Finding the minimum-cut in an undirected graph using the Nagamochi–Ibaraki algorithm

Saideep Sambaraju, MS in Computer Science, The University of Texas at Dallas,
email:sxs125932@utdallas.edu

***Abstract*— This application helps in identifying the minimum cut value for an undirected graph(G) using the Nagamochi-Ibaraki algorithm. We then proceed to study the cut algorithm by studying the behavior of the edge connectivity value λ(G) plotted against the graph density d. We also try to study the behavior of the critical edges value against d.**

I. INTRODUCTION

THIS project attempts to measure the network reliability by finding the minimum cut of a network graph G. The minimum-cut is the number of edges that can be removed from a graph to reduce the connectivity of the graph. Consider a network of V nodes. Any pair of nodes (i, j) amongst these V nodes would have an adjacency value a_{ij} associated with the number of edges connecting i and j.

Let λ(G) be the edge-connectivity of the graph G i.e. the number of edges that are to be removed in order to render the Graph G as disconnected.

Let λ(i, j) be the connectivity between nodes i and j i.e. the number of edges to be removed so that there is no path from i to j (or j to i given the undirected graph).

Let G_{ij} be the graph obtained by contracting vertices i and j. Here by contracting we mean merging node i and node j into one single node ij such that all the edges incident on a i and all the edges incident on j are now incident on ij.

The goal of this project is to compute λ(G) for a variety of test cases and study its behavior against the density of the graph which is defined as the average number of edges incident on any node of the graph.

II. APPROACH

There are a variety of approaches used in this project. We primarily follow the Nagamochi–Ibaraki[1] approach as detailed in the paper.

The Nagamochi-Ibaraki[1] approach uses the following result

For any i and j
$\lambda(G) = \min\{ \lambda(i , j), \lambda(G_{ij}))\}$

As we can see the problem here is picking the right i and j. For V vertices we can get V(V-1)/2 i and j pairs(given that it is an undirected graph). The Nagamochi-Ibaraki[1] algorithm simplifies this for us by introducing the concept of (Maximum Adjacency ordering) MA ordering with which we can arrive at the result within V-1 iterations.

The Maximum Adjacency(MA) ordering of a set of nodes - v1,v2 … vn can be found by the following steps

i. Pick a node v1 at random
ii. Once {v1,….vi} are already chosen pick vi+1 such that the maximum number of edges are connecting vi+1 to the set {v1,….vi}

We have another interesting result from this which is
Given an MA ordering v1,…vn

$$\lambda(vn-1,vn) = d(vn) \qquad (1)$$

where d(vn) is the degree of node vn.

For this approach to work we first define the base case as follows

If i and j are the only two nodes in the entire graph then

$\lambda(i, j) = a_{ij}=a_{ji}$

For all other cases, from a list of k remaining nodes, find the Maximum Adjacency ordering

III. DATA STRUCTURES

The solution can better be described by understanding the data structures involved.

Much of the code is reused from aatn-project1[2], the graph data structure used in [2] has been modified to include connected components code.

The primary data structures involved in the solution

a. Graph : Used to store network graph in the form of an adjacency matrix, and methods for initializing the graph (an empty graph with no edges for this experiment)
   Graph
   a. adjMatrix : Stores the number of parallel edges between any two nodes i and j given by a_{ij}

Given that this is an undirected graph $a_{ij}=a_{ji}$ for any i and j.

b. isConnected() – This method returns a boolean value indicating if the graph is connected or not.

With these two modifications to the code used for project 1[2] we can use the Graph object for representing the undirected graph for this problem.

IV. ALGORITHM

There are two primary steps involved in the algorithm

a. Test Case Generation
b. Graph connectivity λ(G) computation
c. MA ordering computation

a. Test Case Generation:

Test case generation follows the specific set of requirements mentioned in the problem statement.

```
V : initialize to 30
Initialize a set of array lists to store the statistics from
each iteration of the experiment
densities: to store the density of the graph at each
iteration
LambdaG : to store λ(G) values for each graph
criticalEdges : to store the number of critical edges in
each experiment
for m := 60,65,70…600
    Initialize graph G with V=30, all
    G.adjMatrix[i, j] :=0
    edges:=m
      while edges > 0
        randomly pick i and j from 0 to V-1
        set G.adjMatrix[i, j]:= G.adjMatrix[i, j] + 1
        set G.adjMatrix[i, j]:= G.adjMatrix[j, i] + 1
        edges := edges – 1
    Let S:= set of all vertices of the graph
    criticalEdges := 0
    LambdaG:=LambdaG ∪{computeLambda(G, S)}
    densities := densities ∪{ 2*(m)/V}
```

b. Graph connectivity λ(G) computation:

For graph connectivity computation we use a recursive method. For each of the recursive calls the arguments are

```
computeLambda(G, S)
    if sizeof(S) == 2 : only two nodes are remaining
      return G.adjMatrix[S.last, S.last – 1]

    MA := initialize an empty list MA for Maximum
    adjacency ordering

  createMAOrdering(MA, G, S): Fills the MA list with
  elements in MA order

  x:= last element of MA
  y:= last but one element of MA
  by (1) we can conclude that
  λxy := degreeOf(y)

  edgeCriticality:= G.adjMatrix[x, y]
  Gxy:=merge(x, y, G) : will merge the nodes x and y in
  graph G
  λ(Gxy):= computeLambda(Gxy))
  If λxy < λ(Gxy)
    criticalEdges := edgeCriticality + criticalEdges
    return λxy
  else
    return λ(Gxy)
```

c. MA Ordering:

The MA Ordering algorithm needs a graph and a set of vertices an empty MA list as input

