アルゴリズム特論（第4回）

北海道大学 大学院 情報科学研究科
アルゴリズム研究室 湊 真一

前回の内容

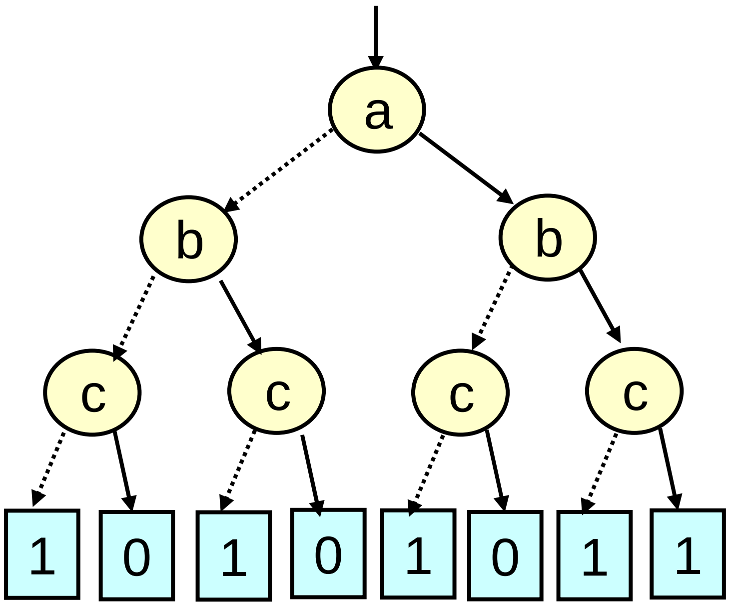
積和形論理式の最小化・簡単化

- 簡単化・最小化・非冗長化
- カルノー図と積和形論理式との対応
  - 最小項、内項、素項、必須素項
- 最小化（Q-M法）
  - アルゴリズム概要と計算量
- 実用的簡単化
  - MINI, Espresso
  - ベンチマーク集合
- 非冗長化（Morreale法）
  - アルゴリズム概要
  - BDDとの組合せ

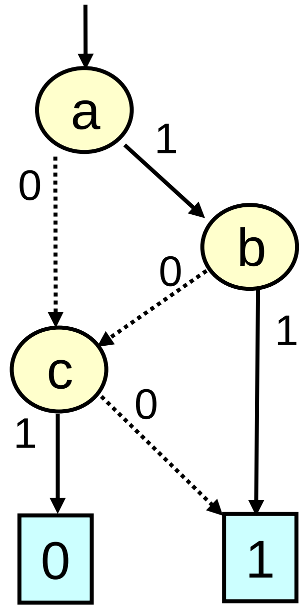
今回の内容

- **二分決定グラフ(BDD)**
  - 基本データ構造
    - シャノンの展開、場合分け2分木とBDD、簡約化規則
  - BDDの特徴
    - 真理値表や積和形との比較、一意性、高速演算、コンパクト性
    - 種々の論理関数のBDD
    - 変数の順序づけの影響
  - BDDの生成アルゴリズム
    - 論理式からのBDD生成手順
    - 二項論理演算アルゴリズム
    - 充足解の探索、最適充足解の探索
    - 真理値表密度(＝充足確率)の計算
  - BDDの改良技術
    - 複数のBDDの共有化
    - 否定枝

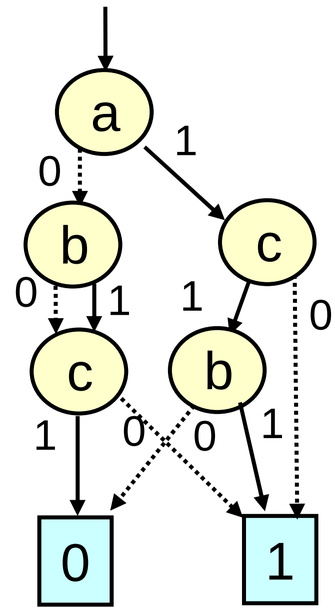
BDD（Binary Decision Diagram)
（二分決定グラフ）とは



真理値表と等価なBDD
(Binary Decision Tree)



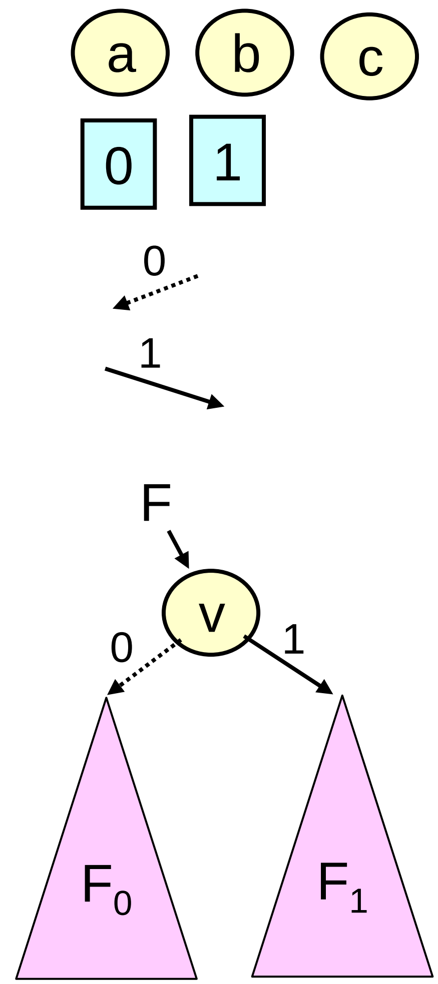
既約な順序付きBDD
(Reduced Ordered BDD)



既約でも順序付き
でもないBDD
(Unordered BDD)

BDDの節点(node)と枝(edge)

- 変数節点(variable node)
- 定数節点(constant node)
- 0-枝(0-edge)
- 1-枝(1-edge)
- 部分グラフ(sub-graph)



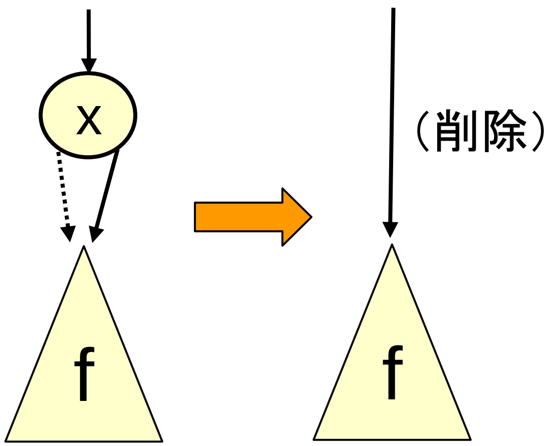
シャノンの展開(Shannon’s expansion)

$$F(v, X) = \sim v\, F(0, X) + v\, F(1, X)$$

ROBDD(Reduced Ordered BDD)

- 同じ論理を表すBDDは複数存在
- 重要な性質を持つのは既約な順序付きBDD(ROBDD)
  - 以後、特に断らない限り、ROBDDのことを単にBDDと呼ぶ。
- 順序付きBDD：
  - 変数に全順序関係が定義されている
  - 根(root)から定数節点に至るすべてのパスについて変数の出現順序が、全順序関係に矛盾しない
- 既約なBDD
  - BDDの２つの簡約化規則がこれ以上適用できなくなるまで適用されている形

