Университет ИТМО

Факультет программной инженерии и компьютерной техники

Лабораторная работа №4

по «Алгоритмам и структурам данных»

Базовые задачи M-P

Выполнил:

Студент группы P3233

Гуменник П. О.

Преподаватели:

Косяков М.С.

Тараканов Д.С.

Санкт-Петербург

2024

Оглавление

Задача M. Цивилизация

Код:

```
#include <iostream>
#include <vector>
#include <queue>
#include <string>
#include <limits>
#include <tuple>
#include <algorithm>

using namespace std;

const vector<pair<int, int>> directions = {{-1, 0}, {0, 1}, {1, 0}, {0, -1}};
const vector<char> dirSymbols = {'N', 'E', 'S', 'W'};

struct Cell {
    int row, col, cost;
    string path;
    Cell(int r, int c, int co, string p) : row(r), col(c), cost(co), path(move(p)) {}

    bool operator>(const Cell& other) const {
        return cost > other.cost;
    }
};

bool isValid(int row, int col, int N, int M, const vector<string>& map) {
    return row >= 0 && row < N && col >= 0 && col < M && map[row][col] != '#';
}

int main() {
    int N, M;
    cin >> N >> M;
    int startX, startY, endX, endY;
    cin >> startX >> startY >> endX >> endY;
    startX--; startY--;
    endX--; endY--;

    vector<string> map(N);
    for (int i = 0; i < N; ++i) {
        cin >> map[i];
    }

    vector<vector<int>> dist(N, vector<int>(M, numeric_limits<int>::max()));
    priority_queue<Cell, vector<Cell>, greater<Cell>> pq;
    dist[startX][startY] = 0;
    pq.emplace(startX, startY, 0, "");

    while (!pq.empty()) {
        Cell current = pq.top();
        pq.pop();

        if (current.row == endX && current.col == endY) {
            cout << current.cost << endl;
            cout << current.path << endl;
            return 0;
        }

        for (int i = 0; i < 4; ++i) {
            int newRow = current.row + directions[i].first;
            int newCol = current.col + directions[i].second;
            if (isValid(newRow, newCol, N, M, map)) {
                int newCost = current.cost + (map[newRow][newCol] == '.' ? 1 : 2);
                if (dist[newRow][newCol] > newCost) {
                    dist[newRow][newCol] = newCost;
                    pq.emplace(newRow, newCol, newCost, current.path + dirSymbols[i]);
                }
            }
        }
    }

    cout << -1 << endl;
    return 0;
}
```

Пояснение к примененному алгоритму:

Алгоритм, который мы используем, основан на алгоритме Дейкстры. Этот алгоритм гарантирует нахождение кратчайшего пути в графах с неотрицательными весами рёбер. Рассмотрим доказательство корректности алгоритма для данной задачи:

1. Постановка задачи:

- У нас есть прямоугольная карта размером N * M, где каждый элемент представляет собой поле ('.'), лес ('W') или воду ('#').

- Перемещение по полю стоит 1 единицу времени, перемещение по лесу стоит 2 единицы времени, перемещение по воде невозможно.

- Требуется найти путь с минимальной суммарной стоимостью перемещения от начальной точки до конечной точки.

2. Модель представления задачи:

- Каждая клетка карты представлена как вершина графа.

- Каждое допустимое перемещение (вверх, вниз, влево, вправо) между соседними клетками представляется как ребро графа с соответствующим весом (1 для поля, 2 для леса).

3. Применение алгоритма Дейкстры:

- Мы используем приоритетную очередь для хранения клеток, которые должны быть обработаны, где приоритетом является суммарная стоимость пути до данной клетки.

- Начальная клетка добавляется в очередь с суммарной стоимостью 0.

- В каждой итерации алгоритм извлекает клетку с наименьшей стоимостью из очереди и обновляет стоимости соседних клеток, если путь через текущую клетку дешевле, чем ранее известный путь.

4. Корректность:

- Алгоритм Дейкстры гарантирует, что когда клетка извлекается из очереди, найденная стоимость является наименьшей возможной стоимостью для достижения этой клетки.

