Flow Networks

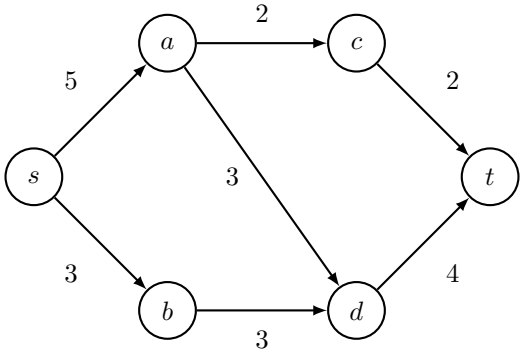
Aaron Mills

February 15, 2023

This document is a summary of core concepts related to flow networks. I am not taking credit for inventing any of the concepts conveyed. I am simply collecting information from other sources and presenting it here. For more information on the sources used, please see the references section of this document. Please let me know if you find any mistakes so I can amend them.

1 Introduction

A flow network is a directed graph where each edge has an associated capacity, representing the maximum amount of *flow* that can be assigned to that edge. In addition, a flow network has two special nodes: a source denoted s, and a sink denoted t. The source node has zero incoming edges and can be considered the node that generates the flow for the remainder of the graph. The sink has zero outgoing edges and can be thought of as absorbing the flow of the network. We will assume nonnegative integral capacities for the flow networks discussed in this document. An example of a flow network is shown below.



2 The Maximum Flow Problem

Let's now work toward formally defining the maximum flow problem. Let $G = (V, E)$, source s, and sink t, be a flow network, and let c_e denote the capacity

of edge $e \in E$. Let the *flow* of our network be a function that maps the set of edges to the nonnegative real numbers, $f : E \longrightarrow \mathbb{R}^+$. Let f_e denote the flow on edge $e \in E$ (i.e. $f_e = f(e)$). The flow must satisfy the following two constraints:

1. Capacity Constraint: $0 \leq f_e \leq c_e, \ \forall e \in E$
2. Flow Conservation Constraint: $\sum_{v \in V} f_{vu_0} = \sum_{w \in V} f_{u_0 w}, \forall u_0 \in V$

The capacity constraint states that the flow on a particular edge must be greater than or equal to zero, yet it must be lesser than or equal to its assigned integral capacity. The flow conservation constraint states that the total flow leaving a particular vertex must be equal to the total flow entering the vertex. The source s and sink t are the only two vertices that do not abide by the flow conservation constraint [1, 2].

We define the *value of the flow* of a network to be the total amount of flow that is generated by the source s. Let us denote this quantity by $v(f)$. Then:

$$v(f) = \sum_{u \in V} f_{su} \tag{1}$$

To simplify notation, we define the following for any vertex $u \in V$:

$$f^{out}(u) = \sum_{v \in V} f_{uv} \tag{2}$$

$$f^{in}(u) = \sum_{v \in V} f_{vu} \tag{3}$$

Then, using our new notation, we have: $v(f) = f^{out}(s)$, and the flow conservation constraint can be written as: $f^{in}(u) = f^{out}(u), \forall u \in V$. Naturally, we want to determine a configuration for the flow such that the value of the flow is maximal. This is the maximum flow problem.

3 The Ford-Fulkerson Algorithm

The *Ford-Fulkerson algorithm* is a well-known algorithm that solves the maximum flow problem with the solution having integral flows. An important component of the algorithm that must first be developed is the concept of a *residual graph*.

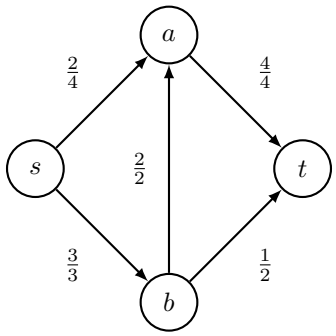
3.1 The Residual Graph

Given a flow network $G = (V, E)$, we define the residual graph G_f to be a flow network with the same set of vertices, V, but a different set of edges, E_f. These edges have what's called *residual capacities*. The residual graph is constructed as follows:

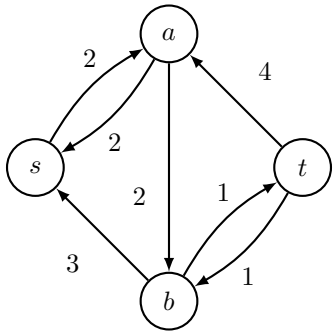
- For each vertex $u \in V \cup \{s, t\}$, we add the same vertex to the residual graph, G_f.

- For each edge $uv \in E$ such that $f_{uv} < c_{uv}$ we add an edge from u to v in G_f with residual capacity $c_{uv} - f_{uv}$. These are called *forward edges*.
- For each edge $uv \in E$ such that $0 < f_{uv}$ we add an edge from v to u in G_f with residual capacity f_{uv}. These are called *backward edges*.

Let us denote the residual capacity of edge $e \in E$ in the residual graph as c'_e. Realize that the maximum number of edges in the residual graph is twice the amount in the original flow network (i.e. $|E_f| \leq 2|E|$).



Consider the above flow network. Each edge has a fraction, with the denominator being the capacity of the edge and the numerator being the flow on that edge. The flow presented above is not maximal, and if we try to improve it with the naïve approach of finding a path from s to t (called an *s-t path*) such that the flow can be increased on the edges, it becomes clear that we can no longer find any such path because of the existing flow. The residual graph allows for the concept of *pushing flow backward* using the backward edges in G_f. The residual graph for the flow network above is shown below [2, 4].



