

Санкт-Петербургский политехнический университет Петра Великого
Институт машиностроения, материалов и транспорта

КУРСОВАЯ РАБОТА

по дисциплине «Объектно-ориентрованное программирование»

Выполнил

студент группы 3331506/00401

Мерзлякова Ю.И.

Руководитель

Ананьевский М.С.

«__» _____ 2023 г

Санкт-Петербург

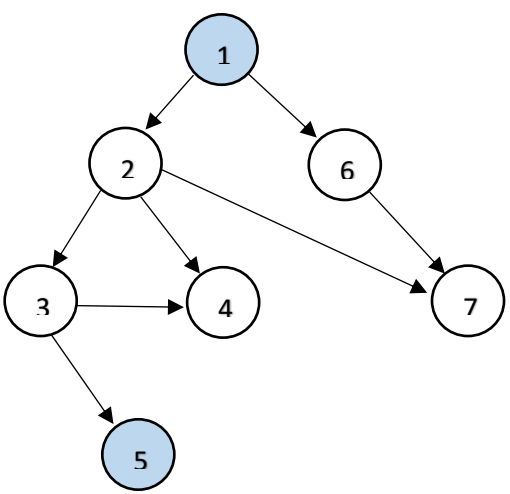
2023

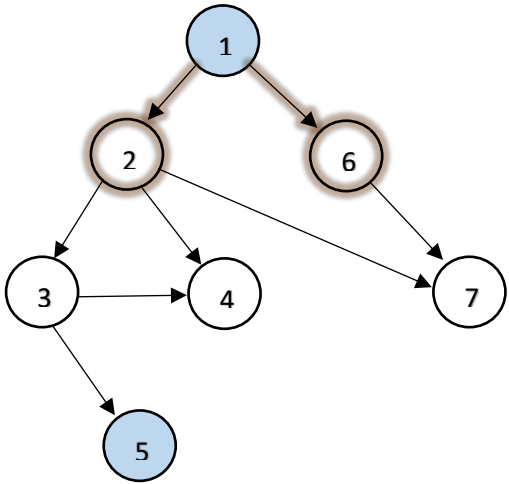
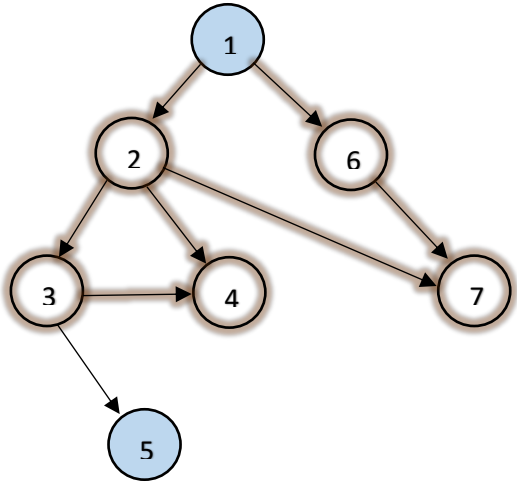
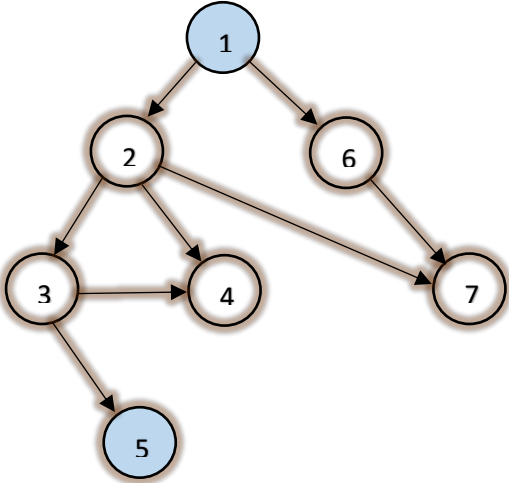
1.1 Предназначение обхода графа

Существует ряд задач, где нужно обойти некоторый граф в глубину или в ширину, так, чтобы посетить каждую вершину один раз. При этом посетить вершины дерева означает выполнить какую-то операцию. Обход графа — это поэтапное исследование всех вершин графа.

1.2 Алгоритм поиска в ширину

1. Двигаемся из начальной вершины.
2. Движемся по ребрам, исходящим из начальной вершины, и по очереди исследуем смежные с ней вершины.
3. Если эти вершины не являются конечными или целевыми, движемся по ребрам, которые исходят из вершин, смежных с начальной.
4. По очереди исследуем вершины этого "уровня". Если эти вершины не являются конечными или целевыми, движемся дальше на следующий уровень по ребрам, которые исходят из этих вершин.
5. Алгоритм повторяется, пока не будут исследованы все вершины и достигнута конечная целевая вершина.

