МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное бюджетное образовательное учреждение высшего образования
«Тольяттинский государственный университет»

Институт математики, физики и информационных технологий
(наименование института полностью)

Кафедра «Прикладная математика и информатика»
(наименование)

02.03.03 Математическое обеспечение и администрирование информационных систем
(код и наименование направления подготовки, специальности)

Мобильные и сетевые технологии
(направленность (профиль)/специализация)

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА (БАКАЛАВРСКАЯ РАБОТА)

на тему Разработка ПО моделирования конфигурации и оценки времени прохождения пакетов в Mesh-сети

Обучающийся Н.И. Карасев
(Инициалы Фамилия) (личная подпись)

Руководитель С.В. Митин
(ученая степень (при наличии), ученое звание (при наличии), Инициалы Фамилия)

Консультант А.В. Москалюк
(ученая степень (при наличии), ученое звание (при наличии), Инициалы Фамилия)

Тольятти 2023

Аннотация

Тема бакалаврской работы – «Разработка ПО моделирования конфигурации и оценки времени прохождения пакетов в Mesh-сети».

В настоящее время информационные технологии стремительно развиваются, что приводит к глобальному распространению вычислительных устройств, используемых в повседневной жизни. С распространением такой концепции, как интернет вещей (IoT), автоматизации промышленных производств, беспилотных автомобилей и других современных направлений развития информационных технологий значимость сетевых технологий для объединения всех устройств существенно возрастает. Одним из развивающихся направлений в этой области являются mesh-сети. Бакалаврская работа посвящена созданию программного обеспечения для моделирования эффективной конфигурации mesh-сети.

Предмет исследования – процесс моделирования конфигурации и оценка времени прохождения пакетов в mesh-сетях.

Объект исследования – подходы для организации и эффективного взаимодействия устройств в mesh-сети и принципы построения mesh-сетей.

Цель работы – разработка программы для моделирования эффективной конфигурации сети и оценки времени прохождения пакетов в ней.

Во введении описывается актуальность данной работы, определяются цель и необходимые для решения задачи, объект и предмет исследования. В первом разделе работы ставится задача на исследование и разрабатывается ее математическая модель. Во втором разделе выбирается алгоритм для реализации программного обеспечения, разрабатывается и тестируется программа для построения эффективной mesh-сети. В заключении приводятся результаты бакалаврской работы.

Выпускная квалификационная работа содержит пояснительную записку объемом 56 страниц, 18 рисунков, 2 таблицы, 2 формулы и список используемой литературы, состоящий из 20 источников.

Abstract

The topic of the graduation work is "Development of software for modeling the configuration and estimation of packet transit time in mesh network".

Currently, information technology is rapidly evolving, leading to the global proliferation of computing devices used in everyday life. With the spread of concepts such as the Internet of Things (IoT), automation of industrial production, driverless cars and other modern trends in information technology, the importance of network technology to connect all the devices is increasing significantly. One of the developing areas in this area are mesh networks. This graduation work is devoted to creating software for modeling an effective mesh network configuration.

The subject of the graduation work is the process of modeling the configuration and evaluation of packet transit time in mesh networks.

The object is approaches for the organization and effective interaction of devices in a mesh network and the principles of mesh networks.

The aim of the work is the development of a program for modeling the effective configuration of the network and the evaluation of the packet transit time in it.

The introduction describes the relevance of this work, defines the goal and the objectives necessary to solve the problem, the object and the subject of the study. The first section of the work sets the research task and develops its mathematical model. The second section selects an algorithm for software implementation, develops and tests a program to build an effective mesh network. The conclusion contains the results of the graduation work.

The graduation work contains of an explanatory note on 56 pages, 18 figures, 2 tables, 2 formulas and a list of references, consisting of 20 sources.

Содержание

Введение

В настоящее время информационные технологии стремительно развиваются, что приводит к глобальному распространению вычислительных устройств, используемых в повседневной жизни. С распространением такой концепции, как интернет вещей (IoT), автоматизации промышленных производств, беспилотных автомобилей и других современных направлений развития информационных технологий значимость сетевых технологий для объединения всех устройств существенно возрастает. Одним из развивающихся направлений в этой области являются mesh-сети.

Топология mesh-сети позволяет построить отказоустойчивую и масштабируемую систему, где каждый узел может быть также точкой доступа для других. Из-за таких преимуществ mesh-сети активно используются для построения локальных, персональных и городских беспроводных сетей. В рамках данной работы в качестве устройств для моделирования конфигурации будут рассматриваться микроконтроллеры серии ESP32, так как они выпускаются с поддержкой протокола для создания WLAN mesh-сети, то есть не нуждаются в дополнительной настройке.

Актуальность бакалаврской работы обусловлена быстрым распространением вычислительных устройств, используемых в быту, производстве, военном деле, здравоохранении и других сферах деятельности человека, и необходимостью моделирования наиболее эффективных сетей для их взаимосвязи.

Предметом исследования данной работы является процесс моделирования конфигурации и оценка времени прохождения пакетов в mesh-сетях.

Объектом исследования являются подходы для организации и эффективного взаимодействия устройств в mesh-сети и принципы построения mesh-сетей.

Целью данной работы является разработка программы для моделирования эффективной конфигурации сети и оценки времени прохождения пакетов в ней.

Для достижения поставленной цели необходимо решить следующие задачи: изучить литературу по теме работы; изучить известные технологии для создания mesh-сетей; выбрать способ для организации взаимодействия устройств mesh-сети; разработать программное обеспечение для моделирования mesh-сети в соответствии с выбранным способом; вычислить время прохождения пакетов внутри полученной модели сети; протестировать разработанную программу.

Пояснительная записка включает в себя введение, два раздела, заключение, список литературы и используемых источников.

Во введении описывается актуальность данной работы, определяются цель и необходимые для решения задачи, объект и предмет исследования.

В первом разделе работы описываются различные протоколы для создания mesh-сетей, выбирается платформа для моделирования сети, ставится задача на исследование и разрабатывается ее математическая модель.

Во втором разделе описываются различные алгоритмы для решения поставленной задачи, выбирается один из алгоритмов для реализации программного обеспечения, разрабатывается и тестируется программа для построения эффективной mesh-сети.

Выпускная квалификационная работа содержит пояснительную записку объемом 56 страниц, 18 рисунков, 2 таблицы, 2 формулы и список используемой литературы, состоящий из 20 источников.

1. Постановка задачи на исследование

1.1. Mesh-сети и их место в современном мире

Mesh-сеть (ячеистая топология) – сетевая топология децентрализованной (одноранговой) беспроводной сети, в которой каждый узел также может являться коммутатором для других устройств в сети. Такая особенность сети позволяет значительно увеличить зону покрытия сигнала, которая будет только расширяться с подключением новых узлов. Также такая топология сети обеспечивает отказоустойчивость, так как при отказе одного из узлов просто произойдет передача пакета по другому маршруту. [3, 6, 7]

В настоящее время mesh-сети часто используются для построения локальных, персональных и городских беспроводных сетей. Одним из ключевых преимуществ ячеистых сетей является их гибкость. Поскольку каждое устройство в сети может взаимодействовать с другими устройствами, ячеистые сети могут быть легко расширены или перенастроены по мере необходимости. Кроме того, ячеистые сети могут использоваться для расширения диапазона существующих беспроводных сетей, обеспечивая больший охват в районах, где традиционных беспроводных сетей может быть недостаточно и позволяют оставаться на связи даже при отключении от глобальной сети. Это делает их наиболее подходящими для таких приложений, как «умные дома» и других направлений набирающей популярность концепции интернета вещей, где со временем могут добавляться или удаляться новые устройства. Также mesh-сети все чаще используются в промышленной сфере, соединяя широкий спектр устройств, включая датчики, исполнительные механизмы и другие системы управления, в одну сеть, что обеспечивает автоматизацию и эффективность промышленных процессов.

Ячеистые сети также обеспечивают бóльшую конфиденциальность и безопасность, чем традиционные беспроводные сети. Поскольку каждое устройство в сети может напрямую взаимодействовать с другими

устройствами, нет необходимости передавать данные через центральный концентратор или сервер. Это снижает риск утечки данных или других уязвимостей в системе безопасности.

К недостаткам mesh-сетей можно отнести сложность их устройства. Поскольку каждый узел в сети может взаимодействовать с другими узлами, для обеспечения надлежащего функционирования сети требуется большая степень координации.

Из-за широких возможностей топологии и необходимости выбора наиболее эффективного алгоритма маршрутизации различные протоколы позволяют реализовывать mesh-сеть по-разному. Например, CJDNS (Caleb James DeLisle's Networking Suite) является протоколом сетевого уровня с открытым исходным кодом, эффективным и масштабируемым алгоритмом маршрутизации в больших сетях и используется для обеспечения безопасной и децентрализованной связи через интернет с использованием архитектуры ячеистой сети, в то время как BATMAN-ADV является протоколом канального уровня с алгоритмом маршрутизации, оптимизированным для передачи данных с низкой задержкой в сетях малого и среднего размера, и используется для создания распределенной сети второго уровня. К другим протоколам для организации mesh-сети можно отнести [1, 2, 4, 5]:

– стандарт IEEE 802.11s, основанный на стандарте IEEE 802.11, предназначен для работы с сетями Wi-Fi;

– Zigbee, основанный на стандарте IEEE 802.15.4, позволяет создать беспроводную персональную сеть (WPAN) и применяется в приложениях с низким энергопотреблением и скоростью передачи данных;

– Bluetooth Mesh, который используется для создания крупномасштабных промышленных приложений интернета вещей (в отличие от Zigbee имеет существенно увеличенный радиус действия);

– ESP-WIFI-MESH, основанный на стандарте IEEE 802.11s, предназначен для таких приложений, как домашняя автоматизация, промышленная автоматизация, интеллектуальная энергетика и т.д.

Таким образом, mesh-сети являются довольно мощной и гибкой топологией сети с большим числом преимуществ перед другими существующими топологиями, поэтому являются отличным выбором для широкого спектра сфер жизнедеятельности человека. Быстрое развитие концепции интернета вещей, автоматизации промышленных процессов обеспечивает повсеместное распространение и активное развитие ячеистых сетей.

1.2. Описание задачи

В рамках данной работы для моделирования конфигурации mesh-сети были выбраны микроконтроллеры серии ESP32. ESP32 – это недорогой и маломощный микроконтроллер. Он разработан китайской компанией Espressif Systems, которая специализируется на проектировании и разработке беспроводных систем связи. ESP32 выделяется на фоне других микроконтроллеров малым потреблением энергии, встроенной возможностью организации беспроводной сети и своей ценой, хоть и уступает некоторым другим микроконтроллерам по мощности (Raspberry Pi, STM32, PIC). Одной из его ключевых особенностей является поддержка протокола ESP-WIFI-MESH. Микроконтроллер ESP32 также хорошо настраивается, для разработчиков доступен целый ряд инструментов разработки и программных библиотек. Все это делает микроконтроллеры ESP32 отличным выбором для широкого спектра IoT-приложений, робототехники, промышленных систем управления и т.д.

В качестве протокола для моделирования конфигурации mesh-сети был выбран ESP-WIFI-MESH. Это связано с тем, что микроконтроллеры серии ESP32 поддерживают его сразу и нет необходимости проводить какую-то предварительную настройку микроконтроллеров. Рассмотрим выбранный протокол и накладываемые им ограничения на создания ячеистой сети.

Протокол ESP-WIFI-MESH – это протокол беспроводной ячеистой сети, предназначенный для обеспечения связи между устройствами в ячеистой сети. Протокол основан на стандарте IEEE 802.11s и используется в различных приложениях, включая автоматизацию «умного дома», промышленные системы управления, сельское хозяйство, «умные города» и здравоохранение.

Стандарт IEEE 802.11s является дополнением стандарта IEEE 802.11 и отвечает за организацию беспроводной mesh-сети (WLAN). Нововведения в стандарте не затрагивают физический уровень, на MAC-подуровне внесены изменения для поддержки ячеистых сетей. Кроме того, IEEE 802.11s включает описание маршрутизации пакетов в mesh-сети (например, Mesh Configuration Protocol (MCP), отвечающий за управление сетью и Hybrid Wireless Mesh Protocol (HWMP), отвечающий за маршрутизацию). В результате точки доступа mesh-сети стандарта IEEE 802.11s способны взаимодействовать друг с другом и объединяться в сеть. На рисунке 1 представлен пример ячеистой сети стандарта IEEE 802.11s. [1, 2, 4, 5]

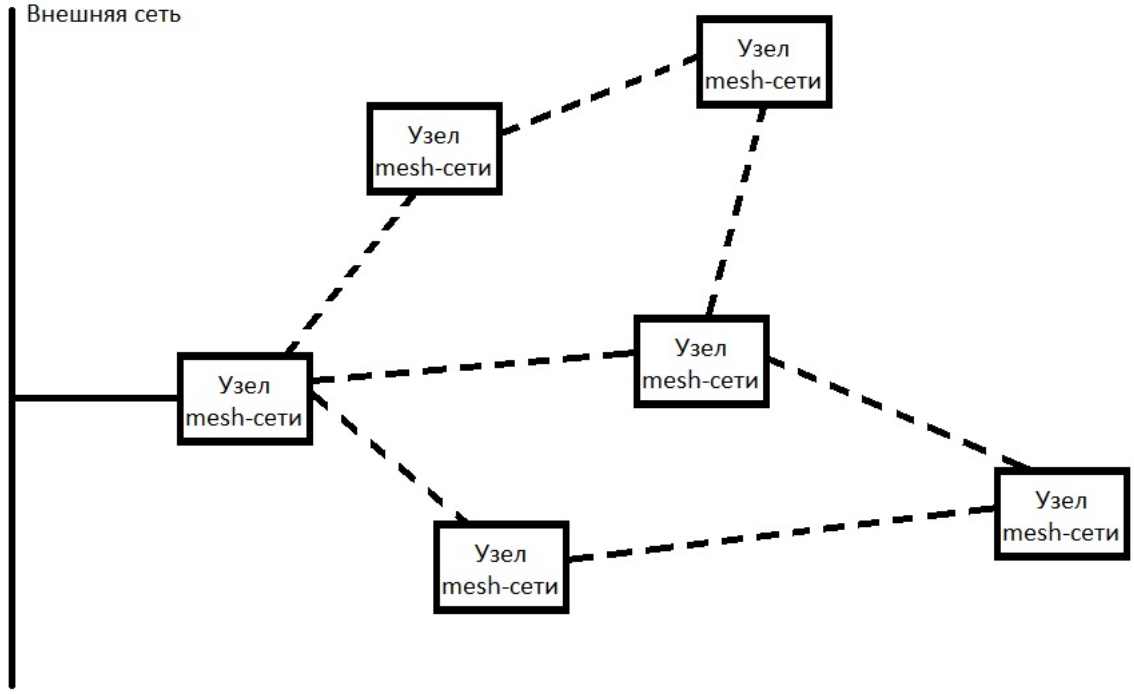


Рисунок 1 – Пример ячеистой сети стандарта IEEE 802.11s

ESP-WIFI-MESH, в свою очередь, позволяет организовать mesh-сеть в виде древовидной структуры. То есть, сеть имеет один корневой узел, промежуточные и конечные узлы. Каждый промежуточный узел имеет один родительский узел и некоторое число дочерних узлов. Для передачи данных по сети протокол использует пакеты ESP-WIFI-MESH, которые содержатся в теле кадра Wi-Fi. Каждый пакет ESP-WIFI-MESH содержит заголовок с MAC-адресами узлов и различные опции пакета. На рисунке 2 представлен один кадр Wi-Fi и место пакета ESP-WIFI-MESH в нем.

