Lesson 12 Graph Algorithms

Combinatorics of Pure Intelligence

Wholeness of the Lesson

Graphs are data structures that do more than simply store and organize data; they are used to model interactions in the world. This makes it possible to make use of the extensive mathematical knowledge from the theory of graphs to solve problems abstractly, at the level of the model, resulting in a solution to real-world problems.

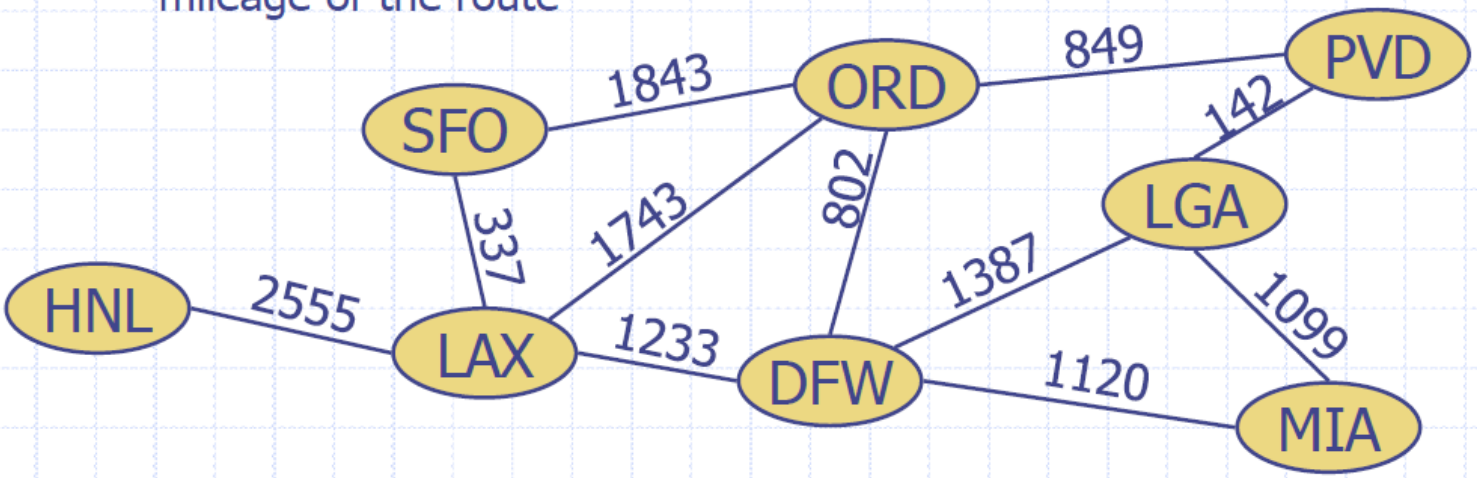
Science of Consciousness: Our own deeper levels of intelligence exhibit more of the characteristics of Nature's intelligence than our own surface level of thinking. Bringing awareness to these deeper levels, as the mind dives inward, engages Nature's intelligence, Nature's know-how, and this value is brought into daily activity. The benefit is greater ability to solve real-world problems, meet challenges, and find the right path for success.

Overview: Graph Algorithms

- Elementary Graph Algorithms
  - Introductory Concepts
  - Graph Traversals:
    - Depth-First Search
    - Breadth-First Search
  - Topological Sort
- Minimum Spanning Trees
  - Kruskal
  - Prim
- Shortest Paths
  - Dijkstra
  - Bellman-Ford

Introductory Graph Concepts -1

- A graph is a pair (V, E), where
  - V is a set of nodes, called vertices
  - E is a collection of pairs of vertices, called edges
  - Vertices and edges can be implemented so that they store elements
- Example:
  - A vertex represents an airport and stores the three-letter airport code
  - An edge represents a flight route between two airports and stores the mileage of the route



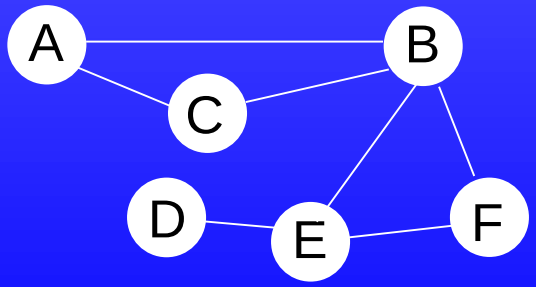
This treatment follows 91.503 textbook Cormen et al. Some definitions differ slightly from other graph literature.

Introductory Graph Concepts -2

- G= (V,E)
- Vertex Degree
- Self-Loops

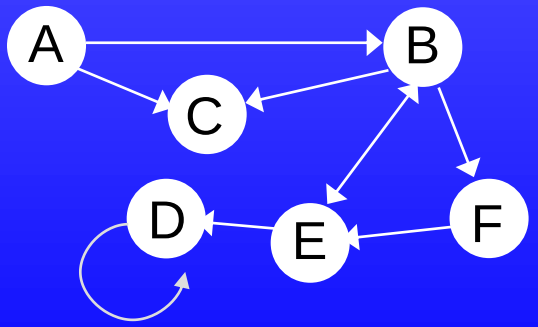
Undirected Graph

- No Self-Loops
- Adjacency is symmetric



Directed Graph (digraph)

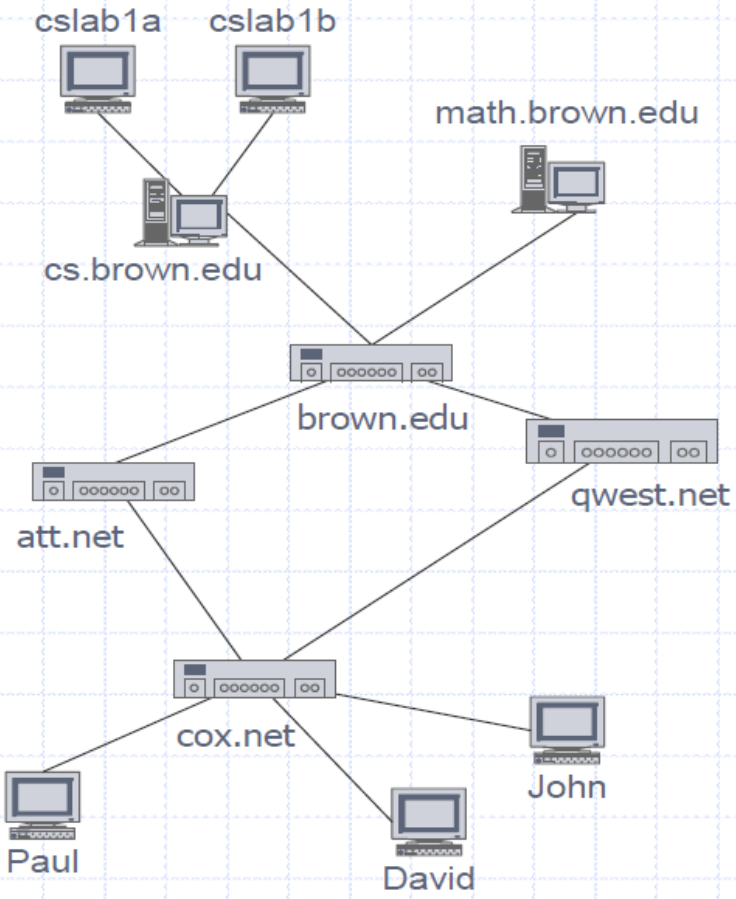
- Degree: in/out
- Self-Loops allowed



This treatment follows 91.503 textbook Cormen et al. Some definitions differ slightly from other graph literature.

Sample Applications

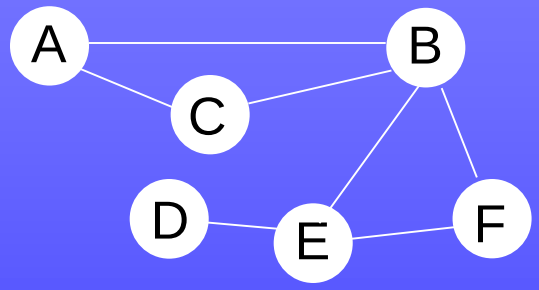
- Electronic circuits
  - Printed circuit board
    (nodes = junctions, edges are the traces)
- Transportation networks
  - Highway network
  - Flight network
- Computer networks
  - Local area network
  - Internet
  - Web
- Databases
  - Entity-relationship diagram
- Physics / Chemistry
  - Atomic structure simulations (e.g. shortest path algs)
  - Model of molecule -- atoms/bonds



This treatment follows 91.503 textbook Cormen et al. Some definitions differ slightly from other graph literature.

Introductory Graph Concepts: Representations

Undirected Graph



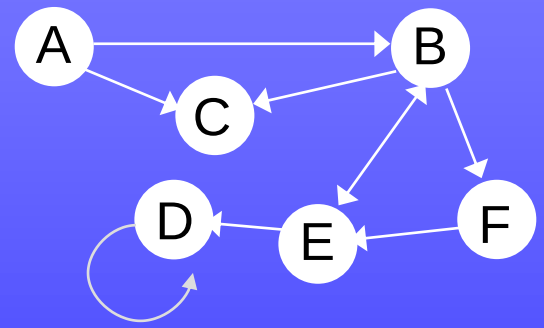
	A	B	C	D	E	F
A	0	1	1	0	0	0
B	1	0	1	0	1	1
C	1	1	0	0	0	0
D	0	0	0	0	1	0
E	0	1	0	1	0	1
F	0	1	0	0	1	0

Adjacency Matrix

A	BC
B	ACEF
C	AB
D	E
E	BDF
F	BE

Adjacency List

Directed Graph (digraph)



	A	B	C	D	E	F
A	0	1	1	0	0	0
B	0	0	1	0	1	1
C	0	0	0	0	0	0
D	0	0	0	1	0	0
E	0	1	0	1	0	0
F	0	0	0	0	1	0

Adjacency Matrix

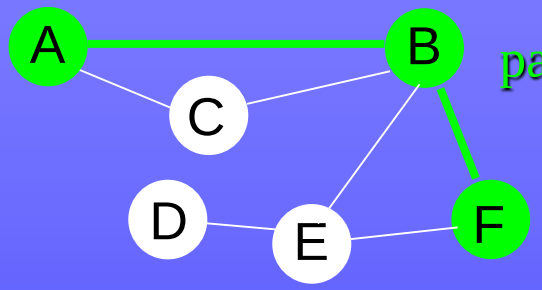
A	BC
B	CEF
C	
D	D
E	BD
F	E

Adjacency List

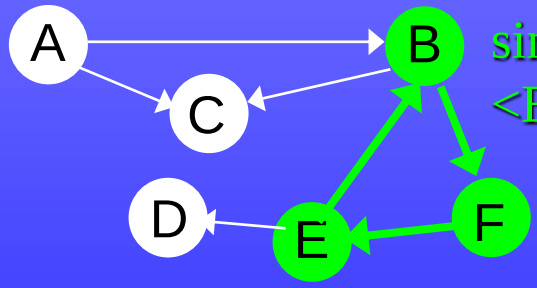
This treatment follows 91.503 textbook Cormen et al. Some definitions differ slightly from other graph literature.

Introductory Graph Concepts: Paths, Cycles

- Path:
  - length: number of edges
  - simple: all vertices distinct
- Cycle:
  - Directed Graph:
    - $<v_0,v_1,...,v_k>$ forms cycle if $v_0=v_k$ and $k>=1$
    - simple cycle: $v_1,v_2..,v_k$ also distinct
    - self-loop is cycle of length 1
  - Undirected Graph:
    - $<v_0,v_1,...,v_k>$ forms (simple) cycle if $v_0=v_k$ and $k>=3$
    - simple cycle: $v_1,v_2..,v_k$ also distinct

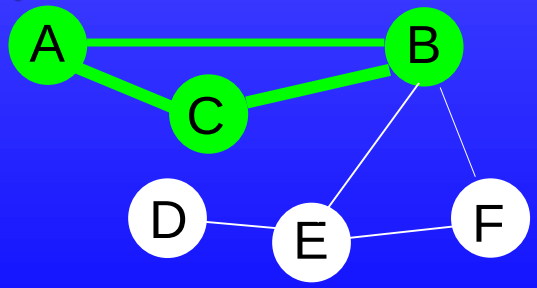


path <A,B,F>



simple cycle <E,B,F,E>

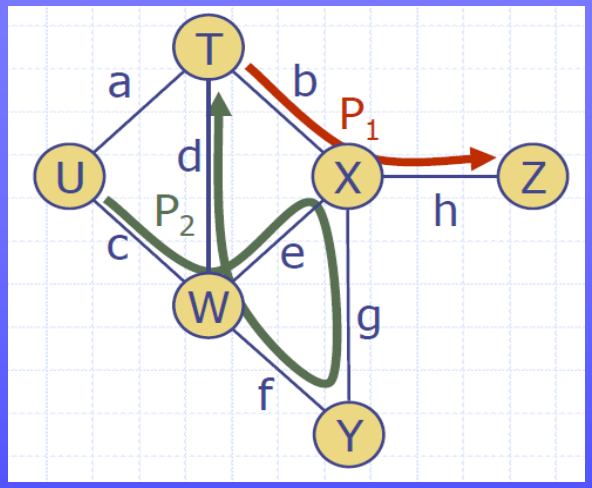
simple cycle <A,B,C,A>= <B,C,A,B>



This treatment follows 91.503 textbook Cormen et al. Some definitions differ slightly from other graph literature.

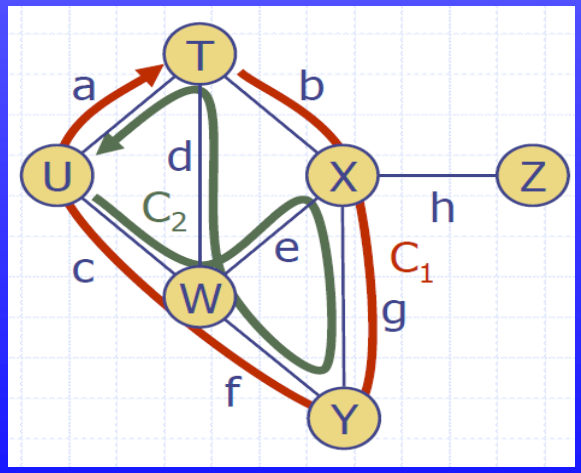
Introductory Graph Concepts: Paths, Cycles

- Non Simple Path:
  - Going thru a node more than once



Non simple path
<U,W,X,Y,W,T>

- Non Simple Cycle:
  - Going thru a node more than once

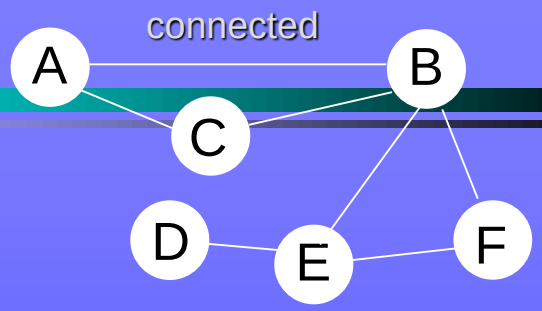


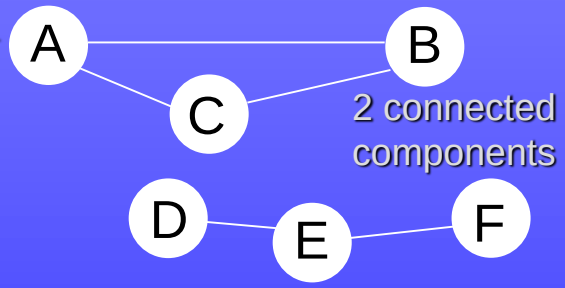
Non simple cycle
<U,W,X,Y,W,T,U>

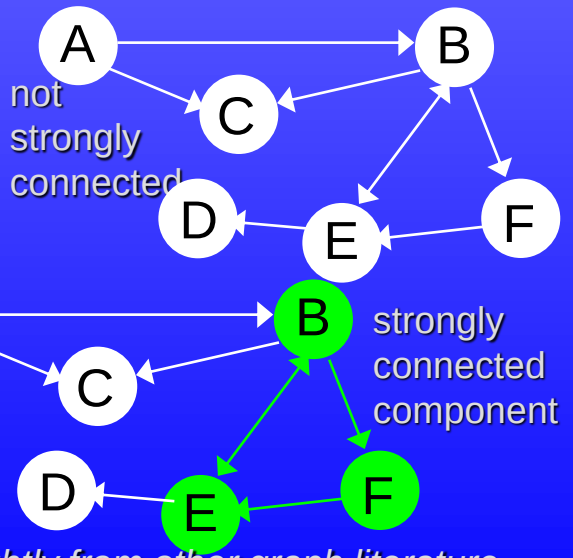
From Corazza.

Introductory Graph Concepts: Connectivity

- Undirected Graph: *connected*
  - every pair of vertices is connected by a path
  - one *connected component*
  - connected components:
    - equivalence classes under "is reachable from" relation
- Directed Graph: *strongly connected*
  - every pair of vertices is reachable from each other
  - one *strongly connected component*
  - strongly connected components:
    - equivalence classes under "mutually reachable" relation



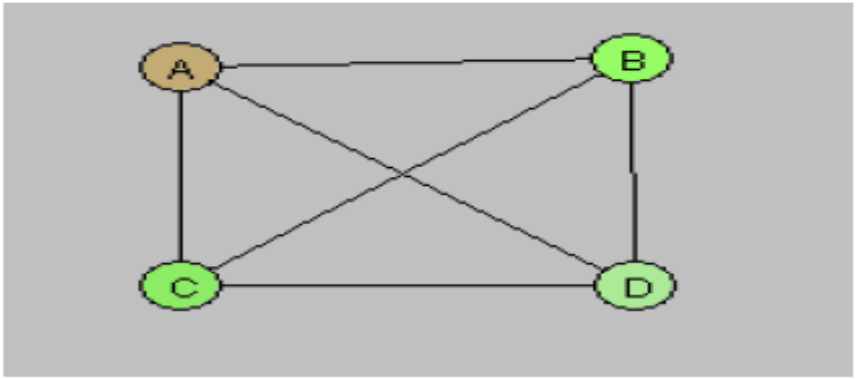




This treatment follows 91.503 textbook Cormen et al. Some definitions differ slightly from other graph literature.

Introductory Graph Concepts: Complete Graph

- A graph G is **complete** if for every pair of vertices u, v, there is an edge $e \in E$ that is incident to u, v. In other words, for every u, v, $(u, v) \in E$.
- This is the complete graph on 4 vertices, denoted K_4.



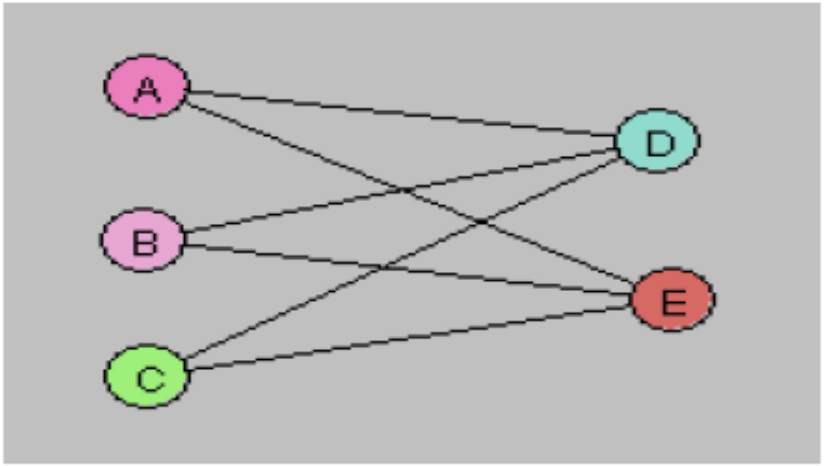
- In general, the complete graph on n vertices is denoted K_n. The one-point graph is K_1.
- **Exercise.** For a complete graph G,

$$\epsilon = \frac{\nu(\nu - 1)}{2}.$$

Therefore, K_n has exactly $n(n-1)/2$ edges.

Introductory Graph Concepts: Bipartite Graph

- A graph G is **bipartite** if there exist sets X, Y that are disjoint and subsets of V with the property that every $e \in E$ is incident with a vertex in X and a vertex in Y. For such graphs, (X, Y) is called a **bipartition** of G. A **complete bipartite graph** G with bipartition (X, Y) is a bipartite graph with the property that for every $u \in X$ and every $v \in Y$, $(u, v) \in E$.
- This an example of a complete bipartite graph.

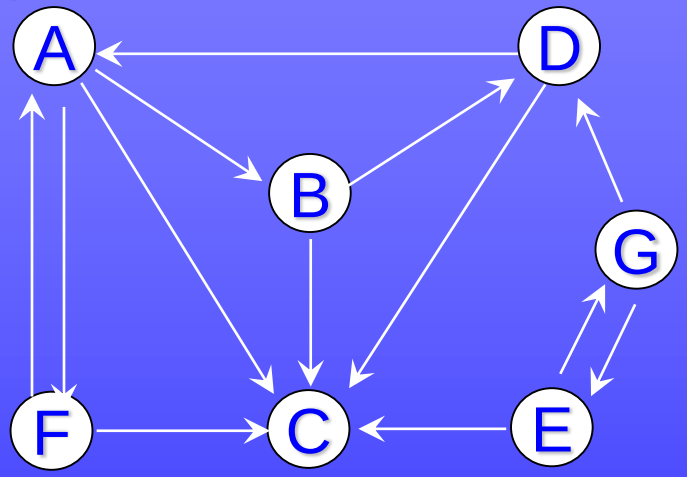


- A simple cycle is *odd* (*even*) if the length of the cycle is odd (*even*).

Depth-First Search (DFS)

Example: DFS of Directed Graph

G=(V,E)



Adjacency List:

A: B,C,F

B: C,D

C: -

D: A,C

E: C,G

F: A,C

G: D,E

Edge Classification Legend:

T: tree edge

B: back edge

F: forward edge

C: cross edge

Source: Graph is from *Computer Algorithms: Introduction to Design and Analysis* by Baase and Gelder.

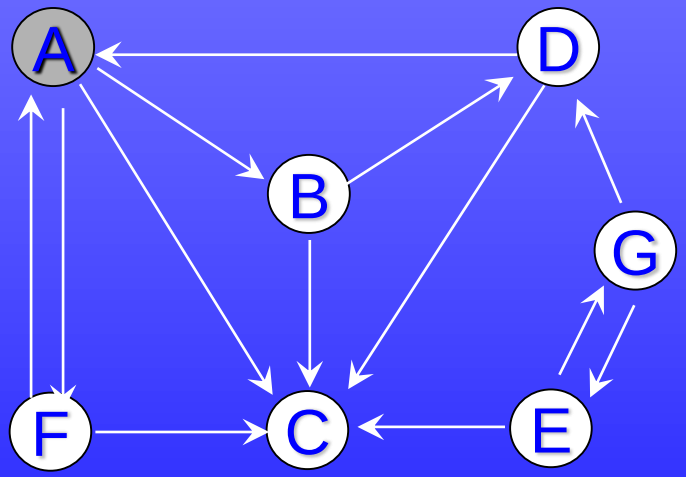
Vertex Color Changes

- Vertex is WHITE if it has not yet been encountered during the search.
- Vertex is GRAY if it has been encountered but has not yet been fully explored.
- Vertex is BLACK if it has been fully explored.

Edge Classification

- Each edge of the original graph G is classified during the search
  - produces information needed to:
    - build DFS or BFS spanning forest of trees
    - detect cycles (DFS) or find shortest paths (BFS)
- When vertex u is being explored, edge e = (u,v) is classified based on the color of v when the edge is *first explored*:
  - e is a *tree edge* if v is WHITE [for DFS and BFS]
  - e is a *back edge* if v is GRAY [for DFS only]
    - for DFS this means v is an ancestor of u in the DFS tree
  - e is a *forward edge* if v is BLACK and [for DFS only] v is a descendent of u in the DFS tree
  - e is a *cross edge* if v is BLACK and [for DFS only] there is no ancestor or descendent relationship between u and v in the DFS tree
- Note that:
  - For BFS we'll only consider tree edges.
  - For DFS we consider all 4 edge types.
  - In DFS of an undirected graph, every edge is either a tree edge or a back edge.

