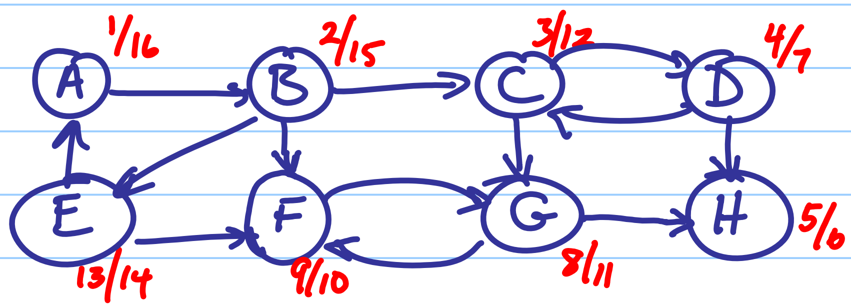
Strongly Connected Components

Note Title 12/3/2007

We define "strongly connected" to mean that for every pair of vertices (u,v) in the component, there is a path from u to v and from v to u.

In the following graph, what are the strongly connected components?

Find strongly connected components in graph G:



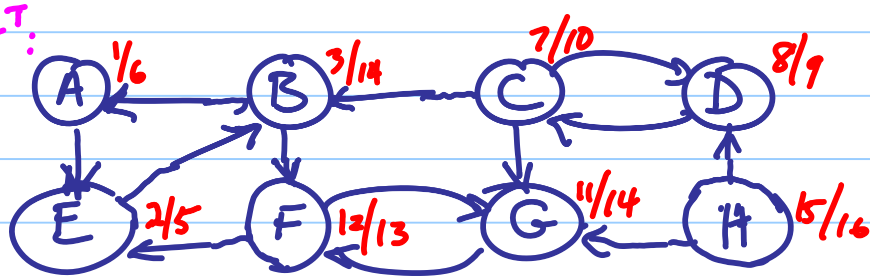
We will use finish time to determine where to resume in G^T

Step 1: Perform DFS on G

Find Strongly Connected Components (cont.)

Step 2: Perform DFS on G^T where when given a choice, choose vertex with largest finish value.

Graph G^T:



G^T must have arrows going opposite way

- Every time you reach a dead end, you finish one strongly connected component + start next.

Dikstra's Algorithm for Shortest Path on a Directed, Weighted Graph

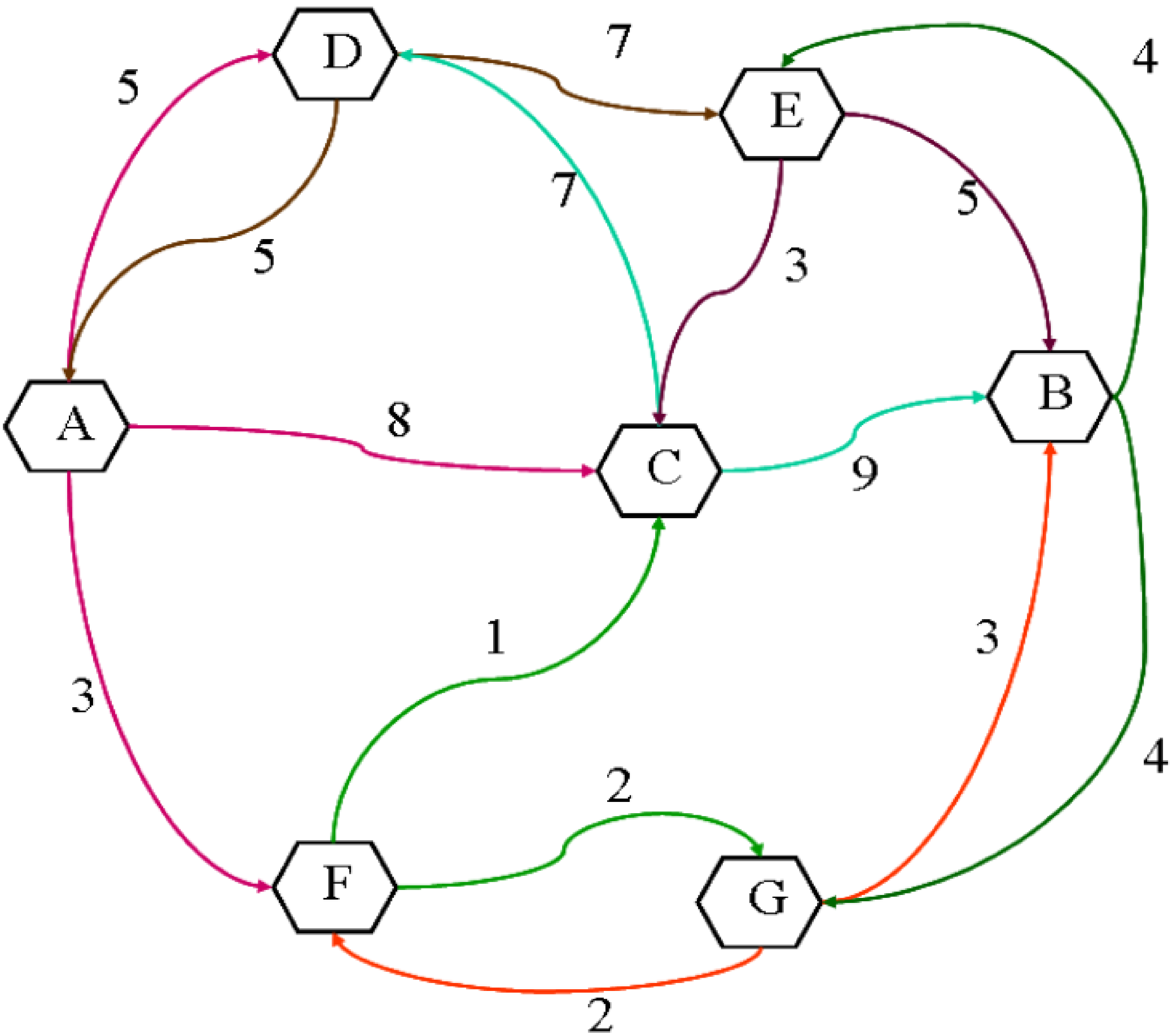
This is a GREEDY algorithm.

```
//finds shortest path between start and all other vertices

Initialize a predecessor array for vertices to all null
 Initialize a cost array which represents cost to start to all ∞
 Set the cost of start to itself as 0

 Q=all vertices in V
 while Q is not empty {
    u = remove the vertex which has the lowest cost from start
    for each vertex v which is adjacent to u {
       if (cost from start to v) > (cost from start to u + cost of u to v)
       then {
                  update the cost from start to v
                  mark u as the predecessor of v
         }
    }
 }
}
```

