FAKULTA INFORMATIKY A INFORMAČNÝCH TECHNOLÓGIÍ

SLOVENSKÁ TECHNICKÁ UNIVERZITA

Ilkovičova 2, 842 16 Bratislava 4

2021/2022

Dátové štruktúry a algoritmy

Zadanie č.2 – Binárny rozhodovací diagram

Cvičiaci: MSc. Mirwais Ahmadzai

Čas cvičení: Štvrtok 11:00 - 12:50

Vypracoval: Maroš Bednár

AIS ID: 116822

Obsah

1. Úvod

1.1 Čo je to BDD

Bdd je skratka pre Binárny Rozhodovací Diagram. Je to dátová štruktúra, ktorá rýchlo a spoľahlivo spracováva boolove dáta. My tak vieme o tvrdení rozhodnúť, či je pravdivé alebo nepravdivé pomerne rýchlo a s použitím málo systémovej pamäte.

1.2 Stručný opis programu

V programe ide o zostrojenie BDD, ktorý využíva čo najmenej pamäte a spotrebuje malé množstvo času. Docielime to postupným upravovaním boolovych funkcii pomocou shanonovej dekompozície a následným redukovaním prebytočných funkcií za behu programu.

Na začiatku sa vygeneruje funkcia, ktorá sa následne rôzne redukuje až sa vybuduje BDD. Od tohto momentu nám stačí do BDD dosadiť istú sekvenciu núl a jednotiek a my tak zistíme, či je funkcia pravdivá alebo nie.

Takýto postup sa v programe zopakuje 100x pre zadaný počet premenných boolovej funkcie a na konci sa vypíšu potrebné informácie o diagrame. Niektoré sa taktiež zapisujú postupne za behu programu do súboru results.txt

Program a všetký potrebne dokumenty sú:

Assignment2_BDD.c – obsahuje main funkciu
chainingHash.c
chainingHash.h
Statistics.xlsx

- **Testovací program je priamou súčasťou Assignment2_BDD.c**

2. Opis štruktúr a funkcií

2.1. Opis štruktúr

```
typedef struct bdd_node {
	char letter;
	char function[MAX_FUNCT_SIZE];
	struct bdd_node* lChild, * rChild;
	struct bdd_node* parent;
}BDD_NODE;
```

- Prvá štruktúra je bdd_node, ktorá slúži na uchovávanie informácie o aktuálnej bunke. Uchováva:
  - letter – znak, ktorý určuje meno aktuálnej bunky
  - function – pole, ktoré uchováva aktuálnu boolovu funkciu
  - lChild/rChild – smerník na ľavé/pravé dieťa
  - parent – smerník na rodiča

```
typedef struct bdd {
	char startingFunction[MAX_FUNCT_SIZE];
	int numOfVariables;
```

```
    int numOfNodes;
    struct bdd_node* root;
}BDD;
```

- Druhá štruktúra, ktorá uchováva informácie o koreňoch rôznych diagramov. Skladá sa z premenných:
  - startingFunction – uloženie pôvodnej neupravenej boolovej funkcie
  - numOfVariables – počet premenných v aktuálnom diagrame
  - numOfNodes – počet buniek v aktuálnom diagrame
  - root – ukazovateľ na prvý prvok upraveného booloveho rozhodovacieho diagramu

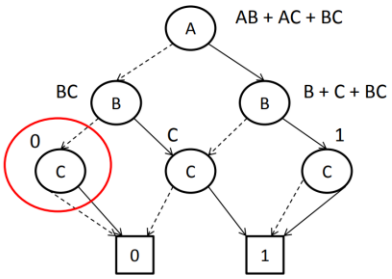
2.2 Opis funkcií

BDD_NODE* BDD_create(char* bfunction, char* order, BDD_NODE* parent);

- Funkcia vytvára celý diagram a vracia jeho začiatok. Funguje na princípe Shannonovej dekompozície, kedy vyberá z jednotlivých funkcií vždy aktuálny znak ktorý je v určitom poradí a následne podľa toho upravuje ľavú a pravú stranu funkcie.
- Ak sa funkcie rovnajú, je potrebné vykonať redukciu typu S odstránením aktuálnej bunky a nastavením dieťaťa rodiča na nasledujúci prvok, ktorý sa následne vytvorí. A rekurzívne vráti.

