Министерство науки и высшего образования Российской Федерации

Пензенский государственный университет

Кафедра «Вычислительная техника»

**ОТЧЁТ**

По лабораторной работе №6

По курсу «Логика и основы алгоритмизации в инженерных задачах»

на тему «Унарныеи бинарные операции над графами»

**Выполнили**

**студенты группы 23ВВВ2:**

Пырков Д. А.

Родионов А. А.

**Приняли:**

Митрохин М. А.

Юрова О. В.

Пенза 2024

**Цель работы:** изучить унарные и бинарные операции над графами в матричном виде и в виде списков смежности, а также преобразование матриц смежности в списки смежности.

**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.

**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.

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

а) отождествления вершин

б) стягивания ребра

в) расщепления вершины

**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.

**\*Задание 4:**

1. Для матричной формы представления графов выполните операцию декартова произведения графов G = G1 X G2.

Результат выполнения операции выведите на экран.

**Ход работы**

**Задание 1**

**1. Генерация матриц смежности**

Функция, генерирующая матрицу смежности для неориентированного графа, имеет вид:

void initialize\_graph(int size, int\_fast8\_t\*\* graph\_pointer, float edge\_probability) {

srand((unsigned int)time(NULL));

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

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

if ((float)rand() / (RAND\_MAX / 100) < edge\_probability) {

graph\_pointer[i][j] = 1;

graph\_pointer[j][i] = 1;

}

else {

graph\_pointer[i][j] = 0;

graph\_pointer[j][i] = 0;

}

}

graph\_pointer[i][i] = 0;

}

}

1. Инициализация генератора случайных чисел:

srand((unsigned int)time(NULL));

Эта строка инициализирует генератор случайных чисел текущим временем, чтобы каждый запуск программы давал разные результаты.

2. Внешний цикл:

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

Этот цикл перебирает все вершины графа.

3. Внутренний цикл:

for (int j = i + 1; j < size; j++)

Этот цикл начинается с j = i + 1, чтобы заполнить только верхнюю треугольную часть матрицы смежности. Это используется для оптимизации, так как граф неориентированный и матрица симметрична.

4. Генерация случайного числа и сравнение с вероятностью:

if ((float)rand() / (RAND\_MAX / 100) < edge\_probability)

Здесь генерируется случайное число от 0 до 100 и сравнивается с заданной вероятностью. Если случайное число меньше вероятности, создается ребро.

5. Создание ребра:

graph\_pointer[i][j] = 1;

graph\_pointer[j][i] = 1;

Если условие выполняется, устанавливается 1 в соответствующих ячейках матрицы смежности. Нужно установить значение как для [i][j], так и для [j][i], чтобы сохранить симметрию матрицы.

6. Отсутствие ребра:

else {

graph\_pointer[i][j] = 0;

graph\_pointer[j][i] = 0;

}

Если условие не выполняется, устанавливается 0, что означает отсутствие ребра.

7. Установка диагональных элементов:

graph\_pointer[i][i] = 0;

Эта строка устанавливает все диагональные элементы в 0, что означает отсутствие петель в графе.

**Результат работы программы**

На рисунке 1 приведён результат работы программы №1.1.

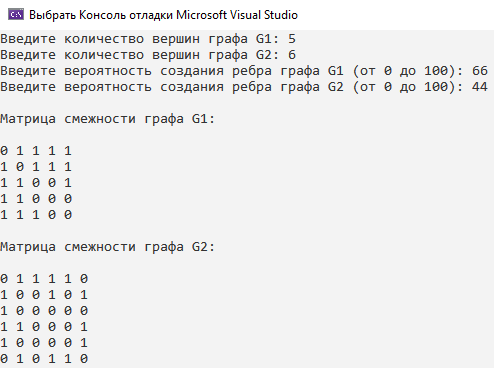


Рисунок 1 – результат работы программы №1.1

**2.** \***Преобразование матриц смежности в списки смежности**

1. Структура списка смежности:

typedef struct Node {

int vertex; // номер вершины

struct Node\* next; // указатель на следующий узел

} Node;

Список смежности представляет собой массив связных списков, где:

* Каждая вершина графа имеет свой список
* В списке хранятся номера смежных вершин
* Каждый элемент списка (Node) содержит номер смежной вершины и указатель на следующий элемент

1. Преобразование матрицы в список (функция conv\_to\_adj\_list):

