

を生成するカウンタのことである．

まず，例として $\mathcal{V}_{6,[2,4]}$ を表 2.3 にしめす．

```
000011 <- u     001111 <- u     011010          100110          110001
000101          010001          011011          100111          110010
000110          010010          011100          101000          110011
000111 <- u     010011          011101          101001          110100
001001          010100          011110 <- o, u  101010          110101
001010          010101          100001          101011          110110 <- o
001011          010110          100010          101100          111000
001100          010111          100011          101101          111001
001101          011000          100100          101110 <- o     111010 <- o
001110          011001          100101          110000          111100
```

表 2.3: $n = 6, i = 2, j = 4$ のときの $nC \leqslant k$ カウンタの列挙例

この例からわかったことは，

初期状態 ベクトル $0^{n-i}1^i$，つまり 1 が i 個右詰めな状態

終了状態 ベクトル 1^j0^{n-j}，つまり 1 が j 個左詰めな状態．終了状態になるまで以下のステップ 1, 2, 3 を繰り返す

ステップ 1 現在のベクトル v_c に 1 を足したベクトル v'_c について，$\sharp(v'_c)$ が i より小さいならばステップ 2 を行い，j より大きいならばステップ 3 を行う．その他の場合は，1 増加させたベクトルを新しいベクトルとする．

ステップ 2 $i - \sharp(v'_c)$ 分だけ右から 1 を追加し，新しいベクトルとする

ステップ 3 v_c の最も右にある 1 の位置を見つけその位置に 1 を足したベクトル v''_c について，$\sharp(v''_c)$ が i より小さいならばステップ 2 を行い，そうでなければ，v''_c を新しいベクトルとする．

定理 2.4. ステップ *2* からステップ *3* へは遷移しない．

Proof. ステップ 2 終了後のベクトル v は必ず $\sharp(v) = i$ となっている．また，$i \leqslant j$．よって，$\sharp(v) \leqslant j$ なので，ステップ 2 よりステップ 3 へは遷移しない． □

アルゴリズムを以下に示す.

---

input n, i, j
initialize $v_c = 0^{n-i}1^i$
output v_c
while $find_{01}(v_c)$ **and** $v_c[n-j-1] == 1$:
　$v'_c = v_c + 0^{n-1}1$;
　if $\sharp(v'_c) < i$:
　　$v_{tmp} = 0^{n-\sharp(v'_c)}1^{\sharp(v'_c)}$;
　else if $\sharp(v'_c) > j$:
　　$p = index_{rightmost1}(v_c)$;
　　$v''_c = v_c + 0^{n-p}10^{p-1}$;
　　if $\sharp(v''_c) < i$:
　　　$v_{tmp} = 0^{n-\sharp(v''_c)}1^{\sharp(v''_c)}$;
　　else:
　　　$v_{tmp} = v''_c$;
　　end if
　else:
　　$v_{tmp} = v'_c$;
　end if
　$v_c = v_{tmp}$;
　output v_c

つぎに，このアルゴリズムでの現在のベクトルから次のベクトルを生成するまでの計算の複雑さについて考察を行う．基本的には先ほど紹介した ${}_nC_{[0,k]}$ カウンタアルゴリズムと同様の動作をしめすが，問題は，足りない 1 の補充にかかる時間である．足りない 1 の数は現在のベクトル v の $\sharp(v)$ と i よりすぐにわかる．そこで，補充する 1 の数を入力としたときに，その数分だけ，1 を右に詰める補充アルゴリズムを次に示す.

```
input p
initialize v_tmp
for q = 0 to log n:
  if p[q] == 1:
    v_tmp = v_tmp << 2^q;
    v_tmp = v_tmp + 0^{n-2^q} 1^{2^q};
  else:
    v_tmp = v_tmp;
  end if
output v_tmp
```

そこで，このアルゴリズムは明らかに $O(\log n)$ で走るので，$nCi \leqslant, \leqslant j$ カウンタアルゴリズムは $O(\log n)$ で走る．

$nCi \leqslant, \leqslant j$ カウンタが $\mathcal{V}_{n,[i,j]}$ を全て出力するまでに，オーバーフローとアンダーフローが何度出現するかを解析する．オーバーフローは次のベクトルで起る

$$\mathcal{V}^o = \{v | v \in (0+1)^{n-1}0, \text{ and } \sharp(v) == j\} - \{1^j 0^{n-j}\}$$

