МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

НИЖЕГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ им. Р.Е.АЛЕКСЕЕВА

Институт радиоэлектроники и информационных технологий

Кафедра «Вычислительные системы и технологии»

Курсовая работа на тему

Сравнение алгоритмов поиска кратчайшего пути на графах.

по дисциплине

Функциональное и логическое программирование

РУКОВОДИТЕЛЬ:

Суркова А.С.

СТУДЕНТ:

Алексеев М.Д.

гр. 21-ПО

Работа защищена « »

С оценкой

Нижний Новгород 2024

Оглавление

[Введение 3](#_Toc167792407)

[Описание рассматриваемых алгоритмов 3](#_Toc167792408)

[Базовые определения 3](#_Toc167792409)

[Алгоритм Беллмана-Форда 5](#_Toc167792410)

[Алгоритм Флойда-Уоршелла 6](#_Toc167792411)

[Выбор языка программирования 7](#_Toc167792412)

[Описание логики кода 7](#_Toc167792413)

[Выбор примеров для сравнения результатов 8](#_Toc167792414)

[Сравнение результатов работы алгоритмов 20](#_Toc167792415)

[Вывод 21](#_Toc167792416)

[Список литературы 21](#_Toc167792417)

[Приложение 22](#_Toc167792418)

[Алгоритм Дейкстры 22](#_Toc167792419)

[Алгоритм Флойда-Уоршелла 23](#_Toc167792420)

[Алгоритм Форда-Беллмана 24](#_Toc167792421)

# Введение

Актуальность темы исследования обусловлена тем, что алгоритмы поиска кратчайшего пути на графах являются важными инструментами в широком спектре областей, включая информационные технологии, компьютерные науки, инженерное дело и даже биологию. Они используются для определения оптимальных маршрутов в навигационных системах, оптимизации сетей связи, анализа социальных сетей и многого другого.

Цель данной курсовой работы — сравнить эффективность и применимость нескольких популярных алгоритмов поиска кратчайшего пути на графах. Для достижения поставленной цели необходимо решить следующие задачи:

Изучить теоретические основы алгоритмов поиска кратчайшего пути на графах.

Провести анализ эффективности и применимости каждого из рассматриваемых алгоритмов.

Проиллюстрировать результаты сравнения на практических примерах.

Сделать выводы о преимуществах и недостатках каждого из алгоритмов и предложить рекомендации по их использованию в зависимости от конкретных условий.

# Описание рассматриваемых алгоритмов

# Базовые определения

Граф — это структура данных, представляющая собой совокупность вершин и рёбер. Вершины — это точки, а рёбра — это линии, соединяющие эти точки. Рёбра могут быть ориентированными (то есть иметь направление) и неориентированными. Неориентированный граф не имеет направлений, а ориентированный — имеет. Граф связный, если от каждой вершины можно дойти до любой другой по этим отрезкам.

Граф называется взвешенным, если каждому ребру соответствует какое-то число (вес). Не может быть двух рёбер, соединяющих одни и те же вершины.

В данной работе, будут сравниваться алгоритмы поиска пути на нескольких неориентированных, связных и взвешенных графах.

Каждый из алгоритмов будет решать какую-то задачу о кратчайших путях на взвешенном связном. Кратчайший путь из одной вершины в другую — это такой путь по рёбрам, что сумма весов рёбер, по которым мы прошли будет минимальна.

В данной работе будут рассматриваться самые популярные алгоритмы поиска пути:

#### Алгоритм Дейкстры

Алгоритм поиска кратчайшего пути разработал голландский учёный Эдсгер Дейкстра в 1956 году. В то время он искал способ продемонстрировать возможности нового компьютера ARMAC и искал задачу, которую мог бы решить ARMAC и при этом понятную незнакомым с компьютерами людям.

Дейкстра взял задачу поиска кратчайшего пути и разработал алгоритм её решения. На базе алгоритма он разработал программу построения маршрутов между городами по транспортной карте Нидерландов.

Алгоритм Дейкстры начинается с установки начальной вершины и работы от этой точки. Он работает по принципу «жадного» алгоритма, что означает, что на каждом шаге он стремится минимизировать текущую общую стоимость пути.

Сначала инициализируются два массива:

Массив, содержащее уже обработанные вершины (изначально пустое).

Массив, содержащее все остальные вершины графа (изначально содержит все вершины графа).

Также каждой вершине графа присваивается вес, который представляет минимальную известную стоимость пути от начальной вершины до данной. Для начальной вершины этот вес равен 0, для всех остальных вершин — бесконечность или любое число, заведомо больше максимального возможного расстояния.

