

**ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА**

**ИССЛЕДОВАНИЕ ПЕРЕМЕЩЕНИЯ МОБИЛЬНЫХ БЕСПРОВОДНЫХ СЕТЕЙ ДЛЯ МОДЕЛИ ДВИЖЕНИЯ УЗЛОВ RANDOM WAYPOINT MODEL**

Работу выполнил В. А. Комаричев

СОДЕРЖАНИЕ

# Введение ................................................................................................................... 3

1. Основные понятия Ad hoc сетей ......................................................................... 4
   1. Основные понятия и определения ................................................................ 4
   2. Область применения сетей Ad hoc ............................................................. 10
   3. Математическая модель представления сетей ad hoc .............................. 11
   4. Базовые понятия теории графов ................................................................. 13
   5. Определения моста, точки сочленения и компоненты связности графа 15
   6. Модель Random Waypoint ........................................................................... 17
   7. Известные проблемы модели Random Waypoint ...................................... 20
2. Практическая реализация .................................................................................. 21
   1. Среда разработки ......................................................................................... 21
   2. Представления графов в практической реализации ................................. 23
   3. Оценка сложности решений поиска мостов, точек сочленения и компонент связности, приведенных в программе .......................................... 25

2.4 Структура программы ................................................................................. 25

* + 1. Основные блоки программы .............................................................. 25
    2. Описание работы программы ............................................................ 36
    3. Анализ результатов ............................................................................. 38

Заключение ............................................................................................................ 50

Список использованных источников .................................................................. 51

Приложение А Код программы ........................................................................... 52

ВВЕДЕНИЕ

В настоящее время сети передачи данных продолжают активно развиваться, в том числе все большее распространение получает такой их класс, как сети Ad hoс. Это одно-ранговые беспроводные сети передачи данных с переменной топологией и отсутствием четкой инфраструктуры, где каждый узел может выполнять функции маршрутизатора и принимать участие в ретрансляции пакетов данных. Минимальное конфигурирование и быстрое развёртывание позволяет применять самоорганизующиеся сети в чрезвычайных ситуациях - во время военных действий, в структурах МЧС во время техногенных катастроф и природных бедствий, а так же в системах транспорта и различных силовых структурах.

Так как на сегодняшний день имитационное моделирование является основным средством, которое позволяет оценить эффективность той или иной модели для самоконфигурируемых мобильных сетей (потому как на данный момент до сих пор не существует формальная основа для оценки эффективности решений, потому что эксперименты на мобильных самоорганизующихся сетях крайне затруднены в силу того, что сложно организовать подобные испытания с большим количеством участников, и уж тем более воссоздать равные условия моделирования для всей серии экспериментов, а так же в связи с внушительными временными затратами), именно имитация сети и оценка результатов были основными целями этой работы.

1 Основные понятия Ad hoc сетей

## 1.1 Основные понятия и определения

Ad hoc-сеть (беспроводная динамическая сеть или же беспроводная самоорганизующаяся сеть) — децентрализованная беспроводная сеть, не имеющая устойчивой постоянной структуры. Клиентские устройства соединяются «на лету», тем самым образуя собой сеть. Каждый узел в сети пытается переслать определенные данные другим узлам. При этом, какому именно из узлов осуществлять пересылку данных, определяется динамически, на основании связности сети. Этим ad hoc-сети отличаются от управляемых беспроводных сетей и проводных сетей, в которых задачу управления потоками данных выполняют точки доступа или маршрутизаторы соответственно. На рисунке 1 изображен пример сети Ad hoc.

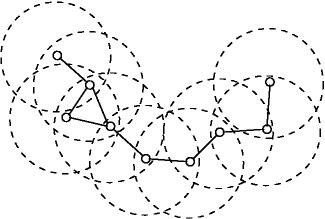


Рисунок 1 - Пример ad hoc сети

Топология сети c течением времени может непредсказуемо меняться по различным причинам: во-первых, потому что работающие узлы мобильны, во-вторых, потому что некоторые из неподвижных узлов могут выйти из строя и перестать работать. Периодически в такой сети происходят изменения, в ходе которых меняется местонахождение узлов, следовательно, меняется и расстояние между соседними узлами. Именно поэтому Ad hoc сети можно называть также эпизодическими сетями. Зачастую в качестве эпизодической сети выступает именно беспроводная сеть.

Эпизодическая сеть является одноранговой, это означает, что все узлы в сети наделены одинаковыми функциями. Каждый из таких узлов может общаться с любым другим, и в этом случае не предусматривается отдельная базовая станция или точка доступа. Но если возникает необходимость подключения этой Ad hoc сети к другой сети, например, такой, как Интернет, один из этих узлов эпизодической сети может быть наделен правами базовой станции или некоторого координатора этой сети. Узлы эпизодической сети имеют ограниченную мощность передатчика, из-за этого они имеют также ограниченную «радиовидимость». Еще одним важным свойством узлов подобной эпизодической сети, не считая прием и передачу, является возможность ретранслировать информацию и маршрутизировать. Каждое устройство в такой сети перемещается независимо от других в абсолютно любом направлении. Самоорганизованность в сетях Ad hoc необходима для того, чтобы была возможность связности узлов. Узлы, во время своего функционирования в сети, используют общий беспроводной канал, причем доступ к нему предоставляется случайным образом. Передача данных в сети реализуется узлами посредством ретрансляции. Поэтому узлы являются не только хостами, но также и маршрутизаторами. Наиболее важными особенностями таких сетей являются:

* наличие децентрализованного управления и одноранговой структуры;
* наличие непредсказуемо и динамически меняющейся топологии;
* наличие функции маршрутизации у всех узлов;
* наличие самоорганизации сети (через ретрансляцию);
* наличие различных ограничивающих факторов в сети в виде дальности радиовидимости или ёмкости батарей устройств; - большая потеря данных.

Существует несколько видов основных эпизодических сетей (рисунок 2):

* мобильная эпизодическая сеть (MANET — mobile ad hoc network);
* фиксированная эпизодическая сеть, пример — беспроводная сенсорная сеть (WSN — wireless sensor network);

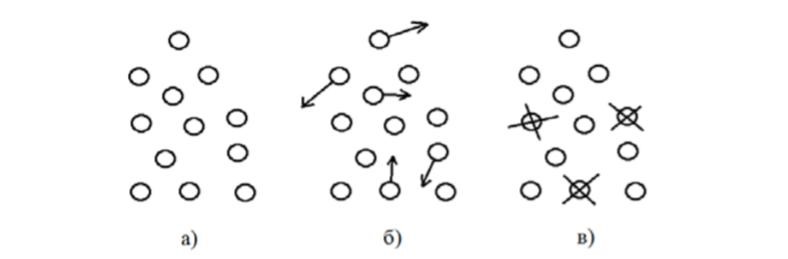


Рисунок 2 – Сети с переменной топологией: a) стационарная сеть; б) сеть с подвижными узлами; в) сеть с отказавшими узлами

У сетей MANET также есть свои разновидности, среди которых:

* VANET — автомобильная ad hoc сеть (vehicular ad hoc network);
* InVANET — эпизодическая сеть на основе роботизированных

мобильных систем (intelligent VANET);

* IMANET — эпизодическая сеть с возможностью выхода на фиксированную сеть Интернет (internet MANET);
* WANET — беспроводная эпизодическая сеть с подвижными базовыми станциями (wireless ad hoc network);
* WMN — беспроводная клеточная (ячеистая) сеть (wireless mesh networks);
* Airborne Ad Hoc Network — эпизодическая сеть, приёмопередатчики которой находятся на воздушных платформах.

В таблице 1 приведена сравнительная характеристика сетей.

Таблица 1 – Сравнительная характеристика сетей

|  |  |  |  |
| --- | --- | --- | --- |
| Тип сети | Мобильность | Энергетика | Вычислительная мощность |
| MANET | Низкая | Ограничена | Средняя |
| VANET | Высокая | Не ограничена | Высокая |
| Проводная | Отсутствует | Не ограничена | Высокая |

Для того чтобы полностью классифицировать эпизодическую сеть используют разные свойства и признаки, такие как:

* архитектура сети (одноранговая или с выходом в другие сети);
* мобильность (мобильные узлы или фиксированные);
* взаимное размещение узлов в пространстве (плоское, линейное или трехмерное);
* число узлов (малое, среднее, большое, огромное);
* скорость (время) изменения топологии (секунды, минуты, часы, дни) и т.д.

Из-за подвижности абонентов топология сети постоянно и непредсказуемо меняется (рисунок 3). Простейшим видом связи является передача между соседними узлами — напрямую (в один скачок). Для передачи информации между произвольными узлами, как правило, требуется многоскачковая (релейная) трансляция, причем набор промежуточных узлов из-за их мобильности может меняться даже в ходе сеанса связи. Поскольку Ad hoc сеть является одноранговой, то в процессе одновременной передачи информации несколькими узлами могут возникать конфликтные ситуации, и не только в отношении общего канала связи. Например, мобильный аппарат должен работать для одного маршрута, как оконечный приемник, а для другого маршрута — как ретранслятор.

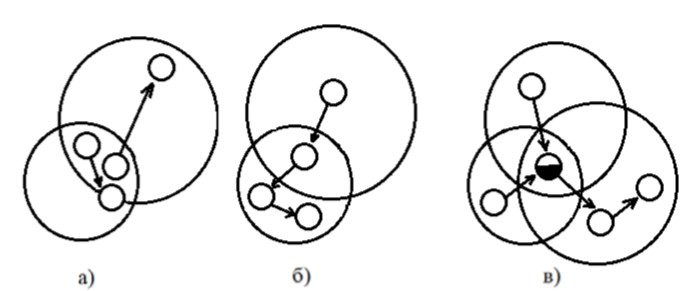


Рисунок 3 – Влияние движения узлов на топологию сети: a) прямая передача;

б) релейная передача(в две скачка); в) конфликт в узле

Эпизодическая сеть может работать независимо или подключаться к другим сетям, например, к Интернет, через соответствующие устройства. Концепция эпизодических сетей имеет большие перспективы как с точки зрения возможностей развития следующего поколения сетей, так и с точки зрения концепции всепроникающих телекоммуникаций (omnicommunications). Базовыми характеристиками эпизодических сетей, отражающими их специфику, являются:

* среднее число функционирующих узлов в сети;
* подвижность узлов;
* предельная ёмкость сети;
* гарантированная емкость сети;
* максимальное (гарантированное, среднее) время пересылки в сети;
* связность или среднее число соседних узлов в зоне радиовидимости;
* живучесть сети или вероятность доставки информации любому действующему узлу в сети;
* мортальность узлов (пропадание, выход из зоны видимости узлов за определенный срок, среднее значение).

Вместе с тем нужно упомянуть о преимуществах эпизодических сетей:

* относительная простота — не требуется предварительная установка стационарного оборудования;
* гибкость – сеть легко разворачивается в любое время в любом месте;
* обеспечение доступа к информации и услугам без привязки к географическому расположению абонентов;
* возможность быстрой реконфигурации в условиях помех или иных неблагоприятных обстановках;
* высокая скорость развертывания;
* возможность передавать данные на большие расстояния без увеличения мощности передатчика;
* устойчивость к изменениям в инфраструктуре сети.

Перечислим особенности эпизодических сетей с точки зрения применения в специализированных системах, обусловленные хорошо развитым алгоритмическим и программным обеспечением:

* поддержка интерактивного обмена информацией в реальном времени;
* высокие требования по энергетической эффективности;
* устойчивость к наведенным помехам; обильность магистральной линии;
* устойчивость к прерываниям связи;
* управление безопасностью и обеспечение живучести;
* обнаружение несанкционированного доступа;
* эффективность и безопасность маршрутизации;
* динамическая адресация;
* адекватная емкость сети.

## 1.2 Область применения сетей Ad hoc

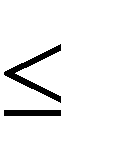
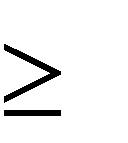
Область применения эпизодических сетей широка и разнообразна. Ad hoc сети могут быть применены в самых различных отраслях науки и техники. В таблице 2 приведены области, в которых могут быть применены эпизодические сети.

Таблица 2 – Области применения мобильных эпизодических сетей

|  |  |
| --- | --- |
| Область применения | Описания/службы |
| Сенсорные сети | Домашние приложения: умные сенсорные узлы и устройства управления могут быть встроены в технику,  чтобы конечные пользователи могли управлять бытовой техникой локально или удаленно. |
| Домашние и представительские  сети | Домашние/офисные беспроводные сети (WLAN), например, использование общедоступных лекционных  экранов, использование карманных компьютеров для печати документов из любого места, торговые презентации. Персональные сети |
| Коммерческие приложения | Электронная коммерция: электронные платежи из любого места. Бизнес. Динамический доступ к данным, которые хранятся в центральном хранилище.  Представление согласованных баз данных агентам.  Мобильный офис. Сервисы для транспортных средств.  Передача новостей, состояния дорог, прогноза погоды, музыки. Локальная эпизодическая сеть с транспортными средствами для получения информации о дороге/дорожных происшествиях. |
| Развлечения | Многопользовательские игры, роботизированные питомцы. Доступ в Internet на улице |
| Работы при чрезвычайных ситуациях | Поисковые и спасательные ситуации. Устранение последствий чрезвычайных ситуаций. Ранее получение и доставка данных о пациенте (запись, статус,  диагностика) из/в госпиталь. Замена фиксированной инфраструктуры в случае землетрясения, урагана или |
| Область применения | Описания/службы |
|  | иного стихийного бедствия. |
| Тактические сети | Военные коммуникации, операции. Автоматизированные поля сражений. |
| Сервисы для определения местоположения | Сервисы по отслеживанию автоматической переадресации вызова, передачи интеллектуальной информации о текущем местоположении.  Информационные сервисы: от абонента - информация о местонахождении определенного сервиса в зависимости  от местоположения, например, заправочной станции, к абоненту - туристический гид в зависимости от  местоположения, информация о доступных сервисах  (принтер, факс, телефон, сервер, заправочные станции). |
| Учебные приложения | Установка виртуальных классов или комнат для конференций. Установление эпизодических коммуникаций во время мероприятий. |

## 1.3 Математическая модель представления сетей ad hoc

Математическая модель расположения на плоскости ad hoc сети представляет собой объединение кругов заданных радиусов. В области расположения сети имеется естественный электромагнитный шум некоторого уровня. В центре одного из кругов (*a*) находится передатчик сигналов. Уровень электромагнитного поля, создаваемого передатчиком, уменьшается при удалении от центра круга. Радиусом *Ra* считается такое значение удаления, при котором отношение сигнал/шум соответствует чувствительности стандартного приемника, находящегося на этом удалении. Иначе говоря, при меньшем, чем *Ra*, удалении приемник способен принять и распознать сигнал, а при большем или равном расстоянии – нет (либо вообще не принимает сигнал, либо не может его декодировать и считает шумом).

