Binary Decision Diagram
Mid Project

תכנות פונקציונלי במערכות מבוזרות ומקבילות - 381.1.0112

מגיש: עומר לוכסמבורג 205500390

תיאור מבנהBDD

כדי לתאר את עץ ה ,BDD-השתמתי בתכונות tuple של ארלנג.

כל שורש בנוי כך: {Variable, Left, Right} כאשר Left, Right, הם עצים בפני עצמם.

- Variableערך הצומת שבה אנו נמצאים, יהווה משתנה מסוים מתוך המשתנים שניתנו בפונקציה הבוליאנית.

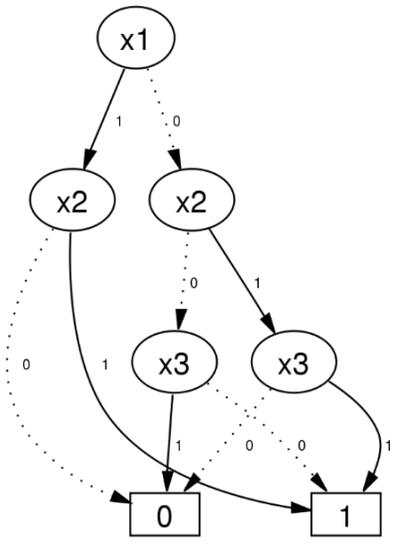
- Right, Leftכאמור, עץ בפני עצמו. יכול להיות ערך מספרי סופי - 1 או 0 (עלה תשובה(

- Left - מייצג את תת-העץ עבור.Variable=0
- Right - מייצג את תת-העץ עבור.Variable=1

*יתכן כי קיים עץ בעל ערך Variable בלבד, לכן בהכרח עבור העץ הנ"ל Variable שייך ל-{0,1.}

<u>דוגמה:</u> עבור העץ הנתון שנפרש ע"י פונקציה בוליאנית ופרמוטציה ספציפית של המשתנים נקבל את הביטוי הבא כ :tuple-

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



<u>יתרונות וחסרונות של הייצוג הנ"ל:</u>

את העץ ייצגתי בצורת pre-order, כלומר בתצורה שבה השורש נמצא טופולוגית לפני הבנים שלו, כאשר האיבר הראשון בביטוי של ה-BDD יהיה ישר השורש.

+ הקריאה בשיטה זו נוחה יותר למשתמש. קריאה משמאל לימין תהיה קריאה של העץ מלמעלה למטה.
+ לצורך חישוב העץ, לא נצטרך לבצע חישובים מיותרים, של תתי עצים שנפסלו כבר מההחלטה הראשונה. (למשל עבור השמה של x1=1 בדוגמה לעיל, נדלג על 3 צמתים בחישוב התוצאה)
− הבעיה בשיטה זו היא שהניתוח הוא רקורסיבי. לא נוכל לגשת לצומת מסוים מבלי לעבור במסלול מהשורש אליו. אך, עבור השימוש שנעשה בתוכנית, אין אנו צריכים לבצע גישה מיידית לצומת.

תיאור פונקציות חיצוניות:

בחלק זה נתאר את הפונקציות הראשיות של התוכנית שנדרשו למימוש.

exp_tobdd(BoolFunc, Ordering) -> BddTree

פונקציה זו מקבלת 2 פרמטרים, הפונקציה הבוליאנית המוגדרת לפי הוראות המשימה - BoolFunc, ואת סדר בחירת העץ - Ordering - כאשר סדר זה מוגבל ל-3 אופציות בלבד וייבחר העץ בעל הפרמטר הנ"ל המינימלי מבין כל העצים שנוצרו.

Ordering הוא אחד מהבאים - {tree_height,num_of_nodes,num_of_leafs} אחרת תוחזר הודעת שגיאה.

```
%%% MAIN FUNCTION:
%% exp_to_bdd - The function receives a Boolean function and returns the corresponding BDD tree
% representation for that Boolean function, by the ordering it'll chose the most efficient one
exp_to_bdd(BoolFunc, Ordering) ->
  Start = os:timestamp(), % saving time
  case Ordering of
    tree_height ->   % 2 is the place of the height
      Best_BDD = getMinOrder( BoolFunc, 2), % get the best bdd, from all kinds of bdds by method 'Ordering'
      Error = 0;
    num_of_nodes ->  % 3 is the place of the number of nodes
      Best_BDD = getMinOrder( BoolFunc, 3),
      Error = 0;
    num_of_leafs ->  % 4 is the place of the number of leaves
      Best_BDD = getMinOrder( BoolFunc, 4),
      Error = 0;
    _ -> Best_BDD = Ordering,
      Error =1
  end,
  case Error of
    0 -> io:format("Total time taken: ~f milliseconds~nAnd the answer is: ",
      [timer:now_diff(os:timestamp(), Start) / math:pow(10,3)]);
    1 -> io:fwrite("WRONG ORDERING METHOD!~nGiven the corresponding value: ")
  end,
  Best_BDD.
```

תחילה, נשמר זמן המערכת. לאחר מכן הפונקציה מתרגמת את ה-Ordering למספרים, שלאחר מכן יפוענחו וישלחו לפונקציה getMinOrder. אם ה-Ordering, תוחזר הודעת שגיאה.

