6.4 Maximum Flow

- introduction
- Ford-Fulkerson algorithm
- maxflow-mincut theorem
- running time analysis
- Java implementation
- applications

Min cut problem

**Input.** An edge-weighted digraph, source vertex $s$, and target vertex $t$.

Each edge has a positive capacity.

**Def.** A $s$-$t$ cut (cut) is a partition of the vertices into two disjoint sets, with $s$ in one set $A$ and $t$ in the other set $B$.

**Def.** Its capacity is the sum of the capacities of the edges from $A$ to $B$.

![Diagram of a digraph with capacities](image)
**MinCut Problem**

**Def.** A st-cut (cut) is a partition of the vertices into two disjoint sets, with $s$ in one set $A$ and $t$ in the other set $B$.

**Def.** Its capacity is the sum of the capacities of the edges from $A$ to $B$.

**Minimum st-cut (mincut) problem.** Find a cut of minimum capacity.

---

**MinCut Application (1950s)**

"Free world" goal. Cut supplies (if cold war turns into real war).

---

**Potential MinCut Application (2010s)**

Government-in-power's goal. Cut off communication to set of people.
**Maxflow problem**

**Input.** An edge-weighted digraph, source vertex \( s \), and target vertex \( t \).

- Each edge has a positive capacity.

**Def.** An *st-flow (flow)* is an assignment of values to the edges such that:
  - Capacity constraint: \( 0 \leq \text{edge's flow} \leq \text{edge's capacity} \).
  - Local equilibrium: inflow = outflow at every vertex (except \( s \) and \( t \)).

**Def.** The value of a flow is the inflow at \( t \).

We assume no edges point to \( s \) or from \( t \).

**Def.** The value of a flow is the inflow at \( t \).

**Maximum st-flow (maxflow) problem.** Find a flow of maximum value.
Maxflow application (1950s)

**Soviet Union goal.** Maximize flow of supplies to Eastern Europe.

"Free world" goal. Maximize flow of information to specified set of people.

Summary

**Input.** A weighted digraph, source vertex $s$, and target vertex $t$.

**Mincut problem.** Find a cut of minimum capacity.

**Maxflow problem.** Find a flow of maximum value.

Remarkable fact. These two problems are dual!
**Ford-Fulkerson algorithm**

**Initialization.** Start with 0 flow.

**Idea: increase flow along augmenting paths**

**Augmenting path.** Find an undirected path from $s$ to $t$ such that:
- Can increase flow on forward edges (not full).
- Can decrease flow on backward edge (not empty).

**2nd augmenting path**

**Augmenting path.** Find an undirected path from $s$ to $t$ such that:
- Can increase flow on forward edges (not full).
- Can decrease flow on backward edge (not empty).
Idea: increase flow along augmenting paths

Augmenting path. Find an undirected path from $s$ to $t$ such that:
- Can increase flow on forward edges (not full).
- Can decrease flow on backward edge (not empty).

4th augmenting path

Termination. All paths from $s$ to $t$ are blocked by either a
- Full forward edge.
- Empty backward edge.

Ford-Fulkerson algorithm

- Start with 0 flow.
- While there exists an augmenting path:
  - find an augmenting path
  - compute bottleneck capacity
  - increase flow on that path by bottleneck capacity

Questions.
- How to compute a mincut?
- How to find an augmenting path?
- If FF terminates, does it always compute a maxflow?
- Does FF always terminate? If so, after how many augmentations?
**Def.** The net flow across a cut \((A, B)\) is the sum of the flows on its edges from \(A\) to \(B\) minus the sum of the flows on its edges from \(B\) to \(A\).

**Flow-value lemma.** Let \(f\) be any flow and let \((A, B)\) be any cut. Then, the net flow across \((A, B)\) equals the value of \(f\).

\[
\text{net flow across cut } = \sum_{u \in A} f^+ (u) - \sum_{v \in B} f^+ (v) = \text{value of } f
\]
Relationship between flows and cuts

**Weak duality.** Let \( f \) be any flow and let \( (A, B) \) be any cut. Then, the value of the flow \( \leq \) the capacity of the cut.

