1 Network Flows

- When one thinks about a network (communication, social, transportation, computer networks etc), many fundamental questions naturally arise: (1) how well-connected is it, (2) how much data (commodity) can it transport, (3) where are its bottlenecks, etc. In fact, many non-network and non-flow problems are also frequently solved using network flow framework.

- Network flows have many practical applications but what makes them a must-know classical topic in algorithms is their elegant theory and beautiful algorithms. The theory itself dates back to 1950s (well before the internet or the web), when Ford and Fulkerson described an augmentation based method for finding maximum flows in a capacitated network, with transportation being the underlying motivation.

- The word flow has natural association with physical commodities such as traffic or data, but mathematically it is an abstract entity which originates at the source nodes, and is absorbed at sink nodes. (The mental image of liquid flowing through a network of pipes, however, is a useful visual aid.)

- Our basic model of network flow has

  1. a directed graph $G = (V, E)$
  2. a non-negative capacity for each edge
  3. a single source node $s$, and
  4. a single sink node $t$.

A single source and a single sink is not a serious limitation.

- We make the following simplifying assumptions, which eliminate some annoying pathologies and let us describe the key ideas more cleanly.
1. no edge enters the source,
2. no edge leaves the sink,
3. at least one edge is incident to edge node, and
4. all capacities are integers.

- Mathematically, a flow is a function $f : E \rightarrow R^+$ that assigns a non-negative number (real number) to each edge. The value $f(e)$ intuitively represents the amount of flow carried by edge $e$. The flow $f$ must satisfy the following two properties:

  1. **Capacity Constraint:** $0 \leq f(e) \leq c(e)$, for all $e \in E$.
  2. **Flow Conservation:** $\sum_{e \text{ into } v} f(e) = \sum_{e \text{ out of } v} f(e)$, for each node $v \neq s, t$.

- The first one is an obvious constraint, limiting the flow on each edge to no more than edge’s capacity. Sometimes, a lower bound $l(e)$ for an edge is also specified. We will discuss those generalizations later.

- The second ensures that flow is conserved at all transit nodes: the incoming flow equals outgoing flow. Obviously, we do not have any conservation constraints at $s$ and $t$.
• The value of the flow, denoted \( v(f) \), is the total amount of flow generated at the source

\[
v(f) = \sum_{e \text{ out of } s} f(e)
\]

Does it equal the flow into the sink node \( t \)? Why? (Note that we can only assume that \( f \) satisfies the capacity and conservation constraints. Anything else we need to derive from those axioms.)

• (Max Flow Problem). What is the maximum amount of flow that can be sustained in \( G \) from \( s \) to \( t \)?

• Clearly, the only obstacle to the flow are the capacities of the edges in \( G \). The bottleneck, however, doesn’t necessarily occur at the edges originating at \( s \) or terminating at \( t \). It can arise from a complicated interaction among the edges as flow snakes through \( G \).

• Applications of Network Flows. There are many applications of network flow. The most obvious applications deal with transfer of a commodity in an interconnected network, such as a road network, rail network, airlines, telecommunication network, gas or oil distribution networks, etc.

• But the abstract settings of network flow can be used to solve many problems that overtly have nothing to do with transportation, such as matrix rounding, or scheduling. The earth mover’s distance, an important metric, in machine learning, is really a
min-cost flow problem. As another example, consider the following project selection problem.

- **Project Selection**: A company needs to decide which of many potential projects to pursue, to make best use of its limited resources.

- Each project is potential source of future revenue but also requires (upfront) cost and investment. In addition, projects may have dependencies: some cannot be done without others, and doing one project may enable future projects etc.

- For instance, a telcom company may want to assess the pros and cons of offering a high-speed service, which will generate *future revenue*, but it will need to upgrade its fiber optic network and install new routers, which require *upfront cost*. Once the fibers and routers are installed, however, other services may benefit as well.

- The project selection problem may have the following form:

1. A set $P$ of projects.
2. Each project $i$ has revenue $p_i$, which can be $+$ or $-$ (revenue or cost)
3. A dependency graph $G = (P, E)$ with projects as nodes $P$, and an edge $(i, j)$ whenever $i$ can *only be selected* if $j$ is also selected.
4. In other words, edge $(i, j)$ means $i$ depends on $j$: so to perform $i$, we must also select $j$.
5. A subset of projects $A$ is feasible if for every $i \in A$, its prerequisites are also in $A$
6. The profit for subset $A$ is $\text{profit}(A) = \sum_{i \in A} p_i$.
7. The Project Selection problem is to *select a set of projects with maximum profit*.

- Fill in details yourself to practice your modeling skills. HW 1 includes some problems similar to this.

- **Open Pit Mining.** A problem with this character was initially studied in the mining literature.

1. OPM is a surface mining operation where the blocks of earth are extracted from (exposed) surface for the ore contained in it.
2. The entire *mining area* is divided into a set $P$ of blocks, and net value $p_i$ of each block is estimated (i.e. value minus the processing cost in isolation). These $p_i$ values can be $+$ or $-$.
3. The set $P$ also has *precedence constraints* because a block can only be mined if all *blocks above* it have been mined.
4. The decision problem is to choose the profit-maximizing subset of blocks.
1.1 Ford Fulkerson Method

- We now turn our attention to the design of efficient algorithms for the maximum flow problem. A priori it is not clear that any such algorithms exist (the problem could be NP-hard). (In fact, the “reducibility” of problems such as projection selection or OPM can be used as an indication that network flow is quite general and therefore may not be so easy to solve.)

- The first efficient algorithm was developed by Ford and Fulkerson in 1956. The paper introduced many influential ideas including Augmentation, Residual Networks, and the famous MaxFlow-MinCut Theorem.

- The Generic form of the FF algorithm is basically a greedy flow algorithm. Start with the all-zero flow and greedily produce flows with ever-higher value.

  while there exists an s-t path with non-zero capacity,
augment flow along that path (and reduce capacity along the path).

- Note that the path search just needs to determine whether or not there is an s-t path in the subgraph of edges with $f(e) \leq c(e)$. This is easily done in linear time using your favorite graph search subroutine, such as BFS or DFS. There may be many such paths; for now, we allow the algorithm to choose one arbitrarily. The algorithm then pushes as much flow as possible on this path, subject to capacity constraints.

