به نام خدا

محمدحسین ارسلان 98243005

پروژه‌ی نهایی شبکه‌های کامپیوتری

توضیحی بر کلیت روند پیشرفت پروژه

در ابتدا به پیاده‌سازی منطق برنامه به زبان پایتون که شامل ساخت و نگهداری اطلاعات شبکه‌ای از روترهای به هم متصل و قابلیت ویرایش هزینه لینک‌ها و اجرای الگوریتم مسیریابی بردار فاصله (Distance Vector Routing) به صورت Distributed در هر روتر که در این پروژه روتر خود دارای یک کلاس مجزا است که در ادامه به طور کامل توضیح داده شده است. پس از پیاده‌سازی منطق پروژه و اطمینان از صحت آن به اضافه کردن ارتباط سوکت به پروژه پرداختم (پیش از آن برای تست به صورت دستی و بدون استفاده از اتصال UDP اطلاعات جدول‌ها را میان روترهای مجاور یک روتر به اشتراک می‌گذاشتم.) همچنین از دو ترد دائمی نیز استفاده کردم تا بحث مالتی‌تردینگ پروژه نیز تکمیل شود. در ادامه گزارش روند ما به این صورت است که در ابتدا به معرفی و توضیح مختصر کتابخانه‌ها و پکیج‌های به کار رفته در پروژه و انگیزه استفاده از هریک پرداخته، سپس منوی برنامه و قابلیت‌های پروژه را به طور کامل بررسی کرده و سپس برای معرفی توابع کمکی و کلاس طراحی شده مخصوص روتر یک مثال ساده را پیش می‌بریم و هریک از توابع موجود در پروژه را با هم بررسی می‌کنیم.

یک فهرست مختصر

1. معرفی پکیج‌ها
2. امکانات پروژه و منوی برنامه، توابع کمکی و معرفی کلاس و توابع روتر
3. پیشبرد یک مثال ساده به همراه نمایش وضعیت شبکه و خروجی گرفتن

کتابخانه‌ها و پکیج‌ها

```
import threading
import socket
import time
```

دو پکیج کلیدی این پروژه که Threading و Socket Programming را به کمک آن‌ها پیش می‌بریم و پکیج Time نیز برای انجام اعمال مرتبط با Timeout استفاده می‌شود.

```
import networkx as nx
import matplotlib.pyplot as plt
import math
```

از اصلی‌ترین پکیج‌هایی که در این پروژه برای جلوگیری از اشتباه در بحث حفظ اطلاعات شبکه روترها استفاده کرده‌ام پکیج Networkx می‌باشد که در درس شبیه‌سازی کامپیوتری در ترم جاری با آن آشنا شدم، می‌باشد. در این کتابخانه می‌توانیم یک گراف ساخته و تمام ساز و کارهای ممکمن با یک گراف را در آن داشته باشیم که در ادامه وقتی توابع پیاده‌سازی شده را بررسی می‌کنیم با قابلیت‌های این پکیج آشنا می‌شویم.

