Ben-Gurion University of the Negev

Binary Decision Diagram

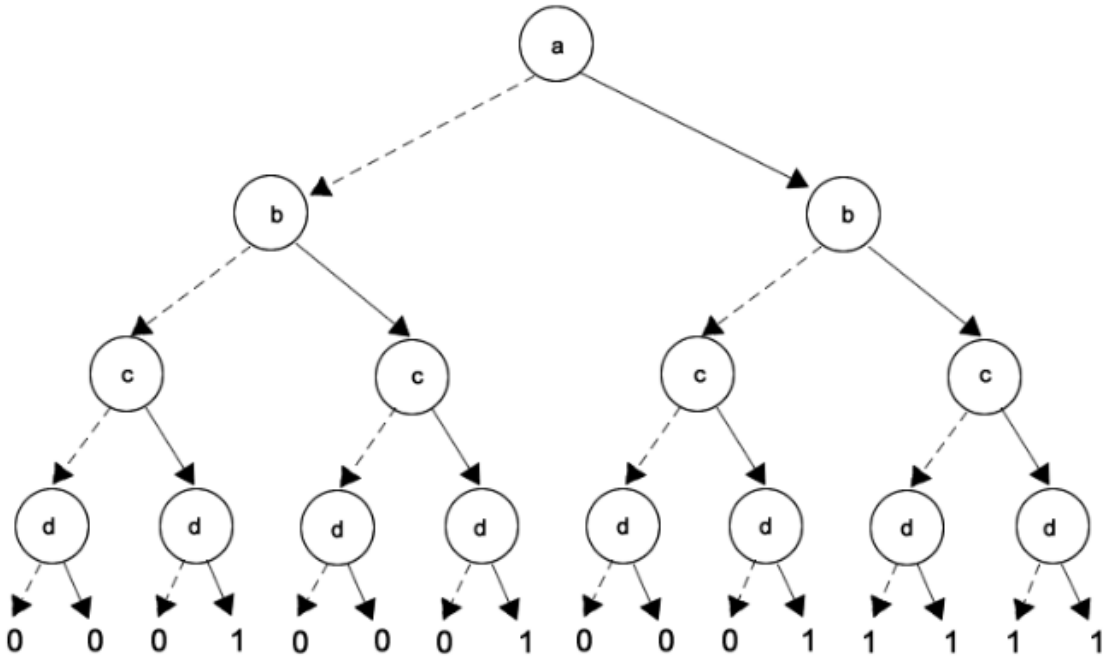
Mid project report

Ben Gurion University of the Negev

Faculty of Engineering Science

Functional Programming in Concurrent and Distributed System

April 2023



written by Shahaf Zohar

205978000

Ben-Gurion University of the Negev

Contents

Ben-Gurion University
of the Negev

Abstract

In this assignment, you are asked to implement an automatic construction machine of a Binary Decision Diagram (BDD) to represent a Boolean function. The API should allow the user getting a BDD representation of a function within a single call to your machine and select the data structure it uses. 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 that's 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 (bonus).
2. Eliminate any node whose two children are isomorphic.

The construction of BDD is based on Shannon expansion theory

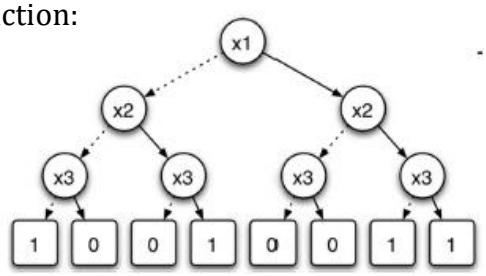
$$f(x_1, x_2, \dots, x_n) = x_1 \cdot f(1, x_2, x_3, \dots, x_n) + \bar{x}_1 f(0, x_2, x_3 \dots, x_n)$$

Example

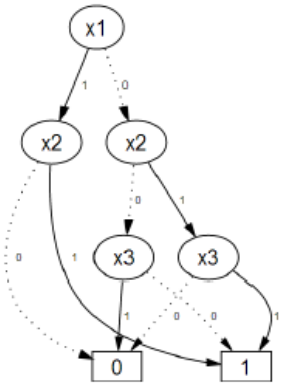
The Boolean function $f(x_1, x_2, x_3) = \bar{x}_1 \cdot \bar{x}_2 \cdot \bar{x}_3 + x_1 \cdot x_2 + x_2 \cdot x_3$ with the following truth table:

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:



Note that the results are the same for both BDD trees, for every assignment of the Boolean function.