Node\*\* conv\_to\_adj\_list(int size, int\_fast8\_t\*\* graph) {

// Создаем массив указателей на списки

Node\*\* adj\_list = (Node\*\*)malloc(size \* sizeof(Node\*));

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

adj\_list[i] = NULL; // Инициализируем список пустым

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

if (graph[i][j]) { // Если есть ребро (1 в матрице)

// Создаем новый узел

Node\* new\_node = (Node\*)malloc(sizeof(Node));

new\_node->vertex = j;

// Добавляем узел в начало списка

new\_node->next = adj\_list[i];

adj\_list[i] = new\_node;

}

}

}

return adj\_list;

}

Алгоритм преобразования:

1. Создается массив указателей размером size (по числу вершин)
2. Для каждой вершины i:
   * Просматривается строка i матрицы смежности
   * Если встречается 1 (есть ребро между i и j)
   * Создается новый узел с номером вершины j
   * Узел добавляется в начало списка для вершины i

Представление графов в виде списков смежности более эффективно для графов с малым количеством рёбер, так как хранит только информацию о существующих рёбрах.

**Результат работы программы**

На рисунке 2 приведён результат работы программы №1.2.

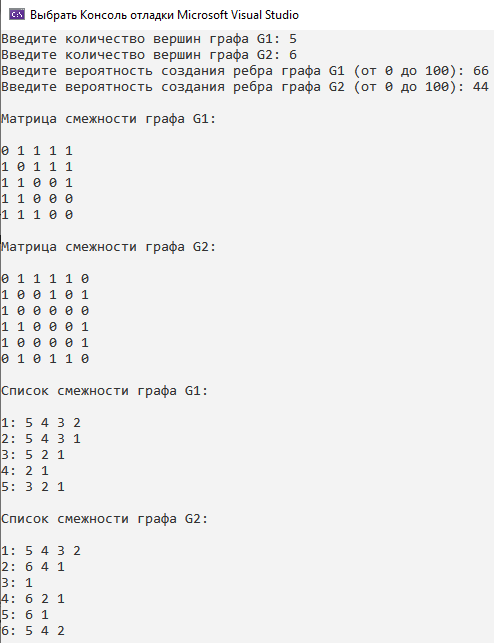


Рисунок 2 – результат работы программы №1.2

**Задание 2**

**1. Унарные операции для матричной формы представления графов**

1. Отождествление вершин (identify\_vertices):

void identify\_vertices(int\_fast8\_t\*\* graph, int\* size, int v1, int v2)

* Находит минимальный и максимальный номер из двух вершин
* Объединяет связи обеих вершин через логическое ИЛИ (если хотя бы одна из вершин была связана с какой-то третьей, то связь сохраняется)
* Удаляет вершину с большим номером, сдвигая все последующие вершины
* Корректирует матрицу смежности, убирая возможные петли
* Уменьшает размер графа на 1

1. Стягивание ребра (contract\_edge):

void contract\_edge(int\_fast8\_t\*\* graph, int\* size, int v1, int v2)

* Похоже на отождествление, но требует наличия ребра между вершинами
* Объединяет связи вершин аналогично отождествлению
* Удаляет одну из вершин и корректирует матрицу
* Убирает петлю у оставшейся вершины
* Уменьшает размер графа на 1

1. Расщепление вершины (split\_vertex):

void split\_vertex(int\_fast8\_t\*\*\* graph, int\* size, int v)

* Увеличивает размер графа на 1
* Выделяет память под новую вершину
* Копирует все связи исходной вершины для новой вершины
* Создает ребро между исходной и новой вершиной
* Устанавливает отсутствие петли у новой вершины

Основные отличия:

1. Отождествление и стягивание уменьшают количество вершин, а расщепление увеличивает;
2. Стягивание требует наличия ребра между вершинами;
3. При расщеплении создается новая вершина, связанная с исходной;
4. Отождествление и стягивание объединяют связи вершин, а расщепление копирует их.

**Результат работы программы**

На рисунке 3 приведён результат работы программы №2.1.

