

Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológií

Dátové štruktúry a algoritmy

Zadanie 2

Binárne rozhodovacie diagramy

Ján Sližik

Obsah

1. Úvod	3
1.1 Obmedzenia môjho riešenia	3
1.2 Definovanie základných pojmov	3
2. Základné štruktúry	4
2.1 BDD Strom	4
2.2. Lineárna Hashovacia tabuľka	5
3. Funkcie Binárneho Rozhodovacieho Diagramu	6
3.1. bddCreate()	6
3.2. bddUse()	7
4. Overenie správnosti riešenia	8
5. Demo	9
5.1. demoTree("AC+AB!B", "CAB", "101")	9
5.2. demoTree("A!C+ABC+!AB+!BC", "CAB", "101")	10
6. Testovanie	11
7. Grafy	12
7.1 Zložitosti hlavných funkcií	15
8. Záver	15
9. Zdroje	15

1. Úvod

Cieľom tohto zadania je zoznámiť sa s konceptom binárnych rozhodovacích diagramov, dátovou štruktúrou, ktorá sa využíva napríklad na reprezentáciu booleovských funkcií. Na to aby bolo moje riešenie, čo najefektívnejšie rozhodol som sa využívať dátové štruktúry, ktoré som dopodrobna opísal v minulom zadaní, konkrétne, pole Lineárnych Hashovacích tabuliek a Binárny strom, ich implementáciu som však upravil konkrétne pre potreby problému zostavenia binárneho rozhodovacieho diagramu z výrazu v tvare UDNF.

1.1 Obmedzenia môjho riešenia

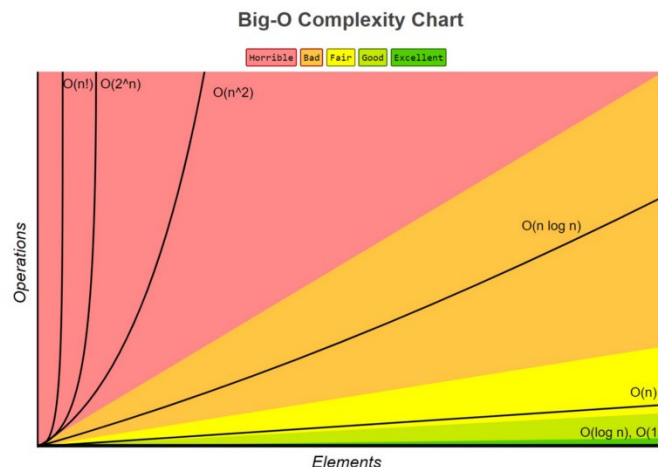
V mojom programe je disjunkcia reprezentovaná ako +, konjunkcia sa nezapíše a negácia je reprezentovaná ako !. Na to, aby nastala, čo najvyššia miera redukcie by mali byť jednotlivé slová oddelené + v abecednom poradí. Poradie premenných a vektor ohodnotenia by mali obsahovať všetky premenné, ktoré daný výraz obsahuje na to, aby bol výsledok správny.

1.2 Definovanie základných pojmov

Binárny strom je dátová štruktúra, ktorej každý vrchol má dvoch potomkov. Na to, aby bolo jednoduché prejsť vytvoreným stromom, napravo bude funkcia pre ktorú premenná nadobúda pozitívnu hodnotu a naľavo negatívnu, pokiaľ sa premenná v slove nenachádza ide na obe strany.

Hashovacia tabuľka je dátovou štruktúrou, ktorá umožňuje v najlepšom prípade prístup k prvkom o výpočtovej zložitosti $O(1)$ a zabezpečuje, že sa údaj bude nachádzať v tabuľke iba raz, vďaka asociácií kľúčov, v mojom prípade booleovských výrazov pretransformovaných pomocou hashovacej funkcie s hodnotami umiestnenia údajov, uzlov stromu v tabuľke.

Výpočtová zložitosť vyjadruje závislosť časových a pamäťových nárokov algoritmu, ktoré sú priamo úmerné veľkosti, a náročnosti riešeného problému.



