18-0: Flow Networks

- Directed Graph $G$
- Each edge weight is a “capacity”
  - Amount of water/second that can flow through a pipe, for instance
- Single source $S$, single sink $t$
- Calculate maximum flow through graph

18-1: Flow Networks

- Flow: Function: $V \times V \rightarrow R$
  - Flow from each vertex to every other vertex
  - $f(u, v)$ is the direct flow from $u$ to $v$
- Properties:
  - $\forall u, v \in V, f(u, v) \leq c(u, v)$
  - $\forall u, v \in V, f(u, v) = -f(v, u)$
  - $\forall u \in V - \{s, t\}, \sum_{v \in V} f(u, v) = 0$
  - Total flow, $|f| = \sum_{v \in V} f(s, v) = \sum_{v \in V} f(v, t)$

18-2: Flow Networks

- Single Source / Single Sink
  - Assume that there is always a single source and a single sink
  - Don’t lose any expressive power – always transform a problem with multiple sources and multiple sinks to an equivalent problem with a single source and a single sink
  - How?

18-3: Flow Networks

- Example: Shipping product to a warehouse
  - Product produced at a factory, put in crates
  - Crates are shipped to warehouse
  - To cut down costs, use “extra space” in other people’s trucks
  - How much product can be produced per day?

18-4: Flow Networks

\[\begin{aligned}
&\text{L.A.} & &\text{Chicago} & &\text{New York} & &\text{New Jersey} \\
&13 & &16 & &12 & &20 \\
&\text{Dallas} & &\text{New York} & &\text{New Jersey} & &\text{Chicago} \\
&14 & &9 & &7 & &4 \\
\end{aligned}\]
• It would be a little silly to ship 4 crates from Dallas to Chicago, and 7 crates from Chicago to Dallas
  • Could just ship 3 crates from Chicago to Dallas instead
• We will assume that there is only every flow in one direction
  • Flow in the opposite direction “cancels” out

18-6: Flow Networks

![Flow Network Diagram]

Is this flow optimal?

18-7: Flow Networks

![Flow Network Diagram]

18-8: Flow Networks

• Negative flow
  • It is perfectly legal for there to be a negative flow from \( v \) to \( u \)
  • Negative flow from \( v \) to \( u \) just means that there is a positive flow from \( u \) to \( v \)
  • Recall that the total flow over all edge incident to a vertex must be zero, except for source & sink

18-9: Flow Networks

• Residual capacity
  • \( c_f(u, v) \) is the residual capacity of edge \( (u, v) \)
  • \( c_f(u, v) = c(u, v) - f(u, v) \)
  • Note that it is possible for the residual capacity of an edge to be greater than the total capacity
  • Cancelling flow in the opposite direction

18-10: Flow Networks
- Residual Network
  - Given a set of capacities, and a set of current flows, we can create a residual network
  - Residual network can have different edges than the capacity network

18-11: Flow Networks

18-12: Flow Networks

18-13: Flow Networks
18-14: Flow Networks

- Given a flow network, with some flows calculated
- Induced residual network
- There is a path from source to sink in the residual network such that:
  - All residual capacities along the path are $> 0$
- How can we increase the total flow?

18-15: Augmenting Path

- An Augmenting path in a flow network is a path through the network such that all residual capacities along the path $> 0$
- Given a flow network and an augmenting path, we can increase the total flow by the smallest residual capacity along the path
  - Increase flow along path by smallest residual capacity along the path
  - May involve some flow cancelling

18-16: Augmenting Path
18-17: Augmenting Path

18-18: Augmenting Path
18-19: Augmenting Path

18-20: Ford-Fulkerson Method

Ford-Fulkerson\((G, s, t)\)

initialize flow \(f\) to 0

while there is an augmenting path \(p\)

augment flow \(f\) along \(p\)

return \(f\)

18-21: Ford-Fulkerson Method

- What is the running time of Ford-Fulkerson Method?
- Find an augmenting path
• Update flows / residuals
• Repeat until there are no more augmenting paths

