Network Flow with Congestion

Derryl Sayo

```python
import numpy as np
import cvxpy as cp
import networkx as nx
```

```
(CVXPY) Apr 15 12:02:36 PM: Encountered unexpected exception importing solver GLOP:
RuntimeError('Unrecognized new version of ortools (9.9.3963). Expected < 9.8.0. Please open a
(CVXPY) Apr 15 12:02:37 PM: Encountered unexpected exception importing solver PDLP:
RuntimeError('Unrecognized new version of ortools (9.9.3963). Expected < 9.8.0. Please open a
```

Network Min-Cost Flow

The Network Flow problem involves finding the **cheapest** way to send some set of supply sources through a collection of paths with set costs to reach some set of demand sinks.

This problem has many applications in fields such as distribution systems in operations planning to internet routing systems.

Adding Congestion Costs

In real-life scenarios, however, we are limited by physics, city zoning, etc. which can impact the amount we can transport over a single wire/road/path. The question we will answer and analyze below is how do we represent this mathematically and how different are the solutions when we add congestion built into our mathematical network flow model?

The core formulation will explain and iterate upon the network flow problem statement in (Vanderbei 2020).

Aside: Some Graph Terminology

Like in Graph Coloring, Network Flow is solved over a graph.

- Let V be the set of vertices/nodes.
- Let $E \subset \{(x, y) \mid x, y \in V, x \neq y\}$ be the set of edges. An edge exists between two distinct vertices and the set of edges can include any number of the possible pairs of vertices. We can represent these as tuples of vertices $(x, y) \in E$.
- Then, let $G = (V, E)$ be a **graph**. A graph can be represented (ie. drawn) using only the set of edges/vertices.
- $|A|$ is the cardinality of set A which is the number of elements in the set.

Representation as a Linear Programming Problem

Consider a network that is represented by a set of vertices V and edges $E \subseteq \{(x, y) \mid x, y \in V, x \neq y\}$ which make a graph $G = (V, E)$.

Then, we can assign a cost to each edge $c_{i,j},\ (i, j) \in E$ which represents the cost of transportation along edge $(i, j) \in E$. Furthermore, we assign a "contribution" value $b_i,\ i = 1, \dots, |V|$ to each node in the network. If $b_i < 0$, we consider this node to be a "supplier"/source where the flow will originate from. If $b_i > 0$ these are "consumers"/sinks where the flow should end. If $b_i = 0$, then these are neutral nodes and could represent something that simply passes along the flow.

Decision Variables

We need to decide how much supply to send over a link to reach a demand destination. So, as is typical of assignment type decision variables we assign each (i, j) into an edge matrix and use the variable x_{ij} to be how much flow we are sending through edge (i, j) in the final solution.

Objective Functions

Network Flow

The typical congestion flow objective is to minimize the total cost of sending x_{ij} over edge (i, j) with edge cost c_{ij}:

$$\min \sum_{(i,j) \in E} c_{ij} x_{ij}$$

Network Flow with Congestion

Congestion occurs when when we try to transport too much over a single link (eg. if we all try to drive on the same highway, we create traffic). Thus, we can modify the objective function by multiplying the cost above again by how much we are sending over (i, j). This will penalize the total cost if we try to allocate too much over a single link since now we have the quadratic objective.

$$\min \sum_{(i,j)\in E} c_{ij}x_{ij}^2$$

Constraints

Further Assumptions

To simplify the model, we will add the following assumption for the design of the problem itself.

In order to make sure that the solution is feasible and all supply is accounted for, we must have a fully contained system where the supply and demand are equal (otherwise, we may non-deterministically add/remove cost from the system).

$$\sum_{i=1}^{|V|} b_i = 0$$

Then, the constraints are common between the two objective functions and are formulated as follows.

