Tomáš Brček, ID: 120761 FIIT STU, Bratislava Po 11.00-12.50 -2.01/a(CPUa)

Dátové štruktúry a algoritmy

Zadanie 2 – Binárne rozhodovacie diagramy

Informácie k zadaniu

Programovací jazyk: Java 19

IDE: Intellij IDEA Community Edition 2022.3.2

OS: MS Windows 11 Home / Ubuntu 22.10

CPU: 11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz, 2419 MHz, počet jadier: 4, počet logických procesorov: 8

RAM: 16 GB

Binárne rozhodovacie diagramy

Binárny rozhodovací diagram (BDD) je grafická reprezentácia logických funkcií pomocou stromu, ktorý je tvorený uzlami a hranami. Každý uzol stromu reprezentuje rozhodovanie na základe jedného vstupného bitu. Tento bit môže byť buď 0 alebo 1 a každá vetva vedúca z uzla predstavuje jednu z možností vstupu.

Listy stromu predstavujú výsledok logického rozhodovania, teda hodnotu výrazu. Napríklad, ak hovoríme o funkciách AND a OR, tak hodnota výrazu môže byť buď 0 alebo 1, teda FALSE alebo TRUE.

Použitie BDD má niekoľko výhod. Jednou z nich je, že umožňuje efektívne vykonávanie logických operácií. Napríklad, ak chceme vykonať operáciu AND alebo OR medzi dvomi logickými funkciami, môžeme jednoducho skombinovať ich BDD a vykonať operáciu na spojenom strome.

Ďalšou výhodou použitia BDD je, že umožňuje efektívne ukladanie a spracovanie veľkých množstiev logických funkcií. To je dôležité pri návrhu a optimalizácii obvodov, programov a systémov, ktoré pracujú s logickými funkciami.

BDD sa používajú v mnohých oblastiach, ako napríklad pri návrhu obvodov, pri analýze a overovaní systémov, v automatizácii návrhu programov, v kryptografii a v mnohých ďalších aplikáciách, kde je potrebné efektívne pracovať s logickými funkciami.

Moje riešenie BDD

V mojom riešení binárneho rozhodovacieho diagramu som použil tri triedy:

1. Node

```
class Node{
    private final char variable;
    private Node parent;
    private Node yesBranch;
    private Node noBranch;
    private String remaining;
    private final int level;

    public Node(char variable, int level){
        this.variable = variable;
        this.remaining = "";
        this.yesBranch = null;
        this.noBranch = null;
        this.level = level;
    }
```

V tejto triede si ukladám ako atribúty:

- **variable** – znak uložený v tomto uzle
- **parent** – ukazovateľ na predchodcu daného uzla
- **yesBranch** – ukazovateľ na nasledujúci uzol, v prípade, že vstup je 1
- **noBranch** – ukazovateľ na nasledujúci uzol, v prípade, že vstup je 0
- **remaining** – zvyšná funkcia po odstránení znaku daného uzla
- **level** – úroveň, v ktorej sa daný uzol nachádza

2. BDD

```
public class BDD {
    private final int variables_count;
    private int nodes_count;
    private int removed_nodes;
    private Node root;
    private final HashChain[] hashChains;
    public String order;

    public BDD(int variables_count) {
        this.root = null;
        this.variables_count = variables_count;
        this.nodes_count = 0;
        this.removed_nodes = 0;
        this.hashChains = new HashChain[variables_count+1];
        for(int i=0; i<variables_count+1; i++){
            this.hashChains[i] = new HashChain( bucketNum: 100);
        }
    }
```

Atribúty:

- **variables_count** – počet premenných vo funkcii
- **nodes_count** – počet vytvorených uzlov v diagrame
- **removed_nodes** – počet odstránených uzlov pri redukcii
- **root** – ukazovateľ na koreň diagramu
- **hashChains** – pole hashovacích tabuliek pre každú úroveň
- **order** – poradie premenných

3. BDDHandle

Trieda, v ktorej pracujem s binárnym rozhodovacím diagramom. V tejto triede sa nachádzajú aj implementované funkcie: BDD_create, BDD_create_with_best_order, BDD_use.

BDDHandle

Hlavná trieda implementácie môjho zadania. V tejto triede som implementoval všetky 3 potrebné funkcie, ktoré pracujú s binárnym vyhľadávacím diagramom.

V tejto triede sa nachádza aj nasledujúci blok kódu, ktorý do atribútu *permutations* uloží 1000 náhodných permutácii pre rôzne počty premenných, až po 26, čo sú všetky písmená abecedy.

```
private static final String[][] permutations;

// vytvorenie maximálne 1000 permutácii pre všetky možné počty premenných
static {
    permutations = new String[26][];
    for(int i=0; i<26; i++){
        permutations[i] = getPermutations( y: i+1);
    }
}
```

Môj program akceptuje **boolovskú funkciu v DNF tvare**, pričom premenné sú veľké tlačené písmená, postupne od A po Z. V každom súčine sú potom tieto premenné usporiadané podľa abecedy. Vo funkcii sa nenachádzajú žiadne medzery ani iné biele znaky.

