# Ch 14 Graph Algorithms

## 14.1 Graphs

***graph***: way of representing relationships that exist between pairs of objects.

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***

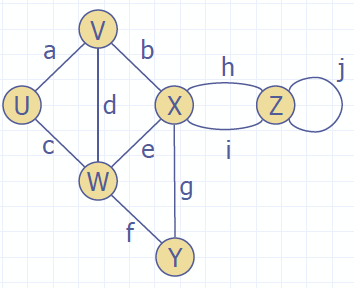
Vertices and edges are positions and store elements

### Edge Types

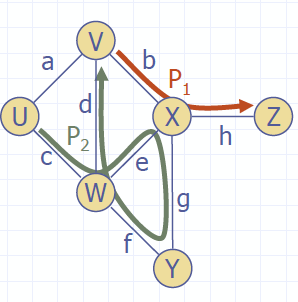
Edges in a graph are either directed or undirected. An edge (u,v) is said to be directed from u to v if the pair (u,v) is ordered, with u preceding v. An edge (u,v) is said to be undirected if the pair (u,v) is not ordered

|  |  |
| --- | --- |
| ***Directed edge***  ordered pair of vertices (u,v)  first vertex u is the origin  second vertex v is the destination  e.g., a flight | ***Undirected edge***  unordered pair of vertices (u,v)  notation: {u, v}  e.g., a flight route |
| ***Directed graph***  all the edges are directed  called a digraph  e.g., route network | ***Undirected graph***  all the edges are undirected  also called mixed graph  e.g., flight network |

### Terminology

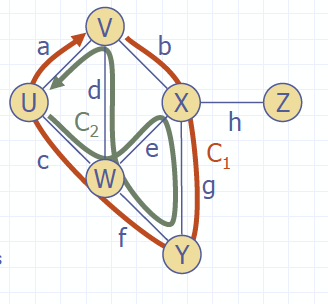


|  |  |  |
| --- | --- | --- |
| ***End vertices (endpoints)*** of an edge   * U and V are the endpoints of a * If edge is directed, its first endpoint is its origin and the other is destination of edge. | ***Edges incident on a vertex***  a, d, and b are incident on V | ***Adjacent vertices***  U and V are adjacent |
| ***Degree of a vertex***  X has degree 5 | ***Parallel edges***  -2 directed edges that have same origin & same destination.  -h and i are parallel edges | ***Self-loop***  -if its 2 endpoints coincide  -j is a self-loop |
| ***Outgoing edges of a vertex*** are the directed edges whose origin is that vertex | ***Incoming edges of a vertex*** are the directed edges whose destination is that vertex. | ***in-degree and out-degree of a vertex v*** are the number of incoming and outgoing edges of v, and are denoted indeg(v) and outdeg(v) |

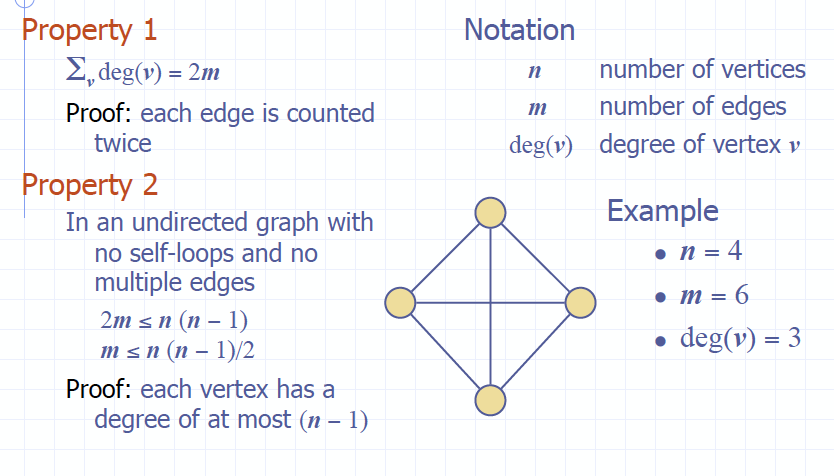


|  |  |  |
| --- | --- | --- |
| ***Path***   * sequence of alternating vertices and edges * begins with a vertex * ends with a vertex * each edge is preceded and followed by its endpoints | ***Simple path***   * path such that all its vertices and edges are distinct | ***Examples***  P1=(V,b,X,h,Z) is a simple path  P2=(U,c,W,e,X,g,Y,f,W,d,V) is a path that is not simple |

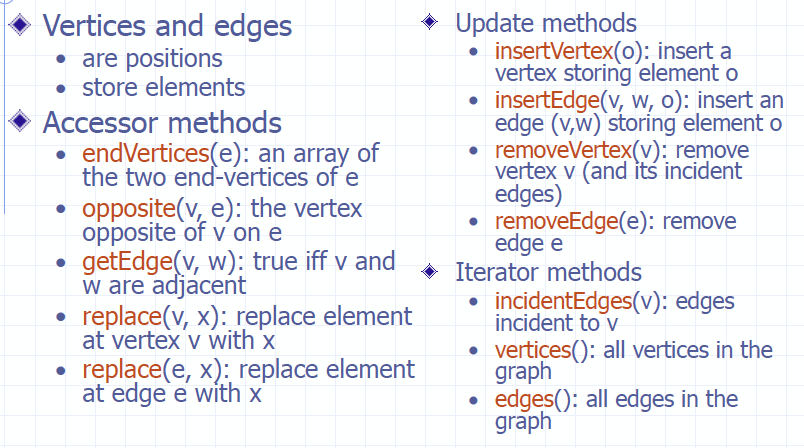
|  |  |  |
| --- | --- | --- |
| ***Cycle***   * path that starts & ends at same vertex, & that includes at least one edge * each edge is preceded and followed by its endpoints | ***Simple cycle***   * cycle such that all its vertices and edges are distinct | ***Examples***  C1=(V,b,X,g,Y,f,W,c,U,a,V) is a  simple cycle  C2=(U,c,W,e,X,g,Y,f,W,d,V,a,U) is a cycle that is not simple |