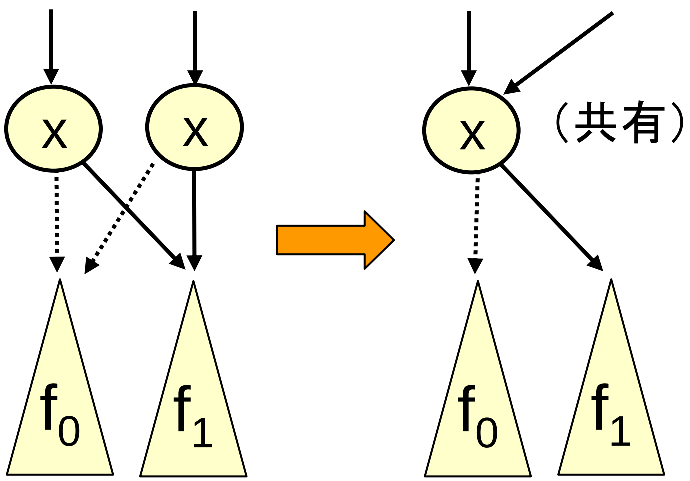
BDDの簡約化規則



(a) 冗長な節点を全て削除

(b) 等価な節点を全て共有

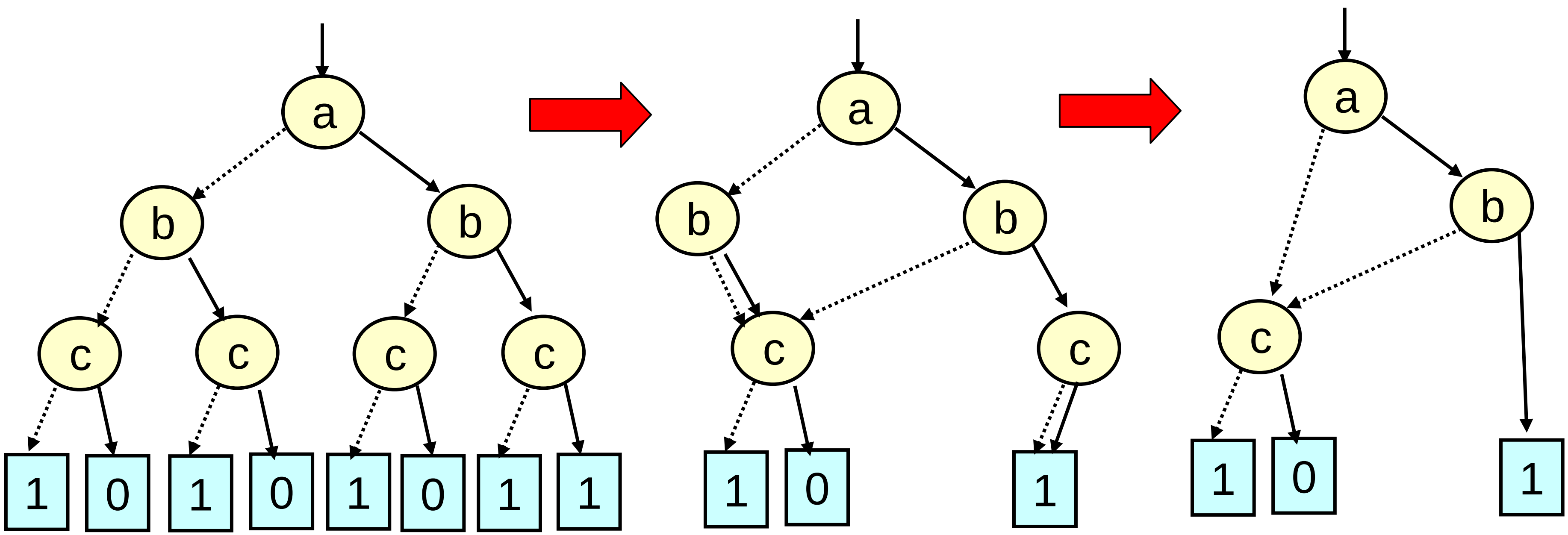
既約なBDDが得られる



参考:
(b)の規則だけを可能な限り適用した形を
「準既約」(Quasi-reduced)なBDD
と呼ぶこともある。

BDDの簡約化の例

- どの部分から簡約化を行っても最終形は同じ
- 論理関数に対する一意な表現(標準形)となる
  – ただし変数順序が異なると同じ論理でも違う形になる
  (偶然一致する場合もあるが)



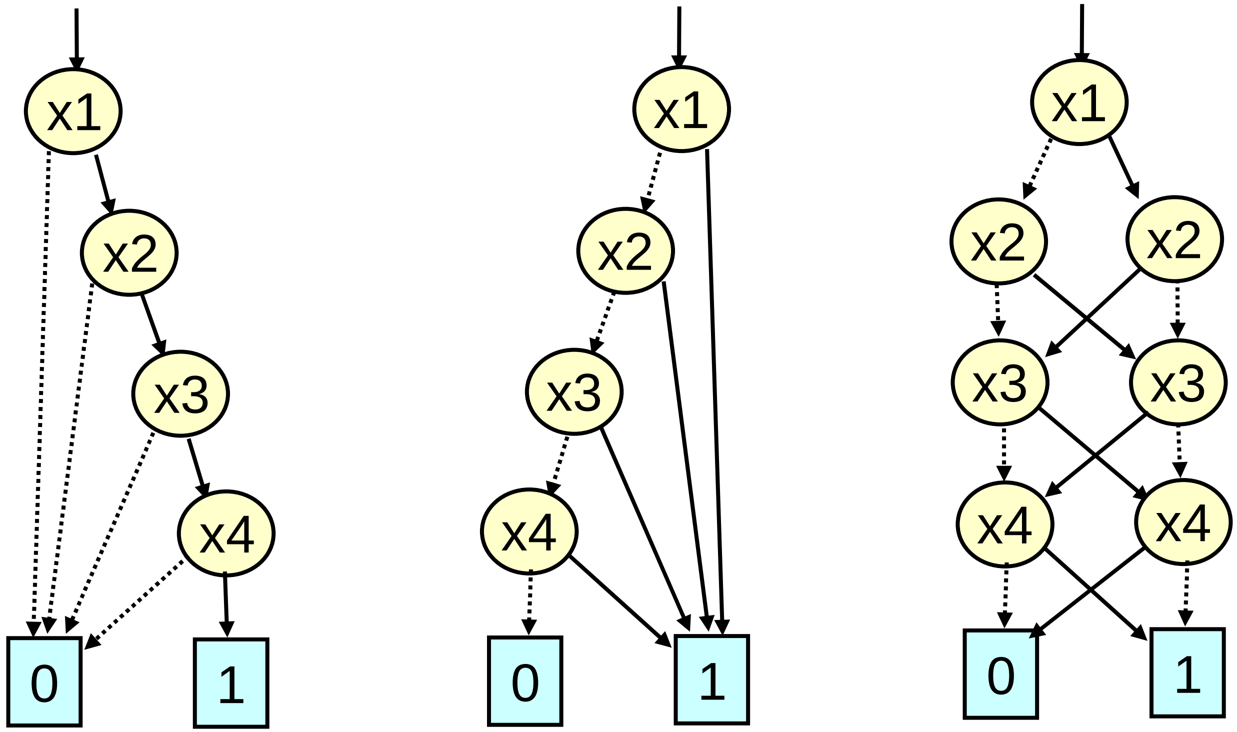
BDDの特徴

- 論理関数に対してグラフの形が一意に定まる。
  – 等価性判定が非常に容易
- 多くの実用的な論理関数がコンパクトに表現できる。
  – パリティ関数や加減算回路も効率よく表現
  – 性質の良い関数では数百入力まで扱える
- 論理関数同士の演算が、グラフのサイズにほぼ比例する計算時間で実行できる。
  – 否定演算も容易
- グラフのサイズが小さくならない場合もある。
  – 乗算回路のBDDは指数サイズ
- 変数の順序づけが悪いとグラフが大きくなる。
  – 比較的良い順序づけを得る方法がいくつか実用化（厳密最小化はNP完全問題）

BDDの節点数

- n入力の論理関数は2^{2^n}種類存在する。
  これを識別するには少なくとも2^nビットの記憶量が必要。
- BDDでも例外ではない。n変数論理関数を表現するための最悪の場合の節点数は$O(2^n/n)$となる。
  - 1個の節点を記憶するのに$O(n)$ビット必要なので、
    全体としては$O(2^n)$ビットとなる。
- ただし、実用上よく用いられる多くの論理関数について、nの多項式に比例する節点数になることが示されている。

n変数の論理積･論理和･パリティ

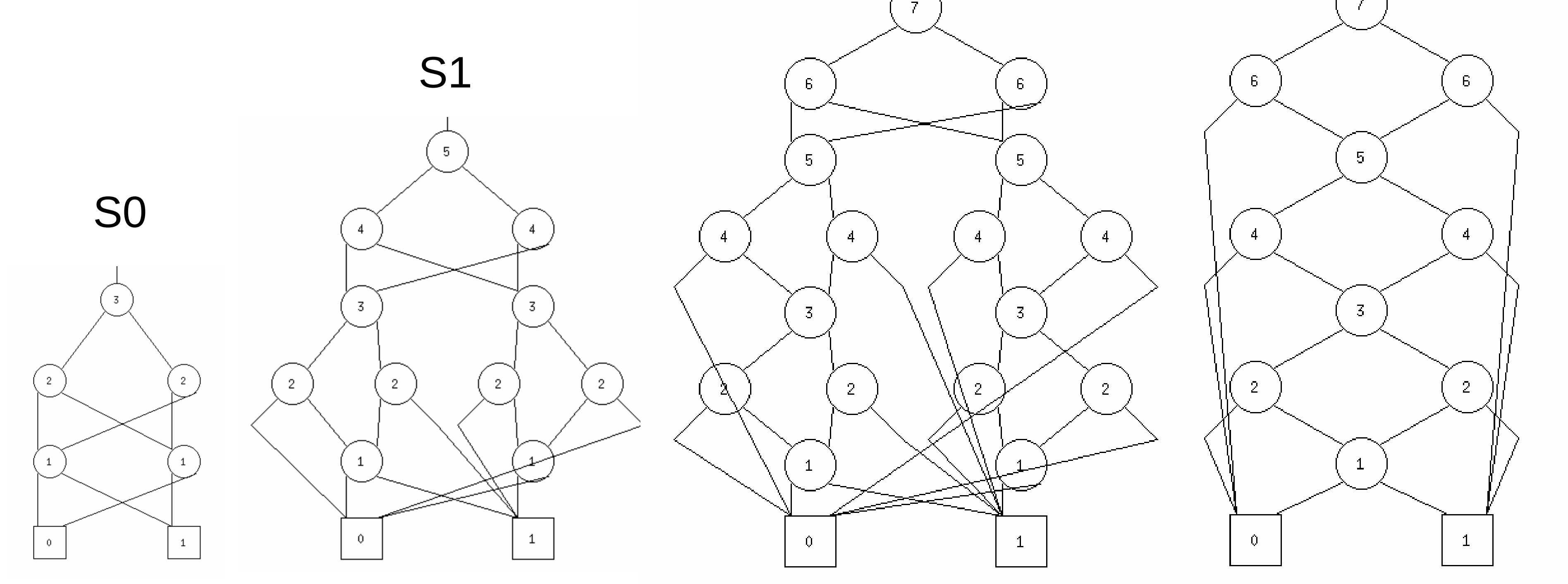


論理積(AND)　　論理和(OR)　　排他的論理和･パリティ(EXOR)

