МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Рязанский государственный радиотехнический университет имени В.Ф. Уткина»

На правах рукописи

НГУЕН ВАН ТИН

МАТЕМАТИЧЕСКОЕ И ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ ПРОЦЕССОВ ИНТЕЛЛЕКТУАЛЬНОЙ МАРШРУТИЗАЦИИ И БАЛАНСИРОВКИ ПОТОКОВ ДАННЫХ В ПРОГРАММНО-КОНФИГУРИРУЕМЫХ СЕТЯХ НА ОСНОВЕ НЕЙРОННЫХ СЕТЕЙ И РОЕВЫХ АЛГОРИТМОВ

Специальность: 2.3.5. «Математическое и программное обеспечение вычислительных систем, комплексов и компьютерных сетей»

Диссертация на соискание ученой степени
кандидата технических наук

Научный руководитель:
Перепелкин Дмитрий Александрович
Доктор технических наук, профессор

Рязань – 2025

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Актуальность темы. Развитие технологии программно-конфигурируемых сетей (ПКС) совершило революцию в управлении и эксплуатации компьютерными сетями за счет разделения плоскости управления и передачи данных. Такое разделение позволяет централизованно управлять сетью, обеспечивая ее гибкость и адаптацию к изменяющимся условиям. Благодаря своим преимуществам перед традиционными сетями, ПКС стали широко использоваться в современных центрах обработки данных и облачных сервисах.

ПКС имеют большой потенциал для дальнейшего развития с быстрым прогрессом таких технологий, как Интернет вещей (Internet of Things, IoT) и искусственный интеллект (ИИ). В настоящее время ПКС развиваются в направлении интеллектуальной и эффективной автоматизации процессов управления и эксплуатации сетей с использованием решений на базе ИИ, а также в сторону более тесной интеграции с облачными сервисами. С быстрым распространением технологии 5G, ПКС стали занимать ключевую роль в управлении компьютерными сетями, для передачи данных которых требуется высокая пропускная способность и минимальная задержка.

Развитие технологии искусственного интеллекта привнесло значительные улучшения в процессы управления и оптимизации ПКС. ИИ-алгоритмы способны анализировать большие объемы данных, что позволяет с высокой точностью прогнозировать будущие состояния сети. В ПКС, технологии ИИ могут применятся для прогнозирования и классификации трафика, балансировки нагрузки, предсказания DDoS-атак, а также для принятия решений по маршрутизации. Постоянный рост объема данных открывает новые возможности для развития ИИ, способствующего созданию более эффективных решений для эксплуатации сетей ПКС.

По сравнению с традиционными сетями, маршрутизация в программно-

конфигурируемых сетях претерпела значительные изменения. Благодаря возможностям обновления информации в режиме реального времени через контроллер и протокол OpenFlow, процессы маршрутизации становятся более гибкими и адаптивными, что позволяет ПКС быстро реагировать на изменения в сети. ПКС обеспечивают программируемость, позволяя применять сложные алгоритмы маршрутизации на основе собранных данных, что способствует оптимизации трафика, повышению производительности и надежности сети.

Вопросам ПКС и протокола OpenFlow посвящены работы Р. Л. Смелянского [68-70], Ю. Л. Леохина [71-74] и др. Методы и алгоритмы маршрутизации в ПКС подробно рассмотрены в работах Д. В. Куракина [75, 76], В. Н. Тарасова [77], Д. А. Перепелкина [80-83] и др. Задачу нахождения кратчайших путей рассматривали в своих трудах ученые E. W. Dijkstra [84], R. Bellman [85], В. А. Евстигнеева [86, 87]. Заметный вклад в разработку методов и алгоритмов управления многопотоковым трафиком в компьютерных сетях внесли В. П. Корячко [88-95], С. И. Макаренко [78], О. Я. Кравец [79]. Развитие методов и алгоритмов управления потоками данных с обеспечением качества сервиса в ПКС подробно рассматривается в работах В. Г. Карташевского [100, 101], М. А. Бурановой [102], П. Н. Полежаева [104, 105], С. В. Малахова [103, 106] и др.

Анализ и исследование существующих методов маршрутизации и балансировки потоков данных в ПКС показал, что большинство имеющихся подходов имеет ряд ограничений, которые значительно снижают эффективность их применения. Во-первых, многие из них не способны адаптироваться к динамическим изменениям в сети, таким как перегрузка трафика или задержки, что может приводить к субоптимальным решениям. Во-вторых, существующие алгоритмы маршрутизации зачастую не обладают достаточной масштабируемостью, что затрудняет их использование в крупных и сложных сетях. Кроме того, высокие вычислительные затраты и время, необходимые для поиска оптимальных маршрутов, делают их менее подходящими для задач, решаемых в режиме реального времени. Наконец,

недостаточная устойчивость к отказам и неспособность быстро адаптироваться к изменениям топологии сети также представляют собой значительные проблемы для многих существующих методов.

Таким образом, в настоящее время актуальной является задача разработки математического и программного обеспечения процессов интеллектуальной маршрутизации и балансировки потоков данных в ПКС на основе нейронных сетей и алгоритмов роевого интеллекта с целью создания более адаптивных и эффективных решений. Алгоритмы роевого интеллекта, такие как оптимизация муравьиной колонии, алгоритм стаи птиц, алгоритм искусственной пчелиной колонии и другие, предлагают более гибкие и устойчивые подходы к маршрутизации за счет их способности адаптироваться к изменениям в сети. Искусственные нейронные сети, в свою очередь, могут обеспечить высокую производительность в условиях реального времени благодаря своей способности обучаться и принимать решения на основе большого объема данных. Комбинирование этих методов имеет высокий потенциал для создания новых, более эффективных систем маршрутизации, способных справляться с современными вызовами и требованиями сетевой инфраструктуры.

Работа выполнена в ФГБОУ ВО «Рязанский государственных радиотехнический университет имени В.Ф. Уткина» в рамках научного направления «Автоматизация проектирования и программное обеспечение высокопроизводительных систем и компьютерных сетей».

Цель и задачи исследования. Цель работы заключается в повышении эффективности процессов передачи и обработки данных в ПКС за счет разработки новых математических моделей, методов, алгоритмов и программных средств интеллектуальной маршрутизации и балансировки потоков данных на основе искусственных нейронных сетей и методов роевого интеллекта, обеспечивающих высокую точность определения оптимальных маршрутов и равномерное распределение нагрузки в условиях динамически изменяющейся сети.

Для достижения поставленной цели необходимо решить следующие основные задачи:

- провести сравнительный анализ традиционных и программно-конфигурируемых сетей, выделив ключевые преимущества и недостатки каждого подхода;

- провести анализ существующих методов и алгоритмов динамической многопутевой маршрутизации в ПКС;

- разработать математическую модель и метод интеллектуальной маршрутизации в ПКС на основе алгоритмов роевого интеллекта;

- спроектировать модель искусственной нейронной сети для решения задачи многопутевой маршрутизации в ПКС;

- оптимизировать гиперпараметры нейронной сети с помощью алгоритмов роевого интеллекта;

- разработать математическую модель и метод балансировки потоков данных для равномерного распределения нагрузки в ПКС;

- разработать программную систему интеллектуальной маршрутизации и балансировки потоков данных в ПКС;

- провести эксперименты и сравнительный анализ предложенных алгоритмов интеллектуальной маршрутизации и балансировки потоков данных в ПКС и оценить их эффективность.

Объект исследования: программно-конфигурируемые сети и методы интеллектуальной маршрутизации и балансировки потоков данных на основе алгоритмов роевого интеллекта и искусственных нейронных сетей.

Предмет исследования: средства математического и программного обеспечения интеллектуальной маршрутизации и балансировки потоков данных в ПКС.

Методы исследования. Для достижения поставленных целей в работе используются методы теории графов для исследования топологии сети, теория алгоритмов для разработки и анализа алгоритмов маршрутизации на основе роевого

интеллекта и искусственных нейронных сетей, а также методы теории матриц для оптимизации маршрутов и оценки их эффективности. Кроме того, для проведения экспериментов и проверки предложенных решений применяются методы компьютерного моделирования, технологии объектно-ориентированного программирования, а также методы статистического анализа.

Тематика работы соответствует следующим пунктам паспорта специальности 2.3.5. «Математическое и программное обеспечение вычислительных систем, комплексов и компьютерных сетей»: п.3 «Модели, методы, архитектуры, алгоритмы, языки и программные инструменты организации взаимодействия программ и программных систем»; п.9 «Модели, методы, алгоритмы, облачные технологии и программная инфраструктура организации глобально распределенной обработки данных».

Научная новизна. В работе получены следующие результаты, отличающиеся научной новизной:

- математическая модель и метод интеллектуальной маршрутизации в ПКС, отличающиеся использованием алгоритмов роевого интеллекта и их адаптацией для условий динамически изменяющейся сети;

- нейросетевая модель многопутевой маршрутизации в ПКС на основе рекуррентной нейронной сети, позволяющая принимать решения о маршрутизации в режиме реального времени;

- математическая модель и метод оптимизации гиперпараметров нейросетевой модели многопутевой маршрутизации в ПКС на основе алгоритмов роевого интеллекта, обеспечивающие высокую точность прогнозирования маршрутов и снижение вычислительных затрат;

- модель и алгоритм динамической балансировки потоков данных в ПКС, обеспечивающие равномерное распределение нагрузки в сети и адаптацию к изменяющимся условиям трафика для увеличения пропускной способности и минимизации потерь пакетов;

- архитектура библиотеки программных компонентов интеллектуальной маршрутизации и балансировки потоков данных в ПКС, отличающаяся наличием программных интерфейсов для взаимодействия с сетевыми приложениями и обеспечивающая эффективное управление потоками данных на основе нейронных сетей и роевых алгоритмов;

- структура программной системы для организации распределенной обработки данных, отличающаяся использованием микросервисной архитектуры и возможностью гибкого конфигурирования параметров и структуры сети.

Положения, выносимые на защиту:

1 Математическая модель и метод интеллектуальной маршрутизации в ПКС на основе алгоритмов роевого интеллекта, обеспечивающие гибкую адаптацию к изменениям параметров сети.

2 Нейросетевая модель многопутевой маршрутизации в ПКС с использованием рекуррентных нейронных сетей, позволяющая принимать решения о маршрутизации в режиме реального времени.

3 Математическая модель и метод оптимизации гиперпараметров нейросетевой модели многопутевой маршрутизации в ПКС на основе алгоритмов роевого интеллекта, обеспечивающие высокую точность прогнозирования маршрутов и снижение вычислительных затрат.

4 Модель и алгоритм динамической балансировки потоков данных в ПКС, позволяющие равномерно распределять нагрузку в сети, устраняя перегрузки на отдельных каналах и увеличивая общую пропускную способность.

5 Архитектура библиотеки программных компонентов интеллектуальной маршрутизации и балансировки потоков данных в ПКС, отличающаяся наличием программных интерфейсов для взаимодействия с сетевыми приложениями и обеспечивающая эффективное управление потоками данных в сети на основе нейронных сетей и роевых алгоритмов.

6 Структура программной системы для организации распределенной обработки

данных, отличающаяся применением микросервисной архитектуры и возможностью гибкого конфигурирования параметров и структуры сети.

Степень достоверности. Обоснованность полученных результатов определяется корректным использованием теории алгоритмов, теории графов, теории множеств, методов компьютерного моделирования и объектно-ориентированного программирования.

Практическая значимость. Предложенные алгоритмы интеллектуальной маршрутизации и балансировки потоков данных в ПКС на основе роевого интеллекта и искусственных нейронных сетей способствуют дальнейшему развитию программных средств управления сетевым трафиком в современных компьютерных сетях. Алгоритмы реализованы в составе визуальной программной системы SDNLoadBalancer, включающей графический редактор, эмулятор сети, генератор трафика и комплексную систему мониторинга и интеллектуального управления параметрами ПКС. Разработанная программная система может быть использована для проектирования и оптимизации сетевых инфраструктур, обеспечивая высокую производительность и адаптивность сетей. Программные компоненты, созданные в рамках данной работы, могут быть интегрированы в существующие системы управления сетью и использованы для динамического распределения нагрузки и повышения надежности сетевых соединений. На элементы разработанных программных средств получены свидетельства о государственной регистрации в реестре Федеральной службы по интеллектуальной собственности.

Реализация и внедрение результатов работы. Разработанные в диссертационной работе модели, алгоритмы и программная система интеллектуальной маршрутизации и балансировки потоков данных в ПКС внедрены в учебном процессе на кафедре САПР ВС Рязанского государственного радиотехнического университета имени В.Ф. Уткина (РГРТУ), а также в инженерной практике компании ООО «Технологии HTR» (Вьетнам).

Апробация результатов диссертации. Основные результаты диссертационной работы докладывались и обсуждались на следующих всероссийских и международных конференциях: Международный научно-технический форум «Современные технологии в науке и образовании – СТНО» (г. Рязань, 2022 и 2023), Mediterranean Conference on Embedded Computing «MECO» (Budva, Montenegro, 2022), ELEKTRO «ELEKTRO» (Krakow, Poland, 2022), International Russian Automation Conference «RusAutoCon» (Sochi, 2022), International Russian Smart Industry Conference «SmartIndustryCon» (Sochi, 2023), International Symposium Problems of Redundancy in Information and Control Systems «REDUNDANCY» (Moscow, 2023).

Публикации. По результатам диссертационного исследования опубликованы 22 научные работы, из них: 7 статей в изданиях из Перечня ведущих рецензируемых научных журналов и изданий ВАК по специальности 2.3.5 (3 – К1, 4 – К2); 5 статей в изданиях, входящих в международные базы научного цитирования Web of Science и Scopus; 3 статьи в других изданиях и материалах конференций; 7 авторских свидетельства о регистрации программ для ЭВМ в ФГБУ «Федеральный институт промышленной собственности» (РОСПАТЕНТ). Все результаты диссертационной работы, включая постановку задач, разработку и исследование методов и алгоритмов, создание программной системы для многопутевой маршрутизации и балансировки потоков данных в программно-конфигурируемых сетях, а также основные научные результаты и выводы, являются результатом самостоятельной работы автора. Направление исследования и концептуальные подходы были определены автором совместно с научным руководителем, что позволило сформулировать актуальные научные задачи и уточнить методологии их решения.

Структура и объем диссертации. Диссертационная работа состоит из введения, 5 глав, заключения, списка литературы, 7 приложений, изложенных на 258 страницах (включая 80 рисунков и 18 таблиц). Список литературы содержит 136 наименования.

ГЛАВА 1 АНАЛИЗ И РАЗВИТИЕ СОВРЕМЕННЫХ КОМПЬЮТЕРНЫХ СЕТЕЙ НА ОСНОВЕ ТЕХНОЛОГИИ ПРОГРАММНО-КОНФИГУРИРУЕМЫХ СЕТЕЙ

1.1 ТРАДИЦИОННЫЕ КОМПЬЮТЕРНЫЕ СЕТИ

Ethernet-коммутатор – это один из наиболее часто используемых сетевых элементов, служащий точкой подключения к сети для хостов в локальных сетях (LAN). Он использует аппаратные адреса, MAC-адреса, для передачи кадров на канальном уровне модели OSI (Open Systems Interconnection). Коммутатор работает на канальном уровне, создавая отдельный сегмент сети для каждого интерфейса, что позволяет устройствам, подключенным к каждому интерфейсу, одновременно передавать и получать данные без коллизий [1-3].

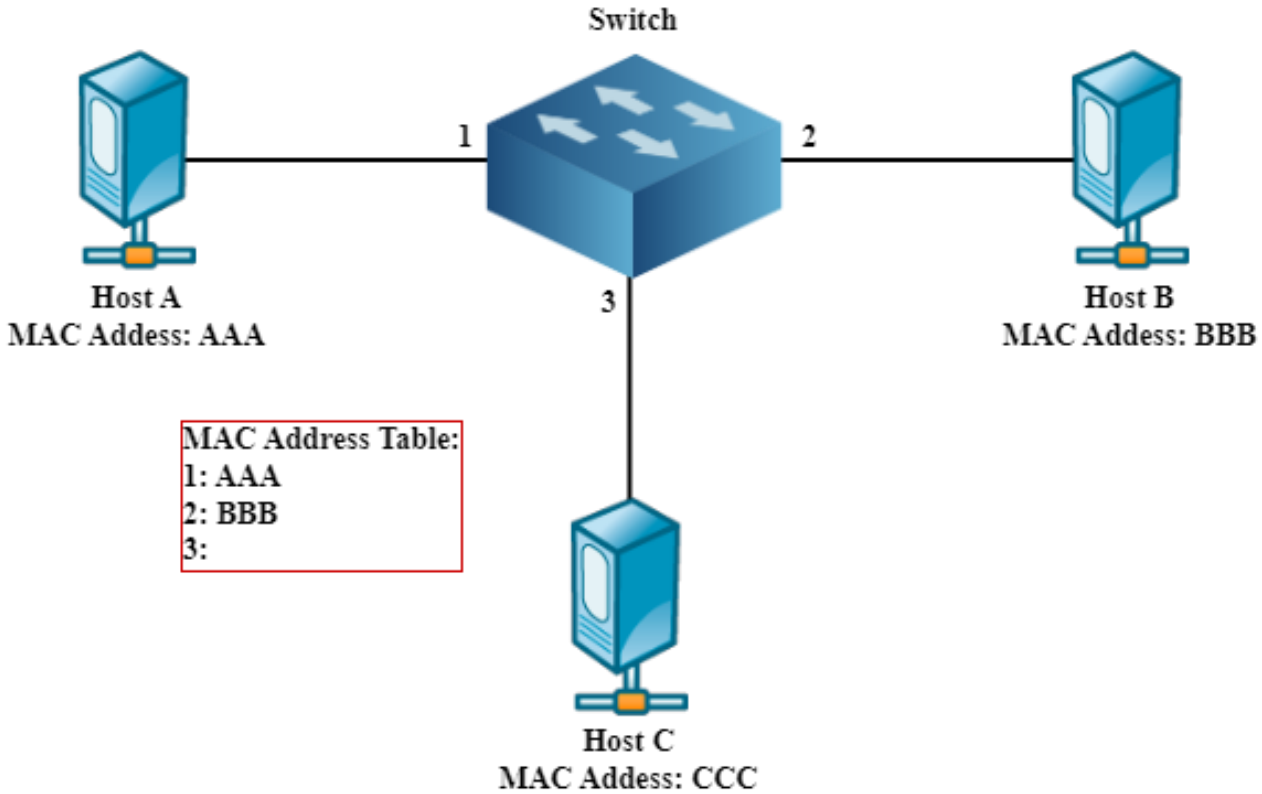


Рисунок 1.1 – Механизм запоминания MAC-адресов в Ethernet-коммутаторах

Ethernet-коммутаторы динамически запоминают MAC-адреса, анализируя MAC-адрес источника во входящих кадрах. Процесс работает следующим образом:

- **Исходное состояние**: При первом включении коммутатора его таблица MAC-адресов пуста.
- **Получение кадра**: Когда кадр поступает на порт, коммутатор проверяет MAC-адрес источника и порт, через который поступил кадр.
- **Изучение MAC-адреса**: Если MAC-адрес источника отсутствует в таблице MAC-адресов, коммутатор добавляет его в таблицу вместе с информацией о порте, на котором он был обнаружен.
- **Передача или широковещательная передача**: Если MAC-адрес назначения присутствует в таблице, коммутатор передает кадр через соответствующий порт. В противном случае коммутатор выполняет широковещательную передачу кадра по всем портам, за исключением того, на который он был получен.
- **Обновление таблицы**: Коммутатор продолжает изучать MAC-адреса и обновлять свою таблицу по мере получения новых кадров.

Современные мобильные устройства, облачные технологии и виртуализация требуют переосмысления традиционной сетевой архитектуры. Старая иерархическая модель, основанная на Ethernet-коммутаторах, неэффективна для динамичных операций в дата-центрах, на предприятиях и у операторов связи. Она не способна гибко реагировать на изменения трафика или обеспечивать необходимую масштабируемость.

Традиционные сети статичны, их настройка под каждый новый запрос осуществляется вручную, что значительно усложняет управление. Сетевые функции распределены между различными устройствами – коммутаторами, маршрутизаторами и балансировщиками нагрузки. Каждое устройство управляется через специфический интерфейс отдельного производителя, что затрудняет внедрение

общих политик управления по всей сети. Существующие инструменты централизованного управления чаще контролируют состояние сети, нежели позволяют ее полноценно конфигурировать. Статическая маршрутизация не подходит для адаптации сети к изменениям в трафике, пиковым нагрузкам или специфическим требованиям приложений.

Для современных сетей критически важна автоматизация, позволяющая самостоятельно реагировать на изменения и эффективно использовать ресурсы. Виртуализация сети упрощает управление сетью, устраняя зависимость от физических устройств. Программно-конфигурируемые сети предлагают решение этих проблем, обеспечивая гибкость и масштабируемость, необходимые для современной инфраструктуры [4].

1.2 ПРОГРАММНО-КОНФИГУРИРУЕМЫЕ СЕТИ (ПКС)

Программно-конфигурируемые сети представляют собой относительно новую парадигму в мире сетевых технологий, которая вносит фундаментальные изменения в способ организации и функционирования современных сетей. Основная идея ПКС заключается в разделении плоскостей управления и передачи данных. Это дает возможность внешнему контроллеру ПКС гибко и динамически управлять правилами, которые применяются к сетевым устройствам с поддержкой ПКС. Такие устройства, получая указания от контроллера, выполняют функции фильтрации, обработки и передачи сетевых пакетов, адаптируясь к текущим условиям сети и требуемым политиками безопасности. Благодаря такой структуре сетевые администраторы могут быстро изменять конфигурации сети, обеспечивая большую гибкость, масштабируемость и оптимизацию использования ресурсов, делая сеть более эффективной и надежной [5-9].

1.2.1 АРХИТЕКТУРНЫЕ ОСОБЕННОСТИ ПКС

Архитектуру ПКС можно представить как совокупность различных уровней: уровня инфраструктуры, уровня управления и уровня приложений. Каждый уровень выполняет свои собственные функции.

На рисунке 1.2 приведена основная архитектура ПКС.

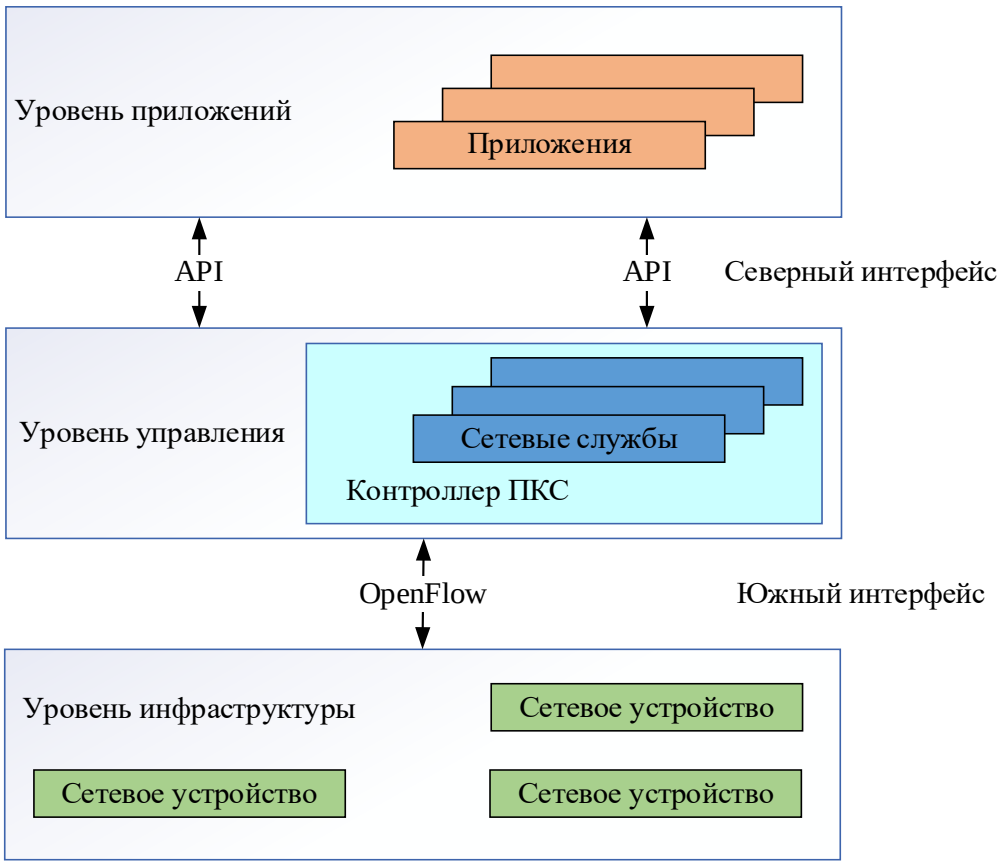


Рисунок 1.2 – Архитектура ПКС

Уровень инфраструктуры

Уровень инфраструктуры состоит из базовых коммутационных устройств, которые функционируют без встроенного интеллекта, обрабатывая пакеты данных согласно правилам, заданным контроллером. Эти правила, включая параметры для передачи пакетов, хранятся в локальной памяти устройств, таких как SRAM и TCAM. В системах ПКС, в отличие от традиционных сетей, правила передачи могут включать дополнительные параметры, такие как VLAN, протоколы TCP или UDP, идентификаторы потоков и порты входящего трафика [10].

Уровень управления

Контроллер ПКС – ключевой элемент, который программирует и управляет устройствами передачи данных через южный интерфейс. Этот уровень управления анализирует требования конечных пользовательских приложений и преобразует их в сетевые политики, которые затем применяются в плоскости данных. Контроллер служит централизованным «мозгом» в рамках сетевой операционной системы NOS, обеспечивая связь требований приложений с правилами для устройств передачи данных.

Уровень приложений

Приложения ПКС, создаваемые на этом уровне, оптимизируют работу контроллера ПКС, предоставляя управление сетью в зависимости от изменяющихся условий, таких как сбои каналов связи или узлов. Приложения могут выполнять функции, включая балансировку нагрузки, мониторинг и управление сетью, обеспечение качества обслуживания (QoS), безопасности и доступа. Связь между приложениями ПКС и уровнем управления осуществляется через северный интерфейс API.

Вместо использования традиционных политик и необходимости освоения многочисленных стандартных протоколов, ПКС значительно облегчает работу с устройствами сетевой инфраструктуры, такими как коммутаторы и маршрутизаторы, превращая их в устройства, ограничивающиеся лишь функциями передачи данных. В то же время, управление и интеллектуальные сетевые функции переходят на уровень контроллеров или сетевой операционной системы NOS, которые через южный интерфейс задают выполнение функций на уровне передачи данных. Уровень управления обеспечивает доступ к различным сервисам и приложениям через северный интерфейс, действующий на уровне приложений. Это позволяет разработчикам и исследователям сосредоточиться на каждом уровне отдельно, не отвлекаясь на проблемы других уровней.

1.2.2 ПРОТОКОЛ OPENFLOW

Самый популярный протокол южного направления, OpenFlow, был основан в Стэнфордском университете в 2008 году и в настоящее время управляется Фондом открытых сетей (ONF) [11, 12]. OpenFlow представляет собой протокол связи, который позволяет контроллеру ПКС напрямую управлять плоскостью данных и принимать решения о маршрутизации на сетевых устройствах. С его помощью контроллер может создавать, удалять и изменять записи в таблице потоков коммутатора.

Коммутатор с поддержкой OpenFlow содержит одну или несколько таблиц потоков. Каждая таблица состоит из серии записей потока, которые коммутатор последовательно проверяет, чтобы определить, как обрабатывать каждый входящий пакет. Основная функция таблиц потока – обеспечить быструю и эффективную обработку пакетов. Когда пакет поступает на коммутатор, коммутатор сверяет поля заголовка пакета с записями потока в своих таблицах, чтобы определить соответствующее действие, например передачу, удаление, изменение или перенаправление пакета [13].

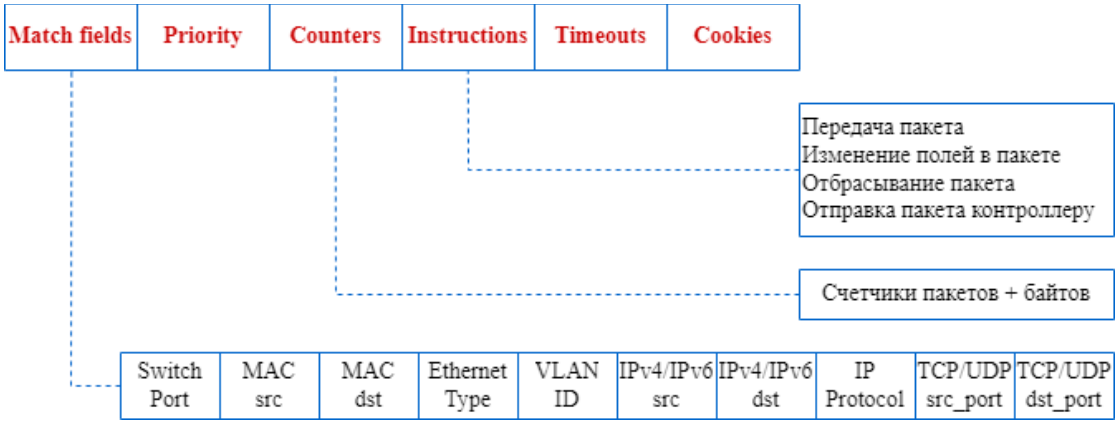


Рисунок 1.3 – Запись потока OpenFlow

Каждая запись потока в таблице потоков состоит из нескольких компонентов:

- **Поле соответствия (Match Fields)**: Они определяют поля заголовка пакета, которые необходимо сопоставить, такие как IP-адрес источника, IP-адрес назначения, порты TCP/UDP, идентификатор VLAN и т. д.

- **Приоритет (Priority**): Определяет приоритет записи потока в таблице. Если одному пакету соответствует несколько записей, выбирается та, которая имеет наивысший приоритет.
- **Счетчики (Counters**): Каждая запись потока включает счетчики, которые отслеживают, сколько пакетов и байтов соответствует записи.
- **Инструкции (Instructions)**: Инструкции сообщают коммутатору, что делать с пакетами, которые соответствуют записи потока. Общие инструкции включают в себя:
  - Передача пакета на определенные выходные порты.
  - Изменение полей в пакете, например перезапись IP-адресов источника или назначения.
  - Отбрасывание пакета.
  - Отправка пакета контроллеру, который затем может принять более сложное решение о том, как с ним обращаться.
- **Таймауты (Timeouts)**: Это время, в течение которого запись потока остается активной в таблице потоков, что помогает избежать конфликтов с устаревшими или больше не используемыми записями.
- **Куки-файл (Cookie)**: Это уникальный идентификатор, который контроллер может использовать для отслеживания записей потока.

Протокол OpenFlow с его структурированной таблицей потоков обеспечивает надежную основу для эффективного управления сетевым трафиком. Благодаря использованию полей соответствия, приоритетов и инструкций, OpenFlow позволяет точно управлять потоками трафика, создавая динамическую программируемую сеть. Эти возможности обеспечивают адаптивность сети к различным требованиям и конфигурациям, улучшая общую производительность и делая управление сетью более гибким и эффективным [14].

1.2.3 КОНТРОЛЛЕРЫ ПКС

Контроллеры ПКС играют ключевую роль в управлении и настройке сетевой инфраструктуры, предоставляя централизованное управление сетью, автоматизацию и гибкость. Они взаимодействуют с коммутаторами в плоскости данных, используя открытые и стандартизированные интерфейсы, известные как интерфейсы южного направления, и протоколы, такие как OpenFlow. Контроллеры управляют элементами передачи для выполнения широкого спектра функций, таких как маршрутизация, коммутация, межсетевой экран, трансляция сетевых адресов и балансировка нагрузки [15, 16].

Существует широкий спектр контроллеров ПКС, которые успешно применяются в различных приложениях, включая корпоративные сети и научные исследования. Обзор некоторых из наиболее используемых и популярных контроллеров с открытым исходным кодом показан в таблице 1.1.

Таблица 1.1 – Типы контроллеров ПКС

Контроллер	**Организация**	**Язык**	**Основные характеристики**
NOX	Nicira Networks	C++	Первый контроллер, управление устройствами низкого уровня
POX	Nicira Networks	Python	Версия NOX на Python, используемая для более быстрой разработки и создания прототипов новых сетевых приложений
Ryu	NTT	Python	Поддержка различных протоколов, простота в использовании
OpenDaylight	Linux Foundation	Java	Многофункциональный контроллер, гибкий и универсальный, поддержка различных протоколов

ONOS	ON.Lab	Java	Высокая доступность, масштабируемость, производительность, подходит для телекоммуникаций
Beacon	Stanford University	Java	Быстрый, легкий, разработан для модульности
Floodlight	Big Switch Networks	Java	Поддержка REST API, модульная конструкция

- NOX является одним из первых контроллеров ПКС, разработанных компанией Nicira и переданных исследовательскому сообществу в 2008 году, став открытым исходным кодом. NOX предоставляет C++ API для OpenFlow (версии 1.0) и использует асинхронную модель программирования, основанную на событиях. NOX часто используется в академических исследованиях сетей для разработки ПКС-приложений.

- POX – это версия NOX на языке Python, которую можно рассматривать как новую итерацию NOX для сообщества Python-разработчиков. POX был разработан для того, чтобы предоставить более доступную среду программирования по сравнению с NOX, сохраняя при этом мощные функции управления сетью. POX использует Python 2.7 и поддерживает высокоуровневый API для ПКС, включая возможность запроса графа топологии и поддержку виртуализации. Это делает процесс разработки ПКС-приложений более удобным и открывает возможности для экспериментов с новыми сетевыми протоколами без необходимости глубокого знания C++.

- OpenDaylight (ODL) – это открытая платформа ПКС, разработанная под эгидой Linux Foundation и впервые выпущенная в 2013 году. Основная цель OpenDaylight – ускорить внедрение ПКС и виртуализации сетевых функций (NFV). Архитектура ODL очень гибкая и модульная, что позволяет поддерживать множество

различных протоколов управления сетью, таких как OpenFlow, NETCONF, BGP-LS и OVSDB.

- ONOS, разработанный Open Networking Foundation в 2014 году, представляет собой контроллер ПКС с открытым исходным кодом, ориентированный на поставщиков услуг связи. ONOS нацелен на обеспечение высокой надежности сети, масштабируемости и производительности. Одной из ключевых особенностей ONOS является его распределенная архитектура, позволяющая контроллеру работать на нескольких физических серверах, что улучшает отказоустойчивость и масштабируемость.
- Beacon – это высокопроизводительный контроллер ПКС, разработанный на языке Java в Стэнфордском университете в 2010 году. Изначально созданный для научных исследований, Beacon благодаря своей высокой производительности и способности обрабатывать пакеты с низкими задержками также подходит для использования в производственных средах.
- Floodlight – это открытый контроллер ПКС, разработанный компанией Big Switch Networks и являющийся одним из самых популярных контроллеров в сообществе ПКС. Архитектура Floodlight модульная, с компонентами, включающими управление топологией сети, управление устройствами (отслеживание MAC и IP), вычисление маршрутов, веб-инфраструктуру для управления сетью и хранилище счетчиков (счетчики OpenFlow). Эти компоненты рассматриваются как загружаемые сервисы с интерфейсами для обмена данными. Floodlight широко используется как в экспериментальных, так и в реальных условиях, благодаря своей простоте внедрения и поддержке сложных функций ПКС.
- Ryu – это один из самых популярных сегодня контроллеров с открытым исходным кодом, разработанных компанией NTT Communications. Контроллер Ryu предлагает ряд преимуществ перед другими контроллерами, что делает его привлекательным выбором для управления сетью. Во-первых, Ryu поддерживает

широкий спектр сетевых протоколов, включая OpenFlow, Netconf и OF-config, обеспечивая большую гибкость в управлении различными сетевыми устройствами. Такая гибкость позволяет сетевым администраторам эффективно реализовывать сложные сетевые политики и операции. Во-вторых, Ryu предлагает хорошо документированный API, который упрощает разработку сетевых приложений, позволяя внедрять инновации и настраивать их. Кроме того, он написан на Python, популярном и простом в освоении языке программирования, который ускоряет циклы разработки и сокращает время обучения для новых пользователей. Эти функции делают Ryu привлекательным вариантом для тех, кто ищет адаптируемый и удобный для разработчиков контроллер ПКС [17].

1.2.4 ЭМУЛЯТОР MININET

Mininet – это популярный сетевой эмулятор, широко применяемый для моделирования сетевой инфраструктуры в средах ПКС. Этот инструмент служит важнейшим ресурсом для тестовых стендов ПКС, облегчая проектирование, тестирование и проверку проектов OpenFlow. Одной из основных сильных сторон Mininet является высокий уровень программируемости: пользователи могут создавать топологии и интегрировать новые функции с помощью Python. Более того, Mininet предлагает масштабируемую среду, способную управлять до 4000 коммутаторами на стандартном компьютере [18-20].

Mininet работает преимущественно на операционной системе Linux, которая считается предпочтительной для исследовательских целей. Однако Mininet также можно запустить на macOS и Windows с использованием виртуальных машин или контейнеров, таких как VirtualBox, VMware или Docker. Кроме того, Mininet может быть использован для моделирования небольшого центра обработки данных, состоящего из хостов и коммутаторов OpenFlow. Благодаря таким возможностям

пользователи могут получать практические результаты, что подтверждает универсальность Mininet в области сетевого моделирования и экспериментов.

Основные команды Mininet

Это команды, которые обычно используются для запуска сеанса Mininet и взаимодействия с ним.

Таблица 1.2 – Основные команды Mininet

Команда	Описание
sudo mn	Запускает Mininet с настройками по умолчанию.
sudo mn --topo	Указывает топологию сети (например, дерево, линейная).
sudo mn --mac	Автоматически устанавливает MAC-адреса.
sudo mn --controller	Устанавливает контроллер (например, *none* - без контроллера, *remote* - внешний контроллер).
exit	Выход из текущего сеанса Mininet.

Команды для управления узлами

Эти команды используются в интерфейсе командной строки Mininet для взаимодействия с конкретными хостами или коммутаторами.

Таблица 1.3 – Команды для управления узлами

Команда	Описание
nodes	Выводит список всех узлов сети (коммутаторов и хостов).
net	Выводит список соединений между узлами сети.
h1 ifconfig	Отображает конфигурацию интерфейса для хоста h1.
h1 ping h2	Выполняет команду ping с хоста h1 на хост h2.
s1 ovs-vsctl show	Отображает конфигурацию Open vSwitch для коммутатора s1.

Команды управления сетевыми каналами

Эти команды предназначены для управления и проверки сетевых каналов в топологии.

Таблица 1.4 – Команды управления сетевыми каналами

Команда	Описание
link h1 h2 up	Устанавливает соединение между h1 и h2.
link h1 h2 down	Разрывает соединение между h1 и h2.
links	Отображает список всех соединений.

Команды отладки

Эти команды помогут диагностировать проблемы в виртуальной сети.

Таблица 1.5 – Команды отладки

Команда	Описание
pingall	Проверяет соединение между всеми узлами.
pingpair	Проверяет соединение между парами узлов.
iperf	Проверяет пропускную способность между узлами. edit

Расширенные команды

Предназначены для тонкой настройки и отладки.

Таблица 1.6 – Расширенные команды

Команда	Описание
xterm h1	Открывает окно терминала xterm для узла h1.
py net.hosts[0].cmd('ifconfig')	Выполняет команду ifconfig на первом узле, используя Python API.
dump	Выводит подробную информацию о всех узлах сети.

Mininet является незаменимым инструментом как для образовательных, так и для исследовательских целей в области программно-конфигурируемых сетей. Его способность быстро создавать прототипы и эмулировать сложные сетевые топологии на одной машине, а также совместимость с OpenFlow и другими протоколами ПКС делают его идеальной платформой для экспериментов и тестирования. Предоставляя

реалистичную и экономически эффективную среду, Mininet позволяет исследователям и студентам изучать новые концепции ПКС, проверять проекты сетей и анализировать производительность без необходимости использования дорогостоящего оборудования. С дальнейшим развитием ПКС, гибкость и простота использования Mininet, несомненно, останутся ценными преимуществами, способствуя инновациям и углубленному пониманию в этой динамичной области.

1.3 СРАНИТЕЛЬНЫЙ АНАЛИЗ ТРАДИЦИОННЫХ И ПРОГРАММНО-КОНФИГУРИРУЕМЫХ СЕТЕЙ

В сфере сетевых технологий традиционные и программно-конфигурируемые сети представляют собой противоположные подходы. Традиционные сети ориентированы на аппаратное обеспечение и объединяют плоскости управления и данных, тогда как ПКС представляет собой гибкую модель, управляемую программным обеспечением.

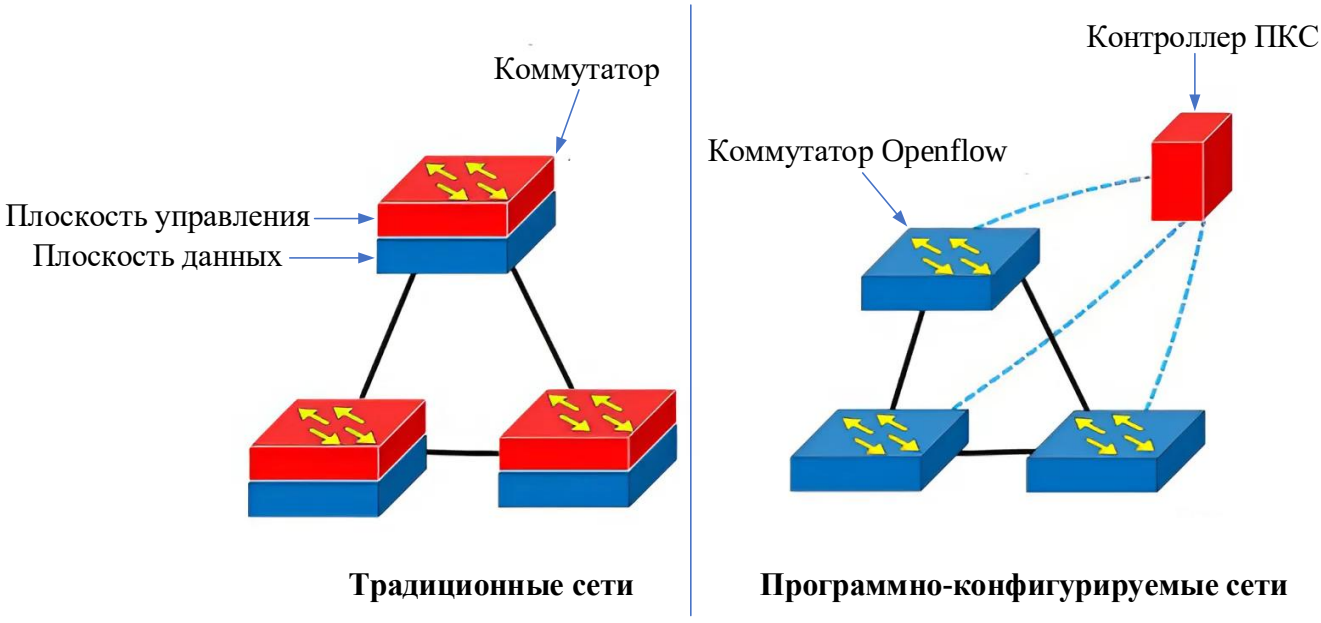


Рисунок 1.4 – Традиционные и программно-конфигурируемые сети

Фундаментальные различия между традиционными сетевыми методами и ПКС показаны в таблице 1.7.

Таблица 1.7 – Сранительный анализ традиционных и ПКС

Характеристика	**Традиционные сети**	**Программно-конфигурируемые сети**
Архитектура	Обычно монолитная, аппаратура и программное обеспечение тесно интегрированы.	Разделение управляющей и передающей плоскостей. Управление централизовано.
Управляющая плоскость	Интегрирована в сетевое оборудование, распределенное управление.	Централизованная управляющая плоскость, которая может управляться программно.
Конфигурация	Ручная настройка через настройку каждого устройства отдельно.	Централизованная и автоматизированная конфигурация через программные контроллеры.
Масштабируемость	Масштабирование может быть сложным и зависит от аппаратуры.	Более легко масштабируема за счет централизованного управления и контроля.
Гибкость	Ограничена возможностями аппаратуры и проприетарным программным обеспечением.	Высокая гибкость, поддерживает быстрое внедрение новых услуг и изменений.
Видимость и контроль	Ограниченная видимость трафика и поведения в сети.	Высокая видимость и детальный контроль над сетевым трафиком.
Безопасность	Безопасность управляется устройство за устройством и может быть непоследовательной.	Централизованные политики безопасности, потенциально улучшая согласованность и время реакции.
Стоимость	Часто более высокие начальные затраты из-за инвестиций в аппаратуру.	Потенциально ниже капитальных затрат, но может потребоваться инвестиции в новые навыки и технологии.
Управление сетью	Обычно сложно и требует много времени.	Упрощено за счет автоматических инструментов и централизованного управления.

Переход от традиционных сетевых архитектур к программно-конфигурируемым сетям означает значительную эволюцию сетевых технологий. ПКС обеспечивает повышенную масштабируемость, экономическую эффективность и безопасность за счет централизации и упрощения управленческих функций, что существенно контрастирует с жесткими и сложными конфигурациями традиционных сетей. Поскольку предприятия продолжают требовать более динамичных и адаптируемых сетевых сред, ПКС может стать новым стандартом, изменяющим то, как организации проектируют, эксплуатируют и управляют сетями во все более взаимосвязанном мире [21].

1.4 МЕТОДЫ МАРШРУТИЗАЦИИ В ПКС

В программно-конфигурируемых сетях, маршрутизация управляется централизованно контроллером, который действует как мозг сети. В отличие от традиционных сетей, где каждое сетевое устройство принимает независимые решения о маршрутизации, в ПКС контроллер имеет целостное представление о сети и принимает все решения о маршрутизации. Такой централизованный контроль позволяет более динамично и гибко управлять сетевым трафиком. Контроллер может быстро корректировать маршруты в ответ на изменяющиеся условия сети, оптимизировать потоки трафика в сети и реализовывать сложные политики безопасности и управления трафиком. В результате ПКС может повысить эффективность, оперативность и масштабируемость сети за счет упрощения и автоматизации процессов маршрутизации.

1.4.1 СТАТИЧЕСКАЯ И ДИНАМИЧЕСКАЯ МАРШРУТИЗАЦИИ В ПКС

Механизмы маршрутизации в программно-конфигурируемых сетях можно разделить на статическую и динамическую маршрутизацию. Статическая

маршрутизация включает в себя заранее определенные, настроенные вручную маршруты, которые не изменяются, если не обновляются вручную. Этот подход обеспечивает простоту и предсказуемость, что делает его подходящим для стабильных небольших сред, где хорошо понятны модели трафика. С другой стороны, динамическая маршрутизация в ПКС опирается на алгоритмы и протоколы для автоматической корректировки путей в зависимости от условий сети в реальном времени. Это обеспечивает более гибкое и эффективное управление потоками данных, что имеет решающее значение в больших и сложных сетевых средах с часто меняющимися структурами трафика. Оба метода являются неотъемлемой частью оптимизации производительности и надежности сети в различных операционных контекстах в рамках ПКС [22].

Статическая маршрутизация

Процесс статической маршрутизации в ПКС выполняется в соответствии с определенными шагами, чтобы гарантировать, что пакеты передаются по заранее настроенным маршрутам. Ниже приведено детальное описание процесса статической маршрутизации в ПКС по шагам:

- **Конфигурация статического маршрута**: Сетевой администратор вручную настраивает статические маршруты на контроллере ПКС. Эти маршруты содержат информацию о промежуточных узлах (например, коммутаторах), через которые будут проходить пакеты.
- **Создание таблицы маршрутизации**: После получения конфигурации от администратора, контроллер ПКС создает таблицу статической маршрутизации, которая хранит правила, связанные с передачей данных. Эта таблица включает пары IP-адресов (источник и назначение), а также соответствующие порты, через которые пакеты должны передаваться.
- **Создание таблицы потоков на коммутаторах**: ПКС контроллер использует такие протоколы, как OpenFlow, для взаимодействия с коммутаторами. Таким образом, контроллер передает статические маршруты на соответствующие

коммутаторы в сети. Эти правила записываются в таблицы потоков каждого коммутатора.

- **Обработка пакетов на коммутаторах**: Когда пакет поступает на коммутатор, коммутатор обращается к таблице потоков, чтобы определить, какое правило соответствует данному пакету, исходя из IP-адресов источника и назначения, а также других критериев (таких как номер протокола, целевой порт и т.д.). В соответствии с заранее установленными правилами статической маршрутизации, коммутатор передает пакет через соответствующий интерфейсный порт. Пакет продолжает передаваться через каждый коммутатор, пока не достигнет конечного пункта назначения.

Динамическая маршрутизация

В отличие от статической маршрутизации, где пути задаются заранее и остаются неизменными, динамическая маршрутизация позволяет контроллеру ПКС постоянно отслеживать состояние сети. На основе этой информации контроллер может принимать обоснованные решения о выборе оптимальных путей для передачи пакетов данных.

В ПКС динамическая маршрутизация может быть разделена на два типа в зависимости от способа расчета стоимости канала связи: статическая и динамическая стоимость.

- **Статическая стоимость канала**: Это фиксированные параметры, которые не меняются со временем или в зависимости от состояния сети. Протоколы маршрутизации в ПКС используют эти параметры для расчета оптимального пути передачи данных. К наиболее распространенным параметрам относятся:
  - Количество переходов: Это количество сетевых устройств (таких как коммутаторы или маршрутизаторы), через которые проходит пакет от источника к назначению. Маршрут с наименьшим числом переходов будет выбран.
  - Расстояние: Расстояние между узлами сети также может использоваться в качестве стоимости канала.

- 1/Пропускная способность: Стоимость канала может рассчитываться как обратная величина от пропускной способности. Канал связи с большей пропускной способностью будет иметь меньшую стоимость и, следовательно, будет предпочтительнее.

- **Динамическая стоимость канала**: Зависит от текущего состояния сети и меняется со временем. Эти параметры отражают текущую нагрузку и фактическую степень использования канала связи в сети, что помогает оптимизировать маршруты в зависимости от реальной ситуации в сети. Динамические параметры могут включать:

  - Нагрузка: Текущая нагрузка на сетевые связи. Канал с меньшей нагрузкой обычно выбираются с приоритетом.
  - Коэффициент загруженности канала: Рассчитывается на основе текущей скорости передачи данных и ограниченной пропускной способности канала. Менее загруженные каналы будут иметь меньшую стоимость и будут предпочтительнее.
  - Оставшаяся пропускная способность: Это доступная пропускная способность на каждом канале. Канал с большей оставшейся пропускной способностью будут иметь меньшую стоимость.
  - Задержка: Может изменяться в зависимости от состояния сети.

При динамической маршрутизации часто используются такие алгоритмы, как алгоритм Дейкстры или Беллмана-Форда, для поиска оптимального пути от источника к пункту назначения, потенциально пересчитывая пути всякий раз, когда происходят изменения в сети. Этот подход уменьшает задержку, позволяет избежать перегрузок и повышает общую устойчивость сети за счет динамической адаптации к сбоям или узким местам. Процесс динамической маршрутизации работает следующим образом.

- **Обработка пакетов на коммутаторах:** Когда пакет поступает на коммутатор, он проверяет таблицу потоков, чтобы определить, какое правило соответствует данному пакету на основе полей соответствия.

- Если в коммутаторе отсутствует правило для обработки пакета, он отправляет событие *Packet-In* на контроллер, который затем устанавливает новое правило для обработки данного пакета.
- В противном случае коммутатор передает пакет согласно установленным правилам маршрутизации по выбранным маршрутам от источника к пункту назначения.

- **Обновление правил маршрутизации**: Контроллер отслеживает сеть в реальном времени и при изменении состояния сети пересчитывает маршруты и обновляет правила.

- Сбор информации о сети: С помощью таких протоколов, как OpenFlow, компоненты мониторинга в контроллере будут собирать данные о состоянии сети (топология или состояние каналов связи).
- Расчет стоимости каналов связи: На основе собранной информации контроллер рассчитывает стоимости каналов связи по определенным критериям. Эти стоимости могут быть статическими или динамическими.
- Вычисление оптимального маршрута: Контроллер использует алгоритмы, такие как Дейкстра или Беллмана-Форда, для расчета оптимальных маршрутов от источника к назначению.
- Установка правил маршрутизации: Контроллер отправляет потоковые правила на коммутаторы, которые записываются в таблицы потоков и определяют, каким образом пакеты должны передаваться через соответствующие порты.

Кроме того, в зависимости от момента времени создания маршрута можно выделить два типа динамической маршрутизации: проактивная маршрутизация и реактивная маршрутизация.

- **Проактивная маршрутизация**: Это метод маршрутизации, при котором маршруты создаются до поступления запроса на передачу данных. Контроллер заранее рассчитывает и устанавливает маршруты в таблицах маршрутизации

коммутаторов. Благодаря этому пакеты могут быстро передаваться, без взаимодействия с контроллером при появлении первого пакета. Однако этот механизм требует больше ресурсов, поскольку маршруты сохраняются в памяти, даже если они не используются. Поэтому данный метод часто применяется в сетях со стабильной нагрузкой и низкими требованиями к задержкам.

- **Реактивная маршрутизация**: Это метод маршрутизации, при котором маршрут создается только при поступлении первого запроса на передачу данных. Когда пакет поступает на коммутатор, и в таблице маршрутизации нет подходящего маршрута, коммутатор отправляет запрос к контроллеру для создания маршрута для этого потока данных. Это помогает экономить память и ресурсы обработки, а также позволяет сети гибко адаптироваться к частым изменениям состояния. Однако процесс создания маршрута увеличивает время обработки, что приводит к более высокой задержке для первого пакета.

В ПКС статическая маршрутизация ценится за простоту и предсказуемость, однако ей не хватает гибкости, и она не способна адаптироваться к изменениям в сети без ручного вмешательства. Динамическая маршрутизация, напротив, обеспечивает адаптивность и отказоустойчивость, автоматически подстраиваясь под изменения в сети, но усложняет управление ею. В конечном счете, выбор между статической и динамической маршрутизацией в ПКС зависит от баланса между потребностью в стабильности и требованиями к адаптивности в условиях динамичной сети.

1.4.2 МНОГОПУТЕВАЯ МАРШРУТИЗАЦИЯ В ПКС

Многопутевая маршрутизация в программно-конфигурируемых сетях – это передовой метод управления трафиком, которая позволяет распределять данные по нескольким маршрутам. Вместо того чтобы выбирать единственный оптимальный путь, как в традиционной маршрутизации, многопутевая маршрутизация использует несколько путей одновременно для повышения производительности сети, надежности

и для балансировки нагрузки. Внедрение многопутевой маршрутизации в сетях требует сложных алгоритмов для вычисления оптимальных маршрутов, тщательного распределения трафика и постоянного мониторинга состояния сети. Хотя реализация такой системы может быть сопряжена с многочисленными трудностями, она значительно улучшает гибкость и эффективность управления и эксплуатации сетевой инфраструктуры. Благодаря этому многопутевая маршрутизация становится все более распространенной в крупных сетевых системах и сетях с высокими требованиями к доступности, что представляет собой важное достижение в области сетевых технологий.

Многопутевая маршрутизация в ПКС может быть разделена на три основные категории:

- **Математические подходы**: Эти методы применяют математические модели и алгоритмы для определения оптимальных путей маршрутизации. Обычно они включают такие методы, как линейное программирование, целочисленное программирование и теорию графов, для решения задач маршрутизации и оптимизации производительности сети.

Методы оптимизации, включая линейное программирование, целочисленное программирование или другие сложные математические структуры, используются для глобальной или локальной оптимизации маршрутов на основе различных сетевых параметров и ограничений. Эти методы часто применяют сложные алгоритмы и модели для минимизации задержек, максимизации пропускной способности и обеспечения эффективного использования сетевых ресурсов [23, 24].

В ряде статей теория графов используется для решения задач маршрутизации в ПКС. Обычно применяются такие методы, как алгоритм Дейкстры, поиск в глубину (DFS) и алгоритмы поиска k-кратчайших путей. Эти методы используют математические свойства графов для эффективного определения оптимальных путей, гарантируя, что пакеты данных передаются по сети наиболее эффективным образом.

Основное внимание уделяется использованию внутренних структур и алгоритмов теории графов для повышения производительности и надежности сети [25-34].

В других статьях анализируется проблема равномерного распределения сетевого трафика по нескольким путям, чтобы избежать перегрузки и оптимизировать использование ресурсов. Эти методы часто включают динамическую настройку путей на основе реальных сетевых условий, таких как текущая нагрузка трафика, доступная пропускная способность и состояние каналов. Методы балансировки нагрузки направлены на предотвращение перегрузки отдельных путей, что способствует улучшению общей производительности, надежности и пропускной способности сети [25, 27, 30, 31, 34-39].

Некоторые статьи посвящены оптимизации нескольких параметров качества обслуживания (QoS), таких как пропускная способность, задержка, джиттер и потеря пакетов при многопутевой маршрутизации. Методы в этой области часто включают сложные алгоритмы, которые определяют приоритетность путей на основе их способности соответствовать конкретным требованиям QoS. Благодаря динамической корректировке решений по маршрутизации на основе показателей QoS в реальном времени, эти методы призваны обеспечить высококачественное взаимодействие с пользователем и поддерживать желаемые уровни производительности для различных сетевых служб и приложений [28, 29, 32, 40, 41].

- **Подходы на основе машинного обучения**: Эти методы используют алгоритмы машинного обучения для прогнозирования и оптимизации путей маршрутизации. Анализируя исторические данные о сетевом трафике, такие подходы позволяют выявлять закономерности и принимать обоснованные решения для повышения эффективности маршрутизации и адаптации к изменяющимся условиям сети. В этой категории обычно используются методы обучения с подкреплением, обучения с учителем и обучения без учителя.

В нескольких статьях рассматривается применение традиционных методов машинного обучения и глубокого обучения для оптимизации многопутевой

маршрутизации в программно-конфигурируемых сетях. Эти подходы прежде всего направлены на классификацию сетевого трафика и динамический выбор оптимальных маршрутов на основе текущего состояния сети и специфических требований различных приложений. Например, в некоторых исследованиях представлены модели, которые используют алгоритмы контролируемого обучения для определения приоритетов сетевых потоков и эффективного управления пропускной способностью, задержкой и джиттером, основываясь на анализе шаблонов трафика. Такие методы, основанные на машинном обучении, не только улучшают качество обслуживания QoS, но и адаптируются к изменениям в состояниях сети без вмешательства человека, значительно повышая эффективность использования сети и производительность для различных приложений [42-44].

Некоторые исследователи используют глубокое обучение с подкреплением (DRL) для разработки стратегий адаптивной многопутевой маршрутизации. Эти исследования фокусируются на применении DRL для анализа и принятия решений в маршрутизации, основываясь непосредственно на многомерных необработанных данных. Такой подход позволяет сети интеллектуально управлять потоками трафика, реагируя на изменения топологии и условий сети в реальном времени. Этот метод оказывается особенно эффективным в сложных сетевых условиях, где традиционные методы статической маршрутизации неэффективны. Благодаря интеграции извлечения признаков с помощью глубоких нейронных сетей и возможностей принятия решений, обеспечиваемых обучением с подкреплением, эти подходы позволяют контроллерам ПКС непрерывно оптимизировать выбор маршрута, минимизируя задержки и максимизируя пропускную способность [45-49].

Еще одним инновационным подходом является использование мультиагентного обучения с подкреплением (MARL) для адаптивной многопутевой маршрутизации. Этот подход расширяет возможности глубокого обучения с подкреплением за счет включения нескольких обучающихся агентов, которые одновременно взаимодействуют с различными сегментами сети для оптимизации решений по

маршрутизации. Этот метод повышает масштабируемость и надежность управления сетью, что делает его пригодным для крупномасштабных и динамично изменяющихся сетевых сред. Используя возможности коллективного обучения нескольких агентов, подходы мультиагентного обучения с подкреплением могут эффективно распределять сетевой трафик, уменьшать перегрузки и повышать общую производительность сети, адаптируясь не только к текущим условиям, но и развиваясь в предвидении будущих состояний сети [50].

- **Подходы, вдохновленные природой**: Применение алгоритмов, вдохновленных природой, таких как генетический алгоритм, оптимизация роя частиц и оптимизация муравьиной колонии, для многопутевой маршрутизации принесло значительные преимущества в оптимизации сетевых ресурсов и повышении качества обслуживания. Эти алгоритмы не только обеспечивают глобальную оптимизацию, но и помогают избегать локальных экстремумов, что повышает эффективность маршрутизации в сложных сетях, таких как ПКС и спутниковые сети. Благодаря способности решать NP-трудные задачи, эти алгоритмы находят наилучшие маршруты с учетом ограничений, таких как пропускная способность, задержка и надежность, превосходя традиционные методы. В результате системы, использующие эти алгоритмы, демонстрируют более высокую производительность, сниженную задержку, оптимизацию пропускной способности и улучшенную скорость доставки пакетов, что значительно улучшает качество обслуживания и эффективность управления сетевыми ресурсами [112-117].

ОСНОВНЫЕ РЕЗУЛЬТАТЫ И ВЫВОДЫ

1. Рассмотрена концепция программно-конфигурируемых сетей. Исследованы архитектурные особенности, включая разделение плоскостей управления и передачи данных, а также использование внешнего контроллера для управления сетевыми

устройствами. Изучены ключевые компоненты архитектуры ПКС: уровень инфраструктуры, уровень управления и уровень приложений.

2. Проанализирован протокол OpenFlow. Изучен процесс управления таблицами потоков в коммутаторах с поддержкой OpenFlow, включая поля соответствия, приоритеты, счетчики, инструкции, тайм-ауты и куки-файл.

3. Рассмотрена роль контроллеров ПКС. Проанализированы различные типы контроллеров, такие как NOX, POX, Ryu, OpenDaylight, ONOS, Beacon и Floodlight, отличающиеся используемыми языками программирования, основными характеристиками и областями применения.

4. Проанализирован эмулятор Mininet. Изучены основные команды Mininet для управления узлами, сетевыми каналами и выполнения отладочных операций, что делает его полезным инструментом для образовательных и исследовательских целей.

5. Проведен сравнительный анализ традиционных сетей и программно-конфигурируемых сетей. Рассмотрены различия по следующим характеристикам: архитектура, управляющая плоскость, конфигурация, масштабируемость, гибкость, видимость и контроль, безопасность и стоимость.

6. Проведены анализ и исследование существующих методов маршрутизации и балансировки потоков данных в ПКС, которые показали, что большинство имеющихся подходов обладают рядом ограничений, значительно снижающих эффективность их применения. Среди этих ограничений можно выделить низкую адаптивность к динамическим изменениям, недостаточную масштабируемость, высокие вычислительные затраты и слабую устойчивость к отказам. Таким образом, в настоящее время актуальной является задача разработки математического и программного обеспечения для интеллектуальной маршрутизации и балансировки потоков данных в ПКС на основе нейронных сетей и алгоритмов роевого интеллекта с целью создания более адаптивных и эффективных решений.

ГЛАВА 2 МАТЕМАТИЧЕСКОЕ ОБЕСПЕЧЕНИЕ ИНТЕЛЛЕКТУАЛЬНОЙ МАРШРУТИЗАЦИИ В ПКС

2.1 ПОСТАНОВКА ЗАДАЧИ И МАТЕМАТИЧЕСКАЯ МОДЕЛЬ

Топологию ПКС можно представить в виде графа $G = (V, E)$, где V обозначает множество узлов сети, а E – множество каналов между узлами. Каждый канал $(v_i, v_j) \in E, v_i, v_j \in V,$ имеет связанный с ним вес w_{ij}. Вес канала рассчитывается на основе коэффициента загруженности канала следующим простым методом:

$$w_{ij} = \frac{1}{1-u_{ij}},$$

где u_{ij} – коэффициент загруженности канала, который рассчитывается как отношение текущей пропускной способности трафика к максимальной пропускной способности этого канала.

$$u_{ij} = \frac{bw_{ij}^{used}}{bw_{ij}},$$

где bw_{ij}^{used} – используемая пропускная способность и bw_{ij} – общая пропускная способность канала.

Когда коэффициент загруженности канала приближается к 100%, его вес значительно увеличивается, что препятствует его выбору для нового трафика. Эта обратная корреляция гарантирует, что более используемые каналы станут менее желательными для маршрутизации, что способствует использованию менее загруженных путей.

Каждый путь представляет собой последовательность узлов от источника до назначения, не содержащую циклов.

$$P_r = [v_s, \dots, v_d],$$

где v_s и v_d – исходный и целевой узлы соответственно. Стоимость пути представляет собой сумму его весов каналов.

$$w_{P_r} = \sum_{(v_i,v_j)\in P_r} w_{ij}.$$

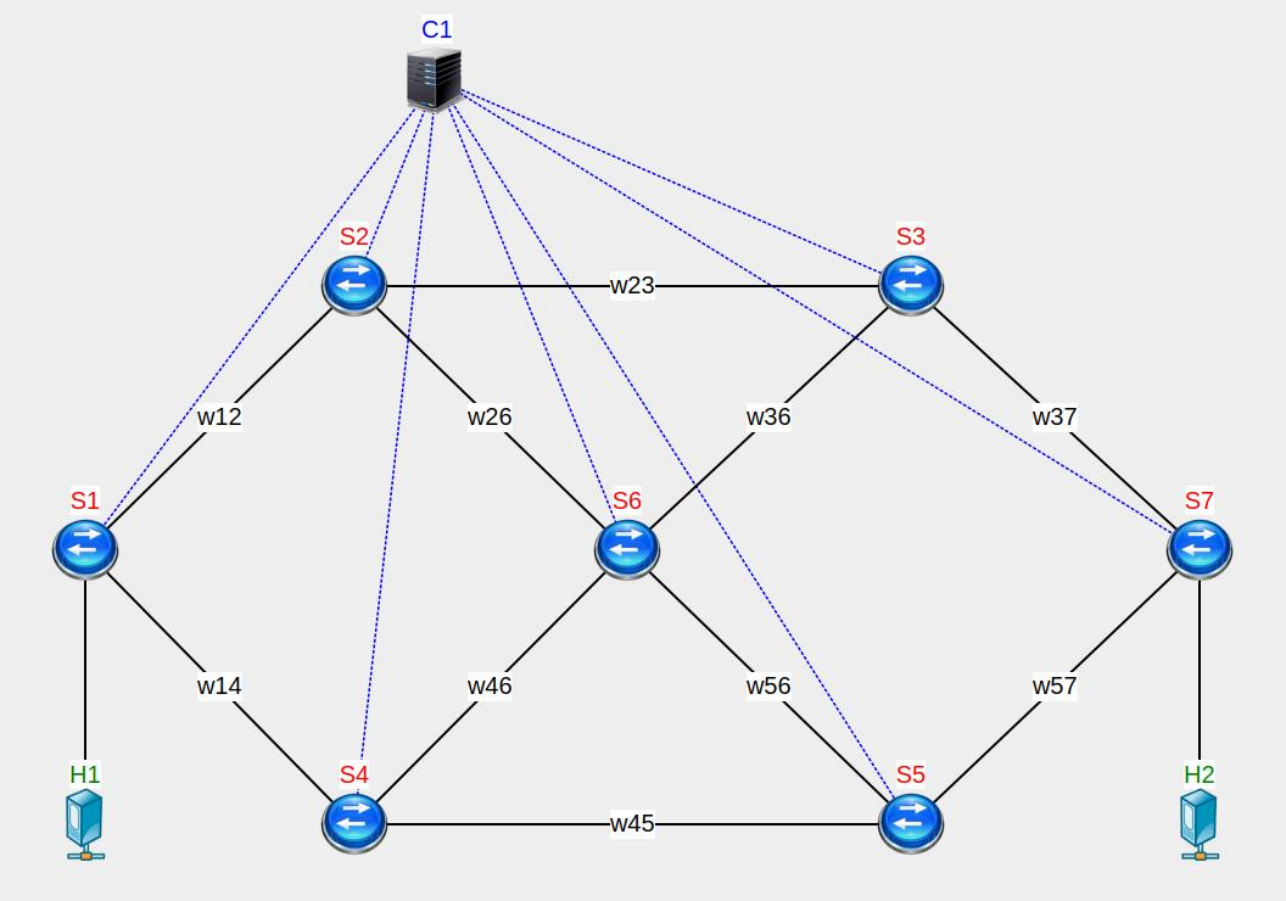


Рисунок 2.1 – Топология ПКС

Задача многопутевой маршрутизации состоит в том, чтобы найти K различных путей от исходного узла v_s до целевого узла v_d так, чтобы общая стоимость каждого пути была минимальной. Пусть $R_M(v_s, v_d)$ обозначается как набор всех различных путей, отсортированных в порядке возрастания по значению стоимости.

$$R_M(v_s, v_d) = \{P_1, P_2, \dots, P_M | w_{P_1} \leq w_{P_2} \leq \cdots \leq w_{P_M}\}.$$

Тогда традиционная задача поиска кратчайших путей без петель может быть определена как поиск подмножества $R_K(v_s, v_d) \subset R_M(v_s, v_d)$ следующим образом:

$$R_K(v_s, v_d) = \{P_1, P_2, \dots, P_K | w_{P_1} \leq w_{P_2} \leq \cdots \leq w_{P_K}\}. \quad (2.1)$$

На самом деле может существовать несколько путей одинаковой длины. Таким образом, может быть более K путей, удовлетворяющих условию (2.1). Следовательно,

определение подмножества R_K расширено. Все эти пути могут быть включены в множество кратчайших путей.