**Pf.** Value of flow \( f = \) net flow across cut \( (A, B) \leq \) capacity of cut \( (A, B) \).

\[
\text{value of flow} = 27 \\
\text{capacity of cut} = 30
\]

Maxflow-mincut theorem

**Augmenting path theorem.** A flow \( f \) is a maxflow iff no augmenting paths.

**Maxflow-mincut theorem.** Value of the maxflow = capacity of mincut.

**Pf.** The following three conditions are equivalent for any flow \( f \):

i. There exists a cut whose capacity equals the value of the flow \( f \).

ii. \( f \) is a maxflow.

iii. There is no augmenting path with respect to \( f \).

[ \[ i \Rightarrow ii \] ]

- Suppose that \( (A, B) \) is a cut with capacity equal to the value of \( f \).
- Then, the value of any flow \( f' \leq \) capacity of \( (A, B) = \) value of \( f \).
- Thus, \( f \) is a maxflow.

[ \[ ii \Rightarrow iii \] ]

- We prove contrapositive: \( \sim iii \Rightarrow \sim ii \).

- Suppose that there is an augmenting path with respect to \( f \).
- Can improve flow \( f \) by sending flow along this path.
- Thus, \( f \) is not a maxflow.

[ \[ iii \Rightarrow i \] ]

- Suppose that there is no augmenting path with respect to \( f \).
- Let \( (A, B) \) be a cut where \( A \) is the set of vertices connected to \( s \) by an undirected path with no full forward or empty backward edges.
- By definition, \( s \) is in \( A \); since no augmenting path, \( t \) is in \( B \).
- Capacity of cut = net flow across cut = value of flow \( f \).
Computing a mincut from a maxflow

To compute mincut \((A, B)\) from maxflow \(f\):

- By augmenting path theorem, no augmenting paths with respect to \(f\).
- Compute \(A\) = set of vertices connected to \(s\) by an undirected path
  with no full forward or empty backward edges.

---

Ford-Fulkerson algorithm

- Start with 0 flow.
- While there exists an augmenting path:
  - find an augmenting path
  - compute bottleneck capacity
  - increase flow on that path by bottleneck capacity

Questions.

- How to compute a mincut? **Easy.**
- How to find an augmenting path? **BFS works well.**
- If FF terminates, does it always compute a maxflow? **Yes.**
- Does FF always terminate? If so, after how many augmentations?

---

Ford-Fulkerson algorithm with integer capacities

**Important special case.** Edge capacities are integers between 1 and \(U\).

**Invariant.** The flow is **integer-valued** throughout Ford-Fulkerson.

**Pf.** [by induction]
- Bottleneck capacity is an integer.
- Flow on an edge increases/decreases by bottleneck capacity.

**Proposition.** Number of augmentations \(\leq\) the value of the maxflow.

**Pf.** Each augmentation increases the value by at least 1.

**Integrality theorem.** There exists an integer-valued maxflow.

**Pf.** Ford-Fulkerson terminates and maxflow that it finds is integer-valued.
Bad case for Ford-Fulkerson

**Bad news.** Even when edge capacities are integers, number of augmenting paths could be equal to the value of the maxflow.

**initialize with 0 flow**

1st iteration

2nd iteration

3rd iteration
Bad case for Ford-Fulkerson

**Bad news.** Even when edge capacities are integers, number of augmenting paths could be equal to the value of the maxflow.

---

![Diagram](image1)

4th iteration

---

![Diagram](image2)

99th iteration

---

![Diagram](image3)

200th iteration
Bad case for Ford-Fulkerson

**Bad news.** Even when edge capacities are integers, number of augmenting paths could be equal to the value of the maxflow. Can be exponential in input size

**Good news.** This case is easily avoided. [use shortest/fattest path]

---

How to choose augmenting paths?

**FF performance depends on choice of augmenting paths.**

<table>
<thead>
<tr>
<th>augmenting path</th>
<th>number of paths</th>
<th>implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>shortest path</td>
<td>( \leq \frac{1}{2} E V )</td>
<td>queue (BFS)</td>
</tr>
<tr>
<td>fattest path</td>
<td>( \leq E \ln(E U) )</td>
<td>priority queue</td>
</tr>
<tr>
<td>random path</td>
<td>( \leq E U )</td>
<td>randomized queue</td>
</tr>
<tr>
<td>DFS path</td>
<td>( \leq E U )</td>
<td>stack (DFS)</td>
</tr>
</tbody>
</table>

Digraph with \(V\) vertices, \(E\) edges, and integer capacities between 1 and \(U\)

---

**6.4 Maximum Flow**

- Introduction
- Ford-Fulkerson algorithm
- Maxflow-mincut theorem
- Running time analysis
- Java implementation
- Applications

**Flow network representation**

**Flow edge data type.** Associate flow \(f_e\) and capacity \(c_e\) with edge \(e = v \rightarrow w\).

**Flow network data type.** Need to process edge \(e = v \rightarrow w\) in either direction: Include \(e\) in both \(v\) and \(w\)'s adjacency lists.

