ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(национальный исследовательский университет)»

Факультет №3 «Системы управления, информатика и электроэнергетика»

Кафедра № 304 «Вычислительные машины, системы и сети»

Отчёт по лабораторной работе

по дисциплине: «Структуры и алгоритмы обработки данных»

на тему: «Поиск кратчайших путей в графе. Построение остовного дерева графа.»

Выполнили:

студенты группы М30-210Б-23
Фомин В. А.

Миронов А. Д.

Проверила:

Дмитриева Е. А.

Москва 2024 г.

Задание (вариант 7)

Лабораторная работа.

Поиск кратчайших путей в графе. Построение остовного дерева графа.

Для взвешенного ориентированного графа, состоящего как минимум из 10 вершин, реализовать по вариантам:

- алгоритм поиска кратчайшего пути;
- сделав тот же самый граф неориентированным, построить его остовное дерево минимальной стоимости.

Граф для первого пункта задания придумывается самостоятельно (не нужно связывать каждую вершину со всеми остальными, не делайте граф полным).

Матрицу смежности (значения весов каждого ребра) лучше определить в начале программы, не вводить значения с клавиатуры или из файла.

Должны быть представлены промежуточные результаты.

По каждому кратчайшему пути указать предшествующие вершины.

Граф и полученное остовное дерево должны быть изображены на рисунках в отчете.

Варианты заданий

№ вар.	Алгоритм, реализуемый в п.1 задания	Алгоритм, п.2 задания
1	Дейкстры	Крускала
2	Беллмана-Форда	Крускала
3	Флойда-Уоршелла	Прима
4	Дейкстры	Прима
5	Беллмана-Форда	Прима
6	Флойда-Уоршелла	Крускала

Код программы

Крускал

```
#include <iostream>

using namespace std;

struct Edge {
   int u, v;    // Вершины ребра
   int weight;
};

void MakeSet(int parent[], int rank[], int V);
int FindSet(int parent[], int x);
void Union(int parent[], int rank[], int x, int y);
void Kruskal(int V, int E, Edge edges[], Edge result[]);

int main() {
   int V = 12;
   int E = 17;
   int weightSum = 0;

   Edge edges[] = {
           {1, 5, 4},
           {5, 7, 9},
           {3, 7, 12},
           {3, 10, 15},
           {10, 8, 7},
           {9, 8, 6},
           {2, 1, 14},
           {9, 1, 10},
           {2, 9, 3},
           {9, 6, 8},
           {6, 3, 2},
           {11, 3, 13},
           {4, 11, 5},
           {12, 11, 8},
           {12, 4, 11},
           {4, 2, 17},
           {12, 2, 19}
   };

   Edge* result = new Edge[V - 1]; // Динамическое выделение памяти

   Kruskal(V, E, edges, result);

   // Количество рёбер в MST равно V - 1
   int edgeCount = V - 1;
```

```
    for (int i = 0; i < edgeCount; ++i) {
        cout << result[i].u << "->" << result[i].v << "     Weight: " <<
result[i].weight  << endl;
        weightSum += result[i].weight;
    }
    cout << endl;
    cout << "Minimum frame weight: " << weightSum << endl;

    delete[] result; // Освобождение памяти
    return 0;
}

void MakeSet(int parent[], int rank[], int V) {
    for (int i = 0; i < V; i++) {
        parent[i] = i; // Каждая вершина является своим родителем
        rank[i] = 0;
    }
}

int FindSet(int parent[], int x) {
    if (parent[x] != x) {
        parent[x] = FindSet(parent, parent[x]);
    }
    return parent[x];
}

void Union(int parent[], int rank[], int x, int y) {
    int rootX = FindSet(parent, x);
    int rootY = FindSet(parent, y);

    if (rootX != rootY) {
        if (rank[rootX] < rank[rootY]) {
            parent[rootX] = rootY;
        } else if (rank[rootX] > rank[rootY]) {
            parent[rootY] = rootX;
        } else {
            parent[rootY] = rootX;
            rank[rootX]++;
        }
    }
}

void Kruskal(int V, int E, Edge edges[], Edge result[]) {
    // Массивы для Union-Find
    int parent[V];
    int rank[V];

    // Шаг 1: Инициализация множества вершин
    MakeSet(parent, rank, V);

    // Шаг 2: Сортировка рёбер по возрастанию веса - пузырьковая
    for (int i = 0; i < E - 1; ++i) {
        for (int j = 0; j < E - i - 1; ++j) {
```

```
            if (edges[j].weight > edges[j + 1].weight) {
                Edge temp = edges[j];
                edges[j] = edges[j + 1];
                edges[j + 1] = temp;
            }
        }
    }

    // Шаг 3: Построение остовного дерева
    int edgeCount = 0;                // Количество рёбер в результате
    int vertices[V];         // Хранилище для вершин
    int vertexCount = 0;              // Текущее количество уникальных вершин

    for (int i = 0; i < E; ++i) {
        int u = edges[i].u - 1; // Индексы с 0
        int v = edges[i].v - 1; // Индексы с 0

        // Проверка, принадлежат ли вершины разным компонентам
        if (FindSet(parent, u) != FindSet(parent, v)) {
            result[edgeCount++] = edges[i]; // Добавляем ребро к результату
            Union(parent, rank, u, v);      // Объединяем множество

            // Добавление вершин в массив (если уникальные)
            bool foundU = false, foundV = false;
            for (int j = 0; j < vertexCount; ++j) {
                if (vertices[j] == edges[i].u) foundU = true;
                if (vertices[j] == edges[i].v) foundV = true;
            }
            if (!foundU) vertices[vertexCount++] = edges[i].u;
            if (!foundV) vertices[vertexCount++] = edges[i].v;

            // Вывод текущего состояния вершин и добавленного ребра
            cout << "Edge added: " << edges[i].u << "->" << edges[i].v << "\t";
            cout << "Current vertices in MST: ";
            for (int j = 0; j < vertexCount; ++j) {
                cout << vertices[j] << " ";
            }
            cout << endl << endl;
        }

        // Если найдено достаточно рёбер для MST (V - 1), можно завершить
        if (edgeCount == V - 1) {
            break;
        }
    }
}
```

