

Documentation for Secret Sharing and Key Scheme Existence

Table of Contents

Introduction.....	3
General Functions.....	3
del_cut_edges().....	3
is_cut_vertex()	4
get_connect_sets()	4
get_intersection_set_H_edges()	5
alt_path_exists()	6
intersection().....	6
class ShareSecret	7
__init__().....	7
get_cut_vertices()	7
get_alternating_path().....	7
get_source_to_target_path().....	8
Example.....	8
class ShareKey.....	8
__init__().....	8
_u_does_not_learn()	9
does_scheme_exists()	9
Example.....	10
Appendix A: Some Notation and Definitions	10
Appendix B: Description of Project Documents	11
References	12

Introduction

This document is intended as a general guide for using the “Key-Dissemination-Simulation” project for secret sharing and key sharing. The document assumes some familiarity with the terms and concepts defined in [1]. To implement some of the concepts from [1], new objects like connectivity sets have been defined.

Appendix A may be useful for some basic terminology and definitions that are used throughout this document. If you would like to see a brief overview of the contents contained in the GitHub project, see Appendix B.

The project GitHub, which includes all code, and this document can be found at:

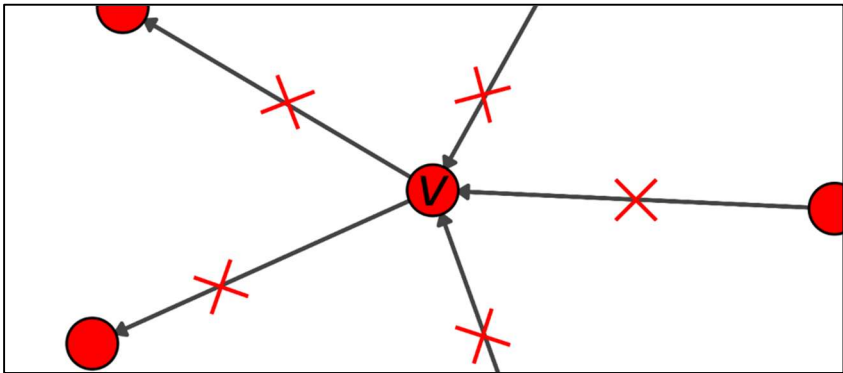
<https://github.com/timjtorres/Key-Dissemination-Simulation>

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	<p>For a vertex v of a graph G, this function determines the incoming and outgoing edges of the vertex and removes them. This has the effect of disconnecting v from the network without affecting the labeling/resizing of the graph. A copy of the graph with these edges removed is returned.</p>  <p>Figure 1: The edges of vertex v are removed.</p>
Purpose	<p>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.</p>

is_cut_vertex()

Returns	bool
Parameters	Graph: ig.Graph, source: int, target: int, u: int
Description	<p>Given a graph $G = (V, E)$ and the vertices $s, t, u \in V$ (where s is a source, t is a target/destination, and u is any vertex in V), this function returns True if u is a cut-vertex and False otherwise. A cut-vertex is a vertex whose removal disconnects $s \rightarrow t$. In other words, if</p> <ol style="list-style-type: none">a path $s \rightarrow t$ initially existsand after we disconnect u no path $s \rightarrow t$ exists, <p>then u is a cut vertex. This function is fundamentally implemented by just checking these two conditions.</p>
Purpose	<p>When looking to share a key we would like to check if a vertex u 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 u 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.</p>

get_connect_sets()

Returns	connectivity_sets: list[list]
Parameters	Graph: ig.Graph
Description	<p>Gets the connect set for each vertex in a graph $G = (V, E)$. A connect set C_v for a vertex $v \in V$ contains all vertices $u \in V$ such that a path $u \rightarrow v$ exists, i.e.,</p> $\text{Given } v: C_v = \{u \in V \mid \exists \text{ path}(u, v)\}$ <p>Note that C_v contains the vertex v (this is more for the convenience of implementation). Although the connect sets for all vertices in V are returned, we are primarily interested in the connect sets for the source, target, and collider.</p> <p>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.</p>
Purpose	<p>To find an alternating path, we need to find a type of zig zagging pattern. Specifically, we want to find a vertex that can generate a random bit (the red vertices in Figure 2) and share it with two vertices (this are usually the source, incoming to cut, and target). From the definition of C, if $C_{v,u} = C_v \cap C_u \neq \emptyset$ then $x \in C_{v,u}$ can reach both v and u to share a random bit.</p>

