ZDD Ruby Package Documentation

対応 NYSOL パッケージ: Ver. 1.0〜

revise history:
March 10,2014：nysol パッケージへの統合に伴いインストール方法変更
February 15, 2014：minor modification
November 26, 2013：first release

2014 年 3 月 10 日
JST-ERATO MINATO Discrete Structure Manipulation System Project

目次

第 1 章

はじめに

1.1 概要

本パッケージは、ZDD(Zero-suppressed Binary Decision Diagrams: ゼロサプレス型二分決定グラフ) を利用し、重み付きのアイテムの組み合わせ集合をコンパクトに格納することを可能とする VSOP (Valued-Sum-Of-Products calculator)[4] を ruby 拡張ライブラリとして実装したものである。ZDD の生みの親である湊真一教授が開発/コーディングされたパッケージを ruby 言語において利用できるように拡張したもので、ERATO 湊離散構造処理系プロジェクトにおいて開発されたものである。

ZDD を使えば、膨大な組み合せ集合を非常にコンパクトに表現することが可能となる。例えば、スーパーマーケットにおける購入履歴から、ある一定の頻度以上購入された商品の組み合せ (頻出アイテム集合) を列挙すると、その数は膨大になることが多いが、ZDD を使えば、それら全ての組み合せを非常にコンパクトに格納することが可能となる。しかも、コンパクトに格納された ZDD オブジェクトに対して、各種演算を直接適用することが可能で、大規模なアイテム集合を非常に効率よく処理することが可能となる。例えば、列挙された数億件の頻出アイテム集合から「納豆」を含むパターンのみを選択したり、男性の頻出アイテム集合と女性の頻出アイテム集合との差異を計算したりすることが、それらの ZDD のサイズにほぼ比例して行うことが可能なのである。理論的な詳細は、巻末の参考文献を参照されたい。

本パッケージにおいて、ZDD は ruby オブジェクト (「**ZDD オブジェクト**」と呼ぶ) として扱われる。そして ZDD オブジェクトに対して定義された各種関数はクラスメソッドとして利用でき、また、ZDD オブジェクトに対する各種演算子 (`+`,`-`,`==`など) も、ZDD に対する演算子としてオーバーロードされており、ZDD と ruby の機能をシームレスに組み合わせて利用することを可能としている。さらに、自動的な型変換もサポートしており、よりストレスなくプログラミングができるように工夫している。

1.2 インストール

ZDD パッケージは、nysol の mining パッケージの一部として配布されている。ソースからのコンパイル、および rubygem によるインストールを選ぶことができる。インストールの詳細は、NYSOL のページの mining パッケージのマニュアルを参照されたい。

1.3 最大節点数の変更

本パッケージにおける ZDD の最大節点数はデフォルトで 4000 万である。この値を超える節点を使おうとするとエラー終了する。最大節点数は環境変数 `ZDDLimitNode` を設定することで変更可能である。例えば、bash シェルにおいては、以下のように設定すれば、最大節点数を 1 億に拡張できる。

```
$ export ZDDLimitNode=100000000
```

1 節点あたりのメモリ消費量は、32bit OS では 21〜25 バイト程度で、デフォルトの最大節点数 4000 万節点を使いきれば、主記憶 1GB 程度を消費する。64bit OS では、単純に実装すると 32bit OS の 2 倍になるが、様々な工夫により 3 割増し程度に抑えられており、1 節点あたり 28〜32 バイト程度である。デフォルトの最大節点数 4000 万節点を使いきれば、主記憶 1.3GB 程度を消費する。利用する環境に応じて最大節点数を変更することで、より大規模な ZDD を構築することができる。

1.4 用語

以下では、本マニュアルで利用する専門用語について解説する。巻末の参考文献で使われている用語と異なることもあるので注意されたい。

アイテム、アイテム集合、項、重み付き積和集合、式

本パッケージでは、集合の要素を「**アイテム**」と呼び、アイテムを要素に持つ集合を「**アイテム集合**」と呼ぶ。スーパーマーケットにおける商品をアイテム、商品の組み合せをアイテム集合と考えれば分かりやすい。そしてアイテム集合に重みを与えた「**項**」を要素としてもつ集合を「**重み付き積和集合**」と呼ぶ。例えば、3 つのアイテム a,b,c についての重み付き積和集合は「`abc+3ab+4bc+7c`」のように表記する (この表記形式を「重み付き積和形式」と呼ぶ)。これは、`abc`,`3ab`,`4bc`,`7c` の 4 つの項から成り立ち、`3ab` は、重みが 3 のアイテム集合 {a,b} であることを意味する。スーパーマーケットで言えば、3 つの商品 `a`,`b`,`c` を同時に購入した顧客が 1 人いて、`a`,`b` を同時に購入した顧客が 3 人いて、といった意味付けをすると分かりやすいであろう。

空アイテム集合、ZDD 定数オブジェクト

要素のないアイテム集合のことを「**空アイテム集合**」と呼ぶ。重み付き積和集合「`abc+3ab+4bc+7c+3`」について考えると、最後の項 `3` は空アイテム集合の重みが 3 であることを示している。スーパーマーケットの例で言えば、何も買わなかった人が 3 人いたと考えればよい。また空アイテム集合の ZDD オブジェクトのことを特に「**ZDD 定数オブジェクト**」と呼ぶ。

アイテム順序表

ZDD は 2 分決定木を縮約した 2 分決定グラフと呼ばれる構造を持つが、その 2 分決定木のレベル (深さ) がアイテム対応している。そしてそのレベル、すなわち根から葉に向かうアイテムの順序は「**アイテム順序表**」と呼ばれる表によって管理されている。このような管理が必要になるのは、アイテムの順序によって ZDD のサイズ (節点数) が大きく影響を受けるからである。ZDD のサイズが大きくなるということは、それに対する演算速度も低下することを意味する。アイテム順序表は symbol 関数によって随時登録されることになる。組み合わせ数が極端に大きくなる場合には、登録順序を考慮する必要があるかもしれない。

1.5 表記

アイテム集合の表記

ruby の実行結果として表示されるアイテム集合は、アイテムおよび重みがスペースで区切られて出力される。ただし、本文やコメントの中では、簡単のために、アイテムを全てアルファベット一文字で表し、スペース区切りを省略している。例えば、重み 3 の {a,b,c} のアイテム集合は、実際の実行結果としては「`3 a b c`」と表示されるが、本文中では「`3abc`」と表記している。

実行例

本マニュアルでは多数の実行例を掲載している。そこで使われている記号の意味は以下の通りである。

- `>`：ruby のメソッド入力を表す。
- `$`：シェルのコマンドライン入力を表す。
- `#`：コメントを表す。
- 記号なし：実行結果を表す。

また実行例は、基本的には ruby のコマンドライン実行ツールである irb で実行した結果のログを貼り付けたものである。

第 2 章

チュートリアル

本チュートリアルでは、まず ZDD の基本的な利用方法を示し、最後に応用例として N クイーン問題の解法を示す。示された実行例は、ruby スクリプトを記述して実行してもよいし、irb で一行ずつ実行してもよい。なお、本チュートリアルの実行は、ZDD ruby 拡張ライブラリが既に正しくインストールされており、また ruby についての基本的な知識があることを前提としている。

2.1 ZDD ruby 拡張ライブラリの require

zdd ruby 拡張ライブラリは rubygems を使ってパッケージングされているので、以下の通り rubygems を require した後に zdd を require する。

```
> require 'rubygems' # ruby 1.9 以降は必要ない
> require 'zdd'
```

2.2 利用するアイテムの宣言と定義

利用するアイテム名を宣言するために ZDD::symbol メソッドを利用する。一つのメソッドで一つのアイテムが宣言できる。アイテム宣言の順序は重要で、その順序が ZDD の木構造における根から見た階層レベルに対応する。宣言の順序により異なる構造の ZDD が作成され、そのサイズに大幅に影響を与えることがある。以下では、"a","b","c","d"の 4 つのアイテムをその順序で宣言している。アイテム順序にこだわらないのであれば、symbol 関数による宣言をスキップし、次に説明する itemset 関数から利用することもできる。

```
> ZDD::symbol("a")
> ZDD::symbol("b")
> ZDD::symbol("c")
> ZDD::symbol("d")
```

symbol では単に利用するアイテムを宣言しただけなので、次に、それらのアイテムから構成されるアイテム集合の定義、すなわち ZDD オブジェクトを作成していく。ZDD::itemset メソッドにアイテム名をスペース区切りで列挙することでアイテム集合の ZDD オブジェクトを作成することができる。以下では 3 つのアイテム"a","b","c"で構成されるアイテム集合 a,b,c を表す ZDD オブジェクトを ruby の変数 a にセットしている。オブジェクトの内容は show メソッドにより積和形式で表示される。

```
> x=ZDD::itemset("a b c")
> x.show
 a b c
```

以下では、後の節のために 1 つのアイテムから構成されるアイテム集合を作っておく。アイテム名と ruby 変数が同じ a であるが、これら 2 つは全く別ものであることに注意する。

```
> a=ZDD::itemset("a")
> b=ZDD::itemset("b")
> c=ZDD::itemset("c")
> d=ZDD::itemset("d")
> a.show
 a
```

2.3　演算

ZDD には四則演算を始めとして様々な演算が定義されており、それらを組み合わせることで、ZDD オブジェクトを自由に加工することが可能となる。以下に、いくつかの例を示す。一般的な多項式と同様に展開されるものもあれば、そうでないものもある。

```
> (a+a).show # 同じアイテム集合の足し算により重みが足しこまれる。
2 a
> (2*a).show # 定数を掛けても同様の結果になる。
2 a
> (a*b).show # 異なるアイテムの掛け算でアイテム集合にアイテムが追加される
a b
> (a*a).show # 同じアイテムの掛け算は、アイテムが 2 つになるので重みが 2 になる。
2 a
> (2*a-a).show # 引き算
a
> ((a*b*c+b*d+c)/b).show # 割り算
a c + d
> ((a+b)*(c+d)).show
a c + a d + b c + b d
> ((a+1)*(a+1)).show # 最後の項の"1"は空アイテム集合の重み。
3 a + 1
> ((a+1)*(a-1)).show
a - 1
```

演算の例として、最後にアイテム集合 {a,b,c,d} の全ての部分集合を列挙してみよう。以下にそれを実現する式とその結果を示す。

```
> f=(a+1)*(b+1)*(c+1)*(d+1)
> f.show
a b c d + a b c + a b d + a b + a c d + a c + a d + a + b c d + b c +
  b d + b + c d + c + d + 1
```

式の展開方法は、上記のケースにおいては、一般的な多項式の展開方法と同様に考えれば良い。右辺の演算結果として構築された ZDD オブジェクトが ruby 変数 f に代入されている。そして結果、$16(=2^4)$ のアイテム集合が列挙されていることがわかる。式の最後の項である 1 は、空のアイテム集合の重みであることに注意する。

2.4　頻出アイテム集合

今、スーパーマーケットで 4 人のお客さん (f,g,h,i) が、商品 a,b,c,d について以下の通り買い物をしていたとする。

- f: a,b,c,d
- g: b,d
- h: a,c,d
- i: a,b,d

これらの購買データから、3 人以上に共通するアイテム集合を求めてみよう。手順は簡単で、それぞれのお客さんが購入したアイテム集合の全部分集合を求めておき、それらを合計すればよい。以下にその方法を示す。

```
> f=(a+1)*(b+1)*(c+1)*(d+1)
> g=(b+1)*(d+1)
> h=(a+1)*(c+1)*(d+1)
> i=(a+1)*(b+1)*(d+1)
> all=f+g+h+i
> all.show
a b c d + a b c + 2 a b d + 2 a b + 2 a c d + 2 a c + 3 a d + 3 a + b c d + b c +
 3 b d + 3 b + 2 c d + 2 c + 4 d + 4
```

