6.4 Maximum Flow

- overview
- Ford-Fulkerson
- implementations
- applications
**Min-cut problem**

*Given.* A weighted digraph with identified source $s$ and target $t$.

**Def.** A cut is a partition of the vertices into two disjoint sets.

**Def.** An st-cut is a cut that places $s$ in one of its sets ($C_s$) and $t$ in the other ($C_t$).

**Def.** An st-cut’s weight is the sum of the weights of its st-crossing edges.

**Minimum st-cut (mincut) problem.** Find an st-cut of minimal weight.

![Diagram of Min-cut problem]

*Note:* don’t count edges from $C_t$ to $C_s$
Typical mincut application

Find cheapest way to cut connection between s and t.

www.blog.spoongraphics.co.uk/tutorials/creating-road-maps-in-adobe-illustrator
Mincut application (1950s)

Rail network connecting Soviet Union with Eastern European countries

“Free world” goal: Know how to cut supplies if cold war turns into real war (map declassified by Pentagon in 1999).
Potential mincut application (2010s)

Facebook graph

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

Flow network.
- Weighted digraph with a source $s$ (indegree 0) and sink $t$ (outdegree 0)
- An edge’s weight is its capacity (positive)
- Add additional flow variable to each edge (no greater than its capacity)

**Maximum st-flow (maxflow) problem:** Assign flows to edges that
- Maintain local equilibrium: inflow = outflow at every vertex (except $s$ and $t$).
- Maximize total flow into $t$. 
A physical model

Oil flowing in pipes

local equilibrium at each vertex

all flow goes down (just to simplify diagrams)

room for improvement?
Typical maxflow application

Find best way to distribute goods from s to t.

www.blog.spoongraphics.co.uk/tutorials/creating-road-maps-in-adobe-illustrator
Maxflow application (1950s)

Rail network connecting Soviet Union with Eastern European countries

Soviet Union goal: Know how to maximize flow of supplies to Eastern Europe.
Potential mincut application (2010s)

Facebook graph

“Free world” goal: Maximize flow of information to specified set of people.
Overview (summary)

**Given.** A weighted digraph, 

source s and target t.

**Minimum st-cut (mincut) problem.** Find an st-cut of minimal weight.

**Maximum st-flow (maxflow) problem:** Assign flows to edges that

- Maintain local equilibrium: inflow = outflow at every vertex (except s and t).
- Maximize total flow into t.

**Remarkable fact.** These two problems are equivalent!

- Two very rich algorithmic problems
- Cornerstone problems in combinatorial optimization
- Beautiful mathematical duality
Maxflow / mincut applications

Maxflow/mincut is a widely applicable problem-solving model

- Data mining.
- Open-pit mining.
- Project selection.
- Image processing.
- Airline scheduling.
- Bipartite matching.
- Baseball elimination.
- Distributed computing.
- Egalitarian stable matching.
- Security of statistical data.
- Network connectivity/reliability.
- Many many more . . .
› overview
› APIs
› Ford Fulkerson
› implementations
› applications
APIs (cf. `EdgeWeightedDigraph`, `DirectedEdge`)

Flow network API

```java
public class FlowNetwork{
    FlowNetwork(int V)
    FlowNetwork(In in)
    int V()
    int E()
    void addEdge(FlowEdge e)
    Iterable<FlowEdge> adj(int v)
    Iterable<FlowEdge> edges()
    String toString()
}
```

Flow edge API

```java
public class FlowEdge{
    FlowEdge(int v, int w, double cap)
    int from()
    int to()
    int other(int v)
    double capacity()
    double flow()
    double residualCapacityTo(int v)
    double addFlowTo(int v, double delta)
    String toString()
}
```

- `empty V-vertex flow network`
- `construct from input stream`
- `number of vertices`
- `number of edges`
- `add e to this flow network`
- `edges pointing from v`
- `all edges in this flow network`
- `string representation`

- `vertex this edge points from`
- `vertex this edge points to`
- `other endpoint`
- `capacity of this edge`
- `flow in this edge`
- `residual capacity toward v`
- `add delta flow toward v`
- `string representation`

manipulate flow values (stay tuned)
public class FlowEdge {
    private final int v;             // from
    private final int w;             // to
    private final double capacity;   // capacity
    private double flow;             // flow

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

    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) {...}
    public void addResidualFlowTo(int vertex, double delta) {...}
}
A flow network is an array of bags of flow edges.

