PYTHON ДЛЯ СЕТЕВЫХ ИНЖЕНЕРОВ

ОБРАБОТКА ВЫВОДА КОМАНД С TEXTFSM

ОБРАБОТКА ВЫВОДА КОМАНД С TEXTFSM

TextFSM это библиотека созданная Google для обработки вывода с сетевых устройств. Она позволяет создавать шаблоны, по которым будет обрабатываться вывод команды.

Использование TextFSM лучше, чем простая построчная обработка, так как шаблоны дают лучшее представление о том, как вывод будет обрабатываться и шаблонами проще поделиться. А значит, проще найти уже созданные шаблоны и использовать их. Или поделиться своими.

Для начала, библиотеку надо установить:

```
pip install textfsm
```

ОБРАБОТКА ВЫВОДА КОМАНД С TEXTFSM

Для использования TextFSM, надо создать шаблон, по которому будет обрабатываться вывод команды.

Пример вывода команды traceroute:

```
r2#traceroute 90.0.0.9 source 33.0.0.2
traceroute 90.0.0.9 source 33.0.0.2
Type escape sequence to abort.
Tracing the route to 90.0.0.9
VRF info: (vrf in name/id, vrf out name/id)
  1 10.0.12.1 1 msec 0 msec 0 msec
  2 15.0.0.5  0 msec 5 msec 4 msec
  3 57.0.0.7  4 msec 1 msec 4 msec
  4 79.0.0.9  4 msec *  1 msec
```

ОБРАБОТКА ВЫВОДА КОМАНД С TEXTFSM

Например, из вывода надо получить хопы, через которые прошел пакет.

В таком случае, шаблон TextFSM будет выглядеть так (файл traceroute.template):

```
Value ID (\d+)
Value Hop (\d+(\.\d+){3})

Start
  ^  ${ID} ${Hop} -> Record
```

ОБРАБОТКА ВЫВОДА КОМАНД С TEXTFSM

Первые две строки определяют переменные:

- Value ID (\d+)
  - эта строка определяет переменную ID, которая описывает регулярное выражение: (\d+) - одна или более цифр
  - сюда попадут номера хопов
- Value Hop (\d+(\.\d+){3})
  - эта строка определяет переменную Hop, которая описывает IP-адрес таким регулярным выражением: (\d+(\.\d+){3})

ОБРАБОТКА ВЫВОДА КОМАНД С TEXTFSM

После строки Start начинается сам шаблон. В данном случае, он очень простой:

- ^ ${ID} ${Hop} -> Record
  - сначала идет символ начала строки, затем два пробела и переменные ID и Hop
  - в TextFSM переменные описываются таким образом: ${имя переменной}
  - слово Record в конце означает, что строки, которые попадут под описанный шаблон, будут обработаны и выведены в результаты TextFSM (с этим подробнее мы разберемся в следующем разделе)

ОБРАБОТКА ВЫВОДА КОМАНД С TEXTFSM

Скрипт для обработки вывода команды traceroute с помощью TextFSM (parse_traceroute.py):

```
import textfsm

traceroute = """
r2#traceroute 90.0.0.9 source 33.0.0.2
traceroute 90.0.0.9 source 33.0.0.2
Type escape sequence to abort.
Tracing the route to 90.0.0.9
VRF info: (vrf in name/id, vrf out name/id)
  1 10.0.12.1 1 msec 0 msec 0 msec
  2 15.0.0.5  0 msec 5 msec 4 msec
  3 57.0.0.7  4 msec 1 msec 4 msec
  4 79.0.0.9  4 msec *  1 msec
"""

template = open('traceroute.textfsm')
fsm = textfsm.TextFSM(template)
result = fsm.ParseText(traceroute)

print(fsm.header)
print(result)
```

ОБРАБОТКА ВЫВОДА КОМАНД С TEXTFSM

Результат выполнения скрипта:

```
$ python parse_traceroute.py
['ID', 'Hop']
[['1', '10.0.12.1'], ['2', '15.0.0.5'], ['3', '57.0.0.7'], ['4', '79.0.0.9']]
```

Строки, которые совпали с описанным шаблоном, возвращаются в виде списка списков. Каждый элемент это список, который состоит из двух элементов: номера хопа и IP-адреса.

ОБРАБОТКА ВЫВОДА КОМАНД С TEXTFSM

Разберемся с содержимым скрипта:

- traceroute - это переменная, которая содержит вывод команды traceroute
- template = open('traceroute.textfsm') - содержимое файла с шаблоном TextFSM считывается в переменную template
- fsm = textfsm.TextFSM(template) - класс, который обрабатывает шаблон и создает из него объект в TextFSM
- result = fsm.ParseText(traceroute) - метод, который обрабатывает переданный вывод согласно шаблону и возращает список списков, в котором каждый элемент это обработанная строка
- В конце выводится заголовок: print(fsm.header), который содержит имена переменных
- И результат обработки

ОБРАБОТКА ВЫВОДА КОМАНД С TEXTFSM

Для работы с TextFSM нужны вывод команды и шаблон:

- для разных команд нужны разные шаблоны
- TextFSM возвращает результат обработки в табличном виде (в виде списка списков)
  - этот вывод легко преобразовать в csv формат или в список словарей

СИНТАКСИС ШАБЛОНОВ TEXTFSM

СИНТАКСИС ШАБЛОНОВ TEXTFSM

Шаблон TextFSM описывает каким образом данные должны обрабатываться.

Любой шаблон состоит из двух частей:

- определения переменных
  - эти переменные описывают какие столбцы будут в табличном представлении
- определения состояний

СИНТАКСИС ШАБЛОНОВ TEXTFSM

Пример разбора команды traceroute:

```
# Определение переменных:
Value ID (\d+)
Value Hop (\d+(\.\d+){3})

# Секция с определением состояний всегда должна начинаться с состояния Start
Start
 #   Переменные      действие
  ^  ${ID} ${Hop} -> Record
```

ОПРЕДЕЛЕНИЕ ПЕРЕМЕННЫХ

В секции с переменными должны идти только определения переменных. Единственное исключение - в этом разделе могут быть комментарии.

В этом разделе не должно быть пустых строк. Для TextFSM пустая строка означает завершение секции определения переменных.

Формат описания переменных:

```
Value [option[,option...]] name regex
```

ОПРЕДЕЛЕНИЕ ПЕРЕМЕННЫХ

Синтаксис описания переменных:

- Value - это ключевое слово, которое указывает, что создается переменная. Его обязательно нужно указывать
- option - опции, которые определяют как работать с переменной. Если нужно указать несколько опций, они должны быть отделены запятой, без пробелов.

