МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение

высшего образования

**«КУБАНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»**

**(ФГБОУ ВО «КубГУ»)**

**Факультет компьютерных технологий и прикладной математики**

**Кафедра вычислительных технологий**

Допустить к защите

Заведующий кафедрой

д-р физ.-мат. наук, профессор

\_\_\_\_\_\_\_\_\_\_\_ А.И. Миков

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_2018 г.

**ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА**

**(БАКАЛАВРСКАЯ РАБОТА)**

**ОПТИМИЗАЦИЯ АЛГОРИТМОВ ПРЕДСКАЗАНИЯ ХАРАКТЕРИСТИК ДИНАМИЧЕСКИХ ГРАФОВ КОМПЬЮТЕРНЫХ СЕТЕЙ**

Работу выполнил \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_А.Н. Сеньков

Направление подготовки 02.03.02 Фундаментальная информатика и информационные технологии

Направленность (профиль)Вычислительные технологии

Научный руководитель

д-р физ.-мат. наук, профессор \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_А.И. Миков

Нормоконтролер

канд. техн. наук, ст. преп.\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Е.Е. Полупанова

Краснодар

2018

РЕФЕРАТ

Выпускная квалификационная работа бакалавра 47с., 18 рис., 10 источников.

Объектами исследования являются ad-hoc сети, алгоритмы исследования характеристик динамических графов.

Цель работы – создание алгоритма предсказывающего характеристику динамического графа компьютерной сети и его последующая оптимизация.

В процессе работы была создана модель ad-hoc сети, которая позволяет разрабатывать и проверять на практике алгоритмы предсказывающие характеристики динамического графа.

Была проверена эффективность работы алгоритма и его оптимизации.

СОДЕРЖАНИЕ

[Введение 4](#_Toc483313130)

[1 Общие сведения 5](#_Toc483313131)

[1.1 Ad-hoc система: общие сведения 5](#_Toc483313132)

[1.2 Технологии, используемые для построения ad-hoc сетей 5](#_Toc483313133)

[1.3 Мобильный агент, собирающий информацию о ad-hoc системе. 9](#_Toc483313134)

[1.4 Алгоритмы, предсказывающие характеристики динамических графов компьютерных сетей и их оптимизация……………………………………….9](#_Toc483313135)

[2 Практическая реализация оптимизированного алгоритма предсказывающего характеристики динамических графов компьютерных систем. 10](#_Toc483313136)

[2.1 Реализация мобильного агента 10](#_Toc483313137)

[2.2 Реализация оптимизированного алгоритма предсказывающего характеристики динамических графов компьютерных сетей…………...….28](#_Toc483313138)

[Заключение 3](#_Toc483313139)3

[Список использованных источников 34](#_Toc483313140)

[Приложение Код программы  [3](#_Toc483313142)](#_Toc483313141)5

# ВВЕДЕНИЕ

В современном мире многие люди начинают предпочитать стационарным компьютерам различного рода мобильные устройства, например, телефоны или планшеты. Не в последнюю очередь это происходит из-за все более и более растущей вычислительной мощности этих устройств, что позволяет использовать их не только для отдыха, но и для работы.

При работе с мобильными устройствами, мы часто сталкиваемся с проблемой объединения этих устройств в единую сеть, так как конфигурация сети динамична, то есть будет постоянно изменяться, что означает невозможность использования некого центрального узла (сервера) в связи с тем, что отсутствуют гарантии того, что в случайный момент времени у центрального узла будет доступ ко всем узлам существующей сети.

Поэтому в целях контроля сетей, создаваемых мобильными устройствами, были созданы специальные контрольные устройства – мобильные агенты, выполняющие множество специфический операций с целью определения конфигурации сети и различнейших её характеристик.

Некоторые из функций, выполняемых мобильным агентом являются крайне сложными и их выполнение занимает достаточно много времени. Поэтому так важно оптимизировать работу алгоритмов, предсказывающих характеристики мобильной сети таким образом, чтобы позволить мобильному агенту работать быстрее за счет сокращения вызовов наиболее сложных его функций.

Целью данной выпускной квалификационной работы является разработка алгоритма предсказывающего некоторую характеристику графа и его дальнейшая оптимизация.

# 1 Общие сведения

## 1.1 Ad-hoc система: общие сведения

Ad-hoc система ̶ беспроводная сеть, у которой отсутствует постоянная структура. В ad-hoc системе отсутствует центральный узел как таковой в виду её устройства, Устройства, входящие в сеть, соединяются динамически, образуя собственно сеть. В пересылке данных узлам системы участвует каждый узел сети. Выбор того, кому из узлов передать информацию осуществляется динамически, на основании связности сети. Это служит одним из наиболее значительных отличий между ad-hoc сетью и проводными сетями или управляемыми беспроводными сетями, в которых потоками данных управляют маршрутизаторы или точки доступа в проводных сетях или управляемых беспроводных сетях соответственно [1,2].

Первыми беспроводными самоорганизующимися сетями были сети «packet radio» начиная с 1970-х годов, финансируемые DARPA после проекта ALOHAnet.

## 1.2 Технологии, используемые для построения ad-hoc сетей

Bluetooth — производственная спецификация беспроводных персональных сетей (Wireless personal area network, WPAN). Bluetooth позволяет совершать обмены информацией между различными устройствами: персональными компьютерами, мобильными телефонами и многими другими устройствами. Bluetooth позволяет этим устройствам участвовать в обмене информацией между собой, при условии, что они расположены в радиусе до 10 м друг от друга (дальность установления связи между устройствами зависит от преград и помех на пути у сигнала).

Bluetooth использует радиоволны для обеспечения связи между устройствами. Осуществляется радиосвязь через Bluetooth в ISM-диапазоне (Industry, Science and Medicine), используемый большинством различных бытовых приборов и беспроводных сетей. В Bluetooth применяется метод расширения спектра со скачкообразной перестройкой частоты (Frequency Hopping Spread Spectrum, FHSS) [3]. Метод FHSS прост в реализации, а также крайне устойчив к широкополосным помехам.

В Bluetooth несущая частота сигнала скачкообразно меняется 1600 раз в секунду (всего выделяется 79 рабочих частот шириной в 1 МГц, а в Японии, Франции и Испании полоса уже — 23 частотных канала). Каждое соединение псевдослучайно переключается между частотами и последовательность переключений известна только передатчику и приёмнику, которые каждые 625 мкс (один временной слот) синхронно перестраиваются с одной несущей частоты на другую. Таким образом, несколько одновременно работающих пар приёмник-передатчик никогда не будут мешать друг другу. Этот алгоритм является также составной частью системы защиты конфиденциальности передаваемой информации: переход определяется отдельно для каждого соединения по псевдослучайному алгоритму. При передаче цифровых данных и аудиосигнала (64 кбит/с в обоих направлениях) используются различные схемы кодирования: аудиосигнал не повторяется (как правило), а в случае утери информации данные будут отправлены повторно.

Протокол Bluetooth поддерживает не только соединение «point-to-point», но и соединение «point-to-multipoint» [4].

Wi-Fi — торговая марка Wi-Fi Alliance для беспроводных сетей на базе стандарта IEEE 802.11. В настоящее время разрабатывается семейство Wi-Fi стандартов пересылки цифровых потоков информации по радиоканалам.

Как правило, схема Wi-Fi сети содержит как минимум одну точку доступа и одного клиента. Также возможно подключение двух клиентов в режиме точка-точка (Ad-hoc), когда клиенты соединяются при помощи сетевых адаптеров «напрямую», минуя точку доступа. Точка доступа передаёт свой идентификатор сети (SSID) специальными сигнальными пакетами на скорости 0,1 Мбит/с каждые 100 мс. Поэтому 0,1 Мбит/с — наименьшая скорость передачи данных для Wi-Fi. Для того, чтобы узнать, возможно ли подключение к точке доступа, клиенту требуется знать SSID сети. Если приемнику доступны несколько точек доступа с идентичными SSID, то он, исходя из сведений об уровне сигнала, может выбирать между ними. Стандарт Wi-Fi предлагает клиенту свободно выбирать критерии для соединения.