Non-negativity

The flow must be non-negative (we obviously can't send negative delivery trucks down a highway):

$$x_{ij} \geq 0, \forall (i,j) \in E$$

Flow Balance

The flow into (ie. $\sum_i x_{ik}$) and out of a node k (ie. $\sum_j x_{kj}$) must be equal to the demand of that node (ie. $-b_k$) (called the **flow balance constraints**). This is combined with assumption 1 above to make sure that the end solution is unique; if total $\mathit{supply}+\mathit{demand} = 0$, then the flows are balanced such that $\mathit{supply} = -\mathit{demand}$ at each node. No cost will not be unaccounted for and therefore we will be able to uniquely find a solution.

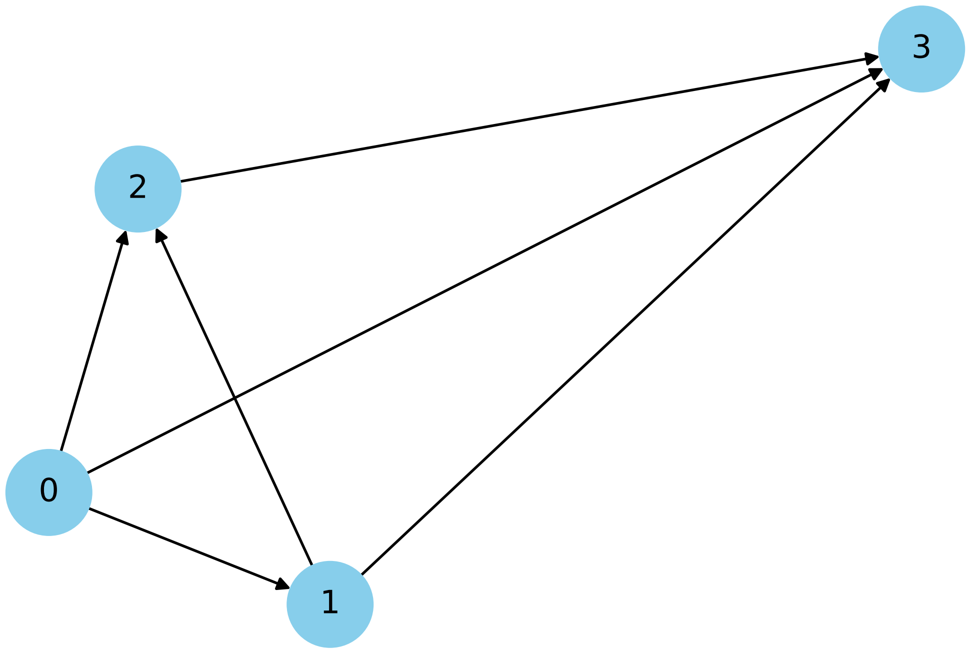
$$\sum_i x_{ik} - \sum_j x_{kj} = b_k,\ \forall (i,k),(k,j) \in E,\ \forall k \in V$$

The flow balance constraint seems pretty abstract, but its representation in matrix form will help explain it more clearly.

Since LP problem constraints are traditionally represented as matrices, the flow balance constraints form a convenient way to fully encode the edges in a incidence matrix (see Wikipedia contributors 2022).

For example, we create the incidence matrix for the following graph (which we will use as an example below)

```
G = nx.DiGraph()
G.add_nodes_from(
  [
    (0, {"demand": -5}),
    (1, {"demand": 0}),
    (2, {"demand": 0}),
    (3, {"demand": 5}),
   ]
)

G.add_weighted_edges_from(
  [
    (0, 1, 3),
    (0, 2, 6),
    (0, 3, 1),
    (1, 3, 1),
    (1, 2, 3),
    (2, 3, 2),
  ]
)

nx.draw(G, with_labels=True, node_color="skyblue", node_size=1000, font_size=12)
```



$$A = \begin{bmatrix} -1 & -1 & -1 & 0 & 0 & 0 \\ 1 & 0 & 0 & -1 & -1 & 0 \\ 0 & 1 & 0 & 0 & 1 & -1 \\ 0 & 0 & 1 & 1 & 0 & 1 \end{bmatrix}$$

Each row is the node i and each column represents some edge with index j where

$$a_{ij} = \begin{cases} -1 & \text{edge } j \text{ leaves node } i \\ 1 & \text{edge } j \text{ enters node } i \end{cases}$$

For example the $j = 3$ column represents the edge $(1, 3)$.