18-22: **Ford-Fulkerson Method**
• What is the running time of Ford-Fulkerson Method?
  • Find an augmenting path
    • Using DFS, $O(|E|)$
  • Update flows / residuals
    • $O(|E|)$
  • Repeat until there are no more augmenting paths
    • Each iteration could increase the flow by 1, could have $|f|$ iterations!
  • Total: $O(|f| \times |E|)$

18-23: **Ford-Fulkerson Method**
• Could take as many as $|f|$ iterations:

![Flow Network Diagram](image)

18-24: **Ford-Fulkerson Method**
• Could take as many as $|f|$ iterations:

<table>
<thead>
<tr>
<th>Flow Network</th>
<th>Residual Network</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Flow Network Diagram" /></td>
<td><img src="image" alt="Residual Network Diagram" /></td>
</tr>
</tbody>
</table>

18-25: **Ford-Fulkerson Method**
• Could take as many as $|f|$ iterations:

**Flow Network**

```
 1/1000000  
\(s\) \(a\) \(0/1000000\) \(t\)  
0/1000000  
\(b\) 1/1000000
```

**Residual Network**

```
 999999  
\(s\) \(a\) \(1000000\) \(t\)  
1000000  
\(b\) 999999
```

18-26: **Ford-Fulkerson Method**
• Could take as many as $|f|$ iterations:

**Flow Network**

```
 1/1000000  
\(s\) \(a\) \(0/1000000\) \(t\)  
0/1000000  
\(b\) 1/1000000
```

**Residual Network**

```
 999999  
\(s\) \(a\) \(1000000\) \(t\)  
1000000  
\(b\) 999999
```

18-27: **Ford-Fulkerson Method**
• Could take as many as $|f|$ iterations:

**Flow Network**

```
 1/1000000  
\(s\) \(a\) \(0/1000000\) \(t\)  
0/1000000  
\(b\) 1/1000000
```

**Residual Network**

```
 999999  
\(s\) \(a\) \(1000000\) \(t\)  
1000000  
\(b\) 999999
```

18-28: **Ford-Fulkerson Method**
• How can we be smart about choosing the augmenting path, to avoid the previous case?

18-29: **Edmonds-Karp Algorithm**
• How can we be smart about choosing the augmenting path, to avoid the previous case?
  • We can get better performance by always picking the shortest path (path with the fewest edges)
• We can quickly find the shortest path by doing a BFS from the source in the residual network, to find the shortest augmenting path
• If we always choose the shortest augmenting path (i.e., smallest number of edges), total number of iterations is \( O(|V| \times |E|) \), for a total running time of \( O(|V| \times |E|^2) \)

18-30: Edmonds-Karp Algorithm

• If we always pick the shortest augmenting path, no more than \(|V| \times |E|\) iterations:
  • Lemma #1: Shortest path from source \( s \) to any other vertex in residual graph can only increase, not decrease.
    • Residual graph changes over time – edges are added and removed
    • However, shortest path from source to any vertex in the residual graph will only increase over time, never decrease

18-31: Edmonds-Karp Algorithm

• Lemma #1: Shortest path from source \( s \) to any other vertex in residual graph can only increase, not decrease. Proof by contradiction
  • Assume shortest path from source to some other vertex changes after an augmentation
  • Let \( f \) be the flow right before the shortest path decrease, and \( f' \) be the flow right after
  • Let \( v \) be a vertex such that \( \delta_{f'}(s, v) < \delta_f(s, v) \). If there is more than once such \( v \), pick the one with the smallest \( \delta_{f'}(s, v) \) value
  • Let \( p = s \rightarrow \ldots \rightarrow u \rightarrow v \) be the shortest path from \( s \) to \( v \) in \( f' \)

