Dátové štruktúry a algoritmy

Dokumentácia k zadaniu

Richard Szarka

Slovenské technická univerzita v Bratislave

Fakulta informatiky a informačných technológií

xszarkar@stuba.sk

ID: 111958

Zadanie: 3

Cvičiaci: Mgr. Martin Sabo, PhD.

Meno: Richard Szarka
ID: 111958

Obsah

Meno: Richard Szarka
ID: 111958

1 Zadanie

V tomto zadaní máme vytvoriť program, ktorý bude vedieť vytvoriť, redukovať a použiť dátovú štruktúru BDD (Binárny Rozhodovací Diagram) so zameraním na využitie pre reprezentáciu Booleovských funkcií.

Konkrétne implementujte tieto funkcie:

- BDD *BDD_create(BF *bfunkcia);
- int BDD_reduce(BDD *bdd);
- char BDD_use(BDD *bdd, char *vstupy);

Funkcia BDD_create má slúžiť na zostavenie úplného (t.j. nie redukovaného) binárneho rozhodovacieho diagramu, ktorý má reprezentovať/opisovať zadanú Booleovskú funkciu (vlastná štruktúra s názvom BF), na ktorú ukazuje ukazovateľ bfunkcia, ktorý je zadaný ako argument funkcie BDD_create.

Funkcia BDD_reduce má slúžiť na redukciu existujúceho (zostaveného) binárneho rozhodovacieho diagramu. Aplikovaním tejto funkcie sa nesmie zmeniť Booleovská funkcia, ktorú BDD opisuje. Cieľom redukcie je iba zmenšiť BDD odstránením nepotrebných (redundantných) uzlov. Funkcia BDD_reduce dostane ako argument ukazovateľ na existujúci BDD (bdd), ktorý sa má redukovať. Redukcia BDD sa vykonáva priamo nad BDD, na ktorý ukazuje ukazovateľ bdd, a preto nie je potrebné vrátiť zredukovaný BDD návratovou hodnotou (na zredukovaný BDD bude totiž ukazovať pôvodný ukazovateľ bdd).

Funkcia BDD_use má slúžiť na použitie BDD pre zadanú (konkrétnu) kombináciu vstupných premenných Booleovskej funkcie a zistenie výsledku Booleovskej funkcie pre túto kombináciu vstupných premenných. V rámci tejto funkcie „sa prejde“ BDD stromom smerom od koreňa po list takou cestou, ktorú určuje práve zadaná kombinácia vstupných premenných.

Meno: Richard Szarka
ID: 111958

2 Riešenie

Binárny rozhodovací diagram je vhodný a užitočný na určenie výsledku boolovskej funkcie.

V mojej implementácii budem používať na reprezentáciu boolovskej funkcie (ďalej aj ako bf) vektor. To znamená, že bf bude reprezentovať reťazec zložený z 0 a 1 ktorého dĺžka je rovná dvojke umocnenej na počet premenných. Bf sa dá takto reprezentovať z toho dôvodu, že vektor reprezentuje stĺpec v pravdivostnej tabuľke (hlavička stĺpca je naša bf). Príklad pravdivostnej tabuľky a vektora funkcie AB+C.

A	B	C	AB+C
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Vektor tejto funkcie by bol v bf zapísaný ako „01010111“.

2.1 Makrá a knižnice

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

#define VARIABLES 13 //pocet premennych
#define DIAGRAM_NUM 100 //pocet diagramov
#define LEFT_CHILD_VECTOR(x) ((x)->left->Vector) //vektor laveho dietata
#define RIGHT_CHILD_VECTOR(x) ((x)->right->Vector) // vektor praveho
dietata
#define LEFT_CHILD(x) ((x)->left) //lave dieta
#define RIGHT_CHILD(x) ((x)->right) //prave dieta
#define PARENT(x) ((x)->parent) //rodic
```

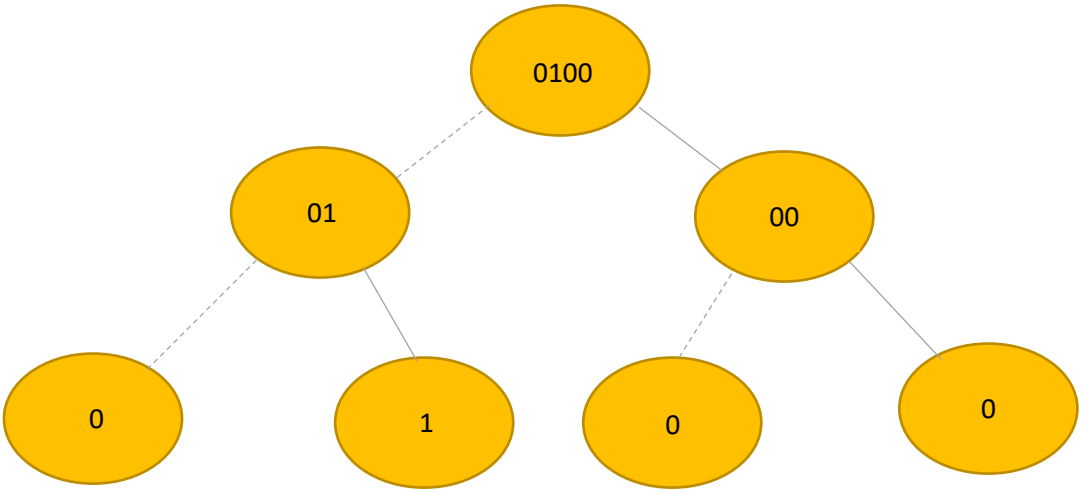
Meno: Richard Szarka
ID: 111958

2.2 Štruktúry

