

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

ФАКУЛЬТЕТ ІНФОРМАТИКИ ТА ОБЧИСЛЮВАЛЬНОЇ ТЕХНІКИ

КАФЕДРА ОБЧИСЛЮВАЛЬНОЇ ТЕХНІКИ

ЛАБОРАТОРНА РОБОТА №4

З КУРСУ

«ТЕХНОЛОГІЯ ПРОЕКТУВАННЯ КОМП’ЮТЕРНИХ СИСТЕМ»

Виконав:

студент IV курсу

групи ІО-21

Журо Георгій Олександрович

Київ – 2015

**Тема:** Автоматизація кодування графу переходів.

**Мета:** Здобуття навичок з автоматизації процедури сумісного кодування графу переходів.

**Завдання:**

1. Розробити алгоритм сумісного кодування графу переходів з попередньої роботи – будь-які вузли, що мають зв'язок повинні мати коди, які відрізняються лише у одному двійковому розряді. Блок-схему та опис розробленого алгоритму надати в протоколі роботи.
2. Реалізувати розроблений алгоритм. Кодування відобразити на графічному представленні графу переходів.
3. Модифікувати формат зберігання графу переходів таким чином, щоб він містив інформацію про коди вузлів. Реалізувати можливість збереження/відновлення закодованого графа переходів.

**Блок-схема алгоритму:**



**Опис програми:**

Для виконання сумісного кодування графу переходів з попередньої лабораторної роботи необхідно натиснути кнопку «Code Graph» (рис. 1). Після чого в новій вкладці буде відображений закодований граф переходів. Для графу, зображеного на рисунку 1, закодований граф переходів показаний на рисунку 2.

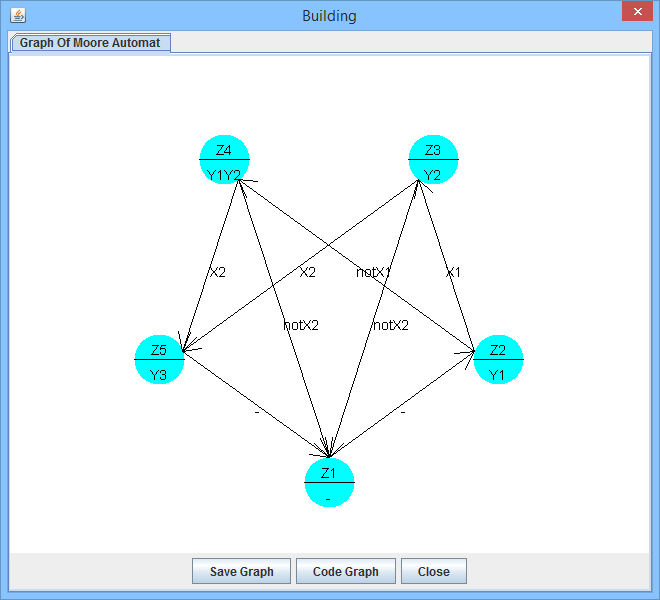


Рисунок 1

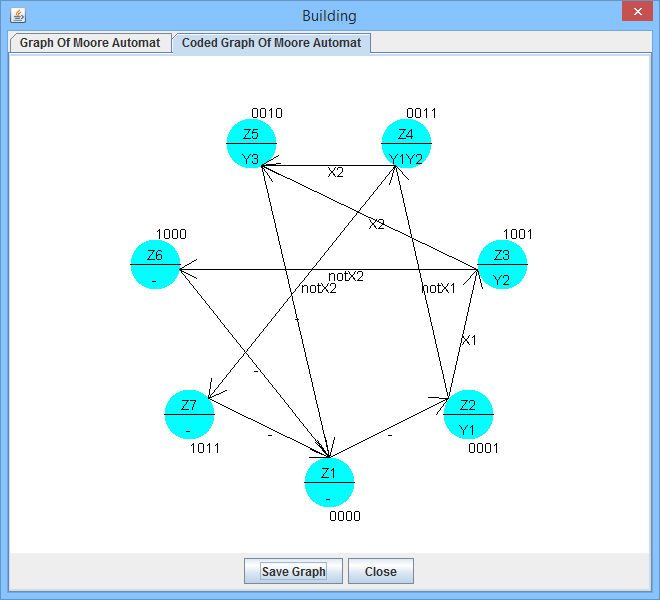


Рисунок 2

Для збереження закодованого графу переходів необхідно натиснути кнопку «Save Graph» у вкладці «Coded Graph Of Moore Automat» та ввести ім’я файлу в діалоговому вікні.