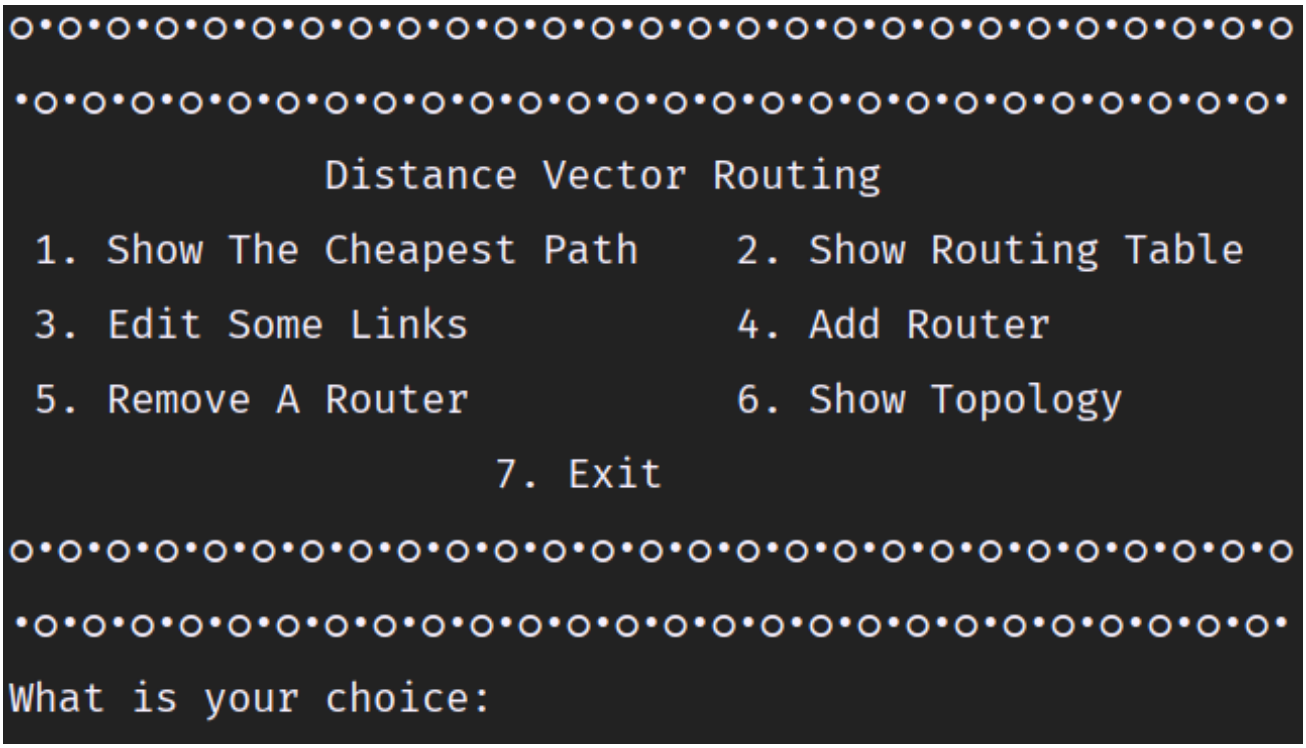
از پکیج matplotlib هم برای نمایش توپولوژی شبکه استفاده می‌کنیم، از کتابخانه math هم صرفا برای استفاده از عدد ∞ بهره بردیم.

```
import json
```

از آنجا که برای ذخیره جدول‌های مسیریابی از ساختمان داده Dictionary زبان پایتون استفاده کرده‌ام، همچنین باید در طول پروژه و اجرای برنامه ما این جدول را از طریق اتصال UDP میان روترها رد و بدل کنیم و از آنجا که نیاز داریم پس از encode و decode مجدد ساختمان داده Dictionary در دست داشته باشیم باید از یک پروتکل ارسال پیام بهره می‌بردم که بعد از جستجو با پکیج json آشنا شدم و پس از مطالعه documentهای مربوطه به کمک دو تابع از این کتابخانه که در ادامه با آن‌ها رو به رو می‌شویم توانستم بحث انتقال Dictionary از طرق Socket را حل کنم.

محیط برنامه و منوی اجرا

برای منوی برنامه ابتدا یک طرح گرافیکی در ذهن داشتم ولی به دلیل کمبود وقت از انجام آن صرف‌نظر کردم و با منوی ساده‌ای در کنسول مواجه می‌شویم، این منو که در یک تابع print_menu هر سری در یک حلقه موجود در تابع main چاپ می‌شود امکانات زیر را ارائه می‌دهد:



در ادامه تک به تک گزینه‌های این منو را بررسی می‌کنیم

1. **نمایش هزینه کمینه** (به کمک مسیریابی بردار فاصله و جدول مسیریابی live روترها) میان یک روتر مبدا و مقصد که پس از انتخاب این گزینه در ابتدا شماره روتر مبدا را از کاربر می‌گیریم که باید شماره روتر معتبری را به ما ارئه دهد و سپس شماره روتر مقصد را می‌گیریم و تابع cheapest_path را با دو مقدار گرفته شده فراخوانی می‌کنیم که پیاده‌سازی این تابع به‌صورت زیر می‌باشد.

```
def cheapest_path(src, dst):
    if(src.routing_table.get(str(dst.name)) != math.inf):
        print(f'Cheapest path between router number
{src.name} and router number {dst.name} cost is
{src.routing_table.get(str(dst.name))} ')
    else:
        print(f'There is no path from router number
{src.name} to router number {dst.name}')
```