Выбор значения K обычно осуществляется администратором сети с учетом реального контекста системы. В небольших сетях, как правило, используются все возможные маршруты для максимального использования ресурсов. В более крупных сетях значение K обычно ограничивается диапазоном от 2 до 5, чтобы сбалансировать производительность сети и затраты на ресурсы, избегая чрезмерного использования путей, что приводит к высоким затратам на вычислительные ресурсы и память [33, 39, 42, 46]. В процессе исследования значение $K = 4$ было выбрано для соответствия практическим требованиям и обеспечения эффективной работы сети.

2.2 ИНТЕЛЛЕКТУАЛЬНАЯ МНОГОПУТЕВАЯ МАРШРУТИЗАЦИЯ В ПКС НА ОСНОВЕ РОЕВОГО ИНТЕЛЛЕКТА

2.2.1 РОЕВОЙ ИНТЕЛЛЕКТ

Роевой интеллект (РИ) – это область искусственного интеллекта, которая занимается разработкой и внедрением интеллектуальных мультиагентных систем, имитирующих коллективное поведение социальных насекомых и животных, таких как муравьи, пчелы, птицы и другие. Эти алгоритмы используют взаимодействие между простыми агентами внутри популяции для моделирования сложной динамики природных систем. В результате этого взаимодействия возникает коллективное разумное поведение без необходимости в централизованном контроле. Каждый агент действует на основе простых правил и локальной информации, однако совместно они способны решать сложные задачи, эффективно оптимизируя процессы. Такой подход особенно полезен для исследования пространств решений большого объема и

нахождения качественных решений там, где традиционные методы сталкиваются с трудностями из-за вычислительной сложности или масштаба задачи [51-54].

За последние два десятилетия роевой интеллект привлек большой интерес исследователей. Многие алгоритмы, основанные на роевом интеллекте, были внедрены в различные области информатики для оптимизации и завоевали большую популярность. Причинами успеха алгоритмов, основанных на роевом интеллекте, являются их динамичность и гибкость, а также высокая эффективность при решении нелинейных задач в реальном мире.

Термин «роевой интеллект» был впервые введен Бени в отношении клеточных роботизированных систем, где простые агенты используют принцип самоорганизации посредством взаимодействия ближайших соседей [55, 56].

Рой обычно рассматривается как группа схожих животных, обладающих качествами самоорганизации и децентрализованного управления. По данным Бонабо и др. типичными характеристиками роя являются следующие [57]:

- Рой состоит из множества особей, работающих коллективно, и обладает свойством самоорганизации.
- Особи роя однородны (идентичны), как, например, рой роботов или некоторые виды одних и тех же животных.
- Особи роя, по сравнению со способностями всего роя, обладают ограниченным интеллектом и меньшими способностями. Рой объединенных особей гораздо мощнее, чем отдельные члены.
- Особи в рое действуют и выполняют свои задачи в соответствии с установленными правилами и задачами.

Роевой интеллект добился успеха в различных областях благодаря своему уникальному подходу к решению проблем, который отражает сложное поведение, наблюдаемое в природе. Ключевыми причинами успеха роевого интеллекта являются:

- **Децентрализация**: Алгоритмы ролевого интеллекта работают без центрального органа управления, что снижает риск единой точки отказа. Такая децентрализованная структура обеспечивает большую гибкость и устойчивость в сложных условиях.
- **Самоорганизация**: Роевой интеллект позволяет системам организовываться через локальные взаимодействия и простые правила. Эта возникающая организация часто приводит к эффективным решениям без необходимости центральной координации.
- **Адаптируемость**: Системы на основе ролевого интеллекта могут адаптироваться к меняющимся условиям или окружающей среде. Эта адаптируемость имеет решающее значение в динамичных контекстах, где область задачи со временем изменяется или где присутствует неопределенность.
- **Устойчивость**: Поскольку роевой интеллект основывается на группе независимых агентов, он по своей природе устойчив. Если один агент выходит из строя или ведет себя непредсказуемо, система может продолжать работать без значительных сбоев.
- **Масштабируемость**: Алгоритмы ролевого интеллекта можно масштабировать для решения более крупных задач, добавляя больше агентов. Эта масштабируемость позволяет применять эти алгоритмы к широкому спектру задач, от простых до сложных.
- **Баланс между разведкой и эксплуатацией**: Роевой интеллект эффективно балансирует между разведкой (поиск новых решений) и эксплуатацией (уточнение известных решений). Этот баланс помогает избежать преждевременной сходимости и обеспечивает тщательный поиск решения.
- **Возникающее поведение**: Взаимодействие между агентами в рое может привести к возникающему поведению, которое решает сложные проблемы

новаторским образом. Это возникающее поведение позволяет находить решения, которые могут быть не очевидны при централизованном подходе.

- **Параллелизм**: Поскольку роевой интеллект включает в себя несколько агентов, работающих одновременно, он может использовать параллелизм для более быстрого решения проблем.

Децентрализованный характер, адаптивность и масштабируемость алгоритмов роевого интеллекта делают их идеальными для решения широкого спектра приложений: от оптимизации логистики и проектирования сетей до решения неотложных проблем в робототехнике и искусственном интеллекте. Поскольку технологии продолжают развиваться, эти алгоритмы могут получить более широкую интеграцию в различные области, используя их надежность и гибкость. Будущее алгоритмов роевого интеллекта может включать в себя гибридные подходы, сочетающие традиционные вычислительные методы с роевыми методами, что приведет к созданию более эффективных и отказоустойчивых систем.

2.2.2 МЕТОД КОДИРОВАНИЯ ПУТИ ДЛЯ ПРИМЕНЕНИЯ АЛГОРИТМОВ РОЕВОГО ИНТЕЛЛЕКТА

При использовании алгоритмов роевого интеллекта для решения задач маршрутизации применяются различные методы кодирования пути.

- Первый метод, описанный в [58], предполагает использование генетического алгоритма с хромосомами фиксированной длины, равной числу узлов в топологии. Каждая хромосома представляет собой последовательность целых чисел, случайно выбранных из списка узлов.

- Второй метод, предложенный в [59], также основан на генетическом алгоритме, но с хромосомами переменной длины. Хромосомы содержат целые числа, которые представляют узлы на маршруте. Первый ген всегда указывает на начальный узел, а последний ген – на конечный узел.

- Третий метод – это приоритетное кодирование [60]. В данном случае каждая хромосома имеет фиксированную длину и представляет собой последовательность значений приоритета, связанных с узлами. Построение пути начинается с начального узла: из рассмотрения исключаются узлы, соседние с текущим, и добавляется узел с наивысшим приоритетом. Процесс продолжается до тех пор, пока не будет достигнут конечный узел.

Для генетического алгоритма и оптимизации муравьиной колонии используется метод прямого кодирования пути, то есть каждое решение представляется в виде последовательности узлов от источника к пункту назначения. Этот подход соответствует описанному во втором методе.

Для алгоритмов стаи птиц, искусственной пчелиной колонии и светлячков используется метод приоритетного кодирования, так как в этих алгоритмах роевого интеллекта решения часто представляются в виде векторов действительных чисел. Эти действительные числа генерируются случайным образом как часть процесса поиска алгоритма, что позволяет эффективно исследовать пространство решений. Однако в контексте маршрутизации, особенно в ПКС, где пути маршрутизации определяются в сетевом графе, эти числовые значения не имеют прямого соответствия узлам или путям в сети. Это является основной причиной, по которой методы прямого кодирования, напрямую преобразующие результаты алгоритма в решения, неосуществимы.

При приоритетном кодировании, каждому узлу в сети назначается значение приоритета в диапазоне от -1.0 до 1.0. Во время маршрутизации, алгоритмы роевого интеллекта оценивают значения приоритета узла, чтобы определить следующий переход. Среди соседних узлов выбирается узел с наивысшим значением приоритета. Этот метод эффективно преобразует непрерывные или случайные результаты алгоритмов роевого интеллекта в дискретное решение, соответствующее структуре сети.

$$\gamma_i \in [-1.0, 1.0],\ i \in (1, 2, \dots, n),$$

$$next = \underset{j \in A_{cr}}{argmax}\, \gamma_j,$$

где γ_i – приоритет узла i, n – количество узлов в топологии, $next$ – следующий узел и A_{cr} – набор узлов, смежных с текущим узлом, не входящих в путь.

Процесс декодирования поясняется следующим примером. Для топологии сети на рисунке 2.2, каждый узел имеет значение приоритета, показанное на таблице 2.1.

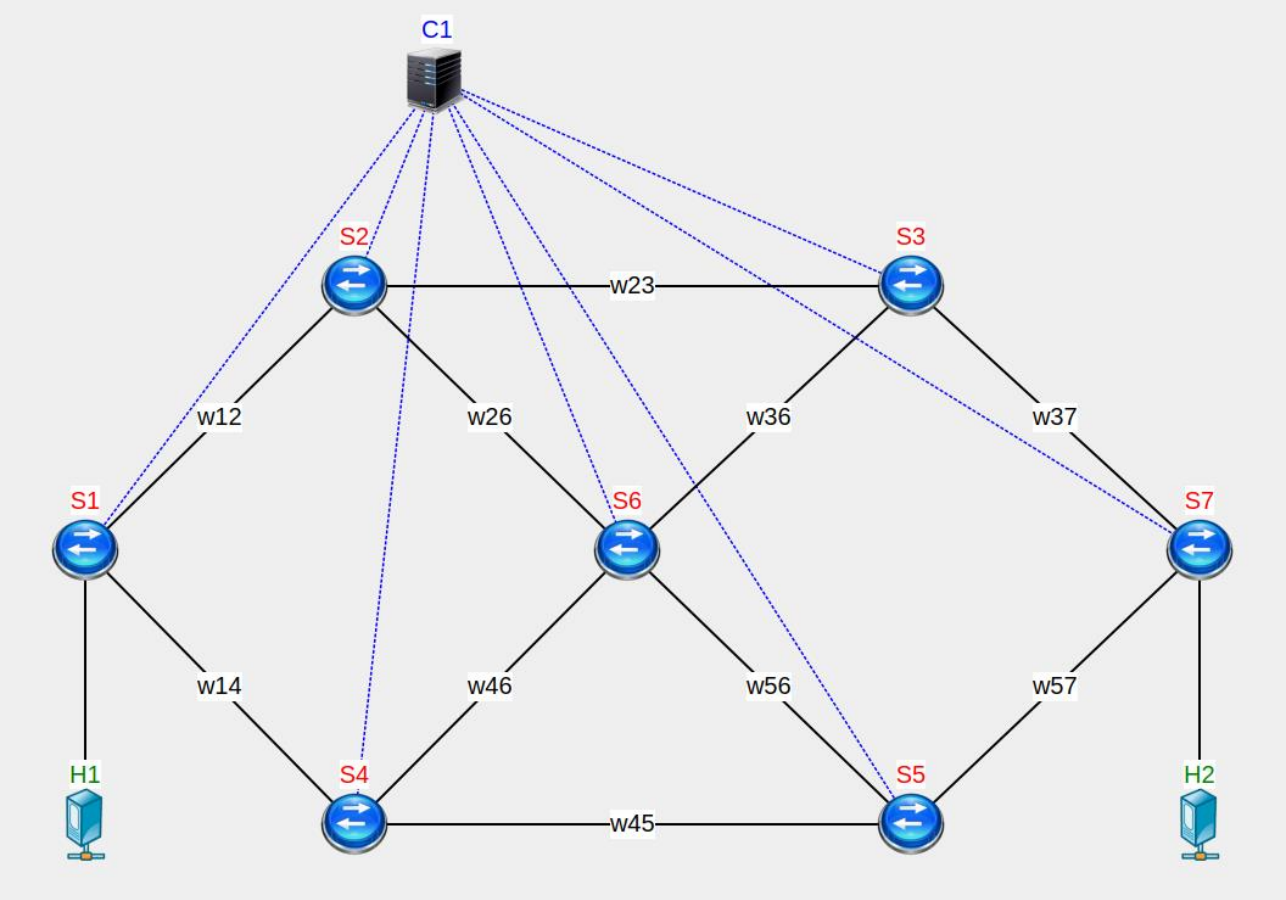


Рисунок 2.2 – Топология ПКС с 7 узлами

Таблица 2.1 – Приоритетное кодирование

Узел	1	2	3	4	5	6	7
Приоритет	-0.1	0.7	0.2	-0.2	-0.4	0.5	-0.3

Первоначально путь начинается с исходного узла [1]. От этого узла можно добавить к пути два соседних узла, 2 и 4. Узел 2 выбран, потому что он имеет наивысший приоритет, таким образом, путь расширяется до $[1, 2]$. Аналогично, из

узла 2, узлы 3 и 6 могут быть добавлены к пути, и выбирается узел 6. Узлы, соседствующие с узлом 6, которые путь еще не прошел, это узлы 3, 4 и 5. Узел 3 имеет наивысшее значение приоритета, поэтому он добавляется к пути. Узел 7 – единственный сосед узла 3, который путь еще не посетил, и он является узлом назначения, поэтому процесс расшифровки завершается. Таким образом, итоговый путь будет $[1, 2, 6, 3, 7]$. Процесс декодирования представлен в таблице 2.2.

Таблица 2.2 – Процесс декодирования

Текущий узел	**Соседние узлы**	**Приоритет**	**Следующий узел**	**Текущий путь**
1	**2**	**0.7**	2	1 – 2
	4	-0.2		
2	3	0.2	6	1 – 2 – 6
	6	**0.5**		
6	**3**	**0.2**	3	1 – 2 – 6 – 3
	4	-0.2		
	5	-0.4		
3	**7**	**-0.3**	7	1 – 2 – 6 – 3 – 7

Проблема заключается в том, что приоритетное кодирование может создавать недопустимые пути. Для решения этой проблемы применяется метод, заключающийся в присвоении высоких значений приспособленности этим недопустимым путям. В данной работе используется суммарная длина всех каналов. После присвоения высоких значений приспособленности недопустимые пути становятся менее привлекательными для процесса выбора в алгоритмах роевого интеллекта. Такой метод присвоения приспособленности способствует сосредоточению поисковой активности алгоритма на действительных и потенциально более эффективных путях, сокращая изучение нежелательных областей пространства решений.

2.2.3 ГЕНЕТИЧЕСКИЙ АЛГОРИТМ

Генетический алгоритм (Genetic Algorithm, GA) – это стохастический метод оптимизации, основанный на принципах эволюции и естественного отбора, предложенных Дарвином. Он был разработан в 1960-х годах и представлен Джоном Х. Холландом в 1975 году [61]. GA широко применяется в различных областях для решения сложных задач оптимизации, особенно там, где невозможно найти оптимальное решение напрямую из-за большого пространства поиска или нелинейного характера задачи. Моделируя процесс выживания особей в популяции, алгоритм постепенно улучшает решения, стремясь выявить особей с наилучшими характеристиками приспособленности, которые представляют собой оптимальное решение.

GA работает на основе трех основных этапов: скрещивания, мутации и отбора. В ходе эволюции, новые особи создаются за счет сочетания лучших характеристик существующих особей, при этом некоторые из них подвергаются небольшим мутациям, чтобы исследовать большее пространство возможных решений. Основная цель алгоритма – найти наилучшее или близкое к наилучшему решению для рассматриваемой задачи. Благодаря использованию биологически мотивированных операций, он также известен как биоинспирированный алгоритм роевого интеллекта.

Блок-схема генетического алгоритма для многопутевой маршрутизации в ПКС показана на рисунке 2.3.

Шаг 1. Инициализация начальных параметров

- N – число особей в популяции.
- Max – число итераций.
- P_c – вероятность скрещивания.
- P_m – вероятность мутации.

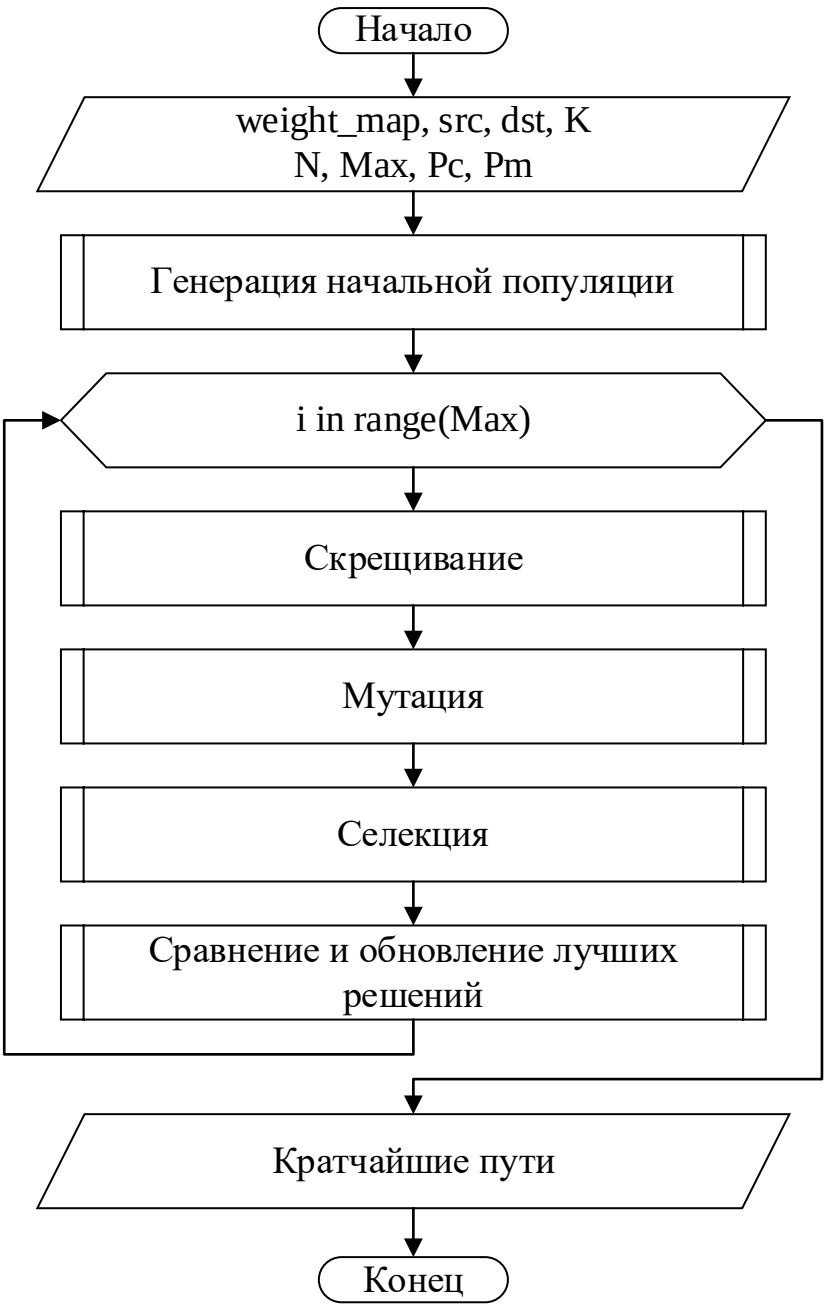


Рисунок 2.3 – Блок-схема генетического алгоритма для многопутевой маршрутизации в ПКС

Шаг 2. Генерация начальной популяции

Начальная популяция состоит из набора хромосом, где каждая хромосома представляет собой решение задачи. В задаче многопутевой маршрутизации каждая хромосома соответствует одному маршруту.

В данной работе применяется метод прямого кодирования пути с нефиксированной длиной. Каждый ген в хромосоме соответствует узлу на пути. Первый ген всегда соответствует исходному узлу, а последний – целевому.

Например: Хромосома C обозначается следующим образом:

$$C = [v_8, v_4, v_3, v_7, v_6, v_5],$$

где v_8 – исходный узел и v_5 – целевой узел.

Шаг 3. Оценка приспособленности

Приспособленность каждой особи измеряется по суммарному весу пути. Особям с лучшей приспособленностью (меньшим весом пути) дается приоритет при размножении, а особи с худшей приспособленностью имеют меньше шансов на воспроизведение.

Функция приспособленности определяется как:

$$f(C_i) = w(C_i),$$

где $w(C_i)$ вычисляет суммарный вес пути, соответствующего хромосоме C_i.

Шаг 4. Инициализация счетчика итерации

Установить начальное значение счетчика: $iter = 1$.

Шаг 5. Скрещивание

Скрещивание – это процесс создания потомков путем обмена частями генов двух родительских хромосом. Существует несколько различных методов скрещивания, таких как: одноточечный, многоточечный и равномерный.

В данной работе используется метод одноточечного скрещивания. Случайно выбранные родители должны удовлетворять условию наличия хотя бы одного общего гена, кроме первого и последнего генов.

Например:

Пусть для скрещивания выбраны две родительские хромосомы:

$$C_f = [v_8, v_4, v_3, \boldsymbol{v_7}, v_6, v_5],$$

$$C_m = [v_8, v_2, v_{10}, \boldsymbol{v_7}, v_9, v_{11}, v_5].$$

Точка скрещивания – v_7.

Точка скрещивания разделяет каждую родительскую хромосому на две части. Обмениваясь этими частями, образуются потомки C_{s1} и C_{s2}.

$$C_{s1} = [v_8, v_4, v_3, \boldsymbol{v_7}, v_9, v_{11}, v_5],$$

$$C_{s2} = [v_8, v_2, v_{10}, \boldsymbol{v_7}, v_6, v_5].$$

В процессе скрещивания могут возникнуть нежелательные потомки, соответствующие путям с петлями. В таких случаях необходимо удалить петлю для получения корректных решений.

Например:

$$C_s = [v_8, v_4, v_3, \boldsymbol{v_7}, \cancel{v_9}, \cancel{v_{11}}, \boldsymbol{v_7}, v_{10}, v_{13}, v_5],$$

$$\Rightarrow C_s = [v_8, v_4, v_3, \boldsymbol{v_7}, v_{10}, v_{13}, v_5].$$

Шаг 6. Мутация

Мутация – это процесс создания новых особей из потомков, полученных в процессе скрещивания. В генетических алгоритмах мутация используется для поиска новых решений и предотвращения застревания алгоритма в локальных оптимумах. Существуют различные методы мутации, такие как одноточечная и многоточечная мутация.

В данной работе применяется метод одноточечной мутации. Случайным образом выбирается ген в качестве точки мутации, который делит родительскую хромосому на две части. Первая часть сохраняется, а вторая часть обновляется, аналогично процессу инициализации хромосомы.

Например:

Пусть выбрана хромосома для мутации:

$$C_m = [v_8, v_9, v_{13}, \boldsymbol{v_{10}}, v_6, v_5].$$

Точка мутации – v_{10}.

Тогда новая хромосома, созданная после мутации, будет:

$$C_s = [v_8, v_9, v_{13}, \boldsymbol{v_{10}}, v_7, v_1, v_{12}, v_5].$$

Шаг 7. Селекция (Отбор)

Селекция – это процесс выбора хромосом, наиболее приспособленных для создания следующего поколения. Существует множество методов отбора, таких как турнирная селекция, метод рулетки, ранжирование, сигма-отсечение и другие.

В данной работе используется метод турнирной селекции. Для отбора случайным образом выбирается определенное количество хромосом (участников турнира). Хромосома с наилучшей приспособленностью среди выбранных участников будет добавлена в следующее поколение. Этот процесс повторяется до тех пор, пока не будет достигнуто требуемое количество хромосом в популяции.

Шаг 8. Сравнение и обновление лучших решений

После каждой итерации выбираются K лучших особей текущего поколения в качестве кандидатов. Каждый кандидат сравнивается с решениями из текущего набора K лучших решений ($Best$). Если кандидат имеет лучшую приспособленность и его решение ранее не присутствовало в наборе $Best$, то это решение добавляется в набор лучших решений.

Шаг 9. Увеличение значения счетчика итерации

Увеличить значение счетчика итерации на единицу: $iter = iter + 1$.

Шаг 10. Проверка условия остановки

- Если $iter < Max$, то вернуться к шагу 5.
- Если $iter = Max$, то перейти к шагу 11.

Шаг 11. Вывод кратчайших путей

После завершения итераций вывести набор лучших решений (кратчайших путей), сохраненных в наборе $Best$.

2.2.4 АЛГОРИТМЫ ОПТИМИЗАЦИИ МУРАВЬИНОЙ КОЛОНИИ

Оптимизация муравьиной колонии (Ant Colony Optimization, ACO) – это метаэвристический алгоритм, основанный на естественном поведении колоний

муравьев при поиске пищи. Этот алгоритм был предложен Марко Дориго в начале 1990-х годов и с тех пор стал мощным инструментом для решения комбинаторных задач оптимизации [62]. В природе, когда муравьи перемещаются, они оставляют химическое вещество, называемое феромоном, на своем пути, что помогает другим муравьям ориентироваться и находить более короткие маршруты. В ACO этот процесс моделируется с помощью виртуальных муравьев, которые передвигаются по вершинам графа, а ребра помечаются количеством феромона. Муравьи перемещаются от начального узла через ряд промежуточных узлов к целевому. В узле i муравей выберет узел j, который не был пройден в наборе соседей $E_{next}(i)$, в соответствии с вероятностью:

$$p_{i,j} = \frac{(\tau_{i,j})^{\alpha} \times (\eta_{i,j})^{\beta}}{\sum_{j \in E_{next}(i)} [(\tau_{i,j})^{\alpha} \times (\eta_{i,j})^{\beta}]}.$$

Эта формула означает, что выбор следующего узла для перехода муравьем осуществляется случайно, но с вероятностями, зависящими от концентрации феромонов на рассматриваемом пути (как у естественных муравьев) и обратно пропорционально длине пути.

Со временем более короткие маршруты начинают доминировать, поскольку на них накапливается больше феромона, что позволяет алгоритму постепенно сходиться к оптимальным или близким к оптимальным решениям. ACO получил широкое применение в различных областях, от задачи коммивояжера (TSP) и маршрутизации в сетях до сложных задач планирования.

Существуют различные модификации алгоритмов ACO, такие как муравьиная система, элитарная муравьиная система, ранговая муравьиная система, система муравьиной колонии и Max-Min муравьиная система. В данной работе рассмотрены два наиболее популярных алгоритма: алгоритм муравьиной системы (Ant System, AS) и алгоритм системы муравьиной колонии (Ant Colony System, ACS).

Блок-схемы алгоритма муравьиной системы и алгоритма системы муравьиной колонии для многопутевой маршрутизации в ПКС показаны на рисунках 2.4 и 2.5.

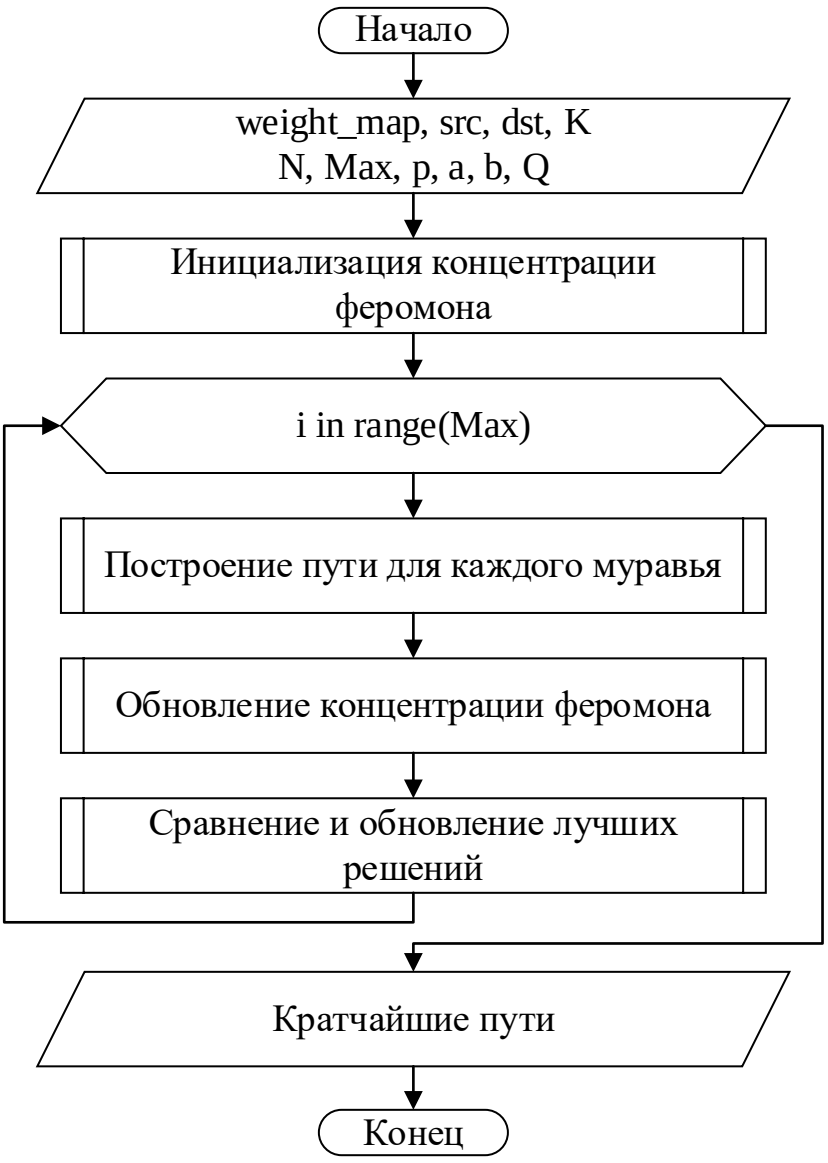


Рисунок 2.4 – Блок-схема алгоритма муравьиной системы для многопутевой маршрутизации в ПКС

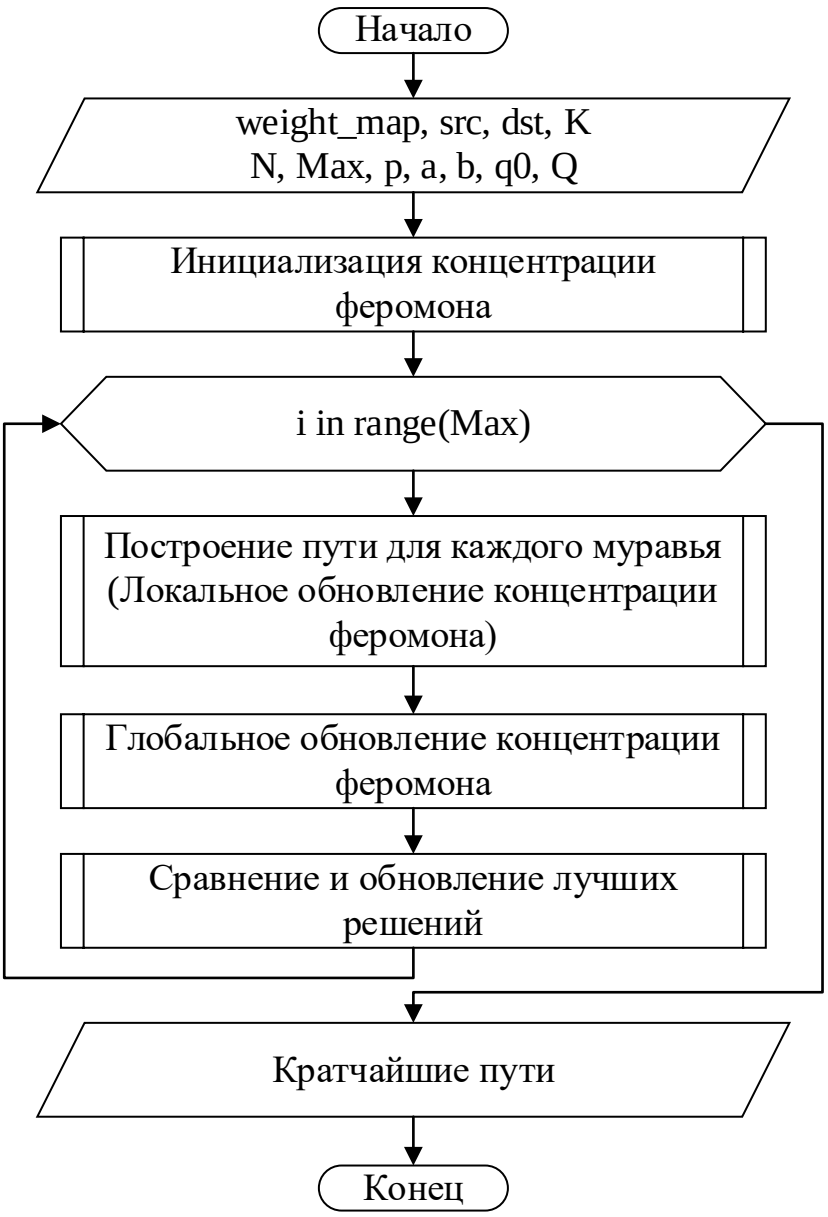


Рисунок 2.5 – Блок-схема алгоритма системы муравьиной колонии для многопутевой маршрутизации в ПКС

Алгоритм муравьиной системы

Шаг 1. Инициализация начальных параметров

- N – число муравьев.
- Max – число итераций.
- ρ – коэффициент испарения феромона.
- α – параметр, контролирующий влияние $\tau_{i,j}$.

- β – параметр, контролирующий влияние $\eta_{i,j}$.
- Q – интенсивность феромона.

где $\tau_{i,j}$ – концентрация феромона на ребре (i,j) и $\eta_{i,j}$ – привлекательность ребра (i,j).

Шаг 2. Инициализация концентрации феромона на всех ребрах

Изначально концентрация феромона на всех ребрах одинакова и задается по формуле:

$$\tau_0 = \frac{1}{n \times L_{nn}},$$

где n – это число узлов графа и L_{nn} – длина пути, найденного методом ближайшего соседа [63].

Шаг 3. Инициализация счетчика итераций

Задается начальное значение счетчика итераций: $iter = 1$.

Шаг 4. Построение пути для каждого муравья

- Построение пути начинается с исходного узла. Текущий узел (узел i) назначается исходным узлом.
- Определяется множество смежных узлов, не входящих в текущий путь $E_{next}(i)$.
- Муравей перемещается из узла i в узел j с вероятностью:

$$p_{i,j} = \frac{(\tau_{i,j})^{\alpha} \times (\eta_{i,j})^{\beta}}{\sum_{j \in E_{next}(i)} [(\tau_{i,j})^{\alpha} \times (\eta_{i,j})^{\beta}]}. \quad (2.2)$$

- Процесс продолжается до тех пор, пока муравей не достигнет целевого узла.

Шаг 5. Вычисление длины пути

Для каждого муравья вычисляется длина пути как сумма весов ребер, по которым муравей прошел.

Шаг 6. Обновление концентрации феромона

После того как все муравьи завершили свои пути, концентрация феромона на всех ребрах обновляется по формуле:

$$\tau_{i,j} = (1 - \rho) \times \tau_{i,j} + \sum_{k=1}^{N} \Delta\tau_{i,j}^{k},$$

$$\Delta\tau_{i,j}^{k} = \begin{cases} \frac{Q}{L_k}, \text{если } k\text{– й муравей двигается по ребру } (i,j) \\ 0, \text{ в противном случае} \end{cases},$$

где L_k – длина пути, пройденного k-м муравьем.

Шаг 7. Сравнение и обновление набора лучших решений ($Best$)

Шаг 8. Увеличение значения счетчика итерации

Увеличиваем значение счетчика итерации: $iter = iter + 1$.

Шаг 9. Проверка условия остановки

- Если $iter < Max$, то вернуться к шагу 4.
- Если $iter = Max$, то перейти к шагу 10.

Шаг 10. Вывод кратчайших путей

После завершения итераций вывести набор лучших решений (кратчайших путей), сохраненных в наборе $Best$.

Алгоритм системы муравьиной колонии

В задаче многопутевой маршрутизации ACS отличается от AS тремя аспектами: правилом перехода состояния, локальным обновлением концентрации феромона и глобальным обновлением концентрации феромона.

- Правило перехода состояния

Инициализируется случайное значение q.

- Если $q \leq q_0$, то муравей перемещается в узел e, который определяется по следующей формуле:

$$e = argmax_{j \in E_{next}(i)}[(\tau_{i,j})^{\alpha} \times (\eta_{i,j})^{\beta}],$$

где q_0 – параметр алгоритма ($0 \leq q_0 \leq 1$).

- Если $q > q_0$, то муравей перемещается в узел j с вероятностью, рассчитанной по формуле (2.2).

- Локальное обновление концентрации феромона

В процессе построения пути каждого муравья, когда муравей проходит через ребро (i,j), феромон на этом ребре обновляется следующим образом:

$$\tau_{i,j} = (1-\rho) \times \tau_{i,j} + \rho \times \Delta\tau_{i,j} ,$$

где $\Delta\tau_{i,j}$ – параметр, определяемый экспериментально. В данной работе выбрано $\Delta\tau_{i,j} = \tau_0$.

- Глобальное обновление концентрации феромона

После того, как все муравьи завершили свои пути, феромон обновляется только на ребрах кратчайшего пути:

$$\tau_{i,j} = (1-\rho) \times \tau_{i,j} + \rho \times \Delta\tau_{i,j}^{best} ,$$

$$\Delta\tau_{i,j}^{best} = \frac{Q}{L_{best}} ,$$

где L_{best} – длина кратчайшего пути, найденного муравьями на текущей итерации.

2.2.5 АЛГОРИТМ СТАИ ПТИЦ

Алгоритм стаи птиц (Bird Flock Algorithm, BFA) представляет собой популярный метод оптимизации, основанный на социальном поведении птиц. Этот метод был разработан в 1995 году Джеймсом Кеннеди и Расселом Эберхартом и применяется в разнообразных областях, включая искусственный интеллект, инженерию и экономику, для поиска оптимальных решений в сложных пространствах поиска [64]. При использовании BFA для решения задачи оптимизации каждое решение проблемы представляет собой позицию птицы в поисковом пространстве. Популяция решений аналогична стае птиц, где каждая птица обладает скоростью, соответствующей ее текущей позиции. Как только птица достигает новой позиции, обновляются как лучшая позиция каждой отдельной птицы (локальная лучшая позиция), так и лучшая позиция всей стаи (глобальная лучшая позиция). Затем новая скорость птицы рассчитывается на основе локальных и глобальных лучших позиций. Это позволяет птицам стремиться к лучшим местам в пространстве поиска. Процесс повторяется до тех пор, пока не будет достигнут критерий остановки.

Блок-схема BFA для многопутевой маршрутизации в ПКС представлена на рисунке 2.6.

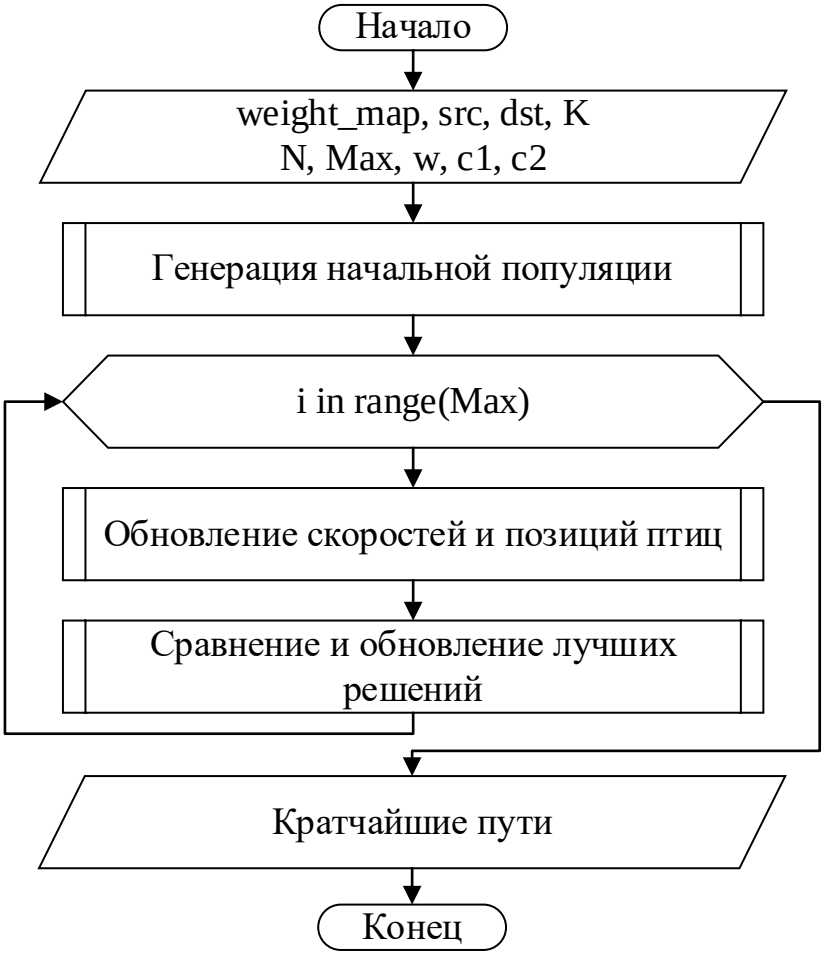


Рисунок 2.6 – Блок-схема алгоритма стаи птиц для многопутевой маршрутизации в ПКС

Шаг 1. Инициализация начальных параметров

- N – число птиц (размер популяции).
- Max – число итераций.
- w – инерционный вес.
- c_1, c_2 – когнитивный и социальный коэффициенты.

Шаг 2. Инициализация позиций и скоростей птиц

Начальная популяция представляется следующим образом:

$$X = [x_1, x_2, \dots, x_N\ ].$$

Начальная позиция и скорость i-ой птицы $(i = 1, 2, \dots, N)$ генерируются случайным образом и описываются вектором:

$$x_i = [x_{i,1}, x_{i,2}, \dots, x_{i,n}],$$
$$v_i = [v_{i,1}, v_{i,2}, \dots, v_{i,n}],$$
$$j \in (1, 2, \dots, n),$$

где $x_{i,j} \in [-1.0, 1.0]$ и $v_{i,j} \in [-0.5, 0.5]$, а n – это число узлов графа.

Шаг 3. Оценка приспособленности

Функция приспособленности для i-ой птицы определяется как:

$$f(x_i) = w(P_{x_i}),$$

где $P_{x_i} = decode(x_i)$ – это путь, полученный в результате декодирования позиции x_i и $w(P_{x_i})$ вычисляет значение длины пути P_{x_i}.

Если путь невозможно построить (например, из-за тупика или наличия циклов), то такой путь считается невалидным, и приспособленность данного решения устанавливается на максимальное значение, чтобы оно не могло быть выбрано в качестве оптимального решения.

Шаг 4. Инициализация текущей локальной и глобальной лучшей позиции

$$p_{local} = [p_1^{local}, p_2^{local}, \dots, p_N^{local}],$$
$$p_{global} = \underset{p_i^{local} \in p_{local}}{argmin} f(p_i^{local}),$$

где p_i^{local}, p_{global} – текущая лучшая позиция для i-ой птицы и для всей популяции.

Шаг 5. Инициализация начального значения счетчика итераций

Инициализируется значение счетчика итераций: $iter = 1$.

Шаг 6. Обновление скоростей и позиций птиц

- Скорость i-ой птицы на $iter$-ой итерации обновляется по следующей формуле:

$$v_i^{iter} = w \times v_i^{iter-1} + c_1 \times r_1 \times \left(p_i^{local} - x_i^{iter-1}\right) + c_2 \times r_2 \times \left(p_{global} - x_i^{iter-1}\right),$$

где r_1, r_2 – случайные числа в интервале $(0, 1)$.

После обновления скорость ограничивается заданным диапазоном:

$$v_i^{iter} = clip(v_i^{iter}, -0.5, 0.5).$$

Это предотвращает слишком резкие изменения скорости и неконтролируемые перемещения птиц в пространстве решений.

- Затем позиция i-ой птицы на $iter$-ой итерации определяется следующим образом:

$$x_i^{iter} = x_i^{iter-1} + v_i^{iter}.$$

Чтобы позиции всех птиц находились в пределах допустимого диапазона $[-1, 1]$, новые значения позиций нормализуются по следующей формуле:

$$x_i^{iter} = -1 + \frac{2\times(x_i^{iter} - min(x_i^{iter}))}{max(x_i^{iter}) - min(x_i^{iter})},$$

где $min(x_i^{iter})$ и $max(x_i^{iter})$ – минимальные и максимальные значения элементов кода позиции птицы на текущей итерации соответственно.

- Вычисляется приспособленность для новой позиции x_i^{iter}.

  - Если $f(x_i^{iter}) \le f(p_i^{local})$, тогда текущая лучшая позиция для i-ой птицы обновляется:

$$p_i^{local} = x_i^{iter}.$$

  - Если $f(p_i^{local}) \le f(p_{global})$, то текущая глобальная лучшая позиция для всех птиц обновляется:

$$p_{global} = p_i^{local}.$$

Шаг 7. Сравнение и обновление набора лучших решений ($Best$)

Шаг 8. Увеличение значения счетчика итерации

Увеличивается значение счетчика итерации: $iter = iter + 1$.

Шаг 9. Проверка условия остановки

- Если $iter < Max$, то вернуться к шагу 6.
- Если $iter = Max$, то перейти к шагу 10.

Шаг 10. Вывод кратчайших путей

По завершении работы алгоритма выводятся лучшие найденные решения в виде последовательностей узлов и соответствующих им длин путей. Эти решения представляют собой K лучших маршрутов, найденных в процессе работы алгоритма.

2.2.6 АЛГОРИТМ ИСКУССТВЕННОЙ ПЧЕЛИНОЙ КОЛОНИИ

Алгоритм искусственной пчелиной колонии (Artificial Bee Colony algorithm, ABC) основан на моделировании поведения пчел при поиске нектара [65]. В природе пчелиная колония состоит из трех основных групп пчел: рабочих пчел, пчел-наблюдателей и пчел-разведчиков. Сначала все источники пищи случайным образом обнаруживаются пчелами-разведчиками. Затем рабочие пчелы выполняют задачу сбора нектара из выбранных источников пищи. В это время пчелы-наблюдатели следят за поведением рабочих пчел, наблюдая за их «танцем», чтобы выбрать наилучшие источники пищи. Со временем запасы нектара в некоторых источниках пищи истощаются, и рабочие пчелы, ответственные за эти источники, превращаются в пчел-разведчиков, чтобы искать новые, более перспективные источники.

В алгоритме ABC каждое решение ассоциируется с местоположением источника пищи, а количество нектара в источнике пищи соответствует качеству решения. Количество рабочих пчел равно количеству источников пищи.

Блок-схема алгоритма искусственной пчелиной колонии для многопутевой маршрутизации в ПКС показана на рисунке 2.7.

Шаг 1. Инициализация начальных параметров

- N – число пчел в каждой фазе (размер популяции).
- Max – число итераций.
- $limit$ – предел безуспешных попыток обновления решения, после которого пчела-разведчик заменяет решение.

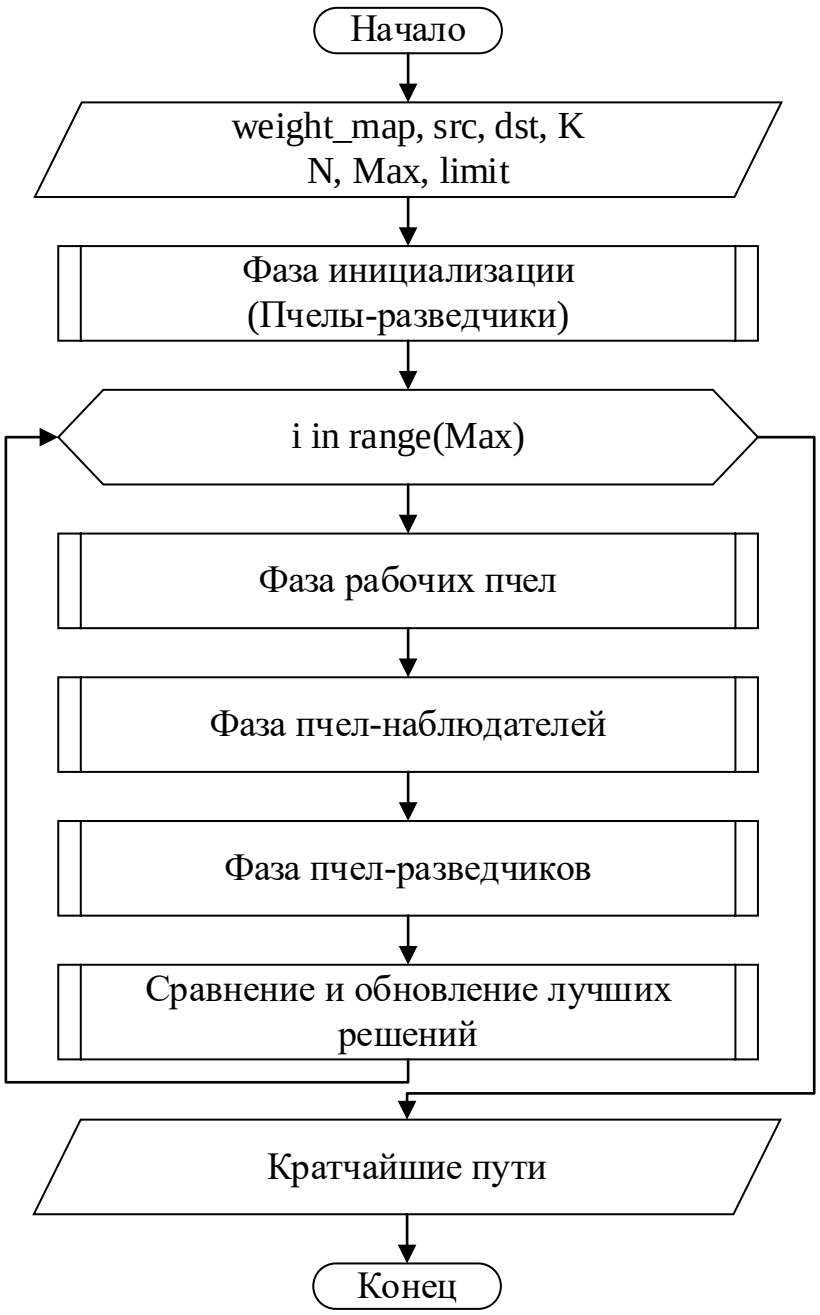


Рисунок 2.7 – Блок-схема алгоритма искусственной пчелиной колонии для многопутевой маршрутизации в ПКС

Шаг 2. Генерация начальной популяции (Фаза инициализации)

Начальная популяция источников пищи генерируется пчелами-разведчиками и представляется следующим образом:

$$X = [x_1, x_2, \dots, x_N\ ].$$

Каждый источник пищи x_i $(i = 1,2, \dots, N)$ является решением задачи оптимизации.

$$x_i = \left[x_{i,1}, x_{i,2}, \dots, x_{i,n}\right],$$

$$j \in (1, 2, \dots, n),$$

где $x_{i,j} \in [-1.0, 1.0]$, а n – это число узлов графа.

Шаг 3. Инициализация счетчика итераций

Инициализируется начальное значение счетчика итераций: $iter = 1$.

Шаг 4. Инициализация счетчиков для пчел-разведчиков

Для каждой пчелы задаются начальные значения счетчиков:

$$counter = [c_1, c_2, \dots, c_N],$$

где $c_i = 0$ и $i \in (1,2, \dots, N)$.

Шаг 5. Оценка популяции

Целевая функция для решения x_i определяется как:

$$f(x_i) = w(P_{x_i}),$$

где $P_{x_i} = decode(x_i)$ – это путь, полученный в результате декодирования решения x_i и $w(P_{x_i})$ вычисляет значение длины пути P_{x_i}.

Затем функция приспособленности рассчитывается по формуле:

$$fit(x_i) = \frac{1}{1+f(x_i)}.$$

Шаг 6. Фаза рабочих пчел

На этом этапе каждая рабочая пчела выбирает текущий источник пищи и ищет новый источник в его окрестности. Если новый источник пищи имеет более высокое значение приспособленности, он заменяет старый источник.

Для создания нового источника пищи используется следующая формула:

$$x_{i,j}^{new} = x_{i,j} + \Phi_i \times (x_{i,j} - x_{d,j}),$$

где $d \in \{1, 2, \dots, N\}$ и $j \in \{1, 2, \dots, n\}$ – случайно выбранные индексы, а Φ_i – случайное число в диапазоне $[-1.0, 1.0]$.

Для нормализации значений используется метод *Min-Max*:

$$x_i^{new} = -1 + \frac{2\times(x_i^{new} - min(x_i^{new}))}{max(x_i^{new}) - min(x_i^{new})},$$

где $min(x_i^{new})$ и $max(x_i^{new})$ – нижняя и верхняя границы параметра x_i^{new} соответственно.

- Если $fit(x_i^{new}) \geq fit(x_i)$, то новый источник пищи сохраняется, т.е. $x_i = x_i^{new}$ и счетчик c_i присваивается ноль ($c_i = 0$).
- Если $fit(x_i^{new}) < fit(x_i)$, то старый источник пищи сохраняется, т.е. x_i неизменяется и счетчик c_i увеличивается на единицу ($c_i = c_i + 1$).

Шаг 7. Фаза пчел-наблюдателей

После фазы рабочих пчел информация о качестве источников пищи передается пчелам-наблюдателям. На этом этапе пчелы-наблюдатели выбирают источник пищи с помощью метода рулетки. Вероятность выбора источника пищи пчелой рассчитывается по следующей формуле:

$$p_i = \frac{fit(x_i)}{\sum_{i=1}^{N} fit(x_i)}.$$

Затем пчелы-наблюдатели обыскивают окрестности выбранного источника пищи аналогично рабочим пчелам.

Шаг 8. Фаза пчел-разведчиков

Пчелы, которые случайным образом выбирают новые источники пищи, называются пчелами-разведчиками. Если источник пищи не обновлялся более ограниченного числа попыток ($c_i > limit$), то этот источник пищи считается застрявшим в локальном минимуме. В этом случае рабочая пчела, отвечающая за этот источник, превращается в пчелу-разведчика и ищет новый источник пищи в пространстве поиска.

Шаг 9. Сравнение и обновление набора лучших решений ($Best$)

Шаг 10. Увеличение значения счетчика итерации

Увеличиваем значение счетчика итерации: $iter = iter + 1$.

Шаг 11. Проверка условия остановки

- Если $iter < Max$, то вернуться к шагу 6.
- Если $iter = Max$, то перейти к шагу 12.

Шаг 12. Вывод кратчайших путей

В конце алгоритма выводятся найденные кратчайшие пути.

2.2.7 АЛГОРИТМ СВЕТЛЯЧКОВ

Алгоритм светлячков (Firefly Algorithm, FA) – это метаэвристический алгоритм, разработанный для решения задач оптимизации и вдохновленный природным поведением светлячков [66]. Основная идея алгоритма заключается в имитации мигания светлячков ночью. В основе алгоритма лежат три ключевых правила:

- Светлячок с меньшей яркостью (худшим решением) всегда привлекается к более яркому светлячку (лучшему решению).
- Яркость каждого светлячка определяется значением целевой функции, которую он оптимизирует.
- Привлекательность светлячка прямо пропорциональна его яркости, но уменьшается с увеличением расстояния между светлячками. Это приводит к тому, что светлячки движутся к более ярким соседям, улучшая свои решения.

Блок-схема алгоритма светлячков для многопутевой маршрутизации в ПКС показана на рисунке 2.8.

Шаг 1. Инициализация начальных параметров

- N – число светлячков (размер популяции).
- Max – число итераций.
- γ – коэффициент поглощения света (снижение привлекательности в зависимости от расстояния).
- β_0 – базовый коэффициент яркости светлячков.

- α_0 – коэффициент случайности.

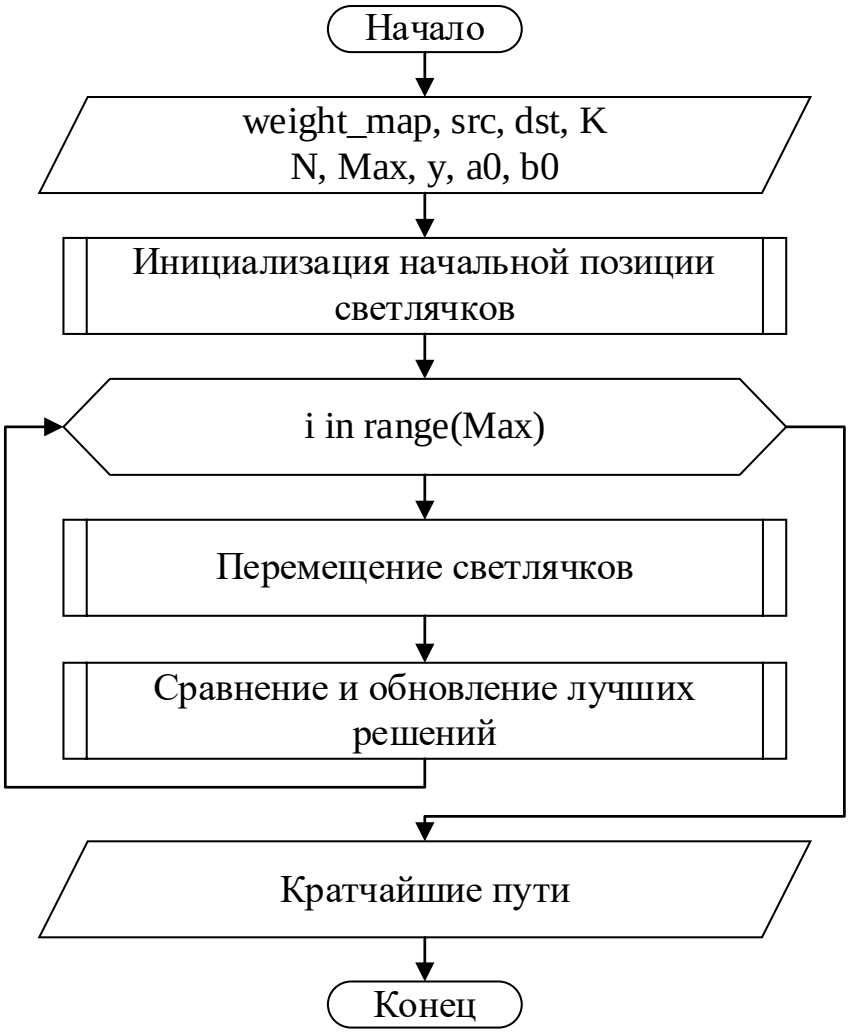


Рисунок 2.8 – Блок-схема алгоритма светлячков для многопутевой маршрутизации в ПКС

Шаг 2. Инициализация начальных позиций светлячков

Начальная популяция представляется в виде:

$$X = [x_1, x_2, \dots, x_N\ ].$$

Позиция i-го светлячка $(i = 1, 2, \dots, N)$ генерируется случайным и представляется следующим образом:

$$x_i = \left[x_{i,1}, x_{i,2}, \dots, x_{i,n}\ \right],$$

$$j \in (1, 2, \dots, n),$$

где $x_{i,j} \in [-1.0, 1.0]$ и n – число узлов графа.

Шаг 3. Оценка яркости (приспособленности) светлячков

Яркость каждого светлячка определяется как:

$$f(x_i) = w(P_{x_i}),$$

где $P_{x_i} = decode(x_i)$ – это путь, полученный в результате декодирования позиции x_i и $w(P_{x_i})$ вычисляет значение длины пути P_{x_i}.

Шаг 4. Инициализация счетчика итераций

Инициализируется начальное значение счетчика итераций: $iter = 1$.

Шаг 5. Уменьшение коэффициента случайности

Коэффициент случайности α_0 уменьшается после каждой итерации по формуле:

$$\alpha = \alpha_0 \times \theta^{iter},$$

где $\theta \in [0.9, 0.99]$ – скорость убывания α.

Шаг 6. Перемещение светлячков

for $i = 1: N$

for $j = 1: N$

Подшаг 6.1. Вычисление размера случайного шага

$$A_{ij} = \left[A_{ij}^1, A_{ij}^2, \dots, A_{ij}^n\right],$$

$$A_{ij}^k = 2 \times \alpha \times (\epsilon - \tfrac{1}{2}),$$

где $k \in (1, 2, \dots, n)$ и ϵ – случайное число в интервале $(0, 1)$.

if $(f(x_j) < f(x_i))$

Подшаг 6.2. Вычисление расстояния между светлячками i и j

$$r_{ij} = \sqrt{\sum_{d=1}^{n}(x_{i,d} - x_{j,d})^2}.$$

Подшаг 6.3. Оценка привлекательности между светлячками i и j

$$B_{ij} = \beta_0 \times e^{-\gamma \times {r_{ij}}^2}.$$

Подшаг 6.4. i-ый светлячок перемещается к j-ому светлячку

$$x_i = x_i + B_{ij} \times \left(x_j - x_i\right) + A_{ij}.$$

else

Подшаг 6.5. Случайное движение

$$x_i = x_i + A_{ij}.$$

end if

Подшаг 6.6. Нормализация позиции светлячков

$$x_i = -1 + \frac{2\times(x_i - min(x_i))}{max(x_i) - min(x_i)},$$

где $min(x_i)$ и $max(x_i)$ – минимальные и максимальные значения позиции светлячков после перемещения.

Подшаг 6.7. Обновление яркости i-го светлячка

end for j

end for i

Шаг 7. Сравнение и обновление набора лучших решений ($Best$)

Шаг 8. Увеличение значения счетчика итерации

Счетчик итераций увеличивается на единицу: $iter = iter + 1$.

Шаг 9. Проверка условия остановки

- Если $iter < Max$, то вернуться к шагу 5.
- Если $iter = Max$, то перейти к шагу 10.

Шаг 10. Вывод кратчайших путей

После завершения всех итераций алгоритм выводит найденные кратчайшие пути с наименьшими значениями приспособленности.

2.3 НЕЙРОСЕТЕВАЯ МНОГОПУТЕВАЯ МАРШРУТИЗАЦИЯ В ПКС НА ОСНОВЕ РОЕВОГО ИНТЕЛЛЕКТА

2.3.1 РЕКУРРЕНТНЫЕ НЕЙРОННЫЕ СЕТИ

Рекуррентные нейронные сети (Recurrent Neural Network, RNN) – это тип искусственных нейронных сетей, предназначенных для обработки последовательных данных, таких как временные ряды, текст, аудио или видео. Основное отличие RNN от традиционных нейронных сетей заключается в их способности сохранять информацию из предыдущих шагов, что позволяет им обрабатывать данные с циклической или временной зависимостью. Это достигается за счет обратных связей, при которых выход одного шага используется как вход для следующего [118].

A. SimpleRNN

SimpleRNN (Простая рекуррентная нейронная сеть) – это базовая форма рекуррентной нейронной сети, предназначенная для обработки последовательных данных, таких как временные ряды или последовательности символов. SimpleRNN имеет способность сохранять информацию из предыдущих временных шагов через скрытое состояние и использовать эту информацию для предсказания следующего шага. Работа сети основана на повторяющемся обновлении скрытых состояний на каждом временном шаге [119].

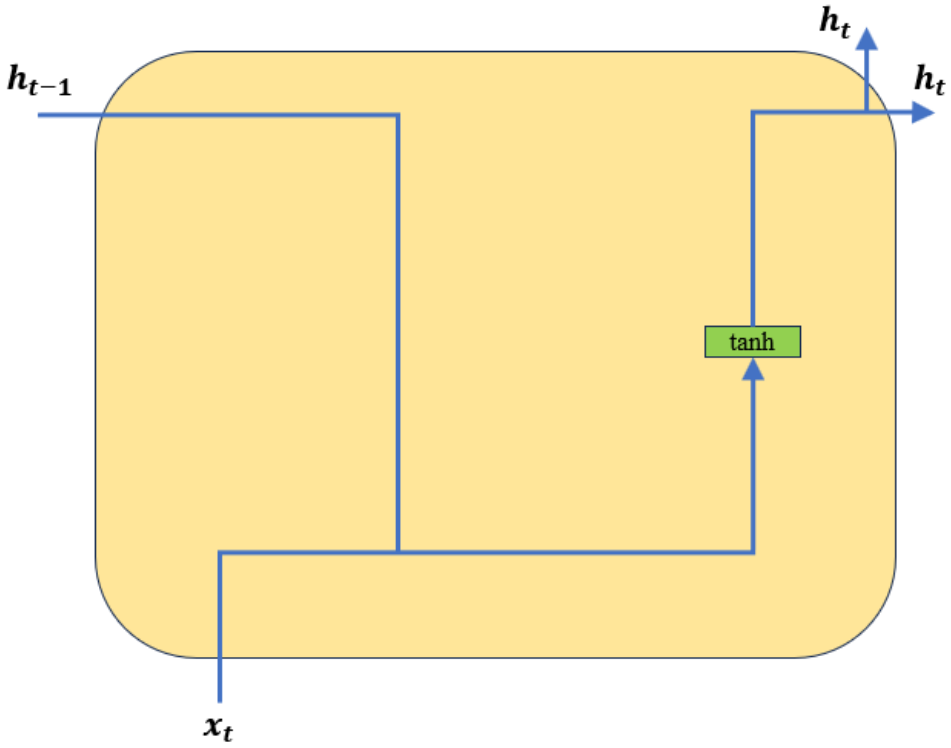


Рисунок 2.9 – Ячейка SimpleRNN

- **Скрытое состояние**

В каждый момент времени t, SimpleRNN обновляет скрытое состояние h_t на основе текущего входного значения x_t и скрытого состояния из предыдущего шага h_{t-1}. Общая формула для скрытого состояния:

$$h_t = tanh(W_h \times x_t + U_h \times h_{t-1} + b_h),$$

где:

- $x_t \in R^{n_x}$: входной вектор в момент времени t.
- $h_t \in R^{n_h}$: скрытое состояние в момент времени t.
- $W_h \in R^{n_h \times n_x}$: матрица весов, применяемая к входу x_t.
- $U_h \in R^{n_h \times n_h}$: матрица весов, применяемая к скрытому состоянию из предыдущего шага h_{t-1}.
- $b_h \in R^{n_h}$: вектор смещения для скрытого состояния.
- $tanh$: функция активации гиперболического тангенса.

- **Выход** (Дополнительно)

Выход y_t на каждом временном шаге t вычисляется на основе скрытого состояния h_t и матрицы весов W_y:

$$y_t = \phi\left(W_y \times h_t + b_y\right),$$

где:

- $y_t \in R^{n_y}$: выходной вектор в момент времени t.
- $W_y \in R^{n_y \times n_h}$: матрица весов, соединяющая скрытое состояние с выходным.
- $b_y \in R^{n_y}$: вектор смещения для выхода.
- ϕ: функция активации (например, *softmax* для задач классификации с несколькими классами).

B. LSTM

LSTM (Long Short-Term Memory или сеть долгой краткосрочной памяти), является улучшенной версией рекуррентной нейронной сети, разработанной для решения проблемы исчезающего и взрывающегося градиента, с которой

сталкиваются обычные RNN (например, SimpleRNN) при обработке длинных последовательностей данных. LSTM была представлена Хохрейтером и Шмидхубером в 1997 году и способна эффективно запоминать как краткосрочные, так и долгосрочные зависимости благодаря своей уникальной архитектуре, включающей три управляющих компонента, или ворот (*gates*), которые контролируют поток информации через каждую ячейку [120].

Состояние ячейки C_t хранит долгосрочную информацию. Это ключевое отличие от SimpleRNN, где есть только скрытое состояние. Состояние ячейки регулируется воротами, которые либо сохраняют, либо отбрасывают информацию.

LSTM работает на основе трех основных ворот: ворота забывания (*forget gate*), ворота входа (*input gate*) и ворота выхода (*output gate*).

