**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра Информационных Систем**

**КУРСОВАЯ РАБОТА**

**по дисциплине «Алгоритмы и Структуры Данных»**

Тема: Алгоритм поиска минимального остова на основе алгоритма Крускала

|  |  |  |
| --- | --- | --- |
| Студент гр. 2372 |  | Науменко А.В. |
| Преподаватель |  | Пелевин М.С. |

Санкт-Петербург

2023

**Аннотация**

Курсовой проект заключается в разработке и реализации алгоритма поиска минимального остовного дерева на основе Крускала. В ходе работы были рассмотрены и реализованы такие методы, как поиск в глубину и ширину, сортировка ребер графа по весу, построение системы независимых множеств. В результате была написана и отлажена программа, принимающая на вход файл, содержащий матрицу смежности связного неориентированного взвешенного графа и возвращающая файл с результатом работы алгоритма Крускала – списком ребер минимального остова, отсортированного в алфавитном порядке имён вершин, связанных каждым ребром.

**содержание**

|  |  |  |
| --- | --- | --- |
|  | Введение | 4 |
| 1. | Графы | 5 |
| 1.1. | Хранение графов | 5 |
| 1.2. | Основные алгоритмы | 6 |
| 1.3. | Система непересекающихся множеств | 6 |
| 2. | Алгоритм поиска минимального остовного дерева | 7 |
| 2.1. | Формальное описание алгоритма | 7 |
| 2.2. | Контрольный пример с описанием ожидаемого и полученного результатов | 7 |
|  | Заключение | 10 |
|  | Приложение А. Исходный код консольного приложения | 11 |
|  | Приложение Б. Исходный код структур Graph, Vertex, Edge | 13 |
|  | Приложение В. Исходный код класса Disjoint Set Union | 16 |

**введение**

Алгоритм Крускала решает задачу о нахождении минимального остовного дерева, которая неформально может встречаться в подобной постановке: есть несколько городов, которые необходимо соединить дорогами так, чтобы можно было попасть в любой город из любого, суммарная длина дорог должна быть минимально возможной.

Города в данной задаче можно рассматривать как вершины графа, а дороги – как ребра графа. Рассматривая её таким образом, можно применить вышеупомянутый алгоритм.

Таким образом, цель этого курсового проекта – реализовать алгоритм Крускала поиска минимального остова. В частности, необходимо исследовать и реализовать такие структуры данных, как граф и система непересекающихся множеств.

Задача: реализовать программу, которая принимает на вход файл, содержащий матрицу смежности в виде:

|  |
| --- |
| **input.txt** |
| A B C  0 3 1  3 0 2  1 2 0 |

где первая строка содержит названия вершин – любой строчный или заглавный символ из английского алфавита. Последующие строки содержат матрицу смежности, содержащие неотрицательные числа: 0 означает отсутствие ребра, отличное от нуля число – вес ребра, связывающего две вершины.

Программа должна считать данные из файла и, применив алгоритм Крускала, создать минимальный остов.

Результатом работы программы должен быть файл, содержащий список смежности, отсортированный в алфавитном порядке, должен быть указан суммарный вес ребер:

|  |
| --- |
| **output.txt** |
| A C  B C  3 |

**графы**

1. **Хранение графов**

Граф – это абстрактный математический объект, представляющий собой множество вершин и ребер. В данной работе рассматривается связный неориентированный взвешенный граф без циклов. То есть граф, в котором у каждой вершины есть не менее чем один сосед, у связей нет направления, у каждого ребра определён свой вес и ни одна вершина не может быть связана сама с собой.

Для хранения графов, в зависимости от решаемой задачи, могут использоваться различные матричные или последовательные структуры. В данной работе будут рассмотрены и использованы три структуры для хранения графов: матрица смежности, список смежности и матрица инцидентности.

Матрица смежности представляет из себя квадратную матрицу (квадратную без учёта заголовков). Значение каждого её элемента определяет наличие или отсутствие ребра между вершинами и соответственно. В случае наличия ребра, значение является весом связующего ребра.

