Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológií

Dátové štruktúry a algoritmy

Zadanie č.2 – Binárne rozhodovacie diagramy

Akademický rok 2021/2022

Meno: Ján Ágh
Cvičiaci: Ing. Lukáš Kohútka, PhD.

Dátum: 7.5.2022
Počet strán: 13

Obsah

1 Úvod

Cieľom druhého zadania z predmetu Dátové štruktúry a algoritmy bolo navrhnúť vlastnú implementáciu binárneho rozhodovacieho diagramu so zameraním na Booleovské funkcie a implementovať dve konkrétne funkcie – BDD_create na vytvorenie a redukciu binárneho rozhodovacieho diagramu a BDD_use na zistenie výsledku pre zvolenú kombináciu vstupných hodnôt. Na implementáciu uvedenej dátovej štruktúry bol použitý programovací jazyk Java 17.0.2 v kombinácií s prostredím Eclipse IDE.

2 Binárny rozhodovací diagram

Binárny rozhodovací diagram (po anglicky Binary decision diagram, skratka BDD) je dátová štruktúra najčastejšie používaná na reprezentáciu Booleovských výrazov. Pozostáva, rovnako ako iné binárne dátové štruktúry, z uzlov a spojení medzi nimi, pričom každý vnútorný uzol má presne dvoch potomkov. Jeho charakteristickou vlastnosťou je skutočnosť, že každý uzol predstavuje jedno čiastkové rozhodnutie a celkový rozhodovací proces prebieha postupným prechodom z jeho koreňa (najvyššie položeného uzla), naprieč celým diagramom, až po niektorý z listov (uzlov na najnižšej úrovni, ktoré už nemajú žiadnych vlastných potomkov).

2.1 Vlastná implementácia

Moja implementácia binárneho rozhodovacieho diagramu pozostáva z piatich tried, z ktorých každá má svoje vlastné využitie. Samotný diagram je reprezentovaný triedou BinDecDiagram, ktorá obsahuje, okrem iného, koreň diagramu, počet uzlov v prípade úplného diagramu a počet uzlov po redukcii. Trieda Node reprezentuje jeden uzol diagramu a jej hlavnou úlohou je uchovávať odkazy na rodiča aj potomkov daného uzla (ak neexistujú, je namiesto nich doplnená hodnota *null*). Triedu DiagramCreator je možné považovať za „mozog“ celého programu, obsahuje totižto implementácie metód BDD_create a BDD_use. Ďalšou nemenej dôležitou triedou je ExpSimplifier, ktorý je naplnený metódami na zjednodušovanie a modifikovanie vstupného Booleovského výrazu podľa rôznych kritérií. Posledná trieda HashCreator v sebe zahŕňa metódu na generovanie hašu z jednotlivých vstupných Booleovských výrazov. Okrem vyššie uvedených tried v programe nájdeme ešte jednu, Main, ktorá obsahuje metódu „main“ a spolu s ňou metódy potrebné na otestovanie správnosti algoritmu.

2.1.1 Trieda BinDecDiagram

Ako už bolo spomenuté, táto trieda reprezentuje celý binárny rozhodovací diagram a obsahuje o ňom niekoľko dôležitých informácií. Medzi tieto informácie patria odkaz na koreň diagramu (najvyššie položený uzol), počet uzlov v úplnom diagrame a počet uzlov po redukcii. Objekt typu BinDecDiagram je návratovou hodnotou metódy BDD_create a zároveň jedným z parametrov metódy BDD_use. Na obr. 2.1 sa nachádza časť triedy BinDecDiagram spolu s jej konštruktorom.

```
public class BinDecDiagram {

    private Node root;
    private String originalExpression;
    private String varTypes;
    private double realNumOfNodes;
    private double maxNumOfNodes;

    public BinDecDiagram(String varTypes, String original) {
        this.originalExpression = original;
        this.root = null;
        this.realNumOfNodes = 1;
        this.maxNumOfNodes = 0;
        this.varTypes = varTypes;

        this.calculateNodes(this.varTypes.length());
    }
```

Obr. 2.1 Premenné triedy BinDecDiagram spolu s implementáciou konštruktora

2.1.2 Trieda Node

Trieda Node (Obr. 2.2) predstavuje jeden uzol vrámci binárneho rozhodovacieho diagramu. Z tohto dôvodu obsahuje všetky dôležité informácie, o ktorých by mal každý uzol vedieť. Medzi ne patria odkazy na svoje rodičovské uzly (používajú sa hlavne pri redukcii, po redukcii sa stávajú nepotrebnými), odkazy na svojho ľavého a pravého potomka, hodnota uzla (tzn. Booleovský výraz, ktorý je v danom uzle vyhodnocovaný) a taktiež informácia o úrovni uzla. Listy diagramu (uzly s hodnotami 0 alebo 1) sú tiež objekty typu Node.

```
public class Node {

    private String value;
    private int level;

    private Node[] parents;
    private Node lowChild;
    private Node highChild;

    public Node(Node parent, String value, int level) {
        this.parents = new Node[1];
        this.value = value;
        this.level = level;
        this.parents[0] = parent;
    }
```

