

Binary Decision Diagram

Mid Project

תכנות פונקציונלי במערכות מבוזרות ומקבילות - 381.1.0112

מגיש: עומר לוכסמבורג

תיאור מבנה BDD

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.

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

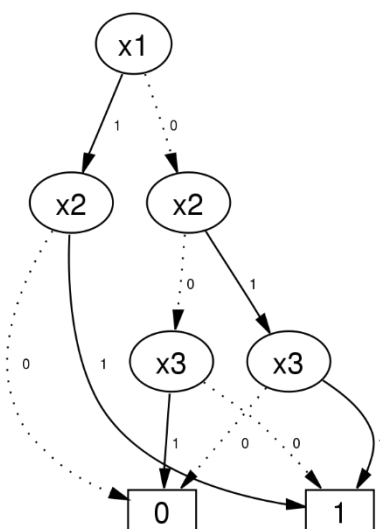
Left, Right כאמור, עץ בפני עצמו. יכול להיות ערך מספרי סופי - 1 או 0 (עלה תשובה)

- 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}.

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

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



יתרונות וחסרונות של הייצוג הנ"ל:

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 מהחלטה הראשונה. (למשל עבור השמה של $x_1=1$ 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 מיידיית 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:fprintf("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: solve_bdd, exp2Tree.

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 - {Leaves#,Nodes#,Height#,BDD}

- 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 יכול 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:

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

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 (milliseconds)	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

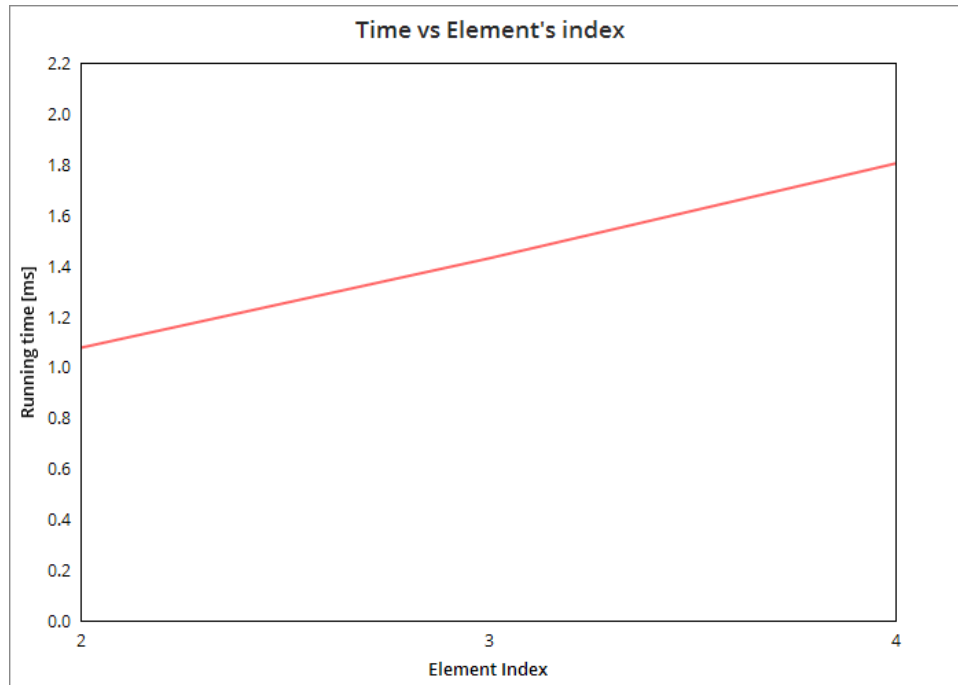
ניתן לראות כי כל הזמנים זהים יחסית וזאת מפני שאופן בחירת האלמנט לפיו ניקח את המשתנה העץ האופטימלי הוא רק לפי האובייקט של כל BDD, כלומר לפי ה-tuple הנ"ל-
{BDD,Height,Nodes,Leafs}, ולכן יצירת כל העצים היא מהות כל הזמן שנלקח, ולאחר מכן חיפוש האופטימלי יעשה ע"י השוואת המיקום הספציפי בכל tuple המייצג עץ.

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

בעמוד הבא השערה להפרש הזמנים המזערי.

