Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования

«Пермский Национальный Исследовательский Политехнический Университет»

Факультет прикладной математики и механики.

Кафедра «Высшая математика»

Курсовая работа

по дисциплине: Оптимизация на графах

тема: «Разработка визуализации графа по матрице весов и нахождение кратчайшего пути с помощью алгоритма Дейкстры»

Выполнил:

Руденко И. И.

ст. гр. МАК-21-1м

Проверила:

к. ф.-м. н., доцент

Плаксина В. П.

Пермь

2021

**Содержание**

[Введение 3](#_Toc51483550)

[Теоретическая часть](#_Toc51483565) 4

Практическая часть 19

[Заключение](#_Toc51483568) 52

[Список литературы:](#_Toc51483573) 54

Приложения 55

# 

# Введение

В наше время достаточно часто можно встретить задачу о нахождении кратчайшего пути в графе, хотя мы об этом даже не задумываемся. Такие задачи могут применяться в различных сферах: IT, службы экстренной помощи, административные и частные организации, повседневная жизнь и т.д.

Т. о. нахождение кратчайшего пути на графе достаточно актуальная тема, хотя существует множество способов и алгоритмов для её решения. В данной курсовой работе будет подробно рассматриваться алгоритм Дейкстры, а также выполняться реализация этого алгоритма с помощью IT.

Цель работы: Разработать программу с визуализацией графа по матрице весов, приятным интерфейсом и нахождением кратчайшего пути с помощью алгоритма Дейкстры.

Задачи:

1. Рассмотреть алгоритм Дейкстры.
2. Реализовать алгоритм с помощью ЯП.
3. Реализовать качественный и приятный вывод результатов алгоритма.
4. Реализовать визуализацию графа по матрице весов.

**Теоретическая часть**

**Алгори́тм Де́йкстры** — [алгоритм](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC) на [графах](https://ru.wikipedia.org/wiki/%D0%93%D1%80%D0%B0%D1%84_(%D0%BC%D0%B0%D1%82%D0%B5%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)), изобретённый нидерландским учёным [Эдсгером Дейкстрой](https://ru.wikipedia.org/wiki/%D0%94%D0%B5%D0%B9%D0%BA%D1%81%D1%82%D1%80%D0%B0,_%D0%AD%D0%B4%D1%81%D0%B3%D0%B5%D1%80_%D0%92%D0%B8%D0%B1%D0%B5) в [1959 году](https://ru.wikipedia.org/wiki/1959_%D0%B3%D0%BE%D0%B4). Находит кратчайшие пути от одной из вершин графа до всех остальных. Алгоритм работает только для графов без [рёбер](https://ru.wikipedia.org/wiki/%D0%A0%D0%B5%D0%B1%D1%80%D0%BE_(%D1%82%D0%B5%D0%BE%D1%80%D0%B8%D1%8F_%D0%B3%D1%80%D0%B0%D1%84%D0%BE%D0%B2)) отрицательного [веса](https://ru.wikipedia.org/wiki/%D0%A1%D0%BB%D0%BE%D0%B2%D0%B0%D1%80%D1%8C_%D1%82%D0%B5%D1%80%D0%BC%D0%B8%D0%BD%D0%BE%D0%B2_%D1%82%D0%B5%D0%BE%D1%80%D0%B8%D0%B8_%D0%B3%D1%80%D0%B0%D1%84%D0%BE%D0%B2#%D0%92). Поскольку в большинстве приложений это выполняется, такое ограничение не снижает популярность алгоритма Дейксты. [1] Алгоритм широко применяется в программировании и технологиях, например, его используют протоколы маршрутизации [OSPF](https://ru.wikipedia.org/wiki/OSPF) и [IS-IS](https://ru.wikipedia.org/wiki/IS-IS).

Литературный обзор и его анализ

В ходе написания работы для формулировки задачи и составления псевдокода была использована следующая литература:

1. *Томас Х. Кормен, Чарльз И. Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн.* Алгоритмы: построение и анализ.

Издание представлено на рисунке 1.



Рисунок 1 – Учебное издание: “Алгоритмы: Построение и анализ”

1. [*Левитин А. В.*](https://www.wikidata.org/wiki/Q21694518) Глава 9. Жадные методы: Алгоритм Дейкстры // [Алгоритмы. Введение в разработку и анализ](https://www.wikidata.org/wiki/Q21694522) — М.: [Вильямс](https://www.wikidata.org/wiki/Q21694521), 2006.

Издание представлено на рисунке 2.

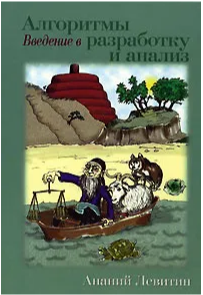


Рисунок 2 – Учебное издание: “Алгоритмы. Введение в разработку и анализ”

1. [Dijkstra E. W.](https://ru.wikipedia.org/wiki/%D0%94%D0%B5%D0%B9%D0%BA%D1%81%D1%82%D1%80%D0%B0,_%D0%AD%D0%B4%D1%81%D0%B3%D0%B5%D1%80_%D0%92%D0%B8%D0%B1%D0%B5) [A note on two problems in connexion with graphs](http://www-m3.ma.tum.de/foswiki/pub/MN0506/WebHome/dijkstra.pdf) (англ.) // [Numer. Math](https://en.wikipedia.org/wiki/Numerische_Mathematik) / [F. Brezzi](https://en.wikipedia.org/wiki/Franco_Brezzi) — [Springer Science+Business Media](https://ru.wikipedia.org/wiki/Springer_Science%2BBusiness_Media), 1959.

Фрагмент монографии представлен на рисунке 3.

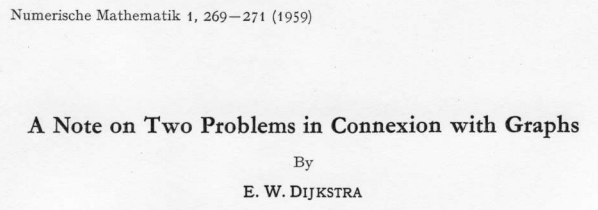


Рисунок 3 – Фрагмент монографии из учебного издания: “Numerische Mathematik”

1. А.В. Кухарев, Е.А. Витько, А.А. Царев. Алгоритмы на графах. Витебск ВГУ имени П.М. Машерова 2016.

Методическое пособие представлено на рисунке 4.

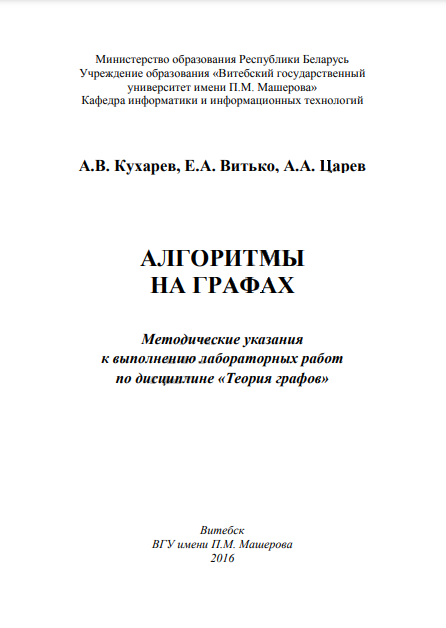


Рисунок 4 – Методическое пособие: “Алгоритмы на графах”

Э. Дейкстра описал две задачи в своей монографии, которая была рассмотрена в литературном обзоре. Формулировку задачи и описание алгоритма было решено взять из этой монографии. Рассмотрим одну из задач:

Problem 2. Find the path of minimum total length between two given nodes

P and Q.

We use the fact that, if R is a node on the minimal path from P to Q, knowledge of the latter implies the knowledge of the minimal path from P to R. In the solution presented, the minimal paths from P to the other nodes are constructed in order of increasing length until Q is reached.

In the course of the solution the nodes are subdivided into three sets:

A. the nodes for which the path of minimum length from P is known; nodes will be added to this set in order of increasing minimum path length from node P;

B. the nodes from which the next node to be added to set A will be selected; this set comprises all those nodes that are connected to at least one node of set A but do not yet belong to A themselves;

C. the remaining nodes.

The branches are also subdivided into three sets:

I. the branches occurring in the minimal paths from node P to the nodes in set A;

II. the branches from which the next branch to be placed in set I will be

selected; one and only one branch of this set will lead to each node in set B;

III. the remaining branches (rejected or not yet considered).

To start with, all nodes are in set C and all branches are in set III. We now transfer node P to set A and from then onwards repeatedly perform the following steps.

Step 1. Consider all branches r connecting the node just transferred to set A

with nodes R in sets B or C. If node R belongs to set B, we investigate whether

the use of branch r gives rise to a shorter path from P to R than the known

path that uses the corresponding branch in set II. If this is not so, branch r is

rejected; if, however, use of branch r results in a shorter connection between P

and R than hitherto obtained, it replaces the corresponding branch in set II

and the latter is rejected. If the node R belongs to set C, it is added to set B and

branch r is added to set II.

Step 2. Every node in set B can be connected to node P in only one way if we restrict ourselves to branches from set I and one from set II.

In this sense each node in set B has a distance from node P: the node with minimum distance from P is transferred from set B to set A, and the corresponding branch is transferred from set II to set I.

We then return to step 1 and repeat the process until node Q is transferred to set A. Then the solution has been found.

Remark 1. The above process can also be applied in the case where the length of a branch depends on the direction in which it is traversed.

Remark 2. For each branch in sets I and II it is advisable to record its two nodes (in order of increasing distance from P), and the distance between P and that node of the branch that is furthest from P. For the branches of set I this is the actual minimum distance, for the branches of set II it is only the minimum thus far obtained. [3]

Перевод с английского на русский:

Задача 2. Найти путь минимальной общей длины между двумя заданными узлами(вершинами) P и Q.

Мы используем тот факт, что, если R – узел(вершина) на минимальном пути от P до Q, знание последнего подразумевает знание минимального пути от P до R. В представленном решении, минимальные пути от P до других узлов(вершин) построены в порядке увеличения длины до достижения Q.

В процессе решения узлы(вершины) подразделяются на три набора:

A. узлы(вершины), для которых известен путь минимальной длины от P; узлы(вершины) будут добавлены к этому набору в порядке увеличения минимальной длины пути от узла(вершины) P;

B. узлы(вершины), из которых будет выбран следующий узел(вершина), который будет добавлен в набор A; этот набор включает все те узлы(вершины), которые подключены по крайней мере к одному узлу(вершине) набора A, но еще не принадлежат самому A;

C. остальные узлы(вершины).

Ветви(рёбра) также подразделяются на три группы:

I. ветви(рёбра), возникающие на минимальных путях от узла(вершины) P к узлам(вершинам) в наборе А;

II. ветви(рёбра), из которых следующая ветвь(ребро) будет выбрана и помещена в набор I; одна и только одна ветвь этого набора приведет к каждому узлу(вершине) в наборе B;

III. остальные ветки(рёбра) (отклоненные или еще не рассмотренные).

Начнем с того, что все узлы(вершины) находятся в наборе C, а все ветви(рёбра) - в наборе III. Мы сейчас передаём узел P в набор A и с этого момента многократно выполнять следующие шаги.

Шаг 1. Рассмотрим все ветви(рёбра) r, соединяющие узел(вершины), только что переданный в набор A с узлами(вершинами) R в наборах B или C. Если узел(вершина) R принадлежит множеству B, мы исследуем, приводит ли использование ветви(рёбра) r к более короткому пути от P к R, чем известный путь, который использует соответствующую ветвь в наборе II. Если, использование ветви(рёбра) r приводит к более короткому пути между P и R, чем полученный на данный момент, он заменяет соответствующую ветвь в наборе II и последний отклоняется. Если узел(вершина) R принадлежит набору C, он добавляется к набору B и ветка(ребро) r добавляется ​​в набор II.

