ZDD Ruby Package Documentation

ZDD version: 1.1.0

Revision history:
November 26, 2013 : First release

January 17, 2014

JST-ERATO MINATO Discrete Structure Manipulation System Project

Contents

Chapter 1

Let’s Start

1.1 Summary

ZDD (Zero-suppressed Binary Decision Diagrams) is a data structure used for efficient manipulation of weighted item combinations based on reduction rules. ZDD VSOP (Valued-Sum-of-Products calculator)[4] is implemented as a Ruby Extension Library for the computation of item combinations.

The founder of ZDD, Professor Shin-ichi Minato, developed a ZDD program which is made available for extended development in Ruby language. The Ruby Extension Library for ZDD is therefore developed in collaboration with ERATO Minato Discrete Structure Manipulation System Project.

ZDD can enumerate huge number of combinations represented in a compact structure. For example, to enumerate all possible combinations of the products (frequent itemsets) purchased from the supermarket in the purchase history at a defined minimum support, in most cases, the number of combinations generated will grow exponentially. The ZDD data structure therefore is designed to provide a powerful framework to store all possible combinations in an unique and compact form efficiently.

Various calculations can be applied directly on ZDD objects where large scale itemsets are stored in a compact form for efficient processing. For example, users will be able to select the patterns containing "natto" from millions of enumerated frequent itemsets, or identify the differences of frequent itemset patterns between male and female. It is also possible to compute the ratio of size of ZDD. For details on theoretical context, please refer to the reference at the end of this document.

In this package, ZDD is handled as (known as **ZDD object**) Ruby object. Various functions for the ZDD objects correspond to class methods, operator overloading is used for ZDD objects using operands for ZDD object (`+`,`-`,`==` etc.), to enable seamlessly integration of functions between Ruby and ZDD.

The ZDD package also supports automatic type conversion designed to reduce stress on programming.

1.2 Installation

ZDD package is distributed as part of the mining package distribution by Nysol. Installation can be done by compiling from source or by installing rubygem. Refer to mining package manual at NYSOL for details on the installation.

1.3 Modify the maximum number of nodes

The maximum number of ZDD nodes in this package is set to 40 million by default. The program terminates with an error if the number of nodes exceeds this value. The maximum number of nodes can be modified by setting the environment variable at `ZDDLimitNode`. For example, the maximum number of nodes can be set to 100 million nodes in the bash shell as follows.

```
$ export ZDDLimitNode=100000000
```

The memory consumption per node is about 21-25 bytes on a 32bit OS machine. The program will consume about 1GB of main memory when the default limit of 40 million nodes is used.

On a 64bit OS machine, the memory allocation is twice of the normal 32bit OS implementation, with 30%percent increase in speed, and each node takes up to 28 to 32 bytes. Approximately 1.3GB of main memory is consumed when the program runs up to the default limit of 40 million nodes. By changing the maximum number of nodes according to the processing environment, the scale of ZDD structure can be increased.

1.4 Terminologies

The following section explains the terminologies used in this manual. Note that there may be differences of the terms used in the reference at the end of this manual.

Item, Itemset, Term, Valued sum of products set, Expression

An element in a set is referred to as "**item**", and an "**itemset**" contains a group of item elements.

The concept is easier to understand when these terms are set into context in a supermarket scenario. Commodity products are considered as items, and the combination of the products are referred to as itemset. When weight is attached to a "**term**" in the itemset, it is known as "**valued sum of products set**". For example, the valued sum of products for 3 items a,b,c is represented as "`abc+3ab+4bc+7c`" (this notation is referred to as "valued sum of products format"). It consists of four terms `abc,3ab,4bc,7c`. In this case, `3ab` consists of weight of `3` for itemset {a,b}. Back to the supermarket scenario, this means that one customer purchased 3 products `a,b,c` at the same time, and there are three customers who bought `a,b` at the same time.

Empty itemset, ZDD constant object

The itemset without element is referred to as "**empty itemset**". Considering the valued sum of products set "`abc+3ab+4bc+7c+3`", the a weight of `3` is attached to an empty itemset and thus it is shown as 3.

In supermarket scenario, it means that there are three customers did not purchase anything. Thus, empty itemsets of ZDD object is referred to as **ZDD constant object**.

Item order table

ZDD is a binary decision tree that contains a compact decision tree graph, and the level of the decision tree (depth) corresponds to the item. Further, this level, the order from root to the leaves is managed by the table known as "**item order table**".

It is important to manage item order since the order significantly impacts the size of ZDD (number of nodes). When the size of the ZDD increases, the processing speed will be reduced accordingly. The item order table can be registered at any time using the symbol function. If the number of combinations is extremely large, it is necessary to use the order table to register order of items.

1.5 Notation

Notation of itemset

The execution results in Ruby present the itemset which contains items and weight separated by space. Itemsets are displayed as character strings within text or comments, spaces between items are removed for simplicity. For example, itemset {a,b,c} with weight of 3 is displayed as "`3 a b c`" in the results. However, it will be displayed as "`3abc`" in text or comment.

Example

Multiple examples are included in this manual. The meaning of symbols used in the examples are as follows.

- `>` : Display Ruby input method.
- `$` : Display shell command line input.

- `#` : Display comments.
- No symbol : Display the execution results.

The example contains the execution results log from the basic Ruby command line tool `irb`.

Chapter 2

Tutorial

This tutorial starts with the basic functions of ZDD, followed by an application example on how to solve the N-Queens problem. The examples can be executed with Ruby script or experimented from command line interactively using irb. This tutorial requires some basic knowledge of Ruby and you should have already installed ZDD Ruby Extension Library.

2.1 Use require method to load ZDD Ruby Extension Library

ZDD Ruby Extension Library uses RubyGems package. Use the require method to load rubygems library and load the zdd library. Note that Ruby 1.9 and above includes RubyGems by default, thus it is not necessary to load gem libraries.

```
> require 'rubygems' #Not required for Ruby 1.9 and above.
> require 'zdd'
```

2.2 Items Declaration

Use ZDD::symbol method to define an individual item. Use one method to define one item. The order of items declaration is important and corresponds to the ZDD tree structure from root level. Items not declared in the proper order significantly affects how ZDD structure is generated and the resulting size of the tree structure. The example below shows a declaration of four items `"a","b","c","d"`. If the item order is not important, users can skip the symbol declaration function and use the itemset function described in the next section.

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

Only items used are declared as symbols, next, the itemsets consisting of defined items will create the ZDD object.

It is possible to create a ZDD object with the ZDD::itemset method to enumerate space delimited item names as itemsets. In the following, the ZDD object is expressed as itemset "abc" consisting of items "a", "b", "c" which is set as Ruby variable a. The content in the object is displayed in valued sum of products format using the show method.

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

In the following, itemset consisting of 1 item is constructed, it will be used in the later section. Although the item name and the Ruby variable are both named as `a`, note that these two are totally different,

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

2.3 Operation

The itemsets are transformed using various operations defined in ZDD. By combining the operations, the ZDD objects can be flexibly processed and manipulated.

Several examples are shown below. Some operations are computed just like general polynomial expressions, while some are not.

```
> (a+a).show # Weights are incorporated by the addition of the identical itemsets.
2 a
> (2*a).show # The result is the same by multiplying the variable.
2 a
> (a*b).show # An item is added to the itemset when multiplying different items.
a b
> (a*a).show # When multiplying the same item, the weight becomes 2 items
since there are two items.
2 a
> (2*a-a).show # Subtraction
a
> ((a*b*c+b*d+c)/b).show # Division
a c + d
> ((a+b)*(c+d)).show
a c + a d + b c + b d
> ((a+1)*(a+1)).show #  "1" in the last term is the weight of the empty item set.
3 a + 1
> ((a+1)*(a-1)).show
a - 1
```

Finally, let's enumerate all subsets of the itemset {a,b,c,d} for this example. The expression and its results are shown below.

```
> 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
```

The method for expanding the expression above is similar expanding polynomial expressions. The calculation results on the right side are constructed as ZDD object and substituted into Ruby variable f. Consequently, $16(=2^4)$ itemsets are enumerated. Note that the last term "1" is the weight of an empty itemset.