3.2 Augmenting Paths

Now that we have defined and shown an example of a residual graph, we need to discuss how we can use it to modify the flow of our network systematically such that we converge to the maximum flow. As hinted at earlier, the core of the Ford-Fulkerson algorithm is *s-t* paths. However, these paths are on the residual

graph rather than the original network. An *augmenting path* in our residual graph is an s-t path such that the residual capacity is nonzero for each edge in the path. Let us denote such a path in G_f by P. We define the bottleneck of P given the current flow configuration f as the minimum residual capacity among all the edges in P. We denote this bottleneck by $\beta_{P,f} = min(c'_e), \forall e \in P$. In our residual graph, we can now define the following procedure for *augmenting a path*, P.

```
procedure AUGMENT(P, f)
    Let β_{P,f} = min(c'_e), ∀e ∈ P
    for e ∈ P do
        if e is a forward edge then
            f_e ← f_e + β_{P,f}
        else                                    ▷ e is a backward edge
            f_e ← f_e − β_{P,f}
        end if
    end for
    return f
end procedure
```

The fact that the updated flow is valid can be verified by ensuring that the capacity and flow conservation constraints hold. Still, does the new flow returned by the *augment* procedure, f', have a greater value than the existing flow in the network G? Indeed, it does. This becomes clear when we consider that each augmenting path P must contain a single forward edge that leaves the source s. Therefore: $v(f') = v(f) + \beta_{P,f} > v(f)$. We can use a strict inequality here as each s-t path P in G_f must have a nonzero residual capacity, meaning $\beta_{P,f} > 0$. After obtaining an improved flow, f', it becomes the new flow for our flow network, f [2].

3.3 Constructing the Algorithm

We have now determined a way in which we can increase the flow of our network. We find an augmenting path in G_f, update the flow in G, reconstruct G_f given the new flow, and repeat. We must determine whether this algorithm will eventually terminate or run forever. Consider that from the capacity constraint on the edges leaving the source s, we can derive an upper bound on the value of the flow, which we denote C.

$$\begin{aligned} & f_{su} \leq c_{su}, \forall u \in V \\ \implies \quad & v(f) = \sum_{u \in V} f_{su} \leq \sum_{u \in V} c_{su} = C \end{aligned} \tag{4}$$

Each time we augment the flow, the value increases by the bottleneck, $\beta_{P,f}$, which is positive. Furthermore, the values of $\beta_{P,f}$ are residual capacities that are integral (this becomes clear when you consider that the flow values, f_e, are

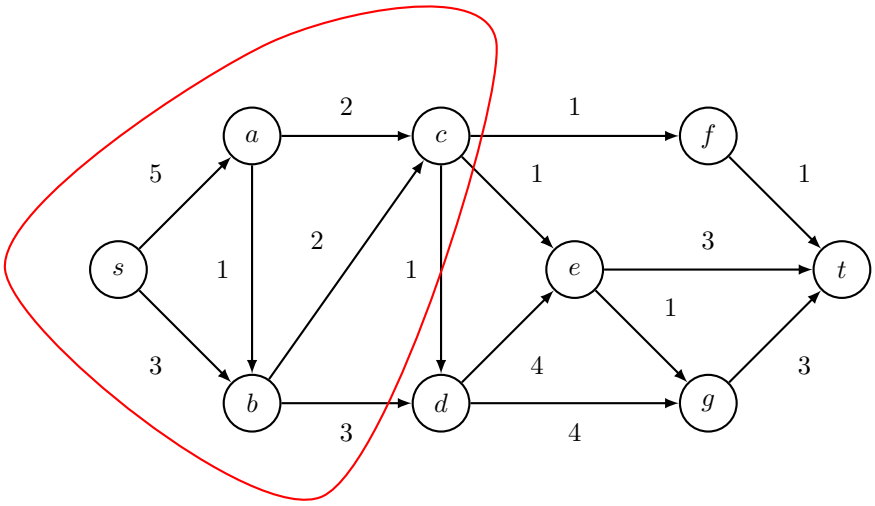
initialized to zero), so $\beta_{P,f}$ is an integer greater than or equal to 1. Therefore, the algorithm we derived above will take C iterations to compute the maximum flow in the worst case. This algorithm is the Ford-Fulkerson algorithm, and its pseudo-code is shown below.

```
procedure FORD-FULKERSON(G)
    Let f_e ← 0, ∀e ∈ E
    From f, construct G_f
    while There exists and s-t path, P, in G_f do
        f' ← augment(P, f)
        f ← f'
        Reconstruct G_f
    end while
    return v(f)
end procedure
```

Given what we have discussed above about the value of the flow, $v(f)$, being bounded from above by the total capacity of the edges leaving the source, C, and the flow value increasing by at least 1 for each augmentation of the flow, we can conclude that we have at most C iterations of the loop in Ford-Fulkerson. The most expensive part of each iteration is finding an s-t path in the residual graph G_f. This can be done with a graph search algorithm such as breadth-first or depth-first, which can be done in $O(|V|+|E|)$ running time [5]. Therefore the Ford-Fulkerson algorithm has a running time of $O(C(|V|+|E|))$, which at first glance may seem like a polynomial running time. However, C is an unbounded constant, so this is not polynomial and, therefore, not considered efficient.

4 The Minimum Cut Problem

Another interesting problem related to the maximum flow problem is the minimum cut problem. An *s-t cut* of a flow network is a disjoint partition of the vertices V such that the source s is in one partition and the sink t is in the other partition. Let (A, B) be an s-t cut with $s \in A$ and $t \in B$. The *capacity of the cut*, denoted $cap(A, B)$, is defined to be the total capacity of all the edges that cross from A to B, meaning each edge that contributes to the cut leaves a vertex in A and enters a vertex in B. Vertices starting in B and ending in A are not considered. An example of an s-t cut is shown below. The set A is encompassed in the red loop.



Let (A, B) be an s-t cut. Let $f^{out}(A)$ denote the total flow leaving the set A and $f^{in}(A)$ be the total flow entering the set. Then:

$$v(f) = f^{out}(A) - f^{in}(A) \tag{5}$$

This intuitively makes sense. All of the flow that is leaving set A minus all of the flow that is entering set A is equal to the total flow generated by the source s, which is the value of the flow. A more important consequence of this statement is the following:

$$\begin{aligned} v(f) =& f^{out}(A) - f^{in}(A) \\ &\leq f^{out}(A) \\ &= \sum_{uv \in E | u \in A, v \in B} f_{uv} \\ &\leq \sum_{uv \in E | u \in A, v \in B} c_{uv} \\ &= cap(A, B) \end{aligned} \tag{6}$$

Therefore, the value of any flow is lesser than or equal to the capacity of any cut [2]. The minimum cut problem involves determining the cut (A, B) such that the capacity of the cut is minimal.

