Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológii
3. marec 2021

Dátové štruktúry a algoritmy

Dokumentácia k zadaniu č. 3

Roland Vdovják
xvdovjak@stuba.sk
ID: 110912

Roland Vdovják
ID: 110912

Obsah

Roland Vdovják
ID: 110912

1 Zadanie

Zadanie 3 – Binárne rozhodovacie diagramy

Vytvorte 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*****);**

Samozrejme môžete implementovať aj ďalšie funkcie, ktoré Vám budú nejakým spôsobom pomáhať v implementácii vyššie spomenutých funkcii, nesmiete však použiť existujúce funkcie na prácu s binárnymi rozhodovacími diagramami.

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**. Štruktúru BF si definujete sami – podstatné je, aby nejakým (vami vymysleným/zvoleným spôsobom) bolo možné použiť štruktúru BF na opis Booleovskej funkcie. Napríklad, BF môže opisovať Booleovskú funkciu ako pravdivostnú tabuľku, vektor, alebo výraz. Návratovou hodnotou funkcie **BDD_create** je ukazovateľ na zostavený binárny rozhodovací diagram, ktorý je reprezentovaný vlastnou štruktúrou BDD. Štruktúra BDD musí obsahovať minimálne tieto zložky: počet premenných, veľkosť BDD (počet uzlov) a ukazovateľ na koreň (prvý uzol) BDD. Samozrejme potrebujete aj vlastnú štruktúru, ktorá bude reprezentovať jeden uzol BDD.

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*). Návratovou hodnotou funkcie **BDD_reduce** je číslo typu int (integer), ktoré vyjadruje počet odstránených uzlov. Ak je toto číslo záporné, vyjadruje nejakú chybu (napríklad ak BDD má ukazovateľ na koreň BDD rovný NULL). Samozrejme, funkcia **BDD_reduce** má aktualizovať aj informáciu o počte uzlov v 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 „prejdete“ BDD stromom smerom od koreňa po list takou cestou, ktorú určuje práve zadaná kombinácia vstupných premenných. Argumentami funkcie **BDD_use** sú ukazovateľ s názvom *bdd* ukazujúci na BDD (ktorý sa má použiť) a ukazovateľ s názvom *vstupy* ukazujúci na začiatok poľa charov (bajtov). Práve toto pole charov/bajtov reprezentuje nejakým (vami zvoleným) spôsobom konkrétnu kombináciu vstupných premenných Booleovskej funkcie. Napríklad, index poľa reprezentuje nejakú premennú a hodnota na tomto indexe reprezentuje hodnotu tejto premennej (t.j. pre premenné A, B, C a D, kedy A a C sú jednotky a B a D sú nuly, môže ísť napríklad o “1010“), môžete si však zvoliť iný spôsob. Návratovou hodnotou funkcie **BDD_use** je char, ktorý reprezentuje výsledok Booleovskej funkcie – je to buď ‘1’ alebo ‘0’. V prípade chyby je táto návratová hodnota záporná, podobne ako vo funkcii **BDD_reduce**.

Okrem implementácie samotných funkcií na prácu s BDD je potrebné vaše riešenie dôkladne otestovať. Vaše riešenie musí byť 100% korektné. V rámci testovanie je potrebné, aby ste náhodným spôsobom generovali Booleovské funkcie, podľa ktorých budete vytvárať BDD pomocou funkcie BDD_create. Vytvorené BDD následne zredukujete funkciou BDD_reduce a nakoniec overíte 100% funkčnosť zredukovaných BDD opakovaným (iteratívnym) volaním funkcie BDD_use tak, že použijete postupne všetky možné kombinácie vstupných premenných. Počet premenných v rámci testovania BDD by mal byť minimálne 13. Počet Booleovských funkcií / BDD diagramov by mal byť minimálne 2000. V rámci testovania tiež vyhodnocujte percentuálnu mieru zredukovania BDD (t.j. počet odstránených uzlov / pôvodný počet uzlov).

Okrem implementácie vášho riešenia a jeho testovania vypracujte aj dokumentáciu, v ktorej opíšete vaše riešenie, jednotlivé funkcie, vlastné štruktúry, spôsob testovania a výsledky testovania, ktoré by mali obsahovať (priemernú) percentuálnu mieru zredukovania BDD a (priemerný) čas vykonania vašich funkcií. Dokumentácia musí obsahovať hlavičku (kto, aké zadanie odovzdáva), stručný opis použitého algoritmu s názornými nákresmi/obrázkami a krátkymi ukážkami zdrojového kódu, vyberajte len kód, na ktorý chcete extra upozorniť. Pri opise sa snažte dbať osobitý dôraz na zdôvodnenie správnosti vášho riešenia – teda dôvody prečo je dobré/správne, spôsob a vyhodnotenie testovania riešenia. Nakoniec musí technická dokumentácia obsahovať odhad výpočtovej (časovej) a priestorovej (pamäťovej) zložitosti vášho algoritmu. Celkovo musí byť cvičiacemu jasné, že viete čo ste spravili, že viete odôvodniť, že to je správne riešenie, a viete aké je to efektívne.

Riešenie zadania sa odovzdáva do miesta odovzdania v AIS do stanoveného termínu (oneskorené odovzdanie je prípustné len vo vážnych prípadoch, ako napr. choroba, o možnosti odovzdať zadanie oneskorene rozhodne cvičiaci, príp. aj o bodovej penalizácii). Odovzdáva sa jeden **zip** archív, ktorý obsahuje zdrojové súbory s implementáciou riešenia a testovaním + jeden súbor s dokumentáciou vo formáte **pdf**. **Vyžaduje sa tiež odovzdanie programu**, ktorý slúži na testovanie a odmeranie efektívnosti týchto implementácií ako jedného samostatného zdrojového súboru (obsahuje funkciu **main**).

