Course Notes for

CS 1501
Algorithm Implementation

By
John C. Ramirez
Department of Computer Science
University of Pittsburgh

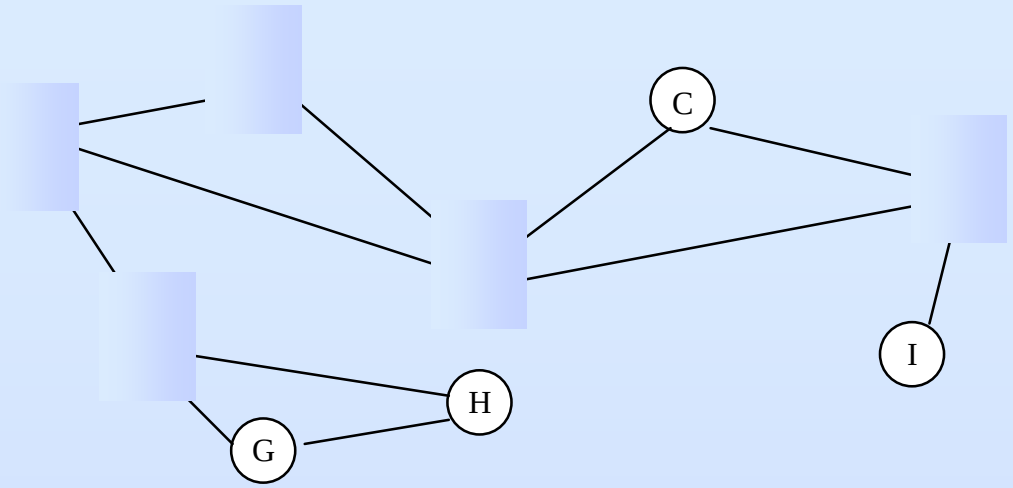
- These notes are intended for use by students in CS1501 at the University of Pittsburgh and no one else
- These notes are provided free of charge and may not be sold in any shape or form
- These notes are NOT a substitute for material covered during course lectures. If you miss a lecture, you should definitely obtain both these notes and notes written by a student who attended the lecture.
- Material from these notes is obtained from various sources, including, but not limited to, the following:
  - Algorithms in C++ by Robert Sedgewick
  - Algorithms, 4th Edition by Robert Sedgewick and Kevin Wayne
  - Introduction to Algorithms, by Cormen, Leiserson and Rivest
  - Various Java and C++ textbooks
  - Various online resources (see notes for specifics)

- Run-times?
  - DFS and BFS have the same run-times, since each must look at every edge in the graph twice
    - During visit of each vertex, all neighbors of that vertex are considered, so each edge will be considered twice
      - Only some edges are subsequently "followed", but all neighbors are considered
    - DFS is recursive and BFS is iterative but both visit each vertex and both consider all neighbors during a visit
  - However, run-times differ depending upon how the graphs are represented
    - Recall that we can represent a graph using either an adjacency matrix or an adjacency list
      - Also recall that text uses an adjacency list (see Graph.java)

- Adjacency Matrix
  - For a vertex to consider all of its neighbors we must traverse a row in the matrix -> Theta(v)
  - Since all v vertices consider all neighbors, the total run-time is **Theta(v^2)**
- Adjacency List
  - For a vertex to consider all of its neighbors we must traverse its list in the adjacency list array
    - Each list may vary in length
    - However, since all vertices consider all neighbors, we know that all adjacency list nodes are considered
    - This is 2e, so the run-time here is **Theta(v + e)**
      - > We need the v in there since e could be less than v (even 0). In this case we still need v time to visit the nodes