ゆえに，

$$|\mathcal{V}^o| = \binom{n-1}{j} - 1$$

となる．

また，アンダーフローは次のベクトルで起る

$$\mathcal{V}^u = \{v | v \in (0+1)^{n-\alpha-1}01^\alpha,\ 2 \leqslant \alpha, \text{ and } \sharp(v) - \alpha \leqslant i\}$$

しかし，このベクトルを解析するのは難しいので, アンダーフロー処理を終えたベクトルについて考える．そこで，アンダーフロー処理後のベクトルは

$$\mathcal{V}^{u'} = \{v | v \in (0+1)^{n-1}1, \text{ and } \sharp(v) == i\} - \{0^{n-i}1^i\}$$

なので，アンダーフローの数は

$$|\mathcal{V}^u| = |\mathcal{V}^{u'}| = \binom{n-1}{i-1} - 1$$

となる．

${}_nC_{[i,j]}$ カウンタは i と j の与え方によって 5 つのモードで動作させることができる．

1. $i = 0$, $j = n$ の場合
   - 2 進カウンタとして動作
2. $i = j$ の場合

- ${}_nC_k$ カウンタとして動作

3. $i = 0,\ i < j$ の場合
   - ${}_nC_{[k,n]}$ カウンタとして動作
4. $i < j,\ j = n$ の場合
   - ${}_nC_{[0,k]}$ カウンタとして動作
5. $0 < i < j < n$ の場合
   - ${}_nC_{[i,j]}$ カウンタとして動作

各モードでの冗長状態の数を解析すると，

1. $i = 0,\ j = n$ の場合
   - 0 (オーバーフローもアンダーフローもないので)
2. $i = j$ の場合
   - $i = j = k$ とすると，

     $$\left(\binom{n-1}{k} - 1\right) + \left(\binom{n-1}{k-1} - 1\right) = \binom{n}{k} - 2$$

     となる．
3. $i = 0,\ i < j$ の場合
   - $\binom{n-1}{j} - 1$
4. $i < j,\ j = n$ の場合
   - $\binom{n-1}{i-1} - 1$
5. $0 < i < j < n$ の場合
   - $\binom{n-1}{j} + \binom{n-1}{i-1} - 2$

第3章　カウンタアルゴリズムの実装と評価

カウンタアルゴリズムは任意の n と Φ に対して定義した．しかし，ここでは回路に実装する上で n は固定されているものと仮定して実装を行った．

ここで行った実装とは，まず，ソフトウェア言語をプログラミングするように，ハードウェア記述言語 (HDL) でアルゴリズムをプログラムを行い．次に, コンパイラで各社の FPGA 固有の回路情報に変換し, FPGA にアップロードし，FPGA 上で動作させる．

このとき，コンパイルされた回路情報に対して動作周波数や回路規模などを解析ツールによって，解析を行う．解析ツールはたいていコンパイラとセットになっており, コンパイル後には解析結果が表示されるようになっている．

また，回路規模を最適に保ちたい場合や，動作周波数について制限を行いたい場合に，コンパイラに最適化オプションを与えると望みの最適化を行ってくれる．本研究では，最適化オプションを全く使用しないコンパイルを行った．

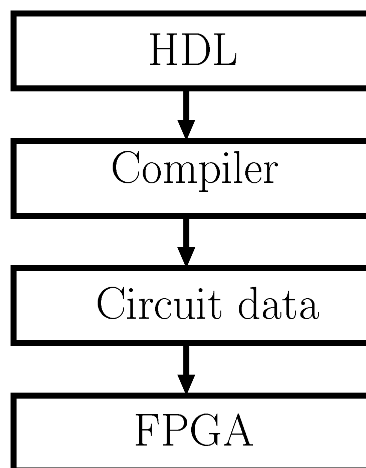


図 3.1: ハードウェア記述言語を FPGA で実行するまでの流れ

3.1 ${}_nC_k$ カウンタの実装と評価

${}_nC_k$ カウンタの実装は，以下の環境で行った．

ハードウェア記述言語 VerilogHDL[6]