بسته به اینکه چه زمانی ما این تابع را فراخوانی کنیم، جدول مسیریابی طرفین این مسیر می‌توانند متفاوت باشند مثلا در اولین لحظه ایجاد شبکه اگر لینک مستقیمی بین مبدا و مقصد وجود نداشته باشد هزینه این مسیر در طرفین برابر با بی‌نهایت است، پس ابتدا بررسی می‌کنیم که در جدول مسیریابی مبدا تا مقصد (و البته برعکس نیز بررسی شود موردی ندارد) هزینه بی‌نهایت ثبت نشده باشد، در این صورت پیغام **مسیری بین روتر مبدا و مقصد وجود ندارد** چاپ می‌شود و اگر چنین نبود، هزینه کمینه (موجود در جدول مسیریابی طرفین) چاپ می‌شود.

2. **نمایش جدول مسیریابی** یک روتر خاص، همانند گزینه اول ابتدا از کاربر یک شماره زوتر درخواست می‌کنیم و اعتبار آن را نیز بررسی می‌کنیم این اعمال به صورت زیر صورت می‌پذیرند:

```
for i in Routers:
    if i.name == int(r):
        R = i
        break
show_routing_table(R)
```

در یک لیست که به‌صورت global تعریف شده است و حاوی اطلاعات روترهای موجود در شبکه است پیمایش می‌کنیم و روتری را در متغیر R ذخیره می‌کنیم که نامش (شماره‌اش) با نام روتری کاربر وارد کرده‌است یکی باشد. سپس تابع show_routing_table را با این روتر فرا می‌خوانیم، این تابع نیز به صورت زیر است:

```
def show_routing_table(R):
    Routers.sort(key=lambda x: x.name, reverse=False)
    print(f'Latest routing table of router number {R.name}')
    for i in Routers:
        if (R.routing_table.get(str(i.name)) == math.inf):
            print(f'{i.name} : ∞')
        else:
            print(f'{i.name} :
{R.routing_table.get(str(i.name))}')
```

ابتدا چون ترتیب ورود روترها به شبکه اصلی را لزوما رعایت نکرده‌ایم به صورت صعودی، برای زیبایی و ترتیب نمایش درست جدول یک مرتب‌سازی صعودی روی روترها انجام می‌دهیم و سپس به ازای هر روتر بررسی می‌کنیم در چه فاصله‌ای از روتر R که پارامتر ورودی این تابع است قرار دارد و سپس در کنار نام روتر مقصد، هزینه موجود در جدول مسیریابی روتر R تا ن رورتر مقصد را چاپ می‌کنیم، اگر این روتر R فعلا راهی به روتر مقصد پیدا نکرده باشد، ∞ چاپ می‌شود و در غیر اینصورت اطلاعات موجود در جدول را عینا چاپ می‌کنیم (درصورتی هم صفر است که فقط هزینه یک روتر تا خودش مدنظر باشد)

مورد 3 را بعد از مورد 4 توضیح می‌دهیم چون یکسری از توابع کلاس روتر در طول این تابع استفاده شده اند

