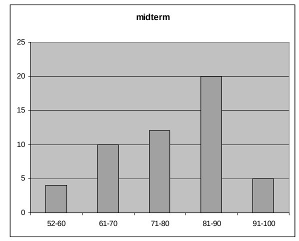
CSE 326: Data Structures
Lecture #16
Graphs I: DFS & BFS

Henry Kautz
Winter Quarter 2002

Midterm

Mean: 77
Std. Dev: 11
High score: 94



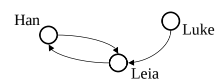
Outline

- Graphs (TO DO: READ WEISS CH 9)
- Graph Data Structures
- Graph Properties
- Topological Sort
- Graph Traversals
  - Depth First Search
  - Breadth First Search
  - Iterative Deepening Depth First
- Shortest Path Problem
  - Dijkstra's Algorithm

Graph ADT

Graphs are a formalism for representing relationships between objects

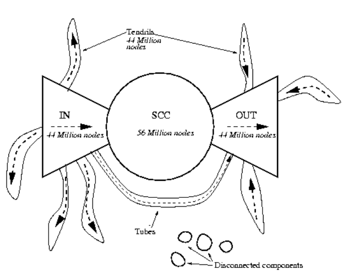
- a graph **G** is represented as
  G = (V, E)
  - **V** is a set of vertices
  - **E** is a set of edges



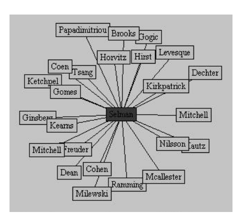

```
V = {Han, Leia, Luke}
E = {(Luke, Leia),
     (Han, Leia),
     (Leia, Han)}
```

- operations include:
  - iterating over vertices
  - iterating over edges
  - iterating over vertices adjacent to a specific vertex
  - asking whether an edge exists connected two vertices

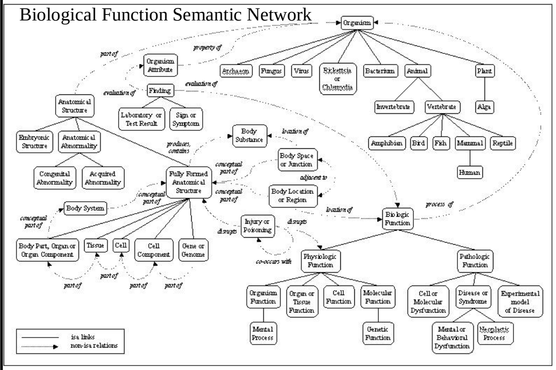
What Graph is THIS?



ReferralWeb
(co-authorship in scientific papers)

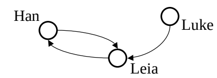


Biological Function Semantic Network



Graph Representation 1: Adjacency Matrix

A `|V| x |V|` array in which an element `(u, v)` is true if and only if there is an edge from `u` to `v`



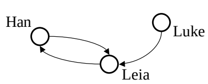
	Han	Luke	Leia
Han			
Luke			
Leia			

Runtime:
iterate over vertices
iterate ever edges
iterate edges adj. to vertex
edge exists?

Space requirements:

Graph Representation 2: Adjacency List

A `|V|`-ary list (array) in which each entry stores a list (linked list) of all adjacent vertices



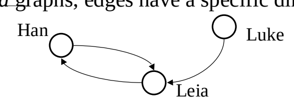
Han	
Luke	
Leia	

Runtime:
iterate over vertices
iterate ever edges
iterate edges adj. to vertex
edge exists?

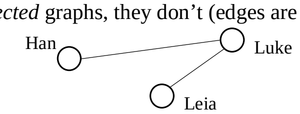
space requirements:

Directed vs. Undirected Graphs

- In *directed* graphs, edges have a specific direction:



- In *undirected* graphs, they don't (edges are two-way):



- Vertices `u` and `v` are *adjacent* if `(u, v)` ∈ `E`

Graph Density

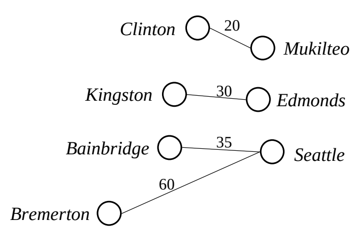
A *sparse* graph has O(|V|) edges

A *dense* graph has Θ(|V|^2) edges

Anything in between is either *sparsish* or *densy* depending on the context.

Weighted Graphs

Each edge has an associated weight or cost.

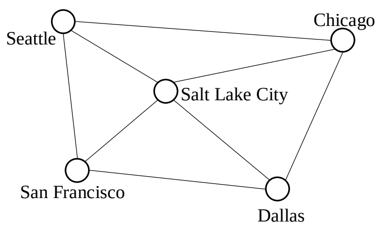


There may be more information in the graph as well.

