Министерство образования и науки Российской Федерации

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | | | | | |
| Федеральное государственное бюджетное образовательное учреждение  высшего образования | | | | | |
| «Алтайский государственный технический университет им. И.И. Ползунова» | | | | | |
| Факультет информационных технологий  Кафедра "Прикладная математика" | | | | | |
|  | | | Отчет защищен с оценкой \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | | |
|  | | | Преподаватель \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_А.В. Астахова  Подпись | | |
|  | | | «\_\_\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_2017 г. | | |
|  | | |  | | |
| **ОТЧЕТ**  **ПО ЛАБОРАТОРНОЙ РАБОТЕ № 2** | | | | | |
|  | | | | | |
| по дисциплине «ОСНОВЫ ЭКОНОМИКИ ПРОГРАММНОЙ ИНЖЕНЕРИИ» | | | | | |
| **ЛР 09.03.04.05.001 О** | | | | | |
| Студент группы ПИ-42 Г.Г. Глушков | | | | | |
|  |  |  | | подпись | и.о.,фамилия |
| Преподаватель профессор А.В. Астахова | | | | | |
| должность, ученая степень подпись | | | | | и.о.,фамилия |

БАРНАУЛ 2017

Постановка задачи

Рассчитать параметры событий и параметры работ сетевого графика.

Составит программу, которая позволит: рассчитывать параметры событий начальной вершины, рассчитывать свободный и полный резерв времени работ, определять топологию и длину критических путей, обнаруживать и удалять циклы и петли в графе.

Алгоритм решения

Расчёт сетевого графика заключается в установлении ранних и поздних начал и окончаний работ, определении общих и частных резервов времени и определении критического пути.

Наиболее удобным способом расчёта сетевого графика является расчёт непосредственно на сети графика.

Для расчёта параметров сетевого графика события обозначают в виде круга, разделённого на четыре сектора. В верхнем секторе помещают порядковый номер события; в нижнем секторе –резерв наступления события. В левый сектор круга заносят значение раннего начала последующей работы, а в правый – раннее окончание предшествующей работы.

Расчёт графика начинают с заполнения левых секторов событий по направлению движения стрелок от начального (исходного) события до завершающего. Предварительно каждому событию должен быть присвоен свой номер.

Если в одно событие входит несколько работ, то в левый сектор этого события (раннее начало последующей работы) заносят наибольшее значение времени на любом из путей, ведущих к данному событию.

В завершающем событии значение максимального значения переносят из левого сектора в правый.

Далее расчёт графика осуществляют по ранним окончаниям работ и ведут его в обратном порядке: от завершающего события к исходному. В этом случае значение правого сектора события получают путём вычитания из данных правого сектора завершающего события продолжительности работы, идущей от предыдущего события.

Если из события выходит несколько работ, то в правый сектор заносят разницу, имеющую наименьшее значение.

Далее, по значениям ранних начал работ и ранних окончаний работ определяют общие и частные резервы времени.

Под общим (полным) резервом времени понимают количество рабочего времени, на которое может быть увеличена продолжительность данной работы при условии, что продолжительность наибольшего из путей, проходящих через эту работу, не превышало длины критического пути.

Частным резервом времени называют количество рабочего времени, на которое может быть увеличена продолжительность данной работы или перенесено её начало так, чтобы при этом не изменилось раннее начало последующих работ.

Резервы времени записывают над стрелками работ в виде дроби: в числителе общий резерв времени, в знаменателе – частный резерв времени.

Общий (полный) резерв времени находят путём вычитания из значения правого сектора события (раннее окончание предшествующей работы), куда эта работа входит, значение левого сектора (раннее начало последующей работы), откуда эта работа выходит, минус продолжительность самой работы.

Частный резерв времени определяют путём вычитания значения левого сектора события, куда эта работа входит, значение левого сектора события, откуда эта работа выходит и минус продолжительность самой работы.

Критический путь проходит по тем работам, у которых значения правого и левого секторов события идентичны и у которых полностью отсутствуют резервы времени (как общие, так и частные).

