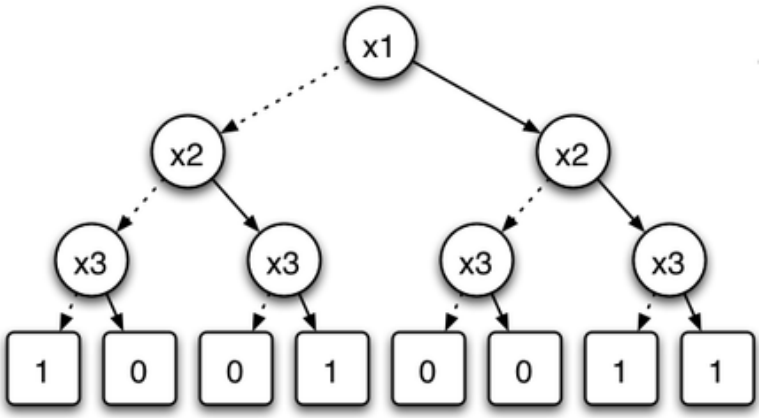


Binary Decision Diagram

Mid project report

Ben-Gurion University of the Negev

Faculty of Engineering Science

Dept. of Communication Systems Engineering

Functional Programming in Concurrent and Distributed Systems

381-1-0112

Spring 2019

Maor Assayag 318550746

Table of contents

Abstract

In this assignment, i implemented in **Erlang** an automatic construction machine of a Binary Decision Diagram (BDD) to represent a Boolean function, so a user is able to get a BDD representation of a function within a single call to your machine.

BDD is a tree data structure that represents a Boolean function. The search for a Boolean result of an assignment of a Boolean function is performed in stages, one stage for every Boolean variable, where the next step of every stage depends on the value of the Boolean variable thats represented by this stage.

A BDD tree is called reduced if the following two rules have been applied to it:

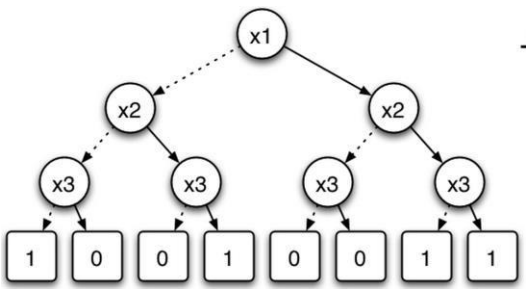
1. Merge any isomorphic (identical) sub-graphs
2. Eliminate any node whose two children are isomorphic

The construction of BDD is based on **Shannon expansion theory**:
$f(x1, x2, \dots, xn) = x1 \cdot f(1, x2, x3, \dots, xn) + \overline{x1} \cdot f(0, x2, x3, \dots, xn)$

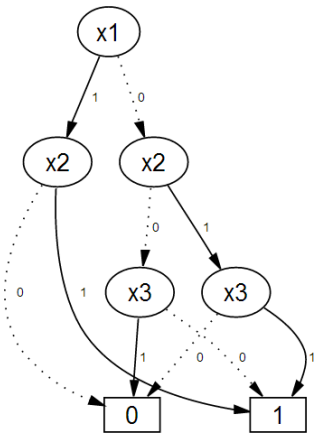
Example :
The Boolean function $f(x1, x2, x3) = \overline{x1}\,\overline{x2}\,\overline{x3} + x1\,x2 + x2\,x3$ with the following truth table:

x1	x2	x3	f
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

The BDD tree representation for this Boolean function:



By applying the reduction rules, we get:



Keywords: BDD(Binary Decision Diagram), Shannon expansion theory, Erlang

System design

Exported functions

The system will consist of 2 main functions that will be exported from the module **'exf_318550746.erl'** : exp_to_bdd & solve_bdd.

exp_to_bdd/2

Description

exp_to_bdd(BoolFunc, Ordering) → BddTree.