```
typedef struct Vertex{ //štruktúra na jeden uzol
    struct Vertex * left;
    struct Vertex * right;
    struct Vertex * parent;
    struct Vertex * previous;
    struct Vertex * next;
    char * Vector;

}VERTEX;

typedef struct BDD{ //štruktúra diagramu
    int numberOfVertices;
    int numberOfVariables;
    VERTEX * root;
}BDD;

typedef struct bool_func{ //boolovska funkcia
    char* vector;
}BF;
```

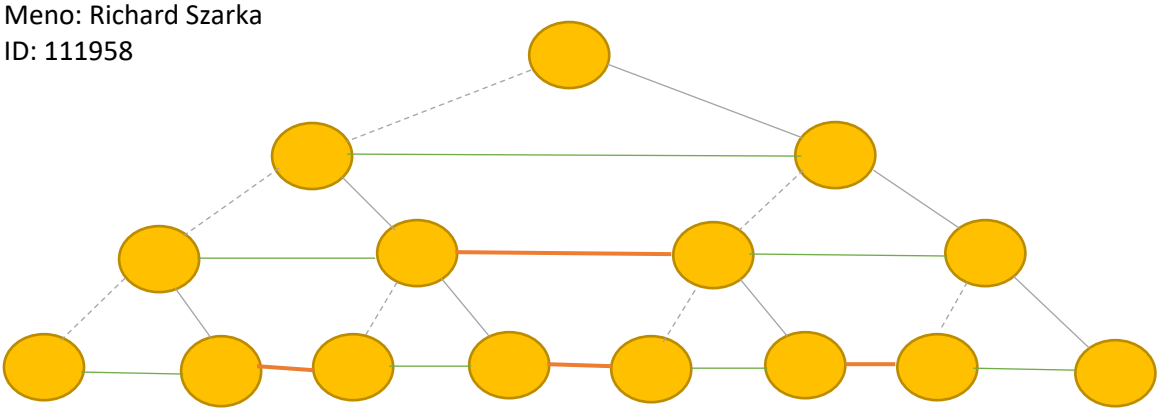
2.3 Funkcia BDD_create

Vo funkcii **BDD_create** sa má vytvoriť úplný nezredukovaný diagram z danej boolovskej funkcie. Vytvorenie takéhoto diagramu docielíme tak, že rekurzívne budeme pridávať nové uzly do nášho diagramu, vektor uzla na ktorom akurát pracuje funkcia sa rozdelí na 2 polovice (funkcie **split_begin a split_end**). 1. polovica ide do jeho ľavého potomka a 2. polovica ide do jeho pravého potomka. Funkcia sa zavolá na potomkoch, kým dĺžka vektora nie je 1. Vtedy sa vytvorili listy a funkcia môže skončiť rekurziu. Príklad vektora „0100“:



V štruktúre **BDD** sa uchová koreň, čiže k ostatným vrcholom sa dostaneme. Ďalšia vec, ktorá sa predpripraví vo funkcii **BDD_create** je prepojenie susedných vrcholov. Toto prepojenie budeme využívať na redukovanie redundantných vrcholov v redukcii (konkrétne pravidlo 2). Príklad:

Meno: Richard Szarka
ID: 111958



Vytváranie zelených „prepojení“ (ukazovatele) je pomerne triviálna záležitosť. Pri vytváraní ľavého a pravého dieťaťa sa navzájom prepoja. Červené prepojenia sa vytvoria tak, že ak zistí že neexistuje nasledovník z daného uzla a zároveň jeho rodič má nasledovníka, tak treba prepojiť dvoch susedov. Krajná vrcholy majú svojho nasledovníka a predchodcu NULL (podľa toho či je to ľavý alebo pravý okraj).

Funkcia v BDD_create, ktorá vytvorí ostatné uzly z koreňa:

```
void BDD_make(BDD * diagram, VERTEX * root){ //vytvaranie BDD
    char * left; //stringy na rozdelene vektory
    char * right;

    left=split_begin(root->Vector); //rozdelenie vektora
    right=split_end(root->Vector);
    root->left=create_vertex(left,root); //vytvorenie nových uzlov
    root->right=create_vertex(right,root);
    attach_siblings(root->left,root->right); //prepojenie uzlov

    diagram->numberOfVertices++;
    diagram->numberOfVertices++;
    if(strlen(left)>1 && strlen(right)>1){ //ak to neboli dlzky vektora 1
chod dalej
        BDD_make(diagram,root->left);
        BDD_make(diagram,root->right);
    }
    else
        return;

}
```

Funkcia na prepojenie susedov(červené hrany vyššie):

