МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ  
РОССИЙСКОЙ ФЕДЕРАЦИИ  
ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ  
Кафедра «Вычислительной техники»

**ОТЧЁТ**

по лабораторной работе №6  
по курсу «Логика и основы алгоритмизации в инженерных задачах»  
на тему «Унарные и бинарные операции над графами»

Выполнили:

Студенты группы 23ВВВ2

Лисов Е.А.

Кочегин В.В.

Приняли:

Митрохин М. А.  
Юрова О.В.

Пенза 2024

**Цель работы**

Изучении представления графов с помощью матриц смежности и списков смежности, выполнении унарных операций: отождествление вершин, стягивание ребер и расщепление вершин и бинарных операций: объединения, пересечения, кольцевой суммы графов и декартового произведения графов.

**Задание**

**Задание 1:**

1. Сгенерируйте (используя генератор случайных чисел) матрицу смежности для неориентированного графа *G*. Выведите матрицу на экран.
2. \*Для указанных графов преобразуйте представление матриц смежности в списки смежности. Выведите полученные списки смежности.

**Задание 2:**

1. Для матричной формы представления графов выполните операцию:  
   а) отождествление вершин

б) стягивание ребра

в) расщепление вершины

Номера выбираемых для операций вершин ввести с клавиатуры.

Результат выполнения  вывести на экран.

1. \*Для представления графов в виде списков смежности выполните операцию:  
   а) отождествление вершин

б) стягивание ребра

в) расщепление вершины

Номера выбираемых для операций вершин ввести с клавиатуры.

Результат выполнения  вывести на экран.

**Задание 3:**

1. Для матричной формы представления графов выполните операцию:

а) объединения

б) пересечения

в) кольцевой суммы

Результат выполнения  вывести на экран.

**Задание 4\*:**

1. Для матричной формы представления графов выполните операцию

декартового произведения графов.

Результат выполнения  вывести на экран.

**Ход работы**

**Задание 1.**

1. Для генерации матрицы смежности используется функция create, в которой происходит выделение памяти под массив для хранения матрицы смежности и его заполнения:

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

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

G[i][j] = rand() % 2;

if (i == j) G[i][j] = 0;

G[j][i] = G[i][j];

}

}

Вывод матрицы происходит с помощью функции printG().

2\*. Для создания списка смежности графа используется матрица смежности графа, Для этого в функции edges() программа проходит по всем элементам матрицы и добавляем ребра в граф, если элемент равен 1:

struct Graph\* graph = createAGraph(size);

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

for (int j = i; j < size; j++) {

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

addEdge(graph, i, j);

}

}

}

return graph;

Далее функция createGraph() выделяет память под структуру графа и инициализирует массив списков смежности:

struct Graph\* graph = (Graph\*)malloc(sizeof(struct Graph));

graph->numVertices = vertices;

graph->adjLists = (node\*\*)malloc(vertices \* sizeof(struct node\*));

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

graph->adjLists[i] = NULL;

}

return graph;

После функция addEdge добавляет ребро между двумя вершинами, проверяя есть ли у верши связи между ними:  
  
 struct node\* temp = graph->adjLists[s];

while (temp) {

if (temp->vertex == d) {

return;

}

temp = temp->next;

}

struct node\* newNode = createNode(d);

newNode->next = graph->adjLists[s];

graph->adjLists[s] = newNode;

temp = graph->adjLists[d];

while (temp) {

if (temp->vertex == s) {

return;

}

temp = temp->next;

}

newNode = createNode(s);

newNode->next = graph->adjLists[d];

graph->adjLists[d] = newNode;

Функция createNode выделяет память под новый узел, инициализирует его:

struct node\* newNode = (node\*)malloc(sizeof(struct node));

newNode->vertex = v;

newNode->next = NULL;

return newNode;

В конце выводится граф с помощью функции printGraph():

for (int v = 0; v < graph->numVertices; v++) {

struct node\* temp = graph->adjLists[v];

printf("\nVertex %d: ", v+1);

while (temp) {

printf("%d -> ", temp->vertex + 1);

temp = temp->next;

}

printf("\n");

}

**Задание 2.**

1. а) Для отождествления вершин в матрице смежности пользователь вводит номера двух вершин, которые нужно объединить, функция unionV объединяет их, переназначая связи одной вершины на другую и удаляя вторую вершину из матрицы смежности:

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

if (G[v2][i] == 1) {

G[v1][i] = G[v2][i];

G[i][v1] = G[i][v2];

}

}

G = delV(G, size, v2);

return G;

Функция del() удаляет вершину из матрицы смежности.