4. **افزودن روتر** به شبکه، این مورد انتخاب شود مستقیما به تابع add_router که یکی از اساسی‌ترین توابع این پروژه است می‌رویم. ابتدا شماره روتر مدنظر کاربر را که می‌خواهد به شبکه اضافه کند می‌گیریم، این شماره اولا نباید صفر باشد و ثانیا باید تا به حال به شبکه وارد نشده باشد، اگر هم قبلا وارد شده باشد و پیش از الان یکبار remove شده باشد هم مشکلی ندارد و می‌تواند وارد کند کاربر. سپس از کاربر می‌خواهیم لینک‌های این روتر جدید را به ما بدهد، این کار با وارد کردن یک شماره روتر موجود در شبکه و سپس درصورت تایید به وارد کردن هزینه این لینک می‌پردازد، هزینه هر لینک باید مقداری صحیح، مثبت و غیرصفر باشد. سپس با تابع add_edge از پکیج networkx به گراف G که یک متغیر global است و معادل است با شبکه روترهای ما این یال را اضافه می‌کنیم، هزینه(وزن) این یال را نیز به کمک همین تابع می‌توانیم در گراف (شبکه) وارد کنیم.

```
G.add_edge(int(name), int(adj), weight= int(weight))
```

سپس اینکار (ورود لینک) تا زمانی کاربر عدد 0 را وارد کند ادامه می‌دهیم، بعد به ساخت لیست همسایه‌های این روتر جدید می‌پردازیم و حال اصلی ترین بخش این تابع فرا می‌رسد، ساخت یک شی از این روتر و فعال کردن تردهای موردنیاز آن، حال یک پرش به بالاتر داریم و کلاس روتر را بررسی می‌کنیم و مجدد به این تابع برمی‌گردیم.

```
port_num = 60000
class Router():
    def __init__(self, name, adj):
        global port_num
        self.name = name
        self.adj = adj
        self.direct_link = [False] * 100
        self.port = port_num
        self.send_port = port_num + 1
        self.bellman_port = port_num + 2
        port_num += 3
        self.routing_table = dict()
        self.calculate_routing_table()
```

همانطور که در قطعه کد بالا می‌بینید کلاس روتر ما در constructor خود نام روتر و لیستی از همسایه‌های روتر را می‌گیریم که البته داشتن این فیلد کمک زیادی نمی‌کند چرا که در گراف لیست همسایه‌ها را داریم برای هر گره که یک روتر است. سایر فیلدها مقدار پیش‌فرضی می‌گیرند. یک لیست بولین داریم که Direct_link نام دارد و هنگامی در یک اندیس true باشد، یعنی در جدول مسیریابی روتر فعلی، این روتر به روتری با شماره برابر با اندیس به صورت مستقیم مسیر بهینه دارد یا به واسطه یک گره دیگر (نتیجه اجرای مسیریابی بردار فاصله)، این آرایه را به طور پیش فرض 100 خانه‌ای درنظر می‌گیریم و ابتدا پیش فرض همه خانه‌ها Falseاند. این گراف در بخش حذف یک روتر از شبکه به ما کمک می‌کند. حال سه عدد پورت به هر روتر اختصاص می‌دهیم، یکی برای دریافت از سایر روترها، یکی برای ارسال به‌دلیل timeout و یکی هم برای ارسال بعد از اجرای الگوریتم مسیریابی می‌باشد. هر بار که یک روتر ساخته می‌شود این شماره پورت‌ها از یک مقدار global که اولیه آن 60000 است به ازای هر پورت یکی زیاد می‌شود، یعنی اولین روتر سه پورت با شماره های 60000 – 60001 – 60002 دارد و روتر بعدی 60003 – 60004 – 60005 را صاحب می‌شود. سپس جدول مسیریابی این روتر نیز به کمک تابع calculate_routing_table ساخته می‌شود. که تابع مذکور به شرح زیر است:

```
def calculate_routing_table(self):
    for node in G.nodes:
        if (G.has_edge(self.name, node)):
            self.routing_table[str(node)] =
G.get_edge_data(self.name, node).get('weight')
            self.direct_link[node] = True
        else:
            weight = math.inf
            if(node == self.name):
                weight = 0
            self.routing_table[str(node)] = weight
```

این تابع به بررسی همه روترها می‌پردازد و فقط اگر یال مستقیمی بین روتر مدنظر (درحال ساخت) با روتر دیگری وجود داشته باشد مقدار آن لینک را در جدول ثبت می‌کنیم و در غیر اینصورت یا فاصله‌ش تا خودش را می‌خواهیم که صفر است و یا لینکی نداریم که بی‌نهایت را قرار می‌دهیم. همچنین اگر یال مستقیم داشتیم به یک روتر، اندیس متناظر آن را True می‌کنیم در لیست direct_link.