Вершины *a* и *b* могут непосредственно обмениваться информацией, тогда и только тогда, когда расстояние между ними *d*(*a*, *b*) < min(*Ra*, *Rb*). Если *d*(*a*, *b*) max(*Ra*, *Rb*), то связи нет. Если min(*Ra*, *Rb*) *d*(*a*, *b*) < max(*Ra*,

*Rb*), то связь односторонняя. Таким образом, граф компьютерной сети является ориентированным (орграф). В частном случае, когда все радиусы одинаковы, граф является неориентированным.

Так как математической моделью сети ad hoc, по сути, является так называемый геометрический граф и данная работа состоит в оценке его характеристик в определенные моменты времени, то имеет смысл привести здесь несколько определений из теории графов, которые помогут понятию сути проведенной работы. Если расположение узлов сети изменяется со временем, то получается динамический граф *G*(*t*), в котором дуги (ребра) между вершинами – узлами, то исчезают (при удалении узлов друг от друга), то возникают (при приближении узлов). Соответственно, для произвольной пары вершин *a* и *b* маршрут, их соединяющий может существовать или не существовать. Если рассматривать связность графа как динамический процесс со значениями (1 – связен, 0 – не связен), то он может быть изображен как скачкообразно меняющаяся функция времени. Также изменяется скачкообразно количество компонент: в момент *t*0 граф состоит из *c*0 компонент, и сохраняет это значение до момента *t*1, когда вследствие появления новых ребер или исключения старых, количество компонент изменяется на целочисленную величину и становится равной *c* 1. И так далее. Если изобразить этот процесс графиком с осью абсцисс *t*, и осью ординат *c*, то получится кусочно-постоянная функция. Конкретный вид функции зависит от закономерностей (*x*(*t*), *y*(*t*)) перемещения вершин на плоскости (модели мобильности). Поскольку узлы сети располагаются в точках плоскости, можно ввести параметр расстояния между компонентами – расстояния между ближайшими узлами двух компонент. Он важен для прогнозирования вероятности и времени возможного соединения компонент при перемещении узлов. Для прогнозирования возможного разделения графа на компоненты (потери связности) важны такие параметры, как наличие мостов и точек сочленения, их взаимное расположение.

## 1.4 Базовые понятия теории графов

Пусть – конечное множество и – подмножество множества его двухэлементных подмножеств. Тогда пара = ( , ) называется (простым) графом c множеством вершин и множеством ребер . Более простое определение графа - совокупность точек и линий, в которой каждая линия соединяет две точки. Ребром может быть прямая или кривая линия. Ребра не могут иметь общих точек кроме вершин (узлов) графа. Замкнутая кривая в E может иметь только одну точку из множества V, а каждая незамкнутая кривая в E имеет ровно две точки множества V. На рисунке 4 изображен пример графа.

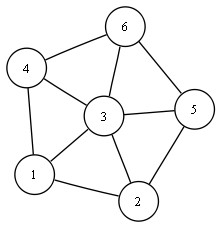


Рисунок 4 – Пример графа

Если V и E конечные множества, то и граф им соответствующий называется конечным. Параллельными ребрами графа называются такие, которые имеют общие узлы начала и конца. Вершины и графа смежны, если последний содержит ребро ( , ):= { , }; в этом случае говорим также, что вершины и инцидентны этому ребру. Две вершины, соединенные ребром, могут совпадать; такое ребро называется петлей. Если степень вершины равна 0, то получается изолированная вершина. Если два ребра инцидентны одной и той же паре вершин, они называются кратными; граф, содержащий кратные ребра, называется мультиграфом. Степенью вершины называется число, вычисляемое по формуле 1:

( ) = | ( )|, (1)

где ( ) = { ∈ : ( , ) ∈ } – множество соседей или окрестность этой вершины. Граф называется регулярным (степени ), если ( ) = степени для всех ∈ . Приведем несколько примеров.

Полный граф с вершинами содержит все ( − 1)/2 ребер; он

регулярен степени − 1.

Пустой граф не содержит ни одного ребра; он регулярен степени 0.

Цепь длины ≥ 0 имеет +1 вершину 0, . . . , и рёбер вида ( ,

+1), где = 0, . . . , −1.

Цикл длины ≥ 3 имеет вершин 1, . . . , , рёбер вида ( , +1), где = 1, . . . , − 1, и ребро ( , 1); это регулярный граф степени 2.

Вершины 0, . . . , ∈ , где ≥ 0, образуют цепь графа если (,

+1) ∈ при = 0, . . . , −1; в этом случае вершины 0 и называются концами цепи, а число −1 её длиной. Цепь называется простой, если все её вершины различны. Отношение на множестве " быть концами цепи в " является отношением эквивалентности. Для каждого класса этой эквивалентности индуцированный им подграф называются компонентой связности графа . Граф называется связным, если он сам является связной компонентой, что эквивалентно тому, что в нём любые две вершины являются концами некоторой цепи.

## 1.5 Определения моста, точки сочленения и компоненты связности графа

Граф называется связным, если любые две его вершины соединены путем. Компонента связности графа G — максимальный (по включению) связный подграф графа G. Ориентированный граф называется сильно связным, если две его произвольные вершины сильно связны. Две вершины i и j любого графа сильно связны, если существует ориентированный путь из i в j и ориентированный путь из j в i. Компонентами сильной связности орграфа называются его максимальные по включению сильно связные подграфы.

Вершина i графа G является точкой сочленения в том случае, если граф G- i состоит из большего числа компонент связности, чем G. Если граф G связный, то граф G- i будет содержать по крайней мере две компоненты связности, то есть будет не связным. Исходя из этого определения изолированная вершина графа не может быть точкой сочленения. Граф, состоящий из одной вершины зовется тривиальным. Тривиальный граф не может иметь точек сочленения. Связный граф, не имеющий точек сочленения зовется неразделимым графом (не связный граф так же может быть разделимым). На рисунке 5 приводится пример разделимого графа с тремя точками сочленения.

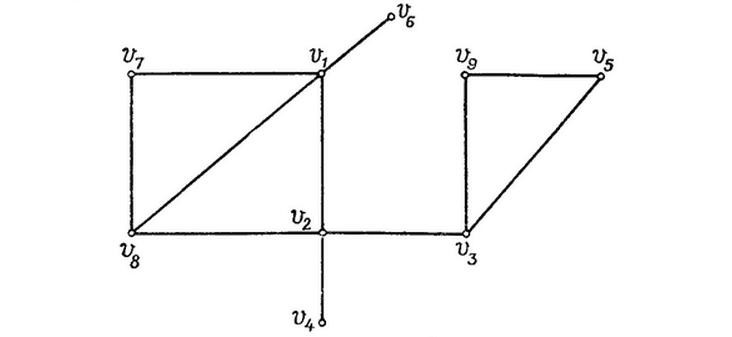


Рисунок 5 – Пример разделимого графа с тремя точками сочленения: 1, 2 и

3

Блоком разделимого графа G зовется максимальный неразделимый подграф графа G. На рисунке 6 приводится пример графа, разбитого на блоки.

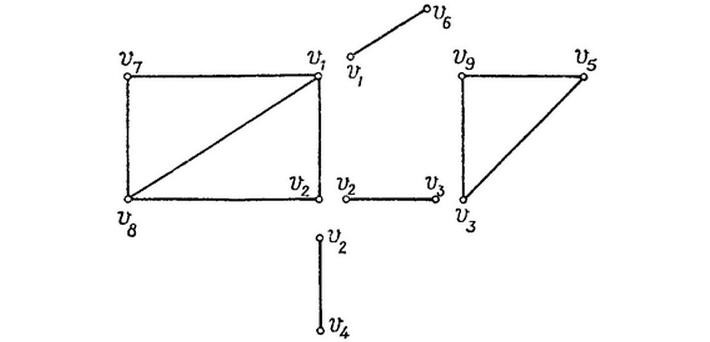


Рисунок 6 – Блоки графа

Мостом называется такое ребро графа G, при удалении которого, количество компонент связности графа G возрастает. На рисунке 5 мостом будет ребро 2 – 3.

## 1.6 Модель Random Waypoint

Мобильные системы характеризуются движением своих компонентов. Характер движения составляющих мобильной системы — их скорость, направление и скорость изменения — может сильно повлиять на протоколы и системы, предназначенные для поддержки мобильности. К сожалению, движение в физическом мире часто неповторимо. На практике реальное использование мобильной системы может обеспечить осмысленное понимание, но не может служить единственным основанием для оценки эксперимента. Вместо этого сообщество мобильных устройств перешло к моделированию движения узлов и пользователей. Разумеется, для моделирования такой модели необходимо получить модель движения. Самой распространенной из них является модель Random Waypoint. Это одна из наиболее популярных моделей перемещения узлов в сети, в силу своей простоты и доступности. В Random Waypoint Model узлы движутся по сети беспорядочно и свободно со случайной скоростью в любой момент времени в случайном направлении. Эта модель была впервые предложена Джонсоном и Мальцем в оценке протокола маршрутизации для MANET с топологией mesh Dynamic Source Routing (DSR), и позже была подтверждена той же исследовательской группой. Переработанная версия этой модели и стала стандартом де-факто в исследованиях мобильных устройств. Например, десять работ в ACM MobiHoc 2002 рассматривали мобильность узлов, а девять из них использовали модель Random Waypoint.

Как уже упоминалось, в модели Random Waypont узлы могут свободно перемещаться случайным образом в любом месте области моделирования сети независимо друг от друга. Это по сути означает, что для них нет какихлибо ограничений. Существует лишь случайный выбор точки назначения, скорости движения узла, направления движения и величины пауз между изменениями в направлении и скорости. Случайный выбор значений для перечисленных выше величин происходит из определенных заранее заданных числовых диапазонов. Моделирование начинается с задания величины паузы и, собственно, паузы всех узлов. После того, как мобильный узел «простоит» на месте в течение определенного периода времени (имеется в виду величина ранее заданной для него паузы), для него выбирается случайная точка назначения в пределах области моделируемой сети и случайная скорость, после чего он продолжает двигаться в направлении вновь выбранной цели с выбранной скоростью. По достижении места назначения, мобильный узел приостанавливается на определенный период времени перед повторным запуском того же самого процесса вновь. Здесь, скорость и время паузы оказывают влияние на подвижность узлов. Низкая скорость и долгое время паузы приводят к получению более стабильных топологии моделируемой сети, тогда как высокая скорость и короткое время паузы к более динамической топологии сети. На рисунке 7 иллюстрируется поведение узла в сети с моделью движения Random Waypoint.

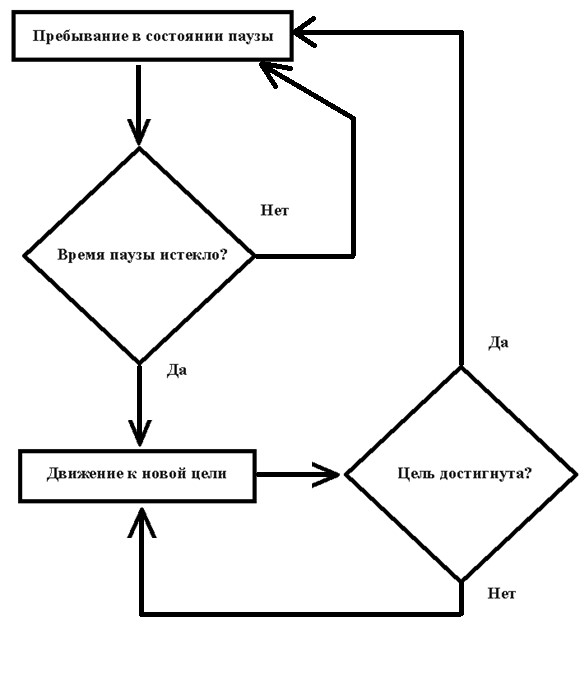


Рисунок 7 – Схема поведения узла в модели Random Waypoint

Алгоритм, реализующий модель Random Waypoint может быть описан в следующие пять шагов:

а) выбираем случайное место назначения в области моделируемой сети;

б) выбираем величину скорости (равномерная распределенная);

в) двигаем узел до тех пор, пока цель не будет достигнута им;

г) выжидаем величину паузы (равномерное распределение);

д) переходим к шагу 1.

## 1.7 Известные проблемы модели Random Waypoint

Всякая модель имеет определенные достоинства и недостатки, Random Waypoint не стала исключением из этого правила. Часто модель описывается как имеющая средняя скорость — просто потому, что скорости узла выбираются из равномерного распределения (0, Vmax) — или некоторого значения между 0 и Vmax, если мы рассматриваем время положительной паузы. Ожидается, что эта модель будет поддерживать эту среднюю скорость по мере развития симуляции, а результаты моделирования почти всегда будут в виде среднего за определенный период времени. Такие средние показатели имеют смысл только в том случае, если имитация достигает устойчивого состояния. К сожалению, это не так. Дело в том, что по мере моделирования с течением времени набор узлов перемещается все медленнее. Все больше узлов «застревает» на больших расстояниях на малых скоростях. Таким образом, модель не может обеспечить устойчивое состояние с точки зрения средней скорости. Накладные расходы и производительность мобильных систем обычно сильно зависят от мобильности узлов. В свете этого случайная точка пути может генерировать вводящие в заблуждение или неверные результаты. В частности, со временем средние результаты сильно меняются; Чем дольше запускается симуляция, тем больше отклоняются дальнейшие результаты. Существует много способов исправить модель. Один простой способ – ограничить минимальную скорость, а также максимальную.

# Практическая реализация

## Среда разработки

В качестве среды разработки была выбрана Java 8, JDK 1.8.0. Язык Java просто устроен и довольно удобен для разработки любых программ для решения практически любых задач. Простота устройства кода позволяет эффективно и легко писать, компилировать и отлаживать код программы.

Так как Java является объектно-ориентированным языком, даже в большей степени, чем C++ (почти все сущности языка являются объектами, за исключением некоторых типов), то существует возможность создавать удобно устроенные модульные программы, легко встраивать любые решения, в уже готовый проект, а так же эффективно устраивать организацию программных блоков, с логической точки зрения на решаемую задачу.

Одним из наиболее важных достоинств Java является кроссплатформен ность, программы, написанные на Java не зависят от платформы как на уровне исходного кода, так и на двоичном уровне, что позволяет запускать их на разных системах. Это достигается за счет перевода написанного кода в специальный байт-код и последующего его выполнения Java-машиной. Так же несомненный плюсом Java является возможность намного более легкой и простой разработки программ, которые не содержат скрытых ошибок, чем на том же C++. Создателями языка анализировались, так называемые «узкие места» исходного кода программ на C++, вследствие чего были выявлены проблемы, которые приводят к появлению ошибок, которые достаточно трудно выявить. Как следствие, проектирование языка Java было организовано таким образом, что появление скрытых ошибок в коде сведено практически к нулю. Для этого были приняты такие меры, как отсутствие в языке Java указателей, множественного наследования, которое было заменено интерфейсами, неявные преобразования типов с потерей точности, исключено явное выделение памяти, как следствие, программист не должен очищать память «руками», за него это сделает сборщик мусора, который специально предназначен для очищения не используемых областей памяти.

