Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológií

Dátové štruktúry a algoritmy

Zadanie č. 2 – Binárne rozhodovacie diagramy

Heorhi Davydau
xdavydau@stuba.sk
ID: 116164

Obsah

Zadanie

Vytvorte program, v ktorom bude možné vytvoriť dátovú štruktúru BDD (Binárny Rozhodovací Diagram) so zameraním na využitie pre reprezentáciu Booleovských funkcií.

Konkrétne implementujte tieto funkcie:

- BDD *BDD_create (string *bfunkcia,* string *poradie*);
- BDD *BDD_create_with_best_order (string *bfunkcia*);
- char BDD_use (BDD **bdd*, string *vstupy*);

Samozrejme môžete implementovať aj ďalšie funkcie, ktoré Vám budú nejakým spôsobom pomáhať vimplementácii vyššie spomenutých funkcii, nesmiete však použiť existujúce funkcie na prácu s binárnymi rozhodovacími diagramami.

Funkcia **BDD_create** má slúžiť na zostavenie redukovaného binárneho rozhodovacieho diagramu, ktorý má reprezentovať/opisovať zadanú Booleovskú funkciu, ktorá je zadaná ako argument funkcie **BDD_create** (argument *bfunkcia*). Booleovská funkcia je poskytnutá funkcii **BDD_create** v tvare výrazu (presný formát a dátový typ je na Vás, napríklad DNF string). Druhým argumentom je *poradie* premenných, ktorým sa definuje, v akom poradí sú použité jednotlivé premenné Booleovskej funkcie bfunkcia na vytvorenie BDD (opäť formát a dátový typ si môžete zvoliť akýkoľvek, napr. spájaný zoznam alebo pole premenných, ktoré sú očíslované od 0 po N-1). Návratovou hodnotou funkcie **BDD_create** je ukazovateľ na zostavený (a zároveň redukovaný) binárny rozhodovací diagram (podľa zvoleného poradia premenných), ktorý je reprezentovaný vlastnou štruktúrou BDD. Štruktúra BDD musí obsahovať minimálne tieto zložky: počet premenných, veľkosť BDD (počet uzlov) a ukazovateľ na koreň (prvý uzol) BDD. Samozrejme potrebujete aj vlastnú štruktúru, ktorá bude reprezentovať jeden uzol BDD (ako vrchol v strome). Súčasťou tejto funkcie je nielen vytvorenie BDD, ale aj samotná redukcia BDD – môžete si vybrať, či redukciu vykonáte až po zostavení úplného BDD, alebo či redukujete BDD priebežne už počas jeho vytvárania. Ak sa však rozhodnete redukovať BDD až po jeho úplnom zostavení, riešenie je penalizované 4 bodmi pre nižšiu efektivitu (vyššiu časovú zložitosť).

Funkcia **BDD_create_with_best_order** má slúžiť na nájdenie čo najlepšieho poradia (vrámci vyskúšaných možností) premenných pre zadanú Booleovskú funkciu. Hľadanie spočíva v opakovanom volaní funkcie **BDD_create**, pričom sa skúšajú použiť rozličné poradia premenných (argument *poradie*). Napríklad pre Booleovskú funkciu s 5 premennými môžeme zavolať **BDD_create** 5-krát, s použitím poradia 01234, 12340, 23401, 34012 a 40123. Alebo môžeme vyskúšať aj všetky možné permutácie poradia premenných, ktorých je N!, kde N je počet premenných (t.j. v tomto prípade 5! = 120). Alebo môžeme použiť X náhodne zvolených poradí premenných. Dôležite však je, aby sa vyskúšalo aspoň N ľubovoľných unikátnych poradí premenných, kde N je počet premenných Booleovskej funkcie. Návratovou hodnotou tejto funkcie je ukazovateľ na BDD, pričom sa použije BDD s najnižším počtom uzlov zo všetkých vyskúšaných poradí premenných.

Funkcia **BDD_use** má slúžiť na použitie BDD pre zadanú (konkrétnu) kombináciu hodnôt vstupných premenných Booleovskej funkcie azistenie výsledku Booleovskej funkcie pre túto

kombináciu vstupných premenných. V rámci tejto funkcie „prejdete“ BDD stromom smerom od koreňa po list takou cestou, ktorú určuje práve zadaná kombinácia hodnôt vstupných premenných. Argumentami funkcie **BDD_use** sú ukazovateľ s názvom *bdd* ukazujúci na BDD (ktorý sa má použiť) a ukazovateľ

s názvom *vstupy* ukazujúci na začiatok reťazca (pole znakov). Práve tento reťazec / pole znakov reprezentuje nejakým (vami zvoleným) spôsobom konkrétnu kombináciu hodnôt vstupných premenných Booleovskej funkcie. Napríklad, index poľa reprezentuje nejakú premennú a hodnota na tomto indexe reprezentuje hodnotu tejto premennej (t.j. pre premenné A, B, C a D, kedy A a C sú jednotky a B a D sú nuly, môže ísť napríklad o “1010“), môžete si však zvoliť iný spôsob. Návratovou hodnotou funkcie **BDD_use** je char, ktorý reprezentuje výsledok Booleovskej funkcie – je to buď ‘1’ alebo ‘0’. V prípade chyby (napr. chybný vstup) je táto návratová hodnota záporná (napr. -1).