Однако стандарт не содержит всех особенностей построения беспроводных локальных сетей Wi-Fi. Поэтому каждый производитель оборудования применяет для решения этого вопроса наиболее подходящие для себя способы построения беспроводной локальной сети. Поэтому существует множество вариантов построения локальных беспроводных сетей по стандартам Wi-Fi, которые можно классифицировать.

По способу объединения точек доступа в систему можно выделить:

* автономные точки доступа;
* точки доступа, работающие под управлением контроллера;
* точки доступа, управляемые без контроллера.

По способу организации и управления радиоканалами можно выделить беспроводные локальные сети:

* со статическими настройками радиоканалов;
* с динамическими (адаптивными) настройками радиоканалов;
* со «слоистой» или многослойной структурой радиоканалов.

ZigBee — спецификация сетевых протоколов верхнего уровня — уровня приложений APS (application support sublayer) и сетевого уровня NWK, — использующих сервисы нижних уровней — уровня управления доступом к среде MAC и физического уровня PHY, регламентированных стандартом IEEE 802.15.4. ZigBee и IEEE 802.15.4 описывают беспроводные персональные вычислительные сети (WPAN). Спецификация ZigBee ориентирована на приложения, требующие гарантий безопасности при передаче данных на небольших скоростях и возможности для сетевых устройств, длительно работать от автономных источников питания (батарей).

One-Net — Первый Открытый Протокол беспроводной сети передачи данных, разработанный для автоматизации зданий и управления распределёнными объектами. One-Net может быть использован вместе со множеством приемопередатчиков и микроконтроллеров от самых разных производителей [5].

Wideband Networking Waveform (WNW) — закрытый сетевой протокол радиосвязи в JTRS-радиостанциях для работы в широких диапазонах радиочастот (от 2 МГц до 2 ГГц). Протокол позволяет строить самоорганизующиеся ad-hoc сети, отличающиеся высокой степенью мобильности, которые могут передавать как голос, так и данные с видеоинформацией с применением различных политик безопасности, не зависящих друг от друга. Данный протокол способен обеспечивать возможности маршрутизации и высокоскоростную передачу данных. В настоящее время находится в разработке компанией Boeing в рамках возимых JTRS-радиостанций GMR (Ground Mobile Radio). На физическом уровне применяется модуляция COFDM. Скорость передачи данных на физическом уровне от 47 Kb/s до 12.1 Mb/s. WNW планируется использовать в защищенных опорных сетях тактического звена управления.

## 1.3 Мобильный агент, собирающий информацию о ad-hoc системе.

Сбор информации о динамическом графе компьютерной сети (ad-hoc) теми же методами, что применяются и для сетей (как проводных, так и беспроводных), имеющих центральный узел, невозможен в виду отсутствия центрального узла (сервера) в такого рода сетях, а также того, что каждому конкретному узлу неизвестна вся конфигурация сети. Для этих целей используются мобильные агенты, которые, собирая локальную информацию с каждого узла, формируют глобальную информацию о конфигурации сети.

## 1.4 Алгоритмы, предсказывающие характеристики динамических графов компьютерных сетей и их оптимизация

Некоторые алгоритмы, предназначенные для сбора информации агентом могут производить свою работу в течении довольно продолжительного периода времени. В целях сокращения этого временного промежутка были созданы оптимизированные алгоритмы, которые позволяют значительно упростить работу мобильного агента.

# 2 Практическая реализация оптимизированного алгоритма предсказывающего характеристики динамических графов компьютерных систем

## 2.1 Реализация мобильного агента

Моделью Распределенной ad-hoc системы является геометрический граф.

Геометрическим графом в некотором пространстве называют сочетание множества точек V = {vi} и множества ребер E = {e i}, такое, что:

* 1. каждая петля в E содержит только одну точку v из множества V;
  2. каждое ребро в E содержит ровно две точки множества V, которые являются ее граничными точками;
  3. ребра из E имеют в качестве общих точек только точки из множества V.

Существует два основных способа представления графа в памяти программы: матричный и списковый.

В данной работе было использовано матричное представление графа, в виду того, что его использование облегчает работу с некоторыми алгоритмами, используемыми в программе, а также упрощает доступ к информации о существовании ребра между определенными вершинами: при матричном представлении графа, сложность доступа к ребру составляет всего лишь один, в то время как для спискового представления графа, сложность доступа составит V + E, где V – количество вершин в графе, а E – количество ребер.

Координаты каждого мобильно устройства, входящего в сеть, а также радиусы их передатчиков будем хранить в отдельных массивах: X, Y и R соответственно, где X – координаты устройства по оси абсцисс, а Y – координаты устройства по оси ординат в некотором линейном пространстве, заданном в прямоугольной системе координат, и R – массив, содержащий в себе информацию о радиусах передатчиков используемых в данной сети устройств.

Кроме того, каждому устройству была добавлена такая характеристика, как скорость его перемещения по координатной плоскости. Хранится эта характеристика для всех устройств в специальном массиве Sp;

Также была реализована возможность подсчета расстояния между узлами системы при помощи функции calc\_dist().

В дальнейшем встает проблема того, как реализовать динамическое изменение конфигурации сети при перемещении узлов в пространстве.

При каждом перемещении узлов запускается функция проверки конфигурации на изменения check\_config(), которая действует следующим образом: Производится проверка каждого устройства, входящего в сеть, при помощи функции check\_device() на то, вышло ли какое-либо устройство из зоны действия его передатчика или вышло ли само устройства из зоны действия передатчика какого-либо другого устройства. Если подобное произошло, то функция обновляет данные о текущей конфигурации в памяти агента, путем вызова функций rebuild\_config() и check\_ways(), после чего она завершается. Если изменений не было обнаружено, то функция завершается без вызова каких-либо дополнительных функций.

Первоначально, матрица конфигурации не заполнена, а также неизвестны координаты устройств. Координаты устройств задаются случайным образом, после чего запускается функция rebuild\_config(), которая строит матрицу смежности начальной конфигурации. Затем запускается функция check\_connections() с целью определить является ли граф сильносвязным, или слабосвязным, или разбит на несколько компонент связности. После чего при помощи функции calc\_dist() строится матрица весов ребер ways. После этого запускается функция check\_ways(). В случае, если построение минимального остовного дерева возможно в текущей конфигурации, то результатом работы функции check\_ways() будет минимальное остовное дерево, заданное в виде списка ребер а также его суммарный вес.

Затем, при каждой итерации цикла каждый из узлов может, или совершить перемещение в пространстве на величину равную его скорости, или остаться на месте. После того, как координаты каждого узла претерпят изменения или сохранят свои прежние значения, инициируется вызов функции check\_config(). На рисунке 1 изображен пример работы функции check-config() после смены местоположения узлов.

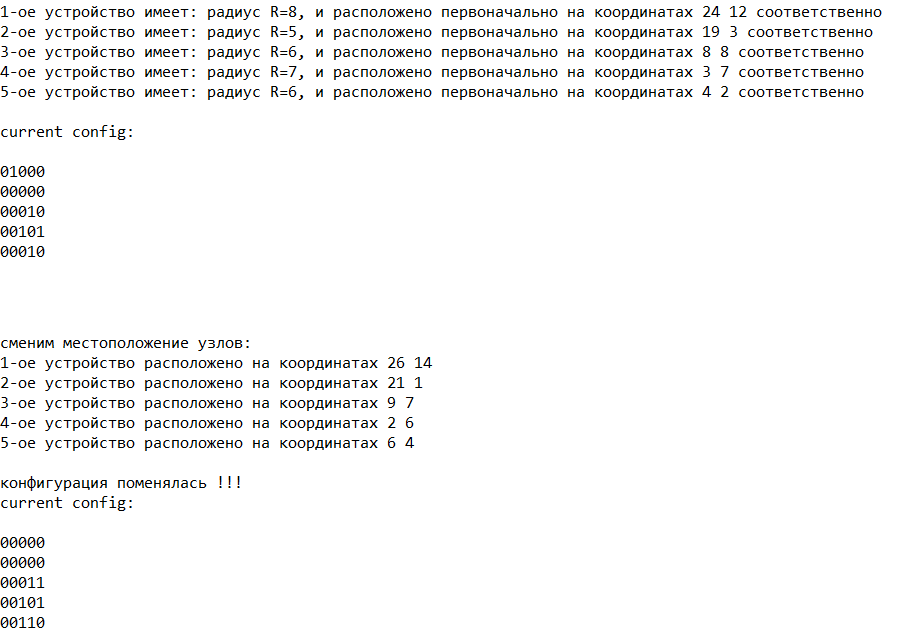


Рисунок 1 ̶ Смена конфигурации в результате работы функции check\_config()

После того, как конфигурация сети была изменена функцией rebuild\_config(), мобильному агенту требуется собрать новую информацию о получившейся конфигурации, в частности, информацию о связности графа и о его компонентах связности.

