Oleksandr Vahabov, Pondelok 16:00

1. Úvod

Cieľom tejto úlohy bolo implementovať dátovú štruktúru Binary Decision Diagram (BDD) na efektívnu reprezentáciu logických funkcií. BDD som implementoval s redukciami typu S a I, vstupnú boolovskú funkciu som reprezentoval ako logickú formulu (reťazec DNF) a vyhodnotil jej správnosť a efektívnosť pomocou testovania.

2. Implementacia BDD

2.1) Datové štruktury

```c
typedef struct BDDNode {
    char var;
    struct BDDNode *low;
    struct BDDNode *high;
} BDDNode;
```

Táto štruktúra definuje uzol v binárnom rozhodovacom diagrame. Každý uzol zodpovedá

rozhodovaciemu bodu pre konkrétnu logickú premennú.

var: identifikuje, ktorá premenná sa v tomto uzle vyhodnocuje, podľa jej indexu.

low: ukazovateľ na podriadený uzol, ak sa premenná vyhodnocuje ako 0.

high: ukazovateľ na podriadený uzol, ak sa premenná vyhodnocuje ako 1.

```c
typedef struct BDD {
    BDDNode *root;
    int size;
    char *var_order;
    HashTable *hash_table;
} BDD;
```

Táto štruktúra predstavuje celkový binárny rozhodovací diagram pre danú booleovskú funkciu.

root: vstupný bod BDD, z ktorého sa začína vyhodnocovanie.

Oleksandr Vahabov, Pondelok 16:00

size: počet unikátnych uzlov BDD vo finálnom redukovanom diagrame.

var_order: Poradie premenných použité pri vytváraní BDD

hash_table: Hašovacia tabuľka, ktorá slúži na redukovanie typu I.

```
typedef struct HashEntry {
    char var;
    BDDNode *low;
    BDDNode *high;
    BDDNode *node;
    struct HashEntry *next;
} HashEntry;
```

HashEntry predstavuje jednu položku v hašovacej tabuľke

Var, low, high – to iste čo v štrukture BDDNode

Node: Už vytvorený uzol BDD, slúži ako výsledok hľadania alebo uloženie, aby sa zabránilo duplicitám

Next: Ukazovateľ na ďalší záznam pri kolízii v haš tabuľke.

```
typedef struct HashTable {
    HashEntry **list;
    int size;
    int num_nodes;
} HashTable;
```

HashTable reprezentuje hašovaciu tabuľku na redukovanie

List: Pole ukazovateľov na zoznamy HashEntry

Size: veľkosť poľa list

Num_nodes: Počet unikatnych uzlov uložených v haš tabuľke

```
typedef struct Minterm {
    int zero_flag;
    char *vars;
    int var_count;
    struct Minterm *next;
} Minterm;
```

Oleksandr Vahabov, Pondelok 16:00

Minterm sa používa na reprezentáciu mintermu v booleovskej funkcii

Zero_flag: Príznak nulového mintermu

Vars: Reťazec premenných, ktorý reprezentuje konkrétny minterm.

Var_count: Počet premenných

Next: Ukazovateľ na ďalší minterm v linked liste mintermov

```
typedef struct Expression {
    Minterm *head;
    int zero_flag;
    int one_flag;
    int minterm_length;
} Expression;
```

Expression reprezentuje booleovskú funkciu vo forme linked lista mintermov

Head: Ukazovateľ na prvý minterm v linked liste mintermov.

Zero_flag: pre nulový výsledok.

One_flag: pre jednotkový výsledok.

Minterm_length: Počet mintermov

2.2) Funckcie

Expression *parse(char *expr)

Táto funkcia slúži na spracovanie reťazca booleovskej výrazu (napr. "abd+cd") a vytvorenie štruktúry Expression, ktorá reprezentuje tento výraz ako zoznam mintermov.

Iteruje cez každý znak reťazca:

- Ak je znak '!', nastaví sa príznak negate, ktorý indikuje, že nasledujúca premenná bude znegovaná.
- Ak je znak medzi 'a' a 'z', volá sa funkcia add_letter, ktorá pridá premennú do aktuálneho mintermu. Negácia sa aplikuje, ak je príznak negate nastavený.
- Ak je znak '+', znamená to, že sa začína nový minterm, a teda vytvorí sa nový minterm a pripojí sa k aktuálnemu.

