* Data Structures for Graphs

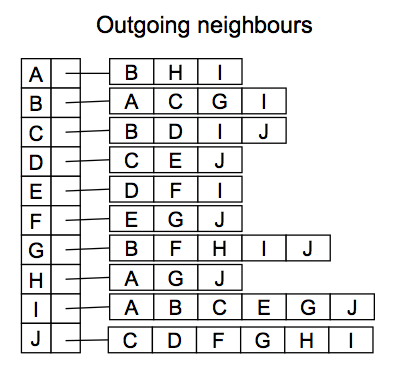
1. Traditional Adjacency Matrix
2. Traditional Adjacency List
3. Object-based structure (Adjacency list-like)

Object-based structure (Adjacency list-like)

* Tries
* Dijkstra’s Algorithm
* A\* Search
* Articulation Points
* Minimum Spanning Tree
* 3D Graphics

1. Draw an adjacency list representation for directed graph

Start

Goal

Adjacency List

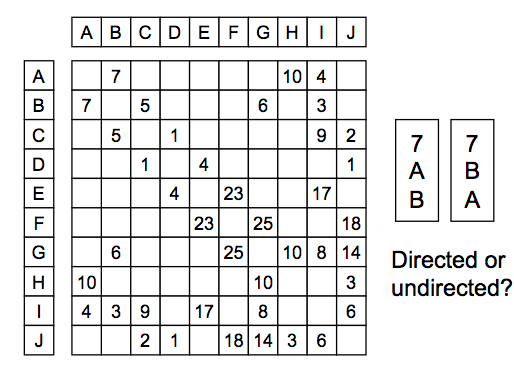
For each node, a list of outgoing neighbours

(directed or undirected?)

Assume only one edge between each pair of nodes,

Cannot handle multi-graph

Adjacency matrix



Matrix(2D array) shows the edges between nodes

-Blank means no edge

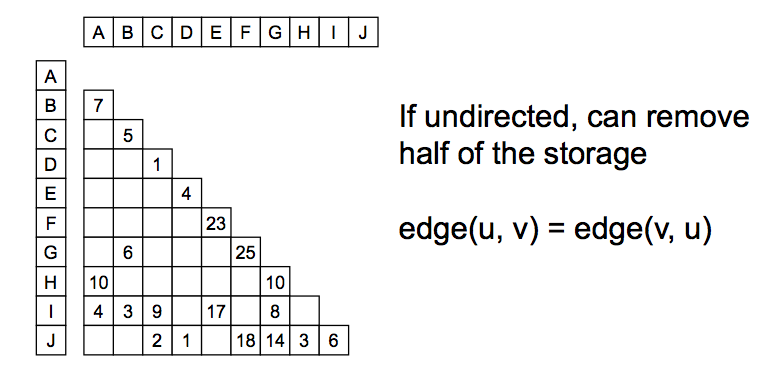
-fast: outgoing and incoming edges of a node

Assume only one edge between each pair of nodes,

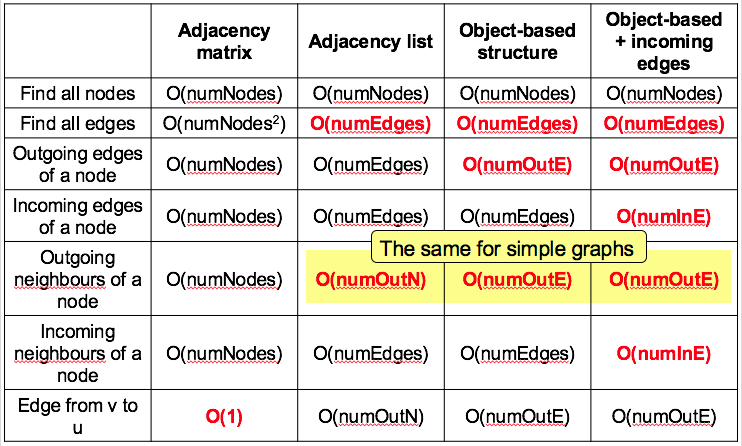
Cannot handle multi-graph

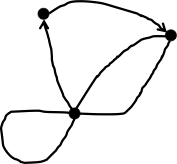
*Adjacency Matrix*

* Complexity of actions
  + Find all nodes
    - O(numNodes)
  + Find all edges
    - O(numNodes2)
  + Outgoing edges of a node
  + Incoming edges of a node
    - O(numNodes)
  + Outgoing neighbours of a node
  + Incoming neighbours of a node
  + Edge from vertex *v* to *u*?
    - O(1)
* If all edges are undirected, can remove more than half of the storage

*Adjacency List*

* Complexity of actions
  + Find all nodes
    - O(numNodes)
  + Find all edges
    - O(numEdges)
  + Outgoing edges of a node
    - O(numEdges)
  + Incoming edges of a node
    - O(numEdges)
  + Outgoing neighbours of a node
    - O(numOutNeighbours)
  + Incoming neighbours of a node
    - O(numEdges)
  + Edge from vertex *v* to *u*?
    - O(numOutNeighbours)



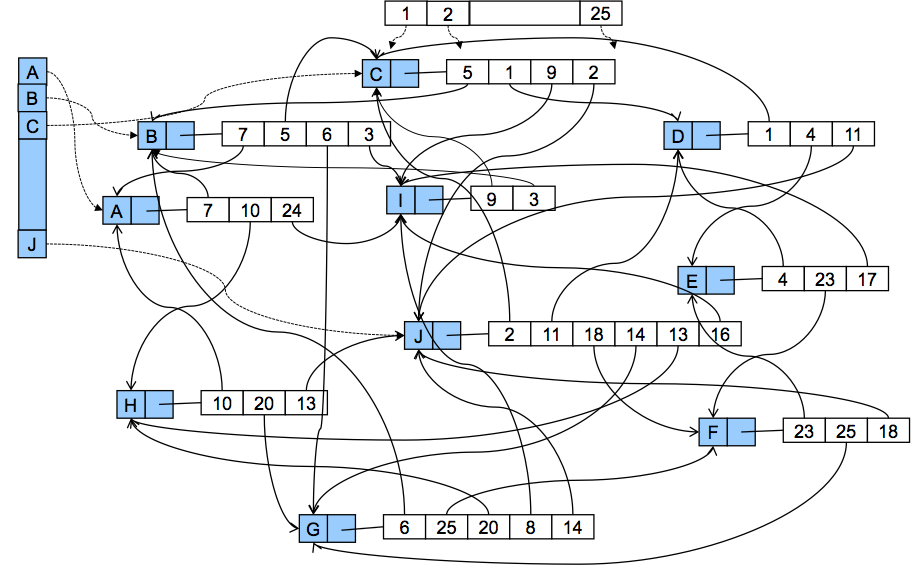
****

**Different types of graphs**

– Multi-graph (multi-lane roads)

– Directed edges (one-way streets)

– Loops

* Possible Data Structure:

– Graph = set of Node objects , Set<Node> nodes;  & set/bag of Edge objects Set<Edge> edges;

– Node = info about the node (label, etc)

– Edge = info about the edge and its two nodes  either two fields: node1, node2, or or Set<Node> (just 2 elements)

*Issue: when are two edges counted as equal?*

File • FileReader • BufferedReader

// Read file line by line File roadFile = new File(dataDirectory+"roadID-roadInfo.tab"); BufferedReader data = new BufferedReader(new FileReader(roadFile));

