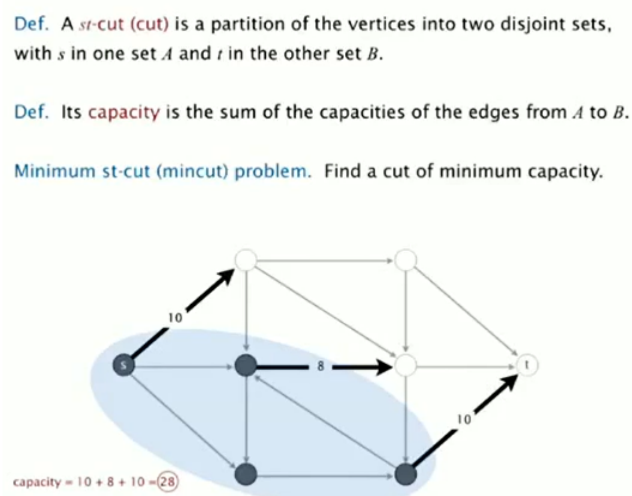
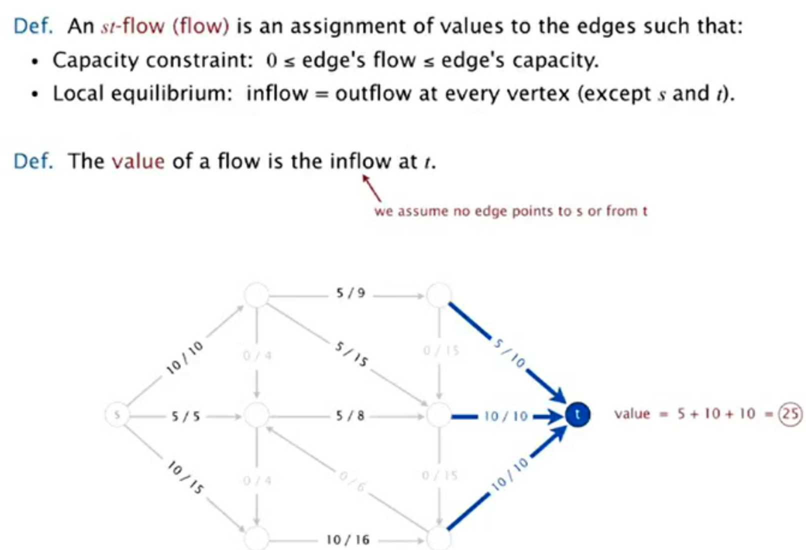
Maximum Flow

Introduction

Mincut Problem



Maxflow Problem



Ford-Fulkerson Algorithm

Basic

- 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)
- 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?---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 augmentation?---Yes, provided that augmenting paths are chosen carefully. Require clever analysis.

Maxflow-mincut Theorem

Relationship between Flow and Cuts

- flow value

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 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.

- week duality

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

Pf. Value of flow f $\underset{\text{flow-value lemma}}{=}$ net flow across cut (A, B) $\underset{\text{flow bounded by capacity}}{\leq}$ capacity of cut (A, B).

- maxflow-mincut theorem

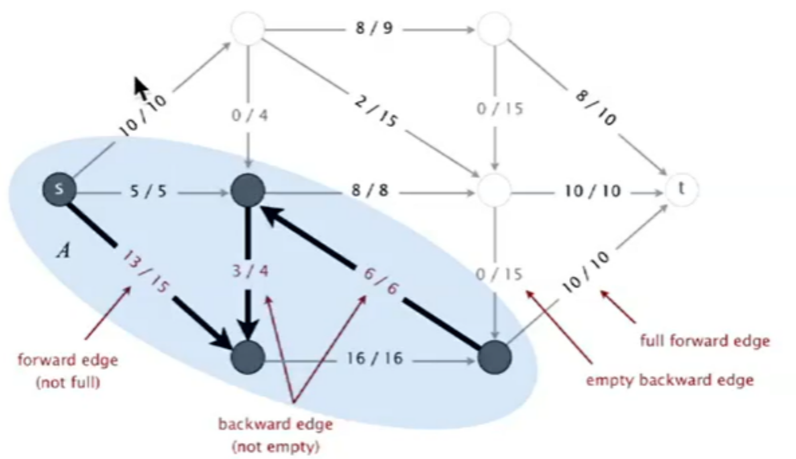
Augmenting path theorem. A flow f is a maxflow iff no augmenting paths.
Maxflow-mincut theorem. Value of the maxflow = capacity of mincut.

