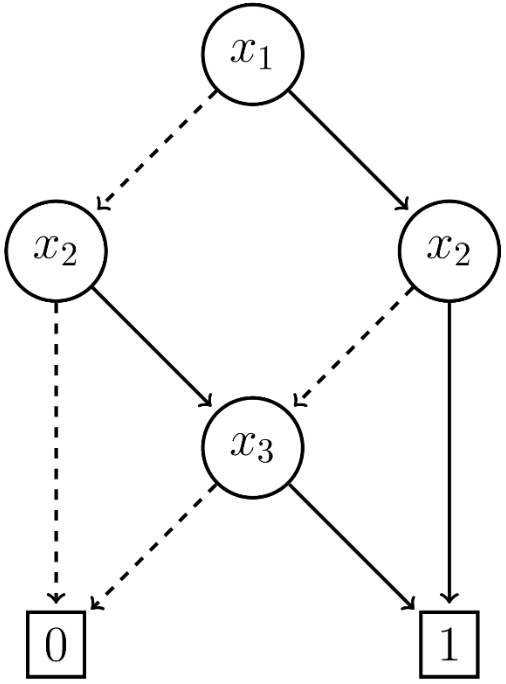
大規模知識処理特論 (第13回)
二分決定グラフの利用 (1)

北海道大学 情報科学研究院
堀山 貴史

復習: 二分決定グラフ (BDD)

- **有向非巡回グラフ**による論理関数の表現法
- **変数順序**
  - 全順序関係にしたがって、変数が出現
- 2つの簡約化規則を適用
  - **冗長な節点**の削除 / **等価な節点**の共有
  - **既約化**:
    冗長/等価な節点がなくなるまで

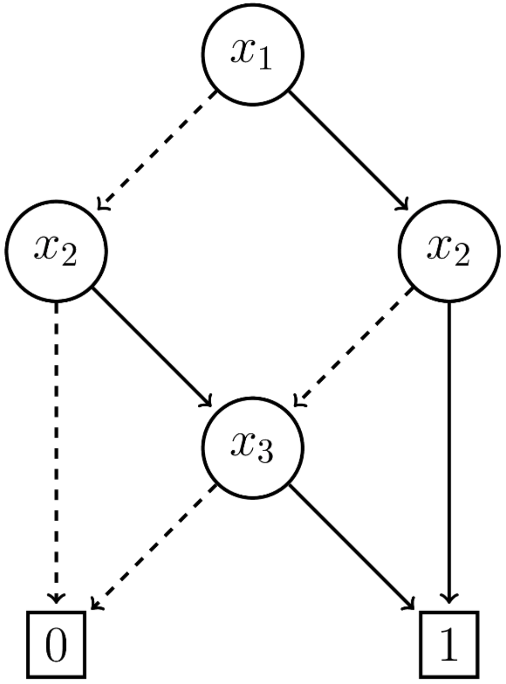


資料「論理関数の表現方法」を参照

Binary Decision Diagram

復習: 二分決定グラフ (BDD)

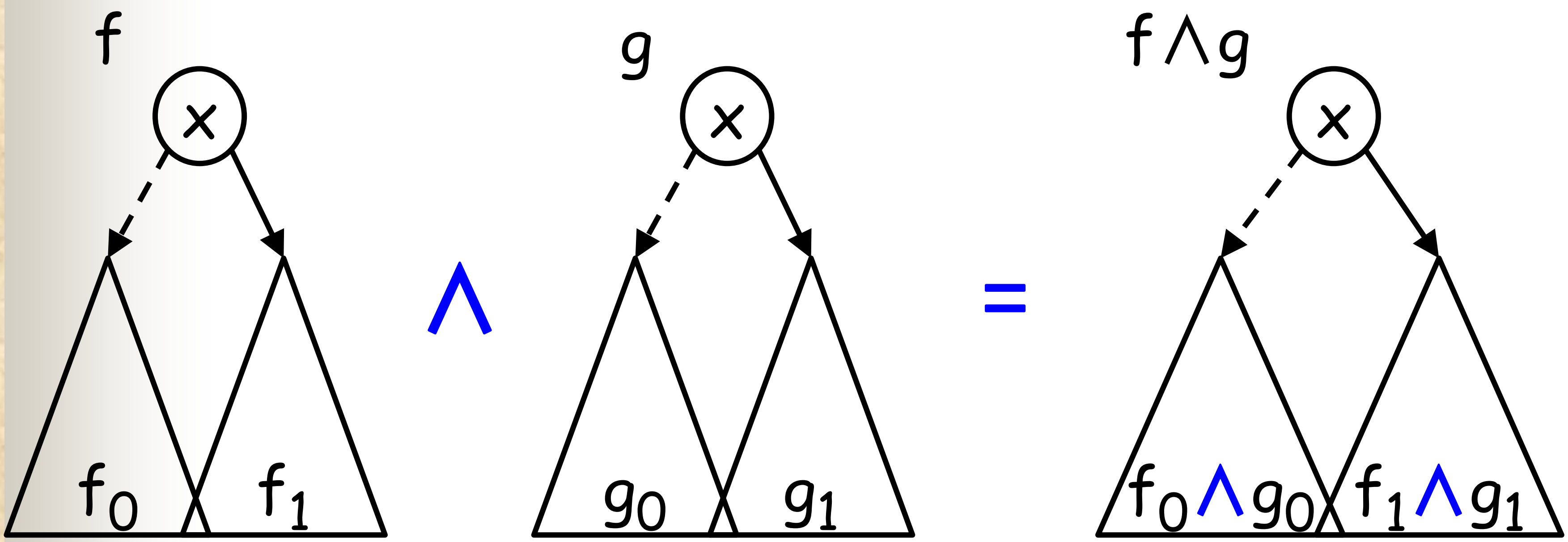
- **有向非巡回グラフ**による論理関数の表現法
- **変数順序**
  - 全順序関係にしたがって、変数が出現
- 2つの簡約化規則を適用
  - **冗長な節点**の削除 / **等価な節点**の共有
  - **既約化**:
    冗長/等価な節点がなくなるまで