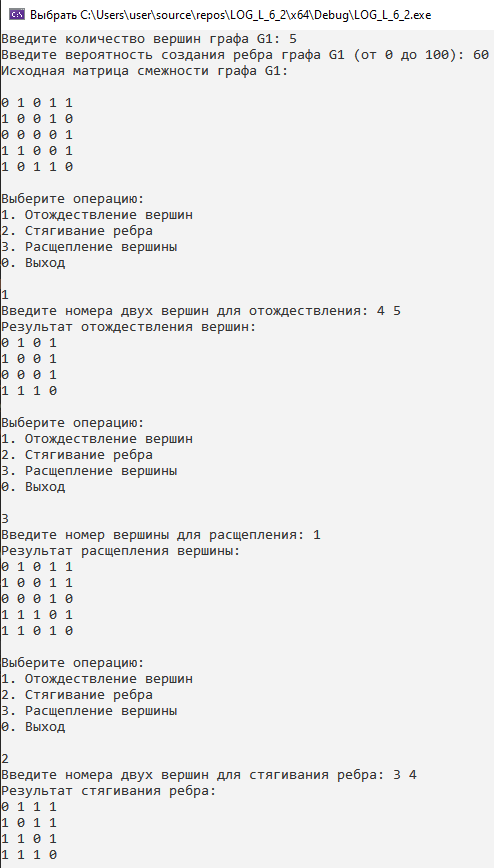


Рисунок 3 – результат работы программы №2.1

**2.** \***Унарные операции для графов в виде списков смежности**

1. Отождествление вершин (identify\_vertices):

void identify\_vertices(Node\*\* adj\_list, int\* size, int v1, int v2)

* Объединяет две вершины v1 и v2 в одну вершину v1
* Алгоритм выполнения:
  1. Копирует все рёбра от v2 к v1 (если их еще нет у v1)
  2. Удаляет все рёбра, ведущие к v2
  3. Сдвигает все вершины после v2 на одну позицию влево
  4. Обновляет индексы вершин в списках смежности
  5. Уменьшает общее количество вершин на 1

1. Стягивание ребра (contract\_edge):

void contract\_edge(Node\*\* adj\_list, int\* size, int v1, int v2)

* Похоже на отождествление, но работает только при наличии ребра между v1 и v2
* Алгоритм выполнения:
  1. Копирует все рёбра от v2 к v1 (кроме самого стягиваемого ребра)
  2. Удаляет все рёбра, ведущие к v2
  3. Освобождает память списка смежности вершины v2
  4. Сдвигает вершины после v2
  5. Обновляет индексы вершин
  6. Уменьшает размер графа

1. Расщепление вершины (split\_vertex):

void split\_vertex(Node\*\*\* adj\_list, int\* size, int v)

* Создает новую вершину и распределяет рёбра между исходной и новой
* Алгоритм выполнения:
  1. Увеличивает размер списка смежности
  2. Подсчитывает количество рёбер исходной вершины
  3. Перемещает половину рёбер новой вершине
  4. Добавляет ребро между исходной и новой вершиной
  5. Увеличивает общее количество вершин на 1

**Результат работы программы**

На рисунке 4 приведён результат работы программы №2.2.

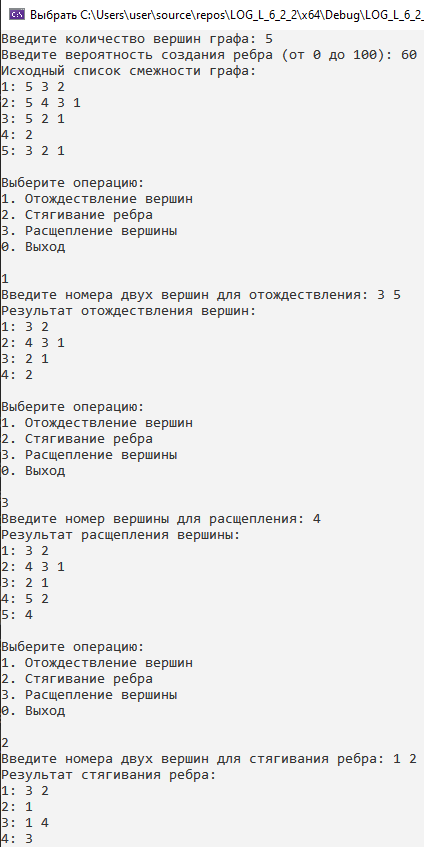


Рисунок 4 – результат работы программы №2.2

**Задание 3**

**Бинарные операции для матричной формы представления графов**

1. Объединение графов (unite\_graphs):

* Результирующий граф имеет размер, равный максимальному из размеров исходных графов
* Для каждой пары вершин (i,j) выполняется операция логического ИЛИ (||):
  + Если ребро существует хотя бы в одном из графов, оно будет в результате
  + Если вершина есть только в одном графе, копируются её рёбра
  + Для несуществующих вершин устанавливается 0