Príklad funkcie, ktorú program akceptuje: !ABD+B!CD+A!C+!A!BCD

Takto zadaná funkcia bude v programe akceptovaná a spracovaná do podoby binárneho vyhľadávacieho diagramu.

Poradie premenných, ktoré je potrebné pre funkciu BDD_create je v tvare čísel (od 0 po „počet premenných“-1) oddelených znakom “- “.

Príklad poradia, ktoré program akceptuje: 2-1-3-0. Kde číslo 2 reprezentuje premennú C, číslo 1 premennú B, číslo 3 premennú D a 0 premennú A.

Príklad korektného volania funkcie BDD_create:

```
BDD b = BDD_create( bfunkcia: "!ABD+B!CD+A!C+!A!BCD",  poradie: "2-1-3-0");
```

Najdôležitejšie funkcie:

1. BDD BDD_create (String bfunkcia, String poradie)

Funkcia slúži na zostavenie redukovaného binárneho rozhodovacieho diagramu, ktorý reprezentuje zadanú Booleovskú funkciu, ktorá je zadaná ako argument funkcie (argument bfunkcia). Booleovská funkcia je poskytnutá funkcii BDD_create v tvare opísanom vyššie. Druhým argumentom je poradie premenných, ktorým sa definuje, v akom poradí sú použité jednotlivé premenné Booleovskej funkcie. Návratovou hodnotou funkcie BDD_create je ukazovateľ na zostavený a redukovaný binárny rozhodovací diagram, ktorý je reprezentovaný vlastnou štruktúrou BDD. Redukcie v mojom riešení vykonávam priebežne počas vytvárania diagramu.

Opis riešenia: V mojom riešení tejto funkcie, vkladám vytvorené a ešte nespracované uzly do radu, z ktorého ich postupne vyberám a spracovávam. Okrem toho všetky uzly vkladám do hashovacej tabuľky pre danú úroveň, čo zrýchľuje vyhľadávanie prípadných rovnakých uzlov. Pre každý uzol potom

vytvorím výraz, ktorý zostane po odstránení príslušnej premennej. Toto vykonávam v metódach *yesExpression* a *noExpression.*

```java
// metóda, ktorá z daného výrazu určí výsledný výraz pre vetvu s vstupnou hodnotou 1
public static String yesExpression(String[] expression, char c){
    String[] result = new String[expression.length];
    int i = 0;
    for (String exp : expression) {
        int j = exp.indexOf(c);     //pozrie sa, či sa zadaný znak nachádza vo výraze
        // ak nie, pridá sa tam celý výraz
        if(j == -1){
            result[i] = exp;
            i++;
        }else{
            // ak sa pred ním nachádza '!', výraz sa preskočí
            if(j!=0 && exp.charAt(j-1) == '!'){
                continue;
            // inak sa zapíše výraz bez danej premennej
            }else{
                // v prípade, ak je to len jeden znak, celá funkcia vráti 1
                if(Objects.equals(exp, Character.toString(c))){
                    return "1";
                }
                result[i] = exp.replace(Character.toString(c), replacement: "");
                i++;
            }
        }
    }
```

```java
// metóda, ktorá z daného výrazu určí výsledný výraz pre vetvu s vstupnou hodnotou 1
public static String noExpression(String[] expression, char c){
    String[] result = new String[expression.length];
    int i = 0;
    for (String exp : expression) {
        int j = exp.indexOf(c);     //pozrie sa, či sa zadaný znak nachádza vo výraze
        // ak nie, pridá sa tam celý výraz
        if(j == -1){
            result[i] = exp;
            i++;
        }else{
            // ak sa pred ním nachádza '!', výraz sa zapíše bez tejto premennej
            if(j!=0 && exp.charAt(j-1) == '!'){
                if(Objects.equals(exp, b: "!"+c)){
                    return "1";
                }
                result[i] = exp.replaceAll( regex: "!"+c, replacement: "");
                i++;
            }
        }
    }
```

Následne v týchto výrazoch odstránim duplikované podvýrazy a vrátim, buď výraz alebo priamo 0 alebo 1 podľa výsledku.

```java
if(o.length == 1){
    String[] z = onlyUnique(bfunkcia.split( regex: "\\+"));
    if(z.length == 1){
        if(Objects.equals(z[0], b: "A")){
            newBDD.getRoot().setNoBranch(no);
            newBDD.getRoot().setYesBranch(yes);
            noUsed = true;
            yesUsed = true;
        }else if(Objects.equals(z[0], b: "!A")){
            newBDD.getRoot().setNoBranch(yes);
            newBDD.getRoot().setYesBranch(no);
            noUsed = true;
            yesUsed = true;
        }
    }else{
        newBDD.getRoot().setNoBranch(yes);
        newBDD.getRoot().setYesBranch(yes);
        yesUsed = true;
    }
    if(yesUsed)
        newBDD.setNodes_count();
    if(noUsed)
        newBDD.setNodes_count();

    newBDD.setRemoved_nodes(fullBDDNodesCount(newBDD.getVariables_count()) - newBDD.getNodes_count());
    return newBDD;
}
```

