1. Trees and related elements.

Def 1: An undirected graph G is called a **tree** if it is *connected,* and *it has no cycles***.**

Reminder:

- A graph is said to be **connected** if there exists a path between any pair of its vertices.
- A **cycle** is a path (a walk with no duplicate vertices) that starts and ends in the same vertex.

Alternative definitions of a tree:

If G is a connected undirected graph, the following are equivalent:

(i) G is a tree.
(ii) G has no cycles.
(iii) For every pair of vertices $u \neq v$, there is exactly one path from u to v.
(iv) Removing any edge from G gives a graph which is not connected.
(v) m = n-1

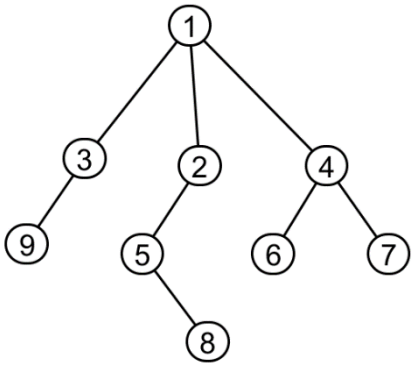

Figure 1

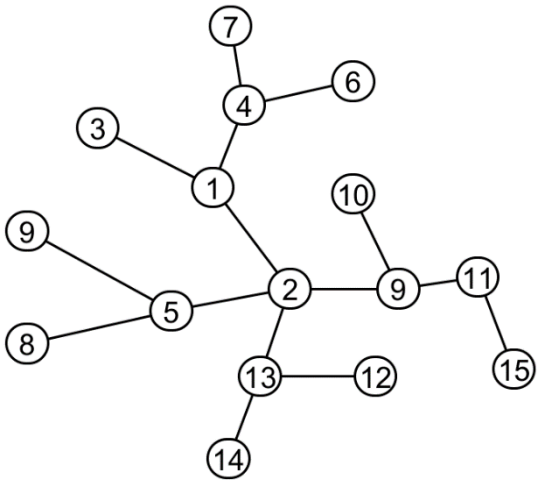

Figure 2

For example, both graphs in Figure 1 and Figure 2 are trees.

Def 2: A **forest** is a graph for which all connected components are trees. In particular, a forest with a single connected component is a tree.

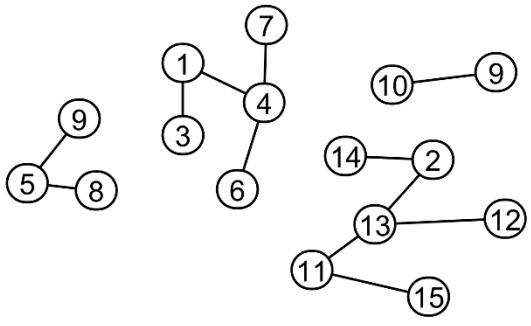

Figure 3

Example – the graph in Figure 3 is a forest.

Def 3: A pair (G, v), consisting of a graph $G = (V, E)$ and a specified vertex v, is called a rooted graph with root v.

Rooting is usually used for trees and usually rooted trees are depicted as the tree on Figure 1, with the root at the top (that graph would be a rooted tree with root 1).

Let (T, r) be a rooted tree. If w is any vertex other than r, let ***r,*** $v_0, v_1, v_2, \ldots, v_k,$ ***w***, be the list of vertices on the unique path from r to w. It is said that v_k is the **parent** of w and w is a **child** of v_k. A vertex with no children is called a **leaf**. All the vertices that are *not* leafs are **internal vertices** of the tree.

2. Spanning Tree

Reminder: Let $G = (V, E)$ be a simple graph. $G' = (V', E')$ is a **spanning** subgraph of G if $V'=V$ (and $E' \subseteq E$ since G' is a subgraph).

Def 4: A spanning tree of a simple undirected graph $G = (V, E)$ is a subgraph $T = (V, E')$ which is a tree and has the same set of vertices as G (i.e. it is a spanning subgraph of G that is a tree).

Since a tree is connected, a graph with a spanning tree must be connected. Or, in other words, a disconnected graph does not have a spanning tree.