- Поскольку все веса ребер неотрицательны (1 или 2), алгоритм корректно обновляет стоимости всех достижимых клеток.

- Если конечная клетка извлекается из очереди, это означает, что найден кратчайший путь до этой клетки.

5. Вывод результата:

- Алгоритм выводит суммарную стоимость и путь, если конечная клетка достижима.

- Если очередь опустела, а конечная клетка не была достигнута, значит путь недостижим и выводится -1.

Оценка алгоритмической сложности

1. Инициализация:

- Заполнение начальных значений стоимости и добавление начальной клетки в очередь требует O(N * M).

2. Работа приоритетной очереди:

- В худшем случае все клетки будут добавлены в очередь и извлечены один раз.

- Добавление и извлечение из приоритетной очереди требует O(log(N * M)).

3. Основной цикл:

- В каждой итерации извлекается одна клетка, и все её соседние клетки обновляются.

- В худшем случае каждая клетка будет обработана один раз, и для каждой клетки будут обновлены до 4 соседних клеток.

- Таким образом, количество операций обновления суммарно составит O(4 * N * M).

С учетом работы с приоритетной очередью, общая сложность алгоритма составляет:

O((N * M) log(N * M))

Задача N. Свинки-копилки

Код:

```cpp
#include <iostream>
#include <vector>
#include <stack>

using namespace std;

void dfs(int v, const vector<vector<int>>& graph, vector<bool>& visited) {
    stack<int> stack;
    stack.push(v);
    while (!stack.empty()) {
        int node = stack.top();
        stack.pop();
        if (!visited[node]) {
            visited[node] = true;
            for (int neighbor : graph[node]) {
                if (!visited[neighbor]) {
                    stack.push(neighbor);
                }
            }
        }
    }
}

int main() {
    int n;
    cin >> n;

    vector<vector<int>> graph(n);
    for (int i = 0; i < n; ++i) {
        int keyLocation;
        cin >> keyLocation;
        keyLocation--; // Convert to zero-indexed
        graph[i].push_back(keyLocation);
        graph[keyLocation].push_back(i); // Make the graph undirected
    }

    vector<bool> visited(n, false);
    int components = 0;

    for (int i = 0; i < n; ++i) {
        if (!visited[i]) {
            dfs(i, graph, visited);
            components++;
        }
    }

    cout << components << endl;

    return 0;
}
```

Пояснение к примененному алгоритму:

Объяснение алгоритма

1. Входные данные:

- Читаем число копилок n.

- Читаем местоположение ключей для каждой копилки и строим граф, добавляя ребра между копилками.

2. Построение графа:

- Граф представлен в виде списка смежности. Для каждой копилки i добавляем ребро к копилке, в которой находится ключ от неё.

3. Поиск компонент связности:

- Используем DFS для обхода графа. Каждый запуск DFS из непосещенной вершины означает нахождение новой компоненты связности.

- Каждый раз, когда мы находим новую компоненту, увеличиваем счётчик компонент.

4. Вывод результата:

- Количество компонент связности равно минимальному количеству копилок, которые необходимо разбить.

Алгоритм для решения задачи состоит из двух основных шагов:

1. Построение графа.

2. Поиск компонент связности в графе.

1. Построение графа:

- Мы представляем каждую копилку как вершину графа.

- Добавляем ребра между вершинами согласно правилам: если ключ от копилки i находится в копилке j, то добавляем ребро между вершинами i и j.

- Граф делаем неориентированным, так как если мы можем получить ключ от копилки i, мы можем открыть копилку j и наоборот.

2. Поиск компонент связности:

- Используем обход в глубину (DFS) для поиска всех компонент связности в графе.

- Каждая компонента связности представляет собой набор копилок, которые можно открыть, разбив хотя бы одну из них.

Доказательство корректности:

- Если две копилки находятся в одной компоненте связности, это означает, что можно получить ключ от одной из них, разбив другую. Следовательно, для доступа ко всем копилкам в компоненте достаточно разбить одну копилку.

- Подсчет количества компонент связности дает минимальное количество копилок, которые необходимо разбить. Каждая компонента связности требует минимум одного разбиения, чтобы получить доступ ко всем копилкам в этой компоненте.

