2021/2022

Dátové štruktúry a algoritmy

Zadanie 2

Cvičiaci: MSc. Mirwais Ahmadzai **Author: Michal Darovec**

Čas cvičení: Thursday 11:00 – 12:50 **ID: 116163**

Contents

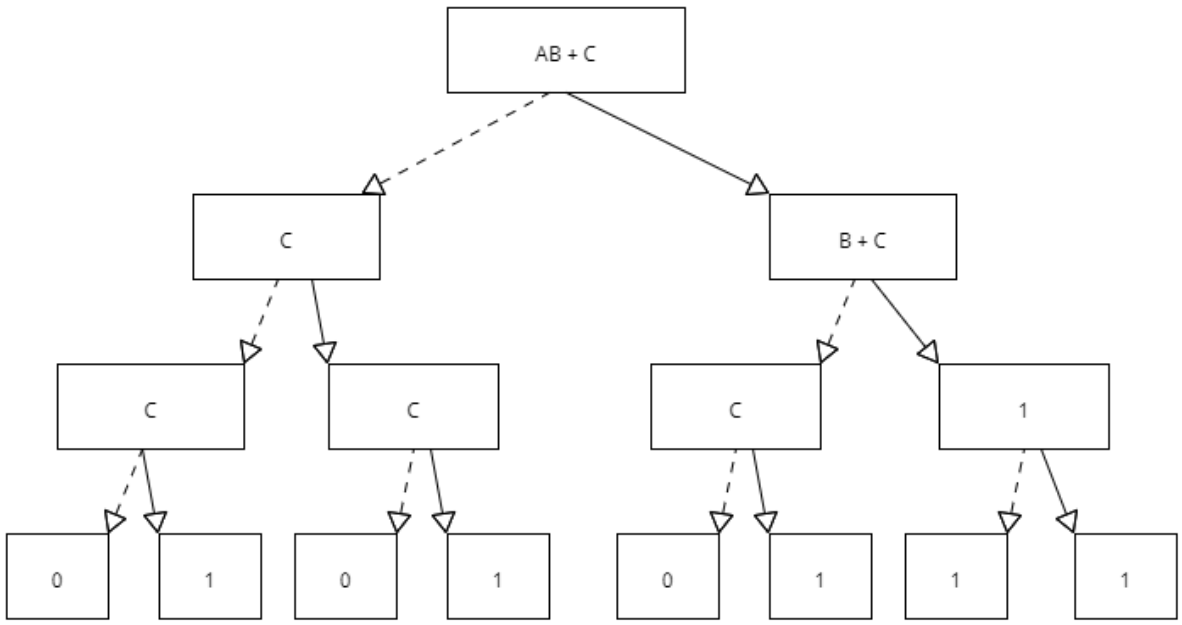
Úvod

V tejto práci zostrojíme binárny rozhodovací diagram. Je to štruktúra, ktorá rozloží výraz v DNF forme do úrovne, keď všetky listy tohto stromu budú obsahovať iba jednotky a nuly. V tomto diagrame budeme rekurzívne rozkladať výraz pomocou rekurzie a zároveň ho budeme redukovať, aby sme použili čo najmenej uzlov. Vo funkcii bdd_create budeme mať na vstupe výraz v DNF forme a poradie premenných, v ktorom ho budeme rozkladať. Výrazy budeme rozkladať pomocou Shannonovej dekompozície. Táto dekompozícia vždy spraví to, že z výrazu odíde daná premenná ľavý potomok daného výrazu bude vyzerať tak, že za premennú dosadíme 0 a pravý potomok tak, že za premennú dosadíme 1. Vo funkcii bdd_use otestujeme daný diagram na určitom vstupe. Čiže výstup by mal byť buď 0 alebo 1. Následne otestujeme celý program aj pomocou vlastných funkcií, ktoré budú generovať náhodné booleovské výrazy v DNF forme. Zmeriame aj mieru redukcie, ktorú daný binárny rozhodovací diagram vykoná.

Binárny rozhodovací diagram

Binárny vyhľadávací strom je stromová dátová štruktúra, ktorá funguje na princípe postupného rozkladu výrazu podľa jednotlivých premenných. Proces rozkladu vyzerá takto: vždy máme nejakú premennú podľa ktorej výraz rozkladáme. Na ľavej strane si vytvoríme výraz po tom, čo by sa daná premenná rovnala 0 a na pravej strane si vytvoríme výraz po tom, čo by sa daná premenná rovnala 1. Táto premenná už vo výraze nebude ani v normálnom, ani v negovanom tvare. Postupne prechádzame úrovne a postupujeme podľa poradia, aké dostaneme na vstupe. Keď už vieme, že daný výraz sa bude rovnať 1 alebo 0, automaticky ho nastavíme na túto hodnotu a vieme, že týmto výrazom sme skončili danú vetvu, vrátime daný uzol.