4.1 Max-Flow Min-Cut Theorem

We have already established that the capacity of any cut in a flow network $G = (V, E)$ is an upper bound to the value of any flow. This fact implies that if we can find a cut and a flow such that the capacity of the cut equals the value of

the flow, then we have the minimum capacity cut (min-cut) and the maximum value of the flow (max-flow). Consider the following theorem:

Theorem (Max-Flow Min-Cut) Let f be a flow for a flow network $G = (V, E)$. The following are equivalent:

1. $v(f)$ is maximal with respect to any valid flow for G.
2. The residual graph G_f has no s-t path.
3. There exists a cut (A, B) such that $cap(A, B) = v(f)$

The max-flow min-cut theorem effectively states that when the Ford-Fulkerson algorithm terminates (i.e., no augmenting path in G_f), the flow returned is the optimal flow. In addition, a cut (the minimal cut) exists such that its capacity is equal to the value of the maximum flow [4, 6].

4.2 Finding the Minimum Cut

We have yet to discuss how we can actually find a min-cut in a flow network. Given a flow network with an optimal flow configuration obtained by Ford-Fulkerson, a minimum cut is formed by partitioning the vertices based on those reachable from the source s in the residual graph G_f. This results in a straightforward algorithm to compute a min-cut. First, run Ford-Fulkerson until termination. Then run a graph search algorithm on the resulting G_f to obtain the set of reachable vertices from s (including s), A^*. The other set of the disjoint partition is $B^* = V \cup \{\mathrm{t}\} \setminus A^*$. Let (A^*, B^*) be a min-cut. The edges that cross the cut from A^* to B^* are *saturated* (i.e., $f_e = c_e$), and the edges from B^* to A^* have a flow of zero. The first claim is easy to justify by contradiction. Suppose that one of the edges, $uv \in E$ where $u \in A^*$ and $v \in B^*$, is not saturated. Then, in the residual graph G_f, there exists a forward edge (with nonzero residual capacity) from u to v, so when a search algorithm is run to determine the partition, v will be in A^*. This is a contradiction, as v is defined to be in B^*. Similarly, suppose there exists an edge $vu \in E$ where $v \in B^*$ and $u \in A^*$ such that it has nonzero residual capacity. Then, a backward edge exists from u to v in G_f with nonzero residual capacity, which again leads to a contradiction [2].

5 Faster Maximum Flow Algorithms

Earlier, we established that the running time of the Ford-Fulkerson algorithm is $O(C(|V| + |E|))$ where $|V|$ is the number of the vertices, $|E|$ is the number of edges, and C is the total capacity of the edges leaving the source s. This algorithm is not polynomial due to the unbounded constant C, represented by $k = O(log(C))$ bits. Our running time is then $O(2^k(|V| + |E|))$, which is exponential in the number of bits required to store our input, so naturally, we wish to find an efficient algorithm (i.e., polynomial with respect to the input). Consider that Ford-Fulkerson is inefficient due to the lack of guarantee of large

improvement to the flow each time we augment the path. Since we only have a guarantee of improvement by a value of 1, a very large value of C results in a slow algorithm.

5.1 Scaling Ford-Fulkerson

To improve the speed of the Ford-Fulkerson algorithm, we will attempt to be more strategic with our choice of augmenting path in G_f. Ideally, we would like to take the greedy approach and always select the path that will allow us to augment the flow maximally. However, it is far too inefficient to do so (we would need to compute the bottlenecks of every s-t path each iteration). The scaling Ford-Fulkerson algorithm is an approach that allows us to select paths with larger bottlenecks and provides better efficiency guarantees than regular Ford-Fulkerson.

To select better augmenting paths, we will use a *scaling parameter*, denoted Δ. The residual graph that we will use in our Ford-Fulkerson algorithm differs depending on the current value of Δ. Let $G_f(\Delta)$ denote a version of the residual graph of G where the edges included are only those whose residual capacities are greater than or equal to Δ. Since every edge in this residual graph will have a residual capacity of at minimum Δ, we know that if we find an s-t path in $G_f(\Delta)$, the bottleneck is at least Δ, and therefore we improve the flow f by at least Δ. If there does not exist an augmenting path in $G_f(\Delta)$, we will divide the scaling factor in half and reconstruct the new residual graph with the smaller Δ. We terminate the algorithm if we do not find an augmenting path when $\Delta = 1$. To ensure that we have clean divisions of the scaling factor by 2, we will initialize Δ to be the largest power of two that is lesser than the maximum capacity of any edge leaving s.

---

procedure SCALING FORD-FULKERSON(G)
  Let $f_e \leftarrow 0$, $\forall e \in E$
  Let Δ be the largest power of 2 lesser than the maximal capacity.
  From f, construct $G_f(\Delta)$
  while $\Delta \geq 0$ **do**
    while There exists and s-t path, P, in G_f **do**
      $f' \leftarrow augment(P, f)$
      $f \leftarrow f'$
      Reconstruct $G_f(\Delta)$
    end while
    $\Delta \leftarrow \Delta/2$
  end while
  return $v(f)$
end procedure

Let us now analyze the performance of this algorithm. First, consider that the number of iterations of the outer loop is $O(log(\Delta))$. To put it in terms of the

constant $C = \sum_{u \in V} c_{su}$ used earlier, we can say that the number of iterations of the outer loop is $O(log(C))$ as $\Delta \leq C$.

We need to determine how many iterations of the inner loop can occur for a fixed Δ. Assume there are no more augmenting paths for the current value of Δ in $G_f(\Delta)$. Let (A, B) be an s-t cut. Then, for each edge $e \in E$ that crosses the cut from A to B, we have: $c_e < f_e + \Delta$. This must be true because otherwise, e would form a forward edge in the $G_f(\Delta)$ with a residual capacity of $c_e - f_e \geq \Delta$, meaning both endpoints of e would be reachable from s, and therefore e would not cross the cut. Similarly, for each edge, $e \in E$ that crosses from B to A, we have that $f_e < \Delta$, as otherwise, e would form a backward edge in $G_f(\Delta)$ with a residual capacity greater than or equal to Δ. Once again, in this case, both endpoints would be reachable from s in $G_f(\Delta)$. Either of these cases contradicts the assumption that e contributes to the cut. Recall equation 5 from earlier:

$$\begin{aligned} v(f) &= f^{out}(A) - f^{in}(A) \\ &= \sum_{uv \in E | u \in A, v \in B} f_{uv} - \sum_{vu \in E | v \in B, u \in A} f_{vu} \\ &\geq \sum_{uv \in E | u \in A, v \in B} (c_{uv} - \Delta) - \sum_{vu \in E | v \in B, u \in A} \Delta \\ &= \sum_{uv \in E | u \in A, v \in B} c_{uv} - \sum_{uv \in E | u \in A, v \in B} \Delta - \sum_{vu \in E | v \in B, u \in A} \Delta \\ &\geq cap(A, B) - \Delta |E| \\ \implies & v(f) + \Delta |E| \geq cap(A, B) \end{aligned} \tag{7}$$