Going though each column, we have that the index of the -1 is the head node and the index of the 1 is the tail node. Then, the flow balance constraints can be interpreted by going through each vertex (ie. the rows) and making sure that all flow determined by going in and out is b_i.

Edge Capacity

If needed, upper bound capacity constraints. If present, we add new constants u_{ij}, $\forall (i,j) \in E$ where we set some limit on how much flow we choose for x_{ij}. If we do not define any, we implicitly set the limit to infinity.

$$x_{ij} \leq u_{ij}, \ \forall (i,j) \in E$$

Solutions using Python

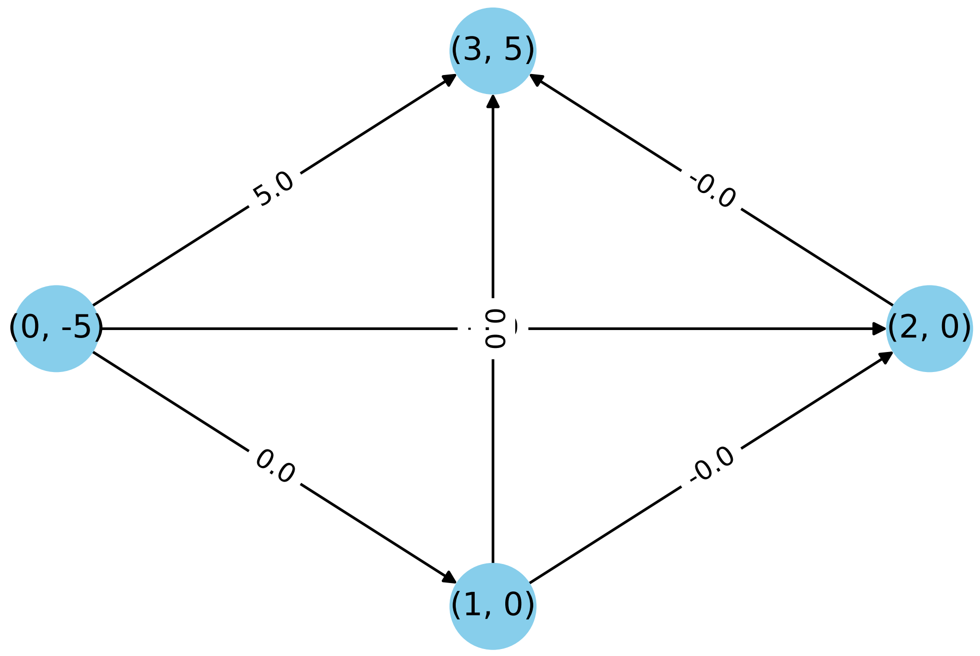
We write the min-cost network flow problem solution using CVXPY in python:

```python
def min_cost_network_flow(G, with_congestion=False):
  """
  Calculates the min-cost network flow for a given graph G.

  Params:
    - G: NetworkX graph
    - congestion: Flag to solve network flow with congestion

  Returns:
    - Min-cost network flow solution dictionary of form {(i,j): flow}
  """

  vertices = G.nodes
  edges = G.edges
  num_vertices = len(vertices)
  num_edges = len(edges)

  X = cp.Variable(num_edges)
  C = np.array([edge[2]["weight"] for edge in G.edges(data=True)])
  b = np.array(list(nx.get_node_attributes(G, "demand").values()))

  # Assertion that supply/demand is balanced
  assert np.sum(b) == 0

  A = nx.incidence_matrix(G, oriented=True).toarray()

  objective = cp.Minimize(cp.sum(C @ X))

  if with_congestion:
    objective = cp.Minimize(cp.sum(C @ cp.square(X)))

  constraints = []

  # Non-negative flow
  constraints = constraints + [X >= 0]
  # Flow balance - we define the incidence matrix using NetworkX
  # and multiply by X to determine the flow in/out of a node
  constraints = constraints + [A @ X <= b]

  problem = cp.Problem(objective, constraints)
  problem.solve()

  flow_dict = None

  if problem.status not in ["infeasible", "unbounded"]:
    flow_dict = {edge: value for edge, value in zip(G.edges, np.round(X.value, 4))}

  # Get linear objective value - needed since congestion will
  # inflate the objective value rather than return the actual
  # cost
  cost = C @ X.value

  return flow_dict, cost


def show_flows(G, flow_dict):
  """
  Display min-cost network flow solution for a graph G using flows in flow_dict

  Params:
    - G: NetworkX graph
    - flow_dict: Dictionary of edges and flow value, from `min_cost_network_flow()`
  """
  pos = nx.shell_layout(G)
  nx.draw(G, pos, with_labels=False, node_color="skyblue", node_size=1000, font_size=12)

  demand_dict = nx.get_node_attributes(G, "demand")
  node_labels = {node_idx: (node_idx, demand) for node_idx, demand in demand_dict.items()}
  nx.draw_networkx_labels(G, pos, labels=node_labels)
  nx.draw_networkx_edge_labels(G, pos, edge_labels=flow_dict)
```