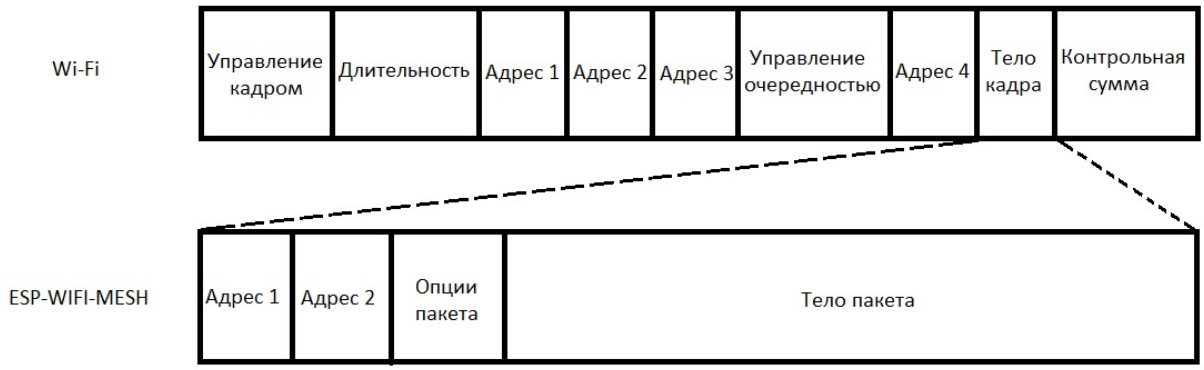


Рисунок 2 – Один кадр Wi-Fi и пакет ESP-WIFI-MESH

Корневой узел устанавливает соединение с маршрутизатором по обычной технологии Wi-Fi. При этом, так как mesh-сеть децентрализована и может быть развернута без доступа к интернету, то этого соединения может не быть и все узлы могут обмениваться информацией только друг с другом, и на построение сети это не влияет. В любом случае протокол строит древовидную структуру и может работать даже без наличия доступа к внешней сети.

Так как масштабируемость – одно из преимуществ mesh-сетей и они расширяются с добавлением новых устройств, очевидно, что один узел не всегда может организовать соединение со всеми другими узлами в сети, например, если некоторые узлы слишком далеко находятся от других. Дерево должно быть построено таким образом, чтобы узлы, для которых нет

возможности организовать прямое соединение, получили соединение через другие узлы. Для этого необходимо, чтобы хотя бы один узел имел возможность установить соединение с каждым из отдаленных узлов. На рисунке 3 представлен пример ячеистой сети протокола ESP-WIFI-MESH.

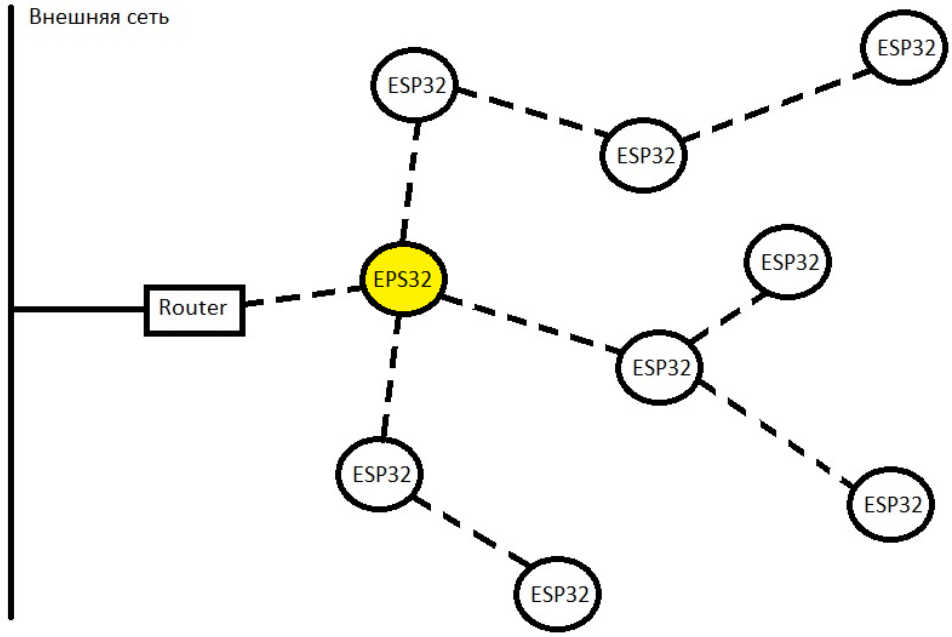


Рисунок 3 – Пример ячеистой сети протокола ESP-WIFI-MESH

При этом количество уровней и число узлов на каждом уровне в протоколе могут быть настроены по желанию администратора сети. Однако, учитывая, что ESP32 обладает достаточно малыми запасами памяти, число дочерних узлов может быть достаточно сильно ограничено. Кроме того, по умолчанию протокол ESP-WIFI-MESH строит ячеистую сеть, выбирая узел для присоединения с самым сильным сигналом, что может привести к построению малоэффективной сети. Именно поэтому стоит задача в наиболее эффективном построении mesh-сети с заданным числом дочерних узлов и минимальным временем прохождения пакетов в сети.

1.3. Аналоги существующих программных решений для моделирования mesh-сетей

Существует несколько популярных решений для моделирования сетей, таких как NS-3, OMNeT++, NetSim и других. Эти программы используются для моделирования различных типов сетей, включая проводные, беспроводные и мобильные сети.

NS-3 – это популярное программное обеспечение для моделирования сетей с открытым исходным кодом, которое широко используется в научных кругах и промышленности. NS-3 написан на C++ и предоставляет интерфейс Python для создания сценариев моделирования. Хотя программу нельзя назвать простой в настройке и использовании и она не имеет встроенного графического интерфейса, NS-3 обладает высокой расширяемостью и предоставляет широкие возможности, что делает ее хорошим выбором среди исследователей и разработчиков.

OMNeT++ – еще одно популярное программное обеспечение для моделирования сетей с открытым исходным кодом, которое широко используется в академических и промышленных кругах. Оно обеспечивает модульную и расширяемую структуру для моделирования различных типов сетей. OMNeT++ написан на C++ и предоставляет графический пользовательский интерфейс для проектирования и запуска симуляций. Он также предоставляет интерфейс сценариев для автоматизации моделирования. OMNeT++ похож на NS-3 и предоставляет практически такие же возможности.

NetSim – это коммерческое программное обеспечение для моделирования сетей, которое широко используется в промышленности. Программа предоставляет полный набор моделей для моделирования различных типов сетей и довольно проста в использовании. NetSim написана на C++ и имеет встроенный графический пользовательский интерфейс для проектирования и запуска симуляций, а также интерфейс сценариев для автоматизации моделирования.

В целом, все описанные продукты предоставляют необходимые возможности для моделирования общих сетей. Они поддерживают различные наборы протоколов, но самые популярные протоколы поддерживает каждая из программ. Для других протоколов и сетей существуют различные специализированные программы, такие как Packet Tracer, созданный компанией Cisco и предназначенный для эмуляции маршрутизаторов Cisco.

Однако, учитывая, что ESP-WIFI-MESH протокол довольно новый и имеет встроенные принципы построения сети, то на сегодняшний день нет специализированной программы для моделирования сетей, построенных на основе протокола ESP-WIFI-MESH. Из рассмотренных программ только NS-3 и OMNeT++ поддерживают необходимый протокол, но и они лишь предоставляют возможности для моделирования и тестирования сети и не позволяют получить наиболее эффективное ее построение. В таблице 1 приведено сравнение описанных выше программных продуктов по интересующим нас критериям.

Таблица 1 – Сравнение программных продуктов

ПО	Встроен-ный GUI	Поддержка ESP-WIFI-MESH	Простота исполь-зования	Возможность моделирования наиболее эффективной сети	Произво-дитель-ность	Бесплатность
NS-3	–	+	–	–	+	+
OMNeT++	+	+	–	–	–	+
NetSim	+	–	+	–	–	–

Как можно заметить, ни один из описанных программных продуктов не предоставляет нужной функциональности, поэтому есть необходимость в разработке собственного программного обеспечения для моделирования сети по протоколу ESP-WIFI-MESH.

1.4. Математическая модель задачи

Mesh-сети, как и любые другие сети, могут быть представлены в виде неориентированного графа, где вершинами являются узлы сети, а ребрами – их соединения. Граф может быть взвешенным и отображать различные показатели соединения между двумя узлами, такие как расстояние между узлами, различные помехи, уровень сигнала, пропускная способность и т.д.

Протокол ESP-WIFI-MESH, как было описано ранее, строит дерево – неориентированный граф без циклов.

Так как нам неважно, есть ли доступ к внешней сети и протокол в любом случае строит древовидную структуру, по задаче на входе имеется граф всех возможных соединений между узлами. Граф должен быть связным, иначе для некоторых узлов никак не получится установить соединение. Однако, очевидно, что граф не обязан быть полным, ведь не всегда каждый узел может установить соединение со всеми другими узлами. По заданному графу необходимо построить дерево с ограничением на степень каждой вершины, отображающее, как следует построить сеть.

В нашем случае граф должен быть взвешенным для построения наиболее эффективной сети. Очевидно, в качестве весов ребер можно использовать любой показатель, в том числе комплексную оценку нескольких показателей. Важно, чтобы для всех ребер были использованы одни показатели для взвешивания. Поэтому при моделировании нами будут использоваться просто различные веса каждого ребра, без какой-либо информации о том, на основе чего было получено данное число и за какую характеристику соединения оно отвечает. На рисунке 4 представлен пример исходного графа всех возможных соединений в сети, а на рисунке 5 изображен пример одного из возможных деревьев, отображающее, как следует построить сеть по выбранному протоколу ESP-WIFI-MESH.

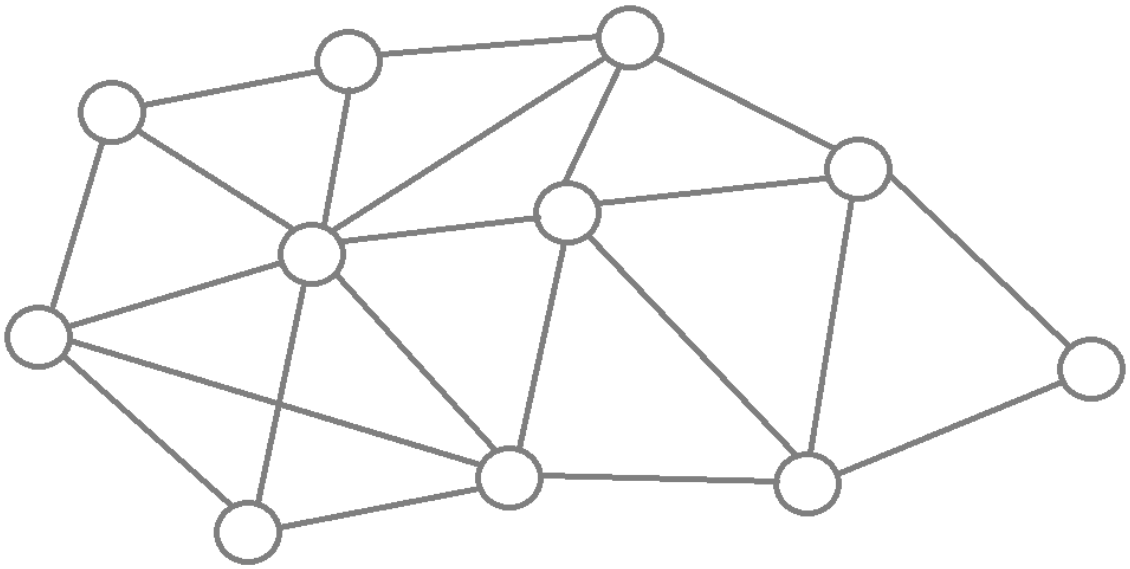


Рисунок 4 – Пример исходного графа всех возможных соединений в сети

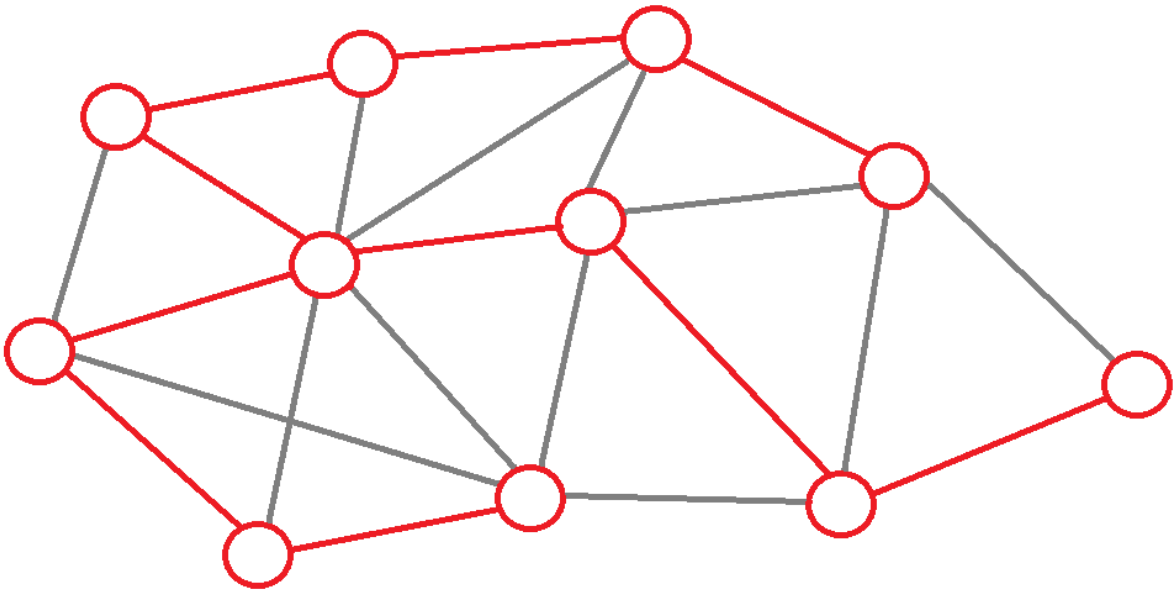


Рисунок 5 – Пример одного дерева, построенного
на основе исходного графа

Для построения наиболее эффективного дерева необходимо, чтобы суммарный вес всех ребер в получившемся дереве был минимален. Очевидно, задача состоит в поиске минимального остовного дерева с ограниченной степенью вершин для заданного графа.

