Peter Brenkus, ID: 120762

**FAKULTA INFORMATIKY A INFORMAČNÝCH TECHNOLÓGIÍ**
**SLOVENSKÁ TECHNICKÁ UNIVERZITA**

Ikovičova 2, 842 16 Bratislava 4

# 2022/2023
# Dátové štruktúry a algoritmy
# Zadanie č.2

# Úvod a zadanie

Účelom tohto článku je zdokumentovať moju prácu na 2. zadaní v predmete Dátové štruktúry a algoritmy. Zdrojový kód som písal v jazyku Java, v JetBrains IntelliJ IDEA 2022.3.3 na zariadení MacBook Pro 2020 13“ s operačným systémom macOS Big Sur 11.2.3, štvorjadrovým procesorom Intel Core i7 (2,3GHz) a 16GB RAM.

Zadaním bolo implementovať binárny rozhodovací diagram (BDD) s redukciami typu I a S, ktoré sa dejú počas vytvárania diagramu.

## Obsah



# Zdrojový kód

## Trieda bdd a trieda node

Tieto triedy aj s atribútmi sú zadefinované priamo v zadaní, okrem toho čo tam musí byť som nič ďalšie nepridával.

Bdd obsahuje: ukazovateľ na koreň diagramu, počet uzlov, počet a poradie premenných boolovskej funkcie, ktorú reprezentuje.

Node obsahuje: ukazovatele na deti, funkciu, ktorú reprezentuje a premennú, podľa ktorej sa jeho deti delia.

## BDDcreate()

Prvú funkciu, ktorú bolo treba implementovať som kvôli zachovaniu formátu funkcii zo zadania vyriešil takto:

1. Do hashmapy pridá uzly 0 a 1
2. Vytvorí objekt bdd
3. Zavolá funckiu **BDDaddnodes()**
4. Pridá stromu ostatné atribúty
5. Vracia ukazovateľ na strom

```
public static bdd BDDcreate(String function, String order){
    nodeMap.put("1", new node( function: "1",  variable: '1'));
    nodeMap.put("0", new node( function: "0",  variable: '0'));
    bdd tree = new bdd(order);
    tree.root = BDDaddNodes(function, order,  index: 0);
    tree.numberOfVariables = order.length();
    if ( tree.root.left != null && tree.root.right != null
    if (tree.root.function == "0"){...}
    else if (tree.root.function == "1"){...}
    else{...}
    return tree;
}
```

Funkcia **BDDaddnodes()** rekurzívne pridáva uzly do stromu, ich bool funkcie im prideľuje funkcia **split()** triedy *SplitFunction.* V tejto funkcii sa teda dejú aj redukcie – typu I pomocou hashmapy, v ktorej mám uložené všetky unikátne uzly a typu S pomocou while cyklu, v ktorom sa opakovane volá funkcia split, pokým nie sú bool funkcie ňou vracané rozdielne, alebo konečné (0-1).