上記の結果から、商品 a,b を購入していた顧客は 2 人で、a,c,d を購入した顧客も 2 人であることがわかる。さらに、重みが 3 以上の項を選択するには以下のように termsGE 関数を用いればよい。

```
> all.termsGE(3).show
 3 a d + 3 a + 3 b d + 3 b + 4 d + 4
```

また、アイテム集合"a b"を含むアイテム集合を選択するには restrict 関数を、逆に、アイテム集合"a b"に含まれるアイテム集合を選択するには permit 関数を用いればよい。

```
> all.restrict("a b").show
a b c d + a b c + 2 a b d + 2 a b
> all.permit("a b").show
2 a b + 3 a + 3 b + 4
```

ZDD パッケージには、上記の方法以外にも頻出アイテム集合を列挙しその結果を ZDD オブジェクトとして格納する方法がいくつかある。詳細は、freqpatA 関数や lcm 関数を参照されたい。

2.5 キャスト

ここまでには明示的に説明はしなかったが、本パッケージでは ZDD の各種演算および関数に対して与えるデータはその型に応じて自動的に型変換 (キャスト) される。例えば、前節で示した演算 2*a は、ZDD オブジェクトである変数 a に ruby の整数である 2 を掛けたものである。ZDD の乗算演算子*は 2 つの ZDD オブジェクトを引数にとる、この例のように一方が ZDD オブジェクトでない場合、その内容を自動的に ZDD オブジェクトに変換する。すなわち 2*a は、内部で ZDD::constant(2)*a が実行されることになる。ここで、constatnt 関数は、空のアイテム集合の重みを定義する関数である。

以下の例では ruby 文字列"a b"は自動的に ZDD オブジェクトに変換される。内部的には、(a+ZDD::itemset("a b")).show を実行している。

```
> (a+"a b").show
 a b + a
```

以下は演算子の引数が二つとも ruby の String オブジェクトであるため、単なる文字列の結合となってしまう。

```
> s="a b"+"c d"
> p s
"a bc d"
```

2.6 制御文との組合せ

2.2 と 2.3 節に示した部分アイテム集合を全列挙する一連の流れは、制御文と組み合わせることでも実現できる。その例を以下に示す。以下の例では symbol 関数によるアイテムの宣言はせず、直接 itemset 関数によりアイテム集合を

定義している。そして乗算代入演算子*=により ruby 変数 t に、次々と演算結果を累積していっている。

```
> t=ZDD::constant(1)
> ["a","b","c","d"].each{|item|
>  t*=(ZDD::itemset(item)+1)
> }
 a b c d + a b c + a b d + a b + a c d + a c + a d + a + b c d + b c +
  b d + b + c d + c + d + 1
```

2.7　N クイーン問題の解法

ここでは、これまでに紹介した ZDD の各種演算を応用して N クイーン問題を解く方法を紹介する。N クイーン問題とは、N×N のチェス盤上に N 個のクイーンを、お互いにとられないように配置する問題である。クイーンは将棋の飛車と角を合わせた動きのできるコマで、図 2.1 に示されるように上下左右と斜め 4 方向の計 8 方向に進むことができる。

	x		x	
		x	x	
x	x	x	o	x
		x	x	x
	x		x	

図 2.1　”o”で示されたクイーンの動けるマス目を”x”で示している。

(0,0)	(0,1)	(0,2)	(0,3)	(0,4)
(1,0)	(1,1)	(1,2)	(1,3)	(1,4)
(2,0)	(2,1)	(2,2)	(2,3)	(2,4)
(3,0)	(3,1)	(3,2)	(3,3)	(3,4)
(4,0)	(4,1)	(4,2)	(4,3)	(4,4)

図 2.2　5 × 5 のチェス盤の座標

o				
		o		
				o
	o			
			o	

図 2.3　5 クイーン問題の解の一例

5 クイーン問題を ZDD を用いて解く Ruby スクリプトを図 2.4 に示す。このスクリプトで想定しているチェス盤の座標を図 2.2 に示している。このスクリプトの基本的な考え方は以下の通りである。まず、クイーンを配置するマス目をアイテムと考え、$5 \times 5 = 25$ のマス目の全組み合せ、すなわち 2^{25} の組み合せのアイテム集合を全列挙する。そのためには 2.3 節で解説した方法を使えばよい。そして、その中から条件に合うアイテム集合を選択する。ここで条件としては以下に示される 2 つを満たせば十分である。

1. お互いに取られるようなアイテム集合を全て削除する。
2. サイズが 5 未満のアイテム集合を削除する。

ちなみに、サイズが 5 より大きいアイテム集合は、1 の条件によって削除されることに注意する。詳細は、図中にコメントとして記しているので、そちらを参考に動きを確認してもらいたい。スクリプトの最後には、ZDD による演算の途中で格納された ZDD オブジェクトの項の数と内部の接点数を出力している。

以下に実行結果を示す。25 マスの前組み合わせは約 3355 万通り (2^25 の値) と膨大であるにも関わらず、ZDD の節点数はわずかに 25 である。restrict 関数によって選択されたお互いに取られる組み合わせ数 (selNG) もほぼ同じ数であるが、その ZDD 節点数は 587 と急激に増えており、時間的にも最も時間を要するところでもある。最終的に 10 のアイテム集合が解として列挙されている。その中の最初の解 (項) である”0,0 1,2 2,4 3,1 4,3”の配置を図 2.3 に示す。

```
all    : 33554432          25
selNG  : 33553970          587
selOK  : 462          193
selLT  : 452          199
ans    : 10          40
time: 0.003749
 0,0 1,2 2,4 3,1 4,3 + 0,0 1,3 2,1 3,4 4,2 + 0,1 1,3 2,0 3,2 4,4 + 0,1
  1,4 2,2 3,0 4,3 + 0,2 1,0 2,3 3,1 4,4 + 0,2 1,4 2,1 3,3 4,0 + 0,3 1,0
  2,2 3,4 4,1 + 0,3 1,1 2,4 3,2 4,0 + 0,4 1,1 2,3 3,0 4,2 + 0,4 1,2 2,0
  3,3 4,1
```

```
#!/usr/bin/env ruby
#encoding: utf-8
require "zdd"

n = 5

# 以下の二重ループにて、25 マス（アイテム）の全組み合わせのアイテム集合が列挙され、変数 all に格納される。
# アイテム名には座標を示した文字列"i,j"を用いている。
all=ZDD.constant(1)
(0...n).each{|i|
        (0...n).each{|j|
                all *= (1+ZDD::itemset("#{i},#{j}"))
        }
}

# 以下で、お互いに取られる 2 マスの組み合せを全列挙し、変数 ng に格納する。
ng=ZDD.constant(0)
(0...n).each{|i| # 行ループ
  (0...n).each{|j| # 列ループ
     # マス目 i,j と同じ行番号（i）を持つアイテムペア
    (j+1...n).each{|k|
      ng+=ZDD::itemset("#{i},#{j} #{i},#{k}") # 横
    }
     # マス目 i,j と同じ列番号（j）を持つアイテムペア
    (i+1...n).each{|k|
      ng+=ZDD::itemset("#{i},#{j} #{k},#{j}") # 横
    }
    # マス目 i,j と、そこから右斜め下方向のアイテムペア
    (1...[n-i,n-j].min).each{|k|
      ng+=ZDD::itemset("#{i},#{j} #{i+k},#{j+k}") #
    }
    # マス目 i,j と、そこから左斜め下方向のアイテムペア
    (1...[n-i,j+1].min).each{|k|
      ng+=ZDD::itemset("#{i},#{j} #{i+k},#{j-k}") #
    }
  }
}
st=Time.new # 時間計測用
selNG=all.restrict(ng)      # 1) 全アイテム集合 all からお互いに取られるアイテムペアを含むアイテム集合を選択する。
selOK=selNG.iif(0,all)      # 2) 全アイテム集合 all から、1) で求めたアイテム集合を除外する。
selLT=selOK.permitsym(n-1) # 3) 2) の結果から、サイズが n-1 以下のアイテム集合を選択する。
ans  =selLT.iif(0,selOK)    # 4) 2) の結果から 3) で求めたアイテム集合を除外する。

# 計算過程で作成された ZDD のアイテム集合（totalweight 関数）の数、および ZDD の接点数（size 関数）を表示する。
# totalweight 関数は ZDD の各項の重みを合計するメソッドである。
# ここでは、全ての項の重みは 1 なので、結果として式に含まれるアイテム集合の数が分かる。
puts "all   : #{all.totalweight}\t #{all.size}"
puts "selNG : #{selNG.totalweight}\t #{selNG.size}"
puts "selOK : #{selOK.totalweight}\t #{selOK.size}"
puts "selLT : #{selLT.totalweight}\t #{selLT.size}"
puts "ans   : #{ans.totalweight}\t #{ans.size}"
puts "time: #{Time.new-st}"
ans.show # 解の表示
```

図 2.4 5 クイーン問題の ZDD による解法。

また、N を 4 から 11 までに設定したときの、ZDD の節点数、解の数、そして計算時間を表 2.1 に示す。ZDD の節点数としては、全升目の全組み合わせの ZDD オブジェクト (all) と、ZDD 節点数が最も多くなる ZDD オブジェクト (selNG) についてのみ掲載している。N=11 で selNG の接点数が約 2000 万となり、1 節点あたり 30 バイトで計算して約 600M バイトのメモリ量を消費することになる。計算途中のワークスペースも含めて、8GB メモリの PC では N=11 あたりが計算の限界となる。ZDD を用いたより効率的な解法が文献 [2] に示されているので、是非ともチャレンジしてもらいたい。

表 2.1　N の値による ZDD の節点数、解の数、処理時間

N	ZDD 節点数 (all)	ZDD 節点数 (selNG)	解数	計算時間 (秒)
4	16	142	2	-
5	25	587	10	-
6	36	2918	4	0.017
7	49	15207	40	0.094
8	64	83962	92	0.65
9	81	489665	352	4.74
10	100	2995555	724	34.7
11	121	19074050	2680	247.5

*OS: Mac OS X 10.6 Snow Leopard, CPU: 2.66GHz Intel Core i7, Memory: 8GB 1067MHz DDR3

第 3 章

ZDD 演算子・メソッド一覧

本章は演算子および関数のリファレンスを提供している。リファレンスは、書式、説明、例、関連の 4 つのパートで構成されている。書式は、以下のように例示される形式によって示されている。

書式

obj.lcm($type, transaction, minsup[, order, ub]$) $\rightarrow$ zdd

$transaction$: string

$type$: string

$minsup$: integer

$order$: string

ub : integer

引数と返り値はイタリック体で示されており、矢印の右側に演算子/関数の返り値が示されている。obj は、メソッドが適用される ZDD オブジェクトを表しており、zdd や $zdd1, zdd2$ は、引数や返り値としての ZDD オブジェクトを表している。その他の ruby オブジェクトの型については、書式の下に明示している。

3.1　%：剰余演算子

書式

$zdd1 \ \% \ zdd2 \rightarrow zdd3$

説明

$zdd1$ を $zdd2$ で除した余りを求め、その値を ZDD オブジェクト $zdd3$ で返す。

例

例 1: 基本例

/メソッドの例を参照のこと。

関連

/：除算演算子

3.2 *: 乗算演算子

書式

$zdd1$ * $zdd2 \rightarrow zdd$

説明

$zdd1$ と $zdd2$ の乗算を行う。一般的な多項式と同様に考えれば良いが、唯一、同じアイテム同士の乗算は 2 乗とはならない点が異なる。例えば、a*a=a と計算される。