- いずれも､nに比例する節点数 O(n) で表現可能
- 0と1の定数節点を入れ替えるとそれぞれの否定論理になる

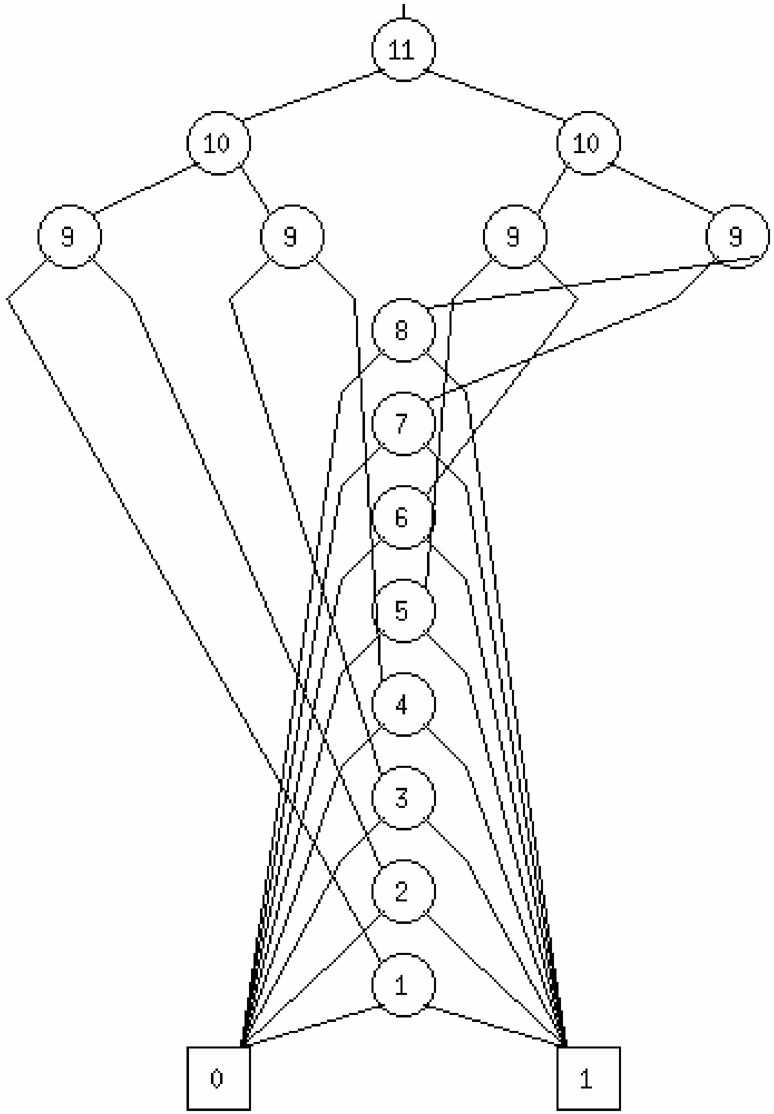
nビット2進数の算術加算

3ビット加算器の論理関数のBDD
（変数は a2, b2, a1, b1, a0, b0, c0 の順）



- nが増えてもBDDは縦方向に伸びるだけで幅は一定
  → 節点数は O(n)
- 減算も同じく O(n) となる。2進数の大小比較も O(n)

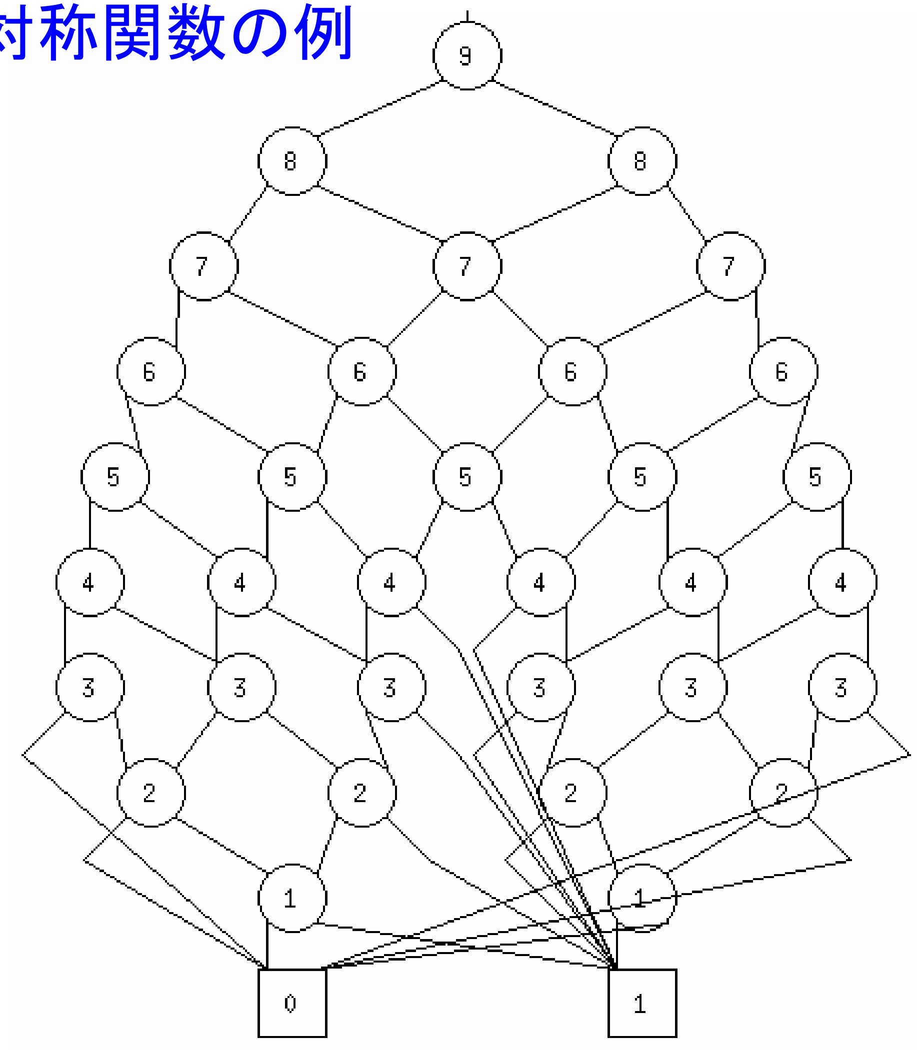
n入力データセレクタ



- n個のデータ入力から1つを選んで出力する論理関数
- (log n)個の制御入力で、どの番号のデータを選ぶかを指定する。
- BDDの節点数は O(n) で表現可能

対称(symmetric)論理関数

9入力変数の
対称関数の例

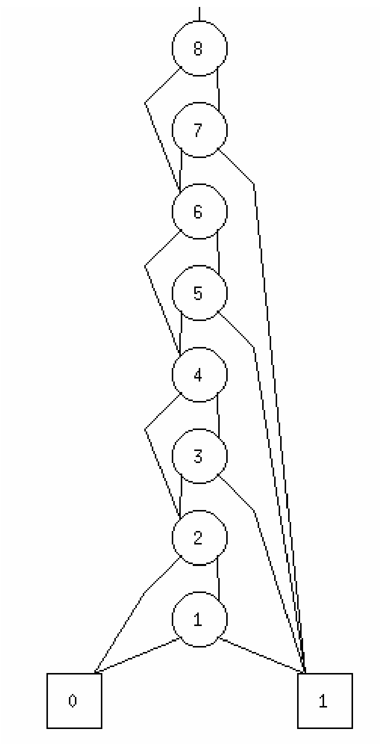


- 対称関数：変数の順序を入れ替えても論理に影響がない論理関数
  - n個の入力変数のうち、"1"になっている変数の個数によって出力の値が決まる
- BDDの場合、各段ごとに、最上位からその変数までの"1"の個数を示している。
  - BDDの幅は最大でn
  - 全体の節点数は$O(n^2)$