Выводы по разделу 1

Таким образом, mesh-сети – это мощный и гибкий тип беспроводной сети, который предлагает множество преимуществ по сравнению с традиционными беспроводными сетями. Хотя они сопряжены с некоторыми проблемами, включая сложность и возможность создания помех, их гибкость, отказоустойчивость и безопасность делают их хорошо подходящими для широкого спектра применений. Поскольку мир становится все более взаимосвязанным с помощью интернета вещей, ячеистые сети, вероятно, будут играть все более важную роль в обеспечении связи между устройствами и повышении эффективности и безопасности наших домов, предприятий и сообществ. В ходе работы над первым разделом бакалаврской работы были изучены различные протоколы создания ячеистых сетей, была выбрана платформа для моделирования сети и ее особенности, поставлена задача на исследование и построена ее математическая модель.

2. Проектирование и разработка программного обеспечения

2.1. Выбор алгоритма решения задачи

Задача поиска минимального остовного дерева (MST) – это хорошо известная задача в теории графов, которая заключается в нахождении дерева, содержащего все вершины взвешенного неориентированного графа и имеющего минимально возможный общий вес. Существует несколько наиболее известных алгоритмов решения задачи MST (такие, как алгоритмы Прима, Крускала и т.д.), которые довольно просто и эффективно позволяют решить задачу. Задача поиска минимального остовного дерева с ограничением по степени вершин (degree-constrained minimum spanning tree, DCMST) – это вариант задачи MST, в котором вершины дерева имеют ограничения по степени. Другими словами, каждая вершина в дереве должна иметь степень, которая меньше или равна заданному значению. [13]

Задача DCMST возникает в различных приложениях, таких как проектирование сетей, планирование транспорта и размещение объектов. Задача довольно сложна для реализации. Она относится к определенному классу задач, называемых NP-трудными, для которых не существует известного алгоритма полиномиального времени, который может решить эту задачу точно для всех случаев. Поэтому задача по сути является задачей оптимизации, где необходимо минимизировать целевую функцию – суммарный вес остовного дерева, в котором накладываются ограничения на степени вершин. Хотя DCMST и является довольно популярной и активно обсуждаемой (довольно часто авторы публикуют модифицированные популярные или новые алгоритмы, которые более эффективно справляются с решением задачи, например, A. Volgenant [19] в своей работе описывает метод множителей Лагранжа для решения задачи DCMST, а Bau Y., Ho C.K., Ewe H.T. [8] в своей работе сравнивают алгоритм оптимизации муравьиной колонии с другими), отсутствие эффективных алгоритмов решения

ограничивает число различных источников и исследований по теме задачи, особенно в русскоязычном сообществе. [8–12, 15–20]

Несмотря на всю сложность задачи, существуют некоторые алгоритмы (в основном алгоритмы оптимизации), способные получить искомое решение или аппроксимировать его. К таким алгоритмам можно отнести: различные генетические алгоритмы, оптимизация муравьиной колонии, имитационный отжиг, табу-поиск, жадный алгоритм и др. Кроме того, можно использовать существующие алгоритмы для решения задачи MST, модифицированные для работы с DCMST путем какой-либо пост-обработки. Так как не существует эффективного алгоритма решения задачи, создаются различные гибридные методы, совмещающие несколько алгоритмов сразу, или модифицированные методы, которые улучшают какой-либо аспект в существующем алгоритме. Учитывая специфичность задачи, очевидно, что большинство существующих алгоритмов для решения задачи DCMST являются эвристическими. Рассмотрим некоторые существующие алгоритмы для решения поставленной задачи.

Жадный алгоритм решения задачи DCMST – эвристический алгоритм, который строит решение путем итеративного добавления ребер в дерево жадным способом. Сначала выбирается случайная вершина, с которой начинается построения итогового остовного дерева с ограничением на степень вершин. На каждом шаге алгоритм выбирает ребро среди всех ребер, инцидентных уже добавленным в дерево вершинам, которое имеет наименьший вес, не создает цикла в дереве, а также удовлетворяет ограничениям на степень вершин. Этот процесс продолжается до тех пор, пока все узлы не будут включены в дерево. Алгоритм является крайне простым и по сути является модификацией алгоритма Прима. В связи с этим алгоритм является малоэффективным для решения задачи DCMST и может застрять в любом локальном экстремуме, так как не имеет никакого механизма расширения области поиска оптимального решения и выхода из локальных экстремумов. Поэтому эффективность решения задачи также сильно зависит

от начальной вершины, добавленной в остовное дерево. На рисунке 6 представлена блок-схема жадного алгоритма.

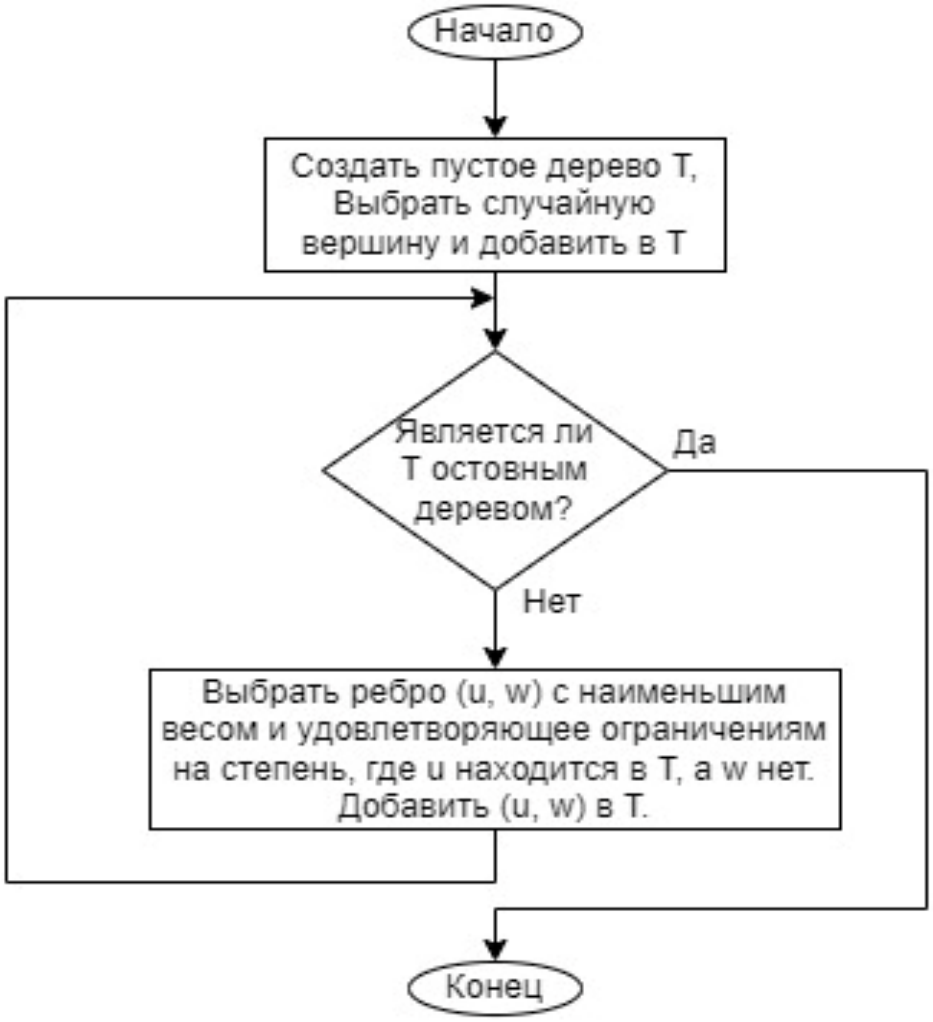


Рисунок 6 – Блок-схема жадного алгоритма для DCMST

Табу-поиск – метаэвристический алгоритм локального поиска, который итеративно улучшает одно решение, внося в него небольшие изменения. Алгоритм ведет список табу, который отслеживает недавно посещенные решения и не позволяет алгоритму возвращаться к ним. Это помогает алгоритму избежать застревания в локальном экстремуме и исследовать большее пространство поиска. Табу-поиск может быть эффективен для быстрого нахождения хороших решений, но он может не справиться с

крупными экземплярами задачи и не найти глобальный экстремум (или делать это очень долго) [20]. На рисунке 7 представлена блок-схема такого алгоритма.

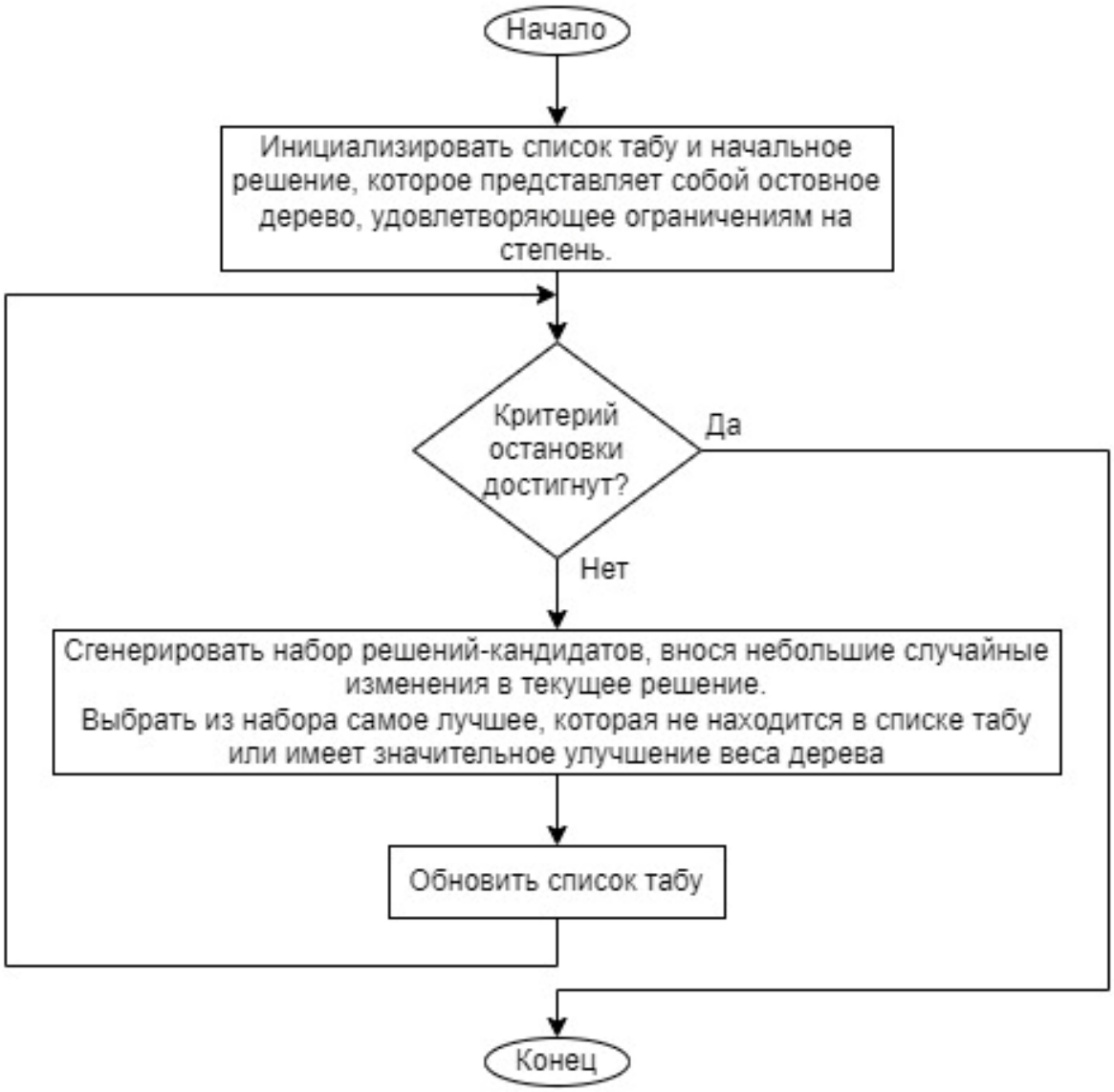


Рисунок 7 – Блок-схема алгоритма табу-поиска для DCMST

Оптимизация муравьиной колонии – метаэвристический алгоритм, который вдохновлен поведением муравьев в поиске кратчайшего пути между их гнездом и источником пищи. Алгоритм использует феромонный след для направления поиска и процедуру локального поиска для улучшения качества решений. [8, 10, 11, 15, 16]

Алгоритм начинается с инициализации набора искусственных муравьев, которые перемещаются в пространстве поиска и строят решения-кандидаты. Муравьи наносят феромон на ребра графа, который привлекает других муравьев следовать тем же путем. Феромонный след обновляется в зависимости от качества решений, найденных муравьями. После каждой итерации применяется процедура локального поиска для улучшения качества решений. Алгоритм завершается при достижении критерия остановки, например, максимального числа итераций или минимального улучшения целевой функции. На рисунке 8 представлена блок-схема алгоритма оптимизации муравьиной колонии.