Hodnotenie

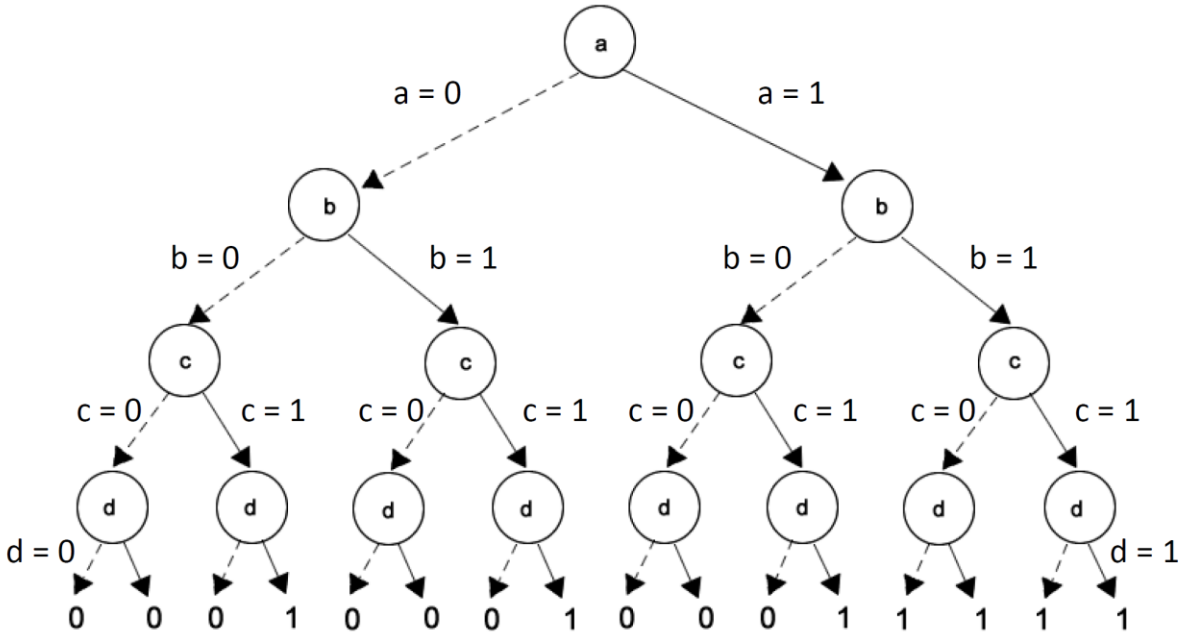
Môžete získať celkovo 15 bodov, **nutné minimum je 6 bodov**.

Za implementáciu riešenia (3 funkcie) je možné získať celkovo 8 bodov, z toho 2 body sú za funkciu **BDD_create**, 4 body za funkciu **BDD_reduce** a 2 body za funkciu **BDD_use**. Pre úspešné odovzdanie implementácie musíte zrealizovať aspoň funkcie **BDD_create** a **BDD_use**. Za testovanie je možné získať 4 body (treba podrobne uviesť aj v dokumentácii) a za dokumentáciu 3 body (bez funkčnej implementácie 0 bodov). Body sú ovplyvnené aj prezentáciou cvičiacemu (napr. keď neviete reagovať na otázky vzniká podozrenie, že to **nie je vaša práca, a teda je hodnotená 0 bodov)**.

2 Teoretické východiská

Binárny rozhodovací diagram[1]

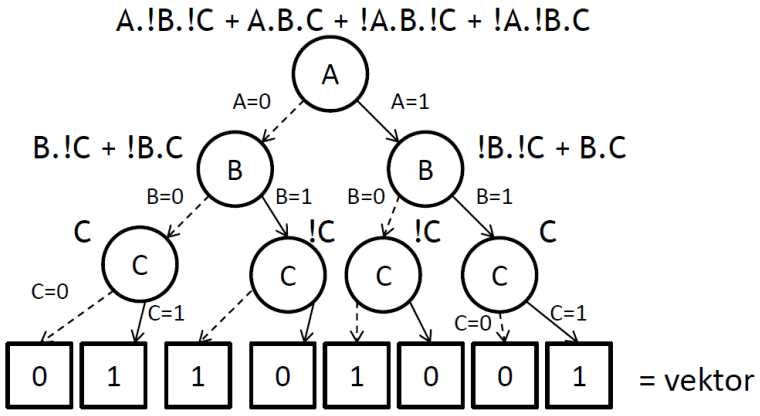
Binárny rozhodovací diagram je binárny strom, ktorý slúži na rozhodovanie. Rozhodovanie prebieha prechodom od začiatku stromu do konca (smerom z koreňu do listu). Každý uzol predstavuje jedno čiastkové rozhodnutie. V našom zadaní sme používali BDD na reprezentáciu Booleovskej funkcie, viď obrázok.



BDD z obrázka sa skladá zo štyroch premenných (a, b, c, d). Vektorová reprezentácia danej Booleovskej funkcie reťazec listov zľava doprava (0001 0001 0001 1111).

Zostrojenie nového BDD – zhora nadol

Pomocou Shannonovej dekompozície vytvárame najskôr koreň. Potom jednu premennú, vytvoríme čiastkové podiely a vložíme do potomkov. Opakujeme pre každú premennú.



[1] Ďalej „BDD“

3 Riešenie

3.1 BDD create

Táto funkcia má za úlohu vytvoriť BDD podľa vstupnej binárnej funkcie. V našom prípade funkcia alokuje pamäť o veľkosti pamäti potrebnej pre našu BDD štruktúru. Priradí počiatočné hodnoty. Veľkosť diagramu, resp. počet vrcholov vypočíta podľa vzorca:

$$2^{N+1} - 1$$

Kde *N* je makro, ktorého hodnotu si programátor zvolí pred spustením programu. Predstavuje to počet vstupných premenných do funkcie BDD. Hodnota tejto premennej sa zároveň priradí do štruktúry na miesto počtu premenných *„var_num“* . Koreň je potrebné nastaviť na počiatok diagramu a zároveň je potrebné korektne vytvoriť diagram.

