ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ
УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО»
Институт машиностроения, материалов и транспорта
Высшая школа автоматизации и робототехники

КУРСОВАЯ РАБОТА

Алгоритм «Обход графа в ширину и глубину (*BFS, DFS*)»
по дисциплине «Объектно-ориентированное программирование»

Выполнил
Студент
гр. 3331506/90401 _____ Ильясов А.Е.
(подпись)

Работу принял _____ Ананьевский М.С.
(подпись)

Санкт-Петербург
2022 г.

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.

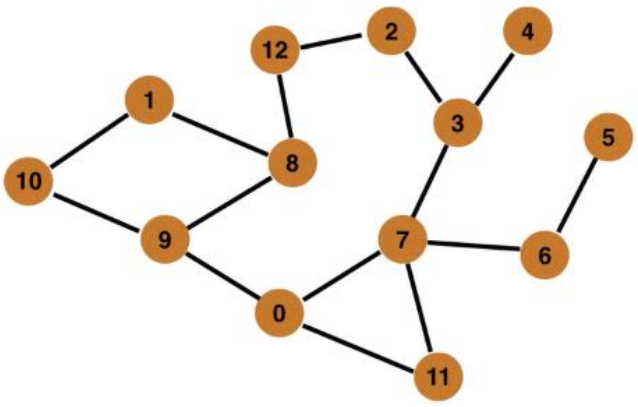


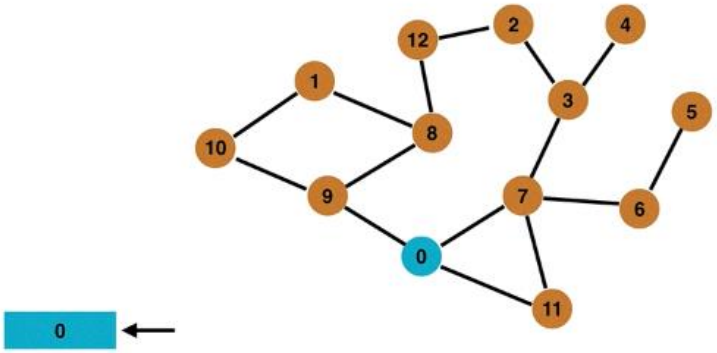
Рисунок 1. Граф для обхода

Каждая вершина может находиться в одном из 3 состояний:

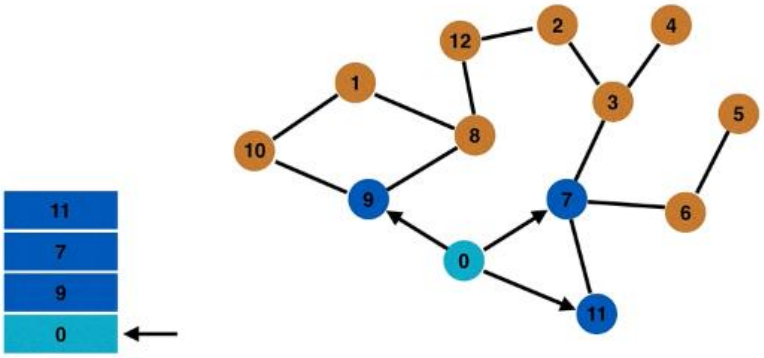
- 0 — коричневый – необнаруженная вершина;
- 1 — синий – обнаруженная, но не посещенная вершина;
- 2 — серый – обработанная вершина.

Голубой – рассматриваемая вершина.

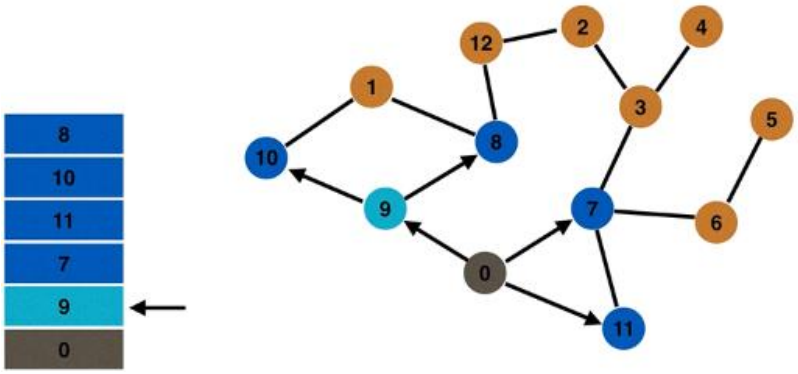
Шаг 1. Добавляем в очередь нулевую вершину.



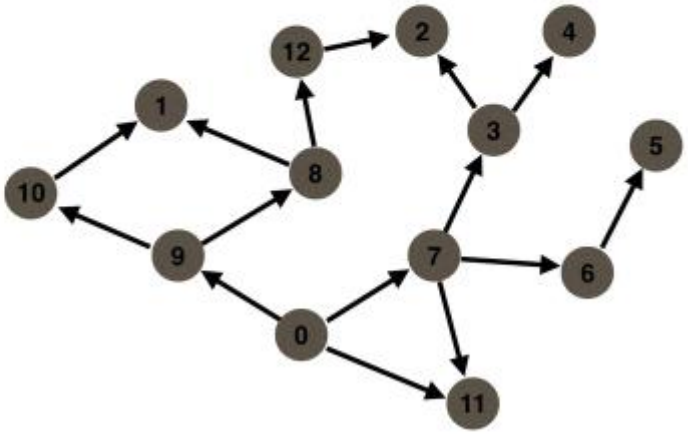
Шаг 2. Добавляем в очередь все вершины, смежные с нулевой вершиной.



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.

В качестве практического применения *BFS* был написан код для поиска всех вершин, лежащих на кратчайшем пути между заданной парой вершин. Код в приложении.

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. 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.

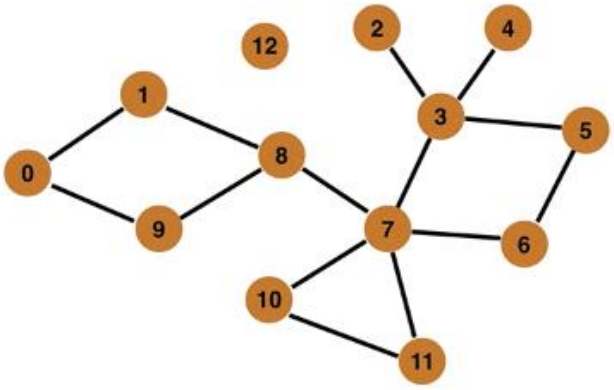


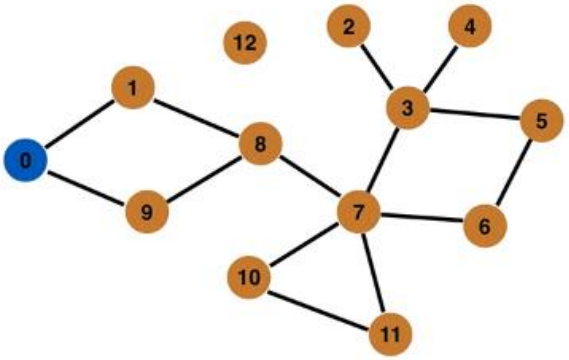
Рисунок 2. Граф для обхода

Каждая вершина может находиться в одном из 3 состояний:

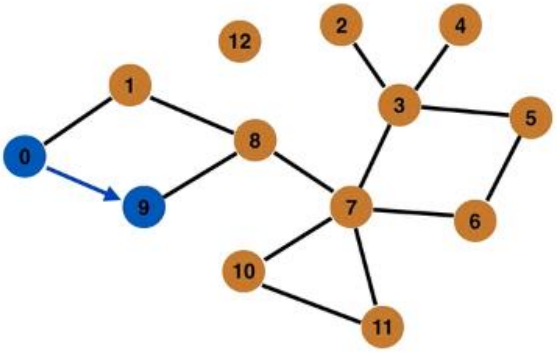
- 0 — коричневый – необнаруженная вершина;
- 1 — синий – обнаруженная, но не посещенная вершина;
- 2 — серый – обработанная вершина.