פונקציית getMinOrder מוצאת את ה-BDD האופטימלי לפי ה-Ordering המבוקש.

* הסבר על getMinOrder(BoolFunc,Ordering) בחלק 'תיאור פונקציות פנימיות'.

solve_bdd(BddTree,Assignments) -> Res

פונקציה זו מקבלת 2 פרמטרים, עץ BDD נתון BddTree, לפי מבנה הנתונים שהוצג, וכן השמה של משתנים Assignments (הוגדר לפי המשימה כרשימה של זוגות סדורים).

```
%%% MAIN FUNCTION:
%% solve_bdd - solving bdd tree with the variables it got assigned
solve_bdd(BddTree,Assign) ->
  Start = os:timestamp(), % saving time
  FixedAssign = assignVar(true,1,assignVar(false,0,Assign)),    % changing true and false to 1,0
  Bdd_solution = solveIt(BddTree,FixedAssign),                  % solving...
  io:format("Total time taken: ~f milliseconds~n", [timer:now_diff(os:timestamp(), Start) / math:pow(10,3)]),
  Bdd_solution.
```

כמו בפונקציה הקודמת, נשמר זמן המערכת. לאחר מכן מתבצעת החלפה של השמות כאטומים {true,false} ל-{1,0}, כדי שנוכל לחשב את הערך לפי קריטריון ספציפי.

בהמשך, ערך הפתרון מתקבל כתוצאה מקריאה לפונקציה solveIt.

פונקציית solveIt מחזיר את ערך הפתרון לפי ההשמה - ערך הפתרון יכול להיות 0 או 1 או עץ BDD מצומצם יותר במידה ולא כל המשתנים קיבלו ערך.

* הסבר על solveIt(BddTree,Assignments) בחלק 'תיאור פונקציות פנימיות'.

תיאור פונקציות פנימיות:

comupteExp(BoolFunc) -> NewBoolFunc

הפונקציה מקבל ביטוי בוליאני, ומחשבת אותו לפי הערכים שהושמו בביטוי הבוליאני, ומצמצמת את הביטוי.

- שימוש בפונקציות פנימיות: אין.
- פונקציות שהשתמשו בפונקציה זו: exp2Tree.

assignVar(Variable,Value,BoolFunc) -> NewBoolFunc

הפונקציה מקבל תמשתנה Variable וערך השמה אליו Value, ומחליפה את כל המופעים של המשתנה בביטוי הבוליאני, ומחזירה את הביטוי הבוליאני לאחר השמה.

- שימוש בפונקציות פנימיות: אין.
- פונקציות שהשתמשו בפונקציה זו: exp2Tree, solve_bdd.

exp2Tree(BoolFunc,Permutation) -> {BDD,H,N,L}

הפונקציה מקבלת 2 משתנים. ביטוי בוליאני BF ופרמוטציה ספציפית Permutation, לפיו הפונקציה תתרגם את הביטוי הבוליאני לעץ BDD בצורת התבנית שהוגדרה בחלק הראשון - ותחזיר אותו.

בנוסף יישמרו הערכים של גובה, מספר קודקודים ומספר עלים - {BDD,#Height,#Nodes,#Leaves}

- שימוש בפונקציות פנימיות: compExp, assignVar, createNode.
- פונקציות שהשתמשו בפונקציה זו: getAllBdds.

createNode(Value,Left,Right) -> {Node,H,N,L}

הפונקציה מקבל 3 משתנים, שיאוחדו לתבנית צומת בעץ. כאשר צומת יכיל tuple של 3 הערכים הנ"ל.

כאשר ניתן לצמצם את הקודקוד - הפונקציה מבצעת צימצום (למשל עבור בנים זהים, הקודקוד שיוחזר יהיה אחד הבנים).

הפונקציה שומרת את הפרמטרים המעידים על ה-Ordering ב-tuple אחד -

{{Value,Left,Right}, #Height, #Nodes, #Leaves}

כאשר מתבצע חישוב של כל אחד מהם בהתאם לבנים הקיימים.

- שימוש בפונקציות פנימיות: אין.
- פונקציות שהשתמשו בפונקציה זו: exp2Tree.

getVars(BoolFunc) -> VarList

הפונקציה מקבל את הביטוי הבוליאני, ומחזירה רשימה של המשתנים הנמצאים בו.

