Searching on Graphs

December 13, 2023

Graph-searching Algorithms

- Searching a graph:
  - Systematically follow the edges of a graph to visit the vertices of the graph.
- Used to discover the structure of a graph.
- Standard graph-searching algorithms.
  - Breadth-first Search (BFS).
  - Depth-first Search (DFS).

Breadth-First Search(BFS) – Basic Idea

Breadth First Search visits vertices in increasing distance from the source.
Given a graph with N vertices and a selected vertex s:

for (i = 1; there are unvisited vertices ; i++)
    Visit all unvisited vertices at distance i

Note that i is the length of the shortest path between s and currently processed vertices

Queue-based implementation

BFS – Algorithm

1. Store source vertex S in a queue and mark as processed
2. while queue is not empty   Read vertex v from the queue

   for all neighbors w:

   If w is not processed

   Mark w as processed

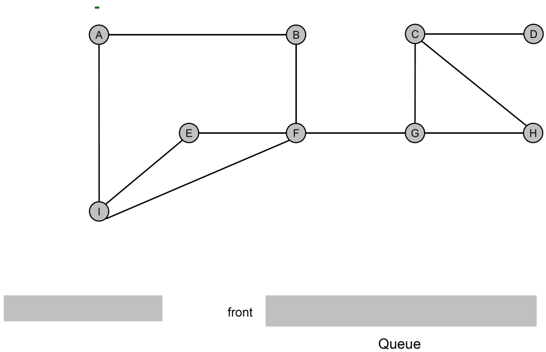
   Enqueue w in the queue

   Record the parent of w to be v

   (necessary only if we need the shortest path tree)

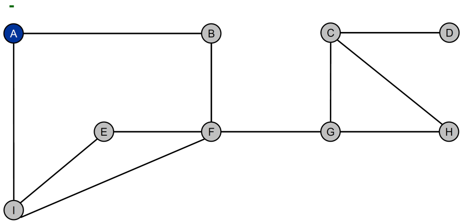
BFS-example

1. Store source vertex S in a queue and mark as processed
2. while queue is not empty
   Read vertex v from the queue
   for all neighbors w:
   If w is not processed
   Mark w as processed
   Enqueue w in the queue
   Record the parent of w to be v
   (necessary only if we need the shortest path tree)



BFS-example

1. Store source vertex S in a queue and mark as processed
2. while queue is not empty
   Read vertex v from the queue
   for all neighbors w:
     If w is not processed
       Mark w as processed
       Enqueue w in the queue
       Record the parent of w to be v
       (necessary only if we need the shortest path tree)



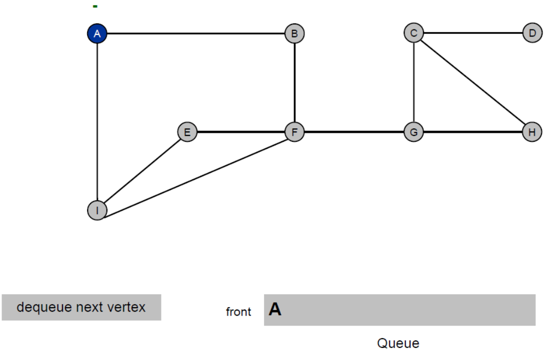
enqueue source node

front **A**

Queue

BFS-example

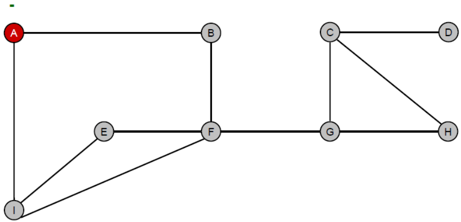
1. Store source vertex S in a queue and mark as processed
2. while queue is not empty
    Read vertex v from the queue
    for all neighbors w:
        If w is not processed
            Mark w as processed
            Enqueue w in the queue
            Record the parent of w to be v
            (necessary only if we need the shortest path tree)



BFS-example

1. Store source vertex *S* in a queue and mark as processed
2. while queue is not empty

   Read vertex *v* from the queue
   for all neighbors *w*:
   If *w* is not processed
   Mark *w* as processed
   Enqueue *w* in the queue
   Record the parent of *w* to be *v*
   (necessary only if we need the shortest path tree)



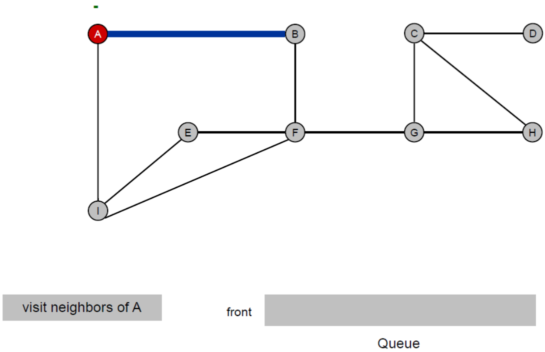
visit neighbors of A

front

Queue

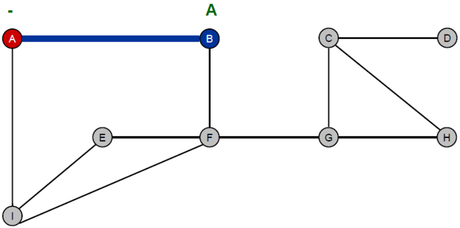
BFS-example

1. Store source vertex S in a queue and mark as processed
2. while queue is not empty
   Read vertex v from the queue
   for all neighbors w:
      If w is not processed
         Mark w as processed
         Enqueue w in the queue
         Record the parent of w to be v
         (necessary only if we need the shortest path tree)



BFS-example

1. Store source vertex S in a queue and mark as processed
2. while queue is not empty
   - Read vertex v from the queue
   - for all neighbors w:
     - If w is not processed
       - Mark w as processed
       - Enqueue w in the queue
       - Record the parent of w to be v
       - (necessary only if we need the shortest path tree)



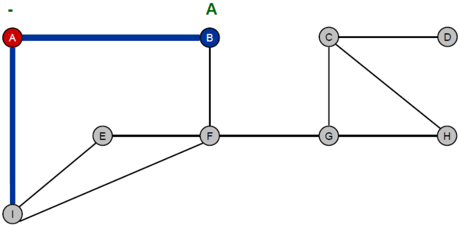
B discovered

front **B**

Queue

BFS-example

1. Store source vertex S in a queue and mark as processed
2. while queue is not empty
   Read vertex v from the queue
   for all neighbors w:
   If w is not processed
   Mark w as processed
   Enqueue w in the queue
   Record the parent of w to be v
   (necessary only if we need the shortest path tree)



