# COL702: Advanced Data Structures and Algorithms

Ragesh Jaiswal, CSE, IITD

# Course Overview

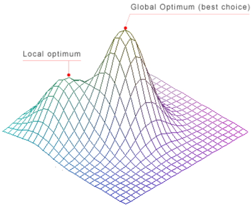
- Graph Algorithms
- Algorithm Design Techniques:
  - Greedy Algorithms
  - Divide and Conquer
  - Dynamic Programming
  - Network Flows
    - Hill-climbing and reduction

## Network Flow

- Reduction
  1. We will obtain an algorithm $A$ for a *Network Flow* problem using Hill-climbing.
  2. Given a new problem, we will *rephrase* this problem as a Network Flow problem.
  3. We will then use algorithm $A$ to solve the rephrased problem and obtain a solution.
  4. Finally, we build a solution for the original problem using the solution to the rephrased problem.

- Hill-climbing optimization strategy:
  - Start with any solution that meets the constraints.
  - Repeat until there is no simple way to improve the solution:
    - Try to improve the solution via a "local" change, still satisfying the constraints.
  - Output the solution.
- A few points to note about Hill-climbing:
  - More often than not hill-climbing does NOT find an optimal solution, just a "local optimum"
  - Often used as an approximation algorithm or heuristic.
  - Also called *gradient ascent*, *interior point method*.

- Hill-climbing optimization strategy:
  - Start with any solution that meets the constraints.
  - Repeat until there is no simple way to improve the solution:
    - Try to improve the solution via a "local" change, still satisfying the constraints.
  - Output the solution.
- Local optima:
  - One can view the set of all possible solutions as a high-dimensional region. The objective function then gives a height for each point.
  - We would like to find the highest point.
  - But we usually find a local optima, a point higher than others near it.
  - So, while global optima are local optima, the reverse is not always true.



- We want to model various kinds of networks using graphs and then solve real world problems with respect to these networks by studying the underlying graph.
- One problem that arises in network design is routing "flows" within the network.
    - Transportation Network: Vertices are cities and edges denote highways. Every highway has certain traffic capacity. We are interested in knowing the maximum amount commodity that can be shipped from a source city to a destination city.
    - Computer Networks: Edges are links and vertices are switches. Each link has some capacity of carrying packets. Again, we are interested in knowing how much traffic can a source node send to a destination node.

- To model these problems, we consider weighted, directed graph $G = (V, E)$ with the following properties:
  - Capacity: Associated with each edge $e$ is a capacity that is a non-negative integer denoted by $c(e)$.
  - Source node: There is a source node $s$ with no in-coming edges.
  - Sink node: There is a sink node $t$ with no out-going edges. All other nodes are called *internal nodes*.

- To model these problems, we consider weighted, directed graph $G = (V, E)$ with the following properties:
  - Capacity: Associated with each edge $e$ is a capacity that is a non-negative integer denoted by $c(e)$.
  - Source node: There is a source node $s$ with no in-coming edges.
  - Sink node: There is a sink node $t$ with no out-going edges. All other nodes are called *internal nodes*.
- Given such a graph, an "$s - t$ flow" in the graph is a function $f$ that maps the edges to non-negative real numbers such that the following properties are satisfied:
  (a) Capacity constraint: For every edge $e$, $0 \leq f(e) \leq c(e)$.
  (b) Flow conservation: For every internal node $v$:

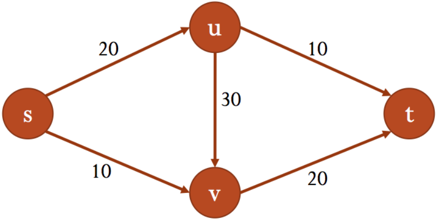
$$\sum_{e \text{ into } v} f(e) = \sum_{e \text{ out of } v} f(e)$$

## Problem

Find an $s - t$ flow $f$ in a given network graph such that the following quantity is maximized:

$$v(f) = \sum_{e \text{ out of } s} f(e)$$

- Example:



**Problem**

Find an $s-t$ flow $f$ in a given network graph such that the following quantity is maximized:

$$v(f) = \sum_{e \text{ out of } s} f(e)$$

- Example:

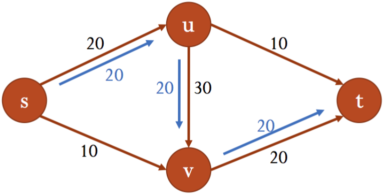



Figure: Routing 20 units of flow from $s$ to $t$. Is it possible to "push more flow"?