We can now compare our solution against NetworkX's `min_cost_flow()` function with a simple graph:

```python
G = nx.DiGraph()
G.add_nodes_from(
  [
    (0, {"demand": -5}),
    (1, {"demand": 0}),
    (2, {"demand": 0}),
    (3, {"demand": 5}),
   ]
)

G.add_weighted_edges_from(
  [
    (0, 1, 3),
    (0, 2, 6),
    (0, 3, 1),
    (1, 3, 1),
    (1, 2, 3),
    (2, 3, 2),
  ]
)


min_cost_no_congestion, objective_value = min_cost_network_flow(G, with_congestion=False)
print(min_cost_no_congestion)
print(objective_value)

nx_flowDict = nx.min_cost_flow(G)
print(nx_flowDict)
print(nx.cost_of_flow(G, nx_flowDict))

show_flows(G, min_cost_no_congestion)
```

```
{(0, 1): 0.0, (0, 2): -0.0, (0, 3): 5.0, (1, 3): 0.0, (1, 2): -0.0, (2, 3): -0.0}
4.999999999897586
{0: {1: 0, 2: 0, 3: 5}, 1: {3: 0, 2: 0}, 2: {3: 0}, 3: {}}
5

/home/derry/.local/share/virtualenvs/MATH441-vlJmoJsg/lib/python3.8/site-packages/cvxpy/redu
    Your problem is being solved with the ECOS solver by default. Starting in
    CVXPY 1.5.0, Clarabel will be used as the default solver instead. To continue
    using ECOS, specify the ECOS solver explicitly using the ``solver=cp.ECOS``
    argument to the ``problem.solve`` method.

  warnings.warn(ECOS_DEPRECATION_MSG, FutureWarning)
```



Comparing Solutions With and Without Congestion

We compare the min-cost network flow solutions for the graph in Vanderbei example 14.1 (see Vanderbei 2020, 230–31)