**Лістинг програми:**

package automat.moore;

import java.io.\*;

import java.util.LinkedList;

import java.util.ListIterator;

public class CodedMooreAutomat extends MooreAutomat {

private String[] stateCodes;

public CodedMooreAutomat(MooreAutomat automat) {

String[] stateNames = automat.getStateNames();

this.stateNames = new String[stateNames.length];

for (int i = 0; i < stateNames.length; i++) {

this.stateNames[i] = stateNames[i];

}

int[][] yNumbers = automat.getyNumbers();

this.yNumbers = new int[yNumbers.length][];

for (int i = 0; i < yNumbers.length; i++) {

if (yNumbers[i] != null) {

this.yNumbers[i] = new int[yNumbers[i].length];

for (int j = 0; j < yNumbers[i].length; j++) {

this.yNumbers[i][j] = yNumbers[i][j];

}

}

else {

this.yNumbers[i] = null;

}

}

int[][] connectionMatrix = automat.getConnectionMatrix();

this.connectionMatrix = new int[connectionMatrix.length][];

for (int i = 0; i < connectionMatrix.length; i++) {

this.connectionMatrix[i] = new int[connectionMatrix[i].length];

for (int j = 0; j < connectionMatrix[i].length; j++) {

this.connectionMatrix[i][j] = connectionMatrix[i][j];

}

}

int[][] xNumbers = automat.getxNumbers();

this.xNumbers = new int[xNumbers.length][];

for (int i = 0; i < xNumbers.length; i++) {

if (xNumbers[i] != null) {

this.xNumbers[i] = new int[xNumbers[i].length];

for (int j = 0; j < xNumbers[i].length; j++) {

this.xNumbers[i][j] = xNumbers[i][j];

}

}

else {

this.xNumbers[i] = null;

}

}

boolean[][] xValues = automat.getxValues();

this.xValues = new boolean[xValues.length][];

for (int i = 0; i < xValues.length; i++) {

if (xValues[i] != null) {

this.xValues[i] = new boolean[xValues[i].length];

for (int j = 0; j < xValues[i].length; j++) {

this.xValues[i][j] = xValues[i][j];

}

}

else {

this.xValues[i] = null;

}

}

codeAutomat();

}

private void codeAutomat() {

while ((!checkForCycles()) || (!checkForSecondOrder())) {}

double temp = Math.log(stateNames.length) / Math.log(2);

int signalCount = (int) temp;

if (temp > signalCount) {

signalCount += 1;

}

while (getMaxStatePower() > signalCount) {

signalCount++;

}

int combinationCount = (int) Math.pow(2, signalCount);

boolean[] isBusy = new boolean[combinationCount];

String[] combinations = new String[combinationCount];

for (int i = 0; i < combinationCount; i++) {

isBusy[i] = false;

combinations[i] = intToBinary(i, signalCount);

}

int[] codes = new int[stateNames.length];

for (int i = 0; i < codes.length; i++) {

codes[i] = -1;

}

int[][] neighboursCombination = new int[combinations.length][];

for (int i = 0; i < neighboursCombination.length; i++) {

neighboursCombination[i] = new int[signalCount];

int nCount = 0;

for (int j = 0; j < combinations.length; j++) {

if (i != j) {

int differentCount = 0;

for (int k = 0; k < signalCount; k++) {

if (combinations[i].charAt(k) != combinations[j].charAt(k)) {

differentCount++;

}

}

if (differentCount == 1) {

neighboursCombination[i][nCount++] = j;

}

}

}

}

stateCodes = new String[stateNames.length];

int edgesCount = 0;

for (int i = 0; i < connectionMatrix.length; i++) {

for (int j = 0; j < connectionMatrix[i].length; j++) {

if ((connectionMatrix[i][j] >= 0) && (i != j)) {

edgesCount++;

