Algorithmic Problem Solving

Le 6 – Graphs part II

Fredrik Heintz
Dept of Computer and Information Science
Linköping University

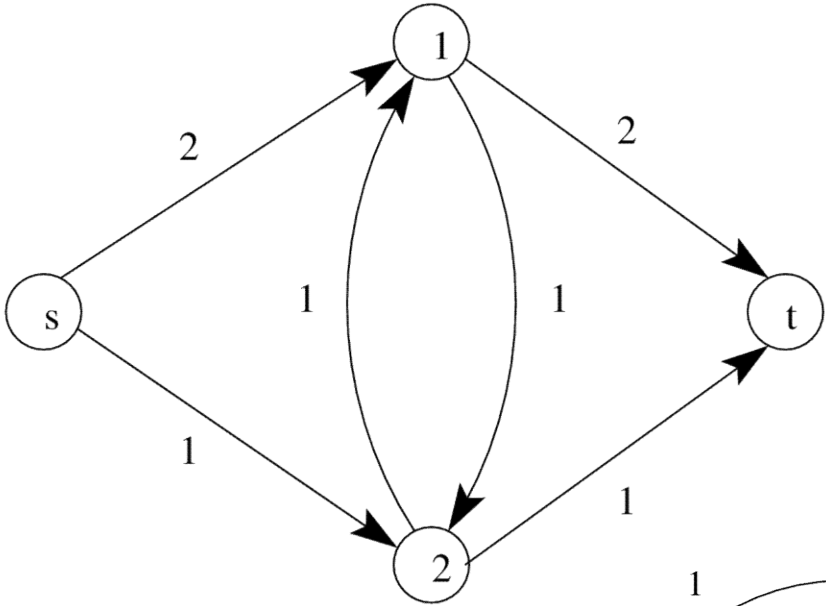
Outline

- Network flow
  - Max Flow (lab 2.6)
  - Min Cut (lab 2.7)
  - Min Cost Max Flow (lab 2.8)

