FIIT STU v Bratislave

Binárny rozhodovací diagram

Norbert Matuška

xmatuskan@stuba.sk  Pondelok 9:00

Obsah

Contents

[Zadanie 2](#_Toc102842110)

[Opis riešenia 2](#_Toc102842111)

[BDD\_create 3](#_Toc102842112)

[BDD\_use 4](#_Toc102842113)

[Testovanie 5](#_Toc102842114)

# Zadanie

Našim zadaním bolo vypracovať a otestovať vlastnú implementáciu dátovej štruktúry s názvom Binárny rozhodovací diagram alebo taktiež BDD. Zadanie som vypracoval v jazyku C. Moje riešenie je rozdelené do troch súborov: bdd.c, tester.c, main.c. Je potrebné mať k nim aj .h súbory aby ich bolo možne spustiť.

# Opis riešenia

Na reprezentáciu boolovskej funkcie v bdd využívam vektor, ktorý získavam s funkcie ToVector, ktorá premení Boolovsku funkciu napríklad AB+C na binárny string, čiže vektor. BDD vytváram spôsobom zhora nadol a používam binárne stromy na uloženie, reprezentáciu a využitie dát v BDD. Všetky moje funkcie na vytváranie a používanie bdd fungujú rekurzívne.

char To\_Vector(char \*bfunkcia, char \*poradie, char \*hodnoty){  
 int tmpvalue = 1;  
 int value = 2;  
 int i,x;  
 int negate = 0;  
 char \*string = (char \*)malloc((strlen(bfunkcia)+1)\*sizeof(char));  
 for(i = 0;i<strlen(bfunkcia);i++){  
 string[i] = bfunkcia[i];  
 }  
 char \* token = strtok(string, "+");  
 while( token != NULL ) {  
 tmpvalue = 1;  
 for(i =0;i<strlen(token);i++) {  
 if(token[i]!='!'){  
 for(x = 0;x<strlen(poradie);x++){  
 int tmp;  
 if(poradie[x]==token[i]){  
 tmp = hodnoty[x]-48;  
 if(negate==1){  
 if(tmp==0)tmp=1;  
 else if(tmp==1)tmp=0;  
 negate = 0;  
 }  
 tmpvalue = tmpvalue\*tmp;  
 }  
 }  
 }else{  
 negate=1;  
 }  
 }  
 if(tmpvalue==1){  
 value=1;  
 }  
 token = strtok(NULL, "+");  
 }  
 if(value==2){  
 value=0;  
 }  
 return value+48;  
}

# BDD\_create

BDD vytváram zhora nadol, funguje vďaka rekurzii, kde si najprv vektor rozdelím na polovicu, vytvoria sa príslušné uzly do ľava a do prava a tie následne redukujem počas vytvárania, buď podľa redukcie S alebo redukcii I a taktiež pomocou hash tabulky. Toto sa opakuje kým novo vytvorený vektor nemá menšiu dĺžku ako 2. Keď sa tak stane, ideme napĺňať posledné uzly 1 alebo 0.

PBDD\_NODE build\_ROBDD(PBDD\_NODE parent, char \*bfunkcia, int lvl, PBDD\_NODE \*\*hashtable, int \*size, int \*one, int \*zero) {  
 int h, h\_size = power(lvl), flag;  
 parent->lvl = lvl;  
  
 if(strlen(bfunkcia) > 2) {  
 PBDD\_NODE child1 = malloc(sizeof(BDD\_NODE));  
 PBDD\_NODE child2 = malloc(sizeof(BDD\_NODE));  
  
 char \*s1 = malloc(sizeof(char) \* strlen(bfunkcia));  
 char \*s2 = malloc(sizeof(char) \* strlen(bfunkcia));  
 //rozdelenie vektora na 2 casti  
 strncpy(s1, bfunkcia, strlen(bfunkcia)/2);  
 s1[strlen(bfunkcia)/2] = '\0';  
 strncpy(s2, bfunkcia + strlen(bfunkcia) / 2, strlen(bfunkcia) - strlen(bfunkcia) / 2);  
 s2[strlen(bfunkcia) / 2] = '\0';  
  
 //vytvorenie rootov a do kazdeho posleme jednu polovicu z vektora  
 parent->left = build\_ROBDD(child1, s1, lvl + 1, hashtable, size, one, zero);  
 parent->right = build\_ROBDD(child2, s2, lvl + 1, hashtable, size, one, zero);  
 //redukcia typu S - korene su ten isty uzol  
 if (parent->right == parent->left) {  
 //ak je to uplne prvy pointer, tak tam musime nieco vlozit, aby to nebolo prazdne aj ked je to redukovane na 0  
 if(lvl == 0){  
 hashInsert(parent, hashtable,0, lvl);  
 }  
 return parent->right;  
 }  
 else {  
 //redukcia typu I - if uz rovnaky uzol existuje  
 flag = 1;  
 h = findHashIndex(parent, hashtable, lvl, h\_size, &flag);  
 //takyto prvok v zozname nie je  
 if(h != -1) {  
 hashInsert(parent, hashtable, h, lvl);  
 (\*size) += 1;  
 return &(\*hashtable)[lvl][h];  
 }  
 return &(\*hashtable)[lvl][flag];  
 }  
 }  
 else {  
 //vytvorenie a redukovanie prvkov na poslednej urovni == lvl  
 PBDD\_NODE child = malloc(sizeof(BDD\_NODE));  
 child->lvl = lvl;  
  
 //[0] je na lavo cize false  
 if(bfunkcia[0] == '0') {  
 child->left = zero;  
 }  
 else {  
 child->left = one;  
 }  
 //[1] je napravo cize true  
 if(bfunkcia[1] == '0') {  
 child->right = zero;  
 }  
 else {  
 child->right = one;  
 }  
  
 if (child->right == child->left) {  
 return child->right;  
 }  
 flag = 1;  
 h = findHashIndex(child, hashtable, lvl, h\_size, &flag);  
 //takyto prvok v zozname nieje  
 if (h != -1) {  
 hashInsert(child, hashtable, h, lvl);  
 (\*size) += 1;  
 return &(\*hashtable)[lvl][h];  
 }  
 return &(\*hashtable)[lvl][flag];  
 }  
}

# BDD\_use

Funkcia use slúži na otestovanie a zistenie výstupu pre zadanú kombináciu 0 a 1 pre zvolený vektor.

int \*x = parent;  
//ak je na konci (v 0 alebo v 1), konci  
if (\*x == 1 || \*x == 0) {  
 return \*x;  
}  
  
//inak sa posuvame do podkorenov  
if (vstupy[parent->lvl] == '1') {  
 x = parent->right;  
}  
else {  
 x = parent->left;  
}  
  
return getResult(x, vstupy);

# Testovanie

Moje testovanie pozostávalo zo zaznamenávania času vykonania funkcii create a use a taktiež aj zo zistenia percentuálne pomeru redukcie bdd.

Časová náročnosť ROBDD\_create() => O(n^n)

Časová náročnosť ROBDD\_use() = > O(n)

Kde n je počet premenných