Список смежности – это списочная структура, в которой каждой вершине графа сопоставляется список вершин, смежных с данной. Является особенно эффективной для хранения разреженных графов, где количество ребер много меньше максимально возможного , где – количество вершин графа, так как в этом случае список будет занимать меньше места, чем матрица.

Матрица инцидентности представляет из себя матрицу. Если вершина и ребро инцидентны друг другу, то в соответствующее поле записывается ненулевое значение. В случае неориентированных графов значение будет положительным, но для ориентированных положительные числа значат, что ребро «выходит» из вершины, а отрицательные – ребро «входит» в вершину.

Разнообразие способов хранения графов объясняется различными задачами, которые решают графы. Так, например, в данной работе матрица инцидентности применена только ради примера, поскольку граф не ориентирован, то есть эта матрица «превращается» в обычную матрицу смежности.

1. **Основные алгоритмы**

Рассмотрим методы графов, которые были реализованы в данной работе, а именно: сортировка и обходы в ширину и в глубину.

Обход в ширину – это алгоритм, используемый для поиска или получения всех узлов графа. Обход начинается с условного корня графа (в текущей реализации – первого добавленного узла) и направляется к каждой смежной вершиной. Для каждой смежной вершины алгоритм повторяется до тех пор, пока все вершины не будут посещены. Этот алгоритм использует очередь для хранения посещенных узлов.

Обход в глубину же заключается в том, что алгоритм от корневой вершины двигается по определенному пути до тех пора, пока не достигнет конца пути или искомой вершины. В случае, когда конец пути не является пунктом назначения (искомой вершиной), мы возвращаемся назад и идём по иному маршруту. Для отслеживания посещенных вершин в данном обходе используется стек.

В данной работе под сортировкой понимается сортировка ребер по весу, то есть получение отсортированного по возрастанию веса ребра списка пар смежных вершин и весов ребер между ними.

1. **Система непересекающихся множеств**

Одной из структур, необходимых для работы алгоритма Крускала, является система непересекающихся множеств.

Система непересекающихся множеств (СНМ) – это структура данных, управляющая несколькими независимыми подмножествами. Она содержит две основные операции: объединение двух подмножеств и поиск представителя подмножества, которому принадлежит некоторый элемент.

**Алгоритм поиска минимального остовного дерева**

1. **Формальное описание алгоритма**

Имеется неориентированный взвешенный граф. Задача алгоритма – найти минимальное остовное дерево – граф, содержащий все вершины исходного графа, соединенные таким образом, чтобы сумма ребер была минимально возможной (вес каждого ребра остаётся неизменным).

Для этого на вход подаётся пустой граф, который будет достраиваться до минимального остова. Каждая вершина исходного графа считается несвязной и является независимым множеством:

1. Вначале производится сортировка ребер исходного графа по не убыванию их весов;
2. Берётся поочередно каждое ребро из полученного списка и добавляется в граф только в том случае, если данное ребро соединяет создаваемый граф с новым независимым множеством. Для этого используется структура СНМ, описанная выше;
3. Когда множество вершин созданного графа совпадёт с множеством вершин исходного граф, алгоритм закончит свою работу.
4. **Контрольный пример с описанием ожидаемых и полученных результатов**

Для расчёта был взят следующий граф (рис. 1):

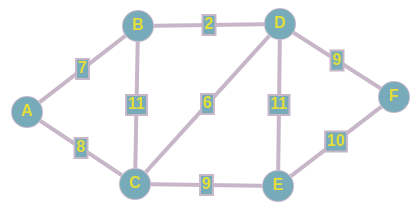


Рисунок . Исходный граф

Как было сказано выше, в первую очередь нам нужно отсортировать ребра графа, после сортировки получим следующий список:

1. B <--> D w = 2
2. C <--> D w = 6
3. A <--> B w = 7
4. A <--> C w = 8
5. C <--> E w = 9
6. D <--> F w = 9
7. E <--> F w = 10
8. B <--> C w = 11
9. D <--> E w = 11,