```
void BDD_attach_neighbours(BDD * bdd, VERTEX * root){ //prepojenie
surodencov na celej hlbke
    if(strlen(root->Vector)==1) //na poslednej hlbke neprepajaj (listy)
    {
        return;
    }
    if(RIGHT_CHILD(root)->next==NULL && root->next!=NULL){ //prepojenie
        RIGHT_CHILD(root)->next=root->next->left;
        root->next->left->previous= RIGHT_CHILD(root);
    }
    BDD_attach_neighbours(bdd,root->left); //rekurzia
    BDD_attach_neighbours(bdd,root->right);

}
```

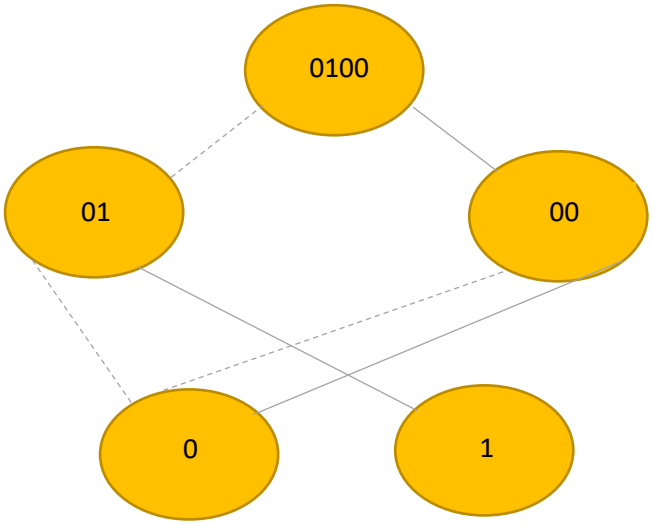
Meno: Richard Szarka
ID: 111958

2.4 Funkcia BDD_reduce

Funkcia **BDD_reduce** sa skladá z troch pravidiel. Prvé pravidlo je spúšťané len jeden krát, ostatné pravidlá sú spúšťané toľkokrát, koľko je počet premenných vo funkcii. Funkcia má vrátiť počet zredukovaných uzlov.

2.4.1 Pravidlo 1 (spojenie rovnakých listov)

Pravidlo jedna slúži na zbavenie sa redundantných vrcholov - listov (nepotrebných listov). Ak si to uvedomíme každý list je buď 1 alebo 0 a zároveň posledná hĺbka stromu je najväčšia počtom vrcholov. Tým pádom môžeme nahradiť všetky listy hodnoty 0 jedným listom 0 a všetky listy hodnoty 1 jedným listom 1. Ukážka:

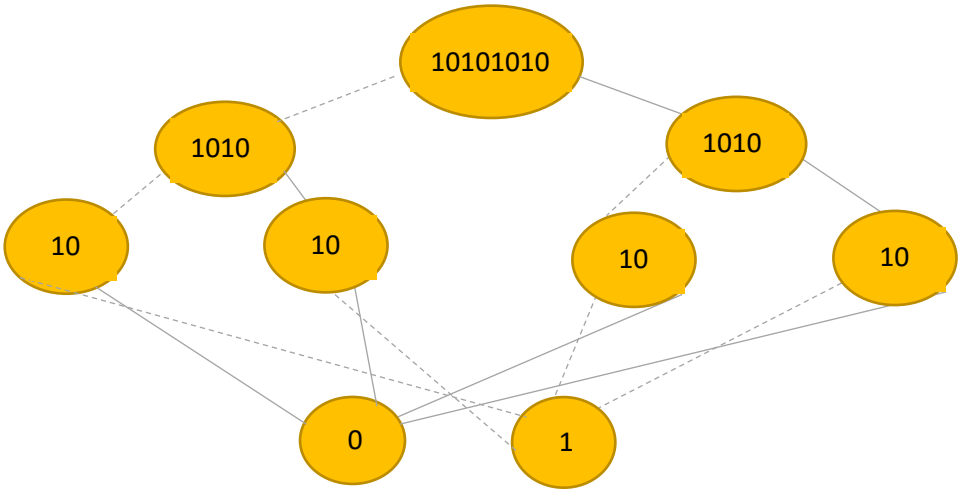


Týmto pravidlom sa zredukuje značne veľký počet vrcholov pri viac premenných. V programe sa vytvoria „konštantné“ vrcholy s názvom **Zero** a **One** (podľa hodnoty). Rekurzívne sa prejde každý vrchoľ BDD. Ak je dĺžka vektora 1, stačí zistiť či to je 0 alebo 1 (podľa toho sa nahradí vrcholom **Zero** alebo **One**) a od rodiča vrcholu treba zistiť či to je jeho ľavý alebo pravý potomok (treba nastaviť jeho ukazovateľ). V kóde redukciu pravidla 1 vykonáva funkcia rule1.

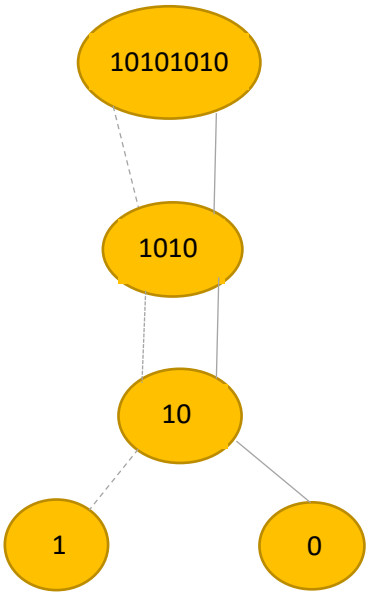
Meno: Richard Szarka
ID: 111958

2.4.2 Pravidlo 2 (spojenie izomorfných uzlov)

Pravidlo dva slúži na spojenie dvoch uzlov na rovnakej úrovni, ktoré sú rovnaké. Príklad BDD, z funkcie 10101010, ktorý prešiel 1. pravidlom.5



Môžeme vidieť, že na rovnakej úrovni (hĺbke) máme viacero uzlov ktoré sa správajú rovnako a môžu byť zjednotené. Je jedno na ktorý z uzlov sa pôjde ak sú rovnaké, právajú sa rovnako, preto nepotrebujeme viac rovnakých. Pravidlo dva teda prechádza od spodku BDD cez susedné vrcholy (ukazovatele/prechody na predchodcu a nasledovníka) a porovnávajú uzly. Ak sa nájdu dva zhodné uzly, tak sa zjednotia. Treba prestaviť aj ukazovateľa rodiča uzla, ktorý vymazávame (uvoľňujeme) z BDD. BDD po aplikovaní 2 pravidla:

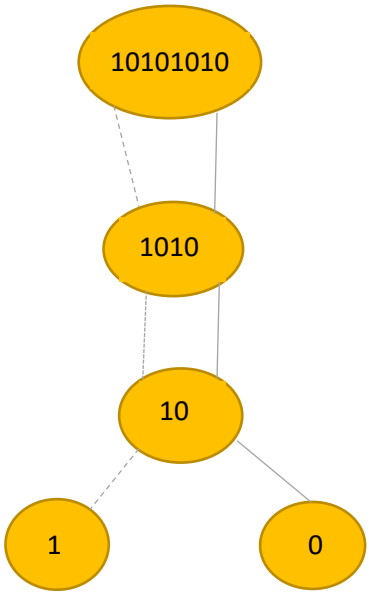


Ako môžeme vidieť, po aplikovaní druhého pravidla sa BDD zredukoval a zjednodušil výrazne. Zbavili sme sa nepotrebných (redundantných vrcholov). V kóde redukciu pravidla 2 vykonáva funkcia **rule2**.

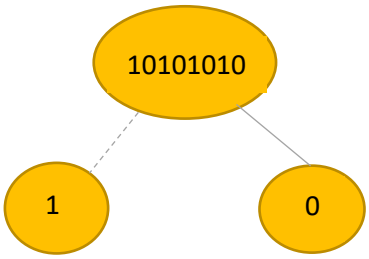
Meno: Richard Szarka
ID: 111958

2.4.3 Pravidlo 3 (Odstránenie redundantných uzlov)

Pravidlo 3 slúži na odstránenie uzlov, ktoré sú vyslovene nepotrebné v BDD. Pozrime sa na príklad funkcie 10101010 po aplikovaní 1. a 2. pravidla:



Je viditeľné, že 2. a 3. uzol (1010 a 10) sú nepotrebné v diagrame. Dôvod je taký, že z ich rodiča sa do nich dostaneme aj cez ukazovateľ na ľavého a pravého potomka. Tým pádom to môžeme brať ako keby je „rozhodnutie“ v binárnom rozhodovacom diagrame pri týchto uzloch určené (z ľavého aj pravého potomka sa dostaneme na ten istý uzol) a môže sa tento uzol vynechať. Diagram po aplikovaní pravidla 3:



Po redukcii pravidlom 3 sa diagram ešte zjednodušil nie len pamäťovo ale aj čas jeho používania bude viacnásobne kratší. Pravidlo 3 v kóde vykonáva funkcia **rule3**.

Meno: Richard Szarka
ID: 111958

2.5 BDD_use

Funkcia **BDD_use** má vedieť „použiť“ daný diagram a podľa zadaných hodnôt premenných a bf zistiť výsledok. Kód funkcie:

```
char BDD_use(BDD *bdd, char *vstupy){ //pouzitie diagramu
    VERTEX * root=bdd->root;
    for(int i=0;i<strlen(vstupy);i++){
        if(strlen(vstupy)-i==log2(strlen(root->Vector))) //moze sa stat ze
premenna ostane nepotrebna a preskoci uroven
        {
            if(vstupy[i]=='0')
            {
                root=root->left;
            }
            else
                root=root->right;
        }
    }
    return root->Vector[0];
}
```

Funkcia by bola triviálna bez redukcie. Stačí vlastne len podľa hodnôt premenných prejsť na ľavých alebo pravých potomkov. Po redukcia sa môže však stať, že na niektorých premenných pri konkrétnych prípadoch nezáleží, a tým pádom nám v redukcii „vypadnú“. Príklad funkcia A+BC. Ak sa zistí že A=1 tak funkcia bude vždy rovná 1 a na hodnote B a C nezáleží.

Túto skutočnosť vieme jednoducho ošetriť. Vieme, že ak sa kontroluje 1. premenná vektora je 2^{N-P} pričom N je počet premenných a P je index (poradie) premennej. Tým pádom vieme, že ak by sa kontrolovala 4. premenná v počte premenných 8, tak vektor na uzle musí mať veľkosť 2^{8-4} čo je 16. Ak dĺžka vektora sa nerovná, môže byť len menšia a tým pádom sa 4. premenná stala redundantnov v konkrétnom prípade. Na vrchole sa stojí a čaká sa na ďalšiu premennú.

Meno: Richard Szarka
ID: 111958

3 Testovanie

V testovaní využívam viacero funkcií:

```
char * binary_var(int number, int threshold) //vypisovanie binarnych cisel po nejaky index

char ** generate_var(int variables) //generovanie hodnot premennych

char * generate_random_vector(int variables){ //generovanie nahodných vektorov

int count_vertices(BDD * bdd) //funkcia na spocitanie uzlov po zredukovani

void test(BDD * bdd,BF * function) //tester
```

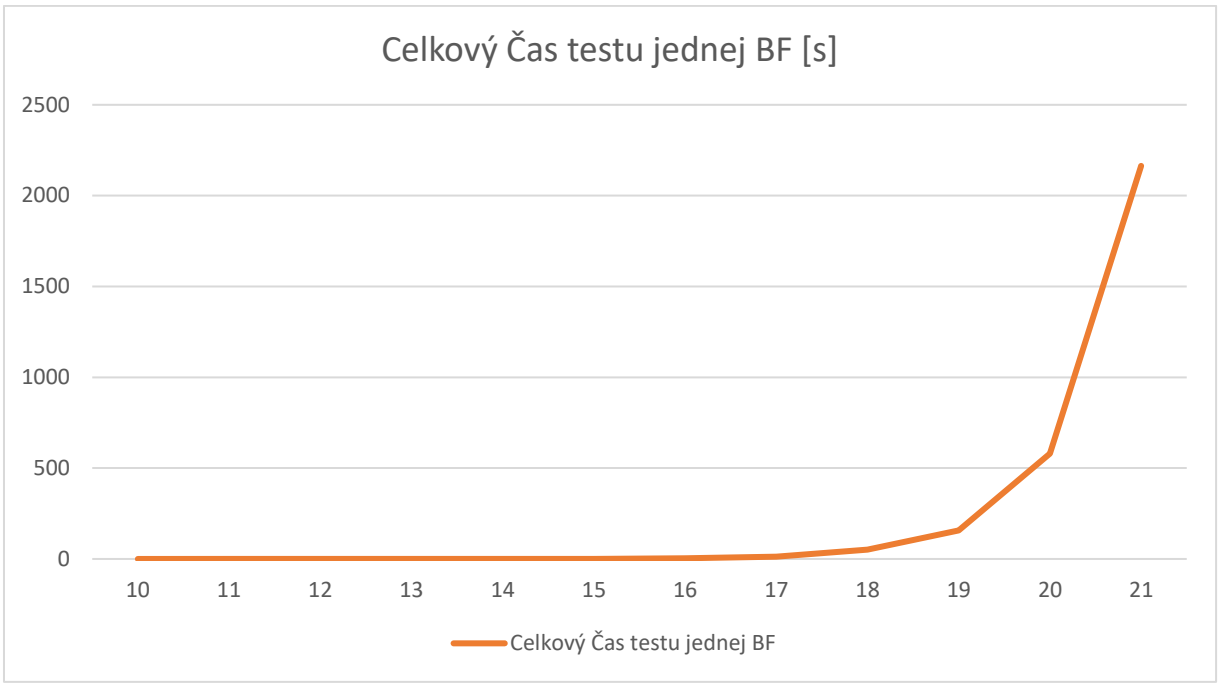
Funkcia generate_var s pomocou binar_var mi vygeneruje pole všetkých možných kombinácii vstupných premenných. Generuje sa tak, že sa vygeneruje 2^N binárnych čísel, začínajúc od 0, pričom N je počet premenných a zoberie prvých N cifier.

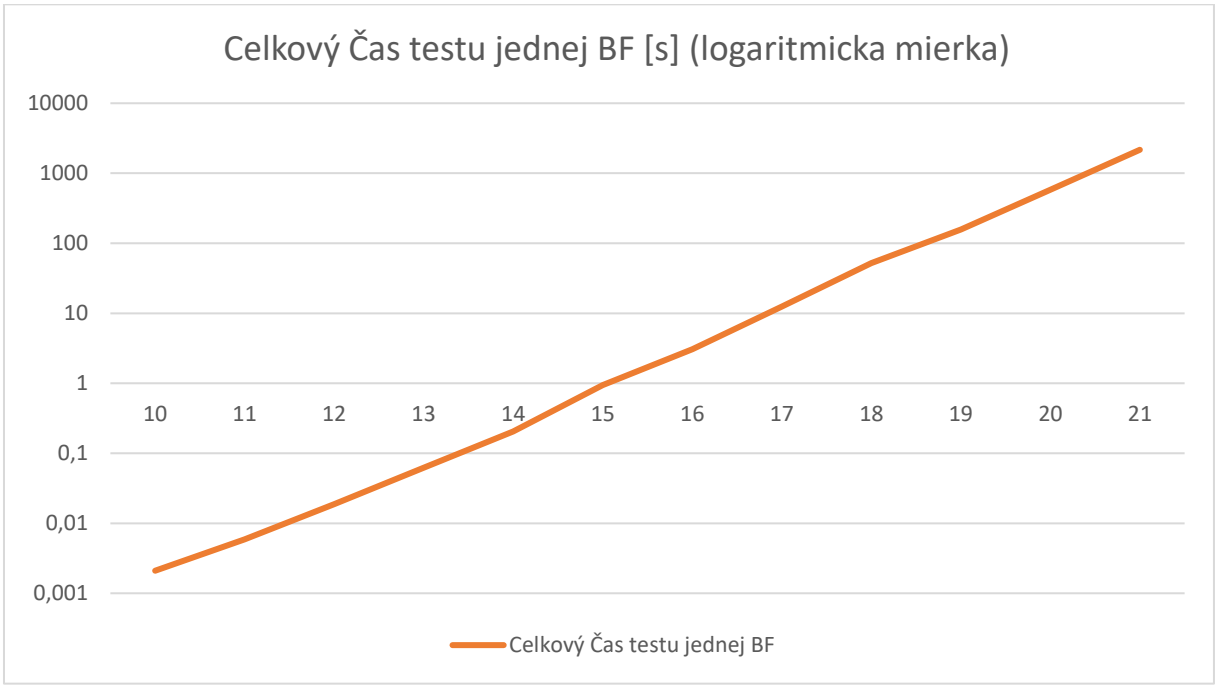
Generovanie náhodného vektora sa robí pomocou generovania náhodných čísel a zisťovania ich deliteľností 2. Ak číslo nie je deliteľné, zoberie sa char 0, ak je deliteľné zoberie sa char 1. Zoberie sa 2^N char-ov a funkcia vráti string vektora.

Funkcia count_vertices spočíta všetky uzly v strome aby sa zistilo ako veľmi sa strom zredukoval. Keďže, v pravidle 3 sa nedá každý uzol uvoľniť, z dôvodu, že pri