Proof of edge-critical strongly-connected digraph generator algorithm

Index

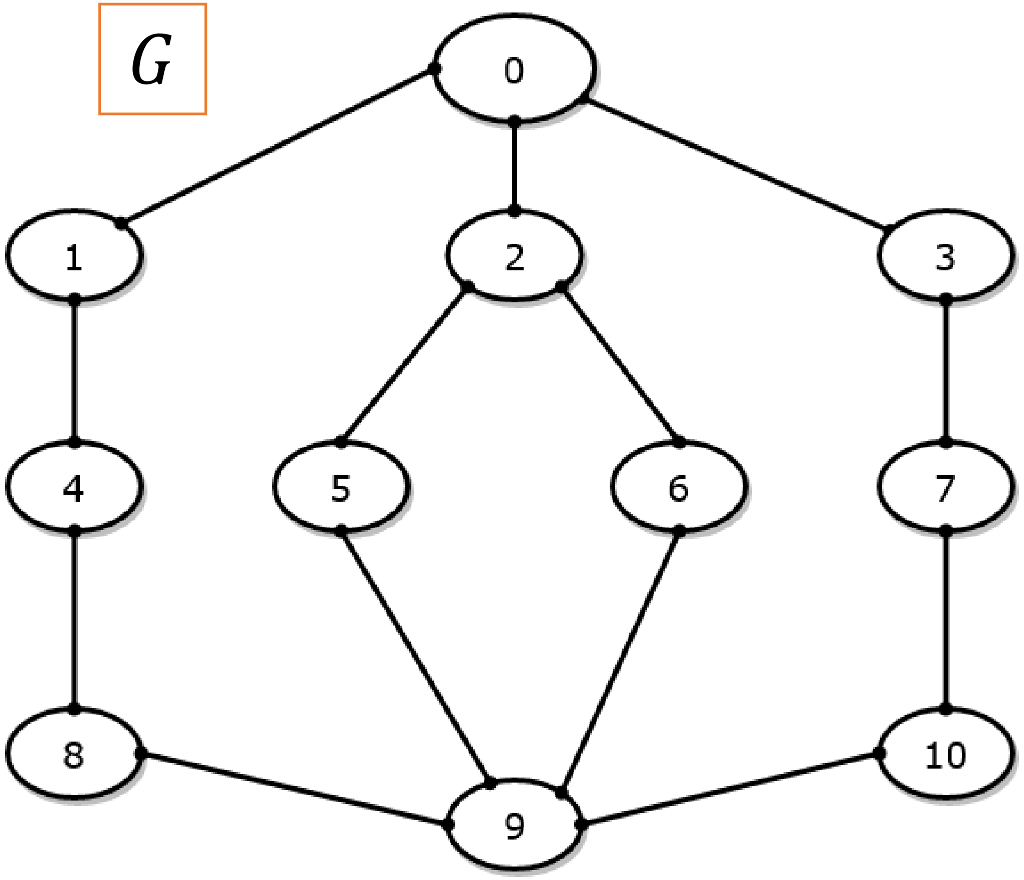
- Definition
    1. Graph
    2. edge-critical strongly-connected digraph
    3. Isomorphic Graphs
    - Summary
- Problem Statement
    - Objective
    - Existing tools
- Prime digraph generation
    - Example and Motivation
    - Algorithm I – Reduce to primes
    - Algorithm II - Find all prime digraphs
    - Results

Index

Graph[1]

A graph G is a triple consisting of a **vertex set** $\mathbf{V}(\boldsymbol{G})$, an **edge set** $\mathbf{E}(\boldsymbol{G})$, and a relation that associates with each edge two vertices (not necessarily distinct) called its endpoints.

- The terms **vertex** and **node** may be used interchangeably.

