Министерство образования Российской Федерации

Пензенский государственный университет

Кафедра «Вычислительная техника»

Отчёт

*по лабораторной работе №3*

*По дисциплине: «Логика и основы алгоритмизации в инженерных задачах»*

*На тему: «Унарные и бинарные операции над графами»*

**Выполнил студенты группы 19ВВ3:**

Ерёмин А.А

**Принял:**

Митрохин М.А

Пенза 2020

**Цель:** изучить и реализовать унарные и бинарные операции над графами.

**Общие сведения.**

Все унарные операции над графами можно объединить в две группы. Первую группу составляют операции, с помощью которых из исходного графа *G*1*,* можно построить граф *G*2 с меньшим числом элементов. В группу входят операции удаления ребра или вершины, отождествления вершин, стягивание ребра. Вторую группу составляют операции, позволяющие строить графы с большим числом элементов. В группу входят операции расщепления вершин, добавления ребра.

*Отождествление вершин.* В графе *G*1 выделяются вершины *и,v.* Определяют окружение *Q*1 вершины *u*,и окружение *Q*2 вершины *v,* вычисляют их объединение *Q* = *Q1* * Q2.* Затем над графом *G*1 выполняются следующие преобразования:

* из графа *G*1 удаляют вершины *u,* *v (H*1 *= G*1 *- u - v);*
* к графу *Н*1присоединяют новую вершину *z (H*1 *= H*1 *+z);*
* вершину *z* соединяют ребром с каждой из вершин *w*1*Q*

*(G*2 *= H*1 *+ zwi*, *i =* 1,2,3*,…).*

*Стягивание ребра.* Данная операция является операцией отождествления смежных вершин *и, v* в графе *G*1.

Наиболее важными бинарными операциями являются: объединение, пересечение, декартово произведение и кольцевая сумма.

*Объединение.* Граф *G* называется объединением или наложением графов *G*1 и *G*2, если *VG = V*1*V*2*; UG = U*1* U*2 (рис. 1).

**U**

*v*1

*v*2

*v*3

*v*4

*v*3

*v*4

*v*5

*v*2

*v*1

*v*3

*v*4

*v*5

Рис. 1. Объединение графов *G*1, *G*2

Объединение графов *G*1 и *G*2 называется дизъюнктным, если *V*1*V*2 *= *. При дизъюнктном объединении никакие два из объединяемых графов не должны иметь общих вершин.

*Пересечение.* Граф *G* называется пересечением графов *G*1, *G*2,если *VG = V*1*V*2и *UG = U*1*U*2 (риc.2). Операция "пересечения" записывается следующим образом: *G = G*1*G*2*.*

**∩**

*v*1

*v*2

*v*3

*v*5

*v*3

*v*4

*v*6

*v*2

*v*1

*v*6

*v*4

*v*5

*v*1

*v*4

*v*6

*v*5

*v*3

*v*2

Рис.2. Пересечение графов *G*1, *G*2*.*

*Декартово произведение.* Граф *G* называется декартовым произведением графов *G*1 и *G*2 если *VG* = *V*1*V*2 —декартово произведение множеств вершин графов *G*1, *G*2, а множество ребер *U*c задается следующим образом: вершины (*zi*, *vk*) и (*zj*, *vl*) смежны в графе *G* тогда и только тогда, когда *zi* = *zj*(*i* = *j*), a *v*k и *vl* смежны в *G*2 или *vk* = *vl*(*k* = *l*), смежны в графе *G*1 (см. рис.3).

**X**

*z*1

*z*2

*v*1

*v*3

*v*2

*z*1*v*1

*z*1*v*2

*z*1*v*3

*z*2*v*1

*z*2*v*2

*z*2*v*3

Рис. 3. Декартово произведение графов *G*1, *G*2

*Кольцевая сумма* графов представляет граф, который не имеет изолированных вершин и состоит из ребер, присутствующих либо в первом исходном графе, либо во втором. Кольцевая сумма определяется следующим соотношением: *G* = *G*1  *G*2 (рис.4).

