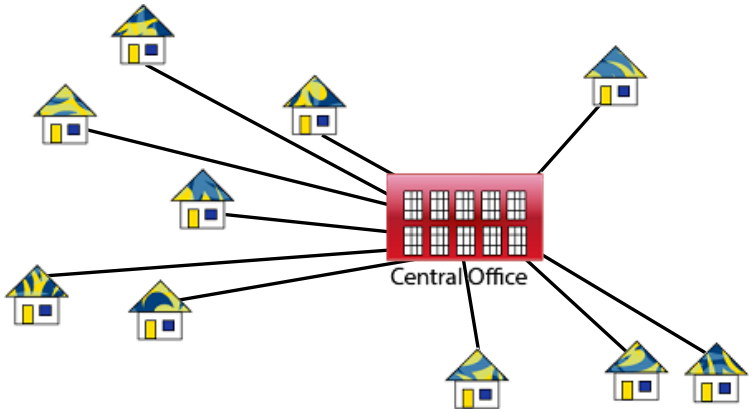
Graphs: Minimum Spanning Trees and Maximum Flows

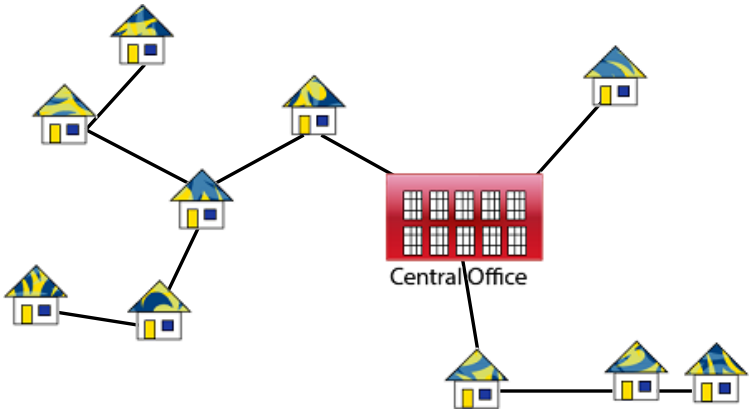
Jordi Cortadella and Jordi Petit
Department of Computer Science

Laying a communication network



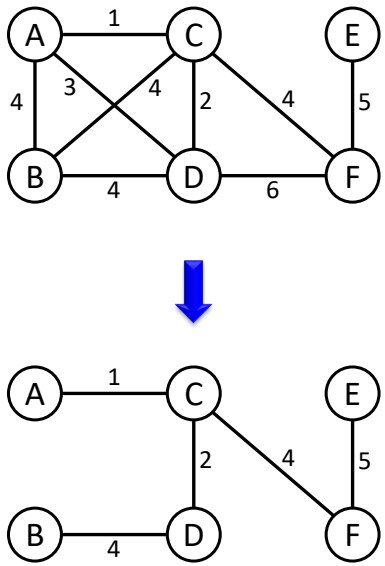
Source: https://www.javatpoint.com/applications-of-minimum-spanning-tree

Laying a communication network



Source: https://www.javatpoint.com/applications-of-minimum-spanning-tree

Minimum Spanning Trees



- Nodes are computers
- Edges are links
- Weights are maintenance cost
- Goal: pick a subset of edges such that
  - the nodes are connected
  - the maintenance cost is minimum

The solution is not unique.
Find another one !

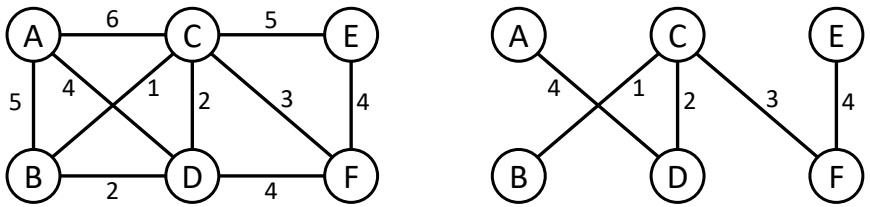
Property:
An optimal solution cannot contain a cycle.

Minimum Spanning Tree

- Given un undirected graph $G = (V, E)$ with edge weights w_e, find a tree $T = (V, E')$, with $E' \subseteq E$, that minimizes

$$\text{weight}(T) = \sum_{e \in E'} w_e \,.$$

- Greedy algorithm: repeatedly add the next lightest edge that does not produce a cycle.

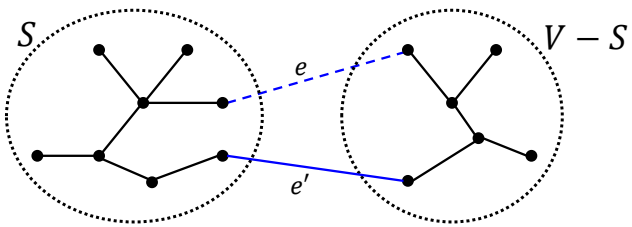


Note: We will now see that this strategy guarantees an MST.

Properties of trees

- **Definition:** A tree is an undirected graph that is connected and acyclic.
- **Property:** Any connected, undirected graph $G = (V, E)$ has $|E| \geq |V| - 1$ edges.
- **Property:** A tree on n nodes has $n - 1$ edges.
  - Start from an empty graph. Add one edge at a time making sure that it connects two disconnected components. After having added $n - 1$ edges, a tree has been formed.
- **Property:** Any connected, undirected graph $G = (V, E)$ with $|E| = |V| - 1$ is a tree.
  - It is sufficient to prove that G is acyclic. If not, we can always remove edges from cycles until the graph becomes acyclic.
- **Property:** Any undirected graph is a tree iff there is a unique path between any pair of nodes.
  - If there would be two paths between two nodes, the union of the paths would contain a cycle.

