SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
Fakulta informatiky a informačných technológií
Ilkovičová 2, 842 16 Bratislava 4

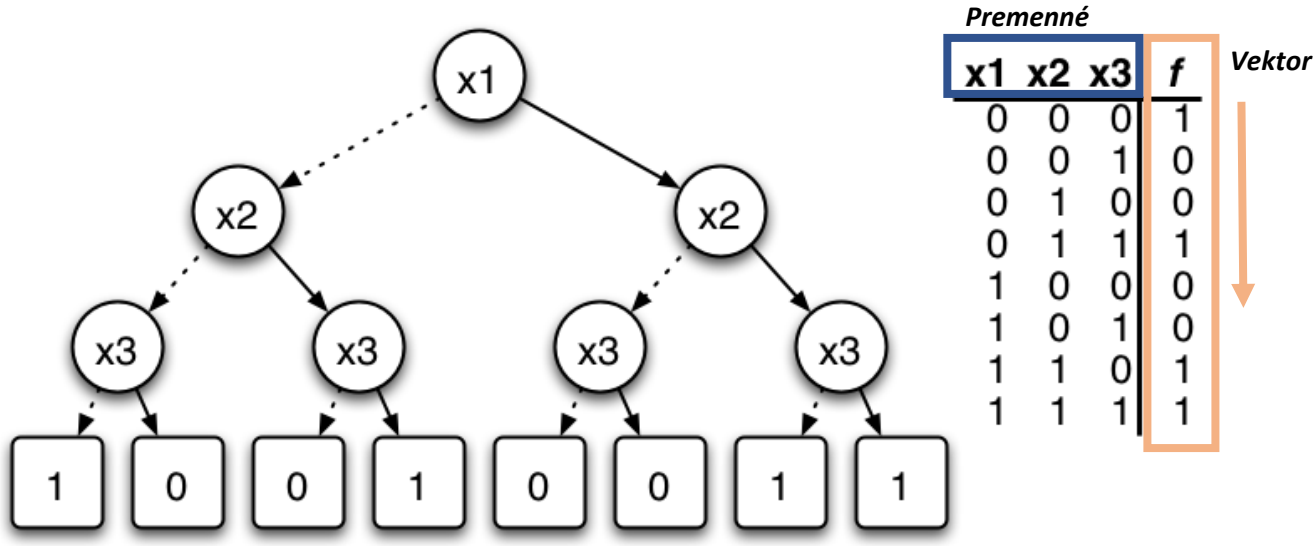
# Zadanie 3 – Binárne rozhodovacie diagramy

ADRIÁN VANČO
ID: 103171

Cvičenie: Utorok 18:00
3.5.2021

# Úvod

Binárny rozhodovací diagram je stromová štruktúra pre reprezentovanie booleovskej funkcie. Strom sa skladá z niekoľkých (rozhodovacích) uzlov a dvoch koncových uzlov. Dva koncové uzly označujú 0 (FALSE) a 1 (TRUE). Každý (rozhodovací) uzol je označený (booleovskou premennou, vektorom,...) a má dva podriadené uzly (potomkov), prechod do potomkov predstavuje pridanie hodnoty FALSE – prechod do ľavého potomka (alebo TRUE – prechod do pravého potomka).




*Obrázok 1 Príklad nezredukovaného BDD s pravdivostnou tabuľkou*

# Reprezentácia BDD

```c
//bunka BDD
typedef struct bdd {
    int numberOfVariables;
    int numberOfNodes;
    struct node* root;
}BDD;

//bunka v BDD
typedef struct node {
    struct bf* nodeVector;
    struct node* left;
    struct node* right;
    struct node* nextRight; //ukazovatel na dalsi na pravo v rovnakej urovni
}NODE;

//bunka funkcie
typedef struct bf {
    int size;
    char* vector;
}BF;
```