**Problem**

Find an $s-t$ flow $f$ in a given network graph such that the following quantity is maximized:

$$v(f) = \sum_{e \text{ out of } s} f(e)$$

- Example:

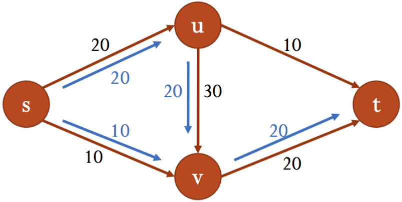



Figure: We should reset initial flow $(u, v)$ to 10.

## Problem

Find an $s-t$ flow $f$ in a given network graph such that the following quantity is maximized:

$$v(f) = \sum_{e \text{ out of } s} f(e)$$

- Example:

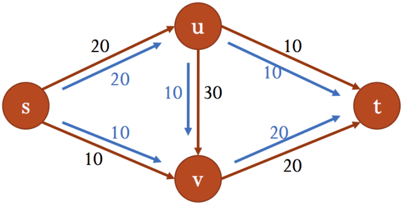
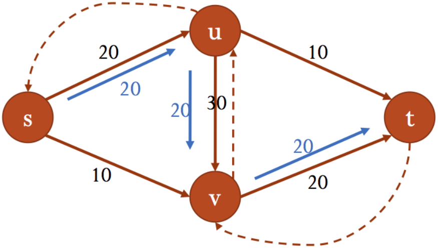



Figure: We should reset initial flow $(u, v)$ to 10. Maximum flow from $s$ is 30.

## Approach

- We will iteratively build larger $s - t$ flows.
- Given an $s - t$ flow $f$, we will build a residual graph $G_f$ that will allow us to reset flows along some of the edges.
- We will find an *augmenting path* in the residual graph $G_f$, push some flow along this path and update the flow $f'$.



## Approach

- We will iteratively build larger $s - t$ flows.
- Given an $s - t$ flow $f$, we will build a residual graph $G_f$ that will allow us to reset flows along some of the edges.
- We will find an *augmenting path* in the residual graph $G_f$, push some flow along this path and update the flow $f'$.

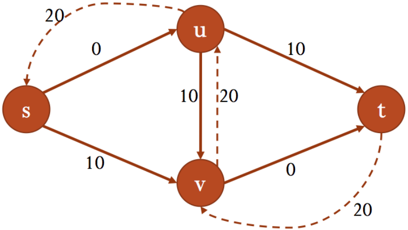



Figure: Graph $G_f$. *(f(s, u) = 20, f(s, v) = 0, f(u, v) = 20, f(u, t) = 0, f(v, t) = 20)*

## Approach

- We will iteratively build larger $s - t$ flows.
- Given an $s - t$ flow $f$, we will build a residual graph $G_f$ that will allow us to reset flows along some of the edges.
- We will find an *augmenting path* in the residual graph $G_f$, push some flow along this path and update the flow $f'$.

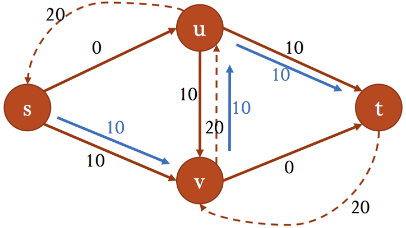



Figure: Augmenting path. $(f'(s, u) = 20, f'(s, v) = 10, f'(u, v) = 10, f'(u, t) = 10, f'(v, t) = 20)$

## Approach

- We will iteratively build larger $s-t$ flows.
- Given an $s-t$ flow $f$, we will build a residual graph $G_f$ that will allow us to reset flows along some of the edges.
- We will find an *augmenting path* in the residual graph $G_f$, push some flow along this path and update the flow $f'$.