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.

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.



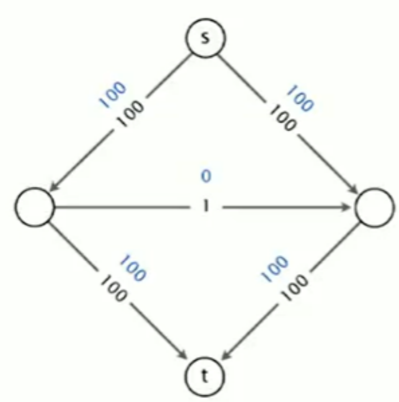
Running Time Analysis

Bad Case

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]

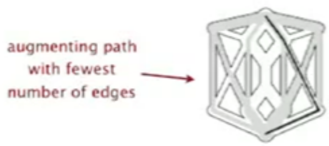


Choose Augmenting Path

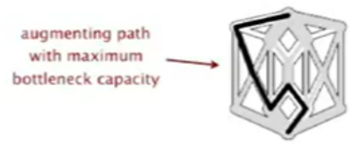
FF performance depends on choice of augmenting paths.

augmenting path	number of paths	implementation
shortest path	$\leq \frac{1}{2} E V$	queue (BFS)
fattest path	$\leq E \ln(E U)$	priority queue
random path	$\leq E U$	randomized queue
DFS path	$\leq E U$	stack (DFS)

digraph with V vertices, E edges, and integer capacities (max U)



shortest path



fattest path

Complexity

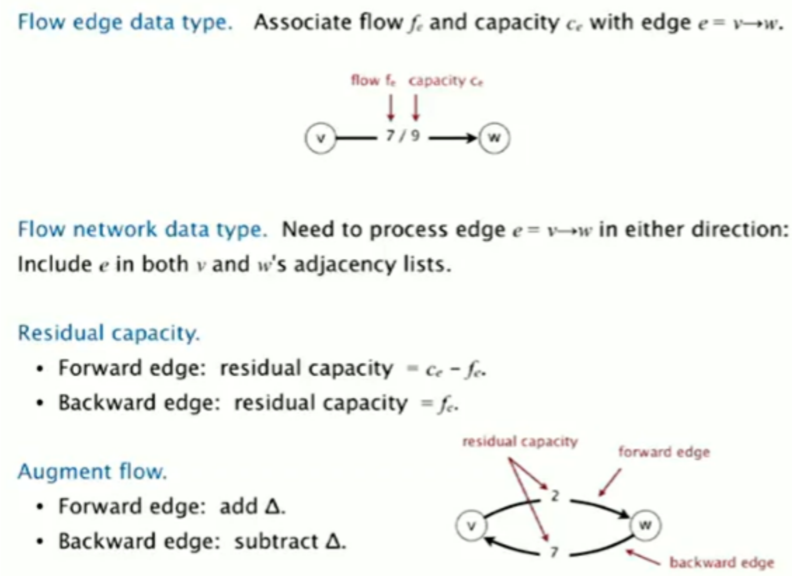
> can't use the worst case as it's often not the practical situation

(Yet another) holy grail for theoretical computer scientists.