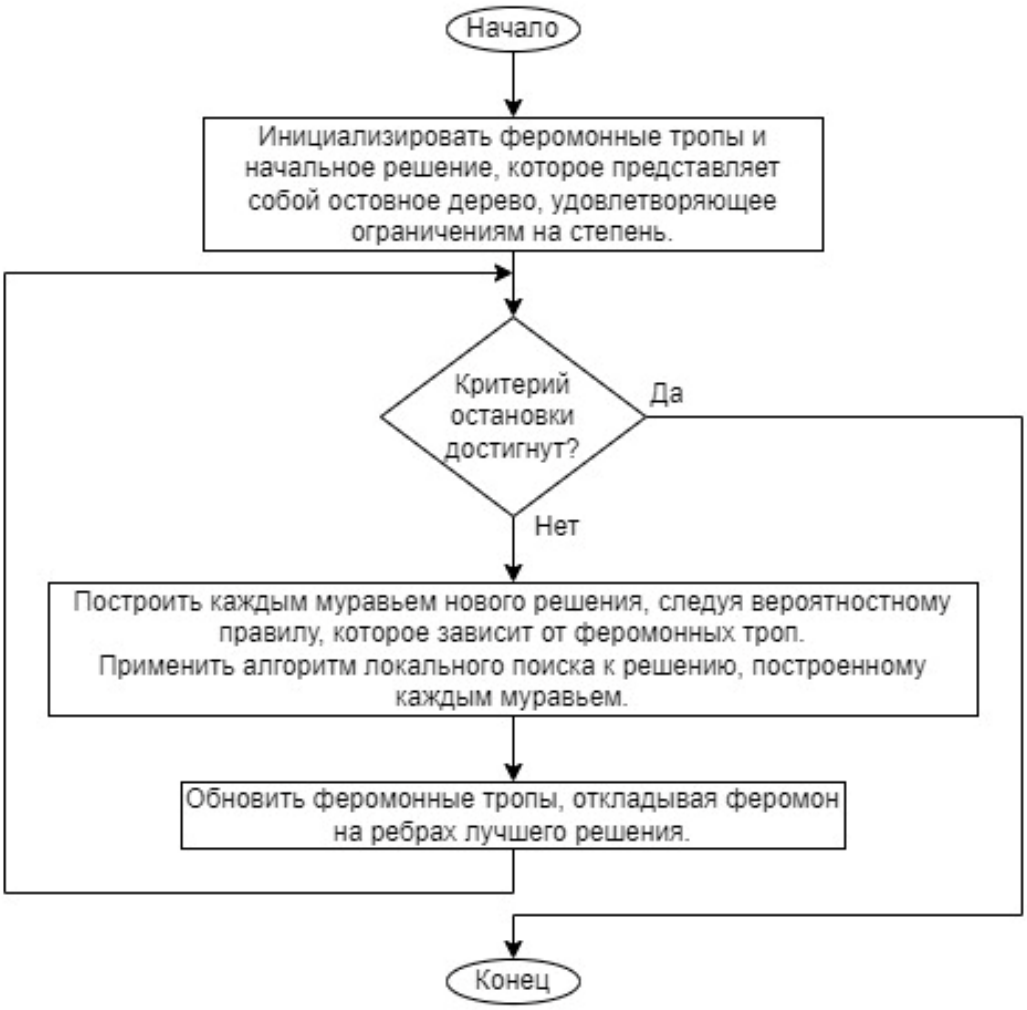


Рисунок 8 – Блок-схема алгоритма оптимизации муравьиной колонии для DCMST

Имитационный отжиг – это метаэвристический алгоритм, который генерирует новые решения путем внесения небольших изменений в текущее решение. Алгоритм начинает с начального решения и итеративно генерирует новые. Изменения вносятся случайным образом, а размер изменений контролируется параметром, называемым размером шага. Алгоритм использует параметр температуры для контроля вероятности принятия худшего решения. На каждой итерации алгоритм генерирует новое решение, которое заменяет текущее, если оно лучше его (например, для задачи на поиск минимального остовного дерева с ограничением на степень вершин лучшим решением будет дерево с минимальным общим весом). Если новое решение хуже текущего, то оно принимается с определенной вероятностью, которая зависит от параметра температуры и разницы в значении объективной функции между текущим и новым решениями. Вероятность принятия худшего решения уменьшается по мере уменьшения температурного параметра.

Температурный параметр постепенно уменьшается с течением времени, чтобы снизить вероятность принятия худших решений. Этот процесс называется отжигом, и он вдохновлен физическим процессом отжига в металлургии, когда металл нагревают, а затем медленно охлаждают для улучшения его свойств.

Алгоритм имитационного отжига продолжает генерировать новые решения и обновлять текущее решение до тех пор, пока не будет достигнут критерий остановки, например, максимальное количество итераций или минимальная температура. Алгоритм позволяет довольно эффективно находить хорошие решения, однако он может оказаться не таким подходящим для больших графов и исследовать недостаточно большое пространство для поиска. На рисунке 9 представлена блок-схема такого алгоритма.

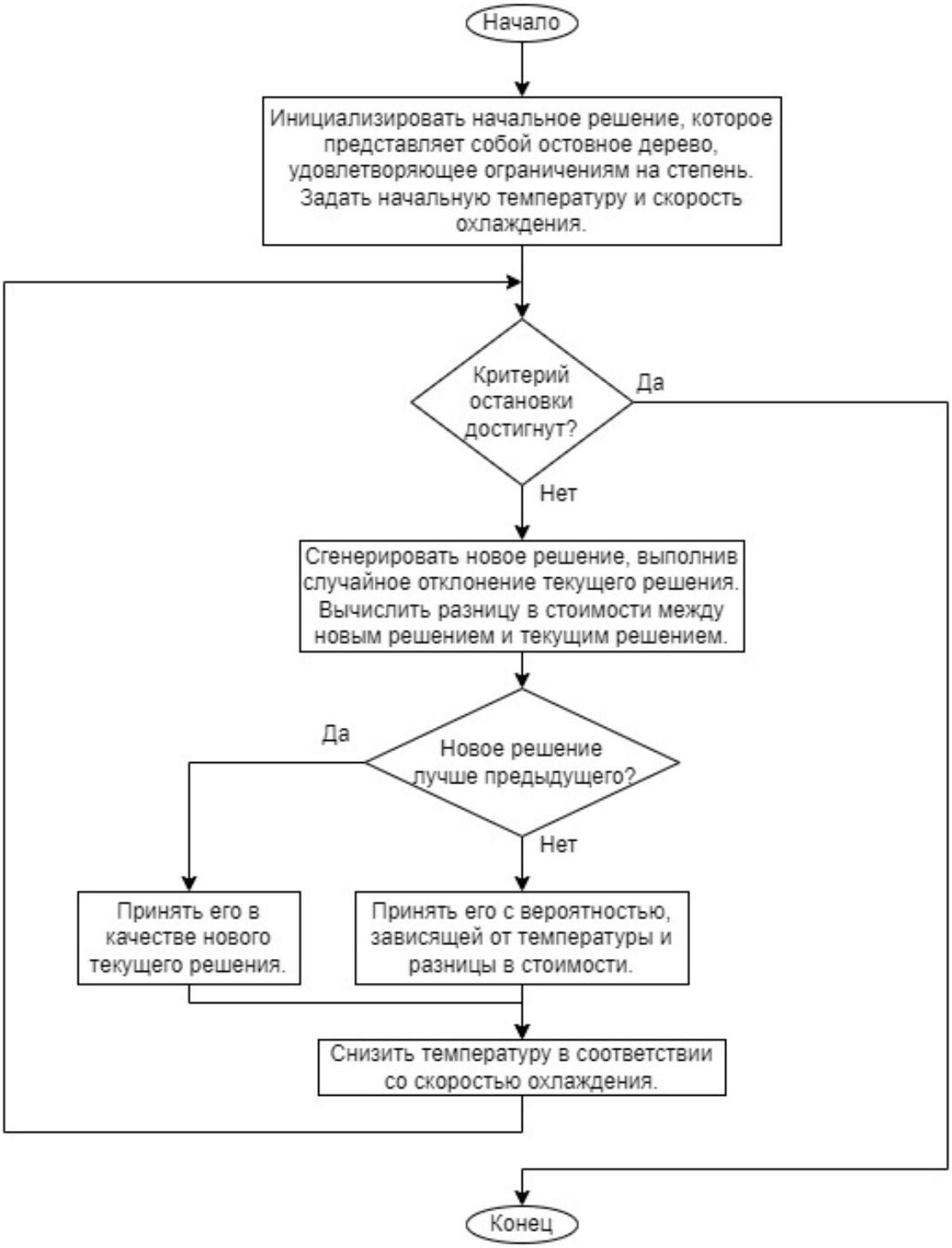


Рисунок 9 – Блок-схема алгоритма имитационного отжига для DCMST

Генетический алгоритм – эвристический алгоритм, который используются для решения задач оптимизации. Он основан на процессе естественного отбора и ищет оптимальное решение задачи, имитируя процесс эволюции. Алгоритм использует методы естественной эволюции:

наследование, мутация, отбор, кроссинговер. При этом различных генетических алгоритмов существует довольно много, которые отличаются различной реализацией описанных методов. [9, 12, 14]

В генетическом алгоритме популяция потенциальных решений генерируется случайным образом. Каждое решение представлено в виде набора параметров (для задачи DCMST каждое решение представляет из себя остовное дерево), которые могут мутировать и рекомбинировать с другими решениями для создания нового потомства. Пригодность каждого решения оценивается по тому, насколько хорошо оно решает поставленную задачу (в нашем случае мы смотрим на вес всех ребер полученного остовного дерева). Затем популяция эволюционирует в течение нескольких поколений, где каждое поколение состоит из фазы отбора, кроссовера и мутации.

На этапе отбора выбирается подмножество популяции, которое становится родителями следующего поколения. Более подходящие решения-кандидаты имеют более высокую вероятность быть выбранными в качестве родителей.

На этапе кроссинговера генетический материал двух родителей (в зависимости от реализации может использоваться другое число родителей) объединяется для создания нового потомства. Оператор кроссинговера может быть одноточечным, двухточечным, равномерным или любым другим алгоритмом объединения. Потомство наследует часть генетического материала от каждого родителя, что позволяет алгоритму эффективно исследовать пространство поиска.

На этапе мутации потомство случайным образом мутирует, чтобы внести новый генетический материал в популяцию. Оператор мутации позволяет алгоритму выходить из локальных экстремумов и более эффективно исследовать пространство поиска.

Со временем генетический алгоритм сходится к набору оптимальных решений, которые хорошо подходят для решения поставленной задачи. Алгоритм позволяет эффективно и результативно исследовать область поиска,

выходить из локальных экстремумов для поиска глобального, а также работать с более крупными задачами по сравнению со многими другими алгоритмами.

Генетические алгоритмы используются в широком спектре приложений и являются мощным алгоритмом оптимизации, который может быть применен к задаче DCMST. При этом реализация генетических алгоритмов может очень сильно отличаться. На рисунке 10 представлена блок-схема алгоритма, которая включает этапы, через которые проходят все различные его реализации.

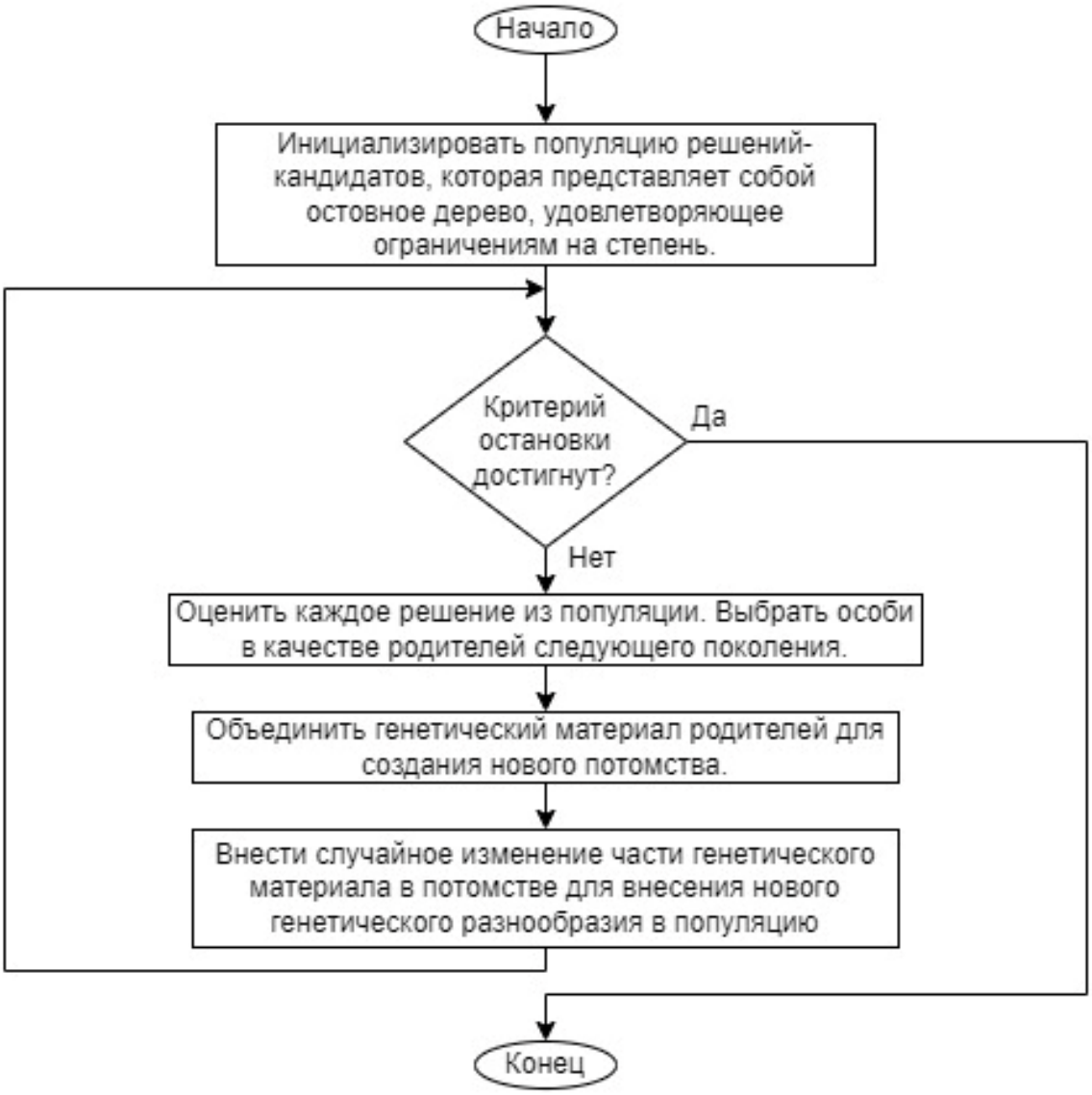


Рисунок 10 – Блок-схема генетического алгоритма для задачи DCMST

Так как генетические алгоритмы позволяют исследовать большую область для поиска оптимального решения и более эффективны и результативны при работе с крупными графами, то в качестве алгоритма для решения задачи был выбран именно генетический алгоритм. Кроме того, большой выбор различных алгоритмов среди всего множества генетических алгоритмов позволяет подобрать нужный алгоритм или даже создать свой собственный.

Среди всех генетических алгоритмов был выбран гибридный устойчивый генетический алгоритм (hybrid steady-state genetic algorithm, HSSGA). HSSGA сочетает в себе устойчивый подход и традиционный генетический алгоритм. При подходе с устойчивым состоянием популяция решений-кандидатов обновляется постепенно, при этом в каждом поколении заменяется лишь несколько особей. Кроме того, HSSGA использует процедуру локального поиска для улучшения качества решений, генерируемых генетическим алгоритмом. Это позволяет выявить и исправить небольшие ошибки или недостатки в решениях, что может привести к значительному улучшению общего качества решений. В целом, алгоритм имеет ряд преимуществ перед другими генетическими алгоритмами, включая улучшенное качество решений, более быструю сходимость, лучшую работу с ограничениями и устойчивость [18].

2.2. Описание выбранного алгоритма

Гибридный генетический алгоритм с устойчивым состоянием (HSSGA) – это тип генетического алгоритма, который позволяет довольно эффективно справляться с задачей DCMST. HSSGA использует комбинацию стационарного подхода, традиционного генетического алгоритма и процедуры локального поиска для генерации высококачественных решений задачи DCMST. Устойчивый алгоритм вместо генерации и замены всей популяции генерирует лишь несколько особей (зачастую даже одну), что позволяет

постепенно улучшать решение. Такой подход помогает поддерживать разнообразие в популяции и предотвращает преждевременную сходимость к неоптимальным решениям. HSSGA также использует локальный поиск, который позволяет улучшить решение и применяется после применения операторов кроссинговера и мутации к отобранным особям в популяции. Опишем принцип работы выбранного алгоритма.

