# Assignment2

October 24, 2023

[ ]:
```
import matplotlib.pyplot as plt
import networkx as nx
from networkx.algorithms.flow import edmonds_karp
import random
import sys
import numpy as np
import graphviz
```

[ ]:
```
!apt install libgraphviz-dev
!pip install pygraphviz
```

[ ]:
```
class Graph:
    def __init__(self, no_of_nodes, connectivity, increase, seed):
        while 1:
            self.graph=nx.gnp_random_graph(no_of_nodes, connectivity, seed, True)
            if nx.is_weakly_connected(self.graph):
                break
            else:
                seed += increase

        #remove backflow
        for u in self.graph.nodes():
            adj_list = self.graph.adj[u]
            for nbr,datdict in adj_list.copy().items():
                # print (u,nbr)
                if (u>nbr) and self.graph.has_edge(u,nbr):
                    self.graph.remove_edge(u,nbr) # prune reverse edges from graph

        #add flow and capacity at random
        for u, v in self.graph.edges():
            capacity=random.randint(1,10)
            flow=random.randint(1,capacity)
            self.graph.add_edges_from([(u, v, {'flow':flow }), (u, v,{'capacity': capacity})])
```

```
        self.graph_flow_edges = nx.get_edge_attributes(self.graph, 'flow')
        self.graph_capacity_edges = nx.get_edge_attributes(self.graph,␣
↪'capacity')

        #insure the conservation is maintained
        inflow = [0 for i in range(no_of_nodes)] #an array of zeros
        for u in self.graph.nodes():
            e_in = self.graph.in_edges(u)
            e_out = self.graph.out_edges(u)
            if len(e_in) != 0 and len(e_out) != 0:
                #check and conserve flow
                for x,y in e_in:
                    search_tuple = (x,y)
                    result_tuple = self.getEdgeFlowAndCapacity(search_tuple)
                    inflow[y] += result_tuple[0]
                sum = inflow[u]
                edge_num = 0
                for u,w in e_out:
                    search_tuple = (u,w)
                    result_tuple = self.getEdgeFlowAndCapacity(search_tuple)
                    capacity = result_tuple[1]
                    if (edge_num < len(e_out) - 1):
                        while 1:
                            val = int(random.uniform(0,1) * sum)
                            if val <= capacity  and val <= sum:
                                break
                        sum = sum - val
                        self.updateFlow(u,w, val, capacity)
                        edge_num = edge_num + 1
                    else:
                        if sum > capacity:
                            #change the capacity
                            difference = sum - capacity
                            factor = random.randint(difference, difference + 9)
                            self.updateFlow(u,w, sum, capacity + factor)
                        else:
                            self.updateFlow(u,w, sum, capacity)

        self.graph_flow_edges = nx.get_edge_attributes(self.graph, 'flow')
        self.graph_capacity_edges = nx.get_edge_attributes(self.graph,␣
↪'capacity')
        # for u, v in self.graph.edges():
        #     search_tuple = (u,v)
        #     result_flow = self.getEdgeFlowAndCapacity(search_tuple)
        #     print("u,v,flow, capacity", u,v,result_flow[0], result_flow[1])
        self.no_of_nodes = self.graph.number_of_nodes()
```

```
        path_iterator = nx.all_simple_edge_paths(self.graph, source=0,␣
↪target=no_of_nodes - 1, cutoff=9)
        self.path = []
        for p in path_iterator:
            self.path.append(p)

    def getEdges(self, node):
        result = []
        for u,v in self.graph.edges():
            if u == node or v == node:
                result.append((u,v))