Obr. 2.2 Premenné triedy Node spolu s implementáciou konštruktora

2.1.3 Trieda HashCreator

Táto trieda je veľmi dôležitá z hľadiska zvýšenia efektivity počas redukcie binárneho rozhodovacieho diagramu. Využíva návrhový vzor singleton, aby sme z nej nemuseli pri každej redukcii diagramu vytvárať novú inštanciu, a obsahuje implementácie metód instance (pre získanie inštancie) a hash[1]. Použitie nachádza v dvoch prípadoch – pri hľadaní duplicít medzi Booleovskými výrazmi vrámci jednej úrovne binárneho rozhodovacieho diagramu a taktiež pri zisťovaní a odstraňovaní duplicitných častí vovnútri Booleovských výrazov. Implementácia triedy je zobrazená na obr. 2.3.

```
public class HashCreator {

    private static HashCreator instance;

    private HashCreator() {}

    public static HashCreator instance() {
        if(instance == null) instance = new HashCreator();
        return instance;
    }

    public int hash(String key, int tableSize) {

        long sum = 0;

        for(int i = 0; i < key.length(); i++) sum += ((int) key.charAt(i)) * (sum + 1) + 31 * i;
        return Math.abs((int)(23 * sum + 197) % tableSize);
    }
}
```

Obr. 2.3 Implementácia triedy HashCreator

[1] Jedná sa o rovnakú implementáciu ako v prípade Zadania č.1 – dynamické množiny.

2.1.4 Trieda ExpSimplifier

ExpSimplifier, ako to aj z jej názvu vyplýva, je trieda slúžiaca primárne na zjednodušenie vstupného Booleovského výrazu podľa rôznych kritérií. Využíva návrhový vzor singleton rovnako, ako trieda HashCreator. Zároveň obsahuje aj metódy na skontrolovanie správnosti vstupných údajov, či sa už jedná o poradie premenných v prípade BDD_create alebo vstupný vektor v prípade BDD_use.

Metóda newExp

Táto metóda má tri parametre. Parameter „expression“ obsahuje Booleovský výraz, ktorý má byť predmetom zjednodušenia. Druhý parameter s názvom „upper“ obsahuje veľké písmeno, ktoré bude z výrazu „vyňaté“ (po dokončení behu funkcie výstupný výraz nebude obsahovať túto premennú). Posledný, tretí parameter pomenovaný „flag“ je typu boolean a určuje, či sa nachádzame v „nulovej“ alebo „jednotkovej“ vetve (inými slovami, či za dané písmeno budeme dosadzovať nulu alebo jednotku).

Prvá časť metódy (Obr. 2.4) slúži na overenie, či náhodou nie sú splnené kritériá, podľa ktorých by sa dal vstupný výraz okamžite vyhodnotiť. Medzi tieto kritériá patrí napríklad situácia, kedy sa práve odstraňované písmeno nachádza samostatne vo výraze (ak sa jedná napr. o výraz A+BC a našou úlohou je doplniť za premennú A hodnotu jedna, celý výraz sa v tejto vetve stáva pravdivým bez ohľadu na to, aké hodnoty sú doplnené za premenné B a C), alebo keď vstupný výraz pozostáva už iba z tej premennej, ktorá sa práve odstraňuje (v tomto prípade dokážeme okamžite vrátiť hodnoty jedna a nula v prípade nenegovaných alebo nula a jedna v prípade negovaných premenných).

```
if(expression.equals("1") || expression.equals("0")) return expression;
String[] parts = expression.contains("+") ? expression.split("\\+") : new String[] {expression};
String[] hash = new String[parts.length * 2];

for(String part: parts) {
    if((flag && part.equals(lower)) || (!flag && part.equals(upper))) return "1";
}

if((flag && expression.equals(lower)) || (!flag && expression.equals(upper))) return "1";
if((!flag && expression.equals(lower)) || (flag && expression.equals(upper))) return "0";
```

Obr. 2.4 Prvá časť metódy newExp

Druhá časť metódy (Obr. 2.5) prestavuje hlavnú logiku odstraňovania danej premennej z výrazu. Na začiatku sa výraz v tvare DNF rozdelí na časti podľa znamienka „+“ a jednotlivé časti sú uložené do poľa. Následne sa cyklom iteruje cez toto pole a zisťuje sa, či jednotlivé časti obsahujú odstraňovanú premennú, a ak áno, v akom tvare. Podľa toho sa rozhodne, či bude odstránená celá časť výrazu (napr. ak do časti ABC dosadíme za premennú A hodnotu 0, dostaneme 0*B*C, čo sa dá vyhodnotiť ako nula) alebo iba uvedená premenná (ak do časti ABC dosadíme za premennú A hodnotu 1, dostaneme 1*B*C, čo sa dá vyhodnotiť ako BC). Po vykonaní tohto kroku získame pole častí výrazu, ktoré už neobsahujú danú odstraňovanú premennú.

```
for(int i = 0; i < parts.length; i++) {
    if(parts[i].isBlank()) continue;
    if(parts[i].contains(flag ? upper : lower)) {
        if(parts[i].contains(lower) && parts[i].contains(upper)) {
            if(++count == parts.length) return "0";
        }
        parts[i] = "";
        temp++;
    }
    else if(parts[i].contains(flag ? lower : upper)) {
        parts[i] = this.removeLetter(parts[i], flag ? lower : upper);
        if(parts[i].isBlank()) temp++;
    }
}
```

