SWITCHING & ROUTING

2018/2019

# PACKET CLASSIFICATION TRIE-BASED ALGORITHM

KASENOV ERBOL 916027

D'ADDA MATTIA 917652

CALZI NICOLO' 903430

**Contents**

**Page**

## 1 INTRODUCTION

The goal of our project is to implement a Packet Classification's method over a trasmission, performing a linear search algorithm or a hierarchical search one (implemented in the three possibile ways of: basic version, set pruning or grid of tries).

The classification is done on a SDN controller in order to be able to measure and compare the performances of the different algorithms used.

The key point of the project is the implementation of the hierarchical search algoritmhs since the linear search is much more easier and already coded; instead, the hierarchical search consists in the introduction of nodes and indexes in a complex way such that the time required in finding the best rule for the incoming packet is reduced.
Later on the differents between these implementations will be better explained.

## 2 REQUIREMENTS

- Theory of Packet classification and its algorithms;
- How the linear search works;
- How the trie based algoritmhs work in the implementation of hierarchical searching (and advance versions);
- Basic knowledge of the language Python;
- Ryu and Mininet installed on the PCs.

# 3 SDN, MININET, RYU AND THE OPENFLOW PROTOCOL

SDN (Software-defined networking) is a dynamic, manageable, cost effective and adaptable architecture which enables functions to control the network in a direct programmable way in order to abstract applications and network services. One of its main characteristic is to separate the control plane (which takes decisions on where to send the incoming packet) from the data plane (which forwards the packet to its corresponding destination), by placing them on different systems. In this way, the network intelligence is centralized in SDN controllers, while the networks devices become simple data forwarding devices.

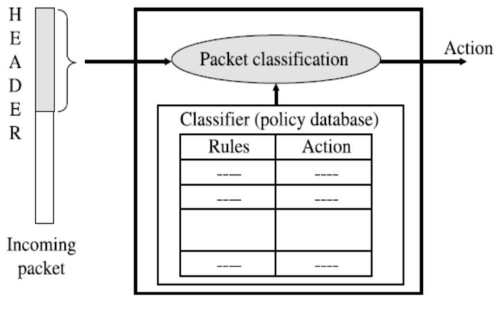
Network devices are programmed by the SDN controllers through the OpenFlow protocol.
in our scenario we have used Mininet, which emulates the use of an OpenFlow protocol and by default supports OpenFlow v 1.0, and Ryu, a framework that supports OpenFlow too and that is used to create the controller.

But why SDN and OpenFlow?
They are the basic "tools" on which rely the principles of flow-aware routing and packet classification.

## 4 PACKET CLASSIFICATION

When a new packet arrives at a router it needs to be forwarded to the right output to reach its destination, possibly in the shortest time the system can guarantee and with an high quality of the trasmission; when the router experiences not just a single arrive but multiple arrivals at the same time, it has to implement some mechanisms that leed to have a classification of the incoming packets based on different requirements to be able to know in advance, with respect to the trasmission time, where and when sends the incoming packet. The division is based on the so called “quality of service” (QoS) that the physical user or company had decided at the subscription of the contract with the service provider.
To meet various QoS requirements, routers need to implemement admission control, resource reservation, per-flow queuing and fair scheduling; all these features are needed to distinguish and classify the incoming traffic into different flows.

The mechanism is simple: when the flow of data arrives at the router, it contains some values in its fields (parts in which is divided the packet's header) that are compared with a set of rules contained in the router itself; a set of these rules is called classifier and it is based on the criteria that has to be applied to classify the packets, finding the action that it has to perform with respect to a given network application. In particular, a rule is usually based on fields that match with values of the packet, such as the source address where they are coming from or the destination address where they are going to, source or destination ports, the protocolls they are using in the trasmission.



Matching the packet header to the rules in the classifier

The aim of the packet classification is to find the rule in the classifier that best matches with the incoming packet, and since there could be differents rules that match with the same packet, find the one with the highest priority.
There are several algorithms for packet classification:

. Linear searchTrie-based classification
. Geometric algorithms
. Heuristic algorithms
. TCAM-based algorithms

In our project, starting from the simple Linear Search we have implemented some Trie-based's algorithms such as Hierarchical trie, Set-pruning trie and Grid of tries.

## 4.1 Trie-based algorithms – Brief theoretical concepts

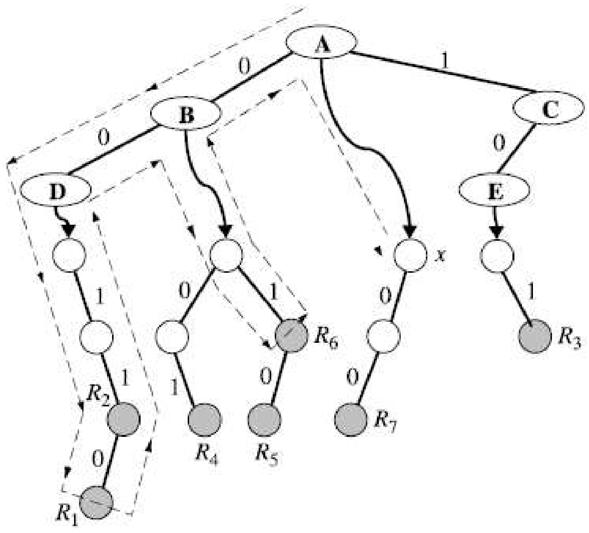
The aim of the following paragraphs is just to give an idea of the concepts from where we started. The functions used in our project are better explained later on.

### 4.1.1 Hierarchical algorithm

A hierarchical algorithm can be seen as an extension of the one-dimension trie to multiple dimension trie, where each dimension represents a field of the packet.
So first thing to do is to create the tree: in the classifier, the set of prefixes related to packets are divided in fields. A first F1 binary radix trie is built for the set of prefixes that belong to F1 of all the rules; then, for each prefix in the F1-trie, a hierarchical trie is recursively constructed for those rules that exacly specify the choosen prefix in the F1 trie with a dimendion equal to the previous dimension -1. This new trie is connected to the prefix with a next-trie pointer stored in the node.
And so on the process is repeated till the end of the prefixes.

When a packet arrives, its header is scanned: the algorithm passes through the F1 trie based on the first value of the header; if a next-trie pointer is encountered, the algorithm goes on with the pointer and queries hierarchical trie recursively till find the point where no other pointer can be considered.

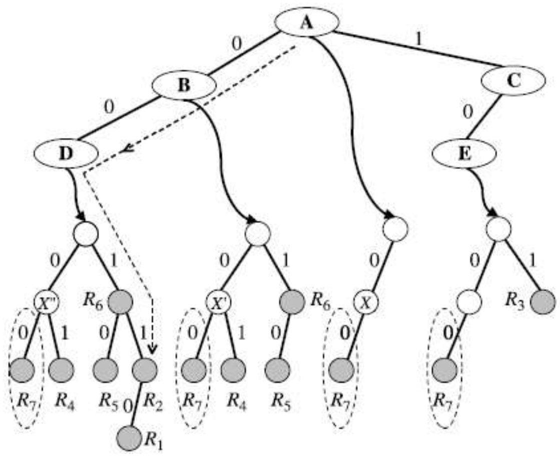
At this point, the algoritmh need to backtrack to previous node (to the lowest ancestor) because it could be that the rule it has found it is not the only one matching the incoming packet, and maybe another one could have an highest priority wrt the one just found.
If, after the backtracking process, more than one rule is found, the one with the highest priority in the classifier will be choosen as the rule that best matches with the considered packet.
Here an example: the first search leeds us to rule R2 and R1, then the backtracking process finds that also R6 could be good match; looking at the priority in the scheduler, R1 will be the choosen rule.



From the basic version of hierarchical algorithm we can implemement other algorithms in order to reduce the searhing time and improve in scalability and performances.

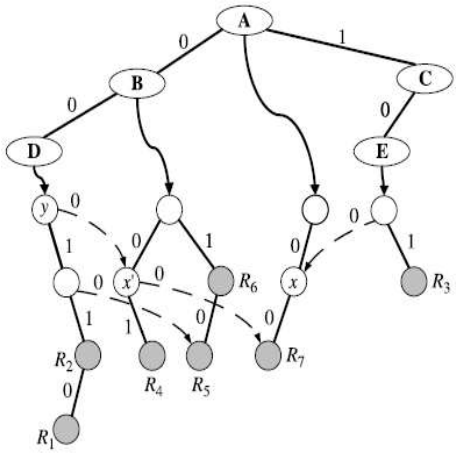
### 4.1.2 Set-pruning trie

The set-pruning trie is a modified version of the hierarchical trie. Basically it avoids the process of backtracking by putting at each leaf node all the previuos rules that the backtracking process will leed to. So all the possible rules for an incoming packet are found a the very last point of the search in the tree.
This implementation is done by storing all possible actual and previous rules in the same node; so each trie node duplicates all rules from its ancestor nodes into its own set of rules and then constructs the next dimension trie based on the new set of rules.



### 4.1.3 Grid-of-tries

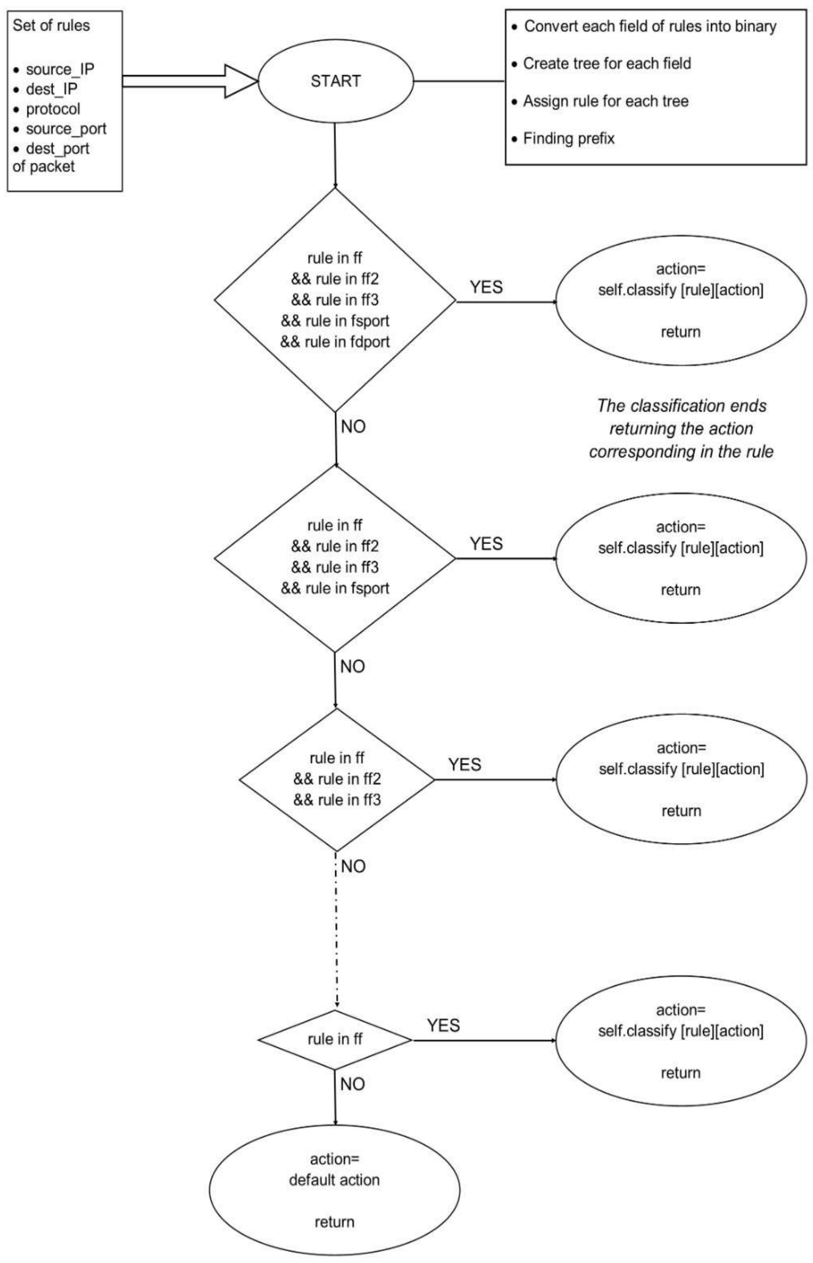
The grid of tries is another modified version of the hierarchical trie. In this case insted of duplicate the rules from ancestor nodes, what it is done is adding pointer at each node that point out to the ancestor nodes where you can find other possible matching rules, again to avoid the backtracking process.



## 5 flow of the program

In this section is described the flow of the code after the choice of the topology and the algorithm used for the classification (explained later on) and the functions used.

### 5.1 Flow chart


Set of rules
• source_IP
• dest_IP
• protocol
• source_port
• dest_port
of packet
START
• Convert each field of rules into binary
• Create tree for each field
• Assign rule for each tree
• Finding prefix
rule in ff && rule in ff2 && rule in ff3 && rule in fsport && rule in fdport
YES
action= self.classify [rule][action] return
NO
The classification ends returning the action corresponding in the rule
rule in ff && rule in ff2 && rule in ff3 && rule in fsport
YES
action= self.classify [rule][action] return
NO
rule in ff && rule in ff2 && rule in ff3
YES
action= self.classify [rule][action] return
NO
rule in ff
YES
action= self.classify [rule][action] return
NO
action= default action return

## 5.2 Functions

Here are represented all the functions we have used in our project, with a brief explanation of what they do and the code we have used to implemement them; the code is comment and self-explanating of what the function is performing at each line of code.

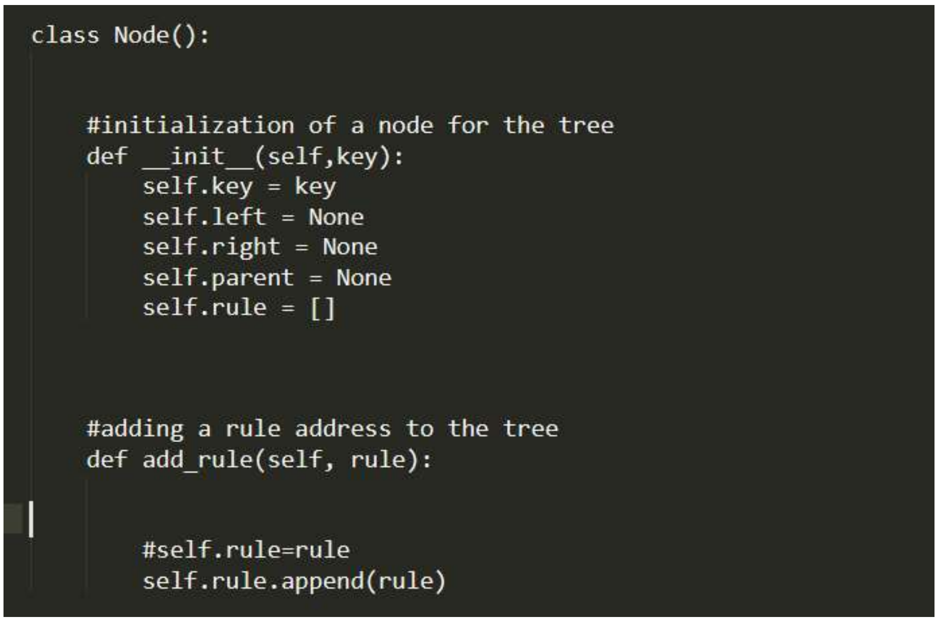
### 5.2.1 Generation tree

This function is common to all the algorithms; so the generation does not change, what change is the assignment of the rules for each algorithm.

