|  |  |  |
| --- | --- | --- |
| *логотип -1501ч-б* | **Департамент образования города Москвы**  **Государственное бюджетное образовательное учреждение города Москвы** | gerbmoscow2 |

**МНОГОПРОФИЛЬНЫЙ ТЕХНИЧЕСКИЙ ЛИЦЕЙ № 1501**

*X* *Городская научно-практическая техническая конференция школьников*«Исследуем и проектируем»  
(место проведения Многопрофильный технический  
лицей №1501)

«Реализация алгоритмов поиска кратчайшего и оптимального маршрутов в компьютерных сетях»

Автор: ***Болашев Кирилл Вячеславович***

### класс: *11-2*

### Научный руководитель:

### *Глубоков Александр Владимирович, к.т.н.* МГТУ «Станкин»

## г. Москва

2012—2013 учебный год

Оглавление

[Введение 3](#_Toc349724772)

[Как работают компьютерные сети. 3](#_Toc349724773)

[Протокол пакетной передачи данных Ethernet 3](#_Toc349724774)

[Стек протоколов TCP/IP. 4](#_Toc349724775)

[Internet Protocol (IP) 4](#_Toc349724776)

[Аппаратные устройства. 5](#_Toc349724777)

[Протоколы динамической маршрутизации 7](#_Toc349724778)

[Routing Infrormation Protocol 7](#_Toc349724779)

[Open Shortest Path First 8](#_Toc349724780)

[Графы. 8](#_Toc349724781)

[Класс Edge 10](#_Toc349724782)

[Класс Node 10](#_Toc349724783)

[Класс Graph 10](#_Toc349724784)

[Реализация алгоритмов 11](#_Toc349724785)

[Алгоритм Беллмана-Форда 11](#_Toc349724786)

Алгоритм Дейкстры…………………………………………………………………………………………………………………………………………11

[Алгоритм Флойда-Уоршелла. 16](#_Toc349724787)

[Интерфейс программы и системные требования 18](#_Toc349724788)

[Выводы 19](#_Toc349724789)

[Список литературы 20](#_Toc349724790)

# Введение

В настоящее время компьютерные сети стали неотъемлемой частью нашей жизни, мы встречаем их повсюду – провайдер, проводящий интернет до дома, по сути, оперирует большой компьютерной сетью; небольшая домашняя сеть из ноутбука и «основного» настольного компьютера, с которого мы передаем данные с одного на другой – маленькая компьютерная сеть внутри большой сети провайдера. Корпоративные сети компаний, беспроводные сети в кафе и, конечно же, сеть, объединяющая большинство этих, более маленьких сетей, Интернет – все эти вещи так или иначе соединяют какое-то количество компьютеров в одну сеть, где совершается обмен пакетами между компьютерами.

***Пакет*** *– единица данных в большинстве ныне существующих сетей. Является специально оформленным блоком данных, несущих в себе, помимо данных пользователя, еще и управляющую информацию, позволяющую определить отправителя, адресата и пр.*

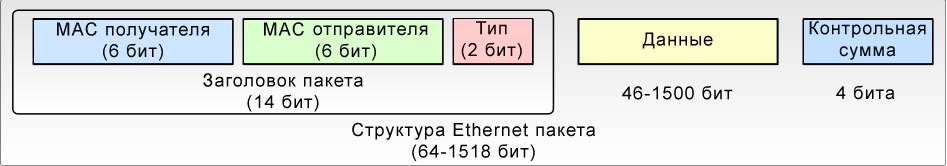
Маршрутизация – один из основных аспектов существования нынешних сетей, так как именно она обеспечивает верную доставку данных до получателя. В своем проекте я рассмотрел ныне используемые протоколы динамической маршрутизации, с помощью каких алгоритмов они находят путь, по которому надо доставить пакет, и реализовал эти алгоритмы в отдельной программе, написанной на Java. Но прежде чем перейти к самой программе, давайте разберемся, как же работают нынешние компьютерные сети.

# 

# Как работают компьютерные сети.

## Протокол пакетной передачи данных Ethernet

В данный момент «королем» проводной передачи данных можно считать протокол Ethernet, разработанный компанией Xerox в 1973 году и ныне являющегося самой распространенной технологией пакетной передачи данных. Предусматривает наличие у каждой сетевой карты уникального 6-байтного MAC-адреса, прошитого в ПЗУ карты непосредственно производителем, позволяющего легко различать сетевые карты между собой.



*Рис 1.1 Структура Ethernet пакета*

Когда в нашем распоряжении находятся всего два компьютера, подключенные напрямую, нам даже необязательно знать MAC-адрес отправителя, компьютеры обмениваются этими пакетами друг с другом напрямую. Однако, если нам понадобится соединить три и более компьютеров вместе, нам понадобится использовать как новую аппаратуру, так и новые протоколы.