Okrem implementácie samotných funkcií na prácu s BDD je potrebné vaše riešenie dôkladne otestovať. Vaše riešenie musí byť 100% korektné. V rámci testovania je potrebné, aby ste náhodným spôsobom generovali Booleovské funkcie, podľa ktorých budete vytvárať BDD pomocou funkcie **BDD_create** a funkcie **BDD_create_with_best_order**. Správnosť BDD môžete overiť opakovaným (iteratívnym) volaním funkcie **BDD_use** tak, že použijete postupne všetky možné kombinácie hodnôt vstupných premenných a porovnávate výsledok **BDD_use** s očakávaným výsledkom, ktorý získate vyhodnotením výrazu. Testujte avyhodnoťte Vaše riešenie pre rozličný počet premenných Booleovskej funkcie (čím viac premenných Váš program zvládne, tým lepšie. Mal by byť aspoň 13). Počet rozličných Booleovských funkcií / BDD diagramov pre rovnaký počet premenných Booleovskej funkcie by mal byť minimálne 100. Vrámci testovania tiež vyhodnocujte percentuálnu mieru zredukovania BDD (t.j. počet odstránených uzlov / počet uzlov pre úplný diagram). Taktiež vyhodnoťte dodatočnú percentuálnu mieru zredukovania BDD dosiahnutú skúšaním rôznych poradí premenných, t.j. porovnajte veľkosť BDD vytvoreného priamo funkciou **BDD_create** s veľkosťou BDD získaného funkciou **BDD_create_with_best_order**.

Príklad veľmi jednoduchého testu (len pre pochopenie problematiky):

```
#include <string.h>
int main(){

        BDD* bdd;
        bdd = BDD_create(“AB+C”);
        if (BDD_use(“000”) != ‘0’) /* ocakavanu hodnotu vieme zistit

        dosadenim hodnot vstupnych premennych do vyrazu – lepsie je spravit funkciu na dosadenie tychto hodnot a vypocitanie
        vysledku */

        printf(“error, for A=0, B=0, C=0 result should be 0.\n”); if (BDD_use(“001”) != ‘1’) printf(“error, for A=0, B=0, C=1 result should
        be 1.\n”); if (BDD_use(“010”) != ‘0’) printf(“error, for A=0, B=1, C=0 result should be 0.\n”); if (BDD_use(“011”) != ‘1’)
        printf(“error, for A=0, B=1, C=1 result should be 1.\n”); if (BDD_use(“100”) != ‘0’) printf(“error, for A=1, B=0, C=0 result should
        be 0.\n”); if (BDD_use(“101”) != ‘1’) printf(“error, for A=1, B=0, C=1 result should be 1.\n”); if (BDD_use(“110”) != ‘1’)
        printf(“error, for A=1, B=1, C=0 result should be 1.\n”); if (BDD_use(“111”) != ‘1’) printf(“error, for A=1, B=1, C=1 result should
        be 1.\n”); return 0;

}
```

Okrem implementácie vášho riešenia a jeho testovania vypracujte aj dokumentáciu, v ktorej opíšete vaše riešenie, jednotlivé funkcie, vlastné štruktúry, spôsob testovania a výsledky testovania, ktoré by

mali obsahovať (priemernú) percentuálnu mieru zredukovania BDD a (priemerný) čas vykonania vašich funkcií. Samozrejme aj v závislosti od počtu premenných Booleovskej funkcie. Dokumentácia musí obsahovať hlavičku (kto, aké zadanie odovzdáva), stručný opis použitého algoritmu s názornými nákresmi/obrázkami a krátkymi ukážkami zdrojového kódu, vyberajte len kód, na ktorý chcete extra upozorniť. Pri opise sa snažte dbať osobitý dôraz na zdôvodnenie správnosti vášho riešenia – teda dôvody prečo je dobré/správne, spôsob a vyhodnotenie testovania riešenia. Nakoniec musí technická dokumentácia obsahovať odhad výpočtovej (časovej) a priestorovej (pamäťovej) zložitosti vášho riešenia (**BDD_create** a **BDD_create_with_best_order**). Celkovo musí byť cvičiacemu jasné, že viete čo ste spravili, že viete odôvodniť, že to je správne riešenie, a viete aké je to efektívne.

Riešenie zadania sa odovzdáva do miesta odovzdania v AIS do stanoveného termínu (oneskorené odovzdanie je prípustné len vo vážnych prípadoch, ako napr. choroba, o možnosti odovzdať zadanie oneskorene rozhodne cvičiaci, príp. aj o bodovej penalizácii). Odovzdáva sa jeden **zip** archív, ktorý obsahuje zdrojové súbory s implementáciou riešenia a testovaním + jeden súbor s dokumentáciou vo formáte **pdf**. **Vyžaduje sa tiež odovzdanie programu**, ktorý slúži na testovanie a odmeranie efektívnosti týchto implementácií ako jedného samostatného zdrojového súboru (obsahuje funkciu **main**).