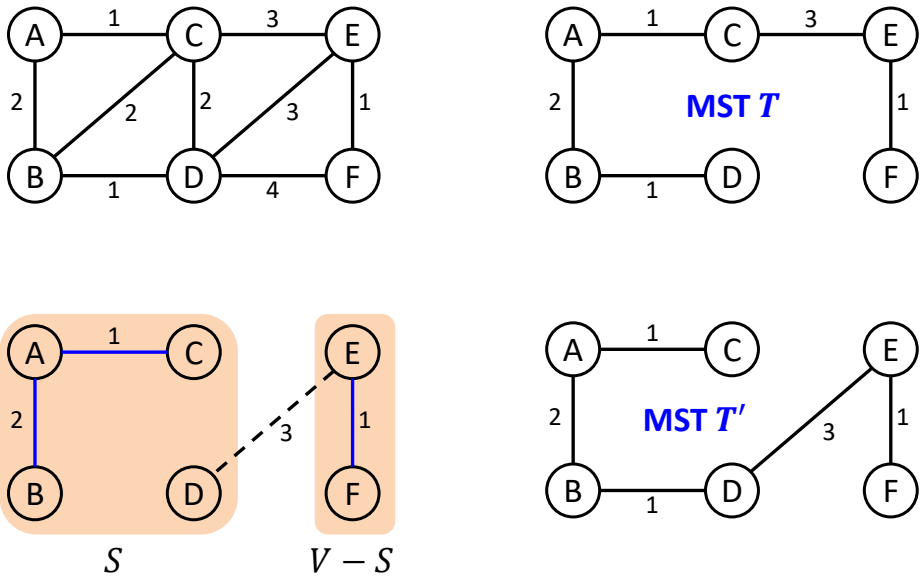
The cut property



Suppose edges X are part of an MST of $G = (V, E)$. Pick any subset of nodes S for which X does not cross between S and $V - S$, and let e be the lightest edge across this partition. Then $X \cup \{e\}$ is part of some MST.

Proof (sketch): Let T be an MST and assume e is not in T. If we add e to T, a cycle will be created with another edge e' across the cut $(S, V - S)$. We can now remove e' and obtain another tree T' with $\text{weight}(T') \leq \text{weight}(T)$. Since T is an MST, then the weights must be equal.

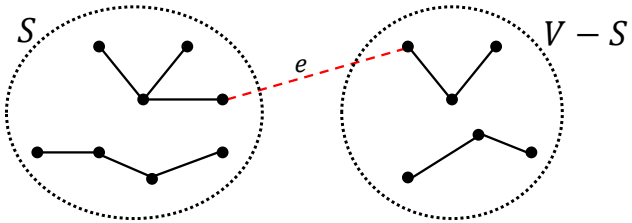
The cut property: example



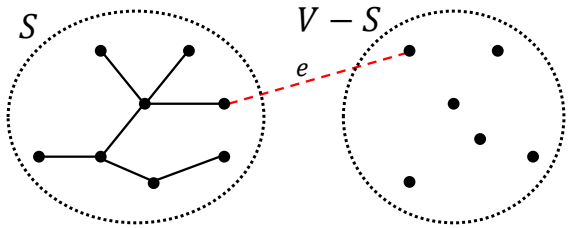
Minimum Spanning Tree

Any scheme like this works (because of the properties of trees):

```
X = { }    // The set of edges of the MST
repeat |V| − 1 times:
  pick a set S ⊂ V for which X has no edges between S and V − S
  let e ∈ E be the minimum-weight edge between S and V − S
  X = X ∪ {e}
```



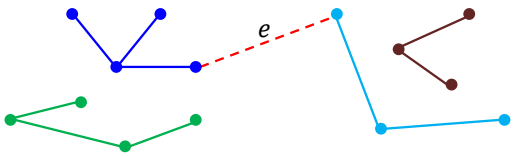
MST: two strategies



Prim's algorithm

Invariant:
- A set of nodes (S) is in the tree.

Progress:
- The lightest edge with exactly one endpoint in S is added.



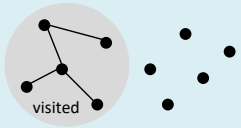
Kruskal's algorithm

Invariant:
- A set of trees (forest) has been constructed.

Progress:
- The lightest edge between two trees is added.

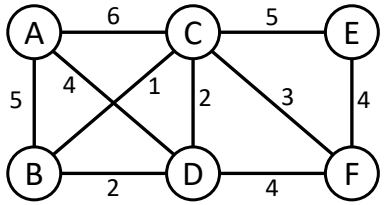
Prim's algorithm

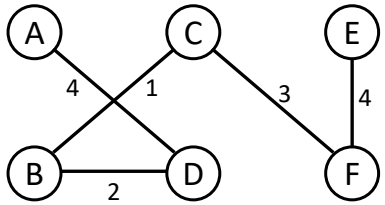
```
function Prim(G, w)
// Input: A connected undirected Graph G(V,E)
//        with edge weights w(e).
// Output: An MST defined by the vector prev.
  for all u ∈ V:
    visited(u)= false
    prev(u)= nil
  pick any initial node u0
  visited(u0) = true
  n = 1

  // Q: priority queue of edges using w(e) as priority
  Q = makequeue()
  for each (u0,v) ∈ E: Q.insert(u0,v)

  while n < |V|:
    (u,v) = deletemin(Q) // Edge with smallest weight
    if not visited(v):
      visited(v) = true
      prev(v) = u
      n = n + 1
      for each (v,x) ∈ E:
        if not visited(x): Q.insert(v,x)
```