Реализация алгоритма на графе	Описание шагов алгоритма
	Начальная вершина — 1. Необходимо найти вершину 5, исследовав все вершины

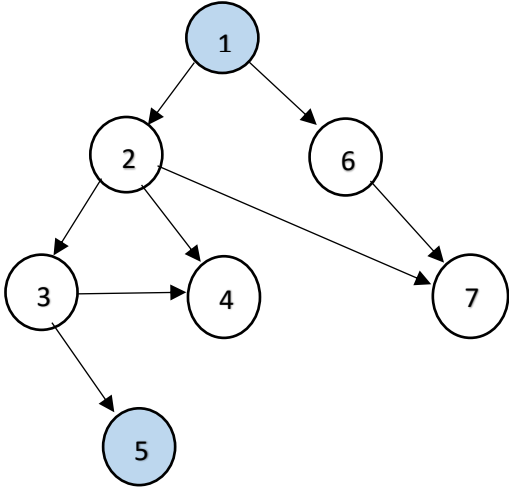
	<p>Из начальной вершины 1 исходит два ребра к двум смежным вершинам 2 и 6. Рассмотрим их по очереди. Вершина 2 не является конечной, так как из нее исходит три ребра. Вершина 6 не является конечной, так как из нее исходит одно ребро. Вершины исследованы и не являются конечными. Двигаемся дальше</p>
	<p>Из вершины 2 исходит три ребра к смежным вершинам 3, 4 и 7. Исследуем их. Вершина 3 не является конечной, так как из нее исходит одно ребро. Вершина 4 является конечной, но не целевой. Вершина 7 тоже является конечной, но не целевой. Вершины исследованы и не являются целевыми, которые требовалось достичь. Двигаемся дальше</p>
	<p>Переходим к вершине 5. Она является той самой конечной целевой вершиной, которую по условию мы должны были достичь. Алгоритм завершен</p>

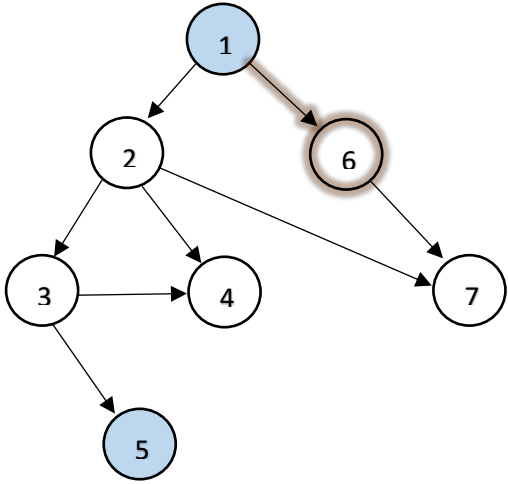
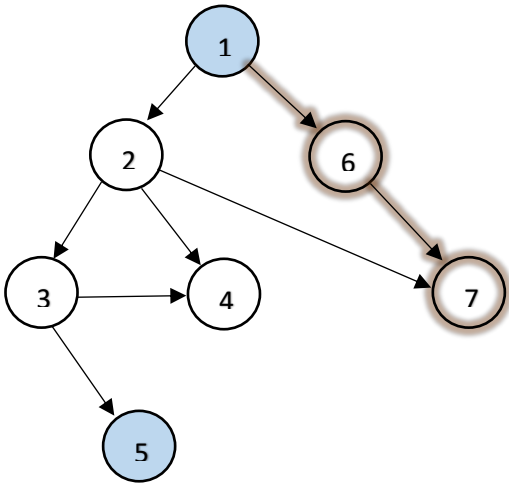
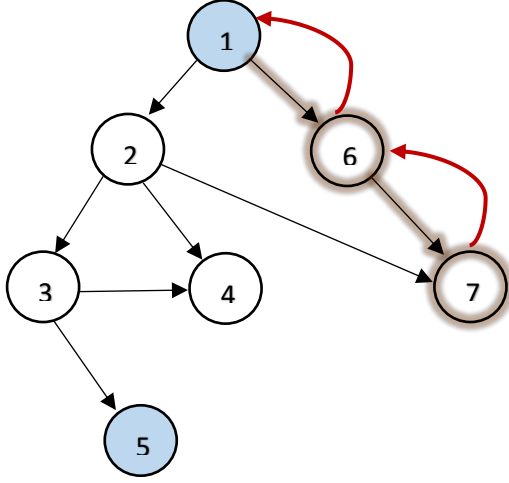
1.3 Описание скорости работы BFS