Ukážka štruktúr:

```
    // Node
typedef struct node {
    char* val;      // Hodnota
    struct node* n_left, * n_right;     // Potomkovia
} NODE;

    // Binary Decision Diagram
typedef struct bdd {
    int var_num;        // Pocet premennych
    int size;       // Velkost BDD, resp pocet vrcholov
    NODE* root;         // Smernik na koren
} BDD;

    // Vektor
typedef struct bf {
    char* vector;
} BF;
```

Ukážka kódu *BDD_create*:

```
if (vLen == pow(2, N)) {

        diagram = (BDD*) malloc(sizeof(BDD));      // Alokacia
            // Pociatocne priradenie hodnot
        diagram->var_num = N;
        diagram->root = createDiagram(bfunkcia);
        diagram->size = pow(2, N + 1) - 1;

    }
    else {
        printf("ERROR - vector is not 2^N");
        return;
    }
```

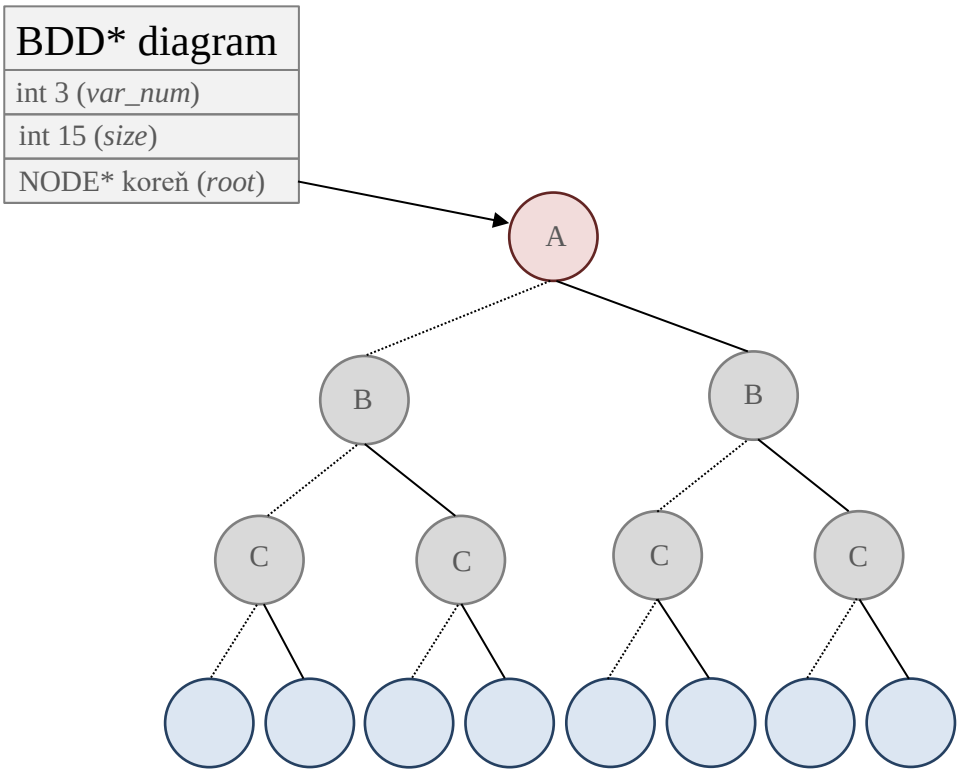
V našom riešení vytvárame diagram metódou zhora nadol, kód sa nachádza vo vlastnej pomocnej funkcii *„createDiagram“*. Táto funkcia funguje na rekurzívnom princípe, vytvára strom nasledovne. Ak vrchol nie je listom, tak sa porovnáva ľavý potomok s *NULL*, ak sa nenachádza, vytvorí sa. Ak sa nachádza, opýta sa na pravý potomok a postupuje analogicky. Hodnota sa vrcholom priradí tak, že pomocná funkcia *„subString“* vráti hľadaný podreťazec hodnoty vrchola, ktorému sa

vytvárajú potomkovia. Ľavý potomok dostane hodnotu rovnú prvej polovici hodnoty rodiča, pravý rovný druhej polovici hodnoty rodiča.

Ukážka kódu *createDiagram*:

```
NODE* root = malloc(sizeof(NODE));      // Alokovanie noveho vrcholu
    root->val = bfunkcia;       // Priradovanie hodnoty vrcholu
    root->n_left = root->n_right = NULL;    // Prednastavenie childov na NULL

    int vLen = strlen(bfunkcia);        // Dlzka aktualnej bFunkcie

    if (vLen != 1) {    //Opakuje sa pokym nie sme v listoch
        if (root->n_left == NULL) {
            root->n_left = createDiagram(subString(bfunkcia, 0, vLen / 2));
        }

        if (root->n_right == NULL) {
            root->n_right = createDiagram(subString(bfunkcia, vLen / 2, vLen));
        }
    }
       return root;
```

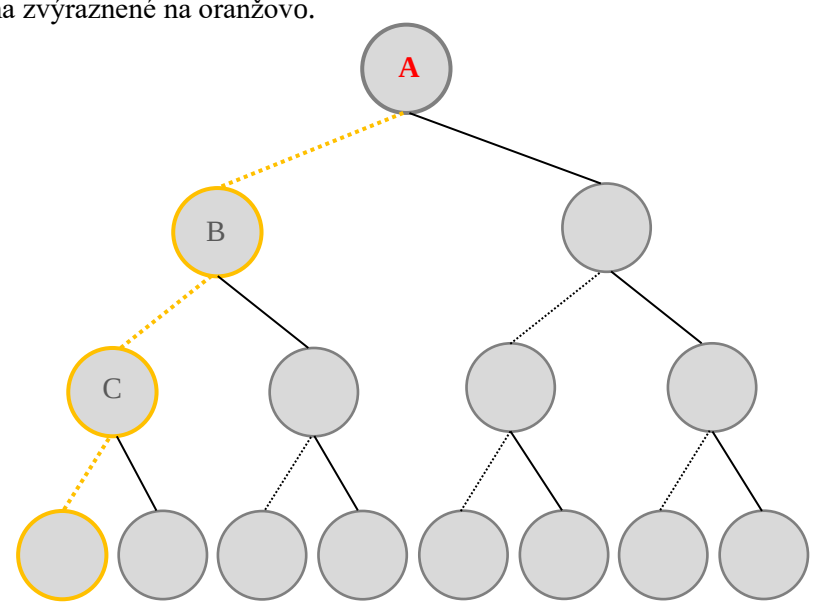
Vizuálne sa dá táto časť programu reprezentovať nasledovne:



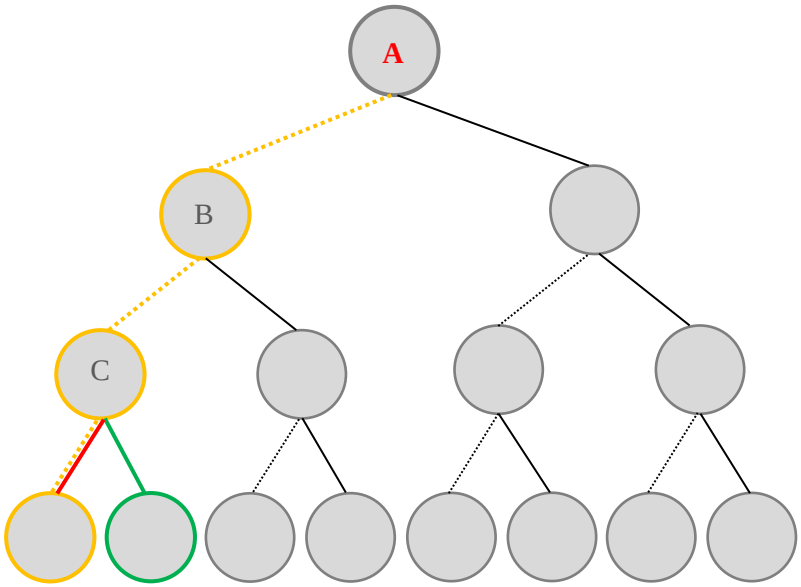
Modré vrcholy reprezentujú listy, je v nich hodnota o veľkosti jedna. Daný príklad je zostrojený pre tri premenné, z toho vyplýva, že má 15 vrcholov a dĺžka vektora je 8.

Roland Vdovják
ID: 110912

Vytváranie stromu sa dá reprezentovať nasledovne, na začiatku neexistuje vrchol, tak ho vytvorí, náš vrchol *"A"*, program zistí, že vrchol nemá ľavého potomka, tak ho vytvorí, takto postupne vytvára vrcholy na zvýraznené na oranžovo.



Potom sa program, z listu vráti do prechádzajúceho vrchola, lebo nie je splnená podmienka, že nový vrchol sa vytvára len na vyššej úrovni. Táto cesta je reprezentovaná červenou Program pokračuje a pýta sa na pravý potomok. Cesta a vrchol sú reprezentované zelenou farbou. Potom sa program vráti znovu do vrcholu *„C"* a analogicky vytvorí celý strom.



3.2 BDD_reduce

Funkcia má za úlohu redukovať BDD na čo najmenej vrcholov, čo najefektívnejšie a zároveň nezmeniť vstupnú BF.

Naša forma redukovania sa skladá zo samotnej funkcia *„BDD_reduce“* a dvoch hlavných pomocných funkcií *„reduce1“* a *„reduce2“*. Naše riešenie je implementované takto z dôvodu využívania rekurzívnych princípov. Okrem týchto pomocných funkcií sme implementovali aj pomocné funkcie *„BDD_fillArray“* a *„Child_Diagram_Free“*.

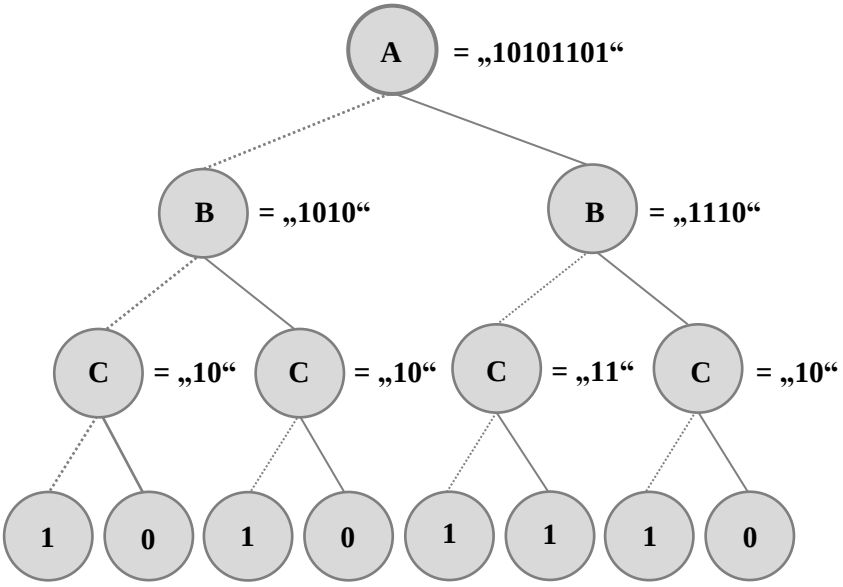
3.2.1 Reduce1

Táto pomocná funkcia sa využíva ako prvá, za úlohu má odstrániť duplicitné vrcholy. Pomocná funkcia *„BDD_fillArray“* prehľadá BDD na princípe prehľadávania inorder, každý vrchol porovná s už uloženými vrcholmi v poli, ak sa vrchol nenachádza v poli, priradí sa na koniec daného poľa. Ak sa nachádza, program pokračuje v prehľadávaní na ďalší vrchol, ktorý porovnáva.

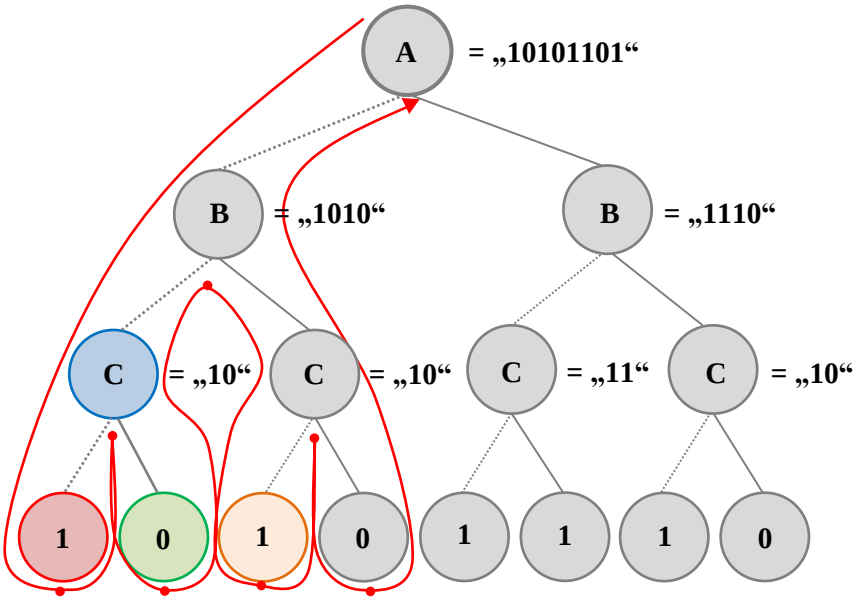
Keď je pole naplnené, spustí sa rekurzívna funkcia *„reduce1“*. Na princípe prehľadávanie preorder prehľadáva BDD. Ak sa vrchol, v ktorom sa nachádza nerovná listu, porovná ľavého a pravého potomka so všetkými unikátnymi vrcholmi v poli. Ak sa hodnota daného vrcholu nachádza v poli a zároveň vrchol nie je identický, zmenia sa smerníky. Nasmerujú sa na vrchol z poľa a program uvoľní celý podstrom, ktorý už nie je využívaný.