Obr. 2.5 Druhá časť metódy newExp

Tretia časť metódy sa venuje identifikovaniu a odstraňovaniu duplicít (Obr. 2.6 vľavo), ktoré mohli vzniknúť počas procesu zjednodušovania (napr. C+C sa zjednoduší na C) a opätovnému poskladaniu výrazu do jedného celku (Obr. 2.6 vpravo). Na identifikáciu duplicít sa používa hašovacia funkcia z triedy HashCreator.

```
for(int i = 0; i < parts.length; i++) {
    if(parts[i].isBlank()) continue;

    int pos = HashCreator.instance().hash(parts[i], hash.length);
    while(hash[pos] != null) {
        if(hash[pos].equals(parts[i])) break;
        if(++pos >= hash.length) pos = 0;
    }
    if(hash[pos] != null) {
        parts[i] = ""; temp--;
    }
    else hash[pos] = parts[i];
}
```

```
for(int i = 0; i < parts.length; i++) {
    if(parts[i].isBlank()) continue;
    else expression += parts[i];

    if(temp-- > 0) expression += "+";
}
if(expression.isBlank()) {
    if(lower.contains(upper)) expression = flag ? "0" : "1";
    else expression = flag ? "1" : "0";
}
return expression;
```

Obr. 2.6 Tretia časť metódy newExp

2.1.5 Trieda DiagramCreator

DiagramCreator je možné považovať za „mozog“ celého programu. Nachádzajú sa v ňom implementácie metód BDD_create, BDD_use a metódy vykonávajúce redukciu binárneho rozhodovacieho diagramu ešte počas jeho stavby.

Metóda BDD_create

Z pohľadu binárnych rozhodovacích diagramov sa jedná o asi najdôležitejšiu metódu celého programu. Po zavolaní jej treba odovzdať dve argumenty – počiatočný Booleovský výraz, ktorý bude tvoriť základ binárneho rozhodovacieho stromu, a reťazec obsahujúci poradie premenných. Na začiatku sa vytvorí uzol obsahujúci počiatočný výraz a uloží sa do koreňa diagramu. Následne začne vykonávanie cyklu (Obr. 2.7), v ktorom sa budú postupne po úrovniach vytvárať ďalšie uzly (objekty typu Node) obsahujúce zjednodušené výrazy a spájať so svojimi rodičmi. Zároveň sú tieto uzly ukladané aj do polí, a to z dôvodu, aby sme k nim vedeli jednoducho pristupovať počas redukcie. Po vytvorení celej úrovne uzlov sa zavolá privátna metóda handleReduction a v dvoch oddelených krokoch sa začne redukovanie danej úrovne diagramu spolu s predchádzajúcou úrovňou.

```
public BinDecDiagram BDD_create(String expression, String order) {
    this.expression = expression;
    this.order = order.toUpperCase();
    this.varTypes = this.order.split("");

    if(!ExpSimplifier.instance().validateVarTypes(expression, order)) return null;

    BinDecDiagram diagram = new BinDecDiagram(order, expression);
    diagram.setRoot(this.previous[0] = new Node(null, this.expression, 0));
    int num, pos = 0;

    while(++pos <= this.varTypes.length) {
        num = 0;
        for(int i = 0; i < this.previous.length; i++) {
            if(this.previous[i] == null) continue;

            Node low = this.reduceExp(this.previous[i], this.previous[i].getValue(), pos, true);
            Node high = this.reduceExp(this.previous[i], this.previous[i].getValue(), pos, false);

            this.previous[i].setLow(this.current[num++] = low);
            this.previous[i].setHigh(this.current[num++] = high);
            if(pos != this.varTypes.length) diagram.countNode();
        }
        this.handleReduction(diagram);
        this.previous = this.current;
        this.current = new Node[this.previous.length * 2];
    }
    return diagram;
}
```

Obr. 2.7 Implementácia metódy BDD_create triedy DiagramCreator

Metóda handleReduction

Po zavolaní tejto metódy sa začne redukcia posledných dvoch úrovní diagramu. Oba úrovne sú uchovávané v poliach, aby bol prístup k jednotlivých uzlom čo najjednoduchší. V prvom kroku (Obr. 2.8) sa vykoná redukcia typu I s uzlami najnovšej (najnižšie položenej) úrovne. Počas tejto redukcie sa pomocou hašovacej funkcie postupne hašujú výrazy nachádzajúce sa v uzloch a ukladajú do špeciálneho poľa na index zodpovedajúci hašu výrazu. Ak sa na danom indexe už nachádza iný výraz, porovná sa s novým výrazom a ak sú zhodné, nový výraz nebude vložený do hašovacej tabuľky a zároveň bude uzol obsahujúci tento výraz vymazaný z diagramu. Ak nie sú zhodné, budeme sa posúvať doprava v hašovacej tabuľke dovtedy, kým nenájdeme výraz zhodný s novým výrazom alebo prázdne miesto. Takto je zabezpečené, že žiadna úroveň nebude obsahovať dva uzly s identickými výrazmi a zároveň sa dosahuje aj najväčšia miera efektivity, keďže musíme medzi sebou porovnávať iba výrazy, ktoré sa zahašovali na rovnaké miesto v hašovacej tabuľke.

```
for(int i = 0; i < this.current.length; i++) {
    if(this.current[i] == null) continue;
    int pos = HashCreator.instance().hash(this.current[i].getValue(), hash.length);

    while(hash[pos] != null) {
        if(hash[pos].getValue().equals(this.current[i].getValue())) break;
        if(++pos >= hash.length) pos = 0;
    }
    if(hash[pos] != null) {
        this.deleteNode(hash[pos], i);
        if(hash[pos].getLevel() != this.varTypes.length) diagram.subtractNode();
    }
    else hash[pos] = this.current[i];
}
```