Oleksandr Vahabov, Pondelok 16:00

```
int i = 0;
while ((c = expr[i++])) {
    if (c == '!') {
        negate = 1;
        continue;
    }

    if (c >= 'a' && c <= 'z') {
        add_letter(current, negate ? -c : c);
        negate = 0;
        continue;
    }

    if (c == '+') {
        Minterm *new_minterm = calloc(1, sizeof(Minterm));
        current->next = new_minterm;
        current = new_minterm;
        expression->minterm_length++;
    }
}
```

Expression *substitution(Expression *expr, signed char letter)

vykonáva dosadenie konkrétnej premennej do booleovského výrazu.

Prechádza každý minterm:

- Ak nájde dosadzovanú premennú, odstráni ju zo zoznamu premenných v danom minterme
- Ak minterm zostane prázdny znamená to, že výraz je vždy pravdivý
- Ak nájde negáciu tejto premennej (-letter), nastaví príznak zero_flag pre daný minterm - tento minterm sa stáva nulovým

Oleksandr Vahabov, Pondelok 16:00

```
while (item) {
    int i = 0;
    while (i < item->var_count) {
        if (item->vars[i] == letter) {
            memmove(&item->vars[i], &item->vars[i + 1],
                    (item->var_count - i - 1) * sizeof(signed char));
            item->var_count--;
            if (item->var_count == 0) {
                result->one_flag = 1;
                result->zero_flag = 0;
                return result;
            }
            continue;
        }
        if (item->vars[i] == -letter) {
            item->zero_flag = 1;
            break;
        }
        i++;
    }

    if (!item->zero_flag) {
        zero_result = 0;
    }
    item = item->next;
}
```

BDDNode *find_or_add_unique_node(HashTable *hash_table, char var, BDDNode *low, BDDNode *high)

Táto funkcia zabezpečuje, že sa v BDD nevytvárajú duplicitné uzly s rovnakou štruktúrou. Tu je jej význam stručne a jasne:

Ak low == high, teda ak obidve vetvy vedú na rovnaký uzol, logicky nie je potrebné vytvárať nový uzol – vráti sa existujúci

Hľadá v hash_table, či už existuje uzol s rovnakou premennou var, low a high.
Ak existuje, vráti ho. Ak nie, vytvorí nový uzol, nastaví jeho low a high vetvy a uloží ho do hash tabuľky, aby sa mohol znovu použiť v budúcnosti.

```
if (low == high) return low;

BDDNode *existing = search(hash_table, var, low, high);
if (existing) {return existing;}

BDDNode *node = create_node(var);
node->low = low;
node->high = high;

insert_node(hash_table, var, low, high, node);

return node;
```

Oleksandr Vahabov, Pondelok 16:00

BDDNode *build_bdd(Expression *expression, char *var_order, int level, HashTable *hash_table)

rekurzívne vytvára binárny rozhodovací diagram (BDD) pre zadaný logický výraz podľa poradia premenných. Z daného logického výrazu (vo forme štruktúry Expression) a poradia premenných (var_order) vybuduje kanonický BDD strom, ktorý reprezentuje ten istý logický výraz.

```
if (!found) {
    return build_bdd(expression, var_order, level + 1, hash_table);
}

Expression *f_high = substitution(expression, current);
Expression *f_low = substitution(expression, -current);

BDDNode *high_node = build_bdd(f_high, var_order, level + 1, hash_table);
BDDNode *low_node = build_bdd(f_low, var_order, level + 1, hash_table);

free_expression(f_high);
free_expression(f_low);

if (high_node == low_node) {
    return high_node;
}

return find_or_add_unique_node(hash_table, current, low_node, high_node);
```

BDD* create_BDD(char *expression, char *var_seq)