Hodnotenie

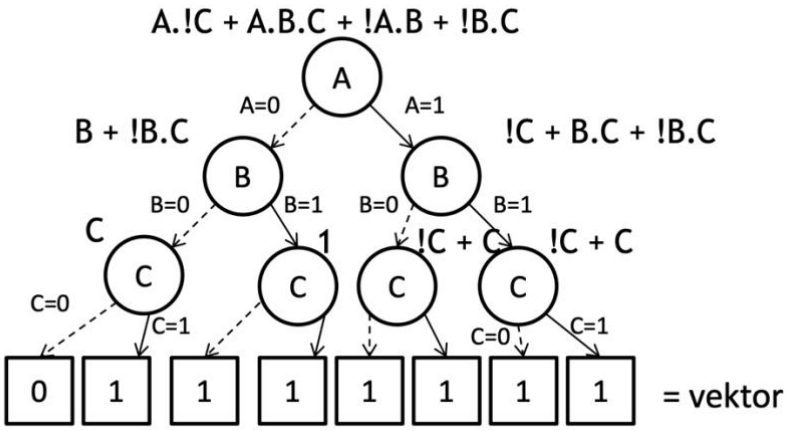
Môžete získať celkovo 20 bodov, **nutné minimum je 8 bodov**.

Za implementáciu riešenia je možné získať celkovo 12 bodov, z toho 8 bodov za funkciu **BDD_create**, 2 body za funkciu **BDD_create_with_best_order** a 2 body za funkciu **BDD_use**. Za testovanie je možné získať 4 bodov (z toho 2 body za automatické zistenie očakávaného výsledku BDD – dosadenie hodnôt do výrazu a jeho vyhodnotenie) a za dokumentáciu 4 body (bez funkčnej implementácie 0 bodov). Body sú ovplyvnené aj prezentáciou cvičiacemu (napr. keď neviete reagovať na otázky vzniká podozrenie, že to **nie je vaša práca, a teda je hodnotená 0 bodov)**. Ak Vaše riešenie nevykonáva žiadnu redukciu alebo ak vytvorený BDD nefunguje 100% správne, riešenie nie je akceptované.

Teoretické východiská

Binárny rozhodovací diagram, známy aj ako BDD, je dátová štruktúra používaná na reprezentáciu a manipuláciu s Booleovskými funkciami. BDD má tvar binárneho stromu, kde každý vnútorný uzol reprezentuje rozhodnutie na základe jednej premennej, a koreň stromu reprezentuje celú Booleovskú funkciu. Prechod cez BDD začína v koreni a pokračuje sa smerom k listom, kde každý list reprezentuje hodnotu Booleovskej funkcie pre danú kombináciu vstupných premenných.

Jednou z najpoužívanejších aplikácií BDD je reprezentácia ľubovoľnej Booleovskej funkcie, čo umožňuje efektívne vykonávať operácie ako sú konjunkcia, disjunkcia, negácia alebo implikácia. Výhodou BDD je ich efektívna reprezentácia a manipulácia s Booleovskými funkciami aj pri veľkých vstupných dátach. Navyše, v porovnaní s inými metódami reprezentácie Booleovských funkcií, ako sú napríklad pravdivostné tabuľky alebo disjunktné normálne formy (DNF), majú BDD mnoho výhod. Napríklad zmena poradia premenných nezmení výsledok funkcie, čo umožňuje optimalizáciu a efektívne vykonávanie operácií s funkciami.



Obrázok 1: Príklad BDD stromu pre booleovskú funkciu

V praxi sa BDD používajú v rôznych oblastiach, ako napríklad v elektronike, automatizácii, softvérovom inžinierstve alebo v modelovaní biologických systémov. BDD sa ukázali ako efektívne nástroje pre analýzu, syntézu a verifikáciu digitálnych obvodov, návrh a optimalizáciu programov a algoritmov, a modelovanie biologických a ekologických systémov.

Realizácia

Štruktúra uzla:

Trieda "Node" predstavuje základnú dátovú štruktúru pre implementáciu binárneho diagramu rozhodnutí (BDD). Obsahuje polia pre uchovávanie hodnoty uzla, jeho úrovne v strome, odkazov na jeho predkov, ľavého a pravého lista. Trieda poskytuje metódy na nastavenie a odstránenie odkazov na predkov, nastavenie ľavého a pravého lista a získanie hodnôt polí.