- 変数順序を定めると**表現が一意**に定まる
- 多くの実用的な論理関数を**コンパクト**に表現
  (論理構造を効率的に圧縮して持てる)
- BDDに対する**効率的な演算** [Bryant 1986]
- 近年、様々な分野での応用に

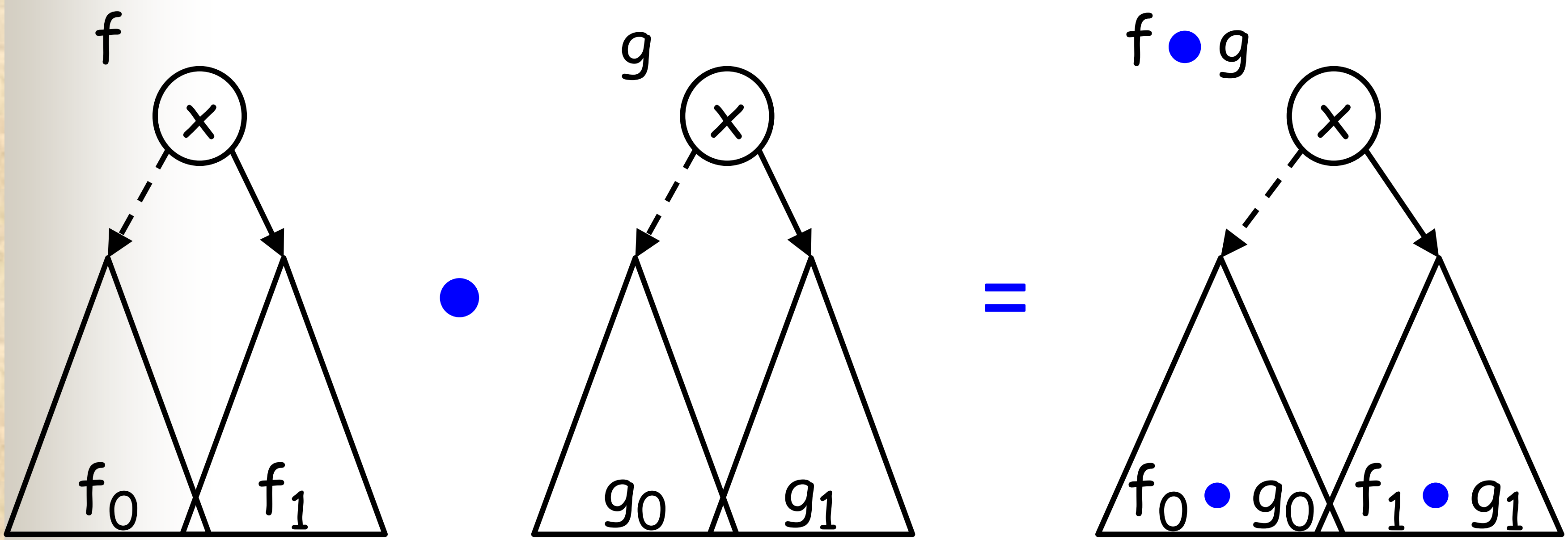
論理演算

- $f = \overline{x} f_0 \vee x f_1$ と
  $g = \overline{x} g_0 \vee x g_1$ との 論理積
- $f \wedge g = \overline{x}(f_0 \wedge g_0) \vee x(f_1 \wedge g_1)$



