This course material is now made available for public usage.
Special acknowledgement to School of Computing, National University of Singapore
for allowing Steven to prepare and distribute these teaching materials.

CS3233 Competitive Programming

Dr. Steven Halim

Week 05 – Graph 1 (“Basics”)

Outline (1)

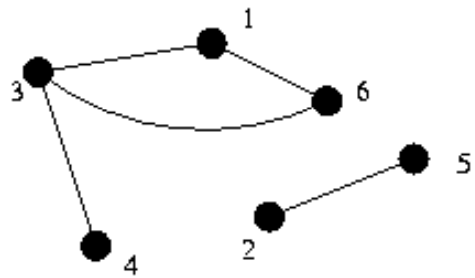
- Mini Contest #4 + Break + Discussion
- Admins
- CS2010/2020 Reviews (not discussed in details except red ones)
  - Graph: Preliminary & Motivation
  - Graph Traversal Algorithms
    - DFS/BFS: Connected Components/Flood Fill
    - DFS only: Toposort/Cut Vertex+Bridges/Strongly Connected Components
  - Minimum Spanning Tree Algorithm
    - Kruskal's and Prim's: Plus Various Applications

Outline (2)

- CS2010/2020 Reviews (not discussed in details except red ones)
  - Single-Source Shortest Paths
    - BFS: SSSP on Unweighted graph/Variants
    - Dijkstra's: SSSP on Weighted (no -ve cycle) graph/Variants
  - All-Pairs Shortest Paths
    - Floyd Warshall's + Variants
  - Special Graphs Part 1
    - Tree, Euler Graph, Directed Acyclic Graph

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
- Directed Acyclic Graph
- Tree/Forest
- Euler/Hamiltonian Path/Cycle
- Bipartite Graph



Kaohsiung 2006

Standings & Felix's Analysis

A.	Print Words in Lines	A.	DP
B.	The Bug Sensor Problem	B.	Graph, MST
C.	Pitcher Rotation	C.	DP + memory reduction
D.	Lucky and Good Months…	D.	Tedious Ad Hoc
E.	Route Planning	E.	Search?
Shift Cipher (N/A in Live Archive)		Complete Search + STL	
F.	A Scheduling Problem	F.	Greedy?
G.	Check the Lines	G.	?
H.	Perfect Service	H.	DA Graph, DP on Tree

Singapore 2007

Standings & Felix's Analysis

A.	MODEX
B.	JONES
C.	ACORN
D.	TUSK
E.	SKYLINE
F.	USHER
G.	RACING

A.	Math, Modulo Arithmetic
B.	DP
C.	DP + memory reduction
D.	Geometry
E.	DS, Segment Tree
F.	Graph, APSP, Min Weighted Cycle++
G.	Graph, Maximum Spanning Tree

Jakarta 2008

Standings & Suhendry's Analysis

A.	Anti Brute Force Lock	A.	Graph, MST
B.	Bonus Treasure	B.	Recursion
C.	Panda Land 7: Casino Island	C.	Trie?
D.	Disjoint Paths	D.	DA Graph, DP on Tree
E.	Expert Enough?	E.	Complete Search (is enough)
F.	Free Parentheses	F.	DP, harder than problem I
G.	Greatest K-Palindrome Sub…	G.	String
H.	Hyper-Mod	H.	Math
I.	ICPC Team Strategy	I.	DP, medium
J.	Jollybee Tournament	J.	Ad Hoc, Simulation

Kuala Lumpur 2008

A.	ASCII Diamondi	A.	Ad Hoc
B.	Match Maker	B.	DP, Stable Marriage Problem?
C.	Tariff Plan	C.	Ad Hoc
D.	Irreducible Fractions	D.	Math
E.	Gun Fight	E.	Graph, MCBM (AlternatingPath)
F.	Unlock the Lock	F.	Graph, SSSP (BFS)
G.	Ironman Race in Treeland	G.	DA Graph, Likely DP on Tree?
H.	Shooting the Monster	H.	Comp Geo
I.	Addition-Substraction Game	I.	?
J.	The Great Game	J.	?
K.	Triangle Hazard	K.	Math

Daejeon 2010

A.	Sales	A.	Brute Force
B.	String Popping	B.	Recursive Backtracking
C.	Password	C.	Recursive Backtracking
D.	Mines	D.	Geometry + SCCs (Graph)
E.	Binary Search Tree	E.	Graph, BST, Math, Combinatoric
F.	Tour Belt	F.	Graph, MST (modified)
G.	String Phone	G.	?
H.	Installations	H.	?
I.	Restaurant	I.	?
J.	KTX Train Depot	J.	?

Depth-First Search (DFS)

Breadth-First Search (BFS)

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

GRAPH TRAVERSAL ALGORITHMS

Motivation (1)

- How to solve these UVa problems:
  - 469 (Wetlands of Florida)
    - Similar problems: 260, 352, 572, 782, 784, 785, etc
  - 11504 (Dominos)
    - Similar problems: 1263, 11709, 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

```
typedef pair<int, int> ii; typedef vector<ii> vi;

void dfs(int u) { // DFS for normal usage
  printf(" %d", u); // this vertex is visited
  dfs_num[u] = DFS_BLACK; // mark as visited
  for (int j = 0; j < (int)AdjList[u].size; j++) {
    ii v = AdjList[u][j]; // 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
  }
}
```

Breadth First Search (using STL)

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

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

while (!q.empty()) {
  int u = q.front(); q.pop(); // queue: layer by layer!
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    ii v = AdjList[u][j]; // for each neighbours of u
    if (!dist.count(v.first)) {
      dist[v.first] = dist[u] + 1; // unvisited + reachable
      q.push(v.first); // enqueue v.first for next steps
    }
  }
}
```

1st Application: Connected Components

- DFS (and BFS) 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);
```

Finding Topological Sort (see text book/CS2010/CS2020)

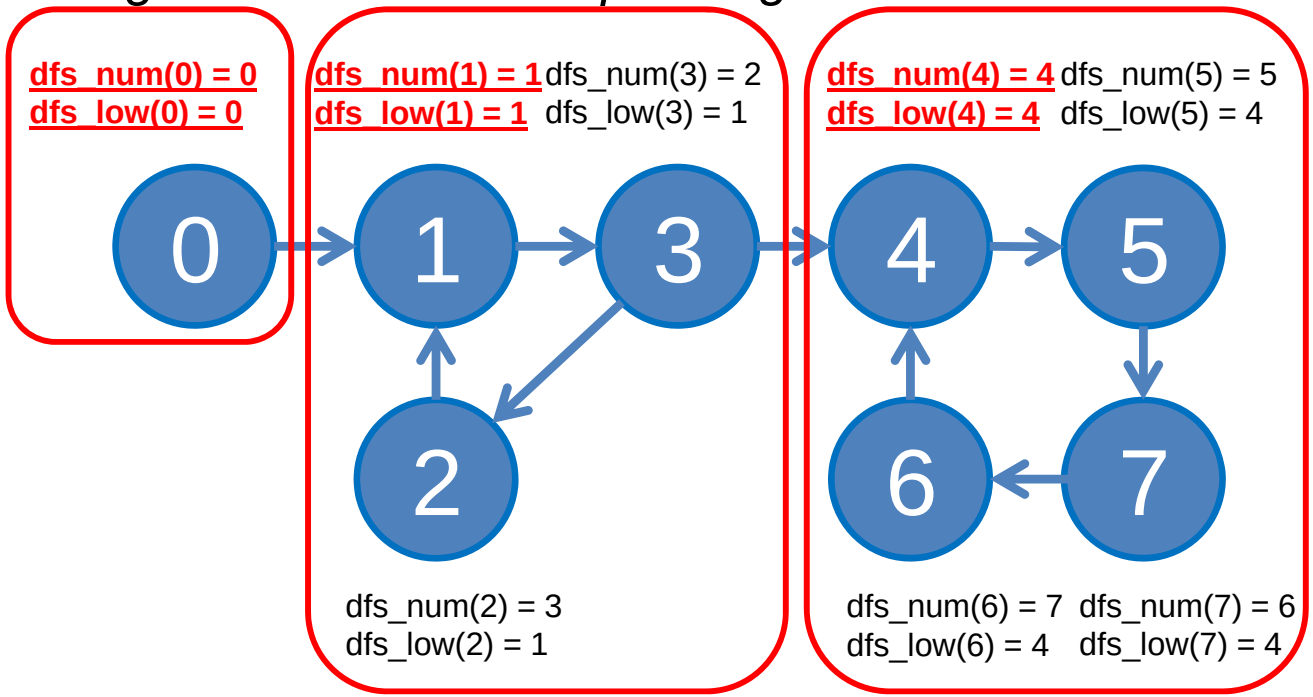
Finding Articulation Points and Bridges (see text book)