حال مجددا به نقطه قبلی یعنی تابع add_router برمی‌گردیم. در ادامه پس از ساخت این روتر سراغ همه روترها رفته و این روتر جدید را به جدول مسیریابی همه آن‌ها اضافه می‌کنیم. ولی سبب اجرای مسیریابی بردار فاصله در آن‌ها نمی‌شویم، این عمل (اجبار کردن روترها به اجرای الگوریتم فقط با ارسال بسته UDP صورت می‌گیرد به‌جز یک‌جا که در ادامه به آن می‌پردازیم.)

```
for r in Routers:
    if (r == router):
        continue
    if(G.has_edge(r.name, router.name)):
        r.routing_table[str(router.name)] =
G.get_edge_data(r.name, router.name).get('weight')
    else:
        r.routing_table[str(router.name)] = math.inf
```

سپس دو ترد دائمی که یکی مسئول ارسال درصورت timeout است به کمک پورت send_port و دیگری مسئول شنود و دریافت اطلاعات از طریق پورت port است را فعال می‌کنیم و به توابع موجود در کلاس روتر تخصیص می‌دهیم.

```
send_thread = threading.Thread(target=
router.send_to_adj).start()
rcv_thread = threading.Thread(target=
router.recieve_from_adj).start()
```

توابعی که این تردها دریافت می‌کنند در کلاس روتر تعریف شده اند و به صورت زیر هستند:

```
def send_to_adj(self):
    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    s.bind(('localhost', self.send_port))
    while True:
        data = json.dumps(self.routing_table)
        for n in Routers:
            if (G.has_edge(n.name, self.name)):
                s.sendto(data.encode('utf-8'), ('localhost',
n.port))
        time.sleep(10) #timeout
```

داده‌ای که هر 10 ثانیه (زمان timeout) ارسال می‌کنیم یک جدول مسیریابی است که به کمک پکیج json این دیکشنری به یک فایل بایت دیتا تبدیل شده و بعد با encode ارسال و سپس decode و تابعی دیگر از پکیج مذکور این دیکشنری به درستی و صحت در مقصد بازیابی می‌شود.

```
def recieve_from_adj(self):
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.bind(('localhost', self.port))
while True:
    data, address = s.recvfrom(2048)
    recieved_table = json.loads(data.decode('utf-8'))
    self.bellman_ford(recieved_table, address[1])
```

این تابع مسئول شنود هر روتر روی پورتی با شماره port است، که همیشه داده‌ها را می‌گیرد و علاوه بر دیتا که حاوی اطلاعت جدول مسیریابی ارسال‌کننده است، پورتی که از سریق آن فرستنده ارسال کرده دریافت می‌کنیم و الگوریتم مسیریابی را با پارامترهای جدول دریافتی و شماره پورت فرستنده فرا می‌خوانیم.

در ادامه به تابع الگوریتم مسیریابی بردار فاصله که نام آن bellman_ford است می‌پردازیم که در کلاس روتر تعریف شده است، ابتدا به دنبال این می‌گردیم که کدام روتر به ما جدول فرستاده، این کار با پیمایش روی روترها و به کمک پارامتر دوم که پورت فرستنده است صورت می‌گیرد.

حال می‌دانیم که روتر R به ما جدول خودش را فرستاده است. حال مجدد پیمایشی روی روترها داریم، هربار با کمک الگوریتم بلمن فورد مقایسه هزینه خودمان تا فرستنده و فاصله فرستنده تا مقصد را با فاصله فعلی خودمان با مقصد بررسی می‌کنیم، اگر فاصله‌ای که فعلا در جدول داریم کمتر بود که تغییری ایجاد نمیکنیم ولی در غیر اینصورت بروزرسانی را انجام می‌دهیم و چون مسیر جدید ما تا مقصد دیگر مستقیم نیست و واسطه (روتر فرستنده جدول) دارد، در لیست Direct_link هم تغییر داده داریم. و همچنین چون جدول خودمان هم تغییر کرده، یک flag به نام change داریم که اگر True شود باید یک سوکت باز کرده و از طریق پورت bellman_port به ارسال جدول خود برای همسایه‌ها بپردازیم.

