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.

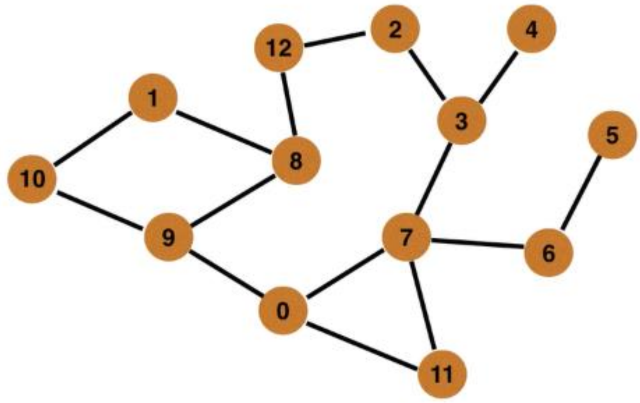


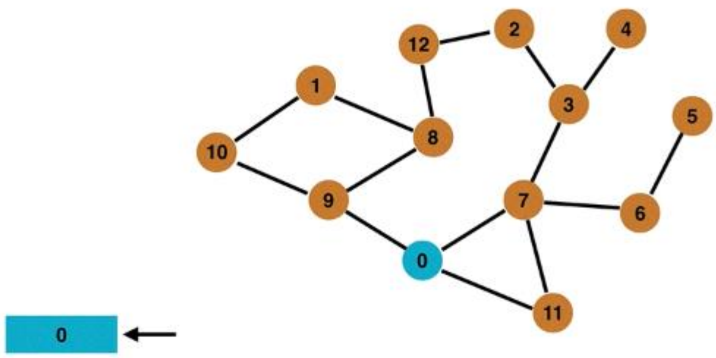
Рисунок 1. Граф для обхода

Каждая вершина может находиться в одном из 3 состояний:

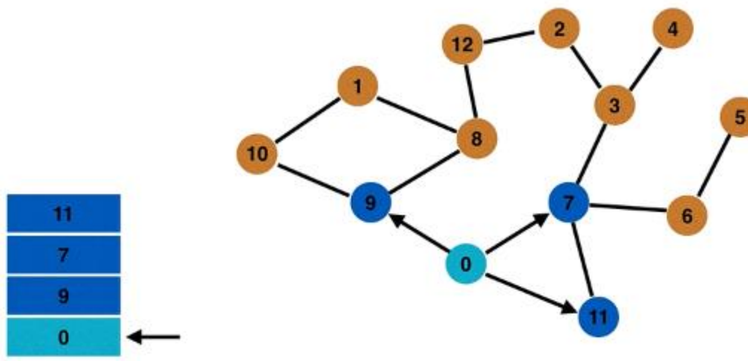
- 0 — коричневый – необнаруженная вершина;
- 1 — синий – обнаруженная, но не посещенная вершина;
- 2 — серый – обработанная вершина.

Голубой – рассматриваемая вершина.

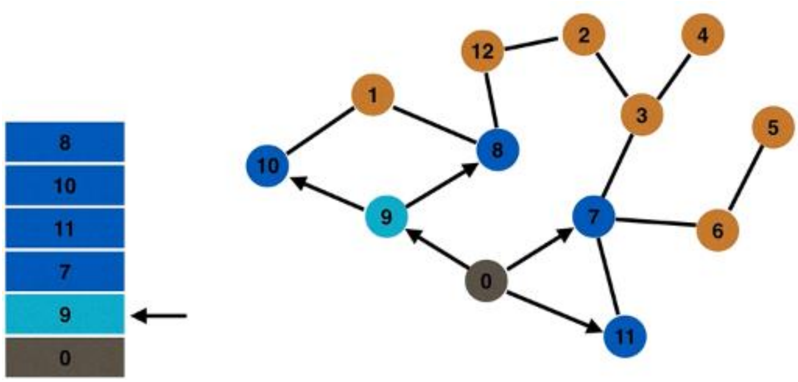
Шаг 1. Добавляем в очередь нулевую вершину.