- The algorithm seems to find the optimal flow on some simple examples, such as the first example. But how can we be sure?
• In general, the flow may not saturate the edges from $s$ or into $t$, in which case how would we convince ourselves that the greedy method has found the maxflow when it terminates? For instance, consider the following example. After 2 units of flow, there is no augmenting path available, but neither $s$ nor $t$ has its incident edges saturated.

![Diagram](image)

• We need a *an efficient (polynomially-checkable) proof of optimality* to be sure that greedy algorithm terminates with the maximum flow.

• Actually, the greedy method is not always optimal. The previous figure is in fact a counterexample.

• The greedy algorithm pushes flow *indiscriminately* along any feasible path, and some of those choices may be harmful for future choices.

### 1.2 Residual Networks and Flow Augmentation

• The insight of Ford and Fulkerson is that greedy augmentation must be coupled with ability to *undo* bad moves.

• A principled approach to this “undoing” of earlier augmentations is through what FF call an *residual networks*.

• The residual networks are defined with respect to an existing (current) flow $f$, and so we will always use the subscript $f$ to make that explicit.
• Define the residual capacity of edge $e$ as

$$c_f(u, v) = c(u, v) - f(u, v)$$

• So, the residual capacity is the additional flow that can be sent along an edge $(u, v)$.

• The important point is that if $(u, v) \in E$, and we have $f(u, v) > 0$, then the reverse edge $(v, u)$ now has a positive residual capacity:

$$f(v, u) = 0 - f(v, u) = 0 + f(u, v)$$

• A residual network $G_f = (V, E_f)$ has the same set of vertices as $G$, but it may have upto twice as many edges: the edges of $E$ as well as their reverse.

$$E_f = \{(u, v) \mid c_f(u, v) > 0\} \quad \text{where} \quad |E_f| \leq 2|E|$$
• More precisely, an edge $(u, v)$ is in $E_f$ if
  1. either $(u, v) \in E$ and $f(u, v) < c(u, v)$, or
  2. $(v, u) \in E$ and $f(v, u) > 0$.
• Note that $E_f$ only includes edges with non-zero capacity.
• An augmenting path $p$ is a simple path in $G_f$ from $s$ to $t$. Residual capacity of a path is defined as (always $> 0$)

\[ c_f(p) = \min\{c_f(u, v) | (u, v) \in p\} \]

**Ford-Fulkerson Augmentation Algorithm**

1. initialize $f = 0$
2. for each edge $(u, v) \in E$, do $f(u, v) = f(v, u) = 0$
3. while there is an augmenting path $p$ in $G_f$ do
   (a) for each edge $(u, v) \in p$
      i. $f(u, v) = f(u, v) + c_f(p)$
      ii. $f(v, u) = -f(u, v)$
   (b) Update $G_f$
• Ford-Fulkerson on an example.
1.3 Proof of Correctness and Maxflow Min-cut Theorem

- In proving that this algorithm always finds the maximum flow, Ford Fulkerson established the famous maxflow-mincut theorem. In fact, this structural result is at least as important, if not more, as the maxflow algorithm.

- We need to prove that no matter which sequence of augmentations the algorithm performs, when $G_f$ has no $s$-$t$ path, the flow must have maximum value.

- The proof will need the concept of an $s$-$t$ cut.

- A $s$-$t$ cut is a partition of nodes $(A, B)$ with $s \in A$ and $t \in B$. (That is, except $s$ being in $A$ and $t$ being in $B$, there is no other restriction; the subsets don’t need to be connected, or equal size, etc.)

- How many cuts are there in $G$? (An exponential number! The powerset of $V$.)

- The capacity of the cut $(A, B)$ is defined as

$$c(A, B) = \sum_{e \text{ out of } A} c(e)$$
• The net flow across a cut \((A, B)\) is defined as

\[
f(A, B) = \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ into } A} f(e)
\]

• In the example, the capacity of the cut \((A, B)\) is 26, and the net flow across it is 19.

• Flow Value Lemma. Let \(f\) be any flow and let \((A, B)\) be any \(s-t\) cut. Then,

\[
\sum_{e \text{ out of } A} f(e) - \sum_{e \text{ into } A} f(e) = v(f)
\]

• That is, the netflow across any cut is always the same. (Intuitively, this follows from the fact that flow must be conserved at all nodes except \(s\).)

• Proof.

\[
v(f) = \sum_{e \text{ out of } s} f(e)
\]

\[
= \sum_{v \in A} (\sum_{e \text{ out of } v} f(e) - \sum_{e \text{ into } v} f(v))
\]

\[
= \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ into } A} f(e)
\]
• In this proof, the second equality follows from flow Conservation—zero netflow except at \( s \). Third equality: for all vertices of \( A \), we are adding the outgoing flow and subtracting the incoming flow. For any edge \( e = (u, v) \), if both of its endpoints are in \( A \), then \( f(e) \) gets canceled—appearing once as outgoing from \( u \), and once as incoming at \( v \)—leaving only those edges that have precisely one endpoint in \( A \).

• **Weak Duality.** Let \( f \) be any flow and let \((A, B)\) be any \( s \)-\( t \) cut. Then, \( v(f) \leq c(A, B) \).

  • **Proof.**

\[
v(f) = \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ into } A} f(e)
\leq \sum_{e \text{ out of } A} f(e)
\leq \sum_{e \text{ out of } A} c(e)
= c(A, B).
\]

• But the weak duality only guarantees that the maxflow cannot exceed the capacity of the mincut. It say nothing about how large it must be!

• **Maxflow Mincut Theorem.** The following statements are equivalent:

  1. \( f \) is a maxflow.
  2. There is no augmenting path in \( G_f \).
  3. \(|f| = c(A, B)|\) for some cut \( s \)-\( t \) cut \((A, B)\).

  • **Proof.** We will prove by the following sequence of implications.

  1. \((1 \rightarrow 2.)\) This is easy: if there were a path, we could increase \( f \), which would mean \( f \) could not be maxflow.