На каждом шаге алгоритм выбирает вершину из непосещенного массива с наименьшим весом, перемещает эту вершину в массив посещенных вершин и обновляет веса всех соседей выбранной вершины. Вес соседа обновляется, если через выбранную вершину можно добраться до этого соседа с меньшей стоимостью.

Процесс продолжается, пока не будут посещены все вершины или пока мы не найдем путь до конечной вершины (если она задана).

Ограничения алгоритма

Алгоритм Дейкстры идеально подходит для ситуаций, заранее неизвестна конечная точка. Он вычисляет кратчайшее расстояние от исходной точки до всех остальных вершин в графе. Это делает его идеальным для задач, где важно знать расстояние до всех точек, а не только до конечной точки. В следствие из этого Алгоритм Дейкстры, строит кратчайший путь, но не всегда делает это наиболее эффективным образом. Он обрабатывает все вершины в графе, что может привести к излишнему использованию ресурсов, особенно в больших графах. При этом Алгоритм Дейкстры работает только с графами, у которых веса всех ребер положительны. Он не способен корректно обработать графы с отрицательными весами ребер, что может быть ограничением в некоторых приложениях.

# Алгоритм Беллмана-Форда

Алгоритм делится на 3 основных шага:

Первый шаг – создается массив размера равного количеству вершин в графе, в котором будут инициализированы расстояния от исходной вершины до всех остальных как бесконечные или очень большие. Расстояние до первоначальной вершины будет равно нулю.

Второй шаг – производятся действия для каждого ребра между вершинами. Сравнивается значения в массиве конечной точки ребра и изначальной точки ребра + вес самого ребра. Если конечная точка ребра больше, тогда число, лежащее в массиве, обновляется на величину равную изначальной точке + вес самого ребра.

Третий шаг – проверяется, есть ли в графе цикл отрицательного веса, для каждого ребра необходимо проверить, больше ли значение конечной точки ребра изначальной + вес ребра. Данная проверка должна быть произведена после присвоения нового значения, то есть второго шага.

Второй и третий шаг повторяются n-1 раз, где n – количество вершин в графе.

Алгоритм Беллмана-Форда работает лучше для распределенных систем (лучше, чем алгоритм Дейкстры). В отличие от Дейкстры, где нам нужно найти минимальное значение всех вершин, в Беллмане-Форде ребра рассматриваются по одному.

Также, этот алгоритм может работать с отрицательными весами, что ставит его выше чем алгоритм Дейкстры.

# Алгоритм Флойда-Уоршелла

Данный алгоритм подходит, если нужно найти кратчайшие пути из каждой вершины в другую каждую вершину. Он работает с отрицательными весами путей, а также может использоваться на ориентированных графах.

Суть алгоритма заключается в получении из матрицы смежности путем обновления данных матрицы, в которой будут расписаны кратчайшие пути.

Алгоритм делится на несколько этапов:

* Сначала строится матрица смежности для данного графа, но в ячейках, где две вершины не имеют прямого пути между собой ставится либо бесконечность, либо очень большое число, а в ячейках, где вершина сопоставляется с этой же вершиной, то есть на главной диагонали, ставятся нули.
* Второй этап состоит из операций над данной матрицей. Берем первый пункт, который есть в матрице и разрешаем проводить путь через него. Дальше сравниваем пути от каждого пункта к другом в изначальной матрице и возможный путь через данную вершину. Если путь через данную вершину оказывается меньше, чем в предыдущем случае, то заменяем значение в матрице.
* После берем следующую вершину и проводим над ней такие же операции

Второй этап повторяется столько раз, сколько вершин находится в графе. В итоге получается матрица, в которой указаны минимальные расстояния от каждой вершины до каждой другой.

# Выбор языка программирования

Перед началом написания курсовой работы, я имел возможность выбрать один из трёх языков программирования: Scala, Haskell, Prolog. В качестве языка программирования был выбран Scala, так как Prolog является логическим языком программирования, и программа в нем строится от предикатов, что не очень удобно. Язык Haskell же я вообще не знаю и это для меня темный лес.

Scala же, в свою очередь имеет несколько плюсов по сравнению с этими двумя языками:

Выразительность: Scala предлагает богатый набор функциональных конструкций, таких как функции высшего порядка, неизменяемые структуры данных, рекурсия и т.д. Это позволяет студентам глубже понять особенности функционального программирования.

Широкая поддержка: Scala имеет активное сообщество и обширную документацию, что упрощает изучение и разработку проектов на этом языке.

Использование в индустрии: Scala широко используется в индустрии, особенно в крупных корпоративных приложениях и Big Data решениях, что делает знание этого языка полезным для будущей карьеры.