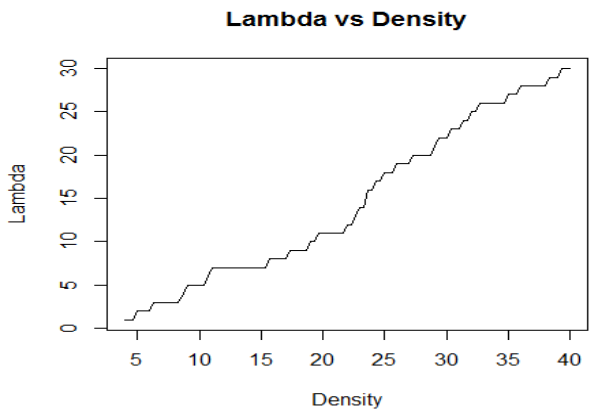
```
createMAOrdering(MA, G, S)
    v1 := pick v1 randomly from S
    MA:= MA ∪{v1}
    For i  in 1..S.size – 1 : for the remaining S.size-1
  nodes
      Max : maximum edges to MA
      Maxind: element with max degree to MA set
      For s in S
        If MA contains s
          Continue

        For m in MA
          If(max < G.adjMatrix[s,m])
              Max := G.adjMatrix[s.m]
              Maxind:=s
      MA:=MA ∪{Maxind}
```

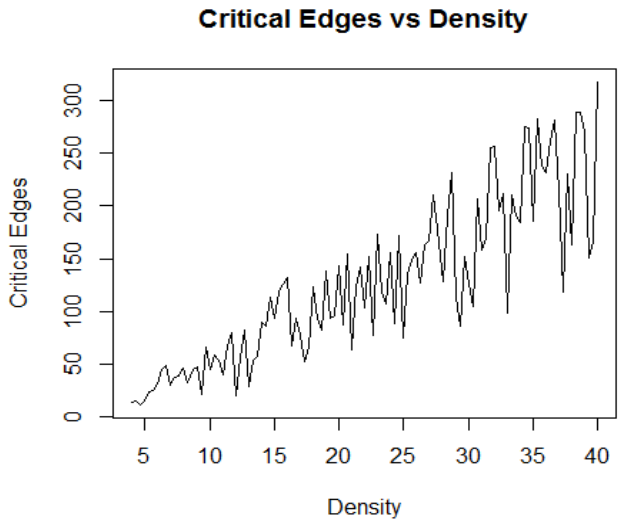
V. RESULTS

This program has been tested on m values [60,65,..,600] and with V=30. The results are computed and stored in files. The following diagrams and graphs were obtained by running R language scripts over these files. All files are available through github repository made for the project[3].

i. λ vs density graphs



ii. Critical edges vs density graphs



VI. Observations from the results

i. λ Dependence on density

λ tends to be non-decreasing on density. This was expected because as m increases, the density of the graph increases as well. This means more edges are being added to the graph, hence λ should be at least as big as the previous value, if anything new parallel edges being added to the critical links of the graph need to be removed so λ would either increase or stay the same.

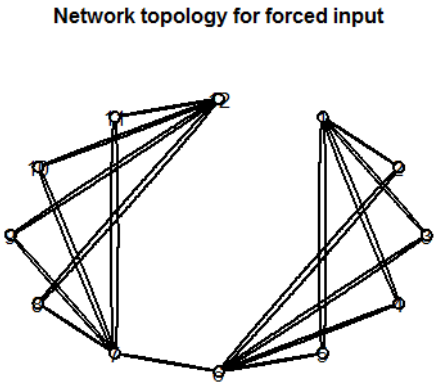
ii. Critical Edges vs Density

The critical edges on the other hand show a more erratic trend than the λ values. The overall trend is still increasing for higher density graphs exhibit higher number of critical edges. This because of two reasons a) the connectivity of graph increases with more density and b) higher connectivity of graph means there is more choice for which of the edges will become critical edges critical edges.

The spikes that can be observed in short intervals can be explained as thus. As we can see from the previous graph, as the m value increases and new edges are added to the graph, the λ value either increases or stays the same, take the cases where the λ value stays the same. This means new edges are added to the graph and some of these new edges may be added to the nodes whose degree is λ, now these nodes are no longer critical. The other scenario is new edges are added and λ value itself has probably increased, and the nodes that were previously not part of critical edges will now become part of critical edges. This tradeoff might explain how the in spite of the increasing trend exhibited by the graph, we tend to see spikes within a small interval period. The overall trend has an increasing slope because of the fact that λ is itself increasing so by definition more nodes need to be removed in order to decrease the connectivity of the graph G.

VII. Additional test cases for the program

An additional test case has been included with the project to test the effectiveness of the algorithm. This test case has the following topology.



As we can see from the diagram, the lowest degree of the any node in the graph is 4, while the λ value of the graph is 1. This behavior is caught by the program. The inputs for such a graph are also contained in submission

VIII. Conclusion

The above approach gives us a very fast program for finding the minimum cut of a graph. As we can observe from the plots, minimum cut directly correlated with the graph density in the sense that edge-connectivity trends in a non-decreasing fashion against the graph density.

References

Basic format for books:

[1] Graph connectivity and it's augmentation: Application of MA orderings – Nagamochi H., Ibaraki T.

[2] GitHub Link for project 1 source code,data and plots https://github.com/ssdeep/aatn-project1

[3] GitHub Link for project 2 source code,data and plots https://github.com/ssdeep/aatn-project2

[4] R Diagram package for visualizing simple graphs http://cran.r-project.org/web/packages/diagram/vignettes/diagram.pdf