## Стек протоколов TCP/IP.

Стек протоколов TCP/IP содержит в себе набор протоколов, взаимодействующих между собой на разных уровнях, соблюдая при этом инкапсуляцию (протокол верхнего уровня работает над протоколом нижнего уровня). Уровни, на которых работает этот стек, описываются моделью, созданной министерством обороны США (Модель DOD)[[1]](#footnote-1).

|  |  |  |
| --- | --- | --- |
| **Уровень** | **Функции** | **Примеры протоколов** |
| 1. Прикладной | Работа сетевых приложений | HTTP, FTP, SSH, DNS |
| 1. Транспортный | Доставка данных | TCP, UDP |
| 1. Сетевой | Связь подсетей друг с другом | IP, DHCP, RIP |
| 1. Канальный | Доставка пакетов | Ethernet, Token Ring |

Нас интересует, прежде всего, сетевой уровень, так как именно он обеспечивает построение топологии самой сети в целом, используя Internet Protocol (IP).

## Internet Protocol (IP)

***Примечание:*** *В данном докладе будет идти речь о IPv4, более современный IPv6, на который уже переходит мир, не будет затронут, т.к. в локальных сетях все еще не требуется количество адресов, предоставляемых IPv6.*

Итак, именно IP позволяет нам соединить какое-то количество узлов в одну сеть. Происходит это благодаря использованию IP-адресов и масок подсетей. Оба состоят из четырех 8-значных двоичных чисел (октетов), которые в итоге и составляют адрес (Пример – 127.0.0.1). О том, зачем нужна маска подсети, мы узнаем чуть позже.

## Аппаратные устройства.

Теперь попробуем разобраться в аппаратуре, которая соединяет компьютеры друг с другом в различные топологии. Самый простой элемент – сетевая карта, стоящая в любом клиентском компьютере.Она принимает сигнал, который идет на нее с провода, отдает его компьютеру на обработку и отправляет другую информацию, если компьютер того требует.

Как я ранее говорил, два компьютера можно легко соединить между собой напрямую, однако для соединения между собой уже трех и более компьютеров, нам понадобится дополнительная аппаратура. Таковыми являются сетевой концентратор и коммутатор.

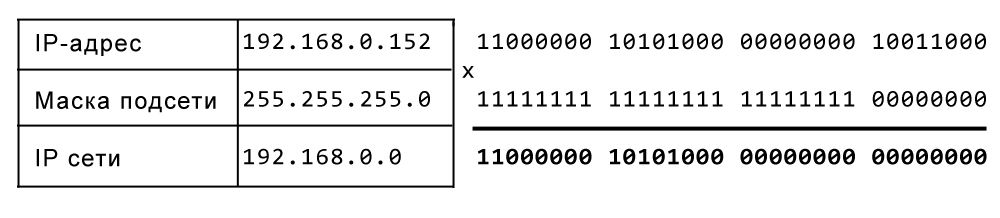
***Сетевой концентратор****(англ. Hub) – устройство, соединяющее сегменты одной сети. При получении пакета на один из портов, передает его на все остальные.*

***Коммутатор*** *(англ. Switch) – более «умная» версия концентратора. В отличие от предшественника, передающего пакеты сразу на все порты, умеет читать пакеты и отправлять их непосредственно адресату; для этого в его памяти находится таблица коммутации, содержащая MAC-адреса сетевых карт, принадлежащих каждому порту. При получении пакета коммутатор «вытягивает» из него MAC получателя и отправляет его на необходимый порт.*



*Рис 1.2 Изображения концентратора (слева) и коммутатора (справа)*

Однако количество портов на данных устройствах не бесконечно, и мы не можем расширять нашу сеть бесконечно, т.к. концентратор крайне неэффективен, а коммутатор перестает правильно функционировать при подключении к другому коммутатору, ведь любой из них видит лишь адреса своих компьютеров и адрес соседа, но не «видит» компьютеры, подключенные к соседу. Тут-то нам и помогают вещи, называемые подсетями.

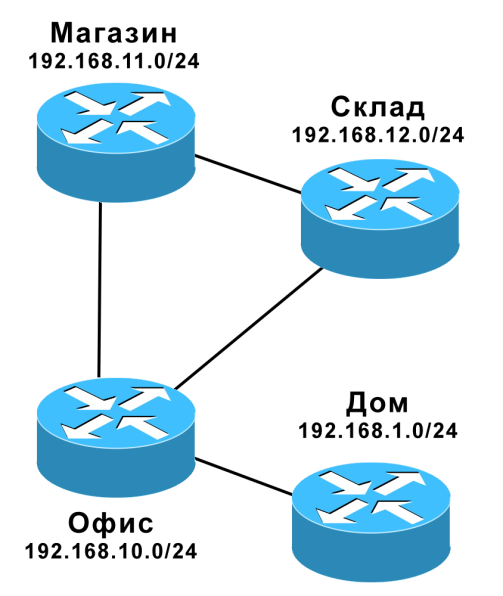
Подсети соединяются т.н. **маршрутизаторами**, которые работают уже на более высоком сетевом уровне – их главная задача – маршрутизация пакетов, т.е. гарантированная доставка пакета из одной подсети в другую. Для определения того, из какой подсети был послан пакет, используется маска подсети, похожая на IP-адрес, но состоящая из последовательности единиц, за которой идет последовательность нулей. Если мы бинарно перемножим IP-адрес компьютера на маску его подсети, мы и получим адрес подсети, в которой он находится.