- שימוש בפונקציות פנימיות: אין.
- פונקציות שהשתמשו בפונקציה זו: getMinOrder.

cleanDuplicates(List) -> NewList

הפונקציה מקבלת רשימה ומחזירה רשימה נקייה משכפולים.

- שימוש בפונקציות פנימיות: אין.
- פונקציות שהשתמשו בפונקציה זו: getVars.

listPerms(List) -> Permutations

הפונקציה מקבלת רשימה של ערכים ומחזירה את רשימה של כל הפרמוטציות שלהם.

- שימוש בפונקציות פנימיות: אין.
- פונקציות שהשתמשו בפונקציה זו: getMinOrder.

getAllBdds(BoolFunc, PermutationList) -> All_BDDs

הפונקציה מקבלת את הביטוי הבוליאני ואת רשימת כל הפרמוטציות של המשתנים שלו, ומחזירה רשימה של כל העצים לאותו הביטוי. הרשימה עוברת על כל פרמוטציה ובונה לה עץ BDD ייחודי מצומצם ככל הניתן.

- שימוש בפונקציות פנימיות: exp2Tree.
- פונקציות שהשתמשו בפונקציה זו: getMinOrder.

getMinOrder(BoolFunc, Ordering) -> BestBDD

הפונקציה נקראת על ידי הפונקציה הראשית exp_to_bdd עם הביטוי הבוליאני וכן Ordering שהוא ערך מספרי. הפונקציה משתמשת בפונקציות הפנימיות[1] כדי לייצר את כל סוגי ה-BDD האפשריים לביטוי הבוליאני, ולאחר מכן לפי Ordering, היא בודקת את הערך הנמצא ב-tuple של כל עץ - {L#,N#,H#,BDD} ובוחרת את המינימלי מבין כל העצים.

* Ordering יהיה 2, 3 או 4, וכך נוכל לבקש את הערך באינדקס של Ordering ב-tuple ע"י שימוש בפונקציה element(Ordering,BDD.)

- [1] שימוש בפונקציות פנימיות: listPerms, getVars, getAllBdds.
- פונקציות שהשתמשו בפונקציה זו: exp_to_bdd.

solveIt(BDD,Assignments) -> Solution

הפונקציה מקבלת עץ BDD מוכן ורשימת השמות Assignments, ומחזירה את הפתרון.

במקרה בו אין השמה לערך מסוים, יוחזר העץ המינימלי האפשרי שייתכן יהיה גם פתרון דטרמיניסטי של ערכי 0 או 1 - כלומר הפתרון יכול להיות מהצורה 0, 1 או BDD.

הפונקציה מתקדמת מהשורש כלפי פנים לפי ההשמות שקיבלה ב-Assignments, וכך בוחרת לאיזה תת עץ להתקדם. אם אין ערך לצומת הנוכחי, היא מחזירה את העץ עם הבנים תחתיו באופן רקורסיבי.

- שימוש בפונקציות פנימיות: אין.
- פונקציות שהשתמשו בפונקציה זו: solve_bdd.

בדיקת התוכנית לפונקציה ספציפית:

נתונה הפונקציה הבאה:

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

ראשית נרשום אותה כביטוי בוליאני כפי שהוגדר במטלה, ב-shell של ארלנג:

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

נבדוק את זמן התגובה של למציאת ה-BDD של הביטוי לפי כל אחת משיטות ה-Ordering, לפי פונקציית exp_to_bdd:

```
2> BF = {'or',{{'or',{{'and',{x1,{'and',{{'not',x2},x3}}}},
2> {'not',{'and',{{'and',{x1,{'not',x3}}},{'or',{{'not',x4},x2}}}}}},{'not',{'and',{x4,x1}}}}}.
{'or',{{'or',{{'and',{x1,{'and',{{'not',x2},x3}}}},
              {'not',{'and',{{'and',{x1,{'not',x3}}},
                             {'or',{{'not',x4},x2}}}}}},
       {'not',{'and',{x4,x1}}}}}
3> BDD_Height = exf_205500390:exp_to_bdd( BF, tree_height).
Total time taken: 1.076000 milliseconds
And the answer is: {x4,1,{x3,{x1,1,{x2,1,0}},1}}
4> BDD_Nodes = exf_205500390:exp_to_bdd( BF, num_of_nodes).
Total time taken: 1.428000 milliseconds
And the answer is: {x4,1,{x3,{x1,1,{x2,1,0}},1}}
5> BDD_Leafs = exf_205500390:exp_to_bdd( BF, num_of_leafs).
Total time taken: 1.803000 milliseconds
And the answer is: {x4,1,{x3,{x1,1,{x2,1,0}},1}}
6> BDD_error =  exf_205500390:exp_to_bdd( BF, error_maker).
WRONG ORDERING METHOD!
Given the corresponding value: error_maker
```

