acm International Collegiate Programming Contest

IBM event sponsor

CS3233
Competitive Programming

Dr. Steven Halim

Week 05 – Graph (Basics)

Outline

- Mini Contest #5 + Break
- Admin
- Graph: Preliminary & Motivation
- Graph Traversal Algorithms
  - DFS: Connected Components (Flood Fill)/Articulation Points/Bridges/Strongly Connected Components/Topological Sort
  - BFS: SSSP on Unweighted graph/Variants
- Minimum Spanning Tree Algorithm
  - Kruskal's: Plus Various Applications

Pot B (NOI): ? → Row 1
Pot C (N/A): ? → Row 2
Pot A (IOI / ICPC): ? → Row 3
Guest #1 = ?

Mid Semester Contest (Week08)

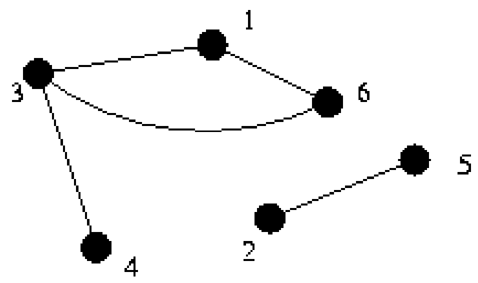
Team 1	Team 2	Team 3	Team 4	Team 5

Final Contest (Week13)

Team 1	Team 2	Team 3	Team 4	Team 5

Graph Terms – Quick Review

- Vertices/Nodes
- Edges
- Un/Weighted
- Un/Directed
- In/Out Degree
- Self-Loop/Multiple Edges (Multigraph) vs Simple Graph
- Sparse/Dense
- Path, Cycle
- Isolated, Reachable
- (Strongly) Connected Component
- Sub Graph
- Complete Graph
- Tree/Forest
- Euler/Hamiltonian Path/Cycle
- Directed Acyclic Graph
- Bipartite Graph



Depth-First Search (DFS)

- Finding Connected Components a.k.a. Flood Fill
- Finding Cycles (Back Edges)
- Finding Articulation Points & Bridges
- Finding Strongly Connected Components
- Topological Sort

Breadth-First Search (BFS)

- Finding Single-Source Shortest Paths on Unweighted Graph
- Variants

GRAPH TRAVERSAL ALGORITHMS

Motivation (1)

- How to solve these UVa problems:
  - 469 (Wetlands of Florida)
    - Similar problems: 260, 352, 572, 782, 784, 785, etc
  - 315 (Network)
    - Similar problems: 610, 796, 10199, etc
  - 11504 (Dominos)
    - Similar problems: 11709, LA 4846, etc
- Without familiarity with **Depth-First Search** algorithm and its variants, they look "hard"

Motivation (2)

- How to solve these UVa problems:
  - 336 (A Node Too Far)
    - Similar problems: 383, 439, 532, 762, 10009, etc
- Without familiarity with **Breadth-First Search** graph traversal algorithm**,** they look "hard"

Graph Traversal Algorithms

- Given a graph, we want to traverse it!
- There are 2 major ways:
  – Depth First Search (DFS)
    - Usually implemented using recursion
    - More natural
    - Most frequently used to traverse a graph
  – Breadth First Search (BFS)
    - Usually implemented using queue (+ map), use STL
    - Can solve special case* of "shortest paths" problem!

Depth First Search – Template

- $O(V + E)$ if using Adjacency List
- $O(V^2)$ if using Adjacency Matrix

```
void dfs(int u) { // DFS for normal usage
  printf(" %d", u); // this vertex is visited
  dfs_num[u] = DFS_BLACK; // mark as visited
  TRvii (AdjList[u], v) // try all neighbors v of vertex u
    if (dfs_num[v->first] == DFS_WHITE) // avoid cycle
      dfs(v->first); // v is a (neighbor, weight) pair
}
```

DFS != Backtracking

- PS: If we remove the visited checking part, DFS becomes backtracking (explore all branches)
  - Slow! Efficient pruning must be done!

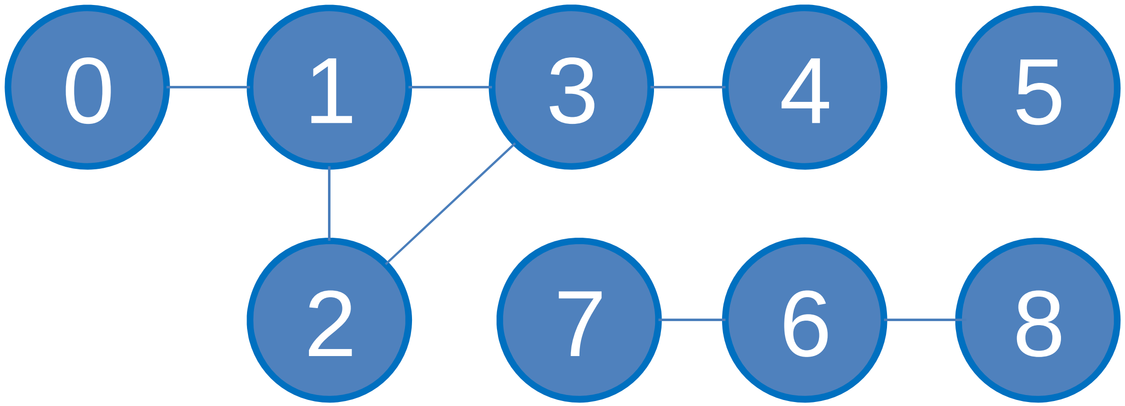
```
void backtracking(vertex) { // invalid state if it
  if (hit end state or invalid state) // causes cycling
    return; // we need terminating/pruning condition
  for each neighbor of vertex // regardless it has been
    backtracking(neighbor);   // visited or not
}
```

Variant: Connected Components

- DFS can find connected components
  - A call of dfs(**u**) visits only vertices connected to **u**

```
int numComp = 0;
dfs_num.assign(V, DFS_WHITE);
REP (i, 0, V - 1) // for each vertex i in [0..V-1]
  if (dfs_num[i] == DFS_WHITE) { // if not visited yet
    printf("Component %d, visit", ++numComp);
    dfs(i); // one component found
    printf("\n");
  }
printf("There are %d connected components\n", numComp);
```

DFS Animation

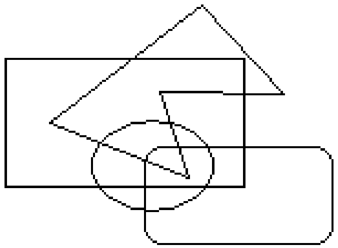


Variant: Flood Fill

- Usually done on implicit graph (2D grid) - UVa [469]
  - Vertices: cells in grid, Edges: N/E/S/W (or to 8 directions)

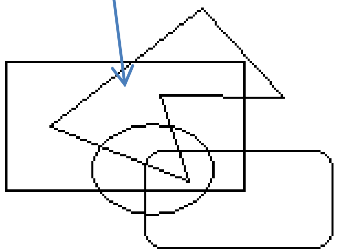
```
int dr[] = {1,1,0,-1,-1,-1, 0, 1}; // S,SE,E,NE,N,NW,W,SW
int dc[] = {0,1,1, 1, 0,-1,-1,-1}; // neighbors
int floodfill(int r, int c, char c1, char c2) {
  if (r<0 || r>=R || c<0 || c>=C) return 0; // outside
  if (grid[r][c] != c1) return 0; // we want only c1
  grid[r][c] = c2; // important step to avoid cycling!
  int ans = 1; // coloring c1 -> c2, add 1 to answer
  for (int d = 0; d < 8; d++) // recurse to neighbors
    ans += floodfill(r + dr[d], c + dc[d], c1, c2);
  return ans;
}
```

Visualization – 469



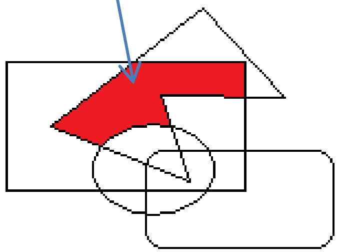
```
LLLLLLLLL
LLWWLLWLL
LWWLLLLLL
LWWWLWWLL
LLLWWWLLL
LLLLLLLLL
LLLWWLLWL
LLWLWLLLL
LLLLLLLLL
```

Starting point



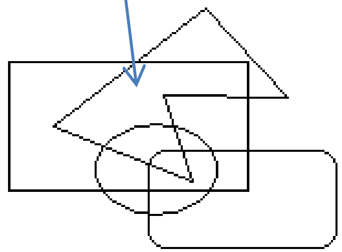
```
LLLLLLLLL
LLWWLLWLL
LWWLLLLLL
LWWWLWWLL
LLLWWWLLL
LLLLLLLLL
LLLWWLLWL
LLWLWLLLL
LLLLLLLLL
```

Flooded +
area counted



```
LLLLLLLLL
LL..LLWLL
L..LLLLLL
L...L..LL
LLL...LLL
LLLLLLLLL
LLLWWLLWL
LLWLWLLLL
LLLLLLLLL
```

To undo the flood fill,
simply reverse the color



```
LLLLLLLLL
LLWWLLWLL
LWWLLLLLL
LWWWLWWLL
LLLWWWLLL
LLLLLLLLL
LLLWWLLWL
LLWLWLLLL
LLLLLLLLL
```