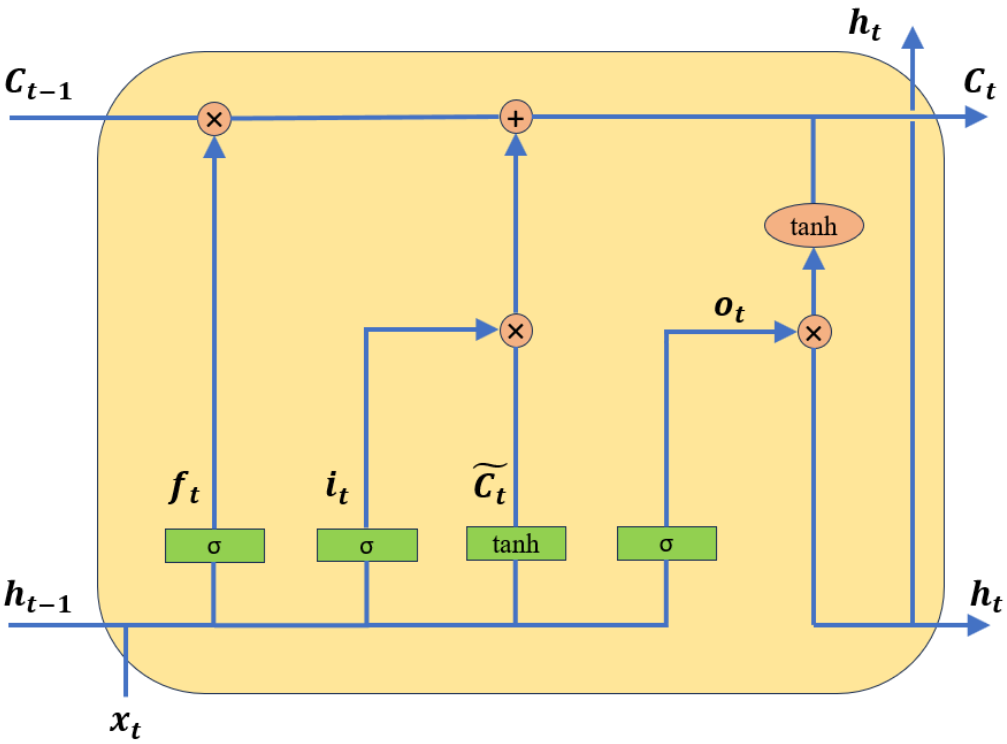


Рисунок 2.10 – Ячейка LSTM

- **Ворота забывания:** Определяют, какую часть информации из предыдущего состояния ячейки C_{t-1} необходимо сохранить или забыть. Формула для ворот забывания:

$$f_t = \sigma\big(W_f \times x_t + U_f \times h_{t-1} + b_f\big),$$

где:

- $f_t \in R^{n_h}$: значение ворот забывания на шаге t.
- $W_f \in R^{n_h \times n_x}$: матрица весов, применяемая к входу x_t.
- $U_f \in R^{n_h \times n_h}$: матрица весов, применяемая к предыдущему скрытому состоянию h_{t-1}.
- $b_f \in R^{n_h}$: вектор смещения ворот забывания.
- σ: сигмоидная функция активации, значения которой находятся в диапазоне от 0 до 1.

- **Ворота входа:** Определяют, сколько новой информации из текущего входа x_t будет добавлено в состояние ячейки. Формула для ворот входа:

$$i_t = \sigma(W_i \times x_t + U_i \times h_{t-1} + b_i),$$
$$\widetilde{C_t} = tanh(W_C \times x_t + U_C \times h_{t-1} + b_C),$$

где:

- $i_t \in R^{n_h}$: значение ворот входа.
- $\widetilde{C_t} \in R^{n_h}$: потенциальное новое состояние ячейки.
- $W_i, W_C \in R^{n_h \times n_x}$: матрицы весов, применяемые к входу x_t для ворот входа и потенциального состояния ячейки.
- $U_i, U_C \in R^{n_h \times n_h}$: матрицы весов, применяемые к предыдущему скрытому состоянию h_{t-1}.
- $W_y \in R^{n_y \times n_h}$: матрица весов, соединяющая скрытое состояние с выходным.
- $b_i, b_C \in R^{n_h}$: соответствующие векторы смещения.

- **Ворота выхода:** Определяют, как будет использована информация из состояния ячейки для генерации нового скрытого состояния h_t. Формула для ворот выхода:

$$o_t = \sigma(W_o \times x_t + U_o \times h_{t-1} + b_o),$$
$$h_t = o_t \odot \tanh(C_t),$$

где:

- $o_t \in R^{n_h}$: значение ворот выхода.
- $h_t \in R^{n_h}$: новое скрытое состояние.
- $W_o \in R^{n_h \times n_x}$: матрица весов от входа x_t к воротам выхода.
- $U_o \in R^{n_h \times n_h}$: матрица весов от предыдущего скрытого состояния h_{t-1}.
- $W_y \in R^{n_y \times n_h}$: матрица весов, соединяющая скрытое состояние с выходным.
- $b_o \in R^{n_h}$: вектор смещения для ворот выхода.
- $\odot$: поэлементное умножение.

Новое состояние ячейки C_t вычисляется на основе информации от ворот забывания и ворот входа следующим образом:

$$C_t = f_t \odot C_{t-1} + i_t \odot \widetilde{C_t},$$

где:

- $f_t \odot C_{t-1}$: часть состояния ячейки с предыдущего шага, которая сохраняется.
- $i_t \odot \widetilde{C_t}$: новая информация, добавленная в состояние ячейки.

C. GRU

GRU (Gated Recurrent Unit или управляемый рекуррентный блок) – это упрощенная версия рекуррентной нейронной сети, разработанная как альтернатива LSTM. GRU имеет более простую структуру и использует меньше параметров по сравнению с LSTM, но при этом эффективно решает проблему исчезающего градиента и обрабатывает долгосрочные зависимости в последовательных данных. GRU была представлена Кёнхёном Чо и его коллегами в 2014 году [121].

GRU, как и LSTM, использует механизм ворот для управления потоком информации через сеть. Однако в GRU есть только два типа ворот: ворота обновления (*update gate*) и ворота сброса (*reset gate*). В отличие от LSTM, в GRU нет отдельного состояния ячейки, информация хранится только в скрытом состоянии.

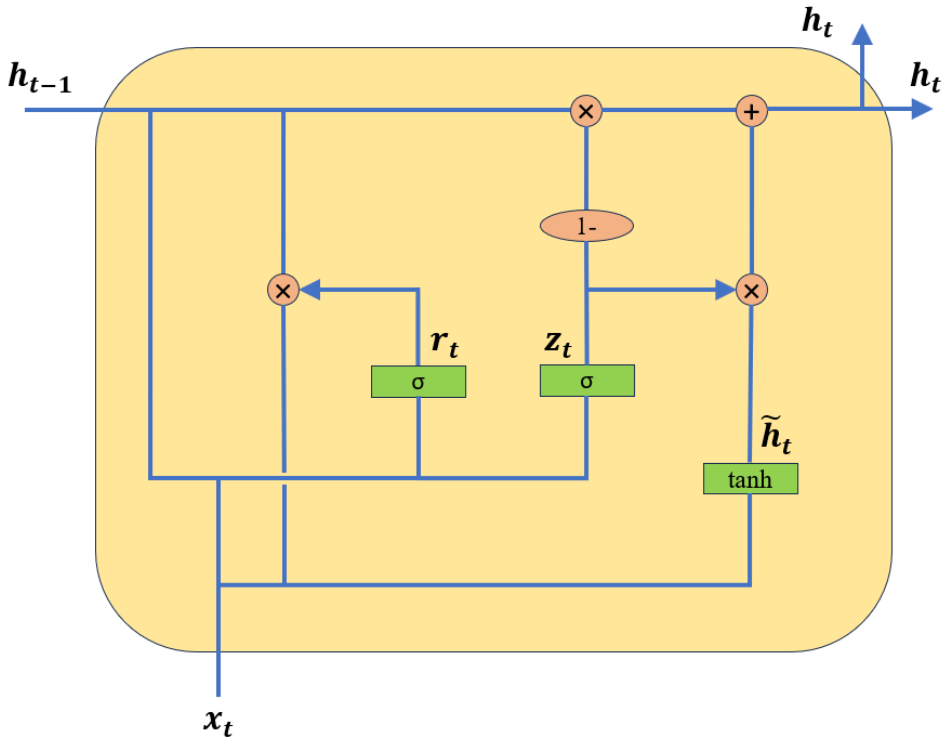


Рисунок 2.11 – Ячейка GRU

- **Ворота обновления**

Ворота обновления z_t в GRU контролируют, какая часть информации из предыдущего скрытого состояния h_{t-1} должна быть сохранена или обновлена новой информацией из текущего входа x_t. Формула для ворот обновления:

$$z_t = \sigma(W_z \times x_t + U_z \times h_{t-1} + b_z),$$

где:

- $z_t \in R^{n_h}$: значение ворот обновления в момент времени t.
- $W_z \in R^{n_h \times n_x}$: матрица весов, применяемая к входу x_t.
- $U_z \in R^{n_h \times n_h}$: матрица весов, применяемая к предыдущему скрытому состоянию h_{t-1}.
- $b_z \in R^{n_h}$: вектор смещения ворот обновления.

- **Ворота сброса**

Ворота сброса r_t контролируют, сколько информации из предыдущего скрытого состояния h_{t-1} нужно «сбросить» при вычислении нового потенциального состояния

$\bar{h}_{t-1}$. Формула для ворот сброса:

$$r_t = \sigma(W_r \times x_t + U_r \times h_{t-1} + b_r),$$

где:

- $r_t \in R^{n_h}$: значение ворот сброса в момент времени t.
- $W_r \in R^{n_h \times n_x}$: матрица весов, применяемая к входу x_t.
- $U_r \in R^{n_h \times n_h}$: матрица весов, применяемая к предыдущему скрытому состоянию h_{t-1}.
- $b_r \in R^{n_h}$: вектор смещения ворот сброса.

- **Новое потенциальное состояние**

После того как ворота сброса решают, сколько информации нужно забыть, вычисляется новое потенциальное состояние $\tilde{h}_t$, которое основано на текущем входе x_t и скорректированном предыдущем скрытом состоянии h_{t-1}:

$$\tilde{h}_t = tanh(W_h \times x_t + U_h \times (r_t \odot h_{t-1}) + b_h),$$

где:

- $\tilde{h}_t \in R^{n_h}$: новое потенциальное скрытое состояние.
- $W_h \in R^{n_h \times n_x}$: матрица весов, применяемая к входу x_t для вычисления нового состояния.
- $U_h \in R^{n_h \times n_h}$: матрица весов, применяемая к предыдущему скрытому состоянию h_{t-1}, модифицированному воротами сброса r_t.
- $b_h \in R^{n_h}$: вектор смещения для нового состояния.

- **Обновление скрытого состояния**

Новое скрытое состояние h_t в момент времени t вычисляется путем комбинации нового потенциального состояния $\tilde{h}_t$ и предыдущего скрытого состояния h_{t-1}, под управлением ворот обновления z_t:

$$h_t = z_t \odot h_{t-1} + (1 - z_t) \odot \tilde{h}_t,$$

где:

- $z_t \odot h_{t-1}$: часть информации, которая сохраняется из предыдущего состояния.

- $(1 - z_t) \odot \tilde{h}_t$: часть новой информации, добавляемая из потенциального состояния $\bar{h}_t$.

2.3.2 ПРОЕКТИРОВАНИЕ МОДЕЛИ НЕЙРОННОЙ СЕТИ

Рассмотрим топологию ПКС, показанную на рисунке 2.12. Задача состоит в том, чтобы найти четыре кратчайших пути передачи данных между серверами H1 и H2.

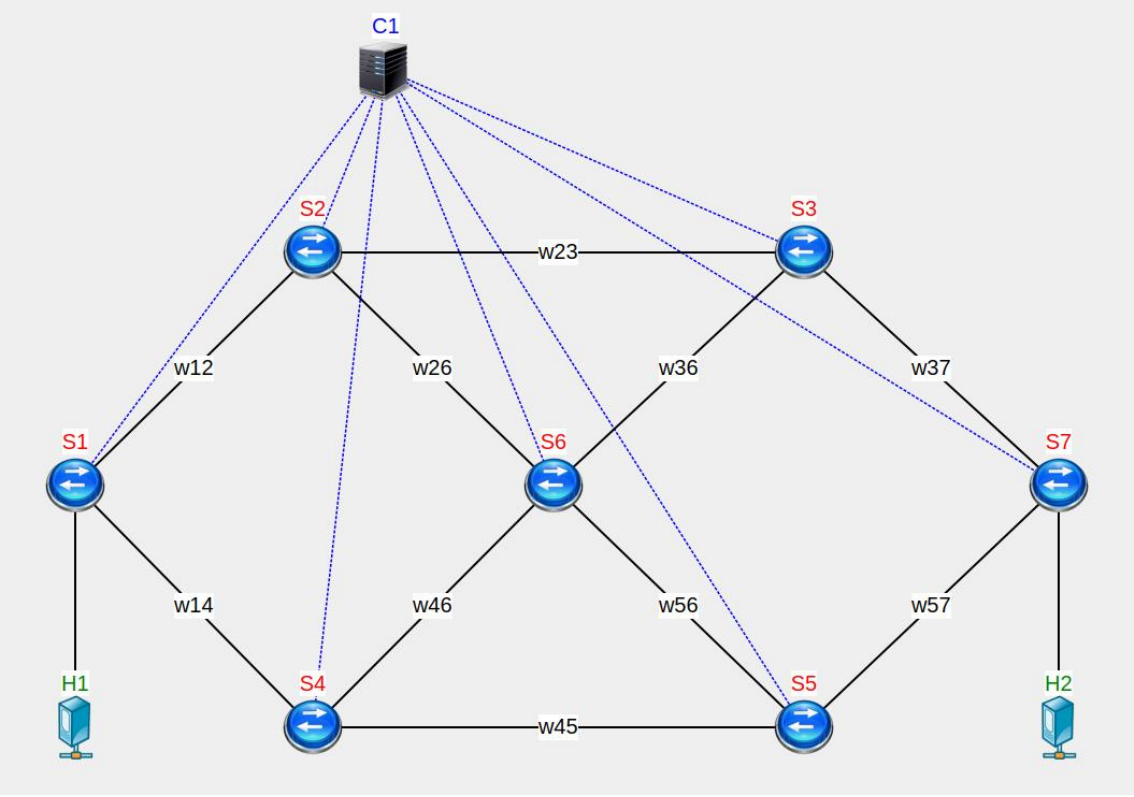


Рисунок 2.12 – Топология ПКС

Топологию ПКС можно описать как матрицу размера $n \times n$, где n – количество вершин, а каждое значение матрицы – стоимость каждого канала связи:

$$W = \begin{bmatrix} w_{11} & \cdots & w_{1n} \\ \vdots & \ddots & \vdots \\ w_{n1} & \cdots & w_{nn} \end{bmatrix},$$

где w_{ij} – вес ребра (i, j). Несуществующие каналы будут представлены как символом «∞».

Для решения этой проблемы используется рекуррентная искусственная нейронная модель, как показано на рисунках 2.13 и 2.14.

- Входные и выходные слои: На каждом шаге в качестве входных данных используются весы всех каналов ($Input$), тогда как выходные данные представляют собой кратчайшие пути, определенные с применением метода бинарного кодирования ($Output$). Это подразумевает, что если значение нейрона составляет 1, то соответствующий канал связи участвует в формировании пути; в противном случае, если значение равно 0, данный канал в маршруте не задействован.

$$Input = \left[w_{ij} | w_{ij} \in W, w_{ij} \neq \infty\right],$$

$$Output = \left[y_{ij} | y_{ij} \in \{0, 1\}\right].$$

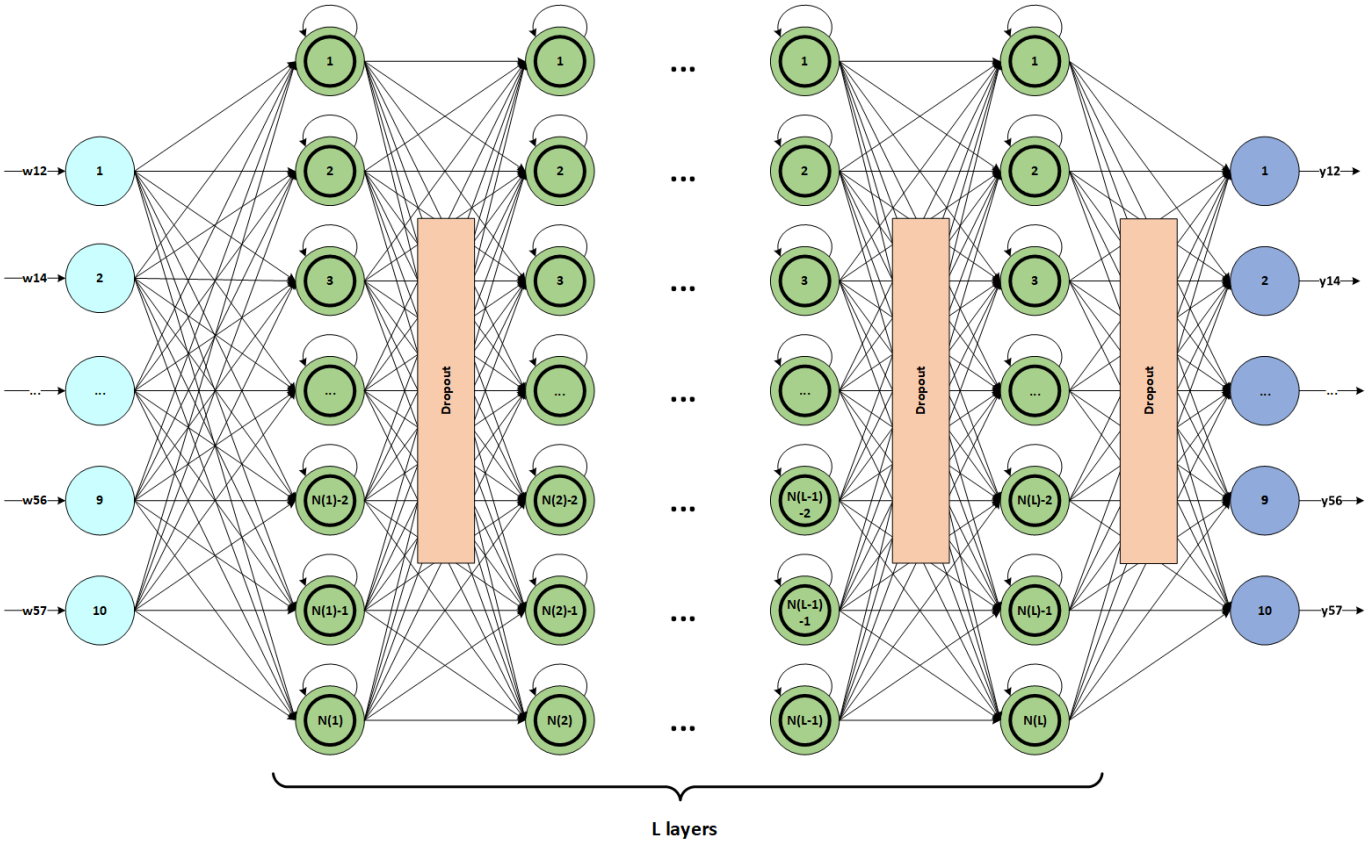


Рисунок 2.13 – Полносвязное представление нейронной сети

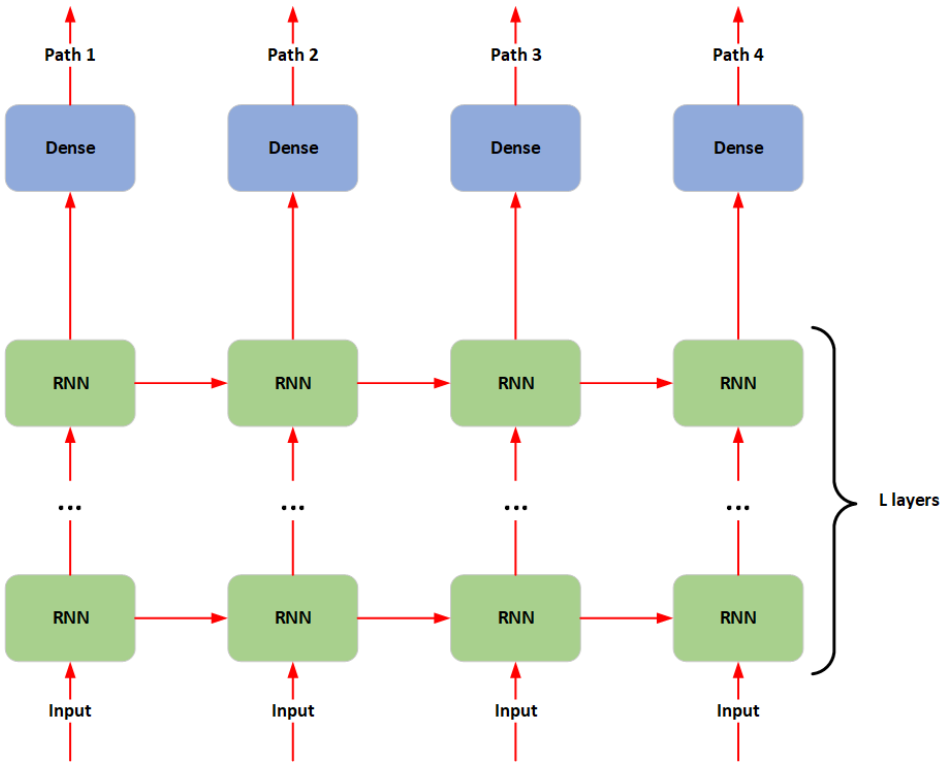


Рисунок 2.14 – Свернутое представление нейронной сети

- Скрытые слои: Эти слои функционируют на основе рекуррентной нейронной сети RNN. Количество слоев и количество нейронов на каждом слое считается гиперпараметром, который необходимо оптимизировать.
- Функция потерь: Для данной модели используется функция потерь BinaryCrossentropy.

$$Loss(\hat{y}, y) = -\sum_{j=1}^{C}\left[y_j \times \log\left(\hat{y}_j\right) + (1 - y_j) \times log\left(1 - \hat{y}_j\right)\right],$$

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} Loss\left(\hat{y}^{(i)}, y^{(i)}\right),$$

где $C = |Edges|$ – это количество каналов, присутствующих в сети ПКС, m – это количество обрацов и θ – весы нейросети.

- Дополнительные параметры настройки:
  - После RNN-слоев включается полносвязный слой с активационной функцией сигмоида.
  - Для предотвращения переобучения после каждого RNN-слоя применяются слои Dropout.

- Входные данные нормализуются с использованием метода StandardScaler.

2.3.3 ОПТИМИЗАЦИЯ ГИПЕРПАРАМЕТРОВ МОДЕЛИ НЕЙРОННОЙ СЕТИ С ПОМОЩЬЮ РОЕВОГО ИНТЕЛЛЕКТА

А. Постановка задачи оптимизации гиперпараметров нейронной сети

В области глубокого обучения модели часто включают множество параметров и гиперпараметров, которые необходимо точно настроить для достижения наилучшей производительности. Гиперпараметры – это параметры, которые не обучаются напрямую на данных, а задаются до начала процесса обучения. К ним относятся, например, скорость обучения, размер батча, количество скрытых слоев, число нейронов в каждом слое и т. д. Выбор оптимальных значений этих гиперпараметров может существенно повлиять на эффективность модели.

Однако подобрать подходящие значения гиперпараметров зачастую бывает непросто, особенно когда их количество увеличивается, и пространство поиска становится очень большим. Это приводит к сложному процессу, требующему значительных временных и вычислительных ресурсов. Задача оптимизации гиперпараметров заключается в поиске наилучших значений, чтобы обеспечить максимальную производительность модели глубокого обучения на заданном наборе данных.

Предположим, что M – это модель искусственной нейронной сети, для которой перед обучением необходимо выполнить оптимизацию гиперпараметров. Множество возможных комбинаций гиперпараметров, называемое пространством поиска, обозначим как $H = \{h_1, h_2, \dots, h_k\}$. Целью задачи является найти такой набор гиперпараметров h^*, при котором модель M покажет наилучшую производительность на тестовом наборе данных D_{test}. Производительность модели обычно оценивается целевой функцией $L(M(h), D_{test})$, например, функцией точности или функцией

потерь на тестовом наборе данных. Таким образом, задачу оптимизации гиперпараметров можно сформулировать следующим образом:

$$h^* = \underset{h \in H}{argmax}\, L(M(h), D_{test}),$$

где H – пространство возможных значений гиперпараметров и $L(M(h), D_{test})$ – значение функции точности или функции потерь модели на тестовом наборе данных D_{test}.

Для решения этой проблемы, в таблице 2.3 приведены некоторые из наиболее распространенных методов оптимизации гиперпараметров. Каждый метод имеет свои преимущества и недостатки и применяется в определенных ситуациях.

Таблица 2.3 – Методы оптимизации гиперпараметров

Метод	**Описание**	**Преимущества**	**Недостатки**
Поиск по сетке (Grid Search, GS)	Проверяются все возможные комбинации гиперпараметров	Легко реализовать и охватывает все значения в сетке	Высокая вычислительная сложность и значительные временные затраты
Случайный поиск (Random Search, RS)	Случайным образом выбирается комбинация гиперпараметров для тестирования	Эффективен при большом числе гиперпараметров	Нестабильность и отсутствие гарантии наилучших результатов
Байесовская оптимизация (Bayesian Optimization, BO)	Использует вероятностную модель (обычно гауссовский процесс) для оценки целевой функции	Высокая эффективность и хорошие результаты	Сложная реализация и требует глубоких знаний теории вероятности и статистики

B. Метод представления решения для гиперпараметров нейронной сети

В данной работе для модели рекуррентной нейронной сети, предназначенной для поиска кратчайших путей, пространство поиска гиперпараметров определяется как в таблице 2.4.

Таблица 2.4 – Пространство поиска гиперпараметров

Гиперпараметры	Описание	Множество значений
Количество скрытых слоев	Число слоев между входным и выходным. Больше слоев улучшает обучение, но увеличивает риск переобучения.	$[1,\ 2,\ 3, 4, 5]$
Количество нейронов в скрытом слое	Число нейронов на каждом слое. Больше нейронов позволяет лучше улавливать сложности данных.	$[10,\ 20,\ 30, \ldots,$ $180,\ 190,\ 200]$
Dropout	Процент исключения нейронов при обучении для предотвращения переобучения.	$[0.0, 0.1, 0.2, 0.3, 0.4, 0.5]$
Тип ячейки	Для RNN важен тип ячейки (например, SimpleRNN, LSTM, GRU), влияет на обработку последовательностей.	$[\mathrm{SimpleRNN}, \mathrm{GRU}, \mathrm{LSTM}]$
Алгоритм оптимизации	Метод обновления весов в процессе обучения.	$[\mathrm{SGD}, \mathrm{RMSprop}, \mathrm{Adam}]$
Скорость обучения (learning_rate)	Величина шага для коррекции весов, влияет на скорость и качество обучения.	[0.0001,0.001,0.01,0.1]
Размер батча (batch size)	Количество примеров за один проход, влияет на скорость обучения и потребление памяти.	$[16, 32, 64, 128,$ $256, 512, 1024]$

При большом размере пространства поиска одним из эффективных методов оптимизации гиперпараметров является использование алгоритмов роевого интеллекта. Члены роя взаимодействуют и учатся друг у друга, что позволяет находить глобально оптимальные решения, избегать локальных экстремумов и, таким образом, определять наилучший набор гиперпараметров для модели.

Количество скрытых слоев является неизвестным гиперпараметром, что приводит к тому, что решения, соответствующие наборам гиперпараметров, имеют различную длину. Однако алгоритмы, основанные на принципах роевого интеллекта, обычно применяются к задачам с решениями фиксированной длины. В данной работе представлен специальный метод представления решений, иллюстрированный в таблице 2.5, который обеспечивает удобное использование алгоритмов роевого интеллекта.

Таблица 2.5 – Метод представления решения

Слой 1	Dropout 1	…	Слой 5	Dropout 5	Тип ячейки	Алгоритм оптимизации	Скорость обучения	Размер батча
N_1	d_1	…	N_5	d_5	T	A	lr	b

В этом методе представления решений, значения представляют собой позиции значений гиперпараметров в наборе значений таблицы 2.4. Например, $b = 2$ означает, что размер батча равен 64, а аналогичные принципы применяются и к другим гиперпараметрам. При выполнении алгоритмов роевого интеллекта для оптимизации гиперпараметров результаты вычислений округляются до ближайшего целого числа.

Начальная популяция представляет собой набор решений, соответствующих случайно инициализированным наборам гиперпараметров. Качество решения оценивается с помощью функции приспособленности, которая измеряет точность модели нейронной сети с заданным набором гиперпараметров на тестовом наборе данных.

Процесс оптимизации гиперпараметров для нейронных сетей может быть долгим и требовательным к ресурсам. Для сокращения времени и затрат применяются следующие методы:

- Используется подмножество данных для оптимизации вместо всего набора, что позволяет сосредоточиться на наиболее информативных данных.

- Если точности на проверочном наборе не улучшаются в течение 10 последовательных эпох, обучение останавливается, чтобы избежать ненужных вычислений при отсутствии прогресса.
- Результаты обучения сохраняются во время оптимизации, что исключает необходимость повторного запуска с теми же гиперпараметрами.
- Чтобы избежать использования слишком больших моделей и минимизировать затраты ресурсов, используется механизм сравнения двух моделей. Если их точность на тестовом наборе одинакова, то предпочтение отдается модели с меньшим количеством параметров.

C. Процесс оптимизации гиперпараметров нейронной сети с помощью роевого интеллекта

Для решения поставленной задачи оптимизации гиперпараметров, алгоритмы роевого интеллекта реализуются следующим образом.

- **Инициализация популяции**: В контексте оптимизации гиперпараметров нейронной сети, каждый агент в популяции представляет собой отдельный набор гиперпараметров модели. Первоначальные решения генерируются случайным образом в пределах предопределенных границ для каждого параметра, как указано в таблице 2.4. Это можно представить следующим образом:

$$x_i = [x_{i1}, x_{i2}, \dots, x_{iD}],$$

где $i \in (1, 2, \dots, N)$, а x_{ik} случайно выбирается в пределах допустимого диапазона для k-го гиперпараметра, N – число решений (размер популяции) и D – размерность задачи (число гиперпараметров). Верхний и нижний пределы для x_{ik} устанавливаются как 0 и $(len_k - 1)$, где len_k – длина набора значений гиперпараметра k.

- **Оценка приспособленности**: Эта оценка имеет решающее значение, так как она направляет поиск в сторону более перспективных областей пространства гиперпараметров. Приспособленность каждого решения $f(x_i)$ определяется его

производительностью, измеряемой как точность на тестовом наборе данных. Более высокая точность указывает на лучший набор гиперпараметров.

- **Итеративная оптимизация**: Поиск оптимальных гиперпараметров осуществляется посредством повторяющихся циклов оценки и обновления. Каждая итерация (или поколение) обновляет популяцию на основе оценок приспособленности и конкретных правил алгоритма.
- **Критерии остановки**: Процесс оптимизации останавливается, когда достигается максимальное количество итераций (Max), превышен лимит времени или если улучшение решений падает ниже порога, что указывает на сходимость.
- **Оптимальные гиперпараметры**: Решение с самой высокой точностью на проверочном наборе данных выбирается в качестве оптимального набора гиперпараметров. Затем этот оптимальный набор используется для обучения окончательной модели на полном обучающем наборе, чтобы подготовить ее к реальному тестированию или развертыванию.

а) Генетический алгоритм

Например:

Хромосома C представлена высшим описанным методом:

$$C = [2, 1, 0, 3, 15, 2, 11, 4, 20, 3, 1, 2, 1, 3].$$

Эта хромосома соответствует следующему набору гиперпараметров:

$$[20, 0.2, -1, 0.4, 150, 0.3, 110, 0.4, -1, 0.3, GRU, Adam, 0.001, 128].$$

- Скрещивание
  - Случайным образом выбирают две родительские хромосомы.

$$P_1 = [7, 3, 0, 1, 14, 2, 20, 4, 10, 3, 2, 1, 3, 2],$$

$$P_2 = [20, 1, 10, 0, 11, 2, 11, 2, 0, 3, 1, 3, 2, 3].$$

  - Случайный бинарный вектор инициализируется и имеет ту же длину, что и родительская хромосома.

$$Mask = [1, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 1, 0].$$

- Для формирования первого потомка используется ген от первого родителя, если значение вектора равно 1, и ген от второго родителя, если значение вектора равно 0. И наоборот, такой же процесс применяется для формирования второго потомка.

$$S_1 = [\mathbf{20}, \mathbf{1}, \mathbf{0}, \mathbf{0}, \mathbf{14}, \mathbf{2}, \mathbf{20}, \mathbf{4}, \mathbf{10}, \mathbf{3}, \mathbf{2}, \mathbf{3}, \mathbf{2}, \mathbf{2}],$$

$$S_2 = [\mathbf{7}, \mathbf{3}, 10, \mathbf{1}, 11, \mathbf{2}, 11, 2, 0, \mathbf{3}, 1, \mathbf{1}, \mathbf{3}, 3].$$

- Мутация
  - Случайным образом выбирается исходная хромосома

$$S_1 = [20, 1, 0, 0, 14, 2, 20, 4, 10, 3, 2, 3, 2, 2].$$

  - Генеруется случайный бинарный вектор

$$Mask = [0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0].$$

  - Выполняется оператор мутации в генах, если значение вектора равно 1.

Хромосома после мутации получается следующим образом

$$M_1 = [20, \mathbf{3}, 0, 0, 14, \mathbf{1}, 20, 4, 10, \mathbf{2}, 2, 3, \mathbf{0}, 2].$$

- Селекция

В данной работе используется метод турнирного отбора. Случайным образом выбирают любых двух особей. С помощью метода сравнения моделей, как описано выше, лучшая особь отбирается для следующего поколения.

b) Оптимизация муравьиной колонии

Алгоритм муравьиной системы

- Инициализация феромонов

Начальный уровень феромонов для каждого значения гиперпараметра инициализируется согласно следующей формуле:

$$\tau_0 = \frac{1}{n_{hp}},$$

где n_{hp} – количество гиперпараметров.

- Равная эвристика

Одно эвристическое значение указывается для каждого гиперпараметра, которое представляет равный приоритет для всех начальных значений. Эта эвристика не демонстрирует первоначального смещения к какому-либо конкретному значению в диапазоне.

$$\eta_{i \in E_{pr}} = \frac{1}{len(E_{pr})},$$

где E_{pr} – набор значений гиперпараметра pr.

- Построение решения для каждого муравья

Каждый муравей создает полный набор гиперпараметров, выбирая значение для каждого гиперпараметра. На выбор влияют как уровни феромонов, так и эвристическая функция. Вероятность выбора значения v в наборе значений гиперпараметра pr рассчитывается по следующей формуле:

$$p_v = \frac{(\tau_v)^{\alpha} \times (\eta_v)^{\beta}}{\sum_{i \in E_{pr}} [(\tau_i)^{\alpha} \times (\eta_i)^{\beta}]}. \quad (2.3)$$

где τ_i – концентрация феромона значения i, α – параметр, контролирующий влияние τ_i и β – параметр, контролирующий влияние η_i.

- Обновление концентрации феромона

После того как муравьи построят решение, феромон значения i для гиперпараметра pr обновится по следующей формуле:

$$\tau_i = (1 - \rho) \times \tau_i + \sum_{k=1}^{N} \Delta\tau_i ,$$

$$\Delta\tau_i = \begin{cases} Q \times f_k \text{ , если } k\text{– й муравей выбирает значение } i \\ 0, \text{ в противном случае} \end{cases},$$

где ρ – скорость испарения феромона и f_k – точность модели, соответствующей k-му муравью.

Алгоритм системы муравьиной колонии

Для оптимизации гиперпараметров модели нейронной сети, алгоритм системы муравьиной колонии также отличается от алгоритма муравьиной системы тремя аспектами.

- Правило перехода состояния

- Инициализируется случайное значение q.

- Если $q \leq q_0$, то для гиперпараметра pr муравей выбирает значение e, который рассчитывается согласно следующей формуле:

$$e = argmax_{i \in E_{pr}}[(\tau_i)^{\alpha} \times (\eta_i)^{\beta}],$$

где q_0 – параметр алгоритма $(0 \leq q_0 \leq 1)$.

- Если $q > q_0$, то муравей выбирает значение гиперпараметра с вероятностью, как в формуле (2.3).

- Локальное обновление концентрации феромона

В процессе построения решения каждого муравья, когда муравей выбирает значение i, концентрация феромона, соответствующее этому значению, обновляется в соответствии со следующей формулой:

$$\tau_i = (1 - \rho) \times \tau_i + \rho \times \Delta\tau_i \ ,$$

где $\Delta\tau_i$ – параметр, определяемый экспериментально. В данной работе выбрано $\Delta\tau_i = \tau_0$.

- Глобальное обновление концентрации феромона

После того, как все муравьи завершат свои решения, только значения лучшего набора гиперпараметров будут обновлены с учетом новой концентрации феромона:

$$\tau_{best} = (1 - \rho) \times \tau_{best} + \rho \times \Delta\tau_{best} \ ,$$

$$\Delta\tau_{best} = Q \times f_{best} \ ,$$

где f_{best} – наивысшая точность модели, обнаруженная колонией муравьев в текущей итерации.

c) Алгоритм стаи птиц

- Инициализация позиции и скорости

Каждая птица i инициализируется случайной позицией x_i в пространстве поиска и случайной скоростью v_i, которая определяет, насколько сильно позиция изменяется на каждой итерации.

- Инициализация личной и глобальной лучшей позиции

Каждая птица сохраняет информацию о своей лучшей достигнутой позиции p_i^{local}, представляющей наилучший найденный набор гиперпараметров. Птица с наилучшим значением p_i^{local} среди всей популяции определяется как обладатель глобальной оптимальной позиции p_{global}.

- Движение птиц

Скорость птицы обновляется согласно формуле, интегрирующей ее текущую скорость, индивидуальную и глобальную оптимальные позиции, а также включающей случайные компоненты. Позиция птицы корректируется в соответствии с новым вектором скорости.

$$v_i^{t+1} = w \times v_i^t + c_1 \times r_1 \times \left(p_i^{local} - x_i^t\right) + c_2 \times r_2 \times \left(p_{global} - x_i^t\right),$$

$$x_i^{t+1} = round(x_i^t + v_i^{t+1}),$$

где x_i^t и v_i^t – позиция и скорость i-ой птицы в итерации t, w – инерционный вес, c_1 и c_2 – коэффициенты ускорений (когнитивный и социальный компоненты), r_1 и r_2 – случайные числа в интервале $(0, 1)$.

Верхний и нижний пределы для скорости устанавливаются как $(-\sqrt{len_k})$ и $\sqrt{len_k}$, соответствующие гиперпараметру k.

d) Алгоритм искусственной пчелиной колонии

Популяция делится на рабочих пчел, наблюдателей и разведчиков, каждый из которых играет свою роль в поиске. Рабочие пчелы эксплуатируют текущие источники пищи (решения), наблюдатели наблюдают и выбирают среди них на основе их качества, а разведчики исследуют новые источники пищи случайным образом.

- Фаза рабочих пчел

На этом этапе каждая рабочая пчела, связанная с определенным решением (набором гиперпараметров), будет пытаться найти новое, потенциально лучшее решение, локально модифицируя текущее. Каждая рабочая пчела изменяет свое текущее решение x_i, исследуя соседние решения:

$$x_{ik}^{new} = round\left[x_{ik} + \Phi_{ik} \times \left(x_{ik} - x_{jk}\right)\right],$$

где $j \neq i$ – случайно выбранный индекс решения, k – случайно выбранный гиперпараметр, x_{ik} – значение i-го решения, соответствующее гиперпараметру k и Φ_{ik} – случайное число в отрезке $[-1.0, 1.0]$. Если x_i^{new} лучше x_i, то x_i заменяется на x_i^{new}.

- Фаза пчел-наблюдателей

Каждая пчела-наблюдатель выбирает источник пищи на основе вероятности, связанной с приспособленностью источников пищи, предоставленных рабочими пчелами. Вероятность выбора i-го решения определяется как:

$$p_i = \frac{f(x_i)}{\sum_{j=1}^{N} f(x_j)},$$

где $f(x_i)$ – приспособленность i-го решения.

После того, как источник пищи выбран, пчела-наблюдатель выполняет поиск, аналогичный поиску рабочих пчел, используя ту же формулу, чтобы найти новое решение.

- Фаза пчел-разведчиков

Если после определенного количества попыток дальнейшее улучшение конкретного решения становится невозможным, соответствующая рабочая пчела превращается в пчелу-разведчика и отказывается от своего текущего источника пищи. Пчела-разведчик случайным образом генерирует новое решение x_i в пределах заранее установленных границ параметров, фактически начиная процесс поиска заново.

е) Алгоритм светлячков

- Определение яркости светлячков

Яркость каждого светлячка напрямую зависит от целевой функции задачи оптимизации. В случае оптимизации гиперпараметров яркость обычно соответствует точности тестирования модели, зависящей от гиперпараметров, представленных светлячком.

- Коэффициент случайности α_0 уменьшается после каждой итерации по формуле:

$$\alpha = \alpha_0 \times \theta^t,$$

где $\theta \in [0.9, 0.99]$ – скорость убывания α и t – индекс текущей итерации.

- Перемещение светлячков

  - Предположим, что светлячок j ярче, чем светлячок i. В этом случае светлячок i стремится приблизиться к светлячку j. Для этого необходимо оценить привлекательность между светлячками i и j.

$$r_{ij} = \sqrt{\frac{\sum_{k=1}^{D}\left(\frac{x_{ik}-x_{jk}}{len_k}\right)^2}{D}},$$

$$B_{ij} = \beta_0 \times e^{-\gamma \cdot {r_{ij}}^2},$$

где r_{ij} – расстояния между светлячками i и j, β_0 – фактор яркости светлячка и γ – коэффициент поглощения.

  - Далее вычисляется размер случайного шага.

$$A_{ij} = \left[A_{ij}^1, A_{ij}^2, \dots, A_{ij}^D\right],$$

$$A_{ij}^k = len_k \times \alpha \times (\epsilon - \frac{1}{2}),$$

где $k \in (1, 2, \dots, D)$ и ϵ – случайное число в интервале $(0, 1)$.

  - После этого обновляется позиция каждого светлячка в направлении более привлекательных светлячков. Новая позиция представляет собой новый набор гиперпараметров.

$$x_i = x_i + B_{ij} \times \left(x_j - x_i\right) + A_{ij}.$$

ОСНОВНЫЕ РЕЗУЛЬТАТЫ И ВЫВОДЫ

1. Предложена математическая модель программно-конфигурируемых сетей. Вес каждого ребра рассчитывается на основе коэффициента загруженности канала,

что способствует выбору наименее загруженных путей для маршрутизации трафика.

2. Введен и проанализирован концепт роевого интеллекта, который используется для создания интеллектуальных мультиагентных систем, имитирующих поведение социальных насекомых и животных, таких как муравьи, пчелы и птицы. Эти системы используют децентрализованное управление и самоорганизацию для эффективного решения сложных задач оптимизации.

3. Рассмотрены различные методы кодирования пути для алгоритмов роевого интеллекта при маршрутизации, включая генетический алгоритм с фиксированной и переменной длиной хромосом, а также приоритетное кодирование, которое позволяет преобразовывать непрерывные результаты алгоритмов роевого интеллекта в дискретные маршруты.

4. Предложена адаптация и модификация алгоритмов роевого интеллекта для решения задачи многопутевой маршрутизации в ПКС:

- Генетический алгоритм: используется для оптимизации маршрутов путем выполнения операций мутации, скрещивания и отбора. Генетический алгоритм строит начальную популяцию маршрутов и применяет эволюционные методы для поиска оптимальных путей.
- Оптимизация муравьиной колонии: имитирует поведение муравьев в природе для поиска кратчайших путей. Муравьи оставляют феромонные следы, которые помогают другим муравьям выбирать оптимальные маршруты на основе вероятности и эвристических данных.
- Алгоритм стаи птиц: оптимизирует маршруты на основе социального поведения птиц. Каждое решение представляет собой позицию птицы, которая обновляется с учетом как индивидуального, так и коллективного опыта стаи.
- Алгоритм искусственной пчелиной колонии: основан на поведении пчел при поиске пищи. Пчелы-наемники и пчелы-наблюдатели совместно оптимизируют

маршруты путем обмена информацией и выбора лучших источников пищи, что соответствует наилучшим маршрутам.

- Алгоритм светлячков: вдохновлен мигающим поведением светлячков, где каждый светлячок привлекается более ярким светлячком. Яркость светлячка соответствует качеству маршрута, и светлячки перемещаются к более привлекательным маршрутам.

5. Предложена нейросетевая модель многопутевой маршрутизации в ПКС на основе рекуррентной нейронной сети, позволяющая принимать решения о маршрутизации в режиме реального времени.

6. Предложены математическая модель и метод оптимизации гиперпараметров нейросетевой модели многопутевой маршрутизации в ПКС на основе алгоритмов роевого интеллекта, обеспечивающие высокую точность прогнозирования маршрутов и снижение вычислительных затрат.

ГЛАВА 3 МАТЕМАТИЧЕСКОЕ ОБЕСПЕЧЕНИЕ БАЛАНСИРОВКИ ПОТОКОВ ДАННЫХ В ПКС

3.1 ПОСТАНОВКА ЗАДАЧИ БАЛАНСИРОВКИ ПОТОКОВ ДАННЫХ В ПКС

В программно-конфигурируемых сетях состояние сети может постоянно изменяться, поэтому для обеспечения эффективного использования сетевых ресурсов и предотвращения перегрузок необходима динамическая маршрутизация и распределение потоков данных. Это позволяет поддерживать стабильную и высокопроизводительную работу сети.

Поставленная задача включает в себя:

- Определение оптимальных маршрутов между коммутаторами, связанными с источником и получателем каждого потока.
- Распределение трафика между найденными маршрутами.
- Адаптацию к изменениям в сети, таким как перегрузка каналов, за счет обновления маршрутов и перераспределения трафика в реальном времени.

Топологию ПКС можно представить в виде графа следующим образом:

$$G = (V, E),$$

где:

- $V = \{v_1, v_2, \dots, v_n\}$ – множество узлов, где каждый узел v_i представляет собой коммутатор.
- $E = \{e_{ij} | (v_i, v_j) \in V \times V\}$ – множество ребер, где каждое ребро e_{ij} представляет собой канал связи между двумя коммутаторами v_i и v_j.

Для эффективного распределения трафика каждому каналу e_{ij} присваивается вес w_{ij}, который определяется следующим образом:

$$w_{ij} = \frac{1}{1-u_{ij}},$$

где u_{ij} – текущий коэффициент загруженности канала. Чем менее загружен канал, тем

меньше его вес, что способствует маршрутизации по менее загруженным путям.

Пусть в сети имеется L потоков, которые необходимо маршрутизировать.

$$Flows = \{flow_1, flow_2, \dots, flow_L\}.$$

Каждый поток $flow_i$ описывается следующим образом:

$$flow_i = \{v_{src}(flow_i), v_{dst}(flow_i), \lambda(flow_i)\},$$

где:

- $i \in (1,2, \dots, L)$.
- $v_{src}(flow_i)$ – исходный коммутатор.
- $v_{dst}(flow_i)$ – конечный коммутатор.
- $\lambda(flow_i)$ – требуемая пропускная способность потока $flow_i$.

Для каждого потока $flow_i$ общий трафик $\lambda(flow_i)$ распределен по набору K путей $P_1, P_2, \dots, P_K$ от $v_{src}(flow_i)$ до $v_{dst}(flow_i)$:

$$\sum_{k=1}^{K} \lambda_k(flow_i) = \lambda(flow_i),$$

где $\lambda_k(flow_i)$ – это часть трафика, распределенная по пути P_k для потока $flow_i$.

Все потоки данных должны быть распределены по путям таким образом, чтобы общий трафик по каждому каналу e_{ij} на пути не превышал его общую пропускную способность C_{ij}:

$$\sum_{\{flow_i | e_{ij} \in P_k(flow_i)\}} \lambda_k(flow_i) \leq C_{ij} \; \forall \; e_{ij} \in E.$$

Стоимость пути P_k рассчитывается как сумма стоимостей w_{ij} каналов, принадлежащих этому пути:

$$w_{P_k} = \sum_{e_{ij} \in P_k} w_{ij}.$$

Задача балансировки потоков данных состоит в том, чтобы найти пути $P_1, P_2, \dots, P_K$ и распределить трафик $\lambda(flow_i)$ по этим путям так, чтобы минимизировать общую стоимость сети и избежать перегрузок:

$$min \sum_{flow_i \in Flows} \sum_{k=1}^{K} w_{P_k} \times \lambda_k(flow_i).$$

Для оценки эффективности распределения трафика между каналами может использоваться индекс Джейна (Jain’s Fairness Index или Load Balancing Index, LBI)

[107-109], который измеряет, насколько равномерно распределена загрузка между всеми каналами:

$$LBI = \frac{(\sum_{e_{ij} \in E} u_{ij})^2}{|E| \times \sum_{e_{ij} \in E} u_{ij}^2},$$

где $|E|$ – количество каналов в сети. Индекс принимает значение от 0 до 1:

- 1 означает полную равномерность в распределении трафика между каналами, то есть каждый канал использован одинаково.
- 0 указывает на крайнее неравенство, когда один или несколько каналов сильно перегружены, а другие практически не используются.

3.2 МОДЕЛЬ И АЛГОРИТМ ДИНАМИЧЕСКОЙ АДАПТИВНОЙ МНОГОПУТЕВОЙ БАЛАНСИРОВКИ ПОТОКОВ ДАННЫХ В ПКС

Метод динамической адаптивной многопутевой балансировки потоков данных в ПКС (Dynamic Adaptive Multipath Load Balancing, ***DAMLB***) построен на тесной координации четырех основных компонентов, что обеспечивает оптимизацию производительности сети. Компонент мониторинга топологии (***Topology Monitor***) постоянно обновляет информацию о состоянии сети, предоставляя точные данные о существующих устройствах и соединениях. На основе этой топологии компонент мониторинга портов (***Port Monitor***) отслеживает трафик на портах и периодически рассчитывает пропускную способность соединений, определяя стоимость канала на основе коэффициента загруженности канала. Компонент мониторинга потоков (***Flow Monitor***) непрерывно обновляет информацию о потоках, проходящих через каждый коммутатор. Информация о стоимости и пропускной способности передается в компонент многопутевой маршрутизации и балансировки потоков данных (***Multipath Load Balancing***), что позволяет системе гибко направлять потоки трафика по оптимальным маршрутам. При изменениях в топологии сети или увеличении трафика

система автоматически адаптируется и перенаправляет потоки, чтобы поддерживать эффективность балансировки нагрузки и гарантировать качество обслуживания.

A. Компонент мониторинга топологии

a) Представление сетевой топологии

На этапе инициализации, этот компонент начинает с пустого графа. По мере появления новых устройств и соединений в сети они последовательно обнаруживаются и добавляются в граф. Этот процесс продолжается до тех пор, пока все устройства и соединения в сети не будут выявлены, что позволяет построить полный граф $G = (V, E)$, точно отражающий текущую топологию сети.

- $V = \{v_1, v_2, \dots, v_n\}$ – множество коммутаторов.
- $E = \{e_{ij} | (v_i, v_j) \in V \times V\}$ – множество каналов связи.

b) Представление канала связи

Каждый канал $e_{ij} \in E$ имеет следующие атрибуты:

$$e_{ij} = \{port_{v_i}, port_{v_j}, w_{ij}\},$$

где:

- $port_{v_i}$ – исходный порт на коммутаторе v_i.
- $port_{v_j}$ – конечный порт на коммутаторе v_j.
- w_{ij} – вес канала, первоначально инициализируется как $w_{ij} = 1$.

c) Обновление графа сети

Когда в сетевой топологии происходят изменения, такие как добавление или удаление коммутатора или изменение состояния канала связи, граф $G = (V, E)$ обновляется.

- Если добавляется новый коммутатор v_k:

$$V = \mathrm{V} \cup \{v_k\}.$$

- Если коммутатор v_i удаляется:

$$V = \mathrm{V} \setminus \{v_k\},$$

$$E' = \{e_{ij} | v_i \notin \{v_i, v_j\}\}.$$

- Если добавляется новый канал связи между v_i и v_j:

$$E' = \mathrm{E} \cup \{e_{ij}\}.$$

- Если канал связи между v_i и v_j удаляется:

$$E' = \mathrm{E} \setminus \{e_{ij}\}.$$

B. Компонент мониторинга портов

a) Начало цикла мониторинга

Компонент работает циклически, каждый цикл начинается в момент времени t_0 и повторяется с фиксированным интервалом T_{port_period}. Компонент осуществляет сбор данных после проверки условий готовности, которые включают:

- Коммутаторы были обнаружены и зарегистрированы.
- Топология ПКС инициализирована компонентом мониторинга топологии.

b) Сбор статистики портов

Когда условия готовности выполнены, компонент отправляет запросы на сбор статистики портов всех коммутаторов в сети. Основные запросы, отправляемые на коммутаторы, включают:

- *OFPPortDescStatsRequest*: Запрос на сбор информации об ограниченной пропускной способности порта p, называемой bw_p, измеряемой в Мбит/с.
- *OFPPortStatsRequest*: Запрос на сбор статистических данных о состоянии порта p, включая: переданные байты (tx_bytes), принятые байты (rx_bytes), момент времени измерения ($duration\_sec$) и ($duration\_nsec$).

В каждый момент времени t_n, статистика для порта p на коммутаторе v_i хранится в виде:

$$Stat_p(t_n) = \{tx\_bytes_n, rx\_bytes_n, duration\_sec_n, duration\_nsec_n\}.$$

После получения ответа на запрос, эта статистика сохраняется в структуру данных:

$$port\_stats(v_i, p) = \{Stat_p(t_{n-limit}), \dots, Stat_p(t_{n-1}), Stat_p(t_n)\}.$$

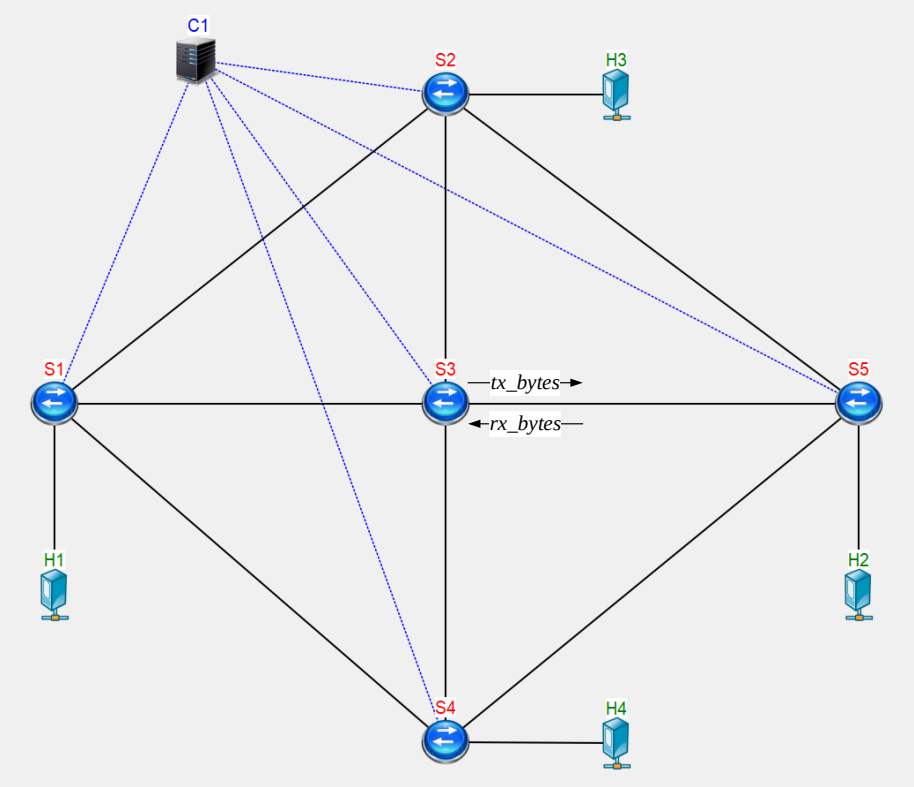


Рисунок 3.1 – Мониторинг портов

c) Расчет пропускной способности

На основе собранной статистики пропускная способность канала между двумя коммутаторами v_i и v_j рассчитывается на основе изменения количества переданных и принятых байтов на порту между двумя измерениями, деленного на время между этими измерениями. Формула для расчета пропускной способности bw_{ij}^{used} (Мбит/с):

$$bw_{ij}^{used} = \frac{8\times(tx\_bytes_n + rx\_bytes_n - tx\_bytes_{n-1} - rx\_bytes_{n-1})}{\Delta t\times 10^6},$$

$$\Delta t = duration\_sec_n + \frac{duration\_nsec_n}{10^9} - duration\_sec_{n-1} - \frac{duration\_nsec_{n-1}}{10^9},$$

где Δt – это разница времени между двумя последовательными измерениями.

d) Расчет коэффициента загруженности канала

Коэффициент загруженности канала u_{ij} рассчитывается как отношение расчетной текущей пропускной способности к максимальной пропускной способности канала:

$$u_{ij} = \frac{bw_{ij}^{used}}{bw_{ij}},$$

где $bw_{ij} = bw_p$.

Коэффициент загруженности канала u_{ij} лежит в диапазоне от 0 до 1, где 0 означает, что канал не используется, а 1 означает, что канал работает на полной мощности.

d) Расчет индекса балансировки нагрузки

Индекс балансировки нагрузки вычисляется по следующей формуле:

$$LBI = \frac{(\sum_{e_{ij} \in E} u_{ij})^2}{|E| \times \sum_{e_{ij} \in E} u_{ij}^2}.$$

e) Расчет стоимости канала

Стоимость канала w_{ij} рассчитывается на основе коэффициента загруженности канала u_{ij}. Чем выше коэффициент загруженности канала, тем выше стоимость канала. Формула для расчета стоимости:

$$w_{ij} = \begin{cases} 100, \text{если } u_{ij} = 1 \\ \frac{1}{1-u_{ij}}, \text{если } 0 \leq u_{ij} < 1 \end{cases}.$$

Если канал работает на полную мощность или перегружен, стоимости канала будет присвоено очень большое значение ($w_{ij} = 100$).

f) Завершение цикла и продолжение мониторинга

После завершения вычисления и обновления метрик каналов, компонент ожидает следующего цикла t_{n+1} для продолжения мониторинга. Этот цикл мониторинга повторяется непрерывно, что обеспечивает своевременное и точное обновление состояния сетевых связей.

C. Компонент мониторинга потоков

a) Начало цикла мониторинга

Компонент мониторинга потоков таже работает циклически, начиная с момента времени t_0 и повторяется через фиксированный интервал T_{flow_period}. Компонент

осуществляет сбор данных о потоках после выполнения условий готовности, как и при мониторинге портов.

b) Сбор статистики потоков

Когда условия готовности выполнены, компонент отправляет запросы для сбора статистики потоков на коммутаторах сети:

- *OFPFlowStatsRequest*: Запрос на сбор информации о текущем состоянии потоков данных, таких как: количество переданных пакетов ($packet\_count$), количество переданных байтов ($byte\_count$), момент времени измерения ($duration\_sec$) и ($duration\_nsec$).

Статистика для потока f_i на коммутаторе v_i в момент времени t_n сохраняется в виде:

$$Stat_{f_i}(t_n) = \{packet\_count_n, byte\_count_n, duration\_sec_n, duration\_nsec_n\}.$$

Эти данные сохраняются в структуру данных:

$$flow\_stats(v_i, f_i, t_n) = \{Stat_{f_i}(t_{n-limit}), \dots, Stat_{f_i}(t_{n-1}), Stat_{f_i}(t_n)\}.$$

c) Расчет пропускной способности потока

На основе собранной статистики компонент вычисляет пропускную способность потока между двумя коммутаторами v_i и v_j на основе изменения количества переданных байтов, деленного на промежуток времени между двумя измерениями. Формула для расчета пропускной способности потока (Мбит/с):

$$speed_{ij}^{f_i} = \frac{8\times(byte\_count_n - byte\_count_{n-1})}{\Delta t\times 10^6},$$

$$\Delta t = duration\_sec_n + \frac{duration\_nsec_n}{10^9} - duration\_sec_{n-1} - \frac{duration\_nsec_{n-1}}{10^9},$$

где Δt – это разница времени между двумя последовательными измерениями.

d) Мониторинг потоков между коммутаторами

Множество потоков между двумя коммутаторами v_i и v_j, и их пропускная способность, записываются в виде:

$$switch\_to\_switch\_flows\_speed_{ij} = \{speed_{ij}^{f_1}, speed_{ij}^{f_2}, \dots, speed_{ij}^{f_k}\},$$

где $speed_{ij}^{f_k}$ – это пропускная способность потока f_k, проходящего между коммутаторами v_i и v_j.

g) Завершение цикла и продолжение мониторинга

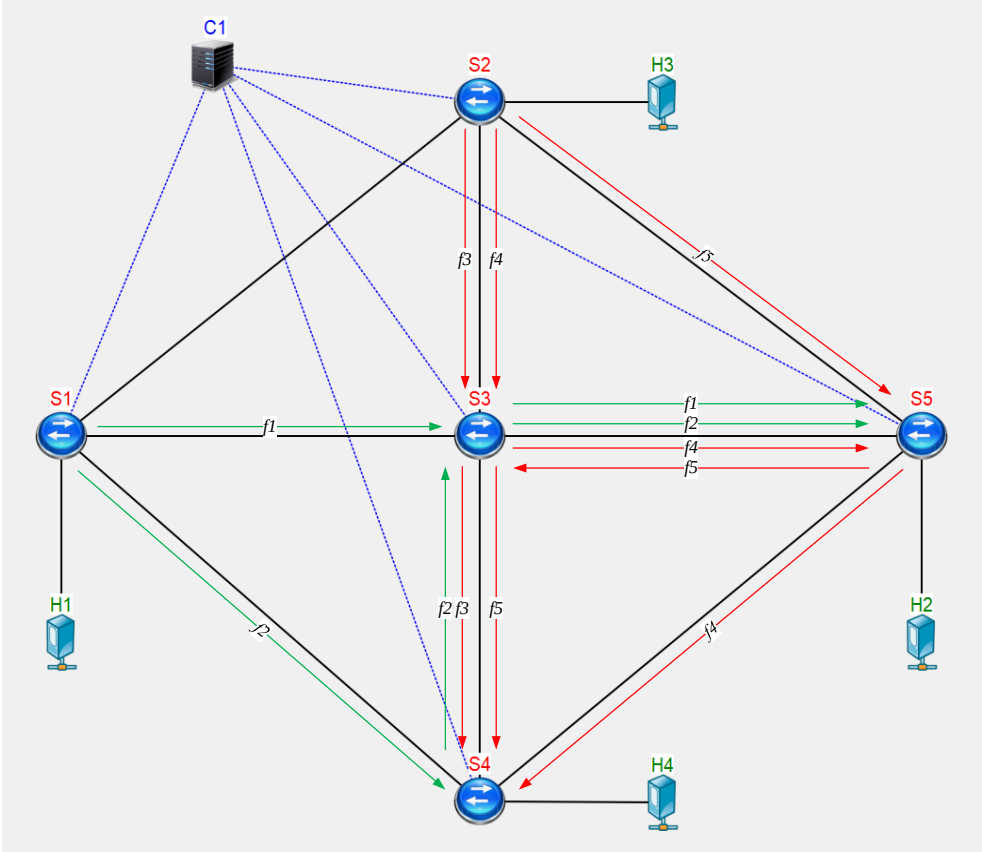


Рисунок 3.2 – Мониторинг потоков

D. Компонент многопутевой маршрутизации и балансировки потоков данных

Этап 1. Динамическая многопутевая маршрутизация

Когда появляется новый поток, не соответствующий ни одной записи в таблице потоков коммутатора, контроллеру передается входящий пакет с информацией, необходимой для установки правила маршрутизации.

Шаг 1. Извлечение информации из заголовков IPv4 и TCP/UDP пакетов

- IP-адреса клиента и сервера:

$$IP_{host\_src} = IPv4.src,$$

$$IP_{host\_dst} = IPv4.dst,$$

где IP_{host_src} и IP_{host_dst} – IP-адреса клиента и сервера, извлекаются из заголовка IPv4.

- TCP/UDP порты клиента и сервера:

$$port_{host\_src} = TCP/UDP.src\_port,$$

$$port_{host\_dst} = TCP/UDP.dst\_port,$$

где $port_{host\_src}$ и $port_{host\_dst}$ – TCP/UDP порты клиента и сервера.

- Коммутаторы, непосредственно подключенные к клиенту и серверу:

$$(v_{src}, port_{v_{src}}) = (dpid_{src}, in\_port),$$

$$(v_{dst}, port_{v_{dst}}) = (dpid_{dst}, in\_port),$$

где v_{src} и v_{dst} – коммутаторы, подключенные к клиенту и серверу, определяется с помощью $dpid$ и порта in_port, через который поступает пакет.

Информации о клиенте и сервере представляются в виде двух массивов:

$$Source = \{IP_{host\_src}, v_{src}, port_{host\_src}, port_{v_{src}}\},$$

$$Destination = \{IP_{host\_dst}, v_{dst}, port_{host\_dst}, port_{v_{dst}}\}.$$

Шаг 2. Расчет оптимальных путей

После извлечения информации о клиенте и сервере компонент рассчитывает оптимальные пути между коммутаторами v_{src} и v_{dst}, основываясь на стоимости каналов между коммутаторами.

Компонент использует алгоритм поиска кратчайших путей для вычисления K маршрутов, с наименьшей стоимостью между v_{src} и v_{dst}:

$$w_{P_k} = \sum_{e_{ij} \in P_k} w_{ij},$$

$$P_1, P_2, \dots, P_K = \underset{P_k}{argmin}(w_{P_k}),$$

где $k \in (1,2, \dots, K)$.

В качестве алгоритма поиска K кратчайших путей можно использовать предоставленные алгоритмы роевого интеллекта или проектированную модель нейронной сети.

Шаг 3. Распределение трафика

- Разбиение трафика на потоки

После того как оптимальные пути $P_1, P_2, \dots, P_K$ между исходным коммутатором v_{src} и конечным коммутатором v_{dst} найдены, трафик между клиентом и сервером разбивается на несколько отдельных потоков. Определение этих потоков зависит от используемой стратегии балансировки нагрузки. В данной работе, iperf3 используется для инициализации параллельных потоков между каждой парой клиента и сервера, поэтому потоки имеют одинаковые IP-адреса источника и назначения (IP_{host_src} и IP_{host_dst}), но отличаются друг от друга портами источника и назначения ($port_{host\_src}$ и $port_{host\_dst}$):

$$f_i = \{IP_{host\_src}, IP_{host\_dst}, port^i_{host\_src}, port^i_{host\_dst}\}.$$

- Определение весов путей

Для каждого пути P_k, рассчитывается вес W_k, который обратно пропорционален стоимости пути w_{P_k}:

$$W_k = \frac{1}{w_{P_k}}.$$

Затем каждый вес W_k нормализуются и преобразуются в целые числа:

$$W_k^{normalized} = round(\frac{10 \times W_k}{\sum_{i=1}^{K} W_i}).$$

- Метод взвешенного циклического распределения

Если потоки данных распределяются случайным образом по маршрутам $P_1, P_2, \dots, P_K$ в соответствии с вероятностями, пропорциональными весам, результат распределения может не соответствовать ожидаемым пропорциям при небольшом количестве потоков. Для решения этой проблемы используется *метод взвешенного циклического распределения*.

Цель метода заключается в том, чтобы гарантировать, что даже при небольшом количестве потоков маршруты с большими весами получают больше потоков данных. Цикл C строится таким образом, чтобы маршруты $P_1, P_2, \dots, P_K$ появлялись в строго

упорядоченной последовательности. При этом маршрут P_k должен появляться ровно $W_k^{normalized}$ раз.

Потоки данных распределяются последовательно в соответствии с порядком в цикле C. После завершения одного цикла процесс продолжается с начала цикла до тех пор, пока все потоки не будут распределены по маршрутам.

Предположим, что имеется три маршрута с нормализованными весами: $W_1^{normalized} = 3,\ W_2^{normalized} = 2,\ W_3^{normalized} = 1$. На основе нормализованных весов формируется следующий цикл:

$$C = \{P_1, P_2, P_3, P_1, P_2, P_1\}.$$

Допустим, имеется 10 потоков $f_1, f_2, \dots, f_{10}$, которые необходимо распределить между маршрутами. Потоки распределяются по порядку в цикле C следующим образом:

$$f_1 \to P_1, f_2 \to P_2, f_3 \to P_3, f_4 \to P_1, f_5 \to P_2,$$
$$f_6 \to P_1, f_7 \to P_1, f_8 \to P_2, f_9 \to P_3, f_{10} \to P_1.$$

Если f_n должен быть назначен на маршрут P_k, то номер маршрута P_k в цикле распределения определяется по формуле:

$$P_k = C((n-1)\%\sum_{i=1}^{K} W_i^{normalized} + 1),$$

где $C(\cdot)$ – это функция, которая возвращает маршрут по его индексу в цикле C.