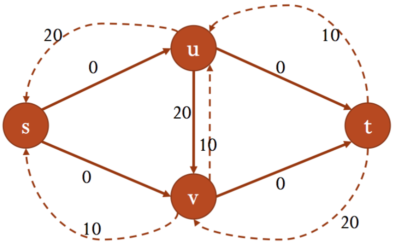



Figure: Graph $G_{f'}$. *(f'(s, u) = 20, f'(s, v) = 10, f'(u, v) = 10, f'(u, t) = 10, f'(v, t) = 20)*

- Residual graph $G_f$:
  - Forward edges: For every edge $e$ in the original graph, there are $(c(e) - f(e))$ units of more flow we can send along that edge. So, we set the weight of this edge $(c(e) - f(e))$.
  - Backward edges: For every edge $e = (u, v)$ in the original graph, there are $f(e)$ units of flow that we can undo. So we add a reverse edge $e' = (v, u)$ and set the weight of $e'$ to $f(e)$.

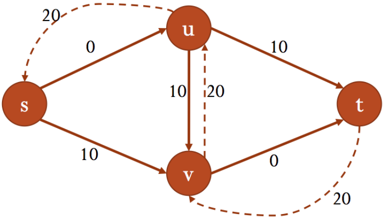



Figure: Graph $G_f$. *(f(s, u) = 20, f(s, v) = 0, f(u, v) = 20, f(u, t) = 0, f(v, t) = 20)*

- Augmenting flow in $G_f$:
  - Let $P$ be a simple $s - t$ path in $G_f$. Note that this contains forward and backward edges.
  - Let $e_{min}$ be an edge in the path $P$ with minimum weight $w_{min}$
  - For every forward edge $e$ in $P$, set $f'(e) \leftarrow f(e) + w_{min}$
  - For every backward edge $(x, y)$ in $P$, set $f'(y, x) \leftarrow f(y, x) - w_{min}$
  - For all remaining edges $e$, $f'(e) = f(e)$

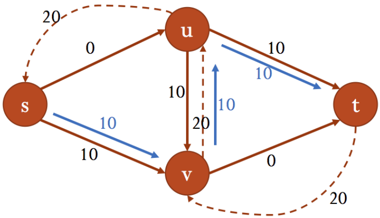



Figure: Augmenting path. *(f′(s, u) = 20, f′(s, v) = 10, f′(u, v) = 10, f′(u, t) = 10, f′(v, t) = 20)*

- Claim 1: $f'$ is an $s - t$ flow.
- Proof sketch:
  - Capacity constraint for each edge is satisfied.
  - Flow conservation at each vertex is satisfied.

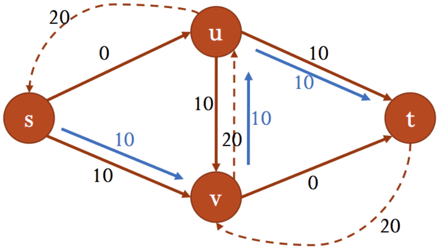



Figure: Augmenting path. *(f'(s, u) = 20, f'(s, v) = 10, f'(u, v) = 10, f'(u, t) = 10, f'(v, t) = 20)*

### Algorithm

`Ford-Fulkerson`

- Start with a flow $f$ such that $f(e) = 0$
- While there is an $s - t$ path $P$ in $G_f$
  - Augment flow along an $s - t$ path and let $f'$ be resulting flow
  - Update $f$ to $f'$ and $G_f$ to $G_{f'}$
- return($f$)

- What is the running time of the above algorithm?

## Algorithm

`Ford-Fulkerson`

- Start with a flow $f$ such that $f(e) = 0$
- While there is an $s - t$ path $P$ in $G_f$
  - Augment flow along an $s - t$ path and let $f'$ be resulting flow
  - Update $f$ to $f'$ and $G_f$ to $G_{f'}$
- return($f$)

- What is the running time of the above algorithm?
  - Claim 2: $v(f') > v(f)$.