        return result

    def getFlows(self, node_pairs, node):
        in_flow = 0
        out_flow = 0
        for u,v in node_pairs:
            search_tuple = (u,v)
            flow = self.getEdgeFlowAndCapacity(search_tuple)
            if v == node:
                in_flow += flow[0]
            else:
                out_flow += flow[0]
        return [in_flow, out_flow]

    def checkConservation(self):
        a = len(self.graph.nodes())
        #only need to loop a - 2 times as we will not check the flow␣
↪conservation for the sink and source node
        for i in range(1,a-1):
            # go over the edges and find all (u,v) combination that has i␣
↪either in u or v
            node_pair = self.getEdges(i)
            # for (u,v) that has i in 1st index add the flow to inflow
            # for (u,v) that has i in the 2nd index add the flow to outflow
            flows = self.getFlows(node_pair, i)
            out_flow = flows[1]
            in_flow = flows[0]
            if in_flow != out_flow and in_flow != 0 and out_flow != 0:
                print('the flow for the node is not conserverd')
                print('node in out', i, flows[0], flows[1])
            # else:
            #     print("conserved")

    # search_tuple is made of nodes.
    # (1, 2) represents the tuple of node 1 and node 2
    def getEdgeFlowAndCapacity(self, search_tuple):
```

```
        flow = -1
        capacity = -1
        for key in self.graph_flow_edges:
            if key == search_tuple:
                flow = self.graph_flow_edges[key]
                capacity = self.graph_capacity_edges[key]
        return (flow, capacity)

    # where u and v are nodes
    def updateFlow(self, u, v, new_flow, capacity):
        self.graph.add_edges_from([(u, v, {'flow': new_flow}), (u, v,{'capacity': capacity})])
        self.graph_flow_edges = nx.get_edge_attributes(self.graph, 'flow')

    def plotGraph(self):
        links = [(u, v) for (u, v, d) in self.graph.edges(data=True)]
        pos = nx.nx_agraph.graphviz_layout(self.graph)
        nx.draw_networkx_nodes(self.graph, pos, node_size=1200, node_color='lightblue', linewidths=0.25) # draw nodes
        nx.draw_networkx_edges(self.graph, pos, edgelist=links, width=4)                              # draw edges
        nx.draw_networkx_labels(self.graph, pos, font_size=20, font_family="sans-serif")                 # node labels
        edge_labels = nx.get_edge_attributes(self.graph, 'flow')                                     # edge weight labels
        # print('edge labels', edge_labels)
        nx.draw_networkx_edge_labels(self.graph, pos, edge_labels)
        plt.show()

    def simulateGraph(self):
        self.graph = nx.DiGraph()
        self.graph.add_edges_from([
            (0, 1, {'flow': 2}), (0, 1,{'capacity': 6}),
            (0, 2,{'flow': 3}), (0, 2,{'capacity': 3}),
            (0,3,{'flow': 4}), (0,3,{'capacity': 5}),
            (1,2, {'flow': 1}), (1,2,{'capacity': 5}),
            (1,5, {'flow': 1}), (1,5,{'capacity': 3}),
            (2,4, {'flow': 2}), (2,4,{'capacity': 9}),
            (2,5, {'flow': 4}), (2,5,{'capacity': 8}),
            (3,2, {'flow': 2}), (3,2,{'capacity': 2}),
            (3,4,{'flow': 2}), (3,4,{'capacity': 3}),
            (4,5, {'flow': 4}), (4,5,{'capacity': 5})])

        self.graph_flow_edges = nx.get_edge_attributes(self.graph, 'flow')
        self.graph_capacity_edges = nx.get_edge_attributes(self.graph, 'capacity')
```

```
        self.no_of_nodes = self.graph.number_of_nodes()
        path_iterator = nx.all_simple_edge_paths(self.graph, source=0,␣
↪target=5, cutoff=9)
        self.path = []
        for p in path_iterator:
            self.path.append(p)

    def makeFlowConsistent(self):
        pass
```