Flow network representation

tinyFN.txt

V

E

adj[]

Flow network representation

References to the same Edge object

Bag objects
public class FlowNetwork
{
    private final int V;
    private int E;
    private Bag<FlowEdge>[] adj;

    public FlowNetwork(int V)
    {
        this.V = V;
        this.E = 0;
        adj = (Bag<FlowEdge>[])
                new Bag[V];
        for (int v = 0; v < V; v++)
            adj[v] = new Bag<FlowEdge>();
    }

    public int V() { return V; }
    public int E() { return E; }

    public void addEdge(FlowEdge e)
    {
        E++;
        int v = e.from();
        int w = e.to();
        adj[v].add(e);
        adj[w].add(e);
    }

    public Iterable<FlowEdge> adj(int v)
    { return adj[v]; }
}

array of bags of flow edges
constructor
add edge (to both adj lists)
iterator for adjacent edges
Typical client code: check that a flow is feasible

```java
private boolean localEq(FlowNetwork G, int v)
{
    // Check local equilibrium at v.
    double EPSILON = 1E-11;
    double netflow = 0.0;
    for (FlowEdge e : G.adj(v))
        if (v == e.from()) netflow -= e.flow();
        else netflow += e.flow();
    return Math.abs(netflow) < EPSILON;
}
private boolean isFeasible(FlowNetwork G)
{
    for (int v = 0; v < G.V(); v++)
        for (FlowEdge e : G.adj(v))
            if (e.flow() < 0 || e.flow() > e.capacity())
                return false;
    for (int v = 0; v < G.V(); v++)
        if (v != s && v != t && !localEq(G, v))
            return false;
    return true;
}
```

- Check that each flow is nonnegative and no greater than capacity.
- Check local equilibrium at each vertex.
APIs

Ford Fulkerson

implementations

applications
Idea: increase flow along augmenting paths
Idea: increase flow along augmenting paths
Idea: increase flow along augmenting paths
Idea: increase flow along augmenting paths
Idea: increase flow along augmenting paths

Problem: Can get stuck with no way to add more flow to $t$. 
Idea: increase flow along augmenting paths

Problem: Can get stuck with no way to add more flow to \( t \).
Solution: Go backwards along an edge with flow (removing some flow).
Idea: increase flow along augmenting paths

Augmenting paths in general
• increase flow on forward edge (if not full)
• decrease flow on backward edge (if not empty)
Idea: increase flow along augmenting paths

\[ 25 + 3 = 28 \]
Idea: increase flow along augmenting paths

Eventually all paths from $s$ to $t$ are blocked by either a

- full forward edge
- empty backward edge

maxflow value = 28
Ford-Fulkerson algorithm

**Generic method for solving maxflow problem.**
- Start with 0 flow everywhere.
- Find an augmenting path.
- Increase the flow on that path, by as much as possible.
- Repeat until no augmenting paths are left.

Questions.
Q. Does this process give a maximum flow?
   A. Yes! It also finds a mincut (!!). [Classic result]

Q. How do we find an augmenting path?
   A. Easy. Adapt standard graph-searching methods.

Q. How many augmenting paths (does the process even terminate)?
   A. Difficult to know: depends on graph model, search method.
Min cut problem (revisited)

**Given.** A weighted digraph with identified source \( s \) and target \( t \).

**Def.** A **cut** is a partition of the vertices into two disjoint sets.  
**Def.** An **st-cut** is a cut that places \( s \) in one of its sets \( (C_s) \) and \( t \) in the other \( (C_t) \).  
**Def.** An st-cut’s **weight** is the sum of the weights of its st-crossing edges.

