TdZdd: フロンティア法のための効率的なトップダウンZDD構築を実現するC++ライブラリ

岩下 洋哲

JST ERATO 湊プロジェクト

2013.11.25　ERATO 合宿＠登別

歴史的なこと

フロンティア法 → 対象拡大 / 一般化 [川原]

SAPPOROBDD [湊] → Graphillion [井上 2013]

リンクパズル [吉仲 2012]

対象ごとの実装 — アプリケーション拡充 → TdZdd

共通部分のライブラリ — アルゴリズム改善 → TdZdd

19 × 19

計数専用 21 × 21 [2012]

並列化 [竹内 2012 など]

おねえさん専用 25 × 25 [2013]

Outline

1. はじめに
2. ライブラリの機能
3. ライブラリの性能
4. おわりに

Outline

1. はじめに
2. ライブラリの機能
   - 再帰的仕様記述
   - 仕様の合成
   - 仕様からの ZDD 構築
3. ライブラリの性能
4. おわりに

トップダウン ZDD 構築

例: 5 個から 2 個選ぶ組み合わせ

Construction

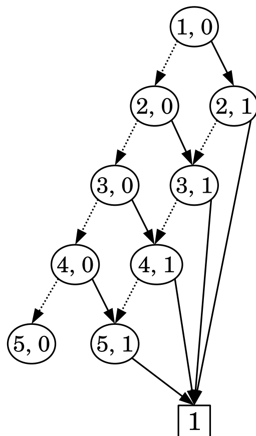
Reduction

トップダウン ZDD 構築の “仕様”

例: n 個から k 個選ぶ組み合わせ

$Comb_{n,k}$.ROOT()
1: **return** $\langle 1, 0 \rangle$;

$Comb_{n,k}$.CHILD($\langle i, s \rangle$, b)
1: $s \leftarrow s + b$;
2: **if** $s = k$, **return** $\boxed{1}$;
3: **if** $i = n$, **return** $\boxed{0}$;
4: **return** $\langle i + 1, s \rangle$.



フロンティア法も“仕様”で記述

例: パス列挙

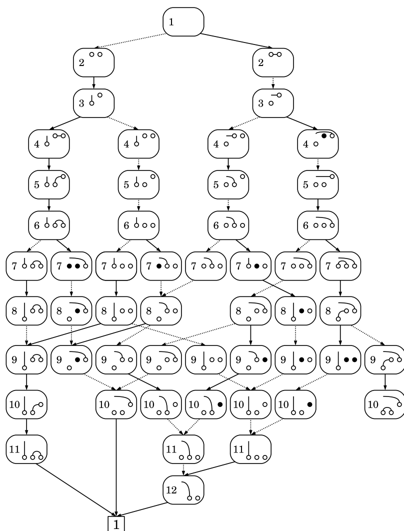
$Path_G$.ROOT()
1: **return** $\langle 1, mate_0 \rangle$;

$Path_G$.CHILD($\langle i, mate \rangle$, b)
1: 辺 i の使用 ($b = 1$) ／不使用 ($b = 0$) に応じて *mate* を更新;
2: **if** パスが完成, **return** $\boxed{1}$;
3: **if** パス完成の見込みなし, **return** $\boxed{0}$;
4: **return** $\langle i + 1, mate \rangle$.

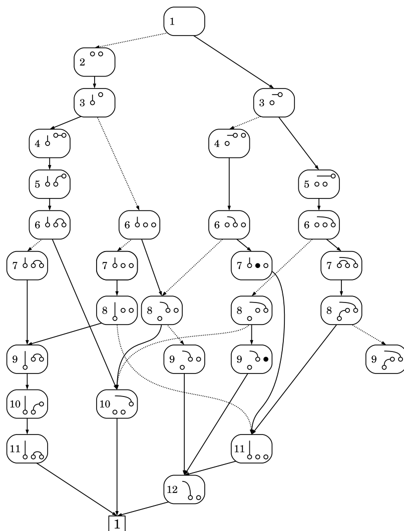
仕様を返す関数

INTERSECT(S, T)	⋯	仕様 S と仕様 T の intersection
WRAP(f)	⋯	ZDD f を仕様とみなす
LOOKAHEAD(S)	⋯	仕様 S にゼロサプレス規則を適用

Lookahead の例



$Path_G$ から構築 (52 ノード)



LOOKAHEAD($Path_G$) から構築 (29 ノード)

Lookahead のしくみ

LOOKAHEAD(S).ROOT()

1: **return** S.ROOT();

LOOKAHEAD(S).CHILD($\langle i, s\rangle$, b)

1: $\langle i', s'\rangle \leftarrow S$.CHILD($\langle i, s\rangle$, b);
2: **while** ラベル $\langle i', s'\rangle$ を持つノードの 1 枝側の子が $\boxed{0}$ **do**
3: $\quad\langle i', s'\rangle \leftarrow S$.CHILD($\langle i', s'\rangle$, 0);
4: **end while**
5: **return** $\langle i', s'\rangle$.