*Рис 1.3 Вычисление IP-адреса сети*

***Примечание:*** *Так как маска подсети является одной последовательностью единиц, а все остальное после этой последовательности – нули, можно вычислить саму маску зная только ее длину, т.е. количество единиц в ней. Краткая запись сети 192.168.0.0 с маской 255.255.255.0 будет выглядеть как 192.168.0.0/24.*

Чем больше у нас становится таких подсетей, тем больше маршрутизаторов нам придется использовать. Если абстрагироваться до сетевого уровня, то мы можем увидеть, что, по сути, наша сеть состоит из нескольких подсетей, по-разному соединенных между собой маршрутизаторами. То есть, наша сеть имеет форму графа, где каждая из точек графа – подсеть, а ребра между точками позволяют нам понять, связаны ли они между собой.

***Граф*** *– совокупность множества вершин, произвольно соединенных ребрами.*

**

*Рис 1.4 Пример графа сети*

Именно в виде графа и представляют нашу сеть протоколы динамической маршрутизации.

## Протоколы динамической маршрутизации

В памяти каждого маршрутизатора, находящегося в нашей сети, лежит таблица маршрутизации, согласно которой он и будет отправлять пакеты из одной сети в другую. Эта таблица может задаваться вручную пользователем (такая маршрутизация будет называться статической), или же таблица будет строиться динамически какими-либо протоколами. В своей работе я, не вдаваясь в излишние подробности, рассмотрю два алгоритма динамической маршрутизации – RoutingInformationProtocol (**RIP**) и OpenShortestPathFirst (**OSPF**)

### Routing Infrormation Protocol

Данный протокол работает на базе алгоритма Беллмана-Форда и является старейшим протоколом динамической маршрутизации. Использовался в сети ARPANET(прототип современного интернета) с ее создания в 1969 году.

Принцип его работы заключается в том, что каждый член сети (здесь и далее – маршрутизатор, «представляющий» одну подсеть) раз в 30 секунд отсылает всем своим соседям свою полную таблицу маршрутизации с метрикой 0 (числовое значение, которое в RIP обозначает количество «прыжков», которое совершила таблица, проходя по сети). При получении этих данных, следующий маршрутизатор сравнивает принятую таблицу со своей, заменяет какие-либо связи на более выгодные в своей таблице или же в полученной, увеличивая при этом метрику на 1, добавляет туда новые сети, которые не были найдены, после чего отправляет эту таблицу всем своим соседям. В итоге у каждого члена в конце оказывается полностью готовая таблица маршрутизации.

Естественно, такой алгоритм очень сильно нагружает канал, поэтому его использование ограничено небольшими сетями. Помимо того, максимальная метрика, которую допускает RIP, составляет 15; это значит, что расстояние между двумя членами не может быть более 15 прыжков. Если количество прыжков превышает 15, пакет отбрасывается, тем самым избавляясь от вероятности того, что один пакет будет вечно бродить по сети туда-обратно.

### Open Shortest Path First

Данный протокол был разработан Инженерным советом Интернета (IETF)в 1988 году и, в отличие от RIP, может работать в больших сетях без излишней перегрузки канала. Каждый член сети обменивается со своими соседями так называемыми hello-пакетами. После чего, когда между двумя членами будет установлена связь, они будут обмениваться данными друг с другом в зависимости от того, есть ли какие-то расхождения в их базах данных. В конце у каждого маршрутизатора будут лежать три базы данных, а именно:

1. База данных канального уровня. В ней будет находиться полная информация о связях между подсетями и метриках, которые вычисляются в зависимости от пропускных способностей каналов.
2. База данных соседей, в которой хранится информация о соседних членах.
3. Сама таблица маршрутизации, строящаяся по алгоритму Дейкстры.

Теперь, когда мы разобрались с тем, как же соединяются между собой связанные компьютеры, для чего нужны концентраторы, коммутаторы и маршрутизаторы и рассмотрели два популярных протокола маршрутизации, я расскажу о написанной мной программе и алгоритмах, которые работают в протоколах динамической маршрутизации.

# Графы.

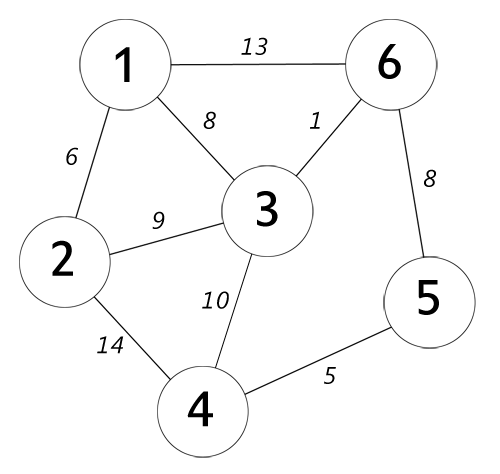
Прежде всего, передо мной стояла задача реализовать графы и найти способ задавать их. Способов задать граф 2:

1. Использовать т.н. списки смежности. В каждом из этих списков будут храниться все соседи единственной вершины.