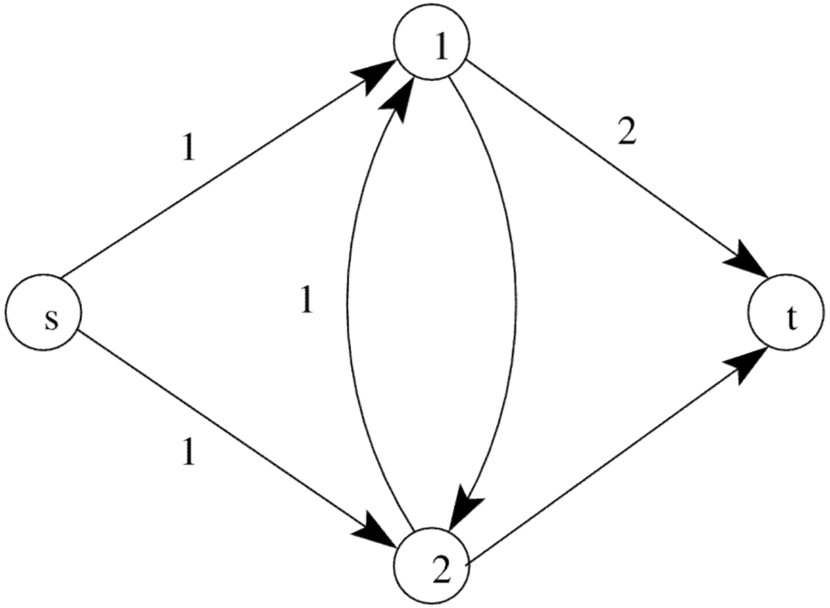
Network Flow

- A network is a directed graph G=(V,E) with a *source* vertex s∈V and a *sink* vertex t∈V. Each edge e=(v,w) from v to w has a defined *capacity*, denoted by u(e) or u(v,w). It is useful to also define capacity for any pair of vertices (v,w)∉E with u(v,w)=0.
- In a network flow problem, we assign a *flow* to each edge.
  - *Raw flow* is a function r(v,w) that satisfies the following properties:
    - **Conservation**: The total flow entering v must equal the total flow leaving v for all vertices except s and t, ∑w∈V r(v,w)=0, for all v∈V\{s,t}.
    - **Capacity constraint**: The flow along any edge must be positive and less than the capacity of that edge, r(v,w)≤u(v,w) for all v,w∈V.
  - *Net flow* is a function f(v,w) that also satisfies the following conditions:
    - **Skew symmetry**: f(v,w)=–f(w,v).
  - With a raw flow, we can have flows going both from v to w and flow going from w to v. In a net flow formulation we only keep track of the difference between these two flows f(v,w)=r(v,w)–r(w,v).
- The value of flow f from source s is defined as |f|=∑v∈V f(s,v).