Шаг 2. Каждый узел в наборе B может быть подключен к узлу P только одним способом если ограничиться ветвями(рёбрами) из набора I и одной из набора II. Т.е. каждый узел(вершина) в наборе B имеет путь от узла(вершины) P: узел(вершина) с минимальной длиной пути из P переносится из набора B в набор A, а соответствующая ветвь(ребро) переносится из набора II в набор I. Затем мы возвращаемся к шагу 1 и повторяем процесс пока узел(вершина) Q не будет переведен в набор A. Тогда считается, что решение было найдено.

Замечание 1. Описанный выше процесс также может применяться в случае, когда ребро является дугой.

Замечание 2. Для каждой ветви в наборах I и II желательно записать её два узла(вершины) (в порядке возрастания расстояния от P), а расстояние между P и тем узлом(вершиной) ветви(ребра), который наиболее удален от P. Для ветвей набора I это фактическое минимальное расстояние, для ветвей набора II это минимальное расстояние, которое получилось на данный момент. [3]

Ниже представлены перефразированные и дополненные формулировки постановки задачи и алгоритма Э. Дейкстры, которые позволят более чётко определиться с концепцией и структурой разработки алгоритма, для создания программы.

Формулировка задачи:

Ограничения:

* В алгоритме применяется граф сеть.
* Дуги имеют неотрицательный вес( Вес ≥ 0).
* Выделяются две вершины: **источник** s(start) и **сток** t(target).

Формальное определение:

Дан [взвешенный](https://ru.wikipedia.org/wiki/%D0%A1%D0%BB%D0%BE%D0%B2%D0%B0%D1%80%D1%8C_%D1%82%D0%B5%D1%80%D0%BC%D0%B8%D0%BD%D0%BE%D0%B2_%D1%82%D0%B5%D0%BE%D1%80%D0%B8%D0%B8_%D0%B3%D1%80%D0%B0%D1%84%D0%BE%D0%B2#%D0%92) [ориентированный](https://ru.wikipedia.org/wiki/%D0%9E%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D1%8B%D0%B9_%D0%B3%D1%80%D0%B0%D1%84) граф [**{\displaystyle G(V,E)}G(V, E)**](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D0%94%D0%B5%D0%B9%D0%BA%D1%81%D1%82%D1%80%D1%8B#%D0%9E%D0%B1%D0%BE%D0%B7%D0%BD%D0%B0%D1%87%D0%B5%D0%BD%D0%B8%D1%8F) без дуг отрицательного веса. Найти кратчайший путь от некоторой вершины **{\displaystyle a}s** графа **G** {\displaystyle G}до вершины графа **t**.

Обозначения:

* *V* — множество вершин графа
* *E* — множество рёбер графа
* *ω[i, j]* — вес (длина) ребра *ij*
* *s* — вершина, расстояние от которой ищется
* *U* — множество посещённых вершин
* *d[u]* — по окончании работы алгоритма равно длине кратчайшего пути из *s* до вершины *t*
* *p[u]* — по окончании работы алгоритма содержит кратчайший путь из *s* в *t*
* *ν* — текущая вершина, рассматриваемая алгоритмом

Составим псевдокод, для реализации программы, проанализировав источники информации. [1, 2, 4]

Присвоим

*d[s]* ← *0, p[s]* ← *0*

Для всех *u* ∈ *V* отличных от *s* присвоим *d[u]* ← ∞

Пока ∃*ν* ∉ *U*. Пусть *ν* ∉ *U* — вершина с минимальным *d[ν]* занесём *ν* в *U*

Для всех *u* ∉ *U* таких, что *νu* ∈ *E*

если *d[u] > d[ν] + w[νu]* то

изменим *d[u] ← d[ν] + w [νu]*

изменим *p[u] ← (p[ν], u)*

Пример

Рассмотрим выполнение алгоритма на примере графа, показанного на рисунке 5.

Пусть требуется найти кратчайшее расстояние от вершины **s** до вершины **t**.

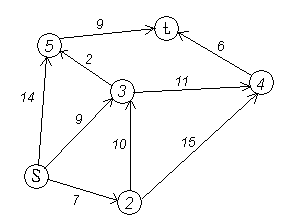


Рисунок 5 – Граф для визуального примера выполнения алгоритма Дейкстры

Кружками обозначены вершины, линиями — пути между ними (дуги графа).

В кружках обозначены номера вершин, над рёбрами обозначен их вес — длина пути.

Рядом с каждой вершиной красным обозначена метка — длина кратчайшего пути в эту вершину из вершины **s**. Подготовительный этап для выполнения алгоритма представлен на рисунке 6.

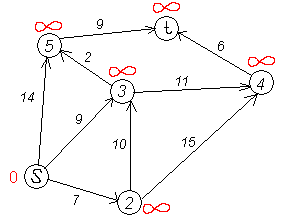


Рисунок 6 – Начальное состояние графа, при выполнении алгоритма Дейкстры

**Первый шаг.**

Минимальную метку имеет вершина **s**. Её соседями являются вершины 2, 3 и 5.

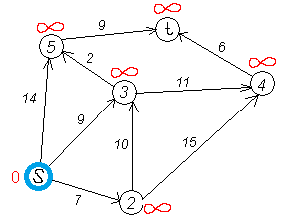


Рисунок 7 – Минимальная метка на первом шаге

Первый по очереди сосед вершины s — вершина 2, потому что длина пути до неё минимальна.

Длина пути в неё через вершину s равна сумме значения метки вершины s и веса ребра, идущего из вершины s в вершину 2, то есть 0 + 7 = 7.

Это меньше текущей метки вершины 2, бесконечности, поэтому новая метка 2-й вершины равна 7.

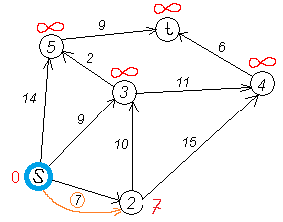


Рисунок 8 – Пересчитываем длину пути до 2-й вершины

Аналогичную операцию проделываем с двумя другими соседями вершины s — 3-й и 5-й.

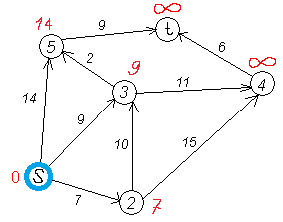


Рисунок 9 – Пересчитываем длину пути до 3-й и 5-й вершин

Все соседи вершины s проверены.

Текущее минимальное расстояние до вершины s считается окончательным и пересмотру не подлежит.

Вычеркнем её из графа, чтобы отметить, что эта вершина посещена.

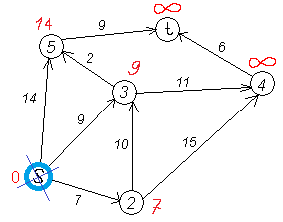


Рисунок 10 – Вычёркиваем вершину s из графа, т.к. она больше не изменится

**Второй шаг.**

Снова находим минимальную из непосещённых вершин. Это вершина 2 с меткой 7.

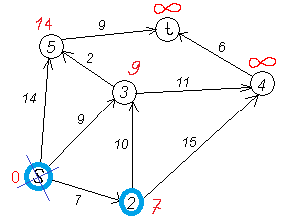


Рисунок 11 – Поиск вершины с минимальной длиной пути

Снова пытаемся уменьшить метки соседей выбранной вершины, пытаясь пройти в них через 2-ю вершину. Соседями вершины 2 являются вершины s, 3 и 4.

Первый (по порядку) сосед вершины 2 — вершина s. Но она уже посещена, поэтому с вершиной s ничего не делаем.

Следующий сосед — вершина 3, так как имеет минимальный номер.

Если идти в неё через 2, то длина такого пути будет равна 17 (7 + 10 = 17). Но текущая метка третьей вершины равна 9, а это меньше 17, поэтому метка не меняется.

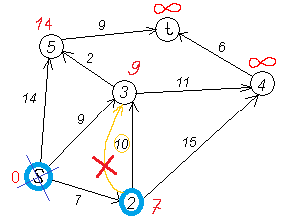


Рисунок 12 – Пересчитываем длину пути в вершине 3

Ещё один сосед вершины 2 — вершина 4.

Если идти в неё через 2-ю, то длина такого пути будет равна сумме кратчайшего расстояния до 2-й вершины и расстояния между вершинами 2 и 4, то есть 22 (7 + 15 = 22).

Поскольку 22 < ∞, устанавливаем метку вершины 4 равной 22.

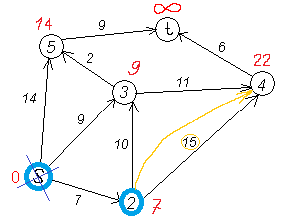


Рисунок 13 – Пересчитываем длину пути в вершине 4

Все соседи вершины 2 просмотрены, фиксируем расстояние до неё и помечаем её как посещённую.

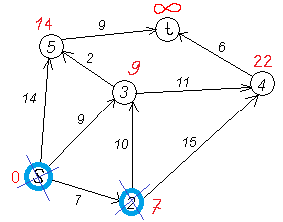


Рисунок 14 – Вычеркиваем вершину 2

**Третий шаг.**

Повторяем шаг алгоритма, выбрав вершину 3, т.к. на данный момент, длина пути до неё является минимальной из вершин 3, 4, 5. После её посещения получим такие результаты:

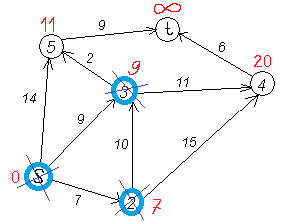


Рисунок 15 – Результат выполнения алгоритма для вершины 3.

**Дальнейшие шаги (Шаг 4, Шаг 5, Шаг 6).**

Повторяем шаг алгоритма для оставшихся вершин. Это будут вершины 5, 4 и t, соответственно порядку.

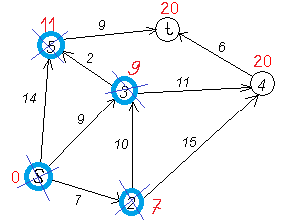


Рисунок 16 – Результат выполнения алгоритма для вершины 5.

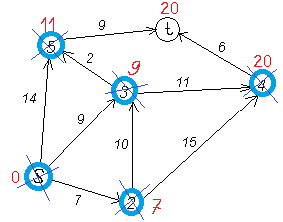


Рисунок 17 – Результат выполнения алгоритма для вершины 4.

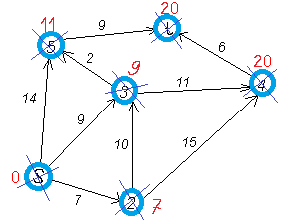


Рисунок 18 – Результат выполнения алгоритма для вершины t.

**Завершение выполнения алгоритма.**

Алгоритм заканчивает работу, когда все вершины посещены.

Результат работы алгоритма виден на рисунке 18: кратчайший путь от вершины s до 2-й составляет 7, до 3-й — 9, до 4-й — 20, до 5-й — 11, до 6-й — 20.

Если в какой-то момент все непосещённые вершины помечены бесконечностью, то это значит, что до этих вершин нельзя добраться (то есть граф несвязный). Тогда алгоритм может быть завершён досрочно. А также, если вершина t уже имеет какую-либо длину пути отличную от бесконечности и она является минимальной, то алгоритм также может быть завершён досрочно, поскольку нас интересует длина кратчайшего пути из вершины s в вершину t. В процессе выполнения алгоритма находятся кратчайшие пути до остальных вершин и это является “приятным бонусом”.