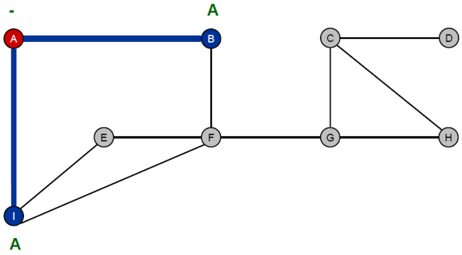
visit neighbors of A

front **B**

Queue

BFS-example

1. Store source vertex S in a queue and mark as processed
2. while queue is not empty
   Read vertex v from the queue
   for all neighbors w:
   If w is not processed
   Mark w as processed
   Enqueue w in the queue
   Record the parent of w to be v
   (necessary only if we need the shortest path tree)



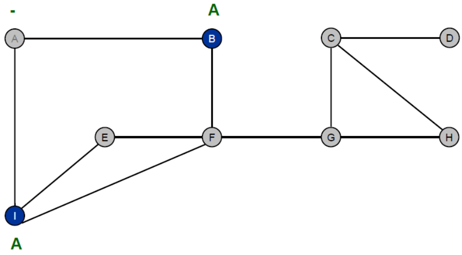
I discovered

front **B I**

Queue

BFS-example

1. Store source vertex S in a queue and mark as processed
2. while queue is not empty
   Read vertex v from the queue
   for all neighbors w:
   If w is not processed
   Mark w as processed
   Enqueue w in the queue
   Record the parent of w to be v
   (necessary only if we need the shortest path tree)



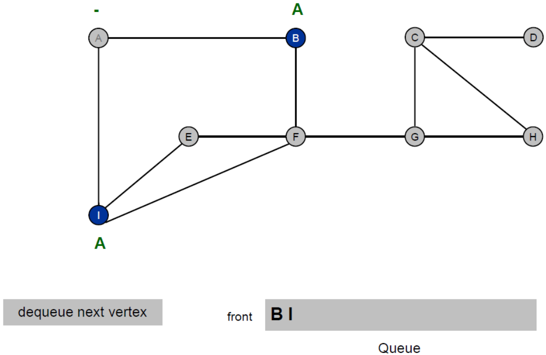
finished with A

front **B I**

Queue

BFS-example

1. Store source vertex S in a queue and mark as processed
2. while queue is not empty
   Read vertex v from the queue
   for all neighbors w:
   If w is not processed
   Mark w as processed
   Enqueue w in the queue
   Record the parent of w to be v
   (necessary only if we need the shortest path tree)



BFS Code

@PS

BFS – Complexity

- Step 1 : read a node from the queue $O(|V|)$ times.
- Step 2 : examine all neighbors, i.e. we examine all edges of the currently read node.
- Hence the complexity of BFS is $O(|V| + |E|)$
  - Undirected graph: $2 \times |E|$ edges to examine

Applications of Breadth-First Search *

- **Shortest Path and Minimum Spanning Tree for unweighted graph:** Details coming soon.
- **Peer to Peer Networks:** In Peer to Peer Networks, BFS is used to find the neighboring nodes in the network.
- **Crawlers in Search Engines:** Crawlers for search engines use BFS to create the index. Starting from a web page they follow the links from the source and keep doing the same for the new pages discovered.
- **Social Networking Websites:** Similar to peer to peer networks, it is possible to find people that are 'k' steps away from a particular person using BFS.

* http://www.algorithmforum.com/2017/09/breadth-first-search-bfs-and-its.html

Applications of Breadth-First Search

- **GPS Navigation systems:** BFS can be used to find all locations that are neighboring a particular place.
- **Broadcasting in Network:** A broadcast package follows BFS to reach all nodes in the network.
- **Cycle detection in undirected graph, To test if a graph is Bipartite, Path Finding** Breadth First or Depth First Search can be used to find a solution to these use cases.

Depth-First Search

- Depth First Search is another approach to visit all nodes of a graph in a systematic manner
- Works with directed and undirected graphs
- Works with weighted and unweighted graphs

Depth-First Search

Procedure dfs(s)

mark all vertices in the graph as not reached

invoke scan(s)

Procedure scan(s)

mark and visit s

for each neighbor w of s

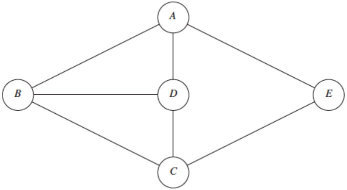
if the neighbor is not reached invoke scan(w)

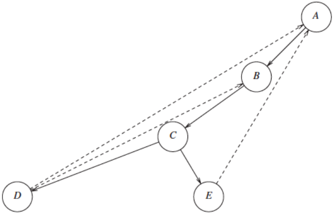
Depth-First Search with Stack

```
Initialization:
        mark all vertices as unvisited,
        visit(s)
         while the stack is not empty:
            pop (v,w)
            if w is not visited
                visit(w)
Procedure visit(v)
        mark v as visited
        for each edge (v,w)
            push (v,w) in the stack
```

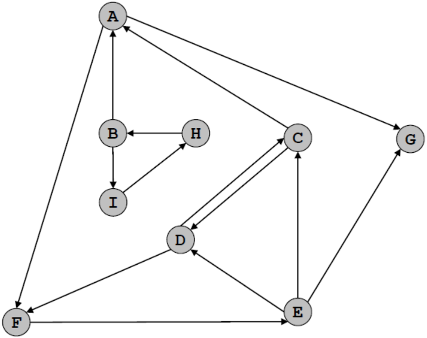
DFS Tree

- The root of the tree is the first vertex visited.
- Each edge in the graph is represented in the tree.
- An edge in the tree shows that child is visited through the parent.
- A back edge is not part of the tree but represents the vertex was checked but found out to be visited.