a. The function receives a Boolean function and returns the corresponding BDD tree representation for that Boolean function.
b. The returned tree must be the one that is the most efficient in the manner specified in variable **Ordering**.
    1. Variable **Ordering** can be one of the following atoms: **tree_height**, **num_of_nodes** or **num_of_leafs**.
    2. In order to extract the most efficient tree you should **Compare all the possible permutations** of BDD trees.
    3. *Permutations:* let's assume that you are asked to convert a Boolean function that has 3 Boolean arguments: $f_S(x_1, x_2, x_3)$. f_S could be expanded in 3! different ways which means 6 BDDs. each BDD is a result of Shannon expansion applied to variables in distinct order. For f_S all the possible permutations are:
    $\{\{x_1, x_2, x_3\}, \{x_1, x_3, x_2\}, \{x_2, x_1, x_3\}, \{x_2, x_3, x_1\}, \{x_3, x_2, x_1\}, \{x_3, x_1, x_2\}\}$
c. The returned tree must be reduced by merging any isomorphic (identical) sub-graphs.

```
%% The function receives a Boolean function and returns the corresponding BDD tree
%% representation for that Boolean function.
%% Variable Ordering can be one of the following atoms: tree_height,
%% num_of_nodes or num_of_leafs.
exp_to_bdd(BoolFunc, Ordering) ->
  % capture the inital time stamp
  InitTime = getTime(),
  Ans = getOptimumBDD(getBDDS(BoolFunc, getPermutations(getParameterList(BoolFunc))), Ordering),
  io:fwrite(io_lib:format("Finished in ~p ns\n", [getTime()-InitTime])),
  Ans.
```

Algorithm Flow

Note : the algorithm example will be performed on the following Boolean function :

$$f_r(x_1, x_2, x_3, x_4) = x_1\,\overline{x_2}\,x_3 + \overline{x_1\overline{x_3}(\overline{x_4} + x_2)} + \overline{x_4 x_1}$$

In addition, any aid function will be fully described in *Aid Functions* section.

1. The Boolean input expression *BoolFunc* is been passed to the function **getParameterList**(*BoolFunc*).
   The results will be a list of the parameters (atoms), e.g. result : [x1, x2, x3, x4].

2. Next the parameters list is been passed to the function **getPermutations**(List), which generates all permutations of the elements in List.
   e.g. result : [[x1,x2,x3,x4], [x1,x2,x4,x3], ⋯, [x4,x3,x2,x1].

3. **getBDDS**(*BoolFunc, PermutationList*) returns a BDDS list, each BDD corresponds to each Parameters permutation.
   Each element in the list is a **Tuple** constructed as follow : *{ {Parameter, Left, Right}, Hight, NumOfNodes, NumOfLeafs}* when *{Parameter, Left, Right}* is also a **Tuple** represents a BDD as defined in Input Definition section.

4. **getOptimumBDD**(*BDDS, Ordering*) returns the optimum BDD tree (a **Tuple**) from the BDDS list according to the Ordering specified.

5. The optimum BDD tree Tuple is been returned from the function, as well io:fwrite to the console the overall time this operation took using **getTime**().

   - Running on the example in Boolean Function form :

     {'or',
     {{'or',{{'and',{x1,{'and',{{'not',x2},x3}}}},{'not',{'and',{x4,x1}}}}},
     {'not',{'and',{{'and',{x1,{'not',x3}}},{'or',{{'not',x4},x2}}}}}}}

     I got that all the permutation produces BDD with the same Height, num of leaf's and num of nodes.

```
6> exf_318550746:getBDDS({'or',{{'or',{{'and',{x1,{'and',{{'not',x2},x3}}}},{'no
t',{'and',{{'and',{x1,{'not',x3}}},{'or',{{'not',x4},x2}}}}}}},  {'not',{'and',{
x4,x1}}}}},exf_318550746:getPermutations([x1,x2,x3,x4])).
[{{x4,1,{x3,{x2,1,{x1,1,0}},1}},3,4,1},
 {{x4,1,{x3,{x1,1,{x2,1,0}},1}},3,4,1},
 {{x4,1,{x2,1,{x3,{x1,1,0},1}}},3,4,1},
 {{x4,1,{x2,1,{x1,1,{x3,0,1}}}},3,4,1},
 {{x4,1,{x1,1,{x3,{x2,1,0},1}}},3,4,1},
 {{x4,1,{x1,1,{x2,1,{x3,0,1}}}},3,4,1},
 {{x3,{x4,1,{x2,1,{x1,1,0}}},1},3,4,1},
 {{x3,{x4,1,{x1,1,{x2,1,0}}},1},3,4,1},
 {{x3,{x2,1,{x4,1,{x1,1,0}}},1},3,4,1},
 {{x3,{x2,1,{x1,1,{x4,1,0}}},1},3,4,1},
 {{x3,{x1,1,{x4,1,{x2,1,0}}},1},3,4,1},
 {{x3,{x1,1,{x2,1,{x4,1,0}}},1},3,4,1},
 {{x2,1,{x4,1,{x3,{x1,1,0},1}}},3,4,1},
 {{x2,1,{x4,1,{x1,1,{x3,0,1}}}},3,4,1},
 {{x2,1,{x3,{x4,1,{x1,1,0}},1}},3,4,1},
 {{x2,1,{x3,{x1,1,{x4,1,0}},1}},3,4,1},
 {{x2,1,{x1,1,{x4,1,{x3,0,...}}}},3,4,1},
 {{x2,1,{x1,1,{x3,{x4,1,...},1}}},3,4,1},
 {{x1,1,{x4,1,{x3,{x2,...},1}}},3,4,1},
 {{x1,1,{x4,1,{x2,1,...}}},3,4,1},
 {{x1,1,{x3,{x4,1,...},1}},3,4,1},
 {{x1,1,{x3,{x2,...},1}},3,4,1},
 {{x1,1,{x2,1,...}},3,4,1},
 {{x1,1,{x2,...}},3,4,1}]
```