Code:

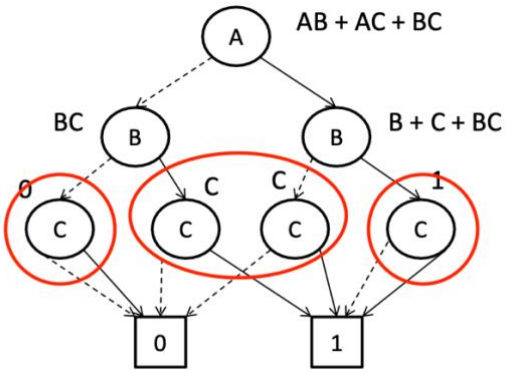
Trieda Node

```
public class Node {
    private final String value;
    private final int level;
    private Node[] ancestors;
    private Node leftChild;
    private Node rightChild;

    public Node(Node parent, String value, int level) {
        // Default constructor for root node
    }

    // Special method for root node
    public void setAncestors(Node[] grandancestor, Node parent) {
        int numParents = ancestors.length;
        int numGrandparents = grandancestor.length;
        Node[] newParents = new Node[numParents + numGrandparents];
        int index = 0;

        for (Node node : ancestors) {
            newParents[index] = node;
            index++;
        }

        for (Node grandparent : grandancestor) {
            newParents[index] = grandparent;
            index++;

            if (parent != null && grandparent != null) {
                if (grandparent.getLeftChild().equals(parent)) {
```

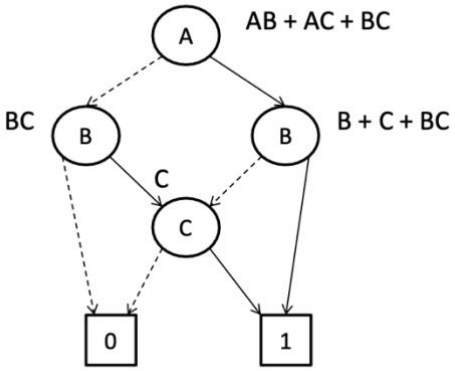
```
                grandparent.setLeftChild(this);
            } else {
                grandparent.setRightChild(this);
            }
        }
    }

    ancestors = newParents;
}

public Node removeAncestor(Node ancestor) {
    for (int i = 0; i < ancestors.length; i++) {
        if (ancestors[i] != null && ancestors[i].equals(ancestor)) {
            ancestors[i] = null;
            return ancestor;
        }
    }
    return null;
}

// All other setters and getters are default
}
```

BDD create

Metoda BDD_create slúžiť na zostavenie redukovaného binárneho rozhodovacieho diagramu, ktorý má reprezentovať/opisovať zadanú Booleovskú funkci a zadaného poradia
V priebehu tvorby Binary Decision Diagram (BDD) používam redukciu jeho veľkosti s cieľom znižovať spotrebu zdrojov. K tomu existujú dva prístupy: zhora-nadol a zdola-nahor.
V mojej implementácii používam prvý prístup. Vytváram koreňový uzol, ktorý reprezentuje celú Booleovu funkciu. Následne vyberám jednu premennú v poradí, ktoré je určené vstupným parametrom. Vytváram dva potomkov a rozdeľujem funkciu na dve časti podľa hodnoty vybranej premennej. Tento proces opakujem v cykle pre každú premennú funkcie, kým nezískam jediný uzol.

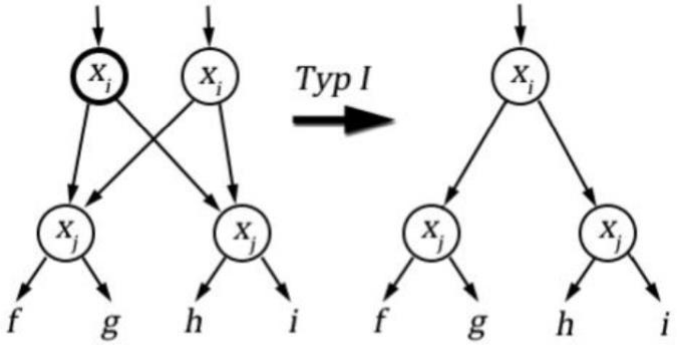


Obrázok 2: BDD pred redukovaním

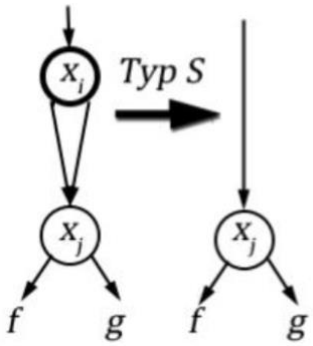


Obrázok 3: BDD po redukovaní

Okrem toho sa strom zjednodušuje na každom kroku pomocou základných pravidiel redukcie BDD. Pravidlo typu I umožňuje odstrániť nadbytočné uzly (v mojej implementácii pomocou porovnania hashu potomkov), porovnávaním dvojíc. Pravidlo typu S umožňuje odstrániť nepotrebné uzly, kontrolou ich potomkov. Obidve pravidlá sa môžu kombinovať na dosiahnutie najlepších výsledkov.