V následujúcom obrázku si ukážeme, ako približne taký binárny rozhodovací diagram vyzerá. Vybral som si funkciu AB+C, ktorú si rozložíme tak, ako je ukázané v diagrame. Je dôležité pripomenúť, že tento diagram ešte nie je redukovaný.



Implementácia

V zostrojovaní som sa zameriaval hlavne na funkčnosť programu, potom som riešil redukcie a na záver som celý program otestoval. Hlavná časť celej implementácie bolo správne rozložiť výraz podľa danej premennej.

1. Shannonova dekompozícia

Shannonova dekompozícia je proces, podľa ktorého rozložíme výraz na menšie časti. Spravíme to tak, že vyjmeme daný výraz, či už negovaný alebo normálny, pred zátvorku a zvyšok dáme do zátvorky.

$$f(X_1, X_2, \ldots, X_n) = X_1 \cdot f(1, X_2, \ldots, X_n) + X_1' \cdot f(0, X_2, \ldots, X_n)$$

Svoju dekompozíciu volám rekurzívne a ak dostanem hodnoty 0 alebo 1, viem, že rekurzia na danej vetve skončila, môžem vrátiť aktuálny uzol. Inak pokračujem ďalej, rozložím si výraz podľa pravidiel Shannonovej dekompozície a využívam tu aj pravidlá booleovskej algebry, aby bol proces čo najjednoduchší.

```
public Node decomposition(Node root, String letter, int level) {

    if(root.value == 1 || root.value == 0) {
        return root;
    }
```

V samotnej dekompozícii som si spravil podmienky, podľa ktorých do nových výrazov pridávam, čo tam patrí. Kód mám rozdelený tak, že najskôr riešim časť, kedy sa premenná rovná 0, čiže vytváram výraz pre ľavého potomka a následne zopakujem proces pre pravého potomka, samozrejme s pozmenenými podmienkami.

Uvádzam príklad pre jednu z podmienok, kde sa kontroluje, či výraz obsahuje iba jednu premennú, ktorá nie je negovaná. V tomto prípade by bol ľavý potomok list, ktorý by obsahoval hodnotu 0.

```
// case 0
StringBuilder exp1 = new StringBuilder();
for(String s : expArray) {
    if((expArray.length == 1 && s.contains(letter))) {
        exp1 = new StringBuilder("0");
        break;
    }
```

Tieto podmienky vyzerajú inak, ak rátame s tým, že premenná, podľa ktorej rozkladáme, je rovná 1. Jeden z prípadov, čo môžu nastať je, ak je premenná jediný člen konjunkcie. V tomto prípade vieme, že celý výraz bude rovný 1.

```
// case 1
StringBuilder exp2 = new StringBuilder();
for(String s : expArray) {
    if(s.equals(letter)) { // A = 1 and A
        exp2 = new StringBuilder("1");
        break;
    }
```

Toto sú iba niektoré podmienky dekompozície, po prvých podmienkach nasledujú ďalšie, ktoré sú menej samozrejmé a ťažšie na implementáciu.

2. Redukcie

Pri binárnych rozhodovacích stromoch sa oplatí robiť redukciu, to znamená zníženie množstva uzlov na minimum, ktoré potrebujeme na skompletizovanie funkčného stromu. Rozlišujeme dva typy redukcií, typ I a typ S.

Typ I implementujeme pomocou ukladania všetkých uzlov s danými výrazmi do hashovacích tabuliek nasledovne:

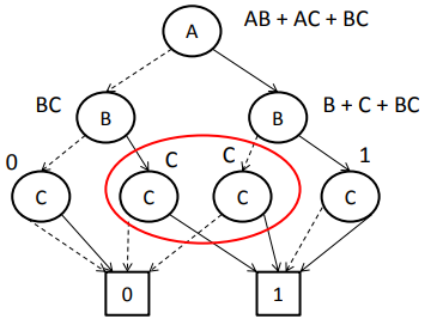
```
private ArrayList<HashMap<String, Node>> mapList = new ArrayList<>();
```

Zakaždým, keď vytvoríme nový uzol, uložíme ho sem a pri vytváraní nového si vždy skontrolujeme, či už takýto uzol náhodou nemáme vytvorený. Ak áno, iba presunieme vytvoreného potomka na už existujúci uzol a nemusíme tak zbytočne vytvárať žiadny uzol. Pre každú úroveň máme vlastnú hashovaciu tabuľku, kde postupne ukladáme výrazy a pri každom novom výraze kontrolujeme, či sme ho už v danej úrovni nevytvorili.

```
if(map.containsKey(exp1.toString())) {
    root.left = map.get(exp1.toString());
```

Pomocou tejto tabuľky vieme vykonávať redukciu už počas vytvárania diagramu a nemusíme neskôr zlučovať uzly po vytvorení.