String line = data.readLine();

// Process each line using split method String[] values = line.split("\t");

int n = Integer.parseInt(values[0]);

double d = Double.parseDouble(values[1]);

BufferedReader bufferedReader;  
try {  
 bufferedReader = new BufferedReader(new FileReader(nodeFile));  
 String line = bufferedReader.readLine();  
 while (line != null) {  
 //pass the whole line information of the node to the node object, which constructor can sort the information properly  
 Node node = new Node(line);  
 //node is indexed by its ID  
 nodeMap.put(node.nodeId, node);  
 line = bufferedReader.readLine(); // next line for next literation  
 }  
 bufferedReader.close();  
} catch (FileNotFoundException e) {  
 System.*out*.println(nodeFile + "File has not been found");  
} catch (IOException e) {  
 System.*out*.println("IO Exception");  
}

public Node(String string) {

String[] data = string.split("\t");

nodeId = Integer.parseInt(data[0]);

double lat = Double.parseDouble(data[1]);

double lon = Double.parseDouble(data[2]);

// transfer lat-lone --> Location Object

location = Location.newFromLatLon(lat, lon);

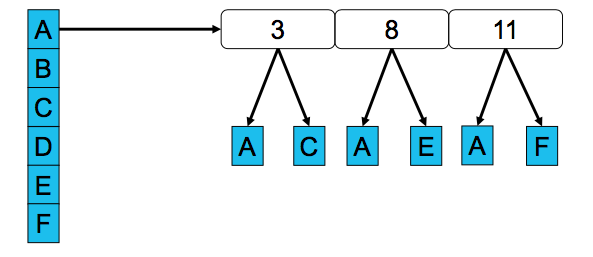
color = DEFAULT\_COLOR; //Initialise color

linkedSegments = new HashSet<>();

neighbourNodeSet = new HashSet<>();

this.count = MAX\_COUNT;

}

 Modern” Data Structures for Graphs

* Keep a set of nodes
* Each node has a collection of outgoing edges
* Each edge has a node on the other side  – Contain a destination node – Might contain the source node as well

*A General Graph Search Strategy*

* + Start a “fringe" with one node
  + Repeat until fringe is empty or other criteria is met:
    - Choose a fringe node to visit
    - Add its neighbours to fringe

Used by both Dijkstra’s Algorithm and A\* Search

*Dijkstra’s Algorithm*

* Find the shortest paths from a start node to all other nodes
* Early stop if the goal node is given
* Use a priority queue
  + Priority: costFromStart
  + Always select the unvisited node with minimal costFromStart
* Given: a *graph* with weighted edges.
* Initialise *fringe* to be a set containing *start* node  
   *start.pathlength* ← 0
* **For each** other *node*, set *node.pathlength* ← *infinity*
* **Repeat until** all nodes have been visited:
  + Choose an unvisited *node* from the *fringe* with minimum *path length* (i.e. length from *start* to *node*)
  + Record the *path* to the current *node*
  + **For each** unvisited *neighbour* of the current *node*
    - Add the *neighbour* to *fringe*
  + Mark the current *node* as visited

*A\* Search*

* Find the shortest path from a start node to a goal node
* Use a priority queue
  + Priority: estimated total cost = costFromStart + estCostToGoal
  + Use heuristic to estimate estCostToGoal
  + Admissible and Consistent heuristic
  + Always select the unvisited node with minimal estTotalCost
* What if the heuristic is not admissible/not consistent?

Dijkstra’s Algorithm is for finding shortest paths from the start node to all other nodes

* If goal node is given, then it can be made faster:

Dijkstra’s Algorithm + early stop (when goal node is visited)

In Dijkstra’s Algorithm, nodes are prioritised by the cost from start to the node (costFromStart)

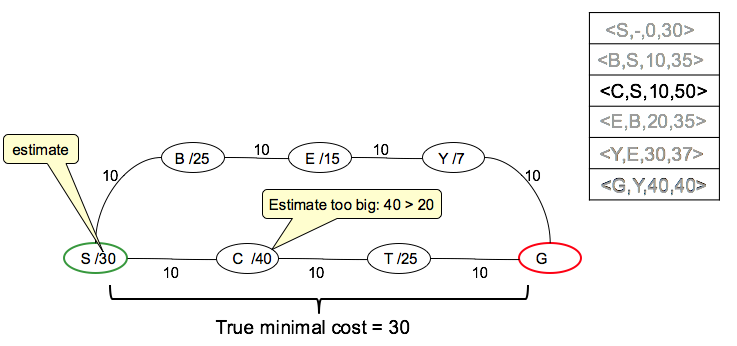
-Priority in the fringe = costFromStart

* In A\* search, priority in the fringe = totalCost

(Estimated) totalCost = costFromStart + (estimated) costToGoal

do better by including the (estimated) cost from the node to the goal as well (costToGoal)

***Admissible Heuristic***

* A heuristic estimate is admissible if it never overestimates the true minimal cost to goal
* If not, then may not find the shortest path

***Consistent/Monotonic Heuristic(一致/单调)***

* A heuristic is consistent/monotonic if for any node and its neighbor
* *For each node, estimated totalCost(node) is no greater than estimated totalCost(neigh node) of any of its neighbours*
* *Otherwise, may not find shortest path without revisiting*

*----------------------------------------------------------------------------------------------------------------------------*

* *estimate(node, goal) <= cost(node, neigh) + estimate(neigh, goal)*

--------------------------------------------------------------------------------------------------------------

* *costFromStart(node) + estimate(node,goal) <=*  *costFromStart(node) + cost(node, neigh) + estimate(neigh, goal)*
* *totalCost(node) <= totalCost(neigh)*

------------------------------------------------------------------------------------------------------------------------------

*estimate(node, goal) – estimate(neigh, goal) <= cost(node, neigh)*

Can A\* Search or variation of it be run on a graph with a non-consistent heuristic?

If it can be, then *what modification would you need to make to the fast A\* Search as used in the lectures*?

1. Nodes should be added to the fringe
2. even if they have been visited
3. rather we should keep “the best path found so far” for each node
4. and how we got it.
5. The algorithm will be slow
6. because we need to keep running it
7. until no better nodes can be added to the fridge
8. and we process them all
9. even if visiting node more than once.

Can A\* Search or variation of it be run on a graph with a non-admissible and

non-consistent heuristic? If not, then what will go wrong?

1. If it is non-admissible
2. then the search will dismiss the correct shorter paths
3. because the estimate is over estimating the distance
4. and find a non- optimal (not shortest) path as a result.

*Minimum Spanning Tree*

A spanning tree of a **connected, undirected** graph, is a **subgraph that contains all the nodes** but is **a tree** (no cycles).

* What is a MST?

Minimum spanning tree of a weighted connected, undirected graph is a spanning tree with total weight no greater than the total weight of any other spanning tree.

