Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Российский химико-технологический университет имени Д.И. Менделеева»
Кафедра информационных компьютерных технологий

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ № 4

Выполнил студент группы КС-30 Суханова Евгения Валерьевна
Ссылка на репозиторий:
https://github.com/MUCTR-IKT-CPP/EVSuhanova_30/blob/main/Algoritms/laba4.cpp

Приняли: Пысин Максим Дмитриевич
Краснов Дмитрий Олегович

Дата сдачи: 20.03.2023

Оглавление

Описание задачи.

В рамках лабораторной работы необходимо реализовать генератор случайных графов, генератор должен содержать следующие параметры:

- Максимальное/Минимальное количество генерируемых вершин
- Максимальное/Минимальное количество генерируемых ребер
- Максимальное количество ребер связанных с одной вершины
- Генерируется ли направленный граф
- Максимальное количество входящих и выходящих ребер

Сгенерированный граф должен быть описан в рамках одного класса(этот класс не должен заниматься генерацией), и должен обладать обязательно следующими методами:

- Выдача матрицы смежности
- Выдача матрицы инцидентности
- Выдача список смежности
- Выдача списка ребер

В качестве проверки работоспособности, требуется сгенерировать 10 графов с возрастающим количеством вершин и ребер(количество выбирать в зависимости от сложности расчета для вашего отдельно взятого ПК). На каждом из сгенерированных графов требуется выполнить поиск кратчайшего пути или подтвердить его отсутствие из точки А в точку Б, выбирающиеся случайным образом заранее, поиском в ширину и поиском в глубину, замерев время требуемое на выполнение операции. Результаты замеров наложить на график и проанализировать эффективность применения обоих методов к этой задаче.

Описание метода/модели.

Поиск в глубину

Цель алгоритма состоит в том, чтобы пометить каждую вершину как “Пройденная”, избегая при этом циклов.

Алгоритм поиска в глубину работает следующим образом:

1. Помещаем любую вершину графа на вершину стека.
2. Берем верхний элемент стека и добавляем помечаем его как пройденный.
3. Создаем список смежных вершин для этой вершины. Помечаем те вершины, которые еще не помеченные как пройденные, в верх стека.
4. Повторяем шаги 2 и 3, пока стек не станет пустым.

Таким образом, мы получим список тех вершин, в которые мы можем попасть из начальной вершины. Откуда мы можем сделать вывод, существует ли путь из вершины А в вершину Б (где А и Б заранее случайно выбранные вершины).

Если такой путь вообще существует, то мы переходим к поиску в ширину для определения кратчайшего пути.

Поиск в ширину

Данный алгоритм позволяет нам найти кратчайший путь из одной вершины в другую.

Алгоритм поиска в ширину работает следующим образом:

1. Помещаем начальную вершину в очередь.
2. Берем вершину из очереди и помечаем ее как посещенную.
3. Проверяем, есть ли смежные с ней вершины, которые мы еще не обнаружили. Если это так, то отмечаем ее как обнаруженную и добавляем в очередь.
4. Повторяем шаги 2 и 3 пока очередь не будет пуста.

Таким образом, мы получили максимально короткий путь от вершины А в вершину Б (де А и Б заранее случайно выбранные вершины).

Выполнение задачи.

Для реализации генератора графов использовался язык программирования C++.

Класс графа имеет параметры количество вершин (ver), количество ребер (reb), ориентированный ли граф (oriented), матрицу смежности (matrix_smezh), матрицу инцидентности (matrix_inc), список смежности (list_smezh), список ребер (list_reb) и количество связей у каждой вершины (binds).

```
class Grapth {
public:
    int ver;
    int reb;
    bool oriented;
    int** matrix_smezh;
    int** matrix_inc;
    list <list<int>> list_smezh;
    list <list<int>> list_reb;
    int* binds;

    void printMatrixSmezh() { ... }

    void printMatrixInc() { ... }

    void printListSmezh() { ... }

    void printListReb() { ... }

};
```

А также методы вывода матрицы смежности (printMatrixSmezh):

```
void printMatrixSmezh() {
    cout << "Матрица смежности\n";
    cout << setw(4) << "|";
    for (int i = 0; i < ver; i++) {
        cout << setw(2) << (char)(i + 65) << " ";
    }
    cout << endl;
    for (int i = 0; i < ver; i++) {
        cout << setw(2) << (char)(i + 65) << " |";
        for (int j = 0; j < ver; j++) {
            cout << setw(2) << matrix_smezh[i][j] << " ";
        }
        cout << endl;
    }
}
```

