Prim’s Algorithm

3EA3 Software Specifications and Correctness: Final Report

Author: *Paul Warnick - 001300963*
Instructor: *Musa Al-hassy*

Introduction

The focus of this report will be on finding the minimum spanning tree (MST) of weighted undirected graphs using Prim's algorithm. I'm going to begin with an overview of what Prim's algorithm is, including its real world applications and specific details on how it works. Also, to understand the previous section I've included some information on the qualifications of creating a minimum spanning tree. I briefly touch on working with graph data types and understanding how regular spanning trees work. I've also included a small section devoted to Kruskal's algorithm including some of the similarities and differences between the two. Finally I'll be moving onto an actual implementation in Alloy (a specification language for declarative modelling). There will be a brief section describing the tool itself followed by a description of the actual model. When working with Alloy the goal was to prove two main features of the algorithm, first being that it does indeed create a spanning tree based on an input graph and second that we can ensure the spanning tree is itself minimal in regards to its edge weights sum.

My inspiration behind choosing Prim's algorithms was to get a broader understanding of how different algorithms are used in the real world. Originally, I planned to uses Dijkstra's shortest path algorithm, an algorithm used for finding the shortest path tree of a graph (i.e. the shortest path from a starting node to every other node in the graph). I've had a few experiences working with Dijkstra's in the past and I figured it'd be interesting to learn more about it. Fortunately, I happened to stumble across Prim's, which really caught my attention because of its different, but still incredibly useful, real world applications. In completing this project I was able to learn significantly more than I expected and it showed me the greater scope of algorithms that are available for use for a huge variety of problems.

For the entire project repository (including source code and instructions on how to run the model) please click here.

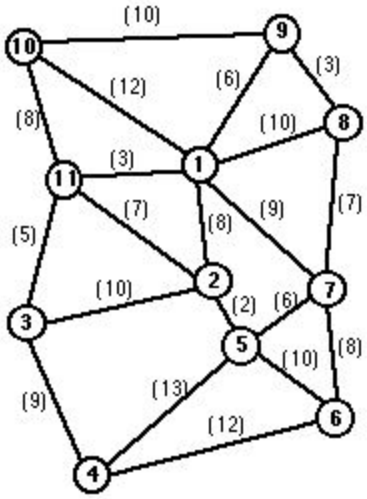
Prim's Algorithm Explained

History

Prim's algorithm was developed in 1930 by a Czech mathematician named Vojtěch Jarník but was later rediscovered and officially published in 1957 by computer scientist Robert Prim. The algorithm itself can be used to find the minimum spanning tree of a weighted undirected graph in a simplistic and efficient manner that's easy to understand. The time complexity of this algorithm is $O(E \log V)$[1] where E represents the number of edges in the graph and V represents the number of vertices. Having such a fast time complexity means Prim's algorithm is ideal when working with dense graphs having a large number of edges (i.e. $E = O(V^2)$).

Minimum Spanning Trees

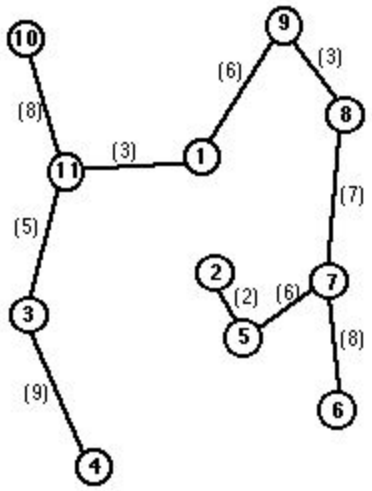
In order to understand how Prim's algorithm works, first we must understand the end goal, a minimum spanning tree (MST). Imagine you are given the following graph:



First, if we wanted to simply find a "spanning tree" of this graph, we would select the smallest possible subset of all edges[2] while ensuring that there still remains a path between each node (i.e. every node in the graph can still be reached by every other node)[3]. In this sense

[1] https://www.quora.com/Is-The-time-complexity-of-Prims-algorithm-the-same-as-Kruskals-algorithm
[2] https://www.tutorialspoint.com/data_structures_algorithms/spanning_tree.htm
[3] https://www.me.utexas.edu/~jensen/exercises/mst_spt/mst_spt.html

our tree *spans* across every node in our graph. In order for us to find the MST, we add another condition that the sum of the weights of our subset of edges must also be minimized. For example:



Here we can see the sum of all the edges is 57 and no other subset selection will produce a tree with a smaller edge weight sum.