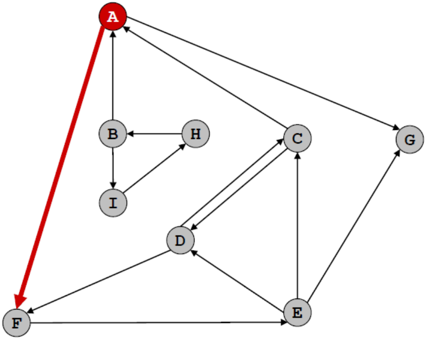
DFS Example



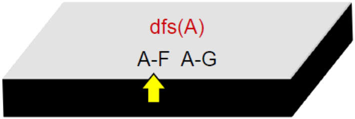
Adjacency Lists

A: F G
B: A I
C: A D
D: C F
E: C D G
F: E
G:
H: B
I: H

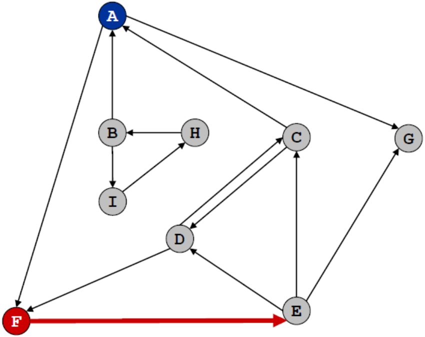
DFS Example



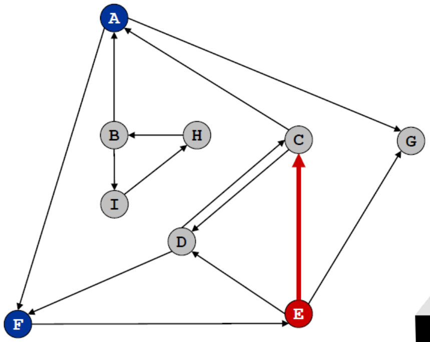
Function call stack:

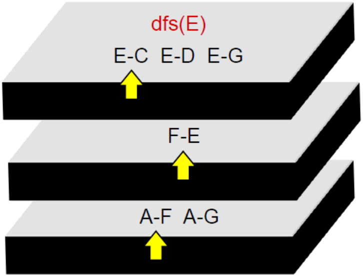


DFS Example



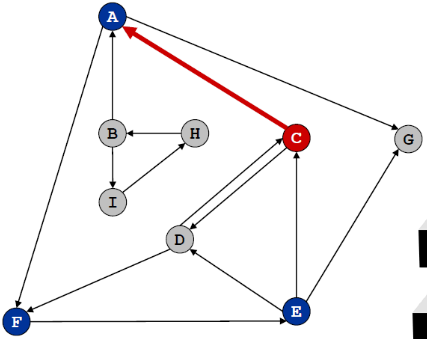
Function call stack:



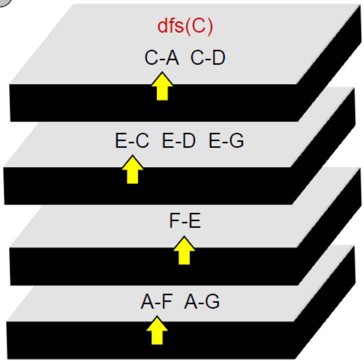


Function call stack:

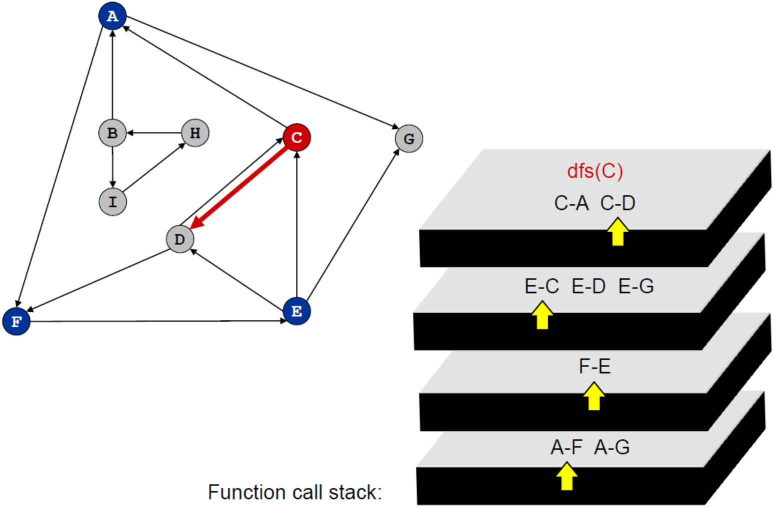
DFS Example



Function call stack:

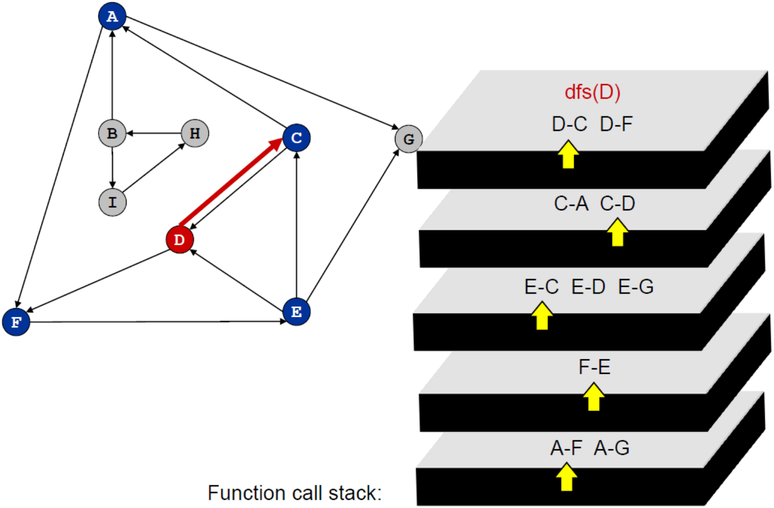


DFS Example

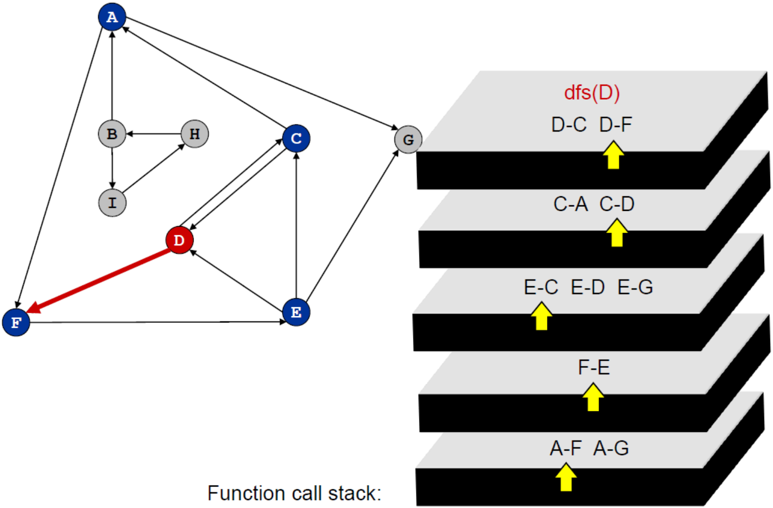


Function call stack:

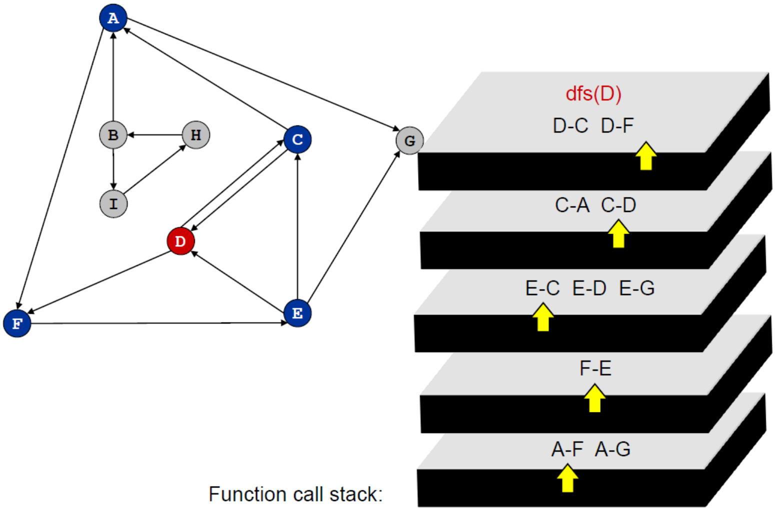
DFS Example


A
B
H
C
G
I
D
F
E
dfs(D)
D-C D-F
C-A C-D
E-C E-D E-G
F-E
A-F A-G
Function call stack:

DFS Example

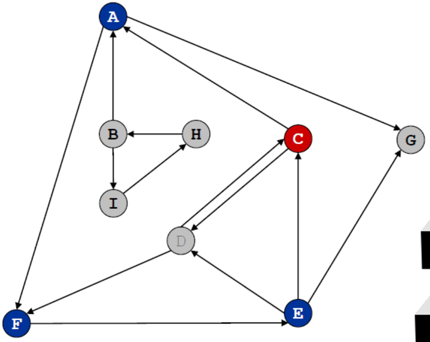

A
B
H
C
G
I
D
F
E
dfs(D)
D-C D-F
C-A C-D
E-C E-D E-G
F-E
A-F A-G
Function call stack:

DFS Example

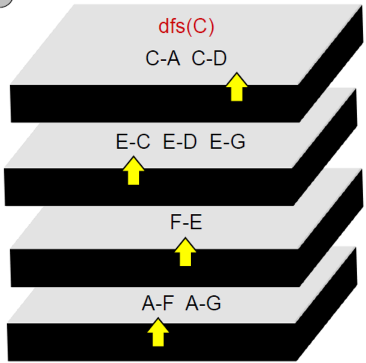


Function call stack:

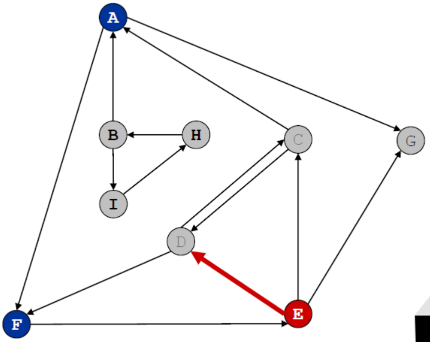
DFS Example



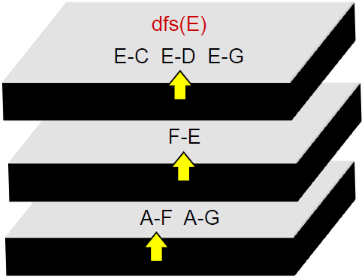
Function call stack:



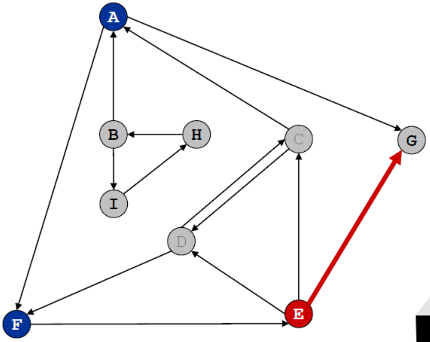
DFS Example



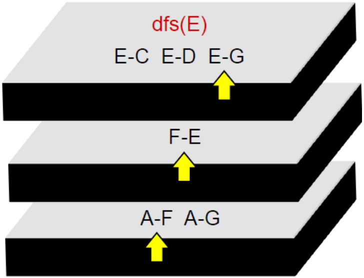
Function call stack:



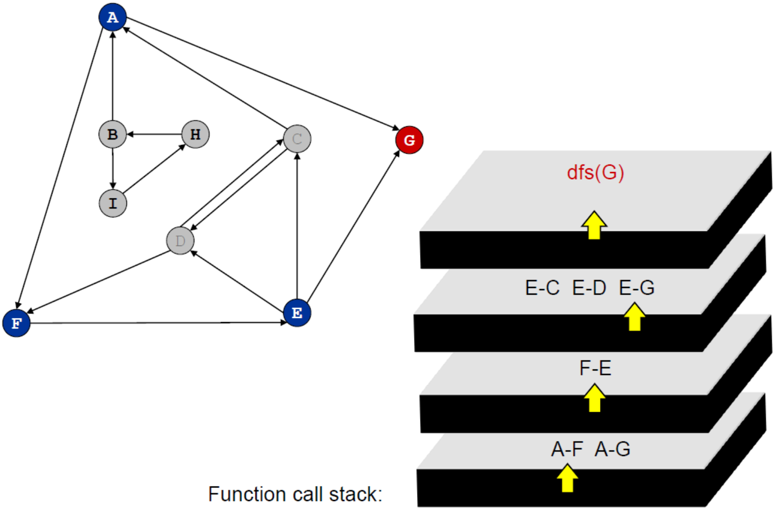
DFS Example



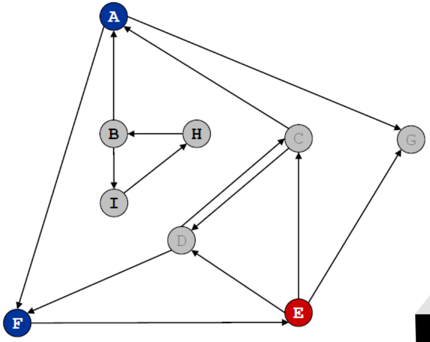
Function call stack:



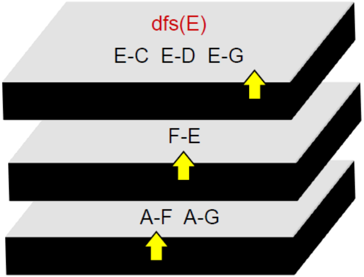
DFS Example



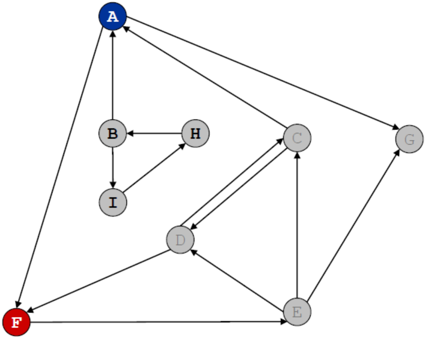
DFS Example



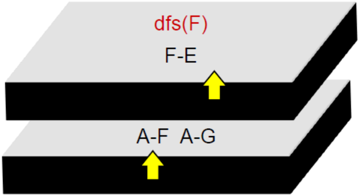
Function call stack:



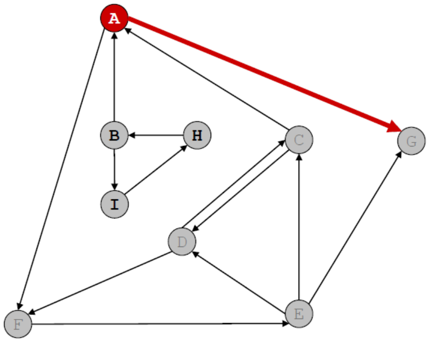
DFS Example



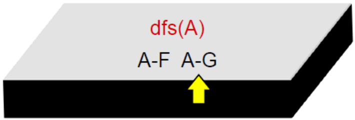
Function call stack:



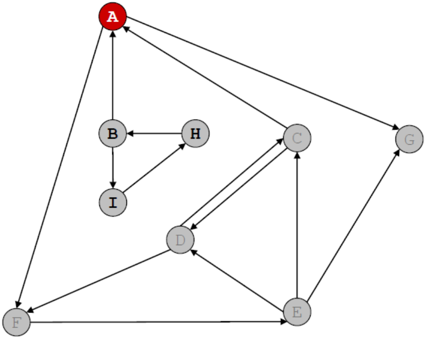
DFS Example



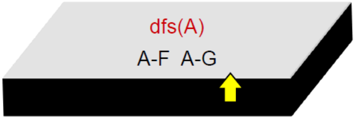
Function call stack:



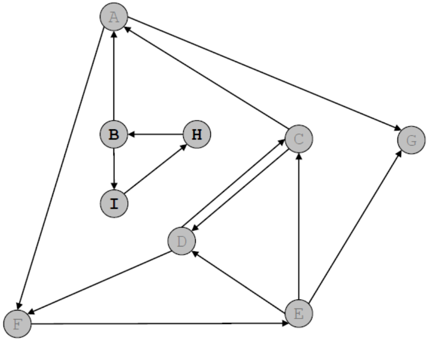
DFS Example



Function call stack:



DFS Example



Nodes reachable from A: A, C, D, E, F, G

Example

Adjacency lists

1: 2, 3, 4

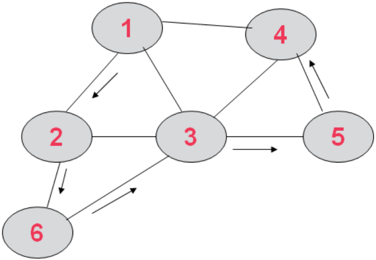
2: 6, 3, 1

3: 1, 2, 6, 5, 4

4: 1, 3, 5

5: 3, 4

6: 2, 3



Depth first traversal: 1, 2, 6, 3, 5, 4

The particular order is dependent on the order of nodes in the adjacency lists

DFS Code

@PS

Comparison of BFS and DFS

BFS

- BFS is a vertex-based alg.
- BFS uses queue data structure
- BFS is more suitable for searching vertices which are closer to the given source.
- BFS considers all neighbors first and therefore not suitable for decision making trees used in games or puzzles.
- The Time complexity of BFS is $O(V+E)$ when Adjacency List is used and $O(V^2)$ when Adjacency Matrix is used

DFS

- DFS is a edge-based alg.
- DFS uses stack data structure
- DFS is more suitable when there are solutions away from source.
- DFS is more suitable for game or puzzle problems. We make a decision, then explore all paths through this decision.
- The Time complexity of DFS is also $O(V+E)$ when Adjacency List is used and $O(V^2)$ when Adjacency Matrix is used

Applications of Depth-First Search

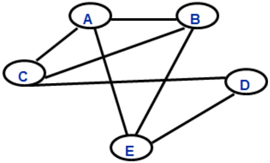
- Graph Connectivity
  - Connectivity
  - Biconnectivity
  - Articulation Points and Bridges
  - Connectivity in Directed Graphs

Connectivity

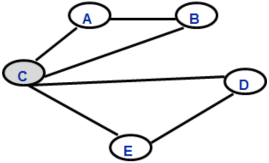
- Definition:
  - An undirected graph is said to be connected if for any pair of nodes of the graph, the two nodes are reachable from one another (i.e. there is a path between them).
  - If starting from any vertex we can visit all other vertices, then the graph is connected

Biconnectivity

- A graph is biconnected, if there are no vertices whose removal will disconnect the graph.



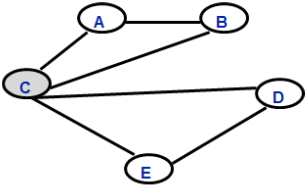
biconnected



not biconnected

Articulation Points

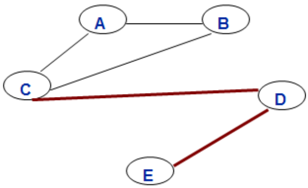
- Definition: A vertex whose removal makes the graph disconnected is called an articulation point or cut-vertex



C is an articulation point
We can compute articulation points using depth-first search and a special numbering of the vertices in the order of their visiting.

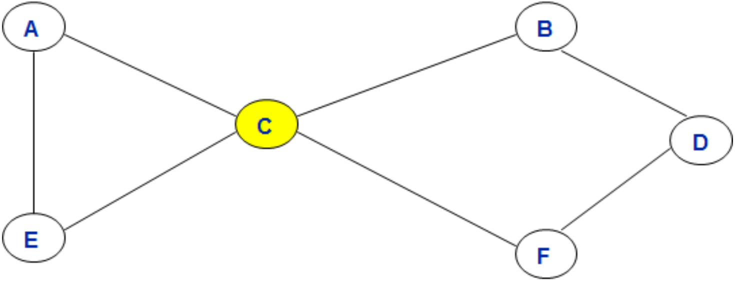
Bridges

- Definition: An edge in a graph is called a bridge, if its removal disconnects the graph.
- Any edge in a graph, that does not lie on a cycle, is a bridge. Obviously, a bridge has at least one articulation point at its end, however an articulation point is not necessarily linked in a bridge.



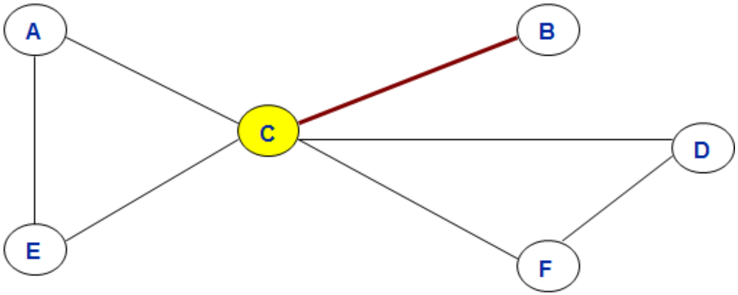
(C,D) and (E,D) are bridges

Example 1



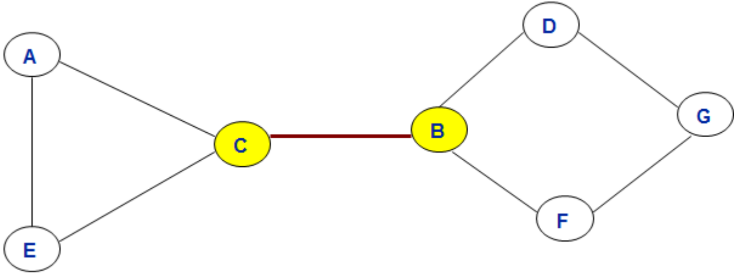
C is an articulation point, there are no bridges

Example 2



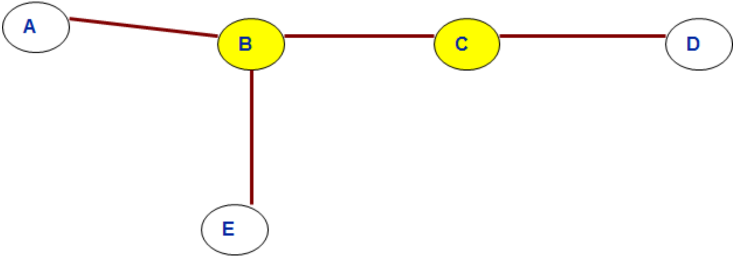
C is an articulation point, (C,B) is a bridge

Example 3



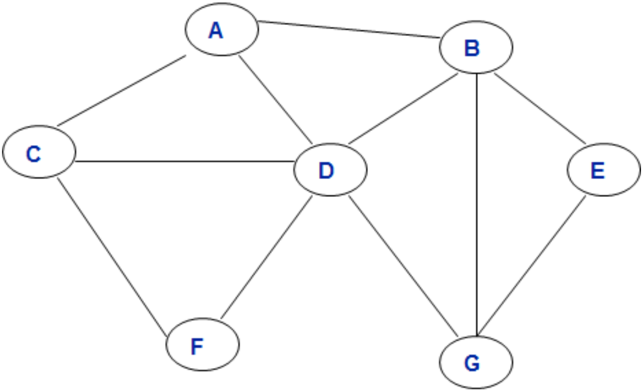
B and C are articulation points, (B,C) is a bridge

Example 4



B and C are articulation points. All edges are bridges

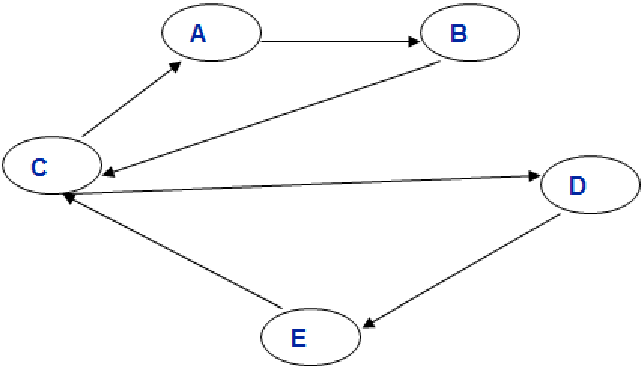
Example 5



Biconnected graph - no articulation points and no bridges

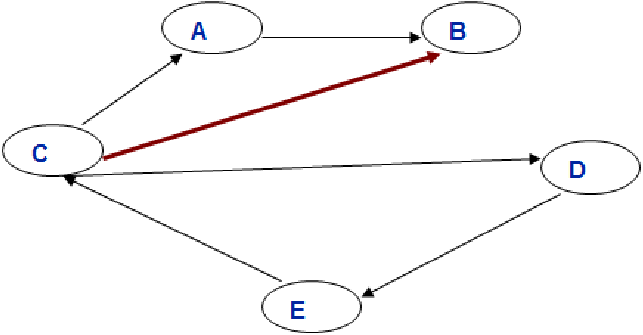
Connectivity in Directed Graphs (I)

- Definition: A directed graph is said to be strongly connected if for any pair of nodes there is a path from each one to the other



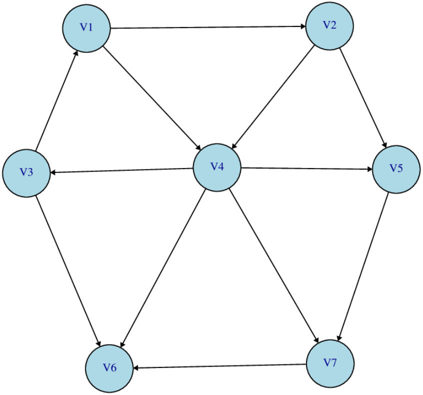
Connectivity in Directed Graphs (II)

- Definition: A directed graph is said to be unilaterally connected if for any pair of nodes at least one of the nodes is reachable from the other
- Each strongly connected graph is also unilaterally connected.



Shortest Paths

Unweighted Directed Graphs



What is the shortest path from V3 to V5?

Problem Data

- The problem: Given a source vertex s, find the shortest path to all other vertices.
- Data structures needed:
  - Graph representation:
    - Adjacency lists / adjacency matrix
  - Distance table:
    - Distances from source vertex
    - Paths from source vertex

Problem Data

Adjacency lists :

V1: V2, V4

V2: V4, V5

V3: V1, V6

V4: V3, V5, V6, V7

V5: V7

V6: -

V7: V6

Let s = V3, stored in a queue

Initialized distance table:

Vertex	Distance	Parent
V1	-1	-1
V2	-1	-1
V3	0	0
V4	-1	-1
V5	-1	-1
V6	-1	-1

Breadth-first search in graphs

- Take a vertex and examine all adjacent vertices.
- Do the same with each of the adjacent vertices .