Paths and Cycles

A *path* is a list of vertices **{v_1, v_2, …, v_n}** such that **(v_i, v_{i+1}) ∈ E** for all **$0 \le i < n$**.

A *cycle* is a path that begins and ends at the same node.

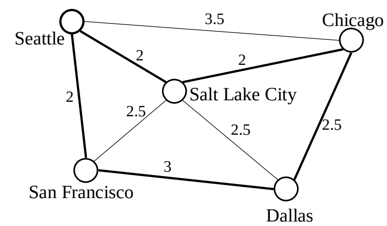


p = {Seattle, Salt Lake City, Chicago, Dallas, San Francisco, Seattle}

Path Length and Cost

Path length: the number of edges in the path

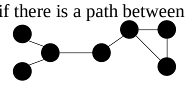
Path cost: the sum of the costs of each edge



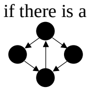
length(p) = 5 cost(p) = 11.5

Connectivity

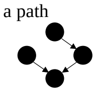
Undirected graphs are *connected* if there is a path between any two vertices



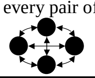
Directed graphs are *strongly connected* if there is a path from any one vertex to any other



Directed graphs are *weakly connected* if there is a path between any two vertices, *ignoring direction*

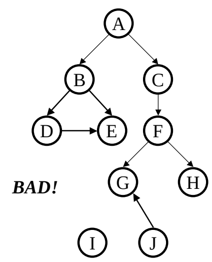


A *complete* graph has an edge between every pair of vertices



Trees as Graphs

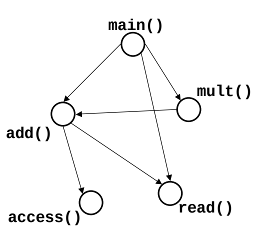
- Every tree is a graph with some restrictions:
  - the tree is *directed*
  - there are *no cycles* (directed or undirected)
  - there is a *directed path from the* root *to every node*



Directed Acyclic Graphs (DAGs)

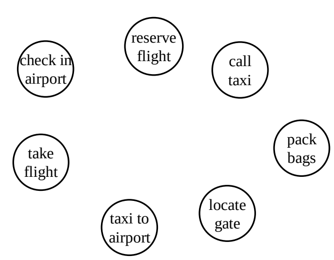
DAGs are directed graphs with no cycles.

if program call graph is a DAG, then all procedure calls can be in-lined



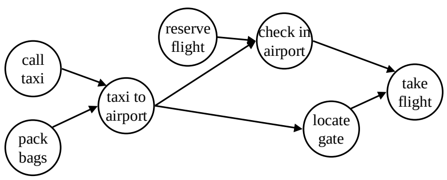
Trees ⊂ DAGs ⊂ Graphs

Application of DAGs: Representing Partial Orders



Topological Sort

Given a graph, **`G = (V, E)`**, output all the vertices in **`V`** such that no vertex is output before any other vertex with an edge to it.



Topo-Sort Take One

Label each vertex's *in-degree* (# of inbound edges)
While there are vertices remaining
    Pick a vertex with in-degree of zero and output it
    Reduce the in-degree of all vertices adjacent to it
    Remove it from the list of vertices

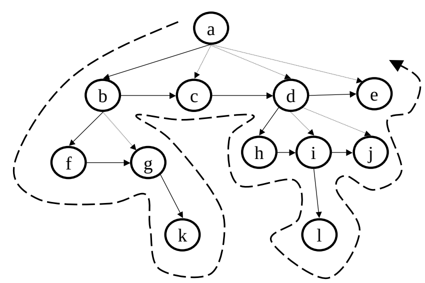
runtime:

Topo-Sort Take Two

Label each vertex's in-degree
Initialize a queue (or stack) to contain all in-degree zero vertices
While there are vertices remaining in the queue
    Remove a vertex *v* with in-degree of zero and output it
    Reduce the in-degree of all vertices adjacent to *v*
    Put any of these with new in-degree zero on the queue

runtime:

Recall: Tree Traversals



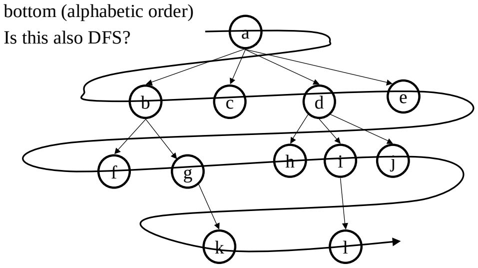
a b f g k c d h i l j e

Depth-First Search

- Both Pre-Order and Post-Order traversals are examples of depth-first search
  - nodes are visited deeply on the left-most branches before any nodes are visited on the right-most branches
    - *visiting the right branches deeply before the left would still be depth-first! Crucial idea is "go deep first!"*
- In DFS the nodes "being worked on" are kept on a stack (where?)
- Recursion is a clue that DFS may be lurking…

Level-Order Tree Traversal

- Consider task of traversing tree level by level from top to bottom (alphabetic order)
- Is this also DFS?