}

}

}

int[][] edges = new int[edgesCount][];

for (int i = 0; i < edges.length; i++) {

edges[i] = new int[4];

}

edgesCount = 0;

for (int i = 0; i < connectionMatrix.length; i++) {

for (int j = 0; j < connectionMatrix[i].length; j++) {

if ((connectionMatrix[i][j] >= 0) && (i != j)) {

boolean isAlready = false;

for (int k = 0; k < edgesCount; k++) {

if (((i == edges[k][0]) && (j == edges[k][1])) || ((i == edges[k][1]) && (j == edges[k][0]))) {

edges[k][2]++;

isAlready = true;

}

}

if (!isAlready) {

edges[edgesCount][0] = i;

edges[edgesCount][1] = j;

edges[edgesCount][2] = 1;

edgesCount++;

}

}

}

}

for (int i = 0; i < edgesCount; i++) {

int max = i;

for (int j = i + 1; j < edgesCount; j++) {

if (edges[j][2] > edges[max][2]) {

max = j;

}

}

if (max != i) {

int[] swapTemp = edges[i];

edges[i] = edges[max];

edges[max] = swapTemp;

}

}

int[] stateWeights = new int[stateNames.length];

for (int i = 0; i < stateWeights.length; i++) {

stateWeights[i] = 0;

for (int j = 0; j < stateNames.length; j++) {

if (i != j) {

if ((connectionMatrix[i][j] >= 0) || (connectionMatrix[j][i] >= 0)) {

stateWeights[i]++;

}

}

}

}

for (int i = 0; i < edgesCount; i++) {

edges[i][3] = stateWeights[edges[i][0]] + stateWeights[edges[i][1]];

}

int z = 0;

int previousEdgeWeight = -1;

while (z < edgesCount) {

if (previousEdgeWeight == edges[z][2]) {

int from = z - 1;

while ((z < edgesCount) && (previousEdgeWeight == edges[z][2])) {

z++;

}

for (int i = from; i < z; i++) {

int max = i;

for (int j = i + 1; j < z; j++) {

if (edges[j][3] > edges[max][3]) {

max = j;

}

}

int[] swapTemp = edges[max];

edges[max] = edges[i];

edges[i] = swapTemp;

}

}

else {

previousEdgeWeight = edges[z][2];

z++;

}

}

for (int i = 0; i < edgesCount; i++) {

if ((codes[edges[i][0]] == -1) && (codes[edges[i][1]] == -1)) {

boolean found = false;

int j = 0;

while (isBusy[j]) {

j++;

}

codes[edges[i][0]] = j;

isBusy[j] = true;

stateCodes[edges[i][0]] = combinations[j];

int k = 0;

while (isBusy[neighboursCombination[j][k]]) {

k++;

}

codes[edges[i][1]] = neighboursCombination[j][k];

isBusy[neighboursCombination[j][k]] = true;

stateCodes[edges[i][1]] = combinations[neighboursCombination[j][k]];

}

else {

if (codes[edges[i][0]] == -1) {

LinkedList<Integer> neighbourCodedStates = new LinkedList<Integer>();

neighbourCodedStates.add(edges[i][1]);

for (int j = 0; j < edgesCount; j++) {

if (i != j) {

if (edges[j][0] == edges[i][0]) {

if (codes[edges[j][1]] != -1) {

neighbourCodedStates.add(edges[j][1]);

}

} else {

if (edges[j][1] == edges[i][0]) {

if (codes[edges[j][0]] != -1) {

neighbourCodedStates.add(edges[j][0]);

}

}

}

}

}

ListIterator<Integer> listIterator = neighbourCodedStates.listIterator();

LinkedList<Integer> freeNeighboursCombinations = new LinkedList<Integer>();

while (listIterator.hasNext()) {

int neighbourCodedState = listIterator.next();

for (int j = 0; j < neighboursCombination[codes[neighbourCodedState]].length; j++) {

if (!isBusy[neighboursCombination[codes[neighbourCodedState]][j]]) {

freeNeighboursCombinations.add(neighboursCombination[codes[neighbourCodedState]][j]);