Ben-Gurion University
of the Negev

Describe basic functions

Function :exp_to_bdd/3

spec exp_to_bdd(BoolFunc, Ordering, DataStructureType) → BddTree.

- The function receives a Boolean function and returns the corresponding BDD tree representation for that Boolean function.
- The returned tree must be the one that is the most efficient in the manner specified in variable **Ordering**.

I. Variable **Ordering** can be one of the following atoms: **tree_height**, **num_of_nodes** or **num_of_leafs** which is the optimal tree (minimum height, minimum number of nodes and minimum number of leaves).

II. To extract the optimal tree, you should **Compare all the possible permutations** of BDD trees.

III. *Permutations:* let's assume that you are asked to convert a Boolean function that has 3 Boolean arguments: $f_s(x_1, x_2, x_3) \cdot f_s$ can 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:

$$\{\{x_1, x_2, x_3\}, \{x_1, x_3, x_2\}, \{x_2, x_1, x_3\}, \{x_2, x_3, x_1\}, \{x_3, x_2, x_1\}, \{x_3, x_1, x_2\}\}$$

- The returned tree must be reduced by Rule 1. Read more about in BDD Introduction.
- Data structure type can be **map** (atom) or **record** (atom) (both methods should be implemented).

My function:

```
%-----------------------------------------------------------------------------------------------------------------------
exp_to_bdd(BoolFunc, Ordering, DataStructureType)->
  %• This function first arranges the information in a
  %• list and then according to the selection prints the tree.
  Tstart = erlang:now(),                          % Start the timer
  Lists = permotations(listmaker(BoolFunc)),      % Arreng the to list
  ListT = case DataStructureType of
                record -> [reducingtRec(recStruct(L,[],BoolFunc)) || L <-Lists ];  %case by coohes record
                map -> [reducingMap(mapStruct(L,[],BoolFunc)) || L <-Lists ]      %case by coohes map
              end,

  %Creating a tree according to the user's request
  Result = case Ordering of
               tree_height ->  [tree_high(Tree) || Tree <- ListT];
               num_of_nodes -> [number_of_nodes(Tree)|| Tree <- ListT];
               num_of_leafs -> [number_of_leafs(Tree)|| Tree <- ListT]
             end,
    Tend=erlang:now(),      % Stop the timer
  io:format("------------------------------------------------------------ ~n"),
  case Ordering of
               tree_height ->  io:format("the height of the tree is ~p.~n", [lists:nth(1,Result)]);
               num_of_nodes -> io:format("The number of nodes is ~p.~n", [lists:nth(1,Result)]);
               num_of_leafs -> io:format("The number of leaves in the tree is ~p.~n", [lists:nth(1,Result)])
             end,
  io:format("runtime = ~p  microseconds~n", [timer:now_diff(Tend,Tstart)]),      %Print and calculate the time
  lists:nth(find_index(lists:min(ResSult),Result),ListT).                        %The function takes two arguments: the index of the element

%-----------------------------------------------------------------------------------------------------------------------
```

At the beginning of the program, I take the template and arrange it in a list so that there are no duplicates of all the variables. then I call the recStruct/mapStruct functions who actually arrange the tree for me in a pattern. For record the tree is built in such a way that Structs are used -

$$recstruct: \{val, sublist\} \backslash node: \{elem, left, right\}.$$

For map we will use the Erlang map template also recursively until it is filled.
Depending on the ordering we will return the height of the tree or the number of vertices or the number of leaves in the tree. And also counts the time of the program.

