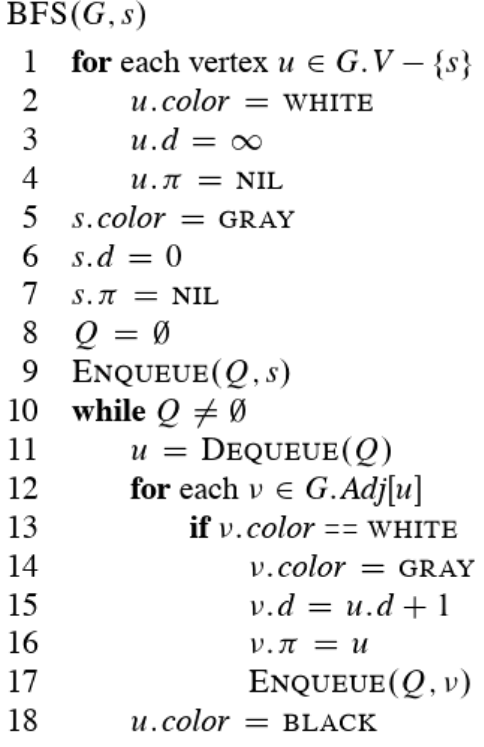
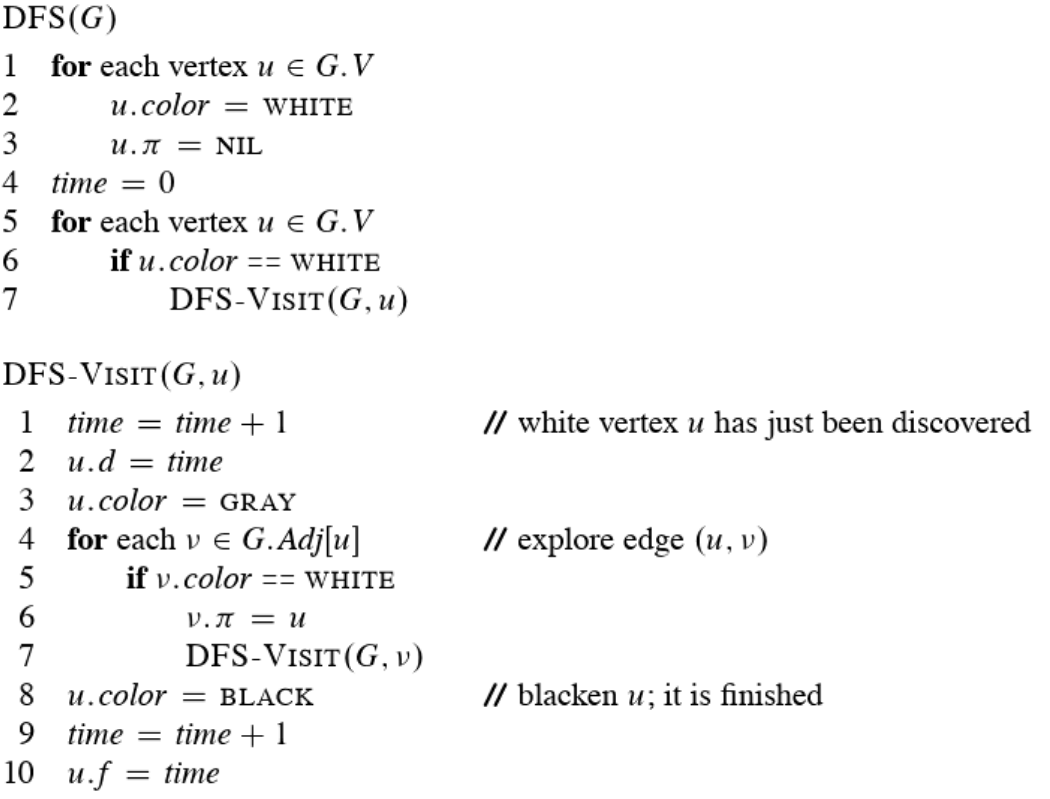
**CS 325 – Module 5 – Graph Algorithms**

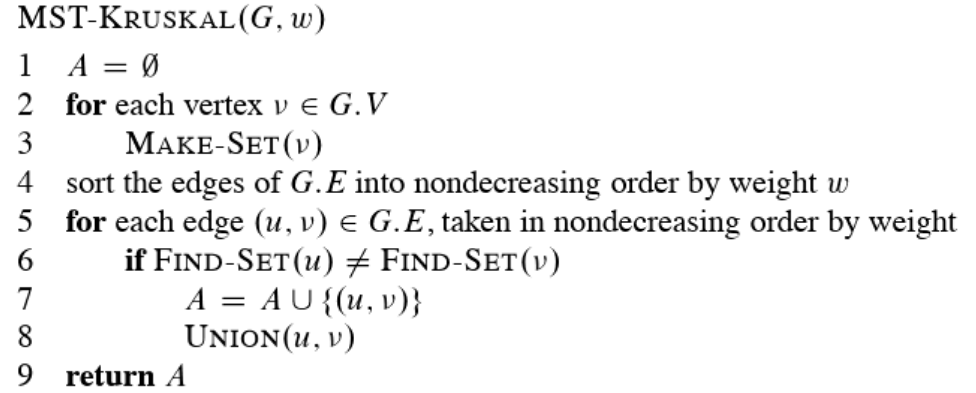
* **Graphs (G)** are mathematical objects consisting of a set of:
  + vertices/nodes (**V**) (can be denoted as **V(G)** {vertices of G} or **VG**)
  + edges (**E**) between pairs of nodes
  + **Graph size** parameters: **n = |V| , m = |E|** (can be denoted as **E(G)** {edges of G} or **EG**)
  + if edge **e = (u, v)**, we say that **u** and **v** are ***adjacent***  or ***neighbors***
  + ***u*** and ***v*** are ***incident*** with e
  + ***u*** and ***v*** *are* ***end-vertices*** of e
  + an edge where the two end vertices are the same is called a ***loop*** or a ***self-loop***
* **Directed Edges** are ordered pairs of vertices (u, v)
* **Undirected Edges** are unordered pairs of vertices, also (u, v)
  + graphs with **directed edges** are called ***directed graphs*** or ***digraph***
  + graphs with **undirected edges** are called ***undirected graphs*** or simply ***graphs***
* A **path** of an undirected graph is a traversal sequence of verts with the property that each consecutive pair of verts is joined by an edge. i.e. nodes can repeat but edges do not.
  + A **path** is considered ***simple*** if all nodes are distinct
* A ***walk*** is a **path** in which edges and nodes can be repeated
* a ***cycle*** is a path in which only the first and final vertices are the same (i.e. like a triangle)
* A ***tree*** (T) is an **undirected graph** such that…
  + T is connected (no outlying nodes)
  + T has no cycles
* A ***forest*** is an **undirected graph** without cycles
  + The connected components of a **forest** are **trees**
* A ***directed acyclic graph***(**DAG**)is a digraph that has no directed cycles