**Min cut problem.** Find an st-cut of minimal weight.

---

Note: don’t count edges from \( C_t \) to \( C_s \)
Mincut problem (revisited with slight change in terminology)

**Given.** A flow network with identified source $s$ and target $t$.

**Def.** A cut is a partition of the vertices into two disjoint sets.
**Def.** An st-cut is a cut that places $s$ in one of its sets ($C_s$) and $t$ in the other ($C_t$).
**Def.** An st-cut’s capacity is the sum of the capacities of its st-crossing edges.

**Mincut problem.** Find an st-cut of minimal capacity.

Amazing fact. Mincut and maxflow problems are equivalent.
Relationship between flows and cuts

**Def.** The *flow across* an st-cut is the sum of the flows on its st-crossing edges minus the sum of the flows of its ts-crossing edges.

**Thm.** For any st-flow, the flow across every st-cut equals the value of the flow.

*Pf.* By induction on the size of $C_t$.

- true when $C_t = \{t\}$.
- true by local equilibrium when moving a vertex from $C_s$ to $C_t$

**Corollary 1.** Outflow from $s =$ inflow to $t =$ value.

**Corollary 2.** No st-flow’s value can exceed the capacity of any st-cut.
Relationship between flows and cuts (example)

For any st-flow, the flow across every st-cut equals the value of the flow.
Maxflow-mincut theorem

Thm. The following three conditions are equivalent for any st-flow f:

i. There exists an st-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.

Pf.

i. implies ii. [no flow’s value can exceed any cut’s capacity]

ii. implies iii. by contradiction [aug path would give higher-value flow, so f could not be maximal].

iii. implies i.

$C_s$: set of all vertices connected to s by an undirected path with no full forward or empty backward edges.
$C_t$: all other vertices.

capacity = flow across [st-crossing edges full, ts-crossing edges empty].

= value of f [capacity of any cut = value of f].
Eventually all paths from $s$ to $t$ are blocked by either a
- full forward edge
- empty backward edge

Mincut:
Consider only paths with no full forward or empty backward edges.
$C_s$ is the set of vertices reachable from $s$; $C_t$ is the set of remaining vertices.
Maxflow/mincut application (1950s)

“bottleneck” is mincut (all forward edges full)
value of flow = 30+17+36+16+24+6+10+5+19 = 163,000 tons
Integrality property

Corollary to maxflow-mincut theorem. When capacities are integers, there exists an integer-valued maxflow, and the Ford Fulkerson algorithm finds it.

Pf. Flow increases by augmenting path value, which is either unused capacity in a forward edge or flow in a backwards edge [and always an integer].

Bottom line: Ford-Fulkerson always works when weights are integers.

Note: When weights are not integers, it could converge to the wrong value!
Possible strategies for augmenting paths

**FF algorithm:** any strategy for choosing augmenting paths will give a maxflow. 
[Caveat: Can have convergence problems when weights are not integers.]

- **Shortest path?**
- **DFS path?**
- **Random path?**
- **Fattest path?**

All easy to implement
- Define residual graph
- Find paths in residual graph.

Performance depends on network properties (stay tuned)
Shortest augmenting path
Fattest augmenting path
Random augmenting path
Bad case for Ford-Fulkerson

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

**Good news:** This case is easily avoided [use shortest augmenting path].
first iteration

\[0 + 1 = 1\]
second iteration

1 + 2 = 2

backwards edge
third iteration

2 + 1 = 3
fourth iteration

3 + 4 = 4
\[(2i-1)^{st}\text{ iteration}\]

\[2i - 2 + 1 = 2i - 1\]
(2i)th iteration

2i - 1 + 1 = 2i
Bad case for Ford-Fulkerson

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

**Good news:** This case is easily avoided [use shortest augmenting path].
Flow network representation (revisited)

Residual digraph. Another view of a flow network

Finding an augmenting path is equivalent to finding a path in residual digraph.
public class FlowEdge
{
    private final int v;     // from
    private final int w;     // to
    private final double capacity;  // capacity
    private double flow;     // flow

