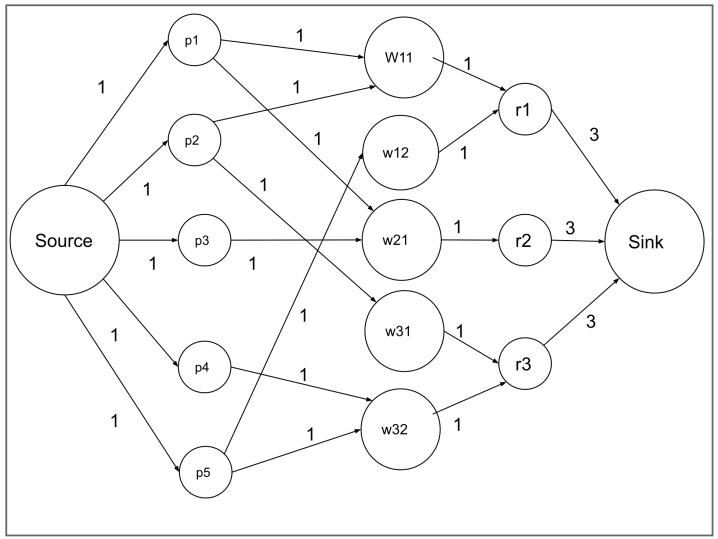
4M26 - Algorithms and Data Structures

February, 2023



Lectures 9-11 - Graph Algorithms

Slides: Ignas Budvytis

What is a Graph?

Graph, $G(V, E)$, is a mathematical object consisting of a set of vertices, V, and a set of edges, E, connecting these vertices. Here, V, amounts to a set of objects, some pairs of which, E, are in some sense related.

Common representations of graphs:

- **Adjacency list** - an array, Adj, of $|V|$ lists. For each vertex $u \in V$, list $Adj\,[u]$ contains all vertices, v, such that there is an edge, $(u, v) \in E$. Memory required: $\Theta(|V| + |E|)$.

- **Adjacency matrix** representation of a graph is a $|V| \times |V|$ matrix, A, such that: $A(u, v) = \begin{cases} 1 & \text{if } (u, v) \in E, \\ 0 & \text{otherwise.} \end{cases}$ Memory required: $\Theta(|V|^2)$.

Types of graphs: directed, undirected, weighted, unweighted.

Graph algorithms

- Graph search/traversal
    - Breadth-first search (BFS), depth-first search (DFS)
- Minimum weight spanning trees
    - Kruskal's and Primm's algorithms
- Shortest path algorithms
    - Bellman-Ford, Dijkstra, Floyd-Warshall
- Maximum flow
    - Ford-Fulkerson

Applications:

- Optimal control and RL: path planning.
- Internet and networking: routing protocols, web crawlers.
- Decision making, analysis and planning: job scheduling, resource allocation, network design, social network analysis.
- Algorithm design: strategies for other algorithms.

Graph Search/Traversal

Breadth-first search

Breadth-first search (BFS) is one of the simplest algorithms for searching a graph and the archetype of many important graph algorithms such as Primm's minimum spanning tree algorithm, Dijkstra's single-source shortest path algorithms.

BFS is called so as it expands the frontier between discovered and undiscovered vertices uniformly accross the breadth of the frontier. This algorithm discovers all vertices at distance k from vertex, s, before discovering any vertices at distance $k + 1$.

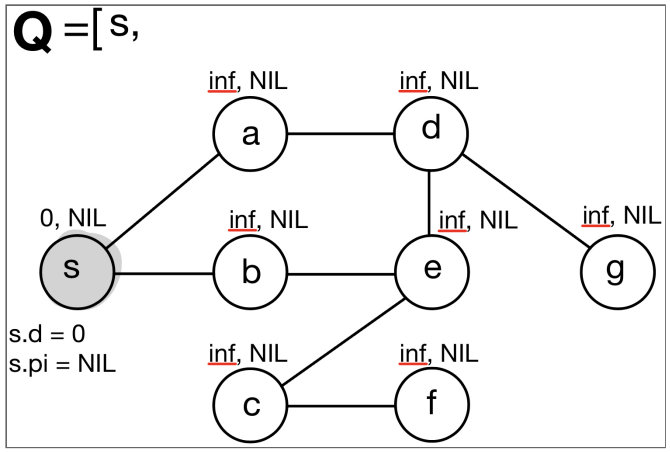
Output: Given a graph, $G = (V, E)$, and a **source** vertex, s:

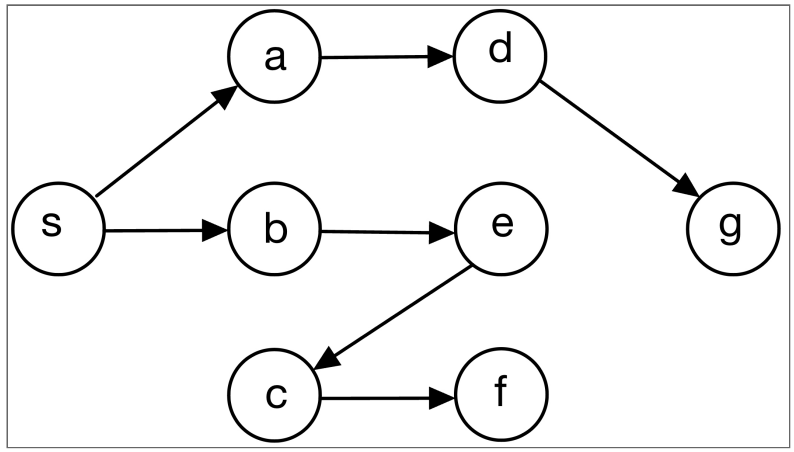
- discover every vertex that is reachable from s,
- discover distance, $v.d$, (smallest number of edges) from s to each reachable vertex, v,
- produce a *breadth-first tree* ($v.\pi$ indicates the parent of vertex, v, in such tree) with root, s, that contains all reachable vertices.

Key steps

- Classify vertices into: undiscovered (white), a frontier between discovered and undiscovered vertices (gray) and discovered vertices (black).
- Starting from the first vertice on frontier (obtained from a FIFO queue) append all undiscovered neighbouring nodes in the queue.
- Proceed until all vertices are discovered.

Example graph and initialisation (left), obtained breadth-first tree (right).





Breadth-first search pseudocode

Runtime: $O(|V| + |E|)$.

```
    INITIALIZE-BFS(G, s)
1   for each vertex, u ∈ G.V − {s}
2       u.color = WHITE
3       u.d = ∞
4       u.π = NIL
5   s.color = GRAY
6   s.d = 0
7   s.π = NIL
8   Q = ∅
9   return Q
```

```
    BFS(G, s)
1   Q = INITIALIZE-BFS (G, s)
2   ENQUEUE (Q, s)
3   while Q ≠ ∅
4       u = DEQUEUE (Q)
5       for each v ∈ G.Adj [u]
6           if v.color == WHITE
7               v.color = GRAY
8               v.d = u.d + 1
9               v.π = u
10              ENQUEUE (Q, v)
11      u.color = BLACK
```

Graph Search/Traversal

Depth-first search

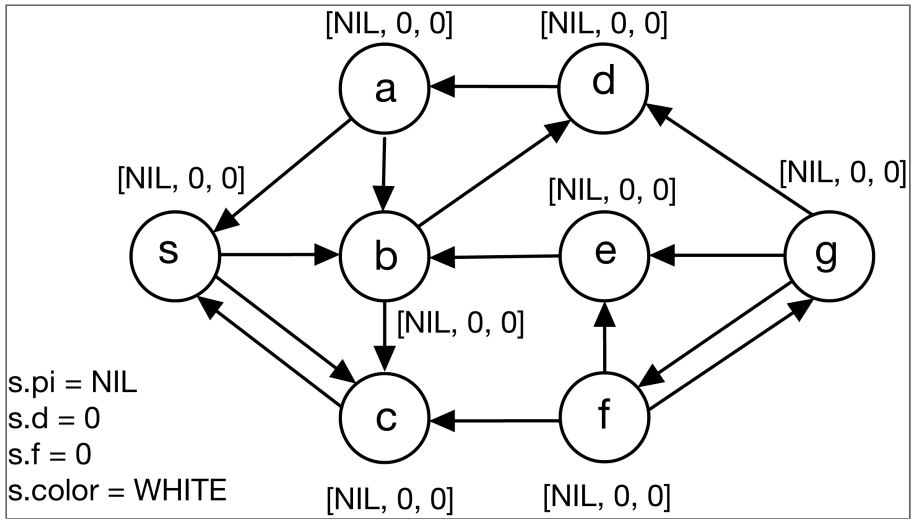
Depth-first search (DFS) works by choosing a first undiscovered vertice and exploring its neighbours recursively until no unexplored neighbours are left. For each vertex, v, once all out-going edges are explored, the algorithm *backtracks* to vertex from which v was discovered.

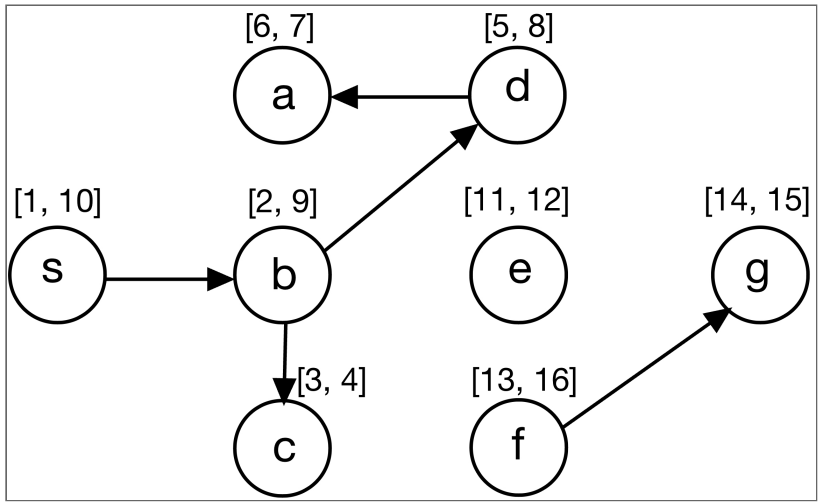
Coloring scheme is used to mark undiscovered vertices in white, the vertices all of which's neighbours were explored in black and gray - the rest.

Output: Given a graph, $G = (V, E)$, compute a predescessor subgraph, $G_\pi = (V, E_\pi)$, where:

- E_π=$\{(v.\pi, v) : v \in V \text{ and } v.\pi \neq \text{NIL}\}$.
- The predecessor graph comprises of several depth-first trees.
- Obtain structural information about the graph by recording **discovery time**, $v.d$, and **finish time**, $v.f$, of the search for each vertex.

Example graph and initialisation (left), obtained depth-first forest (right).





Depth-first search - pseudocode

Runtime: $\Theta(|V| + |E|)$. Here, $time$ is a global variable accesible in DFS-VISIT (G, u).

```
     DFS (G)
1    for each vertex u ∈ G. V
2        u. color = WHITE
3        u. π = NIL
4    time = 0
5    for each vertex u ∈ G. V
6        if u. color == WHITE
7            DFS-VISIT (G, u)
```

```
     DFS-VISIT (G, u)
1    time = time + 1
2    u. d = time
3    u. color = GRAY
4    for each v ∈ G. Adj [u]
5        if v. color == WHITE
6            v. π = u
7            DFS-VISIT (G, v)
8    v. color = BLACK
9    time = time + 1
10   u. f = time
```