Вывод матрицы инцидентности (printMatrixInc):

```
void printMatrixInc() {
    cout << "Матрица инцендентности\n";
    cout << setw(4) << "|";
    for (int i = 0; i < reb; i++) {
        cout << setw(2) << i + 1 << " ";
    }
    cout << endl;
    for (int i = 0; i < ver; i++) {
        cout << setw(2) << (char)(i + 65) << " |";
        for (int j = 0; j < reb; j++) {
            cout << setw(2) << matrix_inc[i][j] << " ";
        }
        cout << endl;
    }
}
```

Вывод списка смежности (printListSmezh):

```
void printListSmezh() {
    list<list<int>> list = list_smezh;
    cout << "Список смежности\n";
    int i = 0;
    while (list.size() > 0) {
        cout << (char)(i + 65) << " | ";
        if (!list.front().empty()) {
            while (list.front().size() > 0) {
                cout << (char)(list.front().front() + 65) << " ";
                list.front().pop_front();
            }
        }
        else {
            cout << "-";
        }
        i++;
        list.pop_front();
        cout << endl;
    }
}
```

Вывод списка ребер (printListReb):

```
void printListReb() {
    cout << "Список ребер\n";
    list<list<int>> list = list_reb;
    int i = 0;
    while (list.size() > 0) {
        cout << i + 1 << " | " << (char)(list.front().front() + 65) <<
            " -> " << (char)(list.front().back() + 65) << endl;
        list.pop_front();
        i++;
    }
}
```

Далее был реализован сам генератор:

```
Grapth grapthGeneration(int i) {
    int max_ver = 100, min_ver = 10;
    int max_reb, min_reb = 10;
    int max_bind;
    bool oriented;
    int max_entry, max_exit;

    Grapth gr;

    gr.ver = (max_ver) / 10 * (i + 1);
    gr.reb = min_reb + i * (min_reb * 2);
    max_bind = ceil( x: ((double)gr.reb * 2) / (double)gr.ver) + rand() % (gr.ver + 1);
    max_entry = max_exit = max_bind / 2 + 1;
    oriented = rand() % 2;

    cout << "\nВершин: " << gr.ver << "\nРебер: " << gr.reb <<
        "\nМаксимальное количество связей: " << max_bind <<
        "\nОриентированный: " << oriented <<
        "\nМаксимальное количество входов: " << max_entry <<
        "\nМаксимальное количество выходов: "<< max_exit << endl;

    gr = genMatSmezh(gr, max_bind, oriented);
    if (oriented) genOriented(gr, max_entry, max_exit);
    gr = genListReb(gr);
    gr = genListSmezh(gr);
    gr = genMatInc(gr, oriented);
```

В нем генерируется количество вершин, ребер, максимальное количество связей у одной вершины, максимальное количество входящих и выходящих ребер, а также определяется, будет ли граф ориентированным.

Далее для данных параметров генерируется матрица смежности:

- Для начала генерируется вектор из M элементов (M - максимально возможное количество ребер), первые N элементов которого равны 1, где N количество ребер в графе. Остальные элементы равны 0. Этот вектор перемешивается функцией random_shuffle.

```
int M = gr.ver * (gr.ver - 1) / 2;
vector <int> arrray_reb(M);
for (int i = 0; i < gr.reb; i++) {
    arrray_reb[i] = 1;
}
random_shuffle(_First: arrray_reb.begin(), _Last: arrray_reb.end());
```

- Далее заполняется матрица смежности так, чтобы выше главной диагонали располагались элементы перемешанного вектора. Ниже главной диагонали зеркально отображены эти же элементы. Также параллельно идет проверка на максимальное количество связей у одной вершины.

```
int k = 0, count = 0;
for (int i = 0; i < gr.ver; i++) {
    gr.matrix_smezh[i][i] = 0;
    for (int j = i + 1; j < gr.ver; j++) {
        if (arrray_reb[k] != 0 && (gr.binds[i] == max_bind || gr.binds[j] == max_bind)) {
            gr.matrix_smezh[i][j] = gr.matrix_smezh[j][i] = 0;
            k++;
        }
        else {
            gr.matrix_smezh[i][j] = gr.matrix_smezh[j][i] = arrray_reb[k];
            k++;
            count++;
        }
        if (abs(_x: gr.matrix_smezh[i][j]) == 1) {
            gr.binds[i]++;
            gr.binds[j]++;
        }
    }
}
```