2.4 Frequent itemsets

Assuming four customers (f, g, h, i) purchased the items a, b, c, d in a supermarket.

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

Let's find out the common itemsets across three or more customers from the purchase data. The procedure is simple. First, find out all subsets of the itemset each customer has purchased, then aggregate them. The method is shown as follows.

```
> 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
```

Based on the above result, there are two customers who purchased products a, b, and two customers who purchased products a, b, c. Now, use the termsGE function to select the terms with a weight that is equal or greater than 3.

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

Alternatively, one may use the restrict function to select the itemsets which include itemset `"a b"`, alternatively, use the permit function to select itemsets which contains itemset `"a b"`.

```
> 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
```

The ZDD package contains several methods to store the enumeration results of frequent itemsets as ZDD object other than the method described above. More details can be found infreqpatA function or lcm function.

2.5 Cast

Up to this point, it is not explicitly explained in this package how data used in various ZDD operations and functions will be automatically converted (cast) to the corresponding data type.

For example, the operation of `2*a` described in the previous section is derived by multiplying ZDD object variable a with Ruby integer 2.

The multiplication operator of ZDD `*` obtains two ZDD object as arguments, if it is not a ZDD object as shown in the example, the contents are automatically converted to ZDD object for calculation.

Internally, `2*a` operates as `ZDD::constant(2)*a`. Here, `constatnt` function defines the weight of empty itemset.

In the following examples, the Ruby character string "a b" is automatically converted to ZDD object. Internally, it is operated as (a+ZDD::itemset("a b")).show.

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

In the following, the character string of the two operands is combined to become a Ruby String object .

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

2.6 Combination with control statement

The series of partial itemset enumeration shown in section 2.2 and 2.3 can be combined with the control statement. The examples are described below.

In the example below, the item is not declared by the symbol function, but the itemset is directly defined by the itemset function. Using the multiplication operator *= on Ruby variable t, subsequent computation results is accumulated one after the other.

```
> 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 Solving the N-queens problem

Let's solve the N-queens problem by applying various ZDD operations introduced in the previous sections. N-queens problem is the number of different ways N number of queens can be set up on an N×N chessboard so that no two queens may attack each other. The queen is a chess that is capable of moving across and at angular movements in Shogi (Japanese chess). As shown in the figure 2.1, the chess can move in a total of eight directions: right, left, up, down and diagonally.

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

Figure 2.1: The queen indicated as "o" can move to the squares marked as "x".

(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)

Figure 2.2: Coordinates of the 5×5 chessboard

o				
		o		
				o
	o			
			o	

Figure 2.3: Example of a solution to 5 queens problem

Figure 2.4 shows a Ruby script for 5 queens problem using ZDD.

The assumed coordinates of the chessboard used in this script is shown in figure 2.2. The basic idea of the script is as follows. First, assume an item as a square where the Queen can be placed on the chessboard. The total number of squares on the chessboard is computed by $5 \times 5 = 25$, and all itemset combinations can be enumerated from 2^{25}.

It is sufficient to apply the explanation in section 2.3 for this example. Then select the itemset that meets the two conditions as follows.

1. Remove all itemsets that may attack each other.
2. Delete the itemset with size less than 5.

In this case, among itemsets whose size is larger than 5, remove those which satisfy the first condition. Next, verify the movement of the chess with the details described in the figures. At the end of the script, the number of terms in ZDD object stored during the operation and the number of internal contact points will be returned as output.

The results are shown as follows. There are about 33.55 million (value of $2^2 5$) possible ways to arrange the queens on 25 squares, represented by only 25 ZDD nodes. The number of combinations (selNG) which attack each other selected by the restrict function is about the same, but the number of ZDD nodes rapidly increases to 587, it is also where most time is consumed. Finally, the solution includes 10 enumerated itemsets.

Among them, the placement of "0,0 1,2 2,4 3,1 4,3" in the initial solution (term) is shown in figure 2.3.

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

n = 5

# In the nested loop below, the itemset combinations of 25 squares (items) are enumerated,
and stored in the variable "all".
# Use the characters "i, j" to indicate the coordinates for the item name.
all=ZDD.constant(1)
(0...n).each{|i|
        (0...n).each{|j|
                all *= (1+ZDD::itemset("#{i},#{j}"))
        }
}

# Enumerate all combinations two squares that attack each other and stored in the variable "ng".
ng=ZDD.constant(0)
(0...n).each{|i| # row loop
  (0...n).each{|j| # column loop
     # Item pair which has the same row number (i) with the square i.j
    (j+1...n).each{|k|
      ng+=ZDD::itemset("#{i},#{j} #{i},#{k}") # Row
    }
     # Item pair which has the same column number (i) with the square i.j
    (i+1...n).each{|k|
      ng+=ZDD::itemset("#{i},#{j} #{k},#{j}") # Row
    }
    # item pair with the square i.j. and lower right direction from that square
    (1...[n-i,n-j].min).each{|k|
      ng+=ZDD::itemset("#{i},#{j} #{i+k},#{j+k}") #
    }
    # item pair with the square i.j. and lower left direction from that square
    (1...[n-i,j+1].min).each{|k|
      ng+=ZDD::itemset("#{i},#{j} #{i+k},#{j-k}") #
    }
  }
}
st=Time.new # Use for time measurement
selNG=all.restrict(ng)     # 1) Select itemsets that contains the items pair which attacks
each other from all itemsets "all".
selOK=selNG.iif(0,all)     # 2) Exclude the itemset obtained in 1) from the all itemsets "all".
selLT=selOK.permitsym(n-1) # 3) From the results in 2), select the itemset where the size is
less than n-1.
ans  =selLT.iif(0,selOK)   # 4) From the results in 2), remove the itemsets obtained in 3).

# Display the number of ZDD itemset (totalweight function) created from the calculation process
 and the number of ZDD nodes (size function).
# totalweight function is a method to aggregate the weight of each term in ZDD.
# Since the weight of all terms is 1, thus the number of itemset in the expression is
known from the expression.
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 # Display the solution
```

Figure 2.4: The solution of 5 queens problem with ZDD.

```
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
```

Furthermore, the number of ZDD nodes, solutions and processing time when N is set from 4 to 11 are displayed

Table 2.1: The number of ZDD nodes expressed as N, number of solutions, processing time

N	ZDD nodes (all)	ZDD nodes (selNG)	number of solutions	processing time (seconds)
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

in Table 2.1.

ZDD objects (all) for all combinations of all squares, and ZDD object with the maximum number of ZDD nodes (selNG) are shown.

The number of nodes of selNG is about 20 million when N=11. It means about 600M bytes memory will be consumed for calculating 30 byte per node.

The workspace during the calculation is limited to N=11 on a PC with 8GB memory. More efficient solution using ZDD is shown in the reference [2], users are welcomed to challenge the program by all means.

Chapter 3

Overview of ZDD Operators and Methods

This chapter provides reference to all ZDD operators and functions. The reference consists of four parts including format, description, examples and links. The reference format of the operators and functions are illustrated as follows.

Format

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

Arguments and return values are shown in italics, and the return value of the operator / function is shown on the right hand side of the arrow. *obj* represents the ZDD object on which the method is applied to, *zdd*,*zdd*1 and *zdd*2 represent the ZDD objects as arguments or return values. Other format of Ruby objects are specified in format.

3.1 % : Remainder operator

Format

$zdd1$ % $zdd2 \rightarrow zdd3$

Description

Find out the remainder from the division of two operands $zdd1$ and $zdd2$, return the value as ZDD object $zdd3$.

Example

Example 1: Basic Example

Refer to the example of/ method.