**Class Node**
Each element of the tree is an object of class *Node*.
It contains five parameters:

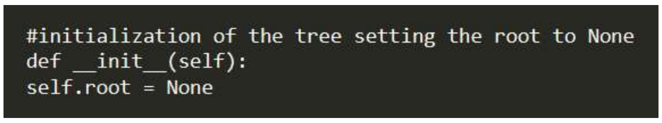
1. The key that identifies the node
2. The pointer to the left "child" node
3. The pointer to the right "child" node
4. The pointer to the "father" node
5. The set of rule which are satisfied by the node itself, taken from the list of rules

The only function of this class is *add_rule*, that allows to add a new rule to the set.



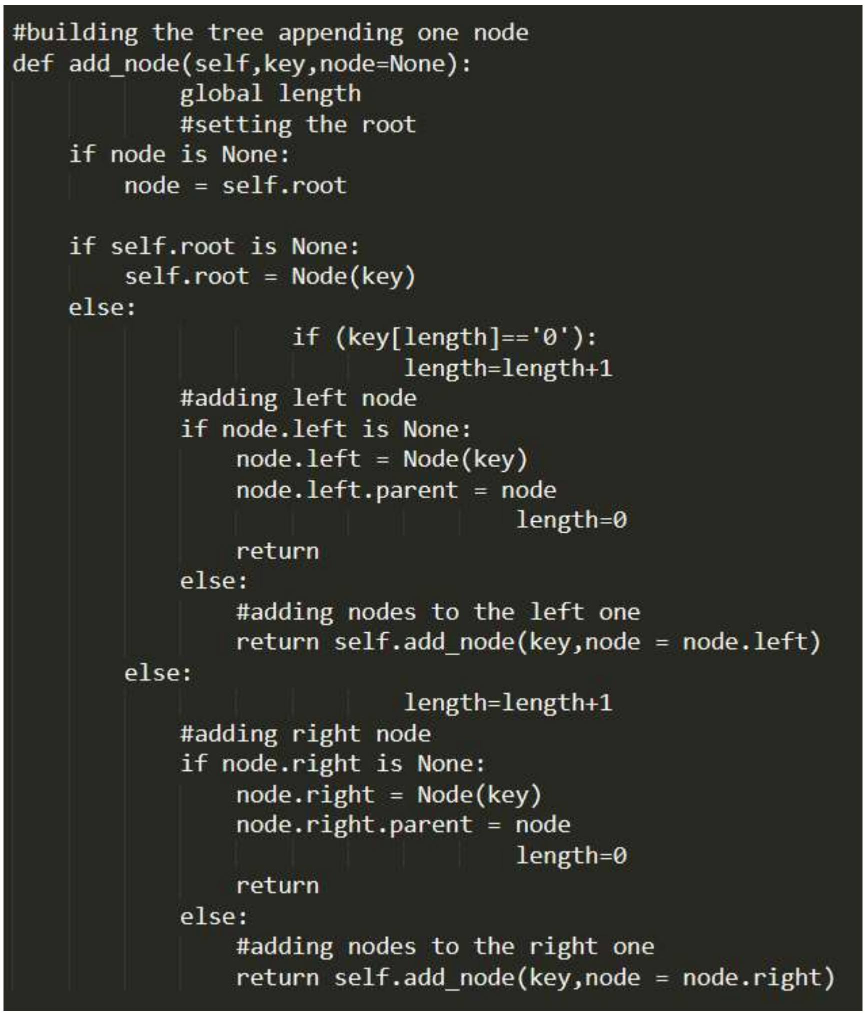
```
class Node():


    #initialization of a node for the tree
    def __init__(self,key):
        self.key = key
        self.left = None
        self.right = None
        self.parent = None
        self.rule = []



    #adding a rule address to the tree
    def add_rule(self, rule):


        #self.rule=rule
        self.rule.append(rule)
```

**Class Tree**
The structure on which the classification algorithms work is based on a tree implemented by the class Tree.
At the beginning the tree is empty, so its root is initialized to None.



```
#initialization of the tree setting the root to None
def __init__(self):
self.root = None
```

Then, the tree is built adding every nodes and assigning them to own father, as "left child" or "right child".



```
#building the tree appending one node
def add_node(self,key,node=None):
            global length
            #setting the root
    if node is None:
        node = self.root

    if self.root is None:
        self.root = Node(key)
    else:
                    if (key[length]=='0'):
                            length=length+1
            #adding left node
            if node.left is None:
                node.left = Node(key)
                node.left.parent = node
                                    length=0
                return
            else:
                #adding nodes to the left one
                return self.add_node(key,node = node.left)
        else:
                            length=length+1
            #adding right node
            if node.right is None:
                node.right = Node(key)
                node.right.parent = node
                                    length=0
                return
            else:
                #adding nodes to the right one
                return self.add_node(key,node = node.right)
```

Once all the nodes have been added, the tree is complete.
Then, it can be print it calling the method *print_tree*.

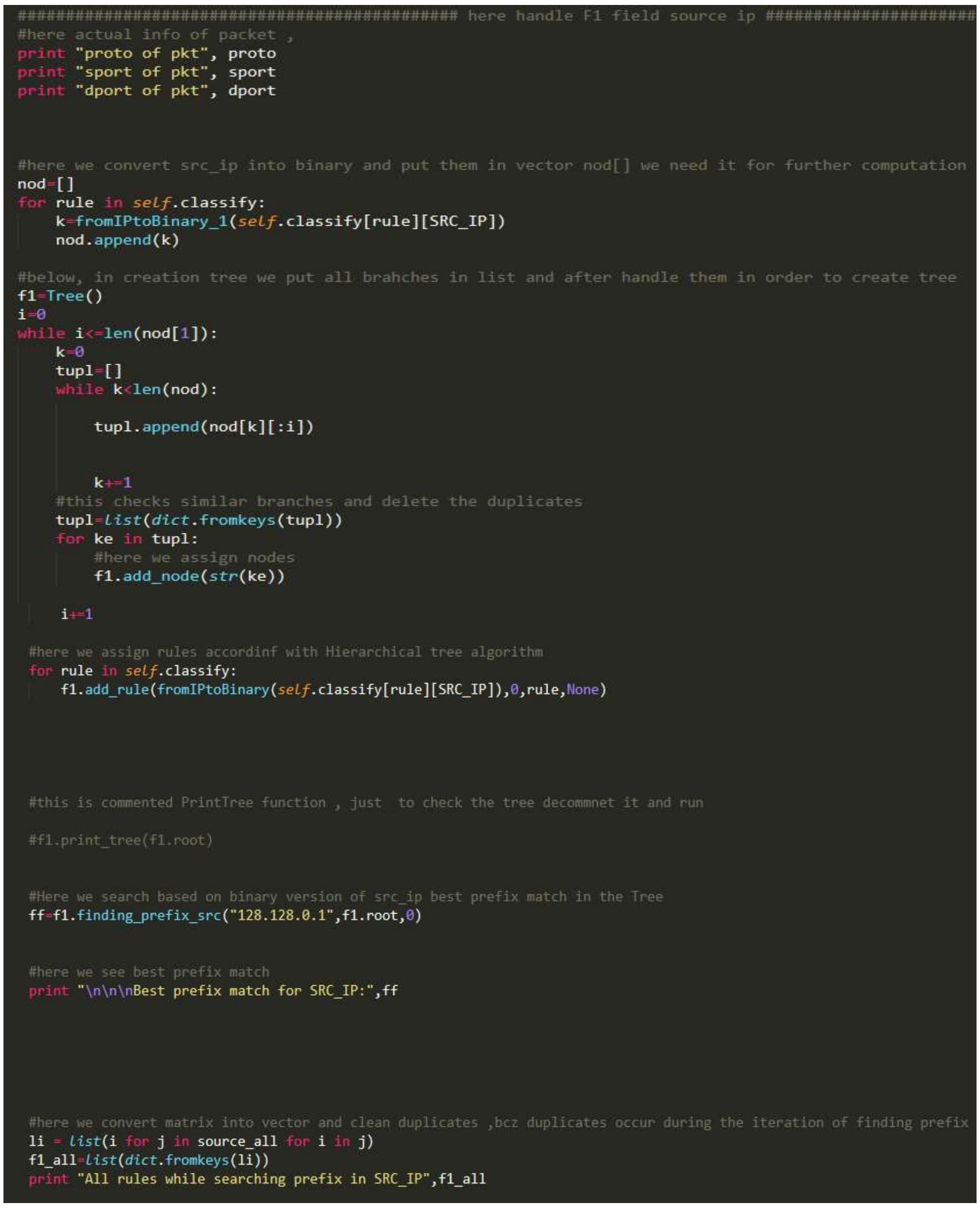
### 5.2.2 From IP to Binary

This function is common to every algorithm it is decided to run and it is a trasformation needed at the beginning of the algorithm, right after the generation of the tree. It's a preset function to handle addresses we have insert in all the algorithms.

So basically: first, generation of the tree; second, IP to binary trasnformation; then classification's algorithm.

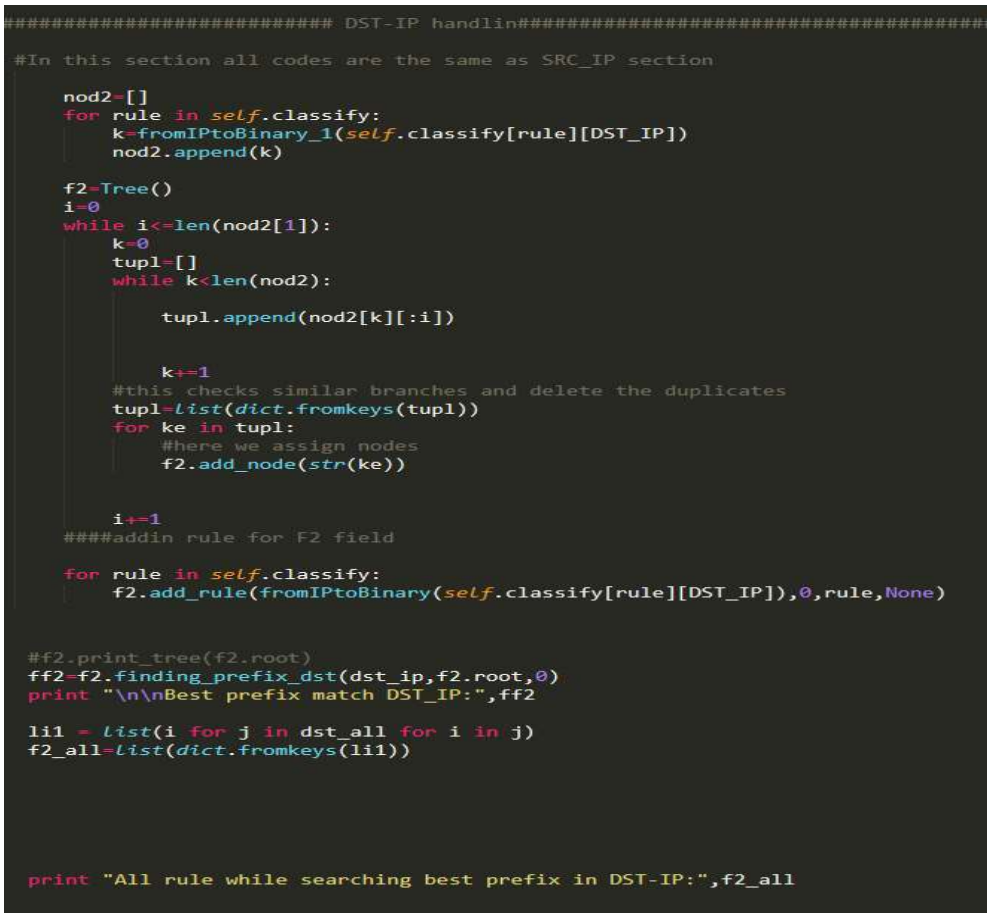
## 5.2.3 Hierarchical algorithm

a) Handling of source IP field of the packet



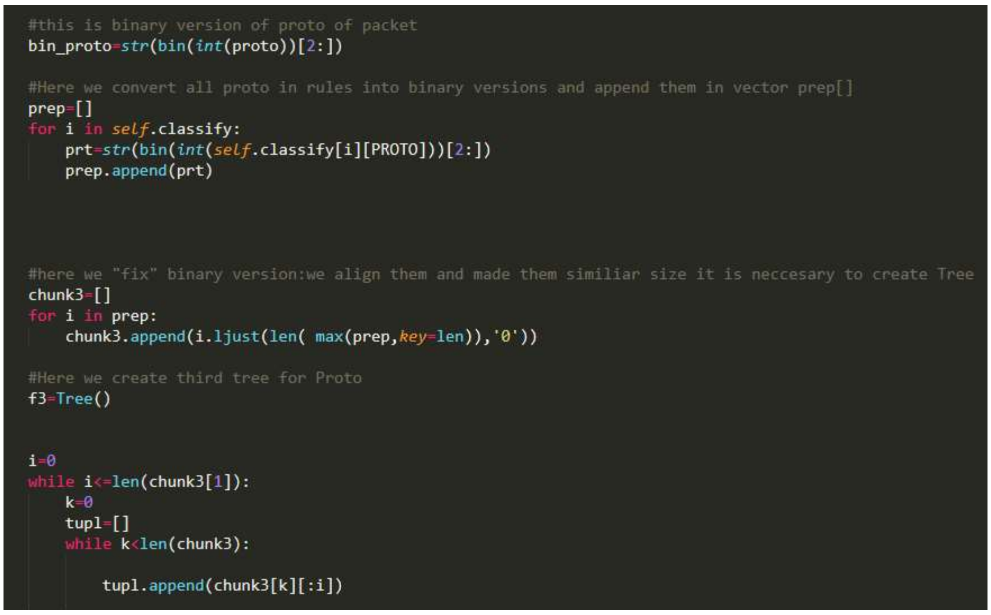
```
############################################## here handle F1 field source ip #######################
#here actual info of packet ,
print "proto of pkt", proto
print "sport of pkt", sport
print "dport of pkt", dport



#here we convert src_ip into binary and put them in vector nod[] we need it for further computation
nod=[]
for rule in self.classify:
    k=fromIPtoBinary_1(self.classify[rule][SRC_IP])
    nod.append(k)

#below, in creation tree we put all brahches in list and after handle them in order to create tree
f1=Tree()
i=0
while i<=len(nod[1]):
    k=0
    tupl=[]
    while k<len(nod):

        tupl.append(nod[k][:i])


        k+=1
    #this checks similar branches and delete the duplicates
    tupl=list(dict.fromkeys(tupl))
    for ke in tupl:
        #here we assign nodes
        f1.add_node(str(ke))

    i+=1

 #here we assign rules accordinf with Hierarchical tree algorithm
 for rule in self.classify:
    f1.add_rule(fromIPtoBinary(self.classify[rule][SRC_IP]),0,rule,None)





 #this is commented PrintTree function , just  to check the tree decommnet it and run

 #f1.print_tree(f1.root)


 #Here we search based on binary version of src_ip best prefix match in the Tree
 ff=f1.finding_prefix_src("128.128.0.1",f1.root,0)


 #here we see best prefix match
 print "\n\n\nBest prefix match for SRC_IP:",ff






 #here we convert matrix into vector and clean duplicates ,bcz duplicates occur during the iteration of finding prefix
 li = list(i for j in source_all for i in j)
 f1_all=list(dict.fromkeys(li))
 print "All rules while searching prefix in SRC_IP",f1_all
```

