Python Project: Routing Algorithms

December 14, 2018

1 Project Description

The goal of the project is to implement two link state and distance vector routing algorithms on weighted graphs in Python. Given a .txt file as input, I use Chinese names of stations. The output is the least delay path or the least switching times with delay as low as possible.

1.1 Dijkstra's algorithm (link state)

In this project, I am using Dijkstra algorithm as the link-state routing algorithm. Since Link-state routing algorithm is a global routing algorithm, which means that this algorithm requires global knowledge, we can implement it just for one node to compute the least cost path between any two nodes. The information includes the topology of the network and all the costs. The process of Dijkstra algorithm is:

Algorithm 1 Dijkstra's algorithm for source node u (Ref: James F. Kurose, Keith W. Ross, *Computer Network: A Top-Down Approach*) [Kur05]

```
1: N = {u}
2: for all nodes v do
3:     if v is a neighbor of u then
4:         then D(v) = ∞
5:     end if
6: end for
7: while N ≠ S do
8:     find w not in N such that D(w) is a minimum
9:     add w to N
10:    update D(v) for each neighbor v of w and not in N:
11:    D(v) = min(D(v), D(w) + c(w, v)
12: end while
```

1.2 Bellman-Ford algorithm (distance vector)

Then I use Bellman-Ford algorithm as the distance vector routing algorithm. Distance vector routing algorithm is a distributed routing algorithm. Here, each node need to calculate path with local information at the beginning. The local information in this project is the set of nodes as neighbors

linked directly to current node. In each iteration, each node will exchange information with it neighbors and update the information it has. After certain numbers of iterations, each node will have all the information of the network and now we get the optimal path with minimum cost.

Algorithm 2 Bellman-Ford algorithm for source node u

```
1: for all nodes v do
2:     if v ≠ u then
3:         D(v) = ∞
4:     else
5:         D(v) = 0
6:     end if
7: end for
8: i = 1
9: for i ≤ |S| − 1 do
10:    for all edge w − v do
11:        if D(v) > D(w) + c(w, v) then
12:            then D(v) = D(w) + c(w, v)
13:        end if
14:    end for
15:    i + +
16: end for
```

1.3 Analysis

Dijkstra algorithm is much faster than Bellman-Ford algorithm. Since Dijkstra algorithm compute the least path using global knowledge from the beginning. In Bellman-Ford algorithm, every node has to exchange local information with its neighbors which takes a long time for convergence. However, Bellman-Ford algorithm is scalable, while Dijkstra algorithm is not.

2 Simulation

In the simulation, I write the two routing algorithms according to the introduction part of the report. These two algorithms can compute the least delay path directly. However, how to compute the least switch times path

can not is a challenge. –¿if there is an intersection point of several lines, number this point for different line with different number. If we want the least delay path, set the delays between these different numbered station (in fact they are the same station with different number) equal to 0. If we want the least switch time path, set the delays between these different numbered station equal to infinity. I changed Wu's solution in the following way: when compute the least delay path, it's the same as Wu's solution; however, when compute the least switch times path I set the delays equal to 1000, which is much larger than any delay between two directly connected nodes. The result of the simulation proves that my solution is effective. Some of my simulation result are shown in figures.

3 Conclusion

From this Routing Algorithm Project I have a deeper understanding of the link state and distance vector algorithm. I implemented these Dijkstra and Bellman-Ford distance vector algorithms via python.

From the simulation, link state routing algorithm is faster than distance vector routing algorithm. The reason is that link state routing algorithm is a global algorithm while distance vector routing algorithm is a distributed algorithm.

However, distance vector routing has its own advantage – it is scalable.