Figure 1.1 – results from getBDDS

This is due to the fact that the Boolean table is always 1 except a single input that produce 0, forcing every BDD tree to have a node for each variable.

a	b	c	d	X
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	0
1	1	1	0	1
1	1	1	1	1

Table 1.1 – Logical table for the example Boolean function

In **getOptimumBDD** I didn't change the current optimum tree if the deciding value is the same, resulting the result of this specific function (in every *Ordering*) **to differ according to the permutation order** (which can be affected from the Identification process of the variable in the Boolean expression.

solve_bdd/2

Description

solve_bdd(*BddTree, [{x1,Val1},{x2,Val2},{x3,Val3},{x4,Val4}]*) → Res.

The function receives a BDD tree and a list of values for every Boolean variable thats used in the Boolean function and returns the result of that function, according to the given BDD tree.

Given values may be either in the form of true/false or 0/1. The list of variables' values (the second argument of solve_bdd function) could be given at any order. Therefore, the function should be capable to handle any given order.

```
%% The function receives a BDD tree and a list of values for every Boolean variable that's used in the
%% Boolean function and returns the result of that function, according to the given BDD tree.
%% Given values may be either in the form of true/false or 0/1.
solve_bdd(Bdd,List) ->
  InitTime = getTime(),
  Ans = solve_bdd_tree(Bdd,List),
  io:fwrite(io_lib:format("Finished in ~p ns\n", [(getTime()-InitTime)])),
  Ans.
```

Algorithm Flow

Note : the algorithm example will be performed on the following Boolean function :

$$f_r(x_1,x_2,x_3,x_4) = x_1\,\overline{x_2}\,x_3 + \overline{x_1\overline{x_3}(\overline{x_4}+x_2)} + \overline{x_4 x_1}$$

In addition, any aid function will be fully described in *Aid Functions* section.

1. The inputs are passed to the aid function **solve_bdd_tree,** which travel on the BDD tree according to the Tuple lists of {Param,Value} –

if a Param equal 1 we go right (to the right child), 0 we go left until we get to a leaf (a binary value).

2. The logical result (0 or 1) are been returned with additional indication on how much time this operation took to solve in ns.

   - Running exp_to_bdd on the example in Boolean Function form

     {'or',
     {{'or',{{'and',{x1,{'and',{{'not',x2},x3}}}},{'not',{'and',{x4,x1}}}}},
     {'not',{'and',{{'and',{x1,{'not',x3}}},{'or',{{'not',x4},x2}}}}}}}

     yields a several BDDs, choosing one of them for e.g. :

```
3> exf_318550746:exp_to_bdd({'or',{{'or',{{'and',{x1,{'and',{{'not',x2},x3}}}},{
'not',{'and',{{'and',{x1,{'not',x3}}},{'or',{{'not',x4},x2}}}}}},  {'not',{'and
',{x4,x1}}}}},tree_height).
Finished in 1024 ns
{x4,1,{x3,{x1,1,{x2,1,0}},1}}
```

     Figure 1.1 – results from exp_to_bdd

     {x4,1,{x3,{x1,1,{x2,1,0}},1}}

     Computing the only input to produce 0 :

```
4> exf_318550746:solve_bdd({x4,1,{x3,{x1,1,{x2,1,0}},1}},[{x1,1},{x2,1},{x3,0},{
x4,1}]).
Finished in 0 ns
0
```

     And another input sequence :

```
5> exf_318550746:solve_bdd({x4,1,{x3,{x1,1,{x2,1,0}},1}},[{x2,1},{x3,1},{x1,0},{
x4,0}]).
Finished in 0 ns
1
```

a	b	c	d	x
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	0
1	1	1	0	1
1	1	1	1	1

Table 1.1 – Logical table for the example Boolean function

Aid functions

Note : the algorithm examples will be performed on the following Boolean function :

$$f_r(x_1,x_2,x_3,x_4) = x_1\,\overline{x_2}\,x_3 + \overline{x_1\overline{x_3}(\overline{x_4}+x_2)} + \overline{x_4}\overline{x_1}$$

removeDuplicates/1

Description

removeDuplicates(*List*) simply remove duplicate occurrence from a list, used to construct a simple variables list from a List that potentially having multiple occurrence of variables.