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

    public void addResidualFlowTo(int vertex, double delta)
    {
        if (vertex == v) flow -= delta;
        else if (vertex == w) flow += delta;
        else throw new RuntimeException("Illegal endpoint");
    }
}
Ford-Fulkerson: Java implementation

```java
public class FordFulkerson {
    private boolean[] marked; // true if s->v path in residual digraph
    private FlowEdge[] edgeTo; // last edge on s->v path
    private double value;

    public FordFulkerson(FlowNetwork G, int s, int t) {
        value = 0;
        while (hasAugmentingPath(G, s, t)) {
            double bottle = Double.POSITIVE_INFINITY;
            for (int v = t; v != s; v = edgeTo[v].other(v))
                bottle = Math.min(bottle, edgeTo[v].residualCapacityTo(v));
            for (int v = t; v != s; v = edgeTo[v].other(v))
                edgeTo[v].addResidualFlowTo(v, bottle);
            value += bottle;
        }
    }

    public double hasAugmentingPath(FlowNetwork G, int s, int t) {
        /* See next slide. */
    }

    public double value() {
        return value;
    }

    public boolean inCut(int v) {
        return marked[v];
    }
}
```
private boolean hasAugmentingPath(FlowNetwork G, int s, int t) {
    edgeTo = new FlowEdge[G.V()];
    marked = new boolean[G.V()];

    Queue<Integer> q = new Queue<Integer>();
    q.enqueue(s);
    marked[s] = true;
    while (!q.isEmpty()) {
        int v = q.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;
                q.enqueue(w);
            }
        }
    }
    return marked[t];
}
Analysis of Ford-Fulkerson (shortest augmenting path)

**Thm.** Ford-Fulkerson (shortest augmenting path) uses at most $EV/2$ augmenting paths.

**Pf.** [see text]

**Cor.** Ford-Fulkerson (shortest augmenting path) examines at most $E^2V/2$ edges.

**Pf.** Each BFS examines at most $E$ edges.
Summary: possible strategies for augmenting paths

All easy to implement
  • Define residual graph
  • Find paths in residual graph.

Shortest path: Use BFS.
DFS path: Use DFS.
Fattest path: Use a PQ, ala shortest paths.
Random path: Use a randomized queue.

Performance depends on network properties
  • how many augmenting paths?
  • how many edges examined to find each augmenting path?
### Analysis of maxflow algorithms

*(Yet another) holy grail for mathematicians/theoretical computer scientists.*

For sparse graphs with $E$ edges, integer capacities (max $U$).

<table>
<thead>
<tr>
<th>year</th>
<th>method</th>
<th>worst case order of growth</th>
<th>discovered by</th>
</tr>
</thead>
<tbody>
<tr>
<td>1951</td>
<td>simplex</td>
<td>$O( E^3 U )$</td>
<td>Dantzig</td>
</tr>
<tr>
<td>1955</td>
<td>augmenting paths</td>
<td>$O( E^2 U )$</td>
<td>Ford-Fulkerson</td>
</tr>
<tr>
<td>1970</td>
<td>shortest aug path</td>
<td>$O( E^3 )$</td>
<td>Edmunds-Karp</td>
</tr>
<tr>
<td>1970</td>
<td>fattest aug path</td>
<td>$O( E^2 \log E \log U )$</td>
<td>Edmunds-Karp</td>
</tr>
<tr>
<td>1973</td>
<td>capacity scaling</td>
<td>$O( E^2 \log U )$</td>
<td>Dinitz-Gabow</td>
</tr>
<tr>
<td>1983</td>
<td>preflow-push</td>
<td>$O( E^2 \log E )$</td>
<td>Sleator-Tarjan</td>
</tr>
<tr>
<td>1997</td>
<td>length function</td>
<td>$\tilde{O}( E^{3/2} )$</td>
<td>Goldberg-Rao</td>
</tr>
<tr>
<td>2011</td>
<td>electrical flow</td>
<td>$\tilde{O}( E^{4/3} )^*$</td>
<td>Christiano-Kelner-Madry-Spielman-Teng</td>
</tr>
<tr>
<td>?</td>
<td></td>
<td>$O( E )$</td>
<td></td>
</tr>
</tbody>
</table>

**Warning:** Worst-case order-of-growth analysis is generally **not useful** for predicting or comparing algorithm performance in practice.