**Residual capacity.**
- Forward edge: residual capacity = \(c_e - f_e\).
- Backward edge: residual capacity = \(f_e\).

**Augment flow.**
- Forward edge: add \(\Delta\).
- Backward edge: subtract \(\Delta\).
Flow network representation

**Residual network.** A useful view of a flow network.

Key point. Augmenting path in original network is equivalent to directed path in residual network.

**Flow edge API**

```java
public class FlowEdge {
    // from and to
    private final int v, w;
    // capacity
    private final double capacity;
    // flow
    private double flow;

    public FlowEdge(int v, int w, double capacity) {
        this.v = v;
        this.w = w;
        this.capacity = capacity;
    }

    public int from() { return v; }
    public int to() { return w; }
    public double capacity() { return capacity; }
    public double flow() { return flow; }

    public int other(int vertex) {
        if (vertex == v) return w;
        else if (vertex == w) return v;
        else throw new RuntimeException("Illegal endpoint");
    }

    public double residualCapacityTo(int vertex) {
        if (vertex == v) return flow;
        else if (vertex == w) return capacity - flow;
        else throw new IllegalArgumentException();
    }

    public void addResidualFlowTo(int vertex, double delta) {
        if (vertex == v) flow -= delta;
        else if (vertex == w) flow += delta;
        else throw new IllegalArgumentException();
    }
}
```

**Flow edge: Java implementation**

```java
public class FlowEdge {
    private final int v, w;  // from and to
    private final double capacity;  // capacity
    private double flow;  // flow

    public FlowEdge(int v, int w, double capacity) {
        this.v = v;
        this.w = w;
        this.capacity = capacity;
    }

    public int from() { return v; }
    public int to() { return w; }
    public double capacity() { return capacity; }
    public double flow() { return flow; }

    public int other(int vertex) {
        if (vertex == v) return w;
        else if (vertex == w) return v;
        else throw new RuntimeException("Illegal endpoint");
    }

    public double residualCapacityTo(int vertex) {
        if (vertex == v) return flow;
        else if (vertex == w) return capacity - flow;
        else throw new IllegalArgumentException();
    }

    public void addResidualFlowTo(int vertex, double delta) {
        if (vertex == v) flow -= delta;
        else if (vertex == w) flow += delta;
        else throw new IllegalArgumentException();
    }
}
```

**Flow edge: Java implementation (continued)**
Flow network API

```java
public class FlowNetwork {
    FlowNetwork(int V) { /* create an empty flow network with V vertices */ }
    FlowNetwork(In in) { /* construct flow network input stream */ }
    void addEdge(FlowEdge e) { /* add edge e to this flow network */ }
    Iterable<FlowEdge> adj(int v) { /* forward and backward edges incident to v */ }
    Iterable<FlowEdge> edges() { /* all edges in this flow network */ }
    int V() { /* number of vertices */ }
    int E() { /* number of edges */ }
    String toString() { /* string representation */ }
}
```

**Conventions.** Allow self-loops and parallel edges.

Flow network: adjacency-lists representation

Maintain vertex-indexed array of FlowEdge lists (use Bag abstraction).

```java
tinyFN.txt
```