## Algorithm

`Ford-Fulkerson`

- Start with a flow $f$ such that $f(e) = 0$
- While there is an $s - t$ path $P$ in $G_f$
  - Augment flow along an $s - t$ path and let $f'$ be resulting flow
  - Update $f$ to $f'$ and $G_f$ to $G_{f'}$
- return($f$)

- What is the running time of the above algorithm?
  - Claim 2: $v(f') > v(f)$.
  - Claim 3: The while loop runs for at most $C = \sum_{e \text{ out of } s} c(e)$ iterations.

## Algorithm

`Ford-Fulkerson`

- Start with a flow $f$ such that $f(e) = 0$
- While there is an $s - t$ path $P$ in $G_f$
  - Augment flow along an $s - t$ path and let $f'$ be resulting flow
  - Update $f$ to $f'$ and $G_f$ to $G_{f'}$
- return($f$)

- What is the running time of the above algorithm?
  - Claim 2: $v(f') > v(f)$.
  - Claim 3: The while loop runs for at most $C = \sum_{e \text{ out of } s} c(e)$ iterations.
  - Claim 4: Finding augmenting path and augmenting flow along this path takes $O(m)$ time.

## Algorithm

`Ford-Fulkerson`

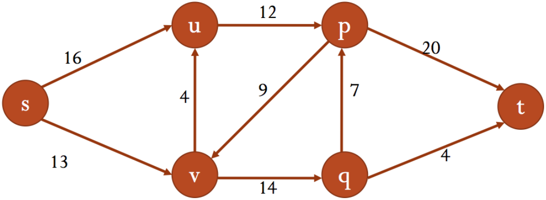
- Start with a flow $f$ such that $f(e) = 0$
- While there is an $s - t$ path $P$ in $G_f$
  - Augment flow along an $s - t$ path and let $f'$ be resulting flow
  - Update $f$ to $f'$ and $G_f$ to $G_{f'}$
- return$(f)$

- What is the running time of the above algorithm? $O(m \cdot C)$
  - Claim 2: $v(f') > v(f)$.
  - Claim 3: The while loop runs for at most $C = \sum_{e \text{ out of } s} c(e)$ iterations.
  - Claim 4: Finding augmenting path and augmenting flow along this path takes $O(m)$ time.

## Algorithm

Ford-Fulkerson

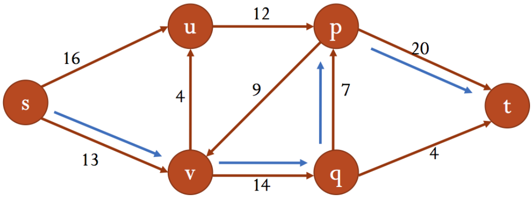
- Start with a flow $f$ such that $f(e) = 0$
- While there is an $s - t$ path $P$ in $G_f$
  - Augment flow along an $s - t$ path and let $f'$ be resulting flow
  - Update $f$ to $f'$ and $G_f$ to $G_{f'}$
- return($f$)



### Algorithm

`Ford-Fulkerson`

- Start with a flow $f$ such that $f(e) = 0$
- While there is an $s - t$ path $P$ in $G_f$
  - Augment flow along an $s - t$ path and let $f'$ be resulting flow
  - Update $f$ to $f'$ and $G_f$ to $G_{f'}$
- return($f$)



**Algorithm**

`Ford-Fulkerson`

- Start with a flow $f$ such that $f(e) = 0$
- While there is an $s-t$ path $P$ in $G_f$
  - Augment flow along an $s-t$ path and let $f'$ be resulting flow
  - Update $f$ to $f'$ and $G_f$ to $G_{f'}$
- return($f$)