2. Základné štruktúry

2.1 BDD Strom

Kmeň stromu, alebo tzv. BddTrunk sa skladá z informácie o jeho veľkosti, počet jeho nodeov, očakávanej veľkosti pokiaľ by sa nevykonávala redukcia pomocou hashovacích tabuliek a premennej levels, ktorá je rovná počtu premenných s ktorými daný strom pracuje, ďalej je tu ukazovateľ na „koreň“ stromu, jeho prvý node a ukazovateľ na pole hashovacích tabuliek a zero, one node.

Samotný uzol, alebo zväz binárneho rozhodovacieho stromu má v sebe výraz a pole ukazovateľov o veľkosti dva na uzly vpravo a vľavo.

```
typedef struct BddTrunk {  
    struct BddNode* root,* zero,* one;  
    struct LinearCell*** table;  
    int size, levels, expected_size;  
} TREE;
```

```
typedef struct BddNode {  
    char expression[STRING];  
    struct BddNode *pointer[2];  
} NODE;
```

Na prácu so stromom sa používa mnoho pomocných funkcií: createNode() vráti ukazovateľ na nový node nainicializovaný na vloženú booleanskú funkciu, createTree() vráti ukazovateľ na nový strom, nainicializuje sa pole tabuliek spolu s koreňom stromu, insert() vloží ukazovateľ na nový node do vybraného nodeu, buď vľavo, alebo vpravo na základe vstupu, printTree() pomocou rekurzívnej funkcie output() vypíše strom spoločne s poľom tabuliek v ňom a informácie o jeho momentálnej veľkosti a jeho veľkosti keby nebola implementovaná redukcia pomocou hashovacích tabuliek, freeTree() strom spoločne s poľom hashovacích tabuliek uvoľní.

```
> NODE* createNode(char* new_expression) { ...  
> TREE* createTree(char* input, int order_size) { ...  
> void insert(NODE* node, NODE* new_node, int right) { ...  
> void output(NODE* current_node, int depth) { ...  
> void printTree(TREE* tree) { ...  
> void freeTree(TREE** tree) { ...
```

2.2. Lineárna Hashovacia tabuľka

Na redukcii sa používa hashovacia tabuľka pre každú úroveň vnorenia, resp. pre každú premennú. Na každej úrovni má tabuľka inú veľkosť 2^n , teda pokiaľ riešime v poradí tretiu premennú jej veľkosť bude 2^3 . 2^n je samozrejme najhoršia možnosť zaplnenia, väčšinu času sa však tabuľka nezaplní a miera kolízií je nízka, presne preto som sa rozhodol pre linear probing. Lineárna hashovacia tabuľka je dobrá práve vtedy pokiaľ nenastáva mnoho kolízií, nie je ju potrebné často resizovať a v tabuľke nemažeme, čo je presne tento prípad.

Bunka mojej hashovacej tabuľky pozostáva z kľúča reprezentovaného stringom, teda booleanskej funkcie a ukazovateľa na uzol stromu.

```
typedef struct LinearCell {  
    char key[STRING];  
    struct BddNode *node;  
} CELL;
```

Hashovacia funkcia je veľmi jednoduchá, kľúč, booleovská funkcia je postupne prechádzaný char za charom, momentálny znak je prekonvertovaný na ascii a prenasobený jeho pozíciou, aby “AB+C” nebolo nemapované na rovnaké miesto ako “A+BC”. Nakoniec je súčet znakov zmodulovaný veľkosťou hashovacej tabuľky.

```
int hashFunction(char* key, int table_size) {  
    int hash = 0;  
  
    int i = 0;  
    while(key[i] != '\0') {  
        hash += (int)key[i] * (i+1);  
        i++;  
    }  
    return (hash % table_size);  
}
```