Príklad

- Y = AB + AC + BC

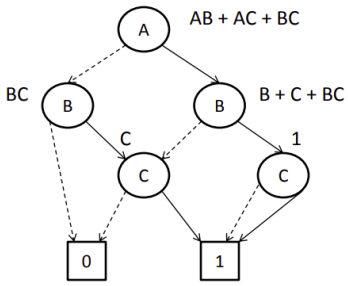


Redukcia typu S
- ľavý potomok je rovnaký ako pravý potomok (pointre sa rovnajú)
- špeciálny prípad – uzol reprezentuje konštantu (0)

Obrázok 1 – Redukcia typu S pred redukovaním

Príklad

- Y = AB + AC + BC

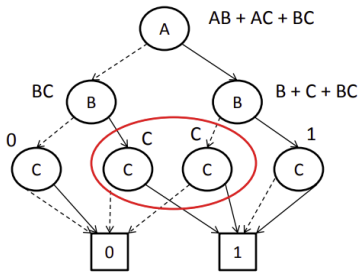


Obrázok 2 – Redukcia typu S po redukovaní

- Taktiež môže nastať situácia, kedy sa vytvorí prvok, ktorý už existuje v diagrame. Preto funkcia overuje hashovaciu tabuľku a postupne pridáva nové prvky aby si program zachoval čo najmenší počet buniek. Nastáva redukcia typu I. Hashovacia tabuľka vráti už existujuci prvok node a naň nastaví rodičov smerník na dieťa.

Príklad

- Y = AB + AC + BC

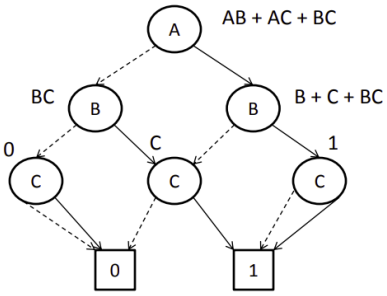


Redukcia typu I
- uzly reprezentujú rovnakú funkciu (C)
- ľavý potomok oboch uzlov je rovnaký a pravý potomok oboch uzlov je rovnaký

Obrázok 3 – Redukcia typu I pred redukovaním

Príklad

- Y = AB + AC + BC



- Keďže funkcia BDD_Create je rekurzívna funkcia, musí v nej byť podmienka overujúca znak 0 alebo 1. Ak sa boolova funkcia rovná jednému z nich, rekurzia končí a vracia odkaz na aktuálnu bunku.

`char BDD_use(BDD_NODE* bdd, char* inputs);`

- Funkcia slúžiaca na výpis pravdivostnej hodnoty binárneho diagramu. Zadané hodnoty inputs sú kombinácie jednotiek a núl, ktoré prideľujú pravdivostné hodnoty jednotlivým znakom.
- Funguje to cyklicky s podmienkou ukončenia. Ak funkcia narazí na znak 1 alebo 0 tak ho hneď vráti ako výslednú hodnotu boolovskej funkcie. Ak nenarazí, tak sa rozhoduje či bude prehľadávať ľavú alebo pravú stranu diagramu. Záleží to na aktuálnom znaku z argumentu inputs.

`char* fix_function(char function[MAX_FUNCT_SIZE], char letter, bool right_path);`

- Funkcia, ktorá vykonáva celé úpravy boolovej funkcie. Stará sa o rôzne úpravy ako sú:
  - Odstránenie duplicitných slov
  - Odstránenie duplicitných znakov v slove
  - Určenie pravdivostnej hodnoty funkcie
- Následne vráti odkaz na novú funkciu, ktorá je už v upravenom a takmer minimálnom tvare.
- Príklad:
- Vstupná funkcia: AaBCABa + BBCBd + ABA
- Úprava pre ľavú stranu a znak A by bola: BCd
- Úprava pre pravú stranu a znak A by bola: BCd + B