1. Пересечение графов (intersect\_graphs):

* Результирующий граф имеет размер, равный минимальному из размеров исходных графов
* Для каждой пары вершин (i,j) выполняется операция логического И (&&):
  + Ребро появляется в результате только если оно существует в обоих графах
  + Берутся только общие для обоих графов вершины

1. Кольцевая сумма графов (ring\_sum\_graphs):

* Результирующий граф имеет размер, равный максимальному из размеров исходных графов
* Для каждой пары вершин (i,j) выполняется операция исключающего ИЛИ (XOR ^):
  + Ребро появляется в результате, если оно есть только в одном из графов
  + Если ребро есть в обоих графах, оно исчезает
  + Если вершина есть только в одном графе, копируются её рёбра

**Результат работы программы**

На рисунке 5 приведён результат работы программы №3.

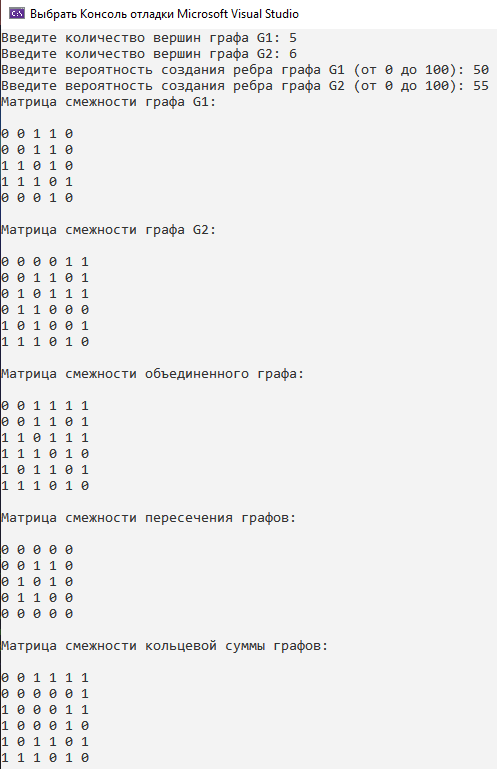


Рисунок 5 – результат работы программы №3

**\*Задание 4**

**Декартово произведение графов**

1. Создание результирующего графа:

int new\_size = size1 \* size2;

int\_fast8\_t\*\* result = create\_graph(new\_size);

* Размер нового графа равен произведению размеров исходных графов
* Создается новая матрица смежности размером new\_size × new\_size

1. Правило соединения вершин:

if ((i1 == j1 && G2[i2][j2]) || (i2 == j2 && G1[i1][j1]))

Две вершины (i1,i2) и (j1,j2) в результирующем графе соединены ребром, если выполняется одно из условий:

* i1 = j1 и вершины i2 и j2 смежны в G2
* i2 = j2 и вершины i1 и j1 смежны в G1

1. Индексация вершин:

result[i1 \* size2 + i2][j1 \* size2 + j2] = 1;

* Каждая вершина результирующего графа имеет координаты (i1,i2)
* Преобразование этих координат в линейный индекс происходит по формуле: i1 \* size2 + i2

1. Процесс построения:

* Используются четыре вложенных цикла для перебора всех возможных пар вершин
* Для каждой пары вершин проверяется условие смежности
* Если условие выполняется, в матрице смежности результирующего графа ставится 1

**Результат работы программы**

На рисунке 6 приведён результат работы программы №4.

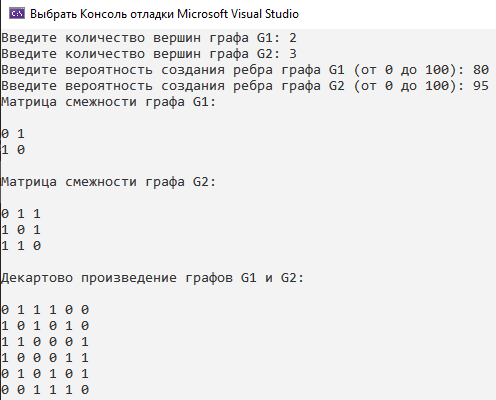


Рисунок 6 – результат работы программы №4

**Выводы**

Были изучены унарные и бинарные операции над графами в матричном виде и в виде списков смежности, а также преобразование матриц смежности в списки смежности.