Национальный Технический Университет Украины

“Киевский Политехнический Институт”

Факультет Информатики и Вычислительной Техники

Кафедра вычислительной техники

Лабораторная работа **№4**

по дисциплине “ Системное программное обеспечение”

Выполнил:

студент 3-го курса

Группы ИВ-01

Редько А.М.

Киев, 2013 г.

Задание на лабораторную работу

Номер зачётной книжки: 0108

Номер варианта: 8 + 1 = 9

Задание варианта: Визначити тимчасову складність алгоритму виділення всіх критичних вершин для заданого зваженого ациклічного графа (число вершин не менше 30)..

Описание алгоритма работы

Для определения транзитивных вершин используется следующее правило: все вершины, не входящие в критический путь графа, являются транзитивными.

**Метод критического пути** — эффективный инструмент [планирования](http://ru.wikipedia.org/wiki/%D0%9F%D0%BB%D0%B0%D0%BD%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) расписания. В основе метода лежит определение наиболее длительной последовательности [задач](http://ru.wikipedia.org/wiki/%D0%97%D0%B0%D0%B4%D0%B0%D1%87%D0%B0) от начала [проекта](http://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B5%D0%BA%D1%82_%28%D0%B4%D0%B5%D1%8F%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D0%BE%D1%81%D1%82%D1%8C%29) до его окончания с учетом их взаимосвязи. Задачи, лежащие на критическом пути (критические задачи), имеют нулевой резерв времени выполнения и в случае изменения их длительности изменяются сроки всего проекта. В связи с этим при выполнении проекта критические задачи требуют более тщательного контроля, в частности, своевременного выявления проблем и рисков, влияющих на сроки их выполнения и, следовательно, на сроки выполнения проекта в целом. В процессе выполнения проекта критический путь проекта может меняться, так как при изменении длительности задач некоторые из них могут оказаться на критическом пути.

В терминологии теории графов **сетевым графиком** называют конечный ориентированный граф без контуров, в котором имеются единственная вершина с отсутствующими прообразами и единственная вершина, не имеющая образов.

Иначе сетевым графиком можно назвать ориентированную транспортную сеть с одним входом и одним выходом, в которой нет путей с повторяющимися вершинами.

В процессе решения — методом «эстафеты» — просматриваются все дуги сетевого графика. Пусть очередная просматриваемая дуга связывает вершины i и j. Если для вершины i определено предположительное время его свершения и это время плюс продолжительность работы больше предположительного времени наступления события j, тогда для вершины j устанавливается новое предположительное время наступления, равное предположительному времени наступления события i плюс продолжительность работы рассматриваемой дуги. Решение заканчивается, когда очередной просмотр дуг не вызывает ни одного исправления предположительного значения времени начала/окончания работ/событий. В результате может быть определено событие с самым поздним временем наступления, и путь от начальной вершины в эту конечную будет считаться критическим и определять продолжительность выполнения проекта. Наряду с общей продолжительностью выполнения проекта, критический путь определяет другие характеристики сетевого графика, играющие важную роль при планировании реализации нововведения, минимизации сроков и расходов на разработку.



Рассчитаем Aj - самое раннее возможное время наступления события j и Bj – самое позднее возможное время наступления события j.

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** |
| **A** | 0 | 2 | 2 | 2 | 5 | 6 | 10 | 13 | 13 | 15 | 17 | 28 |
| **B** | 0 | 5 | 4 | 2 | 15 | 8 | 10 | 19 | 17 | 15 | 21 | 28 |

Найдём критический путь:

В критический путь входят вершины 1, 4, 7, 10, 12.



Следовательно, остальные вершины транзитивны.