Ďalšie pomocné funkcie slúžia na základnú prácu s tabuľkou newCellLinear() vytvorí novú bunku, initLinear() vráti ukazovateľ na novú tabuľku nastavenú na zadanú veľkosť, freeLinear() uvoľní tabuľku printLinear() vyprintuje. Kľúčová funkcia je searchLinear() ak sa v tabuľke nachádza vráti ukazovateľ na bunku, ak nie null a insertLinear() vytvorí a vloží do tabuľky novú bunku

```
> int hashFunction(char* key, int table_size) { ...  
> CELL* newcellLinear(char* new_key, NODE* new_node) { ...  
> CELL** initLinear(int table_size) { ...  
> void freeLinear(CELL** table, int table_size) { ...  
> void printLinear(CELL** table, int table_size) { ...  
> void insertLinear(CELL** table, char* key, NODE* node, int table_size) { ...  
> CELL* searchLinear(CELL** table, char* key, int table_size) { ...
```

3. Funkcie Binárneho Rozhodovacieho Diagramu

3.1. bddCreate()

bddCreate() je srdcom celého projektu, je to funkcia, ktorá slúži na vytvorenie stromu na základe vstupnej booleanskej funkcie a poradia premenných v ktorom majú byť vyhodnocované. Funkcia nainicializuje strom, createTree() a následne volá funkciu create(), ktorá strom naplní, uzly stromu sú však iba ukazovatele na bunky v hashovacej tabuľke, tým, že pre každú úroveň môže existovať daný výraz, uzol, iba raz je zabezpečená redukcia.

```
TREE* bddCreate(char* input, char *searching) {
    char tokenize[STRING];
    strcpy(tokenize, input);

    TREE* tree = NULL;
    int order_size = (int)strlen(searching);
    tree = createTree(input, order_size);
    create(tree, tree->root, tokenize, searching, 0);
    return tree;
}
```

Funkcia create je rekurzívnou funkciou, ktorá končí pokiaľ narazí na posledný index poradia premenných, alebo pokiaľ nový uzol má hodnotu 1 / 0. Pred tým než sa čokoľvek s daným uzlom stane je zanalyzovaný najskôr ako veta pomocou funkcie analyseSentence() zistí sa o ňom, či je šanca, že v ňom môže nastať kontradikcia A!A - ľavý uzol 0, pravý uzol 0. Či v ňom môže nastať negácia !A ľavý uzol 1, pravý uzol 0 a zistí sa počet plusov, teda počet slov.

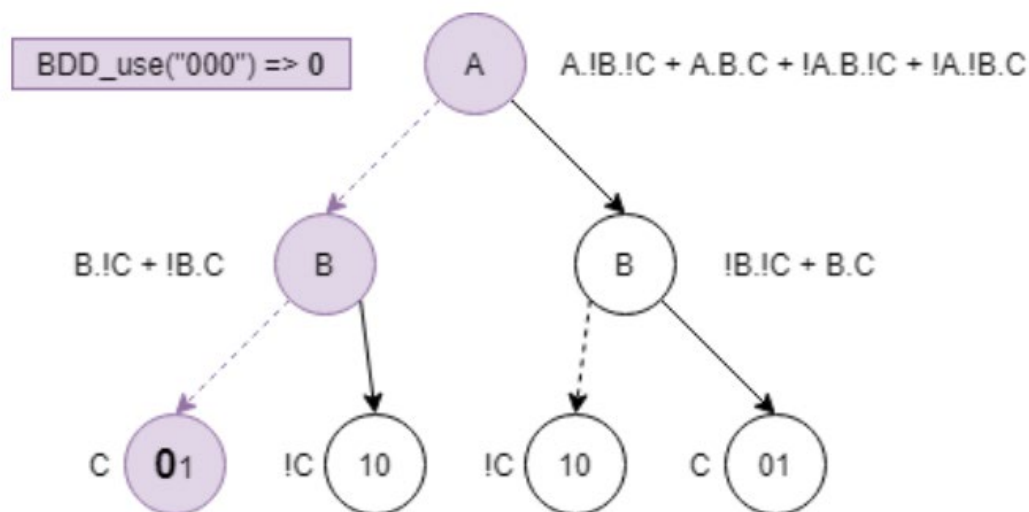
```
// BDD Create
char* replaceChar(char* input, char find, char replace){...
void analyseWord(char* token, char searched, int* positive, int *negative, int *single) {...
void analyseSentence(char* input, int* contradiction_possibility, int* negation_possibility, int* numOfPlus) {...
void create(TREE* tree, NODE* current_node, char* input, char* searching, int index) {...
```

Následne je daná veta rozdelená pomocou funkcie strtok(), ktorá ju rozdelí na základe delimitera +, každé slovo je zanalyzované, zistí sa, či sa jedná o samostatné hľadané písmeno, či slovo obsahuje negáciu, alebo pozitívnu formu vyhľadávaného písmena. Na základe zistených informácií sa slovo prekopíruje do ľavej, pravej strany, preskočí sa, alebo je nastavené na možnú z výstupných hodnôt.

