Documentation for Key Dissemination and Secret Sharing

Table of Contents

[Introduction 3](#_Toc190253885)

[General Functions 3](#_Toc190253886)

[del\_cut\_edges() 3](#_Toc190253887)

[is\_cut\_vertex() 3](#_Toc190253888)

[get\_connect\_sets() 4](#_Toc190253889)

[get\_intersection\_set\_H\_edges() 5](#_Toc190253890)

[alt\_path\_exists() 6](#_Toc190253891)

[intersection() 6](#_Toc190253892)

[class ShareSecret 6](#_Toc190253893)

[\_\_init\_\_() 6](#_Toc190253894)

[get\_cut\_vertices() 6](#_Toc190253895)

[get\_alternating\_path() 6](#_Toc190253896)

[get\_source\_to\_target\_path() 6](#_Toc190253897)

[Example 7](#_Toc190253898)

[class ShareKey 7](#_Toc190253899)

[\_\_init\_\_() 7](#_Toc190253900)

[\_u\_does\_not\_learn() 7](#_Toc190253901)

[does\_scheme\_exists() 7](#_Toc190253902)

[Example 8](#_Toc190253903)

[References 8](#_Toc190253904)

# Introduction

Assumes familiarity with some terms and definitions from [1].

# General Functions

This section contains a description of some more general or basic functions that a user may want when analyzing a graph for either secret sharing or key sharing. In the GitHub project, these functions are implemented in

Key-Dissemination-Simulation/network\_algs/base\_funcs.py.

## del\_cut\_edges()

|  |  |
| --- | --- |
| Returns | G\_tmp: ig.Graph |
| Parameters | Graph: ig.Graph, cut\_vertex: int |
| Description | For a vertex of a graph this function determines the incoming and outgoing edges of the vertex and removes them. This has the effect of disconnecting from the network without affecting the labeling/resizing of the graph. A copy of the graph with these edges removed is returned.  A diagram of a network  AI-generated content may be incorrect.  **Figure 1:** The edges of vertex are removed. |
| Purpose | This function is used so that we may analyze the graph with the cut-vertex disconnected. This is necessary to find the connect sets (described later) which will give us a method for finding the alternating path. |

## is\_cut\_vertex()

|  |  |
| --- | --- |
| Returns | bool |
| Parameters | Graph: ig.Graph, source: int, target: int, u: int |
| Description | Given a graph and the vertices (where is a source, is a target/destination, and is any vertex in ), this function returns True if is a cut-vertex and False otherwise. A cut-vertex is a vertex whose removal disconnects . In other words, if   1. a path initially exists 2. and after we disconnect no path exists,   then is a cut vertex. This function is fundamentally implemented by just checking these two conditions. |
| Purpose | When looking to share a key we would like to check if a vertex in the network will learn the message (part of the key) being sent from a source to a target. It will learn this message if is a cut-vertex and there does not exist an alternating path. Therefore, this is a necessary primitive for checking if a scheme for securely sharing a key exists. |

## get\_connect\_sets()

|  |  |
| --- | --- |
| Returns | connectivity\_sets: list[list] |
| Parameters | Graph: ig.Graph |
| Description | Gets the connectivity set for each vertex in a graph . A connectivity set for a vertex contains all vertices such that a path exists. The connectivity set for a vertex also contains itself (this is more for the convenience of implementation). Although the connect sets for all vertices in are returned, we are primarily interested in the connect sets for the source, target, and collider.  The algorithm works by passing on the connect set of a vertex to a topologically succeeding vertex if they are adjacent. After this we must also remove all duplicate vertices from each list. |
| Purpose | To find an alternating path, we need to find a specific structure in the network. Specifically, we need to find a vertex that can generate a random bit (the red vertices in Figure 2) and share it with two vertices that are in-coming to the cut-vertex. By definition of the connectivity sets, we know that if the sets for two vertices that are incoming to the cut-vertex intersect, then the intersection will be vertices that can share a bit with both of these vertices. Therefore, the intersection of connectivity sets will allow us to determine the alternating path.    A screen shot of a game  AI-generated content may be incorrect.  **Figure 2:** The ellipses represent the connectivity sets for the source , target , and colliders (the green vertices that are in-coming to the cut-vertex ). The black, zig-zagging edges make up the alternating path. |

## get\_intersection\_set\_H\_edges()

|  |  |
| --- | --- |
| Returns | tuple[intersection\_sets: list[list], edges\_H: list] |
| Parameters | Graph: ig.Graph, connectivity\_sets: list, in\_cut\_target\_source: list |
| Description | **Intersection sets:** For a given vertex and its connectivity set, we will check if another vertex and its connectivity set intersect. If the intersection of these connectivity sets is not empty, then the vertex and the intersection set will be added to the list entry associated with vertex . In other words, the entry of vertex in the intersection\_sets list will look like  We now have a list of all connectivity sets and intersect and with which connectivity set they intersect with.  **Edges for Graph :** As we are getting these intersection sets, we will also create a list of edges for the connectivity sets associated with the vertices: source, in-coming to cut, and target. An undirected edge will be formed if the intersection between two connectivity sets is not empty. This set of edges will be used to generate graph , which is used to analyze alternating paths. |
| Purpose | Both the intersection sets and edges for graph are used to determine an alternating path (if it exists). The edges for graph are used to find which group of connectivity sets should be used. To find an alternating path we are looking for a path from the source connectivity set to the target connectivity set in graph . Once we know this path, we translate it to an alternating path in our main graph using the intersection sets.  A diagram of graphing lines and points  AI-generated content may be incorrect.  **Figure 3:** The path found in graph allows us to identify an alternating path. We then convert back to graph to get the actual alternating path (in bold) which protects the cut-vertex 1. |

## alt\_path\_exists()

|  |  |
| --- | --- |
| Returns | bool |
| Parameters | Graph: ig.Graph, source: int, target: int, cut\_vertex: int |
| Description | This function checks if an alternating path exists for a given graph, source, target, and cut-vertex. It is implemented by calling get\_intersection\_set\_H\_edges() and forming the graph with the returned edges. If a path from in graph exists, then we return True. Otherwise, False. |
| Purpose | When looking to share a key securely we want to know if some vertex in the network learns the message from some source. If we have a cut-vertex we need to check if it is protected (i.e., there is an alternating path) to determine if it will learn of the message from the source. Knowing this will help us to determine if a scheme to securely communicate a key exists for a given network. |

## intersection()

|  |  |
| --- | --- |
| Returns | Intersection: list |
| Parameters | l1: list, l2: list |
| Description | Returns the intersection of two lists. |

# class ShareSecret

\*\*include file path/location in project\*\*

## \_\_init\_\_()

## get\_cut\_vertices()

## get\_alternating\_path()

## get\_source\_to\_target\_path()

## Example

# class ShareKey

## \_\_init\_\_()

## \_u\_does\_not\_learn()

## does\_scheme\_exists()

## Example

# References

1. Michael Langberg, Michelle Efros, "Characterizing positive-rate key-cast (and multicast network coding) with eavesdropping nodes", arXiv:2407.01703v1 [cs.IT], 2024.