ОПРЕДЕЛЕНИЕ ПЕРЕМЕННЫХ

Поддерживаются такие опции:

- **Filldown** - значение, которое ранее совпало с регулярным выражением, запоминается до следующей обработки строки (если не было явно очищено или снова совпало регулярное выражение).
  - это значит, что последнее значение столбца, которое совпало с регулярным выражением, запоминается и используется в следующих строках, если в них не присутствовал этот столбец.
- **Key** - определяет, что это поле содержит уникальный идентификатор строки
- **Required** - строка, которая обрабатывается, будет записана только в том случае, если эта переменная присутствует.

ОПРЕДЕЛЕНИЕ ПЕРЕМЕННЫХ

- name - имя переменной, которое будет использоваться как имя колонки. Зарезервированные имена не должны использоваться как имя переменной.
- regex - регулярное выражение, которое описывает переменную. Регулярное выражение должно быть в скобках.

ОПРЕДЕЛЕНИЕ СОСТОЯНИЙ

После определения переменных, нужно описать состояния:

- каждое определение состояния должно быть отделено пустой строкой (как минимум, одной)
- первая строка - имя состояния
- затем идут строки, которые описывают правила
  - правила должны начинаться с пробела и символа ^

ОПРЕДЕЛЕНИЕ СОСТОЯНИЙ

Начальное состояние всегда **Start**. Входные данные сравниваются с текущим состоянием, но в строке правила может быть указано, что нужно перейти к другому состоянию.

Проверка выполняется построчно, пока не будет достигнут **EOF**(конец файла) или текущее состояние перейдет в состояние **End**.

ЗАРЕЗЕРВИРОВАННЫЕ СОСТОЯНИЯ

Зарезервированы такие состояния:

- **Start** - это состояние обязательно должно быть указано. Без него шаблон не будет работать.
- **End** - это состояние завершает обработку входящих строк и не выполняет состояние **EOF**.
- **EOF** - это неявное состояние, которое выполняется всегда, когда обработка долшла до конца файла. Выглядит оно таким образом:

```
EOF
 ^.* -> Record
```

ЗАРЕЗЕРВИРОВАННЫЕ СОСТОЯНИЯ

EOF записывает текущую строку, прежде чем обработка завершается. Если это поведение нужно изменить, надо явно, в конце шаблона, написать EOF:

```
EOF
```

ПРАВИЛА СОСТОЯНИЙ

Каждое состояние состоит из одного или более правил:

- TextFSM обрабатывает входящие строки и сравнивает их с правилами
- если правило (регулярное выражение) совпадает со строкой, выполняются действия, которые описаны в правиле и для следующей строки процесс повторяется заново, с начала состояния.

Правила должны быть описаны в таком формате:

```
^regex [-> action]
```

ПРАВИЛА СОСТОЯНИЙ

В правиле:

- каждое правило должно начинаться с пробела и символа ^
  - символ ^ означает начало строки и всегда должен указываться явно
- regex - это регулярное выражение, в котором могут использоваться переменные
  - для указания переменной, может использоваться синтаксис $ValueName или ${ValueName}(этот формат предпочтителен)
  - в правиле, на место переменных подставляются регулярные выражения, которые они описывают
  - если нужно явно указать символ конца строки, используется значение $$

ДЕЙСТВИЯ В ПРАВИЛАХ

После регулярного выражения, в правиле могут указываться действия:

- между регулярным выражением и действием, должен быть символ ->
- действия могут состоять из трех частей, в таком формате **L.R S**
  - **L - Line Action** - действия, которые применяются к входящей строке
  - **R - Record Action** - действия, которые применяются к собранным значениям
  - **S - State Action** - переход в другое состояние
- если нет указанных действий, то по умолчанию используется действие **Next.NoRecord**.

LINE ACTIONS

Line Actions:

- **Next** - обработать строку, прочитать следующую и начать проверять её с начала состояния. Это действие используется по умолчанию, если не указано другое
- **Continue** - продолжить обработку правил, как-будто совпадения не было, при этом значения присваиваются

RECORD ACTION

Record Action - опциональное действие, которое может быть указано после Line Action. Они должны быть разделены точкой. Типы действий:

- **NoRecord** - не выполнять ничего. Это действие по умолчанию, когда другое не указано
- **Record** - запомнить значение, которые совпали с правилом. Все переменные, кроме тех, где указана опция Filldown, обнуляются.
- **Clear** - обнулить все переменные, кроме тех, где указана опция Filldown.
- **Clearall** - обнулить все переменные.

Разделять действия точкой нужно только в том случае, если нужно указать и Line и Record действия. Если нужно указать только одно из них, точку ставить не нужно.

STATE TRANSITION

После действия, может быть указано новое состояние:

- состояние должно быть одним из зарезервированных или состояние определенное в шаблоне
- если входная строка совпала:
  - все действия выполняются,
  - считывается следующая строка,
  - затем текущее состояние меняется на новое и обработка продолжается в новом состоянии.

Если в правиле используется действие **Continue**, то в нем нельзя использовать переход в другое состояние. Это правило нужно для того, чтобы в последовательности состояний не было петель.

ERROR ACTION

Специальное действие **Error** останавливает всю обработку строк, отбрасывает все строки, которые были собраны до сих пор и возвращает исключение.

Синтаксис этого действия такой:

```
^regex -> Error [word|"string"]
```

ПРИМЕРЫ ИСПОЛЬЗОВАНИЯ TEXTFSM

ПРИМЕРЫ ИСПОЛЬЗОВАНИЯ TEXTFSM

Для обработки вывода команд по шаблону используется скрипт parse_output.py. Он не привязан к конкретному шаблону и выводу: шаблон и вывод команды будут передаваться как аргументы:

```
import sys
import textfsm
from tabulate import tabulate

template = sys.argv[1]
output_file = sys.argv[2]

f = open(template)
output = open(output_file).read()

re_table = textfsm.TextFSM(f)

header = re_table.header
result = re_table.ParseText(output)

print(tabulate(result, headers=header))
```

ПРИМЕРЫ ИСПОЛЬЗОВАНИЯ TEXTFSM

Пример запуска скрипта:

```
$ python parse_output.py template command_output
```

Обработка данных по шаблону всегда выполняется одинаково. Поэтому скрипт будет одинаковый и только шаблон и данные отличаться.