### Properties of Graph

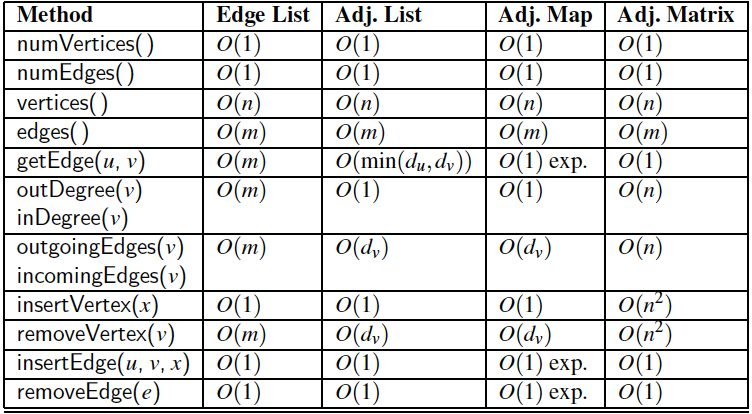


### Main Methods of Graph



## 14.2 Data Structures for Graphs

|  |  |
| --- | --- |
| ***In an edge list*** | we maintain an unordered list of all edges. This minimally suffices, but there is no efficient way to locate a particular edge (u,v), or the set of all edges incident to a vertex v. |
| ***In an adjacency list*** | we additionally maintain, for each vertex, a separate list containing those edges that are incident to the vertex. This organization allows us to more efficiently find all edges incident to a given vertex |
| ***An adjacency map*** | is similar to an adjacency list, but the secondary container of all edges incident to a vertex is organized as a map, rather than as a list, with the adjacent vertex serving as a key. This allows more efficient access to a specific edge (u,v), for example, in O(1) expected time with hashing. |
| ***An adjacency matrix*** | provides worst-case O(1) access to a specific edge (u,v) by maintaining an n×n matrix, for a graph with n vertices. Each  slot is dedicated to storing a reference to the edge (u,v) for a particular pair of vertices u and v; if no such edge exists, the slot will store null. |



### Edge List Structure

All vertex objects are stored in an unordered list V, and all edge objects are stored in an unordered list E.

|  |  |
| --- | --- |
| ***Vertex object***   * element * reference to position in vertex sequence   ***Vertex sequence***   * sequence of vertex objects   ***Edge object***   * element * origin vertex object * destination vertex object * reference to position in edge sequence   ***Edge sequence***   * sequence of edge objects | ***Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2017-12-10 at 3.02.53 PM.png*** |

|  |  |  |
| --- | --- | --- |
| **n vertices, m edges**  **No parallel edges**  **No self loops**  **Bounds are big-Oh** | **Edge List** |  |
| ***Space*** | n+m |  |
| ***incidentEdges(v)*** | m | *must find out who connects to who, iterates over hashlist* |
| ***getEdge(v, w)*** | m | *iterates over hashlist* |
| ***insertVertex(o)*** | 1 | *stable* |
| ***insertEdge(v, w, o)*** | 1 | *stable* |
| ***removeVertex(v)*** | m | *remove jeads to reiterate to redo connections* |
| ***removeEdge(e)*** | 1 | *stable* |

### Adjacency List Structure

|  |  |
| --- | --- |
| ***Edge list structure***  ***Incidence sequence for each vertex***   * sequence of references to edge objects of incident edges   ***Augmented edge objects***   * references to associated positions in incidence sequences of end vertices | ***Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2017-12-10 at 3.01.57 PM.png*** |

|  |  |  |
| --- | --- | --- |
|  | **Adjacency List** |  |
| ***Space*** | n+m |  |
| ***incidentEdges(v)*** | deg(v) | *depends of the degree of v* |
| ***getEdge(v, w)*** | min(deg(v), deg(w)) |  |
| ***insertVertex(o)*** | 1 |  |
| ***insertEdge(v, w, o)*** | 1 |  |
| ***removeVertex(v)*** | deg(v) |  |
| ***removeEdge(e)*** | 1 |  |

### Adjacency Map Structure

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ***Edge list structure***  Hash-based map for incident edges for each vertez | |  | | |
|  | **Adjacency Map** | |  |
| ***Space*** | n+m | |  |
| ***incidentEdges(v)*** | deg(v) | | *depends of the degree of v* |
| ***getEdge(v, w)*** | 1 *(expected)* | | *if lots of collisions it can be n* |
| ***insertVertex(o)*** | 1 | | *if all map to same element, more collisions, n* |
| ***insertEdge(v, w, o)*** | 1 *(expected)* | |  |
| ***removeVertex(v)*** | deg(v) | |  |
| ***removeEdge(e)*** | 1 *(expected)* | |  |