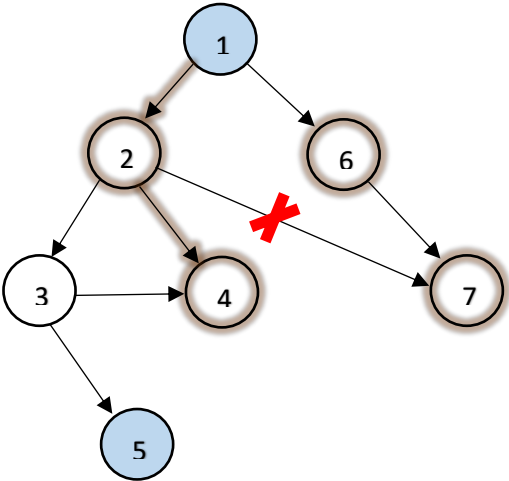
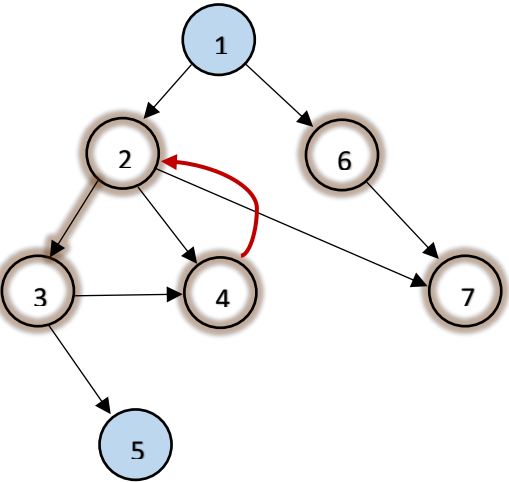
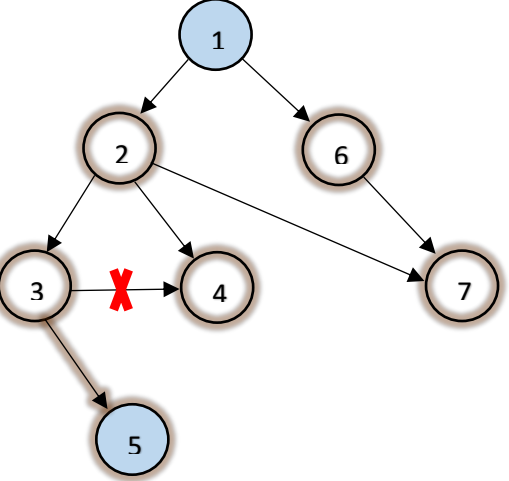
BFS часто используется для нахождения кратчайшего пути между двумя вершинами. Время выполнения BFS составляет $O(V + E)$, а поскольку мы используем очередь, вмещающую все вершины, его пространственная сложность составляет $O(V)$. V — общее количество вершин. E — общее количество граней (ребер).

1.4 Алгоритм поиска в глубину

1. Двигаемся из начальной вершины.
2. Движемся в произвольную смежную вершину.
3. Из этой вершины обходим все возможные пути до смежных вершин.
4. Если таких путей нет или мы не достигли конечной вершины, то возвращаемся назад к вершине с несколькими исходящими ребрами и идем по другому пути.
5. Алгоритм повторяется, пока не будут исследованы все вершины и достигнута конечная вершина.

Реализация алгоритма на графе	Описание шагов алгоритма
	Начальная вершина — 1. Необходимо найти вершину 5. Применяя алгоритм, двигаемся по одному из возможных путей конца и, если не обнаружили вершину 5, возвращаемся и пробуем другой путь.