کد این الگوریتم مسیریابی به صورت زیر می‌باشد:

```
def bellman_ford(self, updated_table, port_number):
    change = False
    Routers.sort(key=lambda x: x.name, reverse=False)
    R = Routers[0]
    for r in Routers:

        if((r.send_port == int(port_number)) or
(r.bellman_port == int(port_number))):
            R = r
    for r in Routers:
        try:
            if((self.routing_table.get(str(R.name)) +
updated_table.get(str(r.name))) <
(self.routing_table.get(str(r.name)))):
                change = True
                self.routing_table[str(r.name)] =
self.routing_table.get(str(R.name)) +
updated_table.get(str(r.name))
                if(updated_table.get(str(r.name)) != 0):
                    self.direct_link[r.name] = False
        except:
            pass
    if(change):
        s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        s.bind(('localhost', self.bellman_port))
        data = json.dumps(self.routing_table)
        for n in Routers:
            if(G.has_edge(n.name, self.name)):
                s.sendto(data.encode('utf-8'), ('localhost',
n.port))
        change = False
```

پس از اینکه یک روتر را به همراه لینک‌های اطرافش به شبکه اضافه کردیم یک نما از شبکه را به کمک فراخوانی تابع show_topology به نمایش می‌گذاریم که به‌صورت زیر پیاده سازی شده است، یک layout به شبکه اختصاص می‌دهیم که spectral نظم و زیبایی بالاتری دارد.

```
def show_topology():
    layout = nx.spectral_layout(G)
    # layout = nx.random_layout(G)
    nx.draw(G, layout, with_labels=True, font_weight='bold')
    edge_weight = nx.get_edge_attributes(G, 'weight')
    nx.draw_networkx_edge_labels(G, layout,
edge_labels=edge_weight)
    plt.show()
```

6. مورد شمار 6 در منوی اصلی هم همین **تابع نمایش توپولوژی** است
3. **ویرایش هزینه یک لینک** مثل گزینه‌های قبل دو روتر معتبر دریافت می‌کنیم و وارد تابع edit_link می‌شویم، هزینه مدنظر که باید معتبر باشد را دریافت می‌کنیم و لینک را بروزرسانی می‌کنیم، اگر این هزینه معادل بود با مقدار موجود در جدول مسیریابی طرفین برای هم یا حتی کمتر بود، ما این مقدار جدید را در جدول طرفین قرار می‌دهیم و همچنین یکبار بلمن‌فورد را برای طرفین لینک (با ورودی جدول خودشان نه دیگری) صدا می‌زنیم. در direct_link هم متناظرا مقدار True قرار می‌دهیم.

```
def edit_link(src, dst):
    if(G.has_edge(src.name, dst.name)):
        weight = input(f'Enter the new link\'s cost: ')
        while (int(weight) < 1 or weight[0] == '-'):
            weight = input(f'Not a valid cost, please re-
enter the new link\'s cost: ')
        src_update = (G.get_edge_data(src.name,
dst.name).get('weight') ==
src.routing_table.get(str(dst.name)))
        dst_update = (G.get_edge_data(src.name,
dst.name).get('weight') ==
dst.routing_table.get(str(src.name)))
        G.add_edge(src.name, dst.name, weight = int(weight))
        if (src_update):
            src.direct_link[dst.name] = True
```

```
            src.routing_table[str(dst.name)] = int(weight)
        if (dst_update):
            dst.direct_link[src.name] = True
            dst.routing_table[str(src.name)] = int(weight)
        src_update = (G.get_edge_data(src.name,
dst.name).get('weight') <
src.routing_table.get(str(dst.name)))
        dst_update = (G.get_edge_data(src.name,
dst.name).get('weight') <
dst.routing_table.get(str(src.name)))
        if(src_update):
            src.direct_link[dst.name] = True
            src.routing_table[str(dst.name)] = int(weight)
        if(dst_update):
            dst.direct_link[src.name] = True
            dst.routing_table[str(src.name)] = int(weight)
        src.bellman_ford(src.routing_table, src.send_port)
        dst.bellman_ford(dst.routing_table, dst.send_port)
    else:
        print(f'There is no link to edit between router
number {src.name} and router number {dst.name}')
```

5. **حذف یک روتر و لینک‌های متصل به آن** این عمل نیز با گرفتن شماره یک روتر معتبر و فراخوانی تابع remove_router با روتر متناظر با آن شماره صورت می‌پذیرد.