pravidle 2 vznikajú pri uzloch viacnásobný rodičia. Tým pádom počítanie odstránených uzlov vo funkcii je s miernou odchylkou. Pre presnejšie výsledky testu, prejdem celý diagram a spočítam uzly.

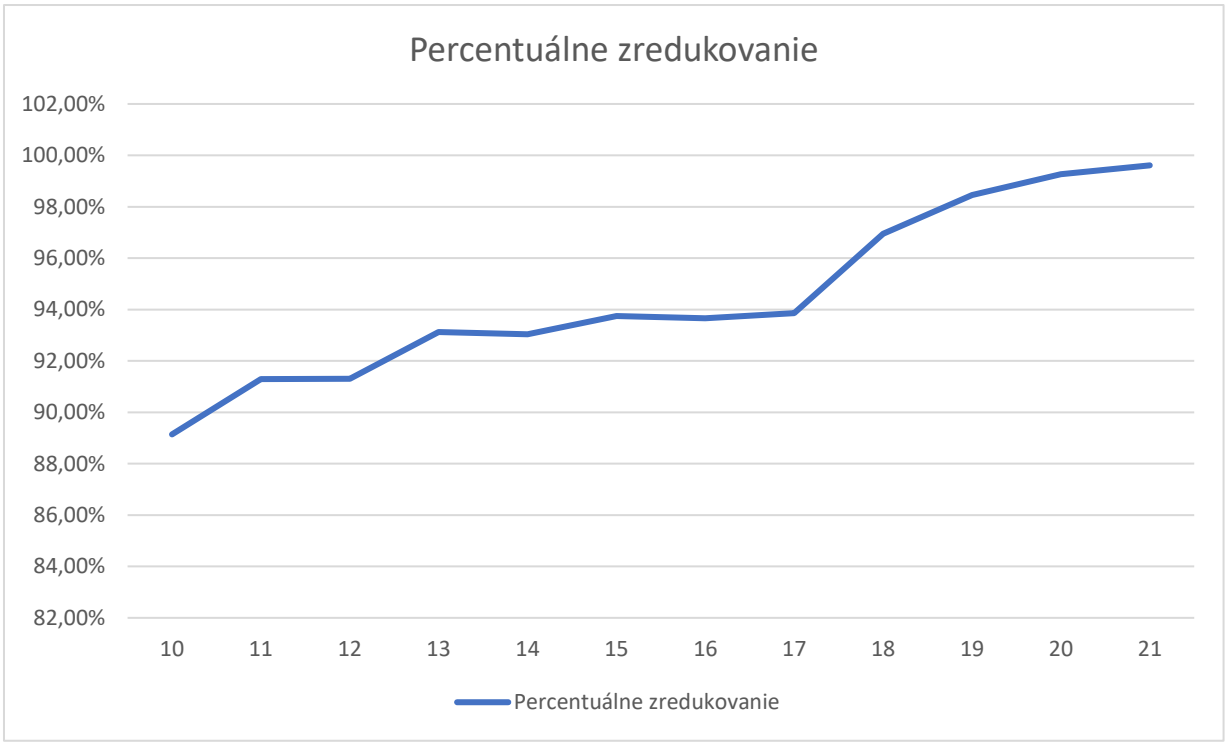
Test je samotné vykonanie testu. V testeri sa vytvorí diagram, zredukuje a následne cez BDD_use otestuje. Vo for cykle sa volá funkcia BDD_use ktorej argumentom je náhodne vygenerovaná boolovská funkcia a i-tý prvok poľa kombinácii premenných. Ak sa hodnota vrátená BDD_use rovná i-tému char-u v boolovskej funkcii, BDD bol správne zredukovaný aj použitý. Ináč sa vypíše správa o chybe. Funkcia zbiera hodnoty vytvorenia, zredukovania, použitia a mieru zredukovania diagramu. Následne vypočíta priemer z množstva otestovaných funkcií.

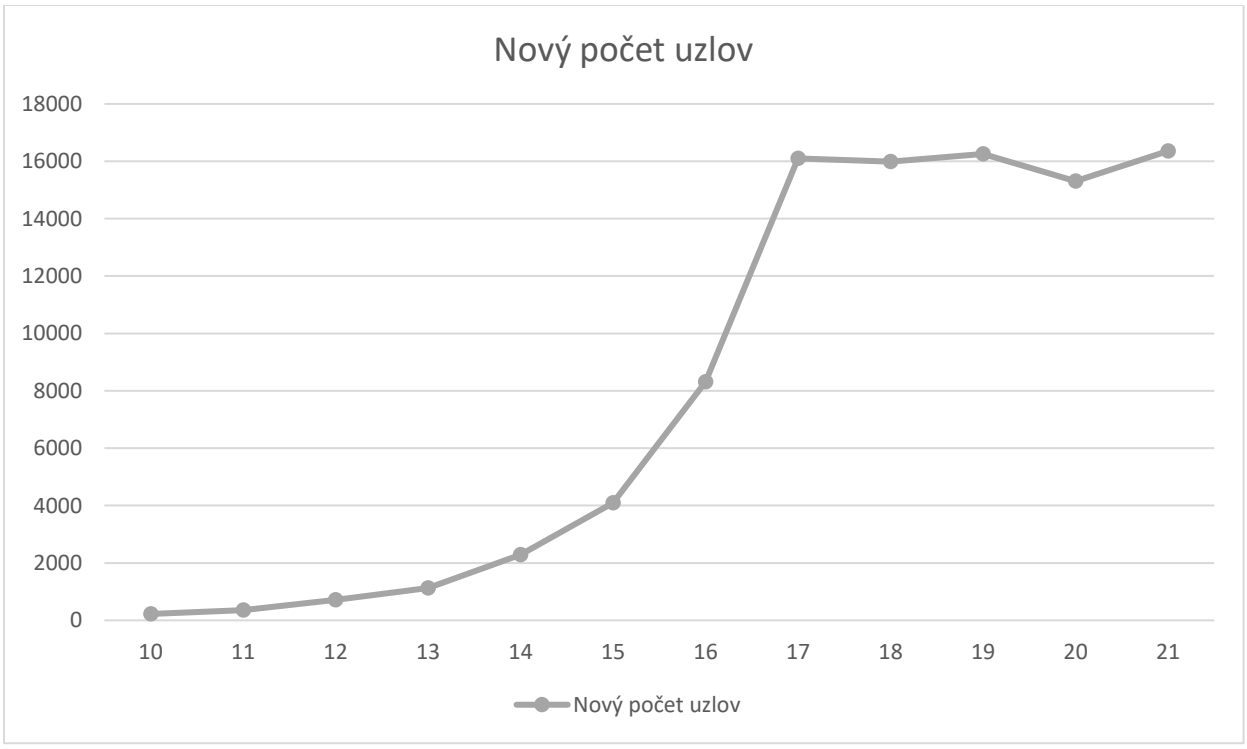
Meno: Richard Szarka
ID: 111958



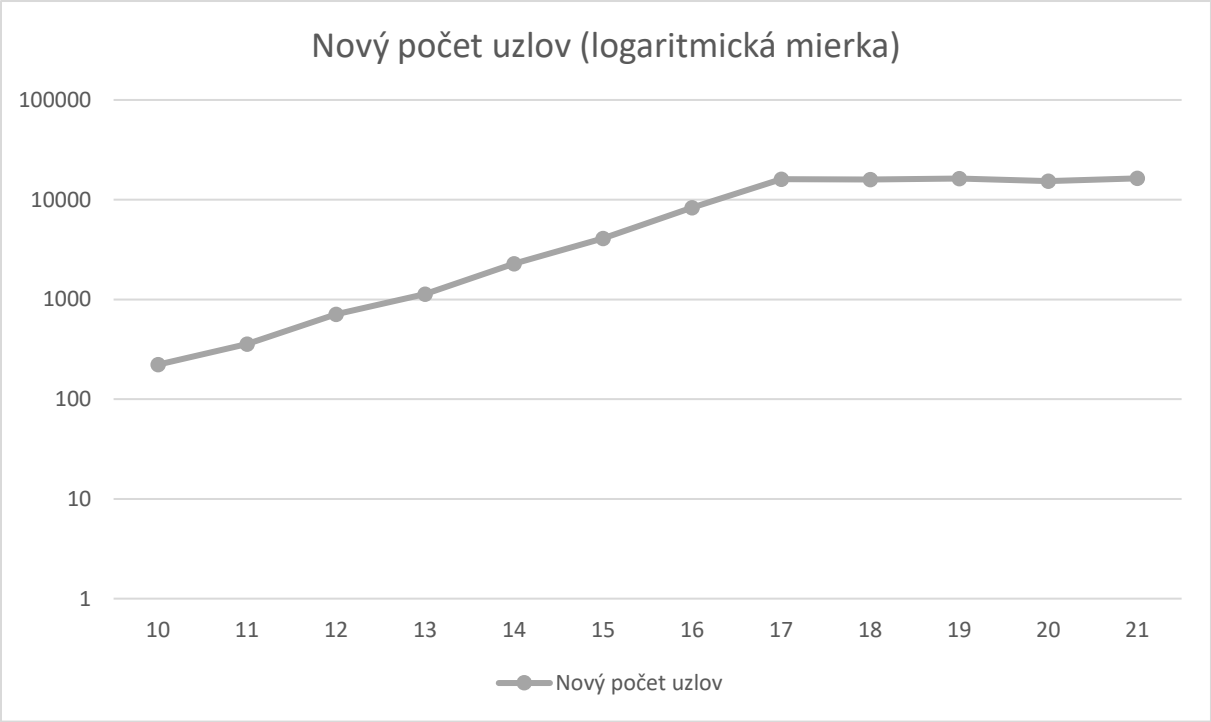


Meno: Richard Szarka
ID: 111958





Meno: Richard Szarka
ID: 111958



Tabuľka testov s výsledkami (časové údaje v [s]):

Priemerné výsledky testov								
