Zadanie č.2*

Hlib Kokin ID: 117991
Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológií
xkokin@stuba.sk

08.05.2022

*Druhé zadanie v predmete Dátové štruktúry a algoritmy, ak. rok 2021/22, vedenie: Peter Lehoczký

Abstrakt

V svojom zadanie som sa implementoval parse funkciu(1), metodu bddCreate(3) z redukovanim stromu po jeho vytvaraní, bddUse (4), a testovaci program ktory otestuje kod do 13 premennych.

Najprv v dokumentácii zvážime postup parsing funkcie, ďalej - bddCreate a redukovanie stromu, potom postup funkcie bddUse a prejdeme k testom a testovacím údajom(9), z výsledkov ktorých bude možné vyvodiť definitívny záver.

Obsah

Dokumentaciá zadania č.2

1 Úvod

Binárne diagramy sú usporiadané podobným spôsobom ako binárne stromy. Každý má určitú hĺbku, ktorá závisí od počtu premenných v booleovskej funkcii. Search na takomto sdiagrame sa vykonáva pohybom nadol doľava alebo doprava, v závislosti od hodnoty premennej.

2 Parse funkcia

Funkcia parsovania v mojom projekte prekladá výraz z formy DNF do binárnej postupnosti núl a jednotiek, na základe ktorej sa potom vytvorí strom a vyplní sa funkcia bddCreate .

2.1 Postup

Najprv si vo funkcii prejdeme celý riadok, zapíšeme si celý počet premenných a vygenerujeme ich hodnoty, tieto hodnoty vložíme do tabuľky, aby sme s nimi mohli neskôr pracovať. Potom vždy, keď narazíme na otvorenú zátvorku, presunieme sa na ďalšiu bunku v tabuľke. V každej ďalšej bunke tabuľky po zapísaných premenných vykonáme logické operácie AND pre každú dvojicu zátvoriek. Po tom, čo sa dostaneme na koniec našej funkcie, postupne každú vytvorenú bunku pridáme do poslednej okrem tých, v ktorých máme zapísané premenné a takto preložíme funkciu z DNF tvaru

(A.!B)+(C.B)

0	A	00001111
1	B	00110011
2	C	01010101
3	(A.!B)	00001100
4	(C.B)	00010001
5	(A.!B)+(C.B)	00011101

Obr. 1: Diagram na pochopenie parsingu

1. Výgenerovať hodnoty pre premenné a zapisať ich do tabuľky.
2. Vykonáť všetky logické operácie AND v zátvorkách
3. Vykonať všetky logické operácie OR na predchádzajúcich bunkách tabuľky

3 bddCreate

Vo funkcii Create rekurzívne vytvoríme samotný strom a nastavíme ukazovatele poslednych odkazov na 1 a 0 postupne od začiatku do konca, podľa vektora, ktorý sme dostali po parsovaní.

```
bdd* bddInit(bdd* head,bdd* one, bdd* zero, int vars, int cnt, int* red, bdd* parent, int right,unsigned char* seq, int* pos) {
    if (cnt == vars) {      // posledne odkazy nastavíme na odkazy '0' alebo '1' podla vektora a ho pozicii
                            //init je rekurzívne spraveni tak kazdy odkaz bude nastaveny postupne
        if (seq[*pos] == '1') {
            (*pos) ++;
            return one;
        }
        else if (seq[*pos] == '0') {
            (*pos) ++;
            return zero;
        }
    }
    if (head == NULL) {
        head = (bdd*)calloc(1, sizeof(bdd));
        head->index = '2';
        head->parent = parent;
        head->isRight = right;
        cnt++;
        printf("node has been created\n");
    }
    head->VarCnt = vars;
    head->NodeCnt = red;
    head->left = bddInit(head->left,one,zero, vars, cnt, red, head, 0,seq, pos);
    head->right = bddInit(head->right,one,zero, vars, cnt, red, head, 1,seq, pos);
    return head;
```

Obr. 2: Screenshot zdrojoveho kódu bddInit

Potom zredukujeme strom zavolaním funkcie bdd ReduceR

```
bdd* bddCreate(bf* func) {
    int red = 0;
    int pos = 0;
    printf("func vars %d\n", func->vars);

    bdd* head = NULL;
    bdd* one = (bdd*)calloc(1, sizeof(bdd));
    if(one) one->index = '1';
    bdd* zero = (bdd*)calloc(1, sizeof(bdd));
    if(zero) zero->index = '0';

    //najprv vytvorime ten strom
    head = bddInit(head,one,zero, func->vars, 0, &red, NULL, -1, func->seq, &pos);
    red = (pow(2, head->VarCnt) + 1);
    printf("amount of nodes = %d\n", red);

    if (head != NULL) printf("Var Cnt %d\n", head->VarCnt);
    //potom reduce
    bddReduceR(head, &red);
    printf("reduced to %d nodes\n", red);

    return head;
```

Obr. 3: ScreenShot zdrojoveho kodu bddCreate

3.1 Redukovanie

Pomocou rekurzie sa postupne presúvame na spodné odkazy zľava doprava, až kým nenájdeme odkaz, v ktorom oba ukazovatele ukazujú na rovnaký odkaz dvoch: one a zero s hodnotami 1 a 0.