  2. \((2 \rightarrow 3.)\) Define a \( s \)-\( t \) cut as follows. Let \( A \) be the set of all vertices that are reachable from \( s \) in \( G_f \) (i.e. there is a path). Let \( B = V \setminus A \).

   This is a valid \( s \)-\( t \) cut because \( s \in A \) by construction, and \( t \in B \) because no augmenting path means \( t \) is not reachable from \( s \).

   Now, in the flow graph, consider any edge \((u, v)\) with \( u \in A \) and \( v \in B \). We must have \( f(u, v) = c(u, v) \) since otherwise \((u, v) \in E_f \), and \( v \) is reachable from \( s \). For any edge \((u, v)\) with \( u \in B \) and \( v \in A \), we must have \( f(u, v) = 0 \) because otherwise \( c_f(v, u) > 0 \), putting \( u \) in \( A \). Therefore, \(|f| = f(A, B) = c(A, B)\).

  3. \((3 \rightarrow 1.)\) \(|f| = f(A, B) \leq c(A, B)\). So, equality means flow is a maxflow.

• Therefore, the correctness of FF (stopping condition is absence of augmenting paths) follows from the maxflow mincut theorem.
1.4 Time Complexity Analysis of Ford Fulkerson.

- If the edge capacities are integers, with the largest one being $C$, then the running time can be bounded by $O(mnC)$:
  1. Each augmentation can be done in $O(m)$ time, and each $G_f$ update in $O(n)$ time.
  2. Each augmentation increases flow by at least 1.
  3. The maxflow has value at most $nC$, because the cut $(s, V - s)$ has capacity $\leq (n - 1)C$.

- A pathological example on 4 nodes, with edge capacity $M$ at 4 edges and capacity 1 in the middle.

- Worse, if the capacities are irrational, the algorithm may not even terminate.
2 Polynomial Algorithms: Edmonds-Karp Augmentations

- FF method does not prescribe how to pick augmenting path. As our previous examples show, careless choice of augmenting path can cause the algorithm to take a long (non-poly) time to reach maxflow.

- We now discuss two famous heuristics of augmenting path selection, analyzed by Edmonds and Karp, that guarantee maxflow computational in worst-case polynomial time.

- One is called **shortest path augmentation** (SPA), and the other **max capacity augmentations** (MCA).

- Heuristics themselves are somewhat “obvious;” cleverness lies in their analysis.

- The analysis is based on identifying an appropriate parameter in the flow network, with a bounded value, and then showing that each augmentation “consumes” enough of this quantity, which in turns gives an upper bound on the maximum number of augmentations possible before maxflow is reached.

- We begin with SPA, which always augments flow along the shortest path from \( s \) to \( t \).

2.1 Shortest Path Augmentation

- Let \( \delta_f(s, v) \) denote the shortest path distance from \( s \) to \( v \) in the residual graph \( G_f \).

- Standard BFS algorithm will compute \( \delta_f(s, t) \), and the path, in \( O(m) \) time.

- The following is an important property of these distances under SPA.

- **Distance Lemma:** \( \delta_f(s, v) \) increase monotonically throughout the algorithm.

- I don’t prove Distance Lemma here, and you are asked to do it in Homework 1.

- With the benefit of Distance Lemma, we now show that SPA delivers following performance guarantee for maxflow computation.

- **Theorem:** SPA algorithm performs \( O(nm) \) augmentations.

- **Proof of SPA** consists of the following steps.

- Given an augmenting path \( p \), call \( (u, v) \in p \) critical if \( c_f(p) = c_f(u, v) \).
• How many times can the same edge become critical?
• By SPA property, when \((u, v)\) first becomes critical, \(\delta_f(s, v) = \delta_f(s, u) + 1\).
• After augmentation, \((u, v)\) is removed. It can reappear only after \((v, u)\) is on some augmentation path.
• If \(f'\) is the existing flow when \((v, u)\) becomes part of augmenting path:
  \[\delta_{f'}(s, u) = \delta_{f'}(s, v) + 1\]
• Since \(\delta_f(s, v) \leq \delta_{f'}(s, v)\), by the Distance Lemma, we have the following:
  \[\delta_{f'}(s, u) = \delta_{f'}(s, v) + 1 \geq \delta_f(s, v) + 1 = \delta_f(s, u) + 2\]
• Thus, \(\delta_f(s, u)\) must increase by at least 2 before it can become critical again.
• Since we always have \(\delta_f(s, u) \leq n - 1\), an edge becomes critical \(O(n)\) times.
• Total number of critical pairs is \(O(nm)\), which is also the bound on number of augmentations—each augmentation has at least one critical edge.
• Since each augmentation can be done in \(O(m)\) time, worst-case complexity of SPA is \(O(nm^2)\). This completes the proof of SPA.

### 2.2 Max Capacity Augmentation (MCA)

• For a path \(p = (e_1, e_2, \ldots, e_k)\), path capacity \(c(p) = \min\{c(e_1), c(e_2), \ldots, c(e_k)\}\).
• MCA always augments along the path of maximum capacity.
• How would you find a max capacity path in \(G_f\)? It’s a simple exercise to show that it can be done in time \(O(m \log n)\).
• Starting with initial flow \(f = 0\), we need to bound the number of times MCA performs an augmentation before reaching maxflow.
• We need the following Flow Decomposition lemma.
• **F-D Lemma:** Starting with zero flow, one can construct the optimal flow $f^*$ in at most $m$ steps, where each step increases the flow along one path in the original graph $G$.

• A constructive proof of the lemma is as follows:

1. Let $G^*$ be graph induced by $f^*$. (That is, $(v, w) \in E^*$ iff $f^*(v, w) > 0$.)
2. Initialize $i = 1$. **repeat:**
   a. Find a path $p_i$ from $s$ to $t$ in $G^*$.
   b. Set $\Delta_i = \min\{f^*(v, w) \mid (v, w) \in p_i\}$.
   c. Decrease $f^*(v, w)$ by $\Delta_i$ for all $(v, w) \in p_i$.
   d. Delete $(v, w)$ from $G^*$ if its flow is zero.
   e. Stop if no edges with non-zero flow left; otherwise go to (a) with $i = i + 1$.
3. Each loop execution deletes at least one edge, so at most $m$ iterations.
4. Reconstruction of $f^*$: for $i = 1, 2, \ldots$, push $\Delta_i$ flow along each $p_i$.