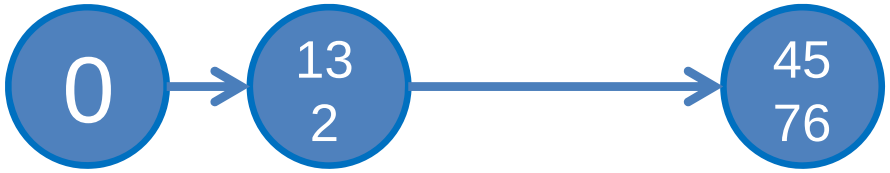
Finding Strongly Connected Component

TARJAN'S DFS ALGORITHMS

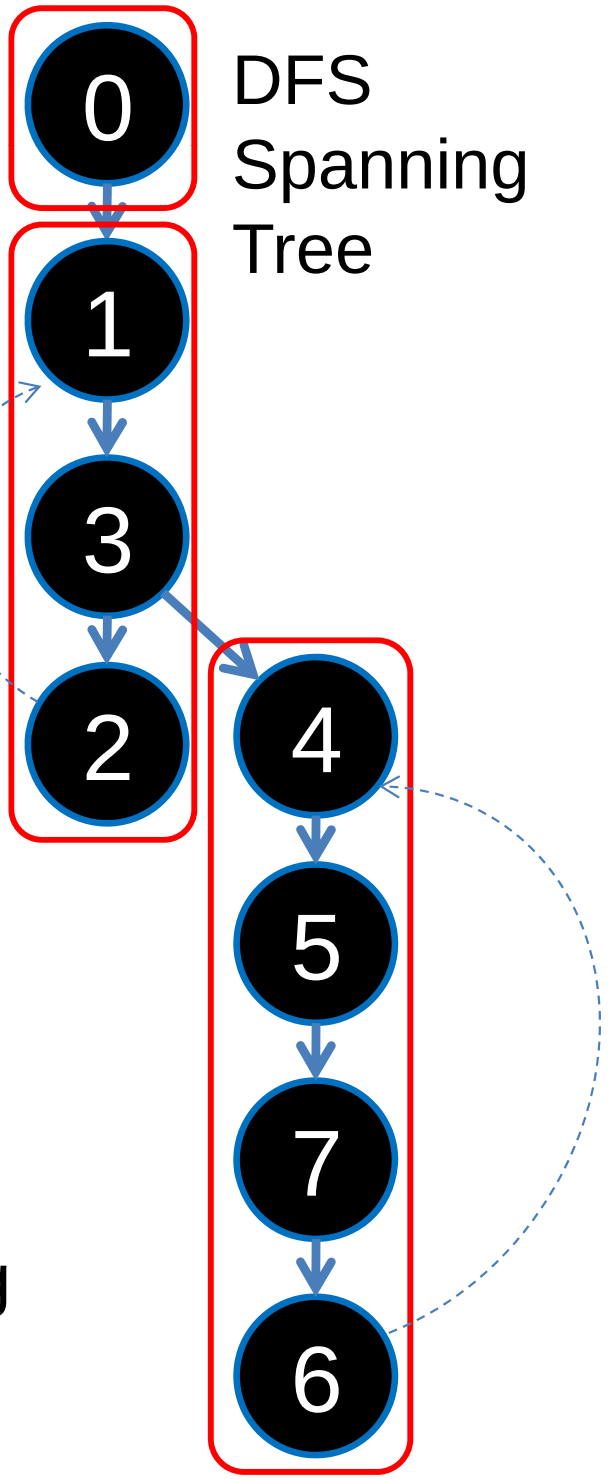
Tarjan's SCC

Input: A Directed Graph
dfs_num: visitation counter
dfs_low: lowest dfs_num reachable from that vertex
*using the **current** DFS spanning tree*





DAG after contracting SCCs



Code: Tarjan's SCC (not in IOI syllabus)

```
vi dfs_num, dfs_low, S, visited; // global variables

void tarjanSCC(int u) {
  dfs_low[u] = dfs_num[u] = dfsNumberCounter++; // dfs_low[u] <= dfs_num[u]
  S.push_back(u); // stores u in a vector based on order of visitation
  visited[u] = 1;
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    ii = AdjList[u][j];
    if (dfs_num[v.first] == DFS_WHITE) // a tree edge
      tarjanSCC(v.first);
    if (visited[v.first]) // 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 (start) of an SCC
    printf("SCC %d: ", ++numSCC); // this part is done after recursion
    while (1) {
      int v = S.back(); S.pop_back(); visited[v] = 0;
      printf(" %d", v);
      if (u == v) break;
    }
    printf("\n");
  }
}
```

Graph Traversal Comparison

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

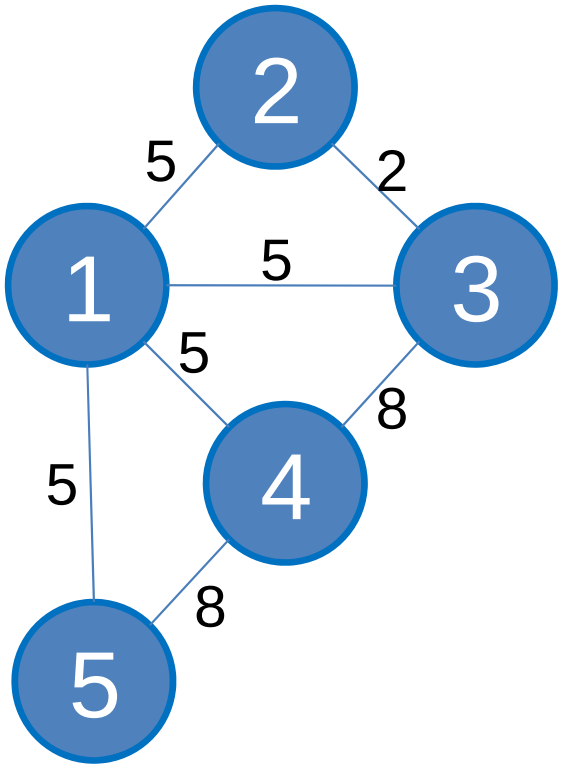
- BFS
- Pros:
  - Can solve SSSP on unweighted graphs (discussed later)
- Cons:
  - Slightly longer to code
  - Use more memory

Prim's algorithm → Read textbook on your own ☺
(or revise CS2010/CS2020 material)

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!



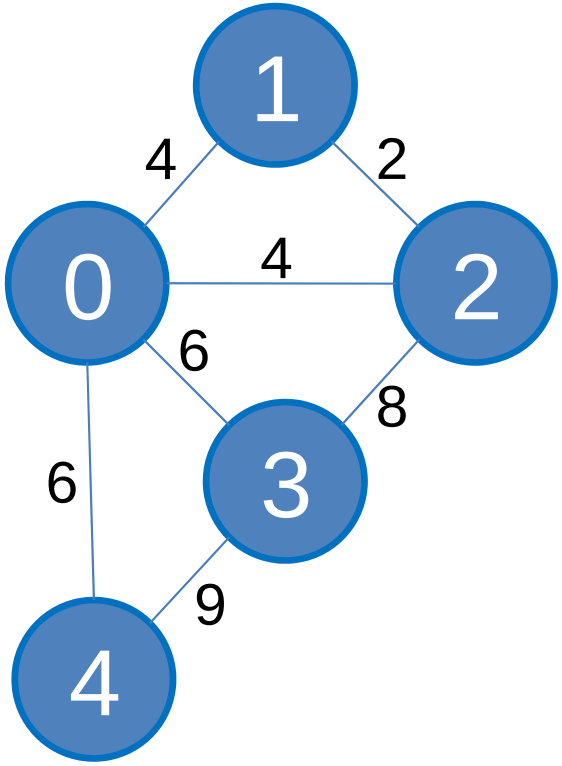
- MST!

Spanning Tree & MST

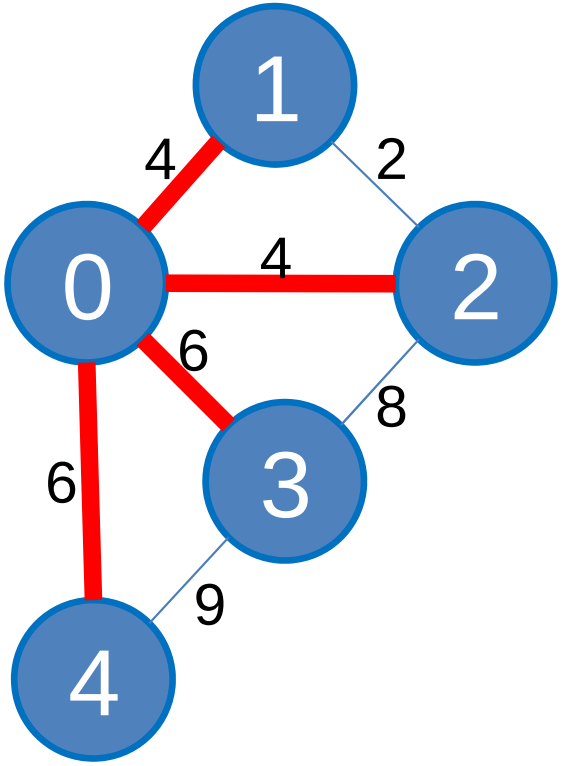
- Given a **connected undirected** graph G, select E $\in$ G such that a tree is formed and this tree **spans** (covers) all V $\in$ G!
  - No cycles or loops are formed!
- There can be **several** spanning trees in G
  - The one where total cost is minimum is called the **Minimum** Spanning Tree (**MST**)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

Example

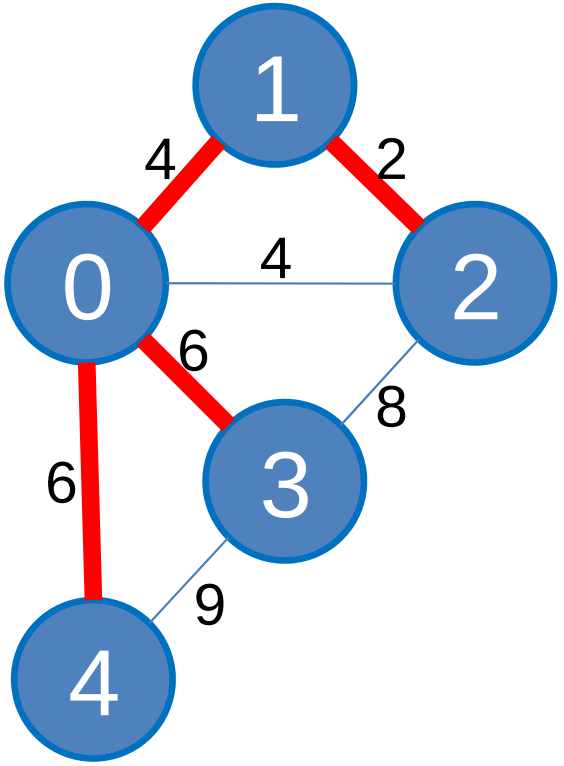
The Original Graph



A Spanning Tree
Cost: 4+4+6+6 = 20



An MST
Cost: 4+6+6+2 = 18



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

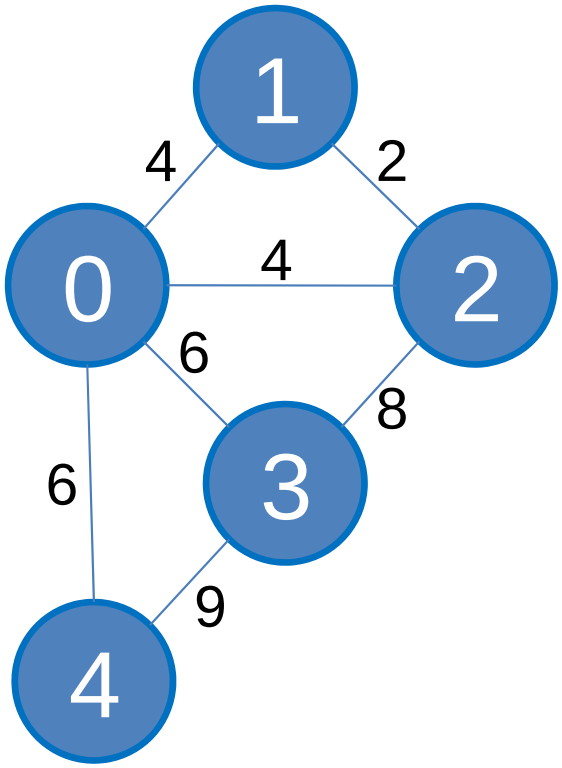
- In my opinion, Kruskal's algorithm is simpler

```
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
```