### Adjacency Matrix Structure

|  |  |
| --- | --- |
| ***Edge list structure***  ***Augmented vertex objects***  Integer key (index) associated with vertex  ***2D adjacency array***  Reference to edge object for adjacent vertices  Null for non-adjacent vertices  ***The “old fashioned” version just has 0 for no edge and 1 for edge*** |  |

|  |  |  |
| --- | --- | --- |
|  | **Adjacency matrix** |  |
| ***Space*** | n2 |  |
| ***incidentEdges(v)*** | n | *need to lopp to know which bit is switched on* |
| ***getEdge(v, w)*** | 1 | *you have index to check in 2D arr* |
| ***insertVertex(o)*** | n2 | *use if static, not dynamic* |
| ***insertEdge(v, w, o)*** | 1 |  |
| ***removeVertex(v)*** | n2 | *use if static, not dynamic* |
| ***removeEdge(e)*** | 1 |  |

## 14.3 Trees, Graphs, Forest

### Subgraphs

|  |  |
| --- | --- |
| A ***subgraph S of a graph G*** is a graph such that  The vertices of S are a subset of the vertices of G  The edges of S are a subset of the edges of G  A ***spanning subgraph of G*** is a subgraph that contains all the vertices of G | Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2017-12-10 at 3.19.34 PM.png |

### Connectivity

|  |  |
| --- | --- |
| A ***graph is connected*** if there is a path  between every pair of vertices  A ***connected component of a graph G*** is a maximally connected subgraph of G | Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2017-12-10 at 3.20.34 PM.png |

### Trees and Forests

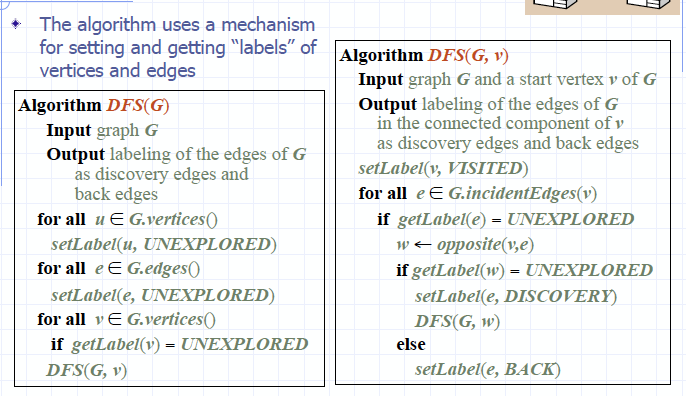
|  |  |
| --- | --- |
| A ***(free) tree*** is an undirected graph T such that  • T is connected  • T has no cycles  *This definition of a tree is different from the one of a rooted tree*  A ***forest*** is an undirected graph without cycles  The connected components of a forest  are ***trees*** | Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2017-12-10 at 3.21.55 PM.png |

### Spanning Trees and Forests

|  |  |
| --- | --- |
| ***Spanning tree of a connected graph*** is a  spanning subgraph that is a tree  Spanning tree is ***not unique*** unless the graph is a tree  Spanning trees have applications to the design of communication networks  ***Spanning forest of a graph*** is a spanning subgraph that is a forest | Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2017-12-10 at 3.23.12 PM.png |

## 14.3.1 Depth-First Search

|  |  |
| --- | --- |
| Depth-first search (DFS) is a ***general technique for traversing a grap***h  A DFS traversal of a graph G  • Visits all the vertices and edges of G  • Determines whether G is connected  • Computes the connected components of G  • Computes a spanning forest of G | DFS on a graph with n vertices and m edges takes ***O(n+m) time***  DFS can be further extended to solve other graph problems  • Find and report a path between two given vertices  • Find a cycle in the graph  Depth-first search is to graphs what Euler tour is to binary trees |



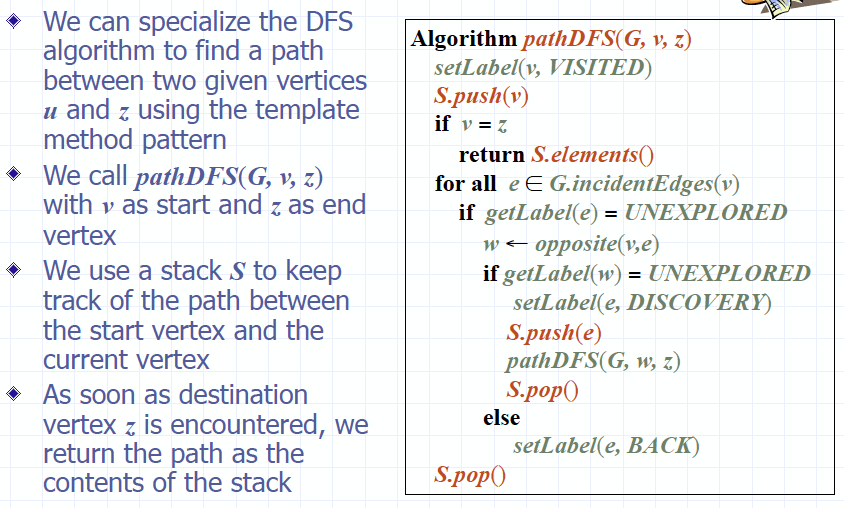
### Properties of DFS

|  |  |
| --- | --- |
| ***Property 1***  DFS(G, v) visits all the vertices and edges in the connected component of v | ***Property 2***  The discovery edges labeled by DFS(G, v) form a spanning tree of the connected component of v |

