アルゴリズム特論（第５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

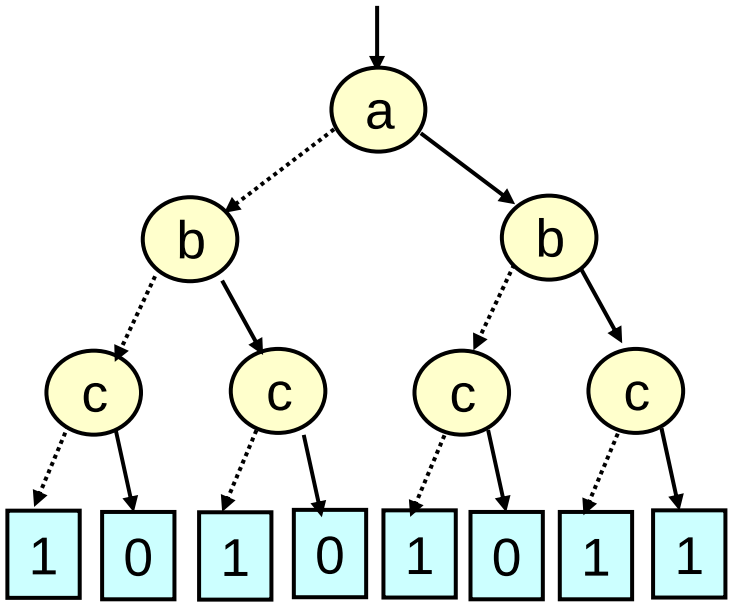
前回の内容

- **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

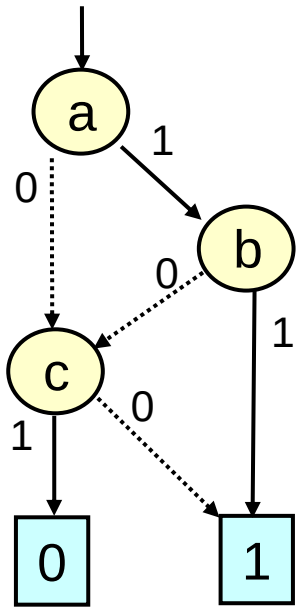
今回の内容

- **BDD処理系の実装技術**
  - BDDデータ構造の復習
  - データ構造
    - 節点の表現、テーブル上の表現
  - 節テーブルによる一意性の保証
    - ハッシュテーブルの実装
  - 論理演算アルゴリズム
    - 再帰的アルゴリズム、シャノンの展開
    - 演算例
    - 演算キャッシュの実装と計算時間
    - 否定演算と否定エッジ
    - 代入演算と計算時間
  - 記憶管理
    - 参照カウンタとガベジコレクション、演算キャッシュの初期化
    - 記憶領域の動的拡張

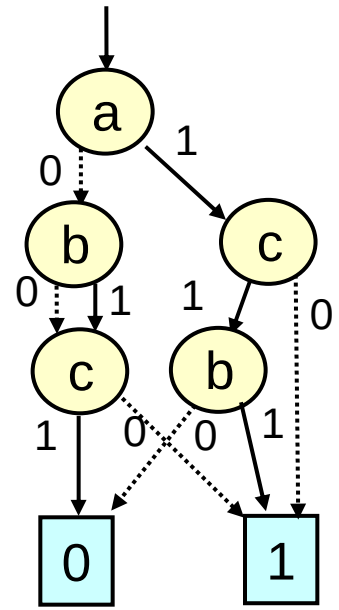
BDD（Binary Decision Diagram)（二分決定グラフ）とは



真理値表と等価なBDD
(Binary Decision Tree)



既約な順序付きBDD
(Reduced Ordered BDD)

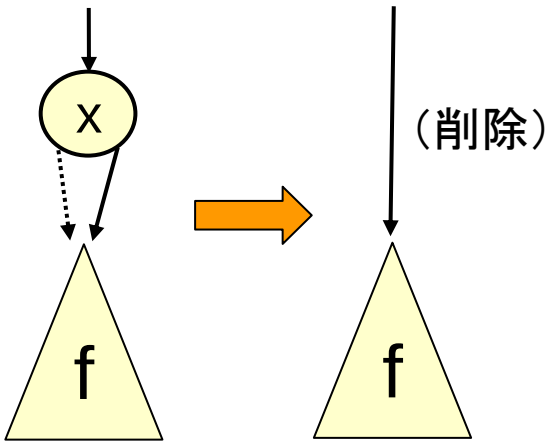


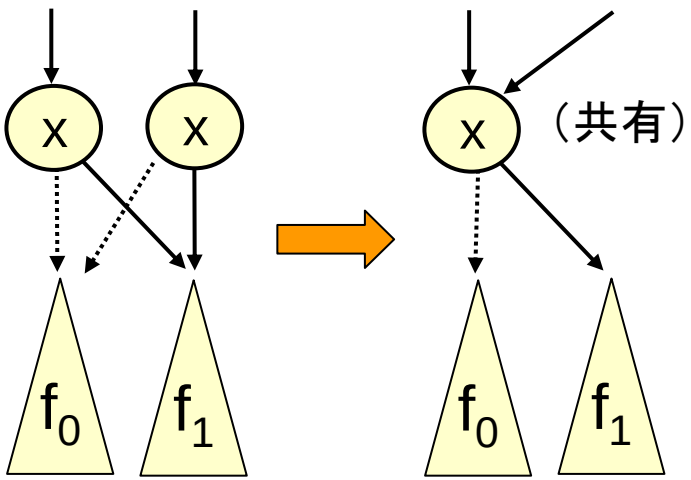
既約でも順序付き
でもないBDD
(Unordered BDD)