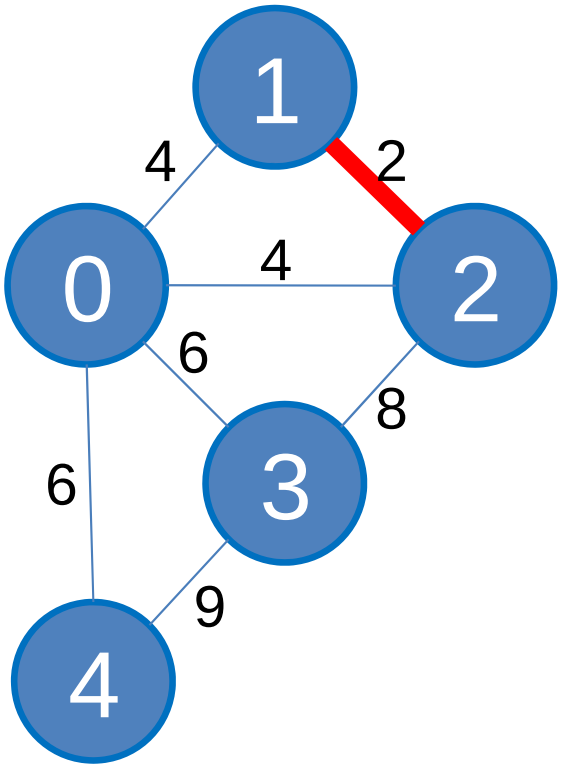
  - Simply store the edges in an array of Edges (EdgeList) and sort them, or use Priority Queue
  - Test for cycles using Disjoint Sets (Union Find) DS

Kruskal's Animation (1)

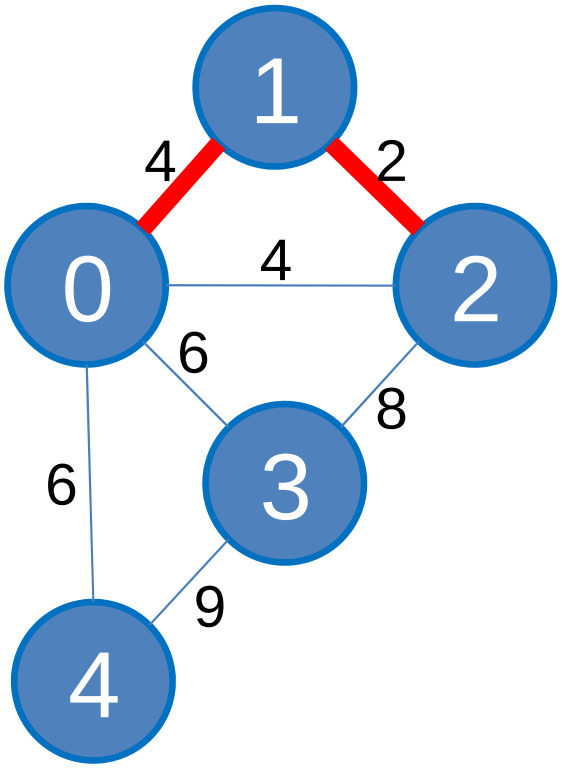
The original graph,
no edge is selected



Connect 1 and 2
As this edge is smallest



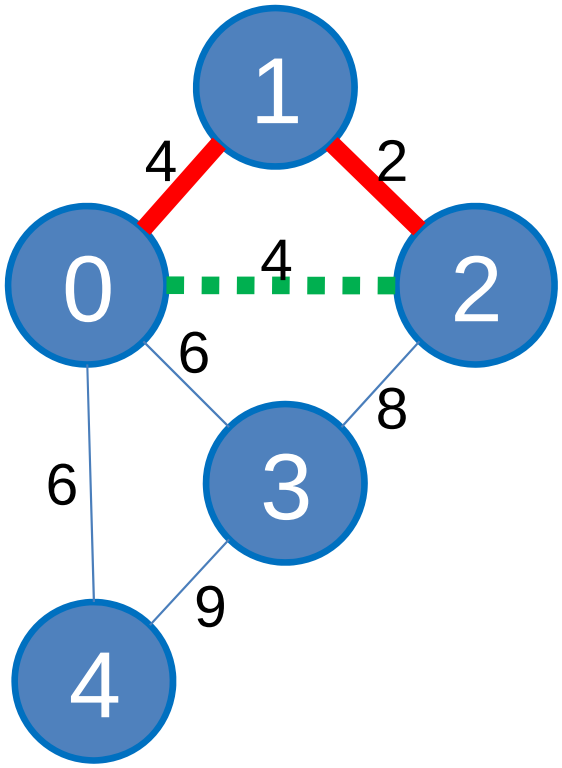
Connect 1 and 0
No cycle is formed



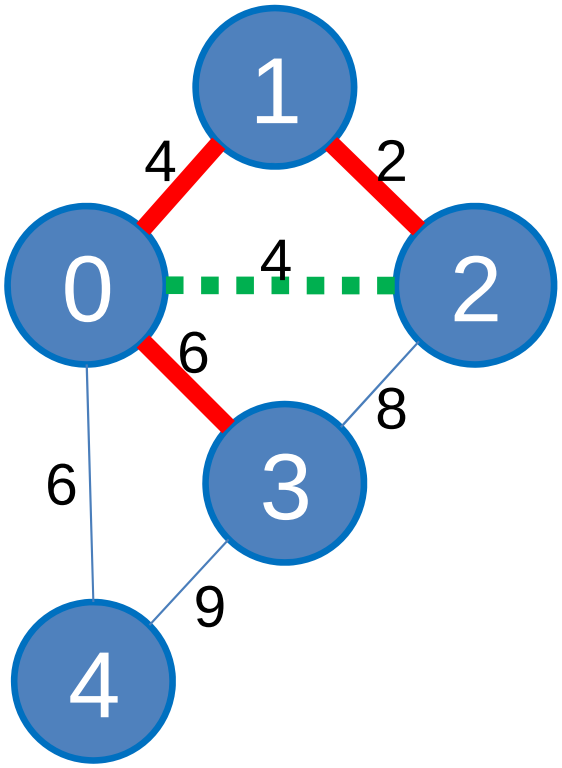
Note: The sorted order of the edges determines how the MST formed. Observe that we can also choose to connect vertex 2 and 0 also with weight 4!

Kruskal's Animation (2)

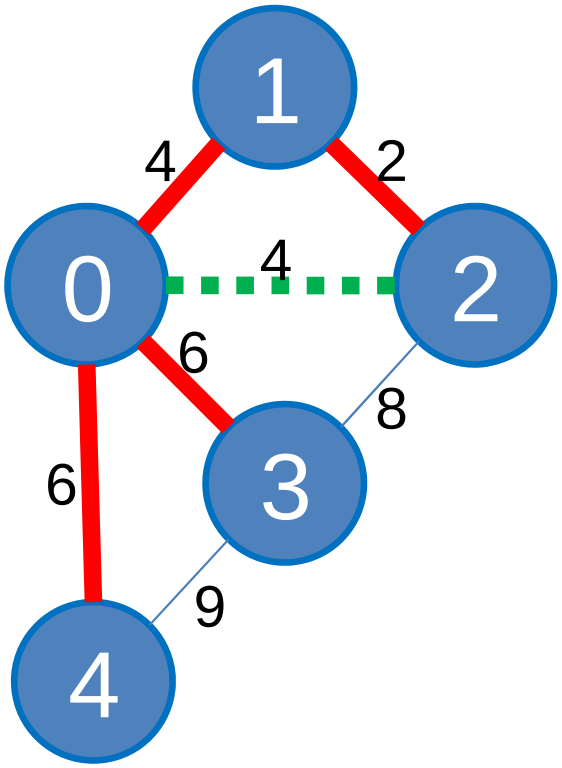
Cannot connect 0 and 2
As it will form a cycle



Connect 0 and 3
The next smallest edge



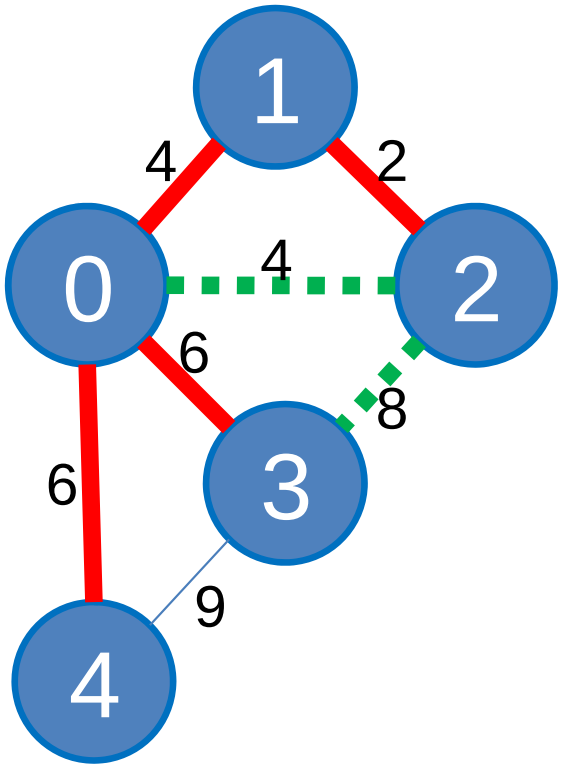
Connect 0 and 4
MST is formed…



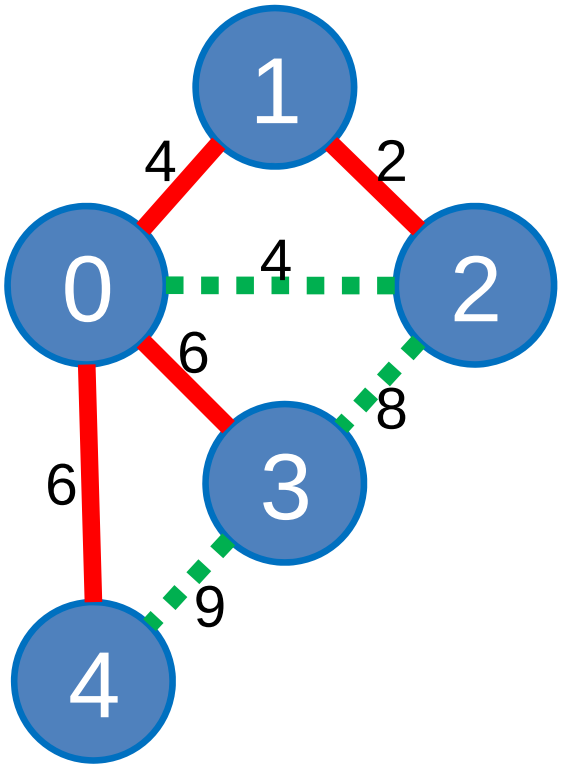
Note: Again, the sorted order of the edges determines how the MST formed; Connecting 0 and 4 is also a valid next move