Таким образом получаем еще один крупный плюс. Программист застрахован от ошибок, связанных с неверной работой с памятью. Так же в языке присутствует строгая типизация и обязательный контроль исключений (исключительных ситуаций). Множество логических ошибок будет обнаружено еще на этапе компиляции. Многие задачи, которые возникают при разработке программного обеспечения уже реализованы и существуют в качестве стандартных библиотек, что позволяет использовать множестве удобных решений и коллекций.

Так же для языка существует просто огромное количество бесплатных библиотек, которые дают множество возможностей программисту. Так же стоит упомянуть явные плюсы и достоинства Intellij IDEA, на которой писался код программы. Главная вещь, отличающая IDEA, это понимание контекста. Именно это имеют в виду сотрудники JetBrains, когда называют свой продукт «интеллектуальным» (intelligent). Дело в том, что IDEA индексирует весь проект, анализирует всё, что в нём есть, и даже строит синтаксическое дерево. Благодаря этому IDEA в любой момент, куда бы не был поставлен разработчиком курсор, знает, в каком конкретно участке кода мы находимся и что тут происходит. Умение понимать контекст выражается в довольно многих аспектах, но наиболее полезными такие возможности являются при отладке кода, так как мы можем видеть значения переменных или выражений, при автозаполнении, потому как Idea всегда предложит программисту именно нужное имя метода, переменной, параметра функции и так далее, а так же при рефакторинге, потому как IDEA учитывает название метода, тип переменной, значение, названия подобных переменных в других местах программы, и те названия, которые программист давали подобным переменным ранее.

IDEA действительно является мощнейшим редактором исходного кода. Здесь есть классические подсветки синтаксиса и несколько буферов обмена. Переходы к определению, автоматическое добавление нужных импортов, а также оптимизация импортов с удалением ненужных, встроенное отображение документации и типов, подсказка по аргументам, переименование аргументов, переменных и модулей буквально за секунду, быстрое указание возвращаемого типа у метода, множество подсказок, всплывающих в ходе написания кода, даже существует автоисправление кода.

## Представления графов в практической реализации

Алгоритмические проблемы, рассматриваемые ниже, описываются входом, как правило, состоящим из нескольких графов (или перестановок) того или иного типа, и выходом, который описывается предикатом (для проблем распознавания) или несколькими графами или же перестановками. Для заданного входа алгоритм совершает некоторое количество элементарных операций в течение заданного времени и завершает работу, создавая выход предписанного типа. Количество элементарных операций в худшем случае именуется далее сложностью алгоритма. Вход и выход имеют длину которая, грубо говоря, равна размеру памяти, в которой он может быть записан. Поэтому сложность алгоритма является функцией длины входа. Алгоритм имеет полиномиальную сложность (работает полиномиальное время), если существует полином ( ) такой, что сложность алгоритма на входе длины не превосходит ( ); в этом случае также говорят, что сложность алгоритма есть (1). Граф = ( , ) на входе алгоритма может быть задан с помощью списка рёбер(инцидентности) и тогда вклад в длину входа равен длине этого списка 2| | = ( 2), где = | |. На рисунке 8 приводится в качестве примера скриншот из программы, где граф, задан в виде списка инцидентности.

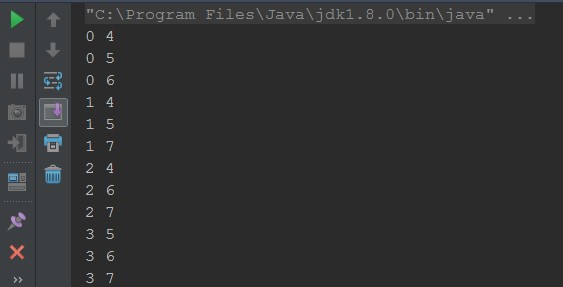


Рисунок 8 – Граф, заданный списком инцидентности

Другой способ состоит в задании матрицы смежности = ( ) графа , т.е. квадратной × матрицы, определяемой формулой 2 следующим образом:

Auv =  (2)

В этом случае вклад в длину входа равен 2. Произвольная биекция :

1 → 2 задаётся множеством пар ( , ); её вклад в длину входа считается равным 2| 1|. На рисунке 9 приводится скриншот, который изображает пример программного представления графа в виде матрицы смежности.

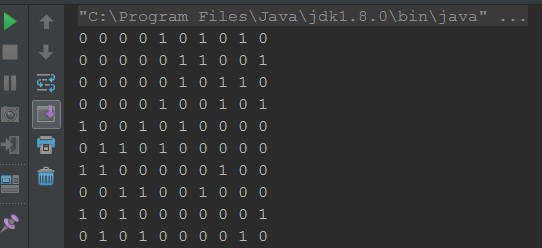


Рисунок 9 – Граф, заданный матрицей смежности

## Оценка сложности решений поиска мостов, точек сочленения и компонент связности, приведенных в программе

Поиск точек сочленения, так же как и мостов, наряду с поиском пути из узла в узел основаны на алгоритме обхода геометрического графа в глубину, сложность которого может быть оценена как O(n+m), где n — количество вершин, m — рёбер. Алгоритм Тарьяна, использовавшийся для поиска сильных компонент связности имеет подобную сложность(O(n+m)).

## Структура программы

### Основные блоки программы

В ходе реализации программы, имитирующей ad hoc сеть с моделью движения узлов Random Waypoint был создан главный класс Network, а так же еще семь классов (два основных: Node – для узлов сети, Destination\_Point – для точек назначения и пять дополнительных: Pair\_Of\_Components – для хранения пар компонент связности, Bridge – для мостов, Answer – для типа

связи между узлами, Way – для путей между узлами и Time\_Of\_Existence\_And\_Absence\_Of\_A\_Routes – для времен существования и отсутствия маршрутов из каждого узла в каждый).

Полями класса Network являются переменные для количества узлов quantity\_nodes и границ сети border\_network, границ скоростей узлов border\_min\_speed и border\_max\_speed, радиусов узлов border\_min\_radius и border\_max\_radius, верхней границы пауз узлов border\_max\_pause, так же логические поля для обозначения характеристик геометрического графа сети(связный или нет, ориентированный или нет) is\_oriented и is\_connected, список узлов сети nodes с элементами класса Node, матрица смежности matrix\_of\_links(содержит сведения о связях между узлами сети), версия матрицы смежности(содержит все ребра графа сети, за исключением лишь ребер, которые связывают слабо связные вершины) для алгоритма поиска точек сочленения и кандидатов среди ребер геометрического графа моделируемой сети в мосты version\_matrix\_of\_links\_for\_algorithm\_finding\_ connection\_points\_and\_bridges и их размерность quantity\_nodes (количество узлов), а так же список смежности list\_adjacency, список для компонент связности графа connectivity\_components, множество для точек сочленения connection\_points, список для мостов bridges, список расстояний между компонентами связности графа distances\_between\_components и список словарей, содержащий времена существования и отсутствия маршрутов между узлами times. Так же в поля класса входят переменные для подсчетов средних значений при моделировании: компонент связности, мостов, точек сочленения, времен существования маршрутов и отсутствия маршрутов между заданной парой узлов, расстояния между компонентами графа – connectivity\_components\_average\_value, bridges\_average\_value, connection\_ points\_average\_value, time\_exist\_of\_the\_route\_average\_value, time\_not\_exist\_ of\_the\_route\_average\_value, distance\_between\_components\_ average\_value. В методы класса входят следующие функции:

public Network(int border\_network, int quantity\_nodes, int border\_min\_

speed, int border\_max\_speed, int border\_pause, int border\_min\_radius, int border\_ max\_radius) – конструктор класса, инициализирует поля принимаемыми значениями и запускает метод старта сети. Принимает в качестве параметров границу для сети, количество узлов, границы для скорости, границу для паузы, границы для радиуса;

* private void Initialize\_Matrix\_Of\_Links() – функция инициализирует матрицу связей;
* private void Initialize\_Times\_Of\_Existence\_And\_Absence\_Of\_A\_ Routes() – функция инициализирует список словарей, который содержит времена существования и отсутствия маршрутов из каждого узла в каждый;
* private void Network\_Start() – функция "запускает" сеть – создаются, инициализируются начальными значениями и активируются узлы;
* private void Control\_Node(Node node) – функция осуществляет управление поданным узлом. Принимает экземпляр класса узла;
* private void Activate\_Node(Node node) – функция активизации узла – задается случайная скорость, направление, пункт назначения и узел начинает движение. Принимает экземпляр класса узла;
* private int Degree\_Of\_Vertex(int vertex) – функция определяет степень вершины. Принимает вершину геометрического графа(представление узла сети);
* private Answer Check\_The\_Existence\_Of\_A\_Link(double x1, double y1, double radius1, double x2, double y2, double radius2) – функция проверяет, есть ли соединение между узлами. Принимает координаты первого узла, радиус первого узла, координаты второго узла и радиус второго узла;
* private void Generate\_Matrix\_Of\_Link() – функция генерирует матрицу связей;
* public void Show\_Info\_About\_Current\_Node(int index) – функция выводит информацию о указанном узле в консоль. Принимает индекс узла;
* public void Show\_Matrix\_Of\_Link() – функция выводит матрицу связей в консоль;
* public void Show\_Paths() – фукнция выводит в консоль информацию о путях от каждого узла к остальным;
* private void Show\_List\_Adjacency() – функция выводит список смежности в консоль;
* private void Show\_Times\_Exist\_And\_Absence\_Of\_A\_Paths() – функция выводит список времен существования и отсутствия маршрутов между узлами в консоль;
* private void Show\_Connection\_Point() – функция выводит список точек сочленения в консоль;
* private void Show\_Bridges() – функция выводит список мостов в консоль;
* private void Define\_The\_Type\_Of\_Graph\_By\_The\_Orientation\_

Characteristic() – функция определяет ориентированный ли у нас геометрический граф или нет;

* private void Define\_The\_Type\_Of\_Graph\_By\_The\_Connectivity\_

Characteristic() – функция определяет связный ли у нас геометрический граф или нет;

* private void Update\_Paths() – функция обновляет информацию о путях из каждого узла в остальные;
* private boolean Way\_Is\_Exist(int v1, int v2) – функция-обертка для функции поиска в глубину (DFS). Принимает вершину из которой ищем путь и вершину, в которую ищем путь;
* private Double Find\_Distance\_Between\_Nodes(int index\_node\_1, int index\_node\_2) – функция возвращает длину пути(ребра) между

узлами(вершинами). Принимает индексы узлов;

private boolean DFS(int v1, int v2, boolean[] used, Way way) – функция

проверяет наличие пути из вершины v1 в вершину v2. Принимает номера вершин, массив уже посещенных вершин и экземпляр класса пути. Возвращает истину или ложь в зависимости от существования пути и так же попутно в ходе своей работы записывает в поля класса пути пройденные расстояния и посещенные вершины (так же происходит их удаление, так называемые откаты, если поиск пути производился в неверном направлении и в итоге пришли в такую вершину, из которой невозможно продолжать пути в указанную);

* private void Initialize\_Version\_Matrix\_Of\_Links\_For\_Algorithm\_ Finding\_Connection\_Points\_And\_Bridges() - функция инициализирует версию матрицы связей для алгоритма поиска точек сочленения и кандидатов в мосты ориентированного геометрического графа сети;
* private Void Find\_Connection\_Point() – функция-обертка для модифицированной функции поиска в глубину (DFS\_Modified);
* private void DFS\_Modified(int x, int parent, int[] number, int[] low, int counter) – функция ищет точки сочленения и кандидатов в мосты в геометрическом графе сети;
* private void Find\_Bridges() – функция ищет мосты в геометрическом графе сети;
* private void Find\_Connectivity\_Components() – функция-обертка для функции dfs;
* private dfs(int v, boolean[] used, ArrayList<Integer> comp) – функция ищет компоненты связанности в геометрическом графе сети;
* private void Find\_Strongly\_Connectivity\_Components() – функцияобертка для алгоритма Тарьяна;
* private void Tarjan\_Algorithm(Integer v, Boolean used[], HashMap

<Integer, ArrayList<Integer>> result, Integer id, Integer lowlinks[], Integer indices[], ArrayDeque<Integer> stack) – поиск в глубину, приспособленный под нахождение сильных компонент связности в геометрическом графе сети;

* private void Show\_Results(HashMap<Integer, ArrayList<Integer>> map) – функция выводит в консоль компоненты связности. Принимает словарь с результатами поисков;
* private double Find\_Minimum\_Distance(ArrayList<Integer> component\_i, ArrayList<Integer> component\_j) – функция находит минимальное расстояние между двумя компонентами геометрического графа сети. Принимает списки компонент связности;
* private void Find\_Distance\_Between\_Components() – функция ищет расстояния между всеми компонентами связности геометрического графа сети;
* private static Network Read\_Input\_Data() – функция производит считывание входных данных с консоли;
* private void Network\_Work(int time, int interval) – функция работы сети.

Принимает время, в течение которого должна происходить имитация.

Полями класса Node являются переменные для координат узла на плоскости по оси абсцисс и оси ординат x и y, скорости speed, паузы pause, радиуса radius, логические переменные для обозначений состояний узла на текущий момент времени(находится ли узел в движении к случайно заданной точке, достиг ли узел пункта назначения и ожидает паузы, узел находится в состоянии паузы или же пауза истекла и узел готов к новой отправке) in\_move, paused, time\_pause\_is\_over, переменная для объекта класса точки назначения(Destination\_Point), переменные для расстояния до пункта назначения и уже пройденного расстояния distance и distance\_traveled, для времени count\_steps, необходимого на путь до пункта назначения, смещений по оси абсцисс и ординат в момент времени offset\_x и offset\_y и словаря с путями для конкретного узла к другим узлам ways. В методы класса входят следующие функции:

public Node(int speed, double radius) – конструктор класса, принимает на

вход скорость для узла и радиус;

* public void Set\_Coordinates(double x, double y) – функция задает координаты узла на плоскости. Принимает значения координат;
* public void Set\_Speed(int speed) – функция задает скорость узла;
* public double Get\_X() – функция возвращает координату узла на плоскости по оси абсцисс. Принимает значение скорости;
* public double Get\_Y() – функция возвращает координату узла на плоскости по оси ординат;
* public int Get\_Speed() – функция возвращает значение скорости узла;
* public double Get\_Radius() – функция возвращает значение радиуса узла;
* public int Get\_Pause() – функция возвращает текущее значение паузы узла;
* HashMap<Integer, Way> Get\_Ways() – функция возвращает словарь, хранящий пути от этого узла к другим;
* public void Assign\_Pause (int pause) – функция задает узлу значение паузы. Принимает значение паузы;
* public void Pause() – функция "паузит" узел;
* public boolean Time\_Pause\_Is\_Over() – функция возвращает значение логической переменной time\_pause\_is\_over(истекло ли время паузы);
* public boolean In\_Move() – функция возвращает значение логической переменной in\_move(находится ли узел в движении);
* public boolean Paused() – функция возвращает значение логической переменной paused(находится ли узел в состоянии паузы);
* public void Calculating\_Parameters\_Of\_Movement(Destination\_Point destination\_point) – функция рассчета параметров движения узла. Принимает экземпляр класса точки назначения. Производит расчет параметров движения узла: времени движения, расстояния до точки назначения и смещений для узла в момент времени;
* public void Move() – функция движения узла. Осуществляет движение узла в сети (по плоскости), смещая его на заранее высчитанные величины в каждый момент времени и контролирует, чтобы узел не «пролетел» точку назначения.