Связный граф – граф, между какой бы то ни было парой вершин которого существует, как минимум, один путь, т.е. связный граф является одной цельной компонентой связности.

Компонента связности графа — некоторая группа вершин графа, такая, что между любыми двумя вершинами из этой группы существует путь, и не существует путей ведущих из вершин этой группы в вершины графа, не принадлежащие этой группе.

Компонента сильной связности ориентированного графа — множество вершин ориентированного графа, для которых существуют пути как из любой вершины множества в любую другую вершину из того же множества, так и обратный.

Пример ориентированного графа, состоящего из нескольких компонент связности расположен на рисунке 2.

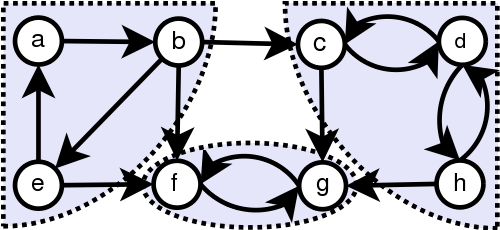


Рисунок 2 ̶ Пример графа, состоящего из 3 компонент сильной связности

Сильно связный граф – ориентированный граф, состоящий из одной компоненты сильной связности. На рисунке 3 изображен элементарный сильно связный орграф.

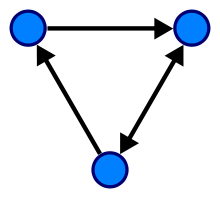


Рисунок 3 ̶ Пример сильно связного орграфа

Слабо связный граф – ориентированный граф, который становится связным, если его ориентированные ребра заменить на неориент ированные. На рисунке 4 изображен элементарный слабо связный граф.

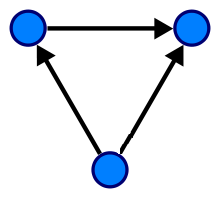


Рисунок 4 ̶ Пример слабо связного орграфа

В целях нахождения компонент сильной связности ориентированного графа могут быть использованы 3 алгоритма: Тарьяна, Косарайю и Габова.

Алгоритм Косарайю — алгоритм, используемый для поиска компонент сильной связности орграфа. Для нахождения компонент сильной связности, используется поиск в глубину (DFS) на графе, содержащем те же ребра, что и исходный, но с инвертированными направлениями этих ребер, во время выполнения которого вычисляется вектор обратного порядка обхода. После чего используется обращение этого вектора для того, чтобы выполнить поиск в глубину на исходном графе. Деревья в лесе DFS, которые выбираются в результате, представляют собой сильные компоненты [6].

На рисунке 5 изображена наглядная визуализация работы алгоритма Косарайю [7].

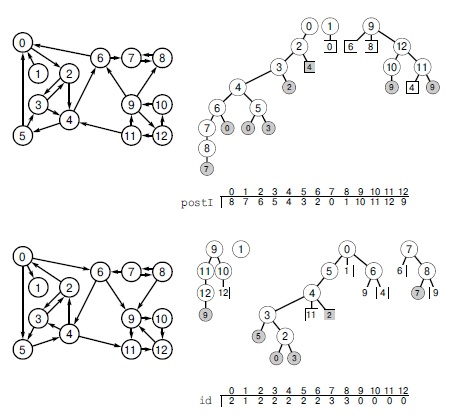


Рисунок 5 ̶ Визуализация работы алгоритма Косарайю

Алгоритм Габова – алгоритм поиска компонент сильной связности в ориентированном графе. В данном алгоритме используются 2 стека: S - для хранения вершин, принадлежащих текущей компоненте связности и P - для хранения пути поиска компонент. В начале работы алгоритма осуществляется обход в глубину исходного графа. При достижении очередной непосещенной вершины алгоритм выполнит следующие действия:

1. вершине v присваивается номер C, после чего переменная C инкрементируется на 1;
2. вершина v помещается в стеки S и P;
3. для каждого ребра, ведущего от v к некоторой вершине w:
   1. если номер вершины w еще не назначен, то аналогичная рекурсивная функция начинает выполняться и для w;
   2. иначе, если w не принадлежит к какой-либо сильной компоненте связности, то из стека P удаляются вершины до тех пор, пока верхним элементом стека P не окажется вершина с номером равным или меньшим, чем номер вершины w.
4. если v является верхним элементом стека P:
   1. из стека S извлекаются все вершины до v, включая эту вершину, после чего извлеченные вершины приписываются новой компоненте связности;
   2. вершина v удаляется из стека P.

Алгоритм начинает выполняться в цикле, осуществляя проверку вершин графа и вызывая свою рекурсивную часть каждый раз при обнаружении непронумерованной вершины. На рисунке 6 изображена наглядная визуализация работы алгоритма Габова .

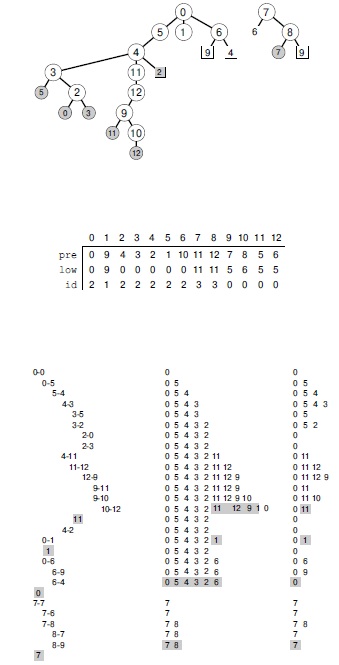


Рисунок 6 ̶ Визуализация работы алгоритма Габова

Алгоритм Тарьяна — алгоритм поиска компонент сильной связности ориентированного графа, работающий за линейное время.

Этот алгоритм основан на том, что:

* просмотр вершин проходит в обратном топологическом порядке, поэтому, в конце исполнения рекурсивной функции для исходной вершины не будет встречено ни одной вершины из той же сильной компоненты, что и исходная вершина, так как все вершины, доступные из исходной уже были обработаны алгоритмом;
* обратные связи в дереве дают второй путь между вершинами а также связывают сильные компоненты.

Алгоритм Тарьяна можно представить в виде модифицированного алгоритма поиска в глубину, в котором над посещаемыми вершинами и над теми вершинами, чья обработка подходит к концу, выполняются некоторые дополнительные действия. При движении от корня к листьям вершины посещаются, а на обратной пути обработка вершин заканчивается. Сразу после того, как вершина была посещена, она проталкивается во вспомогательный стек и после окончания обработки она, соответственно, выталкивается. Индексы компонент связности всех вершин хранятся в векторе id, индексированном номерами вершин. Вектор low позволяет отследить вершину, обладающую наименьшим номером, при прямом обходе, которую можно достигнуть из каждого узла через последовательность прямых связей, за которыми следует одна восходящая связь. Рассматривая вершины в обратном топологическом порядке при помощи поиска в глубину, мы можем вычислить для каждой вершины v максимальную точку, которую возможно достигнуть путем обратной связи из предшественника (low[v]). Когда для вершины v выполняется условие pre[v]=low[v], она и все вершины выше её выталкиваются из стека, после чего им всем присваивается номер следующей компоненты [8]. Наглядная визуализация работы алгоритма Тарьяна изображена на рисунке 7.

