EE/CE 6301: Advanced Digital Logic

Bill Swartz

Dept. of EE
Univ. of Texas at Dallas

Graphs and Algorithms

Graph Theory - Background

Importance

- Many optimization problems employ graph representations to model the core part.

- Graph theory has a history of 300+ years and there are many mature graph-based algorithms exist.

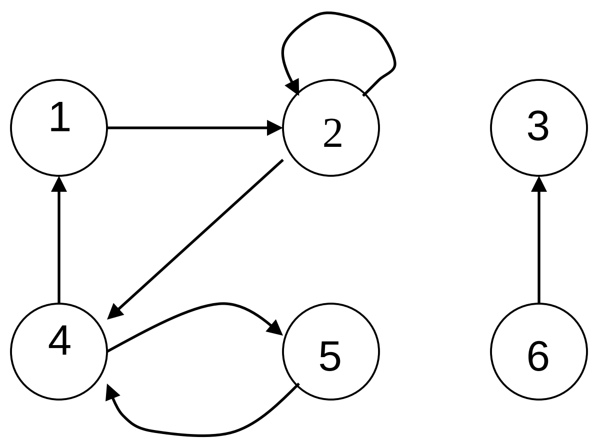
- Example of graph-based algorithms
  - Path traversal
  - Partitioning to sub-graphs
  - Coloring
  - …

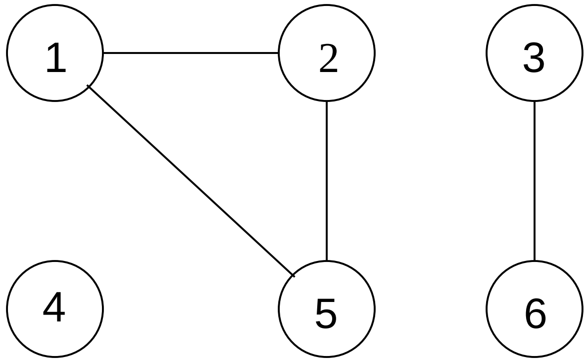
What Can Graphs Model?

- Cost of wiring electronic components together.
- Shortest route between two cities.
- Finding the shortest distance between all pairs of cities in a road atlas.
- Flow of material (liquid flowing through pipes, current through electrical networks, information through communication networks, parts through an assembly line, etc).
- State of a machine (FSM).
- Used in Operating systems to model resource handling (deadlock problems).
- Used in compilers for parsing and optimizing the code.

What is a Graph?

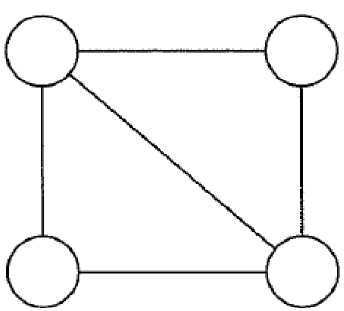
- Informally a *graph* is a set of nodes joined by a set of lines or arrows.



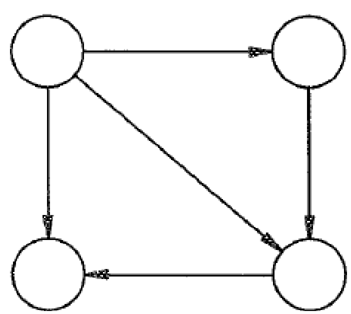


Definition

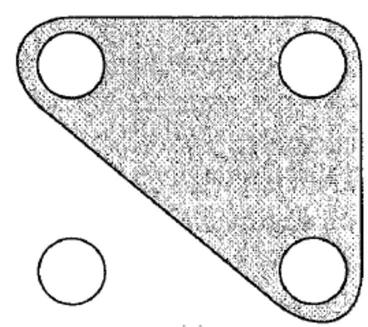
- A graph G(V,E) is a pair (V,E), where V is a set of vertices and E is a set of edges.
  - —Directed graph (digraph): the edges are ordered pairs of vertices, e.g. (v_i , v_j)
  - —Undirected graph: the edges are unordered pairs, e.g. $\{v_i , v_j\}$
  - —The degree of a vertex is the number of edges incident to it.
  - —A hypergraph is an extension of a graph where edges may be incident to any number of vertices



Undirected Graph



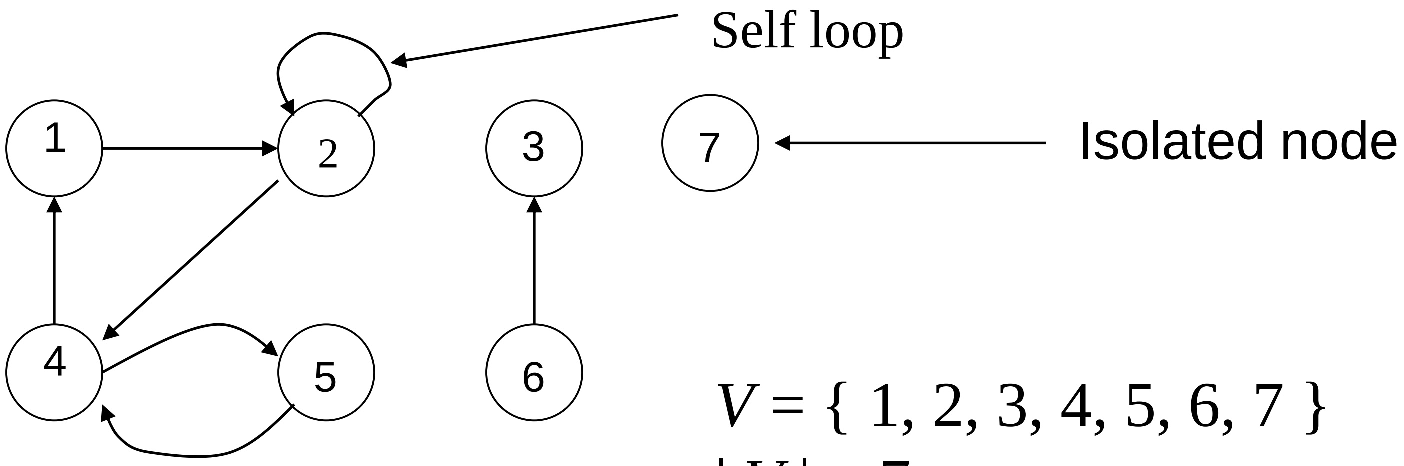
Directed Graph



Hypergraph

Directed Graph

- A ***directed graph,*** also called a ***digraph*** **G** is a pair (*V*, *E*), where the set *V* is a finite set and *E* is a binary relation on *V*.
- The set ***V*** is called the **vertex set** of *G* and the elements are called vertices. The set *E* is called the **edge set** of *G* and the elements are *edges* (also called *arcs*). A edge from node *a* to node *b* is denoted by the ordered pair (*a*, *b*).



$V = \{ 1, 2, 3, 4, 5, 6, 7 \}$

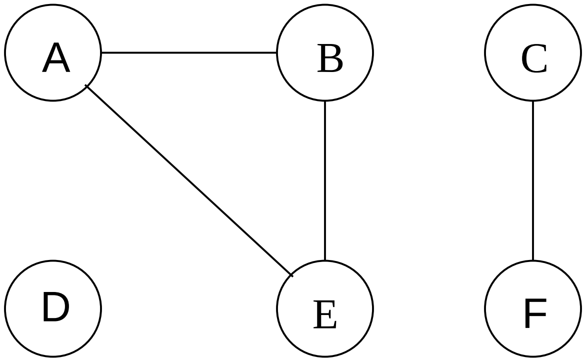
$|V| = 7$

$E = \{ (1,2), (2,2), (2,4), (4,5), (4,1), (5,4),(6,3) \}$

$|E| = 7$

Undirected Graph

- An ***undirected graph*** $G = (V, E)$, but unlike a digraph the edge set E consist of unordered pairs. We use the notation (a, b) to refer to a directed edge, and $\{a, b\}$ for an undirected edge.



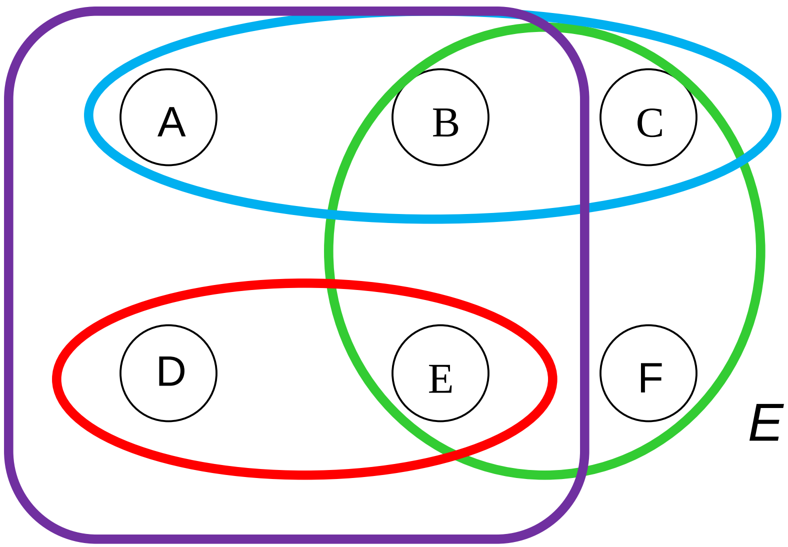
$V = \{$ A, B, C, D, E, F $\}$
$|V| = 6$

$E = \{$ {A, B}, {A,E}, {B,E}, {C,F} $\}$
$|E| = 4$

Some texts use (a, b) also for undirected edges.
So (a, b) and (b, a) refers to the same edge.

Hyper Graph

- A ***hyper graph*** $H = (V, E)$, is the set of vertices V and E is the set of edges which forms sets between the vertices or nodes. Any edge may contain any number of nodes



$V = \{ A, B, C, D, E, F \}$

$|V| = 6$

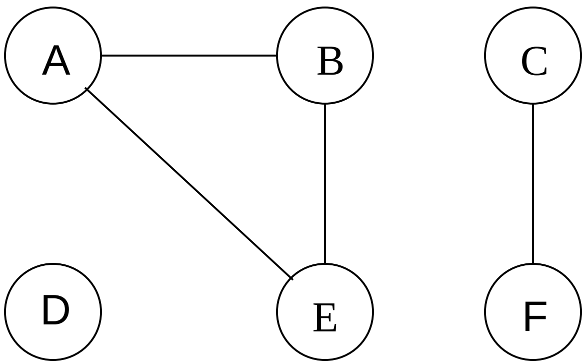
$E = \{ \{B,C,E,F\}, \{A,B,C\}, \{D,E\}, \{A,B,D,E\} \}$

$|E| = 4$

- Natural representation of a circuit description or netlist

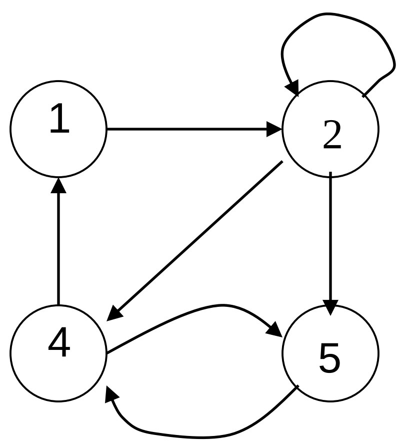
Degree of a Vertex

- ***Degree*** of a Vertex in an undirected graph is the number of edges incident on it. In a directed graph , the ***out degree*** of a vertex is the number of edges leaving it and the ***in degree*** is the number of edges entering it.



The degree of B is 2.

Self-loop



The in degree of 2 is 2 and the out degree of 2 is 3.

Simple Graphs

- ***Simple graphs*** are graphs without multiple edges or self-loops. We will consider only simple graphs.

Proposition: If G is an undirected graph then

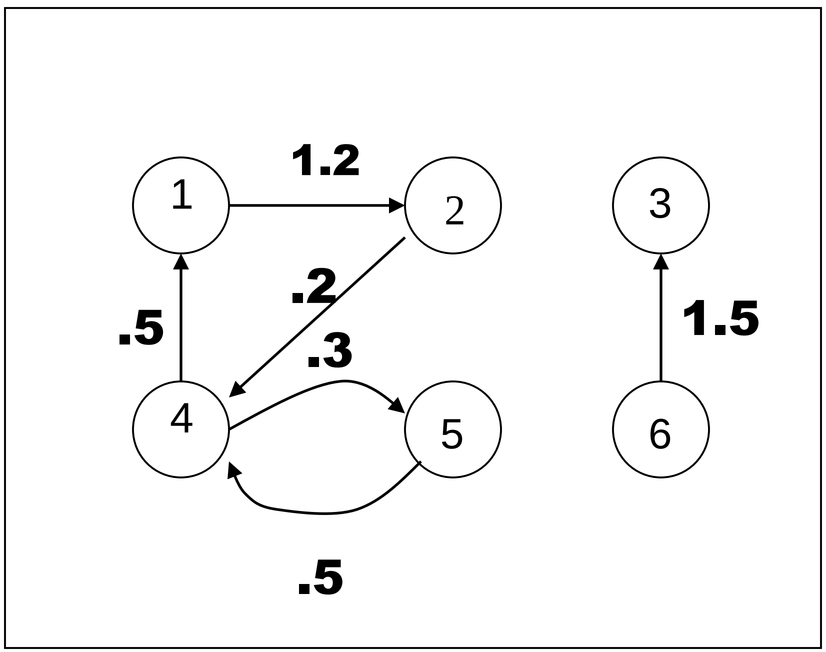
$$\sum_{v \in G} \deg(v) = 2\,|E|$$

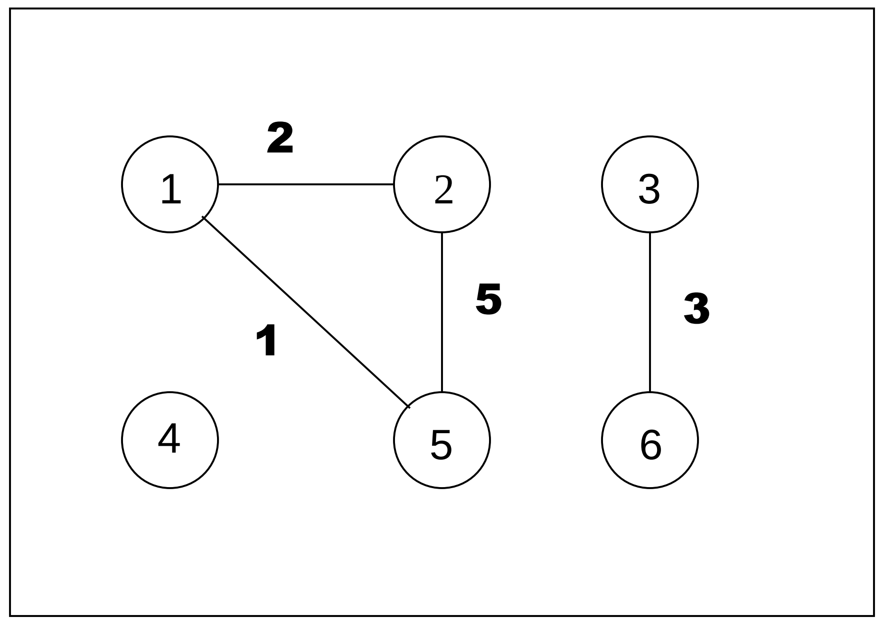
Proposition: If G is a digraph then

$$\sum_{v \in G} \mathrm{indeg}(v) = \sum_{v \in G} \mathrm{outdeg}(v) = |E|$$

Weighted Graph

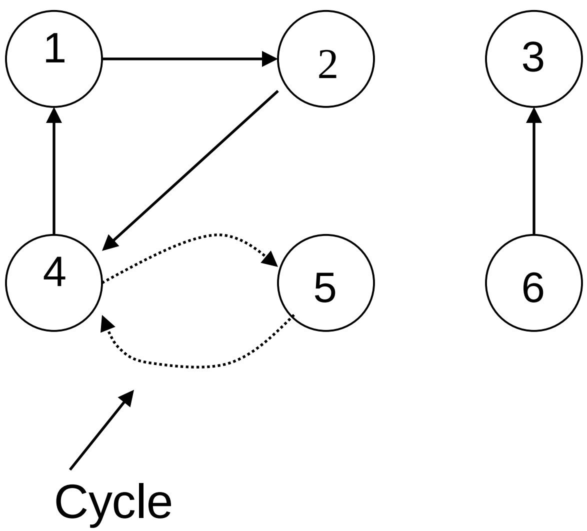
- A ***weighted graph*** is a graph for which each edge has an associated ***weight***, usually given by a ***weight function*** $w: E \rightarrow \mathbf{R}$.
- Directed or undirected graphs can be weighted. Weights can be associated with vertices and/or with edges, i.e. the graph can be vertex weighted and/or edge-weighted.

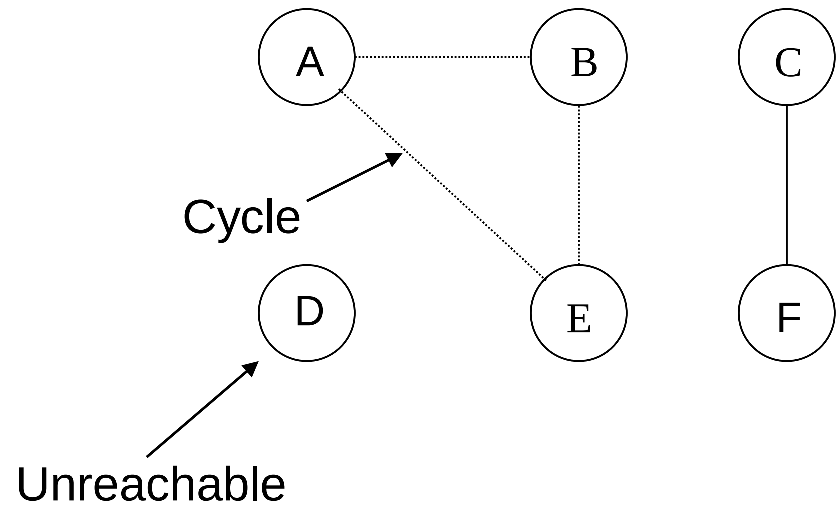




Cycles and Paths

- A ***path*** is a sequence of vertices such that there is an edge from each vertex to its successor. A path from a vertex to itself is called a ***cycle***. A graph is called ***cyclic*** if it contains a cycle; otherwise it is called ***acyclic*** A path is ***simple*** if each vertex is distinct.



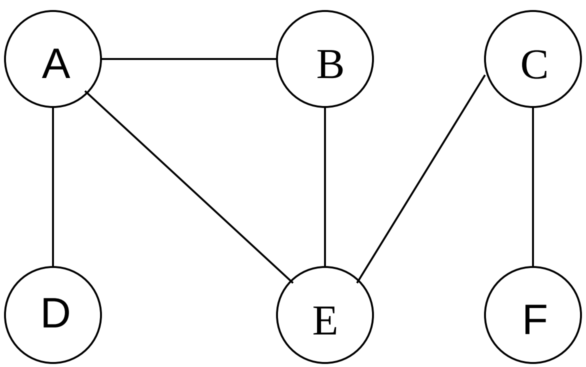


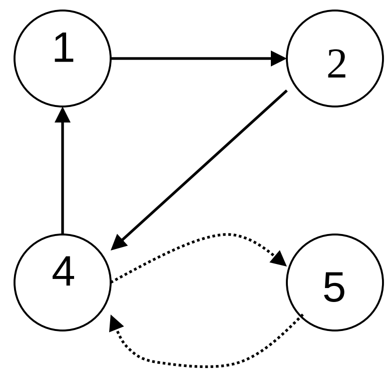
Simple path from 1 to 5
= (1, 2, 4, 5)
or as in our text
((1, 2), (2, 4), (4,5))

If there is path *p* from *u* to *v* then we say *v* is reachable from *u* via *p*.

Connectivity Features

- An undirected graph is ***connected*** if you can get from any node to any other by following a sequence of edges OR any two nodes are connected by a path.
- A directed graph is ***strongly connected*** if there is a directed path from any node to any other node.





- A graph is ***sparse*** if $|E| \approx |V|$
- A graph is ***dense*** if $|E| \approx |V|^2$.

Graph Traversal

- A vertex is adjacent to another vertex when there is an edge incident to both of them.
- An edge with two identical end-points is a loop.
- A graph is simple if it has no loops and no two edges link the same vertex pair. Otherwise, it is called a multi-graph.
- A walk is an alternating sequence of vertices and edges.
- A trail is a walk with distinct edges.
- A path is a trail with distinct vertices.
- A cycle is a closed walk (i.e. such that the two end-point vertices coincide) with distinct vertices.

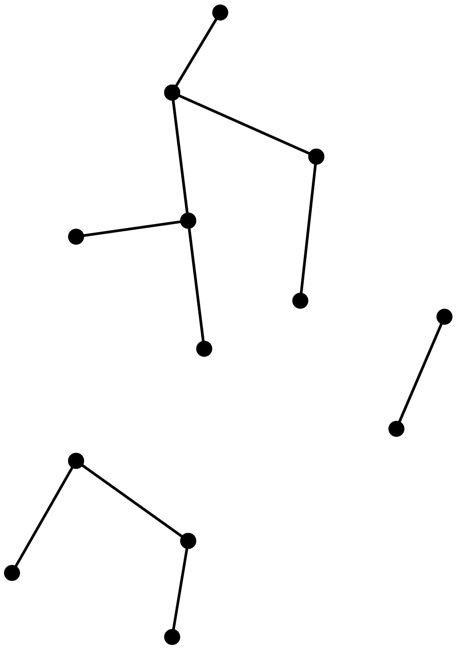
Trees and Forest

- A graph is connected if all vertex pairs are joined by a path.
- A graph with no cycles is called an acyclic graph or a forest.
- A tree is a connected acyclic graph.
- A rooted tree is a tree with a distinguished vertex, called a root.
- Vertices of a tree are also called nodes. In addition, they are called leaves when they are adjacent to only one vertex each and they are distinguished from the root.

Trees and Forest (cont.)

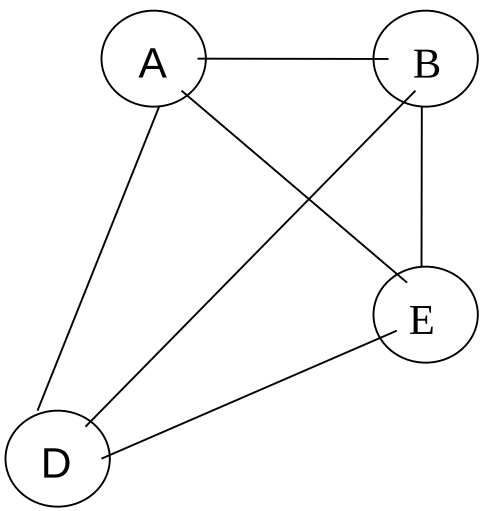
- Let $G = (V, E)$ be an undirected graph. The following statements are equivalent.

  1. G is a tree
  2. Any two vertices in G are connected by unique simple path.
  3. G is connected, but if any edge is removed from E, the resulting graph is disconnected.
  4. G is connected, and $|E| = |V| -1$
  5. G is acyclic, and $|E| = |V| -1$
  6. G is acyclic, but if any edge is added to E, the resulting graph contains a cycle.



Complete Graph

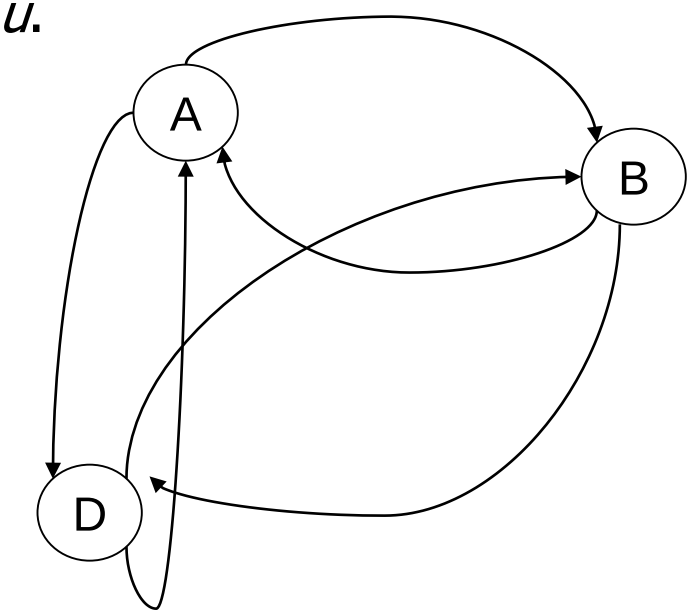
- A complete graph is one such that each vertex pair is joined by an edge. Alternatively, we can say:
- A ***Complete graph*** is an undirected/directed graph in which every pair of vertices is adjacent. If (u, v) is an edge in a graph G, we say that vertex v is ***adjacent*** to vertex u.



4 nodes and (4*3)/2 edges

V nodes and V*(V-1)/2 edges

Note: if self loops are allowed V(V-1)/2 +V edges



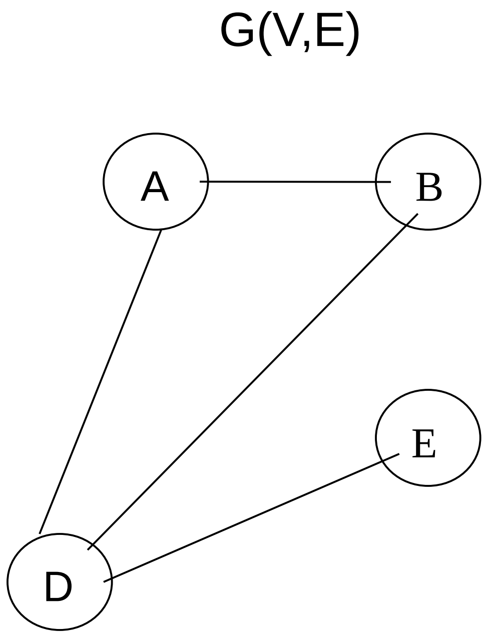
3 nodes and 3*2 edges

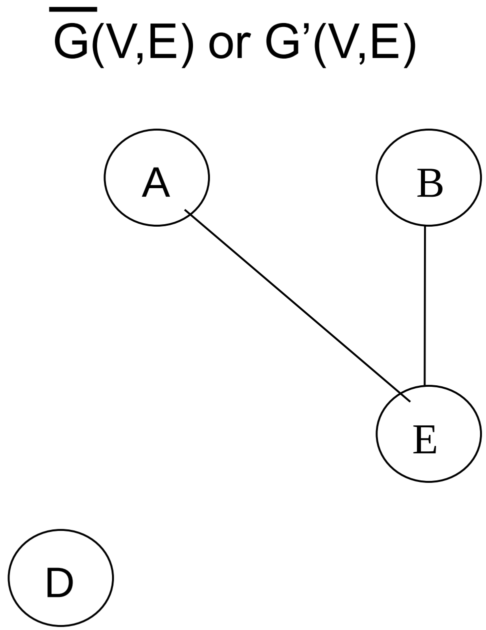
V nodes and V*(V-1) edges

Note: if self loops are allowed V^2 edges

Complement Graph

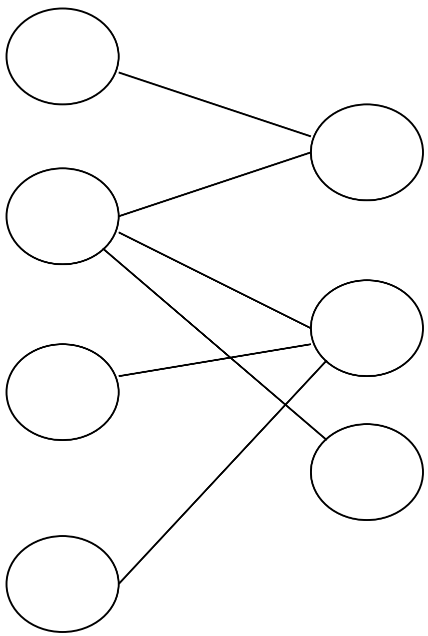
- The complement of a graph G(V,E) is a graph with vertex set V, two vertices being adjacent if and only if they are not adjacent in G(V,E).





Bipartite Graph

- A bipartite graph is a graph where the vertex set can be partitioned into two subsets such that each edge has end-points in different subsets.
- An alternative definition: A ***bipartite graph*** is an undirected graph $G = (V,E)$ in which V can be partitioned into two subsets V_1 and V_2 such that $(u,v) \in E$ implies either $u \in V_1$ and $v \in V_2$ OR $v \in V_1$ and $u \in V_2$.



Undirected Graphs - Other Definitions

- A subgraph of a graph G(V,E) is a graph whose vertex and edge sets are contained in the vertex and edge sets, respectively, of G(V,E).

- Given a graph G(V,E) and a vertex subset $U \subseteq V$ the subgraph induced by U is the maximal subgraph of G(V,E) whose edges have end-points in U.

- A clique of a graph is a complete subgraph; it is maximal when it is not contained in any other clique. (Note that some books refer to "maximal clique" as cliques.)

Undirected Graphs - Other Definitions (cont.)

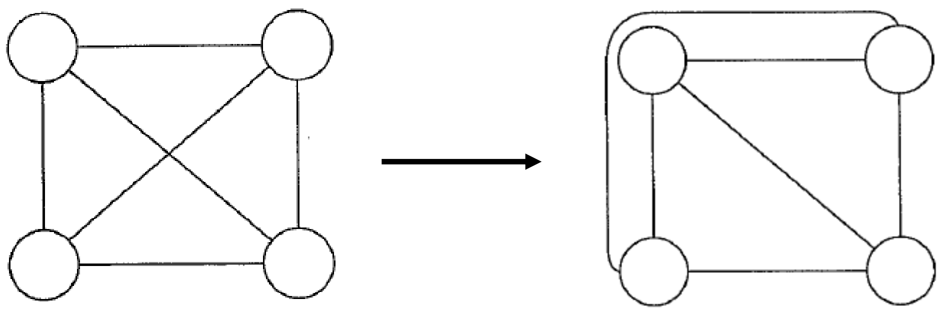
- Given an undirected graph, an orientation is a directed graph obtained by assigning a direction to the edges.

- A cutset is a minimal set of edges whose removal from the graph makes the graph disconnected.

- A vertex separation set is a minimal set of vertices whose removal from the graph makes the graph disconnected.

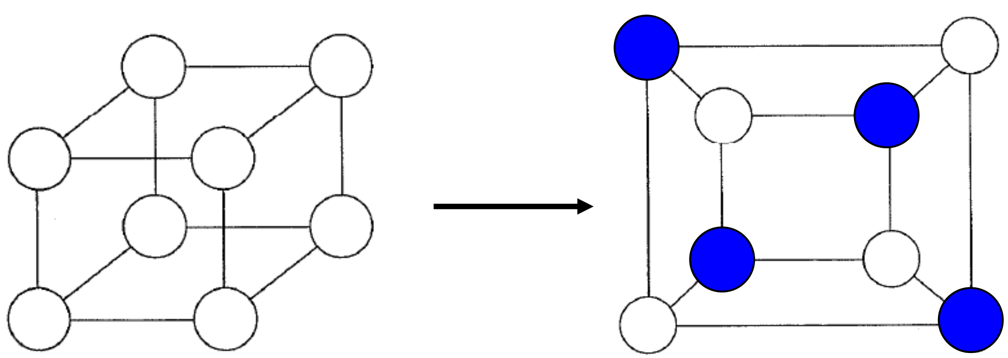
Undirected Graphs - Other Definitions (cont.)

- A graph is said to be planar if it has a diagram on a plane surface such that no two edges cross.



Planar and also complete

- Two graphs are said to be isomorphic if there is a one-to-one correspondence between their vertex sets that preserves adjacency.



Planar, isomorphic and also pipartite

Directed Graph - I

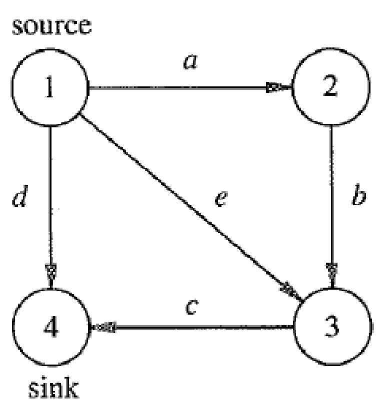
- For any directed edge (v_i, v_j), vertex v_i is called the tail and vertex v_j is called the head.
- The in-degree of a vertex is the number of edges where it is the head.
- The out-degree of a vertex is the number of edges where it is the tail.
- A walk is an alternating sequence of vertices and edges with the same direction. Trails, paths and cycles are defined similarly.

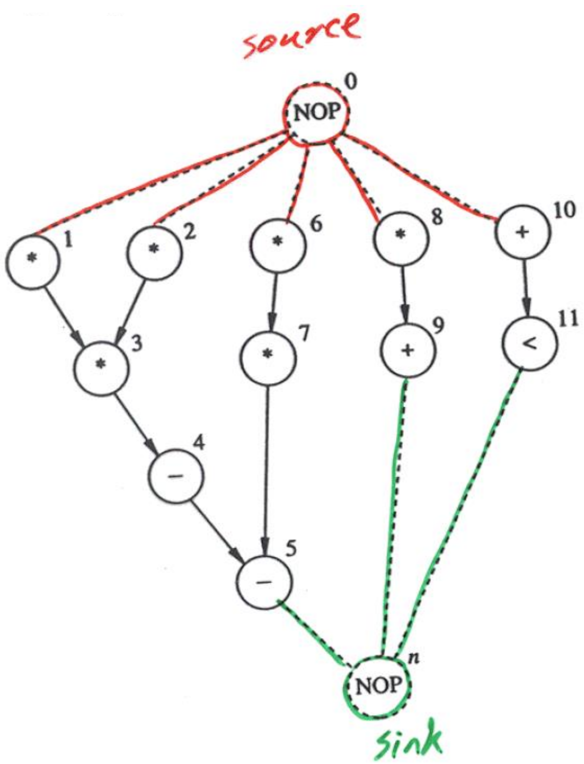
Directed Graph - II

- Directed acyclic graphs is also called dags.
- A vertex v_j is called the successor (or descendant) of a vertex v_i, if v_j is the head of a path whose tail is v_i. We also say that a vertex v_j is reachable from vertex v_i when v_j is a successor of v_i.
- A vertex v_i is called the predecessor (or ancestor) of a vertex v_j, if v_i is the tail of a path whose head is v_j. We also say that a vertex v_j is reachable from vertex v_i when v_j is a successor of v_i.
- Vertex v_j is a direct successor (child or adjacent to) of vertex v_i if v_j is the head of an edge whose tail is v_i. Direct predecessor is similarly defined.

Directed Graph - III

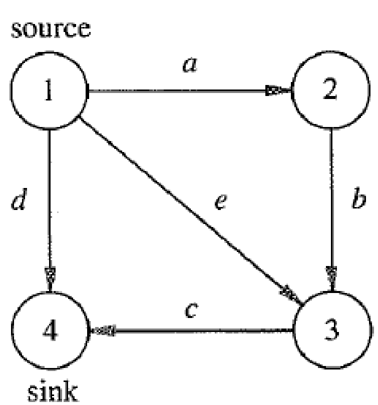
- A polar dag is a graph having two distinguished vertices, a source and a sink, and where all vertices are reachable from the source and where the sink is reachable from all vertices.





Graph Matrix Representation

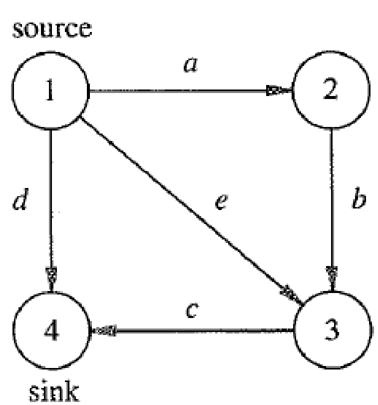
- The incidence matrix can be used to represent a simple graph
  - Number of rows: V
  - Number of columns: E
  - For undirected Graph: Entry (i,j) is 1 if the jth edge is incident to vertex v_i else it is 0.
  - For directed Graph: Entry (i,j) is 1 if vertex v_i is the head of the jth edge, -1 if it is its tail and otherwise 0.



$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{ccccc} a & b & c & d & e \end{array} \\ \begin{bmatrix} -1 & 0 & 0 & -1 & -1 \\ 1 & -1 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{bmatrix} \end{array}$$

Graph Matrix Representation (cont.)

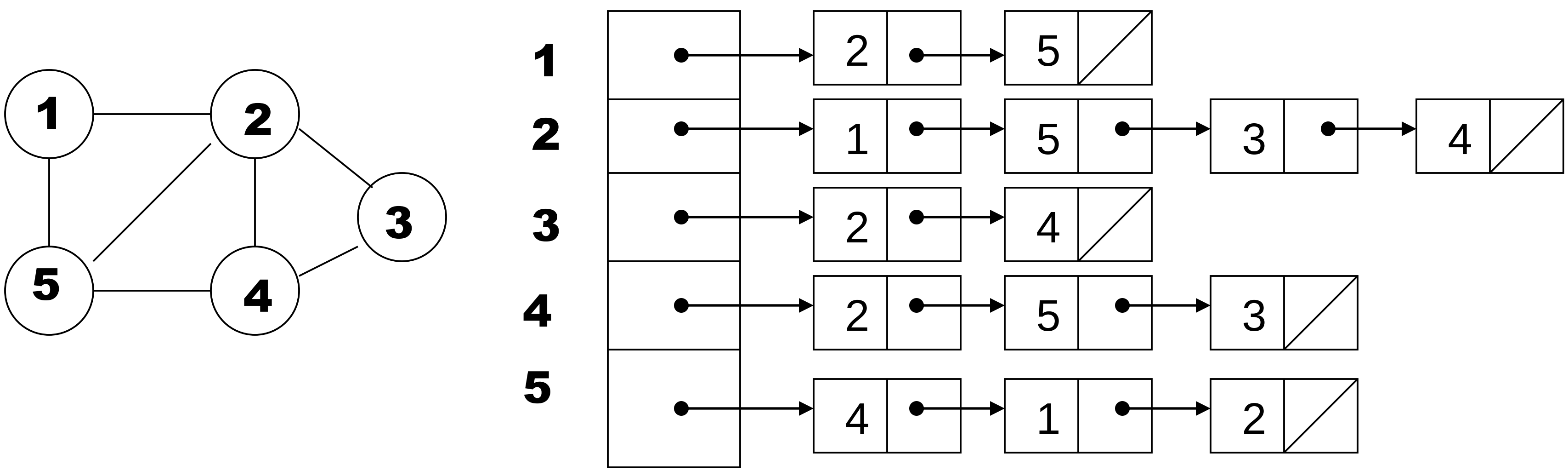
- The adjacency matrix can be used to represent a simple graph. The matrix is symmetric only for undirected graph.
  - Number of rows: V
  - Number of columns: V
  - Entry (i,j) is 1 if vertex v_j is adjacent to vertex v_i else it is 0.



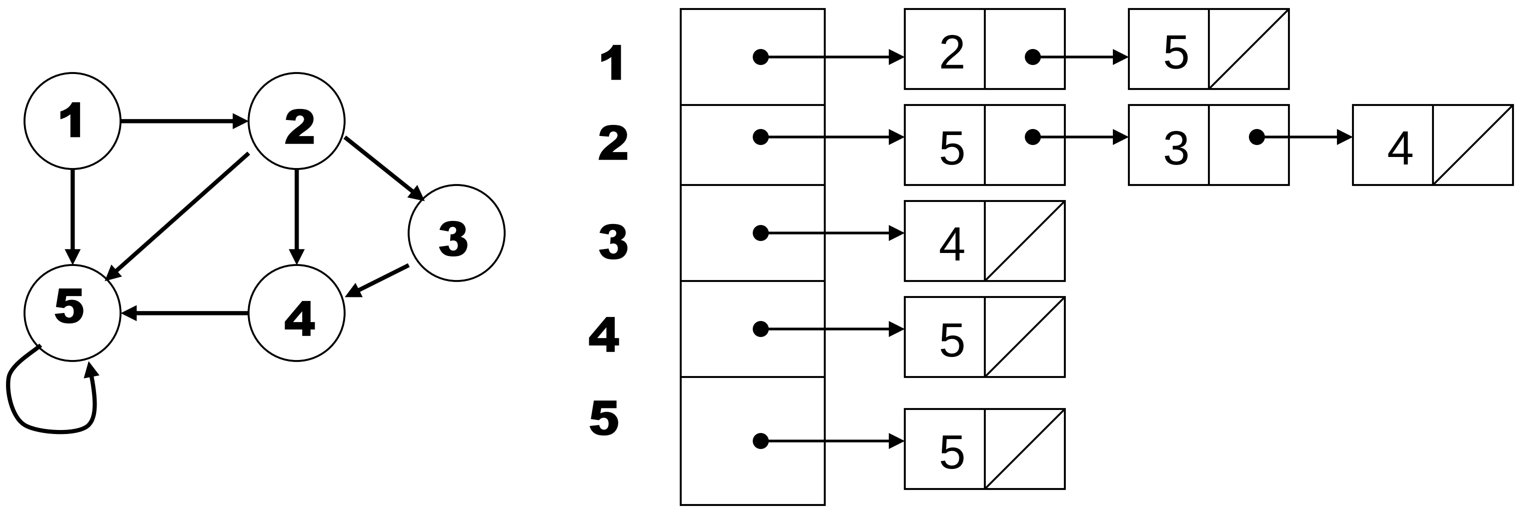
$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

Adjacency List – Undirected Graph

- ***Adjacency-list representation*** of a graph $G = (V, E)$ consists of an array ADJ of $|V|$ lists, one for each vertex in V. For each $u \in V$, $ADJ[u]$ points to all its adjacent vertices.