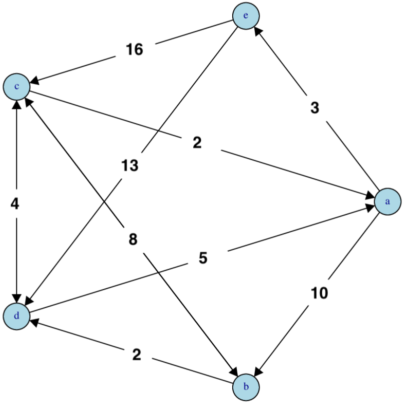
Algorithm

1. Store s in a queue, and initialize distance = 0 in the Distance Table
2. While there are vertices in the queue:
   Read a vertex v from the queue
   For all adjacent vertices w :
   If distance = -1 (not computed)
   Distance = (distance to v) + 1
   Parent = v
   Append w to the queue

Complexity

- Matrix representation: $O(|V|^2)$
- Adjacency lists - $O(|E| + |V|)$
- We examine all edges ($O(|E|)$), and we store in the queue each vertex only once ($O(|V|)$).

Weighted Directed Graphs



- What is the shortest path from e to a?
- Length of a path is the sum of the weights of its edges.

Application Examples

- Internet packet routing
- Flight reservations
- Driving directions

Dijkstra's Algorithm

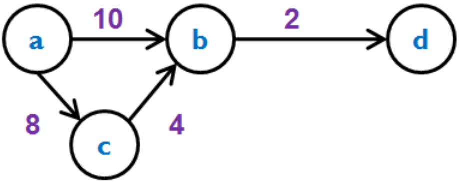
- This algorithm finds the shortest path from a source vertex to all other vertices in a weighted directed graph without negative edge weights.

Dijkstra's Algorithm

- For a Graph $\mathcal{G}$
  - Vertices $\mathcal{V} = v_1, \ldots, v_n$
  - And edge weights w_{ij}, for edge connecting v_i to v_j.
  - Let the source be v_1.
- Initialize a Set $S = \emptyset$.
  - Keep track of the vertices for which we have already computed their shortest path from the source.
- Initialize an array D of estimates of shortest distances.
  - Initialize $D[source] = 0$, everything else $D[i] = \infty$.
  - This says our estimate from the source to the source is 0, everything else is ∞ initially.
- While $S \neq V$ (or while we have not considered all vertices):
  1. Find the vertex with the minimum dist (not already in S).
  2. Add this vertex, v_i to S
  3. Recompute all estimates based on v_i .
     - If $D[i] + w_{ij} < D[j]$ then set $D[j] = D[i] + w_{ij}$

Dijkstra's Algorithm

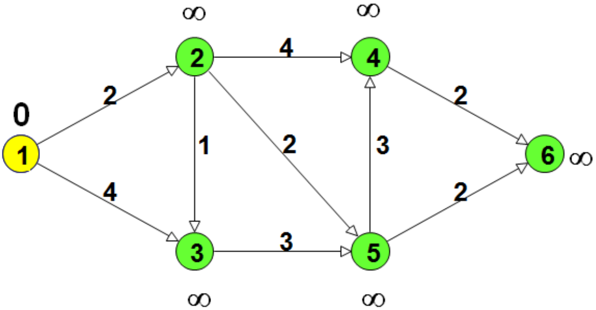
- Dijkstra's algorithm relies on:
- Knowing that all shortest paths contain subpaths that are also shortest paths.
- **Example:** The shortest path from a to b is 10,
  - So the shortest path from a to d has to go through b
  - So it will also contain the shortest path from a to b which is 10, plus the additional 2 = 12.



Greedy Algorithms

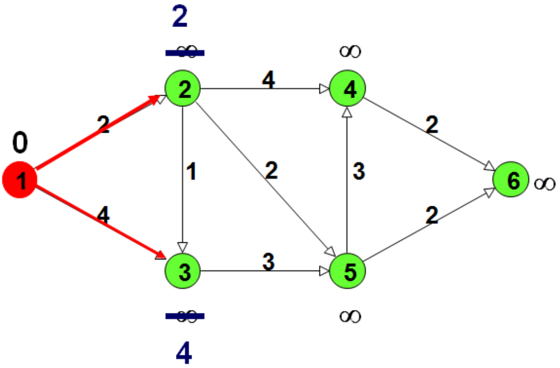
- Greedy algorithms work in phases.
  - In each phase, a decision is made that appears to be good, without regard for future consequences.
  - "Take what you can get now"
  - When the algorithm terminates, we hope that the local optimum is equal to the global optimum.
- This is the reason why Dijkstra's works out well as a greedy algorithm
  - It is greedy because we assume we have a shortest distance to a vertex before we ever examine all the edges that even lead into that vertex.
  - It works since all shortest paths contain subpaths that are also shortest paths.
  - This also works because we assume no negative edge weights.

Example

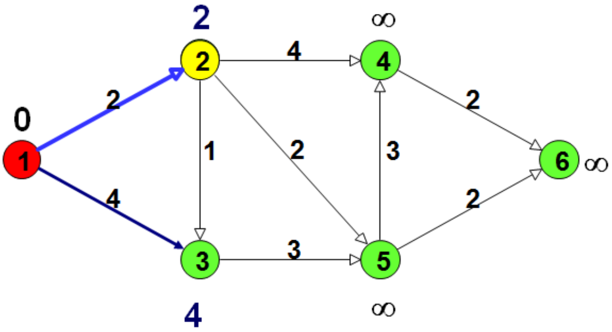


- Initialize
  - Select the node with the minimum temporary distance label.

