Министерство образования и науки Российской Федерации

Федеральное государственное автономное образовательное учреждение высшего профессионального образования

**«Уральский федеральный университет**

**имени первого Президента России Б.Н. Ельцина»**

**Институт математики и компьютерных наук**

**Кафедра алгебры и дискретной математики**

**Автоматизация процесса тестирования коммуникационного оборудования на основе генетических алгоритмов**

|  |  |
| --- | --- |
| Допустить к защите:  Зав. кафедрой  доктор физико-математических наук,  профессор,  Волков М. В.  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  «\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_2015 г.  Нормоконтролер | Дипломная работа  студента 6 курса  Перевощикова И.В.  Научный руководитель**:**  Доцент кафедры алгебры и дискретной математики ИМКН УрФУ,  кандидат технических наук  Синадский Н.И. |

Екатеринбург

2015

# РЕФЕРАТ

Перевощиков И.В. Автоматизация процесса тестирования коммуникационного оборудования на основе генетических алгоритмов, дипломная работа: **TODO**

Ключевые слова: СИНТЕЗ ТРАФИКА, ГЕНЕТИЧЕСКИЙ АЛГОРИТМ, ТЕСТИРОВАНИЕ СЕТЕВОГО ОБОРУДОВАНИЯ

Объект исследования – эволюционные подходы к процессу тестирования коммуникационного оборудования на предмет выявления пороговых значений характеристик устройства.

Цель работы – разработка программного комплекса на базе генетического алгоритма для автоматического тестирования сетевого оборудования.

В процессе работы проводилось исследование работы генетических алгоритмов и их применения в тестировании коммуникационного оборудования.

В результате был разработан программный код, реализующий комплексное тестирование сетевого оборудования с использованием автоматически сгенерированных последовательностей сетевых пакетов, близких по своим характеристикам к последовательностям пакетов в реальной сети.

Результаты работы могут применяться разработчиками сетевого оборудования для автоматического тестирования устройств на предмет выявления условий, являющихся для данного оборудования экстремальными.

# Содержание

[Введение 4](#_Toc411376213)

[Глава 1. Методологии и объект тестирования 7](#_Toc411376214)

[Глава 2. Модель сети 16](#_Toc411376215)

[Глава 3. Реализация тестирующей программы 29](#_Toc411376216)

[Глава 4. Эксперимент 38](#_Toc411376217)

[Заключение 39](#_Toc411376218)

[Список использованной литературы 40](#_Toc411376219)

[Приложение 41](#_Toc411376220)

# Введение

В современном мире люди все больше и больше полагаются на возможности компьютерных сетей, в том числе на доступ к сервисам, предназначенным для обработки и хранения информации. Как известно, одна из трех составляющих защиты информации – это обеспечение ее доступности.

В компьютерных сетях на доступность того или иного узла влияет много факторов, один из главных – работоспособность коммутационного оборудования, объединяющего отдельные устройства в сети. То есть при проектировании той или иной сети, инженер должен знать реальные возможности сетевого оборудования для успешного обеспечения бесперебойного функционирования сети.

Целью данной работы является автоматизация исследования коммуникационного оборудования на выявление таких конфигураций сети, которые будут являться для тестируемого сетевого устройства экстремальными.

Другими словами, требуется разработать программный комплекс, автоматически оптимизирующий конфигурацию сети, в которой установлено тестируемое оборудование, таким образом, чтобы привести некоторые значимые для исследования характеристики данного оборудования к «крайним» (пороговым) значениям. Так как множество возможных вариантов сетей охватить методами грубой силы не представляется возможным из-за временных ограничений, было решено воспользоваться средствами генетического алгоритма.

Тема генетических алгоритмов исследована довольно широко, и генетические алгоритмы нашли свое применение в таких областях науки как

* Экстремальные задачи (нахождение точек минимума и максимума),
* Задачи о кратчайшем пути,
* Задачи компоновки,
* Составление расписаний,
* Аппроксимация функций,
* Отбор (фильтрация) входных данных,
* Настройка искусственной нейронной сети,
* Моделирование искусственной жизни,
* Биоинформатика (свертывание белков и РНК),
* Игровые стратегии,
* Нелинейная фильтрация.

Данная работа является попыткой применить генетический алгоритм в автоматизации тестирования сетевого оборудования. С этой целью был разработан программный комплекс, состоящий из трех частей:

1. Реализация математической модели конфигурации сети, в которую входят подсети, узлы в этих подсетях и сетевые взаимодействия между узлами модели, а также принимающие участие в эволюции функции распределения вероятности принятия некоторой характеристикой потока какого-либо значения из заданного отрезка.
2. Непосредственная реализация генетического алгоритма, который производит формирование популяции, скрещивание и отбор. Индивидом в данном алгоритме является некоторая конфигурация сети, представленная при помощи математической модели.
3. Модуль, отвечающий за тестирование конфигурации сети, полученной из генетического алгоритма, возвращающий для каждой конфигурации некоторую оценку, необходимую для произведения дальнейшего отбора генетическим алгоритмом.

Общая схема взаимодействия модулей представлена на рисунке 1.



Рисунок 1**{SEQ pic \c}**{SEQ pic \c}{SEQ pic \c}{SEQ pic \c}{SEQ pic \c}. Общая схема работы программного комплекса

В качестве языка программирования в данной работе был выбран Python, как язык, наиболее эффективный при реализации сетевых взаимодействий с достаточной степенью гибкости настройки последних.

В качестве экспериментального объекта исследований – типичный представитель семейства «домашних» маршрутизаторов – маршрутизатор производства компании D-Link, модель DIR-300.

# Глава 1. Методологии и объект тестирования

Маршрутизаторы

*Маршрутизатор* – это устройство сетевого уровня эталонной модели OSI, использующее одну или более метрик для определения оптимального пути передачи сетевого пакета на основании информации сетевого уровня. Из этого определения вытекает, что маршрутизатор, прежде всего, необходим для определения пути следования пакетов, движущихся в компьютерной сети.

Ключевым свойством маршрутизатора является продвижение сетевых пакетов с его входной линии на выходную.

Общая схема архитектуры маршрутизатора показана на рисунке 2.



Рисунок 2. Функциональная схема маршрутизатора

****Уровень интерфейсов****

Каждый порт маршрутизатора – это конечный узел для той подсети, к которой он присоединен. Поэтому, как и всем другим конечным узлам, портам маршрутизатора назначаются один (или несколько) аппаратных адресов и один (или несколько) сетевых адресов. Если для перемещения кадра в пределах подсети используется локальный адрес, то для продвижения пакета по сети необходим сетевой адрес. К примеру, протокол IP 4 версии использует сетевые IP-адреса, которые состоят из 4 байт. Сетевые адреса должны быть уникальны в пределах всей составной сети.

Интерфейсы маршрутизатора выполняют полный набор функций физического и канального уровней по передаче пакета, включая получение доступа к среде, формирование двоичных сигналов, прием и передачу кадра, буферизацию кадров в своей оперативной памяти, подсчет его контрольной суммы и отбрасывание поврежденных кадров. Обработка завершается отбрасыванием заголовка кадра и извлечением из поля данных пакета верхнего уровня, который передается модулю сетевого протокола маршрутизатора.

Уровень сетевого протокола

Модуль сетевого протокола анализирует содержимое полей заголовка пакета. Прежде всего, он снова вычисляет контрольную сумму, но уже не для кадра, а для пакета или части пакета сетевого уровня: в частности, в случае пакета IP вычисляется контрольная сумма заголовка. Если пакет пришел поврежденным, то он отбрасывается. Далее проверяется время жизни пакета. Если норма превышена, то пакет также отбрасывается. На этом этапе вносятся корректировки в содержимое некоторых полей: например, уменьшается время жизни пакета, пересчитывается контрольная сумма.

Маршрутизаторы, операционная система которых содержит модуль сетевого протокола, способны производить разбор и анализ отдельных полей пакета сетевого уровня. Маршрутизаторы, как правило, также в состоянии анализировать и заголовки транспортного уровня, поэтому фильтры могут не пропускать в сеть пакеты определенных прикладных сервисов, использующих конкретные программные порты, значения которых и используются при составлении правил фильтрации.

В случае, если интенсивность поступления пакетов превышает скорость, с которой они обрабатываются маршрутизатором, пакеты помещаются в очередь. Программное обеспечение маршрутизатора может реализовать различные дисциплины обслуживания очередей, но при достижении длины очереди некоторого порогового значения вновь поступающие пакеты отбрасываются.

И конечно, на сетевом уровне решается основная задача маршрутизатора — определение маршрута пакета. По номеру сети, извлеченному из поля адреса назначения заголовка пакета, модуль сетевого протокола находит в таблице маршрутизации запись, содержащую сетевой адрес следующего маршрутизатора и идентификатор своего порта, на который нужно передать данный пакет, чтобы он двигался в правильном направлении. Если в таблице отсутствует запись о сети назначения пакета и к тому же нет записи о маршруте по умолчанию, то данный пакет отбрасывается.

С сетевого уровня пакет, аппаратный адрес следующего маршрутизатора и идентификатор выходного порта передаются вниз по иерархии на канальный уровень. На основании идентификатора порта осуществляется перемещение этих данных в выходной буфер одного из интерфейсов маршрутизатора, а затем средствами канального уровня выполняется упаковка пакета в кадр соответствующего формата. В поле адреса назначения заголовка кадра помещается локальный адрес следующего маршрутизатора. Готовый кадр отправляется в сеть.