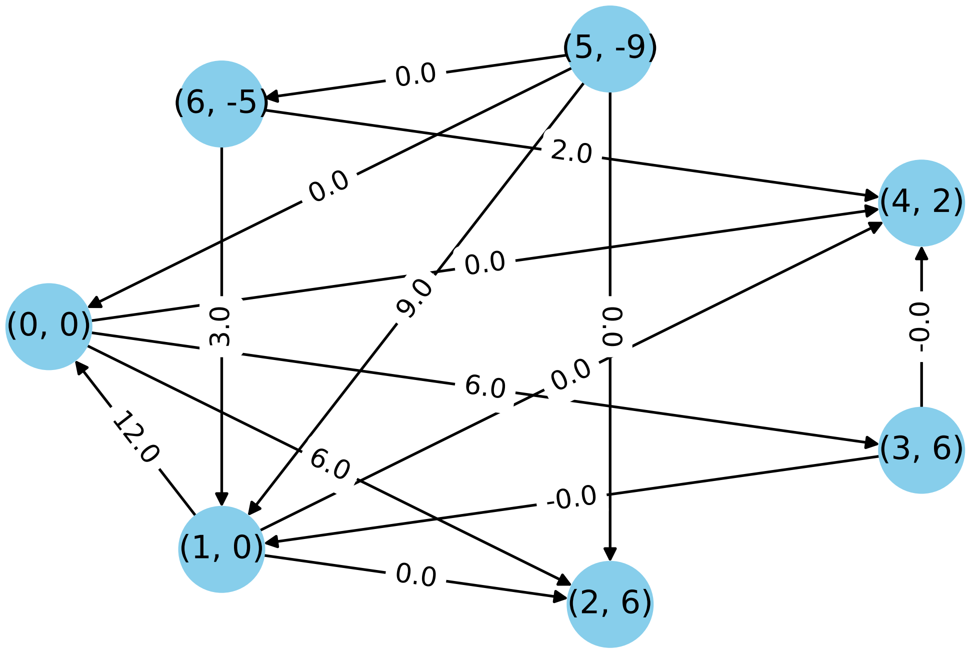
```
G = nx.DiGraph()
G.add_nodes_from(
  [
    (0, {"demand": 0}),
    (1, {"demand": 0}),
    (2, {"demand": 6}),
    (3, {"demand": 6}),
    (4, {"demand": 2}),
    (5, {"demand": -9}),
    (6, {"demand": -5}),
   ]
)

G.add_weighted_edges_from(
  [
    (0, 2, 48), (0, 3, 28), (0, 4, 10),
    (1, 0, 7), (1, 2, 65), (1, 4, 7),
    (3, 1, 38), (3, 4, 15),
    (5, 0, 56), (5, 1, 48), (5, 2, 108), (5, 6, 24),
    (6, 1, 33), (6, 4, 19),
  ]
)
```

Min-cost network flow without congestion

```
min_cost_no_congestion, objective_value = min_cost_network_flow(G, with_congestion=False)
print(min_cost_no_congestion)
print(objective_value)
show_flows(G, min_cost_no_congestion)

# NX solution for extra validation
nx_flowDict = nx.min_cost_flow(G)
print(nx_flowDict)
print(nx.cost_of_flow(G, nx_flowDict))
```

```
/home/derry/.local/share/virtualenvs/MATH441-vlJmoJsg/lib/python3.8/site-packages/cvxpy/reduc
    Your problem is being solved with the ECOS solver by default. Starting in
    CVXPY 1.5.0, Clarabel will be used as the default solver instead. To continue
    using ECOS, specify the ECOS solver explicitly using the ``solver=cp.ECOS``
    argument to the ``problem.solve`` method.

  warnings.warn(ECOS_DEPRECATION_MSG, FutureWarning)
```

