Университет ИТМО

Факультет программной инженерии и компьютерной техники

**Лабораторная работа №4**

по «Алгоритмам и структурам данных»

Базовые задачи / Timus

Выполнил:

Студент группы P32081

Васильченко Роман

Преподаватели:

Косяков М.С.

Тараканов Д.С.

Санкт-Петербург

2023

# Yandex

#### Задача M «Цивилизация»

Пояснение: Если рассмотреть задачу, то можно расписать ячейки карты, как граф и мы должны найти кратчайший путь от точки начала до конечной точки. Для этого в данной задаче наиболее эффективно будет использовать алгоритм BFS, который ищет кратчайший путь от точки до другой точки. В нашем случае мы учитываем время перемещения поселенца. Переменные dx и dy используются для хранения возможных направлений движения поселенца (вверх, вниз, вправо, влево). Стоимость перемещения по клеткам хранится в двумерном массиве cost. Для работы алгоритма начальная стоимость для всех клеток устанавливается как -1, что означает, что эта клетка еще не была посещена.

В каждом шаге, поселенец перемещается из текущей клетки в одну из соседних клеток. Если новая клетка находится в пределах карты и не является водой, то вычисляется новая стоимость перемещения в эту клетку. Если эта клетка еще не была посещена, или если новая стоимость перемещения в эту клетку меньше текущей, то обновляются стоимость перемещения и направление перемещения, а также новая клетка добавляется в очередь.

После того, как все возможные пути были пройдены, алгоритм собирает кратчайший путь от конечной точки до начальной, двигаясь в обратном направлении по направлениям, сохраненным в массиве dir.

Время работы:

**Асимптотика времени**: O(N \* M), где N и M - размеры карты. В худшем случае, алгоритм будет должен пройти все клетки карты.

**Асимптотика памяти**: O(N \* M), так как используются двумерные массивы, размер которых зависит от размеров карты.

Код:

#include <bits/stdc++.h>

using namespace std;

int main() {

int N, M, start\_x, start\_y, goal\_x, goal\_y;

cin >> N >> M >> start\_x >> start\_y >> goal\_x >> goal\_y;

vector<string> grid(N);

for (int i = 0; i < N; i++)

cin >> grid[i];

vector<vector<int>> cost(N, vector<int>(M, -1));

vector<vector<char>> dir(N, vector<char>(M, ' '));

int dx[] = {-1, 0, 1, 0};

int dy[] = {0, 1, 0, -1};

char dc[] = {'N', 'E', 'S', 'W'};

queue<pair<int, int>> q;

q.push({start\_x - 1, start\_y - 1});

cost[start\_x - 1][start\_y - 1] = 0;

while (!q.empty()) {

auto [x, y] = q.front();

q.pop();

for (int i = 0; i < 4; i++) {

int nx = x + dx[i];

int ny = y + dy[i];

if (nx < 0 || nx >= N || ny < 0 || ny >= M || grid[nx][ny] == '#')

continue;

int new\_cost = cost[x][y] + (grid[nx][ny] == 'W' ? 2 : 1);

if (cost[nx][ny] == -1 || cost[nx][ny] > new\_cost) {

cost[nx][ny] = new\_cost;

dir[nx][ny] = dc[i];

q.push({nx, ny});

}

}

}

if (cost[goal\_x - 1][goal\_y - 1] == -1) {

cout << -1 << endl;

return 0;

}

string path = "";

int x = goal\_x - 1, y = goal\_y - 1;

while (x != start\_x - 1 || y != start\_y - 1) {

path = dir[x][y] + path;

switch (dir[x][y]) {

case 'N':

x++;

break;

case 'S':

x--;

break;

case 'E':

y--;

break;

case 'W':

y++;

break;

}

}

cout << cost[goal\_x - 1][goal\_y - 1] << endl;

cout << path << endl;

return 0;

}

#### Задача N «Свинки-копилки»

Пояснение: В задаче копилок можно также изобразить копилки и ключи в виде графа, а именно в виде ориентированного графа, где вершины представляют копилки, а ребра - ключи, соединяющие копилки с соответствующими ключами. Ребро указывает направление от копилки, для которой требуется ключ, к копилке, для которой этот ключ является ключом открытия. Тогда максимальное число копилок, которое стоит сломать можно решить через алгоритм DFS. Для этого нужно определить, в каких случаях Васе придется разбивать копилку. Если существует цикл в графе, то есть цепочка ключей, возвращающая к изначальной копилке, то Васе придется разбить хотя бы одну копилку из этого цикла.

