Vol.2020-AL-177 No.8
2020/3/16

BDDを用いたソーティングネットワークの生成

大西 建輔[1,a)]　宇野 毅明[2]

概要：ソーティングネットワークは, チャンネルと 2 個のチャンネルを結ぶ比較器からなる構造である. チャンネルには値を入力することができ, 比較器で大小関係に応じて値を交換する. この交換を繰り返しおこなうことで, どのような値が入力されてもソーティングネットワークは, ソートされた値を出力する.
我々は, ソーティングネットワークを, 二分決定グラフ BDD を用いて生成する手法について提案する. また, 提案手法を実装し, チャンネル数が 8 以下の比較器数が最小のソーティングネットワークを生成した.

A computation method of minimum-comparison sorting network using BDD

Kensuke Onishi[1,a)]　Takeaki Uno[2]

Abstract: Sorting network consists of channels and comparators connecting two channels. Each channel has a value. A comparator exchanges two values of channels when the value of upper channel is smaller than that of lower. Any values of channels is sorted by the comparators of sorting network.
We propose a computing method of sorting network using binary decision diagram (BDD). We implement the method and generate sorting network with smallest number of comparators for 8 or less channels.

1. はじめに

ソーティングネットワークは, 1970 年代から現在でも研究されている構造である [1]. ソーティングネットワークは, n チャンネルと比較器の列からなる構造である. 図 1 は, 4 チャンネルと 5 比較器をもつソーティングネットワークである. 水平線分は, チャンネルを表現しており, 上から順に 1 から n まで番号を割り振る. 最初に, それぞれのチャンネル i は, 数値 v_i を持つとする. この数値が左から右に流れていく.

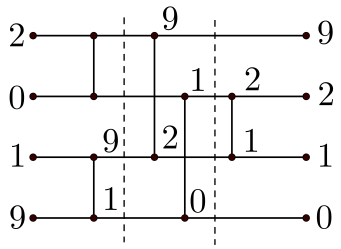

図 1　4 チャンネルをもつソーティングネットワーク

垂直線分は, **比較器** $c_{i,j}(i, j \in \{1, 2, \ldots, n\})$ である. $c_{i,j}(i < j)$ は, チャンネル i とチャンネル j を繋げており, もし, $v_i < v_j$ であれば, チャンネル i に v_j を, チャンネル j に v_i を流す. すなわち, $c_{i,j}$ を通ったあと, チャンネル i の値はチャンネル j の値よりも大き

[1] 東海大学
Tokai University
[2] 国立情報学研究所
National Institute of Informatics
[a)] onishi@tokai-u.jp

ⓒ 2020 Information Processing Society of Japan

いか等しい.

n チャンネルと比較器の列をまとめて**比較器ネットワーク**と呼ぶ. 比較器ネットワークが, どのような n 個の数値であってもソートするとき, 比較器ネットワークを**ソーティングネットワーク**と呼ぶ. すなわち, 比較器ネットワークやソーティングネットワークは, 比較器の列として, 表すことができる. 図 1 のソーティングネットワークは, $\{c_{1,2}, c_{3,4}, |c_{1,3}, c_{2,4}, |c_{2,3}\}$ と表すことができる. ここでは, 比較器列を "|" で区切っている. 部分比較器列が**レイヤ**であるとは, その部分比較器列の交換するチャンネルに同じチャンネルが含まれない場合である. このレイヤ内での比較は, 一度におこなうことが可能である.

多くの研究が [1], pp.219-247 にまとめられている. ここで本稿で利用する重要な性質を説明する.

定理 1. ([1], p.223) n チャンネルの比較器ネットワークが全ての 2^n 個のビット列を全てソートできるなら, そのネットワークはソーティングネットワークである.

定理 1 を用いることで, $n!$ 通りの入力ではなく, 2^n 通りの入力を試すだけで, 比較器ネットワークがソーティングネットワークであるかどうかを確認できる.

また, ソーティングネットワークでは, 次の 2 つの問題がよく研究されている.

- 比較器数の最小値 $\hat{S}(n)$ の値を求める.
- レイヤ数の最小値 $\hat{T}(n)$ の値を求める.

表 1 は, 既知のソーティングネットワークの比較器数とレイヤ数の最小値を集めたものである. 多くの結果は, [1] で見つけることができる. $n = 1, 2, \ldots, 8$ のとき, n チャンネルの場合の既知の最小比較器数 $S(n)$ は $\hat{S}(n)$ に等しい. Codish らは, $n = 9, 10$ のとき, $S(n) = \hat{S}(n)$ であることを示した [3].

