IE260 Final Term Homework

20150359 Chiyoon Park

1. Code Description

(1) Stack.py / Class Stack

```
class Stack:

    def __init__(self):
        self.length = 0
        self.stack = []

    def push(self, data):
        self.stack.append(data)
        self.length += 1

    def pop(self):
        if self.length != 0:
            data = self.stack.pop()
            self.length -= 1
            return data

    def isEmpty(self):
        if self.length == 0:
            return True
        else:
            return False
```

Stack.py 파일에 구현한 stack 자료구조입니다. 파이썬에 내장된 List를 사용하여 구현하였습니다. 메소드는 push( ), pop( ), isEmpty( ) 세 개가 있습니다. Push는 데이터를 stack에 삽입하는 용도이며, pop은 스택의 맨 위 데이터를 제거하는 메소드입니다. isEmpty는 스택이 비었는지 아닌지를 체크합니다.

Stack 메소드는 MultiAgentNetworkCoverage.py 파일의 finRoute( ) 메소드에서 사용됩니다.

(2) MultiAgentNetworkCoverage.py / Class MultiAgentNetworkCoverage

```
import sys
import Dijkstra
import Graph
import Stack
from Evaluator import Evaluator
import time
import csv

class MultiAgentNetworkCoverage:

    def __init__(self,objGraph):
        self.objGraph = objGraph
        self.component = self.objGraph.findComponent()
```

MultiAgentNetworkCoverage.py 의 경우 제공된 Dijkstra.py, Graph.py, Evaluator.py 세 파이썬 파일에 더하여 직접 구현한 Stack.py를 추가로 import 하였습니다. 클래스 MultiAgentNetworkCoverage의 경우 __init__ 메소드를 통해서 Graph 하나를 parameter로 입력받습니다. 입력받은 그래프는 self.objGraph 변수에 저장되며, 이 그래프의 component를 그래프의 findComponent( ) 메소드를 통해 얻어서 self.component에 저장합니다.

Self.objGraph에 저장된 그래프의 경우 MultiAgentNetworkCoverage 클래스의 메소드들에서 사용되며, component도 메소드에서 사용됩니다.

(3) Class MultiAgentNetworkCoverage / Method calculateBetweenness

```
def calculateBetweenness(self):
    vertexes = self.objGraph.vertexes
    numVertexes = len(self.objGraph.vertexes)
    betweenness = {}

    for i in range(numVertexes):
        betweenness[vertexes[i]] = {}
        for j in range(len(vertexes)):
            betweenness[vertexes[i]][vertexes[j]] = 0.0

    for i in range(numVertexes):
        dist, path, routes = Dijkstra.performAllDestinationDijkstra(self.objGraph, vertexes[i])
        for j in range(numVertexes):
            if i == j:
                continue
            if routes[vertexes[j]] != None:
                for k in range(len(routes[vertexes[j]]) - 1):
                    srcEdge = routes[vertexes[j]][k]
                    dstEdge = routes[vertexes[j]][k + 1]
                    betweenness[srcEdge][dstEdge] = betweenness[srcEdge][dstEdge] + 1
                    betweenness[dstEdge][srcEdge] = betweenness[dstEdge][srcEdge] + 1
    return betweenness
```

MultiAgentNetworkCoverage.py의 메소드 calculateBetweenness는 그래프의 vertex 사이의 betweenness를 계산하는 메소드입니다. Self.objGraph에 저장된 그래프가 분석하려는 그래프입니다. betweenness라는 dictionary를 선언하여 여기에 계산한 값을 저장합니다. For 문을 통해서 betweenness를 전부 0으로 초기화한 후, for문을 통해서 graph 내부의 모든 vertex에 대해서 performAllDestinationDihkstra( ) 메소드를 실행합니다. 여기서 얻어진 결과 routes를 통해서 두 지점 src와 dst 사이의 betweenness를 계산합니다. 이후 얻은 결과값을 반환합니다.

이 메소드는 MultiAgentNetworkCoverage.py의 newmanClustering( ) 메소드에서 사용됩니다.