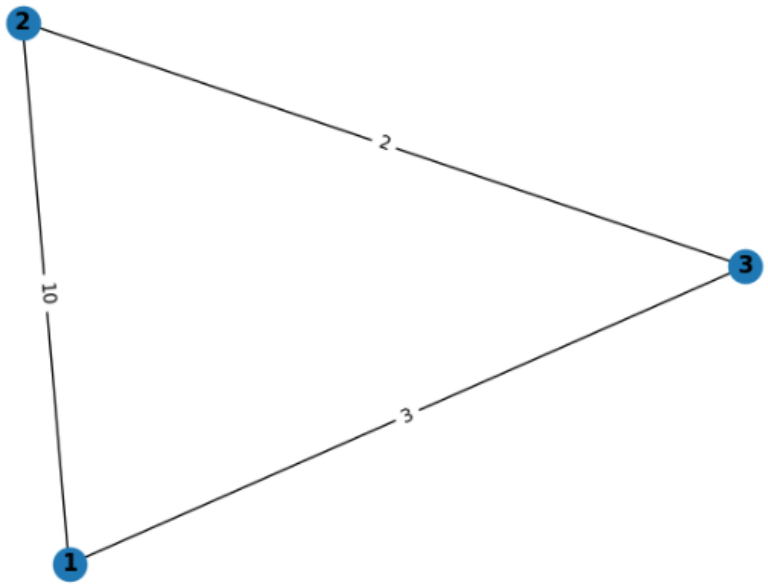
```
def remove_router(R):
    for r in Routers:
        if(r.direct_link[R.name]):
            r.routing_table[str(R.name)] = math.inf
            r.direct_link[R.name] = False
    G.remove_node(int(R.name))
    to_be_removed = Router
    for r in Routers:
```

```
        if (r.name == R.name):
            to_be_removed = r
            break
    Routers.remove(to_be_removed)
    del R
```

برای اینکه این روتر بدون ایجاد مشکل حذف شود، چک می‌کنیم که اگر میان او و دیگران لینک مستقیم داریم یعنی یک روتر تنها راهش به روتر در حال حذف یال مستقیم است، این هزینه به بینهایت تغییر یابد ولی برای حل مشکل اینکه این روتر در مسیر یک مسیر بهینه قرار گرفته باشد مثلا بین روتر 1 و 3 یک مسیر بهینه هست که از 2 می‌گذرد ولی اگر 2 حذف شود ما در لحظه حذف اطلاعی نمی‌توانیم به 1 و 3 بدهیم که مسیری که در جدولشان به یکدیگر دارند دیگر فاقد اعتبار است، این اطلاع‌رسانی بعد از فرارسیدن timeout و راه دیگری نیز قابل حل نبود و همین مشکل count to infinity است که از ایرادات این الگوریتم بود.

اجرای یک مثال ساده و دریافت خروجی‌ها

یک شبکه با حضور 3 روتر 1 و 2 و 3 می‌سازیم



ابتدا روتر 1 وارد شبکه شد و ایزوله بود سپس روتر 2 وارد شد و یک لینک با هزینه 10 به یک داشت. پس در این لحظه در جدول مسیریابی این دو روتر، یک مسیر بهینه به یکدیگر هزینه 10 دارد.

```
What is your choice: 2
Enter the router number, to return to menu enter 0: 1
Latest routing table of router number 1
1 : 0
2 : 10
```

```
What is your choice: 2
Enter the router number, to return to menu enter 0: 2
Latest routing table of router number 2
1 : 10
2 : 0
```