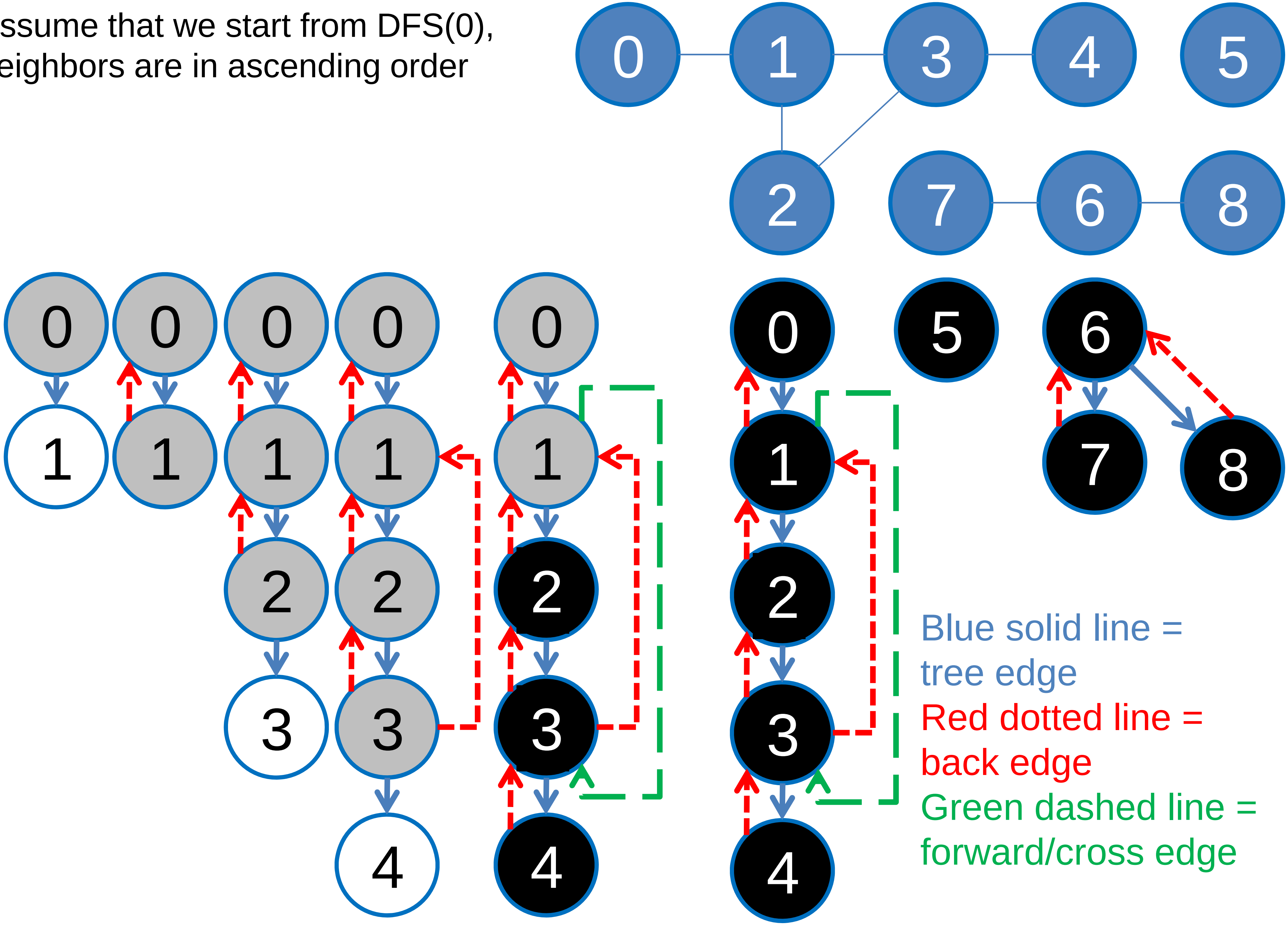
DFS Spanning Tree/Forest

- If DFS runs on a *connected component* of a graph, it will form a **DFS spanning tree**
  - With it, we can classify edges into four types:
    - Tree edges: those selected/traversed by DFS
    - **Back edges: for testing cycle**
    - Forward edges: connect vertex to its descendant
    - Cross edges: all other edges
- If the graph has many components, we have **DFS spanning forest** (for finding components)!

Full Code: White-Gray-Black DFS

```
void graphCheck(int u) { // DFS for checking graph edge properties...
  dfs_num[u] = DFS_GRAY; // color this as DFS_GRAY (temporary)
  TRvii (AdjList[u], v) { // traverse this AdjList
    if (dfs_num[v->first] == DFS_WHITE) { // DFS_GRAY to DFS_WHITE
      // printf("  Tree Edge (%d, %d)\n", u, v->first);
      dfs_parent[v->first] = u; // parent of this children is me
      graphCheck(v->first);
    }
    else if (dfs_num[v->first] == DFS_GRAY) { // DFS_GRAY to DFS_GRAY
      if (v->first == dfs_parent[u])
        printf(" Bidirectional Edge (%d, %d) - (%d, %d)\n", u, v->first, v->first, u);
      else
        printf(" Back Edge (%d, %d) (Cycle)\n", u, v->first);
    }
    else if (dfs_num[v->first] == DFS_BLACK) // DFS_GRAY to DFS_BLACK
      printf(" Forward/Cross Edge (%d, %d)\n", u, v->first);
  }
  dfs_num[u] = DFS_BLACK; // now color this as DFS_BLACK (DONE)
}
```

Assume that we start from DFS(0), neighbors are in ascending order



At the end, we call DFS(0), DFS(5), & DFS(6), i.e. We have 3 DFS spanning trees i.e. a forest of 3 trees

This implies that we have 3 connected components

Finding Articulation Points and Bridges

Finding Strongly Connected Component

Finding Topological Sort

TARJAN'S DFS ALGORITHMS

Articulation Points and Bridges

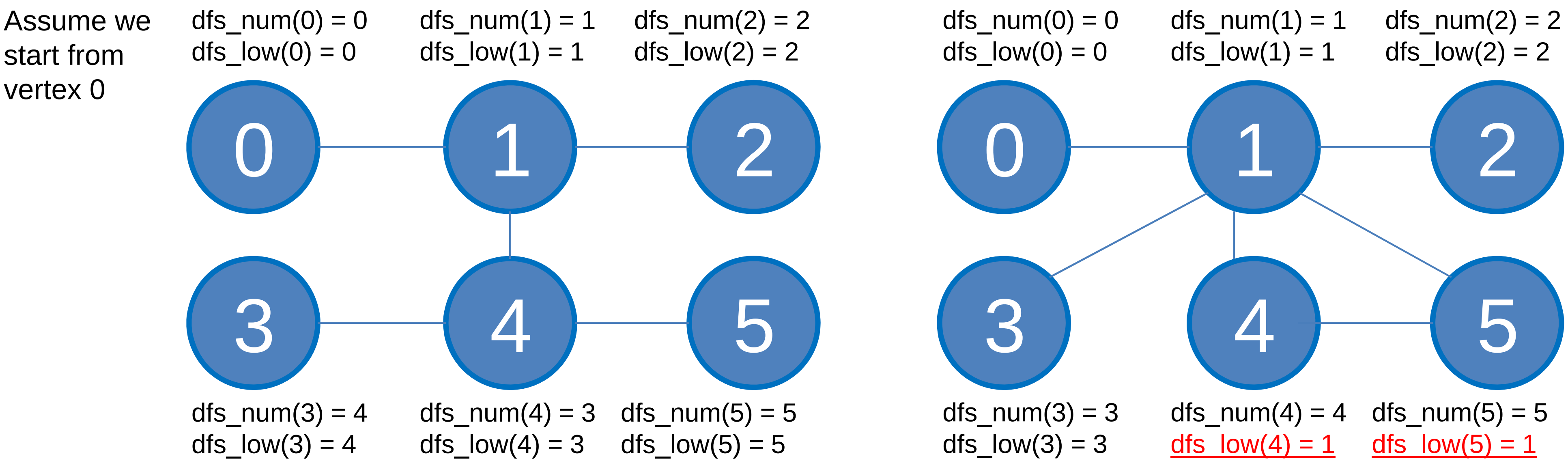
- Given Singapore road map, sabotage either an **intersection** or a **road** that has minimum cost s.t. Singapore road network breaks down.
- This is problem of finding Articulation Points & Bridges
  - Solvable using O(V+E) DFS

Finding Articulation Points (1)

- Articulation Point (UVa: 315, 10199):
  - A vertex in graph G whose removal disconnects G
  - Graph without articulation point is called "Biconnected"
- Trivial Algorithm: $O(V * (V+E)) = O(V^2 + VE)$
  - Run $O(V+E)$ DFS to count number of connected components (cc) of the original graph
  - Repeat for all vertex $O(V)$
    - Cut (remove) one vertex v and its incident edges
    - Run DFS to check if number of cc increase $O(V+E)$
      - If yes, v is an articulation point/cut vertex
    - Restore v and its incident edges

Finding Articulation Points (2)

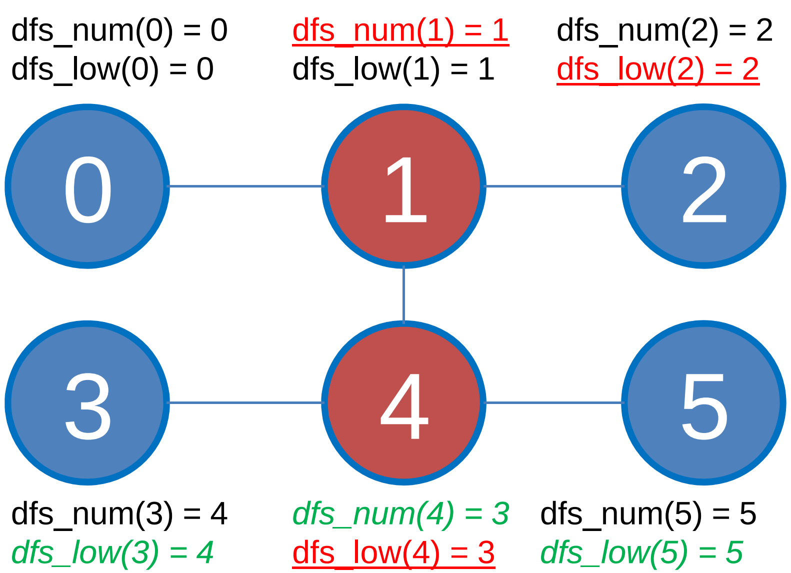
- Better Algorithm: Just modify the O(V+E) DFS
  - Run DFS, but now we count dfsnum(u) and low(u)
    - dfs_num(u) = iteration counter when u is ***first*** visited
    - dfs_low(u) = lowest dfsnum reachable from subtree of u
      - Initially dfs_low(u) = dfs_num(u) when u is first visited
      - Do not update dfs_low(u) with back edge (u, v) if v is a direct parent of u
      - dfs_low(u) can only be smaller if there is a cycle (some other back edges)