Function: solve_bdd/2

spec solve_bdd(BddTree, $[\{x_1, Val1\}, \{x_2, Val2\}, \{x_3, Val3\}, \{x_4, Val4\}] \rightarrow Res$.
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**.
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.
%• A function that, depending on the type of tree, returns the result
solve_bdd(T, Arg) ->
  Tstart=erlang:now(),
  Res = if
          is_map(T) -> solve_Map(T, Arg);
          is_record(T,node) -> solve_bddRec(T,Arg);
          true -> T
        end,
  Tend=erlang:now(),
  io:format("------------------------------------------------------------ ~n"),
  io:format("Run time: ~p  microseconds.~n", [timer:now_diff(Tend,Tstart)]),
  if
      Res =:= true ->
                  io:format("The Result: true.~n");
      Res =:= false ->
          io:format("The Result: false.~n")
    end.
%-----------------------------------------------------------------------------------------------------
```

Also, the function here receives the tree and values for each variable, assembles the function from the tree and outputs a value according to the variables.
Here will know according to the type of the tree pattern the analysis of getting a true or false answer.

Function: listOfLeaves/1

spec listOfLeaves(BddTree) -> Res.

- Returns list of pointers to leaves.

```
%-----------------------------------------------------------------------------------------------------
%• Returns list of pointers to leaves.
listOfLeaves(BddTree) ->  %A function that, depending on the type of tree, returns the list of leaves
  if
      is_record(BddTree,node)-> leavesRecord([],BddTree) ;
      true -> leavesMap([],BddTree)
  end.
%-----------------------------------------------------------------------------------------------------
```

Function: reverseIteration/1

spec reverseIteration(LeafPtr) -> Res.

- Input is pointer to leaf (one of leaves given in result list of step 3).
- Returns list of nodes on shortest path to the root.

```
%-----------------------------------------------------------------------------------------------------------------
%• Input is pointer to leaf (one of leaves given in result list of step 3).
%• Returns list of nodes on shortest path to the root.
%• In my word:"The function get pointer to leaf and return list of variable that represnt the rout to the route."
reverseIteration(LeafPtr) ->
  if               % Checking the the type
    is_map(LeafPtr) ->
              #{list := List} = LeafPtr,            %Creating map to do mach between the input and return the list node like
              List;
    is_record(LeafPtr,recstruct) ->               % Checking if the stract is "recpointer"
              #recstruct{val = _Val,sublist = List} = LeafPtr,   %The same thing like map case, but now return from record "recstruct"
              List;
    true -> true
  end.

%-----------------------------------------------------------------------------------------------------------------
```

structure

I used the structure like this in my code for building the tree