e.g. : input = [x1, x2, x3, x1, x3, x4, x2, x4, x1], output = [x1, x2, x3, x4]

getPermutations/1

Description

removeDuplicates(*List*) generates all permutations of the elements in a list

e.g. : input = [x1, x2, x3, x4]

```
6> exf_318550746:getPermutations([x1,x2,x3,x4]).
[[x1,x2,x3,x4],
 [x1,x2,x4,x3],
 [x1,x3,x2,x4],
 [x1,x3,x4,x2],
 [x1,x4,x2,x3],
 [x1,x4,x3,x2],
 [x2,x1,x3,x4],
 [x2,x1,x4,x3],
 [x2,x3,x1,x4],
 [x2,x3,x4,x1],
 [x2,x4,x1,x3],
 [x2,x4,x3,x1],
 [x3,x1,x2,x4],
 [x3,x1,x4,x2],
 [x3,x2,x1,x4],
 [x3,x2,x4,x1],
 [x3,x4,x1,x2],
 [x3,x4,x2,x1],
 [x4,x1,x2,x3],
 [x4,x1,x3,x2],
 [x4,x2,x1,x3],
 [x4,x2,x3,x1],
 [x4,x3,x1,x2],
 [x4,x3,x2,x1]]
```

getParameterList/1

Description

getParameterList(*BooleanFunc*) gets the parameters (atoms) from the Boolean function.

e.g. : input = {'or',
{{'or',{{'and',{x1,{'and',{{'not',x2},x3}}}},{'not',{'and',{x4,x1}}}}},
{'not',{'and',{{'and',{x1,{'not',x3}}},{'or',{{'not',x4},x2}}}}}}}

Output = [x2, x1, x3, x4]

```
7> exf_318550746:getParameterList({'or',{{'or',{{'and',{x1,{'and',{{'not',x2},x3
}}}},{'not',{'and',{{'and',{x1,{'not',x3}}},{'or',{{'not',x4},x2}}}}}}},  {'not'
,{'and',{x4,x1}}}}}).
[x2,x1,x3,x4]
```

Algorithm flow

1. Using recursion start 'peeling' inside the Tuples in the Boolean Function. In the current Tuple If the first element is the operation 'not' check if the second element is a Tuple, if yes call this function again only on the second element, otherwise add it to the accumulate List result. The same logic is applied to the rest of the operation 'and' & 'or'.

2. The current List results is been passed to RemoveDuplicates and the results of that is returned as the final reduced Parameter List.

evaluate/1

Description

evalute(*{Op, A}*) gets an expression and tries to evaluate it (minimize it) as much as possible. The main idea is to identify cases when the expression *A* can be minimized due to the logical characteristics of the Boolean operations.
One case is when *A* is a ***Tuple*** consist of one number that can determine the expression results without computing the other elements in the *Tuple*
e.g. : input = {'or',{x1, 1}}
Output = {1}
In addition, a ***Tuple*** *A* consist of the same 2 elements can be reduced etc.
This function mainly used after replacing a Parameter with value when using Shannon expansion theorem.

replace/3

Description

replace(*X, Y, A*) replace occurrences of *X* to *Y* in *A* using **recursion** to maintain the integrity of the expression. This function supports *A* as **Tuples** as defined in the Input section below.

e.g. input = x1, 1, {'or',
{{'or',{{'and',{x1,{'and',{{'not',x2},x3}}}},{'not',{'and',{x4,x1}}}}},
{'not',{'and',{{'and',{x1,{'not',x3}}},{'or',{{'not',x4},x2}}}}}}}

Output = {'or', {{'or',{{'and',{1,{'and',{{'not',x2},x3}}}},{'not',{'and',{x4,1}}}}},
{'not',{'and',{{'and',{1,{'not',x3}}},{'or',{{'not',x4},x2}}}}}}}

```
8> exf_318550746:replace(x1,1,{'or',{{'or',{{'and',{x1,{'and',{{'not',x2},x3}}}}
,{'not',{'and',{{'and',{x1,{'not',x3}}},{'or',{{'not',x4},x2}}}}}}},  {'not',{'a
nd',{x4,x1}}}}}}).
{'or',{{'or',{{'and',{1,{'and',{{'not',x2},x3}}}},
              {'not',{'and',{{'and',{1,{'not',x3}}},
                             {'or',{{'not',x4},x2}}}}}}},
   _   {'not',{'and',{x4,1}}}}}
```

getNode/3

Description

getNode(*H, R, L*) creates a Node in the BDD from *{ {Parameter, Left, Right}, Hight, NumOfNodes, NumOfLeafs}*. A node in the BDD tree is {Parameter, Left, Right} and the tree is been construct from the leaf's.