Code

3.1 Link-State

Link-State-routing.py

```
import sys

# Class to represent a graph used in Dijkstra's Alg.
class Graph:
    # To find the nodes with minimum distance(cost), from the queue of nodes wit
    def min_cost(self, cost, queue):
```

```
        # Initialize variables: min value to infinite and min_idx as -1
        min = float("Inf")
        min_idx = -1
        # from the cost array, pick the one which has the min value and is still
        for i in range(len(cost)):
            if i in queue and cost[i] < min:
                min = cost[i]
                min_idx = i
        return min_idx

    # To generate the lines of the output table:
    def line_list(self, set, cost, iteration):
        line_header = 'Iteration' + str(iteration)
        line_list = [line_header]
        for i in range(len(cost)):
            routing = str(cost[i])
            line_list.append(routing)
        line_list.append(str(set))
        return line_list

    # Dijkstra's algorithm
    # Find shortest distances(cost) from all the nodes to the target
    def dijkstra(self, graph, target):
        row = len(graph)
        col = len(graph[0])

        # cost[i] will hold the shortest distance(cost) from target to i
        # Initialize all distances to infinite:
        cost = [float("Inf")] * row

        # Distance of target node from itself is always 0:
        cost[target] = 0

        # Add all nodes in the to-be-determined queue, make a empty set for thos
        queue = []
        for i in range(row):
            queue.append(i)
        found_set = []
```

```
output_lines = []

# Find shortest path for all nodes:
while queue:
    # Pick the minimum dist(cost) node from the queue
    u = self.min_cost(cost, queue)    # the found node!

    # add already found nodes into set and remove it from the queue
    found_set.append(u)
    found_set.sort()
    queue.remove(u)

    # Update cost[j] only if it is in queue, there is an edge from found
    # to current node j, and total weight of path from target to current
    # founded u is smaller than current value of cost[j]
    for j in range(col):
        if graph[u][j] and j in queue:
            if cost[u] + graph[u][j] < cost[j]:
                cost[j] = cost[u] + graph[u][j]
    line = self.line_list(found_set, cost, len(found_set)-1)
    output_lines.append(line)

f = open(table_path, 'w')
# Make the table header
header = "Iterations"
for i in range(row):
    # node_name = "Node" + str(i)
    node_name = "Node" + str(i+1)
    header = header + '\t' + node_name
header = header + '\t' + "Set N"
f.write(header + '\n')
# Write the routing information
for row in output_lines:
    row_str = "\t".join(row)
    f.write('\n')
    f.write(row_str)
f.close()
print("Solution has been output in the ./OUTPUT_dijkstra.txt")
```

```
# Configuration of path
table_path = './OUTPUT_dijkstra.txt'          # path of output table

config = sys.argv[1]
paths = []
switches = []
graph = []


class Node:
    def __init__(self, current_node_info):
        self.curr_node = current_node_info[0]
        self.neighbors = []
        current_node_info = current_node_info[1:]
        i = 0
        while i < len(current_node_info):
            this_neighbor = [current_node_info[i], int(current_node_info[i+1])]
            self.neighbors.append(this_neighbor)
            i += 2
        path = []
        for neighbor in self.neighbors:
            path = [self.curr_node]
            path.append(neighbor[0])
            path.append(neighbor[1])
            paths.append(path)


class Nodes:
    def __init__(self, config):
        global config_file
        config_file = {}
        #reading file
        fo = open(config, "r+")
        data = fo.read()
        items = (data.split())

        global nodes
```

```
        nodes = []

        switchIdx = []

        i = 0
        while i < len(items):
            if items[i] == 'Node':
                switchIdx.append(i)
            i += 1
        lastNodeIdx = switchIdx[len(switchIdx)-1]
        r = 0
        while r+1 < len(switchIdx):
            str = switchIdx[r]
            end = switchIdx[r+1]
            node = Node(items[str+1: end])
            nodes.append(node)
            r += 1

        node = Node(items[lastNodeIdx+1: len(items)])
        nodes.append(node)

        for path in paths:
            if path[0] not in switches:
                switches.append(path[0])
            if path[1] not in switches:
                switches.append(path[1])


class Read:
    def __init__(self):
        cost_table = [[0 for x in range(len(switches))] for y in range(len(switche
        for path in paths:
            cost_table[int(path[0])-1][int(path[1])-1] = path[2]
            cost_table[int(path[1])-1][int(path[0])-1] = path[2]
        # print("2 5 4 : ", cost_table[1][4])
        # print("1 3 2 : ", cost_table[0][2])
        row_idx = 0
        while row_idx < len(cost_table):
```

```
            cost_table[row_idx] = list(map(int, cost_table[row_idx]))
            graph.append(cost_table[row_idx])
            row_idx += 1


print("<------ Implement of Dijkstra's algorithm ------>")
all_nodes = Nodes(config)
target = switches[0]
read = Read()
print("---- show switches in this test ----")
for s in switches:
    print("Node : ", s)
print("---- show paths in this test ----")
for path in paths:
    print(paths.index(path), path)
g = Graph()
g.dijkstra(graph, int(target))
```

Solution Output

```
Iterations  Node1 Node2 Node3 Node4 Node5 Set N

Iteration0  4      0     inf    3     4    [1]
Iteration1  4      0      5     3     4    [1, 3]
Iteration2  4      0      5     3     4    [0, 1, 3]
Iteration3  4      0      5     3     4    [0, 1, 3, 4]
Iteration4  4      0      5     3     4    [0, 1, 2, 3, 4]
```