- После расстановки всех возможных ребер, выставляются те, ребра, которые не смогли на свои места, так как у вершины уже было максимальное количество связей.

```
while (M - count > 0) {
    for (int i = 0; i < gr.ver; i++) {

        if (gr.binds[i] < max_bind) {
            for (int j = i + 1; j < gr.ver; j++) {
                if (gr.matrix_smezh[i][j] == 0 && gr.binds[j] < max_bind) {
                    gr.matrix_smezh[i][j] = gr.matrix_smezh[j][i] = 1;
                    count++;
                    gr.binds[i]++;
                }
                if (M - count <= 0) break;
            }
        }
        if (M - count < 0) break;
    }
}
```

Далее для ориентированного графа идет настройка направлений ребер:

- В случайном порядке идет расстановка направлений (“1” - выход, “-1” - вход)

```
int a[] = { -1, 1 };
for (int i = 0; i < gr.ver; i++) {
    for (int j = i + 1; j < gr.ver; j++) {
        if (gr.matrix_smezh[i][j] == 1) {
            gr.matrix_smezh[i][j] = a[rand() % 2];
            gr.matrix_smezh[j][i] = -gr.matrix_smezh[i][j];
        }
    }
}
```

- Далее идет проверка на максимально количество входов и выходов у одной вершины, и при необходимости идет смена направления.

```
if (countOfEntryExit(array: gr.matrix_smezh, i, N: gr.ver, e: -1) > max_entry) {
    for (int j = 0; j < gr.ver; j++) {
        if (gr.matrix_smezh[i][j] == -1 && countOfEntryExit(array: gr.matrix_smezh, i: j, N: gr.ver, e: -1) < max_entry) {
            swap(&: gr.matrix_smezh[i][j], &: gr.matrix_smezh[j][i]);
            i--;
            break;
        }

    }
}
```

- Далее все “-1” заменяются на “0”.

Далее на основе матрицы смежности идет заполнение списка ребер:

- В список temp записываются начало i и конец j ребра, где i - вершина откуда выходит ребро, а j - вершина, куда это ребро входит.

```
list<int> temp;
for (int i = 0; i < gr.ver; i++) {
    for (int j = 0; j < gr.ver; j++) {
        if (gr.matrix_smezh[i][j] == 1) {
            temp.push_back(_Val: i);
            temp.push_back(_Val: j);

            gr.list_reb.push_back(_Val: temp);
            temp.clear();
        }
    }
}
```

Далее на основе матрицы смежности идет заполнение списка смежности:

- В список temp для каждой отдельной вершины записываются вершины, которые смежны с данной вершиной.

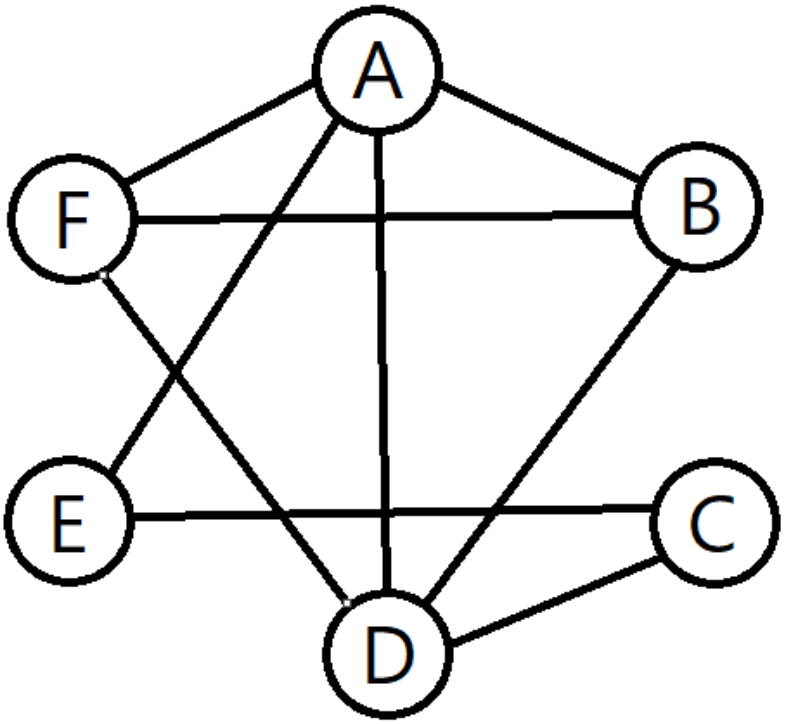
```
list<int> temp;
for (int i = 0; i < gr.ver; i++) {
    for (int j = 0; j < gr.ver; j++) {
        if (gr.matrix_smezh[j][i] == 1) {
            temp.push_back(_Val: j);
        }
    }
    gr.list_smezh.push_back(_Val: temp);
    temp.clear();
}
```