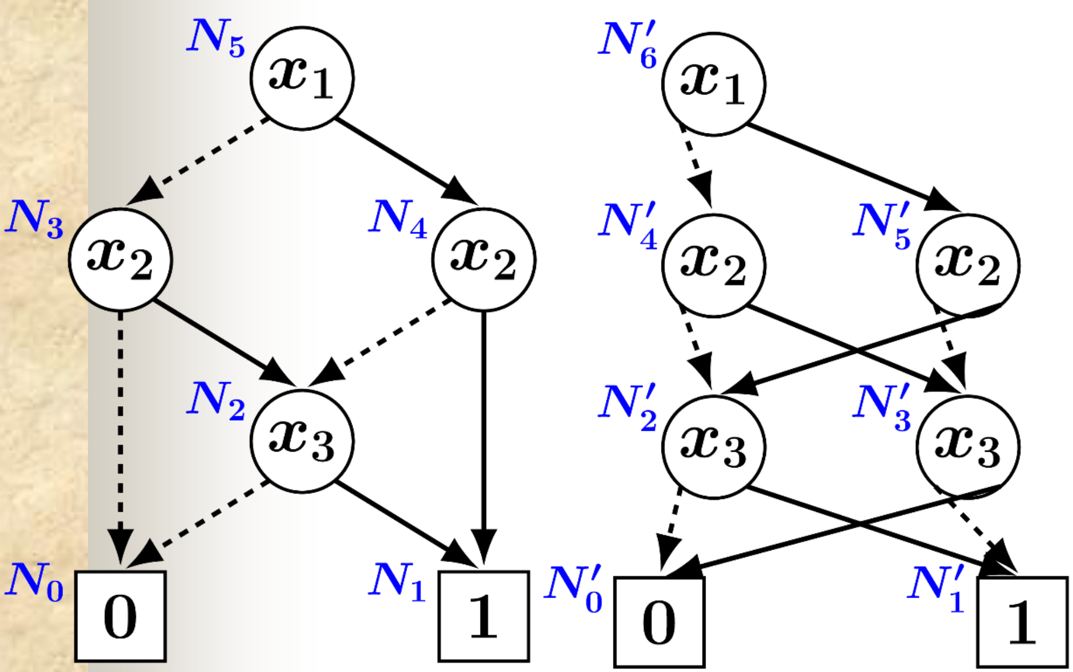
論理演算

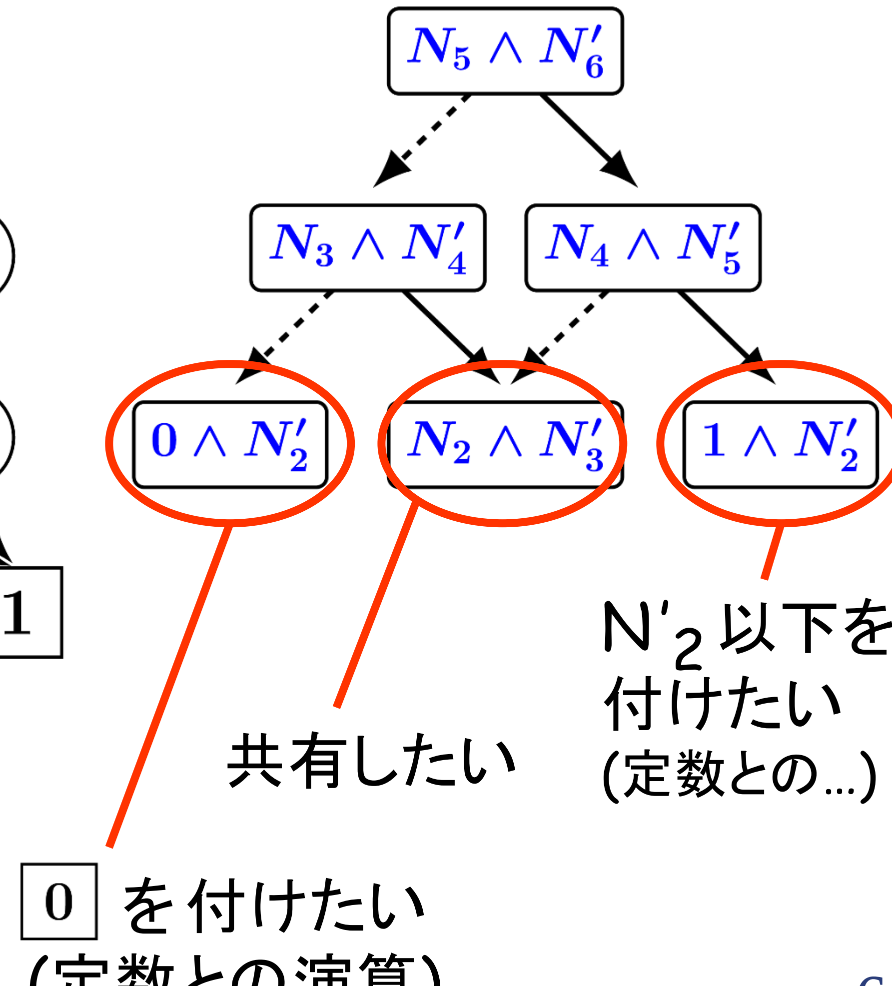
- $f = \overline{x} f_0 \vee x f_1$ と
  $g = \overline{x} g_0 \vee x g_1$ との２項演算 (AND, OR, XOR など)
- $f \bullet g = \overline{x}(f_0 \bullet g_0) \vee x (f_1 \bullet g_1)$