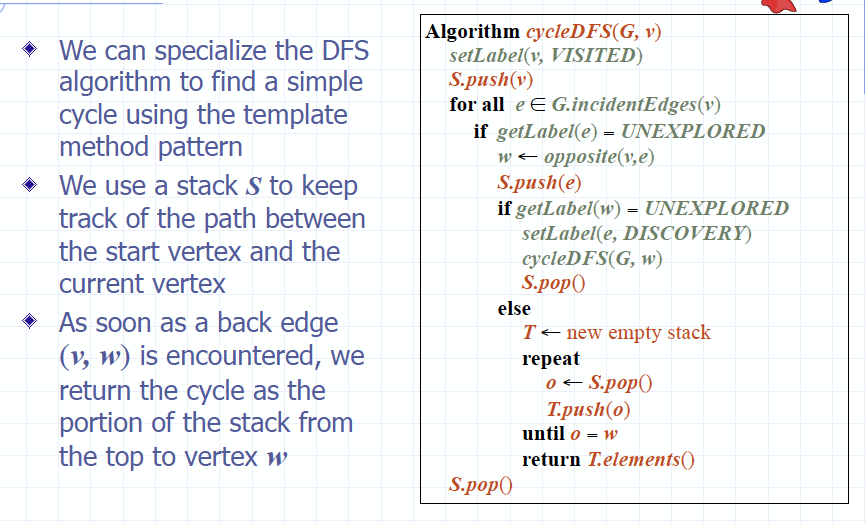
### Analysis DFS

|  |  |
| --- | --- |
| Setting/getting a vertex/edge label takes ***O(1) time***  Each ***vertex*** is ***labeled twice***  • once as UNEXPLORED  • once as VISITED  Each ***edge*** is ***labeled twice***  • once as UNEXPLORED  • once as DISCOVERY or BACK | Method incidentEdges is called once for each vertex  DFS runs in ***O(n + m) time*** provided the graph is represented by the adjacency list structure  • Recall that Σv deg(v) = 2m |

### Path Finding

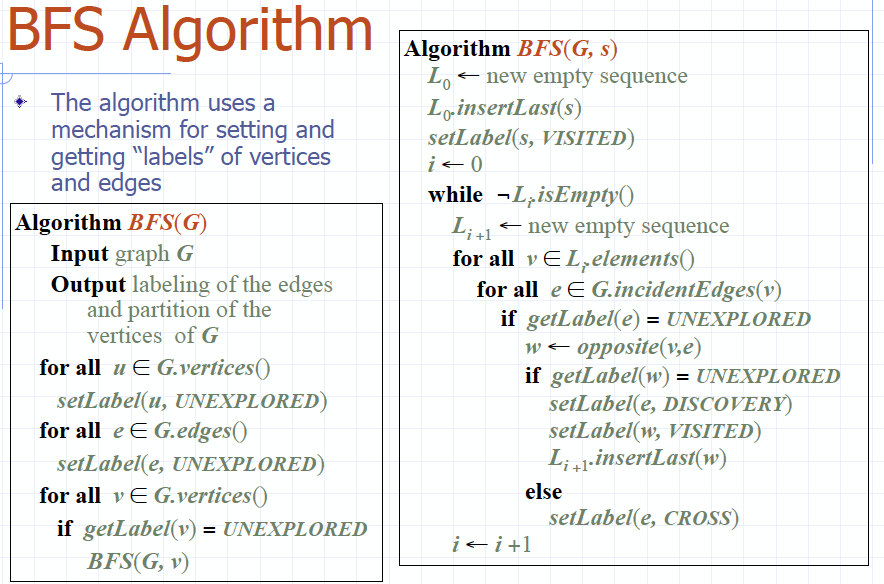


### Cycle Finding



### 14.4 Breadth-First Search (BFS)

|  |  |
| --- | --- |
| BFS is a ***general technique for traversing a graph***  A BFS traversal of a graph G  • Visits all the vertices and edges of G  • Determines whether G is connected  • Computes the connected components of G  • Computes a spanning forest of G | BFS on a graph with n vertices and m edges takes ***O(n + m ) time***  BFS can be further extended to solve other graph problems  • Find and report a path with the minimum number of edges between two given vertices  • Find a simple cycle, if there is one |



### Properties BFS

|  |  |
| --- | --- |
| ***Notation***  Gs: connected component of s  ***Property 1***  BFS(G, s) visits all the vertices and edges of Gs  ***Property 2***  The discovery edges labeled by  BFS(G, s) form a spanning tree Ts of Gs | ***Property 3***  For each vertex v in Li  • The path of Ts from s to v has i edges  • Every path from s to v in Gs has at least i edges |

### Analysis BFS

|  |  |
| --- | --- |
| Setting/getting a vertex/edge label takes ***O(1) time***  Each ***vertex*** is ***labeled twice***  • once as UNEXPLORED  • once as VISITED  Each ***edge*** is ***labeled twice***  • once as UNEXPLORED  • once as DISCOVERY or CROSS | Each vertex is inserted once into a sequence Li  Method incidentEdges() is called once for each vertex  BFS runs in ***O(n + m) time*** provided the graph is represented by the adjacency list structure  • Recall that Σv deg(v) = 2m |