Obr. 2.8 Redukcia typu I vrámci metódy handleReduction

Po vykonaní redukcie typu I sa metóda vráti k poľu obsahujúceho uzly predchádzajúcej úrovne a v druhom kroku (Obr. 2.9) začne vykonávať redukciu typu S nad uzlami tejto úrovne. Tento proces je pomerne jednoduchý – pomocou cyklu prechádzame cez každý uzol úrovne a zisťujeme, či sa jeho pravý potomok nerovná ľavému. V prípade, ak sa rovnajú, daný uzol je zbytočný (nemá žiadnu pridanú hodnotu v diagrame) a je z diagramu vynechaný.

```
for(int i = 0; i < this.previous.length; i++) {
    if(this.previous[i] == null) continue;

    Node low = this.previous[i].getLow();
    Node[] parents = this.previous[i].getParents();

    if(low.equals(this.previous[i].getHigh())) {
        diagram.subtractNode();
        if (this.previous[i].equals(diagram.getRoot())) diagram.setRoot(low);
        low.setParents(parents, this.previous[i]);
        this.previous[i] = low.deleteParent(this.previous[i]);
    }
}
```

Obr. 2.9 Redukcia typu S vrámci metódy handleReduction

Metóda BDD_use

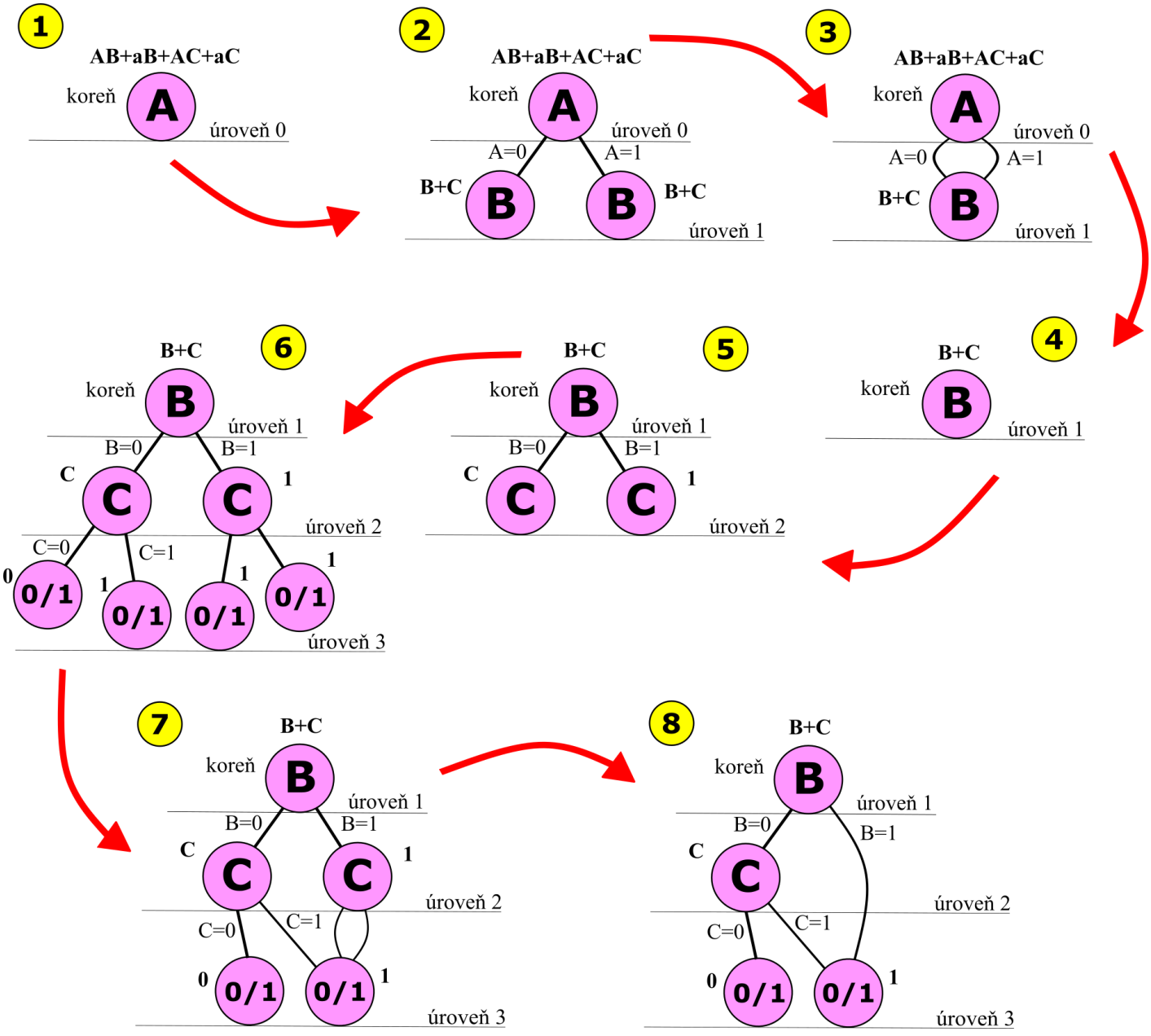
Uvedená metóda (Obr. 2.10) slúži na prechod binárnym rozhodovacím diagramom od koreňa až k listu s cieľom zistiť výsledok Booelovskej funkcie pre danú kombináciu vstupných premenných. Ako argument sa jej odovzdáva vytvorený a zredukovaný diagram a konkrétna kombinácia hodnôt vstupných premenných. Na začiatku sa overí správnosť vstupných hodnôt porovnaním jej dĺžky s počtom premenných výrazu a zároveň testom, či obsahuje iba znaky 0 a 1. Ak je všetko splnené, posun diagramom sa vykonáva pomocou cyklu a po nájdení listu diagramu metóda vráti hodnotu, ktorá bola uložená v danom liste.

```
public String BDD_use(BinDecDiagram diagram, String input) {
    Node current = diagram.getRoot();
    String[] parts = input.split("");
    int pos = current.getLevel() - 1;

    if(parts.length != diagram.getVarTypes().length() || !ExpSimplifier.instance().validateInp(input)) return "-1";

    while(++pos < diagram.getVarTypes().length()) {
        int var = current.getLevel();
        if(pos == var) current = parts[var].equals("0") ? current.getLow() : current.getHigh();
    }
    return current.getValue();
}
```