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

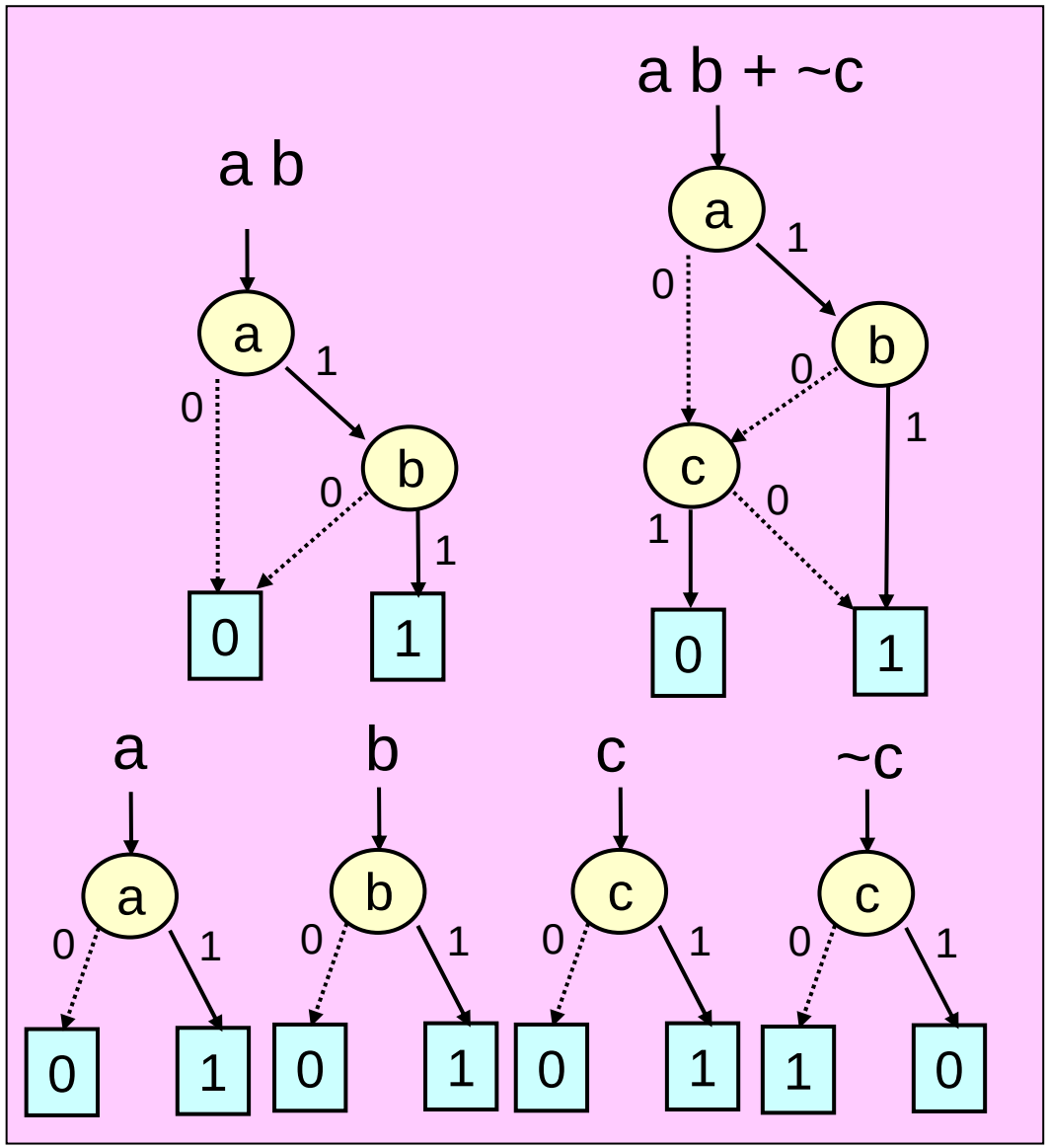
既約なBDDが得られる

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。

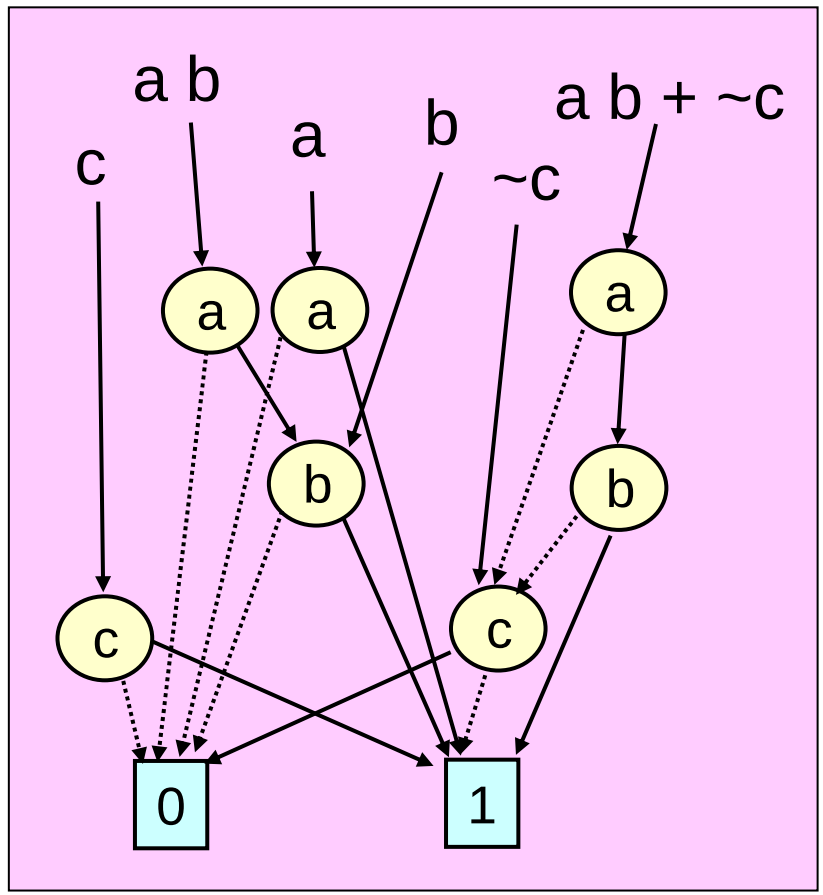
BDDの特徴

- 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）

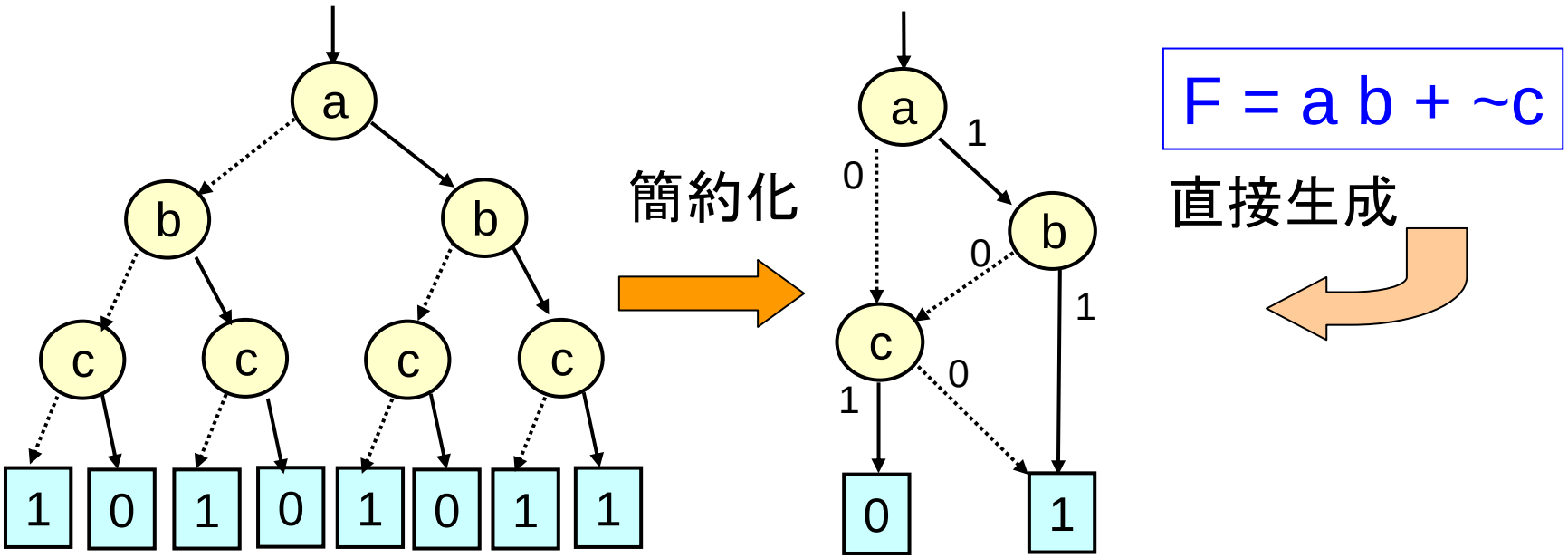
複数のBDDの共有化(Shared BDD)