+ Занимает небольшое количество памяти

+ Прост для понимания и позволяет быстро «построить» граф на бумаге.

– Для того, чтобы хранить еще и веса ребер, нам понадобится делать еще один массив для каждого списка, увеличивая кол-во необходимой для хранения памяти в два раза.

1. Использовать матрицу смежности. Матрица смежности – это двумерный массив размером n\*n(где n-количество точек в графе), где элемент [i,j] будет являться весом ребра между точками i и j.

+ Не требует дополнительных массивов для хранения весов

+ Нахождение веса ребра становится более простой задачей

– Немного сложнее для понимания «на ходу»

– В случае графа с большим количеством точек, но малым кол-вом связей между них, очень высоки затраты памяти в сравнении со списками.

Если между точками нет ребра, то элемент матрицы будет отсутствовать (null).

**Списки смежностей:**

Lists[1]=[2,3,6]

Lists[2]=[1,3,4]

Lists[3]=[1,2,4,6]

Lists[4]=[2,3,5]

Lists[5]=[4,6]

Lists[6]=[1,3,5]

**Матрица смежностей:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **1** | **2** | **3** | **4** | **5** | **6** |
| **1** | -1 | 6 | 8 | -1 | -1 | 13 |
| **2** | 6 | -1 | 9 | 14 | -1 | -1 |
| **3** | 8 | 9 | -1 | 10 | -1 | 1 |
| **4** | -1 | 14 | 10 | -1 | 5 | -1 |
| **5** | -1 | -1 | -1 | 5 | -1 | 8 |
| **6** | 13 | -1 | 1 | -1 | 8 | -1 |

Я решил использовать матрицу смежности для задания графа. Так как некоторые алгоритмы требуют наличия численных данных во всей матрице, я решил использовать значение -1 для обозначения отсутствия ребра между точками (Графы с отрицательными весами рассматриваться не будут, ведь в них нахождение кратчайшего пути невозможно).

Для простоты понятия работы алгоритмов, была создана абстракция графа с помощью классов графа (Graph), точки (Node) и ребра (Edge). Прежде чем приступать к разбору алгоритмов, рассмотрим, что делает каждый из классов, начиная с самого простого, реализующего ребро, и закончив общим классом графа.

## Класс Edge

Класс Edge, реализующий ребро, имеет три поля – startNode, endNode и weight. Первые два поля – точки, которые связывает это ребро, с направлением ребра startNode→endNode. Поле weight отвечает за вес этого ребра. Конструктор Edge принимает все эти три параметра и создает нужное ребро.