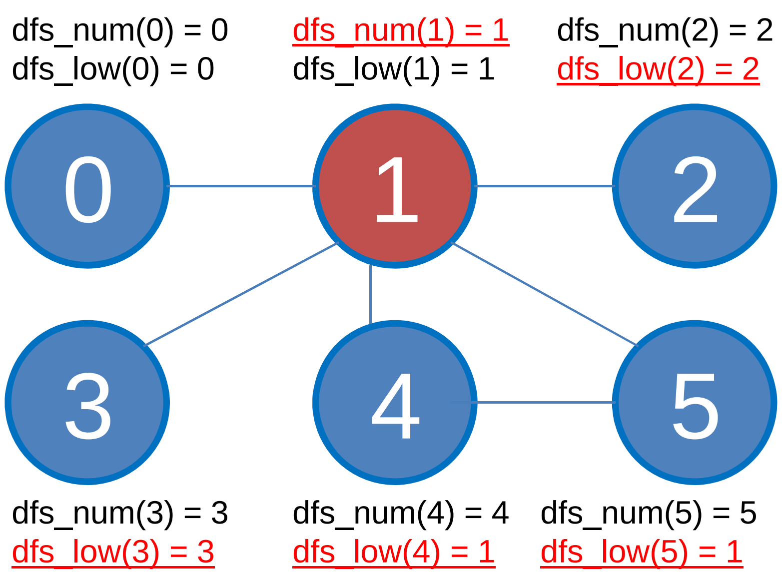
Finding Articulation Points (3)

- Better Algorithm: Just modify the O(V+E) DFS
  - Now, when we are in a vertex u where dfsnum(u) ≤ low(v) and v is a neighbor of u, then u is an articulation vertex
    - The fact that low(v) is not smaller than dfsnum(u) imply that there is no back edge to vertex w that has lower dfsnum(w)
      - To reach parent of u from v, one must pass through u
      - Removing vertex u will thus disconnect the graph

Assume we start from vertex 0

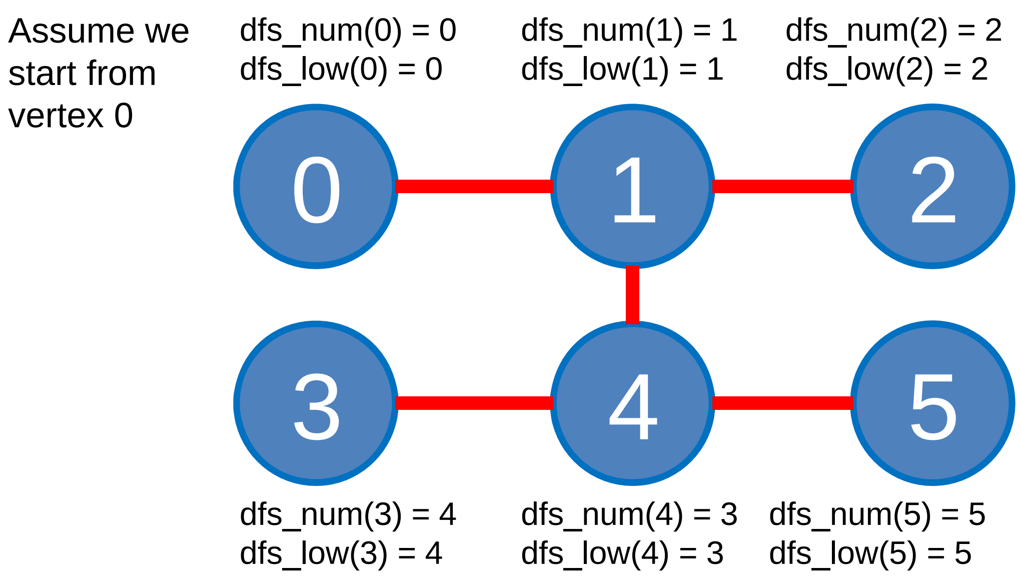


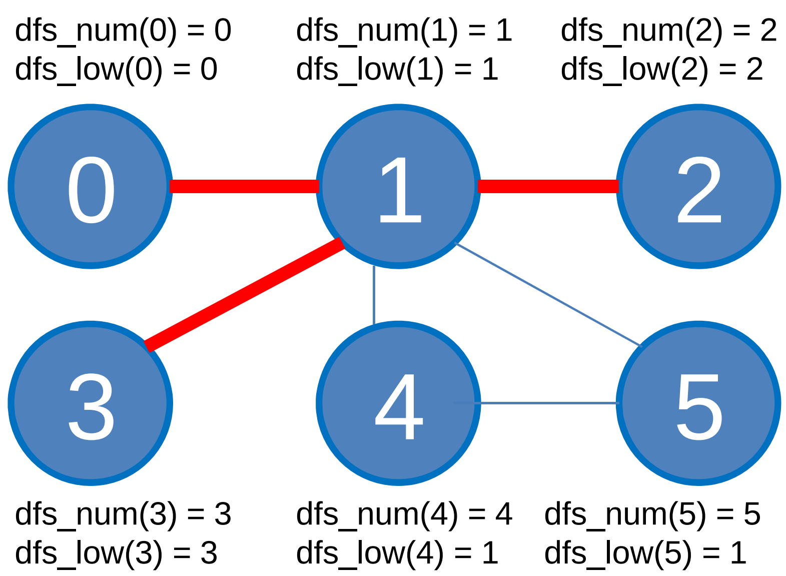
Special case: Root of DFS spanning tree is an articulation point only if it has > 1 children



Finding Bridges

- Bridge (UVa: 796, 610):
  - An edge in graph G whose removal disconnects G
- Simple modification from previous DFS code
  - When dfs_low(v) > dfs_num(u) then edge u-v is a bridge
    - Similar reasoning as previous slide