Шаг 4. Установка правил передачи

После определения пути для каждого потока компонент устанавливает правила передачи на каждом коммутаторе вдоль этого пути. Для каждого коммутатора v_i на пути P_k устанавливается правило $Flow\_entry_k(v_i)$:

$$Flow\_entry_k(v_i) = (match\ fields = \{IP_{host\_src}, IP_{host\_dst}, port_{host\_src}, port_{host\_dst}\},$$
$$actions = \{output = out\_port_{v_i}\}),$$

где $match\ fields$ – это информация о пакете, которую коммутатор использует для идентификации потока, соответствующего предопределенным полям, $actions$ – это действия, которые коммутатор выполняет, если пакет соответствует полям

соответствия, например, передача пакета через определенный порт out_port.

Этап 2. Адаптивная многопутевая маршрутизация

Адаптивная многопутевая маршрутизация используется для перераспределения трафика, если в сети обнаруживается перегрузка на одном или нескольких каналах. Этот процесс включает три ключевых шага: обнаружение перегруженных каналов, определение перегруженных потоков и их перенаправление с использованием новых маршрутов.

Шаг 1. Обнаружение перегруженных каналов

Для канала между двумя коммутаторами v_i и v_j перегрузка обнаруживается, если его стоимость превышает определенный порог θ. Список перегруженных каналов определяется следующим образом:

$$overloaded\_links = \{e_{ij} \in E | w_{ij} \geq \theta\}.$$

Шаг 2. Определение перегруженных потоков

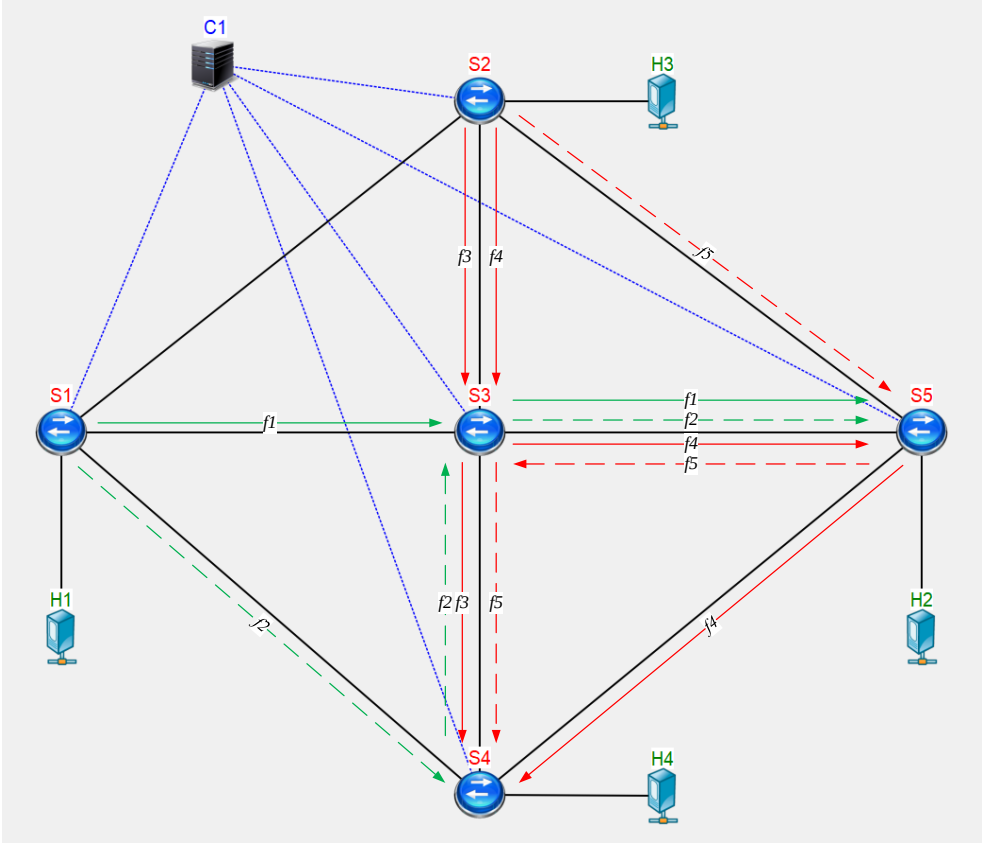


Рисунок 3.3 – Множество перегруженных потоков

- Формирование списка всех потоков через перегруженные каналы

  - Для каждого перегруженного канала e_{ij}, извлекается потоки $F_{ij} = \{f_1, f_2, \dots, f_n\}$, которые проходят через этот канал.

  - Все потоки F_{ij} объединяются в единый список F_{all}, содержащий уникальные потоки, проходящие через все перегруженные каналы:

$$F_{all} = \bigcup_{e_{ij} \in overloaded\_links} F_{ij}.$$

- Сортировка потоков по убыванию скорости

  - Рассчитывается суммарная скорость каждого потока через перегруженные каналы:

$$speed_{total}(f) = \sum_{e_{ij} \in overloaded\_links} speed_{ij}^{f}.$$

  - Список потоков F_{all} сортируется по $speed_{total}(f)$ в порядке убывания. Это гарантирует, что потоки, которые вносят наибольший вклад в перегрузку, будут рассмотрены в первую очередь.

- Определение минимального множества потоков для перенаправления

  - Инициализируется множество потоков для перенаправления $F_{redirect} = \emptyset$.

  - Итеративно просматривается список потоков, отсортированных по убыванию:

    - Временно добавляется поток f в $F_{redirect}$.

    - Проверяется, что после удаления всех потоков из $F_{redirect}$, оставшиеся потоки на каждом канале $e_{ij} \in overloaded\_links$ не вызывают перегрузку:

$$F_{ij}^{remain} = F_{ij} \backslash F_{redirect},$$

$$speed\left(F_{ij}^{remain}\right) = \sum_{f \in F_{ij}^{remain}} speed_{ij}^{f},$$

$$w_{ij}\left(F_{ij}^{remain}\right) = \frac{1}{1 - \frac{speed\left(F_{ij}^{remain}\right)}{bw_{ij}}},$$

где F_{ij}^{remain} – список оставшихся потоков на канале e_{ij}, $speed\left(F_{ij}^{remain}\right)$ – суммарная скорость оставшихся потоков и $w_{ij}\left(F_{ij}^{remain}\right)$ – стоимость канала после удаления подмножество потоков $F_{redirect}$.

- Если все каналы удовлетворяют $w_{ij}\left(F_{ij}^{remain}\right) \leq \theta$, процесс завершен.
- В противном случае переходим к следующему потоку.

В результате $F_{redirect}$ является минимальным множеством потоков, которые нужно перенаправить.

Шаг 3. Корректировка стоимости каналов

Если при перенаправлении перегруженных потоков для расчета маршрутов используется текущая стоимость каналов, результат может быть неточным, поскольку эта стоимость уже учитывает пропускную способность перегруженных потоков. Чтобы избежать этого, стоимость каналов пересчитывается, исключив пропускную способность потоков $f \in F_{redirect}$ на всех каналах, через которые проходят эти потоки.

- Для каждого потока $f \in F_{redirect}$, необходимо определить все каналы e_{ij}, через которые проходит этот поток, и вычесть его скорость $speed_{ij}^{f}$ из текущей загрузки канала.
- Новая пропускная способность канала u_{ij}^{new} после исключения перегруженных потоков рассчитывается следующим образом:

$$u_{ij}^{new} = u_{ij}^{used} - \sum_{f \in F_{redirect}} speed_{ij}^{f}.$$

- На основе новой пропускной способности u_{ij}^{new} рассчитывается новая стоимость канала w_{ij}^{new}, которая будет использована для расчета новых маршрутов:

$$w_{ij}^{new} = \frac{1}{1 - \frac{u_{ij}^{new}}{bw_{ij}}}.$$

Шаг 4. Перерасчет маршрутов для перегруженных потоков

После того как стоимость каналов пересчитана с учетом исключения перегруженных потоков, новые маршруты для потоков $F_{redirect}$ рассчитываются на основе обновленных метрик сети.

$$P'_1, P'_2, \dots, P'_K = \underset{P'_k}{argmin}(w_{P'_k}^{new}),$$

$$k \in (1,2, \dots, K),$$

где $w_{P_k}^{new}$ – это обновленная стоимость маршрутов на основе новой стоимости каналов после исключения перегруженных потоков.

Шаг 5. Распределение перегруженных потоков по новым маршрутам

Перегруженные потоки распределяются также по маршрутам $P'_1, P'_2, \dots, P'_K$ согласно описанному методу взвешенного циклического распределения.

Шаг 6. Обновление правил передачи

После определения маршрута для каждого перегруженного потока, компонент обновляет правила передачи на каждом коммутаторе вдоль нового пути $Flow\_entry'_k(v_i)$, заменяя старые правила $Flow\_entry_k(v_i)$.

ОСНОВНЫЕ РЕЗУЛЬТАТЫ И ВЫВОДЫ

1. Сформулирована задача балансировки потоков данных в программно-конфигурируемых сетях.

2. Предоставлены компоненты управления и мониторинга ПКС, позволяющие собирать информацию о состоянии сети в режиме реального времени, включая: мониторинг топологии сети, мониторинг портов и мониторинг потоков.

3. Разработаны математическая модель и метод динамической адаптивной многопутевой балансировки потоков данных (DAMLB) на основе компонентов мониторинга сети, позволяющие определять оптимальные маршруты, адаптирующиеся к динамическим условиям сети, и перенаправлять потоки данных при обнаружении перегрузки.

ГЛАВА 4 ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ ИНТЕЛЛЕКТУАЛЬНОЙ МАРШРУТИЗАЦИИ И БАЛАНСИРОВКИ ПОТОКОВ ДАННЫХ В ПКС

4.1 СТРУКТУРА ВИЗУАЛЬНОЙ ПРОГРАММНОЙ СИСТЕМЫ

Визуальная программная система SDNLoadBalancer – это передовая программная система, разработанная на платформе PyQt5, предназначенная для проектирования и эмуляции ПКС. Этот инструмент помогает сетевым инженерам и исследователям легко и точно разрабатывать, настраивать и моделировать топологии ПКС. С помощью интуитивно понятного графического интерфейса SDNLoadBalancer позволяет пользователям проектировать сложные сетевые инфраструктуры, разрабатывать сценарии тестирования и выполнять мониторинг сети в реальном времени.

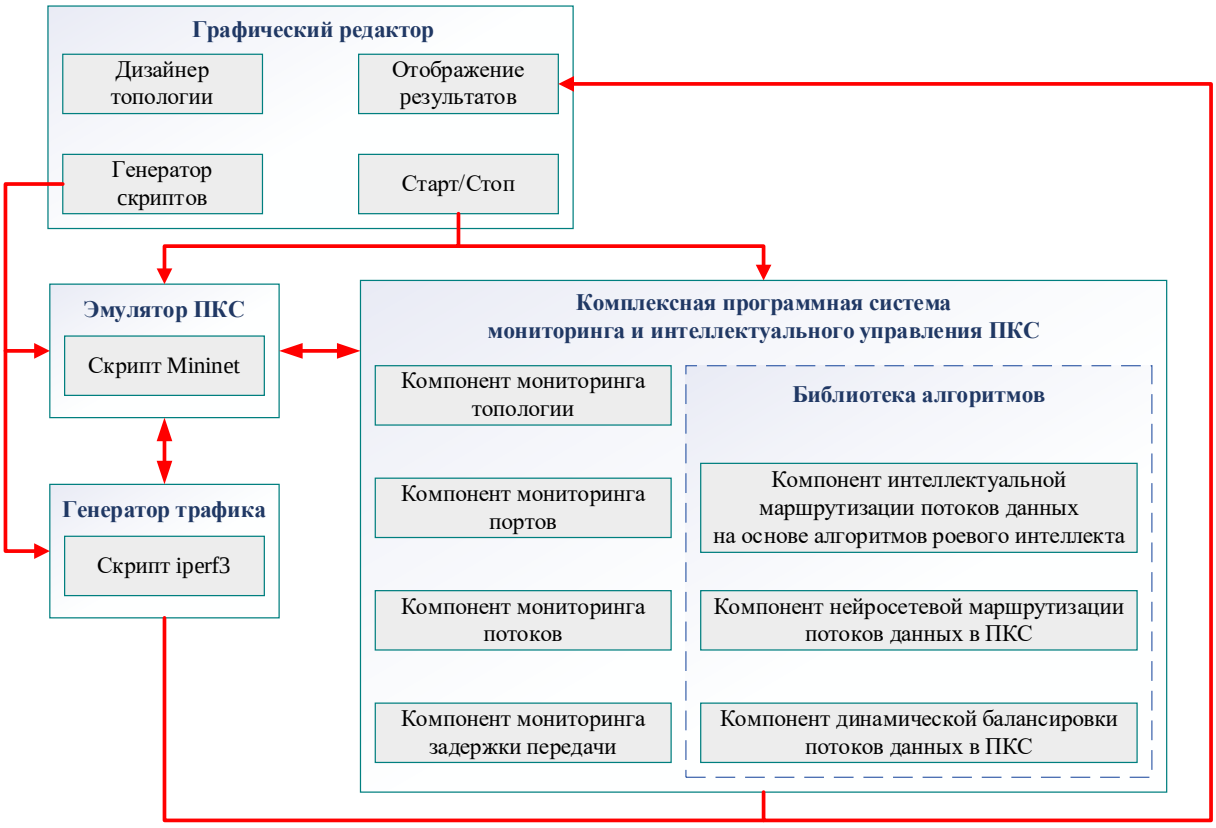


Рисунок 4.1 – Визуальная программная система SDNLoadBalancer

Визуальная программная система SDNLoadBalancer доступна на платформе Linux и включает следующие компоненты: графический редактор, эмулятор ПКС, генератор трафика и комплексную программную систему мониторинга и интеллектуального управления ПКС.

- SDNLoadBalancer интегрирует эмулятор Mininet, который позволяет создавать виртуальные коммутаторы, маршрутизаторы и хосты для моделирования топологии ПКС, что помогает тестировать и экспериментировать с сетевыми структурами без необходимости использовать реальное оборудование.
- Контроллер Ryu отвечает за мониторинг и управление ПКС. Ryu предоставляет API для координации коммутаторов в Mininet с целью реализации сетевых политик, таких как маршрутизация и управление потоками данных в сети.
- Iperf3 используется для генерации трафика и проверки пропускной способности между узлами сети в эмуляторе Mininet.

После проектирования и настройки структуры ПКС через графический интерфейс система автоматически генерирует соответствующие сценарии для Mininet, Ryu и iperf3. Всего одним щелчком мыши можно запустить эксперимент без необходимости вручную вводить команды в терминале. В процессе эксперимента динамические сетевые метрики, такие как пропускная способность, задержка и стоимость, будут отображаться в виде таблицы в реальном времени. Результаты эксперимента сохраняются в формате JSON, что позволяет пользователю легко строить графики для дальнейшего анализа и исследования производительности сети.

На рисунках 4.2 – 4.4 представлены диаграммы основных классов:

- графического редактора
- комплексной программной системы мониторинга и интеллектуального управления ПКС
- компонента интеллектуальной маршрутизации потоков данных на основе алгоритмов роевого интеллекта

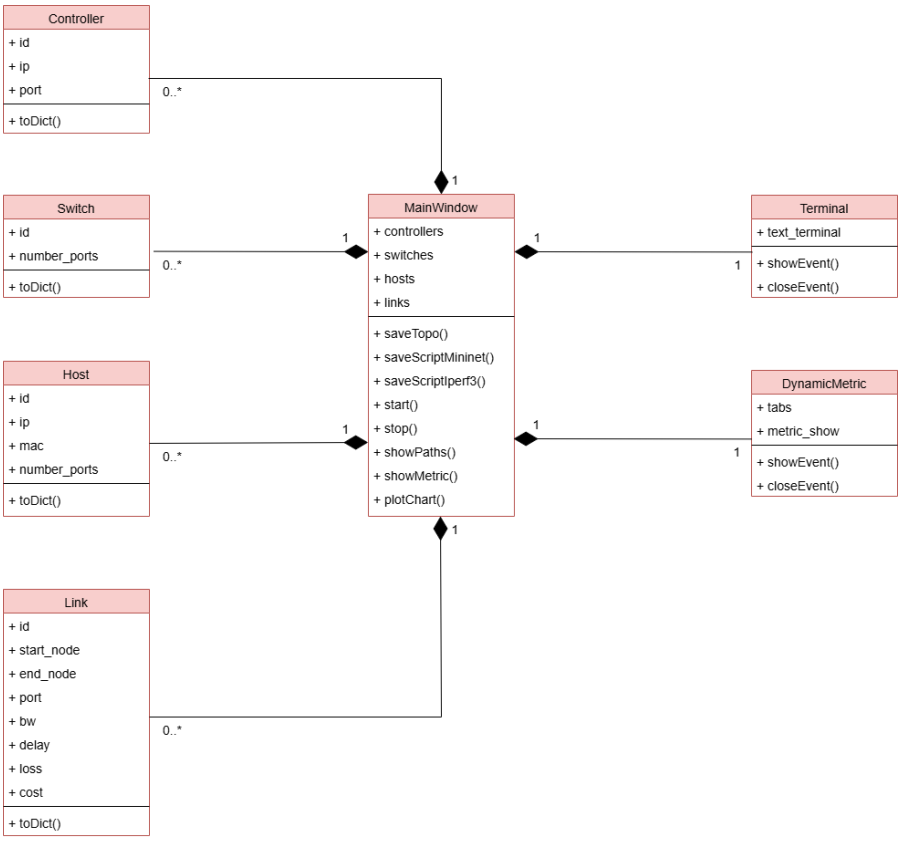


Рисунок 4.2 – Диаграмма основных классов графического редактора

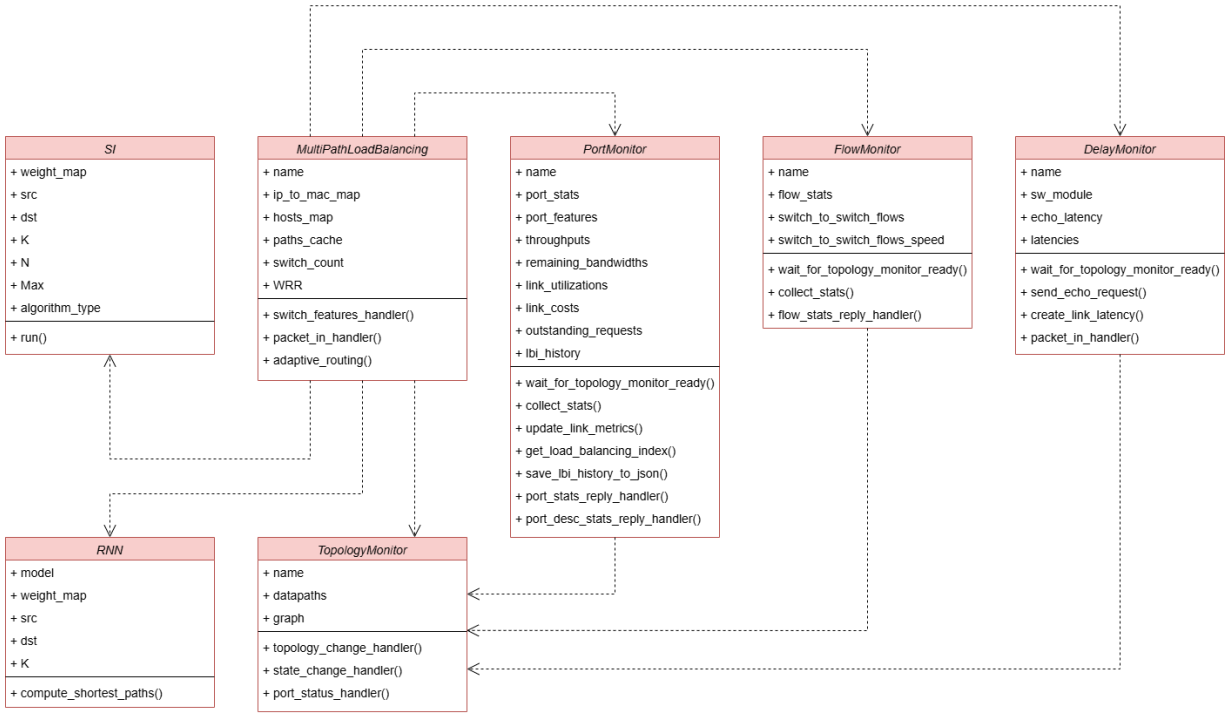


Рисунок 4.3 – Диаграмма основных классов комплексной программной системы мониторинга и интеллектуального управления ПКС

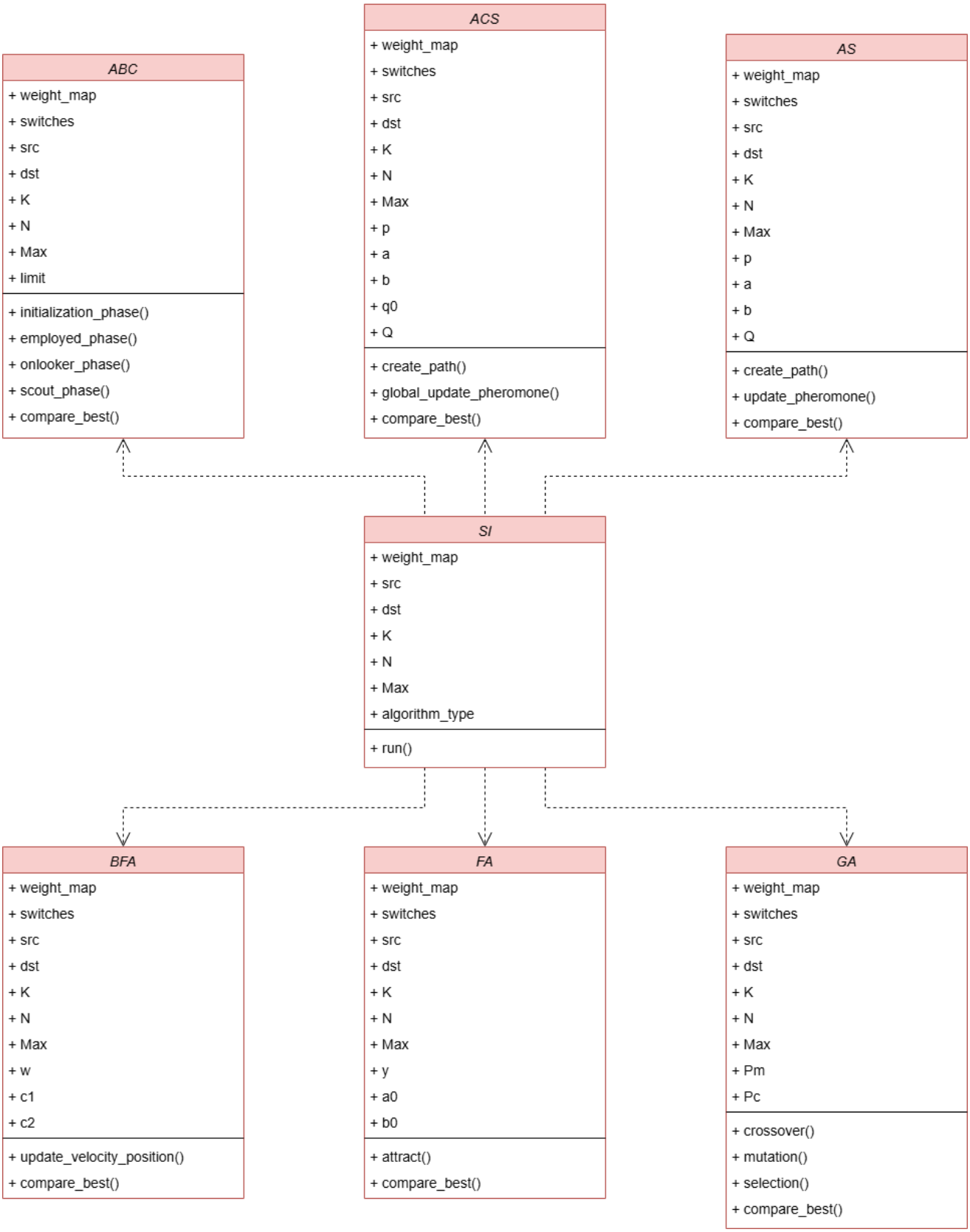


Рисунок 4.4 – Диаграмма основных классов компонента интеллектуальной маршрутизации потоков данных на основе алгоритмов роевого интеллекта

Визуальная программная система SDNLoadBalancer обладает рядом полезных и ключевых функций:

- **Проектирование топологии сети**: Интуитивно понятный интерфейс с функцией перетаскивания для проектирования и визуализации топологии сети, размещения и подключения сетевых компонентов (коммутаторы, хосты, контроллеры и соединения).
- **Конфигурация сетевых компонентов**: Детальная настройка параметров сетевых компонентов.
- **Интеграция с Mininet**: Прямая интеграция с эмулятором сети Mininet для инициализации и сохранения сценариев, отражающих разработанную топологию.
- **Запуск контроллера и эмулятора**: Запуск контроллера Ryu и эмулятора Mininet одним щелчком мыши.
- **Генерация трафика**: Запуск iperf3 по заранее настроенным сценариям на хостах для создания трафика.
- **Получение метрик в реальном времени**: Динамический мониторинг и отображение сетевых показателей в реальном времени в табличном формате.
- **Многопутевая маршрутизация и балансировки потоков данных**: Реализация и визуализация многопутевой маршрутизации и балансировки потоков данных с отображением путей различными цветами для облегчения анализа.
- **Построение диаграмм результатов**: Создание и сохранение графических представлений результатов маршрутизации для исторического анализа и составления отчетов.

4.2 ГРАФИЧЕСКИЙ РЕДАКТОР

На рисунке 4.5 представлены основные компоненты пользовательского интерфейса SDNLoadBalancer.

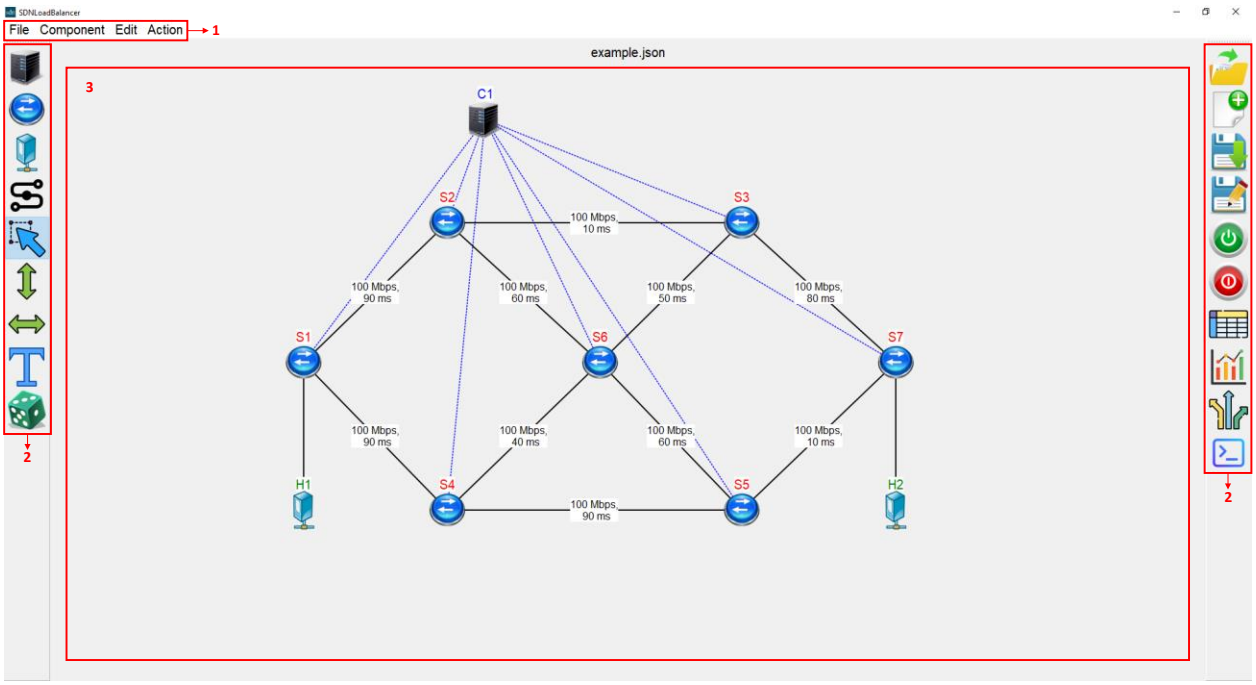


Рисунок 4.5 – Пользовательский интерфейс SDNLoadBalancer

Представленное окно состоит из следующих частей.

Панель главного меню

Строка главного меню расположена в верхней части окна и служит основной областью навигации. Она предлагает различные выпадающие меню, такие как «File», «Component», «Edit» и «Action». Каждое меню содержит список опций, позволяющих пользователю управлять файлами, настраивать параметры, редактировать компоненты и выполнять действия, связанные с проектированием и моделированием сетевой топологии.

- Выпадающее меню «File»: Этот раздел меню включает команды для создания новых проектов, открытия существующих файлов, сохранения текущей конфигурации сети и экспорта проектов или данных.

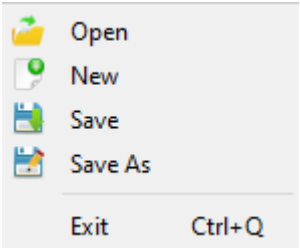


Рисунок 4.6 – Выпадающее меню «File»

- Выпадающее меню «Component»: Позволяет добавлять и настраивать различные сетевые устройства на рабочей области, такие как контроллеры, коммутаторы и хосты, а также устанавливать между ними проводные соединения.

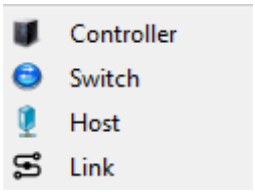


Рисунок 4.7 – Выпадающее меню «Component»

- Выпадающее меню «Edit»: Предлагает инструменты для изменения существующих сетевых проектов. Здесь можно выбирать и перемещать объекты в режиме указателя, создавать вертикальные и горизонтальные компоненты, добавлять текстовые метки и генерировать случайные метрики.

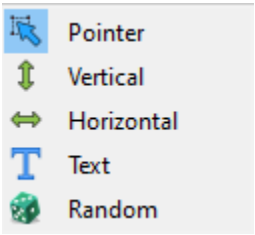


Рисунок 4.8 – Выпадающее меню «Edit»

- Выпадающее меню «Action»: Содержит команды для запуска эмуляции сети, создания и сохранения графических представлений результатов и другие подобные функции.

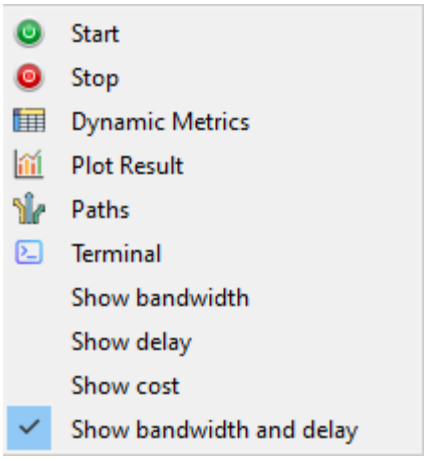


Рисунок 4.9 – Выпадающее меню «Action»

Панели инструментов

Интерфейс содержит две панели инструментов, расположенные слева и справа, что обеспечивает удобный доступ к инструментам и функциям.

- Левая панель инструментов: Предназначена для быстрого добавления и настройки сетевых компонентов (контроллеры, коммутаторы, хосты и соединения).

Рисунок 4.10 – Левая панель инструментов

- Кнопка «Controller» добавляет в топологию сети контроллер, необходимый для управления потоком трафика.
- Кнопка «Switch» помещает в сеть коммутатор, который обеспечивает управление потоком данных между различными сетевыми устройствами.
- Кнопка «Host» вводит в сеть хост, который обычно используется для конечных точек, таких как серверы или устройства пользователей.
- Кнопка «Link» создает соединения между элементами сети, определяя пути прохождения данных.
- Кнопка «Pointer» переключает режим взаимодействия на выбор, позволяя пользователям выбирать и манипулировать компонентами сети на рабочей области.
- Кнопка «Vertical» выравнивает выбранные компоненты сети по вертикали, способствуя организации макета для большей ясности и визуальной привлекательности.
- Кнопка «Horizontal» выравнивает выбранные элементы по горизонтали, что способствует поддержанию структурированной топологии сети.
- Кнопка «Text» позволяет добавлять текстовые метки к диаграмме сети, что помогает создавать аннотации и предоставлять дополнительную информацию о компонентах.

- Кнопка «Random» рандомизирует параметры, такие как пропускная способность, задержка и потери пакетов для сетевых соединений, что идеально подходит для тестирования различных условий и сценариев работы сети.

• Правая панель инструментов: Содержит инструменты для управления и анализа.

Рисунок 4.11 – Правая панель инструментов

- Кнопка «Open» открывает существующий файл топологии сети, позволяя пользователям продолжить работу над ранее сохраненными проектами.

- Кнопка «New» создает новую топологию сети, предоставляя чистый лист для запуска новых проектов.

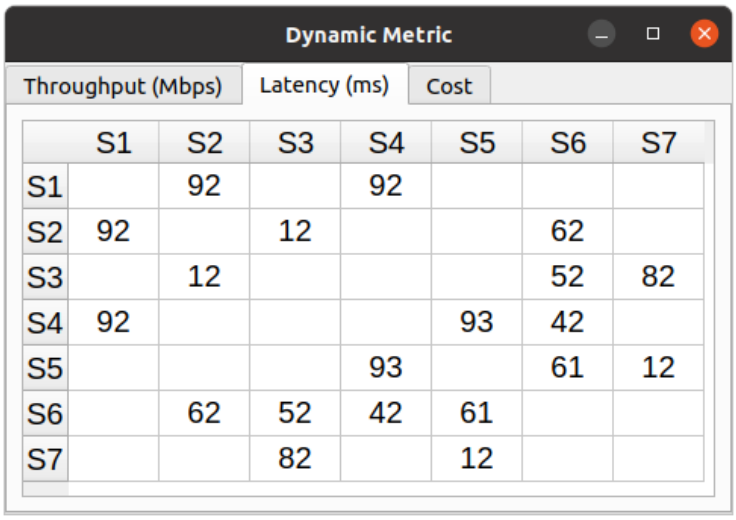
- Кнопка «Save» сохраняет текущую топологию сети вместе со всеми связанными настройками, фиксируя все последние изменения.

- Кнопка «Save as» сохраняет текущие настройки сети под новым именем файла, что упрощает создание нескольких версий одного проекта для сравнительного анализа или для резервного копирования.

- Кнопка «Start» запускает моделирование сети с использованием Mininet и контроллера Ryu на основе текущей конфигурации сети.

- Кнопка «Stop» завершает текущую симуляцию и очищает среду, останавливая процессы Ryu и Mininet.

- Кнопка «Dynamic Metric» отображает в режиме реального времени динамические метрики, такие как пропускная способность, задержка или стоимость, давая представление о производительности сети во время моделирования.



	S1	S2	S3	S4	S5	S6	S7
S1		92		92			
S2	92		12			62	
S3		12				52	82
S4	92				93	42	
S5				93		61	12
S6		62	52	42	61		
S7			82		12		

Рисунок 4.12 – Динамические метрики в режиме реального времени

- Кнопка «Plot» создает графическое представление данных моделирования, включая метрики, такие как пропускная способность или процент потери пакетов, что полезно для детального анализа и презентаций.

- Кнопка «Paths» визуализирует различные пути маршрутизации, определенные в ходе моделирования, разными цветами, облегчая дифференциацию и оценку эффективности каждого пути.

- Кнопка «Terminal» открывает терминал для отображения списка обнаруженных путей в сети, представленных последовательностями коммутаторов.

Рабочая область

Она является ключевым элементом интерфейса, позволяя пользователям проектировать, визуализировать и настраивать свои сети ПКС. Она представляет собой интерактивное пространство, на котором отображается топология сети, включая ее компоненты и связи. Для каждого соединения между компонентами отображаются важные показатели, такие как пропускная способность и задержка.

Дополнительные настройки

- Окно настройки параметров контроллера предоставляет удобный интерфейс для настройки основных рабочих параметров контроллера в ПКС. Этот диалог имеет

решающее значение для конфигурации параметров контроллера и сценария работы, определяющего его поведение в сети.

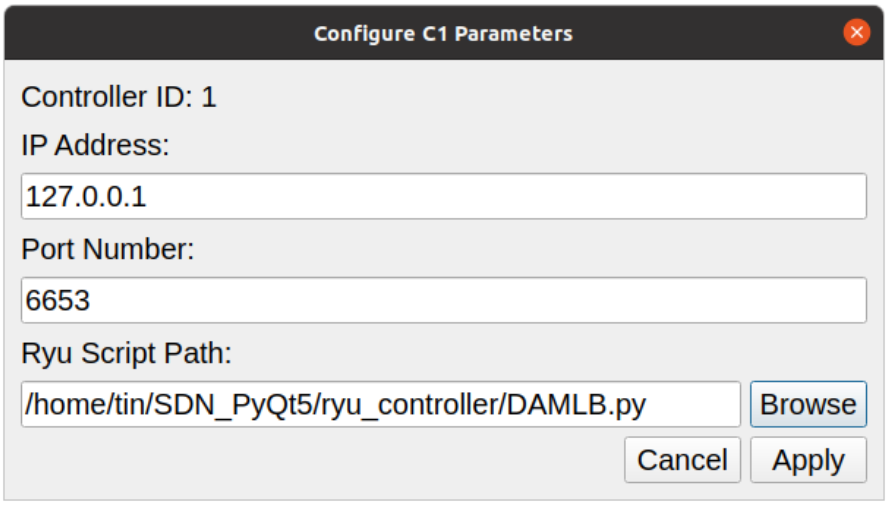


Рисунок 4.13 – Окно настройки параметров контроллера

- Поле «Controller ID» отображает уникальный числовой идентификатор контроллера, обеспечивающий его однозначную идентификацию в сети.
- Поле «IP Address» позволяет задать или изменить IP-адрес, по которому контроллер будет взаимодействовать с другими сетевыми устройствами.
- Поле «Port Number» предназначено для указания сетевого порта, который контроллер будет использовать для приема входящих соединений и данных. По умолчанию для OpenFlow установлен порт 6653.
- Поле «Ryu Script Path» служит для ввода пути к файлу сценария приложения Ryu, определяющего логику работы контроллера. Кнопка «Browse» рядом с этим полем упрощает поиск и выбор файла сценария в системе.
- Кнопка «Cancel» закрывает диалоговое окно конфигурации без сохранения изменений, оставляя настройки контроллера в исходном состоянии.
- Кнопка «Apply» применяет изменения параметров контроллера, внесенные в диалоговом окне, и обновляет настройки контроллера.

• Окно настройки метрик канала позволяет сетевым администраторам настраивать параметры соединения между коммутаторами.

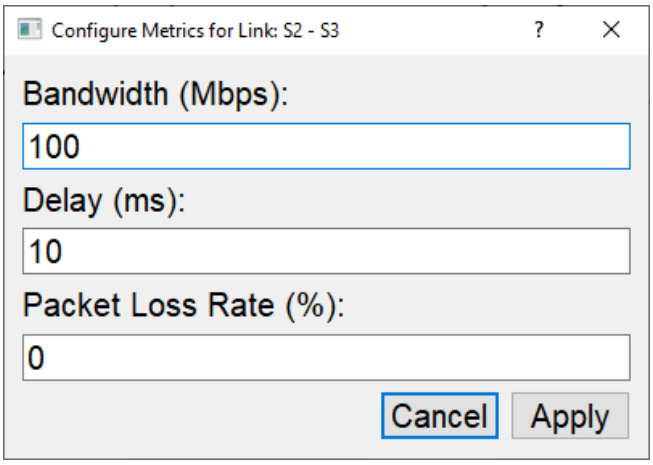


Рисунок 4.14 – Окно настройки метрик канала

- Поле «Bandwidth (Mbps)» позволяет указать максимальную скорость передачи данных по каналу между коммутаторами.
- Поле «Delay (ms)» служит для установки желаемой задержки соединения.
- Поле «Packet Loss Rate (%)» позволяет задать ожидаемый процент потерь пакетов при передаче.
- Кнопка «Cancel» закрывает диалоговое окно без сохранения изменений.
- Кнопка «Apply» сохраняет новые параметры сетевого канала.

• Окно настройки параметров хоста позволяет полностью настроить работу хоста в сети.

- Поле «Host ID» отображает уникальный числовой идентификатор, позволяющий однозначно определить хост в сети.
- Поле «Number of Ports» указывает количество сетевых портов, доступных на данном хосте.
- Поле «IP Address» позволяет указать сетевой адрес, который будет использовать хост для связи.
- Поле «MAC Address» отображает аппаратный адрес сетевого интерфейса хоста, необходимый для работы в локальной сети.

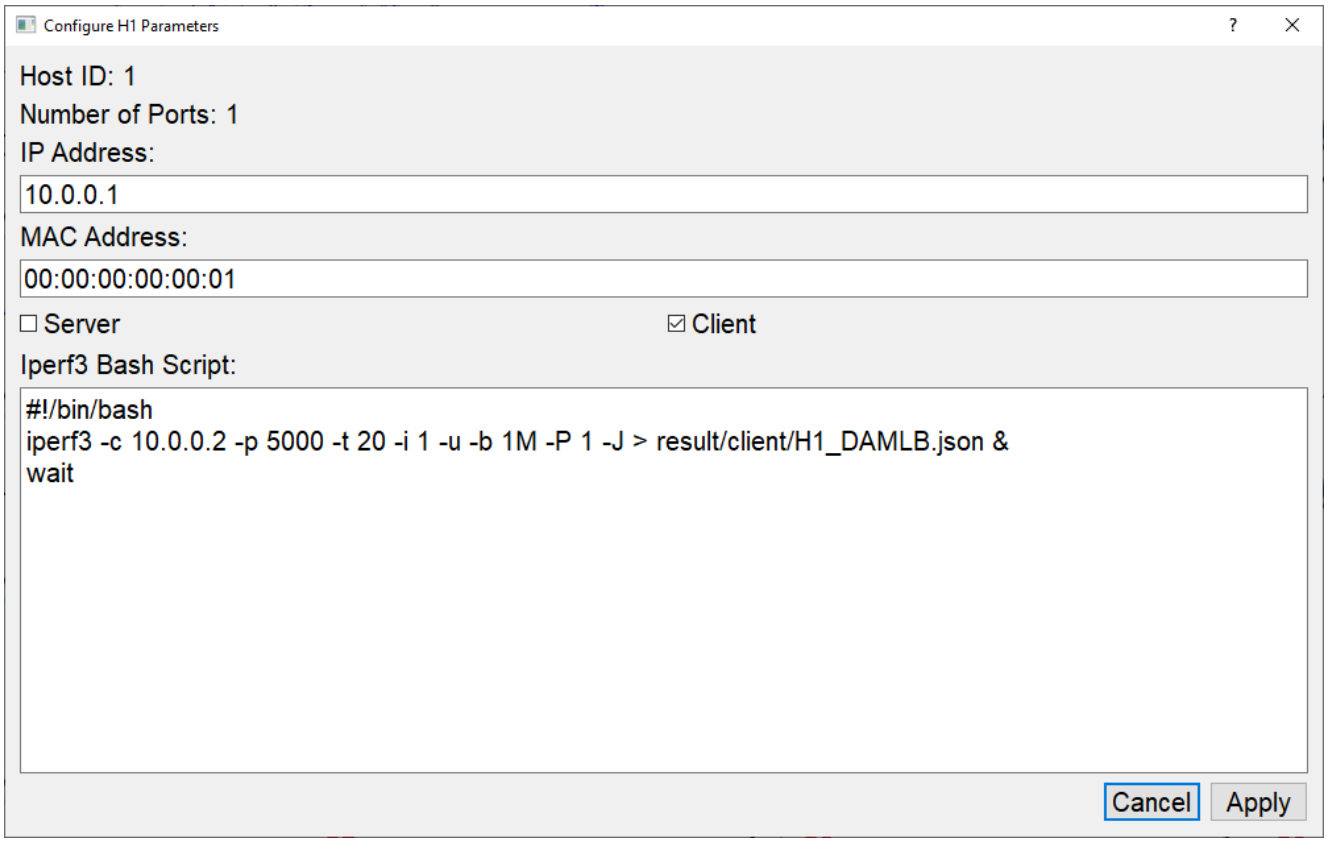


Рисунок 4.15 – Окно настройки параметров хоста

- Флажки «Server» и «Client» позволяют определить роль хоста в сети. Выбор «Server» или «Client» позволяет хосту выступать в качестве сервера или клиента соответственно для сетевых коммуникаций и сервисов.

- Поле «Iperf3 Bash Script» предназначено для ввода или изменения сценария Bash, автоматизирующего команды Iperf3 для тестирования производительности сети. Этот сценарий необходим для запуска сетевых тестов и сбора данных о пропускной способности и качестве соединения.

- Кнопка «Cancel» позволяет закрыть диалоговое окно настройки без сохранения изменений.

- Кнопка «Apply» сохраняет все внесенные в диалоговое окно изменения в конфигурации хоста.

4.3 ЭМУЛЯТОР ПКС

Визуальная программная система SDNLoadBalancer предоставляет расширенные возможности для оптимизации проектирования и анализа топологий программно-конфигурируемых сетей. Функция сохранения позволяет автоматически генерировать сценарии эмулятора Mininet на основе текущей конфигурации топологии.

- Импорт библиотек: Импортируются необходимые библиотеки и модули для создания и управления виртуальной сетью.

```
from mininet.net import Mininet
from mininet.node import RemoteController
from mininet.cli import CLI
from mininet.log import setLogLevel
from mininet.link import TCLink
from functools import partial
from mininet.topo import Topo
from mininet.term import makeTerm
from time import sleep
```

- Определение топологии: Создает структуру сети, добавляя и устанавливая сетевые компоненты, такие как хосты, коммутаторы и каналы связи.

```
class MyTopo(Topo):
    # Функция для создания топологии с хостами, коммутаторами и каналами
    def build(self, **param):
        # Определение хостов и коммутаторов
        H1 = self.addHost('H1', mac='00:00:00:00:00:01', ip='10.0.0.1')
        H2 = self.addHost('H2', mac='00:00:00:00:00:02', ip='10.0.0.2')
        S1 = self.addSwitch('S1')
        S2 = self.addSwitch('S2')
        S3 = self.addSwitch('S3')
        S4 = self.addSwitch('S4')
        S5 = self.addSwitch('S5')
        S6 = self.addSwitch('S6')
        S7 = self.addSwitch('S7')
        # Добавление каналов между узлами
        self.addLink(H1, S1, 1, 1, bw=1000, delay='1ms', loss=0)
        self.addLink(S1, S2, 2, 1, bw=100, delay='90ms', loss=0)
        self.addLink(S2, S3, 2, 1, bw=100, delay='10ms', loss=0)
```

```
        self.addLink(S3, S7, 2, 1, bw=100, delay='80ms', loss=0)
        self.addLink(S7, S5, 2, 1, bw=100, delay='10ms', loss=0)
        self.addLink(S5, S4, 2, 1, bw=100, delay='90ms', loss=0)
        self.addLink(S4, S1, 2, 3, bw=100, delay='90ms', loss=0)
        self.addLink(S2, S6, 3, 1, bw=100, delay='60ms', loss=0)
        self.addLink(S6, S4, 2, 3, bw=100, delay='40ms', loss=0)
        self.addLink(S6, S5, 3, 3, bw=100, delay='60ms', loss=0)
        self.addLink(S6, S3, 4, 3, bw=100, delay='50ms', loss=0)
        self.addLink(S7, H2, 3, 1, bw=1000, delay='1ms', loss=0)
```

- Инициализация и управление сетью: Сеть запускается на основе заданной топологии, настраивается контроллер, открываются терминалы для хостов и интерфейс командной строки для управления.

```
def topology():
    # Инициализация топологии
    topo = MyTopo()
    link = partial(TCLink)
    # Создание сети с удаленным контроллером
    net = Mininet(topo=topo, controller=RemoteController(name='C1', ip='127.0.0.1',
port=6653), link=link)
    # Запуск и настройка сети
    net.start()
    # Открытие терминалов для хостов H1 и H2
    sleep(2)
    makeTerm(net['H2'],cmd='bash traffic/H2.sh')
    sleep(2)
    makeTerm(net['H1'],cmd='bash traffic/H1.sh')
    CLI(net)  # Вход в командную строку Mininet
    net.stop()  # Остановка сети
```

- Запуск программы: Настраивается логирование, затем запускается функция, отвечающая за инициализацию и управление сетью. Это начальная точка выполнения сценария.

```
if __name__ == '__main__':
    setLogLevel('info')
    topology()
```

4.4 КОМПЛЕКСНАЯ ПРОГРАММНАЯ СИСТЕМА МОНИТОРИНГА И ИНТЕЛЛЕКТУАЛЬНОГО УПРАВЛЕНИЯ ПКС

4.4.1 КОМПОНЕНТ МОНИТОРИНГА ТОПОЛОГИИ

Этот компонент представляет собой приложение для платформы Ryu, предназначенное для мониторинга и управления топологией сети. Основные функции включают отслеживание изменений в топологии, таких как добавление или удаление коммутаторов и каналов, обновление состояния портов и предоставление графа сети в формате JSON для дальнейшего анализа.

- Импорт библиотек: Импортируются необходимые библиотеки из Ryu, а также модули для работы с топологией и графами. Эти библиотеки позволяют управлять сетевыми событиями, отслеживать изменения в топологии и строить граф для представления сети.

```
# Импорт библиотек для работы с Ryu и графами сети
from ryu.base import app_manager
from ryu.ofproto import ofproto_v1_3
from ryu.controller.handler import set_ev_cls, MAIN_DISPATCHER, DEAD_DISPATCHER
from ryu.topology import event
from ryu.controller import ofp_event
from ryu.topology.api import get_link, get_switch
import networkx as nx
import logging
```

- Инициализация приложения: Внутри класса инициализируются необходимые атрибуты, такие как словарь *datapaths* для хранения подключенных коммутаторов и граф *networkx.DiGraph* для хранения топологии сети.

```
# Класс приложения для мониторинга топологии сети
class TopologyMonitor(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
    def __init__(self, *_args, **_kwargs):
        # Инициализация атрибутов для хранения информации о сети
        super(TopologyMonitor, self).__init__(*_args, **_kwargs)
```

```
        self.name = "topology_monitor"  # Название приложения
        self.datapaths = {}  # Словарь для хранения datapath коммутаторов
        self.graph = nx.DiGraph()  # Граф для хранения топологии сети
```

- Обновление топологии: Функция отвечает за получение информации о текущих коммутаторах и каналах в сети с помощью API Ryu и обновление графа сети.

```
# Обновление топологии сети с использованием API Ryu
def update_topology(self):
    switch_list = get_switch(self, None)
    links = get_link(self, None)
    self.update_graph(switch_list, links)
```

- Обновление графа: Функция обновляет граф сети на основе полученной информации о коммутаторах и каналах. Для каждого коммутатора и канала в сети обновляется статус в графе, который используется для дальнейшего анализа топологии.

```
# Обновление графа сети с учетом состояния каналов и коммутаторов
def update_graph(self, switch_list, links):
    new_graph = nx.DiGraph(self.graph)
    for switch in switch_list:
        new_graph.add_node(switch.dp.id)
    for link in links:
        src = link.src
        dst = link.dst
        if           self.graph.has_edge(src.dpid,           dst.dpid)           and
self.graph[src.dpid][dst.dpid]['status'] == 'down':
            status = 'down'
        else:
            status = 'up'
        new_graph.add_edge(src.dpid,   dst.dpid,   weight=1,   src_port=src.port_no,
dst_port=dst.port_no, status=status)
    self.graph = new_graph
```

- Обработка событий изменения топологии: Эта функция реагирует на события изменения топологии, такие как добавление или удаление коммутаторов и каналов. При изменениях в сети функция вызывает обновление топологии.

```
# Обработка событий изменения топологии (добавление/удаление коммутаторов и каналов)
```

```
@set_ev_cls([event.EventSwitchEnter,   event.EventSwitchLeave,   event.EventLinkAdd,
event.EventLinkDelete])
def topology_change_handler(self, ev):
    self.update_topology()
```

- Обработка изменения состояния коммутаторов: Эта функция отслеживает изменения состояния коммутаторов (например, когда коммутатор включается или отключается). В зависимости от состояния (*MAIN_DISPATCHER* или *DEAD_DISPATCHER*) коммутатор добавляется или удаляется из словаря *datapaths*.

```
# Обработка событий изменения состояния коммутаторов (подключение/отключение)
@set_ev_cls(ofp_event.EventOFPStateChange, [MAIN_DISPATCHER, DEAD_DISPATCHER])
def state_change_handler(self, ev):
    datapath = ev.datapath
    if ev.state == MAIN_DISPATCHER:
        self.datapaths[datapath.id] = datapath
    elif ev.state == DEAD_DISPATCHER:
        self.datapaths.pop(datapath.id, None)
```

- Обработка состояния портов: Эта функция реагирует на изменения состояния портов (например, когда порт отключен или заблокирован). В зависимости от причины изменения статуса, обновляется информация о статусе канала в графе (например, канал помечается как «down», если порт недоступен).

```
# Обработка событий изменения состояния портов (доступность/недоступность портов)
@set_ev_cls(ofp_event.EventOFPPortStatus, MAIN_DISPATCHER)
def port_status_handler(self, ev):
    msg = ev.msg
    reason = msg.reason
    port_no = msg.desc.port_no
    dpid = msg.datapath.id
    ofproto = msg.datapath.ofproto
    for u, v, data in list(self.graph.edges(data=True)):
        if  (u  ==  dpid  and  data['src_port']  ==  port_no)  or  (v  ==  dpid  and
data['dst_port'] == port_no):
            if reason == ofproto.OFPPR_DELETE или reason == ofproto.OFPPR_MODIFY:
                if msg.desc.state & ofproto.OFPPS_LINK_DOWN:
                    self.graph[u][v]['status'] = 'down'
                elif msg.desc.state & ofproto.OFPPS_BLOCKED:
```

```
                    self.graph[u][v]['status'] = 'down'
                else:
                    self.graph[u][v]['status'] = 'up'
            elif reason == ofproto.OFPPR_ADD:
                self.graph[u][v]['status'] = 'up'
```

- Получение графа топологии: Эта функция возвращает текущий граф топологии сети в формате JSON, который может использоваться для внешней визуализации или анализа состояния сети.

```
# Получение графа топологии сети в формате JSON
def get_topology_graph(self):
    return nx.json_graph.node_link_data(self.graph)
```

4.4.2 КОМПОНЕНТ МОНИТОРИНГА ПОРТОВ

Компонент разработан для мониторинга статистических данных о портах сети. Он собирает информацию о статистике портов, вычисляет пропускную способность, коэффициенты загруженности каналов и индексы балансировки нагрузки. Компонент работает в режиме непрерывного мониторинга, регулярно обновляя метрики каналов на основе поступающих данных.

- Импорт библиотек: Импортируются библиотеки для работы с конфигурацией, управления событиями, математических вычислений и обработки данных в формате JSON.

```
# Импорт библиотек для работы с Ryu, сбором статистики и вычислений
from __future__ import division
from operator import attrgetter
from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import MAIN_DISPATCHER, set_ev_cls
from ryu.ofproto import ofproto_v1_3
from ryu.lib import hub
from math import pow
from ryu.base.app_manager import lookup_service_brick
from decimal import Decimal
```

```
from setting import PORT_PERIOD, LENGTH_LBI, MAX_CAPACITY, time_out
import json
```

- Инициализация приложения: Инициализируются атрибуты приложения для хранения информации о портах, метрик каналов и истории индексов балансировки нагрузки. Запускается основной поток мониторинга.

```
# Инициализация приложения PortMonitor с атрибутами для мониторинга сети
class PortMonitor(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
    def __init__(self, *args, **kwargs):
        # Инициализация атрибутов для хранения статистики портов и каналов
        super(PortMonitor, self).__init__(*args, **kwargs)
        self.name = 'port_monitor'  # Название приложения
        self.topology_monitor   =   lookup_service_brick('topology_monitor')      #
Мониторинг топологии
        self.port_stats = {}  # Статистика портов
        self.port_features = {}  # Характеристики портов
        self.throughputs = {}  # Пропускная способность
        self.remaining_bandwidths = {}  # Остаточная пропускная способность
        self.link_utilizations = {}  # Коэффициент загруженности канала
        self.link_costs = {}  # Стоимость каналов
        self.stats_reply_event = hub.Event()  # Событие ожидания статистики
        self.outstanding_requests = 0  # Количество ожидающих запросов
        self.lbi_history = []  # История индексов балансировки нагрузки
        self.monitor_thread  =  hub.spawn(self.monitor)    #  Запуск  мониторинга  в
отдельном потоке
```

- Основной цикл мониторинга: Приложение выполняет сбор статистики, обновляет метрики каналов и рассчитывает LBI в циклическом режиме. Индексы балансировки нагрузки сохраняются в файл JSON после накопления заданного объема данных.

```
# Основной цикл мониторинга сети, сбор данных и расчет LBI
def monitor(self):
    while True:
        try:
            if self.wait_for_topology_monitor_ready():
                self.collect_stats()
                self.update_link_metrics()
```

```
                lbi = self.get_load_balancing_index()
                if lbi is not None and len(self.lbi_history) < LENGTH_LBI:
                    self.lbi_history.append(lbi)
                if len(self.lbi_history) == LENGTH_LBI:
                    self.save_lbi_history_to_json("result/DAMLB.json")
            hub.sleep(PORT_PERIOD)
        except Exception as e:
            self.logger.error("Error in monitoring loop: %s", str(e))
            continue
```

- Проверка готовности мониторинга топологии: Перед сбором статистики приложение проверяет состояние мониторинга топологии, чтобы избежать ошибок при отсутствии данных о сетевой структуре.

```
# Проверка готовности мониторинга топологии
def wait_for_topology_monitor_ready(self):
    if self.topology_monitor is None:
        return False
    if not self.topology_monitor.datapaths:
        return False
    if not hasattr(self.topology_monitor, 'graph') or not
self.topology_monitor.graph:
        return False
    return True
```

- Сбор статистики портов: Приложение отправляет запросы статистики каждому коммутатору в сети. Ответы используются для обновления данных о пропускной способности портов.

```
# Сбор статистики портов
def collect_stats(self):
    self.stats_reply_event.clear()
    self.throughputs = {}
    self.remaining_bandwidths = {}
    self.link_utilizations = {}
    for datapath in self.topology_monitor.datapaths.values():
        self.port_features.setdefault(datapath.id, {})
        self.request_stats(datapath)
    if not self.stats_reply_event.wait(timeout=time_out):
        self.logger.warning("Stats reply timed out.")
```

```
        return

# Отправка запроса статистики к коммутатору
def request_stats(self, datapath):
    self.logger.debug('send stats request: %016x', datapath.id)
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser
    req = parser.OFPPortDescStatsRequest(datapath, 0)
    datapath.send_msg(req)
    req = parser.OFPPortStatsRequest(datapath, 0, ofproto.OFPP_ANY)
    datapath.send_msg(req)
    self.outstanding_requests += 2
```

- Обработка ответов на запросы: После получения ответов приложение обновляет информацию о скорости передачи данных на портах и их характеристиках, таких как максимальная пропускная способность порта.

```
# Обработка ответа на запрос статистики портов
@set_ev_cls(ofp_event.EventOFPPortStatsReply, MAIN_DISPATCHER)
def port_stats_reply_handler(self, ev):
    body = ev.msg.body
    dpid = ev.msg.datapath.id
    self.throughputs.setdefault(dpid, {})
    for stat in sorted(body, key=attrgetter('port_no')):
        port_no = stat.port_no
        if port_no != ofproto_v1_3.OFPP_LOCAL:
            key = (dpid, port_no)
            value = (stat.tx_bytes, stat.rx_bytes, stat.rx_errors,
                     stat.duration_sec, stat.duration_nsec)
            self.store_stats(self.port_stats, key, value, 5)
            self.calculate_port_speed(key)
    self.outstanding_requests -= 1
    if self.outstanding_requests == 0:
        self.stats_reply_event.set()

# Обработка ответа на запрос характеристик портов
@set_ev_cls(ofp_event.EventOFPPortDescStatsReply, MAIN_DISPATCHER)
def port_desc_stats_reply_handler(self, ev):
    msg = ev.msg
    dpid = msg.datapath.id
```

```
    for p in ev.msg.body:
        for dst in self.topology_monitor.graph[dpid]:
            if self.topology_monitor.graph[dpid][dst]['src_port'] == p.port_no:
                self.port_features[dpid].setdefault(dst, {})
                self.port_features[dpid][dst] = p.curr_speed / 1000 # 100Mbps
                break
    self.outstanding_requests -= 1
    if self.outstanding_requests == 0:
        self.stats_reply_event.set()
```

- Расчет пропускной способности портов: Скорость передачи данных на каждом порту рассчитывается на основе собранных данных статистики, таких как количество переданных байтов и продолжительность передачи.

```
# Расчет пропускной способности порта на основе собранной статистики
def calculate_port_speed(self, key):
    pre = 0
    period = 0
    port_stat_history = self.port_stats.get(key, [])
    if len(port_stat_history) > 1:
        pre = port_stat_history[-2][0]
        period = self.calculate_period(
            port_stat_history[-1][3], port_stat_history[-1][4],
            port_stat_history[-2][3], port_stat_history[-2][4]
        )
    now = port_stat_history[-1][0] if port_stat_history else 0
    speed = round(self.calculate_speed(now, pre, period), 1)
    src = key[0]
    for dst in self.topology_monitor.graph[src]:
        if self.topology_monitor.graph[src][dst]['src_port'] == key[1]:
            self.throughputs[src].setdefault(dst, {})
            self.throughputs[src][dst] = speed
            break
```

- Обновление метрик каналов: Приложение обновляет такие метрики, как оставшаяся пропускная способность, коэффициент загруженности каналов и стоимость каналов. Эти данные используются для дальнейших расчетов и анализа.

```
# Обновление метрик каналов и расчет стоимости каналов
def update_link_metrics(self):
```

```
    capacity = setting.MAX_CAPACITY
    new_link_costs = {}
    for src in self.topology_monitor.graph:
        self.remaining_bandwidths.setdefault(src, {})
        self.link_utilizations.setdefault(src, {})
        new_link_costs.setdefault(src, {})
        for dst in self.topology_monitor.graph[src]:
            if src != dst:
                speed = self.throughputs.get(src, {}).get(dst, 0)
                free_bandwidth = self.calculate_free_bandwidth(capacity, speed)
                self.remaining_bandwidths[src][dst] = free_bandwidth
                link_utilization = round(speed / capacity, 1)
                if self.topology_monitor.graph[src][dst]['status'] == 'down':
                    link_utilization = None
                self.link_utilizations[src][dst] = link_utilization
                if link_utilization == None:
                    new_link_costs[src][dst] = Decimal('1000')
                elif link_utilization == 1:
                    new_link_costs[src][dst] = Decimal('100')
                else:
                    new_link_costs[src][dst]   =   Decimal(str(round(1   /   (1   -
link_utilization), 1)))
    if new_link_costs:
        self.link_costs.clear()
        self.link_costs.update(new_link_costs)
```

- Расчет индекса балансировки нагрузки: Индекс рассчитывается на основе коэффициентов загруженности каналов, чтобы оценить эффективность распределения нагрузки в сети.

```
# Расчет индекса балансировки нагрузки на основе коэффициентов загруженности каналов
def calculate_load_balancing_index(self, utilizations):
    if not utilizations or len(utilizations) == 0:
        return None
    n = len(utilizations)
    sum_utilization = sum(utilizations)
    sum_square_utilization = sum([pow(u, 2) for u in utilizations])
    if sum_square_utilization == 0:
        return 1
    lbi = pow(sum_utilization, 2) / (n * sum_square_utilization)
```

```
    return lbi

# Получение индекса балансировки нагрузки по всем каналам
def get_load_balancing_index(self):
    utilizations = []
    for src in self.link_utilizations:
        for dst in self.link_utilizations[src]:
            utilization = self.link_utilizations[src][dst]
            if utilization is not None:
                utilizations.append(utilization)
    return self.calculate_load_balancing_index(utilizations)

# Сохранение истории индексов балансировки нагрузки в файл JSON
def save_lbi_history_to_json(self, file_name="result/lbi_history.json"):
    with open(file_name, 'w') как json_file:
        json.dump(self.lbi_history, json_file, indent=4)
```

- Вспомогательные функции.

```
# Хранение статистики портов в истории
def store_stats(self, stats_dict, key, value, length):
    if key not in stats_dict:
        stats_dict[key] = []
    stats_dict[key].append(value)
    if len(stats_dict[key]) > length:
        stats_dict[key].pop(0)

# Расчет оставшейся пропускной способности
def calculate_free_bandwidth(self, capacity, speed):
    return max(capacity - speed, 0)

# Расчет периода времени между двумя запросами статистики
def calculate_period(self, n_sec, n_nsec, p_sec, p_nsec):
    return self.get_time(n_sec, n_nsec) - self.get_time(p_sec, p_nsec)

# Преобразование времени в секундах и наносекундах в секунды
def get_time(self, sec, nsec):
    return sec + nsec / (10 ** 9)

# Расчет скорости передачи данных
```

```
def calculate_speed(self, now, pre, period):
    if period:
        return 8 * (now - pre) / (period * 10 ** 6)
    else:
        return 0.0
```

- Получение информации: Предоставляются методы для получения текущих данных о пропускной способности, оставшейся пропускной способности, коэффициентах загруженности и стоимости каналов.

```
# Получение данных о пропускной способности
def get_throughput(self):
    return self.throughputs

# Получение данных о оставшейся пропускной способности
def get_remaining_bandwidth(self):
    return self.remaining_bandwidths

# Получение данных о коэффициенте загруженности канала
def get_link_utilization(self):
    return self.link_utilizations

# Получение данных о стоимости каналов
def get_link_costs(self):
    return self.link_costs
```

4.4.3 КОМПОНЕНТ МОНИТОРИНГА ПОТОКОВ

Этот компонент предназначен для мониторинга потоков данных на коммутаторах. Он анализирует информацию о потоке, рассчитывает скорость передачи данных и регулярно обновляет статистику для каждого потока. Этот компонент работает в режиме активного мониторинга, периодически отправляя запросы коммутаторам для получения обновленных данных.