変数の順序付けの影響（例1）

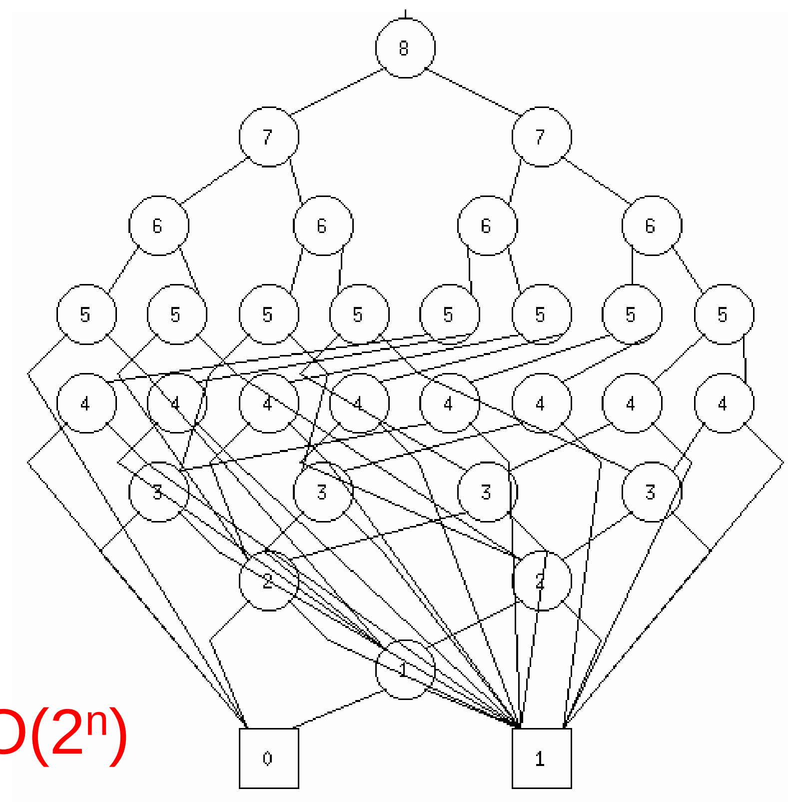
- BDDは変数の順序づけにより節点数が著しく変化する場合がある。

x1 x2 + x3 x4 + x5 x6 + x7 x8



O(n)

x1 x5 + x2 x6 + x3 x7 + x4 x8

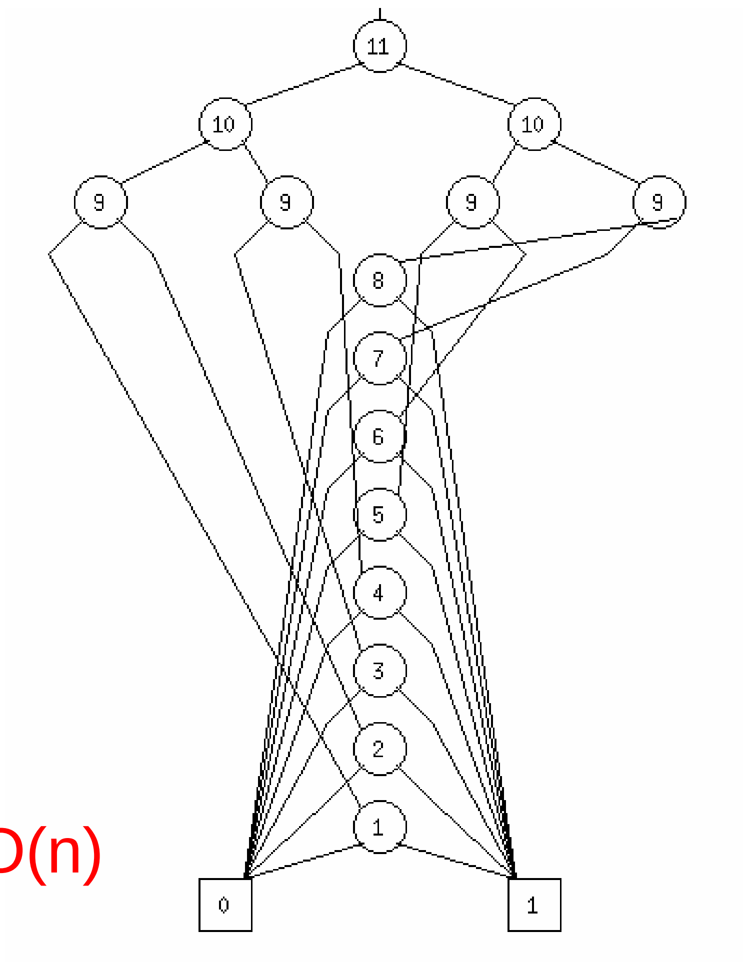


$O(2^n)$

変数の順序付けの影響（例2）

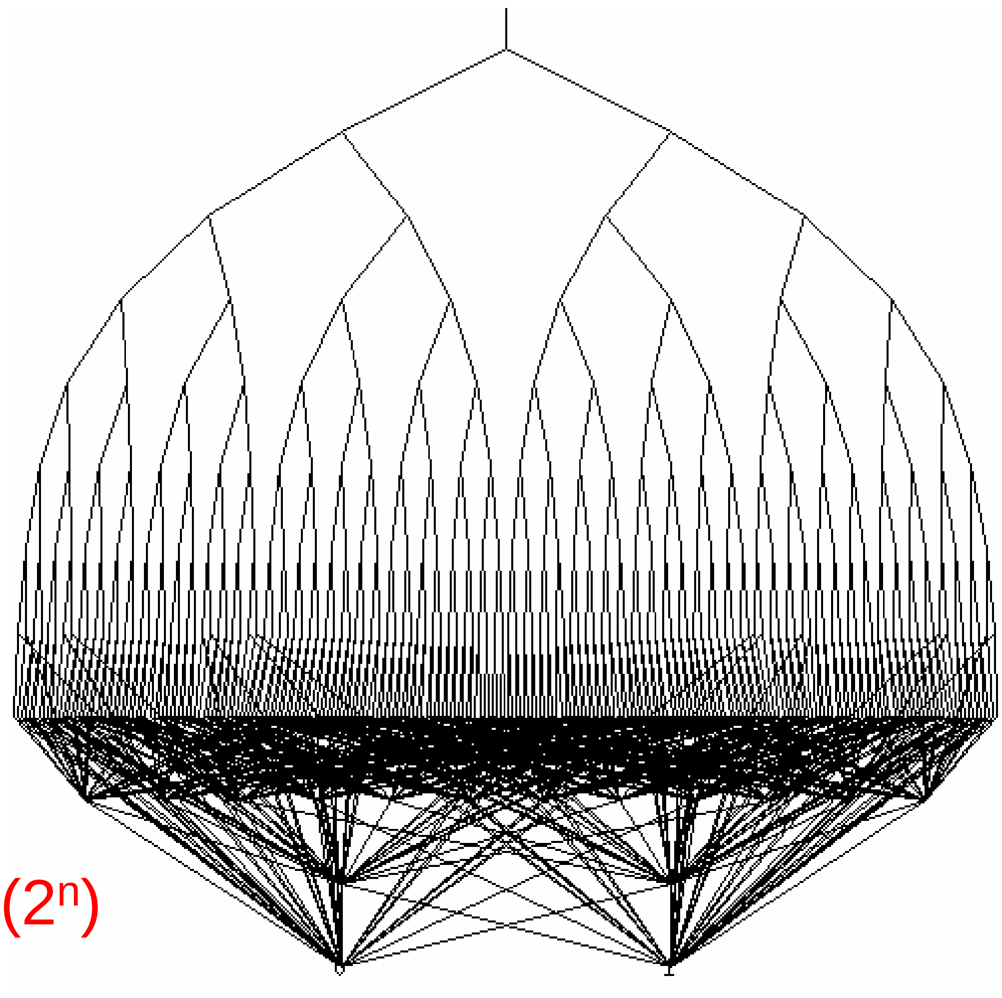
- 8入力データセレクタ

制御入力を上位にした場合



O(n)