ZDD を返す関数

CONSTRUCT(S)	$\cdots$	仕様 S に対応する ZDD を構築
SUBSET(f, S)	$\cdots$	ZDD f を制約として ZDD を構築

Subsetting

- 次の 3 つの結果は同じ
  (1) $\textsc{Subset}(f, S)$
  (2) $\textsc{Construct}(\textsc{Intersect}(\textsc{Wrap}(f), S))$
  (3) $f \cap \textsc{Construct}(S)$
- (1) (2) では S の探索が f を満たす空間に限定される
  - $\textsc{Construct}(S)$ が爆発する場合でも計算できることがある
- (1) は (2) より少し効率が良い

Outline

1 はじめに

2 ライブラリの機能

3 ライブラリの性能
- 基本性能
- 並列性能

4 おわりに

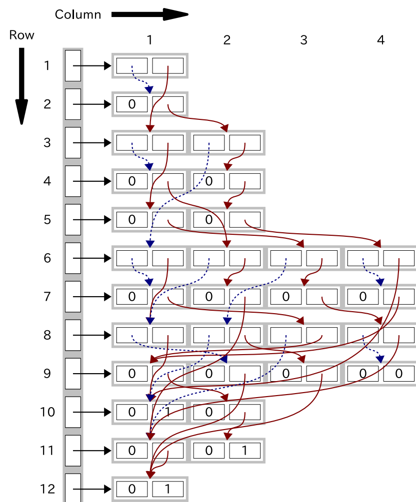
出力を除く Simpath 実行時間を 1 とした相対実行時間

11
10
9
8
7
6
5
4
3
2
1
0

■ Simpath （出力あり）＋ Simpath-Reduce
■ TdZdd
■ TdZdd w/ Lookahead

K17 K18 K19 K20 K21 K22 T11x11 T12x12 T13x13 T14x14 T15x15 T16x16 G13x13 G14x14 G15x15 G16x16 G17x17 G18x18 H22x23 H24x25 H26x27 H28x29 H30x31 H32x33

ZDD のデータ構造



主な構成

- ノード ID ・・・ 8 bytes（20-bit row index + 43-bit column index）
- ノード ・・・ 16 bytes（ノード ID × 2）
- ZDD ・・・ レベル毎のノード配列を保持する配列

データの局所性を意識した幅優先処理

8 段め構築処理開始時

データの局所性を意識した幅優先処理

Hashtable for row 8

ハッシュテーブルによるノード共有判定

データの局所性を意識した幅優先処理

Child 関数の呼び出し

逐次 Construction 処理中の時間消費率

100%
90%
80%
70%
60%
50%
40%
30%
20%
10%
0%

G13x13 (240MB)　G14x14 (770MB)　G15x15 (2.4GB)　G16x16 (7.6GB)　G17x17 (24GB)

- その他
- Child 関数の評価（実行回数 = 枝数）
- ハッシュテーブル要素の検索と挿入（実行回数 = 非終端ノードへの枝数）

呼び出し１回あたりの消費時間（ナノ秒）

Time/Call (nanosec)

180
160
140
120
100
80
60
40
20
0

G13x13 (240MB)
G14x14 (770MB)
G15x15 (2.4GB)
G16x16 (7.6GB)
G17x17 (24GB)

Hash
Child

共有メモリ並列処理

ハッシュコードを用いて状態を複数バケットに分類／並列処理

スレッド数と処理時間　G16x16 (7.6GB〜12GB)

Time (sec)

300
250
200
150
100
50
0

Reduction
Construction

Seq 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30

Number of threads

1スレッド実行に対する速度比

Chart: Speedup (y-axis, 0–20) vs. Number of threads (x-axis, 0–30)

Legend:
- G19x19 (240GB~250GB)
- G18x18 (76GB~82GB)
- G17x17 (24GB~30GB)
- G16x16 (7.6GB~12GB)
- G15x15 (2.4GB~4.7GB)
- G14x14 (770MB~1.8GB)
- G13x13 (240MB~820MB)

おわりに

- 一部の機能は Graphillion から利用できます。
  - cliques, connected_components, cycles, forests, graphs, paths, trees $\longrightarrow$ CONSTRUCT
  - 上の関数に graphset オプション引数を追加 $\longrightarrow$ SUBSET
  - 自前の “仕様” を与えることはできない
  - 並列処理はできない
- 全機能の C++ ライブラリとしての公開を計画しています。