В ходе литературного обзора, кроме описанного выше, было найдено два патента. Первый на ЯП (языке программирования) C++ (рисунок 19 – Патент на ЯП C++). Второй на языке программирования C# (рисунок 20 – Патент на ЯП C#).

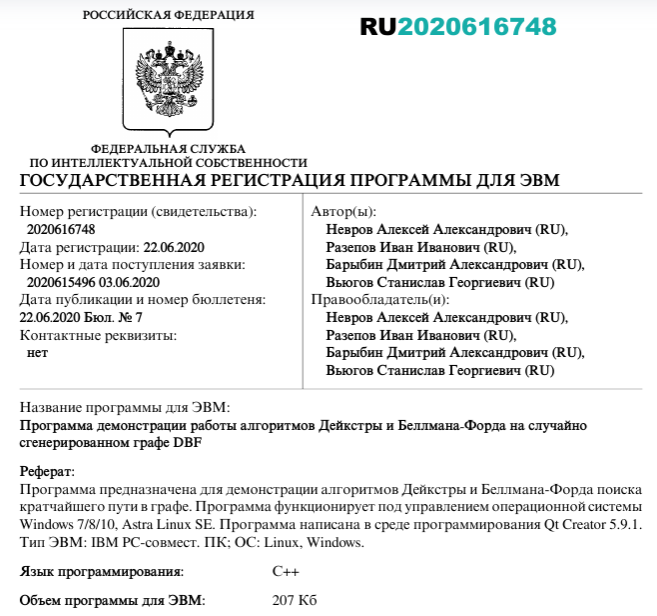


Рисунок 19 – Патент на ЯП C++

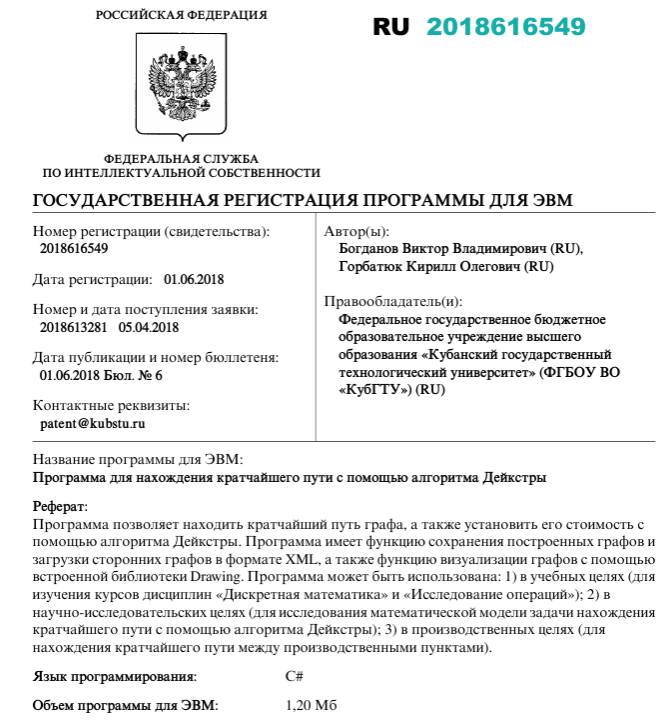
**

Рисунок 20 – Патент на ЯП C#

**Практическая часть**

Прежде чем приступить непосредственно к разработке консольного приложения, выполняющего алгоритм Дейкстры, а затем созданию приложения, визуализирующее граф (возможно создание форматов .jar или .exe), необходимо определить, какие понадобятся инструменты разработки, и установить их.

Для разработки данного приложения был выбран язык Java. Одной из основной причин выбора данного ЯП послужило отсутствие зарегистрированных патентов. (Во время обзора литературы, на настоящий момент, мной не было найдено таких патентов). Второй причиной послужил ряд преимуществ выбранного ЯП.

Ниже перечислены основные преимущества Java:

* Язык Java прост для изучения.

При разработке Java было уделено большое внимание простоте языка, поэтому программы на Java, по сравнению с программами на других языках, проще писать, компилировать, отлаживать и изучать.

* Java - это объектно-ориентированный язык.

Это позволяет создавать модульные программы, исходный код которых может использоваться многократно.

* Язык Java не зависит от платформы.

Одним из основных преимуществ языка Java является возможность переноса программ из одной системы в другую. Поскольку программы на Java не зависят от платформы как на уровне исходного кода, так и на двоичном уровне, их можно запускать в различных системах, что особенно важно для программ, предназначенных для World Wide Web.

Ещё одно преимущество заключается в том, что ЯП Java очень популярен и достаточно большое количество пользователей смогут понять программный код, с помощью которого произведена реализация данной программы.

Индекс сообщества программистов TIOBE - показатель популярности языков программирования. Индекс обновляется раз в месяц. Рейтинги основаны на количестве квалифицированных инженеров во всем мире, курсов и сторонних поставщиков. Популярные поисковые системы, такие как Google, Bing, Yahoo!, Wikipedia, Amazon, YouTube и Baidu, используются для расчета рейтингов. Важно отметить, что индекс TIOBE - это не лучший язык программирования или язык, на котором написано большинство строк кода.

Индекс можно использовать для проверки актуальности ваших навыков программирования или для принятия стратегического решения о том, какой язык программирования следует использовать при создании новой программной системы. [5]

Посмотрим на список 10 лидирующих ЯП по популярности (рисунок 21) и график за последние 20 лет (рисунок 22)

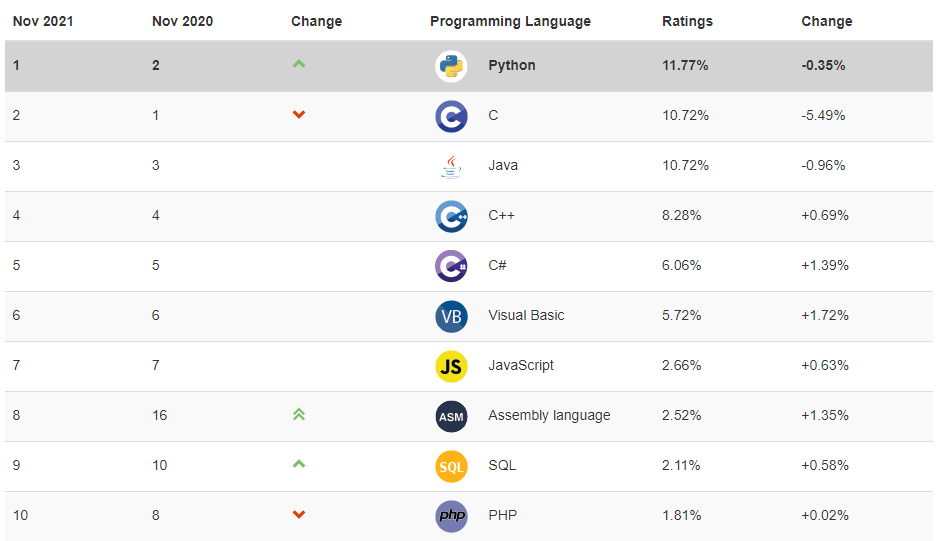


Рисунок 21 – Список 10 лидирующих языков программирования по популярности в 2021 году по индексу TIOBE.

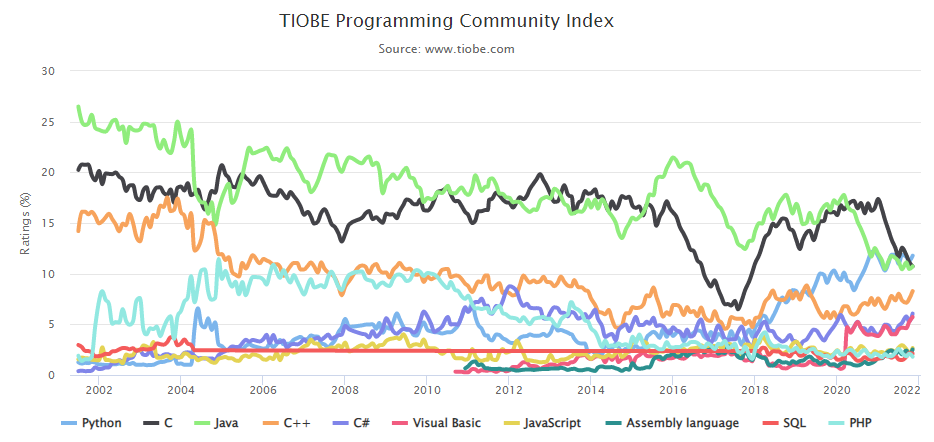


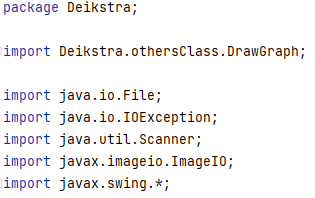
Рисунок 22 – Индекс сообщества программирования TIOBE.

Можно увидеть, что уже более 20 лет ЯП Java занимает 1-3 место каждый год.

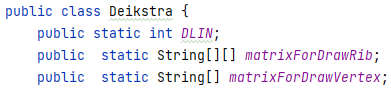
Перейдём к компонентам, необходимым для реализации данной программы. В разрабатываемом приложении используются следующие программные средства и технологии: JDK (Java Development Kit) - набор для разработки приложений на языке Java, который включает в себя компилятор Java, библиотеки, утилиты, документацию и др. IntelliJ IDEA — интегрированная среда разработки программного обеспечения для многих языков программирования, в частности Java, JavaScript, Python, разработанная компанией JetBrains. Эта среда разработки является самой популярной для языка Java, кроме того, исходя из моего опыта, это самая удобная и приятная среда для программиста.

Программная реализация

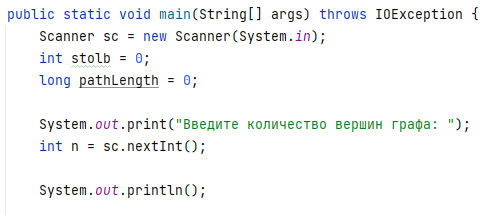
1. Подключаем необходимые библиотеки.



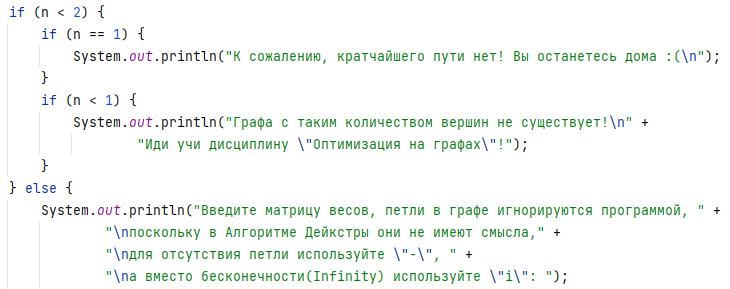
1. Создаём наш класс и объявляем необходимые глобальные переменные и массивы.



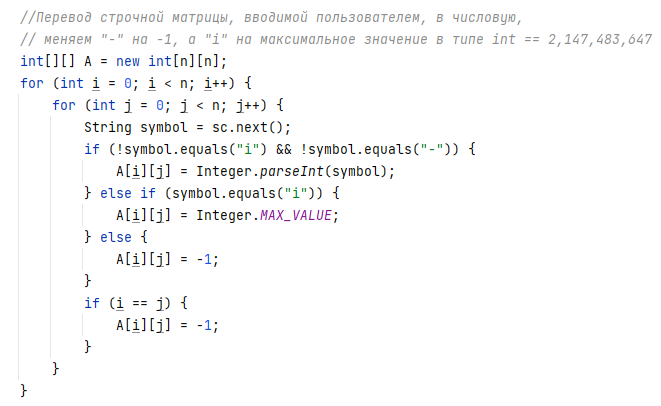
1. В методе main объявляем необходимые для работы переменные, обрабатываем исключение, просим пользователя ввести количество вершин графа и считываем его.