例

例 1: **基本例**

```
> require 'zdd'
# a. 定数乗算
# 重み付き積和集合 x に定数 c を掛ける x*c では、x の各項の重みを c 倍する。
# 通常の多項式と同様に考えれば良い。
> a=ZDD::itemset('a')
> b=ZDD::itemset('b')
> c=ZDD::itemset('c')
> d=ZDD::itemset('d')
> (a*3).show
 3 a
> (-2*a).show
 - 2 a
> ((a+2*b+3*c)*4).show
 4 a + 8 b + 12 c

# b. 1 つのアイテム集合の乗算
# 重み付き積和集合 x に 1 つのアイテム集合 y を掛ける x*y では、x の各項目にアイテム集合 y を加える。
# ただし、同じアイテム同士の掛け算 a*a=a となることに注意する。
> x=a+2*a*b+3*c
> x.show
 2 a b + a + 3 c
> (x*c).show
 2 a b c + a c + 3 c
> (x*b).show
 3 a b + 3 b c
> (4*x*a).show
 8 a b + 12 a c + 4 a

# c. 2 つ以上のアイテム集合の乗算
# 2 つの重み付き積和集合 x,y の乗算 x*y では、x と y それぞれの各項から一つずつ選ぶ組み合わせ全てについて
# 上記 a,b の乗算を付す。
# 乗算の結果同じアイテム集合の項は加減算される。
> ((a+b)*(c+d)).show
 a c + a d + b c + b d
> ((a+b)*(b+c)).show
 a b + a c + b c + b
> ((a+b)*(a+b)).show
 2 a b + a + b
> ((a+b)*(a-b)).show
 a - b
```

関連

/ : 除算演算子

3.3 ＋：加算演算子

書式

$zdd1 + zdd2 \rightarrow zdd3$

説明

$zdd1$ と $zdd2$ の対応するアイテム集合同士の重みの加算を行う。

例

例 1: **基本例**

```
> require 'zdd'
> a=ZDD::itemset("a")
> b=ZDD::itemset("b")
> c=ZDD::itemset("c")
> x=3*a + 2*b
> y=2*a + 2*b + 4*c
> x.show
 3 a + 2 b
> y.show
 2 a + 2 b + 4 c
> (x+y).show
 5 a + 4 b + 4 c
```

関連

－：減算演算子

＋：プラス単項演算子

3.4　＋：プラス単項演算子

書式

＋ $zdd1 \rightarrow zdd2$

説明

$zdd1$ に含まれる全てのアイテム集合をそのまま返す (何もしない)。

例

例 1: 基本例

```
> require 'zdd'
> a=ZDD::itemset("a")
> b=ZDD::itemset("b")
> x=3*a+2*b
> x.show
 3 a + 2 b

> (+x).show
 3 a + 2 b
```

関連

－：マイナス単項演算子

3.5 - : 減算演算子

書式

$zdd1 - zdd2 \to zdd$

説明

$zdd1$ と $zdd2$ の対応するアイテム集合同士の重みの減算を行う。

例

例 1: 基本例

```
> require 'zdd'
> a=ZDD::itemset("a")
> b=ZDD::itemset("b")
> c=ZDD::itemset("c")
> x=3*a + 2*b
> y=2*a + 2*b + 4*c
> x.show
 3 a + 2 b
> y.show
 2 a + 2 b + 4 c

> (x-y).show
 a - 4 c
> (y-x).show
 - a + 4 c
```

関連

+ : 加算演算子

− : マイナス単項演算子

3.6　- : マイナス単項演算子

書式

$-zdd1 \rightarrow zdd2$

説明

$zdd1$ に含まれる全てのアイテム集合の重みの符合を変える。

例

例 1: 基本例

```
> require 'zdd'
> a=ZDD::itemset("a")
> b=ZDD::itemset("b")
> x=3*a+2*b
> x.show
 3 a + 2 b

> (-x).show
 - 3 a - 2 b
```

関連

+ : プラス単項演算子

3.7 / : 除算演算子

書式

$zdd1$ / $zdd2$ → $zdd3$

説明

$zdd1$ を $zdd2$ で除した値を計算する。詳細は以下の例に示す。

例

例 1: **定数除算**

重み付き積和集合 x の定数 c による除算 x/c は、x の各項 (アイテム集合) の重み (頻度) を c で整数除算した商を重みに持つアイテム集合が計算される。

```
> require 'zdd'
> a=ZDD::itemset("a")
> b=ZDD::itemset("b")
> x=13*a+3*b
> x.show
 13 a + 3 b

# a の項:13/5 の商は 2 であるから 2a。b の項:3/5 の商は 0 であるから 0b となり表示されない。
> (x/5).show
 2 a

# a の項:13/5 の余りは 3 であるから 3a。b の項:3/5 の余りは 3 であるから 3b となる。
> (x%5).show
 3 a + 3 b

# 除数の 5 に商を掛けて余りを足せば元の値 x に戻る。
> (5*(x/5)+(x%5)).show
 13 a + 3 b
```

例 2: 1 **つのアイテム集合による除算**

重み付き積和集合 x のアイテム集合 v による除算 x/v は、x の各項を v で除する演算である。通常の多項式と同様に考えれば良い。

```
> require 'zdd'
> a=ZDD::itemset("a")
> b=ZDD::itemset("b")
> c=ZDD::itemset("c")
> x=7*a*b+5*b*c
> y=7*a*b+5*b*c+2*c
> x.show
 7 a b + 5 b c
> y.show
 7 a b + 5 b c + 2 c

# 7ab/b の商は 7a。5bc/b の商は 5c。
> (x/b).show
 7 a + 5 c
```

```
# 7ab/b の余りは 0。5bc/b の余り 0。
> (x%b).show
 0

# 7ab/3b の商は 2a。5bc/3b の商は 1c。2c/3b の商は 0 となり表示されない。
> (y/(3*b)).show
 2 a + c

# 7ab/3b の余りは ab。5bc/3b の余りは 2bc。2c/3b の余りは 2c。
> (y%(3*b)).show
 a b + 2 b c + 2 c

# 除数の 3b に商を掛けて余りを足せば元の値 y に戻る。
> (3*b*(y/(3*b))+(y%(3*b))).show
 7 a b + 5 b c + 2 c
```

例 3: 2 つ以上のアイテム集合による除算

2 つの重み付き積和集合 x,y の除算 x/y は次のように計算される。除数 y の各項を T_i としたとき $Q_i = x/T_i$ を全ての i について求める。得られた Q_i 全てに共通して含まれるアイテム集合 (項) について、重みの絶対値が最小の項を商 Q と定義する。通常の多項式とは異なることに注意する。

```
> require 'zdd'
> a=ZDD::itemset("a")
> b=ZDD::itemset("b")
> c=ZDD::itemset("c")
> d=ZDD::itemset("d")

> x=2*a*b+4*a*c+a*d-2*b*c+3*b*d
> y=a+b
> x.show
 2 a b + 4 a c + a d - 2 b c + 3 b d
> y.show
 a + b

# x/y および x%y は以下のように計算される。
# Q1=(x/a)=2b +4c +d +0  +0  = 0  +2b +4c +d
# Q2=(x/b)=2a +0  +0 -2c +3d = 2a +0  -2c +3d
# Q1,Q2 で共通のアイテム集合は c と d であり、それぞれで絶対値が最小の項は-2c と d である。
# よって求める商は以下の通りとなる。
> (x/y).show
 - 2 c + d
# x%y については x-(y*Q) を計算すればよい。
> (x%y).show
 2 a b + 6 a c + 2 b d

# 除数 y に商を掛けて余りを足せば元の値 x に戻る。
> (y*(x/y)+(x%y)).show
 2 a b + 4 a c + a d - 2 b c + 3 b d
```

関連

% : 剰余演算子

* : 乗算演算子

3.8 ＜：小なり比較演算

書式

$zdd1 < zdd2 \rightarrow zdd$

説明

$zdd1$ に含まれるアイテム集合と $zdd2$ に含まれるアイテム集合を比較し、重みが $zdd2$ より小さいアイテム集合を選択する。

例

例 1: 基本例

==の例を参照のこと。

関連

==, <, <=, >, >=, ne? ：各種比較演算

3.9　<=：以下比較演算

書式

$zdd1$ <= $zdd2 \rightarrow zdd$

説明

$zdd1$ に含まれるアイテム集合と $zdd2$ に含まれるアイテム集合を比較し、重みが $zdd2$ 以下のアイテム集合を選択する。

例

例 1: 基本例

==の例を参照のこと。

関連

==, <, <=, >, >=, ne? ：各種比較演算

3.10 ＞：大なり比較演算

書式

ZDD Object $zdd1 > zdd2$

説明

zdd1 に含まれるアイテム集合と zdd2 に含まれるアイテム集合を比較し、重みが zdd2 より大きいアイテム集合を選択する。

例

例 1: 基本例

==の例を参照のこと。

関連

==, <, <=, >, >=, ne? ：各種比較演算

3.11　==：等価比較演算子

書式

$zdd1 == zdd2 \rightarrow zdd3$

説明

$zdd1$ の項と $zdd2$ の項について、一致する項 (重みとアイテム集合が同じ項) を選択し、そのアイテム集合を返す。アイテム集合のみを返し、重みは 1 となることに注意する。重みも含めて選択したい場合は、iif 関数と組み合わせることで実現可能である。

本パッケージで扱える比較演算子 (関数) は以下の通り。

- zdd1 == zdd2：等価演算子
- zdd1 >= zdd2：以上演算子
- zdd1 > zdd2：大なり演算子
- zdd1 <= zdd2：以下演算子
- zdd1 < zdd2：小なり演算子
- zdd1.ne?(zdd2)：不等関数

また、式全体の等価判定には same?(==) もしくは diff?を使う。

例

例 1: **基本例**

```
> require 'zdd'
> a=ZDD::itemset("a")
> b=ZDD::itemset("b")
> c=ZDD::itemset("c")
> x=3*a + 2*b + 2*c
> y=2*a + 2*b + 4*c
> x.show
 3 a + 2 b + 2 c
> y.show
 2 a + 2 b + 4 c
# (x,y が上記の通り定義されていたとする)

# x と y を比較し、同じ重みを持つアイテム集合を選択する。
> (x==y).show
 b

# x と y を比較し、y 以上の重みを持つアイテム集合を選択する。
> (x>=y).show
 a + b

# x と y を比較し、異なる重みを持つアイテム集合を選択する。
> (x.ne?(y)).show
 a + c

# zdd1 にあって zdd2 にない (もしくはその逆) アイテム集合の重みは 0 と考えればよい。
> z=2*a + 2*b
> z.show
 2 a + 2 b

# アイテム集合 c は z にないので項 0c を考えればよい。
> (x>z).show
```

```
 a + c
> (x.ne?(z)).show
 a + c
> (x==z).show
 b
```

関連

==, <, <=, >, >=, ne? ：各種比較演算

iif：アイテム比較による選択

same? ：式の等価比較

diff? ：式の不等価比較

3.12　>=：以上比較演算

書式

ZDD Object *zdd*1 >= *zdd*2

説明

zdd1 に含まれるアイテム集合と zdd2 に含まれるアイテム集合を比較し、重みが zdd2 以上のアイテム集合を選択する。

例

例 1: 基本例

==の例を参照のこと。

関連

==, <, <=, >, >=, ne? ：各種比較演算

3.13 constant : ZDD 定数オブジェクトの生成

書式

ZDD::constant(*weight*) → *zdd*

weight : integer

説明

weight で指定した整数を ZDD の重みオブジェクト *zdd*(空のアイテム集合の重み) に変換する。

例

例 1: 基本例

```
> require 'zdd'
> c=ZDD::constant(10)
> c.show
 10
# ZDD 重みオブジェクトと ruby 文字列との演算では、
# ruby 文字列はアイテム集合と見なされ自動で ZDD オブジェクトにキャストされる。
> (c*"a").show
 10 a

# ZDD 重みオブジェクトと ruby 整数との演算では、ruby 整数は ZDD 重みオブジェクトと見なされる。
> (0*c).show
 0

# ZDD 重みオブジェクトを ruby 整数に変換し、ruby 整数として演算する。
> puts c.to_i*10
100

# 以下のように、0 の重みを定義しておくと、そのオブジェクトとの演算においては、
# RubyString を自動的にキャストしてくれるので便利である。
> a=ZDD::constant(0)
> a+="x"
> a+="x z"
> a+="z"
> a.show
 x z + x + z
```