```
-record(node,{elem,left,right}).
-record(recstruct,{val,sublist}).
```

Significant functions

Descriptions

Now I will explain about all the functions of my code in the order in which I will use them, of course there are functions that are supported by auxiliary functions that I created which I also explain.

slove_bdd/2

slove_bdd(T, List) function. The inputs are passed to the aid function **solve_bdd**/2, which travel on BDD tree according to the Template presented on record Tuple lists of **#node,{elem,left,right}** , and for map we used the build in function in Erlang.
e.g. List = $[\{a_1, 0\}, \{a_2, 1\}, \{a_3, 1\}, \{a_4, 0\}]$ and say that the root is a_1, then the method will get the value of a_1 and travel to the right side of the tree etc. until we get to a leaf (a logical value).

Permotations/1

The function permotations takes a list L as input.

- If L is an empty list, the function returns a list containing a single empty list. This represents the base case of the recursive since there is only one permutation of an empty list.
- If L is not empty, the function generates all possible permutations of L by iterating over each element H of L and each permutation T of the remaining elements in L (i.e., L with H removed).
- For each combination of H and T, the function creates a new list containing H followed by T and adds this list to the list of permutations. This is done using a list comprehension that iterates over H and T and constructs the new list [[H|T] || H<-L,T<-permotations(L--[H])].
- Once the function has iterated over all possible combinations of H and T, it returns the list of permutations.

In summary, this function recursively generates all possible permutations of a given list by iteratively selecting each element of the list and recursively generating permutations of the remaining elements. The result is a list of lists, where each inner list represents a unique permutation of the input list.

element_Val/2

This function element_Val is a simple implementation of a lookup function that searches a list of key-value pairs and returns the value associated with a given key.

- The function element_Val takes two arguments: a list of key-value pairs [ ] and a key Element to search for in the list.
- If the list is empty (i.e., [ ]), the function outputs an error message indicating that the key was not found in the list.
- If the first key-value pair in the list matches the key Element, the function returns the corresponding value.
- If the first key-value pair does not match the key Element, the function recursively calls itself with the tail of the list (T) to search for the key in the remaining pairs.

The function assumes that each key in the list is unique and that the key-value pairs are ordered in a specific way. Specifically, it assumes that the first key-value pair in the list that matches the search key is the correct one, and that all subsequent pairs can be ignored.

rearrang/2

This is a function that takes two arguments - an operation and a number - as well as a list. It uses pattern matching and conditionals to evaluate the operation and number based on the contents of the list.
If the operation is 'not', the function first checks if the number is a tuple. If it is, the function recursively calls itself with the number and the list as arguments. If the number is not a tuple, the function calls another helper function, element_Val, to check if the number exists in the list. The result of either of these checks is then negated using the not function.

If the operation is 'or' or 'and', the function checks if the number is a tuple. If both arguments are not tuples, the function calls element_Val twice - once for each argument - and evaluates the result using the corresponding logical operator. If one argument is a tuple and the other is not, the function either calls itself recursively with the tuple argument or calls element_Val with the non-tuple argument, depending on which argument is the tuple. If both arguments are tuples, the function recursively calls itself with each tuple argument.

Overall, this function appears to be part of a larger program that is using a recursive approach to evaluate logical expressions with variables.

recStruct/3

This is function that creates a record type tree. The input parameters are a list of elements [H|T], a list List, and a Boolean function BoolFunc.
The function checks if the tail of the list T is empty. If it is, it creates a node in the tree with H as the element, and the left and right branches of the node are determined by calling the rearrang() function with BoolFunc and List appended with {H, false} and {H, true} respectively.

If the tail of the list T is not empty, the function recursively calls itself twice: once with the tail of the list and List appended with {H, false}, and once with the tail of the list and List appended with {H,true}. These two recursive calls determine the left and right branches of the node, and the node is returned with H as the element and the left and right branches determined by the recursive calls.

mapStruct/3

The mapStruct/3 function takes three arguments: a list of elements, a list of key-value pairs, and a Boolean function. It creates a record type tree where each node contains an element from the input list and its left and right branches.
The function starts by checking if the input list has only one element. If so, it creates a leaf node by setting the 'elem', left, and right fields of the record using the rearrang/2 function applied to the Boolean function and the current element with both true and false values.