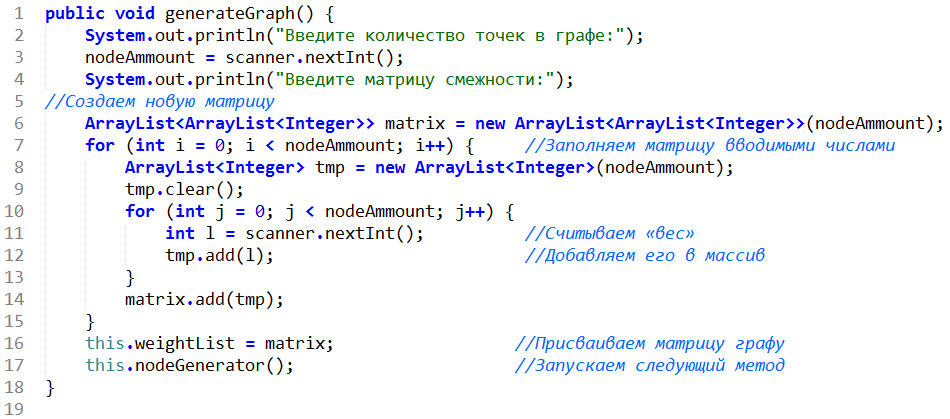
## 

## Класс Node

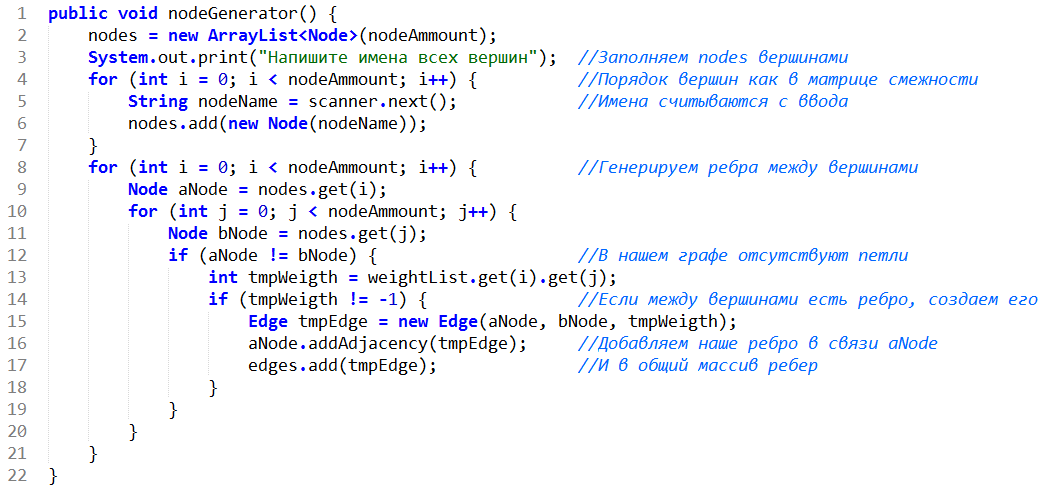
Класс Node реализует точку графа и имеет два главных поля – adjacencies, являющийся массивом всех ребер, соединенных с этой точкой, а также имя этой точки name. Помимо того, этот класс имеет и другие поля и методы, используемые в алгоритмах; о них я расскажу позже.

## Класс Graph

Этот класс является главным, так как он «соединяет» все точки и ребра воедино. Он хранит в себе все данные о графе – он имеет изначальную матрицу смежности weightList, количество точек nodeAmmount (в тексте используется как n) и массив всех точек nodes, а также массив всех ребер edges. Все методы, реализующие алгоритмы, находятся в этом классе, помимо того, генерация графа происходит внутри самого класса (хотя было бы немного логичнее вынести это вне класса).



Метод generateGraph() считывает матрицу с ввода, после чего запускает метод nodeGenerator(), который генерирует по полученной матрице каждую из вершин графа вместе с ребрами.



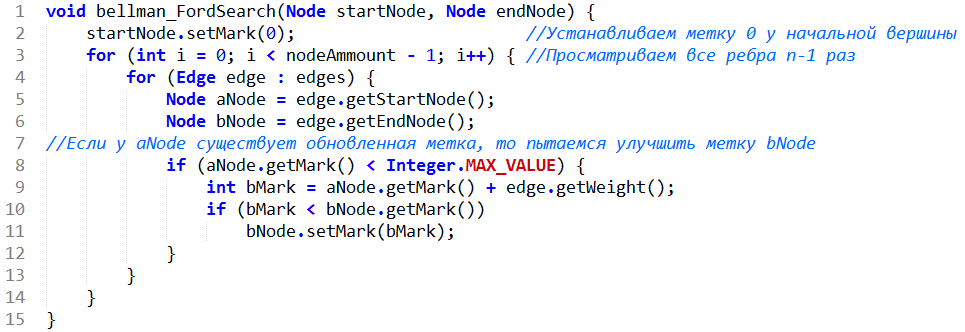
# Реализация алгоритмов

## Алгоритм Беллмана-Форда

Первый алгоритмом, который я хочу разобрать, это алгоритм Беллмана-Форда, используемый в протоколе RIP. Я разберу его первым, так как он является самым простым и понятным среди трех выбранных мной алгоритмов. Итак, для начала разберемся, что же такое кратчайший путь. Кратчайший путь – такая последовательность вершин между вершинамии i и j, где две любые вершины соединены между собой хотя бы одной промежуточной вершиной, при этом этот сумма весов всех последовательных вершин между точками i и j будет иметь наименьшее возможное значение в графе. Однако, я решил немного отойти от терминологии теории графов и назвать этот путь оптимальным, а под кратчайшим путем я буду иметь ввиду путь, имеющий наименьшее количество промежуточных звеньев (прыжков) между i и j.

Работа алгоритма заключается в том, что он просматривает все ребра графа n-1 раз, каждый раз сравнивая метку «целевой» вершины ребра с суммой метки начальной вершины и веса ребра и заменяя ее, в случае если новая метка будет меньше. Метка начальной точки, естественно, равна 0. После выполнения шага n-1 все точки графа, достижимые из начальной, будут иметь метки, соответствующие весу кратчайшего пути до них[[2]](#footnote-2).

Алгоритм Беллмана-Форда работает используя матрицу смежности, однако он легко переносится и на абстракцию графа без каких-либо проблем. Для реализации алгоритма добавим в класс Node параметр mark, который будет нести в себе сиюминутную метку данной вершины. Значение по умолчанию будет равно Integer.MAX\_VALUE, т.е. максимальному значению Integer.