See Also

/ : Division operator

3.2 * : Multiplication operator

Format

*zdd*1 * *zdd*2 → *zdd*

Description

Multiplication of *zdd*1 and *zdd*2.

Consider its similarities with general polynomial, the only difference is when the same item is multiplied by itself, it does not become the power of 2. For instance, it is computed by the expression a*a=a.

Example

Example 1: Basic Example

```
> require 'zdd'
# a. Multiplication of constants.
# Weight is attached to sum of products term x by the expression x*c, the weight
# of each term in x is multiplied by c times.
# The computation is similar to general polynomial.
> 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. Multiplication by 1 itemset
# Weight is attached to sum of products term y by the expression x*y,
# y is added to the weight of each term in x.
# However, note that the multiplication beteween same items is expressed as a*a=a.
> 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. Multiplication of 2 or more itemsets
# Multiplication of the sum of products x,y which is attached with weight is computed
# by the expression x*y,
# The operation enumerates all possible combinations on the two product terms.
# Like terms such as a,b itemset are multiplied.
# The resulting expression includes addition and subtraction operations between
# enumerated item sets.
> ((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
```

See Also

/ :Division operator

3.3 + : Addition operator

Format

*zdd*1 + *zdd*2 → *zdd*3

Description

Addition of weights for the same itemsets corresponding to *zdd*1 and *zdd*2.

Example

Example 1: Basic Example

```
> 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
```

See Also

− : Subtraction operator

+ : Plus unary operator

3.4 + : Plus unary operator

Format

+ *zdd*1 → *zdd*2

Description

Return all itemsets contained in *zdd*1 (without any changes).

Example

Example 1: Basic Example

```
> 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
```

See Also

− : Minus unary operator

3.5 - : Subtraction operator

Format

$zdd1 - zdd2 \rightarrow zdd$

Description

Subtraction of weights for the same itemsets corresponding to $zdd1$ and $zdd2$.

Example

Example 1: Basic Example

```
> 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
```

See Also

$+$: Addition operator

$-$: Unary negative operator

3.6 - : Minus unary operator

Format

$-zdd1 \rightarrow zdd2$

Description

Convert weight contained in *zdd*1 from positive to negative sign and vice versa.

Example

Example 1: Basic Example

```
> 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
```

See Also

+ : Plus unary operator

3.7 / : Division operator

Format

*zdd*1 / *zdd*2 $\rightarrow$ *zdd*3

Description

Compute the division of *zdd*2 by *zdd*1. Refer to the examples below for details.

Examples

Example 1: Division by constant

Division of valued sum of products x by constant c expressed in x/c, where the weight (density) of each term in x is divided by c.

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

# term a: quotient of 13/5 is 2 and thus becomes 2a. term b: quotient of 3/5 is 0 and
# thus not shown.
> (x/5).show
 2 a

# term a: remainder of 13/5 is 3 and thus becomes 3a. term b: remainder of 3/5 is 3 and
# thus becomes 3b.
> (x%5).show
 3 a + 3 b

# restore the original value of x by multiplying the quotient with the divisor 5 and
# adding the reminder.
> (5*(x/5)+(x%5)).show
 13 a + 3 b
```

Example 2: Division by 1 itemset

Division of valued sum of products x by itemset v expressed in x/v, where each term in x is divided by v.

Consider the operation is similar to a general polynomial.

```
> 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

# quotient of 7ab/b is 7a. quotient of 5bc/b is 5c.
> (x/b).show
 7 a + 5 c
```

```
# remainder of 7ab/b is 0. remainder of 5bc/b is 0.
> (x%b).show
 0

# quotient of 7ab/3b is 2a. quotient of 5bc/3b is 1c. quotient of 2c/3b is 0 and thus not shown.
> (y/(3*b)).show
 2 a + c

# remainder of 7ab/3b is ab. remainder of 5bc/3b is 2bc. remainder of 2c/3b is 2c.
> (y%(3*b)).show
 a b + 2 b c + 2 c

# restore the original value of y by multiplying the quotient with the divisor 3b and
# adding the remainder.
> (3*b*(y/(3*b))+(y%(3*b))).show
 7 a b + 5 b c + 2 c
```

Example 3: Division of 2 or more itemsets

Division of 2 valued sum of products x,y is computed as x/y.

Divisor y consists of multiple product terms T_i, find out $Q_i = x/T_i$ for all i. Q_i contains all common itemsets (terms), where Q is defined as the absolute minimum value weight of the terms.

Note that this operation is different than general polynomial.

```
> 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 and x%y are computed as follows.
# Q1=(x/a)=2b +4c +d +0  +0   = 0  +2b +4c +d
# Q2=(x/b)=2a +0   +0 -2c +3d = 2a +0  -2c +3d
# Common itemsets for Q1 and Q2 are c and d, the absolute minimum value is -2c and d.
# The quotient is derived as follows.
> (x/y).show
 - 2 c + d
# compute x-(y*Q) to find out the remainder from the division of x%y.
> (x%y).show
 2 a b + 6 a c + 2 b d

# Restore the original value of x by multiplying the quotient with the divisor y and
# adding the remainder.
> (y*(x/y)+(x%y)).show
 2 a b + 4 a c + a d - 2 b c + 3 b d
```

See Also

% :Remainder operator

* : Multiplication operator

3.8 < : Less than comparison operator

Format

$zdd1 < zdd2 \rightarrow zdd$

Description

Compare itemsets in *zdd1* with itemsets in *zdd2*, select items where the weight is less than *zdd2*.

Example

Example 1: Basic Example

Refer to the example in ==.

See Also

==, ¡, ¡=, ¿, ¿=, ne? : Various comparison operations

3.9 <= : Less than or equal to comparison

Format

$zdd1 <= zdd2 \rightarrow zdd$

Description

Compare itemsets in $zdd1$ with itemsets in $zdd2$, select items where the weight is less than or equal to $zdd2$.

Example

Example 1: Basic Example

Refer to the example in==.

See Also

==, <, <=, >, >=, ne? : Various comparison operations

3.10 > : Greater than comparison operator

Format

ZDD Object $zdd1 > zdd2$

Description

Compare itemsets in $zdd1$ with itemsets in $zdd2$, select items where the weight is greater than $zdd2$.

Example

Example 1: Basic Example

Refer to the example in ==.

See Also

==, <, <=, >, >=, ne? : Various comparison operations

3.11 == : Equal to comparison operator

Format

*zdd*1 == *zdd*2 → *zdd*3

Description

Compare terms in *zdd*1 and *zdd*2, select terms with equal weight (terms with same weight and itemset), and return the itemset. Note that the resulting weight of the itemset returned is 1.

The iif function can be used in cases where weight is also selected.

This package can be used with the following comparison operators (functions).

- zdd1 == zdd2 : Equal to operator
- zdd1 >= zdd2 : Greater than or equal to operator
- zdd1 > zdd2 : Greater than operator
- zdd1 <= zdd2 : Less than or equal to operator
- zdd1 < zdd2 : Less than operator
- zdd1.ne?(zdd2) : Not equal to function

In addition, same?(==) and diff? can be used to evaluate the equivalence on the complete expression.

Examples

Example 1: Basic Example

```
> 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 is defined above)

# Compare x and y, select itemsets with equal weight.
> (x==y).show
 b

# Compare x and y, select itemsets with weight that is greater than or equal to y.
> (x>=y).show
 a + b

# Compare x and y, select itemsets with unequal weights.
> (x.ne?(y)).show
 a + c

# For itemsets present in zdd1 but absent in zdd2 (or vice versa),
# the weight is considered as 0.
> z=2*a + 2*b
> z.show
 2 a + 2 b