- Y = AB + AC + BC



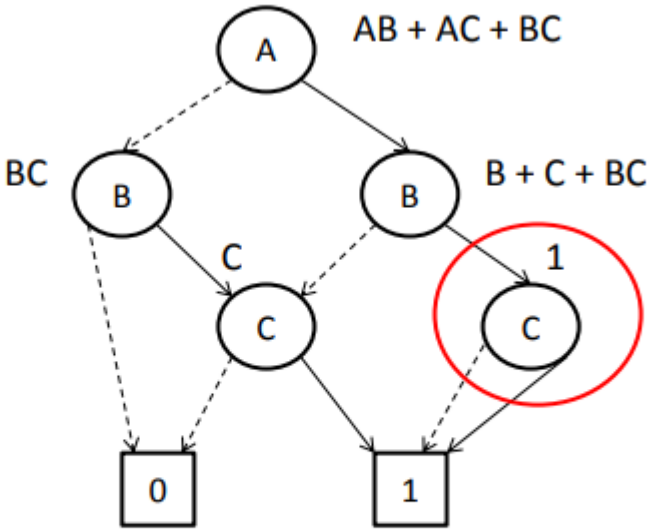
Redukcia typu I

- uzly reprezentujú rovnakú funkciu (C)
- ľavý potomok oboch uzlov je rovnaký a pravý potomok oboch uzlov je rovnaký

Pri redukcii typu S ide o to, že ak má uzol rovnakého ľavého aj pravého potomka, daný uzol nie je potrebný. Môžeme ho bez problémov vymeniť za jeho rodiča a tento uzol zanikne. Pri tomto si vytvoríme metódu, ktorú budeme na každom prvku vykonávať po rekurzívnom vrátení funkcie decomposition, na konci tejto metódy.

```
if(root.left.equals(root.right)) {
    for(Node parent : root.parents) { /
        if(parent.left.equals(root)) {
            parent.left = root.left; //
        }
        if(parent.right.equals(root)) {
            parent.right = root.right;
        }
```

Vizuálna ukážka, kedy nastane takýto prípad, môžeme vidieť na tomto obrázku. Vyzerá to približne takto:



Redukcia typu S

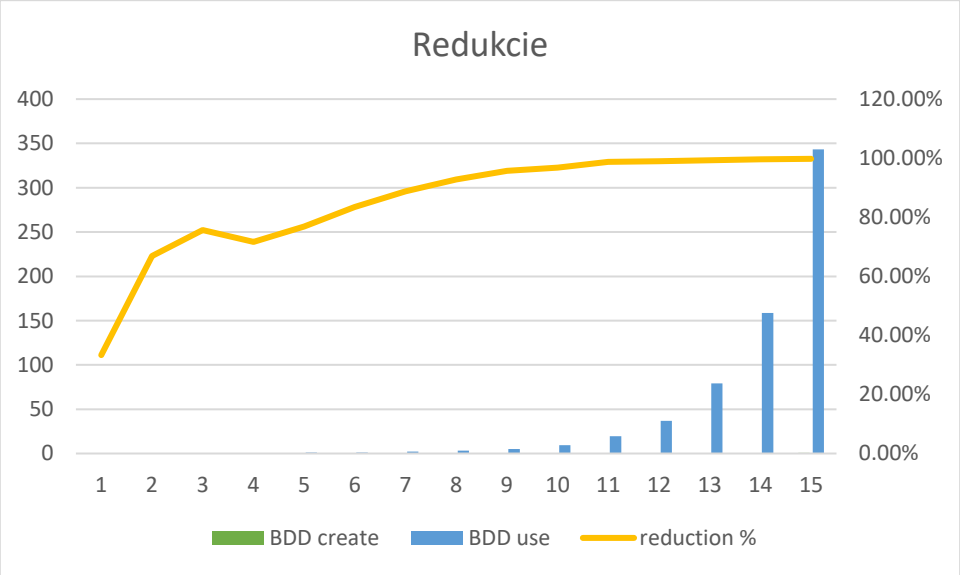
- ľavý potomok je rovnaký ako pravý potomok (pointre sa rovnajú)
- špeciálny prípad – uzol reprezentuje konštantu (1)

