**Prim’s Algorithm to Find a Minimum Spanning Tree Project Report**

Daniel Taylor, Bryon Catlin, Adriana Bottega, Gaspar Chayer

Florida Polytechnic University

COP3530: Data Structures and Algorithms

**Greedy Algorithms and Prim’s Algorithm**

A drawing of a diagram

Description automatically generated Prim’s Algorithm is a type of Greedy algorithm, which is an algorithm that makes a choice that seems optimal at the moment but may not be the overall most optimal solution. For example, in the figure below, the algorithm simply wants to end with the highest number of units collected in a Binary Search Tree. A Greedy algorithm will pick the higher number for each decision, so when faced with three and four, it chooses four, and when faced with fourteen and eleven, it chooses fourteen. However, looking at the other choices, if the algorithm chose three, it could end with twenty, which is the most optimal solution. This is what a non-Greedy algorithm would do.

Figure 1 - Graph showing a greedy algorithm in action

Prim’s Algorithm is a type of Greedy algorithm that is incredibly useful for finding the Minimum Spanning Tree (MST) of an undirected, connected, and weighted graph.

**Minimum Spanning Tree**

A Minimum Spanning Tree (MST) uses the Tree data structure, which is a connected graph of nodes (vertexes). These graphs can be directed, meaning there is a set direction to travel from any given node, or undirected, meaning the algorithm can freely travel from node to node. A Spanning Tree is a subset of the parent graph that connects every node together without forming a cycle. This means that when starting from the first node, there is only one path to travel to get to the last node. However, when the edge (or path) to each node in the tree has an assigned weight, an MST is useful to create a spanning tree with the minimum total edge weight. This can be seen in Figure 2 below.

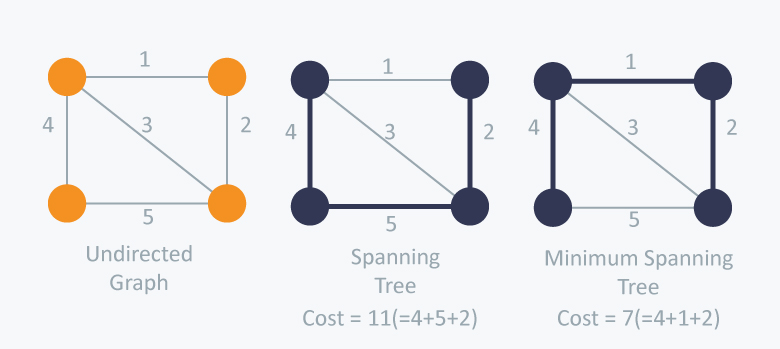


Figure 2 - A tree graph with its Spanning Tree and Minimum Spanning Tree

**Prim’s Algorithm**

Prim’s Algorithm was published by Robert C. Prim in 1957, shown in Figure 3.



Figure 3 - Robert C. Prim, creator of Prim's Algorithm

**Prim’s Algorithm (cont.)**

Prim’s Algorithm works by starting from any vertex (tree node) on the MST graph, then finding the minimum weight edge that connects that vertex to a new one, and then repeating this process until all vertices from the tree have been connected. It can easily be determined how Prim’s Algorithm is an example of a Greedy algorithm because it finds the minimum weight from a vertex and uses that, even though that this may not be the most optimal MST.

**How the Implementation Works Using Prim’s Algorithm with Minimum Spanning Trees**

In our code, we first begin by including the **iostream, vector, list, queue,** and **utility** libraries. We also define the variable “**INF**” as a large constant, 1000 to help with vertex initialization. In the main function, we begin by assigning the value *9* to a variable V, for the number of vertexes in the test graph (Nodes zero to eight). We then create a Graph object g based off the 9 vertexes.

This Graph object’s private section has the number of vertices **(int V)**, and the adjacency list represented by a list of pairs, where the pair is the neighbor vertex, and the corresponding weight required to reach it. In the public section there has a constructor **Graph(int V)** that initializes the number of vertices and dynamically allocates memory for the adjacency list **adj**. There is also a destructor to deallocate the previously mentioned dynamic memory. Then, there is the **addEdge** method to add edges to the graph, the **primMST** method that applies Prim’s Algorithm and prints the output.

Continuing through the main, we then use **addEdge** that adds all the edges needed. The **addEdge** method is a void function that takes the parameters **int u**, **int v**, and **int w**. These parameters are the current vertex, neighbor vertex, and the corresponding weight of the path between those vertexes respectively. We then use the **push\_back** function on the **make\_pair** function from the **<list>** library for the adjacency list to add the edge u-v with weight w, and then do the same with v-u, since the graph is undirected. This is done for all the edges in our test graph.

Lastly in the main, the **primMST** method is calledto apply the algorithm and print the output. Inside the method, a priority queue and vectors are initialized to store the values from the graph. These values are then used to generate an MST using a min-heap priority queue. A priority queue is a useful data structure based off a queue, where elements are inserted into a first-in-first-out (FIFO) order, however with a priority queue, elements are instead ordered by priority, and the highest priority element goes to the top of the queue. A min-heap is a complete binary tree data structure where the parent node must be less than the values of its children nodes. This means that the minimum element in any given data set is at the top of the tree, or the heap.

On the next page is our pseudocode for our implementation of Prim’s Algorithm in our program.

**Pseudocode for the Implementation**

**Class: Graph**

* **Int V**: The number of vertices.
* **List adj**: A list of V elements, where each element is a list of pairs (neighbor vertex, edge weight).

