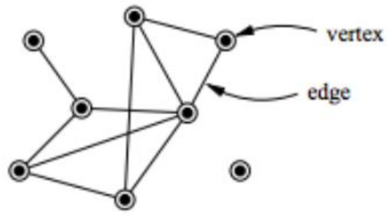
Graph Intro, BFS + DFS, Dijkstra's

1332 Recitation: Week of July 7th

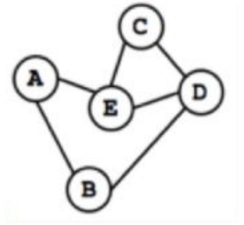
Graphs Intro

- Most of the definitions are on the worksheet: “**worksheet12**” in the Recitation - Worksheets folder.
  - Feel free to post on Piazza, come to office hours, or email TAs if you have any questions about the definitions
- **Vertex:** a position/ node on the graph (set of vertices is denoted by V)
- **Edge:** a connection between two vertices (set of edges is denoted by E)
  - Edge is denoted by (u, v) where u is the origin of the edge and v is its destination
  - Undirected edge: u leads to u and v and v leads to u (i.e. two-way street)
    - Undirected edges are represented and implemented as two directed edges
  - Directed edge: u leads to v, but v may or may not lead to u (i.e. one-way street)

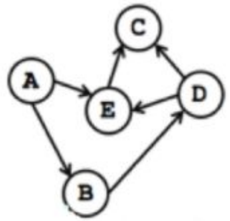


Graphs Intro

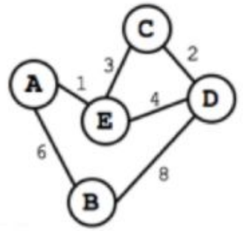
- **Graph:** a set of vertices and edges (denoted by (V,E))
  - **Undirected** graph: graph in which all edges are undirected
  - **Directed** graph: graph with directed edges
  - **Weighted** graph: graph in which each edge has an associated value/weight (e.g. time, distance)
  - **Connected** graph: graph in which every vertex has a path to every other vertex
  - **Unconnected** graph: graph in which any given vertex can't reach all other vertices
  - Connected/unconnected will only be for undirected graphs in this course



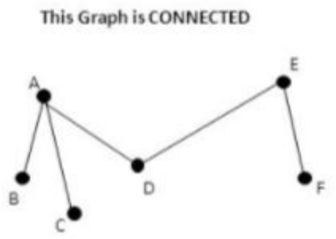
(A) Undirected Graph

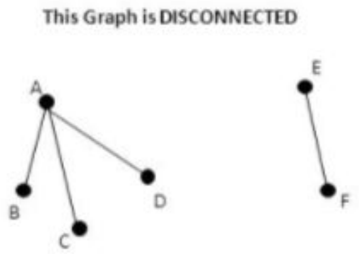


(B) Directed Graph



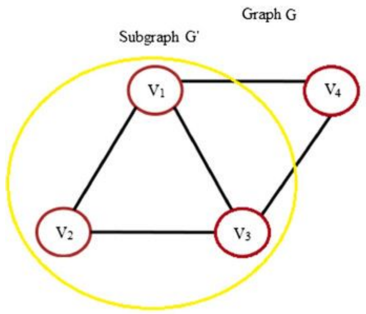
(C) Weighted Graph

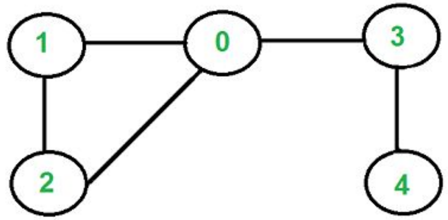


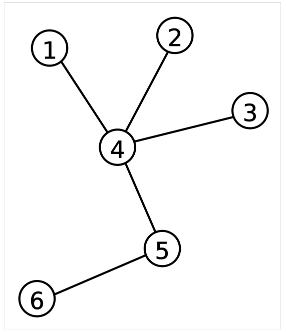


Graph Intro

- **Degree:** number of edges connected to a vertex
- **Indegree:** number of edges going into a vertex
- **Outdegree:** number of edges going out of a vertex
    - Same as indegree for undirected graphs
- **Subgraph:** a graph G' = (V', E') is a subgraph of G = (V,E) if V' is a subset of V and E' is a subset of E
- **Cycle:** a path through other vertices that returns to the initial vertex







