Max-Flow and Min-Cut
Recitation #8

Zhexiong Liu
zhexiong@cs.pitt.edu

Department of Computer Science
University of Pittsburgh

October 30, 2020
Slides adapted from Kevin Wayne
Directed graph

A directed graph (or digraph) is a graph $G(V,E)$ such that edges on the graph have a direction

- A directed acyclic graph is called DAG (no directed cycle)
- A weighted digraph has a weight attached to each edge
Flow graph

A flow graph $G(S,T,c)$, where $S$, $T$, $c$ are source vertex, sink vertex, and edge capacity respectively, is a directed graph such that each edge has a capacity

- **Capacity**: flow on an edge cannot exceed its capacity

Source vertex $S$  
Sink vertex $T$  
Flow graph $G(S,T,c)$
Flow graph

A flow graph $G(S,T,c)$, where $S$, $T$, $c$ are source vertex, sink vertex, and edge capacity respectively, is a directed graph such that each edge has a capacity.

- **Capacity**: bottleneck capacity of S-T path SABT is 2
Flow graph

A flow graph $G(S,T,c)$, where $S$, $T$, $c$ are source vertex, sink vertex, and edge capacity respectively, is a directed graph such that each edge has a capacity.

- **Conservation**: inflow equals to outflow except $S$ and $T$
Max-flow problem

Maximum flow problems involves finding a flow of maximum value of a flow graph

- The value of flow graph equals to the inflow of sink

![Flow graph with a flow of 3](image-url)

- **Source vertex**: S
- **Sink vertex**: T
- **Bottleneck edge**: S to B

Flow

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Flow</th>
<th>Capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>2 / 5</td>
<td>2 / 5</td>
</tr>
<tr>
<td>C</td>
<td>1 / 1</td>
<td>1 / 1</td>
</tr>
<tr>
<td>A</td>
<td>2 / 2</td>
<td>2 / 2</td>
</tr>
<tr>
<td>B</td>
<td>2 / 3</td>
<td>2 / 3</td>
</tr>
<tr>
<td>D</td>
<td>1 / 2</td>
<td>1 / 2</td>
</tr>
<tr>
<td>T</td>
<td>0 / 5</td>
<td>0 / 5</td>
</tr>
</tbody>
</table>

Flow graph with a flow of 3
Max-flow problem

Maximize flow of trail supplies from Soviet Union to eastern European countries (Pentagon 1999)

https://algs4.cs.princeton.edu/home/
Min-cut problem

Cut rail graphs if Soviet Union and Eastern European countries (Pentagon 1999) turns cold war into real war

https://algs4.cs.princeton.edu/home/
Min-cut problem

A cut (of the flow graph) is to divide vertices into two disjoint sets such that source vertex S in one set A and sink vertex T in the other set B.

The cut capacity is the sum of the capacities of all the edges from A to B.

Cut capacity is 7
Min-cut problem

A cut (of the flow graph) is to divide vertices into two disjoint sets such that source vertex S in one set A and sink vertex T in the other set B.

The cut capacity is the sum of the capacities of all the edges from A to B.

Cut capacity is 10
Max-flow min-cut

Max-flow min-cut theorem (or called Ford-Fulkerson theorem) argues that the value of max-flow equals to capacity of min-cut in any graph

- Max-flow “==” min-cut
- Max-flow is a dual problem of Min-cut
- Solve max-flow problem by solving min-cut problem
- Solve min-cut problem by solving max-flow problem
Max-flow min-cut

Ford-Fulkerson theorem argues that the value of max-flow equals to capacity of min-cut in any graph

- Solve min-cut by iterating all cut situations

Min-cut capacity is 8
Max-flow min-cut

Ford-Fulkerson theorem argues that the value of max-flow equals to capacity of min-cut in any graph

- Solve min-cut by iterating all cut situations

Min-cut capacity is 10
Max-flow min-cut

Ford-Fulkerson theorem argues that the value of max-flow equals to capacity of min-cut in any graph

- Solve min-cut by iterating all cut situations

Min-cut capacity is 9
Max-flow min-cut

Ford-Fulkerson theorem argues that the value of max-flow equals to capacity of min-cut in any graph

- Solve min-cut by iterating all cut situations

Min-cut capacity is 7
Max-flow min-cut

Ford-Fulkerson theorem argues that the value of max-flow equals to capacity of min-cut in any graph

- Solve min-cut by iterating all cut situations

Min-cut capacity is 9
Max-flow min-cut

Ford-Fulkerson theorem argues that the value of max-flow equals to capacity of min-cut in any graph

