

**МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра Вычислительной техники**

**ОТЧЕТ
по лабораторной работе №4
по дисциплине «Алгоритмы и структуры данных»
Тема: «ГРАФЫ»**

Студенты гр. 2306 _____ Коробенко С.А. Платто А.А.

Преподаватель _____ Колинько П.Г.

Санкт-Петербург
2023

Цель работы

Целью выполнения курсовой работы «Графы» является исследование алгоритмов на графах.

Задание (Вариант 5)

Подсчёт расстояний от произвольной вершины до всех остальных вершин в ориентированном ненагруженном графе.

Математическая формулировка задачи в терминах теории множеств

Пусть задан неориентированный граф $G=(V,E)$, где V – множество вершин, E – множество ребер. Граф представлен матрицей смежности A , где $a_{ij} \in E$ для $i,j \in V$. Требуется найти максимальное по мощности множество вершин $C \subseteq V$, такое, что для любых двух вершин u,v из C существует ребро e из E , соединяющее u и v , т.е., $\forall u, v \in C: (u,v) \in E$.

Выбор и обоснование способа представления данных

Матрица смежности — это удобное и эффективное представление графа, особенно для задачи поиска клики наибольшей мощности в неориентированном графе. Вот несколько обоснований использования матрицы смежности для данной задачи:

1. Простота представления:

- Матрица смежности представляет собой простой и интуитивно понятный способ описания взаимных связей между вершинами графа.
- Значения в матрице указывают наличие или отсутствие ребра между соответствующими вершинами.

2. Легкость определения смежных вершин:

- С помощью матрицы смежности легко определить смежные вершины для каждой конкретной вершины графа.

- Это особенно важно для задачи поиска клики, так как для формирования клики необходимо знать, какие вершины соединены ребрами.

3. Эффективность проверки наличия ребра:

- В матрице смежности проверка наличия ребра между двумя вершинами выполняется за константное время ($O(1)$).
- Это ускоряет процесс определения смежности вершин при поиске клики.

4. Удобство работы с алгоритмами:

- Многие алгоритмы, используемые для поиска клик в графе, работают эффективнее при использовании матрицы смежности.
- Это связано с тем, что операции с матрицами (например, умножение) могут быть реализованы эффективно на компьютере.

5. Эффективность поиска клик:

- Для поиска клики матрица смежности позволяет легко проверять, является ли набор вершин кликой.
- Алгоритмы поиска клик могут использовать матрицу смежности для эффективного обхода и проверки свойств клик.

Таким образом, использование матрицы смежности обосновано простотой, эффективностью и удобством работы с данными при решении задачи поиска клики наибольшей мощности в неориентированном графе.

Описание алгоритма и оценка его временной сложности

Представление графов:

1. Матрица смежности

Существует два основных способа представления графов в программировании. Один из них, матрица смежности, используется гораздо реже, но очень просто реализуется. Граф из N вершин задаётся матрицей (двумерным массивом) $N \times N$, в которой $g[i][j]$ - логическое значение, true или false, обозначающее, существует ли ребро из вершины i в вершину j .

Преимущества матрицы смежности:

- Сложность проверки наличия ребра между двумя вершинами: $O(1)$

Недостатки матрицы смежности:

- Занимает N^2 памяти, что неприемлемо для достаточно больших графов.
- Сложность перебора всех вершин, смежных с данной: $O(N)$

2. Список смежности

Гораздо чаще для представления графов используется список смежности. Его идея заключается в хранении для каждой вершины расширяемого массива (вектора), содержащего всех её соседей.

Преимущества списка смежности:

- Использует $O(M)$ памяти, что оптимально.
- Позволяет быстро перебирать соседей вершины.
- Позволяет за $O(\log N)$ проверять наличие ребра и удалять его (при использовании `std::set`).

