CS 170 Homework 9

Due **11/1/2023, at 10:00 pm (grace period until 11:59pm)**

1 Study Group

List the names and SIDs of the members in your study group. If you have no collaborators, you must explicitly write "none".

2 Canonical Form LP

Recall that any linear program can be reduced to a more constrained *canonical form* where all variables are non–negative, the constraints are given by $\leq$ inequalities, and the objective is the maximization of a cost function.

More formally, our variables are x_i. Our objective is $\max c^\top x = \max \sum_i c_i x_i$ for some constants c_i. The jth constraint is $\sum_i a_{ij} x_i \leq b_j$ for some constants a_{ij}, b_j. Finally, we also have the constraints $x_i \geq 0$.

An example canonical form LP:

$$\text{maximize } 5x_1 + 3x_2$$

$$\text{subject to } \begin{cases} x_1 + x_2 - x_3 \leq 1 \\ -(x_1 + x_2 - x_3) \leq -1 \\ -x_1 + 2x_2 + x_4 \leq 0 \\ -(-x_1 + 2x_2 + x_4) \leq 5 \\ x_1, x_2, x_3, x_4 \geq 0 \end{cases}$$

For each of the subparts below, describe how we should modify it to so that it satisfies canonical form. If it is impossible to do so, justify your reasoning. Note that the subparts are independent of one another.

(a) Min Objective: $\min \sum_i c_i x_i$

(b) Lower Bound on Variable: $x_1 \geq b_1$

(c) Bounded Variable: $b_1 \leq x_1 \leq b_2$

(d) Equality Constraint: $x_2 = b_2$

(e) More Equality Constraint: $x_1 + x_2 + x_3 = b_3$

(f) Absolute Value Constraint: $|x_1 + x_2| \leq b_2$ where $x_1, x_2 \in \mathbb{R}$

(g) Another Absolute Value Constraint: $|x_1 + x_2| \geq b_2$ where $x_1, x_2 \in \mathbb{R}$

(h) Min Max Objective: $\min \max(x_1, x_2)$

Hint: use a dummy variable!

(i) Unbounded Variable: $x_4 \in \mathbb{R}$

Hint: how can you represent any real number as an operation on two positive numbers?

Solution:

(a) $\max - \sum_i c_i x_i$

(b) $-x_1 \leq -b_1$

(c) $-x_1 \leq -b_1$ and $x_1 \leq b_2$

(d) $x_2 \leq b_2$ and $-x_2 \leq -b_2$.

(e) $x_1 + x_2 + x_3 \leq b_3$ and $-x_1 - x_2 - x_3 \leq -b_3$

(f) First, we represent $|x_1 + x_2| \leq b_2$ using linear constraints as follows:

$$x_1 + x_2 \leq b_2, -x_1 - x_2 \leq b_2$$

Then, we enforce the non-negativity of variables by substituting $x_1 = x_1^+ - x_1^-$ and $x_2 = x_2^+ - x_2^-$:

$$(x_1^+ - x_1^-) + (x_2^+ - x_2^-) \leq b_2$$
$$(x_1^+ - x_1^-) + (x_2^+ - x_2^-) \geq -b_2$$

(g) In order to enforce $|x_1 + x_2| \geq b_2$, we must use

$$(x_1 + x_2 \geq b_2) \cup (x_1 + x_2 \leq -b_2),$$

but an LP can only represent an intersection (not union) of constraints. In other words, it is impossible because we cannot have both $x_1 + x_2 \geq b_2$ and $x_1 + x_2 \leq -b_2$ hold at the same time (unless $b_2 = 0$).

(h) $\max -t\ , \quad x_1 \leq t\ , \quad x_2 \leq t$

(i) Replace x_4 with $x^+ - x^-$ along with $x^+ \geq 0,\ x^- \geq 0$

3 Baker

You are a baker who sells batches of brownies and cookies (unfortunately no brookies... for now). Each brownie batch takes 4 kilograms of chocolate and 2 eggs to make; each cookie batch takes 1 kilogram of chocolate and 3 eggs to make. You have 80 kilograms of chocolate and 90 eggs. You make a profit of 60 dollars per brownie batch you sell and 30 dollars per cookie batch you sell, and want to figure out how many batches of brownies and cookies to produce to maximize your profits.