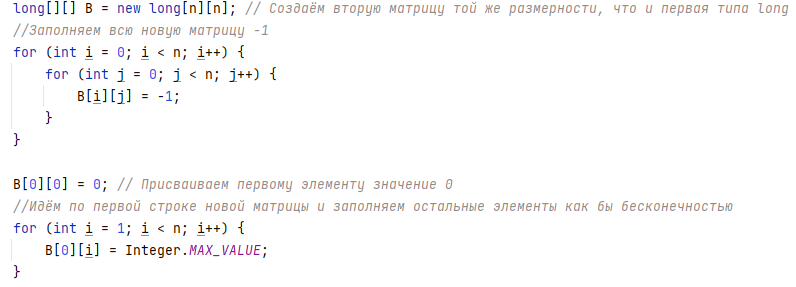
1. Проверяем количество вершин, которое ввёл пользователь и в соответствии с этим выводим на экран “ошибку”, если пользователь ввёл некорректное число вершин или продолжаем работу, если граф с таким количеством вершин существует и прописываем обозначения для ввода матрицы весов.



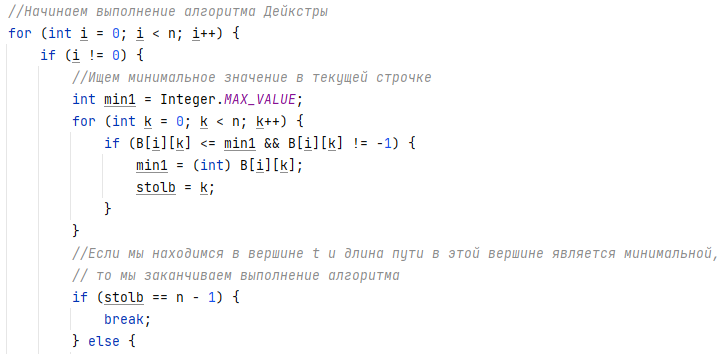
1. Поскольку на входе был массив строк, то нам нужно перевести массив строк в массив чисел, чтобы мы смогли выполнять различные математические операции.

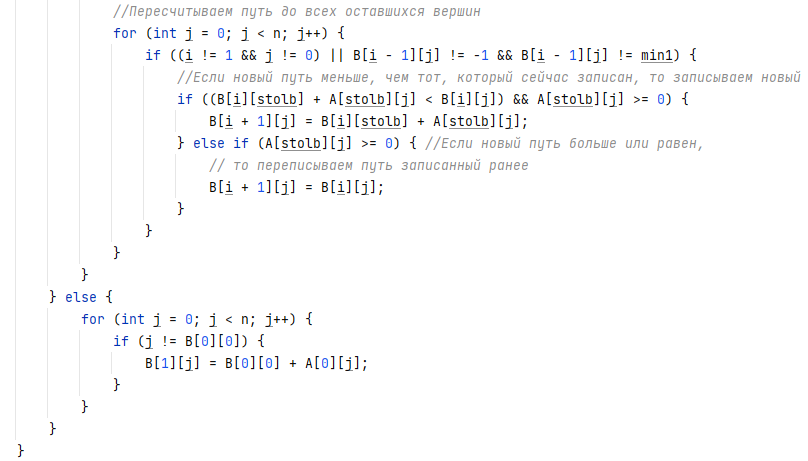


1. Создаём массив для алгоритма Дейкстры и заполняем его “-1” для удобства реализации алгоритма, т.к. у нас не может быть отрицательных значений в этом алгоритме, данный массив пошагово будем заполнять в соответствии с алгоритмом, а также заполняем первую строку алгоритма, поскольку она у нас никогда не меняется.

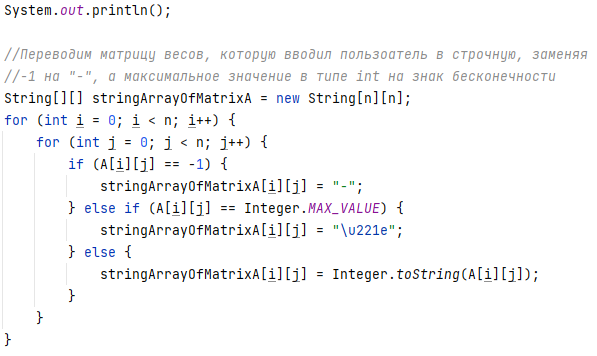


1. Реализуем алгоритм Дейкстры.

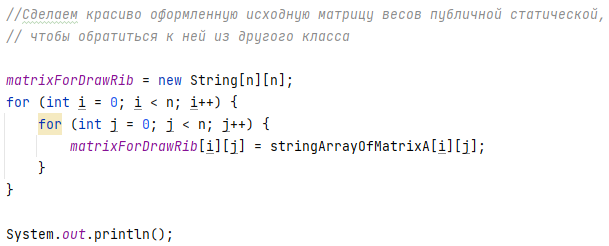




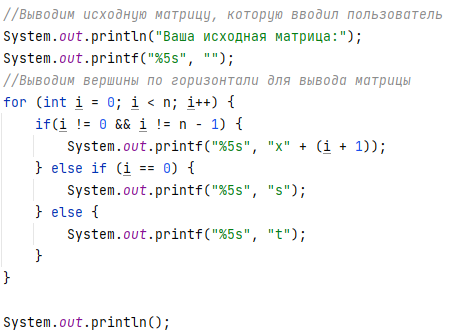
1. Приводим матрицу к красивому виду, которую вводил пользователь, чтобы в дальнейшем её вывести на экран.

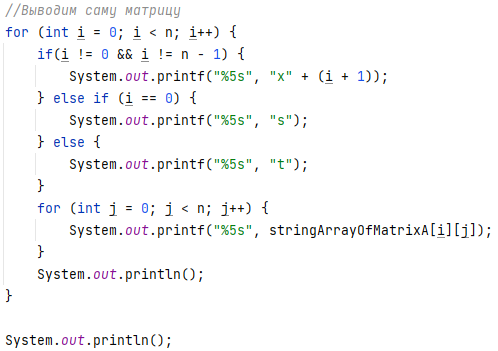


1. Передаём матрицу весов в глобальный массив, чтобы рисовать граф в другом классе.

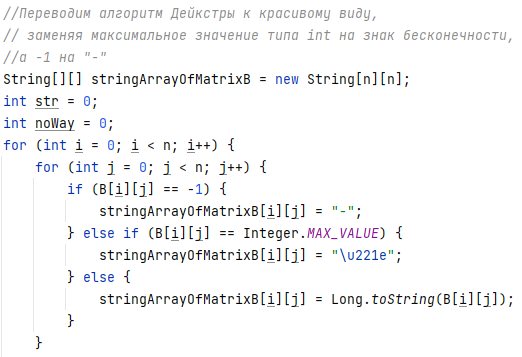


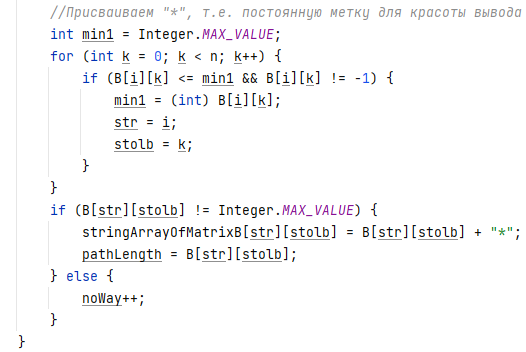
1. Выводим красиво оформленную исходную матрицу на экран.



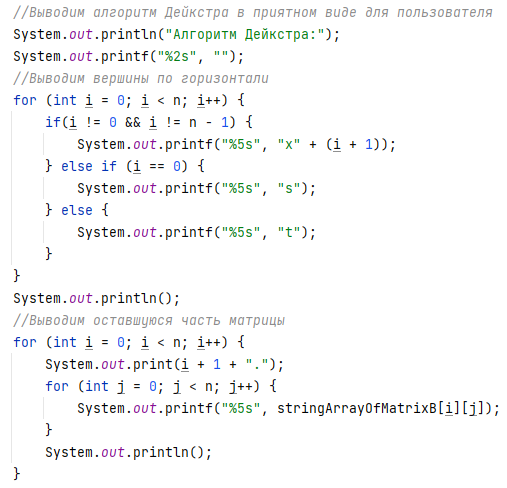


1. Приводим алгоритм Дейкстры к красивому виду, аналогично исходной матрице.

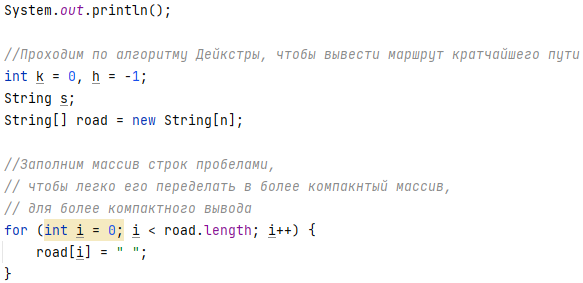


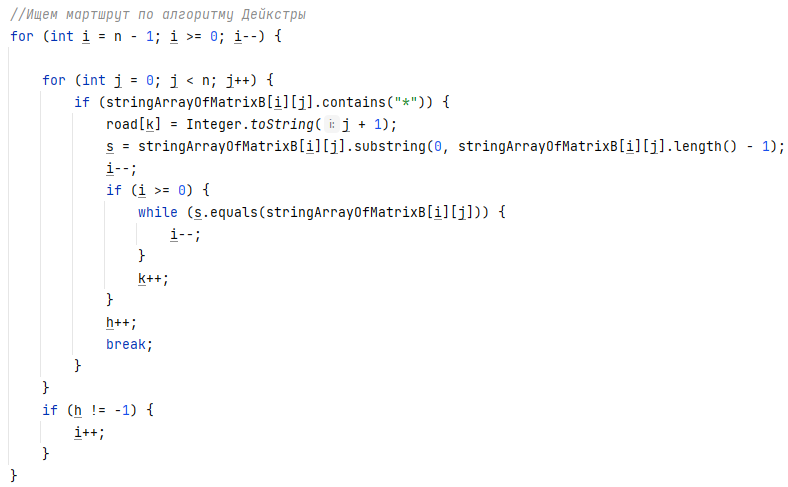


1. Выводим алгоритм Дейкстры в преобразованном виде на экран.

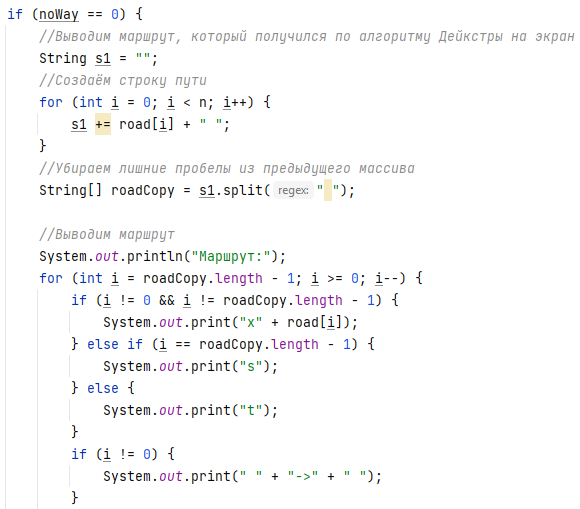


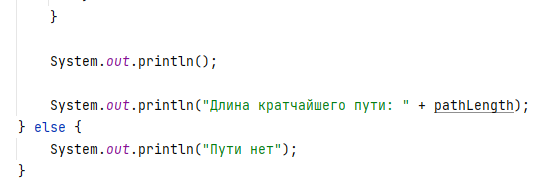
1. Создаём новый массив для поиска маршрута и собираем вершины через которые мы дошли от s к t.