Graficky sa tento algoritmu dá predstaviť nasledovne.

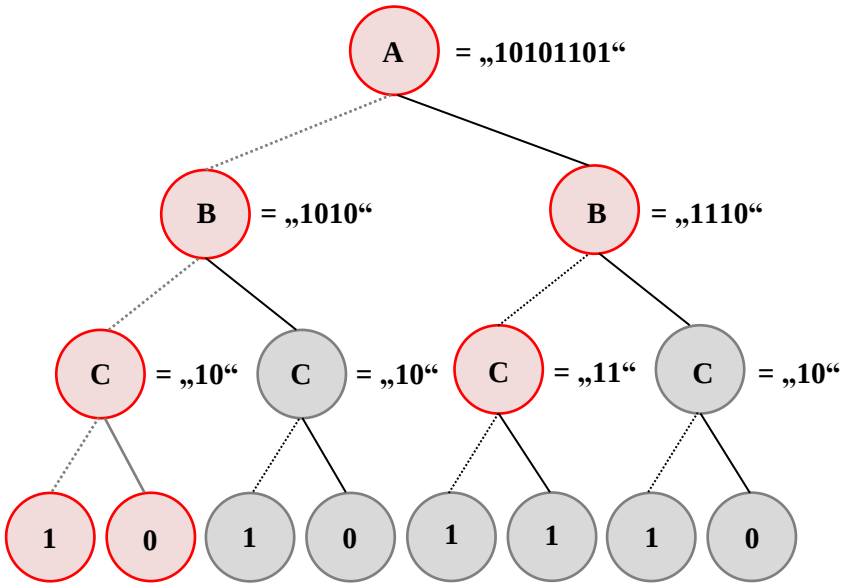
Vstup pre algoritmus:



Algoritmus na naplnenie poľa unikátnych vrcholov. Program ide po červenej krivke, prvý krát sa zastaví vo vrchole s červeným označením, potom modrý, neskôr zelený. Tieto vrcholy sa v poli nenachádzajú, tak ich do neho priradí, prvýkrát kolízia nachádza v oranžovom vrchole, ktorý nepriradí.



Všetky vrcholy, ktoré sa nachádzajú v poli sú zvýraznené na červeno.



Ukážka kódu vytvorenia poľa:

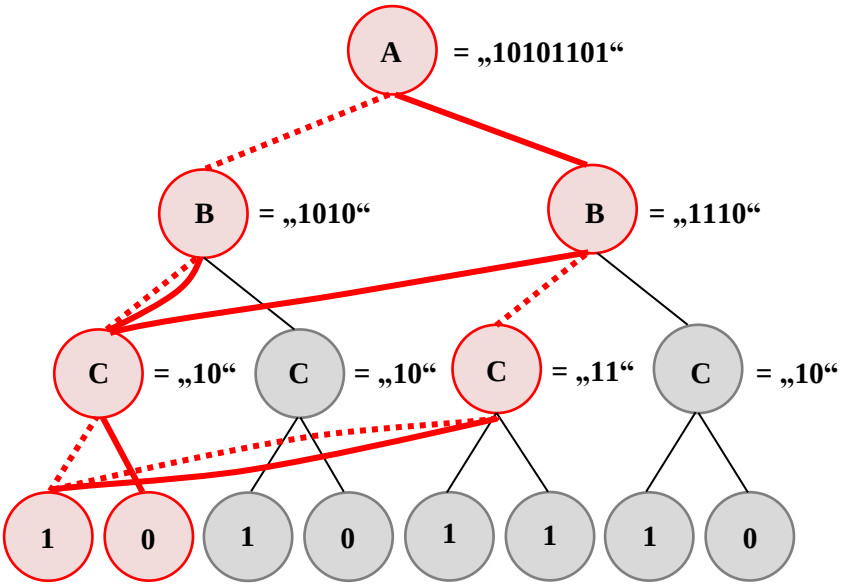
```
if (root->n_left != NULL) {
    BDD_fillArray(counter, root->n_left, array);
}

    // Porovnava, ci sa hodnoty zhoduju
for (int i = 0; i < *counter; i++) {
    if (strcmp(array[i]->val, root->val) == 0) return;
}
array[*counter] = root;// Priradi na podlednu poziciu v liste dany unikatny vrchol
(*counter)++;
```

```
    if (root->n_right != NULL) {
        BDD_fillArray(counter, root->n_right, array);
    }
```

Ukážka kódu prepájania smerníkov:

```
if (root->n_left != NULL) {
        for (int i = 0; i < *counter; i++) {

            if ((strcmp(array[i]->val, lCh->val) == 0) && (lCh != array[i])) {
                Child_Diagram_Free(root->n_left);
                root->n_left = array[i];
                lCh = array[i];
            }

            if ((strcmp(array[i]->val, rCh->val) == 0) && (rCh != array[i])) {
                Child_Diagram_Free(root->n_right);
                root->n_right = array[i];
                rCh = array[i];
            }
        }
    }
```

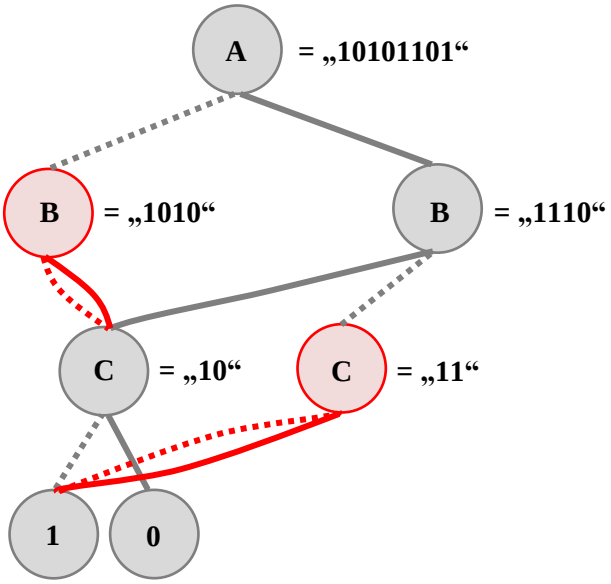
Grafiky to môžeme reprezentovať nasledovne, všetko sivé sa uvoľní a zostanú len červené vrcholy.