Голубой – рассматриваемая вершина.

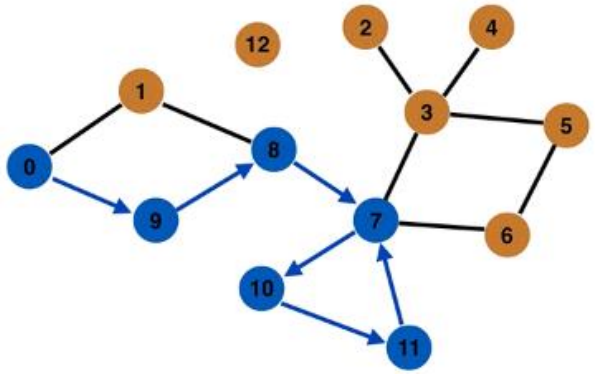
Шаг 1. Начинаем поиск с произвольной (нулевой) вершины.



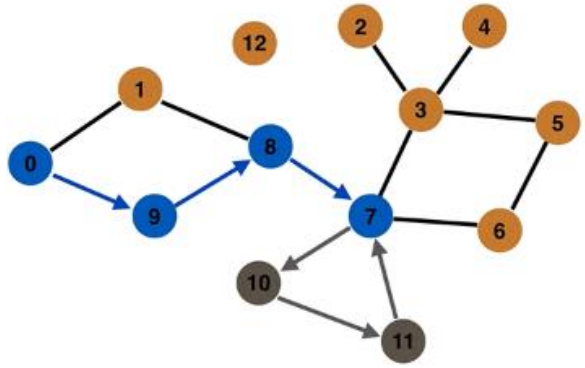
Шаг 2. Переходим к смежной ближайшей вершине.



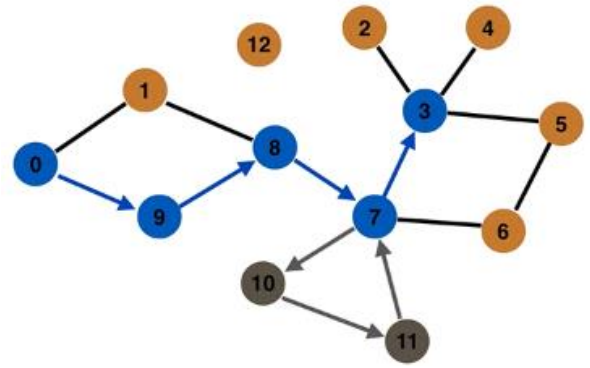
Шаг 3 – Шаг 6. Повторяем шаг 2 до тех пор, пока есть куда двигаться



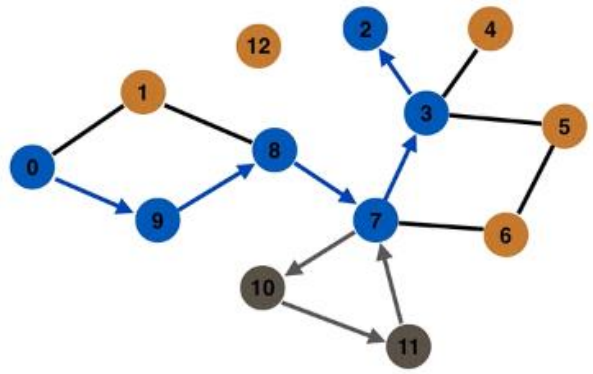
Шаг 7. Возвращаемся в ближайшую вершину с разветвлениями.



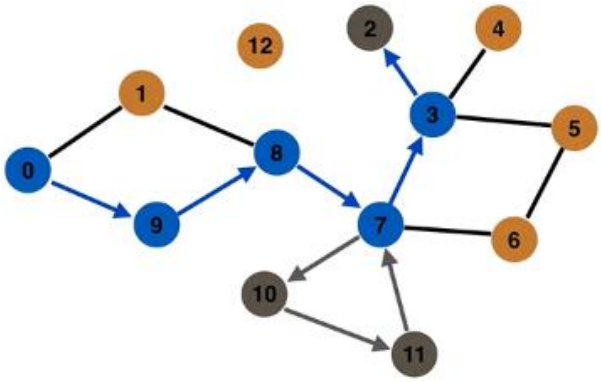
Шаг 8. Переходим к смежной ближайшей вершине (по другому пути).



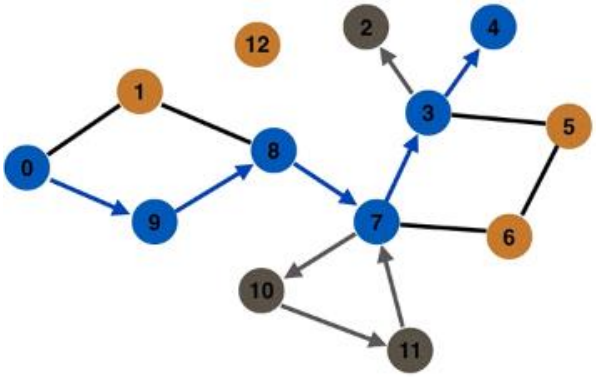
Шаг 9. Повторяем шаг 8 до тех пор, пока есть куда двигаться



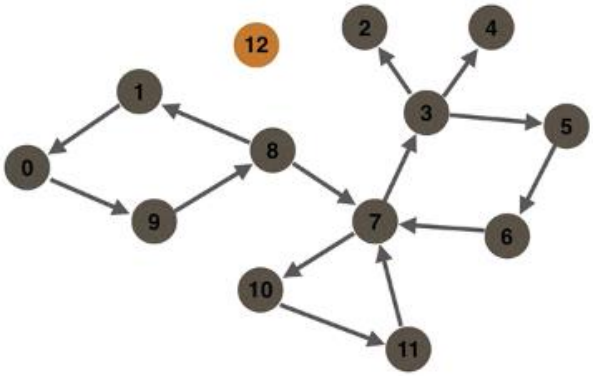
Шаг 10. Возвращаемся в ближайшую вершину с разветвлениями.



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.

В качестве практического применения *DFS* был написан код для топологической сортировки графа. Код в приложении.