Funkcia **split()** vracia dva stringy – jeden pre ľavé dieťa a druhý pre pravé – ktoré získa rozdelením vstupnej bool funkcie podľa vstupnej premennej. Bool funkciu najprv rozdelí podľa charakterov '+' na jednotlivé logické výrazy a potom z nich odstraňuje danú premennú a pridáva ich do korešpodujúcich zoznamov, ktoré na konci naspäť spája pomocou '+' charakteru. Táto funkcia navyše sleduje a opravuje prípadnú duplicitu logických výrazov v bool funkcii, automatickú konečnú hodnotu bool funkcii (napr. 1+AB = 1, A+!A = 1, alebo AB!A = 0,...) a či návratové hodnoty nie sú rovnaké po poprehadzovaní výrazov ( napr. AB+!A!B = !A!B+AB). O tieto úpravy sa starajú pomocné triedy : *expressionChecker*, *expressionModifier* a *equalExpressionFinder.*

```
public static node BDDaddNodes(String function, String order, int index){
    if (function.equals("1")){...}
    if (index == order.length()){...}
    else {
        if (nodeMap.containsKey(function)){
            return nodeMap.get(function);
        }
        else {
            nodeMap.put(function, new node(function, order.charAt(index)));
            String[]next = {" ", " "};
            while (next[0].equals(next[1])){
                next = SplitFunction.split(function, order.charAt(index));
                if (!next[0].equals(next[1])){
                    break;
                }
                if (next[0].equals("1")){...}
                else if (next[0].equals("0")){...}
                else  {...}

            }
            nodeMap.get(function).left = BDDaddNodes(next[0], order,  index: index + 1);
            nodeMap.get(function).right = BDDaddNodes(next[1], order,  index: index + 1);
            if (nodeMap.get(function).left == nodeMap.get(function).right){
                node temp = nodeMap.get(function);
                nodeMap.get(function).function = nodeMap.get(function).left.function;
                nodeMap.get(function).left = null;
                nodeMap.get(function).right = null;
                nodeMap.remove(temp);
            }
            return nodeMap.get(function);
```

```
public static String[] split(String function, char variable) {
    String[] split = function.split( regex: "\\+");
    ArrayList<String> positive = new ArrayList<~>();
    ArrayList<String> negative = new ArrayList<~>();
    for (String i: split){
        if (i.contains("!" + variable)){...}
        else if (i.contains(Character.toString( variable))){...}
        else {...}
    }
    String positiv = String.join( delimiter: "+", positive);
    positiv = expressionModifier.modify(positiv);
    String negativ = String.join( delimiter: "+", negative);
    negativ = expressionModifier.modify(negativ);
    if (positiv.length() == negativ.length()){
        return equalExpressionFinder.find(negativ, positiv);
    }
    else {
        return new String[]{negativ, positiv};
```

## BDDcreate_with_best_order()

Táto funkcia hľadá najvýhodnejšie poradie premenných na zostrojenie BDD pre danú bool funkciu. Najprv z funkcie vybere všetky unikátne premenné pomocou funkcie **get()** triedy *getVariables* a vyrobí toľko rôznych iných poradí týchto premenných, koľko ich je (tj pre premenné ABCD urobí štyri rôzne poradia) pomocou funkcie **shuffle()** triedy *orderShuffle*.

Následne zavolá **BDDcreate()** pre každé poradie a uloží si strom s najmenším počtom uzlov.

```
public static bdd BDDcreate_with_best_order(String function){
    bdd tree = new bdd();
    bdd demoTree;
    String variables = getVariables.get(function);
    String[] orders = orderShuffle.shuffle(variables);
    double minNodes = Double.POSITIVE_INFINITY;
    for (String i : orders){
        demoTree = BDDcreate(function, i);
        if (demoTree.numberOfNodes < minNodes){
            minNodes = demoTree.numberOfNodes;
            tree = demoTree;
        }
        nodeMap.clear();
    }
    return tree;
}
```

## BDD_use()

Táto funkcia postupne prechádza BDD pomocou ukazovatela na uzol stromu. Ak je charakter na momentálnom indexe vstupu 0 a zároveň sedí aj premenná, podľa ktorej uzol delí svoje deti, ukazovateľ sa posunie na ľavé dieťa, ak je 1 a premenná sedí, posunie sa doprava a keď je hocičo iné na vstupe, vráti -1.

Túto funkciu som pozmenil, v zadaní je napísané, že má vracať char, no tiež je tam napísané, že má vracať -1 pri chybnom vstup, čo už je char[]. Keďže ostatné návratové hodnoty majú tiež byť len čísla, moja funcia **BDD_use()** vracia int.

```
public static int BDD_use(bdd tree, String input){
    node current = tree.root;
    int index = 0;
    while (current != null){
        if (input.charAt(index) == '0'){
            if (current.variable == tree.order.toCharArray()[index]) {
                current = current.left;
            }
        }
        else if (input.charAt(index) == '1') {
            if (current.variable == tree.order.toCharArray()[index]) {
                current = current.right;
            }
        }
        else {
            System.out.println("Invalid input");
            return -1;
        }
        if (current.function.equals("0") || current.function.equals("1")){
            return Integer.parseInt(current.function);
        }
        index++;
    }
    return Integer.parseInt(current.function);
```

# Testovanie

Testoval som dve veci:

1. Správnosť mojich BDD
2. Percentuálnu mieru redukcie a časovú zložitosť **BDDcreate()** a **BDDcreate_with_best_order()**

Testoval som na 100 náhodne generovaných bool funkciách v DNF tvare s náhodnymi dĺžkami. Náhodne generované bool funkcie som si uložil do textových súborov a pri testoch som ich čítal z týchto súborov, aby som do testovania nezahŕňal aj čas potrebný na generovanie funkcií. Pre účely testovania som vytvoril 11 súborov so 100 funkciami od 8 rôznych premenných až po 18.

