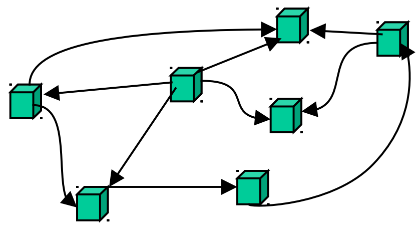
Chap. 9 Graph Algorithms



9.1 Definitions

– Basic Concepts

- A *graph*: $G = (V, E)$

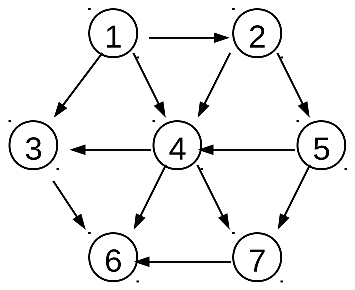
  where V is a set of *vertices*, E is a set of *edges*. Each edge is defined as a pair (v, w) with $v, w \in V$.
- If the pairs that define the edges in E are ordered, G is a *directed* graph (or: digraph)
- If $(v, w) \in E$, w is said to be *adjacent* to v. In an undirected graph, w is adjacent to v if and only if v is adjacent to w.
- A value (*weight*) can be assigned to each edge.
- A *path* in G is a sequence of connected vertices $w_1, w_2, \ldots, w_N \in V$, *i.e.* $(w_i, w_{i+1}) \in E\ (1 \leq i < N)$. The length of the path is $N-1$.
- A zero length path is a path from vertex v to itself (without the edge (v, v), which is called a *loop*).

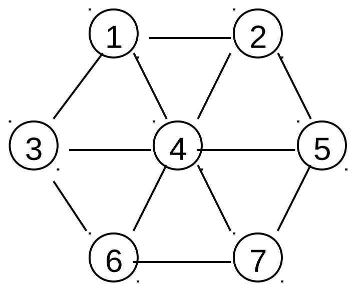
- A simple path is a path in which all vertices are distinct except the *1st* and the last vertices.
- A *cycle* is a path such that $w_1 = w_N$ $(N > 1)$. For an undirected graph, the edge in a cycle are required to be distinct.
- A directed graph is *acyclic* if it has no cycles, and it is called *DAG* (directed acyclic graph).
- A graph is *connected* if there is a path from every vertex to every other vertex.

  If a connected graph is also directed, the graph is said to be *strongly connected*.

  For a directed graph, if the connectivity can only be defined with undirected paths, it is said to be *weakly connected*.
- A complete graph is a graph in which there is an edge between every pair of vertices.
- Example: Airport connections.

– Graph Representations





– Adjacency Matrix

- A 2D $N \times N$ array

$$A[i][j] = \begin{cases} 1: \text{ if there is edge } (i, j) \\ 0: \text{ otherwise} \end{cases}$$

- Weights of the edge can also be stored in the matrix.
- Memory requirement: $\Theta(|V|^2)$
- A good representation for dense graphs, but inefficient if the graph is sparse.

directed

$$\begin{array}{c|ccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ \hline 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 3 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 4 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 5 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 7 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{array}$$

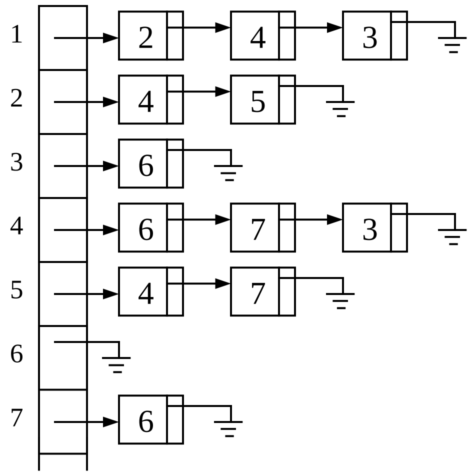
undirected

$$\begin{array}{c|ccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ \hline 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 3 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 4 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\ 5 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 6 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 7 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{array}$$

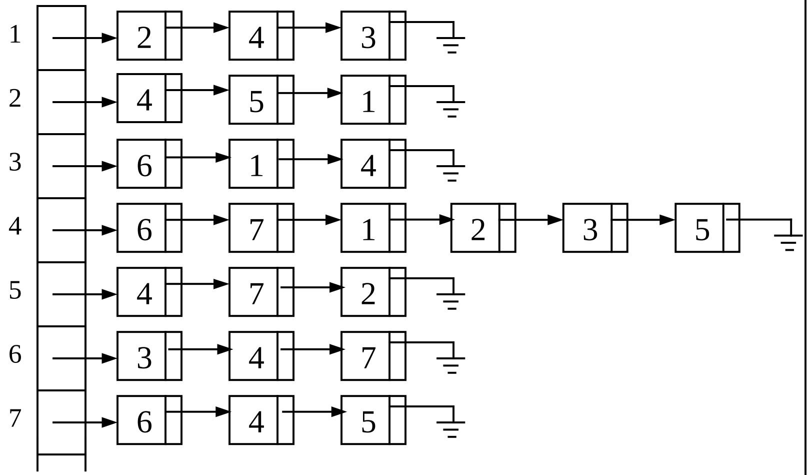
– Adjacency List

- A 1D array of linked lists.
  $A[u]$ is a list of all the vertices that are adjacent to u.
- Memory cost: $\Theta(|V| + |E|)$
- Weights of the edges can be stored in the nodes of the lists.
- For undirected graph, each edge appears in two lists (double memory cost)
- Hashing can be used to map the names of the vertices to the vertex nodes.

directed

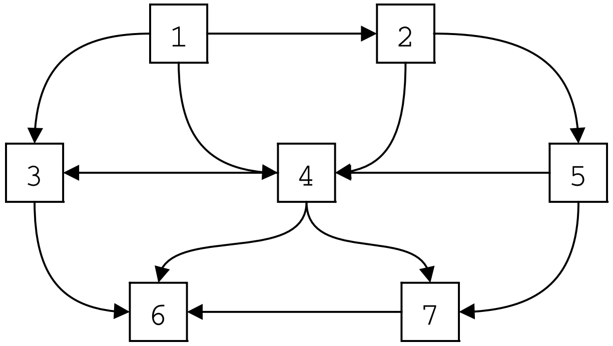


undirected



9.2 Topological Sort

- A ***topological Sort*** is an ordering of vertices in a directed acyclic graph (DAG) such that if there is a path from v_i to v_j, then v_j appears after v_i in the ordering.
- *Topological order* is not possible if there is a cycle in the graph.
- Topological ordering is not unique.
- Example
  - course prerequisite graph (Fig. 9.3)
  - Fig. 9.4: { $1\ 2\ 5\ 4\ 3\ 7\ 6$ or $1\ 2\ 5\ 4\ 7\ 3\ 6$

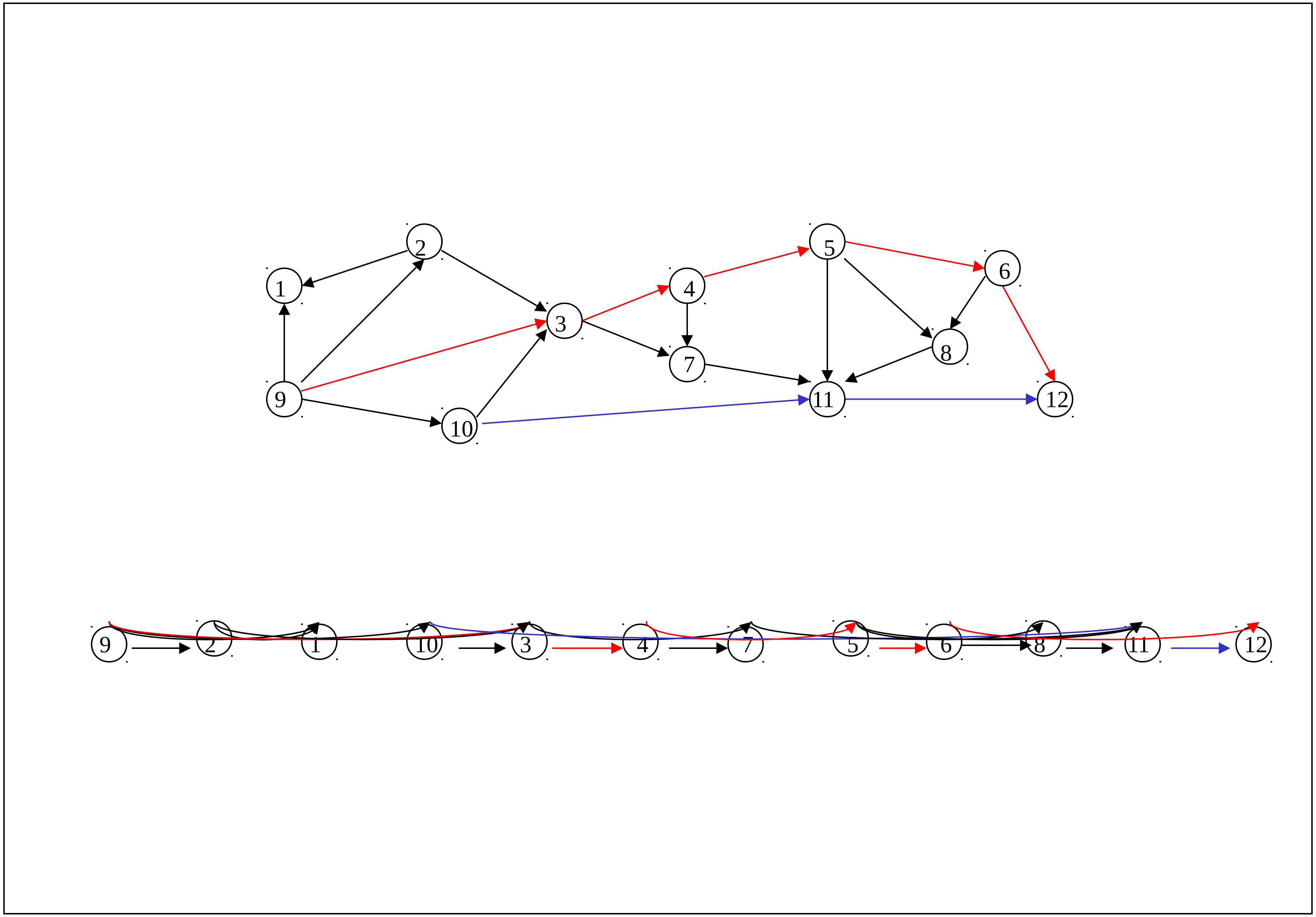


- A simple algorithm
  - *indegree* of vertex v: the number of edges of form (u, v)
  - topological sorting algorithm:

    a.) compute the indegrees of all vertices

    b.) remove a vertex with 0 indegree and its associated edges

    c.) repeat b.) until all vertices are sorted.