On the other hand, every connected graph has a spanning tree (we can always select just the edges that don't form a cycle). => A graph is connected if and only if it has a spanning tree.

The minimum spanning tree problem: Given a weighted undirected connected graph, find a spanning tree of minimum total cost (the sum of the cost of the edges of the spanning tree is the smallest possible of all spanning trees of the graph).

3. Kruskal's algorithm

This algorithm is an algorithm for finding the minimum spanning tree of a graph. The general idea would be to sort the edges by their cost and to add each edge to the tree, one by one, skipping the edges that might form a cycle. The difficult part here is how to test the existence of cycles. Checking if adding an edge would introduce a cycle is really inefficient.

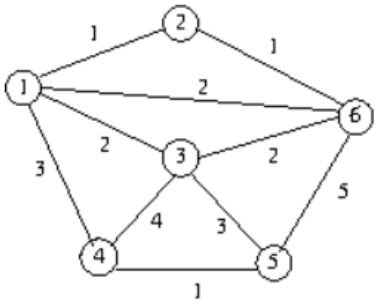

Figure 4

Kruskal took advantage of the fact that linking two separate trees with and edge would result in a tree (it is impossible to introduce a cycle by linking 2 trees with one edge). The only way to introduce a cycle would be by adding an edge connecting 2 vertices that are part of the same tree.

Therefore, the idea of his algorithm is to start with all vertices as trees in a forest (each vertex is a tree with one vertex and no edges) and then continually connect trees in the forest until only one tree remains. To find the minimum tree, we just need to select the edges with lowest cost first.

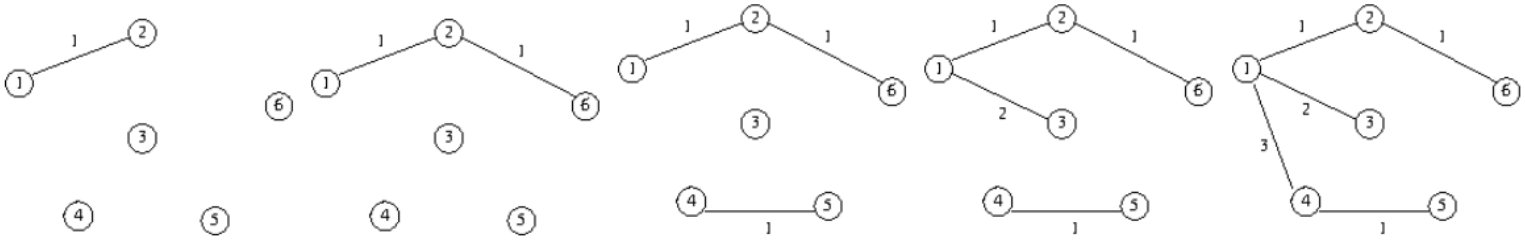

Figure 5

For example, if we consider the graph in Figure 4, in figure 5 it is exemplified how Kruskal's algorithm works: first we have a forest with every vertex as a trivial tree. Then trees {1} and{2} are connected by an edge with cost 1. The next edge with lowest cost is (2,6) which would connect 2 separatetrees: {1, 2} with {6} resulting in tree {1,2,6}. Next edge is (4,5) which connects trees {4} and {5} to tree {4,5}. Next would be (1,6), but both 1 and 6 are from the same tree, so this would inroduce a cycle so it is skipped. Next is (1, 3), which connects tree {1,2,6} with tree {3}. Next is edge (3, 6), but now both of those vertices are part of tree {1,2,3,6}. The next one is (1, 4) and it connects trees {1,2,3,6} with tree {4,5}. Now all vertices are part of the tree, meaning we found the minimum cost spanning tree.