## Správnosť binárnych rozhodovacích diagramov

Tento scenár sa odohráva tak, že sa pre 100 bool funkcii po jednom vytvorí BDD pomocou funkcie **BDDcreate()** a následne sa pre všetky možnosti vstupných premenných zavolá funkcia **BDDuse()** a funkcia **evaluate()** triedy *expressionEvaluator*.

```
public static void testSpravnosti(int numberOfVariables){
    try{
        Scanner scanner = new Scanner(new File( pathname: "DNF"+numberOfVariables+".txt"));
        List<String> list = DNFgenerator.generate(numberOfVariables);
        while (scanner.hasNextLine()){
            bdd tree;
            String DNF = scanner.nextLine();
            tree = BDDcreate(DNF, getVariables.get(DNF));
            for (String j : list){
                if (BDD_use(tree, j) != expressionEvaluator.evaluate(DNF, j)){
                    System.out.println("Error");
                }
            }
            nodeMap.clear();
        }
    }
    catch (Exception e){
        System.out.println("File error");
    }
}
```

Funkcia **evaluate()** dostane na vstupe DNF a tú istú konfiguráciu premenných ako **BDDuse()**, následne v DNF vymení písmená za 0 a 1 podľa danej konfigurácie a ak nejaký výraz v DNF obsahuje iba samé 1 – čiže je splnený – vráti 1, ak nie vráti 0. Príklad:

1. Na vstupe dostane: AB+!AB, 01
2. DNF zmení na 01+11
3. Prejde DNF a nájde že druhý výraz je v tejto konfigurácii pravdivý – vracia 1

```
public static int evaluate(String dnf, String number){
    char[] orderArr = getVariables.get(dnf).toCharArray();
    char[] numberArr = number.toCharArray();
    for (int i = 0; i < number.length(); i++){
        if (dnf.contains("!" + orderArr[i])){
            if (numberArr[i] == '0'){...}
            else {...}
        }
        if (dnf.contains(Character.toString(orderArr[i]))){...}
    }
    String[] dnfSplit = dnf.split( regex: "\\+");
    boolean allOnes = false;
    for (String i: dnfSplit){
        if (!i.contains("0")){
            allOnes = true;
            break;
        }
    }
    if (allOnes){
        return 1;
    }
    else{...}
}
```

Ak sa návratová hodnota funkcie **BDDuse()** nerovná návratovej hodnote **evaluate()**, vypíše sa chybové hlásenie, ak nie program testuje ďalej.

Tento test som urobil so 100 funkciami o 8 až 17 premenných, nebola nájdená ani jedna chyba. Na datasete s 18 premennými som to už netestoval, keďže počet všetkých možných konfigurácii tam je $2^{18}$, a pre 100 stromov a dve funkcie, ktoré každú možnosť vyhodnocujú to je 2.100. $2^{18}$ operácii, a to by už trvalo rádovo v desiatkach hodín. Pre ilustráciu, test so 100 funkciami o 17tich premenných trval okolo 50 minút.

## Trieda DNFgenerator a jej funkcie

Zoznam všetkých možných konfigurácii premenných tvorí pomocná funkcia **generate()** triedy *DNFgenerator*, ktorá sa používa aj pri náhodnej tvorbe bool funkcii v DNF tvare.

Funkcia **generate()** vytvorí pre zadaný počet premenných zoznam všetkých možných kombinácii 1 a 0. To znamená že napríklad pre 3 premenné vráti zoznam: [000, 001, 010, 011, 100, 101, 110, 111].