1
2
3
4
5
6
7
8
9
10
11
12
9
2
1
10
3
4
7
5
6
8
11
12

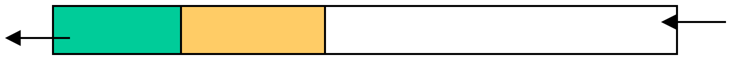
- Example:

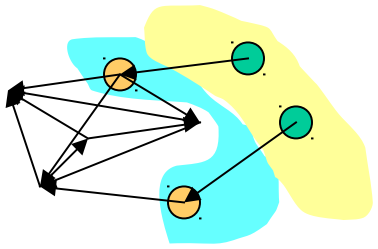
```
topSort ( )
{
  for (counter = 0; counter < no_vertices;counter++) {
            v = findNewVertexOfIndegreeZero ( );
            if (v == NOT_A_VERTEX)
                   found_cycle_error;
            v.topNum = counter;
            for (each w adjacent to v)
                   w.indegree--;
  }
}  //  [Fig. 9.5]
```

- running time: $O(|V|^2)$

– A more efficient algorithm

- Putting all *0* indegree vertices in a *box*.
- New *0* indegree vertices can be deleted when removing the edges of a *0* indegree vertex.

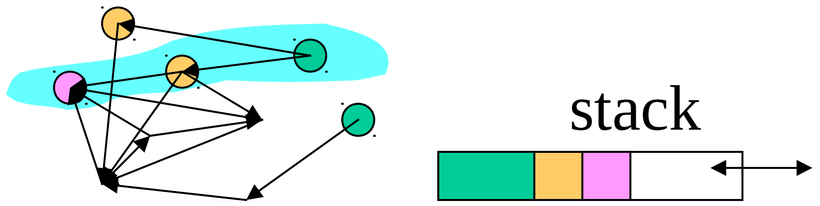


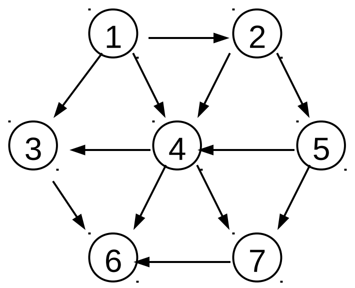


- If the box is implemented by a queue, the algorithm is breadth-first.

- The algorithm: `(Fig. 9.7)`
- Running time $O(|E| + |V|)$ if adjacent list is used
- Example: `(Fig. 9.6)`

	1	*2*	*3*	*4*	*5*	*6*	*7*
v1	0	0	0	0	0	0	0
v2	1	0	0	0	0	0	0
v3	2	1	1	1	0	0	0
v4	3	2	1	0	0	0	0
v5	1	1	0	0	0	0	0
v6	3	3	3	3	2	1	0
v7	2	2	2	1	0	0	0
Enqueue	v1	v2	v5	v4	v3,v7		v6
Dequeue	v1	v2	v5	v4	v3	v7	v6





- If the box is a stack, the algorithm is depth-first.

9.3 Shortest-Path Algorithms

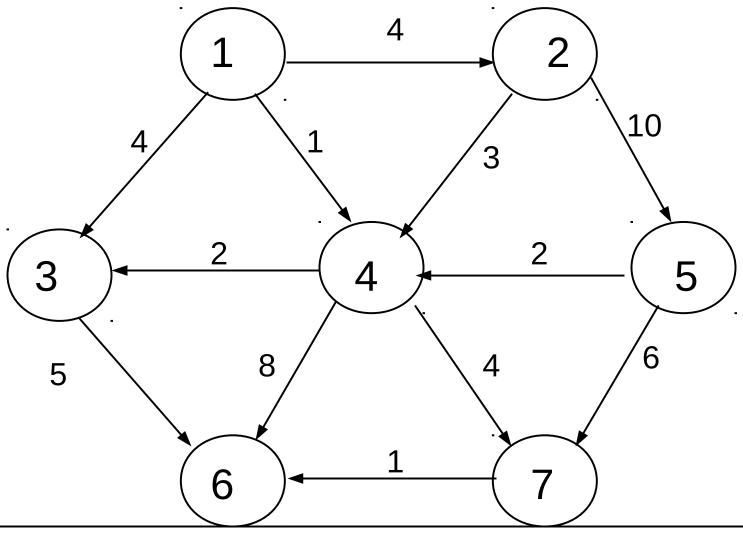
– The problem

- Input: a weighted graph

  i.e. each edge (v_i, v_j) has a cost C_{ij}
- the cost of path $v_1\ v_2\ \ldots v_N$ is $\sum_{i=1}^{N-1} C_{i,i+1}$, also called *weighted path length.*
- *unweighted path length*: $N - 1$
- single-source shortest path problem:

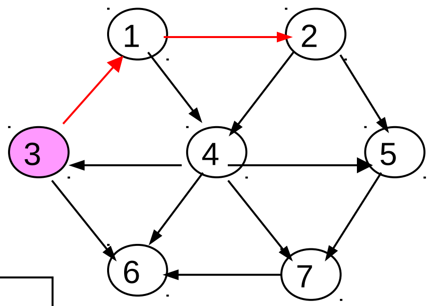
  given as input a weighted graph $G = (V, E)$ and a distinguished vertex, S, find the shortest weighted path from S to every other vertex in G
- example

  $C(v_1, v_6) = 6$



- Applications
  - Network transmission
  - Air travel route

- The single destination problem: find the shortest path from source *s* to a given destination vertex *t*.
  - Currently, there is no algorithms that can find the single destination shortest path any faster (by more than a constant factor) than the shortest paths to all vertices.



- Unweighted shortest paths
  - A special case of weighted shortest path
  - data structure (table)

v:

Known: (T/F)	Dist: distance to s	Path: previous vertex

  - basic idea

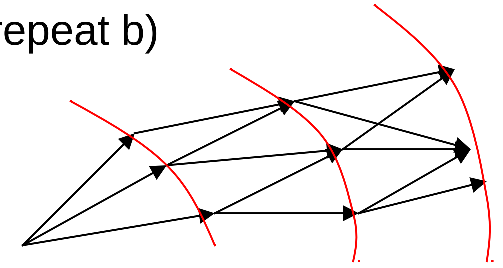
    a) Starting from vertex *s*, let `s.dis=0` and `x=s`

    b) Mark `dist=x.dist+1` for all vertices that are adjacent to *x*.

    c) For each of the vertices, *x*, that are just marked, repeat b)

    d) repeat c) until all vertices are marked.

  - Breadth-first search

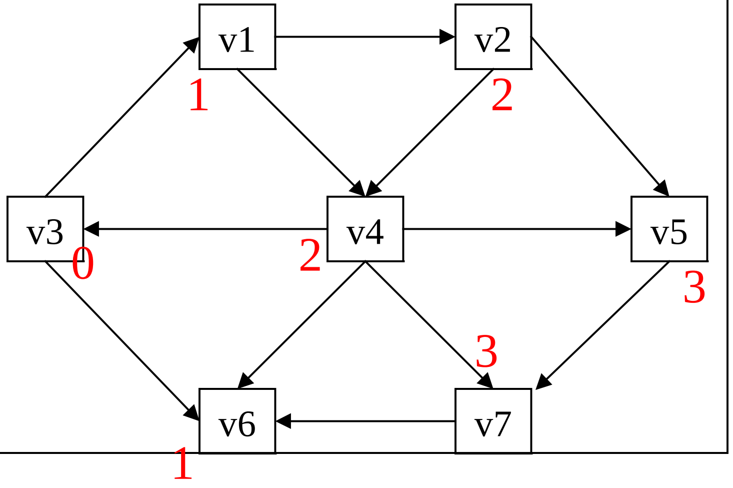


- Algorithm

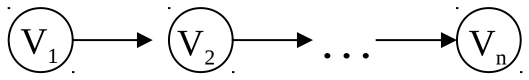
```
unweighted (vertex s)
{
   s.dist = 0
   for (k = 0; k < |V| ; k++)
      for (each v ∈ V)
      if (!v.known && v.dist == k){
         v.known = TRUE
         for (each w adjacent to v)
            if (w.dist == INFINITY){
                 w.dist = k + 1
                 w.path = v
            }
      }
}
```

v	known	dv	pv
v1	F	99999	0
v2	F	99999	0
v3	F	0	0
v4	F	99999	0
v5	F	99999	0
v6	F	99999	0
v7	F	99999	0

- example: (Fig. 9.11 - 9.14)

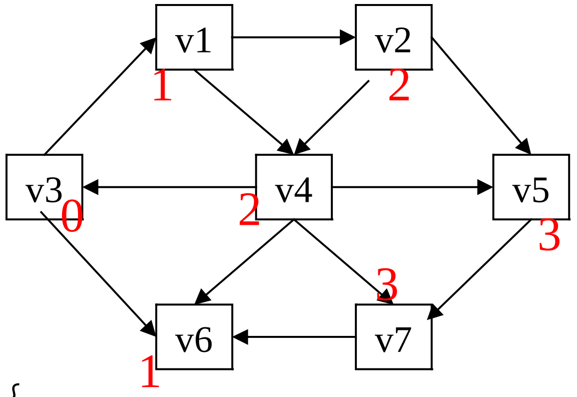


- running time: $O(|V|^2 + |E|)$
- worst case:



- Tracking “path” backward gives the actual shortest path.
- A better algorithm: using a queue to store and process the vertices in order — avoiding searching through V
- Algorithms

```
unweighted-better (vertex s)
 {
   Queue q(|V|);
       q.enqueue(s)
       S.dist=0;
       while (!q.IsEmpty( ))   {
         v = q.dequeue( );
         for (each w adjacent to v)  {
                if (w.dist == INFINITY){
                  w.dist = v.dist + 1;
                  w.path = v;
                  q.enqueue (w);
                 }
             }
         }
   }
```



- running time: $O(|V| + |E|) = O(|E|)$
- example: (Fig. 9.19)

v	known	dv	pv
v1	T	1	v3
v2	T	2	v1
v3	T	0	0
v4	T	2	v1
v5	T	3	v2
v6	T	1	v3
v7	T	3	v4

v3 | **v1, v6** | **v6, v2, v4,** | **v2, v4,** | **v4, v5,** | **v5, v7** | **v7**

v	known	dv	pv
v1	F	99999	0
v2	F	99999	0
v3	F	0	0
v4	F	99999	0
v5	F	99999	0
v6	F	99999	0
v7	F	99999	0

v	known	dv	pv
v1	T	1	v3
v2	F	2	v1
v3	T	0	0
v4	F	2	v1
v5	F	99999	0
v6	F	1	v3
v7	F	99999	0

v	known	dv	pv
v1	F	1	v3
v2	F	99999	0
v3	T	0	0
v4	F	99999	0
v5	F	99999	0
v6	F	1	v3
v7	F	99999	0

v	known	dv	pv
v1	T	1	v3
v2	F	2	v1
v3	T	0	0
v4	F	2	v1
v5	F	99999	0
v6	T	1	v3
v7	F	99999	0

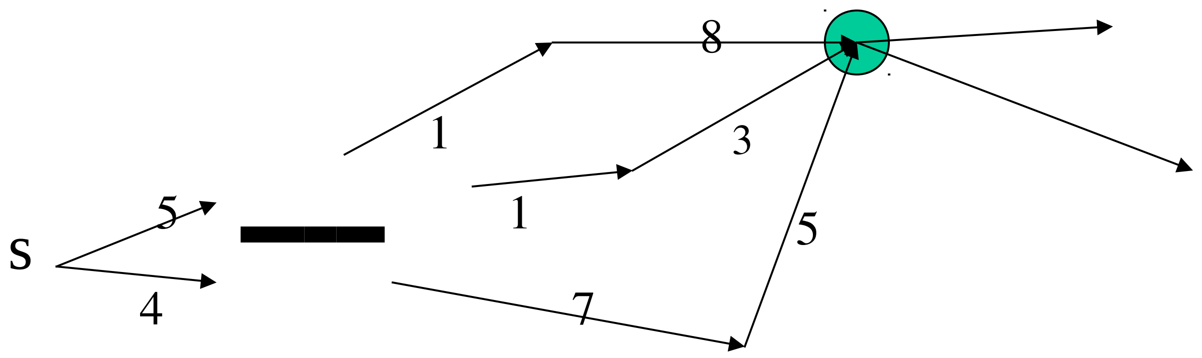
v	known	dv	pv
v1	T	1	v3
v2	T	2	v1
v3	T	0	0
v4	F	2	v1
v5	F	3	v2
v6	T	1	v3
v7	F	99999	0

v	known	dv	pv
v1	T	1	v3
v2	T	2	v1
v3	T	0	0
v4	T	2	v1
v5	T	3	v2
v6	T	1	v3
v7	F	3	v4

v	known	dv	pv
v1	T	1	v3
v2	T	2	v1
v3	T	0	0
v4	T	2	v1
v5	F	3	v2
v6	T	1	v3
v7	F	3	v4

v	known	dv	pv
v1	T	1	v3
v2	T	2	v1
v3	T	0	0
v4	T	2	v1
v5	T	3	v2
v6	T	1	v3
v7	T	3	v4

- Weighted shortest path
    - Mark vertices as known or unknown



- Greedy algorithm
    - greedy strategy: taking currently best solution at each step.
    - similar idea to the unweighted algorithm.

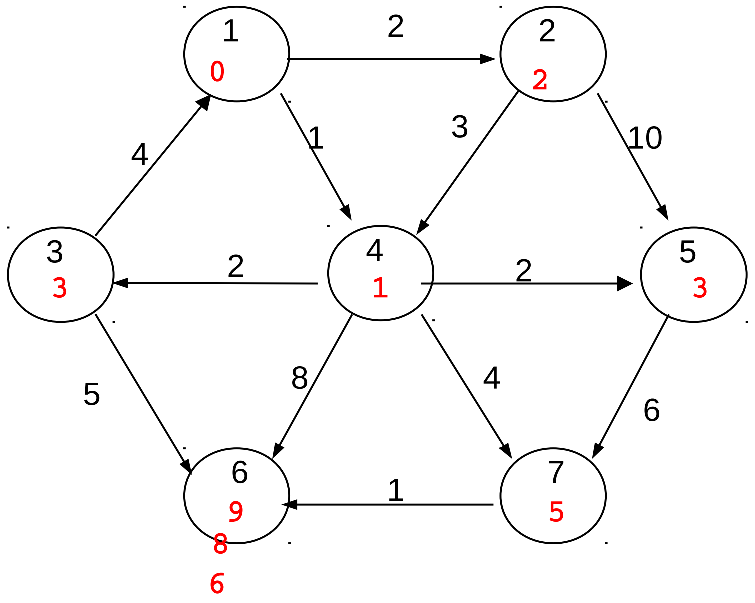
      At each stage, find the shortest unknown distance vertex *v*, and update its adjacent vertices: `w.dist=v.dist+c(v,w)`

      if it is an improvement over the previous value in *w*.
    - example: `(Fig. 9.20 - 9.28)`

v	known	dv	pv
v1	T	0	0
v2	T	2	v1
v3	T	3	v4
v4	T	1	v1
v5	T	3	v4
v6	T	6	v7
v7	T	5	v4

- The algorithm:

```
greedy (Vertex s){
    s.dist = 0;
V   while ( ) {
V      v = smallest unknown distance vertex;
       if (v == NOT_A_VERTEX) break;
       v.known = TRUE;
E      for (each w adjacent to v){
         if(!w.known)
         if(v.dist + c(v, w) < w.dist){
            w.dist = v.dist + c(v, w);
            w.path = v;
         }
       }
    }
}
```



v	known	dv	pv
v1	F	0	0
v2	F	99999999	0
v3	F	99999999	0
v4	F	99999999	0
v5	F	99999999	0
v6	F	99999999	0
v7	F	99999999	0

v	known	dv	pv
v1	T	0	0
v2	F	2	v1
v3	F	3	v4
v4	T	1	v1
v5	F	3	v4
v6	F	9	v4
v7	F	5	v4

v	known	dv	pv
v1	T	0	0
v2	F	2	v1
v3	F	99999999	0
v4	F	1	v1
v5	F	99999999	0
v6	F	99999999	0
v7	F	99999999	0

v	known	dv	pv
v1	T	0	0
v2	T	2	v1
v3	F	3	v4
v4	T	1	v1
v5	F	3	v4
v6	F	9	v4
v7	F	5	v4

v	known	dv	pv
v1	T	0	0
v2	T	2	v1
v3	T	3	v4
v4	T	1	v1
v5	F	3	v4
v6	F	8	v3
v7	F	5	v4

v	known	dv	pv
v1	T	0	0
v2	T	2	v1
v3	T	3	v4
v4	T	1	v1
v5	T	3	v4
v6	F	6	v7
v7	T	5	v4

v	known	dv	pv
v1	T	0	0
v2	T	2	v1
v3	T	3	v4
v4	T	1	v1
v5	T	3	v4
v6	F	8	v3
v7	F	5	v4

v	known	dv	pv
v1	T	0	0
v2	T	2	v1
v3	T	3	v4
v4	T	1	v1
v5	T	3	v4
v6	T	6	v7
v7	T	5	v4

- Running time: $O(|E| + |V|^2) = O(|V|^2)$ *update + vertices * find min*
- For sparse graph, a priority queue can be used to:
  - a) select the smallest unknown distance vertex using *deleteMin( ).*
  - b) update w.dist using *decreaseKey( )* $\rightarrow$ *percolate up needed O(log|V|)*
  - c) an auxiliary data structure (*e.g.* hash table) is needed to find the heap locations of the adjacent vertices
- running time with a heap:

  $O(|E| \log |V| + |V| \log |V|) = O(|E| \log |V|)$
- The greedy algorithm is only correct when there is no negative weight.

– Acyclic graphs (downhill skiing problem)

- Changing the selection rule (the order vertices are declared known): selecting vertices in topological order.
- When a vertex is selected in topological order, its distance can no longer be lowered (no incoming edge from unknown vertices)
- No need for priority queue
- running time: $O(|E| + |V|)$
- example: critical path analysis

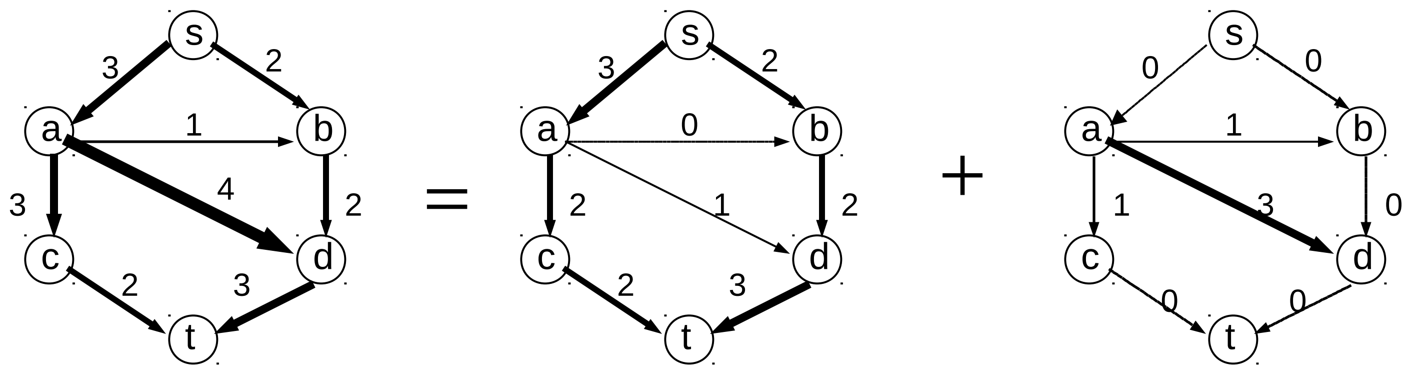
  (Fig. 9.34 - 9.38)

9.4 Network Flow Problem

– The problem

- Given a directed graph $G = (V, E)$, a <u>source</u> vertex $s \in V$, and a <u>*sink*</u> vertex $t \in V$,
  - a.) each edge $(v, w) \in E$ has an edge <u>*capacity*</u> C_{vw} representing the maximum units of flow that can pass through that edge, *i.e.* the actual flow f_{vw} needs to satisfy $f_{vw} \leq C_{vw}$
  - b.) For vertex $v \in V$ and $v \neq s$, $v \neq t$ the total flow coming in must equal to the total flow going out.
- The maximum flow problem:

  determine the maximum amount of flow that can pass from s to t, satisfying conditions a.) and b.)



capacity graph flow graph ($C_{max} = 5$) residual graph

– A simple algorithm

- For a given directed graph $G = (V, E)$, define two additional graphs

  a.) flow graph G_f

  b.) residual graph G_r: the difference between capacity and the actual flow.

  Initially, all edges in G_f have 0 flow

- Basic idea

  a.) find a path from s to t, called augmenting path

  b.) find the minimum edge on the augmenting path

  c.) the value of the minimum edge in G_r is added to all edges on the path in G_f, and subtracted from all edges on the path in G_r

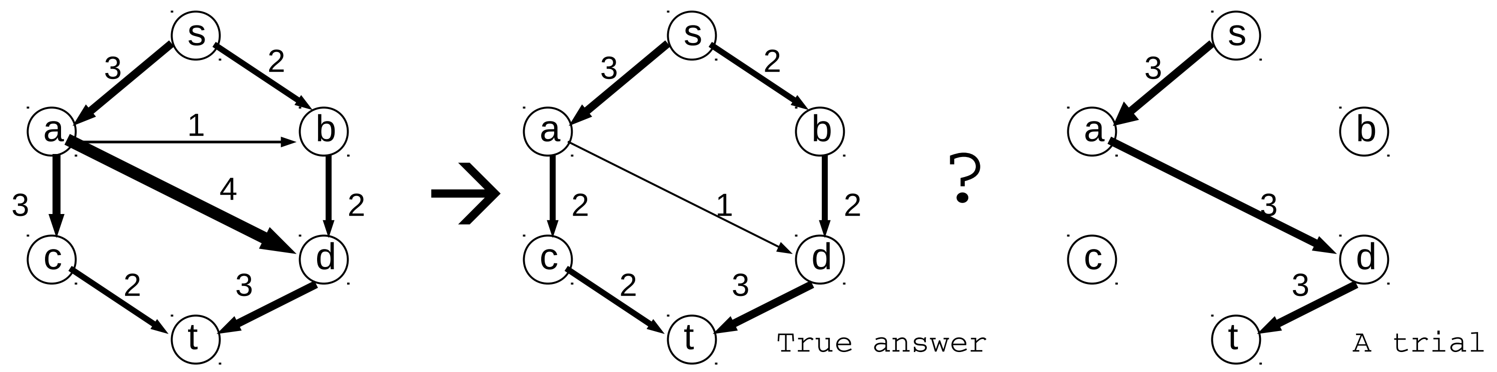
  d.) remove edges in G_r that have a zero value

  e.) repeat a.) - d.) until no more path can be found in G_r

- example: `(Fig. 9.40 - 9.43)`

$$G=G_{f1}+G_{r1} =G_{f1}+(G_{f2}+G_{r2}) = \ldots. = (G_{f1}+G_{f2}+\ldots)+G_{rn}$$

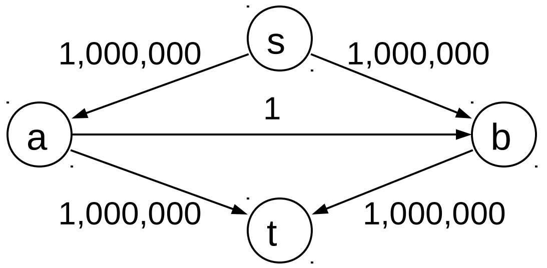
- A problem: (Fig.9.44)



- A solution: Every time a flow is added to an edge *(v, w)* in G_f, the same amount of flow is added to edge *(w, v)* in G_r — allowing the algorithm to undo previous decisions.
- example: (Fig. 9.45 - 9.46)
- Theorem: If all edges capacities are rational numbers, this algorithm always terminate with a maximum flow.

– Analysis

- If all capacities are integers, the running time is $O(f \cdot |E|)$, where f is the maximum capacity, ($O(|E|)$ is used to find each augmenting path using the unweighted shortest path algorithm)
- A bad example:

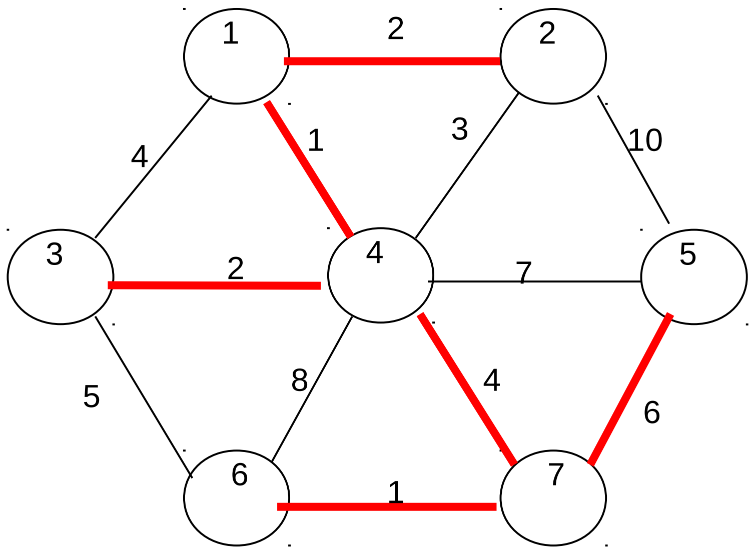


- A solution: finding the augmenting path that allows the largest increase in flow — a (slightly modified) weighted shortest path problem.
- running time: $O(|E| \log f)$ augmenting paths are needed

$\Rightarrow O(|E|^2 \log |V| \log f)$ running time.

9.5 Minimum Spanning Tree

- The problem
  - Consider undirected graph only
  - *Spanning tree*: a tree by graph edges that connect all the vertices of the graph.
  - *Minimum spanning tree*: a spanning tree with the minimum total edge cost.
  - Minimum spanning tree of graph G exists if and only if G is connected.
  - example: `(Fig. 9.48)`
  - tree: acyclic & undirected



– Prim’s algorithm

- a greedy algorithm
- basic steps: starting from an arbitrary vertex as the root. At any point of the algorithm, there is a set of vertices that can be included in the tree, and the rest are not. The algorithm, at each stage, find a new vertex, v, to add to the tree such that the cost of (u, v) is the smallest among all edges where u is in the tree and v is not.
- The same process as the greedy algorithm in shortest path problem, except that:

  a.) the graph is undirected

  b.) the “dist” is defined as the shortest cost of edge (u, v), with u currently in the tree (known) and v currently not (unknown)

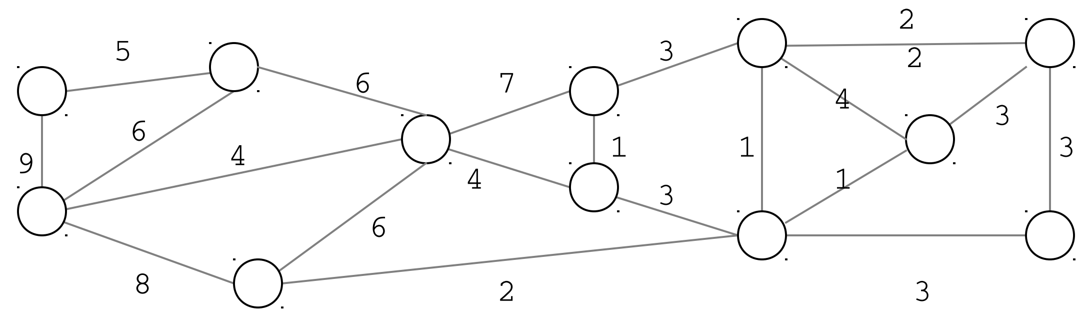
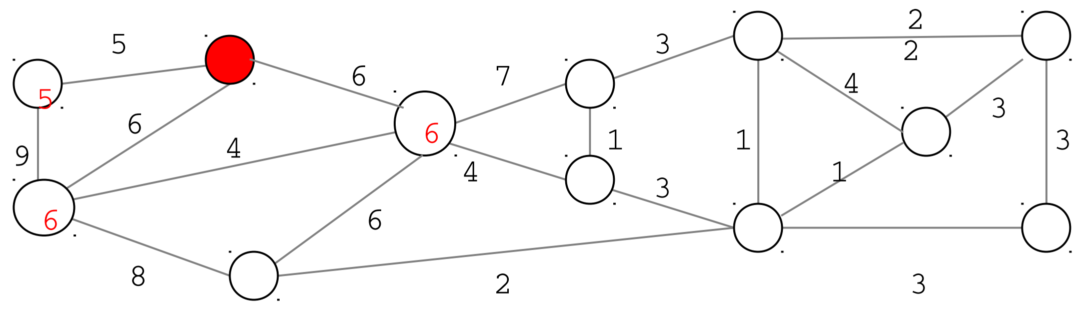
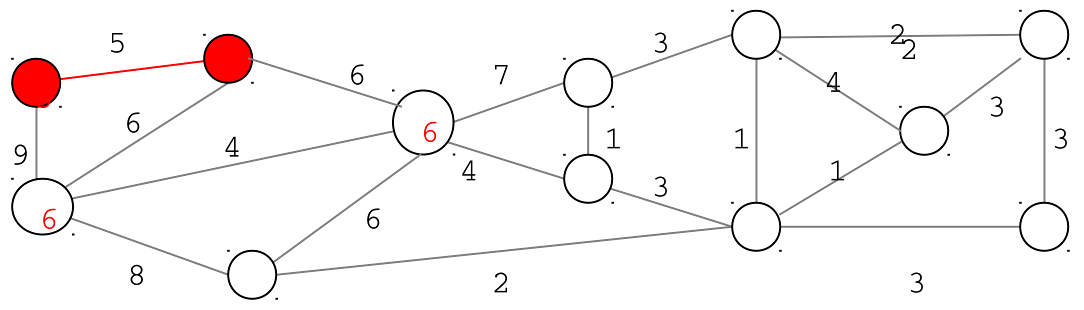
  c.) updating rule:

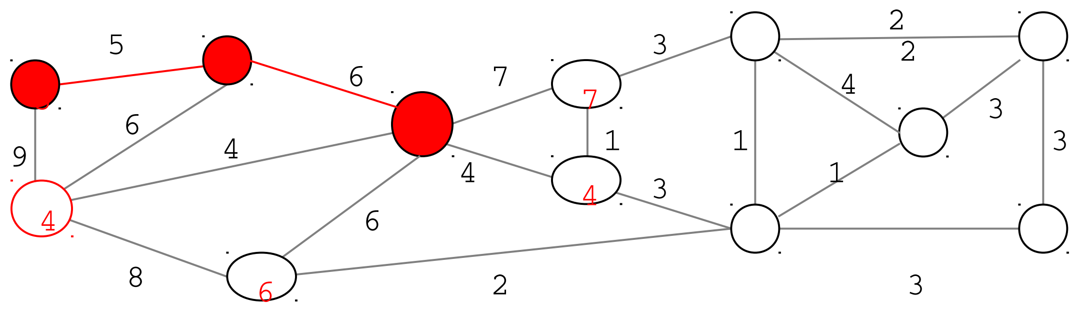
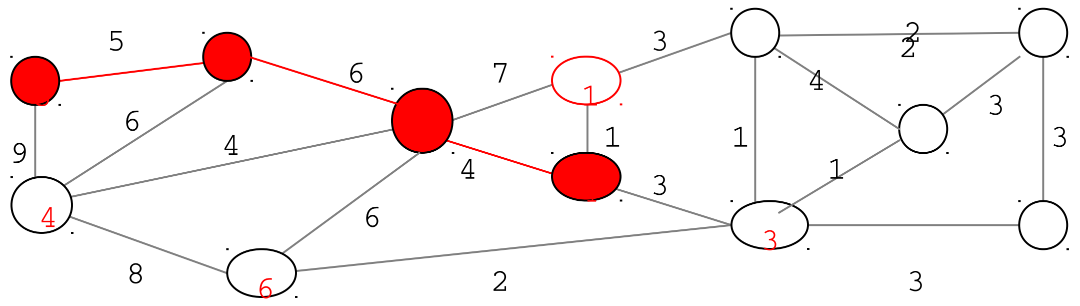
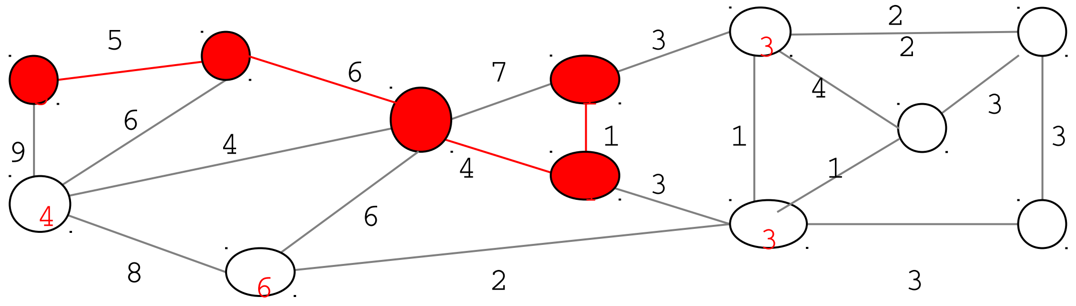
  ```
      for (each vertex w adjacent to v)
        w.dist = min(w.dist, c(w, v))
  ```

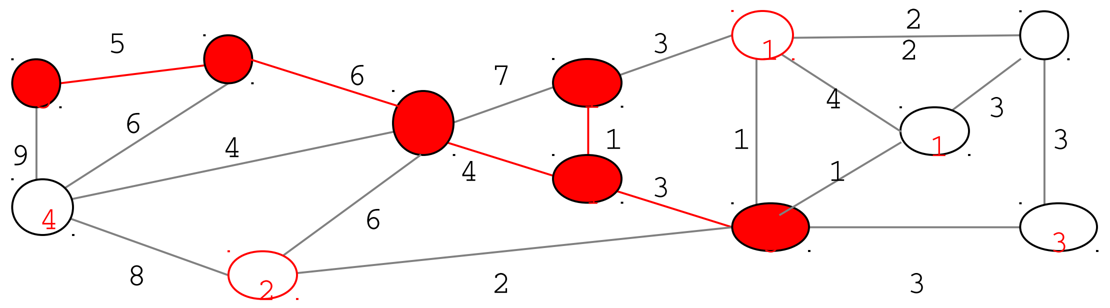
- running time: the same as the shortest path algorithm:

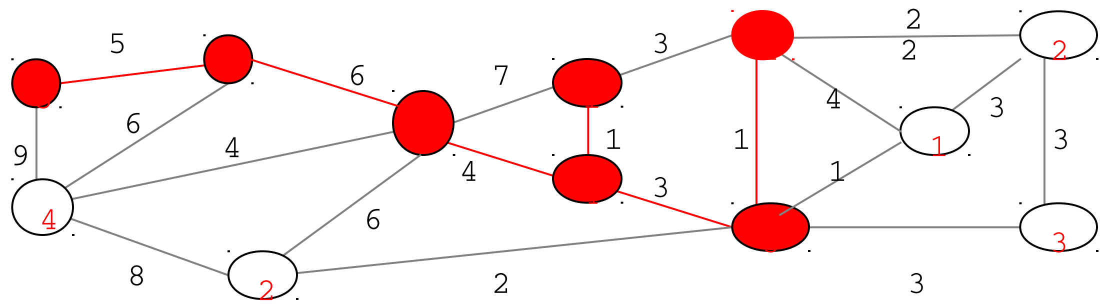
  $O(|V|^2)$: without heap

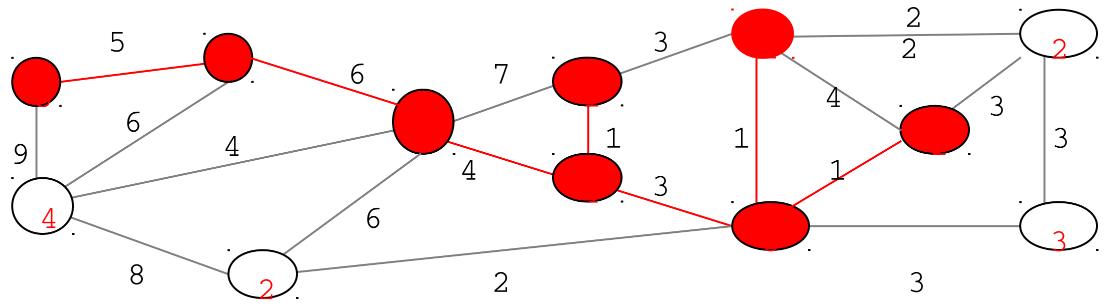
  $O(|E|\log |V|)$: using a binary heap.
- example: `(Fig. 9.48 - 9.55)`

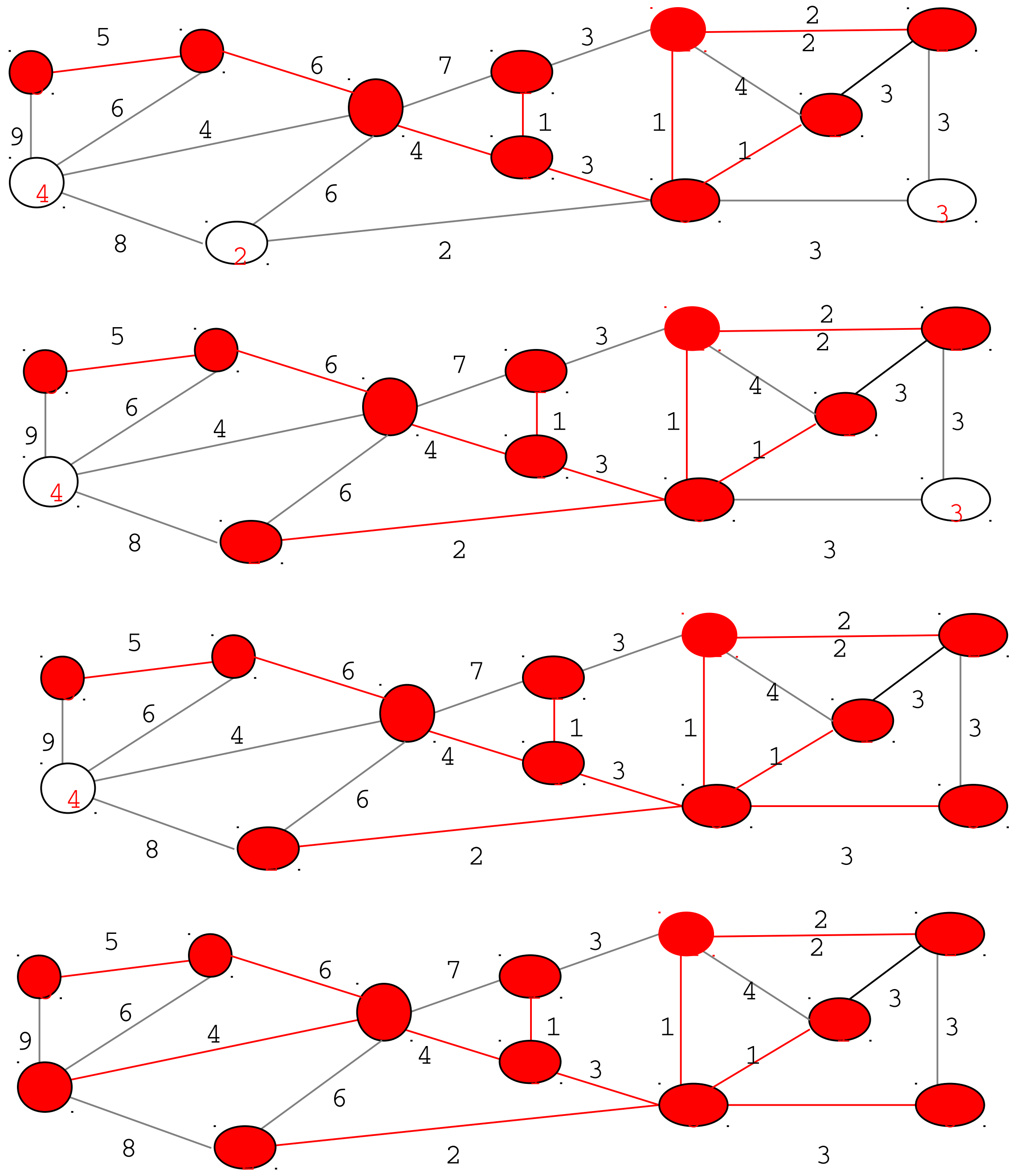





5
6
7
3
2
2
4
6
3
9
4
7
1
1
3
4
1
4
3
6
8
6
2
3

5
6
7
3
2
2
4
6
3
9
4
1
1
1
3
4
1
4
3
3
6
8
6
2
3

5
6
7
3
2
3
2
4
6
3
9
4
1
1
3
4
1
3
4
3
6
8
6
2
3





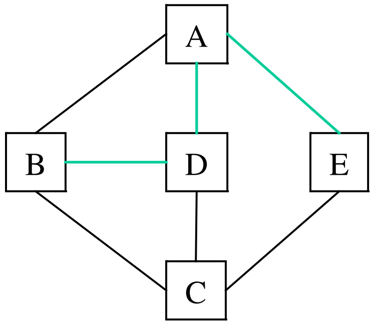


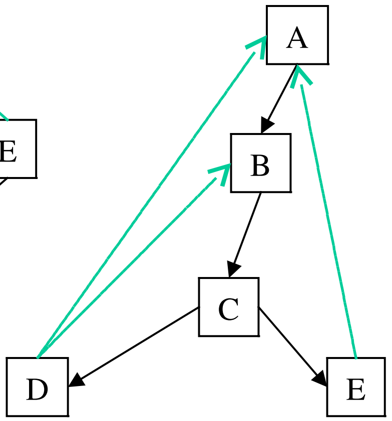


9.6 Depth-First Search

- Depth-first search for graphs
  - Similar to preorder traversal of trees, *i.e.* visit the current vertex first, then recursively search its adjacent vertices
  - To avoid cycles, mark each vertex that is already visited.
  - If the graph is unconnected, multiple search trees may be created.
- Depth-first search of undirected graphs
  - Algorithm:

```
dfs(vertex v)
{
  v.visited = TRUE;
  for (each w adjacent to v)
    if (!w.visited)  dfs(w);
}
```

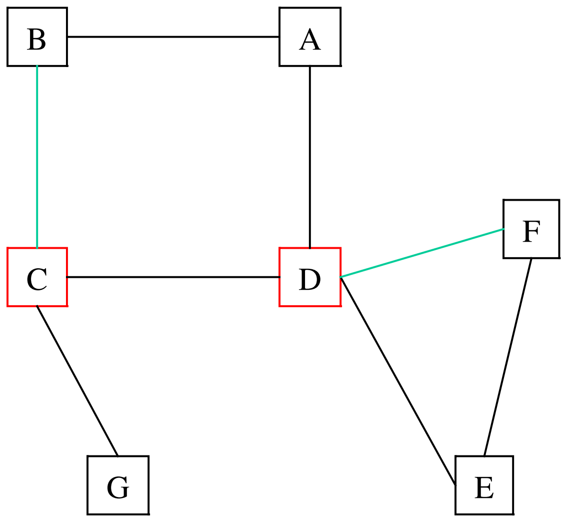


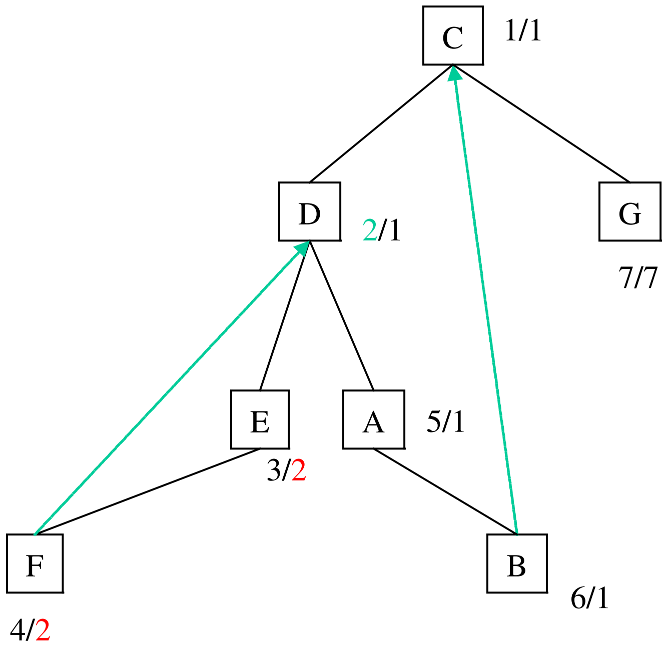


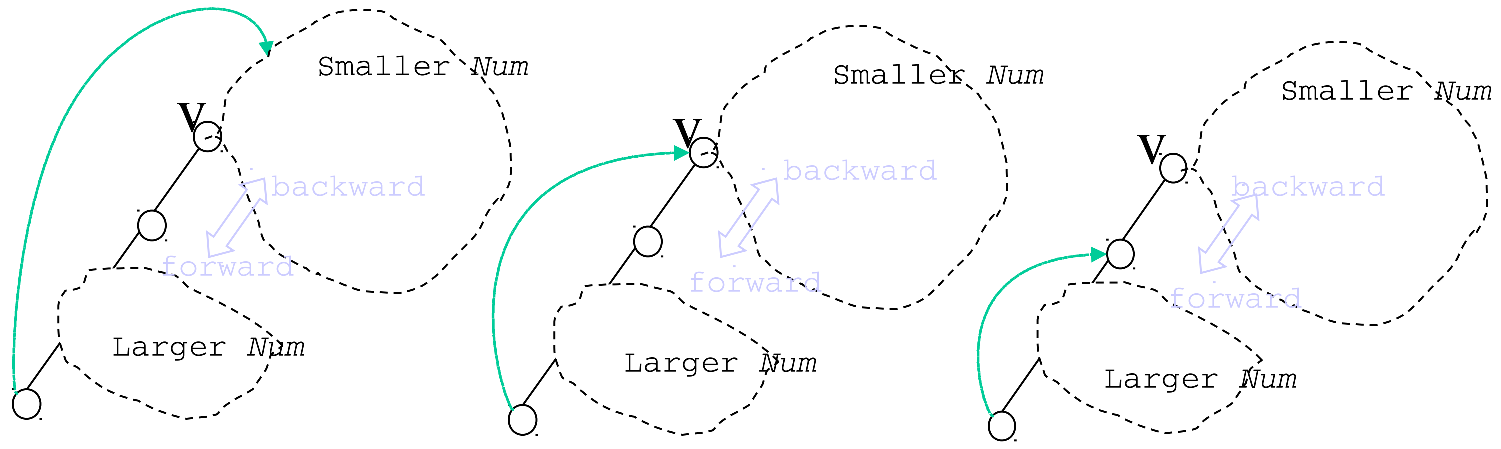
  - The search tree is directional (a spanning tree)
  - Edges of the graph that are not part of the search tree are marked as *back edges*.
  - Example: `(Fig. 9.60 - 9.61)`

– Biconnectivity

- Definition: a connected undirected graph is *biconnected* if there are no vertices whose removal disconnects the rest of the graph.
- Biconnectivity is required in Network connections, transportation network, electrical wiring, etc.
- A vertex in a non-biconnected graph that can disconnect the graph (by its removal) is called an *articulation point*.
- Example: `(Fig. 9.62)`