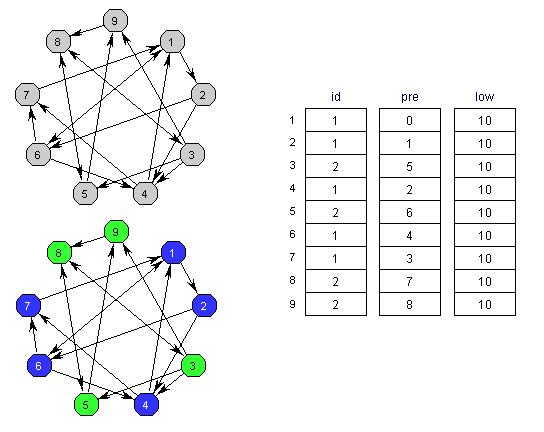


Рисунок 7 ̶ Визуализация работы алгоритма Тарьяна

Для использования в программе был выбран алгоритм Тарьяна, так как он выполняется за линейное время, а также является оптимальным алгоритмом для используемой структуры данных.

Для определения связности графа в программе предназначена функция check\_connections(). Эта функция проверяет граф на сильную связность, используя алгоритм Тарьяна. По результатам работы алгоритма: либо граф будет состоять из 1 компоненты связности, т.е. являться сильносвязным графом, либо будет произведена проверка графа на слабую связность путем вызова функции check\_undirected(). Эта функция работает следующим образом: строится неориентированная копия исходного графа, осуществляется обход графа в глубину (DFS) и если за один обход были посещены все вершины неориентированного графа, то соответствующий ему ориентированный граф будет являться слабосвязным графом – в вызывающую функцию будет возвращено значение true, иначе работа программы завершится в обычном режиме и в вызывающую программу будет возвращено значение false. Если граф окажется ни сильно, ни слабо связным, то функция выведет все его вершины с указанием, к какой именно компоненте связности они принадлежат. Все возможные варианты работы функции check\_connections() приведены на рисунках 8, 9 и 10.

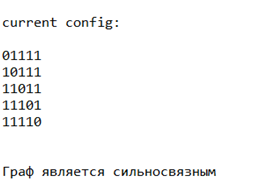


Рисунок 8 ̶ Результат работы функции check\_connections() в случае сильно связного графа

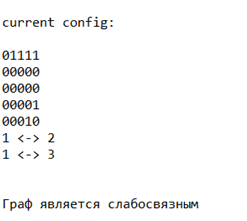


Рисунок 9 ̶ Результат работы функции check\_connections() в случае слабо связного графа

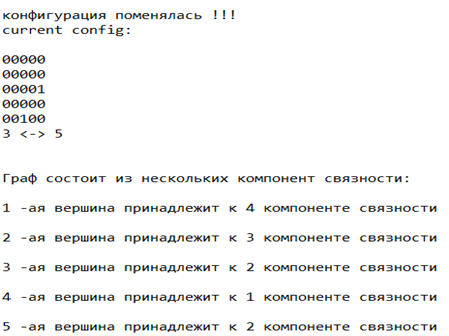


Рисунок 10 ̶ Результат работы функции check\_connections() в случае графа, разбитого на компоненты сильной связности

Функция check\_connections() обрабатывает все возможные варианты построения текущей конфигурации сети, однако она является крайне сложной, в виду чего имеется потребность в создании некоего алгоритма, который выполнял бы предварительную проверку конфигурации с целью выяснения, требуется ли запускать мобильному агенту функцию check\_config(). Функция, запускающая работу этого алгоритма, называется check\_bridges\_equality(), и использует при своей работе алгоритм Тарьяна для поиска мостов в графе. На рисунке 11 отображена работа этого алгоритма [9].

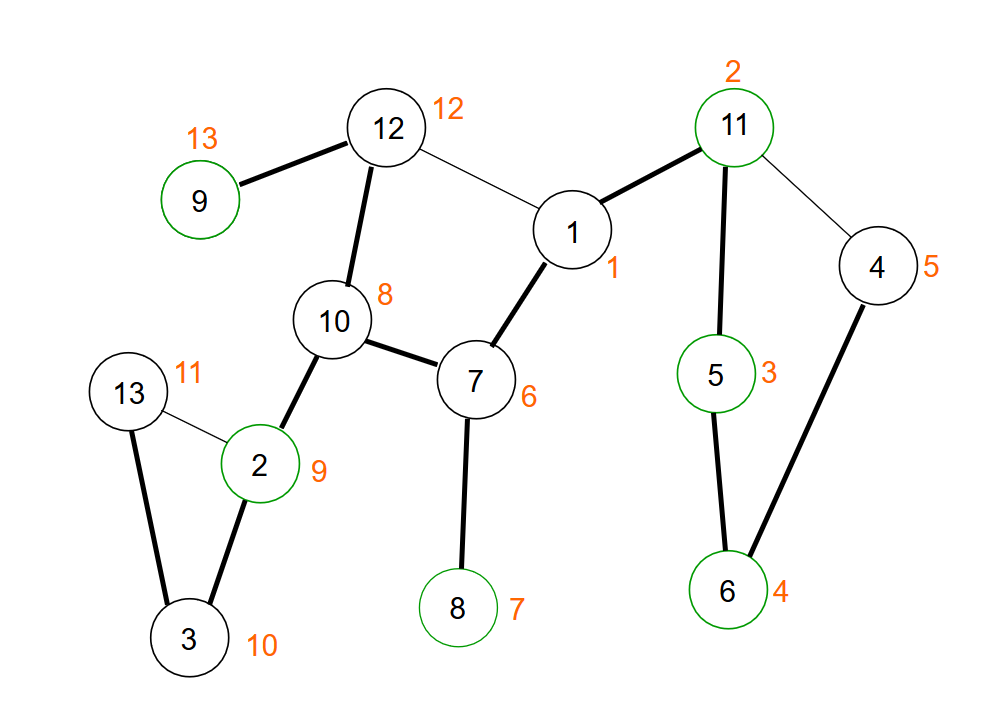


Рисунок 11 – Визуализация работы алгоритма Тарьяна для поиска мостов

Алгоритм работает следующим образом:

1. осуществляется построение остовного леса графа G;
2. создается лес с корнями F из остовного дерева;
3. в лесу F в прямом порядке нумеруются вершины, таким образом, получится, что родительские вершины будут иметь меньшие номера, чем их потомки;
4. для каждой вершины v в порядке прямого обхода выполняются следующие действия:
   1. подсчитывается число потомков ND(v) для вершины, добавляя единицу при переходе к следующей вершине (включая вершину v).
   2. Осуществляется вычисление минимального номера L(v), до которого можно дойти по дугам поддерева из вершины v(за исключением последней дуги). Этот минимальный номер в множестве, состоящем из значений L(w) потомков вершины v и номеров вершин, до которых можно дойти из v по дугам, не принадлежащим F.
   3. Аналогичным образом вычисляется H(v), наибольший номер, до которого можно дойти по дугам начинающегося в вершине v поддерева. Это максимальный номер в множестве значений, состоящем из значений H(w) потомков вершины v и номеров вершин, до которых можно дойти из v по дугам, не принадлежащим F.
   4. Для каждой вершины w, являющейся потомком v проверяем, если L(w)=w и H(w)<w+ND(w), то ребро из v в w является мостом.

Если v 🡪 w — мост, то ясно, что из поддерева с корнем в w возможность выйти на вершину, не из этого поддерева отсутствует. Для проверки этого достаточно проверить L(w), H(w) — условие L(w) = w означает, что мы не выйдем на вершину, лежащую ближе к корню, а условие H(w)<w+ND(w)— что мы не выйдем на другое поддерево. Функция check\_bridges\_equality() при каждой смене конфигурации сети выполняет поиск мостов в графе конфигурации, после чего сравнивает полученный вектор мостов с предыдущим, который хранится в памяти мобильного агента – если изменений в векторах не было обнаружено, то функция выдаст сообщение: «мосты в графе остались без изменений» и завершится в штатном режиме, иначе будет вызвана функция проверки связности check\_connections(). На рисунке 12 приведен пример работы этой функции.