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)
```

4. Исследование алгоритмов поиска в ширину и в глубину

Выполним оценку производительности алгоритмов *BFS* и *DFS* в зависимости от структур хранения данных и подключения оптимизации /O2.

В качестве структур данных будем рассматривать:

1. Матрица смежностей, заданная в виде:
   1.1 Непрерывного динамического массива
   1.2 Разрывного динамического массива
   1.3 Вектора векторов
2. Список смежностей, заданный в виде:
   2.1 Вектора списков
   2.2 Вектора векторов

Анализ затрат по времени для алгоритмов BFS, DFS.

Теоретическое время работы обхода в ширину (*BFS*).

Теоретическое время выполнения алгоритма *BFS*: O(V+E), пространственная сложность: O(V), где V — общее количество вершин. E — общее количество граней (ребер).

Теоретическое время работы обхода в глубину (DFS).

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). Пространственная сложность: O(V), где V — общее количество вершин. E — общее количество граней (ребер).

Анализ времени работы алгоритмов BFS, DFS для графа представленного в виде матрицы смежностей.

На рисунке 3 представлена зависимость алгоритмов BFS и DFS для непрерывного динамического массива при отключенной оптимизации O2.

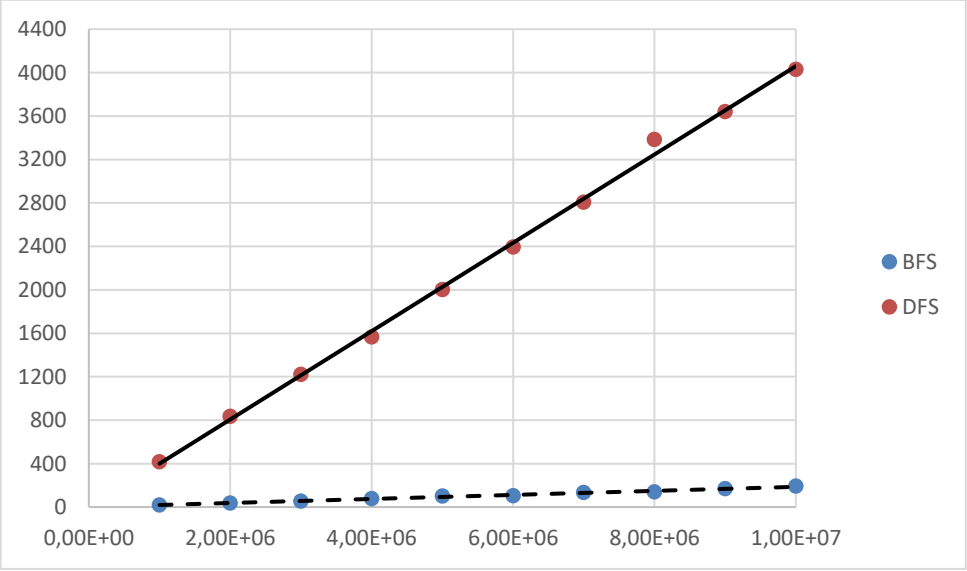


Рисунок 3. Непрерывный динамический массив без оптимизации

Как видно, алгоритмы BFS для данного случае производительнее DFS.

На рисунке 4 представлена зависимость алгоритмов BFS и DFS для разрывного динамического массива при отключенной оптимизации O2.

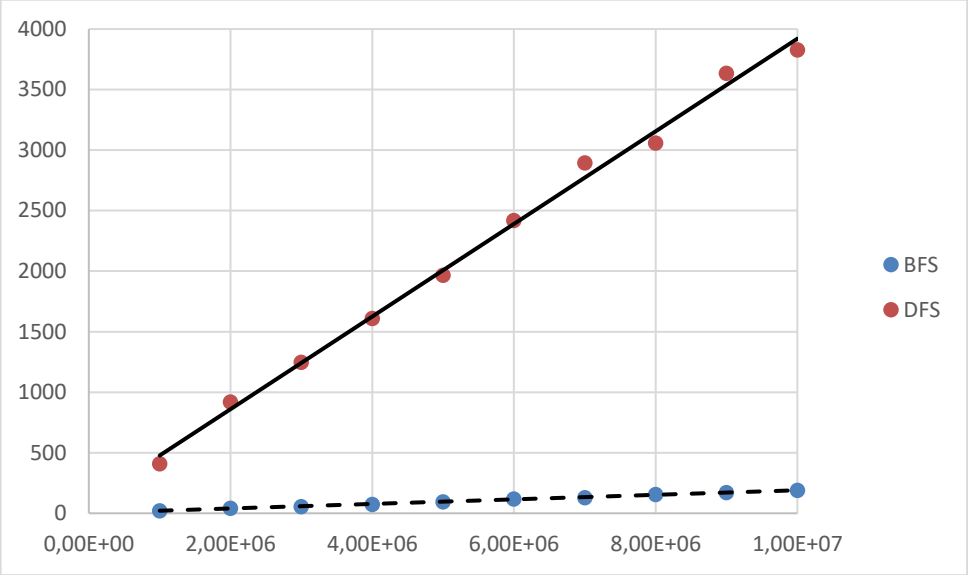


Рисунок 4. Разрывный динамический массив без оптимизации

В теории обработка разрывного динамического массива должно быть менее производительно, однако как видно из графика, мы имеем примерно ту же производительность. Объясняется это тем, что разница будет лишь на большом количестве входных данных.

Наконец, на рисунке 5 представлена зависимость алгоритмов BFS и DFS от (V+E) для вектора векторов при отключенной оптимизации O2.

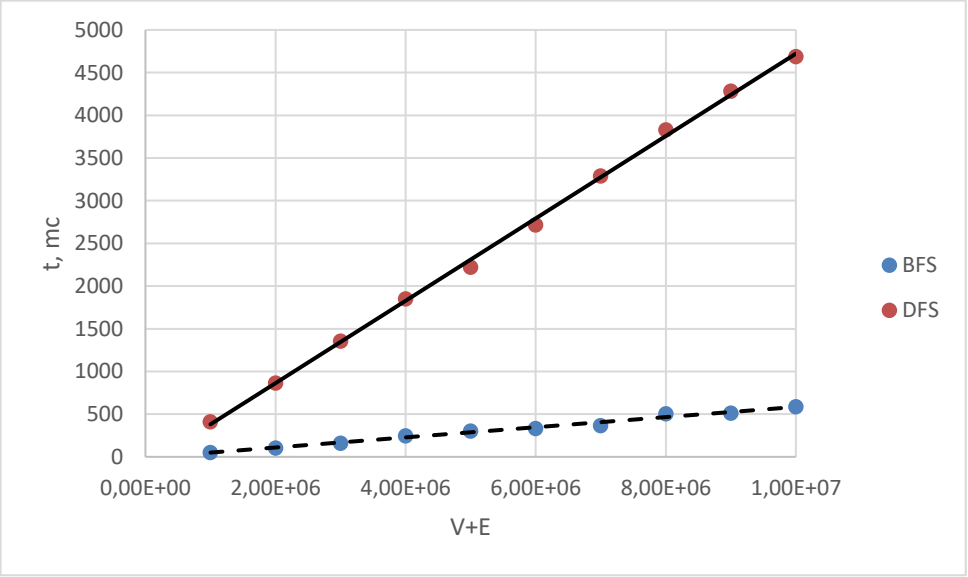


Рисунок 5. Вектор векторов без оптимизации

Как видно из графиков, алгоритмы BFS и DFS наименее производительны для вектора векторов.

Рассмотрим работу алгоритмов **при включенной оптимизации O2.**

На рисунке 7 представлена зависимость алгоритмов BFS и DFS для непрерывного динамического массива при включенной оптимизации O2.

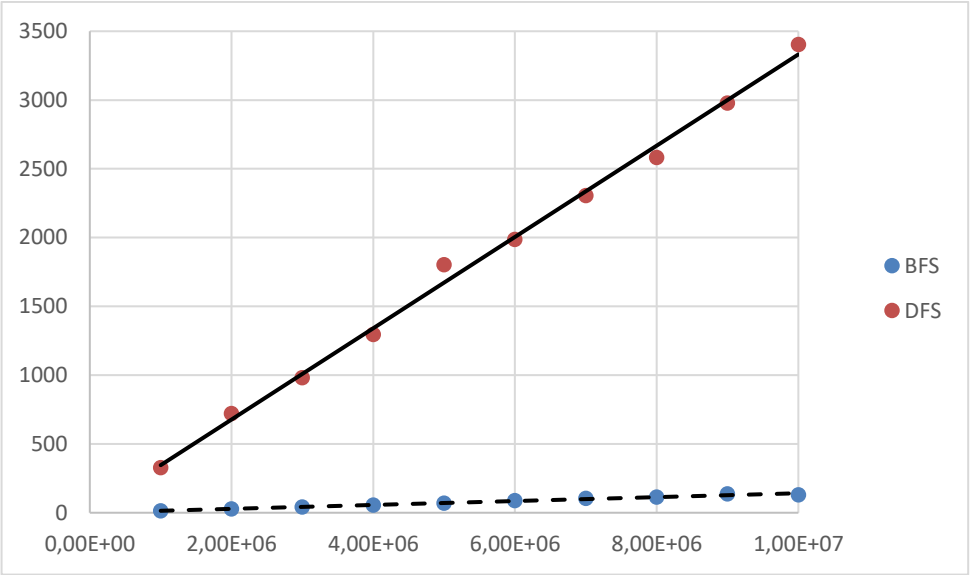


Рисунок 7. Непрерывный динамический массив с оптимизацией

На рисунке 8 представлена зависимость алгоритмов BFS и DFS для разрывного динамического массива при включенной оптимизации O2.

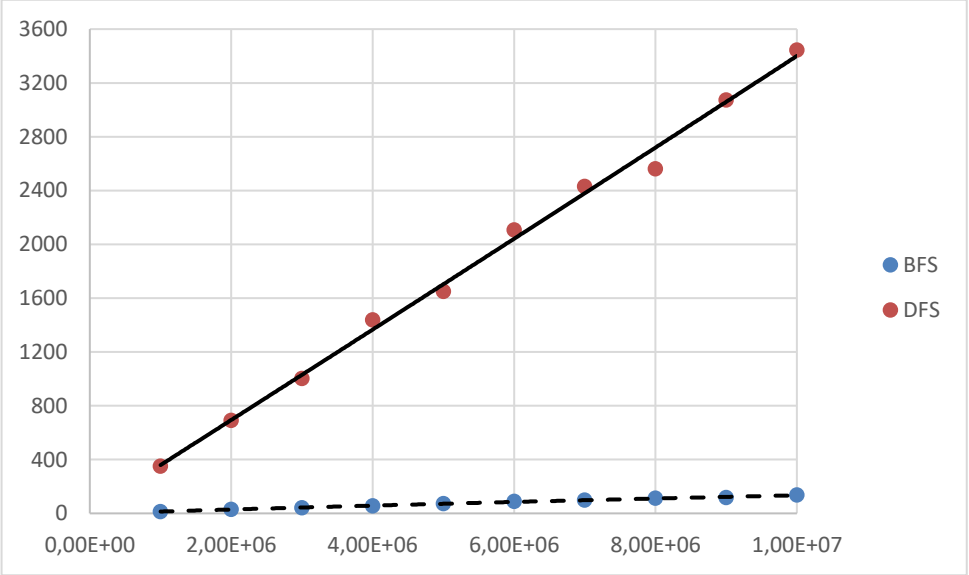