Полями класса Destination\_Point являются переменные для координат точки на плоскости x и y и переменная для объекта класса Random. В методы класса входят следующие функции:

* public Destination\_Point(int border\_network) – конструктор класса, принимает границу сети;
* public int Get\_X() – функция возвращает координату узла на плоскости по оси абсцисс;
* public int Get\_Y() – функция возвращает координату узла на плоскости по оси ординат.

Полями класса Pair\_Of\_Components являются переменные для списков компонент связности component\_1 и component\_2. В методы класса входят следующие функции:

* public Pair\_Of\_Components(ArrayList<Integer> component\_1, ArrayList <Integer> component\_2) – конструктор класса, принимает списки с компонентами связности геометрического графа сети;
* public ArrayList<Integer> Get\_Component\_1() – функция возвращает первую компоненту пары;
* public ArrayList<Integer> Get\_Component\_2() – функция возвращает вторую компоненту пары.

Полями класса Bridge являются переменные для вершин – «опор» моста vertex\_1(начало моста) и vertex\_2(конец моста). В методы класса входят следующие функции:

public Brigde(int vertex\_1, int vertex\_2) – конструктор класса, принимает

на вход индексы вершин – «опор» моста;

* public int Get\_Vertex\_1() – функция возвращает первую «опору»

(начало) моста;

* public int Get\_Vertex\_2() – функция возвращает вторую «опору»

(конец) моста.

Полями класса Answer являются переменные для типа соединения между узлами connection\_type и для индекса вершины (требуется для уточнения при одностороннем виде соединения) number\_of\_vertexes. В методы класса входят следующие функции:

* public Answer(String connection\_type, int ...number\_of\_vertex) – конструктор класса, принимает на вход строку с типом соединения между узлами и индекс узла(при одном из типов соединения);
* public String Get\_Connection\_Type() – функция возвращает тип соединения между узлами;
* public int Get\_Index\_Of\_Vertex() – функция возвращает номер узла для уточнения ответа функции проверки типа связи.

Полями класса Way являются переменные для длины пути way\_length, тип BigDecimal для вычислений с повышенной точностью и для списка вершин, составляющих путь. В методы класса входят следующие функции:

* public Way(BigDecimal way\_length, ArrayList<Integer> way) – конструктор класса, принимает длину пути и список вершин way;
* public BigDecimal Get\_Way\_Length() – функция возвращает длину пути;
* public ArrayList<Integer> Get\_Way() – функция возвращает список вершин(путь);
* public double Get\_Time\_Exist\_Of\_The\_Route() – функция возвращает значение времени жизни пути;
* public double Get\_Time\_Not\_Exist\_Of\_The\_Route() – функция возвращает значение времени отсутствия пути;
* public void Increase\_Way\_Length(BigDecimal value) – функция увеличивает значение длины пути на поданную величину;
* public void Increase\_The\_Time\_Exist\_Of\_The\_Route() – функция увеличивает значение времени жизни пути;
* public void Increase\_The\_Time\_Not\_Exist\_Of\_The\_Route() – функция увеличивает значение времени отсутствия пути;
* public void Decrease\_Way\_Length(BigDecimal value) – функция уменьшает значение длины пути на поданную величину/

Полями класса Time\_Of\_Existence\_And\_Absence\_Of\_A\_Routes являются переменные для времени существования и отсутствия маршрута time\_exist\_ of\_the\_route и time\_not\_exist\_of\_the\_route. В методы класса входят следующие функции:

* public Time\_Of\_Existence\_And\_Absence\_Of\_A\_Routes() – конструктор класса;
* public double Get\_Time\_Exist\_Of\_The\_Route() – функция возвращает значение времени жизни пути;
* public double Get\_Time\_Not\_Exist\_Of\_The\_Route() – функция возвращает значение времени отсутствия пути;
* public void Increase\_The\_Time\_Exist\_Of\_The\_Route() – функция увеличивает значение времени жизни пути;
* public void Increase\_The\_Time\_Not\_Exist\_Of\_The\_Route() – функция увеличивает значение времени отсутствия пути.

Ниже, на рисунках 10 и 11 приведем диаграмму классов, построенную в среде разработки проекта – IntellijIDEA, чтобы отобразить связи классов в проекте:

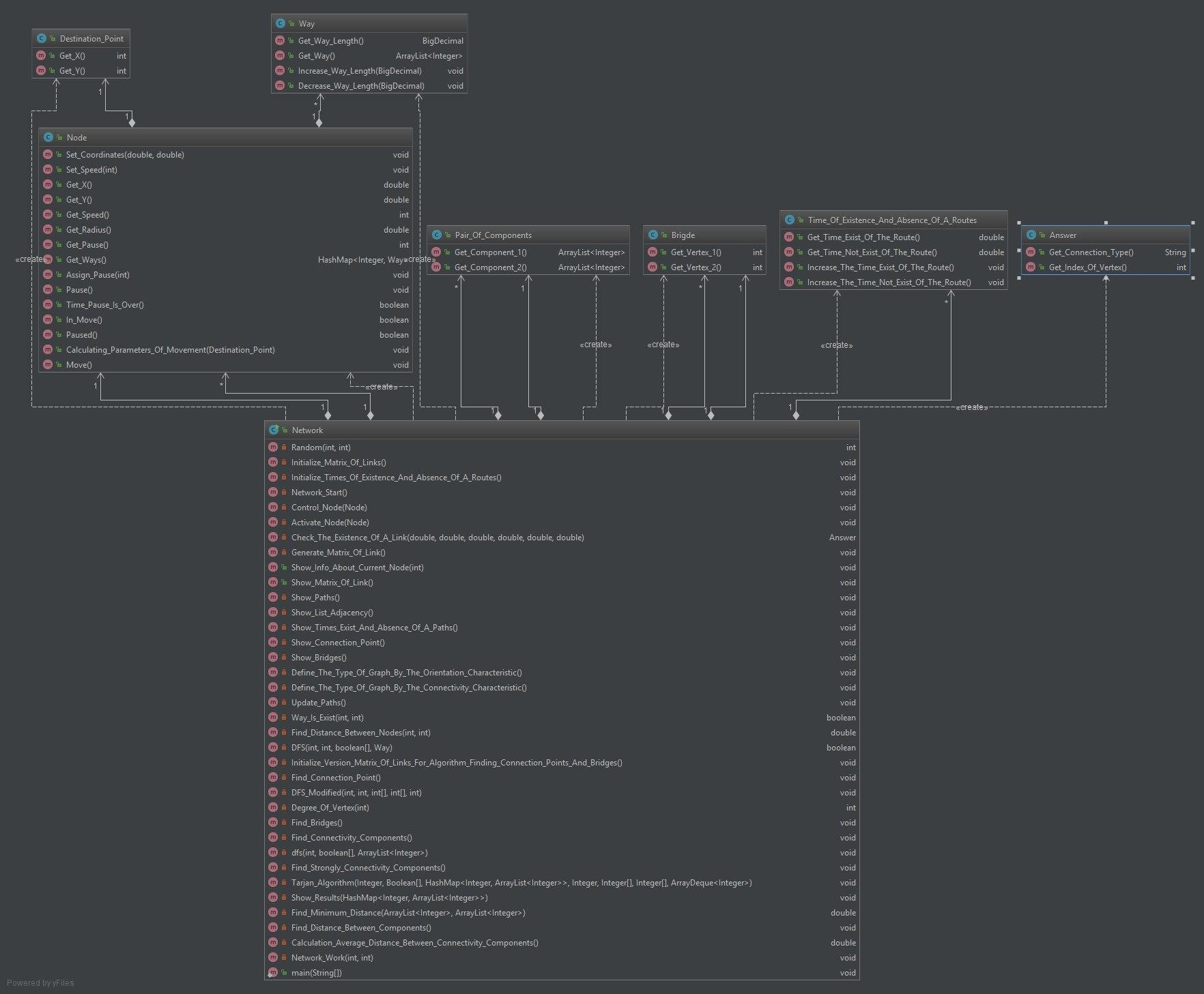


Рисунок 10 – Диаграмма классов проекта. Методы

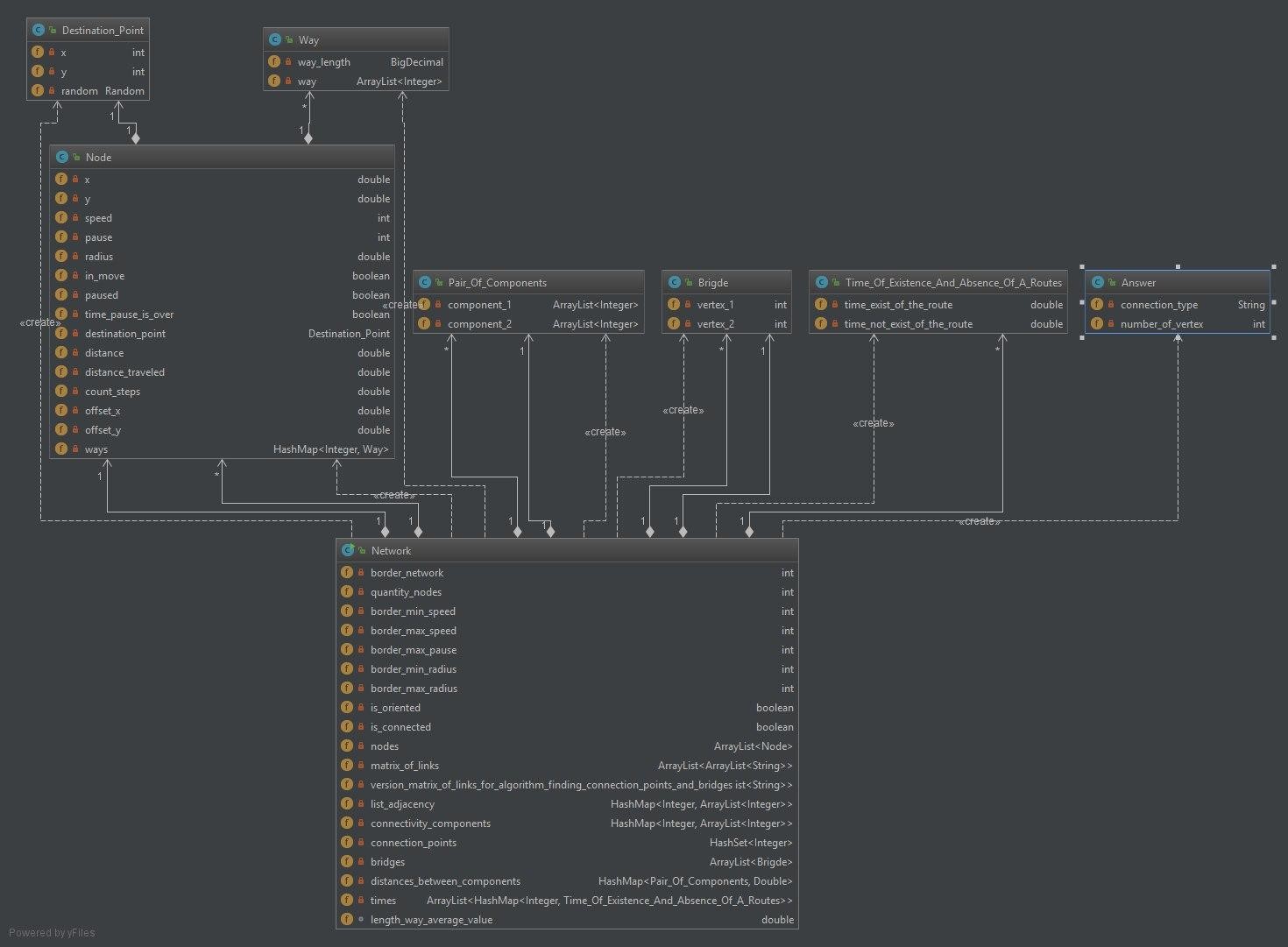


Рисунок 11 – Диаграмма классов проекта. Переменные

### Описание работы программы

Была реализована имитация сети с моделью перемещения Random Waypoint. В начале работы программа принимает на вход значения для границы сети, количества узлов, для границ скоростей, максимальной паузы, радиусов узлов и время работы, затем начинается само моделирование. Создается экземпляр класса сети, создаются экземпляры класса узла в количестве заданном несколько ранее при вводе. Для каждого узла назначается пауза, и начинается общая работа программы, а именно: программа работает до тех пор, пока не истечет указанное время моделирования – в каждый момент времени она проходит в цикле по узлам и проверяет состояние каждого из них. Если узел находится в состоянии паузы, то всякий раз вызывается метод Pause(), который производит уменьшение текущего значения паузы на единицу внутри программного времени. Пауза истекает когда ее значение становится равно 0. Если узел закончил пребывание в состоянии паузы и готов к отправке, то ему назначаются случайная скорость из заданного при вводе диапазона, а так же выбирается случайным образом точка направления на плоскости.

Далее происходит расчет параметров движения узла к заданной точке в сети: вычисляется расстояние до точки, вычисляется время движения, и смещения для координат узла в каждый момент времени. После окончания расчетов параметров передвижения узла к пункту назначения, происходит «активизация» узла, он начинает движение к цели. Пока узел находится в состоянии движения, всякий раз вызывается метод Move(), который производит увеличение(уменьшение, зависит от направления узла и местонахождения точки назначения на плоскости) координат узла на рассчитанное ранее смещение. Так же осуществляется контроль над случаем, когда скорость узла больше оставшегося расстояния до пункта назначения на плоскости. В этом случае перемещение узла происходит на оставшееся до назначенной точки расстояние. Таким образом гарантируется, что узел «не пролетит» точку назначения. Если узел достиг точки назначения, он вводится в состояние случайно выбранной из заранее заданного диапазона паузы.