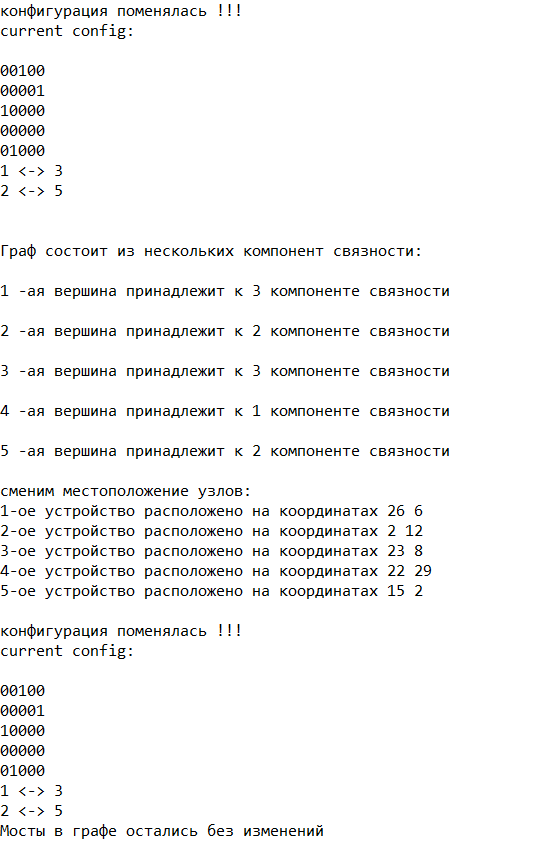
****

Рисунок 12 – Пример работы функции check\_config() вместе с функцией check\_bridges\_equality()

После завершения работы функции check\_connections(), будет вызвана функция check\_ways() для нахождения минимального остовного дерева ориентированного графа. Поиск минимального остовного дерева осуществляется при помощи модифицированного алгоритма Прима.

Суть его работы заключается в следующем: берется произвольная вершина и просматривается соответствующая ей строка матрицы на предмет минимального ребра, исходящего из первой вершины. Когда такое ребро будет найдено, то вершина, из которой выходит ребро, и вершина, в которую ведет это ребро будут помечены, как посещенные в специальном массиве меток.

Далее построение минимального остовного дерева ориентированного графа будет выполняться рекурсивно по следующему принципу:

1. все множество помеченных как посещенные вершины просматривается с целью выбрать ребро минимального веса, такое, что вершина, из которой оно исходит, принадлежит множеству помеченных вершин, а вторая вершина является непомеченной;
2. ребро помещается в список ребер и непосещенная ранее вершина, в которую вело это ребро помечается как посещенная;
3. выполняется проверка условия – если все вершины были посещены, то работа алгоритма прекращается, в противном случае работа алгоритма будет продолжена с шага 1.

Пошаговый пример работы немодифицированного алгоритма Прима для неориентированного графа приведен на рисунке 13 [10].

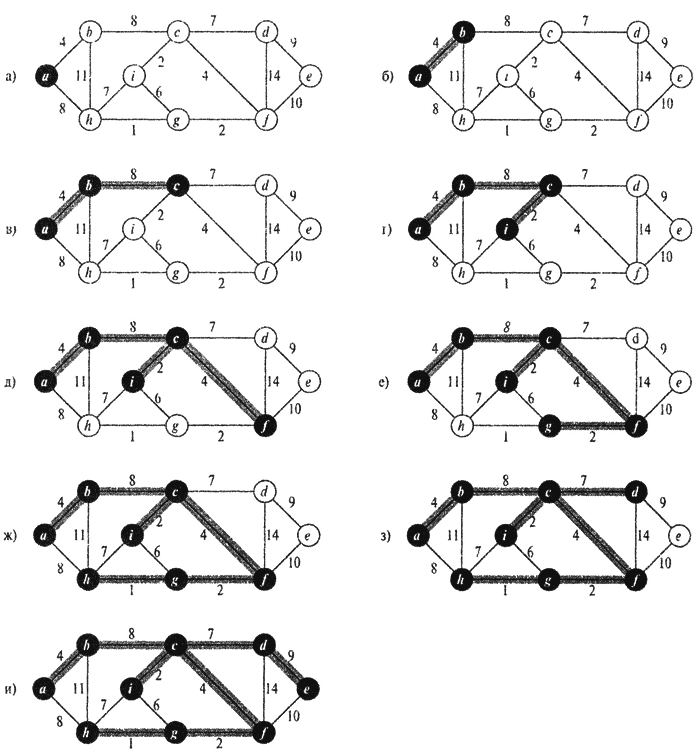


Рисунок 13 – Визуализация пошаговой работы алгоритма Прима для неориентированных графов

На рисунке 14 можно увидеть пример работы функции check\_ways().

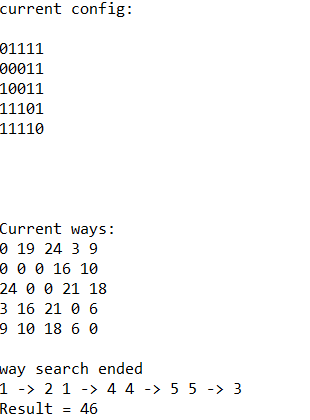


Рисунок 14 – Пример работы функции check\_ways()

2.2 Реализация оптимизированного алгоритма предсказывающего характеристики динамических графов компьютерных сетей

Функция check\_ways() находит минимальное остовное дерево в ориентированном графе нашей конфигурации. Но далеко не во всех возможных ситуациях, которые могут возникнуть при изменении конфигурации сети, построение минимального остовного дерева будет вообще возможно. Минимальное остовное дерево в ориентированном графе можно построить только если он является сильносвязным графом. С целью сократить число пустых прогонов функции check\_ways() был реализован следующий механизм: к функции check\_ways() был добавлен параметр булевского типа strong. При успешном срабатывании функции check\_connections(), если был получен результат о том, что граф является сильносвязным, то параметр strong устанавливается в значение true. И функция check\_ways() каждый раз проверяет перед началом своей работы, истинен ли параметр strong. В случае если он истинен, то функция запустится и отработает как обычно. Иначе будет выведено сообщение о том, что при текущей конфигурации графа динамической сети построение минимального остовного дерева не возможно.

На рисунках 15 и 16 изображены результаты работы оптимизации функции check\_way() в случае сильносвязного графа (рисунок 15) и во всех остальных случаях конфигурации графа, а именно для случая слабосвязного графа или графа разбитого на компоненты связности (рисунок16).

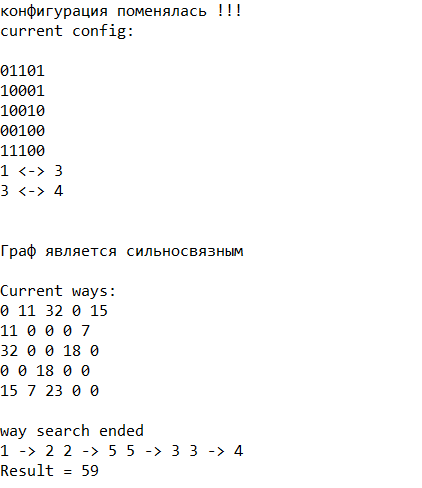


Рисунок 15 – Пример работы оптимизированной функции check\_ways() в случае сильносвязного графа

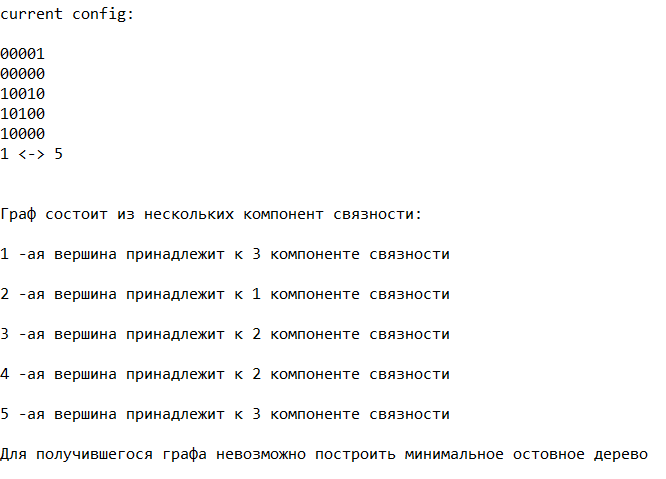


Рисунок 16 – Пример работы оптимизированной функции check\_ways() в случае графа, разбитого на компоненты связности

Используемая оптимизация функции check\_ways()на самом деле позволяет добиться выигрыша по времени по сравнению со старой реализацией модели мобильного агента, лишенной её. Например, средняя сложность работы функции check\_config() без оптимизации составляет 52743 элементарных операции, в то время как с оптимизацией средняя сложность составляет всего 47343 элементарных операций. В обоих случаях работа функций оценивалась для конфигурации из 10 устройств и выполнялась функция на этой конфигурации 1000 раз.