How it Works[4]

As mentioned earlier, Prim's algorithm is surprisingly simplistic to the point where the entire procedure can be described in as little as three steps:

Step One: Initialize

To begin, we select any node at random to be our starting point. This node will become first entry to the set COVERED. All of the remaining nodes in the graph are added seperate set UNCOVERED.

Step Two: Selection

Now consider all of the edges that cut across from the set COVERED to our other set UNCOVERED (i.e. each of these edges will have one node in COVERED and the other in UNCOVERED). Determine which of these cutting edges has the smallest weight. Add this edge to the tree and move whichever of it's two nodes was in the UNCOVERED set to the COVERED set.

[4] https://www.me.utexas.edu/~jensen/exercises/mst_spt/mst_demo/mst1.html

Step Three: Repeat Until Complete

If at anypoint the COVERED set contains all of the nodes in the tree (i.e. every node has been covered) then you will now have the minimum spanning tree for the input graph. If the UNCOVERED set is still populated, repeat Step Two.

Real World Applications[5]

To give some context of the importance of Prim's algorithm I wanted to include a section of where it could be applied in the real world. The best example that helped me understand this was of an engineer attempting to design a network for a telecommunications company. Imagine this company is trying to lay internet lines within a new subdivision with the requirement that they can only build these line on pre existing roads. The total cost of the project is based on how many kilometers of line the company lays down so it's the engineer's job to find the optimal solution with the least cost. Essentially, if he imagines the intersections of this new subdivision to be nodes in a graph, then he can treat all of the roads as edges with their lengths acting as a weight. By using Prim's algorithm he can create a minimum spanning tree of the road network which will in turn have the smallest possible cost to the telecommunications company.

Variations and Similar Problems

Kruskal's Algorithm[6]

Kruskal's algorithm is another popular choice for finding the MST of a graph. It's time complexity of $O(E \log E)$ is comparable to Prim's algorithm and it's the ideal choice if working with a graph that has a small number of edges. The steps to Kruskal's algorithm are as follows:

1. Start by sorting all of the edges in the graph in non-decreasing order of their weights
2. Choose the smallest edge that hasn't already been analyzed and check to see if it forms a cycle[7] with the tree so far. In the event a

[5] https://www.geeksforgeeks.org/applications-of-minimum-spanning-tree/
[6] https://www.geeksforgeeks.org/greedy-algorithms-set-2-kruskals-minimum-spanning-tree-mst/
[7] A cycle within a graph is a path of edges and vertices where a vertex can be reached by itself

cycle would be formed, discard this edge and move on. If a cycle is not created, add this edge to the tree.

3. Repeat the second step until the number of edges in the tree is one less then the number of nodes in the graph. The tree will now be an MST

An Implementation in Alloy

In order to better understand Prim's algorithm I've included an implementation in Alloy (a specification language used for declarative modelling).

About the Tool[8]

Alloy is a popular specification language created in 1997 by Daniel Jackson that can be used for expressing complex constraints over a variety of systems. Alloy uses a declarative approach to modelling, essentially asking the question, "How would I recognize that X has happened?" vs an imperative approach of "How would I make X happen?". The idea behind writing models is to fully describe a small portion of a system and ensure that this section can be constrained to a point in which you're certain there are no issues. By following this approach we are able to create better, more efficient programs, and eventually provide provable correctness to an entire system. In some situations, like safety critical systems, this could result in the difference between life and death.

Goals of the Model

Essentially, by modelling the behaviour of Prim's algorithm we're trying to prove two main ideas:

1. The final output tree is spanning in the sense that every node can be reached by every other node while ensuring the graph is acyclic.
2. The final output tree is minimal in the sense that no other combination of edges and nodes can have a smaller weight sum without compromising the first condition.

