1D-2

最良解保持による BDD 動的変数順序付けの高速化

柏 多恵子 † 梶山 浩嗣 † 佐田 宏史 † 前川 仁孝 † 伊與田 光宏 †
† 千葉工業大学 情報科学部 情報工学科

1 はじめに

Binary Decision Diagram (以下, BDD) [1] [2] は論理関数を木構造により効率良く表現する手法として, LSI の論理設計などの分野で広く応用されている. 論理関数の演算は, BDD のサイズに比例して実行時間が増加するため, 論理関数を小さい BDD で表現することが重要である. BDD を縮小化する手法として動的変数順序付け [3] があるが, 大規模なデータに対しては実行時間が長くなるため, 高速化が望まれている. 従来の動的変数順序付けでは, 注目する変数が探索後に発見した最良なレベルにない場合は, 最良なレベルに移動する必要がある. 従来の動的変数順序付けでは, 解探索後の変数移動の処理量は, 解探索後の注目する変数の位置から解の位置までに存在する節点数に依存する. このため, BDD の節点数が多い場合は処理量も多くなる傾向がある. そこで本稿では, 従来の動的変数順序付けに対して, 最良解への移動処理を削減することで実行時間を短縮する手法を提案する. 本手法では, 解探索時は常に最良解を保持し, 探索終了後, 保持した解をコピーすることで, 解探索後の発見した最良解への移動を削除する.

2 BDD

BDD は変数節点, 定数節点, 0 枝, 1 枝から構成される.図 1 は論理関数 f=$x_1x_2 + x_3x_4 + x_5x_6$ を表す BDD である. 論理関数中の各変数に与えられる値 0, 1 に対し, 変数の値が 0 の場合は 0 枝を, 1 の場合は 1 枝を辿り, 辿り着いた定数節点の値がその関数の解となる. BDD における変数の出現する順序を変数順序という. また, BDD における根節点からの変数の深さをレベルという.

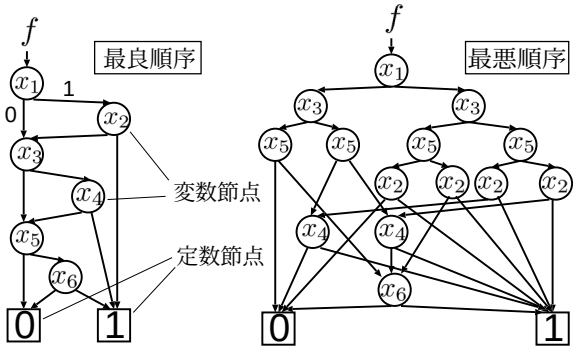

図 1: BDD の変数順序による違い

図 1 のように, BDD は変数順序により大きさや形状が異なるという特徴がある. 例えば, 図 1 の左側の BDD は変数順序が $\{x_1, x_2, x_3, x_4, x_5, x_6\}$ の場合であり, 右側の BDD は変数順序が $\{x_1, x_3, x_5, x_2, x_4, x_6\}$ の場合である. 変数順序により大きさが変化するという BDD の特徴を利用して BDD を縮小化する変数順序を決定する手法に, 静的変数順序付けと動的変数順序付けがある. 静的変数順序付けは BDD を構築する前に変数順序を決定する手法である. 静的変数順序付けは動的変数順序付けと比べて実行時間は短くなる. しかし, 入力となる論理関数の記述に依存するため, 動的変数順序付けと比べて良好な解を得ることが難しい. このため, BDD の変数順序決定には, BDD 構築途中あるいは構築後に変数順序を決定する動的変数順序付けが利用されることが多い.

3 動的変数順序付け

動的変数順序付けの一手法である sifting アルゴリズム [4] は, 図 2 のように, 隣接する変数 (x_i, x_j) の変数順序を交換 (swap 操作) しても, 他のレベルのノードには影響しないという BDD の性質を利用し, swap 操作を繰り返すことでヒューリスティックに近似解を探索する.