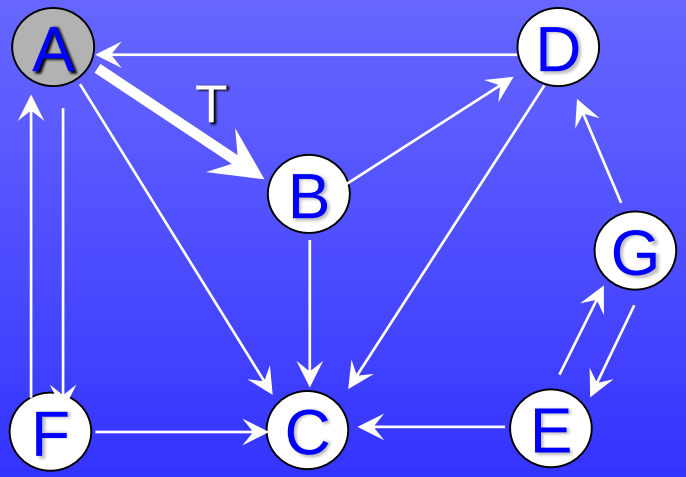
Example: (continued) DFS of Directed Graph



Adjacency List:

A: B,C,F
B: C,D
C: -
D: A,C
E: C,G
F: A,C
G: D,E

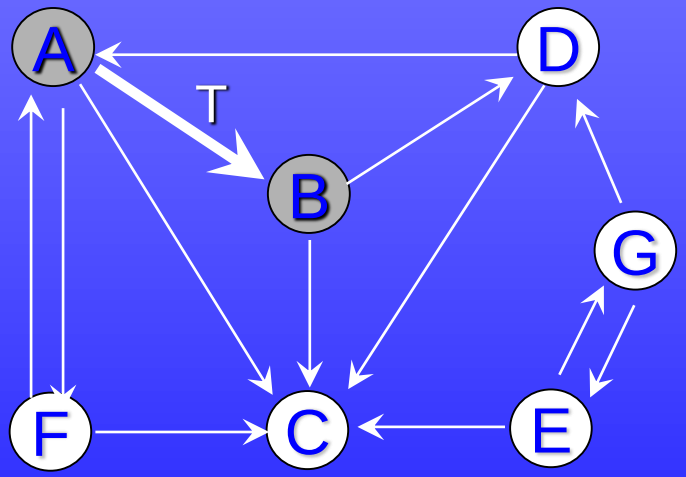
Example: (continued) DFS of Directed Graph



Adjacency List:

A: B,C,F
B: C,D
C: -
D: A,C
E: C,G
F: A,C
G: D,E

Example: (continued) DFS of Directed Graph



Adjacency List:

A: B,C,F

B: C,D

C: -

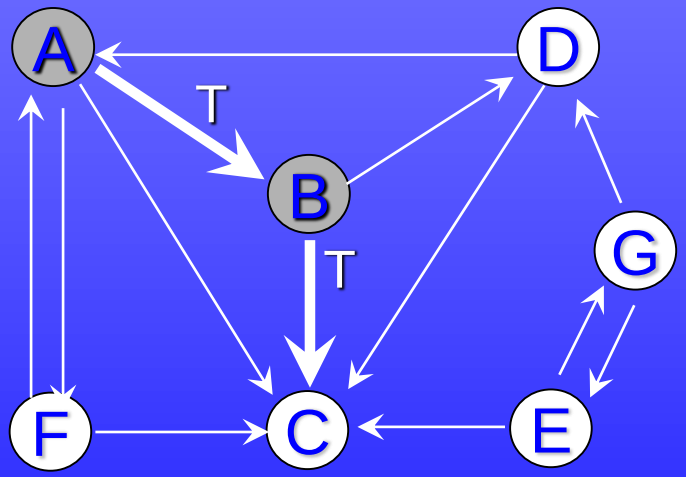
D: A,C

E: C,G

F: A,C

G: D,E

Example: (continued) DFS of Directed Graph



Adjacency List:

A: B,C,F

B: C,D

C: -

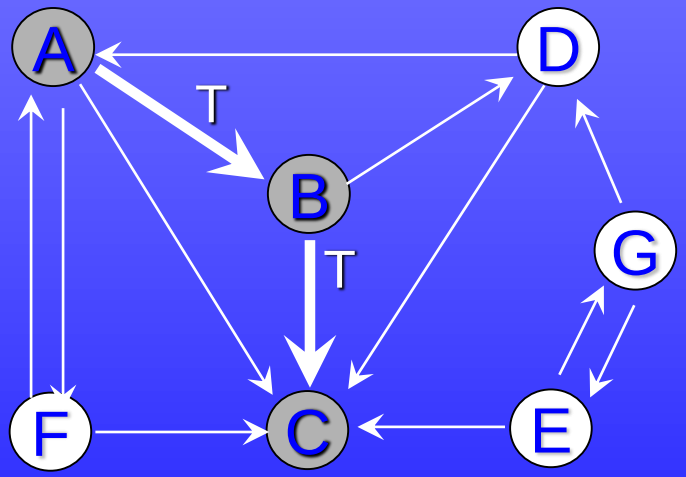
D: A,C

E: C,G

F: A,C

G: D,E

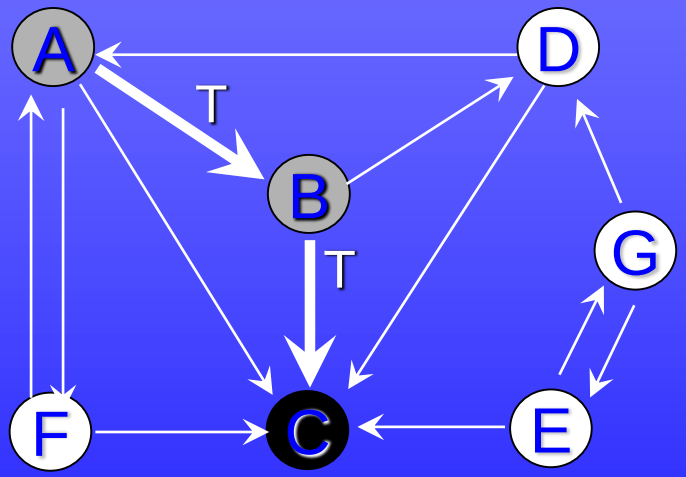
Example: (continued) DFS of Directed Graph



Adjacency List:

A: B,C,F
B: C,D
C: -
D: A,C
E: C,G
F: A,C
G: D,E

Example: (continued) DFS of Directed Graph



Adjacency List:

A: B,C,F

B: C,D

C: -

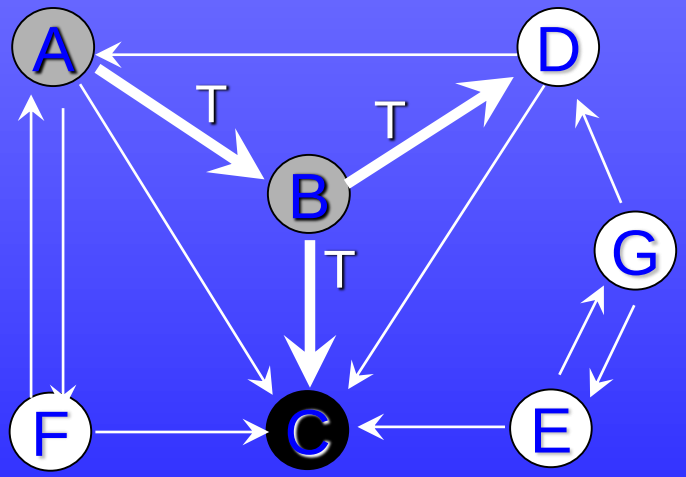
D: A,C

E: C,G

F: A,C

G: D,E

Example: (continued) DFS of Directed Graph



Adjacency List:

A: B,C,F

B: C,D

C: -

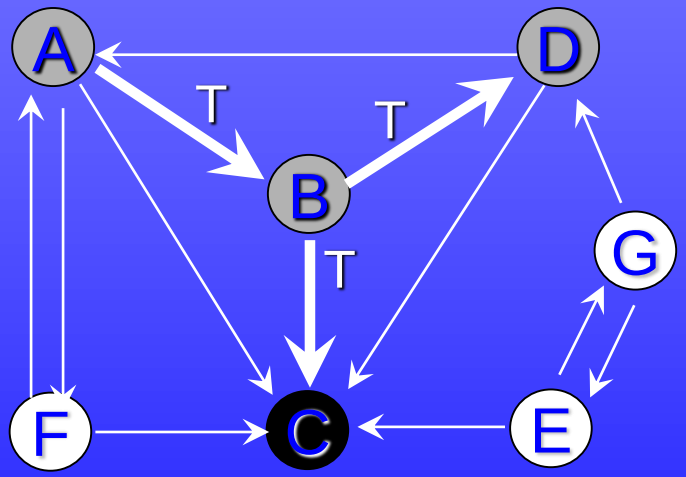
D: A,C

E: C,G

F: A,C

G: D,E

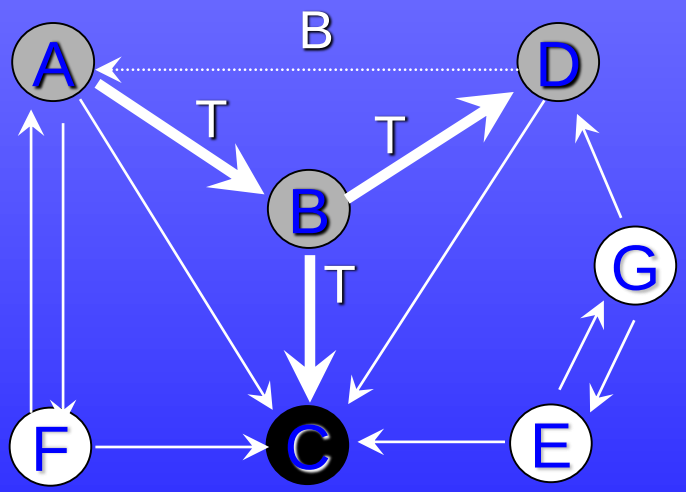
Example: (continued) DFS of Directed Graph



Adjacency List:

A: B,C,F
B: C,D
C: -
D: A,C
E: C,G
F: A,C
G: D,E

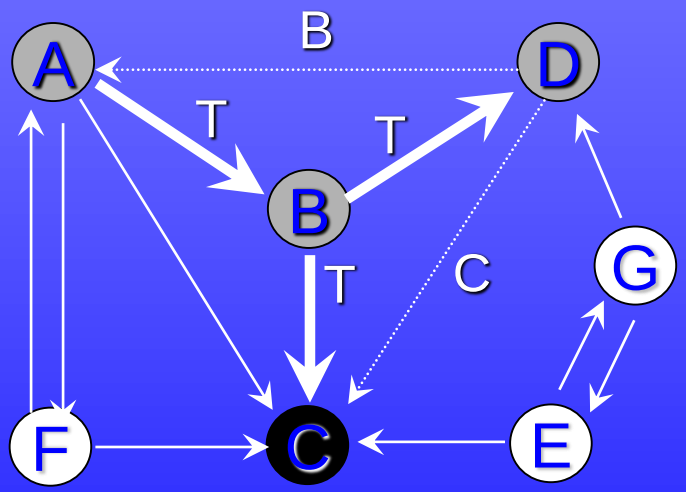
Example: (continued) DFS of Directed Graph



Adjacency List:

A: B,C,F
B: C,D
C: -
D: A,C
E: C,G
F: A,C
G: D,E

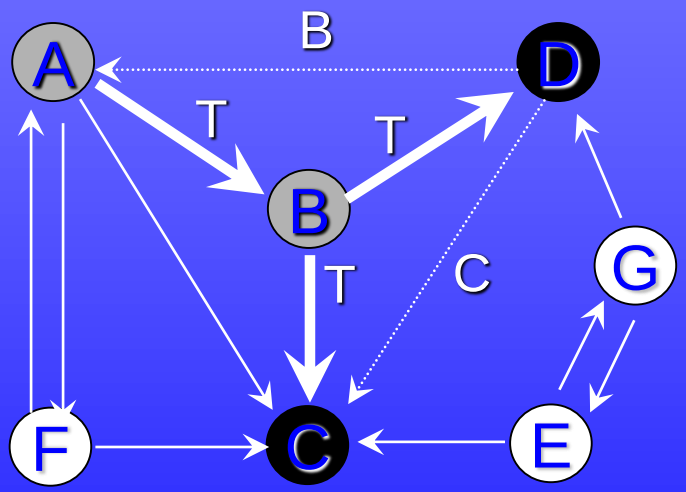
Example: (continued) DFS of Directed Graph



Adjacency List:

A: B,C,F
B: C,D
C: -
D: A,C
E: C,G
F: A,C
G: D,E

Example: (continued) DFS of Directed Graph



Adjacency List:

A: B,C,F

B: C,D

C: -

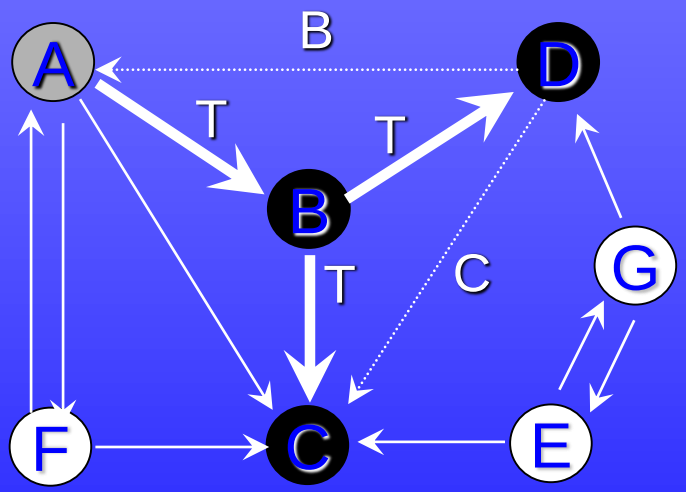
D: A,C

E: C,G

F: A,C

G: D,E

Example: (continued) DFS of Directed Graph



Adjacency List:

A: B,C,F

B: C,D

C: -

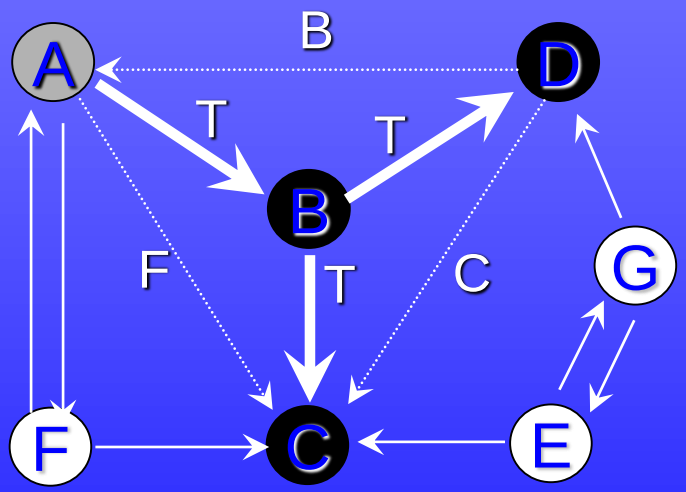
D: A,C

E: C,G

F: A,C

G: D,E

Example: (continued) DFS of Directed Graph



Adjacency List:

A: B,C,F

✓ B: C,D

✓ C: -

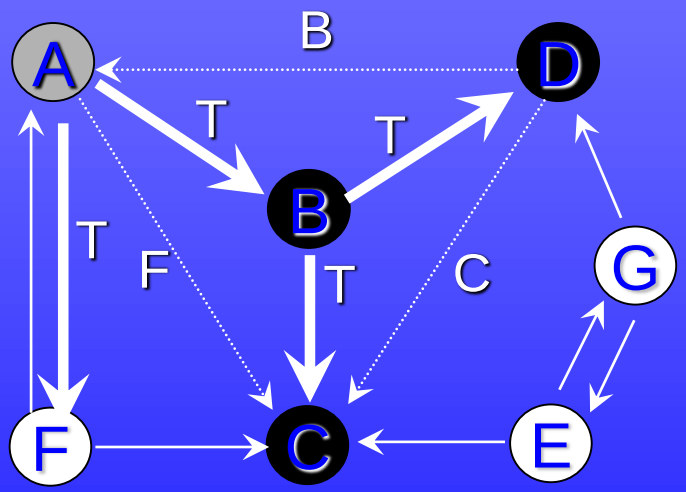
✓ D: A,C

E: C,G

F: A,C

G: D,E

Example: (continued) DFS of Directed Graph



Adjacency List:

A: B,C,F

✓ B: C,D

✓ C: -

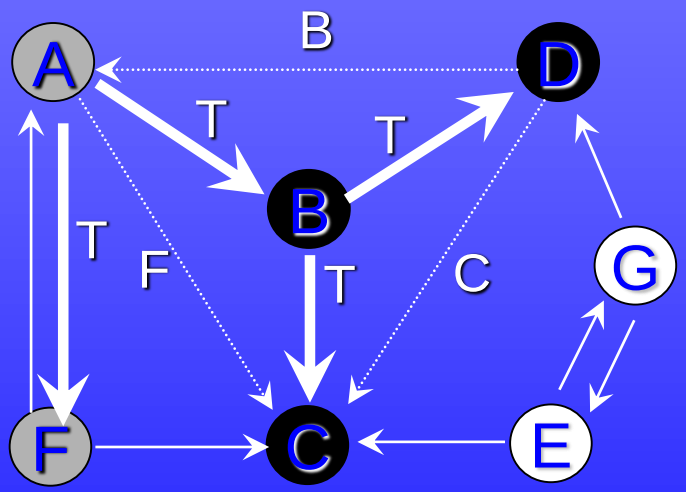
✓ D: A,C

E: C,G

F: A,C

G: D,E

Example: (continued) DFS of Directed Graph



Adjacency List:

A: B,C,F

✓ B: C,D

✓ C: -

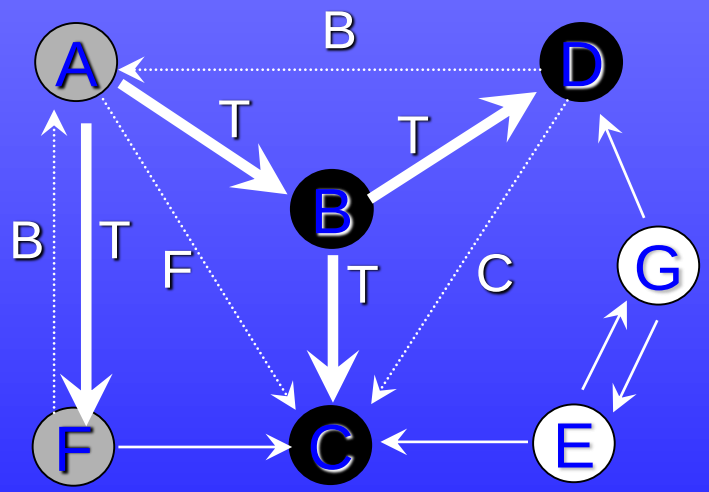
✓ D: A,C

E: C,G

F: A,C

G: D,E

Example: (continued) DFS of Directed Graph



Adjacency List:

A: B,C,F

✓ B: C,D

✓ C: -

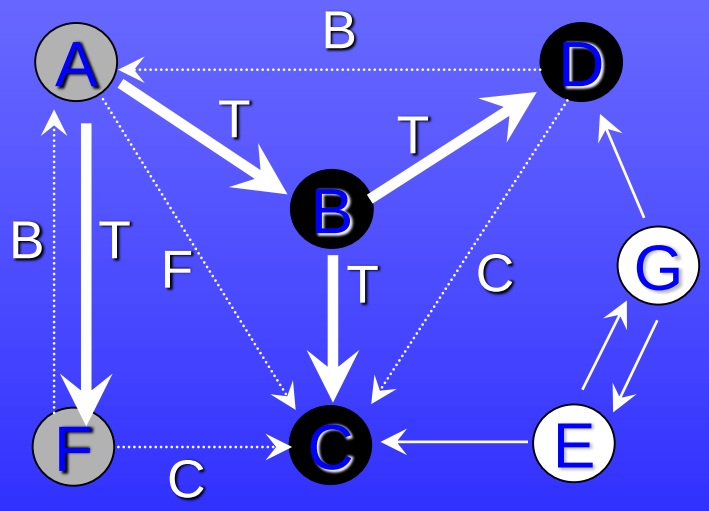
✓ D: A,C

E: C,G

F: A,C

G: D,E

Example: (continued) DFS of Directed Graph



Adjacency List:

A: B,C,F

✓ B: C,D

✓ C: -

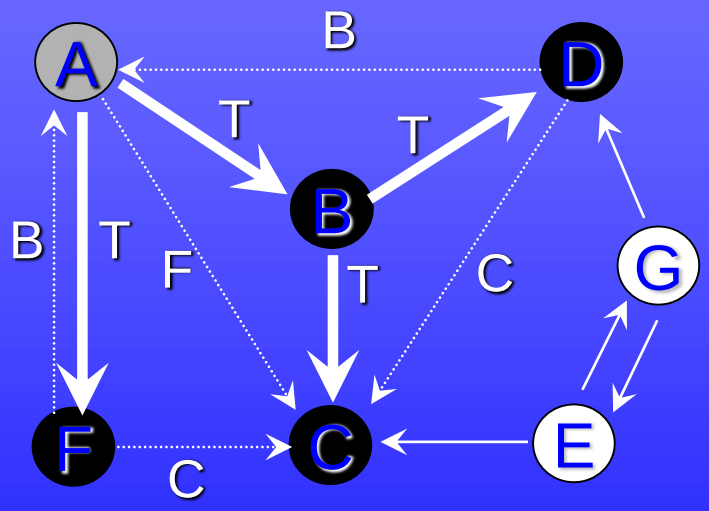
✓ D: A,C

E: C,G

F: A,C

G: D,E

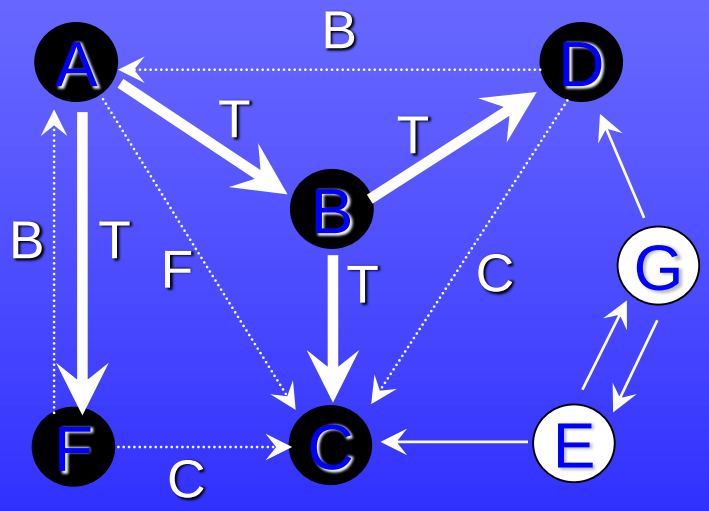
Example: (continued) DFS of Directed Graph



Adjacency List:

- A: B,C,F
- ✓ B: C,D
- ✓ C: -
- ✓ D: A,C
- E: C,G
- ✓ F: A,C
- G: D,E

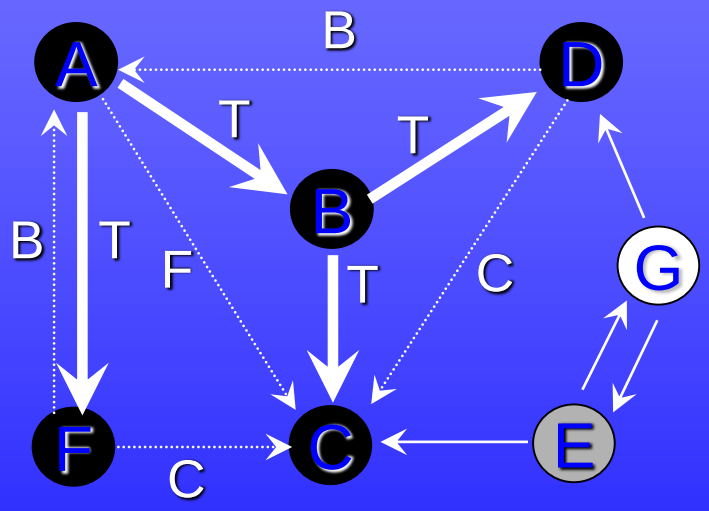
Example: (continued) DFS of Directed Graph



Adjacency List:

- A: B,C,F
- B: C,D
- C: -
- D: A,C
- E: C,G
- F: A,C
- G: D,E

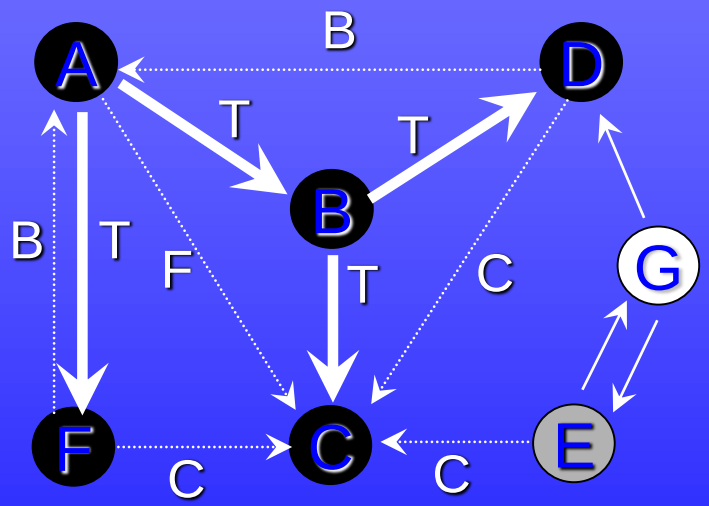
Example: (continued) DFS of Directed Graph



Adjacency List:

- A: B,C,F
- B: C,D
- C: -
- D: A,C
- E: C,G
- F: A,C
- G: D,E

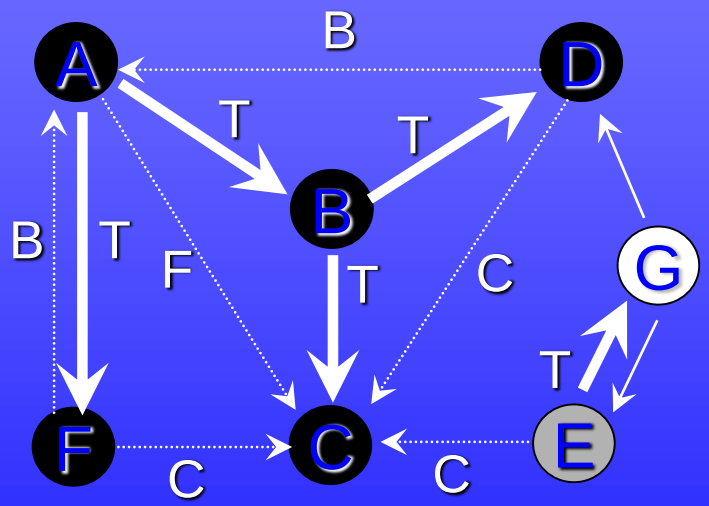
Example: (continued) DFS of Directed Graph



Adjacency List:

- ✓ A: B,C,F
- ✓ B: C,D
- ✓ C: -
- ✓ D: A,C
- E: C,G
- ✓ F: A,C
- G: D,E

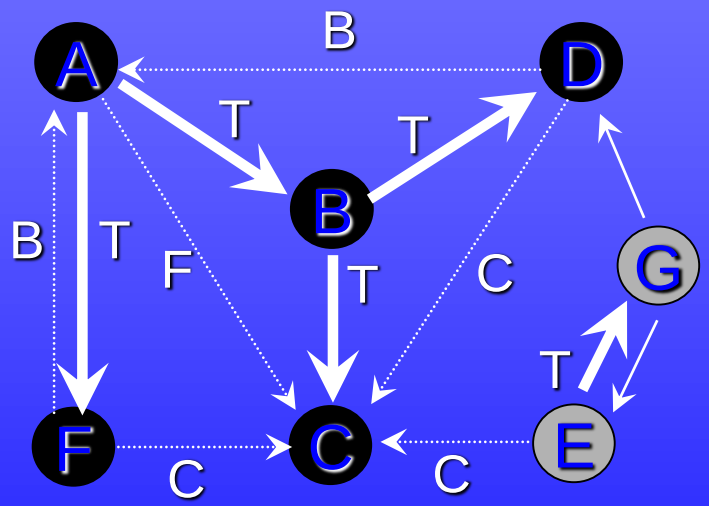
Example: (continued) DFS of Directed Graph



