Slovenská technická univerzita v Bratislave

Fakulta informatiky a informačných technológií

Dátové štruktúry a algoritmy

Zadanie č. 3 – Binárne rozhodovacie diagramy

Martin Rudolf, 97029, ak. rok 2020/2021

Obsah

Opis Riešenia 3
BDD Create 3
BDD Use 5
Spôsob Testovania 5
Dosiahnuté výsledky 6
Príloha A – Zadanie č. 3 Binárne rozhodovacie diagramy 7
Príloha B – Upresnenie a hodnotenie zadania 10

Martin Rudolf, 97029, ak. rok 2020/2021

Opis Riešenia

V tomto zadaní sme sa venovali Binárnym rozhodovacím diagramom (BDD). Implementácia zadania je napísaná v jazyku C a je rozdelená do 2 zdrojových súborov (file.c) a jedného hlavičkového súboru (file.h). Zdrojový súbor na testovanie implementovaného BDD je pomenovaný main.c ktorý includuje hlavičkový súbor BDD.h ako aj samotnú main() funkciu a potrebné funkcie na testovanie popísané nižšie. Zdrojový súbor implementácie BDD nesie názov BDD.c v ktorom sú funkcie BDD_create() a BDD_use().

BDD Create

Vstupným argumentom funkcie je štruktúra ktorá nesie v sebe vektor prislúchajúcej Boolovskej funkcie (BF) a počet premenných BF. Metódou zhora nadol algoritmus funkcie vytvára potomkov koreňa, a to rekurzívnym volaním kde spodok rekurzie je štruktúra s vektorom o veľkosti 1. Funkcia najprv inicializuje štruktúru uzla diagramu vloží doň vektor zo vstupnej Boolovskej funkcie a následne tento vektor rozloží na polovicu. Ak je veľkosť vektora vľavo a vpravo väčšia ako 1 tak sa funkcia rekurzívne zavolá najprv nad ľavou polovicou, potom nad pravou. Inak sa algoritmus dostal na spodok rekurzie, inicializujú sa najmladší potomkovia priradí sa im výsledná hodnota vektoru (0 alebo 1) a vráti štruktúru ktorá nesie rodiča. Ak sa ukončia všetky rekurzívne vetvy, a teda vytvoria sa všetci potomkovia, aktualizuje sa štruktúra diagramu ktorá nesie adresu koreňa, informáciu o počte potomkov a členov Boolovskej funkcie, čo je aj návratová hodnota funkcie BDD_create.

Takáto implementácia BDD má exponenciálnu zložitosť, a problém nastáva už pri n > 20, kde n je počet premenných BF pretože pridanie ďalšej premennej zväčší BDD o ďalšiu úroveň. Veľkosť vektora = 2^n.

```
if (strlen(vec_l) == 1 && strlen(vec_r) == 1) {
    root->one = (struct BDD_node*)malloc(sizeof(struct BDD_node));
    root->one->vector = (char*)malloc(sizeof(char));
    root->zero = (struct BDD_node*)malloc(sizeof(struct BDD_node));
    root->zero->vector = (char*)malloc(sizeof(char));

    strcpy(root->zero->vector, vec_l);
    strcpy(root->one->vector, vec_r);
    root->zero->zero = NULL;
    root->zero->one = NULL;
    root->one->zero = NULL;
    root->one->one = NULL;
    function->node_sum += 2;
    diagram->root = root;
    return diagram;
}
```

Figure 1 Spodok rekurzie

```
else {
    function->vector = vec_l;
    struct BDD* zerooo = create_BDD(function);
    root->zero = zerooo->root;
    function->vector = vec_r;
    struct BDD* oneee = create_BDD(function);
    root->one = oneee->root;
}
```

Figure 2 Rekurzivne volania

```
struct BDD_node {
    struct BDD_node* one;
    struct BDD_node* zero;
    char* vector;
};
```

Figure 3 Štruktúra jedného uzla diagramu

```
struct BDD {
    int var_sum;
    int node_sum;
    struct BDD_node* root;
};
```

Figure 4 Štruktúra BDD

```
struct BF {
    int var_sum;
    int node_sum;
    char* vector;
};
```

Figure 5 Štruktúra Boolovskej funkcie

```
for (int i = 0; i < vector_len; i++) {

    if (i < vector_len / 2) {
        vec_l[i] = vector[i];
    }

    else {
        if (i == size) {
            vec_l[i] = '\0';
        }
        vec_r[j++] = vector[i];
    }
}
```