Kruskal's Animation (3)

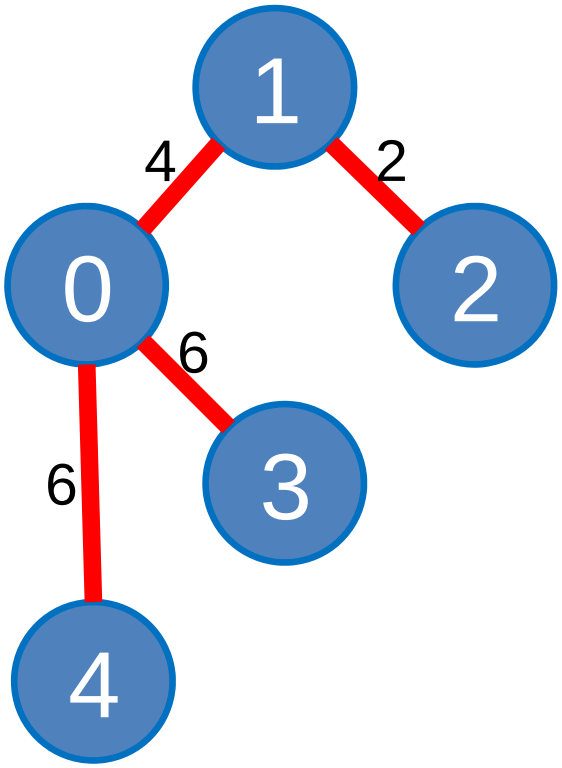
But (standard) Kruskal's algorithm will still continue