FPGA ALTERA 社 APEX20K400EBC-652-1x

コンパイラ・解析ツール ALTERA QUARTUS II

実装を行ったアルゴリズムは

1. 素朴な方法
2. 右シフト法
3. 左シフト法
4. k プロセッサ法

である．

各々のアルゴリズムについて，$n = 16, 32, 48, 64, 80$ について実装を行った．各アルゴリズムを回路として実装したときの，$\mathcal{V}_{n,k}$ の全列挙にかかるクロックサイクル数を以下にしめす．

- 素朴な方法
$$\binom{n}{k}$$
- 右シフト法
$$\binom{n}{k} + \binom{n-1}{k} - \binom{n-k-1}{1}$$
- 左シフト法
$$\binom{n}{i} + \binom{n-1}{i-1} - i$$
- k プロセッサ法
$$\binom{n}{k}$$

図 3.2 に, 解析から求められた動作周波数をしめす.

各々のアルゴリズムで $\mathcal{V}_{n,k}$ の全列挙にかかる時間を解析より得られた動作周波数を用いて計算する．計算式は

$$\text{実行時間} := \frac{\text{クロックサイクル数}}{\text{動作周波数}}$$

n	素朴な方法	右シフト法	左シフト法	kプロセッサ法
16	37.97	110.77	124.52	48.03
32	18.61	89.58	88.94	34.23
48	12.38	44.76	51.09	26.84
64	8.76	38.56	41.03	22.92
80	6.79	29.97	27.35	19.65

図 3.2: 各アルゴリズムに対応する回路の動作周波数 (単位：MHz)

n	素朴な方法	右シフト法	左シフト法	kプロセッサ法
16	0.000339	0.000174	0.000155	0.000268
32	0.565196	0.205481	0.147829	0.307283
48	30.48053	15.45591	8.61696	14.0592
64	505.27	215.2246	121.3609	193.1137
80	4269.151	1837.715	1165.861	1475.193

図 3.3: $k = 8$ のときの各アルゴリズムの実行時間 (単位：秒）

である．

実行時間が割合現実的な値として $k = 8$ のときの各アルゴリズムでの全列挙にかかる時間を図 3.3 しめす．

このままでは，比較がしにくいので，最も遅い素朴な方法の実行時間で正規化する．図 3.4 にしめす．

n	素朴な方法	右シフト法	左シフト法	kプロセッサ法
16	1	1.94	2.18	1.26
32	1	2.75	3.82	1.83
48	1	1.97	3.53	2.16
64	1	2.34	4.16	2.61
80	1	2.32	3.66	2.89

図 3.4: 各アルゴリズムの実行時間の素朴な方法との比較

各アルゴリズムの回路の規模について今回使用した 40 万ゲート相当の FPGA に占める割合を図 3.5 にしめす．

各アルゴリズムの実行時間の比較より，最も高速であるのは左シフト法である．左シフト法は $n = 80$, $k = 8$ のとき，素朴な方法と比べて 3 倍強高速である．また，回路の占有率においても，右シフト法の次に占有率が小さいことがわかる．

n	素朴な方法	右シフト法	左シフト法	kプロセッサ法
16	2%	1%	1%	4%
32	8%	2%	3%	12%
48	19%	3%	5%	30%
64	34%	5%	7%	44%
80	53%	6%	8%	76%

図 3.5: 各アルゴリズムに対応する回路の使用 FPGA における占有率

3.2 ${}_nC_{[0,k]}$ カウンタの実装と評価

${}_nC_{[0,k]}$ カウンタの実装は，以下の環境で行った．

ハードウェア記述言語 VerilogHDL

FPGA Xilinx Vertex II 3000

タイミング解析ツール Xilinx ISE design tools

今回, 新たに 300 万ゲート相当の FPGA を用いて実装を行った．比較対象として ${}_nC_k$ カウンタの代表として左シフト法も同様に実装を行った．

各々のアルゴリズムについて，$n = 32, 64, 128, 256, 512, 1024$ について実装を行った．各アルゴリズムを回路として実装したときの，$\mathcal{V}_{n,\leqslant k}$ の全列挙にかかるクロックサイクル数を以下にしめす．