(a) Formulate this problem as a linear programming problem; in other words, write a linear program (in canonical form) whose solution gives you the answer to this problem. Draw the feasible region, and find the solution using Simplex.

(b) Suppose instead that the profit per brownie batch is C dollars and the profit per cookie batch remains at 30 dollars. For each vertex you listed in the previous part, give the range of C values for which that vertex is the optimal solution.

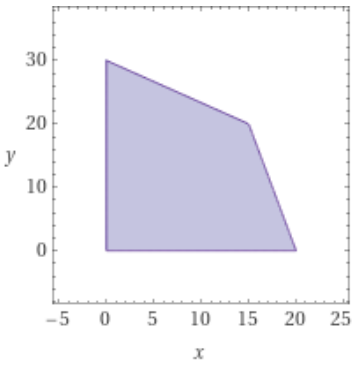
Solution:

(a) x = number of brownie batches
y = number of cookie batches

Maximize: $60x + 30y$

Linear Constraints:
$4x + y \leq 80$
$2x + 3y \leq 90$
$x \geq 0$
$y \geq 0$

The feasible region:



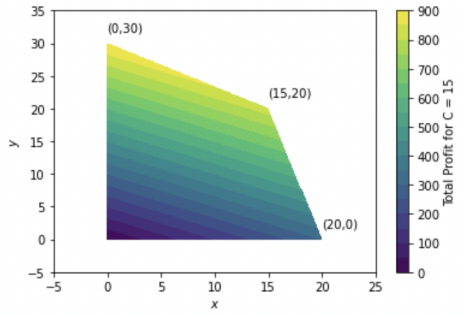
The vertices are $(x = 20, y = 0), (x = 15, y = 20), (x = 0, y = 30)$, and the objective is maximized at $(x = 15, y = 20)$, where $60x + 30y = 1500$. In other words, we can bake 15 batches of brownies and 20 batches of cookies to yield a maximum profit of 1500 dollars.

(b) There are lots of ways to solve this part. The most straightforward is to write and solve a system of inequalities checking when the objective of one vertex is at least as large as the objective of the other vertices. For example, for $(x = 15, y = 20)$ the system of inequalities would be $C \cdot 15 + 30 \cdot 20 \geq C \cdot 20$ and $C \cdot 15 + 30 \cdot 20 \geq 30 \cdot 30$. Doing this for each vertex gives the following solution:

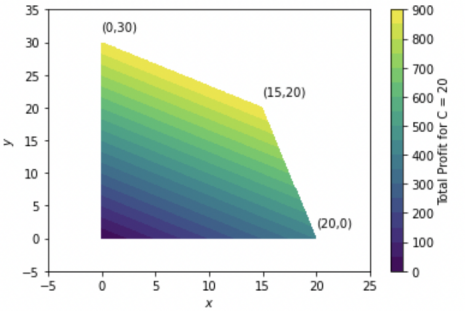
$(x = 0, y = 30) : C \leq 20$
$(x = 15, y = 20) : 20 \leq C \leq 120$
$(x = 20, y = 0) : 120 \leq C$

One should note that there is a nice geometric interpretation for this solution: Looking at the graph of the feasible region, as C increases, the vector $(C, 30)$ starts pointing closer to the x-axis. The objective says to find the point furthest in the direction of this vector, so the optimal solution also moves closer to the x-axis as C increases. When $C = 20$ or $C = 120$, the vector $(C, 30)$ is perpendicular to one of the constraints, and there are multiple optimal solutions all lying on that constraint, which are all equally far in the direction $(C, 30)$.