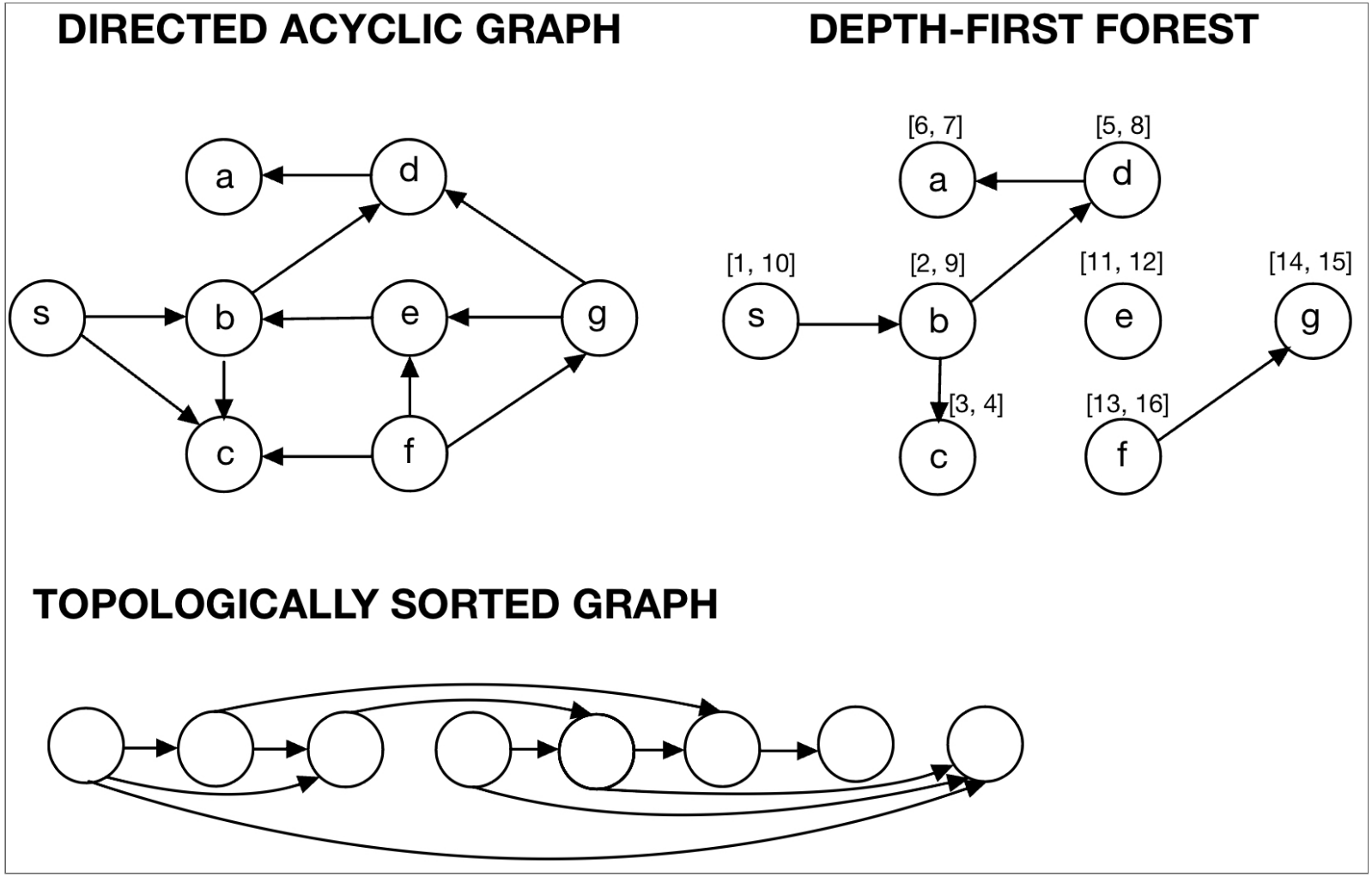
Applications of Depth-First Search

Topological sort

A **topological sort** of a directed acycled graph (DAG), $G = (V, E)$, is a linear ordering of all its vertices such that if G contains an edge, (u, v), then u appears before v in the ordering. If the graph contains a cycle, no such ordering is possible.

	TOPOLOGICAL-SORT (G)
1	call DFS (G) to compute finishing times $v.\ f$ for each vertex v
2	as each vertex is finished, insert it onto the front of a linked list
3	**return** the linked list of vertices

Run time: $O(|V| + |E|)$.


DIRECTED ACYCLIC GRAPH
DEPTH-FIRST FOREST
a
d
s
b
e
g
c
f
[6, 7]
[5, 8]
[1, 10]
[2, 9]
[11, 12]
[14, 15]
[3, 4]
[13, 16]
TOPOLOGICALLY SORTED GRAPH

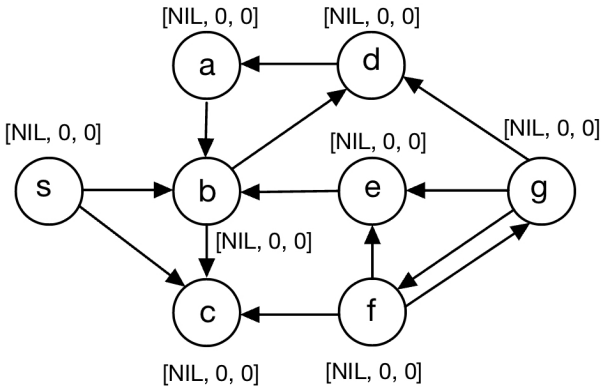
Strongly connected components

A **strongly connected component** of a directed graph, $G = (V, E)$, is a maximal set of vertices, $C \subseteq V$, such that for every pair of vertices, u and v in C, we have both $u \rightsquigarrow v$ and $v \rightsquigarrow u$.

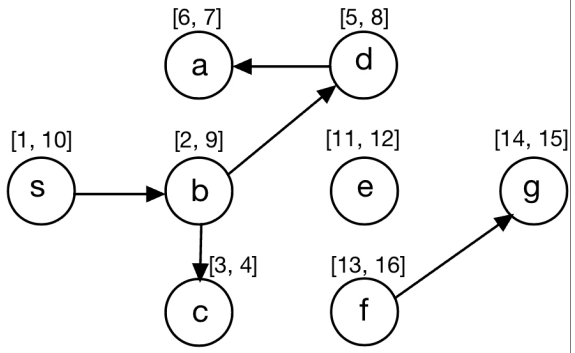
	STRONGLY-CONNECTED-COMPONENTS (G)
1	call DFS (G) to compute finishing times $u.\ f$ for each vertex u
2	compute G^T
3	call DFS (G^T), but in the main loop of DFS consider the vertices in order of decreasing u.f (as computed in line 1)
4	ouput the vertices of each tree in the depth-first forest formed in line 3 as a separate strongly connected component

Runtime: $\Theta(|V| + |E|)$.

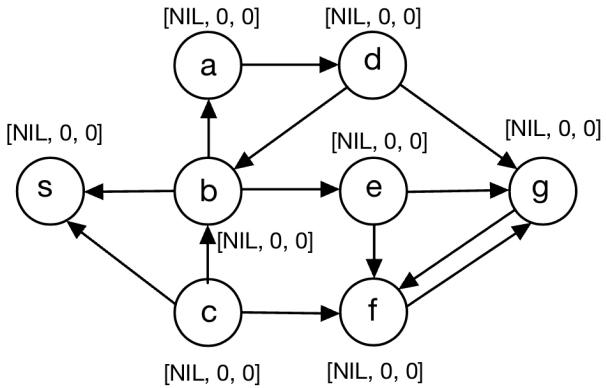
DEPTH-FIRST SEARCH, GRAPH - G



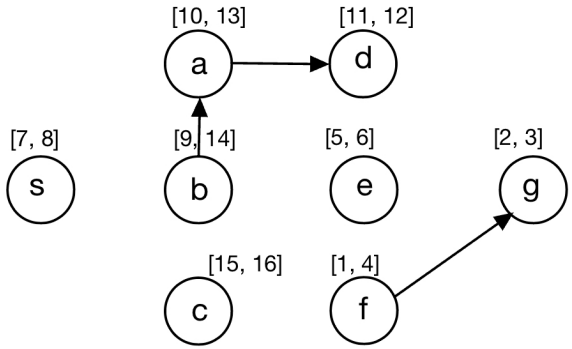
DEPTH-FIRST FOREST, GRAPH - G



DEPTH-FIRST SEARCH, GRAPH - G^T



DEPTH-FIRST FOREST, GRAPH - G^T



STRONGLY CONNECTED COMPONENTS

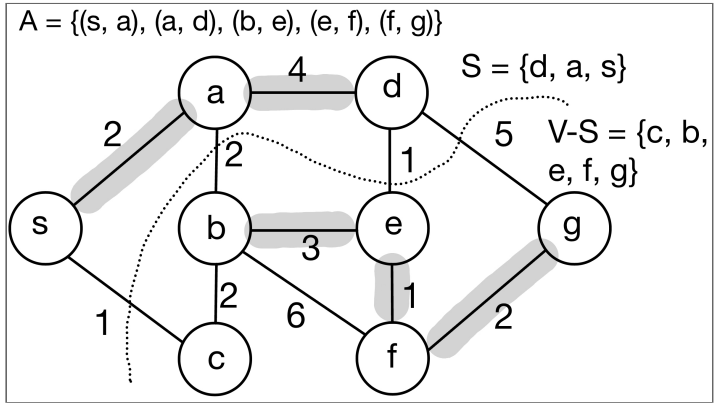
Minimum Weight Spanning Trees

A **minimum weight spanning tree** of undirected graph, $G = (V, E)$, is an acyclic subset of edges, $T \subseteq E$, that connects all vertices and whose total weight, $w = \sum_{(u,v)\in T} w(u, v)$, is minimised. MST's can be constructed by using greedy strategy maintaining the following invariant: *prior each iteration, A is a subset of some minimum spanning tree*. At each step a (**safe**) edge which respects such invariant is added.

	GENERIC-MST (G, w)
1	$A = \varnothing$
2	**while** A does not form spanning tree
3	find and edge (u, v) that is safe for A
4	$A = A \cup \{(u, v)\}$
5	**return** A

Properties:

- A is acyclic and $G_A = (V, A)$ is a forest, each connected component of G_A is a tree.
- **Safe** edge, (u, v), for A connects distinct components of G_A, since $A \cup \{(u, v)\}$ must be acyclic.



Definitions

- A **cut**, $(S, V - S)$, of an undirected graph, $G(V, E)$, is a partition of vertices, V.
- Edge, $(u, v) \in E$, **crosses** the cut, $(S, V - S)$, if one of its endpoints is in S and the other is in $V - S$.
- A cut **respects** a set of edges, A, if no edge in A crosses the cut.
- An edge is a **light edge** crossing a cut if its weight is the minimum of any edge crossing the cut.

Let A be a subset of edges, E, that is included in some minimum spanning tree for G and let $(S, V - S)$ be any **cut** of G that **respects** A. Then any **light edge**, (u, v), **crossing** $(S, V - S)$ is **safe** for A.

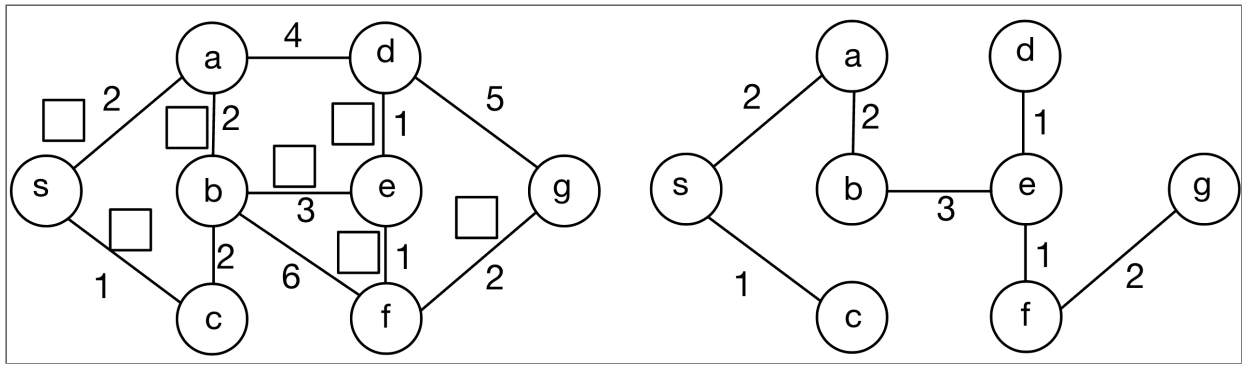
Kruskal's Algorithm

Pseudocode

Kruskal's algorithm iteratively finds a safe edge to add to the growing forest by finding of all edges that connect any two trees in the forest and choosing an edge, (u, v), of least weight.

	MST-KRUSKAL (G, w)
1	$A = \varnothing$
2	**for** each vertex $v \in G.V$
3	MAKE-SET (v)
4	sort the edges of $G.E$ into non-decreasing order by weight, w
5	**for** each edge $(u, v) \in G.E$ taken in non-decreasing order by weight
6	**if** FIND-SET $(u) \neq$ FIND-SET (v)
7	$A = A \cup \{(u, v)\}$
8	UNION (u, v)
9	**return** A

Example and runtime analysis



Runtime: $O(|E| \lg |V|)$, assuming an $O(n \lg n)$ implementation of sequence of n FIND-SET and UNION operations.

MAKE-SET (v) creates **disjoint-set** data structure is used to maintain several disjoint sets of elements (i.e. trees). At first each node, v, is its own set.

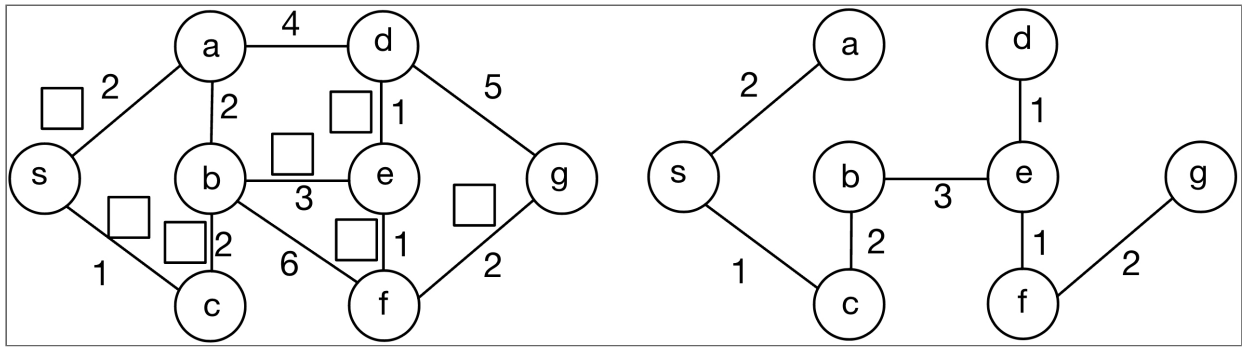
The operation, FIND-SET (u), returns a representative element from a set that contains u.

