Darius Jones
Chad Bloor
Gerardo Carillo
Comp 354
10/16/18

1. Kruskal and Prim's algorithm are both excellent at finding minimum cost spanning trees. Kruskal's algorithm selects the smallest edges in such a way that doesn't create a cycle. Prim's algorithm takes a more natural approach. Prim's starts with an arbitrary starting point and selects the smallest edges in such a way that doesn't create a cycle. Then it moves to the next node and repeats until every node is explored. Both algorithms work on the premise that the given graph is connected. When Kruskal's encounters a non-connected graph, it will return two MCSTs because of it blindly selecting the smallest edges while avoiding cycles. When Prim's is given a non- connected graph, it will return a single MCST. Unlike Kruskal's Prim's moves from node to node exploring the possible edges, this makes it impossible for non-connected nodes to be reached.

2. Prim's algorithm is useful in many real worlds situations. One example that comes to mind is in logistics. Transporting goods by truck requires loading and unloading items onto trucks. The total weight of the truck plus its cargo affects the efficiency of its engine. If we look at this scenario in the form of a graph, we can have positive edges where the truck adds cargo increasing its weight. Negative edges could represent the truck delivering its cargo reducing its weight. A logistics company would benefit from Prim's algorithm in this situation. The company would be able to choose the MCST to save on expenses related to maintenance and fuel cost.

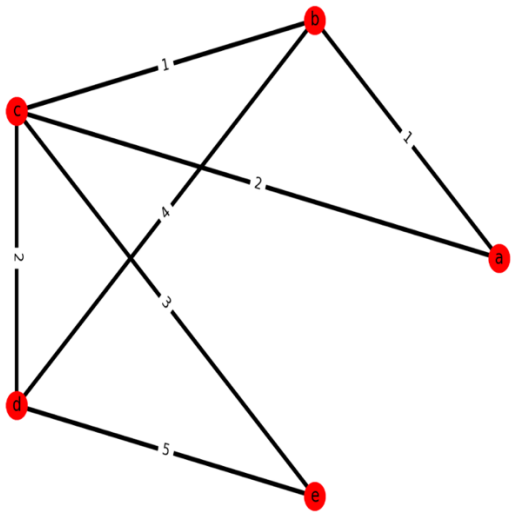
3. We created a class named HeapQ, to encapsulate the heap and its functionality. This class consists of a list initialized with a placeholder in the first slot. The placeholder's purpose is to simplify the math for the methods in the class. For example, the placeholder will occupy the first slot in the list that way we can place the first element in the heap in slot 1. Accessing the right child would be index 2(1) and accessing the left child would be 2(1)+1. Since Prim's algorithm returns an MCST, this implementation is a min heap. A min-heap maintains the property each node is greater than or equal to the value of its parent node; keeping the smallest value at the root.

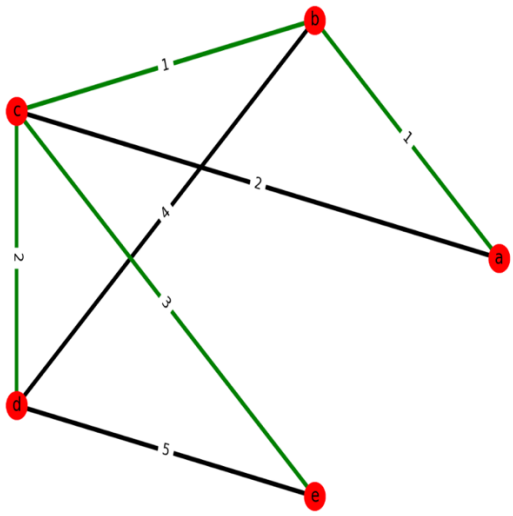
   (weight, frm, to)
   [(1, 'c', 'b'), (2, 'c', 'a'), (3, 'c', 'e'), (2, 'c', 'd')]
   [(1, 'b', 'a'), (1, 'b', 'd'), (3, 'c', 'e'), (2, 'c', 'd'), (2, 'c', 'a')]
   [(1, 'b', 'd'), (2, 'c', 'a'), (3, 'c', 'e'), (2, 'c', 'd')]
   [(1, 'd', 'e'), (2, 'c', 'a'), (3, 'c', 'e'), (2, 'c', 'd')]
   [(2, 'c', 'a'), (2, 'c', 'd'), (3, 'c', 'e')]
   [(2, 'c', 'd'), (3, 'c', 'e')]

[(3, 'c', 'e')]

The output above shows the property is maintained as tuples are pushed and popped from the heap. When an element is pushed onto the heap it is added to the end of the list then it is checked to see if can be floated up the heap. The root is checked first and if this value is less than the root it is then swapped. This procedure repeats until the newly pushed element is placed in its correct slot in the list. Popping an element from the heap in some cases breaks the property of maintaining the smallest value at the root. Since we will always pop the first value in the list we will check if that new root's value holds the property. If it doesn't it needs to sink down to its correct place in the list. We compare the root it's minimum child and if it's larger we swap those values. The min heap is very useful in the implementation of Prim's algorithm because it keeps the smallest weighted edge at the root of the tree and every time you pop from the heap its the smallest value.

Knowing this information and studying how Prim's algorithm works allowed us to realize that the arbitrary starting point will be a node and the heap should be filled with all of the other nodes connecting to it. The starting node and its connecting components are pushed onto the heap before the while loop is executed. We keep a list called bookmark to keep track of the nodes that are visited. The starting node is initially placed in the bookmark list before the while loop is executed. Inside the while loop, we pop from the heap and thanks to its properties the weight can be ignored, so we focus on the "frm" and "to" nodes. If the "to" node is not in the bookmark list, we add it to the bookmark list. Since we are returning the MCST as a dictionary we use "frm" as a key and we add "to" to that key's list. Prim's algorithm considers the lowest edges of the connected components to be added next. The for loop adds the components to the heap and keeping with the heap's property they are floated up to the top of the heap if they are the smallest. This allows the smallest edge of the non-bookmarked components to always be selected. To fully understand what the algorithm was doing we use the networkx library to visualize the graph before and after the algorithm was run. The graph's edges are initially black, the resulting MCST's edges are green.





Prim's
b --> a
b --> c
c --> d
c --> e