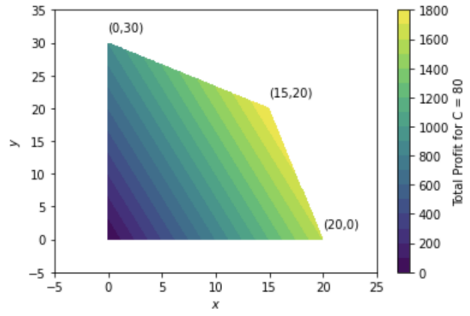
Below are some graphics to help build up your intuition about this problem.
C = 15 : (x = 0, y = 30) optimal



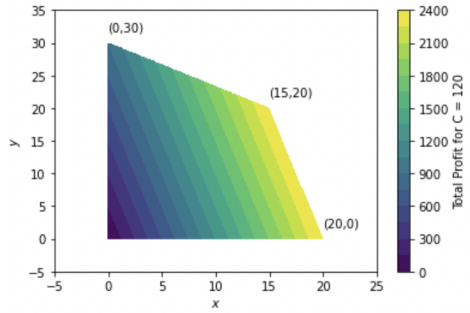
C = 20 : (x = 0, y = 30) and (x = 15, y = 20) optimal



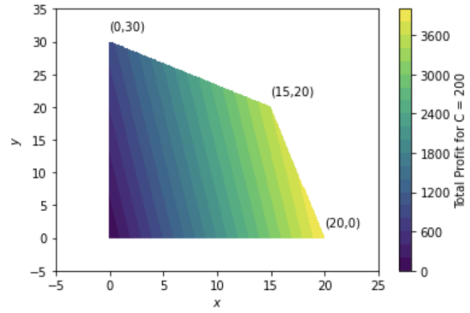
C = 80 : (x = 0, y = 30) optimal



C = 120: (x = 15, y = 20) and (x = 20, y = 0) optimal



C = 200 : (x = 20, y = 0) optimal



4 Meal Replacement

Jonny is planning an "Introduction to CS Theory" overnight summer camp for penguins in Antarctica. Penguins can't solve problems well when they're hungry, so Jonny wants to secure an emergency source of food in case polar bears sneak in and eat everything in the igloo. Unfortunately, he is on a tight budget, and in order to accommodate as many penguins as possible, he needs to minimize the cost of food while still meeting the penguins' minimum dietary needs.

Every penguin needs to consume at least 600 calories of protein per day, 800 calories of carbs per day, and 500 calories of fat per day. Jonny has three options for food he's considering buying: salmon, bread, and squid. The composition of each food is provided in the following table:

Food Type	**Price per Pound**	**Protein Calories per Pound**	**Carb Calories per Pound**	**Fat Calories per Pound**
Salmon	6	400	0	150
Bread	1	50	300	25
Squid	8	300	100	200

Our goal is to find a combination of these options that meets the penguins' daily dietary needs while being as cheap as possible.

(a) Formulate this problem as a linear program.

(b) Take the dual of your LP from part (a).

(c) Suppose now there is a pharmacist trying to assign a price to three pills, with the hopes of getting us to buy these pills instead of food. Each pill provides exactly one of protein, carbs, and fat.

Interpret the dual LP variables, objective, and constraints as an optimization problem from the pharmacist's perspective.

Solution:

(a) Let a be the pounds of salmon we consume, b be the pounds of bread, and c be the pounds of squid. The objective is to minimize cost, and we have a constraint for each of protein/carbs/fat.

$$\begin{aligned} \min\ & 6a + b + 8c \\ 400a + 50b + 300c &\geq 600 \\ 300b + 100c &\geq 800 \\ 150a + 25b + 200c &\geq 500 \\ a, b, c &\geq 0 \end{aligned}$$

(b) Let p, c, f be variables corresponding to the protein, carb, and fat constraints. The

dual is:

$$\begin{aligned} \max\ & 600p + 800c + 500f \\ 400p + 150f &\leq 6 \\ 50p + 300c + 25f &\leq 1 \\ 300p + 100c + 200f &\leq 8 \\ p, c, f &\geq 0 \end{aligned}$$

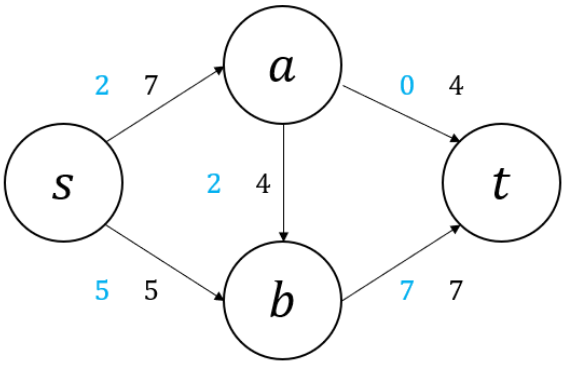
(c) The variables can be interpreted as the price per calorie for the protein, carb, and fat pills.