****Уровень протоколов маршрутизации****

Сетевые протоколы активно используют в своей работе таблицу маршрутизации, но ни ее построением, ни поддержкой данных, хранящихся в ней, они не занимаются. Основная работа по созданию таблиц маршрутизации выполняется автоматически, но, как правило, таблицу можно скорректировать или дополнить вручную.

Для автоматического построения таблиц маршрутизации маршрутизаторы обмениваются информацией о топологии составной сети в соответствии со специальным служебным протоколом. Протоколы этого типа называются протоколами маршрутизации.

Методы коммутации

Важной составляющей работы маршрутизатора является принцип, по которому организована коммутация пакетов, так как этот принцип непосредственно влияет на производительность устройства.

Маршрутизаторы можно разделить на группы по внутренней логической архитектуре коммутации:

* Коммутационная матрица,
* Многовходовая разделяемая память,
* Архитектура с общей шиной.

Наибольшее распространение получила последняя архитектура. Для связи всех сетевых интерфейсов используется одна шина. Для сохранения высокой производительности ее скорость должна быть, по крайней мере, в раз больше, чем скорость поступления данных в порт устройства, где - сумма скоростей всех портов. Кроме этого, многое зависит от способа передачи данных по шине. Очевидно, что кадр целиком передавать нежелательно, так как в это время остальные порты будут простаивать. Что бы обойти это ограничение, обычно применяют следующий метод. Данные разбиваются на небольшие блоки (по несколько десятков байт), и передаются параллельно сразу между несколькими портами.

Данная архитектура реализует метод временной коммутации частей кадров. Решение легко масштабируется, достаточно просто, надежно, и в наше время является наиболее распространенным.

Выбор способа тестирования

Существует несколько методологий тестирования оборудования. Все они отличаются тем, с какой точки зрения рассматривается тестируемое устройство.

*Метод «черного ящика»* заключается в рассмотрении устройства, принимающего входные данные и выдающего некий результат, при этом считается, что о внутреннем устройстве ничего неизвестно.

*Метод «белого ящика»* является обратным методу черного ящика и заключается в глубоком анализе внутреннего устройства прибора. В том числе, к примеру, происходит изучение кодов программного обеспечения.

*Метод «серого ящика»* включает в себя достоинства методов «черного» и «белого ящика», при данном методе тестирования также проверяется непосредственно корректное совпадение входных данных и выходных значений, а также известно внутреннее устройство прибора, что позволяет сконструировать «особые» тестовые случаи, которые, возможно, выявят дефекты в работе устройства.

Для эффективного тестирования сетевого оборудования общепринято применяется именно третий метод, поскольку при помощи первого практически невозможно перебрать все варианты тестовых входных данных и выявить редкие случаи, в которых возможна ошибка. Использование второго метода требует больших трудозатрат, а также оставляет не устранённым человеческий фактор.

Исходя из общих принципов работы маршрутизатора, можно выделить потенциально «узкие места», на основании чего составить алгоритм тестирования и выделить множество значимых тестов. Такими параметрами могут оказаться такие ресурсы как объем памяти, выделяемый под таблицу маршрутизации, вычислительные мощности процессора, размер буфера для приема пакетов и пропускная способность шины.

Вместе именно эти характеристики устройства влияют на скорость прохождения сетевых пакетов через маршрутизатор, а значит, чтобы испытать реальную пропускную способность устройства целиком, нужно варьировать параметры сетевых пакетов, так, чтобы они исчерпывали какой-либо из перечисленных выше ресурсов маршрутизатора.

Так, количество пакетов, передаваемых на вход маршрутизатору за единицу времени будет исчерпывать пропускную способность шины и занимать процессорное время, отводимое на определение маршрута для пакета. Размер сетевых пакетов, передаваемых маршрутизатору, прямым образом влияет на исчерпание свободного места в очереди пакетов, а количество взаимодействующих узлов и сетей, к которым они принадлежат, будет влиять на размер таблицы маршрутизации, который очевидно ограничен, а также на скорость поиска маршрутных записей, что так же занимает процессорное время.

Таким образом, исчерпание ресурсов устройства достаточно взаимосвязано, а значит и варьировать влияющие на это параметры имеет смысл в совокупности. Проблема в том, что количество возможных вариантов конфигурации наборов пакетов, отправляемых на маршрутизатор очень велико, поэтому, было принято решения воспользоваться методами оптимизации на основе генетического алгоритма.

Стоит отметить, что выбранный способ тестирования не только реализует общепринятую методику тестирования коммутационного оборудования, но также выгодно дополняет ее большой степенью вариативности в силу использования генетического алгоритма.

К примеру, методика соответствует рекомендациям по тестированию сетевого оборудования, описанным в RFC 2544 [6]. Схема подключения, описанная в данном документе, используется в полной мере и изображена на рисунке 3.



Рисунок 3. Схема организации тестового стенда

Из схемы видно, что тестируемое устройство подключается к тестирующему узлу при помощи двух двунаправленных интерфейсов.

Кроме того, авторы рекомендуют при тестировании варьировать именно те параметры, которые были описаны выше, а именно:

* Размеры кадров (пакетов),
* Протокольные адреса (то есть эмулировать отправку пакетов с разными узлами отправления и назначения),
* Плотность пакетов (то есть количество пакетов в единицу времени)

Важно добавить, что в указанных рекомендациях так же содержится пункт о подаче наборов пакетов с непостоянной плотностью, в которых можно выделить «пики» активности. Такого распределения плотности сетевых пакетов и помогает достичь генетический алгоритм.

Рекомендованный способ измерения производительности устройства – вычисление времени задержки между отправкой и принятием пакета также реализован и является основным в тестирующей системе.

Используемое программное обеспечение и библиотеки

Тестирующий комплекс может быть развернут на компьютере под управлением операционной системы семейства Linux, с установленным Python 2.7, библиотеками Pyevolve и Scapy.

Разработка велась на компьютере с операционной системой Ubuntu 14.04 LTS, с установленными Python 2.7.6, Pyevolve 0.6rc и Scapy 2.1.1.

Библиотека Pyevolve

На данный момент существует множество способов реализации генетических алгоритмов. Самым быстрым и удобным способом для языка Python является библиотека Pyevolve [4].

Данная библиотека была разработана специально для создания полноценных генетических алгоритмов. Сама библиотека – лишь каркас, реализующий общую логику работы генетического алгоритма, но все же сильно упрощающий процесс разработки.

Для быстрого создания работающего генетического алгоритма авторы библиотеки предлагают реализовать только функциональную составляющую генетического алгоритма, как, например, программное представление особи или алгоритм скрещивания.

Всю общую и неизменную логику работы генетического алгоритма выполняет сама библиотека: генерирует популяции, производит отбор, скрещивания, мутации, проверяет критерий остановки эволюции и т.п.

Библиотека Pyevolve также предоставляет возможности логирования, выгрузки данных в различные типы хранилищ (база данных, электронная таблица), взаимодействия с текущей популяцией в интерактивном режиме.

Таким образом, библиотека Pyevolve предоставляет разработчику возможность сконцентрироваться на своей конкретной задаче, предоставляя удобный интерфейс для быстрой реализации требуемого генетического алгоритма и возлагая на себя всю «сервисную» работу.

Утилита Scapy

Scapy – утилита, написанная на языке Python, позволяющая пользователю посылать, просматривать, анализировать и генерировать сетевые пакеты.  В отличие от аналогичных утилит, утилита Scapy не ограничена только теми протоколами, пакеты которых она может генерировать. Фактически, она позволяет создавать любые сетевые пакеты, что обеспечивает требуемую для решения поставленной задачи гибкость.

Типичные примеры использования утилиты:

* Сканирование,
* Трассировка маршрута,
* Юнит-тестирование всевозможных сетевых функций,
* Различные виды сетевых атак.

Утилита Scapy, является наиболее подходящим инструментом, для создания сетевых пакетов, параметры которых генерируются в процессе работы генетического алгоритма, и полностью избавляет от необходимости самостоятельной реализации модуля создания сетевого пакета с многочисленными варьируемыми параметрами.

Разработанный программный комплекс формирует заголовки пакетов 2, 3 и 4 уровней модели OSI, поскольку именно на этих уровнях работает большая часть коммутационного оборудования, в частности – маршрутизаторы. Представитель 2 уровня – Ethernet, представители 3 уровня это IP и ICMP протоколы, а 4 уровня — UDP и TCP протоколы.

# Глава 2. Модель сети

Генетические алгоритмы

Первая схема генетического алгоритма, основанная на принципах естественного отбора Ч. Дарвина, впервые была предложена Джоном Холландом в 1975 году. А классическая схема генетического алгоритма была предложена чуть позже Дэвидом Голдбергом. Так как схема алгоритма заимствована из биологии, для названий процессов были сохранены биологические названия. Основные термины, используемые в генетических алгоритмах, применимо к условиям решения поставленной задачи, перечислены ниже.

*Хромосома* – упорядоченный набор некоторых данных, каждая отдельная порция данных называется в этом случае *геном.*

*Индивидуум (геном, особь)* – набор хромосом, составляющих вариант решения задачи оптимизации, которой занимается генетический алгоритм. В рамках данной работы особь состоит из одной хромосомы, поэтому далее понятия генома и хромосомы идентичны.

*Кроссинговер (кроссовер)* – операция, при которой две особи обмениваются генами.

*Мутация* – случайное изменение одного или нескольких генов хромосомы.

*Популяция* – совокупность особей.

*Пригодность (приспособленность) особи* – значение функции, вычисляемой для каждого индивидуума, соответствующее тому, насколько хорошо данный вариант решает поставленную задачу.

Классическая схема работы генетического алгоритма представлена на рисунке 4.