- Импорт библиотек: Импортируются необходимые библиотеки из Ryu и Python для работы с мониторингом потоков и управления событиями.

```
# Импорт библиотек для работы с Ryu, потоками и вычислениями
```

```
from __future__ import division
from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import MAIN_DISPATCHER, set_ev_cls
from ryu.ofproto import ofproto_v1_3
from ryu.lib import hub
from ryu.base.app_manager import lookup_service_brick
from setting import FLOW_PERIOD
```

- Инициализация приложения: В классе *FlowMonitor* инициализируются основные параметры, такие как статистика потоков, пропускная способность потоков между коммутаторами и метод для запуска мониторинга в отдельном потоке.

```
# Инициализация приложения FlowMonitor с атрибутами для мониторинга потоков между коммутаторами
class FlowMonitor(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
    def __init__(self, *args, **kwargs):
        # Инициализация атрибутов для хранения статистики потоков и скорости
        super(FlowMonitor, self).__init__(*args, **kwargs)
        self.name = 'flow_monitor'  # Название приложения
        self.topology_monitor = lookup_service_brick('topology_monitor')  # Мониторинг топологии
        self.flow_stats = {}  # Статистика потоков
        self.switch_to_switch_flows = {}  # Потоки между коммутаторами
        self.switch_to_switch_flows_speed = {}  # Скорость потоков между коммутаторами
        self.monitor_thread = hub.spawn(self.monitor)  # Запуск мониторинга в отдельном потоке
```

- Основной цикл мониторинга: Функция мониторинга выполняется циклически, запуская сбор статистики потоков на каждом коммутаторе.

```
# Основной цикл мониторинга, который собирает статистику потоков
def monitor(self):
    while True:
        try:
            if self.wait_for_topology_monitor_ready():
                self.collect_stats()
            hub.sleep(FLOW_PERIOD)
        except Exception as e:
```

```
            self.logger.error("Error in monitoring loop: %s", str(e))
            continue
```

- Проверка готовности мониторинга топологии.

```
# Проверка готовности мониторинга топологии
def wait_for_topology_monitor_ready(self):
    if self.topology_monitor is None:
        return False
    if not self.topology_monitor.datapaths:
        return False
    if not hasattr(self.topology_monitor, 'graph') or not
self.topology_monitor.graph:
        return False
    return True
```

- Сбор статистики потоков: Для каждого коммутатора отправляется запрос на получение статистики потоков. Это позволяет отслеживать активные потоки данных в сети.

```
# Сбор статистики потоков
def collect_stats(self):
    for datapath in self.topology_monitor.datapaths.values():
        self.request_stats(datapath)

# Отправка запроса статистики потоков к коммутатору
def request_stats(self, datapath):
    self.logger.debug('send stats request: %016x', datapath.id)
    parser = datapath.ofproto_parser
    req = parser.OFPFlowStatsRequest(datapath)
    datapath.send_msg(req)
```

- Обработка ответа на запрос статистики потоков: После получения ответа от коммутаторов, компонент анализирует статистику потоков и сохраняет ее для дальнейшего анализа.

```
# Обработка ответа на запрос статистики потоков
@set_ev_cls(ofp_event.EventOFPFlowStatsReply, MAIN_DISPATCHER)
def flow_stats_reply_handler(self, ev):
    body = ev.msg.body
    dpid = ev.msg.datapath.id
    self.flow_stats.setdefault(dpid, {})
```

```
    self.switch_to_switch_flows.setdefault(dpid, {})
    self.switch_to_switch_flows_speed.setdefault(dpid, {})
    for stat in sorted([flow for flow in body if flow.priority == 32768 and
flow.match.get],
                       key=lambda        flow:        (flow.match.get('in_port'),
flow.instructions[0].actions[0].port)):
        out_port = stat.instructions[0].actions[0].port
        key     =     (stat.match.get('ip_proto'),     stat.match.get('ipv4_src'),
stat.match.get('ipv4_dst'),
                stat.match.get('tcp_src'), stat.match.get('udp_src'))
        value = (stat.packet_count, stat.byte_count,
                    stat.duration_sec, stat.duration_nsec)
        dst_switch = self.get_dst_switch(dpid, out_port)
        if dst_switch:
            if dst_switch not in self.switch_to_switch_flows[dpid]:
                self.switch_to_switch_flows[dpid].setdefault(dst_switch, {})
                self.switch_to_switch_flows_speed[dpid].setdefault(dst_switch, {})
            self.store_stats(self.switch_to_switch_flows[dpid][dst_switch],    key,
value, 5)
            self.calculate_flow_speed(key, dpid, dst_switch)
        self.store_stats(self.flow_stats[dpid], key, value, 5)
```

- Расчет пропускной способности потоков: Рассчитывается скорость передачи данных каждого потока между коммутаторами на основе разницы в количестве переданных байтов за определенный период времени.

```
# Расчет скорости потока на основе статистики
def calculate_flow_speed(self, key, src, dst):
    pre = 0
    period = 0
    flow_stat_history = self.switch_to_switch_flows[src][dst][key]
    if len(flow_stat_history) > 1:
        pre = flow_stat_history[-2][1]
        period = self.calculate_period(
            flow_stat_history[-1][2], flow_stat_history[-1][3],
            flow_stat_history[-2][2], flow_stat_history[-2][3]
        )
    now = flow_stat_history[-1][1] if flow_stat_history else 0
    speed = round(self.calculate_speed(now, pre, period), 1)
    self.switch_to_switch_flows_speed[src][dst][key] = speed
```

- Вспомогательные функции.

```
# Хранение статистики портов в истории
def store_stats(self, stats_dict, key, value, length):
    if key not in stats_dict:
        stats_dict[key] = []
    stats_dict[key].append(value)
    if len(stats_dict[key]) > length:
        stats_dict[key].pop(0)

# Расчет периода времени между двумя запросами статистики
def calculate_period(self, n_sec, n_nsec, p_sec, p_nsec):
    return self.get_time(n_sec, n_nsec) - self.get_time(p_sec, p_nsec)

# Преобразование времени в секундах и наносекундах в секунды
def get_time(self, sec, nsec):
    return sec + nsec / (10 ** 9)

# Расчет скорости передачи данных
def calculate_speed(self, now, pre, period):
    if period:
        return 8 * (now - pre) / (period * 10 ** 6)
    else:
        return 0.0

# Получение коммутатора назначения на основе порта выхода
def get_dst_switch(self, src_dpid, out_port):
    if src_dpid in self.topology_monitor.graph:
        for dst_dpid in self.topology_monitor.graph[src_dpid]:
            link = self.topology_monitor.graph[src_dpid][dst_dpid]
            if link['src_port'] == out_port:
                return dst_dpid
    return None
```

4.4.4 КОМПОНЕНТ МОНИТОРИНГА ЗАДЕРЖКИ ПЕРЕДАЧИ

Компонент предназначен для мониторинга задержки каждого канала. Он собирает информацию о задержках между коммутаторами, используя как LLDP-

пакеты, так и echo-запросы OpenFlow. Полученные задержки сохраняются в графе топологии.

- Импорт библиотек: Импортируются необходимые библиотеки для работы с Ryu, задержками и обработкой сообщений OpenFlow.

```
# Импорт библиотек для работы с Ryu, OpenFlow и LLDP
from __future__ import division
from ryu.base import app_manager
from ryu.base.app_manager import lookup_service_brick
from ryu.controller import ofp_event
from ryu.controller.handler import MAIN_DISPATCHER, set_ev_cls
from ryu.ofproto import ofproto_v1_3
from ryu.lib import hub
from ryu.topology.switches import Switches, LLDPPacket
import time
from setting import DELAY_PERIOD
from topology_monitor import TopologyMonitor
```

- Инициализация приложения: В классе *DelayMonitor* инициализируются атрибуты для хранения задержек, таких как задержки echo-запросов и задержки каналов между коммутаторами. Также запускается поток мониторинга задержек.

```
# Инициализация приложения DelayMonitor с атрибутами для мониторинга задержек
class DelayMonitor(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
    def __init__(self, *args, **kwargs):
        # Инициализация приложения DelayMonitor
        super(DelayMonitor, self).__init__(*args, **kwargs)
        self.name = 'delay_monitor'  # Название приложения
        self.topology_monitor:                    TopologyMonitor                    =
lookup_service_brick('topology_monitor')  # Мониторинг топологии
        self.sw_module:  Switches  =  lookup_service_brick('switches')    #  Модуль
переключателей
        self.echo_latency = {}  # Словарь для хранения задержек echo-запросов
        self.latencies = {}  # Словарь для хранения задержек каналов
        self.sending_echo_request_interval = 0.05  # Интервал между echo-запросами
        self.monitor_thread = hub.spawn(self.monitor)  # Запуск потока измерений
```

- Основной цикл мониторинга: Поток мониторинга выполняется циклически, отправляя echo-запросы и вычисляя задержки каналов между коммутаторами.

```
# Основной цикл мониторинга задержек
def monitor(self):
    while True:
        try:
            if self.wait_for_topology_monitor_ready():
                self.send_echo_request()
                self.create_link_latency()
            hub.sleep(DELAY_PERIOD)
        except Exception as e:
            self.logger.error("Error in monitoring loop: %s", str(e))
            continue
```

- Проверка готовности мониторинга топологии.

```
# Проверка готовности мониторинга топологии
def wait_for_topology_monitor_ready(self):
    if self.topology_monitor is None:
        return False
    if not self.topology_monitor.datapaths:
        return False
    if not hasattr(self.topology_monitor, 'graph') or not
self.topology_monitor.graph:
        return False
    return True
```

- Отправка echo-запросов: Для каждого коммутатора отправляются echo-запросы, которые позволяют измерить задержки в сети на основе времени отклика.

```
# Отправка echo-запросов для каждого коммутатора
def send_echo_request(self):
    for datapath in list(self.topology_monitor.datapaths.values()):
        parser = datapath.ofproto_parser
        data_time = "%.12f" % time.time()
        byte_arr = bytearray(data_time.encode())
        echo_req = parser.OFPEchoRequest(datapath, data=byte_arr)
        datapath.send_msg(echo_req)
        hub.sleep(self.sending_echo_request_interval)
```

- Обработка ответа на echo-запрос: После получения ответа от коммутаторов рассчитывается задержка на основе времени запроса и ответа.

```
# Обработка ответа на echo-запрос
@set_ev_cls(ofp_event.EventOFPEchoReply, MAIN_DISPATCHER)
def echo_reply_handler(self, ev):
    now_timestamp = time.time()
    try:
        latency = now_timestamp - eval(ev.msg.data)
        self.echo_latency[ev.msg.datapath.id] = latency
    except:
        return
```

- Расчет задержек каналов: На основе данных о задержках LLDP и echo-запросов, функция вычисляет общие задержки между двумя коммутаторами и обновляет их в графе топологии.

```
# Расчет общей задержки между двумя коммутаторами
def calculate_latency(self, src, dst):
    try:
        fwd_delay = self.topology_monitor.graph[src][dst]['lldpdelay']
        re_delay = self.topology_monitor.graph[dst][src]['lldpdelay']
        src_latency = self.echo_latency[src]
        dst_latency = self.echo_latency[dst]
        latency = round((fwd_delay + re_delay - src_latency - dst_latency) * 1000 /
2)
        return max(latency, 0)
    except:
        return None
```

- Создание задержек каналов: Функция обходит все каналы между коммутаторами и обновляет значения задержек в графе топологии.

```
# Создание и обновление задержек каналов между коммутаторами
def create_link_latency(self):
    new_latencies = {}
    for src in self.topology_monitor.graph:
        new_latencies.setdefault(src, {})
        for dst in self.topology_monitor.graph[src]:
            new_latencies[src].setdefault(dst, {})
            if src == dst:
```

```
                continue
            latency = self.calculate_latency(src, dst)
            new_latencies[src][dst] = latency
    if new_latencies:
        self.latencies.update(new_latencies)
```

- Обработка входящих пакетов LLDP: Эта функция обрабатывает входящие LLDP-пакеты для определения задержек между коммутаторами и обновления данных в графе.

```
# Обработка входящих пакетов, особенно для обработки LLDP
@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
def packet_in_handler(self, ev):
    msg = ev.msg
    try:
        src_dpid, src_port_no = LLDPPacket.lldp_parse(msg.data)
        dpid = msg.datapath.id
        if self.sw_module is None:
            self.sw_module = lookup_service_brick('switches')
        for port in self.sw_module.ports.keys():
            if src_dpid == port.dpid and src_port_no == port.port_no:
                delay = self.sw_module.ports[port].delay
                self.save_lldp_delay(src=src_dpid, dst=dpid, lldpdelay=delay)
    except LLDPPacket.LLDPUnknownFormat as e:
        return
```

- Сохранение задержки LLDP: Функция сохраняет задержки, рассчитанные с использованием LLDP, для дальнейшего анализа.

```
# Сохранение задержки LLDP между двумя коммутаторами
def save_lldp_delay(self, src=0, dst=0, lldpdelay=0):
    try:
        self.topology_monitor.graph[src][dst]['lldpdelay'] = lldpdelay
    except:
        if self.topology_monitor is None:
            self.topology_monitor = lookup_service_brick('topology_monitor')
        return
```

- Получение информации о задержках: Функция возвращает текущие значения задержек в сети.

```
# Получение данных о задержках
```

```
def get_latency(self):
    return self.latencies
```

4.4.5 КОМПОНЕНТ ИНТЕЛЛЕКТУАЛЬНОЙ МНОГОПУТЕВОЙ МАРШРУТИЗАЦИИ И БАЛАНСИРОВКИ ПОТОКОВ ДАННЫХ

Компонент является ключевым компонентом для реализации процессов интеллектуальной многопутевой маршрутизации и балансировки потоков данных в ПКС. Он объединяет функции мониторинга топологии, портов, задержек и потоков для сбора информации о состоянии сети, необходимой для динамической многопутевой маршрутизации и балансировки трафика. В случае возникновения перегрузки компонент быстро рассчитывает новые маршруты для перенаправления потоков, вызывающих перегрузку.

- Импорт библиотек: Импортируются библиотеки для работы с Ryu, обработки событий OpenFlow, поддержки различных сетевых протоколов (ARP, ICMP, TCP, UDP) и реализации вспомогательных компонентов (мониторинг топологии, портов, потоков и задержки), а также алгоритмов маршрутизации.

```
from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import CONFIG_DISPATCHER, MAIN_DISPATCHER, set_ev_cls
from ryu.ofproto import ofproto_v1_3
from ryu.lib.packet import packet, arp, tcp, udp, ethernet, ipv4, ipv6, icmp
from ryu.lib import hub
from ryu.app.wsgi import WSGIApplication, ControllerBase, Response, route
import json
from decimal import Decimal
import copy
from setting import REROUTING_PERIOD, K, MAX_CAPACITY, MAX_VALUE
from tensorflow.keras.models import load_model
from delay_monitor import DelayMonitor
from port_monitor import PortMonitor
from topology_monitor import TopologyMonitor
from flow_monitor import FlowMonitor
```

```
from Yen import Yen
from SI import SI
from RNN import RNN
```

- Инициализация приложения: В классе *MultiPathLoadBalancing* инициализируются компоненты мониторинга (топологии, портов, потоков и задержки), создаются таблицы для хранения данных (ARP, хосты, кеш путей маршрутизации) и запускается фоновая задача для адаптивной маршрутизации.

```
class MultiPathLoadBalancing(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
    _CONTEXTS = {
            'wsgi': WSGIApplication,
            'topology_monitor': TopologyMonitor,
            'port_monitor': PortMonitor,
            'delay_monitor': DelayMonitor,
            'flow_monitor': FlowMonitor
        }
    def __init__(self, *_args, **_kwargs):
        super(MultiPathLoadBalancing, self).__init__(*_args, **_kwargs)
        self.name = 'multipath_load_balancing'
        wsgi: WSGIApplication = _kwargs['wsgi']
        wsgi.register(NetworkStatRest, {'rest_api_app': self})
        # Мониторинг топологии, портов, задержек и потоков
        self.topology_monitor: TopologyMonitor = _kwargs['topology_monitor']
        self.port_monitor: PortMonitor = _kwargs['port_monitor']
        self.delay_monitor: DelayMonitor = _kwargs['delay_monitor']
        self.flow_monitor: FlowMonitor = _kwargs['flow_monitor']
        # Инициализация таблиц и переменных
        self.ip_to_mac_map = {}
        self.hosts_map = {}
        self.paths_cache = {}
        self.switch_count = 0
        self.WRR = {}
        self.model = load_model('model.h5', compile=False)
        # Запуск фоновой задачи для маршрутизации
        self.adaptive_routing_thread = hub.spawn(self.adaptive_routing)
```

- Обработка поступающих пакетов: Функция обрабатывает поступающие на контроллер пакеты. В зависимости от типа пакета (ARP, ICMP, TCP или UDP) вызывает соответствующую функцию обработки.

```
# Обработка события прихода пакета на контроллер
@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
def packet_in_handler(self, ev):
    msg = ev.msg
    datapath = msg.datapath
    parser = datapath.ofproto_parser
    in_port = msg.match['in_port']
    pkt = packet.Packet(msg.data)
    eth = pkt.get_protocol(ethernet.ethernet)
    arp_pkt = pkt.get_protocol(arp.arp)
    udp_pkt = pkt.get_protocol(udp.udp)
    tcp_pkt = pkt.get_protocol(tcp.tcp)
    ip_pkt = pkt.get_protocol(ipv4.ipv4)
    icmp_pkt = pkt.get_protocol(icmp.icmp)
    if eth.ethertype == 35020:
        return
    if pkt.get_protocol(ipv6.ipv6):
        match = parser.OFPMatch(eth_type=eth.ethertype)
        actions = []
        self.install_flow(datapath, 1, match, actions)
        return None
    mac_src = eth.src
    dpid = datapath.id
    if mac_src not in self.hosts_map:
        self.hosts_map[mac_src] = (dpid, in_port)
    if arp_pkt:
        self.handle_arp(datapath, in_port, pkt, arp_pkt, msg)
        return
    if icmp_pkt:
        self.handle_icmp(datapath, in_port, pkt, ip_pkt, icmp_pkt, msg)
        return
    if tcp_pkt or udp_pkt:
        self.handle_tcp_udp(datapath, in_port, pkt, ip_pkt, tcp_pkt, udp_pkt, msg)
        return
```

- Обработка ARP: Обрабатывает ARP-запросы и ответы. При запросе проверяет, существует ли MAC-адрес для указанного IP, и отправляет ARP-ответ. Если MAC-адрес неизвестен, пакет транслируется по сети.

```
# Обработка ARP
def handle_arp(self, datapath, in_port, pkt, arp_pkt, msg):
    parser = datapath.ofproto_parser
    ofproto = datapath.ofproto
    eth = pkt.get_protocol(ethernet.ethernet)
    src_ip = arp_pkt.src_ip
    dst_ip = arp_pkt.dst_ip
    mac_src = eth.src
    mac_dst = eth.dst
    self.ip_to_mac_map[src_ip] = mac_src
    if arp_pkt.opcode == arp.ARP_REQUEST:
        if dst_ip in self.ip_to_mac_map:
            dst_mac = self.ip_to_mac_map[dst_ip]
            arp_reply = packet.Packet()
            arp_reply.add_protocol(
                ethernet.ethernet(
                    ethertype=eth.ethertype,
                    src=dst_mac,
                    dst=mac_src
                )
            )
            arp_reply.add_protocol(
                arp.arp(
                    opcode=arp.ARP_REPLY,
                    src_mac=dst_mac,
                    src_ip=dst_ip,
                    dst_mac=mac_src,
                    dst_ip=src_ip
                )
            )
            arp_reply.serialize()
            actions = [parser.OFPActionOutput(in_port)]
            out = parser.OFPPacketOut(
                datapath=datapath, buffer_id=ofproto.OFP_NO_BUFFER,
                in_port=ofproto.OFPP_CONTROLLER,
```

```
                actions=actions, data=arp_reply.data
            )
            datapath.send_msg(out)
        else:
            actions = [parser.OFPActionOutput(ofproto.OFPP_FLOOD)]
            out = parser.OFPPacketOut(
                datapath=datapath, buffer_id=msg.buffer_id,
                in_port=in_port, actions=actions, data=msg.data
            )
            datapath.send_msg(out)
    elif arp_pkt.opcode == arp.ARP_REPLY:
        if mac_dst in self.hosts_map:
            out_port = self.hosts_map[mac_dst][1]
            actions = [parser.OFPActionOutput(out_port)]
            out = parser.OFPPacketOut(
                datapath=datapath, buffer_id=msg.buffer_id,
                in_port=in_port, actions=actions, data=msg.data
            )
            datapath.send_msg(out)
```

- Обработка ICMP: Отвечает на ICMP-запросы (например, ping) формированием ICMP Echo Reply. Если запрос – не ping, перенаправляет пакет в соответствующий порт назначения.

```
# Обработка ICMP
def handle_icmp(self, datapath, in_port, pkt, ip_pkt, icmp_pkt, msg):
    parser = datapath.ofproto_parser
    ofproto = datapath.ofproto
    eth = pkt.get_protocol(ethernet.ethernet)
    src_ip = ip_pkt.src
    dst_ip = ip_pkt.dst
    mac_src = eth.src
    mac_dst = eth.dst
    if icmp_pkt.type == icmp.ICMP_ECHO_REQUEST:
        icmp_reply = packet.Packet()
        icmp_reply.add_protocol(
            ethernet.ethernet(
                ethertype=eth.ethertype,
                src=mac_dst,
                dst=mac_src
```

```
            )
        )
        icmp_reply.add_protocol(
            ipv4.ipv4(
                dst=src_ip,
                src=dst_ip,
                proto=ip_pkt.proto
            )
        )
        icmp_reply.add_protocol(
            icmp.icmp(
                type_=icmp.ICMP_ECHO_REPLY,
                code=icmp.ICMP_ECHO_REPLY_CODE,
                csum=0,
                data=icmp_pkt.data
            )
        )
        icmp_reply.serialize()
        actions = [parser.OFPActionOutput(in_port)]
        out = parser.OFPPacketOut(
            datapath=datapath, buffer_id=ofproto.OFP_NO_BUFFER,
            in_port=ofproto.OFPP_CONTROLLER,
            actions=actions, data=icmp_reply.data
        )
        datapath.send_msg(out)
    else:
        if mac_dst in self.hosts_map:
            out_port = self.hosts_map[mac_dst][1]
            actions = [parser.OFPActionOutput(out_port)]
            out = parser.OFPPacketOut(
                datapath=datapath, buffer_id=msg.buffer_id,
                in_port=in_port, actions=actions, data=msg.data
            )
            datapath.send_msg(out)
```

- Обработка TCP и UDP: Обрабатывает TCP и UDP пакеты. Вычисляет лучший маршрут между источником и получателем или использует сохраненные маршруты из кеша, после чего устанавливает соответствующие потоки в коммутаторы.

```
# Обработка TCP и UDP
def handle_tcp_udp(self, datapath, in_port, pkt, ip_pkt, tcp_pkt, udp_pkt, msg):
    parser = datapath.ofproto_parser
    ofproto = datapath.ofproto
    eth = pkt.get_protocol(ethernet.ethernet)
    src_ip = ip_pkt.src
    dst_ip = ip_pkt.dst
    mac_src = eth.src
    mac_dst = eth.dst
    src_sw_in_port = self.hosts_map[mac_src]
    dst_sw_in_port = self.hosts_map[mac_dst]
    proto_type = 'tcp' if tcp_pkt else 'udp'
    key     =     (src_sw_in_port[0],     src_sw_in_port[1],     dst_sw_in_port[0],
dst_sw_in_port[1], proto_type)
    if key not in self.paths_cache:
        paths_nodes,            paths_edges,            paths_weights            =
self.calculate_best_paths(src_sw_in_port[0], dst_sw_in_port[0])
        normalized_weights = self.normalize(paths_weights)
        weights = [int(round(i * 10)) for i in normalized_weights]
        src_dst = f"Using DAMLB: {proto_type.upper()} {src_ip} --> {dst_ip}"
        streams = set()
        self.WRR[key] = [weights, 1]
        initial_index    =    self.weighted_periodic_distribution(self.WRR[key][0],
self.WRR[key][1])
        streams.add((initial_index, tcp_pkt, udp_pkt))
        self.paths_cache[key] = [paths_nodes, paths_edges, paths_weights, src_ip,
dst_ip, src_dst, streams]
        out_port   =   self.set_paths_tcp_udp(src_sw_in_port[0],   src_sw_in_port[1],
dst_sw_in_port[0], dst_sw_in_port[1],
                                              src_ip, dst_ip, paths_nodes, initial_index,
tcp_pkt, udp_pkt)
        actions = [parser.OFPActionOutput(out_port)]
        data = None
        if msg.buffer_id == ofproto.OFP_NO_BUFFER:
            data = msg.data
        out = parser.OFPPacketOut(
            datapath=datapath, buffer_id=msg.buffer_id, in_port=in_port,
            actions=actions, data=data)
        datapath.send_msg(out)
```

```
    else:
        paths_nodes, paths_edges, paths_weights, *_ = self.paths_cache[key]
        self.WRR[key][1] += 1
        next_index      =      self.weighted_periodic_distribution(self.WRR[key][0],
self.WRR[key][1])
        self.paths_cache[key][-1].add((next_index, tcp_pkt, udp_pkt))
        out_port   =   self.set_paths_tcp_udp(src_sw_in_port[0],   src_sw_in_port[1],
dst_sw_in_port[0], dst_sw_in_port[1],
                                              src_ip,  dst_ip,  paths_nodes,  next_index,
tcp_pkt, udp_pkt)
        actions = [parser.OFPActionOutput(out_port)]
        data = None
        if msg.buffer_id == ofproto.OFP_NO_BUFFER:
            data = msg.data
        out = parser.OFPPacketOut(
            datapath=datapath, buffer_id=msg.buffer_id, in_port=in_port,
            actions=actions, data=data)
        datapath.send_msg(out)
```

- Установка потоков для TCP и UDP: Устанавливает потоки для маршрутизации TCP/UDP пакетов. Для каждого узла маршрута добавляются правила с указанием входного и выходного портов, а также IP-адресов и портов пакета.

```
# Установка потоков по IP для TCP и UDP
def  set_paths_tcp_udp(self,  src,  in_port_src_sw,  dst,  out_port_dst_sw,  ip_src,
ip_dst, paths_nodes, index, tcp_pkt, udp_pkt):
    paths_with_ports      =      self.paths_ports(paths_nodes,      in_port_src_sw,
out_port_dst_sw)
    selected_path = paths_with_ports[index]
    for node in selected_path:
        dp = self.topology_monitor.datapaths[node]
        ofp = dp.ofproto
        ofp_parser = dp.ofproto_parser
        actions = []
        in_port = selected_path[node][0]
        out_port = selected_path[node][1]
        match_ip = None
        if tcp_pkt:
            match_ip = ofp_parser.OFPMatch(
                    in_port=in_port,
```

```
                    ip_proto=6,
                    eth_type=0x0800,
                    ipv4_src=ip_src,
                    ipv4_dst=ip_dst,
                    tcp_src=tcp_pkt.src_port,
                    tcp_dst=tcp_pkt.dst_port
                )
        elif udp_pkt:
            match_ip = ofp_parser.OFPMatch(
                    in_port=in_port,
                    ip_proto=17,
                    eth_type=0x0800,
                    ipv4_src=ip_src,
                    ipv4_dst=ip_dst,
                    udp_src=udp_pkt.src_port,
                    udp_dst=udp_pkt.dst_port
                )
        actions = [ofp_parser.OFPActionOutput(out_port)]
        self.install_flow(dp, 32768, match_ip, actions)
    return selected_path[src][1]
```

- Удаление потоков для TCP и UDP: Эта функция удаляет потоки TCP или UDP, которые больше не должны использоваться. Для каждого коммутатора на маршруте удаляются правила, соответствующие IP-адресам и портам.

```
# Удаление потоков по IP для TCP и UDP
def delete_paths_tcp_udp(self, src, in_port_src_sw, dst, out_port_dst_sw, ip_src, ip_dst, paths_nodes, index, tcp_pkt, udp_pkt):
    paths_with_ports = self.paths_ports(paths_nodes, in_port_src_sw, out_port_dst_sw)
    selected_path = paths_with_ports[index]
    for node in selected_path:
        dp = self.topology_monitor.datapaths[node]
        ofp_parser = dp.ofproto_parser
        in_port = selected_path[node][0]
        out_port = selected_path[node][1]
        match_ip = None
        if tcp_pkt:
            match_ip = ofp_parser.OFPMatch(
                    in_port=in_port,
```

```
                    ip_proto=6,
                    eth_type=0x0800,
                    ipv4_src=ip_src,
                    ipv4_dst=ip_dst,
                    tcp_src=tcp_pkt.src_port,
                    tcp_dst=tcp_pkt.dst_port
                )
        elif udp_pkt:
            match_ip = ofp_parser.OFPMatch(
                    in_port=in_port,
                    ip_proto=17,
                    eth_type=0x0800,
                    ipv4_src=ip_src,
                    ipv4_dst=ip_dst,
                    udp_src=udp_pkt.src_port,
                    udp_dst=udp_pkt.dst_port
                )
        self.delete_flow(dp, match_ip, out_port)
    return
```

- Фоновая задача маршрутизации: Периодически анализирует состояние сети, идентифицирует перегруженные каналы и вызывает функцию переназначения маршрутов для перераспределения нагрузки.

```
# Фоновая задача для обновления маршрутов
def adaptive_routing(self):
    while True:
        metric = copy.deepcopy(self.port_monitor.get_link_costs())
        self.rerouting(metric)
        hub.sleep(REROUTING_PERIOD)
```

- Переназначение маршрутов при перегрузке: Определяет перегруженные каналы и перенаправляет потоки через альтернативные маршруты, используя алгоритмы поиска кратчайших путей. Перегруженные потоки переустанавливаются на новые маршруты.

```
# Функция переназначения маршрутов при перегрузке
def rerouting(self, metric):
    overloaded_links = self.get_overloaded_links(metric)
    if len(overloaded_links)!=0:
```

```
        flows_overloaded = self.get_flows_overloaded(overloaded_links)
        new_metric = self.get_costs_to_adapt(flows_overloaded, metric)
        tmp_paths_dict = copy.deepcopy(self.paths_cache)
        new_WRR = {}
        for key, path_info in tmp_paths_dict.items():
            old_paths, old_paths_edges, old_pw, src_ip, dst_ip, src_dst, streams =
copy.deepcopy(path_info)
            streams_overloaded = []
            for path_index, tcp_pkt, udp_pkt in streams:
                flow = None
                if tcp_pkt:
                    flow = (6, src_ip, dst_ip, tcp_pkt.src_port, None)
                if udp_pkt:
                    flow = (17, src_ip, dst_ip, None, udp_pkt.src_port)
                if flow in flows_overloaded:
                    streams_overloaded.append((path_index, tcp_pkt, udp_pkt))

            if streams_overloaded:
                src = key[0]
                in_port_src_sw = key[1]
                dst = key[2]
                out_port_dst_sw = key[3]

                # alg = SI(new_metric, src, dst, K, 10, 100, 'ABC')
                # new_paths, new_paths_edges, new_pw = alg.run()

                alg = Yen(new_metric, src, dst, K)
                # alg = RNN(self.model, new_metric, src, dst, K)
                new_paths, new_paths_edges, new_pw = alg.compute_shortest_paths()

                normalized_pw = self.normalize(new_pw)
                new_weights = [int(round(itm*10)) for itm in normalized_pw]
                new_WRR[key] = [new_weights, 0]

                pos_path = len(self.paths_cache[key][0])
                for path_index, tcp_pkt, udp_pkt in streams_overloaded:
                    new_WRR[key][1] += 1
                    next_index                                                    =
self.weighted_periodic_distribution(new_WRR[key][0], new_WRR[key][1])
```

```
                    self.delete_paths_tcp_udp(
                        src, in_port_src_sw, dst, out_port_dst_sw, src_ip, dst_ip,
                        old_paths, path_index, tcp_pkt, udp_pkt
                    )

                    self.set_paths_tcp_udp(
                        src, in_port_src_sw, dst, out_port_dst_sw, src_ip, dst_ip,
                        new_paths, next_index, tcp_pkt, udp_pkt
                    )

                    streams.remove((path_index, tcp_pkt, udp_pkt))
                    streams.add((pos_path+next_index, tcp_pkt, udp_pkt))

                self.paths_cache[key][0].extend(new_paths)
                self.paths_cache[key][1].extend(new_paths_edges)
                self.paths_cache[key][2].extend(new_pw)
                self.paths_cache[key][-1] = streams
```

- Получение списка перегруженных каналов: Возвращает каналы, чья текущая стоимость превышает заданное пороговое значение (например, 10). Это индикатор перегрузки.

```
# Получение списка перегруженных каналов (где стоимость канала > 10)
def get_overloaded_links(self, metric):
    overloaded_links = []
    for src in metric:
        for dst in metric[src]:
            link_cost = metric[src][dst]
            if link_cost is not None and link_cost > Decimal('10'):
                overloaded_links.append((src, dst))
    return overloaded_links
```

- Определение перегруженных потоков: Анализирует трафик на перегруженных каналах, определяя минимальный набор потоков, которые необходимо переназначить для устранения перегрузки.

```
# Поиск минимального множества потоков, удаление которых решает проблему перегрузки
на всех перегруженных каналах
def get_flows_overloaded(self, overloaded_links):
```

```
    capacity = MAX_CAPACITY
    flow_speeds = {}
    for src, dst in overloaded_links:
        for                    flow,                    speed                    in
self.flow_monitor.switch_to_switch_flows_speed[src][dst].items():
            if flow not in flow_speeds:
                flow_speeds[flow] = 0
            flow_speeds[flow] += speed
    sorted_flows = sorted(flow_speeds.items(), key=lambda x: x[1], reverse=True)
    selected_flows = set()
    for flow, speed in sorted_flows:
        selected_flows.add(flow)
        if self.is_valid_solution(selected_flows, overloaded_links, capacity):
            break
    return selected_flows

# Проверка, что удаление указанного множества потоков решает проблему перегрузки
def is_valid_solution(self, flow_subset, overloaded_links, capacity):
    for src, dst in overloaded_links:
        remaining_speed = sum(
                            speed         for         flow,         speed         in
self.flow_monitor.switch_to_switch_flows_speed[src][dst].items()
                            if flow not in flow_subset
                            )
        link_utilization = round(remaining_speed/capacity, 1)
        link_cost = None
        if link_utilization >= 1:
            link_cost = Decimal('100')
        else:
            link_cost= Decimal(str(round(1 / (1 - link_utilization), 1)))
        if link_cost > Decimal('10'):
            return False
    return True
```

- Перерасчет стоимости каналов после удаления перегруженных потоков: Изменяет стоимость каналов с учетом пропускной способности и удаляемых потоков. Помогает адаптировать маршруты к текущим условиям сети.

```
# Вычисление стоимости каналов после удаления перегруженных потоков
```

```
def get_costs_to_adapt(self, flows_overloaded, metric):
    capacity = MAX_CAPACITY
    new_metric = copy.deepcopy(metric)
    for src in self.flow_monitor.switch_to_switch_flows_speed:
        for dst in self.flow_monitor.switch_to_switch_flows_speed[src]:
            flows                                                                    =
set(self.flow_monitor.switch_to_switch_flows_speed[src][dst].keys())
            remain_flows = flows - flows_overloaded
            if len(flows & flows_overloaded)!=0:
                remaining_speed                                                      =
sum(self.flow_monitor.switch_to_switch_flows_speed[src][dst][flow_key] for flow_key
in remain_flows)
                link_utilization = round(remaining_speed/capacity, 1)
                if link_utilization >= 1:
                    new_metric[src][dst] = Decimal('100')
                else:
                    new_metric[src][dst]    =    Decimal(str(round(1    /    (1    -
link_utilization), 1)))
    return new_metric
```

- Вспомогательные функции.

```
# Добавление портов к путям
def paths_ports(self, paths_nodes, in_port_src_sw, out_port_dst_sw):
    paths_p = []
    for path in paths_nodes:
        p = {}
        in_port = in_port_src_sw
        for s1, s2 in zip(path[:-1], path[1:]):
            out_port = self.topology_monitor.graph[s1][s2]['src_port']
            p[s1] = (in_port, out_port)
            in_port = self.topology_monitor.graph[s1][s2]['dst_port']
        p[path[-1]] = (in_port, out_port_dst_sw)
        paths_p.append(p)
    return paths_p

# Удаление потока
def delete_flow(self, datapath, match, out_port):
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser
```

```
    mod = parser.OFPFlowMod(datapath=datapath, command=ofproto.OFPFC_DELETE,
                            out_port=out_port, match=match)
    datapath.send_msg(mod)

# Добавление потока в коммутатор
def install_flow(self, datapath, priority, match, actions):
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser
    inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
                                         actions)]
    mod = parser.OFPFlowMod(datapath=datapath, priority=priority,
                            match=match, instructions=inst)
    datapath.send_msg(mod)

# Нормализация стоимости путей
def normalize(self, paths_weights):
    paths_weights = [MAX_VALUE if itm_pw == 0 else 1/itm_pw for itm_pw in
paths_weights]
    total = sum(paths_weights)
    weights_after_normalizing = [round(float(itm_pw)/total, 2) for itm_pw in
paths_weights]
    weights_after_normalizing[-1] += 1 - sum(weights_after_normalizing)
    weights_after_normalizing[-1] = round(weights_after_normalizing[-1], 2)
    return weights_after_normalizing

# Метод взвешенного циклического распределения
def weighted_periodic_distribution(self, weights_rr, n_index):
    weights = copy.deepcopy(weights_rr)
    if not weights or n_index <= 0:
        return None
    total_weights = sum(weights)
    n_index = n_index % total_weights or total_weights
    current_pick = 0
    for _ in range(total_weights):
        for i, weight in enumerate(weights):
            if weight > 0:
                current_pick += 1
                weights[i] -= 1
                if current_pick == n_index:
```

```
                    return i
    return None

# Получение оптимальных путей для пакетов
def calculate_best_paths(self, src, dst):
    metric = copy.deepcopy(self.port_monitor.get_link_costs())
    alg = Yen(metric, src, dst, K)
    paths_nodes, paths_edges, paths_weights = alg.compute_shortest_paths()
    return paths_nodes, paths_edges, paths_weights
```

4.5 ГЕНЕРАТОР ТРАФИКА

iPerf3 – это широко используемый инструмент для тестирования производительности сети, который предоставляет точные и детализированные результаты о пропускной способности соединений. Он позволяет измерить сетевую пропускную способность между двумя узлами, используя архитектуру «клиент-сервер». iPerf3 поддерживает различные сетевые протоколы, включая TCP и UDP. Благодаря открытому исходному коду и гибкости в настройках, iPerf3 позволяет проводить тестирование сетевой производительности в разнообразных условиях и сценариях. Простота использования и возможность получения подробной статистики делают его незаменимым инструментом для сетевых администраторов, инженеров и разработчиков при решении задач диагностики, оценки производительности и оптимизации сети.

Основные возможности и параметры *iPerf3*:

- Режим работы: *iPerf3* может работать как в режиме сервера (-*s*), так и клиента (-*c* <*имя_хоста*>). В режиме сервера инструмент ожидает входящих подключений, а в режиме клиента инициирует тестирование соединения с указанным сервером. Клиент может передавать данные на сервер, который ведет подсчет и возврат результатов теста.

- Продолжительность теста (*-t*): Определяет продолжительность тестирования в секундах. По умолчанию тест длится 10 секунд, однако можно установить любое значение для более детального анализа производительности в течение более длительных периодов времени.
- Пропускная способность (*-b*): Этот параметр используется для тестирования на основе UDP и определяет целевую пропускную способность в битах в секунду (например, 1G для 1 Гбит/с). В случае TCP-тестов *iPerf3* автоматически использует максимальную возможную пропускную способность, но также можно ограничить ее этим параметром при необходимости.
- Параллельные соединения (*-P*): Позволяет задать количество одновременно открытых соединений для тестирования. Увеличение числа параллельных соединений позволяет смоделировать условия высокой нагрузки на сеть и оценить, как она справляется с несколькими одновременными потоками данных.
- Интервал отчетности (*-i*): Определяет интервал в секундах между периодическими отчетами о текущей пропускной способности во время тестирования. По умолчанию, отчет публикуется каждые 1 секунду.
- Номер порта (*-p*): Указывает порт, который будет использоваться сервером *iPerf3* для прослушивания соединений. По умолчанию это 5201, но можно задать любой другой доступный порт при необходимомти.
- Режим UDP (*-u*): Переключает *iPerf3* на использование UDP вместо TCP. Тестирование на основе UDP полезно для измерения таких параметров, как потери пакетов и джиттер, что делает его важным для оценки качества мультимедийных и реального времени сетевых приложений, таких как видеоконференции или VoIP.
- Вывод в формате JSON (*-J*): Генерирует результаты тестирования в формате JSON, что значительно упрощает их последующий анализ с помощью программных инструментов или интеграцию в автоматизированные системы мониторинга и отчетности.

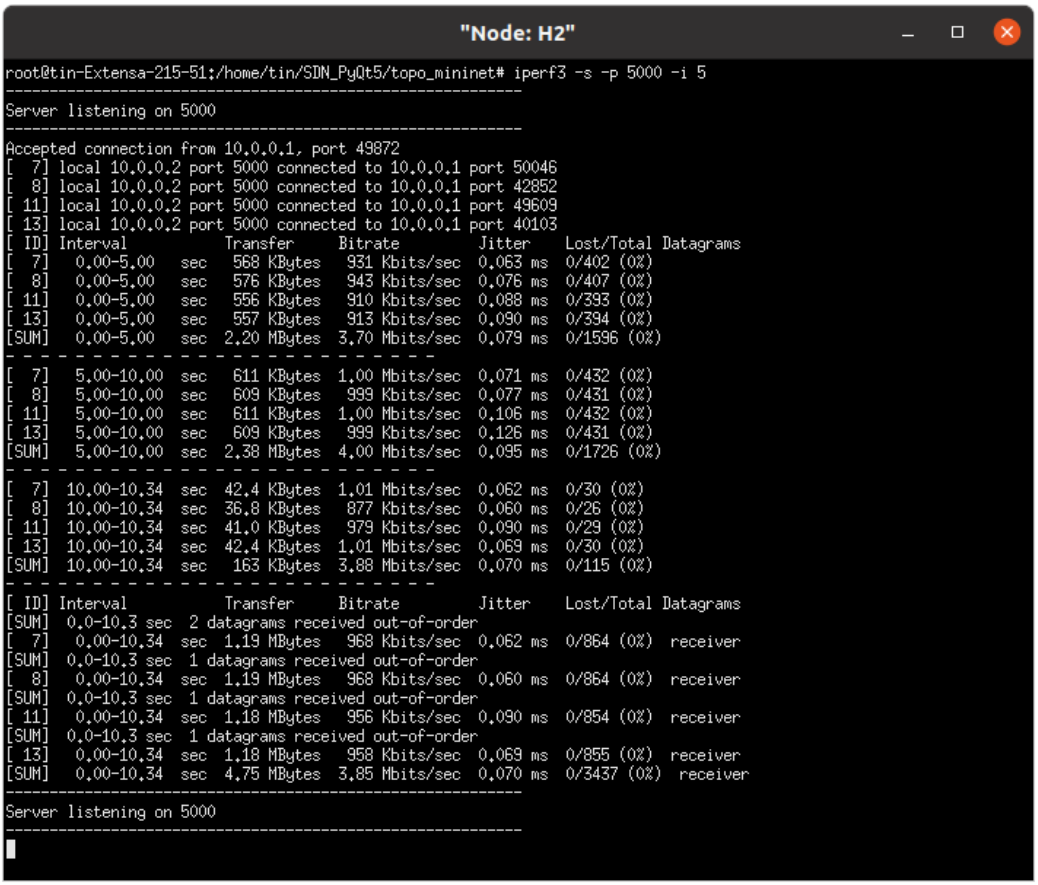


Рисунок 4.16 – Режим сервера

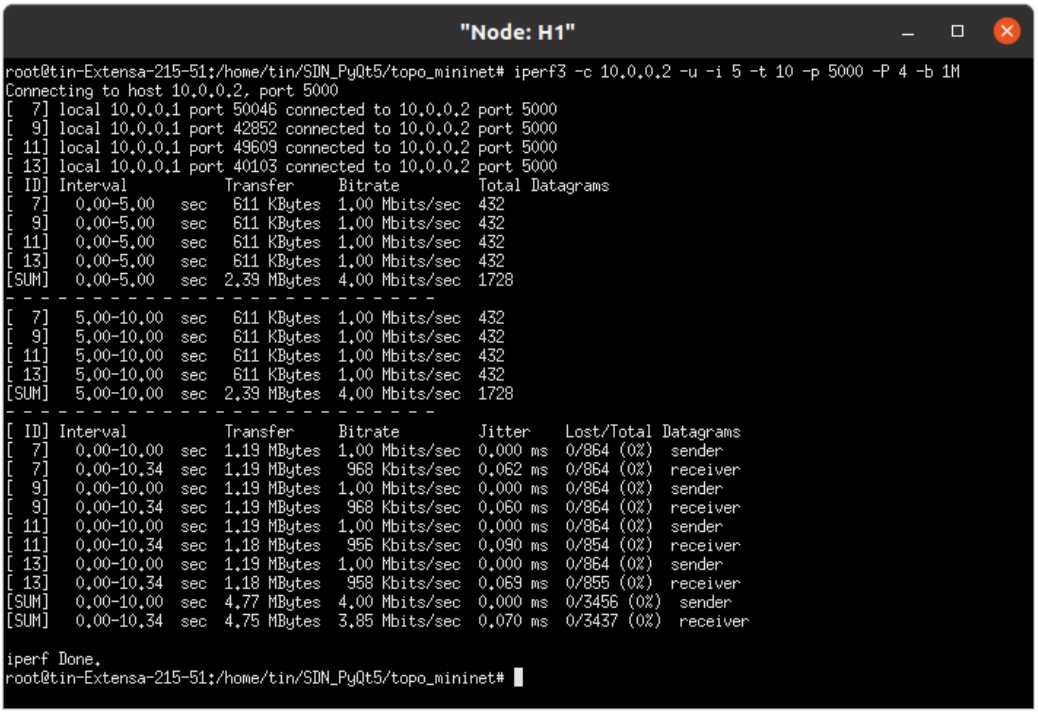


Рисунок 4.17 – Режим клиента

ОСНОВНЫЕ РЕЗУЛЬТАТЫ И ВЫВОДЫ

1. Разработана структура визуальной программной системы SDNLoadBalancer для организации распределенной обработки потоков данных в ПКС. Система реализована на платформе PyQt5 и включает в себя графический редактор, эмулятор ПКС, комплексную систему мониторинга и интеллектуального управления, а также генератор трафика.

2. Разработана архитектура библиотеки программных компонентов интеллектуальной маршрутизации и балансировки потоков данных в ПКС, обеспечивающая эффективное управление потоками данных в сети на основе нейронных сетей и роевых алгоритмов.

3. Изучены возможности графического редактора SDNLoadBalancer, который обеспечивает интуитивно понятный интерфейс для проектирования топологии сети.

4. Анализированы функциональные возможности эмулятора Mininet, который позволяет создавать виртуальные коммутаторы, маршрутизаторы и хосты для моделирования топологии ПКС, помогая тестировать и экспериментировать с сетевыми структурами без необходимости использовать реальное оборудование.

5. Реализована комплексная система мониторинга и интеллектуального управления ПКС, включающая компоненты мониторинга топологии, портов, потоков, задержки, многопутевой маршрутизации и балансировки потоков данных, что позволяет осуществлять интеллектуальное управление сетью, оптимизировать распределение трафика и гарантировать высокую производительность сети за счет постоянного мониторинга состояния сети и динамического обновления таблиц потоков на коммутаторах.

6. Исследован генератор трафиков iPerf3, который используется для тестирования производительности сети. Инструмент позволяет измерять пропускную способность, задержку и потери пакетов в различных сценариях, что полезно для диагностики проблем и оценки производительности сети.

ГЛАВА 5 ИССЛЕДОВАНИЕ ПРОЦЕССОВ ИНТЕЛЛЕКТУАЛЬНОЙ МАРШРУТИЗАЦИИ И БАЛАНСИРОВКИ ПОТОКОВ ДАННЫХ В ПКС

5.1 ИССЛЕДОВАНИЕ АЛГОРИТМОВ ИНТЕЛЛЕКТУАЛЬНОЙ МАРШРУТИЗАЦИИ НА ОСНОВЕ РОЕВОГО ИНТЕЛЛЕКТА

Для исследования процессов интеллектуальной многопутевой маршрутизации в ПКС на основе роевого интеллекта рассматривалась топология ПКС в соответствии с рисунком 5.1.

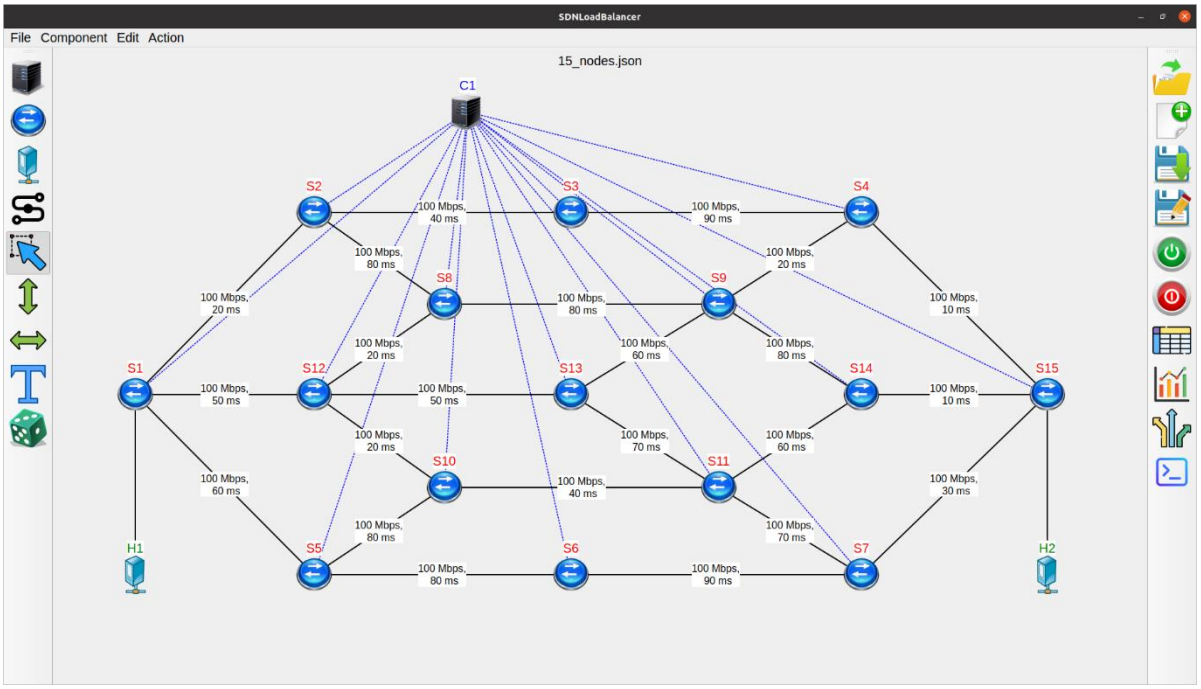


Рисунок 5.1 – Экспериментальная топология ПКС из 15 узлов

Задача состояла в том, чтобы найти четыре кратчайших пути между хостами H1 и H2. Метрики каналов могли принимать различные значения в зависимости от изменений состояния сети. Предположим, что в некоторый момент времени t выполнен снимок состояния сети на основе информации, полученной от компонентов мониторинга. Метрики каналов представлены на рисунке 5.2, а четыре найденных кратчайших пути обозначены разными цветами. Алгоритмы роевого интеллекта выполнялись на основе этих метрик каналов на данный момент времени t.

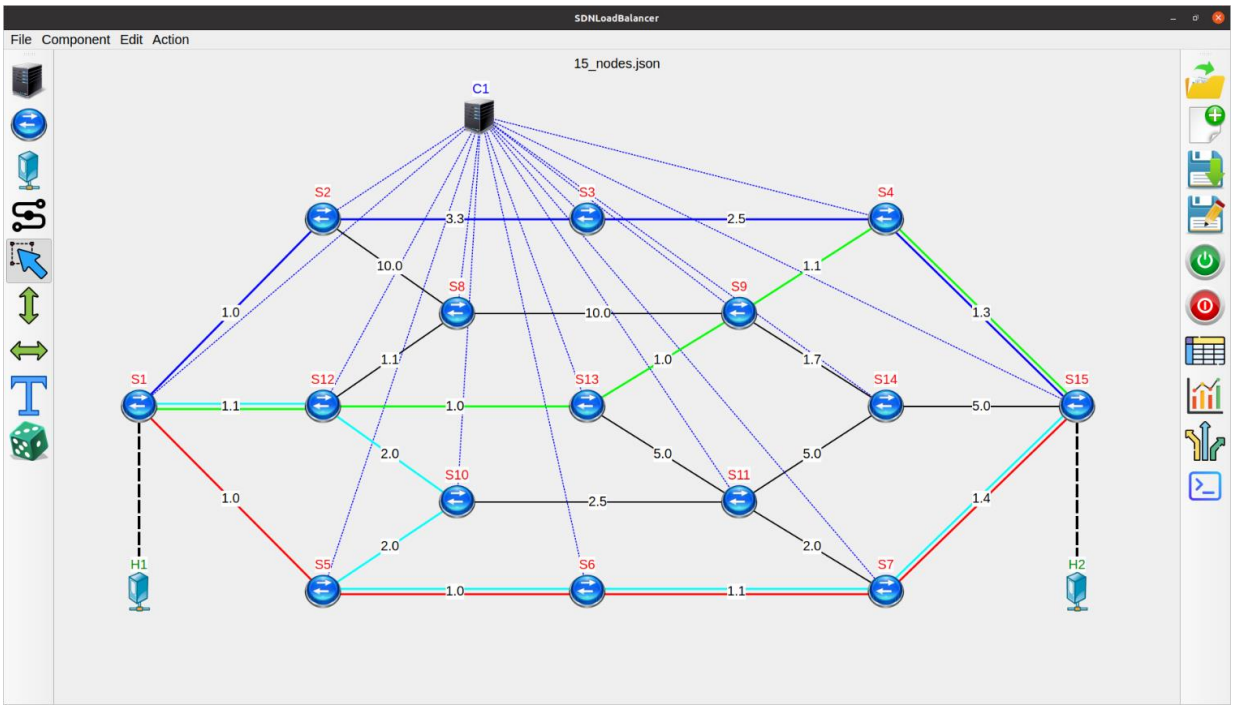


Рисунок 5.2 – Отображение полученных путей ($K = 4$)

Для оценки эффективности алгоритмов роевого интеллекта при многопутевой маршрутизации полученные результаты были сопоставлены с результатами работы алгоритма Йена. Каждый алгоритм был запущен 100 раз для точной оценки его эффективности. Результаты сравнивались по таким показателям, как количество успешных поисков для каждого пути, общая стоимость всех путей, время выполнения алгоритма. В таблице 5.1 представлены кратчайшие пути, найденные алгоритмом Йена.

Таблица 5.1 – Результат работы алгоритма Йена

№	Кратчайшие пути	Стоимость (вес)
1	S1, S5, S6, S7, S15	4.5
2	S1, S12, S13, S9, S4, S15	5.5
3	S1, S2, S3, S4, S15	8.1
4	S1, S12, S10, S5, S6, S7, S15	8.6
		Общая стоимость (вес) всех путей = 26.7

5.1.1 ИССЛЕДОВАНИЕ ГЕНЕТИЧЕСКОГО АЛГОРИТМА В ПКС

Для рассматриваемой экспериментальной топологии параметры генетического алгоритма задавались следующим образом: $N = 10$, $Max = 100, P_c = 0.7, P_m = [0.1, 0.7]$. Полученные результаты показаны на рисунках 5.3, 5.4 и 5.5.

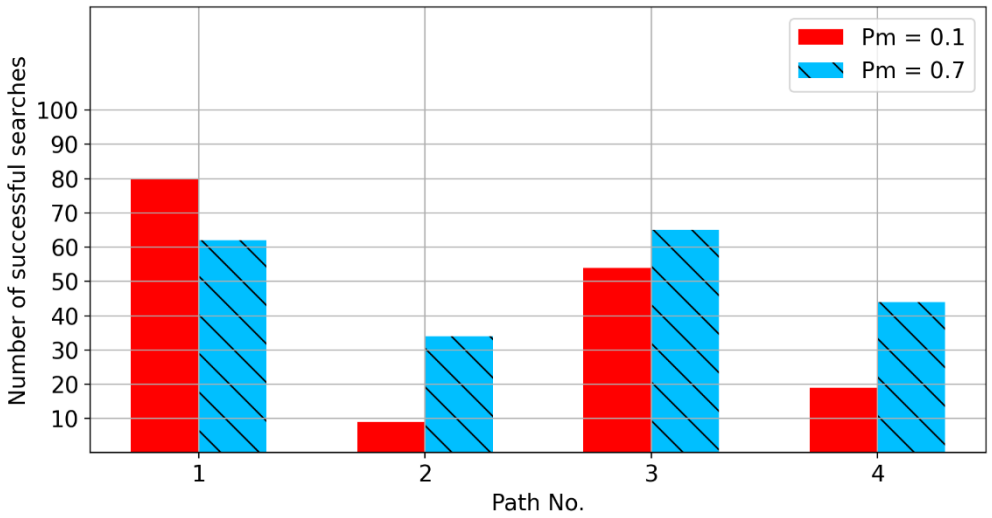


Рисунок 5.3 – Количество успешных поисков для каждого пути ($P_m = [0.1, 0.7]$)

Для первого пути более низкая вероятность мутации $P_m = 0.1$ способствует более успешному поиску по сравнению с высокой вероятностью мутации $P_m = 0.7$. Это показывает, что при низкой вероятности мутации GA быстрее сходится к лучшему решению, так как процесс менее подвержен отклонениям и больше ориентирован на локальное исследование пространства решений.

Для путей 2, 3 и 4 высокая вероятность мутации оказывается эффективнее низкой вероятности мутации. Это указывает на то, что увеличение генетического разнообразия при более высокой вероятности мутации улучшает исследование пространства решений и способствует нахождению альтернативных путей. Это особенно важно при решении задачи поиска нескольких кратчайших путей, так как цель заключается не только в нахождении самого оптимального, но и других близких к нему решений.

Высокая вероятность мутации вводит значительное генетическое разнообразие в популяцию, что предотвращает преждевременную сходимость к локальным

оптимумам. Это особенно полезно для задачи поиска k-кратчайших путей, где требуется найти несколько почти оптимальных решений. Большое количество успешных находок для путей 2, 3 и 4 свидетельствует о том, что увеличение вероятности мутации улучшает исследовательские способности GA, помогая избежать локальных минимумов и обнаружить несколько жизнеспособных маршрутов.

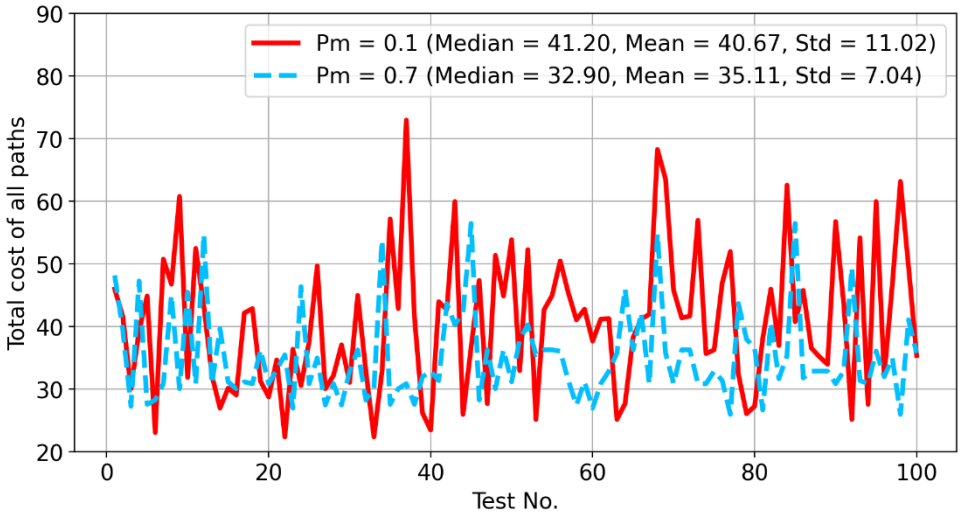


Рисунок 5.4 – Общая стоимость всех путей для каждого запуска ($P_m = [0.1, 0.7]$)

График общей стоимости всех путей, построенный на основе 100 запусков, наглядно показывает, что более высокая вероятность мутации ($P_m = 0.7$, синяя пунктирная линия) в среднем приводит к меньшей общей стоимости, чем при низкой вероятности мутации ($P_m = 0.1$, красная сплошная линия).

Обе вероятности мутации демонстрируют значительные колебания в течение всех 100 запусков, что подтверждается значительными стандартными отклонениями. Это характерно для стохастических алгоритмов, таких как генетические алгоритмы, которые зависят от случайных процессов, таких как мутация и скрещивание.

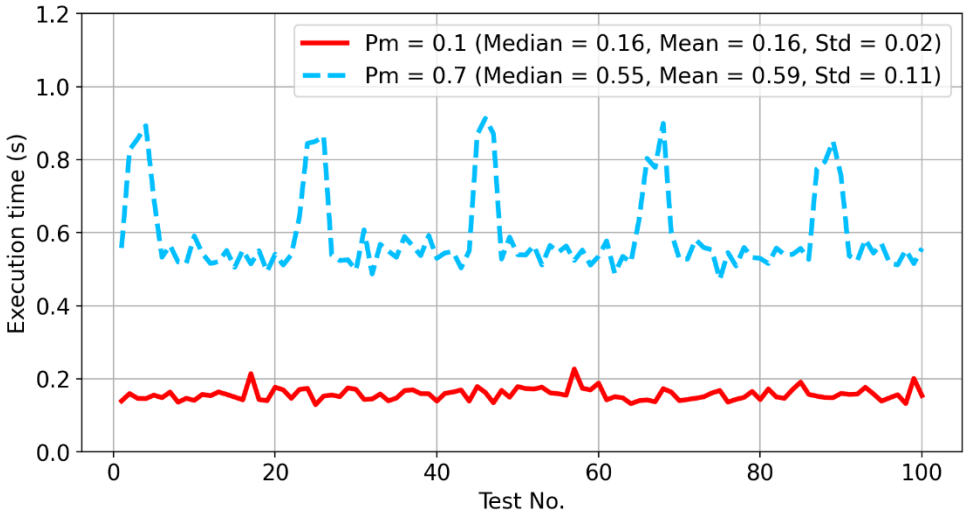


Рисунок 5.5 – Время выполнения алгоритма для каждого запуска ($P_m = [0.1, 0.7]$)

Более высокая вероятность мутации $P_m = 0.7$ приводит к увеличению времени выполнения всех запусков по сравнению с более низкой вероятностью мутации $P_m = 0.1$.

Стандартное отклонение времени выполнения в 0.02 секунды при $P_m = 0.1$ свидетельствует о том, что время выполнения алгоритма стабильно и мало изменяется от запуска к запуску. В то же время стандартное отклонение в 0.11 секунды при $P_m = 0.7$ указывает на более высокую вариативность времени выполнения, что связано с увеличением количества генетических изменений и разнообразием исследуемых решений.

Заметные пики времени работы при $P_m = 0.7$ указывают на ситуации, когда алгоритм мог столкнуться с особенно сложными задачами, требующими больше времени на обработку, возможно, из-за формирования более разнообразных генетических комбинаций, которые требовали дополнительной оценки. Относительно гладкая и низкая линия при $P_m = 0.1$ предполагает меньшее количество интенсивных вычислений, вероятно, из-за реже происходящих генетических изменений, требующих оценки.

Наблюдения за различными показателями, такими как количество успешных поисков, общая стоимость путей и время выполнения, демонстрируют, что более

высокая вероятность мутации ($P_m = 0.7$) обычно способствует успешному поиску путей с меньшей общей стоимостью, но это происходит за счет значительного увеличения времени работы. Это свидетельствует о том, что дополнительная случайность, возникающая из-за повышенной вероятности мутации, улучшает способность генетических алгоритмов избегать локальных оптимумов и тщательнее исследовать пространство решений, что полезно для выявления эффективных и разнообразных решений при многопутевой маршрутизации в ПКС. Однако это требует значительных вычислительных ресурсов, как показывает увеличенное время выполнения. Напротив, более низкая вероятность мутации ($P_m = 0.1$) обеспечивает более быструю обработку, но менее успешные результаты по общей стоимости всех путей и меньшее количество успешных поисков. Эти наблюдения подчеркивают критический баланс между исследовательскими возможностями и вычислительной эффективностью при оптимизации генетических алгоритмов для решения сложных задач многопутевой маршрутизации.

5.1.2 ИССЛЕДОВАНИЕ АЛГОРИТМОВ ОПТИМИЗАЦИИ МУРАВЬИНОЙ КОЛОНИИ В ПКС

Для исследования процессов многопутевой маршрутизации в рассматриваемой экспериментальной топологии ПКС, параметры оптимизации муравьиной колонии устанавливаются аналогично параметрам из статьи Марко Дориго [67]: $N = 10, Max = 100, \rho = 0.1, \alpha = 1, \beta = 2, q_0 = 0.9$ и $Q = 1$. Дополнительно, для алгоритма системы муравьиной колонии рассмотрим еще один случай с коэффициентом жадности $q_0 = 0.5$.

На следующих рисунках представлены полученные результаты.

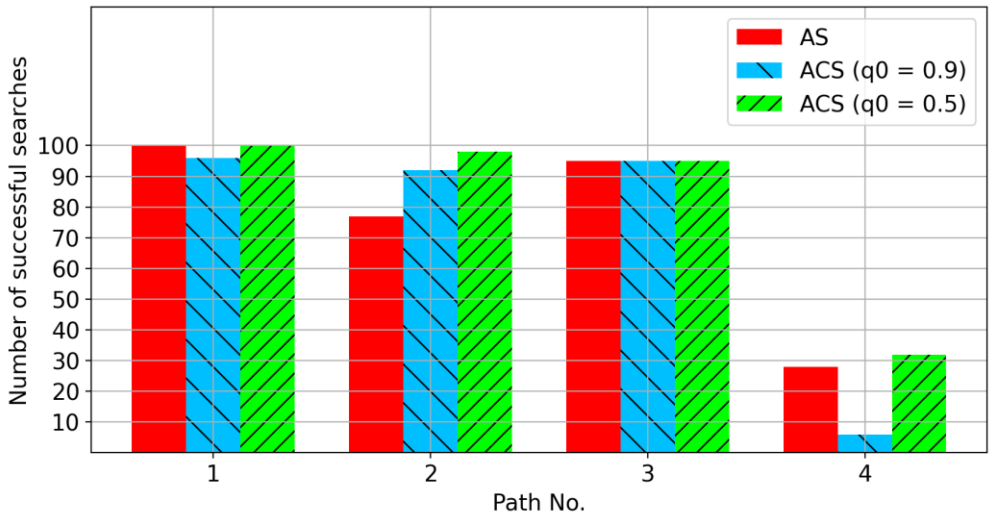


Рисунок 5.6 – Количество успешных поисков для каждого пути

ACS с $q_0 = 0.5$ действительно достигает наибольшего количества успешных поисков по всем путям, превосходя как AS, так и ACS с $q_0 = 0.9$. Это свидетельствует о том, что эта конфигурация лучше всего справляется с задачей поиска нескольких кратчайших путей, эффективно сочетая исследование и эксплуатацию.

AS показала хорошие результаты для пути 1, но была менее эффективной для путей 2 и 3 по сравнению с обеими конфигурациями ACS. Для пути 4 AS дает несколько худшие результаты, чем ACS с $q_0 = 0.5$, но значительно лучше, чем ACS с $q_0 = 0.9$.

Более низкое значение q_0 в ACS увеличивает степень исследования, уменьшая склонность к ранней эксплуатации известных путей. Это особенно важно для задачи поиска нескольких кратчайших путей, где необходимо не только найти лучший путь, но и несколько качественных альтернативных маршрутов. Более глубокое исследование помогает избежать преждевременной сходимости к неоптимальным решениям и способствует поиску разнообразных путей.

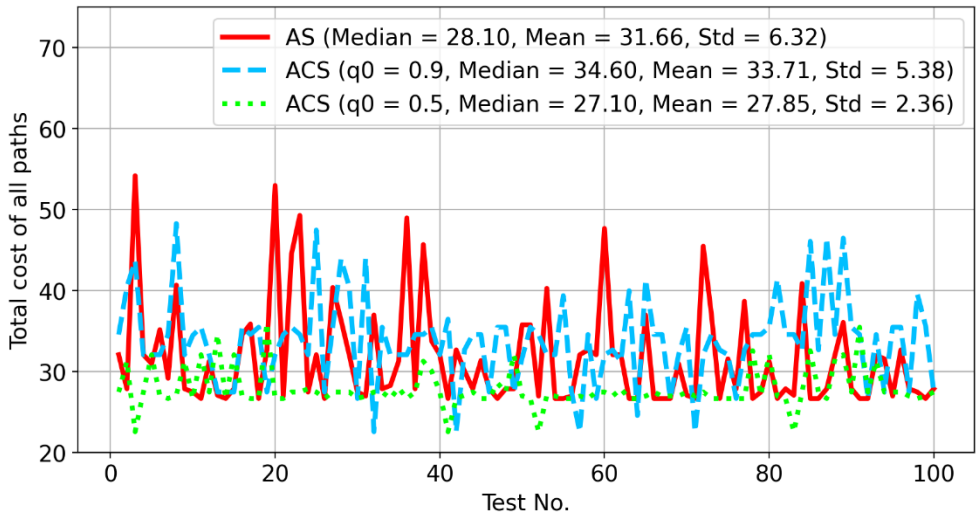


Рисунок 5.7 – Общая стоимость всех путей для каждого запуска

AS, представленная красной линией на графике, демонстрирует высокую изменчивость общей стоимости путей: от минимальных значений около 25 до пиковых значений около 55. Широкий диапазон значений можно объяснить тем, что AS использует более равномерный механизм обновления феромонов, который не делает существенных различий между путями разного качества. Это может приводить к тому, что в разных запусках алгоритм выбирает как оптимальные, так и неоптимальные пути, что объясняет высокую изменчивость и стандартное отклонение 6.32.

ACS с $q_0 = 0.9$, обозначенная пунктирной синей линией, показывает в среднем более высокую общую стоимость путей по сравнению с ACS с $q_0 = 0.5$, а также периодические пики на графике. Это говорит о том, что с более высоким значением q_0 (больше склонность к эксплуатации) алгоритм чаще встречается с неоптимальными путями. Стандартное отклонение 5.38 также указывает на высокую изменчивость результатов, что подтверждает меньшую стабильность при таком параметре. ACS с $q_0 = 0.5$, обозначенная пунктирной зеленой линией, демонстрирует самую низкую среднюю общую стоимость и наиболее стабильно поддерживает более низкую общую стоимость по сравнению с двумя другими настройками. Линия графика более плавная и в основном находится в нижней части шкалы стоимости.