- *preorder number Num(v)*: the number given, in sequence, to each vertex visited in a depth-first searching process.
- *Lowest-numbered vertex Low(v)*: the lowest preorder number among vertices that are reachable from v by taking 0 or more tree edges and then possibly one back edge.
- Low(v) is the smaller of

  a.) Num(v)

  b.) the lowest Num(w) among all back edges (v, w)

  c.) the lowest Low(w) among all tree edges (v, w)

  i.e. Low(v) can be computed by a post-order traversal of the search tree

  $\Rightarrow$ running time: $O(|E| + |V|)$

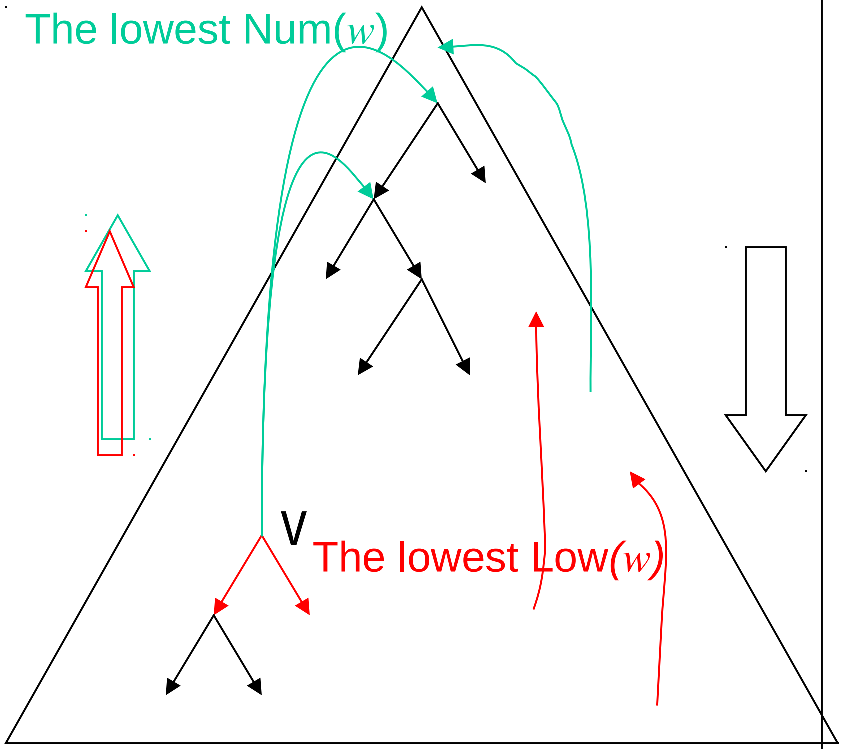
- Finding articulation points

  a.) the root is an articulation point if it has more than one child.

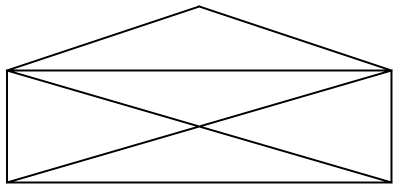
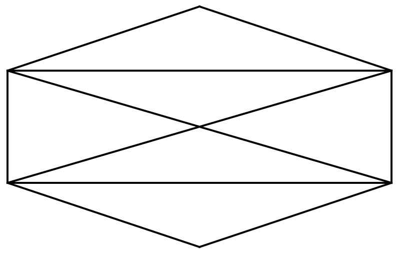
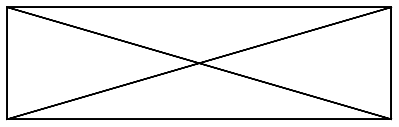
  b.) Any other vertex, *v*, is an articulation point if and only if *v* has some child *w* such that Low(*w*) ≥ Num(*v*).

- The algorithm: `(Fig. 9.65 - 9.67)`

```
Void Graph::assignLow(Vertex v)
{ Vertex w;
  v.visited = true
  v.low = v.num = counter++;// Rule 1
  For each w adjacent to v
  {
   if(w.num>v.num)//Forward edge
   {
     assignLow(w);
     if(w.low>=v.num)
      cout<< v<< is an art-point;
     v.low = min(v.low, w.low);
   }
   else
   if(v.parent !=w) // Back edge
    v.low = min(v.low, w.num);
  }
}
```



- Euler circuits
  - The problem: finding a path in an undirected graph that visit each edge exactly once. This path is called *Euler path*. If the path is a cycle, it is called an *Euler circuit*.

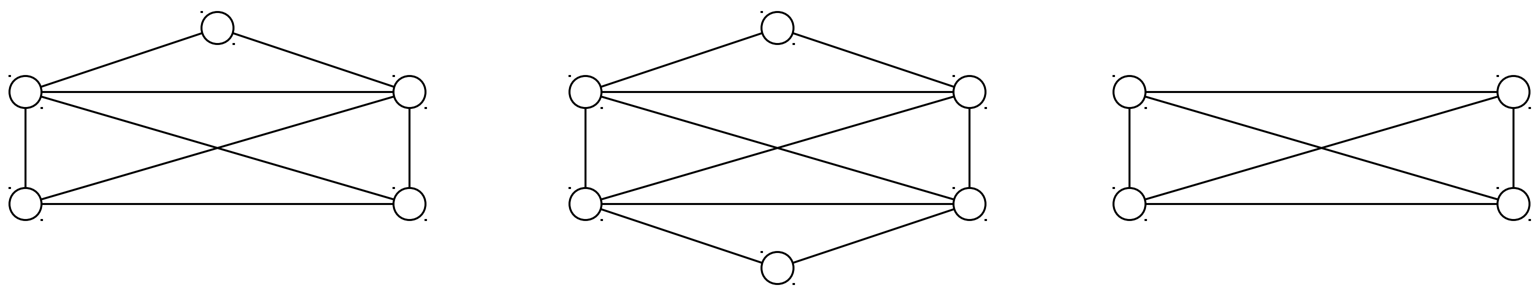




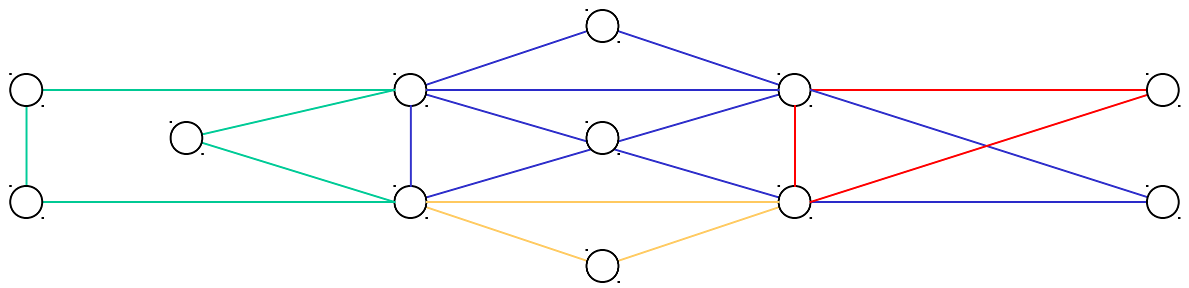
  - An Euler circuit exists in an undirected graph if and only if the graph is connected, and all vertices have even degrees (the number of edges connected directly to the vertex).
  - Finding the Euler circuit

    a.) starting from an arbitrary vertex, and find a path by depth-first search until the path comes back to the *1st* vertex

    b.) If there is still unvisited edges, repeat a.) at a vertex on the existing path, traversing only unvisited edges. This will generate another path, which can be inserted into the exist path at the vertex from which the new path started

    c.) repeat b.) until all edges are visited.





- Linked lists should be used for all paths to enable easy insertion
- Each adjacent list (in the graph representation) should keep a pointer to the next unvisited edge in the list.
- running time: $O(|E| + |V|)$
- example: `(Fig. 9.70 - 9.73)`
- Hamilton cycle problem: finding a path that visits each vertex exactly once — A much harder problem.

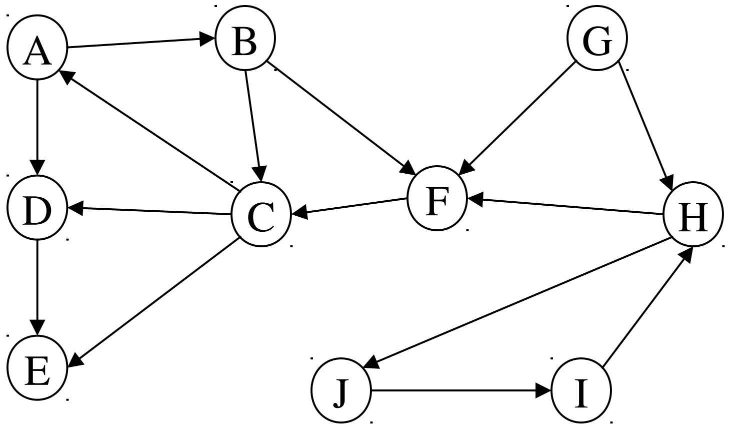
- Depth-first search in directed graph
  - If the graph is not strongly connected, depth-first search can generate multiple search trees (forest)
  - example: `(Fig. 9.74 - 9.75)`
  - Three types of edges that are not included in the search trees because they do not lead to new vertices.

    a.) back edges

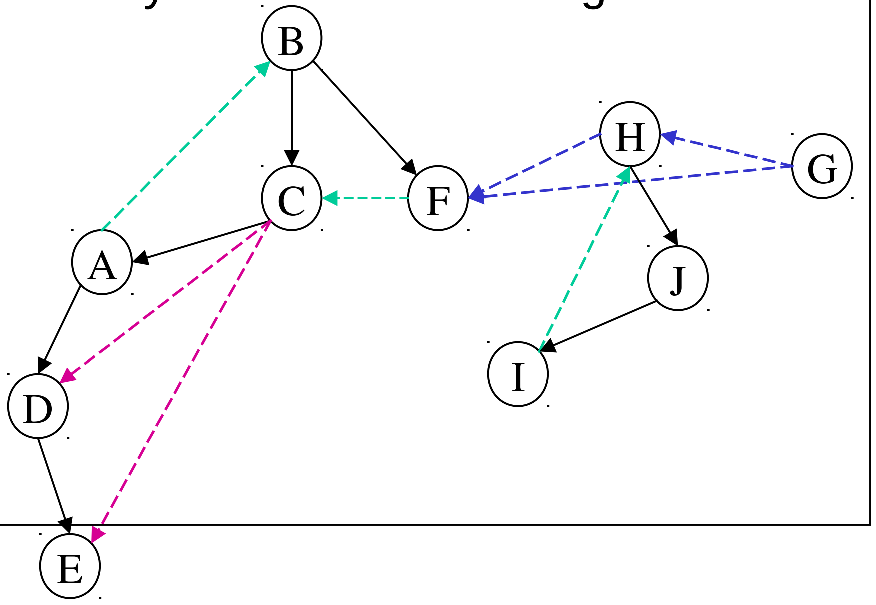
    b.) forward edges: leading from a tree node to a descendant

    c.) cross edges: connect two tree nodes that are not directly related.

