Университет ИТМО

Факультет программной инженерии и компьютерной техники

Лабораторная работа №4

по «Алгоритмам и структурам данных»

Базовые задачи

Выполнил:

Студент группы P3233

Рахматов Неъматджон

Преподаватели:

Косяков М.С.

Тараканов Д.С.

Санкт-Петербург

2024

Задача №N «Свинки-копилки»

Пояснение к примененному алгоритму:

В этой задаче Васе нужно получить деньги из всех свинок-копилок, разбив как можно меньшее количество копилок. Каждая копилка открывается единственным ключом, который может находиться в другой копилке. Необходимо определить минимальное количество копилок, которые нужно разбить, чтобы достать все ключи и открыть все копилки. Дана последовательность из n свинок-копилок, каждая из которых содержит ключ от одной из других копилок. Требуется найти минимальное количество копилок, которые необходимо разбить, чтобы получить доступ ко всем ключам и открыть все копилки. Задача сводится к поиску циклов в ориентированном графе, где каждая вершина представляет копилку, а ребро из вершины i в вершину j означает, что в копилке i находится ключ от копилки j. Чтобы достать все ключи, нужно разбить по одной копилке в каждом цикле графа. Алгоритм включает следующие шаги:

1. Создать граф, где каждая вершина имеет список смежности для представления копилок и их ключей.
2. Выполнить поиск в глубину (DFS) для обнаружения циклов в графе.
3. Подсчитать количество циклов, что соответствует минимальному количеству копилок, которые нужно разбить.

Анализ времени, сложности и памяти

- **Время выполнения**: Время выполнения алгоритма DFS составляет O(N + M), где N — количество вершин (копилок), а M — количество ребер (ключей).
- **Сложность**: Временная сложность алгоритма O(N + M), что достаточно эффективно для $N \leq 100$.
- **Память**: Используется O(N) дополнительной памяти для хранения состояния вершин и самого графа.

Реализованная программа эффективно решает задачу с использованием поиска в глубину для обнаружения циклов в графе копилок и ключей. Это позволяет минимизировать количество разбиваемых копилок, что соответствует условиям задачи. Программа показывает хорошую производительность и эффективное использование памяти при заданных ограничениях.

```
#include <bits/stdc++.h>

using namespace std;

void depthFirstSearch(int node, vector<pair<list<int>, int>>& graph, int&
cycleCount) {
    graph[node].second = 1;
    for (int adjacent : graph[node].first) {
        if (graph[adjacent].second == 0) {
            depthFirstSearch(adjacent, graph, cycleCount);
        } else if (graph[adjacent].second == 1) {
            cycleCount++;
        }
    }
    graph[node].second = 2;
}

void initializeGraph(int n, vector<pair<list<int>, int>>& graph) {
```

```
    int key;
    for (int i = 0; i < n; i++) {
        cin >> key;
        graph[key - 1].first.push_back(i);
    }
}

int main() {
    int n, cycleCount = 0;
    cin >> n;

    vector<pair<list<int>, int>> graph(n);
    initializeGraph(n, graph);

    for (int i = 0; i < n; i++) {
        if (graph[i].second == 0) {
            depthFirstSearch(i, graph, cycleCount);
        }
    }

    cout << cycleCount << endl;

    return 0;
}
```

Задача №О «Долой списывание!»

Пояснение к примененному алгоритму:

Михаил Дмитриевич хочет разделить учащихся на две группы: списывающих и дающих списывать, чтобы любой обмен записками происходил от одной группы к другой. Необходимо определить, возможно ли это сделать, используя информацию о парах учащихся, обменявшихся записками. Даны NNN учащихся и MMM пар учащихся, обменявшихся записками. Требуется определить, можно ли разделить учащихся на две группы так, чтобы каждый обмен записками происходил между учащимися из разных групп. Проблема сводится к проверке, является ли граф двудольным. Граф двудольный, если его вершины можно раскрасить в два цвета так, чтобы никакие две смежные вершины не имели одинакового цвета. Для проверки двудольности графа используется алгоритм на основе поиска в глубину (DFS).

- **Инициализация графа:** Построить граф на основе входных данных, где вершины представляют учащихся, а ребра представляют пары учащихся, обменявшихся записками.
- **Раскраска графа:** Использовать DFS для раскраски графа в два цвета. Если при попытке раскраски обнаружится, что два смежных узла имеют один и тот же цвет, то граф не является двудольным.

Анализ времени, сложности и памяти

- **Время выполнения:** Алгоритм обхода графа с помощью DFS имеет временную сложность O(N+M)O(N + M)O(N+M), где NNN — количество вершин, а MMM — количество ребер.
- **Сложность:** Алгоритм имеет линейную сложность O(N+M)O(N + M)O(N+M).
- **Память:** Используется O(N+M)O(N + M)O(N+M) памяти для хранения графа и информации о цветах узлов.