- 変数の順序を揃える
- 全てのBDDを1つのグラフに共有化



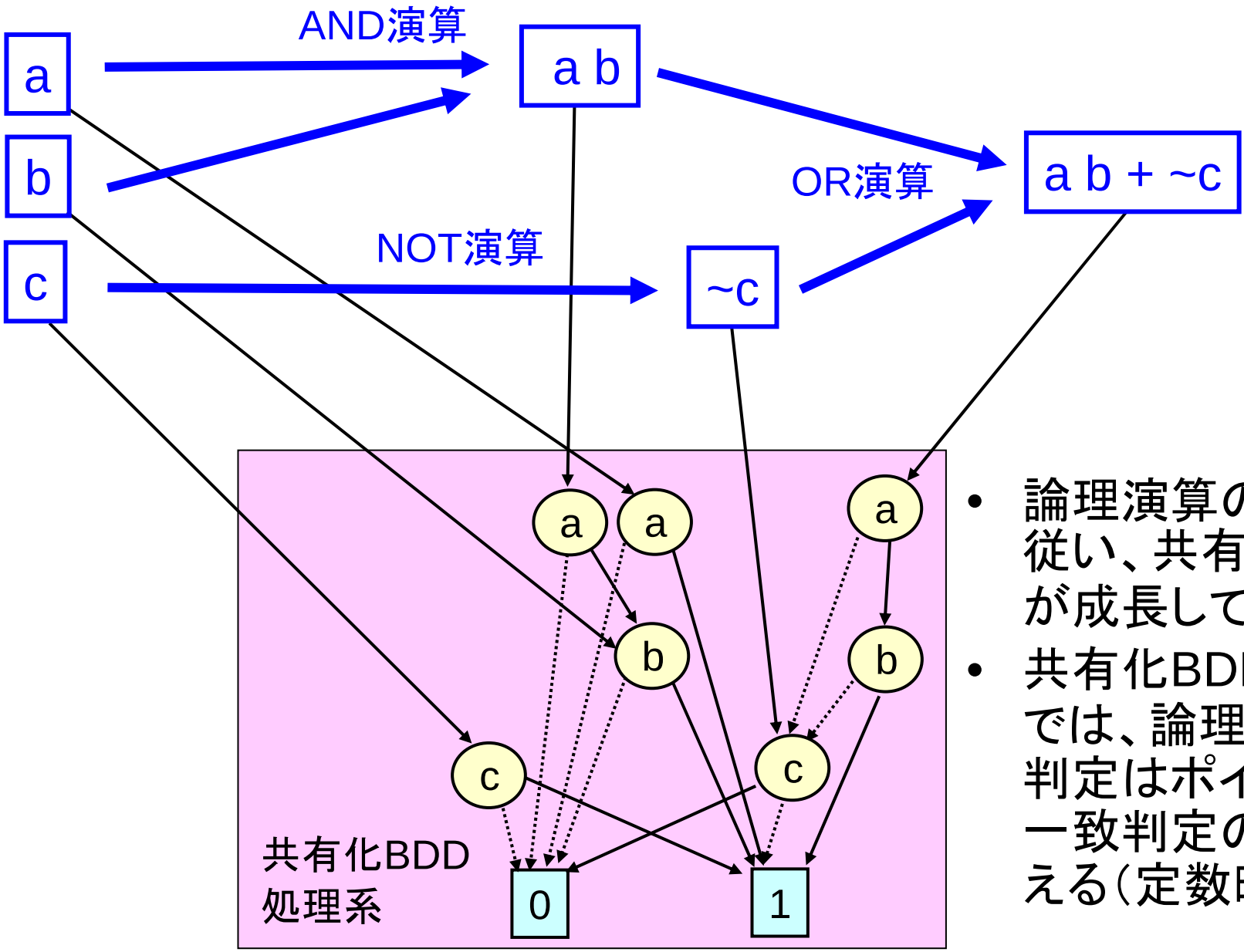
BDDの生成アルゴリズム



- 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

共有化BDDでの論理演算処理

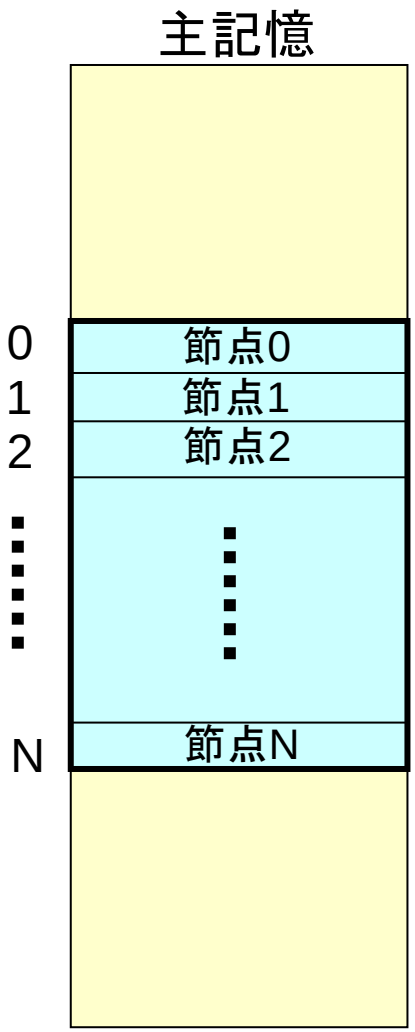


- 論理演算の実行に従い、共有化BDDが成長していく
- 共有化BDD処理系では、論理の一致判定はポインタの一致判定のみで行える（定数時間）

BDD処理系

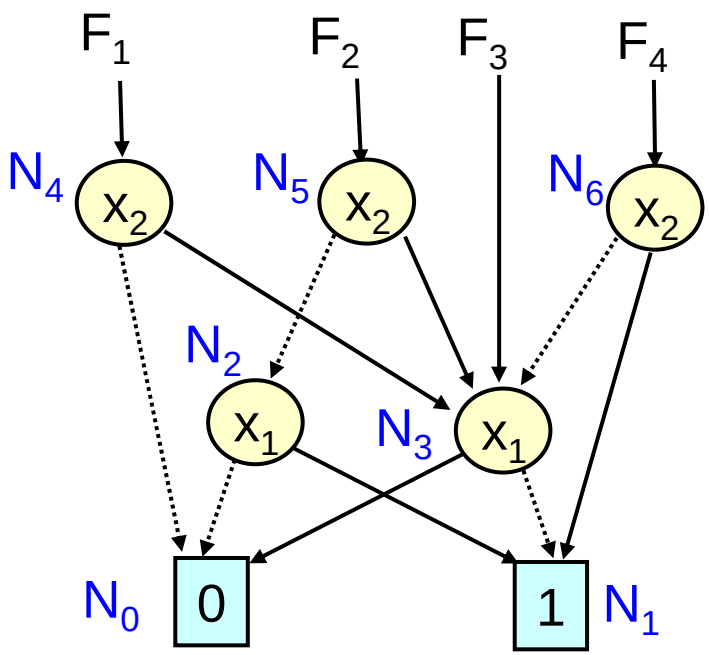
- BDD処理系は世界各地の研究機関で1990年頃から盛んに開発された
  - BDDパッケージとして無料配布されているソフトウエアもいくつかある
- 多くの場合、CまたはC++のライブラリとして提供されている
  - BDDへのポインタを引数としてライブラリ関数を呼び出すと、メモリ上にBDDが生成され、新しいBDDへのポインタが値として戻ってくる。
  - ユーザはBDDの論理演算を呼び出すメインプログラムを書き、BDDパッケージをリンクしてコンパイルすると、BDD処理アプリケーションを作ることができる。
- 複数のBDDを統一的に扱う共有化BDDの技法が広く用いられている
  - これ以降、原則として共有化BDD処理系について解説する

BDDの計算機上での内部表現



- BDD処理系では全てのBDDの節点を、計算機の主記憶（メモリ）に保持し、統一的に管理している
- 各節点は（入力変数番号、0枝、1枝）の3つの属性データと、グラフの管理用のポインタやカウンタなどの付属データを持つ。
- 典型的な実装では、節点を格納する記憶領域は0番地からの連続する領域にテーブルとして確保
  - 0枝、1枝はそれぞれの行き先の節点の番地（インデックス）を格納
  - 変数番号は自然数を使い、最下位を1として上位ほど大きな番号で表示
    （処理系によっては逆に上位を1にしている場合もあるが、上下の区別がつけばどんな番号付けでもよい）