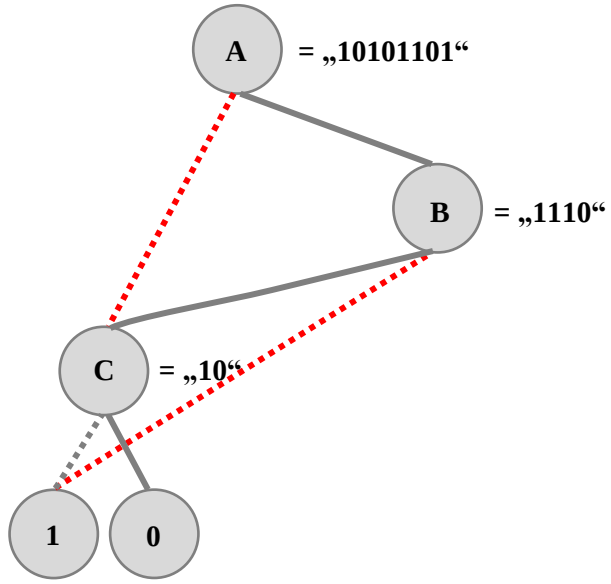
3.2.2 Reduce2

Druhá časť redukcie sa skladá z „preskakovania“ vrcholov, ktoré majú obidvoch potomkov nasmerovaných na totožný vrchol.

Jedná sa o červeno zvýraznené prepojenia a vrcholy.



Po použití *„reduce2“* vyzerá diagram nasledovne. Červené vrcholy z prechádzajúceho diagramu sa odstránia a cesty sa prepoja.



Takto vyzerá diagram po použití *„BDD_reduce“*, v tomto konkrétnom príklade sa počet vrcholov zmenil z 15 na 5, čo je 33,33% pôvodnej veľkosti.

Ukážka kódu:

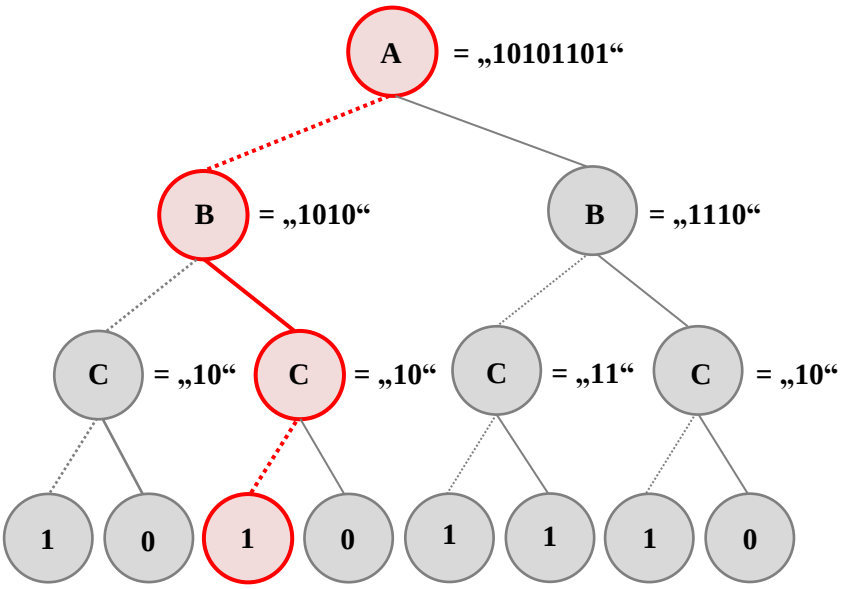
```
if (lCh->n_left != NULL && rCh->n_right != NULL) {
    if (lCh->n_left == lCh->n_right) {
        root->n_left = lCh->n_left;
        test[*counter] = lCh;
        lCh = root->n_left;
        (*counter)++;
    }

    if (rCh->n_left == rCh->n_right) {
        root->n_right = rCh->n_right;
        test[*counter] = rCh;
        rCh = root->n_right;
        (*counter)++;
    }

}
```

3.3 BDD_use

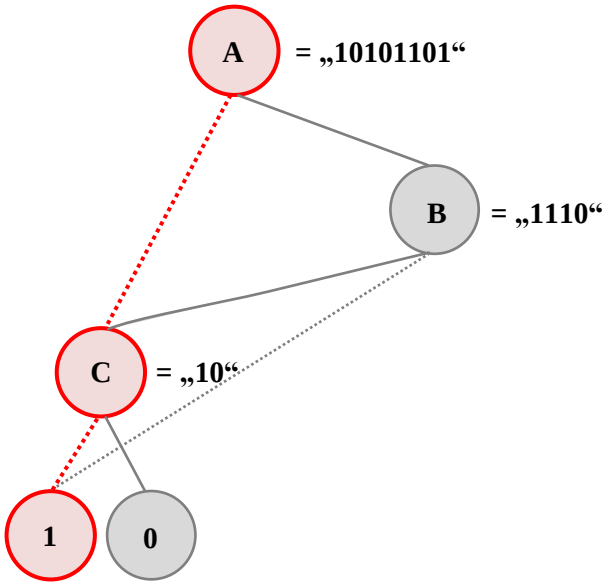
Funkcia má za úlohu vypísať hodnotu listu alebo „-1“ v závislosti od vstupnej postupnosti premenných Booleovskej funkcie.

