Graphs: Undirected Graphs

- Path: Sequence of vertices connected by edges
- Cycle: Path whose first and last vertices are the same
- Two vertices are connected if there is a path between them
- If an edge exist between two vertices, they are adjacent to each other
- The degree of a vertex in an undirected graph is the number of edges that are directly connected to it
- Self Loop: An edge loop that connets a vertex to itself

- Edges that directly connect the same pair of vertices are parallel

- Graphs can be represented by maintaining a list of edges via Array or Linked List
  - ↳ Basic operations are not performed efficiently
- Graphs can be represented through an adjacency-matrix
  - ↳ Needs an enormous amount of space to represent large graphs
  - ↳ Good for small graphs and dense graphs
- Graphs are mostly Sparse, so adjacency matrix is not a good implementation
- Graphs can be represented by an adjacency-list
  - ↳ Maintains a vertex-indexed array of lists
  - ↳ Cannot represent parallel edges
- In practice, adjacency-list graphs are used because real world graphs are sparse

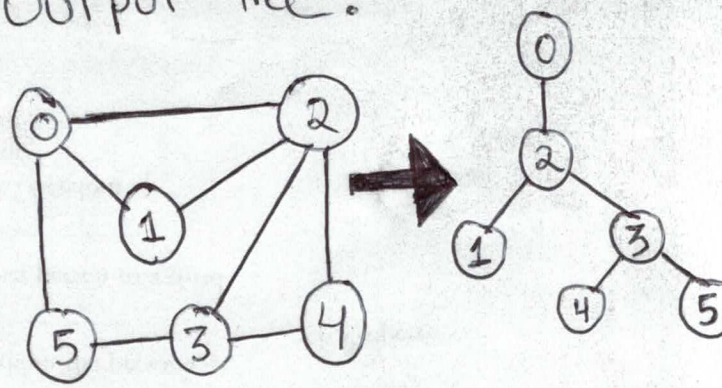
- Sparse graphs contain a huge number of vertices, and small average vertex degree
- Bag data structure is used for the adjacency-list Graph
- Three algorithms to search through a graph:
  - Union find
  - Depth First Search (DFS)
  - Breadth First Search (BFS)
- DFS is recursive algorithm
- Recursive algorithms can be converted to Iterative algorithms via the stack data structure
- Undirected graphs have two edges: an inward edge and an outward edge
- Vertex connections can be direct or indirect

- DFS Pseudo Code:

DFS (to visit a vertex v)

- Mark v as visited.
- Recursively visit all unmarked vertices w adjacent to v.

- When DFS is called on a vertex, it gets marked
- Adjacency list is based off of adding edges/vertices to the graph based on input
  - Adjacency list is based on the initial input
- Output tree:



- Generating the output tree requires the adjacency list
- Every marked vertex is connected to the source vertex
- Marking ensures that each vertex is visited once
- The maximum number of edges traversed is 2·E

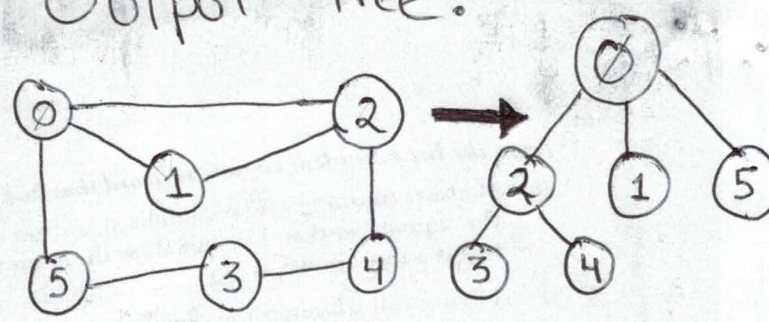
- If the source is connected to all edges, then the max. time is $V + 2E$
- DFS does not give the shortest path
  - ↳ BFS does find shortest path
- Breadth-first search pseudo-code:

BFS (from a source vertex s)

- Put "s" onto a FIFO queue, and mark "s" as visited
- Repeat until the queue is empty:
  - Remove the least recently added vertex
  - Add each of "v's" unvisited neighbours to the queue, and mark them as visited