Стандартное отклонение всего 2.36 подчеркивает высокую стабильность и предсказуемость этого параметра, делая его наиболее надежным выбором.

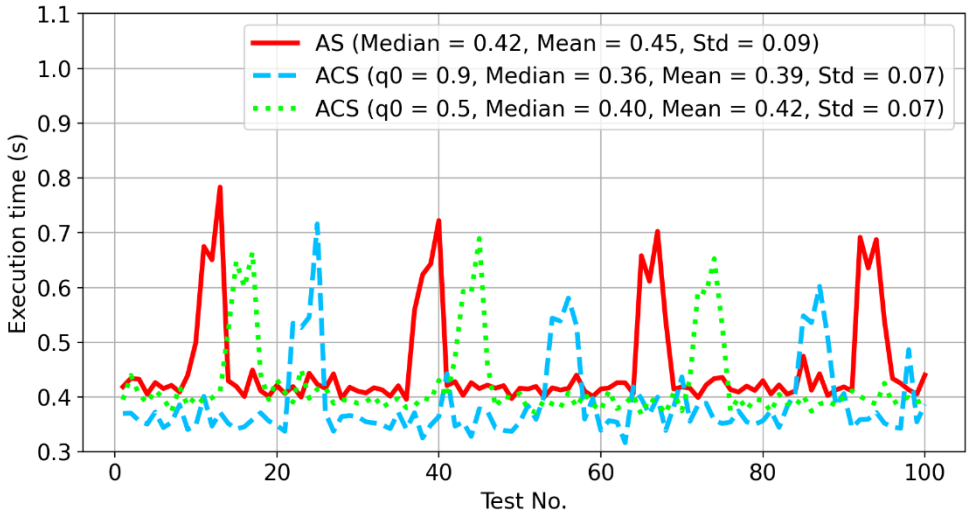


Рисунок 5.8 – Время выполнения алгоритма для каждого запуска

ACS с $q_0 = 0.9$ демонстрирует самое низкое время выполнения среди трех конфигураций (среднее время 0.39 секунды) и стандартное отклонение 0.07. Это свидетельствует о более быстрой сходимости, вероятно, благодаря высокой тенденции к эксплуатации известных решений. ACS с $q_0 = 0.5$ показывает немного большее время работы, чем $q_0 = 0.9$, со средним временем 0.42 секунды и таким же стандартным отклонением 0.07, что предполагает баланс между разведкой и эксплуатацией. У AS самое высокое среднее время работы – примерно 0.45 секунды, со стандартным отклонением 0.09. Время работы более изменчиво по сравнению с обеими конфигурациями ACS, что указывает на менее эффективный процесс поиска с точки зрения времени.

ACS с более низким q_0 демонстрирует превосходную производительность при поиске кратчайших путей по сравнению с ACS с более высоким q_0 и AS. Он эффективно балансирует разведку и эксплуатацию, предоставляя более оптимальные решения с минимальным влиянием на время вычислений. Таким образом, ACS с более низким q_0 является наиболее эффективным подходом для многопутевой маршрутизации в средах ПКС.

5.1.3 ИССЛЕДОВАНИЕ АЛГОРИТМА СТАИ ПТИЦ В ПКС

Для решения задачи многопутевой маршрутизации важно найти баланс между исследованием и эксплуатацией при поиске почти оптимальных решений. В алгоритме стаи птиц коэффициент инерции w играет ключевую роль в поддержании этого баланса. Следовательно, для анализа процесса многопутевой маршрутизации в ПКС, параметры данного алгоритма с двумя различными значениями w устанавливаются следующим образом: $N = 10, Max = 100, w = [0.1, 0.7], c_1 = 2, c_2 = 2$.

- При $w = 0.1$ алгоритм демонстрирует более быструю сходимость к известным хорошим решениям, акцентируя внимание на эксплуатации уже найденных решений.
- При $w = 0.7$ возрастает степень исследования пространства решений, что помогает предотвратить преждевременную сходимость к локальным минимумам и способствует нахождению альтернативных решений.

Результаты разработанного алгоритма показаны на следующих рисунках.

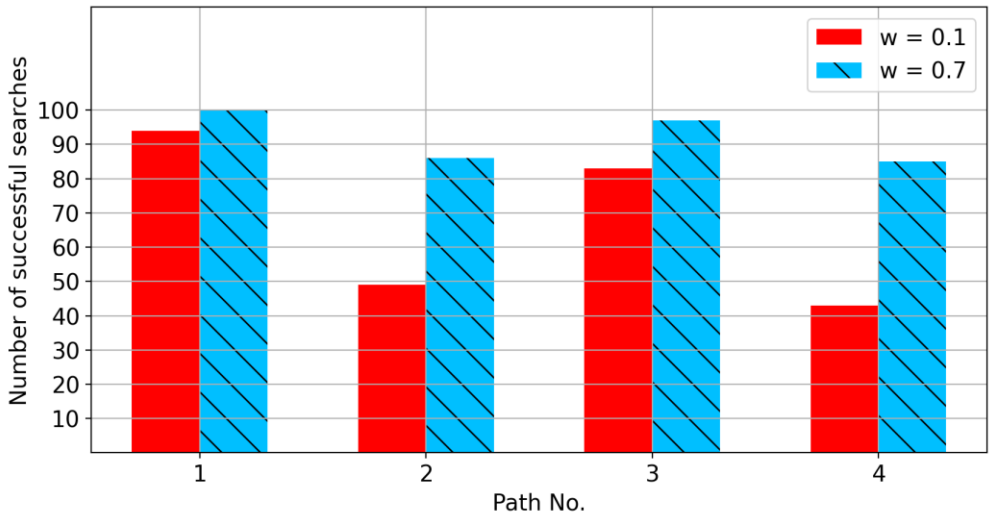


Рисунок 5.9 – Количество успешных поисков для каждого пути

Для всех путей (1, 2, 3 и 4) синие столбцы, соответствующие более высокому весу инерции ($w = 0.7$), превосходят красные столбцы ($w = 0.1$). Это указывает на то, что

увеличение веса инерции положительно влияет на способность алгоритма находить жизнеспособные пути.

Улучшение особенно заметно на сложных путях, например, на путях 2 и 4, где разница между производительностью $w = 0.1$ и $w = 0.7$ наиболее выражена. Это говорит о том, что более высокие значения w могут быть особенно выгодны в сложных сценариях маршрутизации.

Постоянно более высокая производительность при $w = 0.7$ на разных путях свидетельствует о том, что алгоритм с этой настройкой не только эффективен, но и надежен, сохраняя превосходную производительность независимо от конкретных характеристик пути.

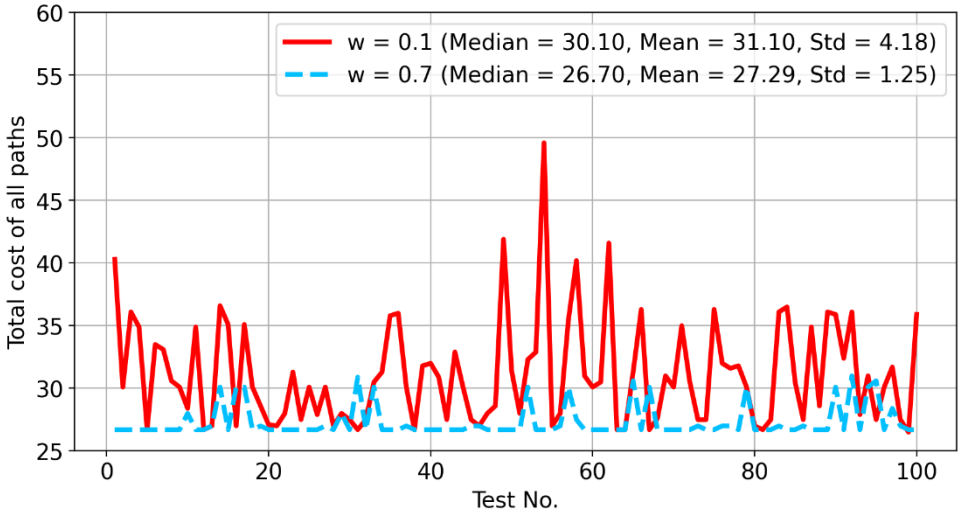


Рисунок 5.10 – Общая стоимость всех путей для каждого запуска

В ходе 100 запусков значение веса инерции $w = 0.7$ (синяя линия) чаще всего обеспечивало меньшую общую стоимость для всех путей по сравнению со значением $w = 0.1$ (красная линия). Средняя общая стоимость путей при использовании $w = 0.7$ составила 27.29, что значительно ниже значения 31.1, полученного при $w = 0.1$. Стандартное отклонение при $w = 0.7$ составило 1.25, указывая на высокую стабильность результатов. Для $w = 0.1$ стандартное отклонение было 4.18, что свидетельствует о более значительной вариабельности в общей стоимости путей.

График для $w = 0.1$ демонстрирует более частые и высокие пики, что указывает на резкое увеличение общей стоимости в некоторых случаях. Это говорит о том, что значение $w = 0.1$ может иногда приводить к менее оптимальным решениям маршрутизации. В отличие от этого, $w = 0.7$ обеспечивает более стабильные результаты во всех запусках.

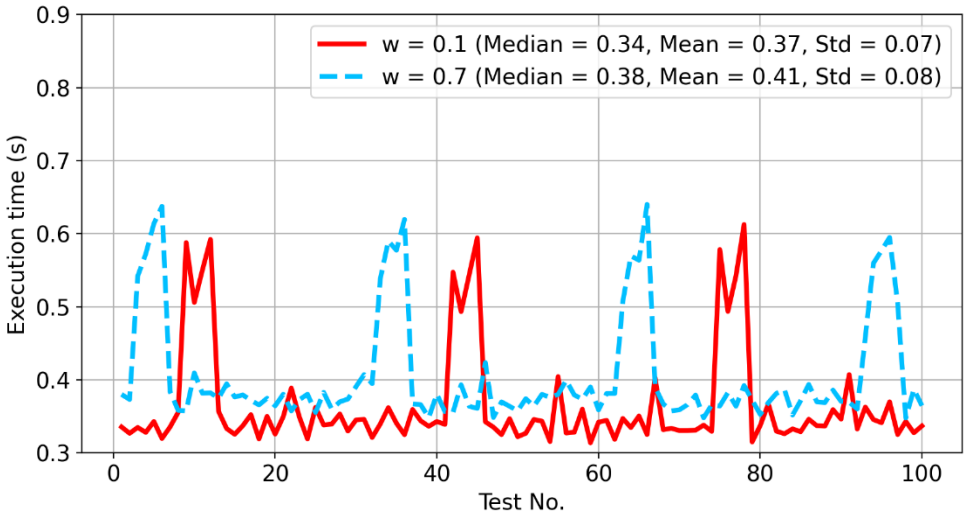


Рисунок 5.11 – Время выполнения алгоритма для каждого запуска

График показывает, что при $w = 0.7$ обычно требует немного большего времени выполнения, чем при $w = 0.1$. Это может означать, что более широкий поиск, которому способствует больший вес инерции, может иногда приводить к незначительному увеличению вычислительных затрат. Стандартное отклонение времени выполнения при $w = 0.1$ составляет 0.07 секунды, тогда как при $w = 0.7$ оно составляет 0.08 секунды, что указывает на немного большую изменчивость в выполнении при более высоком инерционном весе.

Обе линии демонстрируют колебания времени выполнения в зависимости от конкретного запуска. Наличие нескольких пиков указывает на то, что определенные запуски могут требовать больше вычислительных ресурсов, независимо от инерционного веса.

Влияние инерционного веса на производительность алгоритма стаи птиц является комплексным. Более высокое значение инерционного веса расширяет

возможности алгоритма в поиске решений, позволяя ему исследовать более широкую область пространства решений. Это особенно важно при многопутевой маршрутизации, где целью является определение нескольких жизнеспособных путей. Несмотря на то, что исследование с $w = 0.7$ иногда приводит к незначительному увеличению времени выполнения, это увеличение вычислительных затрат является приемлемым и не оказывает существенного влияния на общую эффективность алгоритма.

5.1.4 ИССЛЕДОВАНИЕ АЛГОРИТМА ИСКУССТВЕННОЙ ПЧЕЛИНОЙ КОЛОНИИ В ПКС

В алгоритме искусственной пчелиной колонии параметр, оказывающий наибольшее влияние на баланс между исследованием и эксплуатацией, является параметром $limit$. Этот параметр определяет количество попыток, в течение которых источник пищи (потенциальное решение) не улучшается, прежде чем пчела откажется от него. Более низкое предельное значение $limit$ стимулирует исследование, позволяя быстро отказаться от неэффективных решений и заставляя пчел искать новые. Более высокое предельное значение $limit$ способствует эксплуатации, предоставляя больше времени для доработки существующих решений перед переходом к следующим.

Для оценки влияния параметра $limit$ на эффективность многопутевой маршрутизации, параметры алгоритма искусственной пчелиной колонии задаются следующим образом: $N = 10, Max = 100, limit = [20, 80]$.

Результаты алгоритма искусственной пчелиной колонии показаны на следующих рисунках.

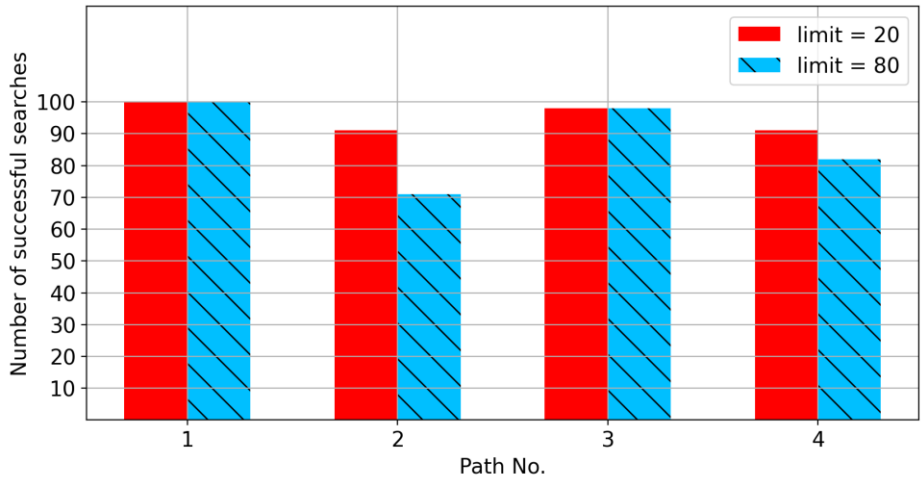


Рисунок 5.12 – Количество успешных поисков для каждого пути

Низкий предел ($limit = 20$, красные столбцы) стабильно обеспечивает высокую производительность по всем четырем путям, с уровнем успешного обнаружения близким или равным 100%. Это свидетельствует о высокой надежности поиска. Высокий предел ($limit = 80$, синие столбцы), хоть и демонстрирует хорошую производительность, обычно имеет немного более низкие показатели успеха по сравнению с $limit = 20$. Установка нижнего предела, вероятно, побуждает алгоритм быстрее отказываться от менее перспективных путей, способствуя более широкому исследованию пространства решений. Это особенно важно в сценариях, подобных многопутевой маршрутизации, где желательно найти несколько хороших решений.

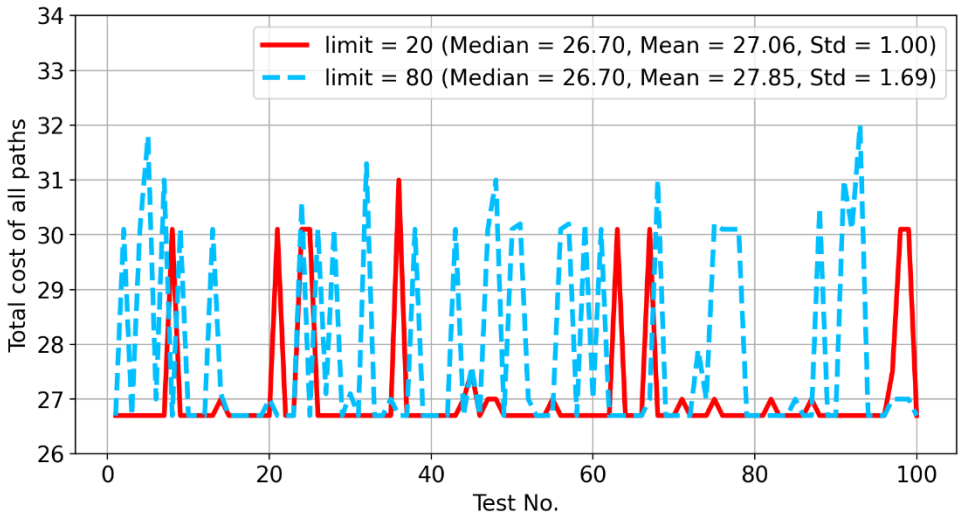


Рисунок 5.13 – Общая стоимость всех путей для каждого запуска

Средняя общая стоимость путей при $limit = 20$ (красная линия) составляет 27.06, что стабильно ниже, чем при $limit = 80$ (синяя линия) – 27.85. Это свидетельствует о том, что пути, найденные с меньшим $limit$, как правило, имеют меньшую общую стоимость. Стандартное отклонение при $limit = 20$ составляет 1, что указывает на высокую стабильность и предсказуемость общей стоимости путей. Для $limit = 80$ стандартное отклонение равно 1.69, что свидетельствует о большей изменчивости и меньшей согласованности общей стоимости путей, найденных в разных запусках.

Наблюдаемые изменения в общей стоимости пути позволяют предположить, что существует необходимость в балансе между разведкой и эксплуатацией. Увеличение $limit$ может привести к чрезмерной эксплуатации отдельных путей и, как следствие, к менее оптимальной общей стоимости. Снижение $limit$ для стимулирования более активной разведки может помочь обнаружить пути, более близкие к оптимальным.

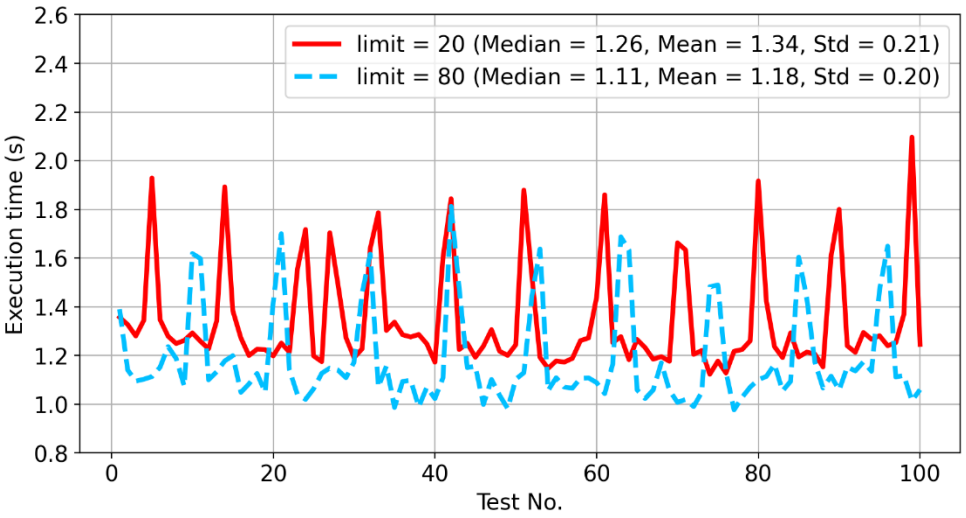


Рисунок 5.14 – Время выполнения алгоритма для каждого запуска

Среднее время выполнения при $limit = 20$ составляет 1.34 секунды, что превышает время выполнения при $limit = 80$ (1.18 секунды). Это свидетельствует о том, что установка нижнего предела limit обычно приводит к увеличению времени выполнения. Стандартное отклонение времени выполнения при $limit = 20$ составляет 0.21 секунды, тогда как при $limit = 80$ оно также составляет 0.20 секунды, что указывает на схожую изменчивость в выполнении независимо от значения $limit$.

Время выполнения существенно различается в обоих случаях, что подтверждается частыми и постоянными пиками и спадами.

Результаты, наблюдаемые на трех графиках, демонстрируют, как параметр $limit$ в алгоритме искусственной пчелиной колонии значительно влияет на его производительность в сценариях многопутевой маршрутизации. Более низкое значение $limit$ способствует более глубокому исследованию, что приводит к успешному поиску путей с меньшим общим весом, хотя и с более переменным временем выполнения. В свою очередь, более высокий $limit$ снижает изменчивость времени выполнения и немного ускоряет процесс, но за счет менее успешного поиска путей. Эти наблюдения свидетельствуют о том, что параметр $limit$ критически влияет на баланс между исследованием и эксплуатацией алгоритма, определяя его общую эффективность и результативность в поиске оптимальных путей.

5.1.5 ИССЛЕДОВАНИЕ АЛГОРИТМА СВЕТЛЯЧКОВ В ПКС

В контексте многопутевой маршрутизации в ПКС, где пространство поиска обширно и сложно, стандартный алгоритм светлячков (FA) имеет тенденцию к быстрой сходимости к локальному оптимуму, что не позволяет ему адекватно исследовать все потенциальные решения. Учитывая это ограничение, был разработан расширенный алгоритм светлячков (EFA).

Основное отличие EFA заключается в увеличении случайных движений светлячка при встрече с другим, не более ярким. Это изменение способствует более активному исследованию пространства решений, позволяя алгоритму выходить за пределы непосредственной близости от известных хороших решений. В контексте многопутевой маршрутизации это помогает исследовать более широкий диапазон путей, предотвращая преждевременную сходимость на неоптимальных путях.

Параметры алгоритма установлены следующим образом: $N = 10, Max = 100, \gamma = 1, \alpha_0 = 1, \beta_0 = 1$.

Полученные результаты показаны на следующих рисунках.

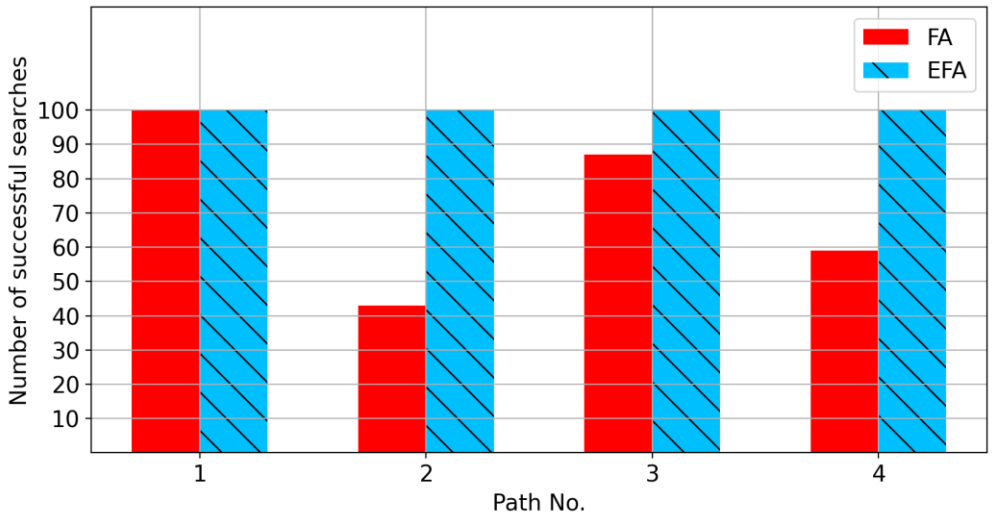


Рисунок 5.15 – Количество успешных поисков для каждого пути

EFA демонстрирует большее количество успешных поисков по всем путям. Это указывает на то, что модифицированная стратегия EFA по увеличению случайных движений эффективно улучшает исследование. Это критически важно при многопутевой маршрутизации, где поиск альтернативных путей так же важен, как и определение оптимального пути. Стабильная производительность EFA на всех путях свидетельствует о его способности эффективно избегать локальных оптимумов, что является распространенной проблемой в сложных пространствах поиска, где часто страдают стандартные алгоритмы, такие как FA.

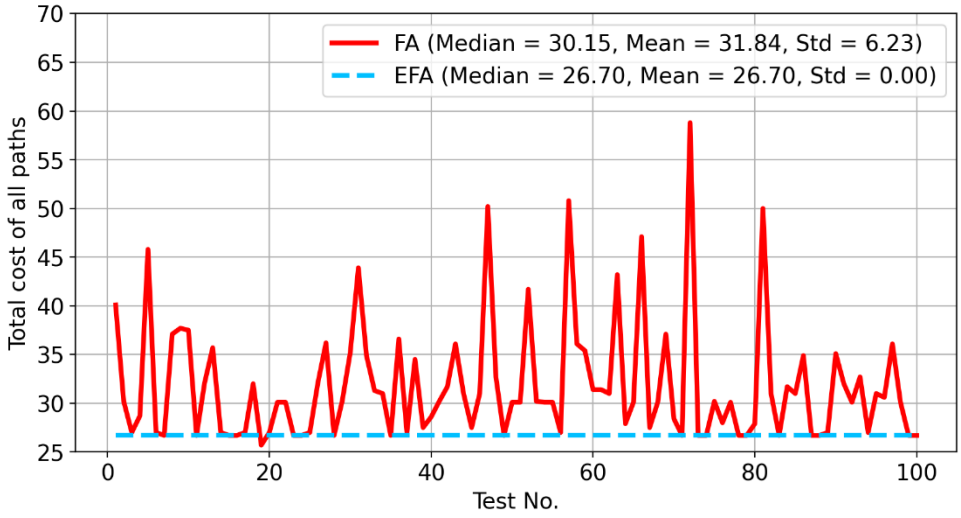


Рисунок 5.16 – Общая стоимость всех путей для каждого запуска

EFA последовательно обеспечивает меньшую общую стоимость путей (в среднем 26.7) по сравнению с FA (в среднем 31.84). Стандартное отклонение стоимости путей у EFA равно 0.0, что указывает на абсолютную стабильность и отсутствие колебаний в результатах. В отличие от этого, FA имеет стандартное отклонение 6.23, демонстрируя значительные колебания и пики общей стоимости путей.

График показывает способность EFA избегать перегруженных путях. Усовершенствованный механизм исследования, увеличивающий вероятность случайных движений при столкновении с тусклыми светлячками, позволяет алгоритму эффективно обходить неоптимальные пути.

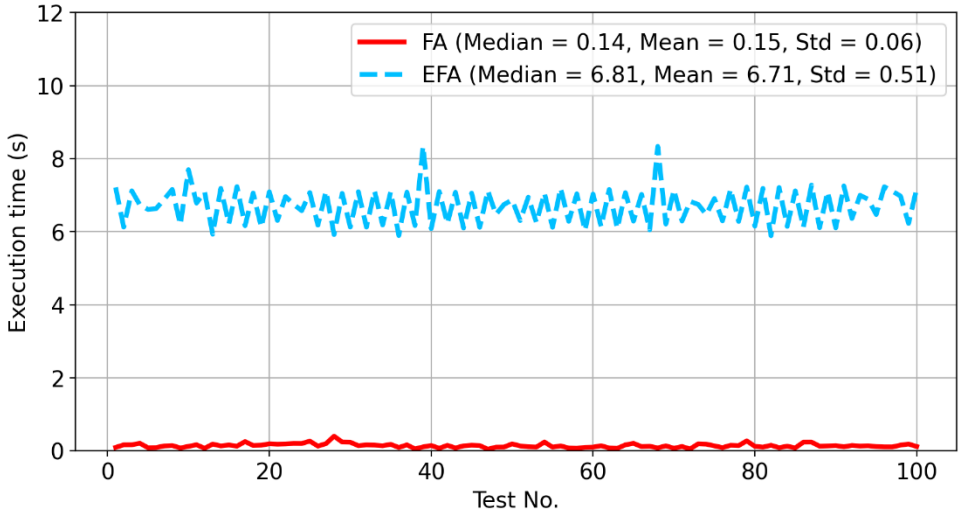


Рисунок 5.17 – Время выполнения алгоритма для каждого запуска

FA имеет значительно меньшее среднее время выполнения (0.15 секунды) по сравнению с EFA (6.71 секунды). Это значительное различие указывает на то, что, хотя EFA обеспечивает преимущества в виде поиска путей с меньшей общей стоимостью и большим количеством успешных поисков, это достигается ценой существенного увеличения времени вычислений.

Стандартное отклонение времени выполнения для FA составляет 0.06 секунды, что свидетельствует о высокой стабильности и предсказуемости алгоритма. Такая стабильность указывает на то, что FA требует относительно постоянного и предсказуемого объема вычислительных ресурсов, вероятно, благодаря менее

сложным процессам поиска пути и исследования. Время выполнения алгоритма EFA более изменчиво, со стандартным отклонением 0.51 секунды, что указывает на значительные колебания времени выполнения, связанные с более сложными вычислительными процессами. Увеличение времени выполнения EFA связано с его расширенными возможностями разведки. Вводя больше случайности в процесс поиска и избегая преждевременной сходимости к локальным оптимумам, алгоритм требует больше вычислительных ресурсов и времени для тщательного исследования пространства решений и нахождения глобального оптимума.

Сравнительный анализ времени выполнения алгоритмов многопутевой маршрутизации в ПКС выявляет ключевой аспект выбора: необходимость баланса между скоростью поиска решений и их качеством. Более длительное время выполнения EFA компенсируется его способностью находить пути с меньшей общей стоимостью и более высокой успешностью поиска, что свидетельствует о потенциале предоставления более качественных решений маршрутизации в ущерб скорости. Данный компромисс требует тщательного рассмотрения с учетом специфических требований и ограничений сетевой среды, в которой развертываются алгоритмы.

5.1.6 СРАВНИТЕЛЬНЫЙ АНАЛИЗ ПРЕДЛОЖЕННЫХ АЛГОРИТМОВ

Для сравнения предложенных алгоритмов рассмотрим экспериментальную топологию ПКС, представленную на рисунке 5.18.

Четыре найденных кратчайших пути обозначены разными цветами на рисунке 5.19.

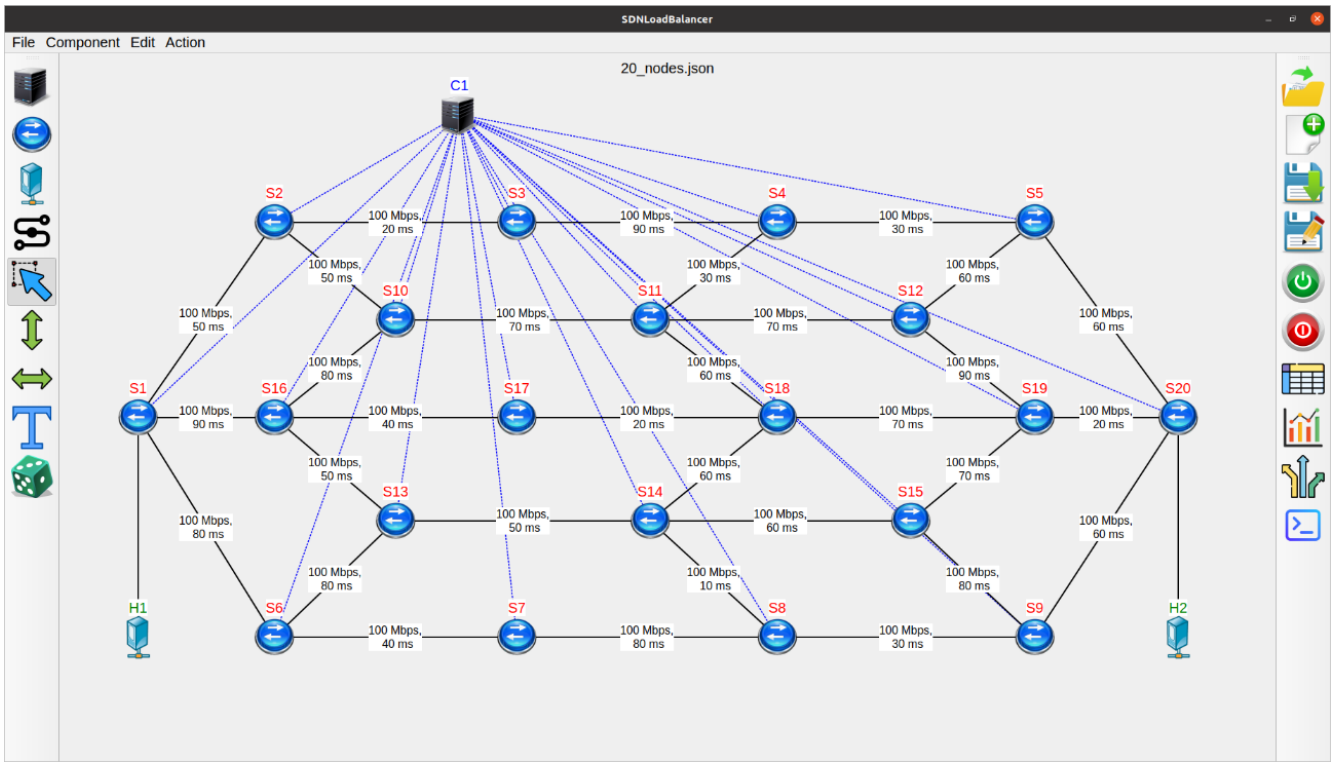


Рисунок 5.18 – Экспериментальная топология ПКС из 20 узлов

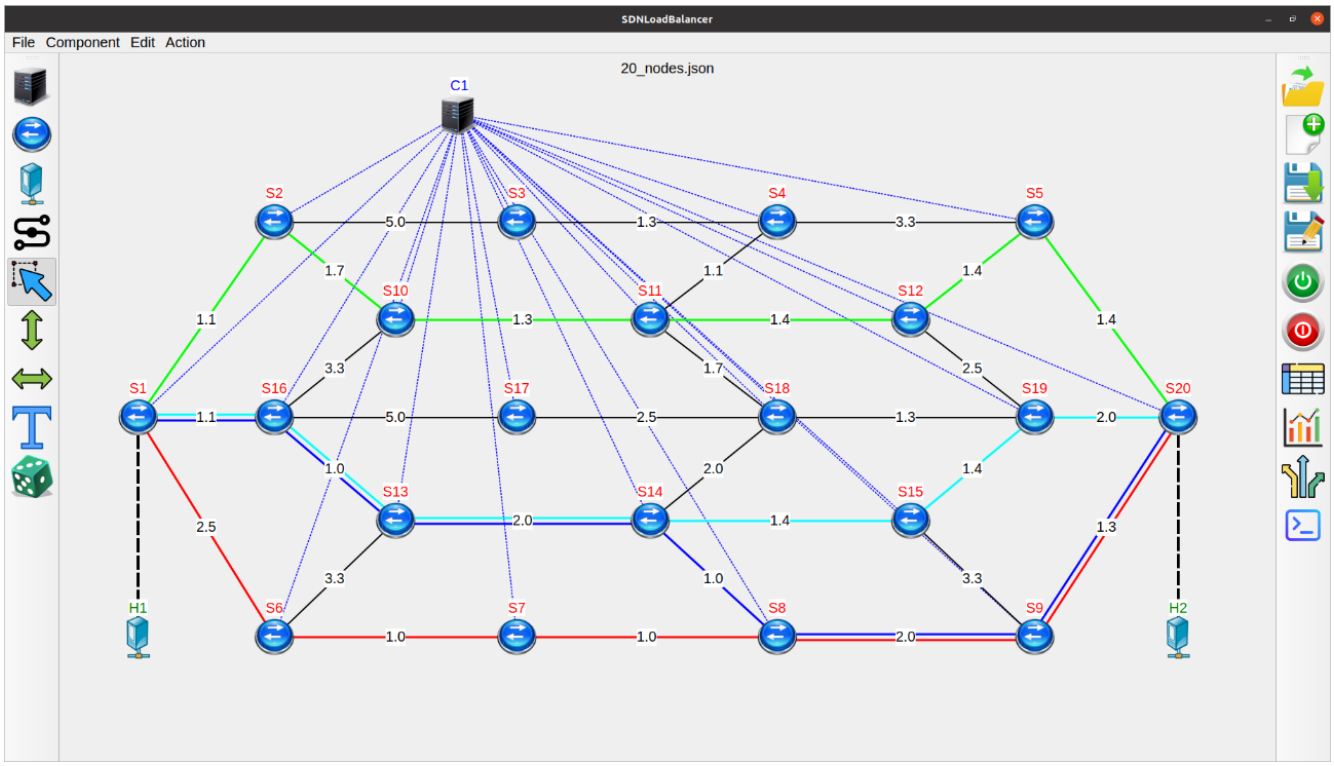


Рисунок 5.19 – Отображение полученных путей ($K = 4$)

Таблица 5.2 демонстрирует кратчайшие пути, выявленные с использованием алгоритма Йена.

Таблица 5.2 – Результат работы алгоритма Йена

№	Кратчайшие пути	Стоимость (вес)
1	S1, S6, S7, S8, S9, S20	7.8
2	S1, S2, S10, S11, S12, S5, S20	8.3
3	S1, S16, S13, S14, S8, S9, S20	8.4
4	S1, S16, S13, S14, S15, S19, S20	8.9
		Общая стоимость (вес) всех путей = 33.4

Экспериментальная топология ПКС инициализируется эмулятором Mininet, а для выполнения алгоритмов маршрутизации используется контроллер Ryu.

Параметры предлагаемых алгоритмов приведены в таблице 5.3.

Таблица 5.3 – Параметры предложенных алгоритмов для данной топологии

Алгоритм	Параметры
Генетический алгоритм	$N = 10,\ Max = 100, P_c = 0.7, P_m = 0.7$
Алгоритм стаи птиц	$N = 10,\ Max = 100, w = 0.7, \mathrm{c}_1 = 2, \mathrm{c}_2 = 2$
Алгоритм искусственной пчелиной колонии	$N = 10, Max = 100, limit = 20$
Оптимизации муравьиной колонии	$N = 10, Max = 100, \rho = 0.1, \alpha = 1, \beta = 2, q_0 = 0.5, Q = 1$
Алгоритм светлячков	$N = 10, Max = 100, \gamma = 1, \beta_0 = 1, \alpha_0 = 1$

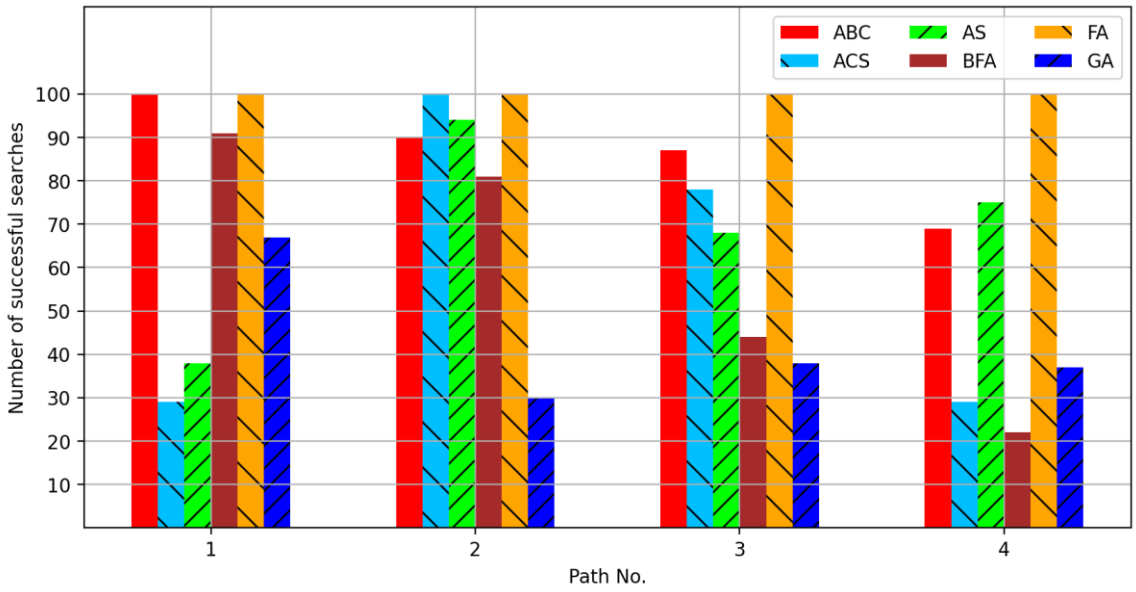


Рисунок 5.20 – Количество успешных поисков для каждого пути

FA стабильно показывает высокую эффективность на всех путях, достигая почти 100% успеха в поиске каждого из них. Он поддерживает этот уровень успеха как на простых, так и на сложных путях, демонстрируя незначительные различия в производительности.

ABC начинает с выдающихся результатов на первой пути, но эффективность постепенно уменьшается по мере усложнения путей. Он сохраняет свою относительную эффективность, однако не достигает столь же высоких показателей, как на первой пути.

BFA демонстрирует высокую эффективность на первом пути, аналогично ABC. Однако, подобно ABC, его успешность снижается при работе с более сложными путями, показывая тенденцию к уменьшению успешности от пути 1 к пути 4.

AS и ACS демонстрируют среднюю эффективность, особенно хорошо справляясь с навигацией в средах средней сложности, что подтверждается их лучшими результатами на путях 2 и 3. Тем не менее, они сталкиваются с проблемами как на простых, так и на сложных путях: AS плохо работает на пути 1, а ACS также показывает слабые результаты на путях 1 и 4.

GA демонстрирует худшую производительность среди рассмотренных алгоритмов. На простейшем пути GA способен хорошо сходиться к оптимальному решению. Однако, с увеличением сложности пути, наблюдается снижение эффективности GA. Это свидетельствует о том, что GA способен эффективно решать простые задачи маршрутизации, но сталкивается с трудностями при оптимизации более сложных путей.

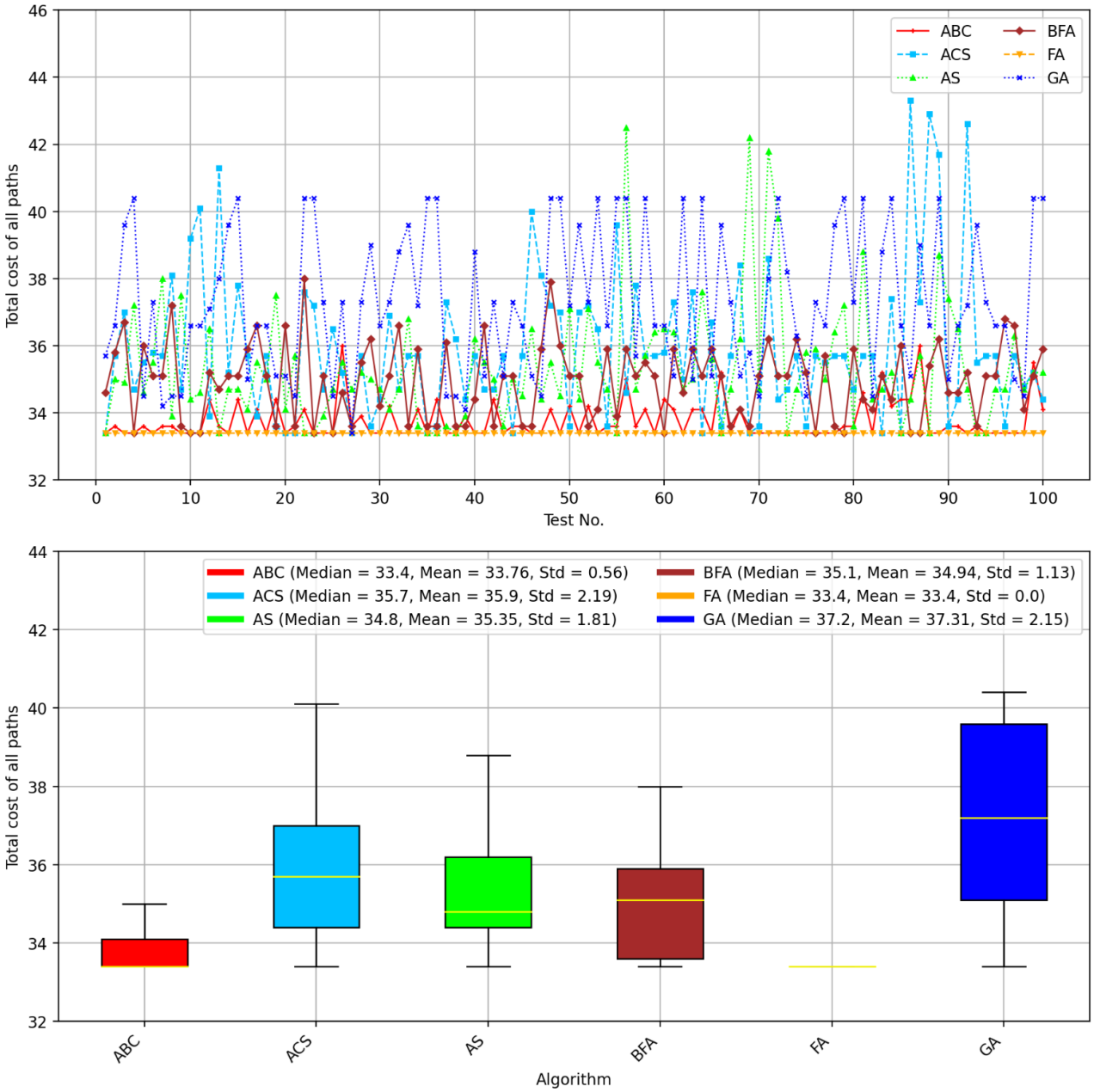


Рисунок 5.21 – Общая стоимость всех путей для каждого алгоритма

FA демонстрирует очень стабильную производительность, с линией, которая идет строго горизонтально на уровне 33.4 во всех 100 запусках. Это означает, что FA всегда находит одинаковую общую стоимости всех путей без колебаний между запусками. FA показывает оптимальные результаты и всегда сходится к наилучшим путям. FA имеет очень маленький размах результатов, почти в виде одной линии на уровне 33.4, что указывает на нулевое стандартное отклонение. Это значит, что алгоритм всегда находит оптимальные пути. FA демонстрирует абсолютную стабильность и очень хорошо подходит для задач, требующих максимальной стабильности.

ABC показывает довольно стабильную производительность, при этом суммарная длина всех путей слегка колеблется вокруг 33.76. Хотя наблюдаются небольшие отклонения, ABC чаще всего находит пути, близкие к оптимальным, с редкими случаями получения немного больших стоимостей. Размах результатов ABC небольшой, с медианой на уровне 33.4, что очень близко к результатам FA. Это указывает на то, что ABC обладает хорошей способностью к эксплуатации и быстро сходится к решениям, близким к оптимальным. Небольшое стандартное отклонение 0.56 также подтверждает, что ABC – очень эффективный алгоритм с высокой стабильностью.

ACS демонстрирует большие колебания, с общей стоимостью всех путей, варьирующейся от 34 до более чем 43. Это указывает на то, что ACS обладает сильной способностью к исследованию, но не всегда сходится к оптимальным решениям, что приводит к значительным различиям в результатах. Размах результатов ACS больше, с медианой на уровне 35.7 и стандартным отклонением 2.19. Это свидетельствует о том, что ACS может находить различные решения, но часто не эффективен, как FA и ABC, при нахождении кратчайших путей.

AS также демонстрирует значительные колебания, аналогичные ACS, с общей стоимостью путей от 34 до более чем 42. Это говорит о том, что AS не является стабильным алгоритмом и в некоторых запусках находит решения с большой общей

стоимостью путей. Размах результатов AS относительно велик, с медианой на уровне 34.8 и стандартным отклонением 1.81. Это указывает на то, что AS обладает сильной исследовательской способностью, но не всегда может эффективно найти наилучшие решения, как это делают ABC и FA.

BFA показывает меньший диапазон, чем ACS и AS, с общей стоимостью путей от 34 до 38. Это указывает на лучшую способность BFA к сходимости, однако в ряде запусков он все же часто находит неоптимальные решения. Размах результатов BFA меньше, чем у ACS и AS, с медианой на уровне 35.1 и стандартным отклонением 1.13. Это говорит о том, что BFA показывает баланс между исследованием и эксплуатацией, но не всегда достигает таких хороших решений, как FA и ABC.

GA показывает наибольшие колебания, с общей стоимостью путей от 34 до 41. Это указывает на то, что GA обладает сильной исследовательской способностью, но часто не сходится к оптимальным решениям. Во многих запусках алгоритм находит пути с большой общей стоимостью. Размах результатов GA самый большой, с медианой на уровне 37.2 и высоким стандартным отклонением 2.15. Это свидетельствует о том, что GA часто находит пути с большой общей стоимостью и менее стабилен при нахождении кратчайших путей. GA может быть полезен в задачах, где требуется исследовать большие пространства решений, но он не лучший выбор для нахождения кратчайших путей с высокой стабильностью.

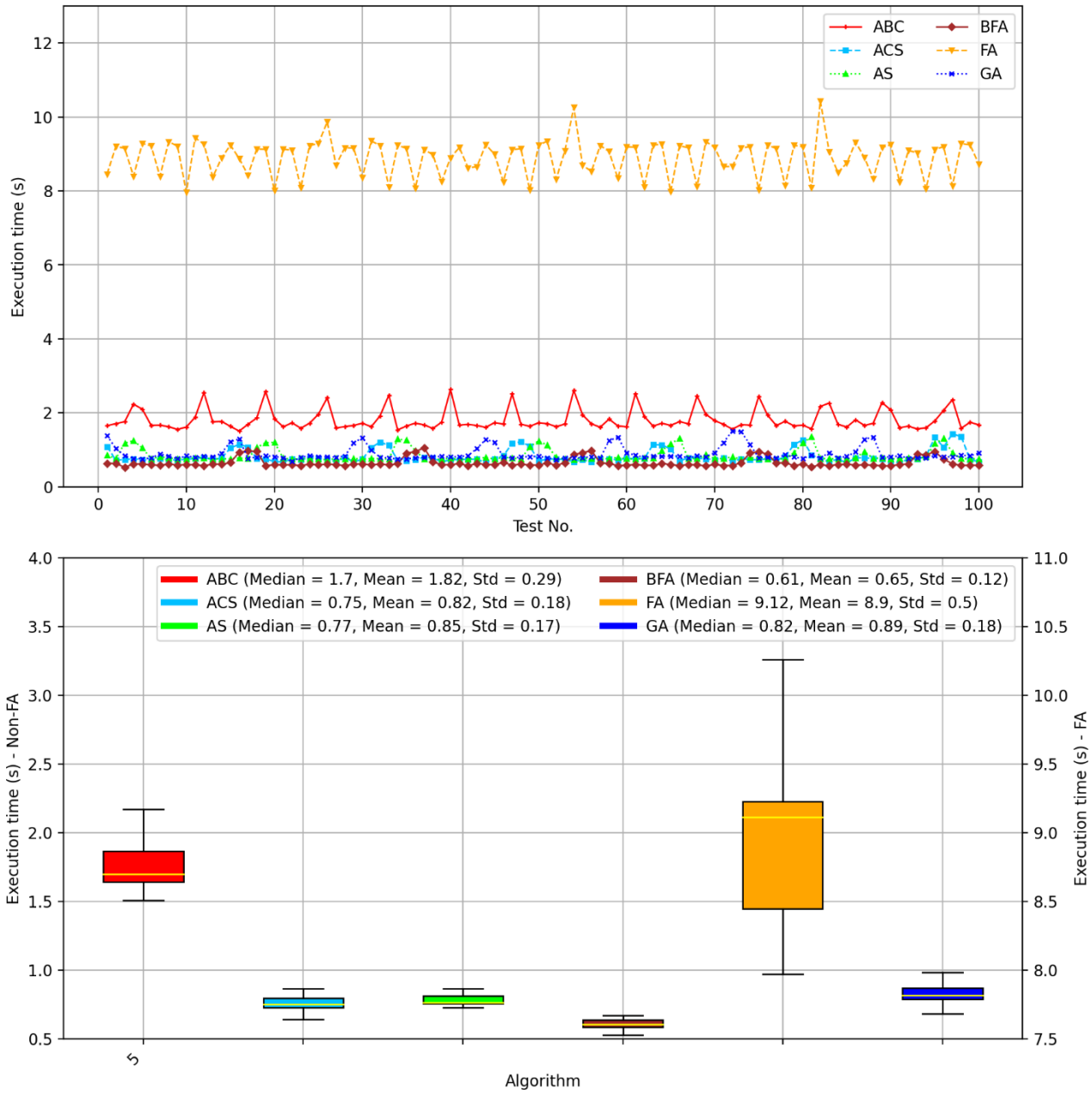


Рисунок 5.22 – Время выполнения каждого алгоритма

FA имеет наибольшее среднее время выполнения среди всех алгоритмов – 9.12 секунды. Это указывает на то, что FA требует больше вычислительных ресурсов, возможно из-за глубокой оптимизации решений для поиска наилучшего результата. Однако, FA демонстрирует нестабильность во времени выполнения, что подтверждается высоким стандартным отклонением 0.5. Это означает, что время выполнения FA значительно варьируется между запусками. Большой размах

результатов и видимые колебания во времени выполнения показывают, что FA является наименее стабильным алгоритмом с точки зрения времени выполнения. Хотя FA достигает высокой точности, его нестабильность во времени выполнения делает его менее подходящим для задач, где требуется стабильная производительность.

ABC имеет среднее время выполнения 1.82 секунды, что меньше, чем у FA, но все еще больше, чем у других алгоритмов. ABC имеет тенденцию к нестабильности во времени выполнения, что выражается в стандартном отклонении 0.29, показывая, что некоторые запуски занимают больше времени. Хотя общая производительность ABC хорошая, она менее эффективна для задач, требующих короткого и стабильного времени выполнения.

ACS имеет низкое среднее время выполнения – всего 0.82 секунды, что говорит о его высокой эффективности по времени. Также ACS демонстрирует низкое стандартное отклонение 0.18, что указывает на стабильность во времени выполнения между запусками. Небольшой размах результатов отражает высокую стабильность времени выполнения. Благодаря быстроте и стабильности, ACS является одним из идеальных алгоритмов для задач, где требуется быстрая обработка и последовательные результаты.

AS также имеет низкое среднее время выполнения – 0.85 секунды. Как и ACS, AS демонстрирует низкое стандартное отклонение 0.17, что свидетельствует о высокой стабильности и скорости выполнения. Небольшой размах результатов и стабильное время выполнения подтверждают, что AS является алгоритмом с быстрым и стабильным временем выполнения, что делает его подходящим для задач, требующих короткого времени выполнения.

BFA имеет самое короткое среднее время выполнения – всего 0.65 секунды. Это делает BFA самым быстрым среди всех протестированных алгоритмов. Также BFA имеет самое низкое стандартное отклонение 0.12, что означает, что его время выполнения остается стабильным на всех запусках. Очень маленький размах результатов показывает, что BFA является самым стабильным алгоритмом с точки

зрения времени выполнения. BFA – отличный выбор, когда требуется быстрая обработка и высокая стабильность времени выполнения.

GA имеет среднее время выполнения 0.89 секунды, что является относительно небольшим и почти таким же, как у ACS и AS. Однако GA имеет несколько большее стандартное отклонение 0.18, что указывает на некоторые колебания времени выполнения на отдельных запусках. Хотя GA демонстрирует низкую производительность, он остается хорошим выбором в случаях, когда требуется быстрое выполнение.

5.1.7 ИССЛЕДОВАНИЕ ПРОЦЕССОВ ИНТЕЛЛЕКТУАЛЬНОЙ МНОГОПУТЕВОЙ МАРШРУТИЗАЦИИ СО СЛОЖНОЙ ЦЕЛЕВОЙ ФУНКЦИЕЙ

В задачах многопутевой маршрутизации в ПКС, целевая функция определяется как функция оценки качества маршрута, также известная как функция приспособленности в алгоритмах роевого интеллекта. Алгоритмы оптимизации стремятся минимизировать или максимизировать эту функцию, чтобы выбрать маршруты, которые наилучшим образом удовлетворяют требованиям к производительности, надежности, пропускной способности и другим важным показателям сети.

Алгоритм Йена является эффективным методом для нахождения K кратчайших путей между двумя узлами в графе на основе суммы весов ребер. Однако этот алгоритм хорошо работает только в тех случаях, когда целевая функция может быть выражена как сумма независимых весов ребер. Это означает, что если целевая функция не может быть разложена на независимые веса ребер, алгоритм Йена сталкивается с трудностями при оптимизации, особенно когда целевая функция зависит от характеристик всего маршрута или включает сложные взаимосвязи между ребрами.

Алгоритм Йена успешно справляется с задачей оптимизации, если целевая функция имеет вид:

$$f(P) = \sum_{e \in P} w(e).$$

или:

$$f(P) = \sum_{e \in P} \big(w(e) + delay(e)\big).$$

где $delay(e)$ – задержка канала.

В этом случае можно пересчитать веса как $w'(e) = w(e) + delay(e)$, и алгоритм Йена будет находить маршруты с наименьшими значениями новой суммы весов.

Алгоритм Йена не может обрабатывать целевые функции, которые зависят от всего маршрута или имеют сложные зависимости между ребрами, поскольку такие функции не могут быть представлены в виде суммы независимых весов ребер.

Например:

$$f(P) = \prod_{e \in P} reliability(e).$$

Данная функция рассчитывает надежность всего маршрута на основе произведения надежности всех ребер. Так как функция требует учета всех ребер маршрута, она не может быть разделена на независимые компоненты.

В отличие от алгоритма Йена, алгоритмы роевого интеллекта обеспечивают гибкость в работе с более сложными целевыми функциями, поскольку они оптимизируют маршрут целиком, не требуя независимости весов ребер. Алгоритмы роевого интеллекта могут напрямую оптимизировать целевые функции, включающие произведения, отношения или другие сложные зависимости между ребрами.

Рассмотрим сложную многокритериальную целевую функцию, определенную следующим образом:

$$f(P) = \frac{\sum_{e \in P} w(e)}{bw_{min}(P)},$$

где:

- $\sum_{e \in P} w(e)$ – сумма весов всех ребер маршрута P, которая отражает уровень использования ресурсов маршрута: маршруты с меньшей загрузкой имеют меньший общий вес, что способствует более эффективному распределению сетевых ресурсов.
- $bw_{min}(P)$ – минимальная пропускная способность среди всех ребер маршрута P. Это значение характеризует способность маршрута выдерживать нагрузку: чем выше $bw_{min}(P)$, тем лучше маршрут справляется с передачей данных без перегрузок.

В данной целевой функции значение $bw_{min}(P)$ зависит от всего маршрута P, так как требует вычисления пропускной способности всех ребер и выбора минимального значения. Это не может быть представлено в виде независимого веса для каждого ребра. Следовательно, алгоритм Йена не может напрямую обработать такую целевую функцию.

Чтобы использовать алгоритм Йена в данном случае, можно выполнить следующие шаги:

Шаг 1. Сначала с помощью алгоритма Йена находятся K кратчайших путей от источника до получателя на основе суммы весов $\sum_{e \in P} w(e)$.

Шаг 2. Для каждого маршрута из набора K вычисляется значение сложной целевой функции $f(P) = \frac{\sum_{e \in P} w(e)}{bw_{min}(P)}$.

Шаг 3. Отсортировать маршруты из набора K по возрастанию значений целевой функции $f(P)$, чтобы определить наиболее эффективные маршруты.

Для алгоритмов роевого интеллекта во время выполнения необходимо лишь заменить функцию приспособленности рассматриваемой целевой функцией.

Рассмотрим следующую экспериментальную топологию ПКС.

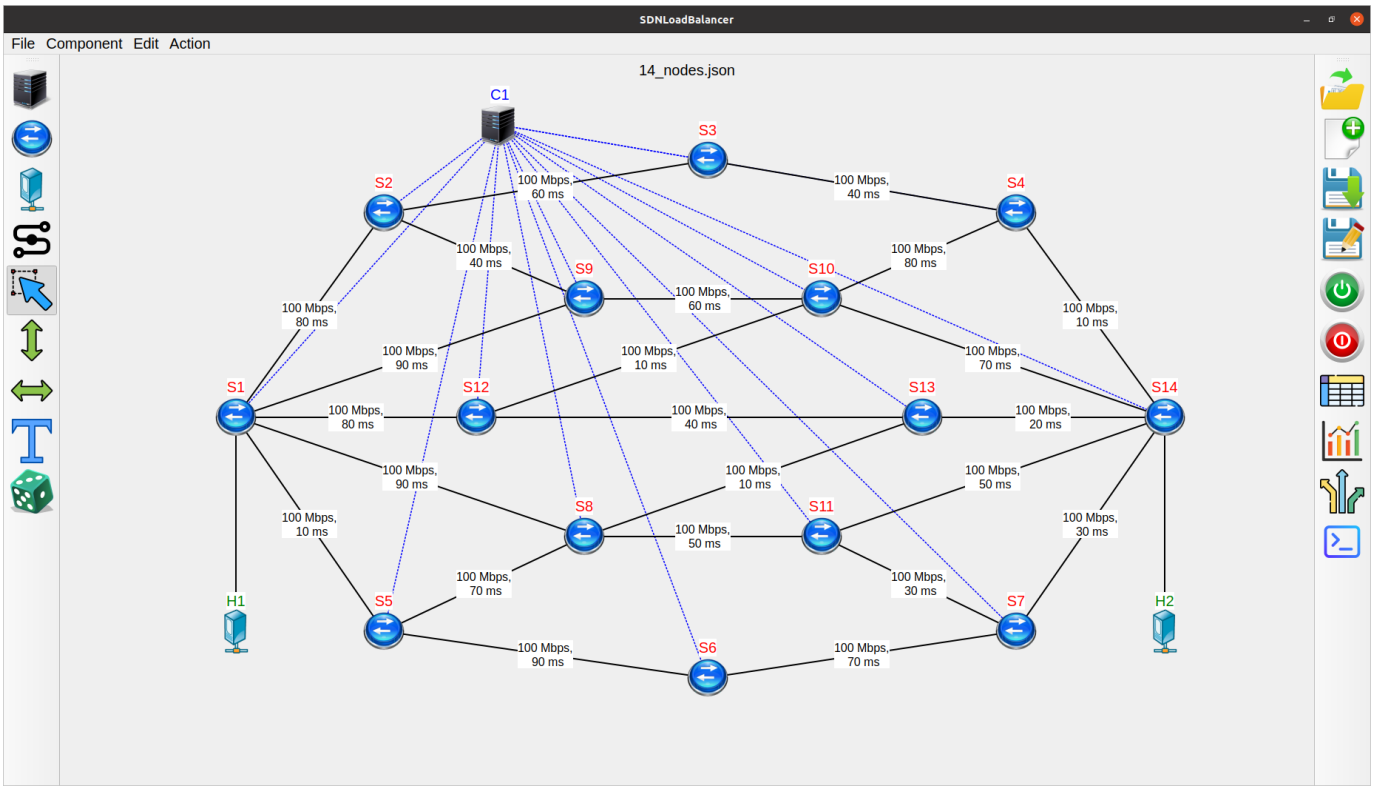


Рисунок 5.23 – Экспериментальная топология ПКС из 14 узлов

Маршруты и соответствующие значения целевой функции, полученные с помощью алгоритма Йена, представлены в таблице 5.4.

Таблица 5.4 – Результат работы алгоритма Йена

№	Кратчайшие пути	Значение целевой функции
1	S1, S8, S11, S14	4.25
2	S1, S5, S8, S11, S14	5.50
3	S1, S5, S6, S7, S14	8.67
4	S1, S12, S10, S14	18.0
		Обшее значение целевой функции всех путей = 36.42

Результаты после 100 запусков алгоритмов роевого интеллекта представлены на следующих рисунках.

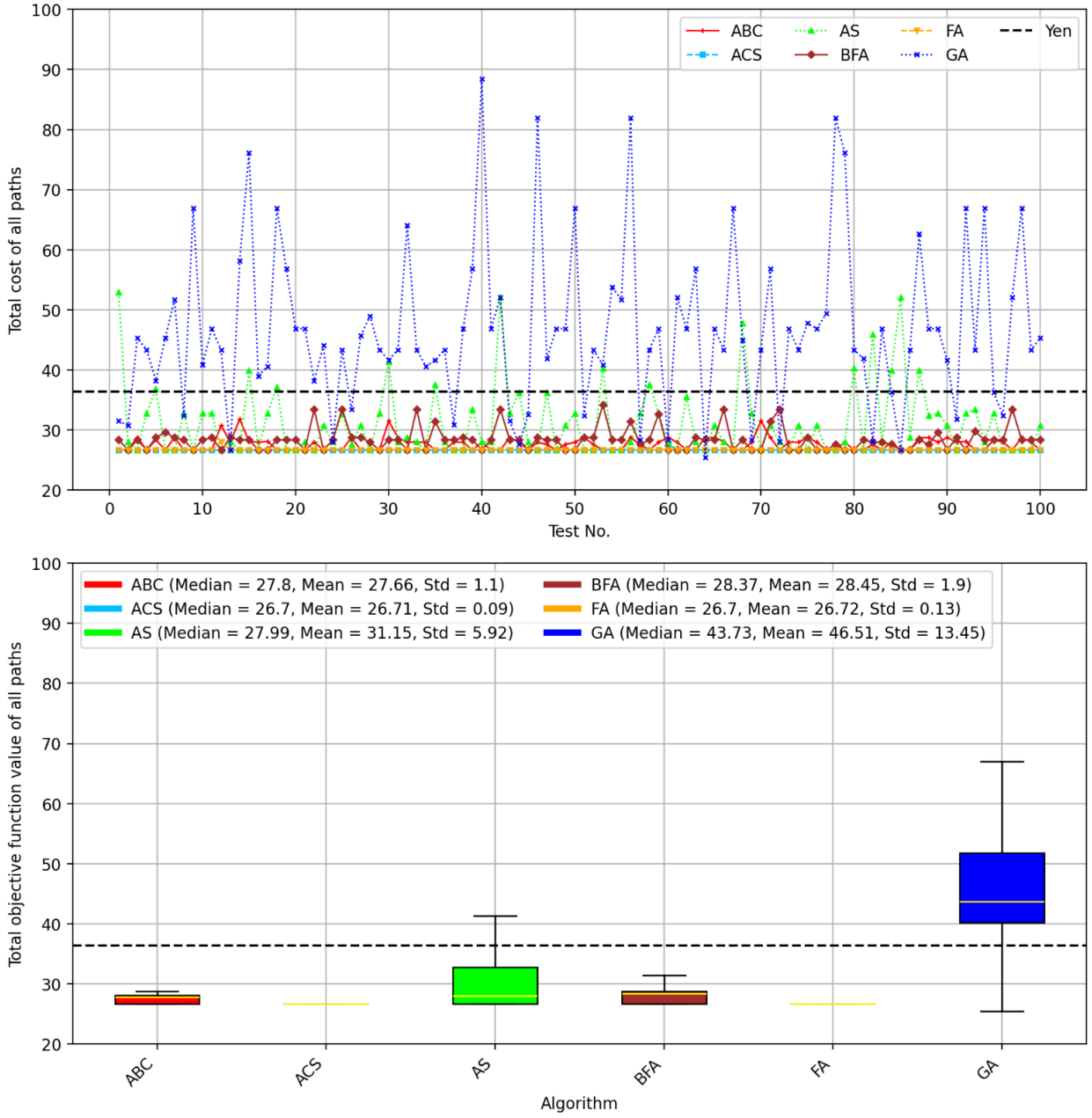


Рисунок 5.24 – Обшее значение целевой функции всех путей для каждого алгоритма

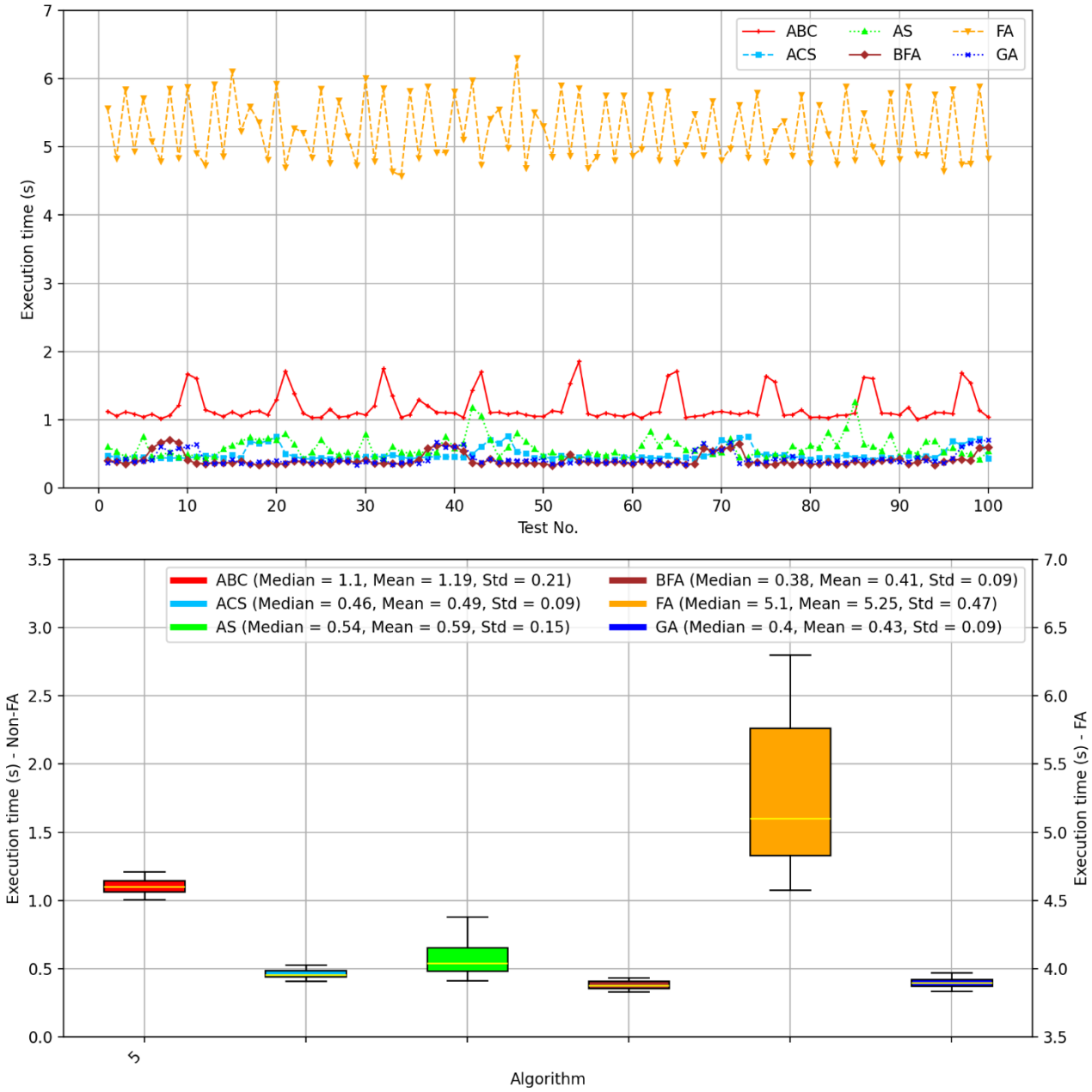


Рисунок 5.25 – Время выполнения каждого алгоритма

- ACS является наиболее эффективным алгоритмом как с точки зрения значения целевой функции, так и времени выполнения. Он обеспечивает стабильные и оптимальные результаты при быстром времени выполнения, что делает его лучшим выбором среди роевых алгоритмов.
- FA достигает очень хороших результатов по значению целевой функции, но время выполнения значительно дольше по сравнению с другими алгоритмами. FA

подходит для задач, где важна максимальная оптимизация, но требования к скорости выполнения не столь критичны.