Реализация

private static HashSet<Integer> nods;

private static HashSet<Integer> left;

private static HashSet<Integer> right;

private static HashSet<Integer> begins;

private static HashSet<Integer> ends;

private static HashSet<Triple> links;

private static HashMap<Integer, Integer> color;

private static HashMap<Integer, Integer> order;

private static HashMap<Integer, Node> Nods;

private static Node start;

private static Node finish;

private static ArrayList<String> pathways;

private static HashMap<Integer, Integer> path;

private static int cycle\_end = 0,cycle\_beg = 0;

/\*\*

\* @param args the command line arguments

\*/

public static void main(String[] args) {

Scanner in = new Scanner(System.in);

while(true) {

nods = new HashSet<>();

left = new HashSet<>();

right = new HashSet<>();

begins = new HashSet<>();

ends = new HashSet<>();

links = new HashSet<>();

Nods = new HashMap<>();

HashSet<Triple> \_links = new HashSet<>();

pathways = new ArrayList<>();

path = new HashMap<>();

out.println("Ввод:");

int cur = in.nextInt();//Считыванием все ребра

while (cur != -1) {

int cur2 = in.nextInt();

int wei = in.nextInt();

//Проверяем на повторения

boolean canadd = true;

for(Triple link: links){

if(link.beg == cur && link.end == cur2) {

canadd = false;

break;

}

}

if(canadd) {

nods.add(cur);

nods.add(cur2);

left.add(cur);

right.add(cur2);

links.add(new Triple(cur, cur2, wei));

}

cur = in.nextInt();

}

//Делим на начальные и конечные точки

for (Integer obj : nods) {

if (!right.contains(obj)) {

begins.add(obj);

}

if (!left.contains(obj)) {

ends.add(obj);

}

}

boolean skip = false;

int choosen = 0;

//Поиск петель

for(Triple link : links){

if(link.beg == link.end){

if(choosen == 0) {

out.println("Была найдена петля.");

out.print("Варианты решения:\n1)Удалить все петли\n2)Ввести рёбра заново\n");

out.print("Выбранный вариант: ");

int choose = in.nextInt();

if (choose == 1) {

choosen = 1;

} else if (choose == 2) {

skip = true;

break;

}

}

}else{

\_links.add(link);

}

}

//Если выбрано ввести все ребра заново

if(skip)

continue;

//Копируются ребра без петель

links = \_links;

//Раскраска графа с целью поиска циклов

color = new HashMap<>();

for(Integer nod : nods){

color.put(nod,0);

}

if(begins.size() == 0){

for(Integer beg : nods)

if(findCycle(beg)){

out.println("Был найден цикл, введите ребра заново.");

ArrayList<Integer> cycle = new ArrayList<>();

cycle.add(cycle\_beg);

for (int v=cycle\_end; v!=cycle\_beg; v=path.get(v))

cycle.add(v);

cycle.add(cycle\_end);

for (int i=cycle.size()-1; i>=0; i--)

out.print(cycle.get(i)+1+" ");

out.print("\n");

continue;

}

}else{

int siz = begins.size();

int beg = 0;

for(int i = 0; i < siz; i++)

beg = begins.iterator().next();

if(findCycle(beg)){

out.println("Был найден цикл, введите ребра заново.");

ArrayList<Integer> cycle = new ArrayList<>();

cycle.add(cycle\_beg);

for (int v=cycle\_end; v!=cycle\_beg; v=path.get(v))

cycle.add(v);

// cycle.add(cycle\_end);

for (int i=cycle.size()-1; i>=0; i--)

out.print(cycle.get(i)+" ");

out.println("\n");

continue;

}

}

//Флаг на то что была добавлена фиктивная точка на место начальной

boolean addeddot = false;

//Если нашлось больше чем одна начальная точка

if (begins.size() > 1) {

out.print("Было найдено " + begins.size() + " начальных точек:");

for (Integer obj : begins) {

out.print(obj + " ");

}

out.print("\nВарианты решения:\n1)Добавить фиктивную точку\n2)Ввести рёбра заново\n");

out.print("Выбранный вариант: ");

int choose = in.nextInt();

if(choose == 1){

int len = nods.size()+1;

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

if(!nods.contains(i)){

nods.add(i);

for(Integer j : begins){

links.add(new Triple(i, j, 0));

}

begins.clear();

begins.add(i);

break;

}

}

addeddot = true;

out.println("Была добавлена начальная фиктивная точка №"+begins.iterator().next());