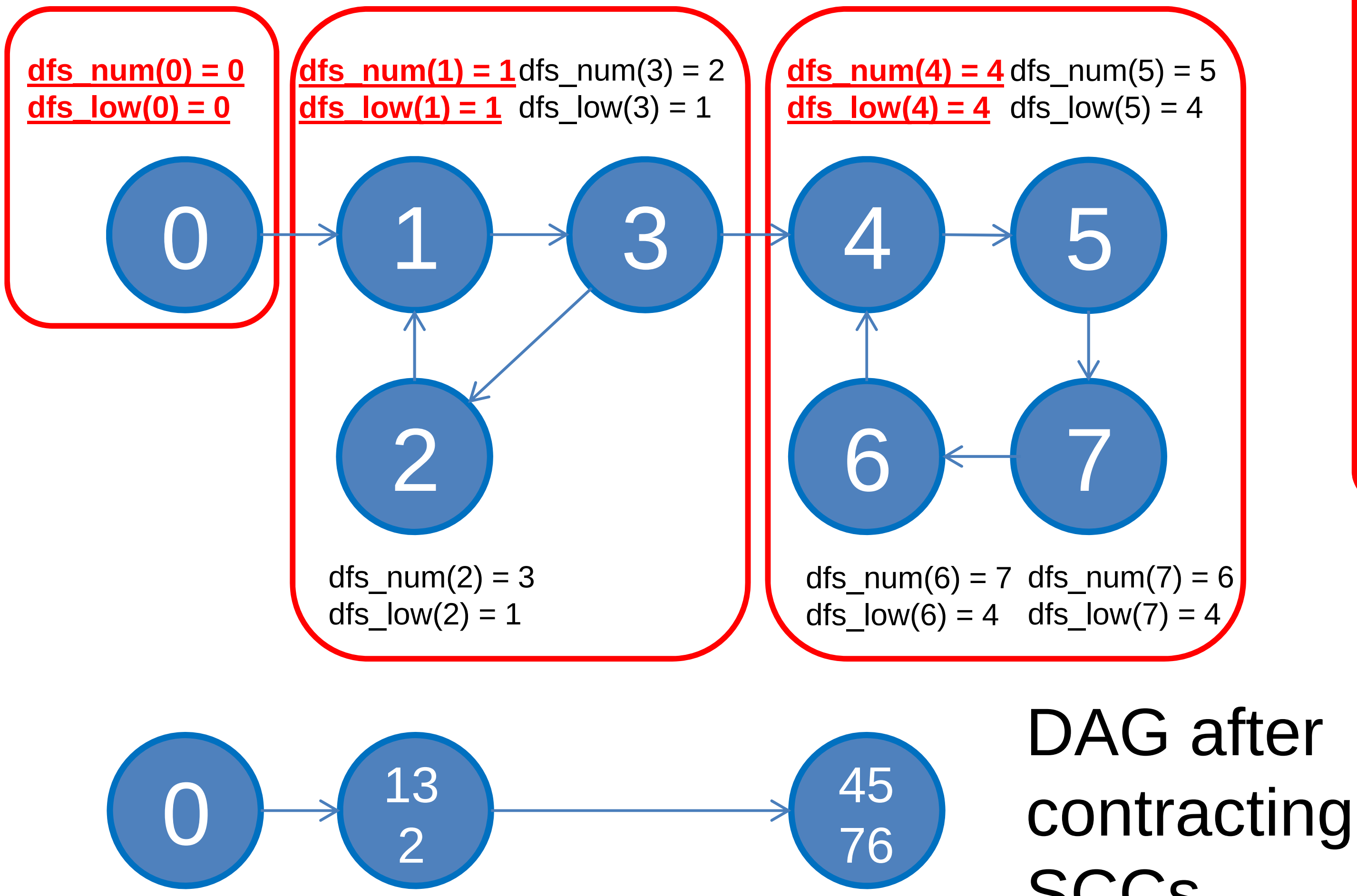




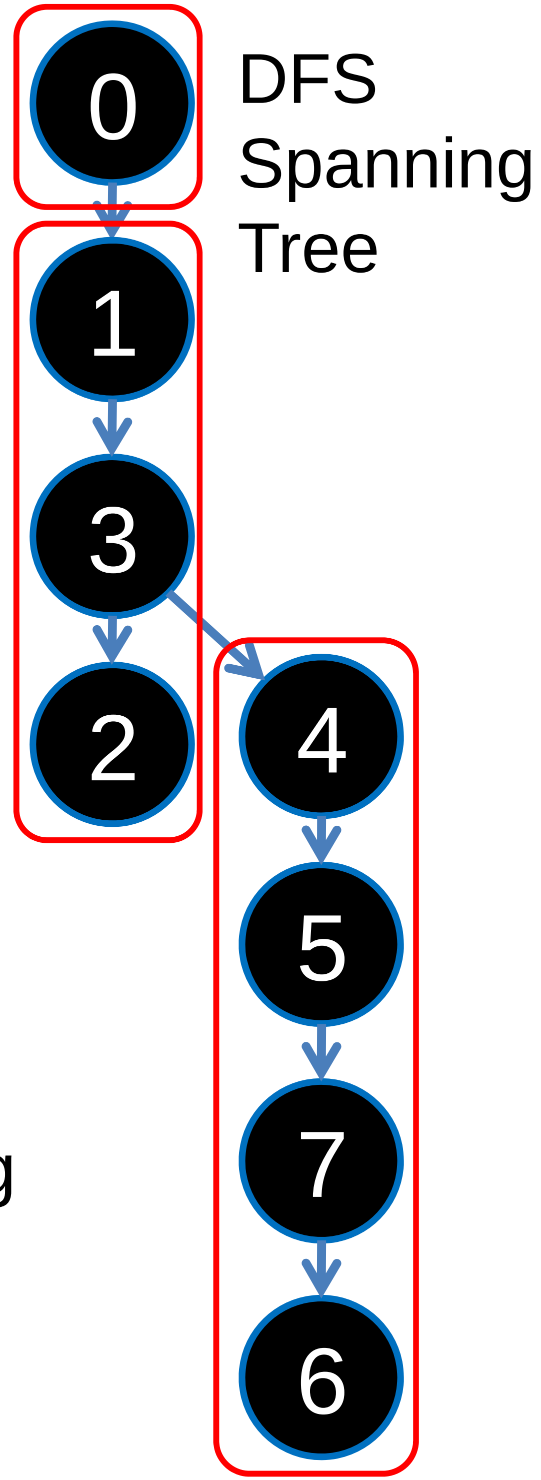
Code: Modified DFS

```
void articulationPointAndBridge(int u) {
  dfs_low[u] = dfs_num[u] = dfsNumberCounter++; // dfs_low[u] <= dfs_num[u]
  TRvii (AdjList[u], v)
    if (dfs_num[v->first] == DFS_WHITE) { // a tree edge
      dfs_parent[v->first] = u; // parent of this children is me
      if (u == dfsRoot) // special case
        rootChildren++; // count children of root
      articulationPointAndBridge(v->first);
      if (dfs_low[v->first] >= dfs_num[u]) // for articulation point
        articulation_vertex[u] = true; // store this information first
      if (dfs_low[v->first] > dfs_num[u]) // for bridge
        printf(" Edge (%d, %d) is a bridge\n", u, v->first);
      dfs_low[u] = min(dfs_low[u], dfs_low[v->first]); // update dfs_low[u]
    }
    else if (v->first != dfs_parent[u]) // a back edge and not direct cycle
      dfs_low[u] = min(dfs_low[u], dfs_num[v->first]); // update dfs_low[u]
}
```

Tarjan's SCC



dfs_num(0) = 0
dfs_low(0) = 0
dfs_num(1) = 1
dfs_low(1) = 1
dfs_num(3) = 2
dfs_low(3) = 1
dfs_num(4) = 4
dfs_low(4) = 4
dfs_num(5) = 5
dfs_low(5) = 4
0
1
3
4
5
2
6
7
dfs_num(2) = 3
dfs_low(2) = 1
dfs_num(6) = 7
dfs_low(6) = 4
dfs_num(7) = 6
dfs_low(7) = 4
13
2
45
76
DAG after contracting SCCs



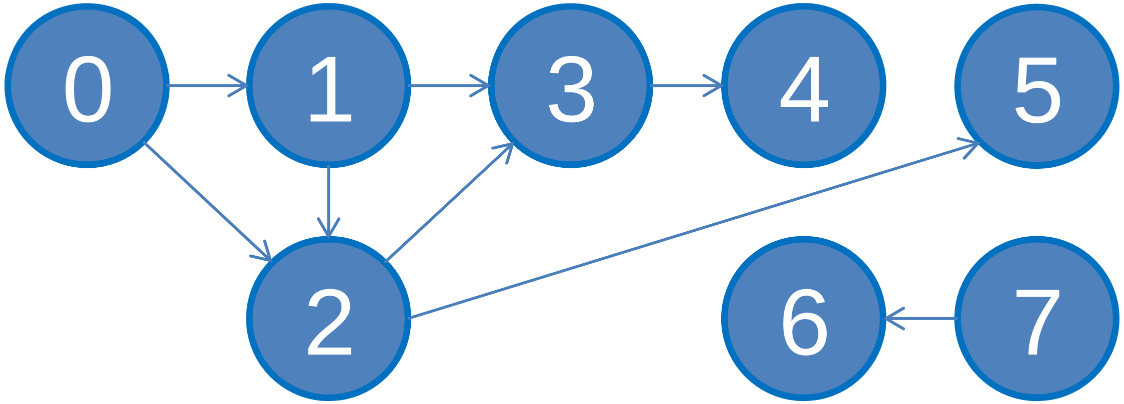
0
1
3
2
4
5
7
6
DFS Spanning Tree

Code: Tarjan's SCC

```
stack<int> dfs_scc; // additional information for SCC
set<int> in_stack; // for dfs_low update check

void tarjanSCC(int u) {
  dfs_low[u] = dfs_num[u] = dfsNumberCounter++; // dfs_low[u] <= dfs_num[u]
  dfs_scc.push(u); in_stack.insert(u); // stores u based on order of visitation
  TRvii (AdjList[u], v) {
    if (dfs_num[v->first] == DFS_WHITE) // a tree edge
      tarjanSCC(v->first);
    if (in_stack.find(v->first) != in_stack.end()) // condition for update
      dfs_low[u] = min(dfs_low[u], dfs_low[v->first]); // update dfs_low[u]
  }
  if (dfs_low[u] == dfs_num[u]) { // if this is a root of SCC
    printf("SCC %d: ", ++numComp);
    while (!dfs_scc.empty() && dfs_scc.top() != u) {
      printf("%d ", dfs_scc.top()); in_stack.erase(dfs_scc.top()); dfs_scc.pop();
    }
    printf("%d\n", dfs_scc.top()); in_stack.erase(dfs_scc.top()); dfs_scc.pop();
} }
```

Topological Sort



Valid toposort: 7, 6, 0, 1, 2, 5, 3, 4
Another toposort: 0, 1, 2, 5, 3, 4, 7, 6
Can you find another valid toposort?

Topological Sort

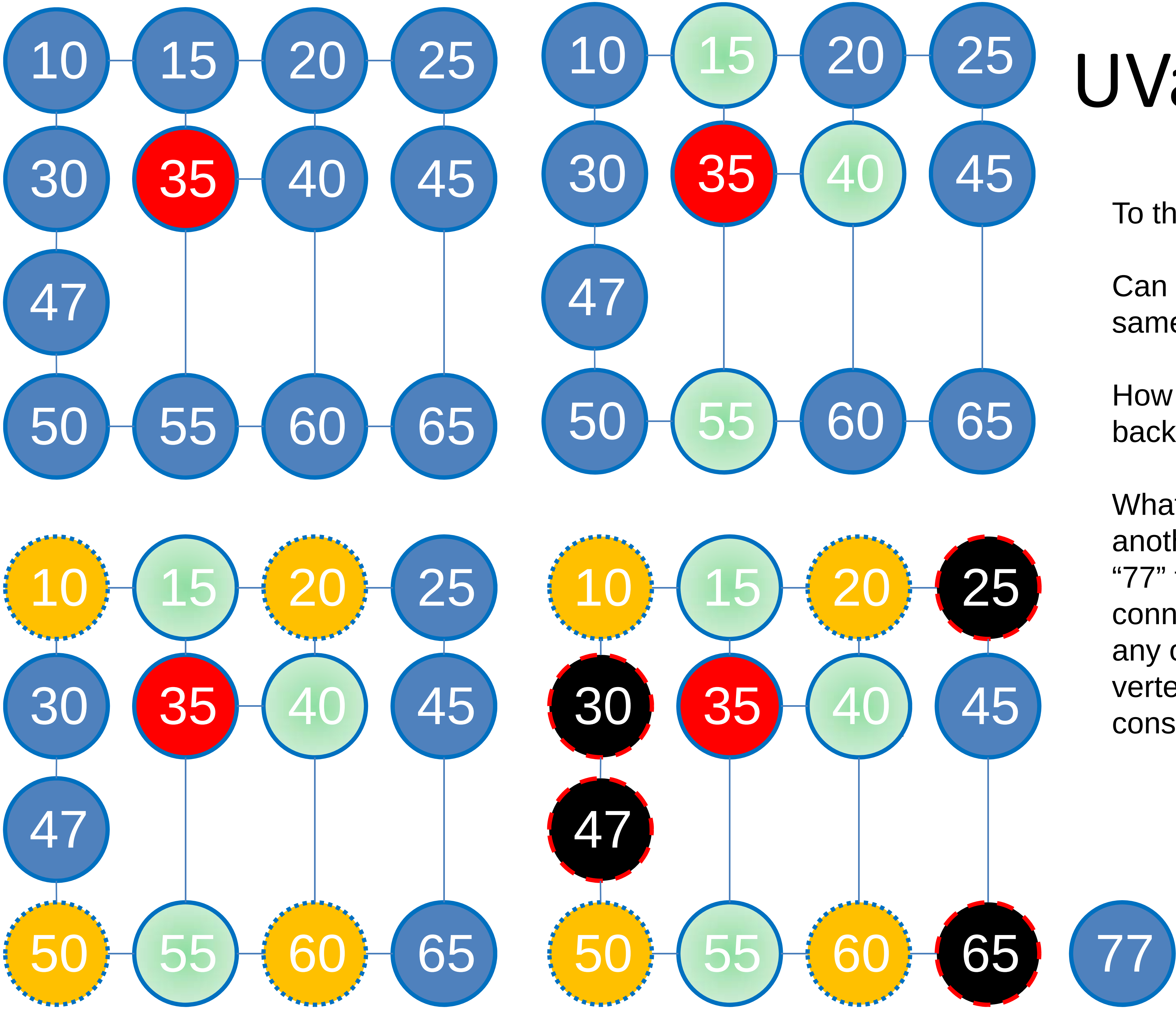
- Yet another simple modification from DFS
  - Append currently visited node to list of visited nodes only after processing all its children
    - This satisfies the topological sort property

```
void topoVisit(int u) {
  dfs_num[u] = DFS_BLACK;
  TRvii (AdjList[u], v)
    if (dfs_num[v->first] == DFS_WHITE)
      topoVisit(v->first);
  topologicalSort.push_back(u); // this is the only change
}
```

BFS for **Special Case** Shortest Paths

- UVa: 336 (A Node Too Far)
- Problem Description:
  - Given an **un-weighted** & un-directed Graph, a starting vertex ***v***, and an integer TTL
  - Check how many nodes are un-reachable from ***v*** or has distance > TTL from ***v***
    - i.e. length(**shortest_path**(v, node)) > TTL

UVa 336



To think about:

Can DFS do the same job?