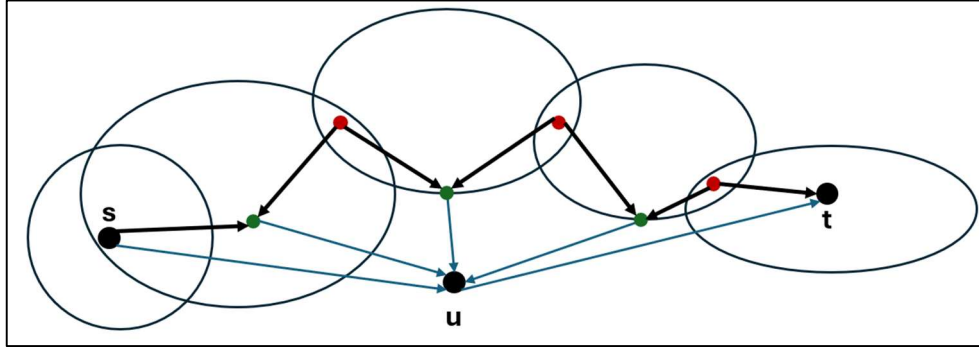
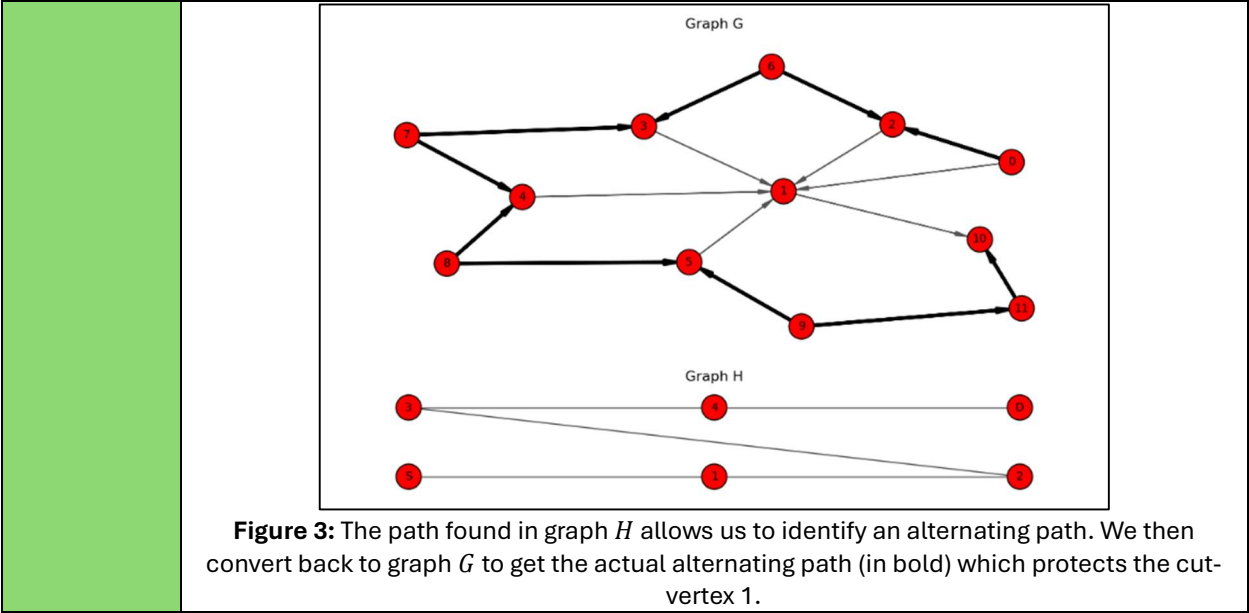


Figure 2: The ellipses represent the connectivity sets for the source s , target t , and colliders (the green vertices that are in-coming to the cut-vertex u). The black, zig-zagging edges make up the alternating path.

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	<p>Intersection sets: For a given vertex v and its connectivity set C_v, we will check if another vertex $u \neq v$ and its connectivity set C_u intersect, i.e., $C_{v,u} = C_v \cap C_u \neq \emptyset$. The vertex u and $C_{v,u}$ will be appended to the intersection set associated with vertex v, i.e.,</p> $\text{Given } v, \text{ intersection_sets}[v] = (u_1, C_{v,u_1}), \dots, (u_i, C_{v,u_i})$ <p>We now have a list containing the intersection of all connectivity sets with each other.</p> <p>Edges for Graph H: 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 H, which can be used to find an alternating path.</p>
Purpose	<p>Both the intersection sets and edges for graph H are used to determine an alternating path (if it exists). The edges for graph H are used to find which group of connectivity sets should be used.</p> <p>Note that in the implementation we are looking for a path from C_{source} to C_{target} in H. We are not considering starting from another connectivity set that may also contain the source. This means that we may miss some structures for alternating paths. However, if these other structures exists, so will the one that we find. In other words, it does not affect our ability to find an alternating path if it exists, but it does limit the variety of alternating paths that we can find.</p>