Рисунок 4. Схема генетического алгоритма

Критерием окончания работы алгоритма может служить исчерпание заранее указанного количества популяций, либо *схождение популяции*, это означает, что все особи популяции мало отличаются друг от друга, то есть находятся в области некоторого экстремума.

Моделирование

Так как целью данной работы является тестирование коммутационного оборудования, конечным результатом работы атомарной итерации генетического алгоритма должен быть некий набор сетевых пакетов, которые можно отправить на один из интерфейсов устройства и принять на другом.

Следовательно, вопрос состоит в том, каким именно образом генерировать такие последовательности пакетов. Общих подходов к решению задачи генерации пакетов два:

1. Создать модель генерации, основываясь на том, как выглядят обрабатываемые пакеты с точки зрения самого тестируемого устройства;
2. Создать процесс генерации на основе моделирования сети, имеющей возможность существовать в реальности и «подключить» к этой сети тестируемое устройство.

Для реализации разработанного программного комплекса был выбран второй вариант, поскольку он обладает рядом преимуществ, по сравнению с первым.

Во-первых, при реализации второго варианта, можно надеяться на помещение тестируемого устройства в условия, близкие к реальным условиям эксплуатации, ведь то, что устройство покажет негативные результаты тестов, которые не могут существовать в реальности, слабо может претендовать на результат качественного тестирования.

Во-вторых, основываясь на первом подходе, будет довольно сложно получить последовательность пакетов, в реальности являющуюся, например, неизвестным до сих пор вариантом атакующего воздействия. Во втором варианте такая ситуация является более вероятной.

Таким образом, одним из ключевых моментов данной работы является задача выявления таких параметров сети, которые будут изменяться в процессе работы генетического алгоритма, сохраняя при этом приближенность к параметрам реальных сетей. А также вопрос программного представления такой модели сети.

Итак, в качестве изменяемых параметров для моделирования сети были предложены:

Количество подсетей в моделируемой сети, а также типы этих сетей и их «положение» относительно тестируемого оборудования. То есть, некоторые из подсетей имеют доступ к устройству через один интерфейс (назовем его *«правым»*), а некоторые – через другой (назовем его *«левым»*).

Подсети, в зависимости от интерфейса, через который они имеют доступ к тестируемому устройству, будем называть соответственно *«левыми»* и *«правыми»*. Для наглядности, одна из возможных моделей изображена на рисунке 5.



Рисунок 5. Принципиальная схема расположения подсетей относительно устройства.

В качестве изменяемых параметров подсети были выбраны тип сети и ее расположение относительно тестируемого устройства («левая» или «правая»).

Для узлов моделируемой сети параметрами оказались принадлежность к какой-либо подсети, а также набор сетевых взаимодействий, или потоков, в которых участвует данный узел.

*Потоком* в данной работе назовем набор сетевых пакетов одного сетевого протокола, отправляемых между двумя узлами сети в обоих направлениях. Другими словами, в данной модели сети поток описывает некое сетевое взаимодействие, например, передачу файла, если протоколом данного потока является TCP, или, к примеру, голосовой вызов, если протоколом данного потока оказался UDP.

У потока предложенных изменяемых параметров гораздо больше:

* Длительность самого потока по времени,
* Направление отправки пакета (от первого узла ко второму, либо от второго к первому),
* Временя жизни сетевого пакета,
* Размер полезной нагрузки пакета,
* Время отправки пакета.

Нетрудно заметить, что значения большинства этих параметров распределены в реально существующих сетях неравномерно, вследствие чего в модель необходимо ввести для каждого, требующего этого изменяемого параметра, закон распределения вероятности случайной величины.

Закон распределения вероятностей случайной величины

*Случайной* называют величину (далее - СВ), которая в результате испытания примет одно и только одно возможное значение, наперед не известное и зависящее от случайных причин, которые заранее не могут быть учтены [1].

*Дискретной* называют СВ, которая принимает отдельные, изолированные воз­можные значения с определенными вероятностями [1].

Следует отметить, что в рамках данной работы все СВ считаются дискретными, с конечным числом возможных значений.

*Законом распределения* дискретной случайной величины называют соответствие между возможными значениями и их вероятностями [1]. Такой закон можно задать, например, в табличной форме.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *X* | *x1* | *x2* | *…* | *xn* |
| *p* | *p1* | *p2* | *…* | *pn* |

Таблица 1. Закон распределения СВ.

Приняв во внимание, что в одном испытании случайная величина принимает· одно и только одно возможное зна­чение, заключаем, что события образуют полную группу, следовательно, сумма вероят­ностей этих событий, т. е. сумма вероятностей второй строки таблицы, равна единице:

(1).

Для того чтобы воспользоваться возможностями получения некоторого случайного значения случайной величины требуется представить ее закон распределения в программном виде. С этой целью был предложен следующий подход. Для каждой СВ будем хранить:

* Тип значений данной СВ – целый или вещественный,
* Минимальное значение, которое может принимать данная СВ,
* Максимальное значение, которое может принимать данная СВ,
* Закон распределения данной СВ.

При этом программное представление первых трех пунктов очевидно, но для закона распределения было предложено использовать особую конструкцию для удобства проведения испытаний в дальнейшем. Представим набор пар вида где а и - минимальное и максимальное значение СВ соответственно. При этом Пример такого набора изображен на рисунке 6.

Рисунок 6. Графическое представление моделируемой конструкции.

Из рисунка наглядно видно, как получить результат испытания. Нужно случайным образом бросить точку на ось абсцисс, а затем, найти ближайшее от оси абсцисс пересечение с какой-либо горизонталью вертикальной прямой, проведенной через эту точку. Координата оси ординат полученной точки и будет результатом испытания.

Формально закон распределения случайной величины из такого набора можно получить следующим образом.

1. Отсортировать элементы набора по возрастанию .
2. Из каждой пары соседних элементов отсортированного набора , где получить столбец табличного представления закона распределения случайной величины, где значение вероятности - , значение, принимаемое СВ с такой вероятностью - .
3. Кроме того, к результирующей таблице необходимо добавить значение вероятности с принимаемым значением

Очевидно, что построенный закон полностью удовлетворяет равенству (1), а значит, является законом распределения некоторой случайной величины.

Программная реализация получения результата испытания при такой конструкции закона распределения становится довольно простой, что иллюстрирует листинг \*.

points = [[0.2, 42], [1.0, 9]]  # создание пар (вероятность, значение)

r = random.random() # генерирование вещественного числа из отрезка [0,1]

i = 0

# в данном цикле реализован «бросок» точки на отрезок и выбор

# соответствующего значения

while r > points[i][0]:

    i += 1

return points[i][1]  # возврат полученного значения

Листинг 1. Пример получения результата испытания.

В строке 1 задается набор пар, описывающий закон распределения. Важно отметить, что в таком наборе обязательно должен присутствовать элемент с , для удовлетворения ограничениям (равенство (1)). В строках 2 – 5 производится генерация случайного вещественного числа и поиск диапазона, в который это число попадает, с последующим нахождением результата текущего испытания. Важно отметить, что генерируемое число – равномерно распределенная СВ, а значит, вероятности возникновения тех или иных значений СВ, для которой ищется результат испытаний, строго совпадают соответствующим законом распределения.

В результате программное представление СВ является базовым классом FX со следующим набором методов и данных:

* Поле *v\_type*, содержащее в себе тип значений СВ – целый или вещественный,
* Поле *v\_from*, содержащее в себе минимальное возможное значение СВ,
* Поле *v\_to*, содержащее максимально возможное значение, принимаемое СВ.
* Массив *points*, который соответствует описанному выше набору пар.
* Метод *random()*, возвращающий результат независимого испытания для данной СВ.
* Вспомогательные методы для работы генетического алгоритма, о которых будет сказано ниже.

Поток

Понятие потока в данной работе введено главным образом для того, чтобы стал понятен процесс генерации сетевых пакетов на основе построенной модели. Программным представлением потока стал базовый класс Flow, представляющий собой программный интерфейс и содержащий в себе следующие данные и методы:

* Поля *node1* и *node2*, содержащие в себе номера узлов сети, которые связывает данный поток;
* Поле ftf содержащее в себе объект класса FX, который описывает СВ – время жизни потока;
* Поле fhf содержащее объект класса FX, описывает СВ – направление отправки пакетов;
* Для каждого из двух направлений – набор полей (fttl, flp, ftp) содержащий в себе СВ – время жизни пакета, СВ – длина полезной нагрузки (данных), СВ – время отправки пакета;
* Метод генерации полезной нагрузки generate\_l5(), данный метод принимает на вход один параметр – требуемую длину данных;
* Метод генерации набора пакетов generate(), параметры которых поддаются законам распределения соответствующих СВ. Входные параметры данного метода – время начала открытия потока между узлами t0 и объект класса Translator, единственная задача которого – преобразовывать виртуальные узлы модели в корректные ip-адреса.

В рамках разработанного программного комплекса данный интерфейс реализуют три класса – FlowTCP, FlowUDP и FlowICMP, которые соответствуют потокам с сетевыми протоколами ЕМЗ, UDP и ICMP.