Ordering	Total time taken (miliseconds)	Optimal BDD
Height	1.076	**{x4,1,{x3,{x1,1,{x2,1,0}},1}}**
Number of Nodes	1.428	**{x4,1,{x3,{x1,1,{x2,1,0}},1}}**
Number of Leafs	1.803	**{x4,1,{x3,{x1,1,{x2,1,0}},1}}**
Other	-	**Error**

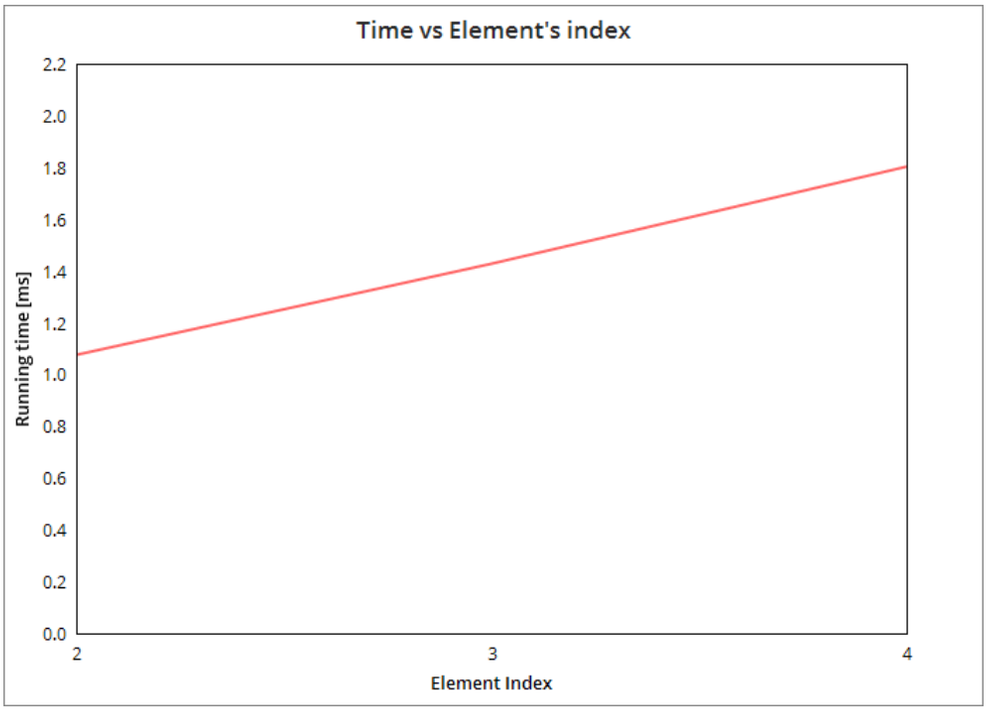
ניתן לראות כי כל הזמנים זהים יחסית וזאת מפני שאופן בחירת האלמנט לפיו ניקח את המשתנה העץ האופטימלי הוא רק לפי האובייקט של כל BDD, כלומר לפי ה-tuple הנ"ל- {BDD,Height,Nodes,Leafs}, ולכן יצירת כל העצים היא מהות כל הזמן שנלקח, ולאחר מכן חיפוש האופטימלי יעשה ע"י השוואת המיקום הספציפי בכל tuple המייצג עץ.

כלומר הזמן לא אמור להיות תלוי ב-Ordering שלפיו נבחר העץ.

בעמוד הבא השערה להפרש הזמנים המזערי.

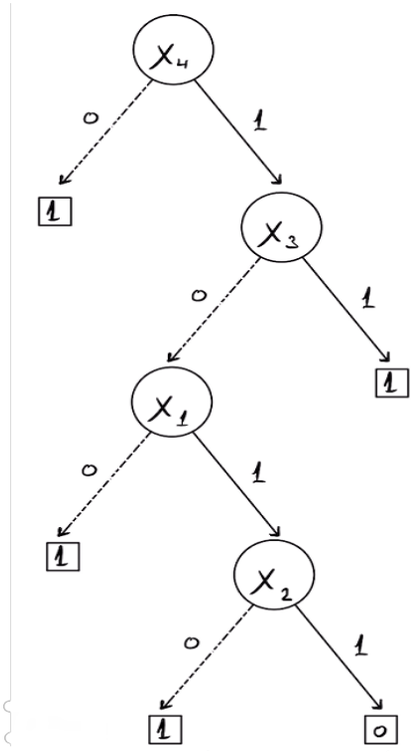
<u>השערה להפרש הזמנים:</u> ניתן לראות כי כאשר אנו מחפשים ערכים ב-tuple שיותר רחוקים מהאינדקס הראשון, כך גם הזמן עולה במקצת. לכן ההשערה היא כי מיקום הערך ב-tuple משפיע על מהירות שליפת המידע.