Princíp algoritmu nášho riešenia tejto funkcie je nasledovať do pravého alebo ľavého potomka v závislosti, či má premenná z danej postupnosti premenných hodnotu „1“ alebo „0“. Ak je hodnota rovná „1“, nasleduje doprava, ak sa rovná „0“, nasleduje doľava.

Výstup pre volanie funkcie *BDD_use(diagram, „010“)*:



Avšak, túto funkciu voláme až po redukovaní, takže nie je vždy nutné, aby funkcia prešla práve počtom vrcholov o počte premenných. Je nutné porovnávať i-tu premennú s i-tou výškou diagramu. Zredukovaný diagram prichádza o redundantný vrchol „B“, ľavý podkoreň vrcholu „A“ smeruje priamo do vrcholu „C“.



Ukážka kódu:

```
    while (i < strlen(vstupy) && !found) {

    if (vstupy[height] == '1') {
        temp = temp->n_right;
    }
    else {
        temp = temp->n_left;
    }

    if (temp->n_left == NULL) found = true;     // Nasiel
    i++;
    height++;
}

val = (strcmp(temp->val, "1")) ? 0 : 1;     // Priradenie koncovej honoty

return val;
```

4 Testovanie

Na testovanie sme využili niekoľko cyklov, merali sme čas a robili aritmetický priemer úspešnosti redukovania aj časovej náročnosti.

Ukážka kódu:

```
for (int i = 0; i < REPEAT; i++) {

start = clock();
        diagram = BDD_create(getBF());
        BDD_reduce(diagram);
end = clock();

startU = clock();
        for(int j = 0; j < pow(2, N); j++){
           a = BDD_use(diagram, u[j]);
           if (a == -1) printf("Er");
        }
endU = clock();

        per = ((float)(diagram->size)/vrchol) * 100;

        printf("povodne : %d \nredukovane : %d\npercenta : %lf\n", vrchol, diagram-
>size, per);
        cnt = 0;
        avgP += per;
        time_used = ((double)(end - start));
        time_usedU = ((double)(endU - startU));
        avgT += time_used;
        avgTU += time_usedU;
}

printf("AVG %% : %lf\nAVG TIME CREATE & REDUCE : %gms\nAVG TIME USE : %gms\n",
 avgP / REPEAT, avgT / REPEAT, avgTU / REPEAT);
```

Booleovskú funkciu vo forme vektora sme vytvárali v pomocnej funkcií *„getBF“*.

Ukážka kódu:

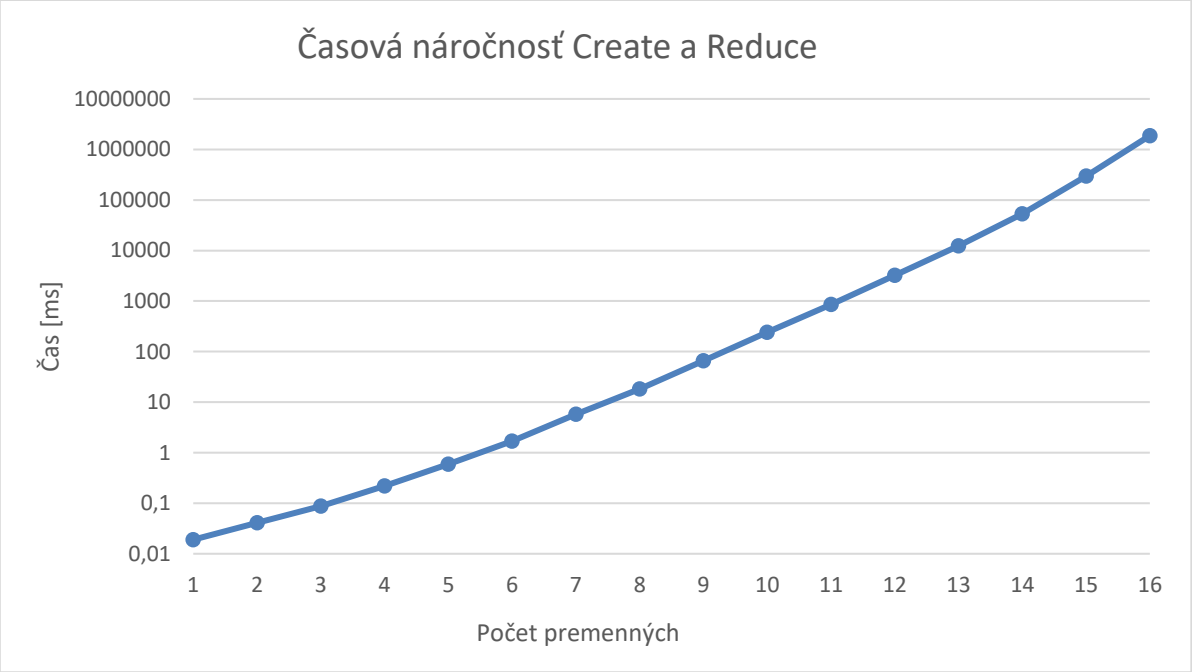
```
        char* getBF() {
            int len = pow(2, N);
            char* vector = malloc(len);

            for (int i = 0; i < len; i++) {
                if (rand() % 10 > 4) {
                    vector[i] = '0';
                }
                else {
                    vector[i] = '1';
                }
            }
            vector[len] = '\0';

            return vector;
        }
```

5 Výsledky

Testovali sme nie len prípad zo zadania (najmenej 13 premenných, 2000 krát), ale aj iný počet premenných. Pri jednej až trinástich sme opakovali cyklus 2000 krát, pri viac premenných 10krát. Dostali sme sa až na 19 premenných. Cieľom bolo vizuálne čo najdôveryhodnejšie reprezentovať naše riešenie pomocou grafov.