The objective says that the pharmacist wants to maximize the total revenue he gets from selling enough of these pills to us to meet our dietary needs.

The constraints say that no combination of pills should cost more than a pound of food providing the same dietary needs (otherwise, we would just buy that food instead of these pills).

5 Practice With Residual Graphs

(a) Consider the following network and flow on this network. An edge is labelled with its flow value (in blue) and capacity (in black). e.g. for the edge (s, a), we are currently pushing 2 units of flow on it, and it has capacity 7.



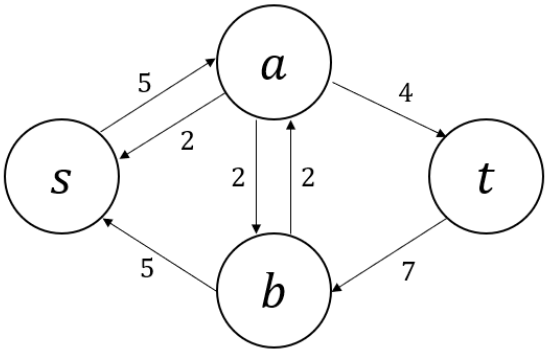
Draw the residual graph for this flow.

(b) We are given a network $G = (V, E)$ whose edges have integer capacities c_e, and a maximum flow f from node s to node t. Explicitly, f is given to us in the representation of integer flows along every edge e, $\{f_e\}_{e \in E}$.

However, we find out that one of the capacity values of G was wrong: for edge (u, v), we used c_{uv} whereas it really should have been $c_{uv} - 1$. This is unfortunate because the flow f uses that particular edge at full capacity: $f_{uv} = c_{uv}$. To fix this, we could run Ford Fulkerson from scratch, but there's a faster way.

Describe an algorithm to fix the max-flow for this network in $O(|V| + |E|)$ time. **Give a three-part solution.**

Solution:



(a)

(b) **Main Idea:** Since we know the flows along every edge, we can construct the residual graph in $O(|V| + |E|)$ time as there are $2|E|$ edges in the residual graph.

In this residual graph with the original capacity for (u, v), consider the set of edges E'

with positive flow, and let $G' = (V', E')$ be the subgraph induced by these edges. Try to find a find a simple path from v to u in G'. If such a path exists, we have a simple cycle involving the edge (u, v) and we can update f by pushing a 1 unit of flow in the reverse direction of the cycle.

Otherwise, there must exist *disjoint* paths p_u and p_v in G' from s to u and v to t, respectively. We use DFS to find p_u and p_v, and join these paths by adding the edge (u, v) in between them to get a path $p = p_u + (u, v) + p_v$. Update f by pushing 1 unit of flow from t to s on p (i.e. for each $(i, j) \in p$, reduce f_{ij} by 1, and increase f_{ji} by 1 for all edges except (v, u)). Finally, use DFS to see if the resulting residual graph has an $s - t$ path. If so, push 1 unit of flow on this path.

Proof of Correctness: Consider the residual graph of G and its induced positive-flow graph G'. If there exists a simple cycle in G' containing (u, v), then by the conservation of flow ($\sum_u f_{uv} = \sum_w f_{vw}$) we can safely push 1 flow along this cycle without modifying any flow properties.

Now, consider the case when there does not exist a simple cycle in G' containing (u, v). If (u, v) is at capacity, then there is a flow of at least 1 unit going from v to t, and since f_e are integral there is a path from v to t with at least one unit of flow moving through it. So the residual graph will have a path from t to v. Similarly, there will be a path from u to s in the residual graph which must be disjoint from v to t, so the algorithm can always find p correctly.