Дейкстра

```
#include <iostream>
#include <limits> // Подключение для использования INT_MAX (максимального значения целого числа)

using namespace std;

// Структура графа
struct Graph{
   int V; // Количество вершин
   int E; // Количество рёбер
   int** Adj; // Матрица смежности для представления графа
};

// Структура узла приоритетной очереди
struct Node {
   int vertex; // Вершина
   int distance; // Расстояние (приоритет)
   Node* next; // Указатель на следующий узел
};

// Структура приоритетной очереди
struct PriorityQueue{
   Node* head; // Указатель на голову очереди
};

// Создание графа с использованием матрицы смежности
Graph* adjMatrixOfGraph(){
   int i, u, v, w;
   Graph* G = new Graph;
   if(!G){
       cout << "Memory Error." << endl;
       return nullptr; // Проверка на выделение памяти
   }

   cout << "Enter the number of vertices: ";
   cin >> G->V;
   cout << "Enter the number of edges: ";
   cin >> G->E;

   // Создание и инициализация матрицы смежности
   G->Adj = new int*[G->V];
   for(u = 0; u < G->V; u++){
       G->Adj[u] = new int[G->V];
       for(v = 0; v < G->V; v++){
           G->Adj[u][v] = 0; // Инициализация всех значений нулями
       }
   }

   // Ввод рёбер графа
   cout << "Enter the edges (format: u v weight):\n";
   for(i = 0; i < G->E; i++){
       cout << "Edge " << i + 1 << ": ";
       cin >> u >> v >> w;
```