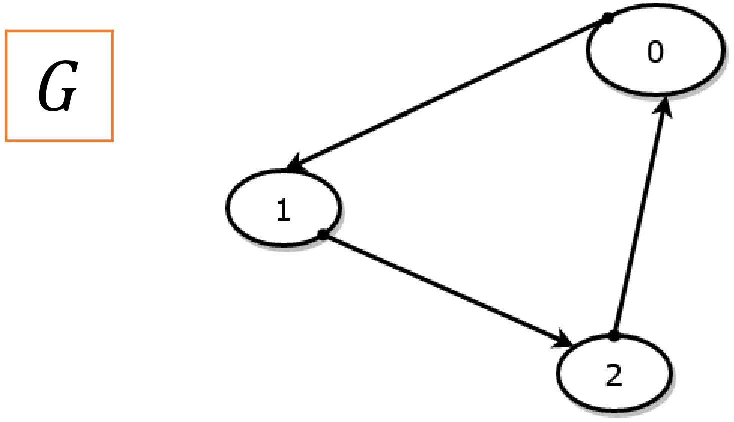


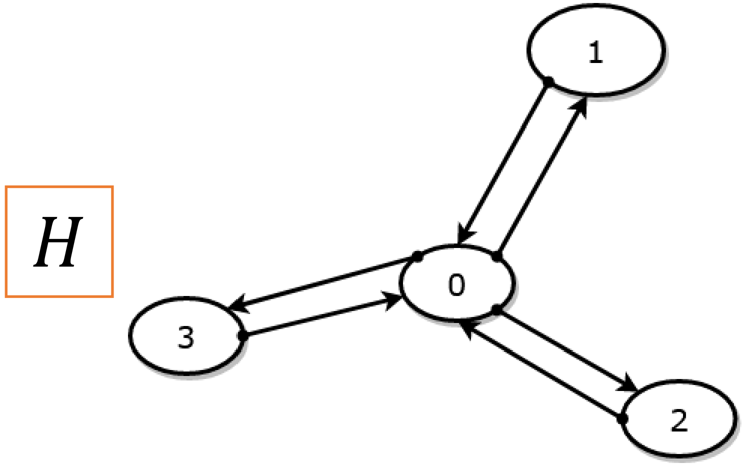
$Here\ V(G) = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$
$And\ E(G) = \{(0,1), (0,2), (0,3), (1,4), (2,5), (2,6), (3,7), (4,8), (5,9), (6,9), (7,10), (8,9), (9,10)\}$

edge-critical strongly-connected digraph

A strongly-connected digraph that ceases to be strongly-connected if **any of its edge is removed** is called edge-critical strongly-connected digraph.

Note: In the codebase [5], an edge-critical strongly-connected digraph is called **prime digraph.** We may use the terms interchangeably.





G and H are prime digraphs.

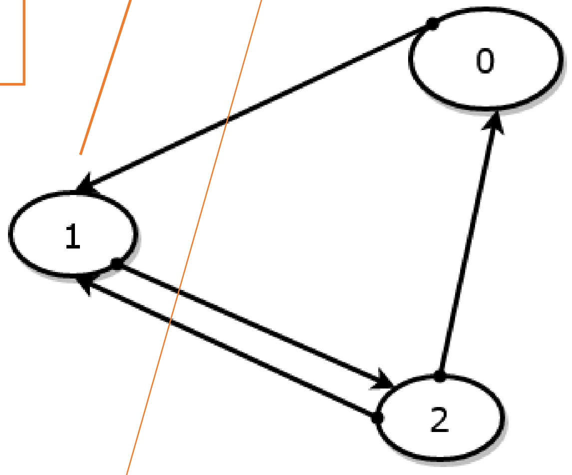
Note: All edge-critical graphs are 1-edge-connected graphs but not necessarily vice versa.