How about backtracking?

What if we have another vertex “77” that is not connected with any other vertex? Any consequences?

Breadth First Search (using STL)

- Complexity: also O(V + E) using Adjacency List

```
queue<int> q; map<int, int> dist;
q.push(source); dist[source] = 0; // start from source

while (!q.empty()) {
  int u = q.front(); q.pop(); // queue: layer by layer!
  TRvii (AdjList[u], v) // for each neighbours of u
    if (!dist.count(v->first)) {
      dist[*v] = dist[u] + 1; // v unvisited + reachable
      q.push(*v); // enqueue v for next steps
    }
}
```

429 – Word Transformation (1)

- Given starting, ending, and list of other words
- Find **shortest** sequence of 1 char transform-ation from starting word to ending word
  - Example:
    - Starting word: **spice**, Ending word: **stock**
    - List of other words (max 200): dip lip mad map maple may pad pip pod pop sap sip slice slick spice stick stock
    - Ans: spice slice slick stick stock (4 transformations)
- No graph in problem description?

429 – Word Transformation (2)

- Where is the graph?
  - Each word is a vertex
  - Connect two vertices (words) with edge if Hamming distance between them is 1
    - (only 1 character difference)
- What is the graph problem?
  - **sssp** from starting word, output dist[ending word]
- What is the appropriate graph algorithm?
  - O(V + E) BFS, as the graph is unweighted

How About BFS Spanning/SP Tree?

- Nothing much…^
  - Typical application is to reconstruct all shortest paths from single source of an un-weighted graph

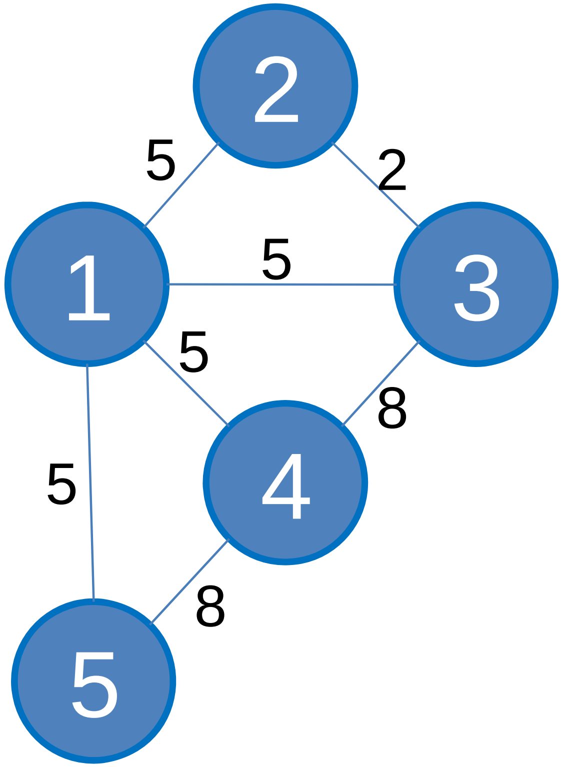
Graph Traversal Comparison

- DFS
- Pro:
  - Slightly easier to code
  - Use less memory
- Cons:
  - Cannot solve SSSP on unweighted graphs

- BFS
- Pro:
  - Can solve SSSP on unweighted graphs
- Cons:
  - Slightly longer to code
  - Use more memory

KRUSKAL'S ALGORITHM FOR MINIMUM SPANNING TREE

How to Solve This?



- Given this graph, select some edges s.t the graph is connected but with minimal total weight!

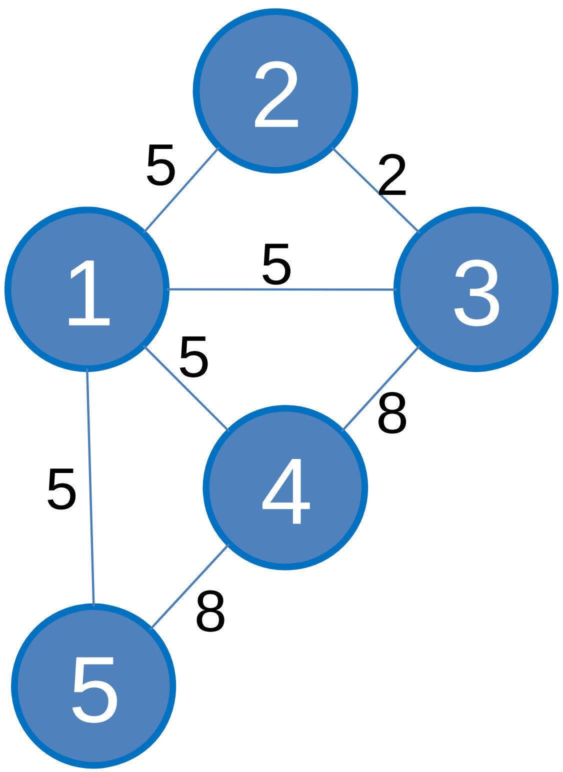
- MST!

Spanning Tree & MST

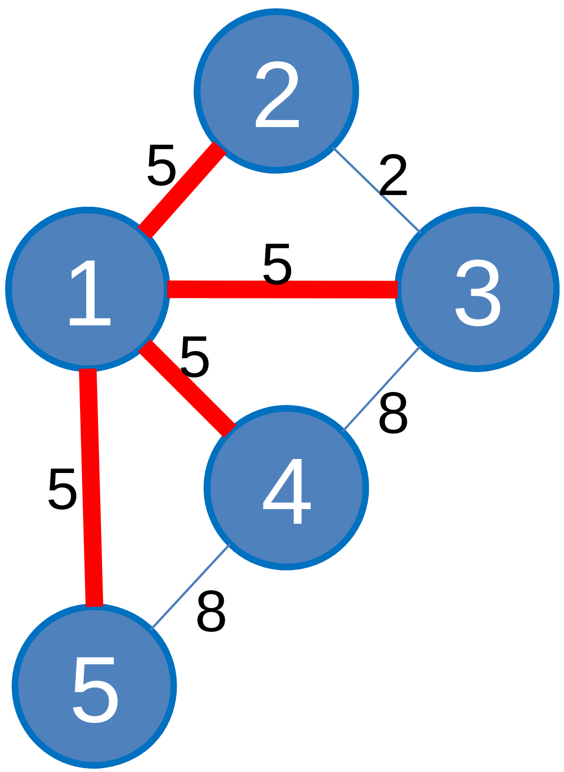
- Given a **connected undirected** graph G, select E $\in$ G such that a tree is formed and this tree **spans** (covers) all V $\in$ G!
  - No cycles or loops are formed!
- There can be **several** spanning trees in G
  - The one where total cost is minimum is called the **Minimum** Spanning Tree (**MST**)
- UVa: 908 (Re-connecting Computer Sites)

Visualization – 908 (1)

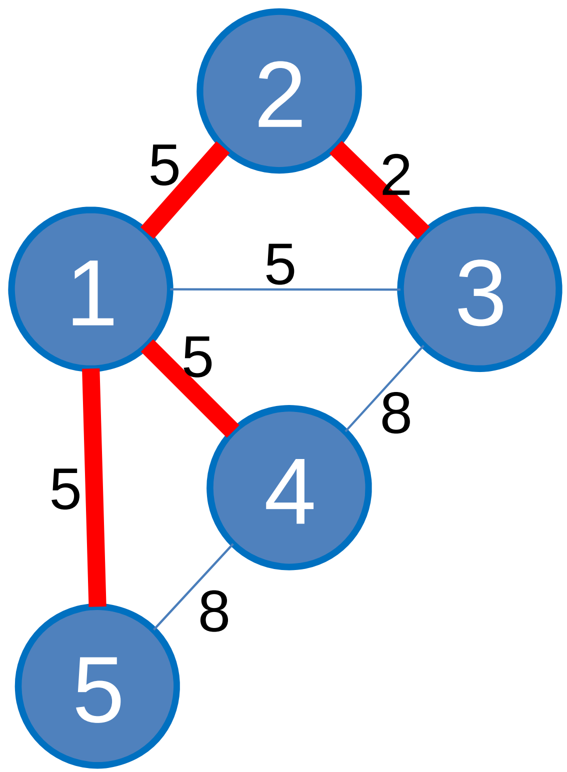
The Original Graph



A Spanning Tree
Cost: 5+5+5+5 = 20



An MST
Cost: 5+5+5+2 = 17



Algorithms for Finding MST

- Prim’s (Greedy Algorithm)
  - At every iteration, choose an edge with minimum cost that does not form a cycle
    - “grows” an MST from a root
- Kruskal’s (also Greedy Algorithm)
  - Repeatedly finds edges with minimum costs that does not form a cycle
    - forms an MST by connecting forests
- Which one is easier to code?