UNION (u, v) operation combines two disjoint sets for which u and v are elements of.

Prim's Algorithm

Prim's algorithm starts from an arbitrary root, r, an has a property that the edges in the set, A, always form a single tree.

- Each step adds to the tree, A, a light edge that connects A to an isolated vertex.
- For each vertex, v, attribute, $v.key$, stores the minimum weight of any edge connecting v to a vertex in the tree, A. Also $v.\pi$ is a parent of v in the tree, A.
- Q is a min-priority queue which is used to find the light edge based on the smallest attribute, $v.key$, in the queue.



Pseudocode

```
     MST-PRIM (G, w, r)
1    for each u ∈ G.V
2        u.key = ∞
3        u.π = NIL
4    r.key = 0
5    Q = G.V
6    while Q ≠ ∅
7        u = EXTRACT-MIN (Q)
8        for each v ∈ G.Adj[u]
9            if v ∈ Q and w(u, v) < v.key
10               v.π = u
11               v.key = w(u, v)
```

Runtime: $O(|E| \lg |V|)$ if **binary min-heap** is used. If **Fibonacci heap** is used then the runtime of Prim's algorithm is $O(|E| + |V| \lg |V|)$.

Shortest Path Problem

Given a weighted, directed graph, $G = (V, E)$, with weight function, $w : E \to R$, mapping edges to real-valued weights. The **weight**, $w(p)$, of path, $p = \langle v_0, v_1, \cdots v_k \rangle$, is the sum of the weights of its constituent edges: $w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$.

We define the **shortest-path weight**, $\delta(u, v)$ from u to v by

$$\delta(u, v) = \begin{cases} min\{w(p) \text{ s.t. } u \overset{p}{\rightsquigarrow} v\} & \text{if there is a path from } u \text{ to } v, \\ \infty, & \text{otherwise.} \end{cases}$$

A **shortest path** from vertex, u, to vertex, v, is then defined as any path, p, with weight $w(p) = \delta(u, v)$.

If the graph, $G = (V, E)$, contains no negative weight cycles reachable from source vertex, s, then for all $v \in V$, the shortest-path weight, $\delta(s, v)$, remains well defined, even if it has negative value. If the graph contains a negative-weight cycle reachable from s, shortest-path weights are not well defined.

Optimal substructure property

Let $p = \langle v_0, v_1, \cdots v_k \rangle$ be a shortest path from vertex, v_0, to vertex, v_k, and, for any i and j such that $0 \leq i \leq j \leq k$, let $p_{ij} = \langle v_i, v_{i+1}, \cdots v_j \rangle$ be the subpath of p from vertex, v_i, to vertex, v_j. Then p_{ij} is a shortest path from v_i to v_j.

Many shortest path algorithms can be formalised as repeated **relaxation** (testing whether the shortest path can be improved) of edges. Here $v.d$ is the length and $v.\pi$ is the parent of the **shortest path estimate** at any time.

Shortest paths are represented as **shortest path tree** rooted as source vertex, s.

	INITIALIZE-SINGLE-SOURCE (G, s)
1	**for** each vertex $v \in G.V$
2	$v.d = \infty$
3	$v.\pi = \text{NIL}$
4	$s.d = 0$

	RELAX (u, v, w)
1	**if** $v.d > u.d + w(u, v)$
2	$v.d = u.d + w(u, v)$
3	$v.\pi = u$

Properties of shortest path relaxation

- **Triangle inequality.** For any edge $(u, v) \in E$, we have: $\delta(s, v) \leq \delta(s, u) + w(u, v)$.

- **Upper-bound property.** We always have $v.d \geq \delta(s, v)$ for all vertices, $v \in V$, and once $v.d$ achieves the value, $\delta(s, v)$, it never changes.

- **No path property.** If there is no path from s to v, then $v.d = \delta(s, v) = \infty$.

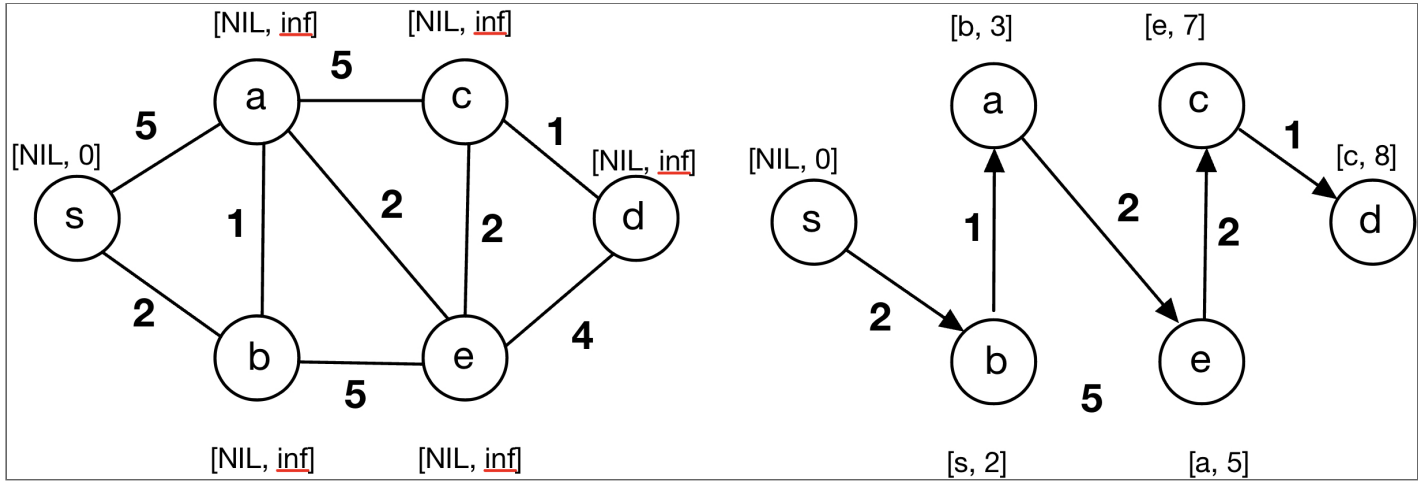
- **Path-relaxation property.** If $p = \langle v_0, v_1, \cdots, v_k \rangle$ is a shortest path from $s = v_0$ to v_k, and we relax the edges of p in the order $(v_0, v_1), (v_1, v_2), \cdots, (v_{k-1}, v_k)$, then $v_k.d = \delta(s, v_k)$. This property holds regardless any other relaxation steps that occur, even if they are intermixed with relaxations of the edges of p.