Пример реализации ключевого метода generate() для класса FlowICMP представлен в листинге \*. Для двух других классов реализация данного метода весьма схожа с представленной реализацией.

def generate(self, translator, t0):

    ip1 = translator.node2ip[self.node1]  # получение адреса узла

    ip2 = translator.node2ip[self.node2]

    l3\_1 = IP(src=ip1, dst=ip2)  # создание IP-пакета

    l4\_1 = ICMP(type=self.type1)  # создание ICMP-пакета

    l3\_2 = IP(src=ip2, dst=ip1)

    l4\_2 = ICMP(type=self.type2)

    l34\_1 = l3\_1 / l4\_1  # помещение ICMP-пакета внутрь IP

    l34\_2 = l3\_2 / l4\_2

    params1 = {'ftp':self.ftp1,'flp':self.flp1,'fttl':self.fttl1}

    params2 = {'ftp':self.ftp2,'flp':self.flp2,'fttl':self.fttl2}

    seq = 0

    ack = 0

    packets = []

    t1 = t0 + self.ftf.random() # вычисление общей длительности потока

    t = t0

# пока текущее время меньше времени жизни потока

    while t < t1:

# вычисление направления отправки, в зависимости от которого

# выбираются используемые законы распределения

        if not self.fhf.random():

            l34 = l34\_1

            params = params1

# присвоение пакету порядкового номера в последовательности

            l34['ICMP'].seq = seq

            ack = seq

            seq += 1

        else:

            l34 = l34\_2

            params = params2

            l34['ICMP'].ack = ack

        tp = params['ftp'].random()  # получение времени отправки пакета

# генерирование полезной нагрузки

        l5 = self.generate\_l5(params['flp'].random())

        l34['IP'].time = t  # установка времени отправки пакета

# вычисление времени жизни пакета

        l34['IP'].ttl = params['fttl'].random()

# помещение полезной нагрузки в пакет

        p = l34 / l5

# добавление созданного пакета к результату

        packets.append(p)

# увеличение прошедшего времени

        t += tp

    return packets  # возвращаем сгенерированный набор пакетов

Листинг 2. Пример генерирования набора пакетов.

Из данного примера ясно видно, что ключевым моментом генерации пакетов является получение того или иного параметра пакета путем вызова метода random(), который отвечает за результат независимого испытания.

Подсети и узлы

Для полноценного моделирования работы сети, к которой «подключено» тестируемое устройство, требуется моделировать не только взаимодействия между узлами сети, но также каким-то образом представлять сами узлы, а также подсети, в которых эти узлы расположены.

Для каждой подсети в рамках данной работы были введены два параметра: тип сети и ее положение относительно тестируемого устройства.

Было выделено несколько типов подсетей: обычные сети, принадлежащие какому-либо классу (A, B, C, D или E), также особые вымышленные подсети, необходимые для того, чтобы обозначить некоторые сетевые явления, например, тип сети «multicast», который требуется для моделирования потока, связанного с широковещательным запросом. Хотя в современных компьютерных сетях сетевые классы уже вышли из употребления, в данной работе они используются для удобства генерации ip-адресов.

Программное представление каждой подсети достаточно тривиально: пара вида (тип, положение), к примеру, листинг \* демонстрирует задание параметров для двух сетей. Первая сеть класса A получает доступ к тестируемому устройству через «левый» интерфейс, вторая сеть – класса B «подключена» к устройству через «правый» интерфейс.

nets = [('a', 'l'), ('b', 'r')]

Листинг 3. Пример задания параметров подсетей.

Для узлов в моделируемой сети значимых параметров два: принадлежность к какой-либо сети и потоки, в образовании которых участвует данный сетевой узел. Так как второй параметр уже учтен при моделировании потока, программное представление узлов также становится тривиальной задачей, что демонстрирует листинг \*.

nets = [('a', 'l'), ('b', 'r')]  # сети задаются парой (положение, ранг)

nodes = [0, 1, 0]  # узлы задаются принадлежностью к номеру сети

Листинг 4. Программное представление узлов сети.

Элемент с индексом i в массиве nodes соответствует узлу с номером i создаваемой модели. А значение nodes[i] равняется номеру подсети, к которой данный узел принадлежит. Сети также нумеруются в соответствии с их индексом в массиве nets. В приведенном примере заданы две сети и три узла. Из сказанного выше следует, что в данной модели сети класса A, которая подключена к «левому» интерфейсу, принадлежит два узла с номерами 0 и 2.

Таким образом, была построена программная модель сети, к которой виртуально подключено тестируемое устройство, обладающая множеством характеристик, способных изменяться. Тем самым было создано пространство для работы генетического алгоритма по поиску таких сетевых конфигураций, в которых тестируемое устройство может проявить собственные уязвимости, либо предел своих возможностей.

# Глава 3. Реализация тестирующей программы

Как было показано выше, при помощи библиотеки Pyevolve, задача программирования генетического алгоритма сводится к представлению модели генома особи в программном виде, что уже было произведено, а также программированию четырех операций: произвольной инициализации генома, мутации генома, скрещивания двух особей и функции вычисления «жизнеспособности» индивидуума.

Геномом особи для работы генетического алгоритма стал класс NetworkGenome, инкапсулирующий в себе все составные части предложенной модели сети. А именно, данный класс содержит в себе набор подсетей, узлов и потоков, которые в совокупности представляют собой моделируемую сеть. Также данный класс содержит в себе требуемые для корректной работы библиотеки Pyevolve методы clone() и copy(), которые возвращают полный клон модели и копируют все свойства одной модели в другую соответственно.

Создание начальной популяции

Инициализация произвольной конфигурации сети довольно тривиальна, при выборе подхода «от общего к частному». В рамках разработанного комплекса такие сложные объекты как классы FX и Flow, обладают собственными методами random\_initialize(), каждый из которых возвращает произвольный объект требуемого класса. Поэтому, основная функция произвольной инициализации конфигурации сети состоит лишь из генерирования случайного числа подсетей, узлов в этих подсетях и потоков между этими узлами. Схема реализации такой функции представлена на рисунке 7.



Рисунок 7. Схема работы функции инициализации объекта класса NetworkGenome.

Скрещивание

Задача скрещивания двух геномов в рамках генетического алгоритма в общем случае выглядит как получение нового генома, с признаками обоих входных геномов или «родителей» – «отца» и «матери». В силу особенностей библиотеки Pyevolve, функция скрещивания должна возвращать не одного потомка, а сразу двоих – «брата» и «сестру», что не отменяет накладываемых на потомков ограничений.

Так как, очевидно, важнейшим свойством модели сети с точки зрения тестирования коммутационного оборудования являются потоки, присутствующие в данной модели, при скрещивании был выбран путь обратный последовательности произвольной инициализации и де-факто являющийся одноточечным кроссинговером.

Каждая родительская особь имеет собственный набор потоков, каждый из этих наборов делится на два в случайно выбранном месте, а затем, обмениваются получившимися частями. Общую схему работы кроссинговера иллюстрирует таблица 2.

|  |  |
| --- | --- |
| Родители | Потомки |
| [f1,f2 ...,fn-1,fn,…,fm] | [f1,f2 ...,fn-1,gn,…,gm] |
| [g1,g2 ...,gn-1,gn,…,gm] | [g1,g2 ...,gn-1,fn,…,fm] |

Таблица 2. Наборы потоков родителей и потомков,

fi и gi - некоторые объекты класса Flow

После того, как произошел обмен потоками, потомки наследуют также сети и узлы, которые участвуют в образовании соответствующих потоков. И, таким образом, реализуется операция скрещивания, при которой потомки получают рекомбинированные гены родителей. Реализация функции скрещивания приведена в листинге \*.

def network\_crossover(genome, \*\*args):  
# получение из аргументов первого родительского генома

g\_mom = args["mom"]

# получение из аргументов второго родительского генома  
 g\_dad = args["dad"]   
  
 sister = g\_mom.clone() # клонирование геномов для последующего   
 brother = g\_dad.clone() # преобразования копий  
 sister.resetStats()   
 brother.resetStats()  
  
 if random.randint(0, 1): # обмен законами распределения

# потоков с вероятностью 0.5  
 sister.fflow, brother.fflow = brother.fflow, sister.fflow  
 if random.randint(0, 1): # аналогичный обмен временем существования  
 sister.texp, brother.texp = brother.texp, sister.texp

# одноточечный кроссовер функций распределения  
 cross = random.randint(0, min(len(sister.flows), len(brother.flows)) - 2) if

min(len(sister.flows), (brother.flows)) > 2 else 0  
  
# формируем геном сестры  
 s\_flows = sister.flows[:cross] + brother.flows[cross:]  
 sister.nets, sister.nodes = translate\_nodes\_and\_nets(s\_flows, sister.nodes, brother.nodes, sister.nets, brother.nets, lambda x: 's' if x < cross else 'b')

sister.flows = s\_flows  
  
# формируем геном брата  
 b\_flows = brother.flows[:cross] + sister.flows[cross:]  
 brother.nets, brother.nodes = translate\_nodes\_and\_nets(b\_flows, sister.nodes, brother.nodes, sister.nets, brother.nets, lambda x: 'b' if x < cross else 's')

brother.flows = b\_flows  
  
 return sister, brother # возвращаем результат скрещивания

*Листинг* 5*. Функция скрещивания*

Как видно из вышеприведенного листинга, скрещивание происходит с помощью выбора «точки рассечения» хромосомы (переменная cross), затем особи обмениваются «хвостами» наборов потоков, длина которых и определяется выбором точки рассечения. Ключевым моментом скрещивания является правильное преобразование узлов и сетей родителей в соответствии с новым набором потоков. За это преобразование отвечает функция translate\_nodes\_and\_nets(), ее реализация приведена в листинге \*.

def translate\_nodes\_and\_nets(flows, sister\_nodes, brother\_nodes, sister\_nets, brother\_nets, lambda\_flag):

# создаем пустые списки для новых индексов  
 b\_nodes = []  
 b\_nets = []  
 node\_dictionary = []  
  
# транслируем узлы в новые  
 for i in xrange(len(flows)):  
 flag = lambda\_flag(i) # вычисляем флаг с помощью переданной функции