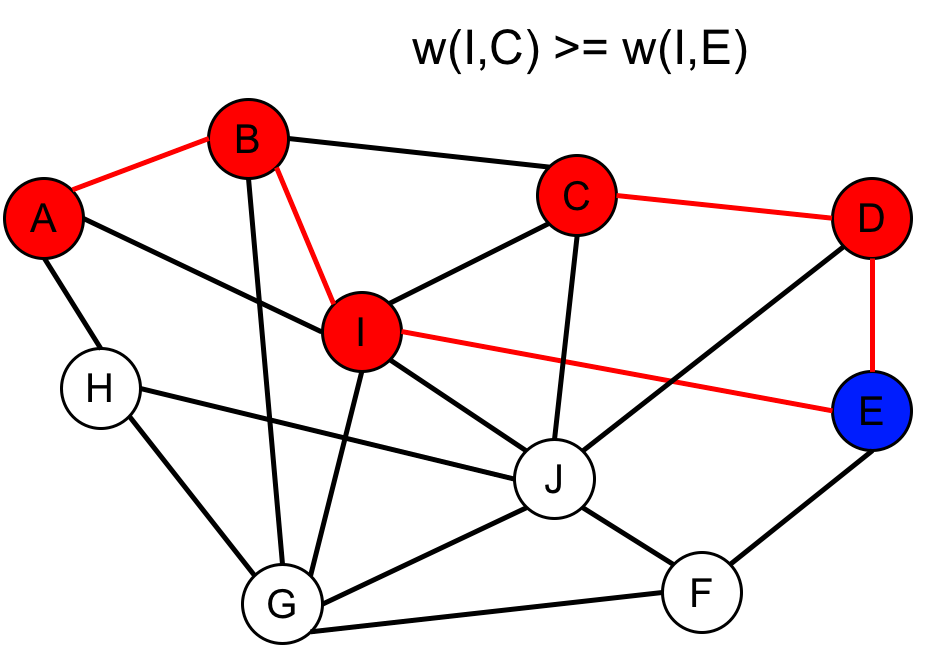
May not be the shortest paths

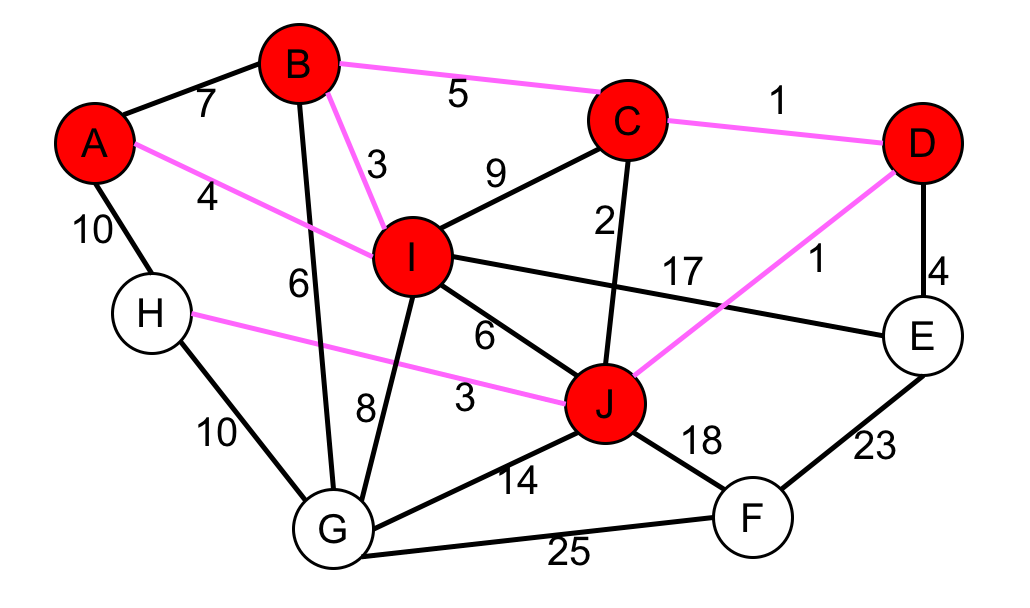
App: Find cheapest way to connect a set of towns/substations/cell towers with power/communication lines.

* Prim’s Algorithm

Idea: Grow the spanning tree from a seed node, always choosing the lowest weight edge to an unconnected node.

1. Grow from a start node
2. At each step, the min-weight edge that connects a node in the tree and a node not in the tree
3. Stop when all the nodes in the tree



* 
* Given: a graph with weighted edges.

Initialise *fringe* to have a start node with *costToTree* = 0 and a dummy edge, all nodes are unvisited

Repeat until all nodes are visited:

Choose from *fringe* the unvisited node with minimum *costToTree*

Add the corresponding edge to the spanning tree, set node visited

For the node at the other end of the edge:

Add the unvisited neighbours of the node to the *fringe*

* Prim’s algorithm is a best first search (rather than depth first)
* Kruskal’s Algorithm

Alternative algorithm for minimum spanning trees:

Idea: Connect small trees together, always choosing the lowest weight edge.

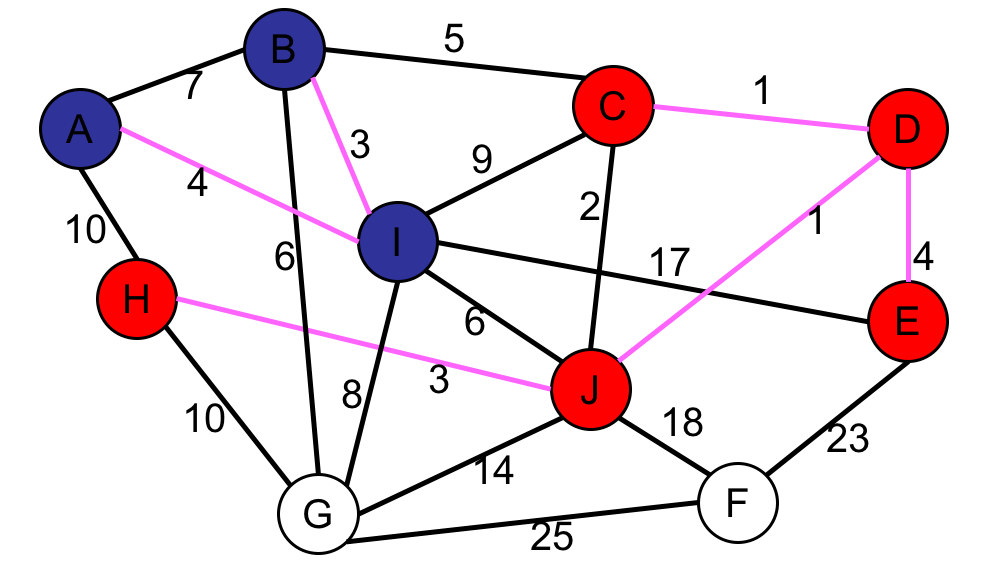
*Given: a graph with weighted edges.*

Initialise *forest* to be a set of trees, each containing one node

**Repeat until** *forest* contains only one tree:

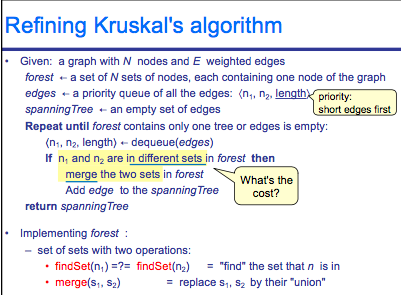
Choose a minimum weight edge that connects two trees in *forest*

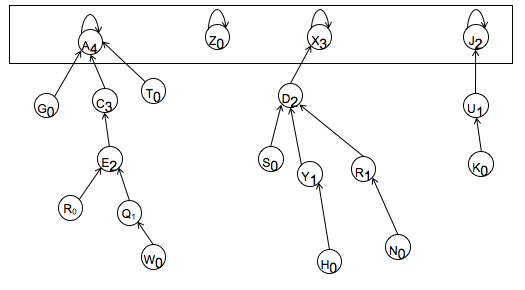
Add the edge to the spanning tree and combine the two trees



* Union-Find Data Structure

1. Start from a lot of single-node trees (forest)
2. At each step, merge two trees into one tree by adding a min-weight edge
3. Stop when there is a single tree





1: forest = set of sets of nodes:

– findSet(n1)

• iterate through all sets, calling s.contains(n1)

– merge (s1 , s2)

• add each element of s1 to s2 and remove s1

2: forest = mark each node with ID of its set

– findSet(n1):

• look up n1.setID

– merge(s1 , s2)

• iterate through all nodes, changing IDs of nodes in s1

3: forest: set of inverted trees of nodes:

• Each set represented by a linked tree with links pointing

***towards*** the root

• The nodes in these trees *are* the nodes of the graph

MakeSet(x):

x.parent ← x

add x to set

Find(x)

**if** x.parent = x

return x

**else**

root ← Find(x.parent)

return root

Union(x, y):

xroot ← Find(x)

yroot ← Find(y)

**if** xroot = yroot

**return**

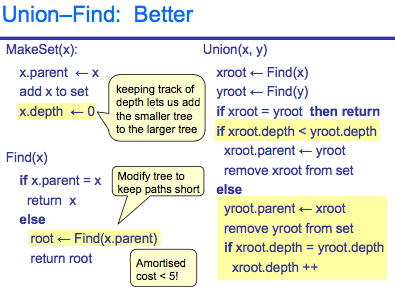
**else**

xroot.parent ← yroot

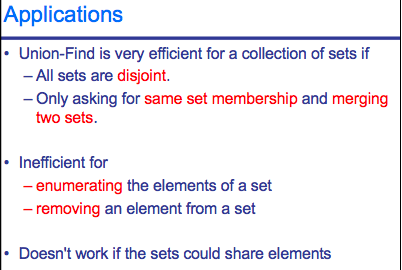
remove xroot from set.

Union(x,y) = Union(y,x)?

**Order will affect tree depth (search complexity)**

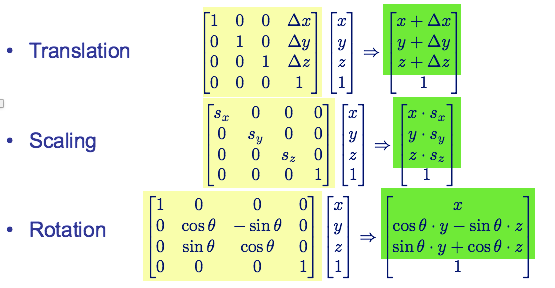


1. How to never merge longer tree into shorter tree?



*3D Graphics*

1. Rendering 3D images
2. Polygons
3. Coordinate system
4. Polygon Transformation



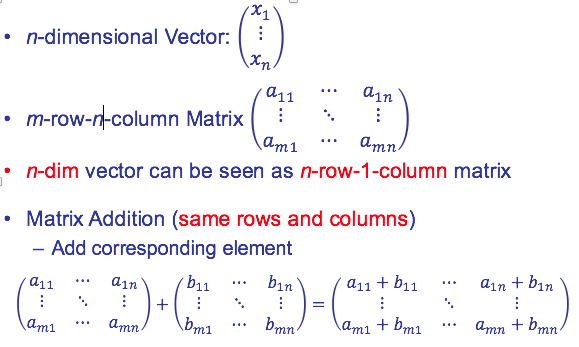
How to represent a polygon?

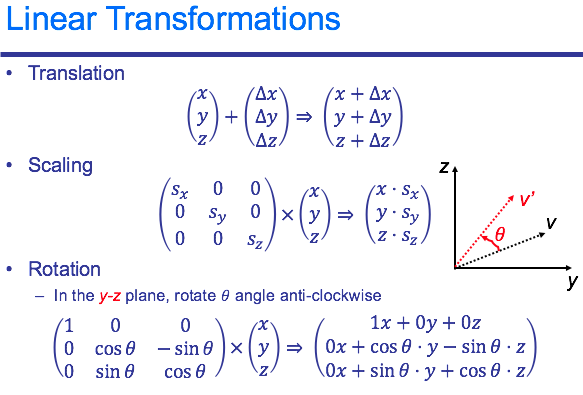
Ordered list of vertices

What the viewer would see based on?

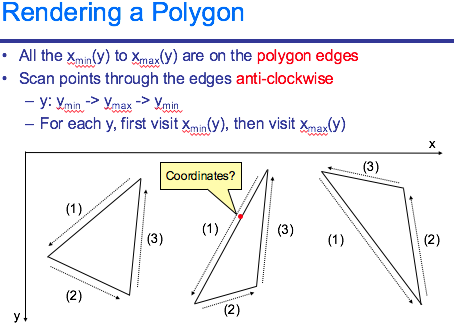
Locations of screen, object, light source, Angles …

*Linear Algebra Basics*

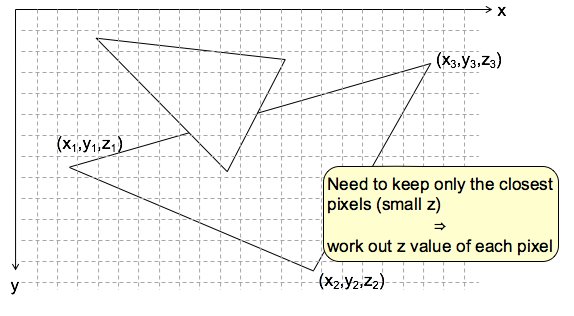




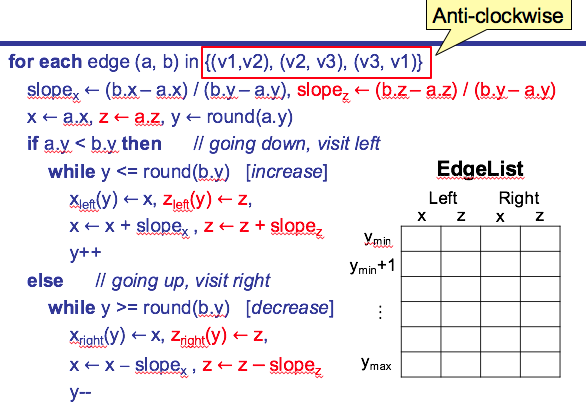
*Rendering a Polygon*

* 
* Calculate EdgeList

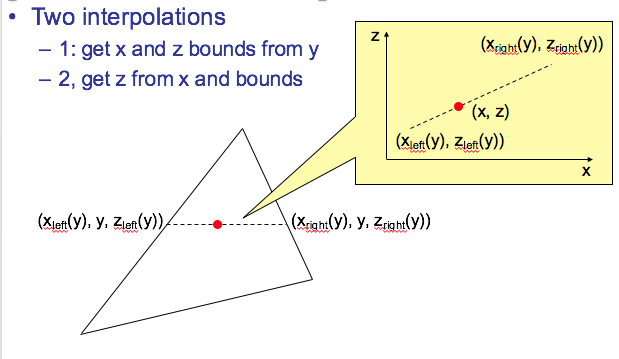
TODO1



*EdgeList with Z-buffer*



*Use EdgeList*



x,y,z converts to/from x/w,y/w,z/w,w

State at least two advantages of 4D homogeneous coordinates over standard 3D representation for polygons.