$n = 1, 2, \ldots, 10$ のとき, n チャンネルの場合の既知の最小レイヤ数 $T(n)$ は $\hat{T}(n)$ に等しい. Bundala らは, $n = 11, \ldots, 16$ のとき, $T(n) = \hat{T}(n)$ であることを示した [4] . Codish らは, $n = 17, \ldots, 20$ のとき, $T(n) = 10$ であることを見つけた. さらに, $n = 17$ のとき, $T(n) = \hat{T}(n)$ であることを示した [5].

これらの結果は, 2 つのアイデアを元に示されている. 1 つは, 最初の 2 レイヤのパターンを同定することである. 対称性から, 最初のレイヤは $c_{1,2}, c_{3,4}, \ldots, c_{2\lfloor n/2 \rfloor - 1, 2\lfloor n/2 \rfloor}$ と考えて良い. [4] では, 最小のレイヤ数のソーティングネットワークを構成するときに, 2 番目のレイヤとして考えられるパターンを同定することに成功した (表 2). これにより, 試すべき比較器ネットワークの個数が激減した.

もう一つは, 充足可能性問題 (SAT) での定式化と, SAT ソルバーの利用である. また, SAT ソルバーを利用する前に, CNF の最適化を BEE(Ben-Gurion University Equi-propagation[*1], [6]) を使っておこなっている.

2. 提案手法

2.1 基本的アイデア

我々は, ソーティングネットワークかどうかの判定 (以下では, SN 判定と呼ぶ) に着目した. これまでの研究では, SN 判定は SAT の充足性や SAT ソルバーを利用しておこなわれている. また, 定理 1 を利用し, 入力をされるデータを $n!$ 通りから $2^n - (n+1)$ 通りに減らすことをおこなっている. $(n+1)$ は初めからソートされているビット列の数である.

今, n ビット列を $b_1 b_2 \ldots b_n$ ($b_i = 0$ または $b_i = 1$) で表すことにする. 比較器を $c_{i,j}$ を n ビット列に適用すると, $b_i = 0$ かつ $b_j = 1$ の時のみ, i ビット目と j ビット目が交換される. 比較器ネットワークは, 比較器の列からなる. そのため, 比較器ネットワークに n ビット列を入力するというのは, n ビット列に比較器を順次適応し, $11\ldots10\ldots0$ にする過程と言える. つまり, ソーティングネットワークの場合は, 2^n 個の n ビット列は $(n+1)$ 個のビット列に収束していく.

そのため, 比較器ネットワークを通過していく中で, ビット列の数は急速に減少していく. 例えば, $n = 12$ の場合, 元々のビット列は $4096 (= 2^{12})$ 個ある. 最初のレイヤ $(c_{1,2}, c_{3,4}, \ldots, c_{11,12})$ を適用すると, 19 個のビット列がソート済みとなるので, 4077 個のビット列は未ソートである. しかし, これらは 13 個のビット列に収束していくため, 重なりが非常に多くなる. 実際に考えなければならないビット列は, 716 個である. これらのソートされていないビット列を**残余ビット列**と呼ぶ. さらに, 2 番目のレイヤ $(c_{1,11}, c_{2,4}, c_{3,5}, c_{6,8}, c_{7,9}, c_{10,12})$ を適用すると, 33 個のソート済みビット列と 308 個の残余ビット列になる. 3 番目のレイヤ $(c_{1,3}, c_{2,10}, c_{4,12}, c_{5,11})$ を適用すると, 61 個のソート済み列と 160 個の残余ビット列となる. つまり, 比較器を追加していくと, 残余ビット列が単調に減っていく. もし, 残余ビット列が無くなると, 全てのビット列がソートされたことになり, ソーティングネットワークが完成する.

この残余ビット列を比較器ネットワークやソーティ

[*1] http://amit.metodi.me/research/bee/

ⓒ 2020 Information Processing Society of Japan

表 1 既知の比較器の最小数 $S(n)$ とレイヤ数の最小値 $T(n)$

# channel($=n$)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$S(n)$	0	1	3	5	9	12	16	19	25[3]	29[3]	35	39	45	51	56	60
$T(n)$	0	1	3	3	5	5	6	6	7	7	8[4]	8[4]	9[4]	9[4]	9[4]	9[4]

表 2 最小レイヤのソーティングネットワークとなる可能性のある 2 レイヤの数 [4]

n	3	4	5	6	7	8	9	10	11	12	13
$\lvert G_n\rvert$	4	10	26	76	232	764	2620	9496	35696	140152	568504
$\lvert R_n\rvert$	1	2	4	5	8	12	22	21	48	50	117

n	14	15	16	17	18	19
$\lvert G_n\rvert$	2390480	10349536	46206736	211799312	997313824	4809701440
$\lvert R_n\rvert$	94	262	211	609	411	1367

ングネットワークの表現として提案する. 形式的に残余ビット列を定義する.