Hľadaná booleanská funkcia pre danú stranu sa vyhladá v danej úrovni hashovacej tabuľky, ak sa už výraz v tabuľke nachádza do stromu sa vloží ukazovateľ naň a veľkosť pokiaľ by sa nevyužívala redukcia sa zinkrementuje. Ak sa v tabuľke nenachádza je vytvorený a jeho ukazovateľ je vložený do tabuľky aj do stromu, veľkosť stromu spolu s očakávanou veľkosťou sa zinkrementuje. Ak by daný výraz bol 1, alebo 0, tak sa zistí, či už 1, nula existuje v kmeni stromu a vloží sa jej ukazovateľ do tabuľky aj stromu, ak neexistuje vytvorí sa.

3.2. bddUse()

Pokiaľ bol strom dobre vytvorený volanie funkcie bddUse() je potom úplne jednoduché stačí jej poslať vektor reprezentujúci ohodnotenie premenných stromu a samotný strom. Pokiaľ strom obsahuje iba jednotky, alebo iba nuly vráti jednotku, alebo nulu inak funkcia prechádza stromom dovtedy než narazí na uzol s booleovskou funkciou 1 / 0 a vráti ich hodnotu, alebo než narazí na koniec vektora. Vektor číta takým spôsobom, že pokiaľ vidí 1 nastaví momentálny node na node napravo od momentálneho nódu a pokiaľ vidí nulu na momentálny node naľavo.



4. Overenie správnosti riešenia

Správnosť svojho riešenia testujem pomocou funkcie check(), ktorá vráti výslednú hodnotu pre daný výraz, prechádza výrazom na základe poradia a ak narázi na hľadaný znak nahradí ho na základe jeho vektorového ohodnotenia, napr. A má hodnotu 1 nachádza sa vo výraze, tak je zamenené za 1, pokiaľ by sa vo výraze nachádzalo !A je nahradené opakom vektora tj. 0. Z výrazu po prejdení všetkými premennými, sa odstránia všetky inštalácie ! a následne je rozdelený na slová pomocou funkcie strtok. Algoritmus prechádza slovo po slove ak sa v slove nachádza 1 bez toho, aby v ňom existovali 0 funkcia vráti 1, ak nie prechádza na ďalšie slovo. Ak prejde celou vetou vráti 0.

```
void removeChar(char * str, char charToRemmove){ ...  
char check(char*expression, char*order, char*vector) { ...
```

```
while(token != NULL) {  
    i = 0;  
    flag = 1;  
    while (token[i] != '\0') {  
        if (token[i] == '0')  
            flag = 0;  
            break;  
    }  
    if (flag)  
        return '1';
```

Program bol skontrolovaný aj na prostredí Linux pomocou nástroja valgrind pre demoTest()

```
kono@kono:~/Desktop$ gcc main.c -lm
```

Kde sa však musel skompilovať pomocou -lm, keďže obsahuje knižnicu <math.h>

```
==1629== HEAP SUMMARY:  
==1629==      in use at exit: 0 bytes in 0 blocks  
==1629==    total heap usage: 20 allocs, 20 frees, 2,720 bytes allocated  
==1629==  
==1629== All heap blocks were freed -- no leaks are possible  
==1629==  
==1629== For lists of detected and suppressed errors, rerun with: -s  
==1629== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```


5. Demo

Moja funkcia main obsahuje dva demo výrazy na vizualizáciu preberanej problematiky a overenie riešenia. Funkcia demo tree vytvorí strom, raz ním prejde a skontroluje, či jeho bddUse() je správne pomocou funkcie check().

5.1. demoTree("AC+AB!B", "CAB", "101")