Матрица инцидентности заполняется на основе списка ребер:

- Берется верхний список вершин из списка ребер.
- Заполняется начальная вершина и конечная у взятого ребра. Для ориентированного графа 1 и -1, а для неориентированного обе вершины будут 1.

```
while (temp.size() > 0) {
    if (oriented) {
        gr.matrix_inc[temp.front().front()][i] = 1;
        gr.matrix_inc[temp.front().back()][i] = -1;

    }
    else {
        if (temp.front().front() < temp.front().back()) {
            gr.matrix_inc[temp.front().front()][i] = gr.matrix_inc[temp.front().back()][i] = 1;
        }
    }
    temp.pop_front();
    i++;
}
```

- Пока цикл не дойдет до конца списка выполняется пункт 2.



Далее сгенерированный граф выводится в консоль в виде:

```
Вершин: 6
Ребер: 9
Максимальное количество связей: 6
Ориентированный: 0
Максимальное количество входов: 4
Максимальное количество выходов: 4
```

- матрицы смежности

```
Матрица смежности
   | A  B  C  D  E  F
 A | 0  1  0  1  1  1
 B | 1  0  0  1  0  1
 C | 0  0  0  1  1  0
 D | 1  1  1  0  0  1
 E | 1  0  1  0  0  0
 F | 1  1  0  1  0  0
```

- матрицы инцидентности (представлен другой сгенерированный граф (ориентированный))

```
Матрица инцендентности
   | 1  2  3
 A | 1 -1  0
 B | 0  1 -1
 C |-1  0  1
```

- списка смежности

```
Список смежности
A | B D E F
B | A D F
C | D E
D | A B C F
E | A C
F | A B D
```

- списка ребер

```
Список ребер
1  | A -> B
2  | A -> D
3  | A -> E
4  | A -> F
5  | B -> A
6  | B -> D
7  | B -> F
8  | C -> D
9  | C -> E
10 | D -> A
11 | D -> B
12 | D -> C
13 | D -> F
14 | E -> A
15 | E -> C
16 | F -> A
17 | F -> B
18 | F -> D
```

После программа переходит к поиску в глубину для того, чтобы определить существует ли путь от вершины А в вершину Б, где А и Б случайно сгенерированные вершины.

1. Создается массив размером равным количеству вершин. Изначально все элементы равны 0. Данный массив будет показывать нам посетили ли мы ту или иную вершину или нет.
2. Генерируем начальную и конечную вершины.
3. Помещаем начальную вершину в стек.
4. Извлекаем верхний элемент из стека и отмечаем его как посещенный.
5. Проверяем для взятой вершины все смежные и не обнаруженные вершины. Если такие имеются, то добавляем ее в стек и помечаем как обнаруженную.
6. Повторяем с 4 пункта, пока стек не будет пустым.

```
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 = gr.ver - 1; j >= 0; j--)
    { // проверяем для нее все смежные вершины
        if (gr.matrix_smezh[node][j] == 1 && nodes[j] != 2)
        { // если вершина смежная и не обнаружена
            Stack.push( _Val: j); // добавляем ее в стек
            nodes[j] = 1; // отмечаем вершину как обнаруженную
            e.begin = node; e.end = j;
            Edges.push( _Val: e);
            if (node == req) break;
        }
    }
    cout << node + 1 << endl; // выводим номер вершины
}
```

```
Поиск в глубину (Проверка на существование пути)

2
5
3
4
1
2
6
5

Время (млсек): 32.4065
Путь до вершины 6
6 <- 2 <- 1 <- 4 <- 3
```

После выполнения данного алгоритма происходит проверка на существование пути из начальной вершины в конечную. Если такой существует, то программа переходит к выполнению поиска в ширину для определения кратчайшего пути.