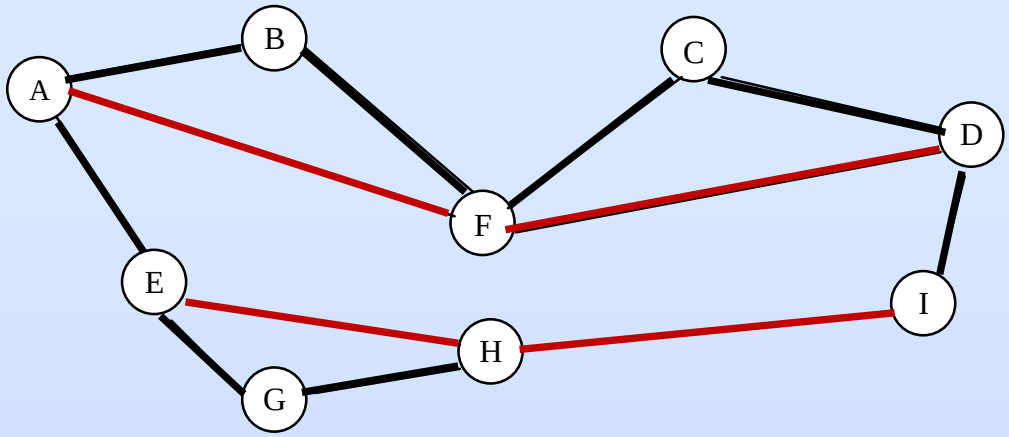
- A biconnected graph has at least 2 distinct paths (no common edges or vertices) between all vertex pairs
  - Idea: If one path is disrupted or broken, there is always an alternate way to get there
- Any graph that is not biconnected has one or more articulation points
  - Vertices, that, if removed, will separate the graph
  - See next slide for example
- Any graph that has no articulation points is biconnected
  - Thus we can determine that a graph is biconnected if we look for, but do not find any articulation points

Biconnected Graph

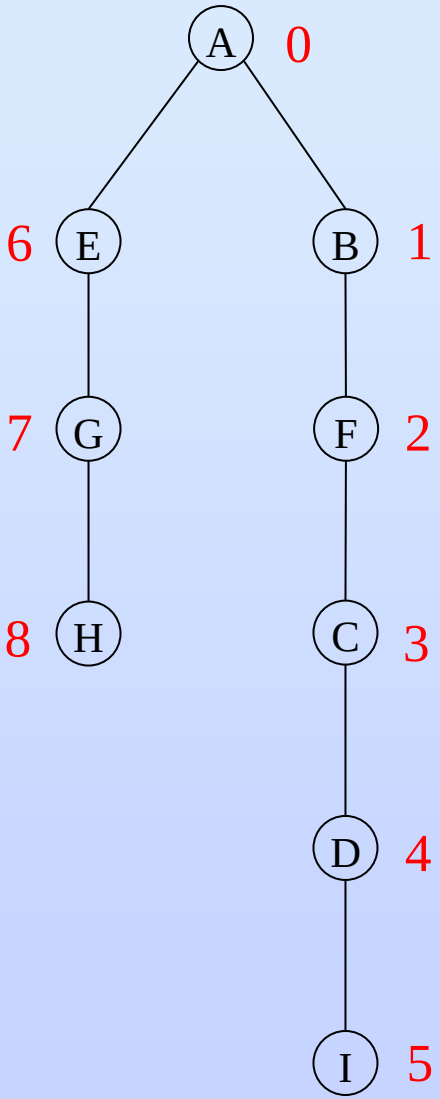


- Articulation points, if removed, will disconnect the graph
  - In the graph above, which vertices are articulation points?
    - A
    - E
    - F
    - D
  - Note that if we remove any other vertex, the graph remains connected
    - Ex: B

- How to find articulation points?
  - Variation of DFS
  - Consider graph edges during DFS
    - Tree Edge: edge crossed when connecting a new vertex to the spanning tree
    - Back Edge: connects two vertices that were already in the tree when it was considered
      - These back edges are critical in showing biconnectivity
      - These represent the "alternate" paths between vertices in the graph
      - Will be used in finding articulation points