`char* generateBFunction();`

- Funkcia, ktorá náhodne vygeneruje boolovsku funkciu a následne ju vráti. Na začiatku sa vypýta počet premenných ktoré chceme využívať vo funkcii. Následne generuje náhodne počty slov a náhodné dĺžky slov, ktorým tiež náhodne priraďuje písmená zo zadanej abecedy (ak je vstup 5 tak abeceda bude A,B,C,D,E). Generuje taktiež veľké (negované) a veľké znaky.

char checkFunction(char* bFunction, char* order, char* values);

- Funkcia, ktorá dosadzovaním jednotiek a núl overuje správnosť riešenia. Funguje jednoduchým spôsobom. Najskôr si všetký písmená abecedy premení na postupnosť jednotiek a núl v DNF tvare. Následne overuje, či sa v jednotlivých slovách funkcie nachádzajú jednotky bez núl. Ak také slovo nájde, vráti pravdivostnú hodnotu 1. Ak nie, overuje ďalšie slová. V prípade nenájdenia slova s hodnotou 1, vráti 0.

3. Dôležité časti kódu

- Funkcia BDD_create overuje, či sa aktuálna funkcia nachádza v hashovacej tabuľke. Ak áno, tak ju vráti a rekurzia s ňou ďalej pracuje. V opačnom prípade vytvára novú boolovu funkciu, ktorú následne pripojí do hashovacej tabuľky. **Prebieha tu tak redukcia typu L.**

```
446     // if function exists, find it in the table and return its node
447     if (tmp_diagram = find(hashTable, bFunction, tableSize)) {
448         return tmp_diagram;
449     }
```

- Taktiež súčasťou funkcie BDD_create je táto časť. Tá nám vytvára upravené boolove funkcie v takmer minimálnom tvare a funkcia BDD_create tak dokaže s funkciou jednoducho pracovať.

```
457     // ease functions
458     strcpy(leftFunction, fix_function(diagram->function, order[0], false));
459     strcpy(rightFunction, fix_function(diagram->function, order[0], true));
460
```

- Opäť časť BDD_create, slúžiaca na redukciu diagramu za chodu programu v prípade, že sa obe strany upravenej funkcie rovnajú. Vtedy celá bunka zaniká a vracia sa jej potomok. **Tu prebieha redukcia typu S.**

```
475     // reduce the nodes which have pointer to the same value
476     if (!strcmp(leftFunction, rightFunction)) {
477         delete(hashTable, diagram->function, tableSize, &num_of_nodes);
478
479         if (diagram->parent == NULL) {
480             diagram = NULL;
481             diagram = BDD_create(leftFunction, tmpStr, diagram);
482         }
483         else {
484             return BDD_create(leftFunction, tmpStr, diagram);
485         }
486     }
```

4. Testovanie

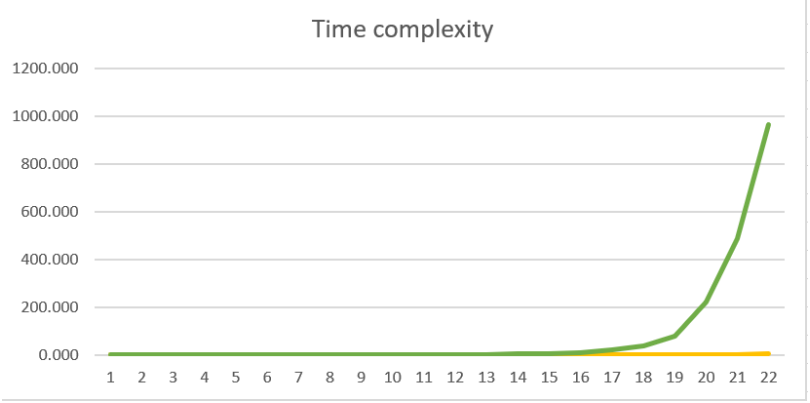
Testovanie diagramu prebieha v kóde. Užívateľ je povinný zadať počet premenných v rozmedzí 1-25. Následne program automaticky spustí 100 náhodne generovaných testov pre tento počet premenných na ktorého konci vypíše časove nároky na daný binárny diagram. Taktiež **dôležité údaje uloží do súboru results.txt**. Tento textový súbor tak umožňuje skontrolovanie správnosti programu.