**Function: Graph(V)**

* Initialize **V**.
* Create an empty adjacency list **adj** with V elements.

**Function: addEdge(u, v, w)**

* Add an edge from **u** to **v** with weight **w** to **adj[u]**.
* Add an edge from **v** to **u** with weight **w** to **adj[v]**.

**Function: primMST()**

* Create a priority queue **pq** to store pairs (weight, vertex).
* Create a list **value** of size **V** initialized to **INF** to store minimum weights of vertices.
* Create a list **parent** of size **V** initialized to **-1** to store the parent of each vertex.
* Create a list **inMST** of size **V** initialized to **False** to track if a vertex is included in MST.
* Start from the source vertex:
  + Set **value[0] = 0** (start with vertex 0).
  + Push **(0, 0)** into the priority queue (value, vertex).
* Process the priority queue:
  + While the priority queue is not empty:
    - Extract the vertex **u** with the smallest weight.
    - Skip if **u** is already in MST.
    - Mark **u** as included in the MST.
    - For each neighbor **v** of **u**:
      * If **v** is not in MST and the edge (**u**, **v**) has a smaller weight than **value[v]**:
        + Update **value[v]** and **parent[v]**.
        + Push **(weight, v)** into the priority queue.
* Output the MST:
  + Use the **parent** array to print the edges of the MST.

**Main Function:**

* Define the number of vertices **V = 8**.
* Create a graph **g** with **V** vertices.
* Add edges to the graph **g** using **addEdge(u, v, w)**.
* Call the function **primMST()** to find and print the MST.

**Algorithm Workflow:**

* **Start from the source vertex**:
  + Set **value[0] = 0** (start with vertex 0).
  + Push **(0, 0)** into the priority queue (value, vertex).
* **Process the priority queue**:
  + While the priority queue is not empty:
    - Extract the vertex **u** with the smallest weight.
    - Skip if **u** is already in MST.
    - Mark **u** as included in the MST.
    - For each neighbor **v** of **u**:
      * If **v** is not in MST and the edge (**u**, **v**) has a smaller weight than **value[v]**:
        + Update **value[v]** and **parent[v]**.
        + Push **(weight, v)** into the priority queue.
* **Output the MST**:
  + Use the **parent** array to print the edges of the MST.

**Real World Applications for Prim’s Algorithm and Minimum Spanning Trees**

Prim’s Algorithm is incredibly useful in the real world. One example of it is being used to find the shortest path between computers when making a communications network (Figure 4). This would minimize the distance needed when communicating and connecting between these computers, which would minimize the cost and routing of data packets through the network. This increases efficiency and decreases delays between communications. It can also be used for air traffic control, pipeline making, and social network optimization (Figure 5).

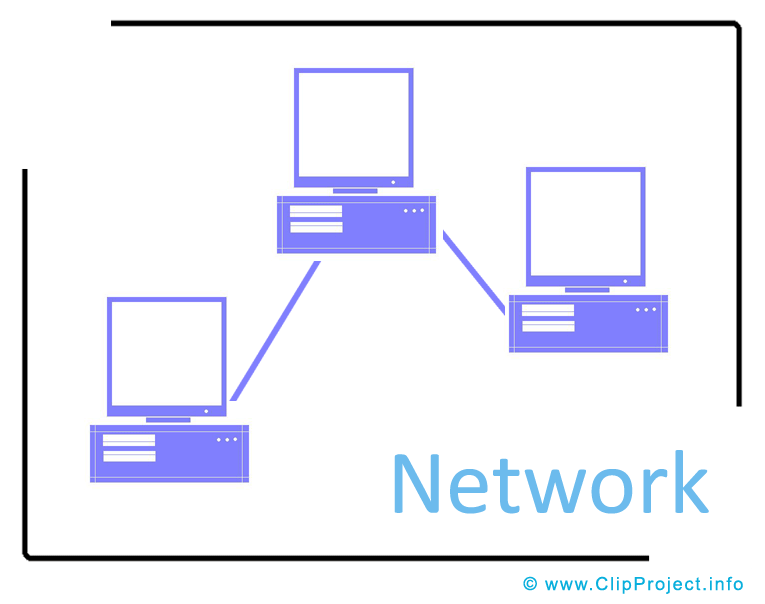




Figure 5 - Example of a connected social network (not an MST)

Figure 4 - Example of computers connected in a network (MST)

**References**

NB, Tantoluwa Heritage Alabi. “What Is a Greedy Algorithm? Examples of Greedy Algorithms.” *freeCodeCamp.Org*, freeCodeCamp.org, 4 Sept. 2024, www.freecodecamp.org/news/greedy-algorithms/.

“Minimum Spanning Tree Tutorials & Notes: Algorithms.” *HackerEarth*, www.hackerearth.com/practice/algorithms/graphs/minimum-spanning-tree/tutorial/. Accessed 24 Nov. 2024.

Chris, Kolade. “Prim’s Algorithm – Explained with a Pseudocode Example.” *freeCodeCamp.Org*, freeCodeCamp.org, 14 Feb. 2023, www.freecodecamp.org/news/prims-algorithm-explained-with-pseudocode/#heading-what-is-prims-algorithm.

“10 Project Ideas That Use Prim’s Algorithm.” *GeeksforGeeks*, GeeksforGeeks, 28 Aug. 2024, www.geeksforgeeks.org/project-ideas-that-use-prims-algorithm/.