The last inequality follows from the fact that the number of edges leaving A plus the number of edges entering A must be lesser than or equal to the total number of edges $|E|$. The final statement above tells us that after we have augmented the flow using $G_f(\Delta)$ until no augmenting paths remain, the value of the flow will be at least within $\Delta|E|$ of the maximum flow, as the capacity of any cut is an upper bound for the value of any flow, including the maximal flow. The next step in our algorithm is to divide Δ in half (i.e., $\Delta \leftarrow \Delta/2$), so then: $v(f) + 2\Delta|E| \geq cap(A, B)$ with our newly constructed $G_f(\Delta)$. Consider that each augmentation increases the flow by at least Δ, so the inner loop can have at most $2|E|$ iterations [2, 7].

We now have all of the pieces to form a performance bound for the scaling Ford-Fulkerson algorithm. The outer loop runs in $O(log(C))$ iterations, and the inner loop runs for $O(|E|)$ iterations. As discussed for the regular Ford-Fulkerson algorithm, each augmentation is $O(|V| + |E|)$. Together, we have a total running time of $O(log(C)|E|(|V| + |E|))$, which seems rather messy but is, in fact, a polynomial running time. Just as before, our constant C is represented by $k = O(log(C))$ bits which implies $C = O(2^k)$. Therefore our running time for scaling Ford-Fulkerson is $O(k|E|(|V| + |E|))$, which is polynomial with respect to the input size.

5.2 Edmonds-Karp Algorithm

The Edmonds-Karp algorithm makes a slight change to the original Ford-Fulkerson algorithm to obtain an algorithm that is polynomial in the number of edges and vertices. When selecting an augmenting path in G_f, the Edmonds-Karp algorithm chooses the shortest path with respect to the number of edges.

---

procedure EDMONDS-KARP(G)
  Let $f_e \leftarrow 0$, $\forall e \in E$
  From f, construct G_f
  while There exists and s-t path, in G_f **do**
    Let P be the shortest s-t path in G_f
    $f' \leftarrow augment(P, f)$
    $f \leftarrow f'$
    Reconstruct G_f
  end while
  return $v(f)$
end procedure

Just like with Ford-Fulkerson, each iteration of the loop has an $O(|V| + |E|)$ running time. Finding the shortest s-t path is itself $O(|V| + |E|)$ using breadth-first search. The number of augmentations done by Edmonds-Karp, and thus the number of iterations of the loop, is $O(|V||E|)$.

Let $d_f(u, v)$ denote the shortest distance from node $u \in V$ to node $v \in V$ in G_f. Consider that when augmenting paths, any edge in the selected s-t path whose residual capacity is equal to the bottleneck will be removed from G_f. Also, consider that for each iteration, the size of the path selected is greater than or equal to the size of the path in the previous iteration (i.e., a shorter path is not created by augmenting the flow). Let uv be an edge that was removed from G_f after a path was selected such that its residual capacity was the bottleneck. Before being removed, it is clear that $d_f(s, v) = d_f(s, u) + 1$. The only way that uv can resurface in the residual graph is if the edge vu ends up on an augmenting path. Suppose we have flow f' when this occurs. Now, $d_{f'}(s, u) = d_{f'}(s, v) + 1 \implies d_{f'}(s, v) = d_{f'}(s, u) - 1$. Then, since $d_{f'}(s, v) \leq d_f(s, v)$ by the fact that the shortest distances are always increasing, we have that $d_f(s, u) + 2 \leq d_{f'}(s, u)$. The furthest distance that u can be from s is $|V|$, so if the distance from s increases by at least 2 each time u is removed from the graph, it can be removed at most $\frac{|V|}{2}$ times. Since there are $O(|E|)$ vertices with an edge between them, the execution time of an iteration of the loop is $O(|V||E|)$. Therefore, in total, we have an execution time of $O(|V||E|(|V| + |E|))$, which is polynomial with respect to the input [5, 8].

5.3 Linear Programming

Linear programs are a class of optimization problems where the function that is being optimized is linear, and the set of feasible solutions is determined by sev-

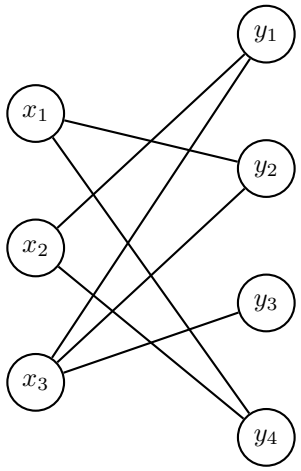
eral linear constraints. What makes linear programs interesting in the context of the maximum flow problem is that they can be solved in polynomial time. Let $G = (V, E)$ be a flow network with source s and sink t. The maximum flow problem can be modeled as the following linear program:

$$\begin{aligned} \text{maximize} \quad & \sum_{su \in E} f_{su} \\ \text{subject to} \quad & \sum_{u_0 v \in E} f_{u_0 v} - \sum_{w u_0 \in E} f_{w u_0} = 0, \forall u_0 \in V \\ & f_e \leq c_e, \forall e \in E \\ & f_e \geq 0, \forall e \in E \end{aligned} \tag{8}$$

The *objective function* of this optimization problem that we are maximizing is the total flow leaving the source s, which is equal to the value of the flow, $v(f)$. The first constraint directly models the *flow conservation constraint*, and the lower two constraints combined model the *capacity constraint*. An interesting note about solving the maximum flow problem with linear programming rather than with Ford-Fulkerson is that there is no guarantee that the optimal flow configuration to be composed of integral flows. Ford-Fulkerson guarantees integral flows, but linear programming does not.