- ${}_nC_k$ (左シフト法)

$$\sum_{0\leqslant i\leqslant k}\left(\binom{n}{i}+\binom{n-1}{i-1}-i\right)$$

- ${}_nC_{[0,k]}$

$$\sum_{0\leqslant i\leqslant k}\binom{n}{i}+\binom{n-1}{k}-1$$

図 3.6 に解析から求められた動作周波数をしめす．$n = 1024$ のとき，左シフト法で異常な値を出力した．

${}_nC_k$ カウンタの評価で行ったように，各々のアルゴリズムで $\mathcal{V}_{n,\leqslant k}$ の全列挙にかかる時間を解析より得られた動作周波数を式 3.1 用いて計算する．$k = 8$ のときを図 3.7 にしめす．

このままでは，比較がしにくいので，遅いほうの ${}_nC_{[0,k]}$ カウンタの実行時間で正規化する．図 3.4 にしめす．

各アルゴリズムの回路の規模について今回使用した 300 万ゲート相当の FPGA に占める割合を図 3.9 にしめす．

n	左シフト法	nCamk
32	67.12	57.38
64	52.14	46.01
128	45.63	36.1
256	27.86	26.46
512	20.81	18.72
1024	1.9	11.8

図 3.6: 各アルゴリズムに対応する回路の動作周波数 (単位：MHz)

n	左シフト法	nCamk
32	2.77E-01	3.24E-01
64	1.10E+02	1.25E+02
128	3.56E+04	4.49E+04
256	1.56E+07	1.64E+07
512	5.49E+09	6.11E+09
1024	1.53E+13	2.49E+12

図 3.7: 各アルゴリズムの実行時間 (単位：秒)

図 3.8 より，$k = 8$ のとき，${}_nC_{[0,k]}$ カウンタは ${}_nC_k$ カウンタで同じ動作を行う方法より多少遅く動作することがわかった．

もう少し詳しくみるために，$n = 8,\ 16,\ 32,\ 64,\ 128,\ 256,\ 512$ と変化させ $k = 1\frac{n}{16}$, $k = 2\frac{n}{16}$, $k = 3\frac{n}{16}$, $k = 4\frac{n}{16}$, $k = 5\frac{n}{16}$, $k = 6\frac{n}{16}$, $k = 7\frac{n}{16}$, $k = 8\frac{n}{16}$, $k = 9\frac{n}{16}$ と変化させたたときに，${}_nC_{[0,k]}$ カウンタは ${}_nC_k$ カウンタの実行時間の何倍かかっているかについて，

$$\text{比較比} := \frac{{}_nC_{[0,k]}\text{カウンタの実行時間}}{{}_nC_k\ \text{カウンタを用いた場合の実行時間}} \tag{3.1}$$

を計算してみると，

図 3.10 のようになる．わかったことは，k が $6\frac{n}{16}$ を超える部分では ${}_nC_{[0,k]}$ カウンタの方が多少高速であることがわかった．

また，300 万ゲート相当の FPGA を占める割合について図 3.9 により，${}_nC_{[0,k]}$ カウンタの方が少ない回路規模であることがわかる．

${}_nC_{[0,k]}$ カウンタのアルゴリズムでは，足し算を行うため，足し算回路のアルゴリズムにどんなものを用いるかによって動作速度や回路規模に違いがある．代表的な足し算回路として，桁上がりを先読みしない素朴な足し算回路と桁上がりを先読みする桁上がり先読み足し算回路を比較する．2 つのアルゴリズムの複雑さの違いをしめすと，先読みなし回路は $O(n)$ の実行時間を要し, 先読みあり回路は $O(\log n)$ の実行時間を要する．それぞれを

n	左シフト法	nCamk
32	1.17	1
64	1.13	1
128	1.26	1
256	1.05	1
512	1.11	1
1024	0.16	1

図 3.8: 各アルゴリズムの実行時間の素朴な方法との比較

n	左シフト法	nCamk
32	3%	0%
64	6%	1%
128	12%	2%
256	25%	5%
512	45%	11%
1024	76%	23%

図 3.9: 各アルゴリズムに対応する回路の使用 FPGA における占有率

${}_nC_{[0,k]}$ カウンタに用いた結果を図 3.11 と図 3.12 にしめす.

図 3.11 の結果より，先読み回路は先読みなしの場合よりも高速に動作することがわかった．特に，変数 n の値が大きくなるとその効果は大きく，例えば，$n = 1024$ のときは先読みあり回路は 2 倍高速に動作することがわかった.