# проверяем, добавлен ли уже узел в словарь соответствий, если нет, добавляем  
 if [flows[i].node1, flag] not in node\_dictionary:  
 node\_dictionary.append([flows[i].node1, flag])

# производим преобразование индекса узла в соответствующем поле потока  
 flows[i].node1 = node\_dictionary.index([flows[i].node1, flag])

# аналогичные операции для другого узла потока  
 if [flows[i].node2, flag] not in node\_dictionary:  
 node\_dictionary.append([flows[i].node2, flag])  
 flows[i].node2 = node\_dictionary.index([flows[i].node2, flag])  
  
 net\_dictionary = []

# копируем сети, к которым принадлежат узлы  
 for i in xrange(len(node\_dictionary)):  
 flag = node\_dictionary[i][1]  
 nodes = sister\_nodes if flag == 's' else brother\_nodes  
 nets = sister\_nets if flag == 's' else brother\_nets  
 old\_index = nodes[node\_dictionary[i][0]]  
 net = nets[old\_index]  
 if [old\_index, flag] not in net\_dictionary:  
 net\_dictionary.append([old\_index, flag])  
 b\_nets.append(net)  
 b\_nodes.append(net\_dictionary.index([old\_index, flag]))

# возвращаем наборы с преобразованными индексами  
 return b\_nets, b\_nodes

*Листинг* 6*. Функция преобразования родительских узлов и сетей*

*в узлы и сети потомков*

На вход функция translate\_nodes\_and\_nets() получает полную информацию об особях, участвующих в скрещивании и на основании этих данных преобразует индексы узлов и сетей в новые, а также проводит соответствующие изменения в потоках. Для этого в самом начале все потоки помечаются флагом ‘s’ или ‘b’, в зависимости от того, к какой особи они принадлежат (брат или сестра). Далее, в зависимости от установленного флага, выбираются требуемые узлы с параллельным построением словаря, хранящего соответствия новых индексов старым, после чего возвращаются результирующие наборы сетей и узлов с преобразованными индексами. Также в процессе преобразования индексов узлов происходит соответствующее преобразование в объектах потоков.

Мутация

Как известно, оператор мутации требуется в генетических алгоритмах для предотвращения схождения эволюции к точке локального экстремума, которая может оказаться не самым оптимальным решением задачи.

В рамках текущей задачи, например, требуется с помощью оператора мутации предотвратить схождение популяции к моделям, просто насыщенным большими объемами передаваемых данных, что по существу есть не что иное, как обыкновенное нагрузочное тестирование.

В целом в моделируемой сети может изменяться одно из трех множеств объектов: множество подсетей, множество узлов или множество потоков. При этом изменение одного множества может повлечь за собой изменение двух других.

Библиотека Pyevolve предоставляет разработчику использовать несколько операторов мутации одновременно, при этом можно выбрать – использовать ли их все, либо использовать каждый раз один произвольный. Наиболее удобным способом явилась реализация трех различных функций мутации для каждого из вышеперечисленных множеств. Логика работы этих операторов довольно схожа: в каждом множестве может мутировать или удалиться существующий элемент, либо добавиться новый. Пример реализации функции мутации множества потоков приведен в листинге \*.

def flow\_mutator(genome, \*\*args):

# произвольный выбор дальнейшей манипуляции над набором

# если выбранное число – индекс массива

# вызываем мутацию соответствующего элемента

    choice = random.randint(0, len(genome.flows) + 1)

    if choice < len(genome.flows):

        genome.flows[choice].mutation()

# если число равно длине массива создаем новый элемент

    elif choice == len(genome.flows):

        genome.flows.append(random\_flow(random.randint(0,len(genome.nodes)1), random.randint(0, len(genome.nodes) - 1)))

# в последнем случае удаляем существующий элемент

    else:

   del genome.flows[random.randint(0,len(genome.flows)-1)]

return 1 # возвращаем количество произведенных мутаций

Листинг 7. Функция мутации множества потоков.

В данном листинге переменная genome – это объект класса NetworkGenome, то есть некоторая модель сети. В строке 2 генерируется решение о том, какое именно изменение произойдет с набором потоков. В строке 4 происходит мутация существующего потока, если выбор пал на это действие. В строках 6-7 происходит создание нового произвольного потока и присоединение его к существующему набору. В строке 10 – удаление существующего потока при соответствующем выборе.

Пригодность

При выборе главного критерия тестирования выбор пал на измерение времени задержки между отправкой и приемом пакета тестирующей системой. Данный параметр легко преобразуется при надобности в пропускную способность маршрутизатора и полученное значение можно сравнить со значением, заявленным производителем устройства.

Для того чтобы протестировать некоторую особь используется набор объектов типа Flow, принадлежащих данной модели сети. Для каждого такого объекта вызывается функция get\_network\_packs(), которая отвечает за разделение пакетов на два множества – «левые» и «правые» пакеты. Принадлежность к какому-либо из этих множеств зависит от положения сети, из которой отправлен пакет относительно тестируемого устройства. Это требуется для того, чтобы в дальнейшем отправить пакеты на правильный интерфейс тестируемого устройства. Реализация описанного метода приведена в листинге \*.

def get\_network\_packs(genome):

# пустые списки, в которых будут аккумулироваться новые пакеты  
 left = []  
 right = []

# данный объект необходим для преобразования индексов улов в IP-адреса  
 translator = Translator(genome.nets, genome.nodes)  
 for f in genome.flows:

# для каждого потока генерируем набор пакетов  
 packs = f.generate(translator, 0)  
 for p in packs:  
 del p.chksum  
 p.src = p['IP'].src  
 p.dst = p['IP'].dst

# добавляем левые пакеты к аккумулирующему списку  
 left.extend([p for p in packs if translator.ip2pos[p['IP'].src] == 'l'])

# аналогично поступаем с правыми пакетами  
 right.extend([p for p in packs if translator.ip2pos[p['IP'].src] == 'r'])

# сортируем пакеты по времени отправки  
 left.sort(key=lambda pack: pack['IP'].time)  
 right.sort(key=lambda pack: pack['IP'].time)  
  
 return left, right

Листинг 8. Формирование двух наборов пакетов для тестирования

Далее полученные наборы пакетов отправляются на тестируемое устройство. Реализация основного метода тестирования особи приведена в листинге \*.

def network\_packets\_count\_tester(genome):  
 left, right = get\_network\_packs(genome)  
 if len(left) > 0:  
 send(left, iface = 'eth0')

if len(right) > 0:  
 send(right, iface = 'eth1')

# генерируем имя генома с уникальным идентификатором

name = """/home/tmp/""" + str(uuid.uuid1())

# сохраняем строковое представление генома в файл для дальнейшего анализа  
 f = open(name, 'w')  
 f.write(str(genome))  
 f.close()

# сохраняем сгенерированные пакеты для возможного дальнейшего анализа  
 if len(left) > 0:  
 wrpcap(name + '--left' + '.cap', left)  
 if len(right) > 0:  
 wrpcap(name + '--right' + '.cap', right)  
 return len(left) + len(right)

*Листинг* 9*. Функция отправки пакетов на тестируемое устройство*

Так как методы обработки очереди пакетов маршрутизатором различны, могут возникать случаи, когда пакеты будут отправлены на тестируемое устройство в одном порядке, а приняты будут в другом. Поэтому, для корректного изменения времени задержки пакета в тестируемом устройстве применяется механизм маркирования пакетов. Данный механизм заключается в помещении в начало полезной нагрузки сетевого пакета уникального идентификатора типа GUID. После приема пакетов от маршрутизатора легко установить соответствие между отправленными и принятыми пакетами и вычислить то самое время задержки пакета в устройстве. Механизм маркирования заключен в функции генерирования полезной нагрузки класса Flow и приведен в листинге 10.

def generate\_l5(length):