Недостатки списка смежности:

- При работе с насыщенными графами (количество рёбер близко к N^2) скорости $O(\log N)$ может не хватать (единственный повод использовать матрицу смежности).
- Для взвешенных графов приходится хранить `vector<pair<int, int>>`, что усложняет код.

В данной лабораторной работе мы будем использовать Матрицу смежности.

Алгоритмы обхода графов:

1. Поиск в глубину – это алгоритм обхода вершин графа.

Поиск в ширину производится симметрично (вершины графа просматривались по уровням). Поиск в глубину предполагает продвижение вглубь до тех пор, пока это возможно. Невозможность продвижения означает, что следующим шагом будет переход на последний, имеющий несколько вариантов движения (один из которых исследован полностью), ранее посещенный узел (вершина).

Отсутствие последнего свидетельствует об одной из двух возможных ситуаций:

- все вершины графа уже просмотрены,
- просмотрены вершины доступные из вершины, взятой в качестве начальной, но не все (несвязные и ориентированные графы допускают последний вариант).

Применения алгоритма поиска в глубину

- Поиск любого пути в графе.
- Поиск лексикографически первого пути в графе.
- Проверка, является ли одна вершина дерева предком другой.
- Поиск наименьшего общего предка.
- Топологическая сортировка.

- Поиск компонент связности.

Алгоритм поиска в глубину работает как на ориентированных, так и на неориентированных графах. Применимость алгоритма зависит от конкретной задачи.

Поскольку мы обходим каждого «соседа» каждого узла, игнорируя тех, которых посещали ранее, мы имеем время выполнения, равное $O(V + E)$. $V+E$ означает, что для каждой вершины мы оцениваем лишь примыкающие к ней грани. Возвращаясь, к примеру, каждая вершина имеет определенное количество граней и, в худшем случае, мы обойдем все вершины ($O(V)$) и исследуем все грани ($O(E)$). Мы имеем V вершин и E граней, поэтому получаем $V+E$. Далее, поскольку мы используем рекурсию для обхода каждой вершины, это означает, что используется стек (бесконечная рекурсия приводит к ошибке переполнения стека). Поэтому пространственная сложность составляет $O(V)$.

2. Поиск в ширину подразумевает поуровневое исследование графа:

- вначале посещается корень – произвольно выбранный узел,
- затем – все потомки данного узла,
- после этого посещаются потомки потомков и т.д.

Вершины просматриваются в порядке возрастания их расстояния от корня. Алгоритм прекращает свою работу после обхода всех вершин графа, либо в случае выполнения требуемого условия (например, найти кратчайший путь из вершины 1 в вершину 6).

Применения алгоритма поиска в ширину

- Поиск кратчайшего пути в невзвешенном графе (ориентированном или неориентированном).
- Поиск компонент связности.
- Нахождения решения какой-либо задачи (игры) с наименьшим числом ходов.
- Найти все рёбра, лежащие на каком-либо кратчайшем пути между заданной парой вершин.

- Найти все вершины, лежащие на каком-либо кратчайшем пути между заданной парой вершин.

Алгоритм поиска в ширину работает как на ориентированных, так и на неориентированных графах.

Очередь предполагает обработку каждой вершины перед достижением пункта назначения. Это означает, что, в худшем случае, BFS исследует все вершины и грани.

Несмотря на то, что BFS может казаться медленнее, на самом деле он быстрее, поскольку при работе с большими графами обнаруживается, что DFS тратит много времени на следование по путям, которые в конечном счете оказываются ложными. BFS часто используется для нахождения кратчайшего пути между двумя вершинами.

Таким образом, время выполнения BFS также составляет $O(V + E)$, а поскольку мы используем очередь, вмещающую все вершины, его пространственная сложность составляет $O(V)$.

Также для нахождения кратчайшего пути будем использовать обход в ширину, так как он наиболее эффективный.

С учетом этих факторов, программа способна эффективно обрабатывать графы с количеством вершин и ребер, не превышающими несколько сотен. При работе с графами большего размера может потребоваться уточнение и оптимизация алгоритма, возможно, с применением более сложных структур данных или параллельных вычислений для улучшения производительности.