1. Easy to deal with 0,0,0.???

2. Easy to combine multiple translations to- gether into one 4D matrix.

The normal is used for which of these purposes?

1. testing if a polygon is backfacing b) shading c) computing depth d) determining if two polygons overlap in the edgelist

In the Z-buffer algorithm, the Z (depth) test is done

a) once per object b) once per polygon c) once per pixel d) The advantage of this algorithm is that it does not need to test depth

If the ”handedness” (right-hand versus left-hand convention) is inconsistent in various parts of your program, which of these might happen:

a) you see the inside or back side of objects b) objects are a reflected or mirror image of what they should be c) the Z-buffer test will fail

Suppose that the particular image being rendered does not contain overlapping objects.

In other words, there are no objects in the 3D scene that are in front of and hiding other objects

when viewed from the camera position. Which would be a correct characterization of the complexity of the rendering algorithm described in class:

a) it is linear in the number of pixels b) it is *O*(*N*2) in the number of polygons c) it is *O*(*N* · log *N*) in the number of polygons

d) the computation time cannot be characterised in this way

Which of these is a reasonable diffuse shading model? In these descriptions, ∗ is regular multiplication, · is the dot product, and × is the cross product.

a) color = reflectance + normal · light + ambient

b) color = reflectance + normal × light + ambient

c) color = reflectance \* (normal · light + ambient)

d) color = (reflectance + normal × light) \* ambient

TODO

TODO

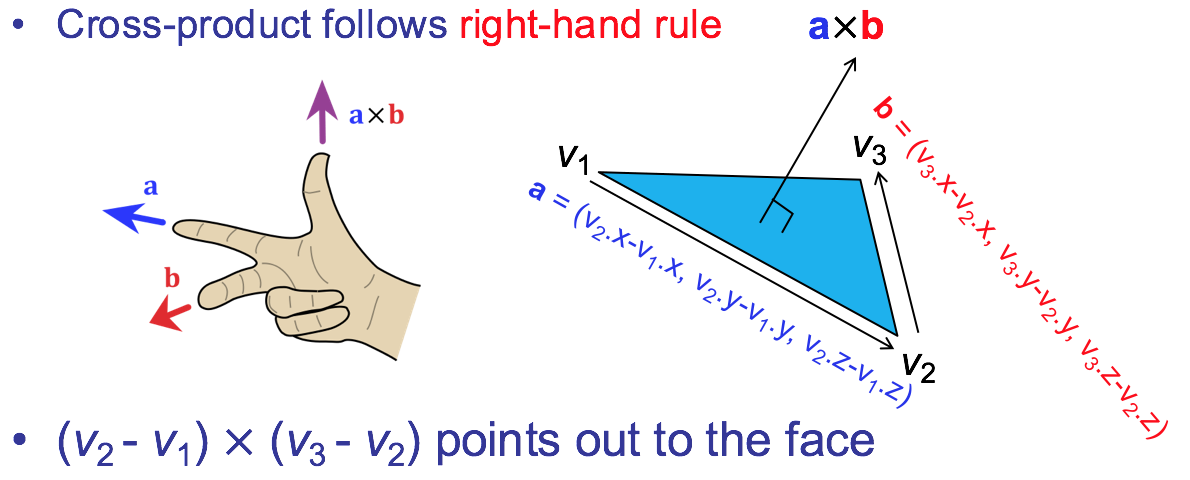
The following pseudocode gives the algorithm for processing triangles into the Z- buffer. However, one very important step is missing.

*Remove Hidden Polygons*

1. Assume viewer looks along z-axis, by Rotate and translate object to make desired view
2. Remove polygons **facing away** from viewer

TODO

* Check whether a polygon is facing away
  1. Order the vertices in a proper way
* Anti-clockwise when facing the viewer



A polygon is facing away from the viewer if the z-component of (*v*2 - *v*1) (*v*3 - *v*2) is **positive**

(*v*2 - *v*1) (*v*3 - *v*2) is called the normal of the polygon

* Store in the file and data structure
  1. Use the cross-product



*Shading Computation*

color = reflectance \* (normal · light + ambient)

Assume uniform reflectance for red, green, blue

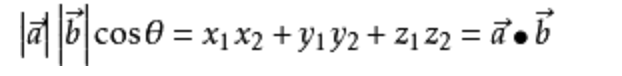
– Assume some ambient light

light ← reflectance x ambient light intensity ([0, 1])

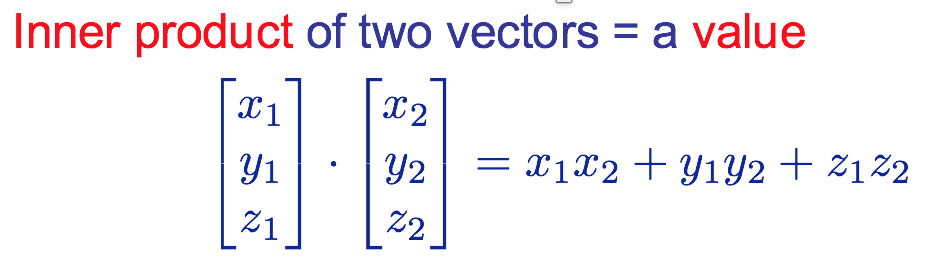
– Diffuse reflection depends on light source direction

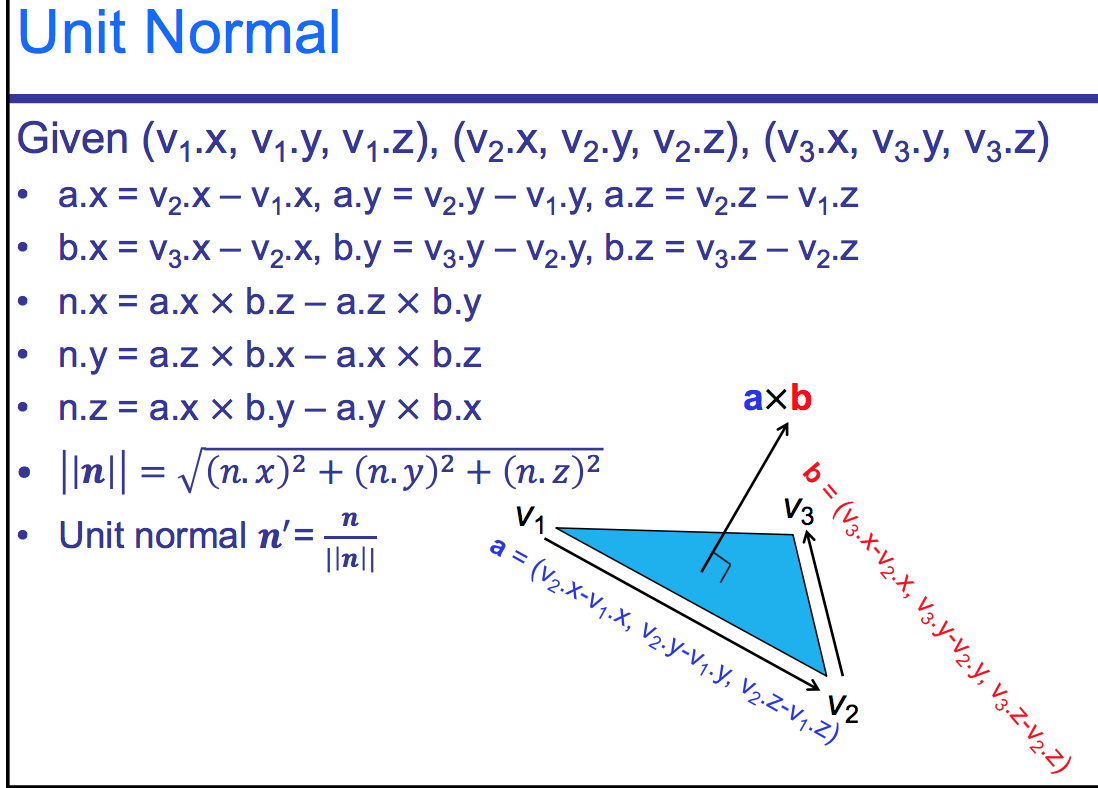
light ← reflectance x light intensity x cos(θ )

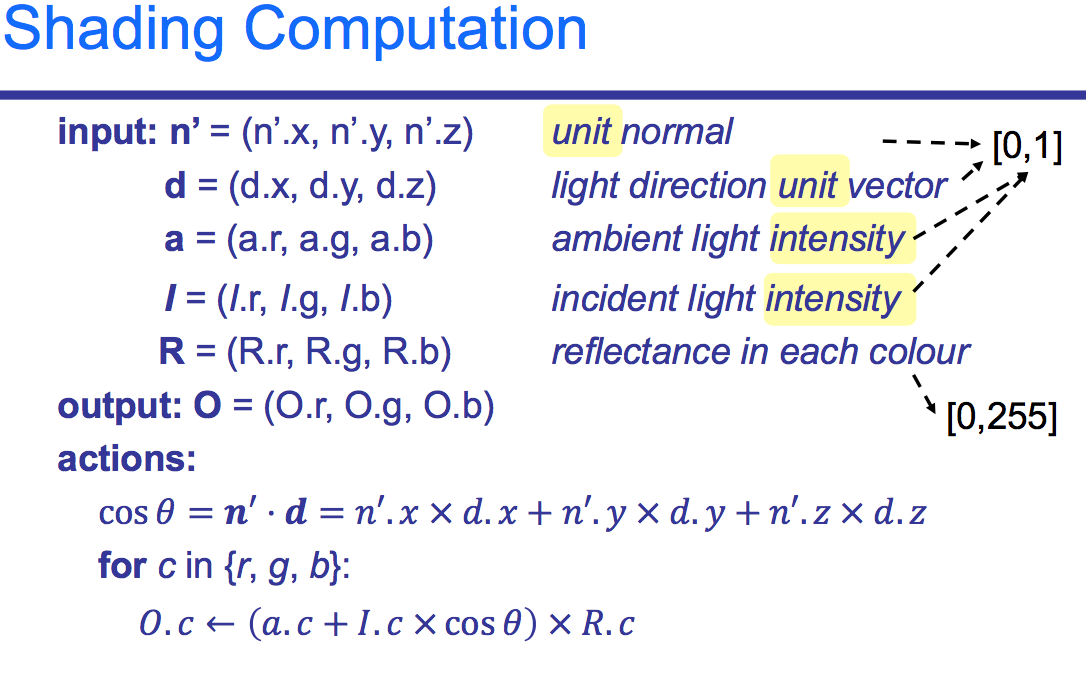
cos(θ ) = normal ⋅ lightdirection (if both unit vectors: length 1)