Complexity: $O(|E| \log |V|)$

Prim's algorithm





Q:
(AD,4) (AB,5) (AC,6)
(DB,2) (DC,2) (DF,4) (AB,5) (AC,6)
(BC,1) (DC,2) (DF,4) (AB,5) (AC,6)
(DC,2) (CF,3) (DF,4) (AB,5) (CE,5) (AC,6)
(CF,3) (DF,4) (AB,5) (CE,5) (AC,6)
(DF,4) (FE,4) (AB,5) (CE,5) (AC,6)
(FE,4) (AB,5) (CE,5) (AC,6)

Kruskal's algorithm

Informal algorithm:

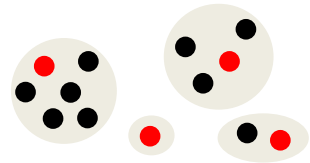
- Sort edges by weight.
- Visit edges in ascending order of weight and add them as long as they do not create a cycle.

How do we know whether a new edge will create a cycle?

```
function Kruskal(G, w)

// Input: A connected undirected Graph G(V,E)
//        with edge weights w_e.

// Output: An MST defined by the edges in X.

  X = { }
  sort the edges in E by weight
  for all (u,v) ∈ E, in ascending order of weight:
      if (X has no path connecting u and v):
          X = X ∪ {(u,v)}
```


Disjoint sets

- A data structure to store a collection of disjoint sets.



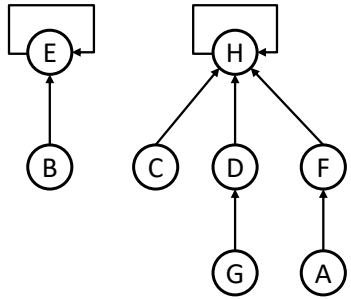
- Operations:
  - makeset(x): creates a singleton set containing just x.
  - find(x): returns the identifier of the set containing x.
  - union(x, y): merges the sets containing x and y.
- Kruskal's algorithm uses disjoint sets and calls
  - makeset: $|V|$ times
  - find: $2 \cdot |E|$ times
  - union: $|V| - 1$ times

Kruskal's algorithm

```
function Kruskal(G, w)

// Input: A connected undirected Graph G(V,E)
//        with edge weights w_e.

// Output: An MST defined by the edges in X.

  for all u ∈ V: makeset(u)

  X = { }
  sort the edges in E by weight
  for all (u,v) ∈ E, in ascending order of weight:
      if (find(u) ≠ find(v)):
          X = X ∪ {(u,v)}
          union(u,v)
```


Disjoint sets

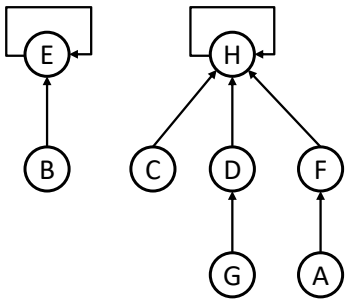
- The nodes are organized as a set of trees. Each tree represents a set.
- Each node has two attributes:
  - parent (π): ancestor in the tree
  - rank: height of the subtree
- The root element is the representative for the set: its parent pointer is itself (self-loop).
- The efficiency of the operations depends on the height of the trees.




```
function makeset(x):
  π(x) = x
  rank(x) = 0

function find(x):
  while x ≠ π(x): x = π(x)
  return x
```

Disjoint sets

```
function union(x, y):
  r_x = find(x)
  r_y = find(y)
  if r_x = r_y: return

  if rank(r_x) > rank(r_y):
    π(r_y) = r_x
  else:
    π(r_x) = r_y
    if rank(r_x) = rank(r_y):
      rank(r_y) = rank(r_y) + 1
```

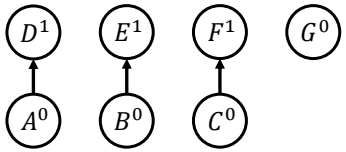



```
function makeset(x):
  π(x) = x
  rank(x) = 0

function find(x):
  while x ≠ π(x): x = π(x)
  return x
```

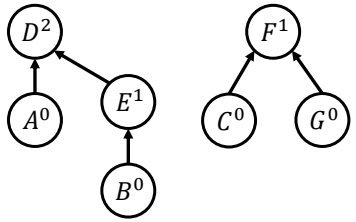

Disjoint sets

After makeset(A),…,makeset(G):

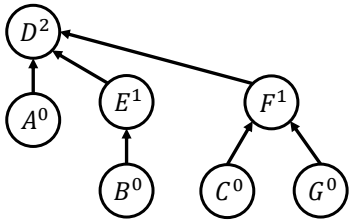
After union(A,D), union(B,E), union(C,F):



After union(C,G), union(E,A):



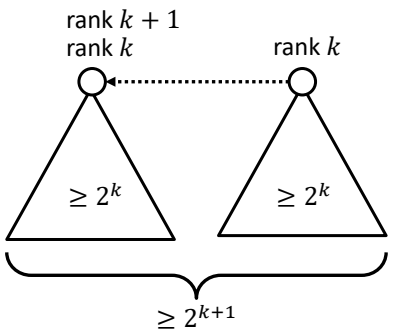
After union(B,G):



Property: Any root node of rank k has at least 2^k nodes in its tree.
Property: If there are n elements overall, there can be at most $n/2^k$ nodes of rank k.
Therefore, all trees have height $\leq \log n$.