Рисунок 8. Разрывный динамический массив с оптимизацией

На рисунке 9 представлена зависимость алгоритмов BFS и DFS для вектора векторов при включенной оптимизации O2.

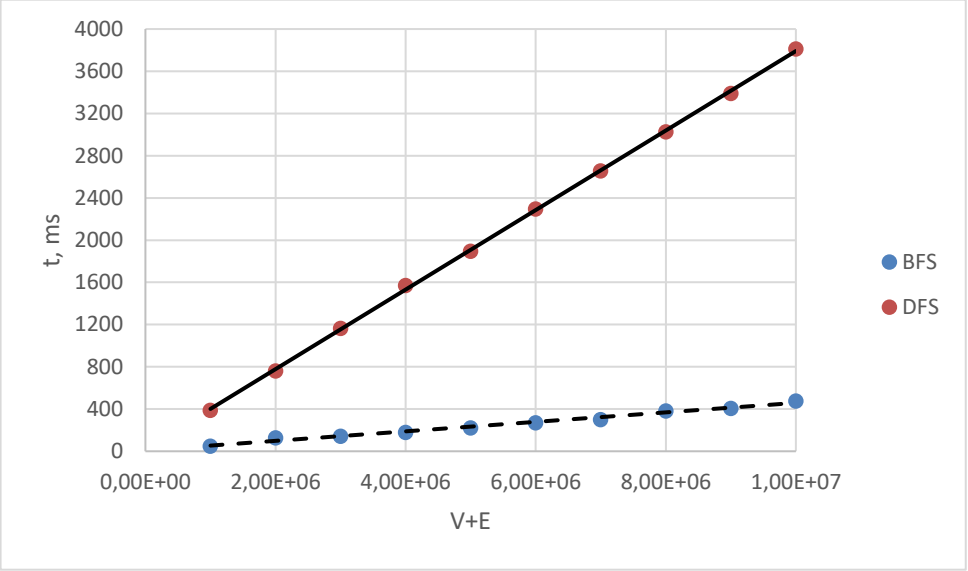


Рисунок 9. Вектор векторов с оптимизацией

Как видно из графиков, оптимизация O2 дает незначительный прирост производительности.

Анализ времени работы алгоритмов BFS, DFS для графа представленного в виде списка смежностей.

На рисунке 11 представлена зависимость алгоритмов BFS и DFS для вектора списков при отключенной оптимизации O2.

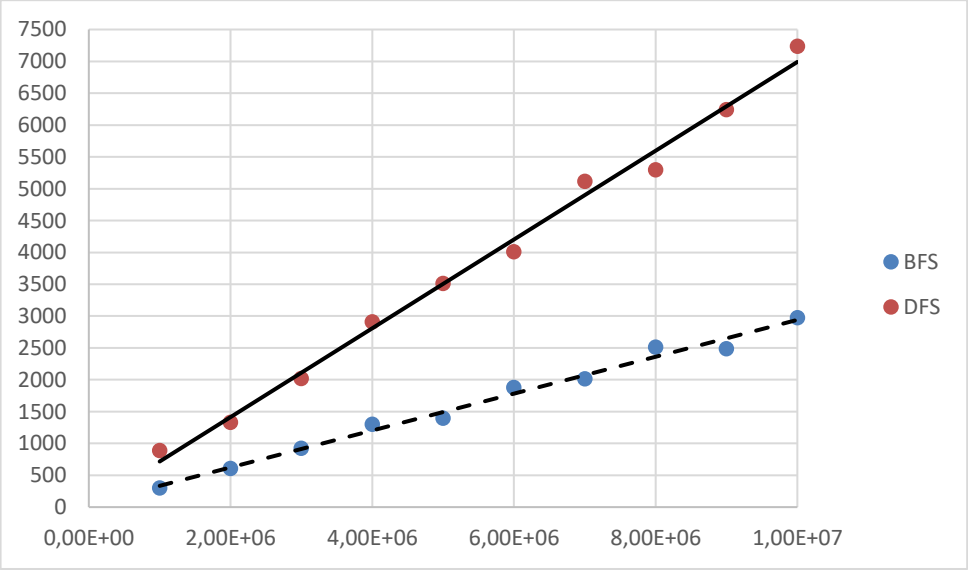


Рисунок 11. Анализ BFS, DFS без оптимизации

На рисунке 12 представлена зависимость алгоритмов BFS и DFS для вектора векторов при отключенной оптимизации O2.

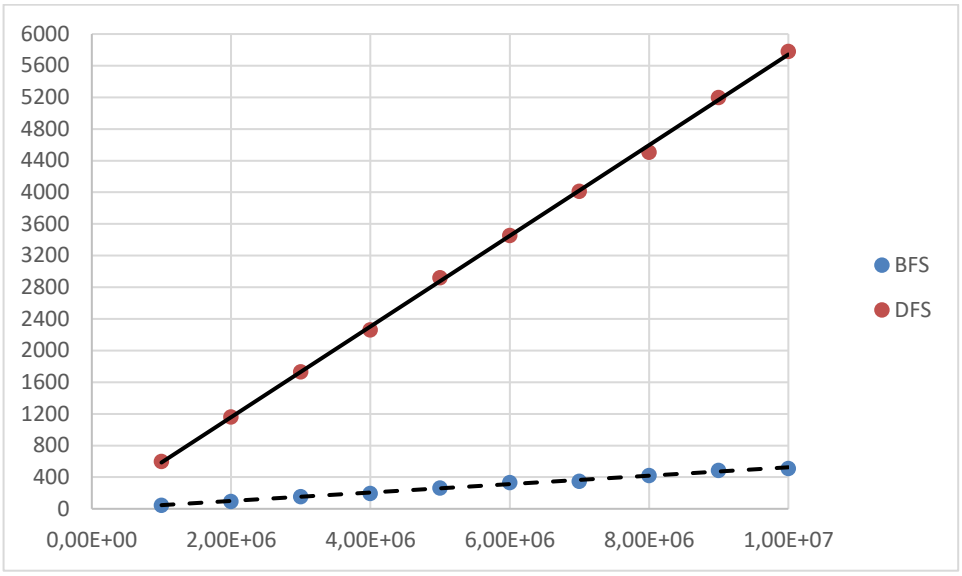


Рисунок 12. Анализ BFS, DFS без оптимизации

На рисунке 13 представлена зависимость алгоритмов BFS и DFS для вектора списков при **включенной** оптимизации O2.

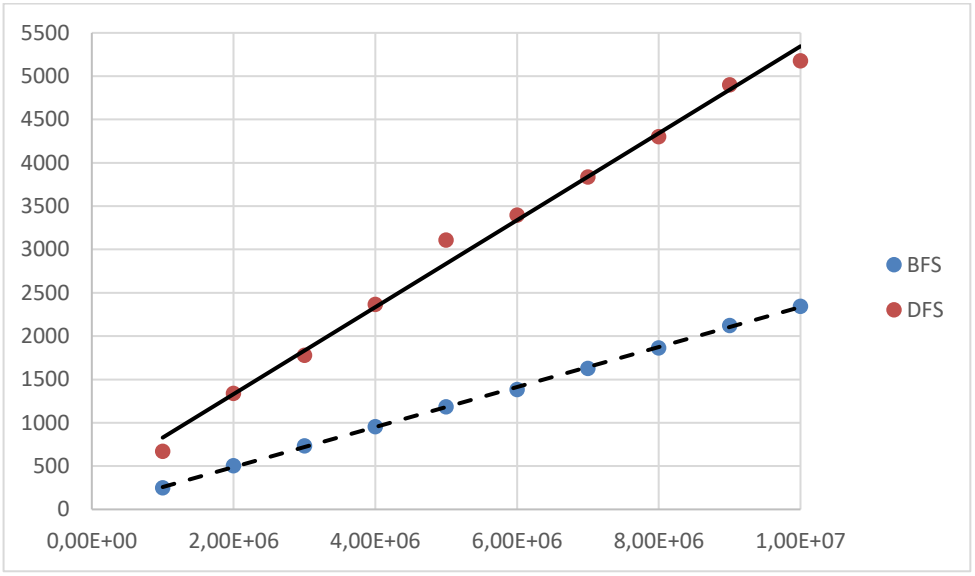


Рисунок 13. Анализ BFS, DFS с оптимизацией

На рисунке 14 представлена зависимость алгоритмов BFS и DFS для вектора векторов при **включенной** оптимизации O2.

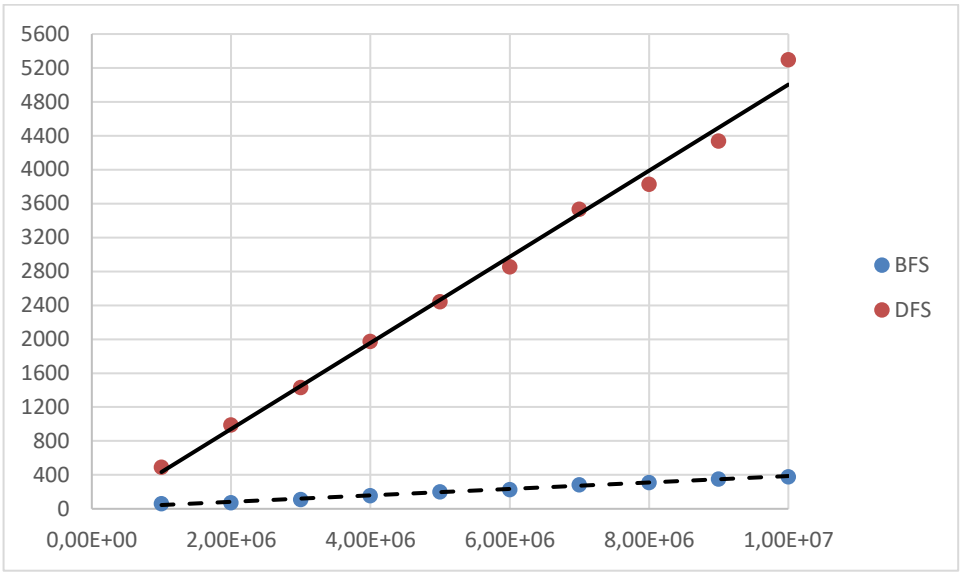


Рисунок 13. Анализ BFS, DFS с оптимизацией

Сравнение производительности работы алгоритмов BFS, DFS в зависимости от структуры данных и подключения оптимизации O2.

На рисунке 13 представлена скорость работы алгоритма BFS в зависимости от структуры данных при **отключенной** оптимизации O2.

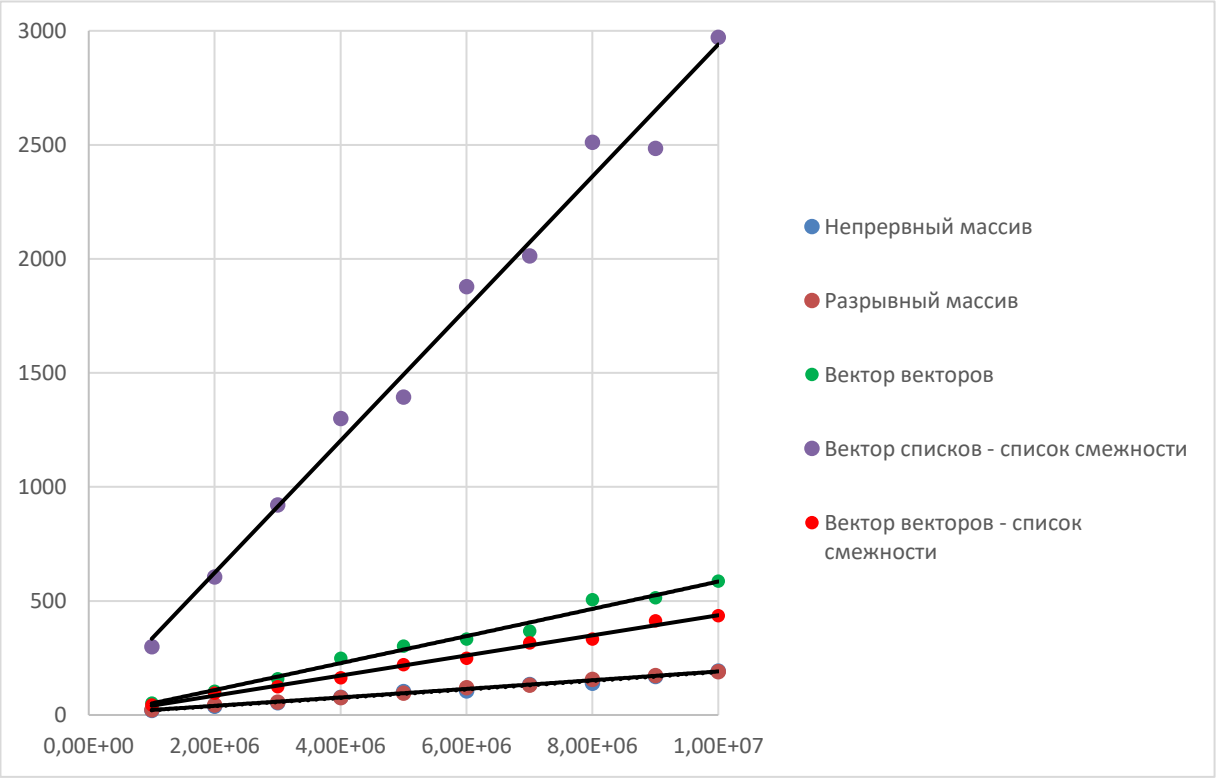


Рисунок 13. Анализ BFS без оптимизации

На рисунке 14 представлена скорость работы алгоритма BFS в зависимости от структур данных при **включенной** оптимизации O2.

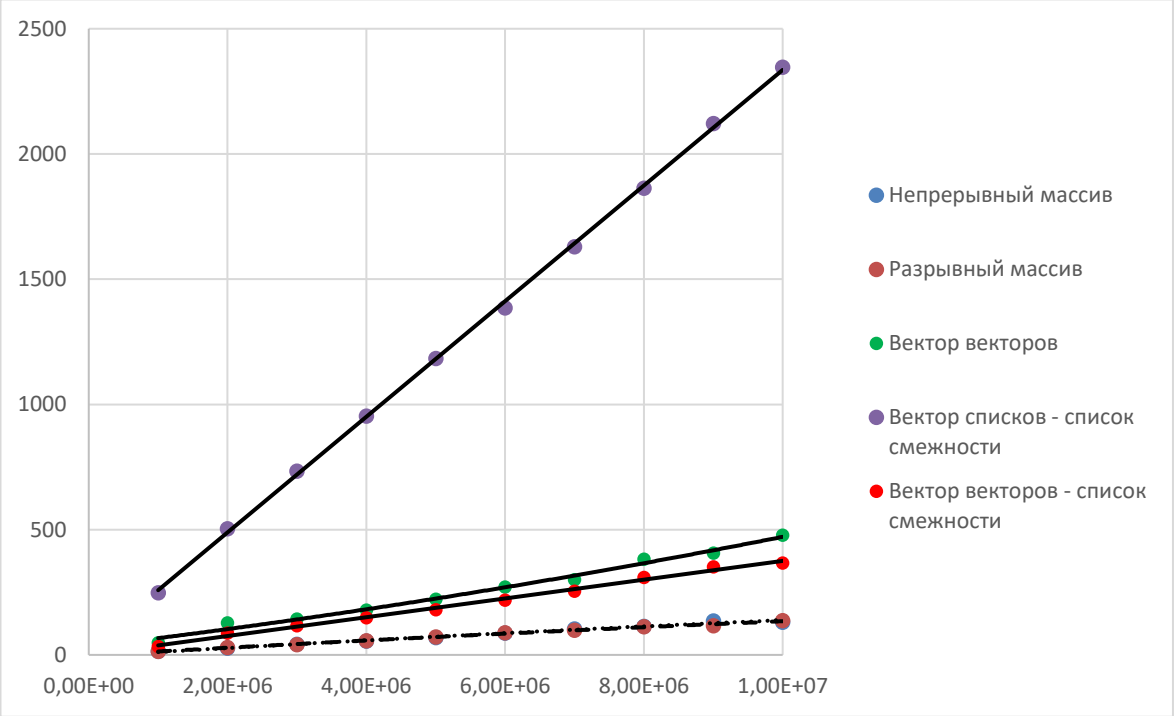


Рисунок 14. Анализ BFS с оптимизацией

На рисунке 15 представлена скорость работы алгоритма DFS в зависимости от структур данных при **отключенной** оптимизации O2.

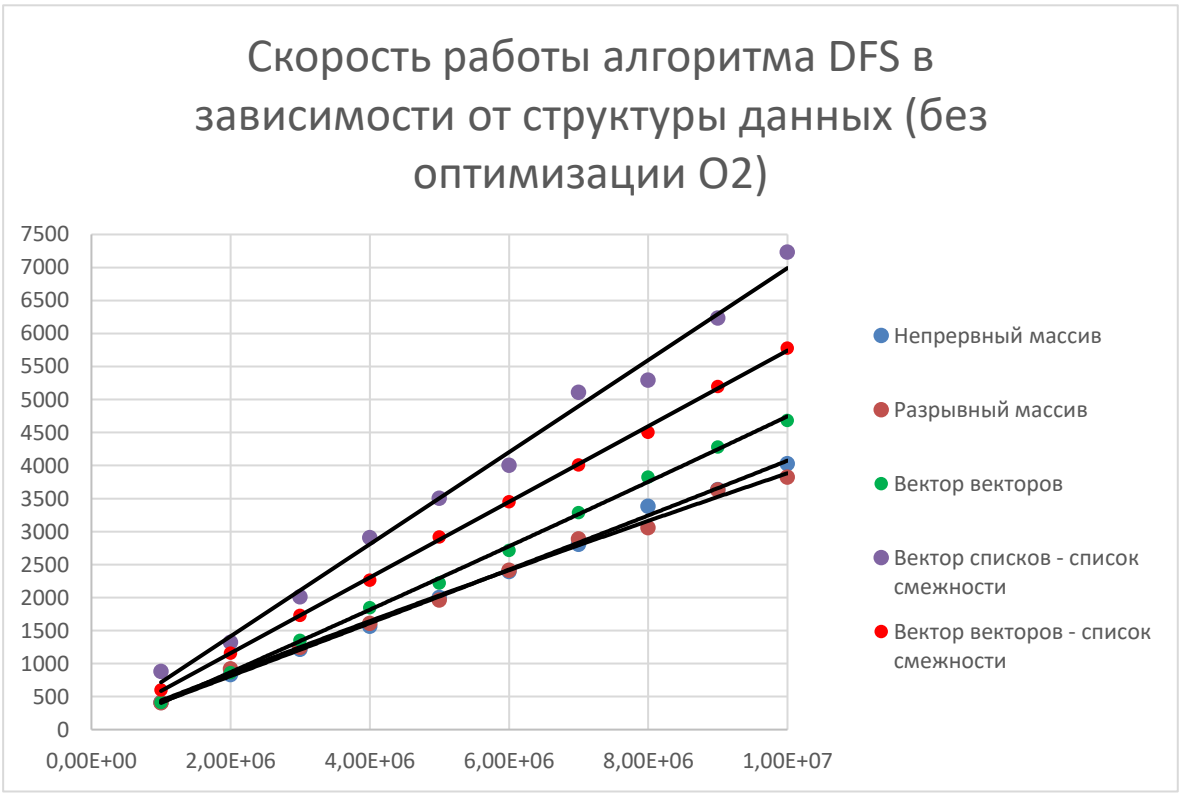


Рисунок 15. Анализ DFS без оптимизации

На рисунке 16 представлена скорость работы алгоритма DFS в зависимости от структур данных при **включенной** оптимизации O2.

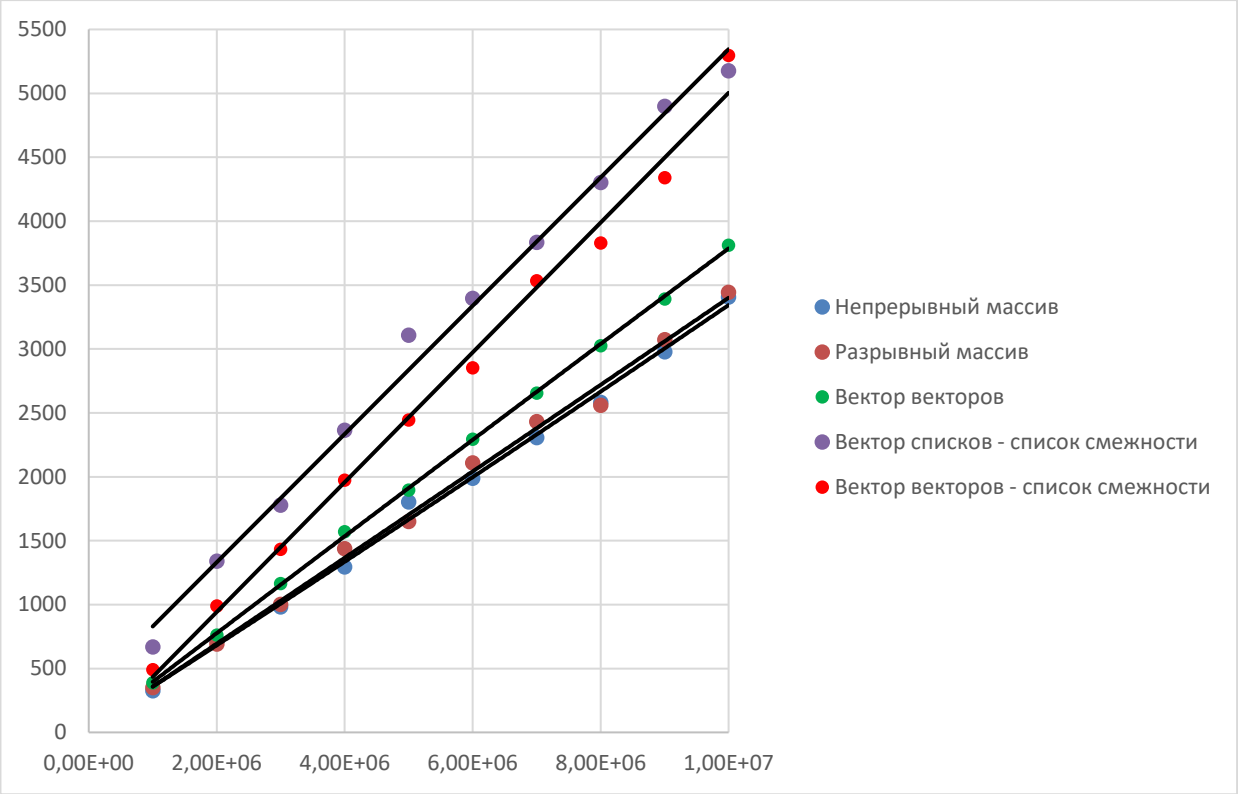


Рисунок 16. Анализ DFS с оптимизацией

Анализ затрат по памяти для алгоритмов BFS, DFS.

Алгоритмы *BFS, DFS* в представленной работе имеют схожую реализацию для различных структур данных. Следовательно, анализ затрат по памяти произведем единожды для одного случая, понимая, что такой анализ справедлив и для других структур данных.

Оценим затраты по памяти для алгоритма BFS.

Алгоритм использует:

1. Дополнительный массив (в представленной работе- это массив nodes[]) для идентификации вершин графа; Размер массива = кол-во элементов.