**⊕**

*v*1

*v*2

*v*3

*v*5

*v*3

*v*4

*v*6

*v*2

*v*1

*v*4

*v*5

*v*1

*v*4

*v*6

*v*5

*v*3

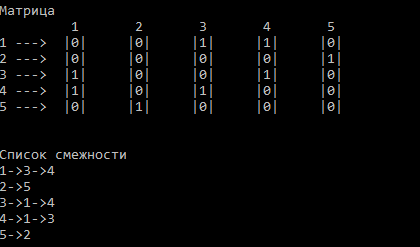
*v*2

Рис.4. Кольцевая сумма графов *G1, G2*

**Ход работы:**

**Задание 1.**

1. Сгенерируйте (используя генератор случайных чисел) две матрицы *M*1*, М*2 смежности неориентированных помеченных графов *G*1, *G*2. Выведите сгенерированные матрицы на экран.
2. \* Для указанных графов преобразуйте представление матриц смежности в списки смежности. Выведите полученные списки на экран.



int\*\* matrix\_malloc(int MatrixOrder) {

int\*\* Matrix = (int\*\*)malloc(sizeof(int) \* MatrixOrder);

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

Matrix[i] = (int\*)malloc(sizeof(int) \* MatrixOrder);

}

return Matrix;

}

int\* vertexes\_malloc(int MatrixOrder) {

int\* Vertexes = (int\*)malloc(sizeof(int) \* MatrixOrder);

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

Vertexes[i] = i + 1;

}

return Vertexes;

}

Graph\* graph\_create(int MatrixOrder) {

Graph\* pGraph = (Graph\*)malloc(sizeof(Graph));

pGraph->Matrix = matrix\_malloc(MatrixOrder);

pGraph->Vertexes = vertexes\_malloc(MatrixOrder);

pGraph->MatrixOrder = MatrixOrder;

return pGraph;

}

void graph\_free(Graph\* GraphG) {

int MatrixOrder = GraphG->MatrixOrder;

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

free(GraphG->Matrix[i]);

}

free(GraphG->Matrix);

free(GraphG->Vertexes);

free(GraphG);

}

int randFunc() {

if (rand() % 101 <= 50) {

return 1;

}

else {

return 0;

}

}

void graph\_random(Graph\* GraphG) {

int MatrixOrder = GraphG->MatrixOrder;

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

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

if (i == j) {

GraphG->Matrix[i][j] = 0;

}

else {

GraphG->Matrix[i][j] = GraphG->Matrix[j][i] = randFunc();

}

}

}

}

void matrix\_print(Graph\* GraphG) {

printf("\n");

int MatrixOrder = GraphG->MatrixOrder;

printf("\t");

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

printf(" %d\t", GraphG->Vertexes[i]);

}

printf("\n");

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

printf("%d --->\t", GraphG->Vertexes[i]);

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

printf("|%d|\t", GraphG->Matrix[i][j]);

}

printf("\n");

}

printf("\n");

}

|  |
| --- |
|  |
| GraphList\* create\_graph(int vertexes) { |
|  | GraphList\* graph = (GraphList\*)malloc(sizeof(GraphList)); |
|  | graph->numVertex = vertexes; |
|  | graph->heads = (Node\*\*)malloc(sizeof(Node\*\*) \* vertexes); |
|  | graph->tails = (Node\*\*)malloc(sizeof(Node\*\*) \* vertexes); |
|  | graph->sequence\_vertexes = (int\*)malloc(sizeof(int) \* vertexes); |
|  | for (int i = 0; i < vertexes; i++) { |
|  | graph->heads[i] = NULL; |
|  | graph->tails[i] = NULL; |
|  | graph->sequence\_vertexes[i] = i; |
|  | } |
|  | return graph; |
|  | } |
|  |  |
|  | Node\* create\_node(int vertex) { |
|  | Node\* node = (Node\*)malloc(sizeof(Node)); |
|  | node->vertex = vertex; |
|  | node->next = NULL; |
|  | return node; |
|  | } |
|  |  |

**Задание 2.**

1. Для матричной формы представления графов выполните операцию:

а) отождествления вершин

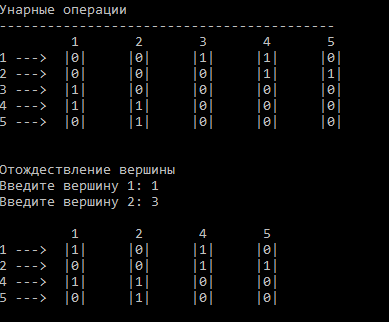
б) стягивания ребра

в) расщепления вершины

Номера выбираемых для выполнения операции вершин ввести с клавиатуры.