Интеграция с Java: Scala работает на JVM и без проблем интегрируется с существующими Java-библиотеками и инструментами, что упрощает использование его в реальных проектах.

# Описание логики кода

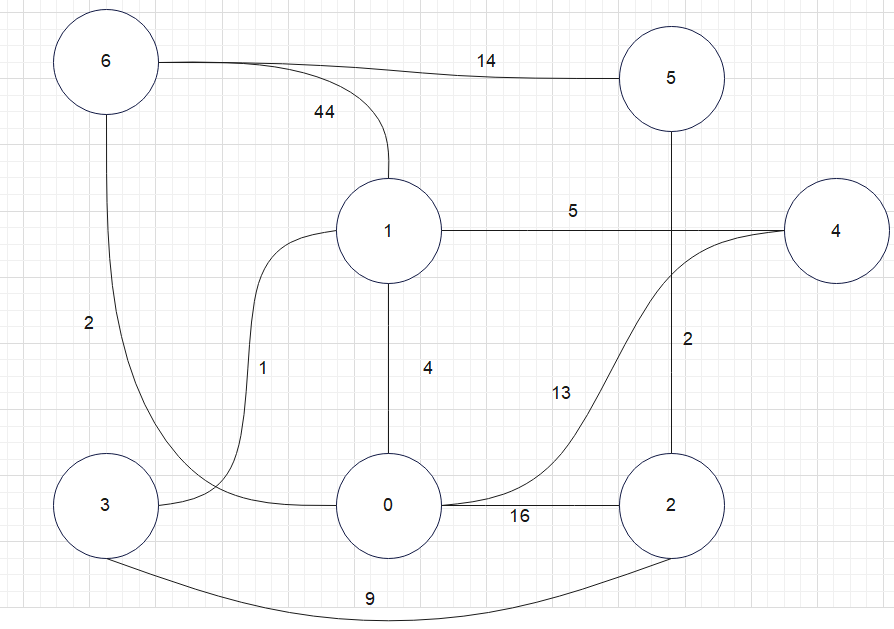
Код во всех трех алгоритмах будет выглядеть идентично. Нужно исследовать работоспособность алгоритмов на разных видах графов, а также количество операций, необходимых для дачи ответа на поставленную задачу.

Код данных алгоритмов состоит из одной функции по расчету, главной функции, где запускается алгоритм и константы. В константе graph будет храниться двумерный массив, который является матрицей смежности графа. Была выбрана именно матрица смежности, так как это самый удобный способ представления графа в программе. При этом главную диагональ графа следует заполнять нулями, так как я беру графы без петель, а бесконечные пути, то есть пути которые напрямую не соединяют вершины нужно заполнять числом, которое априори больше всех путей, допустим 20000.

# Выбор примеров для сравнения результатов

В качестве примеров для сравнения результатов работы я нарисовал несколько разных графов, чтобы точно определить область применения и быстроту работы каждого из алгоритмов. Также у каждого графа, вручную, с помощью обхода были заранее записаны правильные ответы, для проверки корректности работы алгоритмов.

* Взвешенный неориентированный граф, с положительными весами путей

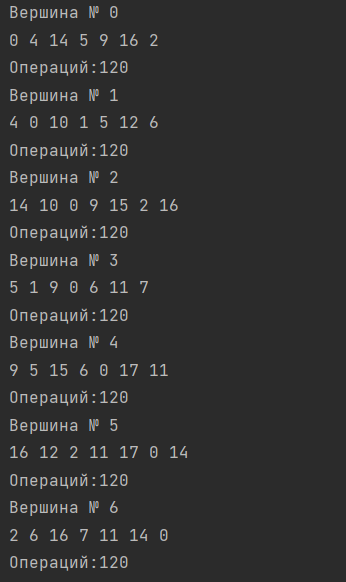


Матрица смежности:

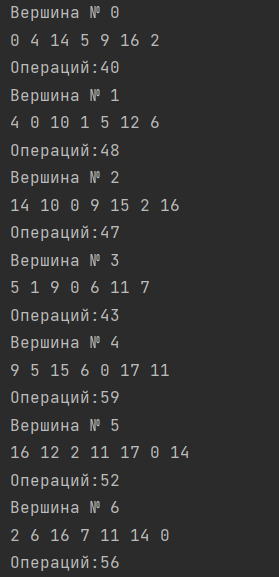
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| 0 | 0 | 4 | 16 | 20000 | 13 | 20000 | 2 |
| 1 | 4 | 0 | 20000 | 1 | 5 | 20000 | 44 |
| 2 | 16 | 20000 | 0 | 9 | 20000 | 2 | 20000 |
| 3 | 20000 | 1 | 9 | 0 | 20000 | 20000 | 20000 |
| 4 | 13 | 5 | 20000 | 20000 | 0 | 20000 | 20000 |
| 5 | 20000 | 20000 | 2 | 20000 | 20000 | 0 | 14 |
| 6 | 2 | 44 | 20000 | 20000 | 20000 | 14 | 0 |