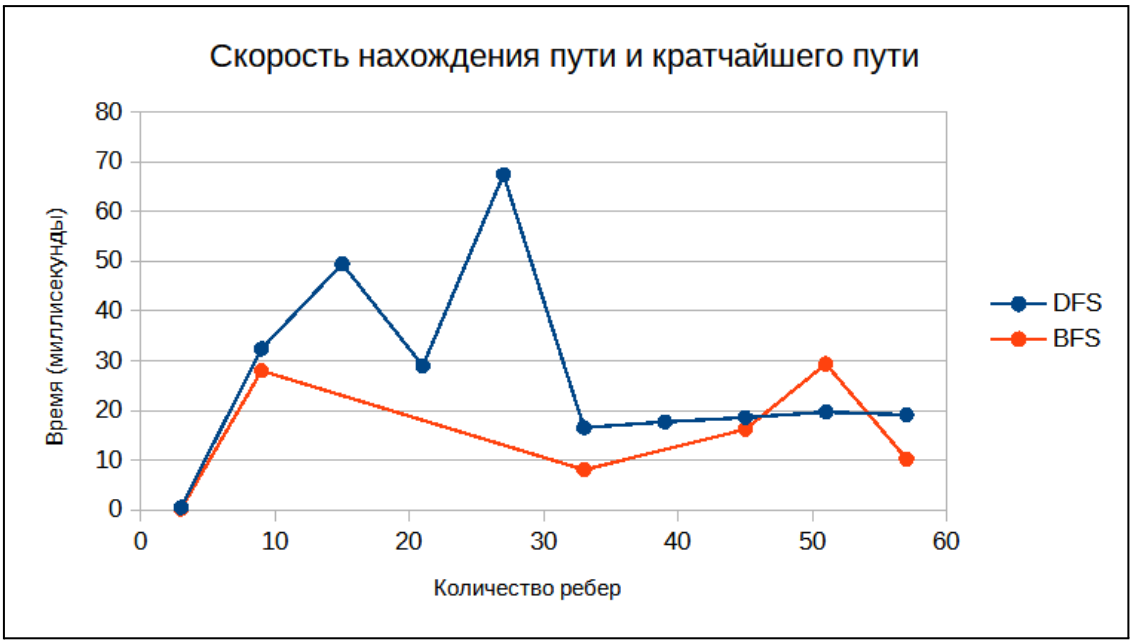
1. Создается массив размером равным количеству вершин. Изначально все элементы равны 0. Данный массив будет показывать нам посетили ли мы ту или иную вершину или нет.
2. Помещаем начальную вершину в очередь.
3. Извлекаем элемент из очереди и отмечаем его как посещенный.
4. Проверяем для взятой вершины все смежные и не обнаруженные вершины. Если такие имеются, то добавляем ее в стек и помечаем как обнаруженную.
5. Повторяем с 4 пункта, пока стек не будет пустым.