Результат выполнения операции выведите на экран.

А)



void VertexIndentification(Graph\* GraphG) {

int u, v;

int MatrixOrder = GraphG->MatrixOrder;

printf("\nОтождествление вершины\n");

printf("Введите вершину 1: ");

scanf("%d", &u);

printf("Введите вершину 2: ");

scanf("%d", &v);

u--;

v--;

if (u > v) {

int temp = v;

v = u;

u = temp;

}

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

GraphG->Matrix[u][i] = GraphG->Matrix[v][i] || GraphG->Matrix[u][i]; // Само отождествление.

GraphG->Matrix[i][u] = GraphG->Matrix[i][v] || GraphG->Matrix[i][u];

}

Graph\* IndentGraph = graph\_create(MatrixOrder - 1); // Создаём temp матрицу с меньшим на 1 порядком.

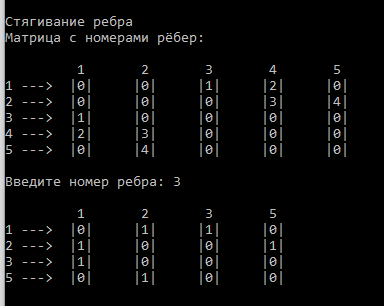
matrix\_copydec(GraphG, IndentGraph, v); // переносим туда значения, пропуская вершину одну из отождествленных вершин.

matrix\_print(IndentGraph);

free(IndentGraph);

}

Б)



void EdgeContract(Graph\* GraphG) {

int MatrixOrder = GraphG->MatrixOrder;

Graph\* NumEdgeGraph = graph\_create(MatrixOrder); // Создаём матрицу с номерами рёбер

int u, v;

int NumberEdge;

bool is\_found = false;

printf("\nСтягивание ребра\n");

printf("Матрица с номерами рёбер:\n ");

matrix\_edgeName(GraphG, NumEdgeGraph);

matrix\_print(NumEdgeGraph);

printf("Введите номер ребра: ");

scanf("%d", &NumberEdge);

for (int i = 0; i < MatrixOrder && is\_found == false; i++) {

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

if (NumEdgeGraph->Matrix[i][j] == NumberEdge) {

u = i;

v = j;

GraphG->Matrix[i][j] = GraphG->Matrix[j][i] = 0; // удаляем петлю.

is\_found = true;

break;

}

}

}

graph\_free(NumEdgeGraph);

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

GraphG->Matrix[u][i] = GraphG->Matrix[v][i] || GraphG->Matrix[u][i]; // Этап отождествления, т.к стягивание ребра – это отождествление с удалением петли.

GraphG->Matrix[i][u] = GraphG->Matrix[i][v] || GraphG->Matrix[i][u];

}

Graph\* IndentGraph = graph\_create(MatrixOrder - 1);

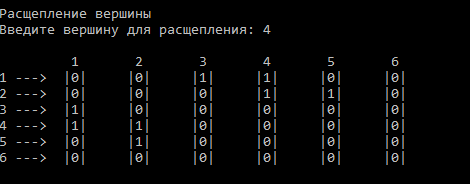
matrix\_copydec(GraphG, IndentGraph, v);

matrix\_print(IndentGraph);

free(IndentGraph);

}

В)



void VertexSplit(Graph\* GraphG) {

int MatrixOrder = GraphG->MatrixOrder;

int u;

printf("\nРасщепление вершины\n");

printf("Введите вершину для расщепления: ");

scanf("%d", &u);

u--;

Graph\* IndentGraph = graph\_create(MatrixOrder + 1); // создаём temp матрицу с порядком на 1 больше, т.к в результате расщепления появится новая вершина.