Алгоритмическая сложность

Построение графа:

- Построение графа требует прохода по всем n копилкам, для каждой из которых добавляется одно ребро.

- Сложность этого этапа: O(n).

Поиск компонент связности с помощью DFS:

- Запускаем DFS из каждой непосещенной вершины. В худшем случае обходим все n вершины и все n ребра.

- Сложность этого этапа: O(n).

Таким образом, общая сложность алгоритма составляет O(n).

Задача O. Долой списывание!

Код:

```cpp
#include <iostream>
#include <vector>
#include <queue>

using namespace std;

bool isBipartite(const vector<vector<int>>& graph, int N) {
    vector<int> color(N, -1);  // -1: uncolored, 0: color 1, 1: c

    for (int start = 0; start < N; ++start) {
        if (color[start] == -1) {
            queue<int> q;
            q.push(start);
            color[start] = 0;

            while (!q.empty()) {
                int node = q.front();
                q.pop();

                for (int neighbor : graph[node]) {
                    if (color[neighbor] == -1) {
                        color[neighbor] = 1 - color[node];
                        q.push(neighbor);
                    } else if (color[neighbor] == color[node]) {
                        return false;
                    }
                }
            }
        }
    }
    return true;
}

int main() {
    int N, M;
    cin >> N >> M;

    vector<vector<int>> graph(N);
    for (int i = 0; i < M; ++i) {
        int u, v;
        cin >> u >> v;
        --u;  // Convert to zero-indexed
        --v;  // Convert to zero-indexed
        graph[u].push_back(v);
        graph[v].push_back(u);
    }

    if (isBipartite(graph, N)) {
        cout << "YES" << endl;
    } else {
        cout << "NO" << endl;
    }

    return 0;
}
```

Пояснение к примененному алгоритму:

Пояснение к реализации

1. Построение графа:

   - Читаем количество вершин N и рёбер M.

   - Строим список смежности графа, добавляя каждое ребро в обе стороны (граф неориентированный).

2. Проверка двудольности:

   - Инициализируем массив цветов для вершин color значением -1 (непокрашенные).

   - Запускаем BFS для каждой непокрашенной вершины, красим вершины и проверяем цвет соседей.

   - Если находим вершину с тем же цветом, что и у текущей вершины, возвращаем false.

3. Вывод результата:

   - Если удалось покрасить граф без конфликтов, выводим YES.

   - Иначе, выводим NO.

Доказательство корректности алгоритма

Для задачи определения возможности разделения графа на две группы (двудольность графа) мы используем BFS для покраски графа в два цвета. Вот почему этот алгоритм корректно решает задачу.

1. Определение двудольного графа:

   - Граф называется двудольным, если его вершины можно разбить на два множества таким образом, что никакие две вершины внутри одного множества не соединены ребром.

2. Идея алгоритма:

   - Мы используем BFS для покраски графа. Начинаем с непокрашенной вершины и красим её в один цвет, затем её соседей в другой цвет и так далее.

   - Если на каком-то этапе мы обнаруживаем, что сосед уже покрашен в тот же цвет, что и текущая вершина, значит граф не является двудольным.

3. Алгоритм:

   - Инициализируем массив color размером N (число вершин), где каждая вершина имеет значение -1 (непокрашена).

- Запускаем BFS для каждой вершины, если она ещё не покрашена. Красим её в один цвет (0), и её соседей в другой цвет (1).

- Проверяем все рёбра: если находим ребро, где обе вершины покрашены в один цвет, возвращаем false (граф не двудольный).

4. Корректность:

- Если алгоритм находит вершины одной компоненты, покрашенные в один цвет, соединённые ребром, это означает, что граф не может быть разделён на две группы без конфликтов.

- Если алгоритм завершает покраску всех компонент без конфликтов, это означает, что граф двудольный, и мы можем разделить вершины на две группы.

Алгоритмическая сложность

1. Построение графа:

- Построение графа из M рёбер занимает O(M) времени.

2. Покраска графа (BFS):

- Каждый узел и каждое ребро будет обработано один раз.