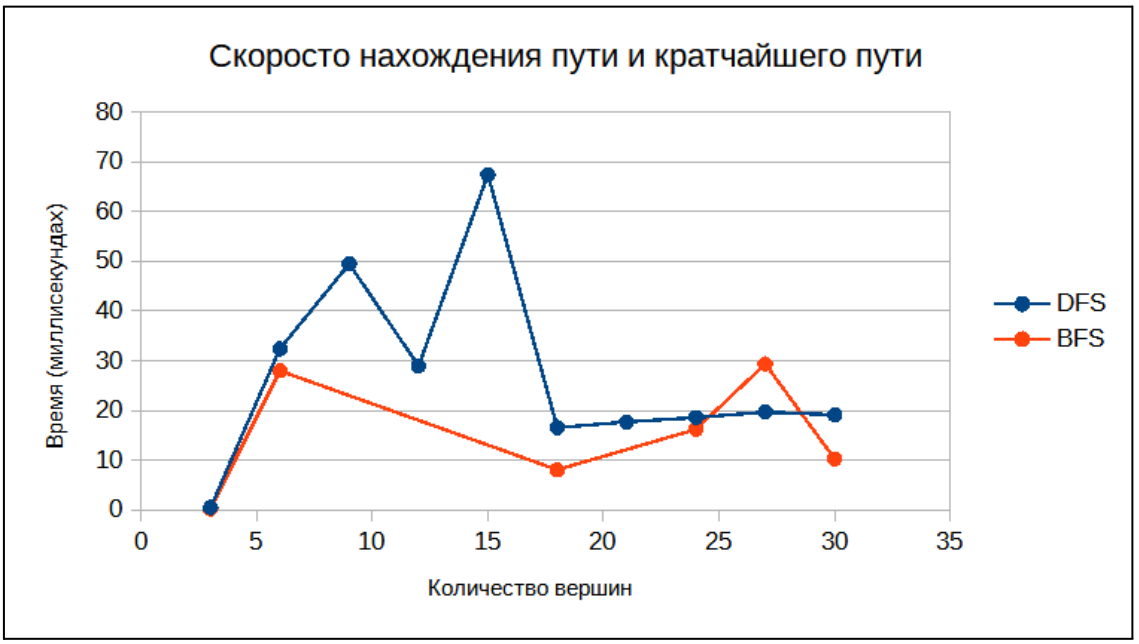
```
Queue.push(_Val: start); // помещаем в очередь первую вершину
while (!Queue.empty())
{
    int node = Queue.front(); // извлекаем вершину
    Queue.pop();
    nodes[node] = 2; // отмечаем ее как посещенную

    for (int j = 0; j < gr.ver; j++)
    {
        if (gr.matrix_smezh[node][j] == 1 && nodes[j] == 0)
        { // если вершина смежная и не обнаружена
            Queue.push(_Val: j); // добавляем ее в очередь
            nodes[j] = 1; // отмечаем вершину как обнаруженную
            e.begin = node; e.end = j;
            Edges.push(_Val: e);
            if (node == req) break;
        }
    }
    cout << endl << node + 1; // выводим номер вершины
}
```

```
Поиск в ширину (Нахождение кратчайшего пути)

3
4
5
1
2
6
Время (млсек): 27.9959

Путь до вершины 6
6 <- 4 <- 3
```

Для каждого поиска было замерено время работы в миллисекундах. Полученные результаты были зафиксированы и на их основе были построены графики зависимостей времени выполнения алгоритма от количества ребер и вершин.

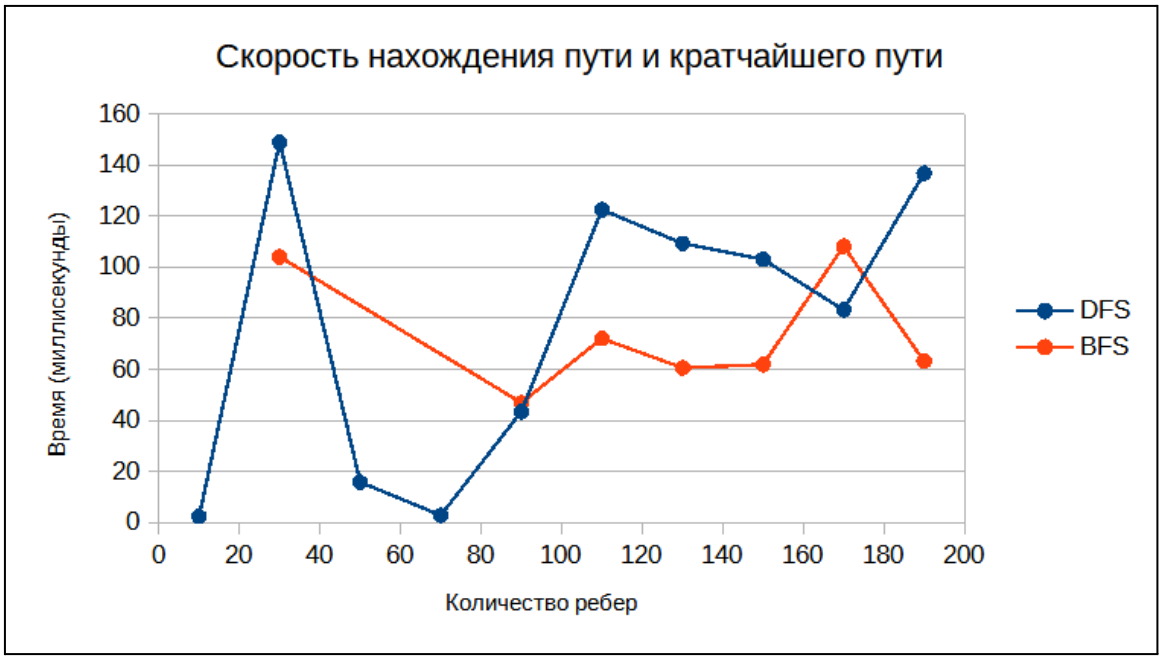
Тест	Вершины	Ребра	Направленный	DFS	BFS
1	3	3	1	0,547	0,2526
2	6	9	0	32,4065	27,9959
3	9	15	1	49,464	-
4	12	21	1	28,9579	-
5	15	27	1	67,4067	-
6	18	33	1	16,5792	8,1156
7	21	39	1	17,7336	-
8	24	45	1	18,6051	16,2327
9	27	51	1	19,7143	29,3735
10	30	57	1	19,1467	10,2899