SHOW CLOCK

Первый пример - разбор вывода команды sh clock (файл output/sh_clock.txt):

```
15:10:44.867 UTC Sun Nov 13 2016
```

Для начала, в шаблоне надо определить переменные:

- в начале каждой строки должно быть ключевое слово Value
  - каждая переменная определяет столбец в таблице
- следующее слово - название переменной
- после названия, в скобках, регулярное выражение, которое описывает значение переменной

SHOW CLOCK

Определение переменных выглядит так:

```
Value Time (..:..:..)
Value Timezone (\S+)
Value WeekDay (\w+)
Value Month (\w+)
Value MonthDay (\d+)
Value Year (\d+)
```

SHOW CLOCK

Подсказка по спецсимволам:

- . - любой символ
- + - одно или более повторений предыдущего символа
- \S - все символы, кроме whitespace
- \w - любая буква или цифра
- \d - любая цифра

SHOW CLOCK

После определения переменных, должна идти пустая строка и состояние **Start**, а после, начиная с пробела и символа ^, идет правило (файл templates/sh_clock.template):

```
Value Time (..:..:..)
Value Timezone (\S+)
Value WeekDay (\w+)
Value Month (\w+)
Value MonthDay (\d+)
Value Year (\d+)

Start
  ^${Time}.* ${Timezone} ${WeekDay} ${Month} ${MonthDay} ${Year} -> Record
```

SHOW CLOCK

Так как, в данном случае, в выводе всего одна строка, можно не писать в шаблоне действие Record. Но лучше его использовать в ситуациях, когда надо записать значения, чтобы привыкать к этому синтаксу и не ошибиться, когда нужна обработка нескольких строк.

Когда TextFSM обрабатывает строки вывода, он подставляет вместо переменных, их значения. В итоге правило будет выглядеть так:

```
^(..:..:..).* (\S+) (\w+) (\w+) (\d+) (\d+)
```

SHOW CLOCK

Когда это регулярное выражение применяется в выводу show clock, в каждой группе регулярного выражения, будет находиться соответствующее значение:

- 1 группа: 15:10:44
- 2 группа: UTC
- 3 группа: Sun
- 4 группа: Nov
- 5 группа: 13
- 6 группа: 2016

SHOW CLOCK

В правиле, кроме явного действия Record, которое указывает, что запись надо поместить в финальную таблицу, по умолчанию также используется правило Next. Оно указывает, что надо перейти к следующей строке текста. Так как в выводе команды sh clock, только одна строка, обработка завершается.

SHOW CLOCK

Результат отработки скрипта будет таким:

```
$ python parse_output.py templates/sh_clock.template output/sh_clock.txt
Time      Timezone    WeekDay    Month      MonthDay    Year
--------  ----------  ---------  -------  ----------  ------
15:10:44  UTC         Sun        Nov              13    2016
```

SHOW CDP NEIGHBORS DETAIL

Теперь попробуем обработать вывод команды show cdp neighbors detail.

Особенность этой команды в том, что данные находятся не в одной строке, а в разных.

SHOW CDP NEIGHBORS DETAIL

В файле output/sh_cdp_n_det.txt находится вывод команды show cdp neighbors detail:

```
SW1#show cdp neighbors detail
-------------------------
Device ID: SW2
Entry address(es):
  IP address: 10.1.1.2
Platform: cisco WS-C2960-8TC-L,  Capabilities: Switch IGMP
Interface: GigabitEthernet1/0/16,  Port ID (outgoing port): GigabitEthernet0/1
Holdtime : 164 sec

Version :
Cisco IOS Software, C2960 Software (C2960-LANBASEK9-M), Version 12.2(55)SE9, RELEASE SOFTWARE (fc1)
Technical Support: http://www.cisco.com/techsupport
Copyright (c) 1986-2014 by Cisco Systems, Inc.
Compiled Mon 03-Mar-14 22:53 by prod_rel_team

advertisement version: 2
VTP Management Domain: ''
Native VLAN: 1
Duplex: full
Management address(es):
  IP address: 10.1.1.2

-------------------------
Device ID: R1
Entry address(es):
  IP address: 10.1.1.1
Platform: Cisco 2825,  Capabilities: Router Switch IGMP
```

SHOW CDP NEIGHBORS DETAIL

Из вывод команды надо получить такие поля:

- LOCAL_HOST - имя устройства из приглашения
- DEST_HOST - имя соседа
- MGMNT_IP - IP-адрес соседа
- PLATFORM - модель соседнего устройства
- LOCAL_PORT - локальный интерфейс, который соединен с соседом
- REMOTE_PORT - порт соседнего устройства
- IOS_VERSION - версия IOS соседа

SHOW CDP NEIGHBORS DETAIL

Шаблон выглядит таким образом (файл templates/sh_cdp_n_det.template):

```
Value LOCAL_HOST (\S+)
Value DEST_HOST (\S+)
Value MGMNT_IP (.*)
Value PLATFORM (.*)
Value LOCAL_PORT (.*)
Value REMOTE_PORT (.*)
Value IOS_VERSION (\S+)

Start
  ^${LOCAL_HOST}[>#].
  ^Device ID: ${DEST_HOST}
  ^.*IP address: ${MGMNT_IP}
  ^Platform: ${PLATFORM},
  ^Interface: ${LOCAL_PORT},  Port ID \(outgoing port\): ${REMOTE_PORT}
  ^.*Version ${IOS_VERSION},
```

SHOW CDP NEIGHBORS DETAIL

Результат выполнения скрипта:

```
$ python parse_output.py templates/sh_cdp_n_det.template output/sh_cdp_n_det.txt
LOCAL_HOST    DEST_HOST    MGMNT_IP    PLATFORM    LOCAL_PORT             REMOTE_PORT          IOS_VERSION
------------  -----------  ----------  ----------  ---------------------  -------------------  -------------
SW1           R2           10.2.2.2    Cisco 2911  GigabitEthernet1/0/21  GigabitEthernet0/0  15.2(2)T1
```

SHOW CDP NEIGHBORS DETAIL

Несмотря на то, что правила с переменными описаны в разных строках, и, соответственно, работают с разными строками, TextFSM собирает их в одну строку таблицы. То есть, переменные, которые определены в начале шаблона, задают строку итоговой таблицы.

Обратите внимание, что в файле sh_cdp_n_det.txt находится вывод с тремя соседями, а в таблице только один сосед, последний.

RECORD

Так получилось из-за того, что в шаблоне не указано действие **Record**. И в итоге, в финальной таблице осталась только последняя строка.