(3) Class MultiAgentNetworkCoverage / Method newmanClustering

```
def newmanClustering(self, agent):
    vertexes = self.objGraph.vertexes
    numVertexes = len(self.objGraph.vertexes)
    if agent == 1:
        components = self.objGraph.findComponent()
        self.component = components
    else:
        while True:
            betweenness = self.calculateBetweenness()
            max = -9999
            idx1, idx2 = None, None

            for i in range(numVertexes):
                for j in range(numVertexes):
                    if betweenness[vertexes[i]][vertexes[j]] > max:
                        max = betweenness[vertexes[i]][vertexes[j]]
                        idx1, idx2 = i, j

            self.objGraph.removeEdge(vertexes[idx1], vertexes[idx2])
            self.objGraph.removeEdge(vertexes[idx2], vertexes[idx1])
            components = self.objGraph.findComponent()
            print("Clustering...")
            if len(components) == agent:
                break
            if len(components) == len(self.objGraph.vertexes):
                print("Num of agent is bigger than Num of Vertexes")
                break
            print("Num of component : ", len(components))
        self.component = self.objGraph.findComponent()
```

MultiAgentNetworkCoverage.py의 메소드 newmanClustering은 Girvan-Newman 알고리즘을 통해서 네트워크를 클러스터로 만드는 메소드입니다. Parameter로 int인 agent를 받아서 이 숫자 만큼의 component를 만듭니다. Agent가 1인 경우 이미 하나의 클러스터이므로 component를 바로 반환하며, 이외의 경우 그래프의 component의 숫자가 agent가 될 때까지 while문을 통해서 betweenness가 가장 큰 Edge를 계속 제거하는 메소드입니다.

이 메소드는 generateCoverage( ), findRoute( ), findBestRoute( ) 메소드에 parameter로 사용되는 component를 생성하는데 사용됩니다.

(4) Class MultiAgentNetworkCoverage / Method findRoute( )

```
def findRoute(self, component, start):
    visited = {}
    path = []
    for i in range(len(component)):
        visited[component[i]] = False

    stack = Stack.Stack()
    stack.push(component[start])

    while stack.isEmpty() != True:
        current = stack.pop()
        if visited[current] == False:
            visited[current] = True
            path.append(current)
            neighborStation, neighborCost = self.objGraph.getNeighbors(current)
            for i in range(len(neighborStation)):
                if visited[neighborStation[i]] == False:
                    stack.push(neighborStation[i])

    # print("Path : ",path)
    return path
```

MultiAgentNetworkCoverage의 findRoute( ) 메소드는 DFS 알고리즘을 통해서 parameter로 입력받은 component를 탐색하는 알고리즘입니다. Parameter로 component와 start를 받습니다. Component의 경우 경로를 탐색하고자 하는 목표 component이고, start는 탐색을 시작하려는 위치를 담은 정수입니다. Visited 변수를 통해서 지나간 경로를 dictionary의 형태로 저장하고, 방문한 vertex의 이름을 path에 저장합니다. 이 메소드에서 위에서 구현한 stack 클래스가 사용됩니다.

Stack 클래스에 탐색을 시작하려는 vertex의 이름을 집어넣습니다. 이후 스택이 빌 때까지 stack에서 원소 하나를 pop한 후 그 vertex의 인접한 vertex를 찾아 만약 그 vertex를 아직 방문하지 않았다면 stack에 push합니다. 이 과정을 반복하여 그래프의 가장 깊숙한 정점까지 순서대로 방문합니다.

이 Homework에서 가장 최소한의 비용으로 주어진 component를 탐색할 때, 이미 방문했던 정점이라도 다른 정점을 방문하기 위해서 거쳐야 한다면 반드시 방문해야 합니다. 따라서 같은 정점을 여러 번 방문하는 것을 최소화 해야 하므로, DFS 알고리즘을 통해서 한 번 탐색할 때 가장

안쪽에 있는 vertex까지 방문하여 그 경로에 있는 정점을 최소한의 횟수로 방문하는 것이 효율적이라고 생각했습니다. 이 알고리즘을 통해서 그래프를 탐색한다면 환승역이 아닌 역의 경우 많아야 2번 들리게 되며, 환승역이라도 자신의 인접한 역 * 2배가 최다 방문 횟수가 됩니다.