However, it will not modify anything else



This is the final MST with cost 18



Kruskal's Algorithm (Sample Code)

```
// sorted by edge cost
vector< pair<int, ii> > EdgeList;
// insert edges in format (weight, (u, v)) to EdgeList
sort(EdgeList.begin(), Edgelist.End();

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

But…

- You have not teach us Union Find DS in CS3233??
  - It is also only covered briefly in CS2010/CS2020
- Yeah, we choose to skip that DS in CS3233…
- If you want to solve MST problems,
  learn Union Find DS on your own (Sec 2.3.2)
- To be fair, I will **not** set any MST problems in CS3233 mini contests problems **A + B** ☺
  - I do not say anything about problem C or mid/final contest

BFS (unweighted)

Dijkstra's (non –ve cycle)

Bellman Ford's (may have –ve cycle), not discussed

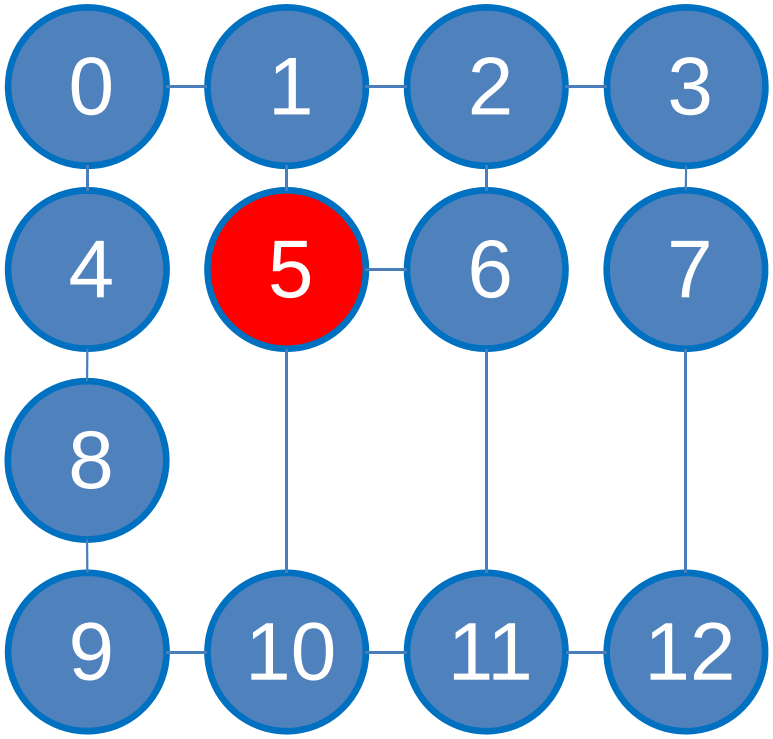
Floyd Warshall's (all-pairs

SHORTEST PATHS

BFS for **Special Case** SSSP

- SSSP is a classical problem in Graph theory:
  - Find shortest paths from **one source** to the rest^
- Special case: 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

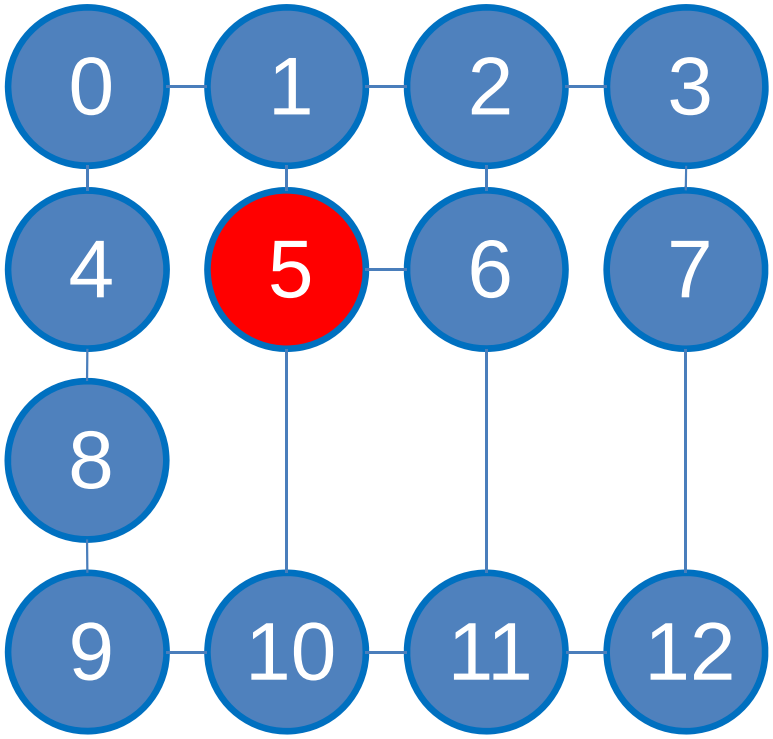
Example (1)



Q = {5}

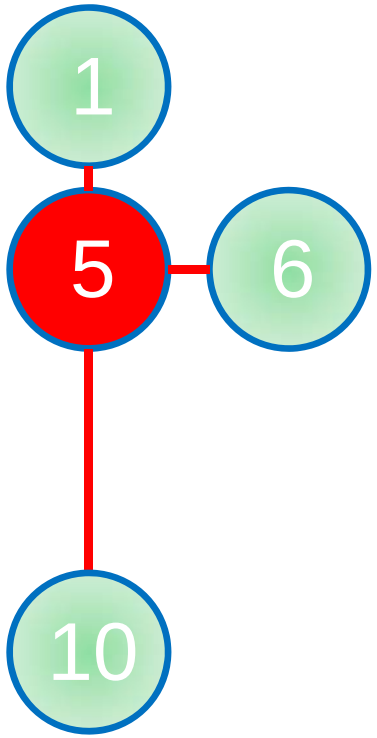
D[5] = 0

Example (2)

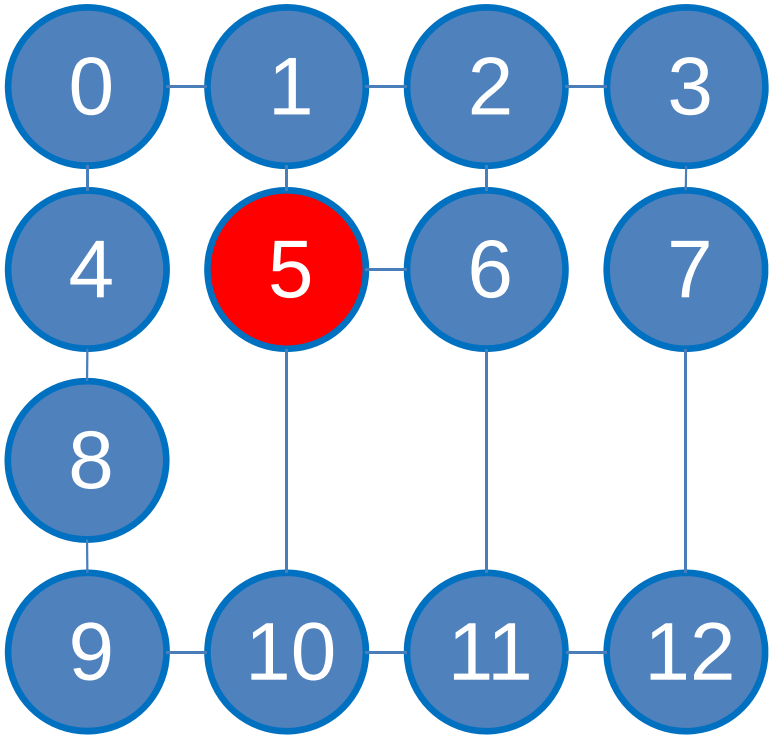


Q = {5}
Q = {1, 6, 10}

D[5] = 0
D[1] = D[5] + 1 = 1
D[6] = D[5] + 1 = 1
D[10] = D[5] + 1 = 1

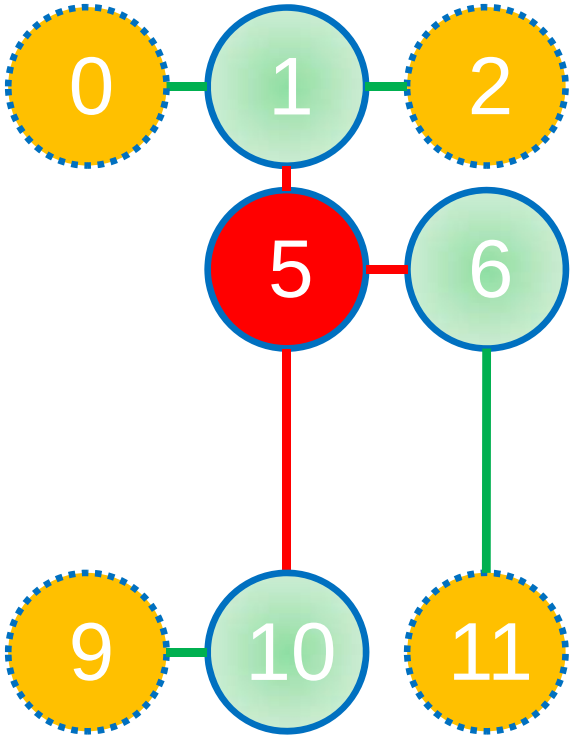


Example (3)

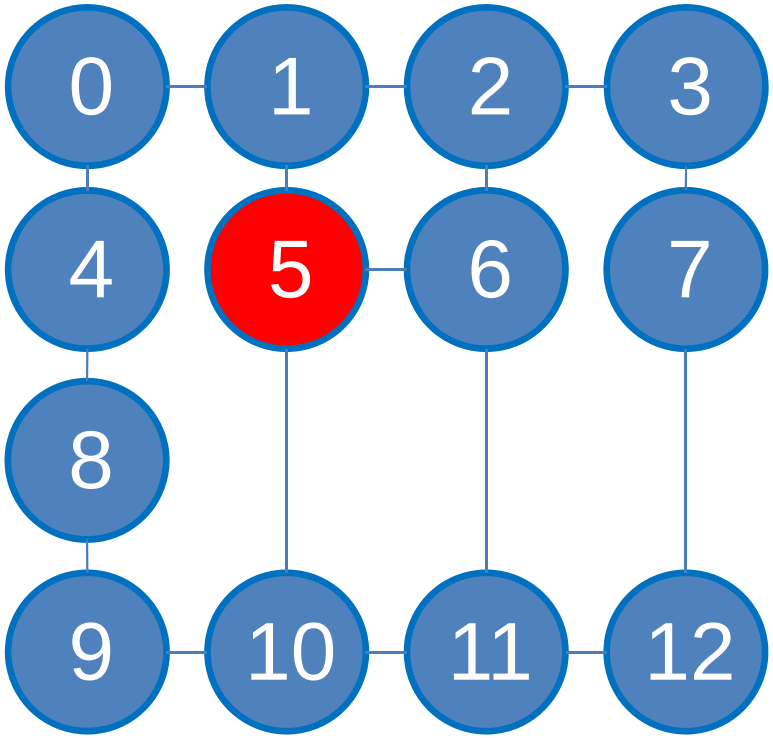


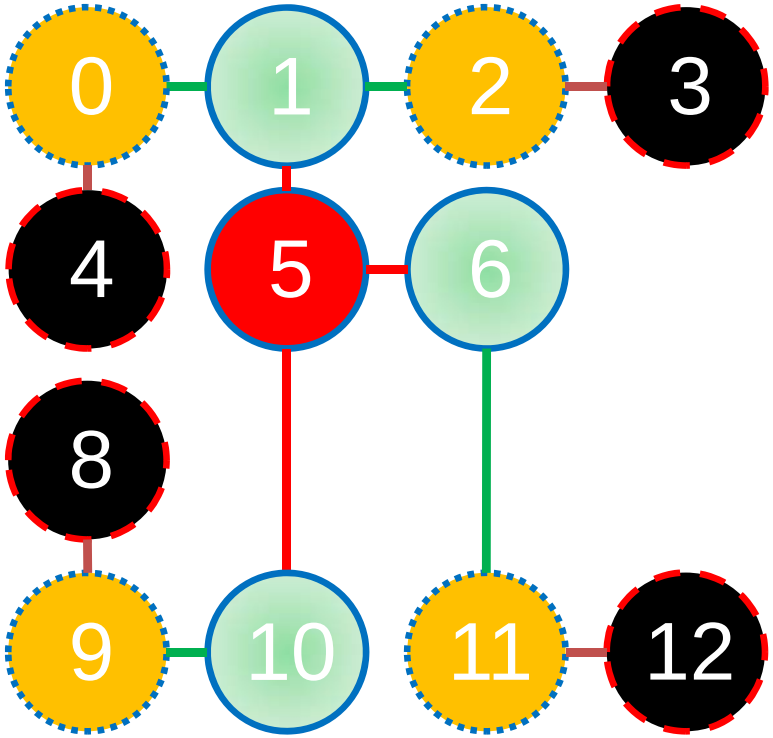
Q = {5}
Q = {1, 6, 10}
Q = {6, 10, **0, 2**}
Q = {10, 0, 2, **11**}
Q = {0, 2, 11, **9**}

D[5] = 0
D[1] = D[5] + 1 = 1
D[6] = D[5] + 1 = 1
D[10] = D[5] + 1 = 1
D[0] = D[1] + 1 = 2
D[2] = D[1] + 1 = 2
D[11] = D[6] + 1 = 2
D[9] = D[10] + 1 = 2



Example (4)

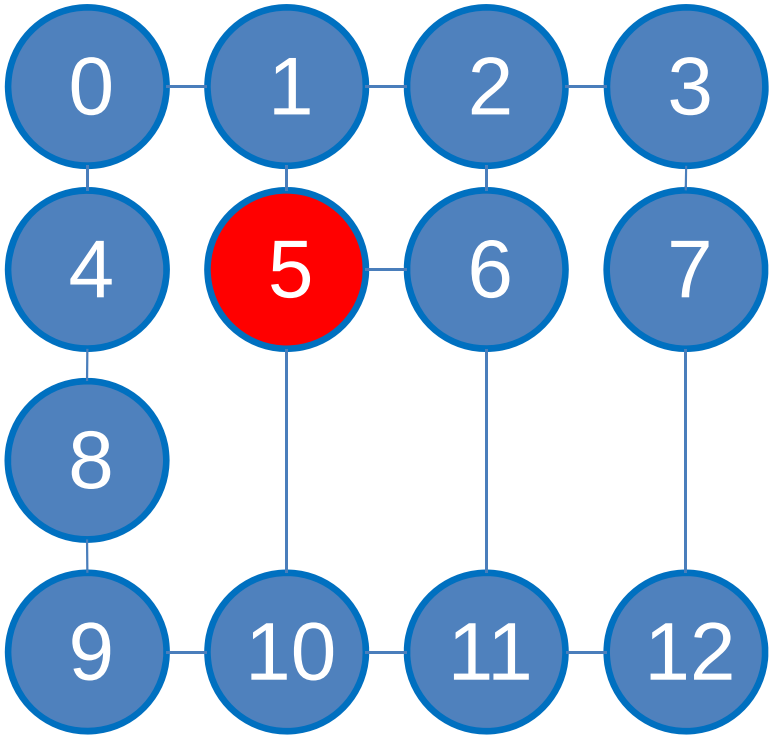


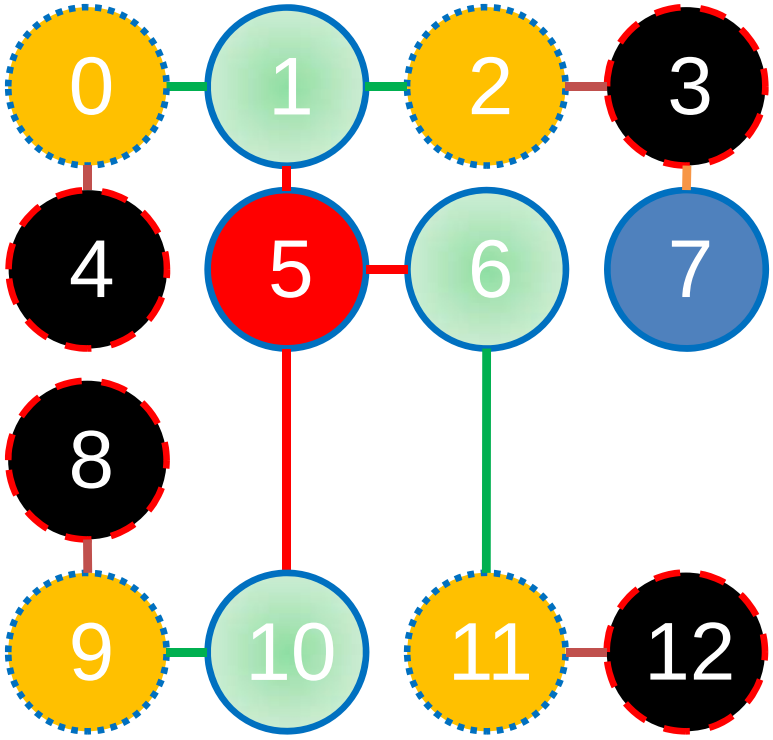


Q = {5}
Q = {1, 6, 10}
Q = {6, 10, **0, 2**}
Q = {10, 0, 2, **11**}
Q = {0, 2, 11, **9**}
Q = {2, 11, 9, **4**}
Q = {11, 9, 4, **3**}
Q = {9, 4, 3, **12**}
Q = {4, 3, 12, **8**}

D[5] = 0
D[1] = D[5] + 1 = 1
D[6] = D[5] + 1 = 1
D[10] = D[5] + 1 = 1
D[0] = D[1] + 1 = 2
D[2] = D[1] + 1 = 2
D[11] = D[6] + 1 = 2
D[9] = D[10] + 1 = 2
D[4] = D[0] + 1 = 3
D[3] = D[2] + 1 = 3
D[12] = D[11] + 1 = 3
D[8] = D[9] + 1 = 3

Example (5)





Q = {5}
Q = {1, 6, 10}
Q = {6, 10, **0, 2**}
Q = {10, 0, 2, **11**}
Q = {0, 2, 11, **9**}
Q = {2, 11, 9, **4**}
Q = {11, 9, 4, **3**}
Q = {9, 4, 3, **12**}
Q = {4, 3, 12, **8**}
Q = {3, 12, 8}
Q = {12, 8, **7**}
Q = {8, 7}
Q = {7}
Q = {}

D[5] = 0
D[1] = D[5] + 1 = 1
D[6] = D[5] + 1 = 1
D[10] = D[5] + 1 = 1
D[0] = D[1] + 1 = 2
D[2] = D[1] + 1 = 2
D[11] = D[6] + 1 = 2
D[9] = D[10] + 1 = 2
D[4] = D[0] + 1 = 3
D[3] = D[2] + 1 = 3
D[12] = D[11] + 1 = 3
D[8] = D[9] + 1 = 3
D[7] = D[3] + 1 = 4

This is the **BFS = SSSP ☺ spanning tree** when BFS is started from vertex 5

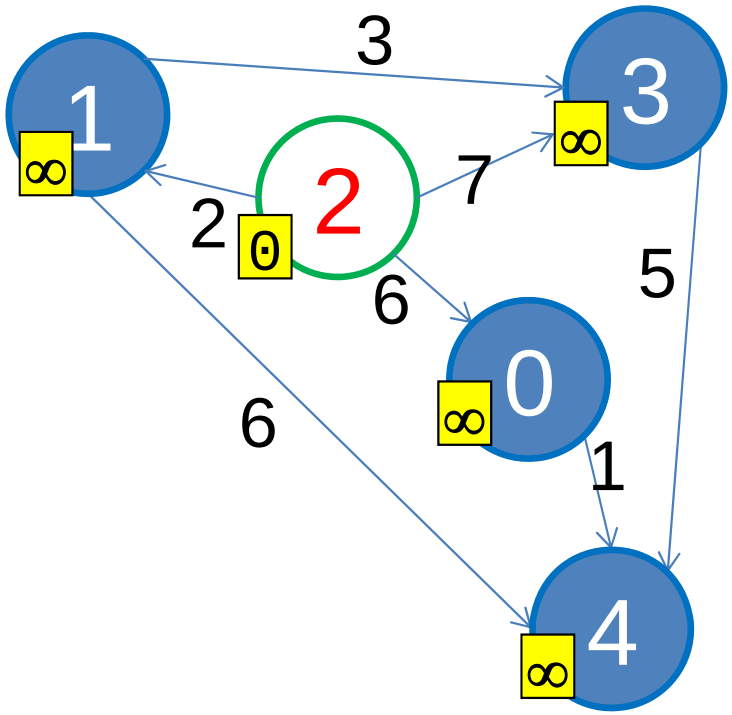
For SSSP on Weighted Graph but without Negative Weight Cycle

DIJKSTRA's

Single-Source Shortest Paths (1)

- If the graph is **un weighted**, we can use BFS