Each node need to be checked for special characteristics : if both child's is an equal value/expression then the current node is a redundant node, if both child's is a value then the current node is a leaf, and if not then construct the current nodes with both child's info **to be passed on up to the root.** The final tree will consist only from the Parameter left/right children's data, and the overall values for the *Ordering* will be passed alongside the BDD tree.

Node info :
{ {Parameter, Left, Right}, Hight, NumOfNodes, NumOfLeafs}

- The Parameter left/right children's
- Tree height = the max height of left and right + 1
- Num of nodes = Num of nodes in the right, in the left + 1 for the current node
- Num of leaf's = Num of leaf's in the right + in the left

getTree/2

Description

getTree(*BooleanFunc, ParameterList*) constructs a tree from a Boolean function *BooleanFunc*, each node represents a Parameter and forked values to the other parameters using Shannon expansion theorem described in the abstract. This method construct the tree recursively from the bottom (leaf's with logical values) up to the root while summing all the information necessary for characterize the BDD tree.

Algorithm flow

1. We think about each parameter in the Parameter List as a new Node. For each Parameter, we need to compute the Right children and the Left children of the tree.
   If the Boolean Function received is not a logical value (0 or 1), do as follow :
   a. Construct the Right child :
      a. *Replace* the Parameter H in the right side to the value 1
      b. *Evaluate* (minimize) the Boolean function again
      c. Call *getTree* on the new minimize expression with the rest of the parameters (the Tail of Parameter List)
   b. Construct the Left child :
      a. *Replace* the Parameter H in the right side to the value 0
      b. *Evaluate* (minimize) the Boolean function again
      c. Call *getTree* on the new minimize expression with the rest of the parameters (the Tail of Parameter List)
   c. Call *getNode* to create a node from the current Parameter *H*, the Right child node & the Left child node.

   Otherwise, we are at a leaf and return a simple Node represent a {logical value, 0 , 0, 0} due to the fact that a leaf does not have children's so it's not carrying additional information.

2. The node that will returned to the function is the root node (which represent the first Parameter in the Parameter List) along side with the BDD characteristic as Hight, NumOfNodes, NumOfLeafs.

getBDDS/2

Description

getBDDs(*BooleanFunc, Permutations*) Return BDDs list, each one corresponds to each Parameters permutation. This method is simply iterating through the *Permutations Parameter Lists* and calling **getTree** with he *BooleanFunc* and the specific Permutation, thus all together forms a List of BDDs tree, each one corresponds to a unique *Permutation* of the *Parameter List*.

getOptimumBDD /2

Description

getOptimumBDD (*BDDs, Ordering*) returns the optimum BDD tree from *BDDs* list according to the *Ordering* specified . Due to the design of the root Node in each BDD tree, we can simply access the Ordering information of each tree as the 2,3, and 4 element in the Tree Tuple correspond to the Hight, NumOfNodes, NumOfLeafs. This method using getOptimumBDD/3 which iterating through the BDDs list to find the minimum *Ordering* value in the tuple, using *element native* method.
Only smaller value will be chosen to represent a new optimum solution.

getTime/0

Description

getTime( ) Return the current Time stamp using *os : system_time(nanosecond)* in nanosecond time resolution.

solve_bdd_tree/2

Description

solve_bdd_tree (*Node, TupleList*) solves the BDD tree according to the Tuple lists of *{Param,Value}*. Essentially this method gets the Value of each current parameter (using **getValue**) in the BDD tree from the Tuple list and travel to the right direction in the tree using recursion on the Node children.
e.g. TupleList = [{x1, 1}, {x2, 0}, {x3, 1}] and say that the root is X1, then the method will get the value of x1 and travel to the right side of the tree etc. until we get to a leaf (a logical value).

getValue/2

Description

getValue (*Param, TupleList*) gets the parameter value from the parameter list [{Param1,Value},{Param2,Value}..] using element(2, lists : keyfind(Param, 1, TupleList)).