Note that the size of the maximum flow in the new network will be either the same or 1 less than the previous maximum flow (because changing one edge can change the capacity of the min-cut by at most 1).

In the former case, one possibility is that we find that there is a simple cycle in G' containing (u, v). Another possibility is when no such cycle exists, after pushing flow from t to s on p, it is possible to push more flow from s to t, so there must be an $s - t$ path in the new residual graph. Hence, the single iteration in Ford Fulkerson (i.e. one path augmentation) will find and push 1 unit of flow because all capacities and flow values are integral. Hence, our algorithm is correct.

The latter case occurs when there exists no simple cycle and the max flow is fully corrected after pushing flow backwards on p, so our algorithm is still correct.

Runtime: The runtime is $O(|V| + |E|)$ because the algorithm is just running DFS or iterating through graph vertices/edges a constant number of times:

(1) $O(|V| + |E|)$ time to construct the residual graph.

(2) $O(|V| + |E|)$ time to construct G'.

(3) $O(|V| + |E|)$ time to determine whether a simple cycle containing (u, v) exists in G'.

(4) If such a simple cycle exists, $O(|E|)$ time to push 1 unit of flow in the reverse direction along the cycle.

(5) If no such simple cycle exists: we need to run twice starting from s and v to find the paths $p_u = s \to u$ and $p_v = v \to t$; this takes $O(|V| + |E|)$ time total.

(6) $O(|E|)$ time to push 1 unit of flow in the backwards direction along $p = p_u + (u, v) + p_v$.

(7) $O(|V| + |E|)$ time to potentially augment a path.

6 (Extra Credit) Huffman and LP

Consider the following Huffman code for characters a, b, c, d: $a = 0, b = 10, c = 110, d = 111$.

Let f_a, f_b, f_c, f_d denote the fraction of characters in a file (only containing these characters) that are a, b, c, d respectively. Write a linear program with variables f_a, f_b, f_c, f_d to solve the following problem: What values of f_a, f_b, f_c, f_d (that can generate this Huffman code) result in the Huffman code using the most bits per character?

Solution:

Our objective is to maximize the bits per character used:

$$\max f_a + 2f_b + 3f_c + 3f_d$$

We know the fractions must add to 1 and be non-negative:

$$f_a + f_b + f_c + f_d = 1, f_a, f_b, f_c, f_d \geq 0$$

We know the frequencies of the characters must satisfy $f_a \geq f_b \geq f_c, f_d$. We also know that $f_c + f_d \leq f_a$, since we chose to merge (c, d) with b instead of merging a. So we get the following constraints:

$$f_c \leq f_b, f_d \leq f_b, f_b \leq f_a, f_c + f_d \leq f_a$$

We can write this LP in canonical form as follows:

$$\max\ f_a + 2f_b + 3f_c + 3f_d$$
$$\text{subject to} \begin{cases} f_a + f_b + f_c + f_d \leq 1 \\ -f_a - f_b - f_c - f_d \leq -1 \\ f_c - f_b \leq 0 \\ f_d - f_b \leq 0 \\ f_b - f_a \leq 0 \\ f_c + f_d - f_a \leq 0 \\ f_a, f_b, f_c, f_d \geq 0 \end{cases}$$

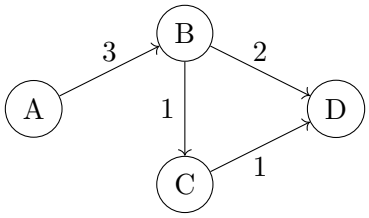
Note that for the second-to-last constraint, we write $f_c + f_d - f_a \leq 0$ instead of just $f_c + f_d \leq f_a$ because canonical form requires all the variables to be on the left, and only constants on the right.

7 (Extra Credit) Flow Decomposition

Let $G = (V, E)$ be a directed graph, and let f be any $s - t$ flow on this graph. Assume that there is no cycle in the graph where $f_e > 0$ for all edges in the cycle. Design an algorithm to decompose f into the sum of at most $|E|$ path flows. That is, your algorithm should find a set of $s - t$ paths $p_1 \dots p_k$ and corresponding flow values $F_1 \dots F_k$ such that:

- The number of paths k is at most $|E|$
- f is the sum of k path flows, where the ith flow sends F_i units of flow on path p_i. That is, for each edge e, if P_e is the set of paths in $p_1 \dots p_k$ that contain e, then $\sum_{p_i \in P_e} F_i = f_e$.