Funkcia vytvára binárny rozhodovací diagram (BDD) zo zadaného logického výrazu a poradia premenných. Najskôr alokuje pamäť pre štruktúru BDD a inicializuje hashovaciu tabuľku, ktorá slúži na ukladanie jedinečných uzlov diagramu.

```
BDD *bdd = calloc(1, sizeof(BDD));

bdd->hash_table = create_hash_table(HASH_SIZE);
bdd->size = 0;
```

Následne sa reťazec s logickým výrazom spracuje do vnútornej štruktúry Expression, s ktorou sa pohodlnejšie pracuje.

```
Expression *expr = parse(expression);
```

Potom zavolá sa funkcia build_bdd, ktorá prechádza výraz podľa poradia premenných a rozdeľuje ho na vetvy podľa hodnôt aktuálnej premennej, čím sa postupne vytvára rozhodovací diagram.

```
bdd->root = build_bdd(expr, vars, 0, bdd->hash_table);
free_expression(expr);

return bdd;
```

BDD *create_BDD_with_best_order(char *expr, char *var_seq)

Oleksandr Vahabov, Pondelok 16:00

Táto funkcia vytvára binárny rozhodovací diagram pre daný logický výraz, pričom hľadá najlepšie poradie premenných pomocou rotacie vyraza zo zadaného reťazca var_seq, ktoré minimalizuje veľkosť BDD

Ak premenné sú prítomné, funkcia skúša rôzne rotácie ich poradia (napr. z "abcd" spraví "bcda", "cdab", "dabc"). Pre každú rotáciu vytvorí nový BDD pomocou funkcie create_BDD a porovná jeho veľkosť s doteraz najlepším výsledkom. Ak má nový BDD menšiu veľkosť, starý sa uvoľní a nový sa uloží ako najlepší. Na konci funkcia vráti najkompaktnejší (najmenší) BDD podľa všetkých vyskúšaných rotácií poradia premenných.

```
for (int i = 0; i < count; i++) {
    char *order = strdup(vars);

    if (i > 0) {        // (i > 0) to check the default case
        char temp = order[0];
        memmove(order, order + 1, count - 1);       // here
        order[count - 1] = temp;                    // abcd
    }

    BDD *b = create_BDD(expr, order);

    if (!best || b->hash_table->num_nodes < best_size) {
        free_bdd(best);
        best = b;
        best_size = b->hash_table->num_nodes;
    } else {
        free_bdd(b);
    }

    free(order);
}
```

char BDD_use(BDD *bdd, char *input_bits)

funkcia slúži na vyhodnotenie BDD s použitím konkrétnych vstupných hodnôt premenných z reťazca input_bits.

Následne funkcia prechádza uzlami BDD podľa toho, ktorá premenná je v uzle, a podľa zodpovedajúceho bitu v mape sa rozhodne, či pôjde na vetvu low (ak je bit 0), alebo high (ak je bit 1). Tento proces pokračuje, až kým nenarazí na listový uzol (TRUE alebo FALSE), ktorého hodnota sa vráti ako výsledok – '1' pre pravdu, '0' pre nepravdu.

```
while (node && node != &TRUE && node != &FALSE) {
    char var = node->var;
    int idx = var - 'a';
    if (idx < 0 || idx >= 26) return -1;

    int decision = bit_map[idx];
    if (decision == 0)
        node = node->low;
    else if (decision == 1)
        node = node->high;
    else
        return -1;
}
```

3. Testovanie

```
Num of variables: 13
Num of expressions: 100
Accuracy: 100.00%
Reduction: 99.90%
Best order reduction: 2.78%
Time for BDD creation: 0.72 seconds
Time for BDD with best order creation: 8.96 seconds
```

Najprv testoval som správnosť vytvoreného BDD. Pomocou funkcie *all_combinations* najprv vygeneroval som všetky možné vstupné kombinácie pre dané premenné, potom pomocou funkcie *test_accuracy* porovnám všetky kombinácie použité v BDD_use s výstupnými hodnotami z funkcie *evaluate_expression*, aby overiť, že správanie BDD zodpovedá očakávanému správaniu boolovskej funkcie pre všetky možné vstupné kombinácie.

```
all_combinations(num_vars, combinations);
for (int i = 0; i < number_combinations; i++) {
    char expected = evaluate_expression(expr, vars, num_vars, combinations[i]);
    char result = BDD_use(bdd, combinations[i]);
    if (expected != result) {
        for (int j = 0; j < number_combinations; j++) {
            free(combinations[j]);
        }
        free(combinations);
        return 0;
    }
}
```

Potom testujem, ako sa zmenší veľkosť BDD po redukcii a vytvorení BDD s

najlepším poradím. Pomocou funkcie double *evaluate_reduction* vypočítam ako dobre redukoval som BDD.