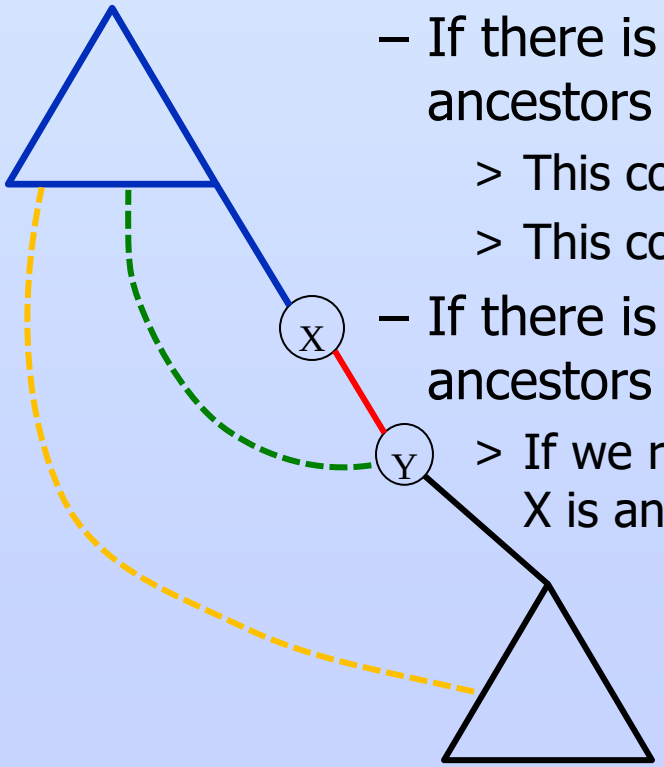
- We can think of the tree edges as the first connection between the vertices
- We can think of the back edges (edges in the graph but NOT in the spanning tree) as "alternative paths" between vertices
- If we have enough back edges then the graph will be biconnected
  - Ex: Consider adding edge (H,I) to this graph
  - Note: In this case, adding (H,I) would change the spanning tree and (H,I) would in fact be a tree edge
  - However, it would clearly make the graph biconnected



- So how do we find articulation points?
  - Consider now the DFS spanning tree in a "top-to-bottom" way
  - Root is the top node in the tree, with DFS # = 0
    - This number is the order that the nodes are encountered in the DFS algorithm
    - Note: Last lecture we started this with 1 but we will start with 0 here
  - Every vertex "lower" in the tree has a DFS number that is larger than the vertices that are "higher" in the tree (along path back to the root)
    - We encounter these later in DFS than the nodes above

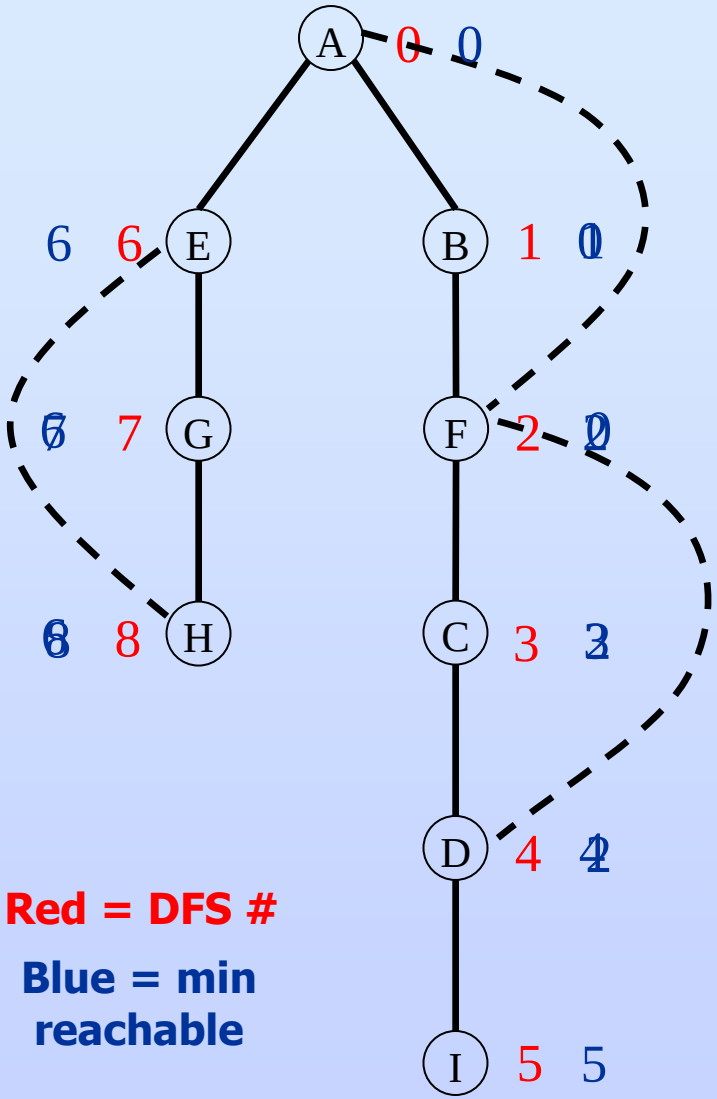
- Given this DFS spanning tree
  - A vertex, X is not an articulation point if every child of X, say Y, is able to connect to a vertex "higher" in the tree (above X, with smaller DFS # than X) without going through X
  - Connection does not have to be a direct edge
    - We could go down through Y's descendents, then eventually back up to a vertex above X
  - If any child is not able to get around X in order to get above X then X is an articulation point