- **Predecessor-subgraph property.** Once $v.d = \delta(s, v)$ for all $v \in V$, the predecessor subgraph is a shortest-path tree rooted at s.

Shortest Path Algorithms

Bellman-Ford

The **Bellman-Ford** algorithm relaxes edges, progressively decreasing an estimate, $v.d$, on the weight of a shortest path from the source, s, to each vertex, $v \in V$, until it achieves the actual shortest-path weight, $\delta(s, v)$. The algorithm returns TRUE if and only if the graph contains no negative-weight cycles that are reachable from the source.



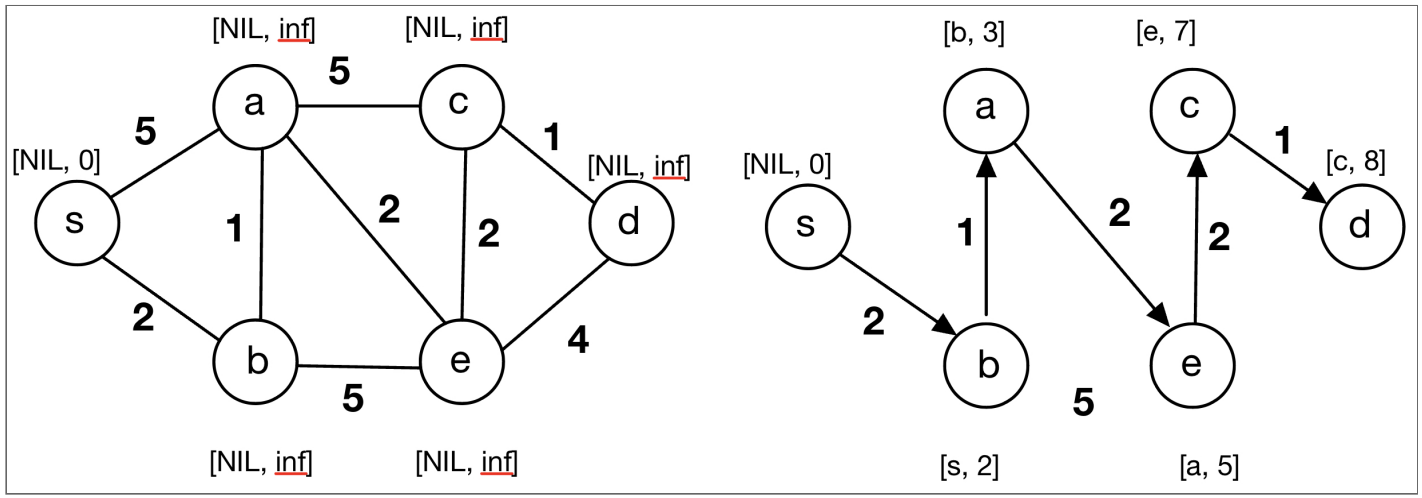
Runtime: $\Theta(VE)$.

```
BELLMAN-FORD (G, w, s)
1  INITIALIZE-SINGLE-SOURCE (G, s)
2  for i = 1 to |G.V| − 1
3    for each edge (u, v) ∈ G.E
4      RELAX (u, v, w)
5  for each edge (u, v) ∈ G.E
6    if v.d > u.d + w(u, v)
7      return FALSE
8  return TRUE
```

Shortest Path Algorithms

Dijkstra's algorithm

Maintains a set, S, of vertices whose final shortest-path weights from the source, s, have already been determined. The algorithm repeatedly selects the vertex, $u \in V - S$, with the minimum shortest-path length estimate, adds u to S, and relaxes all edges leaving u. Min-priority queue Q keyed by $v.d$ estimates of shortest path length is used. *Note, only non-negative weights are permitted in Dijkstra's algorithm*.



Runtime: $O(|E| \lg |V|)$ is binary min-heap is used. $O(|V| \lg |V| + |E|)$ when min-priority queue is implemented via Fibonacci heap.

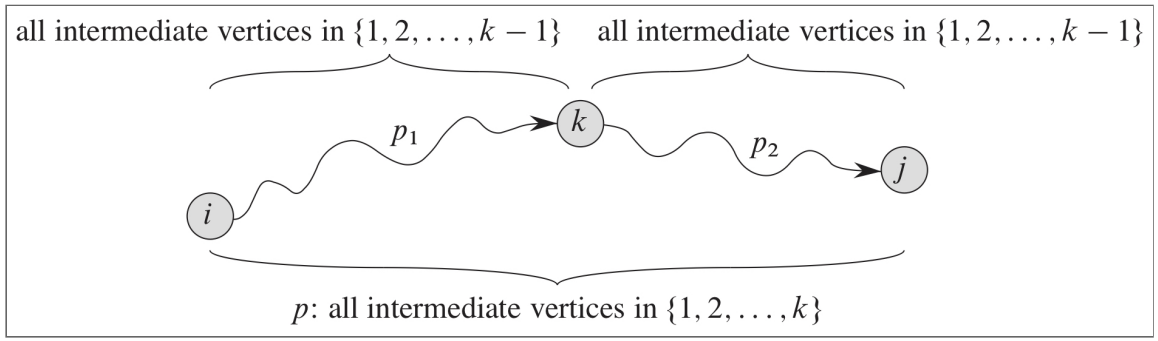
```
    DIJKSTRA (G, w, s)
1   INITIALIZE-SINGLE-SOURCE (G, s)
2   S = ∅
3   Q = G.V
4   while Q ≠ ∅
5       u = EXTRACT-MIN (Q)
6       S = S ∪ {u}
7       for each vertex v ∈ G.Adj [u]
8           RELAX (u, v, w)
```

Shortest Path Algorithms

Floyd-Warshall algorithm

This algortihm solves **all-pairs** shortest path using dynamic programming paradigm.

In particular it relies on the following property that a shortest path, $d_{i,j}^{k}$, from i to j made only of intermediate vertices $\{1 \cdots k\}$ is the minimum of path some $i \rightsquigarrow k \rightsquigarrow j$ and a shortest path, $d_{ij}^{(k-1)}$, from i to j which does not go through k and only uses intermediate vertices in $\{1 \cdots k\}$.