1. Keď stretneme odkaz s rovnakými ukazovateľmi - skontrolujeme, ktorý syn je tento odkaz pre jeho rodiča, vľavo alebo vpravo.

2. Ak vľavo, zmeníme ľavého syna rodiča na ľavého syna nášho odkazu

3. Ak vpravo - podobne zmeňte pravého syna rodiča na pravého syna odkazu

```
void bddReduceR(bdd* head, int* cnt){
    //ak sme dostali do tych poslednych odkazov tak dalej uz nic nemame redukovat
    if (head->index == '1' || head->index == '0') return;

    if (head->right == head->left) {
        int t = compareNodes(head);
        (*cnt) -= t;
        return;
    }
    else {
        bddReduceR(head->left, cnt);    //ak nenasli par rovnakych synov - rekurzivne pokrocujeme
        bddReduceR(head->right, cnt);
        int t = compareNodes(head);
        (*cnt) -= t;
        return;
    }
}
```

Obr. 4: ScreenShot zdrojoveho kodu bddReduceR

4. Uvoľnime pamäť odkazu
5. Znížime počítadlo počtu odkazov
6. Ukoncime funkciu a vratime sa do rekurzie na vyššie odkazý

```
int compareNodes(bdd* head) {
    int cnt = 0;
    if (head->left == head->right) {
        head->right = NULL;
        printf("pair detected: sons index is %c\n", head->left->index);
        if (head->isRight == 1) {
            head->parent->right = head->left;
            printf("pair detected; reducing right \n");
            free(head);
            cnt++;
        }
        else if(head->isRight == 0) {
            head->parent->left = head->left;
            printf("pair detected; reducing left\n");
            free(head);
            cnt++;
        }
        else if (head->isRight == -1) {
            head = head->left;
            printf("pair detected; reducing root\n");
            cnt++;
        }
    }
    return cnt;
}
```

Obr. 5: Zdrojový kód funkcie compareNodes

4 bddUse: Vyhľadavanie odkazu

Na nájdenie odkazu som zvolil cyclus pre počet opakovaní rovnajúci sa počtu premenných (hĺbka).

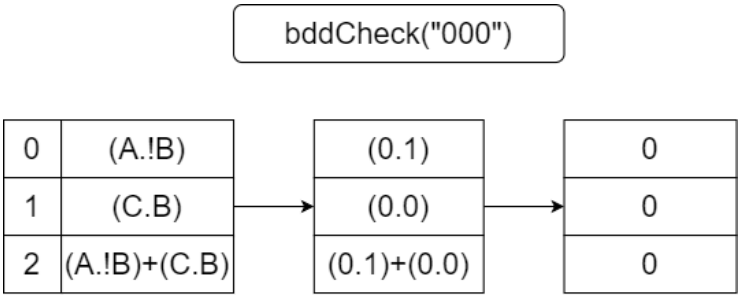
V závislosti od aktuálneho znamienka v reťazci s hodnotami premenných, ktoré boli funkcii odovzdané, sa posunieme doľava - ak 0, doprava - ak 1. Pôjdeme nadol, kým sa cyklus neskončí alebo kým ďalší ukazovateľ neukáže na NULL, čo bude znamenať, že sme dosiahli odkaz 1 alebo 0.

```
char bddUse(bdd* head, unsigned char* values) {
    bdd* temp = head;
    for (int i = 0; i < strlen(values); i++) {
        if (temp->left == NULL)break;
        if (values[i] == '0')temp = temp->left;
        else if (values[i] == '1')temp = temp->right;
    }
    return temp->index;
}
```

Obr. 6: Zdrojový kód funkcie compareNodes

5 Kontrola správnosti výsledku vyhľadávania

bddCheck - funkcia, ktorá funguje podobne ako parsing funkcia, len namiesto písania a generovania kódu premenných ich mení na hodnoty, ktoré sa tam odovzdávajú. A potom rovnakým spôsobom vykoná logické operácie a ako odpoveď dostane jeden znak