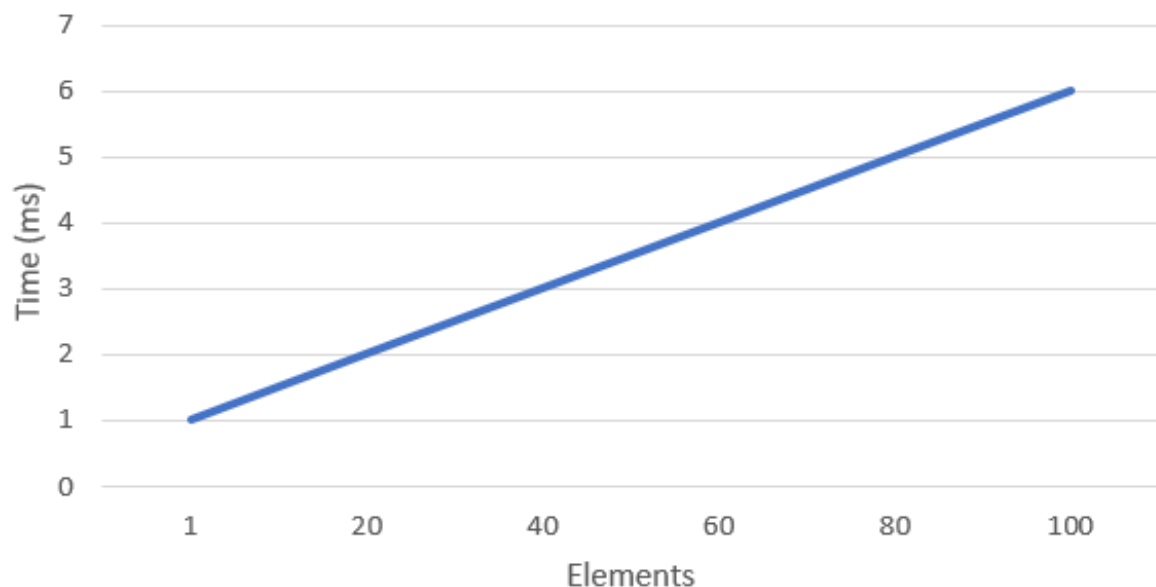
	<p>Произвольно выбираем сначала движение к ближайшей вершине 6. Так как это не конец пути, движемся дальше.</p>
	<p>Движемся к вершине 7. Она является конечной, но не искомой вершиной 5. Движемся дальше.</p>
	<p>Возвращаемся в вершину 6. Она не содержит других путей, поэтому возвращаемся выше к исходной вершине 1. От нее исходит еще один не исследованный путь. Идём по нему к смежной вершине 2.</p>

	<p>Вершина 2 имеет три смежных вершины. Вершина 7 уже изучена, поэтому по этому пути не пойдем. Выбираем вершину 4. Она является конечной, но не искомой вершиной 5. Двигаемся дальше.</p>
	<p>Возвращаемся в вершину 2. Здесь неисследованной осталась вершина 3. Двигаемся к ней.</p>
	<p>Вершина 3 имеет два смежных узла. Вершина 4 уже исследована. Выбираем путь к вершине 5. Она и есть конечная искомая вершина. Алгоритм поиска в глубину закончен. Все вершины исследованы.</p>

1.5 Описание скорости работы DFS

Поскольку мы обходим каждого «соседа» каждого узла, игнорируя тех, которых посещали ранее, мы имеем время выполнения, равное $O(V + E)$. V — общее количество вершин. E — общее количество граней (ребер). $V+E$ 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 $O(V)$.

График зависимости времени выполнения сортировки массива, от его размера, для DFS и BFS будет примерно одинаковым.



Список литературы

1. Хайнеман, Д. Алгоритмы. Справочник. С примерами на C, C++, Java и Python /Д. Хайнеман, Г. Поллис, С. Селков. – Вильямс, 2017.
2. [https://www.yaklass.ru/p/informatika/11-klass/grafy-i-algoritmy-na-grafakh-40408/algoritmy-obkhoda-sviaznogo-grafa-63116/re-c4badbb8-880d-47b9-b11d-352961befb4d#:~:text=Обход%20графа%20—%20это%20поэтапное,ширину%20\(breadth-first%20search%20или%20BFS\)](https://www.yaklass.ru/p/informatika/11-klass/grafy-i-algoritmy-na-grafakh-40408/algoritmy-obkhoda-sviaznogo-grafa-63116/re-c4badbb8-880d-47b9-b11d-352961befb4d#:~:text=Обход%20графа%20—%20это%20поэтапное,ширину%20(breadth-first%20search%20или%20BFS))
3. <https://habr.com/ru/post/504374/>

