6.4 Maximum Flow

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

MinCut Problem

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

- Each edge has a positive capacity

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 $B$. 
Min cut problem

**Def.** A \textit{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 \textit{capacity} is the sum of the capacities of the edges from \( A \) to \( B \).

Min cut application (RAND 1950s)

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

Maxflow: quiz 1

What is the capacity of the \textit{st-cut} \( \{ A, E, F, G \} \)?

A. 34 \((8 + 11 + 9 + 6)\)

B. 45 \((20 + 25)\)

C. 78 \((20 + 8 + 11 + 9 + 6 + 24)\)

D. I don't know.
Potential mincut application (2010s)

Government-in-power’s goal. Cut off communication to set of people.

Maxflow problem

Inputs. 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: $0 \leq$ edge's flow $\leq$ edge's capacity.
- Local equilibrium: inflow = outflow at every vertex (except $s$ and $t$).
**Maxflow problem**

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

**Def.** The value of a flow is the inflow at $t$.

---

**Maxflow application (Tolstoī 1930s)**

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

---

**Potential maxflow application (2010s)**

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

---

We assume no edges point to s or from t.
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.

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

6.4 Maximum Flow

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

3rd augmenting path

4th augmenting path

**Termination.** All paths from \( s \) to \( t \) are blocked by either a
- Full forward edge.
- Empty backward edge.

**Idea:**

- Increase flow along augmenting paths
Maxflow: quiz 2

Which is the augmenting path of highest bottleneck capacity?

A. \( A \rightarrow F \rightarrow G \rightarrow H \)
B. \( A \rightarrow F \rightarrow B \rightarrow G \rightarrow H \)
C. \( A \rightarrow F \rightarrow B \rightarrow G \rightarrow D \rightarrow H \)
D. I don’t know.

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

Fundamental 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?

Relationship between flows and cuts

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 \(A\).
**Relationship between flows and cuts**

*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 \(A\).

\[
\text{net flow across cut } = 10 + 5 + 10 = 25
\]

**Maxflow: quiz 3**

Which is the net flow across the \(st\)-cut \(\{A, E, F, G\}\)?

A. \(11 \ (20 + 25 - 8 - 11 - 9 - 6)\)

B. \(26 \ (20 + 22 - 8 - 4 - 4)\)

C. \(42 \ (20 + 22)\)

D. \(45 \ (20 + 25)\)

E. *I don’t know.*

**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 \(f\).

**Intuition.** Conservation of flow.

**Pf.** By induction on the size of \(B\).

- **Base case:** \(B = \{t\}\).
- **Induction step:** remains true by local equilibrium when moving any vertex from \(A\) to \(B\).

**Corollary.** Outflow from \(s\) = inflow to \(t\) = value of flow.
**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 \((A, B)\).

**Maxflow–mincut theorem**

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

**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.

\([iii \Rightarrow i]\)

- Let \( f \) be a flow with no augmenting paths.
- Let \( A \) be set of vertices connected to \( s \) by an undirected path with no full forward or empty backward edges.
- By definition of cut \( A \), \( s \) is in \( A \).
- By definition of cut \( A \) and flow \( f \), \( s \) 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

**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

Fundamental 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?

6.4 Maximum Flow

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

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

**Invariant.** The flow is integral 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 integral 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.

4th iteration

199th iteration

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.

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

How to choose augmenting paths?

Choose augmenting paths with:
- Shortest path: fewest number of edges.
- Fattest path: max bottleneck capacity.

Use care when selecting augmenting paths.
- Some choices lead to exponential algorithms.
- Clever choices lead to polynomial algorithms.