• We now prove the performance guarantee of MCA.

• **Theorem:** Assuming integer capacities (max edge capacity $C$), MCA does $O(m \log C)$ augmentations.

• **Proof of MCA** consists of the following steps.

• Let $f^*$ be maxflow, and $f$ the current flow.

• $f' = f^* - f$ is a flow in $G_f$, and by F-D Lemma, we can reach $f^*$ from $f$ using $\leq m$ path augmentations.

• So, the max capacity path has residual capacity $\geq \frac{1}{m}(\lvert f^* \rvert - \lvert f \rvert)$. 
  (Why?)

• Look at $2m$ consecutive MCAs. One of them must augment by $\leq \frac{1}{2m}(\lvert f^* \rvert - \lvert f \rvert)$. 
  (Why?)

• In $2m$ augmentations, MCA capacity shrinks by $\frac{1}{2}$.

• Initial MCA capacity $\leq C$, and final is $\geq 1$.

• At most $\lceil \log C \rceil$ iterations of $O(m)$ augmentations.

• Each MCA path can be found in $O(m \log n)$ time.
• MCA runs in worst-case time

\[ O(m^2 \log n \log C). \]

• Strictly speaking, since the running time of MCA depends on flow parameter value \( C \), it is often called “weakly polynomial.” By contrast, SPA is strongly polynomial.

3 Extensions and Applications of Maxflow-Mincut

• As an illustration of how maxflow framework can be used to solve a variety of optimization problems, we consider the problem of assigning different modules of a program to 2 processors in a distributed computing system, with the goal of minimizing the collective cost of computation and inter-processor communication.

• The setting is a distributed computer system with 2 processors, and a large program with many modules.

• The processors differ in memory, control, speed and arithmetic capabilities etc, and so module \( i \) takes \( \alpha_i \) computation time on module 1 and \( \beta_i \) time on module 2.

• Different modules, however, incur high overhead in the form of interprocessor cost if assigned to different processors. In particular, if modules \( i \) and \( j \) are assigned to different processors, there is cost of \( c_{ij} \). This cost is 0 if they are assigned to the same processor.

• The goal is to minimize total processing and communication cost.

• In particular, if the subset of modules \( A_1 \) is assigned to processor 1 and \( A_2 \) to processor 2, then the cost of this assignment is

\[ \sum_{i \in A_1} \alpha_i + \sum_{i \in A_2} \beta_i + \sum_{(i,j) \in A_1 \times A_2} c_{ij} \]

• We can formulate this problem as a mincut problem on an undirected graph as follows. (The undirected graph can be turned into a directed one, easily by replacing each edge with two oppositely directed edges.)

• Introduce node \( i \) for module \( i \). If modules \( i \) and \( j \) interact, then add an edge of capacity \( c_{ij} \) between them.

• Add a source node \( s \), and connect it to each module \( i \) with an edge \((s,i)\) of capacity \( \beta_i \).
• Add a sink node $t$, and connect it to each module $i$ with an edge $(i, t)$ of capacity $\alpha_i$.

• An Example. 4 jobs \{1, 2, 3, 4\}, where $\alpha_i = 6, 5, 10, 4$, and $\beta_i = 4, 10, 3, 8$, for $i = 1, 2, 3, 4$.

• The inter-module costs are: $c_{12} = 5, c_{21} = 5, c_{23} = 6, c_{24} = 2, c_{32} = 6, c_{34} = 1, c_{42} = 2, c_{43} = 1$. Other costs are 0. (Notice that the edge costs are symmetric: $c_{ij} = c_{ji}$.)

![Diagram of a graph with nodes and edges labeled with capacities and costs.]

• Now consider a $s$-$t$ cut that partitions the modules into two subsets $A_1, A_2$. What is the capacity of the corresponding $s$-$t$ cut $(s \cup A_1, t \cup A_2)$?

• This cut contains an arc $(i, t)$ for each $i \in A_1$, whose capacity is $\alpha_i$, and an arc $(s, i)$ for each $i \in A_2$, whose capacity is $\beta_i$. In addition, for any $(i, j)$ for which $i \in A_1$ and $j \in A_2$, the arc of capacity $c_{ij}$ is also in the cut.

• Therefore, the cut $(s \cup A_1, t \cup A_2)$ has capacity precisely

$$\sum_{i \in A_1} \alpha_i + \sum_{i \in A_2} \beta_i + \sum_{(i,j) \in A_1 \times A_2} c_{ij}$$

• A mincut, therefore, gives the assignment of modules to processors with minimum cost.

3.1 Extension to Multiple Sources and Sinks

• Our maxflow framework assumes that all the flow originates at a single source $s$ and is destined for a single sink $t$. 
• In many scenarios, we have a collection of “supply nodes” (sources) and another collection of “demand” nodes (sinks), and we want to find the maximum amount that can be collectively shipped from all sources to all sinks.

• This is quite easy to fix, by introducing a new artificial source $s^*$ and a new artificial sink $t^*$.

• We then connect $s^*$ to each of the $s_i$ node, with an infinite capacity edge, and connect each sink node $t_j$ to $t^*$ also with an infinite capacity edge.

• The maxflow from $s^*$ to $t^*$ maximizes the total amount that can be shipped from $\{s_1, s_2, \ldots, s_a\}$ to $\{t_1, t_2, \ldots, t_b\}$.

3.2 Capacity Constraints at Nodes

• Our maxflow framework has capacity constraints only on edges, but in some applications it is useful to have “transshipment constraints” nodes also, which constrain how much total flow can pass through a “interchange” (node).

• This generalization is also easy. We replace each node $u$ with two nodes $u'$ and $u''$, with edge $(u', u'')$ assigned the capacity of node $u$. We then direct all incoming edges of $u$ into $u'$, and all outgoing edges of $u$, out of $u''$.

• Now we have only edge capacities, so the standard framework works.

3.3 Maximum Cardinality Bipartite Matching

• Given a bipartite graph $G = (X, Y, E)$, a matching is a subset of edges $M \subset E$ in which no two edges have a common vertex. That is, each vertex of $X$ is uniquely matched (paired) with a vertex of $Y$.