In [9]:

```
# Runtime: $O(V^3)$.

MV = 10000

#      s,  a,  b,  c,  d,  e
W = [[ 0,  5,  2, MV, MV, MV], #s
     [ 5,  0,  1,  5, MV,  2], #a
     [ 2,  1,  0, MV, MV,  5], #b
     [MV,  5, MV,  0,  1,  2], #c
     [MV, MV, MV,  1,  0,  4], #d
     [MV,  2,  5,  2,  4,  0], #e
    ]

def Floyd_Warshall(W):
    D = W.copy()
    for k in range(len(D)):
        D_new = D.copy()
        for i in range(len(D)):
            for j in range(len(D)):
                D_new[i][j] = \
                    min(D[i][j],D[i][k]+D[k][j])
        D = D_new
    return D
u, v = 0, 4
print('Shortest path from ',u,' to ',v,' is:')
print(Floyd_Warshall(W)[u][v])
```

```
Shortest path from  0  to  4  is:
8
```

Flow Networks and Flows

A **flow network**, $G = (V, E)$, is a directed graph in which each edge, $(u, v) \in E$, has non-negative **capacity**, $c(u, v) \geq 0$. Also if $(u, v) \in E$ then $(v, u) \notin E$ (i.e. $c(v, u) = 0$) and $(u, u) \notin E$.

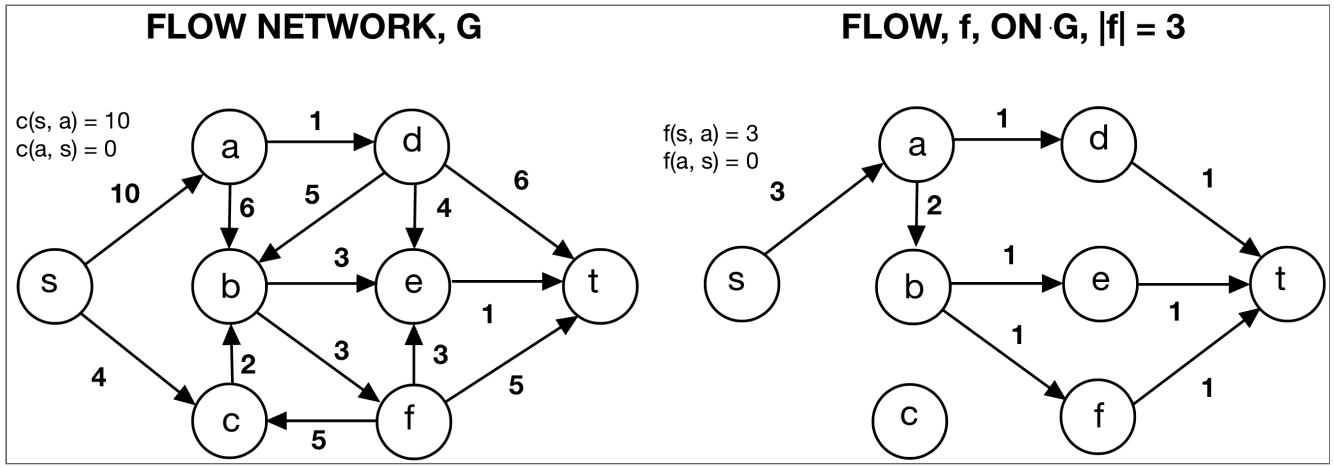
A flow network has two additional distinguished vertices **source**, s, and **sink**, t. For convenience we can assume that all vertices, v, are on some path, $s \rightsquigarrow v \rightsquigarrow t$.

A **flow**, f, in flow network, G, is a real valued function, $f : V \times V \to R$, that satisfies two properties:

- **capacity constraint**: $0 \leq f(u, v) \leq c(u, v)$ for all $u, v \in V$.
- **flow conservation**: $\sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v)$ for all $u \in V - \{s, t\}$. Note when $(u, v) \notin E$ we have $f(u, v) = 0$.

The **value**, $|f|$, of the flow, f, is defined as $|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s)$. Note, typically there are no incomming edges into the source vertex, s, and $|f| \geq 0$.

Example



In the **maximum-flow problem** we are given a flow network, G, with the source, s, and sink, t, and we wish to find a flow of maximum value.

Note:

- **Antiparallel** edges can be handled by inserting additional vertex and replacing one of the edge into two going through the inserted edge.
- The maximum-flow problem may have multiple sources and sinks. You can reduce it by adding a **super-source** and **super-sink** with infinite capacities.

Maximum-Flow Algorithms

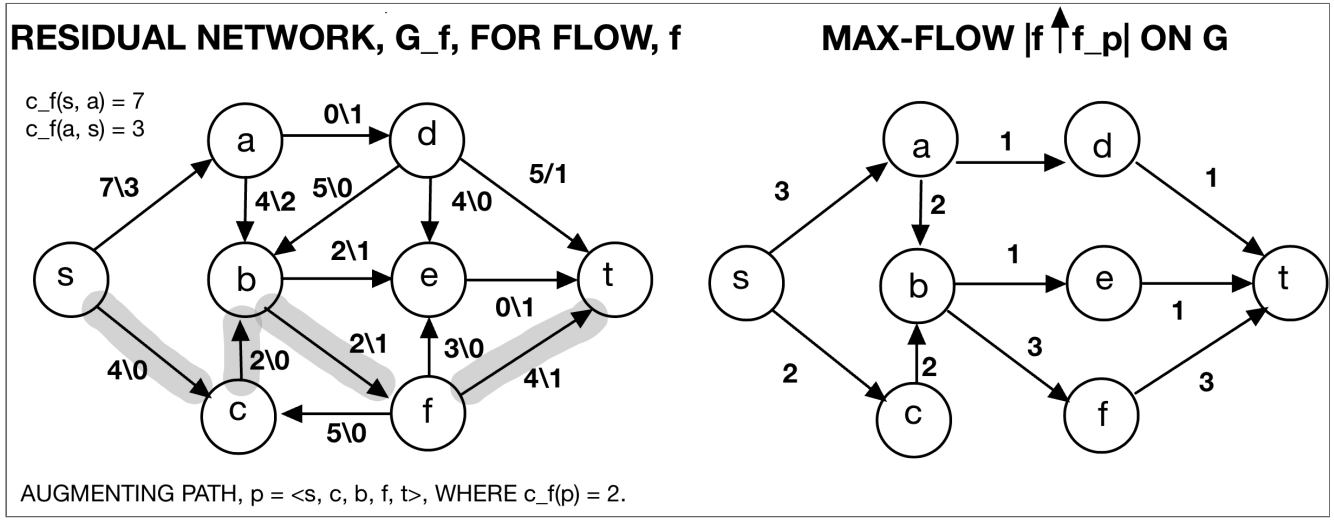
Ford-Fulkerson method

An edge of the flow network can admit an amount of additional flow equal to the edge's capacity minus the flow on that edge. I.e. **residual capacity**, $c_f(u,v)$, (of edge (u,v)) is defined by: $c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & \text{if } (u,v) \in E, \\ f(v,u) & \text{if } (v,u) \in E, \\ 0 & \text{otherwise.} \end{cases}$

Given a flow network, $G = (V,E)$, and a flow, f, **residual network**, $G_f = (V, E_f)$, consists of edges with capacities that represent how we can change the flow on edges of G, where $E_f = \{(u,v) \in V \times V \text{ s.t. } c_f(u,v) > 0\}$ is a set of **residual edges**.