Obr. 2.10 Metóda BDD_use triedy DiagramCreator

2.2 Príklad tvorby diagramu

Táto podkapitola sa venuje jednej konkrétnej Booleovskej funkcii a obsahuje podrobný opis celého procesu tvorby diagramu od začiatku až po koniec. Ide o funkciu **AB+aB+AC+aC**, kde veľké písmená označujú nenegované a malé písmená negované premenné. Z diagramu budú premenné odstraňované v poradí ABC.



V kroku č.1 sa vytvorí prvý uzol obsahujúci kompletnú Booleovskú funkciu a je vyhlásený za koreň diagramu. V nasledujúcom kroku sa vytvorí ďalšia úroveň s dvoma potomkami, pričom ľavý potomok obsahuje výraz po doplnení hodnoty 0 za premennú A a pravý potomok, naopak, výraz po doplnení hodnoty 1. Akonáhle je vytvorená druhá úroveň, nastáva čas na kontrolu, či na danej úrovni nie sú splnené požiadavky na vykonanie redukcie typu I. Keďže oba uzly obsahujú výraz B+C, jeden z nich možeme pokojne zmazať. V tomto bode (krok č.3) sa vrátime o úroveň vyššie a skúmame, či sa nedá vykonať redukcia typu S. Po porovnaní ľavého a pravého potomka uzla označeného „A“ zistíme, že sú identické, týmpádom je daný uzol nepotrebný, vymažeme ho a vyhlásime jeho potomka za nový koreň stromu (krok č.4). Po odstránení premennej B z výrazu v kroku č.5 sa nenaskytne žiadna príležitosť na redukciu, v nasledujúcich krokoch po vytvorení najnižšej vrstvy s hodnotami 0 a 1 však áno. Dojde k postupnej redukcii najnižšej vrstvy dovtedy, kým nezostane len jeden uzol s hodnotou 0 a jeden s hodnotou 1, a po návrate o úroveň vyššie sa odstráni redundantný uzol s hodnotou 1. Po týchto úpravách je diagram maximálne zredukovaný.

3 Testovanie správnosti a efektivity

Rozhodovací diagram bol podrobený testovaniu pomocou testovacieho programu, ktorý umožňuje päť druhov testov: čisto manuálne testovanie, manuálne testovanie s automatickým vkladaním všetkých kombinácií vstupných premenných, testovanie s náhodne generovaným výrazom, automatické testovanie 100 výrazov a testovanie rýchlosti vytvárania diagramov.

3.1 Manuálne testovanie

Pri manuálnom testovaní (Obr. 3.2) musí používateľ zadať konkrétnu Booleovskú funkciu, z ktorej sa má vytvoriť binárny rozhodovací diagram, spolu s poradím premenných a, po vytvorení diagramu, aj konkrétnu kombináciu hodnôt vstupných premenných. Program poskytne používateľovi dva výsledky: výstup metódy BBD_use a zároveň výstup metódy evaluateExpression (Obr. 3.1), ktorá dokáže okamžite vyhodnotiť vstupnú Booleovskú funkciu. Táto metóda vyhodnocuje funkcie zámenou premenných za konkrétne hodnoty (napr. pri funkcii AB+c a vstupných hodnotách A = 1, B = 0 a C = 0 vytvorí výraz 10+1 (keďže malé „c“ označuje negovanú premennú)) a následne zisťuje, či niektorá časť výrazu obsahuje iba samé jednotky. Ak áno, celý výraz má pravdivostnú hodnotu 1.

```
static String evaluateExpression(String expression, String order, String input) {
    String[] letters = order.split("");
    String[] numbers = input.split("");
    String[] pieces = expression.split("");

    if(letters.length != numbers.length || !ExpSimplifier.instance().validateInp(input)) return "-1";

    for(int i = 0; i < letters.length; i++) {

        String lowercase = letters[i].toLowerCase();

        for(int j = 0; j < pieces.length; j++) {
            if(pieces[j].equals(letters[i])) pieces[j] = numbers[i];
            else if(pieces[j].equals(lowercase)) pieces[j] = numbers[i].equals("1") ? "0" : "1";
        }
    }

    for(int j = 0; j < pieces.length; j++) {
        if(!pieces[j].equals("0") && !pieces[j].equals("1") && !pieces[j].equals("+")) return null;
    }
    expression = String.join("", pieces);
    pieces = expression.contains("+") ? expression.split("\\+") : new String[] {expression};

    for(int i = 0; i < pieces.length; i++) {
        if(!pieces[i].contains("0")) return "1";
    }
    return "0";
}
```