# функция стандартной библиотеки uuid1()

# генерирует уникальный идентификатор  
 l5 = str(uuid.uuid1()) + 'A' \* length  
 return l5

*Листинг* 10*. Маркирование пакета.*

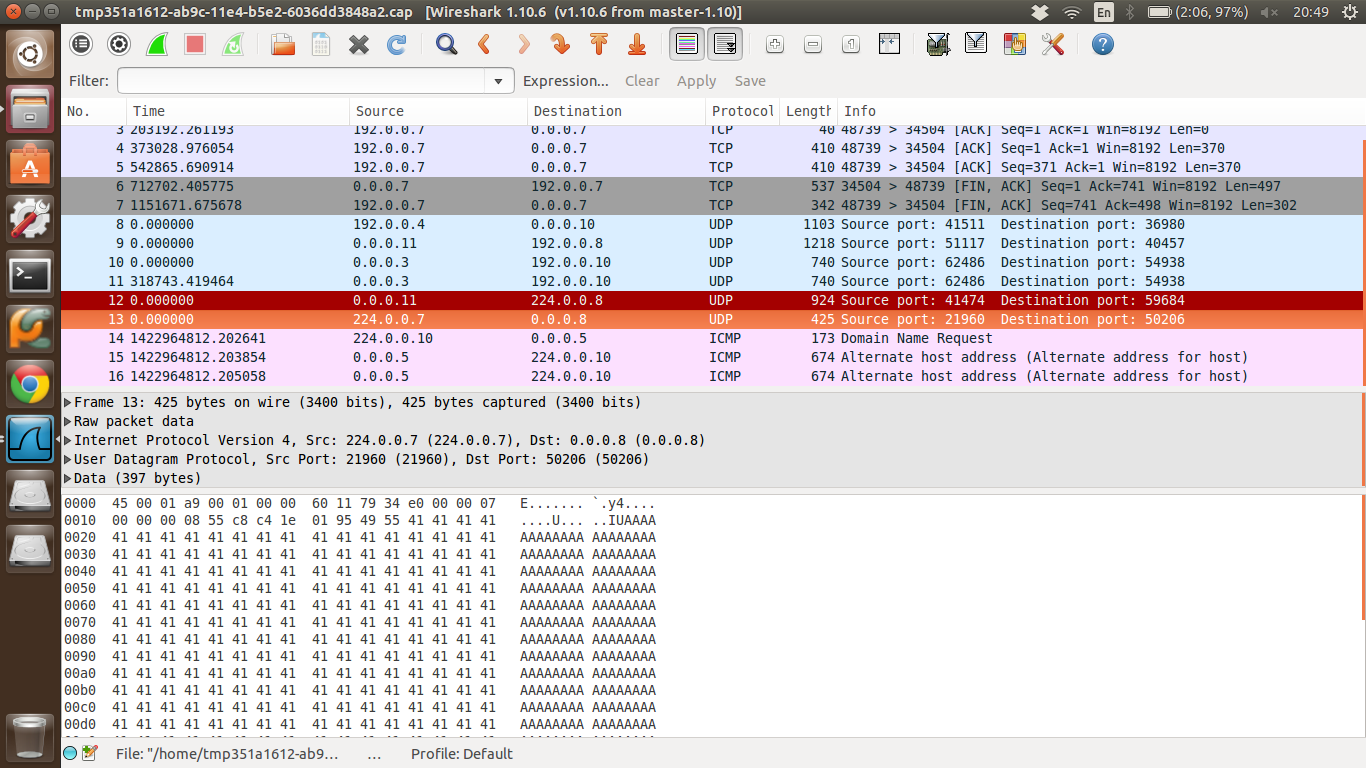
Динамическая маршрутизация

Как было упомянуто выше, одним из тестируемых параметров оборудования является скорость взаимодействия устройства с собственной таблицей маршрутизации. Очевидно, для автоматического тестирования данного параметра требуется применить методы динамической маршрутизации. Данные методы применяются в сетях со сложной и постоянно меняющейся структурой. Под данное определение подходит и весь разработанный цикл тестирования, в котором каждый единичный тест – некая топология сети, меняющаяся от теста к тесту.

В качестве протокола маршрутизации был выбран протокол RIP () – наиболее простой в реализации взаимодействий на основе утилиты Scapy представитель данного семейства протоколов. Функция, сообщающая устройству об изменениях топологии, приведена в листинге 11.

def route\_sender(nets, translator):  
 rip\_packs = []  
 entry\_count = 0  
 rp = RIP()  
 for i in xrange(len(nets)):

# в пакет помещается информация о маске и адресе сети,

# а также метрика = 1, чтобы данный маршрут точно был оптимальным**  
 rp = rp/RIPEntry(metric=1, mask=nets, addr=translator.net2ip[i])  
 entry\_count += 1  
# в один пакет можно поместить информацию лишь о 20 сетях  
# поэтому каждые 20 сетей создаем новый пакет  
 if entry\_count==20:  
 rip\_packs.append(rp)  
 rp = RIP()  
 entry\_count = 0  
# отправляем маршрутную информацию  
 send(rip\_packs, iface='eth0')

*Листинг* 11. Отправка маршрутной информации маршрутизатору.

Также стоит отметить, что в процессе реализации программы, для всех ключевых функций были реализованы модульные тесты, что было необходимо для уверенности в надежной работе программы. Модульное тестирование направлено на проверку корректной работы элементарных частей программы – функций и методов. Такое тестирование позволяет мгновенно локализовать ошибку в логике работы программы и оперативно устранить ее.

В рамках разработанной программы модульное тестирование было неотъемлемо. Так как время оценки одной сгенерированной топологии сети занимает несколько минут, а значит, продолжительность работы алгоритма довольно велика, в то время как любая ошибка может привести к неудачному эксперименту и впустую потраченному времени. Коды модульных тестов приведены в приложении.

# Глава 4. Эксперимент

Одной из задач работы программы является синтез сетевых пакетов для отправки их маршрутизирующему устройству. Убедиться в наличии генерируемых пакетов позволяет, к примеру, прослушивание эфира сети при помощи программы Wireshark. На рисунке 8 изображен снимок окна данной программы во время произведения тестирования.

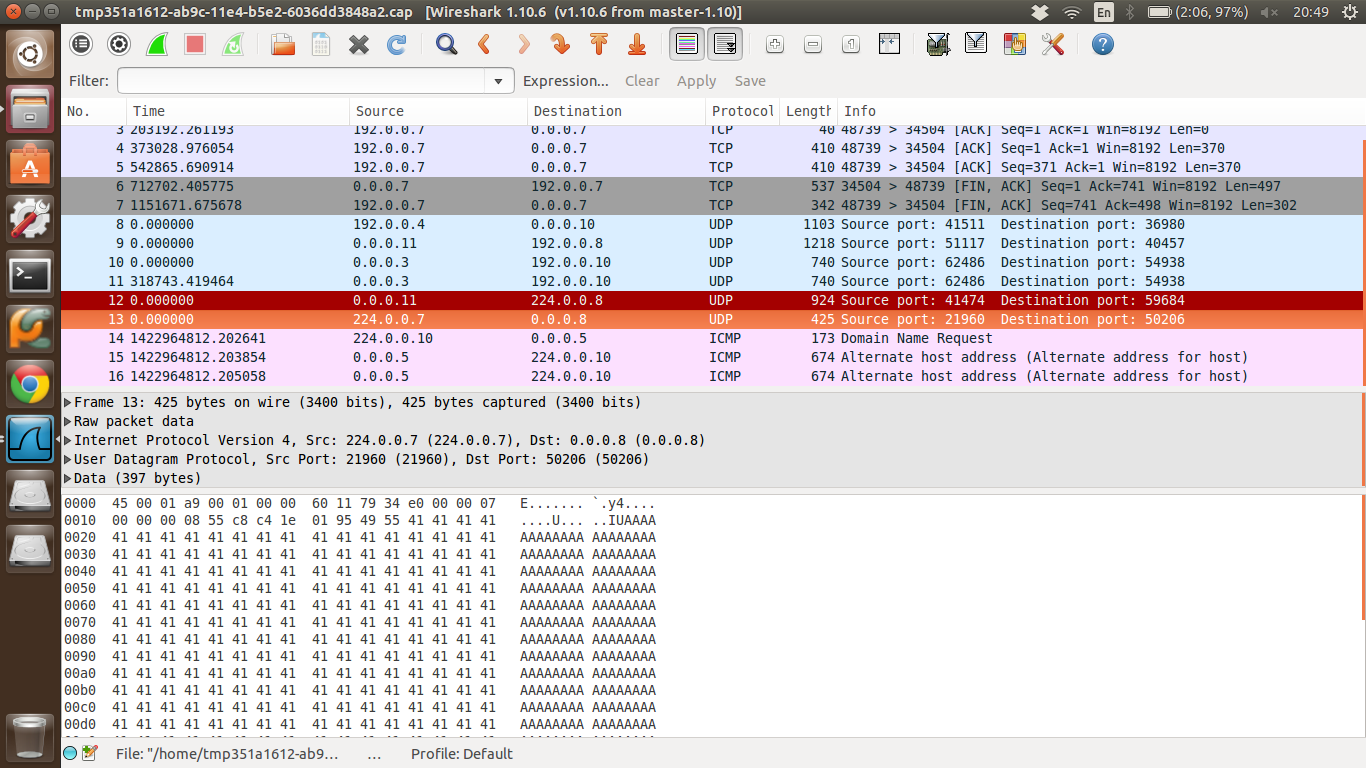


Рисунок 8. Пакеты, генерируемые разработанной программой

Данный рисунок демонстрирует основные особенности работы программы – генерацию пакетов трех различных протоколов (TCP, UDP или ICMP), эмуляцию сообщения между различными узлами сети. Также из шестнадцатеричного представления одного из пакетов видно, что в качестве полезной нагрузки используется «заметная» последовательность из символов «А».

Для корректной и оптимальной работы генетического алгоритма требуется установить верхний предел параметров конфигурации сети, с которыми работает реализованный генетический алгоритм. С этой целью был проведен ряд классических нагрузочных тестов.

Первый тест заключается в отправке на испытуемое устройство постоянно возрастающего количества пакетов в фиксированный промежуток времени (1 сек). При этом все пакеты имеют фиксированный размер (128 Кб), а также адрес отправителя и получателя. Данный тест направлен на выяснение того, как влияет процесс обработки заголовков пакетов на общую скорость маршрутизации.

Полученные результаты представлены на рисунке 9.

Рисунок 9. Зависимость задержки в передаче пакетов от количества пакетов

Результаты эксперимента дали неожиданный результат – в начале теста количество пакетов практически не влияет на производительность устройства, но при достижении уровня примерно 60000 пакетов производительность резко падает. Данный эксперимент выявляет характерный для множества домашних маршрутизаторов недостаток – плохие охлаждающие свойства. Таким образом, когда проводится довольно продолжительная серия тестов с передачей больших объемов информации, процессор устройства перегревается. Для того чтобы уберечь процессор от разрушения, его производительность автоматически снижается, что и влечет за собой резкое падение производительности. Также из данного графика видно, что предельным количеством пакетов, которые устройство может передать за единицу времени является примерно 85000, что указывает на приблизительный размер очереди пакетов маршрутизатора.

Второй тест заключается в отправке на устройство фиксированного количества пакетов фиксированного размера с постепенным увеличением количества различных адресатов, причем конечные узлы должны принадлежать разным подсетям для оказания влияния на скорость маршрутизации. По итогам проведенного теста был составлен график зависимости среднего времени задержки пакета в устройстве от количества маршрутизируемых сетей. Данный график представлен на рисунке 10.