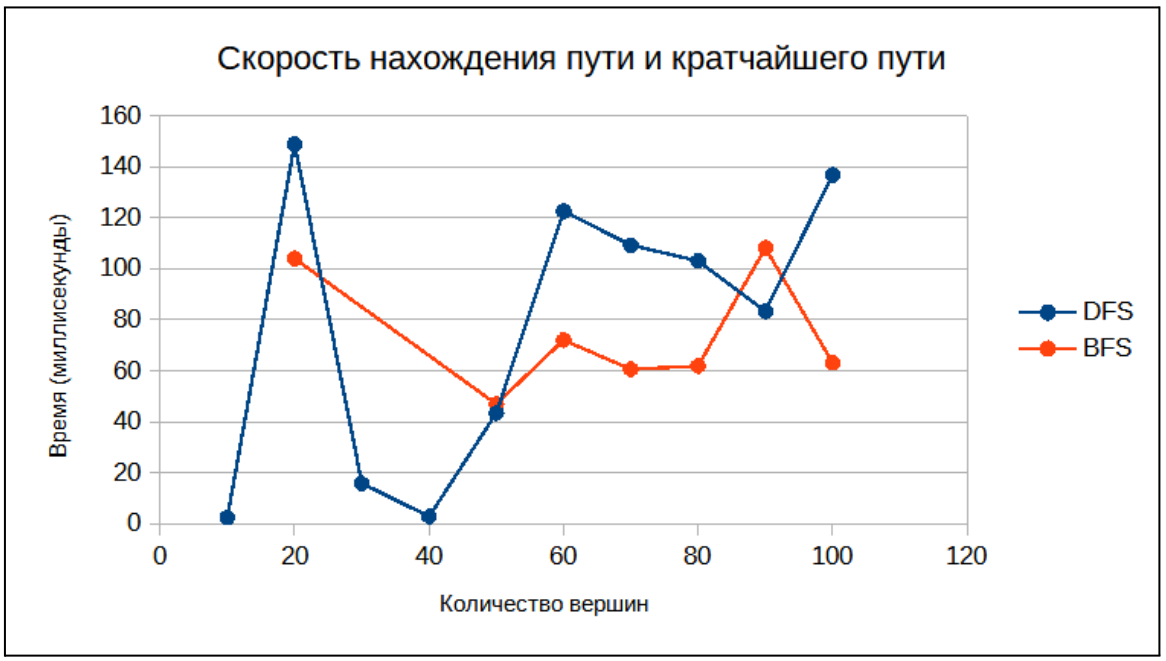




Также были проведены тесты для большего количества вершин и ребер. Аналогично был построен графики зависимостей времени выполнения алгоритмов от количества ребер и вершин.