BDD節テーブルの実装例



	変数	0枝	1枝	
N_0	-	-	-	← 0定数節点
N_1	-	-	-	← 1定数節点
N_2	x_1	N_0	N_1	
N_3	x_1	N_1	N_0	← F_3 (= ~x_2)
N_4	x_1	N_0	N_3	← F_1 (= ~x_1 x_2)
N_5	x_2	N_2	N_3	← F_2 = EXOR(x_1, x_2)
N_6	x_2	N_3	N_1	← F_4 (= ~x_1 + x_2)

$F_1 = \sim x_1\ x_2$
$F_2 = \mathrm{EXOR}(x_1, x_2)$
$F_3 = \sim x_1$
$F_4 = \sim x_1 + x_2$

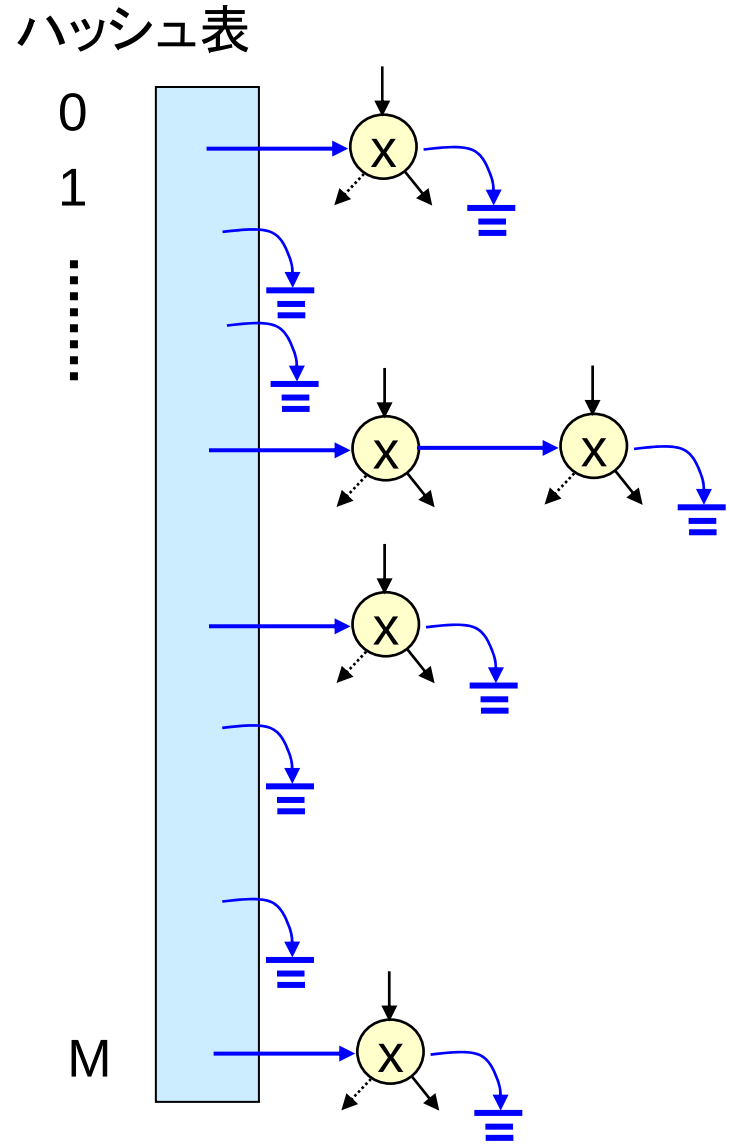
- 全ての論理関数は節テーブルの番地（インデックス）で識別される。
- インデックスは、最大節点数をMとすると log M ビットの記憶量を要する。
  - 通常は1ワード（多くの場合32ビット）の整数を使って表す。（約40億個まで識別可能）

節テーブルによる一意性の保証

- BDD処理系では、共有可能な部分グラフは必ず共有されていなければならない
  - 等価な節点が重複して存在してはならない
  - 変数番号、0枝、1枝の3つの属性が全て一致する節点がすでに存在していたら、新しい節点を作らずに、既存の節点の番地のみを返すようにする
- 節点の一致判定には、ハッシュテーブルの技法を用いる
  - 全ての節点は、(変数番号、0枝の番地、1枝の番地)の3つの数値をキーとするハッシュテーブルに登録しておく
  - 新たな節点を生成する前に必ずハッシュテーブルを検査して、等価な節点があれば共有する。重複する節点は一切作らない。
  - ハッシュテーブルの検査は定数時間
    (節点数が主記憶に収まっている限り)

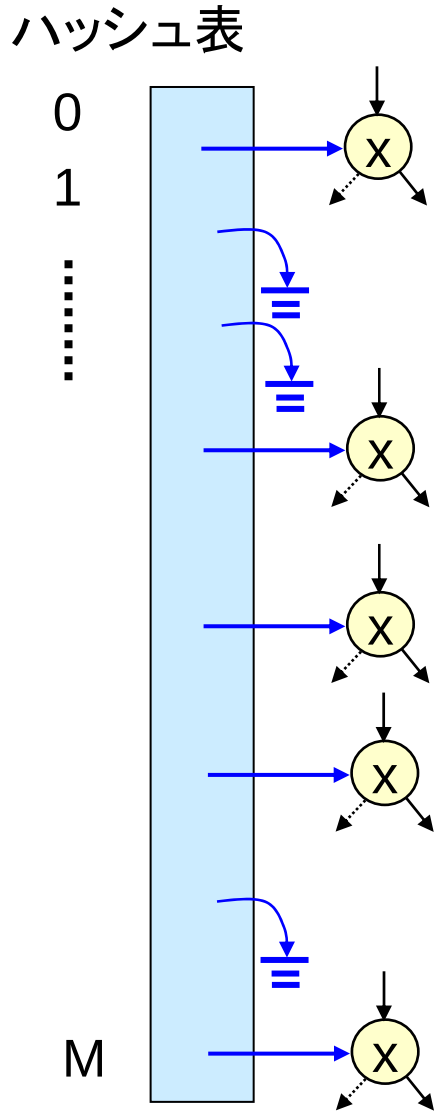
ハッシュテーブルの構成例(1)

- 外部ハッシュ方式
  - 3つの属性を適当に組合せて、ほぼランダムに散らばるハッシュ値を作り、その番地にBDD節点へのポインタ(インデックス)を格納
  - 運悪くハッシュ値が衝突した場合は、リストを作ってつなげて格納する。
  - ハッシュ表の配列に加えて、リストを作るためのポインタの記憶領域(節点ごとに1ワード)が必要
  - ハッシュ表のサイズは最大節点数と同程度あれば良好に動作(平均アクセス時間＜1サイクル)

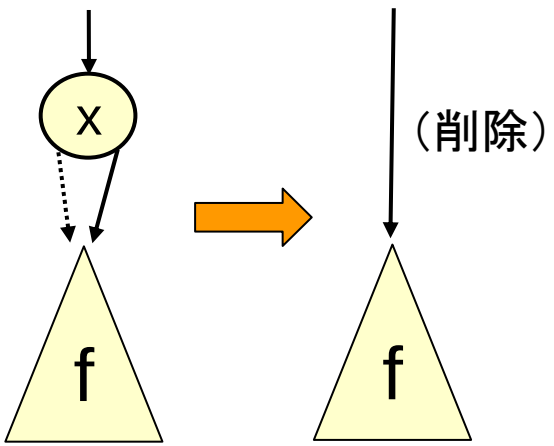


ハッシュテーブルの構成例(2)