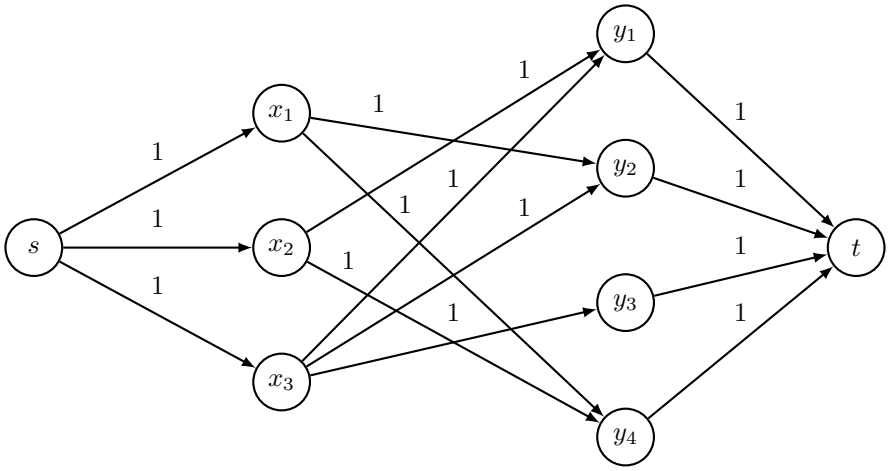
6 Bipartite Matching

A *bipartite graph* is a graph $G = (V, E)$ whose vertices can be partitioned into two disjoint sets X, Y where all of the edges $uv \in E$ are such that $u \in X$ and $v \in Y$. Note that, unlike a flow network, the edges are not directed. An example of a bipartite graph is shown below.



A *matching* is a subset $M \subseteq E$ such that each node in G appears as an endpoint to at most one edge. The goal of the *bipartite matching problem* is

to find the largest possible matching given a bipartite graph, and we will do so using our ability to solve the maximum flow problem. To model the maximum matching problem for the above graph as a flow network problem, we create the following flow network with the proposed capacities.



WLOG, the edges are directed from the set X to the set Y with the source connected to the vertices in X and the vertices of Y connected to the sink. The flow network could be set up such that the roles of the vertices in X and Y are swapped. A capacity of 1 is put on each of the edges corresponding to the edges in the original bipartite graph. If a flow of 1 is placed on one of these edges, it corresponds to that edge being part of the maximal matching. The source s has an edge with each of the vertices in X. Each of these edges has a capacity of 1 as well. This models the idea that each of the vertices in X can be matched to at most one vertex in Y and thus can pass at most a flow of 1. Similarly, each of the vertices in Y has an edge with the sink t with a capacity of 1. This models the idea that each of the vertices in Y can be an endpoint to at most 1 edge coming from the vertices in X. We will use Ford-Fulkerson to find the optimal flow configuration and then select the edges uv such that $u \in X$ and $v \in Y$ have a nonzero flow value to be our maximal matching. Since the Ford-Fulkerson algorithm and its variants, all guarantee integral flow values, we can be confident that each vertex in X connects to at most one node in Y. Furthermore, this solution can be extended to solve the *perfect matching problem*, where a *perfect matching* is a matching such that each vertex appears as an endpoint to an edge in the matching exactly once. If the maximum matching obtained using the above algorithm is such that each vertex is an endpoint of exactly one edge, then it is also a perfect matching. Note that since the capacities in the flow network are all 1, the regular Ford-Fulkerson algorithm can find the maximal matching in $O(|V| + |E|)$ running time, which is polynomial [2, 5].

7 Circulation With Demands

We will now explore another problem that can be solved by reducing it to a flow network problem. Let $G = (V, E)$ be a directed graph with capacities $c_e, \forall e \in E$, and additionally integral *demand values* $d_u, \forall u \in V$. These demand values generalize the concept of a flow network to a directed graph with many sources and sinks. The demand values represent the required difference between the flow entering a vertex and the flow exiting a vertex.

$$d_u = \sum_{w \in V} f_{wu} - \sum_{v \in V} f_{uv} = f^{in}(v) - f^{out}(v) \tag{9}$$

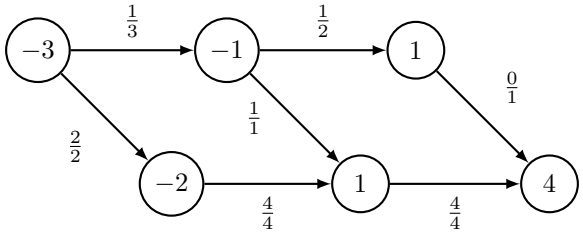
When the demand is negative, it is referred to as *supply* since the amount of flow exiting the vertex is greater than the amount that enters. These vertices generate flow for the network. Similarly, when the demand is positive, the amount of flow entering the vertex is greater than the amount that exits. These vertices absorb flow from the network.

7.1 The Circulation Problem

The problem we want to solve, referred to as the *circulation problem*, is as follows. Given a directed graph $G = (V, E)$, capacities for the edges $c_e, \forall e \in E$, and demand values for the vertices $d_u, \forall u \in V$, determine if there exists a *feasible circulation*. A *circulation* is similar to the concept of flow in the maximum flow problem. Its a function $f : E \longrightarrow \mathbb{R}^+$ such that the following two constraints hold:

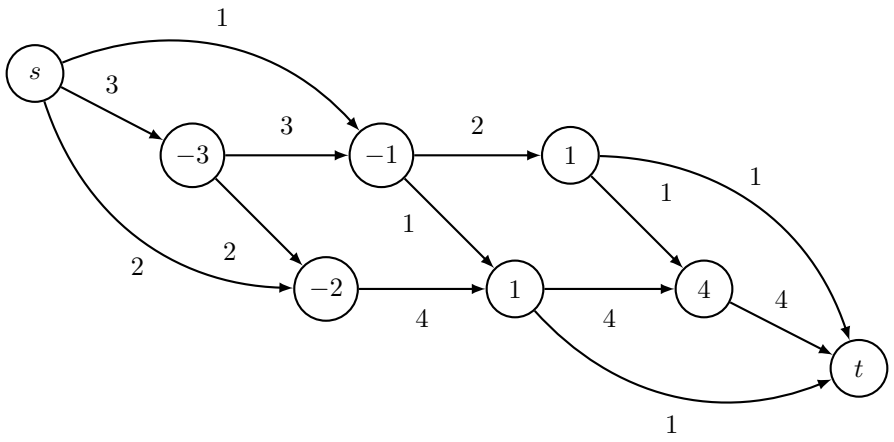
1. Capacity Constraint: $0 \leq f_e \leq c_e, \ \forall e \in E$
2. Demand Constraint: $d_u = \sum_{v \in V} f_{uv} - \sum_{w \in V} f_{wu}, \forall u \in V$

Similar to valid flows, we require the circulation to abide by the capacity constraint. However, instead of the flow conservation constraint, we have the demand constraint. An example of the circulation problem with a feasible circulation is shown below.