Obr. 3.1 Metóda evaluateExpression triedy Main

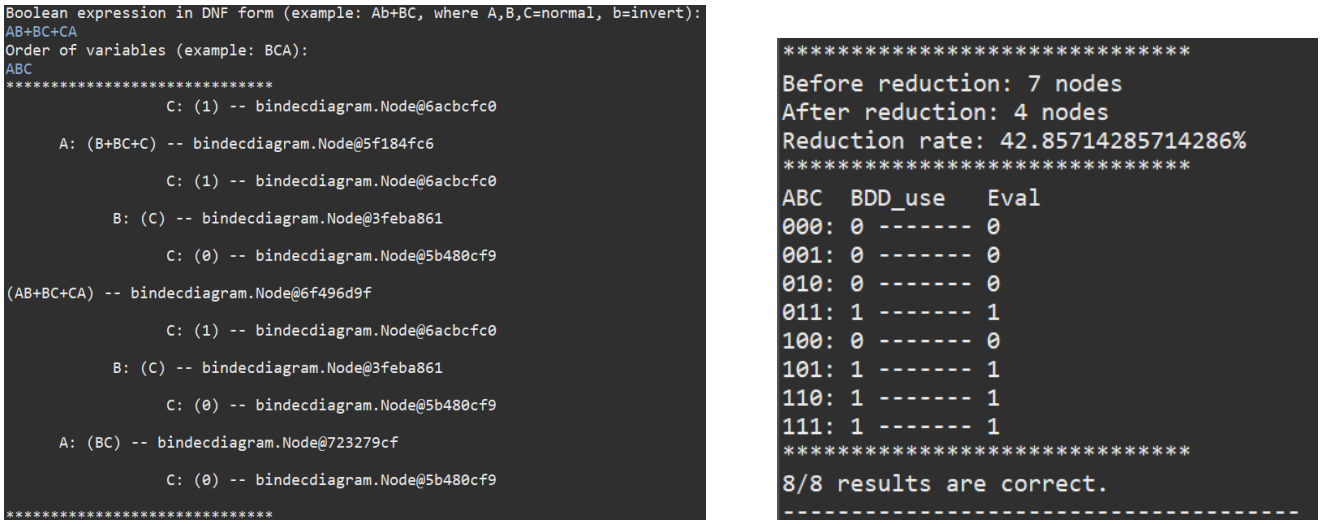
```
Boolean expression in DNF form (example: Ab+BC, where A,B,C=normal, b=invert):
AB+C
Order of variables (example: BCA):
ABC
*****************************
                C: (1) -- bindecdiagram.Node@16b98e56

      A: (B+C) -- bindecdiagram.Node@7ef20235

                C: (1) -- bindecdiagram.Node@16b98e56

           B: (C) -- bindecdiagram.Node@27d6c5e0

                C: (0) -- bindecdiagram.Node@4f3f5b24

(AB+C) -- bindecdiagram.Node@15aeb7ab

                C: (1) -- bindecdiagram.Node@16b98e56

           B: (C) -- bindecdiagram.Node@27d6c5e0

                C: (0) -- bindecdiagram.Node@4f3f5b24

*****************************
```

```
Before reduction: 7 nodes
After reduction: 3 nodes
Reduction rate: 57.142857142857146%
*****************************
Input vector (example: 1010):
001
BDD_use result: 1
Evaluation result: 1
-------------------------------------
```

Obr. 3.2 Ukážka manuálneho testovania (obr. vpravo je pokračovaním obr. vľavo)

3.2 Manuálne testovanie s automatickým vyhodnotením

Tento druh testovania (Obr. 3.3) prebieha podobne ako čisto manuálne testovanie, no s tým rozdielom, že sú do metódy BDD_use automaticky vkladané všetky kombinácie vstupných hodnôt pre daný počet premenných vo výraze. Po skončení testu testovací program poskytne používateľovi tabuľku, ktorá na každom riadku obsahuje danú kombináciu vstupných premenných a výstupy metód BDD_use a evaluateExpression pre túto kombináciu. Toto umožňuje používateľovi aj vizuálne si overiť správnosť riešenia. V poslednom riadku výstupu sa nachádza údaj, v koľkých prípadoch boli výstupy funkcií BDD_use a evaulateExpression identické (riešenie je bezchybné, ak boli výstupy zhodné vo všetkých prípadoch).