演習問題: 論理演算

- 以下の 2 つの BDD の論理積

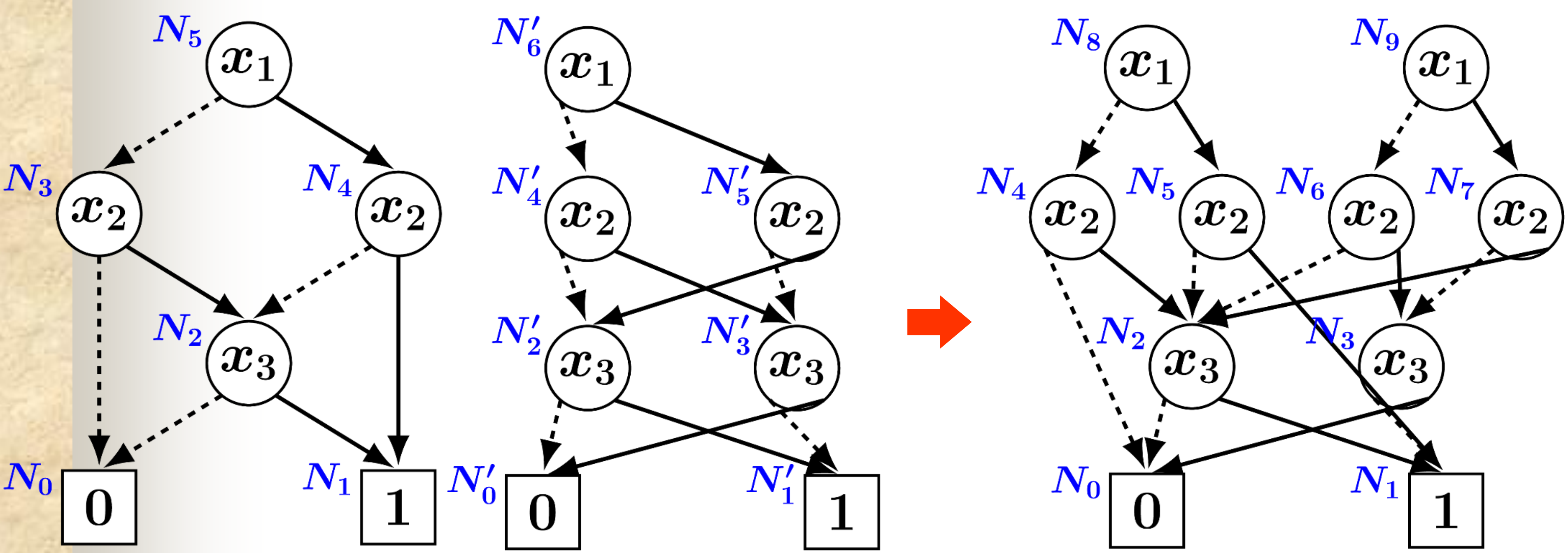




休 憩

- ここで、少し休憩しましょう。
- 深呼吸したり、肩の力を抜いてから、
  次のビデオに進んでください。

Shared BDD



- BDD処理系内で、変数順序をそろえ、等価な節点を共有

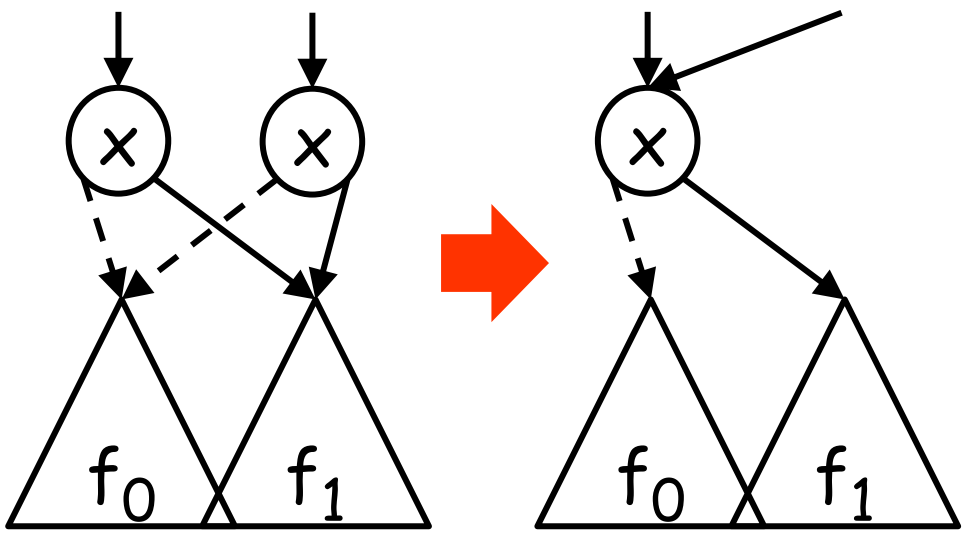
→ 関数の一意性

(処理系内で、同じ関数を表す BDD は 1 つだけ)

処理系内で、一意性を保つ

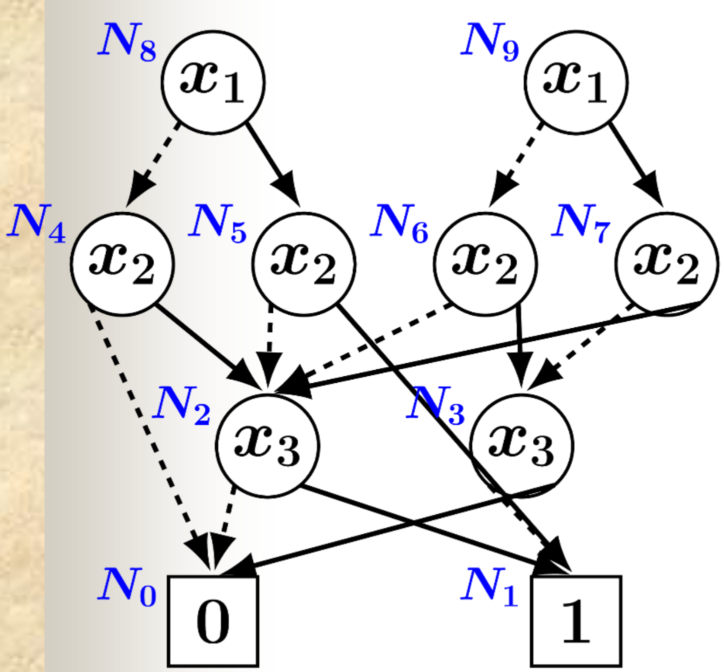
- **等価な節点**を**必ず共有**し、1つにまとめる
(等価な節点が2つ以上あってはいけない)

等価な節点の共有



- 変数番号、0-枝の指す節点番号、1-枝の指す節点番号の3つ組で、節点を管理

処理系内で、一意性を保つ



節点テーブル

	変数	0-枝	1-枝
N_0	-	-	-
N_1	-	-	-
N_2	x_3	N_0	N_1
N_3	x_3	N_1	N_0
N_4	x_2	N_0	N_2
N_5	x_2	N_2	N_1
N_6	x_2	N_2	N_3
N_7	x_2	N_3	N_2
N_8	x_1	N_4	N_5
N_9	x_1	N_6	N_7

- 変数番号、0-枝の指す節点番号、1-枝の指す節点番号の３つ組で、節点を管理

処理系内で、一意性を保つ

- 節点は3つ組で要求する
  - 既に登録されていれば、その節点番号を返す
  - 未登録なら、新しい節点を作る
- 3つ組がキーのハッシュを利用
  - チェックが O(1) 時間

ハッシュの活用が
BDD の高速処理の鍵

節点テーブル

	変数	0-枝	1-枝
N_0	-	-	-
N_1	-	-	-
N_2	x_3	N_0	N_1
N_3	x_3	N_1	N_0
N_4	x_2	N_0	N_2
N_5	x_2	N_2	N_1
N_6	x_2	N_2	N_3
N_7	x_2	N_3	N_2
N_8	x_1	N_4	N_5
N_9	x_1	N_6	N_7