The pseudocode of the algorithm is in the following:

```
Algorithm name: Kruskal's algorithm
Input:
      g: an undirected connected weighted graph
Output:
      t: a graph, the minimum spanning tree of g
Algorithm:
      Graph t
      Set forest #we do not actually need a graph to represent our forest. Just a set
                  of sets, where each set represents a tree and contains the vertices
                  that make up that tree
      For (v1, v2) in g.get_edges(): #we create a list with all the edges
            sorted_edges.append((v1, v2))
      end_for
      sorted_edges.sort({by cost of edge in g}) #sort edges by their cost
      t.set_vertices(g.get_vertices()) #we initialize t to have the same vertices as g
      Int i = 0
      while t.num_edges() < g.num_vertices()-1: # while t has fewer than n-1 edges
            (v1, v2) = sorted_edges[i]
            tree_of_v1 = find_tree(v1, forest)
            tree_of_v2 = find_tree(v2, forest)
            if tree_of_v1 ≠ tree_of_v2:
                  t.add_edge(v1,v2)
                  #combine the 2 sets to represent that they are the same tree
                  make_union(tree_of_v1, tree_of_v2, forest)
            end_if
            i = i + 1
            if i > sorted_edges.length(): throw error "g is disconnected!"
      end_while
```

`find_tree` returns the set (or an id of the set) that contains a vertex. That is needed in order to check whether 2 vertices are part of the same tree or not and add the edge only if they are from different trees. When an edge was added the 2 trees need to be merged – the actual edge where they are merged does not matter only that the sets containing the 2 vertices are now one. `make_union` is the function that does the merging. Both `find_tree` and `make_union` are dependent on the actual implementation and will not be implemented here.

The time complexity of the algorithm does depend on those 2 functions, however. Before that: we have a list that contains all edges sorted by cost. Sorting the edges is Theta(m*log(m)). Then we have the while: that goes through the list of edges, stopping early if the spanning tree is complete, but might have to search through all edges, so it runs O(m) iterations.

Now, how much does each iteration last? It all depends on the functions `find_tree` and `make_union` First, how many times is each function called? `find_tree` is called up to 2m times – it is called 2 times at each iteration and there are up to m iterations. `make_union` on the other hand is called only when an edge is added to the spanning tree. Since the spanning tree has exactly n-1 edges then it is called exactly n-1 times.

So, we have `find_tree` called 2m times and `make_union` called n-1 times. Depending on their implementation we can have different time complexities:

a) If we use the naive implementation, **using set of sets**, find tree has to search for the vertex in all sets, and there are up to n sets. Every time we add an edge a set disappears, but that does change the complexity class, so the find is O(n). Being called 2m times then it is O(m*n).
   For the union we have to add all elements from one set to another. If we always add the elements of the smallest set it can be shown that merging n sets n-1 times to get a final set of n elements is O(n*log(n)) complexity.
   In total we have sort + find + union = Theta(m*log(m)) + O(m*n) + O(n*log(n)) = **O(m*log(m) + m*n)** (the last one is ignored since log(n)<m so n*log(n) is included in m*n)
b) If we use something like **mergeable dictionaries**, for them the find is in log(n) and merge is in log(n). So we have O(m*log(n)) find and O(n*log(n)) merge.
   In total we have sort + find + union = Theta(m*log(m)) + O(m*log(n)) + O(n*log(n)) = **O(m*log(m) + (m+n)*log(n))**
c) We can use **sets of sets, but we can improve the find by using an auxiliary set** that stores what vertex is in what set (using some id for each set). Then the find can be done in Theta(1), therefore up to m finds will be O(m).
   To make this work when merging sets, we also have to change the set of all the added elements (all elements which changed the set). But since this is just a Theta(1) operation on top of the existing add Theta(1) operation called for every added vertex, it will not change the complexity of union, so it is still O(n*log(n)).
   In total we have sort + find + union = Theta(m*log(m)) + O(m) + O(n*log(n)) = **O(m*log(m) + n*log(n))**
   This is the version implemented at the seminar.

4. Prim's algorithm

This algorithm also uses the fact that to avoid cycles when adding edges to a tree, they must be edges that link to the outside of the tree.
Prim's algorithm starts with a single tree consisting in a single vertex (chosen at random), and then grows that tree until it covers all the vertices, by always adding an edge from outside the tree. Among all edges that could be added, it chooses the edge of smallest cost.