Breadth-First Search

- No! Level-order traversal is an example of Breadth-First Search
- BFS characteristics
  - Nodes being worked on maintained in a FIFO Queue, not a stack
  - Iterative style procedures often easier to design than recursive procedures

Put root in a Queue

Repeat until Queue is empty:

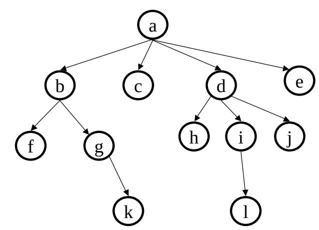
Dequeue a node

Process it

Add it's children to queue

QUEUE

a
b c d e
c d e f g
d e f g
e f g h i j
f g h i j
g h i j
h i j k
i j k
j k l
k l
l



Graph Traversals

- Depth first search and breadth first search also work for arbitrary (directed or undirected) graphs
  - Must mark visited vertices so you do not go into an infinite loop!
- Either can be used to determine connectivity:
  - Is there a path between two given vertices?
  - Is the graph (weakly) connected?
- Important difference: Breadth-first search always finds a shortest path from the start vertex to any other (for unweighted graphs)
  - Depth first search may not!

Demos

DFS

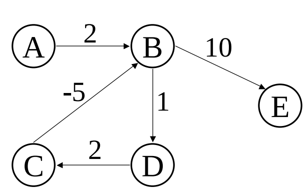
BFS

Single Source, Shortest Path for Weighted Graphs

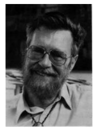
Given a graph **G = (V, E)** with edge costs c(e), and a vertex **s ∈ V**, find the shortest (lowest cost) path from s to every vertex in **V**

- Graph may be directed or undirected
- Graph may or may not contain cycles
- Weights may be all positive or not
- What is the problem if graph contains cycles whose total cost is negative?

The Trouble with Negative Weighted Cycles



Edsger Wybe Dijkstra



Legendary figure in computer science; now a professor at University of Texas.

Supports teaching introductory computer courses without computers (pencil and paper programming)

Also famout for refusing to read e-mail; his staff has to print out messages and put them in his mailbox.

Dijkstra's Algorithm for Single Source Shortest Path

- Classic algorithm for solving shortest path in weighted graphs (with *only positive* edge weights)
- Similar to breadth-first search, but uses a priority queue instead of a FIFO queue:
  - Always select (expand) the vertex that has a lowest-cost path to the start vertex
  - a kind of "greedy" algorithm
- Correctly handles the case where the lowest-cost (shortest) path to a vertex is not the one with fewest edges

Pseudocode for Dijkstra

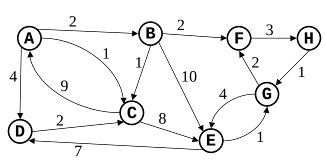
Initialize the cost of each *vertex* to ∞

```
cost[s] = 0;
heap.insert(s);
While (! heap.empty())
    n = heap.deleteMin()
    For (each vertex a which is adjacent to n along edge e)
       if (cost[n] + edge_cost[e] < cost[a]) then
           cost [a] = cost[n] + edge_cost[e]
           previous_on_path_to[a] = n;
           if (a is in the heap) then heap.decreaseKey(a)
                                    else heap.insert(a)
```

Important Features

- Once a vertex is removed from the head, the cost of the shortest path to that node is known
- While a vertex is still in the heap, another shorter path to it might still be found
- The shortest path itself from s to any node a can be found by following the pointers stored in previous_on_path_to[a]

Dijkstra's Algorithm in Action



vertex	known	cost
A		
B		
C		
D		
E		
F		
G		
H		

Demo

Dijkstra's

Data Structures for Dijkstra's Algorithm

|V| times:
Select the unknown node with the lowest cost

→ findMin/deleteMin O(log |V|)

|E| times:
a's cost = min(*a*'s old cost, …)

→ decreaseKey O(log |V|)

runtime: O(|E| log |V|)

Fibonacci Heaps

- A complex version of heaps - Weiss 11.4
- Used more in theory than in practice
- Amortized O(1) time bound for decreaseKey
- O(log n) time for deleteMin

Dijkstra's uses **|V|** deleteMins and **|E|** decreaseKeys

runtime with Fibonacci heaps: O(|E| + |V| log |V|)

for dense graphs, asymptotically better than O(|E| log |V|)