- The cost of the shortest path to any destination is known.

- The path to this destination can be reverse engineered by starting at the destination, and going backwards based on the predecessor list until reaching the starting point

Find the Shortest Path



Dijkstra Run Time

Note Title 12/5/2007

- Let's think about the while loop:
  - It executes exactly |V| times.
  - What are the costly things and how much do they cost?
    - Removing the vertex with the lowest cost from the starting point. Is this a fixed cost or does it depend upon the properties of the graph?
- Let's think about the for loop:
  - It executes exactly |E| times over the entire course of the search.
  - What are the costly things and how much do they cost?
    - Looking up costs, comparing values, storing new costs. Is this a fixed cost or does it depend upon the properties of the graph?

Minimum Spanning Tree

- Assume you want to run cables to connect n locations to each other using existing tunnels and pipes, and you want to do this using the least amount of fibre.

- You can view the locations as vertices and the physical distances between each pair through the different existing conduits as a weighted edge on a complete graph.

- This could easily be adjusted to allow the edge weight to be a combined cost that included the fibre cost as well as the costs for the installation/leasing/etc. within the existing space.

Doing Fast MST

- There are two greedy algorithms to do this "fast":
  - Kruskal's which is O(|E|log|V|)
  - Prim's which is either
    - O(|E|log|V|) using a heap
    - O(|E|+Vlog|V|) using a Fibonacci Heap

- Notice that again we have two variables to consider; the number of vertices and the number of edges.

- Both of these algorithms use the same basic greedy algorithm at a high level, but they utilize different approaches and data structures in their implementations.

"Growing" a MST

MST_edges = { }
while (MST_edges doesn't include every
vertex OR isn't a connected graph yet)
do
find an edge to grow the current MST set and
add that edge to MST_edges

- Finding the next edge to use is the challenging part of this.
  - Need to find an edge that belongs in the MST.
  - Don't necessarily need to add edges in an order that makes the tree grow "from the root".

Finding an Edge to Add

Definitions

A cut (S,V-S) of an undirected graph G(V,E) is a partition of V.

(u,v)∈E crosses the cut if one of the two endpoints is in S and the other is in V-S.

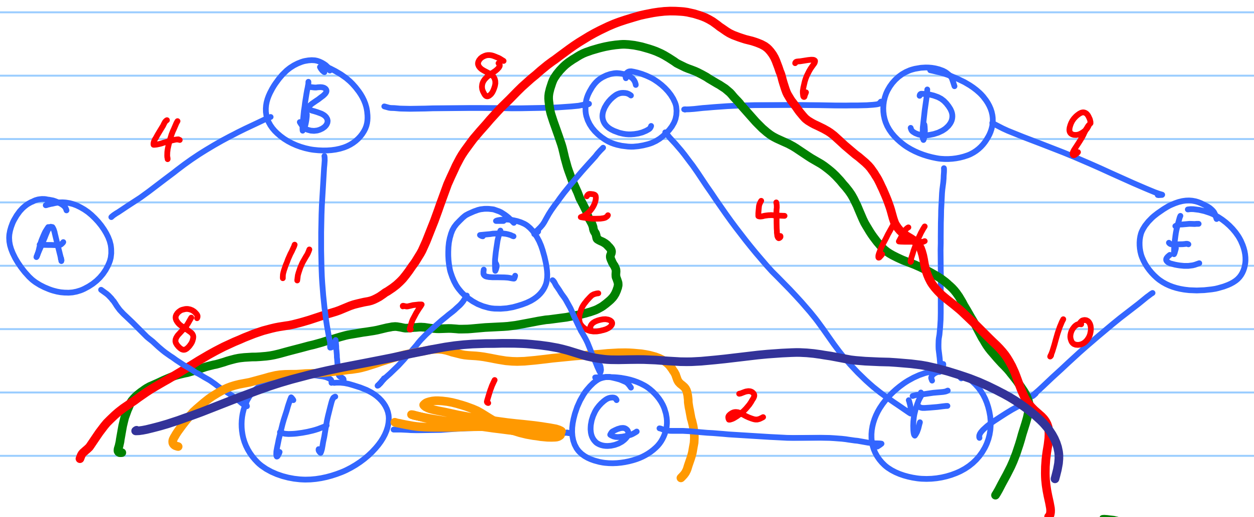
A cut is said to respect a set of edges if no edge in that set crosses the cut.