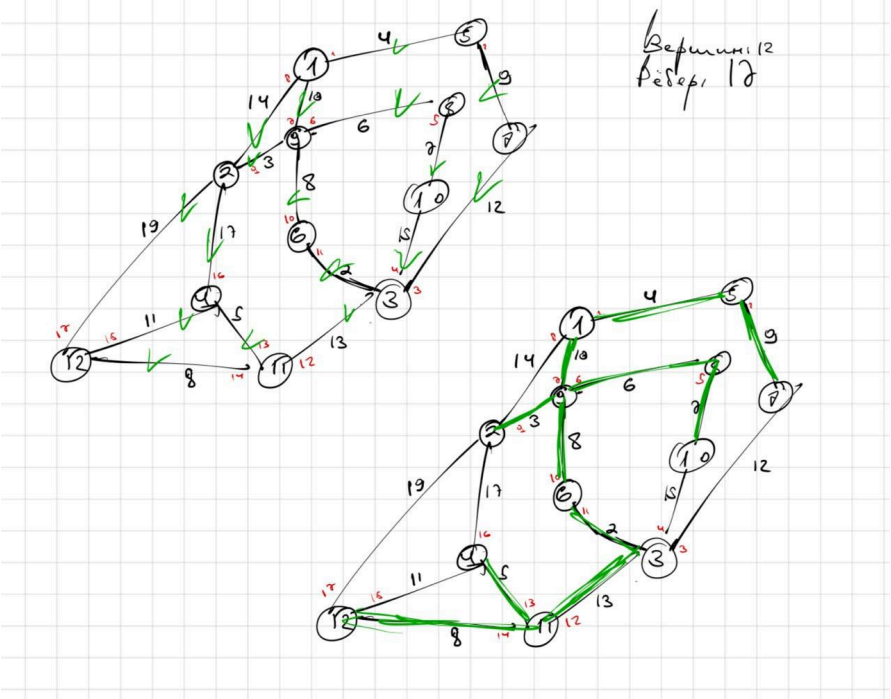
```
        u--; v--; // Приведение вершин к нумерации с нуля
        G->Adj[u][v] = w; // Задание веса ребра
    }

    return G;
}

// Вывод матрицы смежности графа
void printG(Graph* G){
    cout << "Adjacency Matrix of the Graph:\n";
    for(int u = 0; u < G->V; ++u){
        for(int v = 0; v < G->V; ++v){
            cout << G->Adj[u][v] << "\t";
        }
        cout << endl;
    }
}

// Освобождение памяти графа
void freeG(Graph* G){
    for(int i = 0; i < G->V; i++){
        delete[] G->Adj[i];
    }
    delete[] G->Adj;
    delete G;
}

// Инициализация пустой очереди
void initQueue(PriorityQueue& pq){
    pq.head = nullptr;
}

// Проверка на пустоту очереди
bool isEmptyQ(PriorityQueue& pq){
    return pq.head == nullptr;
}

// Вставка элемента в очередь с учетом приоритета
void push(PriorityQueue& pq, int vertex, int distance){
    Node* newNode = new Node;
    newNode->vertex = vertex;
    newNode->distance = distance;
    newNode->next = nullptr;

    // Вставка элемента с упорядочением по расстоянию
    if(isEmptyQ(pq) || pq.head->distance > distance){
        newNode->next = pq.head;
        pq.head = newNode;
    } else {
        Node* current = pq.head;
        while(current->next && current->next->distance <= distance){
            current = current->next;
        }
        newNode->next = current->next;
        current->next = newNode;
    }
```

```
}

// Удаление элемента с наименьшим приоритетом из очереди
void pop(PriorityQueue& pq){
   if(isEmptyQ(pq)) return;
   Node* temp = pq.head;
   pq.head = pq.head->next;
   delete temp;
}

// Получение элемента с наименьшим приоритетом
Node* top(PriorityQueue& pq){
   return pq.head;
}

// Освобождение памяти, выделенной под очередь
void freeQ(PriorityQueue& pq){
   while(!isEmptyQ(pq)){
       pop(pq);
   }
}

// Функция для отображения текущего состояния массивов расстояний и
предшественников
void printIntermediateResults(int dist[], int predecessor[], int V){
   cout << "Vertex\tDistance\tPredecessor" << endl;
   for(int i = 0; i < V; i++){
       cout << i + 1 << "\t" << (dist[i] == INT_MAX ? -1 : dist[i]) << "\t\t";
       if(predecessor[i] != -1){
           cout << predecessor[i] + 1;
       } else{
           cout << "None";
       }
       cout << endl;
   }
   cout << endl;
}

// Вывод пути от исходной вершинны до данной
void printPath(int predecessor[], int vertex){
   if(predecessor[vertex] == -1){
       cout << vertex + 1;
       return;
   }
   printPath(predecessor, predecessor[vertex]);
   cout << " -> " << vertex + 1;
}

// Реализация алгоритма Дейкстры для поиска кратчайших путей
void dijkstra(Graph* G, int source){
   int V = G->V;
   int dist[V]; // Массив для хранения кратчайших расстояний
   int predecessor[V]; // Массив для хранения предшествующих вершин
   bool visited[V]; // Массив для отметки посещённых вершин

   // Инициализация массивов
```