  - But what if the graph is **weighted**?
- UVa 341 (Non Stop Travel)
- Solution: Dijkstra O((V+E) log V)
  - A Greedy Algorithm
  - Use Priority Queue

Modified Dijkstra's – Example (1)

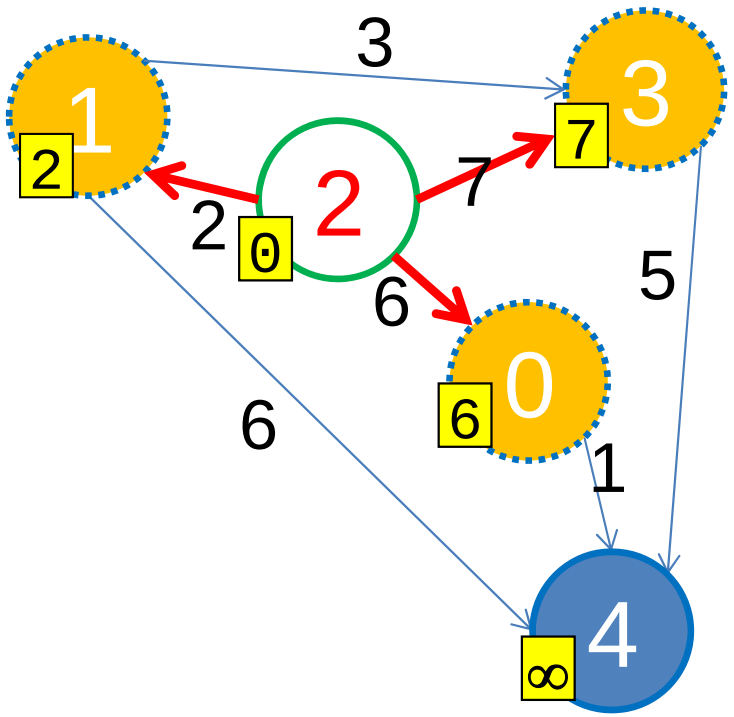


pq = {(0, 2)}

We store this pair of information to the priority queue: (D[vertex], vertex), sorted by increasing D[vertex], and then if ties, by vertex number

See that our priority queue is "clean" at the beginning of (modified) Dijsktra's algorithm, it only contains (0, the source s)

Modified Dijkstra's – Example (2)

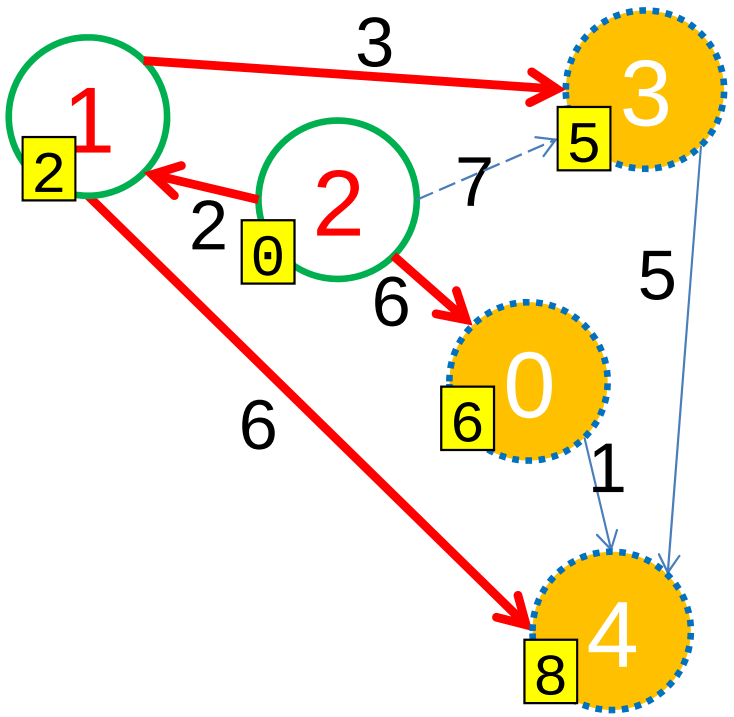


pq = {~~(0, 2)~~}
pq = {(2, 1), (6, 0), (7, 3)}

We greedily take the vertex in the front of the queue (here, it is vertex 2, the source), and then successfully relax all its neighbors (vertex 0, 1, 3).

Priority Queue will order these 3 vertices as 1, 0, 3, with shortest path estimate of 2, 6, 7, respectively.

Modified Dijkstra's – Example (3)



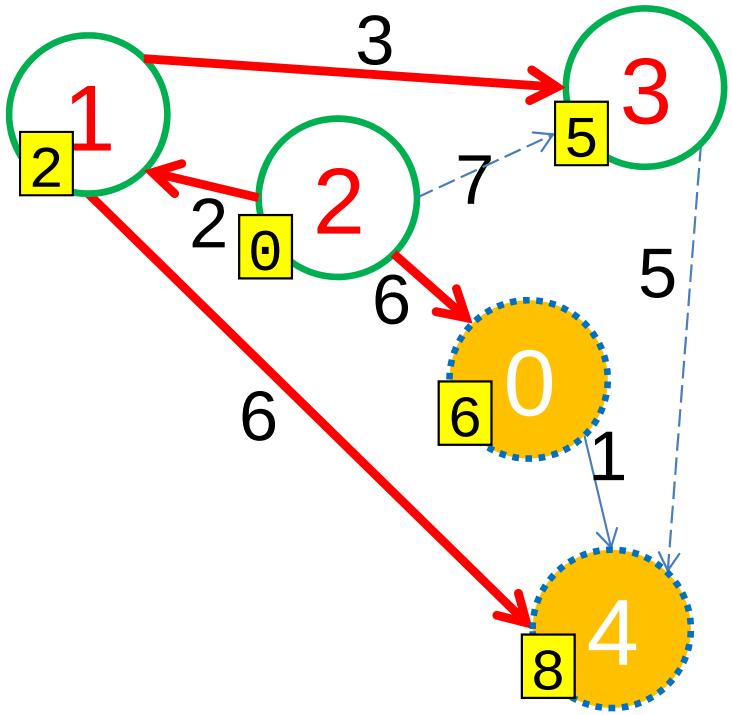
Vertex 3 appears twice in the priority queue, but this does not matter, as we will take only the first (smaller) one

pq = {~~(0, 2)~~}
pq = {~~(2, 1)~~, (6, 0), (7, 3)}
pq = {(5, 3), (6, 0), (7, 3), (8, 4)}

We greedily take the vertex in the front of the queue (now, it is vertex 1), then successfully relax all its neighbors (vertex 3 and 4).

Priority Queue will order the items as 3, 0, 3, 4 with shortest path estimate of 5, 6, 7, 8, respectively.

Modified Dijkstra's – Example (4)

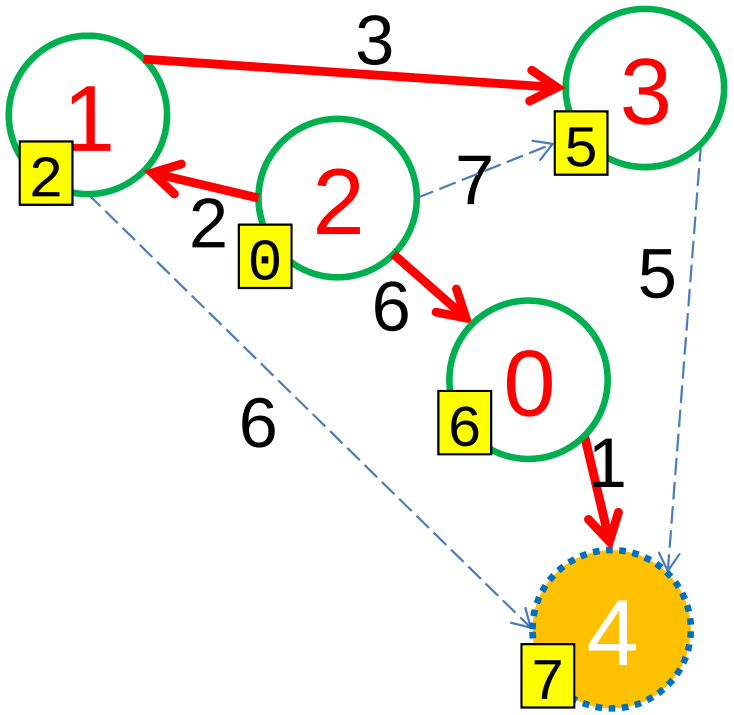


pq = {~~(0, 2)~~}
pq = {~~(2, 1)~~, (6, 0), (7, 3)}
pq = {~~(5, 3)~~, (6, 0), (7, 3), (8, 4)}
pq = {(6, 0), (7, 3), (8, 4)}

We greedily take the vertex in the front of the queue (now, it is vertex 3), then try to relax all its neighbors (only vertex 4). However D[4] is already 8. Since D[3] + w(3, 4) = 5 + 5 is worse than 8, we do not do anything.

Priority Queue will now have these items 0, 3, 4 with shortest path estimate of 6, 7, 8, respectively.

Modified Dijkstra's – Example (5)

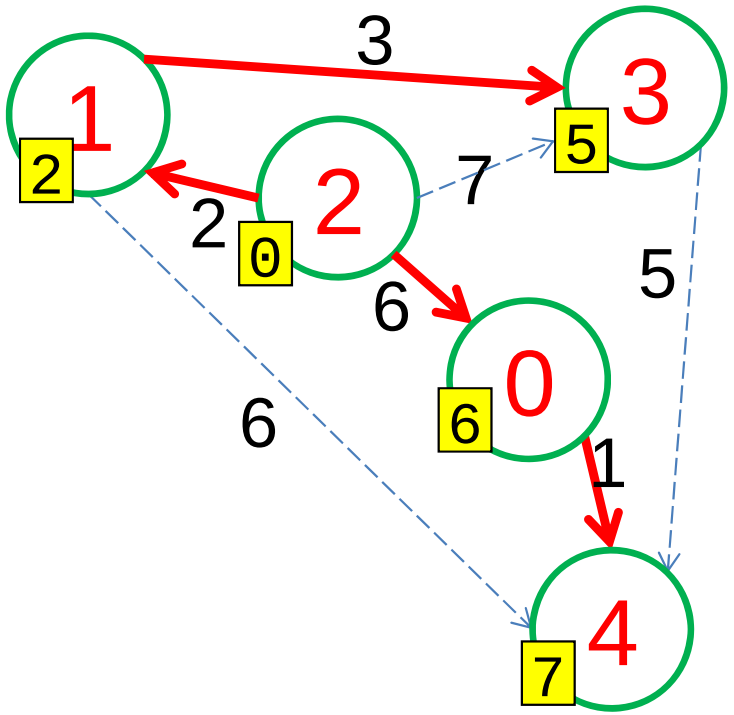


pq = {~~(0, 2)~~}
pq = {~~(2, 1)~~, (6, 0), (7, 3)}
pq = {~~(5, 3)~~, (6, 0), (7, 3), (8, 4)}
pq = {~~(6, 0)~~, (7, 3), (8, 4)}
pq = {(7, 3), (7, 4), (8, 4)}

We greedily take the vertex in the front of the queue (now, it is vertex 5), then successfully relax all its neighbors (only vertex 4).

Priority Queue will now have these items 3, 4, 4 with shortest path estimate of 7, 7, 8, respectively.

Modified Dijkstra's – Example (6)



Remember that vertex 3 appeared twice in the priority queue, but this Dijkstra's algorithm will only consider the first (shorter) one

pq = {~~(0, 2)~~}
pq = {~~(2, 1)~~, (6, 0), (7, 3)}
pq = {~~(5, 3)~~, (6, 0), (7, 3), (8, 4)}
pq = {~~(6, 0)~~, (7, 3), (8, 4)}
pq = {~~(7, 3)~~, (7, 4), (8, 4)}
pq = {~~(7, 4)~~, (8, 4)}
pq = {~~(8, 4)~~}
pq = {}

Similarly for vertex 4. The one with shortest path estimate 7 will be processed first and the one with shortest path estimate 8 will be ignored, although nothing is changed anymore

Dijkstra's Algorithm (using STL)