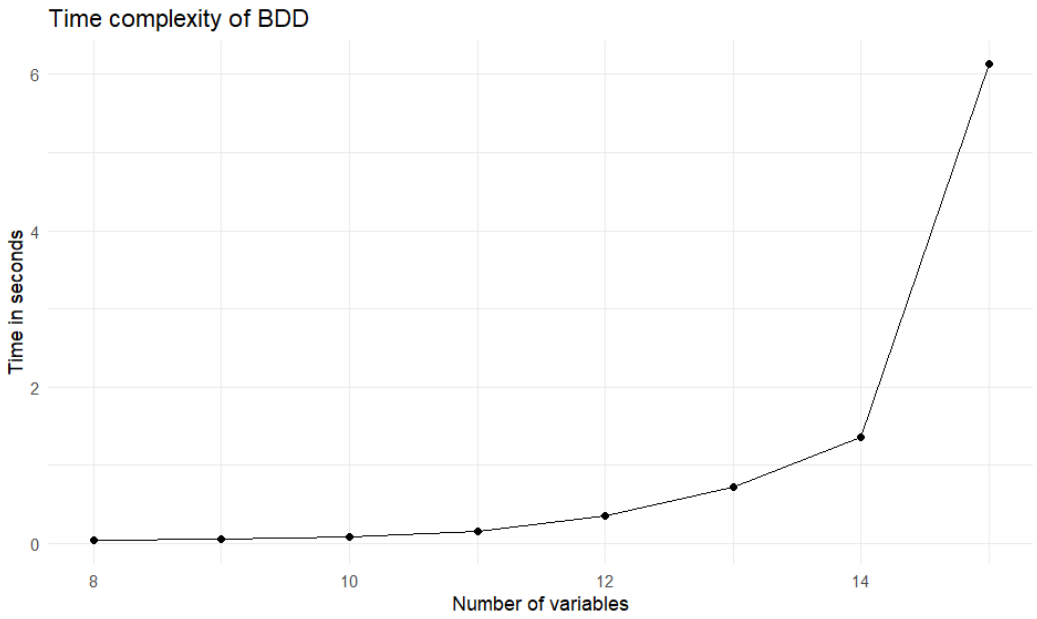
```
    int full_size = (1 << (num_vars + 1)) - 1;
    double reduction = evaluate_reduction(full_size, bdd->hash_table->num_nodes);
    double best_bdd_reduction = evaluate_reduction(bdd->hash_table->num_nodes, best_bdd->hash_table->num_nodes);
```