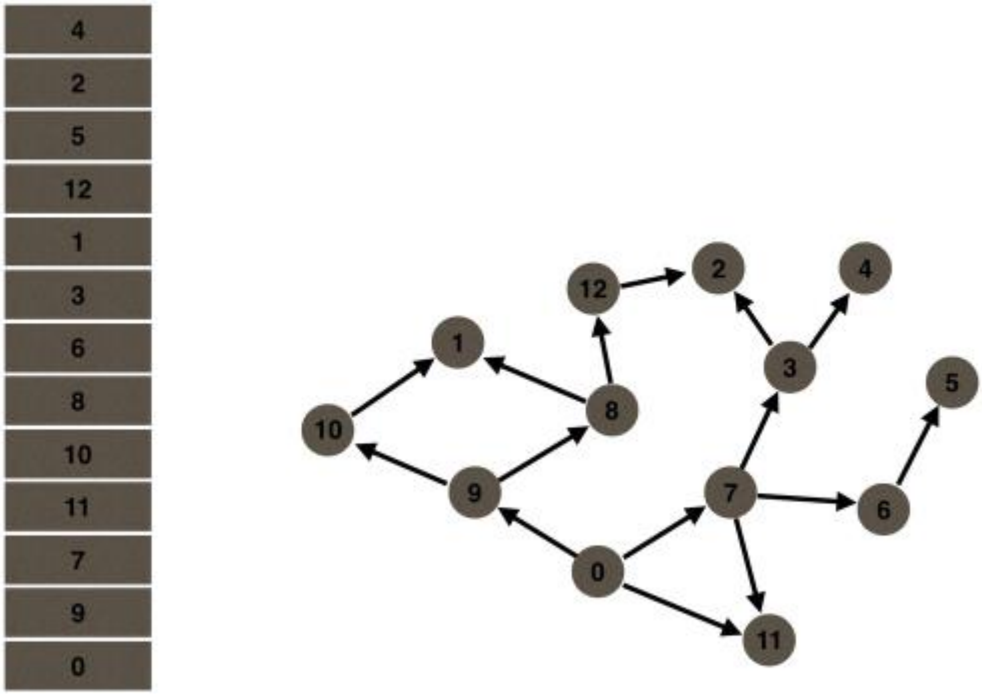
Шаг 2. Добавляем в очередь все вершины, смежные с нулевой вершиной.



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.

Реализация поиска всех вершин, лежащих на кратчайшем пути между заданной парой вершин 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.

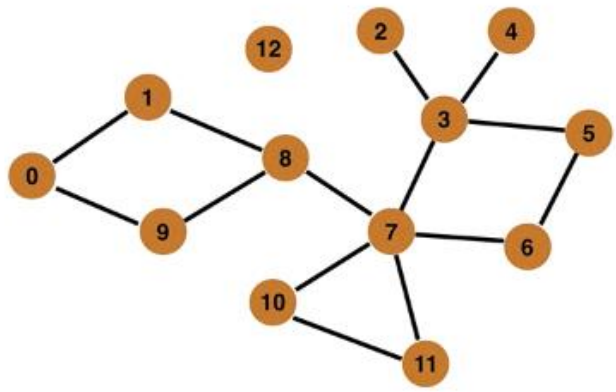


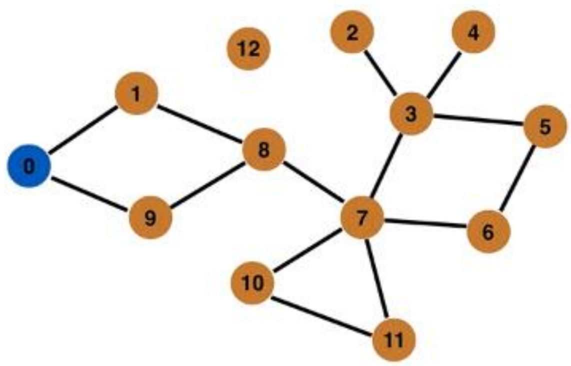
Рисунок 2. Граф для обхода

Каждая вершина может находиться в одном из 3 состояний:

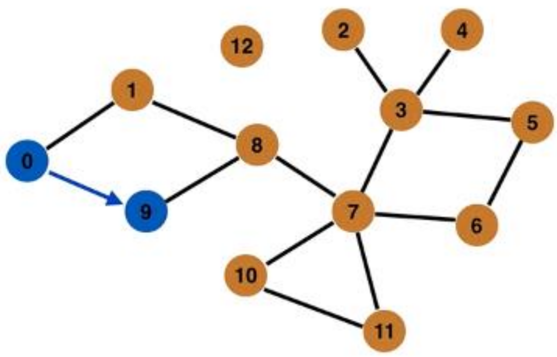
- 0 — коричневый – необнаруженная вершина;
- 1 — синий – обнаруженная, но не посещенная вершина;
- 2 — серый – обработанная вершина.