**Листинг программы**

**package** os.lab4;

**public** **class** Program {

**public** **static** **void** main(String[] args) {

MainFrame frame = **new** MainFrame();

frame.setVisible(**true**);

}

}

**package** os.lab4;

**import** java.awt.Dimension;

**import** java.awt.Font;

**import** java.awt.event.ActionEvent;

**import** java.io.File;

**import** java.io.FileNotFoundException;

**import** javax.swing.AbstractAction;

**import** javax.swing.JFileChooser;

**import** javax.swing.JFrame;

**import** javax.swing.JMenu;

**import** javax.swing.JMenuBar;

**import** javax.swing.JMenuItem;

**import** javax.swing.JScrollPane;

**import** javax.swing.JSplitPane;

**import** javax.swing.JTextArea;

**public** **class** MainFrame **extends** JFrame {

**private** **static** **final** **long** *serialVersionUID* = 4272542240756969890L;

**private** JTextArea textArea;

**private** JFileChooser chooser;

**private** Planner planner;

**public** MainFrame() {

**super**();

setTitle("Лабораторная работа №4");

setMinimumSize(**new** Dimension(800, 600));

setExtendedState(JFrame.*MAXIMIZED\_BOTH*);

setDefaultCloseOperation(JFrame.*EXIT\_ON\_CLOSE*);

JSplitPane splitPane = **new** JSplitPane(JSplitPane.*HORIZONTAL\_SPLIT*);

textArea = **new** JTextArea();

textArea.setFont(**new** Font("Monospaced", Font.*PLAIN*, 12));

textArea.setEditable(**false**);

add(**new** JScrollPane(textArea));

splitPane.setDividerLocation(400);

JMenuBar menuBar = **new** JMenuBar();

JMenu fileMenu = **new** JMenu("Файл");

JMenuItem openItem = **new** JMenuItem();

openItem.setAction(**new** OpenAction(**this**));

openItem.setText("Открыть...");

fileMenu.add(openItem);

menuBar.add(fileMenu);

setJMenuBar(menuBar);

chooser = **new** JFileChooser();

chooser.setDialogType(JFileChooser.*OPEN\_DIALOG*);

chooser.setFileSelectionMode(JFileChooser.*FILES\_ONLY*);

chooser.setCurrentDirectory(**new** File("."));

}

**private** **class** OpenAction **extends** AbstractAction {

**private** **static** **final** **long** *serialVersionUID* = -5099362063131839615L;

**private** MainFrame frame;

**public** OpenAction(MainFrame frame) {

**this**.frame = frame;

}

**public** **void** actionPerformed(ActionEvent arg0) {

**int** answer = chooser.showOpenDialog(frame);

**if** (answer == JFileChooser.*APPROVE\_OPTION*) {

**try** {

planner = Planner.*readGraphFromFile*(chooser.getSelectedFile());

} **catch** (FileNotFoundException e) {}

textArea.setText(planner.getLog());

}

}

}

}

**package** os.lab4;

**import** java.io.File;

**import** java.io.FileInputStream;

**import** java.io.FileNotFoundException;

**import** java.util.ArrayList;

**import** java.util.Scanner;

**public** **class** Planner {

**private** **int**[][] connectivityMatrix;

**private** **int**[] vertexWeight;

**private** ArrayList<Integer> roots = **new** ArrayList<Integer>();

**private** ArrayList<Integer> edges = **new** ArrayList<Integer>();

**private** ArrayList<Integer> tempCriticalWay;

**private** Planner(**int** n) {

connectivityMatrix = **new** **int**[n][];

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

connectivityMatrix[i] = **new** **int**[n];

}

vertexWeight = **new** **int**[n];

}

**public** **static** Planner readGraphFromFile(File file) **throws** FileNotFoundException {

Scanner input = **new** Scanner(**new** FileInputStream(file));

ArrayList<Integer> buf = **new** ArrayList<Integer>();

**while** (input.hasNext()) {

buf.add(input.nextInt());

}

input.close();

**int** k = 0;

Planner planner = **new** Planner(buf.get(k));

**for** (**int** i = 0; i < planner.connectivityMatrix.length; i++) {

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

planner.connectivityMatrix[i][j] = buf.get(++k);

}

}

**for** (**int** i = 0; i < planner.vertexWeight.length; i++) {

planner.vertexWeight[i] = buf.get(++k);

}

*getRootsAndEdges*(planner);

**return** planner;

}

**public** String getLog() {

StringBuilder logger = **new** StringBuilder("");

logger.append("--------------------------------------------\n");

outputMatrix(logger);