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 <code>get_intersection_set_H_edges()</code> and forming the graph H with the returned edges. If a path from $s_{conn} \rightarrow t_{conn}$ in graph H 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

This class is intended to be a high-level interface determining if a secret message can be shared between a single source and target. This algorithm was implemented to work with at most one cut-vertex. The user may wish to make changes to extend this algorithm for any number of cut-vertices.

In the project, this class is implemented in

`Key-Dissemination-Simulation/network_algs/ShareKey.py`

`__init__()`

Returns	-
Parameters	self, Graph: <code>ig.Graph</code> , source: int, target: int
Description	Class initializer. Takes a graph, source, and target, and initializes them as members (along with some other graph quantities for convenience).

`get_cut_vertices()`

Returns	cut_vertices: list
Parameters	self
Description	This method finds all the cut vertices between the initialized source and target over the initialized graph.
Purpose	Before we find an alternating path, we need to find the cut/unprotected vertex. We're assuming that the cut vertex (if there is one) is not known ahead of time. Therefore, we will use this method to find cut vertices. Even though this finds all cut vertices, the secret sharing algorithm is only implemented for at most one cut-vertex.

`get_alternating_path()`

Returns	P_alt: list
Parameters	self
Description	This method is like the <code>alt_path_exists()</code> except it also finds an alternating path for the input graph. This is done by finding a path between the source and target connectivity sets in the H graph (generated using the edges found in <code>get_intersection_set_H_edges()</code>) and translating back to our original graph G . The details of this translation are omitted here as it is complicated to describe and not particularly interesting. See the loop portion of the implementation for specific details on finding an alternating path in G from the path found in H . Note that we are just finding an alternating path. More paths and structures may exist, but we are not concerned with them for this method. The important thing is that if an alternating path does exist, we can find at least one in G .
Purpose	This is exactly what we need if we want to share a secret and there is a cut-vertex.

get_source_to_target_path()

Returns	source_to_target: list
Parameters	self
Description	Finds and returns the shortest path for the initialized source and target over the initialized graph.
Purpose	We need to know the path from source to target if we want to share a secret (or any message).

Example

To help give a feel for how this class can be used, an example script is provided in

Key-Dissemination-Simulation/ `example_share_secret.py`

The script implements finding the alternating path for the below network

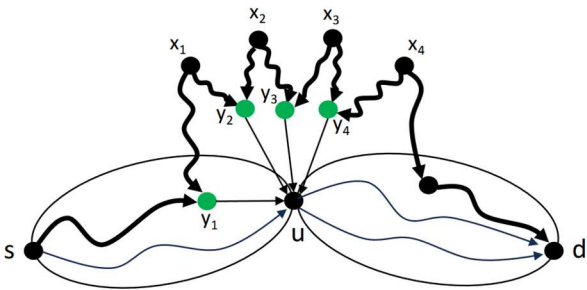


Figure 4: Network example from [1] for alternating path.

class ShareKey

This section provides a description of the methods for the `ShareKey` class. Additionally, it includes an example of using the class to determine if a secret can be shared for two different networks.

__init__()

Returns	-
Parameters	self, Graph: ig.Graph, targets: list
Description	Class initializer. Takes a graph and list of targets, and initializes them as members

_u_does_not_learn()

Returns	: int
Parameters	self, source: int, target: int, u: int
Description	<p>Determines if a vertex u in the graph G learns what was transmitted by a from source to a target. The function returns 0 if u does not learn the message and returns 1 otherwise. To determine what should be returned the following conditions are checked</p> <pre>if source = u then return 0 else if u is not a cut vertex then return 1 else if an alternating path exists then return 1 else return 0 end if</pre>
Purpose	To determine if secure key sharing exists we are checking to see if any vertex u learns the message from every source participating in key sharing. This method allows us to perform this analysis by calling it for every potential source (a vertex that can reach all targets) and every potential intermediate vertex (all vertices in the graph excluding the targets).

does_scheme_exists()