```
vi dist(V, INF); dist[s] = 0; // INF = 2B
priority_queue< ii, vector<ii>, greater<ii> > pq;
pq.push(ii(0, s)); // sort based on increasing distance
while (!pq.empty()) { // main loop
  ii top = pq.top(); pq.pop(); // greedy
  int d = top.first, u = top.second;
  if (d == dist[u]) {
    for (int j = 0; j < (int)AdjList[u].size(); j++) {
      ii  v = AdjList[u][j]; // all outgoing edges from u
      if (dist[u] + v.second < dist[v.first]) {
        dist[v.first] = dist[u] + v.second; // relax
        pq.push(ii(dist[v.first], v.first));
      } // enqueue this neighbor regardless it is
    }   // already in pq or not
  }
}
```

For All-Pairs Shortest Paths

FLOYD WARSHALL's

UVa 11463 – Commandos (1)

Al-Khawarizmi, Malaysia National Contest 2008

- Given:
  - A table that stores the amount of minutes to travel between buildings (there are **at most 100** buildings)
  - 2 special buildings: startB and endB
  - K soldiers to bomb all the K buildings in this mission
  - Each of them start at the same time from startB, choose one building B that has not been bombed by other soldier (bombing time negligible), and then gather in (destroyed) building endB.
- What is the minimum time to complete the mission?

UVa 11463 – Commandos (2)

Al-Khawarizmi, Malaysia National Contest 2008

- How long do you need to solve this problem?
- Solution:
  - The answer is determined by sp from starting building, detonate **furthest building**, and sp from that furthest building to end building
    - max(dist[start][i] + dist[i][end]) for all i $\in$ V
- How to compute **sp** for **many** pairs of vertices?

UVa 11463 – Commandos (3)

Al-Khawarizmi, Malaysia National Contest 2008

- This problem is called: All-Pairs Shortest Paths
- Two options to solve this:
  - Call SSSP algorithms multiple times
    - Dijkstra O(V * (V+E) * log V), if E = V^2 → O(V^3 log V)
    - Bellman Ford O(V * V * E), if E = V^2 → O(V^4)
    - Slow to code
  - Use Floyd Warshall, a clever **DP** algorithm
    - O(V^3) algorithm
    - Very easy to code!
    - In this problem, V is <= 100, so Floyd Warshall is DOABLE!!

Floyd Warshall – Template

- $O(V^3)$ since we have three nested loops!
- Use adjacency matrix: `G[MAX_V][MAX_V];`
  - So that weight of edge(i, j) can be accessed in O(1)