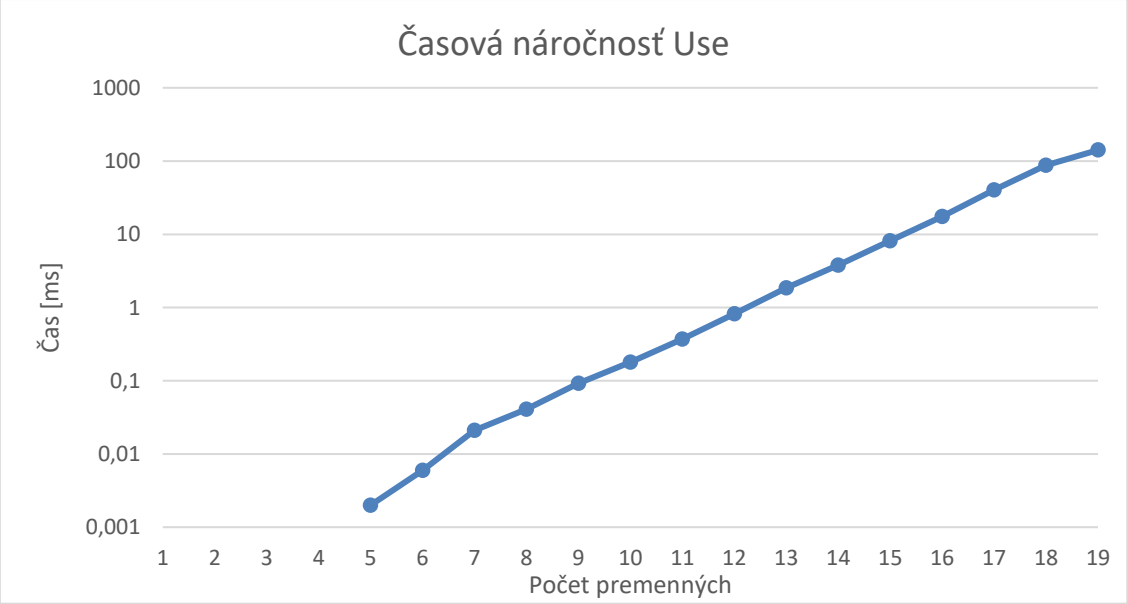
Časové trvanie funkcií *create* a *reduce* sme merali spolu, funkciu *use* samostatne.

Výsledky trvania vytvorenia a redukovania BDD:



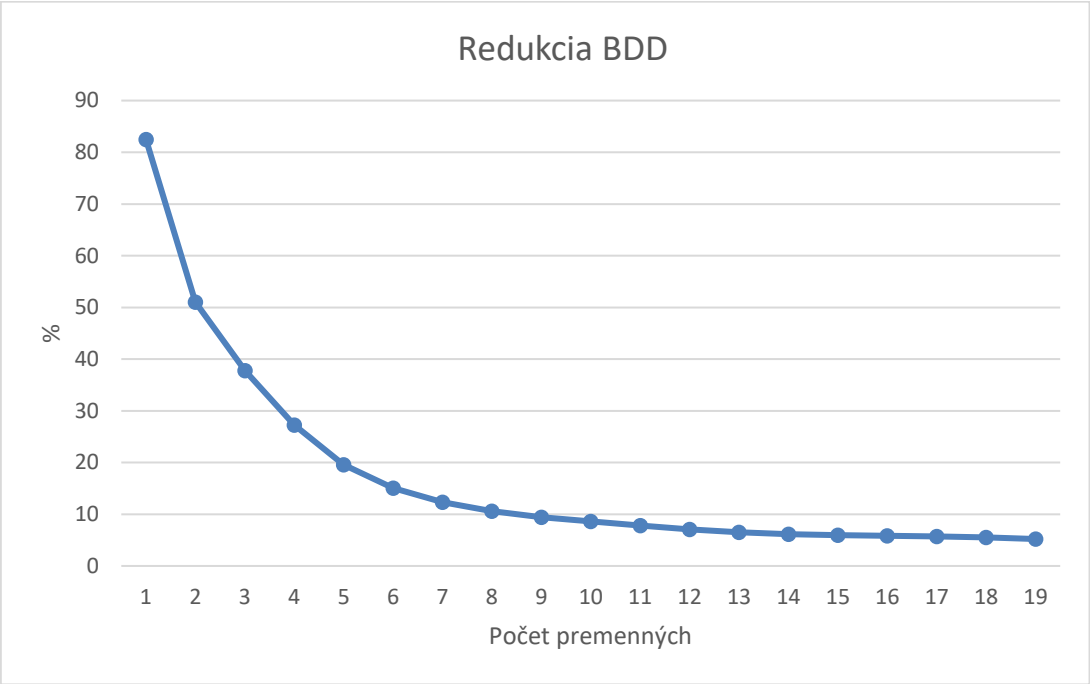
Graf 1

Výsledky trvania používania BDD:



Graf 2

Výsledky redukovania BDD:



Graf 3

Dáta:

Počet premenných	**[2]1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Čas *create, reduce* [ms]	0,002	0,004	0,012	0,019	0,041	0,088	0,218	0,594	1,695	5,738
Redukované BDD [%]	82,467	51,014	37,767	27,278	19,597	15,089	12,32	10,58	9,448	8,609
Čas *use* [ms]	0	0	0	0	0,002	0,006	0,021	0,041	0,092	0,179
Počet premenných	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	
Čas *create, reduce* [ms]	18,34	65,432	242,366	852,65	3224,35	12325,4	53368	298641	1873780	
Redukované BDD [%]	7,82	7,078	6,508	6,158	5,988	5,846	5,715	5,543	5,213	
Čas *use* [ms]	0,37	0,82	1,85	3,8	8.1	17,4	40,2	87,2	141,2	

Tabuľka 1

Časová aj priestorová náročnosť je exponenciálna, časovú vidieť na grafe číslo jedna a dva, ktoré majú logaritmickú škálu. Efektívnosť redukovania sa blíži ku piatim percentám. Bolo by možné dostať lepšie výsledky, ak by Booleovské funkcie boli jednoduchšie, my tvoríme vektor pomocou náhodného generovania čísel, čo nepredstavuje najjednoduchšie B. funkcie. Efektívnosť redukovania v závislosti na počet premenných klesá v dôsledku väčšej šance opakovať rovnakú časť bfunkcie, tým pádom viac totožných hodnôt vo vrcholoch a menej vrcholov po redukovaní.

[2] Oranžovo sú podfarbené premenné, pri ktorých sme opakovali cyklus 2000 krát

Bibliografia

Kohútka, L. (20. marec 2021). Pokročilé algoritmy vyhľadávania. Dostupné na Internete: dokumentový server is.stuba.sk, DSA_05