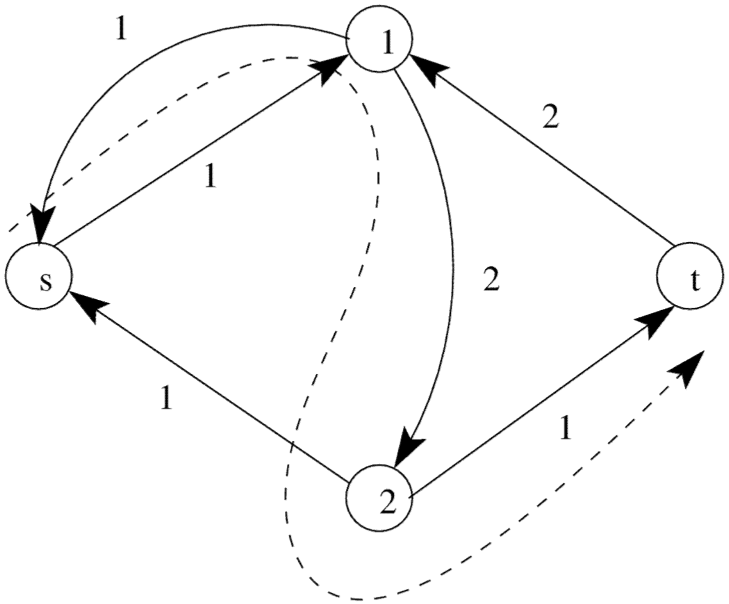
Network Flow – Example Network



Network



Raw flow



Residual graph and augmenting path

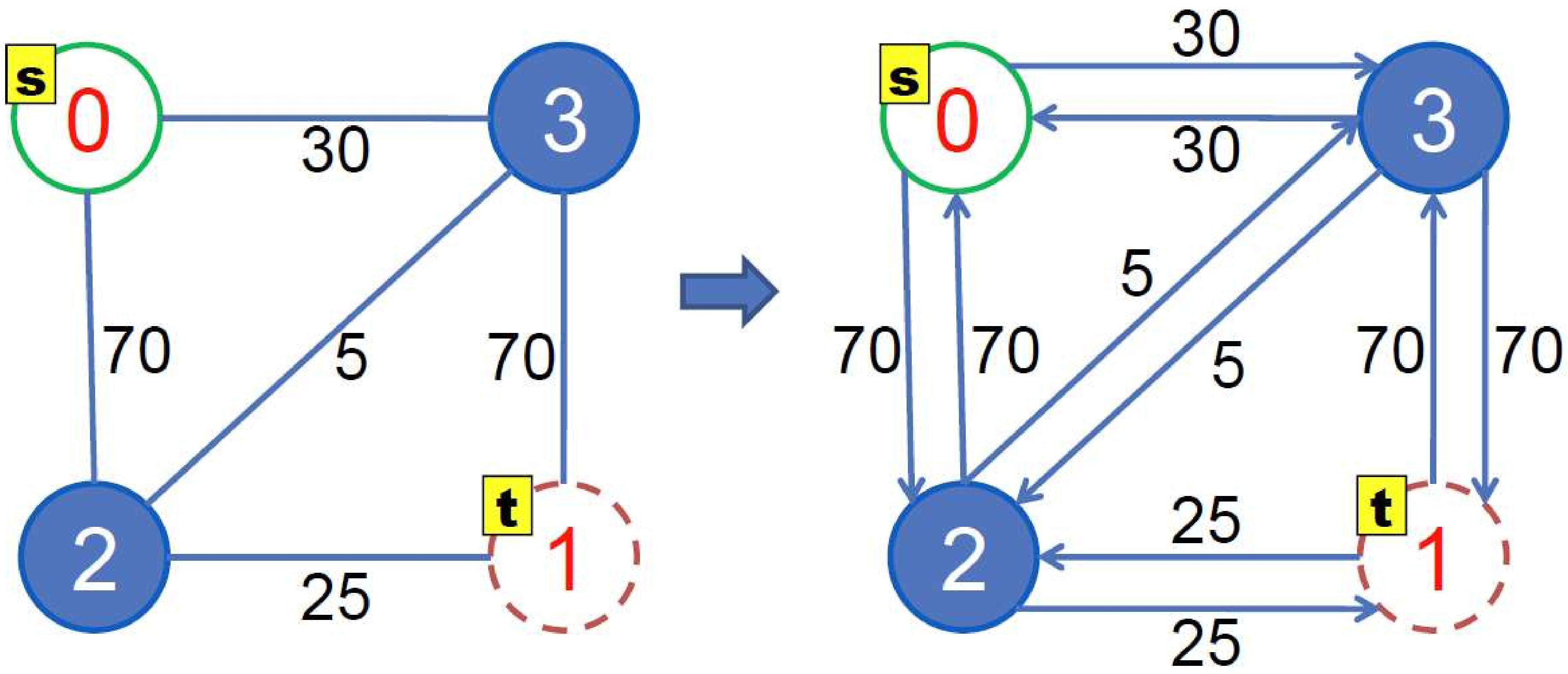
The Ford Fulkerson's Method

- One Solution: Ford Fulkerson's Method

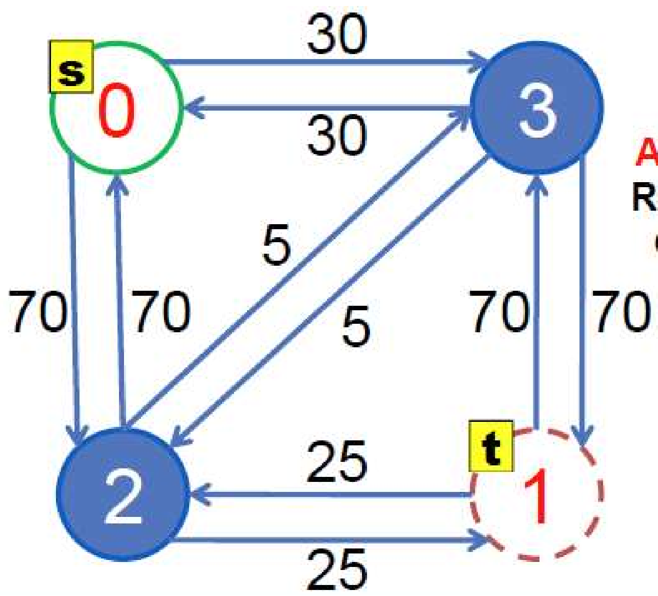
  - A surprisingly **simple** *iterative* algorithm

Send a flow *f* through path *p* whenever there exists an **augmenting path** *p* from *s* to *t*