Počas testovania sa testuje správnosť dvoma funkciami. Prvá je BDD_use a druhá checkFunction. Obe funkcie overujú výsledok, ale rôznymi spôsobmi. V prípade, že sa obe zhodujú, inkrementuje sa počet správnych výsledkov.

4.1 Časová a priestorová náročnosť

Pri **časovej náročnosti** je dôležité rozlišovať funkciu BDD_create a BDD_use. Časové náročnosti sú odlišné v oboch funkciách. Tabuľka 1 znázorňuje dáta zozbierané z testovania diagramu.

Number of variables	BDD_create time (s)	BDD_use time (s)	Average reduction
1	0.003	0.000	48.00%
2	0.004	0.000	67.75%
3	0.007	0.000	75.75%
4	0.019	0.000	78.50%
5	0.024	0.002	81.56%
6	0.029	0.007	86.77%
7	0.031	0.007	86.74%
8	0.075	0.010	90.25%
9	0.119	0.034	92.96%
10	0.190	0.080	94.17%
11	0.240	0.235	96.01%
12	0.369	0.475	97.35%
13	0.474	0.993	98.37%
14	1.088	1.917	98.89%
15	1.176	4.500	96.69%
16	1.062	8.344	98.33%
17	0.864	18.552	98.62%
18	0.910	34.334	98.91%
19	1.340	76.756	99.20%
20	1.886	219.523	99.49%
21	2.162	484.912	99.58%
22	2.842	962.568	99.16%



Tabuľka 1 – Časová náročnosť BDD a jeho percentuálna redukcia

Je viditeľné, že čím viac rôznych prvkov do diagramu vkladáme, tým viac času funkcie zaberajú a taktiež tým väčšia redukcia nastáva. Pri jednotlivých funkciách sú náročnosti nasledovné:

- BDD_create – exponenciálna O(2^n)
- BDD_use – exponenciálna O(2^(n/2))

Priestorová náročnosť, podobne ako časová, stúpa úmerne s počtom premenných. Extrémny prípad nastáva práve vtedy, keď funkcia má každý súčin iný. Napriek tomu je po redukovaní buniek menej, pretože shanonova dekompozícia využíva už existujúce bunky a spája ich dokopy s tými novými.

Problém nastáva pri väčších hodnotách. Časová aj priestorová náročnosť stúpa exponenciálne. Teda môže nastať zahltenie pamäte a tak aj vypnutie programu. Preto je odporúčané využívať počet premenných v rozmedzí 1-15.

Percentuálna redukcia rastie so zvyšujúcim sa počtom prvkov. Je to preto, že vzniká viacej shanonovych dekompozícií a tak aj šanca na nájdenie toho istého, už zredukovaného prvku sa zvyšuje.

5. Záver

Binárny rozhodovácí diagram využívame v prípade, ak pracujeme s veľkým množstvom dát v ktorých potrebujeme rozhodovať či budú postupy pravdivé alebo nie. Je to pomerne rýchly spôsob ako overiť pravdivosť, pretože sa neoverujú všetky bunky, ale len ich zlomok. Tento fakt je spôsobený shanonovými redukciami a prepájaním už existujúcich buniek s tými novými. Šetrí sa tak priestor aj čas.

Na záver chcem zhodnotiť moju implementáciu diagramu. Myslím si, že program je výkonný a spoľahlivý. Bol testovaný na veľkom množstve premenných bez zlyhania. Bez problému vypočíta funkciu zloženú z 20 premenných a 100 rôznych slov. Výsledky mojich testov sú zapísané v tabuľke 1 a tak vieme overiť ako sa program správa. Taktiež program za behu zapisuje všetky potrebné informácie do samostatného textového súboru a my tak vieme skontrolovať rôzne detaily správania sa tohto algoritmu. To ma privádza k záveru, že riešenie je správne.