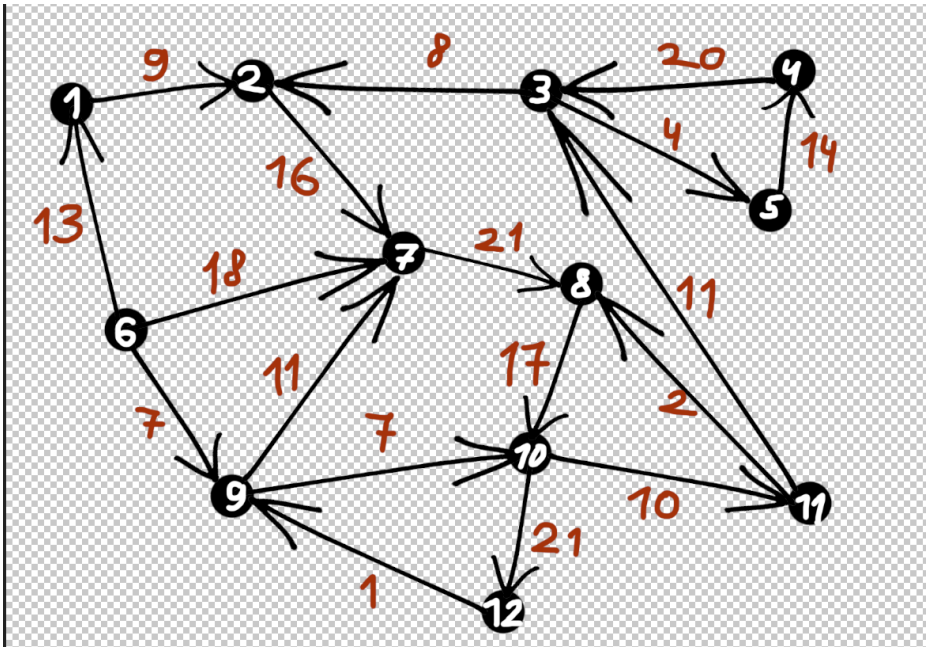
```
    for(int i = 0; i < V; i++){
        dist[i] = INT_MAX; // Все расстояния изначально бесконечны
        predecessor[i] = -1; // Все предшественники изначально отсутствуют
        visited[i] = false; // Все вершины изначально не посещены
    }
    dist[source] = 0;

    PriorityQueue pq;
    initQueue(pq);
    push(pq, source, 0);

    while(!isEmptyQ(pq)){
        Node* minNode = top(pq); // Извлечение вершины с минимальным расстоянием
        int u = minNode->vertex;
        pop(pq);

        if(visited[u]) continue; // Пропуск уже посещенных вершин
        visited[u] = true;

        // Обновление расстояний до соседних вершин
        for(int v = 0; v < V; v++){
            if(G->Adj[u][v] != 0){
                int weight = G->Adj[u][v];
                if(!visited[v] && dist[u] != INT_MAX && dist[u] + weight < dist[v]){
                    dist[v] = dist[u] + weight;
                    predecessor[v] = u; // Установка предшественника
                    push(pq, v, dist[v]); // Добавление вершины в очередь
                }
            }
        }

        // Печать промежуточных результатов
        printIntermediateResults(dist, predecessor, V);
    }

    // Вывод кратчайших путей и расстояний
    cout << "Vertex\tDistance from source " << source + 1 << "\tPath" << endl;
    for(int i = 0; i < V; i++){
        cout << i + 1 << "\t" << (dist[i] == INT_MAX ? -1 : dist[i]) << "\t\t";

        // Вывод пути
        if(dist[i] == INT_MAX){
            cout << "No path";
        } else{
            printPath(predecessor, i);
        }
        cout << endl;
    }

    freeQ(pq); // Освобождение памяти очереди
}

int main(){
    Graph* G = adjMatrixOfGraph(); // Создание графа
    if(G){
        printG(G); // Вывод графа
```

```
        int source;
        cout << "Enter the source vertex: ";
        cin >> source;
        source--; // Приведение к нумерации с нуля
        dijkstra(G, source); // Запуск алгоритма Дейкстры
        freeG(G); // Освобождение памяти графа
    }

    return 0;
}
```