- 変数番号、0-枝の指す節点番号、1-枝の指す節点番号の3つ組で、節点を管理

処理系内で、一意性を保つ

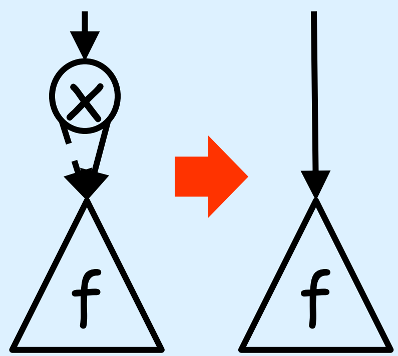
- 節点は3つ組で要求する
  - 既に登録されていれば、その節点番号を返す
  - 未登録なら、新しい節点を作る

節点テーブル

	変数	0-枝	1-枝
N_0	-	-	-
N_1	-	-	-
N_2	x_3	N_0	N_1
N_3	x_3	N_1	N_0
		N_0	N_2
		N_2	N_1
		N_2	N_3
		N_3	N_2
		N_4	N_5
		N_6	N_7

0-枝の先と1-枝の先が同じ場合は、その節点番号を返す

冗長な節点の削除



関数の一意性のため、部分グラフの比較が頂点番号を比較のみで実行できる

- 変数番号、0-枝の指す節点番号、1-枝の指す節点番号の３つ組で、節点を管理

節点要求 GetNode(x, N_{f0}, N_{f1})

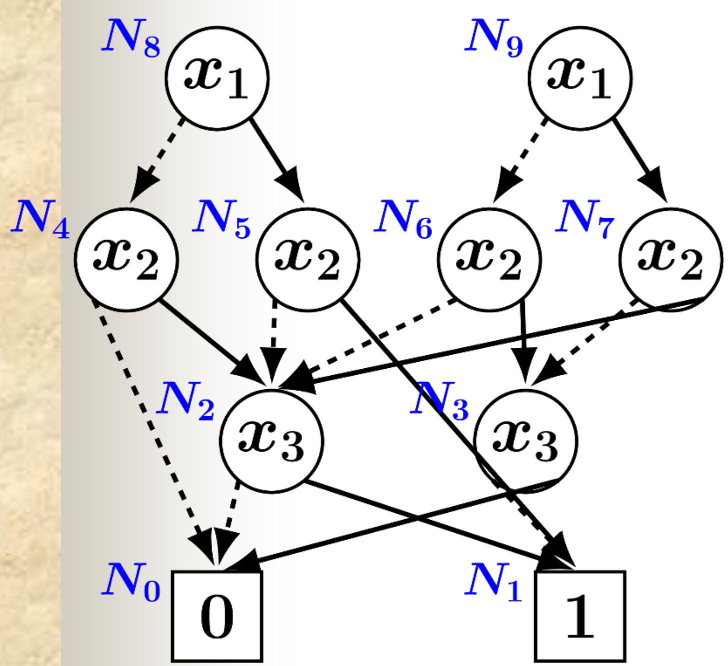
1. $N_{f0} = N_{f1}$ なら、N_{f0} を返す
2. 節点テーブルに (x, N_{f0}, N_{f1}) があれば、
   その節点番号を返す
3. 未登録なら、0-枝の先が N_{f0} で 1-枝の先が N_{f1} となる
   ラベル x の節点を新しく節点テーブルに作って、
   その節点番号を返す

