Algorithm Design Techniques

- Greedy
- Divide and Conquer
- Dynamic Programming
- Network Flows

Network Flow

- Previous topics (greedy, dynamic programming, divide and conquer etc.) were design techniques.
- Network flow relates to a specific class of problems with many applications

- Direct applications:
  - commodities in networks
  - transporting food on the rail network
  - packets on the internet
  - gas through pipes

- Indirect applications:
  - Matching in graphs
  - Airline scheduling
  - Baseball elimination

**Plan:** design and analyze algorithms for max-flow problem, then apply to solve other problems

First, a Story About Flow and Cuts

**Key theme:** flows in a network are intimately related to cuts

Soviet rail network in 1955


Capacity

![Graph](image)

Capacity/Flow

![Graph](image)
Defining Flows

- Flow network
- Directed graph
- Source node \( s \) and target node \( t \)
- Edge capacities \( c(e) \geq 0 \)
- Flow
  - Capacity Constraints: \( 0 \leq f(e) \leq c(e) \) on each edge
  - Conservation Constraints:
    \[
    f^{in}(s) = 0 \ , \ f^{out}(t) = 0 \ , \ \forall v \in V \setminus \{s, t\} \ f^{in}(v) = f^{out}(v)
    \]
    where \( f^{in}(v) = \sum_{e \text{ in to } v} f(e) \) and \( f^{out}(v) = \sum_{e \text{ out of } v} f(e) \)
- Max flow problem: find a flow of maximum value \( v(f) = f^{out}(s) \)

Designing a Max-Flow Algorithm

Something that doesn't work: Repeatedly choose paths and “augment” flow on those paths until we can no longer do so

Residual Graph

Residual graph: data structure to identify opportunities to push more flow on edges with leftover capacity or undo flow on edges already carrying flow.

Original edge \( e = (u, v) \in E \)

- Flow \( f(e) \)
- Capacity \( c(e) \)

Forward residual edge

- \( e = (u, v) \)
- residual capacity \( c(e) - f(e) \)

Backward residual edge

- if \( f(e) > 0 \), create edge \( e' = (v, u) \)
- residual capacity \( f(e) \)

Capacity

Capacity/Flow
Residual Graph

Augmenting Path

Revised Idea: use paths in the residual graph to augment flow

\[
\text{Augment}(f, P) \\
\text{Let } b = \text{bottleneck}(P, f) \\
\text{for edge } e = (u, v) \text{ in } P \text{ do} \\
\quad \text{if } e \text{ is a forward edge then} \\
\quad \quad f(e) = f(e) + b \quad \Rightarrow \text{increase flow on forward edges} \\
\quad \text{else} \\
\quad \quad f(e) = f(e) - b \quad \Rightarrow \text{decrease flow on backward edges} \\
\text{end if} \\
\text{end for}
\]

Capacity/Flow

Residual

Augmenting Path

Old Flow
New Flow

Ford-Fulkerson Algorithm

Repeatedly find augmenting paths in the residual graph and use them to augment flow!

Ford-Fulkerson(G, s, t)
  ▷ Initially, no flow
  Initialize f(e) = 0 for all edges e
  Initialize G_f = G
  ▷ Augment flow as long as it is possible
  while there exists an s-t path P in G_f do
    f = Augment(f, P)
    update G_f
  end while
  return f

Ford-Fulkerson Analysis

Step 1: F-F returns a flow

Claim: If f is a flow then f' = Augment(f, P) is also a flow.

Proof idea. Verify two conditions for f' to be a flow:

▷ f' satisfies capacity constraints: We add at most c(e) − f(e) flow
  along a forward edge that already has f(e) flow so flow doesn’t
  increase above c(e). We add at most f(e) along a backwards
  edge and hence flow doesn’t decrease below 0.

▷ f' satisfies flow conservation: the extra flow into a node equals
  the extra flow out of a node and hence flow is still conserved.

Step 2: Termination and Running Time

Assumption: All capacities are integers. By nature of F-F, all flow
values and residual capacities remain integers during the algorithm.

Running time:

▷ In each F-F iteration, flow increases by at least 1. Therefore,
  number of iterations is at most v(f*), where f* is the final flow.
▷ Let C be the total capacity of edges leaving source s
▷ Then v(f*) ≤ C.
▷ So F-F terminates in at most C iterations

Running time per iteration? O(m + n) to find an augmenting path

Step 3: F-F returns a maximum flow

We will prove this by establishing a deep connection between flows
and cuts in graphs: the max-flow min-cut theorem.

▷ An s-t cut (A, B) is a partition of the nodes into sets A and B
  where s ∈ A, t ∈ B
▷ Capacity of cut (A, B) equals
  \[ c(A, B) = \sum_{e \text{ from } A \text{ to } B} c(e) \]

▷ Flow across a cut (A, B) equals
  \[ f(A, B) = \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ into } A} f(e) \]
Flow Value Lemma

First relationship between cuts and flows

Lemma: let \( f \) be any flow and \((A, B)\) be any \(s-t\) cut. Then

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

Basic idea of proof is to use conservation of flow: all the flow out of \( s \) must leave \( A \) eventually.

Corollary: Cuts and Flows

Really important corollary of flow-value lemma

Corollary: Let \( f \) be any \( s-t \) flow and let \((A, B)\) be any \( s-t \) cut. Then \( v(f) \leq c(A, B) \).

Proof:

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

Implies that if there’s a flow \( f^* \) and cut \((A^*, B^*)\) with \( v(f^*) = c(A^*, B^*) \), then \( f^* \) is a max flow and \((A^*, B^*)\) is a min cut.

F-F returns a maximum flow

Theorem: The \( s-t \) flow \( f^* \) returned by F-F is a maximum flow.

- Since \( f^* \) is the final flow there are no residual paths in \( G_{f^*} \).
- Let \((A^*, B^*)\) be the \( s-t \) cut where \( A^* \) consists of all nodes reachable from \( s \) in the residual graph.
- Then \( v(f) = f(A^*, B^*) = \sum_{e \text{ out of } A^*} f(e) - \sum_{e \text{ into } A^*} f(e) \).
- Any edge out of \( A^* \) must have \( f(e) = c(e) \) otherwise there would be more nodes than just \( A^* \) that reachable from \( s \).
- Any edge into \( A^* \) must have \( f(e) = 0 \) otherwise there would be more nodes than just \( A^* \) that reachable from \( s \).
- Therefore \( v(f) = f(A^*, B^*) = \sum_{e \text{ out of } A^*} f(e) - \sum_{e \text{ into } A^*} f(e) = \sum_{e \text{ out of } A^*} c(e) = c(A^*, B^*) \).