HSSGA начинает работу с генерации начальной популяции решений-кандидатов. Каждое решение-кандидат представляет собой минимальное остовное дерево входного графа, которое удовлетворяет ограничениям на степени вершин. При этом алгоритм генерации может быть разный, при реализации HSSGA была выбрана случайная генерация деревьев.

Устойчивый алгоритм использует турнирный отбор (tournament selection) для выбора двух родительских решений из популяции. Турнирный отбор также может быть реализован по-разному. В общем виде отбор предполагает деление популяции на блоки, выбор чемпионов в каждом блоке и затем выбор лучшего решения из некоторого ограниченного случайного подмножества чемпионов блоков в качестве родителя. На рисунке 11 представлен принцип турнирного отбора одного родителя.

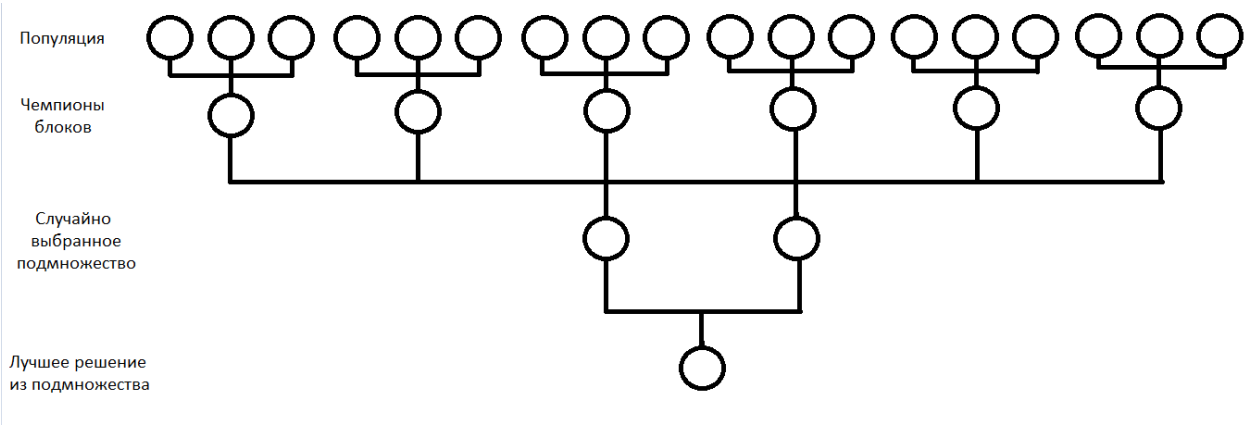


Рисунок 11 – Принцип турнирного отбора одного родителя

Для упрощения этого процесса при турнирном отборе, реализованном в программе, из всей популяции случайным образом выбираются две особи, и

лучшая особь из этого подмножества с большей вероятностью выбирается в качестве родителя. При этом вторая особь тоже может быть выбрана с небольшой вероятностью для расширения области поиска. Для второго родителя процесс отбора повторяется.

HSSGA использует оператор кроссовера для создания нового решения-кандидата из двух родительских решений. При этом оператор старается брать больше ребер от более приспособленного родителя. Так, вероятность взятия ребра от первого родителя может быть получена по формуле

$$\frac{1/F(p_1)}{1/F(p_1)+ 1/F(p_2)},$$

где $F(p_1)$ – вес дерева первого родителя, $F(p_2)$ – вес дерева второго родителя. Формула была приведена в работе J.E Beasley и P.C Chu [9] для выбора более приспособленных особей из популяции без большого ограничения области поиска.

Оператор кроссовера сначала добавляет случайную вершину от первого родителя с вероятностью, описанной выше, или от второго родителя. Затем в цикле оператор ищет случайное ребро, которое соединяет уже добавленную вершину с еще не добавленной вершиной. Это продолжается до тех пор, пока все вершины не будут включены в дерево. При этом, как и с первой вершиной, вероятность выбора каждого ребра от первого родителя определяется по описанной выше формуле. В противном случае ребро выбирается от второго родителя. Разумеется, мы также ищем ребро, которое не нарушает ограничений на степень вершины. Если же ребро не может быть выбрано от родителя, от которого необходимо выбирать следующее ребро (например, по причине нарушения ограничений на степень вершины), то алгоритм переходит к поиску ребра у другого родителя. Если и там нет подходящих ребер, то алгоритм берет случайное ребро из оставшихся.

Генетические алгоритмы используют оператор мутации для внесения небольших изменений в новое решение-кандидат, сгенерированное оператором кроссинговера. Однако для более устойчивого и постепенного

улучшения решения, HSSGA за одну итерацию либо использует оператор кроссовера, либо оператор мутации. Выбор оператора определяется случайным образом с некоторой вероятностью, определенной заранее. Оператор мутации работает путем случайного выбора ребра в решении и замены его другим ребром, которое удовлетворяет ограничениям на степень вершин и позволяет соединить две получившиеся после удаления ребра компоненты связности в одну. Например, при удалении ребра (2, 4) на рисунке 12 есть возможность добавить либо ребро (1, 4), либо ребро (4, 5), чтобы в результате снова получилось остовное дерево, что также отражено в нижней части рисунка 12 (ребра выделены синим цветом).

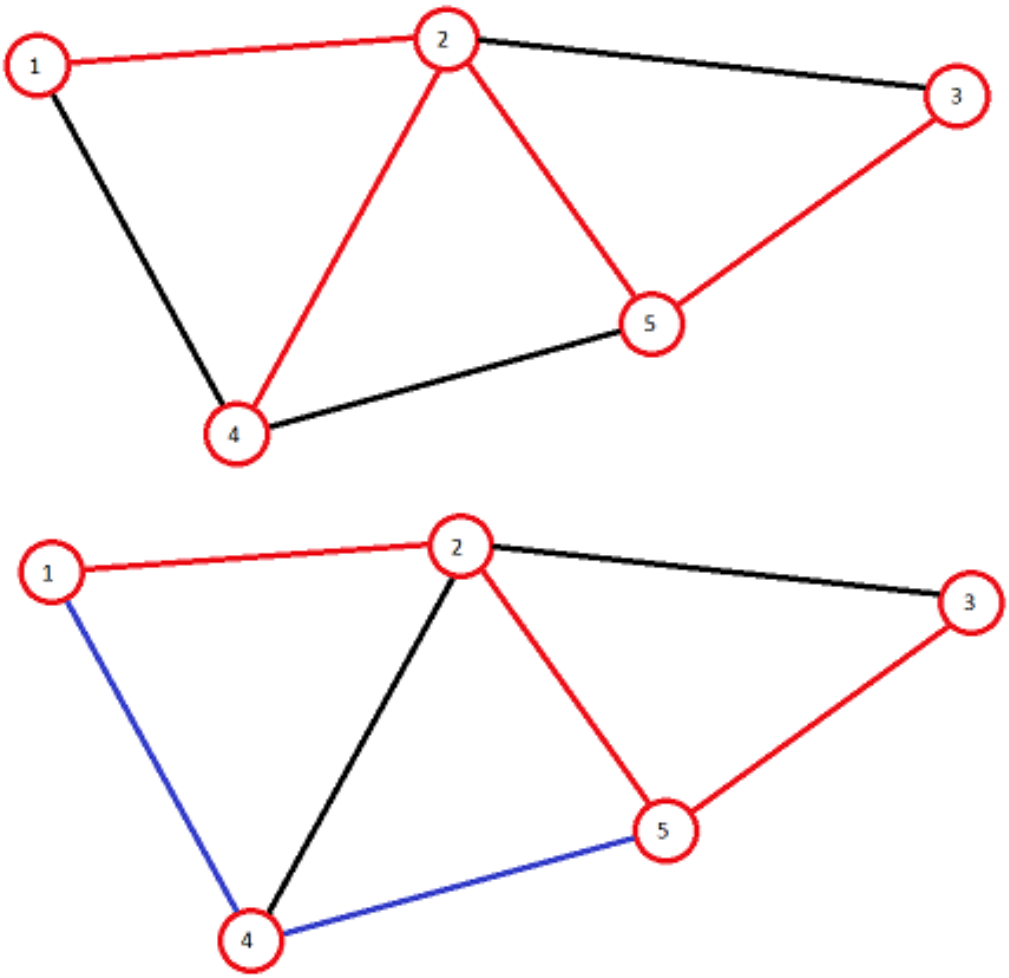


Рисунок 12 – Пример применения оператора мутации
(сверху – до удаления ребра, внизу – возможные варианты замены)

Реализованный HSSGA после операторов кроссовера и мутации использует 2ER (2-edge replacement) и 1ER (1-edge replacement) процедуры локального поиска для улучшения качества нового решения-кандидата. Процедура 2ER локального поиска итеративно для некоторого числа ребер остовного дерева пытается заменить два ребра в текущем решении двумя другими, чтобы проверить, можно ли найти лучшее решение. 2ER просматривает все несмежные ребра для выбранного ребра и, если замена этих двух ребер на два других уменьшит общий вес дерева и все вершины все еще будут соответствовать ограничениям на степень вершин, заменяет их. На рисунке 13 изображен пример применения процедуры 2ER локального поиска, где для несмежных ребер (1, 2) и (3, 4) может быть произведена замена на ребра (1, 4) и (2, 3), если от этого общий вес дерева уменьшится.

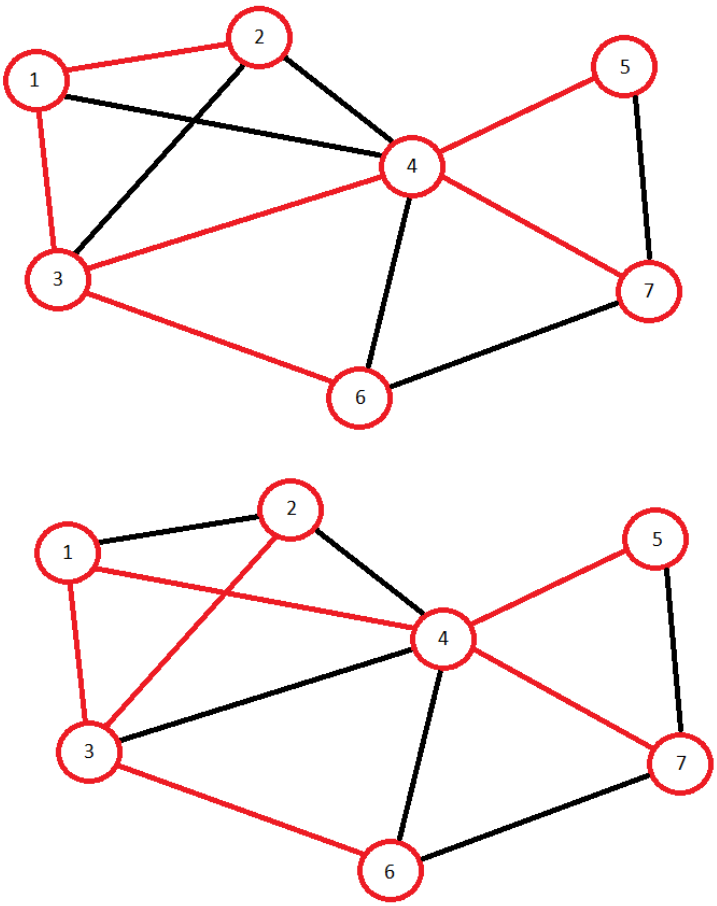


Рисунок 13 – Пример применения процедуры 2ER
(сверху – до замены, внизу – после замены)

Процедура 1ER также итеративно просто пытается заменить одно ребро в текущем решении другим. При этом процедура применяется в два этапа: сначала мы пытаемся заменить ребро, у которого степень хотя бы одной инцидентной вершины уже равна максимальной; а затем проверяются все ребра. Два этапа сделаны для того, чтобы после уменьшения степени у вершины с максимально допустимой степенью существовала вероятность того, что появилась новая возможность замены ребра на другое с меньшим весом. На рисунке 14 представлен пример работы процедуры 1ER, где возможна замена ребра (3, 4) на ребро (6, 7).

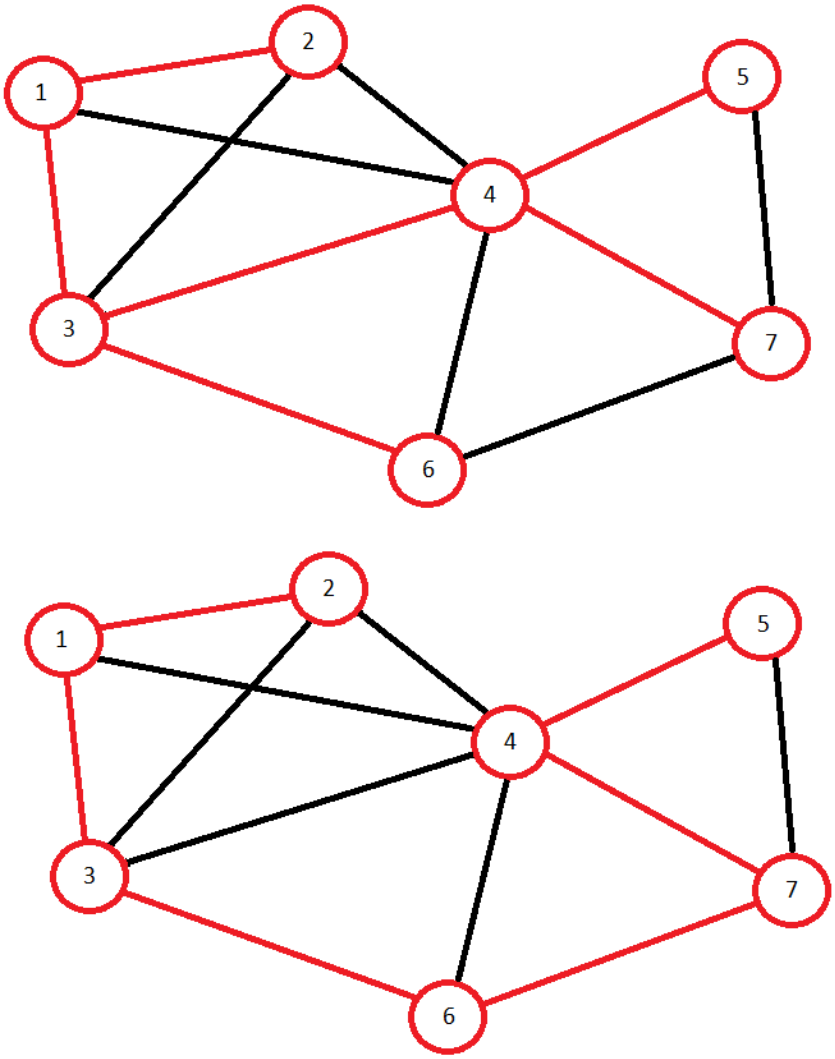


Рисунок 14 – Пример применения процедуры 1ER
(сверху – до замены, внизу – после замены)

Стоит отметить, что хоть и 1ER, и 2ER работают итеративно, количество итераций должно быть ограничено, чтобы ускорить работу алгоритма. Обычно используется количество итераций, равное отношению числа вершин к двум. Однако при реализации просто использовалось три итерации, так как число вершин также может быть довольно велико.