### Applications

Using the template method pattern, we can specialize the ***BFS traversal*** of a graph G to solve the following problems in ***O(n + m) time***

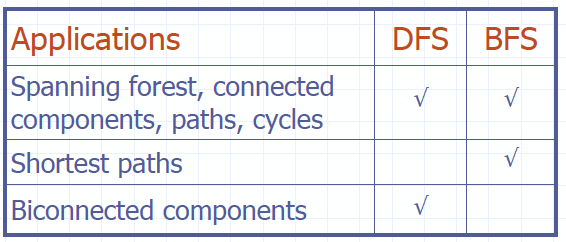
• Compute the connected components of G

• Compute a spanning forest of G

• Find a simple cycle in G, or report that G is a forest

• Given two vertices of G, find a path in G between them with the minimum number of edges, or report that no such path exists

### DFS vs BFS



|  |  |
| --- | --- |
| Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2017-12-10 at 3.41.37 PM.png | Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2017-12-10 at 3.41.41 PM.png |
| ***Back edge (v,w)***  • w is an ancestor of v in the tree of discovery edges | ***Cross edge (v,w)***  • w is in the same level as v or in the next level in the tree of discovery edges |

## 14.5 Digraphs

|  |  |
| --- | --- |
| A digraph is a graph whose edges are all directed  • ***Short for “directed graph”***  ***Applications***  • one-way streets  • flights  • task scheduling | Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2017-12-10 at 3.44.44 PM.png |
| Digraph Properties A graph G=(V,E) such that  • Each edge goes in one direction:  ***Edge (a,b) goes from a to b, but not b to a***.  If G is ***simple***, ***m < n\*(n-1).***  If we keep in-edges and out-edges in separate adjacency lists, we can perform listing of in-edges and out-edges in time proportional to their size | |
| ***Digraph Applications***  Scheduling: edge (a,b) means task a must be completed before b can be started | |

### Directed DFS

|  |  |
| --- | --- |
| We can specialize the traversal algorithms (DFS and BFS) to digraphs by traversing edges only along their direction  In the directed DFS algorithm, we have ***four types of edges***  • discovery edges  • back edges  • forward edges  • cross edges  A directed DFS starting at a vertex s determines the vertices reachable from s | Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2017-12-10 at 3.48.03 PM.png |

### Reachability

DFS tree rooted at v: vertices reachable from v via directed paths.

|  |  |
| --- | --- |
| Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2017-12-10 at 3.51.10 PM.png | Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2017-12-10 at 3.51.14 PM.png |