```
Demo Tree Order ~ CAB
AC+AB!B
left
  AB!B
  left
    0
    end
  right
    B!B
    left
      0
      end
    right
      0
      end
    end
  end
right
  A+AB!B
  left
    0
    end
  right
    1
    end
  end
end
```

```
table[1]
0 | contains = AB!B
1 | contains = A+AB!B

table[2]
0 | contains = 0
1 | contains = 1
2 | contains = B!B
3 | NULL

table[3]
0 | contains = 0
1 | NULL
2 | NULL
3 | NULL
4 | NULL
5 | NULL
6 | NULL
7 | NULL
```

```
Actual Number of Nodes          6
Number of Nodes without Hashtable Reduction  9
Number of Nodes without Reduction  15

Output for bddUse(101)          0
```

5.2. demoTree("A!C+ABC+!AB+!BC", "CAB", "101")

```

Demo Tree Order ~ ABC
A!C+ABC+!AB+!BC
left
    B+!BC
    left
        C
        left
            0
            end
        right
            1
            end
        end
    right
        1
        end
    end
right
    !C+BC+!BC
    left
        !C+C
        left
            1
            end
        right
            1
            end
        end
    right
        !C+C
        left
            1
            end
        right
            1
            end
        end
    end
end

```

```

table[1]
0 | contains = B+!BC
1 | contains = !C+BC+!BC

table[2]
0 | contains = !C+C
1 | contains = 1
2 | NULL
3 | contains = C

table[3]
0 | contains = 0
1 | contains = 1
2 | NULL
3 | NULL
4 | NULL
5 | NULL
6 | NULL
7 | NULL

```








Actual Number of Nodes	7
Number of Nodes without Hashtable Reduction	13
Number of Nodes without Reduction	15
Output for bddUse(100)	1

6. Testovanie

Na to aby som zabezpečil, že je možné testovanie opakovať za rovnakých podmienok som si napísal generátor booleovských funkcií v pythone. Generátor vytvorí 5000 náhodných funkcií pre 13, 15 a 20 premenných spoločne so všetkými permutáciami vektoru ohodnotenia 2^n z 0, 1, aby čas môjho programu nebol skreslený jeho tvorením počas behu. Booleovská funkcia je zoradená abecedne, aby došlo, čo k najvyššej redukcií. Je zabezpečené, aby výraz bol vždy v úplnom normálovom tvare, neobsahuje viacnásobné premenné. A samozrejme je ošetrované, aby sa objavili plus, výkričník iba na správnom mieste.

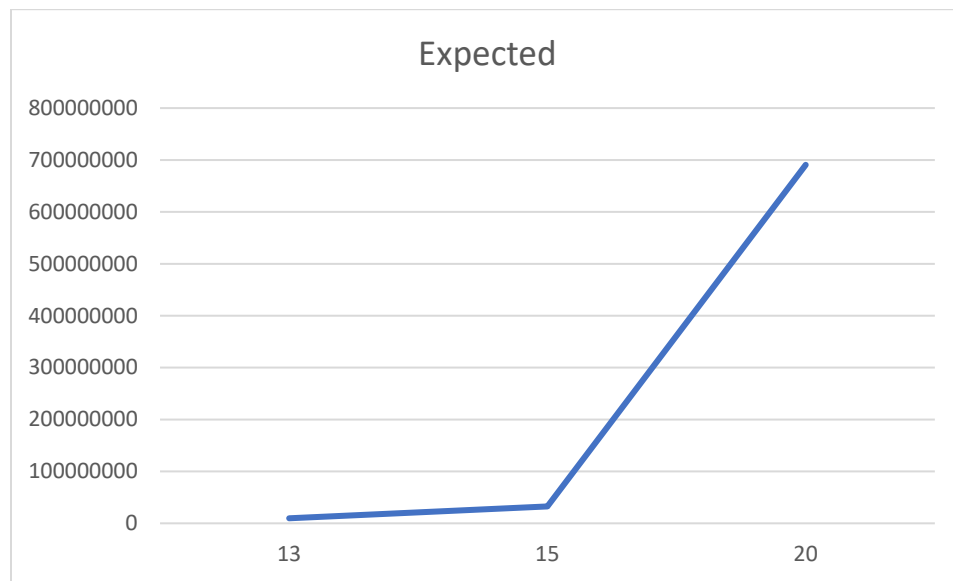
```
# Expressions
> def sort(output): ...
> def checkerino(list, variables): ...
> def expressionToString(list, variables): ...
> def checkUsage(output, variables): ...
> def polish(output, variables): ...
> def generate(variables): ...
> def expression(size, tests, file): ...

