

Extra Point Mini-Project: Graphs and Kruskal's Algorithm in Haskell

Start Assignment

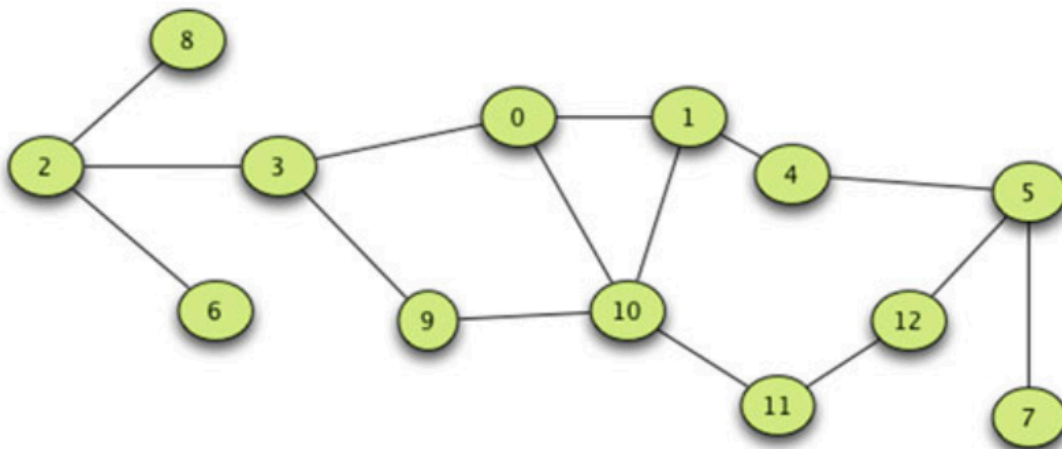
- Due Friday by 11:59pm
- Points 100
- Submitting a file upload
- File Types hs
- Available until Apr 26 at 11:59pm

In this project, you will implement **graph** data structure. Next, you will define **Kruskal's algorithm** on graphs. In the following, we first review the graph data structure, followed by Kruskal's algorithm. Finally, we will describe the project details.

Graphs:

Note: What follows gives a formal definition of graphs, described here for reviewing the concepts. You do not need to stick to this formal definition to implement graphs in Haskell.

A graph $G=(V,E)$ is defined by a set of vertices, named V , and a set of edges, named E . The set of edges are subsets of V where each member of E has cardinality 2. In other words, edges are denoted by pairs of vertices. Consider the simple, undirected graph given in figure below. The sets $V = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$ and $E = \{\{0, 1\}, \{0, 3\}, \{0, 10\}, \{1, 10\}, \{1, 4\}, \{2, 3\}, \{2, 8\}, \{2, 6\}, \{3, 9\}, \{5, 4\}, \{5, 12\}, \{5, 7\}, \{11, 12\}, \{11, 10\}, \{9, 10\}\}$ define this graph. Since each edge is itself a set of cardinality 2, the order of the vertices in each edge set does not matter. For instance, $\{1, 4\}$ is the same edge as $\{4, 1\}$.



Many problems can be formulated in terms of a graph. For instance, we might ask how many colors it would take to color a map so that no two countries that shared a border were colored the same. In this problem the vertices in figure above would represent countries and two countries that share a border

would have an edge between them. The problem can then be restated as finding the minimum number of colors required to color each vertex in the graph so that no two vertices that share an edge have the same color.

A **path** in a graph is a series of edges, none repeated, that can be traversed in order to travel from one vertex to another in a graph. A **cycle** in a graph is a path which begins and ends with the same vertex. A **tree** is a connected acyclic graph. An **acyclic graph** is a graph without any cycles.