**boolean** last = **false**;

**int** i = 1;

**while** (!last) {

ArrayList<Integer> criticalWay = findCriticalWay();

logger.append("Критический путь №");

logger.append(String.*valueOf*(i++));

logger.append("\n");

**int** sum = 0;

**for** (**int** j = 0; j < (criticalWay.size() - 1); j++) {

logger.append(String.*valueOf*(criticalWay.get(j)));

logger.append(" -> ");

sum += vertexWeight[criticalWay.get(j)];

}

sum += vertexWeight[criticalWay.get(criticalWay.size() - 1)];

logger.append(String.*valueOf*(criticalWay.get(criticalWay.size() - 1)));

logger.append("\n");

logger.append("Tкр(вершин) = ");

logger.append(String.*valueOf*(sum));

logger.append("\n");

**int** allSum = 0;

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

allSum += vertexWeight[j];

}

logger.append("Tmax(вершин) = ");

logger.append(String.*valueOf*(allSum));

logger.append("\n");

logger.append("Nlow = ");

logger.append(String.*valueOf*(allSum / sum));

logger.append("\n");

**int** connectSum = 0;

**for** (**int** j = 0; j < (criticalWay.size() - 1); j++) {

connectSum += connectivityMatrix[criticalWay.get(j)][criticalWay.get(j + 1)];

}

logger.append("Tкр(пересылок) = ");

logger.append(String.*valueOf*(connectSum));

logger.append("\n");

**int** allConnectSum = 0;

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

**for** (**int** k = 0; k < connectivityMatrix[j].length; k++) {

allConnectSum += connectivityMatrix[j][k];

}

}

logger.append("Tmax(пересылок) = ");

logger.append(String.*valueOf*(allConnectSum));

logger.append("\n");

logger.append("N = ");

logger.append(String.*valueOf*(connectivityMatrix.length));

logger.append("\n");

**while** (criticalWay.size() > 0) {

**int** min = criticalWay.get(0);

**int** minPosition = 0;

**for** (**int** j = 0; j < criticalWay.size(); j++) {

**if** (criticalWay.get(j) < min) {

min = criticalWay.get(j);

minPosition = j;

}

}

**int**[][] newConnectiveMatrix = **new** **int**[connectivityMatrix.length - 1][];

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

newConnectiveMatrix[j] = **new** **int**[newConnectiveMatrix.length];

}

**for** (**int** j = 0; j < min; j++) {

**for** (**int** k = 0; k < min; k++) {

newConnectiveMatrix[j][k] = connectivityMatrix[j][k];

}

}

**for** (**int** j = min + 1; j < connectivityMatrix.length; j++) {

**for** (**int** k = 0; k < min; k++) {

newConnectiveMatrix[j - 1][k] = connectivityMatrix[j][k];

}

}

**for** (**int** j = 0; j < min; j++) {

**for** (**int** k = min + 1; k < connectivityMatrix[j].length; k++) {

newConnectiveMatrix[j][k - 1] = connectivityMatrix[j][k];

}

}

**for** (**int** j = min + 1; j < connectivityMatrix.length; j++) {

**for** (**int** k = min + 1; k < connectivityMatrix[j].length; k++) {

newConnectiveMatrix[j - 1][k - 1] = connectivityMatrix[j][k];

}

}

criticalWay.remove(minPosition);

**for** (**int** j = 0; j < criticalWay.size(); j++) {

criticalWay.set(j, criticalWay.get(j) - 1);

}

connectivityMatrix = newConnectiveMatrix;

*getRootsAndEdges*(**this**);

**int**[] newVertexWeight = **new** **int**[vertexWeight.length - 1];

**for** (**int** j = 0; j < min; j++) {

newVertexWeight[j] = vertexWeight[j];

}

**for** (**int** j = min + 1; j < vertexWeight.length; j++) {

newVertexWeight[j -1] = vertexWeight[j];

}

vertexWeight = newVertexWeight;

**if** (newConnectiveMatrix.length == 0) {

last = **true**;

}

}

logger.append("Произведено зануление критического пути.\n");

logger.append("--------------------------------------------\n");