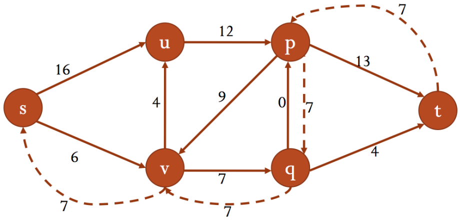
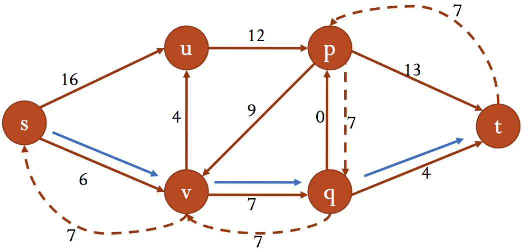



Figure: Graph $G_f$, where $f(s,u) = 0, f(s,v) = 7, f(v,u) = 0, f(v,q) = 7, f(u,p) = 0, f(p,v) = 0, f(p,t) = 7, f(q,p) = 7, f(q,t) = 0$

**Algorithm**

Ford-Fulkerson

- Start with a flow $f$ such that $f(e) = 0$
- While there is an $s - t$ path $P$ in $G_f$
  - Augment flow along an $s - t$ path and let $f'$ be resulting flow
  - Update $f$ to $f'$ and $G_f$ to $G_{f'}$
- return($f$)



**Algorithm**

`Ford-Fulkerson`

- Start with a flow $f$ such that $f(e) = 0$
- While there is an $s - t$ path $P$ in $G_f$
  - Augment flow along an $s - t$ path and let $f'$ be resulting flow
  - Update $f$ to $f'$ and $G_f$ to $G_{f'}$
- return($f$)

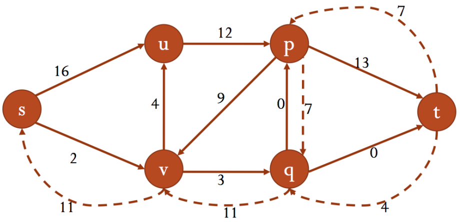
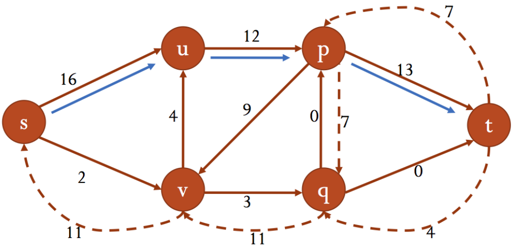



Figure: Graph $G_f$, where $f(s,u) = 0, f(s,v) = 11, f(v,u) = 0, f(v,q) = 11, f(u,p) = 0, f(p,v) = 0, f(p,t) = 7, f(q,p) = 7, f(q,t) = 4$

### Algorithm

`Ford-Fulkerson`

- Start with a flow $f$ such that $f(e) = 0$
- While there is an $s - t$ path $P$ in $G_f$
  - Augment flow along an $s - t$ path and let $f'$ be resulting flow
  - Update $f$ to $f'$ and $G_f$ to $G_{f'}$
- return($f$)



**Algorithm**

`Ford-Fulkerson`

- Start with a flow $f$ such that $f(e) = 0$
- While there is an $s - t$ path $P$ in $G_f$
  - Augment flow along an $s - t$ path and let $f'$ be resulting flow
  - Update $f$ to $f'$ and $G_f$ to $G_{f'}$
- return$(f)$

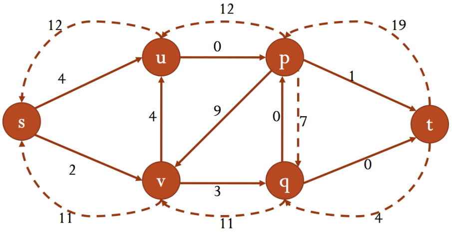



Figure: Graph $G_f$, where $f(s,u) = 12, f(s,v) = 11, f(v,u) = 0, f(v,q) = 11, f(u,p) = 12, f(p,v) = 0, f(p,t) = 19, f(q,p) = 7, f(q,t) = 4$

**Algorithm**

`Ford-Fulkerson`

- Start with a flow $f$ such that $f(e) = 0$
- While there is an $s - t$ path $P$ in $G_f$
  - Augment flow along an $s - t$ path and let $f'$ be resulting flow
  - Update $f$ to $f'$ and $G_f$ to $G_{f'}$
- return($f$)

- How do we prove that the flow returned by the `Ford-Fulkerson` algorithm is the maximum flow?