б) Для осуществления стягивания ребра пользователь вводит две вершины, между которыми нужно стянуть ребро, далее функция contrE проверяет наличие ребра и удаляет его, а также объединяет соседей:

if (G[v1][v2] != 1) {

printf("reb no\n");

\*flag = 1;

return G;

}

else {

G[v1][v2] = 0;

G[v2][v1] = 0;

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

if (G[v2][i] == 1) {

G[v1][i] = G[v2][i];

G[i][v1] = G[i][v2];

}

}

G = delV(G, size, v2);

return G;

в) Для расщепления вершины пользователь вводит вершину, которую нужно расщепить, далее функция splitV() создает новую вершину и перенаправляет связи:

int\*\* Gtemp = createG(size + 1);

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

Gtemp[i][size] = G[i][v];

Gtemp[size][i] = G[v][i];

}

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

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

Gtemp[i][j] = G[i][j];

}

}

Gtemp[v][size] = Gtemp[size][v] = 1;

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

free(G[i]);

}

free(G);

G = NULL;

return Gtemp;

2\*. а) Для объединения двух вершин пользователь вводит их, далее функция unionV объединяет две вершины, удаляя одну из них и перенаправляя все ее соседние вершины на другую:  
 if (v1 == v2) return;

struct node\* temp = graph->adjLists[v2];

while (temp) {

addEdge(graph, v1, temp->vertex);

temp = temp->next;

}

for (int i = 0; i < graph->numVertices; i++) {

struct node\* adj = graph->adjLists[i];

struct node\* p = NULL;

while (adj) {

if (adj->vertex == v2) {

if (p) {

p->next = adj->next;

}

else {

graph->adjLists[i] = adj->next;

}

struct node\* del = adj;

adj = adj->next;

free(del);

}

else {

p = adj;

adj = adj->next;

}

}

}

graph->adjLists[v2] = NULL;

for (int i = v2 + 1; i < graph->numVertices; i++) {

graph->adjLists[i - 1] = graph->adjLists[i];

}

graph->adjLists[graph->numVertices - 1] = NULL;

graph->numVertices--;

for (int i = 0; i < graph->numVertices; i++) {

struct node\* adj = graph->adjLists[i];

while (adj) {

if (adj->vertex > v2) {

adj->vertex--;

}

adj = adj->next;

}

}

б) Для стягивания ребра пользователь вводит две вершины, далее функция styag() удаляет ребро между ними, объединяет их соседей и удаляет одну из них:  
 if (v1 == v2) return;

int flag = 0;

struct node\* temp = graph->adjLists[v1];

while (temp) {

if (temp->vertex == v2) {

flag = 1;

}

temp = temp->next;

}

if (flag) {

struct node\* adj = graph->adjLists[v1];

struct node\* p = NULL;

while (adj) {

if (adj->vertex == v2) {

if (p) {

p->next = adj->next;

}

else {

graph->adjLists[v1] = adj->next;

}

struct node\* del = adj;

adj = adj->next;

free(del);

break;

}

else {

p = adj;

adj = adj->next;

}

}

adj = graph->adjLists[v2];

p = NULL;

while (adj) {

if (adj->vertex == v1) {

if (p) {

p->next = adj->next;

}

else {

graph->adjLists[v2] = adj->next;

}

struct node\* del = adj;

adj = adj->next;

free(del);

break;

}

else {

p = adj;

adj = adj->next;

}

}

struct node\* temp = graph->adjLists[v2];

while (temp) {

addEdge(graph, v1, temp->vertex);

temp = temp->next;

}

for (int i = 0; i < graph->numVertices; i++) {

adj = graph->adjLists[i];

p = NULL;

while (adj) {

if (adj->vertex == v2) {

if (p) {

p->next = adj->next;

}

else {

graph->adjLists[i] = adj->next;

}

struct node\* del = adj;

adj = adj->next;

free(del);

}

else {

p = adj;

adj = adj->next;

}

}

}

graph->adjLists[v2] = NULL;

for (int i = v2 + 1; i < graph->numVertices; i++) {

graph->adjLists[i - 1] = graph->adjLists[i];