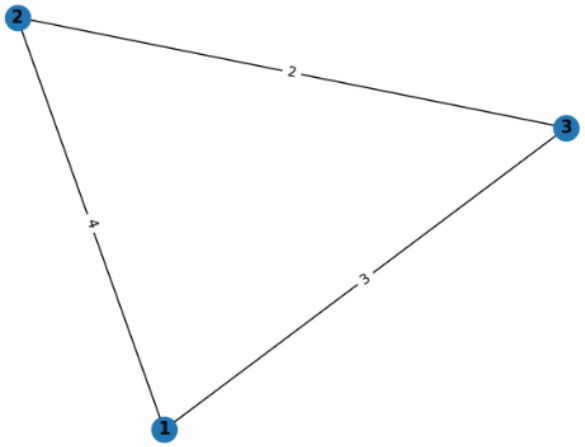
سپس روتر 3 به شبکه اضافه می‌شود که فاصله با روتر 2 لینکی با هزینه 2 و با روتر 1 لینکی با هزینه 3 دارد، حال با ارسال پیام به روترهای همسایه بعد از یک timeout الگوریتم مسیریابی در روتر 1 و 2 یک مسیر بهینه با هزینه 5 به یکدیگر به واسطه روتر 3 پیدا می‌کنند.

```
What is your choice: 2
Enter the router number, to return to menu enter 0: 1
Latest routing table of router number 1
1 : 0
2 : 5
3 : 3
```

```
What is your choice: 2
Enter the router number, to return to menu enter 0: 2
Latest routing table of router number 2
1 : 5
2 : 0
3 : 2
```

```
What is your choice: 2
Enter the router number, to return to menu enter 0: 3
Latest routing table of router number 3
1 : 3
2 : 2
3 : 0
```

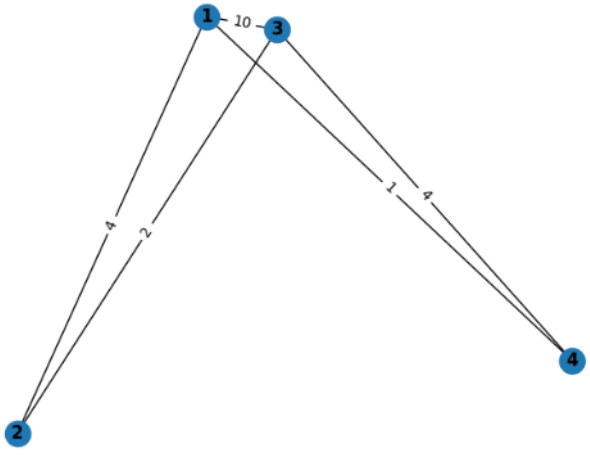
حال لینک بین 1 و 2 را ویرایش کرده و به 4 هزینه آن را کاهش می‌دهیم.




```
What is your choice: 2
Enter the router number, to return to menu enter 0: 1
Latest routing table of router number 1
1 : 0
2 : 4
3 : 3
```

```
What is your choice: 2
Enter the router number, to return to menu enter 0: 2
Latest routing table of router number 2
1 : 4
2 : 0
3 : 2
```

حال یک روتر شماره 4 وارد میکنیم که به روتر شماره 3 با لینک هزینه 4 و به روتر 1 با هزینه 1 وصل است. و سپس هزینه لینک 1 و 3 را به 10 افزایش می‌دهیم، حالا مسیر گذرنده از روتر 4 برای ارتباط روتر 1 با 3 بهینه تر است.



```
What is your choice: 2
Enter the router number, to return to menu enter 0: 1
Latest routing table of router number 1
1 : 0
2 : 4
3 : 3
4 : 1
```

```
What is your choice: 3
Enter the source router number, to return to menu enter 0: 1
Enter the destination router number: 3
Enter the new link's cost: 10
```

```
What is your choice: 2
Enter the router number, to return to menu enter 0: 1
Latest routing table of router number 1
1 : 0
2 : 4
3 : 5
4 : 1
```