Голубой – рассматриваемая вершина.

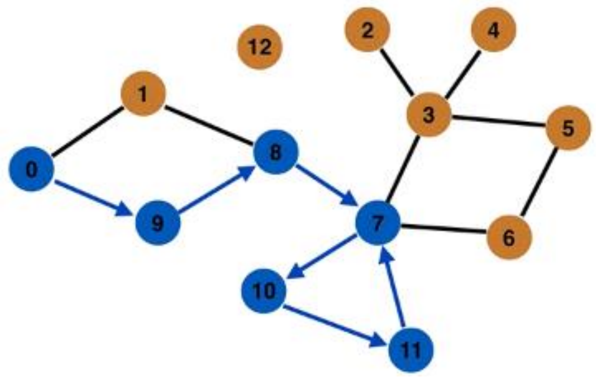
Шаг 1. Начинаем поиск с произвольной (нулевой) вершины.



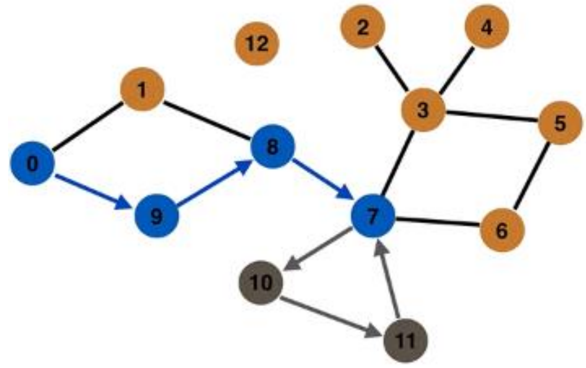
Шаг 2. Переходим к смежной ближайшей вершине.



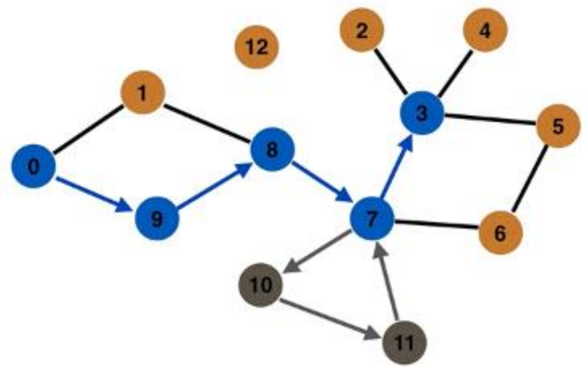
Шаг 3 – Шаг 6. Повторяем шаг 2 до тех пор, пока есть куда двигаться



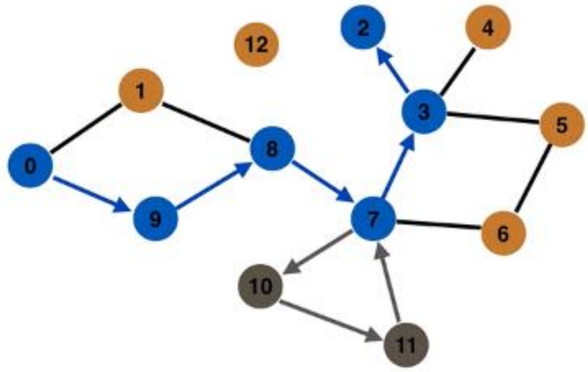
Шаг 7. Возвращаемся в ближайшую вершину с разветвлениями.