定義 1. 残余ビット列とは, n ビット列の集合 (n はチャンネル数) で, 全ての n ビット列に比較器列 $c_{i_1,j_1}, c_{i_2,j_2}, \ldots, c_{i_m,j_m} (i_k, j_k \in [1,n])$ を適用後にソートされていないビット列の集合である.

しかし, 2^n 個の集合を持つためには, 非常に多くのメモリが必要となる. そこで, ビット列を効率的に保持することができるデータ構造 BDD(binary decison diagram, 二分決定グラフ), 特に ZDD[7] や SeqBDD[8] を用いることを考えた.

BDD は, 論理関数をグラフ構造で表したものである. 変数順序を固定することで, 部分グラフを共有することができる. そのため, 論理関数を圧縮して持つことができる. また, BDD 間の AND や OR などの二項論理演算をおこなうアルゴリズムがある. このアルゴリズムを使うことで, 効率的に BDD を構築することができる.

ZDD(Zero-suppressed BDD, ゼロサプレス型 BDD) は, 組合せ集合の処理をおこなうための BDD である. ZDD は疎な組合せ集合の場合に, 組合せ集合を非常にコンパクトに表現できる. ZDD には, 次にあげる有用な操作が定義されている. $P.offset(v)$(v を含まない組合せ部分集合を返す), $P.change(v)$(それぞれの組合せで v の存在を反転させる).

SeqBDD は, 文字列の集合を効率的に保持するためのデータ構造である. SeqBDD にも, 有用な操作が定義されている. 例えば, $P \cup Q$ (和集合を返す), $P \setminus Q$ (差集合を返す), $P.count$ (含まれている文字列の数を返す) などである.

以上の考えを元に比較器数による深さ優先探索のアルゴリズムを設計した. ただし, 実装として, SeqBDD を用いたもの, ZDD を用いたものがある. 以下の節ではこれらを順に説明していく.

2.2 SeqBDD を用いたアルゴリズム

SeqBDD で, ビット列を表現する場合は, 0 を表す文字 c_0 と 1 を表す文字 c_1 を作成する. 文字 c_0, c_1 からなる長さ n の文字列を使い, ビット列の集合を表現する. SeqBDD を用いた場合のアルゴリズムを, アルゴリズム 1 に示す.

アルゴリズム 1 比較器数最小のソーティングネットワークの計算 (SeqBDD)

Require: n: チャンネルの数;
1: min ← 十分大きな整数;
2: 最初のレイヤ $c_{1,2}, c_{3,4}, \ldots, c_{\lfloor n/2 \rfloor - 1, \lfloor n/2 \rfloor}$ を適用したビット列集合を計算し, SeqBDD s_0 として保持;
3: s_0 をスタックに代入;
4: **while** (スタックが空でない) **do**
5: スタックから SeqBDD s を取得;
6: 比較器 $c_{i,j}$ を加え, s を更新;
7: num ← s の比較器数;
8: **if** (num < min) **then**
9: **if** ($s.count$==0) **then**
10: s を出力し, min ← num;
11: **else**
12: (更新された) s をスタックに代入;
13: **end while**

アルゴリズム 1 の 6 行目では, 比較器を加えて, SeqBDD s を更新する. この更新を詳しく説明する. s に含まれる全てのビット列 $b = b_1 b_2 \ldots b_n$ に対して, i ビット目が 0 かつ j ビット目が 1 のときに, i ビット目と j ビット目の値を反転し, 新しいビット列 b' を作成する. b' がソート済みになれば, s から b を除く. また, b' が未ソートであれば, s から b を除き, b' を追加する. b' に含まれるビット列がなくなった時点の比較器列が, ソーティングネットワークとなる.

アルゴリズム 1 の最適性は, 全ての組み合わせの比較器を追加していくことで保証される. しかし, 比較器 $c_{i,j}$ を加えても, s に含まれる全てのビット列も

ⓒ 2020 Information Processing Society of Japan

変化しない場合は, $c_{i,j}$ は最小比較器数のソーティングネットワークには含まれないため, 追加する必要がない. ただし, $c_{i,j}$ はいくつかの交換が実行されたあとに, 追加される可能性はある.