b) Handling of destination IP field of packet

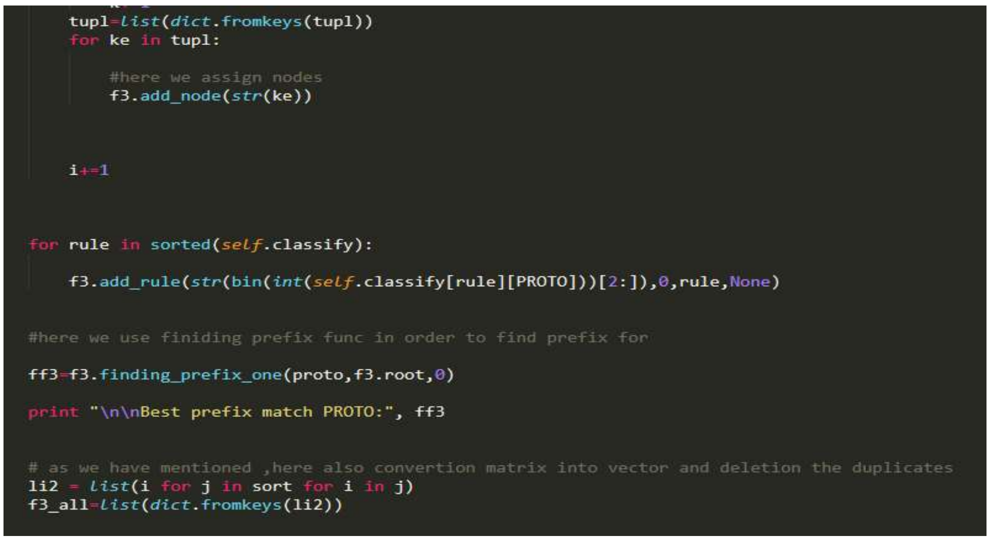


```
############################ DST-IP handlin##########################################

#In this section all codes are the same as SRC_IP section

    nod2=[]
    for rule in self.classify:
        k=fromIPtoBinary_1(self.classify[rule][DST_IP])
        nod2.append(k)

    f2=Tree()
    i=0
    while i<=len(nod2[1]):
        k=0
        tupl=[]
        while k<len(nod2):

            tupl.append(nod2[k][:i])


            k+=1
        #this checks similar branches and delete the duplicates
        tupl=list(dict.fromkeys(tupl))
        for ke in tupl:
            #here we assign nodes
            f2.add_node(str(ke))


        i+=1
    ####addin rule for F2 field

    for rule in self.classify:
        f2.add_rule(fromIPtoBinary(self.classify[rule][DST_IP]),0,rule,None)


 #f2.print_tree(f2.root)
 ff2=f2.finding_prefix_dst(dst_ip,f2.root,0)
 print "\n\nBest prefix match DST_IP:",ff2

 li1 = list(i for j in dst_all for i in j)
 f2_all=list(dict.fromkeys(li1))




 print "All rule while searching best prefix in DST-IP:",f2_all
```

c) Handling of protocol field of packet

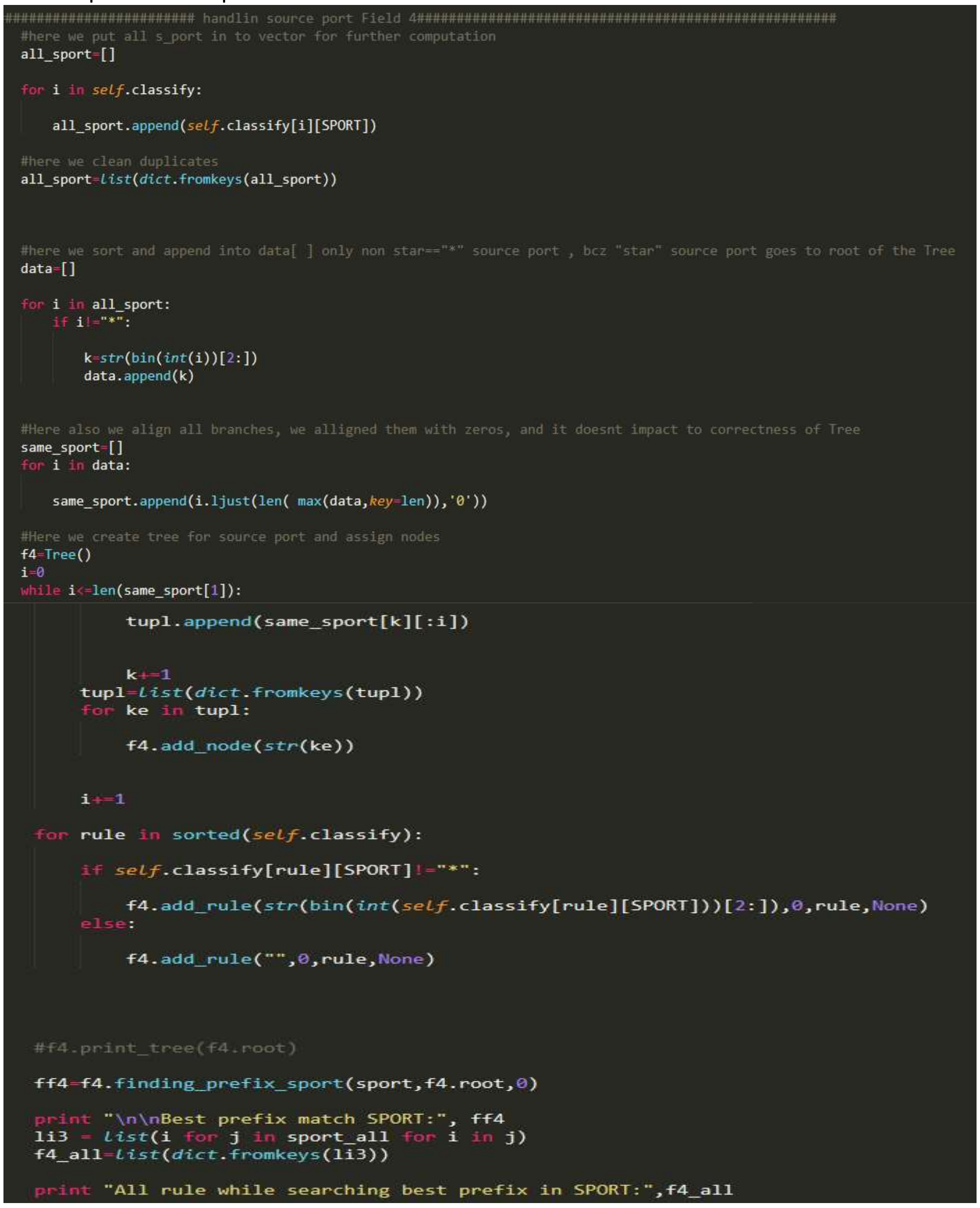


```
#this is binary version of proto of packet
bin_proto=str(bin(int(proto))[2:])

#Here we convert all proto in rules into binary versions and append them in vector prep[]
prep=[]
for i in self.classify:
    prt=str(bin(int(self.classify[i][PROTO]))[2:])
    prep.append(prt)




#here we "fix" binary version:we align them and made them similiar size it is neccesary to create Tree
chunk3=[]
for i in prep:
    chunk3.append(i.ljust(len( max(prep,key=len)),'0'))

#Here we create third tree for Proto
f3=Tree()


i=0
while i<=len(chunk3[1]):
    k=0
    tupl=[]
    while k<len(chunk3):

        tupl.append(chunk3[k][:i])
```



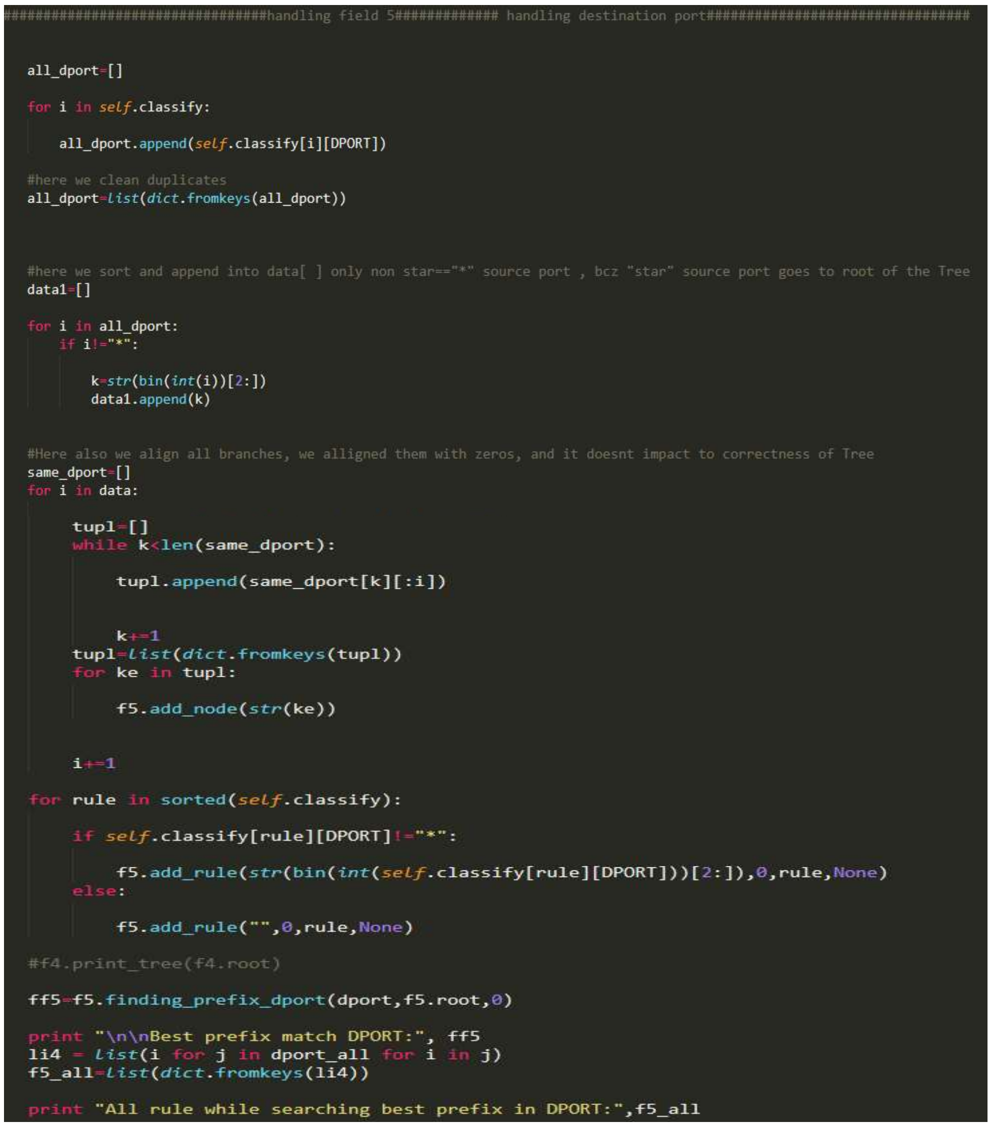
```
        tupl=list(dict.fromkeys(tupl))
        for ke in tupl:

            #here we assign nodes
            f3.add_node(str(ke))



        i+=1



    for rule in sorted(self.classify):

        f3.add_rule(str(bin(int(self.classify[rule][PROTO]))[2:]),0,rule,None)

    #here we use finiding prefix func in order to find prefix for

    ff3=f3.finding_prefix_one(proto,f3.root,0)

    print "\n\nBest prefix match PROTO:", ff3


    # as we have mentioned ,here also convertion matrix into vector and deletion the duplicates
    li2 = list(i for j in sort for i in j)
    f3_all=list(dict.fromkeys(li2))
```

d) Handling of source port field of packet



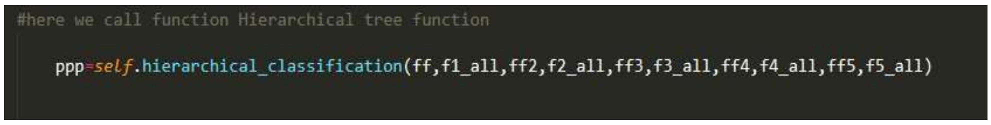
```
######################## handlin source port Field 4#######################################################
  #here we put all s_port in to vector for further computation
  all_sport=[]

  for i in self.classify:

      all_sport.append(self.classify[i][SPORT])

  #here we clean duplicates
  all_sport=list(dict.fromkeys(all_sport))



  #here we sort and append into data[ ] only non star=="*" source port , bcz "star" source port goes to root of the Tree
  data=[]

  for i in all_sport:
      if i!="*":

          k=str(bin(int(i))[2:])
          data.append(k)


  #Here also we align all branches, we alligned them with zeros, and it doesnt impact to correctness of Tree
  same_sport=[]
  for i in data:

      same_sport.append(i.ljust(len( max(data,key=len)),'0'))

  #Here we create tree for source port and assign nodes
  f4=Tree()
  i=0
  while i<=len(same_sport[1]):
            tupl.append(same_sport[k][:i])


            k+=1
        tupl=list(dict.fromkeys(tupl))
        for ke in tupl:

            f4.add_node(str(ke))


        i+=1

    for rule in sorted(self.classify):

        if self.classify[rule][SPORT]!="*":

            f4.add_rule(str(bin(int(self.classify[rule][SPORT]))[2:]),0,rule,None)
        else:

            f4.add_rule("",0,rule,None)




    #f4.print_tree(f4.root)

    ff4=f4.finding_prefix_sport(sport,f4.root,0)

    print "\n\nBest prefix match SPORT:", ff4
    li3 = list(i for j in sport_all for i in j)
    f4_all=list(dict.fromkeys(li3))

    print "All rule while searching best prefix in SPORT:",f4_all
```