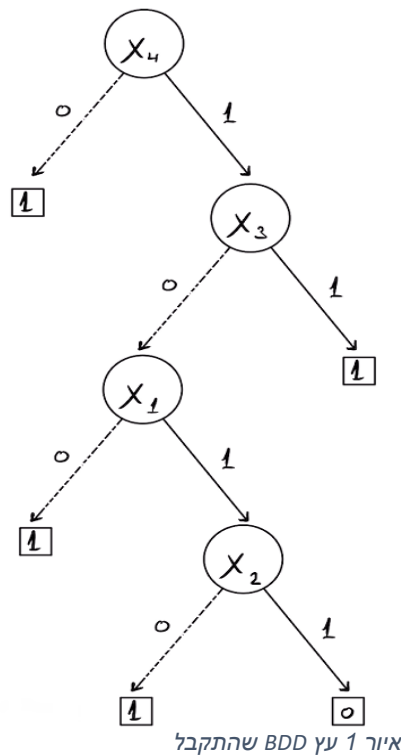
השערה להפרש הזמנים: ניתן לראות כי כאשר אנו מחפשים ערכים ב-tuple שיותר רחוקים מהאינדקס הראשון, כך גם הזמן עולה במקצת. לכן ההשערה היא כי מיקום הערך ב-tuple משפיע על מהירות שליפת המידע.

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:



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 (milliseconds)
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

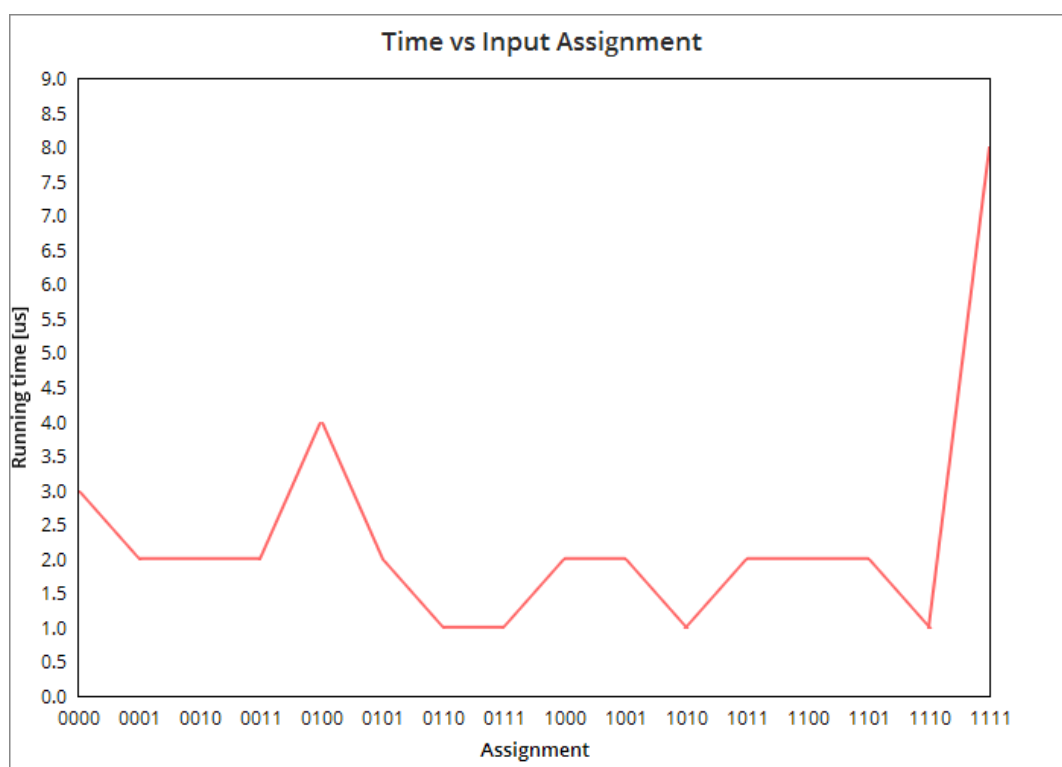
ניתן לשים לב כי זמן הפתרון הוא יחסית זהה לכל השמה אפשרית (בין 1~8 מיקרו-שניות - לאחר הרצה כמה פעמים ניתן לשים לב כי כל התוצאות נעות בתחום זה).

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 ופונקציית הפתרון solve_bdd פותרת את העץ על ידי התקדמות לפי קודקודים, לכן נקבל כי ריצת הפתרון תהיה $O(\text{Height})$ - כלומר לפי גובה העץ, מכיוון שמסלול הפתרון יהיה לכל היותר כגובהו של העץ.

לכן נצפה שריצת הפתרון עבור עצים בעלי $\text{Ordering} = \text{Height}$ תהיה 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 $\{x_1, x_2, \dots, x_n\}$ וכן פרמטר `NumOfEquations` שיקבע את כמות המשוואות.

הפונקציה מחזירה רשימה של פונקציות בוליאניות, 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',...},{...}}}}}}}],
{'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.