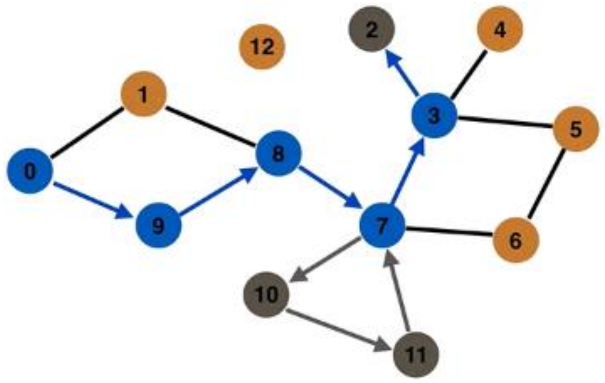
Шаг 8. Переходим к смежной ближайшей вершине (по другому пути).



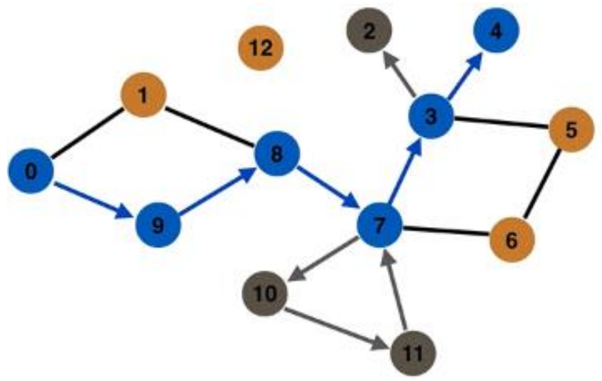
Шаг 9. Повторяем шаг 8 до тех пор, пока есть куда двигаться



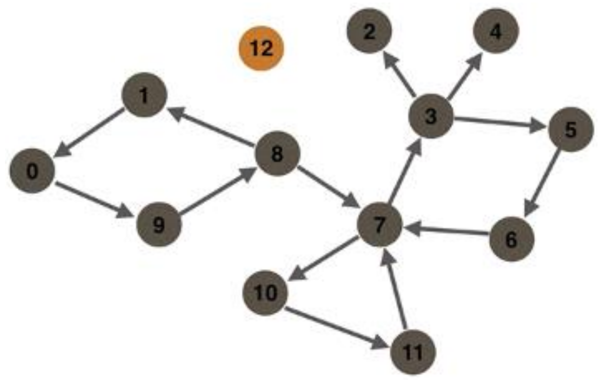
Шаг 10. Возвращаемся в ближайшую вершину с разветвлениями.



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.

Реализация топологической сортировки с применением DFS 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. Непрерывный динамический массив
2. Разрывный динамический массив
3. Вектор векторов

<u>Оценим время работы обхода в ширину (BFS).</u>

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

<u>Оценим время работы обхода в глубину (DFS).</u>

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

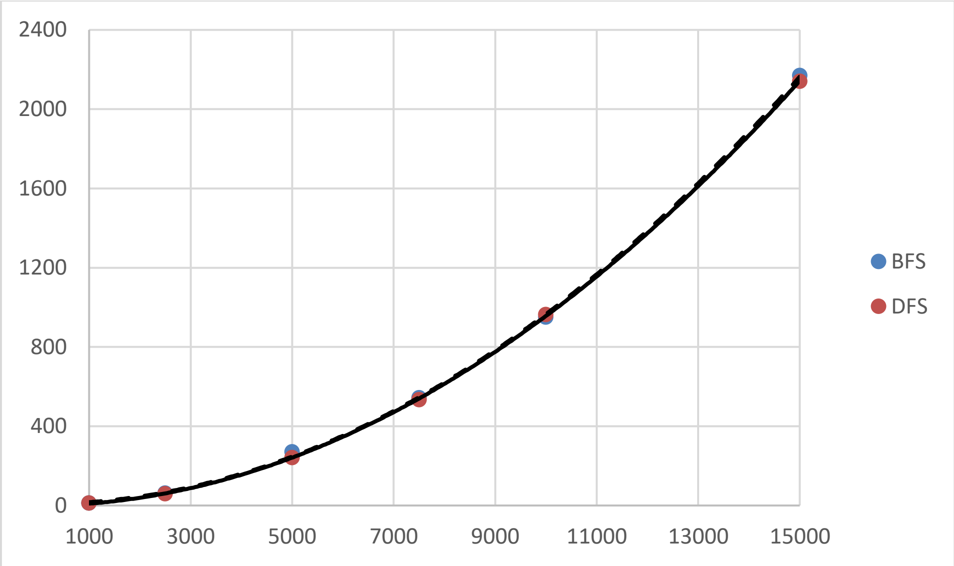


Рисунок 3. Непрерывный динамический массив без оптимизации

Как видно, алгоритмы BFS и DFS имеют приблизительно схожую производительность.