После инициализации всех узлов сети происходит составление матрицы связей (матрица смежности геометрического графа сети), составление списка смежности, проверка наличия и расчет путей между каждой парой вершин, поиск времени существования и отсутствия таких путей, определяется по радиусам узлов, ориентированный граф или нет (в случае одинаковых значений величин радиусов для всех узлов имеем неориентированный граф, если же радиусы разные – то ориентированный, с той оговоркой, что при некоторых редких случаях при разных значениях радиусов мы все же можем получить неориентированный граф, если в сети отсутствуют такие пары узлов, которые связаны ориентированным в одну сторону ребром (иначе можно говорить, что отсутствует второе ориентированное ребро, которое направлено в противоположную сторону), то есть имеют одностороннюю связь, а есть только пары не связанных узлов и пары узлов, которые связаны двусторонней связью, то есть радиусы обоих узлов в конкретной паре больше расстояния между узлами), происходит поиск компонент связности для графа (компоненты в случае неориентированного и компоненты сильной связности в случае ориентированного графа), далее по количеству найденных компонент связности геометрического графа сети определяется связный граф или нет. В случае если наш геометрический граф не связен, есть необходимость найти расстояния между всеми его компонентами. Помимо перечисленных выше процедур, происходит так же поиск точек сочленения и мостов в геометрическом графе сети с предварительным составлением копии матрицы связей геометрического графа (в случае ориентированного графа нам необходимо учитывать лишь сильную связность, поэтому все ребра, связывающие слабо связные вершины исключаются из матрицы смежности).

### Анализ результатов

Были проведены десять тысяч испытаний на случайных геометрических графах. Каждое испытание длилось десять тысяч тиков времени (внутри программное время). Размер сети 500 x 500. Количество узлов в сети 100.

Радиус узлов в диапазоне [25, 50]. Сначала диапазоны скоростей узлов были постоянны: [25, 50], [50, 75], [75, 100], [100, 125], [125, 150], [150, 175], [175, 200], [200, 225], [225, 250], а верхняя граница для паузы узлов увеличивалась с шагом 5, от 5 до 55 соответственно. Были выявлены следующие зависимости:

При постоянной скорости узлов и увеличении максимальной верхней границы для паузы, количество компонент связности, точек сочленения и мостов в графе увеличивалось. Данные зависимости видны на графиках (рисунки 12 – 18):

0

1

2

3

4

5

6

7

0

10

20

30

40

50

60

Среднее количество компонент

Максимальная пауза узлов

Рисунок 12 – График зависимости среднего количества компонент связности от паузы узлов. Скорость узлов в диапазоне от 25 до 50

Мы можем увидеть, что ситуация подобна для всех постоянных диапазонов скоростей:

0

5

,

0

1

5

1

,

2

5

,

2

3

5

,

3

4

5

,

4

5

0

10

20

30

40

50

60

Среднее количество компонент

Максимальная пауза узлов

Рисунок 13 – График зависимости среднего количества компонент связности

от паузы узлов. Скорость узлов в диапазоне от 50 до 75

0

1

2

3

4

5

6

0

10

20

30

40

50

60

Среднее количество

компонент

Максимальная пауза узлов

Рисунок 14 – График зависимости среднего количества компонент связности

от паузы узлов. Скорость узлов в диапазоне от 75 до 100

0

5

,

0

1

5

1

,

2

5

,

2

3

5

,

3

4

5

,

4

5

0

10

20

30

40

50

60

Среднее количество компонент

Максимальная пауза узлов

Рисунок 15 – График зависимости среднего количества компонент связности

от паузы узлов. Скорость узлов в диапазоне от 100 до 125

И так далее, приведем график для последнего набора из данной серии замеров:

0

1

2

3

4

5

6

7

0

10

20

30

40

50

60

Среднее количество

компонент

Максимальная пауза узлов

Рисунок 16 – График зависимости среднего количества компонент связности от значения узлов. Скорость узлов в диапазоне от 225 до 250

0

5

,

0

1

1

5

,

2

2

,

5

3

5

,

3

4

5

,

4

0

10

20

30

40

50

60

Среднее количество точек

сочленения и мостов

Максимальная пауза узлов

Точки сочленения

Мосты

Рисунок 17 – График зависимости среднего количества точек сочленения и мостов от значения паузы у узлов. Скорость узлов в диапазоне от 25 до 50

Ситуация тут, как и с графиками выше повторяется, количество возрастает на всех диапазонах скоростей. Приведем график для последнего диапазона скоростей:

0

1

2

3

4

5

6

0

10

20

30

40

50

60

Среднее количество точек

сочленения и мостов

Максимальная пауза узлов

Точки сочленения

Мосты

Рисунок 18 – График зависимости среднего количества точек сочленения и мостов от значения паузы узлов. Скорость узлов в диапазоне от 225 до 250 Зависимость для количества компонент связности, точек сочленения и мостов от скорости узлов не была выявлена.

Перейдем теперь к рассмотрению времени жизни маршрута и времени его отсутствия. При постоянной скорости узлов и увеличении максимальной верхней границы для паузы, время существования маршрута между парой узлов уменьшалось, а время отсутствия маршрута, напротив, увеличивалось, что будет видно на графиках 19 – 27:

0

20

40

60

80

100

120

140

0

10

20

30

40

50

60

Среднее время жизни маршрута

Максимальная пауза узлов

Рисунок 19 – График зависимости среднего времени жизни маршрута от значения паузы узлов. Скорость узлов в диапазоне от 25 до 50

0

20

40

60

80

100

120

140

160

180

0

10

20

30

40

50

60

Среднее время жизни маршрута

Максимальная пауза узлов

Рисунок 20 – График зависимости среднего времени жизни маршрута от значения паузы узлов. Скорость узлов в диапазоне от 50 до 75

0

20

40

60

80

100

120

140

160

180

0

10

20

30

40

50

60

Среднее время жизни маршрута

Максимальная пауза узлов

Рисунок 21 – График зависимости среднего времени жизни маршрута от значения паузы узлов. Скорость узлов в диапазоне от 75 до 100

0

20

40

60

80

100

120

140

160

180

0

10

20

30

40

50

60

Среднее время жизни маршрута

Максимальная пауза узлов

Рисунок 22 – График зависимости среднего времени жизни маршрута от значения паузы узлов. Скорость узлов в диапазоне от 225 до 250

0

20

40

60

80

100

120

140

160

180

200

0

10

20

30

40

50

60

Среднее время отсутствия

маршрута

Максимальная пауза узлов

Рисунок 23 – График зависимости среднего времени отсутствия маршрута от значения паузы узлов. Скорость узлов в диапазоне от 25 до 50

0

20

40

60

80

100

120

140

160

180

0

10

20

30

40

50

60

Среднее время отсутствия

маршрута

Максимальная пауза узлов

Рисунок 24 – График зависимости среднего времени отсутствия маршрута от значения паузы узлов. Скорость узлов в диапазоне от 50 до 75

0

20

40

60

80

100

120

140

160

180

0

10

20

30

40

50

60

Среднее время отсутствия

маршрута

Максимальная пауза узлов

Рисунок 25 – График зависимости среднего времени отсутствия маршрута от значения паузы узлов. Скорость узлов в диапазоне от 75 до 100

0

20

40

60

80

100

120

140

160

180

0

10

20

30

40

50

60

Среднее время отсутствия

маршрута

Максимальная пауза узлов

Рисунок 26 – График зависимости среднего времени отсутствия маршрута от значения паузы узлов. Скорость узлов в диапазоне от 100 до 125

Приведем график с последним диапазоном скорости (рисунок 26):

0

20

40

60

80

100

120

140

160

180

0

10

20

30

40

50

60

Среднее время отсутствия

маршрута

Максимальная пауза узлов

Рисунок 27 – График зависимости среднего времени отсутствия маршрута от значения паузы узлов. Скорость узлов в диапазоне от 225 до 250

Теперь рассмотрим зависимость средней длины маршрута между парой вершин в зависимости от значений скорости. Паузы были постоянны. Как будет видно из графиков ниже (Рисунки 28 – 30), средняя длина маршрута будет расти с увеличением скорости узлов.

0

100

200

300

400

500

600

0

50

100

150

200

250

300

Средняя длина маршрута

Скорость узлов

Рисунок 28 – График зависимости средней длины маршрута между парой узлов от значения скорости узлов. Максимальная пауза 5

0

50

100

150

200

250

300

350

400

450

500

0

50

100

150

200

250

300

Средняя длина маршрута

Скорость узлов

Рисунок 29 – График зависимости средней длины маршрута между парой узлов от значения скорости узлов. Максимальная пауза 10

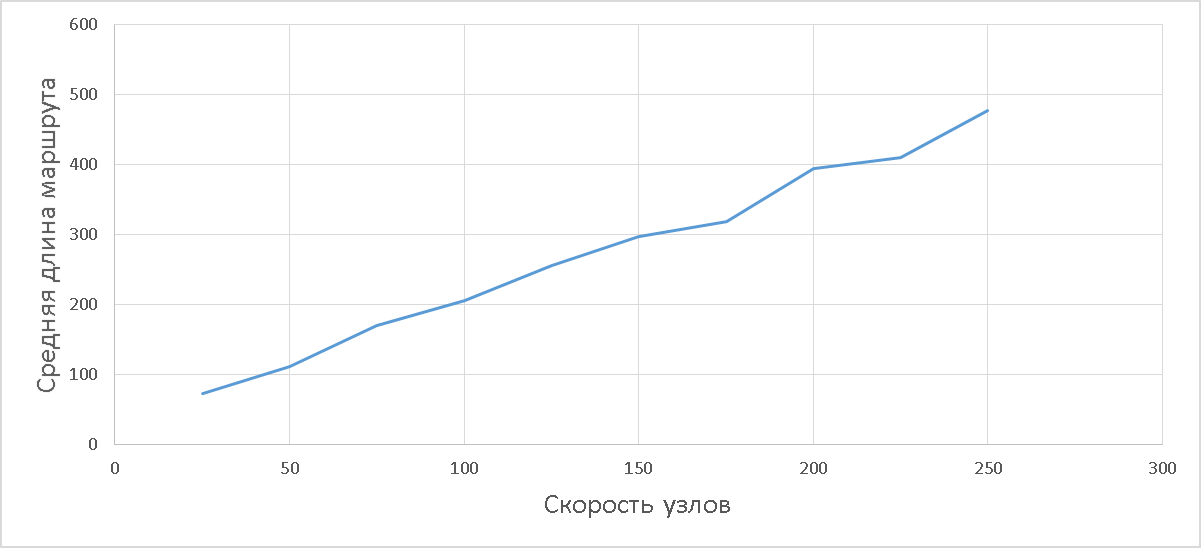


Рисунок 30 – График зависимости средней длины маршрута между парой узлов от значения скорости узлов. Максимальная пауза 55 ЗАКЛЮЧЕНИЕ

Во время выполнения дипломной работы была изучена и сымитирована модель движения Random Waypoint. Были произведены сравнения различных характеристик сети в разные моменты времени с разными диапазонами значений для скоростей и пауз узлов и сделаны выводы. Ясно, что значение пауз узлов оказывает влияние на такие параметры сети, как количество точек сочленения, мостов, компонент связности, а так же на время жизни маршрута между парой узлов и время отсутствия маршрута между парой узлов. Скорость узлов оказывает влияние на среднюю длину маршрута между парой узлов.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Зыков А. А. Основы теории графов. // М.: Наука, 1986. — 456 с.
2. Белов В.В., Воробьев Е.М., Шаталов В.Е. Теория графов. // М.:

Высш.шк., 1976. — 392 с.

1. Емеличев В.А., Мельников О.И., [и др.] Лекции по теории графов. // М.: Наука, 1990. — 384 с.
2. МАРШРУТИЗАЦИЯ В БЕСПРОВОДНЫХ МОБИЛЬНЫХ AD HOC-

СЕТЯХ [Электронный ресурс]. URL: http://cyberleninka.ru/article/n/marshruti zatsiya-v-besprovodnyh-mobilnyh-ad-hoc-setyah (дата обращения: 06.02.2017).

1. Оре О. Графы и их применение. // М.: URSS, 2006. — 268 с.
2. Миков А.И., Мезенцева А.С. Характеристики геометрических графов, моделирующих AD HOC сети // Информатизация и связь. — 2012. — № 5. — с. 85—88.
3. Харари Ф. Теория графов. М.: Эдиториал УРСС, 2003.
4. Что такое MANET или почему WiFi не решение всех телекоммуникационных проблем [Электронный ресурс]. URL: https://habrahabr.ru/post/197860/ (дата обращения: 01.04.2017).
5. Random Waypoint Considered Harmful [Электронный ресурс]. URL: http://web.eecs.umich.edu/~mingyan/pub/infocom03.pdf (дата обращения:

06.02.2017).

1. Random Waypoint [Электронный ресурс]. URL: http://www.routingprotokolle.de/Routing/mobility\_random\_waypoint.htm (дата обращения: 07.05.2017).

ПРИЛОЖЕНИЕ А

# Код программы

import java.io.\*; import java.math.BigDecimal;

import java.util.\*;

/\*\*

\* Created by Vitalik on 06.03.2017.