関連

to_i : ZDD 定数オブジェクトを Ruby 整数に変換

itemset : アイテム集合の ZDD オブジェクトの作成

3.14　cost：アイテム集合のコスト合計

書式

obj.cost → *cost*

cost：float

説明

アイテムに設定されたコスト (symbol 関数を参照のこと) を *obj* の各アイテムに代入したときの式の値を返す。

例

例 1: 基本例

```
> require 'zdd'
# シンボル a, b, c に値 1.0, 0.5, 1.8 をそれぞれ与える。
> ZDD::symbol("a",1.0)
> ZDD::symbol("b",0.5)
> ZDD::symbol("c",2.0)

> a=ZDD::itemset("a")
> b=ZDD::itemset("b")
> c=ZDD::itemset("c")

# 式は 1 つのシンボル a から構成され a=1.0
> puts a.cost
1.0

# 式"a b"に a=1.0,b=0.5 を代入すると 1.0*0.5=0.5
> f=a*b
> f.show
 a b
> puts f.cost
0.5

# 式"a b + 2 a + c + 3"に a=1.0,b=0.5,c=2.0 を代入すると 1.0*0.5+2*1.0+2.0+3=7.5
> f=a*b + 2*a + c + 3
> f.show
 a b + 2 a + c + 3
> puts f.cost
7.5
```

関連

symbol：アイテムの宣言

maxcover：コスト最大のアイテム集合の選択

maxcost：コスト最大のアイテム集合のコスト

mincover：コスト最小のアイテム集合の選択

mincost：コスト最小のアイテム集合のコスト

3.15 count : 項数計算

書式

obj.count → $numItemsets$
$numItemsets$: integer

説明

obj に格納された項の数 (アイテム集合の数) を返す。

例

例 1: **基本例**

```
> require 'zdd'
> a=ZDD::itemset("a")
> b=ZDD::itemset("b")
> f=a+b+a*b
> f.show
 a b + a + b
> puts f.count
3

> g=a+b+a*b+1
> g.show
 a b + a + b + 1
> puts g.count
4

> c=ZDD::constant(0)
> puts c.count
0

> d=ZDD::constant(1)
> puts d.count
1
```

関連

3.16　csvout：CSV ファイル出力

書式

obj.csvout($fileName$) → $self$
　$fileName$：string

説明

ZDD オブジェクト obj の内容を $fileName$ で指定したファイルに CSV 形式で出力する。重みとアイテム集合の 2 項目が出力される。アイテム集合は、アイテムをスペースで区切ることで出力される。ただし、空のアイテム集合 (重みだけの項) は、重みだけ出力され、CSV の第 2 項は null 出力となる。また、null の ZDD オブジェクト (重み 0 の空のアイテム集合) を出力すると 0 バイトファイルとなる。

項目名は出力されない。返り値は self(自分自身)。

例

例 1: 基本例

```
> require 'zdd'
> a=ZDD::itemset("a")
> b=ZDD::itemset("b")
> c=ZDD::itemset("c")
> d=ZDD::itemset("d")
> x=b*a*d + 5*b*c + 3*d + 4
> x.show
 a b d + 5 b c + 3 d + 4
> x.csvout("output.csv")
# output.csv の内容は以下の通り。
# 重み 4 の空のアイテム集合は null が出力される。
# 1,a b d
# 5,b c
# 3,d
# 4,

# null の ZDD オブジェクト（重み 0 の空のアイテム集合）は 0 バイトファイルとなる。
> y=ZDD::constant(0)
> y.csvout("null.csv")
```

関連

hashout：Hash 変換

3.17 delta：排他的論理和演算

書式

obj.delta($zdd1$) → $zdd2$

説明

ZDD オブジェクト obj に含まれるアイテム集合 α と $zdd1$ に含まれるアイテム集合 β の排他的論理和 $\alpha \oplus \beta$ を求め、その結果の ZDD オブジェクト $zdd2$ を返す。例えば、アイテム集合 abc と bcd の排他的論理和は以下の通り。

abc.delta(bcd) = abc⊕bcd = ad

複数のアイテム集合間の演算では全組合せについて排他的論理和を求める。

(abc + a).delta(bcd + b) = abc⊕bcd + abc⊕b + a⊕bcd + a⊕b
= ad + ac + abcd + ab

重みについては、同じアイテム集合を複数に展開して計算すればよい。

(2abc).delta(bcd) = (abc+abc).delta(bcd) = ad + ad = 2ad

ちなみに $\alpha \oplus \beta$ を $\alpha \cap \beta$(共通集合演算) に変更すれば delta 関数となる。

例

例 1: **基本例**

以下では、アイテム a,b,c、およびアイテム集合 2ab + a + 3b、abc + 2ab + bc + 1、ab+a に関する排他的論理和を求める。

```
> require 'zdd'
# まずはアイテム集合の定義
> a=ZDD::itemset("a")
> b=ZDD::itemset("b")
> c=ZDD::itemset("c")
> f=2*a*b+a+3*b
> f.show
 2 a b + a + 3 b
```

(2ab + a + 3b)⊕a = 3ab + 2b + 1

```
> f.delta(a).show
 3 a b + 2 b + 1
```

(2ab + a + 3b)⊕b = ab + 2a + 3

```
> f.delta(b).show
 a b + 2 a + 3
```

(2ab + a + 3b)⊕ab = 3a + b + 2

```
> f.delta(a*b).show
 3 a + b + 2
```

(2ab + a + 3b)⊕1 = 2ab+a+3b 定数 1 は空のアイテム集合なので、それとの排他的論理和を求めると元の集合のままとなる。

```
> f.delta(1).show
 2 a b + a + 3 b
```

(abc + 2ab + bc + 1)⊕(2ab + a) の項別の演算結果は以下の通りとなる。

- abc ⊕ 2ab = 2c
- 2ab ⊕ 2ab = 4
- bc ⊕ 2ab = 2ac
- 1 ⊕ 2ab = 2ab
- abc ⊕ a = bc
- 2ab ⊕ a = 2b
- bc ⊕ a = abc
- 1 ⊕ a = a

結果をまとめると a b c + 2 a b + 2 a c + a + b c + 2 b + 2 c + 4 となる。

```
> g=((a*b*c)+2*(a*b)+(b*c)+1)
> h=2*a*b + a
> g.show
 a b c + 2 a b + b c + 1
> h.show
 2 a b + a
> g.delta(h).show
 a b c + 2 a b + 2 a c + a + b c + 2 b + 2 c + 4
```

関連

3.18 density : ZDD の濃度計算

書式

obj.density → $dens$
$dens$: float

説明

濃度とは、登録されている全アイテムから構成可能なアイテム集合の総数に対する obj に格納されたアイテム集合の数の割合と定義される。

例

例 1: **基本例**

```
> require 'zdd'
> a=ZDD::itemset("a")
> b=ZDD::itemset("b")
> c=ZDD::itemset("c")

# a,b,c の 3 アイテムを使った全組合せ数は 8 通り。
# 以下で、f にはそのすべての組合せが格納されているので濃度は 1.0 となる。
> f=(a+1)*(b+1)*(c+1)
> f.show
 a b c + a b + a c + a + b c + b + c + 1
> puts f.density
1.0

# 以下で、f には 1 通りの組合せ (a b) が格納されているので濃度は 1/8=0.125 となる。
> f=a*b
> f.show
 a b

> puts f.density
0.125

# 以下で、f には 3 通りの組合せ (a b,a,b) が格納されているので濃度は 3/8=0.375 となる。
> f+=a
> f+=b
> f.show
 a b + a + b
> puts f.density
0.375
```

関連

3.19　diff? : 式の不等価比較

書式

obj.diff?(*zdd*) → *bool*

説明

2 つの ZDD オブジェクト *obj* と *zdd* を比較し、同じなら false、異なるなら true を返す

例

例 1: 基本例

```
> require 'zdd'
> a=ZDD::itemset("a")
> b=ZDD::itemset("b")
> c=ZDD::itemset("c")
> puts a.diff?(b)
true
> puts a.diff?(a)
false
> puts (a+b).diff?(a+c)
true
> puts (a+b).diff?(a+b)
false
```

関連

same? : 式の等価比較

3.20 each：アイテム集合の繰り返し

書式

obj.each{|$item$|...} → $Qtrue$

説明

ZDD オブジェクト obj からアイテム集合を 1 つずつ $item$ にセットし，指定されたブロックを実行する．

例

例 1: **基本例**

```
> require 'zdd'
> a=ZDD::itemset("a")
> b=ZDD::itemset("b")
> c=ZDD::itemset("c")
> f=2*a+2*b+4*a*c
> f.show
 4 a c + 2 a + 2 b

> f.each{|x|
>   x.show
> }
 4 a c
 2 a
 2 b
```

関連

each_item：アイテムの繰り返し

3.21　each_item：アイテムの繰り返し

書式

obj.each_item{|$weight, item, top, bottom$| ...} → $Qtrue$

説明

ZDD オブジェクト obj からアイテム集合を 1 つずつ読み込み、重みとアイテムを $weight$ と $item$ にそれぞれセットし，指定されたブロックを実行する．セットされたアイテムが処理中のアイテム集合の最初のアイテムであれば top が true に、最後であれば $bottom$ が false にセットされる。

例

例 1: 基本例

```
> require 'zdd'
> a=ZDD::itemset("a")
> b=ZDD::itemset("b")
> c=ZDD::itemset("c")
> f=2*a+2*b+4*a*c
> f.each_item{|weight,item,top,bottom|
>   puts weight
>   item.show
>   puts top
>   puts bottom
>   puts "----------"
> }
4
 a
true
false
----------
4
 c
false
true
----------
2
 a
true
true
----------
2
 b
true
true
----------
```

関連

each：アイテム集合の繰り返し

3.22 export：ZDD のシリアライズ出力

書式

obj.export($fileName$)
$fileName$：string

説明

obj の ZDD 内部構造をテキストでシリアライズ出力する。$fileName$ を指定すれば、そのファイルに出力する。省略すれば標準出力に出力する。

symbol 関数によって宣言された内容、およびアイテム順序はシリアライズされないので、import 関数によって ZDD オブジェクトを復元するとき、アイテムの宣言を再度同じように実行する必要があることに注意する。具体例は import 関数の例を参照のこと。

例

例 1: 基本例

```
> require 'zdd'
> a=ZDD::itemset("a")
> b=ZDD::itemset("b")
> c=ZDD::itemset("c")
> f=5*a*b*c+3*a*b+2*b*c+c
> f.show
 5 a b c + 3 a b + 2 b c + c

> f.export

_i 3
_o 3
_n 7
4 1 F T
248 2 F 5
276 3 4 248
232 2 F 4
2 2 F T
272 3 232 2
268 3 232 248
276
272
268
```

関連

import：ZDD のシリアライズ入力
csvout：CSV ファイル出力
hashout：Hash 変換

3.23　freqpatA：頻出アイテム集合の列挙

書式

obj.freqpatA(*minsup*) → *zdd*
　minsup : integer

説明

ZDD オブジェクト *obj* から最小サポート *minsup* 以上出現する頻出アイテム集合を全て列挙し、その ZDD オブジェクト *zdd* を返す。

同様の関数に lcm があるが、これはファイルからトランザクションデータを読み込み LCM アルゴリズムにより頻出アイテム集合を列挙するため、本関数に比べて非常に高速である。しかしながら、アイテム名として整数を用いる必要があり、またアイテム順序についても注意が必要であるため、さほど効率性が求められないのであれば、本関数を使えばよい。

