Graphs

* In the beginning, there were LISTS.
* A LIST always has one successor & one predecessor item.
* If we relax the definition of a list to have multiple successor elements, we end up with TREES.
* In a tree, we say there is exactly ONE path from the root of a tree to any leaf.
* Suppose we relax that constraint --- multiple paths from a node to another node.
* We get a ADT called a GRAPH.
* Graphs are useful for modeling any sort of set of relationships between pairs of items.
  + any kind of network---communication network, transportation network (roads, trains, planes)
  + friend network, course network showing pre-req

Terminology:

* Graphs are made up of a collection of 2 types of items: NODES (sometimes called vertices) and EDGES (sometimes called arcs).
* In lists and trees, the links (edges/arcs) always had a direction (predecessor/succ), parent/child. In graphs, sometimes the edges have an associated direction, and sometimes they don’t, depends on the relationship we’re trying to model.
  + A directed graph (digraph): edges have an associated direction, this is used in situations where the relationship between the two things is not reciprocal. Imagine pre-requisites for classes.  
    141 -> 142, 172, 231  
    142 -> 241
  + Undirected graph: facebook
  + What should a transportation network be? roads? bi-direction or uni-direction. one way streets.
* Legal for graphs to have cycles. Self loops are OK as well. Parallel edges are OK.

Define the ADT for a **directed graph**:

* Structure: A collection of elements called nodes/vertices. A collection of directed edges that connect pairs of nodes.
* operations:
* add node
* add edge between 2 nodes
* delete an edge
* delete a node (and all edges involving it)
* check if a node exists
* check if an edge exists
* get # of nodes
* get # edges
* other things: various kinds of traversals, determining if one node is reachable from another node, finding shortest path between nodes.

Often times, there is extra information that we need to store in the graph associated with either a node or a vertex.

example: Transportation. A vertex might need to store the name of a landmark, or a name of a city, or the name of a street intersection.

Edges could store distance between the places, time to travel between the places, the speed limit.  
  
Graph Terminology

* adjacent: vertex w is adjacent to vertex v iff (v,w)E.
  + can get to w from v
    - be careful with directed graphs
  + in an undirected graph with edge (v,w), v is adjacent to w and w is adjacent to v
  + ***Here, vertex v1 is adjacent to v2 if there is a directed edge from v1 to v2.***
* weight:
  + an edge often has a weight or a “cost” associated with it
* a path in a graph is a sequence of vertices w1, w2,…, wn such that (wi, wi+1), 1<= i <= N, where N is the number of vertices.
* A cycle, “loop”, is a directed graph such that w1=wN
  + CDC
* a directed graph is acyclic if it has no cycles (DAG)
* an undirected graph is connected if there is a path from every vertex to every other vertex
  + a directed graph with this property is called strongly connected
* a complete graph is a graph in which there is an edge between every pair of vertices

Say we have the following graph:

5 nodes Memphis, Little Rock, St Louis, Nashville, New Orleans.

Flight network.

Adjacency matrix representation

* We number the vertices 0, 1, 2, 3...
  + If we need to store additional info about a vertex other than this number, then include a map ADT that maps these numbers to the additional information.
* Create a square matrix of |V| rows and |V| columns.
* Unweighted: If there is an edge from vertex A to vertex B, then we put a 1 in the matrix at location [A][B].
* Weighted: Put the weight at location [A][B].

1. Adjacency Matrix

* for each edge (u,v), we set A[u][v] to true (or we can set it equal to the weight)
* Space requirements= V2
* adj matrix pros and cons.
  + pro = easy to test if something is in there.
  + cons = hard to add more things later on.

1. Adjacency List—if a graph is sparse (not dense)

* for each vertex, keep track of all adjacent vertices
  + storage O(V+E)
* Adjacency lists are stand ways to store graphs
* To store undirected graph, each edge U,V appear in two lists so space usage doubles

Adj list pros and cons.

If time, write code.

END OF DAY 1 – covered adj list/adj matrix and basic graph operations, and terminology. Didn’t cover big oh.

DAY 2:

main objectives – edge lists, big oh of operations, introduce V/E notation, traversals

Cover adj list, edge lists representations.