Алгоритм обхода в глубину идеально подходит для нахождения таких циклов. По сути, мы начинаем с первой копилки и двигаемся по графу, следуя ключам, до тех пор, пока не наткнемся на уже посещенную копилку (что обозначает начало цикла) или пока не закончатся не посещенные копилки. Если мы находим цикл, увеличиваем счетчик break\_count.

Время работы:

**Асимптотика времени**: O(n), где n - количество копилок. В худшем случае, алгоритм будет должен пройти все вершины графа (копилки).

**Асимптотика памяти**: O(n), так как используется массив размера n для хранения информации о каждой копилке (ее связи и статус посещения).

Код:

#include <bits/stdc++.h>

using namespace std;

void DFS(int curr\_node, vector<pair<vector<int>, int>> &piggy\_banks,

int &break\_count) {

piggy\_banks[curr\_node].second = 1;

for (int next\_node : piggy\_banks[curr\_node].first) {

if (piggy\_banks[next\_node].second == 0) {

DFS(next\_node, piggy\_banks, break\_count);

} else if (piggy\_banks[next\_node].second == 1) {

++break\_count;

}

}

piggy\_banks[curr\_node].second = 2;

}

int main() {

ios\_base::sync\_with\_stdio(false);

cin.tie(NULL);

int n, key;

cin >> n;

vector<pair<vector<int>, int>> piggy\_banks(n);

for (int i = 0; i < n; ++i) {

cin >> key;

piggy\_banks[--key].first.push\_back(i);

}

int break\_count = 0;

for (int i = 0; i < n; ++i) {

if (piggy\_banks[i].second == 0) {

DFS(i, piggy\_banks, break\_count);

}

}

cout << break\_count << "\n";

return 0;

}

#### Задача O «Долой списывание!»

Пояснение: Эта задача сводится к тому, что мы должны разделить студентов на 2 части, что можно интерпретировать, как двудольный граф, то есть граф, который разделен на 2 части, где вершины графа это студенты, а ребра это списанные работы. Проверка, возможно ли разделить студентов на две группы так, чтобы обмен записками был только между разными группами, в конечном счете сводится к определению, является ли граф двудольным. Если является, выводится "YES", иначе "NO". Для проверки двудольности графа часто используют метод DFS, потому что алгоритм DFS позволяет обнаруживать циклы нечетной длины в графе, что является признаком невозможности разделить вершины на две непересекающиеся группы, что необходимо для двудольного графа. При обходе графа с помощью DFS, если в процессе обхода встречается ребро между вершинами, принадлежащими одной и той же группе, то это свидетельствует о наличии нечетного цикла и невозможности разделения графа на две группы.

Время работы:

**Асимптотика времени**: O(n). В решении для проверки этого используется обход графа в глубину (DFS). Этот метод посещает каждую вершину графа ровно один раз

**Асимптотика памяти**: O(n), так как хранятся два массива размера n: student\_relations и student\_group. Первый массив хранит информацию о связях между студентами, а второй массив - о группе каждого студента.

Код:

#include <bits/stdc++.h>

using namespace std;

vector<int> student\_relations[100];

int student\_group[100];

bool DFS(int student, int group) {

student\_group[student] = group;

for (auto &related\_student : student\_relations[student]) {

if (student\_group[related\_student] == -1) {

if (!DFS(related\_student, !group)) {

return false;

}

} else if (student\_group[related\_student] == student\_group[student]) {

return false;

}

}

return true;

}

void TestCheating() {

int n, m;

cin >> n >> m;

memset(student\_group, -1, sizeof(student\_group));

for (int i = 0; i < m; ++i) {

int student1, student2;

cin >> student1 >> student2;

student1--;

student2--;

student\_relations[student1].push\_back(student2);

student\_relations[student2].push\_back(student1);

}

bool can\_divide = true;

for (int i = 0; i < n; ++i) {

if (student\_group[i] == -1) {

can\_divide &= DFS(i, 0);

}

}

if (can\_divide) {

cout << "YES" << endl;

} else {

cout << "NO" << endl;

}

}

int main() {

ios\_base::sync\_with\_stdio(false);

cin.tie(NULL);

TestCheating();

return 0;