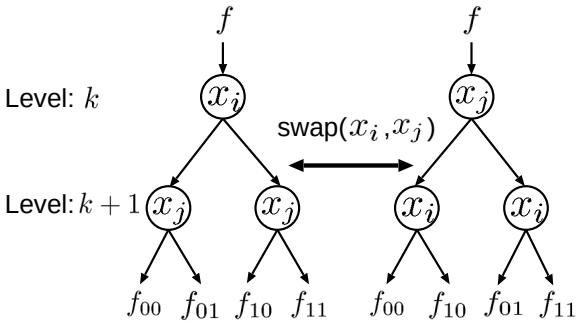

図 2: 変数 x_i, x_j の swap 操作

n 変数論理関数の BDD を sifting により縮小化する場合, 以下の操作を n 個の変数に対して繰り返す.

1. 注目する変数 x_i を決定する.
2. 他の変数は固定し, x_i のみ swap 操作を用いて上下に移動させ, BDD が最小になる最良なレベルを探索する.
3. 2 で求められたレベルに x_i を移動する.

通常は BDD 中でノード数が多い変数から sifting が行われる. このため 2 の処理では, 探索における移動量を減らすため, 注目する変数がグラフの中間より下に存在する場合は下方向に, 上に存在する場合は上方向に探索を開始する.

具体例として, 図 3 に入力変数が 4 個である場合の sifting の処理を示す. ここで, 注目する変数を x_3 と仮定する. x_3 はグラフの中間より下に存在しているので, 下方向への探索から開始する (step1 から step2). x_3 が一番下のレベルに到達した後, 一番上のレベルまで x_3 を移動することで BDD が小さくなるような変数順序を探索する (step3 から step5). 探索の際, BDD が一番小さくなった x_3 のレベルを保存する. 探索後, BDD の変数順序が探索で得られた最良解とは異なる場合は, 再び x_3 を探索時に保持しておいた最良なレベルまで移動する (setp6 から step8).

Fast Dynamic Variable Ordering for BDD by Preserving the Best Solution
† Taeko Kashiwa, Hirotsugu Kajiyama, Hiroshi Sata, Yoshitaka Maekawa, Mitsuhiro Iyoda
Dept. of Computer Science, Chiba Institute of Technology

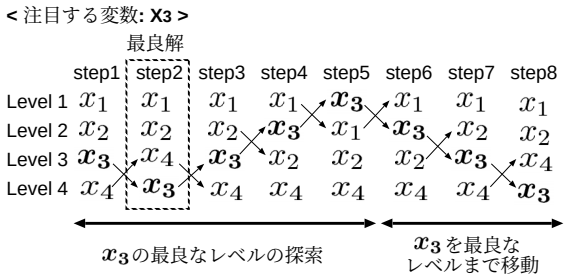

図 3: sifting の例

4 最良解への移動を削除した sifting

sifting の処理は, 注目する変数の最良なレベルの探索と, 注目する変数を探索時に得た最良なレベルまで移動する処理に分割できる. 探索後の移動にかかる処理量は, 探索後の注目する変数のレベルから探索時に発見した変数の最良なレベルに含まれる節点数に依存する. このため, 探索で発見した最良なレベルが探索直後の変数から遠い場合, 探索後の移動にかかる時間が長くなる. そこで, 探索後の変数移動の処理量が大きい場合には探索時に最良解である BDD を一時保持し, 探索後に図 4 の step6 のように一時保存した BDD をコピーすることで処理時間を削減する. 一時保持する解は, 探索が終了するまで, より良い解が発見される度に更新する. また, 探索により発見した最良なレベルが, 探索直後の変数が存在するレベルに近い場合, 最良解までの移動における処理量が少なくなる. この場合は, 変数を移動するコストよりも一時保存した最良解をコピーするコストの方が大きくなる. よって, 解が探索後の変数のレベルに近い場合は探索終了後, 従来の sifting と同様に変数を移動する. 探索後に移動またはコピーするかについては, 探索直後の変数のレベルと最良なレベルの差により決定する.