Počet BDD	Počet Premenných	Čas vytvárania BDD	Čas redukcie BDD	Čas otestovania jednej BF	Celkový Čas testu jednej BF	Percentuálne zredukovanie	Nový počet uzlov	Pôvodný počet uzlov
2000	10	0,0007	0,0004	0,0009	0,0021	89,14%	222	2047
2000	11	0,0023	0,0012	0,0025	0,0059	91,29%	356	4095
2000	12	0,0073	0,0044	0,0071	0,0188	91,30%	712	8191
2000	13	0,0256	0,0145	0,0224	0,0625	93,12%	1127	16383
500	14	0,0963	0,0338	0,0751	0,2053	93,03%	2283	32767
100	15	0,3708	0,3001	0,2703	0,9412	93,75%	4095	65535
2	16	1,449	0,605	1,019	3,073	93,66%	8309	131071
2	17	5,732	2,755	3,946	12,433	93,86%	16095	262143
1	18	22,867	13,453	15,545	51,865	96,95%	15990	524287
1	19	91,362	12,203	54,033	157,598	98,45%	16252	1048575
1	20	365,164	26,651	187,554	579,369	99,27%	15309	2097151
1	21	1464,447	55,531	643,865	2163,843	99,61%	16357	4194303

Meno: Richard Szarka
ID: 111958

4 Záver

Binárne rozhodovacie diagramy (najmä tie zredukované) sú veľmi užitočné pre reprezentáciu, zjednodušovanie a vyhodnocovanie boolovských funkcií.

Pri testovaní je vidieť, že od množstva premenných 17 a viac sa počet uzlov v grafe pohyboval vždy okolo 16000. Práca s diagramom veľkosti približne 16000 je určite efektívnejšia ako práca s diagramom pôvodných veľkostí.