• The maximum matching $M^*$ is one with the largest number of edges.

• An example of unweighted assignment problem: jobs and workers, with edges representing compatibility. Assign as many workers to job as possible.

• Easy to check that simple-minded greedy methods do not work. Example: 3-edge path. However, greedy matching contains at least $1/2$ as many edges as optimal.

• Before we reduce the matching problem to maxflow, let us acknowledge the issue of integrality in the maxflow problem.
• Given a network in which all edges have integer capacities, the maxflow function (which assigns flow values to edges) need not be integral! Example with 1/2 unit flows, merging to form 1-unit flow.

• However, our augmentation-based algorithms constructively find an integral maxflow function. This fact will be useful in many cases, including matching.

• We can transform the matching problem to a maxflow problem, in a fairly transparent way:
  1. Introduce a node $s$, and join it to all nodes in $X$.
  2. Introduce a sink $t$, and join all nodes of $Y$ to $t$.
  3. Direct all edges from $X$ to $Y$.
  4. Given all edges capacity 1.
  5. Compute a maxflow in this network.

• We claim that the value of maxflow equals the cardinality of the max bipartite matching, and in fact the matching can be recovered from the integral maxflow.

• The equivalence of max-matching and maxflow is straightforward.

• First, suppose there is a matching with $k$ edges $(x_i, y_i)$. Then, we can produce a flow of value $k$, in which 1 unit of flow goes through each of the $k$ paths of the form $(s, x_i, y_i, t)$. Easy to check this is a valid flow (respects capacity and conservation).

• Conversely, given a maxflow of value $k$, we show a matching of size $k$.

• But first we need to address an subtle issue of integrality in the maxflow problem.

• Given a network in which all edges have integer capacities, the maxflow function (which assigns flow values to edges) need not be integral! Example with 1/2 unit flows, merging to form 1-unit flow.

• However, our augmentation-based algorithms constructively always find an integral maxflow function, we can assume the the maxflow is actually integral.

• Since the capacity of each edge is 1, and the flow is integral, the only possible values of flow for each edge are 0 and 1. Consider all edges of the form $(x, y)$ with flow 1. Call this set $M'$. 
• We claim that $M'$ contains exactly $k$ edges. To prove this, consider the cut $(A, B)$, where $A = s + X$. The value of flow is the total flow leaving $A$ minus the flow entering $A$. There are no edges entering $A$, and each edge leaving $A$ has capacity 1, so there must be $k$ edges leaving $A$, each carrying 1 unit of flow. These edges form $M'$.

• Next, we claim that each node in $X$ is the tail of at most one edge in $M'$. Suppose, instead, a node $x \in X$ is the tail of two edges in $M'$. Because flow is integer valued, this would mean that at least 2 units of flow leave $x$. By flow conservation, then at least 2 units must enter $x$, but that is impossible because the incoming edge into $x$ has capacity 1.

Similarly, each edge of $M'$ is the head of at most edge in $M'$.

• Therefore, the edges of $M'$ form a matching of size $k$.

3.4 Disjoint Paths: Menger’s Theorem

• In many applications, we want to compute many edge-disjoint paths between $s$ and $t$, for fault tolerance, parallelism etc.

• In addition, the flow as discussed so far has a static feel: we just describe it as a number associated with each edge. It does not describe how the flow travels through the network as in traffic. It would be useful to obtain this kind of traffic picture.

• We say that a set of paths, each path starting at $s$ and ending at $t$, is edge-disjoint if no two paths have a common edge.

• Given a directed graph $G$, and two nodes $s$ and $t$, the Edge-Disjoint Path problem is to find the maximum number of disjoint paths from $s$ to $t$.

• We will prove that the maximum number of edge-disjoint $s$-$t$ paths = maximum $s$-$t$ flow in $G$.

• In the forward direction, if there are $k$ edge-disjoint paths, then maxflow is at least $k$: simply ship 1 unit of flow along each path. It easily satisfies the capacity and conservation properties.

• In the converse direction we prove the following: if $f$ is a 0-1 valued flow of value $k$, then the set of edges with flow $f(e) = 1$ contains $k$ edge-disjoint paths.

• We prove this by induction on the number of edges that carry non-zero flow. If $k = 0$, then the claim is trivial. Otherwise, there is at least one edge $(s, u)$ that carries a unit of flow. We now trace out a path of edges that must carry the flow: by flow
conservation, some edge \((u, v)\) must have flow, and so on, until we either reach \(t\) or return to a node \(v\) for the second time.

- In the former case, we now have a path \(p\) from \(s\) to \(t\), which we add to our list. We reduce the flow on the edge of \(p\) to 0. The new flow has value less than \(k\), and it has fewer edges of non-zero flow, which by induction yields the result.

- In the latter case, we have a cycle \(C\). If we decrease the flow value on all edges of \(C\) to zero, it doesn’t affect the total flow from \(s\) to \(t\), but it has fewer edges with non-zero flow, so induction completes the proof.

- **Menger’s Theorem (1927):** In a directed graph, the maximum number of \(s\)-\(t\) edge-disjoint paths is equal to the minimum number of edges whose removal disconnects \(s\) from \(t\).

  - If the removal of \(F \subset E\) separates \(s\) from \(t\), then each \(s\)-\(t\) path must use at least one edge from \(F\). Thus, the maximum number of disjoint paths is at most \(|F|\).

  - Conversely, the maximum number of disjoint paths equals the maxflow. If this value is \(k\), then the maxflow-mincut theorem says that there is a \(s\)-\(t\) cut \((A, B)\) of capacity \(k\).

  - Let \(F\) be the set of edges that go from \(A\) to \(B\). Since each edge has capacity 1, \(F\) has exactly \(k\) edges. This proves Menger’s theorem.

  - A similar statement holds for vertex-disjoint \(s\)-\(t\) paths. Easy to see how!

### 3.5 Circulations with Demands

- Suppose we have multiple sources and sinks, where each source and sink has its own demand. Our goal now is to find a good flow rather than one that maximizing the total flow, which can be difficult to agree on due to fairness among different flows.