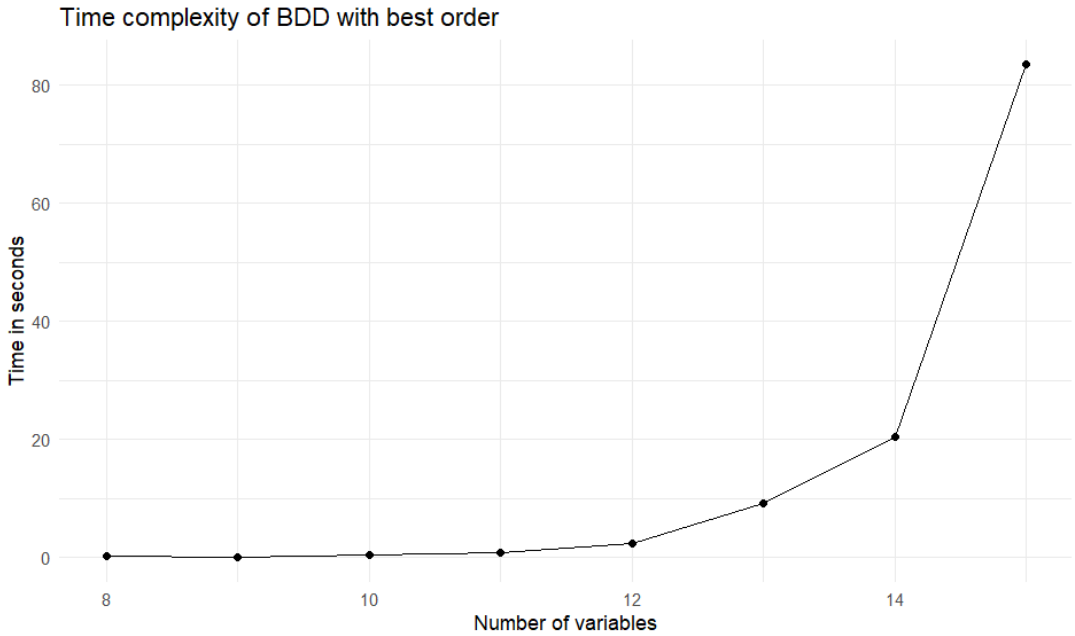
V priemere som dospel k záveru, že pomocou redukcie sa BDD zníži až o

až 99 % nodov, zatiaľ čo nájdenie najlepšieho poradia BDD je menej účinné a znižuje BDD v priemere o 3 %.

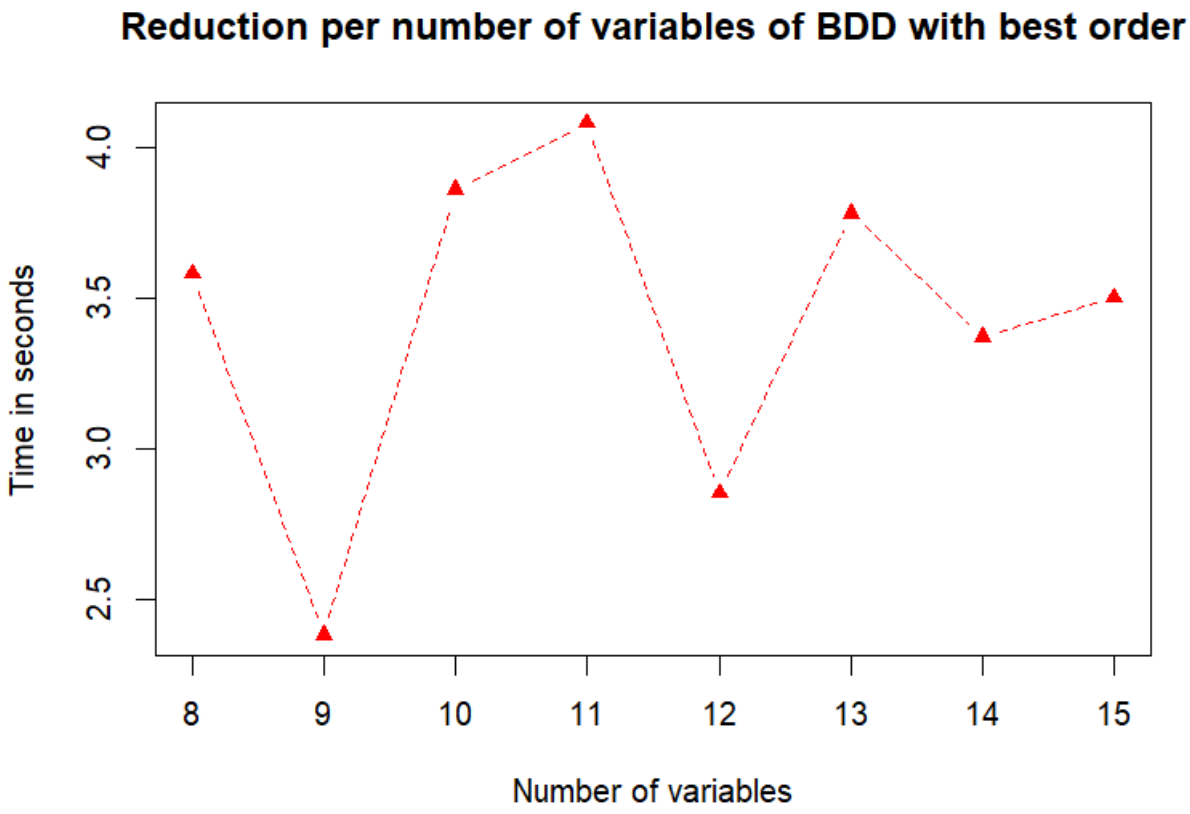
Ďalej som porovnal časovú zložitosť vytvorenia jednoduchého BDD a BDD s najlepším poradím a iné vlastnosti