    Cross edges always go from right to left.
  - A directed graph is acyclic if and only if it has no back edges.



$$\begin{tikzcd}
A \arrow[r] \arrow[d] & B \arrow[d] \arrow[rr] & & F \arrow[ll] & G \arrow[l] \arrow[d] \\
D \arrow[d] & C \arrow[ul] \arrow[l] \arrow[dl] & & & H \arrow[ll] \arrow[dll] \\
E & & J \arrow[r] & I \arrow[ur] &
\end{tikzcd}$$



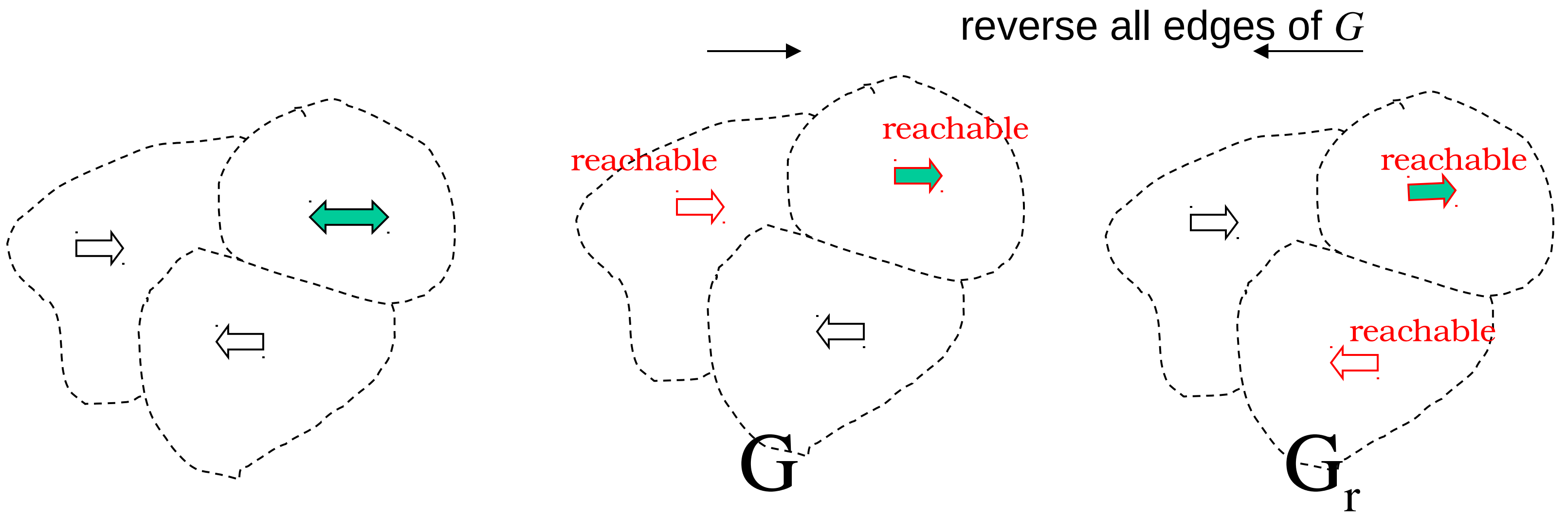
$$\begin{tikzcd}
& & B \arrow[d] \arrow[r] & F \arrow[l, dashed] & H \arrow[l, dashed] \arrow[d] & G \arrow[l, dashed] \arrow[ll, dashed] \\
& A \arrow[uur, dashed] \arrow[d] & C \arrow[l] \arrow[dl, dashed] \arrow[ddl, dashed] & & J \arrow[dl] & \\
& D \arrow[d] & & I \arrow[uur, dashed] & & \\
& E & & & &
\end{tikzcd}$$

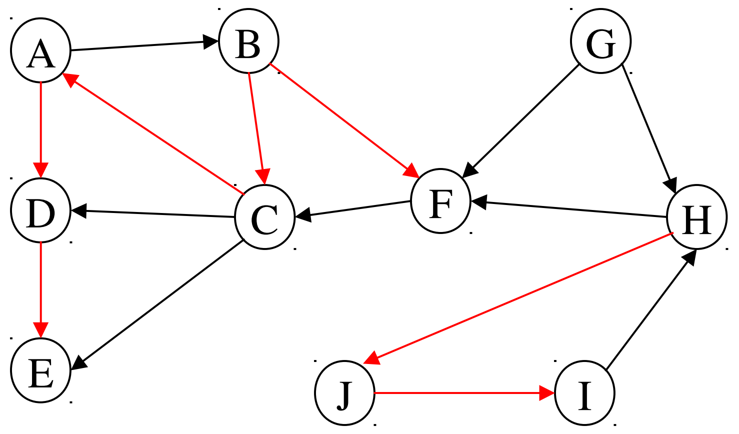
- Strong components in a directed graph
  - Strong component: the subset of the vertices in a directed graph that are strongly connected to itself.
  - Basic idea:

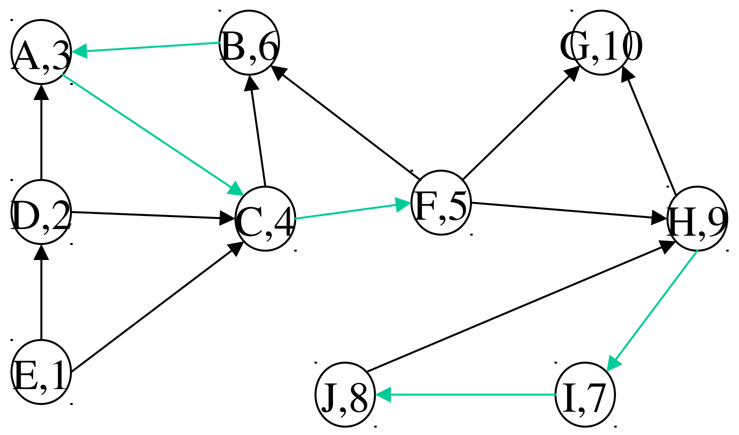
    a.) perform a depth-first search on G, numbering the vertices in the post-order of the depth-first spanning tree (leaves have smaller numbers than roots)

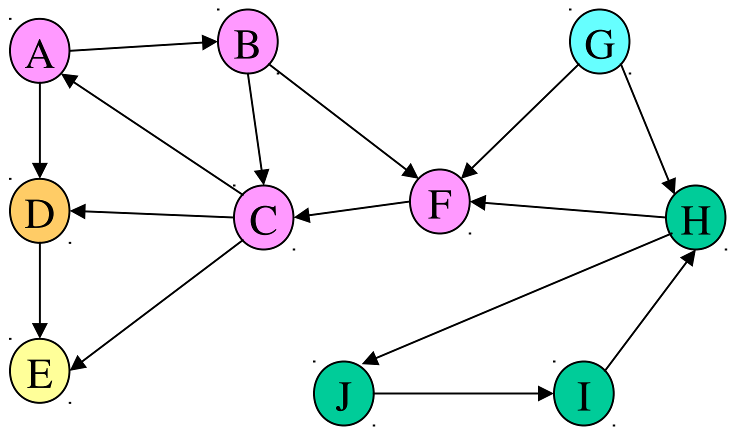
    b.) reverse all edges of G, generate G_r

    c.) perform a depth-first search on G_r, always starting from the highest numbered vertex. The depth-first spanning trees so generated are the strong components of G. (always start from roots)




A
B
G
D
C
F
H
E
J
I


A,3
B,6
G,10
D,2
C,4
F,5
H,9
E,1
J,8
I,7


A
B
G
D
C
F
H
E
J
I

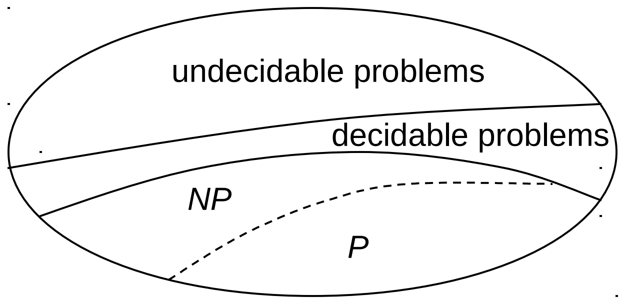
9.7 Introduction to NP-Completeness

- Undecidable problems
  - Certain problems are not solvable by computers — undecidable problems
  - An example: given a program, check whether it contains any infinite loop.
- The Halting problem
  - If a loop checking program exists, produce a program called Loop, which takes as input a program *P* that takes a strong as input, and:

$$Loop(P) = \begin{cases} \textit{Yes: if } P(P) \textit{ loops} \\ \textit{Loop forever: if } P(P) \textit{ termination} \end{cases}$$

  - *Loop(Loop)*?
- Non-deterministic algorithms
- An algorithm is non-deterministic if, at any point of the algorithm, it can choose, non-deterministically, from multiple next steps.
- class *NP*: the class of algorithms with polynomial un-deterministic running time.

- class P: the class of algorithms with polynomial running time.
- $P \subseteq NP$
- The Hamilton cycle problem is in NP
- Not all decidable problems are in NP
- $P \Rightarrow N$



- NP-complete problems
- NP-complete problems: a subset of NP.
- Theorem: if any P-complete problem has a deterministic polynomial solution, then all problems in NP are also in P,

  i.e. $P = NP$
- Hamilton cycle problem is NP-complete
- Traveling Salesman problem:

  Given a complete graph $G = (V, E)$, with edge costs, find the shortest simple cycle that visits all vertices.