Тест	Вершины	Ребра	Направленный	DFS	BFS
1	10	10	1	2,5105	-
2	20	30	0	148,906	103,993
3	30	50	1	15,8995	-
4	40	70	1	2,9103	-
5	50	90	1	43,4422	46,9431
6	60	110	0	122,463	72,0646
7	70	130	0	109,253	60,5822
8	80	150	0	103,043	61,8892
9	90	170	0	83,2822	108,151
10	100	190	1	136,705	63,1581

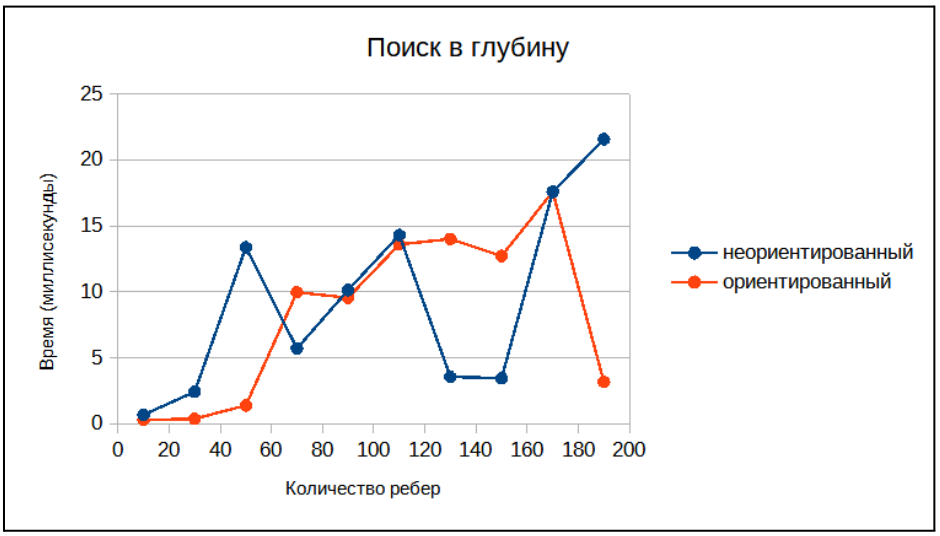




Также для сравнения были сгенерированы 10 направленных графов и 10 ненаправленных графов.

Тест	Вершины	Ребра	Направленный	DFS	BFS
1	10	10	0	0,6887	0,1122
2	20	30	0	2,4301	0,0771
3	30	50	0	13,3667	0,0523
4	40	70	0	5,7109	0,1311
5	50	90	0	10,1315	0,1773
6	60	110	0	14,3	0,1115
7	70	130	0	3,5577	0,252
8	80	150	0	3,465	0,3695
9	90	170	0	17,5705	0,205
10	100	190	0	21,5478	0,291

Тест	Вершины	Ребра	Направленный	DFS	BFS
1	10	10	1	0,2955	-
2	20	30	1	0,3863	-
3	30	50	1	1,3984	-
4	40	70	1	9,9566	-
5	50	90	1	9,5546	-
6	60	110	1	13,5863	-
7	70	130	1	14,0027	-
8	80	150	1	12,7066	0,1773
9	90	170	1	17,5776	0,2729
10	100	190	1	3,174	0,2049



Заключение.

Проанализировав полученные графики, можно заметить, что скорость нахождения пути из одной вершины в другую сильно зависит от количества вершин и ребер у графа. Видно, что чем больше у графа ребер, тем быстрее алгоритм находит нужный путь. Однако, если вершин слишком много, то алгоритму будет необходимо пройти их все, что увеличит время работы. При небольшом количестве ребер алгоритму будет также сложнее пройти все вершины, так как придется идти в обход. Отсюда следует, что большое количество ребер ускоряет процесс работы алгоритма.

Помимо этого, поиск в ширину осуществляется только в том случае, если путь из вершины в вершину существует. Скорость нахождения кратчайшего пути также зависит от количества ребер и вершин. Однако наибольшее влияние оказывает то, насколько далеки выбранные случайно вершины. Если они достаточно далеки, то алгоритму придется проходит несколько других вершин для нахождения кратчайшего пути. Тем самым время работы увеличится. Однако если ребер достаточно много, то путей будет много, что может привести как к ускорению, так и замедлению скорости алгоритма. Все зависит от случайно сгенерированных вершин и случайно сгенерированных путей между ними.

Также можно заметить, что для направленного графа в большинстве случаев алгоритм работает быстрее, так как путей перемещения между вершинами у него в два раза меньше, чем у ненаправленного графа. То есть вершин, которые мы можем посетить из начальной вершины, меньше, что и уменьшает время поиска пути.

Однако стоит заметить, что для неориентированного графа реже находится путь из одной вершины в другую. Это связано с тем, что ребра имеют направление и не позволяют идти по ним в обратную сторону, что лишает графа многих путей.