}

#### Задача P «Авиаперелёты»

Пояснение: Эта задача решается с помощью комбинации алгоритма двоичного поиска и поиска в глубину (DFS). В целом, мы хотим найти минимальный размер топливного бака, который позволит долететь от любого города до любого другого города. Мы делаем это, проверяя каждый возможный размер бака и определяя, позволяет ли он достичь каждого города из каждого другого города. Двоичный поиск используется для определения минимального размера бака. DFS используется для проверки возможности добраться из одного города в другой с текущим размером бака. Причина, почему используется DFS заключается в том, что в задаче требуется найти бак, который позволит посетить все города, а эту задачу можно описать, как посетить все вершины графа, где вершина это город, что является сильной связанностью графа. Поэтому задача сводится к проверке графа на сильную связанность, что решается через DFS.

Время работы:

**Асимптотика времени**: O(n^2 \* log n). Двоичный поиск позволяет найти решение за логарифмическое время, то есть его временная сложность O(log n). Сложность по времени для каждого вызова DFS составляет O(n), где n - количество городов. Таким образом, общая асимптотическая временная сложность алгоритма составляет O(n^2 \* log n), так как для каждого размера бака (до log n раз) мы вызываем DFS дважды (2n).

**Асимптотика памяти**: O(n^2), так как в качестве асимптотической сложности по памяти, мы сохраняем матрицы размером n x n для хранения стоимости полета и возможных полетов, а также массив длиной n для отслеживания посещенных городов.

Код:

#include <iostream>

#include <vector>

using namespace std;

vector<vector<int>> flight\_costs;

vector<vector<bool>> possible\_flights;

vector<bool> cities\_visited;

void DFS(int city, bool reverse) {

cities\_visited[city] = true;

for (int i = 0; i < flight\_costs.size(); ++i) {

if (reverse) {

if (possible\_flights[i][city] && !cities\_visited[i])

DFS(i, reverse);

} else {

if (possible\_flights[city][i] && !cities\_visited[i])

DFS(i, reverse);

}

}

}

bool AllCitiesVisited() {

for (bool visited : cities\_visited)

if (!visited)

return false;

return true;

}

int main() {

ios\_base::sync\_with\_stdio(false);

cin.tie(NULL);

int n;

cin >> n;

flight\_costs.resize(n, vector<int>(n));

possible\_flights.resize(n, vector<bool>(n));

cities\_visited.resize(n);

for (auto &city : flight\_costs)

for (int &cost : city)

cin >> cost;

int left = 0, right = 1e9;

while (left < right) {

int mid = (left + right) / 2;

for (int i = 0; i < n; ++i) {

for (int j = 0; j < n; ++j) {

possible\_flights[i][j] = flight\_costs[i][j] <= mid;

}

}

fill(cities\_visited.begin(), cities\_visited.end(), false);

DFS(0, false);

if (AllCitiesVisited()) {

fill(cities\_visited.begin(), cities\_visited.end(), false);

DFS(0, true);

if (AllCitiesVisited())

right = mid;

else

left = mid + 1;

} else {

left = mid + 1;

}

}

cout << left << endl;

return 0;

}

#### Задача 1162 «Currency Exchange»

Пояснение: Рассматривая эту задачу, в которой необходимо определить, можно ли увеличить свой капитал, осуществляя обмен валюты в разных пунктах обмена можно отметить, что каждый пункт обмена можно представить в виде вершин, а обмен в виде связей, что приведет нас к графу. Так как мы хотим найти схему при которой мы получим доход мы должны найти отрицательный цикл затрат (доход), а в этом нам может может помочь алгоритм Беллмана-Форда, который обычно используется для нахождения кратчайшего пути в взвешенном графе. Однако в данной задаче мы ищем наиболее выгодный путь, что означает максимизацию вместо минимизации.

В общем, алгоритм работает следующим образом:

1. Первоначально инициализируется массив расстояний, который отображает максимальную сумму денег, которую можно получить в каждой валюте. В начале это количество денег, которое у нас есть в начальной валюте, и 0 для всех остальных валют.
2. Затем алгоритм проходит по всем ребрам (обменным пунктам) и пытается рассчитать, сколько денег можно получить, осуществляя обмен через эту точку, с учетом комиссии. Если новая сумма больше текущей записанной для целевой валюты, она обновляется.
3. Этот процесс повторяется n раз, где n - количество валют. Это гарантирует, что все возможные пути через граф будут рассмотрены, поскольку в худшем случае путь может проходить через все вершины.
4. После этого мы делаем еще один проход по всем ребрам для проверки, есть ли какой-либо обмен, который мог бы увеличить нашу сумму денег. Если таковой есть, это означает, что в графе существует цикл положительного веса, и мы можем увеличить свой капитал, делая обмены по этому циклу.