3.2 Distance-Vector

Distance-Vector-routing.py

```
import sys

# Class to represent a graph used in Bellman-Ford Alg.
class Graph:
    def __init__(self, nodes):
        self.N = nodes  # No. of nodes
        self.graph = []
```

```
# To add an edge to graph (unidirectional, from u to v):
def add_edge(self, u, v, w):
    # u:parent node, v:current node w:cost
    self.graph.append([u, v, w])

# To generate the lines of the output table:
def line_list(self, cost, next_node, iteration):
    line_head = 'Iteration'+str(iteration)
    line_list = [line_head]
    for i in range(self.N):
        routing = "(" + str(next_node[i]) + ", " + str(cost[i]) + ")"
        line_list.append(routing)
    return line_list

# Bellman-Ford algorithm
# Find shortest distances from all the nodes to the target:
def bellmanford(self, target):
    # Initialize cost from target to all other nodes as infinite, target to
    # and the parent nodes of all other nodes as -1, of target as itself.
    cost = [float("Inf")] * self.N
    next_node = [int(-1)] * self.N
    cost[target] = int(0)
    next_node[target] = target
    parent_set_iter = [target]
    # Initialization of the output lines
    output_iter = 0
    output_lines = []
    line = self.line_list(cost, next_node, output_iter)
    output_lines.append(line)

    # Relax all edges |N| - 1 times.
    # Update cost value and parent index of the adjacent nodes of
    # the chosen nodes. Only consider those nodes still in queue
    for i in range(self.N - 1):
        parent_set = list(parent_set_iter)          # the parent set only ch
        for u, v, w in self.graph:
            if u in parent_set:
```

```
                    # use a temp parent set to avoid instant updates
                    # of parent nodes set before one iteration is done
                    parent_set_iter.append(v)
                    parent_set_iter = list(set(parent_set_iter))
                    if cost[u] != float("Inf") and cost[u] + w < cost[v]:
                        cost[v] = cost[u] + w
                        next_node[v] = u
            output_iter = output_iter + 1
            line = self.line_list(cost, next_node, output_iter)
            if (line[1:] == output_lines[-1][1:]) is False:
                output_lines.append(line)

        # Output the result into the txt, in a table form as the PPT shows.
        f = open(table_path,'w')
        # Make the table header
        header = "Iterations"
        for i in range(self.N):
            node_name = "Node" + str(i+1)
            header = header + '\t' + node_name
        f.write(header+'\n')
        # Write the routing information
        for row in output_lines:
            row_str = "\t".join(row)
            f.write('\n')
            f.write(row_str)
        f.close()
        print("Solution has been output in the ./OUTPUT_bellmanford.txt")

# Configuration of path
table_path = './OUTPUT_bellmanford.txt'          # path of output table

config = sys.argv[1]
paths = []
switches = []
graph = []


class Node:
```

```
    def __init__(self, current_node_info):
        self.curr_node = current_node_info[0]
        self.neighbors = []
        current_node_info = current_node_info[1:]
        i = 0
        while i < len(current_node_info):
            this_neighbor = [current_node_info[i], int(current_node_info[i+1])]
            self.neighbors.append(this_neighbor)
            i += 2
        path = []
        for neighbor in self.neighbors:
            path = [self.curr_node]
            path.append(neighbor[0])
            path.append(neighbor[1])
            paths.append(path)


class Nodes:
    def __init__(self, config):
        global config_file
        config_file = {}
        #reading file
        fo = open(config, "r+")
        data = fo.read()
        items = (data.split())

        global nodes
        nodes = []

        switchIdx = []

        i = 0
        while i < len(items):
            if items[i] == 'Node':
                switchIdx.append(i)
            i += 1
        lastNodeIdx = switchIdx[len(switchIdx)-1]
        r = 0
```

```
        while r+1 < len(switchIdx):
            str = switchIdx[r]
            end = switchIdx[r+1]
            node = Node(items[str+1: end])
            nodes.append(node)
            r += 1

        node = Node(items[lastNodeIdx+1: len(items)])
        nodes.append(node)

        for path in paths:
            if path[0] not in switches:
                switches.append(path[0])
            if path[1] not in switches:
                switches.append(path[1])


class Read:
    def __init__(self):
        cost_table = [[0 for x in range(len(switches))] for y in range(len(switche
        for path in paths:
            cost_table[int(path[0])-1][int(path[1])-1] = path[2]
            cost_table[int(path[1])-1][int(path[0])-1] = path[2]
        # print("2 5 4 : ", cost_table[1][4])
        # print("1 3 2 : ", cost_table[0][2])
        row_idx = 0
        while row_idx < len(cost_table):
            cost_table[row_idx] = list(map(int, cost_table[row_idx]))
            graph.append(cost_table[row_idx])
            row_idx += 1


print("<------ Implement of Bellman-Ford algorithm ------>")
all_nodes = Nodes(config)
target = switches[0]
print("number switches: ", len(switches))
read = Read()
g = Graph(len(switches))
```

```
print("---- show switches in this test ----")
for s in switches:
    print("Node : ", s)
print("---- show paths in this test ----")
for path in paths:
    print(paths.index(path), path)

for i in range(len(graph)):
    for j in range(len(graph[0])):
        if graph[i][j] != 0:
            u = i
            v = j
            w = graph[i][j]
            g.add_edge(u, v, w)
g.bellmanford(int(target))
```

Solution Output

```
Iterations  Node1    Node2    Node3    Node4    Node5

Iteration0  (-1, inf)(1, 0)(-1, inf)(-1, inf)(-1, inf)
Iteration1  (1, 4)   (1, 0)  (-1, inf)(1, 3)    (1, 4)
Iteration2  (1, 4)   (1, 0)  (3, 5)    (1, 3)    (1, 4)
```

References

[Kur05] James F Kurose. *Computer networking: A top-down approach featuring the internet, 3/E*. Pearson Education India, 2005.