### Strong Connectivity

|  |  |
| --- | --- |
| Each vertex can reach all other vertices |  |

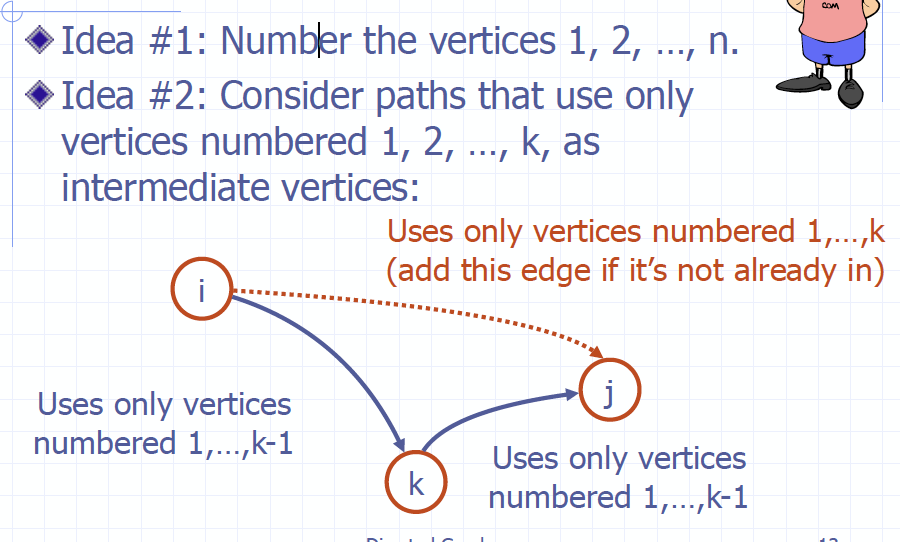
***Strong Connectivity Algorithm***

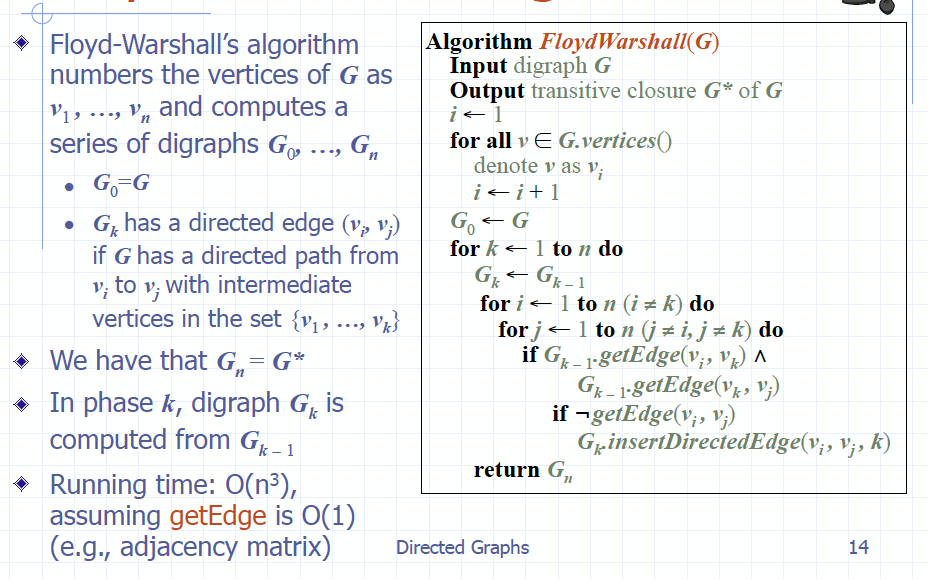
|  |  |
| --- | --- |
| Pick a vertex v in G  Perform a DFS from v in G  • If there’s a w not visited, print “no”  Let G’ be G with edges reversed  Perform a DFS from v in G’  • If there’s a w not visited, print “no”  • Else, print “yes”  Running time: ***O(n+m)*** | Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2017-12-10 at 3.53.57 PM.png |

### Transitive Closure

|  |  |
| --- | --- |
| Given a digraph G, the ***transitive closure of G is the digraph G\**** such that  • G\* has the same vertices as G  • if G has a directed path from u to v (u ≠ v), G\* has a directed edge from u to v  ***Transitive closure provides reachability information about a digraph.***  ***Computing Transitive closure: O(n(n+m))*** | Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2017-12-10 at 3.56.00 PM.png |

### Floyd-Warshall Transitive Closure

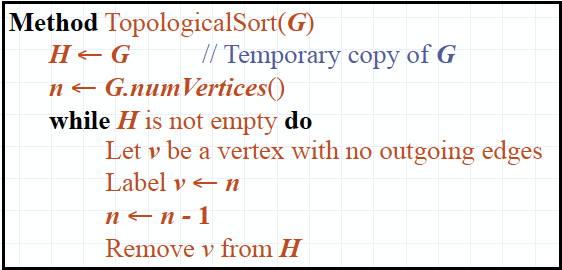




### DAGs and Topological Ordering

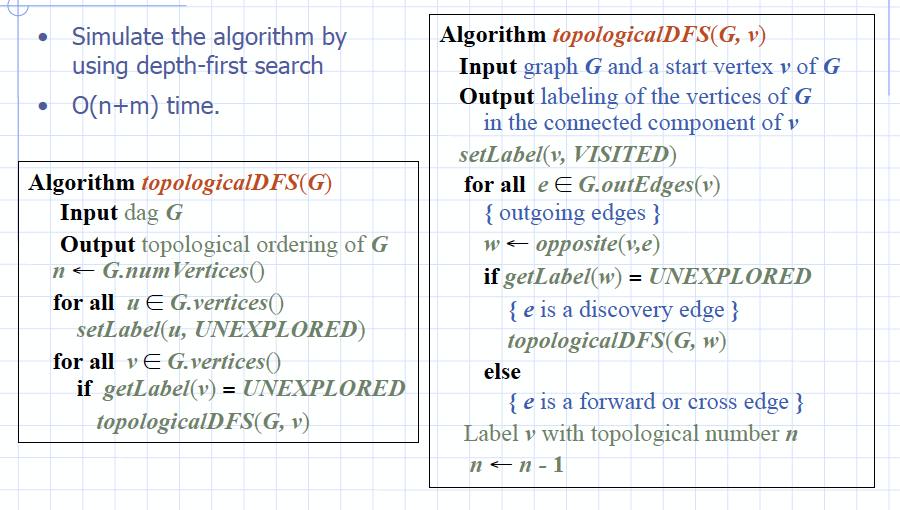
|  |  |
| --- | --- |
| ***directed acyclic graph (DAG)*** is a digraph that has no directed cycles  A ***topological ordering of a digraph*** is a ***numbering***  v1 , …, vn  of ***vertices*** such that for every edge (vi , vj), we have i < j  *Example*: in a task scheduling digraph, a topological ordering of a task sequence that satisfies the precedence constraints  ***Theorem***  A digraph admits a topological ordering if and only if it is a DAG | Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2017-12-10 at 4.03.24 PM.png |

***Algorithm for topological Sorting***



Run time: ***O(n + m)***

O(n) because adding vertex + then need to remove edges O(m)