}else if(choose == 2){

continue;

}

}

//Если не было найдено начальной точки

else if(begins.size() == 0){

out.println("Было найдено 0 начальных точек. Введите ребра заново");

continue;

}

//Было найдено больше одной конечной точки

if (ends.size() > 1) {

out.print("Было найдено " + ends.size() + " конечных точек:");

for (Integer obj : ends) {

out.print(obj + " ");

}

out.print("\nВарианты решения:\n1)Добавить фиктивная точку\n2)Ввести рёбра заново\n");

out.print("Выбранный вариант: ");

int choose = in.nextInt();

if(choose == 1){

int len = nods.size()+1;

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

if(!nods.contains(i)){

nods.add(i);

for(Integer j : ends){

links.add(new Triple(j, i, 0));

}

ends.clear();

ends.add(i);

break;

}

}

out.println("Была добавлена конечная фиктивная точка №"+ends.iterator().next());

}else if(choose == 2){

continue;

}

}

//Не было найдено конечной точки

else if(ends.size() == 0){

out.println("Было найдено 0 конечных точек. Введите ребра заново");

continue;

}

//Порядок рёбер

order = new HashMap<>();

for(Integer nod : nods){

order.put(nod,0);

}

//Упорядочивание вершин

DFS(ends.iterator().next());

//Упорядочивание ребер в зависимости от вершин из которых он выходит

for(Integer nod:nods){

Node tmp = new Node(nod);

if(nod.equals(begins.iterator().next()))

start = tmp;

else if(nod.equals(ends.iterator().next()))

finish = tmp;

Nods.put(nod,tmp);

}

//Заполняем раннее время

fillTr();

//Раннее время конечной точки равно позднему времени

finish.tp = finish.tr;

//Заполняем позднее время

fillTp();

//Выводим список вершин с параметрами

out.println("Вершины:\nНомер | Ранний срок | Поздний срок | Резерв времени |");

for (Integer nod: nods){

Node Nod = Nods.get(nod);

Nod.p = Nod.tp - Nod.tr;

out.format("%5d |%12d |%13d |%15d |\n",Nod.name,Nod.tr,Nod.tp,Nod.p);

}

out.println("Начальная вершина:" + begins.iterator().next());

out.println("Конечная вершина:" + ends.iterator().next());

//Находим максимальный порядок

int maxord = 0;

for (Integer nod : nods) {

maxord = Math.max(order.get(nod),maxord);

}

for(Triple link : links){

link.fullreserv = Nods.get(link.end).tp - Nods.get(link.beg).tr - link.weight;

link.freereserv = Nods.get(link.end).tr - Nods.get(link.beg).tp - link.weight;

}

//Выводим ребра с порядком

out.print("\nРёбра:\n" +

" Откуда | Куда | Вес | Полный резерв | Свободный резерв |");

int curOrder;

if(addeddot)

curOrder = 2;

else

curOrder = 1;

for(; curOrder < maxord+1; curOrder++) {

out.println("\nТекущий порядок: "+curOrder);

for (Integer nod : nods) {

if (order.get(nod) == curOrder-1){

for(Triple link:links){

if(link.beg == nod)

out.format("%7d |%12d |%13d|%15d|%18d|\n", link.beg, link.end, link.weight, link.fullreserv, link.freereserv);

}

}

}

}

//Ищем критический путь

criticalPath(new ArrayList<>(),begins.iterator().next());

//Выводим число критических путей

int quantity = pathways.size();

out.println("\nБыло найдено " + quantity + " критических путей.");

//Выводим по очереди найденные пути

if(quantity != 0) {

out.println("Найденные критические пути:");

for (String s : pathways) {

int leng = 0;

out.print(s);

String[] sp = s.split(" ");

for(int i = 0 ; i < sp.length - 1; i++){

int b = Integer.parseInt(sp[i]);

int e = Integer.parseInt(sp[i+1]);

for(Triple link:links)

if(link.beg == b && link.end == e)

leng += link.weight;

}

out.print(" Длина = "+leng+"\n");

}

}

break;

}

}

//Поиск цикла методом раскраски

private static boolean findCycle(int cur){

color.put(cur,1);

for(Triple link : links){

if(link.beg == cur){

int to = link.end;

if(color.get(link.end) == 0){

path.put(to,cur);

if(findCycle(link.end))

return true;