Disjoint sets

Property 1: proof by induction



Property 2:

For n nodes, the tallest possible tree could have rank k, such that:

$$n \geq 2^k$$

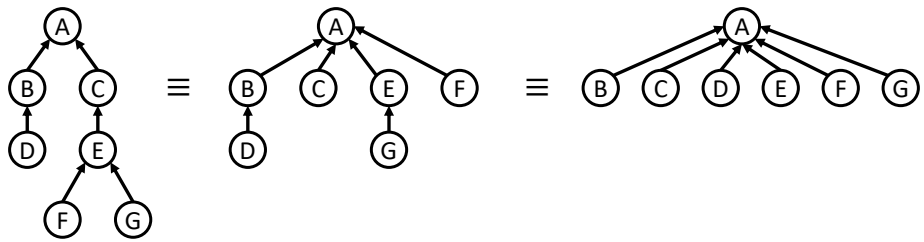
$$k \leq \log_2 n$$

Therefore, find(x) is $O(\log n)$

Property 1: Any root node of rank k has at least 2^k nodes in its tree.
Property 2: If there are n elements overall, there can be at most $n/2^k$ nodes of rank k.
Therefore, all trees have height $\leq \log n$.

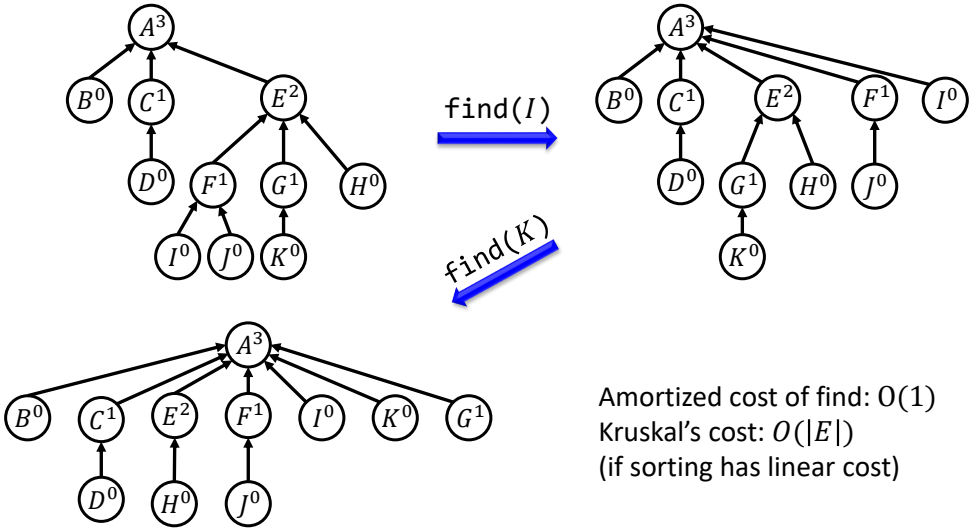
Disjoint sets: path compression

- Complexity of Kruskal's algorithm: $O(|E| \log |V|)$.
  - Sorting edges: $O(|E| \log|E|) = O(|E| \log |V|)$.
  - Find + union ($2 \cdot |E|$ times): $O(|E| \log|V|)$.
- How about if the edges are already sorted or sorting can be done in linear time (weights are integer and small)?
- Path compression:



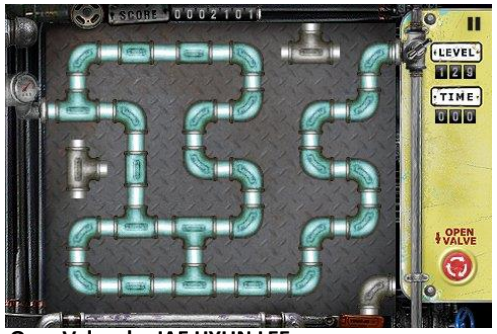
Disjoint sets: path compression

```
function find(x):
  if x ≠ π(x): π(x)= find(π(x))
  return π(x)
```



Amortized cost of find: $O(1)$
Kruskal's cost: $O(|E|)$
(if sorting has linear cost)

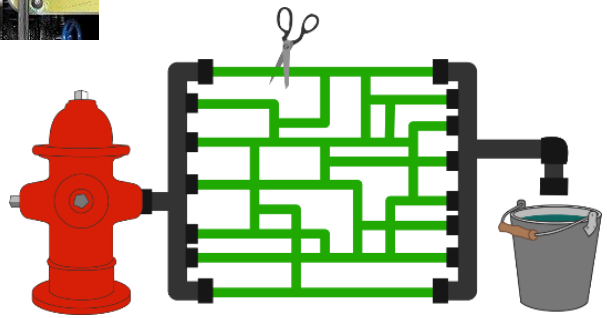
Max-flow/min-cut problems



OpenValve, by JAE HYUN LEE

How much water can you pump from source to target?

What is the fewest number of green tubes that need to be cut so that no water will be able to flow from the hydrant to the bucket?



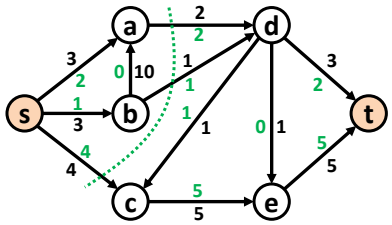
Max-flow/Min-cut algorithm. Brilliant.org.
https://brilliant.org/wiki/max-flow-min-cut-algorithm/

Max-flow/min-cut problems: applications

- Networks that carry data, water, oil, electricity, cars, etc.
  - How to maximize usage?
  - How to minimize cost?
  - How to maximize reliability?
- Multiple application domains:
  - Computer networks
  - Image processing
  - Computational biology
  - Airline scheduling
  - Data mining
  - Distributed computing
  - …

Max-flow problem

Model:
- A directed graph $G = (V, E)$.
- Two special nodes $s, t \in V$.
- Capacities $c_e > 0$ on the edges.