Obrázok 4: Pravidlo I



Obrázok 5: Pravidlo S

Code:

Metoda create:

```
public void create(String bfunction, String order) {
   if (this.root != null) this.clear();

   this.order = order.toUpperCase();

   this.previousNode[0] = new Node(null, bfunction, 0);
   this.root = this.previousNode[0];
   this.increaseNumOfNodesAfterReduce();

   this.varTypesForCreation = this.order.split("");
   for (int i = 1; i <= varTypesForCreation.length; i++) {
      int num = 0;

      for (Node node : this.previousNode) {
         if (node != null) {

            Node low = this.shannonExtension(node, node.getValue(), i, true);
            Node high = this.shannonExtension(node, node.getValue(), i, false);

            node.setLeftChild(this.currentNode[num++] = low);
            node.setRightChild(this.currentNode[num++] = high);
            if (i != this.varTypesForCreation.length)
                              this.increaseNumOfNodesAfterReduce();
         }
      }
```

```
            this.handleReduction();
            this.previousNode = this.currentNode;
            this.currentNode = new Node[this.previousNode.length * 2];
        }

        this.calculateMaxNumberOfNodes();
    }
```

Metoda shannonExtension:

```
    private Node shannonExtension(Node parent, String expression, int varPos, boolean flag) {
        if (varPos > this.varTypesForCreation.length) {
            String value = expression.equals("1") ? "1" : "0";
            return new Node(parent, value, this.varTypesForCreation.length);
        }
        String variable = this.varTypesForCreation[varPos - 1];
        String newExp = this.newExpression(expression, variable, flag);
        return new Node(parent, newExp, varPos);
    }
```

Metoda newExpression:

```
    private String newExpression(String expression, String upper, boolean nullOrZero) {
        String lower = upper.toLowerCase();
        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 ((nullOrZero && part.equals(lower)) ||
                        (!nullOrZero && part.equals(upper))) return "1";
        if ((nullOrZero && expression.equals(lower)) ||
                (!nullOrZero && expression.equals(upper))) return "1";
        if ((!nullOrZero && expression.equals(lower)) ||
                (nullOrZero && expression.equals(upper))) return "0";
        int temp = 0, count = 0;
        for (int i = 0; i < parts.length; i++) {
            if (!parts[i].isBlank()) {
                if (parts[i].contains(nullOrZero ? 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(nullOrZero ? lower : upper)) {
            parts[i] = this.removeLetter(parts[i], nullOrZero ? lower : upper);
            if (parts[i].isBlank()) temp++;
        }
    }
}
temp = parts.length - temp - 1;
for (int i = 0; i < parts.length; i++) {
    if (!parts[i].isBlank()) {
        int pos = this.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];
    }
}
lower = expression;
StringBuilder expressionBuilder = new StringBuilder();
for (String part : parts) {
    if (!part.isBlank()) {
        expressionBuilder.append(part);
        if (temp-- > 0) expressionBuilder.append("+");
    }
}
expression = expressionBuilder.toString();
if (expression.isBlank()) {
    if (lower.contains(upper)) expression = nullOrZero ? "0" : "1";
    else expression = nullOrZero ? "1" : "0";
}
return expression;
}
```

Metoda hash:

```
private int hash(String key, int tableSize) {
    return Math.abs(key.hashCode() % tableSize);
}
```

Metoda handleReduction:

```
private void handleReduction() {
    Node[] hashTable = new Node[this.currentNode.length * 2];
    for (int i = 0; i < this.currentNode.length; i++) {
        if (this.currentNode[i] != null) {
            String value = this.currentNode[i].getValue();
            int pos = this.hash(value, hashTable.length);
            while (hashTable[pos] != null) {
                if (hashTable[pos].getValue().equals(value)) break;
                if (++pos >= hashTable.length) pos = 0;
            }
            if (hashTable[pos] != null) {
                this.deleteNode(hashTable[pos], i);
                if (hashTable[pos].getLevel() != this.varTypesForCreation.length) {
                    this.decreaseNumOfNodesAfterReduce();
                }
            } else {
                hashTable[pos] = this.currentNode[i];
            }
        }
    }
    for (int i = 0; i < this.previousNode.length; i++) {
        if (this.previousNode[i] != null) {
            Node lowChild = this.previousNode[i].getLeftChild();
            Node[] parents = this.previousNode[i].getAncestors();
            if (lowChild.equals(this.previousNode[i].getRightChild())) {
                this.decreaseNumOfNodesAfterReduce();
                if (this.previousNode[i].equals(this.root)) {
                    this.root = lowChild;
                }
                lowChild.setAncestors(parents, this.previousNode[i]);
                this.previousNode[i] =
                                    lowChild.removeAncestor(this.previousNode[i]);
            }
        }
```

```
    }
}
```

BDD create with best order

Metóda createWithBestOrder slúži na nájdenie najlepšieho poradia premenných pre zadanú Booleovskú funkciu. V mojej implementácii createWithBestOrder skúša všetky možné permutácie poradia premenných, čo predstavuje N! možností, kde N je počet rožných premenných v Booleovskej funkcii. Algoritmus následne vyhodnotí percentuálny úspech každej permutácie a nájde tá, ktorá produkuje najlepšie výsledky.