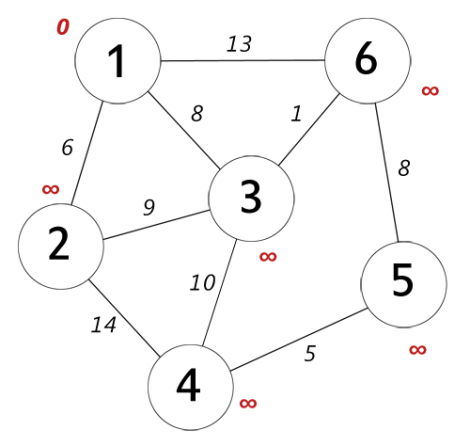
******

Теперь, когда у каждой вершины есть метка и мы спокойно знаем стоимость кратчайшего пути между i и j, нам остается узнать сам путь между ними. Для этого мы добавим в наш класс Node еще один параметр – массив pathTo, который будет содержать в себе последовательно все вершины от начальной и до нее самой (вершины с этим массивом). Модифицируем наш алгоритм Беллмана-Форда, добавим метод pathRefresh(), который будет брать путь от aNode, добавлять в нее bNode и устанавливать bNode.pathTo на только что созданный.

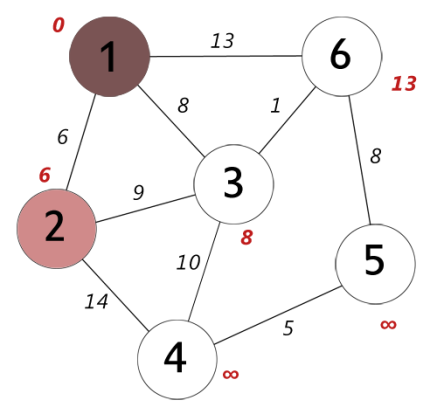
Алгоритм Дейкстры

Вторым алгоритмом, который я собираюсь разобрать, является алгоритм Дейкстры, разработанный Эдсгером Дейкстрой в 1959 году. Этот алгоритм чрезвычайно популярен и используется во многих сферах программирования, таких как нахождение пути для искусственного интеллекта игровых персонажей или реальных роботов. Используется в раннее упомянутом мной протоколе Open Shortest Path First для вычисления маршрутов от подсети до подсети. Принцип работы алгоритма таков:

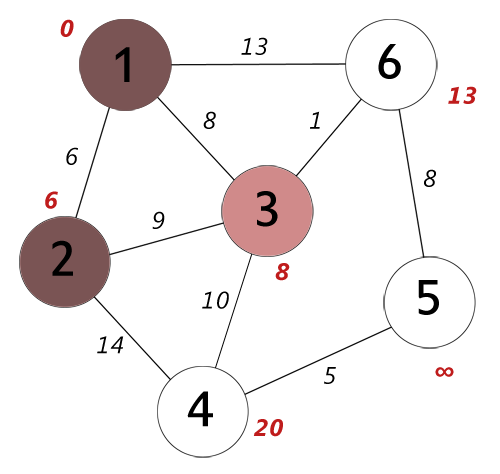
1. Каждой вершине графа кроме начальной присваивается метка ∞, начальной вершине присваивается метка 0. (Аналогично Беллману-Форду)
2. Просматривается каждый сосед стартовой точки, метка соседа улучшается суммой метки начальной точки и ребра до соседа (в самом начале – просто вес ребра).
3. Начальная точка помечается как пройденная и больше не будет затрагиваться при последующих вычислениях.
4. Среди всех существующих на графе точек выбирается точка с наименьшей меткой, после чего она становится следующей «начальной» точкой. Те её соседи, помеченные как пройденные, рассматриваться не будут.
5. В конце действия алгоритма все точки, соединенные с начальной, будут помечены как пройденные, на остальных же будет оставаться бесконечная метка.

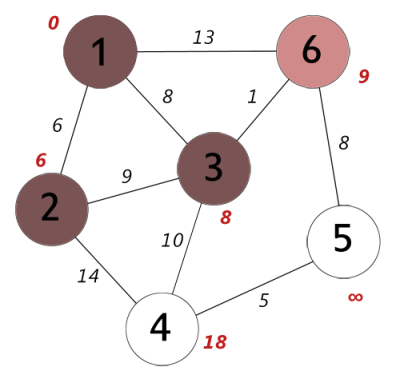
Для упрощения понимания алгоритма, рассмотрим его на примере:

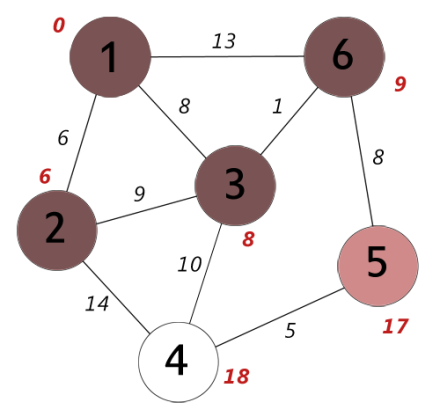
1. Допустим, нам надо найти путь до всех вершин из вершины 1. Для этого ставим метку 0 на 1, на все остальные ставим метку бесконечности.