- BFA демонстрирует хороший баланс между значением целевой функции и временем выполнения, что делает его надежным выбором для многих задач, требующих стабильной производительности.
- ABC показывает хорошие результаты по значению целевой функции, но время выполнения заметно дольше по сравнению с более быстрыми алгоритмами, такими как ACS и BFA.
- AS обеспечивает менее стабильные результаты по значению целевой функции, с колебаниями между запусками. Хотя время выполнения остается конкурентоспособным, эффективность оптимизации ниже, чем у других алгоритмов.
- GA является наименее эффективным алгоритмом в данном эксперименте. Несмотря на очень быстрое время выполнения, значения целевой функции остаются высокими и нестабильными.
- Алгоритмы роевого интеллекта, такие как ACS, FA и BFA, явно превосходят алгоритм Йена по значениям целевой функции. Они находят решения с более низкими значениями целевой функции, и некоторые из них (например, ACS и BFA) обеспечивают быстрое время выполнения, предлагая как стабильность, так и высокую эффективность оптимизации.

Алгоритм Йена является эффективным решением для задач с простыми целевыми функциями, которые могут быть выражены как сумма весов ребер. Однако для сложных целевых функций, таких как отношение суммы весов к минимальной пропускной способности, алгоритм Йена не может быть применен напрямую. Алгоритмы роевого интеллекта превосходят Йена за счет способности оптимизировать сложные целевые функции, зависящие от всех характеристик маршрута, и находить глобально оптимальные решения.

5.2 ИССЛЕДОВАНИЕ ПРОЦЕССОВ НЕЙРОСЕТЕВОЙ МНОГОПУТЕВОЙ МАРШРУТИЗАЦИИ НА ОСНОВЕ РОЕВОГО ИНТЕЛЛЕКТА

Рассмотрим топологию с 10 узлами и 14 каналами, как показано на рисунке 5.26. Задача состоит в том, чтобы найти четыре кратчайших пути между хостами H1 и H2. Метрики каналов могут принимать множество разных значений из-за изменений состояния сети.

Предположим, что в некоторый момент времени t информация о топологии предоставлена, как показано на рисунке 5.26, тогда четыре найденных кратчайших пути представлены разными цветами, как показано на рисунке 5.27.

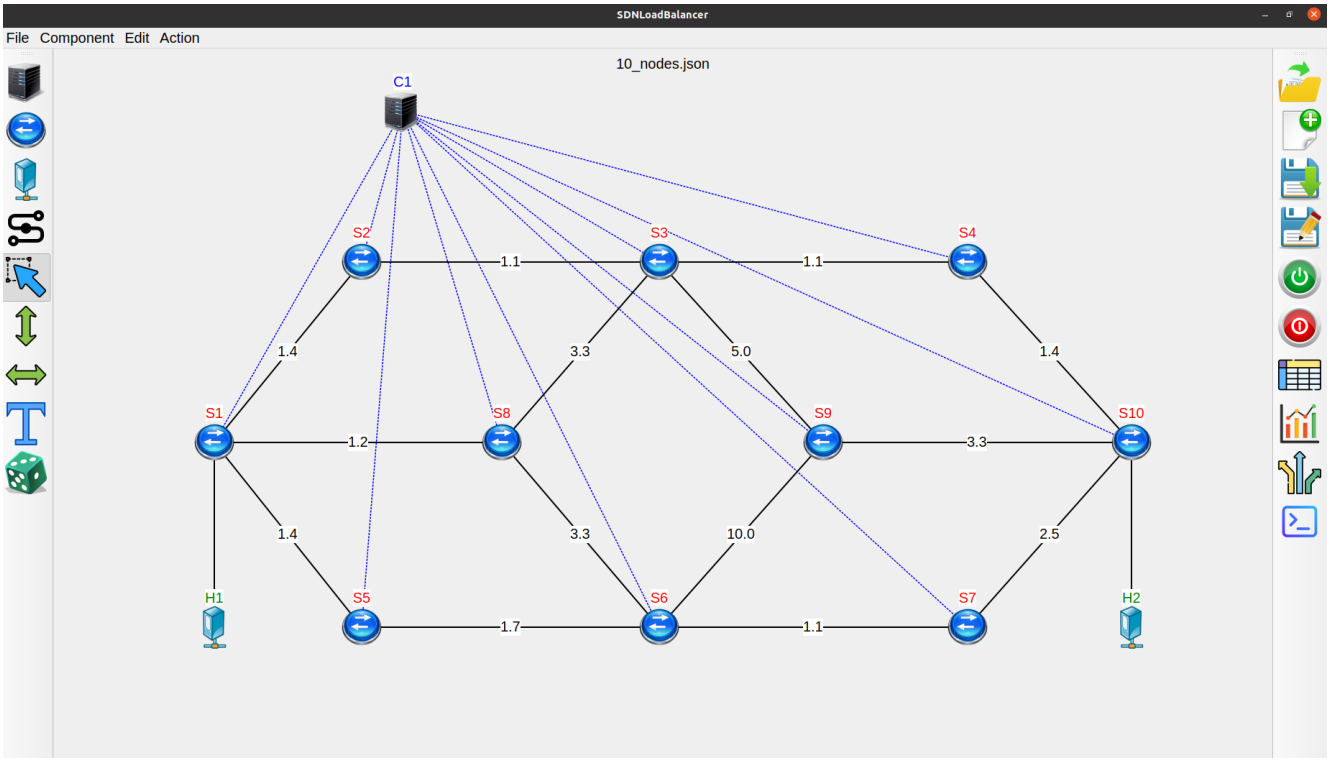


Рисунок 5.26 – Экспериментальная топология ПКС из 10 узлов

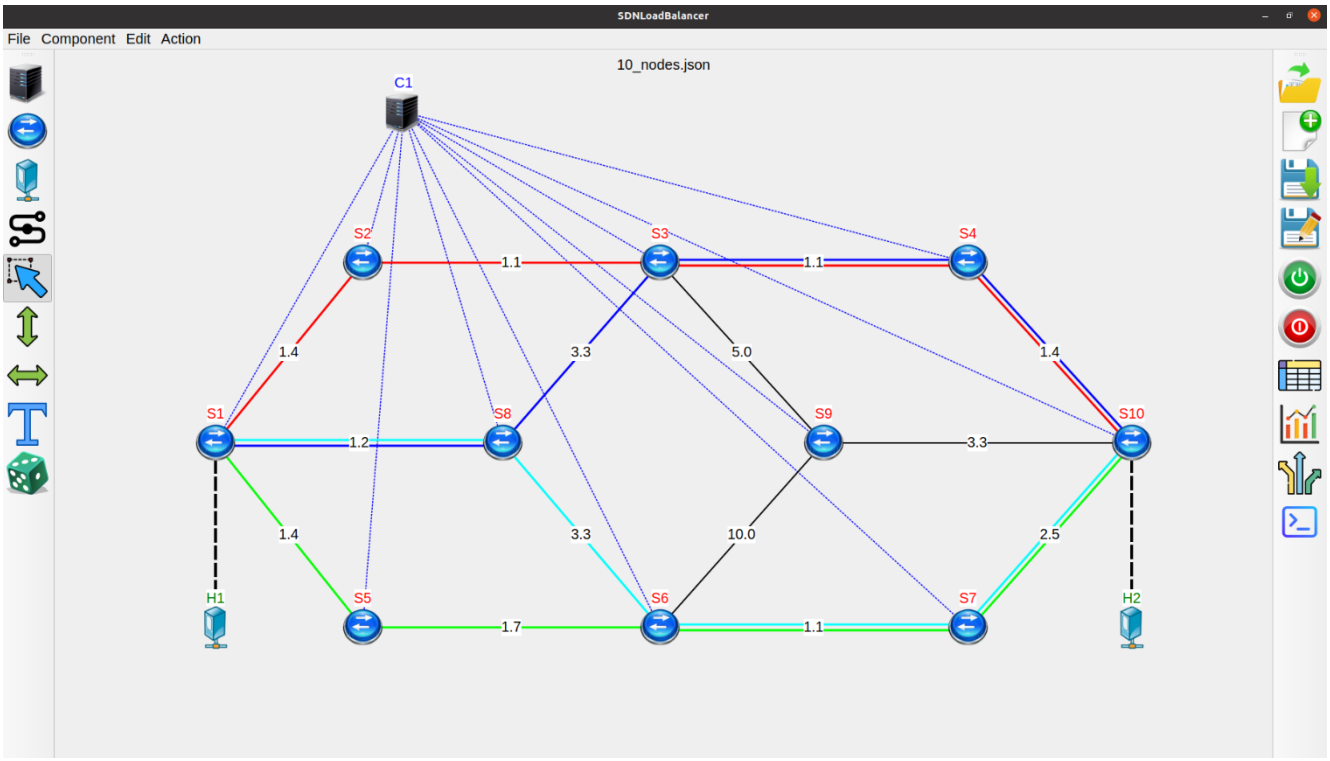


Рисунок 5.27 – Отображение полученных путей

Соотношение набора данных для обучения, проверки и тестирования составляет 60/20/20, что является стандартной практикой для обеспечения баланса между обучением и проверкой эффективности модели, позволяя одновременно избежать переобучения и обеспечить адекватную валидацию и тестирование.

Параметры алгоритмов задаются, как показано в таблице 5.5.

Таблица 5.5 – Параметры предложенных алгоритмов

Алгоритм	**Параметры**
Генетический алгоритм	$N = 10,\ Max = 10, P_c = 0.7, P_m = 0.1$
Алгоритм стаи птиц	$N = 10,\ Max = 10, w = 0.7, \mathrm{c}_1 = 2, \mathrm{c}_2 = 2$
Алгоритм искусственной пчелиной колонии	$N = 10, Max = 10, limit = 5$
Оптимизации муравьиной колонии	$N = 10, Max = 10, \rho = 0.1, \alpha = 1, \beta = 2, q_0 = 0.9, Q = 0.01$
Алгоритм светлячков	$N = 10, Max = 10, \gamma = 1, \beta_0 = 1, \alpha_0 = 1$

Из таблицы 2.4 видно, что пространство гиперпараметров очень широкое и разнообразное, включая множество переменных, таких как количество слоев, количество нейронов, Dropout, тип ячейки, алгоритм оптимизации, скорость обучения и размер батча. В этом контексте выбор подходящего базового метода для сравнения с алгоритмами роевого интеллекта является важным фактором для обеспечения объективности и эффективности исследования.

- Хотя поиск по сетке и является всеобъемлющим, так как проверяет каждую комбинацию гиперпараметров, его вычислительная стоимость становится непомерно высокой при большом количестве гиперпараметров и возможных значений. Это делает поиск по сетке неэффективным в данном сложном пространстве гиперпараметров.
- Байесовская оптимизация, хоть и более интеллектуальна и эффективна в сужении области поиска на основе предыдущих экспериментов, сложнее в реализации и требует больше времени на вычисления, что делает ее непригодной в качестве простой базовой линии для сравнения.
- Напротив, случайный поиск с его простым подходом и широким охватом пространства гиперпараметров не требует сложной настройки и может быть быстро развернут. Алгоритмы случайного поиска и роевого интеллекта полагаются на исследование большого пространства путем оценки множества различных комбинаций гиперпараметров. В то время как алгоритмы роевого интеллекта используют сложные стратегии оптимизации для управления поиском, случайный поиск основывается на случайном выборе. Сравнение случайного поиска и алгоритмов роевого интеллекта помогает подчеркнуть разницу в эффективности использования стратегий интеллектуальной оптимизации по сравнению со случайным подходом. Поэтому использование случайного поиска в качестве базовой линии является разумным выбором для сравнения с алгоритмами роевого интеллекта.

Чтобы обеспечить справедливость при сравнении с алгоритмами роевого интеллекта, количество попыток случайного поиска составляет $N \times Max = 100$. Это

означает, что случайный поиск выполнит 100 случайных испытаний в пространстве гиперпараметров, что эквивалентно общему количеству обновлений или оценок, которые выполняют алгоритмы роевого интеллекта.

Из-за стохастического характера алгоритмов роевого интеллекта каждый их запуск может давать различные результаты. Для обеспечения более точных и надежных оценок эффективности алгоритмов, каждый из них запускается многократно, а именно 30 раз. Такая практика позволяет учесть и минимизировать возможные вариации результатов, предоставляя более полное и объективное представление о производительности алгоритма.

Полученные результаты представлены на следующих рисунках.

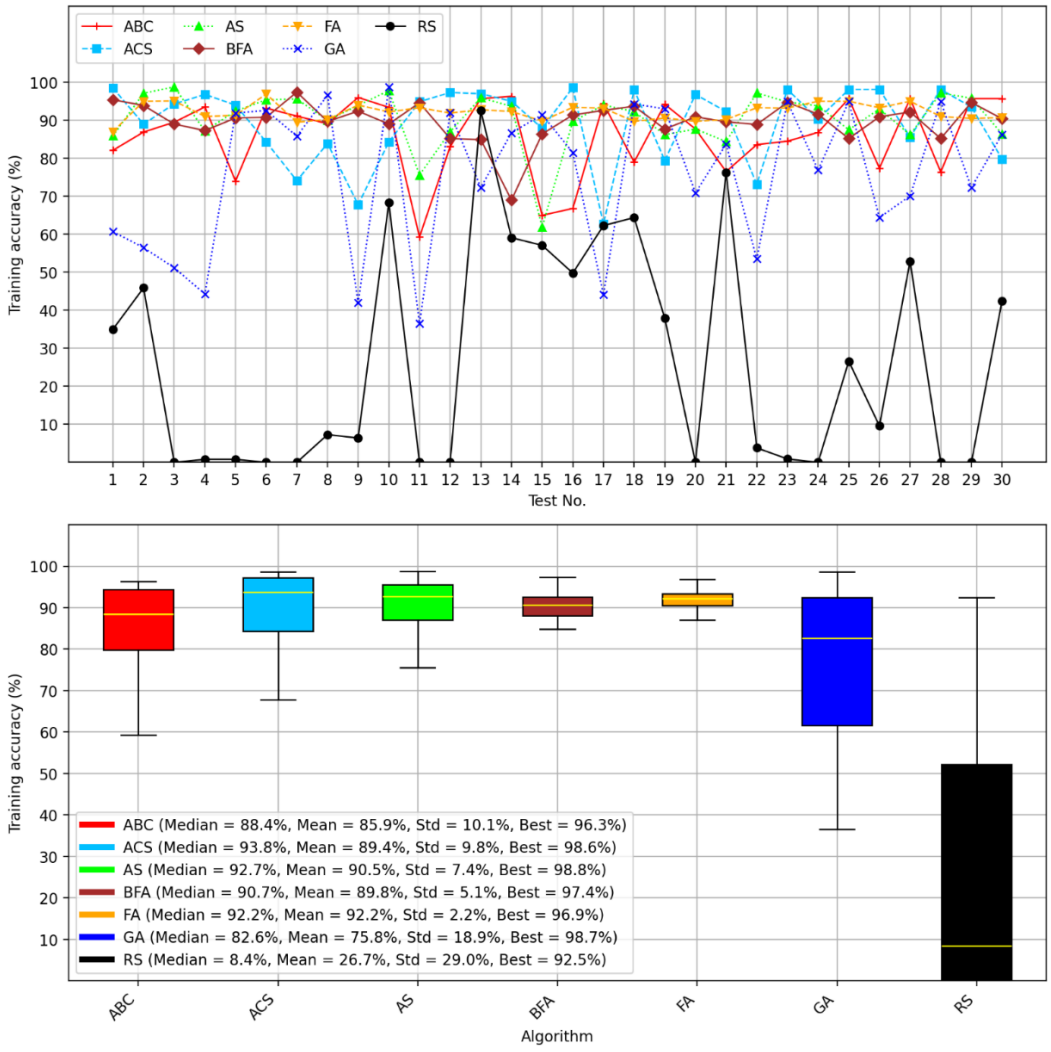


Рисунок 5.28 – Точность обучения каждого алгоритма

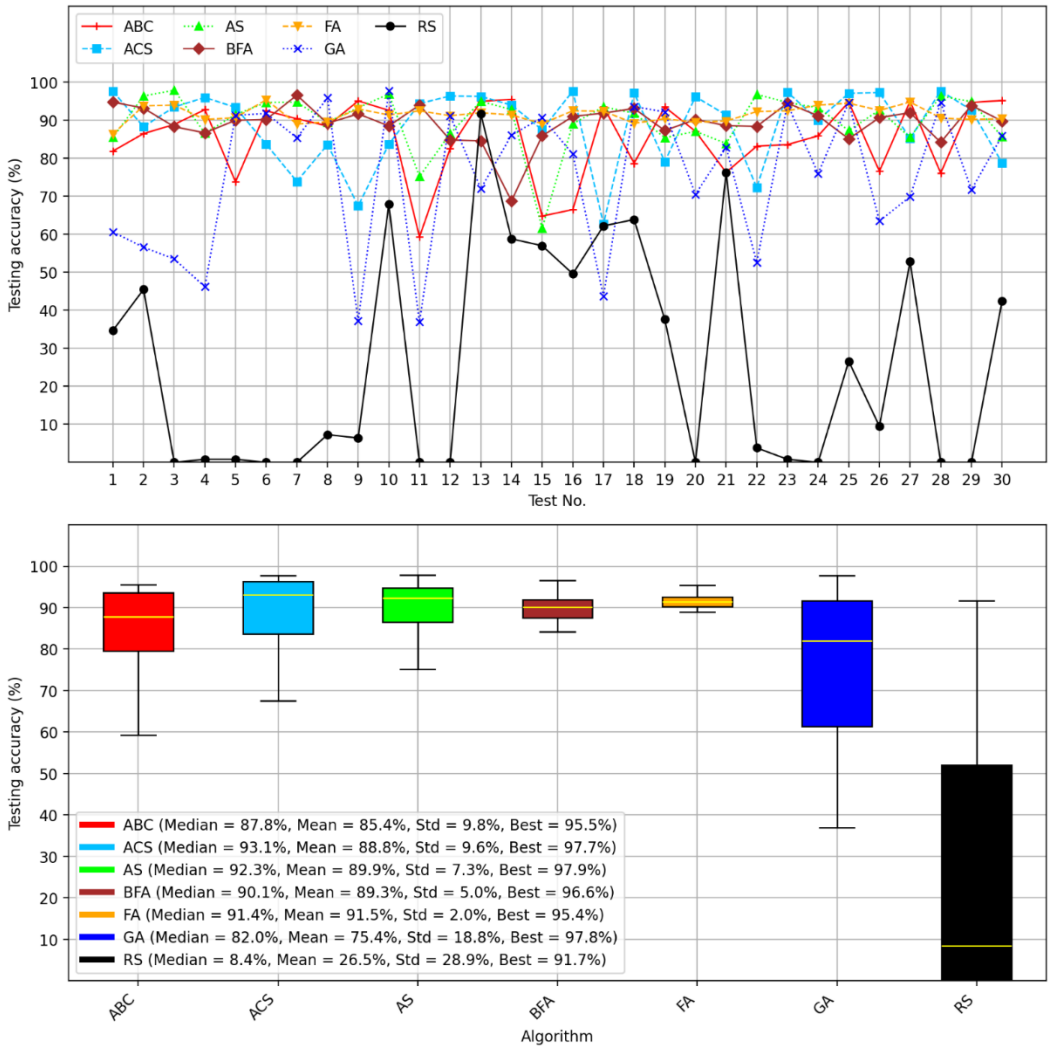


Рисунок 5.29 – Точность тестирования каждого алгоритма

- **Случайный поиск:**

- Диаграммы точности обучения и тестирования ясно демонстрируют явное превосходство алгоритмов роевого интеллекта по сравнению с методом случайного поиска. Согласно данным диаграммам, медиана точности для случайного поиска составляет всего 8.4%, а стандартное отклонение очень велико (29%), что указывает на низкую точность и значительные колебания результатов RS, практически исключая возможность эффективного поиска решений в пространстве гиперпараметров. В то же время алгоритмы роевого интеллекта, такие как ACS, AS, BFA и FA, достигают медианной точности выше 90% с существенно меньшим стандартным отклонением.

- Превосходство этих алгоритмов объясняется их способностью использовать информацию, полученную на предыдущих этапах поиска, для более точного направления процесса оптимизации в потенциально перспективные области пространства гиперпараметров. Эти алгоритмы применяют умные стратегии поиска и адаптации, что позволяет им не только находить лучшие решения, но и делать это более последовательно, снижая разброс результатов. В то время как случайный поиск полагается лишь на случайный перебор комбинаций гиперпараметров, без какого-либо механизма обучения, что приводит к низкой эффективности и нестабильности результатов.

- **Алгоритм искусственной пчелиной колонии:**

- ABC имеет медиану точности на обучающих данных 88.4% и на тестовых данных 87.8%, что показывает способность этого алгоритма эффективно искать гиперпараметры, хотя и не так сильно, как некоторые другие алгоритмы. Среднее значение точности для ABC на обоих наборах данных немного ниже медианы (85.9% на обучающих данных и 85.4% на тестовых данных), а стандартное отклонение довольно велико (10.1% на обучающих данных и 9.8% на тестовых данных). Форма графика показывает, что результаты ABC имеют широкое распределение, с длинными усами (whiskers), что указывает на значительные колебания в производительности между запусками, хотя алгоритм способен находить хорошие решения.

- Учитывая небольшой размер популяции и ограниченное количество итераций, ABC может испытывать трудности с тщательным исследованием пространства поиска. Алгоритм ABC в значительной степени полагается на исследование в своих ранних итерациях, и при всего 10 итерациях алгоритм может не иметь достаточно времени для эффективного перехода от исследования к эксплуатации. В результате ABC может находить неоптимальные решения или застревать в локальных оптимумах. Эти ограничения усугубляются стохастической природой поведения пчел-разведчиков, что также может

приводить к значительным различиям в результатах.

- **Система муравьиной колонии:**

- ACS выделяется высокой медианой точности, достигнув 93.8% на обучающих данных и 93.1% на тестовых данных. Среднее значение точности для ACS также очень близко к медиане (89.4% на обучающих данных и 88.8% на тестовых данных), а стандартное отклонение довольно велико (9.8% на обоих наборах данных), что показывает, что этот алгоритм достаточно эффективен, но не стабилен. Форма графика показывает, что результаты сосредоточены вокруг медианы с короткими усами, что свидетельствует о последовательности в нахождении оптимальных гиперпараметров.

- ACS демонстрирует надежную производительность с относительно высокой и стабильной точностью как на обучающих, так и на тестовых данных. Это говорит о том, что в ACS эффективно поддерживается баланс между эксплуатацией (использованием известных хороших решений) и разведкой (исследованием новых решений). Включение параметра q_0, который определяет относительную важность эксплуатации по сравнению с разведкой, похоже, работает правильно. Однако изменчивость в точности обучения и тестирования, включая некоторые более низкие значения, может указывать на то, что, хотя алгоритм в целом устойчив, иногда он может быть подвержен тем же проблемам, что и GA и AS, например, преждевременной сходимости или недостаточному выходу из локальных оптимумов.

- **Муравьиная система**:

- AS также показывает отличные результаты, с медианой точности 92.7% на обучающих данных и 92.3% на тестовых данных. Среднее значение точности для AS (90.5% на обучающих данных и 89.9% на тестовых данных) близко к медиане, а стандартное отклонение довольно низкое (7.4% на обучающих данных и 7.3% на тестовых данных), что показывает относительно хорошую стабильность алгоритма. Ящиковая диаграмма для AS аналогична ACS, с

короткими усами и сосредоточением данных вокруг медианы, что доказывает, что AS является надежным и последовательным выбором.

- Несколько лучшие показатели производительности по сравнению с ACS делают AS надежным алгоритмом для достижения высокой и стабильной точности в данной задаче. Для AS реализация одинаковой эвристики для всех гиперпараметров означает, что начальные решения муравьев не будут смещены в сторону каких-либо конкретных значений гиперпараметров. Следовательно, точность обучения и тестирования сильно зависит от информации о феромонах. Высокая средняя точность указывает на то, что следы феромонов эффективно направляют муравьев к правильным решениям. Однако изменчивость точности, особенно низкие минимальные значения, предполагает, что в некоторых случаях использование феромонов может приводить к сходимости к неоптимальным решениям.

- **Алгоритм стаи птиц:**

- BFA демонстрирует медиану точности 90.7% на обучающих данных и 90.1% на тестовых данных. Средние значения также близки к медиане (89.8% на обучающих данных и 89.3% на тестовых данных), а стандартное отклонение очень низкое (5.1% на обучающих данных и 5.0% на тестовых данных), что свидетельствует о высокой стабильности BFA. Ящиковая диаграмма показывает, что результаты сосредоточены вокруг медианы с короткими усами, что отражает высокую согласованность и стабильность результатов BFA в различных запусках.

- Это объясняется относительно быстрой скоростью сходимости, которую обеспечивают когнитивные (c_1) и социальные (c_2) компоненты. Эти параметры направляют птиц, объединяя личный опыт и коллективные знания стаи, что позволяет более эффективно исследовать и эксплуатировать пространство поиска. Однако возникновение локальных оптимумов и случайная неконвергенция могут быть вызваны недостаточным количеством итераций или

весом инерции (w), который не всегда обеспечивает идеальный баланс между исследованием и эксплуатацией, что приводит к различиям в производительности между различными запусками.

- **Алгоритм светлячков:**

- FA демонстрирует высокую медиану точности на обучающих и тестовых данных, достигнув 92.2% и 91.4% соответственно. Среднее значение точности для FA (91.5% на обучении и 91.4% на тестировании) практически совпадает с медианой, а стандартное отклонение очень низкое (2.2% на обучении и 2.0% на тестировании), что свидетельствует о высокой стабильности FA. Ящиковая диаграмма с короткими усами показывает, что результаты сосредоточены вокруг медианы, что отражает высокую степень согласованности и надежности этого алгоритма в нахождении оптимальных гиперпараметров.

- Высокие стабильность и точность объясняются комплексным механизмом поиска в FA, который балансирует между исследованием и эксплуатацией. В каждом цикле каждое решение, соответствующее каждому светлячку, сравнивается с решениями всех остальных светлячков, стремясь найти лучшее решение на основе их привлекательности. Такое тщательное исследование и эксплуатация пространства поиска приводят к консистентной сходимости к решениям, близким к оптимальным, что снижает изменчивость между запусками.

- **Генетический алгоритм**

- GA имеет самую низкую медиану точности среди алгоритмов роевого интеллекта, с 82.6% на обучении и 82.0% на тестировании. Среднее значение точности для GA (75.8% на обучающих данных и 75.4% на тестовых данных) значительно ниже медианы, а стандартное отклонение очень велико (18.9% на обучении и 18.8% на тестировании), что свидетельствует о значительной вариативности и недостаточной стабильности результатов GA. Ящиковая диаграмма с длинными усами по сравнению с другими алгоритмами отражает

большую дисперсию результатов и непоследовательность эффективности GA.

- Широкий диапазон точности от самой низкой до самой высокой указывает на то, что, хотя в некоторых случаях были найдены почти оптимальные решения, в других случаях алгоритм мог оказаться в ловушке локального оптимума. Турнирный отбор мог вызвать сильное селекционное давление, которое эффективно подталкивает популяцию к конвергенции, но также может слишком быстро уменьшить генетическое разнообразие, не позволяя алгоритму исследовать все пространство решений. Кроме того, GA часто требует большого размера популяции и количества итераций для достижения оптимальных результатов.

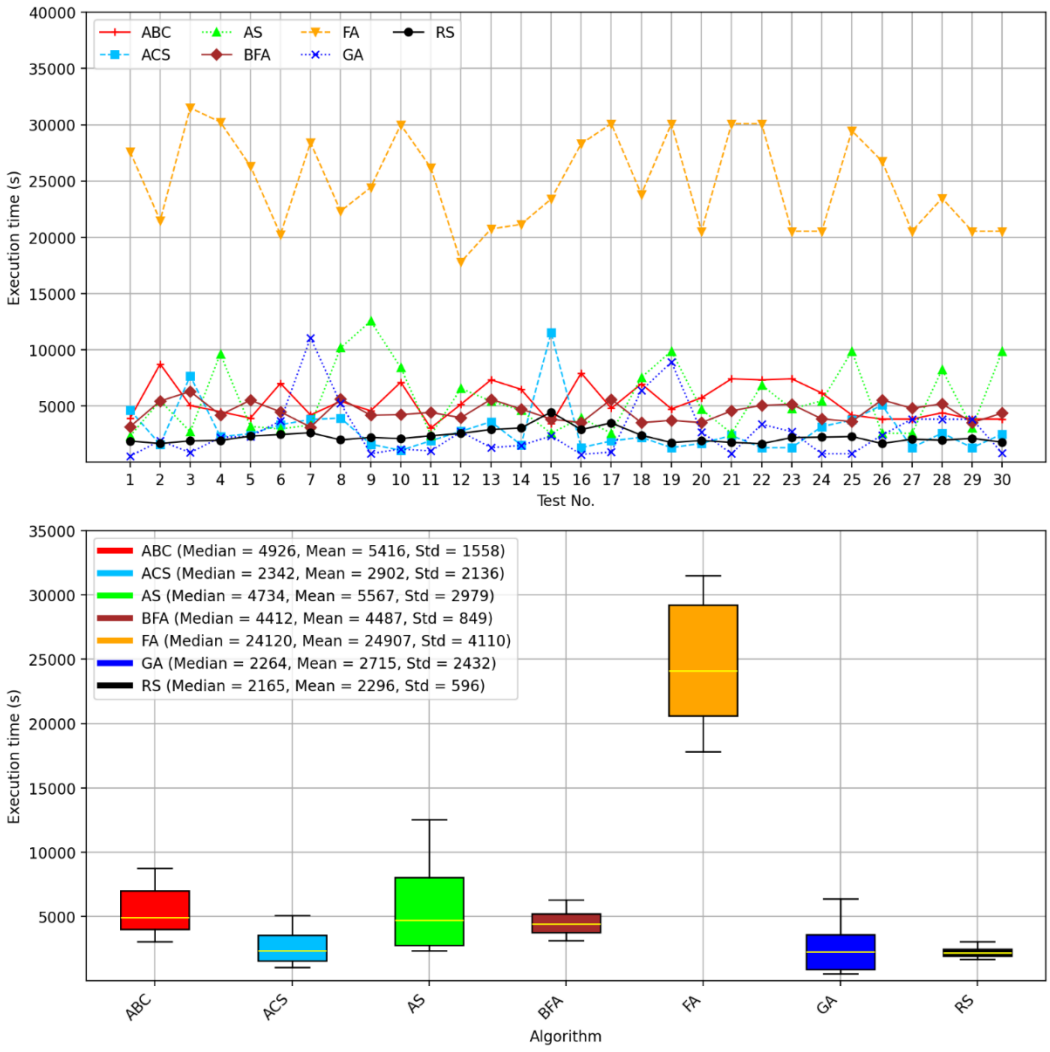


Рисунок 5.30 – Время выполнения каждого алгоритма

- **Случайный поиск:**

- RS выделяется самым коротким временем выполнения среди всех алгоритмов, что делает его эффективным с точки зрения времени. Медиана составляет 2165 секунд, среднее значение 2296 секунд, а стандартное отклонение 596 секунд, что указывает на относительно стабильное выполнение.

- Однако, несмотря на быстрое выполнение, RS обычно показывает более низкую производительность по сравнению с алгоритмами роевого интеллекта в поиске оптимальных гиперпараметров. Поэтому RS может быть хорошим выбором в ситуациях, требующих быстрых экспериментов, но не обеспечивает оптимальную эффективность, как более сложные алгоритмы.

- **Алгоритм искусственной пчелиной колонии:**

- ABC имеет медиану времени выполнения 4926 секунд, среднее значение 5416 секунд и стандартное отклонение 1558 секунд. Это показывает, что время выполнения ABC довольно стабильное, хотя и имеет определенное расхождение, что отражается в довольно высоком стандартном отклонении.

- Это можно объяснить медленной сходимостью алгоритма на ранних стадиях, особенно при ограниченном числе итераций, когда ABC больше фокусируется на исследовании пространства решений, а не на его использовании. В результате сохраняется высокое разнообразие популяции, и разные наборы гиперпараметров могут приводить к значительно разному времени обучения, что вызывает значительные различия во времени выполнения между различными запусками.

- **Система муравьиной колонии:**

- ACS является одним из алгоритмов с самым низким временем выполнения, медиана составляет 2342 секунды, среднее значение 2902 секунды. Стандартное отклонение составляет 2136 секунд, что указывает на некоторые колебания времени выполнения между различными запусками. Несмотря на это, ACS демонстрирует высокую эффективность по времени, что делает его

привлекательным выбором при учете соотношения между производительностью и временем вычислений.

- ACS исследует пространство гиперпараметров с помощью механизма обновления феромонов, управляемого параметром q_0 что позволяет алгоритму быстро находить оптимальное решение, избегая чрезмерной эксплуатации бесперспективных областей. Таким образом, несмотря на разное время выполнения, ACS обычно эффективно сходится к подходящим наборам гиперпараметров, избегая длительных исследований, характерных для AS, или быстрой, но потенциально преждевременной сходимости для GA.

- **Муравьиная система**:

- AS имеет медиану времени выполнения 4734 секунды, среднее значение 5567 секунд и довольно высокое стандартное отклонение 2979 секунд. Это указывает на значительные колебания времени выполнения между запусками. Хотя производительность AS на графиках точности довольно хорошая, нестабильное время выполнения может стать фактором, который следует учитывать при его использовании на практике.

- Поскольку эвристика одинакова для всех гиперпараметров, поиск каждым муравьем в AS представляет собой более исчерпывающее исследование пространства гиперпараметров, что требует больше времени для того, чтобы следы феромонов развили достаточно различий для эффективного направления муравьев. Это означает, что алгоритму может потребоваться обучение на большом количестве наборов гиперпараметров, прежде чем уровни феромонов будут скорректированы таким образом, чтобы выделить наиболее перспективные области в пространстве поиска, что увеличивает время вычислений для процесса оптимизации.

- **Алгоритм стаи птиц:**

- BFA имеет медиану времени выполнения 4412 секунд, среднее значение 4487 секунд и стандартное отклонение всего 849 секунд. Это

показывает, что BFA имеет очень стабильное время выполнения с небольшими колебаниями между запусками. BFA сочетает в себе стабильность времени выполнения и высокую производительность, что делает его хорошо сбалансированным выбором между этими двумя факторами.

- Стабильность времени выполнения может быть обусловлена быстрой сходимостью алгоритма, даже при небольшом размере популяции и ограниченном количестве итераций, что позволяет алгоритму быстро находить оптимальное или близкое к оптимальному решение. Кроме того, сохраняя найденные решения, BFA может сократить время, необходимое для последующих оценок.

- **Алгоритм светлячков:**

- A имеет самое длительное время выполнения среди всех сравниваемых алгоритмов, с медианой до 24120 секунд, средним значением 24907 секунд и стандартным отклонением 4110 секунд. Это показывает, что FA требует много вычислительных ресурсов и времени для завершения вычислений. Колебания времени выполнения также довольно значительные, что может стать ограничивающим фактором при использовании FA в задачах, требующих быстрой оптимизации.

- Такое длительное время выполнения главным образом связано с тем, что на каждой итерации алгоритм оценивает гораздо больше потенциальных решений и делает это более подробно, чем других алгоритмов. Кроме того, большая разница во времени выполнения обусловлена значительной разницей во времени обучения моделей нейронных сетей.

- **Генетический алгоритм**

- GA имеет медиану времени выполнения 2264 секунды, среднее значение 2715 секунд и стандартное отклонение 2432 секунды. Это говорит о том, что GA в целом является самым быстрым среди всех алгоритмов, хотя он

также демонстрирует значительную изменчивость во времени выполнения между различными запусками.

- Относительно низкое среднее время выполнения GA делает его привлекательным вариантом для сценариев, в которых приоритетом является скорость. Однако значительное стандартное отклонение указывает на непоследовательность, при этом некоторые запуски занимают значительно больше времени, чем другие. Несмотря на свою скорость, высокая изменчивость времени выполнения GA и более низкая средняя точность предполагают, что он может быть менее эффективным для поиска оптимальных значений гиперпараметров.

Наилучшая архитектура модели нейронной сети, полученная каждым алгоритмом, приведена в таблице 5.6. Отсутствие нейронов в скрытом слое (обозначено символом «-») означает, что этот слой отсутствует в модели.

Таблица 5.6 – Наилучшие архитектуры модели нейронной сети

Алгоритм	**Слой 1, Dropout**	**Слой 2, Dropout**	**Слой 3, Dropout**	**Слой 4, Dropout**	**Слой 5, Dropout**	**Тип ячейки**	**Точность обучения (%)**	**Точность тестирования (%)**
ABC	180, 0.2	120, 0.2	40, 0.3	120, 0.2	-	GRU	96.3	95.5
ACS	190, 0.0	190, 0.0	190, 0.0	190, 0.0	190, 0.3	GRU	98.6	97.7
AS	-	180, 0.0	180, 0.0	180, 0.5	180, 0.5	LSTM	98.8	97.9
BFA	-	130, 0.0	150, 0.2	120, 0.2	110, 0.1	GRU	97.4	96.6
FA	170, 0.1	-	-	150, 0.2	70, 0.1	LSTM	96.9	95.4
GA	180, 0.0	170, 0.0	80, 0.0	190, 0.0	190, 0.0	LSTM	98.7	97.8

Ячейки LSTM и GRU обеспечивают более высокую точность по сравнению с SimpleRNN. Это указывает на то, что дополнительная сложность LSTM и GRU, способных фиксировать долгосрочные зависимости, полезна для повышения производительности модели.

В некоторых случаях используются не все пять слоев (обозначенные знаком «-»), что означает, что предлагаемый метод позволяет упростить модель без значительной потери производительности. Такая адаптивная сложность может помочь предотвратить переобучение, сохраняя при этом достаточную емкость модели.

Аналогично, максимальное количество нейронов на слой установлено на уровне 200, однако данные в таблице показывают, что алгоритмы выбирают меньшее количество нейронов на слой. Это свидетельствует о том, что предложенные методы находят оптимальный баланс между необходимостью иметь достаточное количество нейронов для отражения сложности данных и избеганием их чрезмерного увеличения, что может привести к переобучению и ненужной вычислительной нагрузке.

Предложенный метод балансировки сложности и точности модели оказался успешным. Он способен упростить архитектуру модели, регулируя количество активных слоев и выборочно определяя количество нейронов, обеспечивая при этом высокую точность, что свидетельствует об эффективности предложенного подхода в достижении баланса между сложностью и производительностью модели.

Ниже представлены графики, отражающие результаты процесса обучения и тестирования модели с заранее оптимизированным набором гиперпараметров.

На рисунке 5.31 показана точность и значение функции потерь модели по эпохам в процессе обучения с включенным и отключенным Dropout.

На рисунке 5.32 представлена точность модели для каждого пути, при этом общая точность определяется как правильное предсказание всех кратчайших путей.

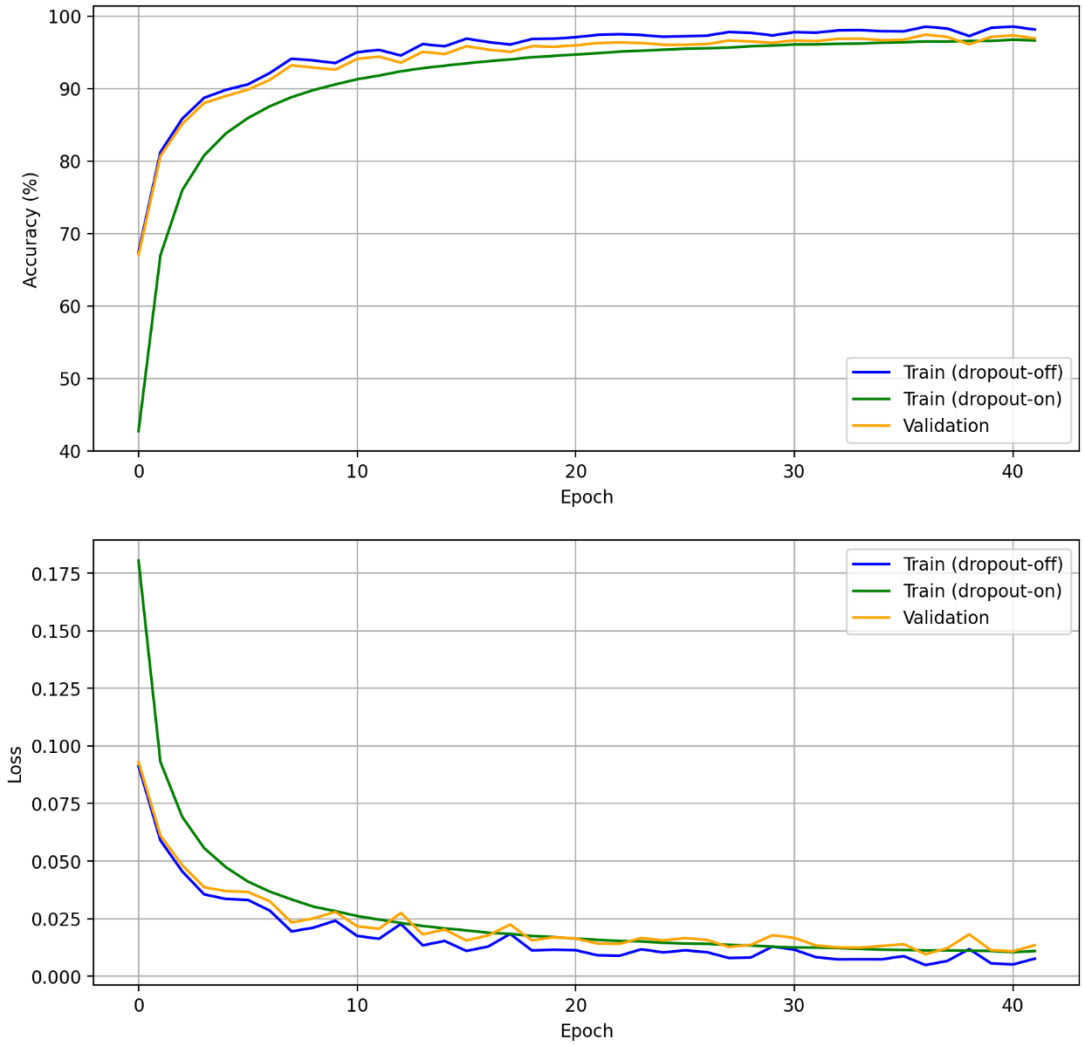


Рисунок 5.31 – Точность и значение функции потерь модели по эпохам

- Точность и функция потерь на обучающем наборе (Dropout-on)

- Точность: Зеленая линия представляет точность модели при применении техники dropout во время обучения. Точность начинается с около 43% и быстро увеличивается в первые 10 эпох. К 20-й эпохе точность достигает 95% и продолжает медленно расти, достигая 98% к концу процесса обучения. Хотя использование dropout временно снижает производительность модели в процессе обучения, это помогает избежать переобучения за счет случайного «отключения» некоторых нейронов на каждом шаге обучения. Это объясняет, почему точность на обучающем наборе при включённом dropout обычно ниже, чем при его отключении (dropout-off).

- Функция потерь: Зеленая линия показывает, что функция потерь в начале довольно высокая – около 0.175 после первой эпохи, но значительно уменьшается в течение первых 10 эпох. После 20-й эпохи функция потерь уменьшается до очень низкого уровня (почти 0), что указывает на то, что модель хорошо сходится.

- Точность и функция потерь на обучающем наборе (Dropout-off)

- Точность: Синяя линия отображает точность при отключенном dropout в процессе предсказания (хотя модель все еще обучается с dropout). В первые эпохи синяя линия почти совпадает с точностью на проверочном наборе (оранжевая линия). В последующих эпохах точность на обучающем наборе без dropout немного выше, чем на проверочном наборе, так как модель обучается непосредственно на этих данных.

- Функция потерь: Функция потерь на обучающем наборе без dropout уменьшается аналогично процессу с dropout, однако снижение происходит быстрее и сильнее, поскольку все нейроны используются в процессе предсказания.

- Точность и функция потерь на проверочном наборе (Dropout-off)

- Точность: Оранжевая линия отображает точность модели на проверочном наборе, где dropout не применяется в процессе предсказания. Точность начинается с 67% после первой эпохи и постепенно увеличивается в последующие эпохи. К 20-й эпохе точность на проверочном наборе достигает 95%, а к 40-й – 97%, почти соответствуя точности на обучающем наборе. Это показывает, что модель хорошо обучилась и способна обобщать на невидимых данных, что доказывает эффективность применения dropout.

- Функция потерь: Оранжевая линия отображает функцию потерь на проверочном наборе, которая начинается с 0.075 после первой эпохи и быстро снижается в последующих эпохах. К 20-й эпохе функция потерь стабилизируется

на очень низком уровне, что указывает на отсутствие переобучения и хорошую обобщающую способность модели на проверочном наборе.

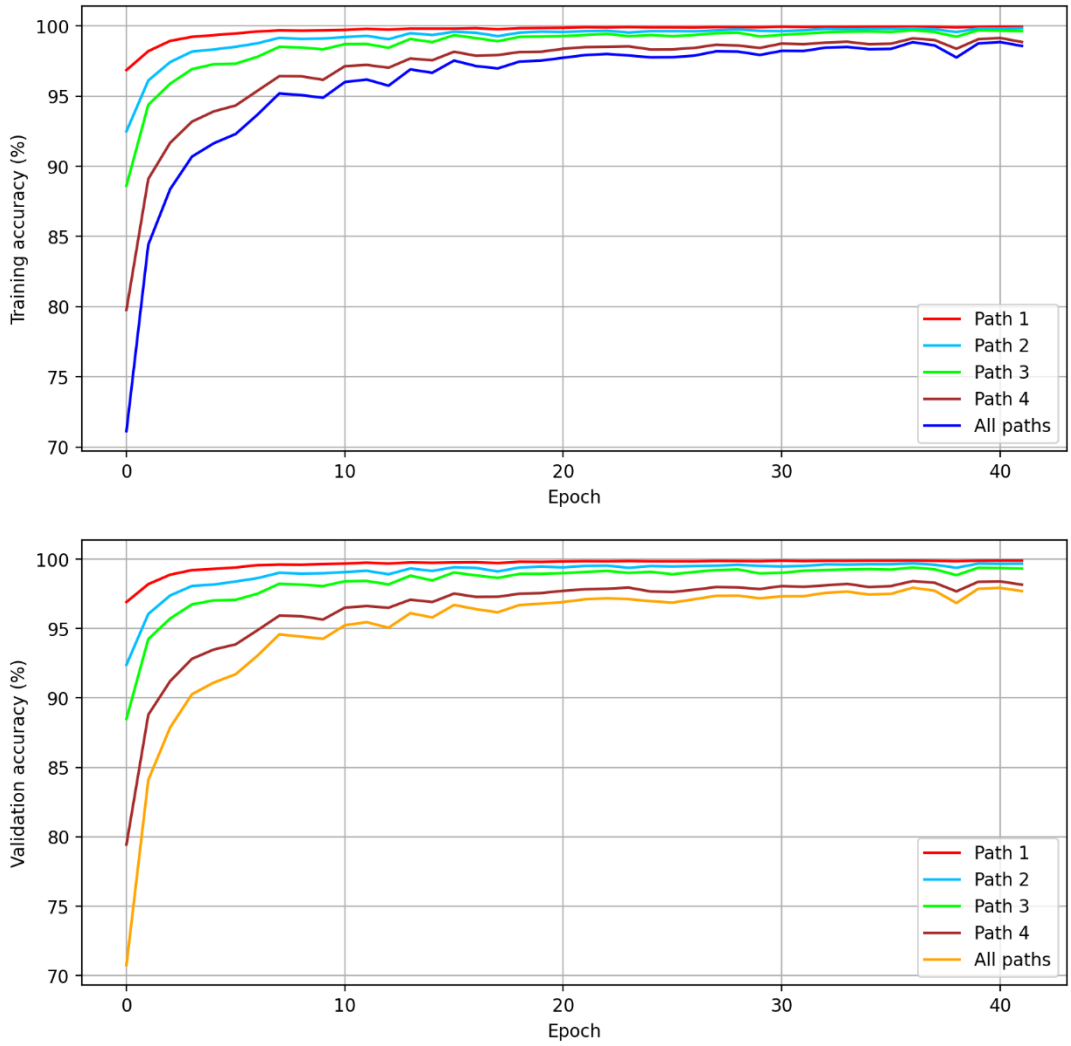


Рисунок 5.32 – Точность модели для каждого пути

- Путь 1: Точность предсказаний для пути 1 очень высока с самого начала, и модель достигает почти 100% точности примерно к 15-ой эпохе обучения. Это указывает на то, что путь 1 является самым простым для предсказания из всех возможных путей. Вероятно, это связано с тем, что путь 1 является самым коротким и наименее сложным по своей структуре, что позволяет модели быстро обучаться и оптимизировать предсказания.
- Путь 2 и Путь 3: Точность предсказаний для пути 2 и пути 3 увеличивается очень быстро, достигая уровня выше 95% уже после 10 эпох и почти приближаясь к

100% к концу обучения. Это говорит о том, что оба пути также являются легкими для обучения, хотя путь 2 и путь 3 могут быть немного сложнее, чем путь 1. Тем не менее, модель успешно обучается и делает точные предсказания для этих путей.

- Путь 4: Для этого пути модель имеет самую низкую точность предсказаний в процессе обучения. Хотя точность предсказаний для пути 4 постоянно увеличивается и превышает 95%, она не достигает почти 100%, как для других путей. Это указывает на то, что путь 4 является самым сложным из всех путей, возможно, из-за его большей длины, большего количества разветвлений или трудностей в распознавании его особенностей, что затрудняет процесс обучения модели и делает его менее эффективным для этого пути.
- Все пути: Прогнозирование модели считается точным, если все необходимые пути предсказаны правильно одновременно. Общая точность ниже по сравнению с прогнозированием отдельных путей, что вполне логично, поскольку ошибка в предсказании хотя бы одного пути снижает общую точность.

Причиной более низкой точности предсказаний для последующих путей по сравнению с предыдущими путями могут быть два основных фактора:

- Более высокая сложность путей: Последующие пути (Путь 3 и Путь 4) обычно имеют более высокую сложность: они могут быть длиннее или проходить через большее количество узлов и связей по сравнению с предыдущими путями (Путь 1 и Путь 2). Это приводит к снижению точности предсказаний модели, особенно если на предыдущих шагах предсказания были не совсем точными. Чем больше сложных узлов и разветвлений содержит путь, тем сложнее для модели корректно определить его структуру.
- Особенности рекуррентной нейронной сети: RNN обладает свойством, при котором последующие шаги зависят от предыдущих. Это может снижать точность предсказаний для последующих путей, если модель сталкивается с проблемой «накопления ошибок» (когда ошибки предыдущих шагов усиливаются в

последующих) или если информация передается по шагам недостаточно эффективно. Если модель не может удерживать достаточное количество информации из предыдущих шагов, это приводит к снижению точности на последних этапах предсказаний, особенно для более сложных путей.

Таким образом, разработанная нейронная сеть доказала свою эффективность и достигла высокой точности в прогнозировании кратчайших путей в сети. Это можно объяснить следующими причинами:

- Способность RNN обучаться на последовательных данных: Рекуррентные нейронные сети хорошо обрабатывают последовательные данные, где последующие прогнозы зависят от предыдущих. Это является важным преимуществом в задаче поиска K кратчайших путей, так как последующие пути зависят от информации о предыдущих (по аналогии с алгоритмом Йена). RNN способны запоминать предыдущие пути, что помогает модели корректно предсказывать последующие пути.
- Моделирование вывода в виде бинарного вектора: Моделирование путей в виде бинарного вектора (каждый элемент которого указывает на принадлежность ребра к пути) упрощает обучение модели и оптимизацию выбора ребер для каждого пути.
- Эффективная обработка весов каналов: Модель использует веса каналов в сети в качестве входных данных, что позволяет ей оптимизировать прогнозирование на основе сетевых характеристик. Это помогает модели легко различать каналы и выбирать лучшие из них для построения кратчайшего пути в ПКС.

5.3 ИССЛЕДОВАНИЕ ПРОЦЕССОВ БАЛАНСИРОВКИ ПОТОКОВ ДАННЫХ В ПКС

Для исследования метода динамической адаптивной многопутевой балансировки потоков данных в ПКС (*DAMLB*) рассматривалась топология ПКС (рисунок 5.33).

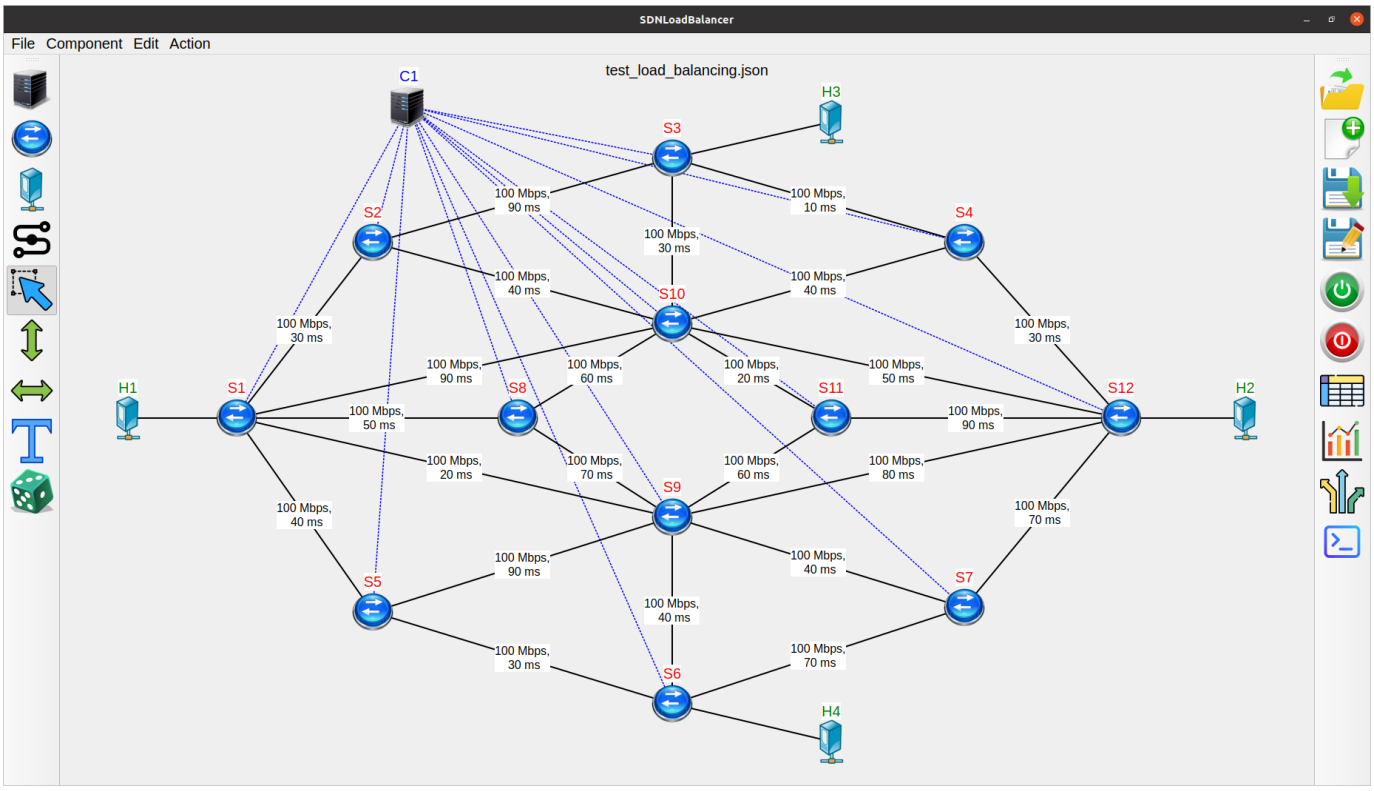


Рисунок 5.33 – Экспериментальная топология ПКС из 12 узлов

В данном эксперименте используется топология ПКС, которая включает 12 коммутаторов и 22 канала передачи данных между ними (не считая каналов к хостам). Каждый канал имеет ограничение пропускной способности 100 Мбит/с.

Сетевые хосты H1, H2, H3 и H4 подключены к коммутаторам S1, S12, S3 и S6 соответственно. Хосты H1 и H3 играют роль клиентов, а H2 и H4 выполняют роль серверов. Клиенты будут генерировать трафик, который передается через сеть, и задача состоит в том, чтобы оценить, как метод многопутевой балансировки нагрузки распределяет этот трафик между различными путями в сети.

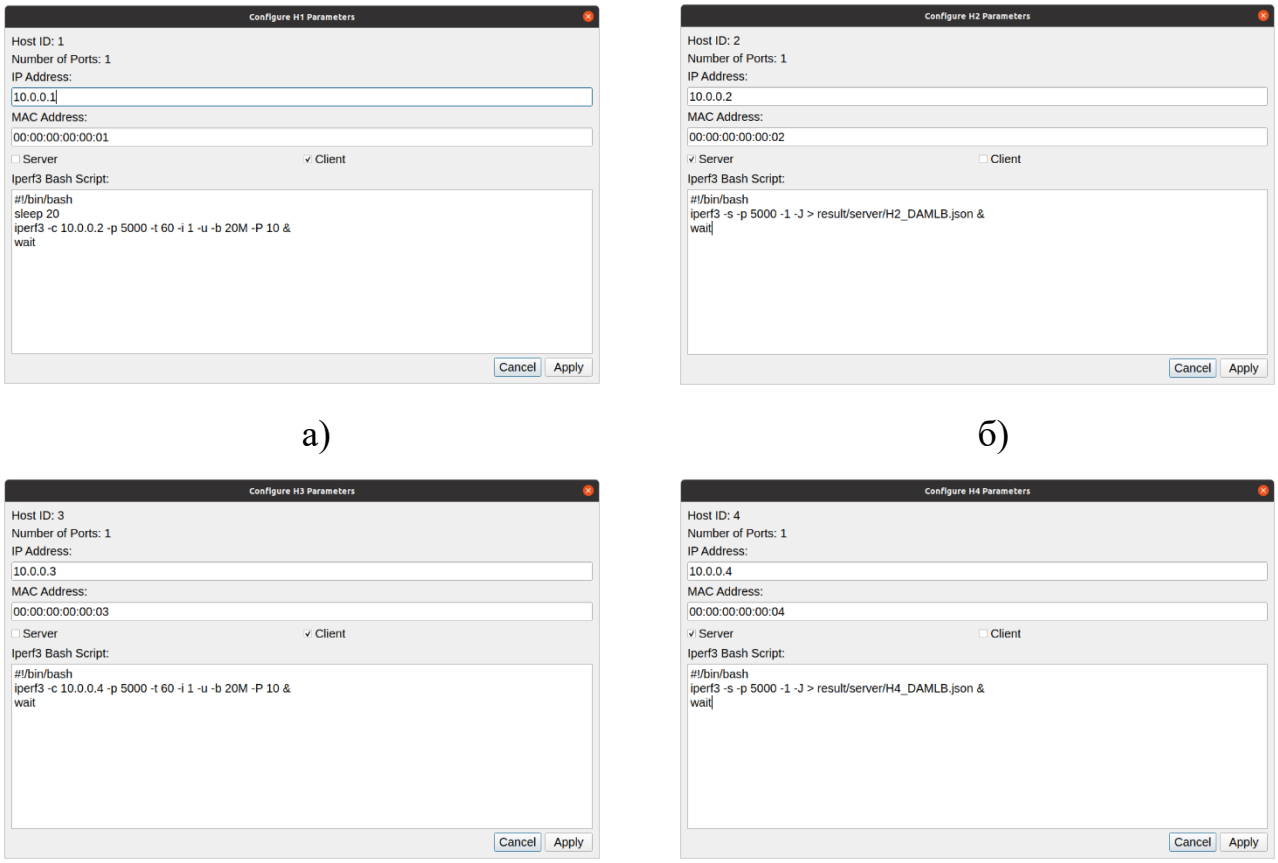
Порядок проведения эксперимента

- Хост H3 начинает передачу данных на сервер H4 через коммутатор S3. Трафик создается с использованием iperf3, которая генерирует 10 параллельных потоков, каждый с пропускной способностью 20 Мбит/с. Передача длится 60 секунд.
- Через 20 секунд после начала передачи от H3 к H4, хост H1 начинает передавать данные на сервер H2 через коммутатор S1. Параметры трафика

аналогичны: 10 параллельных потоков, каждый с пропускной способностью 20 Мбит/с, и продолжительность передачи составляет 60 секунд.

- На серверах H2 и H4 собираются результаты эксперимента, которые записываются в формате JSON для последующего анализа.

Параметры настройки хостов H1, H2, H3 и H4, а также скрипты для генерации трафика с помощью iperf3, показаны на следующих рисунках:



а) б)

в) г)

Рисунок 5.34 – Настройка хостов:

а, б – Конфигурация клиентов H1 и H3; в, г – Конфигурация серверов H2 и H4

- Для выполнения различных методов балансировки потоков данных, необходимо выбрать соответствующий сценарий контроллера Ryu в настройках контроллера C1.

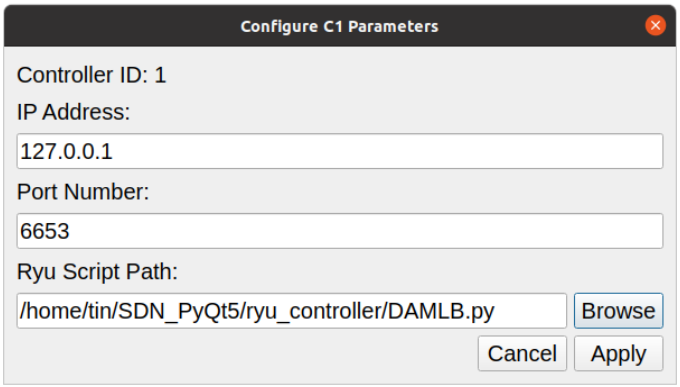
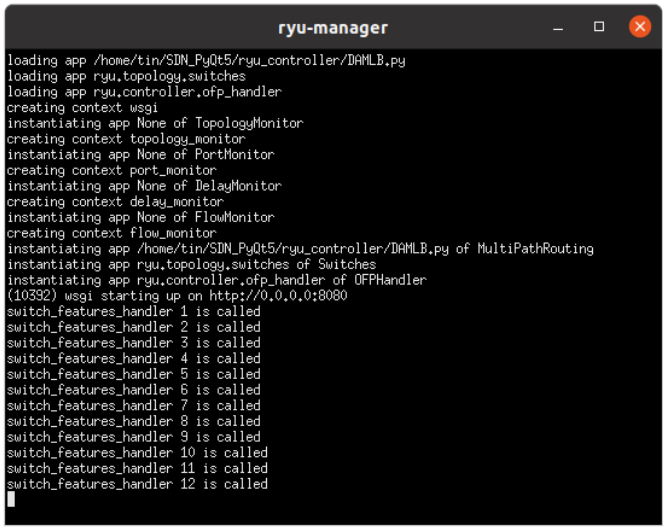


Рисунок 5.35 – Настройка контроллера

- После завершения настройки, выполняется запуск эксперимента. Для этого необходимо последовательно запустить контроллер Ryu и эмулятор сети Mininet, используя предварительно заданные конфигурации сети.