Kruskal's Algorithm

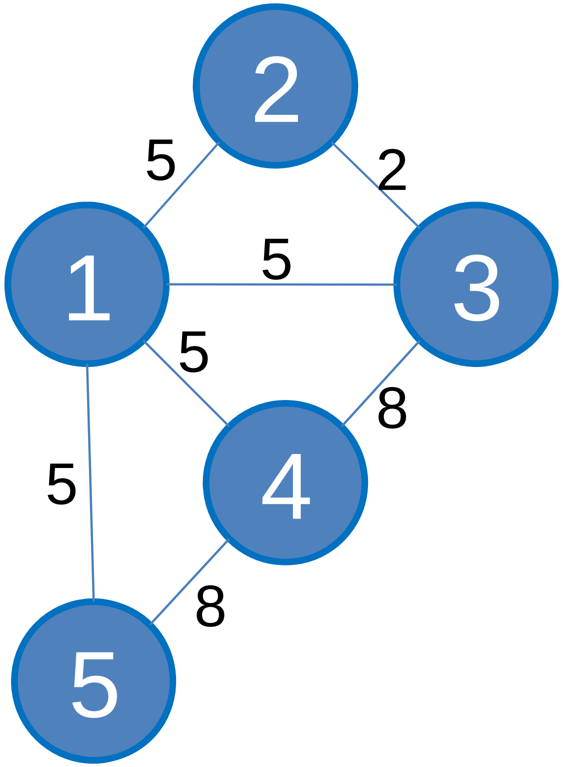
- In my opinion, Kruskal's algorithm is simpler

```
sort edges by increasing weight O(E log E)
while there are unprocessed edges left O(E)
  pick an edge e with minimum cost
  if adding e to MST does not form a cycle
    add e to MST
```

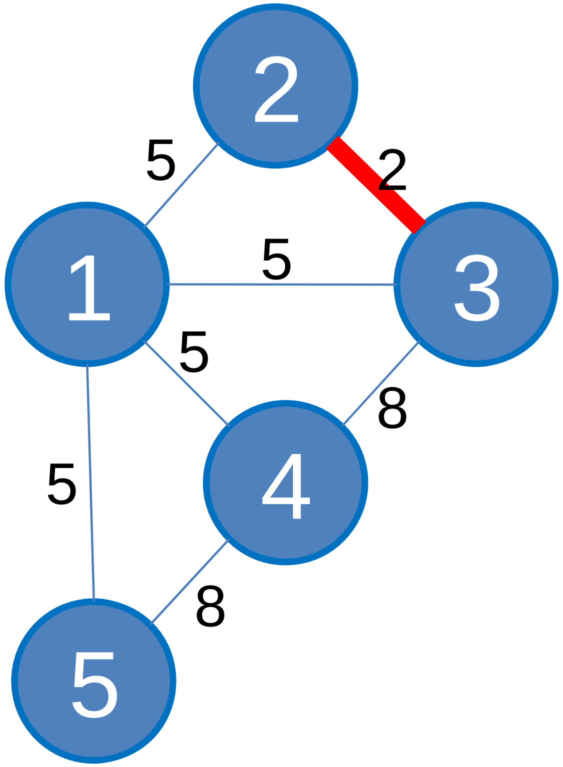
  – Either use Priority Queue or simply sort the edges
    - No need to modify priority_queue in STL <queue>
      – This is an issue for Prim's algorithm
  – Test for cycles using Disjoint Sets (Union Find) DS

Visualization – 908 (2)

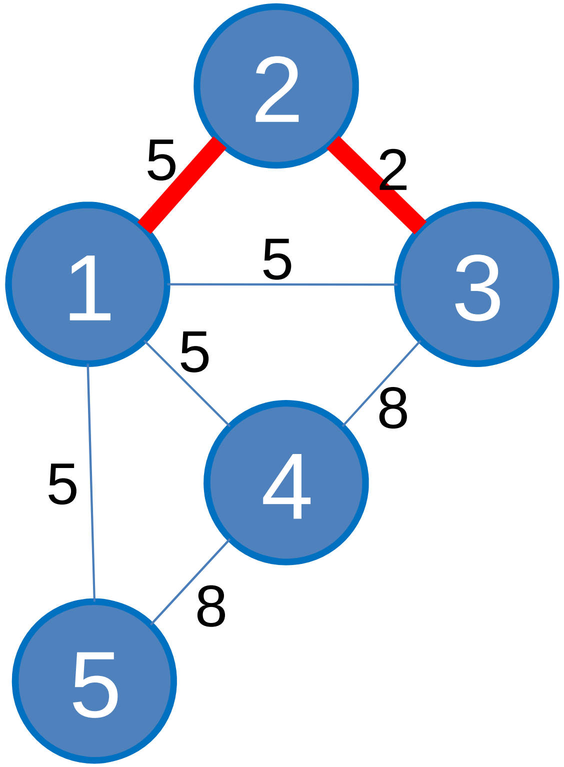
The original graph,
no edge is selected



Connect 2 and 3
As this edge is smallest

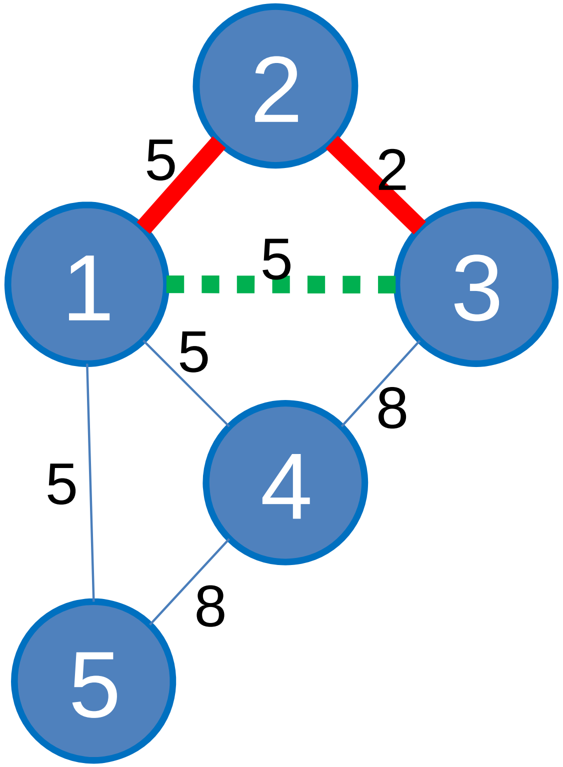


Connect 2 and 1
No cycle is formed

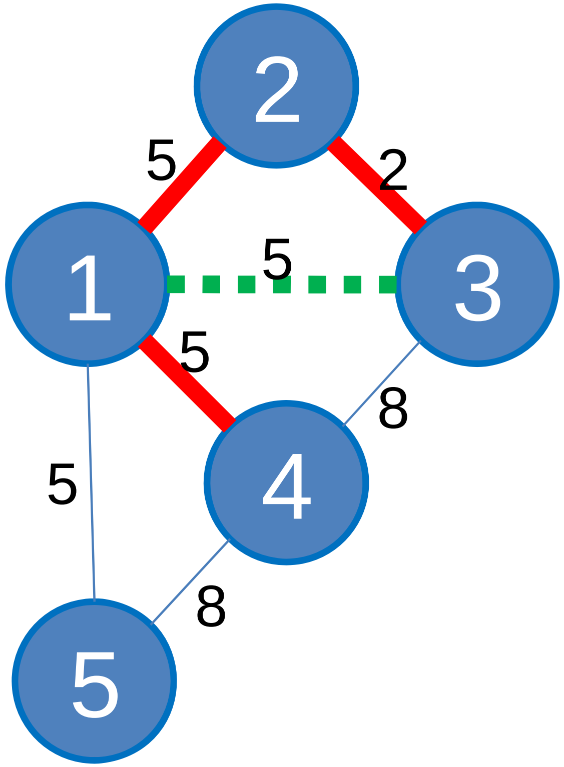


Visualization – 908 (3)

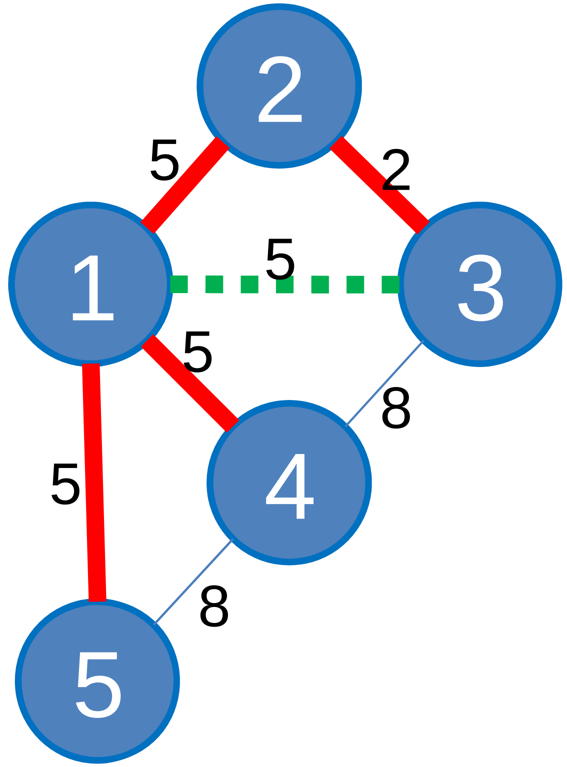
Cannot connect 1 and 3
As it will form a cycle



Connect 1 and 4
The next smallest edge

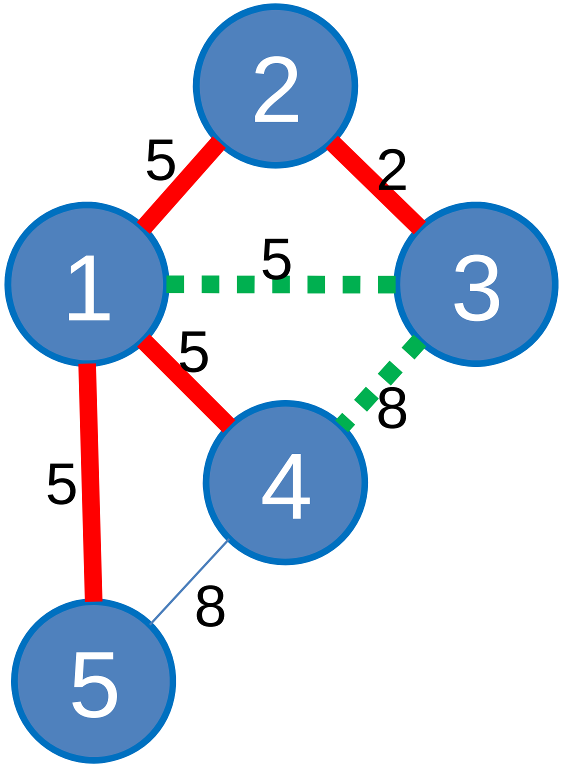


Connect 1 and 5
MST is formed…

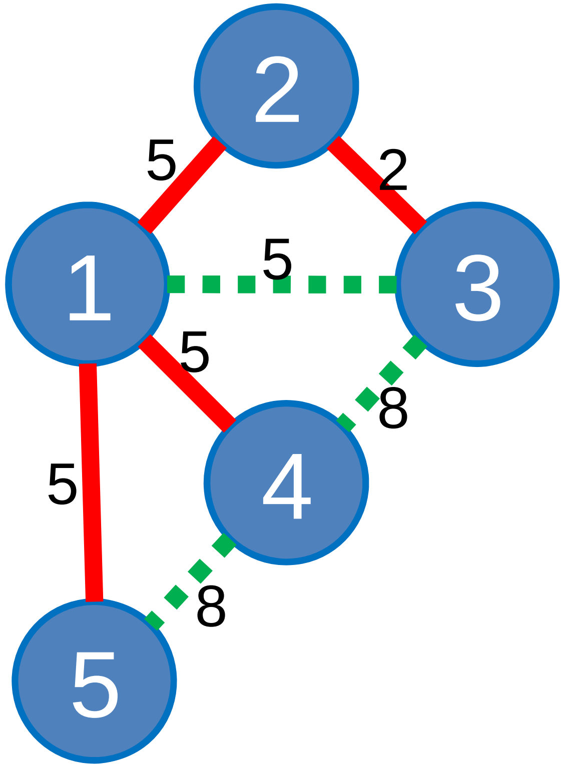


Visualization – 908 (4)