Returns	scheme_exists: bool
Parameters	self
Description	<p>For a graph $G = (V, E)$, determines if a scheme to securely share a key with a set of targets $T \subseteq V$ exists.</p> <p>To do this, we first create two subsets of the vertices V. The first is $V_s = \{v \in V \forall t \in T \exists \text{path}(v, t)\}$. This set consists of all the potential sources. A potential source is exactly a vertex that can reach all targets.</p> <p>This second subset is simply $V_u = V \setminus T$. In other words, it is all vertices excluding the set of targets. We create this subset since we want all targets to learn all the messages from the sources.</p> <p>We now call <u>_u_does_not_learn()</u> for all potential sources and for all potential intermediate vertices. This leaves us with a $V_s \times V_u$ matrix of 1's (u learns) and 0's (u does not learn). Then we can check if a single potential intermediate vertex learns all messages from all potential sources. In other words, we can check if there exists a column of all 0's in the $V_s \times V_u$ matrix. If one does exist, then a scheme for securely sharing a key does not exist and False is returned. If an all 0's column does not exist, then a secure, key-sharing scheme exists and True is returned.</p>
Purpose	Allows us to answer the question:

	Does a secure, key-sharing scheme exist for a specific DAG network and set of targets?
--	--

Example

An example of how to use the `ShareKey` class with two different networks to determine if a scheme for secure key sharing exists is provided in

`Key-Dissemination-Simulation/example_key_scheme.py`.

Appendix A: Some Notation and Definitions

Symbol	Description	Definition
G	A graph or network $G = (V, E)$. In general, this is the original network that we are analyzing or using.	$G = (V, E)$ Where V is a set of vertices and E is a set of edges. An edge is a tuple describing a connection between two vertices. For G , this is a directed connection, i.e., $(v, u) \neq (u, v) \forall v \neq u \in V$
s	The source/transmitter vertex. This vertex holds or generates the information we would like to communicate	The source
t	The target/terminal/destination/receiver vertex. This is the vertex that we are trying to send information to.	The target
T	This represents a set or collection of targets.	A set of targets
C_v	The connectivity set or connect set for a vertex v .	Given v : $C_v = \{u \in V \exists path(u, v)\}$
$C_{v,u}$	The intersection between the connectivity set for v and the connectivity set for u . In other words, this is the set of vertices that can reach both v and u .	Given C_v and C_u $C_{v,u} = C_v \cap C_u$
H	The meta graph H . The nodes are the connectivity sets for $V_H = \{s, t, \text{vertices incoming to cut vertex}\}$. An undirected edge (v, u) is formed if for $v, u \in V_H$, $C_{v,u} \neq \emptyset$	$H = (V_H, E_H)$
V_s	The set of vertices in G which can reach all $t \in T$. In other words, this is the set of potential vertices in G for a set of targets in G .	$V_s = \{v \in V \forall t \in T \exists path(v, t)\}$
V_u	The set of all vertices in G excluding the targets. These are all the vertices that we don't want to learn some information. Only the targets should learn.	$V_u = V \setminus T$

Appendix B: Description of Project Documents

This Appendix aims to add a description to all the folders in files in the GitHub project.

<ul style="list-style-type: none">• docs: Contains documentation for the project. Specifically includes documentation for using the code along with some description of the algorithms.<ul style="list-style-type: none">○ Key_Sharing_Docs.docx: This document in word format.○ Key_Sharing_Docs.pdf: This document in pdf format.
<ul style="list-style-type: none">• images: Contains images that are used in the project.
<ul style="list-style-type: none">• network_algs: Contain the project code that implements all the algorithms.<ul style="list-style-type: none">○ ShareKey.py: Python file implementing the ShareKey class.○ ShareSecret.py: Python file implementing the ShareKey class.○ __init__.py: Initializes the network_algs package. Essentially says how/what to import with the package○ alt_path_primitive.py: Deprecated file with many functions that were initially created. Left for posterity but does not need to be used. This was the file I was working out of until I reorganized the project. May still be helpful for someone trying to implement more features or experimenting.○ base_funcs.py: Python file with some basic functions that can be used by the two classes or directly by the user.
<ul style="list-style-type: none">• NetworkToolsBrainstorm.md: Markdown file discussing some alternative packages for analyzing networks. It also includes links to databases of some real-world networks.• README.md: Markdown file with a high-level description of the project and goals.• example_key_scheme.py: Python script implementing an example for using the ShareKey class. Determines if a key scheme exists for two different networks.• example_secret_share.py: Python script implementing an example for using the Share secret class. Finds an alternating path for a simple network.• requirements.txt: Text file listing the package requirements for the project. This file can be passed to pip to easily install all necessary packages.• test_adjacency_matrix.py: Another example of finding an alternating path on a larger network built from an adjacency matrix.

References

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