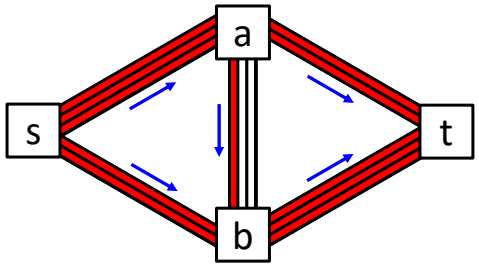
Goal: assign a flow f_e to each edge e of the network satisfying:
- $0 \le f_e \le c_e$ for all $e \in E$ (edge capacity not exceeded)
- For all nodes u (except s and t), the flow entering the node is equal to the flow exiting the node:

$$\sum_{(w,u)\in E} f_{wu} = \sum_{(u,z)\in E} f_{uz}\,.$$

Size of a flow: total quantity sent from s to t (equal to the quantity leaving s):

$$\text{size}(f) = \sum_{(s,u)\in E} f_{su}$$

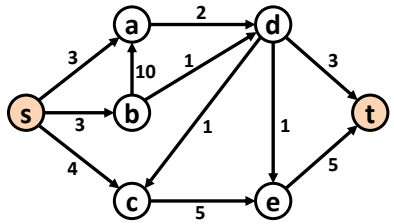
Max-flow problem: intuition

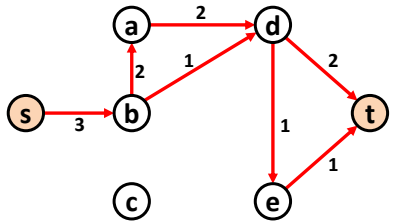


Find an augmenting path

An augmenting path may reverse some of the flow previously assigned

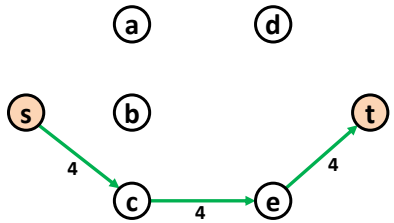
Augmenting paths



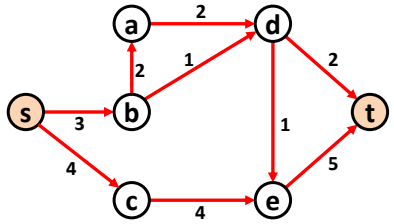


Flow

Given a flow, an ***augmenting path*** represents a feasible additional flow from s to t.

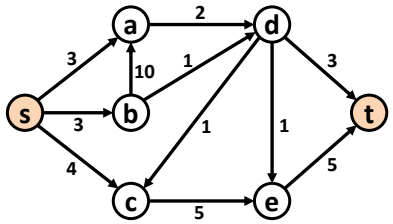


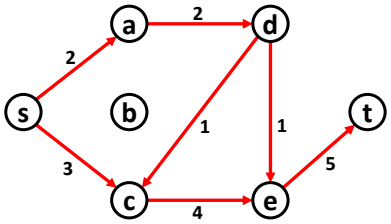
Augmenting path



New flow

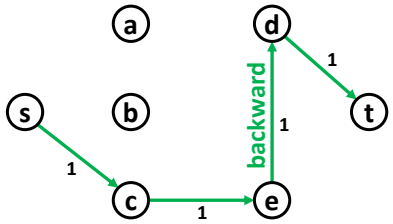
Augmenting paths



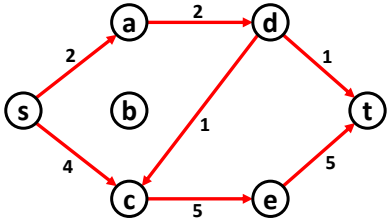


Flow

Augmenting paths can have *forward* and *backward* edges.