* When characterizing the running time of a graph algorithm on a given graph **G = (V, E)**, we usually measure the size of the input in terms of the number of vertices **|V|** and the number of edges **|E|**. This means we describe the size of the input with **two** parameters, not just one. We may say “the graph runs in time **O(VE)**” meaning the algorithm runs in time **O(|V||E|)**. This convention makes running times easier to read.
  + We denote the vertex set of a graph as **G.V** and its edges set by **G.E**
* We can choose between two ways to represent a graph: as a collection of **adjacency lists** or as an **adjacency matrix**. But because the adjacency-list representation provides a compact way to represent ***sparse***graphs (those for which **|E|** is much less than **|V|2**) it is usually the method of choice.
  + adjacency matrix may be preferred if the graph is ***dense*** (**|E|** is close to **|V|2**)
* The **adjaceny-list representation** of a graph **G = (V, E)** consists of an **array** (***Adj***) of **|V|** lists, one for each vertex in **V**.
* **Breadth-First Search (BFS)** is one of the simplest algorithms for searching a graph and is the backbone of many popular graphs and algorithms today (Prim’s Algorihtm, Djikstra’s)
  + Given a graph **G = (V, E)** and a distinguished ***source*** (starting point) vertex ***s***, BFS systematically explores all the edges of G to “discover” every vertex that is reachable from s
  + BFS is good to check ***if there is a path*** between two nodes and “goes wide” in its search, checks to see how many levels a node is away from another node. It is **iterative** and uses a **queue** data structure (FIFO) to keep track of current/next processed nodes.