- 内部ハッシュ方式
  - ハッシュが衝突した場合に、次の空いている番地に記録
  - 参照する際には、空欄が現れるまで順番に番地を増やしてチェックする。
  - ハッシュ表の配列のみで格納可能
  - 良好に動作するためのハッシュ表のサイズは最大節点数の2倍程度必要
  - 良好に動作させるために必要な記憶量は外部ハッシュ方式とほとんど同じ



冗長な節点の生成抑制

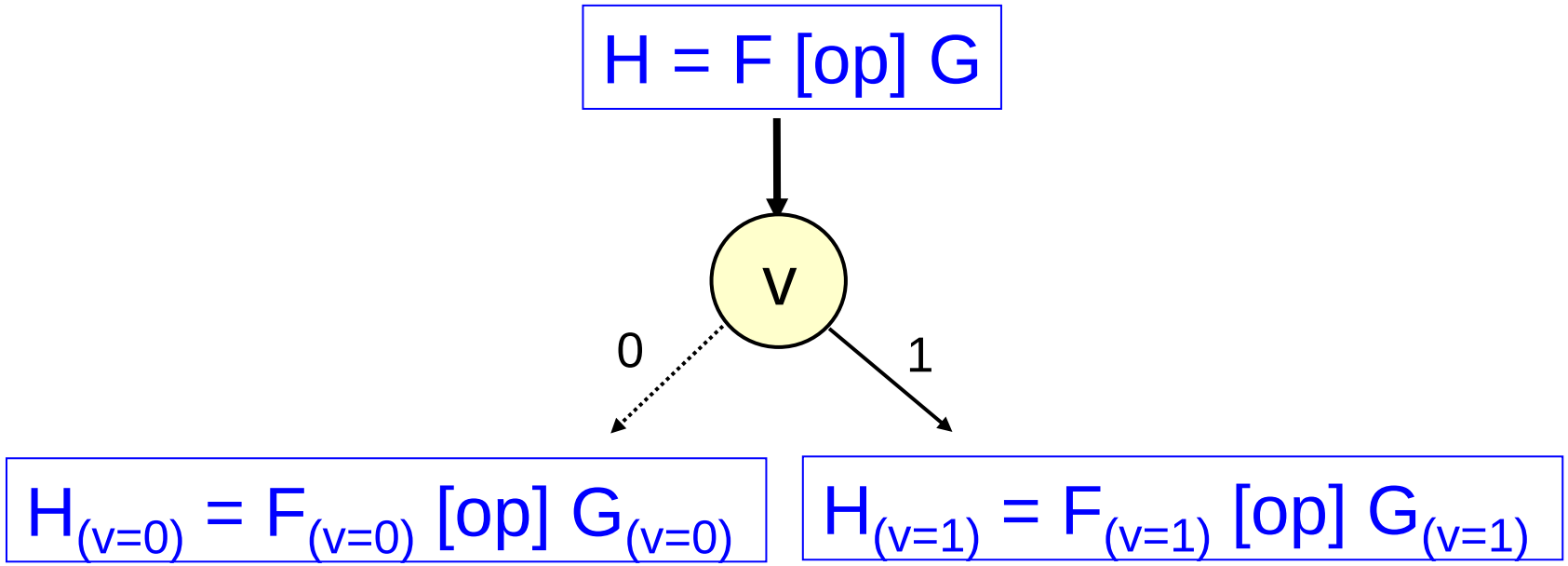


- 0枝と1枝が同じ部分グラフを指している場合は、新しい節点を作らずに、部分グラフをそのまま返す。
- BDD処理系では等価な節点が複数個存在しないことが保証されているので、0枝と1枝の番地を比較するだけで、節点が冗長かどうかが判定できる。

手続き GetNode(v, F0, F1)

- これまで説明した節テーブルの技法を1つの手続きにまとめたもの
  - BDD処理系の中で最も基礎的な手続き
  - 種々の演算の過程で、必要な節点を得るために呼び出される。
- GetNodeの動作（仕様）:
  引数として（変数番号:v, 0枝の番地:F0, 1枝の番地:F1）を与えられたときに、
  - F0とF1が等しければF0をそのまま返す。
  - 節テーブルを検査して、等価な節点が見つかれば、その番地を返す。
  - 等価な節点がなければ新しい節点を作ってその番地を返す。
  - 新しい節点を作ろうとして最大節点数を超えた場合にはエラーを返す。