# Zoznam funkcií

```c
//==========deklaracie funkcii=============

BF* BF_create(int size, char* vector);
NODE* NODE_create(BF* bfunction);
void BDD_fill(NODE** node, BF* bfunction);
int BF_check(BF* bfunction);
BDD* BDD_create(BF* bfunkcia);

void BDD_nextRight_connect_next(NODE* node, int iter);
void BDD_nextRight_connect_over_unwanted(NODE* node, NODE* next);
void BDD_reduce_siblings(NODE* node, int* counter);
void BDD_delete_unwanted(NODE** node, int* counter, NODE* to_free, NODE* root);
void NODE_change_every_parent(NODE* parent, NODE* new_child, NODE* old_child);
void BDD_reduce_level(NODE* node, int* counter, NODE* root);
void BDD_connect_levels(NODE* node);
int BDD_reduce(BDD* bdd);

char BDD_use(BDD* bdd, char* vstupy);

void NODES_free(NODE* node);
void BDD_free(BDD* bdd);
void BDD_free_level(NODE* root, NODE* node);
void BDD_free_nodes(NODE* root, int root_size);

void BDD_print(NODE* node);
```

# BDD_create()

Funkcia **BDD_create** slúži. na vytvorenie úplného binárneho rozhodovacieho diagramu. V tejto funkcii sa alokuje BDD štruktúra, skontroluje sa vstupná funkcia, ktorá je reprezentovaná ako štruktúra BF. V BF štruktúre je uložená funkcia reprezentovaná vektorom ako pole znakov ‚1' a ‚0' a veľkosť vektora. Po kontrole sa do BDD štruktúry uložia parametre ako sú počet premenných a počet uzlov, ktoré budú vytvorené.

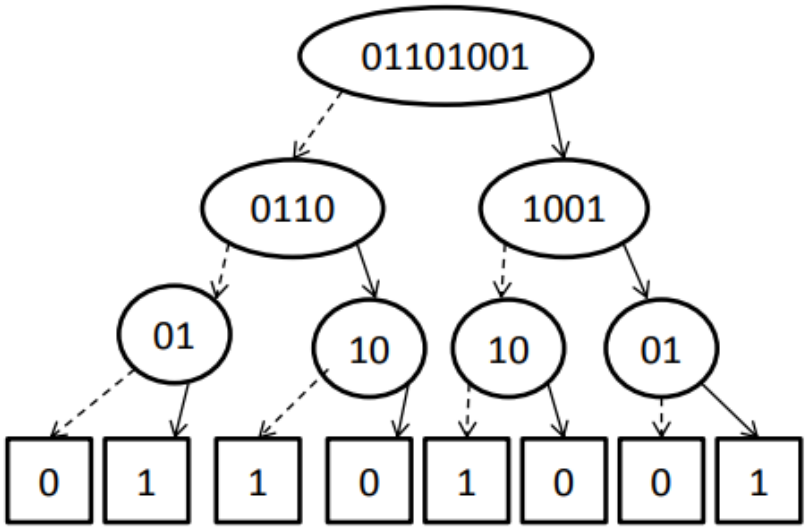
```
temp->numberOfVariables = ((int)log2(bfunction->size)); //počet premenných
temp->numberOfNodes = (bfunction->size * 2) - 1;     //počet prvkov
temp->root = NULL;
```

Potom sa zavolá rekurzívna funkcia, ktorá naplní túto BDD štruktúru a po naplnení sa vráti adresa, kde sa BDD v pamäti nachádza.

```
BDD_fill(&temp->root, bfunction);    //naplnenie BDD

return temp;     //vratenie BDD
```

Po vykonaní je BDD strom nezredukovaný. A napr. pre vstupnú funkciu s vektorom „01101001" vyzerá nasledovne.




K **BDD_create** a **BDD_fill** mám ešte implementované vlastné 4 funkcie

```
//funkcia na vytvorenie bunky pre vektor
BF* BF_create(int size, char* vector){ ... }

//funkcia na vytvorenie bunky v BDD
NODE* NODE_create(BF* bfunction){ ... }

//rekurzivna funkcia na naplnenie BDD
void BDD_fill(NODE** node, BF* bfunction){ ... }

//funkcia na overenie ci sa vstupna funkcia sklada len z 0 a 1;
int BF_check(BF* bfunction){ ... }
```

# BDD_reduce()

Funkcia slúži na zredukovanie počtu uzlov v BDD. Vo funkcií na najprv skontroluje či je BDD alokovaný, ak nie funkcia vráti -1. Ak BDD bol zavolá sa rekurzívna funkcia na pospájanie úrovní v BDD.

```
BDD_connect_levels(bdd->root);  //pospajanie urovny v BDD
```