Augmenting path



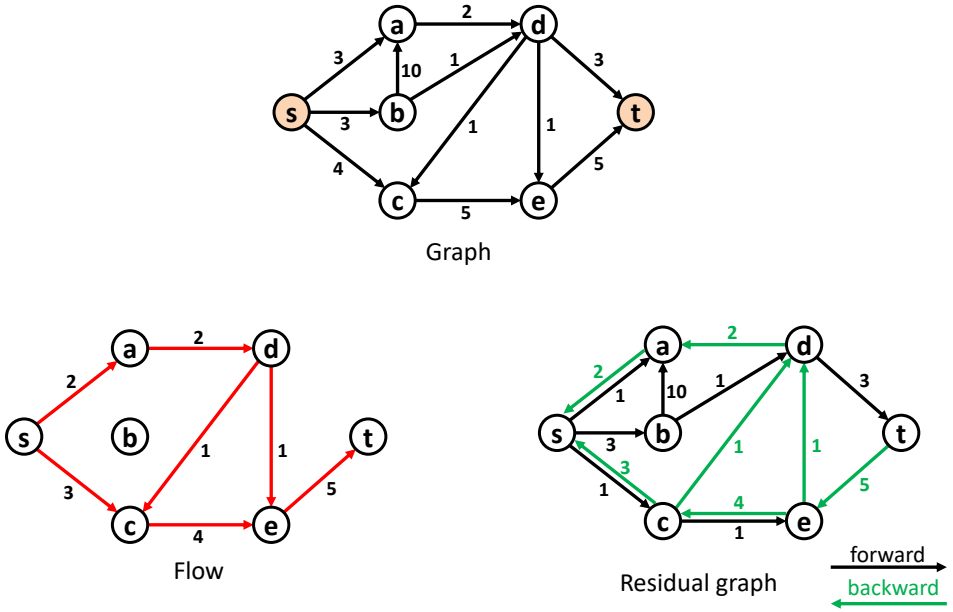
New flow

Augmenting paths

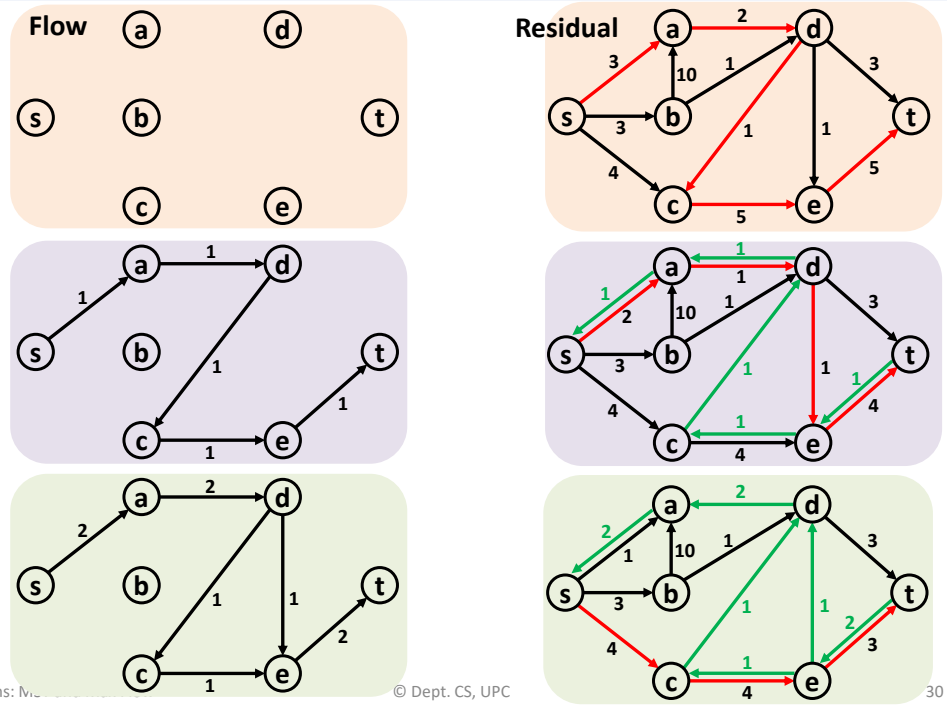
Given a flow f, an augmenting path is a directed path from s to t, which consists of edges from E, but not necessarily in the same direction. Each of these edges e satisfies exactly one of the following two conditions:

- e is in the same direction as in E (forward) and $f_e < c_e$. The difference $c_e - f_e$ is called the *slack* of the edge.
- e is in the opposite direction (backward) and $f_e > 0$. It represents the fact that some flow can be borrowed from the current flow.

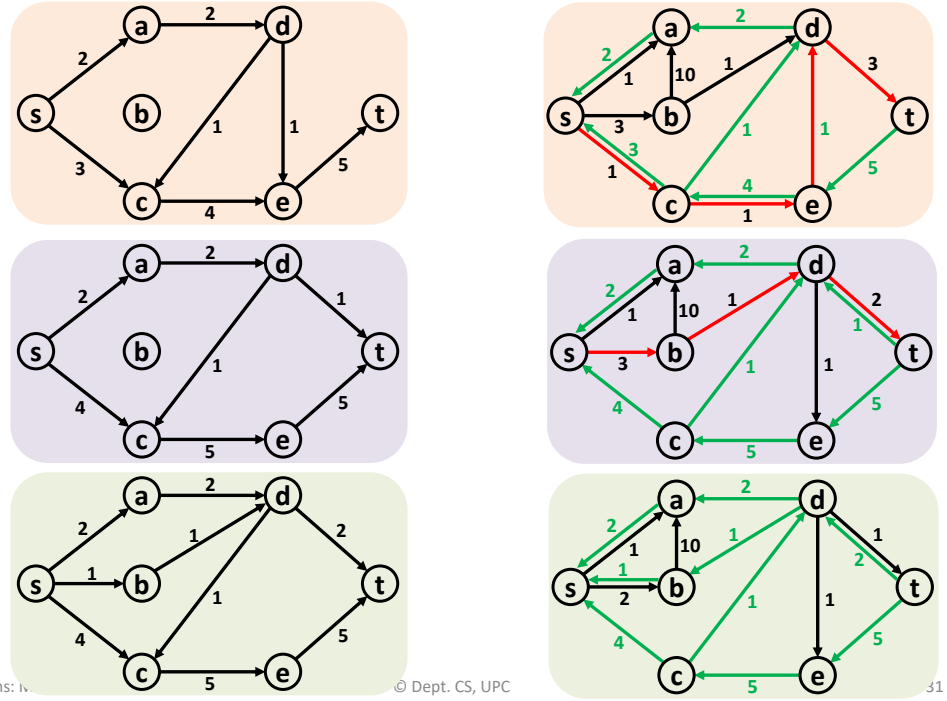
Residual graph



Ford-Fulkerson algorithm: example



Ford-Fulkerson algorithm: example



Ford-Fulkerson algorithm

```
function Ford-Fulkerson(G,s,t)

// Input: A directed Graph G(V,E) with edge capacities c_e.
//        s and t and the source and target of the flow.
// Output: A flow f that maximizes the size of the flow.
//         For each (u,v) ∈ E, f(v,u) represents its flow.

  for all (u,v) ∈ E:
    f(u,v) = c(u,v)   // Forward edges
    f(v,u) = 0        // Backward edges

  while there exists a path p = s ⇝ t in the residual graph:
    f(p) = min{f(u,v): (u,v) ∈ p}
    for all (u,v) ∈ p:
      f(u,v) = f(u,v) − f(p)
      f(v,u) = f(v,u) + f(p)
```