Input definition

A Boolean function is given using the following format:

- Each operator will be described by one of the following tuples:
  - { 'not' , Arg}
  - { 'or' , {Arg1, Arg2}}
  - { 'and' , {Arg1, Arg2}}
- No more than two arguments will be evaluated by a single operator
- Example: the Boolean function

  $f(x_1, x_2, x_3) = x_1 \overline{x_2} + x_2 x_3 + x_3$

  Will be represented as

  { 'or' ,{ { 'or' ,{ { 'and' ,{ x1 , { 'not' , x2} }} , { 'and' ,{ x2 , x3 }} }} , x3 }}

A BDD tree will be construct as the output of exp_to_bdd :

{ {Parameter, Left, Right}, Hight, NumOfNodes, NumOfLeafs}

When *{Parameter, Left, Right}* is the required information for solving the BDD (and the input to solve_bdd).
e.g. Running exp_to_bdd on the example in Boolean Function form

{'or', {{'or',{{'and',{x1,{'and',{{'not',x2},x3}}}},{'not',{'and',{x4,x1}}}}}, {'not',{'and',{{'and',{x1,{'not',x3}}},{'or',{{'not',x4},x2}}}}}}}

yields a several BDDs, choosing one of them for e.g. :

{x4, 1,{x3, {x1, 1,{x2, 1,0}},1}}

Analysis

The analysis will be performed on the following Boolean function :

$$f_r(x_1, x_2, x_3, x_4) = x_1\,\overline{x_2}\,x_3 + \overline{x_1\overline{x_3}(\overline{x_4} + x_2)} + \overline{x_4 x_1}$$

And in Boolean Function form :

{'or', {{'or',{{'and',{x1,{'and',{{'not',x2},x3}}}},{'not',{'and',{x4,x1}}}}},
{'not',{'and',{{'and',{x1,{'not',x3}}},{'or',{{'not',x4},x2}}}}}}}

Ordering	Time took	Results
tree_height	1.831902 ms	{x4,1,{x3,{x1,1,{x2,1,0}},1}}
num_of_nodes	1.809137 ms	{x4,1,{x3,{x1,1,{x2,1,0}},1}}
num_of_leafs	1.914523 ms	{x4,1,{x3,{x1,1,{x2,1,0}},1}}

Table 3.1 – Analysis of different *Ordering*

```
26> exf_318550746:exp_to_bdd({'or',{{'or',{{'and',{x1,{'and',{{'not',x2},x3}}}},
{'not',{'and',{{'and',{x1,{'not',x3}}},{'or',{{'not',x4},x2}}}}}}},  {'not',{'an
d',{x4,x1}}}}},num_of_leafs).
Finished in 1809137 ns
{x4,1,{x3,{x1,1,{x2,1,0}},1}}
```

Figure 3.1 – evaluating on linux

The time this operation took is the 'same' due to the design choice to accumulate those catachrestic(BDD height, num of leafs, num of nodes) while building the BDD tree – so the only difference when changing the second argument of this function is which element in the main BDD Tuple to check for optimum (2nd, 3rd or 4th element) with quick access;

making the time to create the BDDs list and find the minimum *Ordering* **'equal'** (I am using nanosecond resolution for the time unit as described above, and each run I get a range of results depending on the resources & compute power – all around the amount of time mention in the table ~1.8ms).

I got that all the permutation produces BDD with the same Height, num of leaf's and num of nodes.

```
6> exf_318550746:getBDDS({'or',{{'or',{{'and',{x1,{'and',{{'not',x2},x3}}}},{'no
t',{'and',{{'and',{x1,{'not',x3}}},{'or',{{'not',x4},x2}}}}}}},  {'not',{'and',{
x4,x1}}}}},exf_318550746:getPermutations([x1,x2,x3,x4])).
[{{x4,1,{x3,{x2,1,{x1,1,0}},1}},3,4,1},
 {{x4,1,{x3,{x1,1,{x2,1,0}},1}},3,4,1},
 {{x4,1,{x2,1,{x3,{x1,1,0},1}}},3,4,1},
 {{x4,1,{x2,1,{x1,1,{x3,0,1}}}},3,4,1},
 {{x4,1,{x1,1,{x3,{x2,1,0},1}}},3,4,1},
 {{x4,1,{x1,1,{x2,1,{x3,0,1}}}},3,4,1},
 {{x3,{x4,1,{x2,1,{x1,1,0}}},1},3,4,1},
 {{x3,{x4,1,{x1,1,{x2,1,0}}},1},3,4,1},
 {{x3,{x2,1,{x4,1,{x1,1,0}}},1},3,4,1},
 {{x3,{x2,1,{x1,1,{x4,1,0}}},1},3,4,1},
 {{x3,{x1,1,{x4,1,{x2,1,0}}},1},3,4,1},
 {{x3,{x1,1,{x2,1,{x4,1,0}}},1},3,4,1},
 {{x2,1,{x4,1,{x3,{x1,1,0},1}}},3,4,1},
 {{x2,1,{x4,1,{x1,1,{x3,0,1}}}},3,4,1},
 {{x2,1,{x3,{x4,1,{x1,1,0}},1}},3,4,1},
 {{x2,1,{x3,{x1,1,{x4,1,0}},1}},3,4,1},
 {{x2,1,{x1,1,{x4,1,{x3,0,...}}}},3,4,1},
 {{x2,1,{x1,1,{x3,{x4,1,...},1}}},3,4,1},
 {{x1,1,{x4,1,{x3,{x2,...},1}}},3,4,1},
 {{x1,1,{x4,1,{x2,1,...}}},3,4,1},
 {{x1,1,{x3,{x4,1,...},1}},3,4,1},
 {{x1,1,{x3,{x2,...},1}},3,4,1},
 {{x1,1,{x2,1,...}},3,4,1},
 {{x1,1,{x2,...}},3,4,1}]
```