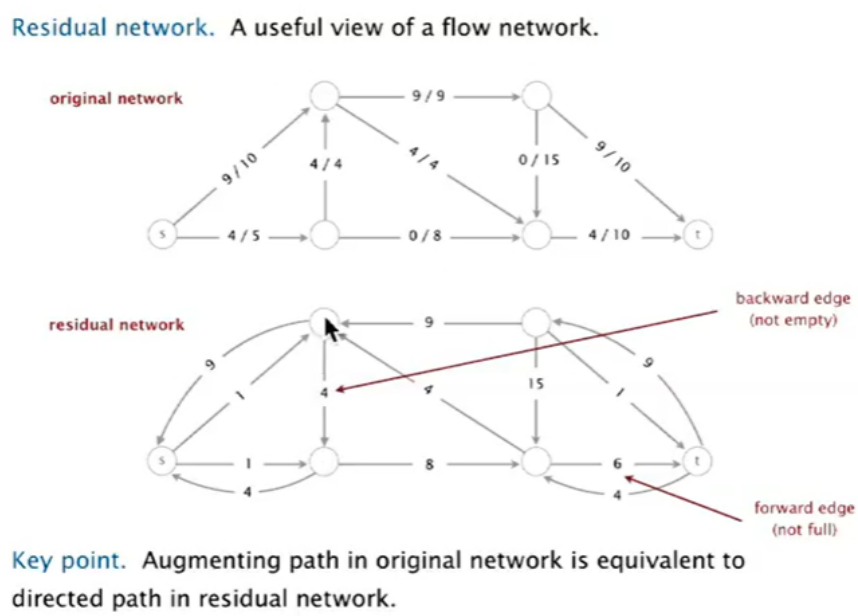
year	method	worst case	discovered by
1951	simplex	$E^3 U$	Dantzig
1955	augmenting path	$E^2 U$	Ford-Fulkerson
1970	shortest augmenting path	E^3	Dinitz, Edmonds-Karp
1970	fattest augmenting path	$E^2 \log E \log(E U)$	Dinitz, Edmonds-Karp
1977	blocking flow	$E^{5/2}$	Cherkasky
1978	blocking flow	$E^{7/3}$	Galil
1983	dynamic trees	$E^2 \log E$	Sleator-Tarjan
1985	capacity scaling	$E^2 \log U$	Gabow
1997	length function	$E^{3/2} \log E \log U$	Goldberg-Rao
2012	compact network	$E^2 / \log E$	Orlin
?	?	E	?

Java Implementations

Flow network representation



Residual Nework



Implementation

Flow edge

```
public class FlowEdge {
    // to deal with floating-point roundoff errors
    private static final double FLOATING_POINT_EPSILON = 1.0E-10;

    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) {
        if (v < 0) throw new IllegalArgumentException("vertex index must be a non-negative integer");
```

```
        if (w < 0) throw new IllegalArgumentException("vertex index must be a non-negative integer");
        if (!(capacity >= 0.0)) throw new IllegalArgumentException("Edge capacity must be non-negative");
        this.v         = v;
        this.w         = w;
        this.capacity  = capacity;
        this.flow      = 0.0;
    }

    public FlowEdge(int v, int w, double capacity, double flow) {
        if (v < 0) throw new IllegalArgumentException("vertex index must be a non-negative integer");
        if (w < 0) throw new IllegalArgumentException("vertex index must be a non-negative integer");
        if (!(capacity >= 0.0))  throw new IllegalArgumentException("edge capacity must be non-negative");
        if (!(flow <= capacity)) throw new IllegalArgumentException("flow exceeds capacity");
        if (!(flow >= 0.0))      throw new IllegalArgumentException("flow must be non-negative");
        this.v         = v;
        this.w         = w;
        this.capacity  = capacity;
        this.flow      = flow;
    }

    public FlowEdge(FlowEdge e) {
        this.v         = e.v;
        this.w         = e.w;
        this.capacity  = e.capacity;
        this.flow      = e.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 IllegalArgumentException("invalid endpoint");
    }

    public double residualCapacityTo(int vertex) {
        if      (vertex == v) return flow;              // backward edge
        else if (vertex == w) return capacity - flow;   // forward edge
        else throw new IllegalArgumentException("invalid endpoint");
    }

    public void addResidualFlowTo(int vertex, double delta) {
        if (!(delta >= 0.0)) throw new IllegalArgumentException("Delta must be non-negative");

        if      (vertex == v) flow -= delta;           // backward edge
        else if (vertex == w) flow += delta;           // forward edge
        else throw new IllegalArgumentException("invalid endpoint");

        // round flow to 0 or capacity if within floating-point precision
        if (Math.abs(flow) <= FLOATING_POINT_EPSILON)
            flow = 0;
        if (Math.abs(flow - capacity) <= FLOATING_POINT_EPSILON)
            flow = capacity;

        if (!(flow >= 0.0))      throw new IllegalArgumentException("Flow is negative");
        if (!(flow <= capacity)) throw new IllegalArgumentException("Flow exceeds capacity");
    }

    public String toString() {
        return v + "->" + w + " " + flow + "/" + capacity;
    }

    public static void main(String[] args) {
        FlowEdge e = new FlowEdge(12, 23, 4.56);
        StdOut.println(e);
    }
}
```