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

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

IndentGraph->Matrix[i][j] = GraphG->Matrix[i][j];

}

}

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

if (GraphG->Matrix[u][i] == 1) {

if (rand() % 101 <= 40) { // функция переноса смежностей в новую вершину

IndentGraph->Matrix[MatrixOrder][i] = 1;

IndentGraph->Matrix[i][MatrixOrder] = 1;

IndentGraph->Matrix[u][i] = 0;

IndentGraph->Matrix[i][u] = 0;

}

else {

IndentGraph->Matrix[MatrixOrder][i] = 0;

IndentGraph->Matrix[i][MatrixOrder] = 0;

}

}

else {

IndentGraph->Matrix[MatrixOrder][i] = 0;

IndentGraph->Matrix[i][MatrixOrder] = 0;

}

}

IndentGraph->Matrix[MatrixOrder][MatrixOrder] = 0;

matrix\_print(IndentGraph);

free(IndentGraph);

}

1. \* Для представления графов в виде списков смежности выполните операцию:

а) отождествления вершин

б) стягивания ребра

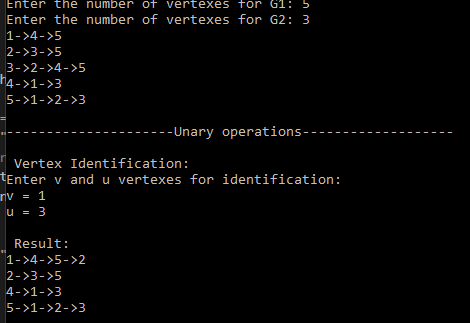
в) расщепления вершины

Номера выбираемых для выполнения операции вершин ввести с клавиатуры.

Результат выполнения операции выведите на экран.

Из-за ошибок с памятью, пришлось сделать операции на списках смежности в отдельной ветке и программе.

А)



void vertex\_identification(GraphList\* graph) {

printf("\n Vertex Identification: \n");

printf("Enter v and u vertexes for identification: \n");

printf("v = ");

int v, u;

scanf("%d", &v);

printf("u = ");

scanf("%d", &u);

if (v > u) {

int tmp = v;

v = u;

u = tmp;

}

v--;

u--;

GraphList\* newGraph = create\_graph(graph->numVertex - 1);

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

Node\* copyCurrent = graph->heads[i];

while (copyCurrent) {

AddLast(newGraph, copyCurrent->vertex, i);

copyCurrent = copyCurrent->next;

}

}

for (int i = u; i < newGraph->numVertex; i++) {

Node\* copyCurrent = graph->heads[i + 1];

while (copyCurrent) {

AddLast(newGraph, copyCurrent->vertex, i);

copyCurrent = copyCurrent->next;

}

}

for (int i = u; i < newGraph->numVertex; i++) {

newGraph->sequence\_vertexes[i] = i + 1;

}

Node\* currentU = graph->heads[u]; // Бежим по вершине с большим номером.

while (currentU) {

if (find\_node(graph, currentU->vertex, v) != NULL) {

currentU = currentU->next;

}

else { // если не нашли в вершине с меньшим номером связи, добавляем

AddLast(newGraph, currentU->vertex, v);

currentU = currentU->next;

}

}

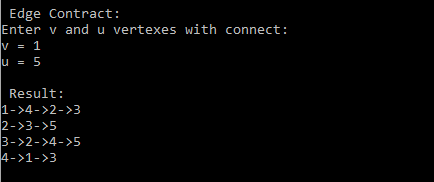
printf("\n Result: \n");

print\_graph(newGraph);

graph\_delete(newGraph);

}

Б)



void edge\_contract(GraphList\* graph) {

printf("\n Edge Contract: \n");

printf("Enter v and u vertexes with connect: \n");

printf("v = ");

int v, u;

scanf("%d", &v);

printf("u = ");

scanf("%d", &u);

if (v > u) {

int tmp = v;

v = u;

u = tmp;

}

v--;

u--;

if (find\_node(graph, u, v) == NULL) {

printf("Connect does not exist!");

return;

}

GraphList\* newGraph = create\_graph(graph->numVertex - 1);

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

Node\* copyCurrent = graph->heads[i];

while (copyCurrent) {

AddLast(newGraph, copyCurrent->vertex, i);

copyCurrent = copyCurrent->next;

}

}

for (int i = u; i < newGraph->numVertex; i++) {

Node\* copyCurrent = graph->heads[i + 1];

while (copyCurrent) {

AddLast(newGraph, copyCurrent->vertex, i);

copyCurrent = copyCurrent->next;