例

例 1: 基本例

```
> require 'zdd'
> a=ZDD::itemset("a")
> b=ZDD::itemset("b")
> c=ZDD::itemset("c")
> d=ZDD::itemset("d")
> t=a*b*c + a*b + a + b*c*d + a
> t.show
 a b c + a b + 2 a + b c d

# t=a*b*c + a*b + a + b*c*d + a において、
# アイテム集合 ab は第 1,2 項の 2 回出現している。
# アイテム集合 a は第 1,2,3,5 項の 4 回出現している。
> t.freqpatA(2).show
 a b + a + b c + b + c + 1

# 極大アイテム集合（他のアイテム集合に包含されないアイテム集合）
# 上記頻出アイテム集合のうち、a は ab に包含され、b と c は bc に包含され、
# 1 は他の全てのアイテム集合に包含されていると考えるため、a,b,1 は出力されない。
> t.freqpatM(2).show
 a b + b c

# 飽和アイテム集合（出現集合が同じアイテム集合族の中で極大のアイテム集合）
# 上記頻出アイテム集合のうち、bc と c のいずれも第 1,4 項に出現している。
# すなわち出現集合が同じアイテム集合族である。
# そしてそのなかで極大な bc のみが出力される。
# その他に列挙された飽和集合は全て異なる出現集合を持つ。
> t.freqpatC(2).show
 a b + a + b c + b + 1

# 最小サポートを 3 にして実行する。
> t.freqpatA(3).show
 a + b + 1
> t.freqpatM(3).show
 a + b
> t.freqpatC(3).show
 a + b + 1
```

関連

freqpatM：頻出極大アイテム集合の列挙
freqpatC：頻出飽和アイテム集合の列挙
lcm：LCM over ZDD

3.24　freqpatC：頻出飽和アイテム集合の列挙

書式

obj.freqpatC(*minsup*) → *zdd*
　minsup：integer

説明

ZDD オブジェクト *zdd* から最小サポート *minsup* 以上出現する頻出飽和アイテム集合を全て列挙し、その ZDD オブジェクト *zdd* を返す。

例

例 1: 基本例
freqpatA の例を参照のこと。

関連

freqpatA：頻出アイテム集合の列挙
freqpatM：頻出極大アイテム集合の列挙
lcm：LCM over ZDD

3.25 freqpatM：頻出極大アイテム集合の列挙

書式

ZDD Object *obj*.freqpatM(*minsup*) → *zdd*
minsup：Ruby Integer

説明

ZDD オブジェクト *obj* から最小サポート *minsup* 以上出現する頻出極大アイテム集合を全て列挙し、その ZDD オブジェクト *zdd* を返す。

例

例 1: 基本例
freqpatA の例を参照のこと。

関連

freqpatA：頻出アイテム集合の列挙
freqpatC：頻出飽和アイテム集合の列挙
lcm：LCM over ZDD

3.26　hashout：Hash 出力

書式

obj.hashout → *hash*

説明

zdd の内容を ruby の Hash オブジェクト *hash* に変換し、そのオブジェクトを返す。ハッシュキーはアイテム集合、対応する値は重み。ただし、出力できるアイテム集合数には上限があり、4,096,000 アイテム集合を超えると、そこで強制的に出力を停止する。途中で停止したかどうかは、partly メソッドを使うことで確認できる。

例

例 1: 基本例

```
> require 'zdd'
> a=ZDD::itemset("a")
> b=ZDD::itemset("b")
> c=ZDD::itemset("c")
> d=ZDD::itemset("d")
> f=a*b + 2*b*c + 3*d + 4
> h=f.hashout
> p h
{"a b"=>1, "b c"=>2, "d"=>3, ""=>4}
```

関連

partly：部分 Hash 出力フラグ
to_a：アイテム集合配列への変換
csvout：CSV ファイル出力
export：ZDD のシリアライズ出力

3.27 iif : アイテム比較による選択

書式

obj.iif(zdd1,zdd2) → *zdd*3

説明

*zdd*1 から *obj* に含まれるアイテム集合の項を選択し、*zdd*2 から *obj* に含まれないアイテム集合の項を選択し、それら選択された項で構成される ZDD オブジェクト *zdd*3 を返す。*obj* に含まれるアイテム集合の重みは動作に影響しない。

例

例 1: **基本例**

```
> require 'zdd'
> a=ZDD::itemset("a")
> b=ZDD::itemset("b")
> c=ZDD::itemset("c")
> d=ZDD::itemset("d")

# iif の第 1 引数の各項から、(a+b) に含まれるアイテム集合 a,b の項 2a,3b を選択し、
# iif の第 2 引数の各項から、(a+b) に含まれるアイテム集合 a,b の項を除外した項 8c,9d を選択する。
> f=(a+b).iif(2*a+3*b+4*c+5*d,6*a+7*b+8*c+9*d)
> f.show
 2 a + 3 b + 8 c + 9 d

# 典型的には比較演算子と組み合わせて以下のように利用する。
> x=3*a+2*b+2*c
> y=2*a+2*b+4*c
> x.show
 3 a + 2 b + 2 c
> y.show
 2 a + 2 b + 4 c

# x と y を比較し、y より大きい重みを持つ項を x から、それ以外を y から選ぶ。
# x>y の結果は a であり、第 1 引数 x から 3a が選択され、
# その他のアイテム集合は第 2 引数 y から 2b,4c が選ばれる。
> r1=(x>y).iif(x,y)
> r1.show
 3 a + 2 b + 4 c

# x と y を比較し、y より大きい重みを持つ項を x から選ぶ。\\
# 上の例と同様であるが、第 2 引数が 0 なので a 以外のアイテム集合は何も選択されない。\\
> r2=(x>y).iif(x,0)
> r2.show
 3 a

# x と y を比較し、y と同じ重みを持つ項を x から選ぶ。
> r3=(x==y).iif(x,0)
> r3.show
 2 b
```

関連

==, <, <=, >, >=, ne? : 各種比較演算

3.28　import：ZDD のインポート

書式

obj.import($fileName$) → zdd
　$fileName$: string

説明

export メソッドでシリアライズ出力された ZDD ファイル (fileName で指定) をインポートし ZDD オブジェクト zdd を復元する。export する時の symbol によるアイテムの宣言順序と import する時の宣言順序は同じでなければならない。

例

例 1: **基本例**

```
> require 'zdd'
> a=ZDD::itemset("a")
> b=ZDD::itemset("b")
> c=ZDD::itemset("c")
> f=5*a*b*c+3*a*b+2*b*c+c
> f.show
 5 a b c + 3 a b + 2 b c + c
> f.export("dat.zdd")
# エキスポートされたファイル dat.zdd の内容は以下の通り。
# _i 3
# _o 3
# _n 7
# 4 1 F T
# 248 2 F 5
# 276 3 4 248
# 232 2 F 4
# 2 2 F T
# 272 3 232 2
# 268 3 232 248
# 276
# 272
# 268
```

例 2: **正しい復元例**

```
> require 'zdd'
# 以下のように symbol を順番に宣言した後に import すれば正しく復元される。
> ZDD::symbol("a")
> ZDD::symbol("b")
> ZDD::symbol("c")
> g1=ZDD::import("dat.zdd")
> g1.show
 5 a b c + 3 a b + 2 b c + c
```

例 3: **正しくない復元例**

```
> require 'zdd'
# もしアイテム b,c の宣言順序を入れ替えると結果においても b と c が入れ替わってしまう。
> ZDD::symbol("a")
```

```
> ZDD::symbol("c")
> ZDD::symbol("b")
> g2=ZDD::import("dat.zdd")
> g2.show
 5 a c b + 3 a c + 2 c b + b
```

例 4: symbol **宣言なしでの復元例**

```
> require 'zdd'
# 宣言せずにインポートすると、x1,x2,x3 のようなアイテム名が使われる。
# この時、各アイテムの後ろに付いた数字は、アイテムの宣言の逆順による連番となる。
# 以下の例では、x1=c, x2=b, x3=c である。
> g3=ZDD::import("dat.zdd")
> g3.show
 5 x3 x2 x1 + 3 x3 x2 + 2 x2 x1 + x1
```

関連

export：ZDD のシリアライズ出力

3.29　items：ZDD を構成する全アイテムの重み集計

書式

obj.items → *zdd*

説明

ZDD オブジェクト *obj* のアイテム別に重みを集計し、新しい ZDD オブジェクト *zdd* を生成し返す。

例

例 1: **基本例**

```
> require 'zdd'
> a=ZDD::itemset("a")
> b=ZDD::itemset("b")
> c=ZDD::itemset("c")
> f=((a*b*c)+(a*b)+(b*c))
> f.show
 a b c + a b + b c

# ZDD オブジェクト f は 3 つのアイテム a,b,c から構成されており、
# それぞれのアイテムの重みを以下の通り計算する。
# アイテム a を含む項は"a b c"と"2 a b"で、その重み合計は 3 となる。
# アイテム b は全ての項に含まれ、その重み合計は 4 となる。
# アイテム c を含む項は"a b c"と"b c"で、その重み合計は 2 となる。
> f.items.show
 2 a + 3 b + 2 c
```

関連

3.30 itemset : アイテム集合の ZDD オブジェクトの作成

書式

obj.itemset(*is*) → *zdd*
is : string

説明

is で指定されたアイテム集合の ZDD オブジェクト *zdd* を生成する。複数のアイテムは半角スペースで区切る。重みを指定することはできない。空文字 (””) が指定された場合は空のアイテム集合と扱われ、積和表現の定数項 1 として設定される (zdd.constant(1) と同様)。

symbol メソッドで宣言されていないアイテムは、アイテムオーダ表の最後に追加される。またアイテムに設定されるコストはデフォルトの 0.5 となる。コストについての詳細は symbol を参照のこと。

例

例 1: **基本例**

```
> require 'zdd'
> a=ZDD::itemset("a b")
> a.show
 a b
> b=ZDD::itemset("b")
> b.show
 b
> c=ZDD::itemset("")
> c.show
 1

# 数字をアイテム名として利用することも可能
> x0=ZDD::itemset("2")
> x0.show
 2

# ただし、表示上重みと区別がつかなくなるので注意が必要。
> (2*x0).show
 2 2

# こんな記号ばかりのアイテム名も OK。
> x1=ZDD::itemset("!#%&'()=~|@[;:]")
> x1.show
 !#%&'()=~|@[;:]

# ただし、ruby で特殊な意味を持つ記号はエスケープする必要がある。
# 以下では、\,$,"の 3 つの文字をエスケープしている例である。
> x2=ZDD::itemset("\\\$\"")
> x2.show
 \$"

# もちろん日本語も利用可。
> x3=ZDD::itemset("りんご ばなな")
> x3.show
 りんご ばなな
```

関連

symbol：アイテムの宣言
constant：ZDD 定数オブジェクトの生成
cost：アイテム集合のコスト合計

3.31 lcm : LCM over ZDD

書式

obj.lcm($type, transaction, minsup[, order, ub]$) → zdd

$transaction$: string

$type$: string

$minsup$: integer

$order$: string

ub : integer

説明

$transaction$ で指定されたトランザクションファイルから、LCM over zdd アルゴリズムを利用し、指定された最小サポート $minsup$ 以上の頻出パターンを列挙しその ZDD オブジェクト zdd を返す。$order$ で ZDD のアイテムオーダファイルを指定し、ub で列挙するアイテム集合のサイズの上限を与える。

列挙する頻出パターンの種別は $type$ で与え、"F"(頻出アイテム集合) ,"M"(極大アイテム集合),"C"(飽和アイテム集合) の三種類が指定できる。また"FQ"のように、"Q"を付けると、列挙された各頻出アイテム集合に頻度を重みとして出力する。"Q"をつけなければ、最小サポート条件を満たす頻出アイテム集合のみ出力され、頻度情報は省かれる。

トランザクションファイルは、以下に示すようなテキストファイルで、一つの行が一つのトランザクションに対応しており、アイテムは 1 から始まる連番で指定し、アイテムの区切りには半角スペースを用いる。アイテムとしてアルファベットを利用することはできない。

```
1 2 3 6
4 5 6
1 2 4 6
2 4 6
1 2 4 5
```

$order$ ファイルは、ZDD のアイテムオーダ表に登録するアイテムの順序を示したテキストファイルである。通常は、以下のように、トランザクションデータに含まれる全アイテムを番号順に並べて与える。またトランザクションのアイテム番号に欠番があった場合でも、その欠番を含めて指定する必要があることに注意する。

```
1 2 3 4 5 6
```