e) Handling of destination field of packet



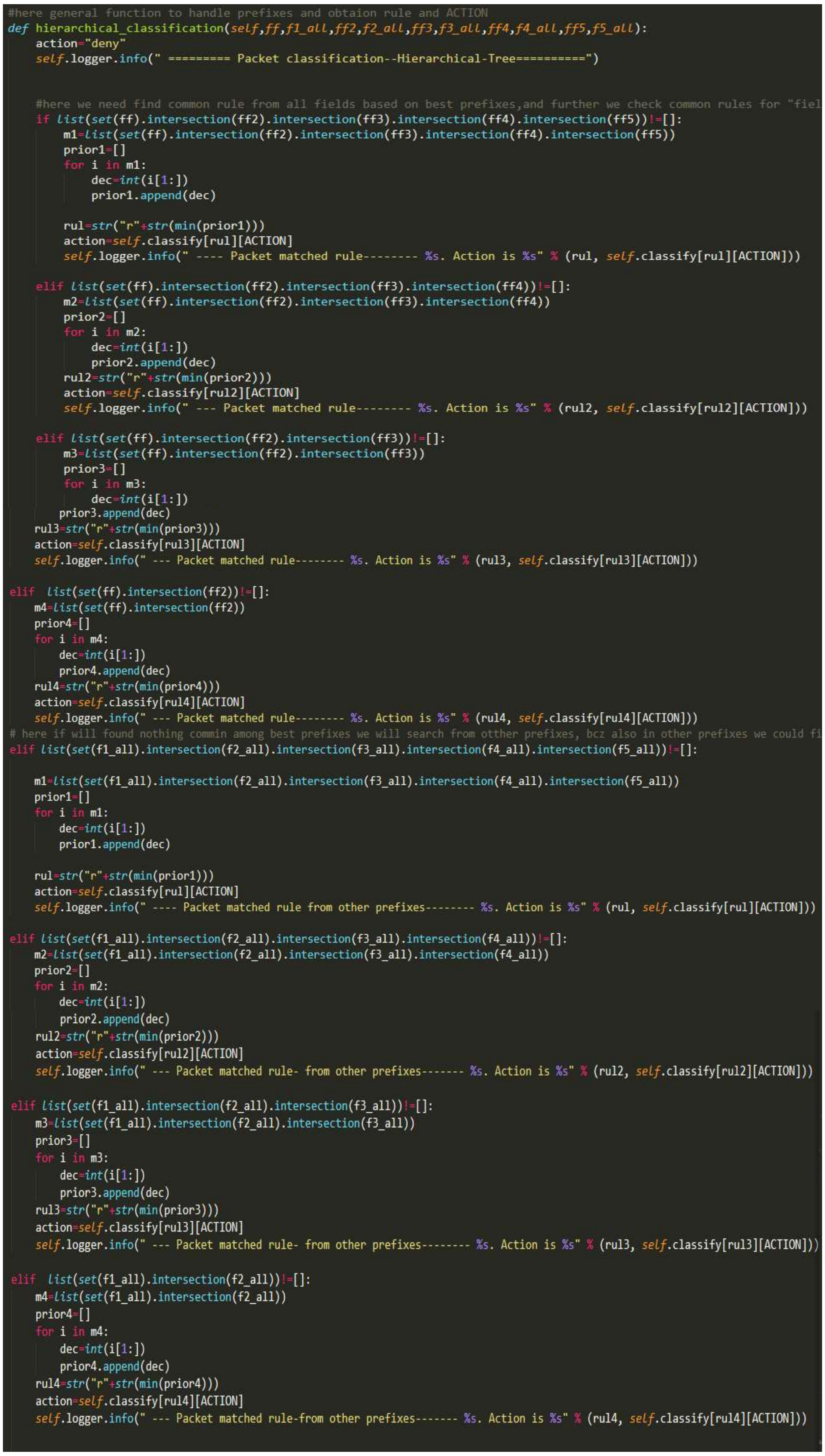
```
#################################handling field 5############# handling destination port##################################


all_dport=[]

for i in self.classify:

    all_dport.append(self.classify[i][DPORT])

#here we clean duplicates
all_dport=list(dict.fromkeys(all_dport))



#here we sort and append into data[ ] only non star=="*" source port , bcz "star" source port goes to root of the Tree
data1=[]

for i in all_dport:
    if i!="*":

        k=str(bin(int(i))[2:])
        data1.append(k)


#Here also we align all branches, we alligned them with zeros, and it doesnt impact to correctness of Tree
same_dport=[]
for i in data:

     tupl=[]
     while k<len(same_dport):

         tupl.append(same_dport[k][:i])


         k+=1
     tupl=list(dict.fromkeys(tupl))
     for ke in tupl:

         f5.add_node(str(ke))


     i+=1

for rule in sorted(self.classify):

     if self.classify[rule][DPORT]!="*":

         f5.add_rule(str(bin(int(self.classify[rule][DPORT]))[2:]),0,rule,None)
     else:

         f5.add_rule("",0,rule,None)

#f4.print_tree(f4.root)

ff5=f5.finding_prefix_dport(dport,f5.root,0)

print "\n\nBest prefix match DPORT:", ff5
li4 = list(i for j in dport_all for i in j)
f5_all=list(dict.fromkeys(li4))

print "All rule while searching best prefix in DPORT:",f5_all
```

f) Call to the function

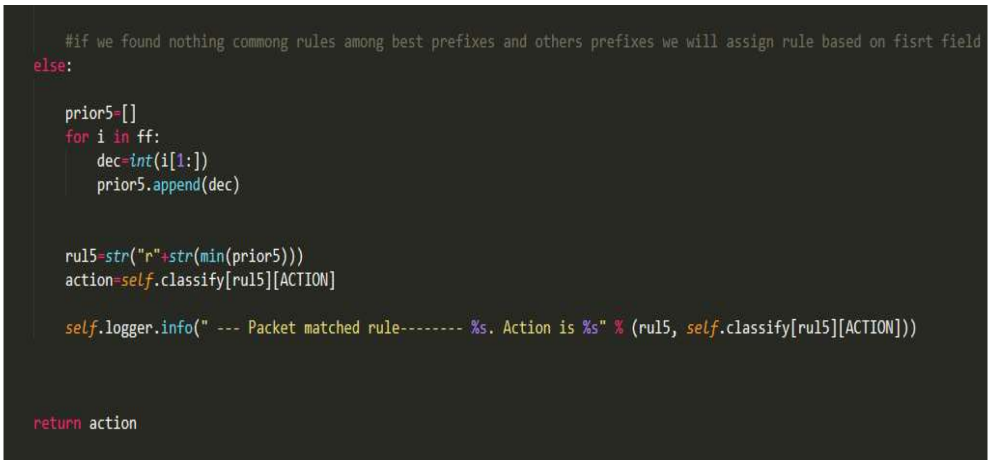


```
#here we call function Hierarchical tree function

    ppp=self.hierarchical_classification(ff,f1_all,ff2,f2_all,ff3,f3_all,ff4,f4_all,ff5,f5_all)
```

e) General function to handle prefixes, obtain rules and the relative action to perform

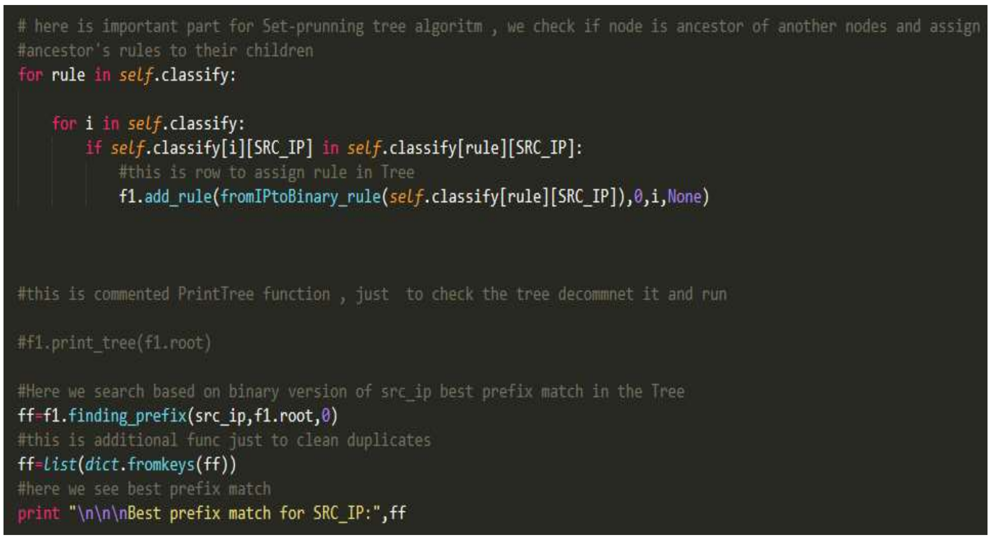


```
#here general function to handle prefixes and obtaion rule and ACTION
def hierarchical_classification(self,ff,f1_all,ff2,f2_all,ff3,f3_all,ff4,f4_all,ff5,f5_all):
    action="deny"
    self.logger.info(" ========= Packet classification--Hierarchical-Tree==========")


    #here we need find common rule from all fields based on best prefixes,and further we check common rules for "fiel
    if list(set(ff).intersection(ff2).intersection(ff3).intersection(ff4).intersection(ff5))!=[]:
        m1=list(set(ff).intersection(ff2).intersection(ff3).intersection(ff4).intersection(ff5))
        prior1=[]
        for i in m1:
            dec=int(i[1:])
            prior1.append(dec)

        rul=str("r"+str(min(prior1)))
        action=self.classify[rul][ACTION]
        self.logger.info(" ---- Packet matched rule-------- %s. Action is %s" % (rul, self.classify[rul][ACTION]))

    elif list(set(ff).intersection(ff2).intersection(ff3).intersection(ff4))!=[]:
        m2=list(set(ff).intersection(ff2).intersection(ff3).intersection(ff4))
        prior2=[]
        for i in m2:
            dec=int(i[1:])
            prior2.append(dec)
        rul2=str("r"+str(min(prior2)))
        action=self.classify[rul2][ACTION]
        self.logger.info(" --- Packet matched rule-------- %s. Action is %s" % (rul2, self.classify[rul2][ACTION]))

    elif list(set(ff).intersection(ff2).intersection(ff3))!=[]:
        m3=list(set(ff).intersection(ff2).intersection(ff3))
        prior3=[]
        for i in m3:
            dec=int(i[1:])
        prior3.append(dec)
    rul3=str("r"+str(min(prior3)))
    action=self.classify[rul3][ACTION]
    self.logger.info(" --- Packet matched rule-------- %s. Action is %s" % (rul3, self.classify[rul3][ACTION]))

elif  list(set(ff).intersection(ff2))!=[]:
    m4=list(set(ff).intersection(ff2))
    prior4=[]
    for i in m4:
        dec=int(i[1:])
        prior4.append(dec)
    rul4=str("r"+str(min(prior4)))
    action=self.classify[rul4][ACTION]
    self.logger.info(" --- Packet matched rule-------- %s. Action is %s" % (rul4, self.classify[rul4][ACTION]))
# here if will found nothing commin among best prefixes we will search from otther prefixes, bcz also in other prefixes we could fi
elif list(set(f1_all).intersection(f2_all).intersection(f3_all).intersection(f4_all).intersection(f5_all))!=[]:

    m1=list(set(f1_all).intersection(f2_all).intersection(f3_all).intersection(f4_all).intersection(f5_all))
    prior1=[]
    for i in m1:
        dec=int(i[1:])
        prior1.append(dec)

    rul=str("r"+str(min(prior1)))
    action=self.classify[rul][ACTION]
    self.logger.info(" ---- Packet matched rule from other prefixes-------- %s. Action is %s" % (rul, self.classify[rul][ACTION]))

elif list(set(f1_all).intersection(f2_all).intersection(f3_all).intersection(f4_all))!=[]:
    m2=list(set(f1_all).intersection(f2_all).intersection(f3_all).intersection(f4_all))
    prior2=[]
    for i in m2:
        dec=int(i[1:])
        prior2.append(dec)
    rul2=str("r"+str(min(prior2)))
    action=self.classify[rul2][ACTION]
    self.logger.info(" --- Packet matched rule- from other prefixes------- %s. Action is %s" % (rul2, self.classify[rul2][ACTION]))

elif list(set(f1_all).intersection(f2_all).intersection(f3_all))!=[]:
    m3=list(set(f1_all).intersection(f2_all).intersection(f3_all))
    prior3=[]
    for i in m3:
        dec=int(i[1:])
        prior3.append(dec)
    rul3=str("r"+str(min(prior3)))
    action=self.classify[rul3][ACTION]
    self.logger.info(" --- Packet matched rule- from other prefixes-------- %s. Action is %s" % (rul3, self.classify[rul3][ACTION]))

elif  list(set(f1_all).intersection(f2_all))!=[]:
    m4=list(set(f1_all).intersection(f2_all))
    prior4=[]
    for i in m4:
        dec=int(i[1:])
        prior4.append(dec)
    rul4=str("r"+str(min(prior4)))
    action=self.classify[rul4][ACTION]
    self.logger.info(" --- Packet matched rule-from other prefixes------- %s. Action is %s" % (rul4, self.classify[rul4][ACTION]))
```

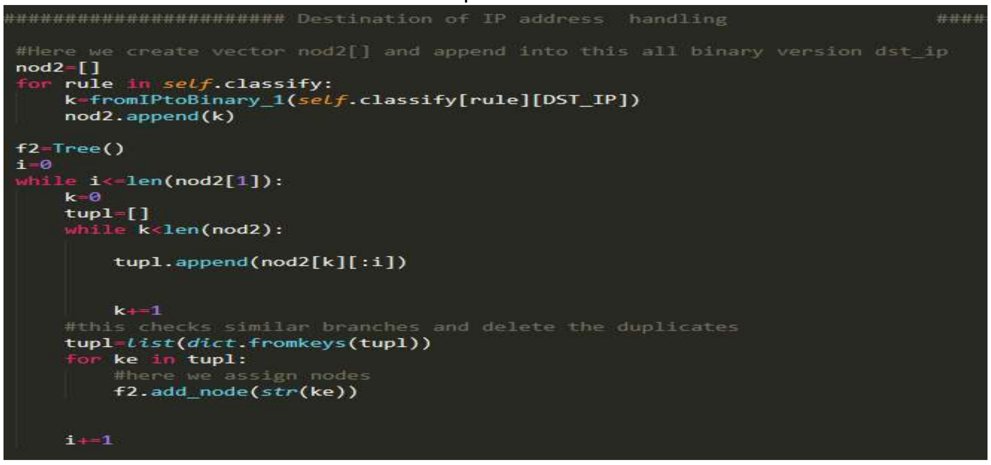

```python
    #if we found nothing commong rules among best prefixes and others prefixes we will assign rule based on fisrt field
else:

    prior5=[]
    for i in ff:
        dec=int(i[1:])
        prior5.append(dec)


    rul5=str("r"+str(min(prior5)))
    action=self.classify[rul5][ACTION]

    self.logger.info(" --- Packet matched rule-------- %s. Action is %s" % (rul5, self.classify[rul5][ACTION]))


return action
```

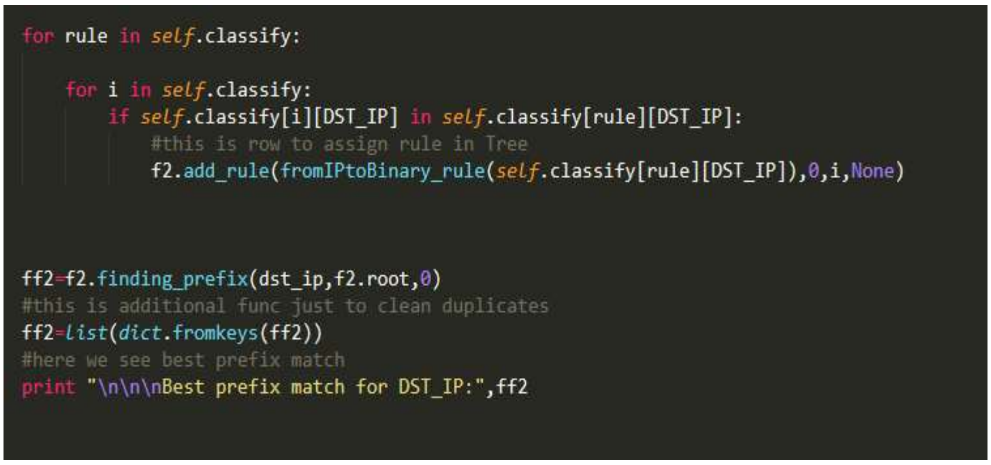
### 5.2.4 Set-pruning trie

a) First thing to do is the creation of the specific tree for the set pruning, that is assign the rules to the tree already created:


```python
# here is important part for Set-prunning tree algoritm , we check if node is ancestor of another nodes and assign
#ancestor's rules to their children
for rule in self.classify:

    for i in self.classify:
        if self.classify[i][SRC_IP] in self.classify[rule][SRC_IP]:
            #this is row to assign rule in Tree
            f1.add_rule(fromIPtoBinary_rule(self.classify[rule][SRC_IP]),0,i,None)




#this is commented PrintTree function , just  to check the tree decommnet it and run

#f1.print_tree(f1.root)

#Here we search based on binary version of src_ip best prefix match in the Tree
ff=f1.finding_prefix(src_ip,f1.root,0)
#this is additional func just to clean duplicates
ff=list(dict.fromkeys(ff))
#here we see best prefix match
print "\n\n\nBest prefix match for SRC_IP:",ff
```