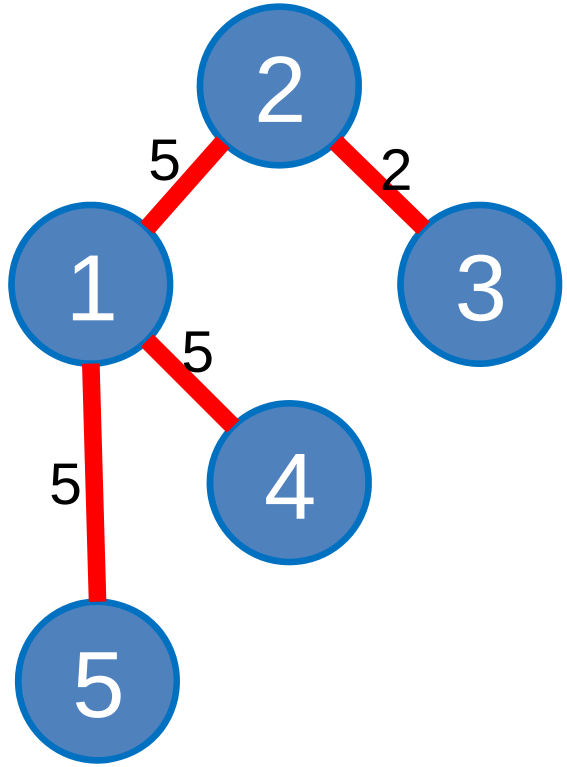
But Kruskal's algorithm will still continue



However, it will not modify anything else



This is the final MST with cost 17



Note: We can actually stop here.
Question: How?

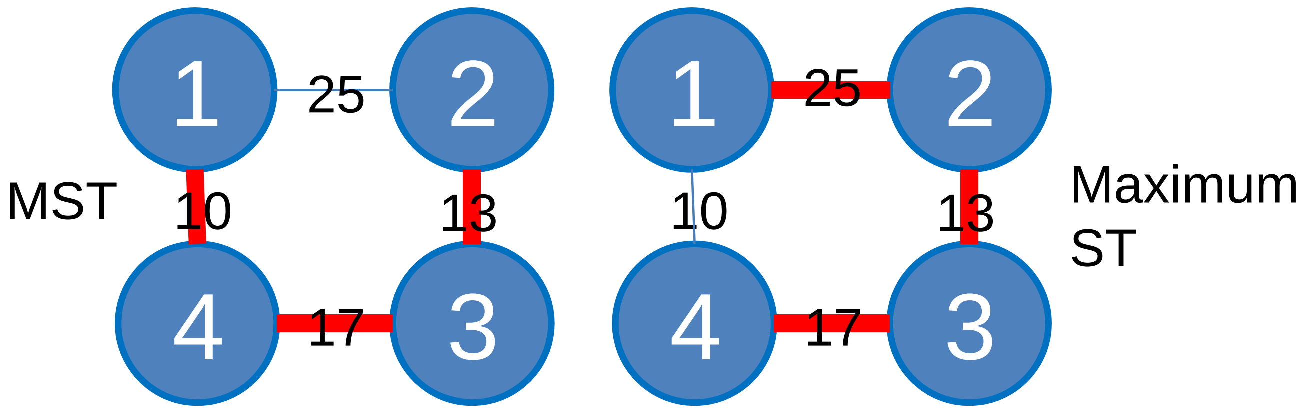
Kruskal's Algorithm (Sample Code)

```
// sorted by edge cost, PQ default: sort descending :(
priority_queue< pair<int, ii> > EdgeList;
// trick: store (negative) weight(i, j) and pair(i, j)
// i.e. EdgeList.push(make_pair(-weight, make_pair(i, j)));

mst_cost = 0; initSet(V); // all V are disjoint initially
while (!EdgeList.empty()) { // while ∃ more edges
  pair<int, ii> front = EdgeList.top(); EdgeList.pop();
  if (!isSameSet(front.second.first, front.second.second)) {
    // if adding e to MST does not form a cycle
    mst_cost += (-front.first); // add –weight of e to MST
    unionSet(front.second.first, front.second.second);
  }
}
```

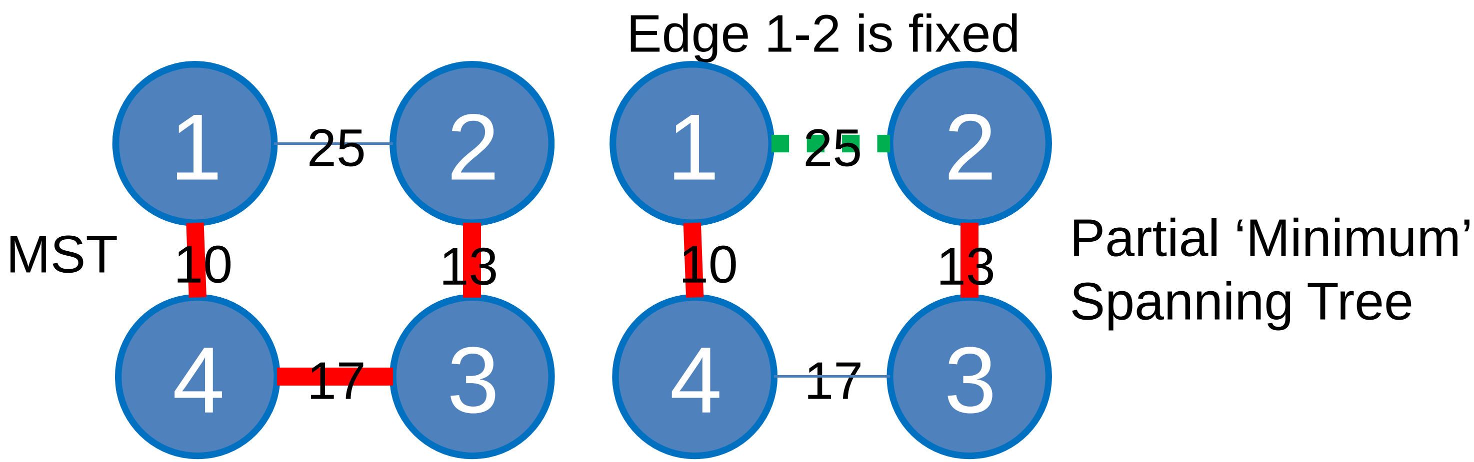
MST Variants (1)

- Variants of basic MST problem are interesting!
  1. Maximum ST (LA 4110)
  - Instead of minimum, we want maximum ST
  - Solution: Reverse the sort order in Kruskal's algorithm!



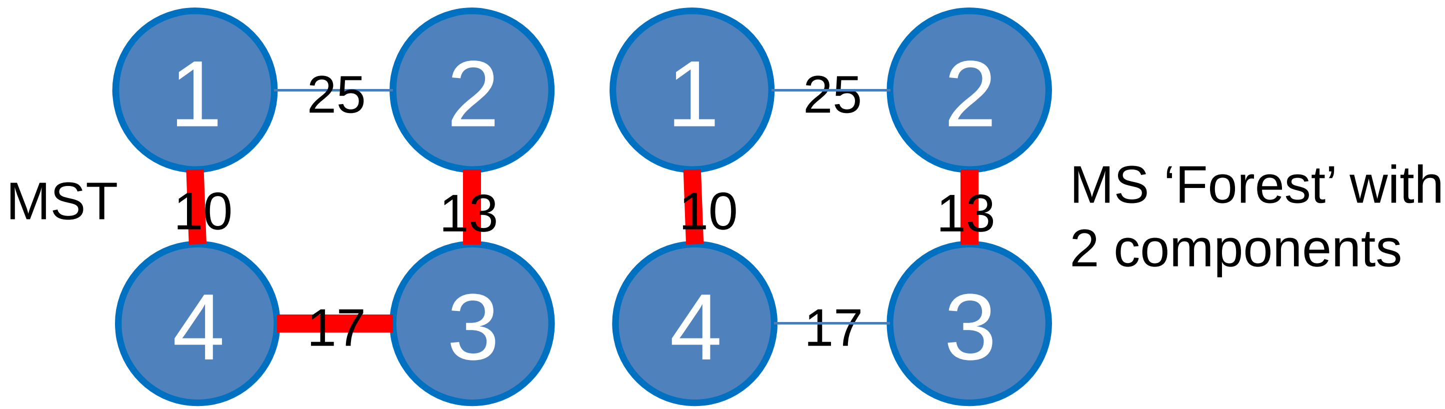
MST Variants (2)

2. Partial 'Minimum' ST (UVa: 10147, 10397)
- Some edges are **fixed**
  - Must be taken as part of the Spanning Tree
- We need to continue building the "M"ST
  - The resulting Spanning Tree perhaps no longer minimum
- Solution: Use Kruskal's algorithm to continue!