Где w – вес ребра между вершинами слева и справа от знака <-->. В соответствии с алгоритмом, после сортировки мы добавляем ребра в граф поочередно (рис. 2):

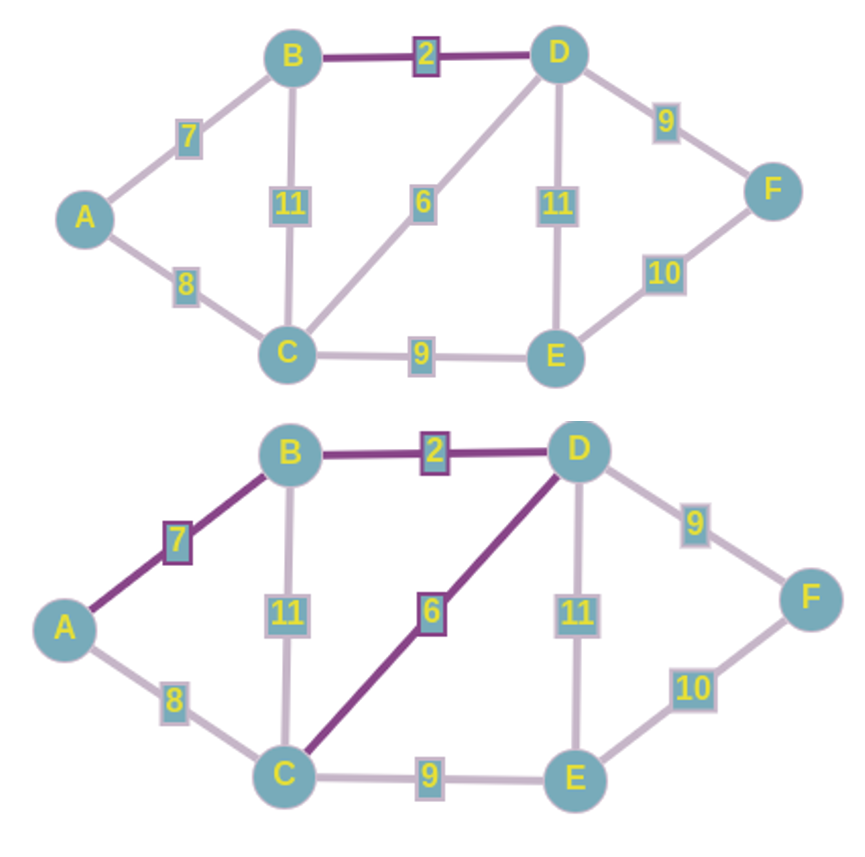


Рисунок . Подграф после добавления первых трех ребер

При попытке добавить ребро №4, соединяющее ребра A и C мы обнаруживаем, что образуется цикл, или, говоря в терминах алгоритма, не происходит объединения независимых подмножеств, ведь A и C принадлежат одному и тому же множеству. В результате мы должны получить минимальное остовное дерево с суммарным весом ребер равным 33 для исходного графа (рис. 3).

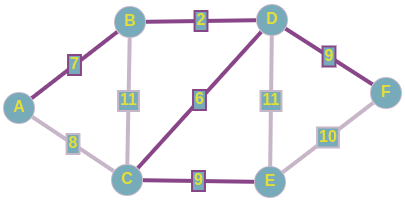


Рисунок . Минимальный остов

Получив расчётный результат, следует проверить написанную в ходе работу программу, исходный код которой представлен в приложениях А – консольное приложение, Б – классы Граф, Вершина и Ребро и В – класс Система Непересекающихся Множеств. На вход программа получает матрицу смежности исходного графа (рис. 4).



Рисунок . Файл с исходным графом

Результатом работы программы является созданный файл “*output.txt*”, содержащий отсортированные в алфавитном порядке пары смежных вершин и суммарный вес ребер полученного минимального остовного дерева (рис. 5).