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

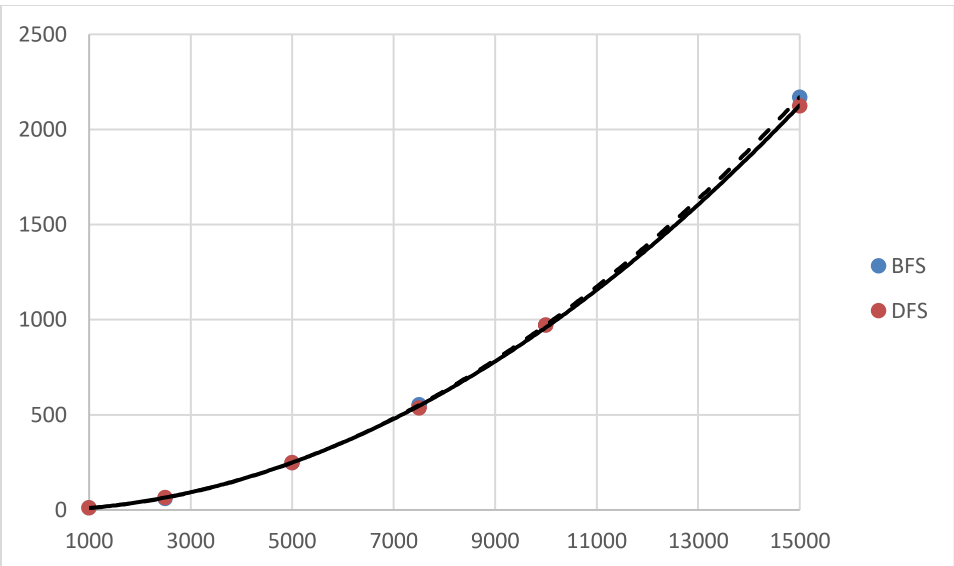


Рисунок 4. Разрывный динамический массив без оптимизации

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 для вектора векторов при отключенной оптимизации O2.