- Solve min-cut by iterating all cut situations

Min-cut capacity is 8
Max-flow min-cut

Ford-Fulkerson theorem argues that the value of max-flow equals to capacity of min-cut in any graph

- Solve min-cut by iterating all cut situations
Max-flow min-cut

Ford-Fulkerson theorem argues that the value of max-flow equals to capacity of min-cut in any graph

- Solve min-cut by iterating all cut situations
Max-flow min-cut

Ford-Fulkerson theorem argues that the value of max-flow equals to capacity of min-cut in any graph

- Solve min-cut by iterating all cut situations

---

**Diagram:**

- Node S
- Node A: capacity 2
- Node B: capacity 3
- Node C: capacity 2
- Node D: capacity 5
- Node T: capacity 5

---

Min-cut capacity is 6
Max-flow min-cut

Ford-Fulkerson theorem argues that the value of max-flow equals to capacity of min-cut in any graph

- Solve max-flow by pushing flow as much as possible
Max-flow min-cut

Ford-Fulkerson theorem argues that the value of max-flow equals to capacity of min-cut in any graph

- Step1: push flow of 1 for S-T path SCDT

Initial flow graph with flow of 0
Flow graph with flow of 1 in 1st step
Max-flow min-cut

Ford-Fulkerson theorem argues that the value of max-flow equals to capacity of min-cut in any graph

- Step2: push flow of 4 for S-T path SADT

Flow graph with flow of 1 in 1st step

Flow graph with flow of 5 in 2nd step
Max-flow min-cut

Ford-Fulkerson theorem argues that the value of max-flow equals to capacity of min-cut in any graph

- Step3: push flow of 1 for S-T path SABT

Flow graph with flow of 5 in 2nd step

Flow graph with flow of 6 in 3rd step
Max-flow min-cut

Ford-Fulkerson theorem argues that the value of max-flow equals to capacity of min-cut in any graph

- Optimal solution with max-flow of 6 \((1+4+1)!\)
- Select three S-T paths in order SCDT→SADT→SABT

```
Current flow graph with flow of 6
```
Max-flow min-cut

Ford-Fulkerson theorem argues that the value of max-flow equals to capacity of min-cut in any graph

- Not optimal if select S-T paths in other orders!
- For example, start with S-T path SADT

Current flow graph with flow of 0
Max-flow min-cut

Ford-Fulkerson theorem argues that the value of max-flow equals to capacity of min-cut in any graph

- Step 1: push flow of 5 for S-T path SADT

![Initial flow graph with flow of 0](image1)

![Flow graph with flow of 5 in 1st step](image2)
Max-flow min-cut

Ford-Fulkerson theorem argues that the value of max-flow equals to capacity of min-cut in any graph

- Cannot add the rest flow of 1 leaving S to arrive T

Flow graph with flow of 5 in 1\textsuperscript{st} step

Flow graph cannot add flow of 1 in next step
Residual edges

Residual edges exist to “undo” the flow on a path

- Pushing flow of 4 for path AD equals to pushing flow of 5 for path AD and “undoing” flow of 1 for path AD
Residual edges

Residual edges exist to “undo” the flow on a path

- Residual edge capacity equals to the flow value on its original edge (at most undo all the flow)
Residual edges

Given a flow graph $G(S,T,c)$ and a flow $f$, in some case, we can push more flow by adding a residual edge
- “Undo” flow of 1 for AD by adding residual edge DA
- The flow of 1 leaving S can reach T through SCDABT!

Flow graph with flow of 5

Flow graph with a residual edge can add another flow of 1
Residual graph

Given a flow graph $G(S,T,c)$ and a flow $f$, a residual graph $G'(S,T,c')$ is a graph with residual edges such that forward edge has capacity $c'=c-f$ and backward edge has capacity $c'=f$.
Residual graph

Given a flow graph $G(S,T,c)$ and a flow $f$, construct a residual graph $G'(S,T,c')$

- Construct $G'(S,T,c')$ by adding edge & its capacity
Residual graph

Given a flow graph $G(S,T,c)$ and a flow $f$, construct a residual graph $G'(S,T,c')$

- Add all backward (residual) edges with capacity $c'=f$
Residual graph

Given a flow graph $G(S,T,c)$ and a flow $f$, construct a residual graph $G'(S,T,c')$

- Add all forward edges with capacity $c'=c-f$
Augmenting paths

An augmenting path is a $S$-$T$ path in the residual graph such that each edge on the path has capacity greater than 0

- Augmenting path SADT

Residual graph $G'(S,T,c')$
Ford-Fulkerson algorithm

Ford-Fulkerson algorithm (FFA) is a greedy algorithm that finds the maximum flow in a flow graph

\[
\text{while (there exists an augmenting path) } \{
    \text{Find an augmenting path} \\
    \text{Compute bottleneck capacity of the path} \\
    \text{Augment flow along the path}
\}
\]

- It is sometimes called a “method” instead of an “algorithm” as the method that finds augmenting paths in a residual graph is not fully specified.
Ford-Fulkerson algorithm

Ford-Fulkerson algorithm (FFA) is a greedy algorithm that finds the maximum flow in a flow graph.

- Initialized the residual graph $G'(S,T,c')$

Original graph $G(S,T,c)$

Initial residual graph $G'(S,T,c')$
Ford-Fulkerson algorithm

Ford-Fulkerson algorithm (FFA) is a greedy algorithm that finds the maximum flow in a flow graph.

- Step 1: select augmenting path SABT; push flow of 2

Initial residual graph $G'(S,T,c')$

Residual graph $G'(S,T,c')$ in 1st step
Ford-Fulkerson algorithm

Ford-Fulkerson algorithm (FFA) is a greedy algorithm that finds the maximum flow in a flow graph

- **Step2:** select augmenting path SADT; push flow of 3

Residual graph $G'(S,T,c')$ in 1st step

Residual graph $G'(S,T,c')$ in 2nd step
Ford-Fulkerson algorithm

Ford-Fulkerson algorithm (FFA) is a greedy algorithm that finds the maximum flow in a flow graph.

- Step 3: select augmenting path SCDT; push flow of 1

Residual graph $G'(S,T,c')$ in 2\textsuperscript{nd} step

Residual graph $G'(S,T,c')$ in 3\textsuperscript{rd} step
Ford-Fulkerson algorithm

Ford-Fulkerson algorithm (FFA) is a greedy algorithm that finds the maximum flow in a flow graph

- Stop if no augmenting paths on the residual graph
- Max-flow is 6 (2+3+1)

Residual graph $G'(S,T,c')$ in 3nd iteration
Ford-Fulkerson algorithm

Ford-Fulkerson algorithm (FFA) can be implemented with multiple variants regarding augmenting paths

- Find the augmenting path with **DFS** (E*OPT augmentation in V iterations)
- Find the largest-bottleneck (most amount of flow to sink) augmenting path (implemented with **Dijkstra** algorithm for ElogU augmentations in (E+ VlogV) iterations)
- Find **shortest** (lowest number of edges) augmenting path (implemented with **BFS** for EV augmentations in V iterations)

OPT is the value of the max flow
# Time complexity

## History of worst-case running times of FFA

<table>
<thead>
<tr>
<th>Year</th>
<th>Discoverer</th>
<th>Finding augmenting paths</th>
<th>Worst time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1951</td>
<td>Dantzig</td>
<td>Simplex</td>
<td>(EV^2 \times \text{OPT} )</td>
</tr>
<tr>
<td>1955</td>
<td>Ford Fulkerson</td>
<td>Augmenting path (DFS)</td>
<td>(EV \times \text{OPT} )</td>
</tr>
<tr>
<td>1970</td>
<td>Edmonds-Karp</td>
<td>Shortest path (BFS)</td>
<td>(E^2V )</td>
</tr>
<tr>
<td>1970</td>
<td>Edmonds-Karp</td>
<td>Max capacity (Dijkstra)</td>
<td>((E+V \log V) \times E \log(\text{OPT}))</td>
</tr>
<tr>
<td>1970</td>
<td>Dinitiz</td>
<td>Improved shortest path (BFS+DFS)</td>
<td>(EV^2 )</td>
</tr>
<tr>
<td>1972</td>
<td>Edmonds-Karp, Dinitz</td>
<td>Capacity scaling (Heuristic)</td>
<td>(E^2 \log(\text{OPT}) )</td>
</tr>
<tr>
<td>1973</td>
<td>Dinitiz-Gabow</td>
<td>Improved capacity scaling</td>
<td>(EV \log(\text{OPT}) )</td>
</tr>
<tr>
<td>1974</td>
<td>Karzanov</td>
<td>Preflow-push</td>
<td>(V^3 )</td>
</tr>
<tr>
<td>1983</td>
<td>Sleator-Tarjan</td>
<td>Dynamic trees</td>
<td>(EV \log V )</td>
</tr>
<tr>
<td>1986</td>
<td>Goldberg-Tarjan</td>
<td>FIFO preflow-push</td>
<td>(EV \log(\frac{V^2}{E}) )</td>
</tr>
<tr>
<td>1997</td>
<td>Goldberg-Rao</td>
<td>Length function</td>
<td>(E^{3/2} \log(\frac{V^2}{E}) \log(\text{OPT}))</td>
</tr>
</tbody>
</table>

OPT is the value of the max flow

https://algs4.cs.princeton.edu/home/
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();
    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)) {
        double bottle = Double.POSITIVE_INFINITY;
        for (int v = t; v != s; v = edgeTo[v].other(v)) {
            // compute bottleneck capacity
            bottle = Math.min(bottle, edgeTo[v].residualCapacityTo(v));
        }
        for (int v = t; v != s; v = edgeTo[v].other(v)) {
            // augment flow
            edgeTo[v].addResidualFlowTo(v, bottle);
        }
        value += bottle;
    }
}
private boolean isFeasible(FlowNetwork G, int s, int t) {
    for (int v = 0; v < G.V(); v++) {
        // check capacity constraints
        for (FlowEdge e : G.adj(v))
            if (e.flow() < 0 || e.flow() > e.capacity()) return false;
    }
    if (Math.abs(value + excess(G, s)) > 0) return false;  // check conservation constrains
    if (Math.abs(value - excess(G, t)) > 0) return false;  // check conservation constrains
    for (int v = 0; v < G.V(); v++) {
        if (v == s || v == t) continue;
        else if (Math.abs(excess(G, v)) > 0) return false;
    }
    return true;
}

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

```java
// edgeTo[] contains a parent-link representation of a S-T path found by shortest augmenting (BFS)
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>(); // breadth-first search
    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 (e.residualCapacityTo(w) > 0) { // if residual capacity from v to w
                if (!marked[w]) {
                    edgeTo[w] = e;
                    marked[w] = true;
                    queue.enqueue(w);
                }
            }
        }
    }
    return marked[t];
}
```
Applications

Ford-Fulkerson algorithm can solve many practical problems, such as the assignment problem:

- Given company offers and student preference for jobs, assignment each student to at most one company.

<table>
<thead>
<tr>
<th>Company</th>
<th>Allen</th>
<th>Bruce</th>
<th>Carol</th>
<th>David</th>
<th>Eliza</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adobe</td>
<td>Adobe</td>
<td>Adobe</td>
<td>Apple</td>
<td>IBM</td>
<td>IBM</td>
</tr>
<tr>
<td>Apple</td>
<td>IMB</td>
<td>IMB</td>
<td>IBM</td>
<td>Yahoo</td>
<td>Google</td>
</tr>
<tr>
<td>Google</td>
<td>Google</td>
<td>Yahoo</td>
<td>Yahoo</td>
<td></td>
<td>Yahoo</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Company</th>
<th>Allen</th>
<th>Carol</th>
<th>Bruce</th>
<th>David</th>
<th>Eliza</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adobe</td>
<td>Allen</td>
<td>Carol</td>
<td>Bruce</td>
<td>David</td>
<td>Eliza</td>
</tr>
<tr>
<td>Apple</td>
<td>Allen</td>
<td>Bruce</td>
<td>David</td>
<td>Eliza</td>
<td></td>
</tr>
<tr>
<td>IBM</td>
<td>Bruce</td>
<td>David</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Google</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Yahoo</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Applications

Solving an assignment problem using an undirected bipartite graph is called bipartite matching

- Represent assignment problem with bipartite graph
Applications

Solving an assignment problem using an undirected bipartite graph is called bipartite matching

- Matching set \{A-1, B-3, C-5, E-4\} (4 matches)

Not Optimal!
Applications

Solving an assignment problem using an undirected bipartite graph is called **bipartite matching**

- Matching set \{A-2, B-1, C-3, D-5, E-4\} (5 matches)

![Optimal solution!](image)
Applications

Optimal solution can be found by solving a maximum bipartite matching (MBP) problem

- MBP can be reduced to a max-flow problem!
Applications

Reduce MBP to a max-flow problem
- Add source vertex $S$ and sink vertex $T$
- Add direction leaving $S$
Applications

Reduce MBP to a max-flow problem
- Add source vertex $S$ and sink vertex $T$
- Add direction leaving $S$
- Add edge capacity (each edge has capacity of 1)
Applications

Reduce procedure
- **Convert** bipartite matching to a max-flow problem
- **Solve** max-flow problem with FFA
- **Convert** max-flow solution to the bipartite solution

Notes
- Many practical problems **reduce** to max-flow problems
- Many **variants of FFA** for solving max-flow problems