Исправленый шаблон:

```
Value LOCAL_HOST (\S+)
Value DEST_HOST (\S+)
Value MGMNT_IP (.*)
Value PLATFORM (.*)
Value LOCAL_PORT (.*)
Value REMOTE_PORT (.*)
Value IOS_VERSION (\S+)

Start
  ^${LOCAL_HOST}[>#].
  ^Device ID: ${DEST_HOST}
  ^.*IP address: ${MGMNT_IP}
  ^Platform: ${PLATFORM},
  ^Interface: ${LOCAL_PORT},  Port ID \(outgoing port\): ${REMOTE_PORT}
  ^.*Version ${IOS_VERSION}, -> Record
```

RECORD

Теперь результат запуска скрипта выглядит так:

```
$ python parse_output.py templates/sh_cdp_n_det.template output/sh_cdp_n_det.txt
LOCAL_HOST    DEST_HOST    MGMNT_IP    PLATFORM              LOCAL_PORT             REMOTE_PORT          IOS_
------------  -----------  ----------  --------------------  ---------------------  -------------------  ----
SW1           SW2          10.1.1.2    cisco WS-C2960-8TC-L  GigabitEthernet1/0/16  GigabitEthernet0/1   12.2
              R1           10.1.1.1    Cisco 3825            GigabitEthernet1/0/22  GigabitEthernet0/0   12.4
              R2           10.2.2.2    Cisco 2911            GigabitEthernet1/0/21  GigabitEthernet0/0   15.2
```

Вывод получен со всех трёх устройств. Но, переменная LOCAL_HOST отображается не в каждой строке, а только в первой.

FILLDOWN

Это связано с тем, что приглашение, из которого взято значение переменной, появляется только один раз. И, для того, чтобы оно появлялось и в последующих строках, надо использовать действие **Filldown** для переменной LOCAL_HOST:

```
Value Filldown LOCAL_HOST (\S+)
Value DEST_HOST (\S+)
Value MGMNT_IP (.*)
Value PLATFORM (.*)
Value LOCAL_PORT (.*)
Value REMOTE_PORT (.*)
Value IOS_VERSION (\S+)

Start
  ^${LOCAL_HOST}[>#].
  ^Device ID: ${DEST_HOST}
  ^.*IP address: ${MGMNT_IP}
  ^Platform: ${PLATFORM},
  ^Interface: ${LOCAL_PORT},  Port ID \(outgoing port\): ${REMOTE_PORT}
  ^.*Version ${IOS_VERSION}, -> Record
```

FILLDOWN

Теперь мы получили такой вывод:

```
$ python parse_output.py templates/sh_cdp_n_det.template output/sh_cdp_n_det.txt
LOCAL_HOST    DEST_HOST    MGMNT_IP    PLATFORM              LOCAL_PORT             REMOTE_PORT          IOS_
------------  -----------  ----------  --------------------  ---------------------  -------------------  ----
SW1           SW2          10.1.1.2    cisco WS-C2960-8TC-L  GigabitEthernet1/0/16  GigabitEthernet0/1   12.2
SW1           R1           10.1.1.1    Cisco 3825            GigabitEthernet1/0/22  GigabitEthernet0/0   12.4
SW1           R2           10.2.2.2    Cisco 2911            GigabitEthernet1/0/21  GigabitEthernet0/0   15.2
SW1
```

Теперь значение переменной LOCAL_HOST появилось во всех трёх строках. Но появился ещё один странный эффект - последняя строка, в которой заполнена только колонка LOCAL_HOST.

REQUIRED

Дело в том, что все переменные, которые мы определили, опциональны. К тому же, одна переменная с параметром Filldown. И, чтобы избавиться от последней строки, нужно сделать хотя бы одну переменную обязательной, с помощью параметра **Required**:

```
Value Filldown LOCAL_HOST (\S+)
Value Required DEST_HOST (\S+)
Value MGMNT_IP (.*)
Value PLATFORM (.*)
Value LOCAL_PORT (.*)
Value REMOTE_PORT (.*)
Value IOS_VERSION (\S+)

Start
  ^${LOCAL_HOST}[>#].
  ^Device ID: ${DEST_HOST}
  ^.*IP address: ${MGMNT_IP}
  ^Platform: ${PLATFORM},
  ^Interface: ${LOCAL_PORT},  Port ID \(outgoing port\): ${REMOTE_PORT}
  ^.*Version ${IOS_VERSION}, -> Record
```

REQUIRED

Теперь мы получим корректный вывод:

```
$ python parse_output.py templates/sh_cdp_n_det.template output/sh_cdp_n_det.txt
LOCAL_HOST    DEST_HOST    MGMNT_IP    PLATFORM              LOCAL_PORT             REMOTE_PORT          IOS_
------------  -----------  ----------  --------------------  ---------------------  -------------------  ----
SW1           SW2          10.1.1.2    cisco WS-C2960-8TC-L  GigabitEthernet1/0/16  GigabitEthernet0/1   12.2
SW1           R1           10.1.1.1    Cisco 3825            GigabitEthernet1/0/22  GigabitEthernet0/0   12.4
SW1           R2           10.2.2.2    Cisco 2911            GigabitEthernet1/0/21  GigabitEthernet0/0   15.2
```

SHOW IP INTERFACE BRIEF

В случае, когда нужно обработать данные, которые выведены столбцами, шаблон TextFSM, наиболее удобен.

Шаблон для вывода команды show ip interface brief (файл templates/sh_ip_int_br.template):

```
Value INT (\S+)
Value ADDR (\S+)
Value STATUS (up|down|administratively down)
Value PROTO (up|down)

Start
  ^${INTF}\s+${ADDR}\s+\w+\s+\w+\s+${STATUS}\s+${PROTO} -> Record
```

SHOW IP INTERFACE BRIEF

В этом случае, правило можно описать одной строкой.