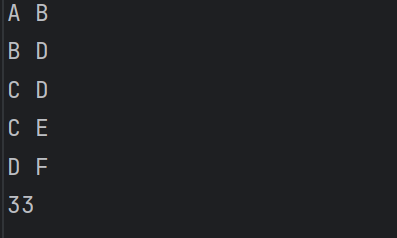


Рисунок . Файл с минимальным остовом

**ЗАКЛЮЧЕНИЕ**

В ходе работы были рассмотрены основные аспекты работы с графами, различные структуры представления и хранения графов. Изучен и реализован алгоритм поиска минимального остова на основе алгоритма Крускала и необходимые для него алгоритмы и структуры данных.

Приложение А

консольное приложение

class TApplication {  
public:  
 static int execute(); *// Execute the main thread*private:  
 static bool menu(char&); *// Get user's choice* static void help(); *// Print the available commands*};

*/// Update <names> & <values> by data from <path>. Return 1 on success  
/\*\* Return values  
 \* 1 Success  
 \* -1 File Not Found  
 \* -2 Invalid Input  
 \* -3 Invalid Size  
 \* -4 Non-Zero Diagonals  
 \*/*void parseFile(const std::string &path, std::vector<char> &names, std::vector<std::vector<int>> &values) {  
 std::ifstream iFile(path);  
 if (!iFile.is\_open()) throw std::runtime\_error("FileNotFound: Couldn't open '" + path + "' file");  
 std::string line;  
  
 *// Get names* std::getline(iFile, line);  
 std::string name;  
 for (auto &c : line) {  
 if (c == ' ') {  
 if (!name.empty()) names.push\_back(name[0]);  
 name.clear();  
 } else {  
 name += c;  
 if (name.length() > 1) throw std::runtime\_error("InvalidName: Invalid vertex name");  
 }  
 }  
 names.push\_back(name[0]);  
  
 *// Get values* while (std::getline(iFile, line)) {  
 std::vector<int> row;  
 std::string number;  
 for (auto &c : line) {  
 if (c == ' ') {  
 if (!number.empty()) row.push\_back(std::stoi(number));  
 number.clear();  
 } else if (isdigit(c)) {  
 number += c;  
 } else {  
 throw std::runtime\_error("InvalidValue: Invalid character '" + std::string(1, c) + "' in file");  
 }  
 }  
 row.push\_back(std::stoi(number)); *// Last row element* if (!values.empty() && row.size() != values[0].size())  
 throw std::runtime\_error("InvalidSize: Invalid row size");  
 values.push\_back(row);  
 }  
 if (values.size() != values[0].size()) throw std::runtime\_error("InvalidSize: Invalid matrix size");  
 for (int i = 0; i < values.size(); ++i)  
 if (values[i][i] != 0) throw std::runtime\_error("LoopError: Non-zero diagonals");  
  
 iFile.close();  
}  
  
  
*/// Execute the main thread*int TApplication::execute() {  
 *// Read input file and create Graph object* std::string path = std::getenv("INPUT\_FILE\_PATH");  
 std::vector<char> names;  
 std::vector<std::vector<int>> values;  
 parseFile(path, names, values);  
 Graph graph(names, values);  
  
 *// Searches* std::vector<Vertex\*> dfs = graph.depthFirstSearch();  
 std::vector<Vertex\*> bfs = graph.breadthFirstSearch();  
 std::cout << "DFS: ";  
 for (auto \*el : dfs) std::cout << \*el << ' ';  
 std::cout << "\nBFS: ";  
 for (auto \*el : bfs) std::cout << \*el << ' ';  
 std::cout << '\n';  
  
 *// Sort by weight* std::vector<Edge> sorted = graph.getSortedByWeight();  
 std::cout << "Edges sorted by weight:\n";  
 for (auto &edge : sorted)  
 std::cout << " " << edge << '\n';  
  
 *// Minimum Spanning Tree using Kruskal's algorithm* std::vector<Edge> mst = graph.getMST();  
  
 *// Sort by vertex name* std::sort(  
 mst.begin(), mst.end(),  
 [](Edge& a, Edge& b) {  
 std::string aNames = std::string(1, a.getU()->getName()) + std::string(1, a.getV()->getName());  
 std::string bNames = std::string(1, b.getU()->getName()) + std::string(1, b.getV()->getName());  
 return aNames < bNames;  
 }  
 );  
  
