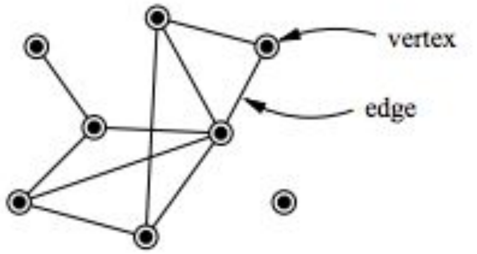
Workshop #3 Competitive Programming: Graph Traversal & Min Spanning Trees

Presentation: Imad Dodin
Slides: Muhammad Huzaifa Elahi

>Compete McGill_

Depth First Search



- Depth First Search — DFS— is a simple algorithm for traversing a graph.
    - Starting from a source vertex, DFS will traverse the graph ‘depth-first’.
    - Every time DFS hits a branching point (a vertex with more than one neighbors),
    - DFS will choose one of the unvisited neighbor(s) and visit this neighbor vertex.
    - DFS repeats this process and goes deeper until it reaches a vertex where it cannot go any deeper.
    - When this happens, DFS will ‘backtrack’ and explore another unvisited neighbor(s), if any.
    - Runtime:
        - O(V + E) if the graph is stored as Adjacency List
        - O(V^2) if graph is stored as Adjacency Matrix

>Compete McGill_

Depth First Implementation

```java
// A function used by DFS
    void DFSUtil(int v,boolean visited[])
    {
        // Mark the current node as visited and print it
        visited[v] = true;
        System.out.print(v+" ");

        // Recur for all the vertices adjacent to this vertex
        Iterator<Integer> i = adj[v].listIterator();
        while (i.hasNext())
        {
            int n = i.next();
            if (!visited[n])
                DFSUtil(n, visited);
        }
    }

    // The function to do DFS traversal. It uses recursive DFSUtil()
    void DFS(int v)
    {
        // Mark all the vertices as not visited

        boolean visited[] = new boolean[V];

        // Call the recursive helper function to print DFS traversal
        DFSUtil(v, visited);
    }
```

Depth First Search - Backtracking

```
void backtrack(state) {
    if (hit end state or invalid state) // we need terminating or
        return; // pruning condition to avoid cycling and to speed up search
    for each neighbor of this state // try all permutation
        backtrack(neighbor);
}
```

>Compete McGill_

Breadth First Search

- Breadth First Search— BFS—is another graph traversal algorithm.
  - Starting from a source vertex, BFS will traverse the graph ‘breadth-first’
  - BFS will visit vertices that are direct neighbors of the source vertex (first layer), neighbors of direct neighbors (second layer), and so on, layer by layer.

    BFS starts with the insertion of the source vertex s into a queue, then processes the queue as follows:
    - Take out the front most vertex u from the queue
    - enqueue all unvisited neighbors of u and mark them as visited.
    - With the help of the queue, BFS will visit vertex s and all vertices in the connected component that contains s layer by layer.
- Runtime:
  - $O(V + E)$ if the graph is stored as Adjacency List
  - $O(V^2)$ if graph is stored as Adjacency Matrix

>Compete McGill_

Breadth First Search Implementation

```java
// prints BFS traversal from a given source s
 void BFS(int s)
 {
     boolean visited[] = new boolean[V];

     LinkedList<Integer> queue = new LinkedList<Integer>();  // Create a queue for BFS

     visited[s]=true; // Mark the current node as visited and enqueue it
     queue.add(s);

     while (queue.size() != 0)
     {
         // Dequeue a vertex from queue and print it
         s = queue.poll();
         System.out.print(s+" ");

         // Get all adjacent vertices of the dequeued vertex s
         // If a adjacent has not been visited, then mark it visited and enqueue it
         Iterator<Integer> i = adj[s].listIterator();
         while (i.hasNext())
         {
             int n = i.next();
             if (!visited[n])
             {
                 visited[n] = true;
                 queue.add(n);
             }
         }
     }
 }
```

>Compete McGill_

Use of BFS/DFS - Connected Components

```
void DFSUtil(int v, boolean[] visited) {
        // Mark the current node as visited and print it
        visited[v] = true;
        System.out.print(v+" ");
        // Recur for all the vertice adjacent to this vertex
        for (int x : adjListArray[v]) {
            if(!visited[x]) DFSUtil(x,visited);
        }

    }
    void connectedComponents() {
        // Mark all the vertices as not visited
        boolean[] visited = new boolean[V];
        for(int v = 0; v < V; ++v) {
            if(!visited[v]) {
                // print all reachable vertices from v
                DFSUtil(v,visited);
                System.out.println();
            }
        }
    }
```