Ak počet rôznych premenných v Booleovskej funkcii presahuje 6, metóda createWithBestOrder náhodne vyberá poradie premenných a skúša aspoň N možností. To je z dôvodu, že počet permutácií pre N rôznych premenných je N!, čo môže byť príliš veľa na to, aby sa všetky možnosti preverili.

Code:

Metoda createWithBestOrder:

```
public void createWithBeastOrder(String bfunction) {
    String order = generateOrder(bfunction);
    String beastOrder = "";
    double minNumOfNodesAfterReduce = Double.MAX_VALUE;
    if (order.length() > 20)
            throw new IllegalArgumentException("Too many variables");
    else if (order.length() > 6) {
        Random random = new Random();
        for (int i = 0; i < order.length(); i++) {
            String combination = getCombination(order,
                                                random.nextLong(getNumberOfCombinations(order)));
            this.create(bfunction, combination);
            if (minNumOfNodesAfterReduce > this.numOfNodesAfterReduce) {
                minNumOfNodesAfterReduce = this.numOfNodesAfterReduce;
                beastOrder = this.order;
            }
        }
    } else {
        for (long i = 0; i < getNumberOfCombinations(order); i++) {
            this.create(bfunction, getCombination(order, i));
            if (minNumOfNodesAfterReduce > this.numOfNodesAfterReduce) {
                minNumOfNodesAfterReduce = this.numOfNodesAfterReduce;
```

```
                beastOrder = this.order;
            }
        }
    }
    this.create(bfunction, beastOrder);
}
```

Metoda getCombination:

```
public String getCombination(String str, long index) {
    int n = str.length();
    long factorial = this.getNumberOfCombinations(str);
    if (index < 0 || index >= factorial) {
        throw new IllegalArgumentException("Invalid index: " + index);
    }
    StringBuilder sb = new StringBuilder(n);
    List<Character> chars = new ArrayList<>();
    for (char c : str.toCharArray()) {
        chars.add(c);
    }
    for (int i = 0; i < n; i++) {
        factorial /= (n - i);
        int j = Math.toIntExact((index / factorial));
        char c = chars.remove(j);
        sb.append(c);
        index -= j * factorial;
    }
    return sb.toString();
}
```

Metoda getNumberOfCombinations:

```
private long getNumberOfCombinations(String order) {
    long factorial = 1;
    for (int i = 1; i <= order.length(); i++) {
        factorial *= i;
    }
    return factorial;
}
```

BDD use

Metoda BDD_use slúžiť 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 ja prehádzam po BDD stromom smerom od koreňa po list takou cestou, ktorú určuje práve zadaná kombinácia hodnôt vstupných premenných.

Code:

Metoda use:

```
public String use(String inputs) {
    Node currentNode = this.root;
    String[] parts = inputs.split("");
    int currentLevel = currentNode.getLevel() - 1;
    for (int i = currentLevel; i < this.order.length(); i++) {
        int nodeLevel = currentNode.getLevel();
        if (i == nodeLevel) {
            currentNode = parts[nodeLevel].equals("0") ?
                    currentNode.getLeftChild() :
                    currentNode.getRightChild();
        }
    }
    return currentNode.getValue();
}
```

Testovanie

Na testovanie BDD stromu bola vytvorená špeciálna programová aplikácia, ktorá obsahuje 3 metódy pre manuálne/automatické testovanie a 2 pre automatické testovanie.