Вывод команды (файл output/sh_ip_int_br.txt):

```
R1#show ip interface brief
Interface                  IP-Address      OK? Method Status                Protocol
FastEthernet0/0            15.0.15.1       YES manual up                    up
FastEthernet0/1            10.0.12.1       YES manual up                    up
FastEthernet0/2            10.0.13.1       YES manual up                    up
FastEthernet0/3            unassigned      YES unset  up                    up
Loopback0                  10.1.1.1        YES manual up                    up
Loopback100                100.0.0.1       YES manual up                    up
```

SHOW IP INTERFACE BRIEF

Результат выполнения будет таким:

```
$ python parse_output.py templates/sh_ip_int_br.template output/sh_ip_int_br.txt
INT               ADDR         STATUS    PROTO
---------------   ----------   --------  -------
FastEthernet0/0   15.0.15.1    up        up
FastEthernet0/1   10.0.12.1    up        up
FastEthernet0/2   10.0.13.1    up        up
FastEthernet0/3   unassigned   up        up
Loopback0         10.1.1.1     up        up
Loopback100       100.0.0.1    up        up
```

SHOW IP ROUTE OSPF

Рассмотрим случай, когда нам нужно обработать вывод команды show ip route ospf и в таблице маршрутизации есть несколько маршрутов к одной сети.

Для маршрутов к одной и той же сети, вместо нескольких строк, где будет повторяться сеть, будет создана одна запись, в которой все доступные next-hop адреса собраны в список.

SHOW IP ROUTE OSPF

Пример вывода команды show ip route ospf (файл output/sh_ip_route_ospf.txt):

```
R1#sh ip route ospf
Codes: L - local, C - connected, S - static, R - RIP, M - mobile, B - BGP
       D - EIGRP, EX - EIGRP external, O - OSPF, IA - OSPF inter area
       N1 - OSPF NSSA external type 1, N2 - OSPF NSSA external type 2
       E1 - OSPF external type 1, E2 - OSPF external type 2
       i - IS-IS, su - IS-IS summary, L1 - IS-IS level-1, L2 - IS-IS level-2
       ia - IS-IS inter area, * - candidate default, U - per-user static route
       o - ODR, P - periodic downloaded static route, H - NHRP, l - LISP
       + - replicated route, % - next hop override

Gateway of last resort is not set

      10.0.0.0/8 is variably subnetted, 10 subnets, 2 masks
O        10.0.24.0/24 [110/20] via 10.0.12.2, 1w2d, Ethernet0/1
O        10.0.34.0/24 [110/20] via 10.0.13.3, 1w2d, Ethernet0/2
O        10.2.2.2/32 [110/11] via 10.0.12.2, 1w2d, Ethernet0/1
O        10.3.3.3/32 [110/11] via 10.0.13.3, 1w2d, Ethernet0/2
O        10.4.4.4/32 [110/21] via 10.0.13.3, 1w2d, Ethernet0/2
                     [110/21] via 10.0.12.2, 1w2d, Ethernet0/1
                     [110/21] via 10.0.14.4, 1w2d, Ethernet0/3
O        10.5.35.0/24 [110/20] via 10.0.13.3, 1w2d, Ethernet0/2
```

SHOW IP ROUTE OSPF

Для этого примера упрощаем задачу и считаем, что маршруты могут быть только OSPF и с обозначением, только O (то есть, только внутризональные маршруты).

Первая версия шаблона выглядит так:

```
Value Network (([0-9]{1,3}.[0-9]{1,3}.[0-9]{1,3}.[0-9]{1,3}))
Value Mask (\/\d{1,2})
Value Distance (\d+)
Value Metric (\d+)
Value NextHop ([0-9]{1,3}.[0-9]{1,3}.[0-9]{1,3}.[0-9]{1,3})

Start
  ^O +${Network}${Mask}\s\[${Distance}\/${Metric}\]\svia\s${NextHop}, -> Record
```

SHOW IP ROUTE OSPF

Результат получился такой:

```
Network     Mask      Distance    Metric  NextHop
---------   ------  ----------  --------  ---------
10.0.24.0   /24            110        20  10.0.12.2
10.0.34.0   /24            110        20  10.0.13.3
10.2.2.2    /32            110        11  10.0.12.2
10.3.3.3    /32            110        11  10.0.13.3
10.4.4.4    /32            110        21  10.0.13.3
10.5.35.0   /24            110        20  10.0.13.3
```

LIST

Всё нормально, но потерялись варианты путей для маршрута 10.4.4.4/32. Это логично, ведь нет правила, которое подошло бы для такой строки.

Воспользуемся опцией **List** для переменной NextHop:

```
Value Network (([0-9]{1,3}.[0-9]{1,3}.[0-9]{1,3}.[0-9]{1,3}))
Value Mask (\/\d{1,2})
Value Distance (\d+)
Value Metric (\d+)
Value List NextHop ([0-9]{1,3}.[0-9]{1,3}.[0-9]{1,3}.[0-9]{1,3})

Start
  ^O +${Network}${Mask}\s\[${Distance}\/${Metric}\]\svia\s${NextHop}, -> Record
```

LIST

Теперь вывод получился таким:

```
Network     Mask      Distance    Metric  NextHop
---------  ------  ----------  --------  -------------
10.0.24.0  /24            110        20  ['10.0.12.2']
10.0.34.0  /24            110        20  ['10.0.13.3']
10.2.2.2   /32            110        11  ['10.0.12.2']
10.3.3.3   /32            110        11  ['10.0.13.3']
10.4.4.4   /32            110        21  ['10.0.13.3']
10.5.35.0  /24            110        20  ['10.0.13.3']
```

LIST

Изменилось то, что в столбце NextHop отображается список, но пока с одним элементом.

Так как, перед записью маршрута, для которого есть несколько путей, надо добавить к нему все доступные адреса NextHop, надо перенести действие **Record**.

LIST

Для этого, запись переносится на момент, когда встречается следующая строка с маршрутом. В этот момент надо записать предыдущую строку и только после этого, уже записывать текущую. Для этого, используется такая запись:

```
^0 -> Continue.Record
```

LIST

В ней действие **Record** говорит, что надо записать текущее значение переменных. А, так как в этом правиле нет переменных, записывается то, что было в предыдущих значениях.

Действие **Continue** говорит, что надо продолжить работать с текущей строкой так, как-будто совпадения не было. Засчет этого, сработает следующая строка.