アルゴリズム 1 は深さ優先探索で, 比較器を次々と加えていく. そのため, 計算を終わらせる比較器数を指定しておく必要がある. そこで, 最大の値を予め min に代入しておく (1 行目). 必ずしもこの値まで全ての比較器を代入する必要はない. 本アルゴリズムでは, 最小の比較器数をもつソーティングネットワークの生成を目的としているので, 1 つでもでソーティングネットワークが生成できた場合, その比較器数以上に比較器を追加する必要はない. そのため, min の値を更新し, 動的に計算の終了する深さを変化させている (10 行目).

2.3 ZDD を用いたアルゴリズム

アルゴリズム 2 は, SeqBDD を使っている部分を, ZDD に変更したアルゴリズムである.

アルゴリズム 2 比較器数最小のソーティングネットワークの計算 (ZDD)

Require: n: チャンネルの数;
1: min ← 十分大きな整数;
2: 最初のレイヤ $c_{1,2}, c_{3,4}, \ldots, c_{\lfloor n/2 \rfloor - 1, \lfloor n/2 \rfloor}$ を適用したビット列集合を計算し, ZDD z_0 として保持;
3: z_0 をスタックに代入;
4: **while** (スタックが空でない) **do**
5: スタックから ZDD z を取得;
6: 比較器 $c_{i,j}$ を加え, z を更新;
7: num ← z の比較器数;
8: **if** (num < min) **then**
9: **if** ($z.count$==0) **then**
10: z を出力し, min ← num;
11: **else**
12: (更新された) z をスタックに代入;
13: **end while**

基本的な部分は, アルゴリズム 1 と同じである. 違う部分は, 6 行目の更新をおこなう部分である. 今回実装に用いた SAPPORO BDD パッケージには, 次の演算がある.

`OnSet`(i)　i 番目の要素が含まれる ZDD を返す.
`OffSet`(i)　i 番目の要素が含まれない ZDD を返す.
`Change`(i)　i 番目の要素の値を反転した ZDD を返す.

取り出した z に対し, `OffSet`(i), `OnSet`(j) を適用することで, i 番目のビットが 0 であり, j 番目のビットが 1 である部分 ZDD z_{ij} を抽出できる. また, `Change`(i) を利用することで, z_{ij} の i 番目と j 番目のビットを反転させた ZDD z'_{ij} を得ることができる. すなわち, 更新を ZDD の演算だけでおこなうことができる.

$$z' = z - z_{ij} + z'_{ij} - sortedZdd$$

ただし, $sortedZdd$ は, ソート済みのビット列の全体である.

2.4 ZDD とハッシュを用いたアルゴリズム

アルゴリズム 2 を実行した際, 全く同一の ZDD が生成されることがある. 次の補題が成立する.

補題 1. 任意の ZDD z に対し, $\{i_1, j_1\} \cap \{i_2, j_2\} = \emptyset$ のとき,

$$z + c_{i_1,j_1} + c_{i_2,j_2} = z + c_{i_2,j_2} + c_{i_1,j_1}.$$

証明. c_{i_1,j_1}, c_{i_2,j_2} でおこなわれる交換は, それぞれ チャンネル i_1 と j_1, チャンネル i_2 と j_2 である. $\{i_1, j_1\} \cap \{i_2, j_2\} = \emptyset$ であるため, 2 つの比較器のおこなう交換は互いに影響がない. よって, 適用する順番に依らず, 操作によりできる ZDD は同一となる. □

すなわち, 同じ状態の ZDD が生成され, もう一度比較器が追加されていくことになる. この状態を避けるために, ハッシュを用いる. SAPPORO BDD では, 同一の ZDD を作成しないように, ZDD の識別子が割り振られている. この識別子は, `GetID`(z) という命令で取得できる. そこで, `GetID`(z) をハッシュ値とし, ハッシュ表 `H[]` を次で作成する.

`H[GetID`(z)`]` ← z

`H[]` に既に ZDD がある場合には, その ZDD は展開しないことで大幅な高速化が見込まれる. ただし, 同一の ZDD であっても, 少ない比較器数で到達することもある. その場合には, ハッシュ表を上書きする必要がある. ハッシュを組み込んだアルゴリズムを, アルゴリズム 2' と呼ぶことにする.

3. 計算機実験

アルゴリズム 1, アルゴリズム 2 及びアルゴリズム 2' を C++言語と SAPPORO BDD パッケージを用いて, 実装した. 表 3 は, これらの実装の出力結果と計算時間である.

