Slovenská technická univerzita
Fakulta informatiky a informačných technológií
Dátové štruktúry a algoritmy

Zadanie 3 - Binárne rozhodovacie diagramy

Jakub Hrnčár
AIS ID: 110 800, ak. rok 2020/2021

1. Úvod do zadania

Úlohou zadania bolo vytvoriť funkčný binárny rozhodovací diagram, ktorý predstavuje ďalšiu významnú dátovu. Mali sme naprogramovať 3 algoritmy na prácu s touto dátovou štruktúrou: create, use a reduce.

Binárny rozhodovací diagram predstavuje reprezentáciu Boolovskej funkcie a vykonáva operácie nad ňou.

Create je najdôležitejší algoritmus, pretože vytvára samotnú dátovú štruktúru. Existuje viac spôsobov na vytváranie, ja som však zvolil istú kombináciu viacerých, ktorá bohužiaľ nie je tak efektívna ako som si myslel, že by mohla byť, ale za to je pomerne prehľadná.

Use používa iba jednoduchý algoritmus na iteratívne prehľadávanie diagramu, od koreňa až po list.

Reduce má vykonať minimalizáciu Boolovskej funkcie, avšak túto funkciu som sa rozhodol neimplementovať.

Kvôli testovaniu bolo nutné vytvoriť aj funkciu free, ktorá uvoľní celý diagram uvoľní.

2. Stručný opis algoritmu

1. **BDD_Create:**
   Táto funkcia pracuje rekurzívne a využíva spojenie vektorového princípu a princípu Boolovskej funkcie. V algoritme sa používajú dve štruktúry:
   UZOL - reprezentuje jeden uzol v diagrame/strome, má 3 atribúty

```
typedef struct uzol {
    char vstup; //v liste predstavuje 0,1 a v obyčajnom uzly predstavuje
písmeno a,b,c,...
    struct uzol *nula; //ukazovateľ na ďalší uzol, ak je ohodnotenie tejto
vrstvy 0
    struct uzol *jedna;//ukazovateľ na ďalší uzol, ak je ohodnotenie tejto
vrstvy 1
} UZOL;
```

BDD - reprezentuje súhrnne celý diagram, má 3 atribúty:

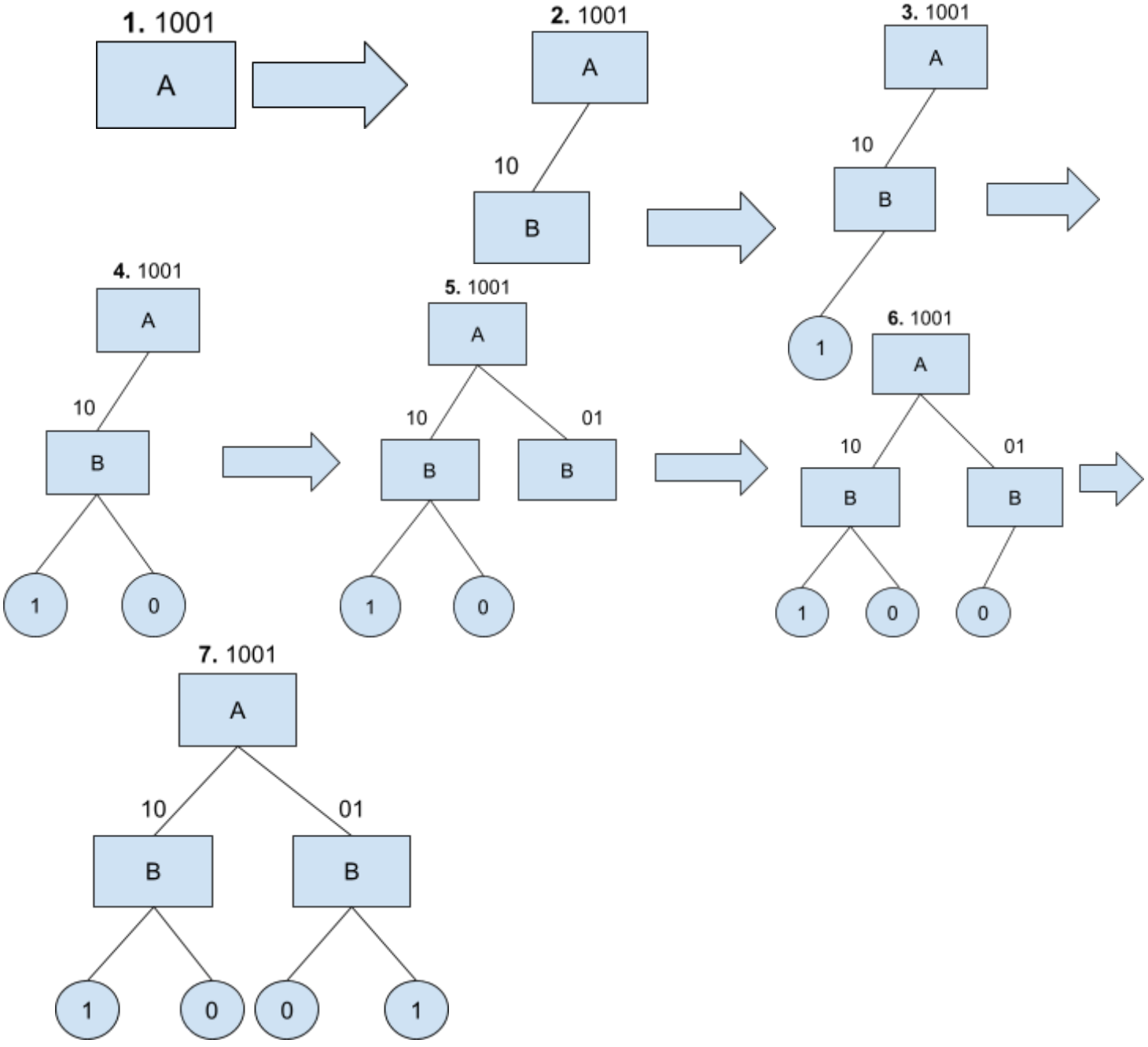
```
typedef struct bdd {
    UZOL *pointer; //ukazovateľ na koreň diagramu
    int pocet;  //počet premenných, z ktorých bol zostavený strom
    int velkost;   //počet uzlov v celom strome
} BDD;
```