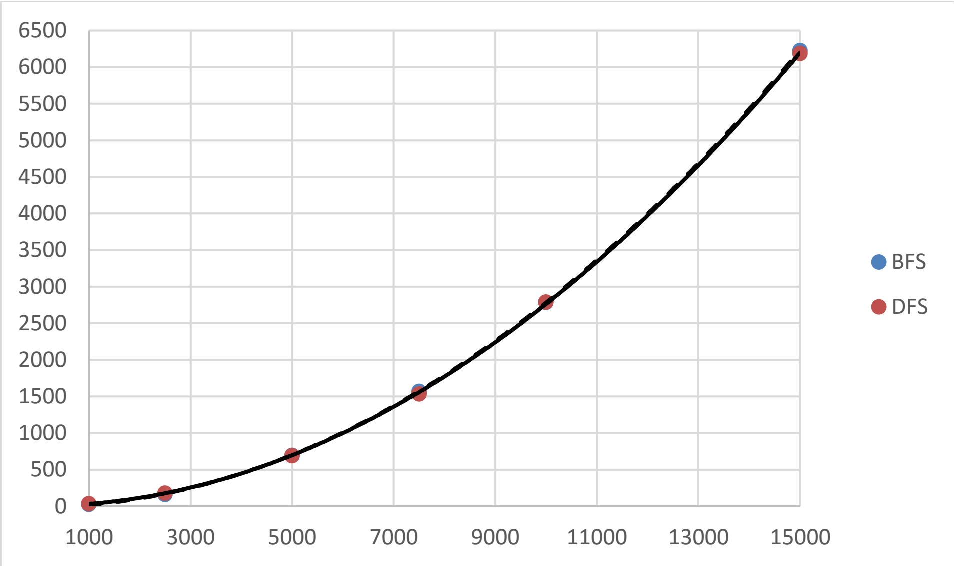
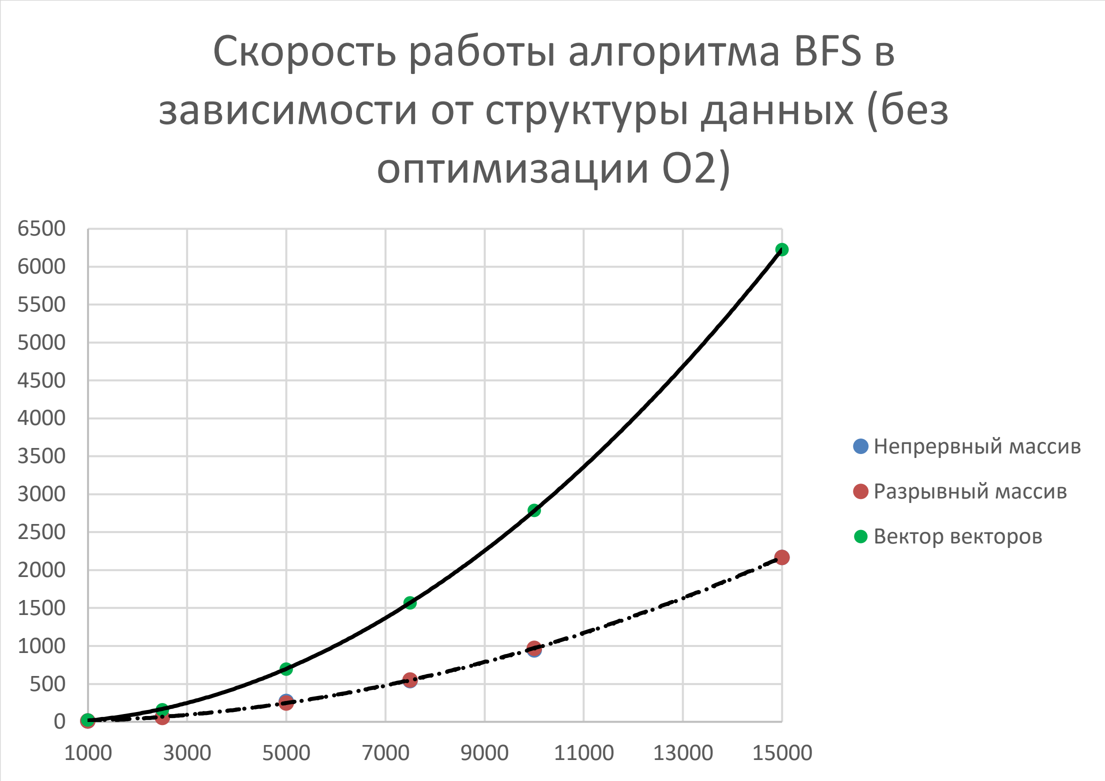


Рисунок 5. Вектор векторов без оптимизации

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

Поскольку алгоритмы BFS и DFS имеют схожую производительность, наложим графики и исследуем производительность алгоритмов в зависимости от структуры данных при выключенной оптимизации.



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

На рисунке 6 представлена зависимость алгоритмов BFS и DFS для непрерывного динамического массива при включенной оптимизации O2.