- BFS uses a queue instead of a stack
  - ↳ Not recursive like DFS

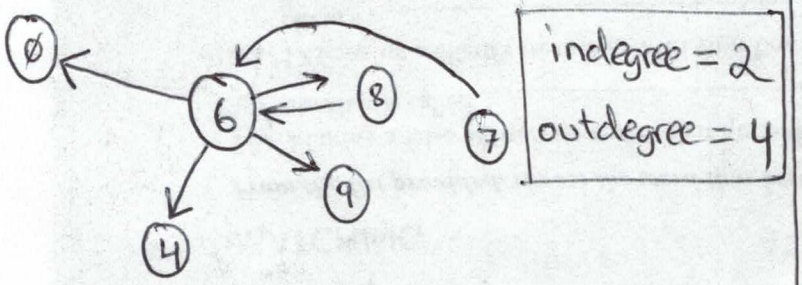
Output Tree:



- The "edgeTo[]" array is needed to generate the output tree
- BFS computes shortest paths from "s" to all other vertices in time proportional to $E + V$
- DFS uses a stack
- BFS uses a queue
- BFS takes the least recently added vertex
- DFS takes the most recently added vertex
- All components in a graph are reflexive
  - ↳ "v" is connected to "v"
  - ↳ self-loop
- DFS is used to find connected components in a graph
  - ↳ Takes $(V + E)$ time and space
- In practice, union find is faster than DFS

Graphs: Directed Graphs

- Directed graphs are also called digraphs
- In a digraph, edges are one way; contain a direction
- The indegree of a vertex is the number of directed edges that point to the vertex
- The outdegree of a vertex is the number of directed edges that point out from it



- Directed graphs are represented using adjacency lists
  - ↳ identical to undirected graphs
- DFS algorithm works for both undirected and directed graph
  - ↳ same with BFS
- Multiple shortest path is achieved via BFS by enqueuing all source vertices

- A directed acyclic graph (DAG) is a digraph with no directed cycles
  - ↳ used to solve precedence constrained scheduling
- Topological sort, sorts the vertices of a digraph such that precedence constraints are maintained
  - ↳ all directed edges point to a vertex later in the order, and there is no back-pointing
  - ↳ is a linear ordering
- DFS is used to compute topological sort
- There are 3 vertex orderings:
  - ↳ Preorder: Put the vertex on a queue before the recursive calls
  - ↳ Postorder: Put the vertex on a queue after the recursive calls; when it's done
  - ↳ Put the vertex on a stack after the recursive call

Reverse Postorder

- The reverse postorder in a DAG is a topological sort
- Topological sort does not work and is not possible if the graph has a cycle
- In topological order:
  - The first vertex has an indegree of 0
  - The last vertex has an outdegree of 0
- A digraph has topological order IFF it is acyclic
- With DFS, topological order can be calculated in time proportional to $(V + E)$
- There can be multiple topological sorts in a DAG
- Second-to-last vertex in topological order can only point to the last vertex
  - Alternatively, second vertex in postorder can only point to first vertex
- DFS is used to detect directed cycles in a digraph

- Vertices "v" and "w" are <u>strongly connected</u> if there is a directed path from "v" to "w" and a directed path from "w" to "v"
  - Note: This is a path, <u>not</u> an edge.
- Each component is <u>strongly connected</u> to itself
  - "v" is strongly connected to "v"
- The Kosaraju-Sharir algorithm is used to find the strongly connected components of a digraph
  - Step 1: Run DFS on G^R to compute reverse postorder
  - Step 2: Run DFS on G, considering vertices in order given by first DFS
  - Time Complexity: $(E+V)$
- Kosaraju-Sharir uses DFS to find the Strongly connected components

Graphs: Minimum Spanning Trees

- MSTs are used for edge weighted graphs
- The purpose of an MST is to find a minimum weight set of edges that connects all of the vertices
- A spanning tree of a Graph, G, is a subgraph, T, that is:
  - Connected
  - Acyclic
  - Includes all of the vertices in G
- An edge-weighted graph can have many spanning trees, but only one MST
  - However, in some cases you can have more than one MST; if some edge weights are identical
- The cut property is used to prove the correctness of an MST
- A cut in a graph is a partition of its vertices into two (non-empty) disjoint sets
- A crossing edge connects a vertex in one set with a vertex in the other set
- Cut Property: Given any cut, the crossing edge of minimum weight is in the MST
  - Out of all the crossing edges, the one with the minimum weight will be included in the MST
- Kruskal's Algorithm:
  - Step 1: Consider edges in ascending order of weight
  - Step 2: Add next edge to MST unless doing so would create a cycle
- In Kruskal's algorithm, the edges are sorted by weight from smallest to largest. Then, you keep on adding edges unless it creates a cycle or all edges are exhausted
- DFS can be used to identify cycles; in $O(V)$ time per cycle check
- Union find can be used to identify cycles; in $\log(V)$ time
  - Works better than DFS