V prípade, že funkcia má len jednu premennú, vytvorí sa uzol, ktorý bude zároveň koreňom stromu a jeho potomkovia, ktorý budú priamo 0 alebo 1. Nastavia sa potrebné atribúty a novovytvorený strom sa vráti pomocou kľúčového slova return.

Ak je premenných viac ako 1, vo while-cykle prechádzam všetky ešte nespracované uzly, ktoré sú uložené v rade.

```java
// while cyklus, kým neprejdeme všetky nespracované uzly
while(!nodes.isEmpty()){
    current=nodes.remove();     // vybratie prvého vloženého prvku
    int level = current.getLevel()+1;
    String[] expressions;

    expressions = current.getRemaining().split( regex: "\\+");

    // vytvorenie dvoch nových uzlov pre true a false
    Node noNode = new Node(order[level], level);
    Node yesNode = new Node(order[level], level);

    // zistenie zostávajúcich výrazov pre tieto uzly
    String noExp = noExpression(expressions, order[level-1]);
    String yesExp = yesExpression(expressions, order[level-1]);

    noNode.setRemaining(noExp);
    noNode.setParent(current);
    yesNode.setRemaining(yesExp);
    yesNode.setParent(current);
```

Následne skontrolujem, či sa takýto uzol už v danej úrovni nenachádza. Ak sa nachádza, vykoná sa redukcia:

```
// v prípade, že sa taký uzol už v tejto úrovni nachádza, prepojí sa aktuálny uzol už s vytvoreným uzlom
}else{
    current.setNoBranch(newBDD.getHashChains( level: level+1).search(noExp));
}
```

Podobne pre druhú vetvu. V prípade, že jeden uzol má obidvoch potomkov rovnakých, vykoná sa druhá redukcia, ktorej implementácia v mojom kóde je nasledujúca:

```
// ak sú uzly pre true, aj pre false rovnaké, odstráni sa tento uzol a jeho rodič sa prepojí priamo s jeho potomkom
if(current.getNoBranch() == current.getYesBranch()){
    // ak je tento uzol potomkom rodiča pre false vetvu
    if(current.getParent()!=null && current == current.getParent().getNoBranch()){
        current.getParent().setNoBranch(current.getNoBranch());
        newBDD.removeNodes_count();
    // ak je tento uzol potomkom rodiča pre false vetvu
    }else if(current.getParent()!=null){
        current.getParent().setYesBranch(current.getYesBranch());
        newBDD.removeNodes_count();
    }
}
```

Na konci sa nastaví počet odstránených uzlov a nový binárny rozhodovací diagram sa vráti pomocou kľúčového slova return.

2. BDD BDD_create_with_best_order(String bfunkcia)

Funkcia slúži na nájdenie čo najlepšieho poradia (v rámci vyskúšaných možností) premenných pre zadanú Booleovskú funkciu. Hľadanie spočíva v opakovanom volaní funkcie BDD_create, pričom sa skúšajú použiť rozličné poradia premenných.

```
// metóda, ktorá vyskúša rôzne permutácie, aby našla najlepšie poradie, pre ktoré bude počet vytvorených uzlov čo najmenší
public static BDD BDD_create_with_best_order(String bfunkcia){
    int variables_count = countUniqueChars(bfunkcia);       // spočítanie počtu premenných
    int best = Integer.MAX_VALUE;
    BDD best_BDD = null;

    // pole maximálne 1000 náhodných permutácii poradí prvkov
    String[] perms = permutations[variables_count-1];

    for(String p : perms){
        BDD newBDD = BDD_create(bfunkcia, p);
        if(newBDD.getNodes_count() < best){
            best = newBDD.getNodes_count();
            best_BDD = newBDD;
            newBDD.order = p;
        }
    }

    return best_BDD;
}
```

Na začiatku tejto funkcie si spočítam počet premenných v danej booleovskej funkcii. Do premennej *perms* si uložím 1000 náhodne vygenerovaných permutácii pre daný počet prvkov. Tieto permutácie boli vygenerované pri prvom volaní triedy BDDHandle. Následne prejdem všetky tieto permutácie, pričom pre každú vytvorím strom a zistím počet uzlov. Ak je počet uzlov menší ako doteraz najmenší počet, tento nový strom sa uloží do premennej *best_BDD* a na konci je vrátený touto funkciou najlepší diagram z 1000 vyskúšaných.