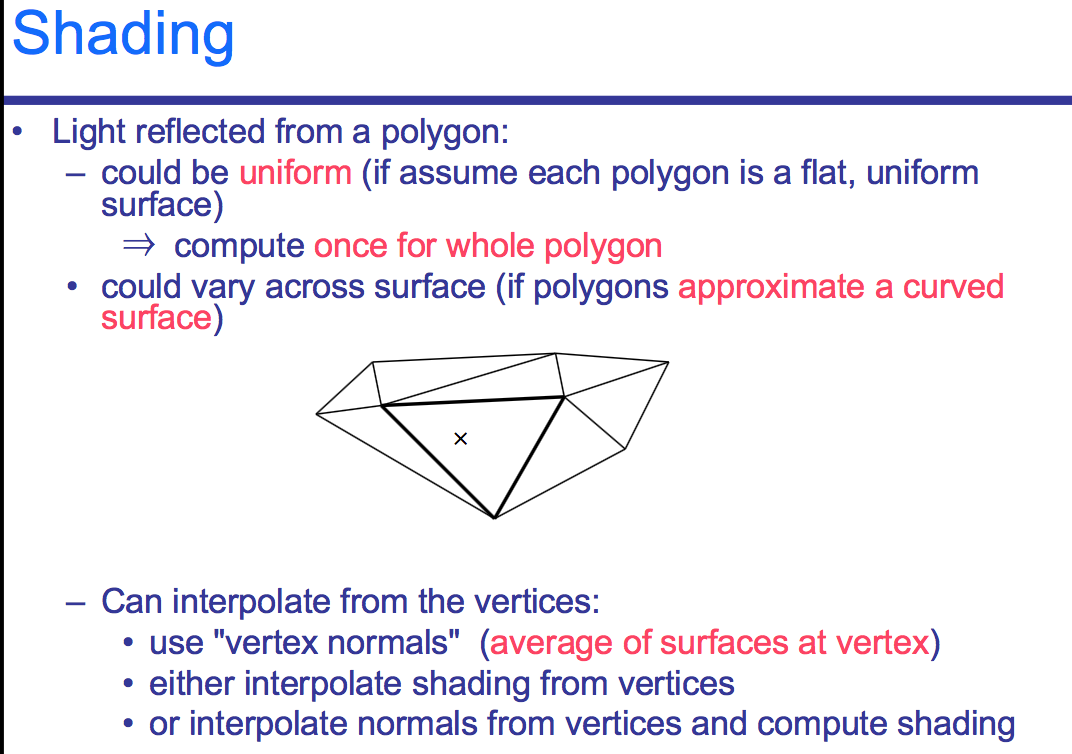


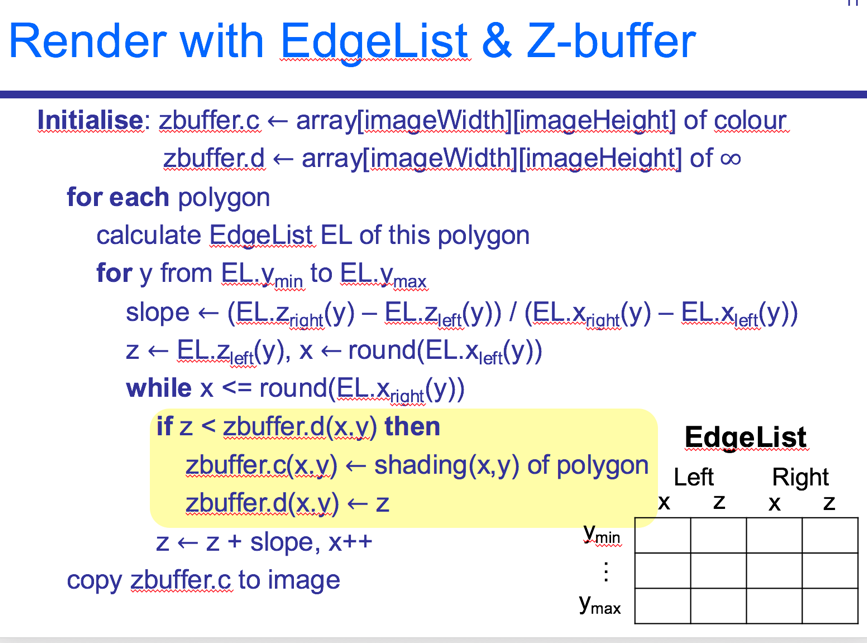
*Get unit normal*

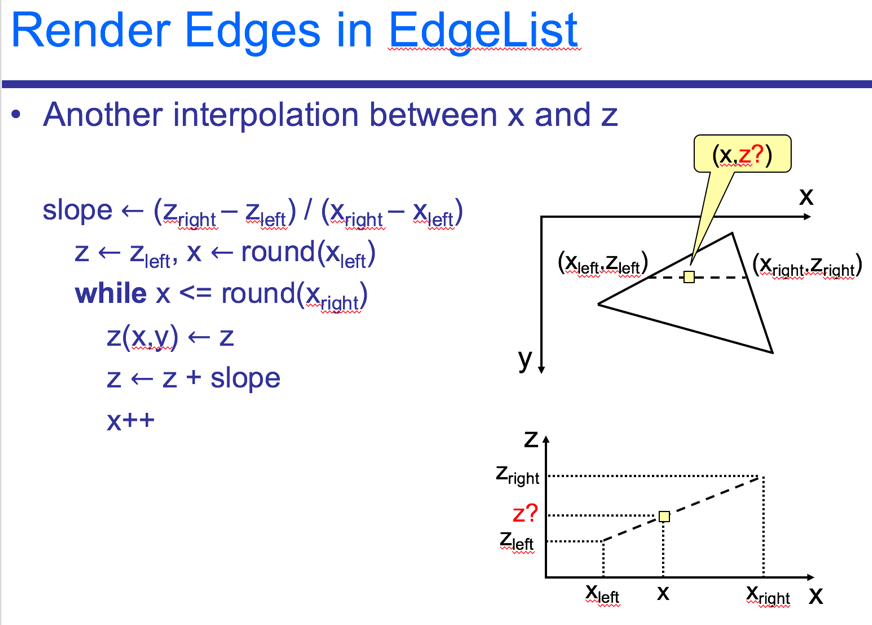




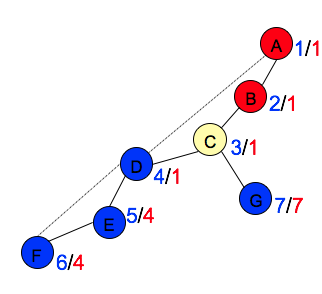


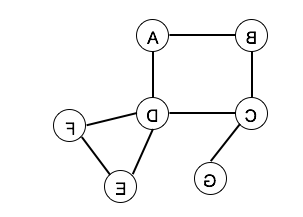


****

****

*Find Articulation Points*

* DFS of the graph, set node count number
* For each node, split other nodes into
  + Subtree nodes
  + Non-subtree nodes
* Whether there is an alternative path (edges in the graph but not in the tree) from each subtree node to the non-subtree nodes?
* Min count number each node can reach back to?
* Articulation point if a subtree node has reach back no smaller than its count number
* Root node



Initialise : **for** each node: node.depth = infinite , articulationPoints = { }

start .depth = 0, numSubtrees = 0

**for** each neighbour of start

**if** neighbour.depth == infinite then

recArtPts( neighbour, 1, start )

numSubtrees ++

**if** numSubtrees > 1 then add start to articulationPoints

recArtPts(node, depth, fromNode):

node.depth = depth, reachBack = depth,

**for** each neighbour of node other than fromNode

**if** neighbour.depth < infinite then

reachBack = min(neighbour.depth, reachBack)

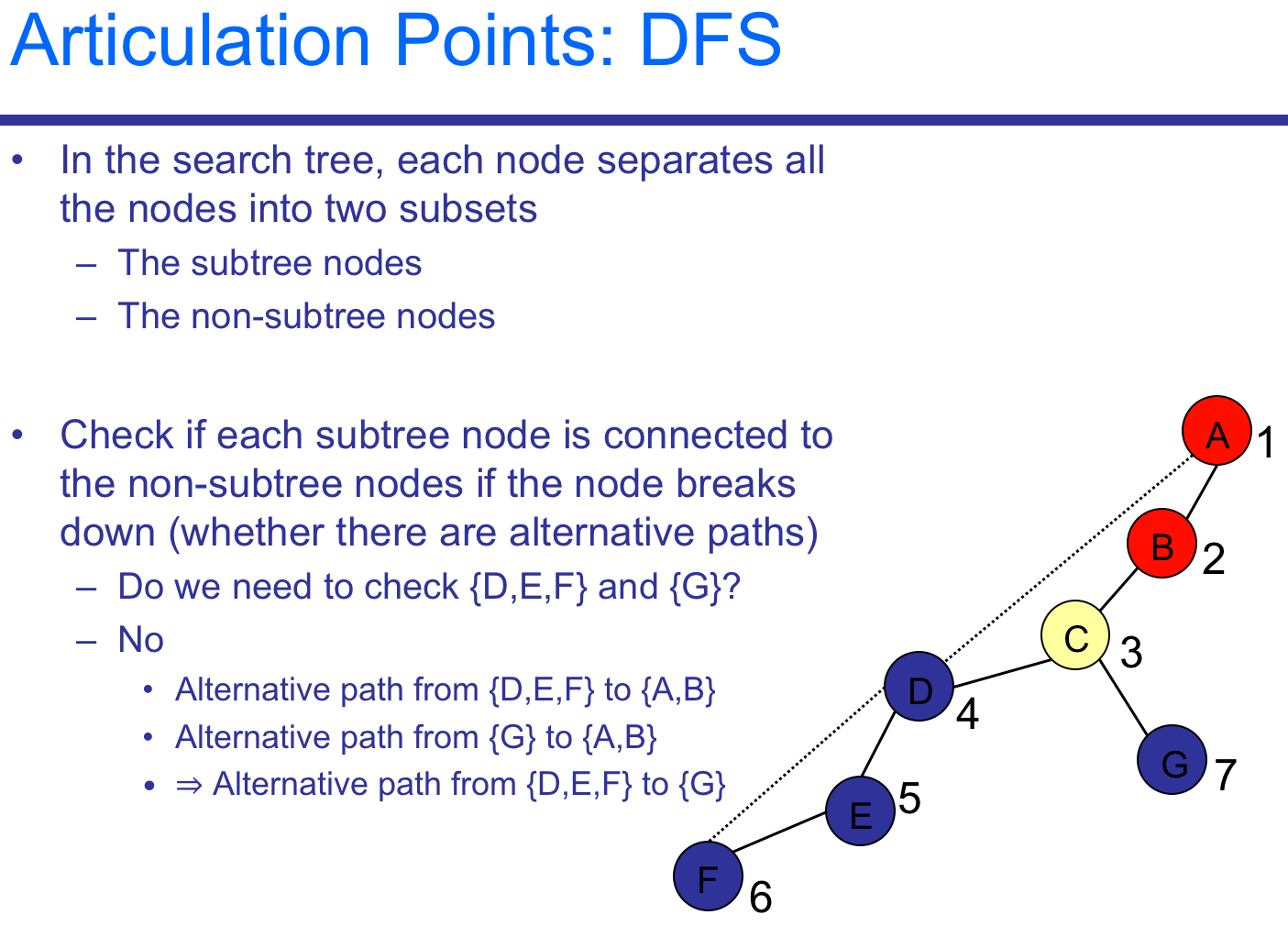
**else**

childReach = recArtPts(neighbour, depth +1, node)

reachBack = min(childReach, reachBack )