- Theorem 1: Let $f$ be the flow returned by the Ford-Fulkerson algorithm. Then $f$ maximizes $v(f) = \sum_{e \text{ out of } s} f(e)$.

**Definition ($f^{in}$ and $f^{out}$)**

Let $S$ be a subset of vertices and $f$ be a flow. Then

$$f^{in}(S) = \sum_{e \text{ into } S} f(e) \quad \text{and} \quad f^{out}(S) = \sum_{e \text{ out of } S} f(e)$$

- Theorem 1: Let $f$ be the flow returned by the `Ford-Fulkerson` algorithm. Then $f$ maximizes $v(f) = \sum_{e \text{ out of } s} f(e)$.

**Definition ($f^{in}$ and $f^{out}$)**

Let $S$ be a subset of vertices and $f$ be a flow. Then

$$f^{in}(S) = \sum_{e \text{ into } S} f(e) \quad \text{and} \quad f^{out}(S) = \sum_{e \text{ out of } S} f(e)$$

**Definition ($s - t$ cut)**

A partition of vertices $(A, B)$ is called an $s - t$ cut iff $A$ contains $s$ and $B$ contains $t$.

**Definition (Capacity of $s - t$ cut)**

The capacity of an $s - t$ cut $(A, B)$ is defined as $C(A, B) = \sum_{e \text{ out of } A} c(e)$.

- Theorem 1: Let $f$ be the flow returned by the `Ford-Fulkerson` algorithm. Then $f$ maximizes $v(f) = \sum_{e \text{ out of } s} f(e)$.

**Proof**

- Claim 1.1: For any $s - t$ cut $(A, B)$ and any $s - t$ flow $f$, $v(f) = f^{out}(A) - f^{in}(A)$.

- Theorem 1: Let $f$ be the flow returned by the `Ford-Fulkerson` algorithm. Then $f$ maximizes $v(f) = \sum_{e \text{ out of } s} f(e)$.

**Proof**

- Claim 1.1: For any $s - t$ cut $(A, B)$ and any $s - t$ flow $f$, $v(f) = f^{out}(A) - f^{in}(A)$.

**Proof of claim 1.1.**

$v(f) = f^{out}(\{s\}) - f^{in}(\{s\})$ and for all other nodes $v \in A, f^{out}(\{v\}) - f^{in}(\{v\}) = 0$. So,

$$v(f) = \sum_{v \in A} (f^{out}(\{v\}) - f^{in}(\{v\})) = f^{out}(A) - f^{in}(A).$$

□

- Theorem 1: Let $f$ be the flow returned by the `Ford-Fulkerson` algorithm. Then $f$ maximizes $v(f) = \sum_{e \text{ out of } s} f(e)$.

**Proof**

- Claim 1.1: For any $s$-$t$ cut $(A, B)$ and any $s$-$t$ flow $f$, $v(f) = f^{out}(A) - f^{in}(A)$.
- Claim 1.2: Let $f$ be any $s$-$t$ flow and $(A, B)$ be any $s$-$t$ cut. Then $v(f) \leq C(A, B)$.

- Theorem 1: Let $f$ be the flow returned by the `Ford-Fulkerson` algorithm. Then $f$ maximizes $v(f) = \sum_{e \text{ out of } s} f(e)$.

**Proof**

- Claim 1.1: For any $s$-$t$ cut $(A, B)$ and any $s$-$t$ flow $f$, $v(f) = f^{out}(A) - f^{in}(A)$.
- Claim 1.2: Let $f$ be any $s$-$t$ flow and $(A, B)$ be any $s$-$t$ cut. Then $v(f) \leq C(A, B)$.