}

}

for (int i = u; i < newGraph->numVertex; i++) {

newGraph->sequence\_vertexes[i] = i + 1;

}

Node\* currentU = graph->heads[u]; // Бежим по вершине с большим номером.

while (currentU) {

if (find\_node(graph, currentU->vertex, v) != NULL) {

currentU = currentU->next;

}

else { если не нашли в вершине с меньшим номером связи, добавляем

AddLast(newGraph, currentU->vertex, v);

currentU = currentU->next;

}

}

Node\* DeleteNode = find\_node(newGraph, v, v);

node\_delete(newGraph, v, DeleteNode);

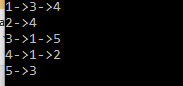
printf("\n Result: \n");

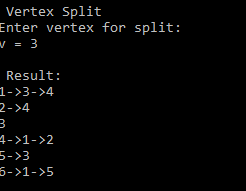
print\_graph(newGraph);

graph\_delete(newGraph);

}

В)





void split\_vertex(GraphList\* graph) {

printf("\n Vertex Split \n");

printf("Enter vertex for split: \nv = ");

int v;

scanf("%d", &v);

v--;

GraphList\* newGraph = create\_graph(graph->numVertex + 1);

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

Node\* copyCurrent = graph->heads[i];

while (copyCurrent) {

AddLast(newGraph, copyCurrent->vertex, i);

copyCurrent = copyCurrent->next;

}

}

int newVertex = graph->numVertex;

Node\* currentV = newGraph->heads[v];

while (currentV) {

if (rand() % 2 == 1) { // Функция случайного переноса.

AddLast(newGraph, currentV->vertex, newVertex);

Node\* OldNode = currentV;

currentV = currentV->next;

node\_delete(newGraph, v, OldNode);

}

else {

currentV = currentV->next;

}

}

printf("\n Result: \n");

print\_graph(newGraph);

graph\_delete(newGraph);

}

**Задание 3.**

1. Для матричной формы представления графов выполните операцию:

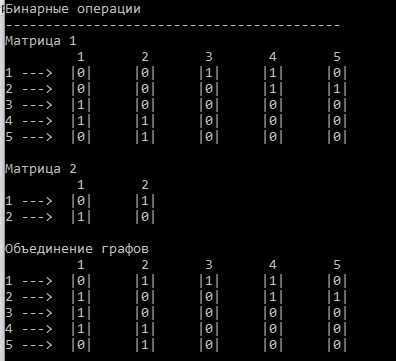
а) объединения *G* = *G*1  *G*2

б) пересечения *G* = *G*1  *G*2

в) кольцевой суммы *G* = *G*1  *G*2

Результат выполнения операции выведите на экран.

А)



void GraphUnion(Graph\* GraphG1, Graph\* GraphG2) {

printf("Объединение графов");

int MatrixOrderMin = GraphG1->MatrixOrder;

int MatrixOrderMax = GraphG2->MatrixOrder;

bool is\_equal = false; // Флаг равенства двух графов

bool is\_moreG1 = false; // Флаг того, что G1 граф больше чем G2

if (MatrixOrderMin == MatrixOrderMax) {

is\_equal = true;

}

else if (GraphG1->MatrixOrder > GraphG2->MatrixOrder) {

MatrixOrderMin = GraphG2->MatrixOrder;

MatrixOrderMax = GraphG1->MatrixOrder;

is\_moreG1 = true;

}

Graph\* GraphG3Union = graph\_create(MatrixOrderMax);

if (is\_equal) {

for (int i = 0; i < GraphG1->MatrixOrder; i++) {

for (int j = 0; j < GraphG1->MatrixOrder; j++) {

GraphG3Union->Matrix[i][j] = GraphG1->Matrix[i][j] || GraphG2->Matrix[i][j];

}

}

}

else {

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

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

GraphG3Union->Matrix[i][j] = GraphG1->Matrix[i][j] || GraphG2->Matrix[i][j]; // Объединяем значения меньшей матрицы

}

}

if (is\_moreG1) { // Перенос значений

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

for (int j = MatrixOrderMin; j < MatrixOrderMax; j++) {

GraphG3Union->Matrix[i][j] = GraphG1->Matrix[i][j];