Network Flow – Example Maximum Flow

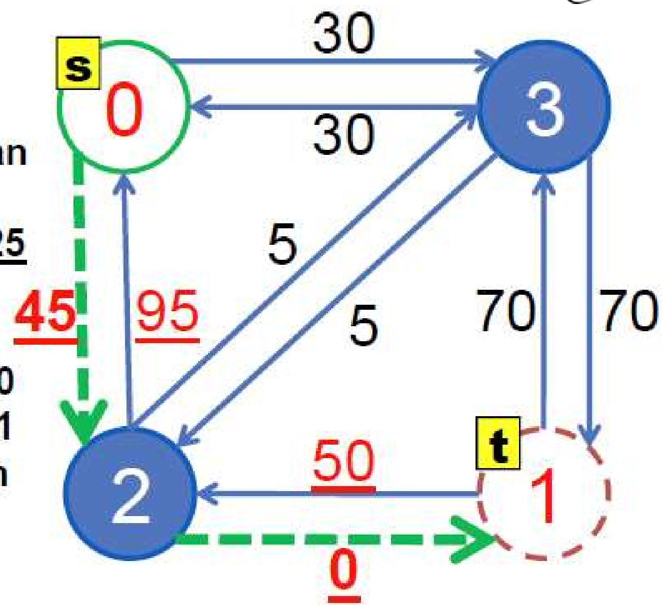


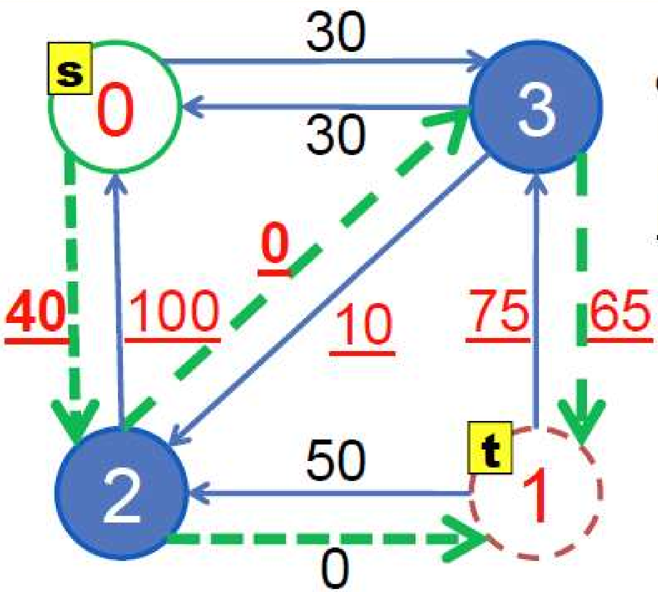
Network Flow – Example Maximum Flow



A. Initial Residual Graph

B. We can pass a flow of <u>25 units</u> from source 0 to sink 1 via path 0-2-1

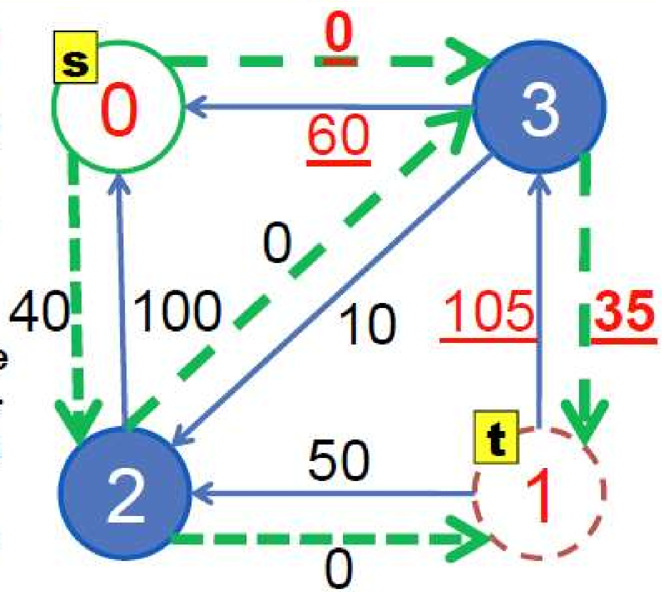




C. We can pass another flow of <u>5 units</u> via path 0-2-3-1

D. We can pass last flow of <u>30 units</u> via path 0-3-1

No more low can be added. 25+ 5+30=60 is the <u>max flow</u> value



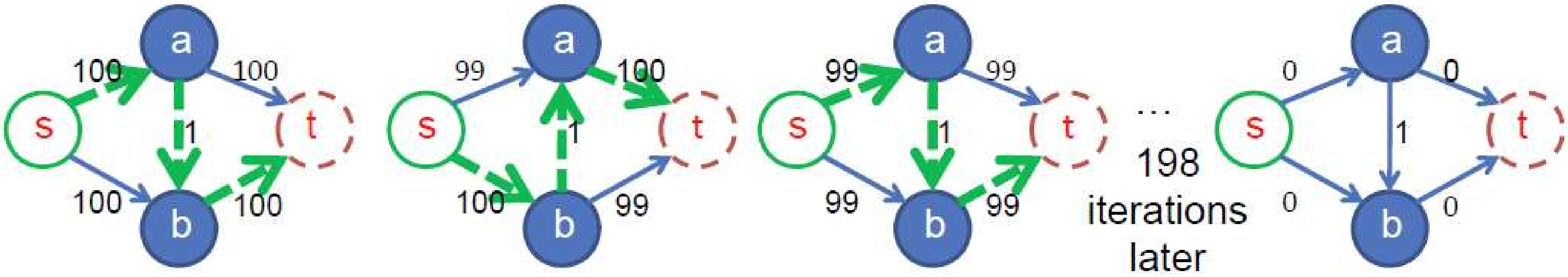
The Ford Fulkerson's Method


```
setup directed residual graph
each edge has the same weight with the original graph

mf = 0 // this is an iterative algorithm, mf stands for max_flow
while (there exists an augmenting path p from s to t) {
   // p is a path from s to t that pass through positive edges in residual graph
  augment/send flow f along the path p (s -> ... -> i -> j -> ... t)
    1. find f, the min edge weight along the path p
    2. decrease the weight of forward edges (e.g. i -> j) along path p by f
       reason: obvious, we use the capacities of those forward edges
    3. increase the weight of backward edges (e.g. j -> i) along path p by f
       reason: not so obvious, but this is important for the correctness of Ford
       Fulkerson's method;
  mf += f // we can send a flow of size f from s to t, increase mf
}
output mf
```

The Ford-Fulkerson Algorithm

- DFS implementation of Ford Fulkerson's method runs in O(|f*|E) and can be very slow on graph like this:
  - Notice the presence of backward edges (only drawn for edge a→b or b→a this time)
    - Q: What if we do not use backward edges?



The Edmond-Karp Algorithm

BFS Implementation