Рисунок 10. Зависимость задержки в передаче пакетов от количества маршрутизируемых сетей

Резкое падение производительности наблюдается на отметке примерно в 1500 - 2000 сетей, что связано с исчерпанием объема буфера быстрого доступа маршрутизатора, который эффективно влияет на производительность лишь при небольшом количестве адресуемых сетей. Дальнейшее падение связано с увеличением таблицы маршрутизации, и, соответственно, увеличением времени, требуемым на ее просмотр.

# Заключение

В рамках данной работы был рассмотрен рекомендованный подход к тестированию сетевого оборудования, на основании которого были сформированы требования к тестирующему алгоритму. В результате был создан программный комплекс, выполняющий задачи по проведению автоматического тестирования оборудования.

Созданная программа генерирует тестовые наборы данных и выделяет те из них, которые оказывают негативное влияние на работу устройства, то есть замедляют процесс обработки принятых устройством пакетов.

Данный метод тестирования соответствует общепринятым стандартам, а также имеет дополнительные преимущества в виде полной автоматизированности, что исключает влияние человеческого фактора, а также большого числа тестовых испытаний при сравнительно небольших временных затратах, что обеспечивается использованием генетического алгоритма.

В качестве дальнейшего направления исследования в данной области может быть выбран эмпирический поиск самых оптимальных параметров работы генетического алгоритма, а также расширение реализованного списка используемых сетевых протоколов.

# Список использованной литературы

1. Гмурман В. Е. Теория вероятностей и математическая статистика: Учеб. пособие для вузов / 9-е изд., стер. — М.: ВЫСШ. ШХ., 2003. — 479 с., ил.
2. Лутц М. Изучаем Python, 4-е издание. / Пер. с англ. – Санкт-Петербург: Символ-Плюс, 2011. – 1280 с., ил.
3. Маршрутизация информации [Электронный ресурс] – Режим доступа: <http://reis.rtf.urfu.ru/portal/prime/net/cisco/index.htm>, свободный.
4. Панченко Т. В. Генетические алгоритмы [Текст]: учебно-методическое пособие / под ред. Ю. Ю. Тарасевича. — Астрахань: Издательский дом «Астраханский университет», 2007. — 87 с.
5. Сегаран Т. Программируем коллективный разум. / Пер. с англ. – Санкт-Петербург: Символ-Плюс, 2008. – 368 с., ил.
6. Bradner S. and McQuaid J. Benchmarking Methodology for Network Interconnect Devices [Электронный ресурс] – Режим доступа: <https://www.ietf.org/rfc/rfc2544.txt>, свободный
7. Christian S. Perone. Pyevolve Documentation Release 0.5 [Электронный ресурс] – Режим доступа: http://pyevolve.sourceforge.net/latex\_0\_5.pdf, свободный.
8. Philippe Biondi and the Scapy community. Scapy v2.1.1-dev. documentation [Электронный ресурс] – Режим доступа: <http://www.secdev.org/projects/scapy/doc/>, свободный.

# Приложение

Исходные коды модульных тестов

from unittest import TestCase  
  
from scapy.all import \*  
from scapy.layers.inet import IP, UDP, TCP, ICMP  
  
from flow import FlowUDP, FlowTCP, FlowICMP, FlowSock  
from fx import \*  
from genetic\_engine import NetworkGenome, network\_initializer, translate\_nodes\_and\_nets, delete\_node, network\_mutator, \  
 node\_mutator  
from nets\_manager import Translator  
  
  
class TestFX(TestCase):  
 def test\_random(self):  
 f = FX(1, 100, int, [[0.2, 42], [1.0, 9]])  
 counts = {}  
 for i in xrange(10000):  
 r = f.random()  
 if r in counts.keys():  
 counts[r] += 1  
 else:  
 counts[r] = 1  
 assert len(counts) == 2  
 assert 0.23 < float(counts[42]) / counts[9] < 0.27  
  
 def test\_mutation(self):  
 f = FX(1, 100, int, [[0.2, 42], [1.0, 9]])  
 old\_points = []  
 for p in f.points:  
 old\_points.append(p[:])  
 f.mutation()  
 success = False  
 if len(f.points) != len(old\_points):  
 success = True  
 else:  
 for i in xrange(len(f.points)):  
 if (f.points[i][0] != old\_points[i][0]) or (f.points[i][1] != old\_points[i][1]):  
 success = True  
 if not (0.0 <= f.points[i][0] <= 1.0):  
 raise ValueError("Некорректное изменение вероятности")  
 if not (f.v\_from <= f.points[i][1] <= f.v\_to):  
 raise ValueError("Некорректное изменение значения")  
 if f.points[-1][0] != 1.0:  
 raise ValueError("Последняя точка всегда 1.0")  
 assert success  
  
 def test\_copy(self):  
 ftp = FTP([[0, 0.075]])  
 ftp2 = FTP([[1, 0.1]])  
 ftp.copy(ftp2)  
 assert ftp2.points[0][1] == 0.075  
 ftp2.points[0][1] = 3  
  
 assert ftp.points[0][1] == 0.075  
 assert ftp.v\_delta == ftp2.v\_delta  
  
 def test\_clone(self):  
 ftp = FTP([[0.5, 0.1], [1.0, 0.05]])  
 ftp2 = ftp.clone()  
 assert isinstance(ftp2, FTP)  
  
  
class TestTranslator(TestCase):  
 def test\_ip\_generate(self):  
 nets = [(24, 'l'), (8, 'r')]  
 nodes = [0, 1]  
 t = Translator(nets, nodes)  
 assert len(t.node2ip[0].split('.')) == 4  
 assert t.node2pos[1] == 'r'  
 assert t.ip2pos[t.node2ip[0]] == 'l'  
 pat = re.compile("\d{1,3}.\d{1,3}.\d{1,3}.\d{1,3}")  
 assert pat.match(t.node2ip[1])  
  
  
class TestFlowSock(TestCase):  
 def test\_copy(self):  
 ftp = FTP([[1.0, 0.1]])  
 flp = FLP([[1.0, 100]])  
 fttl = FTTL([[1.0, 1]])  
 ftf = FTF([[1.0, 100]])  
 fhf = FHF([[0.5, 1]])  
 f = FlowSock(9999, 42, 0, 1, ftp, flp, fttl, ftp, flp, fttl, ftf, fhf)  
 g = FlowSock(19, 20, 2, 3, ftp, flp, fttl, ftp, flp, fttl, ftf, fhf)  
 f.copy(g)  
 assert g.port1 == 9999  
 g.fhf.points = []  
 assert len(f.fhf.points) > 0  
  
 def test\_clone(self):  
 ftp = FTP([[1.0, 0.1]])  
 flp = FLP([[1.0, 100]])  
 fttl = FTTL([[1.0, 1]])  
 ftf = FTF([[1.0, 100]])  
 fhf = FHF([[0.5, 1]])  
 f = FlowSock(9999, 42, 0, 1, ftp, flp, fttl, ftp, flp, fttl, ftf,fhf)  
 g = f.clone()  
 f.port1 = 12  
 assert g.port1 == 9999  
 g.fhf.points = []  
 assert len(f.fhf.points) > 0  
  
  
class TestFlowUdp(TestCase):  
 def test\_generate(self):  
 ftp = FTP([[1.0, 0.1]])  
 flp = FLP([[1.0, 100]])  
 fttl = FTTL([[1.0, 1]])  
 ftf = FTF([[1.0, 100]])  
 fhf = FHF([[0.5, 1]])  
 f = FlowUDP(9999, 42, 0, 1, ftp, flp, fttl, ftp, flp, fttl, ftf, fhf)  
  
 nets = [(8, 'l'), (16, 'r')]  
 nodes = [0, 1]  
 t = Translator(nets, nodes)  
  
 packs = f.generate(translator=t, t0=0)  
 assert len(packs) > 0  
 for p in packs:  
 assert isinstance(p, IP)  
 assert isinstance(p.payload, UDP)  
  
  
class TestFlowTCP(TestCase):  
 def test\_generate(self):  
 ftp = FTP([[1.0, 0.1]])  
 flp = FLP([[1.0, 100]])  
 fttl = FTTL([[1.0, 1]])  
 ftf = FTF([[1.0, 100]])  
 fhf = FHF([[0.5, 1]])  
 f = FlowTCP(9999, 42, 0, 1, ftp, flp, fttl, ftp, flp, fttl, ftf, fhf)  
  
 nets = [(8, 'l'), (16, 'r')]  
 nodes = [0, 1]  
 t = Translator(nets, nodes)  
  
 packs = f.generate(translator=t, t0=0)  
 assert len(packs) > 0  
 for p in packs:  
 assert isinstance(p, IP)  
 assert isinstance(p.payload, TCP)  
  
  
class TestFlowICMP(TestCase):  
 def test\_generate(self):  
 ftp = FTP([[1.0, 0.1]])  
 flp = FLP([[1.0, 100]])  
 fttl = FTTL([[1.0, 1]])  
 ftf = FTF([[1.0, 100]])  
 fhf = FHF([[0.5, 1]])  
 f = FlowICMP(0, 8, 0, 1, ftp, flp, fttl, ftp, flp, fttl, ftf, fhf)  
  
 nets = [(8, 'l'), (16, 'r')]  
 nodes = [0, 1]  
 t = Translator(nets, nodes)  
  
 packs = f.generate(translator=t, t0=0)  
 assert len(packs) > 0  
 for p in packs:  
 assert isinstance(p, IP)  
 assert isinstance(p.payload, ICMP)  
  