Flow network

```
public class FlowNetwork {
    private static final String NEWLINE = System.getProperty("line.separator");

    private final int V;
    private int E;
    private Bag<FlowEdge>[] adj;

    public FlowNetwork(int V) {
```

```
        if (V < 0) throw new IllegalArgumentException("Number of vertices in a Graph must be non-negative");
        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 FlowNetwork(int V, int E) {
        this(V);
        if (E < 0) throw new IllegalArgumentException("Number of edges must be non-negative");
        for (int i = 0; i < E; i++) {
            int v = StdRandom.uniformInt(V);
            int w = StdRandom.uniformInt(V);
            double capacity = StdRandom.uniformInt(100);
            addEdge(new FlowEdge(v, w, capacity));
        }
    }

    public FlowNetwork(In in) {
        this(in.readInt());
        int E = in.readInt();
        if (E < 0) throw new IllegalArgumentException("number of edges must be non-negative");
        for (int i = 0; i < E; i++) {
            int v = in.readInt();
            int w = in.readInt();
            validateVertex(v);
            validateVertex(w);
            double capacity = in.readDouble();
            addEdge(new FlowEdge(v, w, capacity));
        }
    }

    public int V() {
        return V;
    }

    public int E() {
        return E;
    }

    // throw an IllegalArgumentException unless {@code 0 <= v < V}
    private void validateVertex(int v) {
        if (v < 0 || v >= V)
            throw new IllegalArgumentException("vertex " + v + " is not between 0 and " + (V-1));
    }

    public void addEdge(FlowEdge e) {
```

```java
        int v = e.from();
        int w = e.to();
        validateVertex(v);
        validateVertex(w);
        adj[v].add(e);
        adj[w].add(e);
        E++;
    }

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

    // return list of all edges - excludes self loops
    public Iterable<FlowEdge> edges() {
        Bag<FlowEdge> list = new Bag<FlowEdge>();
        for (int v = 0; v < V; v++)
            for (FlowEdge e : adj(v)) {
                if (e.to() != v)
                    list.add(e);
            }
        return list;
    }

    public String toString() {
        StringBuilder s = new StringBuilder();
        s.append(V + " " + E + NEWLINE);
        for (int v = 0; v < V; v++) {
            s.append(v + ":  ");
            for (FlowEdge e : adj[v]) {
                if (e.to() != v) s.append(e + "  ");
            }
            s.append(NEWLINE);
        }
        return s.toString();
    }

    public static void main(String[] args) {
        In in = new In(args[0]);
        FlowNetwork G = new FlowNetwork(in);
        StdOut.println(G);
    }

}
```

FF algorithm

```java
public class FordFulkerson {
    private static final double FLOATING_POINT_EPSILON = 1.0E-11;

    private final int V;          // number of vertices
    private boolean[] marked;     // marked[v] = true iff s->v path in residual graph
    private FlowEdge[] edgeTo;    // edgeTo[v] = last edge on shortest residual s->v path
    private double value;         // current value of max flow

    public FordFulkerson(FlowNetwork G, int s, int t) {
        V = G.V();
        validate(s);
        validate(t);
        if (s == t)               throw new IllegalArgumentException("Source equals sink");
        if (!isFeasible(G, s, t)) throw new IllegalArgumentException("Initial flow is infeasible");

        // while there exists an augmenting path, use it
        value = excess(G, t);
        while (hasAugmentingPath(G, s, t)) {

            // compute bottleneck capacity
            double bottle = Double.POSITIVE_INFINITY;
            for (int v = t; v != s; v = edgeTo[v].other(v)) {
                bottle = Math.min(bottle, edgeTo[v].residualCapacityTo(v));
            }

            // augment flow
            for (int v = t; v != s; v = edgeTo[v].other(v)) {
                edgeTo[v].addResidualFlowTo(v, bottle);
            }
            value += bottle;
        }

        // check optimality conditions
        assert check(G, s, t);
    }

    public double value()  {
        return value;
    }

    public boolean inCut(int v)  {
        validate(v);
        return marked[v];
    }