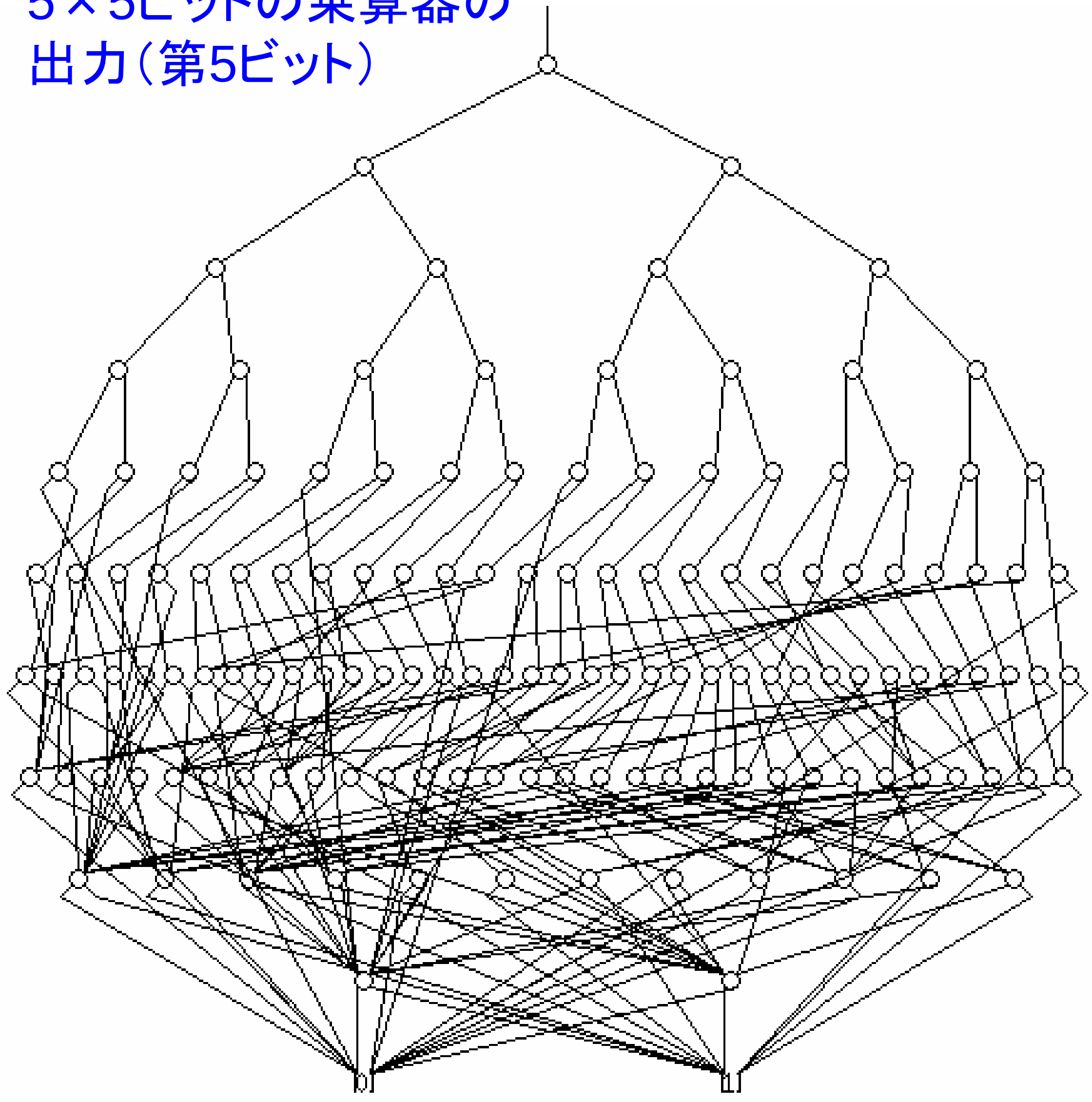
データを上位にした場合



O(2^n)

変数の順序付けの影響（例3）

5×5ビットの乗算器の
出力（第5ビット）

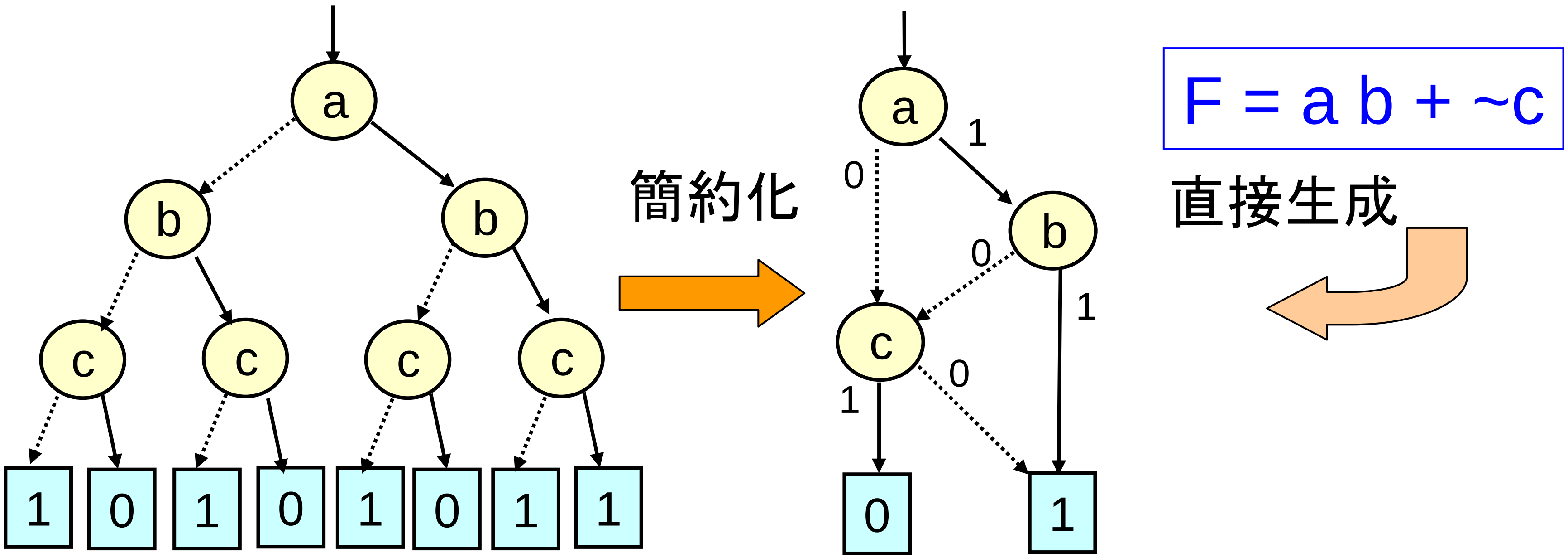


- 2進数の算術乗算は苦手
  - どんな変数順序でも、$O(2^n)$の節点数になることが数学的に証明されている。
  - 除算も同様に常に指数オーダとなる

変数順序の影響パタン：

(a) どんな順序でも常に簡単
(b) 順序付けにより簡単だったり
　複雑だったりする
(c) どんな順序でも常に複雑

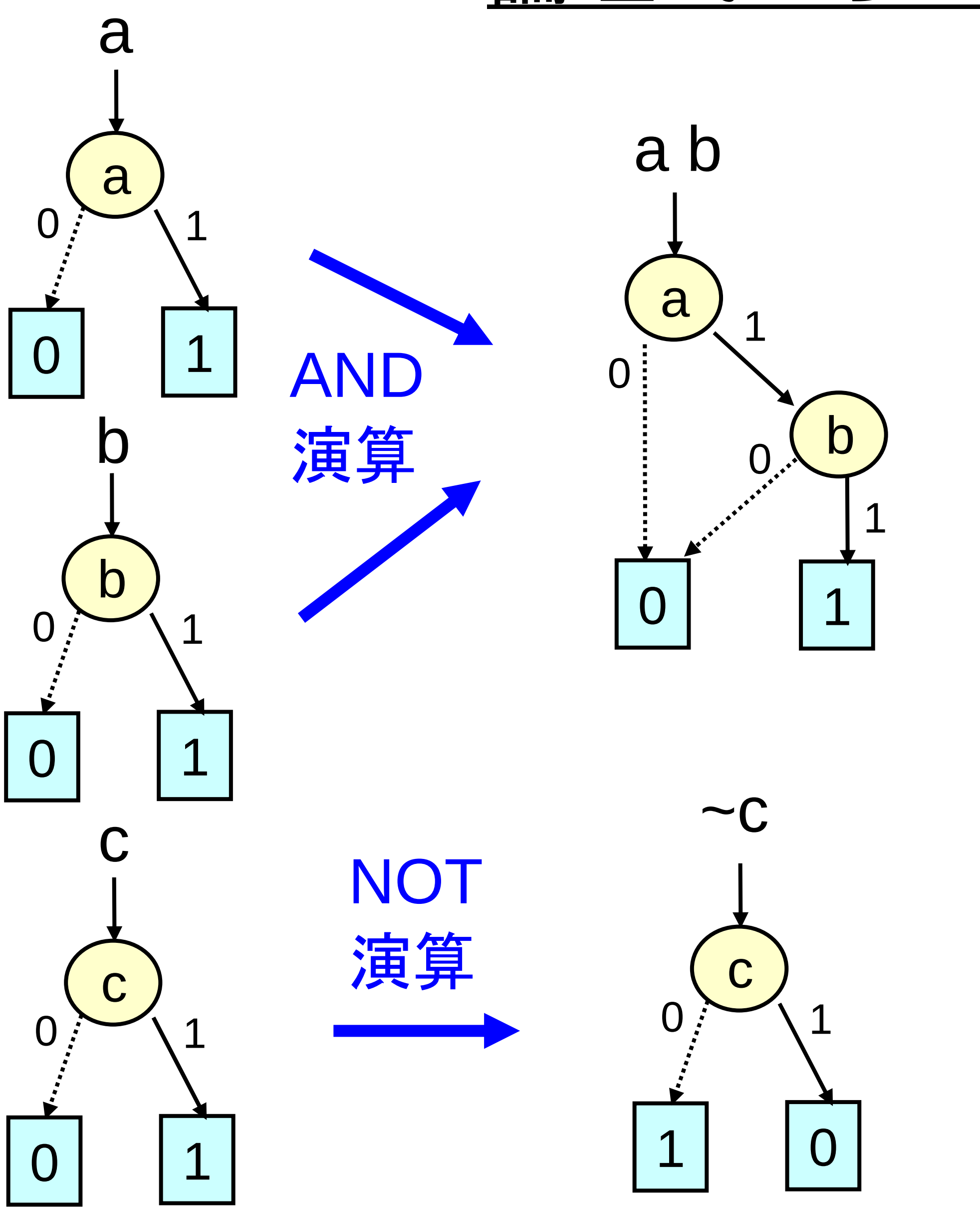
BDDの生成アルゴリズム



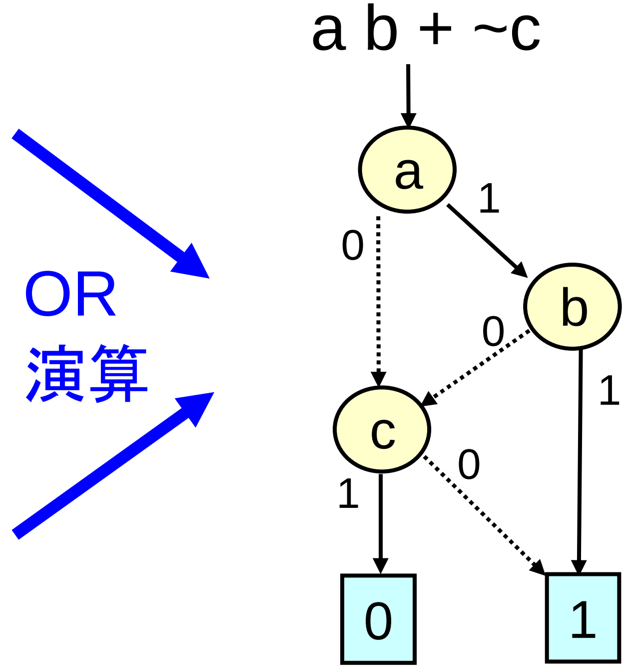
- 真理値表に対応する二分木を簡約化する方法では、常に指数オーダの記憶量と処理時間がかかってしまう。