2. Очередь (в представленной работе - это Queue) для занесения туда

смежных вершин графа; Максимальный размер массива = кол-во элементов.

Следовательно, производительность по памяти O(V), где V – кол-во вершин.

Следует отметить, что в случае хранения графа в виде списка смежностей, мы достигаем рационального использования памяти, поскольку в строках списка смежностей хранятся лишь номера тех вершин, в которые ведут ребра из текущей. Однако, как видно из анализа производительности по времени, в случае плотного графа, скорость работы алгоритма меньше, чем если хранить граф в виде матрицы смежностей и зависит от способа хранения списка смежностей.

Оценим затраты по памяти для алгоритма DFS.

Алгоритм использует:

3. Дополнительный массив (в представленной работе- это массив nodes[]) для идентификации вершин графа; Размер массива = кол-во вершин графа.
4. Стек (в представленной работе - это Stack) для занесения туда смежных вершин графа; Максимальный размер массива

$$\sum_{i=1}^{n-1}(n-i)+1$$

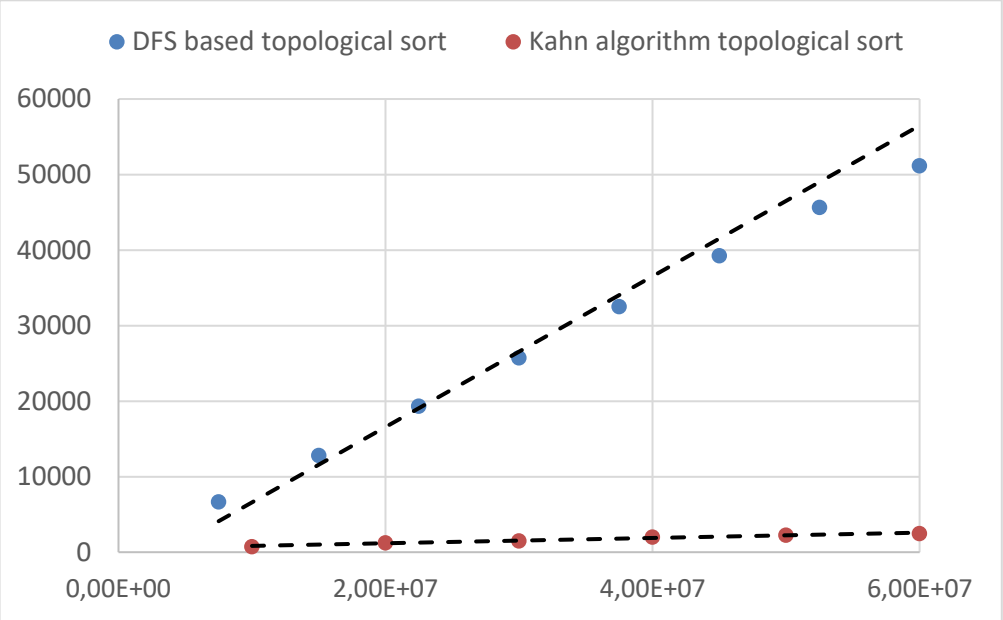
где $n-$ количество вершин графа

Следовательно, производительность по памяти при *текущей* реализации алгоритма *DFS* $O(V^2)$, где V – кол-во вершин.

Следует отметить, время работы алгоритмов BFS и DFS достаточно сильно разнятся во времени. Поэтому, чтобы получить более объективную оценку работы алгоритмов, следует провести замер времени работы алгоритмов не один раз.

5. 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. В случае реализации *DFS* был выбран нерекурсивный способ, однако, как выяснилось в ходе реализации, такой способ имеет пространственную сложность $O(V^2)$, что не очень хорошо. В том числе, для понимания практического применения, были написаны топологическая сортировка нецикличного графа на основе *DFS* и поиск кратчайшего пути между двумя вершинами невзвешенного графа на основе *BFS*. Следует отметить, производительность алгоритма топологической сортировки Кана по сравнению с топологической сортировкой на основе *DFS*.



Для реализации этих алгоритмов было выбран способ представления графа в виде списка смежности, хранящегося в векторе векторов (как оптимальный способ хранения и обработки графа). Также была оценена пространственная и временная сложности алгоритмов *BFS* и *DFS* (реализованных в данной работе) для графов разных размеров и структур данных. Был сделан вывод, что в случае обработки графа алгоритмами *BFS* и *DFS* оптимальный способ хранения графа – список смежностей, хранящийся в векторе векторов.

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://habr.com/ru/company/otus/blog/499138/

Приложение 1. Алгоритм *BFS*