Таким образом, локальный поиск позволяет улучшить решение, которое далее будет использовано в качестве потомства для следующего поколения. Однако процедура локального поиска применяется лишь к решениям, которые либо лучше сгенерированного решения, которое имеет наименьший вес дерева к текущей итерации, либо немного хуже, но при этом имеет большое число различных ребер с ним. Локальный поиск не применяется к решениям, близким к лучшему сгенерированному решению до текущей итерации, чтобы обеспечить разнообразие в популяции, на основе которой потом может быть сгенерировано наилучшее решение. То есть процедура локального поиска применяется только к решениям, для которых следующее условие истинно:

$$\frac{F(T^{gb})}{F(T^{c})} + \alpha \cdot dis(T^{gb}, T^{c}) > 1,$$

где $F(T^{gb})$, $F(T^{c})$ – веса лучшего сгенерированного дерева до текущей итерации и сгенерированное решение на текущей итерации соответственно, α – параметр альфа, определенный заранее, $dis(T^{gb}, T^{c})$ – доля ребер, не совпадающих у двух решений.

HSSGA использует стратегию замены для обновления популяции решений-кандидатов. В стратегии замены новое решение-кандидат, сгенерированное операторами кроссинговера, мутации и локального поиска, проверяется на уникальность во всей популяции. Замена происходит только в том случае, если решение уникально. При этом заменяется либо лучшее сгенерированное решение, если вес текущего дерева меньше, либо случайно выбранное решение, вес дерева которого больше среднего веса во всей популяции. Кроме того, алгоритм подсчитывает число итераций, на которых не происходило замены лучшего решения. Если число таких итераций

слишком велико, то алгоритм выбирает некоторое число особей популяции, изменяет их с помощью оператора мутации и, если новое решение уникально, заменяет старое решение в популяции. Это сделано для того, чтобы избежать застревание алгоритма в локальном экстремуме.

HSSGA завершается при достижении критерия остановки, то есть максимального числа итераций.

Таким образом, реализованный алгоритм позволяет успешно справиться с поставленной задачей (найти оптимальное решение) за разумные временные и вычислительные затраты.

2.3. Формализация требований к программному обеспечению и реализация алгоритма

В ходе бакалаврской работы было разработано программное обеспечение на языке Python. Этот высокоуровневый язык программирования отличается своей простотой (причем и изучения, и использования), обширными функциональными возможностями (для него существует огромное число различных библиотек и фреймворков), большим и активным сообществом, кроссплатформенной совместимостью, универсальностью (может быть использован для широкого спектра приложений). Поэтому Python быстро развивается и набирает популярность, что и делает его отличным выбором для проекта.

При этом входные данные для программы задаются с помощью конфигурационного файла в формате YAML. Формат был выбран из-за его простоты изучения и использования, удобства чтения для человека (даже не знакомого с данным форматом), гибкости (формат поддерживает широкий спектр структур данных). Кроме того, любой файл формата YAML может быть очень просто прочитан встроенной библиотекой Python. На рисунке 15 приведен пример чтения данных из конфигурационного файла средствами

языка Python. Функция load читает данные из файла и возвращает словарь с данными из конфигурационного файла.

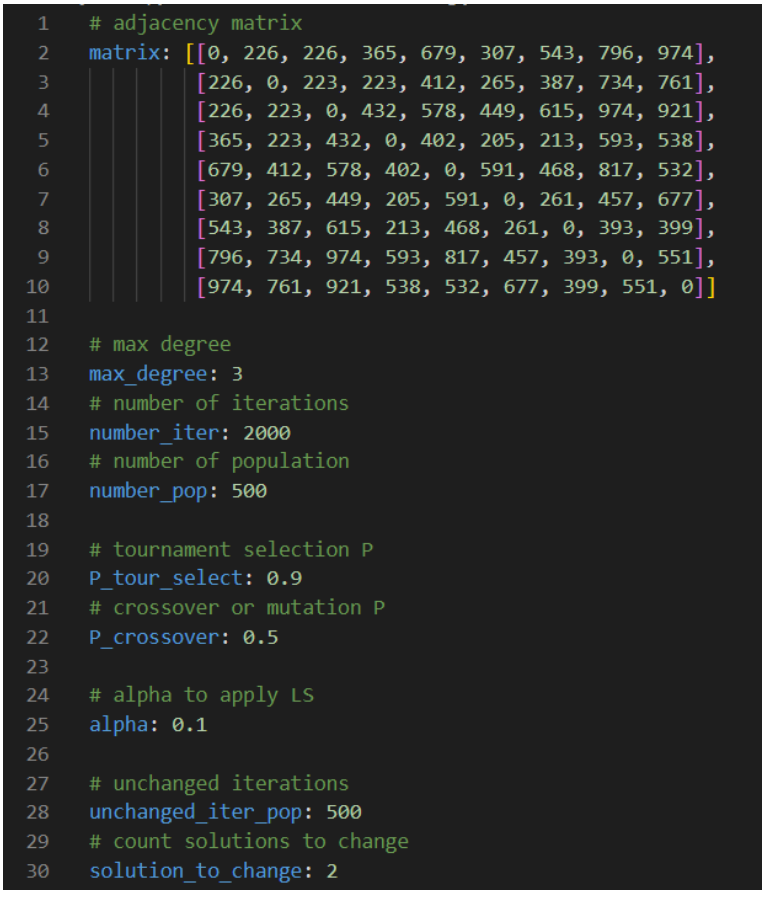
```
5   from yaml import load, Loader
6
7   if __name__ == "__main__":
8       # Read configuration
9       with open('config.yaml', 'r') as f:
10          data = load(f, Loader=Loader)
```

Рисунок 15 – Пример чтения данных из конфигурационного файла

Для предоставления наибольшей гибкости для пользователя программного продукта необходимо, чтобы с помощью конфигурационного файла была возможность задать как можно больше параметров для алгоритма. Поэтому была разработана программа, которая позволяет пользователю ввести следующие параметры: матрица смежности (matrix), которая представляет сам граф возможных соединений в сети; максимально возможную степень вершин в дереве (max_degree); число итераций (то есть число поколений) (number_iter); количество решений в популяции (number_pop); вероятность выбора более приспособленного решения в турнирном отборе в качестве родителя (P_tour_select); вероятность применения оператора кроссовера (в противном случае будет применена операция мутации) (P_crossover); альфа-параметр (alpha) для определения, будет ли применяться локальный поиск к сгенерированному решению; максимальное число итераций (unchanged_iter_pop), при которых не происходило улучшения лучшего решения, для применения операции мутации к случайным решениям; а также число решений (solution_to_change), к которым необходимо применить оператор мутации при достижении максимального числа итерации без изменений; корневой узел (root), с которого начинается построение дерева.

Обязательными при этом являются только матрица смежности и максимальная степень вершин, так как без них теряется смысл всего моделирования. При отсутствии обязательных параметров программа выведет ошибку и завершит свою работу. Остальные параметры можно не указывать. В таком случае они будут заменены значениями по умолчанию, позволяющими построить хотя бы какую-то модель. Однако, очевидно, для получения наиболее эффективной модели, возможно, потребуется регулирование и других параметров.

Конфигурационный файл должен иметь название config.yaml и находиться в директории с файлами .py. На рисунке 16 представлен пример конфигурационного файла со всеми возможными параметрами.



```yaml
# adjacency matrix
matrix: [[0, 226, 226, 365, 679, 307, 543, 796, 974],
         [226, 0, 223, 223, 412, 265, 387, 734, 761],
         [226, 223, 0, 432, 578, 449, 615, 974, 921],
         [365, 223, 432, 0, 402, 205, 213, 593, 538],
         [679, 412, 578, 402, 0, 591, 468, 817, 532],
         [307, 265, 449, 205, 591, 0, 261, 457, 677],
         [543, 387, 615, 213, 468, 261, 0, 393, 399],
         [796, 734, 974, 593, 817, 457, 393, 0, 551],
         [974, 761, 921, 538, 532, 677, 399, 551, 0]]

# max degree
max_degree: 3
# number of iterations
number_iter: 2000
# number of population
number_pop: 500

# tournament selection P
P_tour_select: 0.9
# crossover or mutation P
P_crossover: 0.5

# alpha to apply LS
alpha: 0.1

# unchanged iterations
unchanged_iter_pop: 500
# count solutions to change
solution_to_change: 2
```

Рисунок 16 – Пример конфигурационного файла

Сам алгоритм HSSGA подключается к запускаемой программе в качестве модуля. В главной программе запускается функция hssga, которая

выполняет все необходимые вычисления и возвращает итоговый результат. В функцию передаются все параметры из конфигурационного файла. Функция выполняет вычисления в соответствии с описанным ранее алгоритмом, то есть генерирует популяцию, проходит переданное параметром число итераций и возвращает наилучшее решение в виде класса SpanningTree. Класс просто содержит набор ребер дерева, степень каждой вершины и общий вес.

Кроме того, при реализации потребовалась функция определения, является ли граф деревом. Функция применяется при локальном поиске, так как там при замене ребер граф может перестать являться деревом.

Для генерации одного решения из популяции, оператора кроссовера, оператора мутации, процедур 2ER и 1ER также были реализованы отдельные функции.

Кроме того, в этом модуле реализована функция оценки времени прохождения пакетов в mesh-сети, которая ищет самый короткий путь от одной вершины до всех остальных вершин полученного остовного дерева (для каждой вершины до любой другой существует лишь один путь). В зависимости от показателя, который отображает какую-либо характеристику или комплексную оценку нескольких характеристик, очевидно, что функция сможет оценивать не только время прохождения пакетов в сети, а как раз для каждого узла в сети можно вычислить минимальное значение выбранного показателя от любого узла до всех других.

Стоит уточнить, что выбор корневой вершины дерева по умолчанию также зависит от результатов, полученных после применения функции поиска пути от вершины до всех других вершин. По умолчанию в качестве корневого узла сети берется узел, который ближе всего находится ко всем остальным. То есть имеет минимальную сумму весов до других вершин.

В основной программе после вызова алгоритма с помощью библиотеки pydot, которая предоставляет простой интерфейс для создания, манипулирования и визуализации графиков с помощью программного обеспечения Graphviz, происходит построение итогового минимального

остовного дерева с ограничением на степень вершин и запись в файл model.png. Кроме того, на экран выводится время прохождения пакетов в сети (минимальное значение выбранного в качестве весов ребер показателя) от каждой вершины до всех других узлов.

В разработанную программу также было добавлено измерение времени работы алгоритма HSSGA с помощью библиотеки time.

Разработанная программа позволяет администратору сети, внеся необходимые данные в конфигурационный файл, довольно быстро получить модель эффективной mesh-сети. Кроме того, небольшие изменения в программе позволят использовать ее для автоматизации конфигурирования сети. То есть размещение модифицированной программы на маршрутизатор (или, например, корневой узел ESP32, или другую вычислительную машину, которая может выступать в роли сервера) позволит использовать его для организации сети. Устройство ESP32 посылает запрос на данный маршрутизатор, отправляя данные о своем окружении. Маршрутизатор, используя реализованный алгоритм, определяет, как следует организовать сеть и отсылает ESP32 и другим устройствам в сети результат.

Таким образом, в ходе работы было реализовано программное обеспечение на языке Python для моделирования наиболее эффективной mesh-сети на основе входного графа возможных соединений в сети и других параметров. Теперь можно перейти к тестированию разработанной программы.

2.4. Тестирование разработанного программного обеспечения

Тестирование является крайне важной частью разработки программного обеспечения, поэтому ему следует уделять особое внимание. Оно позволяет обеспечить качество и надежность продукта, найти и исправить большое число возможных ошибок, а также увидеть работоспособность алгоритма.

Тестирование разработанного программного обеспечения проводилось в несколько этапов: во-первых, было проведено тестирование на предмет возможных ошибок в программе; во-вторых, – тестирование корректности работы алгоритма; в-третьих, – тестирование на временные изменения работы программы при различном наборе вершин графа.

При проведении первого этапа тестирования были выполнены тесты на возможные ошибочные действия пользователя, которые заключаются в различных ошибках в конфигурационном файле. Например, если пользователь не указал матрицу смежности или допустил в ней ошибки (матрица смежности неориентированного графа должна быть симметричной относительно главной диагонали). В таком случае пользователь должен получить сообщение. Пример запуска программы без указания матрицы смежности в конфигурационном файле приведен на рисунке 17. Также на данном этапе необходимо проверить корректное поведение программы при отсутствии необязательных параметров в конфигурационном файле. В результате программа успешно прошла все тесты на ошибочные действия пользователя.

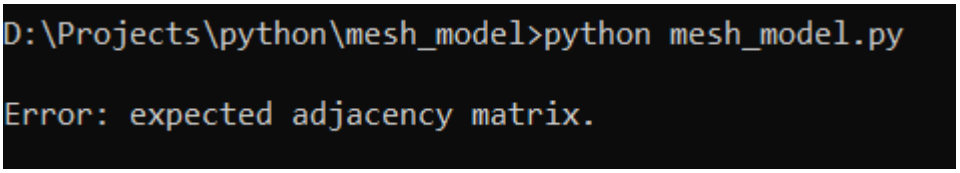


Рисунок 17 – Пример запуска программы без матрицы смежности

На втором этапе тестирования были взяты несколько простых примеров графов возможных соединений в сети и аналитически подсчитано минимальное остовное дерево с ограничением на степень вершин для каждого входного примера. Затем была запущена программа для проверки корректности ее работоспособности. В результате тестирования программа позволяла получить наиболее эффективное решение (разумеется, при необходимом числе итераций), то есть реализованный алгоритм успешно справляется с поставленной задачей.

Наконец, было проведено тестирование на временные изменения работы программы при различном наборе вершин графа. Хотя матрица смежности и является удобным способом представления графа возможных соединений, для слишком большого числа вершин ее описание в конфигурационном файле может занять слишком много времени. Поэтому были добавлены дополнительные записи в конфигурационный файл, которые позволяют включить тестовый режим и указать количество вершин в графе для генерации, а также функция в программе для генерации случайного полного связного графа на основе количества вершин. В таблице 2 представлено время выполнения алгоритма для одинакового количества итераций и различного количества вершин в графе. Для тестирования использовалось 2000 итераций, то есть значение по умолчанию в программе. Это число было выбрано для того, чтобы все различные аспекты алгоритма (такие как турнирный отбор от второго родителя, применение мутации или кроссовера, случайная мутация решения при большом количестве итераций без изменений) имели достаточную высокую вероятность применения хотя бы по одному разу.