If the input list has more than one element, it creates a new node with the current element as the elem field and calls itself recursively with the remaining elements and the current element added to the List with true and false values. The function returns a record that represents the root of the record type tree.

reducingRec/1

Here I used to reduce a binary tree. The input to the function is a binary tree with nodes containing a value and left and right branches. The function recursively traverses the tree and checks if the left and right branches are also nodes. If they are not nodes, then it checks if they have the same value. If they do, then it returns one of them. If they don't have the same value, then it returns the original tree. If the left and right branches are nodes, then it recursively reduces them using the same logic and creates a new node with the same value and reduced branches. Finally, it returns the reduced tree.

reducingMap/1

This is a function reduces a binary tree of maps recursively until all the maps in the tree have a unique value. It takes a binary tree of maps as input and returns a binary tree of maps.

The function first matches the input binary tree against a map pattern with three keys: elem, left, and right. Then, it checks if the left and right nodes are maps or not. If they are not maps, it checks if they are equal. If they are equal, it returns the right node. Otherwise, it returns the input binary tree to stop the recursion.

If the left and right nodes are maps, the function recursively calls itself with each of them, and then checks if they are equal or not. If they are equal, it returns the right node. Otherwise, it returns a new map with the same elem key as the input map, and with the left and right keys being the results of calling the function recursively with the left and right nodes, respectively.

Travel functions

number_of_nodes/1

The function calculates the number of nodes in a binary tree or map. It first checks if the input is a record of type "node" or a map. If it's a node, it recursively calls the function on the left and right subtrees of the node, adds 1 to account for the node itself, and returns the sum. If it's a map, it recursively calls the function on the left and right branches of the map, adds 1 to account for the map itself, and returns the sum. If the input is not a node or a map, the function simply returns 1 to account for the single node.

number_of_leafs/1

This function counts the number of leaf nodes in a tree-like data structure. It takes a single argument Tree, which can be either a record or a map.
First, the function checks if the Tree argument is a record or a map using the is_record/2 and is_map/1 functions. If it is a record, it extracts the left and right fields of the node record using pattern matching, and recursively calls number_of_leafs/1 on each of those sub-trees, summing their results. If it is a map, it extracts the left and right fields of the map using pattern matching, and again recursively calls number_of_leafs/1 on each sub-tree and sums their results.

Once it reaches a leaf node (i.e., a node with no left or right sub-tree), it returns 1. This base case ensures that the recursion eventually terminates and returns a final result.

In summary, number_of_leafs/1 is a function that recursively counts the number of leaf nodes in a tree-like data structure represented as a record or a map.

tree_ height /1

The function takes a tree as input and calculates the height of the tree. The height of a tree is defined as the length of the longest path from the root node to any leaf node in the tree.
The function uses pattern matching with guards to check if the input is a record or a map. If it is a record, the function extracts the left and right nodes of the tree and recursively calls itself on each node to calculate the height of the left and right subtrees. The function then returns the maximum height of the left and right subtrees, plus 1 to account for the root node.
If the input is a map, the function extracts the left and right nodes of the tree and recursively calls itself on each node to calculate the height of the left and right subtrees. The function then returns the maximum height of the left and right subtrees, plus 1 to account for the root node.
If the input is neither a record nor a map, the function returns 1, representing the height of a tree with a single node.

In summary, tree_high/1 recursively calculates the height of a tree by finding the maximum height of its left and right subtrees and adding 1 for the root node.

solve_Map/2

This is a function that traverses a binary tree data structure represented as a map, searching for a specific element based on a binary sequence given as an argument. It converts the binary sequence to a list of Booleans, and then uses the 'find' function to search for the corresponding element. Depending on the value of the element, it continues traversing the tree either to the left or to the right until it finds the element and returns the corresponding subtree.

solve_bddRec/2