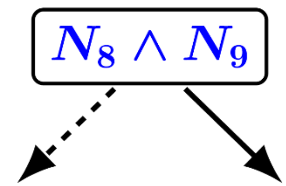
休 憩

- ここで、少し休憩しましょう。
- 深呼吸したり、肩の力を抜いてから、次のビデオに進んでください。

演習問題:　論理演算

- 以下の 2 つの BDD の論理積



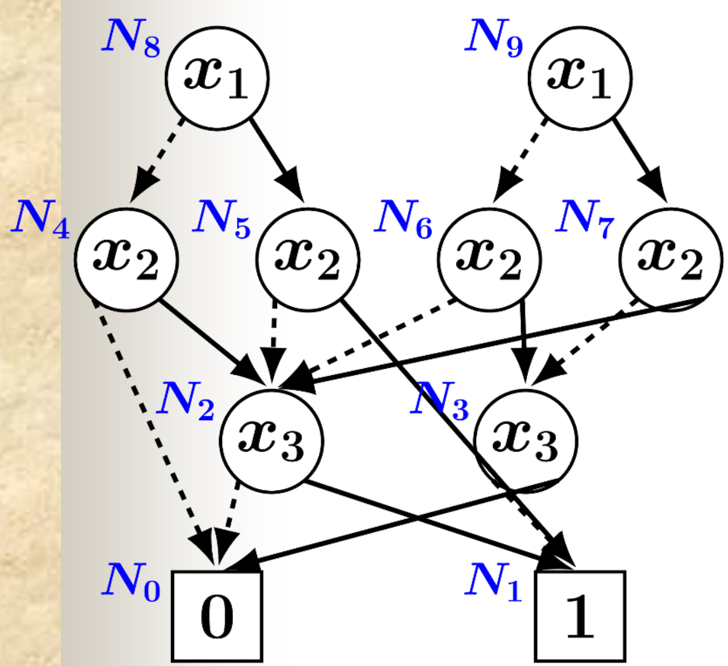


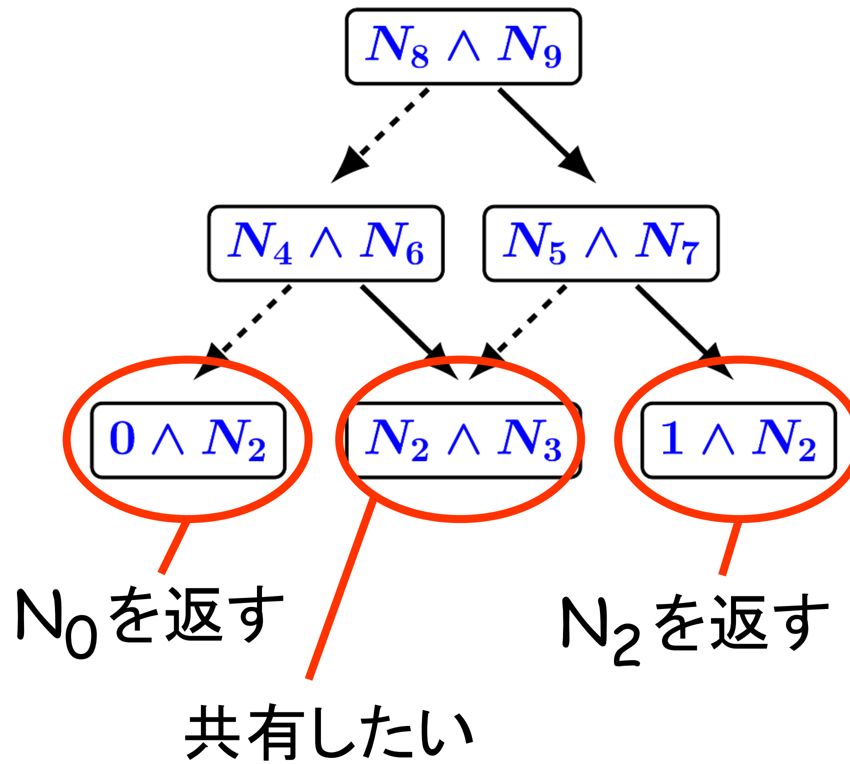
0-枝の指す節点 N_i と
1-枝の指す節点 N_j が
決まってから、
(x_1, N_i, N_j) を要求する

0-枝と 1-枝の子供への再帰的な処理

演習問題: 論理演算

- 以下の 2 つの BDD の論理積

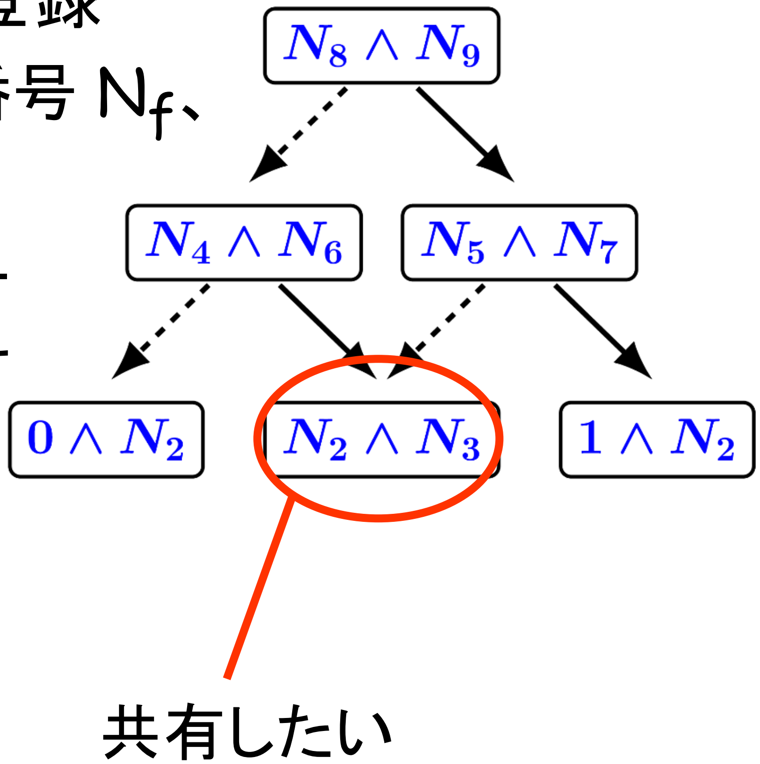




節点を要求するのは、0-枝, 1-枝の先が決まってから
→ $N_2 \wedge N_3$ をそれぞれ計算してから共有することに…

同じ演算を何度も繰り返さない

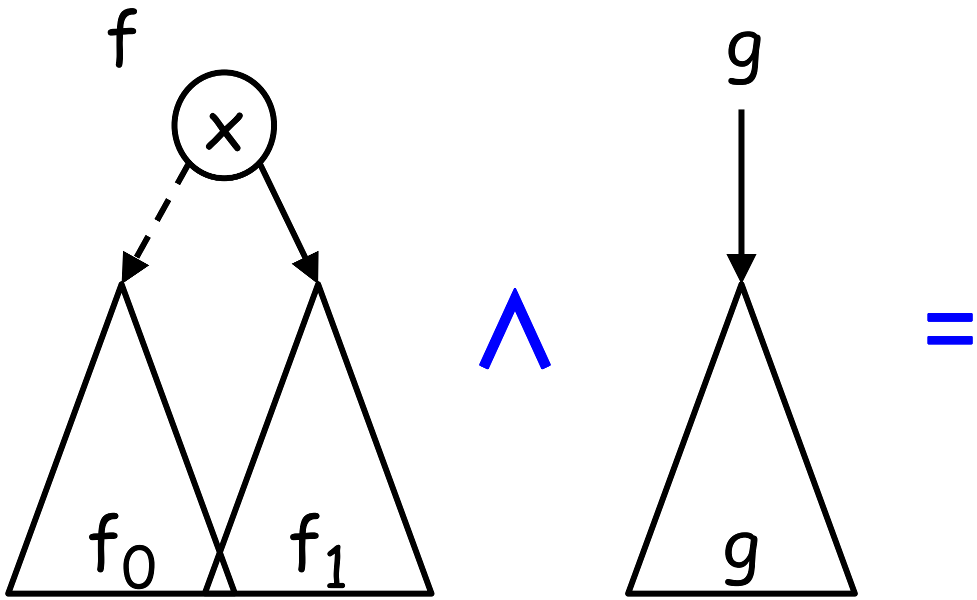
- 過去の演算の結果を、
演算キャッシュ (ハッシュ) に登録
  - 演算の種類 op、f の節点番号 N_f、g の節点番号 N_g の3つ組 (op, N_f, N_g) がキー
  - 演算結果の節点番号を返す