На рисунках 17 и 18 приведены диаграммы распределения сложности работы функции как с оптимизацией алгоритма, так и без неё (сложность измеряется в элементарных операциях).

Рисунок 17 – Диаграмма распределения сложности работы оптимизированной функции check\_ways()

Рисунок 18 – Диаграмма распределения сложности работы функции check\_ways() без оптимизации

При анализе этих диаграмм можно увидеть, что в старом алгоритме большая часть оценок сложности (40%) лежит в интервале больше 50000 элементарных операций, в то время как в новом алгоритме большинство значений (46%) лежит в интервале до 25000 элементарных операций. Из этого следует, что оптимизация функции работает и уменьшает сложность работы мобильного агента, это может подтвердить сравнение средней сложности работы алгоритмов: у старого алгоритма она составляет 52743 элементарных операций, а у нового алгоритма - 47343 элементарных операций, то есть прирост производительности функции составил около 10%.

# ЗАКЛЮЧЕНИЕ

В ходе выполнения работы была пересмотрена модель динамического графа компьютерной сети: была добавлена такая характеристика, как расстояние между вершинами, а также была добавлена функция, позволяющая её подсчитывать, что позволило более реалистично моделировать динамический граф компьютерной сети и работу мобильного агента.

Кроме того, были изучены и успешно реализованы алгоритмы, предсказывающие характеристики графов.

Также был реализован алгоритм, строящий минимальное остовное дерево в динамическом графе компьютерной сети. Реализованный алгоритм был успешно оптимизирован добавлением проверки графа на сильную связанность.

Созданные алгоритмы можно использовать в уже существующих мобильных агентах в целях оптимизации их работы.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Toh, Chai Keong. Ad Hoc Mobile Wireless Networks. MI: Prentice Hall Publishers, 2002. ̶ 302 P.
2. Toh, Chai Keong. Associativity-Based Routing for Ad Hoc Mobile Networks. Neth.: Kluwer Academic Publishers, 1997. ̶ P. 103-139.
3. Бителева А. Технологии мультимедийного доступа // Теле-Спутник. ̶ 2002. ̶ №8. ̶ С. 82.
4. Вишневский и др.. Широкополосные беспроводные сети передачи данных. М.: Техносфера, 2005, ̶ 592 С.
5. Верхулевский К, Шаропин Ю. Открытый стандарт беспроводной сети One-Net и аппаратные решения на его основе // Современная Электроника. ̶ 2008. ̶ № 8 ̶ С. 12.
6. Седжвик Р. Фундаментальные алгоритмы на C++. Часть 5: Алгоритмы на графах. ̶ СПб.: ДиаСофтЮП, 2002. ̶ 496 С.
7. Лекция 19: Орграфы и DAG-графы [Электронный ресурс]. - URL: http://www.intuit.ru/studies/courses/12181/1174/lecture/25266?page=11 (дата обращения: 20.05.2018).
8. Tarjan R.Е. Depth-first search and linear graph algorithms // SIAM Journal on Computing. ̶ 1972. ̶ P. 146–160.
9. Pre-order traversal and finding bridges with Tarjan’s algorithm [Электронный ресурс]. - URL: http://csengerg.github.io/2015/12/26/pre-order-travelsal-and-tarjans-algorithm.html (дата обращения: 30.05.2018).
10. Библиотека алгоритмов на графах. Алгоритм Прима [Электронный ресурс]. - URL: http://urban-sanjoo.narod.ru/prim.html (дата обращения: 01.05.2018).

# ПРИЛОЖЕНИЕ

# Код программы

#include<iostream>

#include<fstream>

#include<cmath>

#include<vector>

#include<algorithm>

#include<stack>

#include<random>

#include<list>

using namespace std;

int calc\_dist(int i, int j, int \* X, int \* Y, int &tim);

bool check\_device(int N, int i, int \*\*Graph, int \* R, int \* X, int \* Y, int &tim);

void rebuild\_config(int N,int \*\* Graph, int \* R, int \* X, int \* Y, int &tim);

void check\_config(int N, int\*\* Graph, int\*\* & ways,int \* R, int \* X, int \* Y, list<pair<int, int> > &Bridges,int &tim, bool&strong);

void check\_connections(int N,int \*\* Graph, int &tim,bool&strong);

bool check\_undirected(int N,int \*\* Graph, int &tim);

void DFS\_undir\_gen(int\*\*tmp, int N,int i,bool \* visited, int &tim);

//Функции для решения задачи построения минимального остовного дерева в ориентированном графе

void check\_ways(int N, int \*\* & ways, int \*\* Graph, int \*X, int \* Y, int &tim);

void find\_way(int N, int \*\* ways, list<int> & way, /\*int & buf,\*/ int & tim);

void find\_way\_rec(int N, int \*\* ways, list<int> & way, bool \* & visited, int & tim);

//Функции для проверки на необходимость поиска компонент связности

list<pair<int, int> > Find\_Bridges(int\*\* Graph,int N, int &tim);

bool check\_bridges\_equality(list<pair<int, int> >&Bridges, list<pair<int, int> >&Pos\_Bridges, int &tim);

//Функции для работы с алгоритмом Тарьяна

void Inverse(int \*\* Graph, int \*\* tmp, int N, int &tim);

void DFS(int \*\* Inv, int N, int i, vector<int> & path,bool\* visited, int &tim);

vector<int> DFS\_Inverse(int\*\* Inv, int N, int &tim);

void Tarjan\_sec(int v, int\* & pre, int\* & low, stack<int> & S, int & Next, int\*\* & Graph, int N, int\* & ind, int & KN, int &tim);

void Tarjan\_Prime(int \*\* Graph, int N,int \* ind, int &tim);

bool in(int elem, stack<int> S, int &tim){

while (!S.empty()){

tim++;

int key = S.top();

S.pop();

if (key == elem)

return true;

}

return false;

}

int main(){

setlocale(LC\_ALL,"");

system("color F0");

ifstream fin("input.txt");

random\_device rd;

uniform\_int\_distribution<int> uid(0, 30);

int N;

fin >> N;

int \* R = new int[N], \*X = new int[N], \*Y= new int [N], \*Sp=new int [N];

int \*\* Graph = new int\*[N], \*\* ways = new int\*[N];

for (int i = 0; i < N; i++){

Graph[i] = new int[N];

Sp[i] = rand() % 3;

for (int j = 0; j < N; j++)

Graph[i][j] = 0;

fin >> R[i];

X[i] = uid(rd);

Y[i] = uid(rd);

cout <<i+1<<"-ое устройство имеет: радиус R=" <<R[i]<<", и расположено первоначально на координатах "<<X[i]<<" "<<Y[i]<<" соответственно"<<endl;

}

cout << endl;

int tim = 0;

rebuild\_config(N, Graph, R, X, Y,tim);

list<pair<int, int> >Bridges = Find\_Bridges(Graph,N,tim);

for (pair<int, int> p : Bridges)

cout << p.first + 1 << " <-> " << p.second + 1 << endl;

bool strong = false;

check\_connections(N, Graph,tim,strong);

if(strong)

check\_ways(N, ways, Graph, X, Y, tim);

else

cout << "\nДля получившегося графа невозможно построить минимальное остовное дерево\n";

ofstream output("Output.txt");

for (int i = 0; i < 5; i++){

tim = 0;

cout << endl << "сменим местоположение узлов: " << endl;

for (int i = 0; i < N; i++){

if (uid(rd) < 15 && X[i] > 0) {

if (X[i] - Sp[i] >= 0)

X[i] -= Sp[i];

else

X[i] = 0;}

else

if (X[i] + Sp[i] <= 30) {

X[i] += Sp[i];}

else

X[i] = 30;

if (uid(rd) < 15 && Y[i] > 0) {

if (Y[i] - Sp[i] >= 0)

Y[i] -= Sp[i];

else

Y[i] = 0;

}

else

if (Y[i] + Sp[i] <= 30) {

Y[i] += Sp[i];