Adjacency List:

- A: B,C,F
- B: C,D
- C: -
- D: A,C
- E: C,G
- F: A,C
- G: D,E

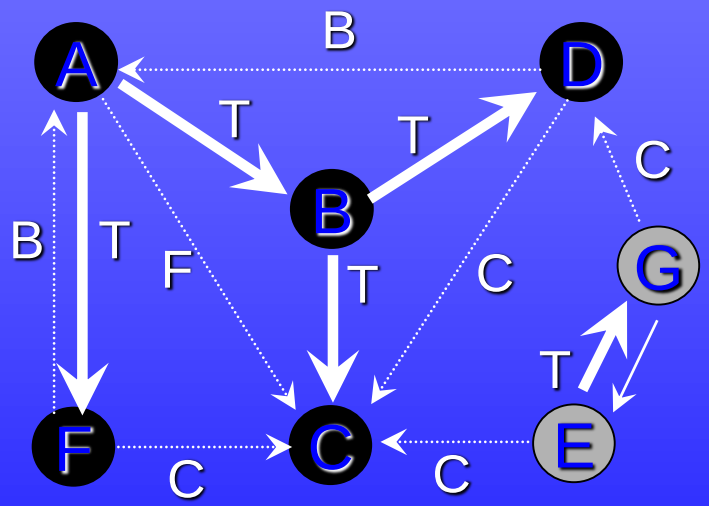
Example: (continued) DFS of Directed Graph



Adjacency List:

- ✓ A: B,C,F
- ✓ B: C,D
- ✓ C: -
- ✓ D: A,C
- E: C,G
- ✓ F: A,C
- G: D,E

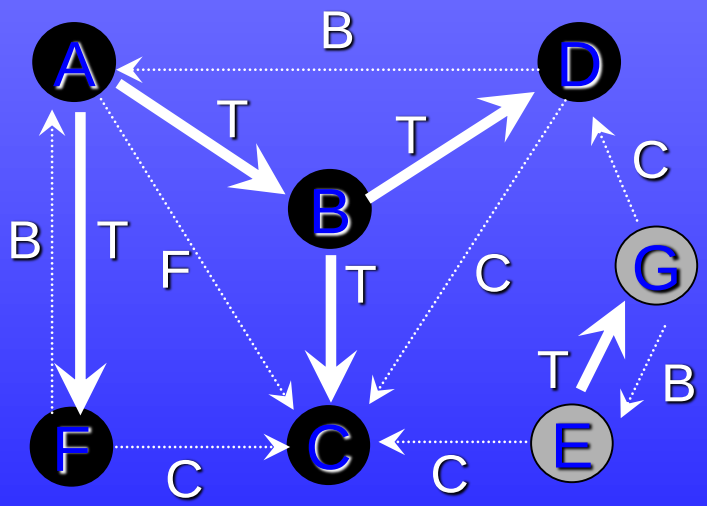
Example: (continued) DFS of Directed Graph



Adjacency List:

- A: B, C, F
- B: C, D
- C: -
- D: A, C
- E: C, G
- F: A, C
- G: D, E

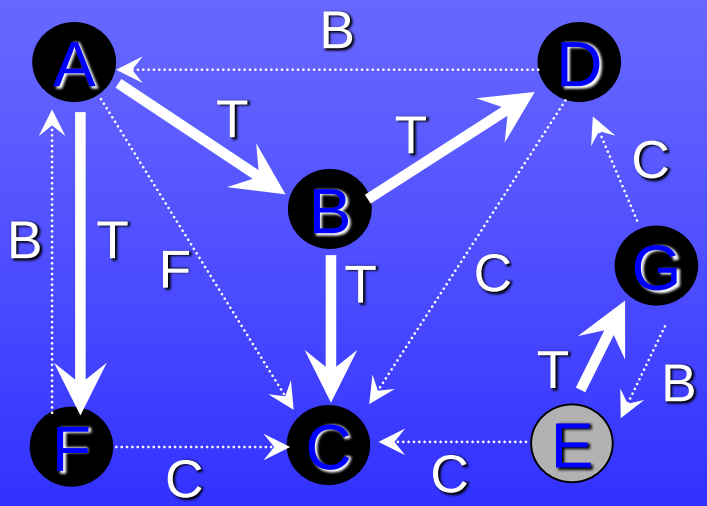
Example: (continued) DFS of Directed Graph



Adjacency List:

- ✓ A: B,C,F
- ✓ B: C,D
- ✓ C: -
- ✓ D: A,C
- E: C,G
- ✓ F: A,C
- G: D,E

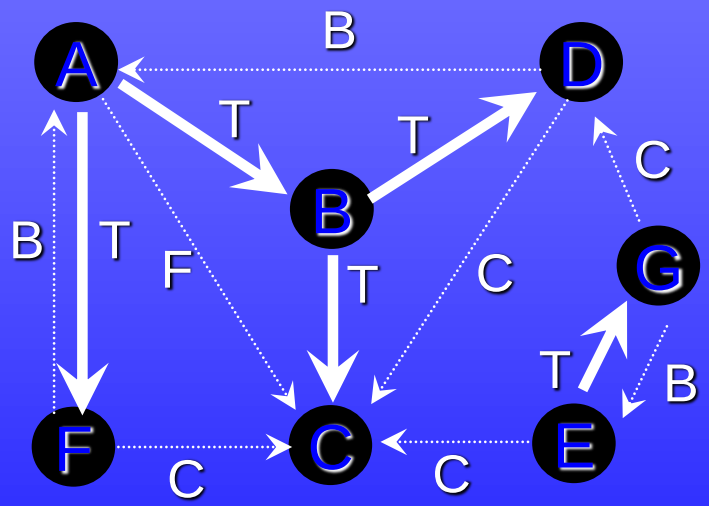
Example: (continued) DFS of Directed Graph



Adjacency List:

- A: B, C, F
- B: C, D
- C: -
- D: A, C
- E: C, G
- F: A, C
- G: D, E

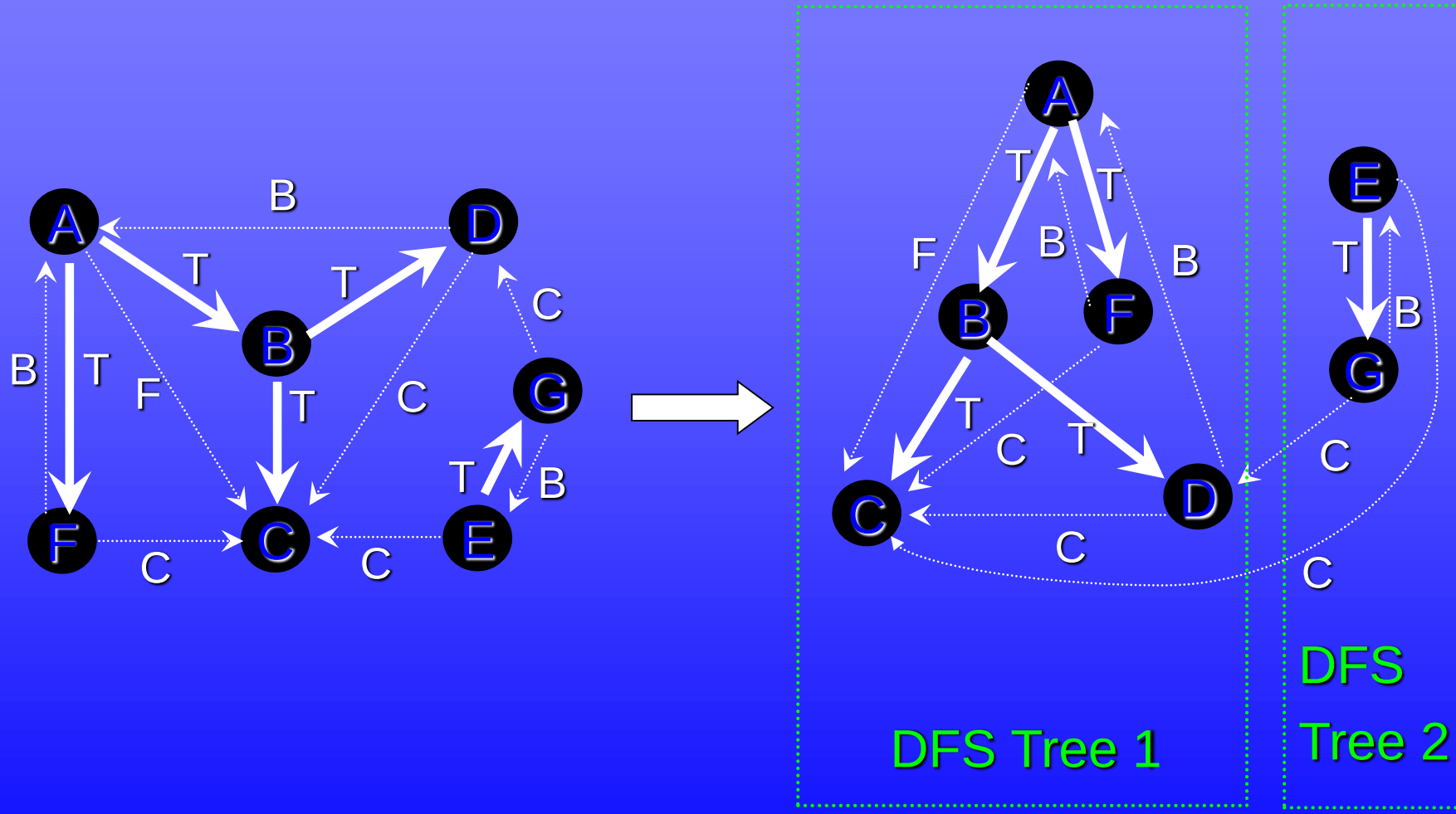
Example: (continued) DFS of Directed Graph



Adjacency List:

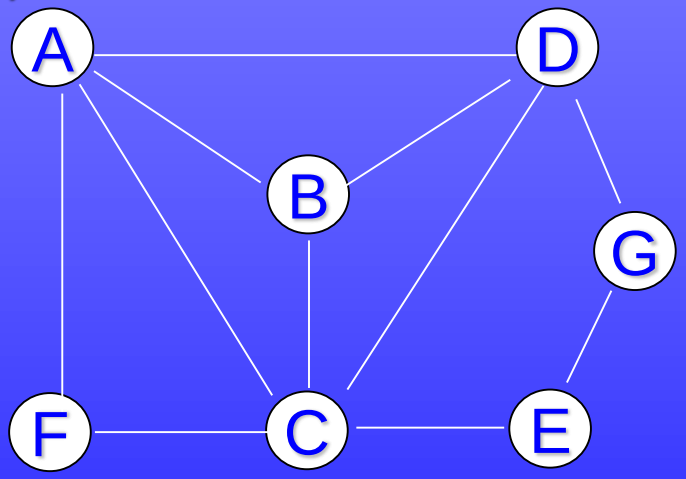
- A: B,C,F
- B: C,D
- C: -
- D: A,C
- E: C,G
- F: A,C
- G: D,E

Example: (continued) DFS of Directed Graph



Example: DFS of Undirected Graph

G=(V,E)



Adjacency List:

A: B,C,D,F

B: A,C,D

C: A,B,D,E,F

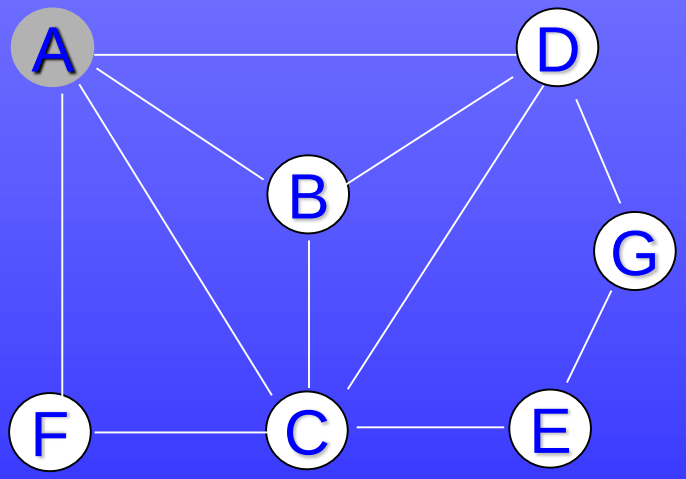
D: A,B,C,G

E: C,G

F: A,C

G: D,E

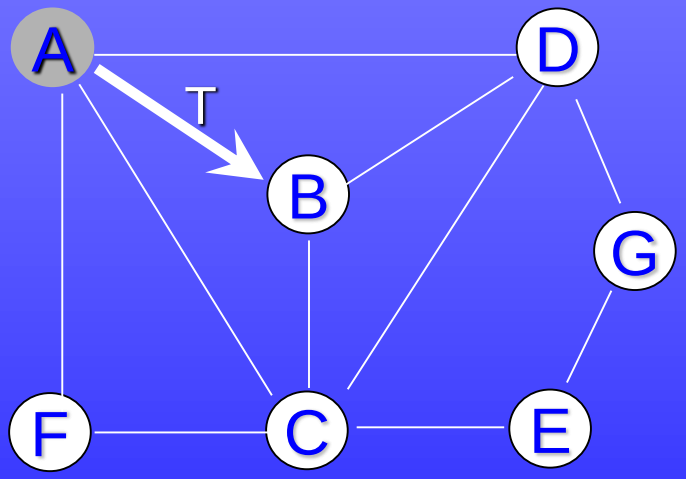
Example: DFS of Undirected Graph



Adjacency List:

A: B,C,D,F
B: A,C,D
C: A,B,D,E,F
D: A,B,C,G
E: C,G
F: A,C
G: D,E

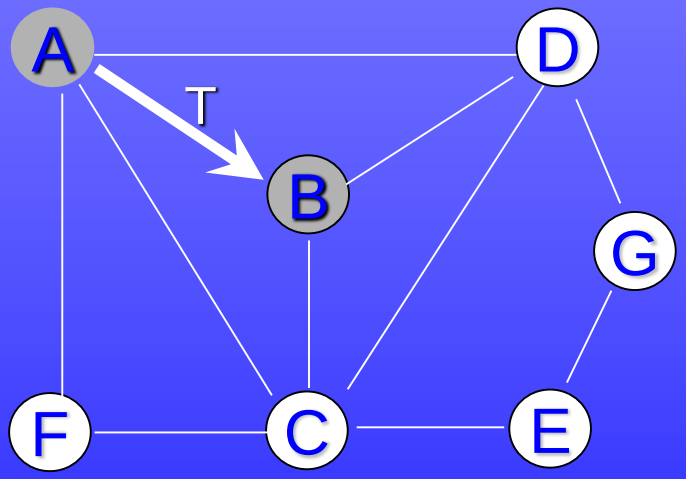
Example: DFS of Undirected Graph



Adjacency List:

A: B,C,D,F
B: A,C,D
C: A,B,D,E,F
D: A,B,C,G
E: C,G
F: A,C
G: D,E

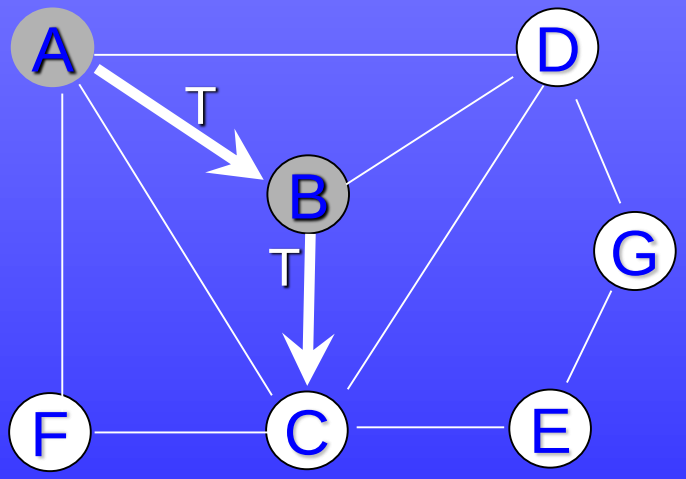
Example: DFS of Undirected Graph



Adjacency List:

A: B,C,D,F
B: A,C,D
C: A,B,D,E,F
D: A,B,C,G
E: C,G
F: A,C
G: D,E

Example: DFS of Undirected Graph



Adjacency List:

A: B,C,D,F

B: A,C,D

C: A,B,D,E,F

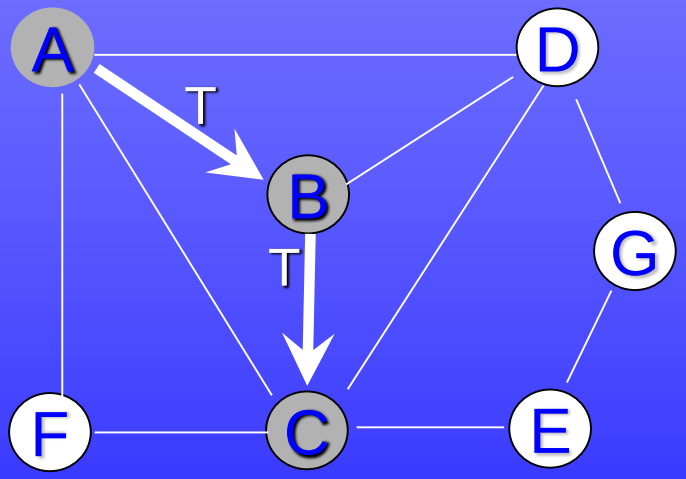
D: A,B,C,G

E: C,G

F: A,C

G: D,E

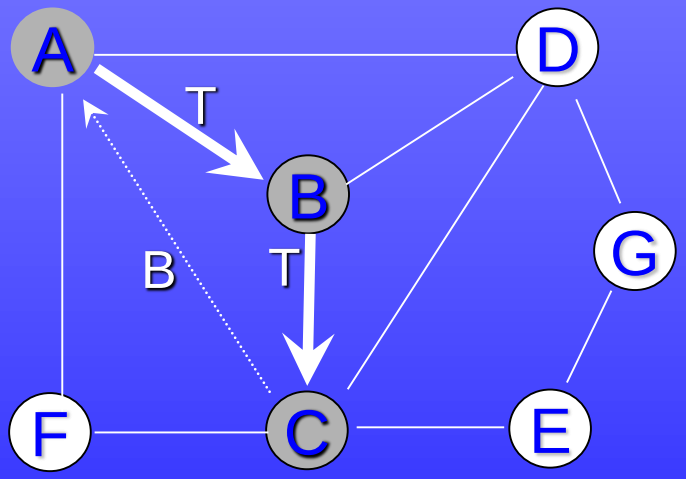
Example: DFS of Undirected Graph



Adjacency List:

A: B,C,D,F
B: A,C,D
C: A,B,D,E,F
D: A,B,C,G
E: C,G
F: A,C
G: D,E

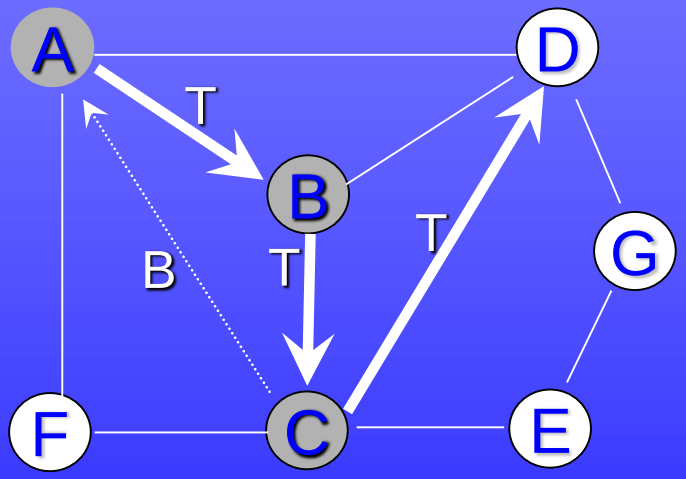
Example: DFS of Undirected Graph



Adjacency List:

A: B,C,D,F
B: A,C,D
C: A,B,D,E,F
D: A,B,C,G
E: C,G
F: A,C
G: D,E

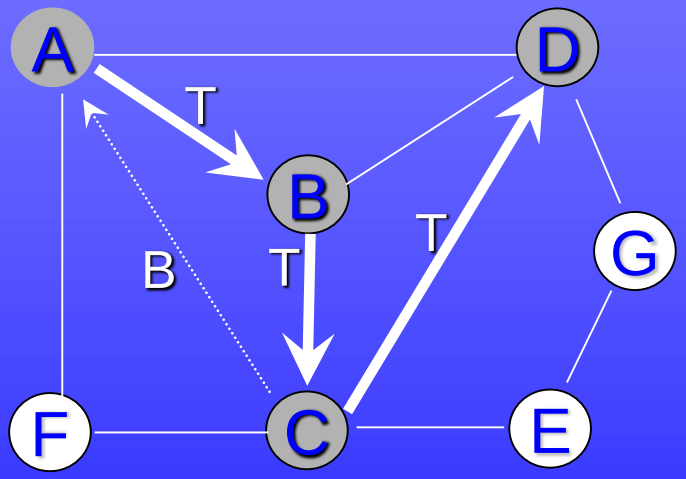
Example: DFS of Undirected Graph



Adjacency List:

A: B,C,D,F
B: A,C,D
C: A,B,D,E,F
D: A,B,C,G
E: C,G
F: A,C
G: D,E

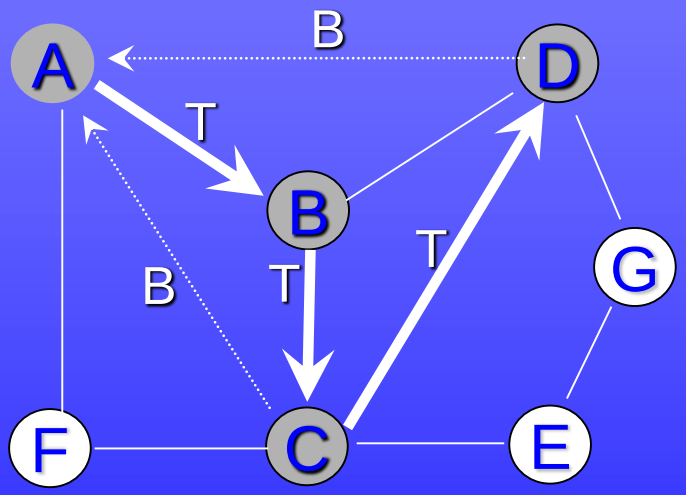
Example: DFS of Undirected Graph



Adjacency List:

A: B,C,D,F
B: A,C,D
C: A,B,D,E,F
D: A,B,C,G
E: C,G
F: A,C
G: D,E

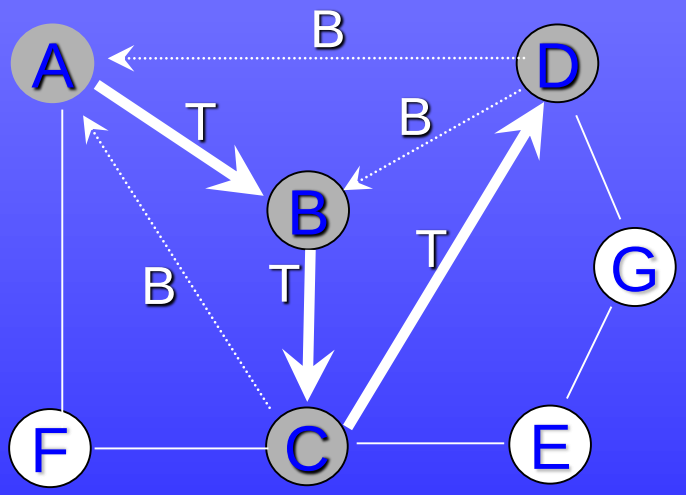
Example: DFS of Undirected Graph



Adjacency List:

A: B,C,D,F
B: A,C,D
C: A,B,D,E,F
D: A,B,C,G
E: C,G
F: A,C
G: D,E

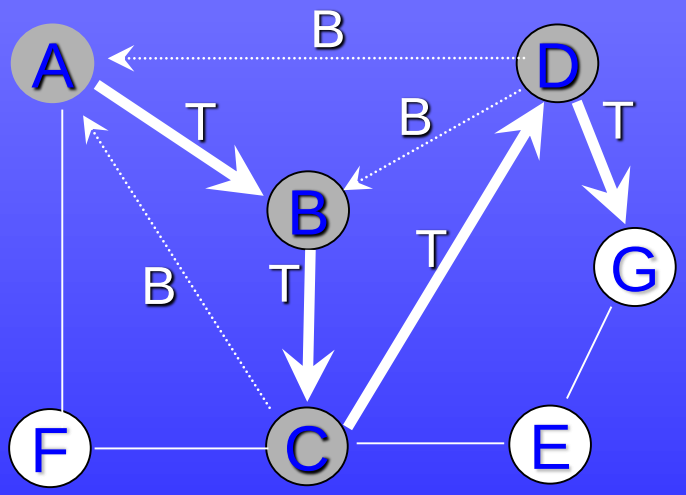
Example: DFS of Undirected Graph



Adjacency List:

A: B,C,D,F
B: A,C,D
C: A,B,D,E,F
D: A,B,C,G
E: C,G
F: A,C
G: D,E

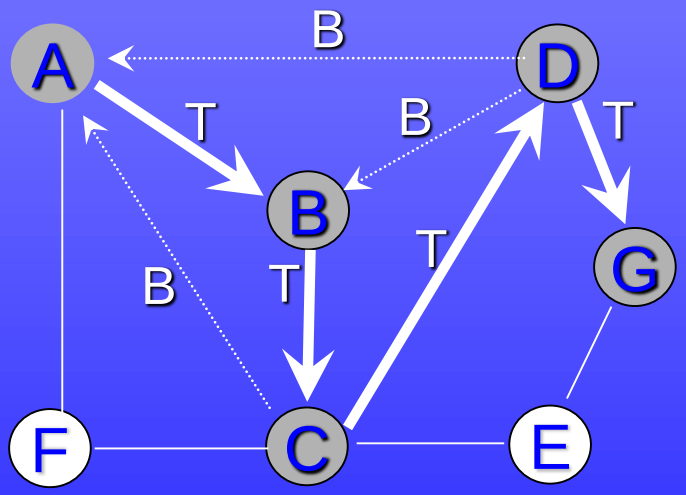
Example: DFS of Undirected Graph



Adjacency List:

A: B,C,D,F
B: A,C,D
C: A,B,D,E,F
D: A,B,C,G
E: C,G
F: A,C
G: D,E

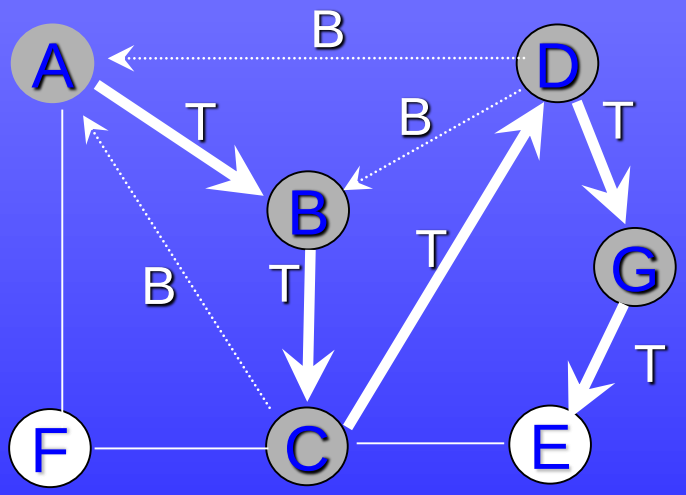
Example: DFS of Undirected Graph



Adjacency List:

A: B,C,D,F
B: A,C,D
C: A,B,D,E,F
D: A,B,C,G
E: C,G
F: A,C
G: D,E

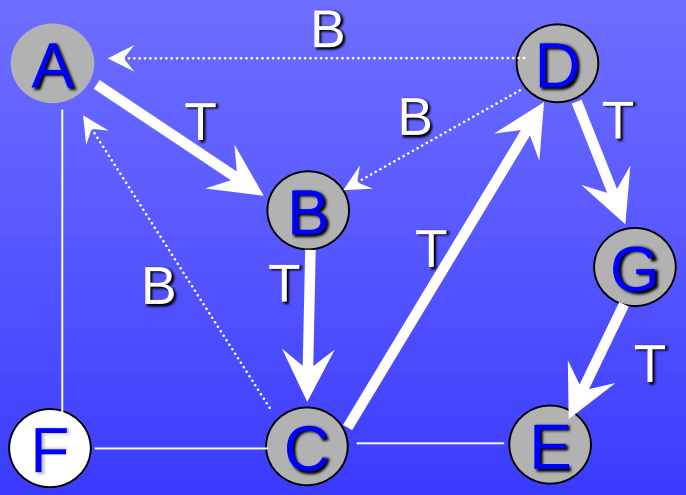
Example: DFS of Undirected Graph



Adjacency List:

A: B,C,D,F
B: A,C,D
C: A,B,D,E,F
D: A,B,C,G
E: C,G
F: A,C
G: D,E

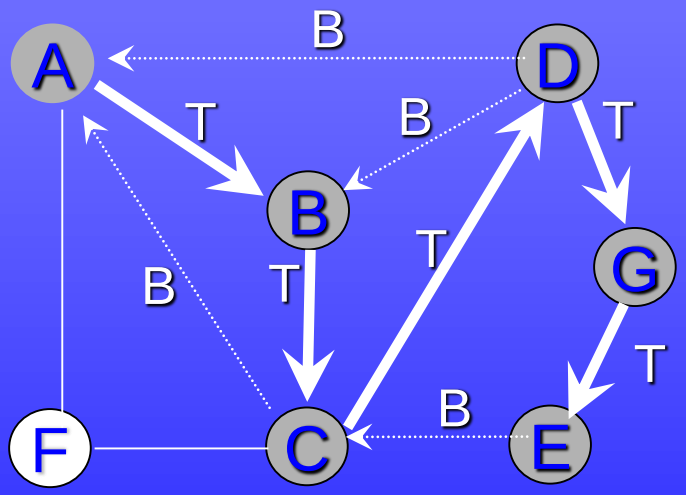
Example: DFS of Undirected Graph



Adjacency List:

A: B,C,D,F
B: A,C,D
C: A,B,D,E,F
D: A,B,C,G
E: C,G
F: A,C
G: D,E

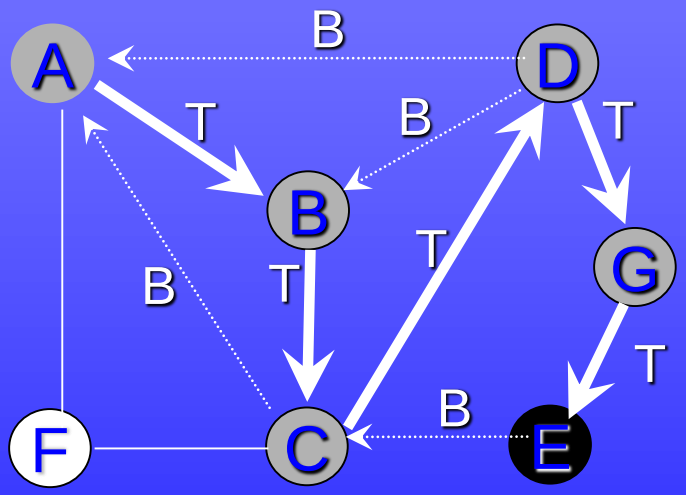
Example: DFS of Undirected Graph



Adjacency List:

A: B,C,D,F
B: A,C,D
C: A,B,D,E,F
D: A,B,C,G
E: C,G
F: A,C
G: D,E

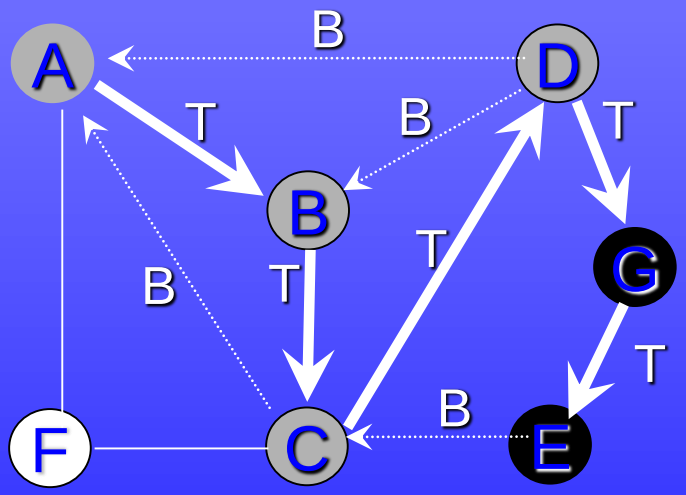
Example: DFS of Undirected Graph



Adjacency List:

A: B,C,D,F
B: A,C,D
C: A,B,D,E,F
D: A,B,C,G
E: C,G
F: A,C
G: D,E

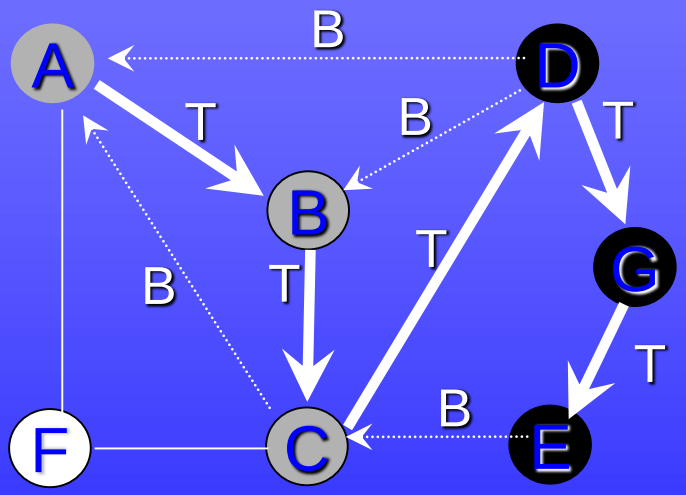
Example: DFS of Undirected Graph



Adjacency List:

A: B,C,D,F
B: A,C,D
C: A,B,D,E,F
D: A,B,C,G
√ E: C,G
F: A,C
√ G: D,E

Example: DFS of Undirected Graph



Adjacency List:

A: B,C,D,F

B: A,C,D

C: A,B,D,E,F

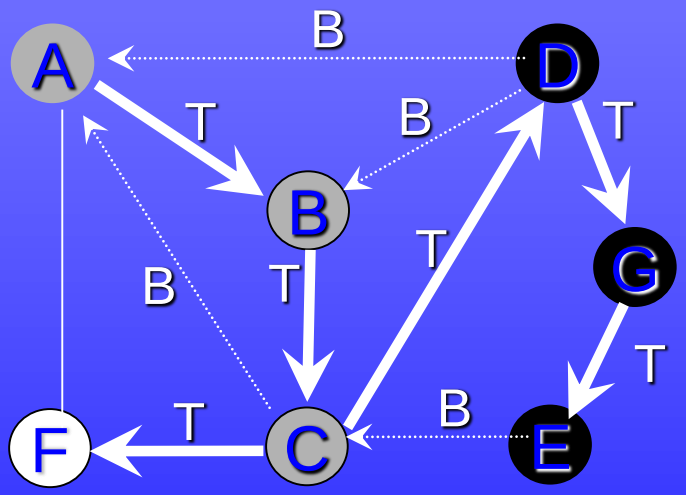
√ D: A,B,C,G

√ E: C,G

F: A,C

√ G: D,E

Example: DFS of Undirected Graph



Adjacency List:

A: B,C,D,F

B: A,C,D

C: A,B,D,E,F

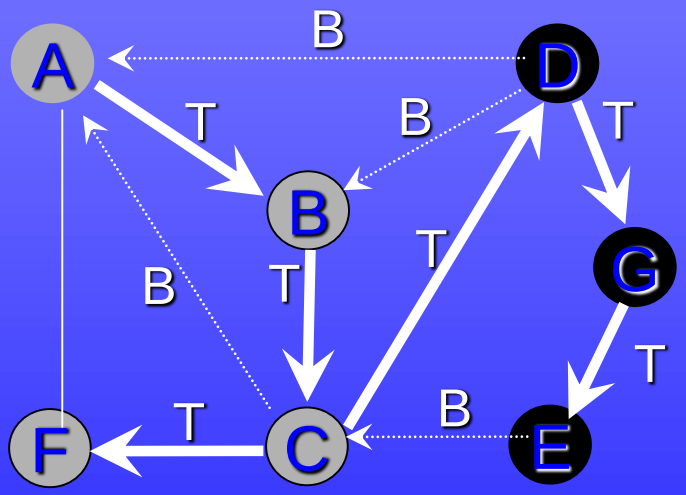
√ D: A,B,C,G

√ E: C,G

F: A,C

√ G: D,E

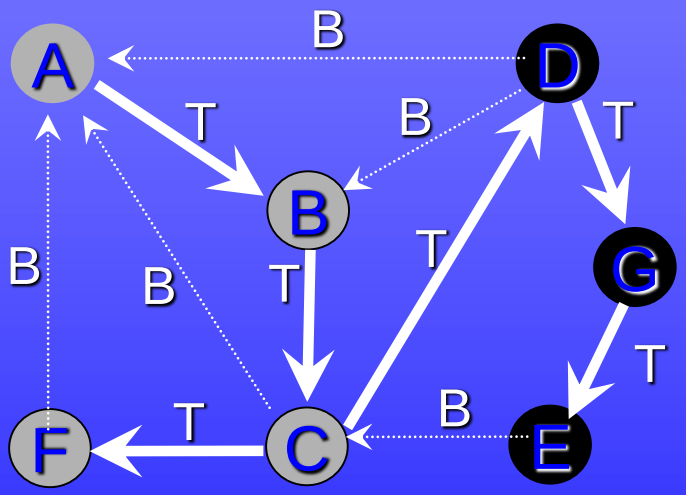
Example: DFS of Undirected Graph



Adjacency List:

- A: B,C,D,F
- B: A,C,D
- C: A,B,D,E,F
- √ D: A,B,C,G
- √ E: C,G
- F: A,C
- √ G: D,E

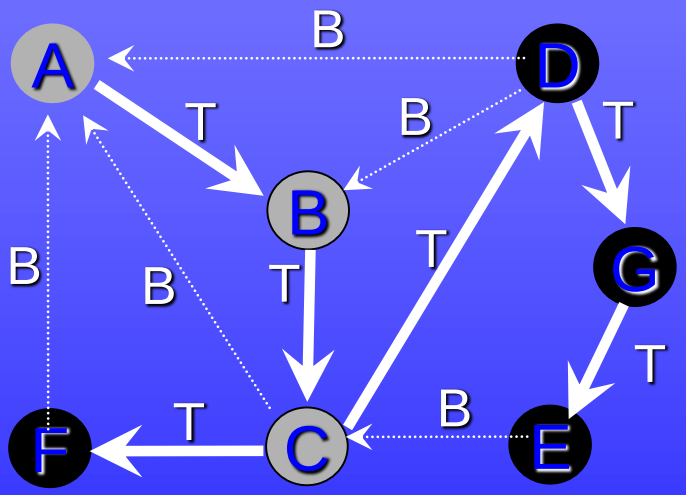
Example: DFS of Undirected Graph



Adjacency List:

A: B,C,D,F
B: A,C,D
C: A,B,D,E,F
√ D: A,B,C,G
√ E: C,G
F: A,C
√ G: D,E

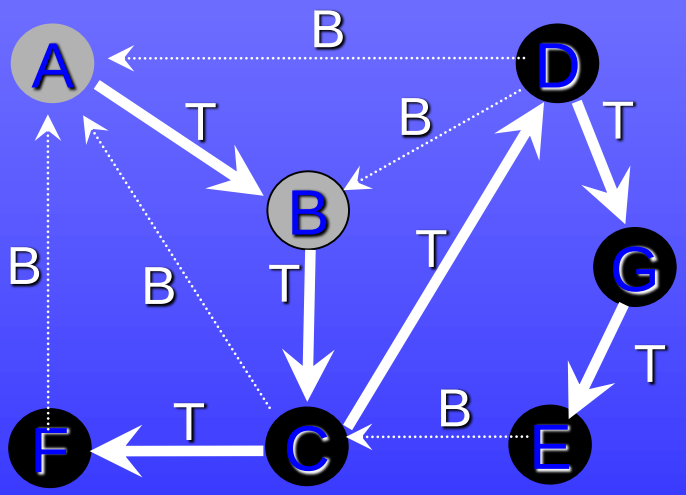
Example: DFS of Undirected Graph



Adjacency List:

- A: B,C,D,F
- B: A,C,D
- C: A,B,D,E,F
- ✓ D: A,B,C,G
- ✓ E: C,G
- ✓ F: A,C
- ✓ G: D,E

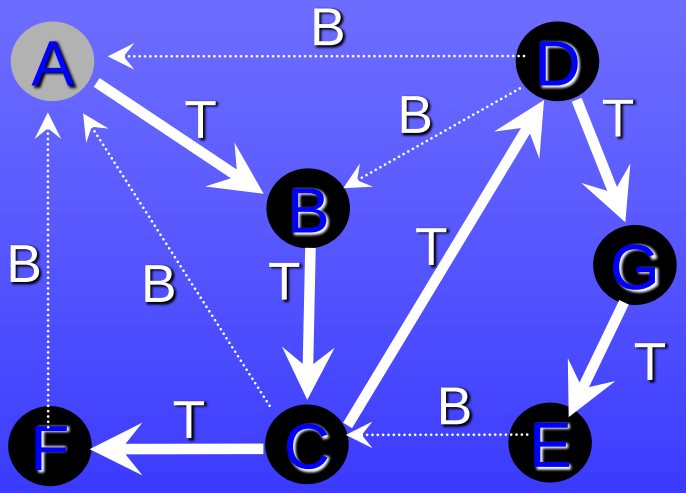
Example: DFS of Undirected Graph



Adjacency List:

- A: B,C,D,F
- B: A,C,D
- ✓ C: A,B,D,E,F
- ✓ D: A,B,C,G
- ✓ E: C,G
- ✓ F: A,C
- ✓ G: D,E

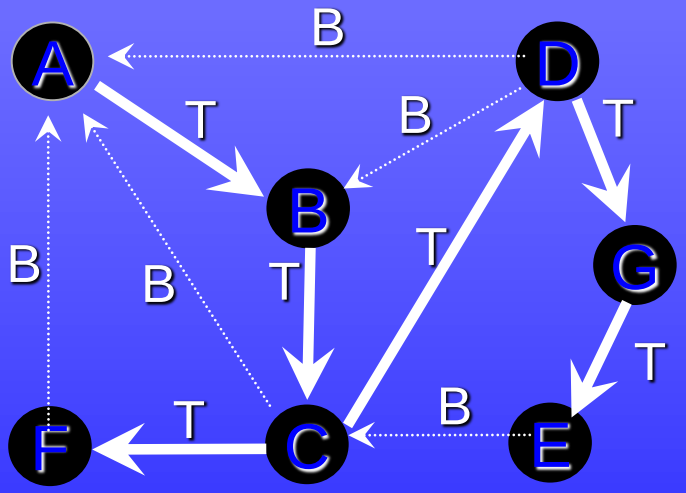
Example: DFS of Undirected Graph



Adjacency List:

- A: B,C,D,F
- ✓ B: A,C,D
- ✓ C: A,B,D,E,F
- ✓ D: A,B,C,G
- ✓ E: C,G
- ✓ F: A,C
- ✓ G: D,E

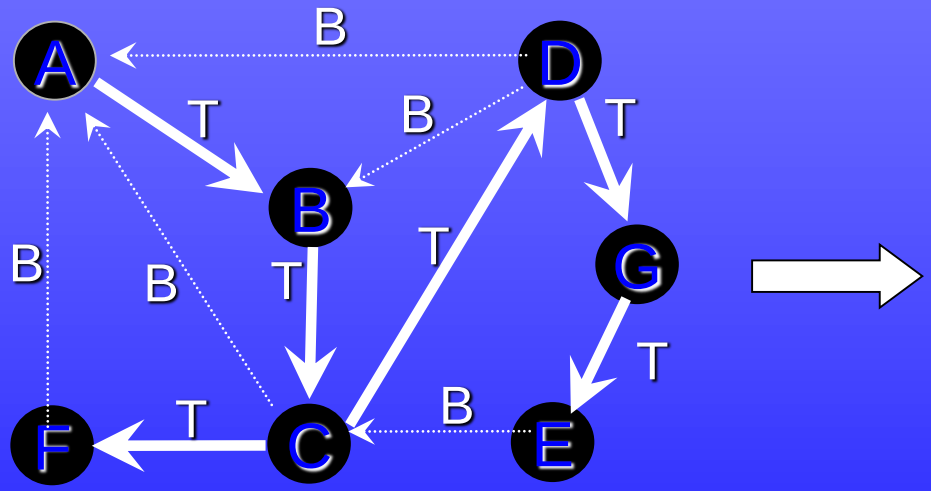
Example: DFS of Undirected Graph

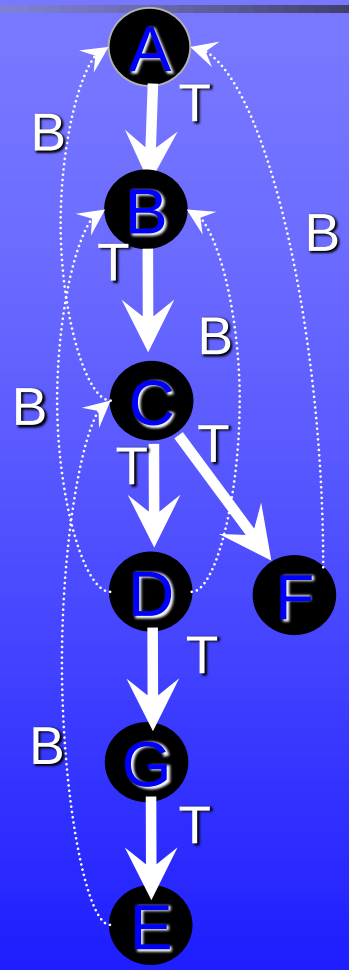


Adjacency List:

- A: B, C, D, F
- ✓ B: A, C, D
- ✓ C: A, B, D, E, F
- ✓ D: A, B, C, G
- ✓ E: C, G
- ✓ F: A, C
- ✓ G: D, E

Example: (continued) DFS of Undirected Graph

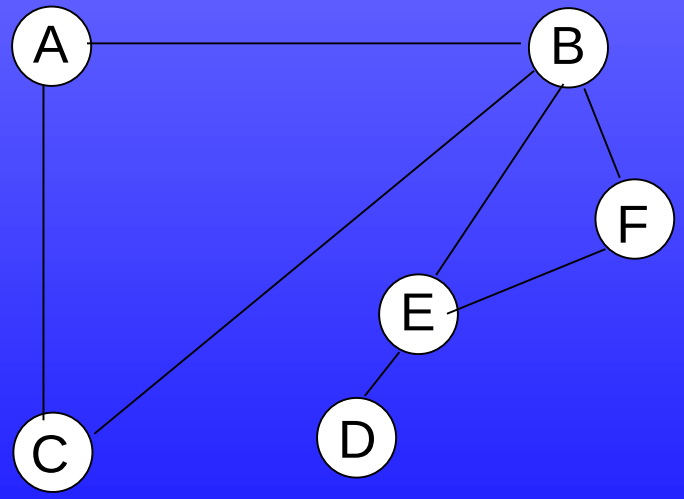


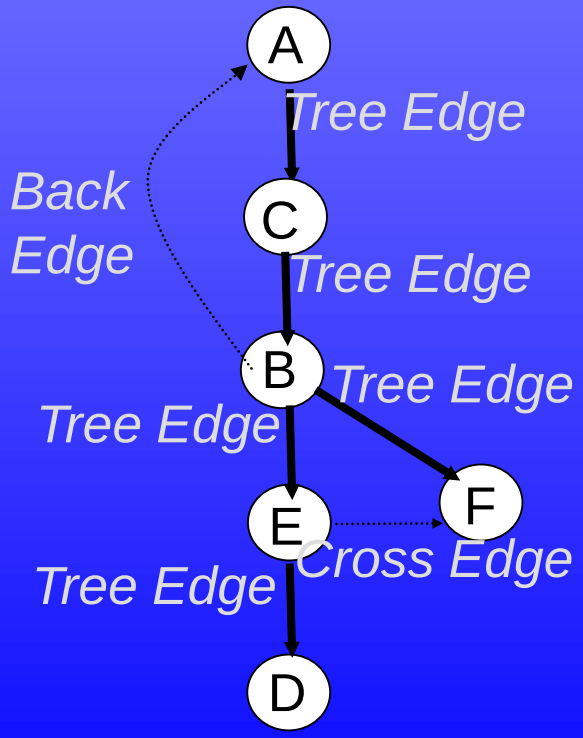


DFS Tree

Elementary Graph Algorithms: DFS

- Review problem: TRUE or FALSE?
  - **The tree shown below on the right can be a DFS tree for some adjacency list representation of the graph shown below on the left.**



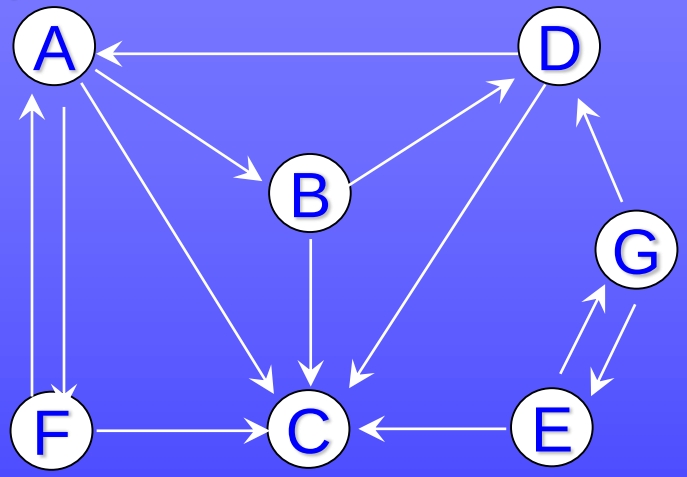


Breadth-First Search (BFS)

BFS PseudoCode

Example: BFS of Directed Graph

G=(V,E)



Adjacency List:

A: B,C,F
B: C,D
C: -
D: A,C
E: C,G
F: A,C
G: D,E

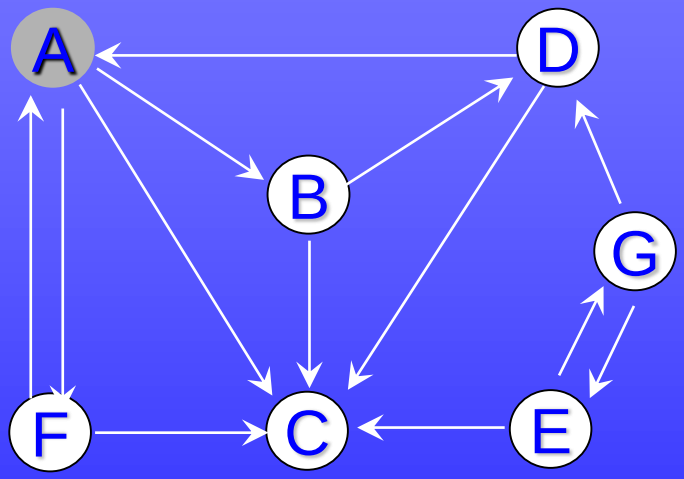
Edge Classification Legend:

T: tree edge

only tree edges are used

Source: Graph is from *Computer Algorithms: Introduction to Design and Analysis* by Baase and Gelder.