```

```java
    // throw an IllegalArgumentException if v is outside prescribed range
    private void validate(int v)  {
        if (v < 0 || v >= V)
            throw new IllegalArgumentException("vertex " + v + " is not between 0 and " + (V-1));
    }

    private boolean hasAugmentingPath(FlowNetwork G, int s, int t) {
        edgeTo = new FlowEdge[G.V()];
        marked = new boolean[G.V()];

        // breadth-first search
        Queue<Integer> queue = new Queue<Integer>();
        queue.enqueue(s);
        marked[s] = true;
        while (!queue.isEmpty() && !marked[t]) {
            int v = queue.dequeue();

            for (FlowEdge e : G.adj(v)) {
                int w = e.other(v);

                // if residual capacity from v to w
                if (e.residualCapacityTo(w) > 0) {
                    if (!marked[w]) {
                        edgeTo[w] = e;
                        marked[w] = true;
                        queue.enqueue(w);
                    }
                }
            }
        }
        // is there an augmenting path?
        return marked[t];
    }

    private double excess(FlowNetwork G, int v) {
        double excess = 0.0;
        for (FlowEdge e : G.adj(v)) {
            if (v == e.from()) excess -= e.flow();
            else               excess += e.flow();
        }
        return excess;
    }

    // return excess flow at vertex v
    private boolean isFeasible(FlowNetwork G, int s, int t) {

        // check that capacity constraints are satisfied
        for (int v = 0; v < G.V(); v++) {
            for (FlowEdge e : G.adj(v)) {
                if (e.flow() < -FLOATING_POINT_EPSILON || e.flow() > e.capacity() + FLOATING_POINT_EPSILON) {
```

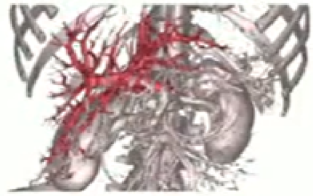
```
                    System.err.println("Edge does not satisfy capacity constraints: " + e);
                    return false;
                }
            }
        }

        // check that net flow into a vertex equals zero, except at source and sink
        if (Math.abs(value + excess(G, s)) > FLOATING_POINT_EPSILON) {
            System.err.println("Excess at source = " + excess(G, s));
            System.err.println("Max flow         = " + value);
            return false;
        }
        if (Math.abs(value - excess(G, t)) > FLOATING_POINT_EPSILON) {
            System.err.println("Excess at sink   = " + excess(G, t));
            System.err.println("Max flow         = " + value);
            return false;
        }
        for (int v = 0; v < G.V(); v++) {
            if (v == s || v == t) continue;
            else if (Math.abs(excess(G, v)) > FLOATING_POINT_EPSILON) {
                System.err.println("Net flow out of " + v + " doesn't equal zero");
                return false;
            }
        }
        return true;
    }

    // check optimality conditions
    private boolean check(FlowNetwork G, int s, int t) {

        // check that flow is feasible
        if (!isFeasible(G, s, t)) {
            System.err.println("Flow is infeasible");
            return false;
        }

        // check that s is on the source side of min cut and that t is not on source side
        if (!inCut(s)) {
            System.err.println("source " + s + " is not on source side of min cut");
            return false;
        }
        if (inCut(t)) {
            System.err.println("sink " + t + " is on source side of min cut");
            return false;
        }

        // check that value of min cut = value of max flow
```

```
        double mincutValue = 0.0;
        for (int v = 0; v < G.V(); v++) {
            for (FlowEdge e : G.adj(v)) {
                if ((v == e.from()) && inCut(e.from()) && !inCut(e.to()))
                    mincutValue += e.capacity();
            }
        }

        if (Math.abs(mincutValue - value) > FLOATING_POINT_EPSILON) {
            System.err.println("Max flow value = " + value + ", min cut value = " + mincutValue);
            return false;
        }

        return true;
    }

    public static void main(String[] args) {

        // create flow network with V vertices and E edges
        int V = Integer.parseInt(args[0]);
        int E = Integer.parseInt(args[1]);
        int s = 0, t = V-1;
        FlowNetwork G = new FlowNetwork(V, E);
        StdOut.println(G);

        // compute maximum flow and minimum cut
        FordFulkerson maxflow = new FordFulkerson(G, s, t);
        StdOut.println("Max flow from " + s + " to " + t);
        for (int v = 0; v < G.V(); v++) {
            for (FlowEdge e : G.adj(v)) {
                if ((v == e.from()) && e.flow() > 0)
                    StdOut.println("   " + e);
            }
        }

        // print min-cut
        StdOut.print("Min cut: ");
        for (int v = 0; v < G.V(); v++) {
            if (maxflow.inCut(v)) StdOut.print(v + " ");
        }
        StdOut.println();
        StdOut.println("Max flow value = " +  maxflow.value());
    }
}
```