Ford-Fulkerson algorithm: complexity

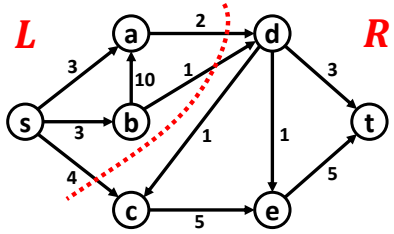
- Finding a path in the residual graph requires $O(|E|)$ time (using BFS or DFS).

- How many iterations (augmenting paths) are required?
  - The worst case is really bad: $O(C \cdot |E|)$, with C being the largest capacity of an edge (if only integral values are used).
  - By selecting the path with fewest edges (using BFS) the maximum number of iterations is $O(|V| \cdot |E|)$.
  - By carefully selecting *fat* augmenting paths (using some variant of Dijkstra's algorithm), the number of iterations can be reduced.

- Ford-Fulkerson algorithm is $O(|V| \cdot |E|^2)$ if BFS is used to select the path with fewest edges (Edmonds-Karp algorithm).

Max-flow problem

Cut: An (s,t)-cut partitions the nodes into two disjoint groups, L and R, such that $s \in L$ and $t \in R$.



For any flow f and any (s,t)-cut (L,R):

$$\text{size}(f) \leq \text{capacity}(L,R).$$

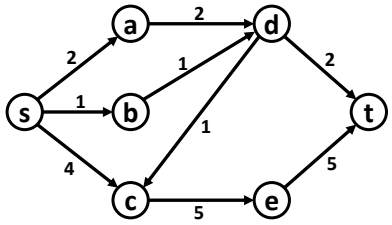
The max-flow min-cut theorem:

The size of the maximum flow equals the capacity of the smallest (s,t)-cut.

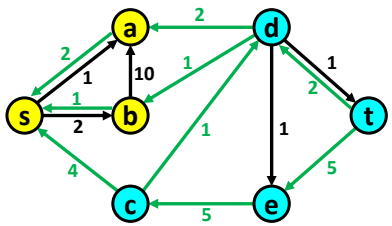
The augmenting-path theorem:

A flow is maximum iff it admits no augmenting path.

Min-cut algorithm



Flow

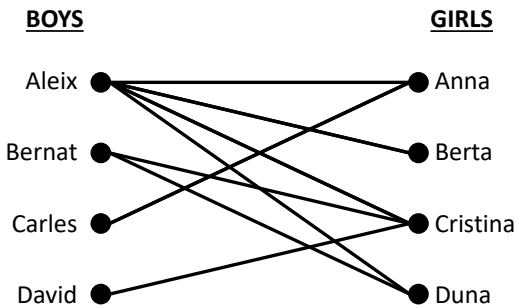


Residual graph

Finding a cut with minimum capacity:

1. Solve the max-flow problem with Ford-Fulkerson.
2. Compute L as the set of nodes reachable from s in the residual graph.
3. Define $R = V - L$.
4. The cut (L,R) is a min-cut.

Bipartite matching



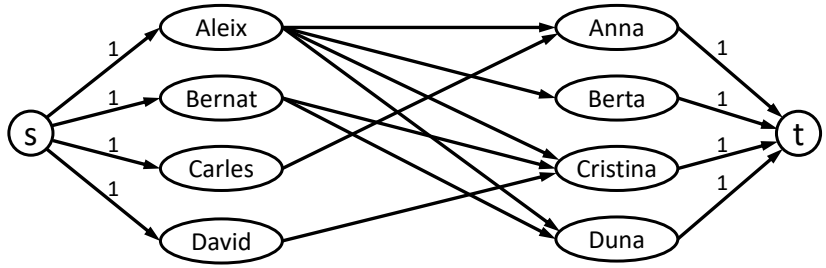
There is an edge between a boy and a girl if they like each other.

Can we pick couples so that everyone has exactly one partner that he/she likes?

Bad matching: if we pick (Aleix, Anna) and (Bernat, Cristina), then we cannot find couples for Berta, Duna, Carles and David.

A ***perfect matching*** would be: (Aleix, Berta), (Bernat, Duna), (Carles, Anna) and (David, Cristina).

Bipartite matching



Reduced to a max-flow problem with $c_e = 1$.

Question: can we always guarantee an integer-valued flow?

Property: if all edge capacities are integer, then the optimal flow found by Ford-Fulkerson's algorithm is integral. It is easy to see that the flow of the augmenting path found at each iteration is integral.

Extensions of Max-Flow

- **Max-Flow with Edge Demands**
  - Each edge e has a demand $d(e)$. The flow f must satisfy $d(e) \leq f(e) \leq c(e)$.
- **Node Supplies and Demands**
  - An extra flow $x(v)$ can be injected (positive) or extracted (negative) at every vertex v. The flow must satisfy:

$$\sum_{u \in V} f(u \to v) - \sum_{w \in V} f(v \to w) = x(v)\,.$$

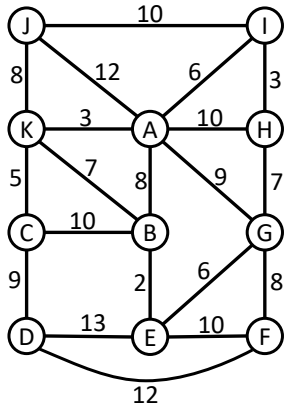
- **Min-cost Max-Flow**
  - Each edge e has a weight w_e. Compute a max-flow of minimum cost:

$$\text{cost}(f) = \sum_{e \in E} w_e \cdot f(e)$$

- **Max-Weight Bipartite Matching**
  - Each edge e has a weight w_e. Find a maximum cardinality matching with maximum total weight.

