به نام خدا

گروه ۱۹: دانیال علی عظیمی ۹۷۲۴۳۰۸۳
حسنا بشیریان ۹۷۲۴۳۱۰۲

تکلیف کامپیوتری دوم شبکه

ابتدا به پیاده سازی کلاس Router شروع میکنیم :

```
class Router():
    def __init__(self, neighboures, node_num) -> None:
        self.node_num = node_num
        # neightboures : array of tuples in the form of (node number , node distance)
        self.neighboures = neighboures
        # init empty table
        self.table = [[MAX_INT, MAX_INT, MAX_INT, MAX_INT]
                      for _ in list(range(len(neighboures)+1))]
        self.updated = False
        self.table[node_num][node_num] = 0
        # pointer to access other routers for event function
        self.routers = []
        # set routers costs
        for i in neighboures:
            self.table[node_num][i[0]] = i[1]
```

کلاس Router دارای فیلد های:

- Node_num : شماره router فعلی را نگاه میدارد
- Neighboures : آرایه ای از جنس tuple که دارای هر همسایه و هزینه رسیدن به آن است
- Table : جدول فواصل که به صورت یک ماتریس 4*4 نگهداری میشود و در شروع مقدار MAX_INT به هر کدام از خانه اش داده شده
- Routers : اشاره گر به لیستی از router ها را برایمان نگه میدارد.

در خط آخر constructor مقدار های فاصله در جدول فواصل قرار داده میشود.

```python
from Router import Router
MAX_INT = 2**31 - 1

NEIGHBORS = {(1, 1), (2, 3), (3, 7)}  # (node , cost)


def rtinit0():
    router = Router(NEIGHBORS, 0)
    print(f"table{router.node_num}", "initialized")
    router.print_table()
    return router
```

ما یک لیستی از neighbors که در آن هر عنصر شماره و هزینه رسیدن به آن router را از router فعلی برای مثال router0 نگهداری میکند.
این مقدار به عنوان ورودی به constructor کلاس Router داده میشود و ورودی بعدی هم شماره node فعلی است.
فایل های دیگر هم به همین صورت برای router1 و router2 و router3 پیاده شده اند.

برای بررسی اولیه جدول فواصل یک تابع print_table در داخل کلاس router پیاده سازی شده است.

```
routers = []

if __name__ == "__main__":

    routers.append(rtinit0())
    routers.append(rtinit1())
    routers.append(rtinit2())
    routers.append(rtinit3())
    for r in routers:
        r.routers = routers
    r0 = routers[0]
    r0.toLayer2()
    r0.print_table()
```

بدنه اصلی کد که در فایل main قرار دارد در شروع توابع init را برای هر یک از router ها فراخوانی می کند و اشاره گر به لیست routers را برای هر کدام set میکند. پس از آن با فراخوانی tolayer2 برای router 0 عملیات routing آغاز میشود.
و پس اجرا مقدار :

```
event received 0
event received 3
event received 3
event received 1
[0, 1, 2, 4]
[1, 0, 1, 3]
[2, 1, 0, 2]
[4, 3, 2, 0]
```

در تمامی جدول فواصل قرار گرفته است.

```
# packet is a tuple of src and distance_table
def toLayer2(self) -> None:
    packet = (self.node_num, self.table)
    for n in self.neighboures:
        if(n[1] < MAX_INT):
            self.routers[n[0]].receive_event(packet)
```

تابع toLayer2 در یک tuple شماره router فعلی و جدول فواصل آن را نگهداری میکند و به تمامی همسایه هایش ارسال میکند.

این ارسال به صورت فراخوانی یک تابع از router های دیگه است که شبیه سازی حالت دریافت event در محیط sequential در نظر گرفته شده مانند یک callback.

```
def receive_event(self, packet) -> None:
    print("event received", self.node_num)
    self.rtUpdate(packet)
```

کاری که تابع recive_events انجام میدهید این است که تابع rtUpdate را با پکتی که دریافت کرده است فراخوانی میکند.

```python
def rtUpdate(self, packet) -> None:
    self.changed = False
    received_packet = packet[1]
    for i in range(len(self.table)):
        for j in range(len(self.table[i])):
            if self.table[i][j] > received_packet[i][j]:
                self.table[i][j] = received_packet[i][j]
                self.changed = True
    for i in range(len(self.table)):
        dist = Router.get_min_dist(
            self.node_num, i, set(), self.table, self.neighboures)
        if(self.table[self.node_num][i] > dist):
            self.table[self.node_num][i] = dist
            self.changed = True
    # if changed send to all neighboures
    if self.changed == True:
        self.toLayer2()
```

تابع در اول جدول فواصل را از پکت دریافتی استخراج میکند و سپس distance_table در router فعلی را update می کند (اگر مقدارش کمتر باشد) و سپس با استفاده از الگوریتم Belmanford به ازای هر node از router فعلی به پیدا کردن کوتاه ترین مسیر میپردازد و پس از آن اگر تغییری در table رخ داده باشد این تغییر را به تمامی router های دیگر با فراخوانی toLayer2 فراخوانی می کنیم.
دقت شود بخاطر ساختار OOP که برای router در نظر گرفته شده است نیازی به پیاده سازی این تابع به ازای هر router نبود.

```python
# compute minimum distance from src to dest , hold seen_nodes to control node expansion

def get_min_dist(src, dest, seen_nodes, table, neighboures) -> int:
    if src == dest:
        return 0
    min_dist = MAX_INT
    for n in neighboures:
        if n[0] in seen_nodes:
            continue
        seen_nodes.add(src)
        new_neighbours = Router.get_neighbours(table, n[0])
        cost = table[src][n[0]] + Router.get_min_dist(
            n[0], dest, seen_nodes, table, new_neighbours)
        seen_nodes.remove(src)
        if cost < min_dist:
            min_dist = cost
    return min_dist
```

تابع get_min_distance کوتاه ترین فاصله را از router فعلی تا router های دیگر را بدست میاورد به صورتی که درخت را پیمایش میکند و به صورت بازگشتی خود را برای router ها فراخوانی می کند و مجموع هزینه را برمیگرداند. برای پیمایش ساده تر گراف تابع get_neighboures پیاده سازی شد.

```python
# get neighbours in form of array
def get_neighbours(table, src) -> list:
    n = []
    for i in range(len(table[src])):
        if table[src][i] == MAX_INT or i == src:
            continue
        neighbour = (i, table[src][i])
        n.append(neighbour)
    return n
```

که با نگاه کردن به یک سطر از جدول فواصل برای router فعلی همسایه ها آن را دریافت میکند.