\*/

public class Network

{ private int border\_network; /\*\* Верхняя граница для размеров сети(Random)

\*\*/

private int quantity\_nodes; /\*\* Количество узлов \*\*/ private int border\_min\_speed; /\*\* Нижняя граница для скорости(Random) \*\*/ private int border\_max\_speed; /\*\* Верхняя граница для скорости(Random)

\*\*/ private int border\_max\_pause; /\*\* Верхняя граница для паузы(Random) \*\*/ private int border\_min\_radius; /\*\* Нижняя граница для радиуса(Random) \*\*/ private int border\_max\_radius; /\*\* Верхняя граница для радиуса(Random) \*\*/ private boolean is\_oriented; /\*\* ориентированный граф или нет \*\*/ private boolean is\_connected; /\*\* Граф связный или нет(Имеет ли ровно одну компоненту связности) \*\*/

private ArrayList<Node> nodes = new ArrayList<Node>(); /\*\* Список узлов сети \*\*/ private ArrayList<ArrayList<String>> matrix\_of\_links; /\*\* Содержит информацию о наличии соединения между узлами "-" - соединения нет, "tw" - соединение двусторонее, "ow" - соединение односторонее (матрица смежности)\*\*/

private ArrayList<ArrayList<String>> version\_matrix\_of\_links\_for\_algorithm\_finding\_connection\_points\_and\_bridges;

/\*\* Версия матрицы смежности для алгоритма поиска точек сочленения и мостов(Будут исключены все ребра(односторонние), которые соединяют слабо связные вершины) \*\*/ private HashMap<Integer, ArrayList<Integer>> list\_adjacency = new HashMap<Integer, ArrayList<Integer>>(); /\*\* Список смежности \*\*/ private HashMap<Integer, ArrayList<Integer>> connectivity\_components = new HashMap<Integer, ArrayList<Integer>>(); /\*\* Список компонент связности

\*\*/

private HashSet<Integer> connection\_points = new HashSet<>(); /\*\* Содержит точки сочленения \*\*/

private ArrayList<Brigde> bridges = new ArrayList<Brigde>(); /\*\* Содержит мосты \*\*/

private HashMap<Pair\_Of\_Components, Double> distances\_between\_components = new HashMap<Pair\_Of\_Components, Double>(); /\*\* Содержит расстояние между компонентами связности \*\*/ private ArrayList<HashMap<Integer,

Time\_Of\_Existence\_And\_Absence\_Of\_A\_Routes>> times; /\*\* Хранит время жизни маршрутов и время отсутствия маршрутов из этого узла в другие. Формат: Ключ - номер узла, значение: объект вспомогательного класса с полями: time\_exist\_of\_the\_route – время существования маршрута и time\_not\_exist\_of\_the\_route - время отсутствия маршрута \*\*/ int connectivity\_components\_average\_value; int connection\_points\_average\_value; int bridges\_average\_value; double distance\_between\_components\_average\_value; double time\_exist\_of\_the\_route\_average\_value; double time\_not\_exist\_of\_the\_route\_average\_value;

private int Random(int min, int max)

{

max -= min;

return ((int) (Math.random() \* ++max) + min);

}

public Network(int border\_network, int quantity\_nodes, int border\_min\_speed, int border\_max\_speed, int border\_max\_pause, int border\_min\_radius, int border\_max\_radius)

{

this.border\_network = border\_network; this.quantity\_nodes = quantity\_nodes; this.border\_min\_speed = border\_min\_speed; this.border\_max\_speed = border\_max\_speed; this.border\_max\_pause = border\_max\_pause; this.border\_min\_radius = border\_min\_radius; this.border\_max\_radius = border\_max\_radius; is\_connected = is\_oriented = false; connectivity\_components\_average\_value = 0; connection\_points\_average\_value = 0; bridges\_average\_value = 0; distance\_between\_components\_average\_value = 0;

/\*time\_exist\_of\_the\_route\_average\_value = 0; time\_not\_exist\_of\_the\_route\_average\_value = 0;\*/

Initialize\_Matrix\_Of\_Links();

//Initialize\_Times\_Of\_Existence\_And\_Absence\_Of\_A\_Routes(); Network\_Start();

}

/\*\* Функция инициализирует матрицу связей \*\*/

private void Initialize\_Matrix\_Of\_Links()

{

matrix\_of\_links = new ArrayList<ArrayList<String>>(); for (int i = 0; i < quantity\_nodes; i++)

{

matrix\_of\_links.add(i, new ArrayList<String>());

for (int j = 0; j < quantity\_nodes; j++) matrix\_of\_links.get(i).add(j, "-");

}

}

/\*\* Функция инициализирует список словарей, который содержит времена существования и отсутствия маршрутов из каждого узла в каждый \*\*/ private void Initialize\_Times\_Of\_Existence\_And\_Absence\_Of\_A\_Routes()

{

times = new ArrayList<HashMap<Integer,

Time\_Of\_Existence\_And\_Absence\_Of\_A\_Routes>>();

for (int i = 0; i < quantity\_nodes; ++i)

{

HashMap<Integer, Time\_Of\_Existence\_And\_Absence\_Of\_A\_Routes> times\_of\_existence\_and\_absence\_of\_a\_routes = new HashMap<Integer,

Time\_Of\_Existence\_And\_Absence\_Of\_A\_Routes>();

for (int j = 0; j < quantity\_nodes; ++j)

{ if (i == j) continue;

Time\_Of\_Existence\_And\_Absence\_Of\_A\_Routes time\_of\_existence\_and\_absence\_of\_a\_routes = new Time\_Of\_Existence\_And\_Absence\_Of\_A\_Routes();

times\_of\_existence\_and\_absence\_of\_a\_routes.put(j,

time\_of\_existence\_and\_absence\_of\_a\_routes);

}

times.add(times\_of\_existence\_and\_absence\_of\_a\_routes);

}

}

/\*\* Функция "запускает" сеть - создаются, инициализируются начальными значениями и активируются узлы \*\*/ private void Network\_Start()

{

for (int i = 0; i < quantity\_nodes; i++)

{

Node node = new Node(Random(border\_min\_speed, border\_max\_speed),

Random(border\_min\_radius, border\_max\_radius));

node.Set\_Coordinates(Random(0, border\_network), Random(0, border\_network));

node.Assign\_Pause(Random(0, border\_max\_pause)); nodes.add(node);

}

}

/\*\* Функция осуществляет управление поданным узлом \*\*/ private void Control\_Node(Node node)

{

if (!node.In\_Move() && !node.Paused()) /\*\* Если узел не движется и не находится в состоянии паузы \*\*/

{

node.Assign\_Pause(Random(0, border\_max\_pause));

}

if (!node.In\_Move() && node.Paused() && !node.Time\_Pause\_Is\_Over())

/\*\* Если узел не движется и находится в состоянии паузы \*\*/

{

node.Pause();

}

if (!node.In\_Move() && node.Paused() && node.Time\_Pause\_Is\_Over()) /\*\*

Если узел не движется, находится в состоянии паузы и готов к отправке(пауза истекла) \*\*/

{

Activate\_Node(node);

}

if (node.In\_Move()) /\*\* Узел находится на пути к цели \*\*/

{

node.Move();

}

}

/\*\* Функция активизации узла - задается случайная скорость, направление, пункт назначения и узел начинает движение \*\*/ private void Activate\_Node(Node node)

{

node.Set\_Speed(Random(border\_min\_speed, border\_max\_speed));

Destination\_Point destination\_point = new Destination\_Point(border\_network);

node.Calculating\_Parameters\_Of\_Movement(destination\_point);

}

/\*\* Функция проверяет, есть ли соединение между узлами \*\*/ private Answer Check\_The\_Existence\_Of\_A\_Link(double x1, double y1, double radius1, double x2, double y2, double radius2)

{

double distance = Math.sqrt(Math.pow(x1 - x2, 2) + Math.pow(y1 - y2, 2));

if (radius1 > distance && radius2 > distance)

{

Answer answer = new Answer("connection two-way"); return answer;

} else

{

if (radius1 > distance)

{

Answer answer = new Answer("connection one-way", 1); return answer;

}

if (radius2 > distance)

{

Answer answer = new Answer("connection one-way", 2); return answer;

}

}

Answer answer = new Answer("no connection"); return answer;

}

/\*\* Функция генерирует матрицу связей \*\*/ private void Generate\_Matrix\_Of\_Link() {

int index\_i = 0; for (Node node: nodes)

{

ArrayList<Integer> list = new ArrayList<>(); int index\_j = 0; for (Node unit : nodes)

{

if (index\_j == index\_i)

{

index\_j++; continue;

}

if (Check\_The\_Existence\_Of\_A\_Link(node.Get\_X(), node.Get\_Y(), node.Get\_Radius(), unit.Get\_X(), unit.Get\_Y(),

unit.Get\_Radius()).Get\_Connection\_Type().equals("no connection")) matrix\_of\_links.get(index\_i).set(index\_j, "-"); else

if (Check\_The\_Existence\_Of\_A\_Link(node.Get\_X(), node.Get\_Y(), node.Get\_Radius(), unit.Get\_X(), unit.Get\_Y(),

unit.Get\_Radius()).Get\_Connection\_Type().equals("connection two-way"))

{

matrix\_of\_links.get(index\_i).set(index\_j, "tw"); list.add(index\_j);

} else

if (Check\_The\_Existence\_Of\_A\_Link(node.Get\_X(), node.Get\_Y(), node.Get\_Radius(), unit.Get\_X(), unit.Get\_Y(),

unit.Get\_Radius()).Get\_Connection\_Type().equals("connection one-way"))

{

matrix\_of\_links.get(index\_i).set(index\_j, "ow");

if (Check\_The\_Existence\_Of\_A\_Link(node.Get\_X(), node.Get\_Y(), node.Get\_Radius(), unit.Get\_X(), unit.Get\_Y(), unit.Get\_Radius()).Get\_Index\_Of\_Vertex() == 1)

list.add(index\_j);

} index\_j++;

}

list\_adjacency.put(index\_i, list); index\_i++;

}

}

/\*\* Функция выводит информацию о указанном узле в консоль \*\*/ public void Show\_Info\_About\_Current\_Node(int index)

{

System.out.println("Узел " + index + ":");

System.out.println("Координаты: x = " + nodes.get(index).Get\_X() + ", y = " + nodes.get(index).Get\_Y());

System.out.println("Скорость узла: " + nodes.get(index).Get\_Speed());

System.out.println("Текущее значение паузы узла: " + nodes.get(index).Get\_Pause());

System.out.println("Радиус узла: " + nodes.get(index).Get\_Radius());

System.out.print("Состояние на данный момент времени: ");

if (!nodes.get(index).In\_Move() && !nodes.get(index).Paused()) /\*\* Если узел не движется и не находится в состоянии паузы \*\*/

{

System.out.println("Узел достиг цели и ожидает значение паузы");

}

if (!nodes.get(index).In\_Move() && nodes.get(index).Paused() && !nodes.get(index).Time\_Pause\_Is\_Over()) /\*\* Если узел не движется и находится в состоянии паузы \*\*/

{

System.out.println("Узел находится в состоянии паузы");

}

if (!nodes.get(index).In\_Move() && nodes.get(index).Paused() && nodes.get(index).Time\_Pause\_Is\_Over()) /\*\* Если узел не движется, находится в состоянии паузы и готов к отправке(пауза истекла) \*\*/

{

System.out.println("Узел готов к отправке к новой цели");

}

if (nodes.get(index).In\_Move()) /\*\* Узел находится на пути к цели \*\*/

{

System.out.println("Узел движется к цели");

}

System.out.println();

}

/\*\* Функция выводит матрицу связей в консоль \*\*/ public void Show\_Matrix\_Of\_Link() {

for (int i = 0; i < matrix\_of\_links.size(); i++)

{

for (int j = 0; j < matrix\_of\_links.size(); j++)

System.out.print(matrix\_of\_links.get(i).get(j) + " ");

System.out.println();

}

System.out.println();

}

/\*\* Фукнция выводит в консоль информацию о путях от каждого узла к остальным \*\*/

private void Show\_Paths()

{

System.out.println("Пути: "); for (int i = 0; i < quantity\_nodes; i++)

{

for (int j = 1 + i; j < quantity\_nodes; j++)

{

System.out.print("Длина пути из узла " + i + " в узел " + j + ": " +

nodes.get(i).Get\_Ways().get(j).Get\_Way\_Length());

if (nodes.get(i).Get\_Ways().get(j).Get\_Way().size() > 1)

{

System.out.print(", Путь: ");

for (int k = 0; k < +nodes.get(i).Get\_Ways().get(j).Get\_Way().size(); k++)

if (k + 1 != nodes.get(i).Get\_Ways().get(j).Get\_Way().size())

System.out.print(nodes.get(i).Get\_Ways().get(j).Get\_Way().get(k) + " -> "); else

System.out.print(nodes.get(i).Get\_Ways().get(j).Get\_Way().get(k));

}

System.out.println();

}

System.out.println();

}

System.out.println();

}

/\*\* Функция выводит список смежности в консоль \*\*/

private void Show\_List\_Adjacency()

{

for (Integer i = 0; i < quantity\_nodes; i++)

{

System.out.print("Узел " + i + " имеет связь с узлами: ");

for (int j = 0; j < list\_adjacency.get(i).size(); j++) System.out.print(list\_adjacency.get(i).get(j) + " ");

System.out.println();

}

System.out.println();

}

/\*\* Функция выводит список времен существования и отсутствия маршрутов между узлами в консоль \*\*/

private void Show\_Times\_Exist\_And\_Absence\_Of\_A\_Paths()

{

for (int i = 0; i < times.size(); ++i) for (Map.Entry<Integer,

Time\_Of\_Existence\_And\_Absence\_Of\_A\_Routes> entry : times.get(i).entrySet())

{

System.out.println("путь из " + i + " в " + entry.getKey() + " существовал: " + entry.getValue().Get\_Time\_Exist\_Of\_The\_Route()); System.out.println("путь из " + i + " в " + entry.getKey() + " не существовал: " + entry.getValue().Get\_Time\_Not\_Exist\_Of\_The\_Route());

}

System.out.println();

}

/\*\* Функция выводит список точек сочленения в консоль \*\*/ private void Show\_Connection\_Point()

{

for (Integer i: connection\_points)

System.out.println(i);

}

/\*\* Функция выводит список мостов в консоль \*\*/

private void Show\_Bridges()

{

for (int i = 0; i < bridges.size(); i++)

System.out.println(bridges.get(i).Get\_Vertex\_1() + " - " + bridges.get(i).Get\_Vertex\_2());

}

/\*\* Функция определяет ориентированный ли у нас геометрический граф или нет \*\*/

private void

Define\_The\_Type\_Of\_Graph\_By\_The\_Orientation\_Characteristic()

{

is\_oriented = false; for (int i = 0; i < matrix\_of\_links.size(); ++i) for (int j = 0; j < matrix\_of\_links.size(); ++j) if (matrix\_of\_links.get(i).get(j).equals("ow"))

{

is\_oriented = true;

break;

}

}

/\*\* Функция определяет связный ли у нас геометрический граф или нет \*\*/ private void

Define\_The\_Type\_Of\_Graph\_By\_The\_Connectivity\_Characteristic()

{

if (connectivity\_components.size() > 1)

is\_connected = false; else

is\_connected = true;

}

/\*\* Функция обновляет информацию о путях из каждого узла в остальные

\*\*/

private void Update\_Paths(int i, int j)

{

for (int i = 0; i < quantity\_nodes; ++i)

for (int j = 0; j < quantity\_nodes; ++j)

{ if (j == i) continue;

if (Way\_Is\_Exist(i, j))

times.get(i).get(j).Increase\_The\_Time\_Exist\_Of\_The\_Route(); else

times.get(i).get(j).Increase\_The\_Time\_Not\_Exist\_Of\_The\_Route();

}

}

/\*\* Функция-обертка для функции поиска в глубину (DFS) \*\*/ private boolean Way\_Is\_Exist(int v1, int v2)

{

BigDecimal way\_length = new BigDecimal(0.0);

ArrayList<Integer> list = new ArrayList<Integer>();

Way way = new Way(way\_length, list); nodes.get(v1).Get\_Ways().put(v2, way);

boolean[] used = new boolean[matrix\_of\_links.size()]; if (DFS(v1, v2, used, way))

return true; return false;

}

/\*\* Функция возвращает длину пути(ребра) между узлами(вершинами) \*\*/ private double Find\_Distance\_Between\_Nodes(int index\_node\_1, int index\_node\_2)