а) б)

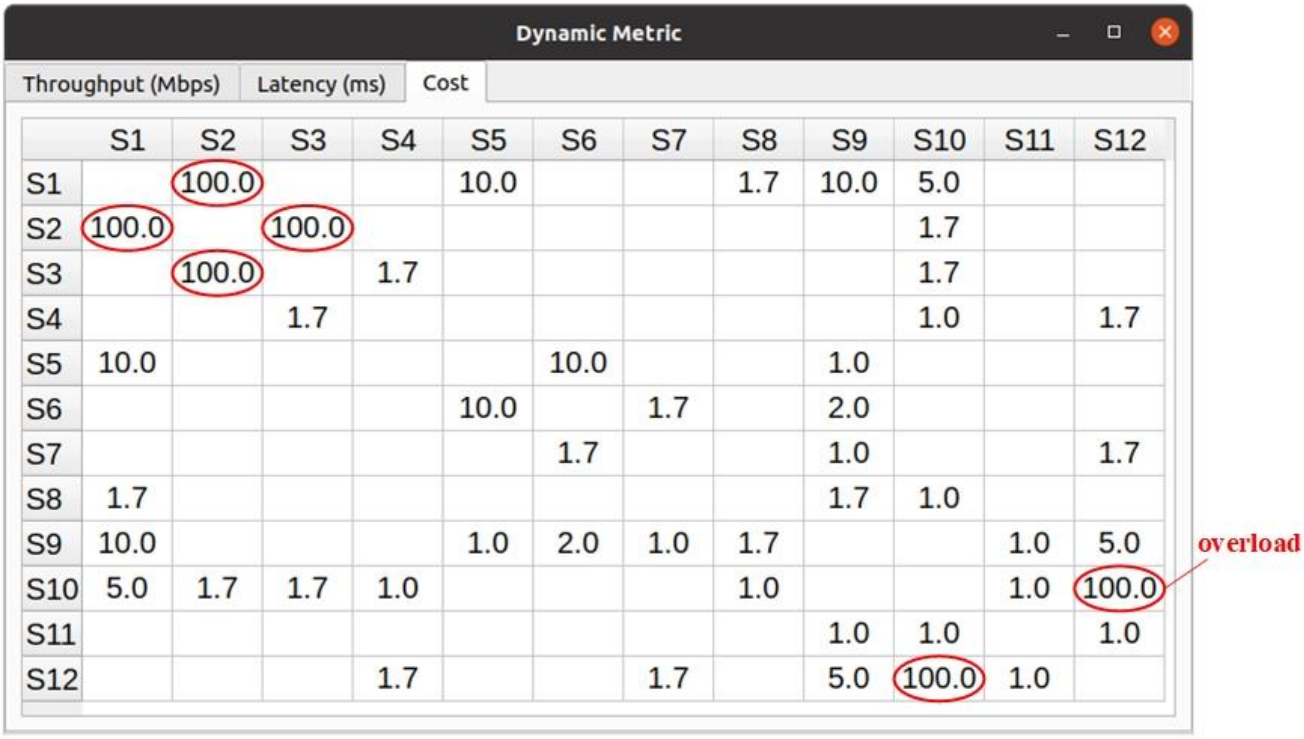
Рисунок 5.36 – Запуск эксперимента:

а – Запуск контроллера Ryu; б – Запуск эмулятора Mininet

Анализ результатов эксперимента

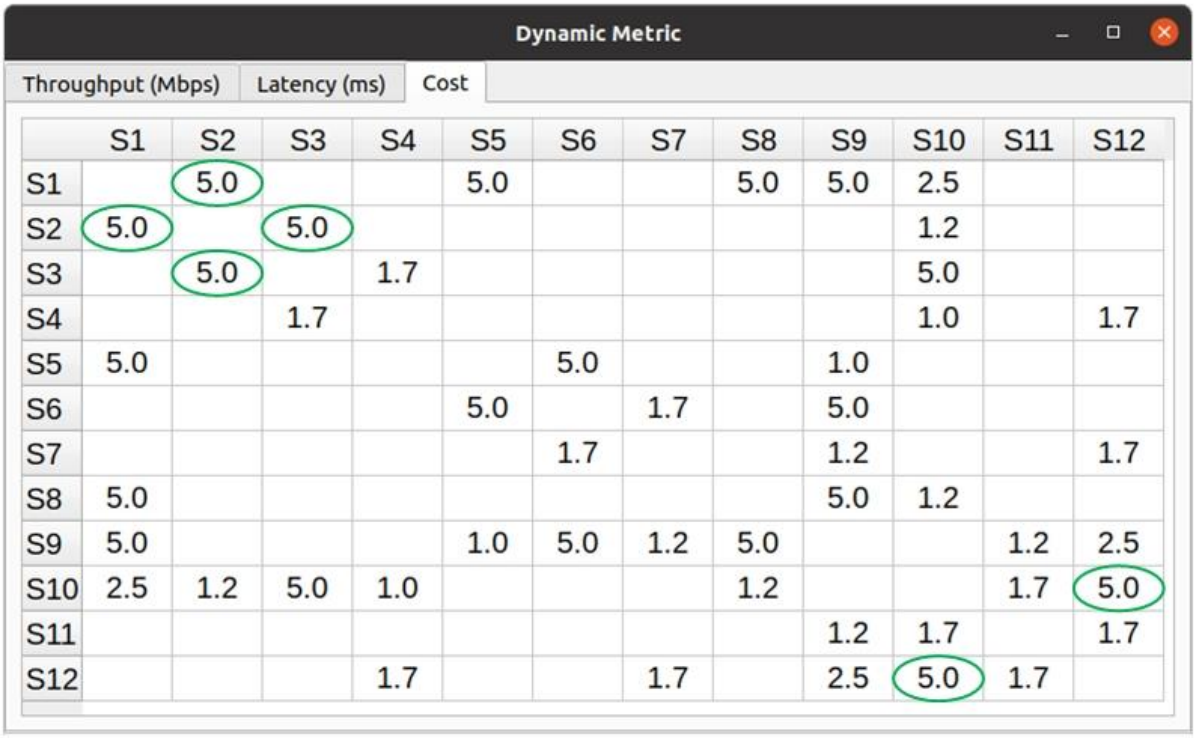
Для оценки эффективности предложенного метода динамической адаптивной многопутевой балансировки потоков данных была проведена серия измерений

стоимости каналов до и после его применения. Результаты сравнения представлены на следующих рисунках.



	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11	S12
S1		100.0			10.0			1.7	10.0	5.0		
S2	100.0		100.0							1.7		
S3		100.0		1.7						1.7		
S4			1.7							1.0		1.7
S5	10.0					10.0			1.0			
S6					10.0		1.7		2.0			
S7						1.7			1.0			1.7
S8	1.7								1.7	1.0		
S9	10.0				1.0	2.0	1.0	1.7			1.0	5.0
S10	5.0	1.7	1.7	1.0				1.0			1.0	100.0
S11									1.0	1.0		1.0
S12				1.7			1.7		5.0	100.0	1.0	

Рисунок 5.37 – Стоимость каналов до применения метода DAMLB



	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11	S12
S1		5.0			5.0			5.0	5.0	2.5		
S2	5.0		5.0							1.2		
S3		5.0		1.7						5.0		
S4			1.7							1.0		1.7
S5	5.0					5.0			1.0			
S6					5.0		1.7		5.0			
S7						1.7			1.2			1.7
S8	5.0								5.0	1.2		
S9	5.0				1.0	5.0	1.2	5.0			1.2	2.5
S10	2.5	1.2	5.0	1.0				1.2			1.7	5.0
S11									1.2	1.7		1.7
S12				1.7			1.7		2.5	5.0	1.7	

Рисунок 5.38 – Стоимость каналов после применения метода DAMLB

- **До применения метода DAMLB**: На первом рисунке показана стоимость каналов, где на некоторых каналах наблюдается перегрузка (показано красным), а другие каналы остаются недозагруженными. Например, канал между S1 и S2 работает на грани полной загрузки ($w = 100$), что указывает на неравномерное распределение трафика.
- **После применения метода DAMLB**: На втором рисунке можно увидеть, что после включения механизма балансировки потоков данных, трафик распределяется более равномерно между различными путями. Стоимость каналов значительно улучшилась, и перегрузка была устранена.

Сравнение метода DAMLB с другими методами балансировки нагрузки

Для дальнейшей оценки эффективности предложенного метода DAMLB был проведен сравнительный анализ с двумя известными методами балансировки нагрузки – ECMP (Equal-Cost Multi-Path) и Round Robin [110, 111]. Результаты эксперимента, проведенного с помощью iperf3, были сохранены в формате JSON, что позволило построить графики, иллюстрирующие пропускную способность и процент потерь пакетов на серверах.

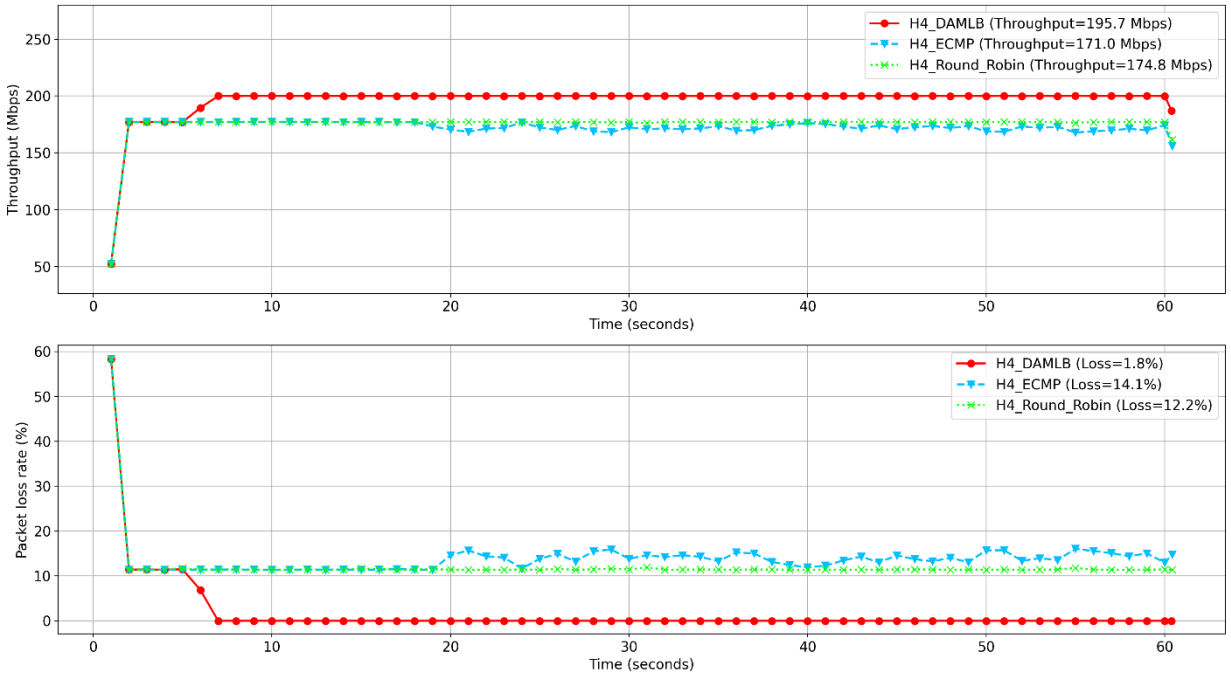


Рисунок 5.39 – Пропускная способность и процент потерь пакетов на H4

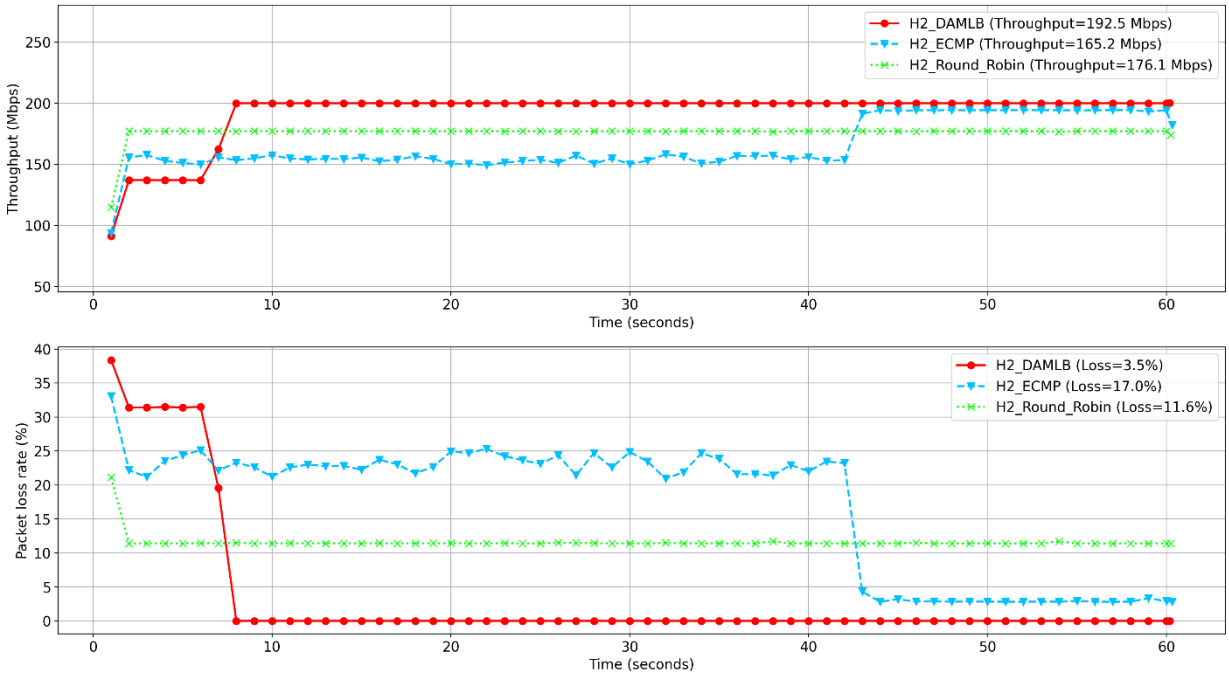


Рисунок 5.40 – Пропускная способность и процент потерь пакетов на H2

- **Пропускная способность (Throughput)**: На верхних графиках показана пропускная способность на серверах H2 и H4. Видно, что метод DAMLB обеспечивает лучшую пропускную способность по сравнению с методами ECMP и Round Robin. Например, на сервере H2 метод DAMLB поддерживает стабильную пропускную способность около 200 Мбит/с, тогда как у ECMP и Round Robin пропускная способность варьируется на более низком уровне.
- **Процент потерь пакетов (Packet loss rate)**: На нижних графиках показаны изменения процентной потери пакетов в течение времени. Здесь также можно заметить преимущество DAMLB: процент потерь пакетов значительно ниже, и после начальной стабилизации он практически сводится к нулю. В то время как методы ECMP и Round Robin показывают более высокий и колеблющийся уровень потерь пакетов в течение всего времени эксперимента.

Анализ индекса балансировки нагрузки (LBI)

На данном этапе эксперимента проводится анализ индекса балансировки нагрузки, который отражает степень равномерного распределения трафика между доступными каналами сети.

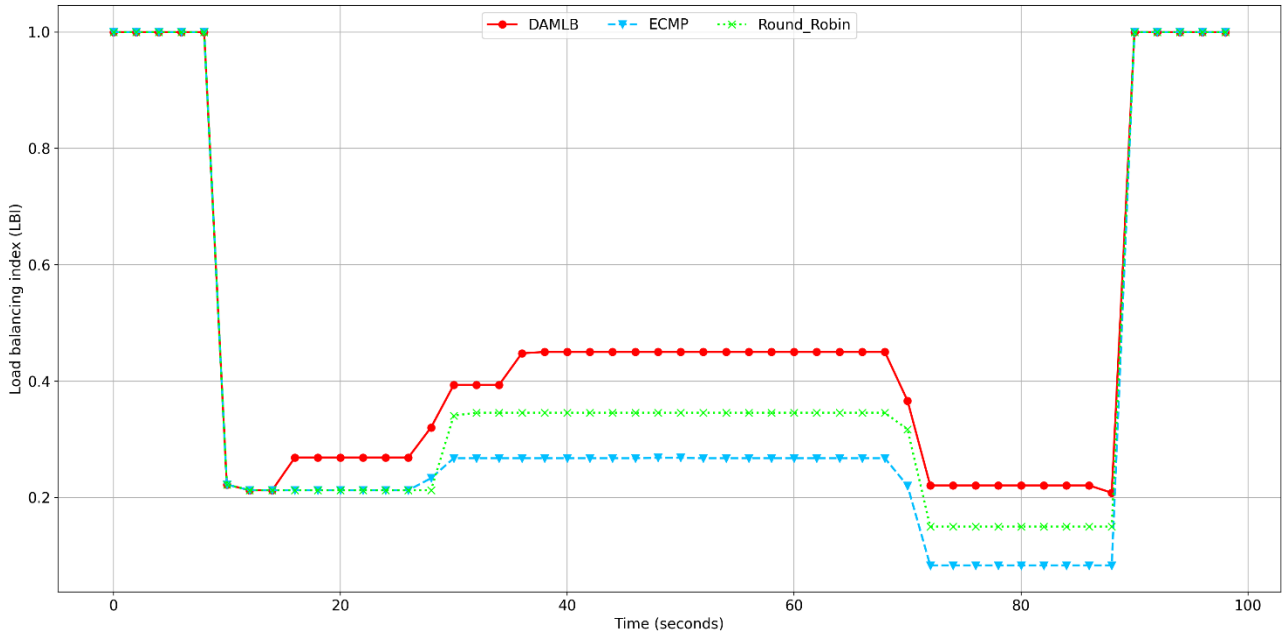


Рисунок 5.41 – Индекс балансировки нагрузки по времени

- **Начальная фаза**: В начале эксперимента, до генерации трафика, использование каналов сети отсутствует, что приводит к идеальному значению индекса LBI, равному 1. Это означает, что нагрузка распределена абсолютно равномерно (фактически отсутствует).

- **Фаза с нагрузкой**: Когда трафик начинает поступать (созданный с помощью iperf3), индекс LBI значительно снижается. Это связано с тем, что нагрузка теперь распределяется по сети, и некоторые каналы могут быть загружены больше, чем другие. Разные методы балансировки (DAMLB, ECMP, Round Robin) показывают различные уровни LBI на протяжении всего эксперимента, что отражает их эффективность в распределении трафика.

- **Завершение эксперимента**: По окончании передачи данных трафик перестает поступать, и нагрузка на каналы возвращается к нулю. В результате индекс LBI снова достигает идеального значения 1, так как все каналы снова свободны.

Метод DAMLB демонстрирует лучшее распределение нагрузки по сравнению с ECMP и Round Robin, так как его индекс LBI находится на более высоком уровне в фазе активного трафика. Это подтверждает его эффективность в более равномерном

распределении трафика между доступными каналами сети, что минимизирует перегрузки и улучшает общую производительность сети.

Анализ индекса балансировки нагрузки при различном количестве потоков данных

Эксперимент проводится с различным количеством потоков в сети с целью оценки эффективности балансировки нагрузки для трех методов маршрутизации в различных сценариях сетевой нагрузки. Среднее значение индекса балансировки нагрузки в зависимости от количества потоков представлено на графике 5.42.

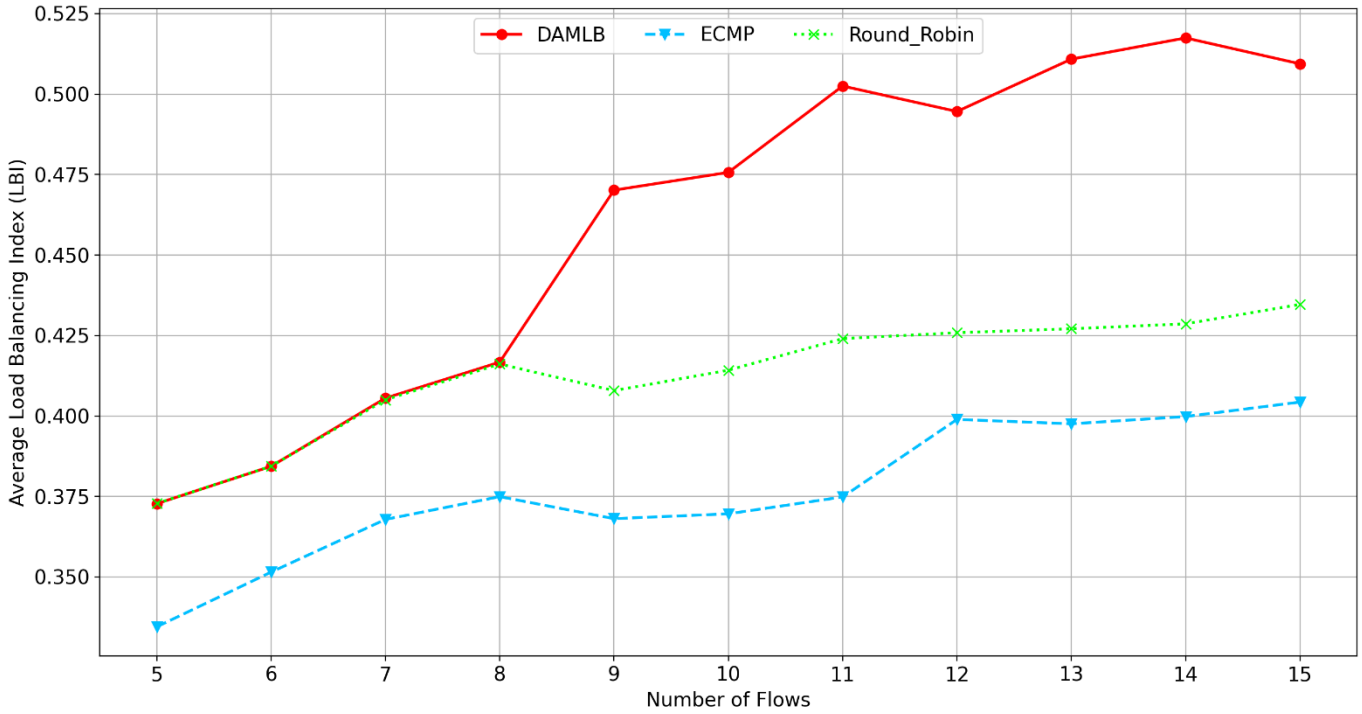


Рисунок 5.42 – Среднее значение индекса балансировки нагрузки по количеству потоков данных

Метод DAMLB показал значительно более высокую эффективность по сравнению с ECMP и Round Robin в обеспечении балансировки нагрузки, особенно при увеличении количества потоков. Этот метод демонстрирует хорошую адаптацию к изменениям сетевой нагрузки, обеспечивая равномерное распределение трафика и эффективное использование ресурсов. В то же время ECMP ограничен из-за своей статичной природы маршрутизации, а Round Robin, хотя и демонстрирует некоторое

улучшение, не достигает уровня гибкости и эффективности DAMLB в сложных сетевых сценариях с большим количеством потоков.

ОСНОВНЫЕ РЕЗУЛЬТАТЫ И ВЫВОДЫ

1. Исследован генетический алгоритм для многопутевой маршрутизации в ПКС. Проведен анализ влияния параметров вероятности мутации на эффективность поиска кратчайших путей. Выяснено, что более высокая вероятность мутации улучшает исследование пространства решений, однако увеличивает время выполнения алгоритма.

2. Рассмотрена оптимизация муравьиной колонии и ее применение в решении задачи многопутевой маршрутизации в ПКС. Доказано, что более низкие значения коэффициента жадности q_0 увеличивают степень исследования и способствуют нахождению более оптимальных решений.

3. Проведено исследование алгоритма стаи птиц и его использование в процессе многопутевой маршрутизации в ПКС. Проведен анализ различных весов инерции, показывающий, что увеличение веса инерции w помогает алгоритму находить более качественные решения, особенно на сложных маршрутах. Выявлено, что такой подход требует больше времени на выполнение, но дает лучшие результаты в сложных условиях сети.

4. Проанализирован алгоритм искусственной пчелиной колонии в контексте маршрутизации. Исследовано влияние параметра $limit$ на процесс поиска. Установлено, что низкий предел повышает исследовательские возможности алгоритма и дает более стабильные результаты, однако может увеличивать общее время выполнения.

5. Рассмотрен алгоритм светлячков и его модифицированная версия для многопутевой маршрутизации. Проведен сравнительный анализ, показывающий, что модифицированный алгоритм демонстрирует лучшее исследование пространства

решений и избегает преждевременной сходимости к локальным оптимумам, что позволяет находить более оптимальные пути в условиях сложных сетевых структур.

6. Проведен сравнительный анализ всех предложенных алгоритмов для многопутевой маршрутизации. Выявлена необходимость нахождения баланса между производительностью алгоритма и временем его выполнения. Существуют алгоритмы, которые обеспечивают хорошие результаты, но требуют больше времени для выполнения, и наоборот.

7. Исследована модель многопутевой маршрутизации на основе рекуррентной нейронной сети. Применение алгоритмов роевого интеллекта для оптимизации гиперпараметров позволило модели достигать высокой точности до 98% при выборе оптимальных маршрутов в режиме реального времени.

8. Проведен сравнительный анализ алгоритмов роевого интеллекта для оптимизации гиперпараметров разработанной модели нейросетевой многопутевой маршрутизации. Результаты экспериментов показали, что роевые алгоритмы превосходят такие методы, как поиск по сетке, байесовская оптимизация и случайный поиск.

9. Проведен эксперимент, который продемонстрировал эффективность метода динамической адаптивной многопутевой балансировки потоков данных (DAMLB) по сравнению с традиционными методами, такими как ECMP и Round Robin. Анализ показал, что применение метода DAMLB позволило более равномерно распределять нагрузку в сети, устраняя перегрузки на отдельных каналах и увеличивая общую пропускную способность.

ЗАКЛЮЧЕНИЕ

В результате проведенного исследования были достигнуты следующие основные результаты:

1. Проведен анализ традиционных и ПКС. Традиционные сетевые архитектуры имеют ряд недостатков, таких как сложность конфигурирования, ограниченная масштабируемость и трудоемкость настройки. ПКС представляют собой более гибкую и масштабируемую альтернативу благодаря разделению плоскостей управления и данных, а также использованию внешних контроллеров.

2. Разработаны математическая модель и метод интеллектуальной маршрутизации на основе алгоритмов роевого интеллекта, таких как генетический алгоритм, оптимизация муравьиной колонии, алгоритм стаи птиц, алгоритм искусственной пчелиной колонии и алгоритм светлячков. Результаты экспериментов показали, что предложенные алгоритмы способны эффективно находить оптимальные маршруты, повышая адаптивность к изменениям в сети.

3. Предложена нейросетевая модель многопутевой маршрутизации в ПКС, основанная на рекуррентной нейронной сети. Применение алгоритмов роевого интеллекта для оптимизации гиперпараметров позволило модели достигать высокой точности до 98% при выборе оптимальных маршрутов в режиме реального времени.

4. Предложены алгоритмы роевого интеллекта для оптимизации гиперпараметров разработанной модели нейросетевой многопутевой маршрутизации, которые позволяют автоматизировать подбор архитектуры и гиперпараметров модели. Результаты экспериментов показали, что роевые алгоритмы превосходят такие методы, как поиск по сетке, байесовская оптимизация и случайный поиск.

5. Разработаны модель и алгоритм динамической балансировки потоков данных в ПКС, обеспечивающие равномерное распределение нагрузки между каналами. Экспериментальные результаты показывают, что предложенный подход превосходит такие методы, как ECMP и Round Robin.

6. Разработана архитектура библиотеки программных компонентов интеллектуальной маршрутизации и балансировки потоков данных в ПКС, обеспечивающая эффективное управление потоками данных в сети на основе нейронных сетей и роевых алгоритмов.

7. Разработана структура визуальной программной системы SDNLoadBalancer для организации распределенной обработки потоков данных в ПКС. Система включает графический редактор для проектирования сетевых топологий, инструменты мониторинга и управления, а также средства тестирования производительности сети. Эта система позволяет эффективно управлять распределением нагрузки и оперативно подстраивать маршрутизацию под изменяющиеся условия сети.

СПИСОК ЛИТЕРАТУРЫ

1. Эндрю С. Таненбаум. Компьютерные сети. – М.: Питер. – 2006. – С. 960.
2. Joann Zimmerman, Charles E. Spurgeon. Ethernet Switches. O'Reilly Media, 2013, 63 p.
3. Behrouz A. Forouzan. Data Communications and Networking. McGraw-Hill Education, 2006, 1168 p.
4. P. Goransson, B. Chuck. “Software-Defined Networks: A Comprehensive Approach”, IEEE Communication Surveys & Tutorials, 2014, pp. 7-17.
5. Корячко В. П., Перепелкин Д. А. Программно-конфигурируемые сети. – М.: Горячая линия – Телеком. – 2020. – 288 с.
6. Красотин А. А., Алексеев И. В. Программно-конфигурируемые сети как этап эволюции сетевых технологий // Моделирование и анализ информационных систем. – 2013. – С. 110-124.
7. Keith Kirkpatrick. “Software-Defined Networking”, Communications of the ACM, September 2013, DOI:10.1145/2500468.2500473.
8. Hamid Farhady, HyunYong Lee, Akihiro Nakao. “Software-Defined Networking: A survey”, Computer Networks, Volume 81, 22 April 2015, pp. 79-95.
9. Akram Hakiri, Aniruddha Gokhale, Pascal Berthou, Douglas C. Schmidt, Gayraud Thierry. “Software-defined Networking: Challenges and Research Opportunities for Future Internet”, Computer Networks, December 2014, DOI: 10.1016/j.comnet.2014.10.015.
10. E. Karpilovsky, M. Caesar, J. Rexford, A. Shaikh, and J. van der Merwe. “Practical networkwide compression of ip routing tables”, IEEE Transactions on Network and Service Management, 2012, pp. 446-458.
11. Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, Jonathan Turner. “OpenFlow: Enabling innovation

in campus networks", ACM SIGCOMM Computer Communication Review, 2008, pp. 69-74.

12. Jean Tourrilhes, Puneet Sharma, Sujata Banerjee, Justin Pettit. "SDN and OpenFlow Evolution: A Standards Perspective", Computer, November 2014, pp. 22-29.
13. "OpenFlow Switch Specification".

Available from https://opennetworking.org/wp-content/uploads/2013/04/openflow-spec-v1.3.1.pdf

14. Wolfgang Braun, Michael Menth. "Software-Defined Networking Using OpenFlow: Protocols, Applications and Architectural Design Choices", Future Internet, May 2014, pp. 302-336, DOI:10.3390/fi6020302.
15. Shavan Askar, Faris Keti. "Performance Evaluation of Different SDN Controllers: A Review", International Journal of Science and Business, 2021, pp. 67-80.
16. Thomas D. Nadeau, Ken Gray. SDN: Software Defined Networks. O'Reilly Media, 2013, 384 p.
17. Rui Kubo, Tomonori Fujita, Yuji Agawa, Hikaru Suzuki. "Ryu SDN Framework – Open-source SDN Platform Software", NTT Technical Review, August 2014, pp. 18-22.
18. Rogério Leão Santos de Oliveira, Christiane Marie Schweitzer, Ailton Akira Shinoda, Ligia Rodrigues Prete. "Using Mininet for emulation and prototyping Software-Defined Networks", IEEE Colombian Conference on Communications and Computing (COLCOM), 2014, DOI: 10.1109/ColComCon.2014.6860404.
19. Deepak Kumar, Manu Sood. "Software Defined Networks (S.D.N): Experimentation with Mininet Topologies", Indian Journal of Science and Technology, August 2016, pp. 1-7.
20. "Mininet: An Instant Virtual Network on your Laptop". [Online]. Available: http://mininet.org/.
21. "Software Defined Networks: The New Norm for Networks".

Available from https://opennetworking.org/wp-content/uploads/2011/09/wp-sdn-newnorm.pdf

22. Erdal Akin, Turgay Korkmaz. “Comparison of Routing Algorithms with Static and Dynamic Link Cost in Software Defined Networking (SDN)”, IEEE Access, 2019, pp. 148629 – 148644, DOI: 10.1109/ACCESS.2019.2946707.
23. Noel Farrugia, Victor Buttigieg, Johann A. Briffa. “A globally optimised multipath routing algorithm using SDN”, 21st Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN), 2018, DOI: 10.1109/ICIN.2018.8401633.
24. Weibin Dai, Xiaoqian Sun, Sebastian Wandelt. “Finding feasible solutions for multi-commodity flow problems”, 35th Chinese Control Conference (CCC), 2016, DOI: 10.1109/ChiCC.2016.7553801.
25. Yi-Chih Lei, Kuochen Wang, Yi-Huai Hsu. “Multipath routing in SDN-based Data Center Networks”, European Conference on Networks and Communications (EuCNC), 2015, DOI: 10.1109/EuCNC.2015.7194100.
26. Черников А. С., Паус А. С. Многопоточная маршрутизация в программно-конфигурируемых сетях // Радиооптика. МГТУ им. Н.Э. Баумана. – 2016. – С. 35-46.
27. Maris Fajar Ramdhani, Sofia Naning Hertiana, Burhanudin Dirgantara. “Multipath routing with load balancing and admission control in Software-Defined Networking (SDN)”, 4th International Conference on Information and Communication Technology (ICoICT), 2016, DOI: 10.1109/ICoICT.2016.7571949.
28. Wu Jiawei, Xiuquan Qiao, Chen Junliang. “PDMR: Priority-based Dynamic Multi-path Routing Algorithm for a Software Defined Network”, IET Communications, 2019, pp. 179-185.
29. Jinyao Yan, Hailong Zhang, Qianjun Shuai, Bo Liu, Xiao Guo. “HiQoS: An SDN-based multipath QoS solution”, China Communications, 2015, pp. 123-133.

30. Daoquan Li, Haoxin Liu, Yingnan Jin. “MPF-MLBS: A Multi-path Load Balancing Strategy for SDN Networks Based on Multiple Performance Factors”, Mathematics and Computer Science, 2020, DOI:10.11648/j.mcs.20200503.11.
31. Farah Chahlaoui, Hamza Dahmouni, Hassan El Alami. “Multipath-routing based load-balancing in SDN networks”, 5th Conference on Cloud and Internet of Things (CIoT), 2022, DOI: 10.1109/CIoT53061.2022.9766801.
32. Md. Sajid Hossen, Md. Habibur Rahman, Md. Al-Mustanjid, Md. Arif Shakil Nobin, Md. Ahsan Habib. “Enhancing Quality of Service in SDN based on Multi-path Routing Optimization with DFS”, International Conference on Sustainable Technologies for Industry 4.0 (STI), 2019, DOI: 10.1109/STI47673.2019.9068057.
33. Khoa Truong Dinh, Slawomir Kuklinski, Wiktor Kujawa, Michał Ulaski. “MSDN-TE: Multipath Based Traffic Engineering for SDN”, Asian Conference on Intelligent Information and Database Systems, 2016, pp. 630-639, DOI:10.1007/978-3-662-49390-8_61.
34. Fiqih Rhamdani, Novian Anggis Suwastika, Muhammad Arief Nugroho. “Equal-Cost Multipath Routing in Data Center Network Based on Software Defined Network”, 6th International Conference on Information and Communication Technology (ICoICT), 2018, DOI: 10.1109/ICoICT.2018.8528730.
35. Wu Jiawei, Qiao Xiuquan, Nan Guoshun. “Dynamic and adaptive multi-path routing algorithm based on software-defined network”, International Journal of Distributed Sensor Networks, 2018, DOI:10.1177/1550147718805689.
36. Jang-Ping Sheu, Lee-Wei Liu, RB Jagadeesha, Yeh-Cheng Chang. “An efficient multipath routing algorithm for multipath TCP in Software-Defined Networks”, European Conference on Networks and Communications (EuCNC), 2016, DOI: 10.1109/EuCNC.2016.7561065.
37. Austin Jerome, Murat Yuksel, Syed Hassan Ahmed, Mostafa Bassiouni. “SDN-based load balancing for multi-path TCP”, IEEE Conference on Computer

Communications Workshops (INFOCOM WKSHPS), 2018, DOI: 10.1109/INFCOMW.2018.8406943.

38. Marcelo Pizzutti, Alberto Egon Schaeffer-Filho. “Adaptive Multipath Routing based on Hybrid Data and Control Plane Operation”, IEEE Conference on Computer Communications, 2019, DOI:10.1109/INFOCOM.2019.8737398.
39. Yao Chun Wang, Y. R. Lin, Guey-Yun Chang. “SDN-based dynamic multipath forwarding for inter-data center networking”, International Journal of Communication Systems, 2018, DOI:10.1002/dac.3843.
40. Alcardo Alex Barakabitze, Lingfen Sun, Is-Haka Mkwawa, Emmanuel Ifeachor. “A Novel QoE-Centric SDN-Based Multipath Routing Approach for Multimedia Services over 5G Networks”, IEEE International Conference on Communications (ICC), 2018, DOI: 10.1109/ICC.2018.8422617.
41. Manan Doshi, Aayush Kamdar, Krishna Kansara. “Multi-constraint QoS Disjoint Multipath Routing in SDN”, Computing, Communication and Signal Processing, 2019, pp. 377-387.
42. Mohamad Khattar Awad, Marwa Hassan Hafez Ahmed, A. F. Almutairi, Imtiaz Ahmad. “Machine Learning-Based Multipath Routing for Software Defined Networks”, Journal of Network and Systems Management, 2021, DOI:10.1007/s10922-020-09583-4.
43. S Thomas Valerrian Pasca, Siva Sairam Prasad Kodali, Kotaro Kataoka. “AMPS: Application aware multipath flow routing using machine learning in SDN”, Twenty-third National Conference on Communications (NCC), 2017, DOI: 10.1109/NCC.2017.8077095.
44. Feng Tian, Yang Zhang, Wei Ye, Cheng Jin. “Accelerating Distributed Deep Learning using Multi-Path RDMA in Data Center Networks”, The ACM SIGCOMM Symposium on SDN Research, 2021, DOI:10.1145/3482898.3483363.

45. Yi Zhang, Lanxin Qiu, Yangzhou Xu, Xinjia Wang, Shengjie Wang, Agyemang Paul, Zhefu Wu. “Multi-Path Routing Algorithm Based on Deep Reinforcement Learning for SDN”, Applied Sciences, 2023, DOI:10.3390/app132212520.

46. Truong Thu Huong, Ngo Do Dang Khoa, Nguyen Xuan Dung, Nguyen Huu Thanh. “A global multipath load-balanced routing algorithm based on Reinforcement Learning in SDN”, International Conference on Information and Communication Technology Convergence (ICTC), 2019, DOI: 10.1109/ICTC46691.2019.8939987.

47. Kai-Cheng Chiu, Chien-Chang Liu, Li-Der Chou. “Reinforcement Learning-Based Service-Oriented Dynamic Multipath Routing in SDN”, Wireless Communications and Mobile Computing, 2022, DOI:10.1155/2022/1330993.

48. Chao Chen, Feifan Xue, Zhengyong Lu, Zhongyun Tang, Chuanhuang Li. “RLMR: Reinforcement Learning Based Multipath Routing for SDN”, Wireless Communications and Mobile Computing, 2022, DOI:10.1155/2022/5124960.

49. Zheng Wang, Zhengyong Lu, Chuanhuang Li. “Research on deep reinforcement learning multi-path routing planning in SDN”, Journal of Physics Conference Series, 2020, DOI:10.1088/1742-6596/1617/1/012043.

50. Long Chen, Bin Hu, Zhi-Hong Guan, Lian Zhao. “Multiagent Meta-Reinforcement Learning for Adaptive Multipath Routing Optimization”, IEEE Transactions on Neural Networks and Learning Systems, 2021, DOI:10.1109/TNNLS.2021.3070584.

51. Anand Nayyar, Dac-Nhuong Le, Nhu Gia Nguyen. Advances in Swarm Intelligence for Optimizing Problems in Computer Science. Chapman and Hall/CRC, 2018, 314 p.

52. Xin-She Yang. Nature-Inspired Computation and Swarm Intelligence: Algorithms, Theory and Applications. Academic Press Inc, 2020, 442 p.

53. Adam Slowik. Swarm Intelligence Algorithms: Modifications and Applications. CRC Press, 2020, 349 p.

54. Aboul Ella Hassanien, Eid Emary. Swarm Intelligence: Principles, Advances, and Applications. CRC Press, 2015, 228 p.
55. Gerardo Beni. “The concept of cellular robotic system”, Proceedings IEEE International Symposium on Intelligent Control 1988, 1988, pp. 57-62.
56. Gerardo Beni, Jing Wang. “Swarm Intelligence in Cellular Robotic Systems”, NATO Advanced Workshop on Robots and Biological Systems, 1989, pp. 703-712.
57. Bonabeau, Eric, Marco Dorigo, Guy Theraulaz. Swarm Intelligence: from Natural to Artificial Systems. Oxford: Oxford University Press, 1999.
58. Inagaki, Haseyama, Kitajima. “A genetic algorithm for determining multiple routes and its applications”, Proceedings of the IEEE International Symposium on Circuits and Systems, pp. 137-140.
59. Chang Wook and R. S. Ramakrishna. “A genetic algorithm for shortest path routing problem and the sizing of populations”, IEEE Transactions on Evolutionary Computation , 2002, pp. 566-579.
60. G. Mitsue, C. Runwei and D. Wang. “Genetic Algorithms for solving shortest path problems”, Proceedings of the IEEE International Conference on Evolutionary Computation, 1997, pp. 401-406.
61. Holland, J. H. Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence. U Michigan Press, 1975.
62. M. Dorigo, V. Maniezzo, A. Colorni. “Ant system: optimization by a colony of cooperating agents”, IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics), 1996, pp. 29-41.
63. D. J. Rosenkrantz, R. E. Stearns, P. M. Lewis. “An analysis of several heuristics for the traveling salesman problem”, SIAM Journal on Computing, 1977, pp. 563-581.
64. J. Kennedy, R. Eberhart. “Particle Swarm Optimization”, Proceedings of the IEEE International Conference on Neural Networks, 1995, pp. 1942-1948.

65. D. Karaboga, B. Basturk. “A powerful and efficient algorithm for numerical function optimization: Artificial Bee Colony (ABC) algorithm”, Journal of Global Optimization, 2007, pp. 459-471.
66. Xin-She Yang. Nature-Inspired Metaheuristic Algorithms. Luniver Press, 2008.
67. M. Dorigo, L. M. Gambardella. “Ant colony system: a cooperative learning approach to the traveling salesman problem”, IEEE Transactions on Evolutionary Computation, 1997, pp. 53-66.
68. Смелянский Р. Л. Концепция программно-конфигурированных сетей: от идеи до стандартизации // CONNECT! Мир связи: Наука. Бизнес. Управление. – 2016. – № 4. – С. 62-67.
69. Смелянский Р. Л. Настоящее и будущее SDN&NFV // Первая миля, издательство АО Рекламно-издательский центр «Техносфера» (Москва). – 2016. – № 3. – С. 78-85.
70. Смелянский Р. Л. Программно-конфигурируемые сети // Открытые системы. – 2012. – № 9. – С. 15-26.
71. Леохин Ю. Л. Корпоративные сети: архитектура, технологии, управление. – М.: Фонд «Европейский центр по качеству», 2008. – С. 148.
72. Леохин Ю. Л., Бекасов В. Ю. Корпоративные сети: состояние, перспективы и тенденции. – М.: Фонд «Европейский центр по качеству», 2008. – С. 148.
73. Леохин Ю. Л., Дворецкий И. Н. Тенденции развития науки и техники в области производства серверного оборудования для дата-центров // Известия высших учебных заведений. Приборостроение. – 2013. – Т. 26. – № 12. – С. 20-24.
74. Леохин Ю. Л., Дворецкий И. Н., Мягков А. С. Отечественная операционная система Cloud/IX для серверов на процессорах архитектуры ARM // Качество. Инновации. Образование. – 2014. – Т. 113. – № 10. – С. 52-59.

75. Куракин Д. В. Маршрутизаторы для глобальных телекоммуникационных сетей и реализуемые в них алгоритмы // Информационные технологии. –1996. – № 2.
76. Куракин Д. В. Маршрутизация в сетях телекоммуникаций, построенных на базе международных стандартов взаимосвязи открытых систем // Автоматизация и современные технологии. – 1996. – № 3. – С. 35-43.
77. Тарасов В. Н. и др. Математические модели облачного вычислительного центра обработки данных с использованием OpenFlow // Вестник Оренбургского государственного университета. – 2012. – № 9 (145).
78. Макаренко С. И. Анализ воздействия преднамеренных помех на функционирование расширенного протокола маршрутизации внутреннего шлюза (EIGRP) // Информационные технологии моделирования и управления. – 2010. – № 2 (61). – С. 223-230.
79. Кравец О. Я., Пономарев А. В., Подерский И. С. Повышение эффективности маршрутизации в переходных режимах функционирования вычислительных сетей // Системы управления и информационные технологии. – 2003. – № 1-2. – С. 73-77.
80. Перепелкин А. И., Перепелкин Д. А. Разработка алгоритма динамической маршрутизации на базе протокола ospf в корпоративных вычислительных сетях // Вестник Рязанского государственного радиотехнического университета. – 2009. – № 28. – С. 68-72.
81. Перепелкин Д. А. Алгоритм адаптивной ускоренной маршрутизации на базе протокола OSPF при динамическом добавлении элементов корпоративной сети // Вестник Рязанского государственного радиотехнического университета. – 2010. – № 34. – С. 65-71.
82. Перепелкин Д. А., Перепелкин А. И. Алгоритм адаптивной ускоренной маршрутизации в условиях динамически изменяющихся нагрузок на линиях

связи в корпоративной сети // Информационные технологии. 2011. № 3. С. 2-7.

83. Перепелкин Д. А. Алгоритм адаптивной ускоренной маршрутизации на базе протокола OSPF при динамическом отказе элементов корпоративной сети // Вестник Рязанского государственного радиотехнического университета. – 2011. – № 37. – С. 53-58.
84. Dijkstra E. W. A note on two problems in connexion with graphs // Numerische mathematik. – 1959. – Т. 1. – № 1. – С. 269-271.
85. Bellman R. On a routing problem // Quarterly of applied mathematics. – 1958. – Т. 16. – № 1. – С. 87-90.
86. Евстигнеев В. А. Применение теории графов в программировании. – 1985.
87. Евстигнеев В. А., Касьянов В. Н. Теория графов: алгоритмы обработки деревьев. – 1994.
88. Koryachko V. P. “Models of Information Flows Control in Switching Networks”, Electrosvyaz [Jelektrosvjaz'], 1992, no. 4, pp. 4-5.
89. Koryachko V. P., Suskin V. V. “Scheduling Algorithm for Computational Process in Real-time Multiprocessor Systems”, Automatic Control and Computer Sciences, 1985, pp. 16-18.
90. Koryachko V. P., Smolyarov N. A. “Statistical Simulation of Conflict Situations in Multiprocessor Systems”, Automatic Control and Computer Sciences, 1981, pp. 89-90.
91. Koryachko V. P., Perepelkin D. A., Byshov V. S. “Improved Multipath Adaptive Routing Model in Computer Networks with Load Balancing”, Proceedings SIBCON 2016 – IEEE 2016 International Siberian Conference on Control and Communications, 2016, pp. 1-4.
92. Koryachko V. P., Perepelkin D. A., Ivanchikova M. A. “Adaptive Accelerated Routing between Data Centers Based on Paired Shifts Data”, Proceedings MECO

2016 – IEEE 5th Mediterranean Conference on Embedded Computing (MECO-2016), 2016, pp. 256-259.

93. Koryachko V. P., Perepelkin D. A., Byshov V. S. “Multipath Adaptive Routing in Computer Networks with Load Balancing”, Proceedings MECO 2016 – IEEE 5th Mediterranean Conference on Embedded Computing (MECO-2016), 2016, pp. 281-285.
94. Koryachko V. P., Perepelkin D. A., Byshov V. S. “Development and Research of Improved Model of Multipath Adaptive Routing in Computer Networks with Load Balancing”, Automatic Control and Computer Sciences, 2017, pp. 63-73.
95. Koryachko V. P., Perepelkin D. A., Ivanchikova M. A. “Adaptive Rerouting of Data Flows in Distributed Data Centers”, Microprocessors and Microsystems, 2017, pp. 505-509.
96. Лемешко А. В., Вавенко Т. В. Разработка и исследование потоковой модели адаптивной маршрутизации в программно-конфигурируемых сетях с балансировкой нагрузки // Доклады Томского государственного университета систем управления и радиоэлектроники. – 2013. – № 3 (29). – С. 100-108.
97. Лемешко А. В., Вавенко Т. В. Усовершенствование потоковой модели многопутевой маршрутизации на основе балансировки нагрузки // Проблемы телекоммуникаций. – 2012. – № 1 (6). – С. 12-29.
98. Лемешко А. В., Вавенко Т. В. Усовершенствование потоковой модели многопутевой маршрутизации на основе балансировки нагрузки // Проблемы телекоммуникаций. – 2012. – № 1 (6). - С. 12-29.
99. Поповский В. В., Лемешко А. В., Мельникова Л. И., Андрушко Д. В. Обзор и сравнительный анализ основных моделей и алгоритмов многопутевой маршрутизации в мультисервисных телекоммуникационных сетях // Прикладная радиоэлектроника. – 2005. – Т. 4. – № 4. – С. 372-382.

100. Карташевский В. Г., Буранова М. А. Оценка показателей качества обслуживания трафика IPTV в мультисервисной сети // Проблемы передачи информации в инфокоммуникационных системах. – 2015. – С. 46.
101. Карташевский В. Г., Буранова М. А. Влияние механизмов управления QoS на показатели качества обслуживания мультимедийного трафика сети Internet // T-Comm: Телекоммуникации и транспорт. – 2013. – Т. 7. – № 8. – С. 54-60.
102. Буранова М. А., Карташевский В. Г. Моделирование джиттера пакетов при передаче по мультисерсисной сети // Инфокоммуникационные технологии. – 2019. – Т. 17. – № 1. – С. 34-41.
103. Малахов С. В. и др. Влияние размера TCP-окна на распределение интервалов между пакетами трафика в программно-конфигурируемых сетях SDN // Инфокоммуникационные технологии. – 2016. – Т. 14. – №. 4. – С. 384-389.
104. Полежаев П. Н., Бахарева Н. Ф., Шухман А. Е. Разработка эффективного генетического алгоритма маршрутизации и обеспечения качества обслуживания для программно-конфигурируемой сети // Вестник Оренбургского государственного университета. – 2015. – № 1 (176).
105. Полежаев П. Н. и др. Применение методов муравьиной колонии в разработке эффективных алгоритмов маршрутизации и обеспечения QoS для корпоративных программно-конфигурируемых сетей // Интеллект. Инновации. Инвестиции. – 2014. – № 4. – С. 106-113.
106. Малахов С. В., Тарасов В. Н., Карташевский И. В. Теоретическое и экспериментальное исследование задержки в программно-кофигурируемых сетях // Инфокоммуникационные технологии. – 2015. – Т. 13. – № 4. – С. 409-413.
107. Rajendra K Jain, Dah-Ming W Chiu, William R Hawe. “A quantitative measure of fairness and discrimination”, Eastern Research Laboratory, Digital Equipment Corporation, Hudson, MA, Volume: 21, 1984, p. 1.

108. A. K. Rangisetti, T. V. Pasca, B. R. Tamma. “QoS Aware load balance in software defined LTE networks”, Computer Communications, 2017, pp. 52-71, DOI:10.1016/j.comcom.2016.09.005

109. Y.-D. Lin, C. C. Wang, Y.-J. Lu, Y.-C. Lai, and H.-C. Yang. “Two-tier dynamic load balancing in SDN-enabled Wi-Fi networks”, Wireless Networks, 2018, pp. 1-13, DOI:10.1007/s11276-017-1504-3.

110. Fiqih Rhamdani, Novian Anggis Suwastika, Muhammad Arief Nugroho. “Equal-Cost Multipath Routing in Data Center Network Based on Software Defined Network”, 6th International Conference on Information and Communication Technology (ICoICT), 2018, DOI: 10.1109/ICoICT.2018.8528730.

111. Mustafa ElGili Mustafa, Amin MubarkAlamin Ibrahim. “Load Balancing Algorithms Round-Robin (RR), Least-Connection and Least Loaded Efficiency”, International Journal of Computer and Information Technology, 2015.

112. Yilan Liu, Yun Pan, Muxi Yang, Wenqing Wang, Chi Fang, Ruijuan Jiang. “The multi-path routing problem in the Software Defined Network”, 11th International Conference on Natural Computation (ICNC), 2015, DOI: 10.1109/ICNC.2015.7377999.

113. Wen Chuan Yang, Shen Yao. “A Multi-path Routing Algorithm based on Ant Colony Optimization in Satellite Network”, IEEE 2nd International Conference on Big Data, Artificial Intelligence and Internet of Things Engineering (ICBAIE), 2021, DOI: 10.1109/ICBAIE52039.2021.9389995.

114. Na Lin, Zhi Xue Shao, Xiang Yu Chen. “Research on QoS Multipath Routing on ACO Algorithm”, Advanced Materials Research, 2011, https://doi.org/10.4028/www.scientific.net/AMR.219-220.762.

115. Chunzhi Wang, Gang Zhang, Hongwei Chen, Hui Xu. “An ACO-based elephant and mice flow scheduling system in SDN”, IEEE 2nd International Conference on Big Data Analysis (ICBDA), 2017, DOI: 10.1109/ICBDA.2017.8078760.

116. Feng Wang, Yuan Man, Lichun Man. “Intelligent optimization approach for the k shortest paths problem based on genetic algorithm”, 10th International Conference on Natural Computation (ICNC), 2014, DOI: 10.1109/ICNC.2014.6975838.

117. H. Kusetogullari, M. S. Leeson, W. Ren, E. L. Hines. “K- shortest path network problem solution with a hybrid Genetic Algorithm: Particle Swarm Optimization algorithm”, 13th International Conference on Transparent Optical Networks, 2011, DOI: 10.1109/ICTON.2011.5970873.

118. I. Goodfellow, Y. Bengio, A. Courville, Deep Learning, MIT Press, 2016.

119. Jeffrey L. Elman. “Finding Structure in Time”, Cognitive Science, 1990, https://doi.org/10.1207/s15516709cog1402_1.

120. S. Hochreiter, J. Schmidhuber. “Long Short-Term Memory”, Neural Computation, 1997, https://doi.org/10.1162/neco.1997.9.8.1735.

121. Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, Yoshua Bengio. “Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation”, Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), 2014, https://doi.org/10.3115/v1/D14-1179.

122. Перепелкин Д. А., Нгуен В. Т. Исследование процессов балансировки нагрузки в программно-конфигурируемых сетях на основе генетического алгоритма // Вестник Рязанского государственного радиотехнического университета. – 2021. – № 77. – С. 43-57.

123. Перепелкин Д. А., Нгуен В. Т. Исследование и анализ процессов многопутевой маршрутизации и балансировки потоков данных в программно-конфигурируемых сетях на основе генетического алгоритма // Вестник Рязанского государственного радиотехнического университета. – 2022. – № 79. – С. 31-48.

124. Перепелкин Д. А., Нгуен В. Т. Интеллектуальная многопутевая маршрутизация в программно-конфигурируемых сетях на основе алгоритма

искусственной пчелиной колонии // Информационные технологии. – 2022. – Т. 28. – № 8. – С. 395-404.

125. Перепелкин Д. А., Иванчикова М. А., Нгуен В. Т. Интеллектуальная многопутевая маршрутизация в программно-конфигурируемых сетях на основе алгоритмов оптимизации муравьиной колонии // Информационные технологии. – 2022. – Т. 28. – № 10. – С. 520-528.
126. Перепелкин Д. А., Иванчикова М. А., Нгуен В. Т. Интеллектуальная многопутевая маршрутизация в программно-конфигурируемых сетях на основе алгоритма миграции стаи птиц // Вестник Рязанского государственного радиотехнического университета. – 2022. – № 82. – С. 44-59.
127. Перепелкин Д. А., Нгуен В. Т. Интеллектуальная многопутевая маршрутизация в программно-конфигурируемых сетях на основе модели поведения роя светлячков // Цифровая обработка сигналов. – 2023. – № 4. – С. 32-40.
128. Перепелкин Д. А., Иванчикова М. А., Нгуен В. Т. Нейросетевая многопутевая маршрутизация в программно-конфигурируемых сетях на основе генетического алгоритма // Информационные технологии. – 2023. – Т. 29. – № 12. – С. 622-629.
129. Перепелкин Д. А., Нгуен В. Т. Нейросетевая многопутевая маршрутизация в программно-конфигурируемых сетях на основе алгоритмов оптимизации муравьиной колонии // Вестник Рязанского государственного радиотехнического университета. – 2024. – № 89. – С. 39-55.
130. Перепелкин Д. А., Нгуен В. Т. Задача многопутевой маршрутизации в программно-конфигурируемых сетях на основе алгоритма искусственной пчелиной колонии // Современные технологии в науке и образовании СТНО. – 2022. – Т. 10. – С. 132-135.
131. Перепелкин Д. А., Нгуен В. Т. Оптимизация гиперпараметров рекуррентной нейронной сети на основе генетического алгоритма для задачи многопутевой

маршрутизации в программно-конфигурируемых сетях // Современные технологии в науке и образовании СТНО. – 2023. – Т. 10. – С. 125-127.

132. Dmitry Perepelkin, Tin Nguyen. “Research of Multipath Routing Processes in Software Defined Networks Based on Ant Colony Optimization”, 11th Mediterranean Conference on Embedded Computing (MECO), 2022, DOI: 10.1109/MECO55406.2022.9797090.
133. Dmitry Perepelkin, Tin Nguyen. “Research of Multipath Routing and Load Balancing Processes in Software Defined Networks Based on Artificial Bee Colony Algorithm”, ELEKTRO, 2022, DOI: 10.1109/ELEKTRO53996.2022.9803416.
134. Dmitry Perepelkin, Tin Nguyen. “Research of Multipath Routing Processes in Software Defined Networks Based on Firefly Algorithm”, International Russian Automation Conference (RusAutoCon), 2022, DOI: 10.1109/RusAutoCon54946.2022.9896320.
135. Dmitry Perepelkin, Maria Ivanchikova, Tin Nguyen. “Research of Multipath Routing and Load Balancing Processes in Software Defined Networks Based on Bird Migration Algorithm”, International Russian Smart Industry Conference (SmartIndustryCon), 2023, DOI: 10.1109/SmartIndustryCon57312.2023.10110788.
136. Dmitry Perepelkin, Maria Ivanchikova, Tin Nguyen. “Neural Network Multipath Routing in Software Defined Networks Based on Artificial Bee Colony Algorithm”, XVIII International Symposium Problems of Redundancy in Information and Control Systems (REDUNDANCY), 2023, DOI: 10.1109/Redundancy59964.2023.10330174.

ПРИЛОЖЕНИЕ А
СВИДЕТЕЛЬСТВА О ГОСУДАРСТВЕННОЙ РЕГИСТРАЦИИ ПРОГРАММЫ ДЛЯ ЭВМ

РОССИЙСКАЯ ФЕДЕРАЦИЯ

СВИДЕТЕЛЬСТВО

о государственной регистрации программы для ЭВМ

№ 2024690323

«Программный компонент многопутевой маршрутизации в программно-конфигурируемых сетях на основе генетического алгоритма»

Правообладатель: ***федеральное государственное бюджетное образовательное учреждение высшего образования «Рязанский государственный радиотехнический университет имени В.Ф. Уткина» (RU)***

Авторы: ***Перепелкин Дмитрий Александрович (RU), Нгуен Ван Тин (VN)***

Заявка № **2024689850**

Дата поступления **09 декабря 2024 г.**

Дата государственной регистрации в Реестре программ для ЭВМ ***13 декабря 2024 г.***

Руководитель Федеральной службы по интеллектуальной собственности

Ю.С. Зубов

РОССИЙСКАЯ ФЕДЕРАЦИЯ

СВИДЕТЕЛЬСТВО

о государственной регистрации программы для ЭВМ

№ 2024691003

«Программный компонент многопутевой маршрутизации в программно-конфигурируемых сетях на основе алгоритмов оптимизации муравьиной колонии»

Правообладатель: ***федеральное государственное бюджетное образовательное учреждение высшего образования «Рязанский государственный радиотехнический университет имени В.Ф. Уткина» (RU)***

Авторы: ***Перепелкин Дмитрий Александрович (RU), Нгуен Ван Тин (VN)***

Заявка № **2024690314**
Дата поступления **11 декабря 2024 г.**
Дата государственной регистрации
в Реестре программ для ЭВМ ***18 декабря 2024 г.***

Руководитель Федеральной службы
по интеллектуальной собственности

Ю.С. Зубов

РОССИЙСКАЯ ФЕДЕРАЦИЯ

СВИДЕТЕЛЬСТВО

о государственной регистрации программы для ЭВМ

№ 2024690740

«Программный компонент многопутевой маршрутизации в программно-конфигурируемых сетях на основе алгоритмов роевого интеллекта»

Правообладатель: ***федеральное государственное бюджетное образовательное учреждение высшего образования «Рязанский государственный радиотехнический университет имени В.Ф. Уткина» (RU)***

Авторы: ***Перепелкин Дмитрий Александрович (RU), Нгуен Ван Тин (VN)***

Заявка № **2024689860**
Дата поступления **09 декабря 2024 г.**
Дата государственной регистрации
в Реестре программ для ЭВМ ***17 декабря 2024 г.***

Руководитель Федеральной службы
по интеллектуальной собственности

Ю.С. Зубов

РОССИЙСКАЯ ФЕДЕРАЦИЯ

СВИДЕТЕЛЬСТВО

о государственной регистрации программы для ЭВМ

№ 2024690797

«Программный компонент нейросетевой маршрутизации потоков данных в программно-конфигурируемых сетях на основе генетического алгоритма»

Правообладатель: ***федеральное государственное бюджетное образовательное учреждение высшего образования «Рязанский государственный радиотехнический университет имени В.Ф. Уткина» (RU)***

Авторы: ***Перепелкин Дмитрий Александрович (RU), Нгуен Ван Тин (VN)***

Заявка № **2024689856**
Дата поступления **09 декабря 2024 г.**
Дата государственной регистрации
в Реестре программ для ЭВМ ***17 декабря 2024 г.***

Руководитель Федеральной службы
по интеллектуальной собственности

Ю.С. Зубов

РОССИЙСКАЯ ФЕДЕРАЦИЯ

СВИДЕТЕЛЬСТВО

о государственной регистрации программы для ЭВМ

№ 2024691119

«Программный компонент нейросетевой маршрутизации потоков данных в программно-конфигурируемых сетях на основе алгоритмов оптимизации муравьиной колонии»

Правообладатель: ***федеральное государственное бюджетное образовательное учреждение высшего образования «Рязанский государственный радиотехнический университет имени В.Ф. Уткина» (RU)***

Авторы: ***Перепелкин Дмитрий Александрович (RU), Нгуен Ван Тин (VN)***

Заявка № **2024689853**
Дата поступления **09 декабря 2024 г.**
Дата государственной регистрации
в Реестре программ для ЭВМ ***19 декабря 2024 г.***

Руководитель Федеральной службы
по интеллектуальной собственности

Ю.С. Зубов

РОССИЙСКАЯ ФЕДЕРАЦИЯ

СВИДЕТЕЛЬСТВО

о государственной регистрации программы для ЭВМ

№ 2024691678

«Программный компонент нейросетевой маршрутизации потоков данных в программно-конфигурируемых сетях на основе алгоритмов роевого интеллекта»

Правообладатель: ***федеральное государственное бюджетное образовательное учреждение высшего образования «Рязанский государственный радиотехнический университет имени В.Ф. Уткина» (RU)***

Авторы: ***Перепелкин Дмитрий Александрович (RU), Нгуен Ван Тин (VN)***

Заявка № **2024689848**
Дата поступления **09 декабря 2024 г.**
Дата государственной регистрации
в Реестре программ для ЭВМ ***23 декабря 2024 г.***

Руководитель Федеральной службы
по интеллектуальной собственности

Ю.С. Зубов

РОССИЙСКАЯ ФЕДЕРАЦИЯ

СВИДЕТЕЛЬСТВО

о государственной регистрации программы для ЭВМ

№ 2025610943

«Программный компонент динамической балансировки потоков данных в программно-конфигурируемых сетях с обеспечением качества сетевого сервиса»

Правообладатель: ***федеральное государственное бюджетное образовательное учреждение высшего образования «Рязанский государственный радиотехнический университет имени В.Ф. Уткина» (RU)***

Авторы: ***Перепелкин Дмитрий Александрович (RU), Нгуен Ван Тин (VN)***

Заявка № **2024689864**
Дата поступления **09 декабря 2024 г.**
Дата государственной регистрации
в Реестре программ для ЭВМ ***16 января 2025 г.***

Руководитель Федеральной службы
по интеллектуальной собственности

Ю.С. Зубов

ПРИЛОЖЕНИЕ Б

АКТЫ О ВНЕДРЕНИИ РЕЗУЛЬТАТОВ ДИССЕРТАЦИОННОЙ РАБОТЫ

Перевод с вьетнамского языка на русский язык

ООО «ТЕХНОЛОГИИ HTR»
Адрес: дом 1, переулок 64, ул. Чай Ка, микрорайон Чыонг Динь, район Хай Ба Чынг, г. Ханой
Тел.: +84 946862521

АКТ

(О внедрении результатов кандидатской диссертации
Нгуен Ван Тин, «Рязанский государственный радиотехнический университет им. В.Ф. Уткина», Российская Федерация)

Данный акт удостоверяет, что математические модели, алгоритмы и программная система SDNLoadBalancer, разработанные в рамках диссертации «Математическое и программное обеспечение процессов интеллектуальной маршрутизации и балансировки потоков данных в программно-конфигурируемых сетях на основе нейронных сетей и роевых алгоритмов», были внедрены в ООО «Технологии HTR».

Методы, предложенные в диссертации, применяются для исследования и разработки и выполняют следующие основные функции:

- Интеллектуальная маршрутизация на основе актуальной сетевой информации в реальном времени.
- Адаптивное распределение трафика в зависимости от состояния сети.

В процессе применения программной системы SDNLoadBalancer компьютерная сеть функционировала стабильно, эффективно справляясь с явлениями перегрузок, увеличивая пропускную способность и снижая процент потерь пакетов. Интерфейс SDNLoadBalancer разработан четким, интуитивно понятным и удобным для использования.

Ханой, 01 ноября 2024 года
ДИРЕКТОР
(Подпись)
(Печать: ООО «ТЕХНОЛОГИИ HTR» - район Хай Ба Чынг – город Ханой – Код бизнеса: 0110842288-С.Т.Т.N.H.H)
Нгуен Хонг Линь

Перевод данного текста выполнен переводчиком Васькиной Мариной Сергеевной

Российская Федерация
Город Москва
Тринадцатого ноября две тысячи двадцать четвёртого года

Я, Акимов Глеб Борисович, нотариус города Москвы, свидетельствую подлинность подписи переводчика Васькиной Марины Сергеевны.
Подпись сделана в моём присутствии.
Личность подписавшего документ установлена.

Зарегистрировано в реестре: № 77/09-н/77-2024- 67-4382
Уплачено за совершение нотариального действия: 400 руб. 00 коп.

Г.Б. Акимов

Всего прошнуровано, пронумеровано
и скреплено печатью 2 листа (ов)
Нотариус:

CÔNG TY TNHH CÔNG NGHỆ HTR
Số 1 ngách 64, ngõ Trại Cá, Phường Trương Định, Quận Hai Bà Trưng, Thành phố Hà Nội
Tel.: +84 946862521

CHỨNG NHẬN

(Việc ứng dụng kết quả luận án Nghiên cứu sinh tiến sĩ Nguyễn Văn Tin trường "Đại học Tổng hợp Kỹ thuật Vô tuyến Điện tử Quốc gia Ryazan mang tên V.F. Utkin – LB Nga)

Giấy chứng nhận này chứng thực các mô hình toán học, thuật toán và hệ thống phần mềm SDNLoadBalancer, được phát triển trong khuôn khổ luận án "Mô hình toán học và phần mềm cho các quá trình định tuyến đa đường và cân bằng tải trong mạng định nghĩa bằng phần mềm dựa trên mạng thần kinh nhân tạo và các thuật toán bầy đàn", đã được ứng dụng tại Công ty TNHH Công nghệ HTR.

Các phương pháp trong luận án được áp dụng vào nghiên cứu và phát triển, đồng thời thực hiện các chức năng chính sau:

- Định tuyến thông minh dựa trên thông tin thời gian thực của mạng.
- Phân bổ lưu lượng thích ứng theo điều kiện của mạng.

Trong quá trình vận hành phần mềm SDNLoadBalancer, hệ thống mạng máy tính đã hoạt động ổn định, xử lý hiệu quả các hiện tượng tắc nghẽn, tăng băng thông và giảm tỷ lệ mất gói. Giao diện của SDNLoadBalancer được thiết kế rõ ràng, trực quan, dễ sử dụng.

Hà Nội, ngày 11 tháng 11 năm 2024

GIÁM ĐỐC

CÔNG TY TNHH CÔNG NGHỆ HTR

Nguyễn Hồng Linh

УТВЕРЖДАЮ
И.о. ректора ФГБОУ ВО «РГРТУ»
_____ к.э.н., доцент Банников С.А.
«16» 12 2024 г.

АКТ

об использовании в учебном процессе ФГБОУ ВО «Рязанский государственный радиотехнический университет им. В.Ф. Уткина» результатов диссертационной работы на соискание ученой степени кандидата технических наук

Нгуен Ван Тин

Настоящим актом удостоверяется, что результаты исследований, полученные в кандидатской диссертации Нгуен В.Т., внедрены в учебный процесс ФГБОУ ВО «Рязанский государственный радиотехнический университет им. В.Ф. Уткина» (РГРТУ).

Разработанные в диссертационной работе модели, методы, алгоритмы и программная система интеллектуальной маршрутизации и балансировки потоков данных в программно-конфигурируемых сетях (ПКС) на основе роевых алгоритмов используются в учебном процессе Рязанского государственного радиотехнического университета имени В.Ф. Уткина (РГРТУ) при чтении лекций, проведении лабораторных и практических занятий по курсам «Проектирование и поддержка программно-конфигурируемых сетей» по направлению 09.03.01 – «Информатика и вычислительная техника» и «Программно-конфигурируемые сети» направления 11.03.03 – «Конструирование и технология электронных средств».

Основные положения и выводы диссертации Нгуен В.Т. позволили качественно и по-новому освещать в учебном процессе вопросы разработки новых методов, алгоритмов и программных средств интеллектуального управления потоками данных в ПКС.

Использование полученных в кандидатской диссертации результатов и научных работ Нгуен В.Т. позволило:

- отразить в лекционном материале современный уровень развития вычислительных систем, комплексов и компьютерных сетей;

- разработать и внедрить лабораторные и практические работы с использованием новых методов и алгоритмов интеллектуальной маршрутизации и балансировки потоков данных в ПКС;

- расширить тематику курсового и дипломного проектирования;

- повысить качество учебного процесса и ознакомить студентов с новыми разработками в области сетевых и облачных технологий.

Председатель
Научно-методического совета РГРТУ
Заведующий кафедрой
«Вычислительная и прикладная математика»
д.т.н., доцент Г.В. Овечкин

Декан факультета вычислительной техники
д.т.н., профессор Д.А. Перепелкин

Заведующий кафедрой
«Систем автоматизированного
проектирования вычислительных средств»
д.т.н., профессор В.П. Корячко