גרף המתאר את התוצאות:



גרף 1 זמן חישוב כפונקציה של אינדקס איבר ב-tuple

נשים לב שבעבור כל סוג של סידור קיבלנו את העץ הבא:



איור 1 עץ BDD שהתקבל

כעת עבור כל אחד מהעצים נבצע פתרון ע"י פונקציית solve_bdd:

```
8> BDD.
{x4,1,{x3,{x1,1,{x2,1,0}},1}}
9> Solution0000 = exf_205500390:solve_bdd( BDD, [{x1,0},{x2,0},{x3,0},{x4,0}] ).
Total time taken: 0.003000 milliseconds
1
10> Solution0001 = exf_205500390:solve_bdd( BDD, [{x1,0},{x2,0},{x3,0},{x4,1}] ).
Total time taken: 0.002000 milliseconds
1
11> Solution0010 = exf_205500390:solve_bdd( BDD, [{x1,0},{x2,0},{x3,1},{x4,0}] ).
Total time taken: 0.002000 milliseconds
1
12> Solution0011 = exf_205500390:solve_bdd( BDD, [{x1,0},{x2,0},{x3,1},{x4,1}] ).
Total time taken: 0.002000 milliseconds
1
13> Solution0100 = exf_205500390:solve_bdd( BDD, [{x1,0},{x2,1},{x3,0},{x4,0}] ).
Total time taken: 0.004000 milliseconds
1
14> Solution0101 = exf_205500390:solve_bdd( BDD, [{x1,0},{x2,1},{x3,0},{x4,1}] ).
Total time taken: 0.002000 milliseconds
1
15> Solution0110 = exf_205500390:solve_bdd( BDD, [{x1,0},{x2,1},{x3,1},{x4,0}] ).
Total time taken: 0.001000 milliseconds
1
16> Solution0111 = exf_205500390:solve_bdd( BDD, [{x1,0},{x2,1},{x3,1},{x4,1}] ).
Total time taken: 0.001000 milliseconds
1
17> Solution1000 = exf_205500390:solve_bdd( BDD, [{x1,1},{x2,0},{x3,0},{x4,0}] ).
Total time taken: 0.002000 milliseconds
1
18> Solution1001 = exf_205500390:solve_bdd( BDD, [{x1,1},{x2,0},{x3,0},{x4,1}] ).
Total time taken: 0.002000 milliseconds
1
19> Solution1010 = exf_205500390:solve_bdd( BDD, [{x1,1},{x2,0},{x3,1},{x4,0}] ).
Total time taken: 0.001000 milliseconds
1
20> Solution1011 = exf_205500390:solve_bdd( BDD, [{x1,1},{x2,0},{x3,1},{x4,1}] ).
Total time taken: 0.002000 milliseconds
1
21> Solution1100 = exf_205500390:solve_bdd( BDD, [{x1,1},{x2,1},{x3,0},{x4,0}] ).
Total time taken: 0.002000 milliseconds
1
22> Solution1101 = exf_205500390:solve_bdd( BDD, [{x1,1},{x2,1},{x3,0},{x4,1}] ).
Total time taken: 0.002000 milliseconds
0
23> Solution1110 = exf_205500390:solve_bdd( BDD, [{x1,1},{x2,1},{x3,1},{x4,0}] ).
Total time taken: 0.001000 milliseconds
1
24> Solution1111 = exf_205500390:solve_bdd( BDD, [{x1,1},{x2,1},{x3,1},{x4,1}] ).
Total time taken: 0.008000 milliseconds
1
```

נשים לב שזו בדיוק טבלת האמת של הפונקציה הנתונה:

x_1	x_2	x_3	x_4	$F(x_1, x_2, x_3, x_4)$	**Total time taken (miliseconds)**
0	0	0	0	1	0.003
0	0	0	1	1	0.002
0	0	1	0	1	0.002
0	0	1	1	1	0.002
0	1	0	0	1	0.004
0	1	0	1	1	0.002
0	1	1	0	1	0.001
0	1	1	1	1	0.001
1	0	0	0	1	0.002
1	0	0	1	1	0.002
1	0	1	0	1	0.001
1	0	1	1	1	0.002
1	1	0	0	1	0.002
1	1	0	1	0	0.002
1	1	1	0	1	0.001
1	1	1	1	1	0.008