```cpp
std::vector<int> Graph::bfs_search() {
    std::vector<int> bfs_search;
    std::vector<int> nodes(num_of_vertices, 0); // вершины графа (0 - все вершины не рассмотрены)
    std::queue<int> Queue; // очередь для хранения смежных вершин
    Queue.push(0); // помещаем в очередь первую вершину
    while (!Queue.empty()) { // пока очередь не пуста
        int node = Queue.front(); // извлекаем вершину
        Queue.pop(); // удаляем вершину
        nodes[node] = 2; // отмечаем ее как посещенную
        for (int i = 0; i < adjacency_list[node].size(); ++i) { // выводим все смежные вершины
            if (nodes[adjacency_list[node][i]] == 0) { // если вершина смежная и не обнаружена
                Queue.push(adjacency_list[node][i]); // добавляем ее в очередь
                nodes[adjacency_list[node][i]] = 1; // отмечаем ее как обнаруженную
            }
        }
        bfs_search.push_back(node);
    }
    return bfs_search;
}
```

Приложение 2. Алгоритм *DFS*

```cpp
std::vector<int> Graph::dfs_search() {
    std::vector<int> dfs_search;
    std::vector<int> nodes(num_of_vertices, 0); // вершины графа (0 - все вершины не рассмотрены)
    std::stack<int> Stack; //стек для хранения смежных вершин
    Stack.push(0); // помещаем в стек первую вершину
    while (!Stack.empty()) { //пока стек не пуст
        int node = Stack.top(); // извлекаем вершину
        Stack.pop(); // удаляем вершину
        if (nodes[node] == 2) continue; // если вершина была посещена, переходим к следующей
        nodes[node] = 2; // отмечаем ее как посещенную
        for (int i = adjacency_list[node].size() - 1; i >= 0; --i) { // проверяем для нее все смежные вершины
            if (nodes[adjacency_list[node][i]] != 2) { // если вершина смежная и не посещена
                Stack.push(adjacency_list[node][i]); // добавляем ее в стек
                nodes[adjacency_list[node][i]] = 1; // отмечаем вершину как обнаруженную
            }
        }
        dfs_search.push_back(node);
    }
    return dfs_search;
}
```

Приложение 3. Алгоритм поиска минимального пути на основе *BFS*

```cpp
std::vector<int> Graph::find_path(const int from, const int to) {
    if (from >= num_of_vertices || to >= num_of_vertices) { //ошибка, если такой вершины нет
        std::cout << "Find min path error. Index exceeds the number of vertices. Index must be not more " << num_of_vertices - 1 << "\n";
        return restored_path;
    }
    if (from == to) { //ошибка, если вход = выход
        std::cout << "Find min path error. Check entrance and exit.";
        return restored_path;
    }
    std::vector<int> nodes(num_of_vertices, 0); // вершины графа (0 - все вершины не рассмотрены)
    std::queue<int> Queue; //здесь хранятся все смежные вершины
    struct Edge { //структура для ребра
        int begin;
        int end;
    };
    Edge Edge_beetween_two_vertices; //ребро с двумя параметрами
    std::stack<Edge> Edges; //список ребер
    Queue.push(from); //заносим исходную вершину
    bool find = false; //изначально путь не найден
    while (!Queue.empty() && !find) { //пока есть смежные вершины и путь не найден
        int node = Queue.front();
        Queue.pop();
        nodes[node] = 2; //отмечаем вершину как посещенную
        for (int i = 0; i < adjacency_list[node].size(); ++i) { //перебираем смежные вершины для текущей
            if (nodes[adjacency_list[node][i]] == 0) { //если вершина не была посещена
                nodes[adjacency_list[node][i]] = 1; // отмечаем вершину как обнаруженную
                Queue.push(adjacency_list[node][i]); //заносим вершину в очередь
                Edge_beetween_two_vertices.begin = node; //заносим начало ребра
                Edge_beetween_two_vertices.end = adjacency_list[node][i]; //заносим конец ребра
                Edges.push(Edge_beetween_two_vertices); //заносим ребро в стек ребер
                if (adjacency_list[node][i] == to) { //если добрались до нужной вершины, останавливаем перебор вершин
                    find = true; //нашли требуемую вершину
                    break; //выходим из цикла
                }
            }
        }
    }
    //здесь путь собирается
    if (find) {
        int update_to = to;
        while (update_to != from && !Edges.empty()) { //перебираем все имеющиеся ребра
            Edge_beetween_two_vertices = Edges.top(); //извлекаем крайнее ребро
            Edges.pop(); //удаляем крайнее ребро
            if (Edge_beetween_two_vertices.end == update_to) { //если конец ребра ведет в искомую вершину
                update_to = Edge_beetween_two_vertices.begin; //теперь искомая вершина - начало этого ребра
                restored_path.push_back(Edge_beetween_two_vertices.end); //заносим конец ребра в вектор ответа
            }
        }
        restored_path.push_back(from); //заносим начало пути в вектор ребра
        std::reverse(restored_path.begin(), restored_path.end()); //реверс, чтобы путь шел от from к to
    }
    //вывод сообщения, если пути нет
    else {
        std::cout << "No path from " << from << " to " << to << ".";
    }
    return restored_path;
}
```

Приложение 4. Алгоритм топологической сортировки на основе *DFS*