3. char BDD_use(BDD bdd, String vstupy)

Funkcia slúži na použitie BDD pre zadanú kombináciu hodnôt vstupných premenných Booleovskej funkcie a zistenie výsledku Booleovskej funkcie pre túto kombináciu vstupných premenných. V rámci tejto funkcie prechádzame diagramom smerom od koreňa po list takou cestou, ktorú určuje práve zadaná kombinácia hodnôt vstupných premenných.

```
// metóda, ktorá na základe vstupu zistí výslednú hodnotu výrazu
public static char BDD_use(BDD bdd, String vstupy){
    // uloženie všetkých premenných aj s ich hodnotou, buď 1-true alebo 0-false
    Map<Character, Integer> map = new HashMap<>();

    if(vstupy.length() > bdd.getVariables_count()){
        return (char) -1;
    }

    for(int i=0; i<vstupy.length(); i++){
        map.put((char) ('A'+i), Integer.parseInt(String.valueOf(vstupy.charAt(i))));
    }

    // postupné prejdenie všetkých uzlov od koreňa
    Node current = bdd.getRoot();
    int j = 0;
    while(j!=vstupy.length() && current.getVariable() != '0' && current.getVariable() != '1'){
        // na základe hodnoty premennej sa určí nasledujúca vetva
        if(map.get(current.getVariable()) == 0){
            current = current.getNoBranch();
        }else if(map.get(current.getVariable()) == 1){
            current = current.getYesBranch();
        }
        j++;
    }
    char var = current.getVariable();       // zistenie výslednej hodnoty
    if(var == '0'){
        return var;
    }else if(var == '1'){
        return var;
    }else{
        return (char)-1;        // v prípade iného výsledku, ako 0 alebo 1, metóda vráti -1
    }
}
```

Funkcia najprv overí dĺžku vstupu, či nie je dlhší ako počet premenných. V tomto prípade vráti znak ‘-1‘. Následne do premennej *map* uloží postupne hodnoty pre všetky premenné, aby sa v nich dalo ľahšie vyhľadávať. Vo while-cykle postupne prejde celý diagram takou cestou, akú určuje vstup. Na konci vráti 0 alebo 1 podľa hodnoty uzla, na ktorý ukazuje uzol s poslednou premennou. V prípade, že sa niečo nepodarí, vráti ‘-1‘.

Testovanie

Testovanie môjho zadania vykonávam postupným vytváraním binárnych vyhľadávacích diagramov podľa počtu premenných a to postupne od 1 po 18 premenných. Pre každý počet premenných potom vygenerujem 100 rôznych náhodných booleovských funkcii, ktoré budem posielať do funkcii *BDD_create* a *BDD_create_with_best_order*.

Zároveň si pre všetky počty premenných vygenerujem všetky možné vstupy, ktoré budem posielať do funkcie *BDD_use* na zistenie výslednej hodnoty funkcie.

Počas vytvárania diagramov meriam čas potrebný na vytvorenie týchto diagramov. Všetky výsledky priebežne ukladám do polí a na záver ich vydelím počtom booleovských funkcii pre daný počet premenných a na záver všetky priemery vypíšem do konzoly.

Počas behu programu tiež kontrolujem správnosť môjho riešenia tým, že kontrolujem výstup, ktorý dostanem z môjho diagramu s očakávaným výstupom, ktorý zisťujem v metóde *expectedOutput*:

```
// metóda, ktorá na základe funkcie a zadaného vstupu zisti očakávaný výstup
public static char expectedOutput(String bfunkcia, String input){
    // uloženie všetkých premenných aj s ich hodnotou, buď 1-true alebo 0-false
    Map<Character, Integer> map = new HashMap<>();

    for(int i=0; i<input.length(); i++){
        map.put((char) ('A'+i), Integer.parseInt(String.valueOf(input.charAt(i))));
    }

    // postupné nahradenie všetkých premenných ich hodnotami
    for(int j=0; j<input.length(); j++){
        bfunkcia = bfunkcia.replace((char)('A'+j), input.charAt(j));
    }

    // nahradenie v prípade, že je pred znakom '!'
    for(int x=0; x<bfunkcia.length(); x++){
        if(bfunkcia.charAt(x) == '!' && (x+1)!=bfunkcia.length()){
            bfunkcia = bfunkcia.substring(0, x) + bfunkcia.substring( beginIndex: x + 1);
            if(bfunkcia.charAt(x) == '0'){
                bfunkcia = bfunkcia.substring(0, x) + "1" + bfunkcia.substring( beginIndex: x+1);
            }else{
                bfunkcia = bfunkcia.substring(0, x) + "0" + bfunkcia.substring( beginIndex: x + 1);
            }
        }
    }

    // rozdelenie výrazu na podvýrazy
    String[] vyrazy = bfunkcia.split( regex: "\\+");
    for(String vyraz : vyrazy){
        boolean only1 = true;
        // ak je aspoň jeden podvýraz tvorený samými 1, výstup je 1, inak 0
        for(int k=0; k<vyraz.length(); k++){
            if(vyraz.charAt(k) == '0'){
                only1 = false;
                break;
            }
        }
        if(only1){
            return '1';
        }
    }
    return '0';
}
```