As mentioned earlier, we will solve this problem by reducing it to a flow network problem and solving the maximum flow problem. To do this, we introduce a source, s, and a sink, t, where the source provides flow to all the

vertices with negative demand, and the sink absorbs flow from all the vertices that have positive demand. To create our flow network, we connect s to each vertex $u \in V$ such that $d_u < 0$ using an edge with a capacity equal to $|d_u|$. Similarly, we connect each vertex $v \in V$ such that $d_v > 0$ to t using an edge with a capacity of d_v. The resulting flow network for the circulation problem shown above is shown below.



We now use one of our various techniques for finding the maximum flow. Let the total demand of the network, denoted D, be the sum of all the positive demands in the network. This must also be the total magnitude of supply for circulation to be possible.

$$D = \sum_{u \in V | d_u > 0} d_u = \sum_{u \in V | d_u < 0} |d_u| \tag{10}$$

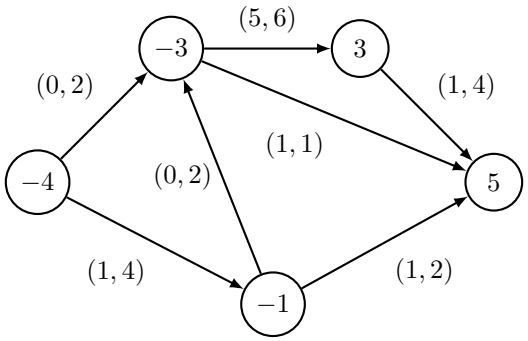
We have feasible circulation for our circulation problem if and only if the maximum flow value is equal to the total demand D [2, 9].

7.2 Circulation with Lower Bounds

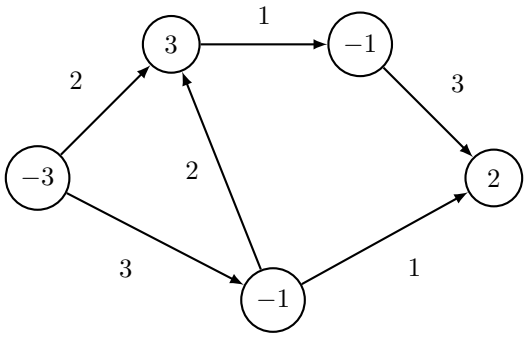
We will now further extend the circulation problem by introducing the concept of lower bounds on the flow. Previously, each edge only had an associated capacity, c_e, but now, each edge will also have an associated lower bound ℓ_e. The circulation problem is now to determine if there exists a feasible circulation $f : E \longrightarrow \mathbb{R}^+$ subject to the following two constraints:

1. Capacity Constraint: $\ell_e \leq f_e \leq c_e$, $\forall e \in E$
2. Demand Constraint: $d_u = \sum_{v \in V} f_{uv} - \sum_{w \in V} f_{wu}, \forall u \in V$

In our graph, the lower bound and capacity associated with each edge are denoted by a tuple (ℓ_e, c_e).



To solve this problem, we convert it to one of standard circulation by constructing a new graph G'. Consider that we must provide as least as much flow on each edge as its lower bound value. For each vertex, $u \in V$, let $L^{in}(u)$ denote the sum of the lower bounds on all of the edges incoming to u, and $L^{out}(u)$ denote the sum of the lower bounds on the edges outgoing from u. Since at least a flow of $L^{in}(u)$ in entering u, we can adjust its demand value by subtracting $L^{in}(u)$ and then remove the lower bounds on the incoming edges as they are assumed satisfied. Similarly, since a flow of at least $L^{out}(u)$ is exiting u, we will further adjust the demand by adding $L^{out}(u)$. So the adjusted demand value is $d'_u = d_u - L^{in}(u) + L^{out}(u)$. After adjusting the demand values, we must also adjust the capacities. Since we are already sending a flow of ℓ_e across each edge e, its capacity in G' should become $c'_e = c_e - \ell_e$.



We then solve the circulation problem by forming a flow network, finding the maximal flow, and confirming it equals the total demand. Let these flows obtained from the optimal flow configuration in G' be denoted f'_e, and let $f'_e = 0$ if e does not appear in the second graph. We then convert this result back to the original graph by setting the flow $f_e, \forall e \in E$ as follows:

$$f_e = \ell_e + f'_e \tag{11}$$

It's quite clear that the extended problem can be reduced to a standard circulation problem in polynomial time. The running time for solving the circulation problem is limited by the running time of determining the maximum

flow, which is also polynomial. Therefore we can be confident that we have efficient algorithms to solve the circulation problem with and without lower bounds [2, 9].

8 Applications

We have already shown that we can use our ability to solve the maximum flow problem to solve problems such as the maximum bipartite matching problem and feasible circulation problem. In this final section, we will review other problems that can be solved using max-flow and outline how we can do so.

