CS5110 - Final Project

Hermès Henry & Daniel Murrow

Fall 2024

1 Prim's Algorithm

Prim's Algorithm is an algorithm that defines a method for finding the Minimum Spanning Tree (MST) of an undirected connected graph. The algorithm can be simplified into the following pseudocode at a very high level of abstraction and should output an MST that has N nodes and $N-1$ edges.

High-Level Pseudocode

1. Initialize the MST based on an arbitrarily chosen node in the graph
2. Consider all edges from the graph **not** in the MST that connect to the nodes in the MST
    - ⋆ To avoid cycles, we make sure to check if the edge we are considering connects to a node that is not already connected to by another edge; i.e., only connect to a node that is in the graph but not in the MST.
3. Select the edge with the lowest weight and add the connected node to the MST
4. Repeat Step 2. and Step 3. until the MST contains every node from the original graph

Time Complexity & Data Structure

A simple implementation of this would initialize lists of edges and nodes; For each of the nodes, N, we look at each edge, E, connected to it to find the minimum, leading to a time complexity of $O(N^2)$. We can improve this by using a priority queue instead of a list to keep track of our edges, E, and nodes, N. With a Priority Queue, keeping in mind that each edge is only ever processed once, we see a total time complexity of $O(E \log E)$ where E represents the total number of edges in the input graph. We can further improve the runtime of this algorithm by keeping track of our nodes, N, instead, ordered by the smallest edge-weight connecting them to a node already in the MST; this method leads to a total time complexity of $O(E \log N)$ which logarithmically becomes an improvement as the input graph becomes complete, aka a graph where $E = \frac{N(N-1)}{2}$.

In order to implement the Priority Queue, I used the Python standard library's `heapq` heap queue algorithm. The important functions are `heapq.heappush(heap, item)` and `heapq.heappop(heap)` which are both $O(\log n)$. This specific heap queue is a binary min-heap; as such, it orders values from smallest to largest (i.e., `heapq.heappop(heap)` returns the smallest value in the heap), as opposed to a max-heap which would order values from largest to smallest. Using a Fibonacci heap instead of a binary heap could further increase the time complexity of this algorithm because the `insert` or `push` operation is $O(1)$ in a Fibonacci heap as opposed to $O(\log n)$ for a binary heap. Therefore, a Fibonacci heap combined with the method keeping track of nodes instead of edges, as discussed in the previous paragraph, would have a total runtime of $O(E + V \log V)$.

In both cases for speeding up the algorithm, the speed-up is only significant as E increases from V up to a maximum of $E = \frac{N(N-1)}{2}$. That is, the closer E is to V the more negligible the speed-up from either or both methods will be.

Low-Level Pseudocode

With all this in mind, we can put together a pseudocode with a much lower level of abstraction that we will call `Prims(G)` where G represents the input graph. This final pseudocode should have a worst case time complexity of $O(E \log E)$ when implemented in Python.

Algorithm 1: `Prims(G)`

Input: Undirected Connected Graph G
Output: Minimum Spanning Tree

```
   // Step 1.
 1 start ← randomNode(G)
 2 MST ← new Graph(start)
   // Initialize the PriorityQueue
 3 pq ← new PriorityQueue()
 4 for each edge e in start.edges do
 5     pq.push(e)
   // Repeat Steps 2 and 3 until Nodes in MST are the same Nodes as in graph
 6 while mst.Nodes != graph.Nodes do
 7     edge = pq.pop()
       // Make sure edge is not connected to a Node alreading in MST
 8     if edge.link_node not in mst.Nodes then
 9         mst.add_Node(edge.link_node)
10         mst.add_edge(edge)
           // Add every edge of this new node to the priority queue
11         for edge_new in edge.link_node.edges do
12             if edge_new not in mst.Nodes then
13                 pq.push(edge_new)
   // Finally, return the MST once the while loop breaks
14 return MST
```

Testing

In order to test my implementation of the algorithm, I set a few simple criteria:

1. Does the MST contain the same number of nodes as the graph?
2. Does the MST contain the minimum possible number $(N - 1)$ of edges?
3. Is the MST a connected graph?
4. Is the sum of all the weights in the MST equal to the sum of all the weights in `networkx`'s built-in MST function?

2 Maximum Clique Approximation

3 Girvan-Newman

4 Bellman-Ford