(5) Class MultiAgentNetworkCoverage / Method findRoute( )

```
def findBestRoute(self, component):
    point = []
    lenNeighbor = []
    lenPath = []

    for i in range(len(component)):
        neighborStation, neighborCost = self.objGraph.getNeighbors(component[i])
        lenNeighbor.append(len(neighborStation))

    point.append(component[0])
    point.append(component[-1])

    for i in range(len(lenNeighbor)):
        if lenNeighbor[i] == 1:
            if not component[i] in point:
                point.append(component[i])

    # print(point)

    for i in range(len(point)):
        Route = self.findRoute(component, component.index(point[i]))
        path = self.findPath(Route)
        lenPath.append(path)

    min = 9999
    idxMin = None
    for i in range(len(lenPath)):
        if len(lenPath[i]) < min:
            idxMin = i

    # print(lenPath[idxMin])
    return lenPath[idxMin]
```

MultiAgentNetworkCoverage의 findBestRoute( ) 메소드는 주어진 component에서 최적의 경로가 될 수 있는 후보 vertex를 찾아서, 그 vertex에 대해서 findRoute( ) 메소드를 실행하는 메소드입니다. 여기서 최적의 경로를 만들 수 있는 후보 vertex의 경우 component의 양 끝과 자신 주위의 역이 1개가 있는, 즉 노선의 끝 역입니다. 따라서 이 vertex를 리스트 point에 삽입한 후 이 point의 원소들에 대해서 findRoute( ) 메소드를 실행하여 경로를 구합니다. 이후 이 경로의 길이가 가장 짧은 효율적인 경로를 찾아 반환합니다.

첨부한 이 알고리즘을 Plan 3라고 하겠습니다.

구상한 다른 plan의 경우, 후보 vertex로 환승역과 연결되어 있는 역의 개수가 최소인 vertex들을 추가로 넣는 것입니다. 이 알고리즘을 plan2라고 하겠습니다.

이 두 가지 plan에 대해서는 뒤에서 더 설명하겠습니다.

(6) Class MultiAgentNetworkCoverage / Method findPath( )

```
def findPath(self, visitedList):
    path = []
    path.append(visitedList[0])
    for i in range(1, len(visitedList)):
        neighborStation = self.objGraph.getNeighbors(path[-1])
        if visitedList[i] in neighborStation:
            path.append(visitedList[i])
        else:
            dist, p, course, list = Dijkstra.performDijkstra(self.objGraph, path[-1], visitedList[i])
            temp = list[1:]
            path = path + temp

    # print("Route : ", path)
    return path
```

MultiAgentNetworkCoverage의 findPath( ) 메소드는 위에서 설명한 (4)번과 (5)번 메소드를 통해서 얻은 결과를 변환해주는 메소드입니다. (4)와 (5)의 메소드를 통해서 얻은 결과는 방문한 vertex를 순서대로 이름을 기록한 리스트입니다. 따라서 그 사이의 경로를 담고 있지는 않으므로 실제로 agent가 이동한 경로가 아닙니다. 그러므로 두 vertex 사이의 경로를 추가하여 실제로 agent가 이동하는 경로를 찾는 메소드입니다.

Parameter를 통해 결과로 얻어진 list를 받은 후, 이 리스트를 하나씩 탐색합니다. 이 리스트의 i 번째 원소와 I +1 번째 원소를 비교하여 만약 두 vertex가 graph에서 인접한 경로일 경우 그대로 path에 추가합니다. 만약 둘이 인접하지 않은 경우 Dijkstra 알고리즘을 통해서 그 사이 최단 경로를 구하여 path에 추가합니다.