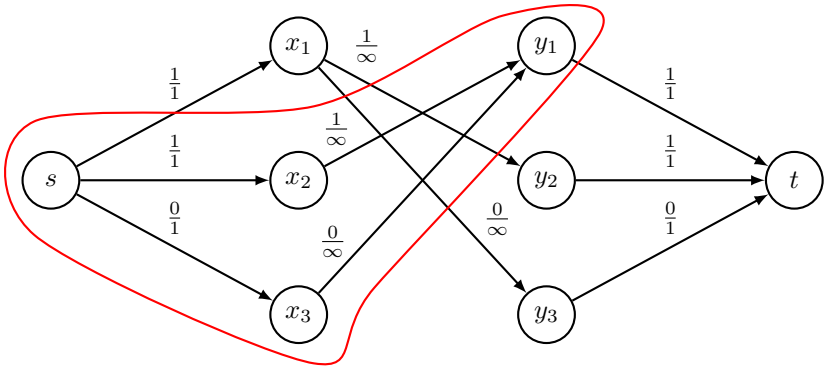
8.1 The Disjoint Paths Problem

The *disjoint paths problem* is as follows. Given a directed graph $G = (V, E)$ with source s and sink t, determine the maximum number of *edge-disjoint* s-t paths in the G. Notice that, unlike a flow network, we are not provided with any capacities for the edges. The solution to this problem is relatively simple. Assign a capacity of value 1 to each edge in the graph, and run Ford-Fulkerson. After doing so, any edge with a flow of 1 is part of some s-t path, and any edge with a flow of 0 is not. Then, the value of the resulting flow, $v(f)$, is equal to the number of edge-disjoint paths in the graph. To find the paths themselves, we remove all edges with a flow value of 0 from the graph entirely. Then, we search for s-t paths by starting from s and making our way to t. When we reach the sink t, we drop all of the edges we have just used for this path and repeat the process until there are no more paths. When doing this, it is possible to get stuck by entering a cycle and revisiting a vertex we have already visited in the current path. This problem is solved by dropping all of the edges that form the cycle, and we resume the attempt to find an s-t path [2].

8.2 Vertex Cover in Bipartite Graphs

A *vertex cover* of an undirected graph $G = (V, E)$ is a subset of the vertices $C \subseteq V$ such that if all of the vertices in C along with their associated edges are removed, then the graph G will contain zero edges. We can use maximum flow to determine the *minimum vertex cover* in a bipartite graph. Let $G = (V, E)$ be a bipartite graph with a disjoint partition of the vertices X, Y such that all the edges have one endpoint in X and one endpoint in Y. We then set up a flow network similar to how we did for the *maximum bipartite matching problem* by adding a source s and sink t and connecting s to the vertices in X and the vertices in Y to t. However, rather than assigning a capacity of 1 to all the edges, we assign a capacity of ∞ to the edges from vertices in X to those in Y and a capacity of 1 on all remaining edges. Realize that the ∞ capacity on the edges does not change the result of the problem. We could have set these capacities to 1 and obtained the same result, but setting them to ∞ makes analysis simple. We then run Ford-Fulkerson to find an optimal flow configuration and run a

graph search algorithm on the residual graph G_f to find a min-cut, (A^*, B^*). Let $X_1 = X \cap A^* \setminus \{s\}$, $X_2 = X \cap B^* \setminus \{t\}$ and $Y_1 = Y \cap A^* \setminus \{s\}$, $Y_2 = Y \cap B^* \setminus \{t\}$. Then the minimum vertex cover is the set of vertices that are neither the source nor the sink and contribute to the minimum cut: $X_2 \cup Y_1$. Having the edges with ∞ capacities makes them ineligible to contribute to the capacity of the minimum cut. We know the capacity of the minimum cut must be finite since the cut $(\{s\}, V \cup \{t\} \setminus \{s\})$ is a valid s-t cut with finite capacity. Consider the example below.



The diagram above shows that the source s connects to the vertices in X, and the vertices in Y connect to the sink t. All of those edges have capacities of 1. The edges that go from the vertices in X to the vertices in Y have a capacity of ∞. Consider the cut shown above by the vertices bounded by the red loop. Based on our earlier definitions we have: $x_2, x_3 \in X_1$, $x_1 \in X_2$, $y_1 \in Y_1$, and $y_2, y_3 \in Y_2$. Our claim states that $X_2 \cup Y_1$ are the vertices that are neither the source nor sink that contribute to the minimum cut and therefore form the minimum vertex cover. In this case, that would be vertices x_1, y_1, with the edges sx_1 and y_1t contributing. There is an interesting relationship between the maximum bipartite matching and the minimum vertex cover of a bipartite graph that is similar to the relationship between the maximum flow value and the minimum cut capacity [11, 12, 13].

Theorem (König) Let G be a bipartite graph. The cardinality of the maximum matching equals the cardinality of the minimum vertex cover.

8.3 Survey Design

The *survey design problem* is one that we can model as a circulation with lower bounds to find a viable solution. In this problem, a company sells m products in total and wants to survey n customers about their experience with their products. However, each customer has only purchased some subset of the m products. In addition, we want to limit the number of survey questions asked to each customer, but we want to get sufficient information about each product.

Let $q_i, i \in \{1, 2, ..., n\}$ be the maximum number of questions asked to customer i and let $p_j, j \in \{1, 2, ..., m\}$ be the minimum number of questions we want to ask about product j. Let's model our products and customers as a bipartite graph $G = (V, E)$ where X is a set of m vertices representing the products and Y is a set of n vertices representing the customers such that $V = X \cup Y$. All the edges in the graph are from vertices in the set X to vertices in the set Y. We add an edge from vertex $x_i \in X$ to the vertex $y_j \in Y$ if product i was purchased by customer j. The capacity on this edge is 1, with a flow of 1 indicating that the customer j is asked about product i.

Let's form our flow network by adding a source s and sink t. We connect s to each of the vertices representing the products $x_i, \forall i \in \{1, 2, ..., m\}$ with the edge to x_i having a lower bound of p_i and capacity of ∞. Similarly, we connect each of the vertices representing the customers $y_j, \forall j \in \{1, 2, ..., n\}$ to t with the edge from y_j having a lower bound of 0 and capacity of q_j. The lower bound of p_i and upper bound of ∞ on the edges entering vertex x_i correspond to the product requiring at least p_i questions asked about it. The lower bound of 0 and upper bound of q_j assigned to the edge exiting vertex y_j corresponds to the fact that customers are asked a maximum of q_j questions. Since we want to set up a circulation problem, we need to add one more edge from t to s with a lower bound of 0 and a capacity of ∞. This edge is required because the amount of flow generated by s and absorbed by t is unknown in advance, so we cannot set explicit demands for those vertices. To determine if our survey design is possible, we check if a feasible circulation exists for the circulation problem we constructed [2, 9].

8.4 Airline Scheduling

The *airline scheduling problem* demonstrates the ability of flow networks to model scheduling problems. Suppose we are given a list of n flights that need to be scheduled. Each flight is composed of four pieces of information: the departure location, the departure time, the arrival location, and the arrival time. We will model this as a circulation problem and use our ability to solve the circulation problem to solve the scheduling problem.