  + Start with your ***source*** node (**s**)…we will give ours the value of “A”
    - Add A to the queue
    - Pull A from queue to process it
    - mark A as “seen” and output A
    - Add A’s children nodes to the queue (back to front would be E, D, C, B)
    - Remove B from the queue to process it
    - mark B as “seen” and output B
    - Add all of B’s children nodes **that haven’t been “seen” yet** to the back of the queue (G and C)
    - new queue is (G, C, E, D, C) (its okay if C is in here twice)
    - Remove C from queue to process it
    - Mark C as “seen” and output C
    - Add all of C’s children nodes **that haven’t been “seen” yet** to the back of the queue
    - And so on…
  + Below is a **BFS** algorithm that assumes the input graph **G= (V, E)** is represented using adjacency lists…



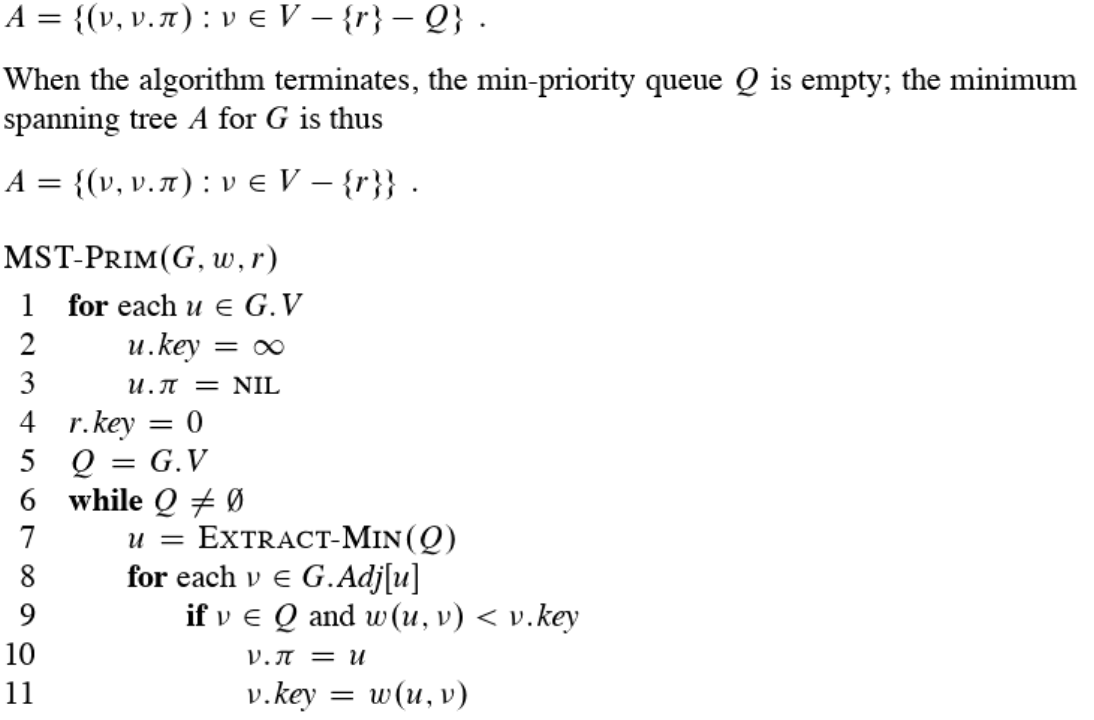
* **Depth-First Search (DFS)** searches by doing “deeper” into the graph whenever possible. **DFS** explores edges out of the most recently discovered vertex ***v*** that still has unexplored edges leaving it. Once all of ***v*** ’s edges have been explored, the search “backtracks” to explore edges leaving the vertex from which ***v*** was discovered.