$order$ ファイルは省略する (もしくは nil を指定する) ことができるが、その場合、効率性のためにアイテムオーダは LCM の内部アルゴリズムによって決まる。しかし、この方法を利用すると、そのオーダに応じてアイテム番号が再び採番されるため、出力される ZDD の頻出アイテム集合におけるアイテム番号は、元のトランザクションの番号とは異なったものとなる。アイテムの内容に関係のない解析をするのであれば、$order$ を省略することで計算効率は高まるが、逆に、アイテムの内容についての意味を解析する目的があるのであれば、上記に示した連番としてのオーダファイルを指定する。

ub には列挙される頻出アイテム集合のサイズの上限を指定する。指定を省略するか、nil を与えると上限なしに列挙する。

例

例 1: **基本例**

```
> require 'zdd'
# tra.txt の内容
```

```
# 1 2 3 6
# 4 5 6
# 1 2 4 6
# 2 4 6
# 1 2 4 5
# order.txt の内容
# 1 2 3 4 5 6
> p1=ZDD::lcm("FQ","tra.txt",3,"order.txt")
> p1.show
 3 x6 x4 + 3 x6 x2 + 4 x6 + 3 x4 x2 + 4 x4 + 3 x2 x1 + 4 x2 + 3 x1 + 5

# オーダファイルを省略した場合、得られる頻出アイテム集合は同じであるが
# そのアイテム番号がトランザクションファイルのアイテム番号と異なったものとなることに注意する。
> p2=ZDD::lcm("FQ","tra.txt",3)
> p2.show
 3 x4 x1 + 3 x4 + 3 x3 x2 + 3 x3 x1 + 4 x3 + 3 x2 x1 + 4 x2 + 4 x1 + 5
```

関連

freqpatA：頻出アイテム集合の列挙

freqpatM：頻出極大アイテム集合の列挙

freqpatC：頻出飽和アイテム集合の列挙

3.32 maxcost：コスト最大のアイテム集合のコスト値を返す

書式

obj.maxcost → *cost*
cost：float

説明

obj 含まれるアイテム集合の中で、最大コストを返す。そのコストの値を持つアイテム集合を得るには maxcover を用いる。コストの設定は symbol 関数で行う。

例

例 1: 基本例
maxcover メソッドの例を参照のこと。

関連

symbol：アイテムの宣言
cost：アイテム集合のコスト合計
maxcover：コスト最大のアイテム集合の選択
mincover：コスト最小のアイテム集合の選択
mincost：コスト最小のアイテム集合のコスト

3.33　maxcover：コスト最大のアイテム集合の表示

書式

obj.maxcover → *zdd*

説明

obj 含まれるアイテム集合の中で、コストが最大となるアイテム集合を ZDD オブジェクト *zdd* として返す。コストの計算に重みは考慮されないことに注意する。

コストの値を得るには maxcost を用いる。コストの設定は symbol 関数で行う。

例

例 1: **基本例**

```
> require 'zdd'
# アイテムごとにコストを設定する。
> ZDD::symbol("a",1.0)
> ZDD::symbol("b",0.5)
> ZDD::symbol("c",2.0)

> a=ZDD::itemset("a")
> b=ZDD::itemset("b")
> c=ZDD::itemset("c")

> f=a*b + b*c + c*a
> f.show
 a b + a c + b c
# a b のコスト=1.0+0.5=1.5
# b c のコスト=0.5+2.0=2.5
# a c のコスト=1.0+2.0=3.0
> puts f.maxcover
a c
> puts f.maxcost
3.0
> puts f.mincover
a b
> puts f.mincost
1.5
```

関連

symbol：アイテムの宣言

cost：アイテム集合のコスト合計

maxcost：コスト最大のアイテム集合のコスト

mincover：コスト最小のアイテム集合の選択

mincost：コスト最小のアイテム集合のコスト

3.34 maxweight：重みの最大値

書式

obj.maxweight → *zdd*

説明

ZDD オブジェクト *obj* に含まれる項 (定数項も含む) のうち、最大の重みを ZDD 定数オブジェクトで返す。

例

例 1: **基本例**

```
> require 'zdd'
> a=ZDD::itemset("a")
> b=ZDD::itemset("b")
> c=ZDD::itemset("c")
> x=5*a+3*b+c
> x.show
 5 a + 3 b + c
> x.maxweight.show
 5

# 最大値は定数項も含めて求められる。
> x=5*a+3*b+c+10
> x.show
 5 a + 3 b + c + 10
> x.maxweight.show
 10

# 最大の重みを持つ項を選択する。
> x=5*a+3*b+5*c+2
> x.show
 5 a + 3 b + 5 c + 2
> x.termsEQ(x.maxweight).show
 5 a + 5 c
```

関連

minweight：重みの最小値
totalweight：重みの合計

3.35　meet：共通集合演算

書式

obj.meet($zdd1$) → $zdd2$

説明

obj に含まれるアイテム集合 α と $zdd1$ に含まれるアイテム集合 β の共通集合 $\alpha \cap \beta$ を求め、その ZDD オブジェクト $zdd2$ を返す。

例えば、アイテム集合 abc と bcd の共通集合は以下の通り。

$$\mathrm{abc.meet(bcd)} = \mathrm{abc} \cap \mathrm{bcd} = \mathrm{bc}$$

複数のアイテム集合間の演算では全組合せについて共通集合を求める。

$$\begin{aligned}(\mathrm{abc} + \mathrm{a}).\mathrm{meet}(\mathrm{bcd} + \mathrm{b}) &= \mathrm{abc} \cap \mathrm{bcd} + \mathrm{abc} \cap \mathrm{b} + \mathrm{a} \cap \mathrm{bcd} + \mathrm{a} \cap \mathrm{b} \\ &= \mathrm{bc} + \mathrm{b} + 1 + 1 \\ &= \mathrm{bc} + \mathrm{b} + 2\end{aligned}$$

重みについては、同じアイテム集合を複数に展開して計算すればよい。

ちなみに $\alpha \cap \beta$ を $\alpha \oplus \beta$(排他的論理和演算) に変更すれば delta 関数となる。

$$(2\mathrm{abc}).\mathrm{meet}(\mathrm{bcd}) = (\mathrm{abc}+\mathrm{abc}).\mathrm{meet}(\mathrm{bcd}) = \mathrm{bc} + \mathrm{bc} = 2\mathrm{bc}$$

例

例 1: **基本例**

```
> require 'zdd'
> a=ZDD::itemset("a")
> b=ZDD::itemset("b")
> c=ZDD::itemset("c")
> f=a+2*a*b+3*b

# a + 2ab + 3b の各アイテム集合と引き数で指定されたアイテム集合 a との共通集合を求めると、
# a + 2a + 3 = 3 a + 3 となる。
> f.meet(a).show
 3 a + 3

# アイテム集合 b との共通集合を求めると 1 + 2b + 3b = 5b + 1 となる。
> f.meet(b).show
 5 b + 1

# アイテム集合 ab との共通集合を求めると a + 2ab + 3b となる。
> f.meet(a*b).show
 2 a b + a + 3 b

# 定数 1 は空のアイテム集合なので、それとの共通集合を求めると係数だけが残り 1 + 2 + 3 = 6 となる。
> f.meet(1).show
 6

# abc + 2ab + bc + 1 の各アイテム集合と引き数で指定された 2ab + a の各アイテム集合との共通集合を
# 求めると、以下の通りとなる（アイテム間のスペースは省略）。
# abc ∩ 2ab = 2ab
```

```
# 2ab ∩ 2ab = 4ab
# bc  ∩ 2ab = 2b
# 1   ∩ 2ab = 2
# abc ∩ a   = a
# 2ab ∩ a   = 2a
# bc  ∩ a   = 1
# 1   ∩ a   = 1
# 結果をまとめると 6ab + 3a + 2b + 4 となる。
#
> f=((a*b*c)+2*(a*b)+(b*c)+1)
> g=2*a*b + a
> f.show
 a b c + 2 a b + b c + 1
> g.show
 2 a b + a
> f.meet(g).show
 6 a b + 3 a + 2 b + 4
```

関連

delta : delta 演算

3.36　mincost：コスト最小のアイテム集合のコスト

書式

obj.mincost → *cost*
　cost：float

説明

obj 含まれるアイテム集合の中で、最小コストを返す。そのコストの値を持つアイテム集合を得るには mincover を用いる。コストの設定は symbol 関数で行う。

例

例 1: 基本例
maxcover メソッドの例を参照のこと。
maxcover メソッドを参照のこと。

関連

symbol：アイテムの宣言
cost：アイテム集合のコスト合計
maxcover：コスト最大のアイテム集合の選択
maxcost：コスト最大のアイテム集合のコスト
mincover：コスト最小のアイテム集合の選択

3.37 mincover : コスト最小のアイテム集合の選択

書式

obj.mincover → $self$

説明

obj 含まれるアイテム集合の中で、コストが最大となるアイテム集合を表示する。そのコストの値を得るには mincost を用いる。コストの設定は symbol 関数で行う。

例

例 1: 基本例

maxcover メソッドの例を参照のこと。

関連

symbol : アイテムの宣言
cost : アイテム集合のコスト合計
maxcover : コスト最大のアイテム集合の選択
maxcost : コスト最大のアイテム集合のコスト
mincost : コスト最小のアイテム集合のコスト

3.38　minweight：重みの最小値

書式

obj.minweight → *zdd*

説明

obj に含まれる項 (定数項も含む) のうち、最小の重みを ZDD 定数オブジェクト *zdd* で返す。

例

例 1: **基本例**

```
> require 'zdd'
> a=ZDD::itemset("a")
> b=ZDD::itemset("b")
> c=ZDD::itemset("c")
> x=5*a-3*b+c
> x.show
 5 a - 3 b + c
> x.minweight.show
 - 3

# 最大値は定数項も含めて求められる。
> x=5*a-3*b+c-10
> x.show
 5 a - 3 b + c - 10
> x.minweight.show
 - 10

# 最大の重みを持つ項を選択する。
> x=5*a-3*b+5*c-3
> x.show
 5 a - 3 b + 5 c - 3
> x.termsEQ(x.minweight).show
 - 3 b - 3
```

関連

maxweight：重みの最大値
totalweight：重みの合計

3.39 ne? : 不等比較演算

書式

obj.ne?($zdd1$) → $zdd2$

説明

obj に含まれるアイテム集合と $zdd1$ に含まれるアイテム集合を比較し、重みが異なるアイテム集合を選択する。

例

例 1: 基本例

==の例を参照のこと。

関連

==, <, <=, >, >=, ne? : 各種比較演算

3.40　partly：部分 hash 出力フラグ

書式

obj.partly → *bool*

説明

hashout メソッドにて全データをセット出来なかった場合、このメソッドは true を返す。詳細は hashout メソッドを参照のこと。

例

例 1: 基本例

```
> require 'zdd'
> require 'zdd'
> a=ZDD::itemset("a")
> b=ZDD::itemset("b")
> c=ZDD::itemset("c")
> d=ZDD::itemset("d")
> f=a*b + 2*b*c + 3*d + 4
> h=f.hashout
> puts f.partly
false
```

関連

hashout：Hash 出力

3.41 permit：部分集合の選択

書式

obj.permit($zdd1$) → $zdd2$

説明

obj に含まれるアイテム集合について、$zdd1$ 中の少なくとも 1 つのアイテム集合に包含されていれば、その項を選択する。より正確には、obj を構成するある項 T_i から重みを省いたアイテム集合を α_i、同じく $zdd1$ のアイテム集合を β_j とすると、$\alpha_i \subseteq \beta_j$ を満たすような j が少なくとも一つあれば、α_i に対応する項 T_i を obj から選択する。ちなみに、条件式 $\alpha_i \subseteq \beta_j$ を $\alpha_i \supseteq \beta_j$ に変えれば restrict 関数となる。