次に, 先読みありの ${}_nC_{[0,k]}$ カウンタについて，先ほどの ${}_nC_k$ カウンタを用いた場合と比較してみる．式 3.1 により，計算したものを図 3.13 にしめす．$n = 512$, $k = 128$ は先読みなしでは負けていたが, 先読みありでは本の少しだけ高速化に成功している.

また，図 3.12 の結果より, 先読みあり回路は先読みなし回路と比べて，より多くの回路量が必要なることがわかった.

n/k	$1\frac{n}{16}$	$2\frac{n}{16}$	$3\frac{n}{16}$	$4\frac{n}{16}$	$5\frac{n}{16}$	$6\frac{n}{16}$	$7\frac{n}{16}$	$8\frac{n}{16}$	$9\frac{n}{16}$
16	2.13	1.88	1.66	1.47	1.31	1.17	1.06	0.96	0.89
32	2.08	1.84	1.63	1.44	1.28	1.13	1.01	0.91	0.84
64	2.07	1.83	1.61	1.42	1.25	1.11	0.98	0.88	0.81
128	2.07	1.82	1.61	1.41	1.24	1.09	0.95	0.85	0.79
256	2.07	1.82	1.60	1.41	1.23	1.08	0.94	0.83	0.78
512	2.06	1.82	1.60	1.41	1.23	1.07	0.93	0.82	0.78

図 3.10: 実行時間の比較

n	先読みなし	先読みあり
32	57.382	43.269
64	46.019	36.566
128	36.13	32.278
256	26.463	28.897
512	18.722	25.471
1024	11.89	21.12

図 3.11: 各アルゴリズムに対応する回路の動作周波数 (単位：MHz)

3.3 $_nC_{[i,j]}$ カウンタの実装と評価

$_nC_{[i,j]}$ カウンタの実装は，以下の環境で行った．

ハードウェア記述言語 VerilogHDL

FPGA Xilinx Vertex II 3000

タイミング解析ツール Xilinx ISE design tools

図 3.14 にしめす．$_nC_k$ カウンタ (左シフト法) と桁上がり先読み加算器を用いた $_nC_{[0,k]}$ カウンタを比較対象とし，実装を行った．ここで，実装した $_nC_{[i,j]}$ カウンタは桁上がり先読み加算器を用いている．また，$_nC_{[i,j]}$ カウンタは変数の与え方で，$_nC_k$ と $_nC_{[0,k]}$ カウンタをシミュレートできるので，それぞれについて比較評価を行った．

まず，各アルゴリズムの $_nC_k$ と $_nC_{[0,k]}$ モードで全列挙にかかるクロックサイクル数を表 3.1 にしめす．

解析結果より，各アルゴリズムに対応する回路の動作周波数を図 3.14 にしめす．

表 3.1 と図 3.14 により，$_nC_{[i,j]}$ カウンタが $_nC_{[0,k]}$ カウンタをシミュレートしたときは動作周波数が $_nC_{[0,k]}$ カウンタより遅く，また，全列挙にかかるクロックサイクル数が同じなので，$_nC_{[i,j]}$ カウンタは $_nC_k$ カウンタよりも遅い．また，前節より，$_nC_{[0,k]}$ カウンタは

n	先読みなし	先読みあり
32	0%	1%
64	1%	3%
128	2%	8%
256	5%	19%
512	11%	42%
1024	23%	85%

図 3.12: 各アルゴリズムに対応する回路の使用 FPGA における占有率

n/k	8	16	32	64	128	256
32	1.91	1.21				
64	2.23	1.74	1.07			
128	2.33	2.06	1.60	0.96		
256	1.81	1.70	1.50	1.16	0.68	
512	1.58	1.54	1.44	1.27	0.98	0.57

図 3.13: 各アルゴリズムに対応する回路の動作周波数 (単位：MHz)

${}_nC_k$ カウンタよりも実用上遅いので, 結果として，${}_nC_{[0,k]}$ モードでは ${}_nC_{[i,j]}$ カウンタが最も遅い.