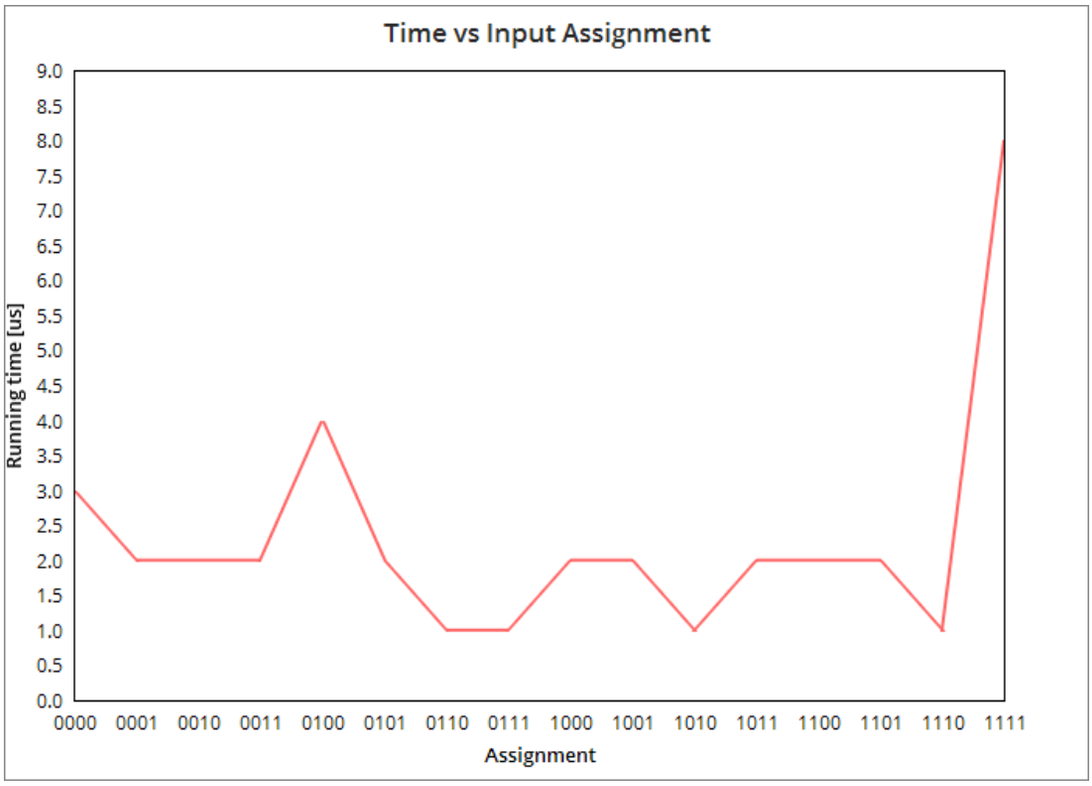
ניתן לשים לב כי זמן הפתרון הוא יחסית זהה לכל השמה אפשרית (בין 1~8 מיקרו-שניות - לאחר הרצה כמה פעמים ניתן לשים לב כי כל התוצאות נעות בתחום זה).

קיימת ריצה יותר טוב כאשר נבחר משתנה שישר נותן תוצאה, כמו למשל במקרה של הפונקציה הנתונה, כאשר $x_4=0$, נקבל כי ישר נבחר תת העץ השמאלי שערכו הוא 1.

ובאופן כללי - נשים לב כי מאחר ופנוקצית הפתרון solve_bdd פותרת את העץ על ידי התקדמות לפי קודקודים, לכן נקבל כי ריצת הפתרון תהיה O(Height) - כלומר לפי גובה העץ, מכוון שמסלול הפתרון יהיה לכל היותר כגובהו של העץ.

לכן נצפה שריצת הפתרון עבור עצים בעלי Ordering = Height תיהיה הטובה ביותר. כמובן שבדוגמה זו ובדוגמאות אחרות יהיה קשה לשים לב להבדל מכיוון שהדוגמאות יהיו בעלות פרמטרים קרובים מאוד (למשל בפונקציה הנתונה, לכל העצים אותו הגובה, למרות ששיטת ה- Ordering שונה).

בעמוד הבא, גרף המתאר את זמני הריצה אל מול ההשמות השונות.



גרף 2 זמן חישוב כפונקציה של השמה

* כאמור, גרף זה לא משקף באופן וודאי את ממוצע התוצאות עבור כל כניסה.

מסקנות:

לאחר עבודה על המטלה, קיבלתי ראשית כלים טכניים להתמודדות עם תכנות בארלנג.

בנוסף שמתי לב כי ככל שנעלה את כמות המשתנים, זמן החישוב לפונקציה שמפענחת את הביטויים הבוליאניים יעלה בהתאם. הסיבה טמונה בעובדה שאנו מחשבים עץ BDD לכל פרמוטציה שונה של הפונקציה הבוליאנית.