Pri tvorení bool funkcii na testovanie sa pomocou funkcie **generateAll()** dotvoria aj všetky ostatné kratšie kombinácie 1 a 0, a následne sa vyberie náhodný počet týchto konfigurácii do finálneho zoznamu a jednotky a nuly sa vymenia za premenné vo forme písmen a ich negované tvary. O tieto operácie sa stará niekoľko ďalších funkcii z triedy *DNFgenerator*. Krátky príklad ako to celé vyzerá:

1. Zvolím počet premenných, napríklad 3
2. Vytvorí sa zoznam všetkých kombinácii pre 3 premenné: 000, 001,… 111
3. Vytvorí sa aj zoznam všetkých kratších kombinácii: 00, 01, 10, 11, 1, 0
4. Náhodne sa vyberú nejaké kombinácie, napr: 000, 011, 10, 1
5. Nuly a jednotky sa zmenia na premenné a spoja do DNF: !A!B!C+!ABC+B!C+A

Tento postup som zvolil najmä preto, že môžem recyklovať funkcie: **generate()** používam aj pri tvorení bool funkcii aj pri tvorení zoznamu včetkých konfigurácii pri testovaní správnosti.

Ďalej ešte stojí za zmienku fakt, že pri tvorení funkcii majú najväčšiu šancu na výber najdlhšie konfigurácie a najmenšiu úplne najkratšie (1 a 0). Je to jednoducho preto, že tých najdlhších je vždy najviac, preto je aj väčšia pravdepodobnosť že nejaká z nich bude vybraná.

```java
public static List<String> generateAll(int numberOfVariables){
    List<String> finalList = new ArrayList<~>();
    for(int i = numberOfVariables; i > 0; i-- ){
        List<String> part = generate(i);
        for (String j : part){
            finalList.add(j);
        }
    }
    return finalList;
}
```

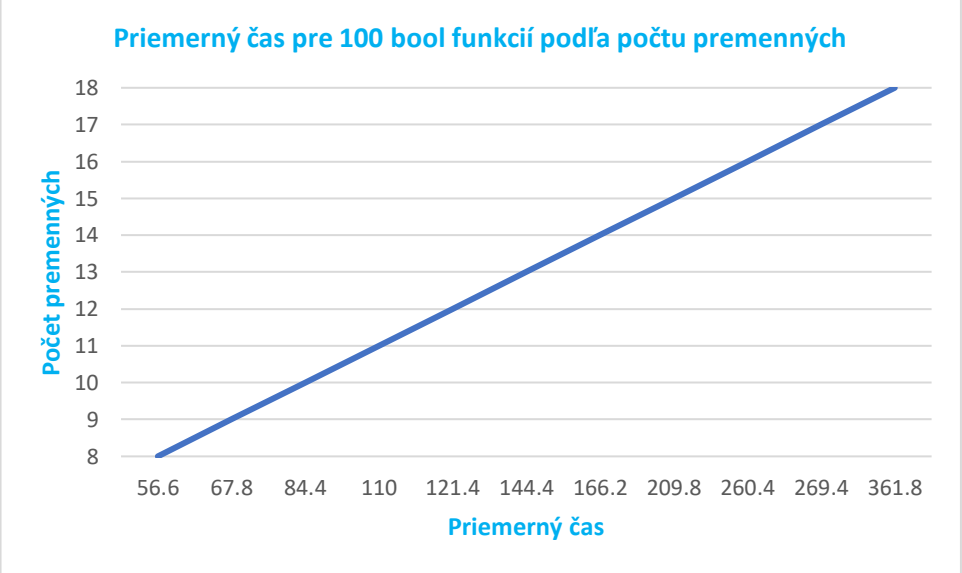
## Miera redukcie a časová zložitosť

Tento scenár sa odohráva pomocou funkcie **testTime()**. Pre každú zo 100 bool funkcii sa vytvorí BDD pomocou **BDDcreate()** aj pomocou **BDDcreate_with_best_order()**. Do polí sa zapíšu jednotlivé počty uzlov oboch týchto stromov a zmerá sa percentuálna redukcia funkcie **BDDcreate()** oproti neredukovanému stromu (t.j. $2^{n+1}$-1, kde n = počet premenných funkcie) a percentuálna redukcia funkcie **BDDcreate_with_best_order()** oproti funkcii **BDDcreate()**, t.j. ešte o koľko lepšia je.