Остается добавить правило, которое будет описывать дополнительные маршруты к сети (в них нет сети и маски):

```
^\s+\[${Distance}\/${Metric}\]\svia\s${NextHop},
```

LIST

Итоговый шаблон выглядит так (файл templates/sh_ip_route_ospf.template):

```
Value Network (([0-9]{1,3}.[0-9]{1,3}.[0-9]{1,3}.[0-9]{1,3}))
Value Mask (\/\d{1,2})
Value Distance (\d+)
Value Metric (\d+)
Value List NextHop ([0-9]{1,3}.[0-9]{1,3}.[0-9]{1,3}.[0-9]{1,3})

Start
  ^O -> Continue.Record
  ^O +${Network}${Mask}\s\[${Distance}\/${Metric}\]\svia\s${NextHop},
  ^\s+\[${Distance}\/${Metric}\]\svia\s${NextHop},
```

LIST

В результате, мы получим такой вывод:

```
Network     Mask      Distance    Metric  NextHop
---------   ------  ----------  --------  ---------------------------------------
10.0.24.0   /24            110        20  ['10.0.12.2']
10.0.34.0   /24            110        20  ['10.0.13.3']
10.2.2.2    /32            110        11  ['10.0.12.2']
10.3.3.3    /32            110        11  ['10.0.13.3']
10.4.4.4    /32            110        21  ['10.0.13.3', '10.0.12.2', '10.0.14.4']
10.5.35.0   /24            110        20  ['10.0.13.3']
```

SHOW ETHERCHANNEL SUMMARY

SHOW ETHERCHANNEL SUMMARY

TextFSM удобно использовать для разбора вывода, который отображается столбцами или для обработки вывода, который находится в разных строках. Менее удобными получаются шаблоны, когда надо получить несколько однотипных элементов из одной строки.

SHOW ETHERCHANNEL SUMMARY

Пример вывода команды show etherchannel summary (файл output/sh_etherchannel_summary.txt):

```
sw1# sh etherchannel summary
Flags:  D - down        P - bundled in port-channel
        I - stand-alone s - suspended
        H - Hot-standby (LACP only)
        R - Layer3      S - Layer2
        U - in use      f - failed to allocate aggregator

        M - not in use, minimum links not met
        u - unsuitable for bundling
        w - waiting to be aggregated
        d - default port


Number of channel-groups in use: 2
Number of aggregators:           2

Group  Port-channel  Protocol    Ports
------+-------------+-----------+-----------------------------------------------
1      Po1(SU)         LACP      Fa0/1(P)   Fa0/2(P)   Fa0/3(P)
3      Po3(SU)          -        Fa0/11(P)   Fa0/12(P)   Fa0/13(P)   Fa0/14(P)
```

SHOW ETHERCHANNEL SUMMARY

В данном случае, нужно получить:

- имя и номер port-channel. Например, Po1
- список всех портов в нем. Например, ['Fa0/1', 'Fa0/2', 'Fa0/3']

Сложность тут в том, что порты находятся в одной строке, а в TextFSM нельзя указывать одну и ту же переменную несколько раз в строке. Но, есть возможность несколько раз искать совпадение в строке.

SHOW ETHERCHANNEL SUMMARY

Первая версия шаблона выглядит так:

```
Value CHANNEL (\S+)
Value List MEMBERS (\w+\d+\/\d+)

Start
  ^\d+ +${CHANNEL}\(\S+ +[\w-]+ +[\w ]+ +${MEMBERS}\( -> Record
```

SHOW ETHERCHANNEL SUMMARY

В шаблоне две переменные:

- CHANNEL - имя и номер агрегированного порта
- MEMBERS - список портов, которые входят в агрегированный порт. Для этой переменной указан тип - List

Результат:

```
CHANNEL    MEMBERS
---------  ----------
Po1        ['Fa0/1']
Po3        ['Fa0/11']
```

SHOW ETHERCHANNEL SUMMARY

Пока что в выводе только первый порт, а нужно чтобы попали все порты. В данном случае, надо продолжить обработку строки с портами, после найденного совпадения. То есть, использовать действие Continue и описать следующее выражение.

Единственная строка, которая есть в шаблоне, описывает первый порт. Надо добавить строку, которая описывает следующий порт.

SHOW ETHERCHANNEL SUMMARY

Следующая версия шаблона:

```
Value CHANNEL (\S+)
Value List MEMBERS (\w+\d+\/\d+)

Start
  ^\d+ +${CHANNEL}\(\S+ +[\w-]+ +[\w ]+ +${MEMBERS}\( -> Continue
  ^\d+ +${CHANNEL}\(\S+ +[\w-]+ +[\w ]+ +\S+ +${MEMBERS}\( -> Record
```

Вторая строка описывает такое же выражение, но переменная MEMBERS смещается на следующий порт.

SHOW ETHERCHANNEL SUMMARY

Результат:

```
CHANNEL    MEMBERS
---------  -------------------
Po1        ['Fa0/1', 'Fa0/2']
Po3        ['Fa0/11', 'Fa0/12']
```

SHOW ETHERCHANNEL SUMMARY

Аналогично надо дописать в шаблон строки, которые описывают третий и четвертый порт. Но, так как в выводе может быть переменное количество портов, надо перенести правило Record на отдельную строку, чтобы оно не было привязано к конкретному количеству портов в строке.

Если Record будет находиться, например, после строки, в которой описаны четыре порта, для ситуации когда портов в строке меньше, запись не будет выполняться.

SHOW ETHERCHANNEL SUMMARY

Итоговый шаблон (файл templates/sh_etherchannel_summary.txt):

```
Value CHANNEL (\S+)
Value List MEMBERS (\w+\d+\/\d+)

Start
  ^\d+.* -> Continue.Record
  ^\d+ +${CHANNEL}\(\S+ +[\w-]+ +[\w ]+ +\S+ +${MEMBERS}\( -> Continue
  ^\d+ +${CHANNEL}\(\S+ +[\w-]+ +[\w ]+ +(\S+ +){2} +${MEMBERS}\( -> Continue
  ^\d+ +${CHANNEL}\(\S+ +[\w-]+ +[\w ]+ +(\S+ +){3} +${MEMBERS}\( -> Continue
```

SHOW ETHERCHANNEL SUMMARY

Результат обработки:

```
CHANNEL    MEMBERS
---------  ---------------------------------------
Po1        ['Fa0/1', 'Fa0/2', 'Fa0/3']
Po3        ['Fa0/11', 'Fa0/12', 'Fa0/13', 'Fa0/14']
```

SHOW ETHERCHANNEL SUMMARY

Возможен ещё один вариант вывода команды sh etherchannel summary (файл output/sh_etherchannel_summary2.txt):

```
sw1# sh etherchannel summary
Flags:  D - down        P - bundled in port-channel
        I - stand-alone s - suspended
        H - Hot-standby (LACP only)
        R - Layer3      S - Layer2
        U - in use      f - failed to allocate aggregator

        M - not in use, minimum links not met
        u - unsuitable for bundling
        w - waiting to be aggregated
        d - default port


Number of channel-groups in use: 2
Number of aggregators:           2

Group  Port-channel  Protocol    Ports
------+-------------+-----------+-----------------------------------------------
1      Po1(SU)         LACP      Fa0/1(P)   Fa0/2(P)   Fa0/3(P)
3      Po3(SU)          -        Fa0/11(P)   Fa0/12(P)   Fa0/13(P)   Fa0/14(P)
                                 Fa0/15(P)   Fa0/16(P)
```