Figure 3.2 – results from getBDDS, '…' is due to the buffer, and not represent the actual trees

This is due to the fact that the Boolean table is always 1 except a single input that produce 0, forcing every BDD tree to have a node for each variable (and each node can be a root).

a	b	c	d	x
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	0
1	1	1	0	1
1	1	1	1	1

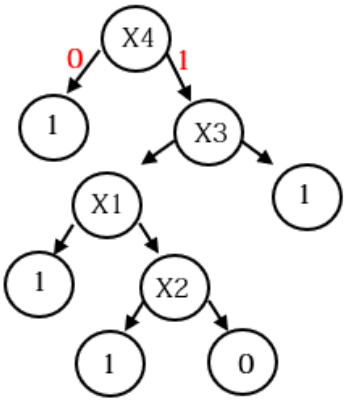


Table 3.2 – Logical table for the example Boolean function & tree from the results

In **getOptimumBDD** I didn't change the current optimum tree if the deciding value is the same, resulting the result of this specific function (in every *Ordering*) **to differ according to the permutation order** (which can be affected from the Identification process of the variable in the Boolean expression.

Solving the BDD in all different Parameters permutation should get us the correct logical table :

BDD {x4,1,{x3,{x1,1,{x2,1,0}},1}}			
Tuple List	x1 | x2 | x3 | x4 Logical values	Results	Time took [ns]
[{x1,0}, {x2,0}, {x3,0}, {x4,0}]	0 | 0 | 0 | 0	1	1153
[{x1,0}, {x2,0}, {x3,0}, {x4,1}]	0 | 0 | 0 | 1	1	1858
[{x1,0}, {x2,0}, {x3,1}, {x4,0}]	0 | 0 | 1 | 0	1	918
[{x1,0}, {x2,0}, {x3,1}, {x4,1}]	0 | 0 | 1 | 1	1	1550
[{x1,0}, {x2,1}, {x3,0}, {x4,0}]	0 | 1 | 0 | 0	1	882
[{x1,0}, {x2,1}, {x3,0}, {x4,1}]	0 | 1 | 0 | 1	1	1405
[{x1,0}, {x2,1}, {x3,1}, {x4,0}]	0 | 1 | 1 | 0	1	1040
[{x1,0}, {x2,1}, {x3,1}, {x4,1}]	0 | 1 | 1 | 1	1	1513
[{x1,1}, {x2,0}, {x3,0}, {x4,0}]	1 | 0 | 0 | 0	1	2980
[{x1,1}, {x2,0}, {x3,0}, {x4,1}]	1 | 0 | 0 | 1	1	1593
[{x1,1}, {x2,0}, {x3,1}, {x4,0}]	1 | 0 | 1 | 0	1	1158
[{x1,1}, {x2,0}, {x3,1}, {x4,1}]	1 | 0 | 1 | 1	1	1530

[{x1,1}, {x2,1}, {x3,0}, {x4,0}]	1	1	0	0	1	1938
[{x1,1}, {x2,1}, {x3,0}, {x4,1}]	1	1	0	1	0	1619
[{x1,1}, {x2,1}, {x3,1}, {x4,0}]	1	1	1	0	1	1042
[{x1,1}, {x2,1}, {x3,1}, {x4,1}]	1	1	1	1	1	1291

Table 3.3 – Results of solve_bdd

Due to the simplicity in traveling the tree & taking into account the power of my local machine its takes in average around 1.7ms to complete.