# Since itemset c does not exists in z, the weight is considered as 0.
> (x>z).show
 a + c
```

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

See Also

==, <, <=, >, >=, ne? : Various comparison operations

iif : Select by comparison of items

same? : Same comparison operator

diff? : Not equal to comparison operator

3.12 >= : Greater than or equal to operator

Format

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

Description

Compare itemsets contained in *zdd*1 with itemsets contained in *zdd*2, select itemsets where the weight is greater than or equal to *zdd*2.

Example

Example 1: Basic Eaxample

Refer to the example in==.

See Also

==, <, <=, >, >=, ne? : Various comparison operations

3.13 constant : Construct ZDD constant object

Format

ZDD::constant(*weight*) → *zdd*

weight : integer

Description

Specify an integer as the weight of ZDD object *zdd* (weight of empty itemset) at *weight* to change the weight of ZDD object.

Example

Example 1: Basic Example

```
> require 'zdd'
> c=ZDD::constant(10)
> c.show
 10
# The operation between ZDD weight object and Ruby character string,
# where Ruby character string is read as an itemset and automatically cast as ZDD object.
> (c*"a").show
 10 a

# The operation between ZDD weight object and Ruby integer, where Ruby integer is
# treated as ZDD weight object.
> (0*c).show
 0

# ZDD weight object is converted to Ruby integer, and the operation is carried out
# in Ruby integer.
> puts c.to_i*10
100

# In the following example, the weight is defined as 0. This object is automatically cast
# to RubyString for computation.
> a=ZDD::constant(0)
> a+="x"
> a+="x z"
> a+="z"
> a.show
 x z + x + z
```

See Also

to_i : Convert ZDD constant object to Ruby integer

itemset : Construct ZDD object for itemset

3.14 cost : Compute cost of itemset

Format

obj.cost → *cost*

cost : float

Description

Return the value of each item where the cost (Refer to the symbol function) is substituted for items in *obj*.

Example

Example 1: Basic Example

```
> require 'zdd'
# Define the value of symbol a, b, c as 1.0, 0.5, 1.8 accordingly.
> 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")

# The expression creates a=1.0 for symbol a
> puts a.cost
1.0

# a=1.0,b=0.5 is substituted into expression "a b" and becomes 1.0*0.5=0.5
> f=a*b
> f.show
 a b
> puts f.cost
0.5

# a=1.0,b=0.5,c=2.0 is substituted into the expression "a b + 2 a + c + 3" and becomes
# 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
```

See Also

symbol : Declare items

maxcover : Select itemset with maximum cost

maxcost : Cost of itemset with maximum cost

mincover : Select itemset with minimum cost

mincost : Cost of itemset with minimum cost

3.15 count : Count the number of terms

Format

obj.count → *numItemsets*

numItemsets : integer

Description

Return the number of terms (number of itemsets) stored in *obj*.

Example

Example 1: Basic Example

```
> 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
```

See Also

3.16 csvout : Output CSV file

Format

obj.csvout(*fileName*) → *self*

fileName : string

Description

Return the contents of ZDD object *obj* for the file specified at *fileName* in CSV format. The output returns 2 fields including weight and itemset. The output of the items in the itemset is delimited by space. However, for empty itemset (terms with weight only), the output will only return the weight in the first field, and null will be printed to the second field in CSV. Nevertheless, null ZDD object (empty itemset with weight of 0) returns file with 0 byte.

The field name will not be printed to the output. Value of self is returned.

Example

Example 1: Basic Example

```
> 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")
# Contents of output.csv are as follows.
# Weight of 4 empty itemsets will be printed as nulll in the output.
# 1,a b d
# 5,b c
# 3,d
# 4,

# null ZDD object (empty itemsets with 0 weight) returns file with 0 byte.
> y=ZDD::constant(0)
> y.csvout("null.csv")
```

See Also

hashout : Hash output

3.17 delta : Exclusive-OR operation

Format

obj.delta(*zdd*1) $\rightarrow$ *zdd*2

Description

Find out the exclusive-OR (XOR) operation on $\alpha \oplus \beta$ on the itemset β from the ZDD object *obj* which contains α and *zdd*1, and return the result of ZDD object as *zdd*2.

For example, XOR operations on itemset `abc` and `bcd` are as follows.

`abc.delta(bcd) = abc`$\oplus$`bcd = ad`

Find out the XOR operation between multiple itemsets.

`(abc + a).delta(bcd + b) = abc`$\oplus$`bcd + abc`$\oplus$`b + a`$\oplus$`bcd + a`$\oplus$`b`

`= ad + ac + abcd + ab`

The weight is computed by expanding the same itemset to multiple instances.

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

In addition, when $\alpha \oplus \beta$ is changed to $\alpha \cap \beta$ (intersection operation), it becomes delta function.

Example

Example 1: Basic Example

In the following, using items `a,b,c`, find out the exclusive OR on itemsets `2ab + a + 3b`, `abc + 2ab + bc + 1`, `ab+a`.

```
> require 'zdd'
# First, define the itemsets
> 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)`$\oplus$`a` $=$ `3ab + 2b + 1`

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

`(2ab + a + 3b)`$\oplus$`b` $=$ `ab + 2a + 3`

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

`(2ab + a + 3b)`$\oplus$`ab` $=$ `3a + b + 2`

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

`(2ab + a + 3b)`$\oplus$`1` $=$ `2ab+a+3b` Since constant 1 is an empty itemset, it is remained in the original set for solving XOR operation.

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

The operation result of the each term in `(abc + 2ab + bc + 1)`⊕`(2ab + a)` are as follows:

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

The result is summarized as `a b c + 2 a b + 2 a c + a + b c + 2 b + 2 c + 4`.

```
> 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
```

See Also

3.18 density : Compute the density of ZDD

Format

obj.density → *dens*

dens : float

Description

Density is defined as the ratio of the number of items to the total number of registered itemsets stored in *obj*.

Example

Example 1: Basic Example

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

# Total number of combinations using 3 items a,b,c is eight.
# All combinations are stored in F in the expression below, with a density of 1.0.
> 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

# In the following, one set of combination (a b) is stored in F.
# The density becomes 1/8=0.125.f=a*b
> f.show
 a b c + a b + a c + a + b c + b + c + 1

> puts f.density
1.0

# When three sets of combinations (a b,a,b) are stored in F, the density becomes 3/8=0.375.
> f+=a
> f+=b
> f.show
 a b c + a b + a c + 2 a + b c + 2 b + c + 1
> puts f.density
1.0
```

See Also

3.19 diff? : Not equal to comparison operator

Format

obj.diff?(*zdd*) → *bool*

Description

Compare between two ZDD objects *obj* and *zdd*, return false if they are the same, and return true if the two objects are different.

Example

Example 1: Basic Example

```
> 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
```

See Also

same? : Equal to comparison operator

3.20 each : Iterate for each itemset

Format

obj.each{|*item*|. . . } → *Qtrue*

Description

Read itemset at one *item* at a time from ZDD object *obj* from the specified block.

Example

Example 1: Basic Example

```
> 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
```

See Also

each_item : Iterate for each item

3.21 each_item : Iterate for each item

Format

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

Description

Read itemset from ZDD object *obj* one at a time, each set of weight and item are returned as *weight* and *item* correspondingly for the specified block.

When the itemsets are processed, if item is the first item, *top* will return as true, if item is the last item, *bottom* will return false.

Example

Example 1: Basic Example

```
> 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
----------
```

See Also

each : Return by itemset

3.22 export : Serialized output of ZDD

Format

obj.export(*fileName*)

fileName : string

Description

Serialize output of ZDD internal structure in *obj* as text. Export the output to the file specified at *fileName*. If the file name is not specified, the result is sent to standard output (STDOUT).

Since the contents declared by the symbol function is not serialised according to item order, thus when restoring the ZDD object with import function, items must be declared again according to the same order.

Refer to import function for specific examples.

Example

Example 1: Basic Example