二項論理演算アルゴリズム

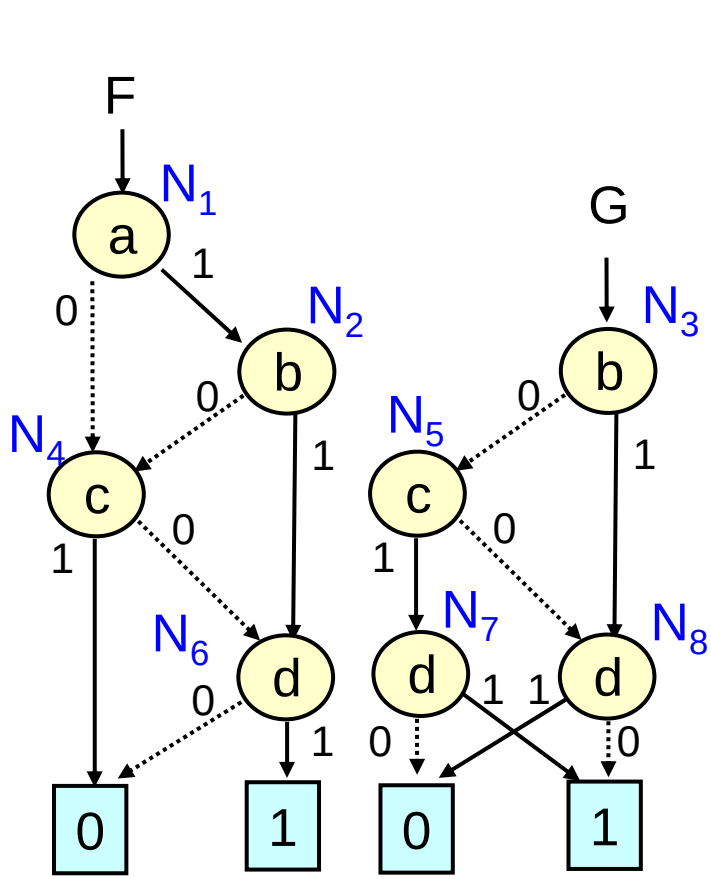


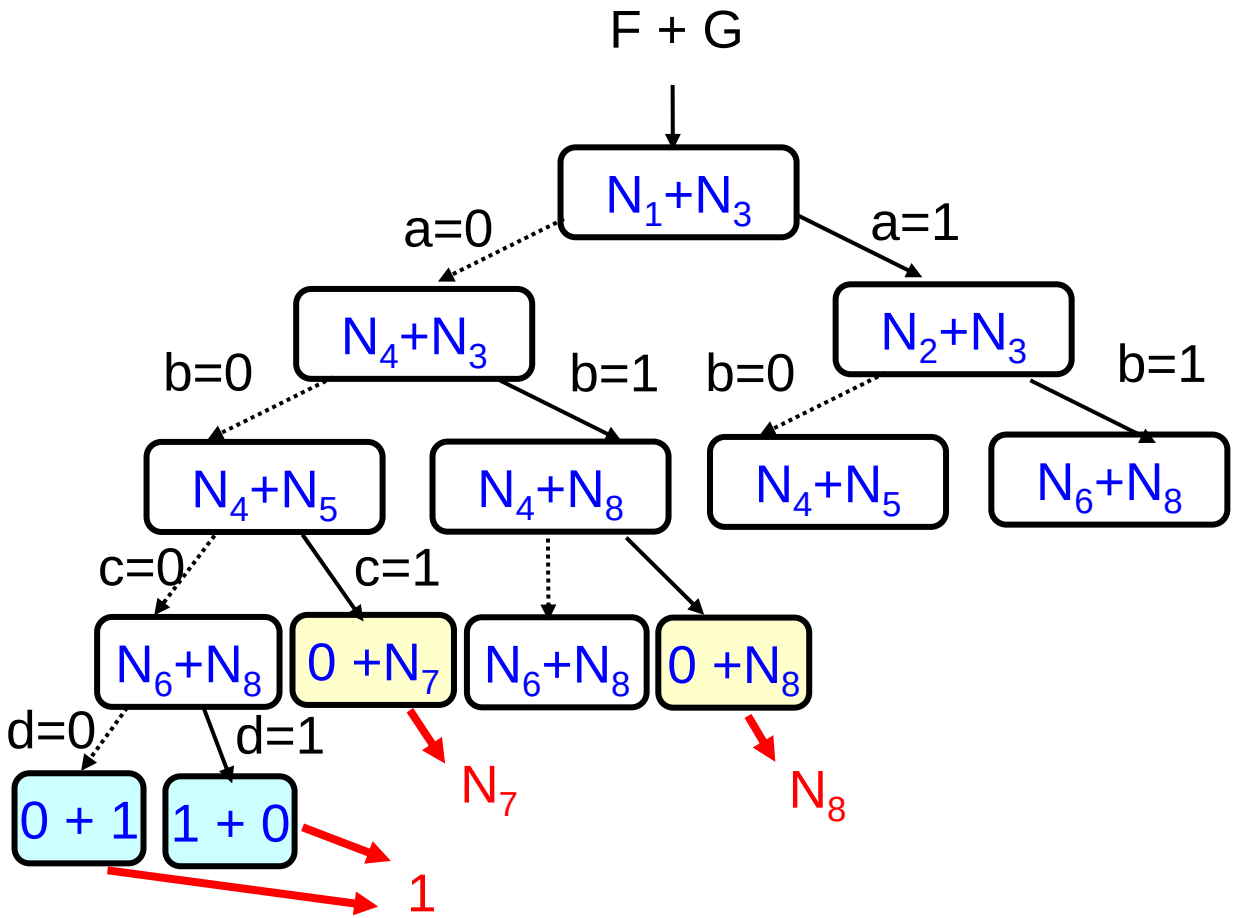
- ある変数vに0,1を代入して再帰的に展開
- 全ての変数を展開すると自明な演算になる
  (OR演算の場合) F+1=1, F+0=F, F+F=F …
- 各演算結果をBDDに組み上げる

二項論理演算手続き APPLY(op, F, G)

1. F,Gのいずれかが定数のとき、およびF=Gのとき
   - 演算子の種類opに応じた演算結果の節点の番地を返す。
     （例）F･0 = 0, F+F = F, EXOR(F,0) = F など
2. 両者の最上位変数 F.vとG.v が同じとき
   - H_0←APPLY(op,F_0,G_0), H_1←APPLY(op,F_1,G_1)を再帰呼出し
   - H_0=H_1であればH_0を返す。そうでなければ、
     GetNode(F.v, H_0, H_1)を呼出し、得られた節点の番地を返す。
3. F.v が G.v よりも上位のとき
   - H_0←APPLY(op,F_0,G), H_1←APPLY(op,F_1,G)を再帰呼出し
   - H_0=H_1であればH_0を返す。そうでなければ、
     GetNode(F.v, H_0, H_1)を呼出し、得られた節点の番地を返す。
4. F.v が G.v よりも下位のとき
   - FとGを入れ替えて、3.と同様に処理

二項論理演算の実行例


F
G
a
b
c
d
N_1
N_2
N_3
N_4
N_5
N_6
N_7
N_8
0
1


F + G
N_1+N_3
a=0
a=1
N_4+N_3
N_2+N_3
b=0
b=1
b=0
b=1
N_4+N_5
N_4+N_8
N_4+N_5
N_6+N_8
c=0
c=1
N_6+N_8
$0+N_7$
N_6+N_8
$0+N_8$
d=0
d=1
0 + 1
1 + 0
N_7
N_8
1

演算キャッシュによる高速化

- APPLY演算の再帰呼出しは二分木状になる
  - 通常の逐次処理系では二分木を深さ優先順にたどりながら実行する
  - 元のBDDに共有があるため、途中で同じ節点の組が多数現れることがある。
- 過去に演算を行った節点の組とその演算結果を記録する「演算キャッシュ」を用意すると処理を高速化できる
  - APPLYを実行するときに、同じ(op,F,G)の組が演算キャッシュに登録されていれば、再帰処理を打ち切り即座に結果を返せる。
  - キャッシュがすべてヒットすれば、F,G,Hの節点数の総和にほぼ比例する時間でAPPLY演算を実行できる。

演算キャッシュのデータ構造の例

op	F	G	H
OR	N_1	N_3	N_8
-	-	-	-
-	-	-	-
AND	N_4	N_0	N_0
AND	N_7	N_8	N_3
-	-	-	-
EXOR	N_3	N_5	N_4

- (op, F, G)の組をキーとするハッシュ表を作り高速に検索
  - 1エントリー当たり3.5ワード程度
- 全ての演算の組を記録しようとすると表が大きくなり過ぎる
  - 最近の演算だけを記録する「キャッシュ」形式とする
  - ハッシュ値が衝突した場合は、後からのデータを上書き
  - 過去のデータが失われた場合、冗長な計算が増えて遅くなるが、結果の正しさには影響しない

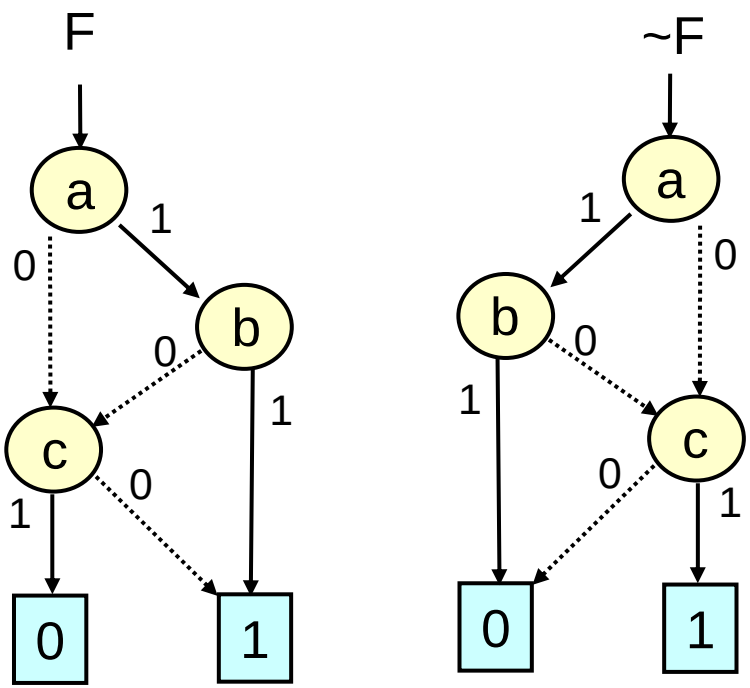
演算キャッシュのサイズと処理速度

- 演算キャッシュのサイズが不十分だと、無駄な演算が多くなり急激に速度が低下する
  - 多くの場合、実験的に最適なサイズを決めている。
    （例えば最大節点数の1/4のエントリー数）
- 演算キャッシュがヒットする確率を上げる工夫:
  - 自明な演算（例えば定数を含む場合など）は登録しない
  - 可換な演算（例えば F+G = G+F）ならば、
    どちらか一方のみ登録する
  - 否定枝が使える場合は、AND(F,G) = ~ OR(~F, ~G)の変換式を用いて、演算子の種類を減らす
  - 枝が2本以上集中している節点のみ記録する