~ ignore log factors
* approximation
O-notation considered harmful (Lecture 2 revisited)

Facebook and Google: Huge sparse graphs are of interest ($10^{10} - 10^{11}$ edges).

Time to solve maxflow:

- **Algorithm A:** $\tilde{O}(E^{3/2})$.
- **Algorithm B:** $\tilde{O}(E^{4/3})$.*

* ignore log factors
* approximation algorithm

Q. Which algorithm should Facebook and Google be interested in?

A. Who knows? These mathematical results are not relevant!

- Upper bound on worst case [may never take stated time].
- Unknown constants [most published maxflow algs never are implemented].
- $E^{1/6}$ savings likely offset by ignored log factors [40-50 vs. 30-40+].
- Performance for practical graph models likely unknown [and not studied].
- Approximation algorithm [cost of accuracy may be too high].
First improvement of fundamental algorithm in 10 years

The max-flow problem, which is ubiquitous in network analysis, scheduling, and logistics, can now be solved more efficiently than ever.

September 27, 2010

The maximum-flow problem, or max flow, is one of the most basic problems in computer science: First solved during preparations for the Berlin airlift, it's a component of many logistical problems and a staple of introductory courses on algorithms. For decades it was a prominent research subject, with new algorithms that solved it more and more efficiently coming out once or twice a year. But as the problem became better understood, the pace of innovation slowed. Now, however, researchers, together with colleagues at... have demonstrated the first improvement of the max-flow algorithm in 10 years.

The max-flow problem is, roughly speaking, to calculate the maximum amount of "stuff" that can move from one end of a network to another, given the capacity limitations of the network's links. The stuff could be data packets traveling over the Internet or boxes of goods traveling over the highways; the links' limitations could be the bandwidth of Internet connections or the average traffic speeds on congested roads.

More technically, the problem has to do with what mathematicians call graphs. A graph is a collection of vertices and edges, which are generally depicted as circles and the lines connecting them. The standard diagram of a communications network is a graph,
The algorithm has not been implemented or tested on graphs the size of the internet (or at all, for that matter). The algorithm would have to be implemented and tested before any claim to immediate practicality could be assessed.

It is likely that simpler approaches involving parallelism will be used in practice.

Moreover, these mathematical results are approximate, ignoring factors that could run into the hundreds for the internet graph.

The algorithm also computes an approximation to the maxflow, not the actual maxflow, and slows down as the approximation improves.

If N is the number of nodes in a graph, and L is the number of links between them, then the execution of the fastest previous max-flow algorithm was proportional to \((N + L)^{3/2}\). The execution of the new algorithm is proportional to \((N + L)^{4/3}\). For a network like the Internet, which has hundreds of billions of nodes, the new algorithm could solve the max-flow problem hundreds of times faster than its predecessor.

The immediate practicality of the algorithm, however, is not what impresses John.

O-notation considered harmful
Summary

**Minimum st-cut (mincut) problem.** Find an st-cut of minimal weight.

**Maximum st-flow (maxflow) problem:** Assign flows to edges that
- Maintain local equilibrium: inflow = outflow at every vertex (except s and t).
- Maximize total flow into t.

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

Open research challenges.
- Practice: Solve maxflow/mincut problems for real networks in linear time.
Maxflow / mincut applications

Maxflow/mincut is a widely applicable problem-solving model

- Data mining.
- Open-pit mining.
- Project selection.
- Image processing.
- Airline scheduling.
- Bipartite matching.
- Baseball elimination.
- Distributed computing.
- Egalitarian stable matching.
- Security of statistical data.
- Network connectivity/reliability.
- Many many more . . .
Bipartite matching problem

N students apply for N jobs

Each get several offers

Is there a way to match all students to jobs?