Результаты:

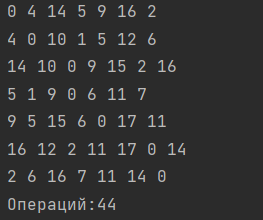
Дейкстра



Форд-Беллман

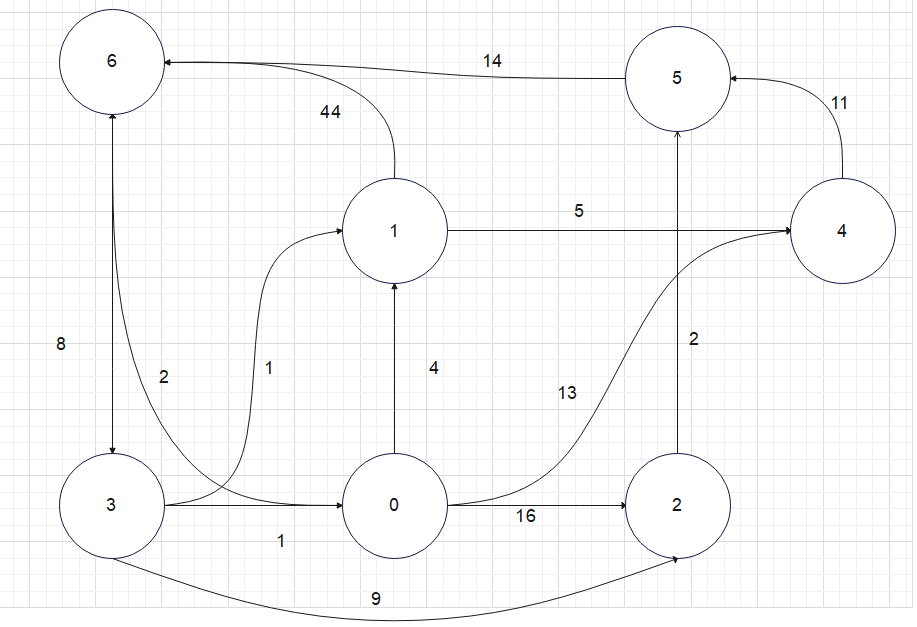


Флойд-Уоршелл



Все алгоритмы отработали корректно

* Взвешенный ориентированный граф, с положительными весами путей

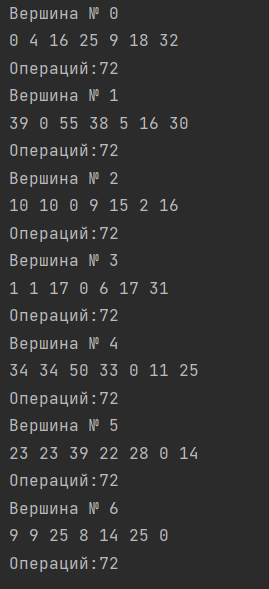


Матрица смежности:

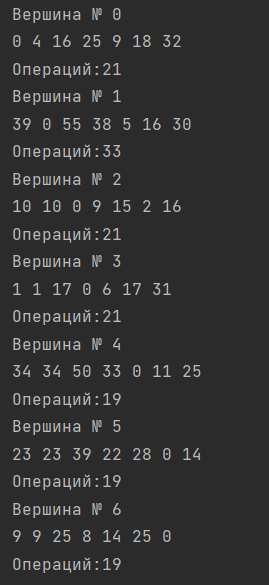
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| 0 | 0 | 4 | 16 | 20000 | 13 | 20000 | 20000 |
| 1 | 20000 | 0 | 20000 | 20000 | 5 | 20000 | 44 |
| 2 | 20000 | 20000 | 0 | 9 | 20000 | 2 | 20000 |
| 3 | 1 | 1 | 20000 | 0 | 20000 | 20000 | 20000 |
| 4 | 20000 | 20000 | 20000 | 20000 | 0 | 11 | 20000 |
| 5 | 20000 | 20000 | 20000 | 20000 | 20000 | 0 | 14 |
| 6 | 20000 | 20000 | 20000 | 8 | 20000 | 20000 | 0 |

Результаты:

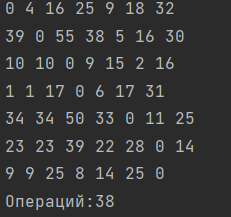
Дейкстра



Форд-Беллман

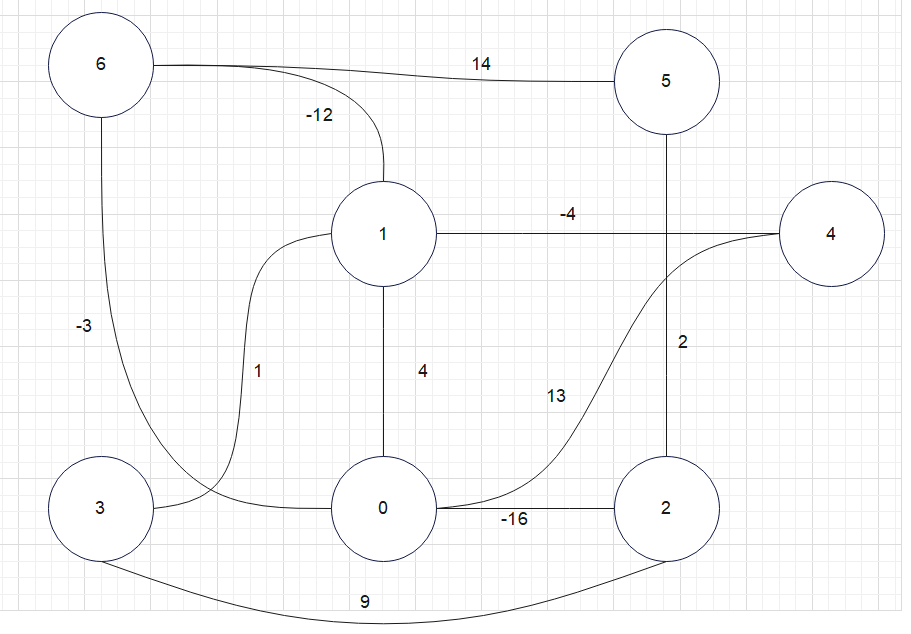


Флойд-Уоршелл



Все алгоритмы отработали корректно

* Взвешенный неориентированный граф, с отрицательными весами путей

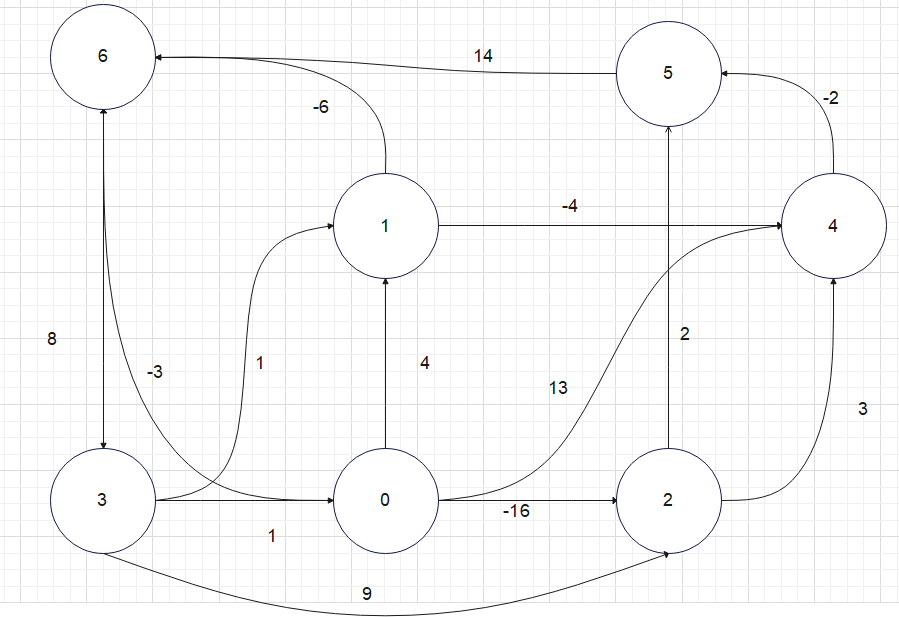


Матрица смежности:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| 0 | 0 | 4 | -16 | 20000 | 13 | 20000 | -3 |
| 1 | 4 | 0 | 20000 | 1 | -4 | 20000 | -12 |
| 2 | -16 | 20000 | 0 | 9 | 20000 | 2 | 20000 |
| 3 | 20000 | 1 | 9 | 0 | 20000 | 20000 | 20000 |
| 4 | 13 | -4 | 20000 | 20000 | 0 | 20000 | 20000 |
| 5 | 20000 | 20000 | 2 | 20000 | 20000 | 0 | 14 |
| 6 | -3 | -12 | 20000 | 20000 | 20000 | 14 | 0 |

На таком графе задачу о нахождении минимального пути невозможно решить, так как даже между двумя соседними вершинами будет бесконечный отрицательный цикл.