The solve_bddRec function takes a binary decision diagram (BDD) Tree and a list of Boolean values Arg as input. It then converts Arg to a list of Booleans ListBool, searches for an element in Tree using the find function, removes that element from ListBool, and depending on the value of that element in Arg, either recursively calls solve_bddRec on the right subtree of Tree or returns the right subtree, or recursively calls solve_bddRec on the left subtree of Tree or returns the left subtree. This function is used to solve Boolean functions represented by BDDs.

remove_duplicates/1

This is a function to remove duplicates from a list. The function takes a list as input and calls the function remove_duplicates/2 with the list and an empty accumulator as arguments.

The remove_duplicates/2 function takes two arguments: the first is the list, and the second is an accumulator to store the unique elements. The function checks if the list is empty, in which case it returns the reversed accumulator. If the list is not empty, it checks if the head of the list is already a member of the accumulator. If it is, the function continues with the tail of the list and the accumulator unchanged. If it is not, the function continues with the tail of the list and the head added to the accumulator. The function keeps recursively processing the list until it is empty, and then it returns the accumulator with unique elements.

listmaker/1

This is a function in Erlang that takes a Boolean function as an argument and returns a list of all the variables in the function. The listmaker function first calls the converted function on the Boolean function argument to get a list of variables in the function. The converted function recursively traverses the Boolean function expression tree and extracts all variables that occur in the tree. The resulting list may contain duplicates, so the remove_duplicates function is called on the list to remove duplicates. Finally, the list of variables is returned.

Overall, this function is useful for extracting variables from a Boolean function and can be used for further processing such as truth table generation or logical simplification.

leavesMap/2 and leavesRec/2

The leavesRec/2 and leavesMap/2 functions are used to traverse a binary decision diagram (BDD) data structure and collect information about its leaf nodes.

leavesRec/2 works and takes two arguments: Acc, which is an accumulator used to build up a list of leaf nodes, and BddTree, which is the BDD being traversed. The function uses pattern matching to check whether the BDD node is a record and, if so, extracts the elem, left, and right fields. It then recursively calls itself on the left and right subtrees, passing along a list containing the elem value of the current node. When a leaf node is reached (i.e., a node that is not a record), the function returns a map containing the leaf value (BddTree) and the list of elem values that led to that leaf (Acc).

leavesMap/2 works similarly to leavesRec/2 but uses maps instead of records for the BDD nodes. It checks if the BddTree is a map and if so, extracts the elem, left, and right fields. It then recursively calls itself on the left and right subtrees, passing along a list containing the elem value of the current node. When a leaf node is reached, the function returns a map containing the leaf value (BddTree) and the list of elem values that led to that leaf.

listOfLeaves

The listOfLeaves function takes a binary decision diagram (BDD) represented by either a record or a map and returns a list of its leaves. It uses a conditional statement (if/else) to determine the type of tree it is given and then calls the appropriate function (leavesRec or leavesMap) to generate the list of leaves.

If the BDD is represented by a record, the leavesRec function is called with an empty list ([]) as the initial accumulator and the BDD as input. The leavesRec function recursively traverses the BDD and accumulates the leaf nodes in the accumulator. It returns a list of records with two fields: val and sublist. The val field contains the value of the leaf node, and the sublist field contains a list of the elements on the path from the root of the BDD to the leaf node.

If the BDD is represented by a map, the leavesMap function is called with an empty list ([]) as the initial accumulator and the BDD as input. The leavesMap function recursively traverses the BDD and accumulates the leaf nodes in the accumulator. It returns a list of maps with two keys: val and list. The val key contains the value of the leaf node, and the list key contains a list of the elements on the path from the root of the BDD to the leaf node.

Finally, the listOfLeaves function uses a conditional statement to determine which function to call based on the type of BDD it is given.

Algorithm Flow

Note: the algorithm example will be preformed om the following Boolean:

$$f_r(x_1, x_2, x_3, x_4) = x_1\bar{x}_2x_3 + \overline{x_1\overline{x_3}(\bar{x}_4 + x_2)} + \overline{x_4x_1}$$

The format of the Boolean function:

$$\{'or', \{\{'or', \{\{'and', \{\{'and', \{x_1, \{'not', x_2\}\}\}, x3, \{'not', \{'and', \{\{'and', \{x_1, \{'not', x_3\}\},$$
$$\{'or', \{\{'not', x_4\}, x_2\}\}\}\}\}\}, \{'not', \{'and', \{x_4, x_1\}\}\}\}$$

1. The Boolean input expression BoolFunc is been passed to the function **listmaker/1**
   The result will be a list of all the parameter without duplication e.g. $f_r(x_1, x_2, x_3)$ result: $[x_1, x_2, x_3]$.
   And then we pass this list to function **permotations/1** the result will be all the permutations for example : For example, 4 element we will get 24 permutation and generate a tree and take the optimal one by reduction. All possible permutations for the report without duplicates:

```
All the premutation:
[[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]]
```

```
All the premutation:
[{node,x1,true,{node,x2,true,{node,x3,{node,x4,true,false},true}}},
 {node,x1,true,{node,x2,true,{node,x4,true,{node,x3,false,true}}}},
 {node,x1,true,{node,x3,{node,x2,true,{node,x4,true,false}},true}},
 {node,x1,true,{node,x3,{node,x4,true,{node,x2,true,false}},true}},
 {node,x1,true,{node,x4,true,{node,x2,true,{node,x3,false,true}}}},
 {node,x1,true,{node,x4,true,{node,x3,{node,x2,true,false},true}}},
 {node,x2,true,{node,x1,true,{node,x3,{node,x4,true,false},true}}},
 {node,x2,true,{node,x1,true,{node,x4,true,{node,x3,false,true}}}},
 {node,x2,true,{node,x3,{node,x1,true,{node,x4,true,false}},true}},
 {node,x2,true,{node,x3,{node,x4,true,{node,x1,true,false}},true}},
 {node,x2,true,{node,x4,true,{node,x1,true,{node,x3,false,true}}}},
 {node,x2,true,{node,x4,true,{node,x3,{node,x1,true,false},true}}},
 {node,x3,{node,x1,true,{node,x2,true,{node,x4,true,false}}},true},
 {node,x3,{node,x1,true,{node,x4,true,{node,x2,true,false}}},true},
 {node,x3,{node,x2,true,{node,x1,true,{node,x4,true,false}}},true},
 {node,x3,{node,x2,true,{node,x4,true,{node,x1,true,false}}},true},
 {node,x3,{node,x4,true,{node,x1,true,{node,x2,true,false}}},true},
 {node,x3,{node,x4,true,{node,x2,true,{node,x1,true,false}}},true},
 {node,x4,true,{node,x1,true,{node,x2,true,{node,x3,false,true}}}},
 {node,x4,true,{node,x1,true,{node,x3,{node,x2,true,false},true}}},
 {node,x4,true,{node,x2,true,{node,x1,true,{node,x3,false,true}}}},
 {node,x4,true,{node,x2,true,{node,x3,{node,x1,true,false},true}}},
 {node,x4,true,{node,x3,{node,x1,true,{node,x2,true,false}},true}},
 {node,x4,true,{node,x3,{node,x2,true,{node,x1,true,false}},true}}]
```

2. Next, we are subject to selection according to the module we were asked for from the user (**DataStructureType**). Accordingly, we will run on all promotions.

3. **recStruct/3** return the BDD List, each BDD corresponds to each Parameters as. Then the same list is put into the **reducingRec/1** or **reducingMap/1** function that reduces our BDD tree, this is how we do for each permutation.

For running the code we will get:

```
{ok,exf_205978000}
99> exf_205978000:exp_to_bdd({'or',{{'or',{{'and',{{'and',{x1,{'not',x2}}},x3}},{'not',{'and',{{'and',{x1,{'not',x3}}},{'or',{{'not',x4},x2}
}}}}}},{'not',{'and',{x4,x1}}}}},num_of_leafs,map).
-----------------------------------------------------------------
The number of leaves in the tree is 5.
runtime = 209  microseconds
#{elem => x1,left => true,
  right =>
      #{elem => x2,left => true,
        right =>
            #{elem => x3,
              left => #{elem => x4,left => true,right => false},
              right => true}}}
```

Input definition

A Boolean function is given using the following format:

- Each operator will be described by one of the following tuples:
  - $\{'not' \ , \ Arg\}$
  - $\{'or' \ ,\{Arg_1, Arg_2\}\}$
  - $\{'and' \ ,\{Arg_1, Arg_2\}\}$
- No more than two arguments will be evaluated by a single operator
- Example: The Boolean function

  $$f_r(x_1, x_2, x_3) = x_1\bar{x}_2 + x_2x_3 + x_3$$

  Will be represented as

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

A BDD tree will be construct as 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

10 time measurements in microseconds

record		
tree_height	num_of_leafs	num_of_nodes
353	408	390
378	173	206
338	306	232
335	213	337
190	336	186
384	331	152
147	323	142
378	333	305
276	161	336
439	172	170

map		
tree_height	num_of_leafs	num_of_nodes
402	418	402
417	190	432
226	362	161
431	427	449
441	400	391
401	217	433
323	358	380
423	479	385
225	426	502
459	400	419

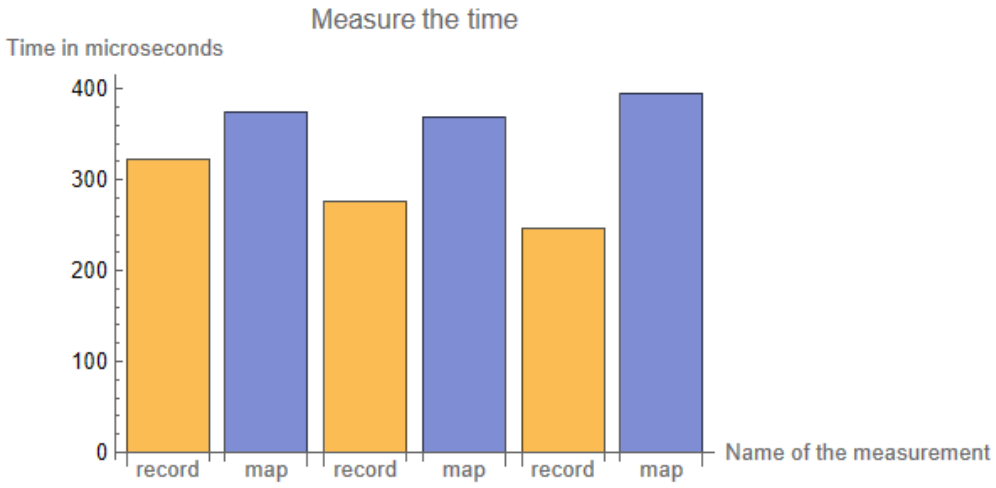
Avg record:

- tree_height = 321.8 μ_s
- num_of_leafs = 275.6 μ_s
- num_of_nodes = 245.6 μ_s

Avg map:

- tree_height = 374.8 μ_s
- num_of_leafs = 368.9 μ_s
- num_of_nodes = 395.4 μ_s

We will see the measurements in the graph:



Conclusions

how is better

The performance of the MAP function versus using records would depend on the specific use case and the size of the data.

The MAP function is a built-in higher-order function in Erlang that applies a given function to each element in a list or other data structure and returns a new list or structure containing the results. MAP can be a very efficient way to process large amounts of data in parallel, as it allows multiple processes to work on different elements of the list simultaneously.

Using records, on the other hand, involves defining a custom data structure with named fields that can hold values of different types.
Records can be convenient for organizing and accessing data, but they may be less efficient than using built-in functions like MAP for processing large datasets, as they typically require more memory and may involve more overhead.

Ultimately, the best approach will depend on the specific requirements of the problem at hand. If the data can be easily represented as a list or other structure and the processing can be parallelized,

MAP may is better option. If because data is more complex and requires a custom data structure,

records is more appropriate. It's important to consider factors like data size, processing time, and memory usage when choosing between these options.

Description

The time this operation took is the 'same' due to the design choice to accumulate those catechistic (tree_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.
Making the time to create the BDDs list and find the minimum Ordering 'equal' .

Conclusions & Further work

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

A Binary Decision Diagram (BDD) is a data structure used in computer science to represent Boolean functions. BDDs have been found to be useful in many applications, including computer-aided design (CAD), hardware verification, and artificial intelligence (AI).
One of the main advantages of using BDDs is that they allow us to efficiently minimize Boolean functions, which can be very helpful when dealing with large amounts of data. However, as the amount of data grows, implementing the Shannon expansion theorem to minimize BDDs becomes increasingly difficult.
The Shannon expansion theorem is a method used to reduce the size of a BDD by recursively splitting it into two smaller BDDs, one for each possible value of a given variable. To do this, we need to compute N! permutations of the BDD's tree, where N is the number of variables in the Boolean function.
This means that as the number of variables increases, the number of permutations that we need to compute grows exponentially. This can quickly become a bottleneck for performance, as it requires a large amount of computational resources and can take a long time to complete.
In addition, creating, reducing, and minimizing BDDs with a large number of variables can also be challenging due to memory limitations. The size of the BDD can grow exponentially with the number of variables, which means that we may run out of memory if we try to store and manipulate it directly.
In summary, while BDDs can be useful for minimizing Boolean functions, the process becomes increasingly difficult and resource-intensive as the amount of data and the number of variables grows. It's important to carefully consider the trade-offs between accuracy and computational resources when using BDDs for large-scale problems.

When using the Shannon expansion theorem to minimize a Binary Decision Diagram (BDD), we need to consider all possible permutations of the BDD's tree structure. This is because we can't predict ahead of time which permutation will result in the most efficient BDD in terms of tree height, number of leaf nodes, and num of nodes.

Because we can't predict the best permutation, we have to compute all of them, which can become very computationally expensive for large BDDs What we explained before.

My algorithm for minimizing a Binary Decision Diagram (BDD) computed each BDD in sequence, one after the other. Even though each BDD computation did not depend on the results of the others, this sequential approach limited the performance gains I could achieve.

Suggestion for improvement A new algorithm that distributes the task of computing individual BDDs across multiple processors or cores, allowing them to run concurrently. An algorithm that utilizes Shannon's expansion theorem to determine the optimal parameter order for each permutation of the BDD variables.

In other words, after creating the permutations, you can utilize and open processes for each permutation that will calculate a tree and finally try to send messages between the processes in an intelligent way to determine which is the optimal tree.