Результаты работы программы

Результаты контрольных тестов представлены на рисунках 1, 2, 3.

```
Выберите режим:
1 - автогенерация
2 - готовые данные
выбор: 2

Матрица смежности (ориентированный граф):
      1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 0
-----
1 |  0  1  1  0  1
2 |  1  0  1  1  1
3 |  1  1  0  0  0
4 |  0  1  0  0  1
5 |  1  1  0  1  0

Введите стартовую вершину (1-5): 4
Расстояния от вершины 4 до остальных вершин:
Вершина 1: 2
Вершина 2: 1
Вершина 3: 2
Вершина 4: 0
Вершина 5: 1
```

Рисунок 1. Результат с заготовленными данными

```
Выберите режим:
1 - автогенерация
2 - готовые данные
выбор: 1

Введите количество вершин (максимум 20): 5

Матрица смежности (ориентированный граф):
      1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 0
-----
1 |  1  0  1  0  1
2 |  1  1  1  0  1
3 |  1  1  1  1  0
4 |  1  1  1  1  1
5 |  1  1  1  1  1

Введите стартовую вершину (1-5): 4
Расстояния от вершины 4 до остальных вершин:
Вершина 1: 1
Вершина 2: 1
Вершина 3: 1
Вершина 4: 0
Вершина 5: 1
```

Рисунок 2. Результат 1 с автогенерацией

```

Выберите режим:
1 - автогенерация
2 - готовые данные
выбор: 1
Введите количество вершин (максимум 20): 10

Матрица смежности (ориентированный граф):
      1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 0
-----
1 | 1  0  1  1  0  0  1  1  0  1
2 | 1  1  0  1  1  1  1  0  1  1
3 | 1  0  0  1  0  1  1  1  1  1
4 | 1  1  1  1  1  1  1  0  1  1
5 | 1  1  1  1  1  1  1  1  1  1
6 | 1  1  0  0  1  1  0  1  1  1
7 | 1  0  0  1  1  1  1  1  1  0
8 | 1  1  1  1  1  1  1  0  1  0
9 | 1  0  1  1  1  1  0  0  1  1
10| 0  1  1  1  1  0  1  1  0  1

Введите стартовую вершину (1-10): 5
Расстояния от вершины 5 до остальных вершин:
Вершина 1: 1
Вершина 2: 1
Вершина 3: 1
Вершина 4: 1
Вершина 5: 0
Вершина 6: 1
Вершина 7: 1
Вершина 8: 1
Вершина 9: 1
Вершина 10: 1

Process returned 0 (0x0)   execution time : 3.254 s
Press any key to continue.

```

Рисунок 3. Результат 2 с автогенерацией

Выводы

В ходе выполнения данной лабораторной работы были изучены и реализованы основные алгоритмы и структуры данных. В частности, был разработан алгоритм поиска кратчайших расстояний в ориентированных графах с использованием обхода в ширину (BFS).

1. Ориентированный граф и матрица смежности: Был реализован класс `'Graph'`, представляющий ориентированный граф с использованием матрицы смежности. Это позволило эффективно хранить информацию о связях между вершинами.

2. Генерация графа: Реализована возможность создания графа двумя способами: автогенерация случайной матрицы смежности и использование предопределенной матрицы для демонстрации работы алгоритма.

3. Алгоритм поиска кратчайших расстояний: В ходе лабораторной работы был реализован и применен алгоритм обхода в ширину для нахождения кратчайших расстояний от выбранной вершины до всех остальных. Этот алгоритм оказался эффективным для ориентированных графов.

4. Вывод результатов: В результате выполненной лабораторной работы были получены кратчайшие расстояния от заданной вершины до всех остальных вершин графа. Временная сложность алгоритма оказалась линейной относительно числа вершин и ребер в графе.