}

}

for (int i = MatrixOrderMin; i < MatrixOrderMax; i++) {

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

GraphG3Union->Matrix[i][j] = GraphG1->Matrix[i][j];

}

}

}

else {

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

for (int j = MatrixOrderMin; j < MatrixOrderMax; j++) {

GraphG3Union->Matrix[i][j] = GraphG2->Matrix[i][j];

}

}

for (int i = MatrixOrderMin; i < MatrixOrderMax; i++) {

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

GraphG3Union->Matrix[i][j] = GraphG2->Matrix[i][j];

}

}

}

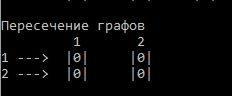
}

matrix\_print(GraphG3Union);

graph\_free(GraphG3Union);

}

Б)



void GraphCross(Graph\* GraphG1, Graph\* GraphG2) {

printf("Пересечение графов");

int MatrixOrderMin = GraphG1->MatrixOrder;

int MatrixOrderMax = GraphG2->MatrixOrder;

if (GraphG1->MatrixOrder > GraphG2->MatrixOrder) {

MatrixOrderMin = GraphG2->MatrixOrder;

MatrixOrderMax = GraphG1->MatrixOrder;

}

Graph\* GraphG3Cross = graph\_create(MatrixOrderMin);

{

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

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

GraphG3Cross->Matrix[i][j] = GraphG1->Matrix[i][j] && GraphG2->Matrix[i][j]; // Логическое И с наименьшей матрицей.

}

}

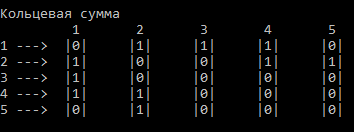
}

matrix\_print(GraphG3Cross);

graph\_free(GraphG3Cross);

}

В)



void GraphSum(Graph\* GraphG1, Graph\* GraphG2) {

printf("Кольцевая сумма");

int MatrixOrderMin = GraphG1->MatrixOrder;

int MatrixOrderMax = GraphG2->MatrixOrder;

bool is\_equal = false;

bool is\_moreG1 = false;

if (MatrixOrderMin == MatrixOrderMax) {

is\_equal = true;

}

else if (GraphG1->MatrixOrder > GraphG2->MatrixOrder) {

MatrixOrderMin = GraphG2->MatrixOrder;

MatrixOrderMax = GraphG1->MatrixOrder;

is\_moreG1 = true;

}

Graph\* GraphG3Sum = graph\_create(MatrixOrderMax);

if (is\_equal) {

for (int i = 0; i < GraphG1->MatrixOrder; i++) {

for (int j = 0; j < GraphG1->MatrixOrder; j++) {

if (GraphG1->Matrix[i][j] == 1 && GraphG2->Matrix[i][j] == 0 || GraphG1->Matrix[i][j] == 0 && GraphG2->Matrix[i][j] == 1) // проверка на присутствие в одном из графов связи между двуми вершинами.

GraphG3Sum->Matrix[i][j] = 1;

else

GraphG3Sum->Matrix[i][j] = 0;

}

}

}

else {

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

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

if (GraphG1->Matrix[i][j] == 1 && GraphG2->Matrix[i][j] == 0 || GraphG1->Matrix[i][j] == 0 && GraphG2->Matrix[i][j] == 1)

GraphG3Sum->Matrix[i][j] = 1;

else

GraphG3Sum->Matrix[i][j] = 0;

}

}

if (is\_moreG1) { // Перенос значений

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

for (int j = MatrixOrderMin; j < MatrixOrderMax; j++) {

GraphG3Sum->Matrix[i][j] = GraphG1->Matrix[i][j];

}

}

for (int i = MatrixOrderMin; i < MatrixOrderMax; i++) {

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

GraphG3Sum->Matrix[i][j] = GraphG1->Matrix[i][j];

}

}

}

else {

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

for (int j = MatrixOrderMin; j < MatrixOrderMax; j++) {

GraphG3Sum->Matrix[i][j] = GraphG2->Matrix[i][j];

}

}

for (int i = MatrixOrderMin; i < MatrixOrderMax; i++) {

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

GraphG3Sum->Matrix[i][j] = GraphG2->Matrix[i][j];