EXERCISES

Minimum Spanning Trees

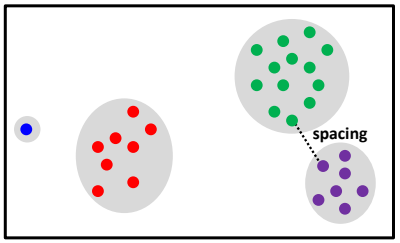


- Calculate the shortest path tree from node A using Dijkstra's algorithm.
- Calculate the MST using Prim's algorithm. Indicate the sequence of edges added to the tree and the evolution of the priority queue.
- Calculate the MST using Kruskal's algorithm. Indicate the sequence of edges added to the tree and the evolution of the disjoint sets. In case of a tie between two edges, try to select the one that is not in Prim's tree.

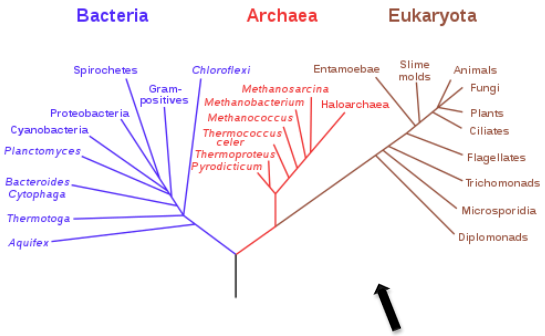
k-clustering of maximum spacing

We want to classify a set of points into k clusters. We define the distance between two points as the Euclidean distance. We define the *spacing* of the clustering as the minimum distance between any pair of points in different clusters.

Describe an algorithm such that, given an integer k, finds a k-clustering such that *spacing* is maximized. Argue about the complexity of the algorithm.

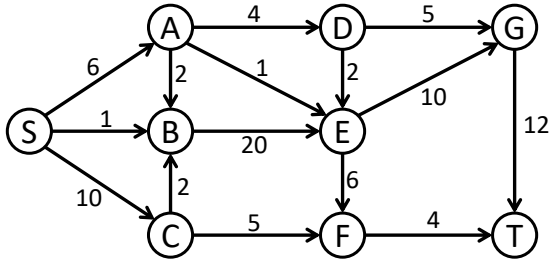


$k = 4$



Note: k-clustering of maximum spacing is the basis for the construction of dendograms.

Flow network (from [DVP2008])



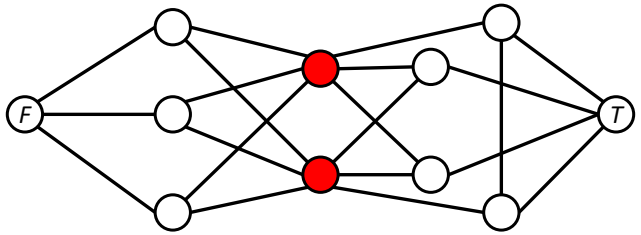
- Find the maximum flow from S to T. Give a sequence of augmenting paths that lead to the maximum flow.
- Draw the residual graph after finding the maximum flow.
- Find a minimum cut between S and T.

Contagious disease

The island of Sodor is home to a large number of towns and villages, connected by an extensive rail network. Recently, several cases of a deadly contagious disease (Covid 19) have been reported in the village of Ffarquhar. The controller of the Sodor railway plans to close down certain railway stations to prevent the disease from spreading to Tidmouth, his home town. No trains can pass through a closed station. To minimize expense (and public notice), he wants to close down as few stations as possible. However, he cannot close the Ffarquhar station, because that would expose him to the disease, and he cannot close the Tidmouth station, because then he couldn't visit his favorite pub.

Describe and analyze an algorithm to find the minimum number of stations that must be closed to block all rail travel from Ffarquhar to Tidmouth. The Sodor rail network is represented by an undirected graph, with a vertex for each station and an edge for each rail connection between two stations. Two special vertices *F* and *T* represent the stations in Ffarquhar and Tidmouth.

For example, given the following input graph, your algorithm should return the number 2.



Source: Jeff Erickson, Algorithms, UIUC, 2015.

Blood transfusion

Enthusiastic celebration of a sunny day at a prominent northeastern university has resulted in the arrival at the university's medical clinic of 169 students in need of emergency treatment. Each of the 169 students requires a transfusion of one unit of whole blood. The clinic has supplies of 170 units of whole blood. The number of units of blood available in each of the four major blood groups and the distribution of patients among the groups is summarized below.

Blood type	A	B	O	AB
Supply	46	34	45	45
Demand	39	38	42	50

Type A patients can only receive type A or O; type B patients can receive only type B or O; type O patients can receive only type O; and type AB patients can receive any of the four types.

Give a maxflow formulation that determines a distribution that satisfies the demands of a maximum number of patients.

Can we have enough blood units for all the students?

Source: Sedgewick and Wayne, Algorithms, 4th edition, 2011.

Edge-disjoint paths

Given a digraph $G = (V, E)$ and vertices $s, t \in V$, describe an algorithm that finds the maximum number of edge-disjoint paths from s to t.

Note: two paths are edge-disjoint if they do not share any edge.