 def test\_clone(self):  
 ftp = FTP([[1.0, 0.1]])  
 flp = FLP([[1.0, 100]])  
 fttl = FTTL([[1.0, 1]])  
 ftf = FTF([[1.0, 100]])  
 fhf = FHF([[0.5, 1]])  
 f = FlowICMP(0, 1, 0, 1, ftp, flp, fttl, ftp, flp, fttl, ftf, fhf)  
 g = f.clone()  
 f.type1 = 12  
 assert g.type1 == 0  
 g.fhf.points = []  
 assert len(f.fhf.points) > 0  
  
  
class TestNetworkGenome(TestCase):  
 def test\_clone(self):  
 fflow = FFlow([[0.1, 1], [0.3, 2], [0.5, 3], [1.0, 4]])  
  
 ftp = FTP([[0.1, 0.01], [0.2, 0.02], [0.8, 0.04], [1.0, 0.06]])  
 flp1 = FLP([[0.1, 110], [0.3, 220], [0.5, 330], [1.0, 440]])  
 flp2 = FLP([[0.1, 110], [0.3, 220], [0.7, 330], [1.0, 440]])  
 fttl = FTTL([[0.1, 0], [0.3, 5], [0.5, 15], [1.0, 25]])  
 ftf = FTF([[0.2, 10], [0.3, 20], [0.6, 30], [1.0, 40]])  
 fhf = FHF([[0.5, 1]])  
  
 f1 = FlowUDP(9995, 42, 0, 1, ftp, flp1, fttl,ftp,flp2,fttl, ftf, fhf)  
 f2 = FlowUDP(9999, 40, 0, 2, ftp, flp2, fttl,ftp,flp1,fttl, ftf, fhf)  
 f3 = FlowTCP(123, 456, 1, 2, ftp, flp1, fttl,ftp,flp2,fttl, ftf, fhf)  
 f4 = FlowTCP(8899, 9800, 2, 0, ftp, flp2,fttl,ftp,flp1,fttl,ftf, fhf)  
  
 flows = [f1, f2, f3, f4]  
 nets = [(8, 'l'), (16, 'r'), (8, 'r')]  
 nodes = [0, 1, 2, 0]  
  
 o1 = NetworkGenome(nets, nodes, flows, fflow, 42.0)  
 o2 = o1.clone()  
  
 assert len(o2.nets) == 3  
 o2.nets[1] = (16, 'l')  
 assert o1.nets[1][1] == 'r'  
  
 def test\_network\_initializer(self):  
 net = network\_initializer(None)  
 assert isinstance(net, NetworkGenome)  
  
 def test\_translate\_nodes\_and\_nets(self):  
  
 ftp = FTP([[0.1, 0.010], [0.2, 0.020], [0.8, 0.040], [1.0, 0.060]])  
 flp1 = FLP([[0.1, 110], [0.3, 220], [0.5, 330], [1.0, 440]])  
 flp2 = FLP([[0.1, 110], [0.3, 220], [0.7, 330], [1.0, 440]])  
 fttl = FTTL([[0.1, 0], [0.3, 5], [0.5, 15], [1.0, 25]])  
 ftf = FTF([[0.2, 10], [0.3, 20], [0.6, 30], [1.0, 40]])  
 fhf = FHF([[0.5, 1]])  
 f1 = FlowUDP(9995, 42, 0, 1, ftp, flp1,fttl, ftp,flp2,fttl, ftf, fhf)  
 f2 = FlowUDP(9999, 40, 1, 0, ftp, flp2,fttl, ftp,flp1,fttl, ftf, fhf)  
 f3 = FlowTCP(123, 456, 0, 1, ftp, flp1,fttl, ftp,flp2,fttl, ftf, fhf)  
 f4 = FlowTCP(8899, 9800, 1, 0, ftp, flp2,fttl,ftp,flp1,fttl,ftf, fhf)  
  
 flows = [f1, f2, f3, f4]  
 nodes = [0, 1]  
 s\_nets = [('a', 'l'), ('a', 'l')]  
 b\_nets = [('b', 'r'), ('b', 'r')]  
 res\_nets, res\_nodes = translate\_nodes\_and\_nets(flows, nodes, nodes, s\_nets, b\_nets, lambda x: 's' if x < 2 else 'b')  
 assert len(res\_nets) == 4  
 assert len(res\_nodes) == 4  
  
 def test\_delete\_node(self):  
 fflow = FFlow([[0.1, 1], [0.3, 2], [0.5, 3], [1.0, 4]])  
 ftp = FTP([[0.1, 0.01], [0.2, 0.020], [0.8, 0.040], [1.0, 0.060]])  
 flp1 = FLP([[0.1, 110], [0.3, 220], [0.5, 330], [1.0, 440]])  
 flp2 = FLP([[0.1, 110], [0.3, 220], [0.7, 330], [1.0, 440]])  
 fttl = FTTL([[0.1, 0], [0.3, 5], [0.5, 15], [1.0, 25]])  
 ftf = FTF([[0.2, 10], [0.3, 20], [0.6, 30], [1.0, 40]])  
 fhf = FHF([[0.5, 1]])  
 f1 = FlowUDP(9995, 42, 0, 1,ftp,flp1,fttl,ftp, flp2, fttl, ftf, fhf)  
 f2 = FlowUDP(9999, 40, 0, 2,ftp,flp2,fttl,ftp, flp1, fttl, ftf, fhf)  
 f3 = FlowTCP(123, 456, 1, 2,ftp,flp1,fttl, ftp, flp2, fttl, ftf, fhf)  
 f4 = FlowTCP(8899, 9800, 2, 0,ftp,flp2,fttl,ftp,flp1, fttl, ftf, fhf)  
 flows = [f1, f2, f3, f4]  
 nets = [(8, 'l'), (16, 'r'), (8, 'r')]  
 nodes = [0, 1, 2, 0]  
 o1 = NetworkGenome(nets, nodes, flows, fflow, 42.0)  
 delete\_node(o1, 2)  
 assert len(o1.nodes) == 3  
 # assert len(o1.nets) == 2  
 assert len(o1.flows) == 1  
  
 def test\_network\_mutator(self):  
 fflow = FFlow([[0.1, 1], [0.3, 2], [0.5, 3], [1.0, 4]])  
 ftp = FTP([[0.1, 0.010], [0.2, 0.020], [0.8, 0.040], [1.0, 0.060]])  
 flp1 = FLP([[0.1, 110], [0.3, 220], [0.5, 330], [1.0, 440]])  
 flp2 = FLP([[0.1, 110], [0.3, 220], [0.7, 330], [1.0, 440]])  
 fttl = FTTL([[0.1, 0], [0.3, 5], [0.5, 15], [1.0, 25]])  
 ftf = FTF([[0.2, 10], [0.3, 20], [0.6, 30], [1.0, 40]])  
 fhf = FHF([[0.5, 1]])  
 f1 = FlowUDP(9995, 42, 0, 1, ftp,flp1,fttl,ftp,flp2,fttl,ftf,fhf)  
 f2 = FlowUDP(9999, 40, 0, 2, ftp,flp2,fttl, ftp,flp1,fttl,ftf,fhf)  
 f3 = FlowTCP(123, 456, 1, 2, ftp, flp1,fttl,ftp, flp2, fttl,ftf, fhf)  
 f4 = FlowTCP(8899, 9800, 2, 0, ftp, flp2,fttl,ftp,flp1,fttl,ftf, fhf)  
 flows = [f1, f2, f3, f4]  
 nets = [(8, 'l'), (16, 'r'), (8, 'r')]  
 nodes = [0, 1, 2, 0]  
 o1 = NetworkGenome(nets, nodes, flows, fflow, 42.0)  
 old\_nets = nets[:]  
 network\_mutator(o1)  
 assert len(o1.nets) != 3 or any(old\_nets[i] != o1.nets[i] for i in xrange(3))  
  
 def test\_node\_mutator(self):  
 fflow = FFlow([[0.1, 1], [0.3, 2], [0.5, 3], [1.0, 4]])  
 ftp = FTP([[0.1, 0.010], [0.2, 0.020], [0.8, 0.040], [1.0, 0.060]])  
 flp1 = FLP([[0.1, 110], [0.3, 220], [0.5, 330], [1.0, 440]])  
 flp2 = FLP([[0.1, 110], [0.3, 220], [0.7, 330], [1.0, 440]])  
 fttl = FTTL([[0.1, 0], [0.3, 5], [0.5, 15], [1.0, 25]])  
 ftf = FTF([[0.2, 10], [0.3, 20], [0.6, 30], [1.0, 40]])  
 fhf = FHF([[0.5, 1]])  
 f1 = FlowUDP(9995, 42, 0, 1,ftp,flp1,fttl,ftp, flp2, fttl, ftf, fhf)  
 f2 = FlowUDP(9999, 40, 0, 2, ftp,flp2,fttl,ftp,flp1, fttl, ftf, fhf)  
 f3 = FlowTCP(123, 456, 1, 2, ftp,flp1,fttl,ftp, flp2, fttl, ftf, fhf)  
 f4 = FlowTCP(8899, 9800, 2, 0, ftp,flp2,fttl,ftp,flp1,fttl, ftf, fhf)  
 flows = [f1, f2, f3, f4]  
 nets = [(8, 'l'), (16, 'r'), (8, 'r')]  
 nodes = [0, 1, 2, 0]  
 o1 = NetworkGenome(nets, nodes, flows, fflow, 42.0)  
 old\_nodes = nodes[:]  
 node\_mutator(o1)  
 assert len(o1.nodes) != 4 or any(old\_nodes[i] != o1.nodes[i] for i in xrange(4))