```cpp
std::vector<int> Graph::topological_sort() {
    //!!!ЗДЕСЬ ТРЕБУЕТСЯ ПРОВЕРКА НА АЦИКЛИЧНОСТЬ ГРАФА!!!

    std::vector<int> nodes(num_of_vertices, 0); // вершины графа (0 - все вершины не рассмотрены)
    std::stack<int> Stack; // стек для хранения смежных вершин
    for (int i = 0; i < num_of_vertices; ++i) { // перебираем все вершины графа
        Stack.push(i); // помещаем в стек вершину
        while (!Stack.empty()) { //пока стек не пуст
            int node = Stack.top(); // извлекаем вершину
            if (nodes[node] == 2) { //если вершина посещена
                Stack.pop(); // удаляем вершину,
                continue; // переходим к следующей;
            }
            if (nodes[node] == 1) { // если вершина была обнаружена
                topological_sorted_graph.push_back(node);
                Stack.pop(); // удаляем вершину,
                nodes[node] = 2; //помечаем как посещенную
                continue; // переходим к следующей;
            }
            nodes[node] = 1; // отмечаем ее как обнаруженную
            bool has_edge = false; // изначально из текущей вершины нет ребер
            for (int i = adjacency_list[node].size() - 1; i >= 0; --i) { // проверяем для нее все смежные вершины
                if (nodes[adjacency_list[node][i]] != 2) { // если вершина смежная и не посещена
                    Stack.push(adjacency_list[node][i]); // добавляем ее в стек
                    has_edge = true; // есть ребро
                }
            }
            if (!has_edge) { // если из текущей вершины нет ребра
                topological_sorted_graph.push_back(node);
                Stack.pop(); // удаляем вершину
                nodes[node] = 2; //помечаем как посещенную
            }
        }
    }
    std::reverse(topological_sorted_graph.begin(), topological_sorted_graph.end());
    return topological_sorted_graph;
}
```