→　実用的には、論理式からBDDを直接生成するアルゴリズム[Bryant86]を用いる

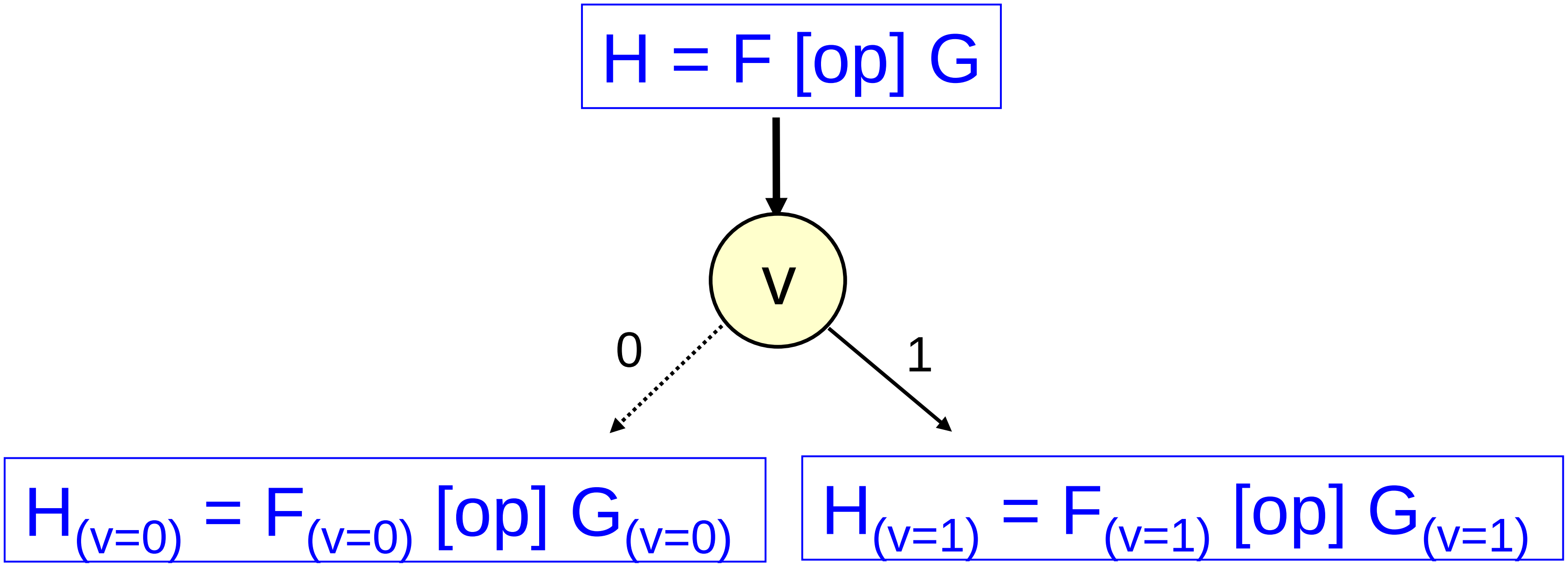
論理式からのBDD生成



- 2つのBDDの間の二項論理演算を繰り返して任意のBDDを生成



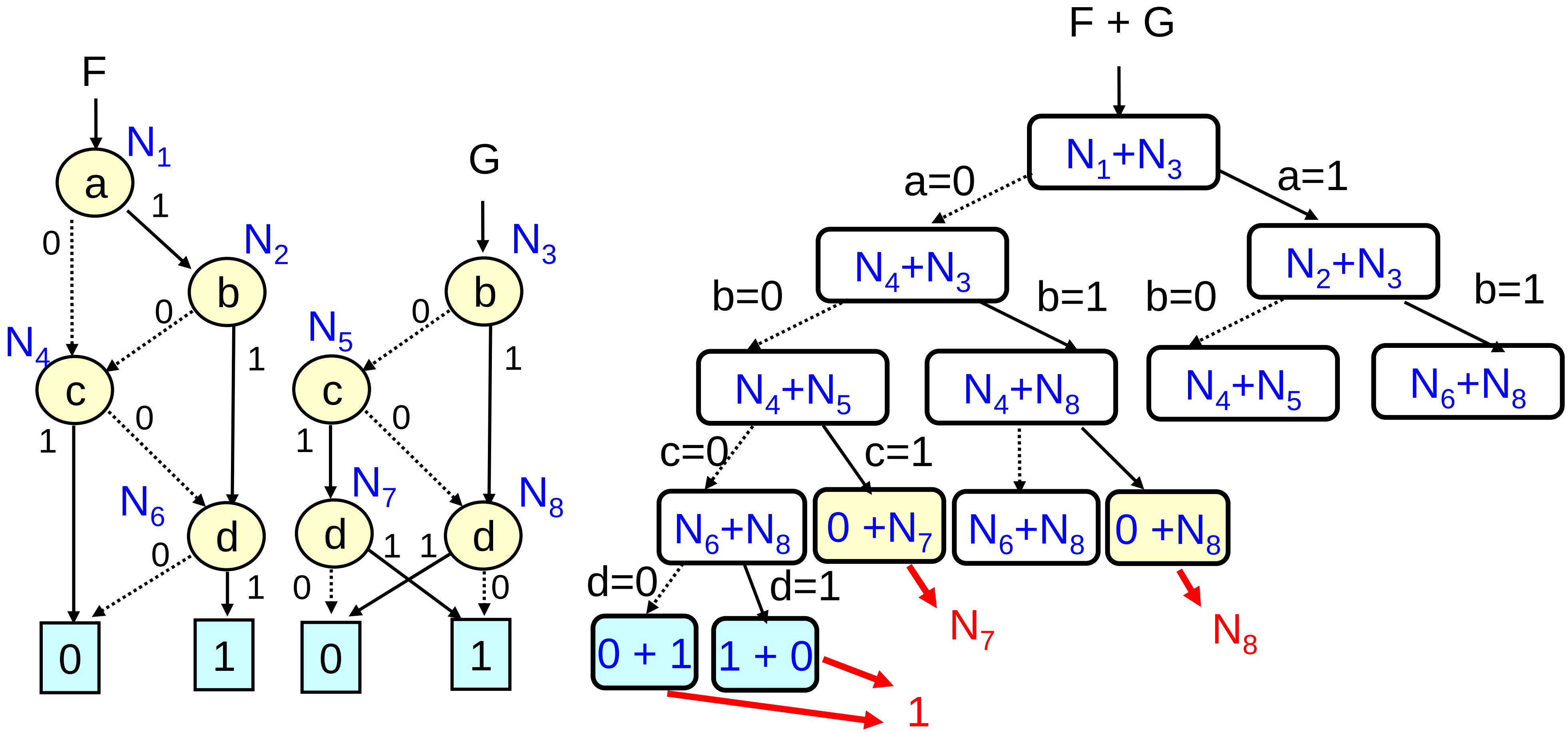
二項論理演算アルゴリズム



- ある変数vに0,1を代入して再帰的に展開
- 全ての変数を展開すると自明な演算になる

  (OR演算の場合) F+1=1, F+0=F, F+F=F …
- 各演算結果をBDDに組み上げる

二項論理演算の実行例

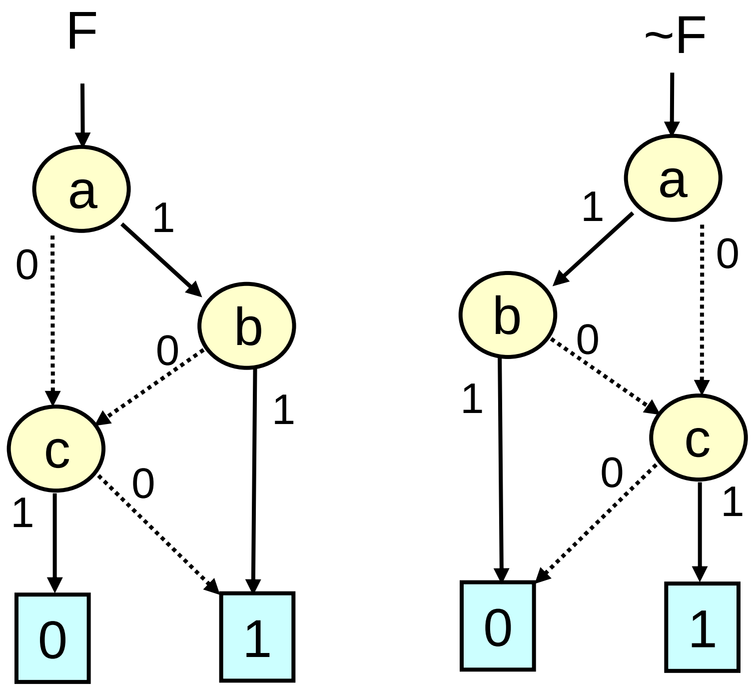


F
N1
a
1
0
N2
b
0
1
N4
c
0
1
N6
d
0
1
0
1
G
N3
b
0
1
N5
c
1
0
N7
d
N8
d
1
1
0
0
0
1
F + G
N1+N3
a=0
a=1
N4+N3
N2+N3
b=0
b=1
b=0
b=1
N4+N5
N4+N8
N4+N5
N6+N8
c=0
c=1
N6+N8
0 +N7
N6+N8
0 +N8
d=0
d=1
0 + 1
1 + 0
N7
N8
1