Obr. 3.3 Ukážka semi-manuálneho testovania (obr. vpravo je pokračovaním obr. vľavo)

3.3 Náhodné testovanie s automatickým vyhodnotením

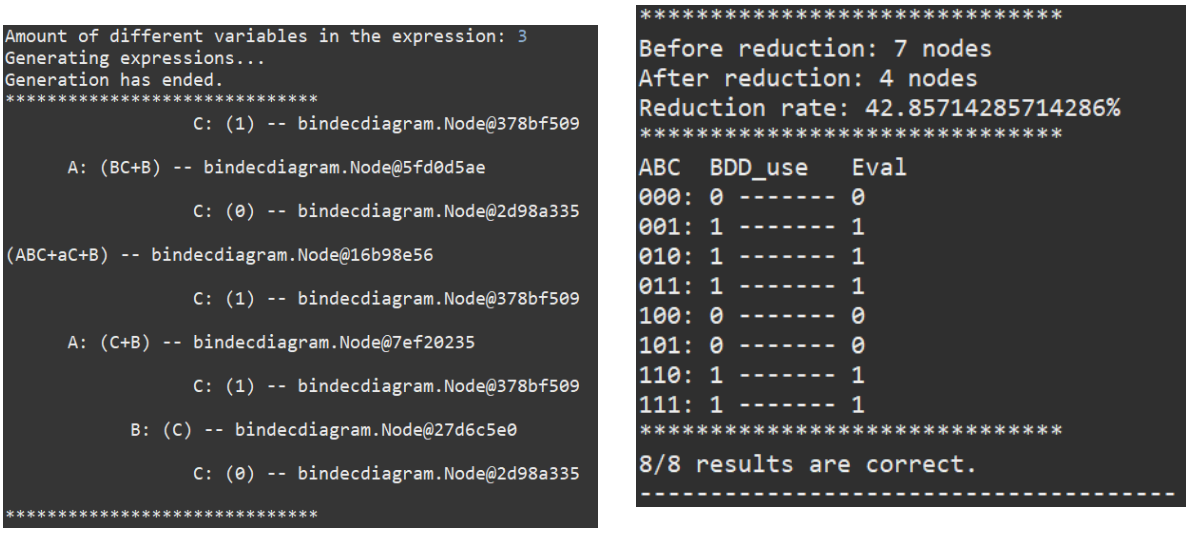
Pri tomto testovaní používateľ nemusí zadávať Booleovskú funkciu, tá bude totiž automaticky vygenerovaná pomocou metódy generateExpressions (Obr. 3.4).

```
static String generateExpressions(int amount, int numOfVars) {
    System.out.println("Generating expressions...");
    expressions = new String[amount];
    String order = "";

    for(int i = 0; i < amount; i++) {
        int parts = r.nextInt(15) + 1;
        String[] exp = new String[parts];
        String newExpression = "";
        Arrays.fill(exp, "");

        for(int j = 65; j < 65 + numOfVars; j++) {
            for(int k = 0; k < exp.length; k++) {
                boolean invert = r.nextBoolean();
                boolean hasLetter = r.nextBoolean();

                if(hasLetter) exp[k] += (invert ? (char)(j + 32) : (char) j);
            }
            order += i > 0 ? "" : (char) j;
        }
        for(int j = 0; j < parts; j++) {
            if(exp[j].isBlank()) continue;

            newExpression += exp[j] + (j + 1 < parts && j + 1 != parts ? "+" : "");
        }
        expressions[i] = newExpression;
    }
    System.out.println("Generation has ended.");
    return order;
}
```

Obr. 3.4 Metóda generateExpressions triedy Main

Generovanie funkcií prebieha nasledovným spôsobom. Používateľ zadá počet funkcií (pri tomto teste sa generuje iba jedna funkcia) spolu s počtom rozličných premenných vo funkciách. Následne sa náhodne určí počet „podčastí“ v každej generovanej funkcii (napr. funkcia AB+BC obsahuje 2 podčasti, a to „AB“ a „BC“) a jednotlivé podčasti sa postupne začínajú napĺňať premennými nasledovne: „Má daná podčasť obsahovať vybranú premennú? Ak áno, má ju obsahovať v priamej alebo negovanej podobe?“ Z uvedeného vyplýva, že nie všetky podčasti obsahujú všetky druhy premenných.

Používateľ pri tomto druhu testovania musí zadať iba počet rozličných premenných vo výraze a zvyšné kroky sa vykonajú automaticky.



Obr. 3.5 Ukážka náhodného testovania (obr. vpravo je pokračovaním obr. vľavo)

3.4 Automatické testovanie 100 výrazov

Tento druh testu slúži na overenie správnosti funkčnosti diagramu. Počas priebehu testu sa náhodne vygeneruje 100 Booleovských výrazov, z ktorých sú postupne vytvárané diagramy a zároveň prebieha vkladanie hodnôt všetkých kombinácií vstupných premenných prostredníctvom metódy BDD_use.