例

例 1: 基本例

```
> require 'zdd'
> a=ZDD::itemset("a")
> b=ZDD::itemset("b")
> c=ZDD::itemset("c")
> d=ZDD::itemset("d")
> x=5*a + 3*b + b*c + 2
> y=a + b + d
> z=a*c
> x.show
 5 a + b c + 3 b + 2
> y.show
 a + b + d
> z.show
 a c

# x に含まれる 3 つのアイテム集合 a,bc,b, Φ（重み 2 の項で空のアイテム集合）のうち、
# y の 3 つのアイテム集合 a,b,d のいずれかに包含されるアイテム集合は
# a と b と Φ（空のアイテム集合はいずれのアイテム集合にも含まれると考える）。
# よって、x から a,b, Φの項が選択される。
> x.permit(y).show
 5 a + 3 b + 2

# x に含まれる 4 つのアイテム集合 a,bc,b, Φのうち、z のアイテム集合 ac に含まれるアイテム集合は a と Φ。
# よって、x から a と Φの項が選択される。
> x.permit(z).show
 5 a + 2

# x に含まれる 3 つのアイテム集合 a,bc,b, Φのうち、アイテム集合 c に含まれるアイテム集合は Φのみ。
# よって、x から Φの項が選択される。
> x.permit(c).show
 2
```

関連

restrict：上位集合の選択

3.42　permitsym：アイテム数によるアイテム集合の選択

書式

obj.permitsym($zdd1$) → $zdd2$

説明

ZDD オブジェクト obj を構成するアイテム集合のうち、$zdd1$ で示された個数以下のアイテムを含むアイテム集合を選択し、その ZDD オブジェクト $zdd2$ を返す。

例

例 1: **基本例**

```
> require 'zdd'
> a=ZDD::itemset("a")
> b=ZDD::itemset("b")
> c=ZDD::itemset("c")
> d=ZDD::itemset("d")
> x=5*a + 3*b + b*c + 2
> x.show
 5 a + b c + 3 b + 2

# アイテムが 1 つ以下のアイテム集合を選択
> x.permitsym(1).show
 5 a + 3 b + 2

# アイテムが 2 つ以下のアイテム集合を選択
> x.permitsym(2).show
 5 a + b c + 3 b + 2

# アイテムのないアイテム集合（すなわち空アイテム集合）を選択
> x.permitsym(0).show
 2
```

関連

3.43 restrict : 上位集合の選択

書式

obj.restrict($zdd1$) → $zdd2$

説明

obj に含まれるアイテム集合について、$zdd1$ 中の少なくとも 1 つのアイテム集合を包含していれば、その項を選択する。より正確には、obj を構成するある項 T_i から重みを省いたアイテム集合を α_i、同じく $zdd1$ のアイテム集合を β_j とすると、$\alpha_i \supseteq \beta_j$ を満たすような j が少なくとも一つあれば、α_i に対応する項 T_i を obj から選択する。ちなみに、条件式 $\alpha_i \supseteq \beta_j$ を $\alpha_i \subseteq \beta_j$ に変えれば permit 関数となる。

例

例 1: **基本例**

```
> require 'zdd'
> a=ZDD::itemset("a")
> b=ZDD::itemset("b")
> c=ZDD::itemset("c")
> x=5*a + b*c + 3*b + 2
> x.show
 5 a + b c + 3 b + 2

# x に含まれる 4 つのアイテム集合 a,bc,b, Φ（重み 2 の項で空のアイテム集合）のうち、
# y の 2 つのアイテム集合 a,b のいずれかを包含するアイテム集合は、a,bc である。
# よって x から a,bc の項が選択される。
> x.restrict(a+c).show
 5 a + b c

# x に含まれる 4 つのアイテム集合 a,bc,b, Φのうち、
# z のアイテム集合 bc を包含するアイテム集合は、bc のみ。
# よって x から bc の項が選択される。
> x.restrict(b*c).show
 b c

# x に含まれる 4 つのアイテム集合 a,bc,b, Φのうち、
# アイテム集合Φ（重み 1 の空アイテム集合）を含むアイテム集合は全てのアイテム集合。
> x.restrict(1).show
 5 a + b c + 3 b + 2

# x に含まれる 4 つのアイテム集合 a,bc,b, Φのうち、アイテム集合 abc を含むアイテム集合はない。
> x.restrict(a*b*c).show
 0
```

関連

permit : 部分集合の選択

3.44　same? : 式の等価比較

書式

obj.same?(zdd) → $bool$ → $bool$
obj === zdd → $bool$

説明

2 つの ZDD オブジェクト obj と zdd を比較し、同じなら true、異なるなら false を返す。”=”が二つの演算子 (== 演算子) は別に定義されており、式内の項の等価比較を行うものであり、本演算子とは全く異なることに注意する。

例

例 1: 基本例

```
> require 'zdd'
> require 'zdd'
> a=ZDD::itemset("a")
> b=ZDD::itemset("b")
> c=ZDD::itemset("c")
> puts a.same?(b)
false
> puts a.same?(a)
true
> puts (a+b).same?(a+c)
false
> puts (a+b).same?(a+b)
true
> puts (a-a).same?(0)
true
> puts (2*a/a)===2
true
```

関連

diff? : 式の不等価比較

3.45 show : ZDD の表示

書式

obj.show([*switch*])
switch : string

説明

ZDD オブジェクト *obj* を多様な形式で標準出力に出力する。出力形式は、表 3.1 に示された値 (文字列) を与えることによって切り替える。*switch* を省略すればアイテムの積和形式で表示する。

表 3.1 ZDD オブジェクトの表示形式のスイッチ一覧

switch の値	機能
(スイッチ無し)	アイテムの積和形での表示
bit	重みの (-2) 進数の各桁別アイテム集合の表示
hex	整数値を 16 進数で表現する積和形表示
map	カルノー図で表示。アイテム変数 6 個まで表示できる
rmap	カルノー図で表示。冗長なアイテム変数は省いて表示
case	整数値ごとに場合分けして積和形表示
decomp	単純直交分解形式での出力

例

例 1: **基本例**

```
> require 'zdd'
> a=ZDD::itemset("a")
> b=ZDD::itemset("b")
> c=ZDD::itemset("c")
> f=5*a*b*c - 3*a*b + 2*b*c + c
> f.show
 5 a b c - 3 a b + 2 b c + c
> ZDD::constant(0).show
 0
```

例 2: bit

```
> require 'zdd'
> a=ZDD::itemset("a")
> b=ZDD::itemset("b")
> c=ZDD::itemset("c")
> f=5*a*b*c - 3*a*b + 2*b*c + c
> f.show
 5 a b c - 3 a b + 2 b c + c

> f.bit
NoMethodError: undefined method `bit' for 5 a b c +  - 3 a b + 2 b c + c:Module
        from (irb):8
        from /Users/hamuro/.rvm/rubies/ruby-1.9.3-p362/bin/irb:16:in `<main>'

# "a b c"の重み 5 の (-2) 進数は 101 となる。
# 1*(-2)^2+0*(-2)^1+1*(-2)^0 = 5
# よって 0 桁目と 2 桁目にアイテム集合"a b c"が表示されている。
# "a b"の重み-3 の (-2) 進数は 1101 となる。
```

```
# 1*(-2)^3+1*(-2)^2+0*(-2)^1+1*(-2)^0 = -3
# よって 0,2,3 桁目にアイテム集合"a b"が表示されている。
```

例 3: hex

```
> require 'zdd'
> a=ZDD::itemset("a")
> b=ZDD::itemset("b")
> c=ZDD::itemset("c")
> d=ZDD::itemset("d")

> f=a*b+11*b*c+30*d+4
> f.show
 a b + 11 b c + 30 d + 4
> f.hex
NoMethodError: undefined method `hex' for a b + 11 b c + 30 d + 4 :Module
        from (irb):9
        from /Users/hamuro/.rvm/rubies/ruby-1.9.3-p362/bin/irb:16:in `<main>'
```

例 4: map

```
> require 'zdd'
> a=ZDD::itemset("a")
> b=ZDD::itemset("b")
> c=ZDD::itemset("c")
> d=ZDD::itemset("d")
> f=2*a*b+3*b+4
> f.show
 2 a b + 3 b + 4
> f.map
NoMethodError: undefined method `map' for 2 a b + 3 b + 4 :Module
        from (irb):8
        from /Users/hamuro/.rvm/rubies/ruby-1.9.3-p362/bin/irb:16:in `<main>'
# アイテム a が１列目のビット列に、アイテム b が１行目のビット列に対応してアイテム集合が表現されている。
# セルの値は重みを表す。左上のセルは a が 0、b が 0、すなわち定数項が 4 であることが示されている。

# 4 アイテムでは以下の通り。
> g=a*b + 2*b*c + 3*d + 4
> g.show
 a b + 2 b c + 3 d + 4
> g.map
NoMethodError: undefined method `map' for a b + 2 b c + 3 d + 4 :Module
        from (irb):15
        from /Users/hamuro/.rvm/rubies/ruby-1.9.3-p362/bin/irb:16:in `<main>'
```

例 5: rmap

```
> require 'zdd'
# 4 つのアイテム a,b,c,d を宣言
> ZDD::symbol("a")
> ZDD::symbol("b")
> ZDD::symbol("c")
> ZDD::symbol("d")

> f=ZDD::itemset("a b") + 2*ZDD::itemset("b c") + 4
> f.show
 a b + 2 b c + 4

# map で表示させると以下の通り。
```

```
> f.map
NoMethodError: undefined method ‘map’ for a b + 2 b c + 4 :Module
        from (irb):12
        from /Users/hamuro/.rvm/rubies/ruby-1.9.3-p362/bin/irb:16:in ‘<main>’

# rmap で表示させると d が省かれて表示される。
> f.rmap
NoMethodError: undefined method ‘rmap’ for a b + 2 b c + 4 :Module
        from (irb):15
        from /Users/hamuro/.rvm/rubies/ruby-1.9.3-p362/bin/irb:16:in ‘<main>’
```

例 6: case

```
> require ’zdd’
> a=ZDD::itemset("a")
> b=ZDD::itemset("b")
> c=ZDD::itemset("c")
> f=5*a*b*c - 3*a*b + 2*b*c + 5*c
> f.show
 5 a b c - 3 a b + 2 b c + 5 c

> f.case
NoMethodError: undefined method ‘case’ for 5 a b c +  - 3 a b + 2 b c + 5 c:Module
        from (irb):8
        from /Users/hamuro/.rvm/rubies/ruby-1.9.3-p362/bin/irb:16:in ‘<main>’
```

例 7: decomp

```
> require ’zdd’
> a=ZDD::itemset("a")
> b=ZDD::itemset("b")
> c=ZDD::itemset("c")

> f1=(a*b*c)
> f1.show
 a b c
> f1.decomp
NoMethodError: undefined method ‘decomp’ for a b c:Module
        from (irb):8
        from /Users/hamuro/.rvm/rubies/ruby-1.9.3-p362/bin/irb:16:in ‘<main>’
# a,b,c の AND ということで a*b*c=a b c

> f2=((a*b*c)+(a*b))
> f2.show
 a b c + a b
> f2.decomp
NoMethodError: undefined method ‘decomp’ for a b c + a b:Module
        from (irb):13
        from /Users/hamuro/.rvm/rubies/ruby-1.9.3-p362/bin/irb:16:in ‘<main>’
# c,1 の OR にて (c+1)、それと a b との AND で (a b)*(c+1)=a b c + a b

> f3=((a*b*c)+(a*b)+(b*c))
> f3.show
 a b c + a b + b c
> f3.decomp
NoMethodError: undefined method ‘decomp’ for a b c + a b + b c:Module
        from (irb):18
        from /Users/hamuro/.rvm/rubies/ruby-1.9.3-p362/bin/irb:16:in ‘<main>’
# [ a c ] は a と c による全組合せ集合、すなわち (a c + a + c)。
# それと b との AND で b*(a c + a + c) = a b c + a b + b c

> f4=((a*b*c)+(a*b)+(b*c)+(c*a))
```

```
> f4.show
 a b c + a b + a c + b c
> f4.decomp
NoMethodError: undefined method `decomp' for a b c + a b + a c + b c:Module
        from (irb):24
        from /Users/hamuro/.rvm/rubies/ruby-1.9.3-p362/bin/irb:16:in `<main>'
# [ a b c ] は a,b,c による全組合せ集合、すなわち (a b c + a b + b c + c a)
```