Obr. 7: Diagram na pochopenie funkcie bddCheck

Na kontrolu všetkých možných prípadov som vytvoril cyclus pre každé číslo od 0 do 2 stupna var, kde var je počet premenných v booleovskej funkcii. A pri každom spustení cyklu preložím číslo do binárnej formy a zavolám funkcie bddUse a bddCheck pre neho. Ak je výsledok funkcií rovnaký, potom je všetko správne nastavene

6 Testovanie

Testovanie sa uskutočnilo vytvorením stromu, stanovením počtu jeho uzlov, jeho redukovanim čo najviac, opätovným stanovením počtu jeho uzlov a následnou kontrolou všetkých možných možností pomocou funkcií bddCheck a bddUse pre

```c
unsigned char* toBinary(int num, int v) {
    unsigned char* res = (unsigned char*)calloc(v, sizeof(unsigned char));
    if (num > pow(2, v)) {
        printf("number is out of limits\n");
        return;
    }
    else {
        for (int i = 1; i <= v; i++) {
            if (num % 2 == 1)res[v - i] = '1';
            else if (num % 2 == 0)res[v - i] = '0';
            num /= 2;
        }
        return res;
    }
}
```

Obr. 8: Zdrojový kód funkcie toBinary

```c
int main() {
    char check;
    char ch;
    bf* func = parseFunc("(!A.B)+(C.D)");
    bdd* head = bddCreate(func);
    printf("\n\n");
    unsigned char* con = NULL;

    for (int i = 0; i < pow(2, func->vars); i++){

        con = toBinary(i, func->vars);
        printf("to binary : %s\n", con);
        check = bddCheck(con, func);
        ch = bddUse(head, con);
        if (check != ch) printf("result must be %c, but it is %c\n", check, ch);
        else printf("results are correct\n");
    }
    return 0;
}
```

Obr. 9: Zdrojový kód funkcie testu

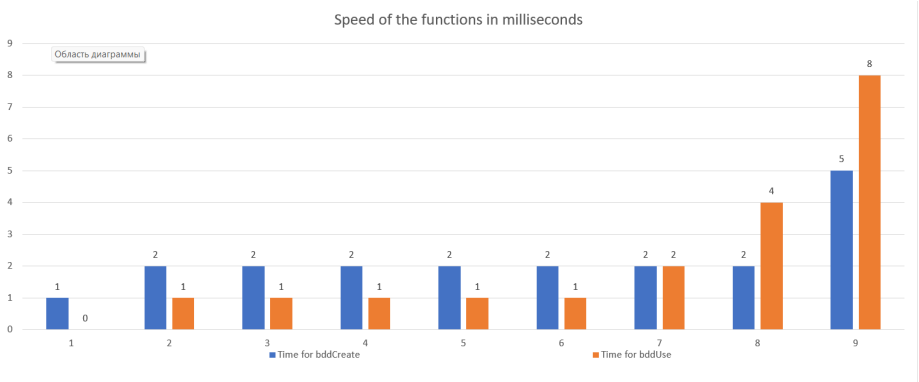
každý možný počet premenných od 1 do 15. Pre každý určitý počet premenných som náhodne vygeneroval 7 booleovských funkcií. Celkovo bolo počas testovania vytvorených a testovaných 105 diagramov

6.1 Test: Vytvorenie a overenie

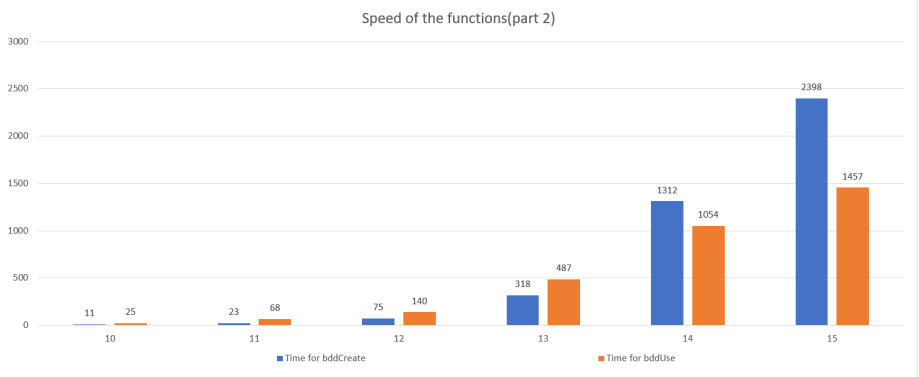
Grafy (10 11) ukazuje, že čím väčšia je hĺbka grafu(počet premenných), tým dlhšie trvá jeho vytvorenie a kontrola. Počas testu bola zaznamenaná pravidelnosť, ktorá spočíva v tom, že rýchlosť overovania závisí nielen od hĺbky diagramu, ale aj od počtu redukovaných uzlov, pretože zmenšenie diagramu môže ovplyvniť jeho hĺbku. V tabuľke so štatistikami (14)hodnoty boli prevzaté z niekoľkých testov.