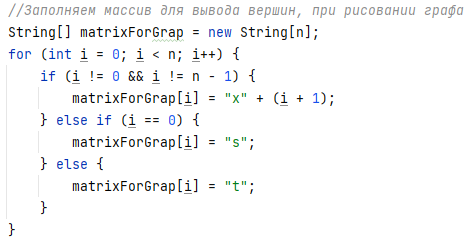


1. Выводим маршрут(путь) на экран, если он существует, если нет, то выводим сообщение: “Пути нет”.

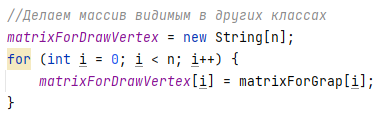




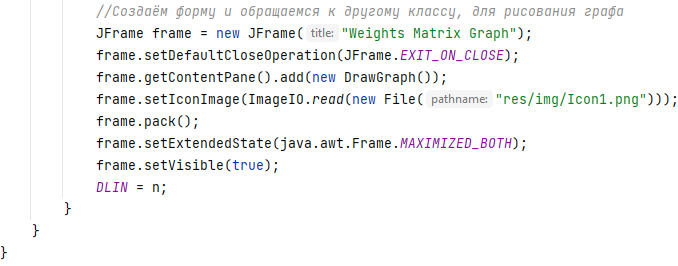
1. Создаём массив вершин и заполняем его для визуализации графа.



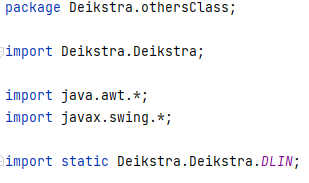
1. Передаём массив вершин в глобальный массив, чтобы использовать его в классе DrawGraph.



1. Создаём форму для визуализации графа; меняем название формы; реализуем реакцию на закрытие окна (программа прекращает работу, когда мы закрываем окно, в котором рисуется граф); рисуем граф; меняем иконку программы; вызываем метод автоматического размещения компонентов на форму, чтобы всё вмещалось и размещалось по умолчанию, если такие имеются; делаем размер окна, в котором рисуется граф, на весь экран; делаем окно, в котором рисуется граф видимым; передаём количество вершин в глобальную переменную для визуализации.



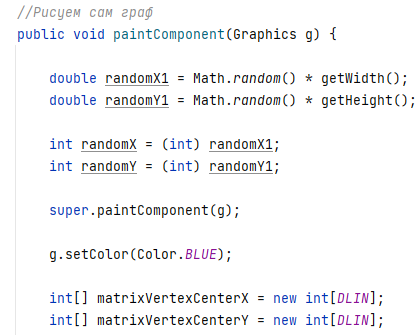
1. Создаём класс, который отвечает за визуализацию графа, подключаем необходимые библиотеки и передаём необходимые переменные из первого класса.



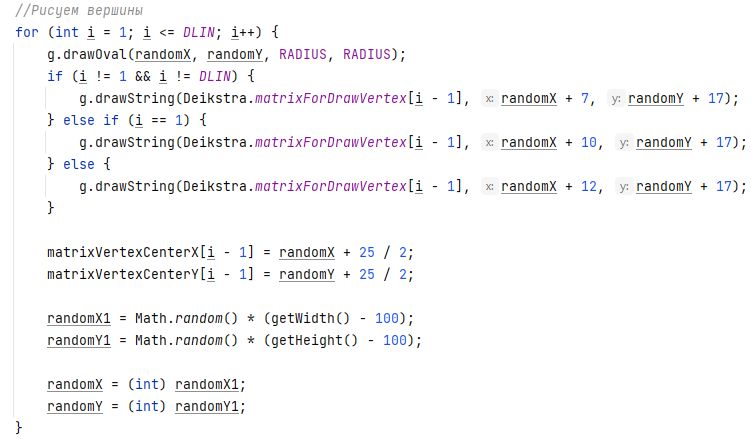
1. Наследуем класс, объявляем необходимые переменные и создаём конструктор класса DrawGraph.



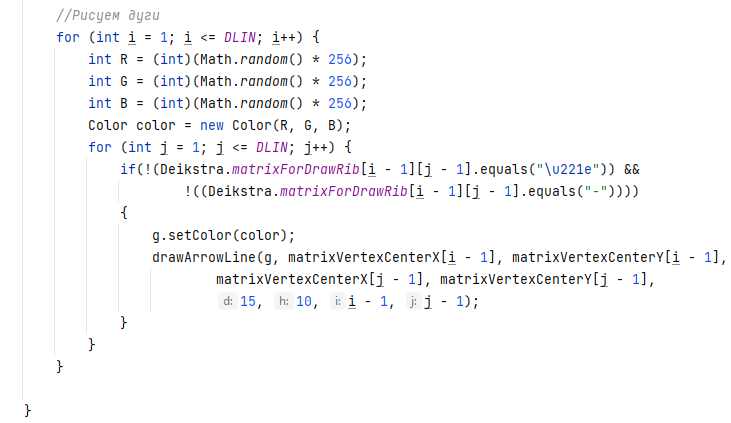
1. Приступаем к визуализации графа. Произвольно задаём координаты вершины s в пределах экрана монитора и создаём два массива, которые будут отвечать координаты центров вершин.



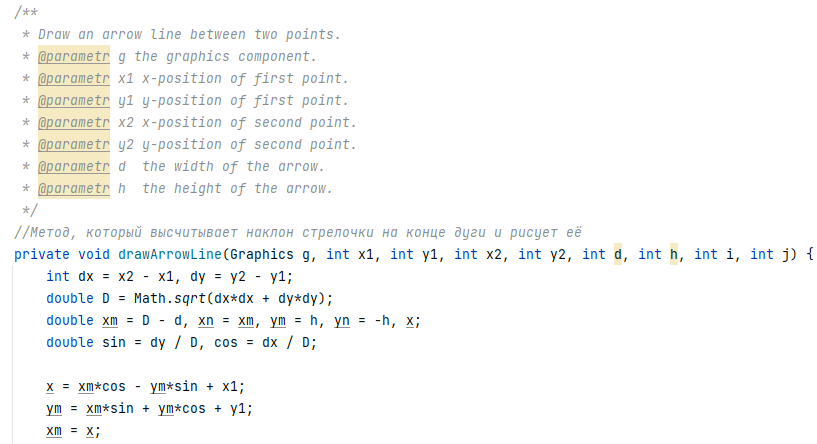
1. Рисуем вершины графа и записываем название(номер) вершины.

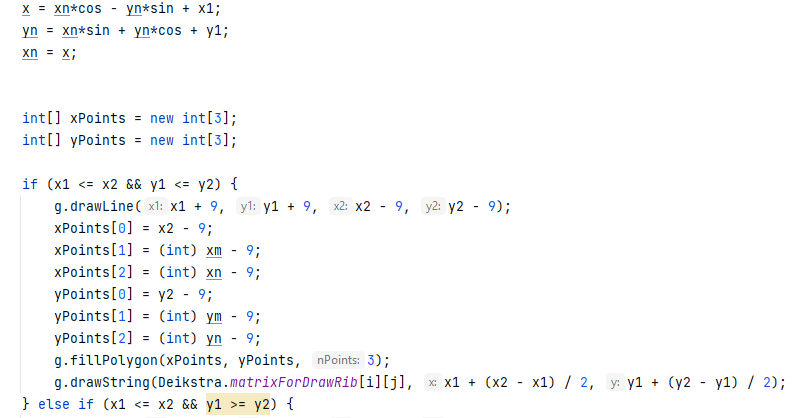


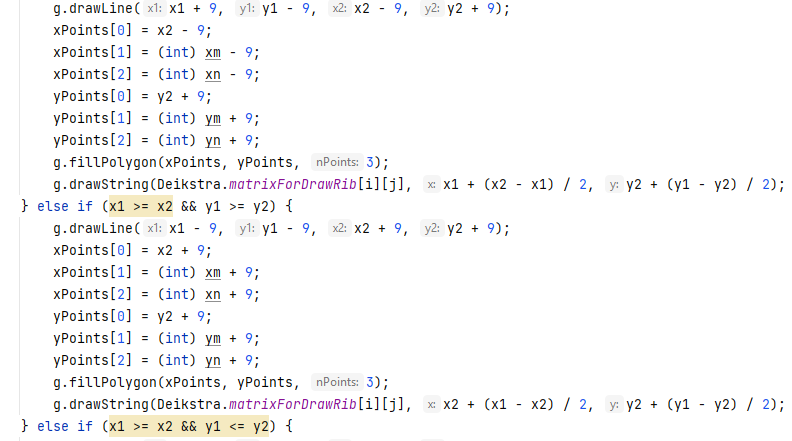
1. Рисуем дуги графа, на данном этапе получатся лишь отрезки от одной вершины до другой, дуги не изображаются, если это петли, а также если весом дуги является бесконечность, т.к. под бесконечностью понимается отсутствие дуги.

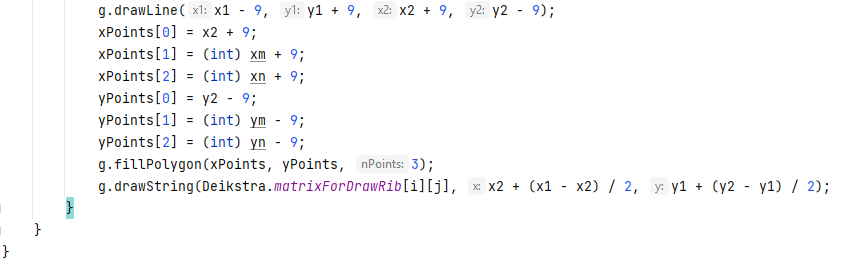


1. Создаём и реализуем метод для отображения конца дуги(стрелочки), высчитываем наклон дуги с используя координаты центров вершин и рисуем конец дуги(стрелочку).