**Proof of claim 1.2.**

$v(f) = f^{out}(A) - f^{in}(A) \leq f^{out}(A) \leq C(A, B)$. □

- Theorem 1: Let $f$ be the flow returned by the `Ford-Fulkerson` algorithm. Then $f$ maximizes $v(f) = \sum_{e \text{ out of } s} f(e)$.

**Proof**

- Claim 1.1: For any $s$-$t$ cut $(A, B)$ and any $s$-$t$ flow $f$, $v(f) = f^{out}(A) - f^{in}(A)$.
- Claim 1.2: Let $f$ be any $s$-$t$ flow and $(A, B)$ be any $s$-$t$ cut. Then $v(f) \leq C(A, B)$.
- Claim 1.3: Let $f$ be an $s$-$t$ flow such that there is no $s$-$t$ path in $G_f$. Then there is an $s$-$t$ cut $(A^*, B^*)$ such that $v(f) = C(A^*, B^*)$. Furthermore, $f$ is a flow with maximum value and $(A^*, B^*)$ is an $s$-$t$ cut with minimum capacity.

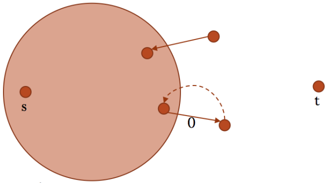
□

- Claim 1.3: Let $f$ be an $s$-$t$ flow such that there is no $s$-$t$ path in $G_f$. Then there is an $s$-$t$ cut $(A^*, B^*)$ such that $v(f) = C(A^*, B^*)$. Furthermore, $f$ is a flow with maximum value and $(A^*, B^*)$ is an $s$-$t$ cut with minimum capacity.

**Proof of claim 1.3**

- Let $A^*$ be all vertices reachable from $s$ in the graph $G_f$ (see figure below). Then we have:

$$\begin{aligned} v(f) &= f^{out}(A^*) - f^{in}(A^*) \\ &= f^{out}(A^*) - 0 \\ &= C(A^*, B^*) \end{aligned}$$




$A^*$ (all vertices reachable from $s$ in $G_f$)

**Theorem (Max-flow-min-cut theorem)**

*In every flow network, the maximum value of s-t flow is equal to the minimum capacity of s-t cut.*

- Summary:
  - `Ford-Fulkerson` Algorithm:
    - Given network with integer capacities, find a source-to-sink path and push as much flow along the path as possible.
    - Update the residual capacity of edges in the residual graph.
    - Repeat.
  - Proof of correctness:
    - The algorithm terminates (since the capacities are integers).
    - Max-flow-min-cut theorem: In every flow network, the maximum value of $s$-$t$ flow is equal to the minimum capacity of $s$-$t$ cut.

- Summary:
  - Ford-Fulkerson Algorithm:
    - Given network with integer capacities, find a source-to-sink path and push as much flow along the path as possible.
    - Update the residual capacity of edges in the residual graph.
    - Repeat.
  - Proof of correctness:
    - The algorithm terminates (since the capacities are integers).
    - Max-flow-min-cut theorem: In every flow network, the maximum value of $s$-$t$ flow is equal to the minimum capacity of $s$-$t$ cut.
- What if the capacities are not integers? Does the algorithm terminate?

- Summary:
  - `Ford-Fulkerson` Algorithm:
    - Given network with integer capacities, find a source-to-sink path and push as much flow along the path as possible.
    - Update the residual capacity of edges in the residual graph.
    - Repeat.
  - Proof of correctness:
    - The algorithm terminates (since the capacities are integers).
    - Max-flow-min-cut theorem: In every flow network, the maximum value of $s$-$t$ flow is equal to the minimum capacity of $s$-$t$ cut.
- What if the capacities are not integers? Does the algorithm terminate?
  - There is a network where the edges have non-integer capacities where the Ford-Fulkerson algorithm does not terminate.

# Applications of Network Flow

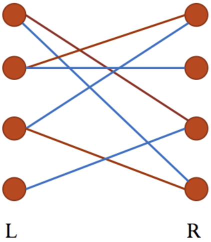
**Definition (Matching in bipartite graphs)**

A subset $M$ of edges such that each node appears in at most one edge in $M$.