Štruktúra BF nie je potrebná, pretože by používala iba 1 atribút a to char *vektor. Po konzultácií s p. prednášajúcim bola odstránená.

BDD_create zistí pomocou logaritmu počet premenných z vektoru v argumente, vytvorí koreň a zavolá spustenie rekurzívnej funkcie, ktorá postupne vytvára všetky uzly, pomocou funkcie bdd_novy().

Táto funkcia alokuje miesto pre jeden uzol a ďalej kontroluje, či má zapísať list alebo normálny uzol, na základe momentálnej vrstvy a maximálnej vrstvy, ktorých označenie má v argumente. V prípade vytvorenia listu vloží do atribútu “vstup” 0 alebo 1, ktoré funkcia našla v argumente char* vektor. Ak ale prečíta v tomto argumente iný vstup ako 0 alebo 1, zavolá sa exit(). V prípade vytvorenia obyčajného uzlu sa vloží písmeno z abecedy, ktorá je globálnou premennou programu.

Volanie funkcie bdd_novy() na správnom mieste zabezpečuje funkcia bdd_insert_rekurz(), ktorá je rekurzívnou funkciou. Diagram vytvára s preferenciou nulovej (ľavej) strany, teda takto:



Vstupom do tejto funkcie by boli iba 2 premenné a teda vektor o veľkosti 4, čo je 2**2. Vidíme, že počet uzlov je (2**(n+1)) - 1, kde n je počet premenných. Rovnako tak veľa je potrebných aj krokov na vytvorenie úplného diagramu. Vnútri v uzle sa zachováva iba písmeno z abecedy, ktoré predstavuje vrstvu. Vektor nad uzlami je naznačený iba pre lepšie pochopenie.

Na koniec BDD_create vytvára štruktúru BDD, ktorá obsahuje diagram, ako aj všetky potrebné údaje o ňom.

2. **BDD_use**:
   Veľmi jednoduchá funkcia, ktorá iba iteratívne prechádza cez diagram, ak nájde chybu tak vráti -1 a ak sa dostane úspešne do listu, vráti jeho atribút “vstup”, teda 0 alebo 1.
3. **BDD_free:**
   Nepovinná, avšak veľmi potrebná funkcia, ktorá rekurzívne prechádza cez diagram a postupne uvoľňuje jeho uzly. Smer rekurzie je opačný ako pri create, teda zdola nahor a s pravou preferenciou.

3. Testovanie programu

Program som testoval na 5 počtoch premenných: 13,14,15,16,17.
Program bol testovaný v IDE CLion, compiler je MinGW. Ak by som použil CLion a kompilovanie v Linuxe, časy create() by boli ešte nižsie, avšak časy use() by sa zväčšili. IDE VSCode s MinGW má extrémne veľké časy, preto som ho vôbec nepoužil.Testovanie prebiehalo nasledovne:

- predgeneroval som si pomocou jednoduchého Python skriptu potrebné vstupné vektory a nastavenia premenných, boli uložené v textových súboroch
- vložil som z textového súboru do programu vstupný vektor a následne som ho uložil do argumentu funkcie BBD_create()
- Na vytvorený diagram som zavolal všetky kombinácie vstupných premenných, ktoré som načítal zo súboru a vložil do argumentu
- Zmeral som časy oboch funkcií a uvolnil som vytvorený diagram. Namerané časy som zapísal do zbieracích premenných
- Vykonal som 2000 opakovaní pre každý počet premenných

Počet premenných (n)	Dokopy create 2000	Dokopy use 2^n	Priemerný create	2000 use*
13	1,38	0,912	0,00069	0,000111328
14	2,812	2,312	0,001406	0,000141113
15	5,459	5,787	0,00273	0,000176605
16	10,915	17,201	0,005458	0,000262466
17	22,471	43,584	0,011236	0,00033252

*1 use by bolo moc malé číslo

4. Záver

Záverom testovania je ohodnotenie priestorovej a časovej zložitosti:
Časová - Konkrétne časy pre konkrétne počty premenných sú uvedené v tabuľke, avšak sa dá pozorovať zložitosť O(2^n) pri create a O(n) pri use. Medzi počtami 15 a 16 je vyšší krok ako inde, ale to môže byť spôsobené aj inými faktormi, pretože krok medzi 16 a 17 je zase ten istý ako medzi 15 a 16.
Priestorová - Pre každý diagram je potrebné vytvoriť O(2^(n+1) - 1) uzlov, kde n je počet premenných. Ak sa správne uvoľňuje, tak program nezaberá v pamäti takmer žiadne miesto, čo viem vidieť tu: