Solving complex reliability networks using conditional probability.

Anthony Yaghi
School of Electrical and Computer Engineering
Lebanese American University
Lebanon, Byblos
Email: anthony.yaghi@lau.edu

***Abstract*—The ability to quickly solve probability systems will improve the process of reliability study, one of the methods that help do this is the conditional probability approach. This project proposes a way to implement this method as an algorithm on a computer. The implementation is tested with a simple and a slightly more complex example to prove that it works for small to medium sized systems, in addition further improvements are proposed.**

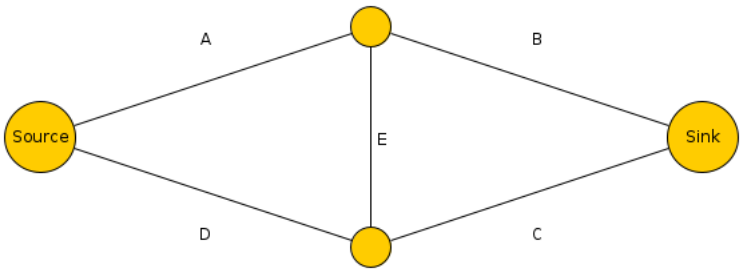
I. Introduction

The field of reliability study started to take shape during the second world war. A lot of new military equipment were introduced and the military needed a measurement of how likely these costly equipment were to fails in order to mitigate the risk of a malfunction during the war. During that time, the main focus of reliability study was the vacuum tube which constituted an integral part of many of these equipment, like radar systems. Today, reliability study is a large field with both military and commercial applications. A common way to represent a system is using a graph or a network where each edge represents a component, by using this representation, one can visualize the system which makes it easier to apply different evaluation methods. One such evaluation method is the conditional probability approach, it consists of simplifying the system by reducing it into a set of series-parallel systems and re-combining their solution. The conditional probability approach is a very powerful tool that can be used to solve any complex system elegantly; in this project we will explore the possibility of implementing this method on a computer. The objective is to have a working implementation that is capable of taking a representation of a system and finding the best way to reduce it using the conditional probability approach. In addition, the implementation could be used as an educational tool to show visually how a system gets divided into 2 new sub-systems for a given component. We will start by explaining the conditional probability method and how to transform it into an algorithm. In the second part of the report, we will present the actual implementation and the technical challenges faced. Finally, we will see how the implementation will perform on a set of different problems/networks.

II. The Conditional Probability Approach

A. Method

The conditional probability approach consists of gradually breaking a complex system into smaller and simpler series-parallel sub-systems. The reliability of the original system can then be obtained by combining the solutions of the sub-systems using conditional probability, hence the name. Take for example the following network of components.



It can be reduced into 2 sub-systems each of which is a series-parallel system.

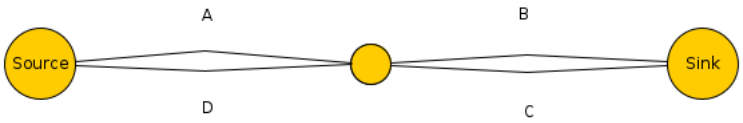


Fig. 1. E is good

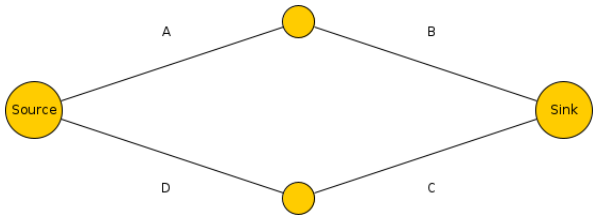


Fig. 2. E is bad

Figure 1 shows the resulting network when we take component E to be always good. Figure 2 on the other hand, shows the network when we take component E to be bad. The reliability of the original system can be then written as:
$R_{sys} = R_E \times R_{sub-sys1} + Q_E \times R_{sub-sys2}$
Where $Q_E = 1 - R_E$, $sub - sys1$: when E is good and $sub - sys2$: when E is bad. All that is left to do is solve $R_{sub-sys1}$ and $R_{sub-sys2}$ which is a trivial task since both are series-parallel. For more complex systems, breaking it down once will not necessarily lead immediately to 2 series-parallel sub-systems. In this case, the process should be repeated on the resulting sub-systems until we end up with only series-parallel networks that can be easily solved.

B. Algorithm

In this section I will propose an algorithm to find all the possible ways a system can be reduced in, then search for the smallest/fastest combination of components that will reduce it to a set of series-parallel systems. The method described in the previous section can be implemented by recursively building a tree, where each node is a sub-system. A graph represents a system where each component is an edge, so in what follows I will be using the word graph instead of system.
The idea is to start with the main graph as the root and expand the tree with 2 branches for each component in the graph. One branch will lead to a node where the component is good and the other to a node where it is bad. Each child node is then expanded in turn. The algorithm stops expanding a branch when a series-parallel graph is reached.

Once the tree is built it is now possible to find the smallest combination of edges/components needed to reduce the system into a set of series-parallel graphs. A depth first search will go over the previously built tree and give a weight to every branch equal to the depth of the sub-tree connected to it. The next step will be to simply cut all the branches at the first level except the pair with the smallest combined weight. Notice that in algorithm 1 each edge in the graph will lead to the creation of 2 nodes and thus 2 branches. It is crucial in this step to consider these pairs of branches together otherwise the logic will be false. This means that there is a single combined score for each pair and if one branch is removed the other one should be removed as well.

Input: node containing a reliability graph
Output: Conditional probability tree

Function `BuildTree(`$root$`)`**:**
  if $root$ *is series-parallel* **then**
    return;
  else
    $graph \leftarrow root.data$;
    for $edge \in graph$ **do**
      $goodNode \leftarrow$ **new** $node$;
      $badNode \leftarrow$ **new** $node$;
      $goodGraph \leftarrow$ **reduce**$(graph, edge,$ **good**$)$;
      $badGraph \leftarrow$ **reduce**$(graph, edge,$ **bad**$)$;
      $goodNode.data \leftarrow goodGraph$;
      $badNode.data \leftarrow badGraph$;
      `BuildTree(`$goodNode$`)`;
      `BuildTree(`$badNode$`)`;
      $root.addChild(goodNode)$;
      $root.addChidl(badNode)$;
      return
    end
  end

Algorithm 1: Algorithm for building the conditional probability tree

Input: root node of a tree

Function `FindWeights(`$root$`)`**:**
  if $root.branches = \emptyset$ **then**
    return 0;
  else
    for $branch \in root.branches$ **do**
      $w =$ `FindWeights(`*branch.node*`)`;
      $branch.weight = w + 1$;
    end
  end

Algorithm 2: Algorithm for finding branches weights

III. Implementation and Challenges

A. Programming language and packages

In order to implement the algorithms proposed above we should be able to easily represent graphs and do operations on the nodes and edges, in the code. For this reason NetworkX [1], a python package, was used. NetworkX allows the creation of many type of graphs, for this implementation a MultiGraph is used. It is a type of graph with undirected edges and which allows more than 1 edge between 2 nodes. This is an important property otherwise it will be impossible to have parallel edges/components. A graph is a list of nodes connected by a list of edges, by using NetworkX a single method can add a whole list of nodes or edges. In addition to be able to create graphs, this packages offers a range of algorithms from graph theory that can be directly used.
Drawing graphs was done using Matplotlib, 'Matplotlib is a Python 2D plotting library which produces publication

quality figures in a variety of hardcopy formats and interactive environments across platforms.' [2]

B. Series-parallel graphs

One of the main challenges of this project was to find a good way to tell if a given graph is in a series-parallel configuration. There are many definition for a SP-graph in the literature, but I ended up adopting the definition introduced by Duffin, R.J. [3]. First the graph should have 2 distinguished nodes, the source and the sink, labeled s and t. Such a graph is a SP-graph if it can be reduces to a K_2 using the following operations:

1) Replace 2 edges connected by a node with degree 2, other than s or t, by 1 edge
2) Replace 2 edges that are parallel by 1 edge

A K_n is a graph having n nodes where each pair of nodes is connected by a unique edge. In our case, this means that K_2 is a graph with a single edge connecting the source and sink nodes.

C. Detecting shorted components

Assuming a component to be good will reduced the graph and can lead to having one or multiple components shorted. Since this operation is carried out multiple times when building the conditional probability tree, it was important to find a way to detect and eliminate these shorted components. In graph theory a loop is a list of edges that connects a node back to itself. Based on this definition, we can detect a series of shorted components/edges by finding a loop that contains only nodes of degree 2, except the origin node which can have any degree. Once such a loop is found, the algorithm will remove all the edges that forms the loop, then check for any nodes with degree 0 and remove them as well. This step should be performed each time a graph is reduced by considering a node to be always good.

IV. Results

A. Usage

First we need to create a file containing a list of all the nodes and edges in our system or graph. The easiest way to create this file is by using a graph/diagram program, which allows us to visually construct the graph. One such program is **yEd**, and I will be using it throughout this section to create graphs and export them into 'tgf' format, which is compatible with the implementation done for this project. Once the graph file is ready we either:

1) Call the python program and give it the file's path.
2) Call the python program and give it the file's path and an edge name.

Option 1 will output the conditional probability tree, while option 2 will output the 2 sub-systems that results from reducing the system based on the specified edge.

B. Example 1: simple case

This first example is using a simple SP-graph with 3 components.

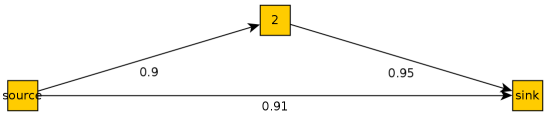


Fig. 3. Simple SP-graph

python main.py simple.tgf

Fig. 4. Simple SP-graph tree

The tree has only 1 node which is the root that represents the original graph. This means that the graph doesn't need any reduction to be solved.

python main.py simple.tgf source-sink-0.91

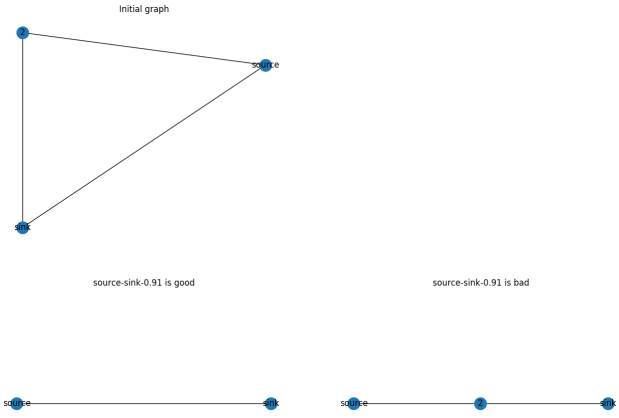


Fig. 5. Simple SP-graph condition on edge

Figure 5 shows the output of the program when the edge parameter is added. The left hand sub-system show the reduction that occurs when the edge is good, it also shows how the **sink-2** and **2-source** edges were removed because they are shorted.

C. Example 2: advanced case

In this example a slightly more complex graph will be used.

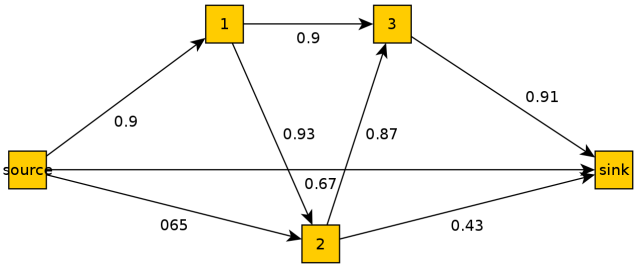


Fig. 6. Complex graph

python main.py advanced.tgf

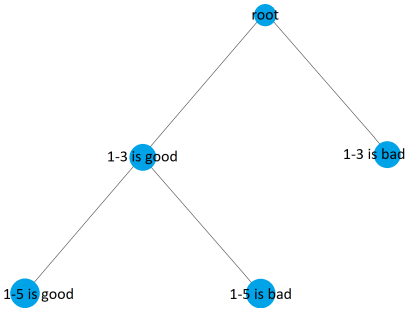


Fig. 7. Advanced conditional tree

Figure 7 shows that the system can be reduces to a set of SP-graph in 2 step, first pick component **1-3** then component **1-5**.

V. Conclusion

The conditional probability method is a very powerful tool in the field of reliability engineering because it allows to solve any system no matter its complexity. In this project we proposed an implementation of this method that can be used to solve and also visualize the solution. The two main parts of the proposed algorithm will first build a tree of all possible combinations and then search for the shortest path to a solution. The examples presented in the last part proves that the proposed implementation works as intended. It is capable of finding the best way to reduce a system and also to show how a system gets divided into 2 sub-systems based on the conditional probability of a given component. Moreover, this last part can serve as an educational tool to help visualize the conditional probability method in an dynamic way. However, further research is needed to determine the performance of this implementation specially for bigger systems. Visualizing the output may not be feasible for such systems, but finding the equivalent reliability quickly can prove to be useful. Furthermore, pruning algorithms could potentially improve the time needed to build the tree, while metaheuristics could be used to improve the search time.

References

[1] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring network structure, dynamics, and function using networkx," in *Proceedings of the 7th Python in Science Conference*, G. Varoquaux, T. Vaught, and J. Millman, Eds., Pasadena, CA USA, 2008, pp. 11 – 15.

[2] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.

[3] R. J. Duffin, "Topology of series-parallel networks." *J. Math. Anal. Appl.*, vol. 10, pp. 303–318, 1965.