For example, considering graph in Figure 5, in figure 7 it is shown how Prim's algorithm works to create a minimum spanning subtree

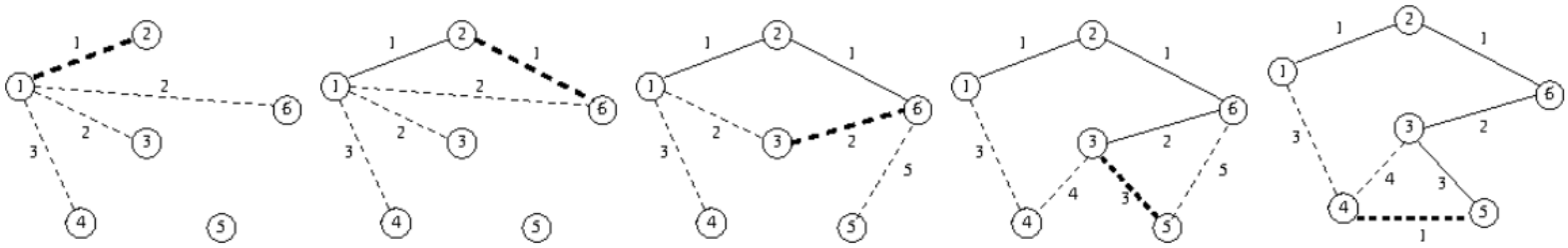

Figure 7

Say it randomly starts at vertex 1. Then the tree only contains that vertex and no edges. All neighbors of 1 are potential edges to add and it chooses the edge with smallest cost: (1,2). After it adds that edge to the graph it adds all the neighbors of 2 to the potential edges to add. It chooses the edge with smallest cost, (2,6) and adds it to the tree. Now all neighbors of 6 are added to potential edges, **except** the neighbors that are already in the tree – that means that edge (1,6) is no longer a potential edge to be added. It then chooses the cheapest potential edge (6,3) and adds it to the tree. The neighbors of 3 outside the tree are added to potential edges while the ones from inside the tree, (3,1), is removed from potential edges to add. The next cheapest potential edge is chosen, (3,5) and added to the tree. Then (5,4) is added to potential edges, as it is a neighbor from outside the tree and (5,6) removed from potential edges as it is from inside the tree. Ultimately (5,4) is added and the spanning tree is complete.

```
Algorithm name: Prim's Algorithm
Input:
      g: an undirected connected weighted graph
Output:
      t: a graph, the minimum spanning tree of g
Algorithm:
      Graph t
      PriorityQueue pq #for the potential edges, priority is the edge cost in g,
                       smallest first
      Dictionary dist  #to not add edges that are clearly worse
      Set visited #the set that contains the vertices in the tree, to avoid cycles
      t.set_vertices(g.get_vertices()) #set vertices of t to be the same as g
      s = 1 #or some other random vertex in g
      visited.add(s)
      for n in g.get_neighbors(s):#initially add all neighbors of s as potential edges
            dist[n] = cost(s, n)
            pq.push((s, n))
      while t.num_edges() < g.num_vertices() – 1: # while t has fewer than n-1 edges
            (v1, v2) = q.pop()
            if v2 not in visited:
```

```
                t.add_edge((v1, v2))
                visited.add(v2)
                for n in g.get_neighbors(v2)
                      #change the distance map and add it to queue if it was not
                        seen before or if the cost is less than other
                      if n not in dist.keys() or cost(v2,n) < dist[n] then
                            dist[n] = cost(v2, n)
                            pq.push((v2, n))
                      end_if
                end_for
          end_if
    end_while
```

Prim's algorithm is kind of a Best First Search, but for edges instead of vertices. It has a priority queue with the edges that it could visit and always adds the best one. For time complexity, A traversal is m+n, but during this traversal we add and remove edges from the priority queue so we have the priority of O((m+n)*log(m))

Note: for all these complexities (including those for kruskal's algorithm) it can be argued that they are O(m*log(m)), since n ≤ m (because it has to be connected there are at least n-1 edges), so all complexities are smaller than the one using m instead of n, so we can replace n with m ad and it still is kind of the same bound (e.g. since n ≤ m, O((m+n)*log(m)) < O((m+m)*log(m)) = O(2m*log(m)) = O(m*log(m)). So all of them are very similar and in O(m*log(m)), but I do prefer the more explicit complexities since it gives more information about how the algorithm behaves.