```
for (int k = 0; k < V; k++)
  for (int i = 0; i < V; i++)
    for (int j = 0; j < V; j++)
      G[i][j] = min(G[i][j], G[i][k] + G[k][j]);
```

  - See more explanation of this three-liner DP algorithm in CP

Tree, Euler Graph, Directed Acyclic Graph (basics)

DAG is also re-visited (next week, Week 06)

Bipartite Graph (Week 08)

SPECIAL GRAPHS (Part 1)

Special Graphs in Contest

- 4 special graphs frequently appear in contest:
  - Tree, keywords: connected, E = V-1, unique path!
  - Eulerian Graph, keywords: must visit each edge once
    - » Actually also rare now
  - Directed Acyclic Graph, keywords: no cycle
  - Bipartite, keywords: 2 sets, no edges within set!
    - » Currently not in IOI syllabus
- Some classical 'hard' problems may have *faster solution* on these special graphs
  - This allows problem setter to increase **input size**!
    - » Eliminates those who are not aware of the faster solution as solution for general graph is slower (TLE)
      or harder to code (slower to get AC)…

TREE

Tree

- Tree is a special Graph. It:
  - Connected
  - Has V vertices and exactly E = V - 1 edges
  - Has no cycle
  - Has one unique path between two vertices
  - Sometimes, it has one special vertex called “root” (rooted tree): Root has no parent
  - A vertex in n-ary tree has either {0, 1,…,n} children
    - n = 2 is called binary tree (most popular one)
  - Has many practical applications:
    - Organization Tree, Directories in Operating System, etc

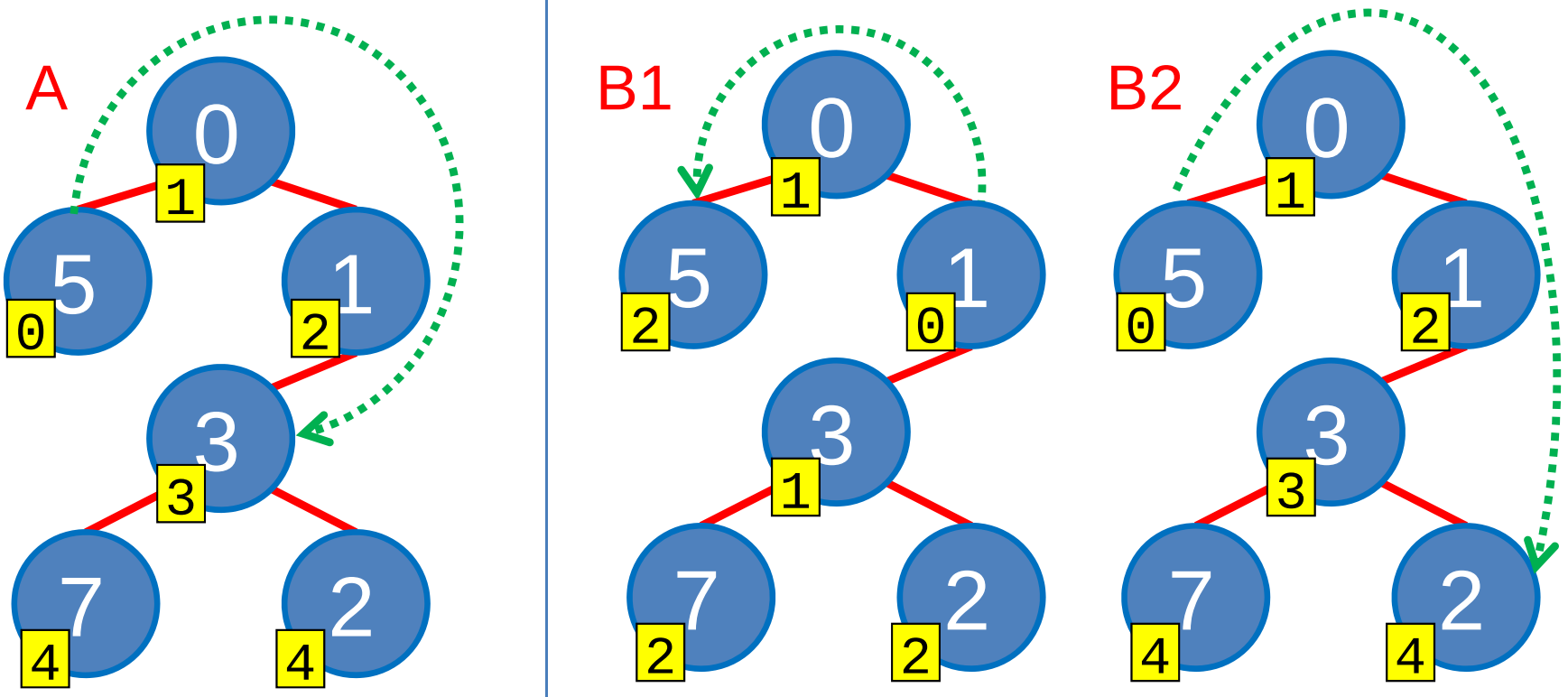
SSSP and APSP Problems
on Weighted Tree

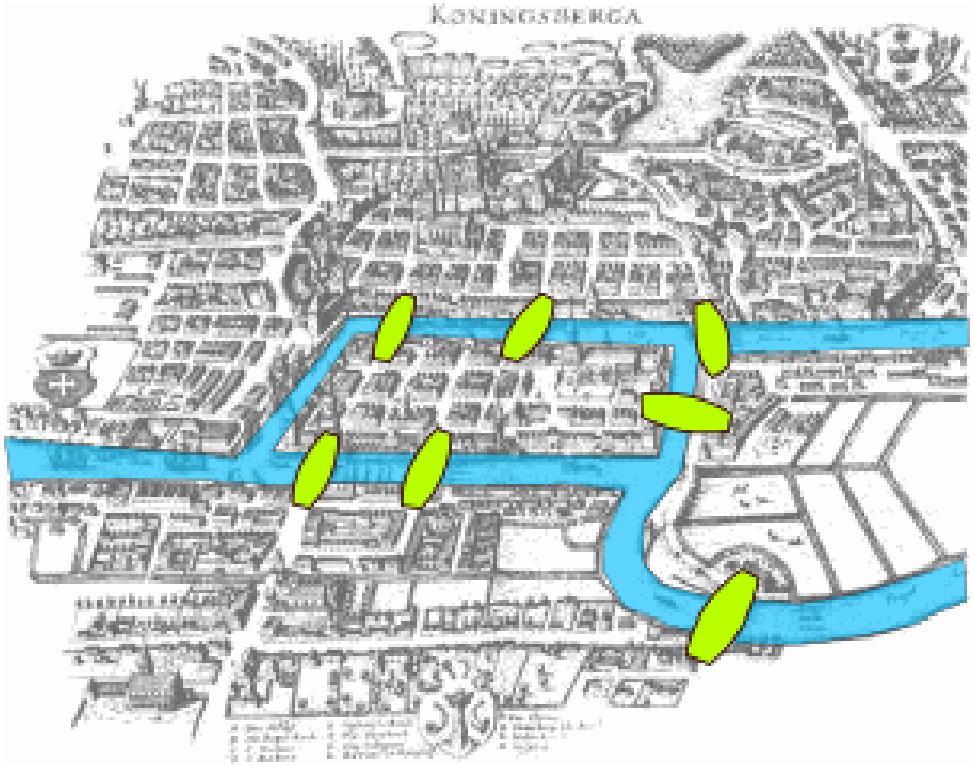
- In general weighted graph
  - SSSP problem: O((V+E) log V) Dijkstra's or O(VE) Bellman Ford's
  - APSP problem: $O(V^3)$ Floyd Warshall's
- In weighted tree
  - SSSP problem: O(V+E = V+V = V) DFS or BFS
    - There is only 1 unique path between 2 vertices in tree
  - APSP problem: simple V calls of DFS or BFS: $O(V^2)$
    - But can be made even faster using LCA... not covered

Diameter
of a Tree

- In general weighted graph
  - We have to run $O(V^3)$ Floyd Warshall's and pick the maximum over all dist[i][j] that is not INF
- In weighted tree
  - Do DFS/BFS twice!
    - From any vertex s, find furthest vertex x with DFS/BFS
    - Then from vertex x, find furthest vertex y with DFS/BFS
    - Answer is the path length of x-y
    - O(V+E = V+V = V) only – two calls of DFS/BFS

Tree Illustration





Euler Graph

Eulerian Graph (1)

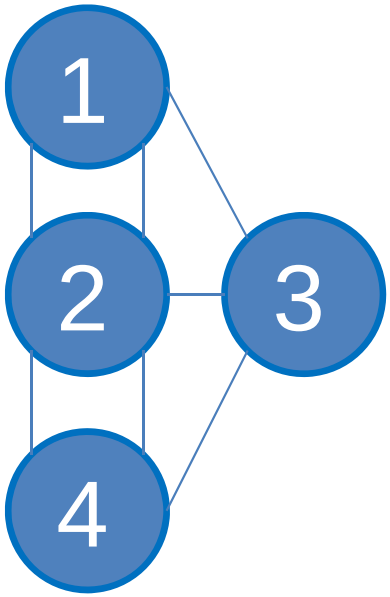
- An **Euler path** is defined as a path in a graph which visits each edge exactly once
- An **Euler tour/cycle** is an Euler path which starts and ends on the same vertex
- A graph which has either Euler path or Euler tour is called Eulerian graph

Eulerian Graph (2)

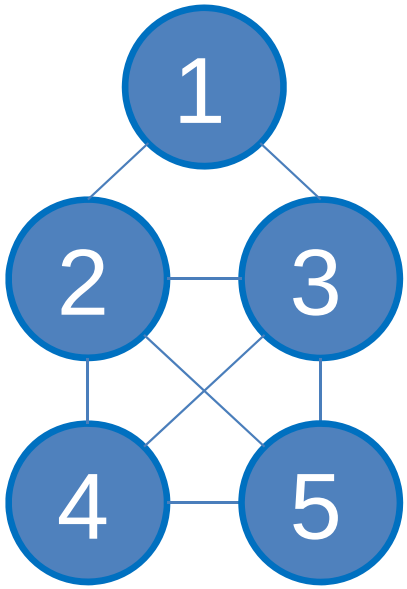
- To check whether an undirected graph has an Euler tour is **simple** ☺
  - Check if all its vertices have even degrees.
- Similarly for the Euler path
  - An undirected graph has an Euler path if all except two vertices have even degrees and at most two vertices have odd degrees. This Euler path will start from one of these odd degree vertices and end in the other
- Such degree check can be done in O(V + E), usually done simultaneously when reading the input graph

Eulerian Graph (3)

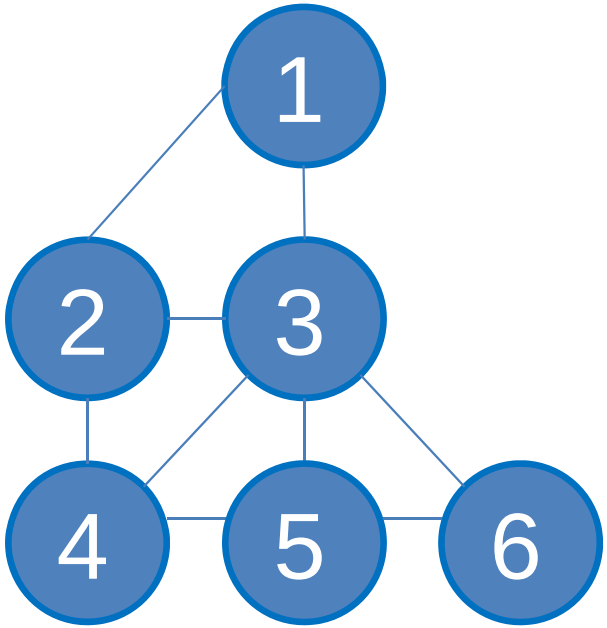
Königsberg
Non Eulerian

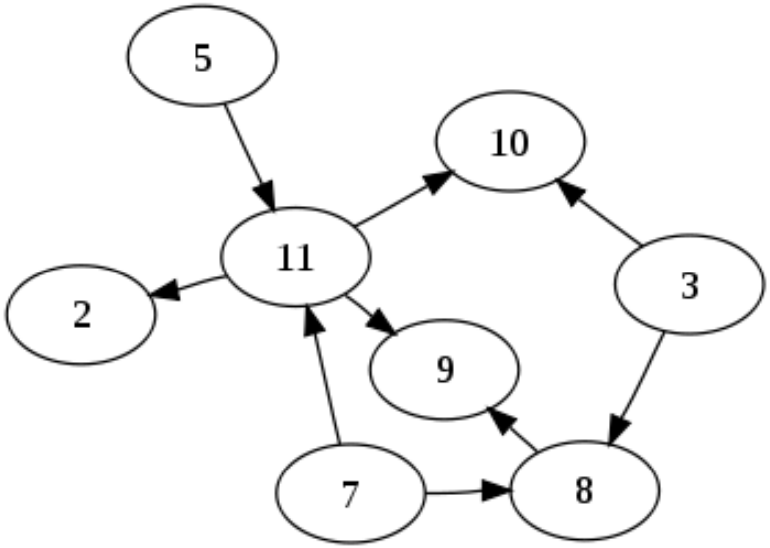


UVa 291
Eulerian



Non Eulerian





DIRECTED ACYCLIC GRAPH (DAG)

Directed Acyclic Graph (DAG)

- Some algorithms become simpler when used on DAGs instead of general graphs, based on the principle of topological ordering
- For example, it is possible to find shortest paths and longest paths from a given starting vertex in DAGs in **linear time** by processing the vertices in a **topological order**, and calculating the path length for each vertex to be the minimum or maximum length obtained via any of its incoming edges
- In contrast, for arbitrary graphs the shortest path may require slower algorithms such as Dijkstra's algorithm or the Bellman-Ford algorithm, and longest paths in arbitrary graphs are NP-hard to find

Single-Source Shortest Paths
in DAG

- In general weighted graph
  - Again this is O((V+E) log V) using Dijkstra's or O(VE) using Bellman Ford's
- In DAG
  - The fact that there is no cycle simplifies this problem substantially!
    - Simply "relax" vertices according to topological order! This ensure shortest paths are computed correctly!
    - One Topological sort can be found in O(V+E)

Single-Source Longest Paths
in DAG

- In general weighted graph
  - Longest (simple) paths is an NP complete problem
- In DAG
  - The solution is the same as shortest paths in DAG, just that we have tweak the relax operator (or alternatively, negate all edge weight in DAG)

Summary (1)

- Today, we have *quickly* gone through various well-known graph problems & algorithms
  - Depth First Search and Breadth First Search
    - Connected versus **Strongly** Connected Components
  - Kruskal's for MST (briefly)
  - Shortest Paths problems
    - BFS (unweighted), Dijkstra's (standard), Floyd Warshall's (all-pairs, three liners)
  - Special Graph: Tree, Eulerian, DAG

Summary (2)

- Note that just knowing these algorithm will not be too useful in contest setting…
- You have to practice **using them**
  - At least code each of the algorithms discussed today on a contest problem!