МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное бюджетное образовательное учреждение высшего образования
"МИРЭА - Российский технологический университет"

РТУ МИРЭА

Институт информационных технологий (ИТ)
Кафедра математического обеспечения и стандартизации информационных технологий (МОСИТ)

ОТЧЕТ ПО ПРАКТИЧЕСКОЙ РАБОТЕ №7.2

по дисциплине
«Структуры и алгоритмы обработки данных»

Тема. Нелинейные структуры
Графы: создание, алгоритмы обхода, важные задачи теории графов

Выполнил студент группы ИКБО-43-23 Кощеев М. И.

Принял старший преподаватель Рысин М.Л.

Москва 2024

СОДЕРЖАНИЕ

1 ЦЕЛЬ 3
2 ХОД РАБОТЫ 4
4 ВЫВОДЫ 9
5 ИСПОЛЬЗУЕМЫЙ ИСТОЧНИК 10

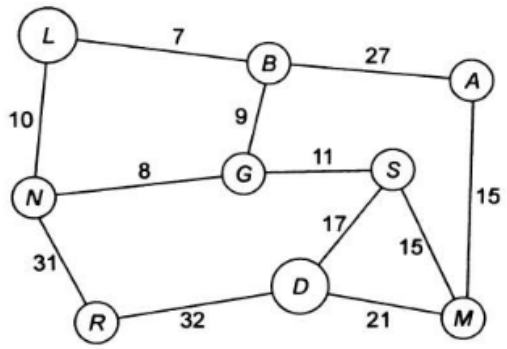
1 ЦЕЛЬ

Освоить приемы создания, алгоритмы обхода, важные задачи теории графов.

2 ХОД РАБОТЫ

Составить программу создания графа и реализовать процедуру для работы с графом, определенную индивидуальным вариантом задания.

Вариант 23:

23	Нахождение кратчайшего пути методом Йена	

Алгоритм (dijkstra):

1. Создаём структуры для хранения расстояний, предыдущих вершин и посещённых вершин.
2. Инициализируем все расстояния как бесконечность, кроме начальной вершины (её расстояние равно 0).
3. Создаём приоритетную очередь для хранения вершин с минимальными расстояниями и добавляем в неё начальную вершину.
4. Пока очередь не пуста:
   a) Извлекаем вершину с минимальным расстоянием.
   b) Если вершина уже была посещена, пропускаем её.
   c) Отмечаем вершину как посещённую.
   d) Для всех соседей текущей вершины: i. Если ребро между вершинами заблокировано, пропускаем его. i. Вычисляем новое расстояние до соседа. i. Если оно меньше текущего, обновляем расстояние и добавляем соседа в очередь.
5. Если достигли конечной вершины, восстанавливаем путь от конца до начала, используя предыдущие вершины.

Код программы

```
pair<vector<char>, int> dijkstra(char start, char end, set<pair<char, char>>& blockedEdges) const {
    map<char, int> dist;
    map<char, char> prev;
    set<char> visited;

    for (auto& pair : adjList) {
        dist[pair.first] = INF;
    }
    dist[start] = 0;

    auto cmp = [&dist](char left, char right) -> bool { return dist[left] > dist[right]; };
    priority_queue<char, vector<char>, decltype(cmp)> pq(cmp);
    pq.push(start);

    while (!pq.empty()) {
        char current = pq.top();
        pq.pop();
        if (visited.count(current)) continue;

        visited.insert(current);

        for (auto& pair : adjList.at(current)) {
            char neighbor = pair.first;
            int weight = pair.second;

            if (blockedEdges.count({ x: &current, y: &neighbor}) || blockedEdges.count({ x: &neighbor, y: &current})) {
                continue;
            }

            int newDist = dist[current] + weight;
            if (newDist < dist[neighbor]) {
                dist[neighbor] = newDist;
                prev[neighbor] = current;
                pq.push(neighbor);
            }
        }
    }

    vector<char> path;
    int totalWeight = dist[end];
    for (char at = end; at != 0; at = prev[at])
        path.push_back(at);
    reverse(first: path.begin(), last: path.end());
    return (path.front() == start) ? make_pair(>>: path, >>: totalWeight) : make_pair(x: vector<char>(), y: INF);
}
```

Алгоритм (yenKShortestPaths):

1. Получаем первый кратчайший путь с помощью алгоритма Дейкстры.
2. Добавляем его в список кратчайших путей.
3. Для каждого следующего пути (от 2-го до k-го): a) Для каждого ребра в последнем найденном пути: i. Удаляем это ребро из графа, блокируем его. i. Запускаем алгоритм Дейкстры с заблокированными ребрами. i. Если найден новый путь, вычисляем его длину и добавляем в список кандидатов. b) Из всех возможных путей выбираем тот, который имеет минимальную длину.
4. Повторяем, пока не найдём k путей или не исчерпаем все возможные пути.