# Vector
> def vectorToString(s): ...
> def vector(size, file): ...
> def main(): ...
```

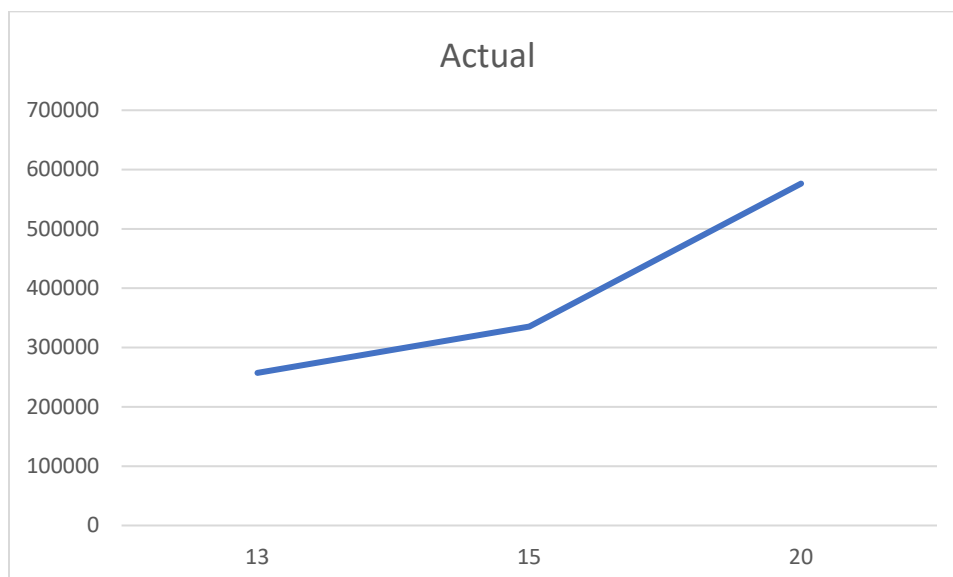
 13_expression.txt	4/23/2022 5:23 PM	Text Document	138 KB
 13_vector.txt	4/23/2022 5:23 PM	Text Document	120 KB
 15_expression.txt	4/23/2022 5:23 PM	Text Document	160 KB
 15_vector.txt	4/23/2022 5:23 PM	Text Document	544 KB
 20_expression.txt	4/23/2022 5:23 PM	Text Document	216 KB
 20_vector.txt	4/23/2022 5:23 PM	Text Document	22,528 KB
 generator.py	4/25/2022 8:38 PM	Python Source File	6 KB

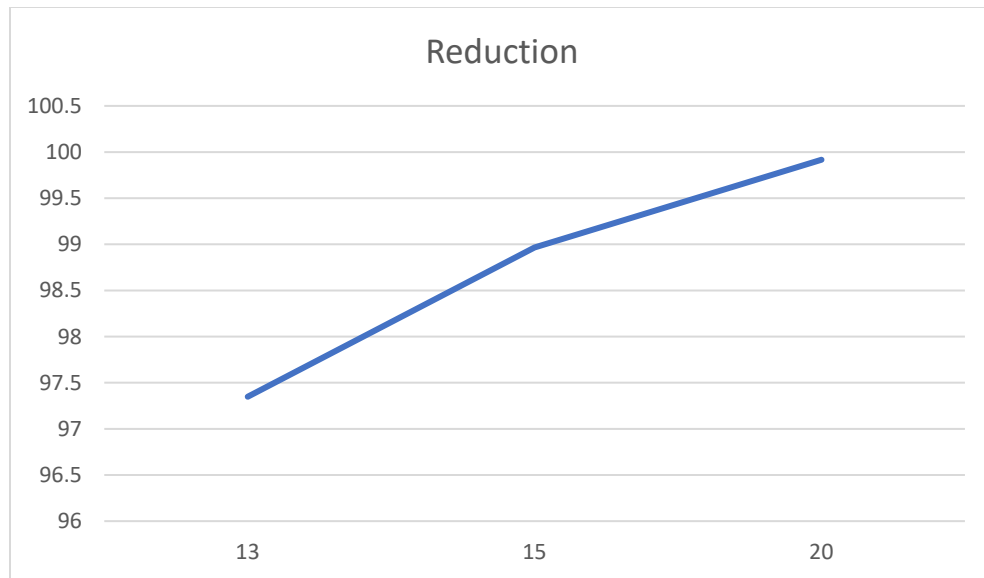
7. Grafy

Testovanie som opakoval tri krát a svoje výsledky som spriemeroval. Počas testovania som sledoval množstvo vytvorených nodeov pokiaľ by som nepoužíval redukciu, množstvo skutočných vytvorených nodeov. Na základe toho som vedel vypočítať mieru redukcie. Ďalej som meral čas za ktorý funkcie bddUse() a bddCreate() zbehli.