For example, in the below graph (where each edge e is labelled with f_e), one can decompose the flow into $p_1 = ((A, B), (B, C), (C, D)), p_2 = ((A, B), (B, D))$ where $F_1 = 1$, $F_2 = 2$.



Provide the algorithm description and a brief explanation of why your algorithm finds at most $|E|$ paths.

Hint: How does Ford-Fulkerson work?

Solution: While $|f| > 0$, do the following: Find an $s - t$ path p using only edges where $f_e > 0$. Let $F = \min_{e \in p} f_e$, i.e. set F to be the smallest amount of flow being sent through any edge on this path. Add p to the set of $s - t$ paths, with the corresponding flow value F, and decrease f_e by F for all edges e in p.

In each iteration, the number of edges where $f_e > 0$ decreases by 1. The algorithm ends when there are no such edges, and starts with at most $|E|$ such edges, so it must run for at most $|E|$ iterations and thus output at most $|E|$ paths.

8 Coding Questions: Max Flow/Min Cut

For this week's coding questions, we'll walk through the **Edmonds Karp** algorithm for max flow and see how to compute the **Minimum Cut** given a solution to the max flow problem. There are two ways that you can access the notebook and complete the problems:

1. **On Local Machine**: `git clone` (or if you already cloned it, `git pull`) from the coding homework repo,

   https://github.com/Berkeley-CS170/cs170-fa23-coding

   and navigate to the `hw09` folder. Refer to the `README.md` for local setup instructions.

2. **On Datahub**: Click here and navigate to the `hw09` folder if you prefer to complete this question on Berkeley DataHub.

Notes:

- *Submission Instructions:* Please download your completed submission `.zip` file and submit it to the Gradescope assignment titled "Homework 9 Coding Portion".
- *OH/HWP Instructions:* Designated coding course staff will provide conceptual and debugging help during office hours and homework parties.
- *Edstem Instructions:* Conceptual questions are always welcome on the public thread. If you need debugging help first try asking on the public threads. To ensure others can help you, make sure to:

  1. Describe the steps you've taken to debug the issue prior to posting on Ed.
  2. Describe the specific error you're running into.
  3. Include a few small test cases, alongside both the output you expected to receive and your function's actual output.

  If staff tells you to make a private Ed post, make sure to include *all of the above items* plus your full function implementation. If you don't provide them, we will ask you to provide them.

- *Academic Honesty Guideline:* We realize that code for some of the algorithms we ask you to implement may be readily available online, but we strongly encourage you to not directly copy code from these sources. Instead, try to refer to the resources mentioned in the notebook and come up with code yourself. That being said, we **do acknowledge** that there may not be many different ways to code up particular algorithms and that your solution may be similar to other solutions available online.

0.1 Network Flow: Edmonds-Karp

In [ ]:
```
# Install dependencies
!pip install -r requirements.txt --quiet
```

In [ ]:
```
import otter
assert (
    otter.__version__ >= "4.4.1"
), "Please reinstall the requirements and restart your kernel."

import numpy as np
import networkx as nx
import queue
import typing
import tqdm
import pickle
from collections import deque

grader = otter.Notebook("edmonds_karp.ipynb")
rng_seed = 42
```

In [ ]:
```
prng1 = np.random.default_rng(seed=rng_seed)

# Load test cases
file_path = "generated_testcases.pkl"

# Load the variables from the pickle file
with open(file_path, "rb") as file:
    loaded_data = pickle.load(file)
file.close()
inputs, outputs = loaded_data
```

In [ ]:
```
def rand_graph(size=None):
    """
    args:
        size:int = number of vertices in randomly generated graph.
    return:
        nx.DiGraph that represents the randomly generated directed graph.
    """
    if not size:
        g = nx.gnp_random_graph(prng1.integers(16, 64), 0.25, seed=rng_seed, directed=True)
    else:
        g = nx.gnp_random_graph(size, 0.25, seed=rng_seed, directed=True)

    for u, v in g.edges:
        g.edges[u, v]["capacity"] = prng1.integers(0, 50)
        g.edges[u, v]["flow"] = 0
    return g
```

Here, we will implement one well-defined implementation of Ford–Fulkerson algorithm: Edmonds–Karp algorithm.