- Kruskal's algorithm compute MST in $\Theta(|E| \log |E|)$ time
  - ↳ runtime is dominated by the sorting of edges
- Prim's Algorithm:
  - ↳ Step 1: Start with vertex 0 and greedily grow tree, T
  - ↳ Step 2: Add to T the min weight edge with exactly one endpoint in T
  - ↳ Step 3: Repeat until V-1 edges in MST
- Two versions to implement Prim's Algorithm
  - ↳ Lazy
  - ↳ Eager
  - Both maintain a priority queue
- In Prim's Algorithm, you maintain a list of all visited vertices, and you use that list each time you want to find the edge with a minimum weight
- Lazy Prim's Algorithm computes the MST in time proportional to (E log E)
- Eager version of Prim's Algorithm uses extra space proportional to V and time proportional to (E log V) in the worst case to compute the MST
- The running time of Kruskal's Algorithm is greater than the Eager implementation of Prim's Algorithm
  - ↳ However, there are instances where one might have a better performance than the other
    - ↳ i.e. Dense graph, sparse graph, etc.
- A Minimum Spanning Forest (MSF) is the set of all MSTs
  - ↳ the collection of MSTs is an MSF
- Lazy Prim's Algorithm does not check the edge weights, it adds them to the Priority Queue, regardless
  - ↳ Eager Prim's Algorithm does checks the weights and only adds the best ones!

Graphs: Shortest Paths

- The goal is to find the shortest paths in an edge-weighted digraph
- BFS finds the shortest paths in undirected graphs and digraphs that do not have edge-weights
- The shortest-paths tree (SPT) for an edge-weighted digraph is a subgraph containing the designated vertex, "s", and all the vertices reachable from "s" that forms a directed tree rooted at "s" such that every tree path is a shortest path in the subgraph
- Every tree has an SPT solution, no matter the case
  - ↳ even trees with a single node
- Minimum Spanning Forest (MSF) is used for undirected edges
- The SPT does not need to have a path to all nodes
  - ↳ only the nodes that are reachable

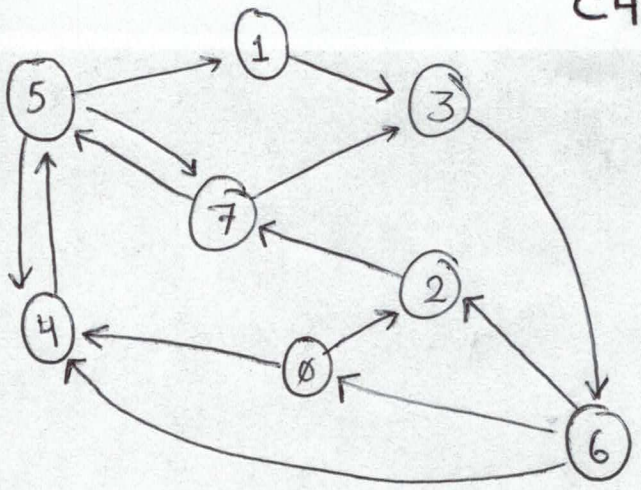


Figure: Shortest-Paths tree from 0

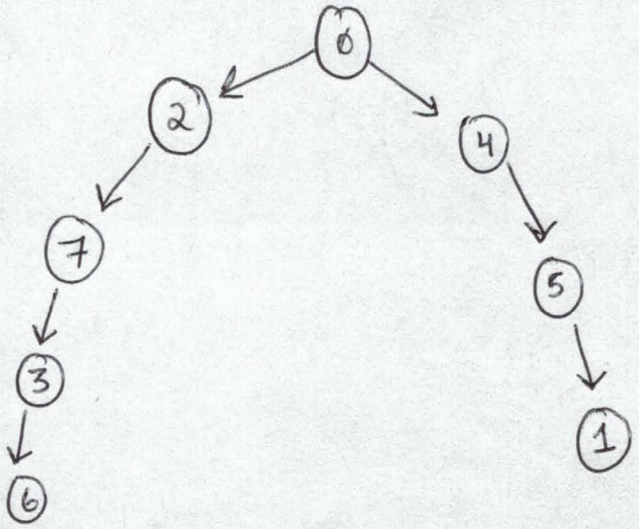


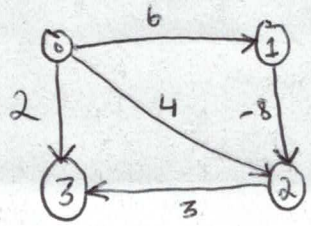
Figure: SPT drawn as tree rooted at 0 for figure above