Isomorphic Graphs [6]

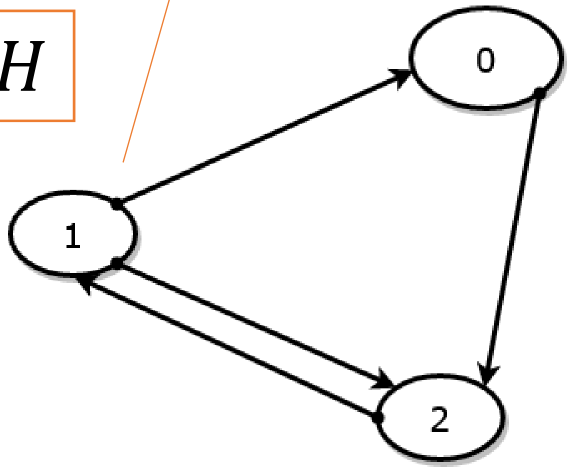
Two graphs G and H are said to be isomorphic if there exists a function f from nodes of G to nodes of H, such that (u, v) is an edge of G if and only if $(f(u), f(v))$ is an edge of H.

Note the difference in tail and head of edge

G



H



G and H are isomorphic as the following f function suffices:

$$f(0) = 0; f(1) = 2; f(2) = 1$$

Definitions-Summary

Term	Definition
Graph	A graph G is a triple consisting of a **vertex set** $\mathbf{V}(\boldsymbol{G})$, an **edge set** $\mathbf{E}(\boldsymbol{G})$, and a relation that associates with each edge two vertices (not necessarily distinct) called its endpoints.
Directed Graph	A directed graph or digraph G is a triple consisting of a vertex set $\mathrm{V}(G)$, an edge set $\mathrm{E}(G)$, and a function assigning each edge an **ordered pair of vertices**.
strongly-connected Directed Graphs	A digraph is strongly connected or strong if for each ordered pair u, v of vertices, there is a path from $u\ to\ v$.
k-edge-connected digraph	A strongly-connected digraph is k-edge-connected directed graph if it remains strongly-connected whenever fewer than k edges are removed.
edge-critical strongly-connected digraph	A strongly-connected digraph that ceases to be strongly-connected if any of its edge is removed is called edge-critical strongly-connected digraph.
Isomorphic Graphs	Two graphs G and H are said to be isomorphic if there exists a function f from nodes of G to nodes of H, such that (u, v) is an edge of G if and only if $(f(u), f(v))$ is an edge of H.

Objective

- Find an algorithm to generate all prime (edge-critical strongly-connected) directed graphs of $n + 1$ nodes given a set of all prime digraphs of n nodes.

Tools at hand

Following algorithms/tools that have been studied in detail is assumed to be present at hand for achieving the mentioned objectives.

1. Isomorphism test[7]: An algorithm that tests whether the two input graphs are isomorphic to each other or not. It does so by finding all matching node candidates for a given node and then running isomorphism test on the remaining subgraph iteratively.
2. Weisfeiler Lehman Isomorphism hash[8] – A hashing algorithm on graph such that different hash for two graphs implies non-isomorphism among the graphs with very high probability. However, the converse is not true i.e. two non-isomorphic graph can have same hash.

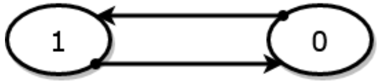
Tools at hand

3. strong-connection test[9,10]: An algorithm that tests whether the given graph is strongly-connected or not. It does so by finding the strongly connected components using Depth First Search algorithm. It is referred as **`is_strongly_connected`**`(digraph)`
4. Prime test [5]: An algorithm that tests whether given graph is edge-critical strongly-connected or not. It does so by removing an edge and checking for strong-connectedness. This is done for all edges. It is referred as **`is_prime`**`(digraph)`

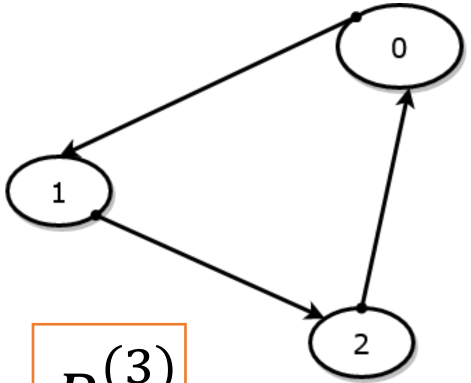
Note: Tools 1,2, and 3 are available in the networkx package. 4th tool is available on link mentioned in reference.