- More specifically, suppose each node \(v\) has an associated demand \(d_v\):
  1. If \(d_v > 0\), we say \(v\) is a demand node
  2. If \(d_v = 0\), \(v\) is a transshipment node
  3. If \(d_v < 0\), \(v\) is a supply node which can supply \(-d_v\) units.

- A circulation is a function \(f\) that satisfies the capacity constraint, namely, \(0 \leq f(e) \leq c(e)\), for all edges \(e\), and demand condition, namely, \(f_{in}(v) - f_{out}(v) = d_v\), for all nodes \(v\).
So, instead of optimization, we now have a feasibility problem (satisfying demands at all nodes subject to capacity constraints).

Clearly, if a feasible circulation exists then \( \sum_v d_v = 0 \). This is because \( \sum_v d_v = \sum_v (f_{in}(v) - f_{out}(v)) \). In the summation on the right, each edge appears twice, once in \( f_{in} \) and once in \( f_{out} \), canceling each other.

We can convert the circulation problem into a flow problem as follows.

1. Introduce a source \( s \), and join it to all supply nodes, with edge capacity equal to \(-d_v\).
2. Similarly, add a sink node \( t \), and join each demand node to \( t \), with edge capacity \( d_v \).
3. Now, the circulation is feasible if and only if the maxflow has value exactly \( \sum_{d_v > 0} d_v \).

3.6 Circulations with Demands and Lower Bounds

- In some applications, certain amount of flow is forced on some edges. That is, there is a lower bound on the value of flow at some edges: \( f(e) \geq l(e) \).

- The conditions for the flow, with demands, now change to:
  1. \( l(e) \leq f(e) \leq c(e) \), for all edges \( ee \)
  2. \( f_{in}(v) - f_{out}(v) = d_v \), for all nodes \( v \).
• If such a circulation feasible? We will do this in two steps.

• First, we reduce the problem to a circulation problem with demands but no lower bounds, which we know how to solve from previous subsection.

• Specifically, we set an initial circulation of $f_0(e) = l(e)$, for all edges. This circulation clearly satisfies the capacity constraint (both upper and lower bounds), but perhaps violates the demands. In particular, let

$$L(v) = f_0^{\text{in}}(v) - f_0^{\text{out}}(v) = \sum_{e \text{ into } v} l(e) - \sum_{v \text{ out of } v} l(e)$$

• If $L(v) = d_v$, we have satisfied the demand at $v$. Otherwise, we need to superimpose another circulation that will clear the imbalance introduced by $f_0$. So, we need to find a circulation $f_1$, where for node $v$,

$$f_1^{\text{in}}(v) - f_1^{\text{out}}(v) = d_v - L(v)$$

• How much capacity do we have to work with?

• Each edge $e$ has available capacity of $c(e) - l(e)$. So, we basically compute a circulation in the network $G'$, where $e$ has capacity $c(e) - l(e)$, and a node $v$ has demand $d_v - L(v)$.
4 Minimum Cost Flow Problem

- We now consider a natural generalization of the maxflow problem where sending flows along edges also incurs costs.
- Formally, we have a flow network $G = (V, E)$ where each edge $(i, j)$ has both a capacity $c_{ij}$ and a per unit cost or price $p_{ij}$. We write this as tuple $(c_{ij}, p_{ij})$.
- If the edge $(i, j)$ has $x$ units of flow on it, then the cost incurred is $xp_{ij}$.
- In addition, each node $v$ has a demand or supply $d(v)$, where we assume that $v$ is a supply node if $d(v) < 0$, a demand node if $d(v) > 0$; otherwise, $v$ is a transshipment node at which the flow must be conserved.
- We assume that the total supply equals total demand, and so all the $d(v)$’s sum to zero: $\sum_j d(v) = 0$. In general, it may suffice to assume that supply $\geq$ demand, and use a artificial sink to absorb the excess supply at zero cost.
- We wish to find a flow in which each node $v$ has net inflow $d(v)$ and the total cost of the flow is minimized.
- MCF generalizes shortest paths, maxflow, and bipartite matching.
  - The Shortest Path problem is a special case of MCF: capacity = 1 for all edges; $d(s) = -1$, and $d(t) = 1$.
  - The Min Cost matching is also a special case: $d(x) = -1$ for all $x \in X$; and $d(y) = 1$ for all $y \in Y$; edges have capacity 1.
  - The max flow problem is a special case when costs are zero.

4.1 The Min Cost Flow Algorithm

- The key lies in the residual graph of $f$. In the min cost formulation, if the forward edge $(i, j)$ has cost (price) $p_{ij}$, then the reverse edge $(j, i)$ has cost $-p_{ij}$. This corresponds to the fact that canceling a flow reduces the cost.
- A simple MCF algorithm starts by computing the maximum flow in $G$ ignoring the costs entirely.
- Specifically, we create a single source vertex $s$, add edges from $s$ to each supply node $v$ with capacity $-d(v)$.
- We also create a sink node $t$, add edges from each demand node $v$ to $t$ of capacity $d(v)$.
• In this graph, if the maxflow does not saturate all the edges coming out of s, then clearly no feasible solution exists.

• Otherwise the solution is feasible, but the maxflow computed may have very high cost.

• In order to reduce the cost of the maxflow, we construct the residual graph, and check to see if has a negative cost cycle. Remember that reverse edge in $G_f$ have negative costs. (How do you detect a negative cycle?)

• If $G_f$ does have a negative cycle, then we can reduce the cost of the flow, by pushing flow around the cycle, until we cannot push any more flow.

• Pushing a flow of $\delta$ around the cycle reduces the total cost of the flow by $\delta \times |P|$, where $P$ is the total (negative) cost of the of the cycle.

• Pushing the flow around a cycle does not change the net flow at any vertex; so flow remains conserved, and all demands and supplies are satisfied. We only reduce the cost of the flow.

• What if there is no negative cycle in $G_f$? Then, we must have a min cost circulation!

• **Theorem.** A flow $f$ is a min-cost flow if and only if $G_f$ has no negative cycles in it.

• **Proof.** The only-if part follows because if $G_f$ has a negative cycle, then we can reduce the cost of the flow, so $f$ can’t be min cost flow.