**if** (!last) {

outputMatrix(logger);

}

}

**return** logger.toString();

}

**private** **static** **void** getRootsAndEdges(Planner planner) {

planner.roots = **new** ArrayList<Integer>();

planner.edges = **new** ArrayList<Integer>();

**for** (**int** i = 0; i < planner.connectivityMatrix.length; i++) {

**boolean** edgeFlag = **true**;

**boolean** rootFlag = **true**;

**int** j = 0;

**while** (j < planner.connectivityMatrix.length) {

**if** (planner.connectivityMatrix[i][j] != 0) {

edgeFlag = **false**;

}

**if** (planner.connectivityMatrix[j][i] != 0) {

rootFlag = **false**;

}

j++;

}

**if** (edgeFlag) {

planner.edges.add(i);

}

**if** (rootFlag) {

planner.roots.add(i);

}

}

}

**private** **void** outputMatrix(StringBuilder logger) {

logger.append("Матрица связности графа:\n");

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

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

logger.append(String.*valueOf*(connectivityMatrix[i][j]));

logger.append(" ");

}

logger.append("\n");

}

logger.append("Веса вершин графа:\n");

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

logger.append(String.*valueOf*(vertexWeight[i]));

logger.append(" ");

}

logger.append("\n");

}

**private** ArrayList<Integer> findCriticalWay() {

**int** maxSum = 0;

ArrayList<Integer> way = **new** ArrayList<Integer>();

**for** (**int** i = 0; i < roots.size(); i++) {

ArrayList<Integer> tempWay = **new** ArrayList<Integer>();

**int** tempSum = 0;

tempSum = step(roots.get(i), tempWay, tempSum);

**if** (tempSum > maxSum) {

maxSum = tempSum;

way = tempCriticalWay;

}

}

**return** way;

}

**private** **int** step(**int** position, ArrayList<Integer> way, **int** sum) {

**if** (!edges.contains(position)) {

ArrayList<Integer> criticalWay = **new** ArrayList<Integer>();

**int** criticalSum = 0;

**for** (**int** i = 0; i < connectivityMatrix[position].length; i++) {

**if** (connectivityMatrix[position][i] > 0) {

ArrayList<Integer> tempWay = **new** ArrayList<Integer>();

**for** (**int** j = 0; j < way.size(); j++) {

tempWay.add(way.get(j));

}

tempWay.add(position);

**int** tempSum = sum + vertexWeight[position];

tempSum = step(i, tempWay, tempSum);

**if** (tempSum > criticalSum) {

criticalSum = tempSum;

criticalWay = tempCriticalWay;

}

}

}

tempCriticalWay = criticalWay;

**return** criticalSum;

}

**else** {

way.add(position);

tempCriticalWay = way;

sum += vertexWeight[position];

**return** sum;

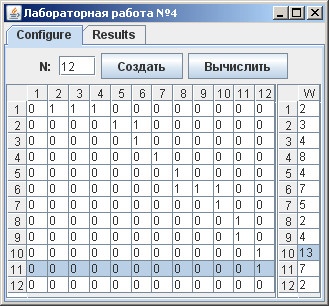
}

}

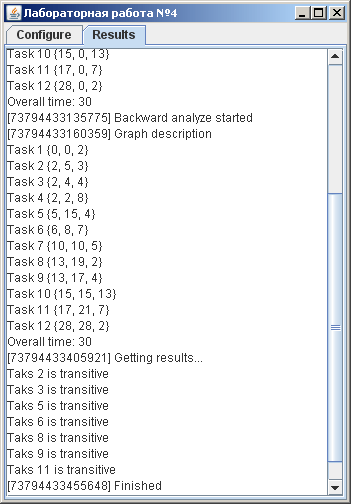
}

Результаты моделирования

Запустим написанную программу на обработку данной задачи. Для этого опишем граф с помощью 2 таблиц: матрицы смежности и матрицы весов вершин.



После чего запустим выполнение обработки.



Как видно, решение, полученное программной, совпадает с результатами, полученными вручную.

Выполнив обработку матриц различных размерностей, получили следующую зависимость времени выполнения программы от величины N:

