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

Курсовая работа

Дисциплина: Объектно-ориентированное программирование

Тема: Алгоритмы поиска в ширину (BFS) и в глубину (DFS)

Выполнил студент группы 3331506/90401: Ильясов А.Е.

Преподаватель: Ананьевский М.С.

«___»_____2022 г.

Санкт-Петербург

2022

1. Введение

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.

Для решения таких задач используются два основных алгоритма:

- Поиск в ширину (*breadth-first search* или *BFS*)
- Поиск в глубину (*depth-first search* или *DFS*)

2. Описание алгоритма поиска в ширину

Поиск в ширину подразумевает поуровневое исследование графа:

1. Вначале посещается корень – произвольно выбранный узел.
2. Затем – все потомки данного узла.
3. После этого посещаются потомки потомков и т.д. пока не будут исследованы все вершины.

Вершины просматриваются в порядке роста их расстояния от корня.

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

Для реализации алгоритма удобно использовать очередь.

Рассмотрим работу алгоритма на примере графа на рисунке 1.

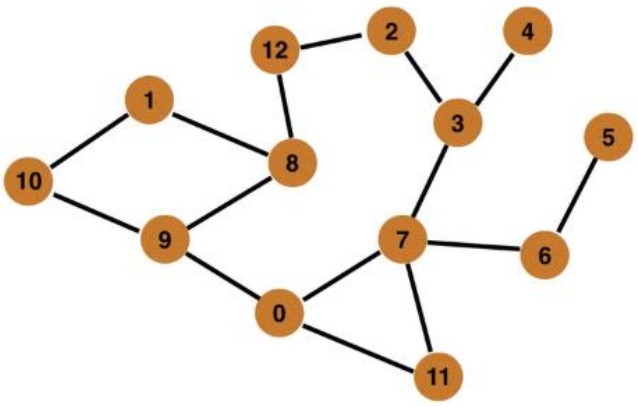


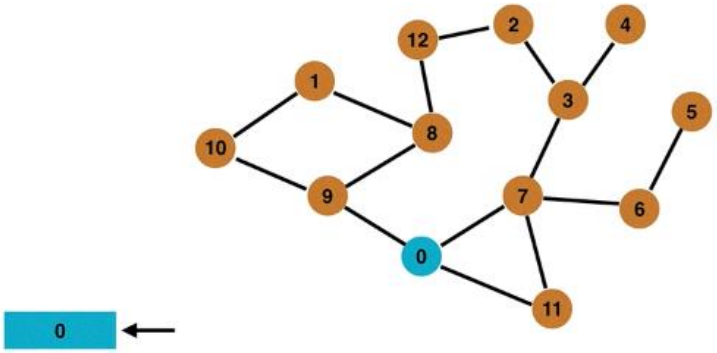
Рисунок 1. Граф для обхода

Каждая вершина может находиться в одном из 3 состояний:

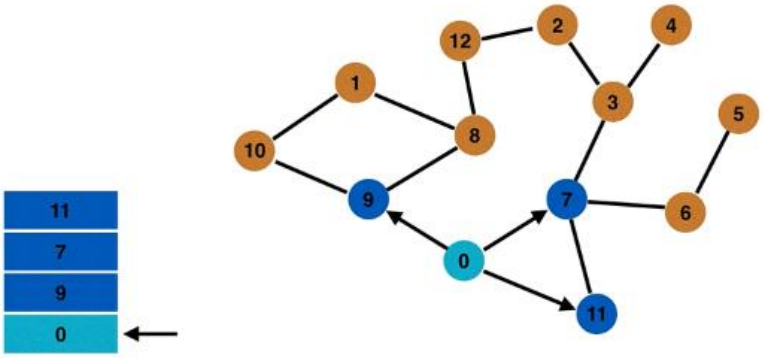
- 0 — коричневый – необнаруженная вершина;
- 1 — синий – обнаруженная, но не посещенная вершина;
- 2 — серый – обработанная вершина.