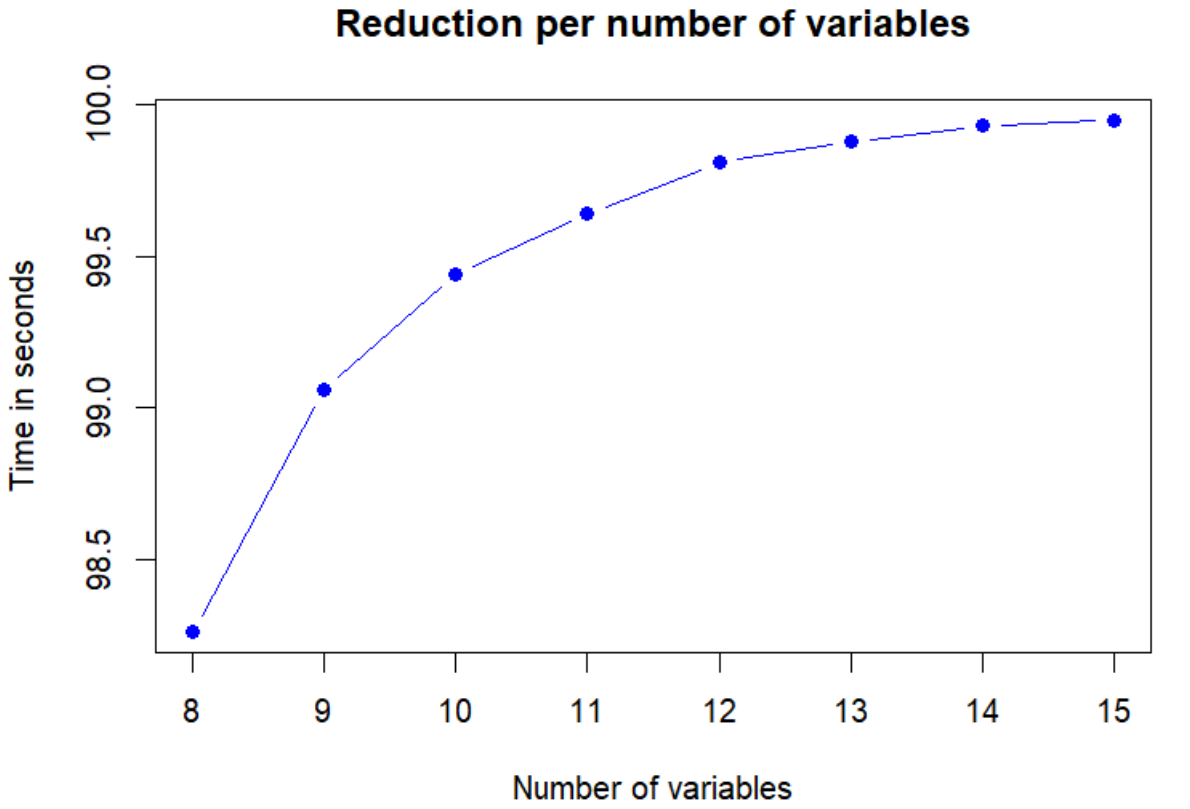
Oleksandr Vahabov, Pondelok 16:00



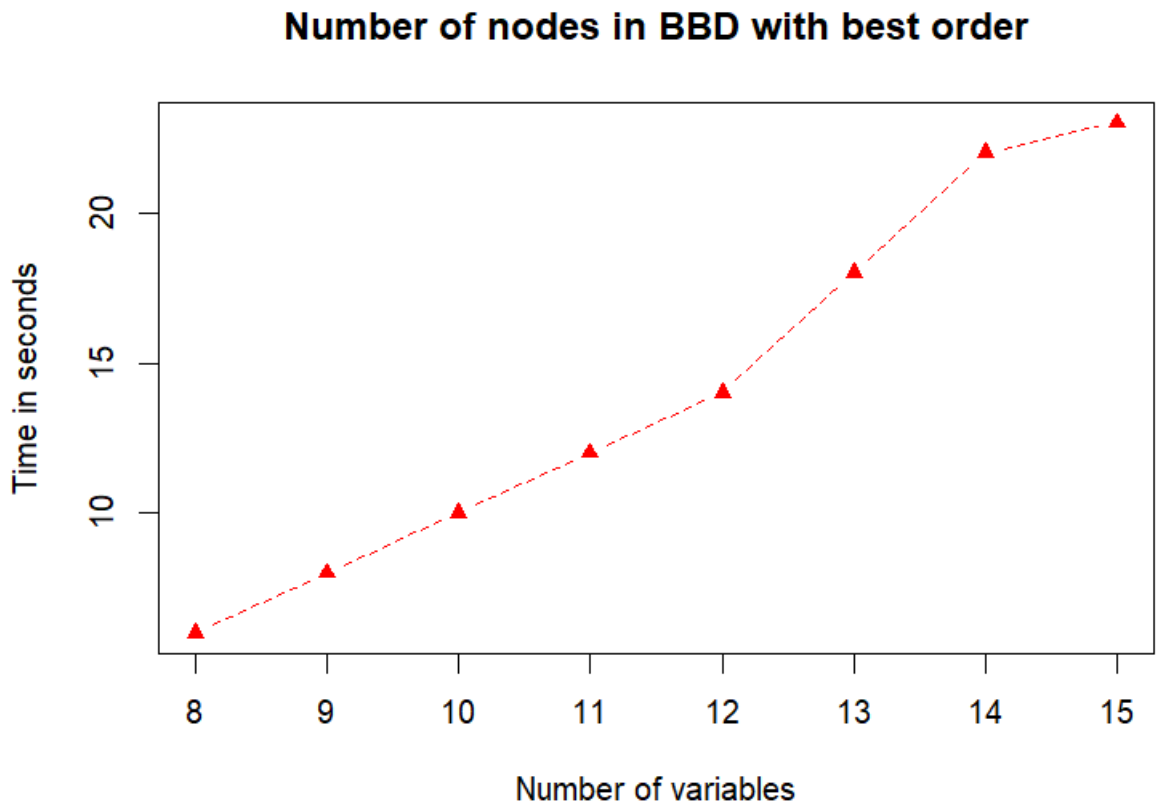


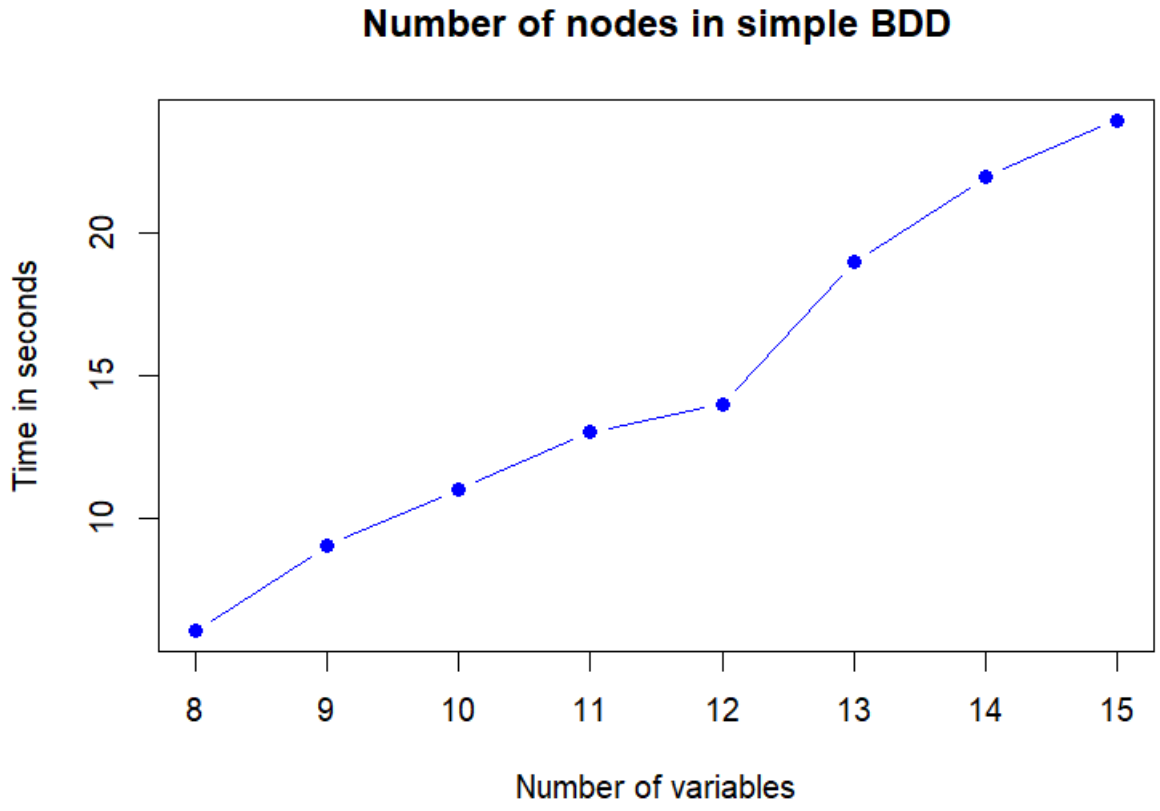
Oleksandr Vahabov, Pondelok 16:00





Oleksandr Vahabov, Pondelok 16:00





Oleksandr Vahabov, Pondelok 16:00

Všetky testy sa vykonali so 100 funkciami. Z grafov môžeme sa dozvedieť, že časová zložitosť pre BDD je O(2^n), kde n je počet premenných. Priestorová zložitosť bude tiež O(2^n), pretože závisí od množstva uzlov v BDD. Z grafu možno jednoznačne vyvodiť aj časovú zložitosť pre BDD s najlepším poradím, ktorá je O(n * 2^n), ak vezmeme do úvahy, že musíme vytvoriť n náhodných