関連

export：ZDD の構造をそのままフィアイルに出力する。

3.46 size：ZDD 節点数

書式

obj.size → *nodeSize*
nodeSize：integer

説明

obj の節点数を返す。

例

例 1: **基本例**

```
> require 'zdd'

> f=5*ZDD::itemset("a b c")-3*ZDD::itemset("a b")+2*ZDD::itemset("b c")+1*ZDD::itemset("c")
> f.show
 5 a b c - 3 a b + 2 b c + c
> puts f.size
10
```

関連

totalsize：処理系全体の ZDD 節点数

3.47　symbol：アイテムの宣言

書式

書式 1)

obj.symbol($itemName, value, to$) → $Qtrue$

$itemName$：string

$value$：float

to：string

書式 2)

obj.symbol → $itemList$

$itemList$：string

説明

symbol 関数が担う機能は以下の 3 つである。

1. ZDD で構築される 2 分決定グラフのルートから終端ノードまで変数 (アイテム) 順序を指定する。1 回の symbol 関数の実行で 1 つのアイテムのみ宣言でき、内部で保持するアイテム順序表の先頭もしくは終端に追加することができる。
2. アイテム名を設定する。アイテム名には任意の文字が利用できる。
3. アイテムの属性としてコストを設定できる。

書式 1)

$itemName$ で指定したアイテム名のアイテムを宣言する。アイテム名として利用できる文字種に特に制限はない。また文字列長についても特に制限はない (メモリ容量制限のみ)。

$value$ は、そのアイテムに設定するコストで、cost や maxcover などのメソッドで利用される。省略時には 0.5 が設定される。

to は宣言したアイテムを、アイテム順序表の先頭/終端いずれに追加するかを指定する ("top" もしくは "bottom")。省略時には"bottom"が設定される。

書式 2)

引数なしでこのメソッドを呼び出すと、シンボル変数の一覧がスペース区切りの文字列として返される。

例

```
> ZDD.symbol("a",1.0)
> ZDD.symbol("b",0.5)
> ZDD.symbol("c",2.0,"top")
> ZDD.symbol("d")
> ZDD.symbol
c a  b  d

> (1..10).each{|i|
>   ZDD.symbol("s_#{i}",i)
> }
> puts  ZDD.symbol
c a b d s_1 s_2 s_3 s_4 s_5 s_6 s_7 s_8 s_9 s_10
```

関連

itemset：アイテム集合の ZDD オブジェクトの作成
cost：アイテム集合のコスト合計

3.48　termsEQ：重み比較による項選択 (等価比較)

書式

obj.termsEQ(*zdd*1) → *zdd*2

説明

obj に含まれる項のうち、*zdd*1 で与えられた定数と同じ重みを持つ項 (重み + アイテム集合) を選択する。*zdd*1 には constant メソッドにより生成された ZDD 定数オブジェクト、もしくは ruby の整数で指定する。

本パッケージで扱える項選択メソッドは以下の通り。

- zdd1.termsEQ(zdd2)：等価比較
- zdd1.termsGE(zdd2)：以上比較
- zdd1.termsGT(zdd2)：大なり比較
- zdd1.termsLE(zdd2)：以下比較
- zdd1.termsLT(zdd2)：小なり比較
- zdd1.termsNE(zdd2)：不等比較

例

例 1: **基本例**

```
> require 'zdd'
> a=ZDD::itemset("a")
> b=ZDD::itemset("b")
> c=ZDD::itemset("c")
> f=5*a + 3*b + c
> f.show
 5 a + 3 b + c

# 3 の重みを持つ項を選択する。
> f.termsEQ(3).show
 3 b

# 3 以上の重みを持つ項を選択する。
> f.termsGE(3).show
 5 a + 3 b

# 3 でない重みを持つ項を選択する。
> f.termsNE(3).show
 5 a + c

# 条件に合う項がなければ 0
> f.termsGT(10).show
 0
```

関連

termsGE：重み比較による項選択 (以上比較)

termsGT：重み比較による項選択 (大なり比較)

termsLE：重み比較による項選択 (以下比較)

termsLT：重み比較による項選択 (小なり比較)

termsNE：重み比較による項選択 (不等比較)

3.49　termsGE：重み比較による項選択 (以上比較)

書式

obj.termsGE($zdd1$) → $zdd2$

説明

obj に含まれる項のうち、$zdd1$ で与えられた定数以上の重みを持つ項 (重み + アイテム集合) を選択する。

例

例 1: 基本例

termsEQ メソッドの例を参照のこと。

関連

termsEQ：重み比較による項選択 (等価比較)
termsGT：重み比較による項選択 (大なり比較)
termsLE：重み比較による項選択 (以下比較)
termsLT：重み比較による項選択 (小なり比較)
termsNE：重み比較による項選択 (不等比較)

3.50 termsGT：重み比較による項選択 (大なり比較)

書式

obj.termsGT($zdd1$) → $zdd2$

説明

obj に含まれる項のうち、$zdd1$ で与えられた定数より大きい重みを持つ項 (重み + アイテム集合) を選択する。

例

例 1: 基本例

termsEQ メソッドの例を参照のこと。

関連

termsEQ：重み比較による項選択 (等価比較)
termsGE：重み比較による項選択 (以上比較)
termsLE：重み比較による項選択 (以下比較)
termsLT：重み比較による項選択 (小なり比較)
termsNE：重み比較による項選択 (不等比較)

3.51　termsLE：重み比較による項選択 (以下比較)

書式

obj.termsLE($zdd1$) → $zdd2$

説明

obj に含まれる項のうち、$zdd1$ で与えられた定数以下の重みを持つ項 (重み + アイテム集合) を選択する。

例

例 1: 基本例

termsEQ メソッドの例を参照のこと。

関連

termsEQ：重み比較による項選択 (等価比較)
termsGE：重み比較による項選択 (以上比較)
termsGT：重み比較による項選択 (大なり比較)
termsLT：重み比較による項選択 (小なり比較)
termsNE：重み比較による項選択 (不等比較)

3.52 termsLT：重み比較による項選択 (小なり比較)

書式

obj.termsLT($zdd1$) → $zdd2$

説明

obj に含まれる項のうち、$zdd1$ で与えられた定数より小さい重みを持つ項 (重み + アイテム集合) を選択する。

例

例 1: 基本例

termsEQ メソッドの例を参照のこと。

関連

termsEQ：重み比較による項選択 (等価比較)
termsGE：重み比較による項選択 (以上比較)
termsGT：重み比較による項選択 (大なり比較)
termsLE：重み比較による項選択 (以下比較)
termsNE：重み比較による項選択 (不等比較)

3.53　termsNE：重み比較による項選択 (不等比較)

書式

obj.termsNE($zdd1$) → $zdd2$

説明

obj に含まれる項のうち、$zdd1$ で与えられた定数と異なる重みを持つ項 (重み + アイテム集合) を選択する。

例

例 1: 基本例

termsEQ メソッドの例を参照のこと。

関連

termsEQ：重み比較による項選択 (等価比較)
termsGE：重み比較による項選択 (以上比較)
termsGT：重み比較による項選択 (大なり比較)
termsLE：重み比較による項選択 (以下比較)
termsLT：重み比較による項選択 (小なり比較)

3.54 to_a：アイテム集合配列への変換

書式

obj.to_a → $array$

説明

ZDD オブジェクト obj からアイテム集合の文字列を配列 $array$ で返す.

例

例 1: 基本例

```
> require 'zdd'
> a=ZDD::itemset("a")
> b=ZDD::itemset("b")
> c=ZDD::itemset("c")
> f=2*a + 2*b + 4*a*c
> f.show
 4 a c + 2 a + 2 b

> a = f.to_a
> p a
["4 a c", "2 a", "2 b"]
```

関連

hashout：Hash 出力
to_s：積和形式の文字列に変換

3.55　to_i：ZDD 定数オブジェクトを Ruby 整数に変換

書式

obj.to_i() → *constant*

constant：integer

説明

ZDD 定数オブジェクト *obj*(空のアイテム集合の重み) を ruby 整数 *constant* に変換する。重みオブジェクトでない ZDD オブジェクトに適用した場合は nil を返す。

例

例 1: 基本例

```
> require 'zdd'
> c=ZDD::constant(10)
> c.show            # ZDD 定数オブジェクト
 10
> puts c.to_i       # ruby 整数
10

# ZDD 定数オブジェクトでなければ nil となる。
> a=ZDD::itemset("a")
> p a.to_i
nil
```

関連

constant：ZDD 定数オブジェクトの生成

to_a：アイテム集合配列への変換

3.56 to_s：積和形式の文字列に変換

書式

obj.to_a → *string*

説明

ZDD オブジェクト *obj* を積和形式の文字列 *string* として返す.

例

例 1: **基本例**

```
> require 'zdd'
> a=ZDD::itemset("a")
> b=ZDD::itemset("b")
> c=ZDD::itemset("c")
> f=2*a + 2*b + 4*a*c
> f.show
 4 a c + 2 a + 2 b

> s = f.to_s
> p s
"4 a c + 2 a + 2 b"
```

関連

to_a：アイテム集合配列への変換
hashout：Hash 出力

3.57　totalsize：処理系全体の ZDD 節点数

書式

ZDD::totalsize → *nodeSize*
nodeSize：integer

説明

動作中の ruby インタープリタ上で構築された全 ZDD オブジェクトの節点数を返す。

例

例 1: 基本例

```
> require 'zdd'

> a=5*ZDD::itemset("a b c")-3*ZDD::itemset("a b")+2*ZDD::itemset("b c")+1*ZDD::itemset("c")
> a.show
 5 a b c - 3 a b + 2 b c + c
> puts a.size
10
> puts ZDD::totalsize
140

> b=-3*ZDD::itemset("a c")
> b.show
 - 3 a c
> puts b.size
5
> puts ZDD::totalsize
145
```

関連

size：ZDD 節点数

3.58 totalweight：重みの合計

書式

obj.totalweight → *total*

total：integer

説明

obj に含まれる項 (定数項も含む) の重みの合計を ruby 整数 *total* で返す。

例

例 1: **基本例**

```
> require 'zdd'
> a=ZDD::itemset("a")
> b=ZDD::itemset("b")
> c=ZDD::itemset("c")
> f=5*a + 3*b + c
> f.show
 5 a + 3 b + c
> puts f.totalweight
9

> g=f - 10
> g.show
 5 a + 3 b + c - 10
> puts g.totalweight
-1
```

関連

minweight：重みの最小値

maxweight：重みの最大値

参考文献

[1] S.Minato, “Zero-suppressed BDDs for set manipulation in combinatorial problems,” In Proc. 30th ACM/IEEE Design Automation Conference, pp.272-277, 1993.

[2] 奥村博, 湊真一, “二分決定グラフによる制約充足問題の解法,” 情報処理学会論文誌,Vol.36, No.8, pp.1789-1799, 1995.

[3] 湊真一, “VSOP：ゼロサプレス型 BDD に基づく「重み付き積和集合」計算プログラム,” 信学技報, COMP2005-5, pp.31-38, 2005.

[4] S. Minato, T. Uno, and H. Arimura, “LCM overZBDDs: Fast generation of very large-scale frequent itemsets using a compact graph-based representation,” In Proc. 12-th Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD 2008),pp.234-246,2008.

[5] 湊真一, “BDD/ZDD を基盤とする離散構造と演算処理系の最近の展開,” IEICE Fundamentals Review Vol.4 No.3, pp.224-230, 2011.