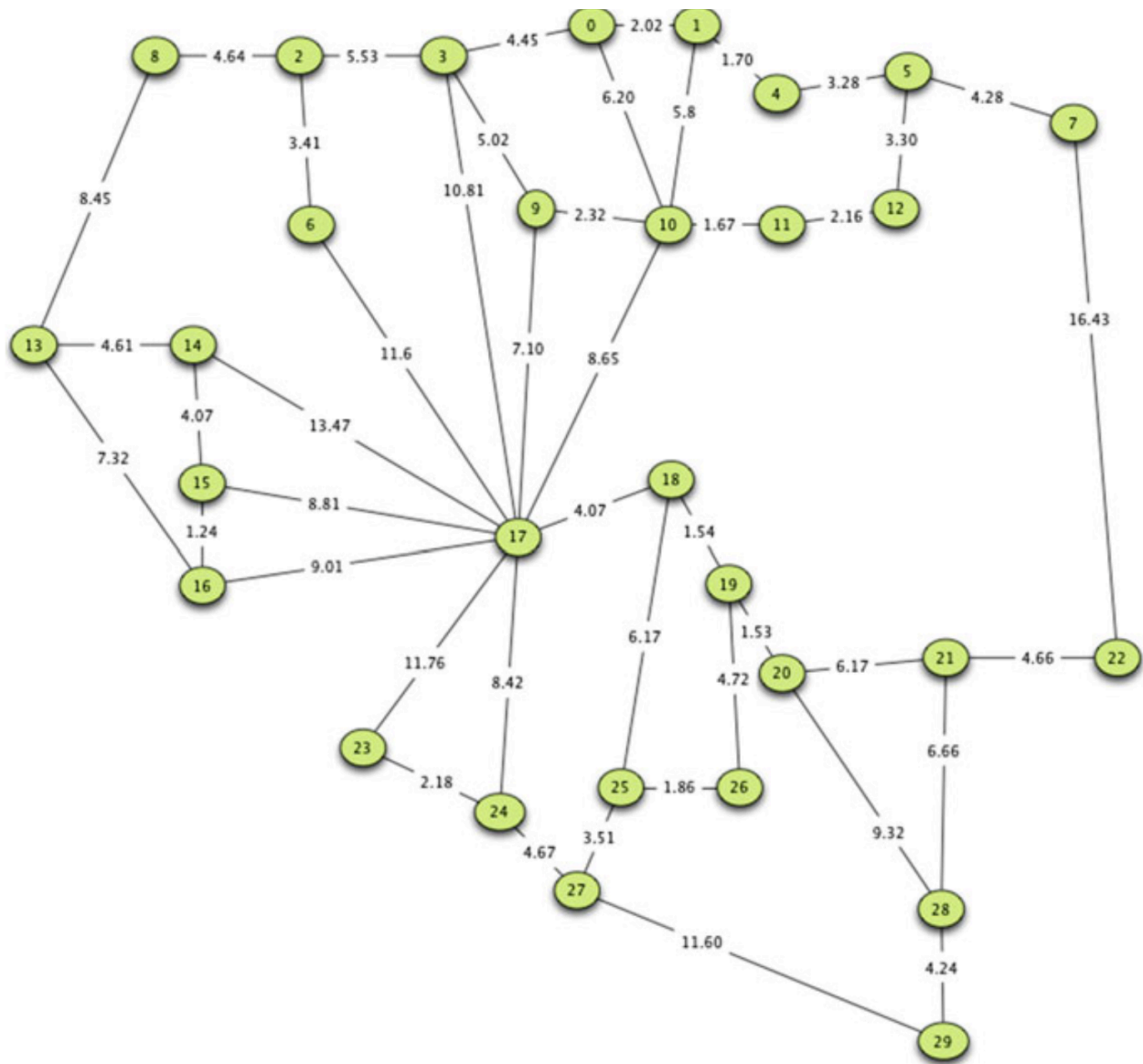
A **weighted graph** is a graph where every edge has a weight assigned to it. More formally, a weighted graph $G = (V, E, w)$ is a graph with the given set of vertices, V , and edges, E . In addition, a weighted graph has a weight function, w , that maps edges to real numbers. For instance, a weighted graph might provide information about roads and intersections.

Kruskal's Algorithm:

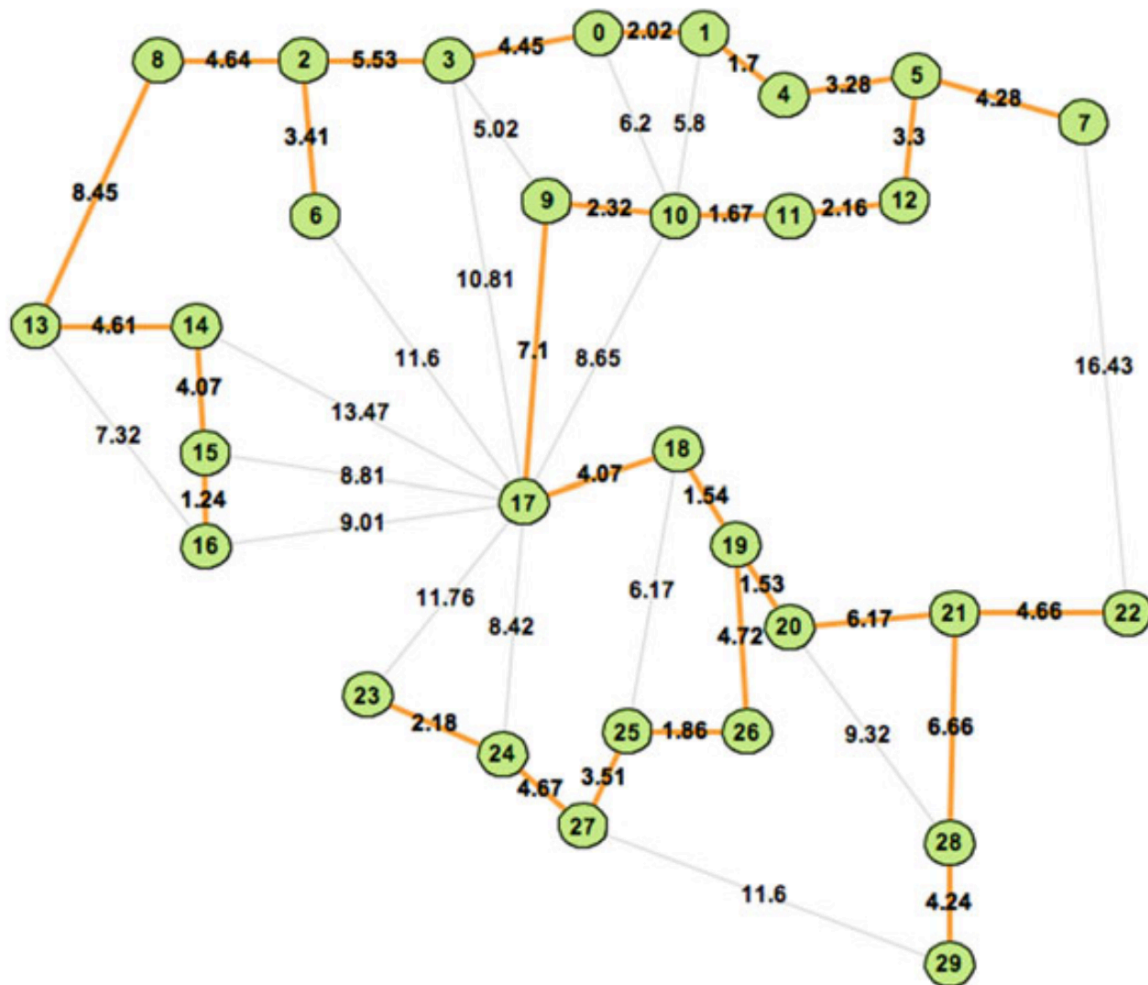
Consider for a moment a county which is responsible for plowing roads in the winter but is running out of money due to an unexpected amount of snow. The county supervisor has been told to reduce costs by plowing only the necessary roads for the rest of the winter. The supervisor wants to find the shortest number of total miles that must be plowed so any person can travel from one point to any other point in the county, but not necessarily by the shortest route. The county supervisor wants to minimize the miles of plowed roads, while guaranteeing you can still get anywhere you need to in the county.

Joseph Kruskal was an American computer scientist and mathematician who lived from 1928 to 2010. He imagined this problem, formalized it in terms of a weighted graph, and devised an algorithm to solve this problem. His algorithm is commonly called Kruskal's Algorithm. Kruskal's paper presented an algorithm to find a **minimum weighted spanning tree** for a graph.

For instance, consider the following graph.



A minimum weighted spanning tree for this graph is given below.



The algorithm details: The algorithm begins by sorting all the edges in ascending order of their weights. Assuming that the graph is fully connected, the spanning tree will contain $|V|-1$ edges. The algorithm forms sets of all the vertices in the graph, one set for each vertex, initially containing just that vertex that corresponds to the set. In the example in graph above there are initially 30 sets each containing one vertex. The algorithm proceeds as follows until $|V|-1$ edges have been added to the set of spanning tree edges.

1. The next shortest edge is examined. If the two vertex end points of the edge are in different sets, then the edge may be safely added to the set of spanning tree edges. A new set is formed from the union of the two vertex sets and the two previous sets are dropped from the list of vertex sets.
2. If the two vertex endpoints of the edge are already in the same set, the edge is ignored since adding that edge introduces a cycle.