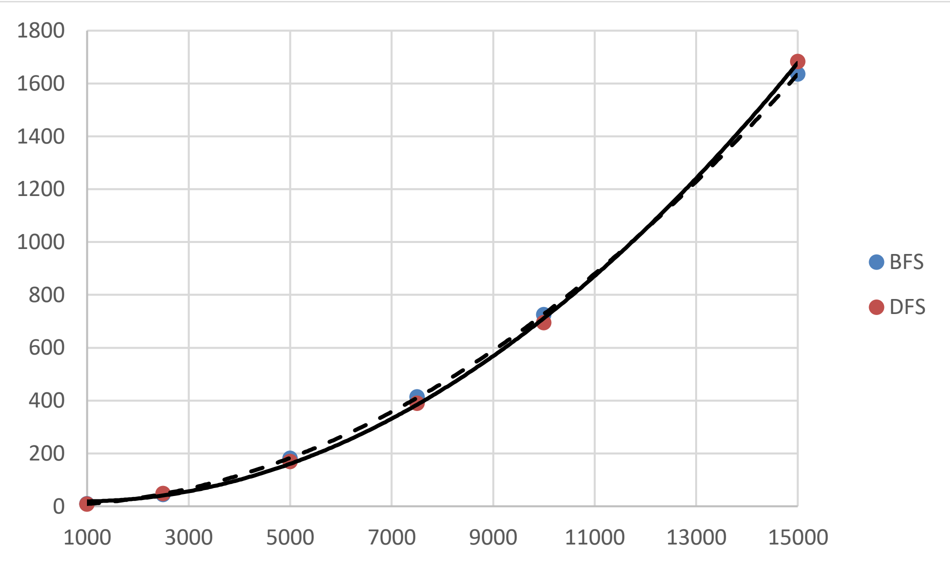


Рисунок 6. Непрерывный динамический массив с оптимизацией

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

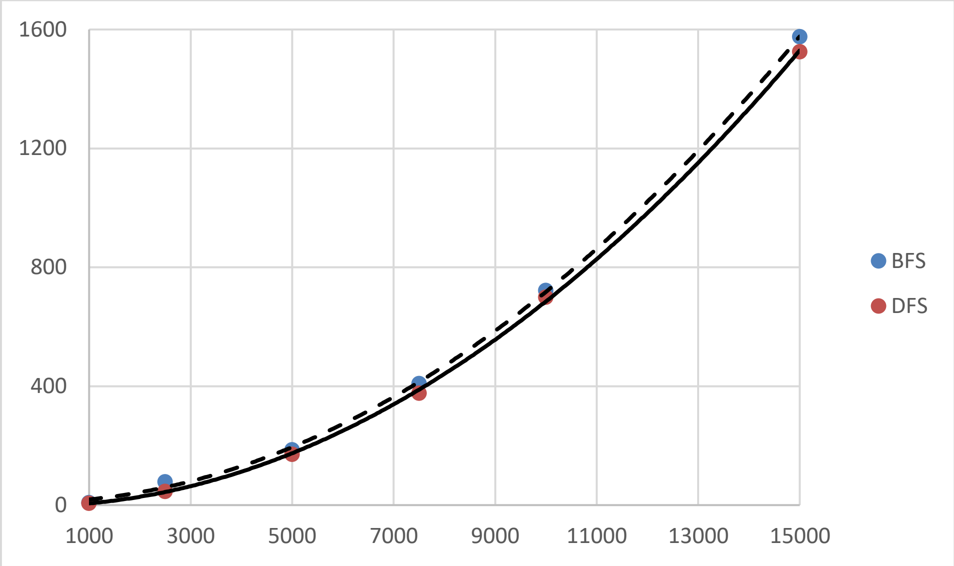


Рисунок 7. Разрывный динамический массив с оптимизацией

На рисунке 5 представлена зависимость алгоритмов BFS и DFS для вектора векторов при включенной оптимизации O2.