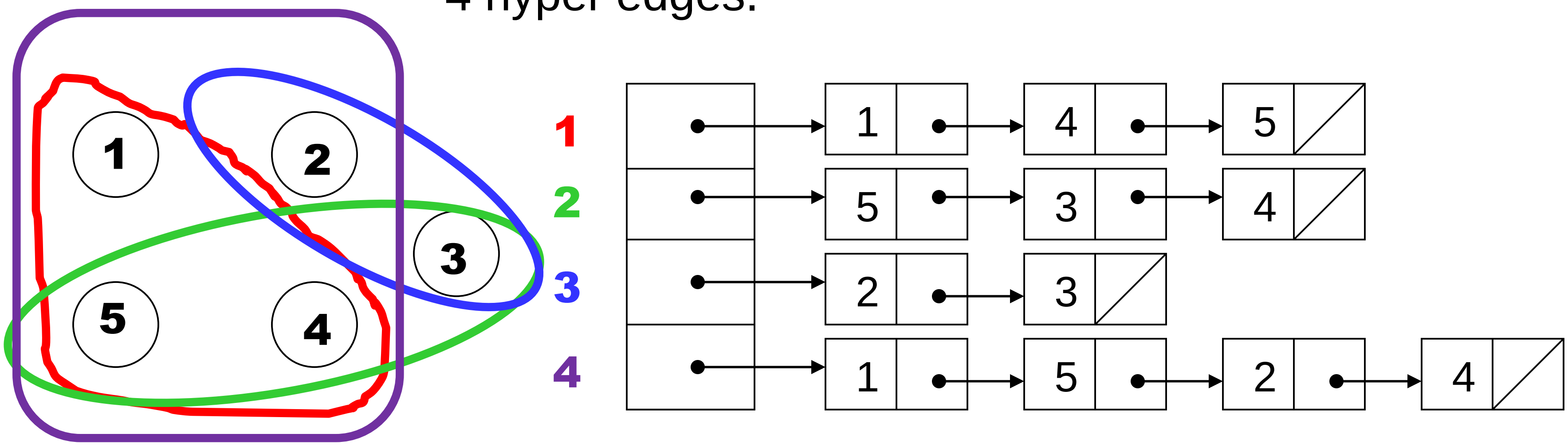
Adjacency List – Directed Graph



Variation: Can keep a second list of edges coming into a vertex.

Adjacency List – Hyper Graph

4 hyper edges.



inv c1 (.in(net1), .out(net4)) ;
buffer c2 (.in(net3), .out(net4)) ;
inv c3 (.in(net2), .out(net3)) ;
and c4 (.a(net1), .b(net2), .out(net4)) ;
xor c5 (.out(net1), .a(net2), .b(net4)) ;

Hypergraph models netlist

Features of Adjacency lists

- Advantage:
  - Saves space for sparse graphs. Most graphs are sparse.
  - "Visit" edges that start at v
    - Must traverse linked list of v
    - Size of linked list of v is degree(v)
    - θ(degree(v))
- Disadvantage:
  - Check for existence of an edge (v, u)
    - Must traverse linked list of v
    - Size of linked list of v is degree(v)
    - θ(degree(v))

Features of Adjacency List (cont.)

- Storage
  - We need V pointers to linked lists
  - For a directed graph the number of nodes (or edges) contained (referenced) in all the linked lists is

$$\sum_{v \in V} (\text{out-degree}\ (v)) = |\ E\ |.$$

So we need $\Theta(\ V + \ E\ )$

  - For an undirected graph the number of nodes is

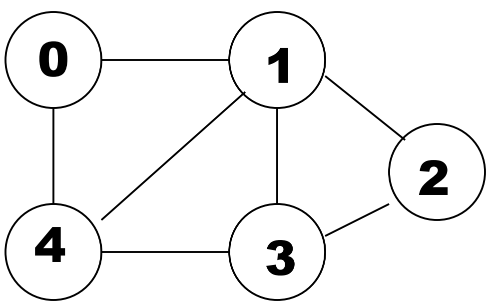
$$\sum_{v \in V} (\text{degree}\ (v)) = 2\ |\ E\ |$$

Also $\Theta(\ V + \ E\ )$

Adjacency Matrix – Undirected Graph

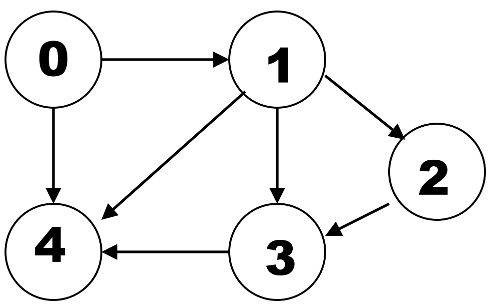
- ***Adjacency-matrix-representation*** of a graph $G = (V, E)$ is a $|V|$ x $|V/$ matrix $A = (a_{ij})$ such that

  a_{ij} = 1 (or some Object) if $(i, j) \in E$ and
  0 (or null) otherwise.



	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

Adjacency Matrix – Directed Graph



$$
\begin{array}{c|ccccc|}
 & 0 & 1 & 2 & 3 & 4 \\
\hline
0 & 0 & 1 & 0 & 0 & 1 \\
1 & 0 & 0 & 1 & 1 & 1 \\
2 & 0 & 0 & 0 & 1 & 0 \\
3 & 0 & 0 & 0 & 0 & 1 \\
4 & 0 & 0 & 0 & 0 & 0 \\
\hline
\end{array}
$$

Features of Adjacency Matrix

- Advantage:
  - Saves space on pointers for dense graphs, and on
  - small unweighted graphs using 1 bit per edge.
  - Check for existence of an edge (v, u)
    - (adjacency [i] [j]) == true?)
    - So adjacency [i][j] can be found in constant time: $\theta(1)$
- Disadvantage:
  - “visit” all the edges that start at v
    - Row v of the matrix must be traversed.
    - So finding all vertices that are adjacent to v: $\theta(|V|)$.

Features of Adjacency Matrix (cont.)

- Storage
  - $\Theta(|V|^2)$ (We usually just write, $\Theta(V^2)$)
  - For undirected graphs you can save storage (only $1/2(V^2)$) by noticing the adjacency matrix of an undirected graph is symmetric.
  - Need to update code to work with new representation.
  - Gain in space is offset by increase in the time required by the methods.

Properties of an Undirected Graph

- Each undirected graph G(V,E) can be characterized by four numbers
  1. Clique number: $\omega(G)$
  2. Clique cover number: $\kappa(G)$
  3. Stability number: $\alpha(G)$
  4. Chromatic number: $\chi(G)$

Clique Number - $\omega(G)$

- Clique number of a graph G is the cardinality of its largest (maximum) clique.
- A graph is said to be partitioned into cliques if its vertex set is partitioned into (disjoint) subsets, each one inducing a clique.
- A graph is said to be covered by cliques when the vertex set can be subdivided into (possibly overlapping) subsets, each one inducing a clique.
- A clique partition is a disjoint clique cover.

Clique Cover - $\kappa(G)$

- Clique cover number of a graph G is the cardinality of a minimum clique partition which is equal to the cardinality of a minimum clique cover.

Stability Number - $\alpha(G)$

- A stable (or independent) set is a subset of vertices with the property that no two vertices in the stable set are adjacent.

- The stability number of a graph is the cardinality of its largest stable set.

Chromatic Number - $\chi(G)$

- A coloring of a graph is a partition of the vertices into subsets, such that each is a stable set.
- The chromatic number is the smallest number that can be the cardinality of such a partition. Visually, it is the minimum number of colors needed to color the vertices, such that no edge has end-points with the same color.

Relationship Among Four Graph Numbers

- The size of the maximum clique is a lower bound for the chromatic number because all vertices in that clique must be colored differently. So:

$$\omega(G) \leq \chi(G)$$

- Similarly, the stability number is a lower bound for the clique cover number, since each vertex of the stable set must belong to a different clique of a clique cover. Thus,

$$\alpha(G) \leq \kappa(G)$$

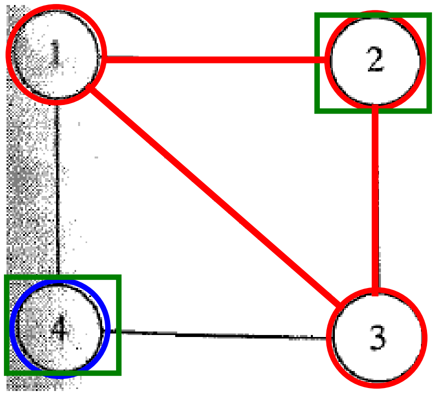
- A graph is said to be perfect if:

$$\omega(G) = \chi(G) \;\; and \;\; \alpha(G) = \kappa(G)$$

Example

1. Clique number: $\omega(G) = 3$
   - The size of maximum clique $\{v_1, v_2, v_3\}$ is 3.
2. Clique cover number: $\kappa(G) = 2$
   - The graph can be partitioned into cliques $\{v_1, v_2, v_3\}$ and $\{v_4\}$. Alternatively, it can be covered by cliques $\{v_1, v_2, v_3\}$ and $\{v_1, v_3, v_4\}$. The clique cover number is 2.
3. Stability number: $\boxed{\alpha(G) = 2}$
   - The largest stable set is $\{v_2, v_4\}$. The stability number is 2.
4. Chromatic number: $\chi(G) = 3$
   - A minimum coloring would require three colors for $\{v_1, v_2, v_3\}$. Vertex v_4 can have the same color as v_2. Hence, the chromatic number is 3.

- This graph is perfect.



More on Trees

- A tree is a connected acyclic graph.
- A tree with two or more vertices is 2-chromatic
- A tree is a minimally-connected graph
  - there is exactly one path between every pair of vertices in the graph
- A graph with n vertices is a tree if it is connected and has n-1 edges.
  - in a tree we have $|E|=|V|-1$
- Proof: ?

Basic Search Algorithms

Breadth-first search

Graph Searching: Breadth-First Search

Graph $G = (V, E)$, directed or undirected with adjacency list repres.

GOAL: Systematically explores edges of G to

- discover every vertex reachable from the source vertex s
- compute the shortest path distance of every vertex from the source vertex s
- produce a breadth-first tree (BFT) G_{Π} with root s
  - BFT contains all vertices reachable from s
  - the unique path from any vertex v to s in G_{Π} constitutes a shortest path from s to v in G

IDEA: Expanding frontier across the breadth -greedy-

- propagate a wave 1 edge-distance at a time
- using a FIFO queue: O(1) time to update pointers to both ends

Breadth-First Search Algorithm

Maintains the following fields for each $u \in V$

- color[u]: color of u
  - WHITE : not discovered yet
  - GRAY : discovered and to be or being processed
  - BLACK: discovered and processed
- $\Pi[u]$: parent of u (NIL of $u = s$ or u is not discovered yet)
- $d[u]$: distance of u from s

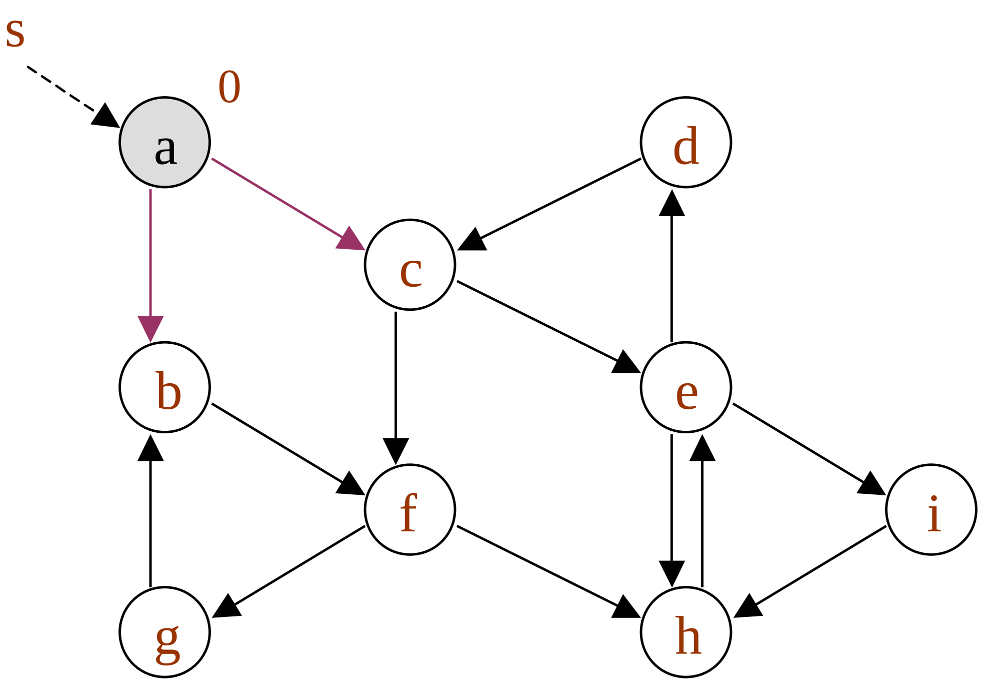
Processing a vertex = scanning its adjacency list

Breadth-First Search Algorithm

```
BFS(G, s)
    for each u ∈ V − {s} do
        color[u] ← WHITE
        Π[u] ← NIL; d [u] ← ∞
    color[s] ← GRAY
    Π[s] ← NIL; d [s] ← 0
    Q ← {s}
    while Q ≠ ∅ do
        u ← head[Q]
        for each v in Adj[u] do
            if color[v] = WHITE then
                color[v] ← GRAY
                Π[v] ← u
                d [v] ← d [u] + 1
                ENQUEUE(Q, v)
        DEQUEUE(Q)
        color[u] ← BLACK
```

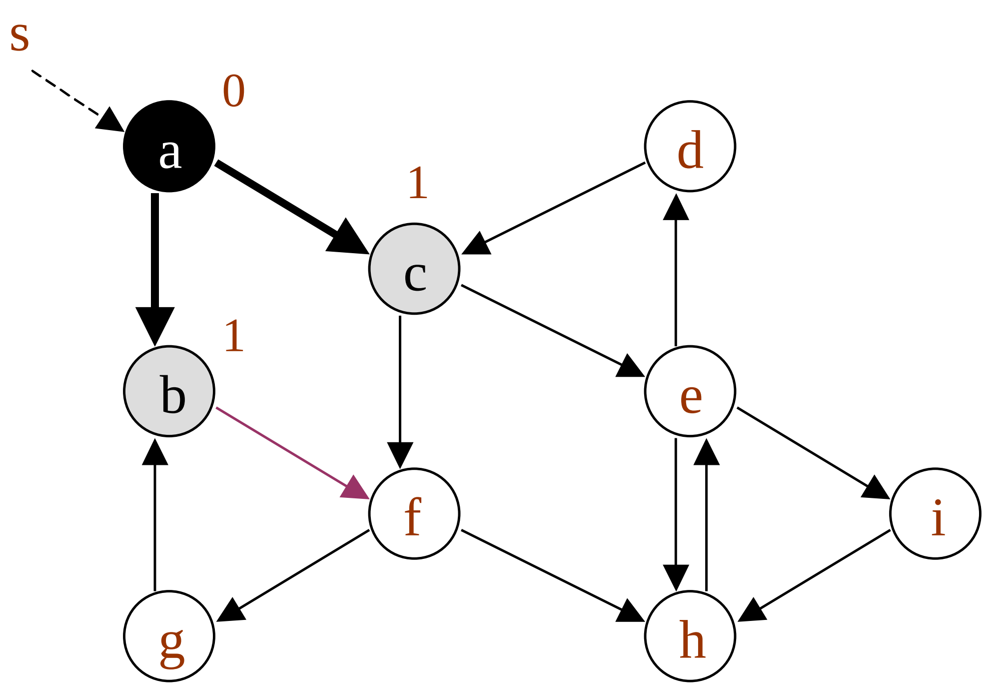
Breadth-First Search

Sample Graph:



FIFO queue Q	just after processing vertex
⟨a⟩	-

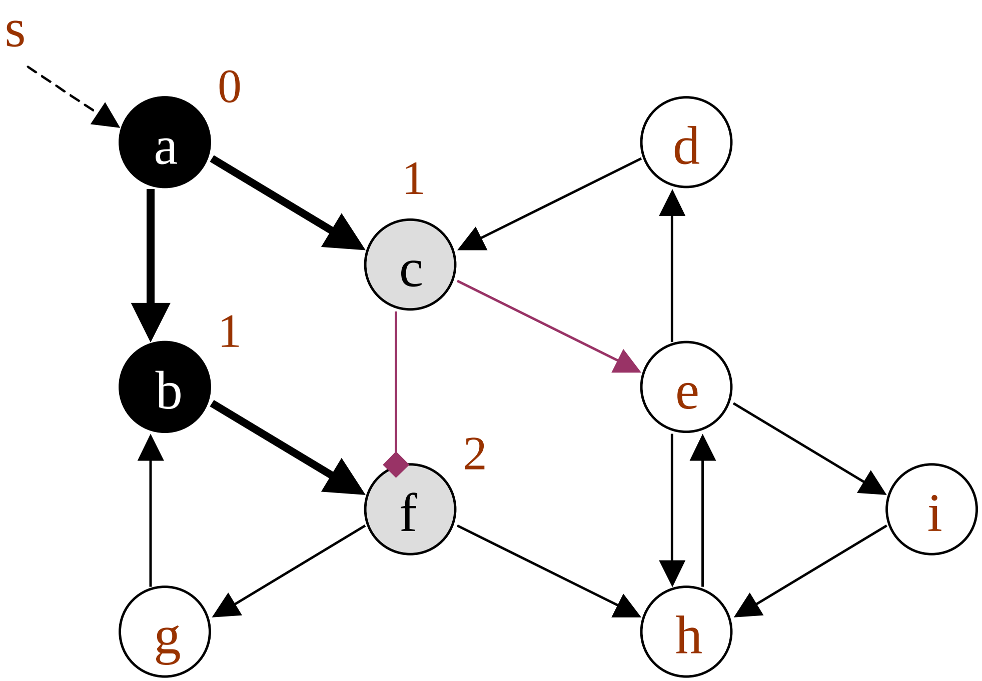
Breadth-First Search



FIFO queue Q	just after processing vertex
⟨a⟩	-
⟨a,b,c⟩	a

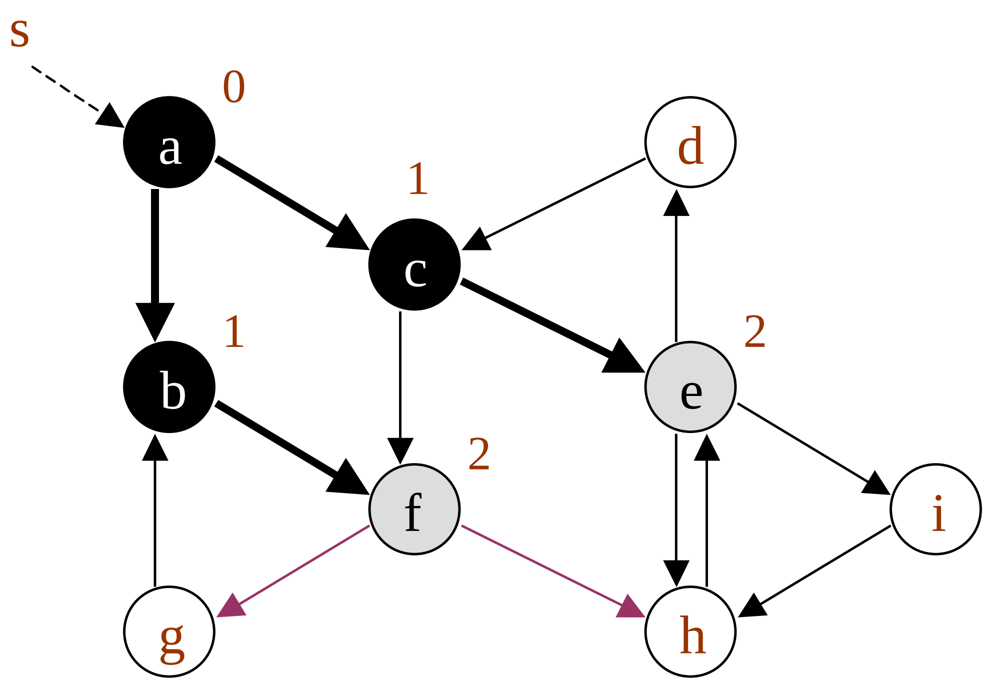
↑

Breadth-First Search



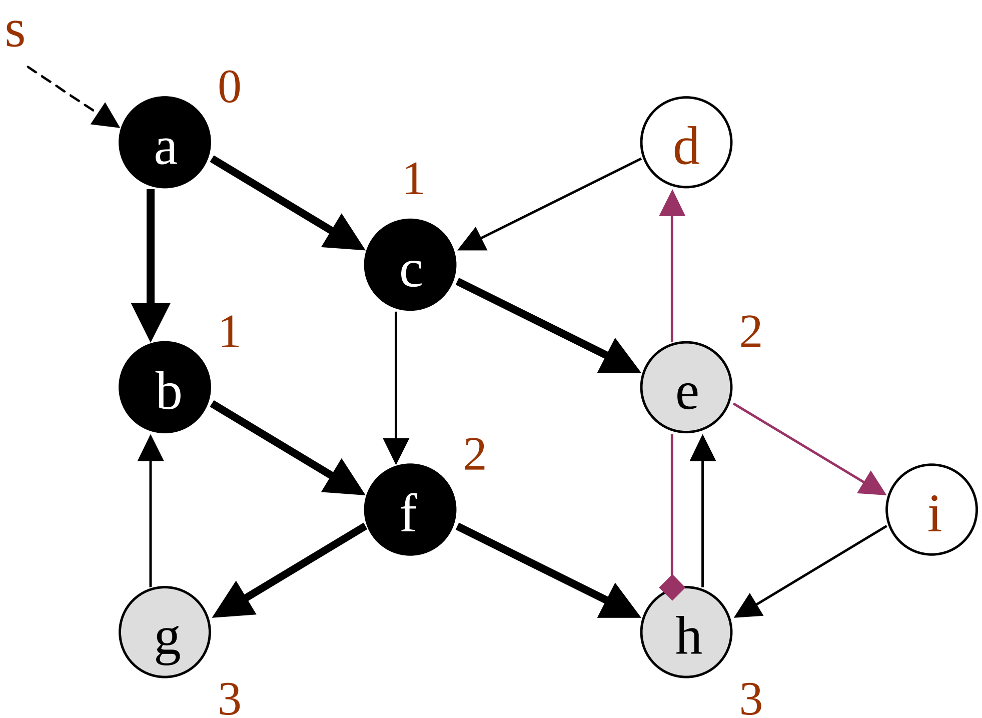
FIFO queue *Q*	**just after processing vertex**
⟨a⟩	**-**
⟨a,b,c⟩	**a**
⟨a,b,c,f⟩	**b**

Breadth-First Search



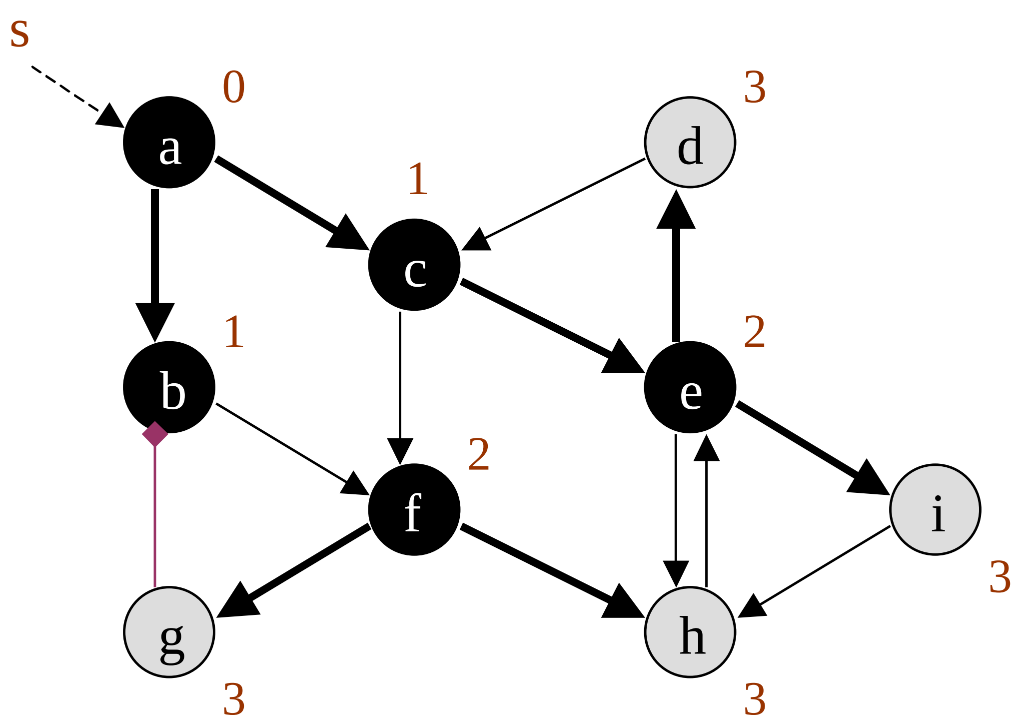
FIFO queue Q	**just after processing vertex**
⟨a⟩	**-**
⟨a,b,c⟩	**a**
⟨a,b,c,f⟩	**b**
⟨a,b,c,f,e⟩	**c**

Breadth-First Search



FIFO queue Q	just after processing vertex
⟨a⟩	-
⟨a,b,c⟩	a
⟨a,b,c,f⟩	b
⟨a,b,c,f,e⟩	c
⟨a,b,c,f,e,g,h⟩	f

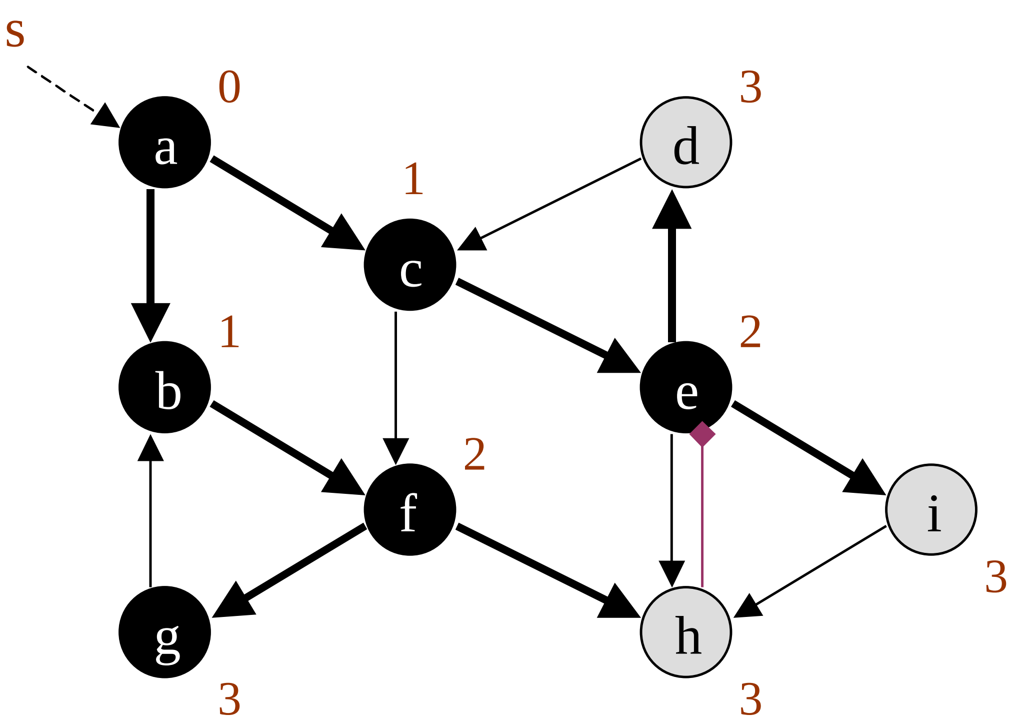
Breadth-First Search



FIFO queue *Q*	just after processing vertex
⟨a⟩	-
⟨a,b,c⟩	a
⟨a,b,c,f⟩	b
⟨a,b,c,f,e⟩	c
⟨a,b,c,f,e,g,h⟩	f
⟨a,b,c,f,e,g,h,d,i⟩	e

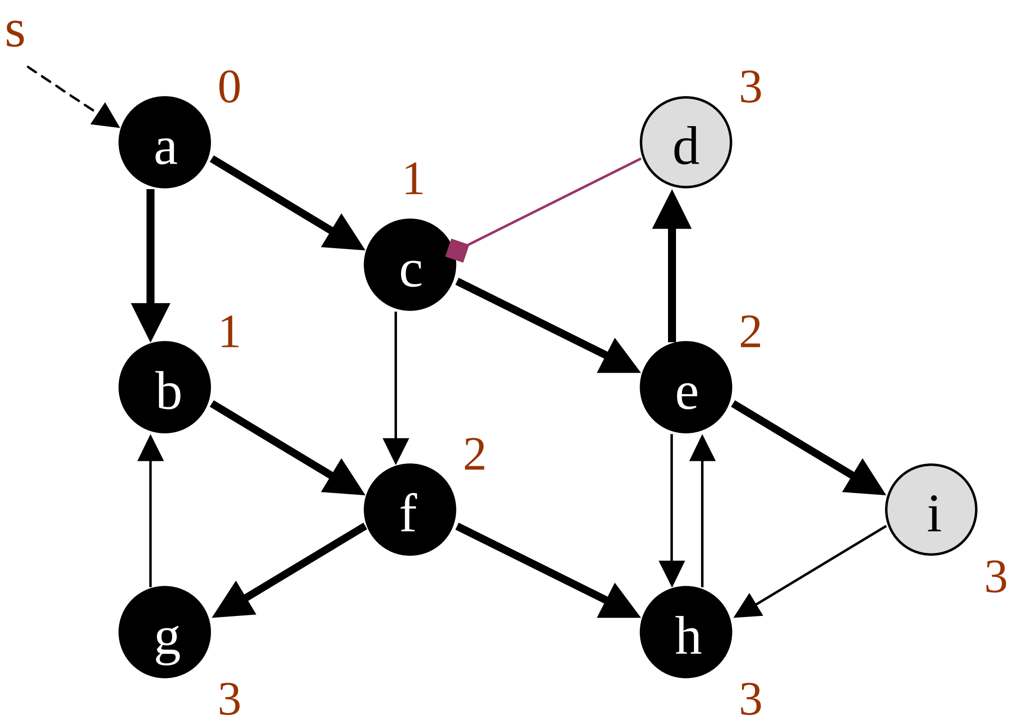
all distances are filled in after processing e

Breadth-First Search



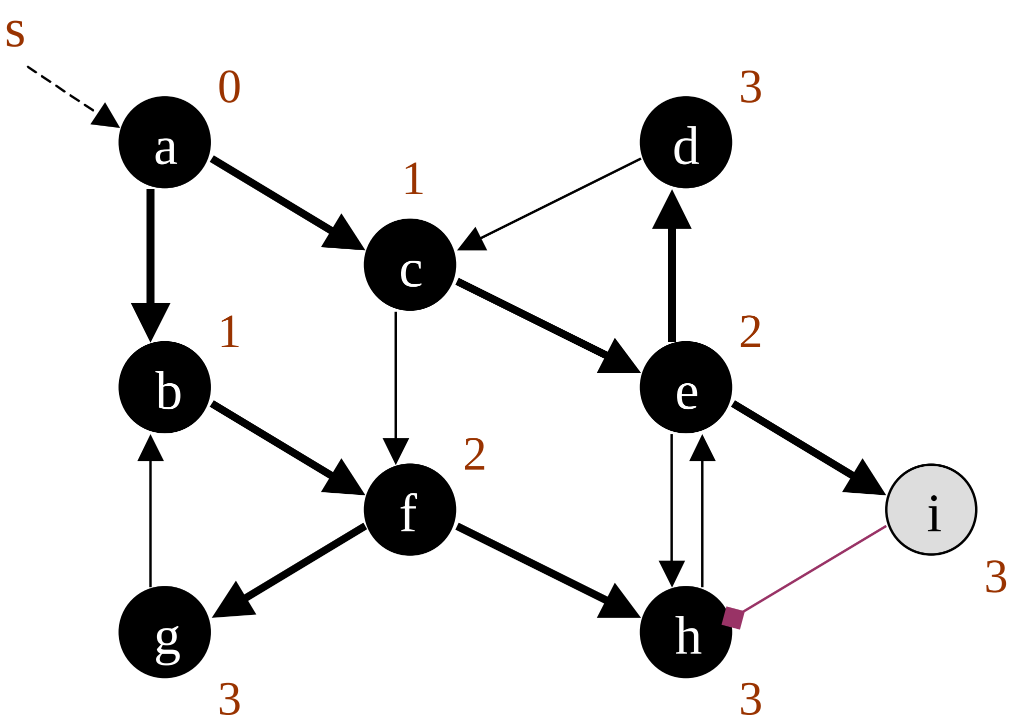
FIFO queue *Q*	just after processing vertex
⟨a⟩	-
⟨a,b,c⟩	a
⟨a,b,c,f⟩	b
⟨a,b,c,f,e⟩	c
⟨a,b,c,f,e,g,h⟩	f
⟨a,b,c,f,e,g,h,d,i⟩	g

Breadth-First Search



FIFO queue *Q*	just after processing vertex
⟨a⟩	-
⟨a,b,c⟩	a
⟨a,b,c,f⟩	b
⟨a,b,c,f,e⟩	c
⟨a,b,c,f,e,g,h⟩	f
⟨a,b,c,f,e,g,h,d,i⟩	h

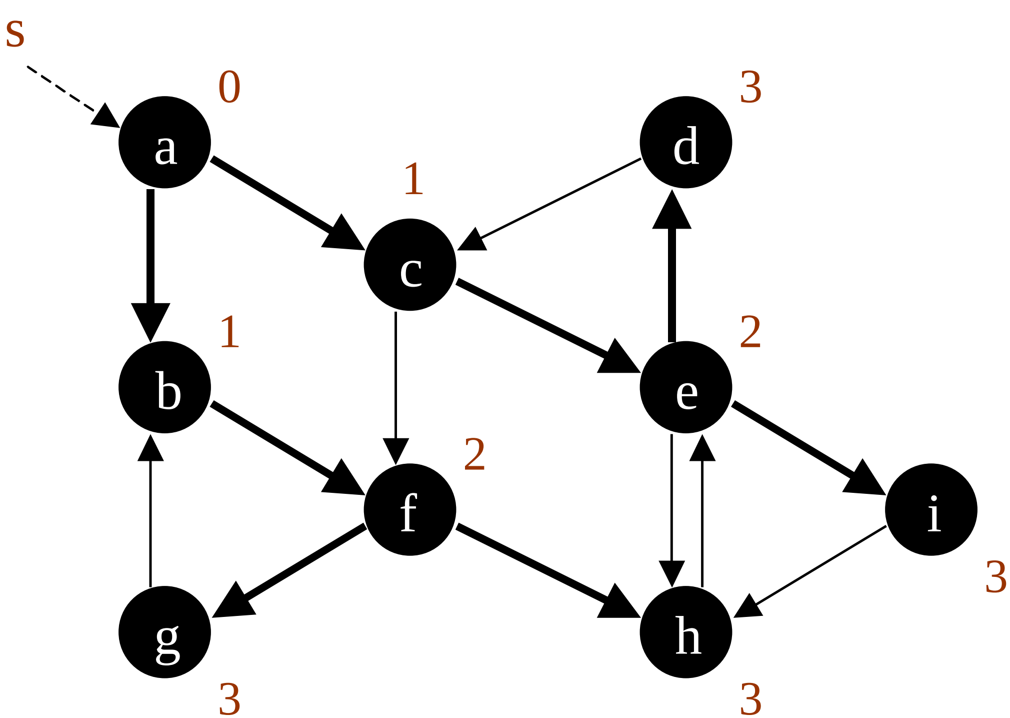
Breadth-First Search



FIFO queue *Q*	just after processing vertex
⟨a⟩	-
⟨a,b,c⟩	a
⟨a,b,c,f⟩	b
⟨a,b,c,f,e⟩	c
⟨a,b,c,f,e,g,h⟩	f
⟨a,b,c,f,e,g,h,d,i⟩	d

↑

Breadth-First Search



FIFO queue *Q*	just after processing vertex
⟨a⟩	-
⟨a,b,c⟩	a
⟨a,b,c,f⟩	b
⟨a,b,c,f,e⟩	c
⟨a,b,c,f,e,g,h⟩	f
⟨a,b,c,f,e,g,h,d,i⟩	i

algorithm terminates: all vertices are processed

Breadth-First Search Algorithm

Running time: $O(V+E)$ = considered linear time in graphs

- initialization: $\Theta(V)$
- queue operations: $O(V)$
  - each vertex enqueued and dequeued at most once
  - both enqueue and dequeue operations take $O(1)$ time
- processing gray vertices: $O(E)$
  - each vertex is processed at most once and

$$\sum_{u \in V} |Adj[u]| = \Theta(E)$$

Basic Search Algorithms

Depth-first search

Depth-First Search

- Graph $G=(V,E)$ directed or undirected
- Adjacency list representation
- Goal: Systematically explore every vertex and every edge
- Idea: search deeper whenever possible
  - —Using a LIFO queue (Stack; FIFO queue used in BFS)

Depth-First Search

- Maintains several fields for each $v \in V$
- Like BFS, colors the vertices to indicate their states. Each vertex is
  —Initially white,
  —grayed when discovered,
  —blackened when finished
- Like BFS, records discovery of a white v during scanning Adj[u] by $\pi[v] \leftarrow u$

Depth-First Search

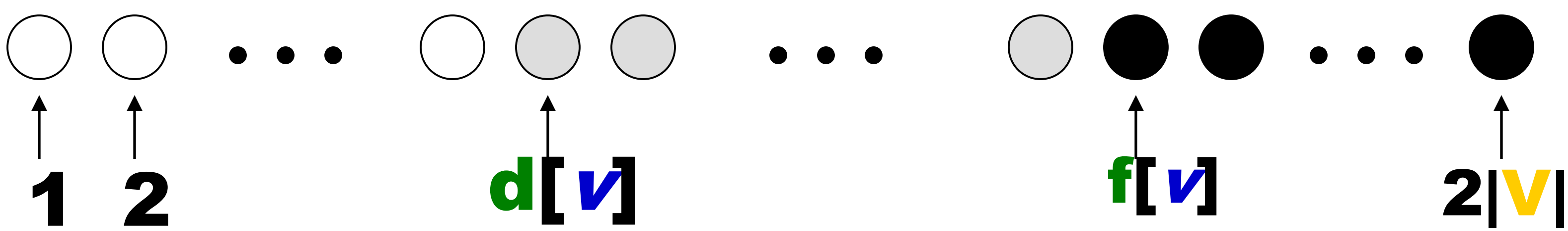
- Unlike BFS, predecessor graph G_π produced by DFS forms spanning forest
- $G_\pi=(V,E_\pi)$ where

  $$E_\pi=\{(\pi[v],v): v\in V \text{ and } \pi[v]\neq \text{NIL}\}$$
- $G_\pi=$ depth-first forest (DFF) is composed of disjoint depth-first trees (DFTs)

Depth-First Search

- DFS also timestamps each vertex with two timestamps
- d[*v*]: records when *v* is first discovered and grayed
- f[*v*]: records when *v* is finished and blackened
- Since there is only one discovery event and finishing event for each vertex we have $1 \le d[v] < f[v] \le 2|V|$