Остовное дерево (Крускал)



Граф (Дейкстра)



Результаты работы

Крускал

```
Edge added: 6->3        Current vertices in MST: 6 3

Edge added: 2->9        Current vertices in MST: 6 3 2 9

Edge added: 1->5        Current vertices in MST: 6 3 2 9 1 5

Edge added: 4->11       Current vertices in MST: 6 3 2 9 1 5 4 11

Edge added: 9->8        Current vertices in MST: 6 3 2 9 1 5 4 11 8

Edge added: 10->8       Current vertices in MST: 6 3 2 9 1 5 4 11 8 10

Edge added: 9->6        Current vertices in MST: 6 3 2 9 1 5 4 11 8 10

Edge added: 12->11      Current vertices in MST: 6 3 2 9 1 5 4 11 8 10 12

Edge added: 5->7        Current vertices in MST: 6 3 2 9 1 5 4 11 8 10 12 7

Edge added: 9->1        Current vertices in MST: 6 3 2 9 1 5 4 11 8 10 12 7

Edge added: 11->3       Current vertices in MST: 6 3 2 9 1 5 4 11 8 10 12 7
```

```
6->3     Weight: 2
2->9     Weight: 3
1->5     Weight: 4
4->11     Weight: 5
9->8     Weight: 6
10->8     Weight: 7
9->6     Weight: 8
12->11     Weight: 8
5->7     Weight: 9
9->1     Weight: 10
11->3     Weight: 13
```

Дейкстра

```
Vertex  Distance from source 1  Path
1       0               1
2       9               1 -> 2
3       84              1 -> 2 -> 7 -> 8 -> 10 -> 11 -> 3
4       102             1 -> 2 -> 7 -> 8 -> 10 -> 11 -> 3 -> 5 -> 4
5       88              1 -> 2 -> 7 -> 8 -> 10 -> 11 -> 3 -> 5
6       92              1 -> 2 -> 7 -> 8 -> 10 -> 12 -> 9 -> 6
7       25              1 -> 2 -> 7
8       46              1 -> 2 -> 7 -> 8
9       85              1 -> 2 -> 7 -> 8 -> 10 -> 12 -> 9
10      63              1 -> 2 -> 7 -> 8 -> 10
11      73              1 -> 2 -> 7 -> 8 -> 10 -> 11
12      84              1 -> 2 -> 7 -> 8 -> 10 -> 12
```

Вывод

В ходе работы были изучены алгоритмы работы с ориентированными и неориентированными взвешенными графами. Разработаны реализации на языке C++ алгоритм Дейкстры и алгоритм Крускала. Также были изучены алгоритм Прима, Беллмана-Форда, Флойда-Уоршелла