CS 225 Spring 2019 :: TA Lecture Notes
4/17 Traversal + MST

By Wenjie

DFS

- Idea:
  - similar idea with BFS, but we can use either
    - A stack
    - A recursion
  - Recursive algorithm: we visit a vertex **v**
    1. Check all adjacent vertices of **v**
       a. if it's not been discovered, label the edge "discovery edge". Visit the new vertex
       b. label the edge that leads us to a already discovered vertex "back edge"
          i. since it *usually* brings us to a closer vertex
          ii. the distance difference is unbounded
  - Observations
    - The discovery edges make a spanning tree
    - **d** does not find the shortest path
    - the benefit is: it discovers new vertices very quickly

- The code: use system stack as our workstack (recursion)

```
~~BFS~~DFS(G):
     Input: Graph, G
     Output: A labeling of the edges on
                  G as discovery and ~~cross~~ back edges

     foreach (Vertex v : G.vertices()):
          setLabel(v, UNEXPLORED)
     foreach (Edge e : G.edges()):
          setLabel(e, UNEXPLORED)
     foreach (Vertex v : G.vertices()):
          if getLabel(v) == UNEXPLORED:
               ~~BFS~~DFS(G, v)
               //components++;
```

CS 225 Spring 2019 :: TA Lecture Notes
4/17 Traversal + MST

By Wenjie

```
~~BFS~~ DFS(G, v):
    ~~Queue q~~
    setLabel(v, VISITED)
    ~~q.enqueue(v)~~

    ~~while !q.empty():~~
        ~~v = q.dequeue()~~
        foreach (Vertex w : G.adjacent(v)):
            if getLabel(w) == UNEXPLORED:
                setLabel(v, w, DISCOVERY)
                ~~setLabel(w, VISITED)~~
                ~~q.enqueue(w)~~
                DFS(G, w)
            elseif getLabel(v, w) == UNEXPLORED:
                setLabel(v, w, ~~CROSS~~ BACK)
                // cycleExists = true;
```

- Use cases and functionality:
    - Counts the number of components
    - Detects cycles
- Running time:
    - Expect: visit every edge and vertex, so O(n+m)
    - Looking at specific parts of the code:
        - This whole chunk is $O(n \times deg(v))$
        - $deg(v)$ is not very informative, but we know that we will have $n \times deg(v) = \sum^{n} deg(v) = 2m \Rightarrow O(2m)$
    - Total running time is $O(n+m)$.
    - This is optimal running time because we know we have to visit every edge and vertex, therefore we cannot do better than $O(n+m)$.
- DFS doesn't give a unique solution either

Minimum Spanning Tree

- Example:
    - Connect all houses in Urbana with roads

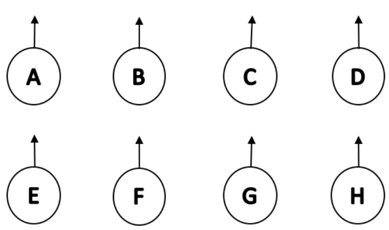
CS 225 Spring 2019 :: TA Lecture Notes
4/17 Traversal + MST

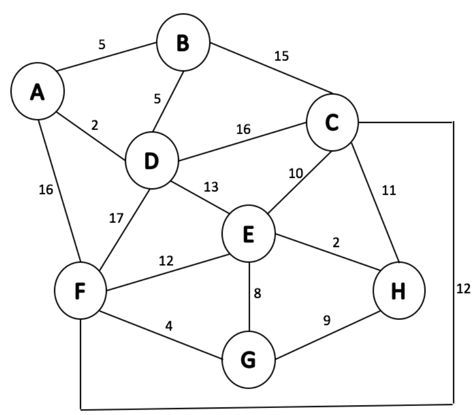
By Wenjie

- Every road has a cost, we want the **min** cost
- **Algorithm:**
    - Input: connected, undirected graph G with edge weights that are additive.
    - Output: A graph G’ with the following properties:
        - G’ is a spanning graph of G
        - G’ is a tree (connected, acyclic graph, with no cycles)
        - G’ has a minimal total weight among all possible spanning trees

Kruskal’s Algorithm

- Setup
    - Maintain a list of edges sorted by weight in increasing order → min heap
    - Initialize a disjoint set for every vertex
        - If two vertices are in the same upTree, they are connected





(A,D)
(E,H)
(F,G)
(A,B)
(B,D)
(G,E)
(G,H)
(E,C)
(C,H)
(E,F)
(F,C)
(D,E)
(B,C)
(C,D)
(A,F)
(D,F)

CS 225 Spring 2019 :: TA Lecture Notes 4/17 Traversal + MST

By Wenjie

- The code

```
KruskalMST(G):
     DisjointSets forest
     foreach (Vertex v : G):
          forest.makeSet(v)

     PriorityQueue Q    // min edge weight
     foreach (Edge e : G):
          Q.insert(e)

     Graph T = (V, {})

     while |T.edges()| < n-1:
          Vertex (u, v) = Q.removeMin()
          if forest.find(u) != forest.find(v):
               T.addEdge(u, v)
               forest.union( forest.find(u),
                                    forest.find(v) )
     return T
```

- The algorithm logic
  - Take the edge with the smallest weight from the heap
  - If the two endpoints are not already connected
    - add the edge into our spanning tree
    - union the two vertices
  - If they are already connected
    - skip the edge and do nothing, otherwise we create a cycle