Лабораторная работа позволила более глубоко понять принципы работы алгоритмов на графах, их применимость и эффективность. Освоение этих концепций важно для развития навыков разработки эффективных алгоритмов и использования их в реальных задачах.

Список использованных источников

Колинко П. Г. Пользовательские структуры данных: Методические указания по дисциплине “Алгоритмы и структуры данных, часть 1”. - СПб.: СПбГЭТУ “ЛЭТИ”, 2023. - 64 с. (вып.2309).

Приложение. Исходные тексты программ.

Основная программа

```
#include <iostream>
#include <queue>
#include <climits>
#include <time.h>

using namespace std;

class Graph {
private:
    int** matrix;
    int N;

public:
    Graph(int n);
    void CreateGraph();
    void CreateRandGraph();
    void PrintGraph();
    void ShortestDistances(int start);
    ~Graph();
};

Graph::Graph(int n) : N(n) {
    matrix = new int*[n];
    for (int i = 0; i < n; ++i) {
        matrix[i] = new int[n];
    }
}
```

```

Graph::~~Graph() {
    for (int i = 0; i < N; ++i) {
        delete[] matrix[i];
    }
    delete[] matrix;
}

void Graph::CreateRandGraph(){
    for(auto i = 0; i < N; ++i){
        for (auto j = 0; j < N; ++j)
            matrix[i][j] = rand() % 15 > 2;
    }
}

void Graph::CreateGraph(){
    int tempmatrix[5][5] = {
        {0, 1, 1, 0, 1},
        {1, 0, 1, 1, 1},
        {1, 1, 0, 0, 0},
        {0, 1, 0, 0, 1},
        {1, 1, 0, 1, 0}
    };

    for(auto i = 0; i < N; ++i){
        for (auto j = 0; j < N; ++j){
            matrix[i][j] = tempmatrix[i][j];
        }
    }
}

```

```

void Graph::PrintGraph(){
    cout << "\nМатрица смежности (ориентированный граф):";
    cout << "\n    1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 0";
    cout << "\n-----";
    for (auto i = 0; i < N; ++i)
    {
        if (i+1 >= 10) cout << "\n " << i+1 << " | ";
        else cout << "\n " << i+1 << " | ";
        for(auto j = 0; j < N; ++j)
            cout << " " << matrix[i][j] << " ";
    }
}

```

```

void Graph::ShortestDistances(int start) {
    queue<int> q;
    int* distances = new int[N];

    for (int i = 0; i < N; ++i) {
        distances[i] = INT_MAX;
    }

    distances[start] = 0;
    q.push(start);

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

        for (int i = 0; i < N; ++i) {
            if (matrix[current][i] && distances[i] == INT_MAX) {

```

```

        distances[i] = distances[current] + 1;
        q.push(i);
    }
}

cout << "Расстояния от вершины " << start + 1 << " до остальных вершин:\n";
for (int i = 0; i < N; ++i) {
    cout << "\nВершина " << i + 1 << ": " << distances[i] << " ";
}

delete[] distances;
}

int main() {
    int n, choose;
    setlocale(LC_ALL, "Russian");
    srand(time(nullptr));

    cout << "Выберите режим:\n 1 - автогенерация\n 2 - готовые данные\n выбор: ";
    cin >> choose;

    if (choose == 1) {
        // Генерация матрицы смежности неориентированного графа
        cout << "Введите количество вершин (максимум 20): ";
        cin >> n;
        Graph A(n);
        A.CreateRandGraph();
        A.PrintGraph();
    }
}

```

```

    int startVertex;
    cout << "\nВведите стартовую вершину (1-" << n << "): ";
    cin >> startVertex;

    A.ShortestDistances(startVertex - 1);
} else if (choose == 2) {
    Graph A(5);
    A.CreateGraph();
    A.PrintGraph();

    int startVertex;
    cout << "\nВведите стартовую вершину (1-5): ";
    cin >> startVertex;

    A.ShortestDistances(startVertex - 1);
}

return 0;
}

```