**Report for BDD Assignemnt – Erlang**

1. **Intro:**

In this report, I will describe shortly how I implemented the required functions exp\_to\_bdd and solve\_bdd. Afterwards I will include the required tasks of comparing elapsed time between permutations. The function description will be short and will include parts from the code comments, because it's pretty much self-explanatory.

1. **Design Report:**
   1. **exp\_to\_bdd:**
      1. Main Approach: The main approach that was taken here (done in function **ite/5**) is to start from the first variable (for the example let's take the order [x1, x2..xn]) , assigning this variable in the Boolean function as 1 (will be called High), and 0 (aka Low), and recursively returning {Low, High }. Meaning that every recursive call will assign a different variable until all of them are assigned and no more atoms except 'or', 'and' and 'not' are present in the function. When we reach that stage (no more variable atoms), the function result is evaluated and a leaf created with the value 0 or 1. The recursive calls go back until all the nodes are construct.
      2. Tree Representation: {varAtom, {Low, High}}. For example, the tree in Fig1 will be represented as **{x1, {{x2, {1,0}},1}}**

Fig1 – Example Tree

* + 1. Ordering Implementation: For the **ordering** requirement to work I created a function called **rotateIte/5** which receives as input the Bool function, a measurement function and a list of all the permutations of the sequence 1 to n (for x1,x2..xn) which symbolize the order that the function ite/5 should go over the variables. The function will invoke the item/5 function for each permutation using list-comprehension and then will measure the quality of each tree according to the parameter **treeHeight, NumOfNodes, numOfLeafs**. Finally the best matching tree is chosen and returned.
    2. Important Helping functions:
       1. assignNext/3 - Assigning a value inside the Boolean Function. **input** - Boolean function {operaion,{A,B}} , Value (1\0) , atom (x1,x2...) which represent a variable in the function. **output** - The initial Boolean function with the given 'Value' instead of every instance of the given atom. **example** - assignNext({'or',{x1,x2}},1,x1) -> {'or',{1,x2}}.
       2. eval/1 - returns the final answer of a given an assigned Boolean function (with x1..xn is replaces with 0,1). It's done recursively while each recursive call deals with the evaluation of one operation (not, or, and).
       3. flatten/1 - Takes a Boolean function and returns a flat list with all the atom variables in it, excluding the 'or', 'and', 'not' operators. the list is sorted in the order of appearance. may include duplicated variables which will be deleted later. **Example -**  flatten({'or',{x1,x2}}) -> [x1,x2].

1. **Time Measurements:**
   1. **General:** From my measurements on my personal computer (running Ubuntu on VMware virtual machine), the elapsed time between each similar run changes quite a lot, meaning that this subject in the assignment doesn’t tell us much. Numbers detailed below.
   2. **Measuring exp\_to\_bdd:** The elapsed time that it takes to create trees and choose the most efficient one ranges from 20,000 micro secs to 8,000 micro secs. The first runs usually are the longer ones.