Таким образом, если в графе существует цикл положительного веса, ответ будет "YES", в противном случае "NO".

Время работы:

**Асимптотика времени**: O(n\*m) - Временная сложность алгоритма Беллмана-Форда, где n - количество валют, а m - количество пунктов обмена, поскольку алгоритм делает n проходов по m ребрам.

**Асимптотика памяти**: O(n+m), где n - количество валют (или вершин в графе), а m - количество обменных пунктов (или ребер в графе), так как мы сохраняем массив расстояний для каждой валюты и массив всех ребер.

Код:

#include <bits/stdc++.h>

using namespace std;

struct Edge {

int from, to;

double rate, commission;

};

int main() {

ios::sync\_with\_stdio(false);

cin.tie(0);

int n, m, s;

double v;

cin >> n >> m >> s >> v;

s--;

vector<Edge> edges;

for (int i = 0; i < m; i++) {

int a, b;

double rab, cab, rba, cba;

cin >> a >> b >> rab >> cab >> rba >> cba;

a--; b--;

edges.push\_back({a, b, rab, cab});

edges.push\_back({b, a, rba, cba});

}

vector<double> dist(n, 0.0);

dist[s] = v;

for (int i = 0; i < n; i++) {

for (Edge e : edges) {

dist[e.to] = max(dist[e.to], (dist[e.from] - e.commission) \* e.rate);

}

}

bool increased = false;

for (Edge e : edges) {

if ((dist[e.from] - e.commission) \* e.rate > dist[e.to]) {

increased = true;

break;

}

}

if (increased)

cout << "YES\n";

else

cout << "NO\n";

return 0;

}

#### Задача 1329 «Галактическая история»

Пояснение:

Это задача является примером обхода графа, и она относится к классу задач, которые можно решить с использованием такого алгоритма, как поиск в глубину. Изначальная задача представляет собой дерево, в котором каждая веха истории представляет узел, а связи между вехами представляют ребра. В задаче требуется определить, находится ли один узел (веха) в поддереве другого. Для этого вам нужно провести поиск по дереву, чтобы увидеть, можно ли добраться от одной вехи до другой, идя по ребрам дерева. Поиск в глубину идеально подходит для такого типа задачи. В решении используется два ключевых концепта: время входа и время выхода для каждого узла. Время входа tin[v] - это номер шага, когда мы первый раз посещаем узел, а время выхода tout[v] - это номер шага, когда мы покидаем узел после посещения всех его детей. Эти понятия позволяют нам быстро определять, является ли один узел предком другого: узел u является предком узла v, если и только если время входа u меньше или равно времени входа v, и время выхода u больше или равно времени выхода v.

Время работы:

**Асимптотика времени**: Построение дерева и заполнение tin и tout требует обхода всех вех, что занимает O(N), где N - общее число вех. Обработка каждого запроса выполняется за константное время O(1).

Таким образом, общая асимптотика времени составляет O(N + Q), где Q - количество запросов.

**Асимптотика памяти**: Для хранения дерева используется структура данных map, которая требует память O(N) для N вех. Дополнительно используется память для хранения tin и tout, которая также требует O(N) памяти.

Общая асимптотика памяти составляет O(N).

Код:

#include <bits/stdc++.h>

using namespace std;

map<int, vector<int>> g;

map<int, int> tin, tout;

int timer;

void dfs(int v) {

tin[v] = ++timer;

for (int u : g[v]) {

dfs(u);

}

tout[v] = ++timer;

}

bool is\_ancestor(int u, int v) {

return tin[u] <= tin[v] && tout[u] >= tout[v];

}

int main() {

int n, id, parent;

cin >> n;

for (int i = 0; i < n; ++i) {

cin >> id >> parent;

if (parent != -1) {

g[parent].push\_back(id);

}

}

for (auto &p : g) {

if (tin[p.first] == 0) {

dfs(p.first);

}

}

int q;

cin >> q;

while (q--) {

int a, b;

cin >> a >> b;

if (is\_ancestor(a, b)) {

cout << 1 << "\n";

} else if (is\_ancestor(b, a)) {

cout << 2 << "\n";

} else {

cout << 0 << "\n";

}

}

return 0;

}