- Idea:
  - We traveled along a **tree edge** from X to Y
    - Through X, we can get from Y to all of the **ancestors of X**
    - If there is some **alternate way** to get from Y to the ancestors of X, then X is NOT an articulation point
      - > This could be a **back edge from Y**
      - > This could be an **edge from one of Y's descendants**
    - If there is NO alternate way to get from Y to the ancestors of X, then X is an articulation point
      - > If we remove the dashed green and orange lines, clearly X is an articulation point in this graph



- How to determine this on the computer?
  - We will modify the DFS algorithm
  - For each vertex, when we visit it, we need to keep track of the minimum DFS # that we can reach from this vertex and any of its descendants
  - We modify this number as we see back edges and as we make recursive calls
  - We then use this min reachable DFS # to determine if a vertex is an articulation point or not:
    - For any vertex X, and its child in the tree Y, if min reachable DFS # for Y is >= DFS #(X), then X is an articulation point

Biconnected Components



- Start at root A (0 and 0)
  - Recurse to B (1 and 1)
  - Recurse to F (2 and 2)
    - Back edge to A – min reachable is now 0
    - This will also cause parent B to have min reachable of 0 when call returns
  - Recurse to C (3 and 3)
  - Recurse to D (4 and 4)
    - Back edge to F – min reachable is now 2
    - This will also cause parent C to have min reachable of 2 when call returns
  - Recurse to I (5 and 5)
- Backtrack all the way to A
  - Recurse to E (6 and 6)
  - Recurse to G (7 and 7)
  - Recurse to H (8 and 8)
    - Back edge to E -- min reachable is now 6
    - This will also cause parent G to have min reachable of 6 when call returns
- Articulation points are now
  - D (because of child I)
  - F (because of child C)
  - E (because of child G)

- What about A?
- Algorithm shown works for all vertices except the root of the DFS tree
  - Child of root cannot possibly reach a smaller DFS # than that of the root, since the root has DFS # 0
  - Root of DFS tree is an articulation point if it has two or more children in the DFS tree
    - Idea is that since we backtrack only when necessary in the DFS algorithm, having two children in the tree means that we couldn't reach the second child except by going back through the root
    - This implies that the first child and second child have no way of connecting except through the root
      - > Thus in this case the root is an articulation point

- For working code see:
  - BiconnectedTrace.java and output file bicon-out.pdf
    - Use data file classGraph.txt
    - I updated the author's code to show the recursive process and used ANSI colors to differentiate one call from the next
      - > These will not show in windows but you can still see the output in the .pdf file.
  - NOTE:
    - The output will not match the slide trace
    - The ordering of the edges in the adjacency list in the trace differs from that shown in the slides
    - However, the algorithm will work as described

- In unweighted graphs, all edges are equivalent (?)
- Often in modeling we need edges to represent distance, flow, capacity, bandwidth, etc.
  - Thus we must put WEIGHTS on the edges to represent these values.
- Representations:
  - In an adjacency matrix, we can substitute the weight for one (zero still means no edge)
  - In an adjacency list, we can add a field to our linked list nodes for weight of the edge

- To implement we could make our weighted graph a subclass of the Graph class previously discussed

- Sedgewick uses the following approach:
  - Define the Edge class to be comparable and to represent a single weighted edge in the graph
  - Define the EdgeWeightedGraph class to have an adjacency list of Edge objects
    - In the original Graph the adjacency list was simply of Integer values, representing vertices in the graph
    - However, we can still find the neighbors by simply looking at the "other" index of each edge
  - See EdgeWeightedGraph.java