Голубой – рассматриваемая вершина.

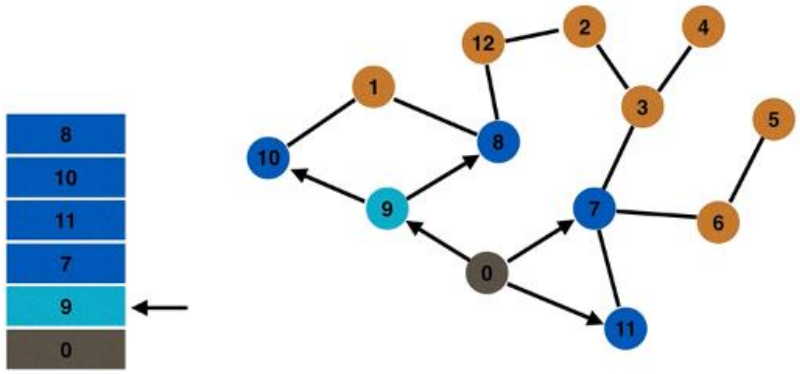
Шаг 1. Добавляем в очередь нулевую вершину.



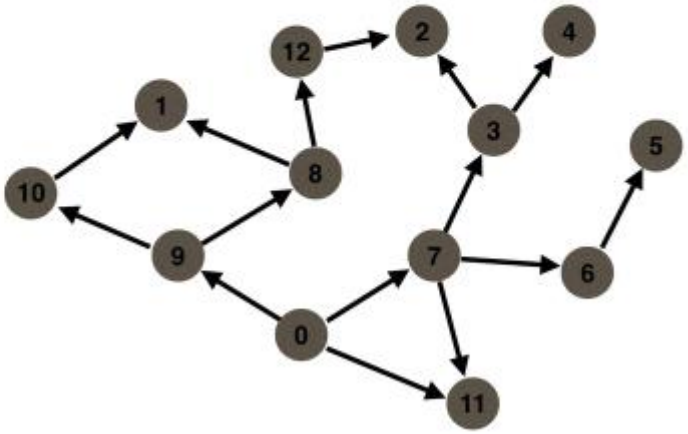
Шаг 2. Добавляем в очередь все вершины, смежные с нулевой вершиной.



Шаг 3. Добавляем в очередь все вершины, смежные с вершиной, находящейся следующей в очереди.



Шаг 4 и далее. Повторить шаг 3 до тех пор, пока в очереди есть непосещенные вершины.



В результате работы алгоритма получаем просмотр каждой вершины графа один раз.

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.

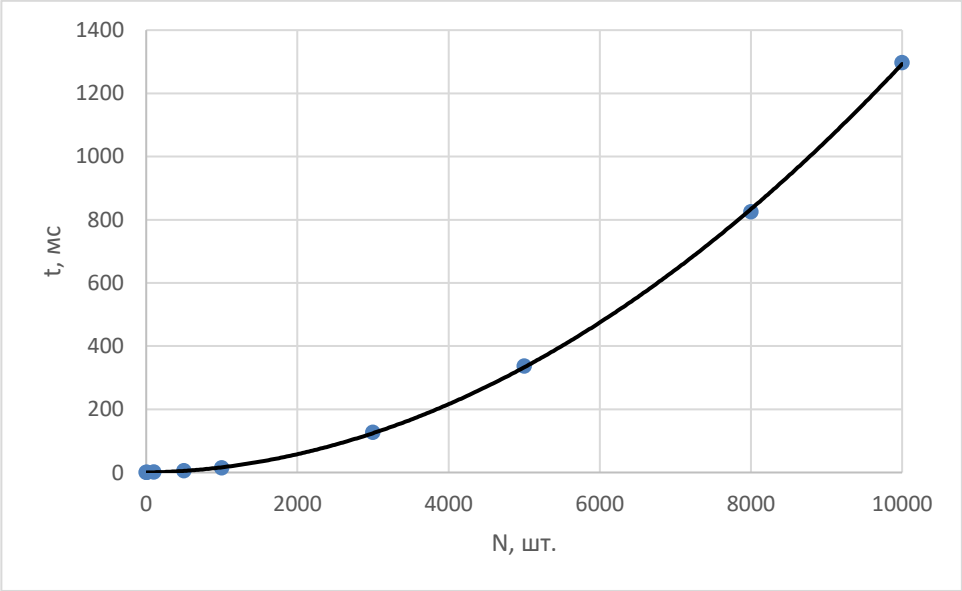
Псевдокод алгоритма поиска в ширину:

```
BFS(start_node) {
 for(all nodes i) visited[i] = false; // изначально список посещённых узлов
                                         пуст
 queue.push(start_node);              // начиная с узла-источника
 visited[start_node] = true;
 while(! queue.empty() ) {            // пока очередь не пуста
  node = queue.pop();                 // извлечь первый элемент в очереди
  foreach(child in expand(node)) {    // все преемники текущего узла
   if(visited[child] == false) {      // ... которые ещё не были посещены
    queue.push(child);                // ... добавить в конец очереди...
    visited[child] = true;            // ... и пометить как посещённые
   }
  }
 }
}
```

3. Исследование алгоритма поиска в ширину

Время выполнения BFS составляет O(V+E), а посколькумы используем очередь, вмещающую все вершины, его пространственная сложность составляет O(V). V — общее количество вершин. E — общее количество граней (ребер).

На графике изображена зависимость времени работы алгоритма поиска в ширину (BFS) в зависимости от количества случайных входных данных.



4. Описание алгоритма поиска в глубину

Стратегия поиска в глубину, как и следует из названия, состоит в том, чтобы идти «вглубь» графа, насколько это возможно.

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 есть, куда идти.

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

Для реализации алгоритма удобно использовать стек или рекурсию.

Рассмотрим работу алгоритма на примере графа на рисунке 2.

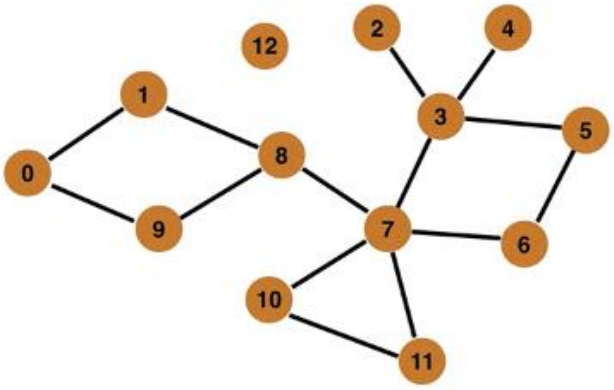


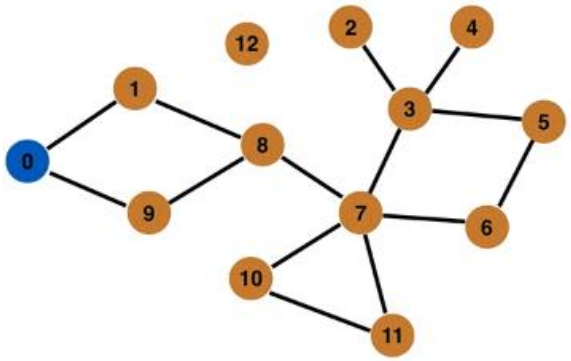
Рисунок 2. Граф для обхода

Каждая вершина может находиться в одном из 3 состояний:

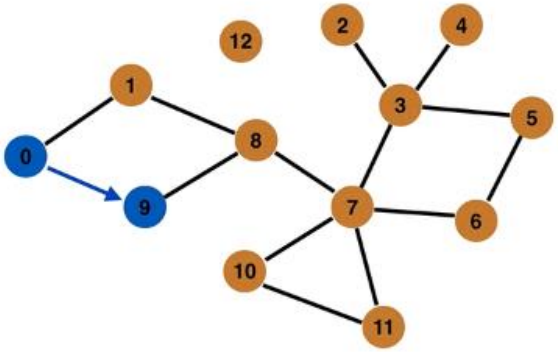
- 0 — коричневый – необнаруженная вершина;
- 1 — синий – обнаруженная, но не посещенная вершина;
- 2 — серый – обработанная вершина.

Голубой – рассматриваемая вершина.

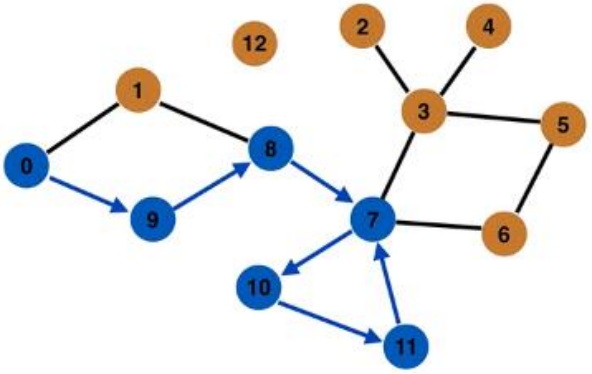
Шаг 1. Начинаем поиск с произвольной (нулевой) вершины.



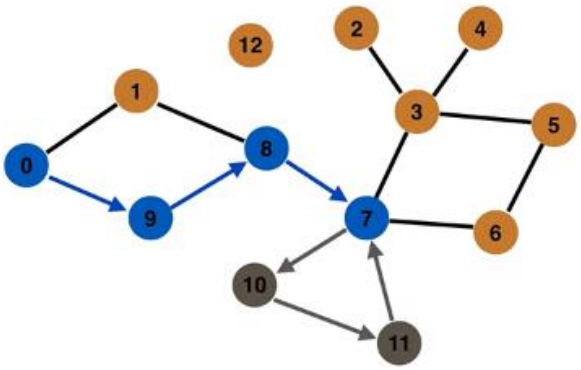
Шаг 2. Переходим к смежной ближайшей вершине.



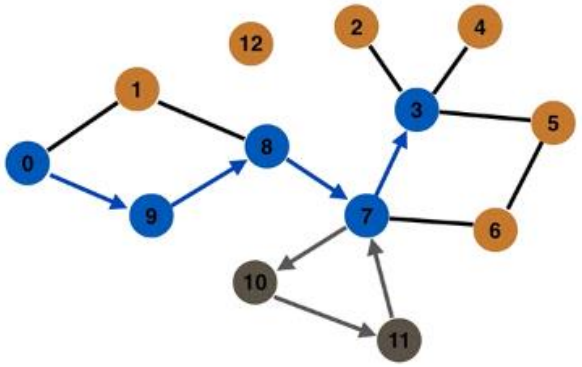
Шаг 3 – Шаг 6. Повторяем шаг 2 до тех пор, пока есть куда двигаться



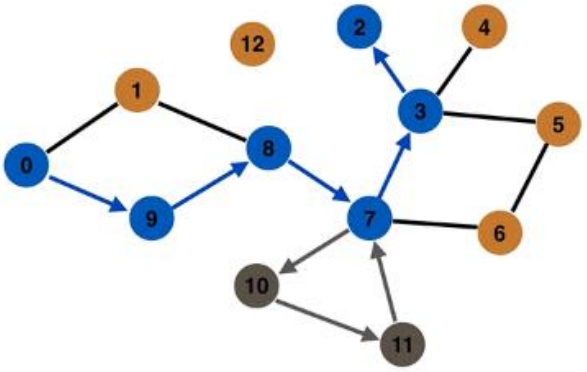
Шаг 7. Возвращаемся в ближайшую вершину с разветвлениями.



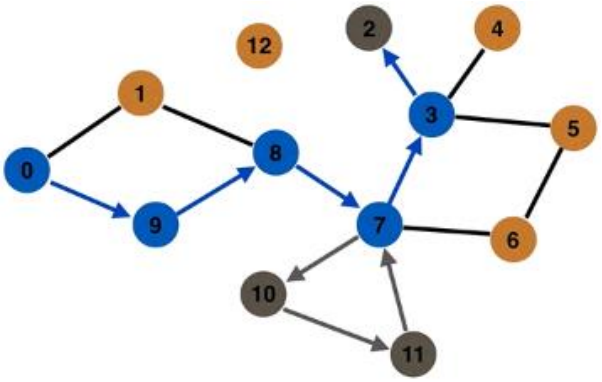
Шаг 8. Переходим к смежной ближайшей вершине (по другому пути).



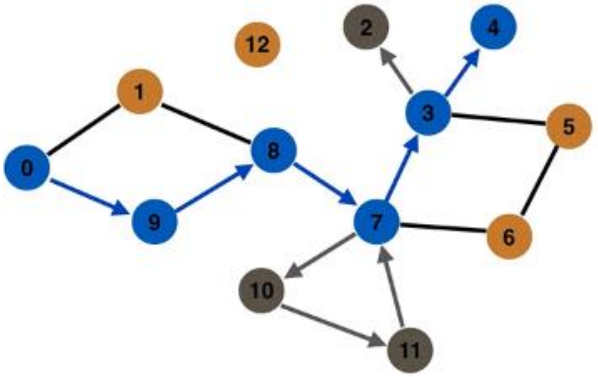
Шаг 9. Повторяем шаг 8 до тех пор, пока есть куда двигаться



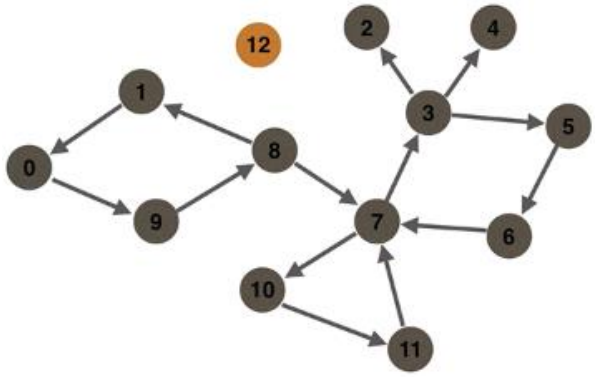
Шаг 10. Возвращаемся в ближайшую вершину с разветвлениями.



Шаг 11. Переходим к смежной ближайшей вершине (по другому пути).



Шаг 11. Повторяем алгоритм до тех пор, пока есть непосещенные вершины.



В результате работы алгоритма получаем просмотр каждой вершины графа один раз.

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.

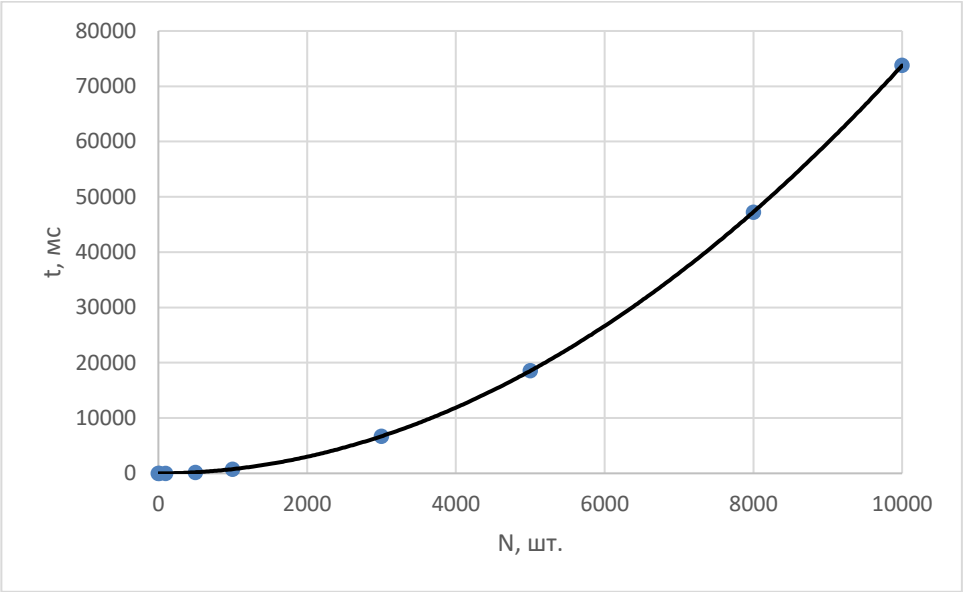
Псевдокод алгоритма поиска в глубину:

```
function doDfs(G[n]: Graph): // функция принимает граф G с количеством вершин n и выполняет обход в глубину во всем графе
   visited = array[n, false]  // создаём массив посещённых вершины длины n, заполненный false изначально

   function dfs(u: int):
      visited[u] = true
      for v: (u, v) in G
         if not visited[v]
            dfs(v)

   for i = 1 to n
      if not visited[i]
         dfs(i)
```

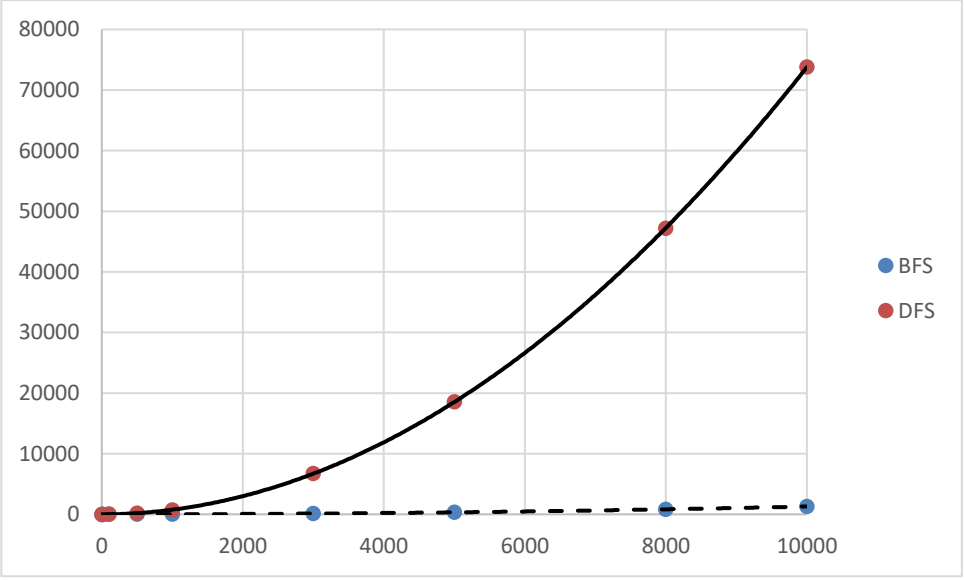
5. Исследование алгоритма поиска в глубину

Оценим время работы обхода в глубину. Процедура dfs вызывается от каждой вершины не более одного раза, а внутри процедуры рассматриваются все ребра. Всего таких ребер для всех вершин в графе O(E), следовательно, время работы алгоритма оценивается как O(V+E).

На графике изображена зависимость времени работы алгоритма поиска в глубину (DFS) в зависимости от количества случайных входных данных.



Наложим производительность алгоритмов BFS и DFS в 1 график:



В теории время работы алгоритмов должно быть одинаковым. Но на практике получаем, что время работы алгоритма BFS существенно быстрее для того же количества данных в отличие алгоритма DFS.

6. Заключение

В ходе выполнения работы были разобраны 2 алгоритма обхода графа: поиск в ширину (BFS) и поиск в глубину (DFS). Для обоих алгоритмов был написан код реализации. Также было произведено сравнение скорости выполнения этих алгоритмов (реализованных в данной работе) для графов различных размеров. По итогам сравнения алгоритм поиска в ширину (BFS) оказался быстрее.

Список литературы

1. Хайнеман, Д. Алгоритмы. Справочник. С примерами на C, C++, Java иPython /Д. Хайнеман, Г. Поллис, С. Селков. – Вильямс, 2017.
2. Седжвик Роберт. Фундаментальные алгоритмы на C++. Анализ/Структуры данных/Сортировка/Поиск: Пер. с англ./Роберт Седжвик.- Издательство «ДиаСофт», 2001.
3. https://prog-cpp.ru/data-graph/

Приложение 1. graph.h

```h
#ifndef BIF_AND_DIF_H
#define BIF_AND_DIF_H
#include <string>

typedef enum {
    ZERO,RANDOM
} GraphType;

class Graph { //граф, представленный матрицей смежности
private:
    bool** adjMatrix;
    bool* data;
    int64_t numVertices;
public:
    Graph();
    Graph(int64_t numVertices, bool* other_matrix);
    Graph(int64_t numVertices, GraphType type);
    ~Graph();
public:
    void addEdge(int64_t i, int64_t j);
    void removeEdge(int64_t i, int64_t j);
    bool isEdge(int64_t i, int64_t j);
    void toString();
public:
    void bfs_search();
    void dfs_search();
};

#endif
```