הבעיה במימוש BDD מינימלי לפי סדר מסוים הוא שאיננו יודעים מראש מהי הפרמוטציה שתתן לנו את התוצאה הטובה והמינימלית ביותר.

בתוכנית שכתבתי אמנם זמני הריצה קטנים לפי הדוגמה, אך עבור תוכנית עם n פרמטרים, זמן החישוב יהיה כסדר גודל של מספר הפרמוטציות !.n

הדבר מוביל אותנו לחשיבה מקבילית - אם נשתמש במאגר ת'רדים שכל אחד מהם יריץ חישוב של יצירת BDD לפי פרמוטציה מסוימת, נוכל לקצר את זמן הריצה בצורה משמעותית.

לעומת זאת, אם נבצע התייעלות זהה במציאת פתרון ל-BDD מסוים, יתכן שנחשב קודקודים מיותרים (כמו בדוגמה הנתונה - כאשר $x_4=0$ אין צורך לחשב אף קודקוד אחר).

ממטלה זאת נוכל ללמוד בסופו של דבר על היעילות של ריצה מקבילית לעומת ריצה טורית.

הצעות לביצוע ריצה במקביל בכל אחת מהפונקציות:

- עבור הפונקציה של מציאת BDD אופטימלי לפי סדר מסוים - ניתן לחשב כמה BDD-ים במקביל, ולבסוף לאגד את כולם לתוך רשימה, ולבסוף לבחור את האופטימלי. זמן הריצה יוקטן פי מספר הת'רדים העובדים.
- עבור הפונקציה שפותרת את ה-BDD לפי השמה מסוימת - ניתן לחשב את כל הערך בכל קודקוד כאשר כל ת'רד יתחיל מעלה אחר. שיטה זו פחות יעילה בפעמים מסוימות (יעיל יותר לעץ שלם).

תוספות:

הגדרתי את הפונקציה הבאה:

booleanGenerator(NumOfVars,NumOfEquations) -> [Bf1, Bf2,...]

הפונקציה מקבל פרמטר NumOfVars שיקבע את כמות המשתנים $\{x_1,x_2...x_n\}$ וכן פרמטר NumOfEquations שיקבע את כמות המשוואות.

הפונקציה מחזירה רשימה של פונקציות בולינאיות, כמספר NumOfEquations עם כמות המשתנים בהתאם.

בפונקציה פנימית – randomBoolFunc(N,M) - מקבלת משתנה N, שהוא מספר המשתנים, ובנוסף הוגדר משתנה M שמגדיר את הכמות המקסימלית של אופרטור בתוך אופרטור (כדי שהפונקציה לא תרוץ הרבה זמן ללא בקרה) ומחזיר פונקציה בוליאנית באופן רנדומלי, לפי התבנית שהוגדרה במטלה. את M בחרתי על ידי המשתנה N+1.

להלן התוצאות:

```
4> List = exfb_205500390:booleanGenerator(4,3).
[{'and',{{'not',{'or',{{'and',{{'not',x2},
                                {'not',{'or',{{'or',{{'or',{{'not',{'or',{{...},...}}},
                                                                {'not',x1}}},
                                                        x2}},
                                                  {'or',{{'not',x4},{'not',{'and',{{'not',...},{...}}}}}}}}}},
                         x4}}},
          {'not',x4}}},
 {'or',{{'not',x3},x3}},
 {'or',{{'not',{'or',{{'and',{x1,{'not',{'and',{x4,x1}}}}},
                      {'or',{{'not',x3},
                             {'or',{{'not',{'and',{{'not',x3},{'not',{'and',{...}}}}}},
                                    {'not',x2}}}}}}},
        x4}}]
```

ולאחר בדיקה עם אחת הפונקציות קיבלנו תשובות תקינות:

```
5> exfb_205500390:exp_to_bdd(lists:nth(1,List),tree_height).
Total time taken: 1.596000 milliseconds
And the answer is: {x4,1,0}
6> exfb_205500390:exp_to_bdd(lists:nth(2,List),tree_height).
Total time taken: 0.015000 milliseconds
And the answer is: 1
7> exfb_205500390:exp_to_bdd(lists:nth(3,List),tree_height).
Total time taken: 0.954000 milliseconds
And the answer is: {x4,0,1}
```

נספחים:

הגדרת המטלה מצורפת מהעמוד הבא.

Ben-Gurion University of the Negev

Communication Systems Engineering Department

381-1-0112 Functional Programming in Concurrent and Distributed Systems

Submission Procedure

Report file has to be pdf: exf_<ID>.pdf

Main module is named as: exf_<ID>.erl

Upload separately these two files to moodle. Do not upload any zip.

Wrong submission procedure leads to penalty of 10 points! which means maximal possible grade is 90.

The program would be checked on Ubuntu 18.04 with Erlang 21. You have to ensure that your code is compatible with this system, otherwise you risk at failure.