[8] For an in depth analysis of Alloy please visit the GitHub repository here

Implementation[9]

The implementation can be found below. Please note that comments are represented by green text describing the functionality of each section of the code.

```
/*
Imports the ordering module to be used with the Time signature below
*/
open util/ordering[Time]

/*
  By including ordering, Time is used to represent each state of the algorithm with the
  initial time (x.first) being before the first step of the algorithm has run and the
  final time (x.last) being the final stage of the algorithm (i.e. Once the MST has been found).
*/
sig Time {}

/*
  The first structural element of model, here we are modelling the nodes in a graph and defining
  a relation "covered" which will represent the set of nodes that have already been added to our tree
  (meaning the nodes that the algorithm has already considered). Additionally, this relation will include
  at which time each node has been covered. For example: Only Node1 might be covered in the
  initial state while the entire set of nodes (Node1, Node2, ... , NodeN) should be covered in the
  final state.
*/
sig Node {covered: set Time}

/*
  Our second structural element of the model, edges represent a link between two other nodes. For
  the purpose of our algorithm we need each edge to have a weight to represent the value of the
  edge. A real world example could be the length of a highway between two cities. Within the body
  of the signature we specify the weight to be a non-negative integer (as is the requirement of our
  algorithm). Additionally, we have another relation "nodes" which represents the two nodes the
  current edge connects. Within the body we specify each edge to have 2 nodes. Finally, another
  relation "chosen" represents at which time state each edge was chosen (i.e. added to the
  tree). For example: There should be no chosen edges in the initial state and the final state of
  chosen should represent our MST.
*/
sig Edge {weight: Int, nodes: set Node, chosen: set Time}  {
   weight >= 0 and #nodes = 2 // Ensures weight is a non-negative int and each edge has 2 nodes
}
```

[9] Please visit here if you're interested in running the model for yourself

```
/*
  This predicate is used to check whether an edge has one node that is covered and another that
  has not yet been covered (i.e. it's cutting between the set of covered nodes and the set of
  uncovered ones). This is a major aspect of Prim's algorithm as it works of the basis of selecting
  the cutting edge with the least weight at every step. Within the body, the left half of the "and"
  represents the nodes within the covered set at time "t" while the right half represents the set of
  node not in the covered set at time "t".
*/
pred cutting (e: Edge, t: Time) {
    (some e.nodes & covered.t) and (some e.nodes & (Node - covered.t))
}

/*
  The step predicate is used to describe the behaviour of our model at each "step" of the
  algorithm's execution. Here, " t " and " t' " will be used to represent the state of the model in
  consecutive time instances (more detail in "fact prim" below). There are two options within this
  predicate with the first being "Condition One" and the second being "Condition Two".

  Condition One represents the state of the model when the algorithm is finished, specifically, if all the
  nodes have been covered, then the nodes covered at time t' is equal to the nodes covered at time t,
  AND the edge chosen at time t' equals the edge chosen at time t.

  Condition Two represents each state of the model while the algorithm is executing, specifically, the
  edges chosen at time t' is equal to the edges chosen at time t PLUS a "new edge" AND the nodes
  covered at time t' equals the nodes covered at time t PLUS the new node introduced by the new
  edge.
*/
pred step (t, t': Time) {
    // Condition One
    covered.t = Node =>
        chosen.t' = chosen.t and covered.t' = covered.t
    // Condition Two
    else some e: Edge {
        /*
          Here we use our cutting predicate from above to find the "new edge". We specify this new
          edge has to be a cutting edge AND it there must not be another cutting edge with a smaller
          weight
        */
        cutting[e,t] and (no e2: Edge | cutting[e2,t] and e2.weight < e.weight)
        // The chosen set at time t' is equal to the previous chosen set plus our above edge "e"
        chosen.t' = chosen.t + e
        // The nodes covered at time t' equals the previously covered nodes plus the nodes of "e"
        covered.t' = covered.t + e.nodes}
}
```

```
/*
  Here we list all of the constraints to ensure our model behaves to the specifications of Prim's
  algorithm. The first constraint tells Alloy that the initial state of the covered set should contain only
  one element (i.e. our starting node) and that the initial state of the chosen set should be empty (we
  haven't chosen any edges yet). The second constraint specifies that all pairs of consecutive time
  instances (t and t.next) satisfy the step predicate. Lastly, the final constraint tells Alloy that in the
  final state, all of the nodes in our graph should be covered.
*/
fact prim {
    one covered.first and no chosen.first
    all t: Time - last | step[t, t.next]
    covered.last = Node
}

// The below section is used to test our code to ensure we've properly modelled Prim's algorithm

/*
  First we need to ensure our model generates a spanning tree such that:
   - (First Condition) The final set of chosen nodes must match the set of all nodes
   - (Second Condition) The tree must contain the smallest number of edges possible, specifically,
     the total number number of edges is ones less then the number of nodes in the graph
   - (Third Condition) All nodes are reachable by every other node
*/
pred spanningTree (edges: set Edge) {
    // First Condition
    (one Node and no Edge) => no edges else edges.nodes = Node
    // Second Condition
    #edges = (#Node).minus[1]
    // Third Condition
    let adj = {a, b: Node | some e: edges | a + b in e.nodes} |
      Node -> Node in *adj
}

/*
  We check that the previous predicate holds for the final state of the chosen nodes (limited scope)
*/
correct: check { spanningTree [chosen.last] } for 5 but 10 Edge, 5 Int

/*
  Secondly, we need to ensure the model generates a Minimal Spanning Tree (MST). We check
  that for all combinations of edges that satisfy a spanning tree, none exist with a total weight sum
  less then the total edge weight sum of our selected "chosen" edges. Notice again we've specified
  a limited scope: **This operation is very robust and takes around 20 minutes to complete**
*/
smallest: check {
    no edges: set Edge {
       spanningTree[edges]
       (sum e: edges | e.weight) < (sum e: chosen.last | e.weight)}
} for 5 but 10 Edge, 5 Int
```