What Ford–Fulkerson does can be summarised as 3 steps:

- Find an augmenting path in the graph
- Construct the residual graph by augmenting the path based on the original graph and augmenting path.
- Do this until no augmenting path can be found.

Ford-Fulkerson is not well-defined because there could be multiple ways of doing step 0: by DFS, by A* search, by some RL model that takes 200 RTX A6000 to train, or by magic……

If we use BFS to find the augmenting path, this algorithm is called Edmonds–Karp algorithm which is what we will implement today.

> Fun fact: Edmonds–Karp algorithm was first introduced by Soviet mathematician Yefim Dinitz/Dinic, yet Dinic's Algorithm is the name of a more complex and efficient algorithm for Network Flow problems (out of scope for CS170).

```
In [ ]: nx.draw_networkx(rand_graph(10))
```

0.1.1 Augmenting Path

First, you will implement the BFS portion of Edmonds–Karp algorithm. Given a graph and `s` and `t` vertices, find a augmenting path (a list of vertices from `s` to `t` **including** `s` and `t`) with BFS. If there's no path, return None.

0.1.2 Networkx tricks

There are some `networkx` tricks you might want to use below to make your life way easier:

To access flow from edge `(u, v)`, do `g.edges[u, v]["flow"]`; sub `flow` for `capacity` for capacity.

To access outgoing edges from `c` like adjacency list, do `g.edges(c)`. Note the returned type is an iterable of vertex tuples, so you should do something like:

```
for u, v in g.edges(c):
```

```
e = g.edges[u, v]
print(e["flow"])
```

To update some edge (u, v) to have flow of `f`, we can do either `g.add_edge(u, v, flow=f, capacity=c)` or `g.edges[u, v]["flow"] = f`, where `c` is the capacity of the edge - make sure you don't change edge capacities!

For the purpose of this coding HW, instead of decreasing `g.edges[u, v]["capacity"]` and check if it's still greater than zero each time you find or augment a path, please increase `g.edges[u, v]["flow"]` and check if it's still greater than `g.edges[u, v]["capacity"]`.

While `g.edges[u, v]` looks like adjacency matrix it is not - if `(u, v)` doesn't exist there would be `KeyError`. Later, when implementing `edmonds_karp()`, you can use `nx.algorithms.flow.build_residual_network` to prevent `KeyErrors`. `nx.algorithms.flow.build_residual_network` takes in an input graph and a key variable (for this assignment, the key will always be "capacity"), and returns a graph with 0-capacity edges placed wherever there is no edge in the input graph. For more information, see the documentation.

0.1.3 Q1) Find Augmenting Path

Task * Given a graph represented via a networkx graph, find the BFS augmenting path that has > 0 flow available. * Return a list of length equal to the number of nodes where the i-th element in this list is the i-th node on the s-t path. The first and last nodes should be s and t respectively. * If no such path is available return an empty list.

We recommend you use a `deque` for your BFS queue. You can read more about `deque` here.

Points: 3

```
In [31]: def bfs_augmenting_path(g, s, t):
             """
             args:
                 g:nx.DiGraph = directed network flow graph
                 s:int = source node s
                 t:int = sink node t
             return:
                 List[int] representing the augmenting path that BFS finds. First
                     and last nodes should be s and t respectively. Return None if
                     path doesn't exist. Ex: [s, a, b, t] if (s, a), (a, b), (b, t)
                     is the shortest length augmenting path.
             """
             queue = deque([(s, [s])])
             # BEGIN SOLUTION
             visited = set([s])
             while queue:
                 curr, path = queue.popleft()
                 if curr == t:
```

```
                return path
            for u, v in g.edges(curr):
                if v not in visited:
                    edge_flow_cap = g.edges[u, v]
                    if g.edges[u, v]["capacity"] > g.edges[u, v]["flow"]:
                        visited.add(v)
                        new_path = path[:]
                        new_path.append(v)
                        queue.append((v, new_path))
        return []
        # END SOLUTION
```

```
In [ ]: grader.check("q1")
```