- BFS implementation of Ford Fulkerson's method (called Edmonds Karp's algorithm) runs in $O(VE^2)$

Edmonds Karp's (using STL) (1)

```
int res[MAX_V][MAX_V], mf, f, s, t; // global variables
vi p; // note that vi is our shortcut for vector<int>

// traverse the BFS spanning tree as in print_path (section 4.3)
void augment(int v, int minEdge) {
  // reach the source, record minEdge in a global variable 'f'
  if (v == s) { f = minEdge; return; }
  // recursive call
  else if (p[v] != -1) { augment(p[v], min(minEdge, res[p[v]][v]));
  // alter residual capacities
                          res[p[v]][v] -= f; res[v][p[v]] += f; }
}

// in int main()
  // set up the 2d AdjMatrix 'res', 's', and 't' with appropriate values
```

Edmonds Karp's (using STL) (2)

```
mf = 0;
while (1) { // run O(VE * V^2 = V^3*E) Edmonds Karp to solve the Max Flow problem
  f = 0;

  // run BFS, please examine parts of the BFS code that is different than in Section 4.2.23
  vi dist(MAX_V, INF); dist[s] = 0; // #define INF 2000000000
  queue<int> q; q.push(s);
  p.assign(MAX_V, -1); // (we have to record the BFS spanning tree)
  while (!q.empty()) { // (we need the shortest path from s to t!)
    int u = q.front(); q.pop();
    if (u == t) break; // immediately stop BFS if we already reach sink t
    for (int v = 0; v < MAX_V; v++) // note: enumerating neighbors with AdjMatrix is 'slow'
      if (res[u][v] > 0 && dist[v] == INF) dist[v] = dist[u] + 1, q.push(v), p[v] = u;
  }

  augment(t, INF); // find the min edge weight 'f' along this path, if any
  if (f == 0) break; // if we cannot send any more flow ('f' = 0), terminate the loop
  mf += f; // we can still send a flow, increase the max flow!
}

printf("%d\n", mf); // this is the max flow value of this flow graph
```