Os x reprezentuje množstvo premenných a os y počet nodeov.





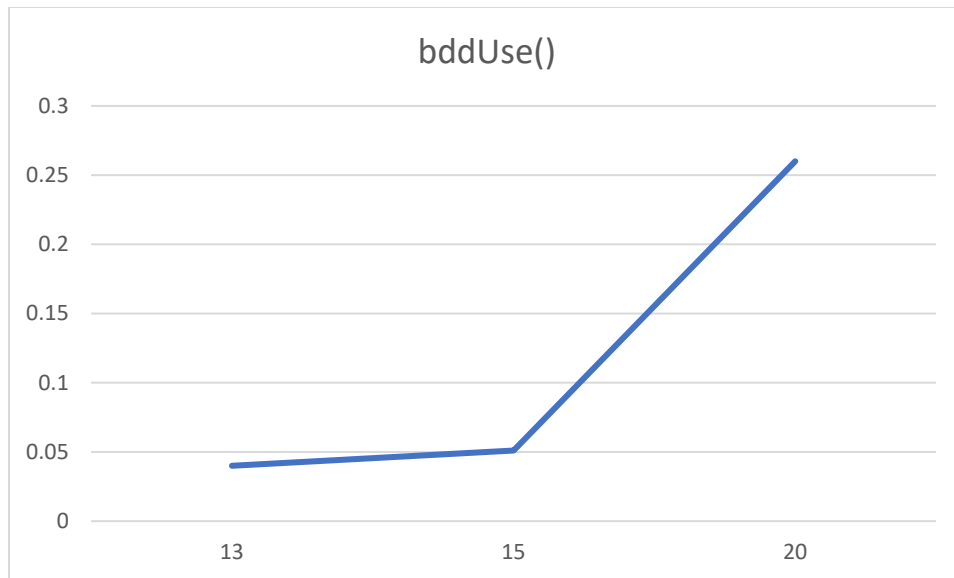
Os x reprezentuje množstvo premenných a os y percentuálnu mieru redukcie. Mieru redukcie som počítal takým spôsobom, že som vypočítal koľko percent predstavuje skutočná veľkosť z predpokladanej veľkosti pokiaľ by nenastalo k redukcii a odpočítal som výsledné číslo od sto.

Z výsledkov jasne vidno, že pamäťová náročnosť by mala rásť exponenciálne avšak v mojom prípade rastie pomalšie. Miera redukcie je pri menšom počte premenných nižšia, no pri vyššom dochádza k množstvu rovnakých výrazov a preto je o dosť vyššia.

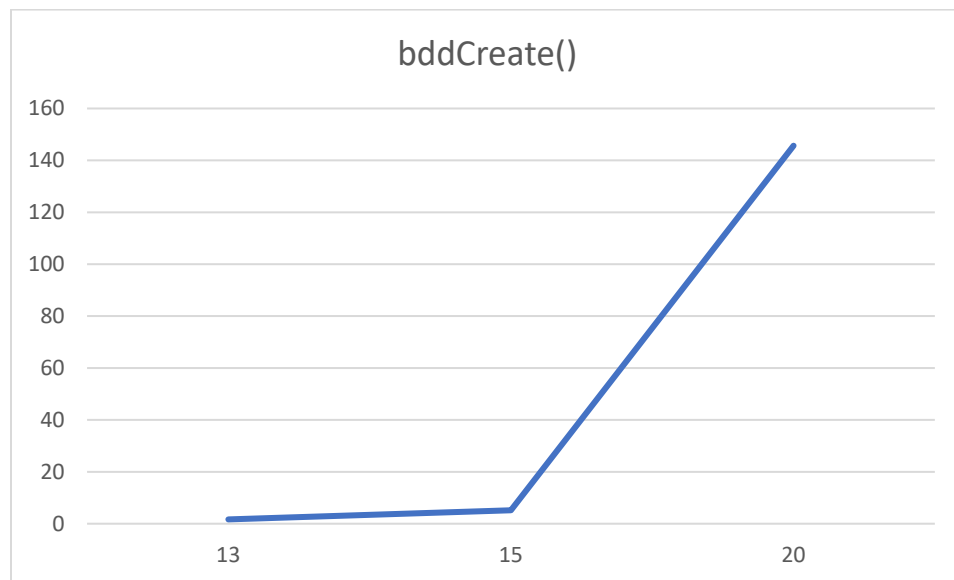
```
BEG;
tree = bddCreate(buffer, order);
END;
createTime += (double)(-start_t + end_t)/ CLOCKS_PER_SEC;
expected_size += tree->expected_size;
actual_size += tree->size;

