6.4 Maximum Flow

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

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

**Input.** An edge-weighted digraph, source vertex $s$, and target vertex $t$. Each edge has a positive capacity.
Max flow / min cut problem

\[
\begin{align*}
\text{flow} & \quad \text{capacity} \\
0 / 10 & \quad 0 / 10 \\
0 / 5 & \quad 0 / 5 \\
0 / 3 & \quad 0 / 3 \\
0 / 15 & \quad 0 / 15 \\
0 / 10 & \quad 0 / 10 \\
0 / 12 & \quad 0 / 12 \\
0 / 10 & \quad 0 / 10 \\
\end{align*}
\]

value of flow = 0
Max flow / min cut problem
Max flow / min cut problem

Q: What is the value of the max flow?
A. 20 [191071]  
B. 19 [175058]  
C. 16 [170059]  
D. 11 [170148]  
E. 10 [170215]  

Extra: Find a cut whose total capacity equals the max flow.
Q: What is the value of the max flow?
D. 11

Extra: Find a cut whose total capacity equals the max flow.
Non Obvious Fact. There is always a cut such that:

- Set A contains the source (s).
- Set B contains the sink (t).
- The capacity of this cut is equal to the value of the max flow.
- All edges from A to B are full.
- All edges from B to A are empty.
**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*.

capacity = 10 + 5 + 15 = 30
**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*.

\[
\text{capacity} = 10 + 8 + 16 = 34
\]
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.
**Maxflow problem**

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

Each edge has a positive capacity.
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 $t$).

![Diagram of maxflow problem](image-url)
**Maxflow problem**

**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 $t$

```
value = 5 + 10 + 10 = 25
```
Maxflow problem

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 \(t\).

Maximum *st-flow (maxflow) problem*. Find a flow of maximum value.

![Graph with values and arrows showing flow](image)

\[
\text{value} = 8 + 10 + 10 = 28
\]
**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!
6.4 **Maximum Flow**

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

**1st augmenting path**

![Diagram with labeled capacities and flows]

- **bottleneck capacity = 10**
- \( 0 + 10 = 10 \)
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
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).

3rd augmenting path
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).

4\textsuperscript{th} augmenting path
**Idea:** increase flow along augmenting paths

**Termination.** All paths from $s$ to $t$ are blocked by either a

- **Full** forward edge.
- **Empty** backward edge.

---

**no more augmenting paths**

![Graph diagram with flow values and edge types](image)
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

Questions.
- How to find an augmenting path?
- If FF terminates, does it always compute a maxflow?
- How to compute a mincut?
- Does FF always terminate? If so, after how many augmentations?
6.4 Maximum Flow

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

Questions.

- How to find an augmenting path?
- If FF terminates, does it always compute a maxflow?
- How to compute a mincut?
- Does FF always terminate? If so, after how many augmentations?
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: Java implementation

Ford-Fulkerson inspired details

- Both forward and backward edges are provided.

```
s 4 / 7 1 4 / 4 2 4 / 9 t
```

```
0 1 4.0 7.0
0 1 4.0 7.0 1 2 4.0 4.0
2 3 4.0 9.0 1 2 4.0 4.0
2 3 4.0 9.0
```
Flow network: Java implementation

Ford-Fulkerson inspired details

- Both forward and backward edges are provided.
- Edges can report their residual capacity.

\[ \begin{align*}
    &e.\text{edgeFrom}(): 0 & e.\text{edgeTo}(): 1 \\
    &e.\text{residualCapacityTo}(1): 3 & \text{Forward edge, residual capacity} = \text{capacity} - \text{flow} \\
    &e.\text{residualCapacityTo}(0): 4 & \text{Backward edge, residual capacity} = \text{flow}
\end{align*} \]
Flow network: Java implementation

Residual Network

- Edge weighted digraph representing how much spare (used) capacity is available on a forward (backward) edge. If none, no edge.
- Represented *IMPLICITLY* by `e.residualCapacityTo()`.
Residual Networks - Questions to ponder

Draw the residual network corresponding to the graph below.

![Original network diagram](image)

What is the result of the code below?

```java
for (FlowEdge e : G.adj(s)) {
    int v = e.from(); int w = e.to();
    System.out.println(e.residualCapacityTo(w));
}
```

How can you find an augmenting path using the residual network graph?
Residual Networks

Draw the residual network corresponding to the graph below.

original network

residual network
What is the result of the code below (s is the source vertex)?

```java
for (FlowEdge e : G.adj(s)) {
    int v = e.from(); int w = e.to();
    System.out.println(e.residualCapacityTo(w));
}
```

- The two FlowEdges adjacent to e have residual capacity of 0 and 3 when examined in the forward direction.
  - Prints 0 on a new line, and 3 on a new line.
Residual Networks

Draw the residual network corresponding to the graph below.

How can you find an augmenting path using the residual network graph?
- Find any path from $s$ to $t$. Edge only exists if weight > 0.
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);
    while (!queue.isEmpty()) {
        int v = queue.dequeue();
        for (FlowEdge e : G.adj(v)) {
            int w = e.other(v);
            if (!marked[w] && e.residualCapacityTo(w) > 0) {
                marked[w] = true;
                queue.enqueue(w);
                edgeTo[w] = e;
            }
        }
        // how do we know if a path exists to t?
    }
    return marked[t];
}
```
Ford-Fulkerson: Java implementation

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

    // ... (additional code)
}
```

Diagram:
- **Value of flow**: Value of the flow is calculated as the sum of the bottleneck capacities of the augmenting paths.
- **Augmenting Path**: The algorithm repeatedly finds an augmenting path from source to sink.
- **Bottleneck Capacity**: The bottleneck capacity of an augmenting path is the minimum residual capacity of any edge along the path.
- **Backward Walk**: The algorithm walks backward from the sink along the augmenting path to update the residual network.

The diagram illustrates the flow network with capacities on the edges, showing how the flow is augmented and how the residual network is updated.
Residual Networks

Any path in residual network is an augmenting path in original network

original network

residual network
6.4 Maximum Flow

- introduction
- Ford-Fulkerson algorithm
- Java implementation
- maxflow-mincut theorem
- running time analysis
- applications
**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 find an augmenting path? BFS (or other).
- If FF terminates, does it always compute a maxflow?
- How to compute a mincut?
- Does FF always terminate? If so, after how many augmentations?
Maxflow-mincut theorem

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

Value of the maxflow = capacity of mincut.

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

i. There exists an \( s-t \)-cut cut whose capacity equals the value of the flow \( f \).
ii. \( f \) is a maxflow.
iii. There is no augmenting path with respect to \( f \).

Overall Goal:

• Prove that i \( \Rightarrow \) ii. [Trivial]
• Prove that ii \( \Rightarrow \) iii. [Trivial]
• Prove that iii \( \Rightarrow \) i. [A little work]
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 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 $f$.

[$i \Rightarrow ii$]: Trivial by analogy with water flow [see book for technical proof].
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 an st-cut cut whose capacity equals the value of the flow $f$.

ii. $f$ is a maxflow.

iii. There is no augmenting path with respect to $f$.

[ ii $\Rightarrow$ iii ] Trivial, we prove contrapositive: $\neg$iii $\Rightarrow$ $\neg$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.
Computing a mincut from a maxflow

Find an augmenting path.
Computing a mincut from a maxflow

- We’ve found a cut whose capacity equals the value of the flow.

Find an augmenting path.
- Couldn’t find an augmenting path (some edges block us).
  - These edges form a cut.
  - There is no backward flow from t to s.
  - All edges from s to t are full.
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 \( f \).

Overall Goal:

• Prove that i \( \Rightarrow \) ii. [Analogy with water, see book for technical proof]
• Prove that ii \( \Rightarrow \) iii. [Trivial by proving contrapositive]
• Prove that iii \( \Rightarrow \) i. [Constructive proof, see book for technical proof]
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

Questions.
• How to find an augmenting path? BFS (or other).
• If FF terminates, does it always compute a maxflow?
  - Yes, because non-existence of augmenting path implies max flow.
  - iii ⇒ i ⇒ ii
• How to compute a mincut?
• Does FF always terminate? If so, after how many augmentations?
Computing a mincut from a maxflow

Find an augmenting path.

- Couldn’t find an augmenting path (some edges block us).
  - These edges form a cut.
  - There is no backward flow from t to s.
  - All edges from s to t are full.
- We’ve found a cut whose capacity equals the value of the flow.
6.4 Maximum Flow

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

- How to find an augmenting path? BFS (or other).
- If FF terminates, does it always compute a maxflow? Yes. ✔
- How to compute a mincut? Easy. ✔
- Does FF always terminate? If so, after how many augmentations?

Yes, provided edge capacities are integers (or augmenting paths are chosen carefully) requires clever analysis
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.
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.

1\textsuperscript{st} iteration
Bad news. Even when edge capacities are integers, number of augmenting paths could be equal to the value of the maxflow.

2nd iteration

2nd 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.
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.
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.
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.
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.

![Network diagram](attachment:image.png)

**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} EV )</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 EU )</td>
<td>randomized queue</td>
</tr>
<tr>
<td>DFS path</td>
<td>( \leq EU )</td>
<td>stack (DFS)</td>
</tr>
</tbody>
</table>

*digraph with \( V \) vertices, \( E \) edges, and integer capacities between 1 and \( U \)*
Non-integer weights (beyond scope of course)

If:

- $e_1 = 1$
- $e_2 = (\sqrt{5} - 1)/2$
- $e_3 = 1$
- Other edges of weight 2 or greater

Can show:

- Ford Fulkerson can get stuck in infinite loop.
- Does not converge even if it runs forever.
6.4 Maximum Flow

- introduction
- Ford-Fulkerson algorithm
- Java implementation
- maxflow-mincut theorem
- running time analysis
- applications
"Free world" goals. Understand peak Soviet supply rate. Cut supplies (if cold war turns into real war).

rail network connecting Soviet Union with Eastern European countries (map declassified by Pentagon in 1999)
Maxflow application (1950s)

**Soviet Union goal.** Maximize flow of supplies to Eastern Europe.
- Originally studied by writer Alexei Tolstoi in the 1930s (ad hoc approach).
- Later considered by Ford & Fulkerson via min cut approach.
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.
N students apply for N jobs.

Each gets several offers.

Is there a way to match all students to jobs?
Bipartite matching problem

Given a bipartite graph, find a perfect matching.

**perfect matching (solution)**

<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>Facebook</td>
</tr>
<tr>
<td>Dave</td>
<td>Yahoo</td>
</tr>
<tr>
<td>Eliza</td>
<td>Amazon</td>
</tr>
</tbody>
</table>

**bipartite graph**

1 —— 2 —— 3 —— 4 —— 5

6 —— 7 —— 8 —— 9 —— 10

**bipartite matching problem**

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

6 | Adobe, Alice, Bob, Carol |
7 | Amazon, Alice, Bob, Dave, Eliza |
8 | Facebook, Carol |
9 | Google, Alice, Carol |
10 | Yahoo, Dave, Eliza |
Network flow formulation of bipartite matching

- Create $s$, $t$, one vertex for each student, and one vertex for each job.
- Add edge from $s$ to each student (capacity 1).
- Add edge from each job to $t$ (capacity 1).
- Add edge from student to each job offered (infinite capacity).
Network flow formulation of bipartite matching

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
2 Bob Adobe Amazon
3 Carol Adobe Facebook Google
4 Dave Amazon Yahoo
5 Eliza Amazon Yahoo
6 Adobe Alice Bob Carol
7 Amazon Alice Bob Dave Eliza
8 Facebook Carol
9 Google Alice Carol
10 Yahoo Dave Eliza
What the mincut tells us

Goal. When no perfect matching, explain why.

\begin{itemize}
\item \text{S = \{ 2, 4, 5 \}}
\item \text{T = \{ 7, 10 \}}
\item student in S can be matched only to companies in T
\item |S| > |T|
\end{itemize}

no perfect matching exists
What the mincut tells us

**Minicut.** 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>

Montreal is mathematically eliminated.

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

**AL East (August 30, 1996)**

**Detroit is mathematically eliminated.**

- Detroit finishes with \( \leq 76 \) 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.
**Baseball elimination problem: maxflow formulation**

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

Build a flow network for EACH team. Below is graph for 4.

**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>fattest augmenting path</td>
<td>(E^2 \log E \log(U))</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>
<tr>
<td>?</td>
<td>?</td>
<td>(E)</td>
<td>?</td>
</tr>
</tbody>
</table>

Maxflow algorithms for sparse digraphs with \(E\) edges, integer capacities between 1 and \(U\)
Maximum flow algorithms: practice

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


On Implementing Push-Relabel Method for the Maximum Flow Problem

Boris V. Cherkassky\textsuperscript{1} and Andrew V. Goldberg\textsuperscript{2}

\textsuperscript{1} Central Institute for Economics and Mathematics, Krasikova St. 32, 117418, Moscow, Russia
cher@ceemr.msk.ru

\textsuperscript{2} Computer Science Department, Stanford University
Stanford, CA 94305, USA
goldberg@cs.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 codes, and much faster on some problem families. The speedup is due to the combination of heuristics used in our implementations. We also exhibit a family of problems for which the running time of all known methods seem to have a roughly quadratic growth rate.
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