```
> 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
```

See Also

import : Serialized import of ZDD

csvout : Output CSV file

hashout : Hash output

3.23 freqpatA : Enumerate frequent itemsets

Format

obj.freqpatA(*minsup*) → *zdd*

minsup : integer

Description

Enumerate all frequent itemsets from ZDD object *obj* that is more than the minimum support *minsup*, and return the ZDD object as *zdd*.

This is a similar LCM function which reads the transaction data from a file and enumerates frequent itemset by LCM algorithm, which is much faster than this function. However, the item name must be specified in whole integers, and the items must be arranged in order, thus if efficiency is not required, this function is sufficient.

Example

Example 1: Basic Example

```
> 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

# At the expression t=a*b*c + a*b + a + b*c*d + a,
# Itemset ab appeared twice in first and second term.
# Itemset a appeared four times in first, second, third, and fifth term.
> t.freqpatA(2).show
 a b + a + b c + b + c + 1

# Maximal itemset (itemsets that are not included in other itemsets)
# Among the frequent itemsets above, a is included in ab, b and c is include in bc,
# Considering 1 will be included in other itemsets, a,b,1 will not be returned as output.
> t.freqpatM(2).show
 a b + b c

# Closed itemsets (emerged itemsets are in the same itemset group as in maximal itemset)
# Among the above frequent itemsets, bc and c appeared as 1st and 4th term.
# The emerged itemsets belong to the same itemset group.
# Within this group, only bc is the maximum.
# Other enumerated closed itemsets emeraged as a different group.
> t.freqpatC(2).show
 a b + a + b c + b + 1

# Run the three functions with a minimum support of 3.
> t.freqpatA(3).show
 a + b + 1
> t.freqpatM(3).show
 a + b
> t.freqpatC(3).show
 a + b + 1
```

See Also

freqpatM : Enumerate maximal itemsets

freqpatC : Enumerate closed itmesets

lcm : LCM over ZDD

3.24 freqpatC : Enumerate closed itemsets

Format

obj.freqpatC(*minsup*) → *zdd*

minsup : integer

Description

Enumerate all closed itemsets from ZDD object *obj* that is more than the minimum support *minsup*, and return the ZDD object as *zdd*.

Example

Example 1: Basic Example

Refer to example in freqpatA.

See Also

freqpatA : Enumerate frequent itemsets

freqpatM : Enumerate closed itemsets

lcm : LCM over ZDD

3.25 freqpatM : Enumerate maximal frequent itemsets

Format

ZDD Object *obj*.freqpatM(*minsup*) → *zdd*

minsup : Ruby Integer

Description

Enumerate all maximal itemsets from ZDD object *obj* that is more than the minimum support *minsup*, and return the ZDD object as *zdd*.

Example

Example 1: Basic Example

Refer to the example in freqpatA.

See Also

freqpatA : Enumerate frequent itemsets

freqpatC : Enumerate closed itemsets

lcm : LCM over ZDD

3.26 hashout : Hash output

Format

obj.hashout → *hash*

Description

Convert contents in *zdd* to Ruby *hash* object and return the object.

Each hash key corresponds to weight of item set.

However, there is an upper limit to the maximum number of itemsets that can be returned, if it exceeds 4,096,000 itemsets, the output process will be forced to stop.

The partly method can be used to check whether the process stopped during processing.

Example

Example 1: Basic Example

```
> 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}
```

See Also

partly : Partial Hash output flag

to_a : Convert to itemset array

csvout : CSV file output

export : Serialized output of ZDD

3.27 iif : Select by comparison of items

Format

obj.iif(zdd1,zdd2) → *zdd*3

Description

Select terms in itemset included in *obj* from *zdd*1, select terms in itemset not included in *obj* from *zdd*2, and return the selected term which created the ZDD object as *zdd*3.

The weight of the itemset in *obj* does not affect the operation.

Examples

Example 1: Basic Example

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

# From all terms in the first argument of iif, select terms 2a, 3b which contains
# (a+b) with items a,b, and from all terms in the second argument of iif,
# select terms 8c, 9d which contains (a+b) with items a,b.
> 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

# Used in conjunction with typical comparison operators as follows.
> 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

# Compare x and y, select term(s) where the weight of x is great than y, otherwise select y.
# If the result of x>y includes a, select 3a from the first argument x,
# Select the other itemsets 2b and 4c from the second argument y.
> r1=(x>y).iif(x,y)
> r1.show
 3 a + 2 b + 4 c

# Compare x and y, select term(s) where the weight of x is greater than y.
# Similar to the above example, since the second term is 0,
# itemset other than a is not selected.
> r2=(x>y).iif(x,0)
> r2.show
 3 a

# Compare x and y, select term(s) where the weight of x is the same weight as y.
> r3=(x==y).iif(x,0)
> r3.show
 2 b
```

See Also

==, <, <=, >, >=, ne? : Various comparison operations

3.28 import : Import ZDD

Format

obj.import(*fileName*) → *zdd*

fileName : string

Description

Import restores the serialised output of ZDD object *zdd* object previously saved in the ZDD file (defined at fileName) with the export method.

The declaration order of symbols during import must be defined in the same order during export.

Examples

Example 1: Basic Example

```
> 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")
# Contents in the export file dat.zdd are as follows.
# _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
```

Example 2: Example of correctly restored file

```
> require 'zdd'
# After sybmol is declared in order, import correctly restores the file.
> 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
```

Example 3: Example of incorrectly restored file

```
> require 'zdd'
# If item b,c are not declared in order, b and c will be represented incorrectly
# in the results.
> 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
```

Example 4: Restore file without symbol declaration

```
> require 'zdd'
# Item names such as x1,x2,x3 will be used if the symbols are not declared before import.
# In this case, the sequence number followed after x will be in reverse order
# corresponding to the declaration order of the item.
# In the following example, x1=c, x2=b, x3=c.
> g3=ZDD::import("dat.zdd")
> g3.show
 5 x3 x2 x1 + 3 x3 x2 + 2 x2 x1 + x1
```

See Also

export : Serialized output of ZDD

3.29 items : Aggregate weight of all items in ZDD

Format

obj.items → *zdd*

Description

Construct a new ZDD object *zdd* by aggregating the weights from items in ZDD object *obj*.

Examples

Example 1: Basic Example

```
> 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 object f is made up of 3 items a, b, c.
# The weight of each item is computed as follows.
# The terms "a b c" and "2 a b" contains item a, the total weight is 3.
# All terms contains item b, the total weight is 4.
# The terms "a b c" and "b c" contains item c, the total weight is 2.
> f.items.show
 2 a + 3 b + 2 c
```

See Also

3.30 itemset : Construct ZDD object for itemset

Format

obj.itemset(*is*) → *zdd*

is : string

Description

Constuct the ZDD object *zdd* for itemset specified at *is*. Multiple items can be delimited by half-width space. The weight do not need to be specified. Empty character ("") is treated as empty itemset. The sum of products of the constant term is set as 1 (same as zdd.constant(1)).

Items which are not declared in the symbol method will be added to the end of the item order table. The default value of item is set to 0.5. Refer to symbol for details on cost.