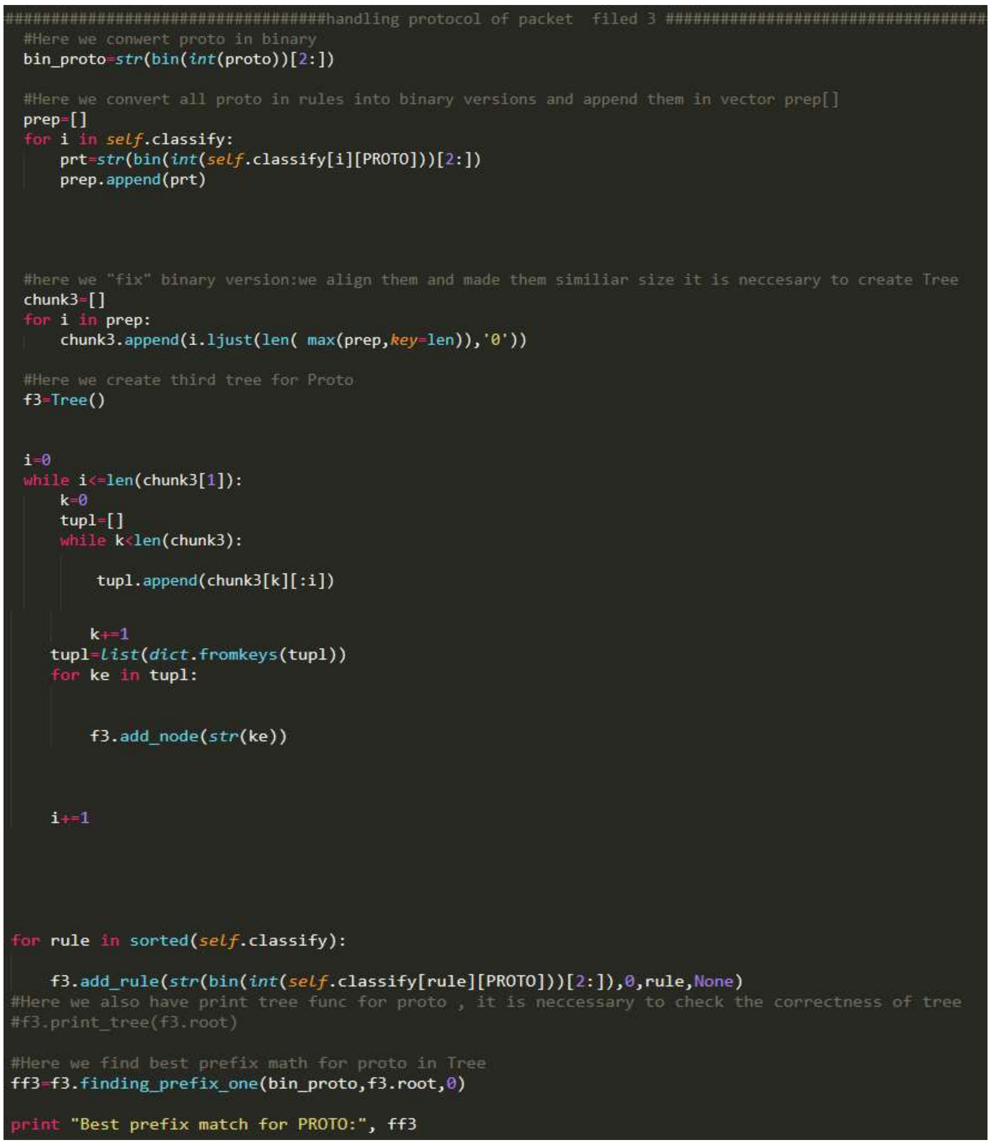
b) Handling of Destination IP address in the field of the packet:


```python
######################## Destination of IP address  handling                ####

#Here we create vector nod2[] and append into this all binary version dst_ip
nod2=[]
for rule in self.classify:
    k=fromIPtoBinary_1(self.classify[rule][DST_IP])
    nod2.append(k)

f2=Tree()
i=0
while i<=len(nod2[1]):
    k=0
    tupl=[]
    while k<len(nod2):

        tupl.append(nod2[k][:i])


        k+=1
    #this checks similar branches and delete the duplicates
    tupl=list(dict.fromkeys(tupl))
    for ke in tupl:
        #here we assign nodes
        f2.add_node(str(ke))


    i+=1
```



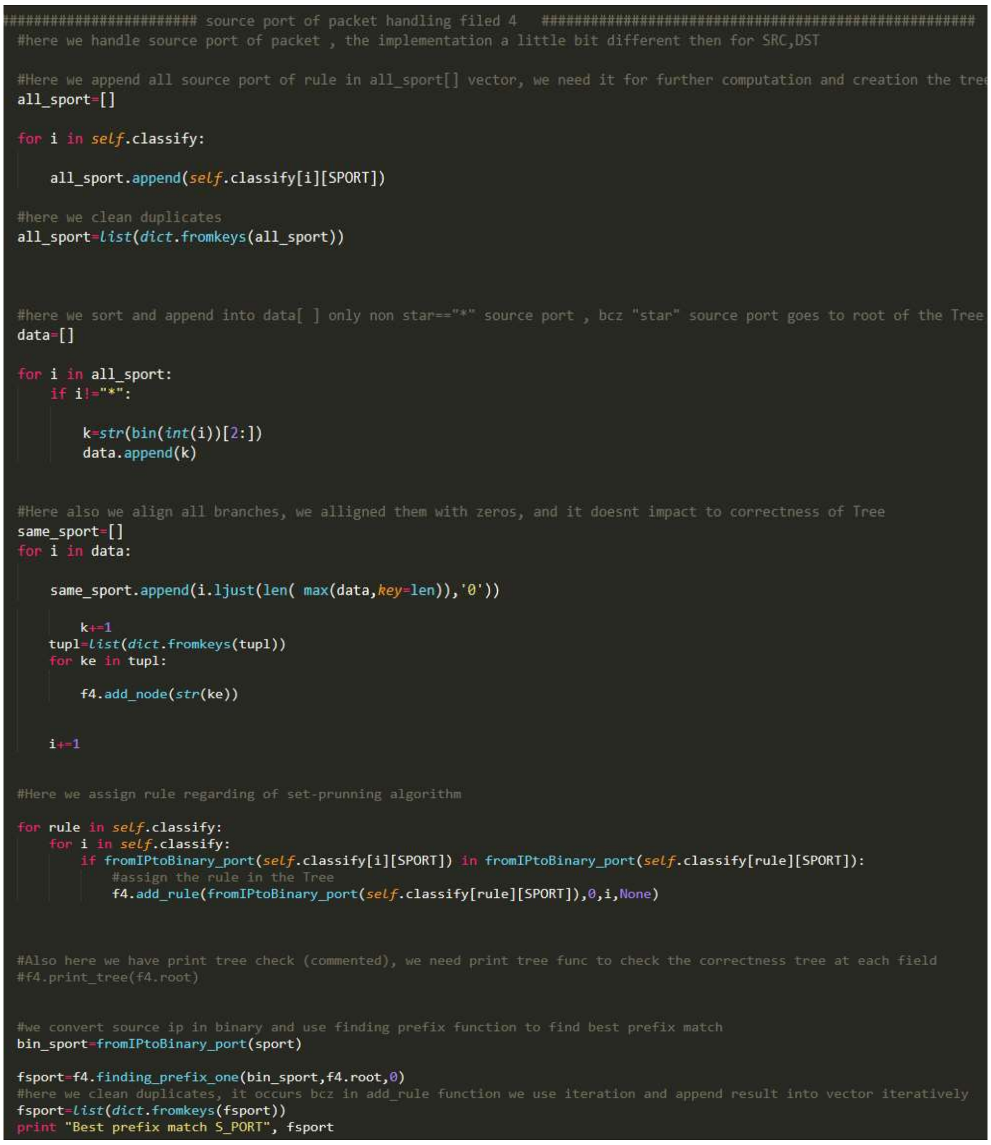
```
for rule in self.classify:

    for i in self.classify:
        if self.classify[i][DST_IP] in self.classify[rule][DST_IP]:
            #this is row to assign rule in Tree
            f2.add_rule(fromIPtoBinary_rule(self.classify[rule][DST_IP]),0,i,None)



ff2=f2.finding_prefix(dst_ip,f2.root,0)
#this is additional func just to clean duplicates
ff2=list(dict.fromkeys(ff2))
#here we see best prefix match
print "\n\n\nBest prefix match for DST_IP:",ff2
```

c) Handling of the protocoll in the field of the packet:



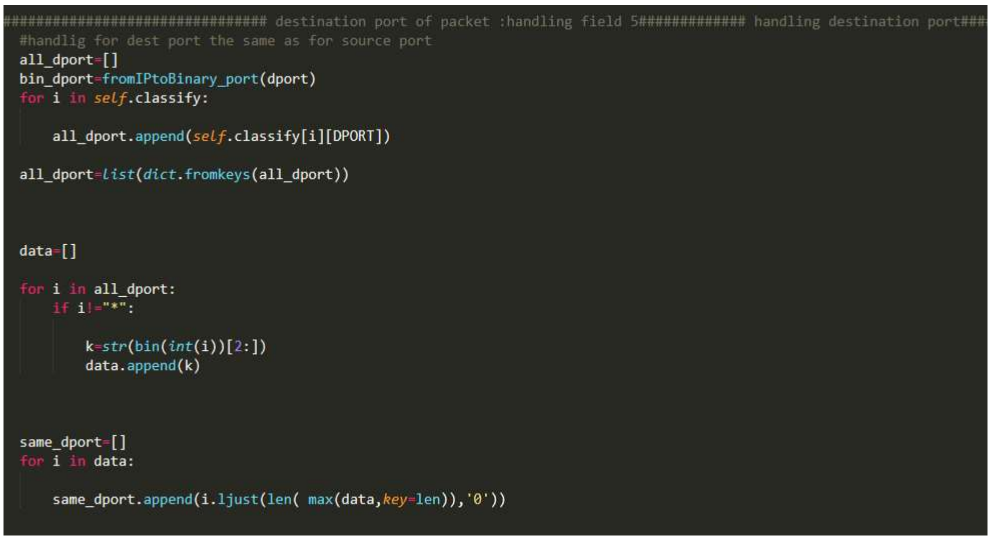
```
####################################handling protocol of packet  filed 3 ####################################
  #Here we conwert proto in binary
  bin_proto=str(bin(int(proto))[2:])

  #Here we convert all proto in rules into binary versions and append them in vector prep[]
  prep=[]
  for i in self.classify:
      prt=str(bin(int(self.classify[i][PROTO]))[2:])
      prep.append(prt)




  #here we "fix" binary version:we align them and made them similiar size it is neccesary to create Tree
  chunk3=[]
  for i in prep:
      chunk3.append(i.ljust(len( max(prep,key=len)),'0'))

  #Here we create third tree for Proto
  f3=Tree()


  i=0
  while i<=len(chunk3[1]):
      k=0
      tupl=[]
      while k<len(chunk3):

          tupl.append(chunk3[k][:i])


         k+=1
     tupl=list(dict.fromkeys(tupl))
     for ke in tupl:


         f3.add_node(str(ke))



     i+=1





for rule in sorted(self.classify):

    f3.add_rule(str(bin(int(self.classify[rule][PROTO]))[2:]),0,rule,None)
#Here we also have print tree func for proto , it is neccessary to check the correctness of tree
#f3.print_tree(f3.root)

#Here we find best prefix math for proto in Tree
ff3=f3.finding_prefix_one(bin_proto,f3.root,0)

print "Best prefix match for PROTO:", ff3
```

d) Handling of the source port:

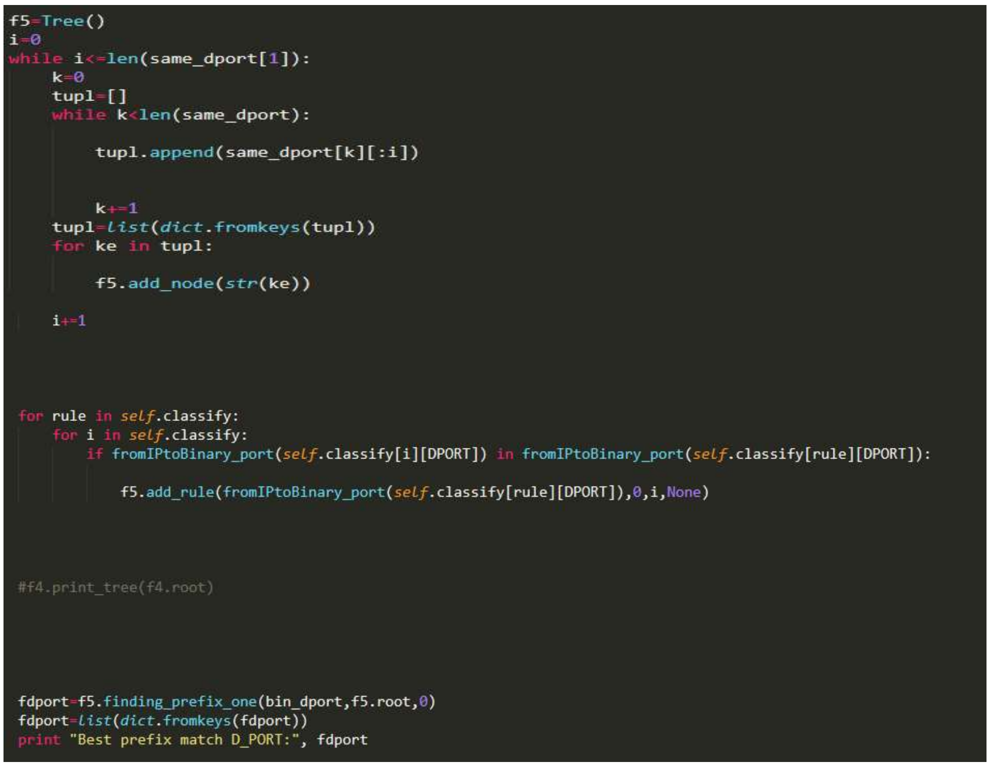


```
######################## source port of packet handling filed 4   #####################################################
 #here we handle source port of packet , the implementation a little bit different then for SRC,DST

 #Here we append all source port of rule in all_sport[] vector, we need it for further computation and creation the tree
 all_sport=[]

 for i in self.classify:

     all_sport.append(self.classify[i][SPORT])

 #here we clean duplicates
 all_sport=list(dict.fromkeys(all_sport))



 #here we sort and append into data[ ] only non star=="*" source port , bcz "star" source port goes to root of the Tree
 data=[]

 for i in all_sport:
     if i!="*":

         k=str(bin(int(i))[2:])
         data.append(k)


 #Here also we align all branches, we alligned them with zeros, and it doesnt impact to correctness of Tree
 same_sport=[]
 for i in data:

     same_sport.append(i.ljust(len( max(data,key=len)),'0'))

         k+=1
     tupl=list(dict.fromkeys(tupl))
     for ke in tupl:

         f4.add_node(str(ke))


     i+=1


 #Here we assign rule regarding of set-prunning algorithm

 for rule in self.classify:
     for i in self.classify:
         if fromIPtoBinary_port(self.classify[i][SPORT]) in fromIPtoBinary_port(self.classify[rule][SPORT]):
             #assign the rule in the Tree
             f4.add_rule(fromIPtoBinary_port(self.classify[rule][SPORT]),0,i,None)



 #Also here we have print tree check (commented), we need print tree func to check the correctness tree at each field
 #f4.print_tree(f4.root)


 #we convert source ip in binary and use finding prefix function to find best prefix match
 bin_sport=fromIPtoBinary_port(sport)

 fsport=f4.finding_prefix_one(bin_sport,f4.root,0)
 #here we clean duplicates, it occurs bcz in add_rule function we use iteration and append result into vector iteratively
 fsport=list(dict.fromkeys(fsport))
 print "Best prefix match S_PORT", fsport
```

e) Handling of the destination port:

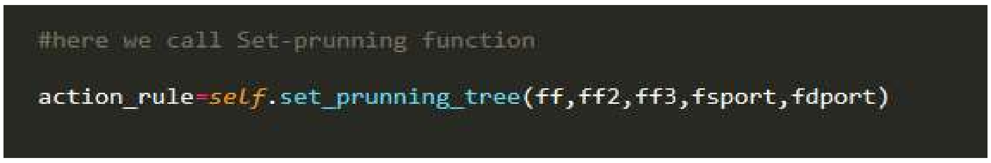


```
################################ destination port of packet :handling field 5############# handling destination port###
 #handlig for dest port the same as for source port
 all_dport=[]
 bin_dport=fromIPtoBinary_port(dport)
 for i in self.classify:

     all_dport.append(self.classify[i][DPORT])

 all_dport=list(dict.fromkeys(all_dport))



 data=[]

 for i in all_dport:
     if i!="*":

         k=str(bin(int(i))[2:])
         data.append(k)



 same_dport=[]
 for i in data:

     same_dport.append(i.ljust(len( max(data,key=len)),'0'))
```



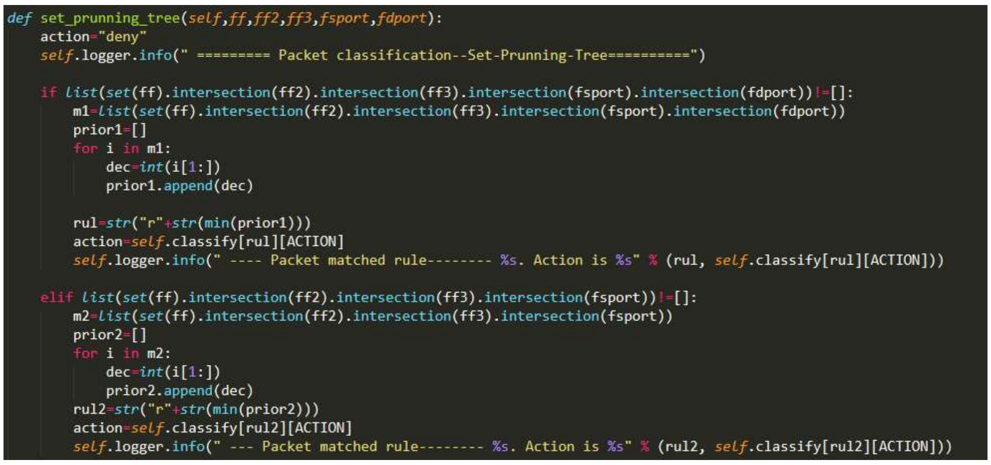
```
f5=Tree()
i=0
while i<=len(same_dport[1]):
    k=0
    tupl=[]
    while k<len(same_dport):

        tupl.append(same_dport[k][:i])


        k+=1
    tupl=list(dict.fromkeys(tupl))
    for ke in tupl:

        f5.add_node(str(ke))

    i+=1



for rule in self.classify:
    for i in self.classify:
        if fromIPtoBinary_port(self.classify[i][DPORT]) in fromIPtoBinary_port(self.classify[rule][DPORT]):

            f5.add_rule(fromIPtoBinary_port(self.classify[rule][DPORT]),0,i,None)




#f4.print_tree(f4.root)





fdport=f5.finding_prefix_one(bin_dport,f5.root,0)
fdport=list(dict.fromkeys(fdport))
print "Best prefix match D_PORT:", fdport
```

f) Call to the set-pruning function:

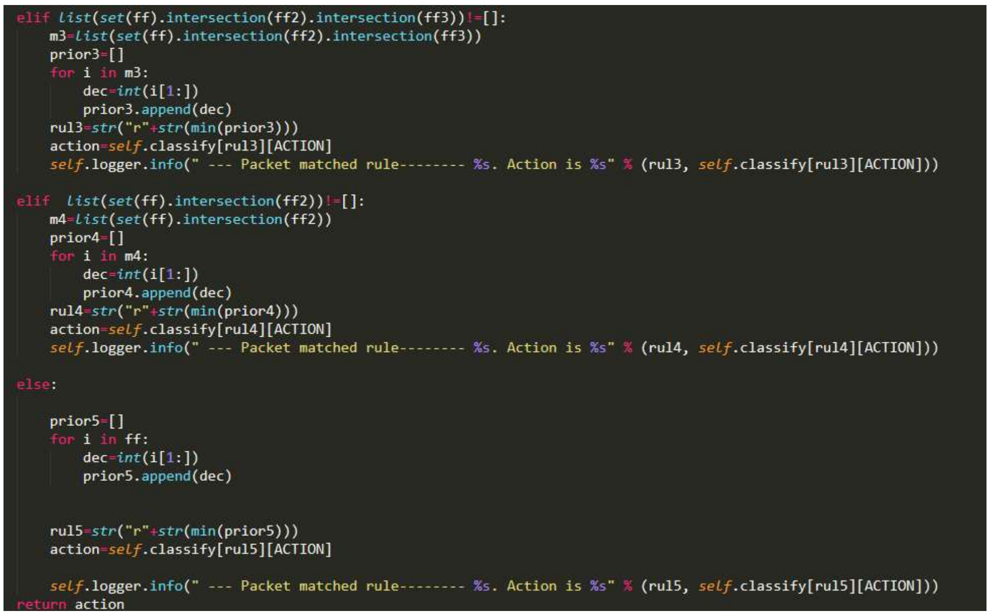


```
#here we call Set-prunning function

action_rule=self.set_prunning_tree(ff,ff2,ff3,fsport,fdport)
```

g) Handling results of the tree regarding the algorithm and the priorities:



```
def set_prunning_tree(self,ff,ff2,ff3,fsport,fdport):
    action="deny"
    self.logger.info(" ========= Packet classification--Set-Prunning-Tree==========")

    if list(set(ff).intersection(ff2).intersection(ff3).intersection(fsport).intersection(fdport))!=[]:
        m1=list(set(ff).intersection(ff2).intersection(ff3).intersection(fsport).intersection(fdport))
        prior1=[]
        for i in m1:
            dec=int(i[1:])
            prior1.append(dec)

        rul=str("r"+str(min(prior1)))
        action=self.classify[rul][ACTION]
        self.logger.info(" ---- Packet matched rule-------- %s. Action is %s" % (rul, self.classify[rul][ACTION]))

    elif list(set(ff).intersection(ff2).intersection(ff3).intersection(fsport))!=[]:
        m2=list(set(ff).intersection(ff2).intersection(ff3).intersection(fsport))
        prior2=[]
        for i in m2:
            dec=int(i[1:])
            prior2.append(dec)
        rul2=str("r"+str(min(prior2)))
        action=self.classify[rul2][ACTION]
        self.logger.info(" --- Packet matched rule-------- %s. Action is %s" % (rul2, self.classify[rul2][ACTION]))
```



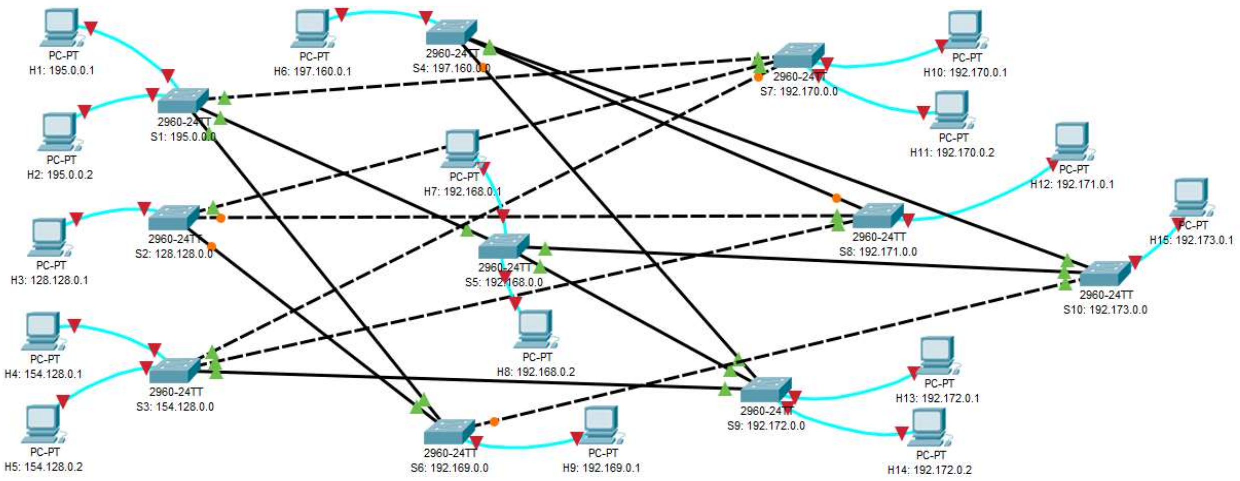
```python
elif list(set(ff).intersection(ff2).intersection(ff3))!=[]:
    m3=list(set(ff).intersection(ff2).intersection(ff3))
    prior3=[]
    for i in m3:
        dec=int(i[1:])
        prior3.append(dec)
    rul3=str("r"+str(min(prior3)))
    action=self.classify[rul3][ACTION]
    self.logger.info(" --- Packet matched rule-------- %s. Action is %s" % (rul3, self.classify[rul3][ACTION]))

elif  list(set(ff).intersection(ff2))!=[]:
    m4=list(set(ff).intersection(ff2))
    prior4=[]
    for i in m4:
        dec=int(i[1:])
        prior4.append(dec)
    rul4=str("r"+str(min(prior4)))
    action=self.classify[rul4][ACTION]
    self.logger.info(" --- Packet matched rule-------- %s. Action is %s" % (rul4, self.classify[rul4][ACTION]))

else:

    prior5=[]
    for i in ff:
        dec=int(i[1:])
        prior5.append(dec)


    rul5=str("r"+str(min(prior5)))
    action=self.classify[rul5][ACTION]

    self.logger.info(" --- Packet matched rule-------- %s. Action is %s" % (rul5, self.classify[rul5][ACTION]))
return action
```

## 6 SETUP

In this section are explained all the complementary files and implementations we have used to create the topologies and the IP addressing to link hosts and switches together (only the third one is presented since the first and the second were already coded in the default version of program) and the set of rules we have written for the packet classification.

## 6.1 Topology n°3

### 6.1.1 Mapping

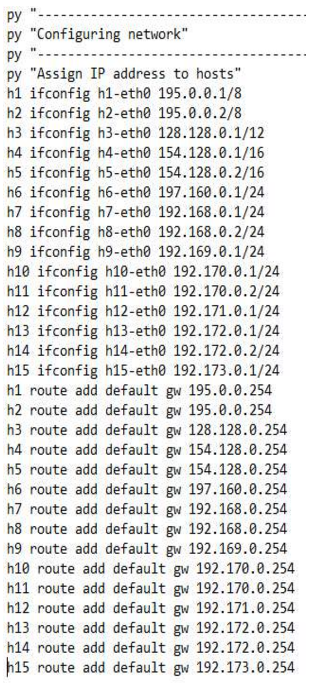


| Switch → Host | |
|---|---|
| S1 {H1,H2} | S6 {H9} |
| S2 {H3} | S7 {H10,H11} |
| S3 {H4,H5} | S8 {H12} |
| S4 {H6} | S9 {H13,H14} |
| S5 {H7,H8} | S10 {H15} |

| Switch → Switch | |
|---|---|
| S1 {S5,S6,S7} | S6 {S10,S1,S2} |
| S2 {S6,S7,S8} | S7 {S1,S2,S3} |
| S3 {S7,S8,S9} | S8 {S2,S3,S4} |
| S4 {S8,S9,S10} | S9 {S3,S4,S5} |
| S5 {S9,S10,S1} | S10 {S4,S5,S6} |

### 6.1.2 Configuration file

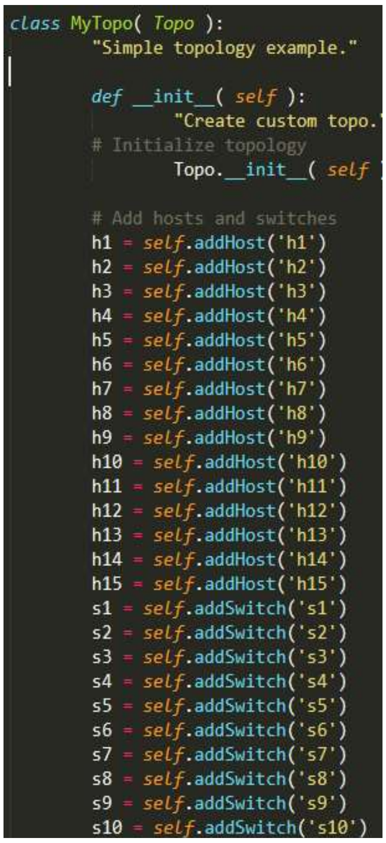
Simple text file where we first configure the host assigning them IP addresses, type of link and default gateways



```
py "----------------------------------
py "Configuring network"
py "----------------------------------
py "Assign IP address to hosts"
h1 ifconfig h1-eth0 195.0.0.1/8
h2 ifconfig h2-eth0 195.0.0.2/8
h3 ifconfig h3-eth0 128.128.0.1/12
h4 ifconfig h4-eth0 154.128.0.1/16
h5 ifconfig h5-eth0 154.128.0.2/16
h6 ifconfig h6-eth0 197.160.0.1/24
h7 ifconfig h7-eth0 192.168.0.1/24
h8 ifconfig h8-eth0 192.168.0.2/24
h9 ifconfig h9-eth0 192.169.0.1/24
h10 ifconfig h10-eth0 192.170.0.1/24
h11 ifconfig h11-eth0 192.170.0.2/24
h12 ifconfig h12-eth0 192.171.0.1/24
h13 ifconfig h13-eth0 192.172.0.1/24
h14 ifconfig h14-eth0 192.172.0.2/24
h15 ifconfig h15-eth0 192.173.0.1/24
h1 route add default gw 195.0.0.254
h2 route add default gw 195.0.0.254
h3 route add default gw 128.128.0.254
h4 route add default gw 154.128.0.254
h5 route add default gw 154.128.0.254
h6 route add default gw 197.160.0.254
h7 route add default gw 192.168.0.254
h8 route add default gw 192.168.0.254
h9 route add default gw 192.169.0.254
h10 route add default gw 192.170.0.254
h11 route add default gw 192.170.0.254
h12 route add default gw 192.171.0.254
h13 route add default gw 192.172.0.254
h14 route add default gw 192.172.0.254
h15 route add default gw 192.173.0.254
```