${}_nC_k$ モードについては，$k=8$ のときについて実行時間を式 3.1 を用いて各々のアルゴリズムについて求めたものを図 3.15 にしめす．図 3.15 より，遅い ${}_nC_{[i,j]}$ カウンタに正規化したものを図 3.16 にしめす．図 3.16 より，${}_nC_{[i,j]}$ カウンタは ${}_nC_k$ モードでは ${}_nC_k$ カウンタよりも多少遅くなる．

${}_nC_{[i,j]}$ モードについては，$j-i=8$ のときについて実行時間を式 3.1 を用いて各々のアルゴリズムについて求めたものを図 3.17 にしめす．図 3.17 より，遅い ${}_nC_{[i,j]}$ カウンタに正規化したものを図 3.18 にしめす．図 3.18 より，${}_nC_{[i,j]}$ カウンタは ${}_nC_{[i,j]}$ モードでは ${}_nC_k$ カウンタよりも多少遅くなる．

各アルゴリズムに対応する回路の使用 FPGA における占有率を図 3.19 にしめす.

モード	${}_nC_k$	${}_nC_{[0,k]}$	${}_nC_{[i,j]}$
${}_nC_k$ (左シフト法)	$\binom{n}{k}+\binom{n-1}{k-1}$	$\sum_{0\leqslant i\leqslant k}\left(\binom{n}{i}+\binom{n-1}{i-1}\right)$	$\sum_{i\leqslant p\leqslant j}\left(\binom{n}{i}+\binom{n-1}{p-1}\right)$
${}_nC_{[0,k]}$	-	$\sum_{0\leqslant i\leqslant k}\binom{n}{i}+\binom{n-1}{k}$	-
${}_nC_{[i,j]}$	$2\binom{n}{k}-2$	$\sum_{0\leqslant i\leqslant k}\binom{n}{i}+\binom{n-1}{k}$	$\sum_{i\leqslant p\leqslant j}\binom{n}{p}+\binom{n-1}{j}+\binom{n-1}{i-1}-2$

表 3.1: 各アルゴリズムの各モードでのクロックサイクル数

n	左シフト法	nCamk	nCiaj
32	67.12	43.26	42.04
64	52.14	36.56	34.66
128	45.63	32.27	31.12
256	27.86	28.89	26.81
512	20.81	25.47	24.94
1024	1.9	21.1	22.59

図 3.14: 各アルゴリズムに対応する回路の動作周波数 (単位：MHz)

n	左シフト法	nCiaj
32	2.74E-01	5.00E-01
64	1.59E+02	2.55E+02
128	6.07E+04	9.19E+04
256	2.89E+07	3.06E+07
512	1.06E+10	8.89E+09
1024		2.58E+12

図 3.15: ${}_nC_k$ モードでの各アルゴリズムの実行時間

${}_nC_{[i,j]}$ カウンタは高機能であるが，${}_nC_k$ カウンタや ${}_nC_{[0,k]}$ カウンタに比べて少々遅い．応用する問題にもよるが，FPGA で実装するにあたってはより小さな回路規模のアルゴリズムでの実装が望まれる．

n	左シフト法	nCiaj
32	1	1.82
64	1	1.60
128	1	1.51
256	1	1.06
512	1	0.84

図 3.16: ${}_nC_k$ モードでの各アルゴリズムの実行時間の比較

n	左シフト法	nCiaj
32	2.74E-01	5.00E-01
64	1.59E+02	2.55E+02
128	6.07E+04	9.19E+04
256	2.89E+07	3.06E+07
512	1.06E+10	8.89E+09
1024		2.58E+12

図 3.17: ${}_nC_{[i,j]}$ モードでの各アルゴリズムの実行時間

n	左シフト法	nCiaj
32	1.00	1.21
64	1.00	1.74
128	1.00	2.06
256	1.00	1.70
512	1.00	1.54

図 3.18: ${}_nC_{[i,j]}$ モードでの各アルゴリズムの実行時間の比較

n	左シフト法	nCamk	nCiaj
32	3%	1%	1%
64	6%	3%	4%
128	12%	8%	9%
256	25%	19%	21%
512	45%	42%	45%
1024	76%	85%	90%

図 3.19: 各アルゴリズムに対応する回路の使用 FPGA における占有率

第4章　結論

本研究では，NP 完全問題の全解探索法の高速化を行うために, 条件式を回路化し，任意の真理値割当に対して高速に，値を得る手法について提案した.

真理値割当に制限がある問題に対して，真理値の組合せを効率よく生成するアルゴリズムを提案した.