 *// Print MST* std::cout << "Minimum Spanning Tree:\n";  
 int weight = 0;  
 for (auto &edge : mst) {  
 std::cout << " " << edge << '\n';  
 weight += edge.getWeight();  
 }  
 std::cout << " Full weight = " << weight << '\n';  
  
 *// Write answer to the output file* std::ofstream oFile(std::getenv("OUTPUT\_FILE\_PATH"));  
 if (!oFile.is\_open()) throw std::runtime\_error("FileNotFound: Couldn't open 'output.txt' file");  
 for (auto &edge: mst) oFile << \*(edge.getU()) << ' ' << \*(edge.getV()) << '\n';  
 oFile << weight;  
 oFile.close();  
 std::cout << "Answer has been written to the `output.txt` file\n";  
  
 return 0;  
}

int main () {  
 return TApplication::execute();  
}

Приложение Б

СТруктуры ГРАФ, ВЕРШИНА, РЕБРО

*/// Node of the graph. Contains the <name> and map of adjacent <edges>*class Vertex {  
 friend class Graph;  
  
private:  
 char \_name;  
 std::map<Vertex\*, int> edges;  
  
public:  
 explicit Vertex(char);  
 void setEdge(Vertex\*, int);  
 [[nodiscard]] char getName() const;  
 friend std::ostream& operator<< (std::ostream&, const Vertex&);  
};  
  
  
*/// Edge between two vertices. Represented as <first> - <weight> - <second>*class Edge {  
 friend class Graph;  
  
private:  
 int weight;  
 Vertex \*u;  
 Vertex \*v;  
  
public:  
 Edge(int, Vertex\*, Vertex\*);  
 [[nodiscard]] int getWeight() const;  
 Vertex \* getU();  
 Vertex \* getV();  
 friend std::ostream& operator<< (std::ostream&, Edge&);  
 bool operator< (const Edge&) const;  
};  
  
  
*/// Weighted undirected graph. Contains the <amount> and <vertices>*class Graph {  
private:  
 std::vector<Vertex\*> vertices; *// Store each vertex*public:  
 *// Constructors and destructors* Graph();  
 Graph(const std::vector<char>&, const std::vector<std::vector<int>>&);  
  
 *// Searches* std::vector<Vertex\*> depthFirstSearch();  
 std::vector<Vertex\*> breadthFirstSearch();  
  
 *// Kruskal's algorithm* std::vector<Edge> getSortedByWeight();  
 std::vector<Edge> getMST();  
};

*/// Vertex constructor*Vertex::Vertex(char name) {  
 if (!std::regex\_match(std::string(1, name), std::regex("[a-zA-Z]")))  
 throw std::invalid\_argument("Invalid vertex name");  
 \_name = name;  
}  
  
  
*/// Connect this to the <vertex> and set the edge <weight>*void Vertex::setEdge(Vertex \*vertex, int weight) {  
 edges[vertex] = weight;  
}  
  
  
*/// Return char name of the vertex*char Vertex::getName() const {  
 return \_name;  
}  
  
  
*/// Print the vertex`s name*std::ostream& operator<< (std::ostream& os, const Vertex& vertex) {  
 if (vertex.\_name) os << vertex.\_name;  
 return os;  
}  
  
  
*/// Edge constructor*Edge::Edge(int weight, Vertex \*first, Vertex \*second) {  
 this->weight = weight;  
 u = first;  
 v = second;  
}  
  
  
*/// Access to the weight*int Edge::getWeight() const {  
 return weight;  
}  
  
  
*/// Access to the first vertex*Vertex\* Edge::getU() {  
 return u;  
}  
  
  
*/// Access to the second vertex*Vertex\* Edge::getV() {  
 return v;  
}  
  
  
*/// Print the edge*std::ostream& operator<< (std::ostream& os, Edge& edge) {  
 os << \*(edge.getU()) << " <--> " << \*(edge.getV()) << " w = " << edge.getWeight();  
 return os;  
}  
  