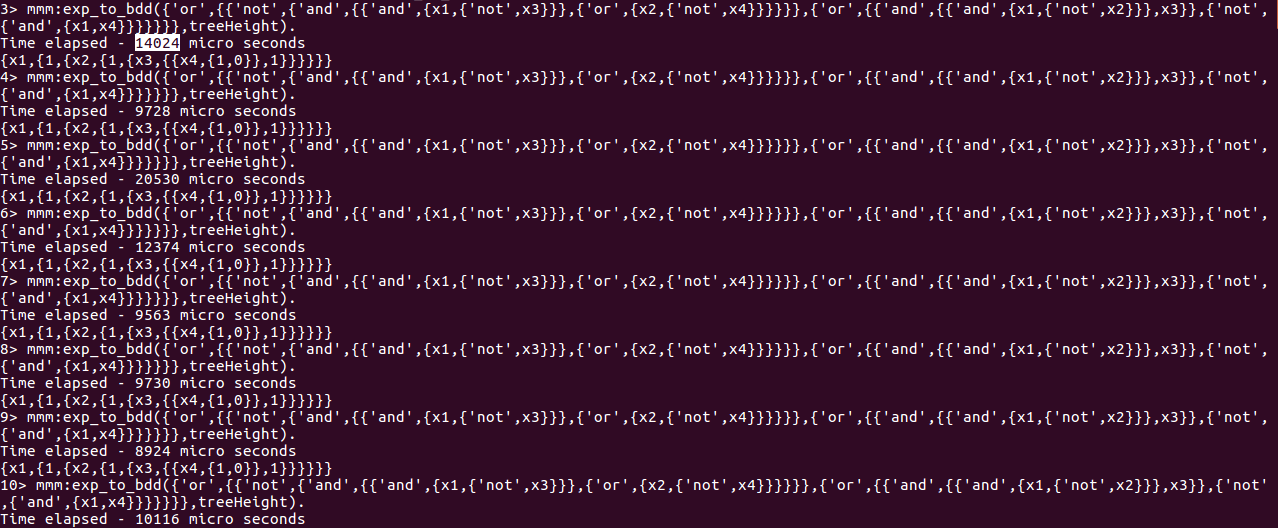
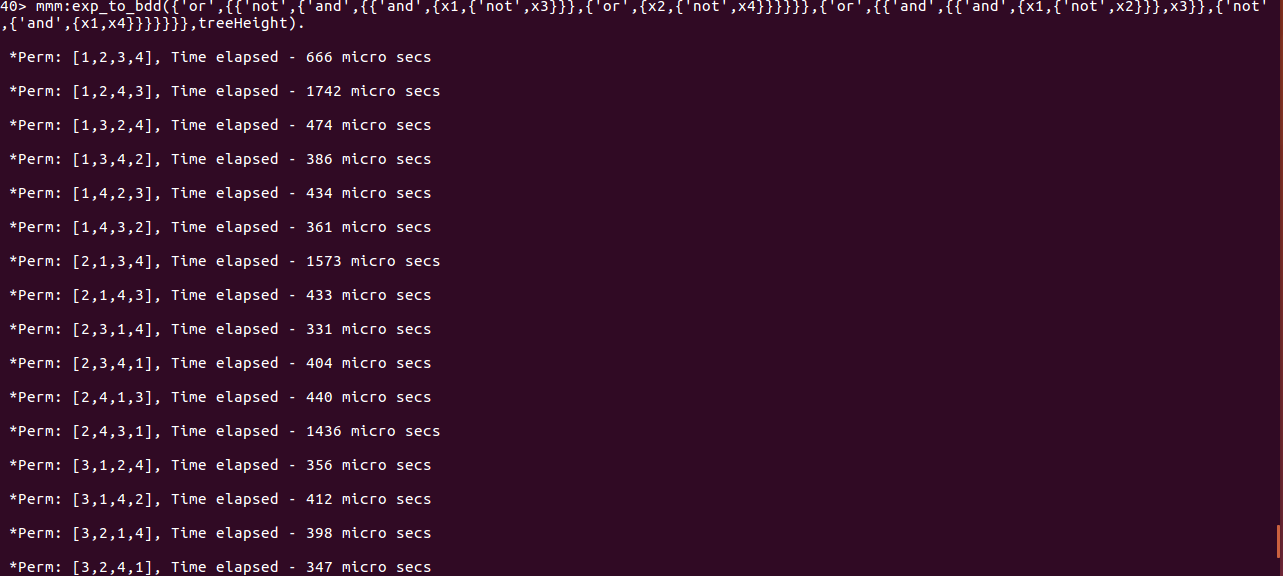


Fig2 exp\_to\_bdd different runs

* 1. **Measuring each permutation:** when measuring the time elapsed to create a specific permutation related tree we get an average of 400-500 micro secs. But again, the times differs a lot from run to run. Fig 3 shows the different permutation times on a single run of exp\_to\_bdd/1 and Fig4 will compare and show how variant are the times of each permutation in multiple runs of exp\_to\_bdd.



**Fig3** single run of exp\_to\_bdd measuring each permutation

Fig4 Each series represent the elapsed time of single permutation over different runs.

Fig5 average elapsed time for different permutations.

As one can see, Fig4 show complete chaos. But after doing the average on the runs we get the picture that **the first permutation ([1,2,3,4]) Is the fastest**. Whether it's because it's the first one that is executed or not, this is the fasted permutation on my computer at least. It's hard to explain the results because they are not consistent.

1. **Conclusions from the Assignment:** My conclusion is that erlang Is very friendly and useful language when it comes to manipulation lists and data. The language encourages you to use recursion which is a powerful technic and makes the programing faster.
2. **Paralleling the Assignment:** If we had to make this project parallel I think the best way to do it will be to create processes as the number of permutations and each process will create and calculate the **Ordering** measurement and a father process will determine which permutation is the most efficient. This will reduce the run time from X to X/n! (when n is the number of variables).