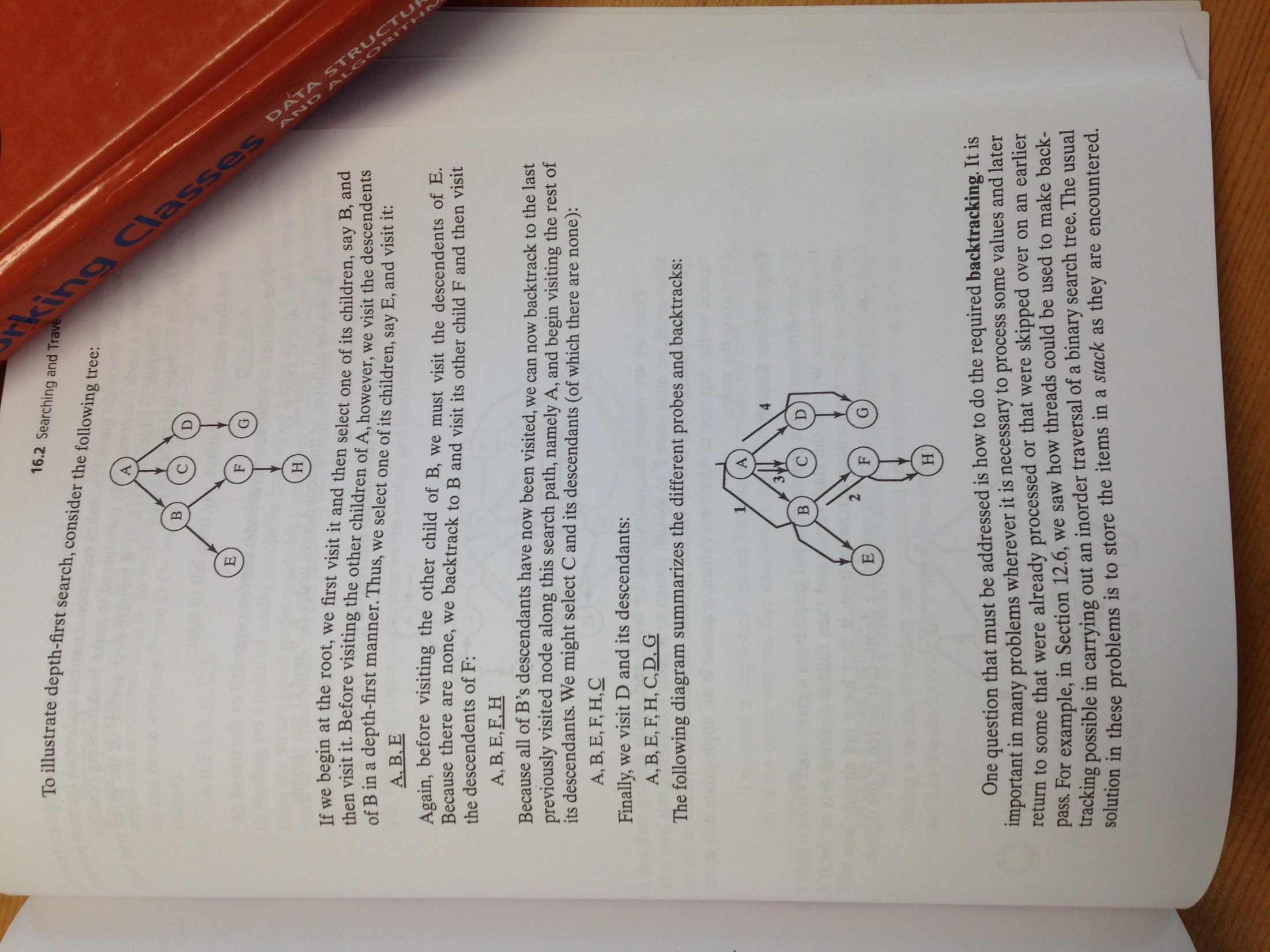
**FIX THIS FROM LAST TIME:**

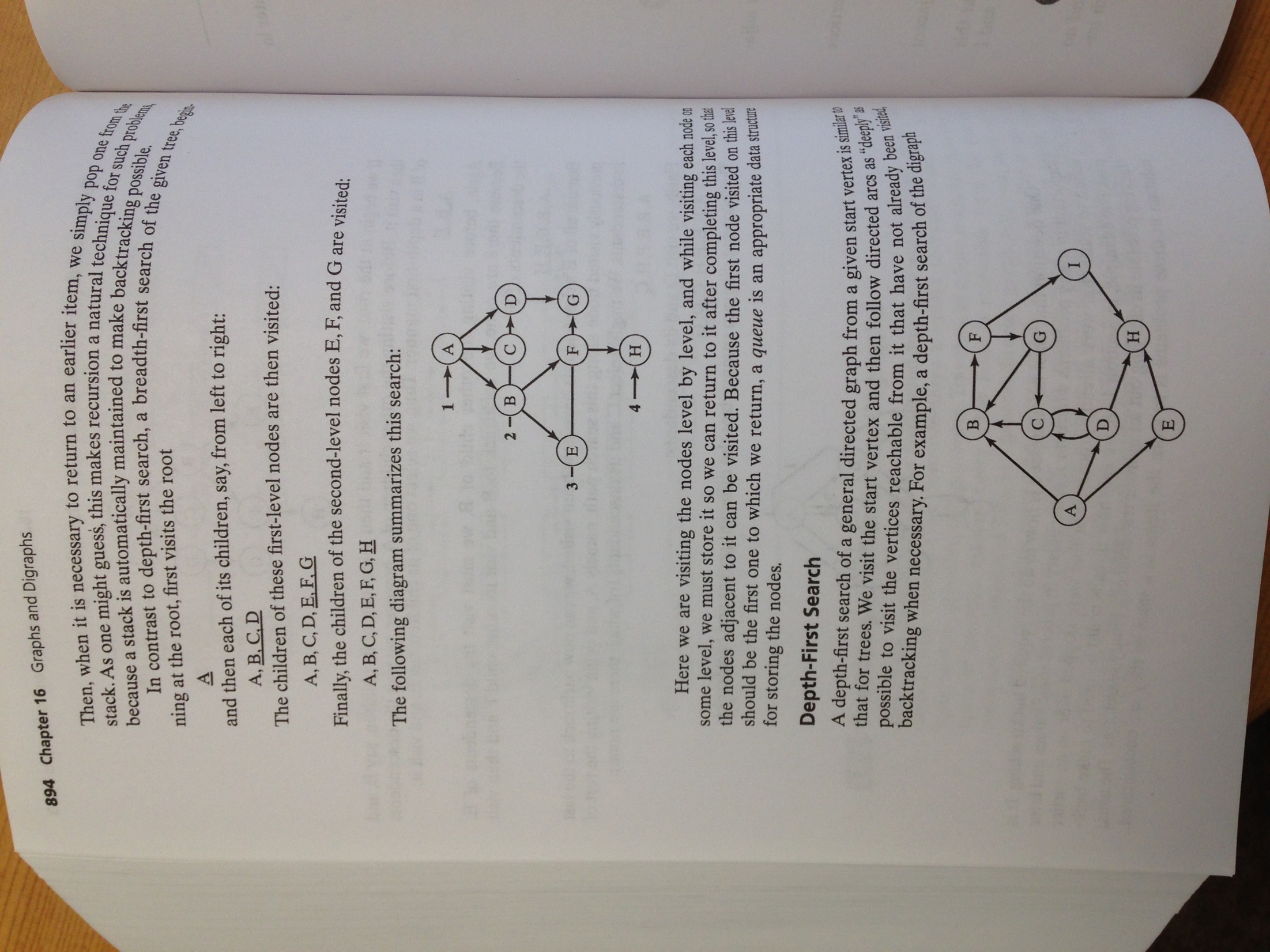
***Note: Here, vertex v1 is adjacent to v2 if there is a directed edge from v1 to v2.***