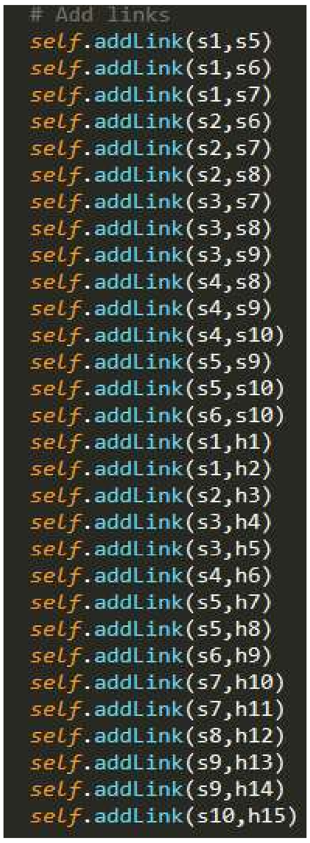
### 6.1.3 Creation of the topology

Here is the Python code for the creation of the topology itself



```
class MyTopo( Topo ):
        "Simple topology example."

        def __init__( self ):
                "Create custom topo.
        # Initialize topology
                Topo.__init__( self

        # Add hosts and switches
        h1 = self.addHost('h1')
        h2 = self.addHost('h2')
        h3 = self.addHost('h3')
        h4 = self.addHost('h4')
        h5 = self.addHost('h5')
        h6 = self.addHost('h6')
        h7 = self.addHost('h7')
        h8 = self.addHost('h8')
        h9 = self.addHost('h9')
        h10 = self.addHost('h10')
        h11 = self.addHost('h11')
        h12 = self.addHost('h12')
        h13 = self.addHost('h13')
        h14 = self.addHost('h14')
        h15 = self.addHost('h15')
        s1 = self.addSwitch('s1')
        s2 = self.addSwitch('s2')
        s3 = self.addSwitch('s3')
        s4 = self.addSwitch('s4')
        s5 = self.addSwitch('s5')
        s6 = self.addSwitch('s6')
        s7 = self.addSwitch('s7')
        s8 = self.addSwitch('s8')
        s9 = self.addSwitch('s9')
        s10 = self.addSwitch('s10')
```

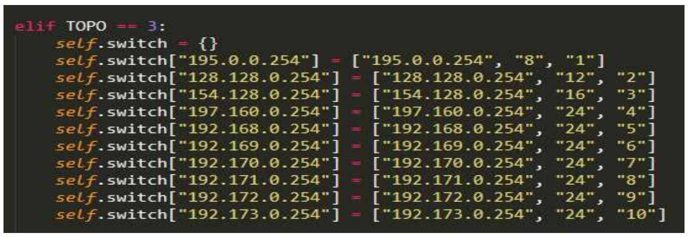
creation and naming of hosts and switches



```
# Add links
self.addLink(s1,s5)
self.addLink(s1,s6)
self.addLink(s1,s7)
self.addLink(s2,s6)
self.addLink(s2,s7)
self.addLink(s2,s8)
self.addLink(s3,s7)
self.addLink(s3,s8)
self.addLink(s3,s9)
self.addLink(s4,s8)
self.addLink(s4,s9)
self.addLink(s4,s10)
self.addLink(s5,s9)
self.addLink(s5,s10)
self.addLink(s6,s10)
self.addLink(s1,h1)
self.addLink(s1,h2)
self.addLink(s2,h3)
self.addLink(s3,h4)
self.addLink(s3,h5)
self.addLink(s4,h6)
self.addLink(s5,h7)
self.addLink(s5,h8)
self.addLink(s6,h9)
self.addLink(s7,h10)
self.addLink(s7,h11)
self.addLink(s8,h12)
self.addLink(s9,h13)
self.addLink(s9,h14)
self.addLink(s10,h15)
```

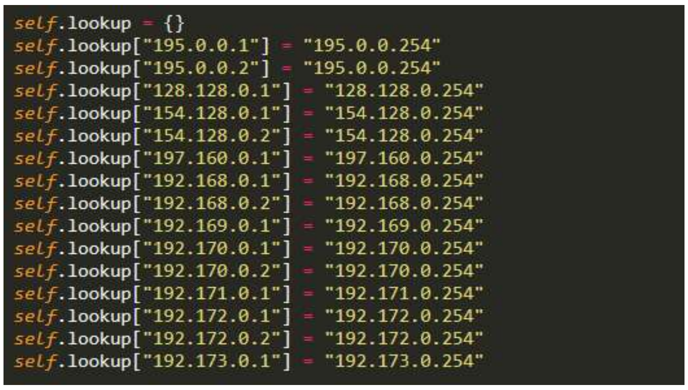
linkage of hosts

### 6.1.4 IP and MAC addressing



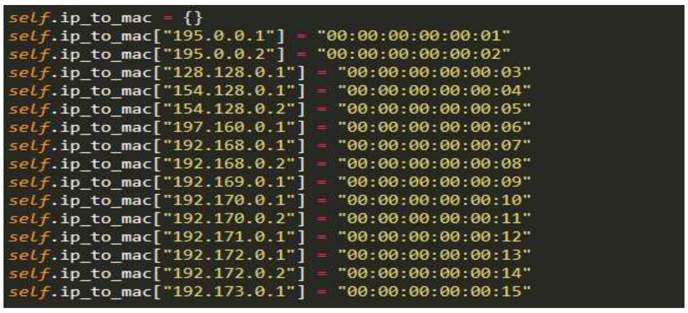
```
elif TOPO == 3:
    self.switch = {}
    self.switch["195.0.0.254"] = ["195.0.0.254", "8", "1"]
    self.switch["128.128.0.254"] = ["128.128.0.254", "12", "2"]
    self.switch["154.128.0.254"] = ["154.128.0.254", "16", "3"]
    self.switch["197.160.0.254"] = ["197.160.0.254", "24", "4"]
    self.switch["192.168.0.254"] = ["192.168.0.254", "24", "5"]
    self.switch["192.169.0.254"] = ["192.169.0.254", "24", "6"]
    self.switch["192.170.0.254"] = ["192.170.0.254", "24", "7"]
    self.switch["192.171.0.254"] = ["192.171.0.254", "24", "8"]
    self.switch["192.172.0.254"] = ["192.172.0.254", "24", "9"]
    self.switch["192.173.0.254"] = ["192.173.0.254", "24", "10"]
```

IP addressing of the switches



```
self.lookup = {}
self.lookup["195.0.0.1"] = "195.0.0.254"
self.lookup["195.0.0.2"] = "195.0.0.254"
self.lookup["128.128.0.1"] = "128.128.0.254"
self.lookup["154.128.0.1"] = "154.128.0.254"
self.lookup["154.128.0.2"] = "154.128.0.254"
self.lookup["197.160.0.1"] = "197.160.0.254"
self.lookup["192.168.0.1"] = "192.168.0.254"
self.lookup["192.168.0.2"] = "192.168.0.254"
self.lookup["192.169.0.1"] = "192.169.0.254"
self.lookup["192.170.0.1"] = "192.170.0.254"
self.lookup["192.170.0.2"] = "192.170.0.254"
self.lookup["192.171.0.1"] = "192.171.0.254"
self.lookup["192.172.0.1"] = "192.172.0.254"
self.lookup["192.172.0.2"] = "192.172.0.254"
self.lookup["192.173.0.1"] = "192.173.0.254"
```

lookup IP addressing

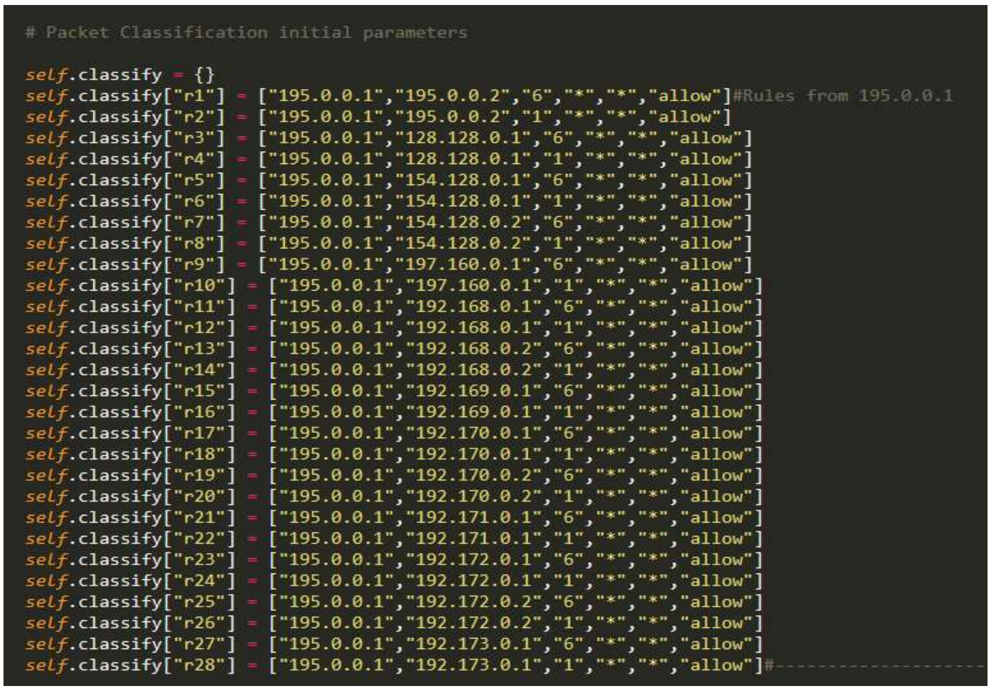


```
self.ip_to_mac = {}
self.ip_to_mac["195.0.0.1"] = "00:00:00:00:00:01"
self.ip_to_mac["195.0.0.2"] = "00:00:00:00:00:02"
self.ip_to_mac["128.128.0.1"] = "00:00:00:00:00:03"
self.ip_to_mac["154.128.0.1"] = "00:00:00:00:00:04"
self.ip_to_mac["154.128.0.2"] = "00:00:00:00:00:05"
self.ip_to_mac["197.160.0.1"] = "00:00:00:00:00:06"
self.ip_to_mac["192.168.0.1"] = "00:00:00:00:00:07"
self.ip_to_mac["192.168.0.2"] = "00:00:00:00:00:08"
self.ip_to_mac["192.169.0.1"] = "00:00:00:00:00:09"
self.ip_to_mac["192.170.0.1"] = "00:00:00:00:00:10"
self.ip_to_mac["192.170.0.2"] = "00:00:00:00:00:11"
self.ip_to_mac["192.171.0.1"] = "00:00:00:00:00:12"
self.ip_to_mac["192.172.0.1"] = "00:00:00:00:00:13"
self.ip_to_mac["192.172.0.2"] = "00:00:00:00:00:14"
self.ip_to_mac["192.173.0.1"] = "00:00:00:00:00:15"
```

MAC addressing

## 6.2 Set of rules

In this section are shown the rules that are scanned by the classifier to find the best match for the incoming packet. Take into account that:
1- only rules for host n°1 are here presented as example
2- all the rules are first set to "allow" action, but they can be changed by simply setting the "deny" action to see a different result while performing the algorithms



```
# Packet Classification initial parameters

self.classify = {}
self.classify["r1"] = ["195.0.0.1","195.0.0.2","6","*","*","allow"]#Rules from 195.0.0.1
self.classify["r2"] = ["195.0.0.1","195.0.0.2","1","*","*","allow"]
self.classify["r3"] = ["195.0.0.1","128.128.0.1","6","*","*","allow"]
self.classify["r4"] = ["195.0.0.1","128.128.0.1","1","*","*","allow"]
self.classify["r5"] = ["195.0.0.1","154.128.0.1","6","*","*","allow"]
self.classify["r6"] = ["195.0.0.1","154.128.0.1","1","*","*","allow"]
self.classify["r7"] = ["195.0.0.1","154.128.0.2","6","*","*","allow"]
self.classify["r8"] = ["195.0.0.1","154.128.0.2","1","*","*","allow"]
self.classify["r9"] = ["195.0.0.1","197.160.0.1","6","*","*","allow"]
self.classify["r10"] = ["195.0.0.1","197.160.0.1","1","*","*","allow"]
self.classify["r11"] = ["195.0.0.1","192.168.0.1","6","*","*","allow"]
self.classify["r12"] = ["195.0.0.1","192.168.0.1","1","*","*","allow"]
self.classify["r13"] = ["195.0.0.1","192.168.0.2","6","*","*","allow"]
self.classify["r14"] = ["195.0.0.1","192.168.0.2","1","*","*","allow"]
self.classify["r15"] = ["195.0.0.1","192.169.0.1","6","*","*","allow"]
self.classify["r16"] = ["195.0.0.1","192.169.0.1","1","*","*","allow"]
self.classify["r17"] = ["195.0.0.1","192.170.0.1","6","*","*","allow"]
self.classify["r18"] = ["195.0.0.1","192.170.0.1","1","*","*","allow"]
self.classify["r19"] = ["195.0.0.1","192.170.0.2","6","*","*","allow"]
self.classify["r20"] = ["195.0.0.1","192.170.0.2","1","*","*","allow"]
self.classify["r21"] = ["195.0.0.1","192.171.0.1","6","*","*","allow"]
self.classify["r22"] = ["195.0.0.1","192.171.0.1","1","*","*","allow"]
self.classify["r23"] = ["195.0.0.1","192.172.0.1","6","*","*","allow"]
self.classify["r24"] = ["195.0.0.1","192.172.0.1","1","*","*","allow"]
self.classify["r25"] = ["195.0.0.1","192.172.0.2","6","*","*","allow"]
self.classify["r26"] = ["195.0.0.1","192.172.0.2","1","*","*","allow"]
self.classify["r27"] = ["195.0.0.1","192.173.0.1","6","*","*","allow"]
self.classify["r28"] = ["195.0.0.1","192.173.0.1","1","*","*","allow"]#--------------------
```

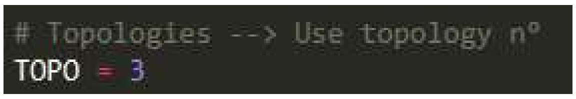
rules from host 1 with IP address 195.0.0.1

# 7 HOW TO RUN THE PROGRAM

In this section there are the set of initial commands to perform to run the program.

## 7.1 Choice of the topology

Change of the parameter 1, 2 or 3 to choose which of the three networks we want the algorithmswork on



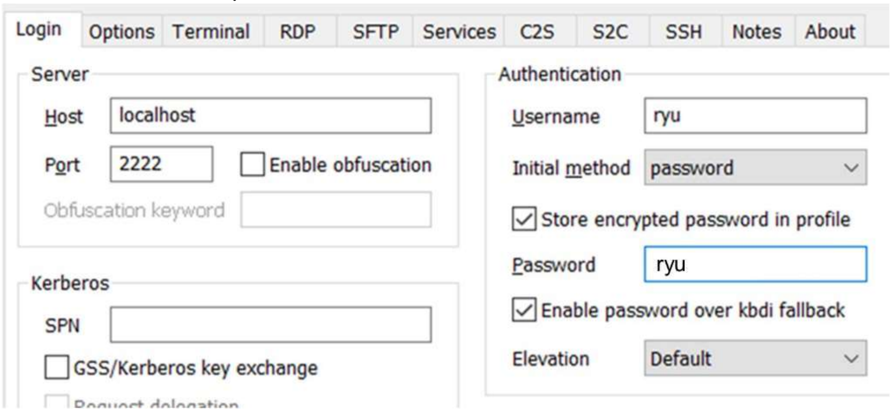
```
# Topologies --> Use topology n°
TOPO = 3
```

## 7.2 Choice of the algorithms

Before launching the program we need to choose which algorithm we want to run for the packet classification.
In order to simplify the coding and do not commit any errors in the handling of variables, we have decided to create one application for each of the algorithms; so basically if we want to run the hierarchical classification the corresponding application has to be run, if set-pruning another application will be run and the same for grid-of-tries. There isn't any common application for all of them.

## 7.3 Demonstration

After starting the virtual machine, you have to launch Bitwise.