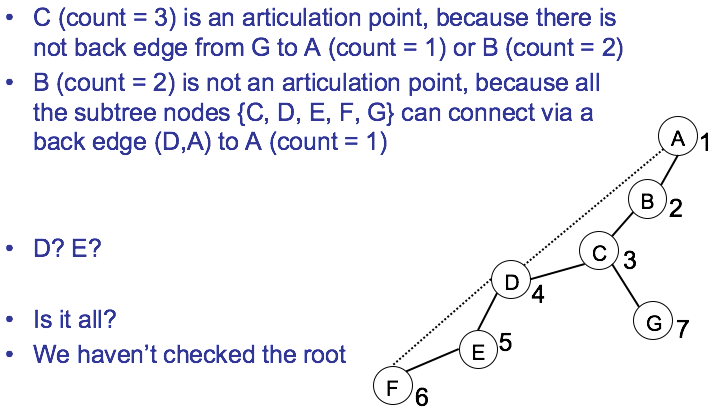
**if** childReach >= depth then add node to articulationPoints

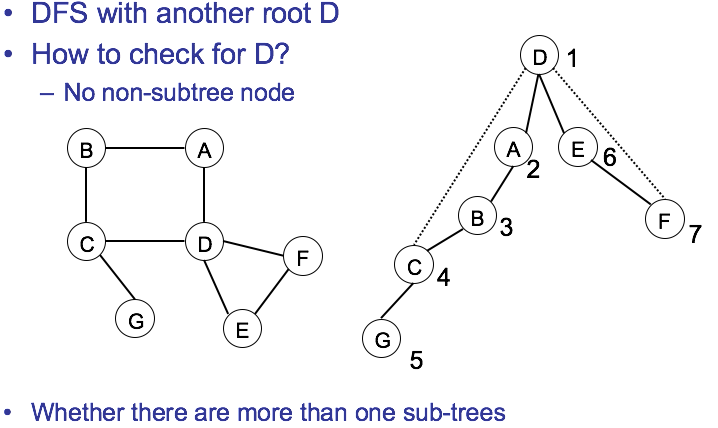
**return** reachBack



* A node is an articulation point if there exists a subtree node that is connected to non-subtree nodes only through the node, i.e. there is not alternative path
* An alternative path uses the edges that are in the graph but not in the search tree (back edges)
* How to check that the back edge connects to non-subtree node?
  + The other end-node has a smaller count number
  + All the nodes with smaller count numbers are non-subtree nodes

Articulation Points: DFS

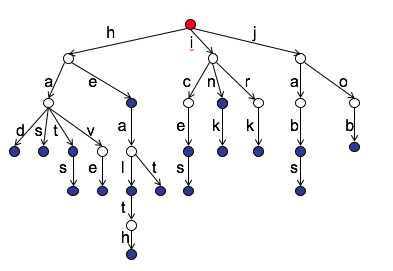




* Key ideas of algorithm:
  + Record count number of nodes as you search
  + From each recursive search of a subtree, return the minimum count number that the subtree nodes can "reach back" to.
  + Compare the "reach back" of each subtree to count number of this node, if smaller then there is an alternative path

*Tries*

* *Tree: set of strings/keys (if map string → value )* 
  + children indexed by elements of the key
  + nodes corresponding to complete key are marked ( contain a value)
  + tree under a node = set of all keys with the prefix so far**.**

****

had

has

hat

hats

have

he

heal

health

heat

ice

iced

in

ink

irk

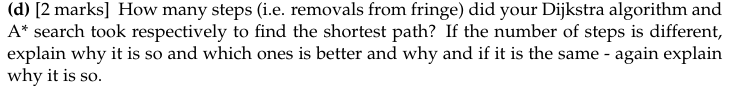
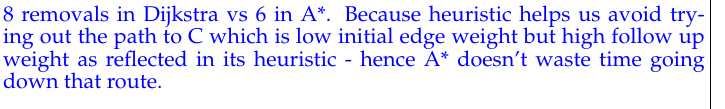
iron

jab

jabs

job

* Each node contains
  + marker (if set) or associated value (if hashmap)
  + a map: char → node
* label the edges, not the nodes mark the end of a word
* Tries are Commonly Used For:
  + –  Storing large dictionaries in minimal space, but fast access time
  + –  Doesn’t need to store common prefixes multiple times.
  + –  Allows auto completion, spelling correction
  + –  Can store numbers (use the bits as the index elements)🡪binary trie
* How to add a word into the trie?
* How to get a given word from the trie?
* How to get all words that match a given prefix from the trie?

 (c) [5 marks] The above two are known as greedy algorithms. What makes an algorithm

greedy? State another greedy algorithm you learned in COMP 261. Describe what it is and

explain what makes it greedy.

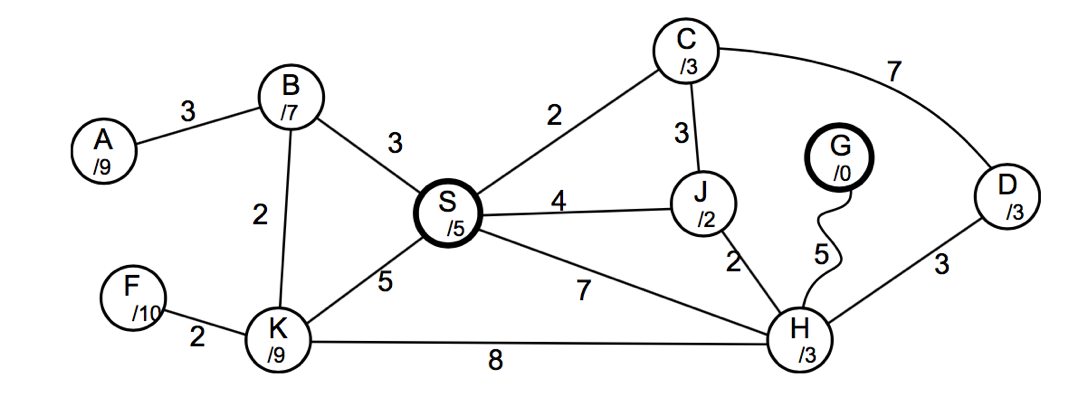
Dijkstra’s single source the shortest path Algorithm

Dijkstra SSSP algorithm that finds a shortest path as in the start of this test!

Greedy algorithms never backtrack and make a simple to compute choice

at every step that happens to be correct. Dijkstra picks the shortest edge

on the fringe.



J would be put on the queue (from S) with an estimated total cost of 13, and H would be put on the queue with a cost of 10. The algorithm would therefore visit H (from S, estimate 10) before visiting J (estimate 13), and would then put G on the queue with a cost of 11.

It would then visit G (from H, cost 12) before taking J off the queue. Therefore, it would return the path S-H-G, rather than the (better) path through J.