permutácií, a pre každú z nich vytvoriť novú BDD s iným poradím. Priestorová zložitosť však bude rovnaká ako pri jednoduchom BDD O(2^n), pretože každá permutácia vytvorí BDD s najväčším množstvom uzlov

Záver:

V rámci zadania sa nám podarilo úspešne implementovať dátovú štruktúru BDD, ktorá slúži na efektívnu reprezentáciu Booleovských funkcií. Implementované funkcie umožňujú vytvorenie a redukciu BDD podľa zvoleného poradia premenných, ako aj výber najefektívnejšieho poradia z viacerých možností. Správnosť implementácie bola overená testovaním – generovaním náhodných Booleovských funkcií a porovnávaním výsledkov funkcie BDD_use s očakávanými výsledkami získanými priamym vyhodnotením výrazu.

Pri testovaní sme zaznamenali výraznú mieru zredukovania BDD pri použití optimalizovaného poradia premenných a použitia hašovacej tabuľky. Redukovanie dosiahlo 99%, čo potvrdzuje efektívnosť navrhnutého riešenia.

Z pohľadu výpočtovej zložitosti sa vytváranie BDD pohybuje v závislosti od počtu premenných a zložitosti výrazu, pričom sme kládli dôraz na priebežnú redukciu počas vytvárania diagramu. Vďaka tomu je riešenie efektívnejšie aj pre väčší počet premenných. Výsledky testovania preukázali, že naša implementácia je správna a optimalizovaná vzhľadom na priestorovú aj časovú náročnosť.