3.5 Testovanie efektivity programu

Piaty a zároveň posledný druh testovania je zameraný na zistenie efektivity celej implementácie binárneho rozhodovacieho diagramu, vrátane samotnej tvorby diagramu a následnej redukcie. Úlohou používateľa je zadať počet funkcií, ktoré sa majú vygenerovať, spolu s počtom rôznych premenných vo funkciách. Na generovanie funkcií slúži metóda generateExpressions. Po zhotovení funkcií sa spustí meranie času a pomocou cyklu sa postupne vytvoria binárne rozhodovacie diagramy pre každú vygenerovanú funkciu (volanie metódy BDD_create). Zároveň sa po vytvorení každého diagramu zaznamená počet jeho uzlov v prípade úplného diagramu a po redukcii. Po vytvorení diagramov sa používateľovi sprístupní záznam obsahujúci informácie uvedené na obr. 3.6.

```
Amount of expressions to be generated: 5000
Amount of different variables in each expression: 6
Generating expressions...
Generation has ended.
****************************
Execution time: 615ms
****************************
Total number of nodes combined: 314559
Number of nodes combined after reduction: 59537
Reduction rate: 81.0728670932957%
------------------------------------
```

Obr. 3.6 Ukážka testovania efektivity

3.5.1 Namerané skutočnosti

V tabuľkách nižšie sú zaznamenané namerané hodnoty pri rôznom množstve funkcií. Pre všetky množstvá (500, 2000, 5000 a 10000) boli uskutočnené tri merania, s počtom rôznych premenných 5, 9 a 13.

Počet funkcií - 500

Premenné	Uzly - úplný diagram	Uzly po redukcii	Miera redukcie	Čas
5	15 438	3 326	78.46%	58ms
9	255 500	22 007	91.39%	128ms
13	4 095 500	50 403	98.77%	256ms

Počet funkcií - 2000

Premenné	Uzly - úplný diagram	Uzly po redukcii	Miera redukcie	Čas
5	61 907	13 621	78.00%	216ms
9	1 022 000	83 588	91.82%	444ms
13	16 382 000	198 311	98.79%	1168ms

Počet funkcií - 5000

Premenné	Uzly - úplný diagram	Uzly po redukcii	Miera redukcie	Čas
5	154 473	33 461	78.34%	400ms
9	2 554 489	209 089	91.81%	1464ms
13	40 955 000	497 105	98.78%	3140ms

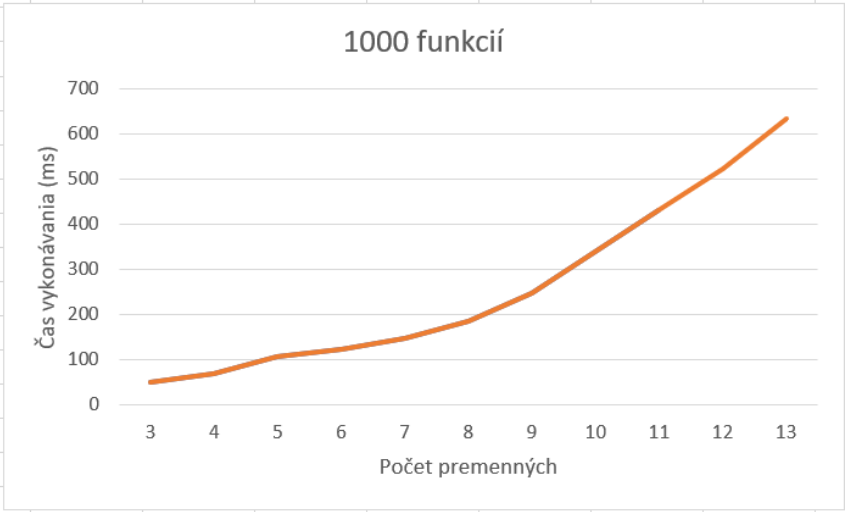
Počet funkcií - 10000

Premenné	Uzly - úplný diagram	Uzly po redukcii	Miera redukcie	Čas
5	309 566	68 547	77.86%	717ms
9	5 108 467	420 629	91.77%	2765ms
13	81 910 000	994 659	98.79%	5882ms

Zo zistených údajov vyplýva viacero zaujímavých skutočností. Miera redukcie diagramov nesúvisí od počtu uzlov ani od množstva funkcií, iba od počtu rôznych premenných. Pri všetkých počtoch funkcií v prípade identického množstva premenných došlo k porovnateľnej miere redukcie (pri piatich premenných sa miera pohybuje v rozmedzí 77-79%, v prípade deviatich premenných 91-92% a pri trinástich premenných až 98-99%). Zároveň sme zistili, že čas zhotovenia stromov a počet uzlov po redukcii sa priamo úmerne zvyšujú s narastajúcim počtom Booleovských funkcií.

3.5.2 Odhad výpočtovej a priestorovej zložitosti

Graf nižšie obsahuje čas vykonávania metódy BDD_create 1000-krát zasebou pri rôznych počtoch premenných. Ako môžeme vidieť, graf počas celej svojej dĺžky narastá rýchlejšie, ako konštantná funkcia, ale nie tak rýchlo ako exponenciálna. Z tohto dôvodu by sa približná výpočtová zložitosť algoritmu mohla pohybovať na kvadratickej úrovni niekde okolo $\mathbf{O(n^2)}$.



Ak by náš algoritmus neobsahoval redukciu, približná pamäťová zložitosť by sa približovala hodnote $O(2^n)$. Vychádzajúc z miery redukcie, ktorá môže v niektorých prípadoch dosahovať až 99%, sa však pamäťová zložitosť znižuje približne až na úroveň $\mathbf{O(n)}$.