SHOW ETHERCHANNEL SUMMARY

Для того чтобы шаблон обрабатывал и этот вариант, надо его модифицировать (файл templates/sh_etherchannel_summary2.txt):

```
Value CHANNEL (\S+)
Value List MEMBERS (\w+\d+\/\d+)

Start
  ^\d+.* -> Continue.Record
  ^\d+ +${CHANNEL}\(\S+ +[\w-]+ +[\w ]+ +${MEMBERS}\( -> Continue
  ^\d+ +${CHANNEL}\(\S+ +[\w-]+ +[\w ]+ +\S+ +${MEMBERS}\( -> Continue
  ^\d+ +${CHANNEL}\(\S+ +[\w-]+ +[\w ]+ +(\S+ +){2} +${MEMBERS}\( -> Continue
  ^\d+ +${CHANNEL}\(\S+ +[\w-]+ +[\w ]+ +(\S+ +){3} +${MEMBERS}\( -> Continue
  ^ +${MEMBERS} -> Continue
  ^ +\S+ +${MEMBERS} -> Continue
  ^ +(\S+ +){2} +${MEMBERS} -> Continue
  ^ +(\S+ +){3} +${MEMBERS} -> Continue
```

Результат будет таким:

```
CHANNEL    MEMBERS
---------  ------------------------------------------------------------
Po1        ['Fa0/1', 'Fa0/2', 'Fa0/3']
Po3        ['Fa0/11', 'Fa0/12', 'Fa0/13', 'Fa0/14', 'Fa0/15', 'Fa0/16']
```

TEXTFSM CLI TABLE

TEXTFSM CLI TABLE

Благодаря TextFSM, можно обрабатывать вывод команд и получать структурированный результат. Но, всё ещё надо вручную прописывать каким шаблоном обрабатывать команды show, каждый раз, когда используется TextFSM.

Было бы намного удобней иметь какое-то соответствие между командой и шаблоном. Чтобы можно было написать общий скрипт, который выполняет подключения к устройствам, отправляет команды, сам выбирает шаблон и парсит вывод в соответствиее с шаблоном.

TEXTFSM CLI TABLE

В TextFSM есть такая возможность.

Для того, чтобы ей можно было воспользоваться, надо создать файл в котором описаны соответствия между командами и шаблонами. В TextFSM он называется index.

TEXTFSM CLI TABLE

Файл index должен находится в каталоге с шаблонами и должен иметь такой формат:

- первая строка - названия колонок
- каждая следующая строка - это соответствие шаблона команде

TEXTFSM CLI TABLE

- обязательные колонки, местоположение которых фиксировано (должны быть обязательно первой и последней, соответственно):
  - первая колонка - имена шаблонов
  - последняя колонка - соответствующая команда
    - в этой колонке используется специальный формат, чтобы описать то, что команда может быть написана не полностью

TEXTFSM CLI TABLE

- остальные колонки могут быть любыми
  - например, в примере ниже будут колонки Hostname, Vendor. Они позволяют уточнить информацию об устройстве, чтобы определить какой шаблон использовать.
    - например, команда show version может быть у оборудования Cisco и HP. Соответственно, только команды недостаточно, чтобы определить какой шаблон использовать. В таком случае, можно передать информацию о том, какой тип оборудования используется, вместе с командой, и тогда получится определить правильный шаблон.
- во всех столбцах, кроме первого, поддерживаются регулярные выражения
  - в командах, внутри [[]] регулярные выражения не

TEXTFSM CLI TABLE

Пример файла index:

```
Template, Hostname, Vendor, Command
sh_cdp_n_det.template, .*, Cisco, sh[[ow]] cdp ne[[ighbors]] de[[tail]]
sh_clock.template, .*, Cisco, sh[[ow]] clo[[ck]]
sh_ip_int_br.template, .*, Cisco, sh[[ow]] ip int[[erface]] br[[ief]]
sh_ip_route_ospf.template, .*, Cisco, sh[[ow]] ip rou[[te]] o[[spf]]
```

TEXTFSM CLI TABLE

Обратите внимание на то, как записаны команды:

- sh[[ow]] ip int[[erface]] br[[ief]]
  - эта запись будет преобразована в выражение sh((ow)?)? ip int((erface)?)? br((ief)?)?
  - это значит, что TextFSM сможет определить какой шаблон использовать, даже если команда набрана не полностью
  - например, такие варианты команды сработают:
    - sh ip int br
    - show ip inter bri

КАК ИСПОЛЬЗОВАТЬ CLI TABLE

Посмотрим как пользоваться классом clitable и файлом index.

В каталоге templates такие шаблоны и файл index:

```
sh_cdp_n_det.template
sh_clock.template
sh_ip_int_br.template
sh_ip_route_ospf.template
index
```

КАК ИСПОЛЬЗОВАТЬ CLI TABLE

Сначала попробуем поработать с CLI Table в ipython, чтобы посмотреть какие возможности есть у этого класса, а затем посмотрим на финальный скрипт.

Для начала, импортируем класс clitable:

```
In [1]: import clitable
```

КАК ИСПОЛЬЗОВАТЬ CLI TABLE

Проверять работу clitable будем на последнем примере из прошлого раздела - выводе команды show ip route ospf. Считываем вывод, который хранится в файле output/sh_ip_route_ospf.txt, в строку:

```
In [2]: output_sh_ip_route_ospf = open('output/sh_ip_route_ospf.txt').read()
```

КАК ИСПОЛЬЗОВАТЬ CLI TABLE

Сначала надо инициализировать класс, передав ему имя файла, в котором хранится соответствие между шаблонами и командами, и указать имя каталога, в котором хранятся шаблоны:

```
In [3]: cli_table = clitable.CliTable('index', 'templates')
```

КАК ИСПОЛЬЗОВАТЬ CLI TABLE

Надо указать какая команда передается и указать дополнительные атрибуты, которые помогут идентифицировать шаблон. Для этого, нужно создать словарь, в котором ключи - имена столбцов, которые определены в файле index. В данном случае, не обязательно указывать название вендора, так как команде sh ip route ospf соответствует только один шаблон.

```
In [4]: attributes = {'Command': 'show ip route ospf' , 'Vendor': 'Cisco'}
```

КАК ИСПОЛЬЗОВАТЬ CLI TABLE

Методу ParseCmd надо передать вывод команды и словарь с параметрами:

```
In [5]: cli_table.ParseCmd(output_sh_ip_route_ospf, attributes)
```

В результате, в объекте cli_table, получаем обработанный вывод команды sh ip route ospf.