[ ]:
```
class Agent:
    def __init__(self, no_of_nodes, connectivity, simulate = 0):
      seed = 1000
      self.graph = Graph( no_of_nodes, connectivity, 0.01, seed)
      if simulate == 1:
        self.graph.simulateGraph()
        print("INITIAL GRAPH")
        self.graph.plotGraph()

    def getMaxIndex(self, heuristic_value_list):
        maxx = -1
        maxIndex = -1
        index = 0
        for v in heuristic_value_list:
          if v > maxx:
            maxIndex = index
            maxx = v
          index += 1
        return (maxIndex, maxx)


    def heuristicFunction(self):
        # 1. calculate the heuristic value for each path and store in␣
↪heuristic_value_list
        # Nodes make up edges and collection of nodes make up path so,
        # iterate over each path
        heuristic_value_list = []
        for p in self.graph.path:
            minn = sys.maxsize
            # print('path ', p)
            # once inside the path iterate over each edges i.e the connectors␣
↪between nodes
            for u,v in p:
                search_tuple = (u,v)
                result_tuple = self.graph.getEdgeFlowAndCapacity(search_tuple)
                remaining_flow = result_tuple[1] - result_tuple[0]
                # print(u,v , remaining_flow)
```

```python
                if remaining_flow == 0:
                    minn = 0
                    break
                else:
                    minn = min(minn, remaining_flow)
            # print('heuristic value is', minn)
            heuristic_value_list.append(minn)
        # 2. find the max heuristic in the heuristic_value_list and return the␣
↪path
        if np.any(heuristic_value_list):
            result = self.getMaxIndex(heuristic_value_list)
            max_heuristic_index = result[0]
            max_heuristic_value = result[1]
            return (self.graph.path[max_heuristic_index], max_heuristic_value)
        else:
            print("all zeross")
            return ()

    def stocasticHeuristicFunction(self):
        # 1. calculate the heuristic value for each path and store in␣
↪heuristic_value_list
        # Nodes make up edges and collection of nodes make up path so,
        # iterate over each path
        heuristic_value_list = []
        for p in self.graph.path:
            minn = sys.maxsize
            # once inside the path iterate over each edges i.e the connectors␣
↪between nodes
            for u,v in p:
                search_tuple = (u,v)
                result_tuple = self.graph.getEdgeFlowAndCapacity(search_tuple)
                remaining_flow = result_tuple[1] - result_tuple[0]
                if remaining_flow == 0:
                    minn = 0
                    break
                else:
                    minn = min(minn, remaining_flow)
            heuristic_value_list.append(minn)
        # 2. find the max heuristic in the heuristic_value_list and return the␣
↪path stocastically
        # calculate the probability of each heuristic value
        sum_array = sum(heuristic_value_list)
        probability_weight = []

        if sum_array != 0:
            for h in heuristic_value_list:
                probability_weight.append(int((h/sum_array) * 100 ))
```

```
            result = random.choices(heuristic_value_list, weights=␣
↪probability_weight, k=1)
            heuristic_index = heuristic_value_list.index(result[0])
            return (self.graph.path[heuristic_index], result[0])
        else:
            # print("all zeross")
            return ()


    def optimizeHeuristic(self, max_heuristic_child_list, increase_factor):
        for u, v in max_heuristic_child_list:
            search_tuple = (u, v)
            result_tuple = self.graph.getEdgeFlowAndCapacity(search_tuple)
            new_flow = result_tuple[0] + increase_factor
            self.graph.updateFlow(u,v, new_flow, result_tuple[1])

    def getOptimizedFlow(self):
        # do not consider the nodes that cant be reached to as edmonds karp␣
↪does the same
        ignore_node = []
        for n in range(self.graph.no_of_nodes):
            e_in = self.graph.graph.in_edges(n)
            e_out = self.graph.graph.out_edges(n)
            source_node = 0
            sink_node = self.graph.no_of_nodes - 1
            if len(e_in) == 0 and n != source_node:
                ignore_node.append(n)
            else:
                #check if the only in node is from the ignored node
                if len(e_in) == 1:
                    for u,v in e_in:
                        for i in ignore_node:
                            if u == i:
                                ignore_node.append(n)
                                break

        sink_node = self.graph.no_of_nodes - 1
        flow = 0
        for u,v in self.graph.graph_flow_edges:
            if v == sink_node:
                if u in ignore_node:
                    continue
                search_tuple = (u, v)
                result_tuple = self.graph.getEdgeFlowAndCapacity(search_tuple)
                flow = flow + result_tuple[0]
        return flow
```

```
    def optimize(self):
        # 1. compute the optimized flow using edmonds karp algorithm
        R = edmonds_karp(self.graph.graph, 0, self.graph.no_of_nodes - 1,␣
↪'capacity')
        flow_value = nx.maximum_flow_value(self.graph.graph, 0, self.graph.
↪no_of_nodes - 1)

        print('OPTIMAL FLOW', flow_value)

        while(1):
            # 2. compute heuristic function on each path
            res = self.heuristicFunction()

            # 3. optimize the flow for the chosen path
            if res == ():
                print("Done optimizing")
                break
            else:
                max_heuristic_child_list = res[0]
                heuristic = res[1]
                self.optimizeHeuristic(max_heuristic_child_list, heuristic)

            print("After applying heuristic function")
            self.graph.plotGraph()

        print("The optimised flow and capacity are")
         # 4. print the edges flow and capacity
        for u, v in self.graph.graph.edges():
            search_tuple = (u,v)
            result_flow = self.graph.getEdgeFlowAndCapacity(search_tuple)
            print("u -> v, flow , capacity", u,v,result_flow[0], result_flow[1])

        # 5. once optimized print the optimized flow
        optimised_flow = self.getOptimizedFlow()
        print('optimised flow', optimised_flow)

    def optimizeStostacilly(self):
        # 1. compute the optimized flow using edmonds karp algorithm
        R = edmonds_karp(self.graph.graph, 0, self.graph.no_of_nodes - 1,␣
↪'capacity')
        flow_value = nx.maximum_flow_value(self.graph.graph, 0, self.graph.
↪no_of_nodes - 1)

        print('OPTIMAL FLOW', flow_value)

        while(1):
            # 2. compute heuristic function on each path
```

```python
            res = self.stocasticHeuristicFunction()

            # 3. optimize the flow for the chosen path
            if res == ():
                print("Done optimizing")
                break
            else:
                max_heuristic_child_list = res[0]
                heuristic = res[1]
                self.optimizeHeuristic(max_heuristic_child_list, heuristic)

        # 4. once optimized print the optimized flow
        optimised_flow = self.getOptimizedFlow()
        print('optimised flow', optimised_flow)

        return (flow_value, optimised_flow)
```