Examples

Example 1: Basic Example

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

# Numbers can be used as name of item
> x0=ZDD::itemset("2")
> x0.show
 2

# However, bear in mind that it  may be difficult to distinguish between weight and
# numerical item name.
> (2*x0).show
 2 2

# Symbols can be used as name of item
> x1=ZDD::itemset("!#%&'()=~|@[;:]")
> x1.show
 !#%&'()=~|@[;:]

# However, special symbols in Ruby must be escaped with a backward slash (\).
# In the following example, the 3 characters \,$," are escaped.
> x2=ZDD::itemset("\\\$\"")
> x2.show
 \$"

# Japanese characters can be used to name an item as well.
> x3=ZDD::itemset("りんご ばなな")
> x3.show
 りんご ばなな
```

See Also

symbol : Declare items

constant : Construct ZDD constant object

cost : Total cost of itemset

3.31 lcm : LCM over ZDD

Format

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

transaction : string

type : string

minsup : integer

order : string

ub : integer

Description

Using LCM over ZDD algorithm, enumerate frequent patterns above the specified minimum support *minsup* from the transaction file specified at *transaction*, and return as ZDD object *zdd*. Specify the ZDD item order file at *order*, and specify the upper limit of the itemset size at *ub*.

Three types of frequent patterns can be enumerated which is defined at *type* including "F" (frequent itemset), "M" (maximal itemset), "C" (closed itemsets).

Further, "FQ", attached with "Q", returns the frequencies and weight of each frequent itemset enumerated. When "F" is used without "Q", itemsets that meet the minimum support is returned without frequency information.

The transaction file as shown in the text file below, one row corresponds to one transaction. Items are specified by sequential numbers starting from 1, with a space delimiter between items.

Alphabet cannot be used as an item.

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

order file is a text file that shows the order of items registered in the ZDD item order table. Typically, all items contained in the transaction data are assigned to sequential numbers. In addition, note that when if there is a missing transaction item number, the missing number must be specified.

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

When *order* file is not specified (or specify as nil), the item order will be determined by the internal algorithm of LCM to increase efficiency.

However, this method, the item number is assigned sequentially in order, thus, the item number assigned to ZDD frequent itemset will be different than the original transaction number.

As long as the analysis is not related to the content of the item, it is computationally more efficient to exclude *order*. Conversely, if the purpose is to analyze the contents of the items, an order file as shown above should be specified.

Specify the upper limit for the size of frequent itemsets to be enumerated at *ub*. When the parameter is not specified, nil will be assigned when there is no maximum limit to the enumeration.

Examples

Example 1: Basic Example

```
> require 'zdd'
# Contents of tra.txt
# 1 2 3 6
# 4 5 6
# 1 2 4 6
# 2 4 6
# 1 2 4 5
# Contents of order.txt
# 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

# If order file is not specified, the resulting frequent itemset will be the same
# Note that the item number in the results is different from the item number
# in the transaction file.
> 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
```

See Also

freqpatA : Enumerate frequent itemsets

freqpatM : Enumerate maximal itemsets

freqpatC : Enumerate closed itemsets

3.32 maxcost : Return itemset with maximum cost

Format

obj.maxcost → *cost*

cost : float

Description

Among all itemsets in *obj*, return the one with maximum cost. The cost value of the itemset is obtained by maxcover.

Specify the cost with symbol function.

Example

Example 1: Basic Example

Refer to example in maxcover method.

See Also

symbol : Declare items

cost : Total cost of itemset

maxcover : Select itemset with maximum cost

mincover : Select itemset with minimum cost

mincost : Cost of itemset with minimum cost

3.33 maxcover : Select itemset with maximum cost

Format

obj.maxcover → *zdd*

Description

Among all itemsets in *obj*, return itemset with maximum cost as ZDD object *zdd*. Note that the weight is not taken into account when calculating the cost.

The value of cost is obtained by maxcost. Specify the cost with symbol function.

Example

Example 1: Basic Example

```
> require 'zdd'
# Set the cost of each item.
> 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
# Cost of a b =1.0+0.5=1.5
# Cost of b c =0.5+2.0=2.5
# Cost of a c =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
```

See Also

symbol : Declare items

cost : Total cost of itemsets

maxcost : Cost of itemset with maximum cost

mincover : Select itemset with minimum cost

mincost : Cost of itemset with minimum cost

3.34 maxweight : Value of maximum weight

Format

obj.maxweight → *zdd*

Description

Among all terms (including constant terms) contained in ZDD object *obj*, return the maximum weight as ZDD constant object.

Example

Example 1: Basic Example

```
> 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

# Obtain the maximum value including constant terms.
> x=5*a+3*b+c+10
> x.show
 5 a + 3 b + c + 10
> x.maxweight.show
 10

# Select the term with the maximum weight.
> 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
```

See Also

minweight : Value of minimum weight

totalweight : Sum of weight

3.35 meet : Intersection operation

Format

obj.meet($zdd1$) $\to$ $zdd2$

Description

Find out the intersection $\alpha \cap \beta$ of itemsets between itemsets α in *obj* and itemsets β in $zdd1$, and return as ZDD object $zdd2$.

For example, the intersection of itemset `abc` and `bcd` are as follows.

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

Find out the intersection of all combinations on multiple operations across itemsets.

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

Weight is computed by expanding the same itemset to multiple instances.

In addition, when $\alpha \cap \beta$ is changed to $\alpha \oplus \beta$ (exclusive OR operation), it becomes delta function.

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

Example

Example 1: Basic Example

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

# Find out the intersection of itemset a with each term in the expression a + 2ab + 3b,
# the result becomes a + 2a + 3 = 3 a + 3.
> f.meet(a).show
 3 a + 3

# The intersection with itemset b becomes 1 + 2b + 3b = 5b + 1.
> f.meet(b).show
 5 b + 1

# The intersection with itemset ab becomes a + 2ab + 3b.
> f.meet(a*b).show
 2 a b + a + 3 b

# Empty itemset is represented by constant number 1, thus the intersection with 1
# with all coefficients becomes 1 + 2 + 3 = 6.
> f.meet(1).show
 6

# Find out the interaction of each itemset in abc + 2ab + bc + 1 with each itemset
# in the specified argument 2ab + a as follows (remove space between items)
# abc ∩ 2ab = 2ab
# 2ab ∩ 2ab = 4ab
# bc  ∩ 2ab = 2b
# 1   ∩ 2ab = 2
# abc ∩ a   = a
# 2ab ∩ a   = 2a
# bc  ∩ a   = 1
# 1   ∩ a   = 1
# The result is summarized as 6ab + 3a + 2b + 4.
#
> 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
```

Related

delta : Delta operation

3.36 mincost : Cost of itemset with minimum cost

Format

obj.mincost → *cost*

cost : float

Description

Among all itemsets in *obj*, return the one with minimum cost. The cost value of the itemset is obtained by mincover.

Specify the cost with symbol function.

Example

Example 1: Basic Example

Refer to example in maxcover.

Refer to maxcover method.

See Also

symbol : Declare items

cost : Compute cost of itemset

maxcover : Select itemset with maximum cost

maxcost : Cost of itemset with maximum cost

mincover : Select itemset with minimum cost

3.37 mincover : Select itemset with minimum cost

Format

obj.mincover → *self*

Description

Among all itemsets in *obj*, return the itemset with the minimum cost. The cost value is obtained by mincost Specify the cost with symbol function.

Example

Example 1: Basic Example

Refer to example in maxcover method.

See Also

symbol : Declare items

cost : Total cost of itemsets

maxcover : Select itemset with maximum cost

maxcost : Cost of itemset with maximum cost

mincost : Cost of itemset with minimum cost

3.38 minweight : Value of minimum weight

Format

obj.minweight → *zdd*

Description

Among all terms (including constant) contained in *obj*, return the minimum weight of ZDD constant object to *zdd*.

Example

Example 1: Basic Example

```
> 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

# The maximum constant term is returned.
> x=5*a-3*b+c-10
> x.show
 5 a - 3 b + c - 10
> x.minweight.show
 - 10

# Select the term with the maximum weight.
> 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
```

See Also

maxweight : Value of maximum weight

totalweight : Sum of weight

3.39 ne? : Comparison of inequality operation

Format

obj.ne?($zdd1$) $\rightarrow$ $zdd2$

Description

Compare itemsets include in obj with the itemsets included in $zdd1$, select the itemsets with a different weight.

Example

Example 1: Basic Example

Refer to example in ==.

See Also

==, <, <=, >, >=, ne? : Various comparison operations

3.40 partly : Partial hash output flag

Format

obj.partly → *bool*

Description

This method returns true if hashout method did not output all data set. Refer to hashout method for more details.

Example

Example 1: Basic Example

```
> 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
```

See Also

hashout : Hash output

3.41 permit : Selection of subset

Format

obj.permit(*zdd*1) → *zdd*2

Description

For itemsets included in *obj*, select the term that is included in at least 1 itemset contained in *zdd*1.