二項論理演算の計算時間

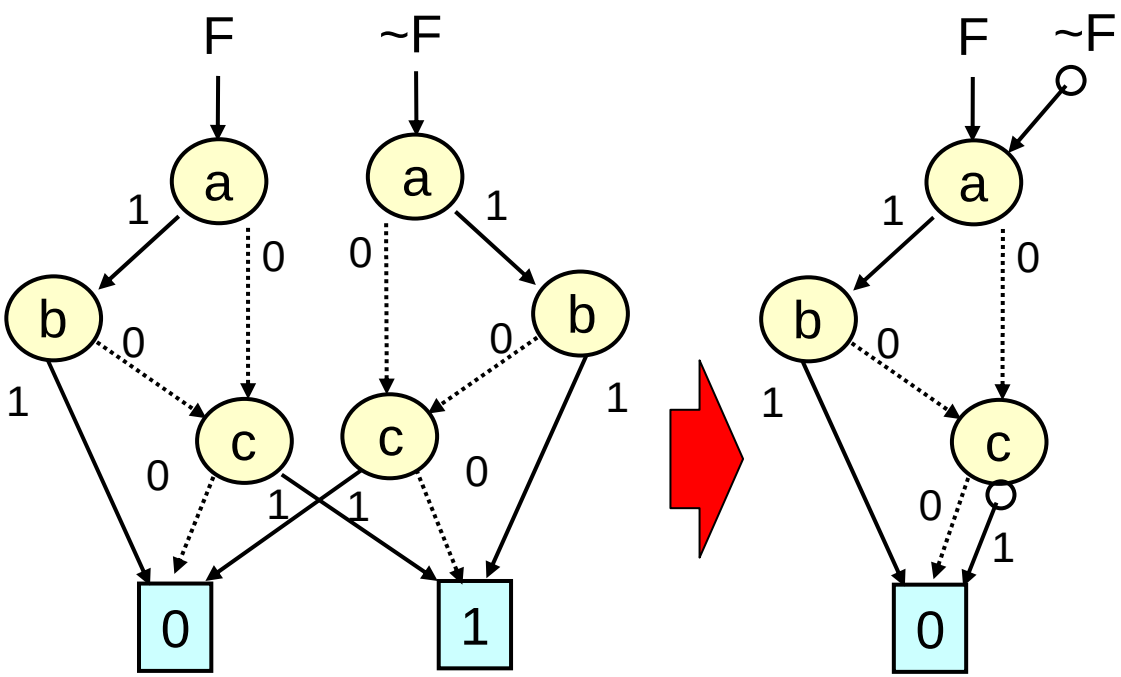
- グラフ F の節点数を |F| と表すと、H = F [op] G の節点数は、最悪の場合、$|H| = |F| \times |G|$ となる。
- 論理演算に要する計算時間は何も工夫しなければ、入力変数をnとすると$O(2^n)$となる。
  - しかし、演算キャッシュなどの効率化技法を使うことにより、$O(|F| + |G| + |H|)$ で抑えられる。
    - |F|, |G| が大きくても |H| が小さくなる場合がある。その逆もある。
  - グラフの節点数が小さくなるような場合であれば、nが大きくても劇的に高速に論理演算を行うことができる。
  - グラフのサイズが指数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

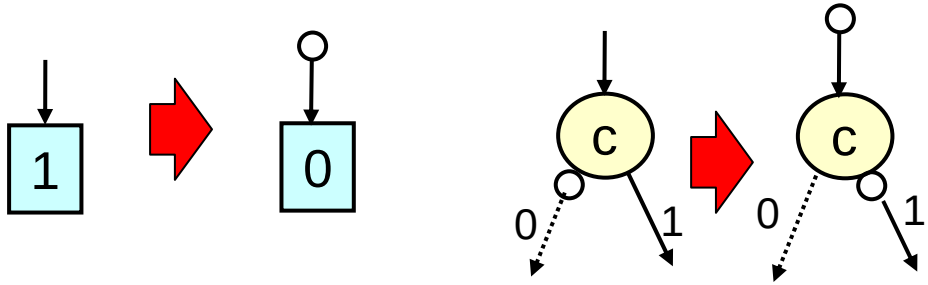
否定演算アルゴリズム



- 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

否定枝(Negative edge)

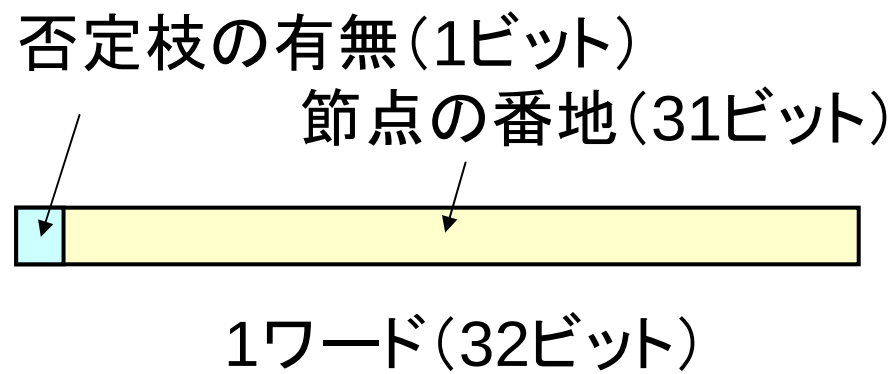




- 否定演算を表すマークを枝につけることで、否定同士の関係にあるBDDを共有化する技法
- 否定演算が定数時間で実行可能に
  - グラフの根の枝の否定枝をon/offするだけ
- グラフの一意性を保つため、否定枝の使用場所を制限している
  - 1の定数節点は使用しない
  - 0枝に否定枝を使用しない

否定枝の実装

- 否定枝の情報は、節点のインデックスに埋め込むことにより効率よく扱える
  - インデックスを整数とみなすと、正負の符号に相当する
  - BDDの否定演算は、インデックスの符号部分を反転するだけ（定数時間）
  - 否定枝も含めて、BDDの一致判定を1ワード整数の比較で行うことが可能
  - ただし否定枝に1ビット割当てるため、処理系で扱える最大節点数が 2^{32}個から2^{31}個に減る
    - 普通はその限界より前に実メモリが足りなくなる



代入演算

- BDD F, 変数x, 定数値0（または1）が与えられたとき、変数xに定数値を代入したときのFを表すBDDを作り、その節点へのインデックスを返す演算
  - 代入変数xがFの最上位変数F.vよりも上位にあるとき：
    Fのインデックスをそのまま返す。
  - xがF.vと同じとき：
    F_0（またはF_1）をそのまま返す。
  - xがF.vよりも下位にあるとき：
    F_0, F_1それぞれについて代入演算を再帰的に呼出し、
    その演算結果の節点を組み上げてBDDを作る。
- 演算キャッシュを効果的に使えば、Fの中でxより上位にある節点の個数に比例する計算時間で実行可能。