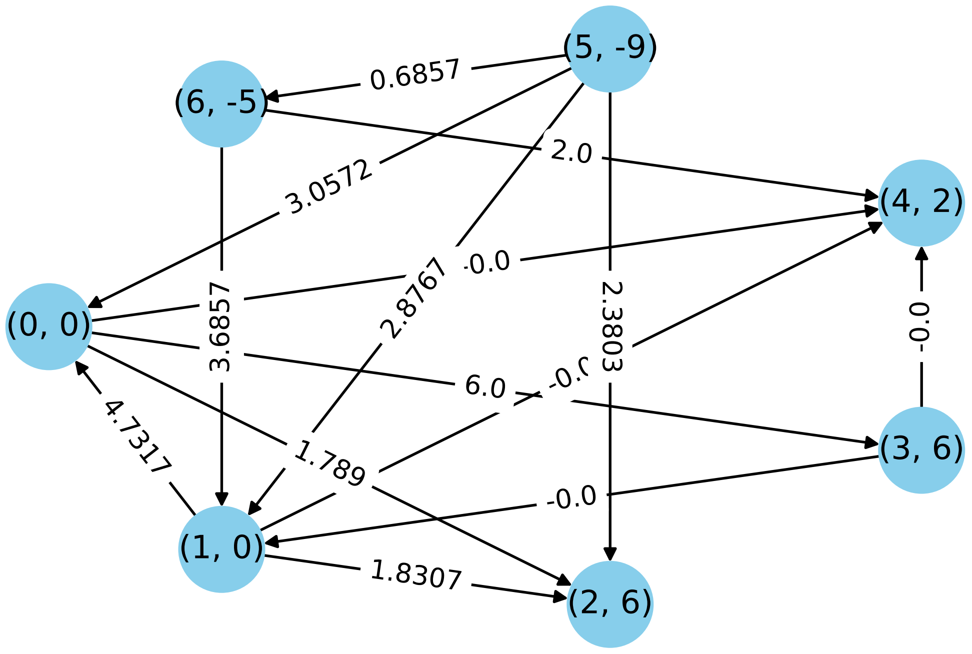
```
{(0, 2): 6.0, (0, 3): 6.0, (0, 4): 0.0, (1, 0): 12.0, (1, 2): 0.0, (1, 4): 0.0, (3, 1): -0.0
1109.0000000447496
{0: {2: 6, 3: 6, 4: 0}, 1: {0: 12, 2: 0, 4: 0}, 2: {}, 3: {1: 0, 4: 0}, 4: {}, 5: {0: 0, 1: 9
1109
```



Min-cost network flow with congestion

```
min_cost_with_congestion, objective_value = min_cost_network_flow(G, with_congestion=True)
print(objective_value)
show_flows(G, min_cost_with_congestion)
```

```
1148.4367330171174
```



Conclusion and Analysis of Results

As seen in the above, adding the congestion term works in spreading out the flow load around a larger subset of the edges (also depending on the network transportation costs). There could be many uses for adding congestion costs in order to spread out the flow along more edges. For example, packet delivery in an internet network is sensitive to congestion enough to add specific protocols to add mechanisms to control the entry of packets into the network. This could help give a more realistic analysis on packet flow as to not overwhelm the network.

Another interesting lesson this can teach us is about the **design** of optimization problems. This congestion term adds a more "soft constraint" similar to regularization terms (eg. max $f(x) - \lambda g(x)$ where $\lambda g(x)$ is a penalty) to make sure that the problem is able to consider "warnings" rather than restricting it using explicit constraints. For example, there is nothing restricting the feasibility of sending a lot of data through a single link, but there might be other concerns with slowing down the network if it does happen.

In summary, there are a variety of ways to improve the mathematical model for an optimization problem to better fit the intricacies of a generic problem statement which can include the technique detailed above of restricting flow by quadratically increasing the cost as more flow is sent through a single edge.

References

Vanderbei, Robert J. 2020. "Network Flow Problems." In *Linear Programming: Foundations and Extensions*, 229–56. Cham: Springer International Publishing. https://doi.org/10.1007/978-3-030-39415-8_14.

Wikipedia contributors. 2022. "Incidence Matrix — Wikipedia, the Free Encyclopedia." https://en.wikipedia.org/w/index.php?title=Incidence_matrix&oldid=1109920869.