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.
- Egalitarian stable matching.
- Security of statistical data.
- Multi-camera scene reconstruction.
- Sensor placement for homeland security.



liver and hepatic vascularization segmentation

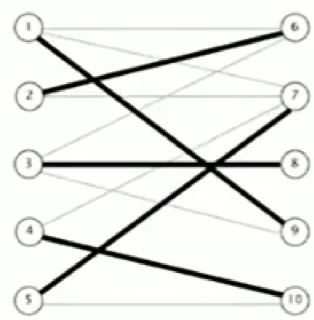
Bipartite matching problem

Given a bipartite graph, find a perfect matching.

perfect matching (solution)

Alice — Google
Bob — Adobe
Carol — Facebook
Dave — Yahoo
Eliza — Amazon

bipartite graph

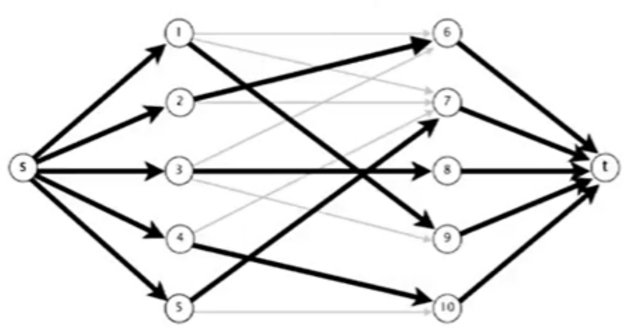


bipartite matching problem

1 Alice: Adobe, Amazon, Google	6 Adobe: Alice, Bob, Carol
2 Bob: Adobe, Amazon	7 Amazon: Alice, Bob, Dave, Eliza
3 Carol: Adobe, Facebook, Google	8 Facebook: Carol
4 Dave: Amazon, Yahoo	9 Google: Alice, Carol
5 Eliza: Amazon, Yahoo	10 Yahoo: Dave, Eliza

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

flow network



bipartite matching problem

1 Alice: Adobe, Amazon, Google	6 Adobe: Alice, Bob, Carol
2 Bob: Adobe, Amazon	7 Amazon: Alice, Bob, Dave, Eliza
3 Carol: Adobe, Facebook, Google	8 Facebook: Carol
4 Dave: Amazon, Yahoo	9 Google: Alice, Carol
5 Eliza: Amazon, Yahoo	10 Yahoo: Dave, Eliza

Baseball Elimination

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

i	team	wins	losses	to play	ATL	PHI	NYM	MON
0	Atlanta	83	71	8	–	1	6	1
1	Philly	80	79	3	1	–	0	2
2	New York	78	78	6	6	0	–	0
3	Montreal	77	82	3	1	2	0	–

Montreal is mathematically eliminated.

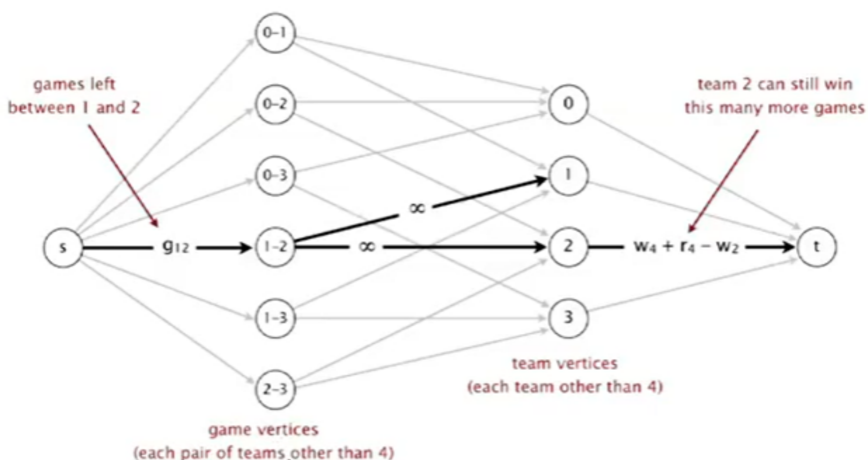
- Montreal finishes with ≤ 80 wins.
- Atlanta already has 83 wins.

Philadelphia is mathematically eliminated.

- Philadelphia finishes with ≤ 83 wins.
- Either New York or Atlanta will finish with ≥ 84 wins.

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

Intuition. Remaining games flow from s to t.



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