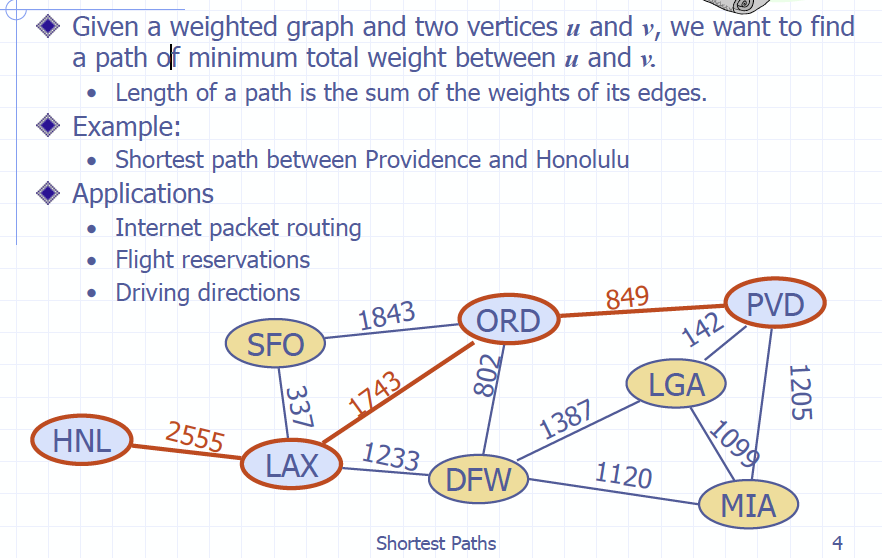
***Implementation with DFS***

## 14.6 Shortest Paths

### Weighted Graphs

|  |  |
| --- | --- |
| In a weighted graph, ***each edge has an associated numerical value, called the weight of the edge***  Edge weights may represent, distances, costs, etc. | *Example*: In a flight route graph, the weight of an edge represents the distance in miles between the endpoint airports.  Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2017-12-10 at 4.12.21 PM.png |

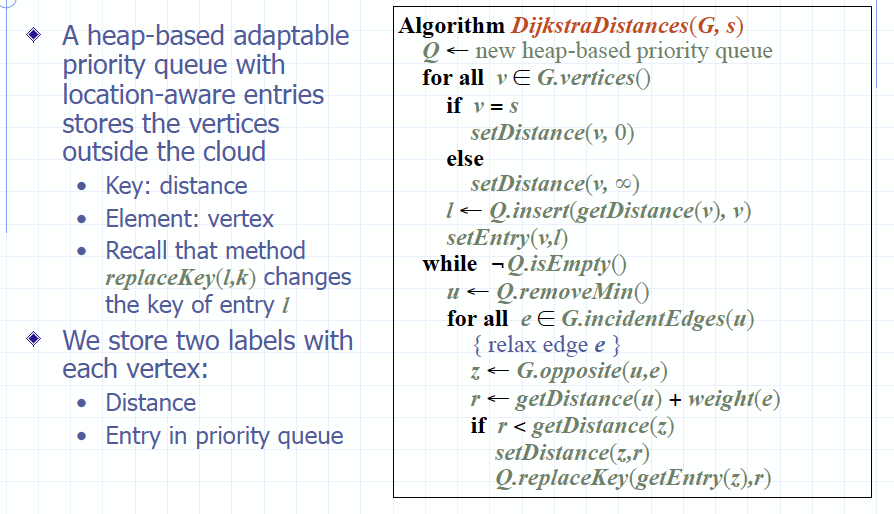
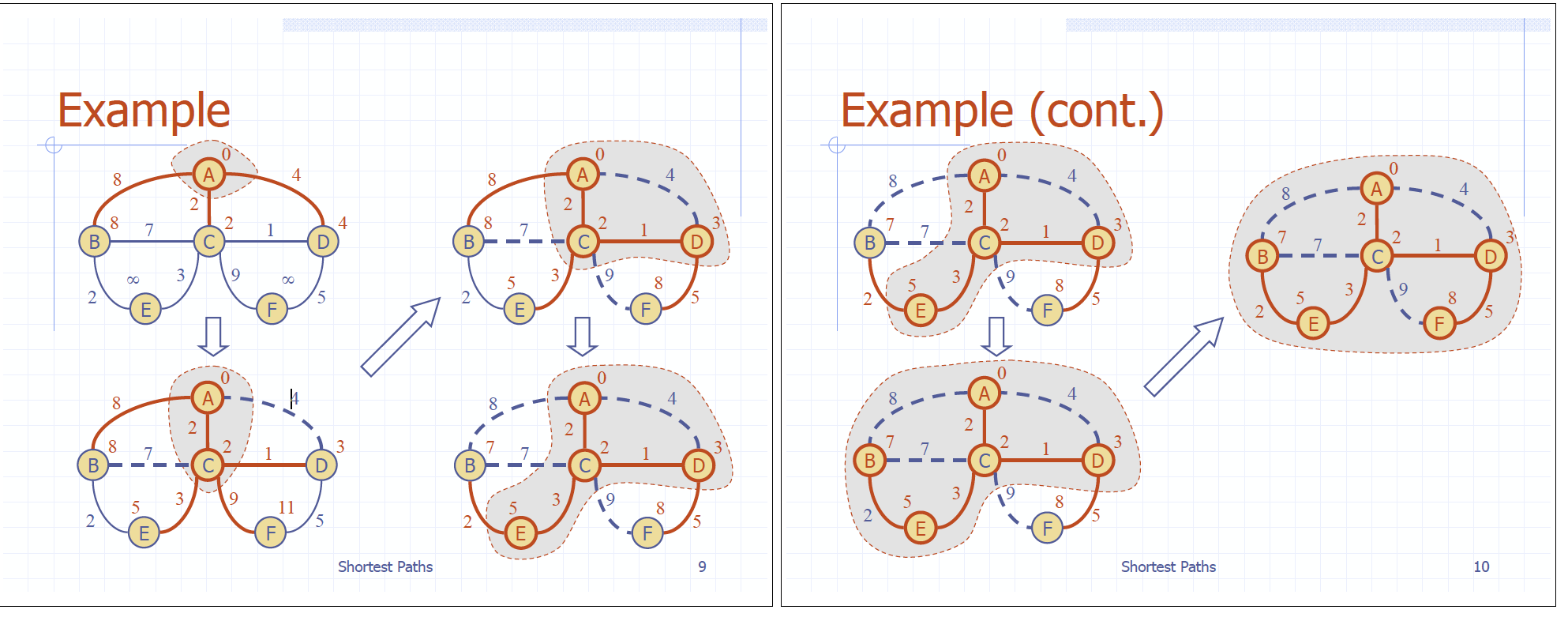
### Shortest Path Problem