Figure 6 rozdelenie vektora na pravú a ľavú stranu

BDD Use

Vstupným argumentom je štruktúra ktorá nesie adresu koreňa BDD a pole znakov reprezentujúcich kombináciu premenných Boolovskej funkcie. Funkcia prechádza daný BDD od koreňa po list cestou definovanou vstupnou kombináciou premenných. Návratová hodnota je char ktorý reprezentuje výsledok Boolovskej funkcie. Algoritmus pozerá na dané indexy vstupného poľa, ak hodnota na i-tom indexe je 1 prejdeme do pravého (pointer one v štruktúre BFF_node) potomka ak 0 tak do ľavého potomka (pointer zero v štruktúre BFF_node), takto algoritmus prejde celý BDD a ak sa dostal k listu vráti char ktorý sa v ňom nachádza (0 alebo 1) ak nie sme v liste a došiel na koniec vstupného poľa tak vráti char -1.

```
struct BDD_node* akt = bdd->root;
for (int i = 0; i < len; i++) {
    if (vstupy[i] == '1') {
        akt = akt->one;
    }
    else {
        akt = akt->zero;
    }
}
if (strlen(akt->vector) > 1) {
    return '-1';
}
return akt->vector[0];
```

Spôsob Testovania

Keďže sa nepodarilo implementovať funkciu BDD_reduce, tak nám ostalo pozorovať len veľkosť BDD respektíve počet uzlov v diagrame s daným počtom premenných a čas na vytvorenie príslušného diagramu.

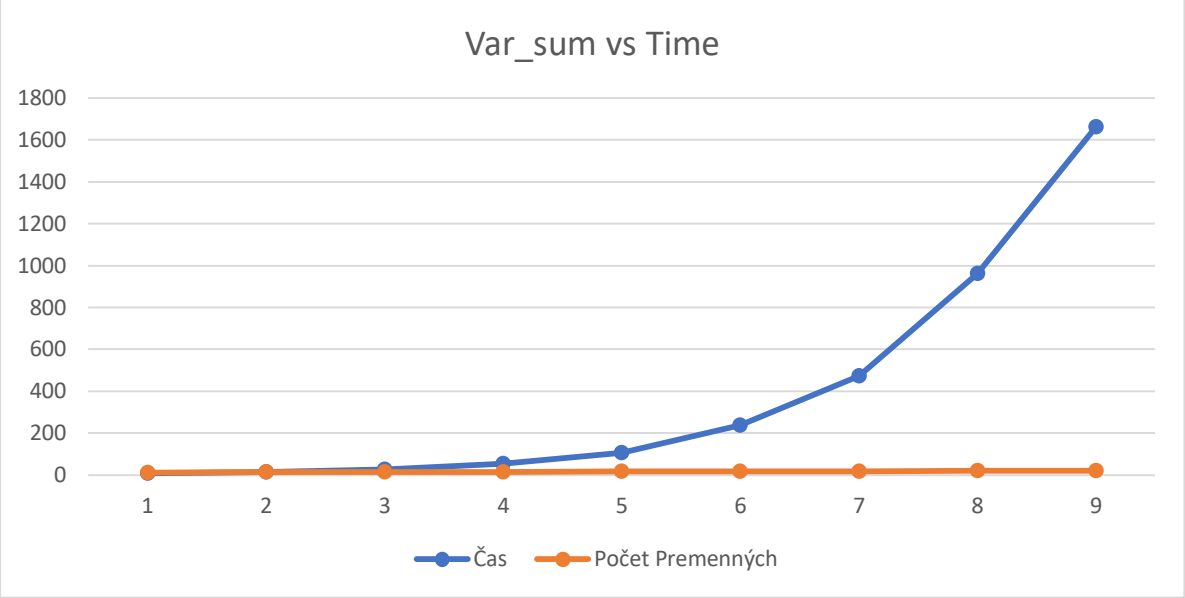
Na testovanie používame funkciu generate_vector() ktorá slúži na vygenerovanie vektora pre daný počet premenných. Vstupným argumentom je n, a teda počet premenných boolovskej funkcie. Výstupom funkcie je pole charov reprezentujúci vektor pre príslušné n, ktorého veľkosť sa vypočíta pomocou funkcie pow() ktorá počíta mocniny pre dané parametre, v našom prípade pow(2, n). Následne funkcia prechádza cyklom od 0 po veľkosť vektora a priradzuje vektoru náhodné hodnoty 0, 1.