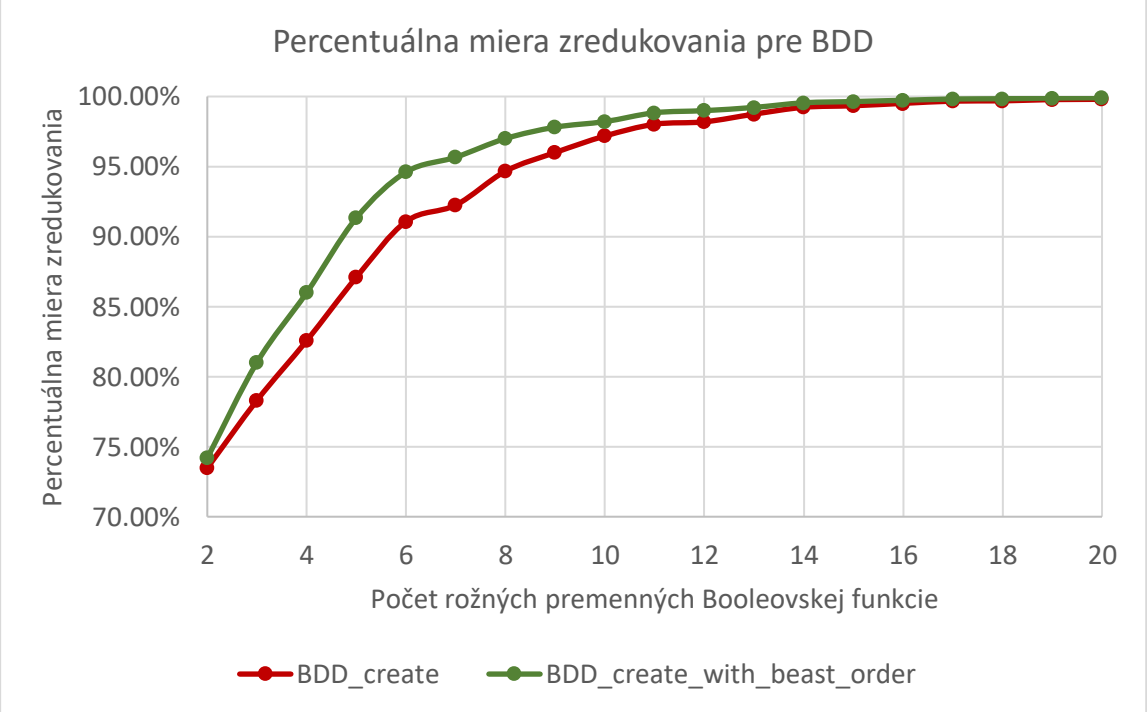
1. Test vstupu (BDD_create) - test s vašou Booleovskou funkciou, poradím premenných a vstupom
2. Test vstupu (BDD_create_with_best_order) - test s vašou Booleovskou funkciou a vstupom, ale s najlepším poradím premenných
3. Test pravdivostnej tabuľky - test s vašou Booleovskou funkciou, poradím premenných a vstupom
4. Test redukcie - test percentuálnej mery redukcie pre náhodné Booleovské funkcie rôznej dĺžky a poradia premenných
5. Test času - test času na vytvorenie BDD pre náhodné Booleovské funkcie rôznej dĺžky a poradia premenných

Testy obsahujú odhad výpočtovej (časovej) zložitosti pre metódy BDD_create a BDD_create_with_best_order. Tiež je test pre vyhodnotenie percentuálnej miery redukcie BDD (t.j. počet odstránených uzlov / počet uzlov pre úplný diagram).

Test pre percentuálnu mieru zredukovania BDD

Táto tabuľka obsahuje informácie o priemernom percentuálnom redukovaní pre 100 náhodných boolean funkcií pre každý počet rôznych premenných od 2 do 21 pomocou metód "BDD create" a "BDD create with beast order".

Tabuľka výsledkov testov pre percentuálnu mieru zredukovania BDD										
Počet rožných premenných Booleovskej funkcie	2	3	4	5	6	7	8	9	10	11
BDD	73,5%	78,3%	82,6%	8,7%	91,1%	92,3%	94,7%	96,0%	97,2%	98,0%
BDD beast order	74,2%	81,0%	0,9%	91,3%	94,6%	95,7%	97,0%	97,8%	98,2%	98,8%
Počet rožných premenných Booleovskej funkcie	12	13	14	15	16	17	18	19	20	21
BDD	98,2%	98,7%	99,2%	99,3%	99,5%	99,7%	99,7%	99,8%	99,0%	99,9%
BDD beast order	99,0%	99,2%	99,6%	99,6%	99,7%	99,8%	99,8%	99,9%	99,9%	99,9%



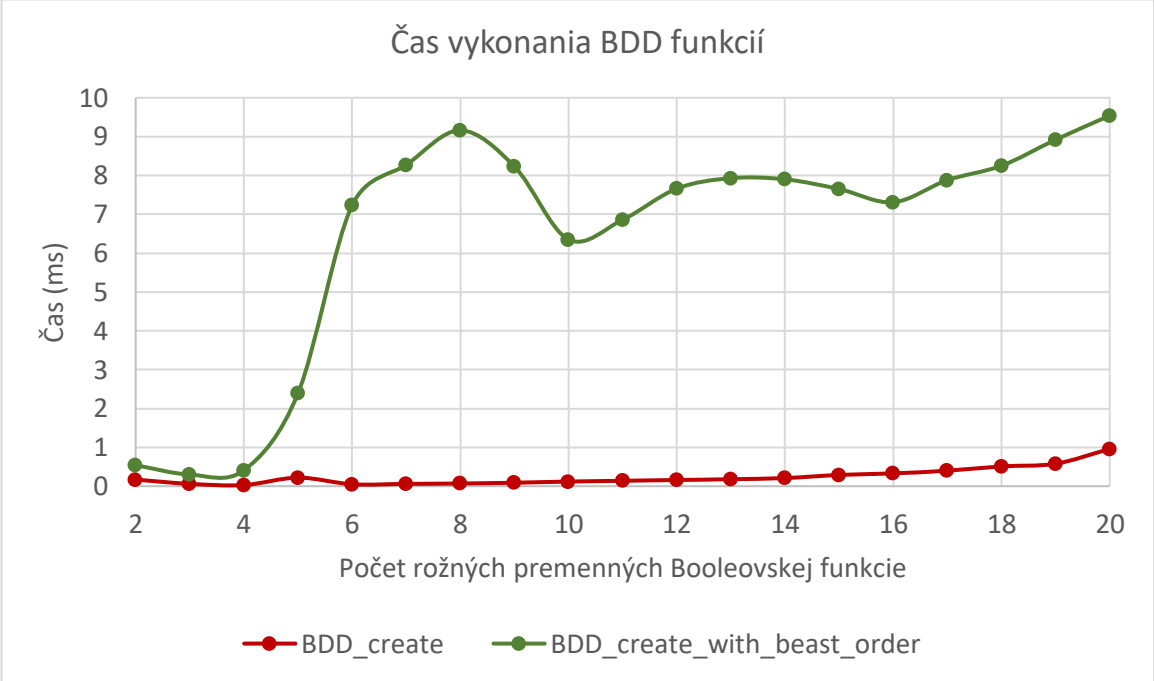
Na grafe vidno, že metóda "BDD create" sa zhoršuje pri zjednodušovaní BDD viac než metóda "BDD create with beast order". Avšak pri zvyšovaní počtu premenných a zložitosti Booleových