6.2 Test: Redukovanie

Na tychto grafach (12 13) je zrejmé, že čím viac premenných, tým viac počiatočných uzlov. Súdiac podľa algoritmu redukcie diagramu, môžeme povedať, že počet redukovaných uzlov priamo závisí od vlastností booleovskej funkcie. Z tabuľky so všetkými štatistikami je možné zdôrazniť vzorec rastu percenta skrátených buniek v závislosti od počtu premenných, počnúc 5 premennými sa percento zvýšilo nad 80 percent a rástlo, keď pre 15 premenných dosiahlo 97 per-



Obr. 10: Test: Vytvotenie a overenie



Obr. 11: Test: Vytvotenie a overenie

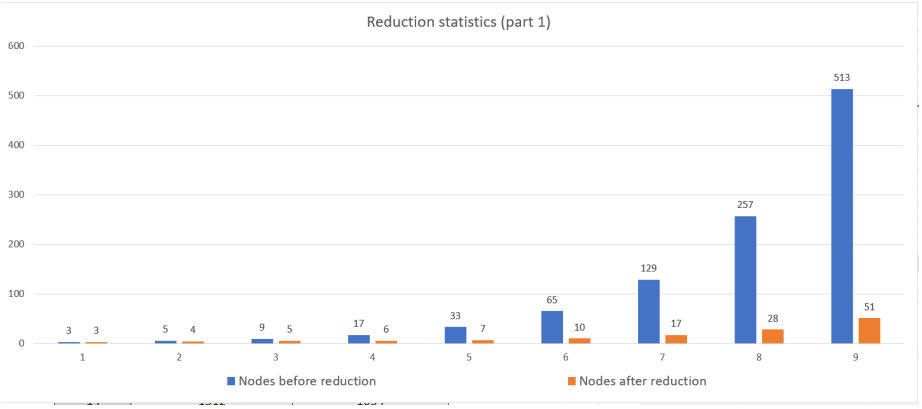
cent. Bude zrejmé, že po 15 premenných bude percento naďalej rásť a priblížiť sa k 100.

Nižšie je tabuľka (14), ktorá zaznamenáva presné hodnoty zo všetkých testov v projekte.

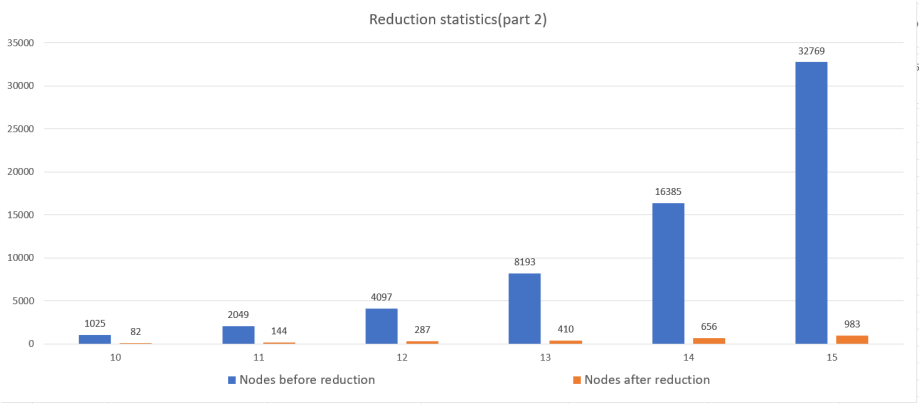
7 Záver

Na záver je dôležité spomenúť, že rýchlosť vykonávania funkcií pre diagram závisí v prvom rade od jeho hĺbky.

Po zmenšení grafu bude rýchlosť jeho kontroly parity závisieť nielen od hĺbky, ale aj od toho, do akej miery sa graf zmenšil, pretože zmenšenie ovplyvňuje hĺbku určitých sekvencií.



Obr. 12: Test: Redukovanie



Obr. 13: Test: Redukovanie

Variables	Time for bddCreate	Time for bddUse	Nodes before reduction	Nodes after reduction	Reduction percentage
1	1	<1	3	3	0%
2	2	1	5	4	29%
3	2	1	9	5	51%
4	2	1	17	6	66%
5	2	1	33	7	83%
6	2	1	65	10	85%
7	2	2	129	17	87%
8	2	4	257	28	89%
9	5	8	513	51	90%
10	11	25	1025	82	92%
11	23	68	2049	144	93%
12	75	140	4097	287	93%
13	318	487	8193	410	95%
14	1312	1054	16385	656	96%
15	2398	1457	32769	983	97%

Obr. 14: Tabuľka s testovacími hodnotami