* Взвешенный ориентированный граф, с отрицательными весами путей

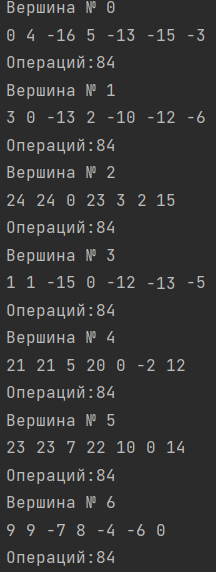


Матрица смежности:

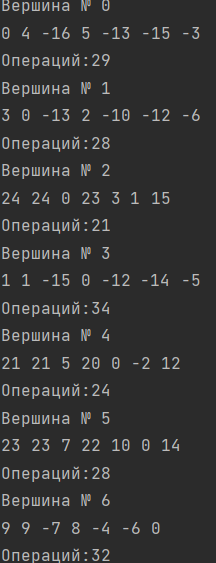
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| 0 | 0 | 4 | -16 | 20000 | 13 | 20000 | -3 |
| 1 | 20000 | 0 | 20000 | 20000 | -4 | 20000 | -6 |
| 2 | 20000 | 20000 | 0 | 20000 | 3 | 2 | 20000 |
| 3 | 1 | 1 | 9 | 0 | 20000 | 20000 | 20000 |
| 4 | 20000 | 20000 | 20000 | 20000 | 0 | -2 | 20000 |
| 5 | 20000 | 20000 | 20000 | 20000 | 20000 | 0 | 14 |
| 6 | 20000 | 20000 | 20000 | 8 | 20000 | 20000 | 0 |

Результаты:

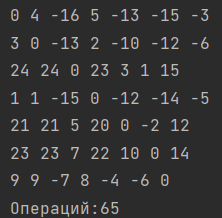
Дейкстра



Форд-Беллман



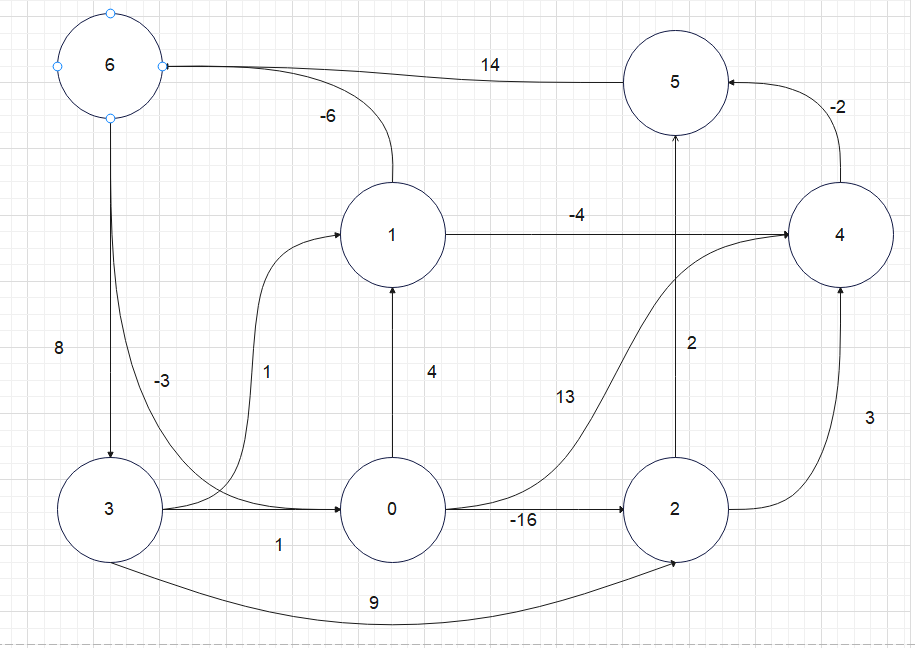
Флойд-Уоршелл



Как видно, все алгоритмы выдали ответ, но если присмотреться, в некоторых местах алгоритма Дейкстры, а именно вершине 2 и 3 есть ошибки. Связаны они с тем, что путь из вершины 2 в 5 короче через промежуточную вершину 4, но алгоритм откидывает этот вариант, беря во внимание меньшее число из двух путей 2-5 или 2-4.

* Взвешенный ориентированный граф, с бесконечным циклом и с отрицательными весами путей

Изменю этот граф, чтобы путь был из вершины 6 в 0, а не наоборот, тогда по логике цикл 0-1-6-6 будет бесконечно отрицательным.