Приложение 1 (DFS)

```
#include <iostream>
#include <stack> // стек
#include <string>
#include <limits>
using namespace std;

int main()
{
    setlocale(LC_ALL, "Russian");
    short int n;
    int mas[7][7];
    string input[7];
    do {
        cout << "Введите количество вершин: ";
        cin >> n;
    } while ((n < 2) || (n > 7));
    cout << "Введите списки смежности вершин\n";
    cout << "Замечания:\n";
    cout << "2) По концу вводу списка нажимается Enter\n";
    cout << "3) Список задаётся в порядке возрастания номера смежной вершины\n";
    cin.ignore(numeric_limits<streamsize>::max(), '\n');
    for (int i = 0; i < n; i++) {
        cout << "Для " << i + 1 << " вершины:" << endl;
        getline(cin, input[i]);
        for (int j = 0; j < input[i].length(); j++) {
            if (input[i][j] == ' ') {
                input[i].erase(j, 1);
                j--;
            }
        }
    }

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            mas[i][j] = 0;
        }
    }

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < input[i].length(); j++) {
            mas[i][static_cast<int>(input[i][j]) - 48 - 1] = 1;
        }
    }

    //начинаем обход графа в глубину, используя стек
    stack<int> Stack;
    int nodes[7]; // вершины графа
    for (int i = 0; i < n; i++) nodes[i] = 0; // изначально все вершины равны 0
    Stack.push(0); // помещаем в очередь первую вершину
    while (!Stack.empty())
    { // пока стек не пуст
        int node = Stack.top(); // извлекаем вершину
        Stack.pop();
        if (nodes[node] == 2) continue;
        nodes[node] = 2; // отмечаем ее как посещенную
        for (int j = 0; j < 7; j++)
        { // проверяем для нее все смежные вершины
            if (mas[node][j] == 1 && nodes[j] != 2)
            { // если вершина смежная и не обнаружена
                Stack.push(j); // добавляем ее в стек
                nodes[j] = 1; // отмечаем вершину как обнаруженную
            }
        }
    }
}
```

```
        }  
    }  
    cout << node + 1; // выводим номер вершины  
}  
return 0;  
}
```

Приложение 2 (BFS)

```
#include <iostream>
#include <fstream>
#include <list>
#include <algorithm>
using namespace std;
bool bfs(int** graph, int n, int to, int* visited, list<int> &buffer, int* parents);
void print_path(int from, int to, int* parents, ofstream& ofst);
int main() {
    setlocale(LC_ALL, "rus");
    char filenamein[] = "in.txt";
    char filenameout[] = "out.txt";
    ifstream fin(filenamein);
    ofstream fout(filenameout);
    if (!fin || !fout) {
        cout << "ошибка при открытие файла\n";
        system("pause");
        return 1;
    }
    int n, m;
    fin >> n >> m;
    int from, to;
    fin >> from >> to;
    from--; to--;
    int** graph = new int* [n];
    int* visited = new int[n];
    int* parents = new int[n];
    for (int i = 0; i < n; i++) {
        graph[i] = new int[n];
    }
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            graph[i][j] = 0;
    for (int i = 0; i < m; i++)
    {
        int a, b;
        fin >> a >> b;
        graph[a - 1][b - 1] = graph[b - 1][a - 1] = 1;
    }
    for (int i = 0; i < n; i++)
        visited[i] = 0;
    list<int> buffer;
    buffer.push_back(from);
    if (false == bfs(graph, n, to, visited, buffer, parents)) {
        fout << "path does not exist" << endl;
    }
    else {
        print_path(from, to, parents, fout);
    }
    for (int i = 0; i < n; i++)
        delete[] graph[i];
    delete[] graph;
    delete[] visited;
    delete[] parents;
    fin.close();
    fout.close();
}

bool bfs(int** graph, int n, int to, int* visited, list<int>& buffer, int* parents) {
    if (buffer.empty())
        return false;
    int from = buffer.front(); //извлечение первой(текущей) вершины из буфера
    buffer.pop_front();
```

```

visited[from] = true;
if (from == to) {
    return true;
}
for (int i = 0; i < n; ++i) { //наполнение буфера вершинами, смежными с текущей
    if (graph[from][i] == 0) //пропускаются вершины, не смежные с текущей строки
        continue;
    if (visited[i] == true) //пропускаются вершины, уже посещенные ранее
        continue;
    if (find(buffer.begin(), buffer.end(), i) != buffer.end()) //пропускаются
        //вершины, уже добавленные в буфер
        continue;
    parents[i] = from;
    buffer.push_back(i);
}
return bfs(graph, n, to, visited, buffer, parents);
}
void print_path(int from, int to, int* parents, ofstream& ofst) {
    if (to == from) {
        ofst << to + 1;
        return;
    }
    int prefrom = parents[to];
    print_path(from, prefrom, parents, ofst);
    ofst << " -> " << to + 1;
}

```