Таблица 2 – Время выполнения алгоритма для различного количества вершин в графе

Количество вершин	Время работы алгоритма (с)
10	0,77
20	1,69
50	15,47
100	171,49
150	945,55
200	3757,6
300	23761,54

На рисунке 18 представлен график зависимости времени выполнения от количества вершин во входном графе возможных соединений.

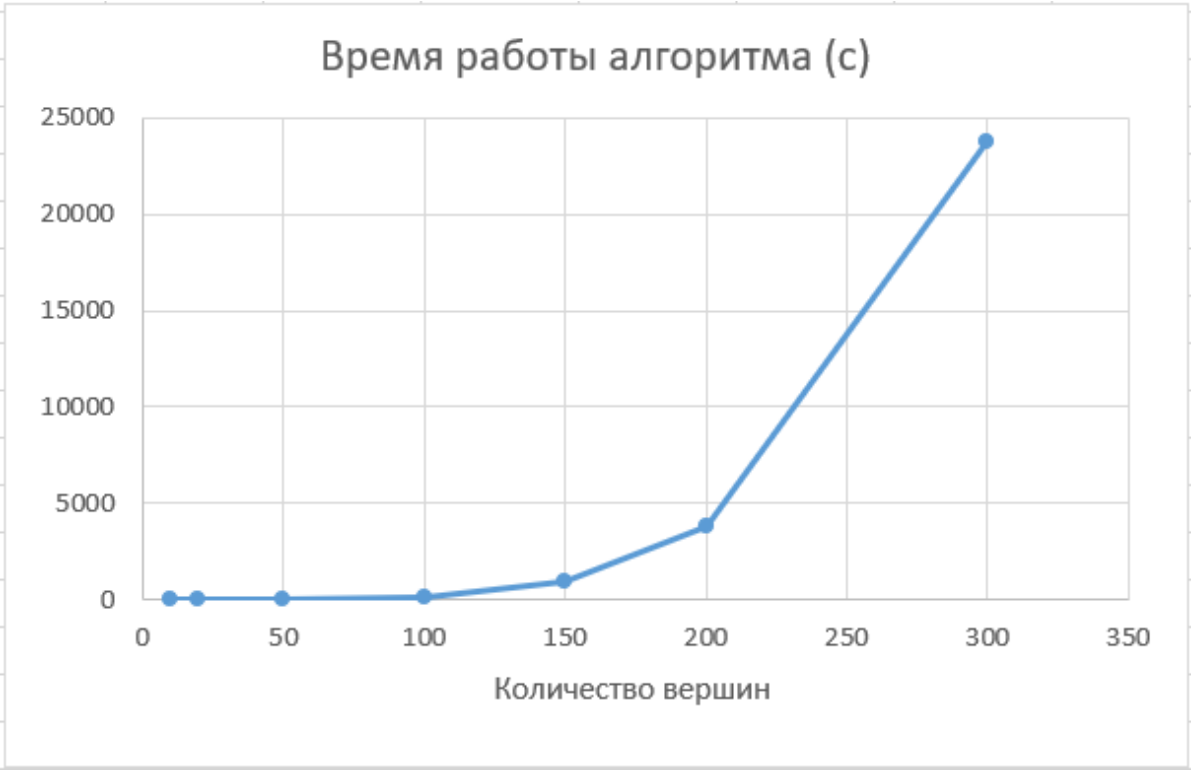


Рисунок 18 – График зависимости времени выполнения от количества вершин во входном графе возможных соединений

Таким образом, время работы алгоритма сильно зависит от количества вершин в графе. Кроме того, при большом количестве вершин для получения эффективного дерева потребуется значительно большее число итераций, что увеличит итоговое время работы алгоритма, несмотря на то, что число итераций можно увеличивать без особого ущерба по времени (линейная зависимость). Однако, с необходимым в повседневной жизни количеством узлов алгоритм справляется довольно быстро.

В результате проведения тестирования алгоритм справился с поставленной задачей и показал достаточно хорошие результаты.

Выводы по разделу 2

Таким образом, в результате проделанной работы над вторым разделом была изучена поставленная сложная оптимизационная задача поиска минимального остовного дерева с ограничением на степень вершин, были изучены различные популярные алгоритмы для решения задачи, такие как табу-поиск, имитационный отжиг, оптимизация муравьиной колонии, генетические алгоритмы и т.д., из которых был выбран и подробно описан алгоритм HSSGA из-за его способности эффективно находить глобальный экстремум за разумное время. Кроме того, было реализовано и протестировано программное обеспечение на языке программирования Python для моделирования наиболее эффективной mesh-сети на основе входного графа возможных соединений.

Заключение

В настоящее время информационные технологии быстро развиваются, что способствует распространению компьютерных устройств в повседневной жизни. Инновационные направления информационных технологий обеспечивают необходимость развития сетевых технологий, одной из которых и являются mesh-сети. Они обладают большим числом преимуществ (отказоустойчивость, масштабируемость, гибкость, безопасность и т.д.) перед другими топологиями сети. При этом сложность организации mesh-сети и отсутствие готовых решений для этого создают необходимость разработки собственного программного обеспечения для моделирования конфигурации сети, что и являлось целью данной работы.

На первом этапе бакалаврской работы были изучены различные протоколы для создания mesh-сетей, выбрана платформа для моделирования сети, поставлена задача на исследование и разработана ее математическая модель.

На втором этапе была рассмотрена поставленная задача, описаны различные алгоритмы для ее решения, был выбран и подробно изучен один из алгоритмов для реализации программного обеспечения, была реализована и протестирована программа для построения эффективной mesh-сети.

Таким образом, в ходе исследования были решены следующие задачи: изучена литература по теме работы; рассмотрены известные технологии для создания mesh-сетей; выбран способ для организации взаимодействия устройств mesh-сети; разработано программное обеспечение для моделирования mesh-сети в соответствии с выбранным способом; добавлена функция оценки времени прохождения пакетов внутри полученной модели сети; протестирована разработанная программа. То есть все поставленные задачи решены, цель достигнута. Уверен, что полученные знания и опыт пригодятся для успешного выполнения своих трудовых обязанностей в профессиональной деятельности.

Список используемой литературы

1. Нанс, Б. Компьютерные сети от А до Я / Б. Нанс. – Москва : БИНОМ, 2006. – 400 с.

2. Олифер, В.Г. Компьютерные сети. Принципы, технологии, протоколы : учебник для вузов / В.Г. Олифер, Н.А. Олифер. Изд. 4-е. – Санкт-Петербург : Питер, 2010. – 944 с.

3. Осипов, И.Е. Mesh-сети: технологии, приложения, оборудование // Технологии и средства связи. – 2006. – № 4. – С. 39–45.

4. Пролетарский, А.В. Беспроводные сети Wi-Fi // Интернет-Университет Информационных технологий; БИНОМ; Лаборатория знаний. – 2007. – С. 28–32.

5. Сергеев, А. Основы локальных компьютерных сетей / А. Сергеев – Санкт-Петербург : Лань, 2016. – 184 с.

6. Филиппов, А.Н. Свойства и характеристики Ad Hoc сетей // Молодой ученый. – 2016. – №11. – С. 522–525. – Режим доступа: https://moluch.ru/archive/115/31245.

7. Baccour, N., Krief, F., Zeghlache, D. Mesh networks: A survey // Computer Networks 54(15). – 2010. – P. 2591–2608.

8. Bau Y., Ho C.K., Ewe H.T., Ant Colony Optimization Approaches to the Degree-constrained Minimum Spanning Tree Problem // Journal of information science and engineering 24. – 2008. – P. 1081–1094.

9. Beasley J.E., Chu P.C. A genetic Algorithm for the Set Covering Problem // European Journal of Operational Research. London. – 1994. – 19 p.

10. Bui T.N., Zrncic C.M. An Ant-Based Algorithm for Finding Degree-Constrained Minimum Spanning Tree // GECCO’06. Seattle, Washington, USA. – 2006. – P. 11–18.

11. Bui T.N., Deng X., Zrncic C.M. An improved ant-based algorithm for the degree-constrained minimum spanning tree problem // IEEE Transactions on

Evolutionary Computation. Pennsylvania State University at Harrisburg, Middletown, PA 17057, United States. – 2012. – P. 266–278.

12. Davis L. Handbook of genetic algorithms // Artificial Intelligence 100. New York : Van Nostrand Reinhold. – 1991. – P. 325–330.

13. Gao X, Jia L, Kar S Degree-constrained minimum spanning tree problem of uncertain random network // J Ambient Intell Humaniz Comput 8(5) – 2017. – P. 747–757. Режим доступа: https://www.researchgate.net/publication/316362524_Degree-constrained_minimum_spanning_tree_problem_of_uncertain_random_network.

14. Lin L. Gen M. Node-Based Genetic Algorithm for Communication Spanning Tree Problem // IEICE Transactions on Communications. – 2006. – P. 1091–1098. Режим доступа: https://www.researchgate.net/publication/31271875_Node-Based_Genetic_Algorithm_for_Communication_Spanning_Tree_Problem

15. Minh N. Doan. An Effective Ant-Based Algorithm for the Degree-Constrained Minimum Spanning Tree Problem // IEEE Congress on Evolutionary Computation. Singapore. – 2007. – P. 485–491.

16. Niaz Imtiaz Md., Akkas Ali Md. An Improved Ant-Based Algorithm for Minimum Degree Spanning Tree Problems // IOSR Journal of Computer Engineering. Bangladesh. – 2012. – P. 6–10.

17. Silvestri S, Laporte G, Cerulli R. A branch-and-cut algorithm for the minimum branch vertices spanning tree problem // Comput Oper Res 81. Canada. – 2017. – P. 322–332. Режим доступа: http://collections.banq.qc.ca/ark:/52327/bs2503296.

18. Singh K., Sundar S. A hybrid genetic algorithm for the degree-constrained minimum spanning tree problem // Soft Computing 24(3). – 2020. – P. 2169–2186. Режим доступа: https://www.researchgate.net/publication/333100644_A_hybrid_genetic_algorithm_for_the_degree-constrained_minimum_spanning_tree_problem.

19. Volgenant, A., A Lagrangean Approach to the Degree-Constrained Minimum Spanning Tree Problem // European Journal of Operational Research, 39. – 1989. – P. 325–331.

20. Wamiliana. Solving the degree constrained minimum spanning tree problem using tabu and modified penalty search methods // Journal Teknik Industri 6(1). – 2004. – P. 1–9. Режим доступа: https://www.researchgate.net/publication/47716374_Solving_the_degree_constrained_minimum_spanning_tree_problem_using_tabu_and_modified_penalty_search_methods