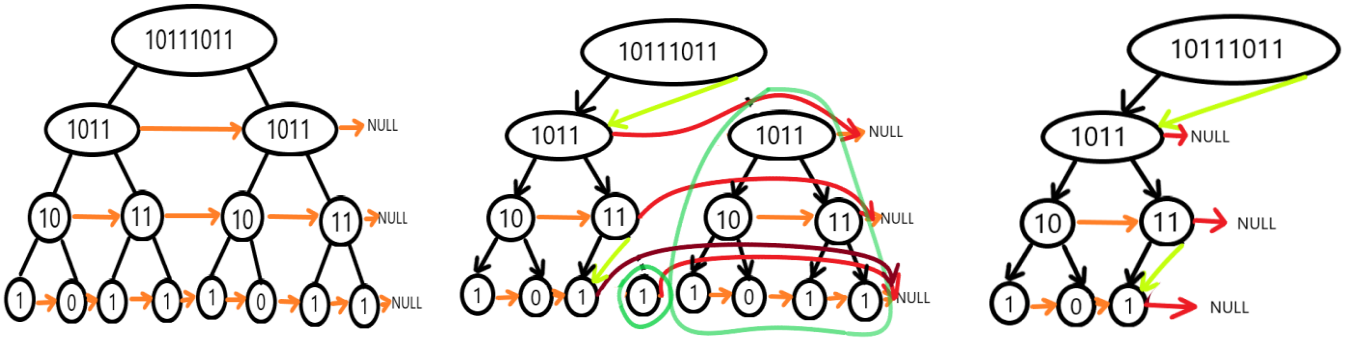
Ďalej vo funkcií mám count, iter na použitie pre poslednú funkciu na redukciu BDD značí začiatočnú úroveň pre poslednú funkciu a temp tiež na využitie pre poslednú funkciu uvolňovania.

```
int count = 0;  //pocet NODOV, ktore sa odstranili
int iter = 1;
NODE* temp = NULL;
```

BDD redukujem 3 funkciami.

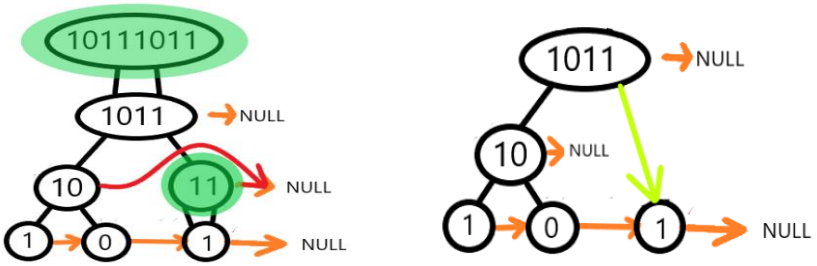
```
//redukuje v BDD potomkov s rovnakym vektorom
BDD_reduce_siblings(bdd->root, &count);
```

V **BDD_reduce_siblings** sa redukuje zhora nadol. Ak v uzle sú potomkovia s totožným vektorom, zavolá sa funkcia **BDD_nextRight_connect_next** na obídenie stromu pravého potomka v úrovniach následne sa v uzle ukazovateľ na pravého potomka nastaví na ľavého potomka. Potom nasleduje uvoľnenie stromu pravého potomka, pričítanie počtu uvoľnených uzlov a ďalšie vnáranie.



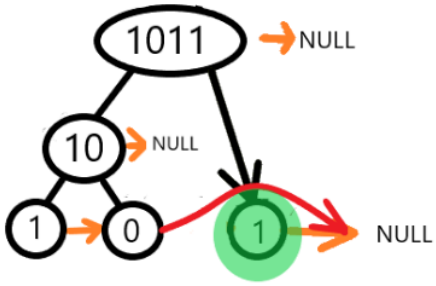

```
//redukuje v BDD NODE, ktory ma potomkov jeden a ten isty NODE
BDD_delete_unwanted(&bdd->root, &count, NULL, bdd->root);
```

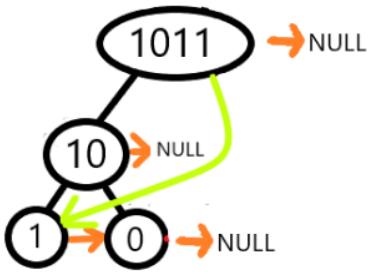
Ďalej v **BDD_delete_unwanted** sa odstraňujú uzly, ktoré ukazujú na toho samého potomka.