More precisely, α_i is the set of items that make up *obj* where the weight of term T_i is removed, assuming that β_j represents the same itemsets in *zdd*1, if at least one j satisfy $\alpha_i \subseteq \beta_j$, select term T_i that corresponds to α_i from *obj*.

In relation, the conditional expression $\alpha_i \subseteq \beta_j$ is changed to $\alpha_i \supseteq \beta_j$ and become restrict function.

Example

Example 1: Basic Example

```
> 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

# 4 itemsets a,bc,b, Φ (the term for empty itemset has a weight of 2) in x,
# 3 itemsets a,b,d in y are included in itemset
# a and b and Φ (empty itemset is also considered for inclusion as itemset).
# Therefore, select terms a,b, Φ from x.
> x.permit(y).show
 5 a + 3 b + 2

# Among 4 itemsets a,bc,b, Φ contains in x,
# itemset z contains ac, which includes itemset a and Φ.
# Therefore, terms a and Φ are selected from x.
> x.permit(z).show
 5 a + 2

# Among 4 itemsets a,bc,b, Φ contain in x,
# itemset c includes itemset Φ.
# Therefore, term Φ is selected from x.
> x.permit(c).show
 2
```

See Also

restrict : Selection of superset

3.42 permitsym : Select itemsets by number of items

Format

obj.permitsym(*zdd*1) → *zdd*2

Description

Among all itemsets that make up ZDD object *obj*, select the itemsets containing the items equal to or below the number defined in *zdd*1, and return as ZDD object *zdd*2.

Example

Example 1: Basic Example

```
> 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

# Select itemsets with less than or equal to 1 item
> x.permitsym(1).show
 5 a + 3 b + 2

# Select itemsets with less than or equal to 2 items
> x.permitsym(2).show
 5 a + b c + 3 b + 2

# Select itemsets without any item (that is empty itemsets)
> x.permitsym(0).show
 2
```

See Also

3.43 restrict : Selection of superset

Format

obj.restrict(*zdd*1) → *zdd*2

Description

For itemsets included in *obj*, select the term as long as *zdd*1 contains at least one itemset.

More precisely, α_i is the set of items that make up *obj* where the weight of term T_i is removed, assuming that β_j represents the same itemsets in *zdd*1, if at least one j satisfy $\alpha_i \supseteq \beta_j$, select term T_i that corresponds to α_i from *obj*.

In relation, the conditional function $\alpha_i \supseteq \beta_j$ is changed to $\alpha_i \subseteq \beta_j$ to become permit function.

Example

Example 1: Basic Example

```
> 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

# Among 4 itemsets a,bc,b, Φ (empty itemset has a weight of 2) in x,
# 2 itemsets a,b in y included in any of the itemset are a,bc.
# Therefore terms a,bc are selected from x.
> x.restrict(a+c).show
 5 a + b c

# Among 4 itemsets a,bc,b, Φ in x,
# itemset z only includes itemset bc.
# Therefore term bc is selected from x.
> x.restrict(b*c).show
 b c

# Among 4 itemsets a,bc,b, Φ in x,
# all itemsets containing itemset Φ (empty itemsets with weight of 1) is in a set of items.
> x.restrict(1).show
 5 a + b c + 3 b + 2

# Among 4 itemsets a,bc,b, Φ in x, there is no itemset contained in itemset abc.
> x.restrict(a*b*c).show
 0
```

See Also

permit : Selection of subset

3.44 same? : Same comparison operator

Format

obj.same?(*zdd*) → *bool* → *bool*

obj === *zdd* → *bool*

Description

Compare 2 ZDD objects *obj* and *zdd*, returns true if the operands are the same, and false if they are different.

Two "=" operator (== operator) has a different definition, it is used to carry out equal to comparison of the terms within the expression, thus, take note that this operator is completely different.

Example

Example 1: Basic Example

```
> 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
```

See Also

diff? : Not equal to comparison operator

3.45 show : Display ZDD

Format

obj.show([*switch*])

switch : string

Description

Display various format of ZDD object*obj* through standard output. Output format converts the given value (character string) as shown in Table 3.1. If *switch* is not specified, items will be displayed in sum-of-products format.

Table 3.1: List of switches for ZDD object display format

Value and function of *switch*	
(No switch)	Display item's sum-of-products
bit	Display the weight in (-2) base for each digit by itemset
hex	Display integer value sum-of-products in hexadecimal digits
map	Display in Karnaugh map. The map displays up to 6 item variables
rmap	Display in Karnaugh map. Redundant item variables are removed
case	Display case analysis of sum of products for each integer value
decomp	Display simple disjoint decomposition format

Example

Example 1: Basic Example

```
> 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
```

Example 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/stephane/.rvm/rubies/ruby-1.9.3-p448/bin/irb:16:in ‘<main>’

# "a b c" has weight of -5, expressed in (-2) base is 101.
# 1*(-2)^2+0*(-2)^1+1*(-2)^0 = 5
# Therefore 0 digit and 2nd digit of itemset "a b c" is display.
# "a b" has weight of -3 expressed in (-2) base is 1101.
# 1*(-2)^3+1*(-2)^2+0*(-2)^1+1*(-2)^0 = -3
# Therefore 0,2nd,3rd digit of itemset "a b" is displayed.
```

Example 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/stephane/.rvm/rubies/ruby-1.9.3-p448/bin/irb:16:in ‘<main>’
```

Example 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/stephane/.rvm/rubies/ruby-1.9.3-p448/bin/irb:16:in ‘<main>’
# Itemset is displayed with item a as the first sequence in the bit string,
# and item b corresponds to the first row of the bit string.
# Weight is displayed as the cell value. Upper left cell contains 0 in a, and 0 in b,
# and constant term 4 is shown.

# The 4 items are as follows.
> 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):17
        from /Users/stephane/.rvm/rubies/ruby-1.9.3-p448/bin/irb:16:in ‘<main>’
```

Example 5: rmap

```
> require 'zdd'
# Declare 4 items a,b,c,d
> 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

# The following displays as map.
> f.map
NoMethodError: undefined method ‘map’ for a b + 2 b c + 4 :Module
        from (irb):12
        from /Users/stephane/.rvm/rubies/ruby-1.9.3-p448/bin/irb:16:in ‘<main>’
```

```
# d is omitted when displayed as rmap.
> f.rmap
NoMethodError: undefined method ‘rmap’ for a b + 2 b c + 4 :Module
        from (irb):15
        from /Users/stephane/.rvm/rubies/ruby-1.9.3-p448/bin/irb:16:in ‘<main>’
```

Example 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/stephane/.rvm/rubies/ruby-1.9.3-p448/bin/irb:16:in ‘<main>’
```

Example 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/stephane/.rvm/rubies/ruby-1.9.3-p448/bin/irb:16:in ‘<main>’
# AND of a,b,c is expressed as 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/stephane/.rvm/rubies/ruby-1.9.3-p448/bin/irb:16:in ‘<main>’
# OR of c,1 is expressed as (c+1), in addition, AND of a b is expressed as (a b),
# the complete expression becomes (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):19
        from /Users/stephane/.rvm/rubies/ruby-1.9.3-p448/bin/irb:16:in ‘<main>’
# [ a c ] enumerates all combinations from a and c as (a c + a + c).
# AND of b and the above expression becomes 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):25
        from /Users/stephane/.rvm/rubies/ruby-1.9.3-p448/bin/irb:16:in ‘<main>’
# [ a b c ] enumerates all combinations from a,b,c as (a b c + a b + b c + c a)
```

See Also

export : Save the structure of ZDD in a file.

3.46 size : Number of ZDD nodes

Format

obj.size → *nodeSize*

nodeSize : integer

Description

Returns the number of nodes in *obj*.

Example

Example 1: Basic Example