(7) Class MultiAgentNetworkCoverage / Method generateCoverage( )

```
def generateCoverage(self, intNumAgents):
    lstPlan = []
    self.newmanClustering(intNumAgents)
    print("Finished clusturing")
    for i in range(intNumAgents):
        print("Compnent : ", i, " ", self.component[i])
        a = self.findBestRoute(self.component[i])
        print("Route : ", a)
        print("-----------------------------------------------")
        lstPlan.append(a)
    print(lstPlan)
    return lstPlan
```

MultiAgentNetworkCoverage의 generateCoverage( ) 메소드는 parameter를 통해 입력받은 agent의 숫자를 통해서 그래프를 클러스터로 나눈 후 각 클러스터에 대해서 최적의 route를 찾아 그 결과를 list에 저장하는 메소드입니다.

intNumAgents 만큼 newmanclustering을 진행하여 그래프를 나눈 후, 각 component에 대해서 findBestRoute 메소드를 진행합니다. 이후 그 결과를 lstPlan에 삽입하고 반환합니다.

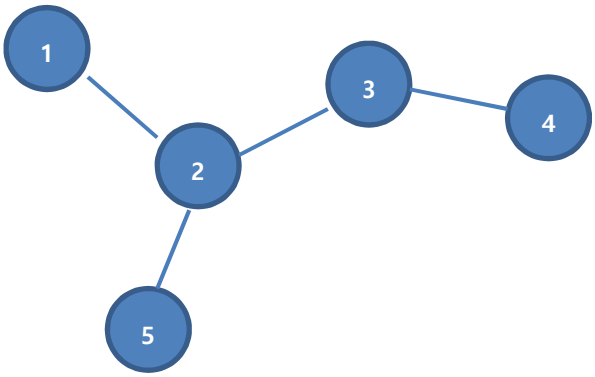
2. Algorithm Description

제한시간이 있는 만큼 제한 시간 내에 최적의 경로를 찾기 위하여 총 3가지 plan을 계획했습니다. 먼저 위에서 설명한 plan3, plan2가 있습니다. 그리고 시간이 촉박한 경우 빠른 시간에 적당한 경로를 찾는 plan1이 있습니다

```
def generateCoverage(self, intNumAgents):
    lstPlan = []
    self.newmanClustering(intNumAgents)
    print("Finished clusturing")
    for i in range(intNumAgents):
        print("Compnent : ", i, " ", self.component[i])
        a = self.findRoute(self.component[i], 0)
        p = self.findPath(a)
        print("-----------------------------------------------")
        lstPlan.append(p)

    return lstPlan
```

다음 코드처럼 findBestRoute( ) 알고리즘이 아닌 그냥 findRoute( ) 메소드를 통해서 첫 번째 정점에 대한 DFS 만 진행하는 것입니다.



다음과 같은 그래프에서 각 plan이 어떤 식으로 동작하는지 설명하겠습니다.

Plan1의 경우 첫 번째 정점인 1번 vertex에 대해서 DFS를 진행합니다. 따라서 1, 2, 3, 4, 3, 2, 5의 Route가 최적의 route로 구해지게 됩니다.

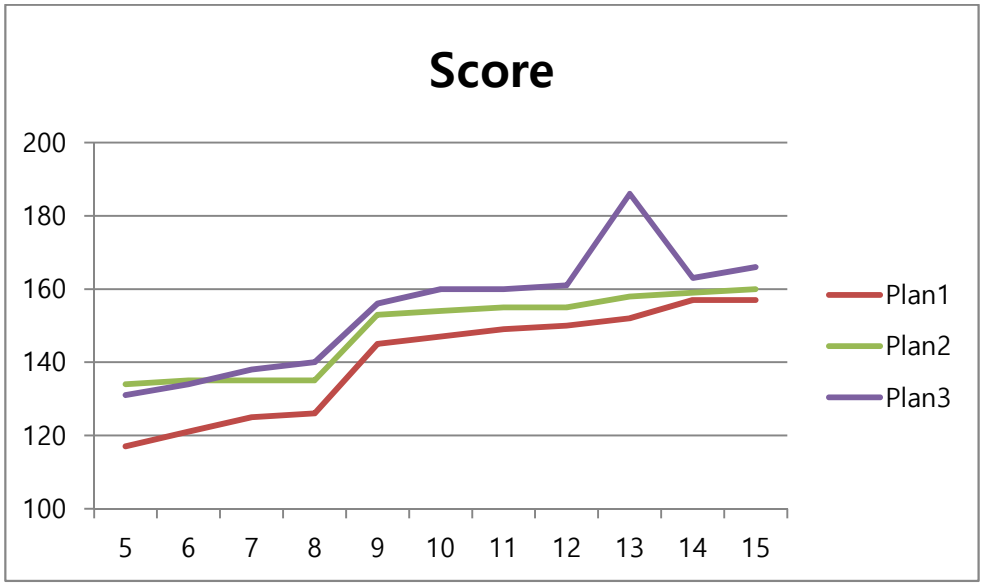
Plan2의 경우 1, 4, 5번 정점과 최다 환승역인 2번 정점에 대해서 모두 DFS를 진행합니다. 그리고 4, 3, 2, 1, 2, 5가 최단 경로로 구해집니다.

