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-36 Алёшин Михаил Алексеевич

Ссылка на репозиторий: https://github.com/MUCTR-IKT-CPP/MAAleshin\_36\_algo

Приняли: Пысин Максим Дмитриевич

Краснов Дмитрий Олегович

Дата сдачи: 14.02.2022

Оглавление

[Описание задачи. 2](#_Toc63548272)

[Описание метода/модели. 2](#_Toc63548273)

[Выполнение задачи. 2](#_Toc63548274)

[Заключение. 2](#_Toc63548275)

# Описание задачи.

Необходимо реализовать генератор случайных графов, генератор должен содержать следующие параметры:

* Максимальное/Минимальное количество генерируемых вершин
* Максимальное/Минимальное количество генерируемых ребер
* Максимальное количество рёбер, 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.

В данной лабораторной работе у нас присутствуют 2 вида обхода графа: **DFS** и **BFS**  
**DFS** (**Deep first search**) следует концепции «погружайся глубже, головой вперед» («go deep, head first»). Идея заключается в том, что мы двигаемся от начальной вершины (точки, места) в определенном направлении (по определенному пути) до тех пор, пока не достигнем конца пути или пункта назначения (искомой вершины). Если мы достигли конца пути, но он не является пунктом назначения, то мы возвращаемся назад (к точке разветвления или расхождения путей) и идем по другому маршруту.

**BFS (Breadth first search)** следует концепции «расширяйся, поднимаясь на высоту птичьего полета» («go wide, bird’s eye-view»). Вместо того, чтобы двигаться по определенному пути до конца, BFS предполагает движение вперед по одному соседу за раз. Вместо следования по пути, BFS подразумевает посещение ближайших к s соседей за одно действие (шаг), затем посещение соседей соседей и так до тех пор, пока не будет обнаружено t.

**Алгоритм DFS:**

1. Выбираем любую вершину из еще не пройденных, обозначим ее как **u**.
2. Запускаем процедуру dfs(u)
   1. Помечаем вершину **u** как пройденную
   2. Для каждой не пройденной смежной с u вершиной (назовем ее v) запускаем dfs(v)
3. Повторяем шаги 1 и 2, пока все вершины не окажутся пройденными.

Сложность алгоритма определяется количеством вершин и количеством ребер в графе, вся процедура вызывается для каждой вершины не более одного раза, а в рамках работы процедуры рассматриваются все ребра, исходящие из вершины.

**Алгоритм BFS:**

1. Поместить узел, с которого начинается поиск, в изначально пустую очередь.
2. Извлечь из начала очереди узел u и пометить его как развёрнутый.
   1. Если узел u является целевым узлом, то завершить поиск с результатом «успех».
   2. В противном случае, в конец очереди добавляются все преемники узла u, которые ещё не развёрнуты и не находятся в очереди.
3. Если очередь пуста, то все узлы связного графа были просмотрены, следовательно, целевой узел недостижим из начального; завершить поиск с результатом «неудача».
4. Вернуться к п. 2.

Сложность алгоритма определяется количеством вершин и количеством ребер в графе, вся процедура вызывается для каждой вершины не более одного раза, а в рамках работы процедуры рассматриваются все ребра, исходящие из вершины.

# Выполнение задачи.

Данная лабораторная работа была реализована на языке python. Реализованы классы Generate и Graph. При объявлении класса Generate при помощи метода generate\_graph() генерируется новый граф с заданными условиями. Далее в классе Graph при помощи разных методов класса можно получить разные виды представления графа: матрица смежности, матрица инцидентности, список смежности и список ребер. На рис. 1 представлены все виды представления графа.

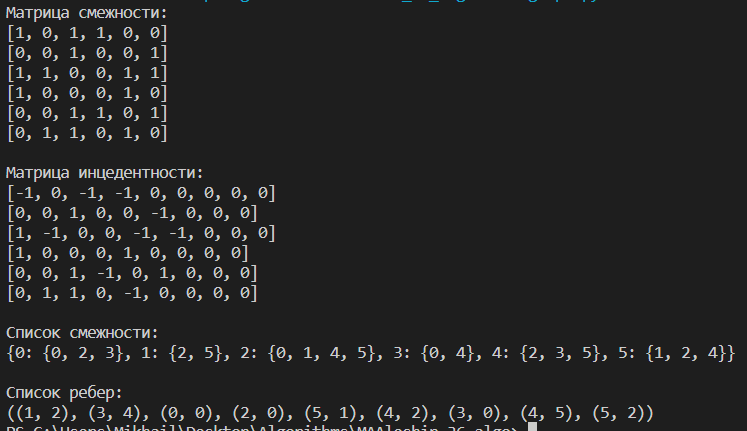
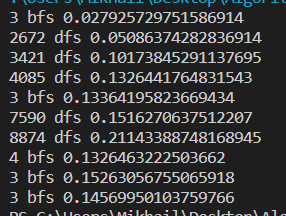
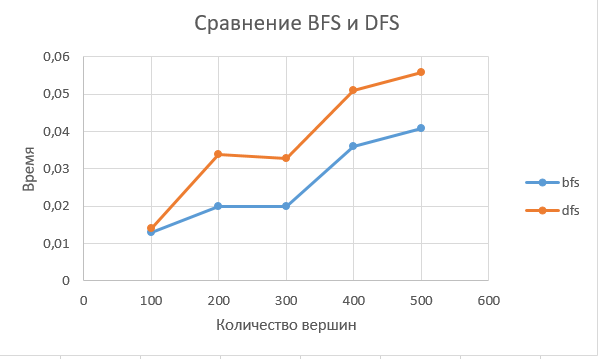


Рис. 1. Представление графа.

Далее в функции analys() генерируются 10 разных графов, каждый из которых больше предыдущего. Генерируются точки A и B для обхода по графу. После этого в случайном порядке происходит обход по графу методом DFS или BFS. Далее на рис. 1 можно увидеть результаты работы алгоритмов:

  
Рис. 2. Результаты анализа.

Из рис. 2 можно увидеть, что длина пути для BFS не сильно различается, в то время как для DFS длина пути увеличивается с увеличением количества вершин. Далее посмотрим результаты времени работы алгоритмов:

  
Рис. 3. График зависимости времени от количества вершин для разных методов.

Если сравнить данный графики, то можно увидеть, что графики идут почти параллельно, однако для DFS понадобиться большее количество времени, чем для BFS. Так же, вне зависимости от метода, время работы алгоритмов растет с ростом количества вершин.

# Заключение.

Исходя из полученных данных и из горького опыта, можно сделать вывод, что более стабильным является BFS, так как в DFS при большом количестве ребер и вершин происходит переполнение стека, в то время как в BFS переполнение очереди почти невозможно. Для проверки существования пути лучше подойдет DFS, в то время как для поиска оптимального пути лучше подойдет BFS, так как DFS не всегда может выдать точный результат. DFS может выдать результат лучше, если наша вторая искомая вершина будет, например, на конце иерархического дерева. Так же, можно увидеть закономерность, что при получении хотя бы одного представления графа можно выразить все остальные, однако необходимо жестко регулировать входные данные, так как, например, для матрицы инцидентности особенно важен порядок вершин в пути.