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Alice</td>
<td>6</td>
<td>Adobe</td>
<td>7</td>
<td>Alice</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Adobe</td>
<td>2</td>
<td>Amazon</td>
<td>8</td>
<td>Bob</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Amazon</td>
<td>3</td>
<td>Facebook</td>
<td>9</td>
<td>Dave</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Bob</td>
<td>4</td>
<td>Amazon</td>
<td>10</td>
<td>Frank</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Adobe</td>
<td>5</td>
<td>Yahoo</td>
<td>11</td>
<td>IBM</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Amazon</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Carol</td>
<td>7</td>
<td>Facebook</td>
<td>12</td>
<td>Yahoo</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Facebook</td>
<td>8</td>
<td>Google</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Google</td>
<td>9</td>
<td></td>
<td>Eliza</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Dave</td>
<td>10</td>
<td>IBM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Adobe</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Amazon</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Eliza</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Google</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>IBM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Frank</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>IBM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Yahoo</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

N students apply for N jobs

Each get several offers

Is there a way to match all students to jobs?
Network flow formulation of bipartite matching

To formulate a bipartite matching problem as a network flow problem
• create s, t, one vertex for each student, and one vertex for each job
• add edge from s to each student
• add edge from each job to t
• add edge from student to each job offered
• give all edges capacity 1
Bipartite matching problem formulated as a network flow problem

1-1 correspondence between maxflow solution and bipartite matching solution
Maxflow solution (FF shortest augmenting path)
Maxflow solution (FF shortest augmenting path)
Maxflow solution (FF shortest augmenting path)
Maxflow solution (FF shortest augmenting path)
Maxflow solution (FF shortest augmenting path)
Maxflow solution (FF shortest augmenting path)

path with back edges
Maxflow solution
**Maxflow solution corresponds directly to matching solution**

<table>
<thead>
<tr>
<th>1</th>
<th>Alice</th>
<th>7</th>
<th>Adobe</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Alice</td>
</tr>
<tr>
<td></td>
<td>Adobe</td>
<td>8</td>
<td>Amazon</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bob</td>
</tr>
<tr>
<td></td>
<td>Amazon</td>
<td>9</td>
<td>Facebook</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Dave</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10</td>
<td>Google</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>IBM</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Bob</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>Amazon</td>
</tr>
<tr>
<td></td>
<td>Adobe</td>
<td></td>
<td>Alice</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bob</td>
</tr>
<tr>
<td></td>
<td>Amazon</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Yahoo</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Carol</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>9</td>
<td>Facebook</td>
</tr>
<tr>
<td></td>
<td>Facebook</td>
<td></td>
<td>Alice</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Carol</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10</td>
<td>Google</td>
</tr>
<tr>
<td></td>
<td>IBM</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Dave</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>Amazon</td>
</tr>
<tr>
<td></td>
<td>Adobe</td>
<td></td>
<td>Eliza</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9</td>
<td>IBM</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Eliza</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>9</td>
<td>Google</td>
</tr>
<tr>
<td></td>
<td>Google</td>
<td></td>
<td>Eliza</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10</td>
<td>IBM</td>
</tr>
<tr>
<td></td>
<td>IBM</td>
<td></td>
<td>Frank</td>
</tr>
<tr>
<td></td>
<td>Yahoo</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Frank</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>IBM</td>
</tr>
<tr>
<td></td>
<td>IBM</td>
<td></td>
<td>Eliza</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>10</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>

Alice — Amazon
Bob — Yahoo
Carol — Facebook
Dave — Adobe
Eliza — Google
Frank — IBM
Overview (summary)

Given. A weighted digraph, source s and target t.

Minimum st-cut (mincut) problem. Find an st-cut of minimal weight.

Maximum st-flow (maxflow) problem: Assign flows to edges that
• Maintain local equilibrium: inflow = outflow at every vertex (except s and t).
• Maximize total flow into t.

Remarkable fact. These two problems are equivalent!
• Two very rich algorithmic problems
• Cornerstone problems in combinatorial optimisation
• Beautiful mathematical duality
• Still much to be learned!