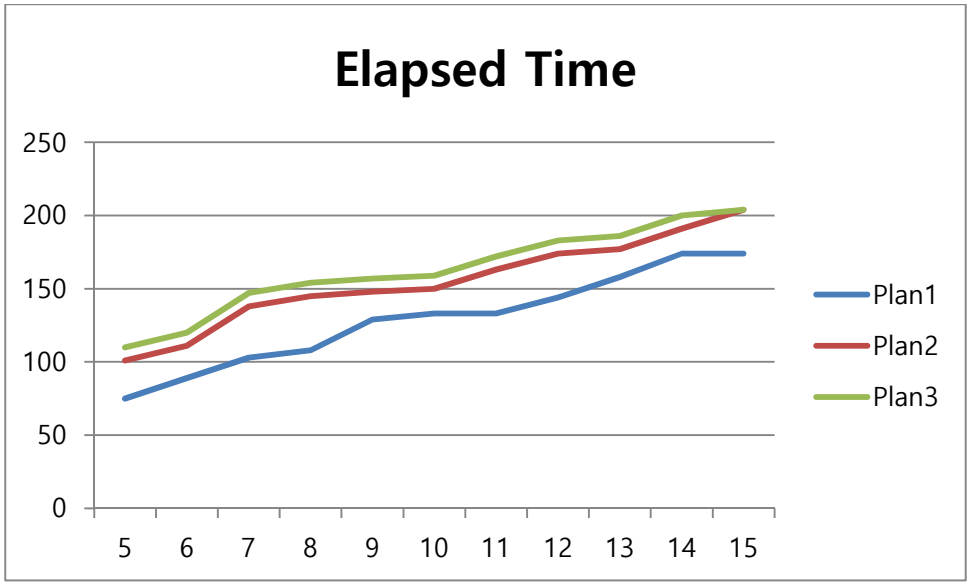
Plan3의 경우 1, 4, 5번 정점에 대해서만 DFS를 진행하고, Plan2와 동일한 결과를 얻습니다.

이렇게 세 가지 plan을 짠 후 agent가 5명부터 15명까지 변화할 때 Evaluate의 결과와 시간이 어떻게 되는지 분석하였습니다.

3. Result

agent	5	6	7	8	9	10	11	12	13	14	15
plan1 time	117	121	125	126	145	147	149	150	152	157	157
score	75	89	103	108	129	133	133	144	158	174	174
plan2 time	134	135	135	135	153	154	155	155	158	159	160
score	101	111	138	145	148	150	163	174	177	191	204
plan3 time	131	134	138	140	156	160	160	161	186	163	166
score	110	120	147	154	157	159	172	183	186	200	204





결과는 다음과 같았습니다. 보면 plan3가 시간과 결과 모든 면에 대해서 plan2에 비해 우월한 solution을 얻었습니다. Plan1에 비해서는 시간 대비 충분한 효율을 보여주므로 plan3를 선택하였습니다. 해당 실험을 진행한 pc에 비해서 연산 장치가 좋지 않은 경우 elapsed time의 격차가 더 커졌습니다. Plan1은 제한 시간 내에 진행하나 plan2 경우에는 진행하지 못하는 경우가 다수여서 더욱 plan3를 최적의 plan으로 선택하였습니다.