二項論理演算の計算時間

- グラフ F の節点数を |F| と表すと、H = F [op] G の節点数は、最悪の場合、|H| = |F| × |G| となる。
- 論理演算に要する計算時間は何も工夫しなければ、入力変数をnとすると$O(2^n)$となる。
  - しかし、ハッシュテーブルなどの効率化技法を使うことにより、O(|F| + |G| + |H|) で抑えられる。
    - |F|, |G| が大きくても |H| が小さくなる場合がある。その逆もある。
  - グラフの節点数が小さくなるような場合であれば、nが大きくても劇的に高速に論理演算を行うことができる。
  - グラフのサイズが指数的に大きくなる場合（乗算器の例など）では、BDDの効果はほとんどない。

否定演算アルゴリズム



- BDDをコピーして、0と1の定数節点を交換するだけ
- グラフの節点数に比例する時間で計算可能
- 後で述べる「否定枝」の技法を用いると定数時間で計算可能

充足解探索・最適解探索

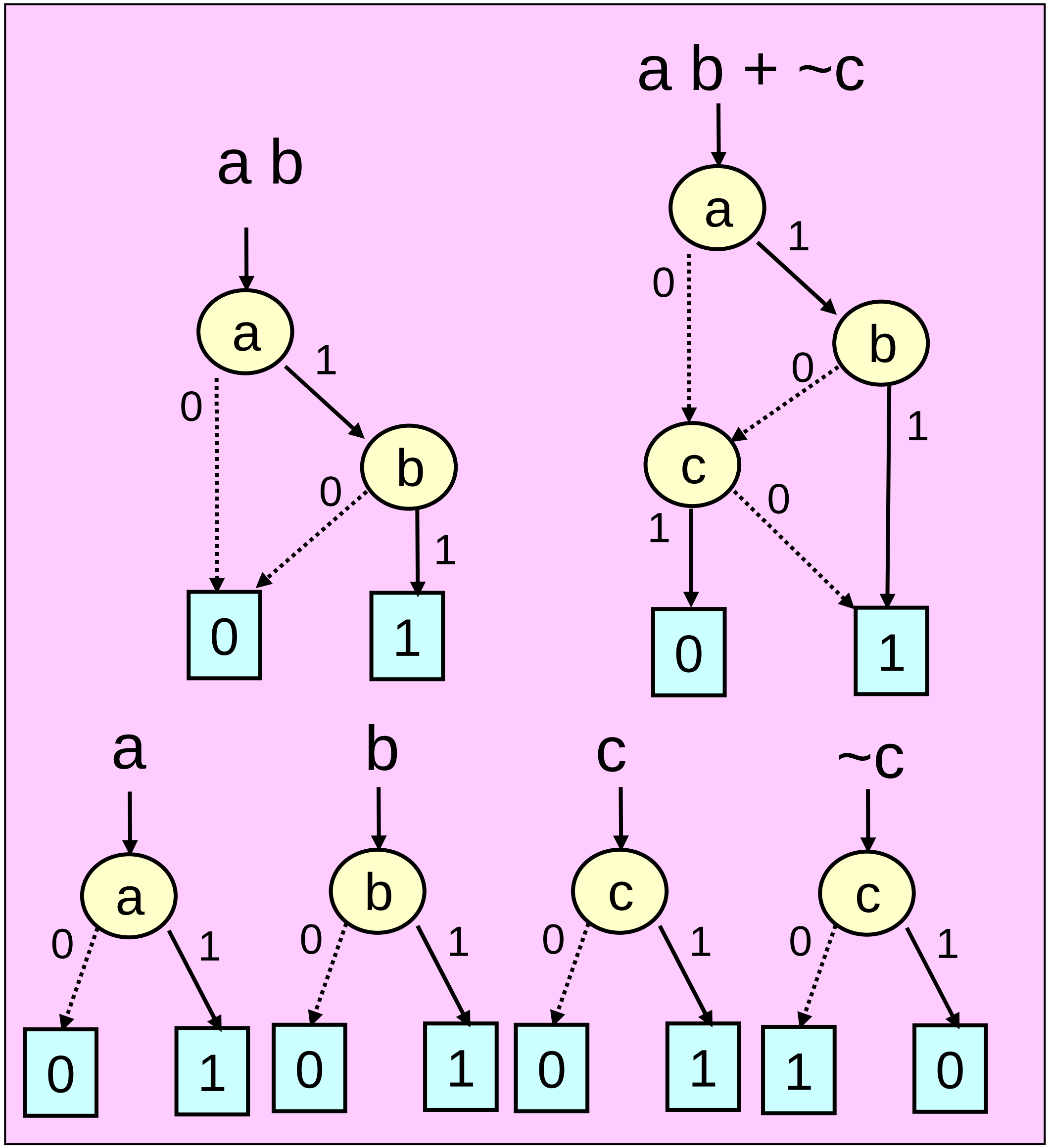
- 既約なBDDが与えられたとき、
その論理関数が充足可能かどうか（恒偽関数でないかどうか）の判定はただちに可能
- 恒偽関数でない場合に、充足解（論理を1にするための各変数の0,1の割当）を求めることも容易
  - 0定数節点を指さない枝をたどっていけば、必ず1定数節点に到達できる
  - グラフの節点数ではなく変数の数に比例
- 入力変数に1を代入するときのコストが与えられているとき、コスト最小の充足解（最適解）を求められる
  - グラフの節点数に比例する時間
- 論理関数が1になる割合（真理値表密度）や、1になる確率も求められる
  - グラフの節点数に比例する時間

BDDの改良技術

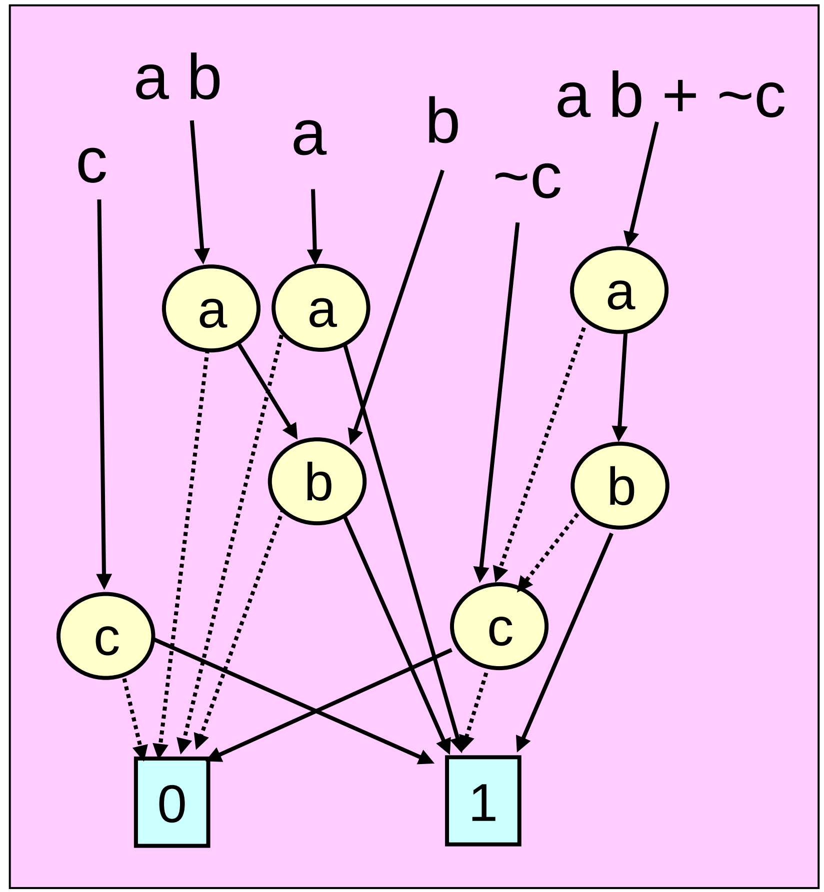
- BDDの演算処理を効率化するための様々な改良技術が研究開発されている
- 実用上、重要な技法として以下の2つがある
  1. 複数のBDDを統一的に共有して扱う技法
  2. 否定枝の技法