After executing both of our check conditions we can see Alloy was unable to find any counterexamples for our assertions. This tells us that our model has been properly constrained and is correct within our specified scope.

Conclusion

Overall, I found this project to very useful as a means to better understand different varieties of algorithms and how they can be applied to real world situations. I had to develop a deep understanding of Prim's algorithm in order to properly understand the alloy code. This involved learning more than I previously knew about graphs (undirected vs directed), trees (spanning and mst), and other properties like how graph cycles can be created. I look forward to working with declarative modelling again in the future and by completing this assignment I've had the chance to get some hands on experience with little consequence if I made mistakes.

References

GeeksforGeeks. (2018). *Applications of Minimum Spanning Tree Problem - GeeksforGeeks*. [online] Available at: https://www.geeksforgeeks.org/applications-of-minimum-spanning-tree/ [Accessed 22 Apr. 2018].

GeeksforGeeks. (2018). *Greedy Algorithms | Set 2 (Kruskal's Minimum Spanning Tree Algorithm) - GeeksforGeeks*. [online] Available at: https://www.geeksforgeeks.org/greedy-algorithms-set-2-kruskals-minimum-spanning-tree-mst/ [Accessed 22 Apr. 2018].

Me.utexas.edu. (2018). *Minimal Spanning Tree Demo*. [online] Available at: https://www.me.utexas.edu/~jensen/exercises/mst_spt/mst_demo/mst1.html [Accessed 22 Apr. 2018].

Me.utexas.edu. (2018). *MST and SPT*. [online] Available at: https://www.me.utexas.edu/~jensen/exercises/mst_spt/mst_spt.html [Accessed 22 Apr. 2018].

Prim's Algorithm. (2018). *Prim's Algorithm*. [online] Available at: http://http%3a%2f%2fwww.xfront.com%2fAlloy%2fUsing-Alloy-to-model-the-Prim-algorithm.pptx&p=DevEx.LB.1,5359.1 [Accessed 22 Apr. 2018].

Quora. (2018). *Is The time complexity of Prim's algorithm the same as Kruskal's algorithm?*. [online] Available at: https://www.quora.com/Is-The-time-complexity-of-Prims-algorithm-the-same-as-Kruskals-algorithm [Accessed 22 Apr. 2018].

www.tutorialspoint.com. (2018). *Data Structures and Algorithms Spanning Tree*. [online] Available at: https://www.tutorialspoint.com/data_structures_algorithms/spanning_tree.htm [Accessed 22 Apr. 2018].