}

}

}

}

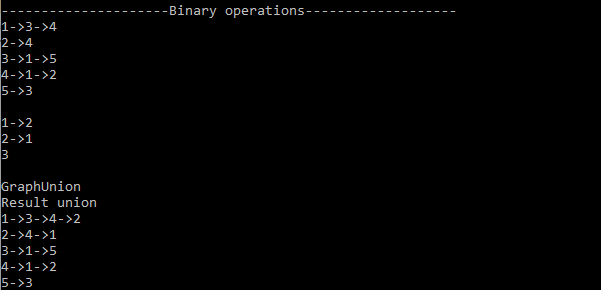
matrix\_print(GraphG3Sum);

graph\_free(GraphG3Sum);

}

**Операции на списках смежности.**

А)



oid GraphUnion(GraphList\* G1, GraphList\* G2) {

print\_graph(G1);

printf("\n");

print\_graph(G2);

printf("\nGraphUnion\n");

int num\_vertexesG1 = G1->numVertex;

int num\_vertexesG2 = G2->numVertex;

GraphList\* G3\_union;

if (num\_vertexesG1 < num\_vertexesG2) { // Проверка на разность порядков графов.

G3\_union = create\_graph(num\_vertexesG2);

for (int i = 0; i < num\_vertexesG2; i++) {

Node\* current = G2->heads[i];

while (current) {

AddLast(G3\_union, current->vertex, i);

current = current->next;

}

}

for (int i = 0; i < num\_vertexesG1; i++) {

Node\* current = G1->heads[i];

while (current) {

if (find\_node(G3\_union, current->vertex, i)) { // Находим связь, пропускаем.

current = current->next;

}

else { // Иначе, добавляем в новый temp список узел с номером вершины.

AddLast(G3\_union, current->vertex, i);

current = current->next;

}

}

}

}

else {

G3\_union = create\_graph(num\_vertexesG1);

for (int i = 0; i < num\_vertexesG1; i++) {

Node\* current = G1->heads[i];

while (current) {

AddLast(G3\_union, current->vertex, i);

current = current->next;

}

}

for (int i = 0; i < num\_vertexesG2; i++) {

Node\* current = G2->heads[i];

while (current) {

if (find\_node(G3\_union, current->vertex, i)) {

current = current->next;

}

else {

AddLast(G3\_union, current->vertex, i);

current = current->next;

}

}

}

}

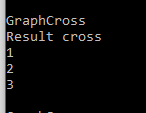
printf("Result union\n");

print\_graph(G3\_union);

graph\_delete(G3\_union);

}

Б)



void GraphCross(GraphList\* G1, GraphList\* G2) {

printf("\nGraphCross\n");

int num\_vertexesG1 = G1->numVertex;

int num\_vertexesG2 = G2->numVertex;

GraphList\* G3\_cross;

if (num\_vertexesG1 < num\_vertexesG2) { // Проверка на разность порядков графов.

G3\_cross = create\_graph(num\_vertexesG1);

for (int i = 0; i < num\_vertexesG1; i++) {

Node\* current = G1->heads[i];

while (current) {

if (find\_node(G2, current->vertex, i)) { // Если нашли связь, добавляем в новый temp список узел с номером вершины.

AddLast(G3\_cross, current->vertex, i);

current = current->next;

}

else {

current = current->next;

}

}

}

}

else {

G3\_cross = create\_graph(num\_vertexesG2);

for (int i = 0; i < num\_vertexesG2; i++) {

Node\* current = G2->heads[i];

while (current) {

if (find\_node(G1, current->vertex, i)) {

AddLast(G3\_cross, current->vertex, i);

current = current->next;

}

else {

current = current->next;

}

}

}

}

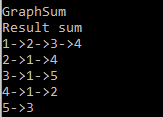
printf("Result cross\n");

print\_graph(G3\_cross);

graph\_delete(G3\_cross);

}

В)



void GraphSum(GraphList\* G1, GraphList\* G2) {

printf("\nGraphSum\n");

int num\_vertexesG1 = G1->numVertex;

int num\_vertexesG2 = G2->numVertex;

GraphList\* G3\_sum;

if (num\_vertexesG1 < num\_vertexesG2) { // Проверка на разность порядков графов

G3\_sum = create\_graph(num\_vertexesG2);

for (int i = 0; i < num\_vertexesG1; i++) {

Node\* current = G1->heads[i]; // Бежим по вершинам графа G1

while (current) {

if (find\_node(G2, current->vertex, i)) { // Если есть вершина в G2, пропускаем

current = current->next;

}

else { // Иначе добавляем узел с current вершиной.