  
*/// Compare the weight of two edges, used for sorting*bool Edge::operator< (const Edge& other) const{  
 return weight < other.getWeight();  
}  
  
  
*/// Default constructor for empty graph*Graph::Graph() {  
 vertices = {};  
}  
  
  
*/// Graph constructor from adjacency matrix*Graph::Graph(const std::vector<char> &names, const std::vector<std::vector<int>> &weights) {  
 *// Create all vertices* for (auto &name : names)  
 vertices.push\_back(new Vertex(name));  
  
 *// Add edges between the vertices* for (int i = 0; i < vertices.size(); ++i)  
 for (int j = 0; j < vertices.size(); ++j)  
 if (i != j && weights[i][j] != 0)  
 vertices[i]->setEdge(vertices[j], weights[i][j]);  
}  
  
  
*/// Return a vector of vertices in DFS order*std::vector<Vertex\*> Graph::depthFirstSearch() {  
 if (vertices.empty()) return {};  
  
 std::vector<Vertex\*> result;  
 std::map<Vertex\*, bool> visited;  
 std::stack<Vertex\*> stack;  
  
 stack.push(vertices[0]);  
 visited[vertices[0]] = true;  
 while (!stack.empty()) {  
 *// Move front vertex from stack to result* Vertex\* top = stack.top();  
 stack.pop();  
 result.push\_back(top);  
  
 *// Add unvisited neighbors to stack* for (auto &edge : top->edges) {  
 if (!visited[edge.first]) {  
 stack.push(edge.first);  
 visited[edge.first] = true;  
 }  
 }  
 }  
  
 return result;  
}  
  
  
*/// Return a vector of vertices in BFS order*std::vector<Vertex\*> Graph::breadthFirstSearch() {  
 if (vertices.empty()) return {};  
  
 std::vector<Vertex\*> result;  
 std::map<Vertex\*, bool> visited;  
 std::queue<Vertex\*> queue;  
  
 queue.push(vertices[0]);  
 visited[vertices[0]] = true;  
 while (!queue.empty()) {  
 *// Move from queue to result* Vertex\* front = queue.front();  
 queue.pop();  
 result.push\_back(front);  
  
 *// Add unvisited neighbors to queue* for (auto &edge : front->edges) {  
 if (!visited[edge.first]) {  
 queue.push(edge.first);  
 visited[edge.first] = true;  
 }  
 }  
 }  
  
 return result;  
}

Приложение В

СИСТЕМА НЕПЕРЕСЕКАЮЩИХСЯ МНОЖЕСТВ

class DisjointSet {  
private:  
 std::map<Vertex\*, Vertex\*> parent = {}; *// Map of {vertex : parent}* std::map<Vertex\*, int> rank = {}; *// Map of {vertex : rank}*public:  
 void makeSet(Vertex\*);  
 Vertex\* find(Vertex\*);  
 void unite(Vertex\*, Vertex\*);  
};

*/// Make a new set*void DisjointSet::makeSet(Vertex \*vertex) {  
 parent[vertex] = vertex;  
 rank[vertex] = 0;  
}  
  
  
*/// Find vertex's parent*Vertex\* DisjointSet::find(Vertex \*vertex) {  
 return (parent[vertex] == vertex) ? vertex : parent[vertex] = find(parent[vertex]);  
}  
  
  
*/// Unite two sets*void DisjointSet::unite(Vertex \*x, Vertex \*y) {  
 Vertex\* xRoot = find(x);  
 Vertex\* yRoot = find(y);  
  
 if (xRoot == yRoot) return; *// Already united* if (rank[xRoot] < rank[yRoot]) parent[xRoot] = yRoot; *// (x < y) yRoot becomes parent of xRoot* else {  
 parent[yRoot] = xRoot; *// (x >= y) xRoot becomes parent of yRoot* if (rank[xRoot] == rank[yRoot]) rank[xRoot]++; *// (x == y) xRoot's rank increases by 1* }  
}