If f is a flow in G and f' is a flow in the corresponding residual network, G_f, we define $f \uparrow f'$, the **augmentation** of flow, f, by f', to be a function:
$(f \uparrow f')(u,v) = \begin{cases} f(u,v) + f'(u,v) - f'(v,u) & \text{if } (u,v) \in E, \\ 0, & \text{otherwise.} \end{cases}$

It is easy to show that $|f \uparrow f'| = |f| + |f'|$.



An **augmenting path**, p, is a *simple path* from s to t in the *residual network*, G_f.

The maximum amount by which we can increase the flow on each edge in an augmenting path, p, is called the **residual capacity** of p and given by $c_f(p) = \min\{c_f(u, v) \text{ s.t. } (u, v) \text{ is on } p\}$.

A flow in G_f which reaches the residual capacity of path, p, (ie. $|f_p| = c_f(p) > 0$) can be constructed as: $f_p(u, v) = \begin{cases} c_f(p) & \text{if } (u, v) \text{ is on } p, \\ 0 & \text{otherwise.} \end{cases}$. Also $|f \uparrow f_p| = |f| + |f_p| > |f|$.

	FORD-FULKERSON-METHOD (G, s, t)
1	initialize flow f to 0
2	**while** exists an augmenting path p in the residual network G_f
3	augment flow f along p
4	**return** f

Proof by **max-flow min-cut theorem**:

If f is a flow in flow network, $G = (V, E)$, with source, s, and sink, t, then the following conditions are equivalent:

- f is a maximum flow of G.
- The residual network, G_f, contains no augmenting paths.
- $|f| = c(S, T)$ for some cut, (S, T), of G, where $c(S, T) = \sum_{u \in S} \sum_{v \in T} c(u, v)$ is the **capacity** of the cut (S, T). Here, $s \in S$ and $t \in T$.

A **minimum cut** of a network is a cut whose capacity is minimum over all cuts of the network.

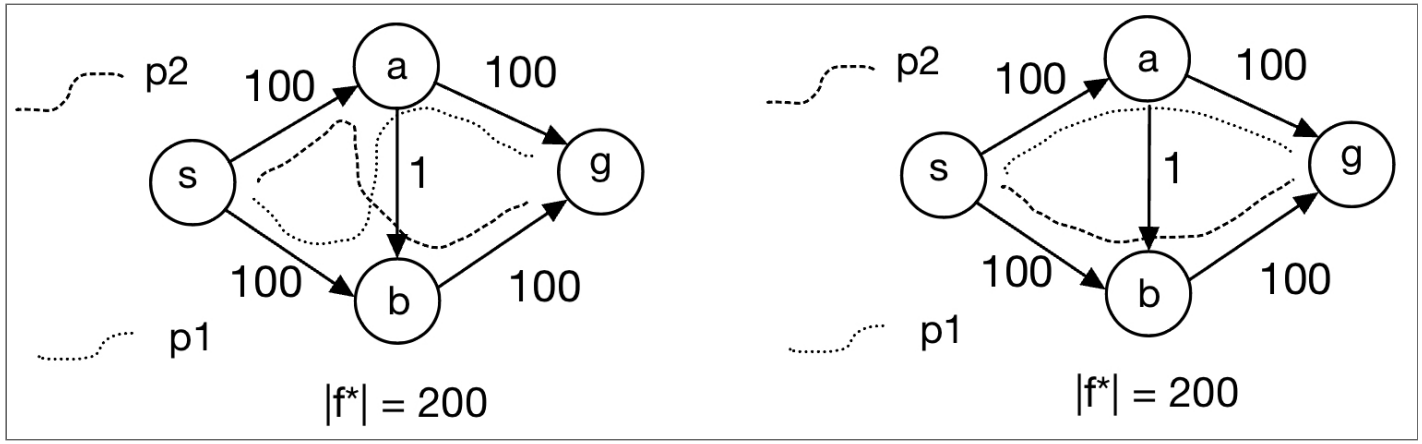
Ford-Fulkerson algorithm

Runtime: $O(|E||f^*|)$, where $|f^*|$ is the maximum flow (assuming integer flows).

	FORD-FULKERSON (G, s, t)
1	**for** each edge $(u, v) \in G.E$
2	$(u, v).f = 0$
3	**while** exists a path p from s to t in the residual network G_f
4	$c_f(p) = \min\{c_f(u, v) \text{ s.t. } (u, v) \text{ is in } p\}$
5	**for** each edge (u, v) in p
6	**if** $(u, v) \in E$
7	$(u, v).f = (u, v).f + c_f(p)$
8	**else** $(v, u).f = (v, u).f - c_f(p)$

Maximum-Flow Algorithms

Edmonds-Karp algorithm

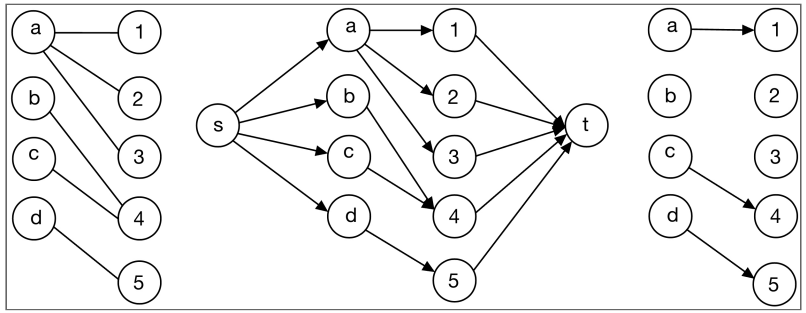


Choose the augmenting path as a shortest path from s to t in the residual network where each edge has unit distance (weight).

Runtime: $O(|V||E|^2)$.

Applications of Max-Flow Algorithm

Maximum bipartite matching



Let $G = (V, E)$ be a bipartite graph with vertex partition, $L \cup R$, and let $G' = (V', E')$ be its corresponding flow network. If M is a matching in G, then there is an integer-valued flow, f, in G' with value, $|f| = |M|$. Conversely, if f is an integer-valued flow in G' then there is a matching, M, in G with cardinality $|M| = |f|$.

The cardinality of a maximum matching, M, in bipartite graph G equals the value of a maximum flow, f ,in its corresponding flow network, G'.