>Compete McGill_

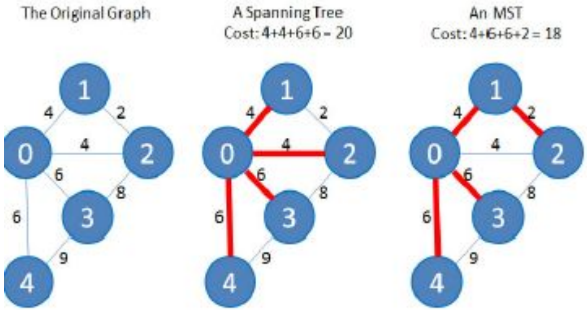
Use of BFS/DFS - Bipartite Graph Check

```
// function to check whether a graph is bipartite or not
bool isBipartite(vector<int> adj[], int v, vector<bool>& visited, vector<int>& color)
{
    for (int u : adj[v]) {

        if (visited[u] == false) { // if vertex u is not explored before

            visited[u] = true;     // mark present vertic as visited

            color[u] = !color[v];  // mark its color opposite to its parent

            // if the subtree rooted at vertex v is not bipartite
            if (!isBipartite(adj, u, visited, color))
                return false;
        }

        // if two adjacent are colored with same color then the graph is not bipartite
        else if (color[u] == color[v])
            return false;
    }
    return true;
}
```

>Compete McGill_

Minimal Spanning Trees

- Given a connected, undirected, and weighted graph G select a subset of edges E ∈ G such that the graph G is connected and the total weight of the selected edges E is minimal!
- How?
  - Prim's Algorithm
  - Kruskal's Algorithm
- To satisfy the connectivity criteria:
  - We need at least V −1 edges that form a tree T
  - This tree must spans (covers) all V ∈ G—the spanning tree!
  - There can be several valid spanning trees in G

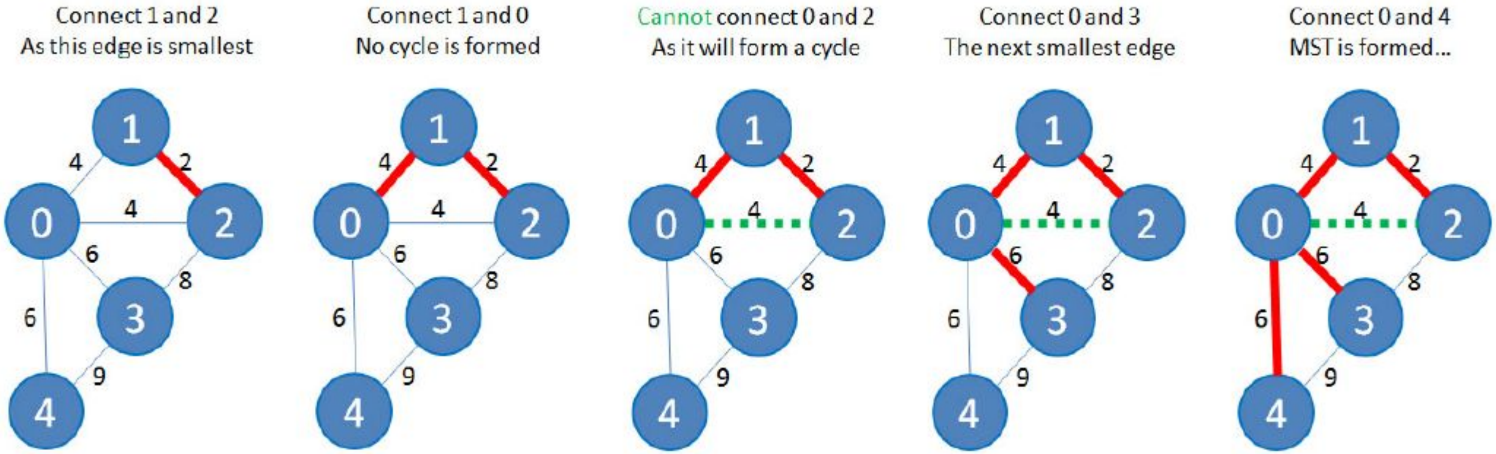


- Practical applications:
  - We can model a problem of building road network in remote villages as an MST problem.
  - The vertices are the villages.
  - The edges are the potential roads that may be built between those villages.
  - The cost of building a road that connects village i and j is the weight of edge (i, j).
  - The MST of this graph is minimum cost road network connecting these villages

>Compete McGill_

Minimal Spanning Trees - Kruskal's

- Kruskal algorithm:
  - First sorts E edges based on ascending order of weights
  - Then, greedily tries to add each edge into the MST (without cycle) picking lowest weight
  - Repeat this step until all vertices connected (MST established)
  - Runtime of this algorithm is O(E log V).