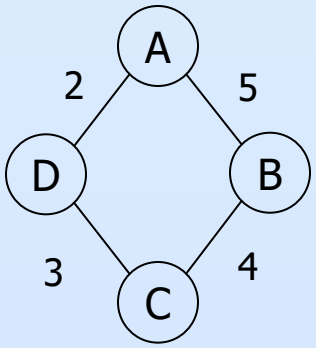
```
public class Edge implements Comparable<Edge> {

    private final int v;
    private final int w;
    private final double weight;

    // much not shown

    public int either()  // get one end of edge
    {
        return v;
    }

    public int other(int vertex)  // get opposite vertex
    {
        if      (vertex == v) return w;
        else if (vertex == w) return v;
        else throw new RuntimeException("Illegal endpoint");
    }
}
```

```
public class EdgeWeightedGraph {
    private final int V;
    private int E;
    private Bag<Edge>[] adj;  // Bag of Edges

    // much not shown

    public void addEdge(Edge e) {
        int v = e.either();
        int w = e.other(v);
        adj[v].add(e);
        adj[w].add(e);
        E++;
    }
```

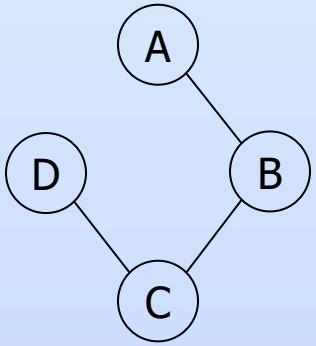
- Now we are adding an Edge in each position of the adjacency list instead of a vertex id

- In an unweighted graph, we have seen algorithms to determine
  - Spanning Tree of the graph
    - Both DFS and BFS generate this during the traversal process, if the graph is connected
  - Shortest Path between two vertices
    - BFS does this by determining spanning tree based on number of edges vertex is away from starting point
    - The spanning tree generated by BFS shows the shortest path (in terms of number of edges) from the starting vertex to all other vertices in the graph
      - > Recall BreadthFirstPaths.java handout
- However, in a weighted graph, these ideas can be more complex

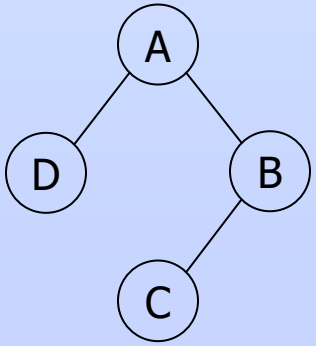
- Minimum Spanning Tree (MST)
  - The spanning tree of a graph whose edge weights sum to the minimum amount
  - Clearly, a graph has many spanning trees, not all of which are minimum
- Weighted Shortest Path
  - The path between two vertices whose edge weights sum to the minimum value
  - Note that now the fewest edges does not necessarily imply the shortest path



DFS Tree



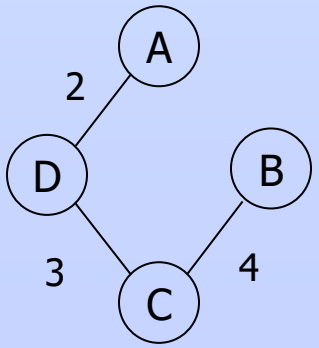
BFS Tree

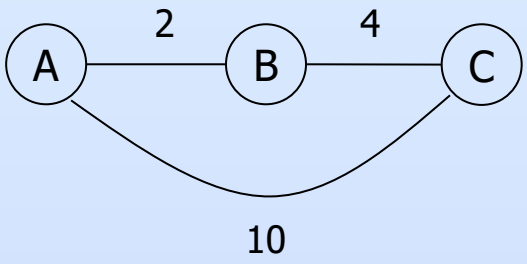


MSTs and Weighted Shortest Path

- Consider the graph on the left
  - Assume edges are stored in alphabetical order
  - Note DFS spanning tree
    - Sum of edge weights = 12
  - Note BFS spanning tree
    - Sum of edge weights = 11
  - Neither is a minimum spanning tree
    - Sum of edge weights = 9

MST





- Consider the graph on the left
  - BFS will find that C is a distance of 1 from A
  - Shortest path by hops
    - BFS is a good algorithm to find this
  - However, clearly by weight ABC is shorter, since it is only 6 and AC is 10
- So how do we calculate the MST and the weighted shortest path?
  - We will look at MST next lecture and shortest path soon