• For the if part, we argue that if $G_f$ does not have a negative cycle, then it has min cost.

  1. Suppose $f$ is not optimal, and there is another flow $f^*$ with the same value as $f$ but lower cost.
  2. Consider the flow $d = f^* - f$. Since both flows have the same value, $d$ is a circulation in which flow is conserved at all nodes.
  3. Consider the subgraph $H$ spanned by the edges with non-zero flow in $f^* - f$. Then, the following holds:
     (a) $H$ is a subgraph of $G_f$, and
     (b) $H$ consists of a set of cycles—since $d$ is a circulation.
  4. We can, therefore, decompose $d$ into a set of cycles in $G_f$.
  5. By pushing flow along these cycles, we can transform $f$ into $f^*$.
  6. But since all the cycles have non-negative costs (by assumption), this can only increase the cost, implying $p(f) \leq p(f^*)$.  

25
7. Since $f^*$ is optimal, we must have $p(f) = p(f^*)$, which completes the proof.

- Another way to write this is: Consider the flow $d = f^* - f$. The cost of $d$ is negative, and $f + d = f^*$, Which means that we can augment $f$ with $d$ and get $f^*$. The flow $d$ is a circulation, and it has negative cost. We can decompose $d$ into a sum of a bunch of cycles. To do this we just start following edges of $d$ that have positive flow until we come back to the starting point. This must happen because flow is conserved at all vertices. Now remove that cycle from $d$, and continue removing cycles until $d$ is zero. Since the total cost of $d$ is negative, we know that at least one of these cycles must have had a negative value. This proves the existence of a negative cycle in $G_f$.

- Time Complexity: The algorithm terminates because the cost of the flow decreases with each negative cycle augmentation, but not necessarily in polynomial time.

- Several polynomial time algorithms for min cost flow are known, but we won’t discuss their details or analysis. See Chapter 10 of AMO book.

- If we don’t demand polynomial time, then the following algorithm achieves $O(nmF)$ worst-case time.

1. Start with a zero flow $f$. Repeat the following augmenting step until the flow value reaches target value (say, $F$).
2. Find the shortest path from $s$ to $t$ in the residual graph $G_f$, namely the minimum cost path.
3. Augment along this path as much as possible (although not exceeding $F$).
4. Update the flow $f$ by this augmenting path.
5. If we have reached $F$, terminate.

- **Claim:** The residual graph in this algorithm never contains a negative cycle.

- **Proof.**

1. The initial residual graph contains no negative cycle because all initial costs are non-negative.
2. So we just need to show that the augmentation does not produce a negative cycle.
3. We do this by applying Johnson’s idea of “reduced costs” as follows.
4. In the original $G_f$ (before augmenting) do a full single source shortest path computation from the source $s$.
5. Let $d_v$ be the distance to vertex $v$. We know that $d_v \leq d_u + cost(u,v)$, for any $u$, because if this were not so there would be a shorter path to $v$ via $u$. 

26
6. Now adjust the costs of all edges as follows:

\[ \text{cost}'(u,v) = d_u - d_v + \text{cost}(u,v) \]

7. These costs are \( \geq 0 \).

8. But what about the new edges that we just added as a result of applying the augmenting path? Could they create a negative cycle?

9. The answer is no. Because the path we augment along is a shortest path, for such edges we have:

\[ d_u + \text{cost}(u,v) = d_v \Rightarrow \text{cost}'(u,v) = 0 \]

10. Not only that, but the adjusted cost of the reverse edge that we are adding is also zero.

\[ \text{cost}'(v,u) = d_u - d_v + \text{cost}(v,u) = d_u - d_v - \text{cost}(u,v) = 0 \]

11. The cost around any cycle with the original costs is the same as the cost around the cycle with the modified costs. Thus there can be no negative cycle.

- Time Complexity: Each step requires doing a Bellman-Ford search, which takes time \( O(nm) \). The number of times we have to do this is \( O(F) \), where \( F \) is the flow value. So the total running time is \( O(mnF) \).

5 Global Min Cut

- We now discuss a slightly off-topic: finding a global mincut. The goal is to delete the minimum number of edges of \( G = (V,E) \) that disconnect the graph into two non-empty subsets.

- This is different from finding a mincut separating a specific pair of nodes \( s \) and \( t \).

- Formally, we assume \( G = (V,E) \) is an undirected, multi-graph with \( n \) vertices and \( m \) edges. Multigraphs can have multiple (parallel) edges between a pair \( (u,v) \).

- A cut is a partition \( (C, V - C) \) of the vertices of \( G \) into 2 non-empty sets.

- We refer to \( C \) as the cut, and its size is the number of edges crossing the cut \( (C, V - C) \).

- We want to find a minimum size cut of \( G \), called a min cut.
• In the example graph, mincut has size 2.

• Global mincuts have many applications, ranging from network connectivity to reliability, TSP heuristics, databases partitioning etc.

• Traditionally, mincuts are computed using maxflow algorithms.
  1. The maxflow algorithm finds a $s-t$ min cut.
  2. By trying all different $s-t$ pairs, we can compute the global mincut.
  3. This requires $\binom{n}{2}$ maxflow computations needed.
  4. A better idea is to try $n - 1$ $s-t$ cuts, by fixing $s$ and varying $t$ over the remaining nodes.
  5. So, we can find the global mincut using $O(n)$ invovations of a maxflow algorithm.

• Surprisingly, in 1993, Karger discovered a simple (randomized) algorithm for computing global mincut in roughly $O(n^2)$ time. So the global mincut can be computed much faster than a single maxflow computation!

• The algorithm is both elegant and has been used to design algorithms for a number of other combinatorial problems.

• The main idea is to use edge contractions.

5.1 Edge Contraction

• Karger’s algorithm repeatedly contracts edges, where contracting $(x, y)$ means merging $x, y$.

• Specifically, we delete $x$ and $y$ and replace it with a new node $z$, which is called a meta vertex. All edges incident to $x$ or $y$, but not both, become incident to $z$.

• The contracted graph is denoted $G/(x, y)$. Note that contraction creates parallel edges (which is why we assumed input is a multigraph), and meta vertices.