{

double difference\_between\_coordinates\_x =

nodes.get(index\_node\_1).Get\_X() - nodes.get(index\_node\_2).Get\_X(); /\*\* Катет(Ось абсцисс) \*\*/

double difference\_between\_coordinates\_y =

nodes.get(index\_node\_1).Get\_Y() - nodes.get(index\_node\_2).Get\_Y(); /\*\* Катет(Ось ординат) \*\*/

double distance = Math.sqrt(Math.pow(difference\_between\_coordinates\_x, 2)

+ Math.pow(difference\_between\_coordinates\_y, 2)); return distance;

}

/\*\* Функция проверяет наличие пути из вершины v1 в вершину v2 \*\*/ private boolean DFS(int v1, int v2, boolean[] used, Way way)

{

if (v1 == v2)

return true;

used[v1] = true;

for (int i = 0; i < matrix\_of\_links.size(); ++i)

{

if (!matrix\_of\_links.get(v1).get(i).equals("-") && !used[i])

{

BigDecimal distance = new

BigDecimal(Find\_Distance\_Between\_Nodes(v1, i)); way.Increase\_Way\_Length(distance);

way.Get\_Way().add(i);

if (DFS(i, v2, used, way)) return true; way.Decrease\_Way\_Length(distance); for (int j = 0; j < way.Get\_Way().size(); ++j) if (way.Get\_Way().get(j) == i)

{

way.Get\_Way().remove(j); break;

}

} }

return false;

}

/\*\* Функция инициализирует версию матрицы связей для алгоритма поиска точек сочленения и кандидатов в мосты ориентированного геометрического графа сети \*\*/ private void

Initialize\_Version\_Matrix\_Of\_Links\_For\_Algorithm\_Finding\_Connection\_Points \_And\_Bridges()

{

version\_matrix\_of\_links\_for\_algorithm\_finding\_connection\_points\_and\_bridges = new ArrayList<ArrayList<String>>();

for (int i = 0; i < quantity\_nodes; i++)

{

version\_matrix\_of\_links\_for\_algorithm\_finding\_connection\_points\_and\_bridges.a dd(i, new ArrayList<String>());

for (int j = 0; j < quantity\_nodes; j++)

version\_matrix\_of\_links\_for\_algorithm\_finding\_connection\_points\_and\_bridges.g et(i).add(j, "-");

}

for (int i = 0; i < matrix\_of\_links.size(); ++i) for (int j = 0; j < matrix\_of\_links.size(); ++j)

{

if (matrix\_of\_links.get(i).get(j).equals("ow"))

version\_matrix\_of\_links\_for\_algorithm\_finding\_connection\_points\_and\_bridges.g et(i).set(j, "-"); else version\_matrix\_of\_links\_for\_algorithm\_finding\_connection\_points\_and\_bridges.g et(i).set(j, matrix\_of\_links.get(i).get(j));

}

}

/\*\* Функция-обертка для модифицированной функции поиска в глубину

(DFS\_Modified) \*\*/

private void Find\_Connection\_Point()

{

if (is\_oriented)

Initialize\_Version\_Matrix\_Of\_Links\_For\_Algorithm\_Finding\_Connection\_Points

\_And\_Bridges();

else

version\_matrix\_of\_links\_for\_algorithm\_finding\_connection\_points\_and\_bridges = matrix\_of\_links;

connection\_points.clear(); bridges.clear();

int[] number = new int[quantity\_nodes]; /\*\* Массив, хранящий время обнаружения вершин \*\*/

int[] low = new int[quantity\_nodes]; /\*\* Минимум из значений d[x] и значений d[y] среди всех обратных ребер (z, y) \*\*/ int counter = 1;

for (int x = 0; x < quantity\_nodes; x++)

if (number[x] == 0)

DFS\_Modified(x, -1, number, low, counter);

}

/\*\* Функция ищет точки сочленения и кандидатов в мосты в геометрическом графе сети \*\*/

private void DFS\_Modified(int x, int parent, int[] number, int[] low, int counter)

{

int kids = 0;

number[x] = low[x] = counter++; for (int i = 0; i <

version\_matrix\_of\_links\_for\_algorithm\_finding\_connection\_points\_and\_bridges.s ize(); i++)

if

(!version\_matrix\_of\_links\_for\_algorithm\_finding\_connection\_points\_and\_bridges.

get(i).get(x).equals("-"))

{

if (i != parent)

{

if (number[i] == 0)

{

kids++;

DFS\_Modified(i, x, number, low, counter);

if (low[i] == number[i])

{

Brigde brigde = new Brigde(x, i); /\*\* ребро (x,i) - предполагаемый мост \*\*/

bridges.add(brigde);

}

/\*\*===================\*\*/

if (low[i] >= number[x])

connection\_points.add(x); /\*\* вершина x - точка сочленения

\*\*/

/\*\*===================\*\*/

low[x] = Math.min(low[x], low[i]);

} else {

/\*\* обратное ребро \*\*/

low[x] = Math.min(low[x], number[i]);

}

}

}

if (parent == -1 && kids == 1) /\*\* проверка для корня \*\*/

connection\_points.remove(x); /\*\* вершина x - точка сочленения \*\*/

}

/\*\* Функция определяет степень вершины \*\*/ private int Degree\_Of\_Vertex(int vertex)

{

int degree = 0; for (int i = 0; i <

version\_matrix\_of\_links\_for\_algorithm\_finding\_connection\_points\_and\_bridges.s ize(); i++)

if

(!version\_matrix\_of\_links\_for\_algorithm\_finding\_connection\_points\_and\_bridges.

get(i).get(vertex).equals("-")) degree++;

return degree;

}

/\*\* Функция ищет мосты \*\*/ private void Find\_Bridges()

{

for (int i = 0; i < bridges.size(); i++)

{

/\*\* Проверка на степень вершин, точкой сочленения могут быть только не висячие вершины. А у моста оба его конца должны являться точками сочленения \*\*/

if ((Degree\_Of\_Vertex(bridges.get(i).Get\_Vertex\_1()) <= 1) ||

(Degree\_Of\_Vertex(bridges.get(i).Get\_Vertex\_2()) <= 1)) bridges.remove(i);

}

}

/\*\* Функция-обертка для функции dfs \*\*/ private void Find\_Connectivity\_Components()

{

boolean[] used = new boolean[quantity\_nodes];

ArrayList<Integer> component = new ArrayList<Integer>();

HashMap<Integer, ArrayList<Integer>> result = new HashMap<Integer,

ArrayList<Integer>>();

for (int i = 0; i < quantity\_nodes; ++i) used[i] = false;

for (int i = 0; i < quantity\_nodes; ++i)

if (!used[i])

{

component = new ArrayList<Integer>();

dfs(i, used, component); if (component.size() > 0)

result.put(result.size() + 1, component);

}

connectivity\_components = result;

Show\_Results(result);

}

/\*\* Функция ищет компоненты связанности \*\*/ private void dfs(int v, boolean[] used, ArrayList<Integer> comp)

{

used[v] = true; comp.add(v); for (int i = 0; i < list\_adjacency.get(v).size(); ++i)

{

int to = list\_adjacency.get(v).get(i);

if (!used[to])

dfs(to, used, comp);

}

}

/\*\* Функция-обертка для алгоритма Тарьяна \*\*/ private void Find\_Strongly\_Connectivity\_Components()

{

ArrayDeque<Integer> stack = new ArrayDeque<>(); /\*\* Стек,в который записываются в прямом порядке обхода все вершины, а при определении компоненты она выталкивается из стека \*\*/

Integer quantity\_vertexes = list\_adjacency.size(); /\*\* Количество вершин

\*\*/

Integer lowlinks[] = new Integer[quantity\_vertexes]; /\*\* Наименьшая вершина прямого порядка, которая доступна из некоторой вершины v \*\*/

Integer indices[] = new Integer[quantity\_vertexes]; /\*\* Сюда записываются вершины графа в прямом порядке поиска в глубину \*\*/

Integer id = 0; /\*\* Определяет, какая по счету текущяя вершина в прямом порядке \*\*/

Arrays.fill(indices,0);

Arrays.fill(lowlinks, quantity\_vertexes);

HashMap<Integer, ArrayList<Integer>> result = new HashMap<Integer,

ArrayList<Integer>>();

Boolean used[] = new Boolean[quantity\_vertexes];

Arrays.fill(used, false);

for (Integer i = 0; i < quantity\_vertexes; i++) if (!used[i])

Tarjan\_Algorithm(i, used, result, id, lowlinks, indices, stack); connectivity\_components = result;

Show\_Results(result);

}

/\*\* Поиск в глубину, приспособленный под нахождение сильных компонент связности \*\*/ private void Tarjan\_Algorithm(Integer v, Boolean used[], HashMap<Integer,

ArrayList<Integer>> result, Integer id, Integer lowlinks[], Integer indices[],

ArrayDeque<Integer> stack)

{

used[v] = true;

lowlinks[v] = id;

indices[v] = id++; stack.push(v);

for (Integer val : list\_adjacency.get(v))

{

if (!used[val]) {

Tarjan\_Algorithm(val, used, result, id, lowlinks, indices, stack); lowlinks[v] = Integer.min(lowlinks[v], lowlinks[val]);

} else

if (stack.contains(val))

lowlinks[v] = Integer.min(lowlinks[v], indices[val]);

}

ArrayList<Integer> result\_set = new ArrayList<>();

if (indices[v] == lowlinks[v])

{

while (stack.peek() != v) result\_set.add(stack.pop());

result\_set.add(stack.pop());

result.put(result.size() + 1, result\_set);

}

}

/\*\* Функция выводит в консоль компоненты связности \*\*/ private void Show\_Results(HashMap<Integer, ArrayList<Integer>> map)

{

for (Map.Entry<Integer, ArrayList<Integer>> entry : map.entrySet())

{

System.out.print(entry.getKey() + " компонента связности: ");

String s = "";

for (Integer i : entry.getValue())

{

s += i + ", ";

}

System.out.println(s.substring(0, s.lastIndexOf(',')));

}

System.out.println();

}

/\*\* Функция находит минимальное расстояние между двумя компонентами

\*\*/

private double Find\_Minimum\_Distance(ArrayList<Integer> component\_i, ArrayList<Integer> component\_j)

{

Double minimum\_value = Double.MAX\_VALUE;

for (int i = 0; i < component\_i.size(); ++i)

for (int j = 0; j < component\_j.size(); ++j)

{

double difference\_between\_coordinates\_x =

nodes.get(component\_i.get(i)).Get\_X() - nodes.get(component\_j.get(j)).Get\_X(); double difference\_between\_coordinates\_y =

nodes.get(component\_i.get(i)).Get\_Y() - nodes.get(component\_j.get(j)).Get\_Y(); double distance =

Math.sqrt(Math.pow(difference\_between\_coordinates\_x, 2) +

Math.pow(difference\_between\_coordinates\_y, 2));

if (distance < minimum\_value)

minimum\_value = distance;

}

return minimum\_value;

}

/\*\* Функция ищет расстояния между всеми компонентами связности \*\*/ private void Find\_Distance\_Between\_Components()

{

distances\_between\_components.clear(); for (int i = 1; i <= connectivity\_components.size(); ++i) for (int j = i + 1; j <= connectivity\_components.size(); ++j)

{

Pair\_Of\_Components pair\_of\_components = new Pair\_Of\_Components(connectivity\_components.get(i), connectivity\_components.get(j));

distances\_between\_components.put(pair\_of\_components, Find\_Minimum\_Distance(connectivity\_components.get(i), connectivity\_components.get(j)));

System.out.println("Расстояние между " + i + " и " + j + " = " +

Find\_Minimum\_Distance(connectivity\_components.get(i),

connectivity\_components.get(j)));

}

}

private double

Calculation\_Average\_Distance\_Between\_Connectivity\_Components()

{ double sum = 0; for (Map.Entry<Pair\_Of\_Components, Double> entry : distances\_between\_components.entrySet())

sum += entry.getValue();

return (sum / distances\_between\_components.size());

}

/\*\* Работа сети \*\*/

private void Network\_Work(int time, int interval)

{

System.out.println("Скорость: " + border\_min\_speed + " - " + border\_max\_speed);

System.out.println("Пауза: " + border\_max\_pause);

for (int t = 0; t < time; t++) {

System.out.println("Тик " + t);

int index = 0;

for (Node node : nodes)

{

Show\_Info\_About\_Current\_Node(index);

Control\_Node(node);

index++;

}

Generate\_Matrix\_Of\_Link();

Show\_Matrix\_Of\_Link();

Show\_List\_Adjacency();

Update\_Paths();

Show\_Paths();

if (t % interval == 0) {

Define\_The\_Type\_Of\_Graph\_By\_The\_Orientation\_Characteristic();

if (is\_oriented)

{

Find\_Strongly\_Connectivity\_Components(); connectivity\_components\_average\_value += connectivity\_components.size();

} else

{

Find\_Connectivity\_Components(); connectivity\_components\_average\_value +=

connectivity\_components.size();

}

Define\_The\_Type\_Of\_Graph\_By\_The\_Connectivity\_Characteristic();

Find\_Connection\_Point();

Find\_Bridges();

Show\_Connection\_Point();

Show\_Bridges();

connection\_points\_average\_value += connection\_points.size(); bridges\_average\_value += bridges.size();

if (!is\_connected)

{

Find\_Distance\_Between\_Components(); distance\_between\_components\_average\_value +=

Calculation\_Average\_Distance\_Between\_Connectivity\_Components();

}

for (Map.Entry<Integer,

Time\_Of\_Existence\_And\_Absence\_Of\_A\_Routes> entry : times.get(0).entrySet())

{

time\_exist\_of\_the\_route\_average\_value += entry.getValue().Get\_Time\_Exist\_Of\_The\_Route(); time\_not\_exist\_of\_the\_route\_average\_value +=

entry.getValue().Get\_Time\_Not\_Exist\_Of\_The\_Route();

}

for (Node node : nodes)

{

for (Map.Entry<Integer, Way> entry : node.Get\_Ways().entrySet()) length\_way\_average\_value +=

entry.getValue().Get\_Way\_Length().doubleValue(); break;

}

Show\_Times\_Exist\_And\_Absence\_Of\_A\_Paths();

}

}

}

public static void main(String[] args)

{

System.out.println("Введите время моделирования:");

Scanner scanner = new Scanner(System.in);

int time = scanner.nextInt();

System.out.println("Введите временной интервал:"); int interval = scanner.nextInt();

int border\_network = 500; int quantity\_nodes = 100; int border\_min\_speed = 0; int border\_max\_speed = 25; int border\_max\_pause = 0; int border\_min\_radius = 25; int border\_max\_radius = 50;

int border\_test = 10000;

int border\_node = 100; int border\_speed = 250;

int border\_pause = 55;

int step\_node = 10; int step\_speed\_node = 25; int step\_pause\_node = 5;

int q = 0;

/\*for (int n = quantity\_nodes; n < border\_node; n += step\_node)

{

if (q > 9)

{

quantity\_nodes += step\_node;

}

border\_min\_speed = 0;

border\_max\_speed = 25;\*/

for (int speed = border\_max\_speed; speed < border\_speed; speed += step\_speed\_node)

