• Homework 4
  • Due Friday at 2pm
• Midterm: Tuesday, March 8, 2016
**Exercise**: In an undirected graph, suppose you are doing a DFS and have processed one child $c_1$ of the root. After the child has finished, there is another child $c_2$ still in the INIT state that must be processed. What happens to the graph if we remove the root?
• DFS can also be used to find bridges and articulation points
• **Definition**: a bridge is an edge whose removal increases the number of connected components
• **Definition**: an articulation point is a vertex whose removal increases the number of connected components
• Problems involving finding bridges and articulation points usually defined for undirected, connected graphs
  • Harder for directed graphs
  • E.g., book’s example: sabotaging road networks
  • Recall naïve approach for finding bridges and articulation points
    • Runtime $O(V^2 + VE)$
• Consider the root of a DFS tree
  • After a child is processed, if other unprocessed children exist, then the root is an articulation point
• For non-roots:
  • Each vertex in the DFS tree will be visited in some order, keep track of two numbers:
    • dfs_num: counter for when the vertex is visited for the first time
    • dfs_low: lowest dfs_num reachable from the vertex
      o But ignore immediate parents
  • Initially, dfs_num = dfs_low for all vertices
  • dfs_low can only be made smaller if there are cycles (back edges)
Articulation Points

dfs_num(0) = 0  
dfs_low(0) = 0

dfs_num(1) = 1  
dfs_low(1) = 1

0

1

2

3

4

5

dfs_num(3) = 4  
dfs_low(3) = 4

dfs_num(4) = 3  
dfs_low(4) = 3

dfs_num(5) = 5  
dfs_low(5) = 5

dfs_num(0) = 0  
dfs_low(0) = 0

dfs_num(1) = 1  
dfs_low(1) = 1

dfs_num(2) = 2  
dfs_low(2) = 2

0

1

2

3

4

5

dfs_num(3) = 3  
dfs_low(3) = 3

dfs_num(4) = 4  
dfs_low(4) = 1

dfs_num(5) = 5  
dfs_low(5) = 1
If a vertex $u$ has neighbor $v$ with $\text{dfs}\_\text{low}(v) \geq \text{dfs}\_\text{num}(u)$, then $u$ is an articulation point
- No back edge from $v$ to ancestors of $u$
- In order for $v$ and descendants to reach ancestors of $u$, traversal must pass through $u$

Exercise: how do we adapt this algorithm to find bridges?
• Same algorithm can be used to find bridges, except: \( \text{dfs\_low}(v) > \text{dfs\_num}(u) \) implies that \((u,v)\) is a bridge
  • Note that it no longer includes equality
• **Spanning tree**
  
  • Given: a connected, undirected graph $G = (V, E)$
    • $(V$ is the set of vertices, $E$ is the set of edges)
  
  • A spanning tree is a set of edges that is a tree and “covers” all vertices $V$
    • There can be several trees
  
  • The spanning tree with the minimum cost (sum of edge weights) is called the **Minimum Spanning Tree**
Minimum spanning tree
Cost: 4 + 2 + 6 + 6 = 18
• Kruskal's algorithm for finding the MST
  • Repeatedly finds edges with minimum costs that does not form a cycle
  • Greedy algorithm, provably correct

• Kruskal's algorithm pseudocode
  • Sort edges by increasing weight
  • While there are unprocessed edges left
    • Pick an edge e with minimum cost
    • If adding e to the MST does not form a cycle, add e to MST
• **Kruskal's algorithm pseudocode**
  - How to store and sort edges?
    - Using an edge list and Collections.sort
  - How to test for cycles?
    - using disjoint sets and union-find (*Exercise*: how?)
  - Runtime?
    - Sort: \(O(|E| \log |E|)\)
    - Processing: for each edge, check union-find: \(O(|E|) \times O(1)\)
    - Total: \(O(|E| \log |E|) = O(|E| \log |V|)\)

• **Exercise**: If the weights of the edges are integers within a small range (e.g., \([0, 100]\)), can Kruskal’s be made faster?
Kruskal’s Algorithm: Example

Pick smallest edge

Pick smallest edge
No cycle
Kruskal’s Algorithm: Example

Pick smallest edge
Cycle formed, ignore

Pick smallest edge
No cycle

Pick smallest edge
No cycle
Algorithm not done!
The edge list hasn't yet been exhausted

Pick smallest edge
Cycle formed, ignore

Pick smallest edge
Cycle formed, ignore
Kruskal’s Algorithm: Partial Code

```java
ArrayList<Edge> edgeList = parseEdgeList();
Collections.sort(edgeList);

int mstCost = 0;
UnionFind uf = new UnionFind(nVertices);

for (Edge e : edgeList) {
    // for each edge
    if (!uf.isSameSet(e.A, e.B)) {
        // if no cycle
        mstCost += e.w; // add it
        uf.union(e.A, e.B);
    }
}

System.out.println(mstCost);
```
Quick summary of Prim’s algorithm:

1. Begin with a set of vertices V, initialized with an arbitrary vertex, and empty set of edges E
2. From all edges, add edge (u, v) with least weight such that u is in V and v is not
   • We’re building a tree, and continuously adding vertices to the tree
3. Repeat step 2 until all vertices have been added to the tree

Greedy algorithm

Also $O(|E| \log |V|)$ running time

More detail in textbook
Exercise: Given a connected weighted graph length that stores the road length between E pairs of cities i and j (1 ≤ V ≤ 1000, 0 ≤ E ≤ 10000), the price p[i] of fuel at each city i, and the fuel tank capacity c of a car (1 ≤ c ≤ 100), determine the cheapest trip cost from starting city s to ending city e using a car with fuel capacity c. All cars use one unit of fuel per unit of distance and start with an empty fuel tank.
Competitive Programming 4.1-4.4