Ford-Fulkerson: Java implementation

```java
public class FordFulkerson {
    private final int V;
    private FlowEdge[] adj;
    private Bag<Edge> edgeTo;
    private double value; // value of flow

    public FordFulkerson(FlowNetwork G, int s, int t) {
        value = 0.0;
        while (hasAugmentingPath(G, s, t)) {
            double bottle = Double.POSITIVE_INFINITY;
            for (int v = s; v != t; v = edgeTo[v].other(s))
                bottle = Math.min(bottle, edgeTo[v].residualCapacityTo(s));
            for (int v = s; v != t; v = edgeTo[v].other(s))
                edgeTo[v].addResidualFlowTo(s, bottle);
            value += bottle;
        }
    }

    public double hasAugmentingPath(FlowNetwork G, int s, int t) {
        /* See next slide. */
        return value;
    }
}
```
Finding a shortest augmenting path (cf. breadth-first search)

```java
private boolean hasAugmentingPath(FlowNetwork G, int s, int t) {
    edgeTo = new FlowEdge(G.V());
    marked = new boolean[G.V()];
    Queue<Integer> queue = new Queue<Integer>();
    queue.enqueue(s);
    marked[s] = true;
    while (!queue.isEmpty()) {
        int v = queue.dequeue();
        for (FlowEdge e : G.adj(v)) {
            int w = e.other(v);
            if (e.residualCapacityTo(w) > 0 && !marked[w]) {
                edgeTo[w] = e;
                marked[w] = true;
                queue.enqueue(w);
            }
        }
    }
    return marked[t];
}
```

Maxflow and mincut applications

Maxflow/mincut is a widely applicable problem-solving model.

- Data mining.
- Open-pit mining.
- Bipartite matching.
- Network reliability.
- Baseball elimination.
- Image segmentation.
- Network connectivity.
- Distributed computing.
- Security of statistical data.
- Egalitarian stable matching.
- Multi-camera scene reconstruction.
- Sensor placement for homeland security.
- Many, many, more.

Bipartite matching problem

N students apply for N jobs.

Each gets several offers.

Is there a way to match all students to jobs?

bipartite matching problem

<table>
<thead>
<tr>
<th>Student</th>
<th>Company</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>Adobe</td>
</tr>
<tr>
<td>Bob</td>
<td>Amazon</td>
</tr>
<tr>
<td>Carol</td>
<td>Google</td>
</tr>
<tr>
<td>Dave</td>
<td>Facebook</td>
</tr>
<tr>
<td>Eliza</td>
<td>Amazon</td>
</tr>
<tr>
<td>Yahoo</td>
<td></td>
</tr>
<tr>
<td>Alice</td>
<td>Bob</td>
</tr>
<tr>
<td>Alice</td>
<td>Carol</td>
</tr>
<tr>
<td>Eliza</td>
<td>Alice</td>
</tr>
<tr>
<td>Eliza</td>
<td>Carol</td>
</tr>
<tr>
<td>Bob</td>
<td>Alice</td>
</tr>
<tr>
<td>Eliza</td>
<td>Bob</td>
</tr>
<tr>
<td>Dave</td>
<td>Alice</td>
</tr>
<tr>
<td>Eliza</td>
<td>Carol</td>
</tr>
</tbody>
</table>

Liver and hepatic vascularization segmentation
Bipartite matching problem

Given a bipartite graph, find a perfect matching.

Network flow formulation of bipartite matching

1-1 correspondence between perfect matchings in bipartite graph and integer-valued maxflows of value $N$.

What the mincut tells us

Goal. When no perfect matching, explain why.
What the mincut tells us

**MinCut.** Consider mincut $(A, B)$.
- Let $S =$ students on $s$ side of cut.
- Let $T =$ companies on $s$ side of cut.
- Fact: $|S| > |T|$; students in $S$ can be matched only to companies in $T$.

**Bottom line.** When no perfect matching, mincut explains why.

---

**Baseball elimination problem**

**Q.** Which teams have a chance of finishing the season with the most wins?

<table>
<thead>
<tr>
<th>i</th>
<th>team</th>
<th>wins</th>
<th>losses</th>
<th>to play</th>
<th>ATL</th>
<th>PHI</th>
<th>NYM</th>
<th>MON</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Atlanta</td>
<td>83</td>
<td>71</td>
<td>8</td>
<td>-</td>
<td>1</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>Philly</td>
<td>80</td>
<td>79</td>
<td>3</td>
<td>1</td>
<td>-</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>New York</td>
<td>78</td>
<td>78</td>
<td>6</td>
<td>6</td>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>Montreal</td>
<td>77</td>
<td>82</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>-</td>
</tr>
</tbody>
</table>

**Philadelphia is mathematically eliminated.**
- Philadelphia finishes with $\leq 83$ wins.
- Either New York or Atlanta will finish with $\geq 84$ wins.