Network Flow – Scaling

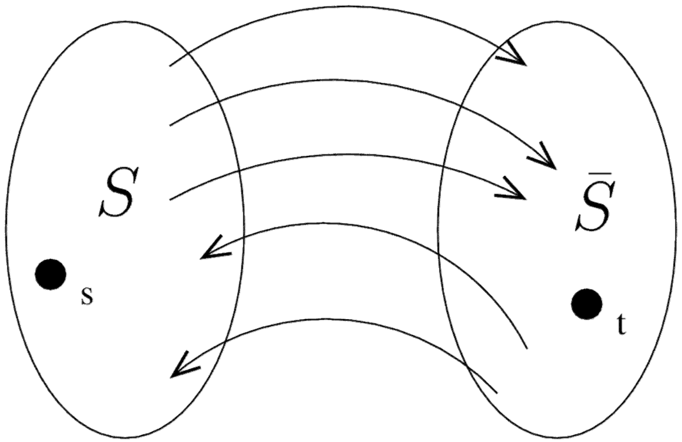
- We can improve the running time of the Ford-Fulkerson algorithm by using a scaling algorithm. The idea is to reduce our max flow problem to the simple case where all edge capacities are either 0 or 1 (Gabow in 1985 and Dinic in 1973):
  - Scale the problem down somehow by rounding off lower order bits.
  - Solve the rounded problem.
  - Scale the problem back up, add back the bits we rounded off, and fix any errors in our solution.
- In the specific case of the maximum flow problem, the algorithm is:
  - Start with all capacities in the graph at 0.
  - Shift in the higher-order bit of each capacity. Each capacity is then either 0 or 1.
  - Solve this maximum flow problem.
  - Repeat this process until we have processed all remaining bits.
- To scale back up:
  - Start with the maximum flow for the scaled-down problem. Shift the bit of each capacity by 1, doubling all the capacities. If we then double all our flow values, we still have a maximum flow.
  - Increment some of the capacities. This restores the lower order bits that we truncated. Find augmenting paths in the residual network to re-maximize the flow.

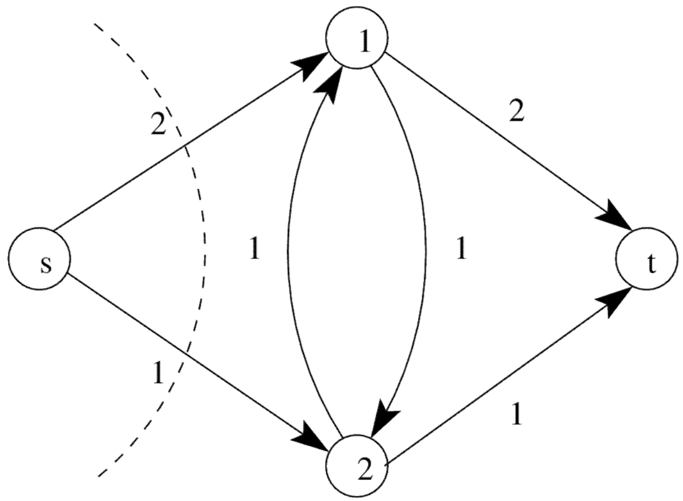
Maximum Flow Algorithms

- Ford-Fulkerson with DFS $O(|f|\ E)$
- Edmond-Karp (Ford-Fulkerson with BFS) $O(VE^2)$
- Dinic's $O(V^2E)$
- Push-relabel $O(V^3)$
- Binary blocking flow algorithm $O(min(V^{2/3}, E^{1/2})\ E\ log(V^2/E)\ log(|f|))$