Zlepšenie mojej implementácie by bolo možné pridaním do štruktúry uzlov pole rodičov. Tým by sa vyriešilo uvoľňovanie uzlov pri treťom pravidle, a taktiež aj presnejšie počítanie uvoľnených uzlov. Ďalšie zlepšenie by mohlo nastať pri funkcii BDD_use, ktorá v sebe používa funkcii log2. Funkcia log2 je časovo a výpočtovo náročná a tým pádom je BDD_use pomalší ako keby sa BDD_use používal bez log2 funkcie.

Vytváranie grafu je pomerne časovo náročné, ale na druhú stranu redukcia diagramu ukazuje výrazne dobré časy. 4,2 milióna vrcholov zredukovala za menej ako 1 minútu na 16000. Pri testoch nebol výpis ani jedného erroru, čo značí o správnosti a korektnosti mojej implementácie.

Časové zložitosti:

BDD_create: $O(2^{N+1}-1)$ – pričom N je počet premenných (počet vrcholov krát 2)

BDD_reduce: $O: \sum_{x=2}^{N} \binom{2^{x-1}}{2}$ – pričom N je počet premenných

BDD_use: O(N) – pričom N je počet premenných

Pamäťové zložitosti:

BDD_create: $O(2^{N+1}-1)$ – pričom N je počet premenných

BDD_reduce – O(1)

BDD_use – O(1)