If we look at the tree structure as described above, we can see that when x4=0 the results are faster (instant leaf of 1) in comparison to x4=1 (continue traveling the tree). A similar behavior is observed regarding other branching (instant leaf/continue to travel), unless we get some random bumps in running time due to current available resources & compute power.

```
6> exf_318550746:solve_bdd({x4,1,{x3,{x1,1,{x2,1,0}},1}},[{x1,0},{x2,0},{x3,1},{x4,1}]).
Finished in 1550 ns
1
7> exf_318550746:solve_bdd({x4,1,{x3,{x1,1,{x2,1,0}},1}},[{x1,0},{x2,1},{x3,0},{x4,0}]).
Finished in 882 ns
1
8> exf_318550746:solve_bdd({x4,1,{x3,{x1,1,{x2,1,0}},1}},[{x1,0},{x2,1},{x3,0},{x4,1}]).
Finished in 1405 ns
1
9> exf_318550746:solve_bdd({x4,1,{x3,{x1,1,{x2,1,0}},1}},[{x1,0},{x2,1},{x3,1},{x4,0}]).
Finished in 1040 ns
1
10> exf_318550746:solve_bdd({x4,1,{x3,{x1,1,{x2,1,0}},1}},[{x1,0},{x2,1},{x3,1},{x4,1}]).
Finished in 1513 ns
1
11> exf_318550746:solve_bdd({x4,1,{x3,{x1,1,{x2,1,0}},1}},[{x1,1},{x2,0},{x3,0},{x4,0}]).
Finished in 2980 ns
1
12> exf_318550746:solve_bdd({x4,1,{x3,{x1,1,{x2,1,0}},1}},[{x1,1},{x2,0},{x3,0},{x4,1}]).
Finished in 1593 ns
1
13> exf_318550746:solve_bdd({x4,1,{x3,{x1,1,{x2,1,0}},1}},[{x1,1},{x2,0},{x3,1},{x4,0}]).
Finished in 1158 ns
1
14> exf_318550746:solve_bdd({x4,1,{x3,{x1,1,{x2,1,0}},1}},[{x1,1},{x2,0},{x3,1},{x4,1}]).
Finished in 1530 ns
1
15> exf_318550746:solve_bdd({x4,1,{x3,{x1,1,{x2,1,0}},1}},[{x1,1},{x2,1},{x3,0},{x4,0}]).
Finished in 1938 ns
1
16> exf_318550746:solve_bdd({x4,1,{x3,{x1,1,{x2,1,0}},1}},[{x1,1},{x2,1},{x3,0},{x4,1}]).
Finished in 1619 ns
0
```

Figure 3.3 – solve_bdd partial results

Conclusions & Further work

In this project I got a broader introducing to Erlang functionality on a subject from computer science computing.

Although we can see the improvements on minimizing a BDD, this approach can become very difficult to preserve (good performance) on scaling data – to implement Shannon expansion theorem we need to compute N! permutation of BDDs tree (creating, reducing, and minimizing) with N parameters.

We can't foreshadow which premutation will yields the best performance in terms of height, num of leaf's / num of nodes – so computing all of them is necessary in this simple approach to Shannon expansion theorem.

My algorithm approach to accumulating the results is computing each BDD after the other. Not in parallel computing – although each computation is not dependents on the other permutation. Hence, we can develop and implement an algorithm which handle this task in parallel (distribute the task of computing individual BDDs based on permutation of Parameter order [Shannon expansion theorem]) and then collect the results using messaging queues. Such implementation over distribute computing system will greatly improve the running time of exp_to_bdd.

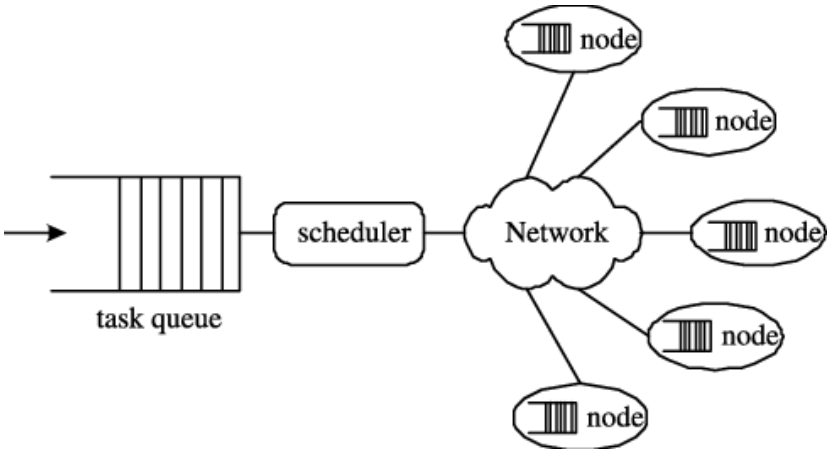


Figure 3.3 – an example of distributed system. Parallel computing in multi-core machine can be another/additional approach (Threads/Processes).

Thank you.