}

graph->adjLists[graph->numVertices - 1] = NULL;

graph->numVertices--;

for (int i = 0; i < graph->numVertices; i++) {

struct node\* adj = graph->adjLists[i];

while (adj) {

if (adj->vertex > v2) {

adj->vertex--;

}

adj = adj->next;

}

}

}

else { return; }

в) Для расщепления вершины пользователь вводит номер вершины, далее функция splitV() создает новую вершину из введенной пользователем, сохраняя все ее связи и соединяет новую с введенной:

if (v < 0 || v >= graph->numVertices) return;

int newV1 = graph->numVertices;

int newV2 = graph->numVertices + 1;

graph->numVertices += 1;

graph->adjLists = (struct node\*\*)realloc(graph->adjLists, graph->numVertices \* sizeof(struct node\*));

graph->adjLists[newV1] = NULL;

struct node\* temp = graph->adjLists[v];

while (temp) {

addEdge(graph, newV1, temp->vertex);

temp = temp->next;

}

addEdge(graph, newV1, v);

**Задание 3.**

1. а) Для объединения графов используется функция unionG(), определяется максимальный и минимальный размер графа, создается новый граф с максимальным размером, который будет содержать все ребра из обоих графов:

int sizemax = (size1 > size2) ? size1 : size2;

int sizemin = (size1 < size2) ? size1 : size2;

int\*\* Gmax = (size1 > size2) ? G1 : G2;

int\*\* Gmin = (size1 < size2) ? G1 : G2;

int\*\* Gtemp = createG(sizemax);

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

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

Gtemp[i][j] = Gmin[i][j] | Gmax[i][j];

}

}

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

for (int j = sizemin; j < sizemax; j++) {

Gtemp[i][j] = Gmax[i][j];

Gtemp[j][i] = Gtemp[i][j];

}

}

return Gtemp;

б) Для пересечения графов используется функция intersectionG(), она находит пересечения двух графов, создает новый граф, который содержит только те ребра, которые присутствуют только в обоих графах:

int sizemin = (size1 < size2) ? size1 : size2;

int\*\* Gtemp = createG(sizemin);

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

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

Gtemp[i][j] = G1[i][j] & G2[i][j];

}

}

return Gtemp;

в) Для кольцевой суммы гравоф используется функция xorG(), определяется максимальный и минимальный граф, создает новый граф, размер который соответствует большему, далее выполняется операция xor между соответствующими элементами и происходит запись остальных элементов:

int sizemax = (size1 > size2) ? size1 : size2;

int sizemin = (size1 < size2) ? size1 : size2;

int\*\* Gmax = (size1 > size2) ? G1 : G2;

int\*\* Gmin = (size1 < size2) ? G1 : G2;

int\*\* Gtemp = createG(sizemax);

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

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

if (i < sizemin && j < sizemin) {

Gtemp[i][j] = Gmax[i][j] ^ Gmin[i][j];

}

else {

Gtemp[i][j] = Gmax[i][j];

}

}

}

return Gtemp;

**Задание 4\*.**

1. Для выполнения декартового произведения используется функция decUG(), создается матрица размером (Размер1\*Зазмер2) х (Размер1\*Размер2), далле используются 4 цикла, если есть ребро между i и k в 1-ом графе и между j и l во 2-ом графе => устанавливается соединение в созданном графе для пар (i,j)( и (k,l):

for (int i = 0; i < size1 \* size2; i++) {

for (int j = 0; j < size1 \* size2; j++) {

G3[i][j] = 0;

}

}

for (int i = 0; i < size1; i++) {

for (int j = 0; j < size2; j++) {

for (int k = 0; k < size1; k++) {

for (int l = 0; l < size2; l++) {

if (G1[i][k] == 1 && j == l) {

G3[i \* size2 + j][k \* size2 + l] = 1;

}

if (G2[j][l] == 1 && i == k) {

G3[i \* size2 + j][k \* size2 + l] = 1;

}

}

}

}

}

return G3;