• Observation: Effect of contracting a set of edges $F$ is independent of the order in which edges of $F$ are contracted.
5.2 The Basic Contraction Algorithm

1. **Input:** A multigraph $G = (V, E)$.

2. **Output:** A cut $C$.

3. $H \leftarrow G$.

4. while $H$ has more than 2 vertices do
   - Uniformly at random pick $(x, y)$ in $H$.
   - $H \leftarrow H/(x, y)$.

5. Output $(C, V - C)$ as the set of vertices corresponding to two meta vertices in $H$.
   - An example is shown in figure below.

- The sequence of edge contractions is: $\{1, 2\}, \{3, 4\}, \{(1, 2), 5\}$.
- The final mincut output by the algorithm is $(\{3, 4\}, \{1, 2, 5\})$. 
5.3 Analysis Karger’s Algorithm

- Interesting part is to show that Contraction produces a mincut with non-negligible probability.

- Clearly, a cut $C$ is produced as output by Contraction if and only if none of the edges crossing this cut is contracted by the algorithm.

- The proof uses two elementary observations:
  1. If $G$ has $n$ vertices, and its min cut value is $k$, then the number of edges in $G$ is $m \geq nk/2$. (This follows because the minimum vertex degree of $G$ must be $\geq k$.)
  2. The min cut value of $G/(x, y)$ is at least as large as in $G$. Why?

- Let us focus on a particular min cut $K$, and compute the probability that $K$ is output by the algorithm.

- Vertex count of $G$ drops by 1 in each iteration of Contraction.

- Let $n_i = n - i + 1$ be the vertex count at the start of $i$th iteration.

- Suppose none of the edges in $K$ are contracted in first $i - 1$ iterations.

- Since $K$ is also a cut in $H$, the graph has mincut value $k$. Thus, $H$ has $\geq n_ik/2$ edges.

- The probability that an edge of $K$ is contracted in this iteration is at most $\frac{k}{n_ik/2} = 2/n_i$. (Recall that we pick contraction edge uniformly, taking edge “multiplicity” into account.)

- We now compute the prob. that no edge of $K$ is ever contracted.

\[
\text{Pr [K output]} \geq \prod_{i=1}^{n-2} \left(1 - \frac{2}{n_i}\right)
= \prod_{i=1}^{n-2} \left(1 - \frac{2}{n - i + 1}\right)
= \prod_{j=n}^{3} \left(\frac{j - 2}{j}\right)
= \frac{1}{\binom{n}{2}}
= \Omega(n^{-2})
\]

- A specific min cut $K$ is output by algorithm with probability $\Omega(n^{-2})$. 
• We now a standard technique to amplify the probability of success. In particular, we run the algorithm $O(n^2 \log n)$ times, and return the smallest cut found in any of the iterations. $K$ will be produced almost surely (with prob. of success at least $1 - O(1/n)$).

• Each round can be implemented in $O(n^2)$ time.

• This is the basic algorithm, which can be improved (FastCut), to run in $O(n^2 \log n)$ time, and to produces a min cut with probability $\Omega(1/\log n)$ (in each execution).

• An interesting Consequence: How many mincuts are in $G$?

• Since a particular mincut is generated by the randomized algorithm with probability $\Omega(n^{-2})$, the total number of mincuts in $G$ is no more than $O(n^2)!$ Contrast this with the total number of cuts, which is exponential!

6 Multi-Commodity Flows and Misc

• In all of our discussions so far, we may have had multiple sources and sinks but they all send and receive the same commodity, namely, an indistinguishable good such as oil, water, natural gas, or broadcast of the same video.

• This is called the single commodity flow, and is quite different from the multi-commodity flow problem in which different sources and sink wish to exchange distinguishable goods, e.g. telecom data.

• When the flow (traffic) is of multiple types, but there is no interaction among them: for instance, each type has its own reserved capacity on edges, then we can still model the problem as multiple instances of single commodity flows.

• However, if the network resources (capacity) are to be shared among different commodities, with no pre-specified reservation, the problem become trickier.

• In the general case of the multi-commodity problem, we assume there are $k$ different types of goods.

• Each node can be the source or sink of one or more such commodities.

• The edge capacity is the upper bound on the total sum of the commodities flowing across that edge.

The general problem can be formulated as a Linear Program, and solved in polynomial time. We will return to this topic later.
• A critical issue in the multi-commodity flow is whether the flow needs to be integral or not. Unlike the single-commodity flows, the integrality in multi-commodity flows is harder to achieve.

• In fact, with the requirement that individual flows be integer-valued, the problem becomes NP-Complete. Some simple instance of this intractability are the following.

• **Disjoint Paths.**
  1. Given are a graph $G = (V, E)$, and a collection of disjoint vertex pairs $(s_1, t_1), \ldots, (s_k, t_k)$.
  2. Decide if $G$ contains $k$ mutually *vertex-disjoint* paths, one connecting each $s_i$ to its matching $t_i$.
  3. This problem is NP-complete, both for directed or undirected $G$. Even for planar graphs.

• **Directed 2-Commodity Integral Flow.**
  1. Given a directed graph $G = (V, E)$, and two pairs $(s_1, t_1)$, and $(s_2, t_2)$.
  2. There is positive integer capacity $c(e)$, for each edge, and integer demands $r_1$ and $r_2$ at nodes $t_1, t_2$, respectively.
  3. Decide if there integer flows $f_1, f_2$ such that
     - for each edge $e$, $f_1(e) + f_2(e) \leq c(e)$;
     - flow is conserved at each node $v$ other than $s_i$ and $t_i$;
     - for $i = 1, 2$, the net flow into $t_i$ under flow $f_i$ is at least $r_i$.

• This problem is also NP-complete, even if $c(e) = 1$ for all edges, and $r_1 = 1$.

• If we relax the *integrality requirement*, and are content with flow values that are fractional (acceptable in many applications), then MCF problem can be solved using Linear Programming.

• **Maxcut in Graphs.** By the way, unlike the mincut, finding a *maximum cut* in a graph is a much harder. In fact, the problem is $NP$-hard. So, while the two problems may appear superficially very similar, minimization and maximization of objective functions can have dramatic effect on computational complexity.