Time axis for the color of a vertex



Depth-First Search

DFS(G)

for each $u \in V$ **do**

color[u] $\leftarrow$ *white*

$\pi[u] \leftarrow$ NIL

$time \leftarrow 0$

for each $u \in V$ **do**

if color[u] = *white* **then**

DFS-VISIT(G, u**)**

DFS-VISIT(G, u**)**

color[u] $\leftarrow$ *gray*

d[u] $\leftarrow time \leftarrow time + 1$

for each $v \in$ Adj[u] **do**

if color[v] = *white* **then**

$\pi[v] \leftarrow u$

DFS-VISIT(G, v**)**

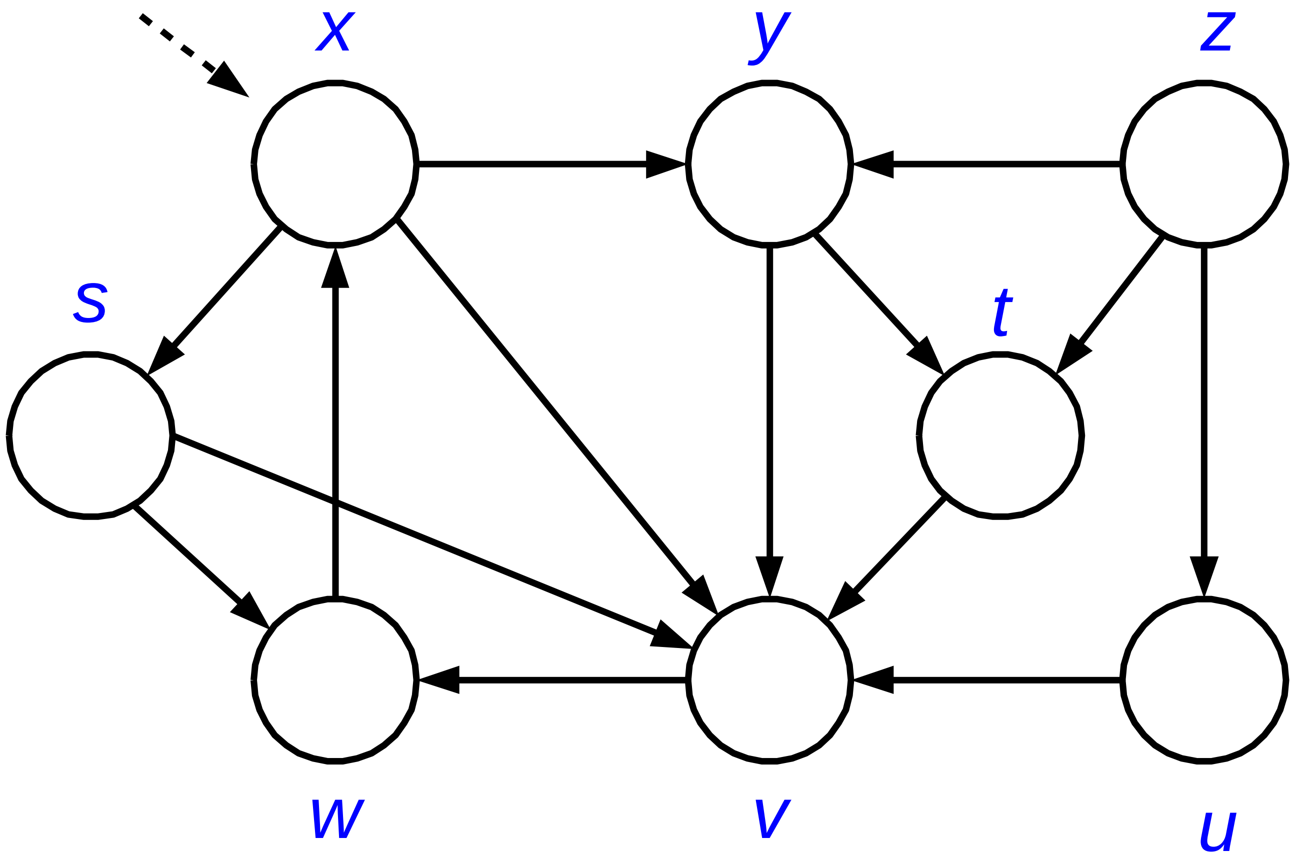
color[u] $\leftarrow$ *black*

f[u] $\leftarrow time \leftarrow time + 1$

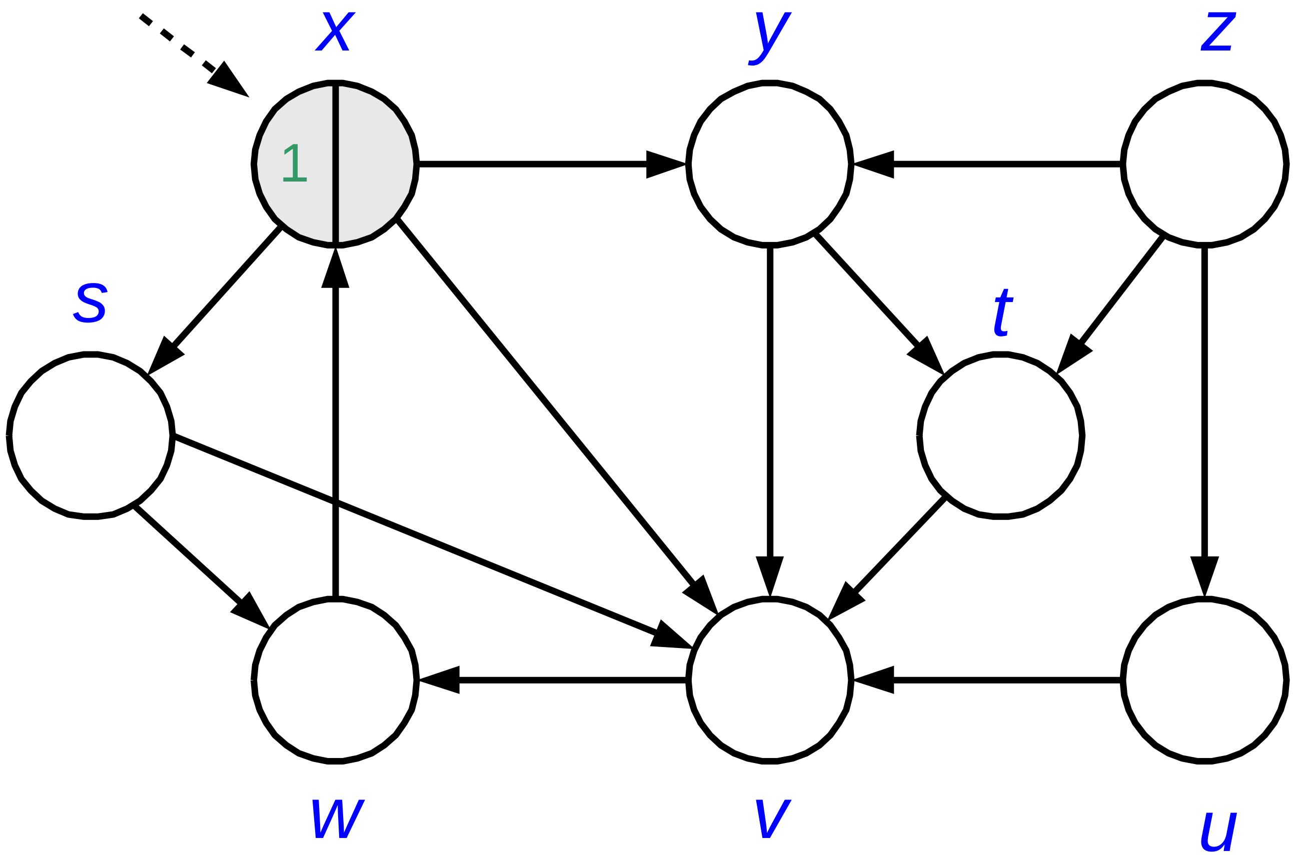
Depth-First Search

- Running time: $\Theta(V+E)$
- Initialization loop in **DFS** : $\Theta(V)$
- Main loop in **DFS**: $\Theta(V)$ exclusive of time to execute calls to **DFS-VISIT**
- **DFS-VISIT** is called exactly once for each $v \in V$ since
  - —**DFS-VISIT** is invoked only on white vertices and
  - —**DFS-VISIT(**G, u**)** immediately colors u as gray
- For loop of **DFS-VISIT(**G, u**)** is executed $|Adj[u]|$ time
- Since $\Sigma\ |Adj[u]| = E$, total cost of executing loop of **DFS-VISIT** is $\Theta(E)$

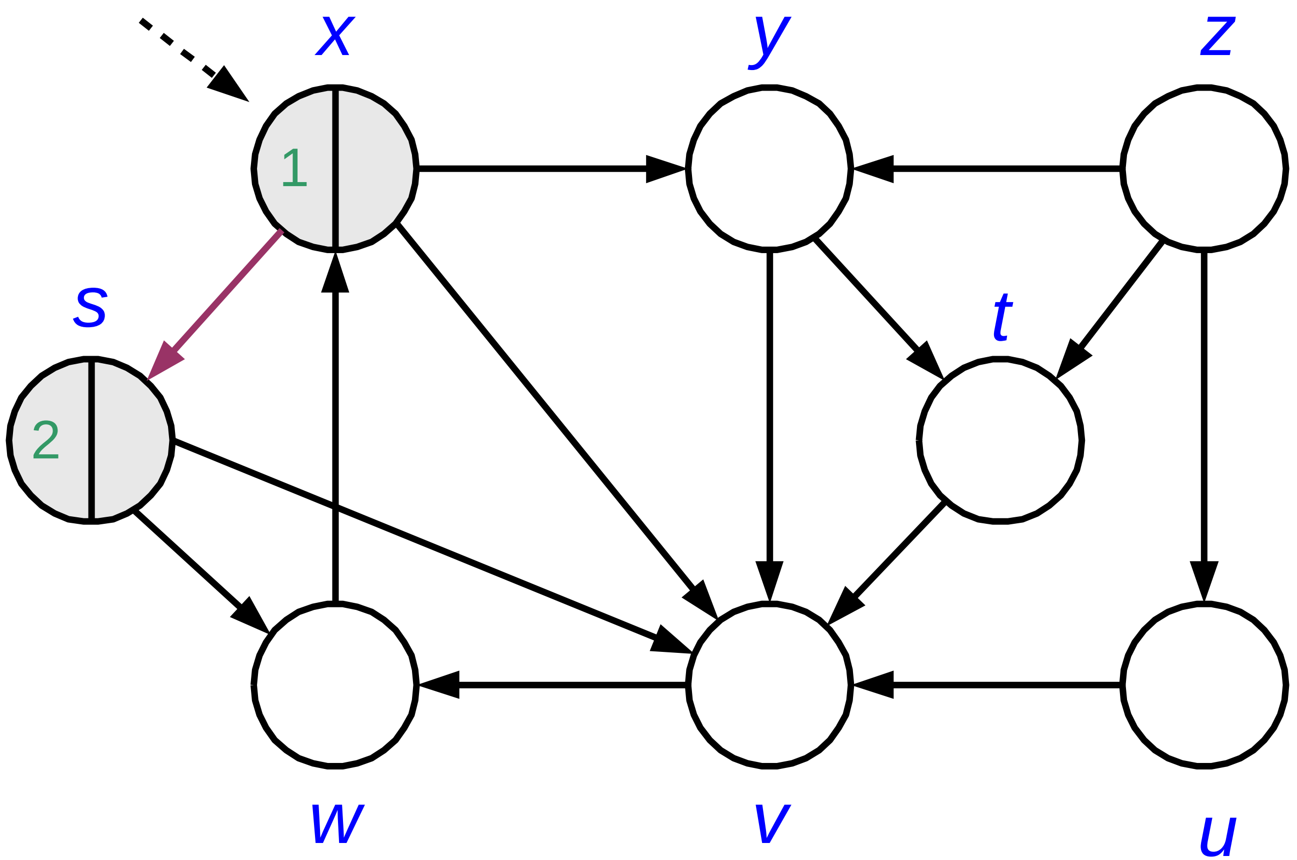
Depth-First Search: Example



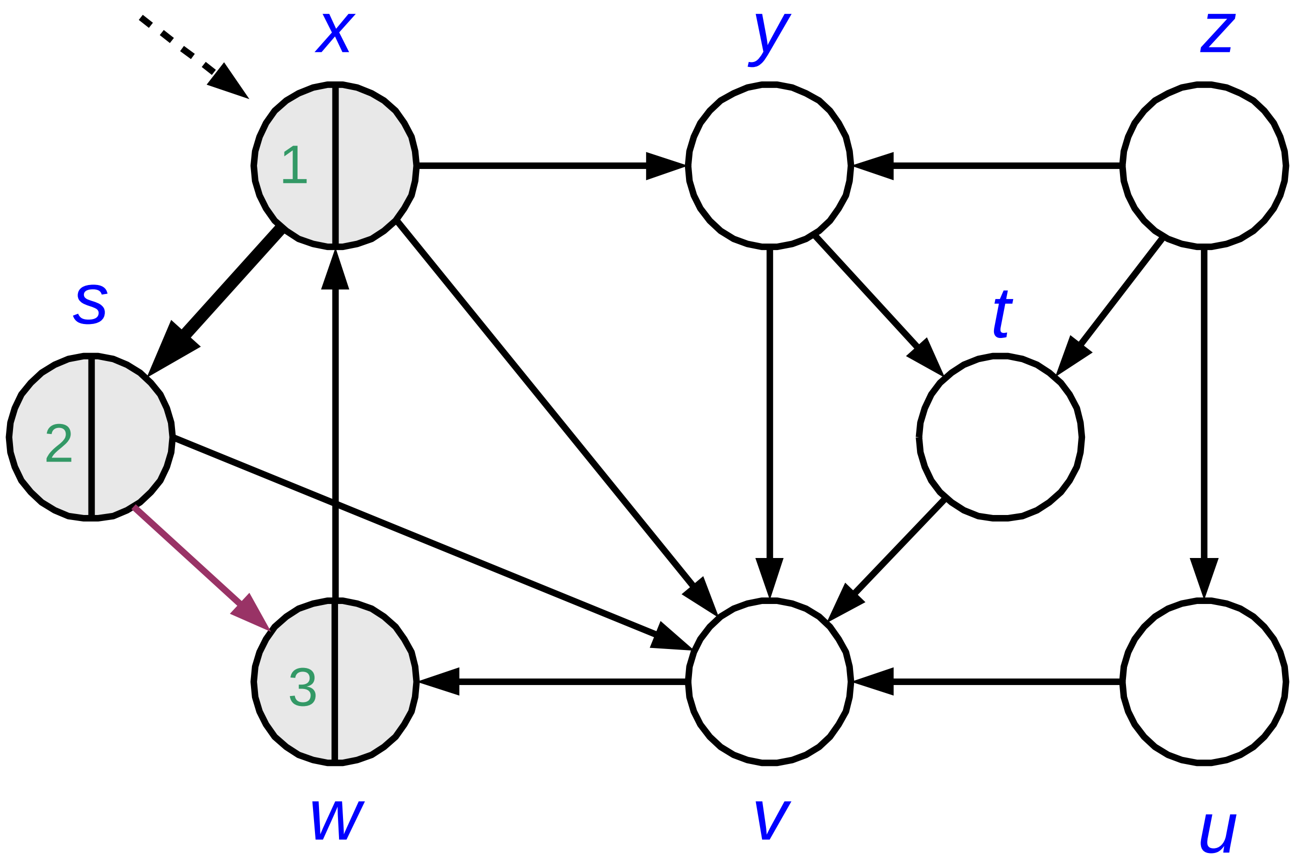
Depth-First Search: Example



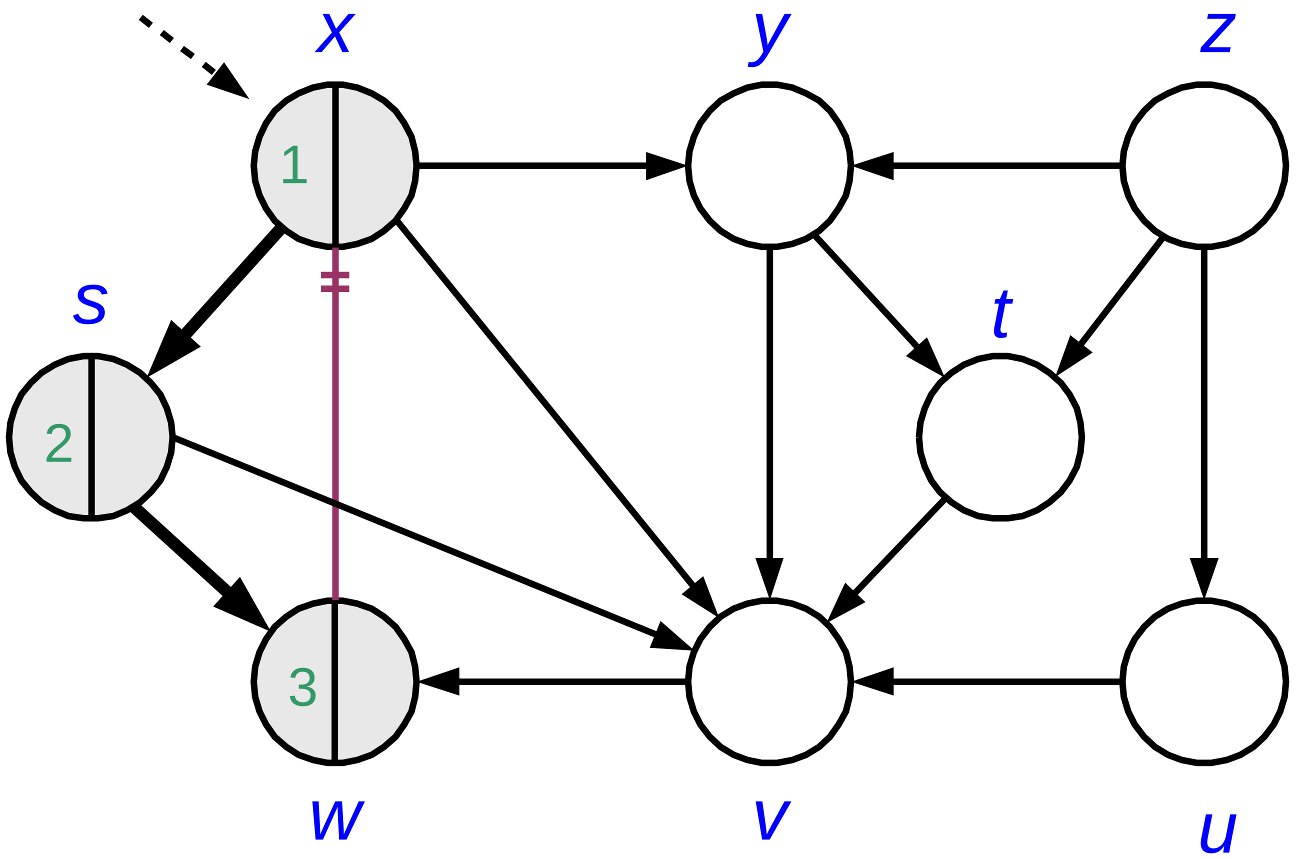
Depth-First Search: Example



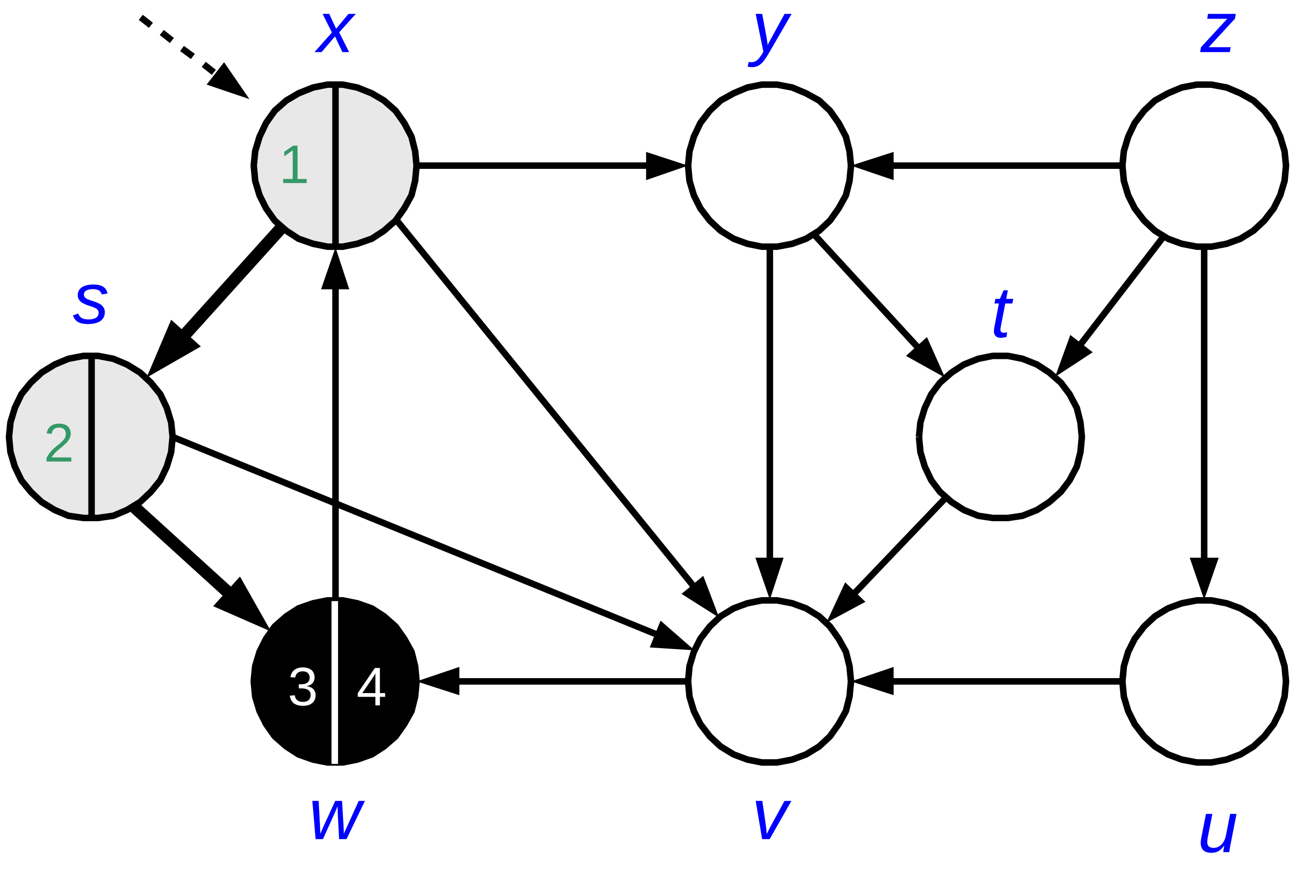
Depth-First Search: Example



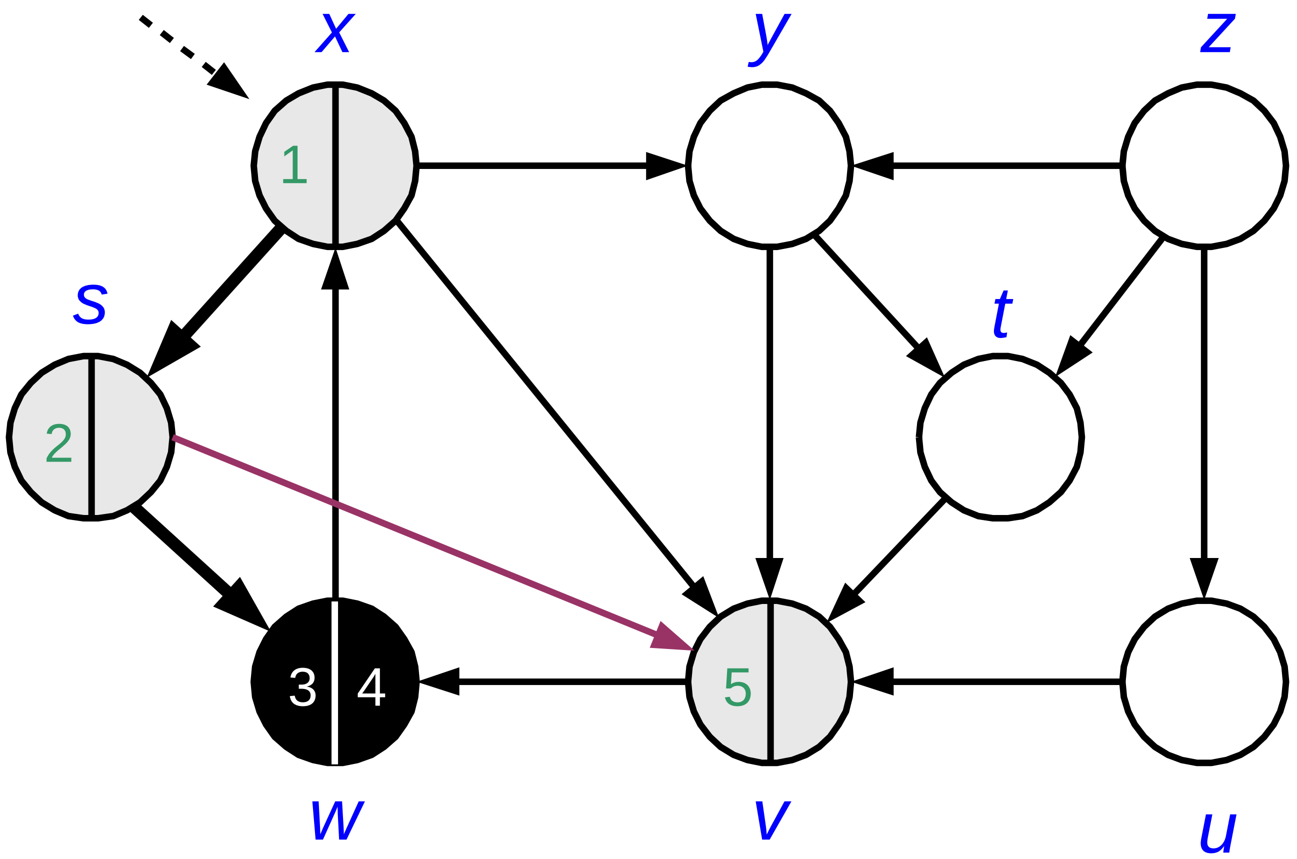
Depth-First Search: Example



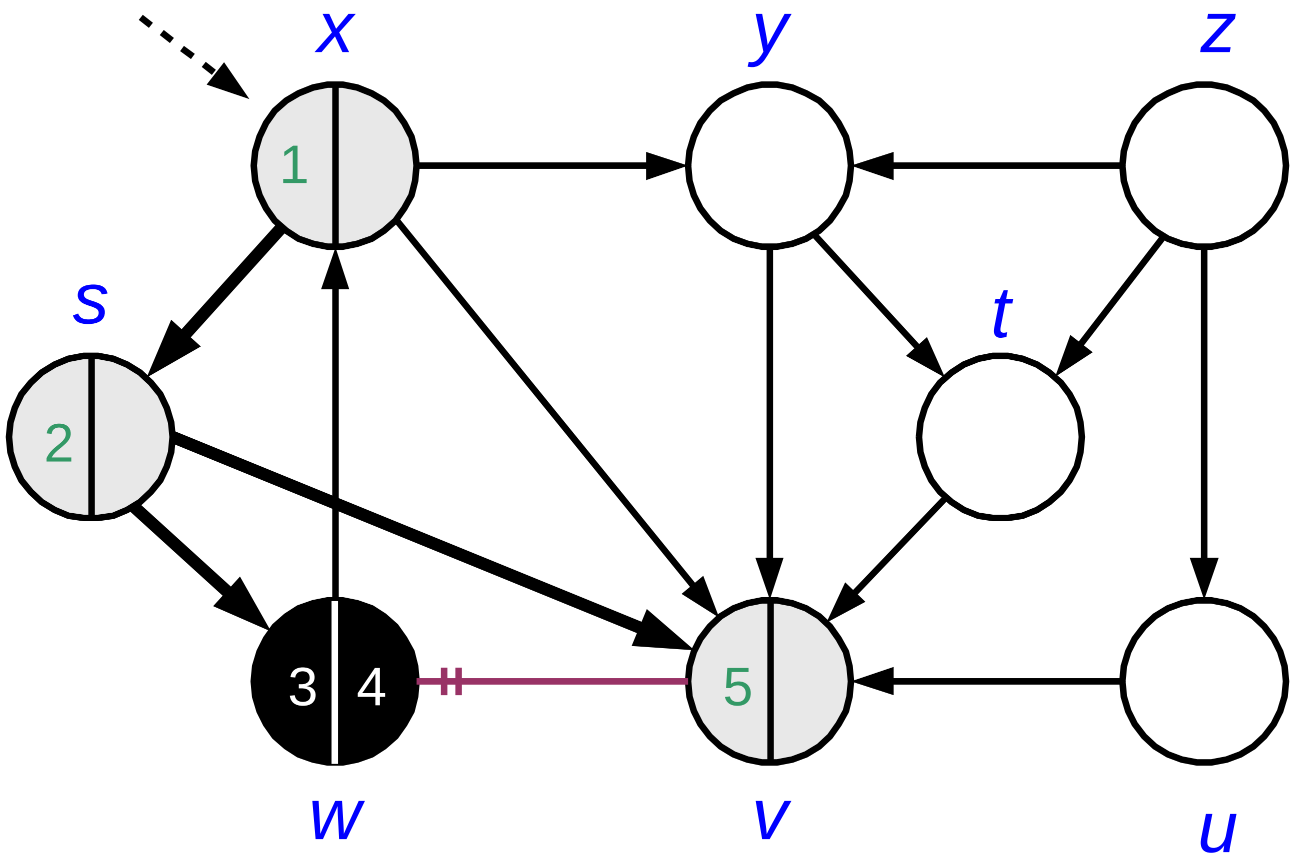
Depth-First Search: Example



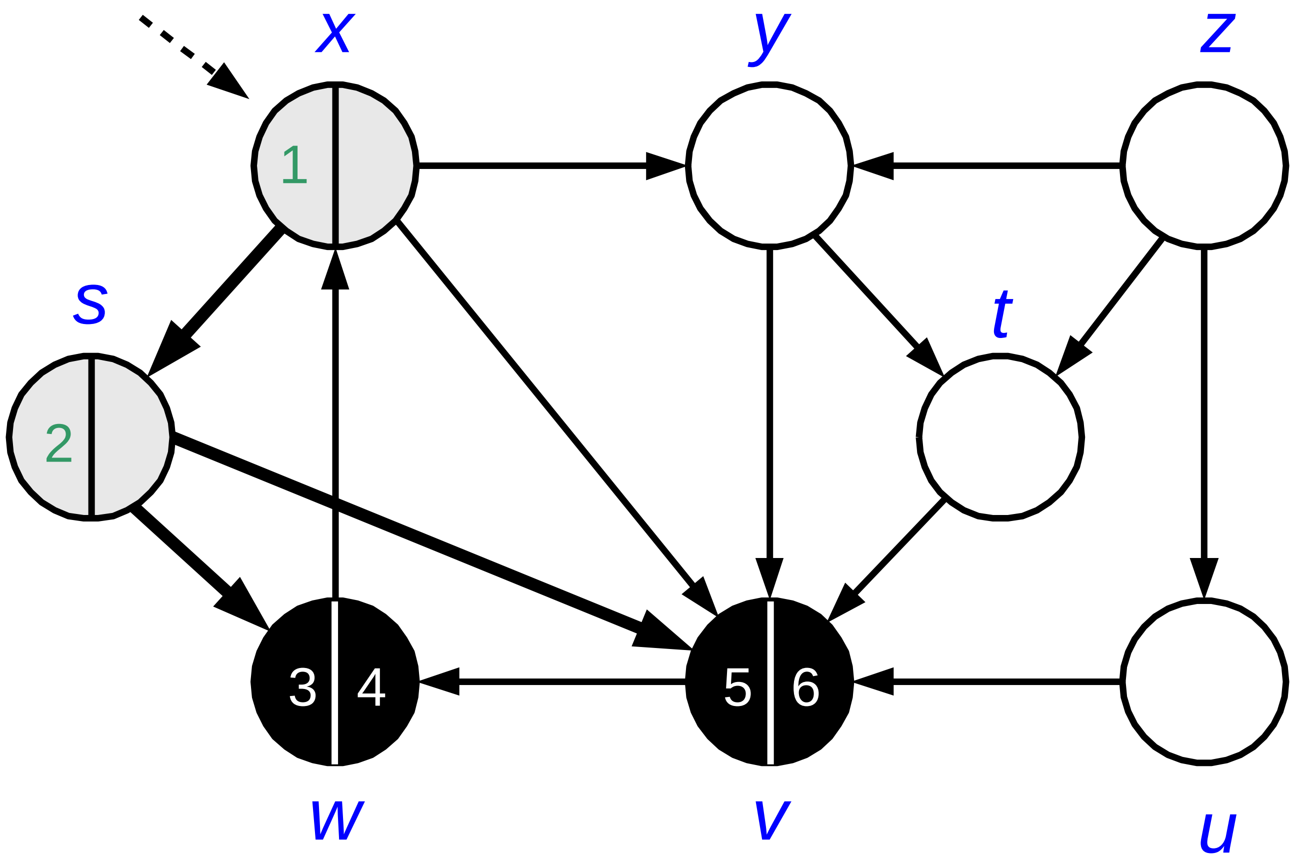
Depth-First Search: Example



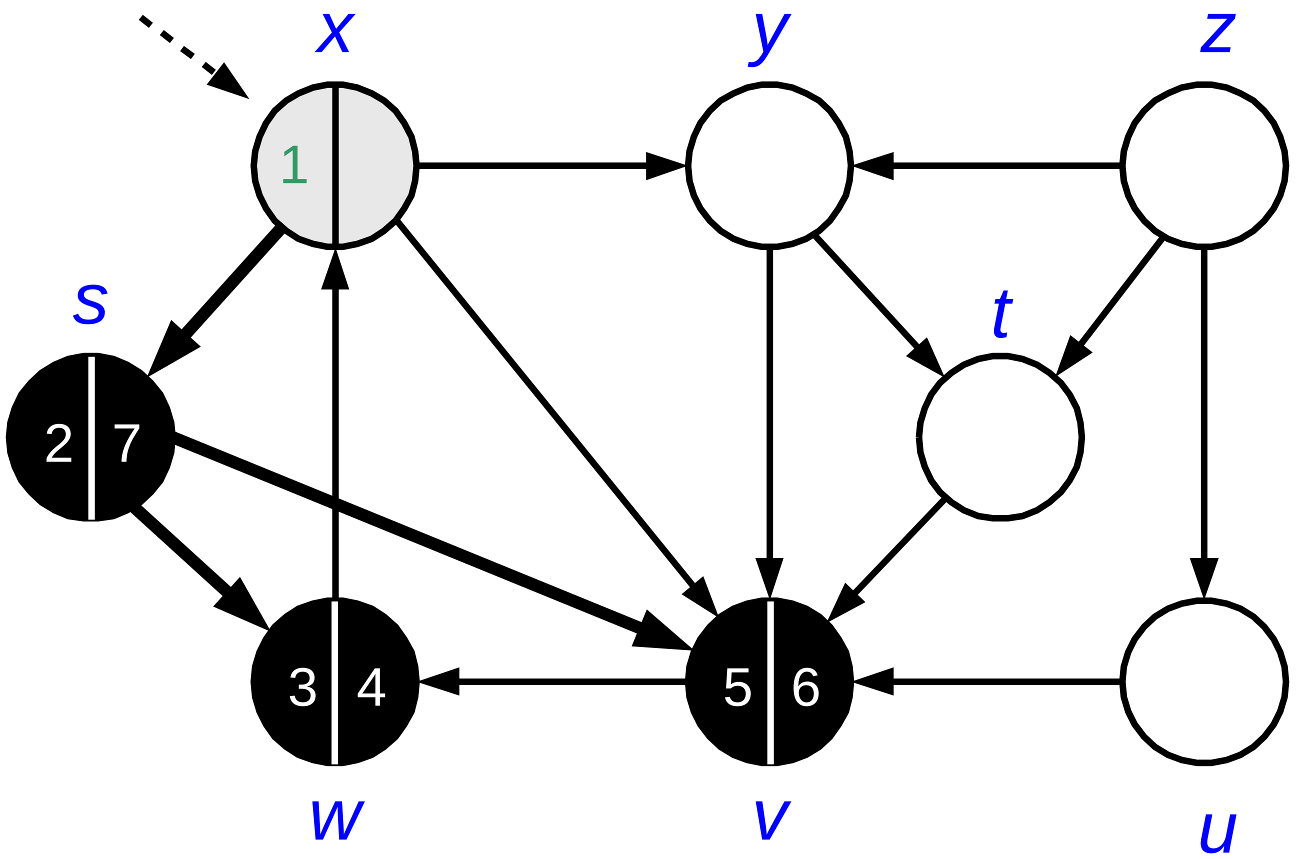
Depth-First Search: Example



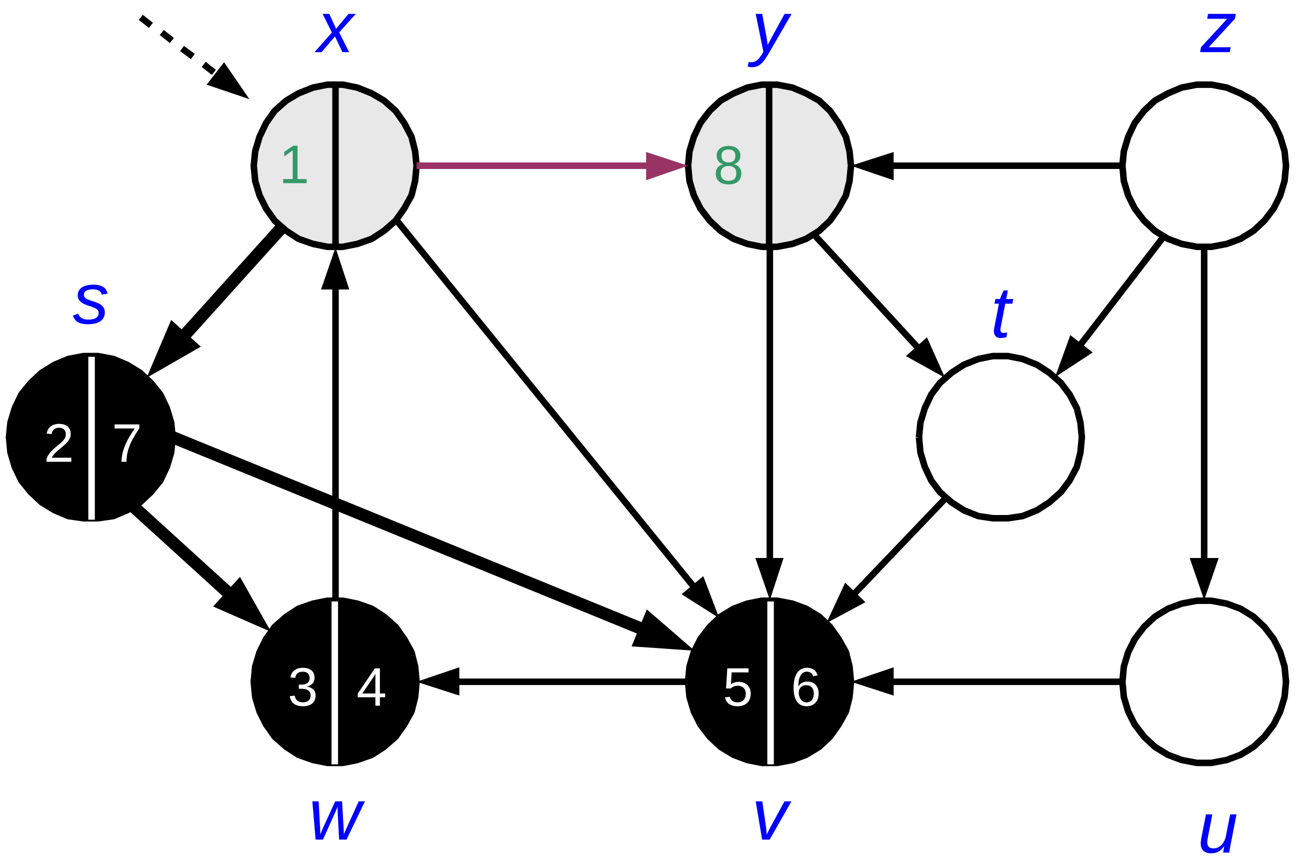
Depth-First Search: Example



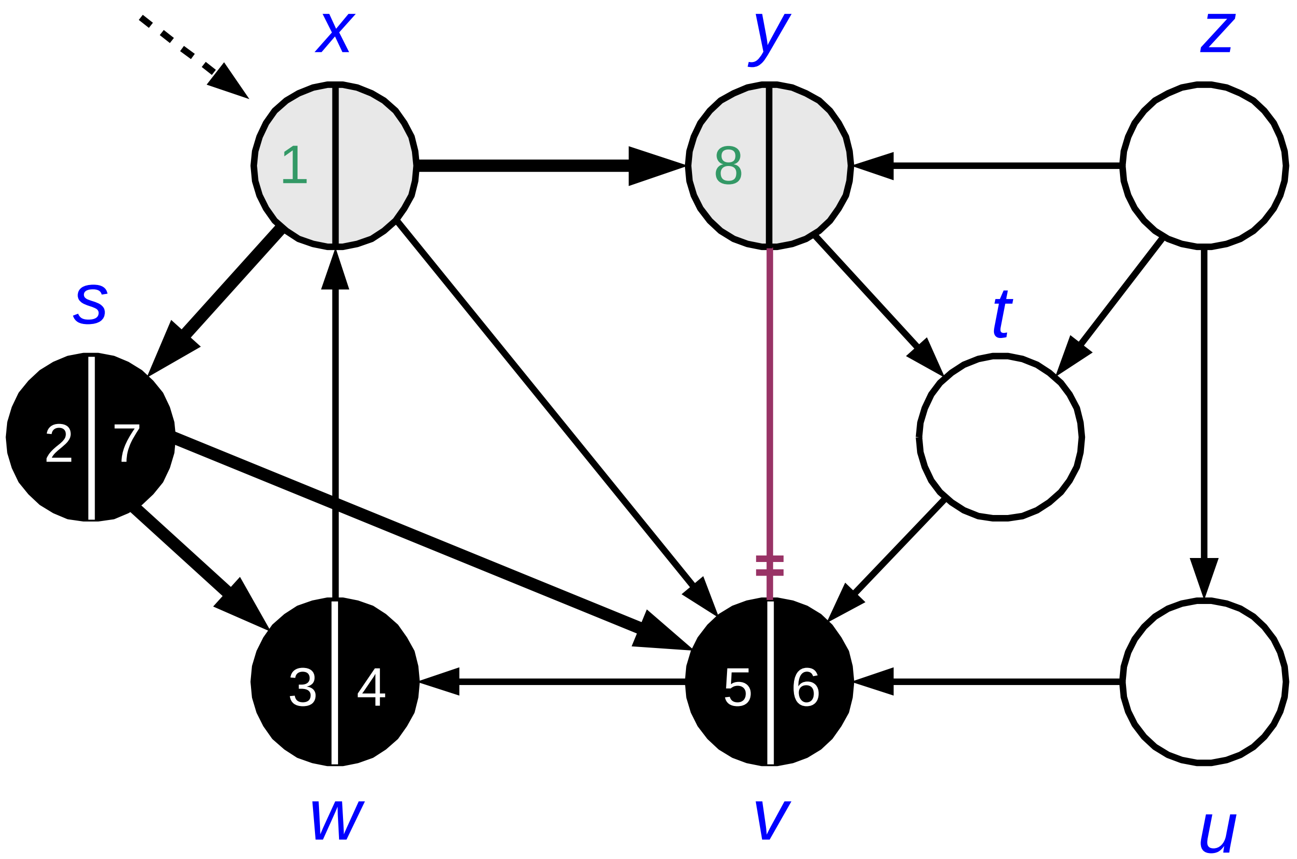
Depth-First Search: Example



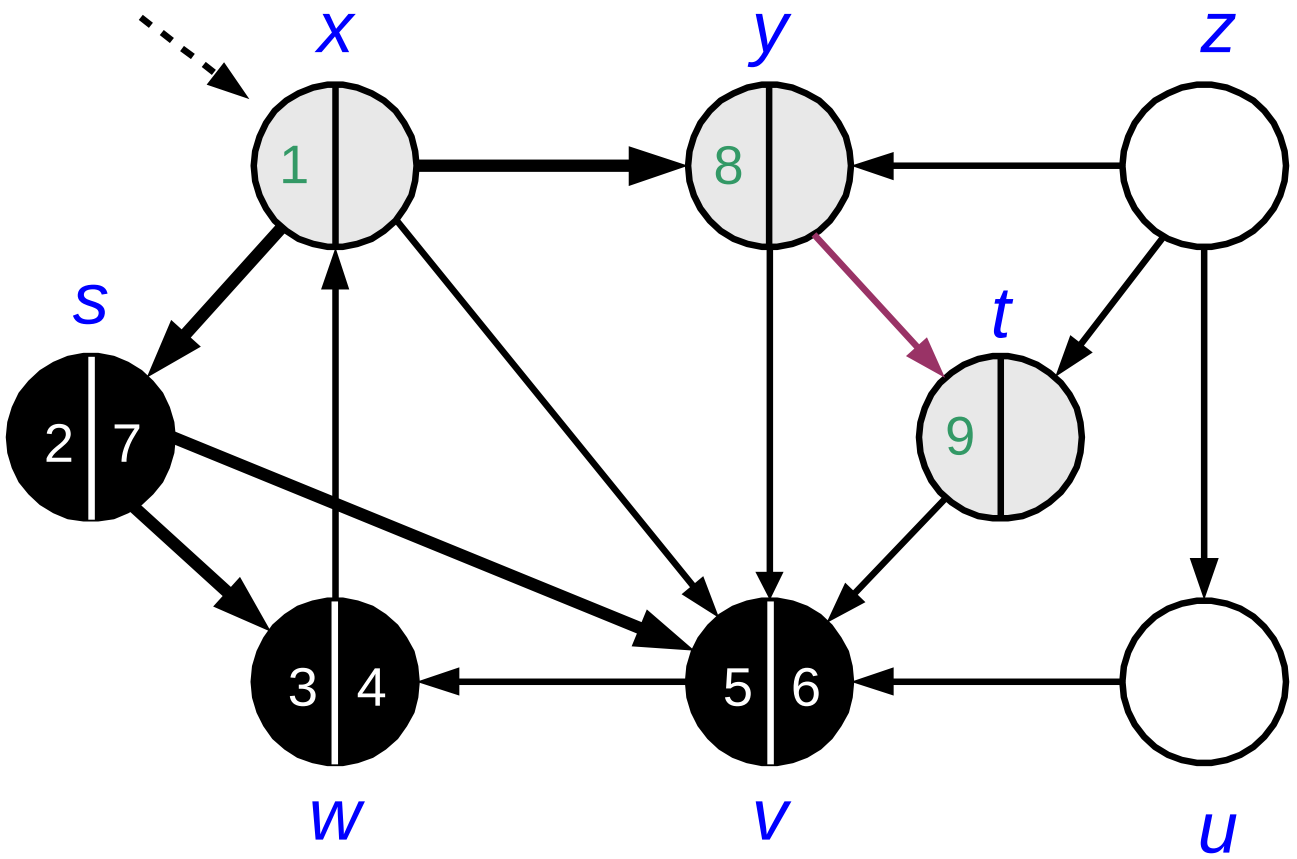
Depth-First Search: Example



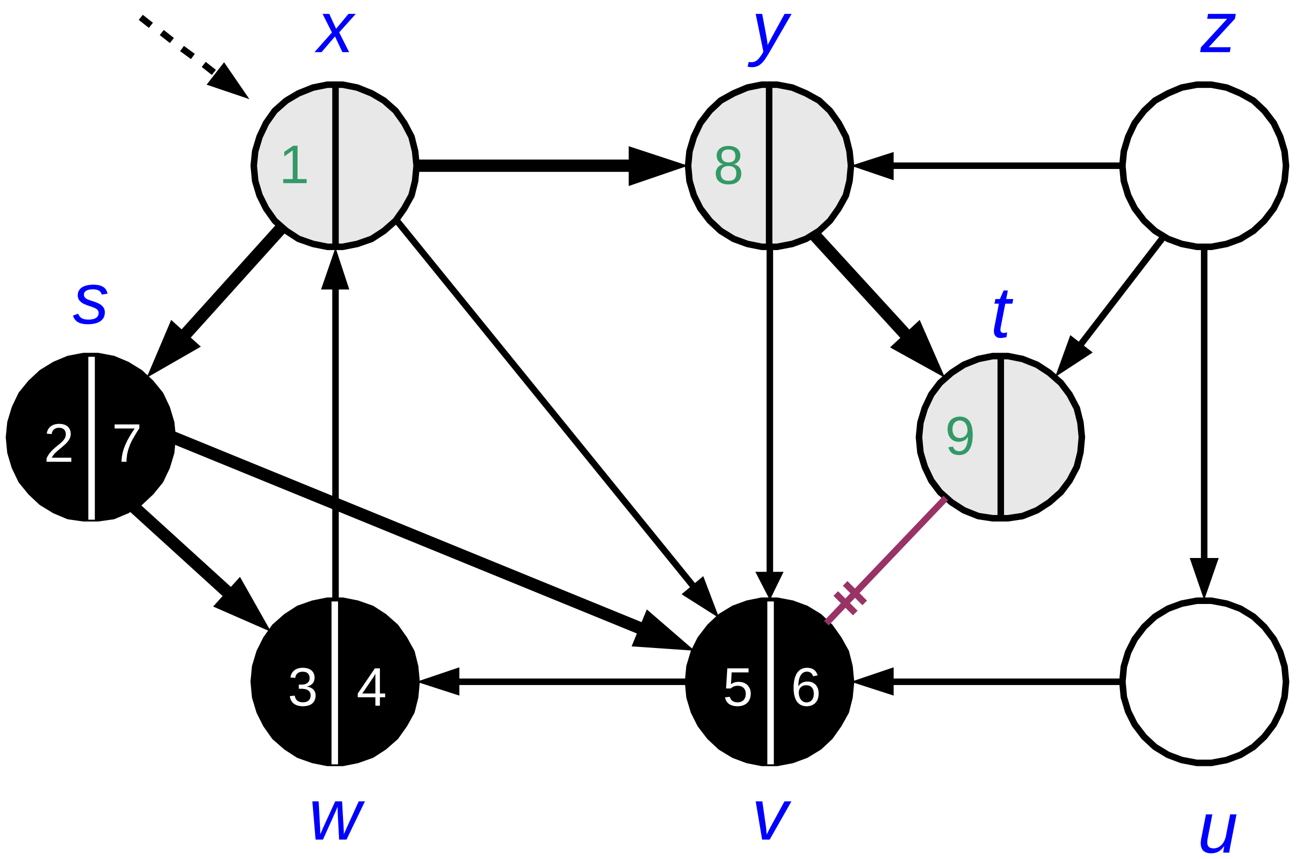
Depth-First Search: Example



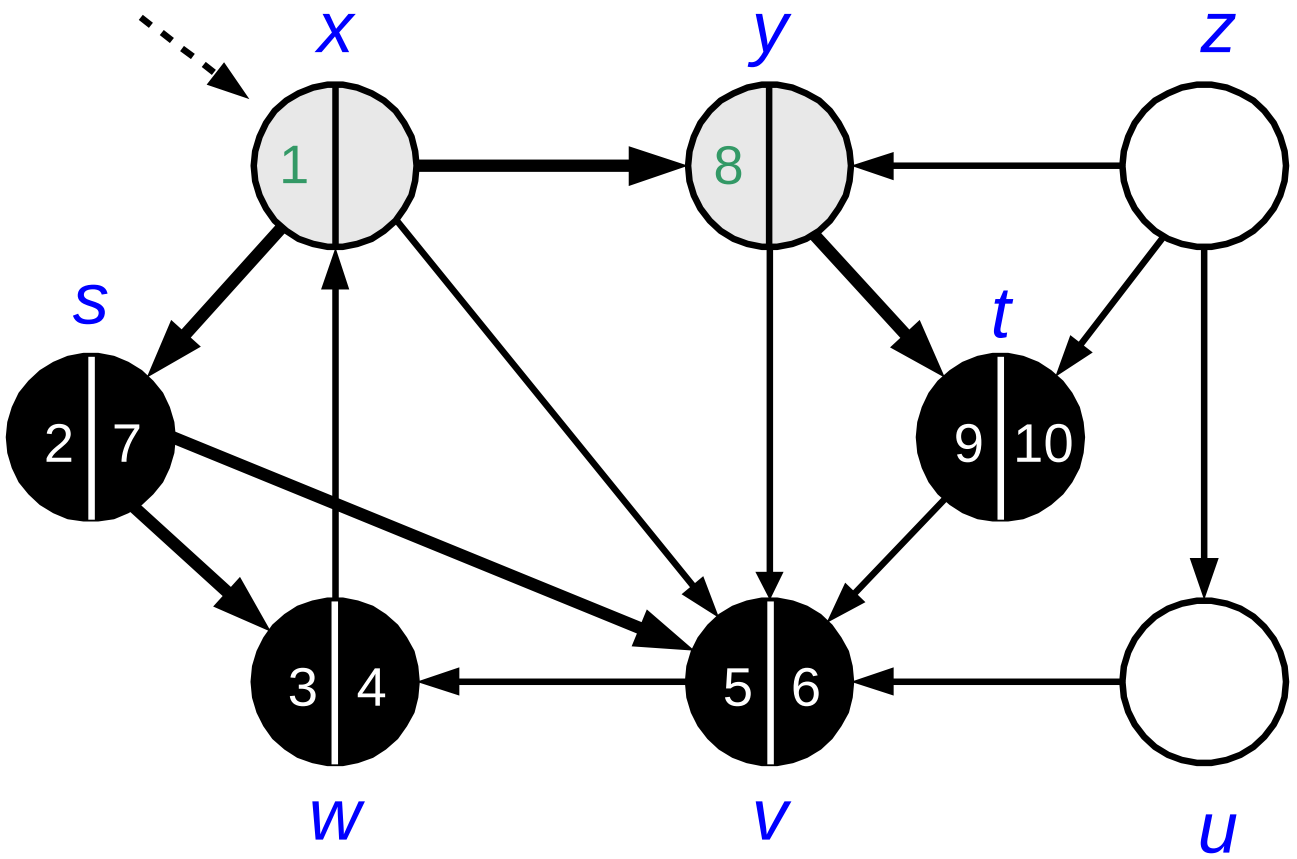
Depth-First Search: Example



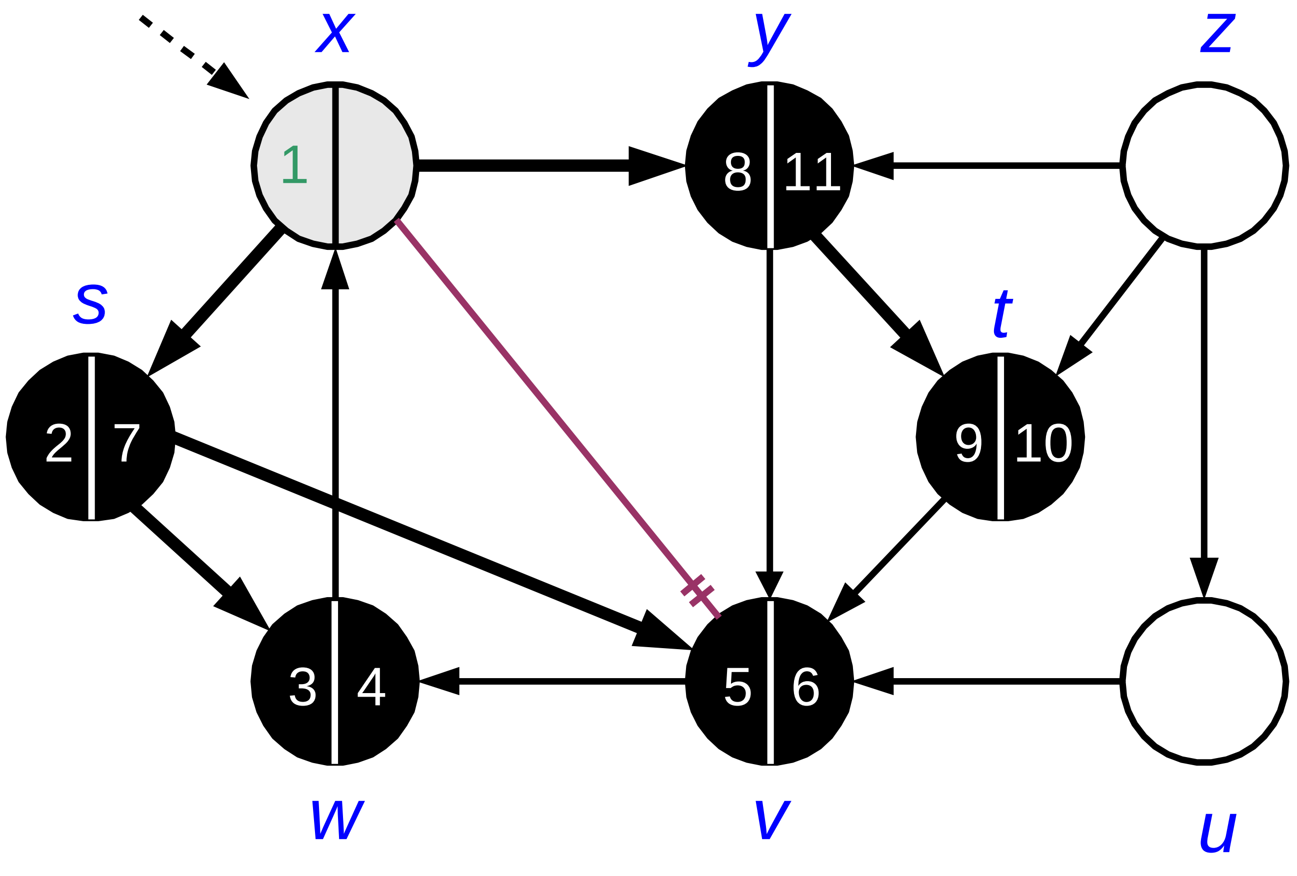
Depth-First Search: Example



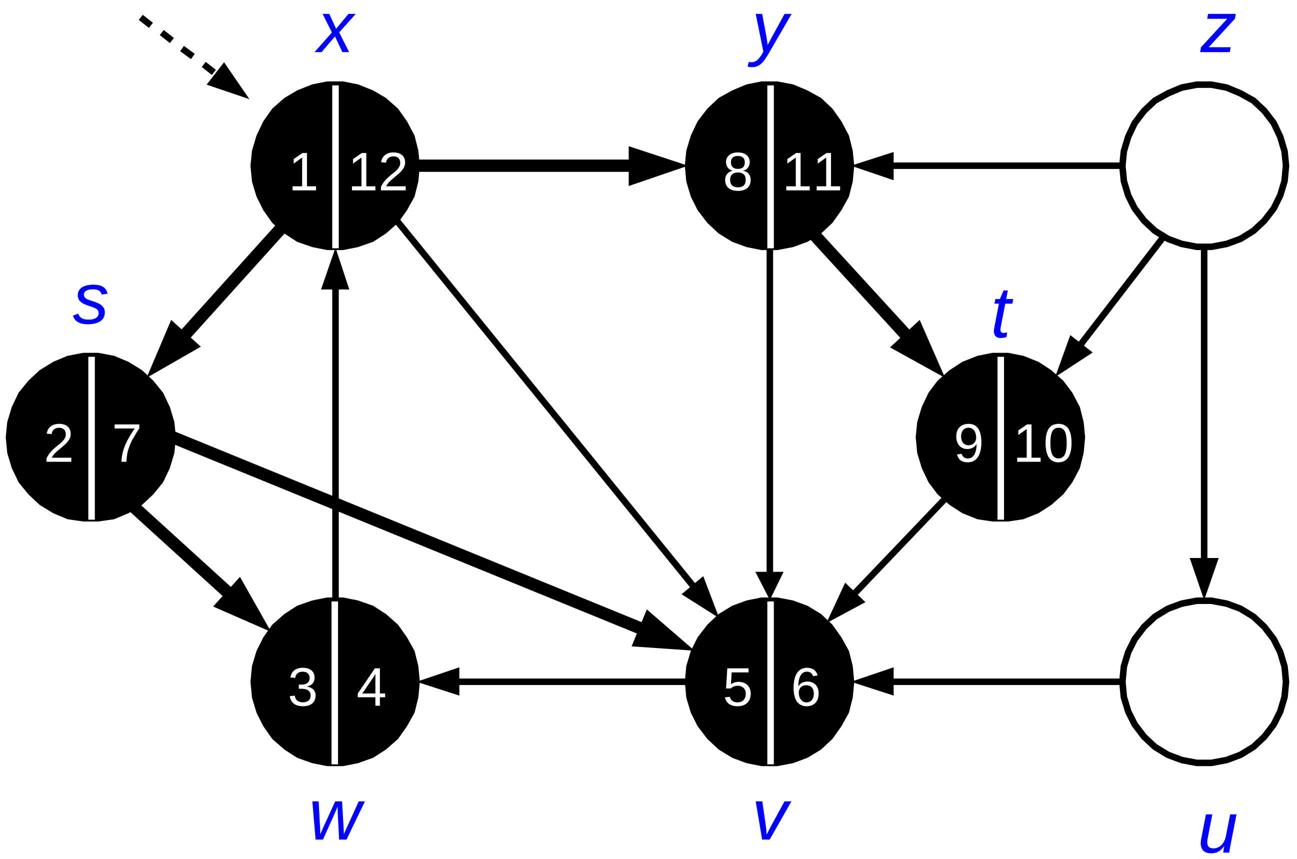
Depth-First Search: Example



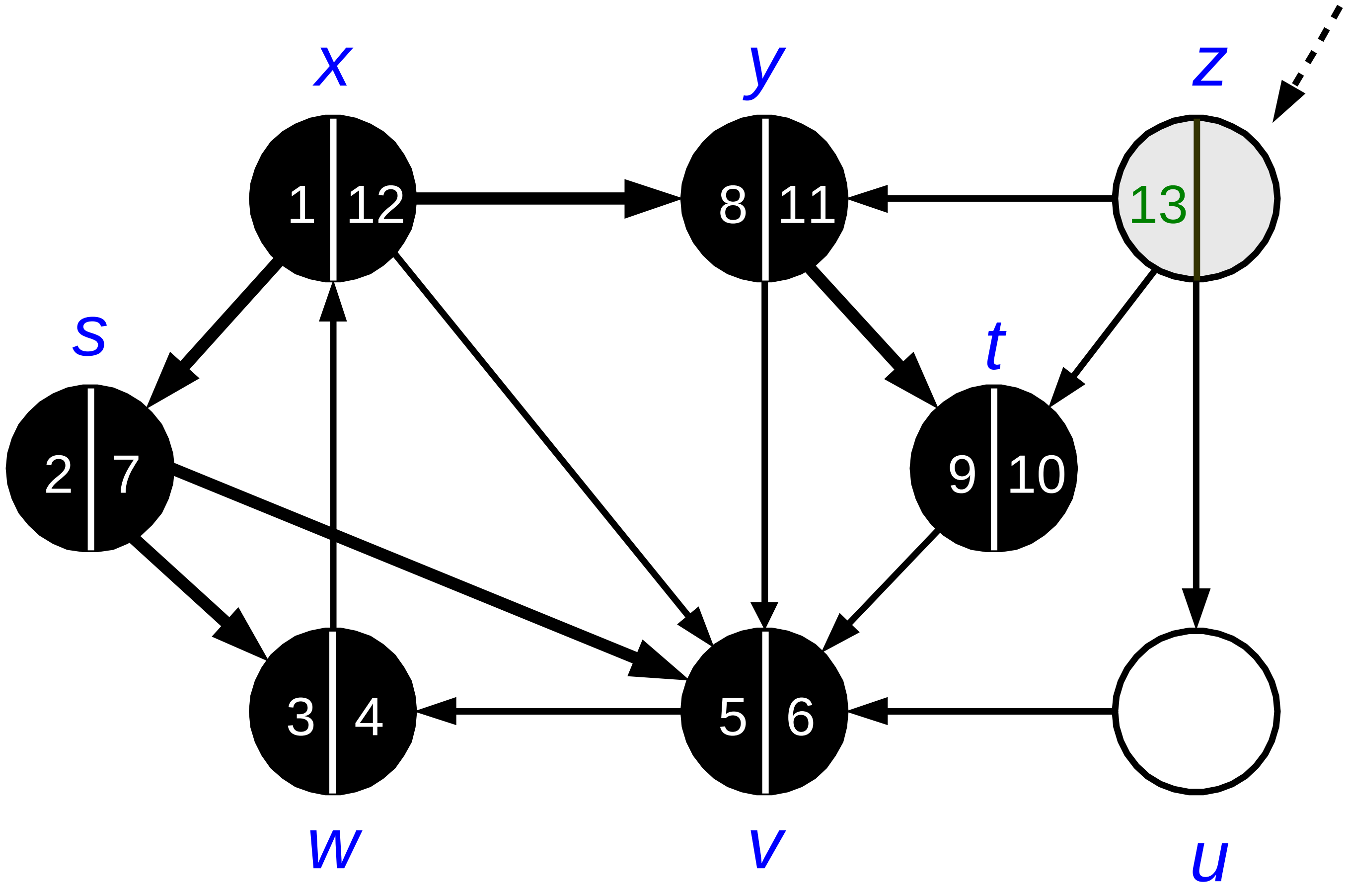
Depth-First Search: Example



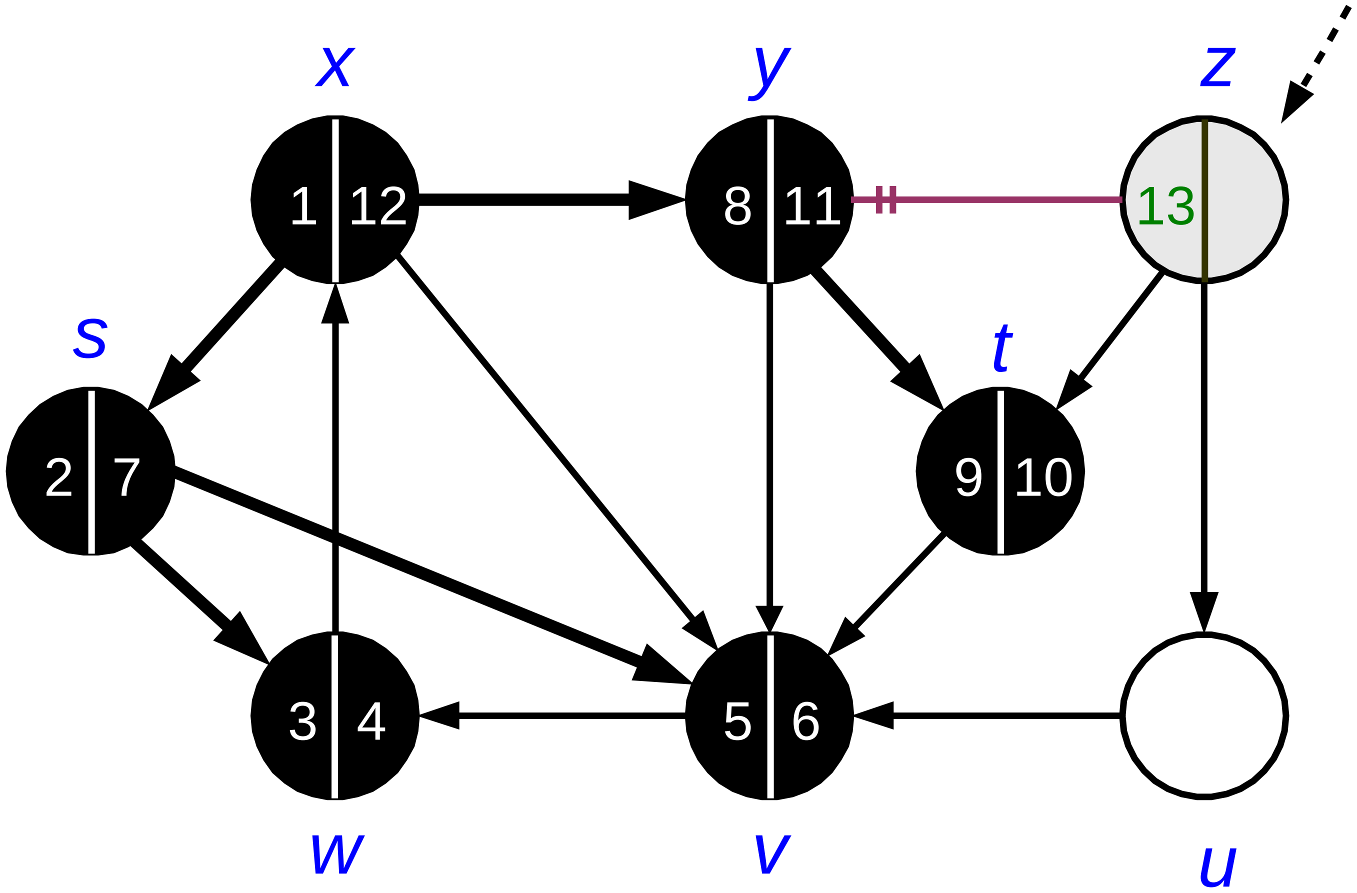
Depth-First Search: Example



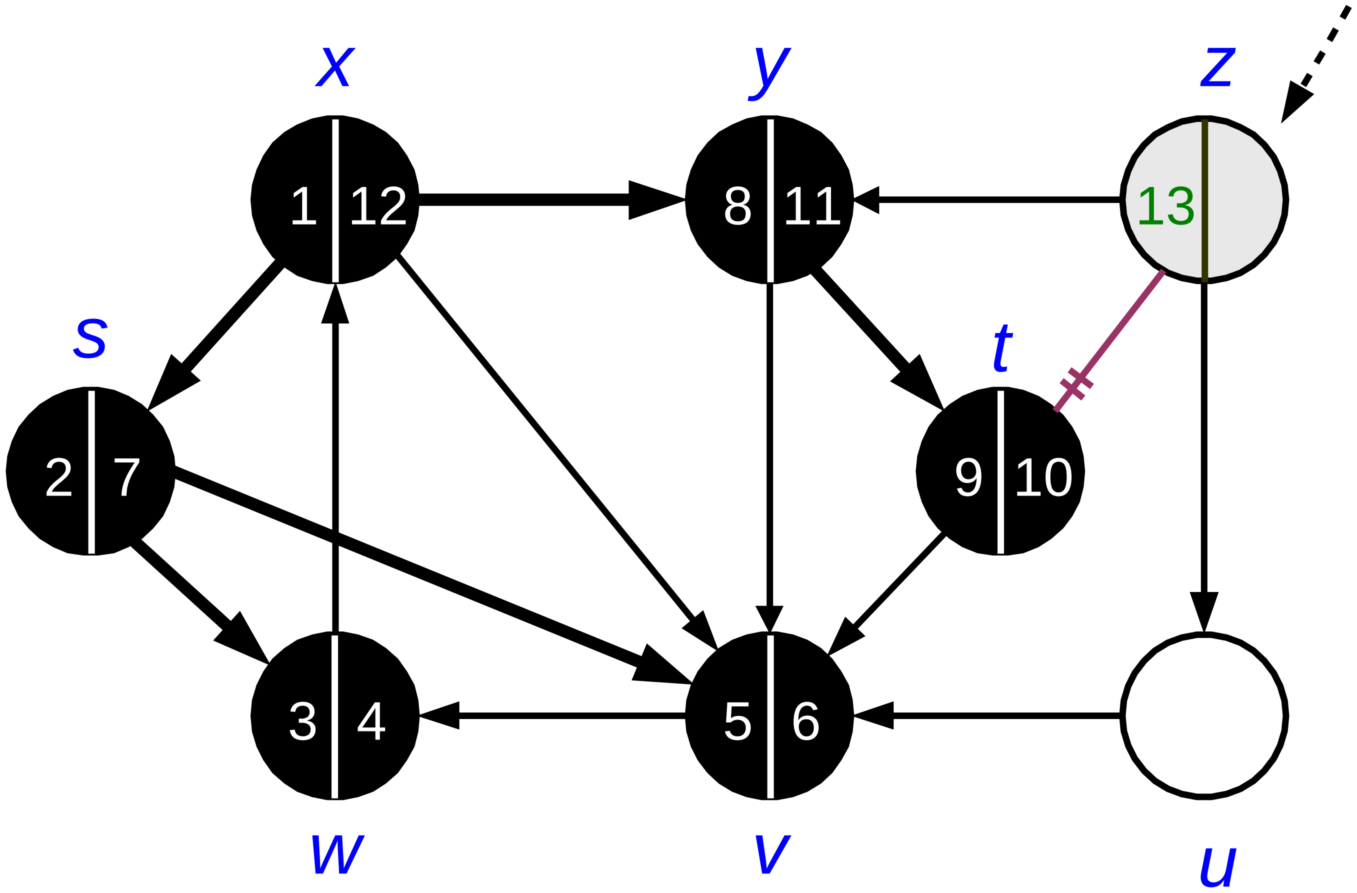
Depth-First Search: Example



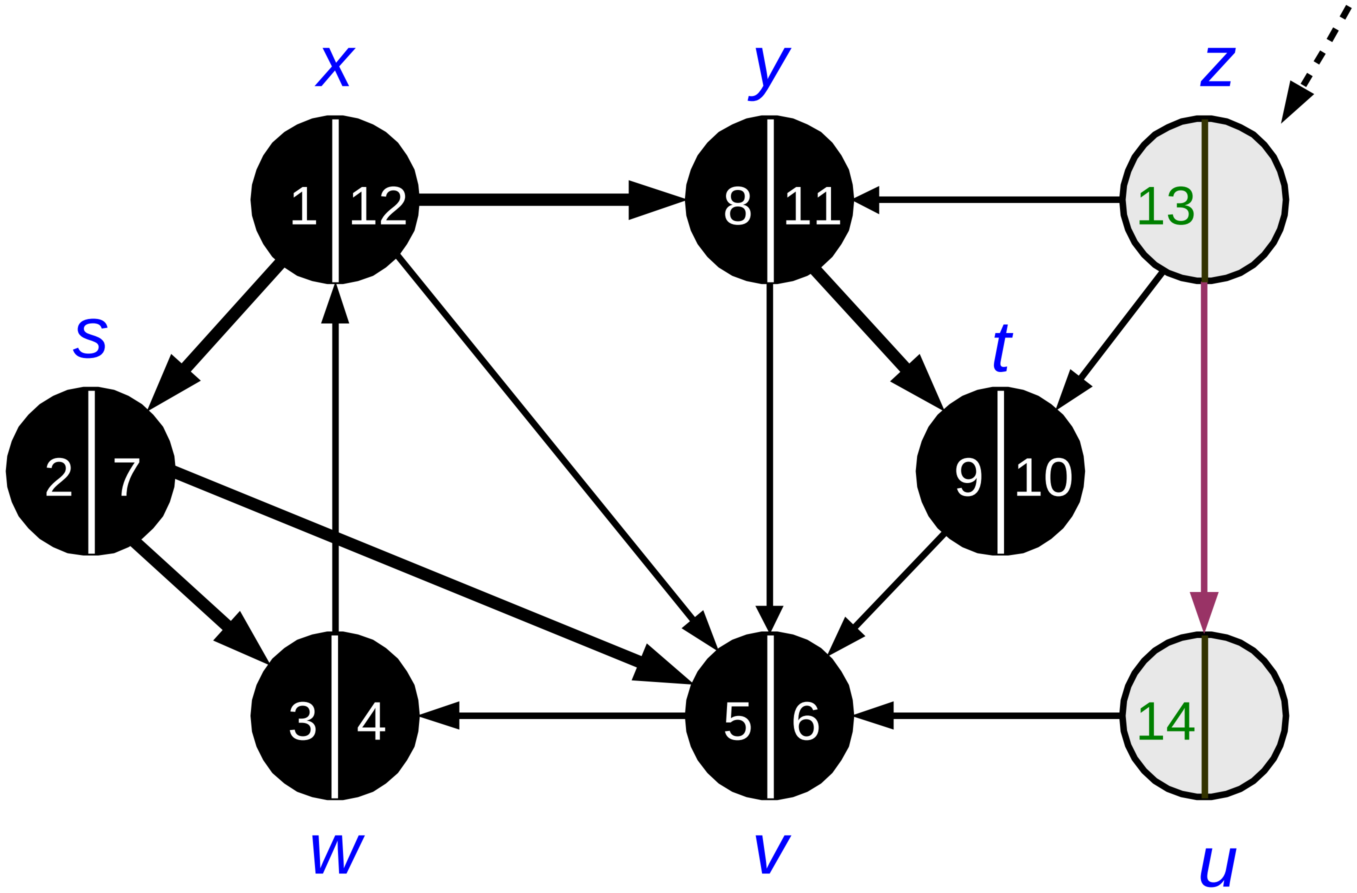
Depth-First Search: Example



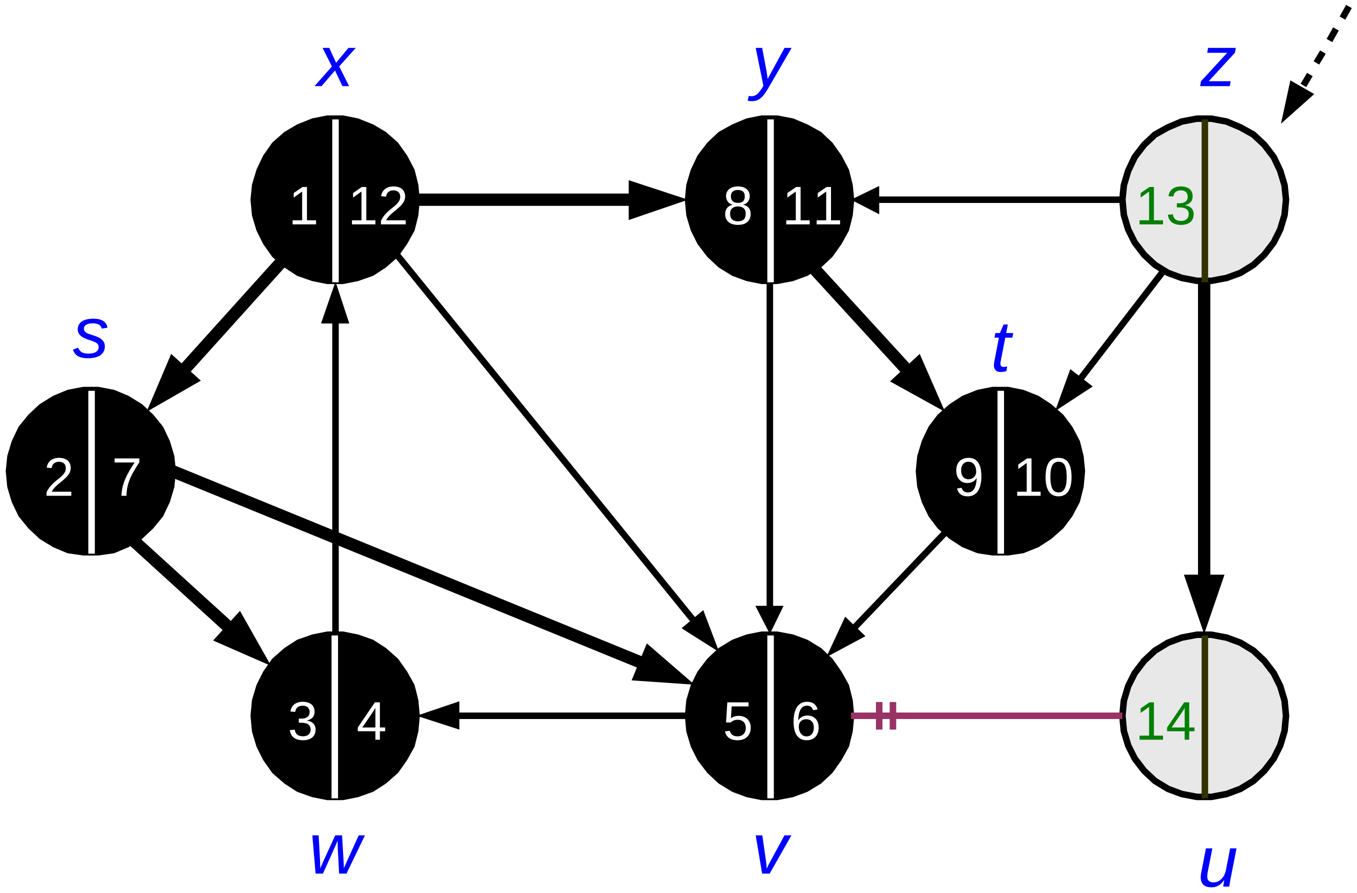
Depth-First Search: Example



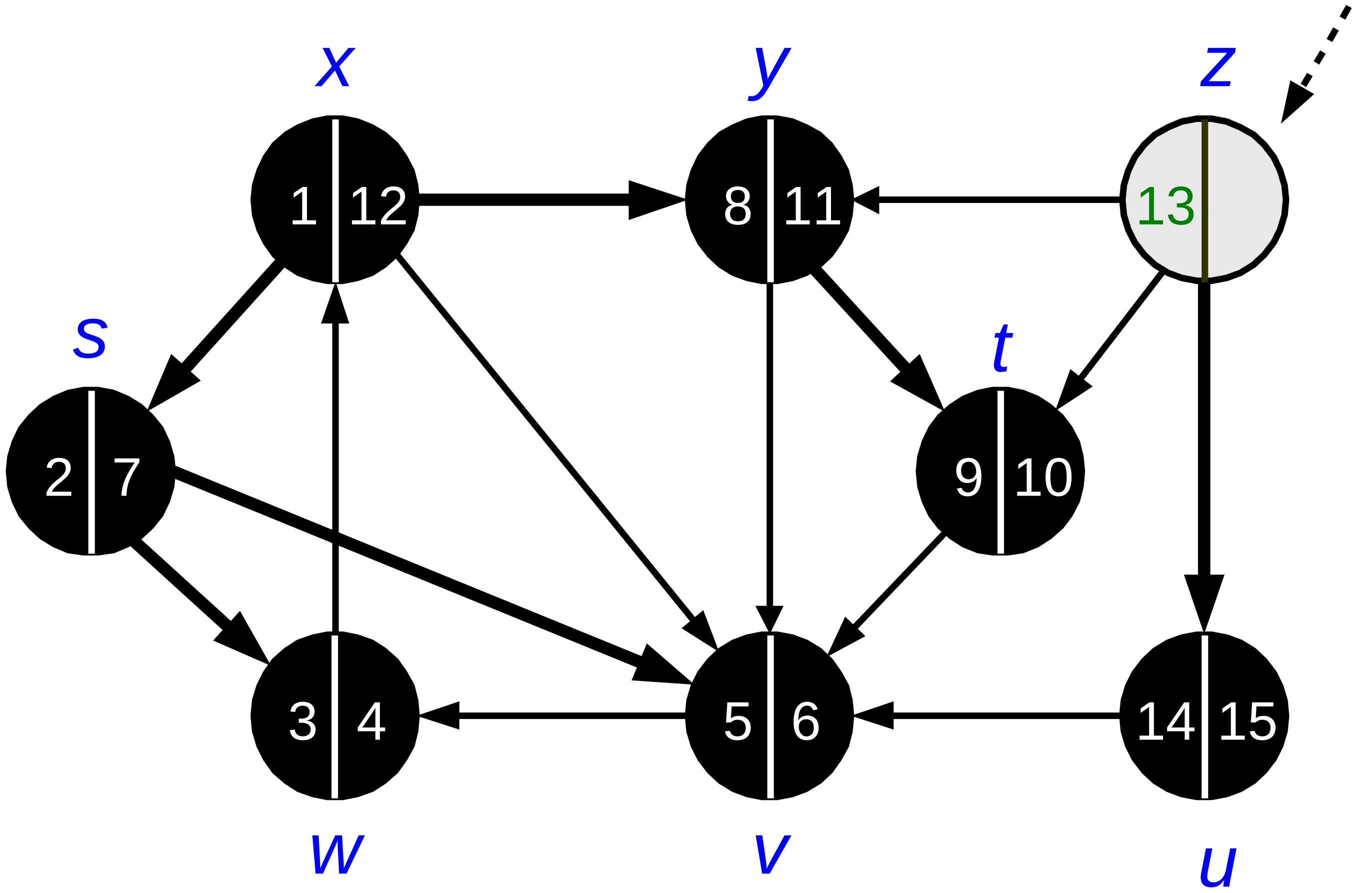
Depth-First Search: Example



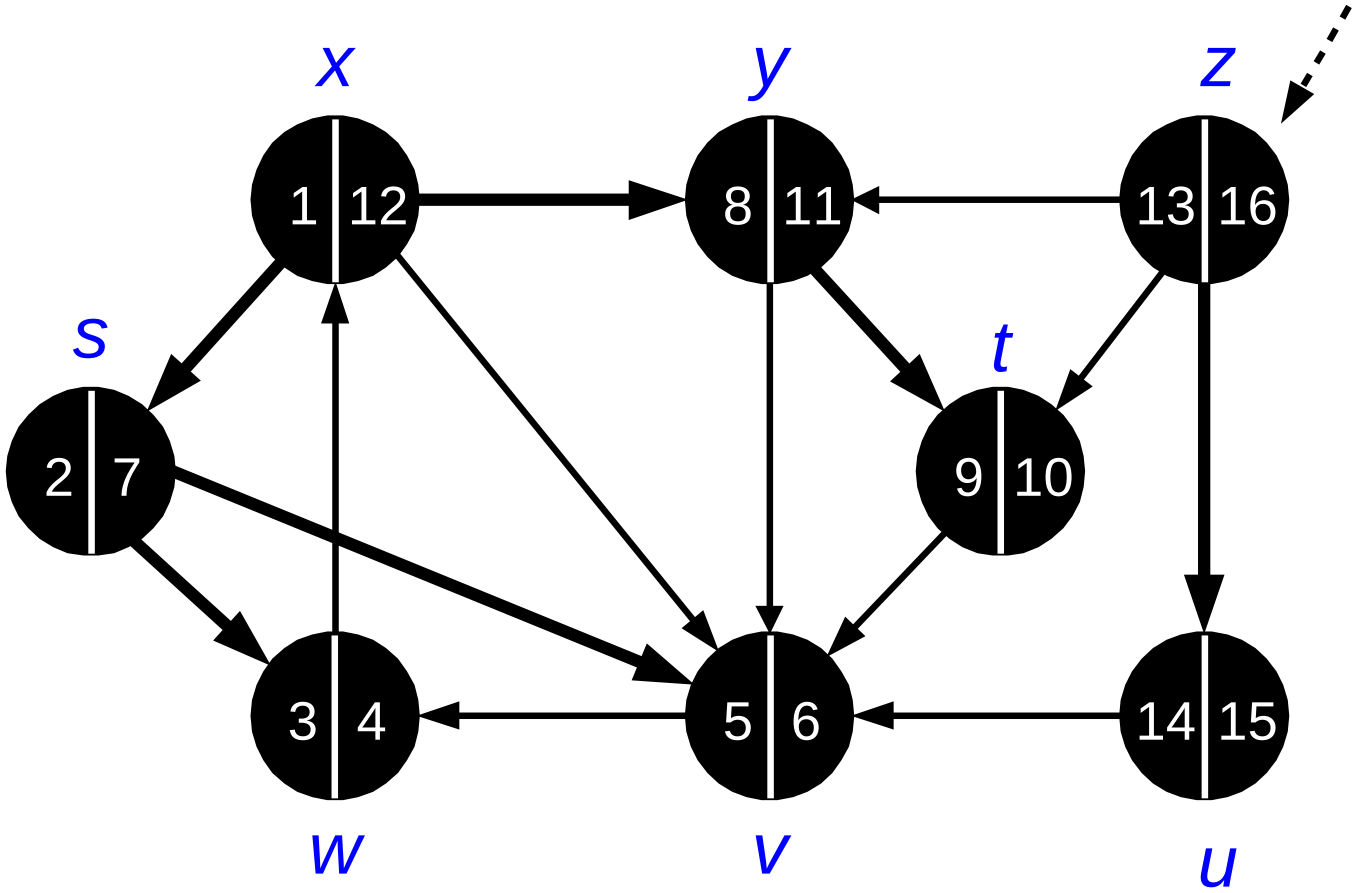
Depth-First Search: Example



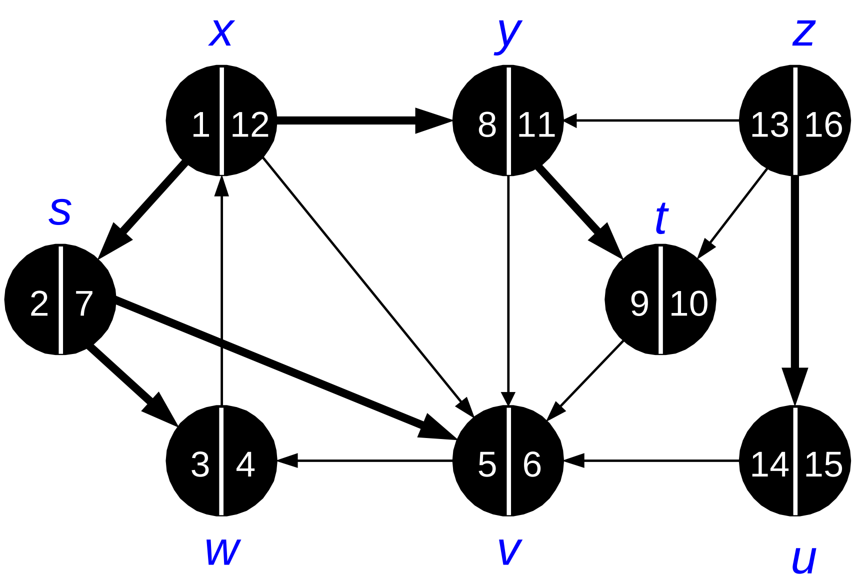
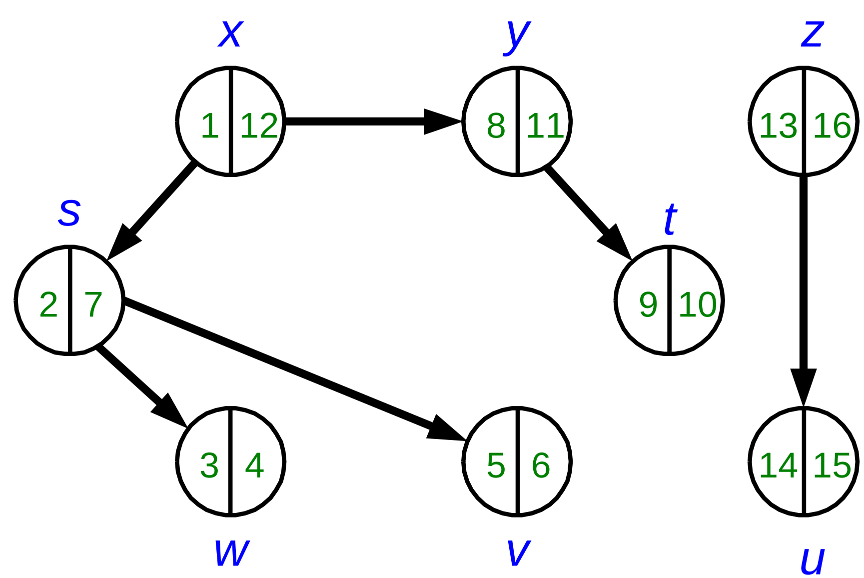
Depth-First Search: Example



Depth-First Search: Example



Depth-First Search: Example

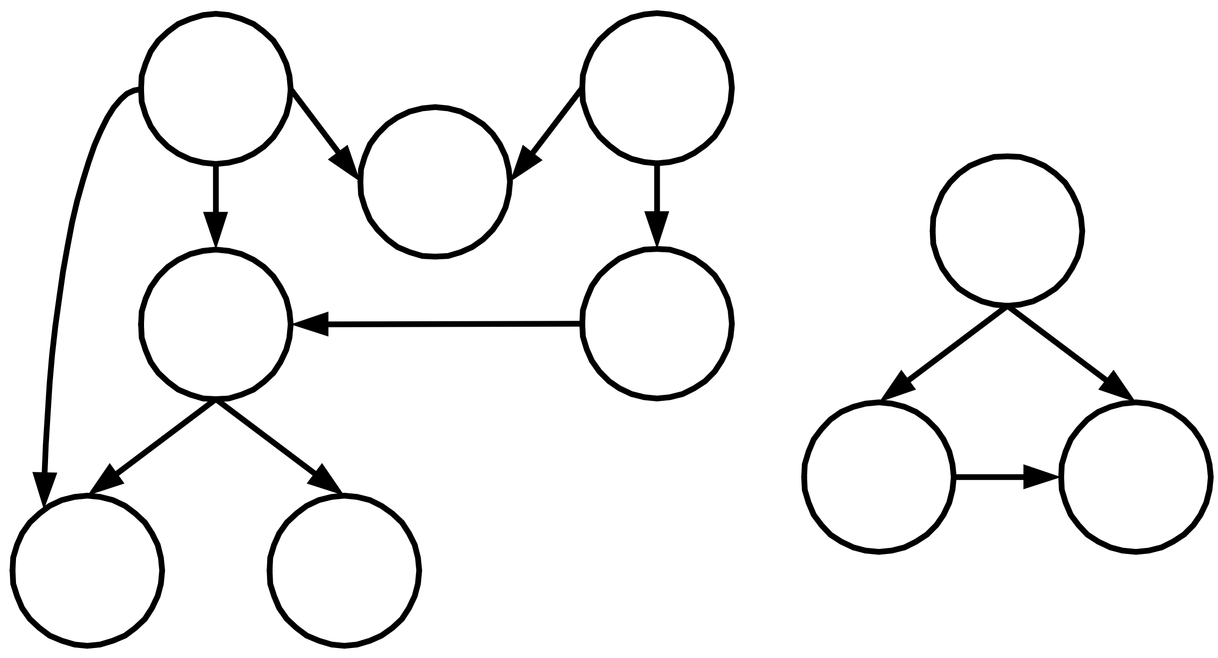
DFS(G) terminated	Depth-first forest (DFF)
	

Topological Sorting

Directed Acyclic Graphs (DAG)

No directed cycles

Example:



Directed Acyclic Graphs (DAG)

Theorem: a directed graph G is acyclic iff DFS on G yields no Back edges

Proof (acyclic $\Rightarrow$ no Back edges; by contradiction):

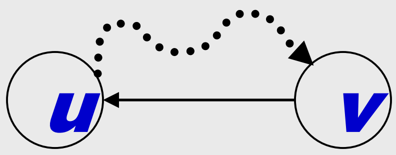
Let (v,u) be a Back edge visited during scanning Adj[v]

$\Rightarrow$ color[v] = color[u] = GRAY and d[u] < d[v]

$\Rightarrow$ int[v] is contained in int[u] $\Rightarrow$ v is descendent of u

$\Rightarrow$ $\exists$ a path from u to v in a DFT and hence in G

$\therefore$ edge (v,u) will create a cycle (Back edge $\Rightarrow$ cycle)



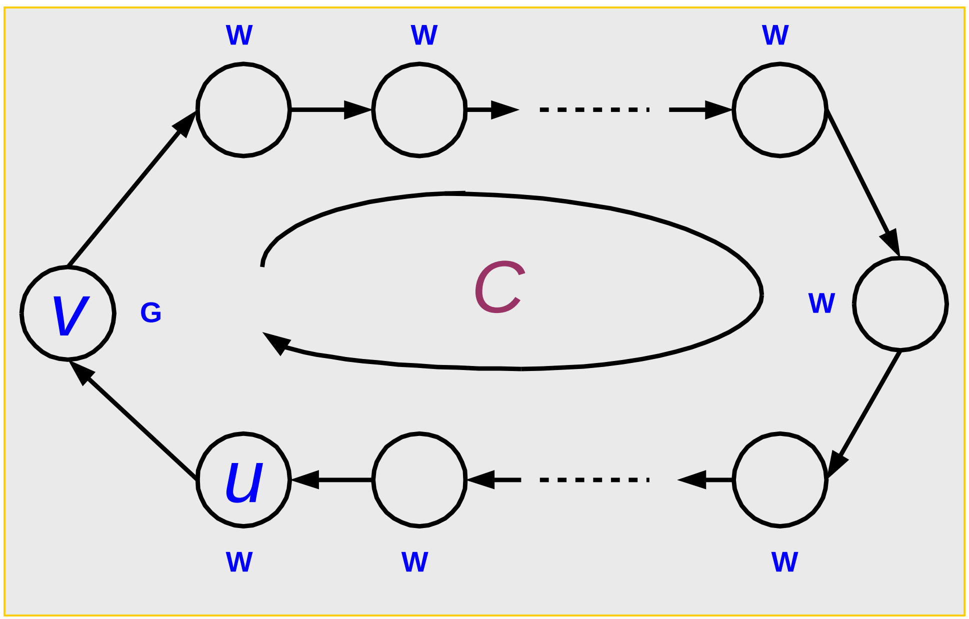
path from u to v in a DFT and hence in G

acyclic iff no Back edges

Proof (no Back edges $\Rightarrow$ acyclic):

Suppose G contains a cycle *C* (Show that a DFS on G yields a Back edge; proof by contradiction)

Let *v* be the first vertex discovered in *C* and let (*u*,*v*) be proceeding edge in *C*



At time d[*v*]: ∃ a white path from *v* to *u* along *C*

By White Path Thrm *u* becomes a descendent of *v* in a DFT

Therefore (*u*,*v*) is a Back edge (descendent to ancestor)

Topological Sort of a DAG

- Linear ordering '$<$' of V such that

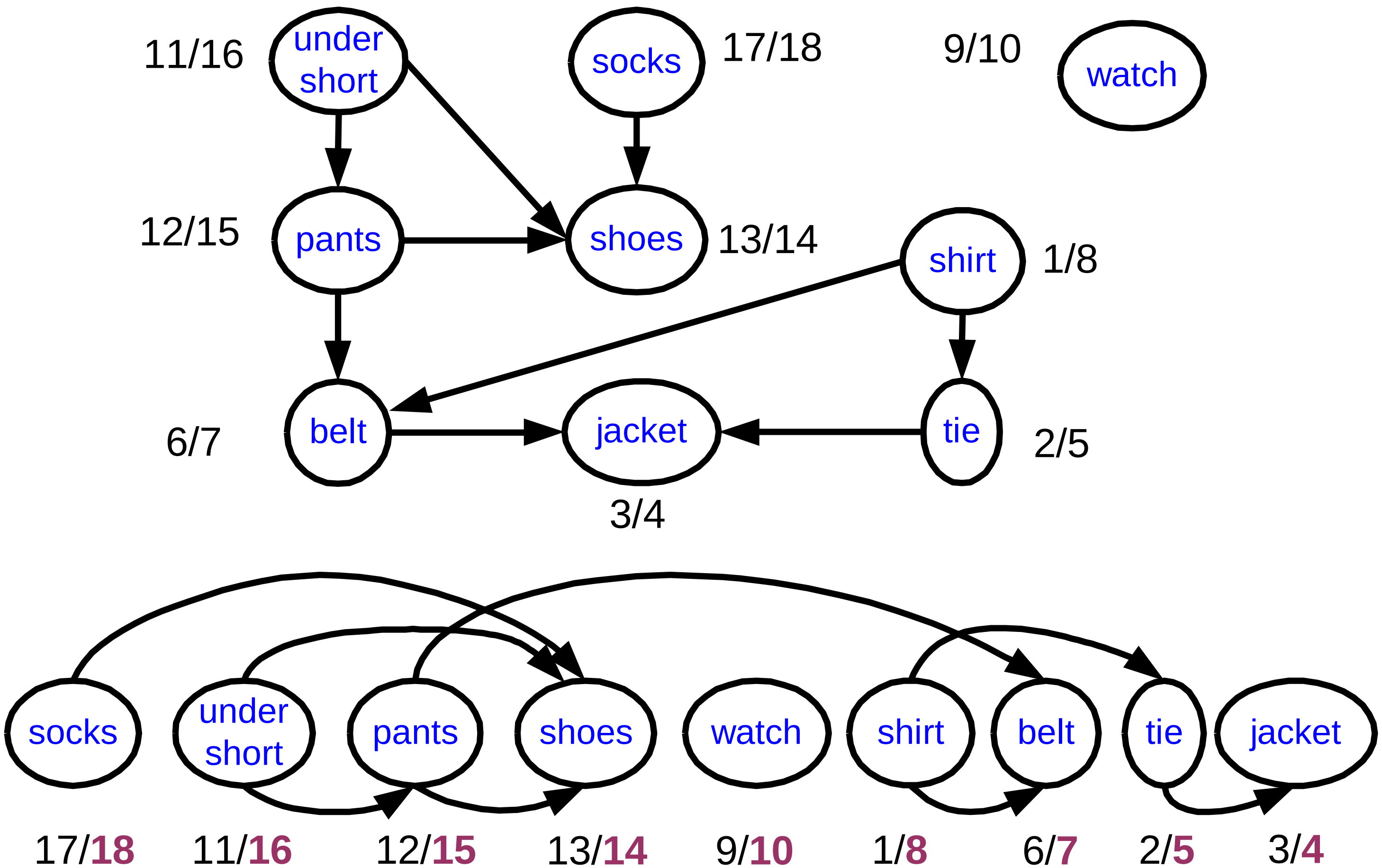
  $(u,v) \in E \Rightarrow u < v$ in ordering

  —Ordering may not be unique

  —i.e., mapping the partial ordering to total ordering may yield more than one orderings

Topological Sort of a DAG

Example: Getting dressed



Topological Sort of a DAG

Algorithm

run DFS(G)

when a vertex finished, output it

vertices output in reverse topologically sorted order

Runs in O(V+E) time

Topological Sort of a DAG

Correctness of the Algorithm

Claim: $(u,v) \in E \Rightarrow f[u] > f[v]$

Proof: consider any edge (u,v) explored by DFS

when (u,v) is explored, u is GRAY

- —if v is GRAY, (u,v) is a Back edge (contradicting acyclic theorem)
- —if v is WHITE, v becomes a descendent of u (b WPT) $\Rightarrow$ $f[v] < f[u]$
- —if v is BLACK, $f[v] < d[u] \Rightarrow f[v] < f[u]$

QED

Iterative Algorithms

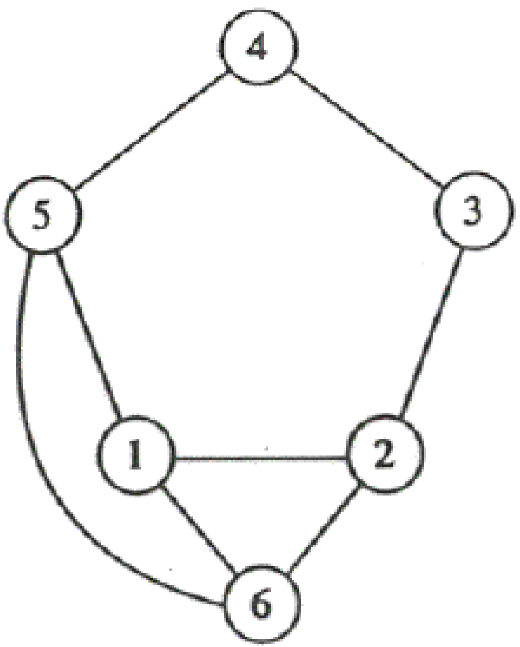
Iterative Algorithm – Vertex Coloring

- Vertex-color method using an iterative approach
  - Iterative and often constructive

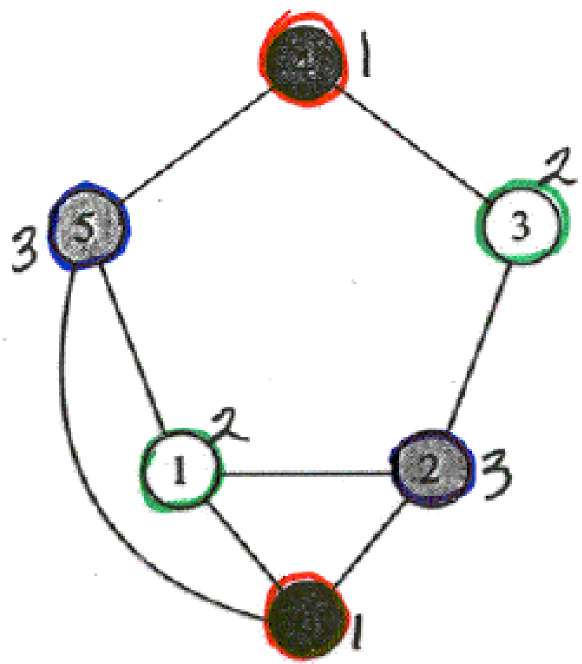
$VERTEX\_COLOR(G(V, E))$ {
    for ($i = 1$ *to* $|V|$) {
        $c = 1$;
        while ($\exists$ a vertex adjacent to v_i with color c) **do** {
            $c = c + 1$;
        }
        Label v_i with color c;
    }
}

Iterative Algorithm – Vertex Coloring (cont.)

- As there is no look-back correction or refinement, there is no guarantee of optimality.
- Example

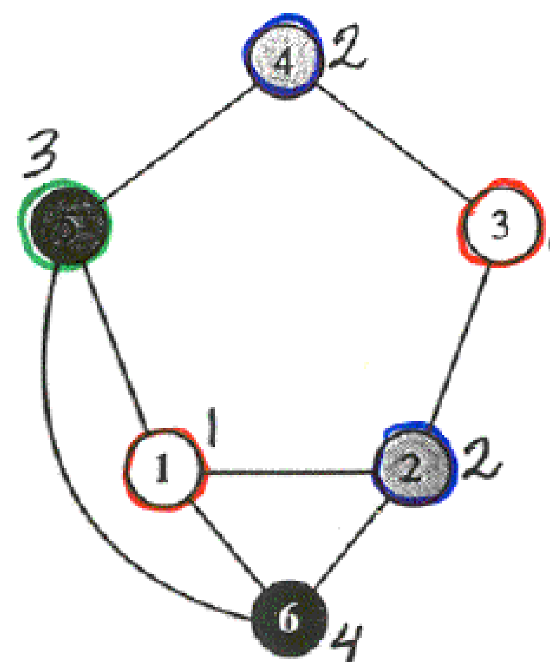


Graph



Order of coloring: v6→v1→v5→v4→v3→v2

Result: 2 colors



Order of coloring: v1→v2→v3→v4→v5→v6

Result: 4 colors

Iterative Algorithm – Vertex Covering

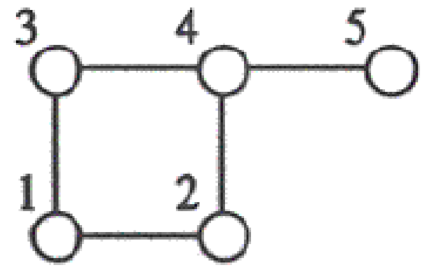
- Two versions using : 1) vertex and 2) edge to make decisions

$VERTEX\_COVER\_V(G(V,E))$ {
  $C = \emptyset$;
  while $(E \neq \emptyset)$ **do** {
    Select a vertex $v \in V$;
    Delete v from $G(V,E)$;
    $C = C \cup \{v\}$;
  }
}

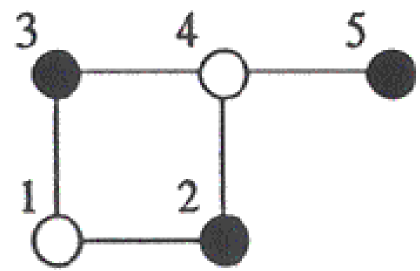
$VERTEX\_COVER\_E(G(V,E))$ {
  $C = \emptyset$;
  while $(E \neq \emptyset)$ **do** {
    Select an edge $\{u, v\} \in E$;
    $C = C \cup \{u\} \cup \{v\}$;
    Delete u and v from $G(V,E)$;
  }
}

Iterative Algorithm – Vertex Covering (cont.)

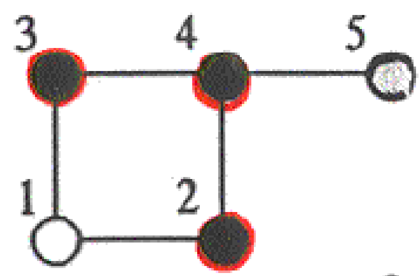
- Example: Graph



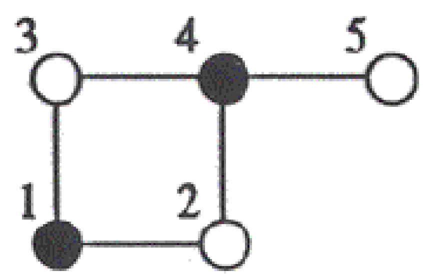
- Apply Vertex_Cover_V (v5→v2→v3)



- Apply Vertex_Cover_E ({v2,v4}→…)



- Best solution (not obtained)



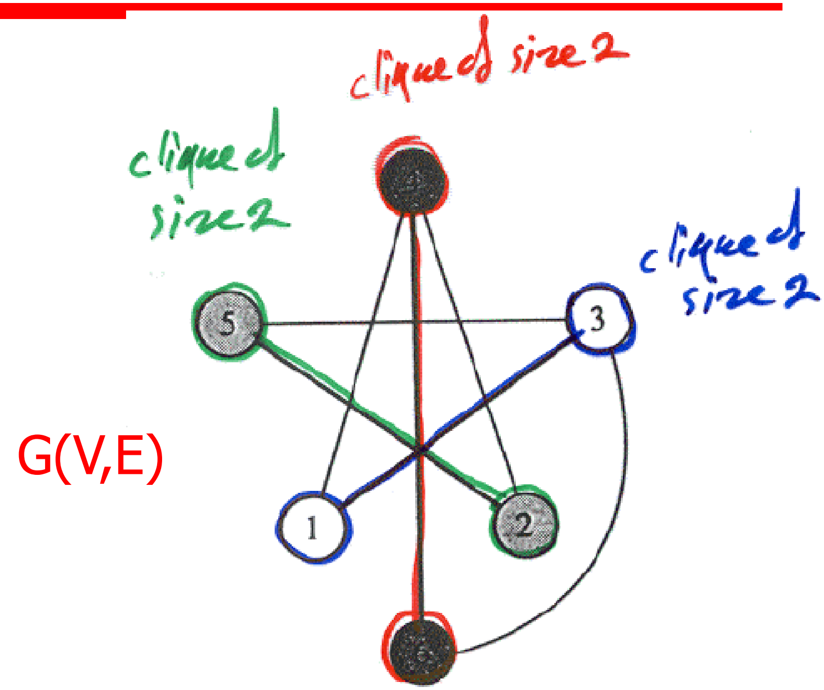
Role of Heuristics – Clique Partitioning

- Some guidelines (heuristics) are needed to make decisions/selections in the process.
- The optimality is not often guaranteed.

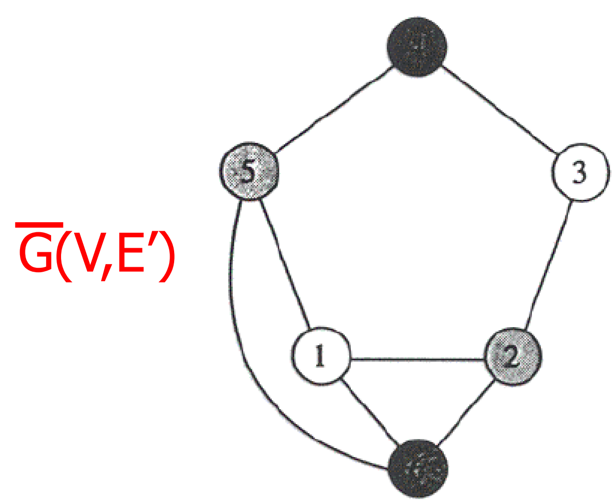
```
CLIQUE_PARTITION(G(V, E)) {
    Π = ∅;
    while (G(V, E) not empty ) do {
        C = MAX_CLIQUE(G(V, E));
        Π = Π ∪ C;
        Delete C from G(V, E);
    }
}
MAX_CLIQUE(G(V, E)) {
    C = vertex with largest degree;
    repeat {
        U = {v ∈ V : v ∉ C and adjacent to all vertices of C};
        if (U = ∅)
            return(C);
        else {
            Select vertex v ∈ U;    → heuristic?!
            C = C ∪ {v};
        }
    }
}
```

Role of Heuristics – Clique Partitioning (cont.)

- Example
  - Clique partitioning of G is the same as coloring G'



- Order of vertex consideration in the process
  - C=v4→ Clique={v4,v6}
  - C=v3→ Clique={v3,v1}
  - C=v5→ Clique={v5,v2}



Greedy Algorithms

Greedy Algorithms

- Greedy algorithms make **good local choices** in the hope that they result in an optimal solution.
  - They result in feasible solutions.
  - **Not** necessarily an optimal solution.

- A proof is needed to show that the algorithm finds an optimal solution.

- A counter example shows that the greedy algorithm does not provide an optimal solution.

Left-Edge Algorithm

- This is a greedy (and constructive) method
- Used in VLSI channel routing, register minimization etc.

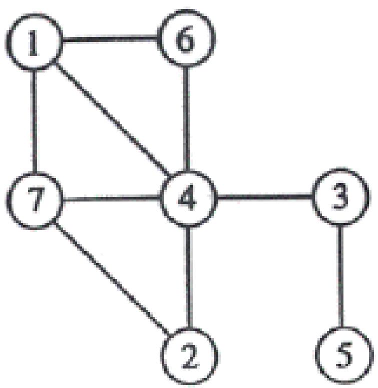
```
LEFT_EDGE(I) {
    Sort elements of I in a list L in ascending order of l_i;
    c = 0;
    while (some interval has not been colored ) do {
        S = ∅;
        r = 0;                      /* initialize coordinate of rightmost edge in S */
        while ( ∃ an element in L whose left edge coordinate is larger than r) do{
            s = First element in the list L with l_s ⩾ r;
            S = S ∪ {s};
            r = r_s;                /* update coordinate of rightmost edge in S */
            Delete s from L;
        }
        c = c + 1;
        Label elements of S with color c;
    }
}
```

Left-Edge Algorithm - Example

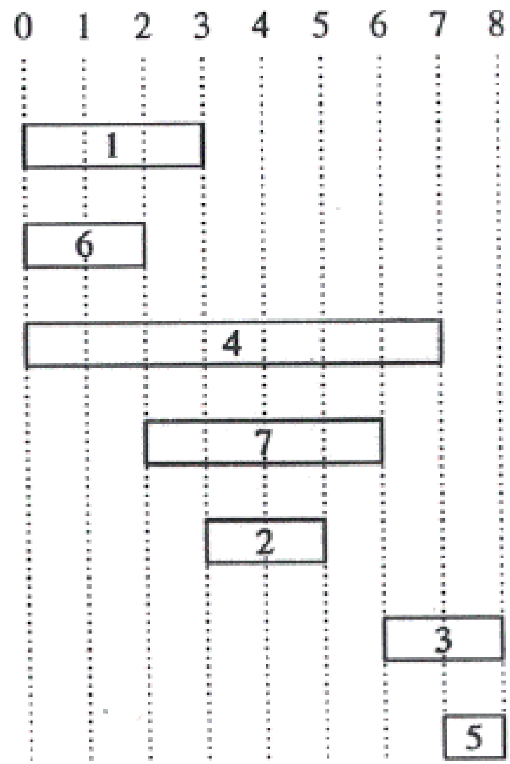
- Data given (e.g. life spans of signals and we are looking for minimum number of registers)

Vertex	Left edge (ℓ)	Right edge (r)
v_1	0	3
v_2	3	5
v_3	6	8
v_4	0	7
v_5	7	8
v_6	0	2
v_7	2	6

Left & right edges



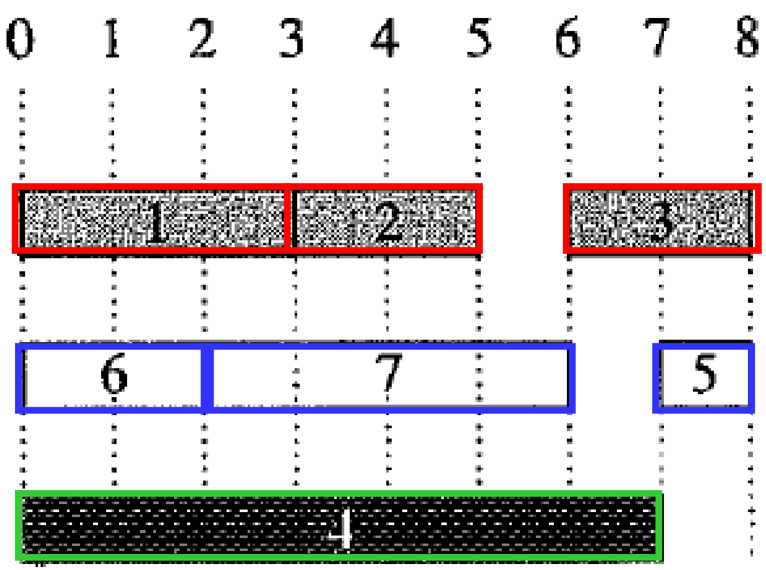
Conflict Graph



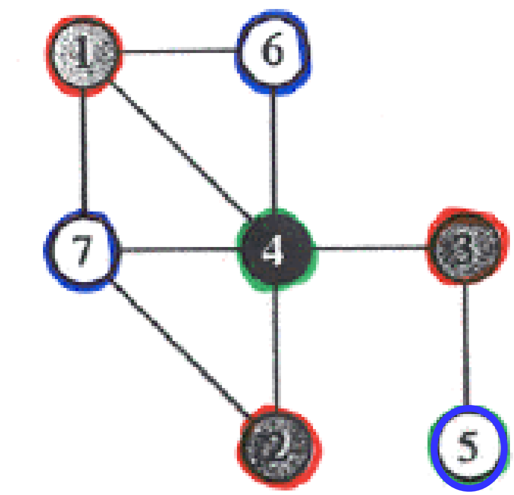
Ordered list (based on left edge values)

Left-Edge Algorithm – Example (cont.)

- Order of decisions: {v1→v6→v4→v7→v2→v3→v5}



Final results (3 channels/ registers are needed)



Graph coloring achieves the same

Unsuccessful Termination in Greedy Methods

- If termination heuristic (e.g. to find a non-optimal solution) is not devised well, it may not produce any result while it exists.
- Example: Task Scheduling

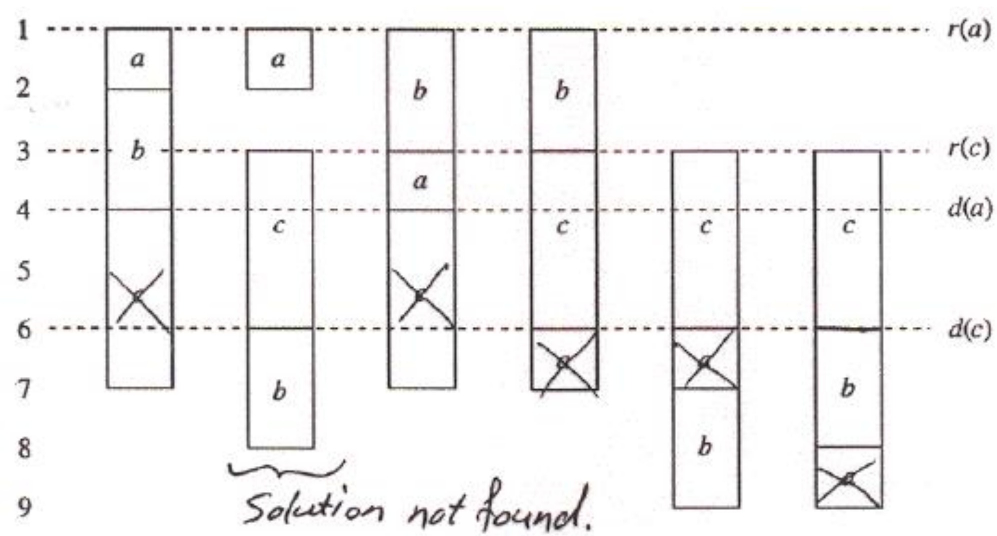
```
GREEDY_SCHEDULING(T) {
    i = 1;
    repeat {
        while ((Q = {unscheduled tasks with release time < i}) == ∅)  do
            i = i + 1;
        if (∃ an unscheduled task p : i + l(p) > d(p)) return(FALSE);
        Select q ∈ Q with smallest deadline;
        Schedule q at time i;
        i = i + l(q);
    } until (all tasks scheduled);
    return(TRUE);
}
```

unsuccessful termination

Task Scheduling Algorithm

- Iteration 1:
  - —i=1, Q={a,b}
  - —Choose "a" since d(a)=4<d(b)=infinity.
  - —i=i+l(a)=2
- Iteration 2:
  - —i=2, Q={b}
  - —Choose "b"
  - —i=i+l(b)=4
- Iteration 3:
  - —i=4, Q={c}
  - —4+l(c)=7> d(c)=6
  - —Terminate unsuccessfully.

Tasks	Release r	Deadline d	Length ℓ
a	1	4	1
b	1	∞	2
c	3	6	3



Dynamic Programming Algorithms

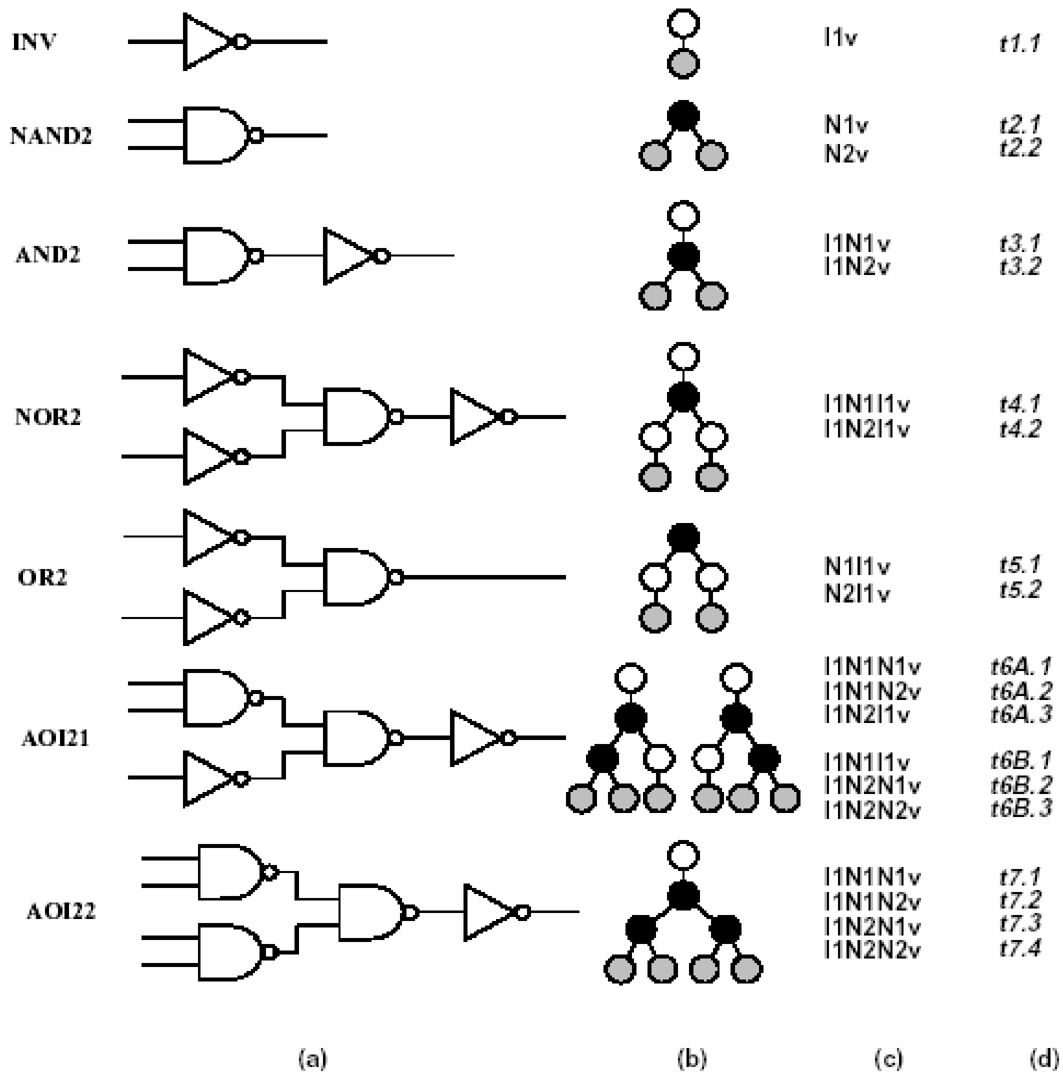
Dynamic Programming

- An algorithm that finds the optimum solution to a problem involving *N* objects in terms of the solutions to a series of smaller problems that involve subsets of those objects.

- Often we break down the problem to smaller one of the same nature and solve it recursively.

- Example: Tree-Based Covering Algorithm

Simple Library



Tree-Based Matching

u: the root of the pattern tree

v: the vertex of the subject tree

$MATCH(u, v)$ {
  if (u is a leaf) **return** (TRUE); /* Leaf of the pattern graph reached */
  else {
    if (v is a leaf) **return** (FALSE); /* Leaf of the subject graph reached */
    if ($degree(v) \neq degree(u)$) **return**(FALSE); /* Degree mismatch */
    if ($degree(v) == 1$) { /* One child each: visit subtree recursively */
      u_c = child of u ; v_c = child of v ;
      return ($match(u_c, v_c)$)
    }
    else { /* Two children each: visit subtrees recursively */
      u_l = left-child of u ; u_r = right-child of u ;
      v_l = left-child of v ; v_r = right-child of v ;
      return ($MATCH(u_l, v_l) \cdot MATCH(u_r, v_r) + MATCH(u_r, v_l) \cdot MATCH(u_l, v_r)$);
    }
  }
}

Tree-Based Covering

- Dynamic programming
  - Visit subject tree bottom-up.
- At each vertex attempt to match
  - Locally rooted subtree.
  - Check all library cells for a match.
- Optimum solution for the subtree.

TREE_COVER$(T(V, E))$ {

  Set the cost of the internal vertices to -1;

  Set the cost of the leaf vertices to 0;

  while (some vertex has negative weight) **do** {

    Select a vertex $v \in V$ whose children have all nonnegative cost;

    M = set of all matching pattern trees at vertex v;

    $$\text{cost}(v) = \min_{m \in M(v)} \left(cost(m) + \sum_{u \in L(m)} cost(u) \right);$$

  }

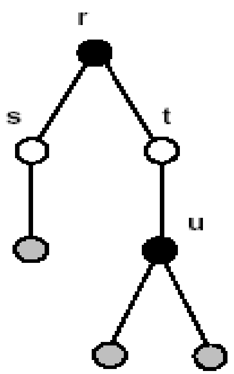
}

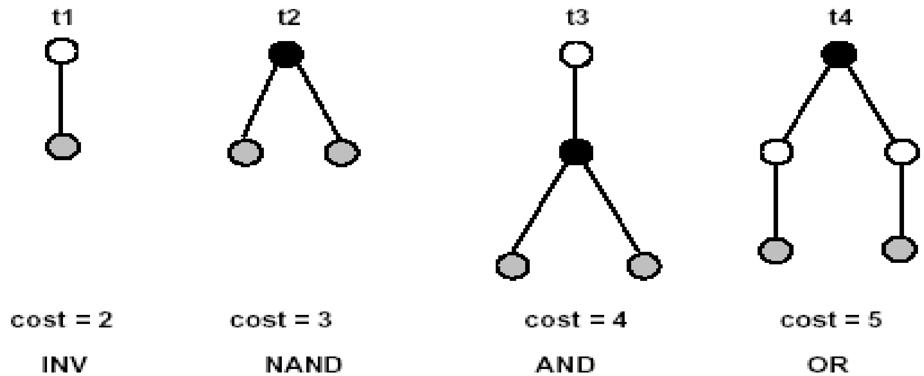
Vertices of a subject tree corresponding to the leaves of a matching pattern tree

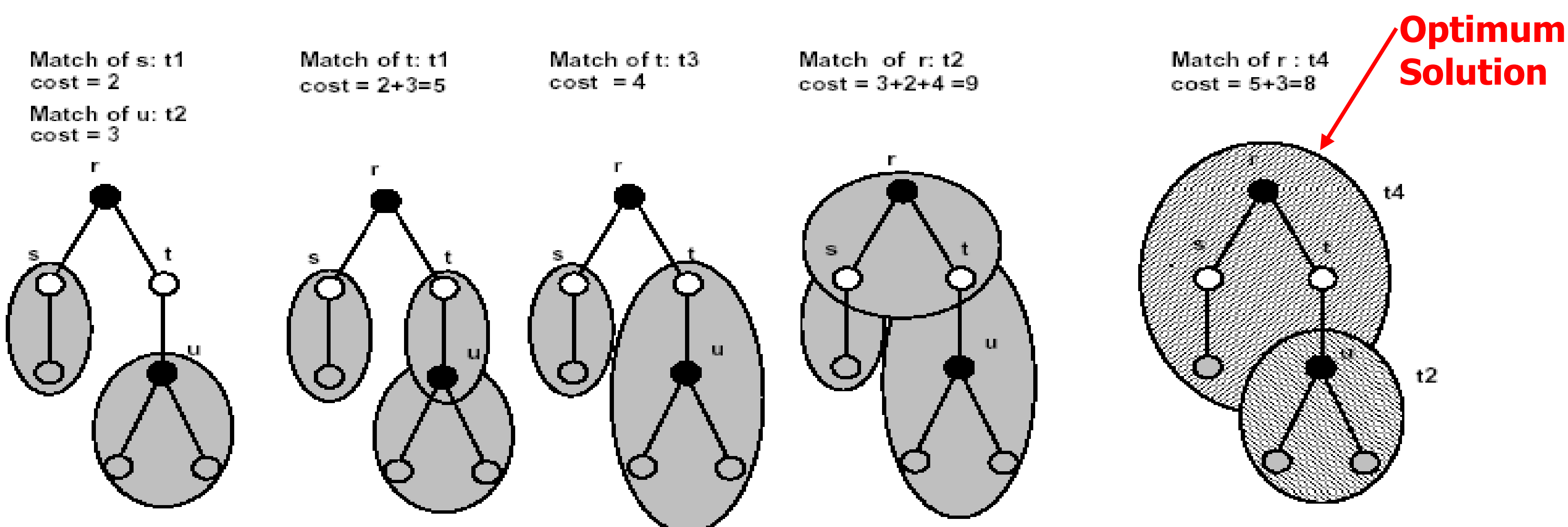
Example

SUBJECT TREE

PATTERN TREES

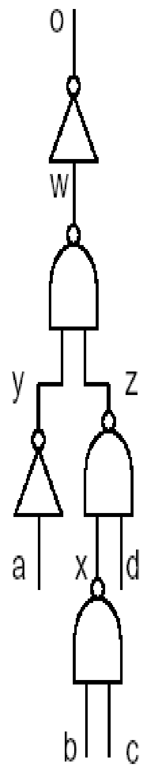
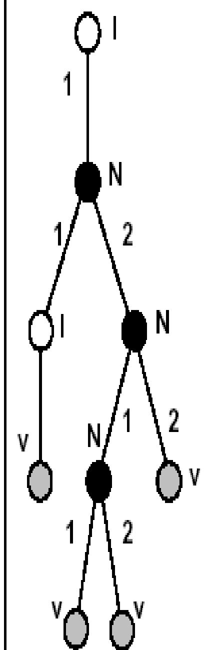






Minimum Area Cover Example

- Minimum-area cover.
- Area costs
  - INV:2; NAND2:3; AND2:4; AOI21:6.
- Best choice
  - AOI21 fed by a NAND2 gate.

Network	Subject graph	Vertex	Match	Gate	Cost
		x	t2	NAND2(b,c)	3
		y	t1	INV(a)	2
		z	t2	NAND2(x,d)	2* 3 = 6
		w	t2	NAND2(y,z)	3 * 3 + 2 = 11
		o	t1	INV(w)	3 * 3 + 2 * 2 = 13
			t3	AND2(y,z)	2 * 3 + 4 + 2 = 12
			t6B	AOI21(x,d,a)	3 + 6 = $\boxed{9}$

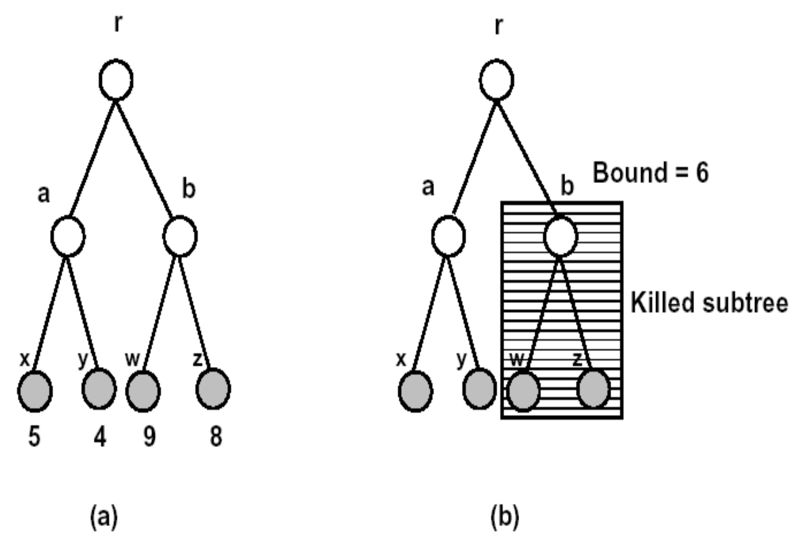
Branch & Bound Algorithms

Branch and Bound Algorithm

- Devise a branch selection (decision tree)
- Define a bounding function
  - Need to be fast for evaluations of many subtrees
- Evaluate the lower bound cost for subtrees
- Prune the subtree whose cost is higher than the existing solution found so far (space reduction)

Branch and Bound Algorithm (cont.)

- Tree search of the solution space
  - Potentially exponential search.
- For each branch, a lower bound is computed for all solutions in subtree.
- Use bounding function
  - If the lower bound on the solution cost that can be derived from a set of future choices exceeds the cost of the best solution seen so far
    - **Kill the search**.
- Good pruning may reduce run-time.



Branch and Bound Algorithm (cont.)

```
BRANCH AND BOUND {
    Current best = anything; Current cost = ∝ ; S = s0;
    while (S ≠ 0) do {
        Select an element  s ∈ S; Remove s from S ;
        Make a branching decision based on s yielding sequences {s_i, i = 1, 2, … , m};
        for ( i = 1 to m) {
                Compute the lower bound b_i of s_i;
                if (b_i ≥ Current cost) Kill s_i;
                else  {
                        if (s_i is a complete solution )&(cost of s_i < Current cost) {
                                Current best = s_i; Current cost = cost of s_i ;
                        } else if (s_i is not a complete solution ) Add s_i to set S;
                }
        }
    }
}
```

- *S denotes a solution or group of solutions with a subset of decisions made*
- *s0 denotes the sequence of zero length corresponding to initial state with no decisions made*

Shortest Path Algorithms

Shortest Path Algorithms

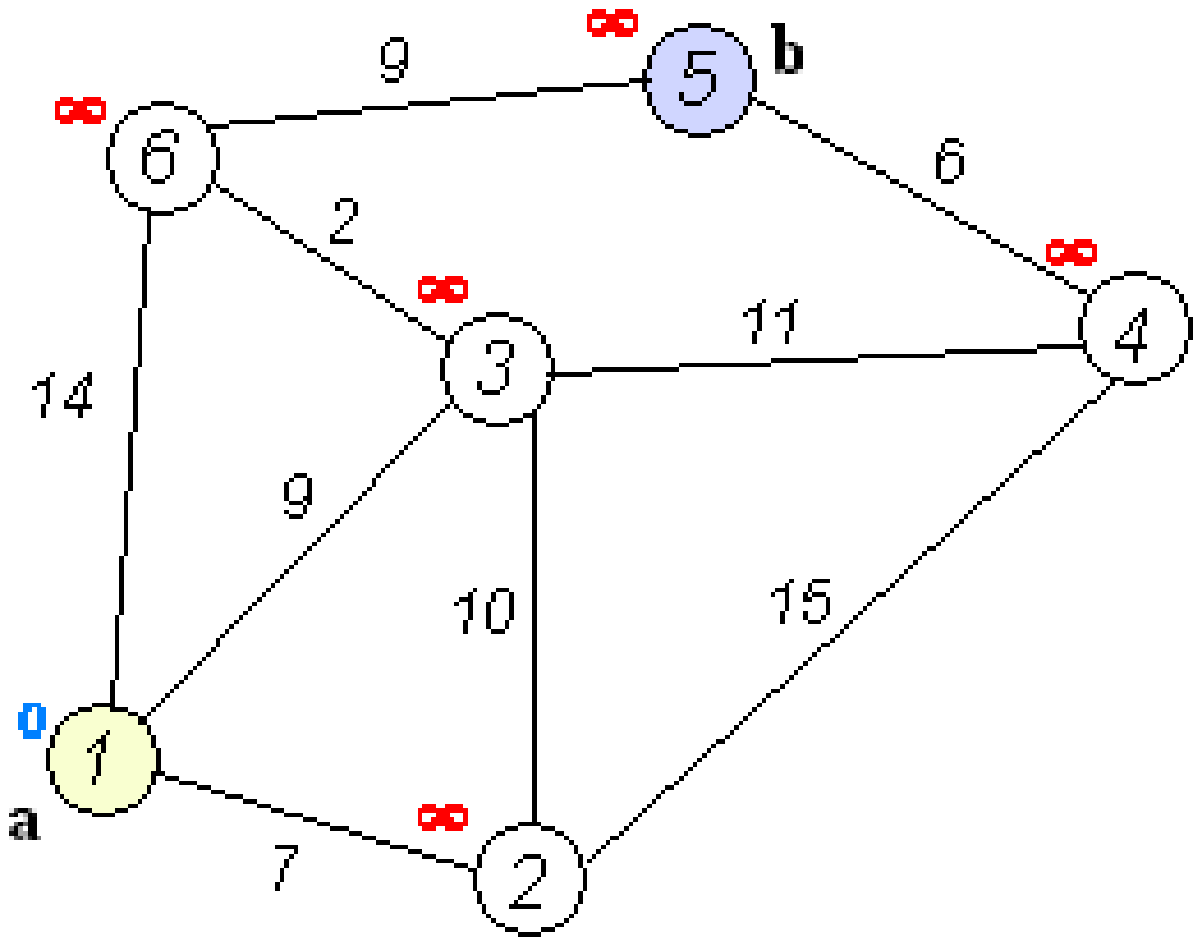
- Finds the critical path on weighted graphs with weights as delays
- Applied on *directed, weighted* graphs
- Different algorithms for different graphs
  - DAG shortest path algorithm: on DAGs
  - Dijkstra's Algorithm: no negative weights
  - The Bellman-Ford Algorithm: most general
- All 3 algorithms share the same kernel
  - initialization and relaxation

Dijkstra's Algorithm

- works on graphs with *non-negative* weights
- needs a priority queue with est(*v*) as keys
- extracts the node *u* with minimum est(*u*) in *minQ* and relaxes all edges incident from *u* to all nodes still in *minQ*

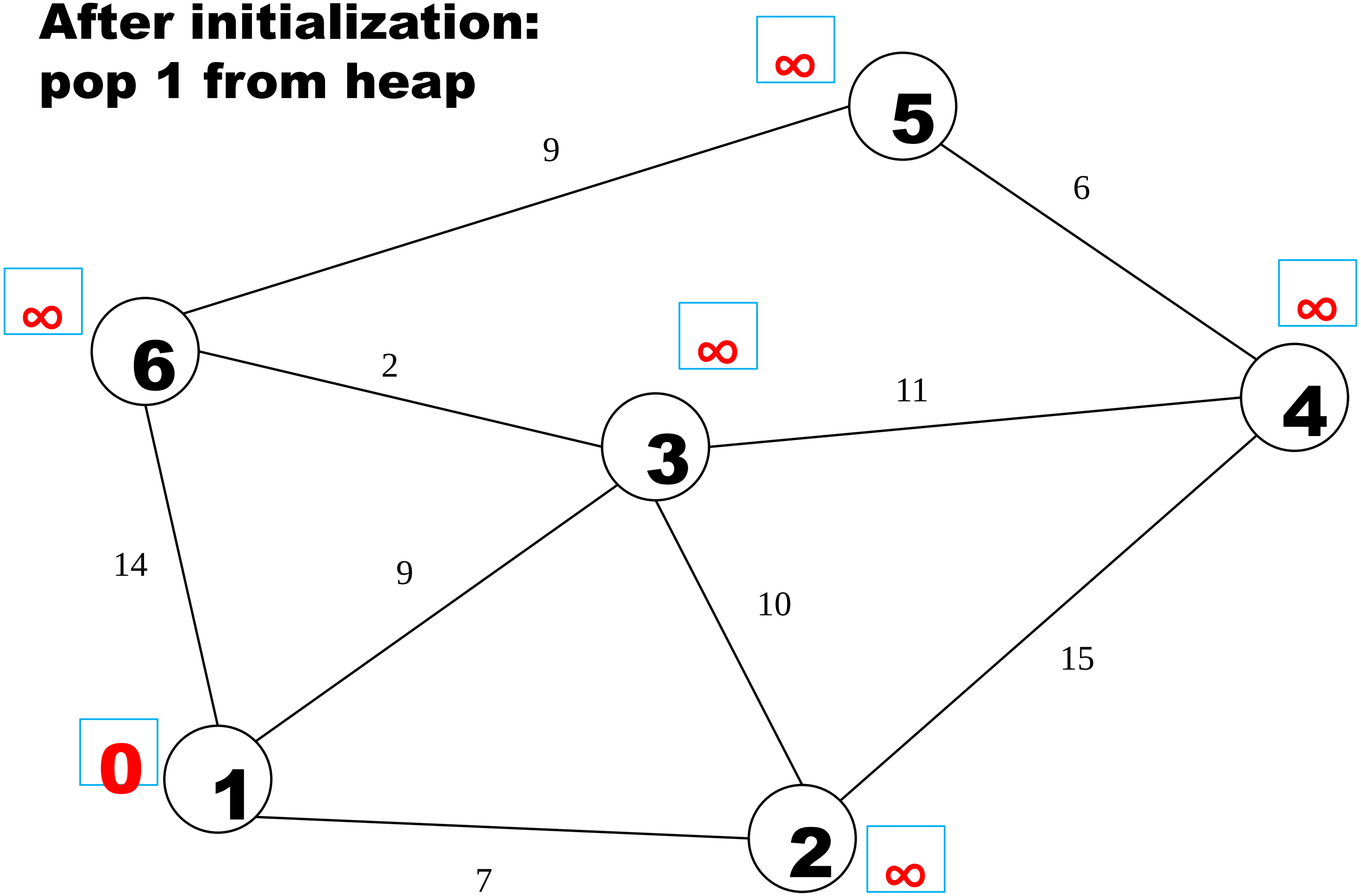
```
Dijkstra(Graph G, Vertex s)
Initialize(G, s);
Priority_Queue minQ = {all vertices in V};
while(minQ ≠ ∅){
   Vertex u = ExtractMin(minQ); // minimum est(u)
   for(each v ∈ minQ such that (u, v) ∈ E)
      Relax(u, v);
}
```

Dijkstra’s Algorithm Example



Dijkstra's Algorithm Example

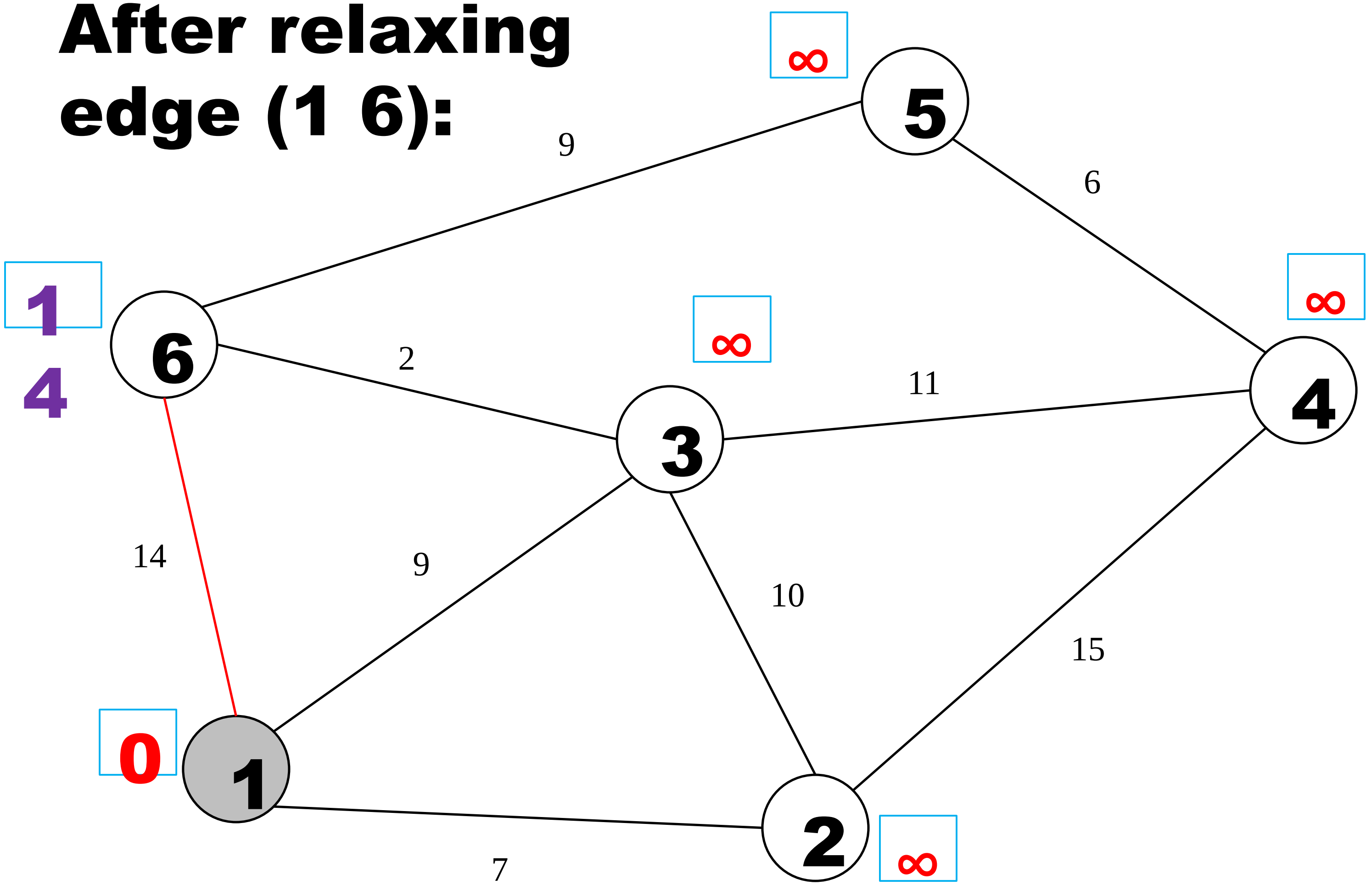
After initialization:
pop 1 from heap



minQ = { {1,0}, {6,∞}, {3,∞}, {2,∞}, {4,∞}, {5,∞} }

Dijkstra's Algorithm Example

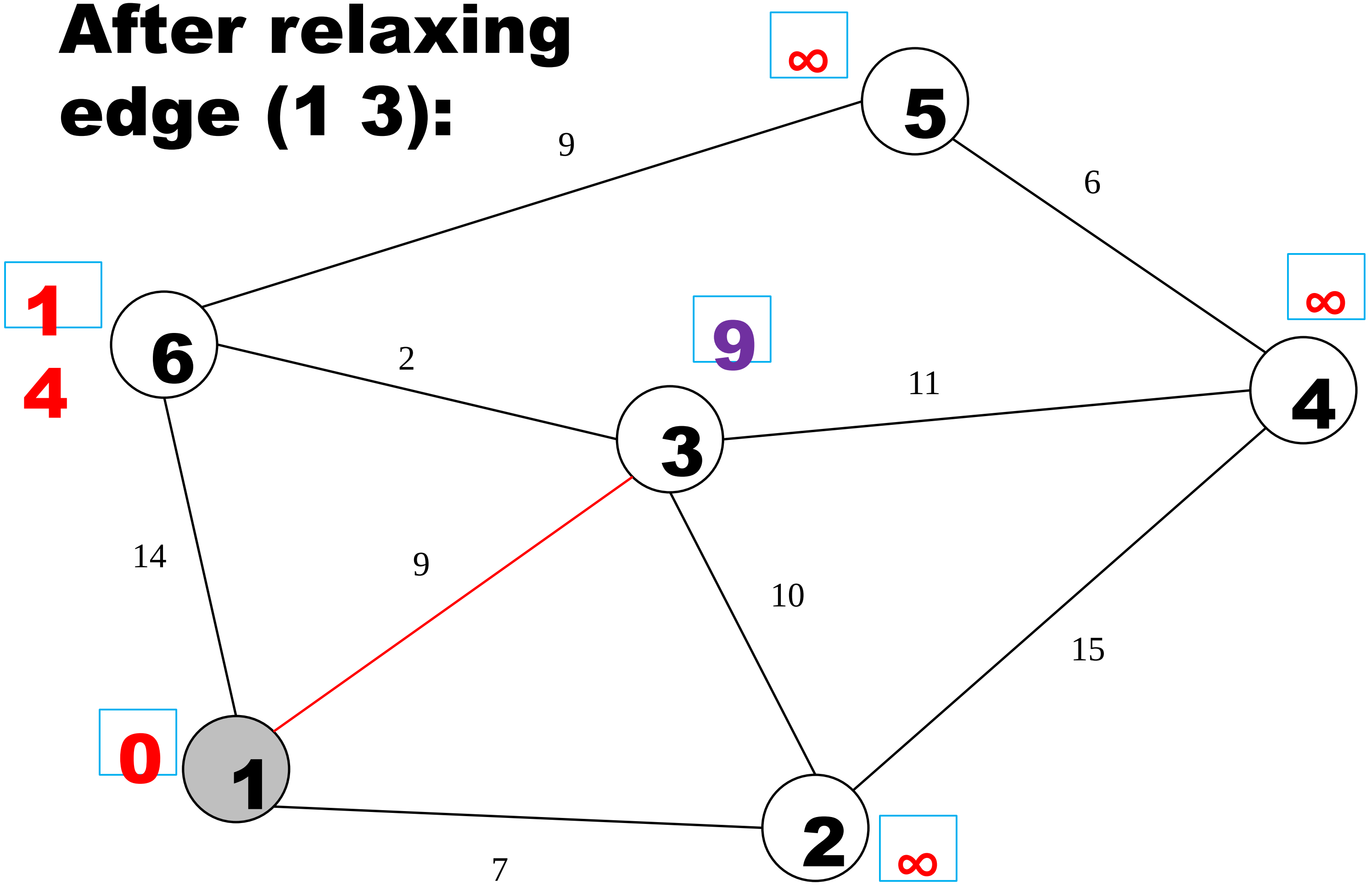
After relaxing edge (1 6):



minQ = { {6,14}, {3,∞}, {2,∞}, {4,∞}, {5,∞} }

Dijkstra's Algorithm Example

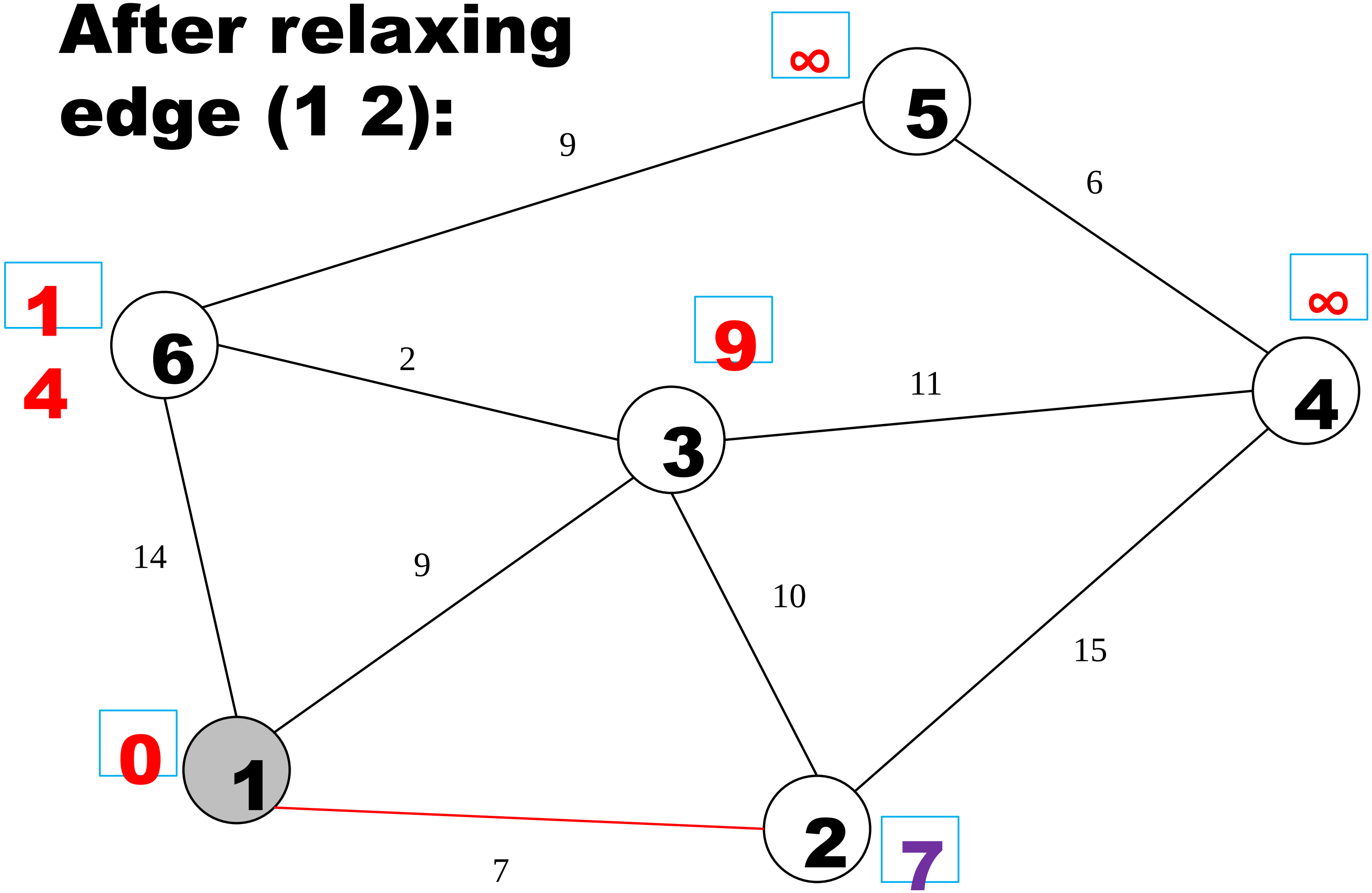
After relaxing edge (1 3):



minQ = { {3,9}, {6,14}, {2,∞}, {4,∞}, {5,∞} }

Dijkstra's Algorithm Example

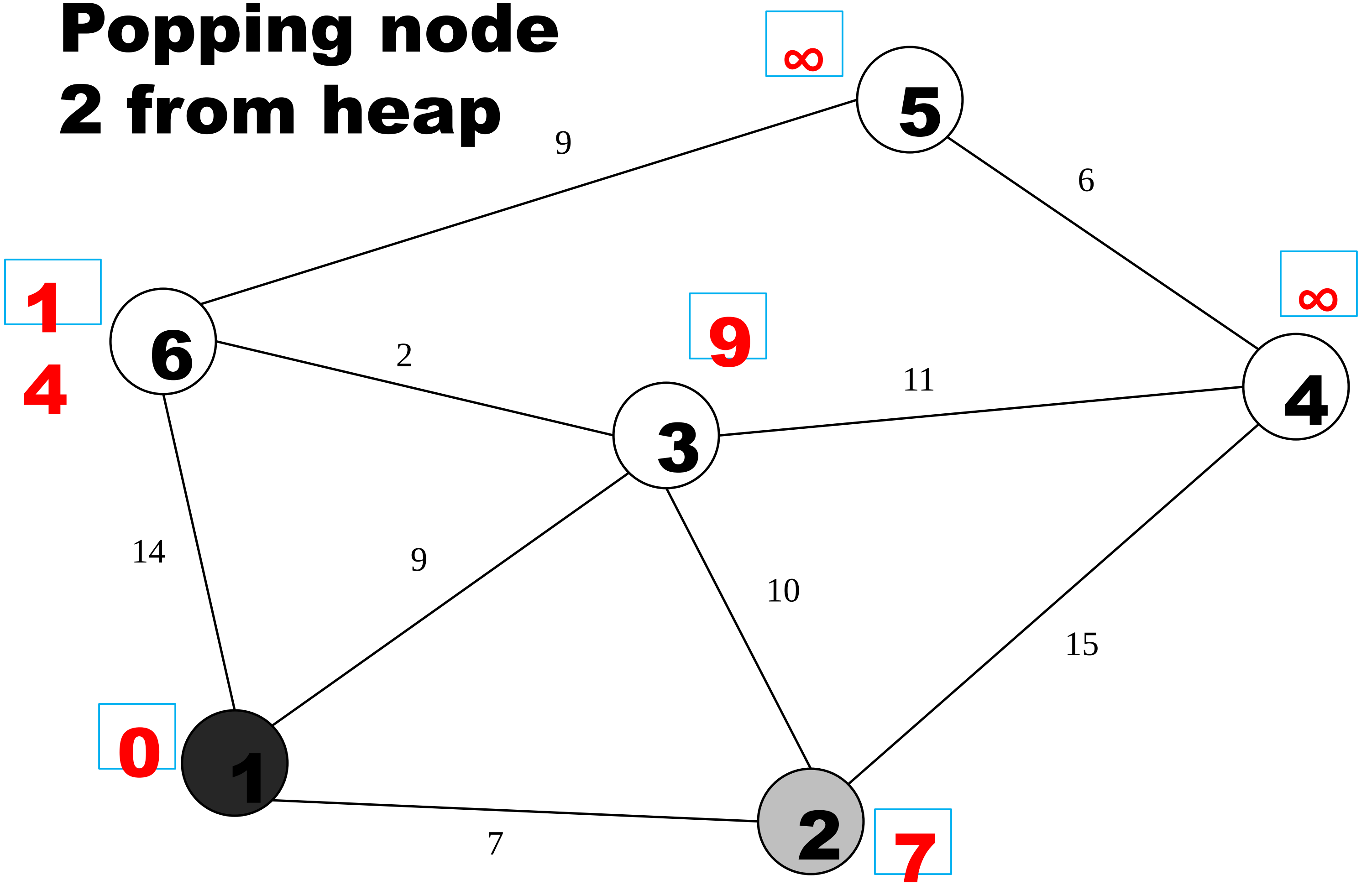
After relaxing edge (1 2):



minQ = { {2,7}, {3,9}, {6,14}, {4,∞}, {5,∞} }

Dijkstra's Algorithm Example

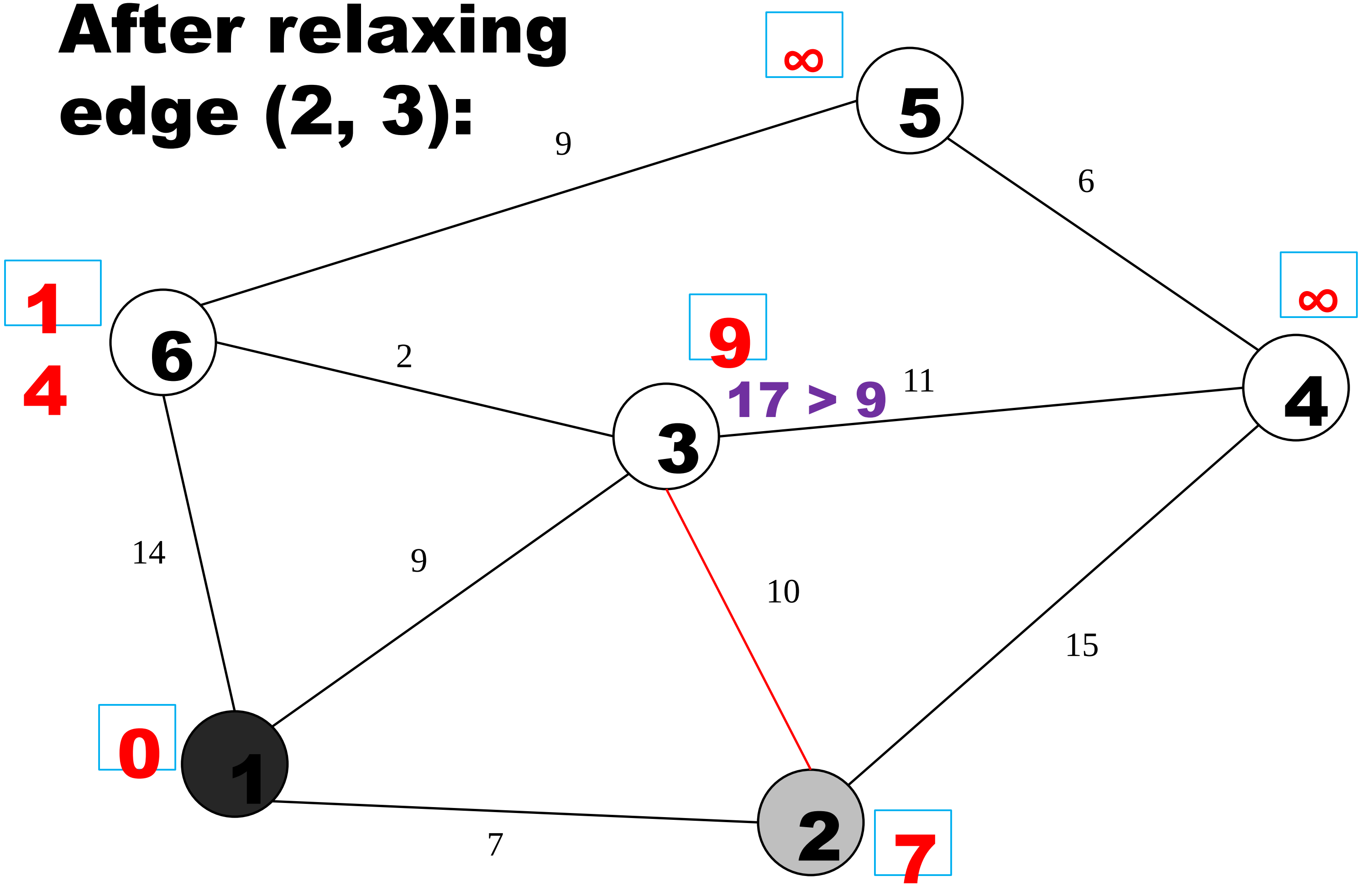
Popping node 2 from heap



minQ = { {3,9}, {6,14}, {4,∞}, {5,∞} }

Dijkstra's Algorithm Example

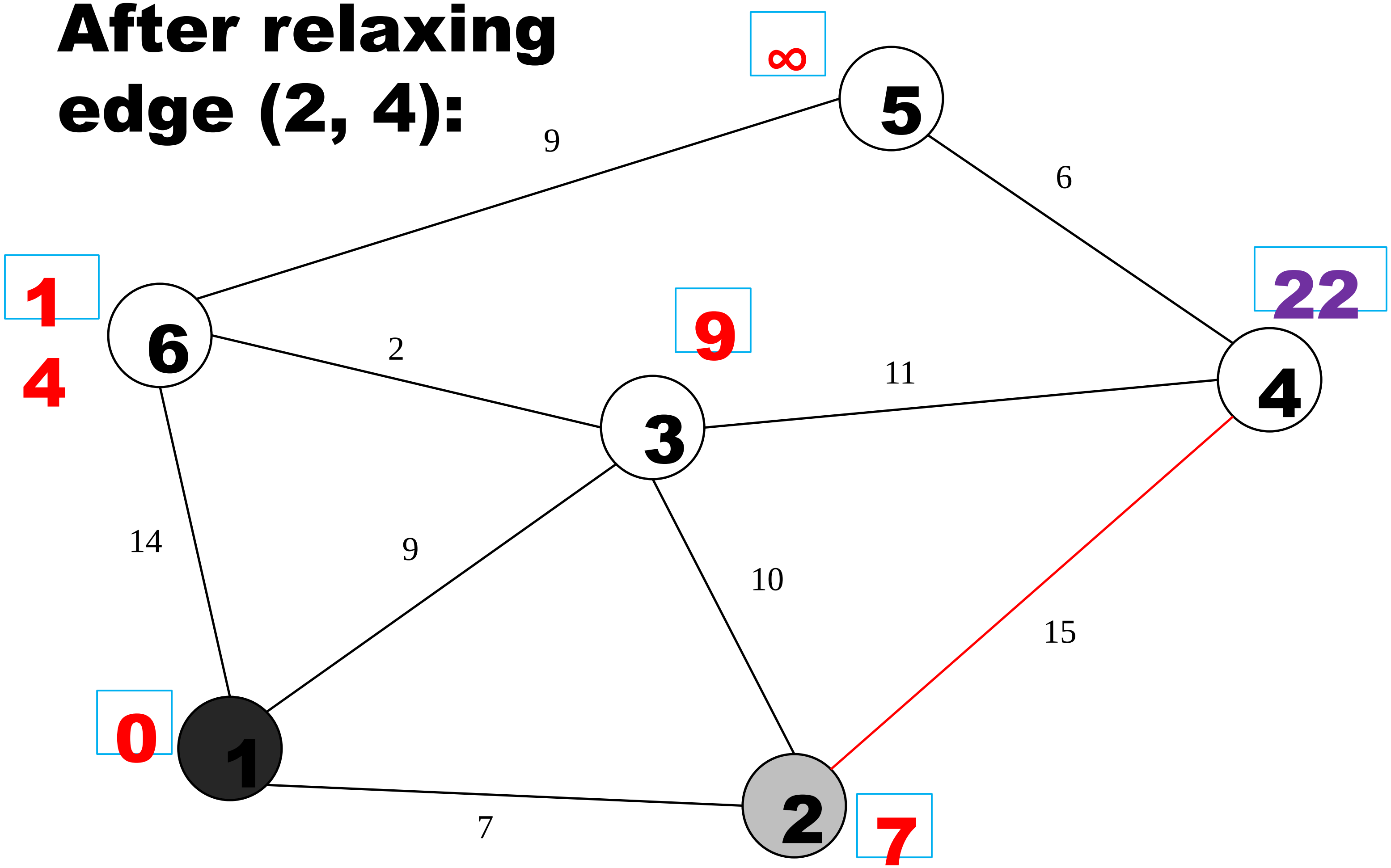
After relaxing edge (2, 3):



minQ = { {3,9}, {6,14}, {4,∞}, {5,∞} }

Dijkstra's Algorithm Example

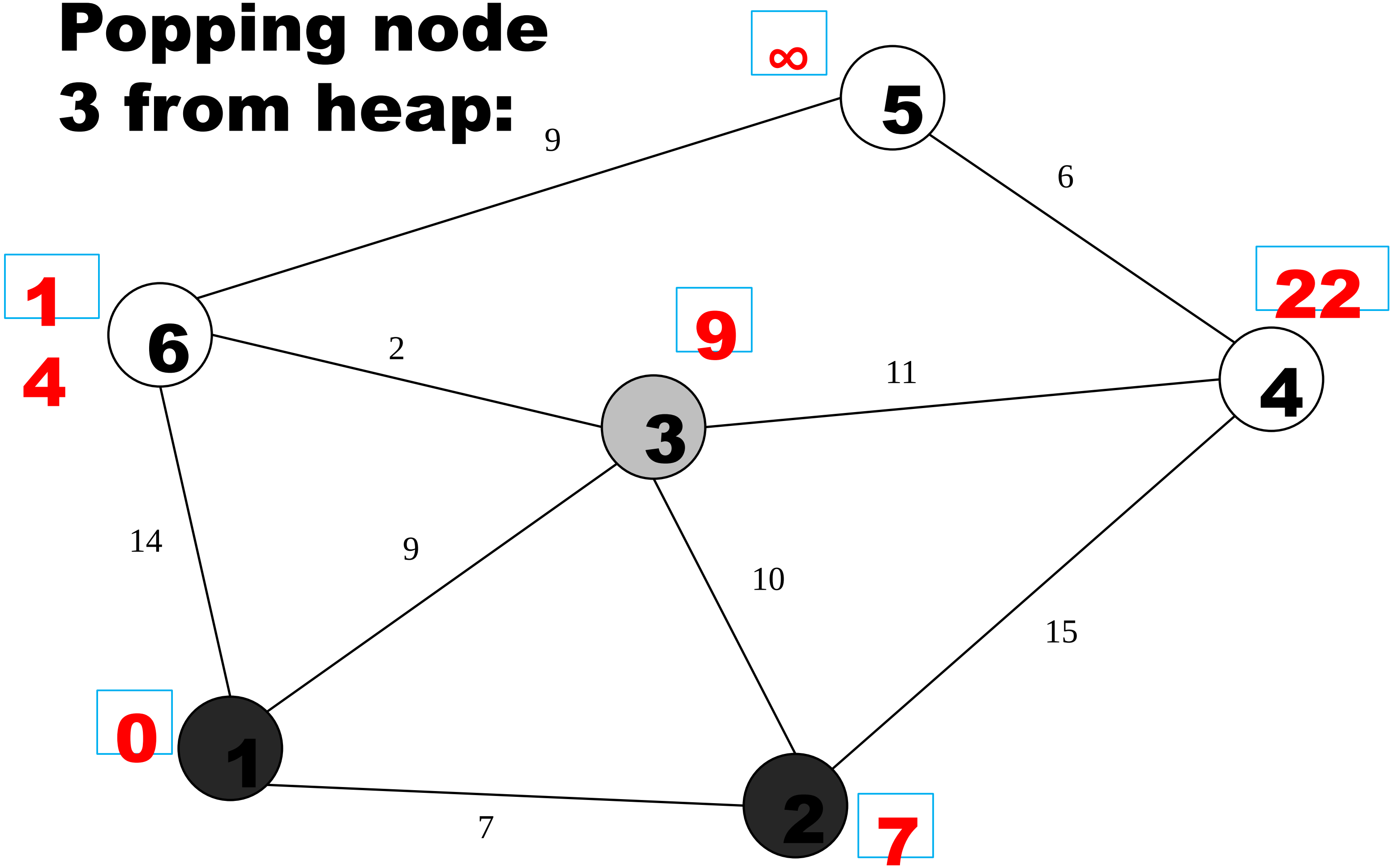
After relaxing edge (2, 4):



minQ = { {3,9}, {6,14}, {4,22}, {5,∞} }

Dijkstra's Algorithm Example

Popping node 3 from heap:

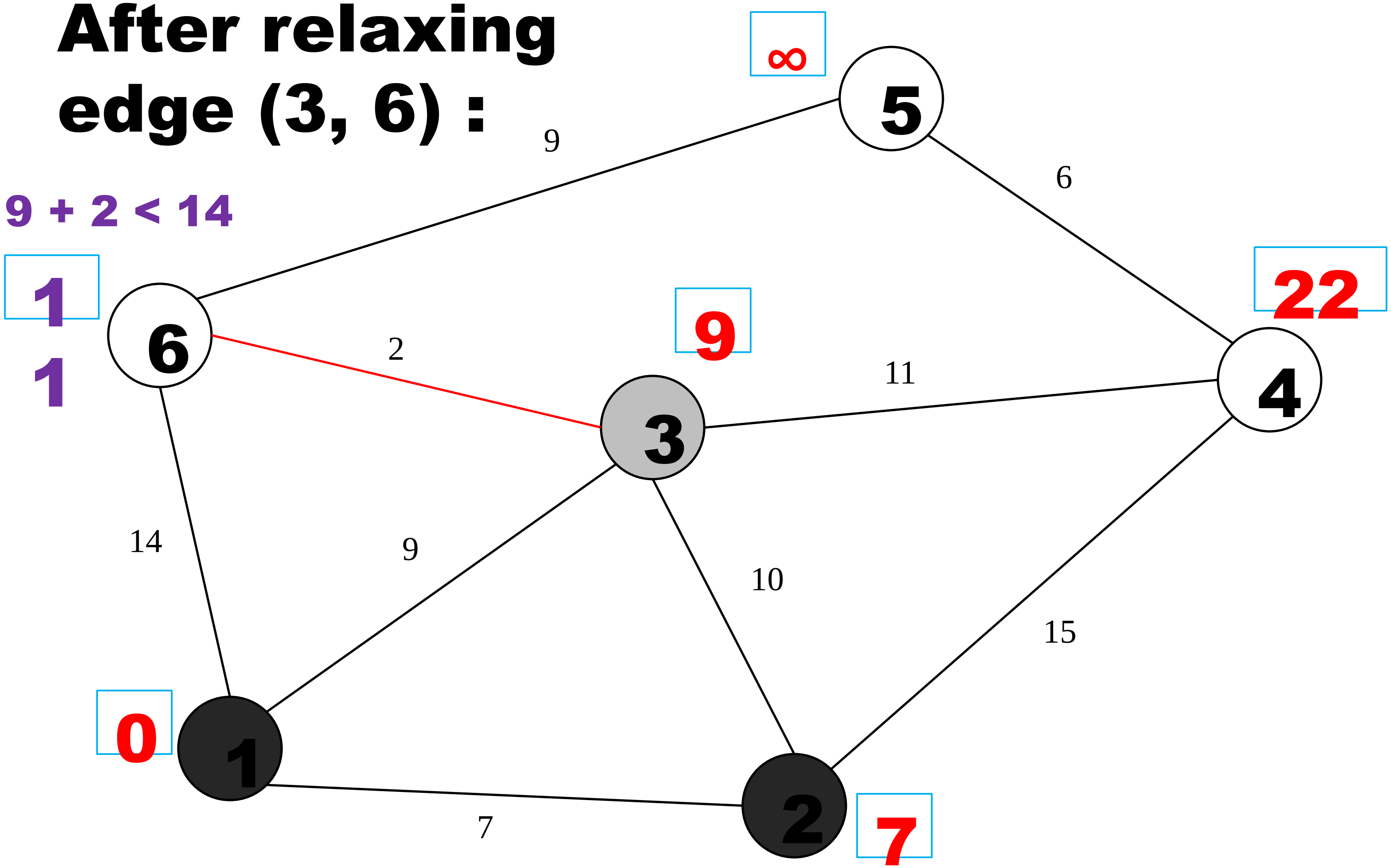


minQ = { {3,9}, {6,14}, {4,22}, {5,∞} }

Dijkstra's Algorithm Example

After relaxing edge (3, 6) :

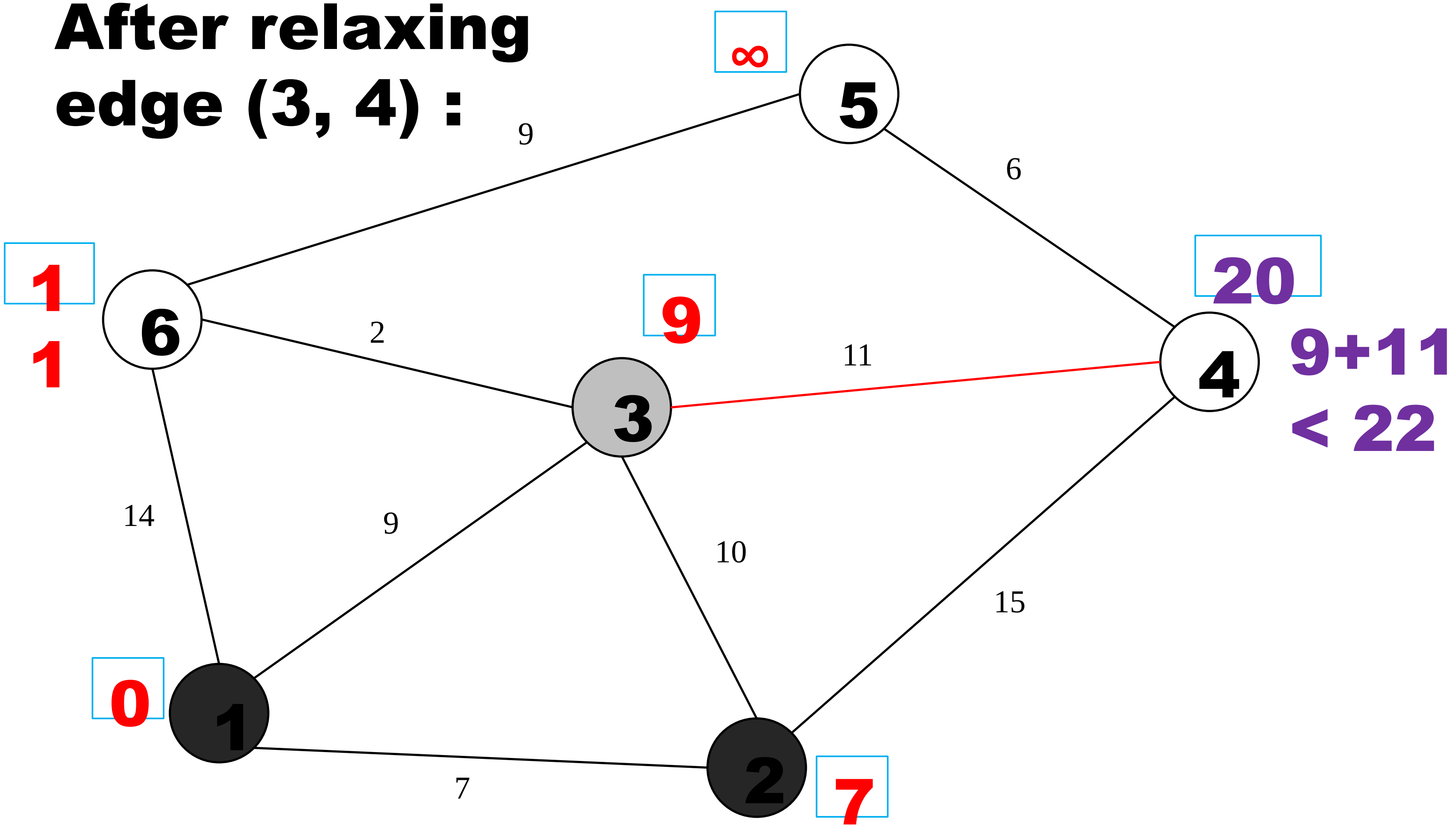
9 + 2 < 14



minQ = { {6,11}, {4,22}, {5,∞} }

Dijkstra's Algorithm Example

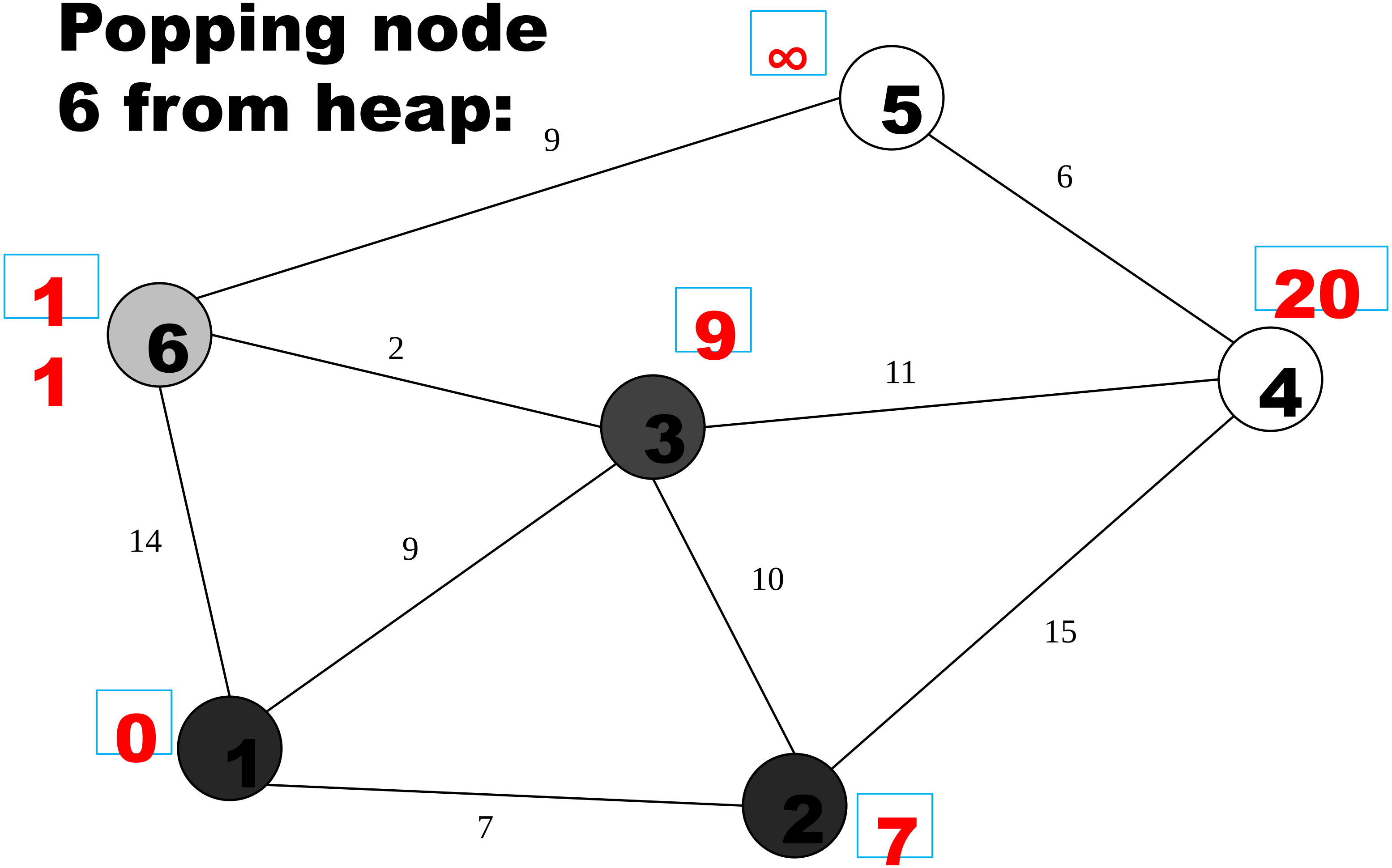
After relaxing edge (3, 4) :



minQ = { {6,11}, {4,20}, {5,∞} }

Dijkstra's Algorithm Example

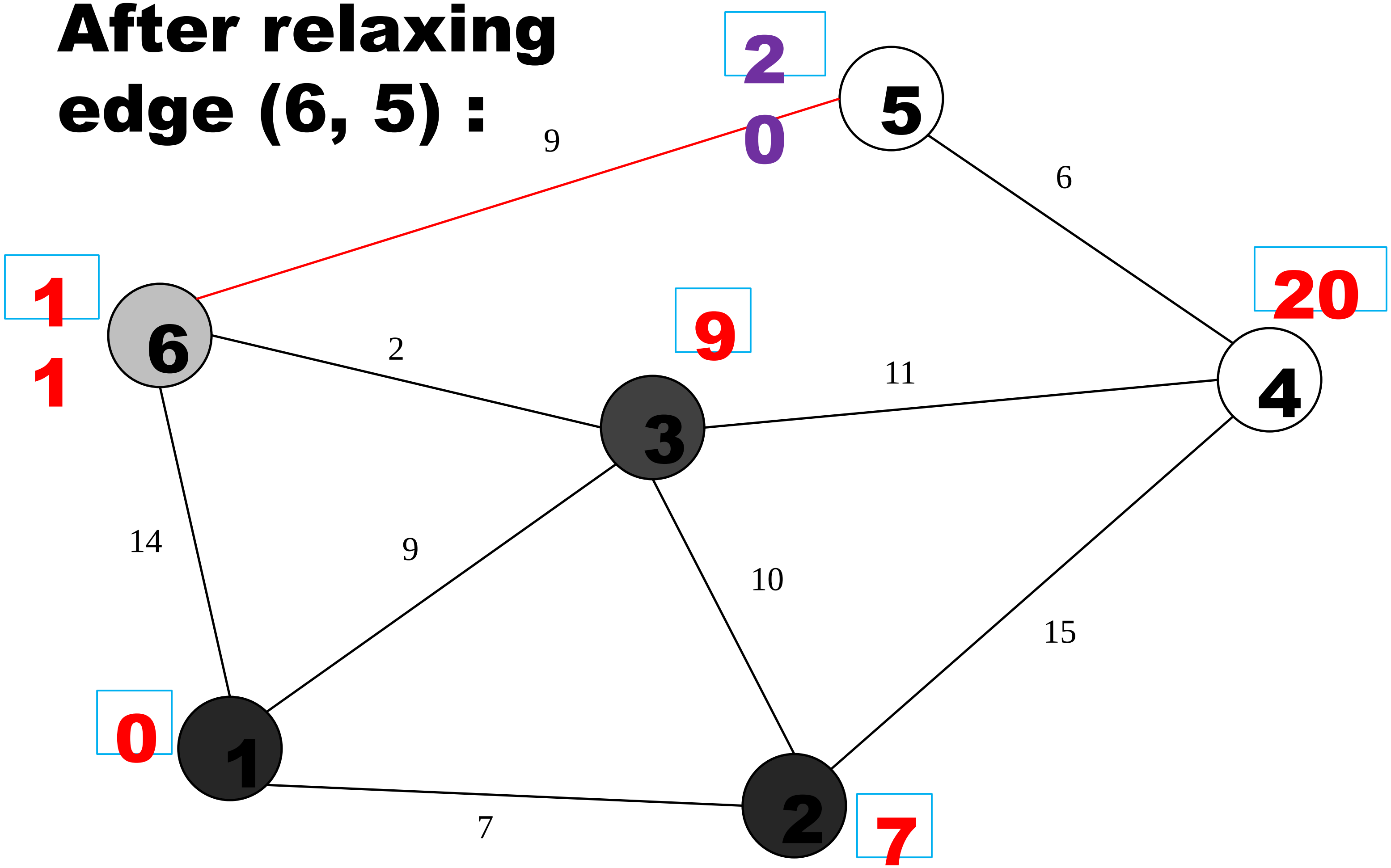
Popping node 6 from heap:



minQ = { {4,20}, {5,∞} }

Dijkstra’s Algorithm Example

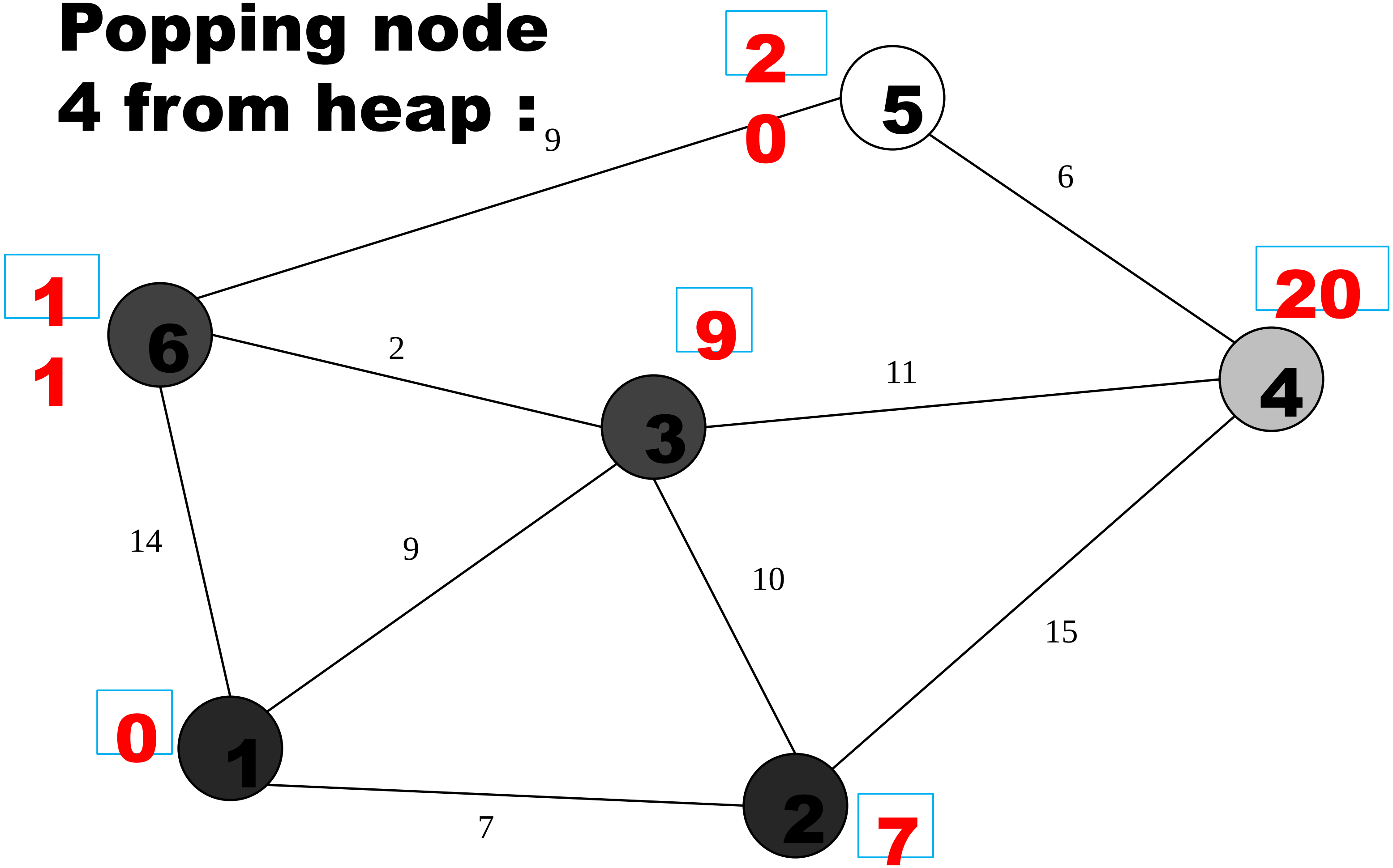
After relaxing edge (6, 5) :



minQ = { {4,20}, {5,20} }

Dijkstra's Algorithm Example

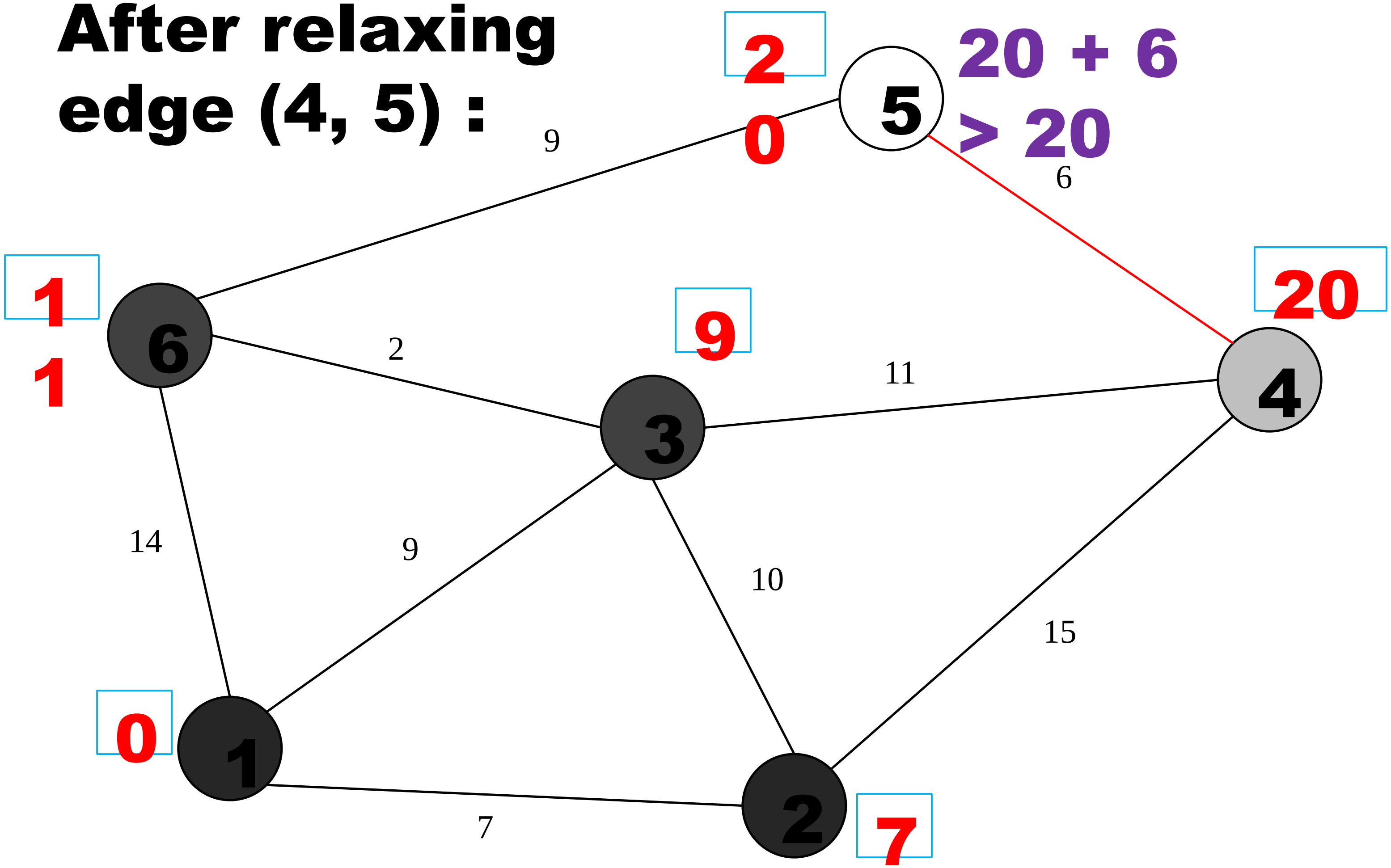
Popping node 4 from heap :



minQ = { {5,20} }

Dijkstra's Algorithm Example

After relaxing edge (4, 5) :

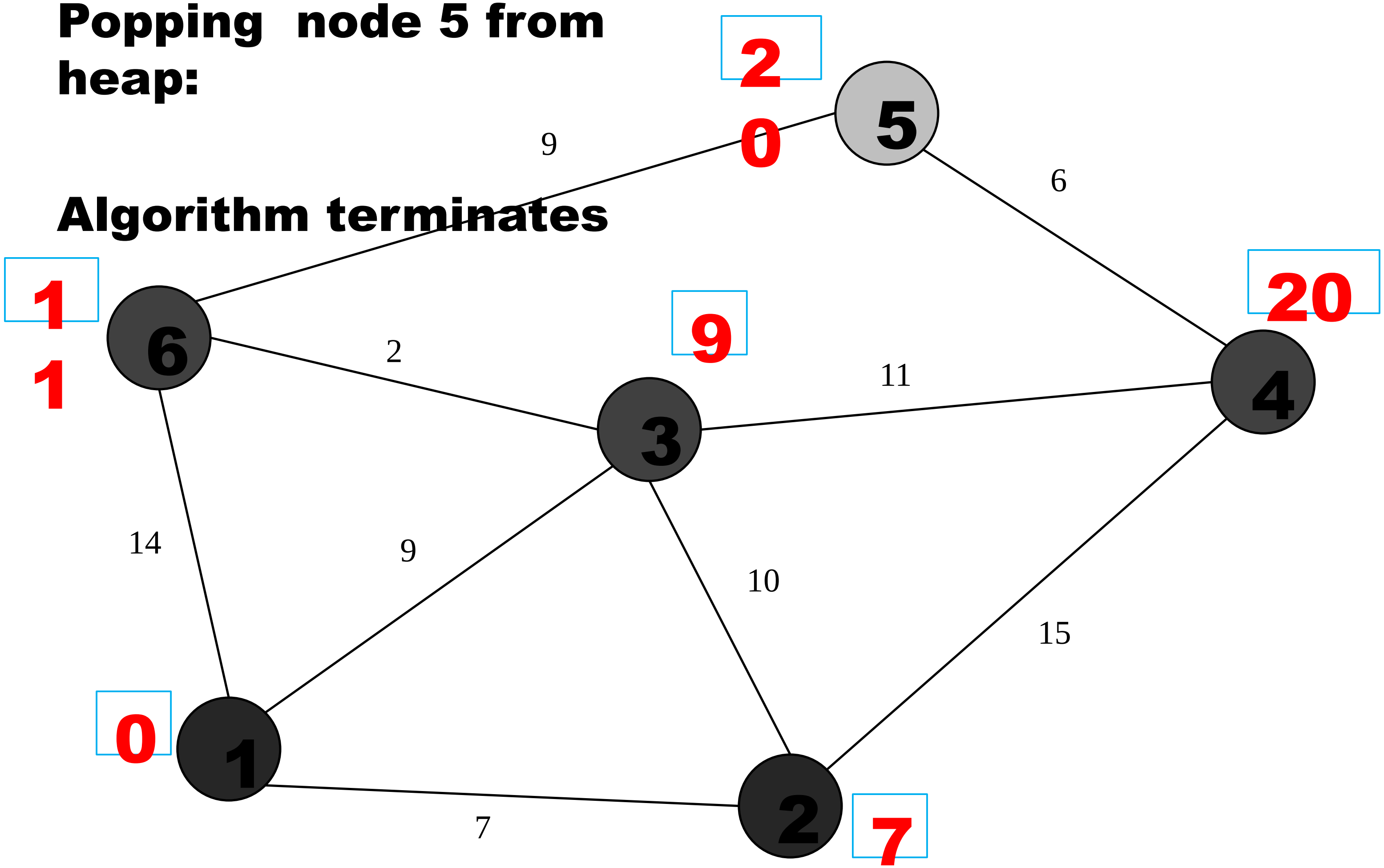


minQ = { {4,20}, {5,20} }

Dijkstra's Algorithm Example

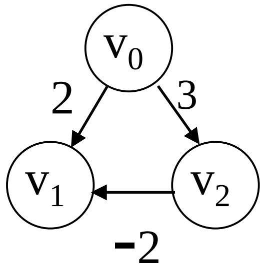
Popping node 5 from heap:

Algorithm terminates



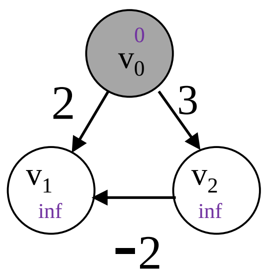
minQ = {}

Dijkstra's Algorithm



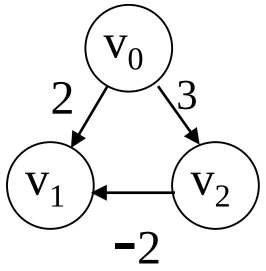
	Predecessors			Shortest-Path Estimates		
	v_0	v_1	v_2	v_0	v_1	v_2
Dijkstra's	NIL	v_0	v_0	0	2	3
Correct path	NIL	v_2	v_0	0	1	3

Dijkstra's doesn't work with negative edge weights



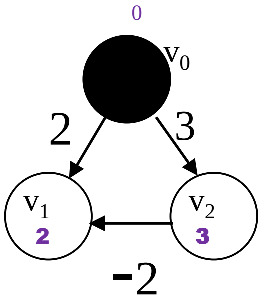
Queue: {0,0}, {1,inf} {2,inf}

Dijkstra's Algorithm



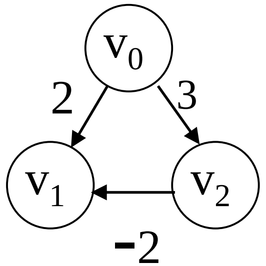
	Predecessors			Shortest-Path Estimates		
	v_0	v_1	v_2	v_0	v_1	v_2
Dijkstra's	NIL	v_0	v_0	0	2	3
Correct path	NIL	v_2	v_0	0	1	3

Dijkstra's stops due to coloring of node

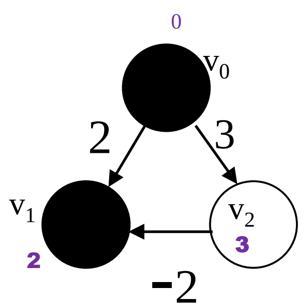


Queue: {1,2} {2,3}

Dijkstra's Algorithm

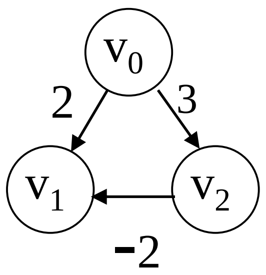


	Predecessors			Shortest-Path Estimates		
	v_0	v_1	v_2	v_0	v_1	v_2
Dijkstra's	NIL	v_0	v_0	0	2	3
Correct path	NIL	v_2	v_0	0	1	3



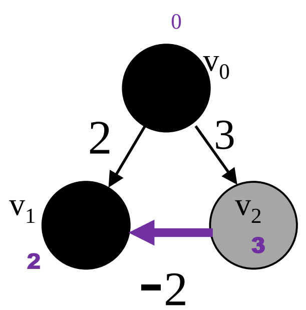
Queue: {2,3}

Dijkstra’s Algorithm



	Predecessors			Shortest-Path Estimates		
	v_0	v_1	v_2	v_0	v_1	v_2
Dijkstra's	NIL	v_0	v_0	0	2	3
Correct path	NIL	v_2	v_0	0	1	3

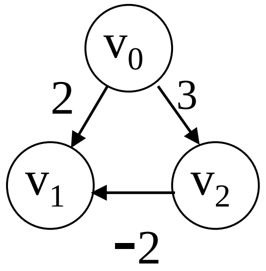
Dijkstra’s stops due to coloring of node



Queue: {2,3}

Purple edge is never explored when relaxing node 2 because node 1 is colored black

Dijkstra's Algorithm



	Predecessors			Shortest-Path Estimates		
	v_0	v_1	v_2	v_0	v_1	v_2
Dijkstra's	NIL	v_0	v_0	0	2	3
Correct path	NIL	v_2	v_0	0	1	3

- produces incorrect results if weights are negative
- Time complexity depends on the implementation of the priority queue *minQ*
  - A linear array: $O(V^2)$
  - A Fibonacci / binary heap: $O(E + V \cdot \lg V)$
  - reference implementation:
    - ~wps100020/itools/lib/std/src/graph.c

The Bellman-Ford Algorithm

- Relax every edge (|*V*| - 1) times
  - since negative cycles should not exist in a shortest-path problem
- The most general algorithm
  - Also the most time-consuming: *O*(*VE*)

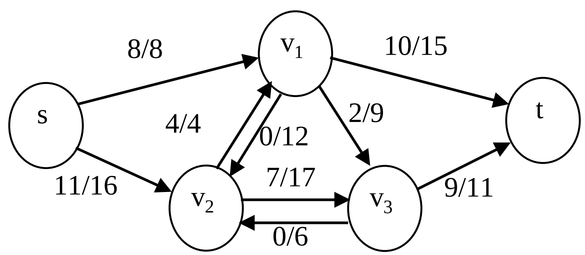
```
Bellman-Ford(Graph G, Vertex s)
Initialize(G, s);
for(counter = 1 to |V| - 1)
   for(each edge (u, v) ∈ E)
      Relax(u ,v);
for(each edge (u, v) ∈ E)
   if(est(v) > est(u) + w((u, v)))
      report "negative-weight cycles exist";
```

Flow Network

- A variant of connected, directed graphs
- Two special nodes:
  - source s: no edge incident to s
  - sink t: no edge incident from t
  - Every flow starts at s and ends at t
- Every edge (u, v) has two attributes:
  - capacity $c(u, v)$: the flow it can hold
  - Flow $f(u, v)$ satisfies 3 constraints:
    - Capacity constraint: $f(u, v) \leq c(u, v)$
    - Skew symmetry: $f(u, v) = -f(v, u)$
    - Flow conservation (exceptions: s and t): $\sum_{v \in V} f(u, v) = 0$

Maximum-Flow Problem

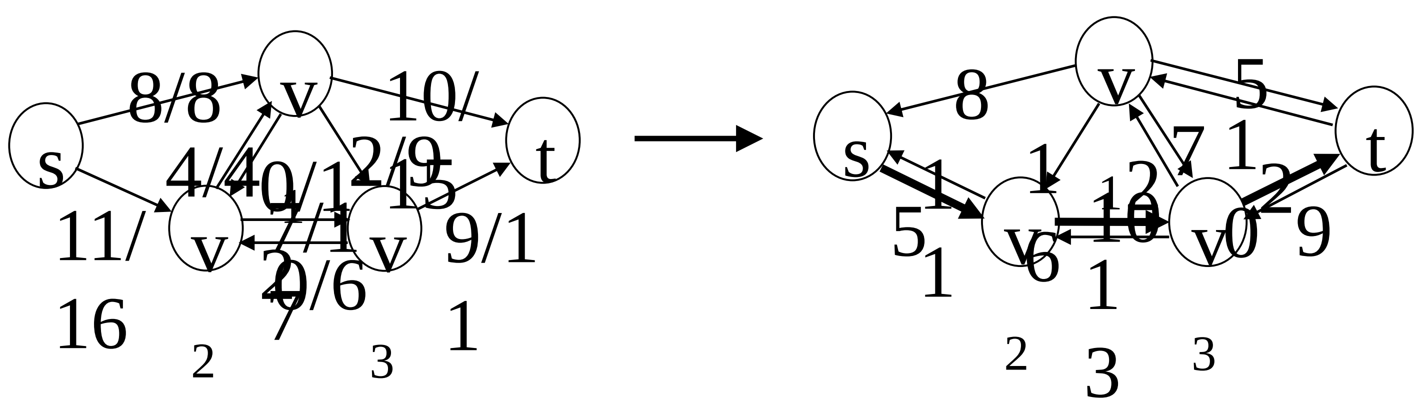
- The value of a flow: $|f| = \sum_{v \in V} f(s,v)$
- Maximum flow problem
  - —finds the flow with the maximum value in a flow network



- Numbers on edges: $f(u, v)/c(u, v)$
- $|f| = 19$, not maximum
  - —More flow can be pushed into path $s \rightarrow v_2 \rightarrow v_3 \rightarrow t$
    - – An *augmenting path*

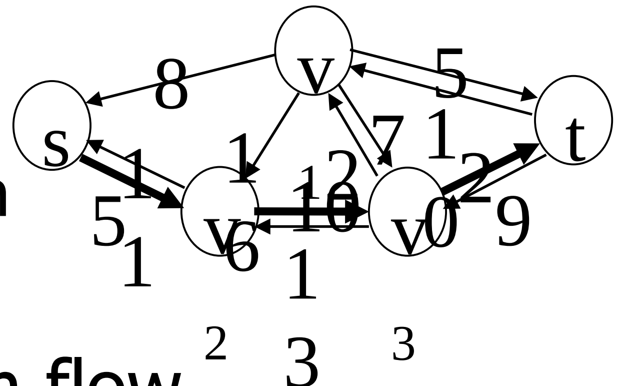
Residual Network

- Facilitates finding augmenting paths
- Residual capacity:
  - —defined with respect to a flow f
  - —$c_f(u, v) = c(u, v) - f(u, v)$
  - —For both directions of every pairs of nodes
- $G_f = (V, E_f)$
  - —E_f: edges with residual capacity as weights



Residual Network

- Augmenting paths
  - —paths in the residual network from *s* to *t*
  - —E.g. $p = s \rightarrow v_2 \rightarrow v_3 \rightarrow t$
- Residual capacity of a path
  - —Minimum edge weight on the path
  - —$c_f(p) = c_f(v_3, t) = 2$



- Intuitive algorithm for maximum flow
  - —Finds augmenting paths in residual networks and push flows equal to their residual capacity
  - —Updates residual networks according to the new flow until no augmenting paths can be found

The Ford-Fulkerson Method

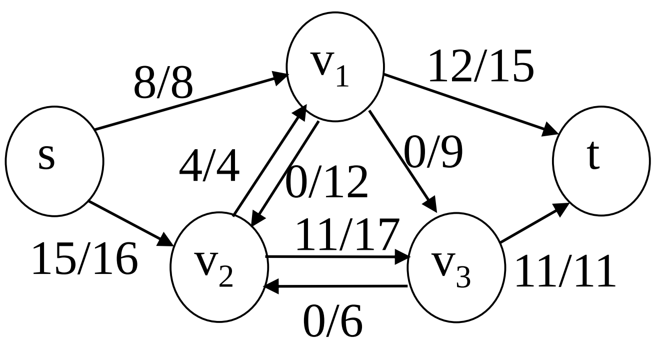
- An intuitive method
  - Finds augmenting paths *p* on the residual network
  - Push more flow according to $c_f(p)$
  - Update the residual network

```
Ford-Fulkerson(Graph G, Source s, Sink t)
  for(each (u, v) ∈ E) f[u, v] = f[v, u] = 0;
  Build a residual network G_f based on flow f;
  while(there is an augmenting path p in G_f){
     c_f(p) = min(c_f(u, v) : (u, v) ∈ p);
     for(each edge (u, v) ∈ p){
        f[u, v] = f[u, v] + c_f(p);
        f[v, u] = -f[u ,v];
     }
     Rebuild G_f based on new flow f;
  }
```

- Time complexity: $O(E \cdot |f^*|)$
  - f^*: the maximum flow
  - $|f^*|$ can be very large
  - Very inefficient if $|f^*|$ is large

The Edmonds-Karp Algorithm

- In Ford-Fulkerson, how to find augmenting paths is unspecified
  - Ford-Fulkerson: a "method"
  - Edmonds-Karp uses *breadth-first search* to find augmenting paths
- Time complexity: $O(E \cdot VE) = O(VE^2)$