BEG;
while (fgets(buffer, STRING, f_vector)) {
    use = bddUse(tree, buffer);
    if (use != check(tree->root->expression, order, buffer))
        printf("%s %s", tree->root->expression, buffer);
}
END;

useTime += (double)(-start_t + end_t)/ CLOCKS_PER_SEC;
```



Os x reprezentuje množstvo premenných os y rýchlosť v sekundách bddUse() je vďaka redukciám a vďaka tomu, že som bdd diagram písal v jazyku C veľmi rýchla, no napriek tomu závažne rastie, keďže množstvo ohodnotení výrazov pre každ sa stále zvyšuje.



7.1 Zložitosti hlavných funkcií

bddCreate()

Pamäťová zložitosť: Vďaka redukciám nie je $O(\sum 2^n)$, ale blíži sa skôr $O(n \log(n))$

Časová zložitosť: $O(n^3)$, rekúzia a v nej dva forcykly

bddUse()

Pamäťová zložitosť: okrem vstupných hodnôt, nemá žiadne pamäťové nároky

Časová zložitosť: Vďaka redukciám $O(h)$, kde h je výška diagramu pre danú cestu

8. Záver

V tomto dokumente som sa pokúsil predstaviť spôsob fungovania môjho binárneho rozhodovacieho stromu, ktorý vzniká z booleovskej funkcie. Vysvetlil som spôsob akým som ho skontroloval pomocou kontrolnej funkcie a programu valgrind. Ďalej obsahuje demo funkcie pri ktorých demonštrujem fungovanie môjho algoritmu aj pri náročnejších prípadoch a samotné testy vďaka, ktorých som nazbieral hodnoty, ktoré som mohol využiť pri vytvorení grafov spoločne s algoritmickou zložitosťou pre hlavné funkcie programu. Binárny rozhodovací diagram v kombinácii s redukciami je veľmi efektívna dátová štruktúra schopná podstatne zjednodušiť náročné problémy v informatike a moje nazbierané dáta to potvrdzujú.

9. Zdroje

https://cdn-media-1.freecodecamp.org/images/1*KfZYFUT2OKfjekJlCeYvuQ.jpeg

https://en.wikipedia.org/wiki/Binary_decision_diagram

<https://slideplayer.com/slide/8278496/>

<http://vtmathcoalition.org/math-beyond-horizon/topics/mbth-2021-05-08-bdd.pdf>