Funkcia postupne nahradí všetky premenné hodnotou získanou z argumentu *input.* Následne si tento výraz rozdelí na podvýrazy podľa znaku ‘+‘ a pozrie sa, či aspoň jeden z podvýrazov tvoria samé jednotky. Ak áno, očakávaný výsledok je 1. Ak nie, očakávaný výsledok bude 0.

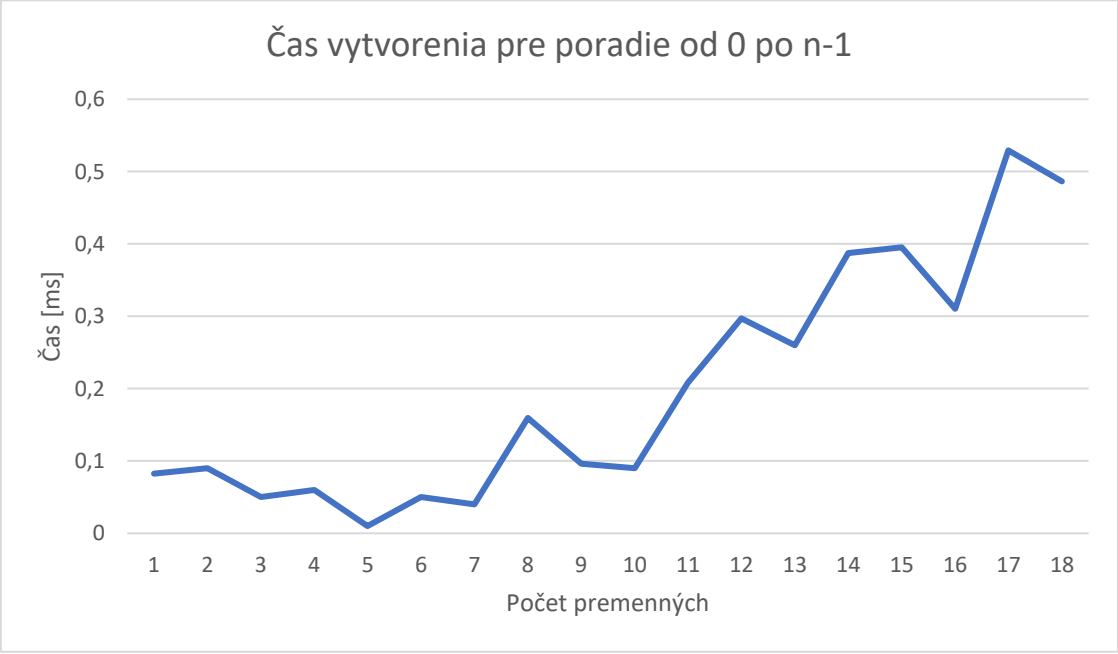
Výsledky testovania

Výsledky mojich testov sú zobrazené v tabuľke nižšie:

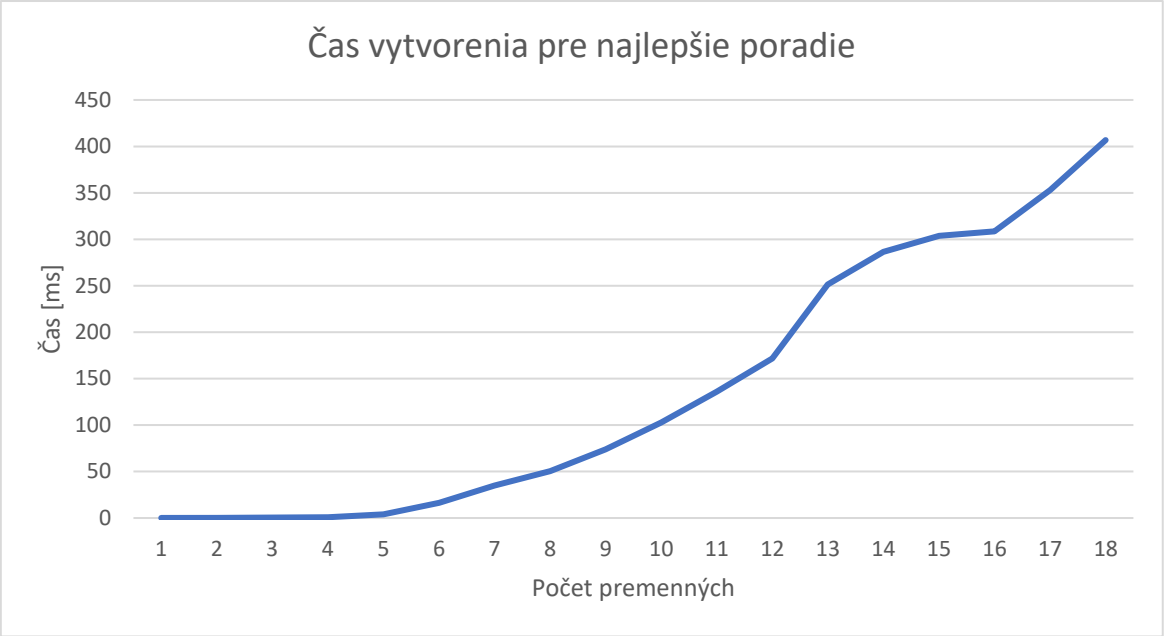
Počet premenných	Percentuálna miera zredukovania		Čas vytvorenia [ms]		Čas použitia [ms]	
	od 0 po n-1	best order	od 0 po n-1	best order	od 0 po n-1	best order
1	17,33333%	17,33333%	0,082266	0,079988	0,113404	0,039978
2	54,14286%	57,28571%	0,089991	0,090026	0,059982	0,020014
3	72,06667%	77,20000%	0,050019	0,325085	0,009998	0,029959
4	79,22581%	84,93548%	0,060113	0,939203	0,120064	0,050007
5	83,79365%	89,00000%	0,009992	3,87132	0,180296	0,149927
6	87,84252%	92,26772%	0,049945	16,332155	0,121042	0,148861
7	89,92549%	93,63922%	0,040031	34,907984	0,375884	0,358155
8	92,35421%	95,16047%	0,159671	50,37842	0,75578	0,775635
9	94,41056%	96,29130%	0,096033	73,854703	1,846198	1,851289
10	96,35125%	97,32975%	0,090111	102,811043	4,087597	3,911845
11	97,63175%	98,13724%	0,208309	135,864708	8,685338	8,529375
12	98,43743%	98,77292%	0,29704	171,909605	19,734861	19,893836
13	98,97479%	99,11988%	0,26009	251,456931	53,147119	52,60827
14	99,38414%	99,47420%	0,387259	286,558756	111,408161	108,387788
15	99,65756%	99,69953%	0,395414	303,658371	238,921102	237,345946
16	99,81973%	99,83518%	0,310263	308,576017	510,714476	510,566735
17	99,89477%	99,90112%	0,529334	352,87863	1336,290911	1347,625964
18	99,94005%	99,94451%	0,48663	406,796069	2605,147565	2611,808285

Grafy

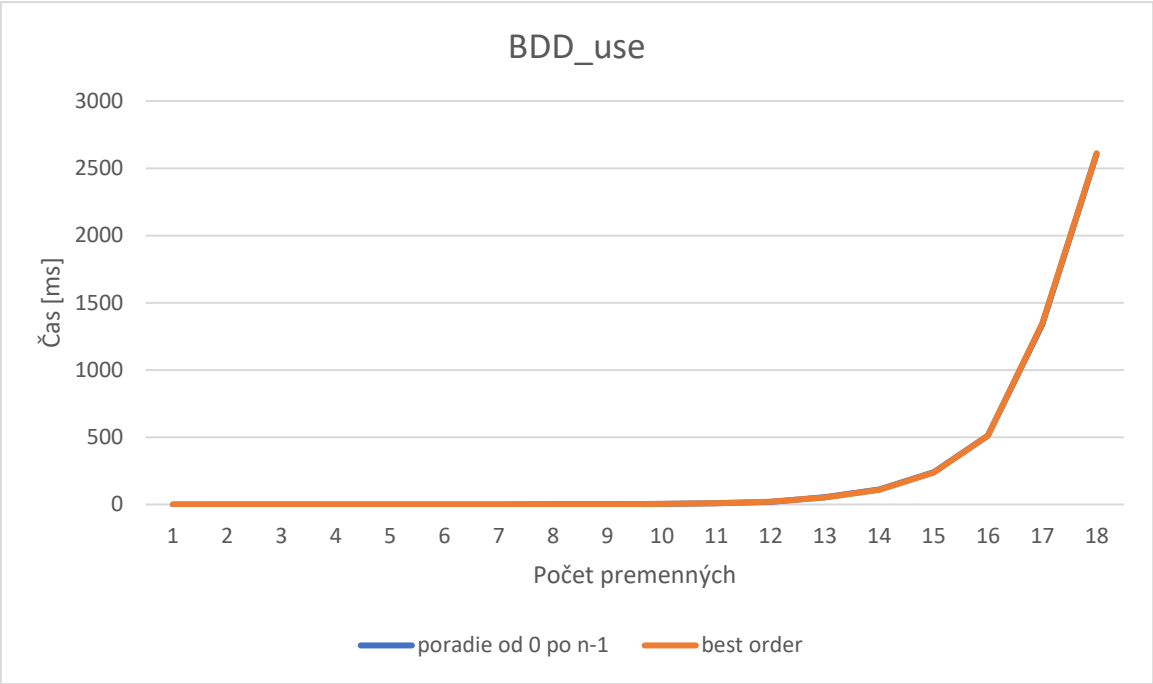
Na tomto grafe je zobrazená závislosť percentuálnej miery zredukovania od počtu premenných v booleovskej funkcii. Môžeme vidieť, že čím je viac premenných, tým je táto miera vyššia, až sa postupne blíži k hranici 100%. Ďalej môžeme vidieť rozdiel v tejto funkcii pre poradie od 0 po n-1 oproti poradiu, ktoré dostaneme volaním funkcie BDD_create_with_best_order. Z grafu je zrejmé, že pre nižší počet premenných, je rozdiel medzi týmito poradiami zjavný, no postupne pre väčší počet premenných sa tento rozdiel pomaly stráca.