The pseudo code for Kruskal's algorithm is given below in imperative style. For more information about the algorithm you may look online, e.g., the wiki page:

https://en.wikipedia.org/wiki/Kruskal%27s_algorithm 

[\(https://en.wikipedia.org/wiki/Kruskal%27s_algorithm\)](https://en.wikipedia.org/wiki/Kruskal%27s_algorithm)


```

algorithm Kruskal(G) is
  F := ∅
  for each v ∈ G.V do
    MAKE-SET(v)
  for each (u, v) in G.E ordered by weight(u, v), increasing do
    if FIND-SET(u) ≠ FIND-SET(v) then
      F := F ∪ {(u, v)} ∪ {(v, u)}
      UNION(FIND-SET(u), FIND-SET(v))
  return F

```

Project Details:

In this project you will define the following:

1. Define polymorphic data type `Vertex a`. This way you can have arbitrary data in graph nodes. One value constructor would suffice. Make this type an instance of type class `Show` and `Eq` implicitly.
2. Define polymorphic data type `Edge a`. This way you can have arbitrary data in graph nodes. One value constructor would suffice. The value constructor must receive two vertices (of type `Vertex a`) and a weight (of type `Float`). Make this type an instance of type class `Show` and `Eq` implicitly.
3. Make type `Edge a` an instance of type class `Ord` explicitly by comparing the weights on the edges. This will be needed to sort out the edges in Kruskal's algorithm. (We have not discussed explicit instantiation of types in type classes. Refer to textbook for how to do this:
<http://learnyouahaskell.com/making-our-own-types-and-typeclasses#typeclasses-102> 
<http://learnyouahaskell.com/making-our-own-types-and-typeclasses#typeclasses-102>.)
4. Define polymorphic type `Graph a`. One value constructor would suffice. The value constructor would receive a list of vertices and a list of edges as input. Make this type an instance of type class `Show` and `Eq` implicitly.
5. Define function `kruskaHelper :: (Eq a) => [Edge a] -> [[Vertex a]] -> [Edge a] -> Int -> Graph a` that receives: 1) a list of sorted edges based on the weights, 2) a list of list of vertices: shows the sublist of vertices connected in the minimum spanning tree (MST), 3) a list of edges: all edges accepted so far as part of the MST, and 4) a number representing number of edges added, where the max is $|V| - 1$. The function returns the MST. This function is used in `kruskal` function as the main engine of the algorithm.
6. Define function `kruskal :: (Eq a) => Graph a -> Graph a` that receives a graph and returns the MST by sorting the edges of the graph, initializing the list of list of vertices, the list of edges, and calling `kruskaHelper`. Use quick sort, merge sort, or heap sort (from previous extra point assignment) for sorting purposes.

Dos and Don'ts:

1. Submit your work as a single `.hs` file.
2. Follow the naming conventions suggested here for types and functions to help with grading.
3. Do not use any library associated with graphs, sorting, etc. The goal is to define the data structures from scratch, only relying on lists (and functions on lists).
4. Feel free to define arbitrary number of helper functions as part of defining the functions requested in the assignment.
5. Annotate the types. This would help you make sure you are defining functions according to the spec. Moreover, if you don't annotate the type, type inference algorithm may come up with a type that is not what you need.
6. Go step by step in defining your functions. After each definition unit test your functions with different input.
7. Note that this is an individual assignment (not group-based).

Graphs and Kruskal's Algorithm in Haskell

Criteria	Ratings		Pts
Define polymorphic data type Vertex a.	10 to >0.0 pts Full Marks	0 pts No Marks	10 pts
Define polymorphic data type Edge a.	10 to >0.0 pts Full Marks	0 pts No Marks	10 pts
Make type Edge a an instance of type class Ord explicitly by comparing the weights on the edges.	20 to >0.0 pts Full Marks	0 pts No Marks	20 pts
Define function kruskaHelper :: (Eq a) => [Edge a] -> [[Vertex a]] -> [Edge a] -> Int -> Graph a	40 to >0.0 pts Full Marks	0 pts No Marks	40 pts
Define function kruskal :: (Eq a) => Graph a -> Graph	10 to >0.0 pts Full Marks	0 pts No Marks	10 pts
Define polymorphic data type Graph a.	10 to >0.0 pts Full Marks	0 pts No Marks	10 pts
Total Points: 100			