最上位の変数のラベルが異なる場合

もともとは、$f \wedge g = \overline{x}(f_0 \wedge g_0) \vee x(f_1 \wedge g_1)$

g が x に依存しないなら？

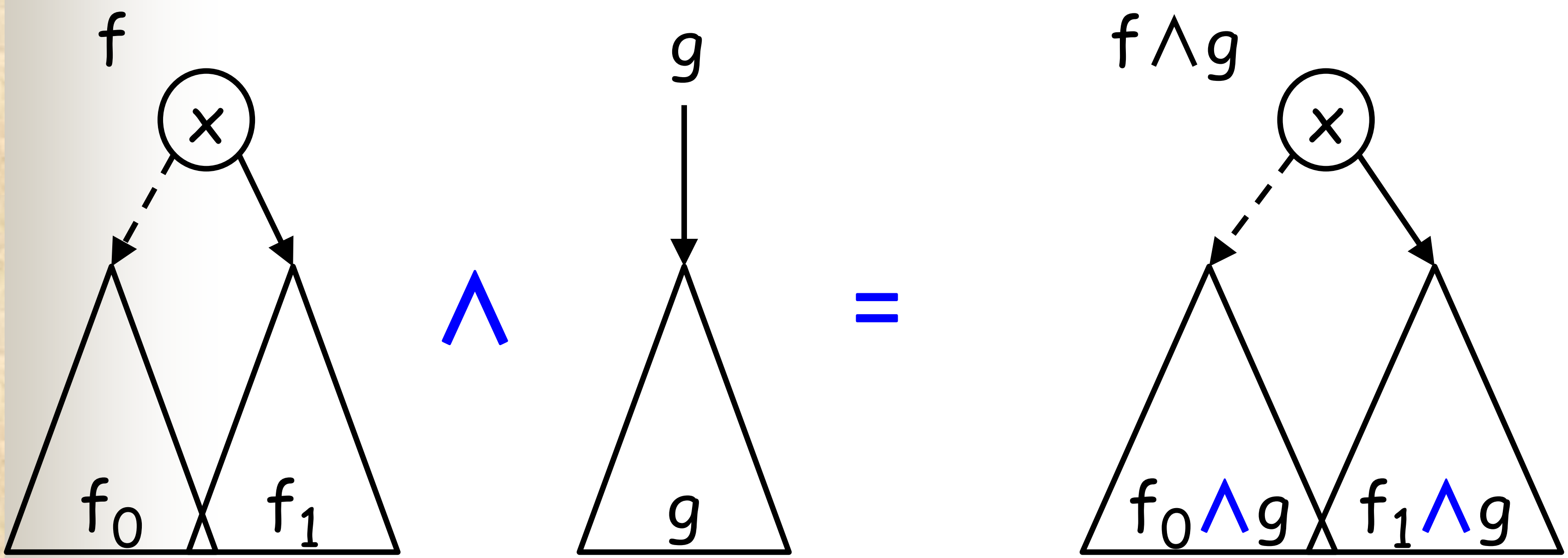


最上位の変数のラベルが異なる場合

もともとは、$f \wedge g = \overline{x}(f_0 \wedge g_0) \vee x(f_1 \wedge g_1)$

g が x に依存しないなら？

$$f \wedge g = \overline{x}(f_0 \wedge g) \vee x(f_1 \wedge g)$$



2項演算 Apply(op, N_f, N_g)

N_f: (x_f, N_{f0}, N_{f1})

N_g: (x_g, N_{g0}, N_{g1})

1. N_f, N_g の少なくとも一方が定数節点 or $N_f = N_g$ なら op に応じた節点番号を返す
2. 演算キャッシュに (op, N_f, N_g) があれば、その節点番号を返す
3. 変数 x_f と x_g が同じなら
   - N_{h0} := Apply(op, N_{f0}, N_{g0}), N_{h1} := Apply(op, N_{f1}, N_{g1})
   - $N_{h0} = N_{h1}$ なら N_{h0} を返す
   - そうでないなら GetNode(x_f, N_{h0}, N_{h1}) の結果を返す
4. 変数 x_f が 変数 x_g よりも上位なら
   - N_{h0} := Apply(op, N_{f0}, N_g), N_{h1} := Apply(op, N_{f1}, N_g)
   - 以降は 3 と同様
5. 変数 x_f が 変数 x_g よりも下位なら
   - 4 と同様 (N_f, N_g の役割を交換する)

Apply 演算の計算量

- 最悪の場合の計算時間は、O(|f| |g|)
  出力の BDD のサイズが O(|f| |g|) になりえるため

- 長らく、出力の BDD のサイズが小さければ、
  O(|f| |g|) 時間より速く計算できると思われていた
- 入力や出力の BDD のサイズが小さくても、
  O(|f| |g|) 時間かかる例が見つかった
  [Yoshinaka et al. 2012]

- 経験的には、f, g の BDD のサイズ |f|, |g| に
  比例する時間 O(|f| + |g|) で
  Apply 演算ができることが多い

ハッシュの活用

おまけ: 参照カウンタ

- 節点テーブルでは、各節点が他の節点から参照されている回数 (指されている回数; 入次数) を管理することが多い
- なぜ？
  - GetNode を繰り返すと、節点テーブルがあふれる
  - 参照されている回数が 0 の節点を回収して再利用
- 考慮すべき点
  - 節点を回収しただけでは、演算キャッシュが問題に
    → 演算キャッシュをクリアする
  - 参照カウンタが 0 になるたびに回収すると、演算キャッシュの効率が悪い
    → まとめて回収する