Tento graf zobrazuje závislosť času potrebného na vytvorenie BDD od počtu premenných pre poradie od 0 po n-1.



Tento graf zobrazuje závislosť času potrebného na vytvorenie BDD od počtu premenných pre najlepšie poradie premenných.



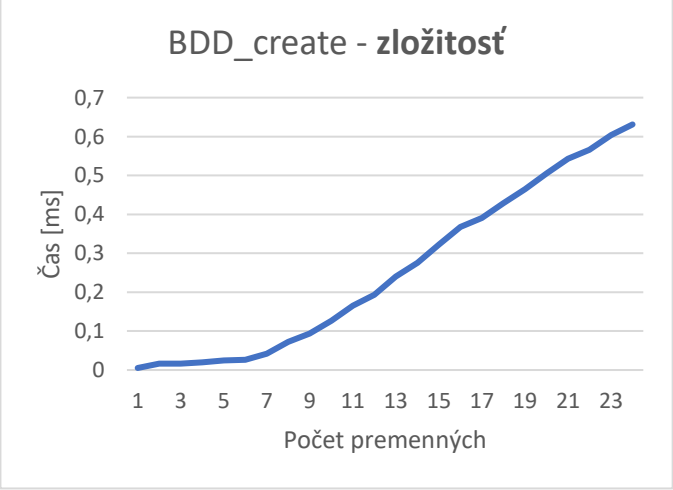
Na tomto grafe je vyjadrená závislosť času potrebného na použitie BDD od počtu premenných. Z grafu môžeme vidieť, že grafy pre najlepšie poradie a pre poradie od 0 po n-1 sú takmer identické. Prekrývajú sa a nevidíme medzi nimi žiadny veľký rozdiel. Vidíme, že tento čas rastie exponenciálne.

Zložitosť

1. Výpočtová časová zložitosť

Počet premenných	Čas [ms]	
	BDD_create	BDD_create_with_best_order
1	0,005204	0,022512
2	0,016337	0,061777
3	0,01643	0,110232
4	0,019349	0,511518
5	0,024221	1,72688
6	0,026389	2,442402
7	0,041607	4,112352
8	0,072499	5,659494
9	0,093914	8,324377
10	0,126911	11,356473
11	0,165607	14,876857
12	0,193317	19,542795
13	0,240305	25,933852
14	0,275817	30,801279
15	0,322814	36,42613
16	0,367492	41,665979
17	0,391003	47,285154
18	0,428693	48,444805
19	0,464765	52,747681
20	0,505054	58,340921
21	0,542718	63,458301
22	0,565975	66,852192
23	0,603499	72,635222
24	0,631071	71,695735

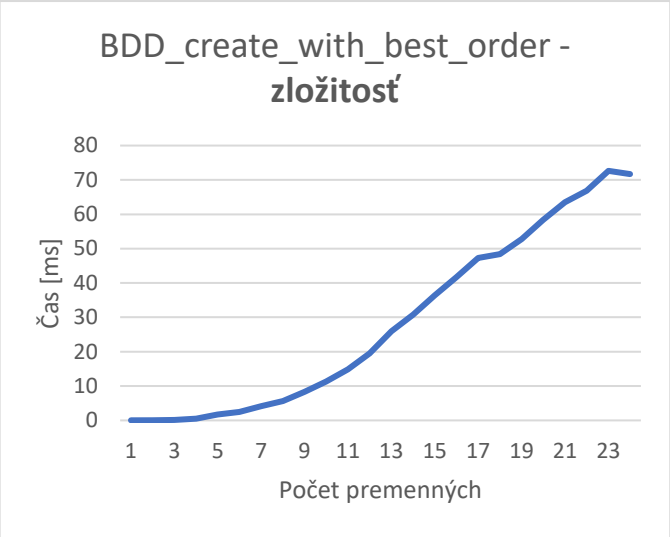
a) BDD_create



Z grafu môžeme vidieť, že závislosť medzi časom potrebným na vytvorenie BDD a počtom premenných môže byť aproximovaná kvadratickou funkciou.