この2つの技法は、現在のほとんどのBDD処理系で用いられている

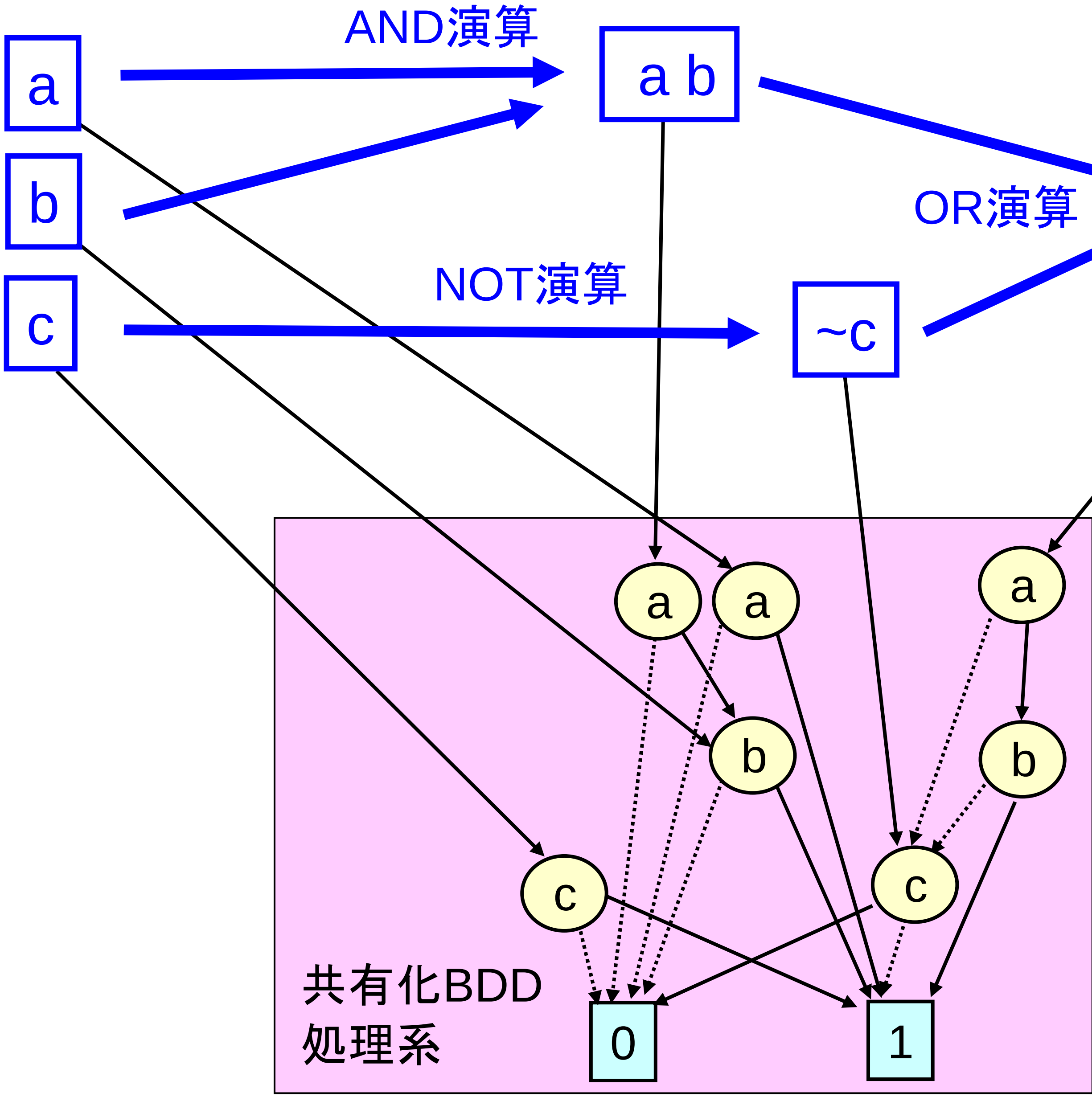
複数のBDDの共有化(Shared BDD)



- 変数の順序を揃える
- 全てのBDDを1つのグラフに共有化

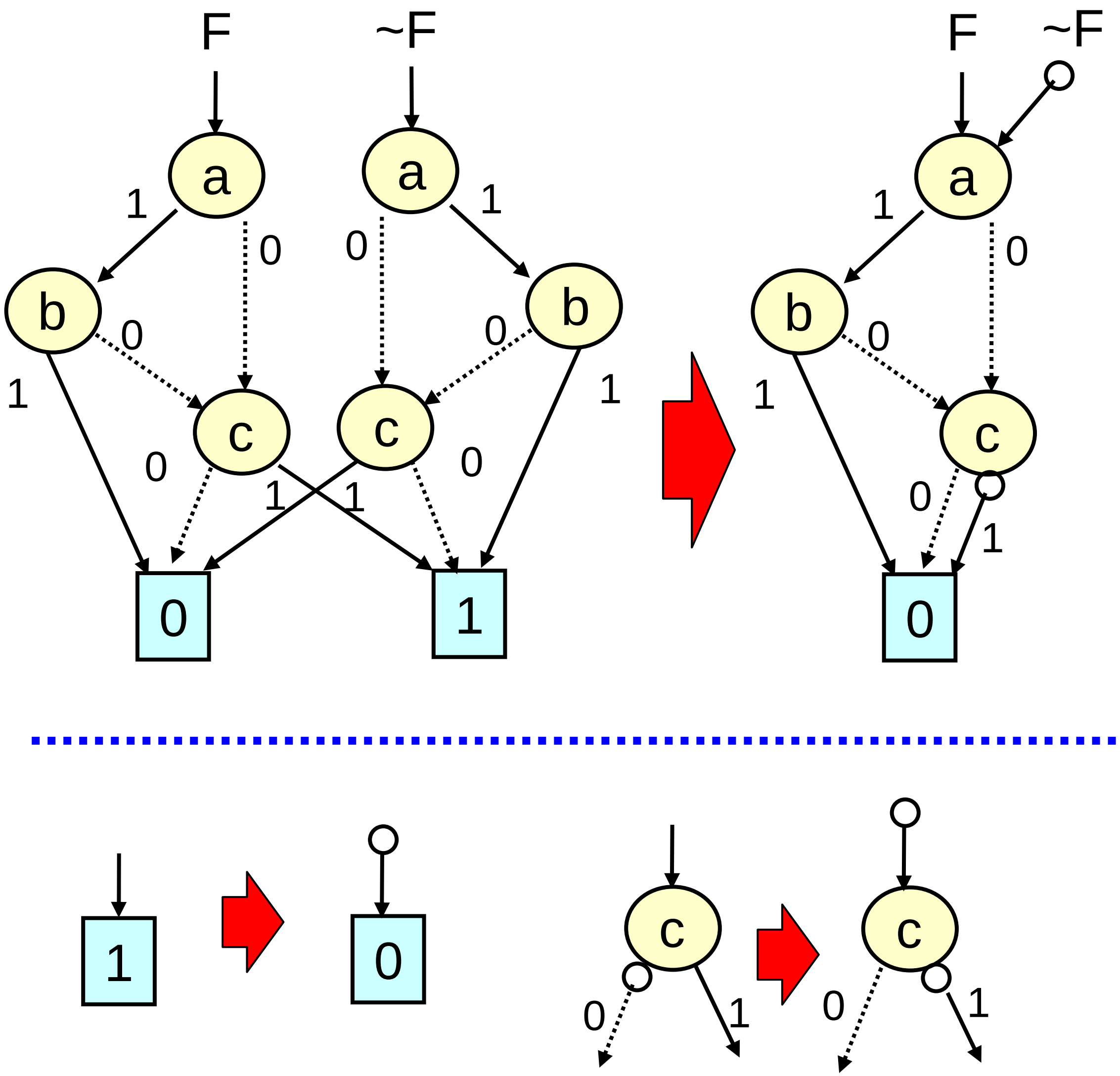


共有化BDDでの論理演算処理



- 論理演算の実行に従い、共有化BDDが成長していく
- 共有化BDD処理系では、論理の一致判定はポインタの一致判定のみで行える（定数時間）

否定枝(Negative edge)



- 否定演算を表すマークを枝につけることで、否定同士の関係にあるBDDを共有化する技法
- 否定演算が定数時間で実行可能に
  - グラフの根の枝の否定枝をon/offするだけ
- グラフの一意性を保つため、否定枝の使用場所を制限している
  - 1の定数節点は使用しない
  - 0枝に否定枝を使用しない

BDDパッケージ

- BDD処理系は世界各地の研究機関で1990年頃から盛んに開発された
  - BDDパッケージとして無料配布されているソフトウエアもいくつかある
- 多くの場合、CまたはC++のライブラリとして提供されている
  - BDDへのポインタを引数としてライブラリ関数を呼び出すと、メモリ上にBDDが生成され、新しいBDDへのポインタが値として戻ってくる。
  - ユーザはBDDの論理演算を呼び出すメインプログラムを書き、BDDパッケージをリンクしてコンパイルすると、BDD処理アプリケーションを作ることができる。
- BDDパッケージの実装技術については次週説明予定