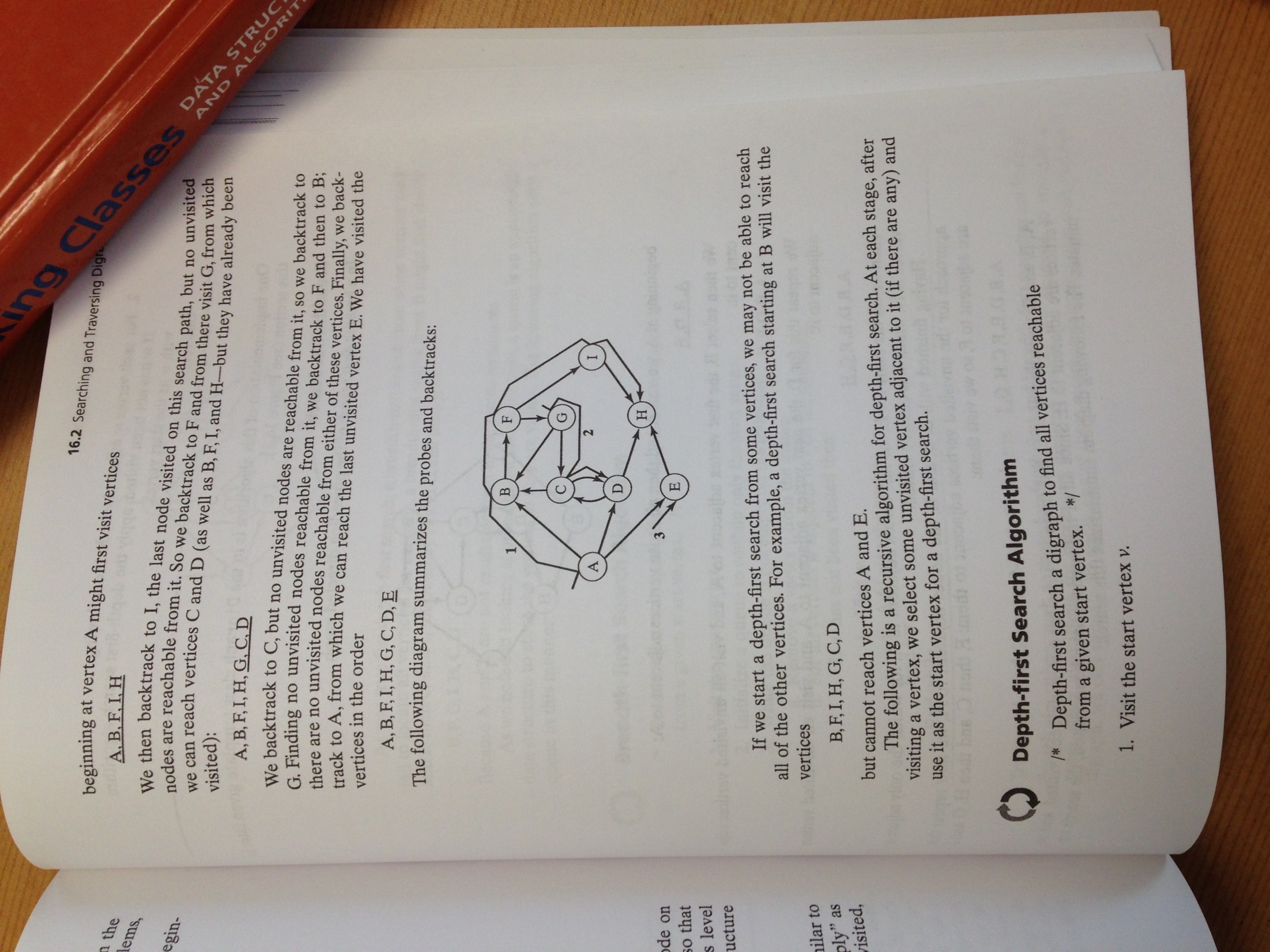
**DFS**

* do it on a tree first.
* then a graph

**BFS**







====cut====

Betsy’s notes

Searching and Traversing a DiGraph

* When talking about trees we talked about pre, in, and post order traversals
  + Always possible to traverse because each node is reachable from the root
  + Not true of digraph—there may not be a node from which every other vertex can be reached

Depth-First Search

* Generalization of preorder traversal
* Starting at some vertex v recursively traverse all vertices adjacent to v
* Need to be careful to avoid cycles—by marking each vertex as visited as we visit it

void Graph::dfs(Vertex v)

{

v.visited=true

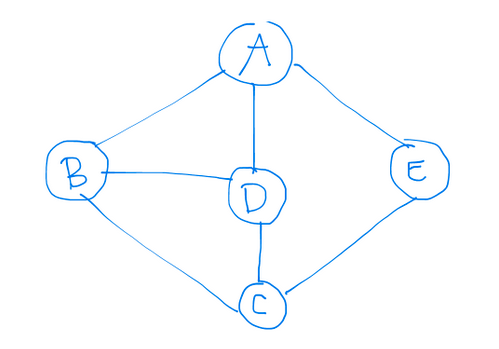
for each w adjacent to v

if !w.visited

dfs(w)

}

Example:



* We start at A
* Mark A as visited
* Call dfs(B)
* Mark B as visited
* call dfs(C)
* Mark C as visited
* Call dfs(D)
* Mark D as visited

Breadth-First Search

1. Visit the start vertex
2. Initialize a queue to contain only the start vertex
3. While queue is not empty:
   1. Remove vertex from queue
   2. For all vertices w adjacent to v do the following:
      1. if w has not been visited, then visit and add to queue

DiGraph Traversal: visit all nodes in a digraph

* Initialize a visited array to false

Shortest Path Algorithms (Review of Dijkstra)

You have a weighted graph—that is, associated with each edge (Vi, Vj) is a cost Ci,j to traverse the edge.

Thus, the weighted path length, or the cost of path V1V2...VN is .

(The unweighted path length is just the number of edges on the path, N-1)

Single source shortest-path problem:

Given as input a weighted graph, G=(V,E) and a distinguished vertex, s, find the shortest weighted