Приложение А

Листинг исходного кода с алгоритмом HSSGA

```
import random
import math
import numpy as np

def is_graph_a_tree(_n_v, _edges):
    def is_cyclic_util(v, _visited, parent):
        _visited[v] = True
        # For all vertex connected to v
        for e in [_ for _ in _edges if v in _]:
            _i = e[0] if e[1] == v else e[1]
            # Recursive vertex checking
            if not _visited[_i]:
                if is_cyclic_util(_i, _visited, v):
                    return True
            # If found a cycle
            elif _i != parent:
                return True
        return False

    visited = [False] * _n_v
    if is_cyclic_util(0, visited, -1):
        return False
    for _ in range(_n_v):
        if not visited[_]:
            return False
    return True

class SpanningTree:
    def __init__(self, _edges, _degree, _cost):
        self.edges = _edges
        self.degree = _degree
        self.cost = _cost

    def is_tree(self):
        return is_graph_a_tree(len(self.degree), self.edges)

def prim_rst(_n, _adj_mat, _V, _E, _max_degree):
    T = set()
    s = random.choice(tuple(_V))
    C = {s}  # set of connected nodes
    A = {_ for _ in _E if s in _}  # eligible edges, e.g. [(1,2), (2,3), ...]
    _degree = [0] * _n
    cost = 0
    while C != _V:
        uv = random.choice([_ for _ in A if (_[0] in C or _[1] in C)])  # choose an edge (u, v) \in A,
u \in C at random
        A.remove(uv)
        u, v = (uv[0], uv[1]) if uv[0] in C else (uv[1], uv[0])  # make sure u \in C
```

```
        if v not in C and _degree[u] < _max_degree and _degree[v] < _max_degree:  # connect v to
the partial tree
            T.add(uv)
            _degree[u] += 1
            _degree[v] += 1
            cost += _adj_mat[u][v]
            C.add(v)
            A = A.union({_ for _ in _E
                    if _[0] == v and _[1] not in C
                    or _[1] == v and _[0] not in C})
    return SpanningTree(T, _degree, cost)

def x_over(_n, _adj_mat, _V, _E, p_1, p_2, _max_degree):
    T = set()
    degree = [0] * _n
    cost = 0
    mark = [0] * _n
    v_1 = random.choice(tuple(_V))
    S = {v_1}  # set of connected nodes
    mark[v_1] = 1
    Pb_p1p2 = 1 / p_1.cost / (1 / p_1.cost + 1 / p_2.cost)
    u_01 = random.random()
    if u_01 < Pb_p1p2:
        e_v1v2 = random.choice([_ for _ in p_1.edges if v_1 in _])
    else:
        e_v1v2 = random.choice([_ for _ in p_2.edges if v_1 in _])
    v_2 = e_v1v2[0] if e_v1v2[1] == v_1 else e_v1v2[1]
    mark[v_2] = 1
    S.add(v_2)
    T.add(e_v1v2)
    degree[v_1] += 1
    degree[v_2] += 1
    cost += _adj_mat[v_1][v_2]
    while _V - S != set():
        u_01 = random.random()
        if u_01 < Pb_p1p2:
            try:  # search in p_1
                e_ij = random.choice([_ for _ in p_1.edges
                                      if _[0] in S and _[1] in _V - S and degree[_[0]] < _max_degree
                                      or _[1] in S and _[0] in _V - S and degree[_[1]] < _max_degree])
            except IndexError:  # if the search in p_1 is not successful
                try:  # search in p_2
                    e_ij = random.choice([_ for _ in p_2.edges
                                          if _[0] in S and _[1] in _V - S and degree[_[0]] < _max_degree
                                          or _[1] in S and _[0] in _V - S and degree[_[1]] < _max_degree])
                except IndexError:  # if the searches in p_1 and p_2 are not successful
                    e_ij = next(_ for _ in [e for e in _E if e not in T]  # E is a list while T is a set
                                if _[0] in S and _[1] in _V - S and degree[_[0]] < _max_degree
                                or _[1] in S and _[0] in _V - S and degree[_[1]] < _max_degree)
                i, j = (e_ij[0], e_ij[1]) if e_ij[0] in S else (e_ij[1], e_ij[0])
```

```
            mark[j] = 1
            S.add(j)
            T.add(e_ij)
            degree[i] += 1
            degree[j] += 1
            cost += _adj_mat[i][j]
        else:  # if the search in p_2 is successful
            i, j = (e_ij[0], e_ij[1]) if e_ij[0] in S else (e_ij[1], e_ij[0])
            mark[j] = 1
            S.add(j)
            T.add(e_ij)
            degree[i] += 1
            degree[j] += 1
            cost += _adj_mat[i][j]
    else:  # if the search in p_1 is successful
        i, j = (e_ij[0], e_ij[1]) if e_ij[0] in S else (e_ij[1], e_ij[0])
        mark[j] = 1
        S.add(j)
        T.add(e_ij)
        degree[i] += 1
        degree[j] += 1
        cost += _adj_mat[i][j]
else:
    try:  # search in p_2
        e_ij = random.choice([_ for _ in p_2.edges
                              if _[0] in S and _[1] in _V - S and degree[_[0]] < _max_degree
                              or _[1] in S and _[0] in _V - S and degree[_[1]] < _max_degree])
    except IndexError:  # if the search in p_2 is not successful
        try:  # search in p_1
            e_ij = random.choice([_ for _ in p_1.edges
                                  if _[0] in S and _[1] in _V - S and degree[_[0]] < _max_degree
                                  or _[1] in S and _[0] in _V - S and degree[_[1]] < _max_degree])
        except IndexError:  # if the searches in p_2 and p_1 are not successful
            e_ij = next(_ for _ in [e for e in _E if e not in T]
                        if _[0] in S and _[1] in _V - S and degree[_[0]] < _max_degree
                        or _[1] in S and _[0] in _V - S and degree[_[1]] < _max_degree)
        i, j = (e_ij[0], e_ij[1]) if e_ij[0] in S else (e_ij[1], e_ij[0])
        mark[j] = 1
        S.add(j)
        T.add(e_ij)
        degree[i] += 1
        degree[j] += 1
        cost += _adj_mat[i][j]
    else:  # if the search in p_2 is successful
        i, j = (e_ij[0], e_ij[1]) if e_ij[0] in S else (e_ij[1], e_ij[0])
        mark[j] = 1
        S.add(j)
        T.add(e_ij)
        degree[i] += 1
        degree[j] += 1
```

```
                cost += _adj_mat[i][j]
            else:  # if the search in p_2 is successful
                i, j = (e_ij[0], e_ij[1]) if e_ij[0] in S else (e_ij[1], e_ij[0])
                mark[j] = 1
                S.add(j)
                T.add(e_ij)
                degree[i] += 1
                degree[j] += 1
                cost += _adj_mat[i][j]
    return SpanningTree(T, degree, cost)

def connected_component(_n, _v, _edges):
    def dfs_util(temp, _v, _visited):
        _visited[_v] = True  # Mark the current vertex as visited
        temp.add(_v)  # Store the vertex to list
        adj_edges = [_ for _ in _edges if _v in _]
        for i in [_[0] if _[1] == _v else _[1] for _ in adj_edges]:  # Repeat for all vertices adjacent to
this vertex v
            if not _visited[i]:
                temp = dfs_util(temp, i, _visited)  # Update the list
        return temp

    return dfs_util(set(), _v, [False] * _n)

def del_ins(_n, _adj_mat, _V, _E, p, _max_degree):
    edges = p.edges.copy()
    degree = p.degree.copy()
    cost = p.cost
    e_del = random.choice(tuple(edges))
    u, v = e_del
    edges.remove(e_del)
    degree[u] -= 1
    degree[v] -= 1
    cost -= _adj_mat[u][v]
    T_u = connected_component(_n, u, edges)
    T_v = _V - T_u
    e_ins = random.choice([_ for _ in set(_E) - p.edges if
                           _[0] in T_u and degree[_[0]] < _max_degree and
                           _[1] in T_v and degree[_[1]] < _max_degree or
                           _[1] in T_u and degree[_[1]] < _max_degree and
                           _[0] in T_v and degree[_[0]] < _max_degree])
    edges.add(e_ins)
    degree[e_ins[0]] += 1
    degree[e_ins[1]] += 1
    cost += _adj_mat[e_ins[0]][e_ins[1]]
    return SpanningTree(edges, degree, cost)

def dijkstra_in_tree(adj_mat, source, tree):
    _n=len(adj_mat)
    _V=set(range(len(adj_mat)))
```

```
    _E=tree.edges
    Q = _V.copy()
    dist = [math.inf] * _n
    dist[source] = 0
    while Q != set():
        u = min([_ for _ in enumerate(dist) if _[0] in Q], key=lambda x: x[1])[0]
        Q.remove(u)
        for e in [_ for _ in _E if u in _]:
            v = e[0] if e[1] == u else e[1]
            alt = dist[u] + adj_mat[u][v]
            if alt < dist[v]:
                dist[v] = alt
    return dist

def two_er(_n, _adj_mat, p, max_iter):
    edges = p.edges.copy()
    cost = p.cost
    best_wx = ()  # either best_wx or best_w, best_x cannot launch the other
    best_w, best_x = -1, -1  # because order matters!
    for t in range(max_iter):
        cost_diff = math.inf  # reset every iteration!
        e_uv = random.choice([_ for _ in edges])
        u, v = e_uv
        for e in [_ for _ in edges if u not in _ and v not in _]:  # non_adjacent edge of e_uv
            w, x = e
            if not is_graph_a_tree(_n, edges - {e_uv, e} | {(u, w), (v, x)}):
                w, x = e[1], e[0]
            if _adj_mat[u][w] == -1 or _adj_mat[v][x] == -1:
                continue
            temp = _adj_mat[u][w] + _adj_mat[v][x] - _adj_mat[u][v] - _adj_mat[w][x]  # the less, the
better
            if temp < cost_diff:
                cost_diff = temp
                best_wx = e
                best_w, best_x = w, x
        edges = edges - {e_uv, best_wx} | {(u, best_w), (v, best_x)}
        cost = cost + cost_diff
    return SpanningTree(edges, p.degree.copy(), cost,)

def one_er(_n, _adj_mat, _E, p, _max_degree, max_iter):
    edges = p.edges.copy()
    degree = p.degree.copy()
    cost = p.cost
    for t in range(max_iter):
        # First stage of 1ER
            for e_uv in [_ for _ in edges if degree[_[0]] == _max_degree or degree[_[1]] ==
_max_degree]:
            u, v = e_uv
            try:
                e_xy = random.choice([_ for _ in set(_E) - edges if
```

```
                        _adj_mat[_[0]][_[1]] <= _adj_mat[u][v] and
                        degree[_[0]] + 1 <= _max_degree and
                        degree[_[1]] + 1 <= _max_degree and
                        is_graph_a_tree(_n, edges - {e_uv} | {_})])
        except IndexError:
            continue
        else:  # if the search is successful
            x, y = e_xy
            edges = edges - {e_uv} | {e_xy}
            degree[u] -= 1
            degree[v] -= 1
            degree[x] += 1
            degree[y] += 1
            cost = cost - _adj_mat[u][v] + _adj_mat[x][y]
    # Second stage of 1ER
    for e_uv in edges:
        u, v = e_uv
        try:
            e_xy = random.choice([_ for _ in set(_E) - edges if
                        _adj_mat[_[0]][_[1]] < _adj_mat[u][v] and
                        degree[_[0]] + 1 <= _max_degree and
                        degree[_[1]] + 1 <= _max_degree and
                        is_graph_a_tree(_n, edges - {e_uv} | {_})])
        except IndexError:
            continue
        else:  # if the search is successful
            x, y = e_xy
            edges = edges - {e_uv} | {e_xy}
            degree[u] -= 1
            degree[v] -= 1
            degree[x] += 1
            degree[y] += 1
            cost = cost - _adj_mat[u][v] + _adj_mat[x][y]
    return SpanningTree(edges, degree, cost)

def hssga(_adj_mat, _max_degree, number_pop, number_iter, P_crossover, P_tour_select, alpha,
unchanged_iter_pop, solution_to_change):
    _n = len(_adj_mat)
    _V = set(range(_n))  # vertices set
    _E = {(i, j) for i in range(_n) for j in range(i + 1, _n) if _adj_mat[i][j] != -1}
    _E = sorted(_E, key=lambda x: _adj_mat[x[0]][x[1]])  # edge list sorted by weight in descending
order

    pop = [prim_rst(_n, _adj_mat, _V, _E, _max_degree) for _ in range(number_pop)]
    pop_avg_cost = sum([_.cost for _ in pop]) / number_pop
    # pop_min_cost = min([_.cost for _ in pop])

    T_gb = min(pop, key=lambda x: x.cost)  # best-so-far generated
    unchanged_iter = 0
```

```
    data = {'pop_avg_cost': [0] * number_iter, 'best_cost': [0] * number_iter}

    for _iter in range(number_iter):
        u_01 = random.random()
        if u_01 < P_crossover:
                p_1 = np.random.choice(sorted(random.sample(pop, 2), key=lambda x: x.cost),
p=[P_tour_select, 1-P_tour_select])
                p_2 = np.random.choice(sorted(random.sample(pop, 2), key=lambda x: x.cost),
p=[P_tour_select, 1-P_tour_select])
            T_C = x_over(_n, _adj_mat, _V, _E, p_1, p_2, _max_degree)
        else:
                p_1 = np.random.choice(sorted(random.sample(pop, 2), key=lambda x: x.cost),
p=[P_tour_select, 1-P_tour_select])
            T_C = del_ins(_n, _adj_mat, _V, _E, p_1, _max_degree)
        edge_diff = 1 - len(T_gb.edges.intersection(T_C.edges)) / (_n - 1)
        if T_gb.cost/T_C.cost+alpha*edge_diff > 1:
            T_C = two_er(_n, _adj_mat, T_C, max_iter=3)
            T_C = one_er(_n, _adj_mat, _E, T_C, _max_degree, max_iter=3)
        for T in pop:
            if T.edges == T_C.edges:  # T_C not unique
                break
        else:  # T_C unique, apply replacement strategy
            if T_C.cost < T_gb.cost:
                T_gb = T_C
                unchanged_iter = 0
            else:
                unchanged_iter += 1
            inferior_T = random.choice([_ for _ in pop if _.cost > pop_avg_cost])
              pop = [T_C if _.edges == inferior_T.edges else _ for _ in pop]  # replace T_C with
inferior_T
            if unchanged_iter > unchanged_iter_pop:  # apply population update strategy
                count = 0
                while count < solution_to_change:
                    i, T_i = random.choice(list(enumerate(pop)))
                    T_p = del_ins(_n, _adj_mat, _V, _E, T_i, _max_degree)  # perturbed solution
                    for T in pop:
                        if T.edges == T_p.edges:  # T_p not unique, do not replace
                            break
                    else:  # T_p unique, do replacement
                        pop[i] = T_p  # replace T_i with T_p
                        count += 1
        pop_avg_cost = sum([_.cost for _ in pop]) / number_pop
        data['pop_avg_cost'][_iter] = pop_avg_cost
        data['best_cost'][_iter] = T_gb.cost
    return T_gb, data
```

Приложение Б

Листинг исходного кода запускаемого файла

```
from hssga import *
from yaml import load, Loader
import pydot
from collections import deque
from sys import exit
import time
import random

def generate_graph(num):
    matrix = [0] * num
    for i in range(num):
        matrix[i] = [0] * num
    for i in range(num):
        for j in range(i, num):
            element = random.random() * 10000
            matrix[i][j] = element
            matrix[j][i] = element
    return matrix



if __name__ == "__main__":
    # Read configuration
    with open('config.yaml', 'r') as f:
        data = load(f, Loader=Loader)

    test = False if "test" not in data else data["test"]

    if test:
        num_v = 100 if "num_vertexes" not in data else data["num_vertexes"]
        adj_mat = generate_graph(num_v)
        print("Generated full graph with", num_v, "vertexes!")
    elif "matrix" in data:
        adj_mat = data["matrix"]
    else:
        print("\nError: expected adjacency matrix.")
        exit(0)

    flag = False
    for i in range(len(adj_mat) - 1):
        for j in range(i + 1, len(adj_mat[0])):
            if adj_mat[i][j] != adj_mat[j][i]:
                flag = True
                adj_mat[j][i] = adj_mat[i][j]
    if flag:
        print("\nMatrix is asymmetrical! Only the numbers above the main diagonal will be taken.")
```

```
    if "max_degree" in data:
        max_degree = data["max_degree"]
    else:
        print("\nError: expected max degree.")
        exit(0)

    number_pop = 300 if "number_pop" not in data else data["number_pop"]
    number_iter = 2000 if "number_iter" not in data else data["number_iter"]
    P_tour_select = 0.9 if "P_tour_select" not in data else data["P_tour_select"]
    P_crossover = 0.5 if "P_crossover" not in data else data["P_crossover"]
    alpha = 0.1 if "alpha" not in data else data["alpha"]
    unchanged_iter_pop = 500 if "unchanged_iter_pop" not in data else data["unchanged_iter_pop"]
    solution_to_change = 1 if "solution_to_change" not in data else data["solution_to_change"]

    start = time.time()
     T_gb, data_of_hssga = hssga(adj_mat, max_degree, number_pop, number_iter, P_crossover,
P_tour_select, alpha, unchanged_iter_pop, solution_to_change)
    end = time.time()

    graph = pydot.Dot("Mesh model", graph_type="graph", size="20!")

    count = len(str(T_gb.cost))
    _sum = set()
    if not test:
        print("\nMinimum distance to vertexes matrix:", end='\n ')
        for i in range(len(adj_mat)):
            print(" | " + ("%" + str(count) + "d") % i, end='')
        print()
    for i in range(len(adj_mat)):
        dist = dijkstra_in_tree(adj_mat=adj_mat, source=i, tree=T_gb)
        _sum = _sum | {(i, sum(dist))}
        if not test:
            print(i, end='')
            for j in range(len(adj_mat)):
                print(" | "  + ("%" + str(count) + "d") % dist[j], end='')
            print()

    print("\nMinimum spanning tree cost:", T_gb.cost)
    print("Time of computing: %.4f" % (end - start))
    print("\nResult tree in model.png file", end="\n\n")

    center = (min(_sum,key=lambda x: x[1])[0]) if "root" not in data else data["root"]
    deque = deque()
    deque.append(center)
    list_ = list()

    while len(deque) != 0:
        curEdge = deque.popleft()
        for i in T_gb.edges:
            if (i[0] == curEdge and i[1] not in list_):
```

```
            edge = pydot.Edge(i[0], i[1], label=adj_mat[i[0]][i[1]], fontsize="10")
            graph.add_edge(edge)
            deque.append(i[1])
        if (i[1] == curEdge and i[0] not in list_):
            edge = pydot.Edge(i[1], i[0], label=adj_mat[i[1]][i[0]], fontsize="10")
            graph.add_edge(edge)
            deque.append(i[0])
    list_.append(curEdge)

if not test:
    graph.write_png('model.png')
```