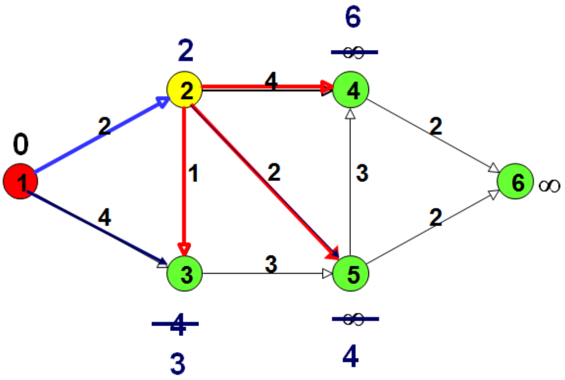
Update Step



Choose Minimum Temporary Label

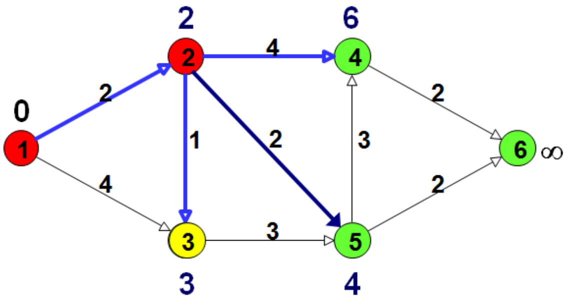


Update Step

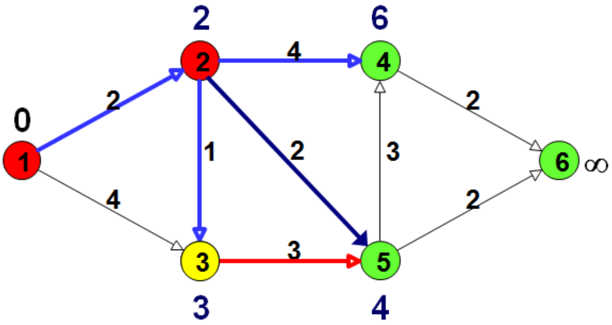


- The predecessor of node 3 is now node 2

Choose Minimum Temporary Label

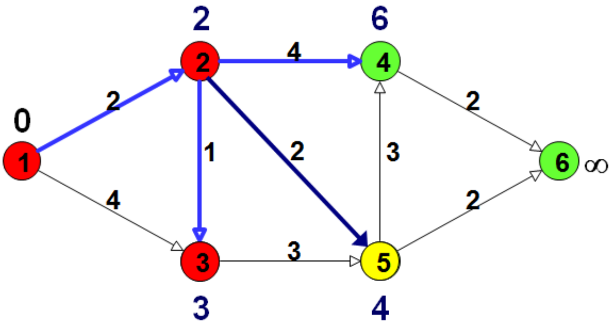


Update Step

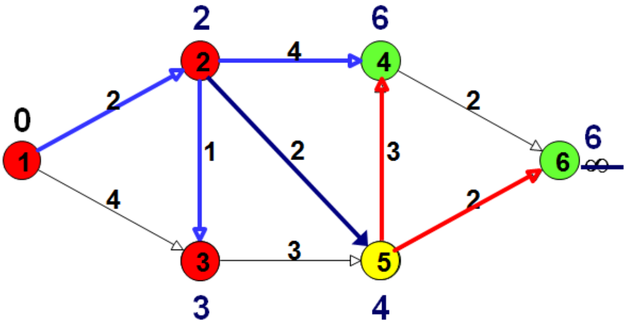


- d(5) is not changed.

Choose Minimum Temporary Label

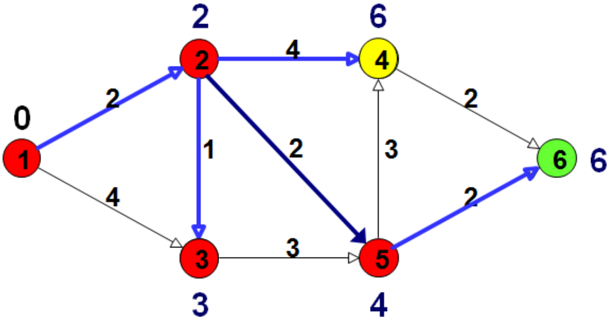


Update Step

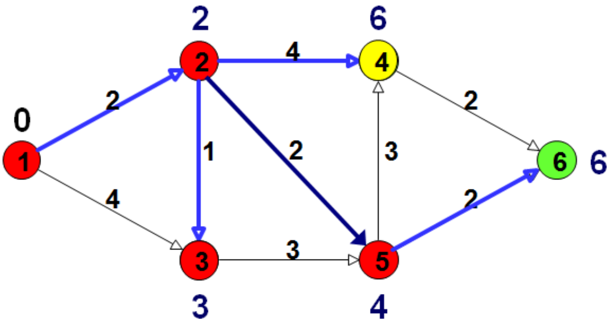


- d(4) is not changed.

Choose Minimum Temporary Label

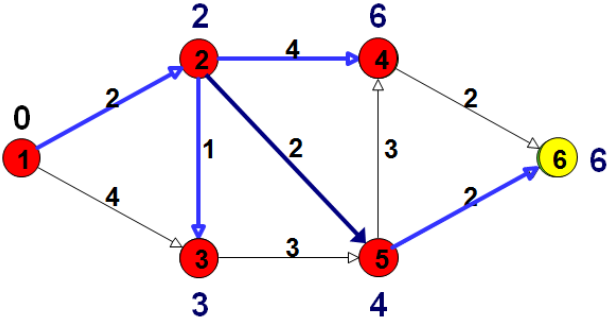


Update Step



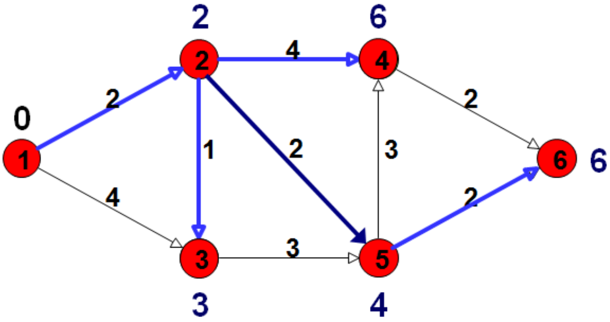
- d(6) is not changed.

Choose Minimum Temporary Label



- There is nothing to update

End of Algorithm



- All nodes are now permanent
- The predecessors form a tree
- The shortest path from node 1 to node 6 can be found by tracing back predecessors

Time Complexity: Using List

- The simplest implementation of the Dijkstra's algorithm stores vertices in an ordinary linked list or array
  - Good for dense graphs (many edges)
- $|V|$ vertices and $|E|$ edges
- Initialization $O(|V|)$
- While loop $O(|V|)$
  - Find and remove min distance vertices $O(|V|)$
- Potentially $|E|$ updates
  - Update costs $O(1)$
- Total time $O(|V^2| + |E|) = O(|V^2|)$

Time Complexity: Using List

- For sparse graphs, (i.e. graphs with much less than $|V^2|$ edges)
  - Dijkstra's algorithm can be implemented more efficiently by priority queue
- Initialization $O(|V|)$ using $O(|V|)$ buildHeap
- While loop $O(|V|)$
  - Find and remove min distance vertices $O(log|V|)$ using $O(log|V|)$ deleteMin
- Potentially $|E|$ updates
  - Update costs $O(log|V|)$
- Total time $O(|V|log|V| + |E|log|V|) = O(|E|log|V|)$
  - $|V| = O(|E|)$ assuming a connected graph

Dijkstra's Algorithm Implementation

@PS