MST Variants (3)

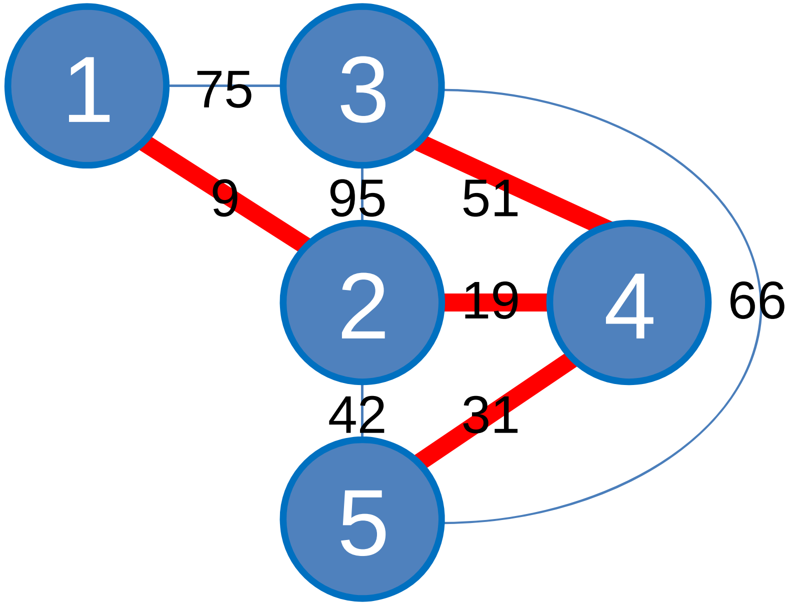
3. Minimum Spanning Forest (UVa: [10369]())
   - Spanning: All vertices must be covered
     - But we can stop even though the Spanning Tree has not been formed as long as the spanning criteria is satisfied!
   - The desired number of components is told beforehand
     - The result is a “forest”
   - Solution: Use Kruskal’s algorithm again, stop when number of connected component = desired number



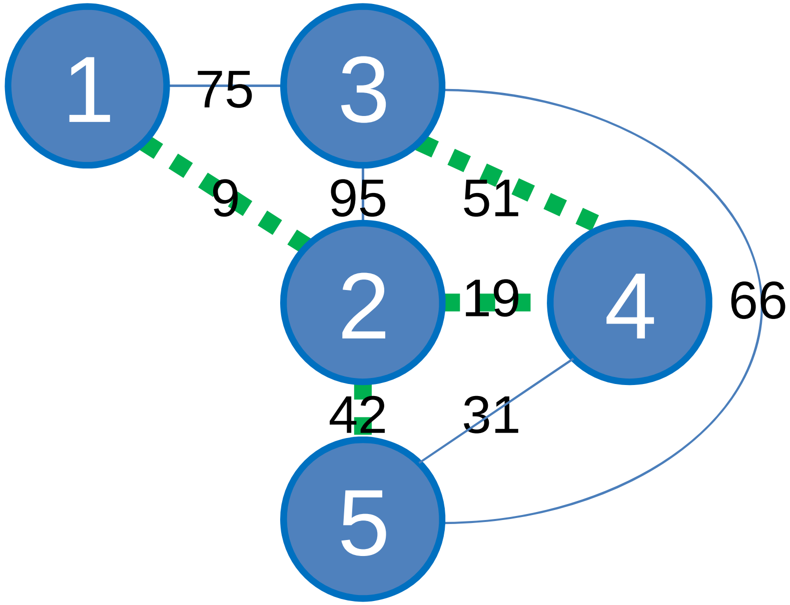
MST Variants (4)

4. Second best ST (UVa: 10600, PKU: 1679), e.g.:

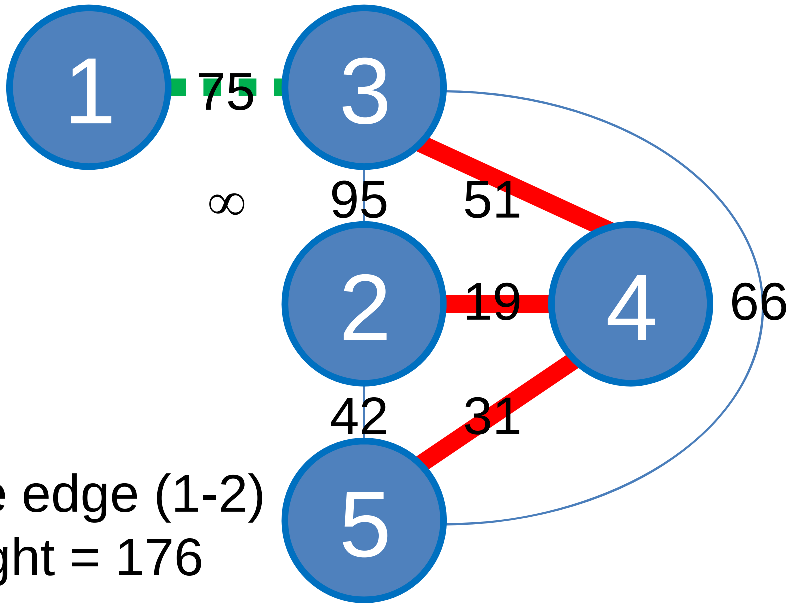
MST (red edges)
Weight = 9+19+51+31 = 110



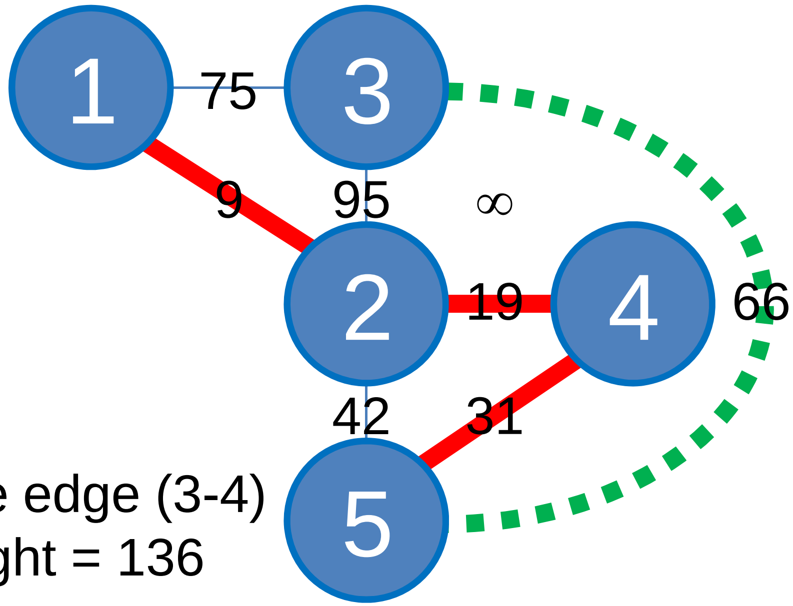
Second best ST (green edges)
Weight = 9+19+51+42 = 121



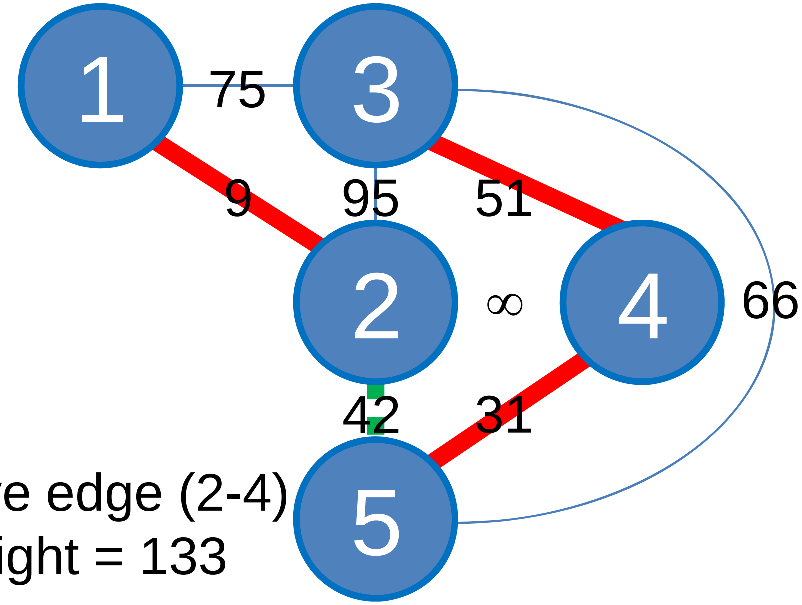
2^{nd} best MST is always MST with 2 edges difference.
1 edge is taken out from MST, another **chord^** edge is added to MST!
In this example: edge (4-5) → taken out and (2-5) → added in.



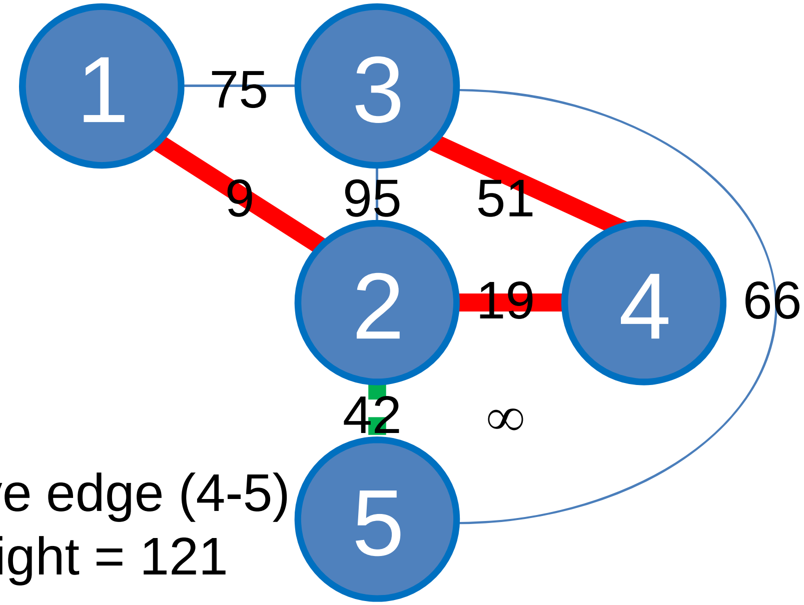
Remove edge (1-2)
ST Weight = 176



Remove edge (3-4)
ST Weight = 136



Remove edge (2-4)
ST Weight = 133



Remove edge (4-5)
ST Weight = 121

Simple solution^: Sort edges O(E log E), find MST using Kruskal in O(E), then for each edge in MST, make its weight INF ("delete"), and try to find the (2nd best) MST O(**V**E).
It is V*E because |E| in MST is |V|-1

Summary

- Today, we have gone through various well-known graph problems & algorithms
  - Depth First Search and (lots of) variants
  - Breadth First Search for SSSP on unweighted graph
  - Kruskal's for MST and (lots of) variants