КАК ИСПОЛЬЗОВАТЬ CLI TABLE

Методы cli_table (чтобы посмотреть все методы, надо вызвать dir(cli_table)):

```
In [6]: cli_table.
cli_table.AddColumn        cli_table.NewRow          cli_table.index           cli_table.size
cli_table.AddKeys          cli_table.ParseCmd        cli_table.index_file      cli_table.sort
cli_table.Append           cli_table.ReadIndex       cli_table.next            cli_table.superkey
cli_table.CsvToTable       cli_table.Remove          cli_table.raw             cli_table.synchronised
cli_table.FormattedTable   cli_table.Reset           cli_table.row             cli_table.table
cli_table.INDEX            cli_table.RowWith         cli_table.row_class       cli_table.template_dir
cli_table.KeyValue         cli_table.extend          cli_table.row_index
cli_table.LabelValueTable  cli_table.header          cli_table.separator
```

КАК ИСПОЛЬЗОВАТЬ CLI TABLE

Например, если вызвать print cli_table, получим такой вывод:

```
In [7]: print(cli_table)
Network, Mask, Distance, Metric, NextHop
10.0.24.0, /24, 110, 20, ['10.0.12.2']
10.0.34.0, /24, 110, 20, ['10.0.13.3']
10.2.2.2, /32, 110, 11, ['10.0.12.2']
10.3.3.3, /32, 110, 11, ['10.0.13.3']
10.4.4.4, /32, 110, 21, ['10.0.13.3', '10.0.12.2', '10.0.14.4']
10.5.35.0, /24, 110, 20, ['10.0.13.3']
```

КАК ИСПОЛЬЗОВАТЬ CLI TABLE

Метод FormattedTable позволяет получить вывод в виде таблицы:

```
In [8]: print(cli_table.FormattedTable())
 Network    Mask  Distance  Metric  NextHop
=================================================================
 10.0.24.0  /24   110       20      10.0.12.2
 10.0.34.0  /24   110       20      10.0.13.3
 10.2.2.2   /32   110       11      10.0.12.2
 10.3.3.3   /32   110       11      10.0.13.3
 10.4.4.4   /32   110       21      10.0.13.3, 10.0.12.2, 10.0.14.4
 10.5.35.0  /24   110       20      10.0.13.3
```

Такой вывод это просто строка, который может пригодится для отображения информации.

КАК ИСПОЛЬЗОВАТЬ CLI TABLE

Чтобы получить из объекта cli_table структурированный вывод, например, список списков, надо обратиться к объекту таким образом:

```
In [9]: data_rows = [list(row) for row in cli_table]

In [11]: data_rows
Out[11]:
[['10.0.24.0', '/24', '110', '20', ['10.0.12.2']],
 ['10.0.34.0', '/24', '110', '20', ['10.0.13.3']],
 ['10.2.2.2', '/32', '110', '11', ['10.0.12.2']],
 ['10.3.3.3', '/32', '110', '11', ['10.0.13.3']],
 ['10.4.4.4', '/32', '110', '21', ['10.0.13.3', '10.0.12.2', '10.0.14.4']],
 ['10.5.35.0', '/24', '110', '20', ['10.0.13.3']]]
```

КАК ИСПОЛЬЗОВАТЬ CLI TABLE

Отдельно можно получить названия столбцов:

```
In [12]: header = list(cli_table.header)

In [14]: header
Out[14]: ['Network', 'Mask', 'Distance', 'Metric', 'NextHop']
```

Теперь вывод аналогичен тому, который был получен в прошлом разделе.

КАК ИСПОЛЬЗОВАТЬ CLI TABLE

Соберем всё в один скрипт (файл textfsm_clitable.py):

```
import clitable

output_sh_ip_route_ospf = open('output/sh_ip_route_ospf.txt').read()

cli_table = clitable.CliTable('index', 'templates')

attributes = {'Command': 'show ip route ospf' , 'Vendor': 'Cisco'}

cli_table.ParseCmd(output_sh_ip_route_ospf, attributes)
print("CLI Table output:\n", cli_table)

print("Formatted Table:\n", cli_table.FormattedTable())

data_rows = [list(row) for row in cli_table]
header = list(cli_table.header)

print(header)
for row in data_rows:
    print(row)
```

КАК ИСПОЛЬЗОВАТЬ CLI TABLE

Вывод будет таким:

```
$ python textfsm_clitable.py
CLI Table output:
Network, Mask, Distance, Metric, NextHop
10.0.24.0, /24, 110, 20, ['10.0.12.2']
10.0.34.0, /24, 110, 20, ['10.0.13.3']
10.2.2.2, /32, 110, 11, ['10.0.12.2']
10.3.3.3, /32, 110, 11, ['10.0.13.3']
10.4.4.4, /32, 110, 21, ['10.0.13.3', '10.0.12.2', '10.0.14.4']
10.5.35.0, /24, 110, 20, ['10.0.13.3']

Formatted Table:
 Network    Mask  Distance  Metric  NextHop
==========================================================================
 10.0.24.0  /24   110       20      10.0.12.2
 10.0.34.0  /24   110       20      10.0.13.3
 10.2.2.2   /32   110       11      10.0.12.2
 10.3.3.3   /32   110       11      10.0.13.3
 10.4.4.4   /32   110       21      10.0.13.3, 10.0.12.2, 10.0.14.4
 10.5.35.0  /24   110       20      10.0.13.3

['Network', 'Mask', 'Distance', 'Metric', 'NextHop']
['10.0.24.0', '/24', '110', '20', ['10.0.12.2']]
['10.0.34.0', '/24', '110', '20', ['10.0.13.3']]
['10.2.2.2', '/32', '110', '11', ['10.0.12.2']]
['10.3.3.3', '/32', '110', '11', ['10.0.13.3']]
['10.4.4.4', '/32', '110', '21', ['10.0.13.3', '10.0.12.2', '10.0.14.4']]
['10.5.35.0', '/24', '110', '20', ['10.0.13.3']]
```