Samotné testovanie sa odohráva vo funkcií main kde sa v cykle pre počet premenných BF od 12 do 20 generujú vektory a pre dané vektory sa vytvárajú príslušné BDD, meria sa čas vytvorenia BDD a sledujeme počet uzlov v diagrame pre daný počet premenných.

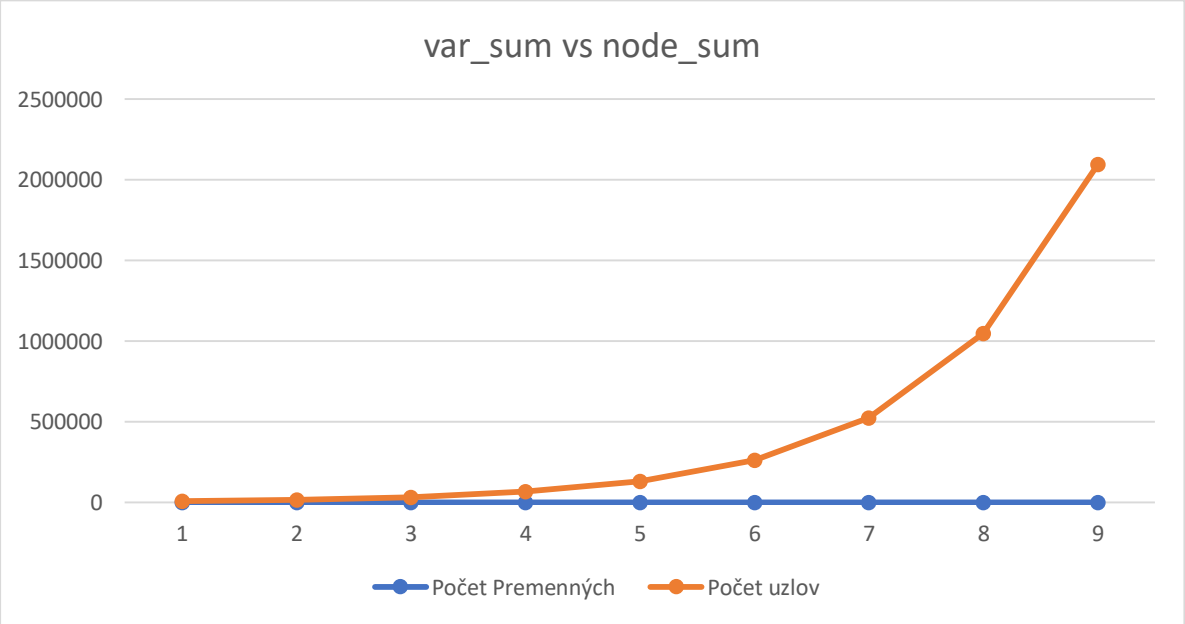
```
for (int i = 12; i <= 20; i++) {
    ftime(&start);
    char* vector = generate_vector(i);
    struct BF* function = (struct BF*)malloc(sizeof(struct BF));
    function->var_sum = i;
    function->node_sum = 0;
    function->vector = vector;
    struct BDD* root = create_BDD(function);
    ftime(&end);

    diff = (int)(1000.0 * (end.time - start.time) + (end.millitm - start.millitm));
    printf("v case: %dms, s poctom premennych: %d bolo vytvorenych %d uzlov\n", diff, root->var_sum, root->node_sum);
```

Dosiahnuté výsledky

Namerané výsledky nám potvrdili exponenciálny rast času aj pamäťového priestoru pre príslušné počty premenných BDD. Viď graf.



Obrázok 1 graf pre zobrazenie exponencionálnosti algoritmu pre počet premenných 12-20



Obrázok 2 graf pre zobrazenie exponencionálnej pamäťovej náročnosti pre počet premenných 12-20.

```
v case: 8ms, s poctom premennych: 12 bolo vytvorenych 8191 uzlov
v case: 13ms, s poctom premennych: 13 bolo vytvorenych 16383 uzlov
v case: 25ms, s poctom premennych: 14 bolo vytvorenych 32767 uzlov
v case: 51ms, s poctom premennych: 15 bolo vytvorenych 65535 uzlov
v case: 146ms, s poctom premennych: 16 bolo vytvorenych 131071 uzlov
v case: 239ms, s poctom premennych: 17 bolo vytvorenych 262143 uzlov
v case: 515ms, s poctom premennych: 18 bolo vytvorenych 524287 uzlov
v case: 976ms, s poctom premennych: 19 bolo vytvorenych 1048575 uzlov
v case: 1691ms, s poctom premennych: 20 bolo vytvorenych 2097151 uzlov
```