An edge that crosses a cut is a light edge if its weight is less than or equal to the weight of the other edges that cross that cut.

To select an edge to add to the MST_edges set, we need an edge (u,v) such that given any cut of the graph (S,V-S) that respects MST_edges, (u,v) is a light edge.

Example

Let's trace this algorithm on the following graph:



Start w/ lowest weights

MST_Edges:

H - G ① A B I C F D E
HG

G - F ② A B I C D E
HGF

F - C ④ A B I D E
HGFC

I - C ② A B D E
HGFCI

⇓

C - D (7) A B E

HGFCID

B - C (8) A E

HGFCIDB

A - B (4) E

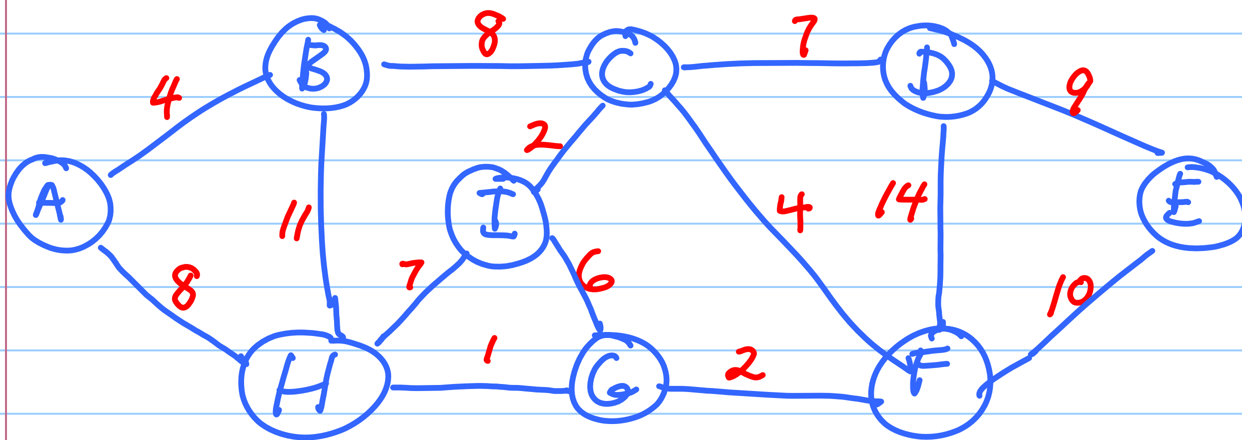
HGFCIDBA

D - E (9) DONE

Does this work?

What if we let an adversary pick the cut?

Adversary Picks Cut



Adversary picks:

D ⁞ F

(To try to get that <u>14</u> into my MST_edges)

But, <u>C</u> has to be on one side or the other, which will lead to either C-D or C-F being a lighter edge!

Proving this algorithm works

- We could formally prove that adding each light edge brings us closer to our MST.
- We would do this using induction on the edge set.
    - Our base case would be MST_edges as empty.
    - Our inductive hypothesis would be that MST_edges is a subset of the MST so far.
    - Our inductive step would be that the next light edge is part of the MST we are trying to build.
        - We would actually show that if the next light edge were NOT in the MST, then we'd have a contradiction.
- We will not do this proof this semester.

Implementing this Algorithm

- The key to implementing this efficiently is to be able to find *light edges* quickly.

- Kruskal's Algorithm makes use of data structures designed for use with disjoint sets (often referred to as Union-Find problems).

- In Union-Find problems you have the ability to quickly:
  - MakeSet(x) – make a set containing only x, where x is in no other set yet
  - Union(x,y) – merge the set that contains x and the set that contains y
  - Find(x) – find the set that contains x

Kruskal's Pseudocode

```
MST_edges = {}
foreach v in V MakeSet(v);              |V| calls to MakeSet
sort the edges by weight                |E| log |E|
foreach (u,v) in sorted edge list {
    if Find(u)!=Find(v) {               |E| Find
        MST_edges += (u,v);             |E| Union
        Union(u,v);
    }
}
```

Reminder: This is a *GREEDY* algorithm.
Note: Its speed relies on the speed of the *MakeSet*, *Union*, and *Find* operations.