[ ]:
```python
# Requirement 1
if __name__ == "__main__":
    agent = Agent(30,0.1,1)
    agent.graph.checkConservation()
    agent.optimize()
```

[ ]:
```python
# Requirement 2
# Mostly overfits (performs better than the edmonds karp algorithm), sometimes␣
 ↪underperforms or has equal flow
# The output in 3a will give you an idea of what I mean
# The hill climbing algorithm works perfectly though
if __name__ == "__main__":
    agent = Agent(30,0.1,0)
    agent.graph.checkConservation()
    flows = agent.optimizeStostacilly()
```

[ ]:
```python
# Requirement 3a
if __name__ == "__main__":

    max = 0
    hill = 0
    for i in range(30):
        agent = Agent(30,0.1,0)
        agent.graph.checkConservation()
        flows = agent.optimizeStostacilly()
        max += flows[0]
        hill += flows[1]
        print()

    print("Average EK flow =", max/30)
```

```
    print("Average HC flow =", hill/30)
```

[ ]:
```
# Requirement 3b
if __name__ == "__main__":

    max = 0
    hill = 0
    for i in range(30):
        agent = Agent(30,0.07)
        agent.graph.checkConservation()
        flows = agent.optimizeStostacilly()
        max += flows[0]
        hill += flows[1]
        print()

    print("Average EK flow =", max/30)
    print("Average HC flow =", hill/30)
```

[ ]:
```
# !sudo apt-get install texlive-xetex texlive-fonts-recommended␣
 ↪texlive-plain-generic

!jupyter nbconvert --to pdf /content/drive/MyDrive/Colab\ Notebooks/Assignment2.
 ↪ipynb
```