Server

Host: localhost

Port: 2222

☐ Enable obfuscation

Obfuscation keyword:

Kerberos

SPN:

☐ GSS/Kerberos key exchange

☐ Request delegation

Authentication

Username: ryu

Initial method: password

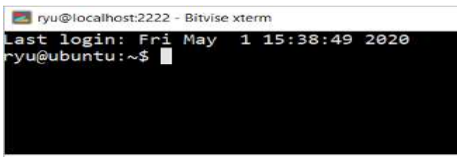
☑ Store encrypted password in profile

Password: ryu

☑ Enable password over kbdi fallback

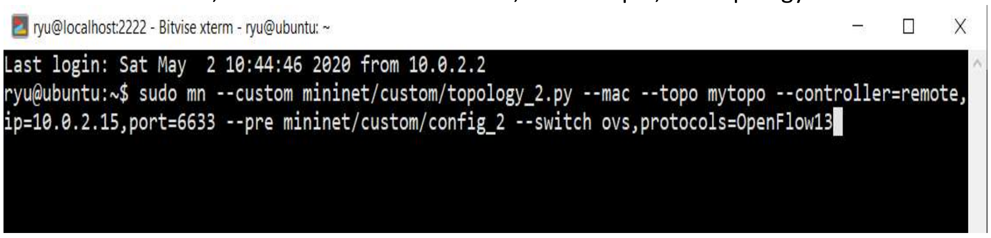
Elevation: Default

Put the parameters like in the figure above and log in. Then, you have to open a new terminal console clicking on the appropriate button. So now you see on your screen two windows of an Ubuntu terminal, like the figure below, and at the same time your virtual machine is running in background.



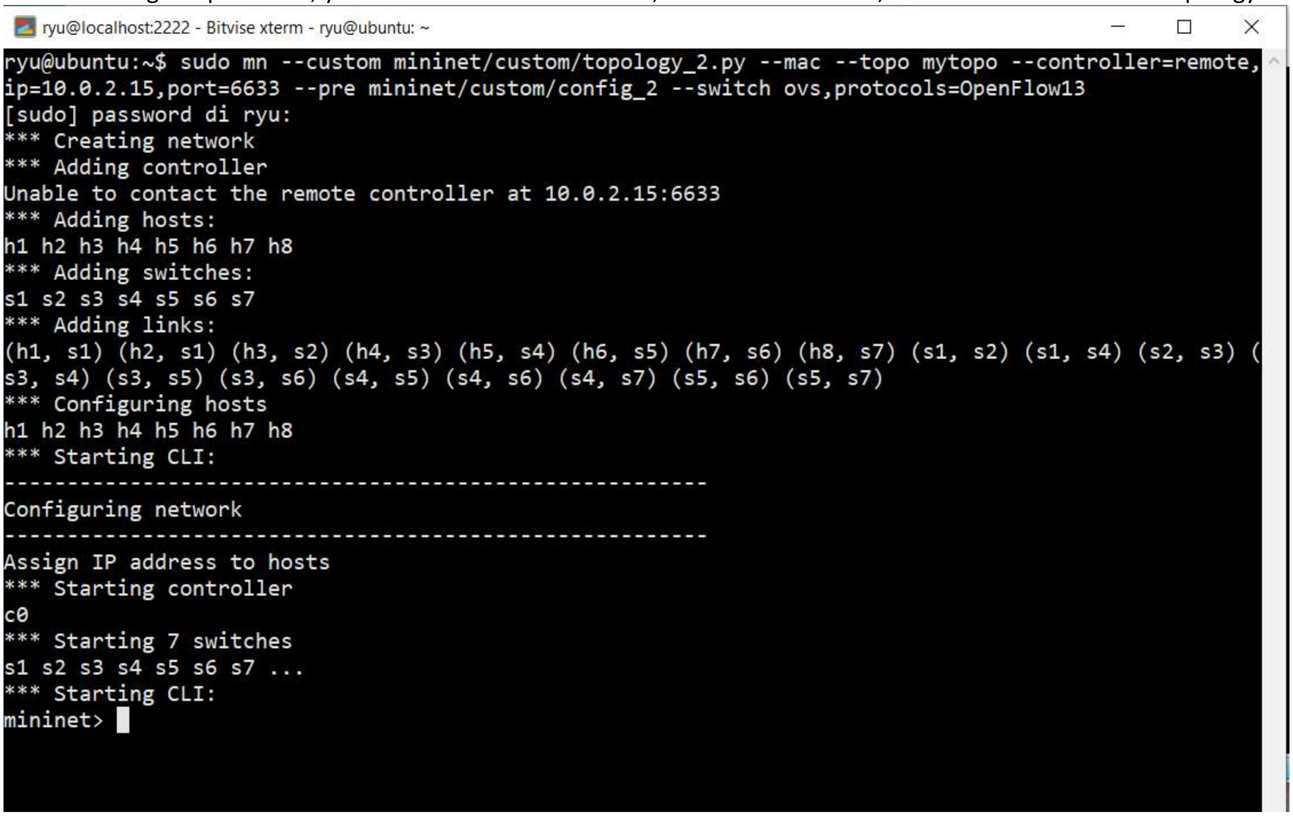
```
Last login: Fri May  1 15:38:49 2020
ryu@ubuntu:~$ 
```

In the first terminal, write this command to run, for example, the topology 2



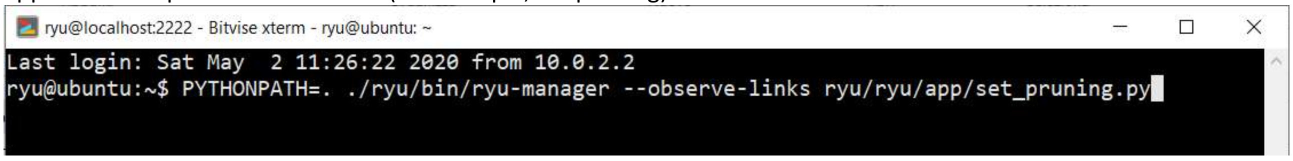
```
Last login: Sat May  2 10:44:46 2020 from 10.0.2.2
ryu@ubuntu:~$ sudo mn --custom mininet/custom/topology_2.py --mac --topo mytopo --controller=remote,
ip=10.0.2.15,port=6633 --pre mininet/custom/config_2 --switch ovs,protocols=OpenFlow13
```

After inserting the password, you should obtain this window, that shows all hosts, switches and links of the topology.



```
ryu@ubuntu:~$ sudo mn --custom mininet/custom/topology_2.py --mac --topo mytopo --controller=remote,
ip=10.0.2.15,port=6633 --pre mininet/custom/config_2 --switch ovs,protocols=OpenFlow13
[sudo] password di ryu:
*** Creating network
*** Adding controller
Unable to contact the remote controller at 10.0.2.15:6633
*** Adding hosts:
h1 h2 h3 h4 h5 h6 h7 h8
*** Adding switches:
s1 s2 s3 s4 s5 s6 s7
*** Adding links:
(h1, s1) (h2, s1) (h3, s2) (h4, s3) (h5, s4) (h6, s5) (h7, s6) (h8, s7) (s1, s2) (s1, s4) (s2, s3) (
s3, s4) (s3, s5) (s3, s6) (s4, s5) (s4, s6) (s4, s7) (s5, s6) (s5, s7)
*** Configuring hosts
h1 h2 h3 h4 h5 h6 h7 h8
*** Starting CLI:
-------------------------------------------------------
Configuring network
-------------------------------------------------------
Assign IP address to hosts
*** Starting controller
c0
*** Starting 7 switches
s1 s2 s3 s4 s5 s6 s7 ...
*** Starting CLI:
mininet>
```

At this point, the network has been configurated and initializated. So you can, on the other terminal, launch one of the application for packet classification (for example, set pruning).



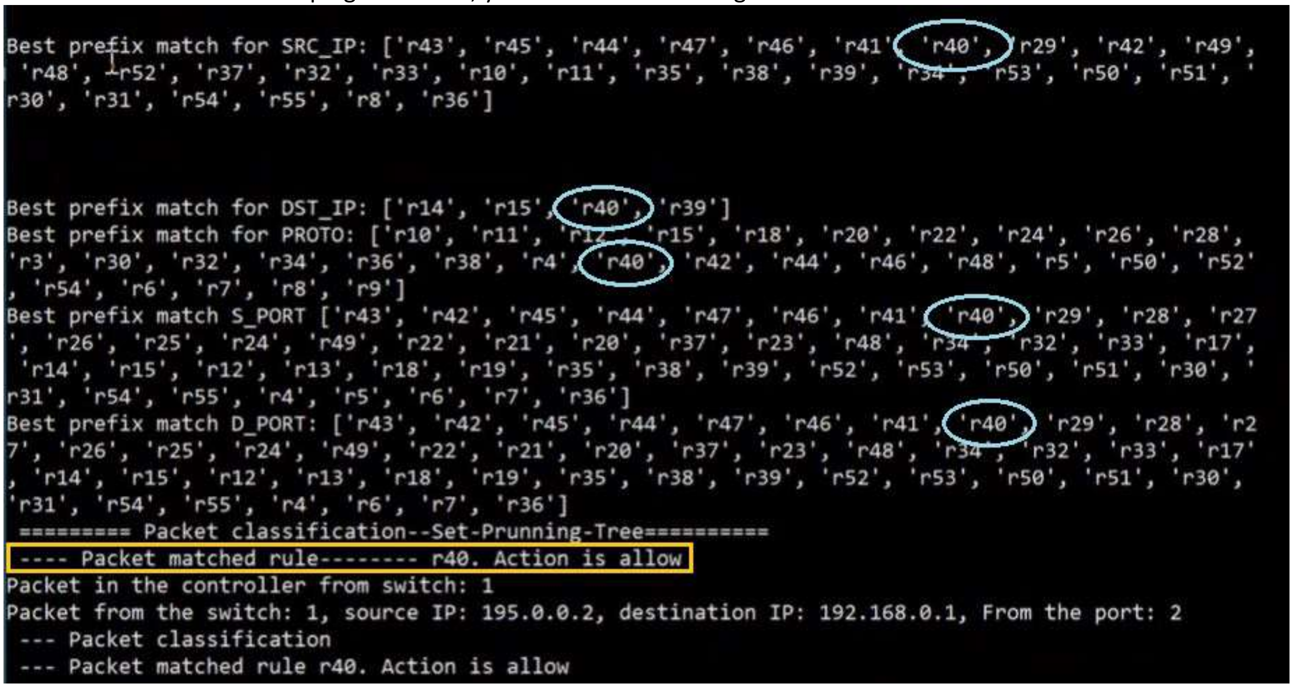
```
Last login: Sat May  2 11:26:22 2020 from 10.0.2.2
ryu@ubuntu:~$ PYTHONPATH=. ./ryu/bin/ryu-manager --observe-links ryu/ryu/app/set_pruning.py
```

Now, the application is running correctly.
Then, in first terminal (mininet) you should write commands to test the network and to obtain results about packet classification.

```
@mininet>
@mininet>
@mininet h1 ping h2
```

## 8 RESULTS

At the end of execution of a ping command, you obtain the following result:



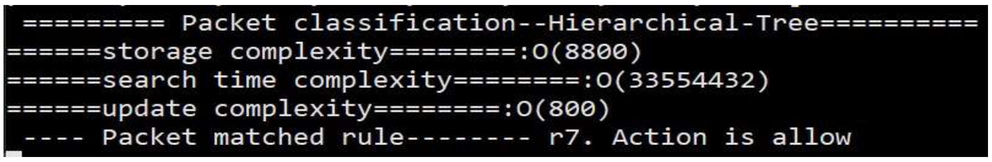
```
Best prefix match for SRC_IP: ['r43', 'r45', 'r44', 'r47', 'r46', 'r41', 'r40', 'r29', 'r42', 'r49',
 'r48', 'r52', 'r37', 'r32', 'r33', 'r10', 'r11', 'r35', 'r38', 'r39', 'r34', 'r53', 'r50', 'r51', '
r30', 'r31', 'r54', 'r55', 'r8', 'r36']



Best prefix match for DST_IP: ['r14', 'r15', 'r40', 'r39']
Best prefix match for PROTO: ['r10', 'r11', 'r12', 'r15', 'r18', 'r20', 'r22', 'r24', 'r26', 'r28',
'r3', 'r30', 'r32', 'r34', 'r36', 'r38', 'r4', 'r40', 'r42', 'r44', 'r46', 'r48', 'r5', 'r50', 'r52'
, 'r54', 'r6', 'r7', 'r8', 'r9']
Best prefix match S_PORT ['r43', 'r42', 'r45', 'r44', 'r47', 'r46', 'r41', 'r40', 'r29', 'r28', 'r27
', 'r26', 'r25', 'r24', 'r49', 'r22', 'r21', 'r20', 'r37', 'r23', 'r48', 'r34', 'r32', 'r33', 'r17',
 'r14', 'r15', 'r12', 'r13', 'r18', 'r19', 'r35', 'r38', 'r39', 'r52', 'r53', 'r50', 'r51', 'r30', '
r31', 'r54', 'r55', 'r4', 'r5', 'r6', 'r7', 'r36']
Best prefix match D_PORT: ['r43', 'r42', 'r45', 'r44', 'r47', 'r46', 'r41', 'r40', 'r29', 'r28', 'r2
7', 'r26', 'r25', 'r24', 'r49', 'r22', 'r21', 'r20', 'r37', 'r23', 'r48', 'r34', 'r32', 'r33', 'r17'
, 'r14', 'r15', 'r12', 'r13', 'r18', 'r19', 'r35', 'r38', 'r39', 'r52', 'r53', 'r50', 'r51', 'r30',
'r31', 'r54', 'r55', 'r4', 'r6', 'r7', 'r36']
 ========= Packet classification--Set-Prunning-Tree==========
 ---- Packet matched rule-------- r40. Action is allow
Packet in the controller from switch: 1
Packet from the switch: 1, source IP: 195.0.0.2, destination IP: 192.168.0.1, From the port: 2
 --- Packet classification
 --- Packet matched rule r40. Action is allow
```

You can see all the rules that match with each field of packet.
In this example, the best prefix found is the rule number 40, that appears in all of field matches. So the final result is the action corresponding to that rule, in this case “allow”.

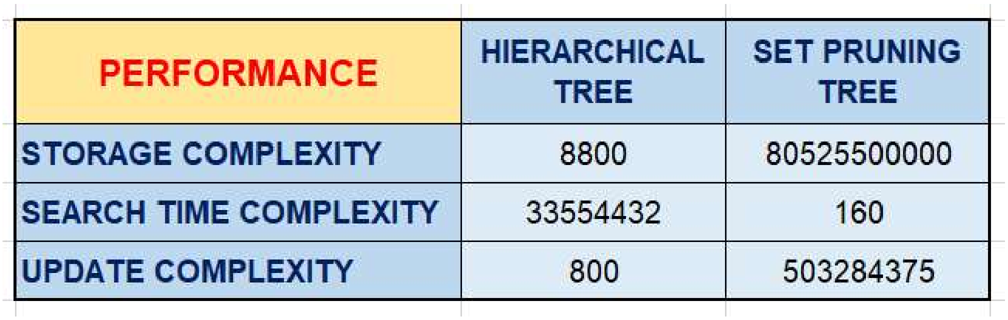
## 8.1 Performance

After printing the packet classification result, you can find the performance parametres. In particular, we analize for each algorithm:

- Storage complexity
- Search time complexity
- Update complexity



```
 ========= Packet classification--Hierarchical-Tree==========
======storage complexity========:O(8800)
======search time complexity========:O(33554432)
======update complexity========:O(800)
 ---- Packet matched rule-------- r7. Action is allow
```



| PERFORMANCE | HIERARCHICAL TREE | SET PRUNING TREE |
|---|---|---|
| STORAGE COMPLEXITY | 8800 | 80525500000 |
| SEARCH TIME COMPLEXITY | 33554432 | 160 |
| UPDATE COMPLEXITY | 800 | 503284375 |

In this table, there are collected the results of the analysis, that is also represented in the following graph (Note: on vertical axis, we use the logarithm scale).