{

String file = "Data" + q + ".csv";

try (BufferedWriter writer = new BufferedWriter(new FileWriter(file))) {

border\_min\_speed = border\_max\_speed;

border\_max\_speed += step\_speed\_node;

writer.write("Border Network");

writer.write(";"); writer.write("Time"); writer.write(";");

writer.write("Border min speed"); writer.write(";");

writer.write("Border max speed"); writer.write(";");

writer.write("Min Border Radius"); writer.write(";");

writer.write("Max Border Radius"); writer.write(";");

writer.write("Border Max Pause"); writer.newLine();

writer.write("" + border\_network);

writer.write(";"); writer.write("" + time); writer.write(";");

writer.write("" + border\_max\_speed); writer.write(";");

writer.write("" + border\_min\_radius); writer.write(";");

writer.write("" + border\_max\_radius);

writer.write(";"); writer.write("" + 5); writer.write(";"); writer.write("" + border\_pause); writer.newLine(); writer.write("Quantity nodes"); writer.write(";");

writer.write("Quantity component");

writer.write(";");

writer.write("Quantity connection points");

writer.write(";"); writer.write("Quantity bridges");

writer.write(";");

writer.write("Distance Between Component"); writer.write(";");

border\_max\_pause = 0;

writer.write("way between pair nodes exist "); writer.write(";");

writer.write("way between pair nodes not exist "); writer.newLine();

writer.write("length way between pair nodes "); writer.newLine(); writer.newLine();

for (int pause = border\_max\_pause; pause < border\_pause; pause += step\_pause\_node)

{

border\_max\_pause += step\_pause\_node;

double connectivity\_components\_average\_value = 0; double connection\_points\_average\_value = 0; double bridges\_average\_value = 0;

double distance\_between\_components\_average\_value = 0;

for (int quantity\_tests = 0; quantity\_tests < border\_test; quantity\_tests++)

{

System.out.println("испытание " + quantity\_tests);

Network network = new Network(border\_network, quantity\_nodes, border\_min\_speed, border\_max\_speed, border\_max\_pause, border\_min\_radius, border\_max\_radius);

network.Network\_Work(time, interval);

connectivity\_components\_average\_value += network.connectivity\_components\_average\_value / (time / interval);

connection\_points\_average\_value += network.connection\_points\_average\_value / (time / interval);

bridges\_average\_value += network.bridges\_average\_value /

(time / interval);

distance\_between\_components\_average\_value +=

network.distance\_between\_components\_average\_value;

time\_exist\_of\_the\_route\_average\_value +=

network.time\_exist\_of\_the\_route\_average\_value / (time / interval); time\_not\_exist\_of\_the\_route\_average\_value += network.time\_not\_exist\_of\_the\_route\_average\_value / (time / interval);

length\_way\_average\_value +=

network.length\_way\_average\_value;

if (quantity\_tests == border\_test - 1)

{

/\*writer.write("" + quantity\_nodes); writer.write(";");\*/

writer.write("" + (connectivity\_components\_average\_value /

(border\_test/\* / 10\*/)));

writer.write(";");

writer.write("" + (connection\_points\_average\_value /

(border\_test/\* / 10\*/)));

writer.write(";");

writer.write("" + (bridges\_average\_value / (border\_test/\* /

10\*/)));

writer.write(";");

writer.write("" +

(distance\_between\_components\_average\_value / (border\_test/\* / 10\*/))); writer.write(";");

writer.write("" + quantity\_nodes); writer.write(";");

writer.write("" + (time\_exist\_of\_the\_route\_average\_value /

(border\_test/\* / 10\*/)));

writer.write(";");

writer.write("" + (time\_not\_exist\_of\_the\_route\_average\_value

/ (border\_test/\* / 10\*/)));

writer.write(";");

writer.write("" + (length\_way\_average\_value / (border\_test/\* /

10\*/)));

writer.write(";"); writer.newLine();

}

}

}

writer.close();

}

catch (IOException exception)

{

exception.printStackTrace();

} q++;

}

//}

}

}

import java.util.HashMap;

/\*\*

\* Created by Vitalik on 28.02.2017.

\*/

public class Node

{

private double x; /\*\* Координата узла на плоскости по оси абсцисс \*\*/ private double y; /\*\* Координата узла на плоскости по оси ординат \*\*/ private int speed; /\*\* Скорость движения узла \*\*/ private int pause; /\*\* Время паузы узла \*\*/ private double radius; /\*\* Радиус узла \*\*/ private boolean in\_move; /\*\* Находится ли узел в движении \*\*/ private boolean paused; /\*\* Находится ли узел в состоянии паузы \*\*/ private boolean time\_pause\_is\_over; /\*\* Готов ли узел к новой

отправке(Истекло ли время паузы) \*\*/

private Destination\_Point destination\_point; /\*\* Пункт назначения(Цель, куда движется узел) \*\*/

private double distance; /\*\* Расстояние от узла до пункта назначения(Гипотенуза) \*\*/

private double distance\_traveled; /\*\* Уже пройденное узлом расстояние на данный момент времени \*\*/

private double count\_steps; /\*\* Количество тиков времени(шагов), за которое узел достигнет пункта назначения \*\*/

private double offset\_x; /\*\* Смещение в единицу времени(тик) для абсциссы узла \*\*/

private double offset\_y; /\*\* Смещение в единицу времени(тик) для ординаты узла \*\*/

private HashMap<Integer, Way> ways; /\*\* Хранит узлы и пути от этого узла

к ним(если такие существуют). Формат: Ключ - номер узла, к которому пролегает путь, значение: объект вспомогательного класса с полями:

way\_length - длина пути и way - сам путь(порядок вершин) \*\*/

public Node(int speed, double radius)

{

this.speed = speed;

this.radius = radius;

in\_move = paused = time\_pause\_is\_over = false;

offset\_x = offset\_y = 0;

ways = new HashMap<Integer, Way>();

}

/\*\* Функция задает координаты узла на плоскости \*\*/ public void Set\_Coordinates(double x, double y)

{ this.x = x; this.y = y;

}

/\*\* Функция задает скорость узла \*\*/ public void Set\_Speed(int speed)

{

this.speed = speed;

}

/\*\* Функция возвращает координату узла на плоскости по оси абсцисс \*\*/ public double Get\_X()

{

return x;

}

/\*\* Функция возвращает координату узла на плоскости по оси ординат \*\*/ public double Get\_Y()

{

return y;

}

/\*\* Функция возвращает значение скорости узла \*\*/ public int Get\_Speed()

{

return speed;

}

/\*\* Функция возвращает значение радиуса узла \*\*/

public double Get\_Radius()

{

return radius;

}

/\*\* Функция возвращает текущее значение паузы узла \*\*/ public int Get\_Pause()

{

return pause;

}

/\*\* Функция возвращает словарь, хранящий пути от этого узла к другим

\*\*/

public HashMap<Integer, Way> Get\_Ways()

{

return ways;

}

/\*\* Функция задает узлу значение паузы \*\*/

public void Assign\_Pause(int pause)

{

paused = true; speed = 0; this.pause = pause;

}

/\*\* Функция "паузит" узел \*\*/

public void Pause()

{

if (pause != 0) pause--; else

time\_pause\_is\_over = true;

}

/\*\* Функция возвращает значение логической переменной time\_pause\_is\_over \*\*/

public boolean Time\_Pause\_Is\_Over()

{

return time\_pause\_is\_over;

}

/\*\* Функция возвращает значение логической переменной in\_move \*\*/ public boolean In\_Move()

{

return in\_move;

}

/\*\* Функция возвращает значение логической переменной paused \*\*/ public boolean Paused()

{

return paused;

}

/\*\* Функция расчета параметров движения узла \*\*/

public void Calculating\_Parameters\_Of\_Movement(Destination\_Point destination\_point)

{

this.destination\_point = destination\_point;

double difference\_between\_coordinates\_x = destination\_point.Get\_X() - x;

/\*\* Катет(Ось абсцисс) \*\*/

double difference\_between\_coordinates\_y = destination\_point.Get\_Y() - y;

/\*\* Катет(Ось ординат) \*\*/

distance = Math.sqrt(Math.pow(difference\_between\_coordinates\_x, 2) +

Math.pow(difference\_between\_coordinates\_y, 2));

count\_steps = distance / speed; if (count\_steps >= 1)

{

offset\_x = difference\_between\_coordinates\_x / count\_steps; offset\_y = difference\_between\_coordinates\_y / count\_steps;

}

distance\_traveled = 0; in\_move = true;

time\_pause\_is\_over = true;

}

/\*\* Функция движения узла \*\*/

public void Move()

{

if (speed > distance - distance\_traveled)

{

distance\_traveled += distance - distance\_traveled; x = destination\_point.Get\_X();

y = destination\_point.Get\_Y();

} else {

x += offset\_x; y += offset\_y;

distance\_traveled = Math.sqrt(Math.pow(destination\_point.Get\_X() - x, 2) + Math.pow(destination\_point.Get\_Y() - y, 2));

}

if (distance == distance\_traveled) in\_move = false;

}

}

import java.util.Random;

/\*\*

\* Created by Vitalik on 27.04.2017.

\*/

public class Destination\_Point

{ private int x; /\*\* Координата узла на плоскости по оси абсцисс \*\*/ private int y; /\*\* Координата узла на плоскости по оси ординат \*\*/ private Random random = new Random(System.currentTimeMillis());

public Destination\_Point(int border\_network)

{

x = random.nextInt(border\_network); y = random.nextInt(border\_network);

}

/\*\* Функция возвращает координату узла на плоскости по оси абсцисс \*\*/ public int Get\_X()

{

return x;

}

/\*\* Функция возвращает координату узла на плоскости по оси ординат \*\*/ public int Get\_Y()

{ return y; }

}

import java.util.ArrayList;

/\*\*

\* Created by Vitalik on 10.05.2017.

\*/

public class Pair\_Of\_Components

{

private ArrayList<Integer> component\_1 = new ArrayList<Integer>(); /\*\* Первая компонента связности \*\*/

private ArrayList<Integer> component\_2 = new ArrayList<Integer>(); /\*\* Вторая компонента связности \*\*/

public Pair\_Of\_Components(ArrayList<Integer> component\_1,

ArrayList<Integer> component\_2)

{

this.component\_1 = component\_1; this.component\_2 = component\_2;

}

/\*\* Функция возвращает первую компоненту пары \*\*/ public ArrayList<Integer> Get\_Component\_1()

{

return component\_1;

}

/\*\* Функция возвращает вторую компоненту пары \*\*/ public ArrayList<Integer> Get\_Component\_2()

{

return component\_2;

}

}

/\*\*

\* Created by Vitalik on 05.05.2017.

\*/

public class Brigde

{ private int vertex\_1; /\*\* Начало моста \*\*/ private int vertex\_2; /\*\* Конец моста \*\*/

public Brigde(int vertex\_1, int vertex\_2)

{

this.vertex\_1 = vertex\_1; this.vertex\_2 = vertex\_2;

}

/\*\* Функция возвращает первую "опору"(начало) моста \*\*/ public int Get\_Vertex\_1()

{

return vertex\_1;

}

/\*\* Функция возвращает вторую "опору"(конец) моста \*\*/ public int Get\_Vertex\_2()

{

return vertex\_2;

}

}

/\*\*

\* Created by Vitalik on 11.05.2017.

\*/

public class Answer

{

private String connection\_type; /\*\* Строка "Тип соединения" может содержать 3 вида связи между узлами: нет соединения(no connection), соединение двусторонее(connection two-way), соединение односторонее(connection one-way) \*\*/ private int number\_of\_vertex = -1; /\*\* Это поле не пусто только при втором типе соединения(connection one-way) и уточняет вершину, с большим радиусом \*\*/

public Answer(String connection\_type, int ...number\_of\_vertex)

{

this.connection\_type = connection\_type; if (number\_of\_vertex.length != 0)

this.number\_of\_vertex = number\_of\_vertex[0];

}

/\*\* Функция возвращает тип соединения между узлами \*\*/

public String Get\_Connection\_Type()

{

return connection\_type;

}

/\*\* Функция возвращает номер узла для уточнения ответа функции проверки типа связи \*\*/ public int Get\_Index\_Of\_Vertex()

{

return number\_of\_vertex;

}

}

import java.math.BigDecimal;

import java.util.ArrayList;

/\*\*

\* Created by Vitalik on 13.05.2017.

\*/

public class Way

{ private BigDecimal way\_length; /\*\* Длина пути \*\*/

private ArrayList<Integer> way = new ArrayList<Integer>(); /\*\* Содержит последовательность из номеров вершин(узлов) \*\*/

public Way(BigDecimal way\_length, ArrayList<Integer> way)

{

this.way\_length = way\_length;

this.way = way;

}

/\*\* Функция возвращает длину пути \*\*/ public BigDecimal Get\_Way\_Length()

{

return way\_length;

}

/\*\* Функция возвращает список вершин(путь) \*\*/

public ArrayList<Integer> Get\_Way()

{

return way;

}

/\*\* Функция увеличивает значение длины пути на поданную величину \*\*/ public void Increase\_Way\_Length(BigDecimal value)

{

way\_length = way\_length.add(value).setScale(6,

BigDecimal.ROUND\_HALF\_DOWN);

}

/\*\* Функция уменьшает значение длины пути на поданную величину \*\*/ public void Decrease\_Way\_Length(BigDecimal value)

{

way\_length = way\_length.subtract(value).setScale(6,

BigDecimal.ROUND\_HALF\_DOWN);

}

}

/\*\*

\* Created by Vitalik on 02.06.2017.

\*/

public class Time\_Of\_Existence\_And\_Absence\_Of\_A\_Routes

{ private double time\_exist\_of\_the\_route; /\*\* Время жизни маршрута \*\*/ private double time\_not\_exist\_of\_the\_route; /\*\* Время отсутствия маршрута \*\*/

public Time\_Of\_Existence\_And\_Absence\_Of\_A\_Routes()

{

time\_exist\_of\_the\_route = time\_not\_exist\_of\_the\_route = 0;

}

/\*\* Функция возвращает значение времени жизни пути \*\*/ public double Get\_Time\_Exist\_Of\_The\_Route()

{

return time\_exist\_of\_the\_route;

}

/\*\* Функция возвращает значение времени отсутствия пути \*\*/ public double Get\_Time\_Not\_Exist\_Of\_The\_Route()

{

return time\_not\_exist\_of\_the\_route;

}

/\*\* Функция увеличивает значение времени жизни пути \*\*/ public void Increase\_The\_Time\_Exist\_Of\_The\_Route()

{

time\_exist\_of\_the\_route++;

}

/\*\* Функция увеличивает значение времени отсутствия пути \*\*/ public void Increase\_The\_Time\_Not\_Exist\_Of\_The\_Route()

{

time\_not\_exist\_of\_the\_route++;

}

}