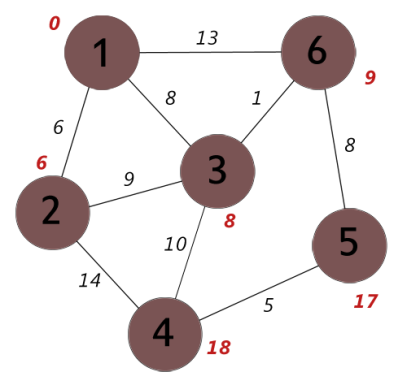
1. Обновляем метки для всех вершин, соседних с вершиной 1, после чего вычеркиваем ее и выбираем новую вершину с минимальной меткой – это будет вершина 2.



1. Улучшаем метки соседей вершины 2, игнорируя «вычеркнутую» вершину 1, и переходим к следующей вершине, которая будет вершиной 3.
2. Улучшаем метки соседних с вершиной 3 вершин 4 и 6, переходим к метке 6, имеющей минимальную метку в графе равную 9.

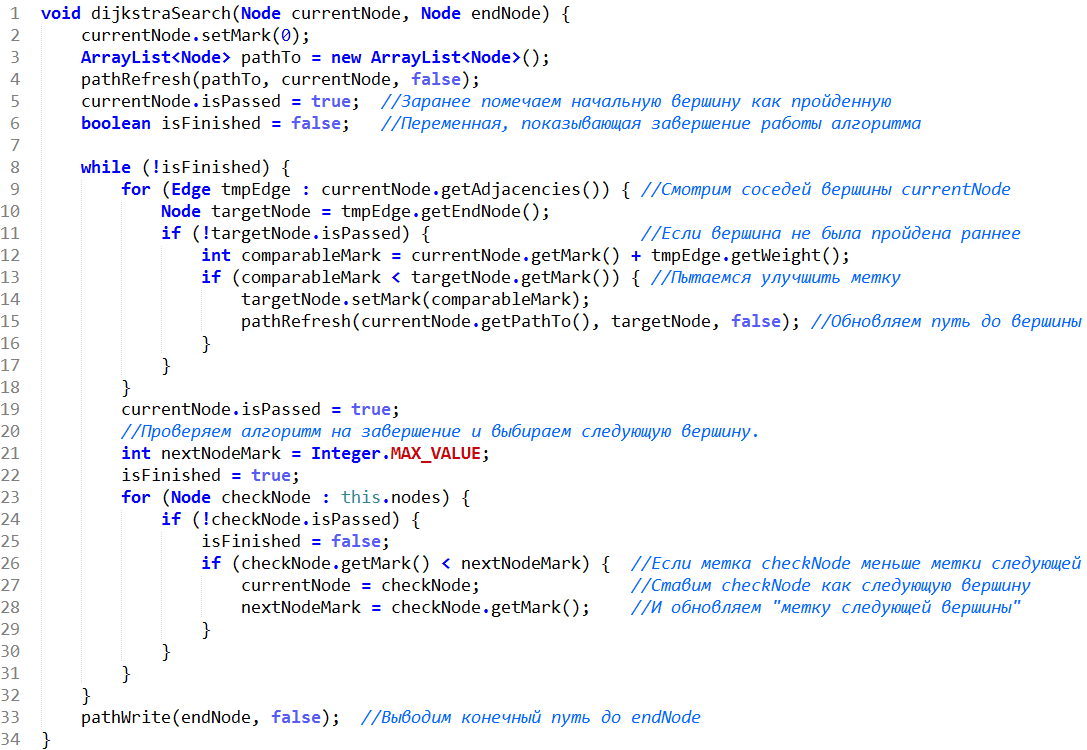


1. Теперь делаем те же шаги с вершиной 6 и переходим к вершине 5.



1. Пытаемся обновить метку вершины 4 из вершины 5, вычеркиваем вершину 5 и вершину 4, т.к. все ее соседи уже были пройдены. Работа алгоритма закончена.

Данный алгоритм будет работать верно только на графе без отрицательных граней, но, как я уже упоминал, данные графы рассматриваться не будут. Реализация алгоритма будет выглядеть так:

******

## Алгоритм Флойда-Уоршелла.

Этот алгоритм был разработан вместе Робертом Флойдом и Стивеном Уоршеллом в 1962 году. Несмотря на то, что алгоритм наименован в их честь, он был опубликован еще ранее в 1959 году Бернардом Ройем, но эта публикация осталась незамеченной.

Алгоритм разделяет процесс высчитывания оптимального пути на фазы и работает используя матрицу смежностей, в конце выстраивая матрицу pathMatrix, где pathMatrix[i,j] будет обозначать длину кратчайшего пути от вершины i до вершины j. В самом начале pathMatrix идентична weightList (матрице смежности).

***Важная особенность****: Для корректной работы алгоритма Флойда-Уоршелла граф не должен иметь т.н. петель (связей точки с самой собой), а pathMatrix[i,i] всегда должен быть равен 0. В ином случае алгоритм не будет работать.*

На k-ой фазе берется вершина k, и через нее стараются улучшить все остальные вершины (путь i→j заменяется на i→k+k→j, в случае, если последний короче).