Breadth-First Search

- Traverses a graph starting from a given start vertex using a Queue and a visited Set
  - **We would scan vertices one edge away from start, then two, etc**
    - This allows us to find shortest paths on unweighted graphs (not in weighted graphs though!)
- Given a graph and starting vertex, should return a list of vertices in visited order
- Really a *search algorithm* -- usually instead of adding to an output list, you would check if this vertex is one that you wanted to find

Breadth-First Search - Algorithm

- Add start into the Queue
- While the Queue isn’t empty, and not all vertices have been visited:
    - Remove a vertex, check if it has been visited, and if not:
        - Visit node (in our case, add to list) and mark it as visited (add to visited Set)
        - Add each neighbor that has not been visited into the Queue
- Example!
- Efficiency: O(|V| + |E|); accesses each vertex and edge once
- We use a Set to mark vertices as visited - O(1) to access
- Levelorder() in BSTs and AVLs was a BFS

Depth-First Search

- Another way of traversing a graph starting from a start vertex, but using a Stack instead (and still using a visited Set)
    - Can use a Stack data structure, but in this class, we'll use the **recursive stack** instead
- Given a graph and starting vertex, should return a list of vertices in visited order (same goal as BFS, but different search ordering)

Depth-First Search - Algorithm

- If curr is not visited:
    - Visit node (in our case, add to list) and mark as visited (add to visited Set)
    - For each neighbor:
        - If not visited (note that this check is not necessary because we already check this in the beginning of the next recursive call, but we will use it for diagramming to simplify things):
            - Recurse with neighbor as curr
- Example!
- Efficiency: O(|V| + |E|) - accesses each vertex and edge once (same reasoning from BFS)
- We also use a Set to keep track of visited nodes like in BFS
- The order in which we visit nodes (for DFS and BFS) in implementation will be determined by whatever order is in the adjacency list

Dijkstra's

- Given a start vertex in a weighted graph with non-negative (edges with 0 weight are fine) edge weights, finds the shortest path from start to every other vertex in the graph (**single-source shortest path** algorithm)
- Also works with unweighted graphs (since we can just think of all edges having weight 1) -- this is actually exactly BFS
- Uses a PriorityQueue of VertexDistance
    - VertexDistance is just an auxiliary data structure that holds a vertex and a distance (we will call this VD); for Dijkstra's, the d in **(v, d)** is the **cumulative path cost from source to vertex v**
- Given a graph and a start vertex, returns a Map of Vertices to shortest distance from start vertex to said vertex

Dijkstra's - Algorithm

- Set the distance for each vertex (except start) to infinity (in the HW, Integer.MAX_VALUE)
- Add start to the PQ as a VD with a distance of 0
- While the PQ isn't empty and not all vertices have been visited:
    - Dequeue a vertex, check if it has been visited, and if not:
        - Mark it as visited (add to visited Set)
        - For each neighbor that has not been visited:
            - If the path to this neighbor through the vertex is shorter than the existing path:
                - Update distance in the map with new shortest distance and add a new VD with the new total distance
- Example!

Dijkstra’s (cont.)