}

else

Y[i] = 30;

cout << i + 1 << "-ое устройство расположено на координатах " << X[i] << " " << Y[i] << endl;

}

tim += N;

cout << endl;

check\_config(N, Graph, ways,R, X, Y, Bridges,tim, strong);

output << tim << " cекунд прошло" << endl;

}

delete[]X;

delete[]Y;

delete[]R;

for (int i = 0; i < N; i++)

delete[] Graph[i];

delete[] Graph;

cout << endl << endl;

system("pause");

return 0;

}

int calc\_dist(int i, int j, int \* X, int \* Y, int &tim){

tim += 8;

int result = round(sqrt(pow((X[j] - X[i]),2) + pow((Y[j] - Y[i]),2)));

return result;

}

//Функции для решения задачи построения минимального остовного дерева ориентированного графа

void check\_ways(int N, int \*\* & ways, int \*\* Graph, int \*X, int \* Y, int &tim) {

cout << endl << "Current ways:\n";

for (int i = 0; i < N; i++)

ways[i] = new int[N];

for (int i = 0; i<N; i++) {

ways[i][i] == 0;

for (int j = 0; j<N; j++) {

if (Graph[i][j] == 1)

ways[i][j] = calc\_dist(i, j, X, Y, tim);

else

ways[i][j] = 0;

cout << ways[i][j] << " ";

}

cout << endl;

}

list <int> way;

find\_way(N, ways, way, tim);

int result = 0, first, second;

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

if (i % 2 == 0) {

first = way.front();

cout << (way.front() + 1) << " ";

}

else {

cout << "-> " << (way.front() + 1) << " ";

second = way.front();

result += ways[first][second];

}

way.push\_back(way.front());

way.pop\_front();

}

cout << "\nResult = " << result << endl;

}

void find\_way(int N, int \*\* ways, list<int> & way, int & tim) {

int min = INT\_MAX,next;

bool \* visited = new bool[N];

for (int i = 0; i<N; i++){

visited[i] = false;

if ((min > ways[0][i]) && ways[0][i] != 0) {

min = ways[1][i];

next = i;

}}

way.push\_back(0);

way.push\_back(next);

visited[0] = true;

visited[next] = true;

if (way.size() == (2 \* N -2)){

cout << "\nway search ended \n";

return ;}

else

find\_way\_rec( N, ways, way, visited, tim);

}

void find\_way\_rec(int N, int \*\* ways, list<int> & way, bool \* & visited, int & tim) {

int min = INT\_MAX, root = -1, next=-1;

for (int i = 0; i < N; i++){

if(visited[i])

for (int j = 0; j < N; j++)

if (!visited[j])

if ((min > ways[i][j])&&ways[i][j]>0) {

min = ways[i][j];

root = i;

next = j;

}

}

way.push\_back(root);

way.push\_back(next);

visited[next] = true;

if (way.size() == (2 \* N - 2 )){

cout << "\nway search ended \n";

return ;}

else

find\_way\_rec( N, ways, way, visited, tim);

}

void rebuild\_config(int N, int \*\* Graph, int \* R, int \* X, int \* Y, int &tim) {

cout << "current config: " << endl;

for (int i = 0; i < N; i++) {

cout << endl;

for (int j = 0; j < N; j++) {

double tmp1 = (X[i] \* X[i]) + (Y[i] \* Y[i]);

double tmp2 = (X[j] \* X[j]) + (Y[j] \* Y[j]);

double tmp3 = abs(sqrt(tmp2) - sqrt(tmp1));

if (tmp3 < R[i])

Graph[i][j] = 1;

else

Graph[i][j] = 0;

if (tmp3 < R[j])

Graph[j][i] = 1;

else

Graph[j][i] = 0;

Graph[j][j] = 0;

tim += 15;

cout << Graph[i][j];

}

}

cout << endl;

}

bool check\_device(int N, int i, int \*\*Graph, int \* R, int \* X, int \* Y, int &tim){

for (int j = 0; j < N; j++){

tim++;

double tmp1 = (X[i] \* X[i]) + (Y[i] \* Y[i]);

double tmp2 = (X[j] \* X[j]) + (Y[j] \* Y[j]);

double tmp3 = abs(sqrt(tmp2) - sqrt(tmp1));

tim += 13;

if (tmp3 < R[i]){

if (Graph[i][j] == 0){

tim += 2;

return true;}}

else{

if (Graph[i][j] == 1){

tim += 2;

return true;}};

tim += 2;

if (tmp3 < R[j]){

if (Graph[j][i] == 0){

tim += 2;

return true;

}

}

else{

if (Graph[j][i] == 1){

tim += 2;

return true;

}

};

tim += 2;

}

return false;

}

void check\_config(int N, int \*\* Graph, int\*\* & ways, int \* R, int \* X, int \* Y, list<pair<int, int> > &Bridges,int & tim, bool&strong){

for (int i = 0; i < N; i++){

tim++;

if (check\_device(N, i, Graph, R, X, Y,tim)){

cout << "конфигурация поменялась !!!" << endl;

rebuild\_config(N, Graph, R, X, Y,tim);

list<pair<int, int> > Pos\_Bridges = Find\_Bridges(Graph,N,tim);

for (pair<int, int> p : Pos\_Bridges)

cout << p.first + 1 << " <-> " << p.second + 1 << endl;

if (check\_bridges\_equality(Bridges,Pos\_Bridges,tim)){

cout << "Мосты в графе остались без изменений" << endl;

break;}

else{

int size = Pos\_Bridges.size(), tmpsize=Bridges.size();

tim += tmpsize+2;

for (int j = 0; j < tmpsize; j++)

Bridges.pop\_front();

for (int k = 0; k < size; k++){

tim++;

pair<int, int> tmp = Pos\_Bridges.front();

Bridges.push\_back(tmp);

Pos\_Bridges.pop\_front();}

check\_connections(N, Graph,tim,strong);

if (strong)

check\_ways(N, ways, Graph, X, Y, tim);

else

cout << "\nДля получившегося графа невозможно построить минимальное остовное дерево\n";

break;}}}}

bool check\_bridges\_equality(list<pair<int, int> >&Bridges, list<pair<int, int> >&Pos\_Bridges, int &tim){

list<pair<int, int> > temp\_Bridge1 = Bridges, temp\_Bridge2 = Pos\_Bridges;

int size1 = Bridges.size(), size2 = Pos\_Bridges.size();

tim += 5;

if (size1 != size2)

return false;

for (int i = 0; i < size1;i++) {

tim += 5;

pair<int, int> tmp1 = temp\_Bridge1.front(),tmp2=temp\_Bridge2.front();

if (tmp1.first != tmp2.first || tmp1.second != tmp2.second)

return false;

temp\_Bridge1.pop\_front();

temp\_Bridge2.pop\_front();

}

return true;

}

void check\_connections(int N, int \*\* Graph, int &tim, bool&strong)

{

bool flag = true;

int \* ind = new int[N];

tim += 2;

Tarjan\_Prime(Graph,N,ind,tim);

for (int i = 0; i < N; i++){

tim += 2;

if (ind[i] != 1){

flag = false;

break;}}

if (flag){

strong = true;

cout << "Граф является сильносвязным\n";

delete[] ind;

return;}

if (check\_undirected(N, Graph,tim)){

cout<< "Граф является слабосвязным\n";

delete[] ind;

return;

}

cout<< "Граф состоит из нескольких компонент связности:\n";

for (int i = 0; i < N; i++)

cout << endl << i + 1 << " -ая вершина принадлежит к " << ind[i] << " компоненте связности\n";

delete[] ind;

}

bool check\_undirected(int N, int \*\* Graph, int &tim) {

int \*\* tmp = new int \*[N];

tim++;

for (int i = 0; i < N; i++) {

tmp[i] = new int[N];

tim++;

for (int j = 0; j < N; j++){

tim += 2;

tmp[i][j] = Graph[i][j];}}

for (int i = 0; i < N; i++)

for (int j = 0; j < N; j++){

tim += 2;

if (tmp[i][j] == 1)

tmp[j][i] = 1;}

bool \* visited = new bool[N];

for (int i = 0; i < N; i++)

visited[i] = false;

tim += N + 1;

for (int i = 0; i < N; i++) {

tim++;

if (!visited[i])