funkcií sa toto rozdielovanie zmenšuje a tvorí menej percent. Graf predstavuje logaritmickú funkciu. Komplexita algoritmu: O(logn).

Test pre čas vykonania BDD funkcií

Táto tabuľka obsahuje informácie o priemernom čase (v milisekundách) potrebnom na vytvorenie 100 náhodných boolean funkcií pre každý počet rôznych premenných od 2 do 21 pomocou metód "BDD create" a "BDD create with beast order".

Tabuľka výsledkov testov pre čas vykonania BDD funkcií										
Počet rožných premenných Booleovskej funkcie	2	3	4	5	6	7	8	9	10	11
Čas pre BDD (ms)	0,17	0,06	0,03	0,22	0,05	0,06	0,07	0,09	0,12	0,14
Čas pre BDD beast order (ms)	0,54	0,3	0,41	2,39	7,23	8,27	9,16	8,23	6,35	6,86
Počet rožných premenných Booleovskej funkcie	12	13	14	15	16	17	18	19	20	21
Čas pre BDD (ms)	0,16	0,18	0,21	0,29	0,33	0,4	0,51	0,58	0,96	1,69
Čas pre BDD beast order (ms)	7,66	7,93	7,9	7,65	7,31	7,88	8,25	8,92	9,54	10,2



Ako vidíme na grafe, metóda "BDD create" je rýchlejšia pri tvorbe BDD, pretože neprechádza všetky možnosti a nehľadá najlepšiu možnosť pre vytvorenie BDD. Na druhej strane metóda

"BDD create with beast order" tvorí BDD oveľa dlhšie, pretože tvorí BDD viackrát pre hľadanie najlepšej postupnosti pre maximálne zmenšenie BDD. Na grafe je tiež vidieť, že po 6 rôznych premenných "BDD create with beast order" tvorí BDD rýchlejšie, to je spôsobené realizáciou a tým, že "BDD create with beast order" overuje menší počet možností. Komplexita algoritmu: O(n).

Záver:

BDD (Binary Decision Diagram) sú dôležitým nástrojom pre formálne overovanie softvéru a iných systémov. Pri tvorbe BDD existujú dva hlavné prístupy - "BDD create" a "BDD create with beast order".

Metóda "BDD create" je rýchlejšia, pretože nevyžaduje prechádzanie všetkých možností a nehľadá najlepšiu možnosť pre tvorbu BDD. Táto metóda sa zvyčajne používa pre BDD s malým počtom premenných alebo pre BDD, ktoré sa musia vytvoriť rýchlo.

Na druhej strane metóda "BDD create with beast order" tvorí BDD oveľa pomalšie, pretože pre každú postupnosť premenných tvorí BDD viackrát pre nájdenie najlepšieho výsledku. Táto metóda sa zvyčajne používa pre BDD s väčším počtom premenných a pre BDD, ktoré musia byť vytvorené s maximálnym možným zmenšením.

V súvislosti s implementáciou a použitím týchto metód platí, že "BDD create" je jednoduchšie implementovať a je vhodnejšie pre použitie v jednoduchších prípadoch, zatiaľ čo "BDD create with beast order" je náročnejšie na implementáciu, ale môže byť účinnejšie pre zložitejšie prípady.

Zhrnutím, voľba medzi "BDD create" a "BDD create with beast order" závisí od počtu premenných a zložitosti Booleových funkcií. Pre jednoduchšie prípady s menším počtom premenných a zložitosti Booleových funkcií je lepšie použiť "BDD create", zatiaľ čo pre zložitejšie prípady s väčším počtom premenných a zložitosti Booleových funkcií je lepšie použiť "BDD create with beast order".

Bibliografia

- Kohútka, L. Binárne Rozhodovacie Diagramy
- Kohútka, L. Binárne Rozhodovacie Diagramy 2