Реализованный алгоритм эффективно проверяет двудольность графа, что позволяет определить, можно ли разделить учащихся на две группы так, чтобы любой обмен записками происходил между учащимися из разных групп. Алгоритм показывает хорошую производительность при заданных ограничениях.

```cpp
#include <bits/stdc++.h>

using namespace std;

bool bipartiteCheck(int node, vector<list<int>>& adjacencyList, vector<int>&
color, int currentColor) {
    color[node] = currentColor;
    for (int neighbor : adjacencyList[node]) {
        if (color[neighbor] == 0) {
            if (!bipartiteCheck(neighbor, adjacencyList, color, (currentColor ==
2) ? 1 : 2)) return false;
        } else if (color[neighbor] == currentColor) {
            return false;
        }
    }
    return true;
}
```

```
void initializeGraph(int nodes, int edges, vector<list<int>>& adjacencyList,
vector<int>& color) {
    int u, v;
    for (int i = 0; i < nodes; i++) {
        color[i] = 0;
    }

    for (int i = 0; i < edges; i++) {
        cin >> u >> v;
        u--; v--;
        adjacencyList[u].push_back(v);
        adjacencyList[v].push_back(u);
    }
}

int main() {
    int nodes, edges;
    cin >> nodes >> edges;

    vector<list<int>> adjacencyList(nodes);
    vector<int> color(nodes);

    initializeGraph(nodes, edges, adjacencyList, color);

    bool isBipartite = true;
    for (int i = 0; i < nodes; i++) {
        if (!isBipartite) break;
        if (color[i] == 0) {
            isBipartite = bipartiteCheck(i, adjacencyList, color, 1);
        }
    }

    if (isBipartite) {
        cout << "YES";
    } else {
        cout << "NO";
    }

    return 0;
}
```

Задача №P «Авиаперелёты»

Пояснение к примененному алгоритму:

Для разработки новой модели самолёта компании «Air Бубундия» необходимо подобрать оптимальный размер топливного бака, чтобы самолёт мог долететь от любого города до любого другого с минимально возможным запасом топлива. Задача сводится к нахождению минимального необходимого объема топливного бака, обеспечивающего возможность перелета между любыми двумя городами с учетом возможных дозаправок. Даны nnn городов и матрица n×nn \times nn×n, где элемент aija_{ij}aij представляет расход топлива для перелета из города iii в город jjj. Необходимо определить минимальный размер топливного бака, который позволит самолёту долететь от любого города до любого другого. Для решения задачи используется алгоритм Флойда-Уоршелла, который позволяет найти кратчайшие пути между всеми парами вершин в графе.

- **Инициализация графа:** Построить матрицу смежности на основе входных данных.
- **Алгоритм Флойда-Уоршелла:** Применить алгоритм для нахождения кратчайших путей между всеми парами городов.
- **Определение максимального кратчайшего пути:** Найти максимальное значение среди всех кратчайших путей, что будет оптимальным размером топливного бака.

Анализ времени, сложности и памяти

- **Время выполнения:** Алгоритм Флойда-Уоршелла имеет временную сложность O(n3)O(n^3)O(n3), что приемлемо для n≤100n \leq 100n≤100.
- **Сложность:** Алгоритм имеет временную сложность O(n3)O(n^3)O(n3).
- **Память:** Используется O(n2)O(n^2)O(n2) памяти для хранения матрицы смежности.

Реализованный алгоритм эффективно находит минимальный размер топливного бака, который обеспечивает возможность перелета между любыми двумя городами. Алгоритм Флойда-Уоршелла показал хорошую производительность для данной задачи с учетом ограничений.

```
#include <bits/stdc++.h>

using namespace std;

void depthFirstSearch(int node, const vector<uint32_t>& graph, vector<int>&
colors, int N, int& componentCount, uint32_t limit, int type) {
    colors[node] += 1;
    for (int i = 0; i < N; i++) {
        uint32_t id = type ? (i * N + node) : (node * N + i);
        if (i != node && colors[i] == type && graph[id] <= limit) {
            depthFirstSearch(i, graph, colors, N, componentCount, limit, type);
        }
    }
    componentCount++;
}

void initializeGraph(int N, vector<uint32_t>& graph, uint32_t& minWeight,
```

```
uint32_t& maxWeight) {
    minWeight = numeric_limits<uint32_t>::max();
    maxWeight = 0;
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            uint32_t weight;
            cin >> weight;
            graph[i * N + j] = weight;
            if (i != j) {
                if (weight < minWeight) minWeight = weight;
                if (weight > maxWeight) maxWeight = weight;
            }
        }
    }
}

bool checkComponents(int N, const vector<uint32_t>& graph, vector<int>& colors,
uint32_t limit) {
    fill(colors.begin(), colors.end(), 0);
    int componentCount = 0;
    if (N > 1) {
        depthFirstSearch(0, graph, colors, N, componentCount, limit, 0);
        depthFirstSearch(0, graph, colors, N, componentCount, limit, 1);
    }
    return componentCount == 2 * N;
}

int findMinimumLimit(int N, const vector<uint32_t>& graph, uint32_t minWeight,
uint32_t maxWeight) {
    vector<int> colors(N, 0);
    uint32_t left = minWeight, right = maxWeight;
    while (left < right) {
        uint32_t middle = (left + right) / 2;
        if (checkComponents(N, graph, colors, middle)) {
            right = middle;
        } else {
            left = middle + 1;
        }
    }
    return (checkComponents(N, graph, colors, left) ? left : 0);
}

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

    vector<uint32_t> graph(N * N);
    uint32_t minWeight, maxWeight;
    initializeGraph(N, graph, minWeight, maxWeight);

    int result = findMinimumLimit(N, graph, minWeight, maxWeight);
    cout << result;

    return 0;
}
```