Na konci sa všetky informácie vypíšu do terminálu. Okrem toho sa ešte vypíšu priemery počtu uzlov oboch funkcií a priemery jednotlivých redukcii. Funkcia **testTime()** ako jej meno napovedá, ešte meria aj čas, za ktorý sa daných 100 funkcii spracuje na stromy. Časovú zložitosť som meral podobne ako v prvom zadaní – funkciu som nechal zbehnúť 5 krát a tieto hodnoty som spriemeroval.

| *Počet premenných* | *Maximálny počet uzlov* | *Priemerný počet uzlov BDDcreate()* | *Priemerný počet uzlov BDDcreate_wbo()* | *Priemerná redukcia BDDcreate()* | *Priemerná redukcia BDDcreate_wbo()* |
|---|---|---|---|---|---|
| 8 | 511 | 25 | 22 | 95.1% | 9.6% |
| 9 | 1 023 | 36 | 31 | 96.5% | 12% |
| 10 | 2 047 | 39 | 34 | 98.1% | 11.7% |
| 11 | 4 095 | 51 | 45 | 98.7% | 10.8% |
| 12 | 8 191 | 55 | 48 | 99.3% | 11% |
| 13 | 16 383 | 69 | 59 | 99.6% | 13% |
| 14 | 32 767 | 75 | 65 | 99.8% | 11.3% |
| 15 | 65 535 | 90 | 77 | 99.9% | 12.6% |
| 16 | 131 071 | 109 | 93 | 99.92% | 12.9% |
| 17 | 262 143 | 103 | 90 | 99.96% | 11.2% |
| 18 | 524 287 | 144 | 126 | 99.97% | 11.5% |

| Počet premenných | Beh 1 | Beh 2 | Beh 3 | Beh 4 | Beh 5 | Priemerný čas |
|---|---|---|---|---|---|---|
| **8** | 51 | 58 | 63 | 46 | 65 | 56.6 |
| **9** | 68 | 61 | 63 | 70 | 77 | 67.8 |
| **10** | 80 | 106 | 68 | 75 | 93 | 84.4 |
| **11** | 111 | 101 | 114 | 114 | 110 | 110 |
| **12** | 125 | 115 | 122 | 126 | 119 | 121.4 |
| **13** | 148 | 143 | 147 | 144 | 140 | 144.4 |
| **14** | 164 | 172 | 162 | 170 | 163 | 166.2 |
| **15** | 210 | 207 | 212 | 207 | 213 | 209.8 |
| **16** | 259 | 260 | 258 | 264 | 261 | 260.4 |
| **17** | 268 | 272 | 271 | 267 | 269 | 269.4 |
| **18** | 374 | 289 | 381 | 383 | 382 | 361.8 |



# Záver

V tejto časti zhrniem poznatky z testov, ktoré som vykonal na mojich algoritmoch.

Z testu správnosti vyplýva, že môj program funguje 100%. Priestorová zložitosť môjho programu v O notácii je $2^n$ +2, kde n = počet premenných bool funkcie.

Z nameraných údajov vidíme, že s nárastom počtu rôznych premenných sa percentuálna miera redukcie **BDDcreate()** postupne približuje ku 100%, pretože zatiaľ, čo priemerný počet uzlov stromu pomaly rastie, maximálny počet uzlov (počet uzlov nezredukovaného stromu) rastie exponenciálne.

Percentuálna miera redukcie mojej implementácie **BDDcreate_with_best_order()** sa hýbe okolo 11%. To znamená, že ten istý strom, ktorý by sa dal zostrojiť pomocou **BDDcreate()**, sa dá pomocou tejto funkcie v priemere zostrojiť ešte o ďalších 11% efektívnejšie. Toto číslo by sa určite dalo ešte vylepšiť, keby som túto funkciu implementoval tak, že nájde a vyskúša všetky rôzne možnosti usporiadania premenných (ktorých je N!), no zo zadania stačilo nájsť a vyskúšať iba N možností, čo som aj urobil.

Ako môžme vidieť z grafu, časová zložitosť mojich algoritmov rastie lineárne. Toto je priamy dôsledok vysokej miery redukcie stromov, ktorú algoritmy dosahujú pomocou redukcii typu I a S.

## Zdroje

Informácie na pochopenie binárnych rozhodovacích diagramov som čerpal iba z prednášok predmetu. Spolu so zdrojovými kódmi a dokumentáciou odovzdávam v zip archíve aj textové súbory, ktoré som použil na testovanie.