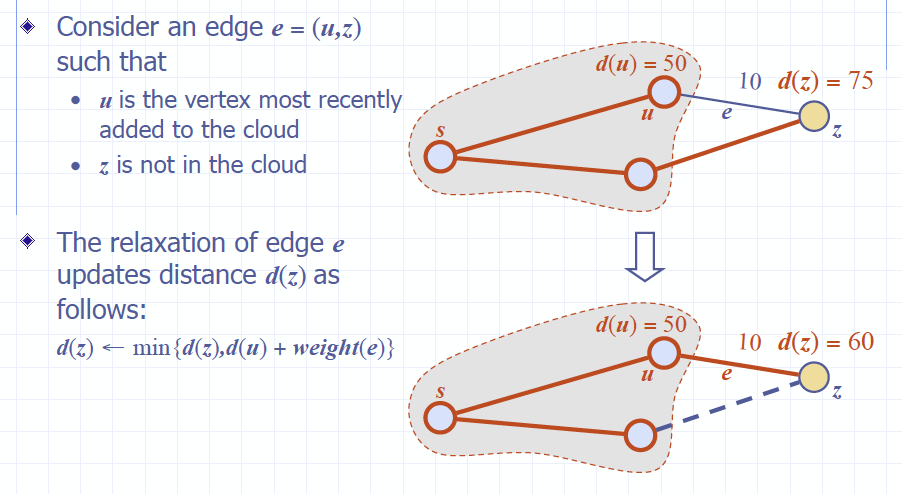
|  |  |
| --- | --- |
| ***Shortest Path Properties***  ***Property 1:***  A subpath of a shortest path is itself a shortest path  ***Property 2:***  There is a tree of shortest paths from a start vertex to all the other vertices | *Example*:  Tree of shortest paths from Providence |

### Dijkstra’s Algorithm

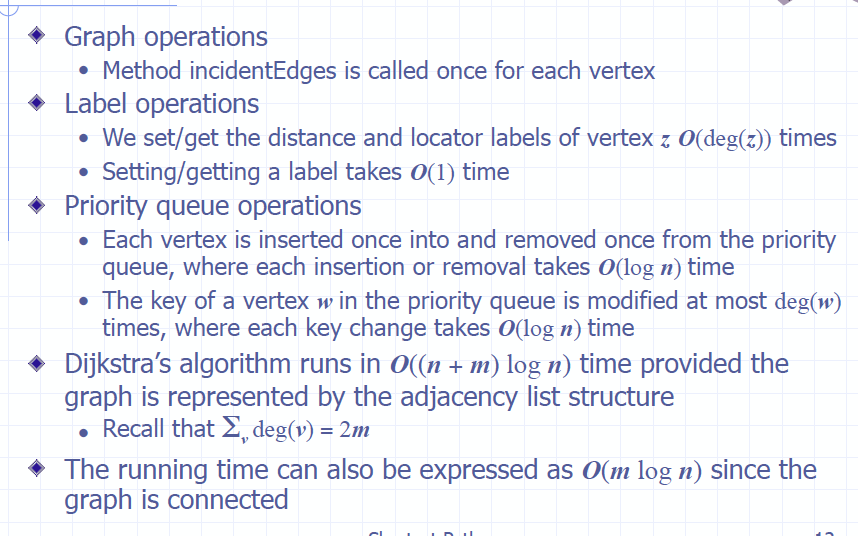
|  |  |
| --- | --- |
| The distance of a vertex *v* from a vertex *s* is the length of a shortest path between *s* and *v*  Dijkstra’s algorithm computes the distances of all the vertices from a given start vertex s  ***Assumptions***:  • the graph is connected  • the edges are undirected  • the edge weights are nonnegative | We grow a “cloud” of vertices, beginning with *s* and eventually covering all the vertices  We store with each vertex *v* a label d(*v*) representing the distance of *v* from *s* in the subgraph consisting of the cloud and its adjacent vertices  ***At each step***  • We add to the cloud the vertex u outside the cloud with the smallest distance label, d(u)  • We update the labels of the vertices adjacent to u |



### Edge Relaxation



### Analysis Dijkstra



### Why Dijkstra works

|  |  |
| --- | --- |
| Dijkstra’s algorithm is ***based on the greedy method. It adds vertices by increasing distance.*** | • Suppose it didn’t find all shortest distances. Let F be the first wrong vertex the algorithm processed.  • When previous node, D, on the true shortest path was considered, its distance was correct.  • But edge (D,F) was relaxed at that time!  • Thus, so long as d(F)>d(D), F’s distance cannot be wrong. That is, there is no wrong vertex. |

### Why Dijkstra doesn’t work for negative-weight edges