**Observation.** Answer depends not only on how many games already won and left to play, but on whom they’re against.

**Baseball elimination problem**

**Q.** Which teams have a chance of finishing the season with the most wins?

<table>
<thead>
<tr>
<th>i</th>
<th>team</th>
<th>wins</th>
<th>losses</th>
<th>to play</th>
<th>NYY</th>
<th>BAL</th>
<th>BOS</th>
<th>TOR</th>
<th>DET</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>New York</td>
<td>75</td>
<td>59</td>
<td>28</td>
<td>-</td>
<td>3</td>
<td>8</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>Baltimore</td>
<td>71</td>
<td>63</td>
<td>28</td>
<td>3</td>
<td>-</td>
<td>2</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>Boston</td>
<td>69</td>
<td>66</td>
<td>27</td>
<td>8</td>
<td>2</td>
<td>-</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>Toronto</td>
<td>63</td>
<td>72</td>
<td>27</td>
<td>7</td>
<td>7</td>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>Detroit</td>
<td>49</td>
<td>86</td>
<td>27</td>
<td>3</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
</tbody>
</table>

**Detroit is mathematically eliminated.**
- Detroit finishes with $\leq 76$ wins.
- Wins for $R = \{ NYY, BAL, BOS, TOR \} = 278$.
- Remaining games among $\{ NYY, BAL, BOS, TOR \} = 3 + 8 + 7 + 2 + 7 = 27$.
- Average team in $R$ wins $305/4 = 76.25$ games.
Baseball elimination problem: maxflow formulation

**Intuition.** Remaining games flow from $s$ to $t$.

![Diagram showing the maxflow formulation for the baseball elimination problem.]

- **Fact.** Team 4 not eliminated iff all edges pointing from $s$ are full in maxflow.

Maximum flow algorithms: theory

(Yet another) holy grail for theoretical computer scientists.

<table>
<thead>
<tr>
<th>Year</th>
<th>Method</th>
<th>Worst Case</th>
<th>Discovered By</th>
</tr>
</thead>
<tbody>
<tr>
<td>1951</td>
<td>simplex</td>
<td>$E^3 U$</td>
<td>Dantzig</td>
</tr>
<tr>
<td>1955</td>
<td>augmenting path</td>
<td>$E^2 U$</td>
<td>Ford-Fulkerson</td>
</tr>
<tr>
<td>1970</td>
<td>shortest augmenting path</td>
<td>$E^3$</td>
<td>Dinitz, Edmonds-Karp</td>
</tr>
<tr>
<td>1970</td>
<td>fastest augmenting path</td>
<td>$E^2 \log E \log(EU)$</td>
<td>Dinitz, Edmonds-Karp</td>
</tr>
<tr>
<td>1977</td>
<td>blocking flow</td>
<td>$E^{5/2}$</td>
<td>Cherkasky</td>
</tr>
<tr>
<td>1978</td>
<td>blocking flow</td>
<td>$E^{7/3}$</td>
<td>Galil</td>
</tr>
<tr>
<td>1983</td>
<td>dynamic trees</td>
<td>$E^{2} \log E$</td>
<td>Sleator-Tarjan</td>
</tr>
<tr>
<td>1985</td>
<td>capacity scaling</td>
<td>$E^{2} \log U$</td>
<td>Gabow</td>
</tr>
<tr>
<td>1997</td>
<td>length function</td>
<td>$E^{3/2} \log E \log U$</td>
<td>Goldberg-Rao</td>
</tr>
<tr>
<td>2012</td>
<td>compact network</td>
<td>$E^{2} / \log E$</td>
<td>Orlin</td>
</tr>
</tbody>
</table>

maxflow algorithms for sparse digraphs with $E$ edges, integer capacities between 1 and $U$

Summary

- **Mincut problem.** Find an $st$-cut of minimum capacity.
- **Maxflow problem.** Find an $st$-flow of maximum value.
- **Duality.** Value of the maxflow = capacity of mincut.

Proven successful approaches.
- Ford-Fulkerson (various augmenting-path strategies).
- Preflow-push (various versions).

Open research challenges.
- Practice: solve real-word maxflow/mincut problems in linear time.
- Theory: prove it for worst-case inputs.
- Still much to be learned!
6.4 Maximum Flow

- introduction
- Ford-Fulkerson algorithm
- maxflow-mincut theorem
- running time analysis
- Java implementation
- applications