**Problem**

Given a bipartite graph $G = (L, R, E)$, design an algorithm to give a maximum matching in the graph.

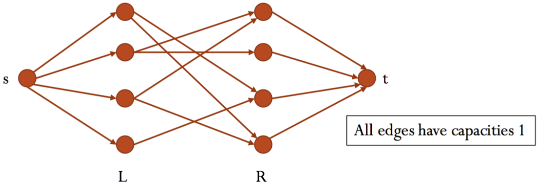
- Example:



## Problem

Given a bipartite graph $G = (L, R, E)$, design an algorithm to give a maximum matching in the graph.

- Consider the network graph below constructed from the bipartite graph.

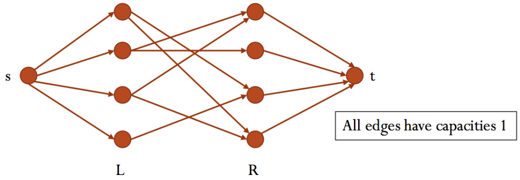



- Claim 1: Suppose there is an integer flow of value $k$ in the network graph. Then the bipartite graph has a matching of size $k$.

**Problem**

Given a bipartite graph $G = (L, R, E)$, design an algorithm to give a maximum matching in the graph.

- Consider the network graph below constructed from the bipartite graph.

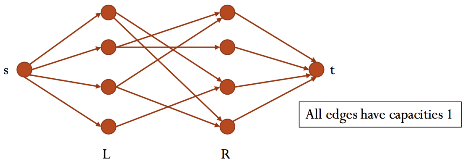



- Claim 1: Suppose there is an integer flow of value $k$ in the network graph. Then the bipartite graph has a matching of size $k$.
  - Consider those bipartite edges along which the flow is 1. Argue that due to flow conservation these edges form a matching.

**Problem**

Given a bipartite graph $G = (L, R, E)$, design an algorithm to give a maximum matching in the graph.

- Consider the network graph below constructed from the bipartite graph.

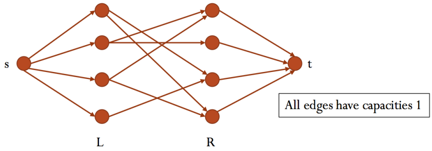



- Claim 1: Suppose there is an integer flow of value $k$ in the network graph. Then the bipartite graph has a matching of size $k$.
  - Consider those bipartite edges along which the flow is 1. Argue that due to flow conservation these edges form a matching.
- Claim 2: Suppose the bipartite graph has a matching of size $k$. Then there is an integer flow of value $k$ in the network graph.

**Problem**

Given a bipartite graph $G = (L, R, E)$, design an algorithm to give a maximum matching in the graph.

- Consider the network graph below constructed from the bipartite graph.




- Claim 1: Suppose there is an integer flow of value $k$ in the network graph. Then the bipartite graph has a matching of size $k$.
  - Consider those bipartite edges along which the flow is 1. Argue that due to flow conservation these edges form a matching.
- Claim 2: Suppose the bipartite graph has a matching of size $k$. Then there is an integer flow of value $k$ in the network graph.
  - Consider the flow where the flow along the edges in the matching is 1.

## Problem

Given a bipartite graph $G = (L, R, E)$, design an algorithm to give a maximum matching in the graph.

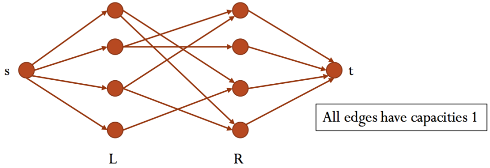



Figure: Network construction from Bipartite graph

## Algorithm

Max-Matching($G$)

- Construct the network $G'$ using $G$ as shown in Figure
- Execute the Ford-Fulkerson algorithm on $G'$ to obtain flow $f$
- Let $M$ be all bipartite edges with flow value 1
- return($M$)

## End