各アルゴリズムごとに, 出力されたソーティングネットワークの中で最小の比較器数を記述してる (表

ⓒ 2020 Information Processing Society of Japan

表 3　アルゴリズム 1, アルゴリズム 2 及び アルゴリズム 2' による実験結果

チャンネル数	n	7	8	9	10	11	12
既知の最小値	$S(n)$	16	19	25	29	35	39
アルゴリズム 1	(比較器数)	16	19	27	38	53	63
(SeqBDD)	(計算時間)	1 h	24.5 h	実行中	実行中	実行中	実行中
アルゴリズム 2	(比較器数)	16	19	27	39		
(ZDD)	(計算時間)	1.3 h	6 days	実行中	実行中		
アルゴリズム 2'	(比較器数)	16	19	計算不可			
(ZDD+Hash)	(計算時間)	291 ms	4971 ms	-			

3, 3 行目, 5 行目, 7 行目). また, 計算の終了までの時間を記載している (4 行目, 6 行目, 8 行目). まだ計算の終わっていない実験もある. そのため, 幾つかの欄には, 現在までの計算で得られた最小の比較器数を記載している.

また, 表 4 に, 今回の計算機実験に用いた計算機のスペックと使用言語を示す.

表 4　計算機のスペックと使用言語

CPU	Core i7-8086K(12 コア, 4GHz)
メモリ量	32GB
ビデオカード	GeForce GTX 1050Ti
OS	Ubuntu 18.04LTS
言語	C++言語 (g++ 7.4.0)
使用ライブラリ	Boost, SAPPORO BDD

4. 議論

表 3 について議論をおこなう. いずれのアルゴリズムであっても, チャンネル数 n が 7 と 8 の場合は, 最小比較器をもつソーティングネットワークを生成できている. また, 計算が終了していることから, $n = 7$ 及び, $n = 8$ の場合は, 最小比較器の数が 16 及び 19 であることも示したことになる.

チャンネル数が 9 以上になると, ソーティングネットワークを生成することはできているが, 最小比較器数のソーティングネットワークは得られていない. アルゴリズム 1 の計算中となっている部分は, 約 160 日程度の計算を続けているが, 終了が見えない状態である (2020 年 2 月現在). 特に, $n = 11, 12$ の場合は, 2019 年 9 月頃から変化がみられない. これらの計算は, 使用メモリが 30MB から 60MB 程度 (表 4 のメモリ量の 0.1%から 0.2%) であり, メモリの使用量は少ない. メモリの使用状態は, アルゴリズム 2 でもほぼ同様である.

一方で, アルゴリズム 2' は他のアルゴリズムに比べ, 非常に高速である. 既に計算をおこなった ZDD をハッシュとして保持しているため, 同じ計算をおこなわないためである. その効果を見るために, 図 2 を用意した. 図 2 は, アルゴリズム 2' をチャンネル数 8 で実行した場合に生成された ZDD の個数と使用ノードの最大値のグラフである. 使用ノードは, ZDD の節を表しており, 同一の節はいくつかの ZDD で使われていても, メモリ上に 1 個の使用ノードとして存在する.

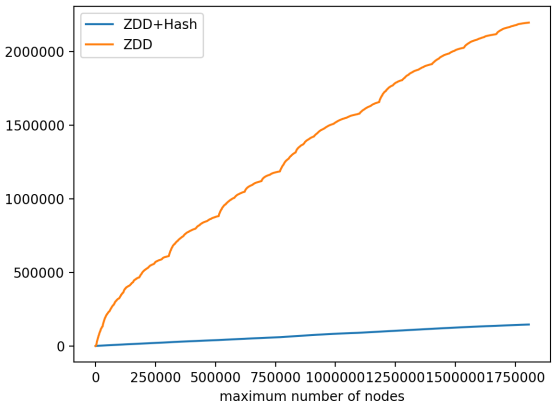

図 2　ノードの最大値と作成された ZDD 数

アルゴリズムが進むにつれ, 新たな ZDD が作成されるため, ノード数は単調に増えていく. 実行の終了までに, 1,738,534 個のノードを使い, 115,048 個の相異なる BDD を生成した. そのため, 1 つの BDD には, 平均で 15.1 個のノードが使われていることになる. 一方で, アルゴリズム 2 の場合は, 使用ノード数は同じであるが, 生成された BDD は, 2,195,993 個である. よって, ハッシュを使うことで, BDD の生成回数が 5.2%程度になっていることがわかる.