Матрица смежности:

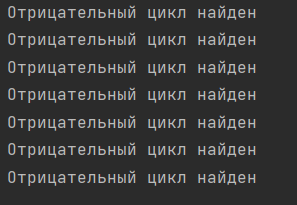
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| 0 | 0 | 4 | -16 | 20000 | 13 | 20000 | 20000 |
| 1 | 20000 | 0 | 20000 | 20000 | -4 | 20000 | -6 |
| 2 | 20000 | 20000 | 0 | 20000 | 3 | 2 | 20000 |
| 3 | 1 | 1 | 9 | 0 | 20000 | 20000 | 20000 |
| 4 | 20000 | 20000 | 20000 | 20000 | 0 | -2 | 20000 |
| 5 | 20000 | 20000 | 20000 | 20000 | 20000 | 0 | 14 |
| 6 | -3 | 20000 | 20000 | 8 | 20000 | 20000 | 0 |

Результаты:

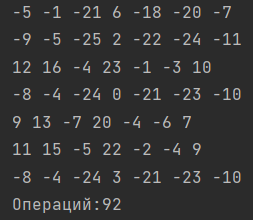
Дейкстра

Алгоритм Дейкстры уходит в бесконечный цикл и загрузку, ничего не выводя.

Форд-Беллман



Флойд-Уоршелл



Как видно ни один из алгоритмов корректно не сработал. В алгоритме Дейкстры образовался бесконечный цикл, в алгоритме Форда-Беллмана не прошла проверка на отрицательный цикл, в связи с чем выдало предупреждение, а алгоритме же Флойда-Уоршелла все величины посчитались очень странно, а на главное диагонали образовались отрицательные числа.

# Сравнение результатов работы алгоритмов

В данной таблице расписаны каждые алгоритмы на каждом примере по критериям.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Пример | Дейкстры | Форда-Беллмана | Флойда-Уоршелла | Критерии |
| Взвешенный неориентированный граф, с положительными весами путей | 120  +  - | 49,2  +  - | 44  +  - | Быстродействие  Корректность  Выдача ошибки |
| Взвешенный ориентированный граф, с положительными весами путей | 72  +  - | 21,8  +  - | 38  +  - | Быстродействие  Корректность  Выдача ошибки |
| Взвешенный неориентированный граф, с отрицательными весами путей | -------------------------- | -------------------------- | ---------------------------- |  |
| Взвешенный ориентированный граф, с отрицательными весами путей | 84  -  - | 28  +  - | 65  +  - | Быстродействие  Корректность  Выдача ошибки |
| Взвешенный ориентированный граф, с отрицательными весами путей и бесконечным циклом |  | + | 92  +-  - | Быстродействие  Корректность  Выдача ошибки |

Как видно из эксперимента, в большинстве случаев алгоритм Форда-Беллмана требует меньше операций для выполнения поставленной задачи, если в графе нет бесконечного отрицательного цикла. Самый же медленный алгоритм – это алгоритм Дейкстры. Алгоритм Флойда-Уоршелла же в свою очередь может работать на ориентированных графах с бесконечным отрицательным циклом, а также сразу показывает матрицу кратчайших путей, то есть для него не нужно указывать начальную точку.

# Вывод

В ходе проведенной работы и поставленного эксперимента, были разработаны программы для осуществления алгоритмов поиска кратчайших путей на графах. Было экспериментально подтверждены ограничения и функциональность каждого алгоритма при его работе.

# Список литературы

<https://habr.com/ru/articles/119158/>

<https://habr.com/ru/articles/330816/>

<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>

<https://habr.com/ru/companies/otus/articles/484382/>

https://habr.com/ru/articles/105825/

# Приложение

# Алгоритм Дейкстры

import scala.collection.mutable  
object main {  
 var *Operations* = 0  
 def dijkstra(graph: Array[Array[Int]], start: Int) = {  
 // Размерность графа  
 val n = graph.length  
 // Массив для хранения расстояний от стартовой вершины  
 val distances = Array.*fill*(n)(20000)  
 distances(start) = 0  
  
 // Очередь с приоритетами для выбора вершины с минимальным расстоянием  
 val priorityQueue = mutable.PriorityQueue[(Int, Int)]((0, start))  
  
 // Перебираем вершины  
 while (priorityQueue.nonEmpty) {  
 // Извлекаем вершину с минимальным расстоянием  
 val (distance, current) = priorityQueue.dequeue()  
  
 // Если текущее расстояние больше уже известного, пропускаем эту вершину  
 if (distance <= distances(current)) {  
  