Sequential Erlang - Assignment

Introduction

In this assignment, you are asked to implement 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 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
2. Eliminate any node whose two children are isomorphic

The construction of BDD is based on Shannon expansion theory:

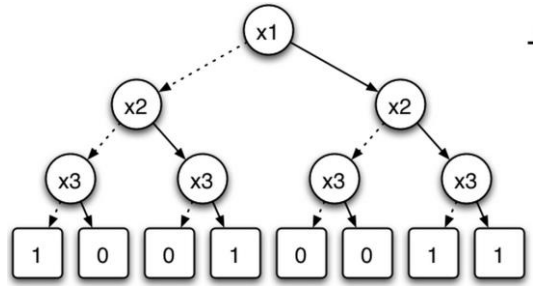
$f(x_1, x_2, \dots, x_n) = x_1 \cdot f(1, x_2, x_3, \dots, x_n) + \overline{x_1} \cdot f(0, x_2, x_3, \dots, x_n)$

Example

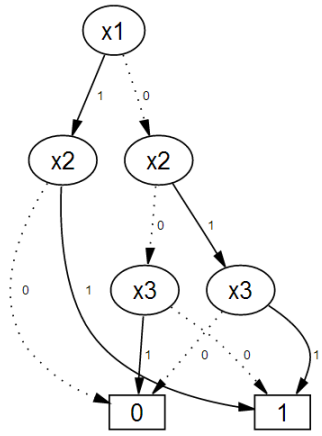
The Boolean function $f(x_1, x_2, x_3) = \overline{x_1}\,\overline{x_2}\,\overline{x_3} + x_1\,x_2 + x_2\,x_3$ with the following truth table:

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.

Technical Details

Your mission is to implement two functions:

1. **spec 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**.
        i. Variable **Ordering** can be one of the following atoms: **tree_height**, **num_of_nodes** or **num_of_leafs**.
        ii. In order to extract the most efficient 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: $f_s(x_1, x_2, x_3)$. 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: $\{\{x_1, x_2, x_3\}, \{x_1, x_3, x_2\}, \{x_2, x_1, x_3\}, \{x_2, x_3, x_1\}, \{x_3, x_2, x_1\}, \{x_3, x_1, x_2\}\}$
    c. The returned tree must be reduced by Rule 1. Read more about in BDD Introduction.

2. **spec 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 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.

When returning from each function call, **the execution time must be printed**.

Note: signatures of these functions have to be exactly the same as specs. This assignment will be checked by a testing machine. Any mismatches might lead to points lose.

<u>Input definition:</u>

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
- First element of an operator tuple is an atom.

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

Will be represented as

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

Your BDD tree needs to follow the following structure:

- An empty tree (root only)

  {Val}
- A node with two sons

  {Left, Right}
- A node with a single son (e.g. a left son)

  {Left, {Val}}
- A leaf (represents a result)

  {Val}

You can pick any data structure to be the tree's data structure but you must supply detailed explanation in your report why it was picked and also count advantages and disadvantages.

Important notes

1. Make sure that your code is well documented, efficient and tested.
2. The machine needs to identify the variables that take part in the Boolean expression, there is no guarantee that the variables will follow the structure of **x1,x2,…** or any other structure.
3. You are required to submit a short design report to your project with detailed explanation about your BDD tree structure.
4. You are required to add a project report.
   a. The report is performed on the following Boolean function:
   $$f_r(x_1, x_2, x_3, x_4) = x_1\,\overline{x_2}\,x_3 + \overline{x_1\overline{x_3}(\overline{x_4} + x_2)} + \overline{x_4 x_1}$$
   b. Measure the Time elapsed for your program to convert f_r Boolean function to a BDD. Repeat on this stage each time with different ordering argument.
   Write the results in table.
   c. To each of the trees created in 'b' apply solve_bdd function with all the possible inputs. Measure the time it took to each tree. Write the results in table and mention which is the most efficient.
   d. Explain the results, You can use any graphical tool to generate graphs and plots which support your explanations.
   e. Write your conclusions from working on this assignment.
   f. Suggest an approach of a parallel computing of this problem (no implementation, only explanation).
5. Bonus part of 5 extra points (Maximum grade for assignment is still 100). Implement the following function:
   booleanGenerator(NumOfVars,NumOfEquations)->[Eq1,Eq2,Eq3…]
   Eq<#> is a Boolean equation given in the format of tuples (As mentioned in "input definitions" section).
   booleanGenerator function generates list of equations with length NumOfEquations.
   Each equation is composed of NumOfVars variables $(x_1, x_2, \dots, x_{NumOfVars})$.
   Export this function and change your erlang file name to: exfb_<ID>.erl
   Otherwise the automatic checking machine will not check the bonus part.