Код

```
    vector<pair<vector<char>, int>> yenKShortestPaths(char start, char end, int k) {
        vector<pair<vector<char>, int>> shortestPaths;
        set<pair<vector<char>, int>> candidates;

        set<pair<char, char>> blockedEdges;
        auto basePath :pair<vector<char>, int> = dijkstra(start, end, [&]blockedEdges);
        if (basePath.first.empty()) return shortestPaths;

        shortestPaths.push_back(basePath);
        blockedEdges.clear();

        for (int i = 1; i < k; ++i) {
            const auto& lastPath :const vector<char>& = shortestPaths.back().first;

            for (size_t j = 0; j < lastPath.size() - 1; ++j) {
                char spurNode = lastPath[j];

                Graph tempGraph = *this;
                set<pair<char, char>> tempBlockedEdges = blockedEdges;

                for (size_t pathIdx = 0; pathIdx <= j; ++pathIdx) {
                    char u = lastPath[pathIdx], v = lastPath[pathIdx + 1];
                    tempGraph.removeEdge(u, v);
                    tempBlockedEdges.insert(x: ∴{x: [&]u, y: [&]v});
                }

                auto spurPath :pair<vector<char>, int> = tempGraph.dijkstra(start:spurNode, end, [&]tempBlockedEdges);
                if (spurPath.first.empty()) continue;

                vector<char> totalPath(first:lastPath.begin(), last:lastPath.begin() + j);
                totalPath.insert(position: ∴totalPath.end(), first:spurPath.first.begin(), last:spurPath.first.end());
                int totalWeight = 0;

                for (size_t idx = 0; idx < totalPath.size() - 1; ++idx) {
                    char u = totalPath[idx], v = totalPath[idx + 1];
                    for (auto& edge :pair<char, int>& : adjList.at(u)) {
                        if (edge.first == v) {
                            totalWeight += edge.second;
                            break;
                        }
                    }
                }
```

```
                candidates.insert(x:make_pair([>>]totalPath, [>>]totalWeight));
            }

            if (candidates.empty()) break;

            shortestPaths.push_back(*candidates.begin());
            candidates.erase(position:candidates.begin());
        }

        return shortestPaths;
    }

    void removeEdge(char u, char v) {
        adjList[u].erase(
            first: ∴remove_if(first:adjList[u].begin(), last:adjList[u].end(), pred:[v](const auto& edge) { return edge.first == v; }),
            last: ∴adjList[u].end());
        adjList[v].erase(
            first: ∴remove_if(first:adjList[v].begin(), last:adjList[v].end(), pred:[u](const auto& edge) { return edge.first == u; }),
            last: ∴adjList[v].end());
    }
};
```

Алгоритм (removeEdge):

1. Для каждой вершины находим ребро с указанным соседним узлом и удаляем его.
2. Повторяем для второй вершины, удаляя обратное ребро.

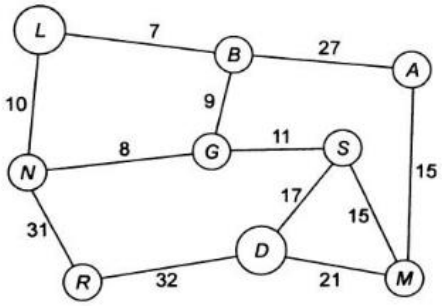
Код

```
    void removeEdge(char u, char v) {
        adjList[u].erase(
            first: remove_if(first: adjList[u].begin(), last: adjList[u].end(), pred: [v](const auto& edge) { return edge.first == v; }),
            last: adjList[u].end());
        adjList[v].erase(
            first: remove_if(first: adjList[v].begin(), last: adjList[v].end(), pred: [u](const auto& edge) { return edge.first == u; }),
            last: adjList[v].end());
    }
};

```

Тестирование

1) Граф из варианта




```
Enter the number of edges:12
 Enter the edges in the format (from, to, weight):
L B 7
B A 27
A M 15
M S 15
M D 21
B G 9
D S 17
S G 11
D R 32
R N 31
L N 10
N G 8
Enter the start and end vertices:L M
 Enter the number of shortest paths to find:1
 The shortest paths are:
Path 1(Length: 42): L B G S M
```

2) Произвольный граф


```
Enter the number of edges:4
 Enter the edges in the format (from, to, weight):
A B 1
B C 1
A D 1
D C 1
Enter the start and end vertices:A C
 Enter the number of shortest paths to find:2
 The shortest paths are:
Path 1(Length: 2): A B C
Path 2(Length: 2): A D C
```

4 ВЫВОДЫ

В результате проделанной работы были освоены приемы создания, алгоритмы обхода, важные задачи теории графов.

5 ИСПОЛЬЗУЕМЫЙ ИСТОЧНИК

1. М.Л. РЫСИН, М.В. САРТАКОВ, М.Б. ТУМАНОВА Учебно-методическое пособие СиАОД часть 2