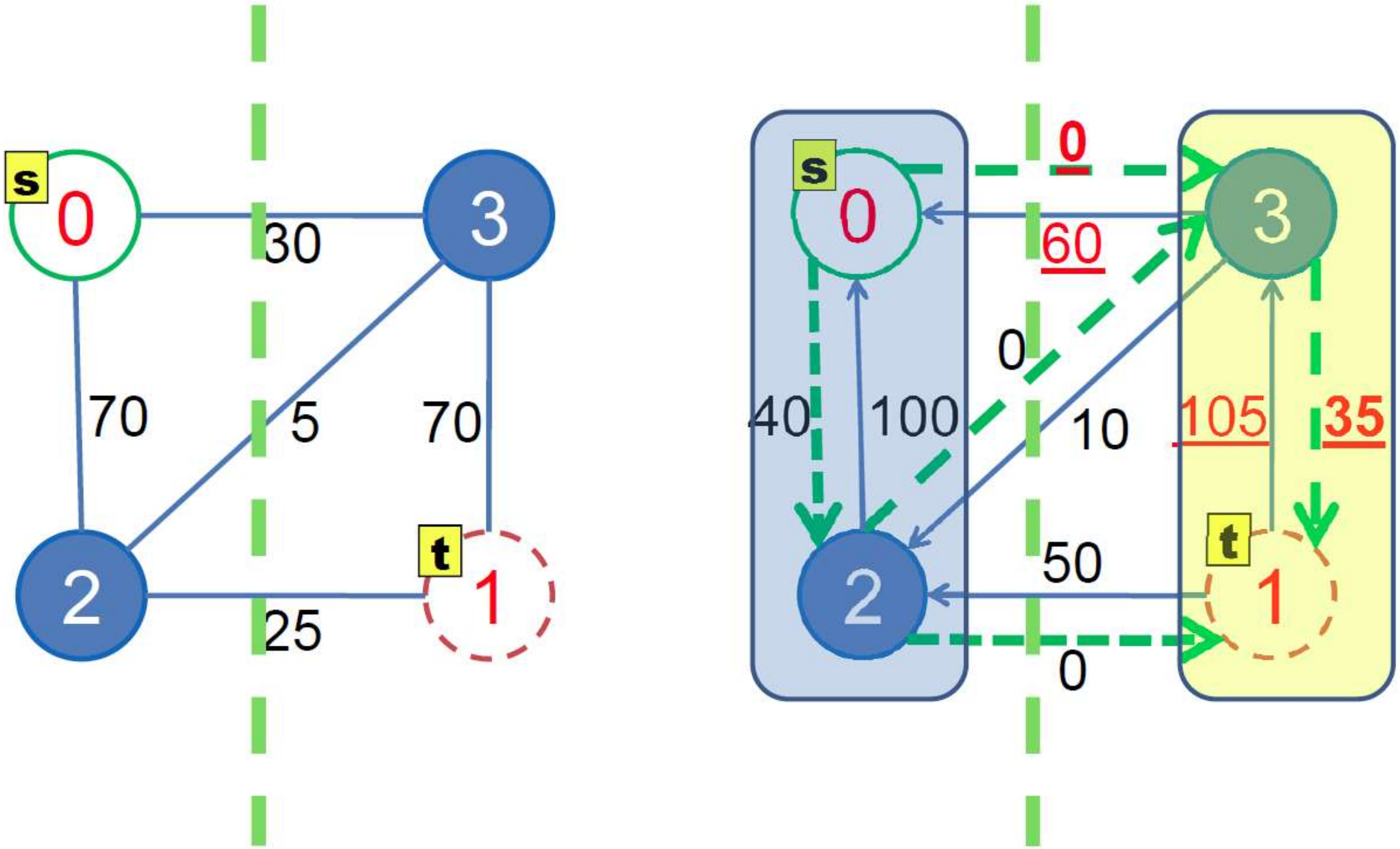
Minimum Cut

- An *s-t cut* of network G is a partition of the vertices V into 2 groups: S and $\bar{S}$=V\S such that s∈S and t∈$\bar{S}$.
  - The net flow along cut (S,$\bar{S}$) is defined as $f(S)=\sum_{v \in S} \sum_{w \in \bar{S}} f(v,w)$.
  - The value (or capacity) of a cut is defined as $u(S)=\sum_{v \in S} \sum_{w \in \bar{S}} u(v,w)$.
- For a flow network, we define a *minimum cut* to be a cut of the graph with minimum capacity.
- To find the minimum cut, compute the maximum flow and find the set of vertices reachable from s with positive edges in the residual graph, this is the set S.