Example: BFS of Directed Graph

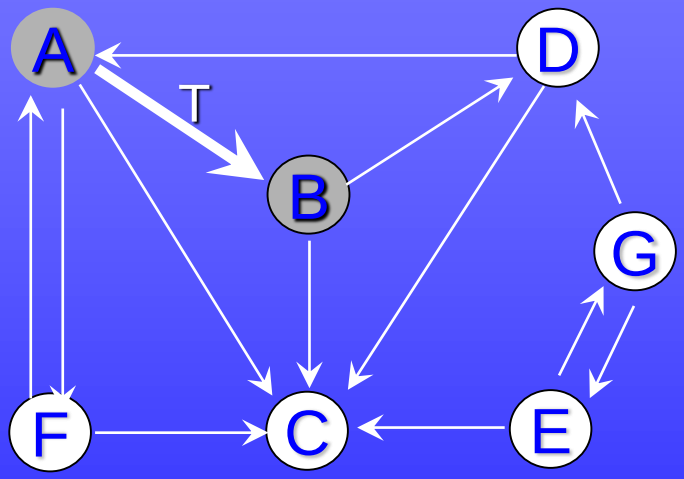


Adjacency List:

A: B,C,F
B: C,D
C: -
D: A,C
E: C,G
F: A,C
G: D,E

Queue: A

Example: BFS of Directed Graph



Adjacency List:

A: B,C,F

B: C,D

C: -

D: A,C

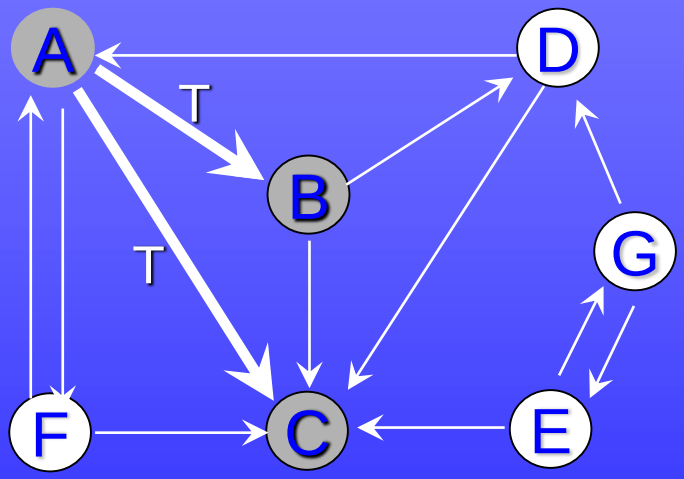
E: C,G

F: A,C

G: D,E

Queue: AB

Example: BFS of Directed Graph



Adjacency List:

A: B,C,F

B: C,D

C: -

D: A,C

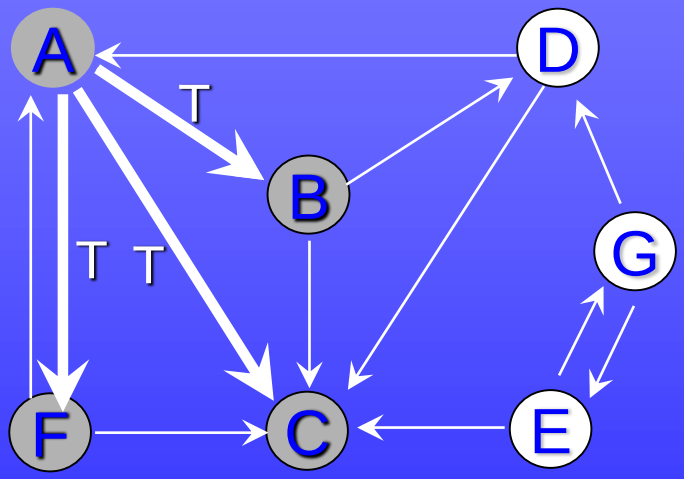
E: C,G

F: A,C

G: D,E

Queue: ABC

Example: BFS of Directed Graph



Adjacency List:

A: B,C,F

B: C,D

C: -

D: A,C

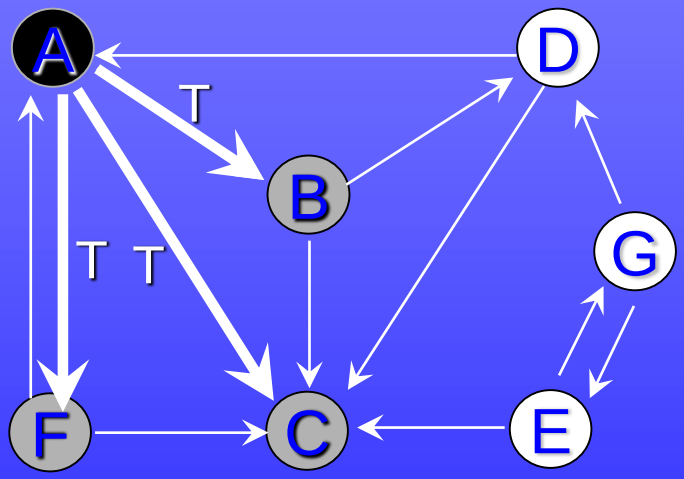
E: C,G

F: A,C

G: D,E

Queue: ABCF

Example: BFS of Directed Graph



Adjacency List:

√ A: B,C,F

B: C,D

C: -

D: A,C

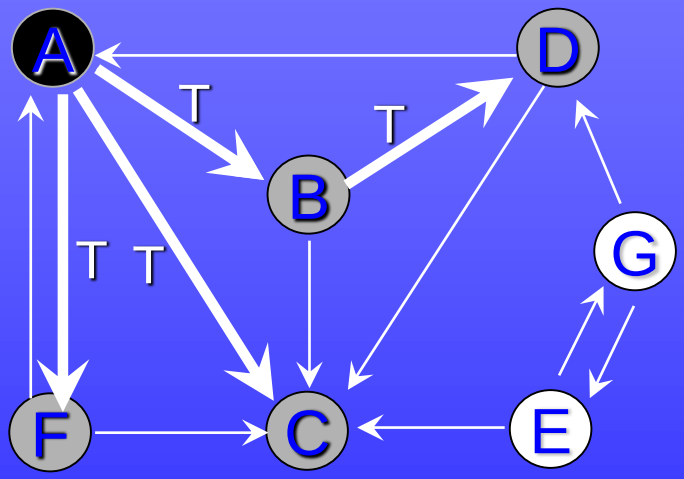
E: C,G

F: A,C

G: D,E

Queue: BCF

Example: BFS of Directed Graph



Adjacency List:

A: B,C,F

B: C,D

C: -

D: A,C

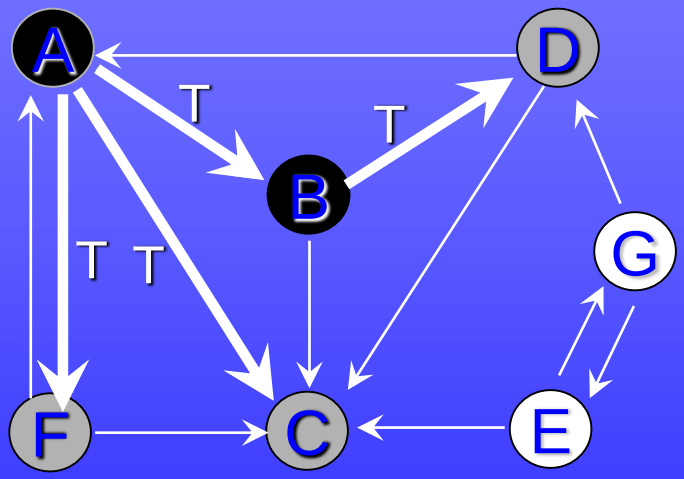
E: C,G

F: A,C

G: D,E

Queue: BCFD

Example: BFS of Directed Graph



Adjacency List:

√ A: B,C,F

√ B: C,D

C: -

D: A,C

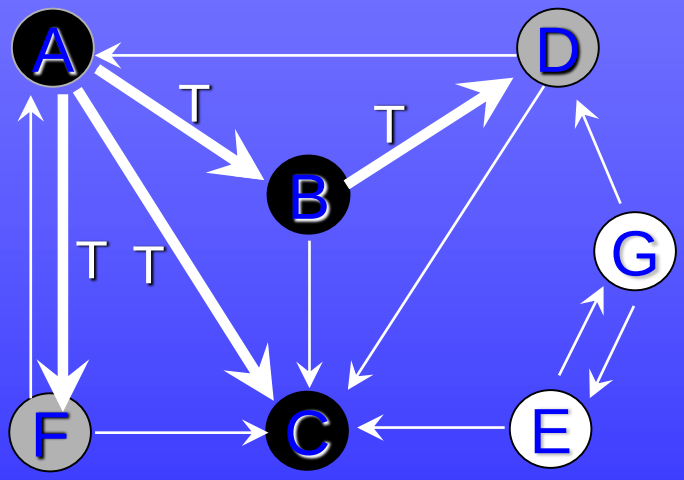
E: C,G

F: A,C

G: D,E

Queue: CFD

Example: BFS of Directed Graph



Adjacency List:

A: B,C,F

B: C,D

C: -

D: A,C

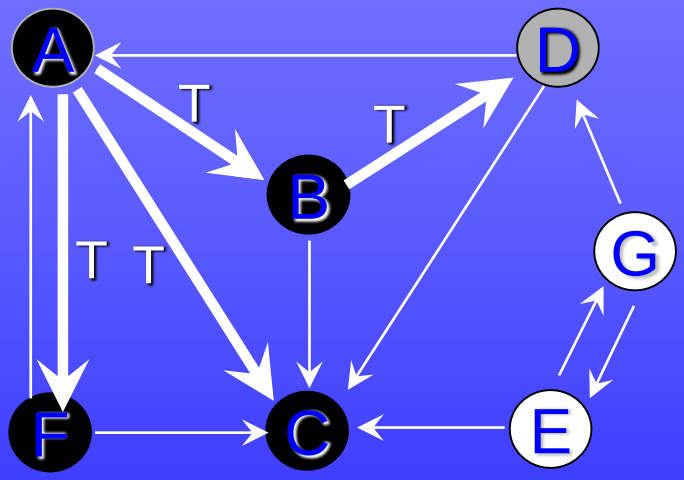
E: C,G

F: A,C

G: D,E

Queue: FD

Example: BFS of Directed Graph

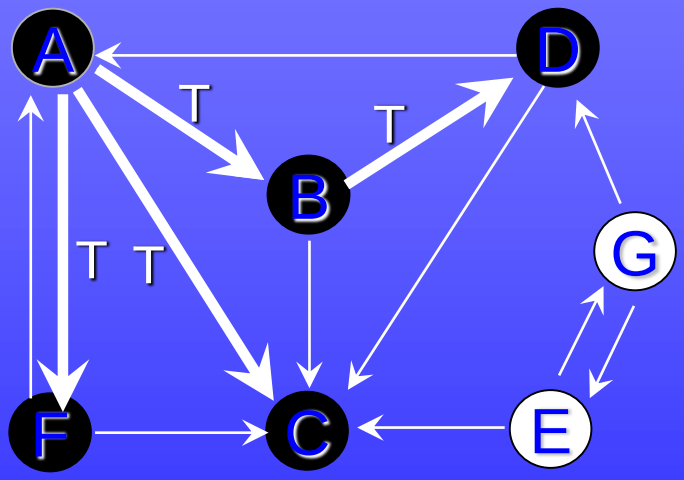


Adjacency List:

√ A: B,C,F
√ B: C,D
√ C: -
D: A,C
E: C,G
√ F: A,C
G: D,E

Queue: D

Example: BFS of Directed Graph

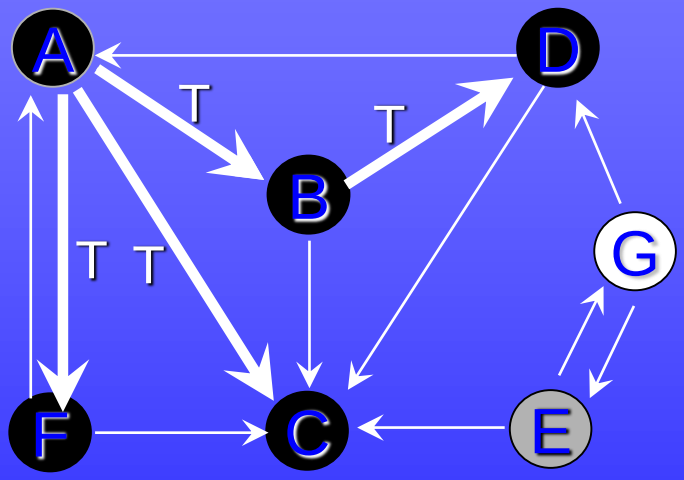


Adjacency List:

√ A: B,C,F
√ B: C,D
√ C: -
√ D: A,C
E: C,G
√ F: A,C
G: D,E

Queue: -

Example: BFS of Directed Graph

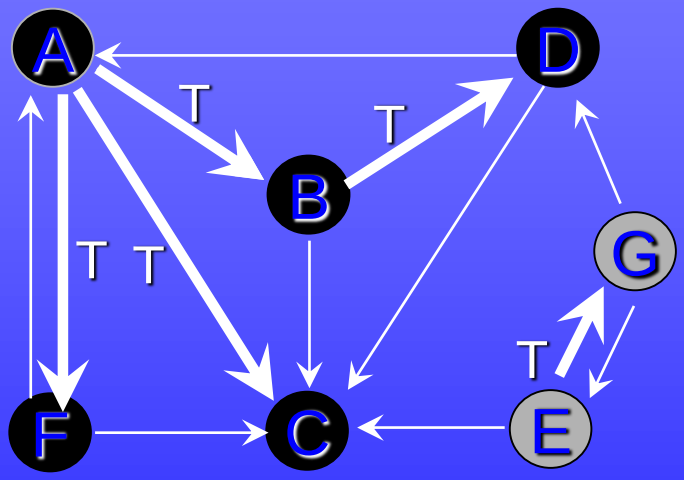


Adjacency List:

- √ A: B,C,F
- √ B: C,D
- √ C: -
- √ D: A,C
- E: C,G
- √ F: A,C
- G: D,E

Queue: E

Example: BFS of Directed Graph

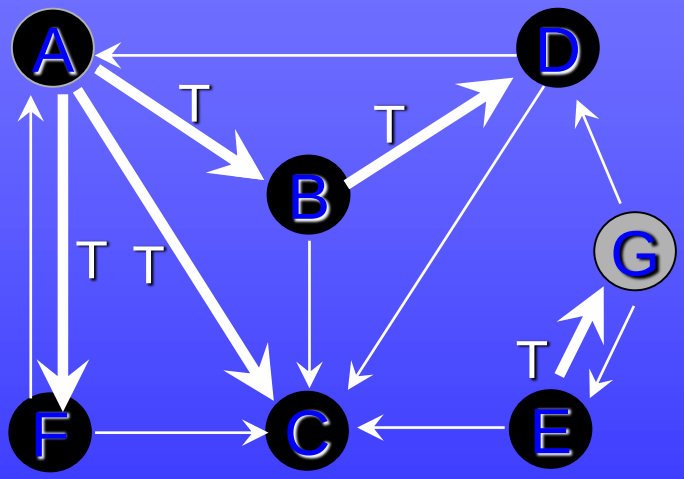


Adjacency List:

- ✓ A: B,C,F
- ✓ B: C,D
- ✓ C: -
- ✓ D: A,C
- E: C,G
- ✓ F: A,C
- G: D,E

Queue: EG

Example: BFS of Directed Graph

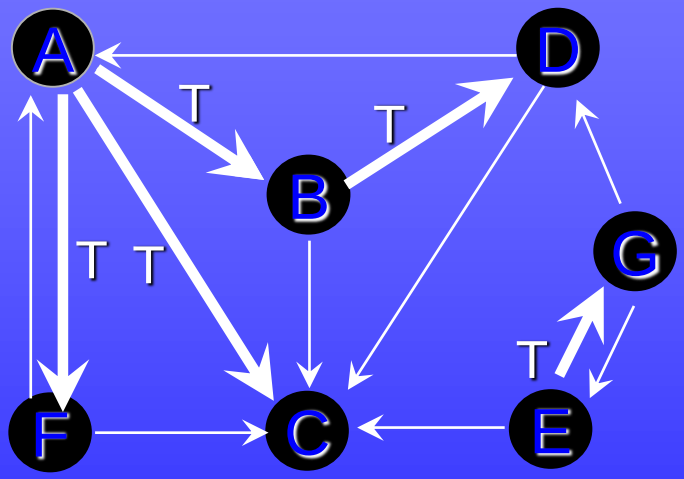


Adjacency List:

- A: B,C,F
- B: C,D
- C: -
- D: A,C
- E: C,G
- F: A,C
- G: D,E

Queue: G

Example: BFS of Directed Graph

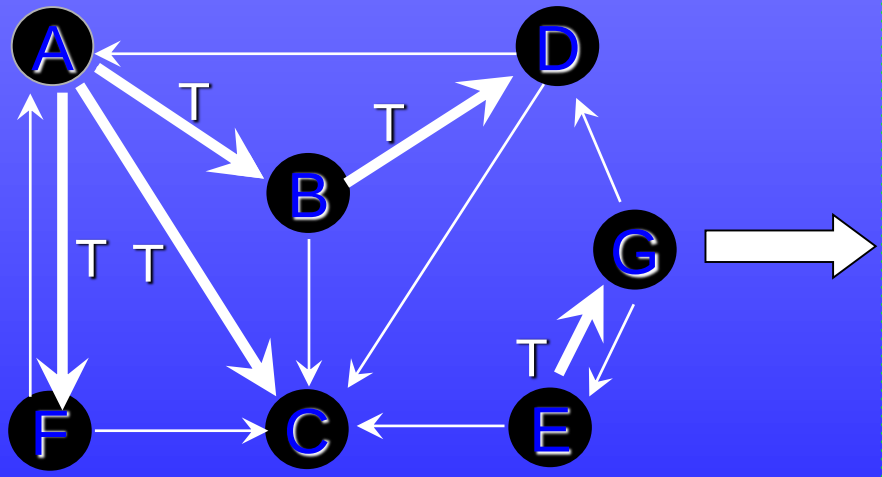


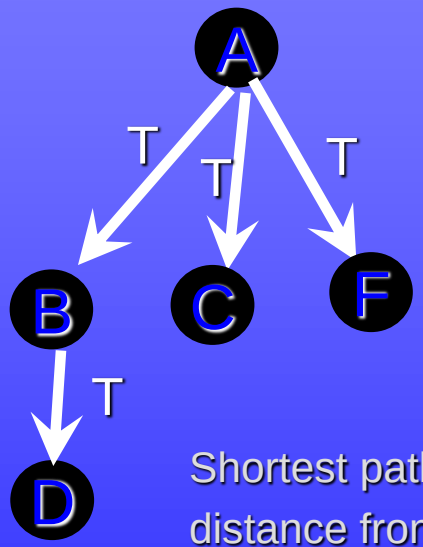
Adjacency List:

- A: B, C, F
- B: C, D
- C: -
- D: A, C
- E: C, G
- F: A, C
- G: D, E

Queue: -

Example: (continued) BFS of Directed Graph





Shortest path distance from :

A to B = 1

A to C = 1

A to F = 1

A to D = 2

BFS Tree 1

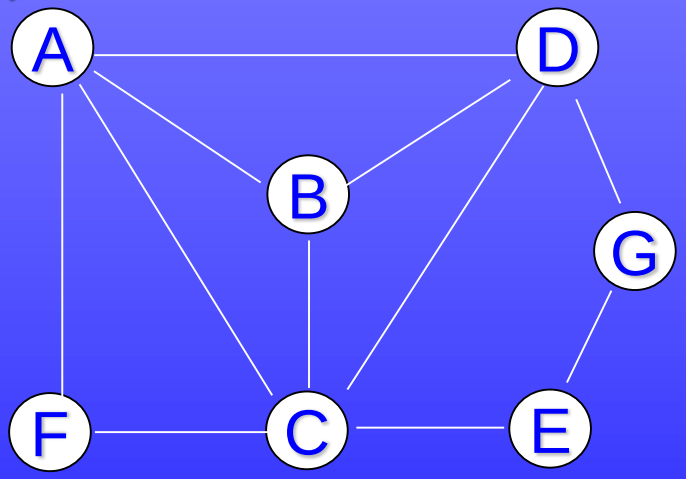
Shortest path distance from :

E to G = 1

BFS Tree 2

Example: BFS of Undirected Graph

G=(V,E)



Adjacency List:

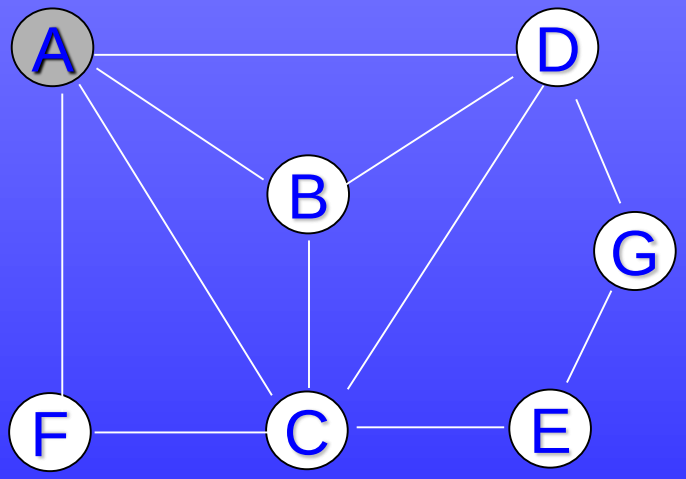
A: B,C,D,F
B: A,C,D
C: A,B,D,E,F
D: A,B,C,G
E: C,G
F: A,C
G: D,E

Edge Classification Legend:

T: tree edge

only tree edges are used

Example: BFS of Undirected Graph

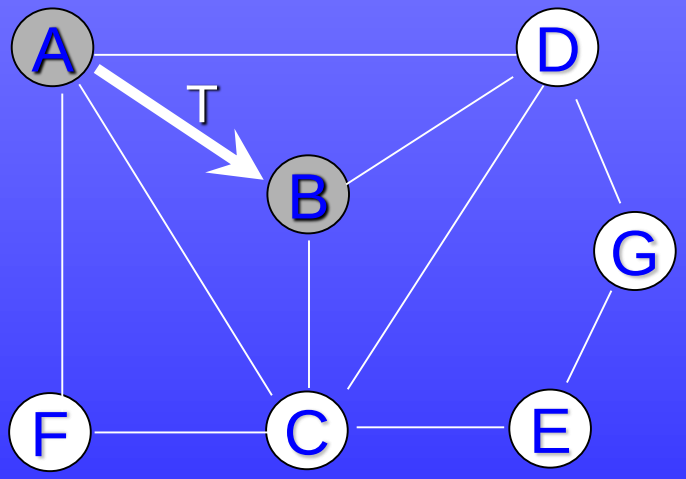


Adjacency List:

A: B,C,D,F
B: A,C,D
C: A,B,D,E,F
D: A,B,C,G
E: C,G
F: A,C
G: D,E

Queue: A

Example: BFS of Undirected Graph

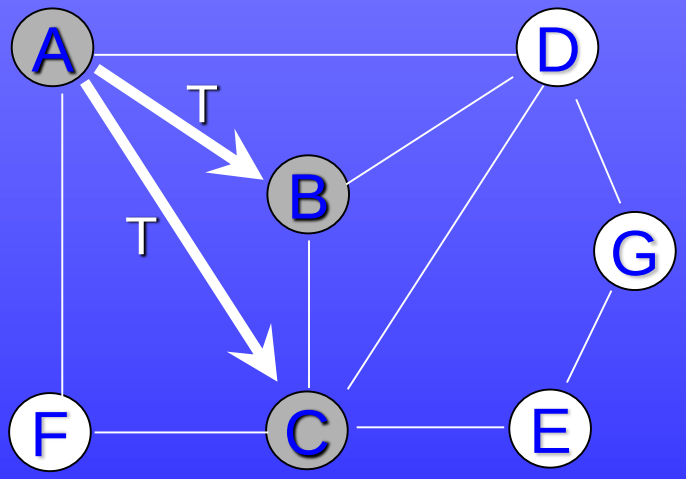


Adjacency List:

A: B,C,D,F
B: A,C,D
C: A,B,D,E,F
D: A,B,C,G
E: C,G
F: A,C
G: D,E

Queue: AB

Example: BFS of Undirected Graph

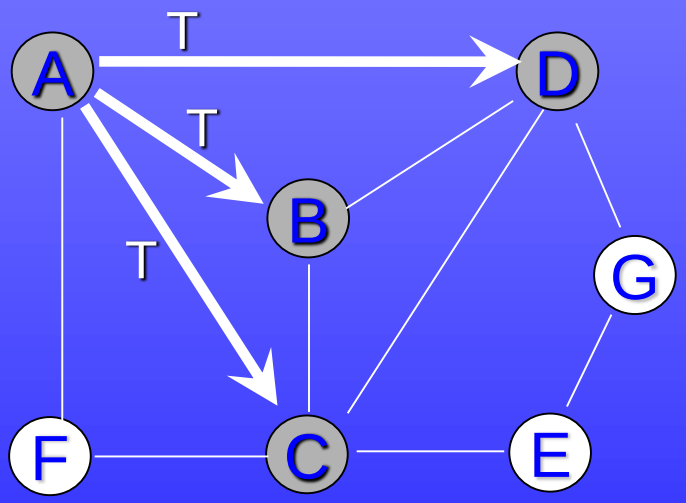


Adjacency List:

A: B,C,D,F
B: A,C,D
C: A,B,D,E,F
D: A,B,C,G
E: C,G
F: A,C
G: D,E

Queue: ABC

Example: BFS of Undirected Graph

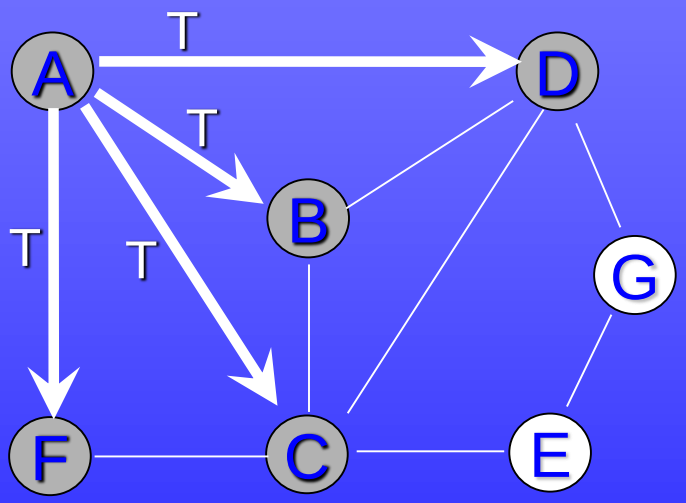


Adjacency List:

A: B,C,D,F
B: A,C,D
C: A,B,D,E,F
D: A,B,C,G
E: C,G
F: A,C
G: D,E

Queue: ABCD

Example: BFS of Undirected Graph

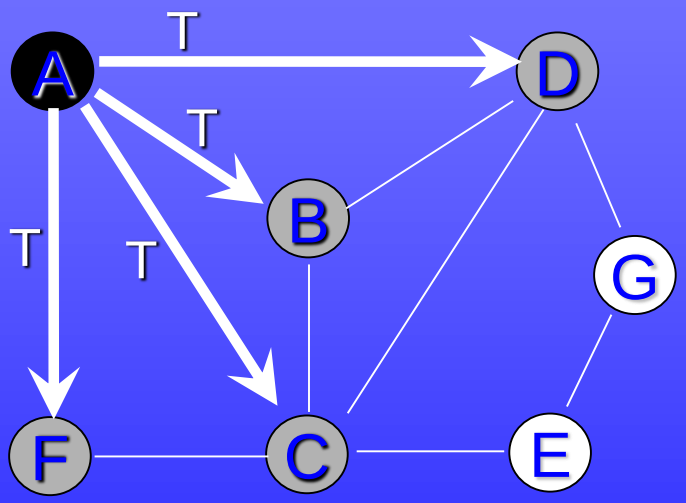


Adjacency List:

A: B,C,D,F
B: A,C,D
C: A,B,D,E,F
D: A,B,C,G
E: C,G
F: A,C
G: D,E

Queue: ABCDF

Example: BFS of Undirected Graph



Adjacency List:

A: B,C,D,F

B: A,C,D

C: A,B,D,E,F

D: A,B,C,G

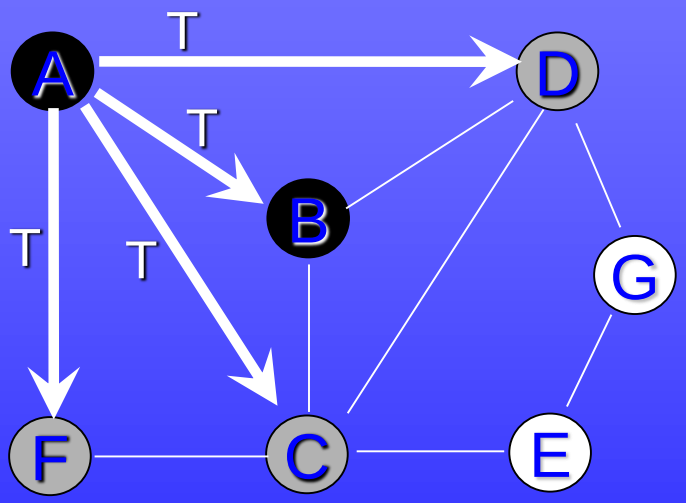
E: C,G

F: A,C

G: D,E

Queue: BCDF

Example: BFS of Undirected Graph



Adjacency List:

A: B,C,D,F

B: A,C,D

C: A,B,D,E,F

D: A,B,C,G

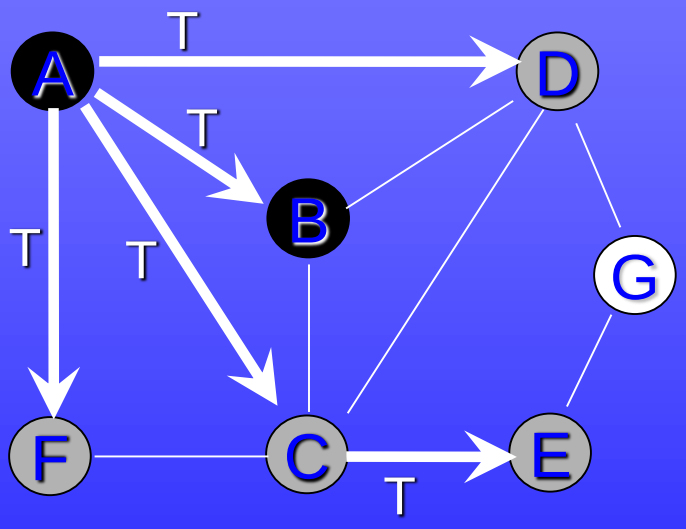
E: C,G

F: A,C

G: D,E

Queue: CDF

Example: BFS of Undirected Graph



Adjacency List:

A: B,C,D,F

B: A,C,D

C: A,B,D,E,F

D: A,B,C,G

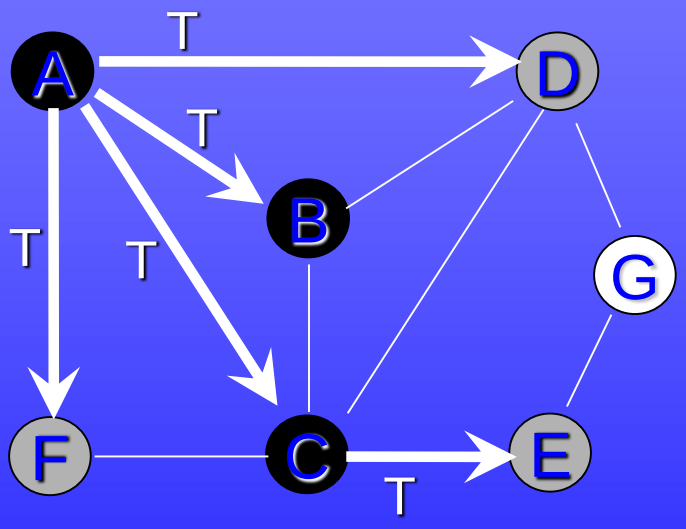
E: C,G

F: A,C

G: D,E

Queue: CDFE

Example: BFS of Undirected Graph

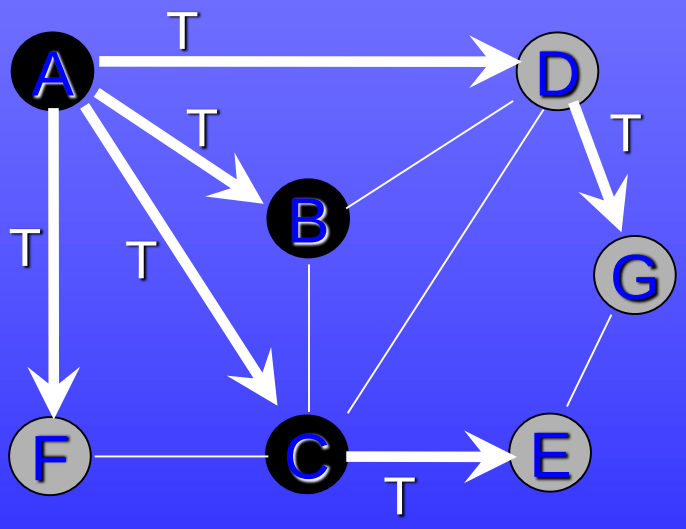


Adjacency List:

- A: B,C,D,F
- B: A,C,D
- C: A,B,D,E,F
- D: A,B,C,G
- E: C,G
- F: A,C
- G: D,E

Queue: DFE

Example: BFS of Undirected Graph

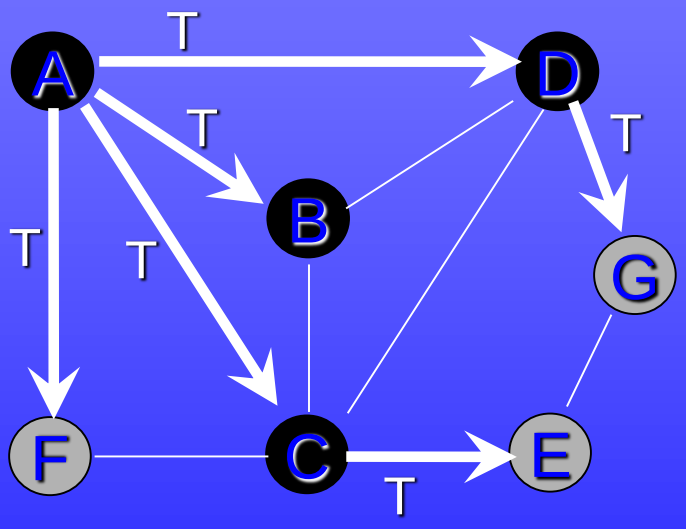


Adjacency List:

- A: B,C,D,F
- B: A,C,D
- C: A,B,D,E,F
- D: A,B,C,G
- E: C,G
- F: A,C
- G: D,E

Queue: DFEG

Example: BFS of Undirected Graph

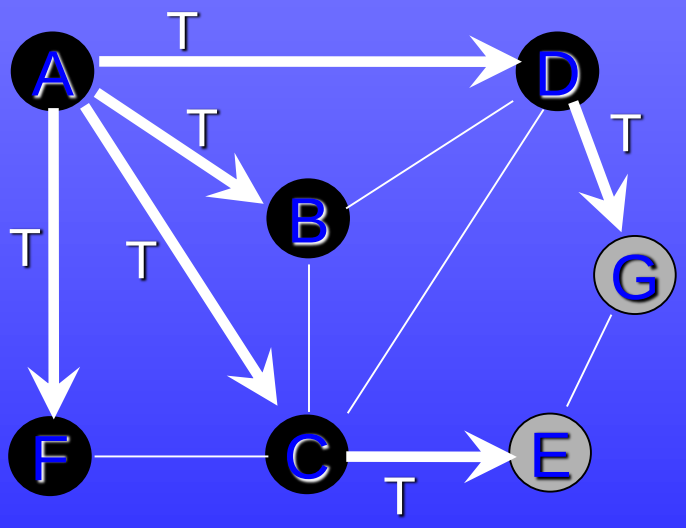


Adjacency List:

- A: B,C,D,F
- B: A,C,D
- C: A,B,D,E,F
- D: A,B,C,G
- E: C,G
- F: A,C
- G: D,E

Queue: FEG

Example: BFS of Undirected Graph

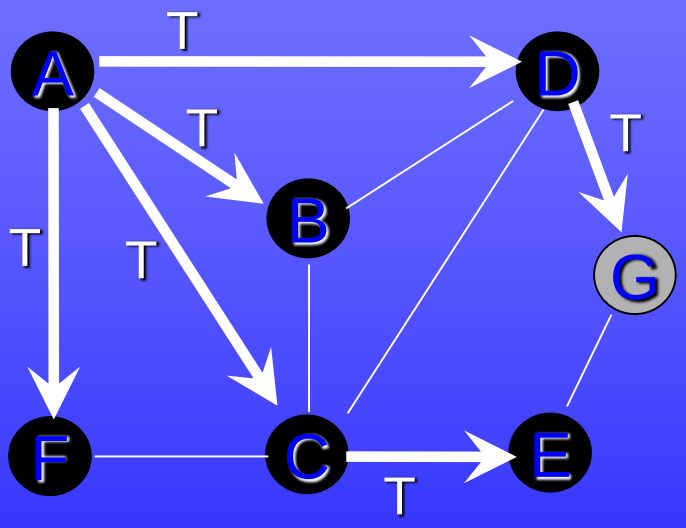


Adjacency List:

- A: B,C,D,F
- B: A,C,D
- C: A,B,D,E,F
- D: A,B,C,G
- E: C,G
- F: A,C
- G: D,E

Queue: EG

Example: BFS of Undirected Graph

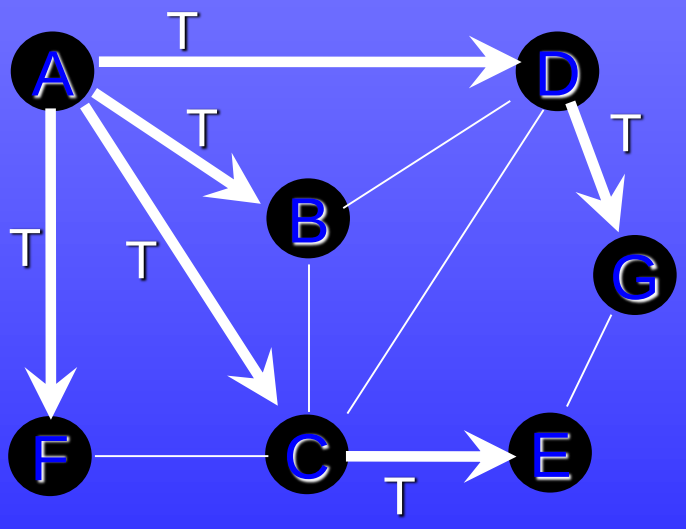


Adjacency List:

- A: B,C,D,F
- B: A,C,D
- C: A,B,D,E,F
- D: A,B,C,G
- E: C,G
- F: A,C
- G: D,E

Queue: G

Example:
BFS of Undirected Graph

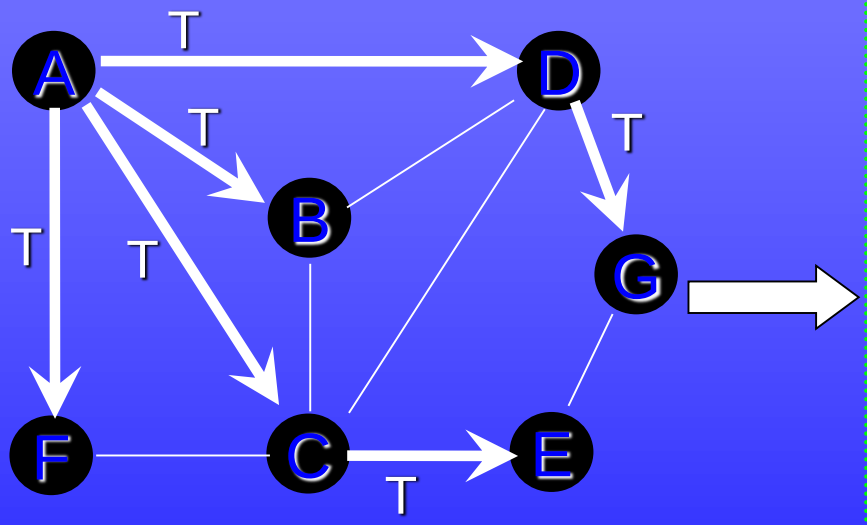


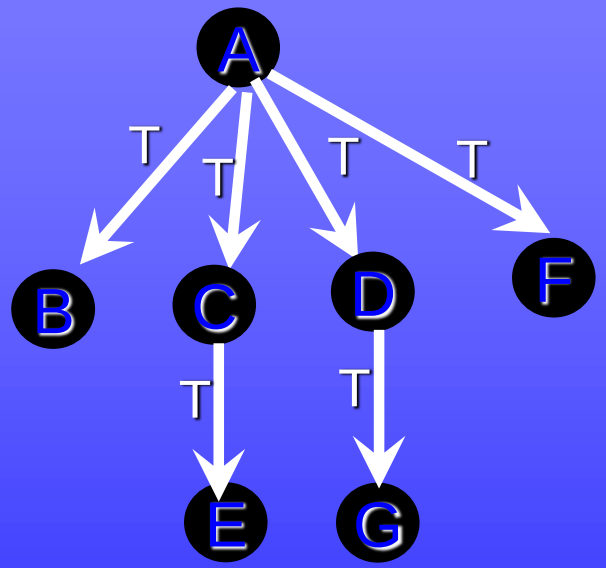
Adjacency List:

- A: B,C,D,F
- B: A,C,D
- C: A,B,D,E,F
- D: A,B,C,G
- E: C,G
- F: A,C
- G: D,E

Queue: -

Example: (continued) BFS of Undirected Graph





Shortest path distance from :

A to B = 1

A to C = 1

A to D = 1

A to F = 1

A to E = 2

A to G = 2

BFS Tree

Depth-First Search (DFS)
&
Breadth-First Search (BFS)

Using the Results of DFS & BFS
Running Time Analysis

Using the Results of DFS & BFS

- Using DFS to Detect Cycles:

A directed graph G is acyclic if and only if a Depth-First Search of G yields no back edges.

- Using BFS for Shortest Paths:

see p. 486 of text for proof

Note: DFS can also be used to detect cycles in undirected graphs if notion of cycle is defined appropriately.

A Breadth-First Search of G yields shortest path information:

For each Breadth-First Search tree, the path from its root u to a vertex v yields the shortest path from u to v in G.

see p. 472-475 of text for proof

Elementary Graph Algorithms: *SEARCHING*: DFS, BFS

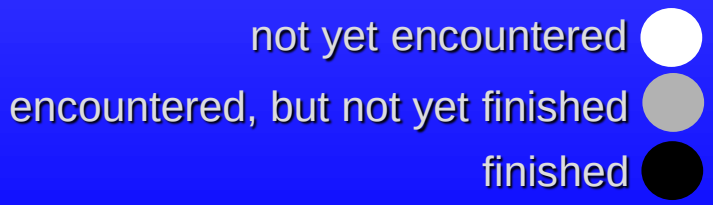
for *unweighted* <u>directed</u> or <u>undirected</u> graph G=(V,E)

<u>Time</u>: O(|V| + |E|) adj list O($|V|^2$) adj matrix

predecessor subgraph = forest of spanning trees

- Breadth-First-Search (BFS):
  - Shortest Path Distance
    - From *source* to each reachable vertex
    - Record during traversal
  - Foundation of many "*shortest path*" algorithms

Vertex color shows status:



- Depth-First-Search (DFS):
  - Encountering, finishing times
    - "well-formed" nested (( )( )) structure
  - DFS of undirected graph produces only back edges or tree edges
  - Directed graph is acyclic if and only if DFS yields no back edges

See DFS, BFS Handout for PseudoCode

Running Time Analysis

- Key ideas in the analysis are similar for DFS and BFS. In both cases *we assume an Adjacency List representation*. Let's examine DFS.
- Let **t** be number of DFS trees generated by DFS search
- Outer loop in DFS(G) executes **t** times
  - each execution contains call: DFS_Visit(G,u)
  - each such call constructs a DFS tree by visiting (recursively) every node reachable from vertex u
  - Time: $\sum_{i=1}^{t} time\ to\ construct\ DFS\ tree\ i$
- Now, let r_i be the number of vertices in DFS tree i
- Time to construct DFS tree i:

$$\sum_{j=1}^{r_i} |AdjList[\ jth\ vertex\ in\ DFS\ tree\ i]|$$

```
DFS(G)
1   for each vertex u ε V[G]
2       do color[u] ◄——— WHITE
3   for each vertex u ε V[G]
4       do if color[u] is WHITE
5           then DFS_VISIT(G, u)
```

```
DFS_VISIT(G, u)
1   color[u] ◄——— GRAY
2   for each vertex v ε AdjList[u]
3       do if color[v] is WHITE
4           then DFS_VISIT(G,v)
5   color[u] ◄——— BLACK
```

continued on next slide...

Running Time Analysis

- Total DFS time: $\sum_{i=1}^{t} \sum_{j=1}^{r_i} |AdjList[\ jth\ vertex\ in\ DFS\ tree\ i]|$
- Now, consider this expression for the extreme values of t:
  - if t=1, all edges are in one DFS tree and the expression simplifies to O(|E|)
  - if t=|V|, each vertex is its own (degenerate) DFS tree with no edges so the expression simplifies to O(|V|)
  - O(|V|+|E|) is therefore an upper bound on the time for the extreme cases
- For values of t in between 1 and |V| we have these contributions to running time:
  - 1 for each vertex that is its own (degenerate) DFS tree with no edges
    - upper bound on this total is O(|V|)
  - |AdjList[u]| for each vertex u that is the root of a non-degenerate DFS tree
    - upper bound on this total is O(|E|)
  - Total time for values of t in between 1 and |V| is therefore also O(|V|+|E|)

Total time= $O(|V| + |E|)$

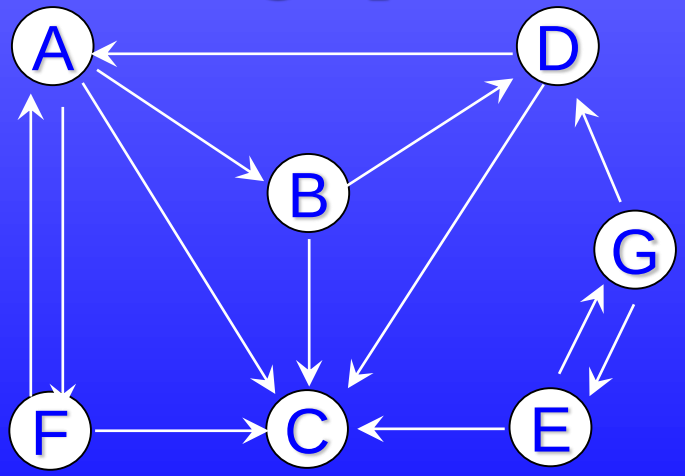
Note that for an Adjacency Matrix representation, we would need to scan an entire matrix row (containing |V| entries) each time we examined the vertices adjacent to a vertex. This would make the running time $O(|V|^2)$ instead of O(|V|+|E|).

Topological Sort

Source: Previous 91.404 instructors

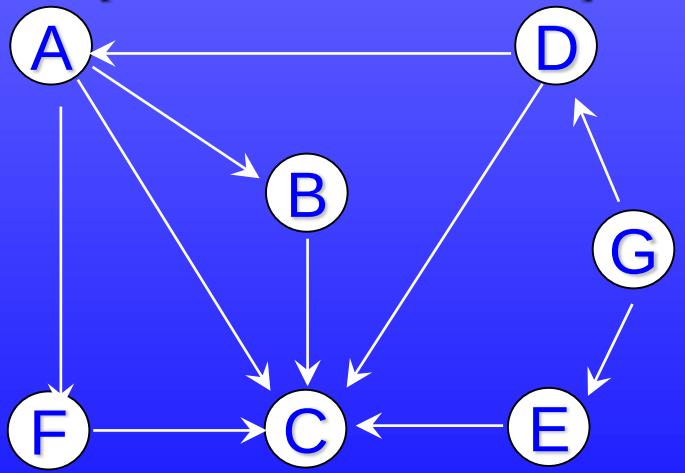
Definition: DAG

- A **Directed Acyclic Graph often abbreviated DAG**
- DAGs used in many applications to indicate precedences among events.
- **If DFS of a directed graph yields no back edges, then the graph contains no cycles [Lemma 23.10 in text]**



This graph has more than one cycle.

Can you find them all?



This graph has no cycles, so it is a DAG.

Definition: Topological Sort

- A **topological sort of a DAG G** = (V, E) is a linear ordering of all its vertices such that if G contains an edge (u, v), then u appears before v in the ordering.
  - If the graph is not acyclic, then no linear ordering is possible.
  - A topological sort of a graph can be viewed as an ordering of its vertices along a horizontal line so that all directed edges go from left to right.
  - Topological sorting is thus different from the usual kind of "sorting" .

Elementary Graph Algorithms: Topological Sort

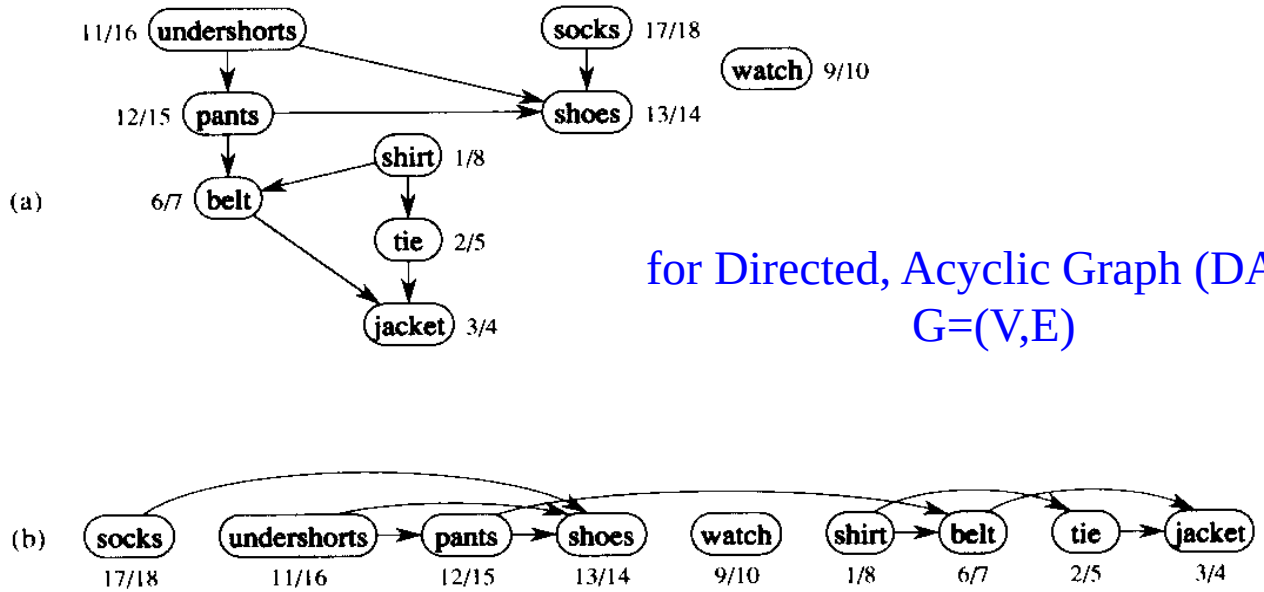


Figure 23.7 (a) Professor Bumstead topologically sorts his clothing when getting dressed. Each directed edge (u, v) means that garment u must be put on before garment v. The discovery and finishing times from a depth-first search are shown next to each vertex. (b) The same graph shown topologically sorted. Its vertices are arranged from left to right in order of decreasing finishing time. Note that all directed edges go from left to right.

for Directed, Acyclic Graph (DAG)
G=(V,E)

TOPOLOGICAL-SORT(G)
1 DFS(G) computes "finishing times" for each vertex
2 as each vertex is finished, insert it onto front of list
3 return list

Produces linear ordering of vertices.
For edge (u,v), u is ordered before v.

See also 91.404 DFS/BFS slide show

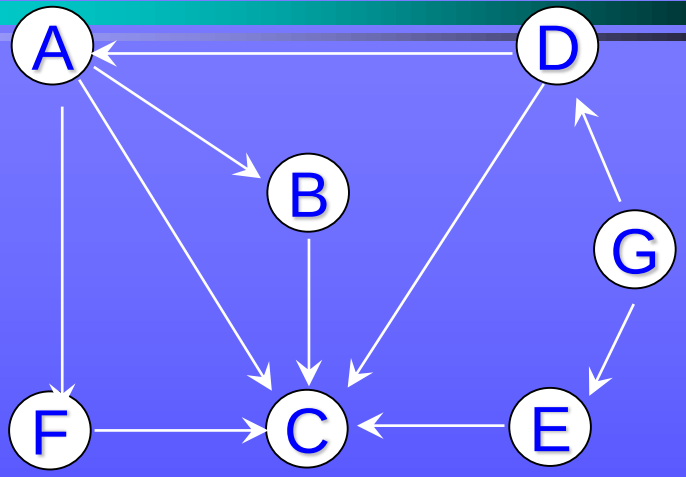
source: 91.503 textbook Cormen et al.

Topological Sort

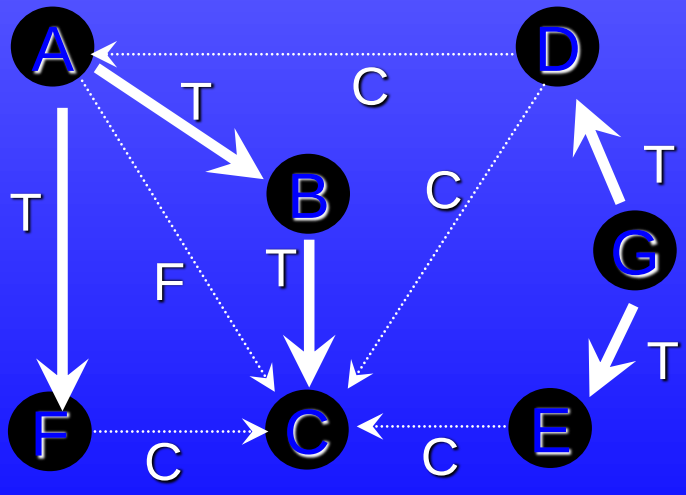
- The following algorithm topologically sorts a DAG:
- TOPOLOGICAL-SORT(G)
  - call DFS(G) to compute finishing times f[v] for each vertex v (this is equal to the order in which vertices change color from gray to black)
  - as each vertex is finished (turns black), insert it onto the front of a linked list
  - return the linked list of vertices

Example

- For this DAG:

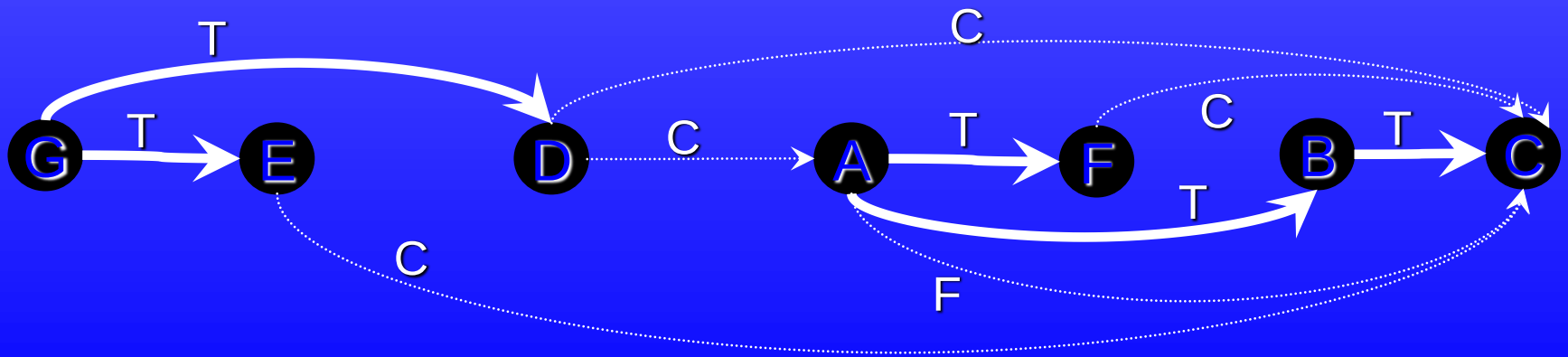


- DFS produces this result:
  - this contains 2 DFS trees
- Vertices are blackened in the following order:
  - C, B, F, A, D, E , G



Example

- Vertices are added to *front* of a linked list in the blackening order.
- Final result is shown below
- Note that all tree edges and non-tree edges point to the right



Topological Sort

- We can perform a topological sort in time O(V + E), since depth-first search takes O(V + E) time and it takes 0(1) time to insert each of the |V| vertices onto the front of the linked list.

Topological Sort

- **Theorem 23.11:** TOPOLOGICAL-SORT(G) produces a topological sort of a directed acyclic graph G.
- **Proof:** Suppose that DFS is run on a given dag G = (V, E) to determine finishing times for its vertices. It suffices to show that for any pair of distinct vertices u,v Î V, if there is an edge in G from u to v, then f[v] < f[u]. Consider any edge (u,v) explored by DFS(G). When this edge is explored, v cannot be gray, since then v would be an ancestor of u and (u,v) would be a back edge, contradicting Lemma 23.10. Therefore, v must be either white or black. If v is white, it becomes a descendant of u, and so f[v] < f[u]. If v is black, then f[v] < f[u] as well. Thus, for any edge (u,v) in the dag, we have f[v] < f[u], proving the theorem.

Chapter 24
Minimum Spanning Trees

Kruskal
Prim

[Source: Cormen et al. textbook except where noted]

Minimum Spanning Tree: Basics

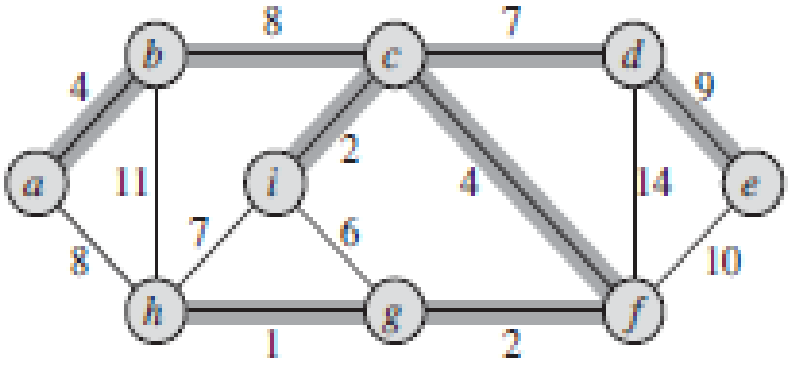


Figure 23.1 A minimum spanning tree for a connected graph. The weights on edges are shown, and the edges in a minimum spanning tree are shaded. The total weight of the tree shown is 37. This minimum spanning tree is not unique: removing the edge (b, c) and replacing it with the edge (a, h) yields another spanning tree with weight 37.

source: 91.503 textbook Cormen et al.

Growing Minimum Spanning Tree:

GENERIC-MST(G, w)

```
1  A = ∅
2  while A does not form a spanning tree
3      find an edge (u, v) that is safe for A
4      A = A ∪ {(u, v)}
5  return A
```

We use the loop invariant as follows:

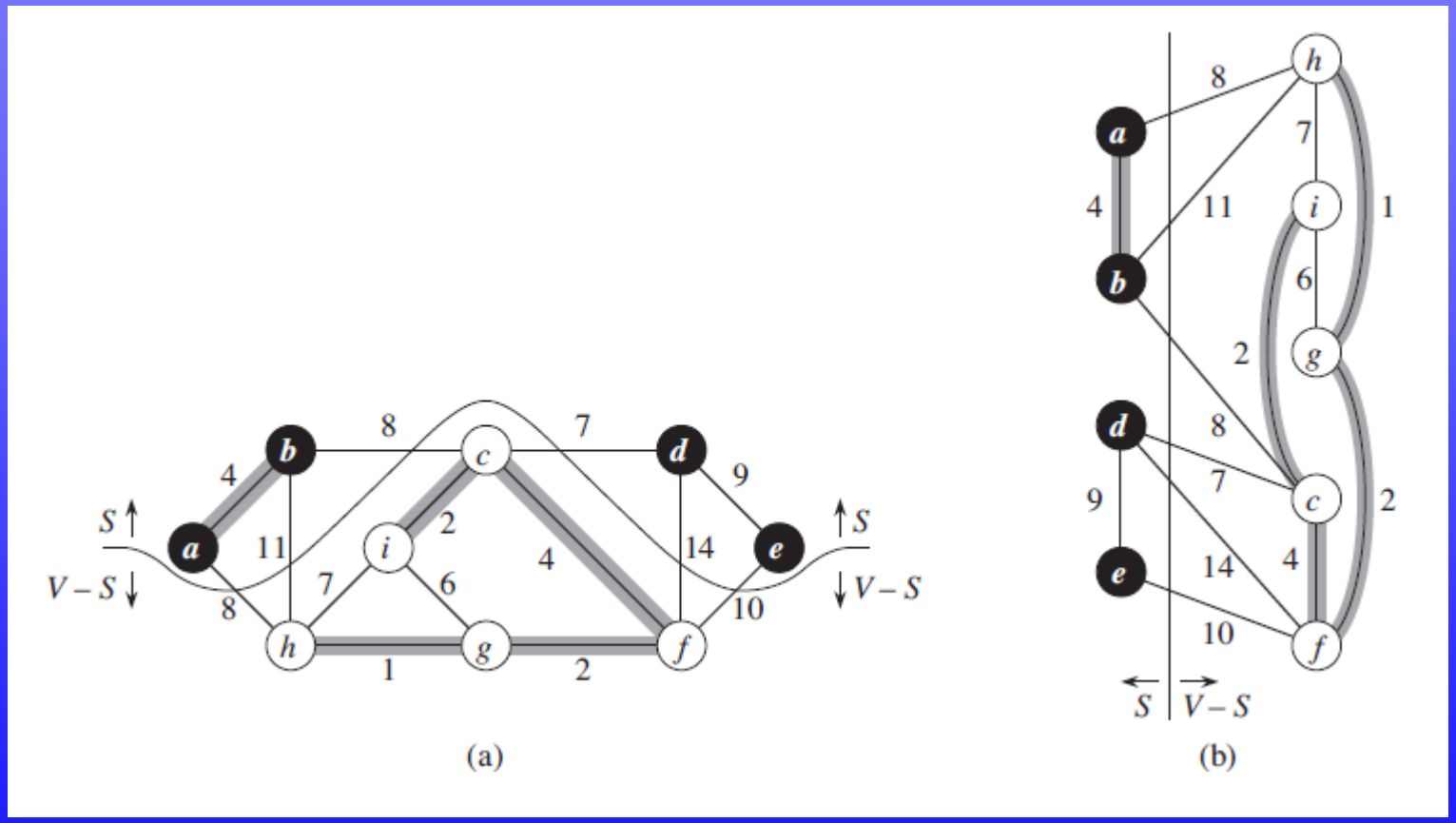
Initialization: After line 1, the set A trivially satisfies the loop invariant.

Maintenance: The loop in lines 2–4 maintains the invariant by adding only safe edges.

Termination: All edges added to A are in a minimum spanning tree, and so the set A returned in line 5 must be a minimum spanning tree.

source: 91.503 textbook Cormen et al.

Growing Minimum Spanning Tree: Concept of CUT



source: 91.503 textbook Cormen et al.

Growing Minimum Spanning Tree: KRUSKAL (Concept)

Theorem 23.1

Let $G = (V, E)$ be a connected, undirected graph with a real-valued weight function w defined on E. Let A be a subset of E that is included in some minimum spanning tree for G, let $(S, V - S)$ be any cut of G that respects A, and let (u, v) be a light edge crossing $(S, V - S)$. Then, edge (u, v) is safe for A.

source: 91.503 textbook Cormen et al.

Growing Minimum Spanning Tree: Kruskal

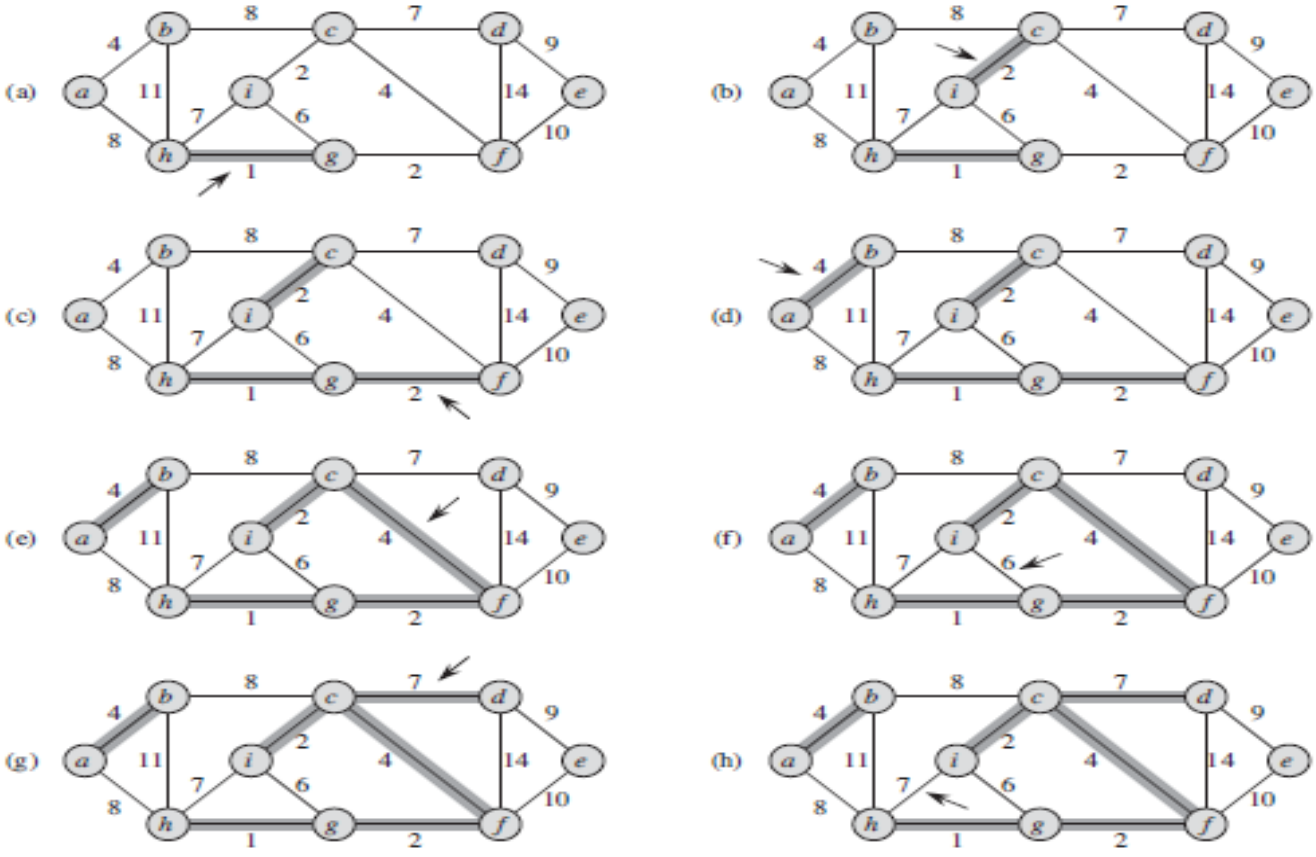


Figure 23.4 The execution of Kruskal's algorithm on the graph from Figure 23.1. Shaded edges belong to the forest A being grown. The algorithm considers each edge in sorted order by weight. An arrow points to the edge under consideration at each step of the algorithm. If the edge joins two distinct trees in the forest, it is added to the forest, thereby merging the two trees.

source: 91.503 textbook Cormen et al.

Growing Minimum Spanning Tree: Kruskal (Contd.)

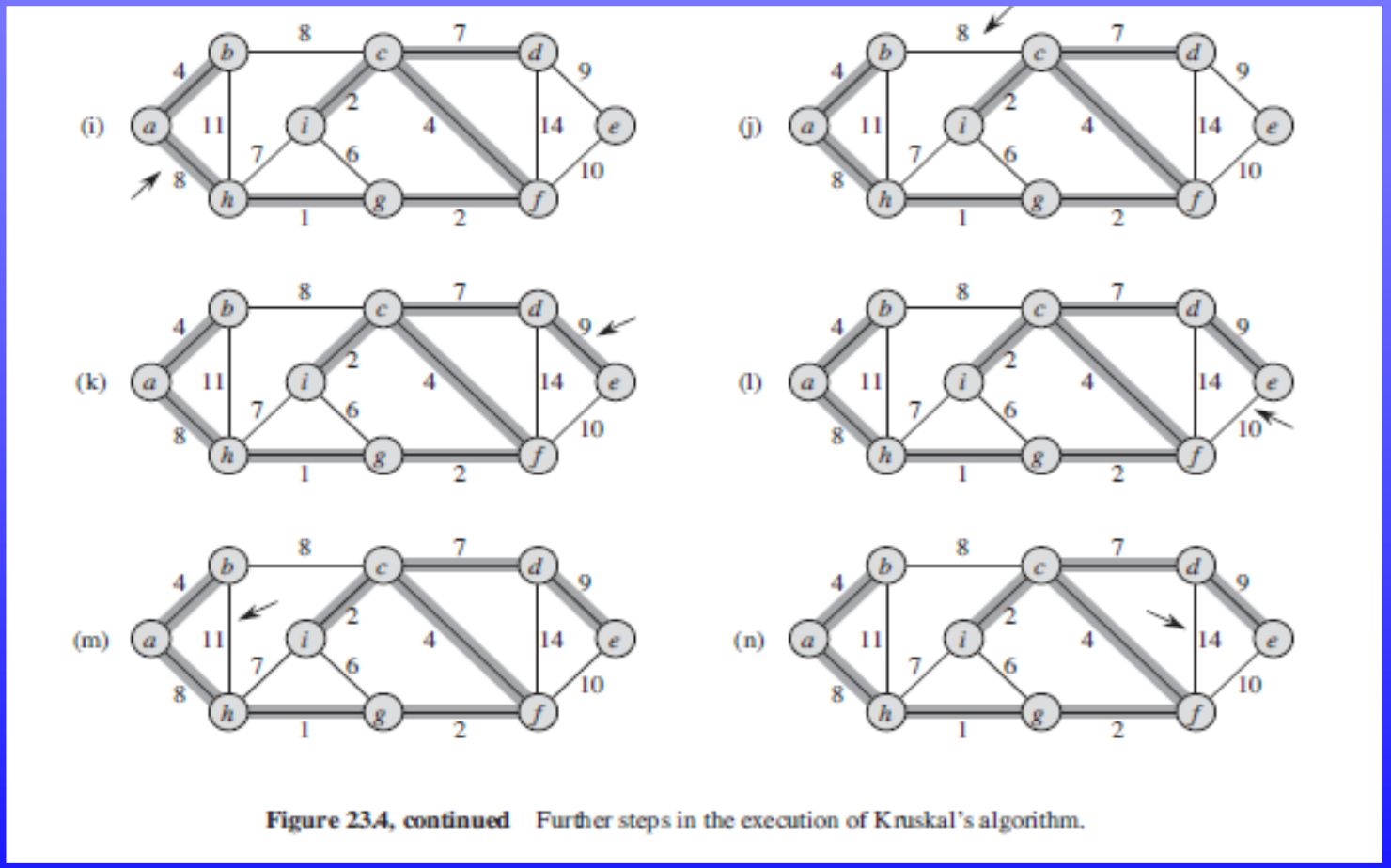


Figure 23.4, continued Further steps in the execution of Kruskal's algorithm.

source: 91.503 textbook Cormen et al.

Minimum Spanning Tree: Greedy Algorithms

Time: O(|E|lg|E|) given fast FIND-SET, UNION

```
MST-KRUSKAL(G, w)
1  A ← ∅
2  for each vertex v ∈ V[G]
3       do MAKE-SET(v)
4  sort the edges of E by nondecreasing weight w
5  for each edge (u, v) ∈ E, in order by nondecreasing weight
6       do if FIND-SET(u) ≠ FIND-SET(v)
7            then A ← A ∪ {(u, v)}
8                 UNION(u, v)
9  return A
```

Invariant: Minimum weight spanning forest

Becomes single tree at end

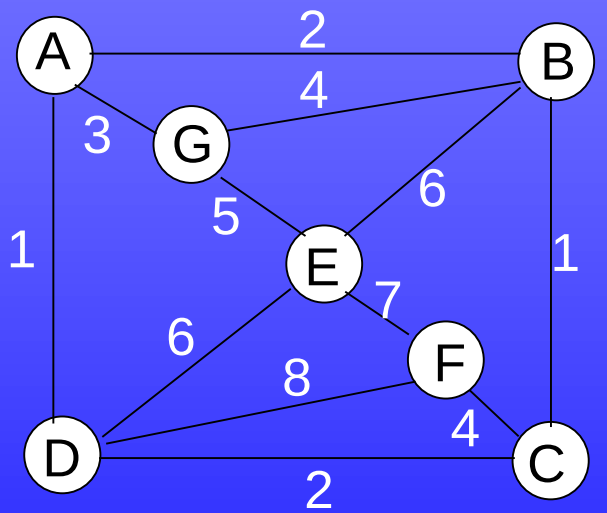
Time: O(|E|lg|V|) = O(|E|lg|E|) slightly faster with fast priority queue

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

Invariant: Minimum weight tree

Spans all vertices at end

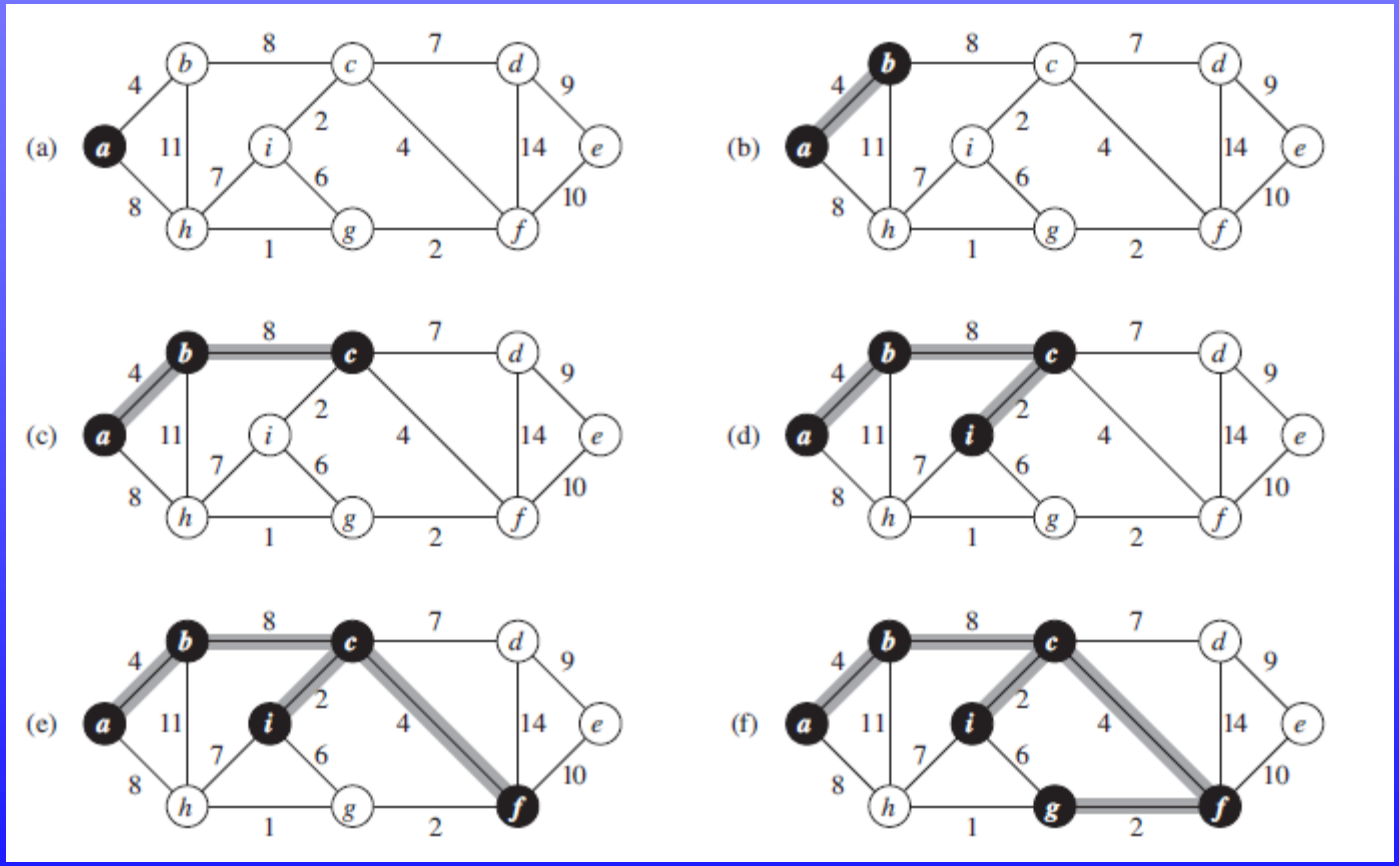
Produces minimum weight tree of edges that includes every vertex.



for Undirected, Connected, Weighted Graph G=(V,E)

source: 91.503 textbook Cormen et al.

Minimum Spanning Tree: Prim's Algorithm



source: 91.503 textbook Cormen et al.