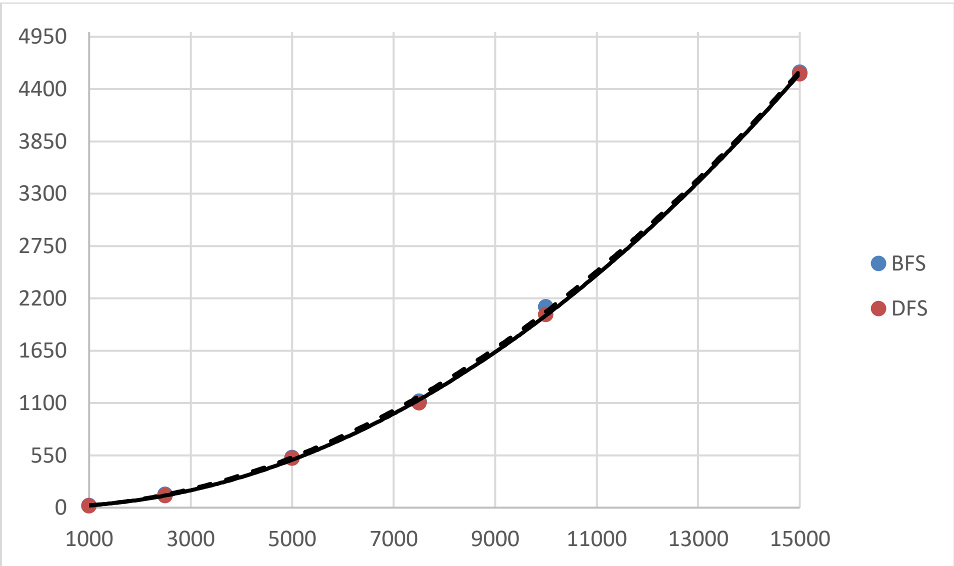
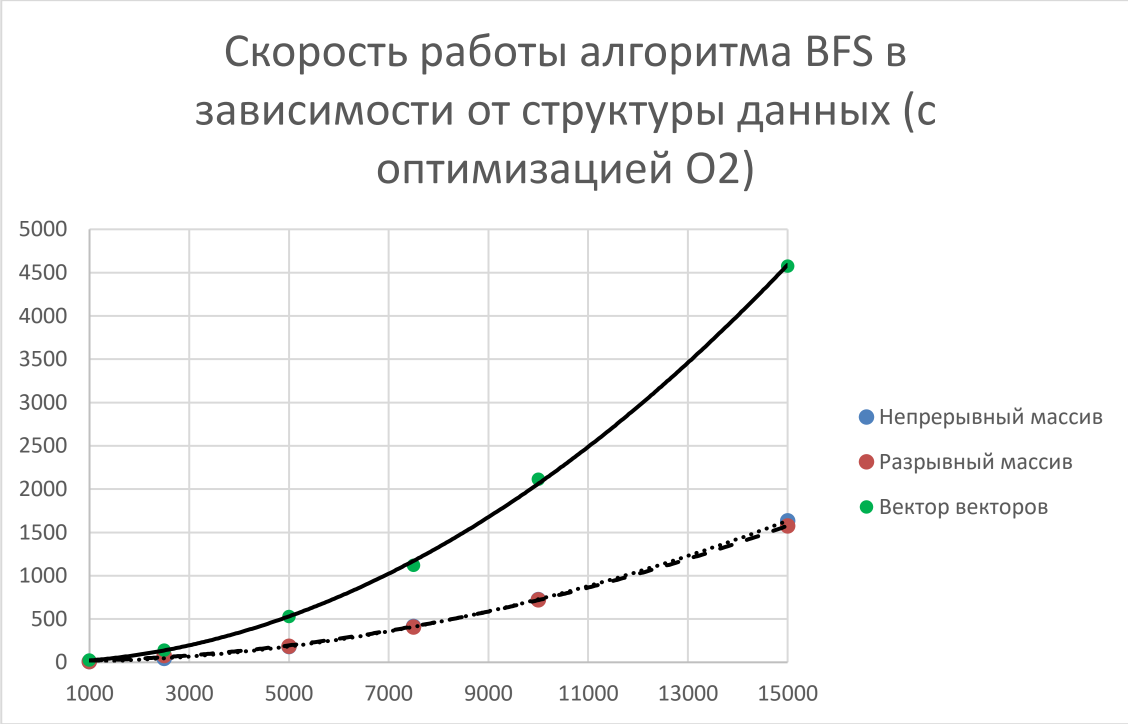


Рисунок 5. Вектор векторов с оптимизацией

Как видно из графиков, оптимизация O2 дает средний прирост производительности примерно в 1,5 раза.

Наложим графики и исследуем производительность алгоритмов в зависимости от структуры данных при включенной оптимизации.



Как видно из графиков, выполняются те же соотношения производительности, как и в случае с выключенной оптимизацией.

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 том числе, были написана топологическая сортировка нецикличного графа на основе DFS и поиск кратчайшего пути между двумя вершинами невзвешенного графа на основе BFS. Также было произведено сравнение скорости выполнения этих алгоритмов (реализованных в данной работе) для графов различных размеров и разных структур данных.

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/