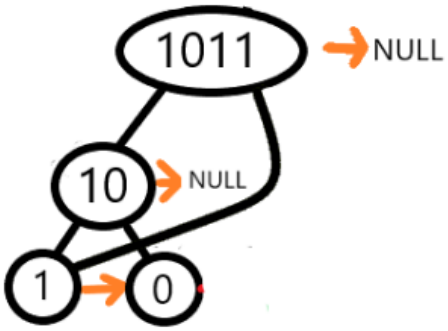
Na záver sa ide na iteratívne volanie funkcie na zredukovanie úrovní. Začína sa redukovať od najspodnejšej úrovne postupujúc smerom k vrcholu. Ak sa v úrovni nachádzajú uzly z rovnakým vektorom, jeden uzol sa uvoľní. A všetci, ktorý na tento uzol ukazovali začnú ukazovať na neuvoľnený pomocou funkcie **NODE_change_every_parent**.

```
//cyklus na zlucovanie v urovniach, zacina sa od spodku postupujuc k vrcholu
while (iter != bdd->root->nodeVector->size) {
    temp = search_first_from_left(bdd->root, iter); //najdem zaciatocny NODE pre uroven
    BDD_reduce_level(temp, &count, bdd->root);  //funkcia na uvolnenie
    iter = iter * 2;    //zvacsuje sa * 2 lebo aj vektory sa zvacsuju dvojnasobne
}
```




Po vykonaní výsledný BDD vyzerá nasledovne




Na záver sa aktualizuje počet uzlov v BDD a funkcia vráti počet uvoľnených uzlov.

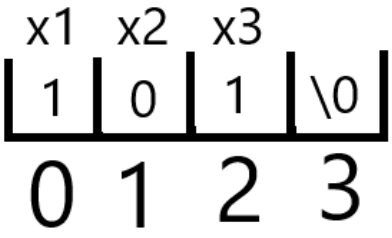
```
bdd->numberOfNodes = bdd->numberOfNodes - count;

return count;
```

# BDD_use()

Vstupné pole znakov je vektor, ktorý reprezentuje hodnotu premenných. Ak je BDD nealokovaný alebo vstupný vektor je zlého rozmeru funkcia vráti -1 značiacu chybu. Ak je všetko v poriadku prejde sa na cyklus, ktorý skončí keď sme v uzle, ktorého vektor je veľkosti 1.

Rozhodovanie vnárania je na základe hodnoty na indexe v poli znakov. Táto hodnota značí hodnotu premennej




```
//cyklus na najdenie vyslednej hodnoty pre zadany vektor premennych
//podla hodnoty na danom indexe vo vektore premennych sa vnori
while (temp->nodeVector->size != 1) {
    if (vstupy[index] == '0') {
        temp = temp->left;
        index++;
    }
    else if (vstupy[index] == '1') {
        temp = temp->right;
        index++;
    }
    else {
        printf("Zle zadany vstup");
        return -1;
    }
}

return temp->nodeVector->vector[0]; //vratenie vyslednej hodnoty
```

# Testovanie

Na testovanie som si spravil 3 funkcie

```
char* rand_string(char* str, int size);
void Decimal_to_binary_string(int number, int array_size, char* variables);
void test_all(int numberOfVariables, int iter, int use);
```

Funkcia **rand_string** vygeneruje náhodný vektor o požadovanej veľkosti, ktorý využívam vo funkcii **test_all** ako vektor booleovskej funkcie.