Minimum Spanning Tree: Prim's Algorithm

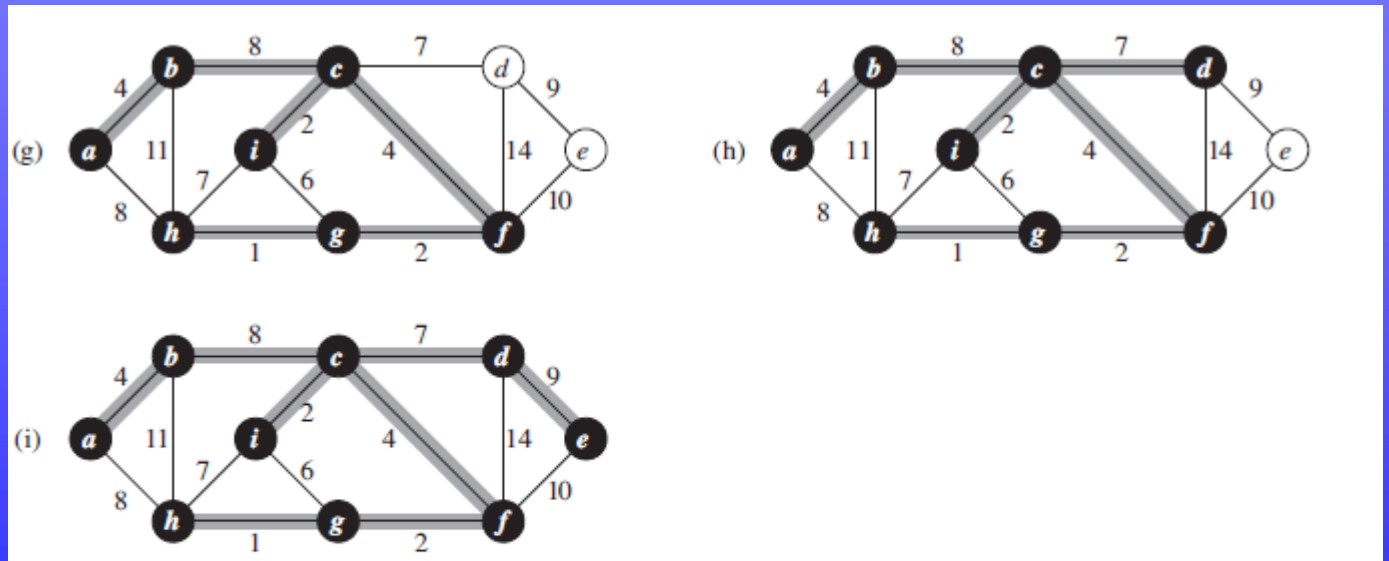
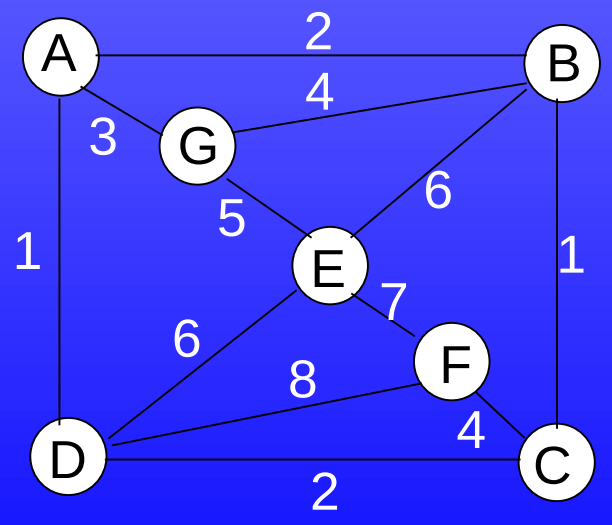


Figure 23.5 The execution of Prim's algorithm on the graph from Figure 23.1. The root vertex is a. Shaded edges are in the tree being grown, and black vertices are in the tree. At each step of the algorithm, the vertices in the tree determine a cut of the graph, and a light edge crossing the cut is added to the tree. In the second step, for example, the algorithm has a choice of adding either edge (b, c) or edge (a, h) to the tree since both are light edges crossing the cut.

source: 91.503 textbook Cormen et al.

Minimum Spanning Trees

- Review problem:
  - **For the undirected, weighted graph below, show 2 different Minimum Spanning Trees. Draw each using one of the 2 graph copies below. Thicken an edge to make it part of a spanning tree. What is the sum of the edge weights for each of your Minimum Spanning Trees?**



Chapter 25
Shortest Paths

Dijkstra & Bellman-Ford

[Source: Cormen et al. textbook except where noted]

BFS as a Basis for Shortest Path Algorithms

*for **unweighted**, **undirected** graph G=(V,E)*

BFS

Time: O(|V| + |E|) adj list
O($|V|^2$) adj matrix

Problem: Given 2 vertices u, v, find the shortest path in G from u to v.

Source/Sink Shortest Path

Solution: BFS starting at u. Stop at v.

Problem: Given a vertex u, find the shortest path in G from u to each vertex.

Single-Source Shortest Paths

Solution: BFS starting at u. Full BFS tree.

Problem: Find the shortest path in G from each vertex u to each vertex v.

All-Pairs Shortest Paths

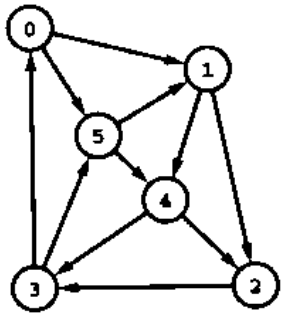
Solution: *For each u*: BFS starting at u; full BFS tree.
Time: O(|V|(|V| + |E|)) adj list
O($|V|^3$) adj matrix

source: based on Sedgewick, Graph Algorithms

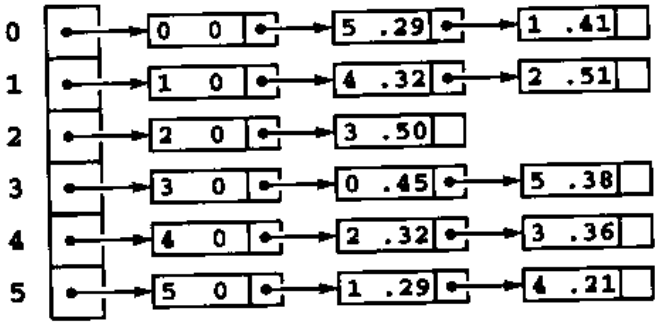
Shortest Path Applications

for weighted, directed graph G=(V,E)

```
0-1  .41
1-2  .51
2-3  .50
4-3  .36
3-5  .38
3-0  .45
0-5  .29
5-4  .21
1-4  .32
4-2  .32
5-1  .29
```



	0	1	2	3	4	5
0	0	.41				.29
1		0	.51		.32	
2			0	.50		
3	.45			0		.38
4			.32	.36	0	
5		.29			.21	0

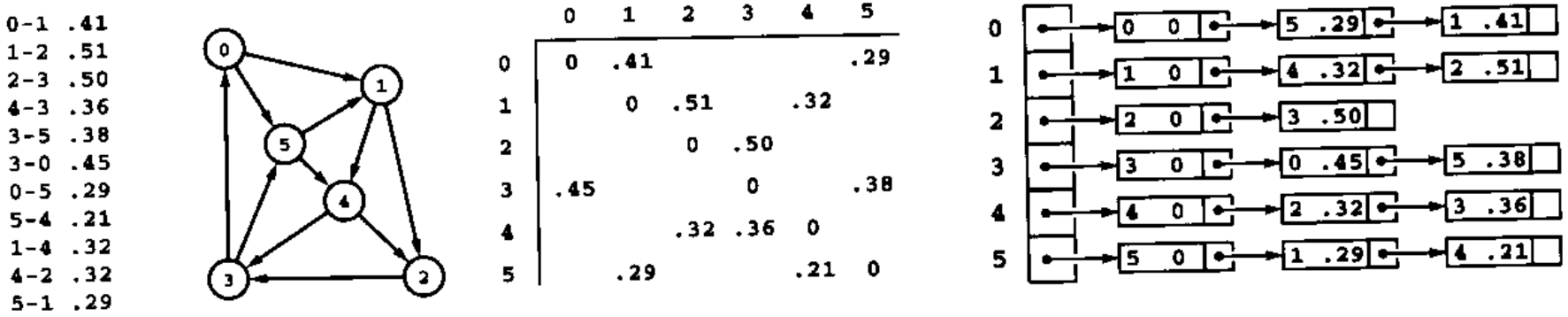


Weight ~ Cost ~ Distance

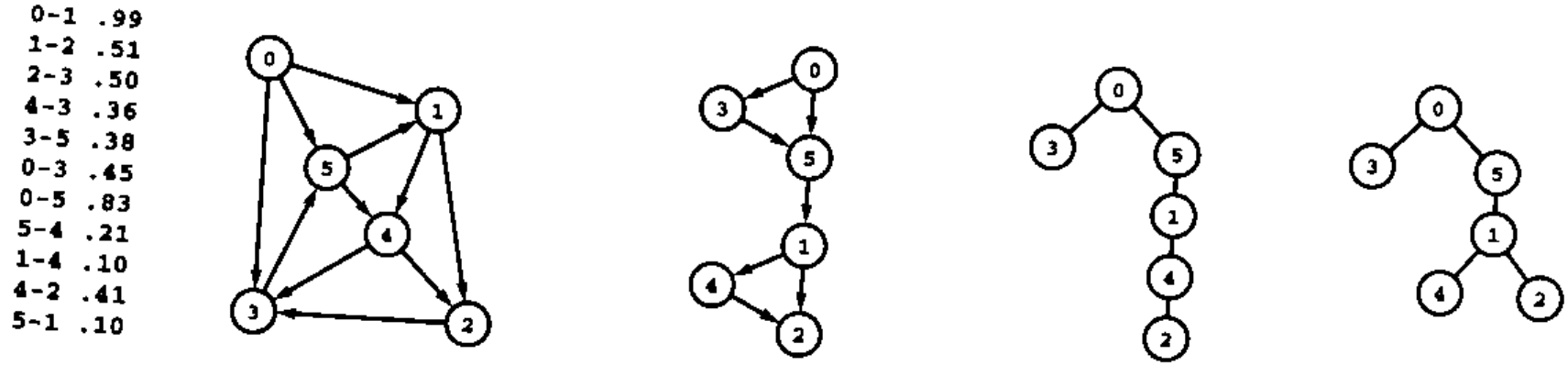
- Road maps
- Airline routes
- Telecommunications network routing
- VLSI design routing

source: based on Sedgewick, Graph Algorithms

Shortest Path Trees



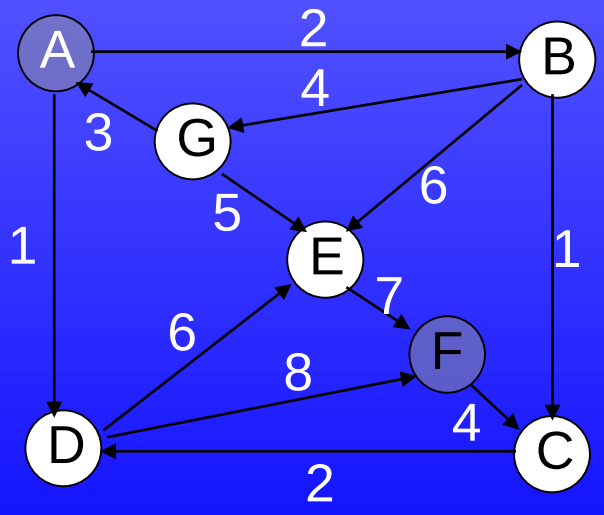
Shortest Path Tree gives shortest path from root to each other vertex



source: Sedgewick, Graph Algorithms

Shortest Path Principles: Relaxation

- "Relax" a constraint to try to improve solution
- "Rubber band" analogy [Sedgewick]
- Relaxation of an Edge (u,v):
  - test if shortest path to v [found so far] can be improved by going through u



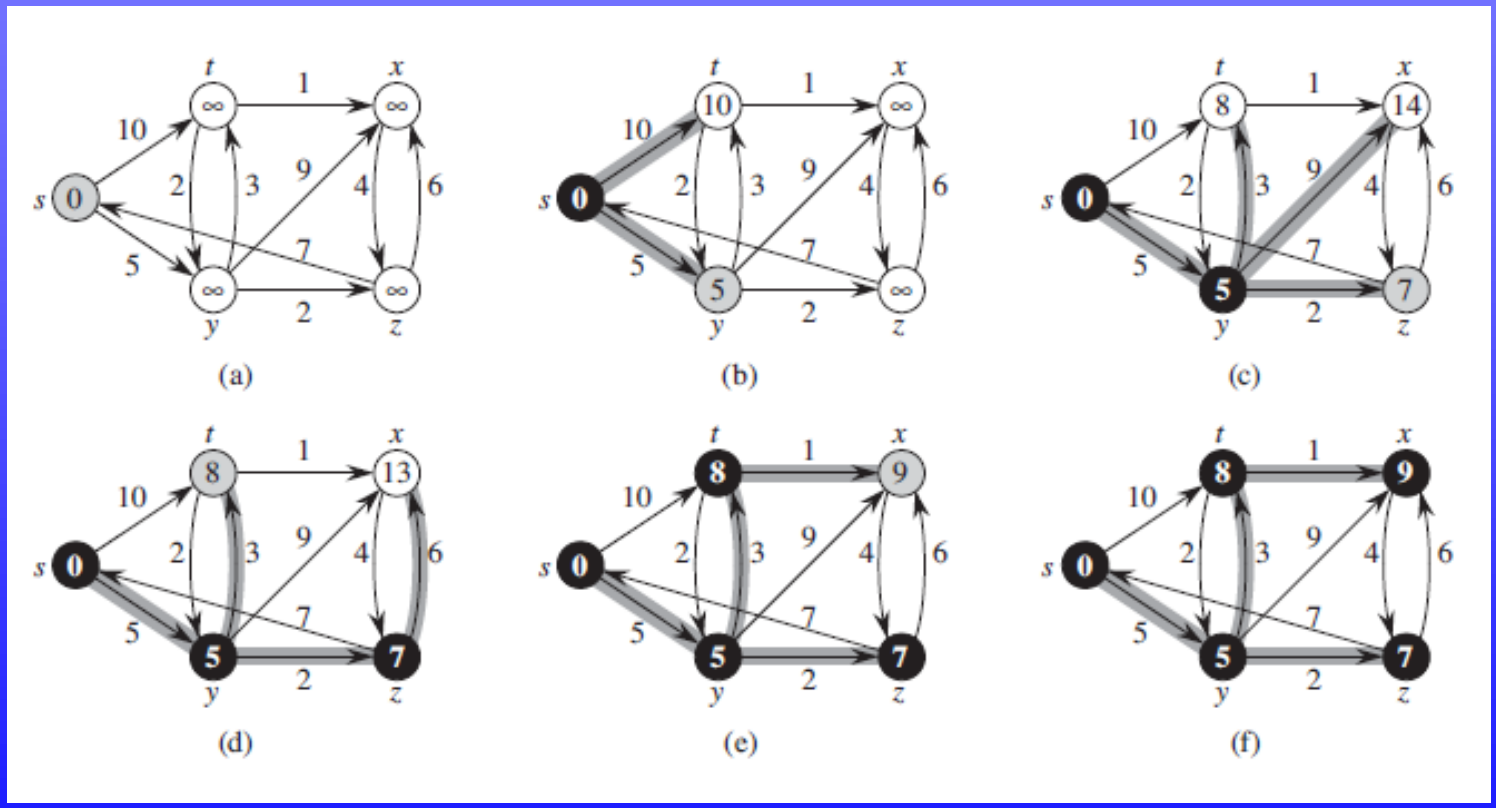
Single Source Shortest Paths
Dijkstra's Algorithm

for (nonnegative) weighted, directed graph G=(V,E)

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

source: 91.503 textbook Cormen et al.

Shortest Path: Dijkstra's Algorithm



source: 91.503 textbook Cormen et al.

Single Source Shortest Paths
Dijkstra's Algorithm

for (nonnegative) weighted, directed graph G=(V,E)

- See separate ShortestPath 91.404 slide show

Triangle inequality (Lemma 24.10)
For any edge $(u, v) \in E$, we have $\delta(s, v) \leq \delta(s, u) + w(u, v)$.

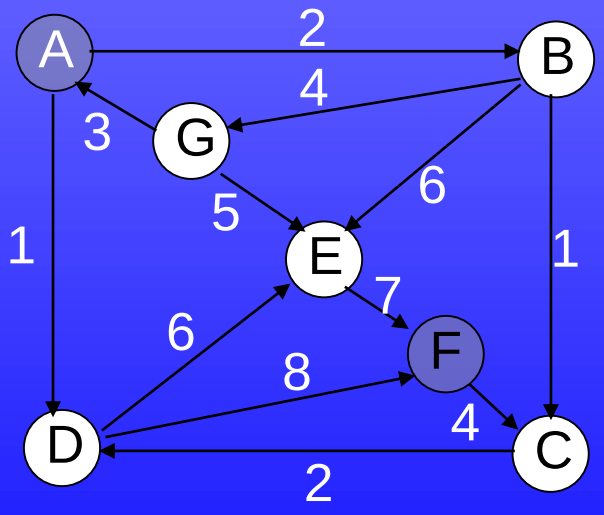
Upper-bound property (Lemma 24.11)
We always have $d[v] \geq \delta(s, v)$ for all vertices $v \in V$, and once $d[v]$ achieves the value $\delta(s, v)$, it never changes.

No-path property (Corollary 24.12)
If there is no path from s to v, then we always have $d[v] = \delta(s, v) = \infty$.

Convergence property (Lemma 24.14)
If $s \rightsquigarrow u \rightarrow v$ is a shortest path in G for some $u, v \in V$, and if $d[u] = \delta(s, u)$ at any time prior to relaxing edge (u, v), then $d[v] = \delta(s, v)$ at all times afterward.

Path-relaxation property (Lemma 24.15)
If $p = \langle v_0, v_1, \ldots, v_k \rangle$ is a shortest path from $s = v_0$ to v_k, and the edges of p are relaxed in the order $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, then $d[v_k] = \delta(s, v_k)$. This property holds regardless of any other relaxation steps that occur, even if they are intermixed with relaxations of the edges of p.

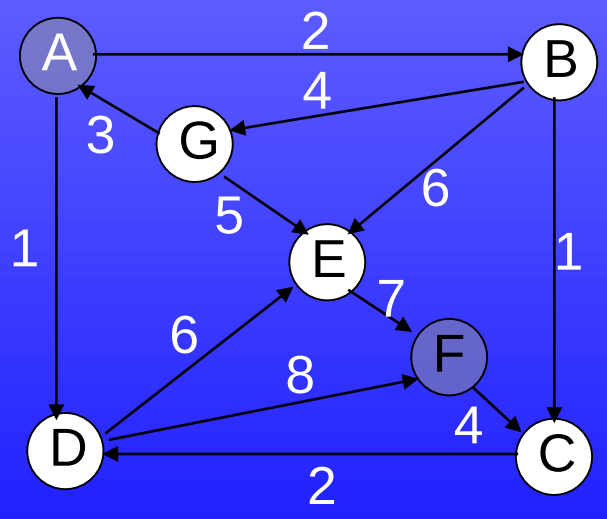
Predecessor-subgraph property (Lemma 24.17)
Once $d[v] = \delta(s, v)$ for all $v \in V$, the predecessor subgraph is a shortest-paths tree rooted at s.



source: 91.503 textbook Cormen et al.

Single Source Shortest Paths
Dijkstra's Algorithm

- Review problem:
  - For the directed, weighted graph below, find the shortest path that begins at vertex A and ends at vertex F. List the vertices in the order that they appear on that path. What is the sum of the edge weights of that path?



Why can't Dijkstra's algorithm handle negative-weight edges?

Single Source Shortest Paths
Dijkstra's Algorithm

for (nonnegative) weighted, directed graph G=(V,E)

Table 21.2 Cost of implementations of Dijkstra's algorithm

This table summarizes the cost (worst-case running time) of various implementations of Dijkstra's algorithm. With appropriate priority-queue implementations, the algorithm runs in linear time (time proportional to V^2 for dense networks, E for sparse networks) except for networks that are extremely sparse.

algorithm	worst-case cost	comment
classical	V^2	optimal for dense graphs
PFS, full heap	$E \lg V$	simplest ADT code
PFS, fringe heap	$E \lg V$	conservative upper bound
PFS, d-heap	$E \lg_d V$	linear unless extremely sparse

PFS – Priority First Search

source: Sedgewick, Graph Algorithms

Shortest Path Algorithms

Table 21.4 Costs of shortest-paths algorithms

This table summarizes the cost (worst-case running time) of various shortest-paths algorithms considered in this chapter. The worst-case bounds marked as conservative may not be useful in predicting performance on real networks, particularly the Bellman–Ford algorithm, which typically runs in linear time.

weight constraint	algorithm	cost	comment
single-source			
nonnegative	Dijkstra	V^2	optimal (dense networks)
nonnegative	Dijkstra (PFS)	$E \lg V$	conservative bound
acyclic	source queue	E	optimal
no negative cycles	Bellman–Ford	VE	room for improvement?
none	open	?	NP-hard
all-pairs			
nonnegative	Floyd	V^3	same for all networks
nonnegative	Dijkstra (PFS)	$VE \lg V$	conservative bound
acyclic	DFS	VE	same for all networks
no negative cycles	Floyd	V^3	same for all networks
no negative cycles	Johnson	$VE \lg V$	conservative bound
none	open	?	NP-hard

source: Sedgewick, Graph Algorithms

Shortest Path: Bellman-Ford Algorithm

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

source: 91.503 textbook Cormen et al.

Shortest Path: Bellman-Ford Algorithm

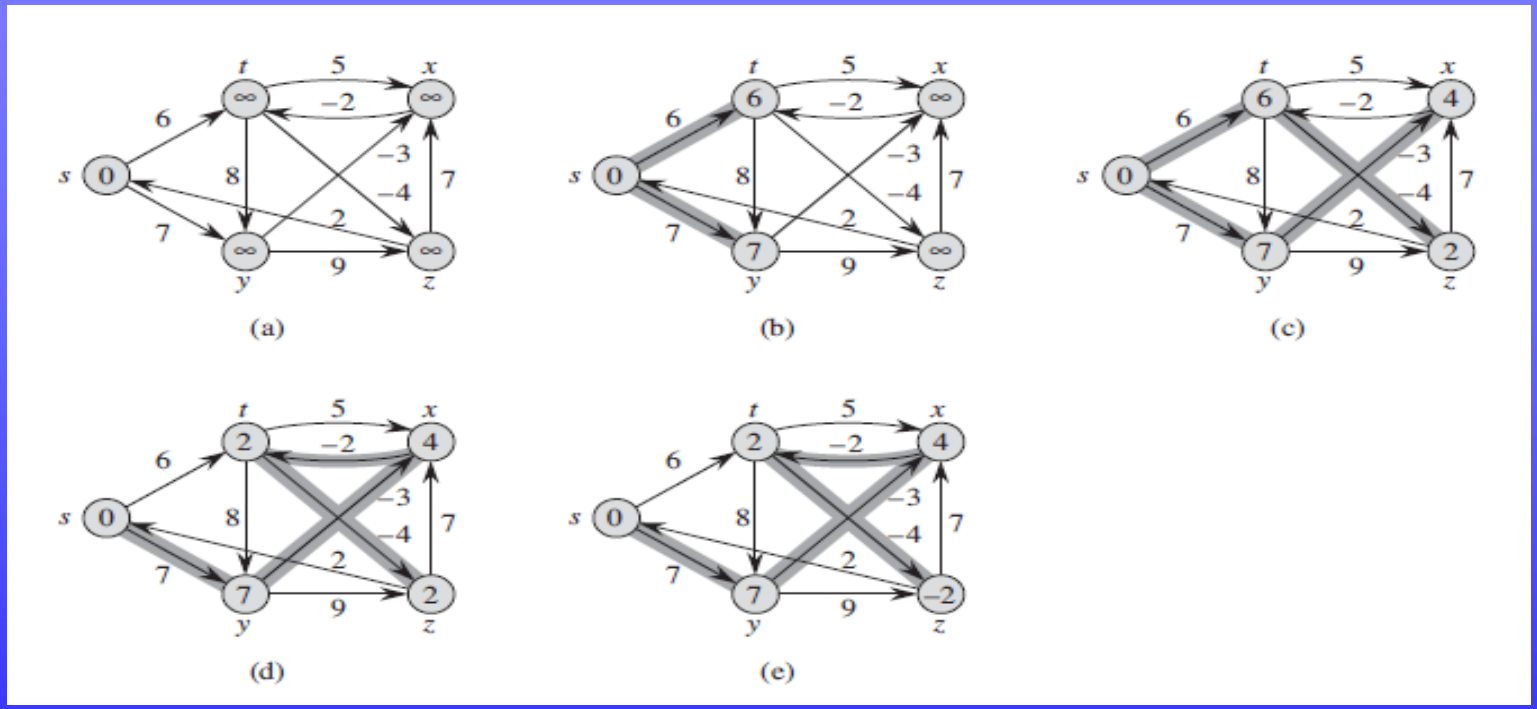


Figure 24.4 The execution of the Bellman-Ford algorithm. The source is vertex s. The d values appear within the vertices, and shaded edges indicate predecessor values: if edge (u, v) is shaded, then $v.\pi = u$. In this particular example, each pass relaxes the edges in the order $(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)$. **(a)** The situation just before the first pass over the edges. **(b)–(e)** The situation after each successive pass over the edges. The d and π values in part (e) are the final values. The Bellman-Ford algorithm returns TRUE in this example.

source: 91.503 textbook Cormen et al.

Shortest Path: Bellman-Ford Algorithm

Figure 24.4 The execution of the Bellman-Ford algorithm. The source is vertex s. The d values appear within the vertices, and shaded edges indicate predecessor values: if edge .u; / is shaded, then : D u. In this particular example, each pass relaxes the edges in the order .t; x/; .t; y/; .t; ’/; .x; t /; .y; x/; .y; ’/; .’; x/; .’; s/; .s; t /; .s; y/. **(a)** The situation just before the first pass over the edges. **(b)–(e)** The situation after each successive pass over the edges. The d and values in part (e) are the final values. The Bellman-Ford algorithm returns TRUE in this example.

source: 91.503 textbook Cormen et al.