### Результаты работы программы

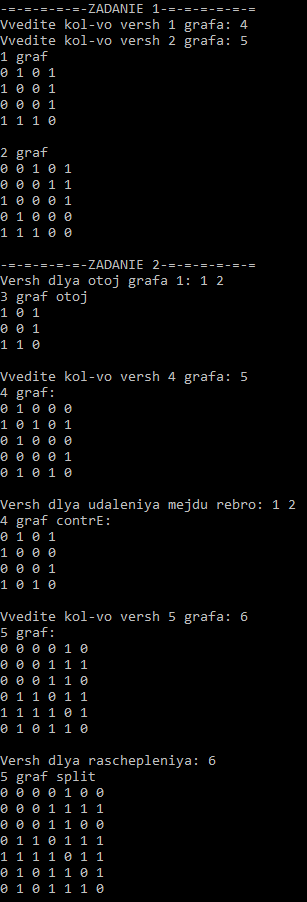


Рисунок 1 — Результаты работы программы для заданий 1 и 2 без \*

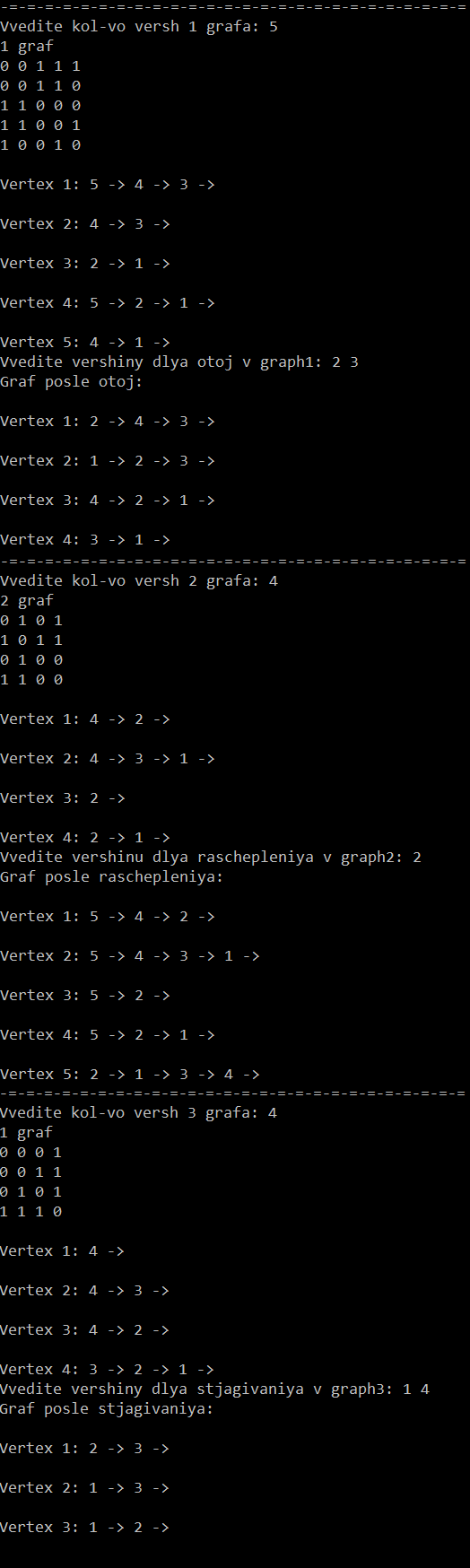
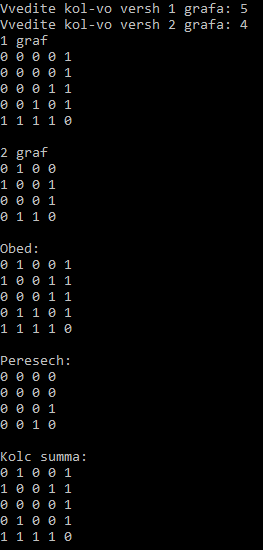
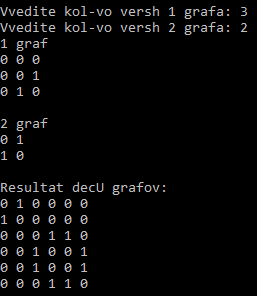


Рисунок 2 — Результаты работы программы для заданий 1 и 2 с \*

  
Рисунок 3 — Результаты работы программы для задания 3

  
Рисунок 4 — Результаты работы программы для задания 4\*

### Вывод

В ходе выполнения работы были изучены представления графов с помощью матриц смежности и списков смежности, способы выполнение унарных операций: отождествление вершин, стягивание ребер и расщепление вершины и способы выполнения бинарных операций: объединения, пересечения, кольцевой суммы графов и декартового произведения графов.