Takže zložitosť algoritmu pre vytvorenie redukovaného BDD je približne **$O(N^2)$**, kde N je počet premenných.

b) BDD_create_with_best_order

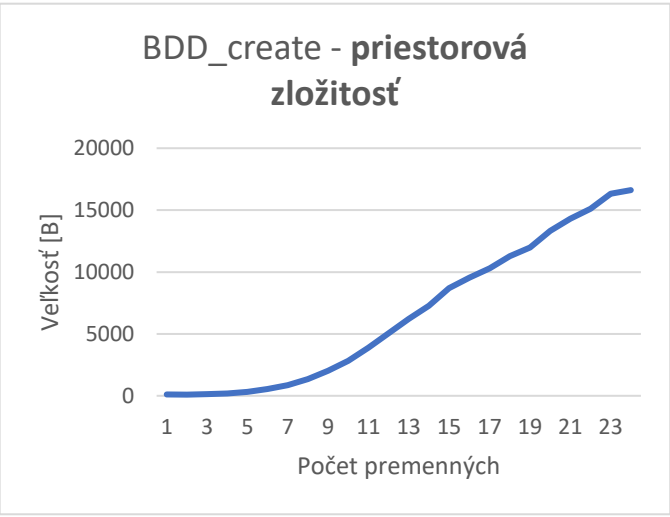


Z grafu môžeme vidieť, že závislosť medzi časom potrebným na vytvorenie BDD a počtom premenných môže byť aproximovaná kvadratickou funkciou.

Takže zložitosť algoritmu pre vytvorenie redukovaného BDD je približne **O(N²)**, kde N je počet premenných.

2. Priestorová zložitosť

a) BDD_create



Z grafu môžeme vidieť, že závislosť medzi veľkosťou potrebnou na uloženie BDD a počtom premenných môže byť aproximovaná kvadratickou funkciou.

Takže zložitosť algoritmu pre vytvorenie redukovaného BDD je približne **O(N²)**, kde N je počet premenných.

b) BDD_create_with_best_order

Z grafu môžeme vidieť, že závislosť medzi veľkosťou potrebnou na uloženie BDD a počtom premenných môže byť aproximovaná kvadratickou funkciou.

Takže zložitosť algoritmu pre vytvorenie redukovaného BDD je približne **O(N²)**, kde N je počet premenných.

Vyhodnotenie

V tomto zadaní bolo našou úlohou implementovať 3 funkcie pre prácu s binárnymi rozhodovacími diagramami (BDD) so zameraním na využitie pre reprezentáciu Booleovských funkcii. Týmito funkciami sú: BDD_create, BDD_create_with_best_order a BDD_use.

V mojom riešení som implementoval všetky tri funkcie, pričom som vytvoril aj testovací súbor, v ktorom som náhodne generoval booleovské funkcie v DNF tvare a tiež som si vytvoril funkciu na určenie očakávaného výstupu na základe funkcie a daného vstupu.

Moje riešenie som overil pre počty premenných od 1 do 18, pričom pre každý počet som vygeneroval 100 náhodných booleovských funkcii, z ktorých som potom vypočítal priemer a tieto dáta som zapísal do tabuľky vyššie.

Následne som z tejto tabuľky vytvoril grafy, v ktorých som si všímal rôzne závislosti a porovnával som najmä diagramy vytvorené funkciou BDD_create_with_best_order s diagramami vytvorenými v poradí premenných od 0 po n-1. Z grafov je možné vidieť, že najmä čo sa týka percentuálnej miery zredukovania sú diagramy s najlepším nájdeným poradím lepšie ako diagramy vytvorené v poradí od 0 po n-1. Pre väčší počet premenných sa však tento rozdiel zmenšoval, a takmer zanikol.

Keď sa však pozrieme na čas vytvorenia diagramu, tak BDD s poradím od 0 po n-1 sa vytvárali oveľa rýchlejšie, na druhej strane majú však väčší počet uzlov. Ďalší problém pre nájdenie najlepšieho poradia premenných je, že je veľmi časovo náročné nájsť najlepšie poradie, keďže by sme museli vyskúšať všetkých n! permutácii, čo je pre väčší počet premenných veľmi náročné. Preto som moje riešenie obmedzil na 1000 náhodných permutácii, z ktorých sa vyberie tá najlepšia.

Aj napriek tomu je BDD s čo najlepším poradím prvkov lepšie, keďže obsahuje menší počet uzlov.