Приложение 2. graph.cpp

```cpp
#include <iostream>
#include "graph.h"
#include <queue> // очередь
#include <stack> // стек

Graph::Graph() {
    numVertices = 0;
    adjMatrix = nullptr;
    data = nullptr;
}

Graph::Graph(int64_t numVertices, bool* other_matrix) { //матрица смежности создается на основе исходной матрицы
    this->numVertices = numVertices;
    data = new bool[numVertices * numVertices];
    adjMatrix = new bool* [numVertices];
    for (int64_t i = 0; i < numVertices; i++)
    {
        adjMatrix[i] = &data[i * numVertices];
    }
    memcpy(data, other_matrix, numVertices * numVertices * sizeof(bool));
}

Graph::Graph(int64_t numVertices, GraphType type) { //матрица смежности с разными данными
    this->numVertices = numVertices;
    data = new bool[numVertices * numVertices];
    adjMatrix = new bool* [numVertices];
    for (int64_t i = 0; i < numVertices; i++)
    {
        adjMatrix[i] = &data[i * numVertices];
    }
    switch (type)
    {
    case ZERO:
        for (int64_t i = 0; i < numVertices * numVertices; i++)
        {
            data[i] = false;
        }
        break;
    case RANDOM:
        //srand(time(NULL));
        for (int64_t i = 0; i < numVertices; i++) {
            for (int64_t j = 0; j < numVertices && i != j; j++) {
                adjMatrix[i][j] = rand() % 2;
            }
            adjMatrix[i][i] = 0;
        }
        break;
    }
}

void Graph::addEdge(int64_t i, int64_t j) {
    adjMatrix[i][j] = true;
    adjMatrix[j][i] = true;
}

void Graph::removeEdge(int64_t i, int64_t j) {
    adjMatrix[i][j] = false;
    adjMatrix[j][i] = false;
}

bool Graph::isEdge(int64_t i, int64_t j) {
    return adjMatrix[i][j];
}

void Graph::toString() {
    for (int64_t i = 0; i < numVertices; i++) {
        for (int64_t j = 0; j < numVertices; j++)
            std::cout << adjMatrix[i][j] << " ";
        std::cout << "\n";
    }
}
```

```cpp
void Graph::bfs_search() {
    std::cout << "\n";
    int64_t* nodes = new int64_t[numVertices]; // вершины графа (0 - все вершины не рассмотрены)
    for (int64_t i = 0; i < numVertices; i++) {
        nodes[i] = 0;
    }

    std::queue<int64_t> Queue;
    Queue.push(0); // помещаем в очередь первую вершину
    while (!Queue.empty()) // пока очередь не пуста
    {
        int64_t node = Queue.front(); // извлекаем вершину
        Queue.pop();
        nodes[node] = 2; // отмечаем ее как посещенную
        for (int64_t j = 0; j < numVertices; j++) { // проверяем для нее все смежные вершины
            if (adjMatrix[node][j] == true && nodes[j] == 0) { // если вершина смежная и не обнаружена
                Queue.push(j); // добавляем ее в очередь
                nodes[j] = 1; // отмечаем вершину как обнаруженную
            }
        }
        std::cout << node << " "; // выводим номер вершины
    }
    delete[] nodes;
    nodes = nullptr;
}

void Graph::dfs_search() {
    std::cout << "\n";
    int64_t* nodes = new int64_t[numVertices]; // вершины графа (0 - все вершины не рассмотрены)
    for (int64_t i = 0; i < numVertices; i++) {
        nodes[i] = 0;
    }

    std::stack<int64_t> Stack;
    Stack.push(0); // помещаем в очередь первую вершину
    while (!Stack.empty()) {
        int64_t node = Stack.top(); // извлекаем вершину
        Stack.pop();
        nodes[node] = 2; // отмечаем ее как посещенную
        for (int64_t j = numVertices - 1; j >= 0; j--) { // проверяем для нее все смежные вершины
            if (adjMatrix[node][j] == true && nodes[j] == 0) { // если вершина смежная и не обнаружена
                Stack.push(j); // добавляем ее в стек
                nodes[j] = 1; // отмечаем вершину как обнаруженную
            }
        }
        std::cout << node << " "; // выводим номер вершины
    }
    delete[] nodes;
    nodes = nullptr;
}

Graph::~Graph() {
    delete[] data;
    delete[] adjMatrix;
    data = nullptr;
    adjMatrix = nullptr;
}
```

Приложение 3. main.cpp

```cpp
#include<iostream>
#include "graph.h"
#include <queue> // очередь
#include <stack> // стек
#include <ctime>

void bfs_search_time(int64_t numVertices) {
    Graph a(numVertices, RANDOM);
    int start = clock();
    a.bfs_search();
    int end = clock();
    std::cout << "elements: " << numVertices << ", milliseconds: ";
    std::cout << (end - start) * 1000 / CLOCKS_PER_SEC << std::endl;
}

void dfs_search_time(int64_t numVertices) {
    Graph a(numVertices, RANDOM);
    int start = clock();
    a.dfs_search();
    int end = clock();
    std::cout << "elements: " << numVertices << ", milliseconds: ";
    std::cout << (end - start) * 1000 / CLOCKS_PER_SEC << std::endl;
}

int main() {
    //тест BFS
    std::cout << "BFS tests...\n ";
    bfs_search_time(10000);
    bfs_search_time(20000);
    bfs_search_time(30000);
    bfs_search_time(40000);
    bfs_search_time(50000);

    //тест DFS
    std::cout << "\nDFS tests...\n ";
    dfs_search_time(10000);
    dfs_search_time(20000);
    dfs_search_time(30000);
    dfs_search_time(40000);
    dfs_search_time(50000);

    Graph aa(6, *a);
    aa.bfs_search();
    aa.dfs_search();

    bool b[5][5] = { 0,0,1,1,0,
        0,0,1,0,0,
        1,1,0,0,1,
        1,0,0,0,1,
        0,0,1,1,0 };

    Graph bb(5, *b);
    bb.bfs_search();
    bb.dfs_search();

    return 0;
}
```