}else if(color.get(link.end) == 1){

cycle\_end = cur;

cycle\_beg = to;

return true;

}

}

}

color.put(cur,2);

return false;

}

//Поиск длиннейшего пути до вершины

private static int DFS(int cur){

int max = -1;

for(Triple link : links){

if(link.end == cur){

if(link.beg == begins.iterator().next())

max = Math.max(max,0);

else

max = Math.max(max,DFS(link.beg));

}

}

order.put(cur,max+1);

return max + 1;

}

//Заполняем раннее время у вершин обходом в ширину от начальной точки

private static void fillTr(){

LinkedList<Node> queue = new LinkedList<>();

queue.add(start);

while(!queue.isEmpty()) {

Node cur = queue.poll();

for (Triple link : links) {

if (link.beg == cur.name){

Node ending = Nods.get(link.end);

queue.add(ending);

ending.tr = Math.max(ending.tr,link.weight+cur.tr);

}

}

}

}

//Заполняем позднее время у вершин обходом в ширину от конечной точки

private static void fillTp(){

LinkedList<Node> queue = new LinkedList<>();

queue.add(finish);

while(!queue.isEmpty()) {

Node cur = queue.poll();

for (Triple link : links) {

if (link.end == cur.name){

Node begining = Nods.get(link.beg);

queue.add(begining);

begining.tp = Math.min(begining.tp,cur.tp-link.weight);

}

}

}

}

//Поиск критического пути обходом в глубину, без сохранения посещенных вершин

private static void criticalPath(ArrayList<Integer> previous, int current){

previous.add(current);

if(current == ends.iterator().next()){

String s = "";

for(Integer node : previous){

s+=node+" ";

}

s = s.substring(0,s.length()-1);

pathways.add(s);

return;

}

for(Triple link:links){

if(link.beg == current && link.fullreserv == 0){

criticalPath((ArrayList<Integer>)previous.clone(),link.end);

}

}

}

//Ребро начало, конец, вес

private static class Triple{

private int beg;

private int end;

private int weight;

private int fullreserv;

private int freereserv;

private Triple(int \_b,int \_e, int \_w){

beg = \_b;

end = \_e;

weight = \_w;

fullreserv = 0;

freereserv = 0;

}

}

//Узел название, раннее время, конечное время, резерв

private static class Node{

private int name;

private int tr;

private int tp;

private int p;

Node(int \_n){

name = \_n;

tr = 0;

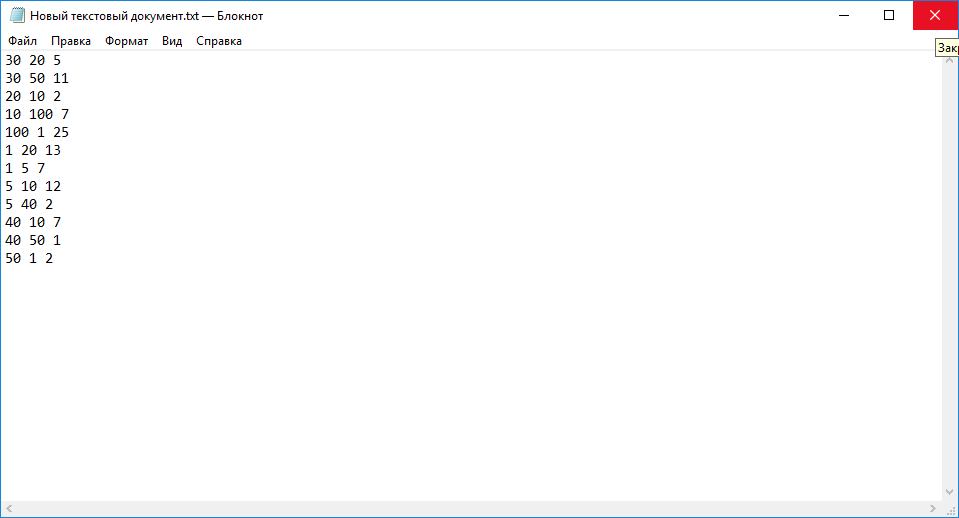
tp = Integer.MAX\_VALUE;

p = 0;

}

}

Тестовый пример



Результат