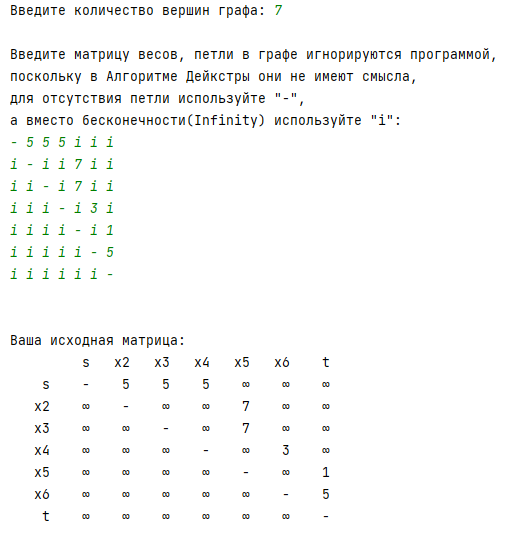


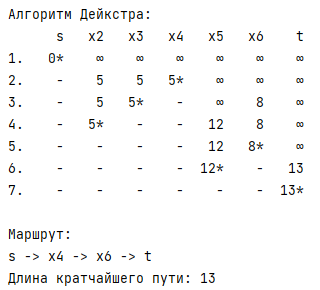


Примеры выполнения программы

Пример 1

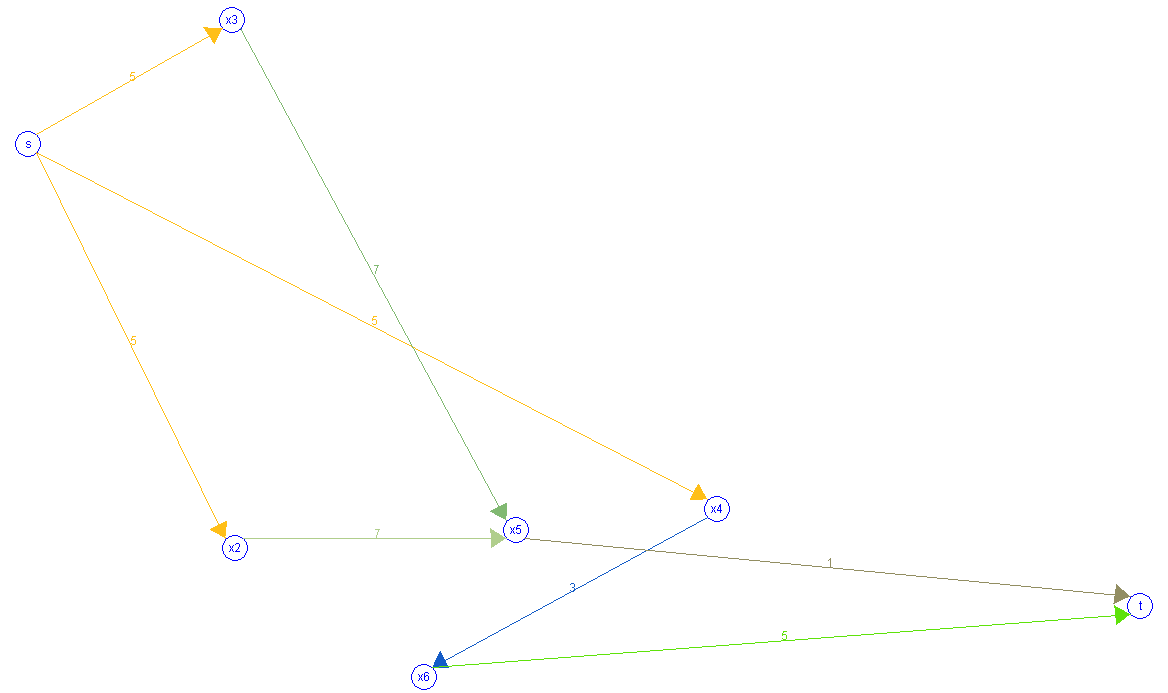
Вывод в консоль:



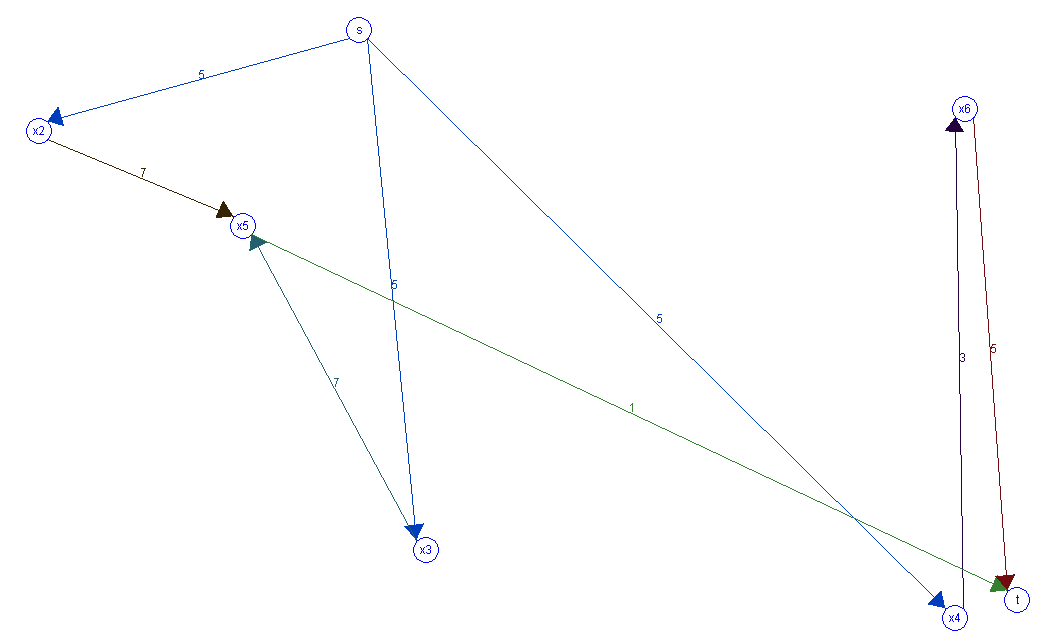


Вывод в форму:

Граф по матрице весов

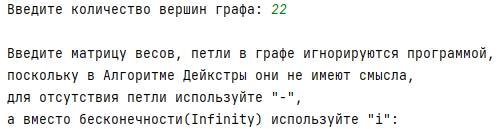


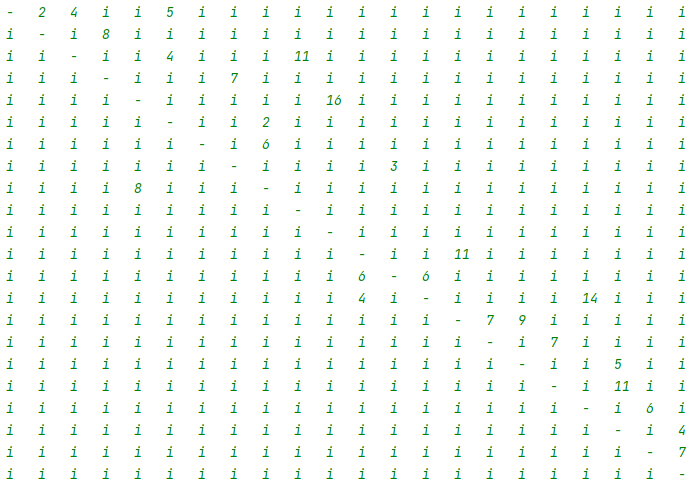
Этот же граф, только перерисованный с помощью двойного щелчка по иконке программы на панели инструментов.

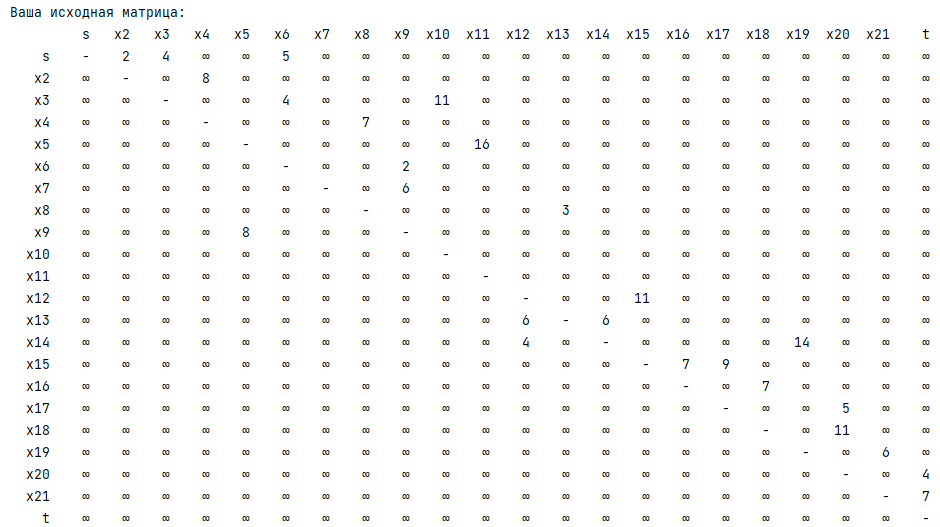


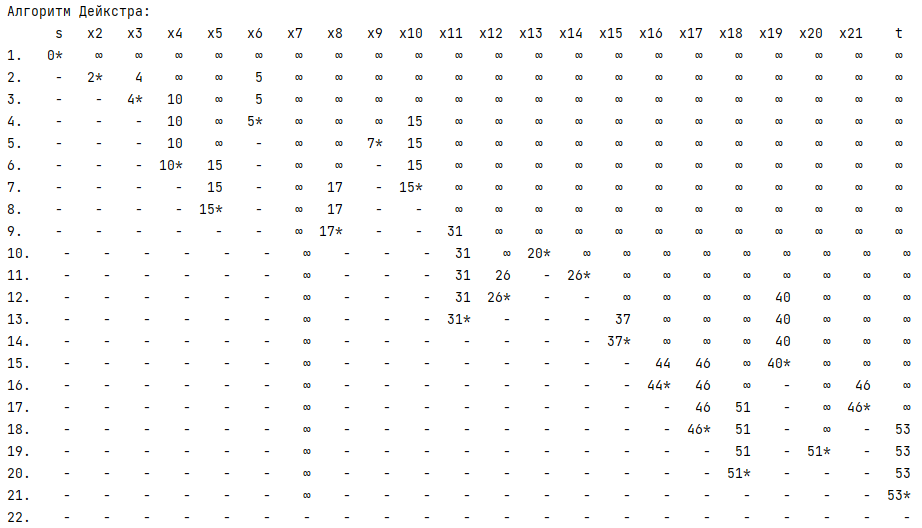
Пример 2

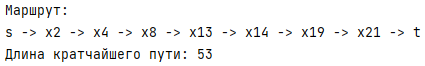
Вывод в консоль:



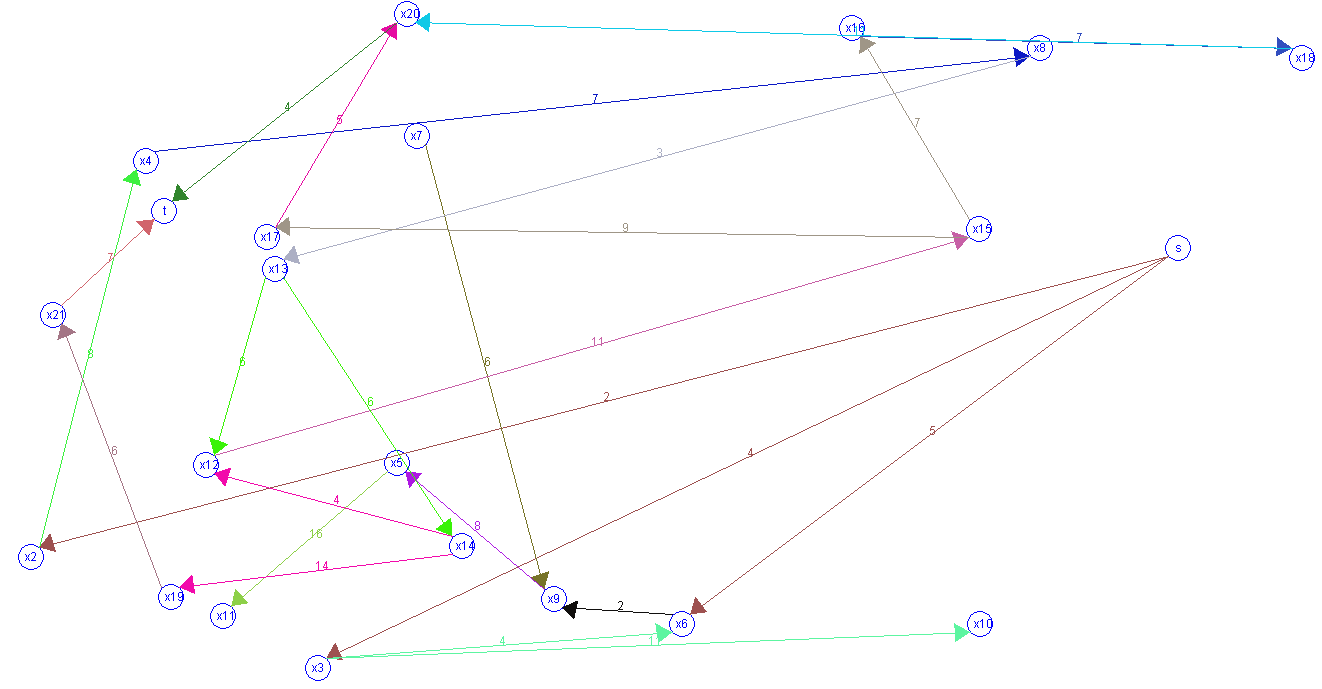




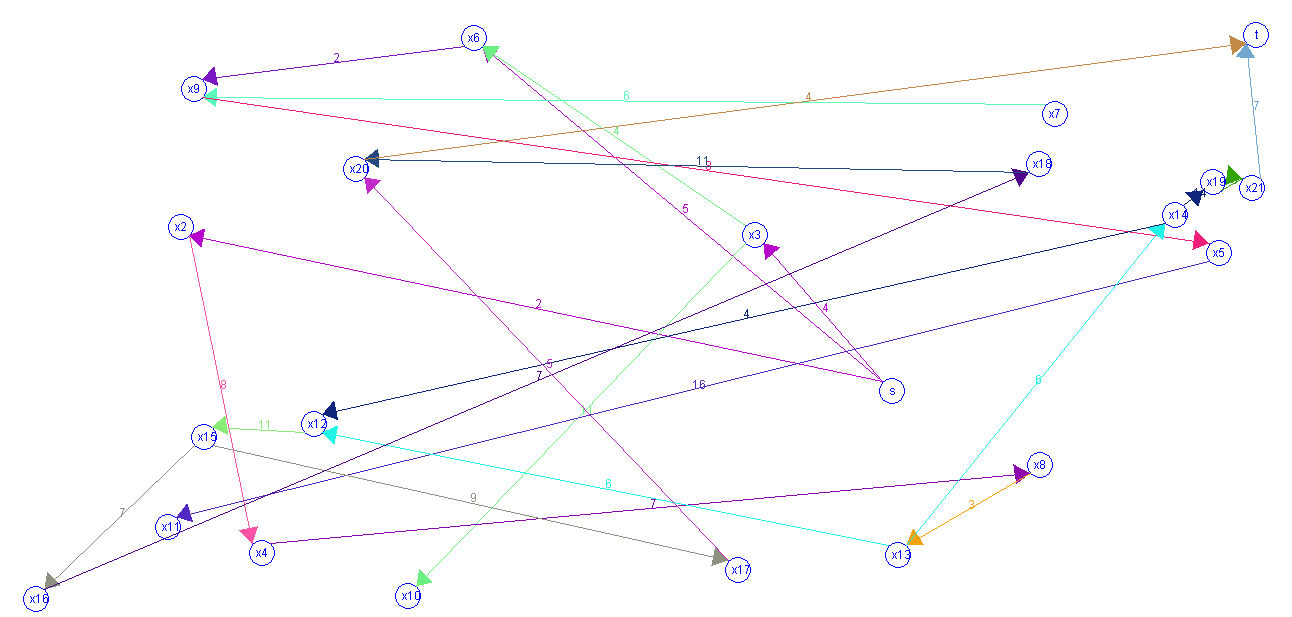




Вывод в форму:



Этот же граф, но перерисованный:



Анализ результатов

Big O

В XXI веке самой популярной оценкой сложности алгоритмов является Big O. Поэтому, это один из методов, который будем использовать, при сравнении программ. Используя Big O сравним только алгоритмы написанные на ЯП, а не весь программный код.

Основные причины, по которым стоит изучить и понимать Big O:

1. Концепцию Big O необходимо понимать, чтобы уметь видеть и исправлять неоптимальный код в программе.

Зная концепцию Big O легко увидеть код программы, который является не оптимальным, а также сразу исправить его.

1. Ни один серьёзный проект, как и ни одно серьёзное собеседование не могут обойтись без вопросов о Big O.

Интервьюеры, нанимающие программистов, всегда просят оценить алгоритм в рамках Big O, если человек, проходящий собеседование, не может оценить алгоритм или допускает ошибки при оценке, то при совершении хотя бы одной ошибки человек не проходит собеседование. Считается, что если человек, желающий устроиться на работу программистом, не может оценить алгоритм в рамках Big O или допускает ошибки, то он не понимает код, который написал.

1. Непонимание Big O ведёт к серьёзной потере производительности алгоритмов.

Временная оценка

В прошлом веке, когда программирование только начинало развиваться, возникла проблема. Как понять, какой алгоритм быстрее и эффективнее работает? Если запускать один и тот же алгоритм на разных компьютерах и сравнивать их по времени, то на современном компьютере, в котором находится современный процессор, алгоритм может отработать за 0,00001 секунду, а используя компьютер, к примеру, 1997 года, алгоритм может выполниться за 2 секунды, что объясняет невозможность сравнения алгоритмов по привычному нам времени. Но Дональд Кнут нашёл выход из этой ситуации. Любой алгоритм имеет определённое количество тактов, которое выполняет процессор прежде, чем алгоритм закончится. Т. о. неважно устарелый или современный процессор используется, т. к. количество шагов на обоих компьютерах будет одинаковым. Т. е. идея Big O – это показать, какое количество шагов необходимо, чтобы алгоритм закончил своё выполнение.

Определение:

Big O – это математическое обозначение для сравнения асимптотического поведения функций.

Это определение не даёт нам чёткого понимания, чтобы оценивать алгоритмы. Поэтому перефразируем это определение с точки зрения программиста и информатики (computer science): Big O показывает верхнюю границу зависимости между входными параметрами функции и количеством операций, которые выполнит процессор.

Рассмотрим самые распространённые сложности алгоритмов. Сначала посмотрим на график, на котором изображена зависимость количества операций от количества элементов (рисунок 23).

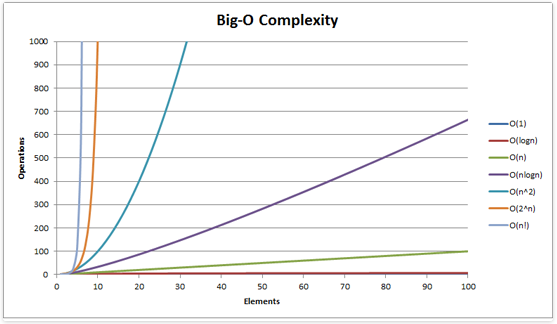


Рисунок 23 – График сложности Big O

По графику можно сделать вывод, что лучшие сложности для алгоритмов это O(1), O(log n), O(n) и O(n \* log n). Но практика показывает, что также часто встречаются алгоритмы сложность которых O(n^2).

Нулевая - O(0).

Означает, что кода программы нет. Т.е. вообще ничего не происходит. Если есть хотя бы одна команда в программе, то сложность будет как минимум O(1).

Константная - O(1).

Означает, что вычислительная сложность алгоритма не зависит от входных данных. Однако, это не значит, что алгоритм выполняется за одну операцию или требует очень мало времени. Это означает, что время не зависит от входных данных.

Линейная - O(n).

Означает, что сложность алгоритма линейно растёт с увеличением входных данных. Другими словами, удвоение размера входных данных удвоит и необходимое время для выполнения алгоритма.

Такие алгоритмы легко узнать по наличию цикла по каждому элементу массива.

Логарифмическая - O(log n).

Означает, что сложность алгоритма растёт логарифмически с увеличением входных данных. В программировании почти всегда в основании log n подразумевается 2. Другими словами это такой алгоритм, где на каждой итерации берётся половина элементов.

К алгоритмам с такой сложностью относятся алгоритмы типа “Разделяй и Властвуй” (Divide and Conquer), например бинарный поиск.

Линеарифметическая или линеаризованная - O(n \* log n).

Означает, что удвоение размера входных данных увеличит время выполнения чуть более, чем вдвое.

Примеры алгоритмов с такой сложностью: Сортировка слиянием или множеством n элементов.

Квадратичная - O(n²), O(n^2).

Означает, что удвоение размера входных данных увеличивает время выполнения в 4 раза. Например, при увеличении данных в 10 раз, количество операций (и время выполнения) увеличится примерно в 100 раз. Если алгоритм имеет квадратичную сложность, то это повод пересмотреть необходимость использования данного алгоритма. Но иногда этого не избежать.

Такие алгоритмы легко узнать по вложенным циклам.

Константы(const).

Стоит учитывать, что при оценке сложности константы отбрасываются, поскольку при оценке сложности алгоритма мы рассматриваем, что n → ∞.

Пример 1.

Сложность O(n / const) или O(n \* const) будет оценена так же, как O(n).

Пример 2.

Сложность O(n ± const) будет оценена так же, как O(n).

Неважная сложность.

Стоит учитывать скорость роста функции. Функции, которые растут намного медленнее отбрасываются.

Пример 1.

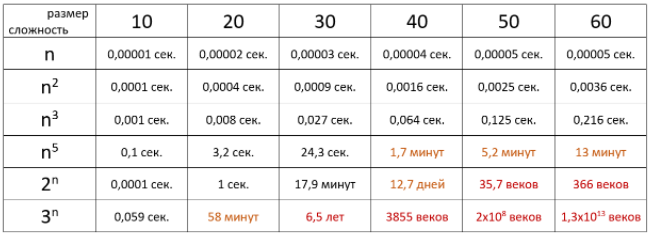
Сложность O(n² + n) будет оценена так же, как O(n²).

Пример 2.

Сложность O( + ) будет оценена так же, как O().

Время выполнения алгоритма с определённой сложностью в зависимости от размера входных данных при скорости операций в секунду представлено в таблице 1:

Таблица 1 – Время алгоритма разных функций при скорости операций в секунду

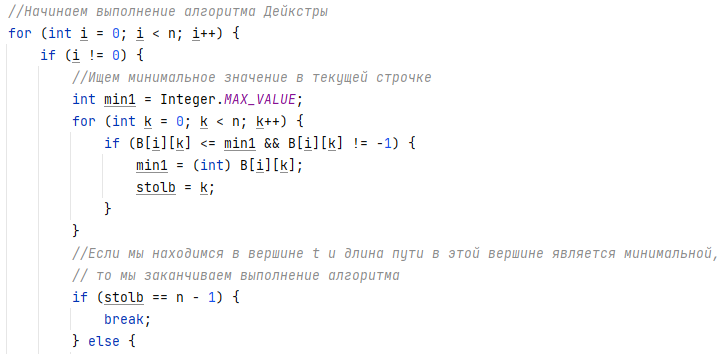


Длина программы.

Бывают ситуации, когда программа с более длинным текстом будет работать быстрее. Поэтому ограничения на длину программы могут быть важны с точки зрения расхода остальных ресурсов. Изучению этого фактора посвящена теория колмогоровской сложности, в том числе с ограничением на ресурсы. [6]

Сравнения программ

Сравнение по времени выполнения алгоритма



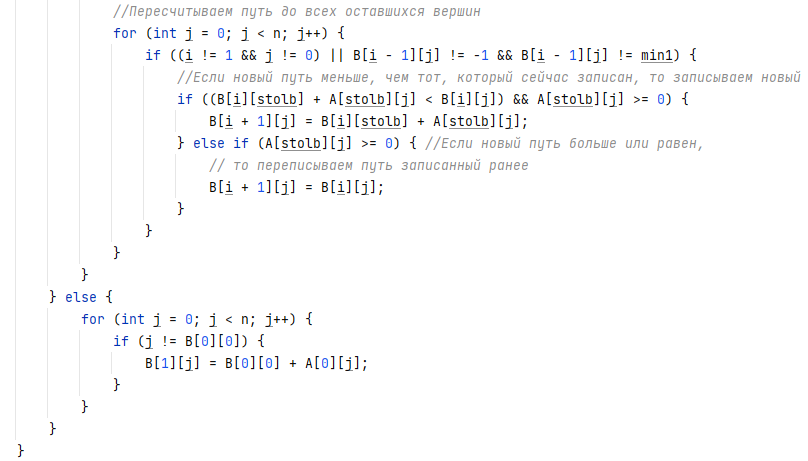


Рисунок 24 - Фрагмент моего кода программы, реализующий алгоритм Дейкстры, написанный на ЯП Java.