0.1.4 Q2) Edmonds-Karp

Now, implement Edmonds-Karp with the functions you made above:

0. Find an augmenting path in the graph (with `bfs_augmenting_path`)
1. Construct the residual graph by augmenting the path based on the original graph and augmenting path (with `augment` described below)
2. Do this until no augmenting path can be found.

Return the capacity and the final residual graph; but the autograder will only check capacity. Do not modify G (you can do it by making a copy of G by invoking `g.copy()`).

> *Hint:* You may find it helpful to fill in the helper function `augment`: given an s-t path, determine the maximum amount of flow that can be pushed through this path (this is bottlenecked by the smallest capacity edge the augmenting path). Then, augment the path by decreasing available capacity (or increasing flow) for u-v and do the opposite for v-u. Finally, return the capacity of the flow you augmented.

```
In [20]: def edmonds_karp(g, s, t):
            """
            args:
                g:nx.DiGraph = directed network flow graph
                s:int = source node s
                t:int = sink node t
            return:
                Tuple(int, nx.DiGraph) where the first value represents the max flow that
                    was successfully pushed, and the second value represents the graph
                    where the flow values along each edge represents the flow pushed through
                    that edge.
            """
```

```
    graph = nx.algorithms.flow.build_residual_network(g.copy(), "capacity")
    nx.set_edge_attributes(graph, 0, "flow")
    flow = 0

    def augment(g, path):
        # BEGIN SOLUTION
        flow = float("inf")
        for i in range(len(path) - 1):
            e = g.edges[path[i], path[i + 1]]
            flow = min(flow, e["capacity"] - e["flow"])

        for i in range(len(path) - 1):
            g.edges[path[i], path[i + 1]]["flow"] += flow
            g.edges[path[i + 1], path[i]]["flow"] -= flow

        return flow
        # END SOLUTION

    # Use the augment function to compute the max flow
    # BEGIN SOLUTION
    while True:
        path = bfs_augmenting_path(graph, s, t)
        if not path:
            return flow, graph
        flow += augment(graph, path)
    # END SOLUTION
    return flow, graph
```

```
In [ ]: grader.check("q2")
```

0.1.5 Q3) Min-Cut

Recall that the dual of the Max-Flow problem is the Min-Cut problem. In the Min-Cut problem, we are given a network, and we wish to find a vertex cut of this graph, (S, T) where $S \cup T = V$, $S \cap T = \emptyset$, $s \in S, t \in T$ such that the sum of the capacities of the edges spanning this cut is minimized.

Here, we'll see how, given a solution to the max-flow problem on a graph, we can easily find the min cut. Implement `find_min_cut` to find the minimum S-T cut of this graph (which also yields the edges with saturated flow). Make sure to return all the vertices that belong to the S partition of the cut.

```
In [23]: def find_min_cut(g, s, t):
             """
             args:
                 g:nx.DiGraph = directed network flow graph
                 s:int = source node s
                 t:int = sink node t
             return:
                 Set(int) that contains all vertices in the min cut of g that includes s.
```

```
    """
    vertex_cut_s = set([s])
    flow, graph = edmonds_karp(g, s, t)
    # BEGIN SOLUTION
    q = [s]
    while q:
        curr = q.pop()
        for (u, v) in graph.edges:
            if v not in vertex_cut_s and graph.edges[u, v]["flow"] > 0:
                vertex_cut_s.add(v)
                q.append(v)
    # END SOLUTION
    return vertex_cut_s
```

In [ ]:
```
grader.check("q3")
```

0.2 Submission

Make sure you have run all cells in your notebook in order before running the cell below, so that all images/graphs appear in the output. The cell below will generate a zip file for you to submit.

In [ ]:
```
grader.export(pdf=False, force_save=True, run_tests=True)
```