$n = 9$ の場合, 計算の途中でメモリが足りなくなる. ZDD を作成する部分は, アルゴリズム 2 と同一であるため, ハッシュの部分でメモリを使用していると考えられる.

ⓒ 2020 Information Processing Society of Japan

5. まとめ

提案手法はいずれもソーティングネットワークを生成できている. チャンネル数が 8 以下の場合は, 既知の結果が最小であることも示せている. アルゴリズム 1 と アルゴリズム 2 は, チャンネル数が 9 から 12 の場合にもソーティングネットワークを生成している. しかしながら, その計算は非常に遅い. 現在, 9 個のプログラムを同時に動かしている. プロセスを見る限り, それぞれのコアをほぼ100%占有し, プログラムは動作している. また, 32GB のメモリは, 606MB(約 18.5%) が使われ, 26.5GB(約 80.7%) がバッファとして, 22.5MB がスワップとして使われている.

アルゴリズム 2' は, 他のアルゴリズムと比較して非常に高速であるが, 非常に大きなメモリを使っている. そのため, チャンネル数が 9 を超えるとメモリ不足で間違った計算をおこなう.

我々の生成したい最小比較器数のソーティングネットワークは, チャンネル数が 11 から 13 程度のものである. そのため, 現状の手法では, 目的のソーティングネットワークは生成できていない. 現状の手法を元に, 目的とするソーティングネットワークを生成するには, さらなる改善が必要となる. 例えば, 既存の最小比較器数を組み込み, それ以上の比較器数になる場合は, 計算を打ち切るという方法がある. また, 既存の最小比較器数のソーティングネットワークは, 途中でチャンネル 1 とチャンネル n が使われなくなることがある. この場合, チャンネル数が $n-2$ チャンネルとなることで, 計算が減ることとなる. これを実現するには, 上記の状態を満たす最小の比較器列を作成し, 残ったビット列の集合を出力する. その上で, 先頭ビットと最後尾ビットを削除したビット列を新たな入力とし, 計算を続けることが必要となる.

一方で, メモリの効率を考えると, SAPPORO BDD のパッケージを使わず, ビット演算で全てを再実装するという手法も考えられる. これは, 一つの ZDD サイズがさほど大きくなく, 圧縮の効果がさほどないことがわかっているからである. ただし, 有用な演算が無くなるため, ビット列に対する操作やハッシュ値の計算などを自身で実装する必要がある.

謝辞 本研究の多くの部分は, 第一著者の北海道大学での滞在時におこなわれたものである. 第一著者の滞在を, 快く受け入れていただいた北海道大学大学院情報科学研究科[*2]湊真一教授に深謝いたします.

[*2] 現, 京都大学大学院情報学研究科

参考文献

[1] Donald E. Knuth, The Art of Computer Programming, Volumes 3: Sorting and Searching, Addison-Wesley, 1997.
[2] José A. R. Fonollosa, Joint Size and Depth Optimization of Sorting Networks, arXiv:1806.00305.
[3] Michael Codish, Luís Cruz-Filipeb, Michael Frank, Peter Schneider-Kamp, Sorting nine inputs requires twenty-five comparisons, Journal of Computer and System Sciences, Vol. 82(2016), pp.551-563.
[4] Daniel Bundala, Michael Codish, Luís Cruz-Filipe, Peter Schneider-Kamp, Jakub Závodný, Optimal-depth sorting networks, Journal of Computer and System Sciences, Vol. 84(2017), pp.185-204.
[5] Michael Codish, Luís Cruz-Filipeb, Thorsten Ehlersc, Mike Müller, Peter Schneider-Kamp, Sorting networks: To the end and back again, Journal of Computer and System Sciences, Vol. 104(2019), pp.184-201.
[6] Amit Metodi, Michael Codish, Compiling finite domain constraints to SAT with BEE, TPLP Vol. 12(4–5)(2012), pp.465–483.
[7] Shin-ichi Minato, Zero-suppressed BDDs for set manipulation in combinatorial problems, *Proc. 30th ACM/IEEE Design Automation Conference (DAC'93)*, pp.272-277(1993).
[8] Shuhei Denzumi, Ryo Yoshinaka, Hiroki Arimura and Shin-ichi Minato, Sequence binary decision diagram: Minimization, relationship to acyclic automata, and complexities of Boolean set operations, Discrete Applied Mathematics, vol. 212(2016), pp.61-80.

ⓒ 2020 Information Processing Society of Japan