 // Перебираем соседей текущей вершины  
 for (neighbor <- 0 until n if graph(current)(neighbor) != 20000) {  
 *Operations* += 1  
 // Рассчитываем новое расстояние  
 val newDistance = distance + graph(current)(neighbor)  
 // Если новое расстояние меньше известного, обновляем известное расстояние и добавляем в очередь  
 if (newDistance < distances(neighbor)) {  
 distances(neighbor) = newDistance  
 priorityQueue.enqueue((newDistance, neighbor))  
  
 }  
  
 }  
 }  
  
 }  
  
 // Возвращаем массив расстояний  
 distances  
 }  
  
 // Пример использования  
 val *graph* = *Array*(  
 *Array*(),  
 *Array*(),  
 *Array*(),  
 )  
  
  
 def main(args: Array[String]): Unit = {  
 for (i <- 0 until 7) {  
 val shortestPaths = *dijkstra*(graph, i)  
 val n = shortestPaths.length  
 *println*("Вершина № " + i)  
 for (k <- 0 until n) {  
 *print*(shortestPaths(k))  
 *print*(" ")  
 }  
 *println*("")  
 *println*("Операций:" + *Operations*)  
 *Operations* = 0  
 }  
 }  
}

# Алгоритм Флойда-Уоршелла

import scala.collection.mutable  
object main {  
 var *Operations* = 0  
 def floydWarshall(matrix: Array[Array[Int]]) = {  
 // Размерность графа  
 val n = matrix.length  
  
 // Копируем исходную матрицу смежности в матрицу расстояний  
 val dist = Array.*ofDim*[Int](n, n)  
 for (i <- 0 until n; j <- 0 until n) {  
 dist(i)(j) = matrix(i)(j)  
 }  
  
 // Алгоритм Флойда-Уоршелла  
 for (k <- 0 until n) {  
 for (i <- 0 until n) {  
 for (j <- 0 until n) {  
 if (dist(i)(k) + dist(k)(j) < dist(i)(j)) {  
 dist(i)(j) = dist(i)(k) + dist(k)(j)  
 *Operations* += 1  
 }  
 }  
 }  
 }  
 dist  
 }  
 // Пример использования  
 val *graph* = *Array*(  
 *Array*(),  
 *Array*(),  
 *Array*(),  
 )  
  
 def main(args: Array[String]): Unit = {  
 val shortestPaths = *floydWarshall*(graph)  
 val n = shortestPaths.length  
 for (k <- 0 until n) {  
 for (i <- 0 until n) {  
 *print*(shortestPaths(k)(i))  
 *print*(" ")  
 }  
 *println*("")  
 }  
 *println*("Операций:" + *Operations*)  
 }  
  
}

# Алгоритм Форда-Беллмана

object main {  
 var *Operations* = 0  
 def bellmanFord(graph: Array[Array[Int]], source: Int): Option[(Array[Int], Array[Int])] = {  
 // Размерность графа  
 val n = graph.length  
  
 // Инициализация массива расстояний и предшественников  
 var distances = Array.*fill*(n)(20000)  
 var predecessors = Array.*fill*(n)(-1)  
  
 // Расстояние от источника до самого себя равно 0  
 distances(source) = 0  
  
 // Релаксация рёбер  
 for (\_ <- 0 until n - 1) {  
 for (u <- 0 until n) {  
 for (v <- 0 until n if u != v && graph(u)(v) != 20000) {  
 *Operations* += 1  
 val newDist = distances(u) + graph(u)(v)  
 if (newDist < distances(v)) {  
 distances(v) = newDist  
 predecessors(v) = u  
 }  
 }  
 }  
 }  
  
 // Поиск отрицательного цикла  
 for (u <- 0 until n) {  
 for (v <- 0 until n if u != v && graph(u)(v) != 20000) {  
 val newDist = distances(u) + graph(u)(v)  
 if (newDist < distances(v)) {  
 return None  
 }  
 }  
 }  
  
 // Возвращаем кортеж с массивом расстояний и массивом предшественников  
 *Option*((distances, predecessors))  
 }  
  
 // Пример использования  
 val *graph* = *Array*(  
 *Array*(),  
 *Array*(),  
 *Array*(),  
 )  
  
 def main(args: Array[String]): Unit = {  
 for (i <- 0 until 7) {  
 val result = *bellmanFord*(graph, i)  
 if (!result.isEmpty) {  
 val (distances, predecessors) = result.get  
 *println*("Вершина № " + i)  
 for (k <- 0 until distances.length) {  
 *print*(distances(k))  
 *print*(" ")  
 }  
 *println*("")  
 *println*(*Operations*)  
 *Operations* = 0  
 } else {  
 *println*("Отрицательный цикл найден")  
 *Operations* = 0  
 }  
 }  
 }  
}