Figure 7konzolový výstup z testovacieho súboru

Príloha A – Zadanie č. 3 Binárne rozhodovacie diagramy

Zadanie 3 – Binárne rozhodovacie diagramy

Martin Rudolf, 97029, ak. rok 2020/2021

Vytvorte program, ktorý bude vedieť vytvoriť, redukovať a použiť dátovú štruktúru BDD (Binárny Rozhodovací Diagram) so zameraním na využitie pre reprezentáciu Booleovských funkcií. Konkrétne implementujte tieto funkcie:

- BDD *BDD_create(BF **bfunkcia*);
- int BDD_reduce(BDD **bdd*);
- char BDD_use(BDD **bdd*, char **vstupy*);

Samozrejme môžete implementovať aj ďalšie funkcie, ktoré Vám budú nejakým spôsobom pomáhať v implementácii vyššie spomenutých funkcii, nesmiete však použiť existujúce funkcie na prácu s binárnymi rozhodovacími diagramami.
Funkcia **BDD_create** má slúžiť na zostavenie úplného (t.j. nie redukovaného) binárneho rozhodovacieho diagramu, ktorý má reprezentovať/opisovať zadanú Booleovskú funkciu (vlastná štruktúra s názvom BF), na ktorú ukazuje ukazovateľ *bfunkcia*, ktorý je zadaný ako argument funkcie **BDD_create**. Štruktúru BF si definujete sami – podstatné je, aby nejakým (vami vymysleným/zvoleným spôsobom) bolo možné použiť štruktúru BF na opis Booleovskej funkcie. Napríklad, BF môže opisovať Booleovskú funkciu ako pravdivostnú tabuľku, vektor, alebo výraz. Návratovou hodnotou funkcie **BDD_create** je ukazovateľ na zostavený binárny rozhodovací diagram, 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.
Funkcia **BDD_reduce** má slúžiť na redukciu existujúceho (zostaveného) binárneho rozhodovacieho diagramu. Aplikovaním tejto funkcie sa nesmie zmeniť Booleovská funkcia, ktorú BDD opisuje. Cieľom redukcie je iba zmenšiť BDD odstránením nepotrebných (redundantných) uzlov. Funkcia **BDD_reduce** dostane ako argument ukazovateľ na existujúci BDD (*bdd*), ktorý sa má redukovať. Redukcia BDD sa vykonáva priamo nad BDD, na ktorý ukazuje ukazovateľ *bdd*, a preto nie je potrebné vrátiť zredukovaný BDD návratovou hodnotou (na zredukovaný BDD bude totiž ukazovať pôvodný ukazovateľ *bdd*). Návratovou hodnotou funkcie **BDD_reduce** je číslo typu int (integer), ktoré vyjadruje počet odstránených uzlov. Ak je toto číslo záporné, vyjadruje nejakú chybu (napríklad ak BDD má ukazovateľ na koreň BDD rovný NULL). Samozrejme, funkcia **BDD_reduce** má aktualizovať aj informáciu o počte uzlov v BDD.
Funkcia **BDD_use** má slúžiť na použitie BDD pre zadanú (konkrétnu) kombináciu vstupných premenných Booleovskej funkcie a zistenie 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 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 poľa charov (bajtov). Práve toto pole charov/bajtov reprezentuje nejakým (vami zvoleným) spôsobom konkrétnu kombináciu 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