For each of our n flights, add two vertices to our circulation, $u_i, v_i, \forall i \in \{1, 2, ..., n\}$, with each pair having an edge between them with a lower bound of 1 and capacity of 1. This lower bound and capacity combination ensures that the edge must have a flow value of 1 in our solution, corresponding to the requirement that the flight be serviced. Next, for each pair of flights i, j such that the arrival location of i is the departure location of j and the arrival time of i is before the departure time of j, add an edge from v_i to u_j with a lower bound of 0 and capacity of 1. The lower bound and capacity combination describes the idea that the plane servicing the second flight after servicing the first is possible but optional. We now introduce the source vertex s and the sink vertex t. Since a plane can begin with any flight, we will add an edge from s to each vertex in the set X with a lower bound of 0 and a capacity of 1. Similarly, any flight can be the final one for a particular plane, so we will add an edge from each

vertex in the set Y to t with a lower bound of 0 and a capacity of 1. Suppose we have k planes at our disposal. We will set the demand of the source vertex to be $-k$ and the demand of the sink vertex to be k. Since we would like to use any number of planes lesser than or equal to k to service our flights, we will add an edge from s to t with a lower bound of 0 and capacity of k for any excess planes that are not needed to form a feasible circulation in the remainder of the network. To determine if we can service these flights, we check for a feasible circulation [2, 14].

The solution described above allows us to determine if we can feasibly service the n flights given k planes. If we wanted to determine the minimum number of planes that are required to service the n flights, we could use a binary-search approach with respect to the value of k. First, realize that n is an upper bound for k. We know that we can service the n flights with n planes. Select k to be the center value in the interval $[0, n]$ and check for a feasible circulation. If one exists, reduce the interval to $[0, k-1]$. Otherwise, reduce the interval to $[k+1, n]$. Repeat the process with the new interval until the interval is no longer valid (i.e., the upper endpoint is lesser than the lower endpoint). Keep track of the smallest value of k for which a feasible circulation is possible.

8.5 Project Selection

The *project selection problem* demonstrates how flow networks can be used to solve certain optimization problems involving dependencies. Suppose we have a set of m projects P, with each project $i \in P$ having an associated revenue $r_i \in \mathbb{R}$. This revenue can be positive, indicating that the associated project is profitable, or negative, meaning that completing the project results in a loss. For each project i, there is a subset of projects in P that must be selected in order for project i to be feasibly selected. Let these dependencies be modeled by a directed graph $G = (V, E)$. In this graph, there is an edge from the vertex for project i to the vertex for project j if project i depends on project j. We would like to select a subset, $F \subseteq P$, of feasible projects such that profit, $\sum_{i \in F} r_i$, is maximized.

As we have done with other problems, we will model this problem as a flow network to solve it. Add a source vertex s and a sink vertex t to the graph. We connect the source to each profitable project i with an edge of capacity equal to the revenue of that project r_i. We connect each project j such that completing it results in a loss to the sink with an edge of capacity equal to the value of the loss $|r_j|$. Realize that since $(\{s\}, V \cup \{t\})$ is a cut with capacity $C = \sum_{i \in P | r_i > 0} r_i$, we have an upper bound on the value of the flow for this flow network. We will design the rest of the network such that the minimum cut yielded will be the subset of projects that maximize profit. To do this, we will assign a capacity of ∞ on all of the edges that describe a dependency. Since we know the capacity of the minimum cut is finite, these infinite capacities prevent the edges from contributing to the capacity of the cut, which would result in the dependency requirements of a project being invalidated.

Choose any cut (A, B) where $A \setminus \{s\}$ is a set of feasible projects. Then:

$$\begin{aligned} c(A,B) &= C - \sum_{i \in A \setminus \{s\}} r_i \\ \implies \sum_{i \in A \setminus \{s\}} r_i &= C - c(A,B) \end{aligned} \tag{12}$$

So the profit obtained from the set $A \setminus \{s\}$ is equal to the sum of all of the positive revenues minus the capacity of the selected cut. This clearly indicates that we want to select the cut with the minimum capacity, (A^*, B^*), and then select $A^* \setminus \{s\}$ as our feasible subset of projects to maximize profit [2, 10, 12, 14].

References

[1] "Network flow I - Carnegie Mellon University." [Online]. Available: https://www.cs.cmu.edu/ avrim/451f11/lectures/lect1025.pdf.

[2] J. Kleinberg and É. Tardos, Algorithm design. Malmö: MTM, 2022.

[3] "Design and analysis of algorithms: Electrical engineering and computer science," MIT OpenCourseWare. [Online]. Available: https://ocw.mit.edu/courses/6-046j-design-and-analysis-of-algorithms-spring-2015/.

[4] "Lecture 9 1 flows in networks - theory.stanford.edu." [Online]. Available: https://theory.stanford.edu/ trevisan/cs261/lecture09.pdf.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, Introduction to algorithms. Cambridge (Inglaterra): Mit Press, 2009.

[6] "6.854/18.415J: Advanced Algorithms (fall 2006)," 6.854/18.415J: Advanced Algorithms. [Online]. Available: https://courses.csail.mit.edu/6.854/06/.

[7] "1 last time - Massachusetts Institute of Technology." [Online]. Available: https://people.csail.mit.edu/moitra/docs/6854lec8.pdf.

[8] "CS 4820 spring 2021," CS 4820: Introduction to Analysis of Algorithms - Spring 2021. [Online]. Available: https://www.cs.cornell.edu/courses/cs4820/2021sp/.

[9] CMSC 451: Section 0101. [Online]. Available: https://www.cs.umd.edu/class/fall2017/cmsc451-0101/.

[10] "Applications of Flows and Cuts," Algorithms by Jeff Erickson. [Online]. Available: http://jeffe.cs.illinois.edu/teaching/algorithms/.

[11] "Lecture 14 - Stanford University." [Online]. Available: https://theory.stanford.edu/ trevisan/cs261/lecture14.pdf.

[12] "CS31 (algorithms), Spring 2020 : Lecture 16 1 matchings in Graphs." [Online]. Available: https://www.cs.dartmouth.edu/ deepc/Courses/S20/lecs/lec16.pdf.

[13] "1 bipartite matching and vertex covers - Princeton University." [Online]. Available: https://www.princeton.edu/ aaa/Public/Teaching/ORF523/ORF523_Lec6.pdf.

[14] "CS 473 algorithms, spring 2021," CS 473: Algorithms, Spring 2021. [Online]. Available: https://courses.engr.illinois.edu/cs473/sp2021/.