```
> 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
```

See Also

totalsize : Number of ZDD nodes in the processing system

3.47 symbol : Declare item

Format

Format 1)

obj.symbol$(itemName, value, to) \rightarrow Qtrue$

$itemName$: string

$value$: float

to : string

Format 2)

obj.symbol $\rightarrow itemList$

$itemList$: string

Description

The 3 features of the symbol function are as follows.

1. Specify the variable (item) in corresponding order from the root of the binary decision graph to the end node in the ZDD structure. 1 item can be declared by the symbol function each time, and can be added to the end or beginning of the item table stored internally.
2. Set name of item. Any characters can be used in item name.
3. Set the cost of item attribute.

Format 1)

Declare the name of item at *itemName*. There is no particular restriction to the character type for item name nor the length of character string (except for limitation in memory capacity).

value assigns cost to an item used in cost and maxcover methods. When not specified, the default value is set at 0.5.

to specifies whether to add the declared item to the top or bottom of the item order table ("top" or "bottom"). When the value is not specified, default value is set to "bottom".

Format 2)

When the symbol method is used without arguments, it returns a list of space delimited symbol variables.

Example

```
> 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
```

See Also

itemset : Create ZDD objects for itemsets.

cost : Calculate total cost of itemsets.

3.48 termsEQ : Select terms by weight comparison (equal to comparison)

Format

obj.termsEQ(*zdd*1) → *zdd*2

Description

Among the terms in *obj*, select the terms (weight + itemset) where the weight is the same as the constant defined at *zdd*1.

*zdd*1 is a ZDD constant object generated by the constant method, or specified Ruby integer.

The methods used in this package are as follows.

- zdd1.termsEQ(zdd2) : equal to comparison
- zdd1.termsGE(zdd2) : greater than or equal to comparison
- zdd1.termsGT(zdd2) : greater than comparison
- zdd1.termsLE(zdd2) : less than comparison
- zdd1.termsLT(zdd2) : less than or equal to comparison
- zdd1.termsNE(zdd2) : not equal to comparison

Example

Example 1: Basic Example

```
> 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

# Select terms with weight of 3.
> f.termsEQ(3).show
 3 b

# Select terms with weight greater than or equal to 3.
> f.termsGE(3).show
 5 a + 3 b

# Select terms with weight not equal to 3.
> f.termsNE(3).show
 5 a + c

# Return 0 if none of the terms herein matches the condition.
> f.termsGT(10).show
 0
```

See Also

termsGE : Select terms by weight comparison (greater than or equal to comparison)

termsGT : Select terms by weight comparison (greater than comparison) termsLE : Select terms by weight comparison (less than or equal to comparison)

termsLT : Select terms by weight comparison (less than comparison)

termsNE : Select terms by weight comparison (not equal to comparison)

3.49 termsGE : Select terms by weight comparison (greater than or equal to comparison)

Format

obj.termsGE(*zdd*1) → *zdd*2

Description

Among the terms in *obj*, select the terms (weight + itemset) where the weight is greater than or equal to the constant defined at *zdd*1.

Example

Example 1: Basic Example

termsEQ Refer to example of the method.

See Also

termsEQ : Select terms by weight comparison (equal to comparison)

termsGT : Select terms by weight comparison (greater than comparison)

termsLE : Select terms by weight comparison (less than or equal to comparison)

termsLT : Select terms by weight comparison (less than comparison)

termsNE : Select terms by weight comparison (not equal to comparison)

3.50 termsGT : Select terms by weight comparison (greater than comparison)

Format

obj.termsGT(*zdd*1) → *zdd*2

Description

Among the terms in *obj*, select the terms (weight + itemset) where the weight is greater than the constant defined at *zdd*1.

Example

Example 1: Basic Example

termsEQ Refer to example of the method.

Related

termsEQ : Select terms by weight comparison (equal to comparison)

termsGE : Select terms by weight comparison (greater than or equal to comparison)

termsLE : Select terms by weight comparison (less than or equal to comparison)

termsLT : Select terms by weight comparison (less than comparison)

termsNE : Select terms by weight comparison (not equal to comparison)

3.51 termsLE : Select terms by weight comparison (less than or equal comparison)

Format

obj.termsLE(*zdd*1) → *zdd*2

Description

Among the terms in *obj*, select the terms (weight + itemset) where the weight is less than or equal to the constant defined at *zdd*1.

Example

Example 1: Basic Example

termsEQ Refer to example of the method.

Related

termsEQ : Select terms by weight comparison (equal to comparison)

termsGE : Select terms by weight comparison (greater than or equal to comparison)

termsGT : Select terms by weight comparison (greater than comparison)

termsLT : Select terms by weight comparison (less than comparison)

termsNE : Select terms by weight comparison (less than or equal to comparison)

3.52 termsLT : Select terms by weight comparison (less than comparison)

Format

obj.termsLT(*zdd*1) → *zdd*2

Description

Among the terms in *obj*, select the terms (weight + itemset) where the weight is less than the constant defined at *zdd*1.

Example

Example 1: Basic Example

termsEQ Refer to example of the method.

See Also

termsEQ : Select terms by weight comparison (equal to comparison)

termsGE : Select terms by weight comparison (greater than or equal to comparison)

termsGT : Select terms by weight comparison (greater than comparison)

termsLE : Select terms by weight comparison (less than or equal to comparison)

termsNE : Select terms by weight comparison (less than or equal to comparison)

3.53 termsNE : Select terms by weight comparison (not equal to comparison)

Format

obj.termsNE(*zdd*1) → *zdd*2

Description

Among the terms in *obj*, select the terms (weight + itemset) where the weight is not equal to the constant defined at *zdd*1.

Example

Example 1: Basic Example

termsEQ Refer to example of the method.

See Also

termsEQ : Select terms by weight comparison (equal to comparison)

termsGE : Select terms by weight comparison (greater than or equal to comparison)

termsGT : Select terms by weight comparison (greater than comparison)

termsLE : Select terms by weight comparison (less than or equal to comparison)

termsLT : Select terms by weight comparison (less than comparison)

3.54 to_a : Convert to itemset array

Format

obj.to_a → *array*

Description

Return character string array *array* of itemset from ZDD object *obj*.

Example

Example 1: Basic Example

```
> 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"]
```

See Also

hashout : Output Hash

to_s : Convert to sum-of-products string

3.55 to_i : Convert ZDD constant object to Ruby integer

Format

obj.to_i() → *constant*

constant : integer

Description

Convert ZDD constant object *obj* (weight of empty itemset) to Ruby integer *constant*. Return nil if the weight object is not a ZDD object.

Example

Example 1: Basic Example

```
> require 'zdd'
> c=ZDD::constant(10)
> c.show          # ZDD constant object
 10
> puts c.to_i     # ruby integer
10

# Return nil if not ZDD constant object.
> a=ZDD::itemset("a")
> p a.to_i
nil
```

See Also

constant : Create ZDD constant object

to_a : Convert to itemset array

3.56 to_s : Convert to sum-of-products string

Format

odj.to_a → *string*

Description

Return sum-of-products string *string* from ZDD object *obj*.

Example

Example 1: Basic Example

```
> 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"
```

See Also

to_a : Conversion to itemset array

hashout : Output Hash

3.57 totalsize : Number of ZDD nodes in the processing system

Format

ZDD::totalsize → *nodeSize*

nodeSize : integer

Description

Return the number of nodes from all ZDD objects constructed by Ruby interpreter.

Examples

Example 1: Basic Example

```
> 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
10

> b=-3*ZDD::itemset("a c")
> b.show
 - 3 a c
> puts b.size
5
> puts ZDD::totalsize
14
```

See Also

size : Number of ZDD nodes

3.58 totalweight : Sum of weights

Format

obj.totalweight → *total*

total : integer

Description

Return the sum of weights contained in *obj* (including constant term) as Ruby integer *total*.

Example

Example 1: Basic Example

```
> 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
```

See Also

minweight : Minimum value of weights

maxweight : Maximum value of weights

Bibliography

[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.