Martin Rudolf, 97029, ak. rok 2020/2021

BDD_use je char, ktorý reprezentuje výsledok Booleovskej funkcie – je to buď ‘1’ alebo ‘0’. V prípade chyby je táto návratová hodnota záporná, podobne ako vo funkcii **BDD_reduce**.
Okrem implementácie samotných funkcií na prácu s BDD je potrebné vaše riešenie dôkladne otestovať. Vaše riešenie musí byť 100% korektné. V rámci testovanie je potrebné, aby ste náhodným spôsobom generovali Booleovské funkcie, podľa ktorých budete vytvárať BDD pomocou funkcie BDD_create. Vytvorené BDD následne zredukujete funkciou BDD_reduce a nakoniec overíte 100% funkčnosť zredukovaných BDD opakovaným (iteratívnym) volaním funkcie BDD_use tak, že použijete postupne všetky možné kombinácie vstupných premenných. Počet premenných v rámci testovania BDD by mal byť minimálne 13. Počet Booleovských funkcií / BDD diagramov by mal byť minimálne 2000. V rámci testovania tiež vyhodnocujte percentuálnu mieru zredukovania BDD (t.j. počet odstránených uzlov / pôvodný počet uzlov).
Príklad veľmi jednoduchého testu (len pre pochopenie problematiky):

```
#include <string.h>
int main(){
BDD* bdd;
bdd = BDD_create("AB+C"); // alebo vektorom BDD_create("01010111")
BDD_reduce(bdd);
if (BDD_use("000") == '1')
printf("error, for A=0, B=0, C=0 it should be 0.\n");
if (BDD_use("001") == '0')
printf("error, for A=0, B=0, C=1 it should be 1.\n");
if (BDD_use("010") == '1')
printf("error, for A=0, B=1, C=0 it should be 0.\n");
if (BDD_use("011") == '0')
printf("error, for A=0, B=1, C=1 it should be 1.\n");
if (BDD_use("100") == '1')
printf("error, for A=1, B=0, C=0 it should be 0.\n");
if (BDD_use("101") == '0')
printf("error, for A=1, B=0, C=1 it should be 1.\n");
if (BDD_use("110") == '0')
printf("error, for A=1, B=1, C=0 it should be 1.\n");
if (BDD_use("111") == '0')
printf("error, for A=1, B=1, C=1 it 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. 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 algoritmu. 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 15 bodov, **nutné minimum je 6 bodov**.

Za implementáciu riešenia (3 funkcie) je možné získať celkovo 8 bodov, z toho 2 body sú za funkciu **BDD_create**, 4 body za funkciu **BDD_reduce** a 2 body za funkciu **BDD_use**. Pre úspešné odovzdanie implementácie musíte zrealizovať aspoň funkcie **BDD_create** a **BDD_use**. Za testovanie je možné získať 4 body (treba podrobne uviesť aj v dokumentácii) a za dokumentáciu 3 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)**.

Príloha B – Upresnenie a hodnotenie zadania

Opis riešenia

... pár slov na úvod o BDD, o tom ako rámcovo funguje Vami vytvorené dielo

BDD Create

- popis Boolovskej funkcie v rámci ktorej ste navrhovali samotnú funkciu a aký postup ste zvolili,
- popis štruktúry BF
- časová zložitosť, výňatky kódu, a pod. ...

BDD Reduce

- popis samotnej funkcie vrátane výňatkov kódu,
- časová zložitosť,
- obdobne ako pri BDD Create a pod. ...

BDD Use

- obdobne ako pri predchádzajúcich dvoch funkciách,
- kladťe dôraz na zdokumentovanie overenia funkčnosti,
- má slúžiť na použitie BDD pre zadanú (konkrétnu) kombináciu vstupných premenných Booleovskej funkcie a zistenie výsledku Booleovskej funkcie pre túto kombináciu vstupných premenných.

Ak máte pomocné funkcie

- dôvod, opis funkčnosti

Spôsob testovania

- stručne opísať vytvorený testovací program,
- počet premenných v rámci testovania min. 13 - max. 20,
- počet Booleovských funkcií min. 2000,
- **Vyžaduje sa tiež odovzdanie programu, ktorý slúži na testovanie a odmeranie efektívnosti týchto implementácií ako jedného samostatného zdrojového súboru (obsahuje funkciu main).**

Martin Rudolf, 97029, ak. rok 2020/2021

Dosiahnuté výsledky

- priemerný čas vykonania Vašich funkcií,
- priestorová náročnosť,
- percentuálna miera zredukovania.
- z toho:

    - BDD_create – 02 body
    - BDD_reduce – 04 body
    - BDD_use – 02 bod
    - testovací program – 04 body
    - dokumentácia – 03 body
        - opis riešenia
        - spôsob testovania
        - dosiahnuté výsledky
    - **celkom: – 15 bodov**

Poznámky:

- je nutné prezentovať funkčnú implementáciu min. na úrovni BDD_create a BDD_use,
- nutné minimum 6 bodov.

Martin Rudolf, 97029, ak. rok 2020/2021