DFS\_undir\_gen(tmp, N, i, visited,tim);

for (int j = 0; j < N; j++)

if (!visited[j])

return false;}

return true;

}

void DFS\_undir\_gen(int \*\* Undir\_Graph, int N, int i, bool \*visited, int &tim) {

visited[i] = true;

tim += N + 1;

for (int j = 0; j < N; j++) {

if (Undir\_Graph[i][j] == 1) {

if (!visited[j])

DFS\_undir\_gen(Undir\_Graph, N, j, visited,tim);}}}

// Функции для Тарьяна

void Inverse(int \*\* Graph, int \*\* tmp, int N, int &tim){

for(int i=0;i<N;i++)

for (int j = 0; j < N; j++){

if (Graph[i][j] == 1)

tmp[j][i] = 1;}

tim += N^2;}

void DFS(int \*\* Inv,int N, int i, vector<int> & path, bool \*visited, int &tim){

path.push\_back(i);

visited[i] = true;

tim += 2;

for (int j = 0; j < N; j++){

tim++;

if (Inv[i][j] == 1){

if (!visited[j])

DFS(Inv,N, j, path, visited,tim);}}}

vector<int> DFS\_Inverse(int\*\* Inv,int N, int &tim){

vector<int> path;

bool \* visited = new bool[N];

for (int i = 0; i < N; i++)

visited[i] = false;

tim += N + 2;

for (int i = 0; i < N; i++)

if (!visited[i])

DFS(Inv, N, i, path, visited,tim);

tim += N;

return path;

}

void Tarjan\_sec(int v, int\* & pre, int\* & low, stack<int> & S, int & Next, int\*\* & Graph,int N, int\* & ind, int & KN, int &tim){

pre[v] = Next;

low[v] = Next;

Next++;

S.push(v);

tim += 4;

for(int j=0; j<N;j++){

tim++;

if(Graph[v][j]==1)

if (pre[j] == -1){

Tarjan\_sec(j, pre, low, S, Next, Graph,N, ind, KN,tim);

low[v] = min(low[v], low[j]);}

else

if (in(j, S,tim)){

low[v] = min(low[v], low[j]);}}

if (low[v] == pre[v]){

while (in(v, S,tim)){

int u = S.top();

S.pop();

ind[u] = KN;

tim += 3;}

KN++;}}

void Tarjan\_Prime(int \*\* Graph, int N, int \* ind, int &tim){

int KN = 1, Next = 0;

int\*pre = new int[N], \*low = new int[N], x = 0, y = 0;

for (int i = 0; i < N; i++){

pre[i] = -1;

low[i] = 0;

ind[i] = 0;

}

tim += 6 + 3 \* N;

int \*\*Inv = new int\*[N];

for (int i = 0; i < N; i++)

Inv[i] = new int[N];

tim += N+1;

for (int i = 0; i < N; i++)

for (int j = 0; j < N; j++)

Inv[i][j] = 0;

tim += N ^ 2;

Inverse(Graph, Inv, N,tim);

vector<int> path = DFS\_Inverse(Inv, N,tim);

int nn = path.size();

stack<int> S;

for (int i = 0; i < nn; i++){

int v = path.back();

path.pop\_back();

if (pre[v] == -1)

Tarjan\_sec(v, pre, low, S, Next, Inv,N,ind, KN,tim);

}

tim += nn;

cout << endl;

cout << endl;

delete[]pre;

delete[]low;

}

//Функции для алгоритма поиска мостов в Графе

vector<vector<bool> > MakeSpanningForest(int\*\* Graph,int N, int &tim){

vector<vector<bool> > SpanningForest(N);

for (vector<bool> &vect : SpanningForest)

vect.resize(N, 0);

vector<int> used(N, false);

list<int> Queue;

for (int i = 0; i < N; ++i)

if (!used[i]){

Queue.push\_back(i);

used[i] = true;

while (!Queue.empty()){

tim++;

int v = Queue.front();

Queue.pop\_front();

for (int u = 0; u < N; ++u)

if ((Graph[v][u]==1) && !used[u]){

Queue.push\_back(u);

used[u] = true;

SpanningForest[v][u] = true;}

tim += N;}}

return SpanningForest;}

void DFS\_N\_ND(vector<vector<bool>> Graph, vector<int> &Number, vector<int> &ND, vector<bool> &used, int v, int &currentNumber, int &tim)

{

used[v] = true;

Number[v] = currentNumber++;

for (int u = 0; u < Graph.size(); ++u){

tim++;

if (Graph[v][u] && !used[u]){

tim++;

DFS\_N\_ND(Graph, Number, ND, used, u, currentNumber,tim);

ND[v] += ND[u];}}

ND[v] += 1;

tim++;

}

void DFS\_Number\_ND(vector<vector<bool>> Graph, vector<int> &Number, vector<int> &ND, int &tim){

int N = Graph.size();

Number.assign(N, -1);

ND.assign(N, 0);

tim++;

vector<bool> used(N, false);

int currentNumber = 0;

tim++;

for (int v = 0; v < N; ++v){

tim++;

if (!used[v])

DFS\_N\_ND(Graph, Number, ND, used, v, currentNumber,tim);}}

void DFS\_MIN\_MAX(int\*\* Graph, int N, const vector<vector<bool> > &SpanningForest, const vector<int> Number, vector<bool> used, int v, int &MIN, int &MAX, int &tim){

used[v] = true;

if (Number[v] < MIN)

MIN = Number[v];

if (Number[v] > MAX)

MAX = Number[v];

tim += 3;

for (int u = 0; u < N; ++u){

tim++;

if (Graph[v][u]==1 && !SpanningForest[v][u] && !SpanningForest[u][v] && !used[u])

DFS\_MIN\_MAX(Graph, N, SpanningForest, Number, used, u, MIN, MAX,tim);}}

void DFS\_L\_H(int\*\* Graph, int N, const vector<vector<bool> > &SpanningForest, const vector<int> Number, vector<int> &Lowest, vector<int> &Highest, int v, int &tim)

{

int currentLowest = Number[v];

int currentHighest = Number[v];

vector<bool> used;

for (int u = 0; u < N; ++u){

tim++;

used.assign(N, false);

if (SpanningForest[v][u] && (Lowest[v] == -1)){

tim++;

DFS\_L\_H(Graph, N, SpanningForest, Number, Lowest, Highest, u,tim);

if (Lowest[u] < currentLowest)

currentLowest = Lowest[u];

if (Highest[u] > currentHighest)

currentHighest = Highest[u];

tim += 2;}

else if ((Graph[v][u]==1) && !SpanningForest[u][v]){

tim++;

DFS\_MIN\_MAX(Graph, N, SpanningForest, Number, used, u, currentLowest, currentHighest,tim);}}

Lowest[v] = currentLowest;

Highest[v] = currentHighest;

tim += 2;

}

void DFS\_Lowest\_Highest(int\*\* Graph, int N,const vector<vector<bool> > &SpanningForest, const vector<int> &Number, vector<int> &Lowest, vector<int> &Highest, int &tim)

{

Lowest.assign(N, -1);

Highest.assign(N, -1);

for (int v = 0; v < N; ++v){

tim++;

if (Lowest[v] == -1)

DFS\_L\_H(Graph, N, SpanningForest, Number, Lowest, Highest, v,tim);}

}

list<pair<int, int> > Find\_Bridges(int\*\* Graph,int N, int &tim)

{

vector<vector<bool> > SpanningForest = MakeSpanningForest(Graph,N,tim);

vector<int> Number;

vector<int> ND;

DFS\_Number\_ND(SpanningForest, Number, ND,tim);

vector<int> Lowest;

vector<int> Highest;

DFS\_Lowest\_Highest(Graph, N,SpanningForest, Number, Lowest, Highest,tim);

list<pair<int, int> > Bridges;

for (int v = 0; v < N; ++v)

for (int u = 0; u < N; ++u)

if (SpanningForest[v][u] && (Lowest[u] == Number[u]) && (Highest[u] < Number[u] + ND[u]))

{

int nv, nu;

if (v < u)

nv = v;

else

nv = u;

if (v > u)

nu = v;

else

nu = u;

Bridges.emplace(Bridges.end(), nv, nu);

}

tim += N ^ 2;

return Bridges;

}