- Relaxing an edge means to update the "distTo[]" and "edgeTo[]" array
  - ↳ If an edge connecting vertex V and W gives a shorter path to W through V, then update both "distTo[w]" and "edgeTo[w]".
- In vertex relaxation, all the edges pointing from a given vertex are relaxed

- Generic Algorithm (to compute SPT from "s")
  - Initialize "distTo[s] = 0 and distTo[v] = ∞ for all other vertices
  - Repeat until optimality conditions are satisfied
    - Relax any edge

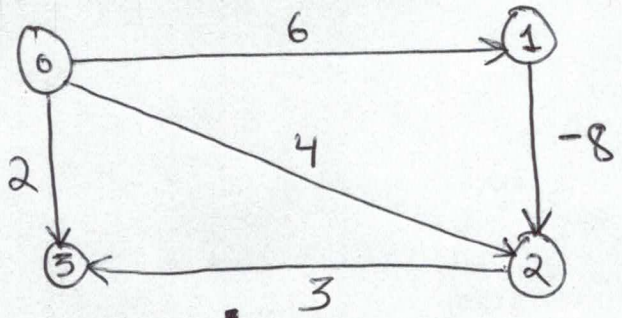
~~If distTo[v]'s correct reachable from s~~

- If "v" is not reachable from "s", then "distTo[v]" remains at infinity
- The generic shortest paths algorithm does not specify in which order the edges are to be relaxed
  - 3 Efficient Implementations
    - Dijkstra's Algorithm
    - Topological Sort Algorithm
    - Bellman-Ford Algorithm
- Dijkstra's is a greedy algorithm
- Dijkstra's Algorithm cannot handle ~~negative~~ negative weights or negative cycles
- Topological sort cannot handle directed cycles
  - only works on DAGs
  - can handle negative weights
- Bellman-Ford algorithm cannot handle negative cycles
  - can handle negative weights
  - can handle directed cycles
- An SPT does not exist if there IS a negative cycle
- In Dijkstra's Algorithm, the distance to all vertices is initialized to infinity
  - However, the distance to the source vertex is initialized to 0
- Dijkstra's Algorithm uses extra space proportional to "V" and time proportional to (E log V)
  - Only true when using a min. priority queue implemented as a binary heap
- Topological sort computes the SPT in edge-weighted DAGs in time proportional to (E+V)
  - uses DFS
- Dijkstra's Algorithm does not work for negative weights

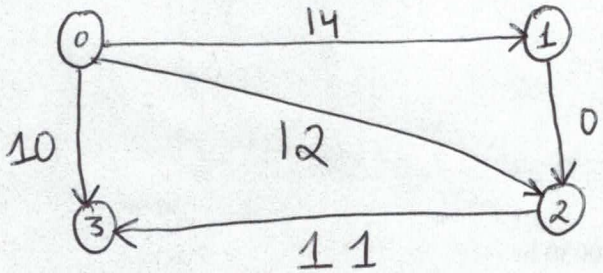


- Dijkstra selects vertex 3, immediately after 0. But, the shortest path from 0→3 is 0→1→2→3

- Adding a constant to every edge-weight, to remove the negatives, does not work



(8 added to every edge)



- Adding a constant to every edge-weight changes the properties of the graph
  - ↳ Shortest path from 0→3 is 0→3
- A negative cycle is a directed cycle where the sum of the edge weights is negative
  - $5 \to 4 \equiv -0.66$
  - $4 \to 7 \equiv 0.37$
  - $7 \to 5 \equiv 0.28$
  - ↳ negative cycle: $(-0.66 + 0.37 + 0.28)$
- An SPT (Shortest path) exists IFF there are no negative cycles

- Bellman-Ford Algorithm
  - ↳ Initialize distTo[s] = 0 and distTo[v] = ∞ for all other vertices
  - ↳ Repeat V times:
    - Relax each edge
  - ↳ Running time: O(EV)
  - ↳ ~~[illegible] queue is used to keep track of vertices~~

- Queue based Bellman-Ford computes the shortest path in any edge-weighted digraph with no negative cycles in time proportional to (E × V) in the worst case
  - ↳ However, it is much faster in practice; takes (E + V) time

- Cost Summary for single source shortest-path implementation

Algorithm	Restriction	Typical case	Worst case	Extra space
topological sort	no directed cycles	E + V	E + V	V
Dijkstra (binary heap)	no negative weights	E log V	E log V	V
Bellman-Ford	no negative cycles	EV	EV	V
Bellman-Ford (queue-based)	no negative cycles	E + V	EV	V