5. Directed Acyclic Graphs (DAGs)

Def 5: A **directed acyclic graph** (DAG) is a directed graph having no cycles.

Important: A directed graph with no cycles is **not** a tree!
A directed graph is a tree if the underlying graph is a tree (i.e. we ignore edge orientation and if it is connected and has no cycles it is a tree).

For example, in Figure 8 you can se a DAG example as well as a counter example where it has a cycle.

Please note that the definition has no requirement for the graph to be connected. It is possible for a disconnected graph to be a DAG.

One of the most important algorithms for a DAG is topological sorting – it has applications in many different problems, including scheduling, dependency trees, references and citations, finding longest path in DAG and many others.

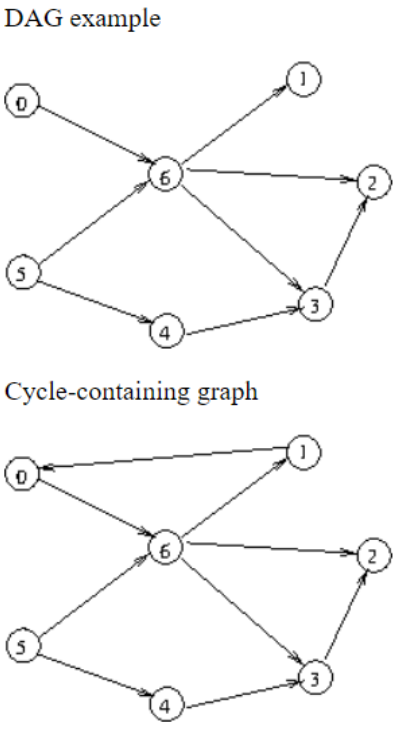

Figure 8

Topological sorting: arrange the vertices of a DAG $G=(V, E)$ in a list such way that $\forall(v1,v2)\in E$, v_1 comes before v_2 in the list.

While there are multiple ways to implement topological sorting, we will review an algorithm based on predecessor counting:

```
Algorithm name: Topological sorting (based on predecessor counting)
Input:
	g: a directed acyclic graph
Output:
	sorted: a list of vertices in topological sorting order,
	throws error if g is not a DAG
Algorithm:
	List sorted
	Queue q
	Dictionary count
	for v in g.get_vertices():
		count[v] = g.indegree(v) #we count the number of predecessors of each node
		if count[v] == 0: #we add the vertices that have no predecesors
			q.push(v)
		end_if
	end_for
	while q is not empty:
		current = q.pop()
		sorted.append(current) #we add to the list the next vertex with no
					predecessors
		for n in g.get_vertices(current):
			count[n] = count[n] – 1 #when a vertex was added to the list, it
				was ‘processed’ so it is not a dependency anymore for its
				neighbors, therefore we decrease their predecessor count
			if count[n] == 0:
				q.push(n) #we add the vertices that have no more predecessors
			end_if
		end_for
	end_while
	if sorted.length() < g.num_vertices():
	  throw error “Not a DAG!” #if we did not reach all the vertices it means we have
				a loop and some dependencies could not be solved
	end_if
	return sorted
end_algorithm
```

For time complexity, we have 2 parts: first, the initialization, and second the main algorithm. For the initialization, it depends on the implementation. For a double list of neighbors, where the indegree is Theta(1) it will take Theta(n) for initialization.

For other implementation we have a problem: for a simple list of neighbors, indegree is O(m+n) therefore we have $O(m*n + n^2)$ initialization complexity. This is a really bad complexity, but it doesn’t have to be. Actually, since we want the indegree for all vertices at initialization, we could do a traversal in Theta(m+n) and for every neighbor just increase the count for the neighbor by 1.

So, we have Theta(n) for double list of neighbors or Theta(n+m) for simple list.