AddLast(G3\_sum, current->vertex, i);

current = current->next;

}

}

}

for (int i = 0; i < num\_vertexesG1; i++) {

Node\* current = G2->heads[i];

while (current) {

if (find\_node(G1, current->vertex, i)) {

current = current->next;

}

else {

AddLast(G3\_sum, current->vertex, i);

current = current->next;

}

}

}

for (int i = num\_vertexesG1; i < num\_vertexesG2; i++) {

Node\* current = G2->heads[i];

while (current) {

AddLast(G3\_sum, current->vertex, i);

current = current->next;

}

}

}

else {

G3\_sum = create\_graph(num\_vertexesG1);

for (int i = 0; i < num\_vertexesG2; i++) {

Node\* current = G2->heads[i];

while (current) {

if (find\_node(G1, current->vertex, i)) {

current = current->next;

}

else {

AddLast(G3\_sum, current->vertex, i);

current = current->next;

}

}

}

for (int i = 0; i < num\_vertexesG2; i++) {

Node\* current = G1->heads[i];

while (current) {

if (find\_node(G2, current->vertex, i)) {

current = current->next;

}

else {

AddLast(G3\_sum, current->vertex, i);

current = current->next;

}

}

}

for (int i = num\_vertexesG2; i < num\_vertexesG1; i++) {

Node\* current = G1->heads[i];

while (current) {

AddLast(G3\_sum, current->vertex, i);

current = current->next;

}

}

}

printf("Result sum\n");

print\_graph(G3\_sum);

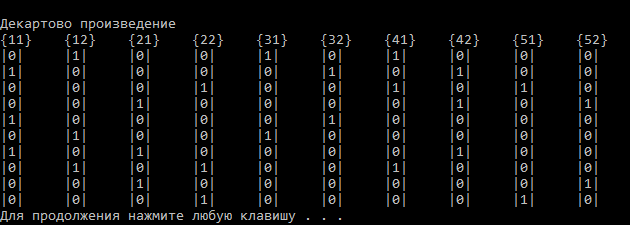
graph\_delete(G3\_sum);

}

**Задание 4.**

1. Для матричной формы представления графов выполните операцию декартова произведения графов *G = G*1X *G*2.

Результат выполнения операции выведите на экран.



void CartesianProduct(Graph\* GraphG1, Graph\* GraphG2) {

printf("Декартово произведение\n");

int matrixOrderG1 = GraphG1->MatrixOrder;

int matrixOrderG2 = GraphG2->MatrixOrder;

int vertexProduct = GraphG1->MatrixOrder \* GraphG2->MatrixOrder;

int u = 0, v = 0;

Graph\* GraphG3CartProd = graph\_create(vertexProduct);

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

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

GraphG3CartProd->Matrix[i][j] = 0;

}

}

for (int i = 0; i < matrixOrderG1; i++) { // x1 соответствует G1, x2 соответствует G2.

for (int j = 0; j < matrixOrderG2; j++, u++) {

v = 0;

for (int x1 = 0; x1 < matrixOrderG1; x1++) {

for (int x2 = 0; x2 < matrixOrderG2; x2++, v++) {

if ((x1 == i && GraphG2->Matrix[x2][j]) || (x2 == j && GraphG1->Matrix[x1][i])) {

GraphG3CartProd->Matrix[u][v] = 1;

}

}

}

}

}

for (int x1 = 0; x1 < matrixOrderG1; x1++) {

for (int x2 = 0; x2 < matrixOrderG2; x2++) {

printf("{%d%d}\t", x1 + 1, x2 + 1);

}

}

printf("\n");

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

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

printf("|%d|\t", GraphG3CartProd->Matrix[i][j]);

}

printf("\n");

}

graph\_free(GraphG3CartProd);

}

**Вывод:** в ходе выполнения лабораторных заданий изучил и реализовал унарные и бинарные операции над графами на языке Си.