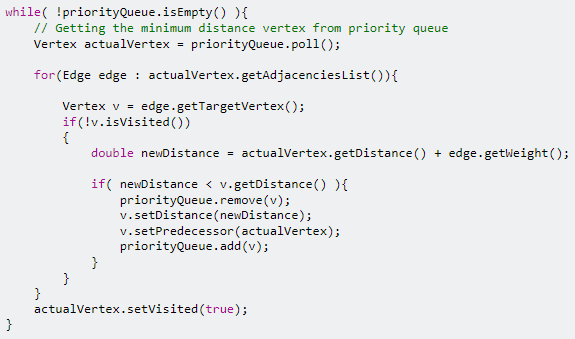


Рисунок 25 - Фрагмент кода программы из интернета, реализующий алгоритм Дейкстры, написанный на ЯП Java.

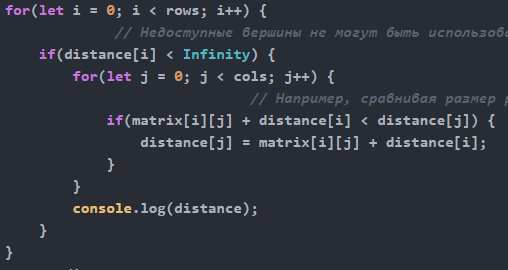
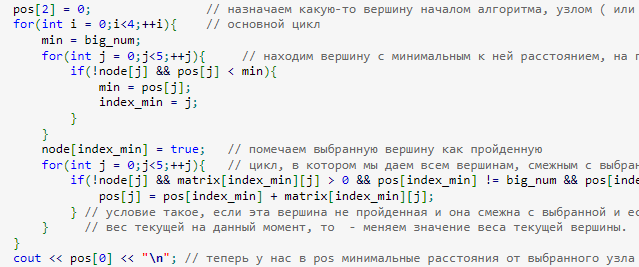


Рисунок 26 - Фрагмент кода программы из интернета, реализующий алгоритм Дейкстры, написанный на ЯП JavaScript.



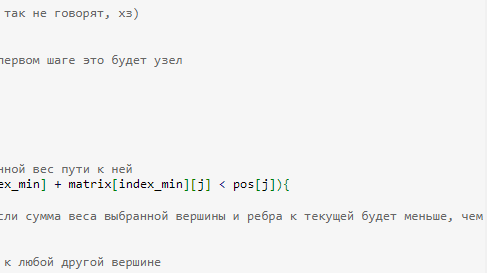


Рисунок 27 - Фрагмент кода программы из интернета, реализующий алгоритм Дейкстры, написанный на ЯП C++

Анализируя и вычислив время работы алгоритмов по концепции Big O, представленных на рисунках 24, 25, 26 и 27 можно сделать вывод, что время действия всех алгоритмов в общем виде равно O(n²).

Сравнение программ по расходуемой памяти

На рисунках 19 и 20 представлены патенты. На рисунке 19 программа реализована на ЯП С++, которая считает и выводит только кратчайший путь по алгоритму Дейкстры. Расходуемая память 207 Кб. На рисунке 20 программа реализована на ЯП С#, которая ищет кратчайший путь по алгоритму Дейкстры, а также рисует граф по матрице весов. Расходуемая память 1,20 Мб. Программа, реализованная при выполнении курсового проекта выводит алгоритм Дейкстры, исходную матрицу, маршрут, длину кратчайшего пути, а также рисует граф по матрице весов. Расходуемая память 107 Кб. А созданный файл с разрешением .jar занимает 207 Кб, что на порядок лучше, чем программа реализованная на ЯП C#, которая выполняет такие же функции, что и программа реализованная при выполнении курсовой работы на ЯП Java. Расход памяти программ на ЯП C# и Java различается не сильно, но значительно отличается их функционал, что указывает на превосходство программы, реализованной на ЯП Java, по сравнению с программами конкурентами.

Анализ объёма программы

Программа содержит 326 строк в классе Deikstra и 146 строк кода в классе DrawGraph. Итого 472 строки кода. Сам алгоритм Дейкстры занимает 30 строк кода, что составляет около 6% от всей программы. Благолепие вывода занимает около 275 строк, что составляет около 58%. Отображение графа занимает около 145 строк, что составляет примерно 30% программы. Всё остальное(подключение пакетов, библиотек, скобки тел класса и т.д.) составляет приблизительно 6%. Т.о. можно сделать вывод, что приятный интерфейс занимает 58%, т.е. на это было потрачено более половины строк кода, что объясняет достаточно большой вес программы при достаточно коротком алгоритме, а также 30% занимает визуализация графа. Итого 88% строк было потрачено на приятный интерфейс, а также комфортабельность и практичность ввода и вывода для пользователя.

Разработка и выбор иконки программы

Было разработано более 20 иконок для программы, среди которых были отобраны 3 наиболее подходящих и изящных, которые изображены на рисунках 28, 29, 30.



Рисунок 28 – Первая иконка программы



Рисунок 29 – Вторая иконка программы

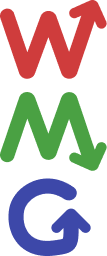


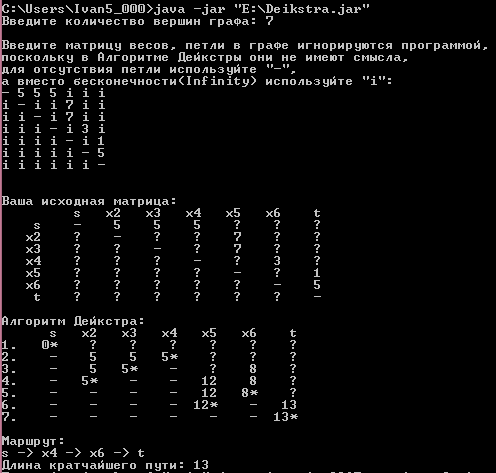
Рисунок 30 – Третья иконка программы

На данный момент, при запуске программы, отображается иконка изображённая на рисунке 28.

Создание JAR-файла и его запуск

Был создан файл типа .jar, что даёт возможность запустить этот файл с любого компьютера и любой ОС, где установлена Java или JDK.

Запуск был осуществлён с помощью командной строки.



Плюсы и минусы программы

Плюсы:

1. Программа достаточно оптимально вычисляет кратчайший путь по алгоритму Дейкстры.
2. Реализован приятный интерфейс для пользователя
3. Программа красочно рисует граф по матрице весов
4. Программа протестирована на нестандартные ситуации (цикл, петли и т.д.).
5. Создан файл с разрешением .jar
6. Разработана авторская иконка.
7. Реализована перерисовка графа.

Минусы:

1. Программа достаточно объёмная
2. Почти не используются основные принципы ООП

**Заключение**

В ходе работы были выполнены поставленные задачи в полном объёме, а также была достигнута поставленная цель. Реализация программы оказалась достаточно успешной, что подтверждают сравнения с программами конкурентами. Также в ходе работы задействовалось достаточно большое количество ресурсов (временных, литературных и т.д.). Был произведён огромный обзор литературы, отобраны самые значимые источники для выполнения данной работы. Найденная информация, полученные и имеющиеся знания в ходе работы были достаточно сильно сжаты и компактно, но понятно изложены.

В дальнейшем планируется доработать программу до “идеала”.

Рекомендации:

1. Реализовать и внедрить алгоритм топологической сортировки
2. Переработать программу в соответствии с основными принципами ООП.
3. Выделить отображение кратчайшего пути при визуализации графа
4. Нивелировать выше изложенные минусы.
5. Сделать содержание курсовой работы более подробным.
6. Реализовать полноценное приложение, которое можно будет запатентовать.

**Список литературы**

1. Томас Х. Кормен, Чарльз И. Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн. Алгоритмы: построение и анализ = Introduction to Algorithms. — 2-е изд. — М.: [«Вильямс»](https://ru.wikipedia.org/w/index.php?title=%D0%92%D0%B8%D0%BB%D1%8C%D1%8F%D0%BC%D1%81_(%D0%B8%D0%B7%D0%B4%D0%B0%D1%82%D0%B5%D0%BB%D1%8C%D1%81%D1%82%D0%B2%D0%BE)&action=edit&redlink=1), 2006. — С. 1296. — [ISBN 0-07-013151-1](https://ru.wikipedia.org/wiki/%D0%A1%D0%BB%D1%83%D0%B6%D0%B5%D0%B1%D0%BD%D0%B0%D1%8F:%D0%98%D1%81%D1%82%D0%BE%D1%87%D0%BD%D0%B8%D0%BA%D0%B8_%D0%BA%D0%BD%D0%B8%D0%B3/0070131511).
2. [Левитин А. В.](https://www.wikidata.org/wiki/Q21694518) Глава 9. Жадные методы: Алгоритм Дейкстры // [Алгоритмы. Введение в разработку и анализ](https://www.wikidata.org/wiki/Q21694522) — М.: [Вильямс](https://www.wikidata.org/wiki/Q21694521), 2006. — С. 189—195. — 576 с. — [ISBN 978-5-8459-0987-9](https://ru.wikipedia.org/wiki/%D0%A1%D0%BB%D1%83%D0%B6%D0%B5%D0%B1%D0%BD%D0%B0%D1%8F:%D0%98%D1%81%D1%82%D0%BE%D1%87%D0%BD%D0%B8%D0%BA%D0%B8_%D0%BA%D0%BD%D0%B8%D0%B3/9785845909879)
3. [Dijkstra E. W.](https://ru.wikipedia.org/wiki/%D0%94%D0%B5%D0%B9%D0%BA%D1%81%D1%82%D1%80%D0%B0,_%D0%AD%D0%B4%D1%81%D0%B3%D0%B5%D1%80_%D0%92%D0%B8%D0%B1%D0%B5) [A note on two problems in connexion with graphs](http://www-m3.ma.tum.de/foswiki/pub/MN0506/WebHome/dijkstra.pdf) (англ.) // [Numer. Math](https://en.wikipedia.org/wiki/Numerische_Mathematik) / [F. Brezzi](https://en.wikipedia.org/wiki/Franco_Brezzi) — [Springer Science+Business Media](https://ru.wikipedia.org/wiki/Springer_Science%2BBusiness_Media), 1959. — Vol. 1, Iss. 1. — P. 269—271. — ISSN [0029-599X](https://www.worldcat.org/issn/0029-599X); [0945-3245](https://www.worldcat.org/issn/0945-3245) — [doi:10.1007/BF01386390](https://dx.doi.org/10.1007/BF01386390)
4. А.В. Кухарев, Е.А. Витько, А.А. Царев. Алгоритмы на графах. Витебск ВГУ имени П.М. Машерова 2016.
5. Индекс TIOBE. [Электронный ресурс] URL: <https://www.tiobe.com/tiobe-index/> (дата обращения 04.12.2021).
6. Д. В. Мусатов СЛОЖНОСТЬ ВЫЧИСЛЕНИЙ 2016.

**Приложения**

1. Файл типа .jar в электронном виде
2. Электронная версия курсовой работы
3. Презентация курсовой работы
4. Файл с несколькими готовыми тестами по алгоритму Дейкстры.