>Compete McGill_

Kruskal’s Implementation

```
public static List<Edge> kruskalAlgorithm(List<Edge> edges, int nodeCount) {

	DisjointSet ds = new DisjointSet(nodeCount);
	List<Edge> spanningTree = new ArrayList<Edge>();

	// Sort edges by weight
	Collections.sort(edges);
	int i = 0;

	// While MST is not created & all edges haven’t been explored
	while (i != edges.size() && spanningTree.size() != nodeCount - 1) {
		Edge e = edges.get(i);

		// Cycle check
		if(ds.find(e.getFrom()) != ds.find(e.getTo())){
			spanningTree.add(e);
			ds.union(e.getFrom(), e.getTo());
		}
		i++;
	}
	return spanningTree;
}
```

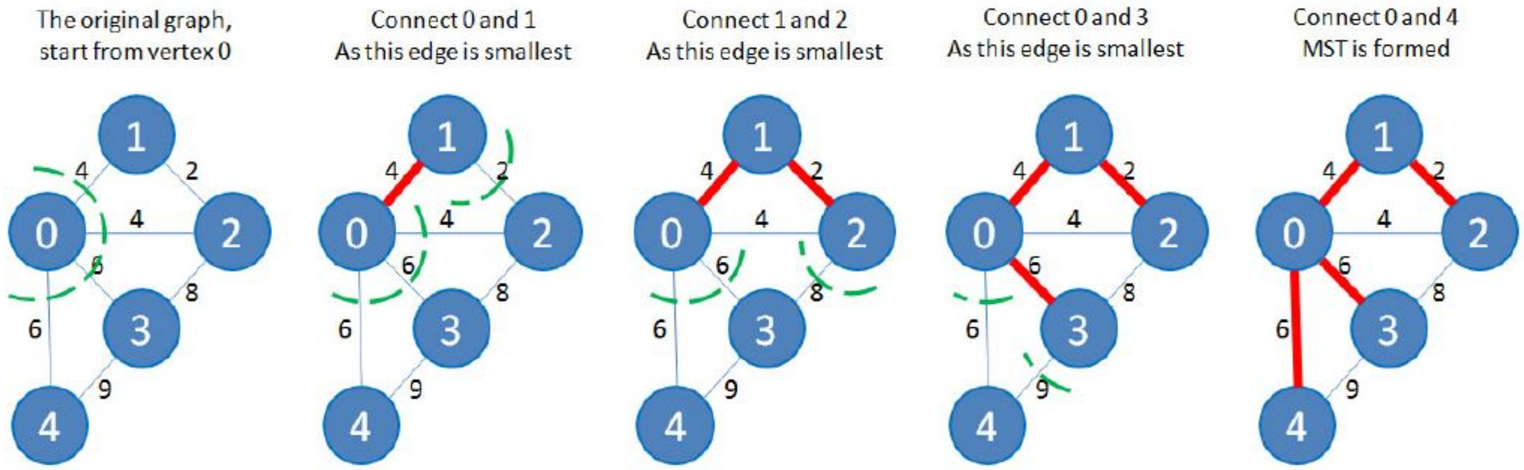
>Compete McGill_

Minimum Spanning Trees - Prim's

- Prim's algorithm:
  - First takes a starting vertex flags it as 'taken'
  - Then enqueues a pair of information into a priority queue:
    - The weight w and the other end point u of the edge 0 → u that is not taken yet.
  - These pairs are sorted in the priority queue based on increasing weight
  - Then, greedily selects the pair (w, u) in front of the priority queue
    - IF AND ONLY IF the end point of this edge - u - has not been taken before (prevent cycle)
  - Then the weight w is added into the MST cost
  - u is marked as taken
  - Pair (w, v) of each edge u → v with weight w that is incident to u is enqueued into the priority queue if v has not been taken before.
  - This process is repeated until the priority queue is empty.
  - Runtime: O(E log V).

>Compete McGill_

Minimum Spanning Trees - Prim's In Action



Prim's Implementation

```
void primMST(int graph[V][V]) {
    int parent[V];        // Array to store constructed MST
    int key[V];           // Key values used to pick minimum weight edge in cut
    bool mstSet[V];       // To represent set of vertices not yet included in MST

    for (int i = 0; i < V; i++)    // Initialize all keys as INFINITE
        key[i] = INT_MAX, mstSet[i] = false;

    key[0] = 0;       // Include first 1st vertex in MST with key 0 so this vertex is picked as first
    parent[0] = -1;  // First node is always root of MST

    for (int count = 0; count < V-1; count++) {     // The MST will have V vertices
        int u = minKey(key, mstSet);                // Pick min key vertex from vertices not yet in MST
        mstSet[u] = true;                          // Add the picked vertex to the MST Set
        // Update key value and parent index of the adjacent vertices of the picked vertex.
        // Consider only those vertices which are not yet included in MST
        for (int v = 0; v < V; v++) {

        // graph[u][v] is non zero only for adjacent vertices of m
        // mstSet[v] is false for vertices not yet included in MST
        // Update the key only if graph[u][v] is smaller than key[v]
        if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v])
            parent[v] = u, key[v] = graph[u][v];
    }
```

>Compete McGill_