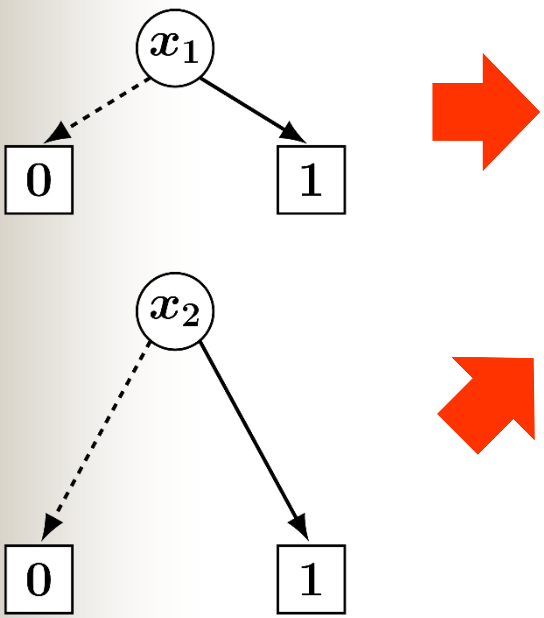
演習問題: BDD の作成

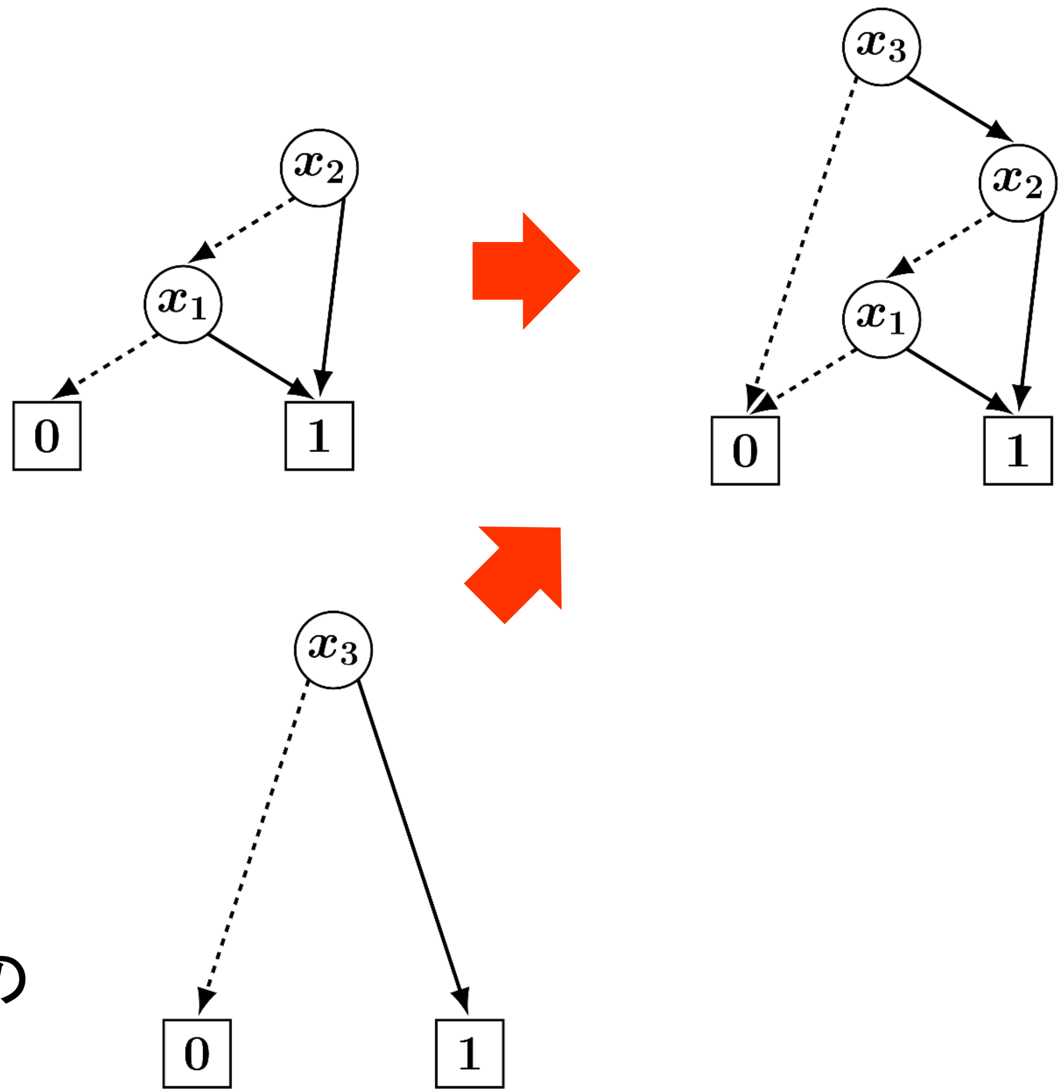
- 以下の論理関数を、BDD で表しなさい
  1. 論理積 (AND): $x1\ x2\ x3\ x4$
  2. 論理和 (OR): $x1 \vee x2 \vee x3 \vee x4$
  3. AND, OR の組合せ: $(x1 \vee x2)\ x3$
  4. 排他的論理和 (XOR): $x1 \oplus x2 \oplus x3 \oplus x4$

- 補足
  - 真理値表 → 決定木 → BDD の方法
  - 意味を考えて、上から BDD を作る方法
  - Apply 演算を繰り返して BDD を作る方法

Apply演算を繰り返して BDD を作る

- (x1 ∨ x2) x3





実際には、Shared BDD の
形で処理される

まとめ

- 二分決定グラフ (BDD)
- 2つの BDD の論理演算

高速化の仕組み

- Shared BDD: 処理系内で、関数を一意に表す
- 2つのハッシュを利用して、演算を高速化
  - 節点テーブル: 等価な節点を何個も作らない
  - 演算結果テーブル: 同じ演算を何回も実行しない