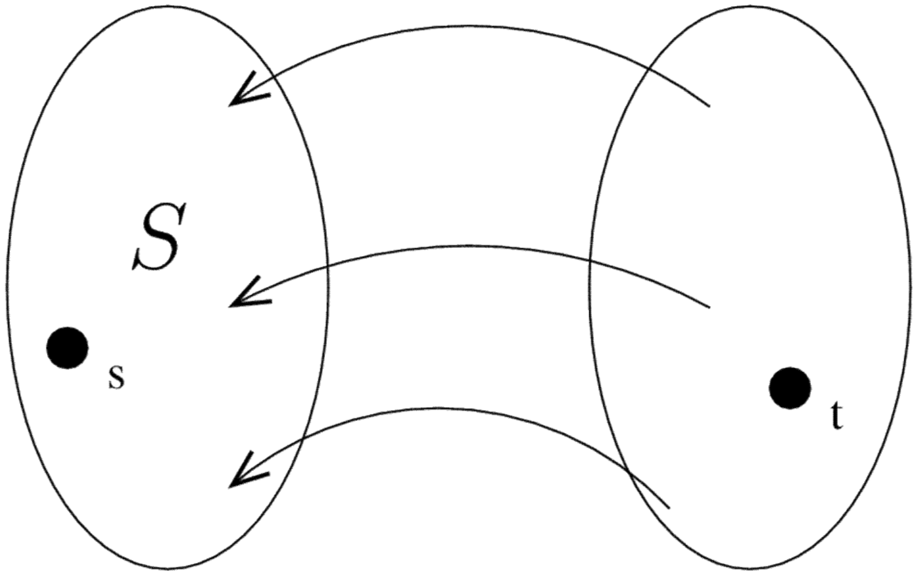


Minimum Cut Example



Max-Flow Min-Cut Theorem

- In a flow network G, the following conditions are equivalent:
  - A flow f is a maximum flow.
  - The residual network G_f has no augmenting paths.
  - $|f|=u(S)$ for some cut S.
- These conditions imply that the value of the maximum flow is equal to the value of the minimum s-t cut: $\max_f |f|=\min_S u(S)$, where f is a flow and S is an s-t cut.



Minimum Cost Maximum Flow

- Extend the definition of a network flow with a cost per unit of flow on each edge: $c(v,w) \in R$, where $(v,w) \in E$.
- The cost of a flow f is defined as: $c(f)=\sum_{e \in E} f(e) \cdot c(e)$
- A minimum cost maximum flow of a network G=(V,E) is a maximum flow with the smallest possible cost.
  - Note that costs can be negative.
  - Note that edges in the residual graph of a network need to have their costs determined carefully. Consider an edge (v,w) with capacity u(v,w), cost per unit flow c(v,w). Let f(v,w) be the flow of the edge. Then the residual graph has two edges corresponding to (v,w). The first edge is (v,w) with capacity u(v,w)–f(v,w) and cost c(v,w), and second edge is (w,v) with capacity f(v,w) and cost –c(v,w).
  - It's clear that minimum cost maximum flow generalizes maximum flow by assigning a cost of 0 to every edge.
  - It also generalizes shortest path, if we set each cost equal to its corresponding edge length while assigning the same capacity to every edge.
- A flow is **optimal (min-cost)** iff there are no negative cost cycles in the residual network.

Network Flow Variants

- Multi-source, multi-sink max flow
  - Create a super-source/sink with infinite capacity edges to the sources/sinks
- Vertex capacities
  - Split each vertex into two vertices and add a bi-directional edge with the vertex capacity between them. Remember to change the edges to the vertex.
- Min-Cost Circulation
  - Equivalent to min-cost max-flow (simply disconnect the source and sink)
- Maximum Independent and Edge-Disjoint Paths

Summary

- Network flow
  - Max Flow (lab 2.6)
  - Min Cut (lab 2.7)
  - Min Cost Max Flow (lab 2.8)