  + DFS is good for backtracking, complete search, exhausting all possible paths. It is all about going “deep”. It goes deep into a path, explores all of it, then comes back outwards and decides whether or not to go to another path
  + It is **recursive** and uses a **stack** (LIFO) data structure to keep track of current/next processed nodes. This stack can be one we create ourselves or one that is the “call stack” (given to us)
  + Start with your ***source*** node (s)…we will give ours the value of “A”
    - push A to the stack
    - pop A from the stack to process it
    - If A hasn’t been “seen” yet, mark A as “seen” and output A
    - Add all of A’s children nodes **that haven’t been “seen” yet** to the stack (from top to bottom is B, C, D, E)
    - pop B from the stack to process it
    - Mark B as “seen” and output B
    - Add all of B’s children nodes **that haven’t been “seen” yet** to the stack (new stack from top to bottom is C, G, C, D, E)
    - pop C from the stack to process it
    - Mark C as “seen” and output C…and so on…
  + Below is a **DFS** algorithm



* + sometimes, when DFS returns, every vertex **u** has been assigned a ***discovery time (u.d)*** and a ***finishing time (u.f)***
  + **DFS** can also be used to classify edges of the input graph **G = (V, E)**
  + There are four “types” of edges
    - **Tree edges –** edges in the depth-first forest. Edge (*u, v*) is a **tree edge** if *v* was first discovered by exploring edge (*u, v*)
    - **Back edges –** edges (*u, v*) connecting a vertex *u* to another ancestor *v* in a depth-first tree. We consider self-loops, which may occur in directed graphs, to be back edges
    - **Forward edges –** are nontree edges (*u, v*)connecting a vertex *u* to a descendant *v* in a depth-first tree
    - **Cross edges –** are all other edges. They can go between vertices in the same depth-first tree, as long as one vertex is not an ancestor of the other, or they can go between vertices in different depth-first trees
    - the **color** of the edge can also tell us what they are
    - **White** indicates a tree edge
    - **Gray** indicates a back edge
    - **Black** indicates a forward or cross edge
* **Topological Sort** can be used by **DFS** to sort a ***directed acyclic graph*** or a **“dag”.** A **topological sort (TS)** of a dag **G = (V, E)** is a linear ordering of all its vertices such that if G contains an edge (*u, v*) then *u* appears before *v* in the ordering. We can view a TS of a graph as an ordering of its vertices along a horizontal line so that all directed edges go from left to right. It is different from the usual kind of “sorting”. A real world example would be of someone getting dressed (your socks must go on first before your shoes, your underwear before your pants, etc.)
  + an edge from ***x*** to ***y*** means one must be done with ***x***  before one can do ***y***
  + a TS algorithm would look something like…
    - Run DFS
    - When a vertex is finished, output it
    - Vertices are output in revers topological order
  + Runtime would be: **θ(V + E)**
  + As each vertex is finished, we insert it into the **front** of a **linked list**
* A **minimum spanning tree** is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight.
  + It is basically a **spanning tree** whose sum of edge weights is as small as possible
  + We basically want a subset of edges that is going to be minimum in cost but also connects all of the vertices
* **Kruskal’s and Prim’s Algorithms** are two commonly used algorithms to solve the **minimum spanning-tree problem**
  + We can easily make each of them run in time **O(E lg V)** using ordinary binary heaps
  + Fibonacci heaps helps Prim’s run in **O(E + V lg V)**
  + **Both are greedy algorithms**
* **No algorithm can find a spanning tree for two un-connected graphs**
* **Kruskal’s Algorithm** – finds a safe edge to add to the growing forest by finding, of all the edges that connect any two trees in the forest, an edge **(u, v)** of least weight
* Below is pseudocode for Kruskal’s



* See page 632 of textbook
* **Prim’s Algorithm** – Operates much like Dijkstra’s to find the shortest paths in a graph. Prim’s has the property that edges in the set ***A*** always form a single tree
  + The tree starts from an arbitrary root vertex ***r*** and grows until the tree spans all the vertices in ***V***. The root chosen should have the lowest cost edge out of all the edges connected to it.
  + Each step adds to the tree ***A***, which is the next vertex that is connected to the already chosen vertices that has the lowest cost available. Once all vertices are written out, the algorithm terminates and gives us the MST
* Below is pseudocode for Prim’s



* See page 635 of textbook
* Start off on the start vertex, the one with the lowest cost edge attached to it, A, and we will “grow greedily” (making the best looking choice at the time)
  + Make dotted lines to nearby nodes and choose the smallest edge i.e. the one with the lowest cost, in this case B
  + make dotted lines from all of B’s nearby nodes and choose the edge with the lowest cost, in this case E
  + Make dotted lines off of E’s nearby nodes and choose the edge with the lowest cost, in this case G….an so on