${}_nC_k$ カウンタとして最も適しているアルゴリズムは左シフト法であった．その理由は，左シフト法で行う処理が非常に単純で, 回路規模が小さいため，高速に動作するからである.

${}_nC_{[0,k]}$ カウンタは興味深いアルゴリズムであるが, 入力変数 n, k について理論的には，k の値が少なくとも $\sqrt{n}$ 以下では，${}_nC_k$ カウンタで代用する方が高速に動作することがわかった．実験的には，$n = 512$ 以下のとき，図 3.10 にしめされているように，$6\frac{n}{16}$ 以下では，${}_nC_k$ カウンタで代用する法が高速に動作することがわかった．この実験結果より，${}_nC_k$ カウンタを用いた実装の方が良いと思われる境界は，理論的にもっと上方にあることを示すことができるならば，${}_nC_k$ カウンタは ${}_nC_{[0,k]}$ カウンタよりも優れているといえる．しかし，現実的に解くことができる k の値はかなり小さい値で，$\sqrt{n}$ 以下で ${}_nC_k$ が十分高速であることが示せただけでも，十分な成果である．つまり，$O(n^{\sqrt{n}})$ 時間かけるぐらいであれば，${}_nC_{[0,k]}$ カウンタではなく，${}_nC_k$ カウンタを用いれば十分ということになる.

${}_nC_{[k,n]}$ カウンタは ${}_nC_{[0,k]}$ カウンタで代用できるので, 結果として，このカウンタも ${}_nC_k$ カウンタで代用できる.

${}_nC_{[i,j]}$ カウンタは汎用性が高いが，${}_nC_{[0,k]}$ カウンタと同様 ${}_nC_k$ カウンタで代用できる.

結論として，${}_nC_k$ は，その他のアルゴリズムを高速にシミュレートすることができる．また，回路規模も小さいという優れた性質もある.

謝辞

本研究を進めるにあたり，ご鞭撻を賜りました情報科学研究科 浅野 哲夫 教授，中野 浩嗣 助教授， 元木 光雄 助手に深く感謝致します．

参考文献

[1] Michael R. Garey, David S. Johnson. COMPUTERS AND INTRACTABILITY A Guide to Theory of NP-Completeness. 1979.

[2] L. M. Pochet, M. Linderman, R. Kohler, and S. Drager. An FPGA Based Graph Coloring Accelerator. MAPLD International Conference. September 2000.

[3] J. Bingham, M. Serra. Solving Hamiltonian Cycle on FPGA Technology via Instance to Circuit Mappings. Workshop on Enginnering of Reconfigurable Hardware/Software Objects, PDPTA. June 2000.

[4] A. Dandalis, M. Redekopp. A Parallel Pipelined SAT Solver for FPGAs. International Workshop on Field Programmable Logic and Applications. September 2000.

[5] Ronald L.Graham, Donald E. Knuth, Oren Patashnik. Concrete Mathematics Second Edition. 1994.

[6] Donald E. Thomas, Philip R. Moorby. The Verilog Hardware Description Language. Fourth Edition. 1998.

[7] Shuichi Ichikawa, Hidemitsu Saito, Lerdtanaseangtham Udorn, Kouji Konishi. Evaluation of Accelerator Designs for Subgraph Isomorphism Problem. 10th International Conference, FPL. Villach, Austria, August 27-30 2000.

[8] P.Zhong, M. Martonosi, P. Ashar, and S. Malik. Accelerating Boolean satisfiability with configuable hardware. Proc. Symposum on Field Programmable Custom Compution Machines. pp.186-195. 1998.

[9] Samuel P. Harbison, Guy L. Steele Jr. C: A Reference Manual Prentice-Hall, second edition. pp.176. 1987.

[10] Dexter C. Cozen. The Design and Analysis of Algorythms. 1991.

[11] 伊藤暁一．FPGA を用いた NP 完全問題の探索法に関する研究．名古屋工業大学大学院 工学研究科 博士前期課程 修士論文．1999．

[12] 須山 敬之, 横尾 真，澤田 宏，名古屋 彰．再構築可能なハードウェアを用いた充足可能性問題の解法．電子情報通信学会論文誌 D-I Vol.J84-D-I No.4．pp.410-420．2001 年 4 月．