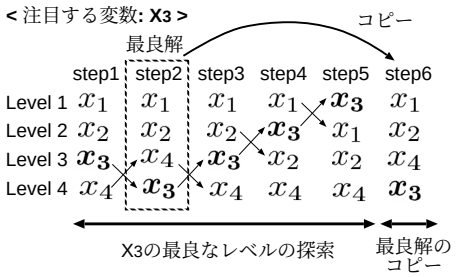

図 4: 最良解への移動を削除した sifting の例

計算機上の BDD はノードテーブルで各ノードが管理される [5]. このため, 最良解である BDD の保持はノードテーブルを別に一時格納することで実現する. また, BDD の処理系では, ノードにアクセスをする際, ハッシュテーブルを利用するので, ノードテーブルとハッシュテーブル間では整合性を保たなくてはならない. このため, 探索終了後には一時保存した最良解におけるノードテーブルのコピーに加え, ハッシュテーブルを書き換える.

5 性能評価

評価には, CPU に Xeon 3.6GHz, メモリが DDR2-400 2GB, OS に Linux2.6.10 を使用した. 評価に用いたデータは論理合成ベンチマーク LGSynth93 [6] である. 表 1 に従来手法と提案手法による全体の実行時間を示す. また, 表 2 に従来手法における探索後の移動時間と提案手法における探索時の解の更新時間, 探索後の最良解のコピー時間, コピーが実行されない場合に行われる移動時間を示す. 表 2 中の従来手法における探索後の移動は, 提案手法での探索時の解の更新と, 最良解のコピーとコピーが実行されない場合に行われる移動に相当する. 本稿では提案手法において, 探索直後の変数のレベルと最良なレベルの差の閾値を 1/2 として評価した.

表 1: 従来手法と提案手法における全体の処理時間

データ	従来 [s]	提案 [s]	高速化率 [倍]
i10	48.50	45.79	1.06
apex5	14.62	13.62	1.07
pair	30.61	28.81	1.06

表 2: 従来手法と提案手法の探索後の処理時間と更新時間

	従来	提案			
データ	移動 [s]	更新 [s]	コピー [s]	移動 [s]	合計 [s]
i10	6.28	0.04	1.72	1.82	3.58
apex5	2.46	0.07	0.72	0.66	1.45
pair	4.15	0.06	1.37	0.92	2.35

表 1 より, 提案手法では, 従来手法の実行時間と比べて, 1.06 倍から 1.07 倍の高速化が確認できた. これは表 2 に示すように, 探索後に一時保存した最良解をコピーすることによって, 従来手法における探索後の移動時間が短縮したためである. また, 提案手法では, 最良解の位置が探索直後の変数の位置から離れている場合に解をコピーするため, 探索時に一時保持する解を, 最良解発見の度に更新する必要がある. しかし, 探索時の解の更新によるコストは, 表 2 から分かるように, 提案手法におけるコピーと移動時間に対して十分小さく, 無視することができる. このため, 提案手法は, 従来手法における探索後の移動時間が短縮できたので, 高速化できたと考えられる.

6 おわりに

本稿では, 解の探索時に最良解である BDD を保持し, 探索後に一時保持した BDD をコピーすることで, 探索後の移動を削除する手法を提案し, 評価した. 評価の結果, 探索後に最良解をコピーすることで, 従来手法に比べ 1.06 倍程度の高速化が得られたことが確認できた. 探索後に移動またはコピーを実行するかを決定する閾値に関しては, 今後有効な値を検討する必要がある.

参考文献

[1] 石浦 菜岐佐, “BDD とは”, 情報処理, Vol.34, No.5, pp.585-592, 1993.

[2] Bryant,R.E. “Graph-based Algorithms for Boolean Function Manipulation”, IEEE Transactions on Computers, Vol.35, No.8, pp.677-691, 1986.

[3] R.Rudell, “Dynamic Variable Ordering for Ordered Binary Decision Diagrams”, In Proceedings of the International Conference on Computer-Aided Design, pp.42-47, 1993.

[4] 藤田 昌宏, “フォーマル・ベリフィケーションの基本原理 (3) ”, Design Wave Magazine, システム LSI 検証技術入門, CQ 出版社, 2003.

[5] 湊 真一, “計算機上での BDD の処理技法”, 情報処理, Vol.34, No.5, pp.593-599, 1993.

[6] http://www.bdd-portal.org/