18-32: Edmonds-Karp Algorithm

• Lemma #1: Shortest path from source \( s \) to any other vertex in residual graph can only increase, not decrease. Proof by contradiction
  • Edge \((u, v)\) (last edge on path from \( s \) to \( v \) in \( G_{f'} \)) must not be in \( G_f \)
    • \( \delta_{f'}(s, u) \geq \delta_f(s, u) \)
    • Because \( \delta_{f'}(s, u) < \delta_f(s, v) \), and we picked \( v \) to be the vertex with the smallest \( \delta_{f'}(s, v) \) value that changed
  • If \((u, v) \in G_f\)

\[
\delta_f(s, v) \leq \delta_f(s, u) + 1
\]

\[
\leq \delta_{f'}(s, u) + 1
\]

\[
\leq \delta_{f'}(s, v)
\]

18-33: Edmonds-Karp Algorithm

• Lemma #1: Shortest path from source \( s \) to any other vertex in residual graph can only increase, not decrease. Proof by contradiction
  • Edge \((u, v)\) must be in \( G_{f'} \) but not in \( G_f \) – so the augmenting path must include \((v, u)\)
  • We always choose shortest paths as our augmenting path
  • Shortest path from \( s \) to \( u \) must include \((v, u)\)

\[
\delta_f(s, v) = \delta_f(s, u) - 1
\]

\[
\leq \delta_{f'}(s, u) - 1
\]

\[
\leq \delta_{f'}(s, v) - 2
\]
• Contradiction!

18-34: **Edmonds-Karp Algorithm**

• If we always pick the shortest augmenting path, no more than \(|V| \cdot |E|\) iterations:
  
  • An edge on an augmenting path is critical if it is removed when the flow is augmented (why must there always be at least one critical edge)?
  
  • Each edge can only be critical at most \(|V|/2\) times

18-35: **Edmonds-Karp Algorithm**

• Each edge can only be critical at most \(|V|/2\) times

  • When edge \((u, v)\) is critical:
    
    • \(\delta_f(s, v) = \delta_f(s, u) + 1\)
    
    • Critical edge is removed – before it can become critical again, it must be added back by some augmenting path – that path must contain edge \((u, v)\)
    
    • Let \(f'\) be the flow when the edge is added back.
      
      \[
      \delta_{f'}(s, u) = \delta_{f'}(s, v) + 1 \\
      \geq \delta_f(s, v) + 1 \\
      = \delta_f(s, u) + 1 + 1
      \]

  • If an edge \((u, v)\) becomes critical twice, the shortest path from \(s\) to \(u\) must increase by 2
  
  • Each edge can only be critical \(|V|/2\) times

18-36: **Edmonds-Karp Algorithm**

• Each edge can only be critical at most \(|V|/2\) times

  • Let \(f'\) be the flow when the edge is added back.
    
    \[
    \delta_{f'}(s, u) = \delta_{f'}(s, v) + 1 \\
    \geq \delta_f(s, v) + 1 \\
    = \delta_f(s, u) + 1 + 1
    \]

  • If an edge \((u, v)\) becomes critical twice, the shortest path from \(s\) to \(u\) must increase by 2
  
  • Each edge can only be critical \(|V|/2\) times

18-37: **Edmonds-Karp Algorithm**

• If we always pick the shortest augmenting path, no more than \(|V| \cdot |E|\) iterations:
  
  • An edge on an augmenting path is critical if it is removed when the flow is augmented (why must there always be at least one critical edge)?
  
  • Each edge can only be critical at most \(|V|/2\) times
  
  • \(|E|\) total edges – no more than \(|E| \cdot |V|/2\) iterations
18-38: **Matching Problem**

- Given an undirected graph $G = (V, E)$ a matching $M$ is
  - Subset of edges $E$
  - For any vertex $v \in V$, at most one edge in $M$ is incident to $v$
  - Maximum matching is a matching with largest possible number of edges

18-39: **Matching Problem**

- Bipartite graph
  - Vertices can be divided into two groups, $S_1$ and $S_2$
  - Each edge connects a vertex in $S_1$ with a vertex in $S_2$
18-42: Matching Problem