}

}

}

boolean found = false;

int j = 0;

while ((!found) && (j < freeNeighboursCombinations.size())) {

int tempCount = 1;

for (int k = j + 1; k < freeNeighboursCombinations.size(); k++) {

if (freeNeighboursCombinations.get(j) == freeNeighboursCombinations.get(k)) {

tempCount++;

}

}

if (tempCount == neighbourCodedStates.size()) {

found = true;

}

j++;

}

codes[edges[i][0]] = freeNeighboursCombinations.get(--j);

isBusy[freeNeighboursCombinations.get(j)] = true;

stateCodes[edges[i][0]] = combinations[freeNeighboursCombinations.get(j)];

}

else {

if (codes[edges[i][1]] == -1) {

LinkedList<Integer> neighbourCodedStates = new LinkedList<Integer>();

neighbourCodedStates.add(edges[i][0]);

for (int j = 0; j < edgesCount; j++) {

if (i != j) {

if (edges[j][0] == edges[i][1]) {

if (codes[edges[j][1]] != -1) {

neighbourCodedStates.add(edges[j][1]);

}

} else {

if (edges[j][1] == edges[i][1]) {

if (codes[edges[j][0]] != -1) {

neighbourCodedStates.add(edges[j][0]);

}

}

}

}

}

ListIterator<Integer> listIterator = neighbourCodedStates.listIterator();

LinkedList<Integer> freeNeighboursCombinations = new LinkedList<Integer>();

while (listIterator.hasNext()) {

int neighbourCodedState = listIterator.next();

for (int j = 0; j < neighboursCombination[codes[neighbourCodedState]].length; j++) {

if (!isBusy[neighboursCombination[codes[neighbourCodedState]][j]]) {

freeNeighboursCombinations.add(neighboursCombination[codes[neighbourCodedState]][j]);

}

}

}

boolean found = false;

int j = 0;

while ((!found) && (j < freeNeighboursCombinations.size())) {

int tempCount = 1;

for (int k = j + 1; k < freeNeighboursCombinations.size(); k++) {

if (freeNeighboursCombinations.get(j) == freeNeighboursCombinations.get(k)) {

tempCount++;

}

}

if (tempCount == neighbourCodedStates.size()) {

found = true;

}

j++;

}

codes[edges[i][1]] = freeNeighboursCombinations.get(--j);

isBusy[freeNeighboursCombinations.get(j)] = true;

stateCodes[edges[i][1]] = combinations[freeNeighboursCombinations.get(j)];

}

}

}

}

}

private boolean checkForCycles() {

boolean result = true;

for (int i = 0; i < connectionMatrix.length; i++) {

for (int j = 0; j < connectionMatrix[i].length; j++) {

if (connectionMatrix[i][j] > -1) {

if (i != j) {

int[] cycle = new int[2];

cycle[0] = i;

cycle[1] = j;

boolean flag = stepForCycles(cycle, j);

if (!flag) {

result = false;

}

}

}

}

}

return result;

}

private String intToBinary(int i, int n) {

String s = "";

int temp = i;

int divider = (int) Math.pow(2, n - 1);

while (divider > 1) {

s += String.valueOf(temp / divider);

temp = temp % divider;

divider /= 2;

}

s += String.valueOf(temp);

return s;

}

public String[] getStateCodes() {

return stateCodes;

}

public static void writeToFile(File file, CodedMooreAutomat automat) throws IOException {

ObjectOutputStream output = new ObjectOutputStream(new FileOutputStream(file));

output.writeObject(automat);

output.close();

}

public static CodedMooreAutomat readFromFile(File file) throws IOException, ClassNotFoundException {

ObjectInputStream input = new ObjectInputStream(new FileInputStream(file));

CodedMooreAutomat automat = (CodedMooreAutomat) input.readObject();

input.close();

return automat;

}

}

**Висновки:** При виконанні даної лабораторної роботи я здобув навички з автоматизації сумісного кодування графу переходів автомату Мура. Сумісне кодування графу я реалізував за допомогою евристичного алгоритму кодування. Також мною була реалізована можливість зберігання/відновлення закодованого графу переходів за допомогою механізму серіалізації об’єктів мови програмування Java.