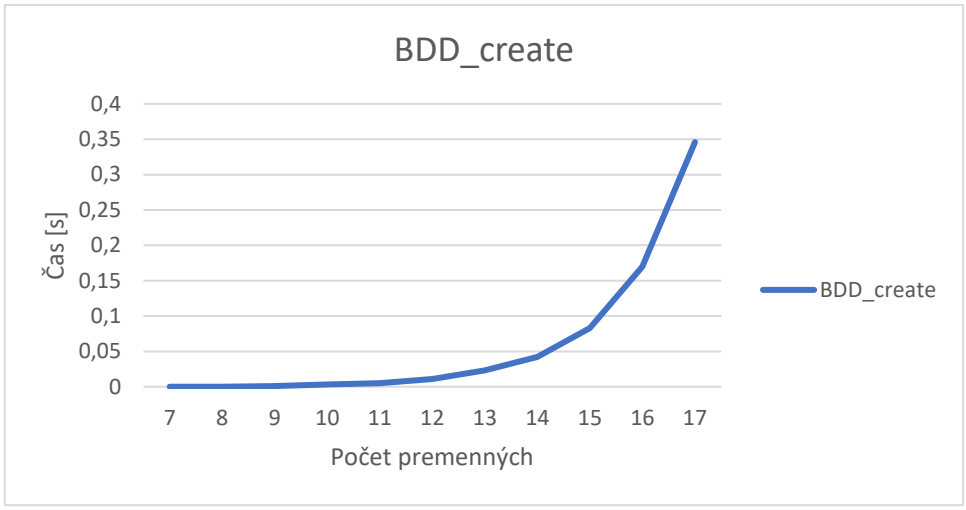
```
//generator stringov o pozadovanych rozmeroch
char* rand_string(char* str, int size)
{
    const char charset[] = "01101010101000000111111";
    if (size) {
        --size;
        for (int i = 0; i < size; i++) {
            int key = rand() % (int)(sizeof charset - 1);
            str[i] = charset[key];
        }
        str[size] = '\0';
    }
    return str;
}
```

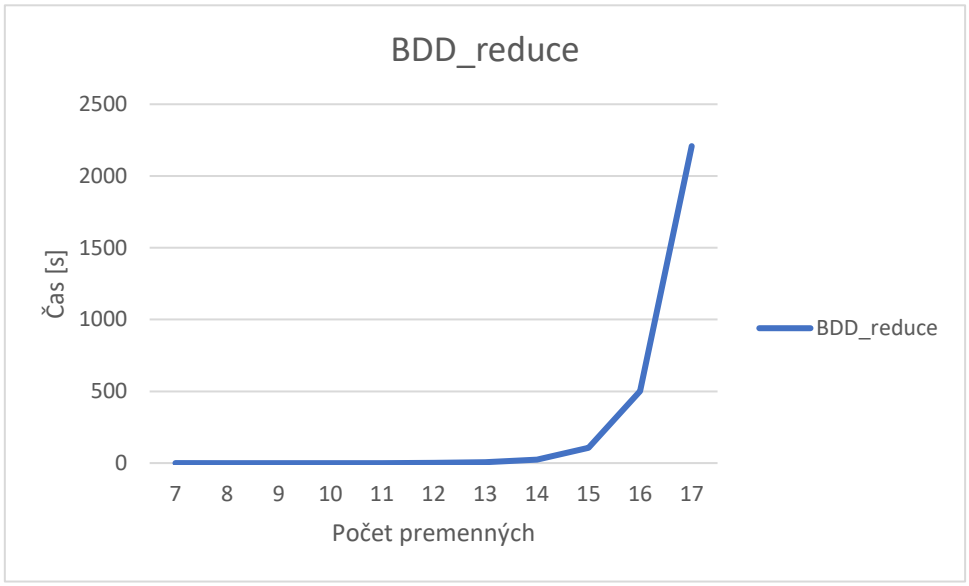
Funkcia **Decimal_to_binary_string**, do tejto funkcie posielam číslo, ktoré chcem uložiť v binárnej reprezentácií do poľa znakov variables.