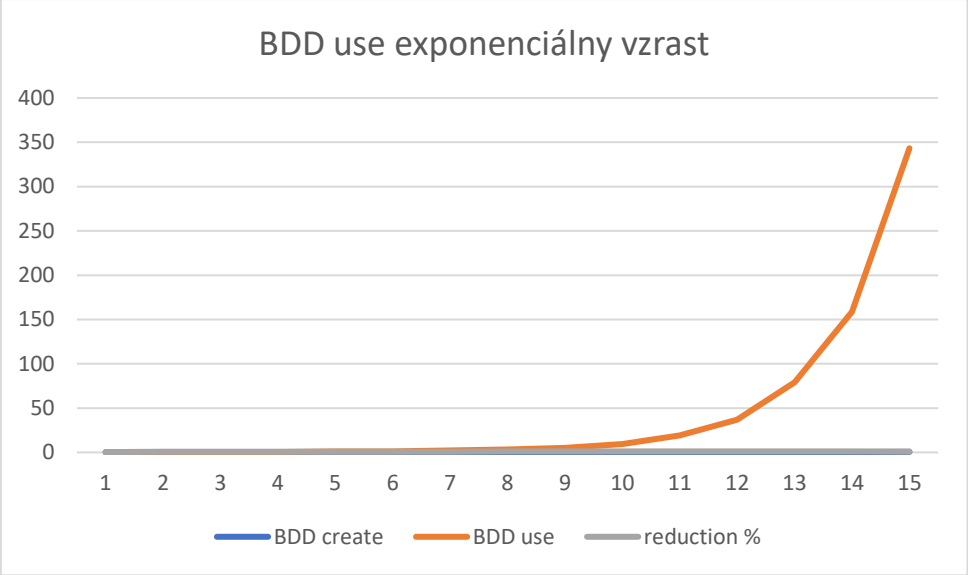
Testovanie

Testovanie je vykonávané prostredníctvom niekoľkých užitočných funkcií. Najskôr si vytvoríme náhodnú postupnosť písmen, s dĺžkou, akú si určíme na vstupe. Do konzoly hneď po spustení programu zadáme počet premenných, koľko chceme mať v diagrame. Vytvorí sa nám funkcia z náhodne vytvorených písmen a takisto náhodné poradie premenných. Iteratívne vytvoríme 100 rôznych DNF funkcií, pre každú zmeriame čas vytvorenia BDD a čas jeho použitia, spravíme aritmetický priemer, aby sme zistili priemerný čas pre jednu funkciu.

variables	BDD create	BDD use	reduction %
1	0.21	0.09	33.33%
2	0.24	0.16	66.86%
3	0.24	0.28	75.67%
4	0.3	0.48	71.61%
5	0.38	0.96	76.90%
6	0.34	1.15	83.43%
7	0.38	2.05	88.81%
8	0.4	3.12	92.78%
9	0.39	5.21	95.73%
10	0.39	9.5	96.77%
11	0.4	19.34	98.73%
12	0.41	36.87	98.92%
13	0.49	79.3	99.35%
14	0.51	158.63	99.67%
15	0.55	343.27	99.79%

V tejto tabuľke sú zapísané časy testovania jednotlivých premenných. Časy BDD create sa veľmi nelíšia, je to kvôli vytváraniu funkcií podobnej dĺžky ako pri testovaní iných počtov premenných. Miera redukcie je vypočítaná vydelením počtu uzlov v redukovanom strome a počtu uzlov v úplnom nezredukovanom strome. Vždy sa počtom premenných zvyšuje, lebo je tu zakaždým viac redukcií.





V tomto grafe vidíme exponenciálny vzrast funkcie BDD use. Je to z toho dôvodu, že ak chceme skontrolovať všetky prípady, vyjde nám 2^n možností, pre ktoré kontrolujeme výraz.

Zložitosť funkcie: O(2^n) – exponenciálna zložitosť

Záver

V tomto zadaní sme vytvárali binárny rozhodovací diagram, ktorý fungoval na princípe rozkladu booleovských výrazov pomocou Shannonovej dekompozície. Popri vytváraní stromu sme ho zároveň aj redukovali, celý proces sme vytvárali pomocou rekurzie. Po vytvorení diagramu sme ho dôkladne otestovali na náhodne vytvorených funkciách, ktorých bolo vždy 100 a z výsledkov sme spravili aritmetický priemer. Výsledné údaje sme zapísali do tabuľky a vytvorili grafy redukcií a času trvania funkcie BDD use. Pri každom zavolaní BDD use porovnávame výsledok s priamou evaluáciou funkcie. Ak sa výsledok nerovná, vypíše sa táto informácia do konzole. Keďže sa táto správa nevypisuje, môžeme predpokladať, že vytvorenie BDD stromu prebehlo úspešne a BDD use teda vracia správny výsledok.