- Сложность BFS составляет O(N + M), где N — количество вершин, а M — количество рёбер.

Таким образом, общая сложность алгоритма составляет O(N + M).

Задача P. Авиаперелёты

Код:

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <algorithm>
#include <climits>

using namespace std;

struct Edge {
    int v, weight;
};

bool bfs(const vector<vector<Edge>>& graph, int n, int maxFuel) {
    vector<bool> visited(n, false);
    queue<int> q;
    q.push(0);
    visited[0] = true;

    int visitedCount = 1;

    while (!q.empty()) {
        int u = q.front();
        q.pop();

        for (const auto& edge : graph[u]) {
            if (!visited[edge.v] && edge.weight <= maxFuel) {
                visited[edge.v] = true;
                q.push(edge.v);
                visitedCount++;
            }
        }
    }

    return visitedCount == n;
}

bool isStronglyConnected(int maxFuel, const vector<vector<Edge>>& graph, const vector<vector<Edge>>& reverseGraph, int n) {
    return bfs(graph, n, maxFuel) && bfs(reverseGraph, n, maxFuel);
}

int main() {
    int n;
    cin >> n;

    vector<vector<Edge>> graph(n);
    vector<vector<Edge>> reverseGraph(n);
    int maxWeight = 0;

    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            int weight;
            cin >> weight;
            if (i != j) {
                graph[i].emplace_back(Edge{j, weight});
                reverseGraph[j].emplace_back(Edge{i, weight});
                maxWeight = max(maxWeight, weight);
            }
        }
    }

    int left = 0, right = maxWeight;
    int result = maxWeight;

    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (isStronglyConnected(mid, graph, reverseGraph, n)) {
            result = mid;
            right = mid - 1;
        } else {
            left = mid + 1;
        }
    }

    cout << result << endl;

    return 0;
}
```

Пояснение к примененному алгоритму:

Постановка задачи

Нам нужно определить минимальный размер топливного бака, при котором можно долететь из любого города в любой другой с дозаправками, учитывая, что граф ориентированный и вес рёбер может быть разным в разные стороны.

Подход

Мы используем бинарный поиск по возможным размерам топливного бака и проверяем, можно ли достичь всех городов из любого другого города при данном максимальном размере бака. Для этого мы проверяем связность графа как в прямом, так и в обратном направлениях.

Алгоритм

1. Бинарный поиск: Мы используем бинарный поиск по весам рёбер, чтобы найти минимальный возможный размер бака.

2. Проверка связности: Для каждого значения максимального веса ребра (mid), используем BFS, чтобы проверить, можно ли достичь всех городов, учитывая только рёбра с весом, не превышающим mid. Эта проверка выполняется как для исходного графа, так и для обратного графа, чтобы удостовериться, что граф сильно связный (каждая вершина достижима из любой другой).

Корректность

1. Бинарный поиск: Бинарный поиск гарантирует, что мы найдём минимальное значение максимального веса ребра, при котором граф остаётся сильно связным.

2. Проверка связности:

   - Функция bfs проверяет связность графа при данном максимальном весе ребра, используя поиск в ширину (BFS).

   - Функция isStronglyConnected проверяет связность как для исходного графа, так и для обратного графа, что гарантирует сильную связность.

Алгоритмическая сложность

1. Инициализация:

   - Построение графа и обратного графа занимает O(n^2), где n — количество городов.

2. Бинарный поиск:

   - Бинарный поиск по максимальному весу ребра выполняется за O(log W), где W — максимальный вес ребра.

3. Проверка связности:

- Для каждого значения в бинарном поиске мы выполняем две проверки BFS (одну для графа, другую для обратного графа). Каждая проверка BFS занимает O(n + m), где m — количество рёбер.

Итого, общая сложность алгоритма:

- O(n^2) для инициализации.

- O(log W) итераций бинарного поиска.

- Каждая итерация бинарного поиска включает O(n + m) операций на проверку связности.

Таким образом, общая сложность алгоритма составляет:

O((n^2) + (log W) * (n + m))

где n — количество городов, m — количество рёбер, W — максимальный вес ребра.