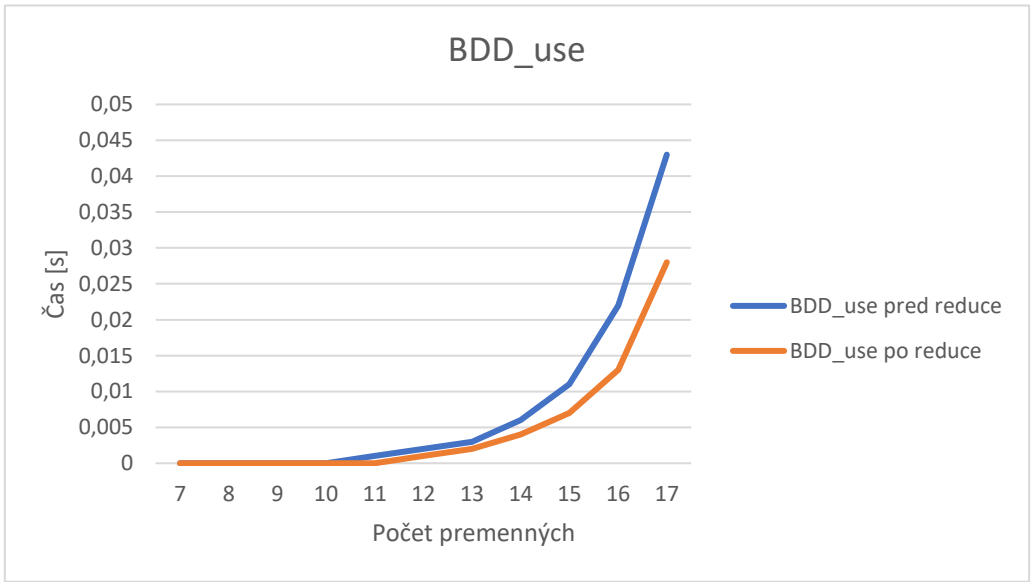
```
//do pola variables o presnej velkosti, ktore je vo funkcii test_all vlozi binarne dane cislo
void Decimal_to_binary_string(int number, int array_size, char* variables) {
    int bit;
    int index = array_size - 2; //posledny index kde sa bude vkladat '1' alebo '0'
    if (variables != NULL) {
        for (int i = 0; i < array_size - 1 ; i++) {
            bit = number >> i;  //binarny posun
            if (bit & 1) {  //ak je po posune na kraji jednotka ulozime '1' inak '0'
                *(variables + index) = '1';
            }
            else {
                *(variables + index) = '0';
            }
            index--;
        }
        *(variables + array_size - 1) = '\0';
    }
    else {
        printf("Chyba pri decimal to binary string");
    }
}
```

Funkcia **test_all**, do tejto funkcie posielam pre aký počet premenných chcem vytvoriť náhodnú funkciu, ďalej koľko krát chcem aby test prebehol zakaždým s iným vektorom, a či chcem odtestovať aj BDD_use pre všetky možné kombinácie nad BDD.

# Grafy testovania



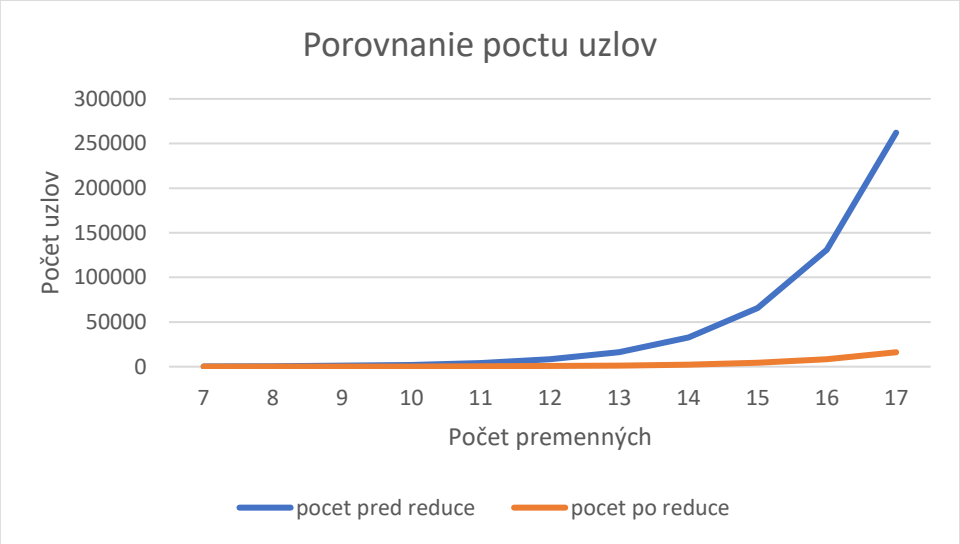



# Zhodnotenie

Na vyhodnotenie hodnôt pre jednotlivé premenné sa bral priemerný čas zo vzorky 200 testov. Pri testoch na premenných 14 až 16 zo vzorky 10 testov a pri 17 premenných 1 test. BDD_use sa odtestoval vo všetkých testoch s tým že sa vyskúšali všetky kombinácie BDD_use pre daný vektor.

Časová zložitosť podľa počtu premenných rástla exponenciálne.

BDD_reduce zredukoval strom v priemere o 90%. Pri 17 premenných zredukoval strom o 93% a pri 7 premenných 82%.



# Zdroje

Obrázok 1: https://en.wikipedia.org/wiki/Binary_decision_diagram#/media/File:BDD.png