Стоит заметить, что из-за особенностей языка Java, мы не можем просто так копировать pathMatrix из weightList напрямую через weightList.clone(), так как это сделает только т.н. «поверхностную» копию; это значит, что любые изменения, сделанные в одной из матриц, будут применены и для другой матрицы, так как каждая матрица будет содержать лишь ссылки на объекты. Для решения этой проблемы, перед выполнением самого алгоритма, мы сначала делаем глубокую копию weightList.

Базовая реализация этого алгоритма проста и выглядит так:



Однако нам необходимо знать не только длину кратчайшего пути от i до j, но и сам этот путь, однако наша матрица не несет в себе такой информации, а метод, используемый в предыдущих двух алгоритмах не работает, так как в данном случае возникнет одна проблема. Если мы возьмем вершину l с путем i→l и какое-то множество точек (j), достижимое из l, при изменении пути i→l на i→l = i→k→l нам придется динамически менять пути до всех точек, входящих в (j), что значительно усложняет алгоритм программы.

Но у этой проблемы существует решение. Мы создаем еще одну матрицу stepMatrix, где stepMatrix[i,j] будет являться номером фазы, на которой было получено кратчайшее расстояние, то есть stepMatrix[i,j] – номер «промежуточной» вершины между этими двумя точками. Отследив рекуррентно в обратном порядке все вершины от stepMatrix[i,j] и до stepMatrix[m,i] (m в данном случае – вершина, идущая в общем пути i→m→…→j сразу после i), мы можем последовательности вывести путь одним рекуррентным алгоритмом.



# Интерфейс программы и системные требования

Программа, написанная мной, имеет текстовый консольный интерфейс, написана на Java SE 6 и может запуститься на компьютере с любой ОС, которая может запустить виртуальную машину Java. Рекомендуемые системные требования:

Частота CPU: 1 ГГц и более.

Объем ОЗУ: 512 Мбайт  
ОС: MS Windows/Linux/MacOS/BSD

Наличие установленной Java SE 6.

Процесс работы программы:

1. Программа спрашивает, откуда считывать ввод. Если вводится 0, то ввод будет считываться из стандартного ввода, в ином случае требуется ввести путь к файлу, экранируя при этом некоторые символы (Пример: C:\\Graph.txt, знак “\” экранируется еще одним знаком \, в ином случае JVM воспримет это как специальную последовательность.)
2. Необходимо ввести количество вершин графа, после чего ввести матрицу по строкам и затем ввести имена всех вершин, имя каждой следующей вершины с новой строки.
3. Необходимо выбрать алгоритм, по которому будет осуществляться поиск, введя необходимую цифру:

0 для алгоритма Дейкстры

1 для алгоритма Беллмана-Форда

2 для алгоритма Флойда-Уоршелла

1. Вводятся индексы вершин startNode и endNode, между которыми надо найти расстояние (если надо найти расстояние между вершинами 2 и 9, то вводятся цифры 1 и 8 соответственно).
2. Алгоритм выводит путь, метку endNode/матрицу путей и матрицу «шагов» в случае Флойда-Уоршелла.
3. Для выхода из программы потом достаточно ввести «Exit »

Естественно, вводить все это самостоятельно - немного длительное занятие, поэтому существует возможность считывания из файла, в котором пошагово записаны все эти команды.

# Выводы

В своей работе я проанализировал процесс маршрутизации пакетов между сегментами сети, какие инструменты и протоколы за это отвечают. Я проанализировал алгоритмы, по которым работают 2 протокола динамической маршрутизации, и смог реализовать эти алгоритмы в отдельной программе, где был реализован также и дополнительный алгоритм Флойда-Уоршелла.

Исходный код работы вы можете найти на GitHub: https://github.com/rlokc/GraphsForRouting

# Список литературы

1. Таненбаум Э.: Компьютерные сети, 2007, 992 стр, ид - ISBN 978-5-318-00492-6,5-318-00492-6
2. Олифер В., Олифер Н.: Компьютерные сети. Принципы, технологии, протоколы, 2010, 944 стр, ид - ISBN 978-5-49807-389-7
3. Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К.: Алгоритмы. Построение и анализ, 2012, 1296 стр, ид - ISBN 978-5-8459-0857-5, 0-07-013151-1
4. www.habrahabr.ru – информационный ресурс
5. www.opennet.ru – информационный ресурс
6. www.e-maxx.ru/algo - информационный ресурс

1. Существует также эталонная модель OSI, но TCP/IP имеет собственную модель по причине того, что он попросту был разработан раньше введения этой модели. Помимо того, каждый уровень модели TCP/IP так или иначе включает в себя один или несколько уровней модели OSI. По этим причинам, я не собираюсь рассматривать эту модель в своем проекте. [↑](#footnote-ref-1)
2. Достаточно выполнить цикл n-1 раз, так как в ином случае расстояние между вершинами i и j будет больше количества вершин в графе, что невозможно при любом случае кроме как при наличии в графе ребер с отрицательным весом [↑](#footnote-ref-2)