18-43: Matching Problem
Finding a matching in a bipartite graph can be considered a maximum flow problem. How?
• New algorithm for calculating maximum flow

• Basic idea:
  • Allow vertices to be “overfull” (have more inflow than outflow)
  • Push full capacity out of edges from source
  • Push overflow at each vertex forward to the sink
  • Push excess flow back to source

18-47: **Push-Relabel Algorithms**

• Think of graph as a bunch of water containers connected by pipes.
• We will raise and lower the vertices, and allow water to flow between them
  • Water can only flow from higher vertex to a lower vertex
• Initially, source is at height $|V|$, all other vertices are at height 1
• Full capacity of each pipe out of the source flows to each vertex adjacent to the source

18-48: **Push-Relabel Algorithms**

• Full capacity of each pipe out of the source flows back to each vertex adjacent to the source
  • This causes some vertices to be overfull – inflow greater than outflow
• Raise some vertex whose inflow is greater than outflow, to allow water to flow to different vertices
• Repeat until all vertices (other than the sink, which stays at level 0) are at the same level as the source
• If there are still overfull vertices, continue to raise them so that the extra flow spills back into the source

18-49: **Push-Relabel Algorithms**

![Diagram of a flow network with heights and capacities labeled on the edges.](image)

18-50: **Push-Relabel Algorithms**
18-51: Push-Relabel Algorithms

18-52: Push-Relabel Algorithms
Push-Relabel Algorithms

Heights

18-53: Push-Relabel Algorithms

Heights

18-54: Push-Relabel Algorithms
18-55: Push-Relabel Algorithms

18-56: Push-Relabel Algorithms
18-57: Push-Relabel Algorithms

18-58: Push-Relabel Algorithms
18-59: Push-Relabel Algorithms

18-60: Push-Relabel Algorithms
18-61: Push-Relabel Algorithms

18-62: Push-Relabel Algorithms
18-63: Push-Relabel Algorithms

18-64: Push-Relabel Algorithms
18-65: Push-Relabel Algorithms

18-66: Push-Relabel Algorithms
18-67: Push-Relabel Algorithms

Heights

18-68: Push-Relabel Algorithms
18-69: Push-Relabel Algorithms

18-70: Push-Relabel Algorithms
18-71: Push-Relabel Algorithms

18-72: Push-Relabel Algorithms
18-73: Push-Relabel Algorithms

18-74: Push-Relabel Algorithms
18-75: Push-Relabel Algorithms

18-76: Push-Relabel Algorithms
18-77: Push-Relabel Algorithms

18-78: Push-Relabel Algorithms
18-79: Push-Relabel Algorithms

18-80: Push-Relabel Algorithms
18-81: Push-Relabel Algorithms

18-82: Push-Relabel Algorithms
18-83: **Push-Relabel Algorithms**

**Push**($u, v$)
- Applies when:
  - $u$ is overflowing
  - $c_f(u, v) > 0$
  - $h[u] = h[v] + 1$
- **Action:**
  - Push $\min(\text{overflow}[u], c_f(u, v))$ to $v$

18-84: **Push-Relabel Algorithms**

**Relabel**($u$)
- Applies when:
  - $u$ is overflowing
  - For all $v$ such that $c_f(u, v) > 0$
    - $h[v] \geq h[u]$
- **Action:**
  - $h[u] \leftarrow h[u] + 1$

18-85: **Push-Relabel Algorithms**

**Push-Relabel**($G$)
  - Initialize-Preflow($G, s$)
  - while there exists an applicable push/relabel
    - implement push/relabel

18-86: **Push-Relabel Algorithms**

**Push-Relabel**($G$)
  - Initialize-Preflow($G, s$)
while there exists an applicable push/relabel
implement push/relabel

- Pick the operations (push/relabel) arbitrarily, time is $O(|V|^2E)$
  - (We won’t prove this result, though the proof is in the book)
- Can do better with relabel-to-front
  - Specific ordering for doing push-relabel
  - Time $O(|V|^3)$, also not proven here, proof in text