充足解探索

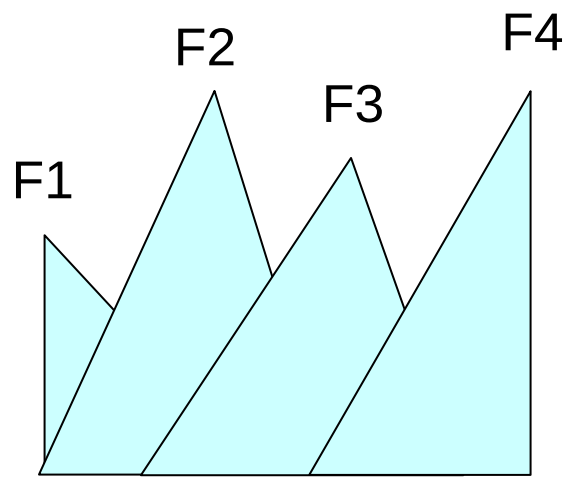
- ある論理関数が充足可能かどうか（恒偽関数でないかどうか）の判定はただちに可能
  - BDDを指すインデックスが0定数節点でなければよい
- 充足可能な場合に、充足解（論理を1にするための各変数の0,1の割当）を求めることも容易
  - 0定数節点を指さない枝をたどっていけば、必ず1定数節点に到達できる。その経路が充足解を表す。
  - グラフの節点数ではなく変数の数に比例する計算時間
- 充足解の個数のカウント
  - 0枝側の解の個数と1枝側の解の個数を加えればよい（変数番号に飛び越しがあるときは個数を2倍する）
  - 演算キャッシュが機能すれば、節点数に比例する時間で計算可能

最適解探索・真理値表密度の計算

- 入力変数に1を代入するときのコスト（例えばxを1にすると100円、yを1にすると150円）が与えられているとき、コスト最小の充足解（最適解）を求める問題
  - 0枝側の（最小コスト）と（1枝側の最小コスト＋最上位変数vのコストの和）を比較して小さい方が全体の最小コスト。
  - 演算キャッシュに最小コストを保存すると、節点数に比例する時間で計算できる。
  - 各節点での最小コストがわかれば、コストが小さい枝をたどっていけば、その経路が最適解となる。
- 論理関数が1になる割合（真理値表密度）や確率も求められる
  - 0枝側と1枝側の真理値表密度の平均を取ればよい。
  - 演算キャッシュが機能すれば節点数に比例する計算時間
  - 各入力変数が1になる確率（例えばxは70%, yは40%）が与えられている場合でも同じ計算時間で実行できる。

BDD処理系の記憶管理

- １節点あたりの記憶量
  - 節テーブル2.5ワード、ハッシュ表2ワード、演算キャッシュ３．５ワード×1/4 ……合計約5.5ワード（22バイト）
  - 100MBの主記憶で約400万節点を格納できる。
  - 主記憶からあふれると急激に(100倍以上）遅くなる
    - ハッシュテーブルは極めてランダムなアクセスなので、ハードディスクのバッファやキャッシュが全く役に立たない
- 記憶領域の有効利用のための技法
  - 参照カウンタによる記憶管理：
    使用済みの節点を
    再利用したい
  - 動的な領域確保：
    小さいBDDも巨大BDDも
    効率よく扱いたい



参照カウンタによる記憶管理

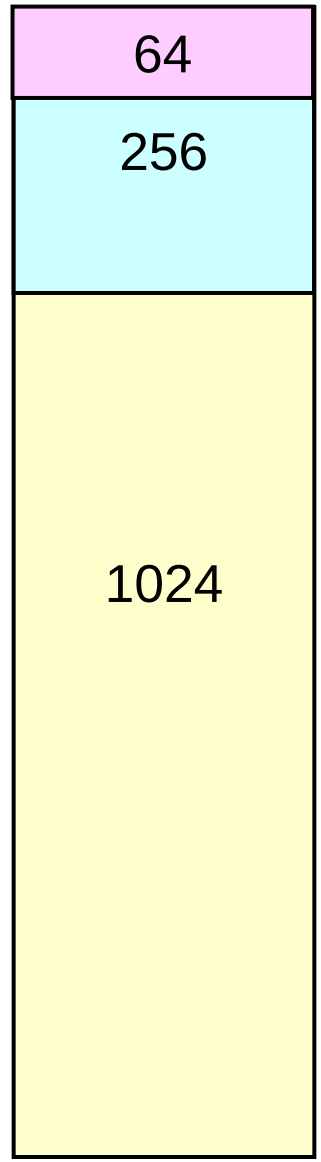
- 計算の途中結果のように、処理中に一時的に生成されて二度と参照されないBDDが多数発生する。
  - 不要なBDDは解放して記憶の再利用を図ることが実用上不可欠
  - 他のBDDと共有している節点は削除せず残す必要がある
- 各節点ごとに参照されている枝の本数を保持する「参照カウンタ」の技法が多くのBDD処理系で用いられている。
  - 1節点あたり1ワードの記憶量が余計に必要（合計で約26バイトとなる）
  - あるBDDが不要になったときは、根の節点の参照数を1減らし、0になれば実際に削除する。削除した場合は0枝1枝の参照数を減らし、再帰的に必要性を検査する
  - BDDへのインデックスを勝手にコピーすることは許されない。
    - 参照カウンタの正当性を維持するため、処理系が提供する複製命令と削除命令を用いる。
    - C++やJavaを使えば、カウンタの管理をコンパイラに任せることができる

演算キャッシュの初期化とガベジコレクション

- 節点を削除したとき、演算キャッシュにその節点のデータが残っていると、演算結果の正当性が失われてしまう。
  - 削除した節点を再利用すると、インデックスの値は同じでも、以前と異なる論理関数になる。
  - 削除した節点に関するデータをすべて探し出して除去することは困難なので、キャッシュ全体を初期化する等の対策が必要。
    →処理速度低下
- 処理速度低下を防ぐため、参照カウンタが0になっても削除せず温存しておき、どうしても足りなくなったときに一気にまとめて削除する技法（ガベジコレクション）が有効
  - 演算キャッシュの初期化回数を最小限に抑えられる
  - 削除してすぐまた必要になったときに即座に回復できる
  - ガベジコレクションは他の言語処理系でも使われる一般的な技法

記憶領域の動的拡張

- BDD処理系ではあらかじめ節点を記憶する領域を主記憶上に確保しておく必要がある
  - どのくらい確保しておけばよいかは、実行してみないとわからない
  - サイズが大きすぎると、実質的計算時間よりも初期化の時間の方が長くかかってしまう。
  - サイズが小さすぎるとあふれるかも知れない
- 最初は小さいサイズにしておいて、足りなくなったら定数倍（例えば4倍）の領域を確保しなおす技法が有効
  - 最後は主記憶のサイズで頭打ち
  - 記憶を確保しなおすために必要な時間は全体の1/4以下ですむ



BDD処理系の実験結果の例

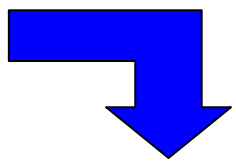
- 実用的な論理回路を集めたベンチマークで実験
  - 組合せ論理回路（ループや記憶素子を持たない回路）で、全ての出力端子および内部の途中計算部分の論理関数を同時にBDDで表現

	回路規模				
回路名	入力数	出力数	内部信号線数	BDD節点数	時間(秒)
sel8	12	2	29	78	0.3
enc8	9	4	31	56	0.3
adder8	18	9	65	119	0.4
adder16	33	17	129	239	0.7
mult4	8	8	97	524	0.5
mult8	16	16	418	66161	24.8
c432	36	7	203	131299	55.5
c499	41	32	275	69217	22.9
c880	60	26	464	54019	17.5
c1355	41	32	619	212196	89.9
c1908	33	25	938	72537	33.0
c5315	178	123	2608	60346	31.3

(計算時間はSun3, 24MByte)

BDD処理アルゴリズムのまとめ

- BDD処理の特長をまとめると、
  - 変数の展開順序を固定することにより、
    論理関数が内包する冗長性を自動的に抽出している
  - 「同じ節点は2個持たない」という効率化を徹底的に行う
  - 「同じ計算を2度しない」という効率化をできる限り行う
- BDD処理の本質とは:
  - ハッシュテーブルによる高速検索
  - ポインタによるリスト構造の操作



「主記憶上の任意の番地のデータに定数時間でアクセス可能」
という現在の計算機モデルの特長を最大限に活用している。