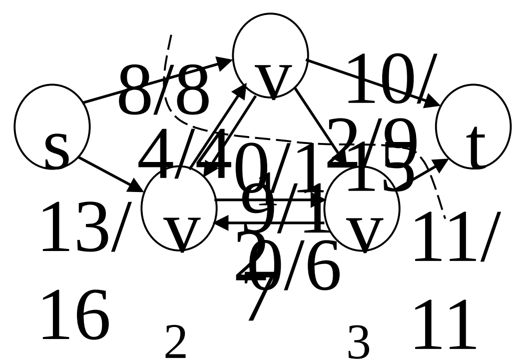
- Resultant network
  - The maximum flow
  - $|f^*| = 23$

Cuts in flow networks

- A cut (S, T)
  - a partition of the node set V into S and $T = V - S$
  - source $s \in S$ and sink $t \in T$
    - $S = \{s, v_2, v_3\}$, $T = \{t, v_1\}$
  - net flow across the cut, $f(S, T)$:
    - $f(S, T) = 21$
  - capacity of the cut, $c(S, T)$:
    - $c(S, T) = 23$
    - $f(S, T) \le c(S, T)$

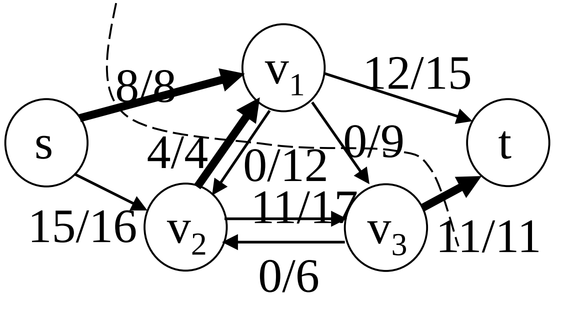
$$f(S,T) = \sum_{u \in S, v \in T} f(u,v)$$

$$c(S,T) = \sum_{u \in S, v \in T} c(u,v)$$



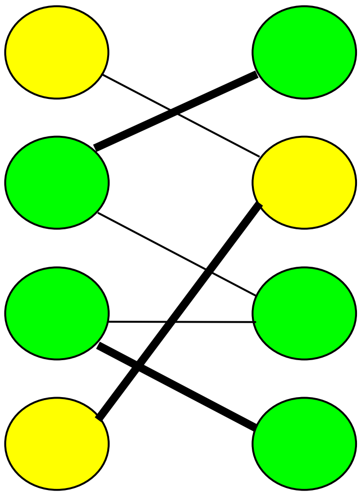
The Max-Flow Min-Cut Theorem

- The following 3 things are equivalent:
  - f is a maximum flow in G
  - The residual network G_f has no augmenting paths
  - $|f| = c(S, T)$ for some cut of G
- Finding maximum flow = finding minimum cut
  - $|f^*| = 23 = c(S, T) =$ $c(\{s, v_2, v_3\}, \{t, v_1\})$



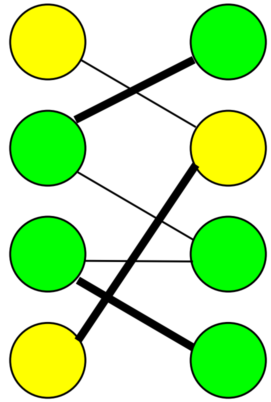
Maximum Bipartite Matching

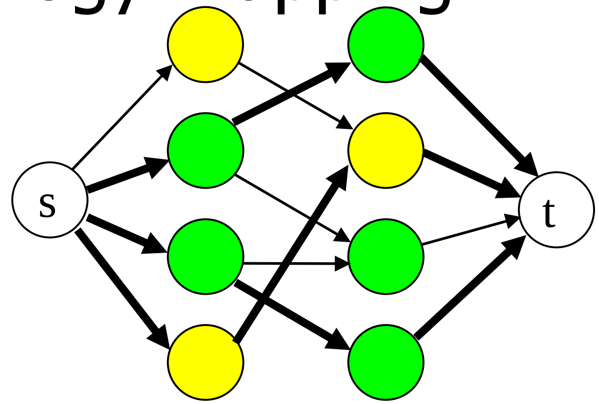
- A bipartite graph $G = (V, E)$
  - V is partitioned into two sets L and R
  - For every edge $(u, v) \in E$, if $u \in L$, then $v \in R$, and vice versa
- A matching
  - A subset of edges $M \subseteq E$
  - At most one edge of M is incident on V
  - 3 thick edges in the bipartite graph



Maximum Bipartite Matching

- Maximum bipartite matching finds a matching with maximum edges
  - Add a source *s* and link *s* to all nodes in *L*
  - Add a sink *t* and link all nodes in *R* to *t*
  - Every edge has unit capacity, solve the maximum flow problem
- Ford-Fulkerson solves in $O(VE)$
- Applications: technology mapping





Heuristic Algorithms

- Applies heuristics, or rules of thumb
- Finds good but not always optimal solutions
- Efficient in time
  - Best for hard (NPC or NP-hard) problems
- Solution quality cannot always be guaranteed
  - Nearest Neighbor for TSP
- Either directly searches the solution space
  - Greedy algorithm, dynamic programming, branch and bound
- Or exerts perturbations on solutions
  - Simulated annealing, genetic algorithms

Greedy Algorithm

- General idea:
  - —stages the optimization problem
  - —makes *locally optimal* choices at each stage
- Real life example:
  - —giving change with minimum #coins
  - —heuristic: pick the coin with the greatest value
  - —36 cents: quarter → dime → penny: 3 coins
- Two properties make greedy algorithms work
  - —Greedy choice
  - —Optimal substructure
- Applications: Dijkstra’s, Prim’s algorithms

Greedy Choice Property

- The global optimal solution can be made by making locally optimal choices
- Does not consider the impact of the current choice on future choices
- Counterexamples
  - Nearest Neighbor for TSP
  - Giving change of 40 cents if there were 20-cent coins
    - (Greedy) quarter → dime → nickel: 3 coins
    - (Optimal) 2 20-cent coins: 2 coins

Optimal Substructure Property

- The global optimal solution consists of optimal solutions to its subproblems
  - —The problem is divisible into subproblems
  - —The combination of optimal solutions to subproblems is globally optimal
- Giving change of 36 cents
  - —into 26 cents + 10 cents:
  - —(quarter → penny) + dime : global optimal

Dynamic Programming (DP)

- Combines solutions to its *dependent* subproblems by utilizing the dependency
  - Unlike divide-and-conquer: subproblems are independent
  - avoids repeatedly solving the same subproblems
- Example: matrix-chain multiplication
  - Find the multiplication sequence with least #scalar multiplications
  - Matrices A, B, C: 30 x 100, 100 x 2, 2 x 50
    - (AB)C: #scalar multiplications = 9000
    - A(BC): #scalar multiplications = 160,000

Two Properties for DP

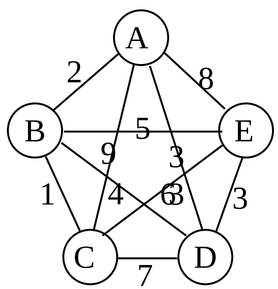
- Overlapping Subproblems
  - —The decomposed subproblems are dependent or overlapped
- Optimal Substructure
  - —the same as in greedy algorithms
  - —DP or greedy?
    - – whether the problem has "*overlapping subproblems*" or "*greedy choice*"
- Matrix-chain multiplication has both

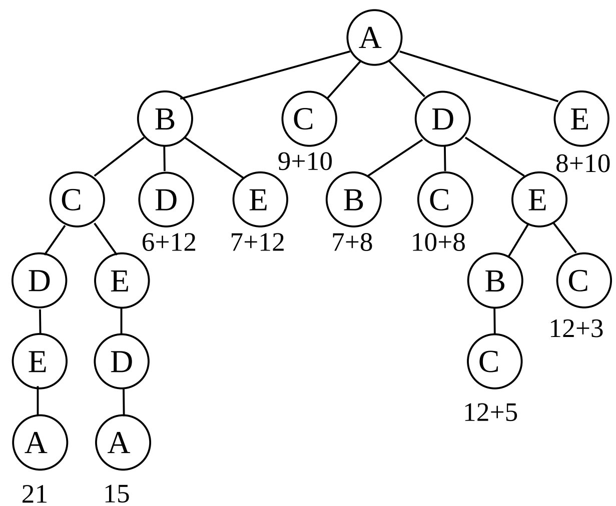
Branch and Bound

- Branching
  - makes several choices at the same point to branch out into the search space
  - the solution space forms a tree-like structure
  - fully-branched space: too vast to explore
- Bounding and pruning
  - estimates a lower bound on solution quality to prune out obviously impossible branches
  - efficiently reduces the solution space made with branching

Branch and Bound for TSP

- Branching: the next node on the route
- Bounding: use MST to estimate the cost lower bound of unvisited route





- Other important applications:
  — DPLL Boolean Satisfiability Search Scheme

Mathematical Programming

- Problem formulation
  - ⇨ minimize(or maximize) $f(x)$;
  - ⇨ subject to $X = \{ x \mid g_i(x) \leq b_i,\ i = 1 \ldots m \}$;

  where
  - — $x = (x_1, \ldots, x_n)$ are optimization (or decision) variables,
  - — $f : R^n \rightarrow R$ is the objective function
  - — $g_i : R^n \rightarrow R$ and $b_i \in R$ form the constraints for the valid values of x.

Categories of Mathematical Programming Problems

1. If $X = R^n$, the problem is unconstrained;
2. If f and all the constraints are linear, the problem is called a ***linear programming*** (LP) problem
   — Can then be represented in the matrix form: $Ax \leq B$
   where A is an m×n matrix corresponding to the coefficients in $g_i(x)$
3. If the problem is linear, and all the variables are constrained to integers, the problem is called an ***integer linear programming*** (ILP) problem
   — If only some of the variables are integers, it is called a ***mixed integer linear programming*** (MILP or MIP) problem.

Categories of Mathematical Programming Problems

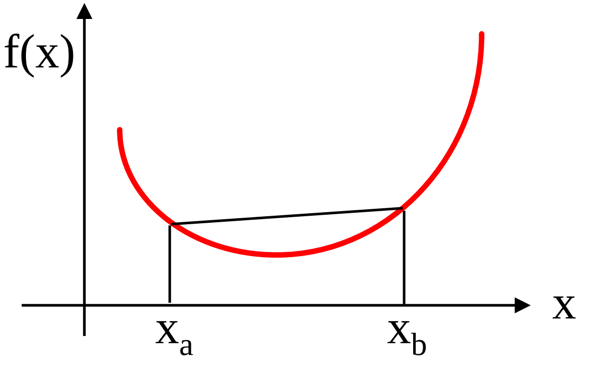
4. If the constraints are linear, but the objective function f contains some quadratic terms, the problem is called a ***quadratic programming*** (QP) problem.
5. If f or any of $g_i(x)$ is not linear, it is called a ***nonlinear programming*** (NLP) problem
6. If all the constraints have the following convexity property: $g_i(\alpha x_a + \beta x_b) \leq \alpha g_i(x_a) + \beta g_i(x_b)$
   — where $\alpha \geq 0$, $\beta \geq 0$, and $\alpha + \beta = 1$

   then the problem is called a ***convex programming*** or ***convex optimization*** problem
7. If the set of feasible solutions defined by f and X are discrete, the problem is called a ***discrete*** or ***combinatorial optimization*** problem.

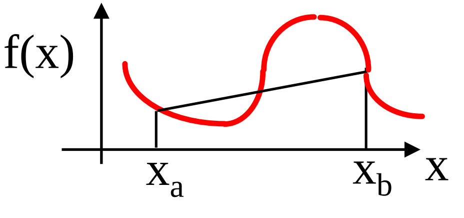
Convex Functions

$f(\underline{\mathbf{x}})$ is a **convex function** if given any two points $\underline{\mathbf{x}}_a$ and $\underline{\mathbf{x}}_b$, the line joining the two points lies on or above the function

Convex f:



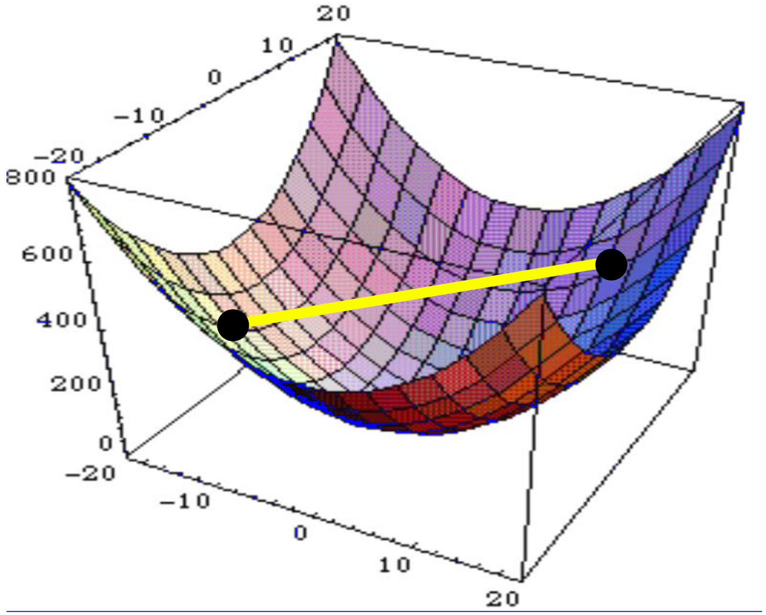
Nonconvex f:



Example

- Convex functions in two dimensions

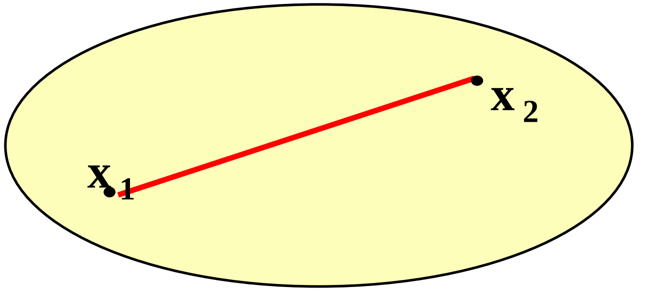
$$f(x_1,x_2) = x_1^2 + x_2^2$$



Convex sets

- Convex sets

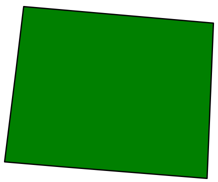
  A set S is a **convex set** if given any two points $\underline{\mathbf{x}}_a$ and $\underline{\mathbf{x}}_b$ in the set, the line joining the two points lies entirely within the set



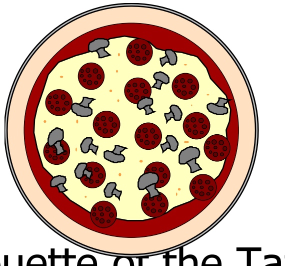
Examples

- Convex sets

  Shape of Wyoming



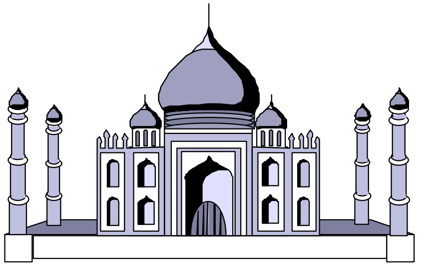
  Shape of an ideal pizza



- Nonconvex Sets

  Shape of CA

  Silhouette of the Taj Mahal



Two Important Properties

- If $f(\underline{\mathbf{x}})$ is a convex function, $f(\underline{\mathbf{x}}) \leq c$ is a convex set
  - Example:

    $f(x_1,x_2) = x_1^2 + x_2^2 \leq c$ is a convex set

- An intersection of convex sets is a convex set

Linear Programming (LP) Problem

- **Intuition**: solving LP problems should be simpler than solving the general mathematical optimization problems

- **Fact**: A polynomial-time algorithm was not available until 1970’s

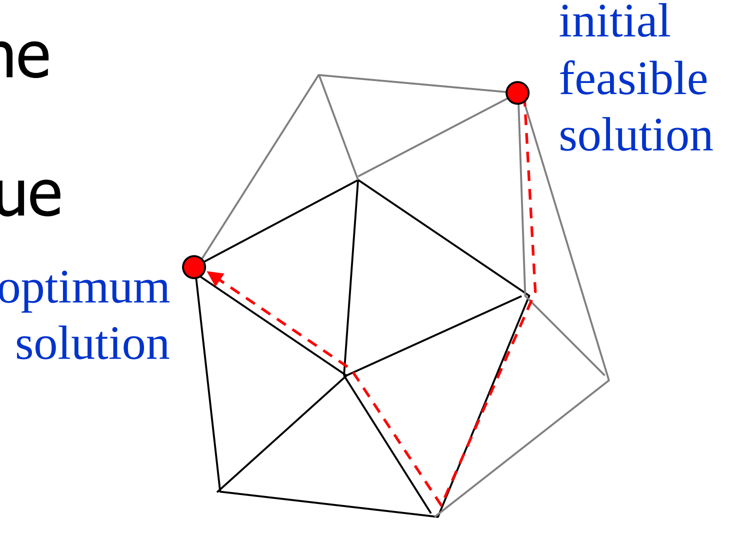
LP Formulation

- One linear equation forms a hyper-plane
- One linear inequality (constraint) forms a hyper half-space
- The set of linear constraints forms a polyhedron in a higher-dimensional space
- The optimal value of a linear objective function over a set of linear constraints occurs at the *extreme point* of the polyhedron

Simplex Method

- Developed by George Dantzig in 1947
  - First practical procedure used to solve the LP problems

1. Finds a basic feasible solution that satisfies all the constraints
   - A basic solution is conceptually a *vertex* (i.e., an *extreme point*) of the convex polyhedron
2. Moves along the *edges* of the polyhedron in the direction towards finding a better value of the objective function
   - Guaranteed to eventually terminate at the optimal solution



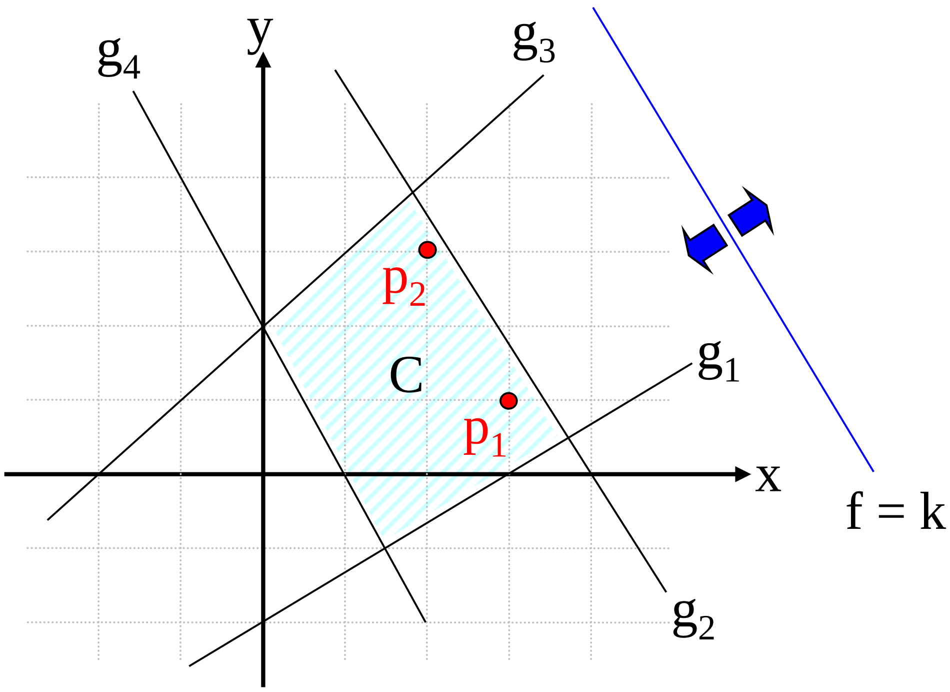
Integer Linear Programming (ILP) Problem

- Fact:
  - Many EDA problems are best formulated with integer variables
    - e.g. signal values in a digital circuit are under a modular number system
    - e.g. problems that need to enumerate the possible cases, or are related to scheduling of certain events
  - In general more difficult than the LP counterpart

An ILP Example

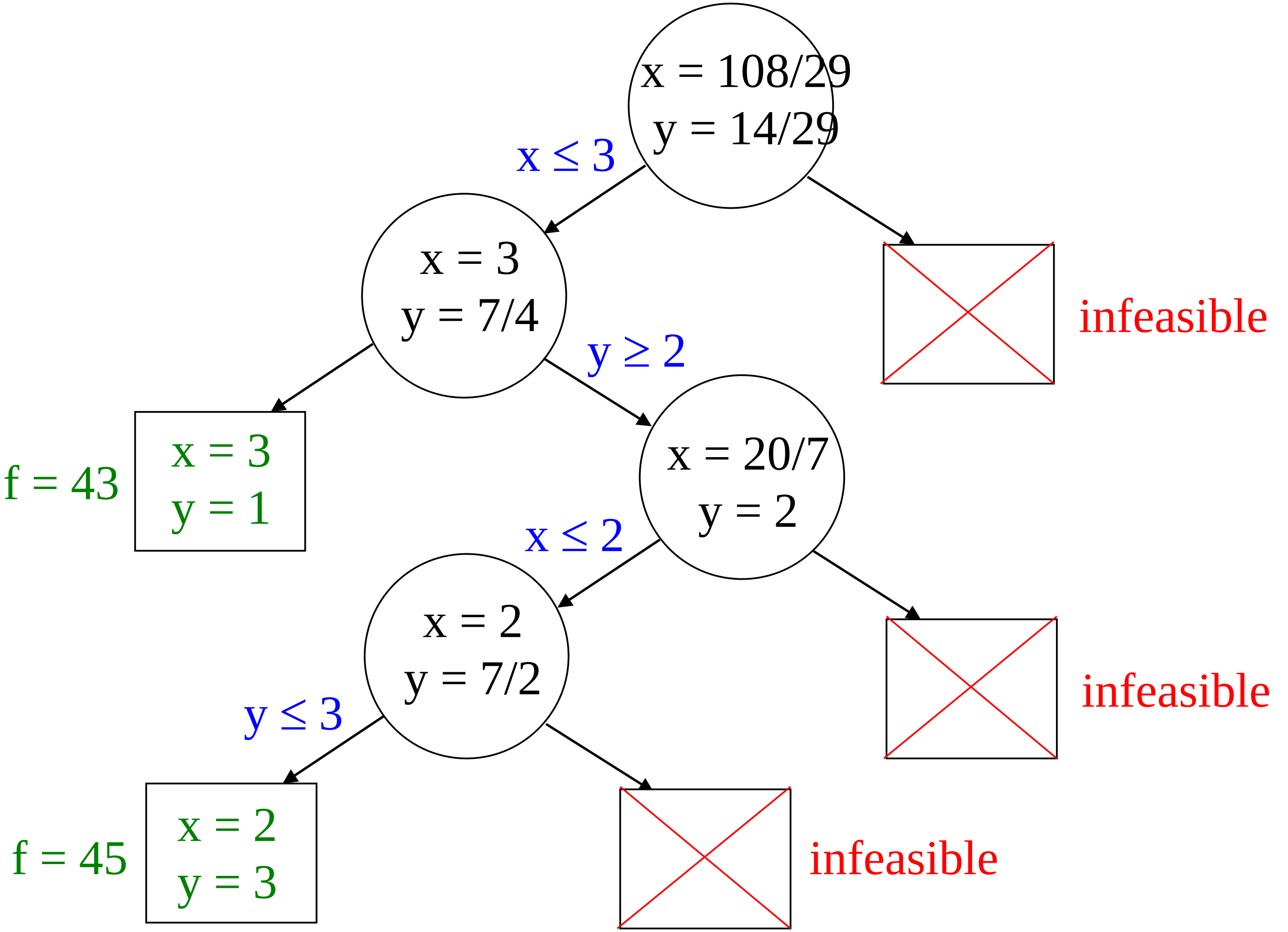
- maximize
  - f: $12x + 7y$
- subject to
  - g_1: $2x - 3y \leq 6$
  - g_2: $7x + 4y \leq 28$
  - g_3: $-x + y \leq 2$
  - g_4: $-2x - y \leq 2$

  where $x, y \in Z$



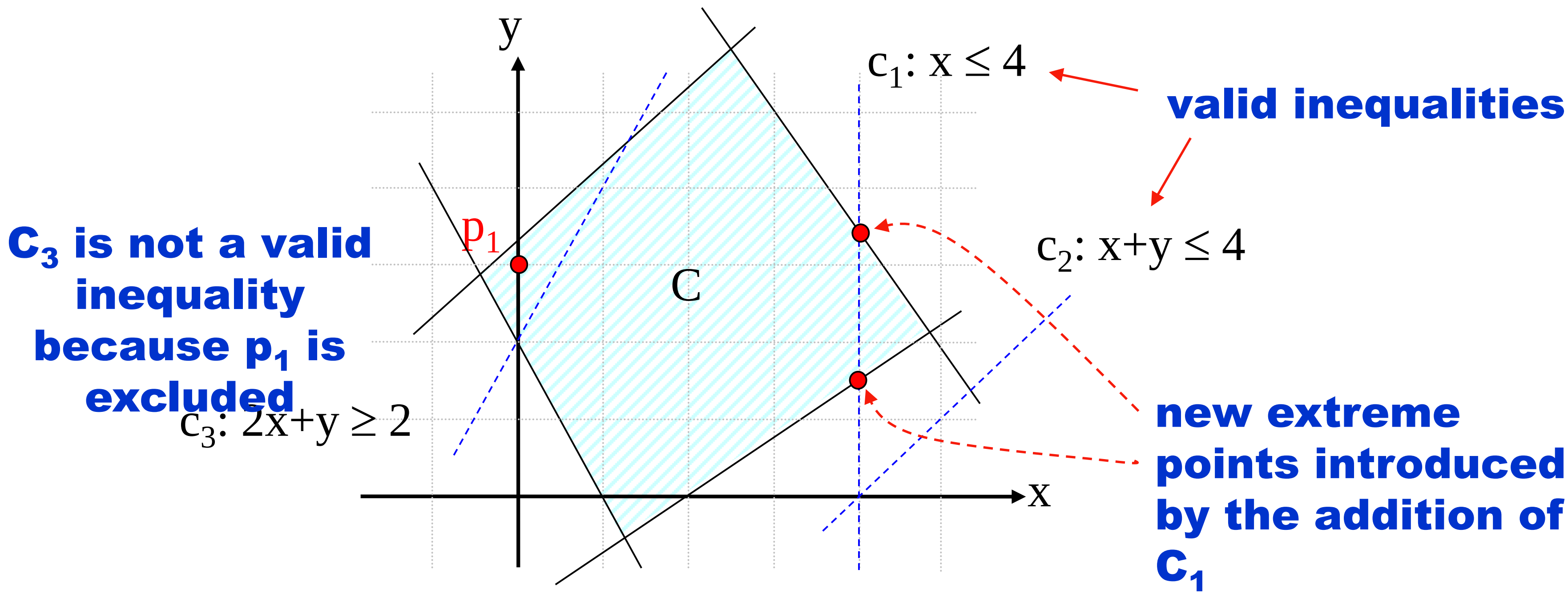
p_1 and p_2 are two possible points for optimal solution

LP Relaxation and Branch-and-Bound Procedure



Cutting Plane Algorithm

- Iteratively adds valid inequalities to the original problem in order to narrow the search area enclosed by the constraints while retaining the feasible points



Interior-Point Method

- An effective methods that can solve the convex optimization problems in polynomial time within a reasonably small number of iterations
- **Idea**: by introducing a *barrier function*, the original problem is rewritten into an *equality formula* so that Newton's method can be applied to find the optimal solution

Interior-Point Method

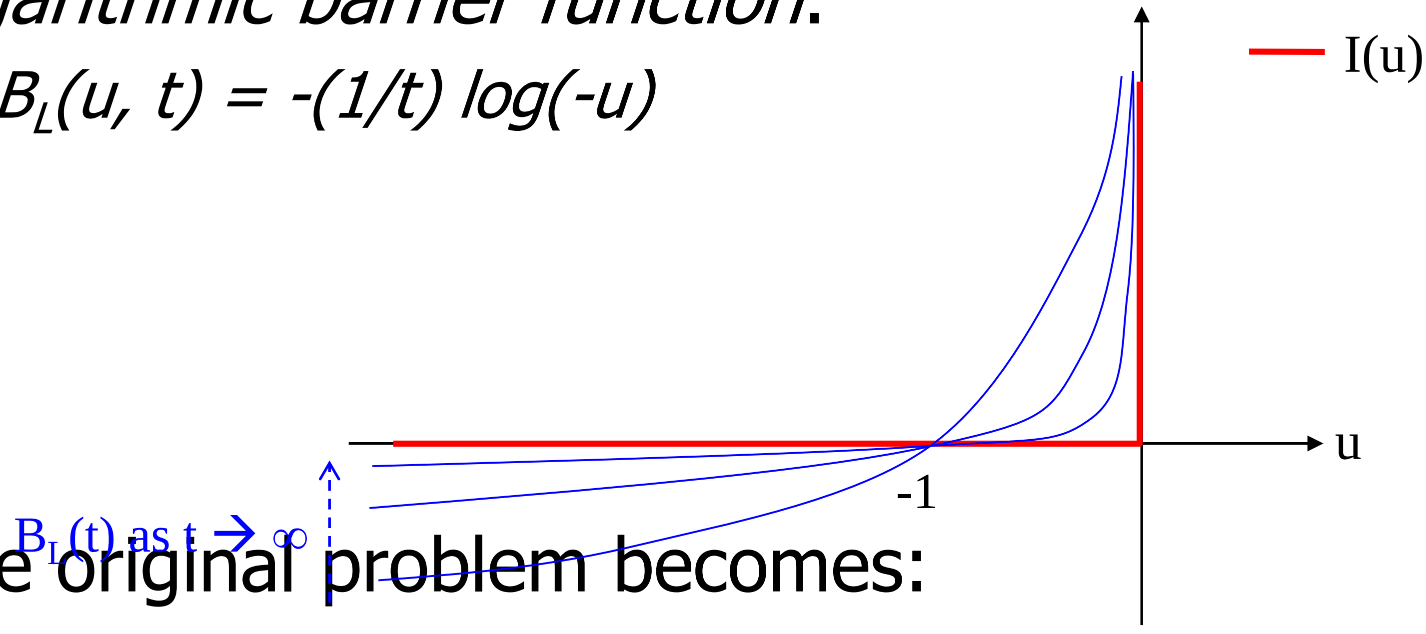
- *Indicator function $I(u)$:*
  - $I(u) = 0$ if $u \le 0$,
  - $I(u) = \infty$ otherwise
- The original problem can be rewritten as:

$$\min\left(f(x) + \sum_{1}^{m} I\left(g_i(x)\right)\right)$$

➔ However, this is not twice differentiable (near u = 0), so Newton's method cannot work

Interior-Point Method

- *logarithmic barrier function*:
  - —$B_L(u, t) = -(1/t) \log(-u)$



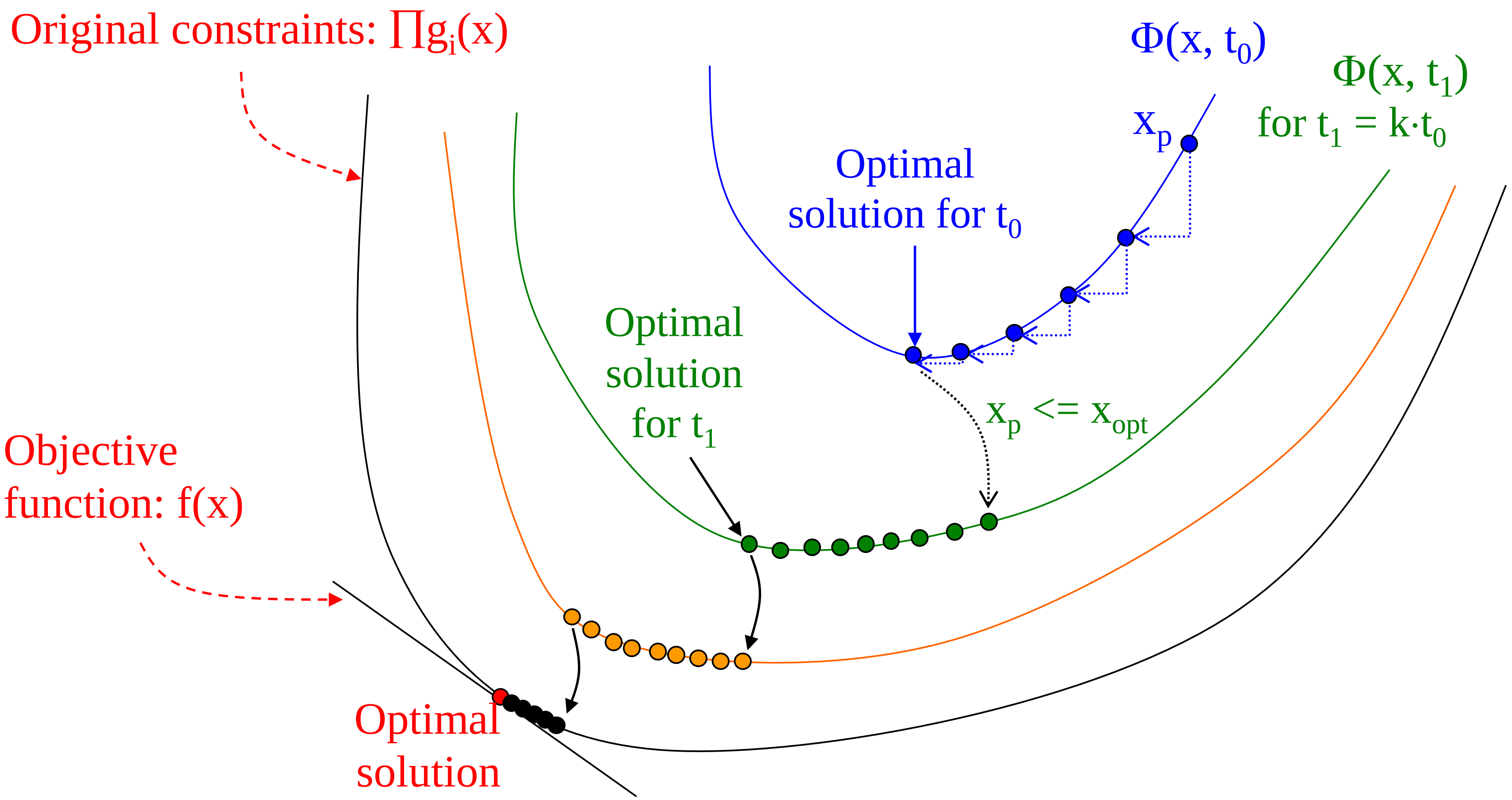
- The original problem becomes:

$$\min\left(f(x) + \sum_{1}^{m} -(1/t)\log(-g_i(x)) \right)$$

Interior-Point Method

1. Let $\Phi(x, t) = \min\left(f(x) + \sum_{1}^{m} -(1/t)\log(-g_i(x)) \right)$
2. Given initial t, tolerance e;
3. Find an interior feasible point x_p *s.t.* $\forall i. g_i(x_p) < 0$
4. Starting from x_p, apply Newton's method to find the optimal solution x_{opt}
5. If (1/t < e) return optimality as { x_{opt}, $\Phi(x_{opt}, t)$ };
6. Let $x_p = x_{opt}$, $t = k \cdot t$ for $k > 1$, repeat 4

An illustration of the interior point method



Convex program

- Convex programming problem

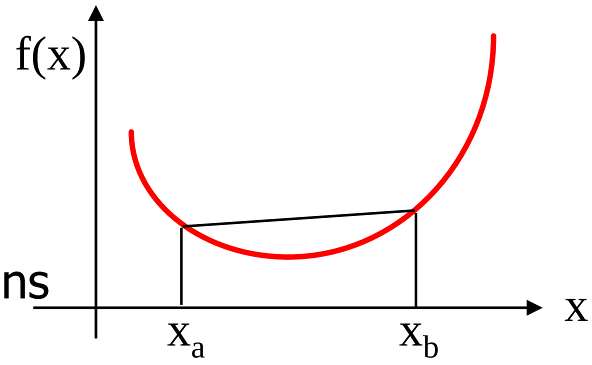
minimize $f(\underline{\mathbf{x}})$

such that $\cap\,[g_i(\underline{\mathbf{x}}) \leq c_i]$

where f and all g_i's are convex functions

- Any local minimum is a global minimum

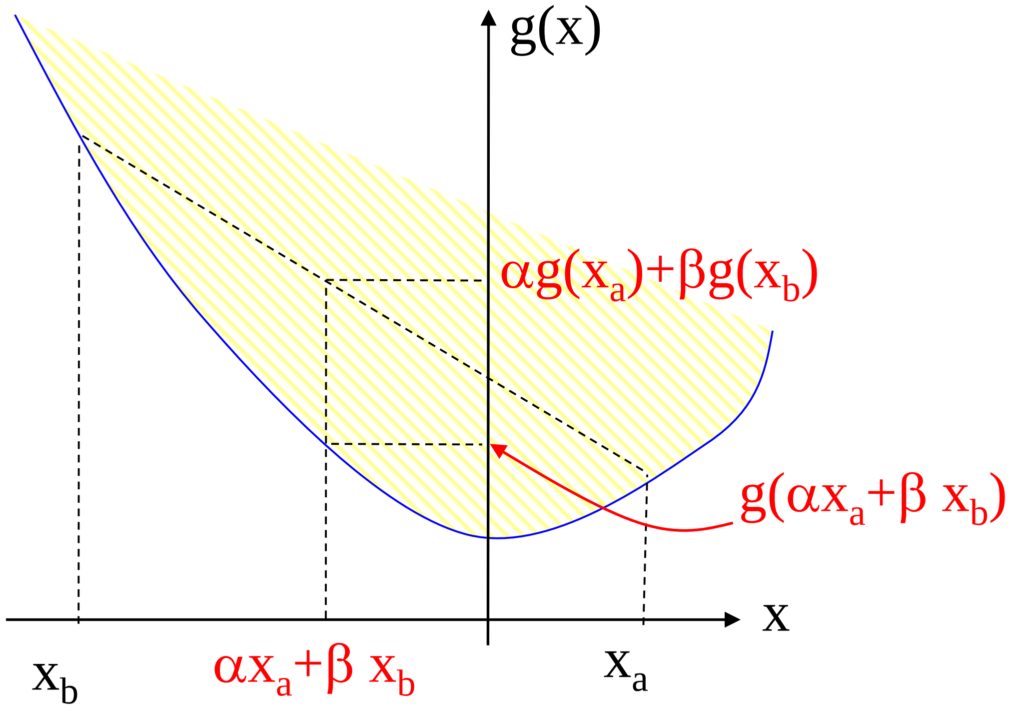
  (Nonrigorous) explanation:



- Linear program: $f(\underline{\mathbf{x}})$, $g_i(\underline{\mathbf{x}})$ are linear [affine] functions

Convex Optimization Problem

- Convexity property
  - $g_i(\alpha x_a + \beta x_b) \le \alpha\, g_i(x_a) + \beta\, g_i(x_b)$
  - For a convex function, a local optimal solution is also a global optimal solution



Linear Programming (LP)

Linear Programming

- Linear programming (LP) formulation requires the objective function and all constraints to be linear relationships (equalities or inequalities).
  - ILP (integer linear programming)
  - ZOLP (zero-one linear programming)
- General Form:

$$\text{Minimize} \quad c^T x$$

$$\text{Subject to:} \quad A^T x \geq b$$

where: $c^T = [c_1 \; c_2 \; \cdots \; c_n]$

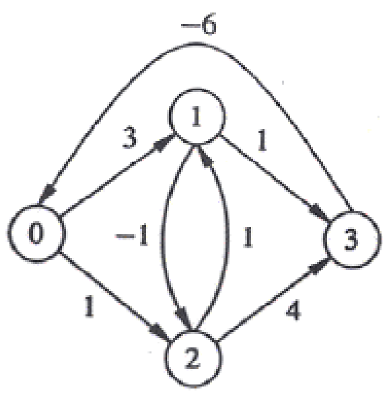
$$x \in \mathbb{R}^n \; ; \; x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

$$b \in \mathbb{R}^m \; ; \; b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}$$

$$A \in \mathbb{R}^{n \times m} \; ; \; A^T = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ \vdots & & & \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}_{m \times n}$$

Example

- Shortest path problem using ILP



Minimize $x_0 + x_1 + x_2 + x_3$

S.t.:

$$x_1 \geq x_0 + 3 \qquad x_1 - x_0 \geq 3$$
$$x_1 \geq x_2 + 1 \qquad x_1 - x_2 \geq 1$$
$$x_2 \geq x_0 + 1 \qquad x_2 - x_0 \geq 1$$
$$x_2 \geq x_1 - 1 \qquad x_2 - x_1 \geq -1$$
$$x_3 \geq x_1 + 1 \qquad x_3 - x_1 \geq 1$$
$$x_3 \geq x_2 + 4 \qquad x_3 - x_2 \geq 4$$
$$x_0 \geq x_3 - 6 \qquad x_0 - x_3 \geq -6$$

- $c^T = [1 \; 1 \; 1 \; 1]$
- $x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}$
- $b = \begin{bmatrix} 3 \\ 1 \\ 1 \\ -1 \\ 1 \\ 4 \\ -6 \end{bmatrix}$

$$\mathbf{A} = \begin{bmatrix} -1 & 0 & -1 & 0 & 0 & 0 & +1 \\ +1 & +1 & 0 & -1 & -1 & 0 & 0 \\ 0 & -1 & +1 & +1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & +1 & +1 & -1 \end{bmatrix}$$