<table>
<thead>
<tr>
<th>augmenting path</th>
<th>number of paths</th>
<th>implementation</th>
</tr>
</thead>
<tbody>
<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>
<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(EU) )</td>
<td>priority queue</td>
</tr>
</tbody>
</table>

flow network with \( V \) vertices, \( E \) edges, and integer capacities between 1 and \( U \)
Flow network representation

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

$$
\begin{array}{c}
\text{flow } f_e \quad \text{capacity } c_e \\
V \quad 7/9 \quad W
\end{array}
$$

Flow network data type. Must be able to process edge $e = v \rightarrow w$ in either direction: include $e$ in adjacency lists of both $v$ and $w$.

Residual (spare) 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.

Flow edge API

public class FlowEdge

```java
public class FlowEdge

FlowEdge(int v, int w, double capacity) create a flow edge $v \rightarrow w$

int from() vertex this edge points from
int to() vertex this edge points to
int other(int v) other endpoint
double capacity() capacity of this edge
double flow() flow in this edge
double residualCapacityTo(int v) residual capacity toward $v$
void addResidualFlowTo(int v, double delta) add delta flow toward $v$
```

Flow edge: Java implementation

```java
public class FlowEdge

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 IllegalArgumentException();
}

public double residualCapacityTo(int vertex)

public void addResidualFlowTo(int vertex, double delta) {...}
```

Key point. Augmenting paths in original network are in one-to-one correspondence with directed paths in residual network.
Flow edge: Java implementation (continued)

```java
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 network API

```java
public class FlowNetwork
    public class FlowNetwork

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

    public void addEdge(FlowEdge 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];
    }
```

Flow network: Java implementation

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

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

    public void addEdge(FlowEdge 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];
    }
```

Flow network: adjacency-lists representation

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

Conventions. Allow self-loops and parallel edges.

Note. Adjacency list includes edges with 0 residual capacity.
(residual network is represented implicitly)
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>();
    marked[s] = true; // can stop BFS as soon as augmenting path is discovered
    while (!queue.isEmpty() && !marked[t]) {
        int v = queue.dequeue();
        for (FlowEdge e : G.adj(v)) {
            int w = e.other(v);
            if (!marked[w] && (e.residualCapacityTo(w) > 0)) {
                edgeTo[w] = e;
                marked[w] = true; // mark w
                queue.enqueue(w); // add w to the queue
            }
        }
    }
    return marked[t]; // is t reachable from s in residual network?
}
```
Bipartite matching problem

Problem. Given $N$ people and $N$ tasks, assign the tasks to people so that:
- Every task is assigned to a qualified person.
- Every person is assigned to exactly one task.

Network flow formulation of bipartite matching

- Create $s$, $t$, one vertex for each task, and one vertex for each person.
- Add edge from $s$ to each task (of capacity 1).
- Add edge from each person to $t$ (of capacity 1).
- Add edge from task to qualified person (of infinite capacity).

Network flow formulation of bipartite matching

1–1 correspondence between perfect matchings in bipartite graph and integer-valued maxflows of value $N$ in flow network.
What the mincut tells us

**Goal.** When no perfect matching, explain why.

![Diagram showing a graph with nodes and edges, illustrating the mincut concept.](image)

\[ S = \{ 2, 4, 5 \} \]
\[ T = \{ 7, 10 \} \]

Tasks in \( S \) can be matched only to people in \( T \)

\[ |S| > |T| \]

No perfect matching exists

### 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>WAS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Atlanta</td>
<td>83</td>
<td>71</td>
<td>8</td>
<td>1</td>
<td>6</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Philly</td>
<td>80</td>
<td>79</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td></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>0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Washington</td>
<td>77</td>
<td>82</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Washington is mathematically eliminated.
- Washington finishes with \( \leq 80 \) wins.
- Atlanta already has 83 wins.

---

**What the mincut tells us**

**Mincut.** Consider mincut \((A, B)\).
- Let \( S \) = tasks on \( s \) side of cut.
- Let \( T \) = people on \( s \) side of cut.
- Fact: \(|S| > |T|\); tasks in \( S \) can be matched only to people in \( T \).

![Diagram showing the mincut and its implications.](image)

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>WAS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Atlanta</td>
<td>83</td>
<td>71</td>
<td>8</td>
<td>1</td>
<td>6</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Philly</td>
<td>80</td>
<td>79</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td></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>0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Washington</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>3</td>
<td>8</td>
<td>7</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Baltimore</td>
<td>71</td>
<td>63</td>
<td>28</td>
<td>3</td>
<td>2</td>
<td>7</td>
<td>4</td>
<td></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>0</td>
<td>0</td>
<td></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>0</td>
<td></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 ≤ 76 wins.
- Wins for \( R = \{ \text{NYY, BAL, BOS, TOR} \} = 278.
- Remaining games among \( \{ \text{NYY, BAL, BOS, TOR} \} = 3 + 8 + 7 + 2 + 7 = 27.
- Average team in \( R \) wins \( 305/4 = 76.25 \) games.

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^2 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>fattest augmenting path</td>
<td>( E^3 \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>Cherkassky</td>
</tr>
<tr>
<td>1978</td>
<td>blocking flow</td>
<td>( E^{3/2} )</td>
<td>Galil</td>
</tr>
<tr>
<td>1983</td>
<td>dynamic trees</td>
<td>( E^3 \log E )</td>
<td>Sleator-Tarjan</td>
</tr>
<tr>
<td>1985</td>
<td>capacity scaling</td>
<td>( E^3 \log U )</td>
<td>Gabow</td>
</tr>
<tr>
<td>1997</td>
<td>length function</td>
<td>( E^{5/2} \log E \log U )</td>
<td>Goldberg-Rao</td>
</tr>
<tr>
<td>2012</td>
<td>compact network</td>
<td>( E^{3/2} \log E )</td>
<td>Orlin</td>
</tr>
</tbody>
</table>

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

Baseball elimination problem: maxflow formulation

Intuition. Remaining games flow from \( s \) to \( t \).

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

Maximum flow algorithms: practice

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

Best in practice. Push-relabel method with gap relabeling: \( E^{3/2} \).

Computer vision. Specialized algorithms for problems with special structure.

On Implementing Push-Relabel Method for the Maximum Flow Problem

Benor V. Cherkaev1 and Andrew V. Goldberg2

1 Central Institute for Economic and Mathematics, Krakow, 30-059, Poland. E-mail: cherk@ciem.krakow.pl
2 Computer Science Department, Stanford University, Stanford, CA 94305, USA. E-mail: goldberl@stanford.edu

Abstract. We study efficient implementations of the push-relabel method for the maximum flow problem. The resulting codes are faster than the previous code, and work better on some problems. The new code is an order of magnitude faster and works in the range of E < 100. We also exhibit a family of problems for which the running time of all known methods seems to have a roughly quadratic growth rate.

Theory and Methodology

Computational investigations of maximum flow algorithms

Ravindra K. Ahuja1, Manesh Koladkar2, Anuj K. Mishra3, James B. Orlin4,5

1 Department of Industrial and Systems Engineering, Duke University, Durham, NC 27708, USA. E-mail: rkevah@duke.edu
2 Department of Management and Engineering, University of Technology, Sydney, NSW 2007, Australia. E-mail: manesh.koladkar@uts.edu.au
3 Indian Institute of Technology, Kharagpur, India. E-mail: anujk@iitkgp.ac.in
4 Operations Research Laboratory, Stanford University, Stanford, CA 94305, USA. E-mail: orlin@stanford.edu
5 Department of Computer Science and Engineering, Carnegie Mellon University, Pittsburgh, PA 15213, USA. E-mail: goldberl@cs.cmu.edu

Received 26 August 1995; accepted 27 June 1996.
Summary

**Min cut problem.** Find an $st$-cut of minimum capacity.

**Max flow 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-world maxflow/mincut problems in linear time.
- Theory: prove it for worst-case inputs.
- Still much to be learned!