- Efficiency: O((|V| + |E|) • log|V|)
    - |V| • log |V| to visit each vertex and perform |V| dequeue operations from the PQ
    - |E| • log |V| to consider every path through each |E| edges making |E| enqueue() to PQ
- Intuitively, Dijkstra’s is a generalization of BFS for weighted graphs
    - The edge weights can be thought of “how many edges” there are between vertices and the PQ acts as a Queue that takes into account that information
    - Based on this, we can think of why Dijkstra’s is not guaranteed to work if there are negative edge weights in the graph
        - Example with negative weights!
- Is a greedy algorithm; assumes that the shortest distance is known once a vertex is dequeued
    - This is why Dijkstra’s doesn’t work with negative edge weights

MST (Briefly)

- **Tree:** A graph/subgraph with **no** cycles (acyclic graph)
- **Spanning tree:** A subgraph of an undirected graph that includes all vertices, is acyclic and connected
  - If the graph is disconnected to begin with, we cannot make it connected
- **Minimum spanning tree (MST):** A spanning tree that has minimum weight (i.e. it minimizes the total weight of the set of edges)
  - Intuitively, it is a “stripped down” graph with “unnecessary” edges removed
  - For the same reasoning as before, disconnected graphs do not have an MST
  - A graph may contain more than one MST if there are multiple set of edges with the same minimum weight
  - Will always have |V| - 1 undirected edges or 2(|V| - 1) directed edges
    - Undirected edges are represented and implemented as two directed edges

Prim’s

- Greedy algorithm that finds an MST in a given graph
- Only works for undirected graphs(because of the definition of MSTs)
- We will be storing the MST as a Set of edges, which we will return
- Like in BFS, DFS and Dijkstra’s, we will use a Set to keep track of visited vertices(so we will use two Sets)

Prim's - Algorithm

- Initialize visited set, priority queue (of **edges**), and MST set (also edges)
- Add start to visited
- while (mst not complete and pq not empty)
    - Edge (u, v) <- dequeue
    - If (v not visited):
        - Add (u, v) and its reverse (v, u) to MST
        - Add v to visited
        - For each neighboring edge (v, w):
            - If (w not visited):
                - Enqueue (v, w)

-> Check validity of MST before returning(check if there are |V| vertices in visited Set or 2(|V| - 1) edges in MST)

- Intuition is that we grow the MST as a cloud of vertices from the starting vertex

Prim's

- Example!
- Efficiency: O((|E| + |V|) log|V|)
    - Intuition is that we explore the graph similar to BFS/DFS, but when visiting vertices there is a O(log|V|) cost associated with PQ operations (this is the same intuition for Dijkstra's)
    - It is best to not overthink this though, as there are multiple ways of defining big-O's of graph algorithms due to the fact that we can express things in terms of |E| or |V| , and we prioritize |V|
- Since we keep track of a visited set, we can either check if it has |V| vertices or if the MST has 2(|V| - 1) edges
- When adding edges, since the graph is undirected, it is OK for them to create another edge going the opposite direction to add it to the MST

Graph HW - Helper Classes

- Edge<T>
  - Class representing a directed edge from a vertex u to a neighbor vertex v in the graph
  - Useful methods: getU(), getV(), getWeight()
- Vertex<T>
  - Class representing a vertex in the graph
  - Useful methods: getData()
- VertexDistance<T>
  - Class representing the distance to a vertex from a current vertex
  - Probably the most useful class for this homework(further explanation under Graph<T>)
  - Useful methods: getVertex(), getDistance()

Graph HW - Helper Classes

- Graph<T>
  - Class representing a directed graph, with a vertex set, edge set and adjacency list
  - Adjacency list:
    - One way to represent a graph
    - Maps each vertex in the graph to a list of neighboring vertices/ vertices it shares an edge with
    - Implemented as a HashMap with key = Vertex<T>, value = List<VertexDistance<T>>
    - Given a vertex, you can use this structure to obtain a list of neighboring vertices along with
    - distances from the given vertex to them
    - VertexDistance<T> is what stores the neighboring vertex and distance to it in the list
  - Useful methods:
    - getAdjList() - returns the adjacency list which you can store for looking up neighboring vertices
    - getVertices() - returns a set of all the vertices in the graph