Further Work: Find a better algorithm to conclude primality of directed graph.

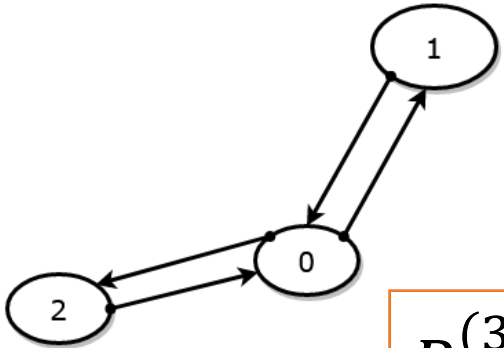
All prime digraphs of 2,3, and 4 nodes



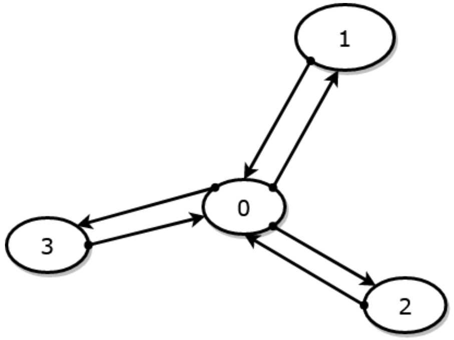
$P_1^{(2)}$



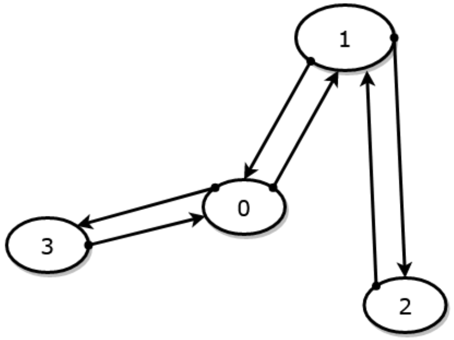
$P_1^{(3)}$



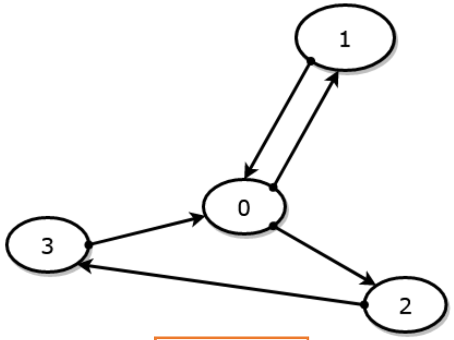
$P_2^{(3)}$



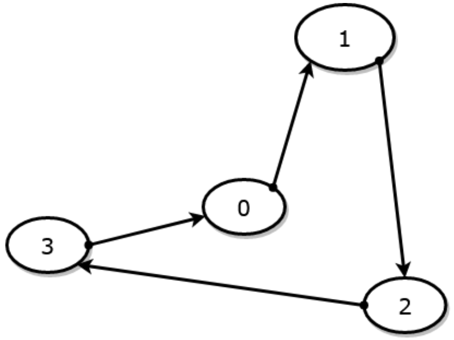
$P_1^{(4)}$



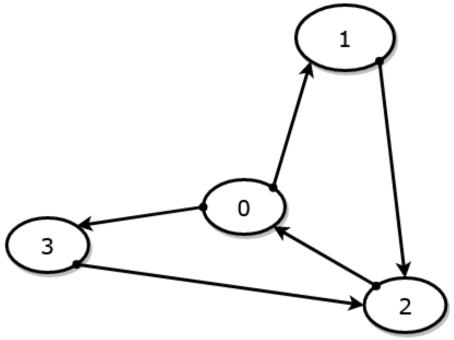
$P_2^{(4)}$



$P_3^{(4)}$



$P_4^{(4)}$



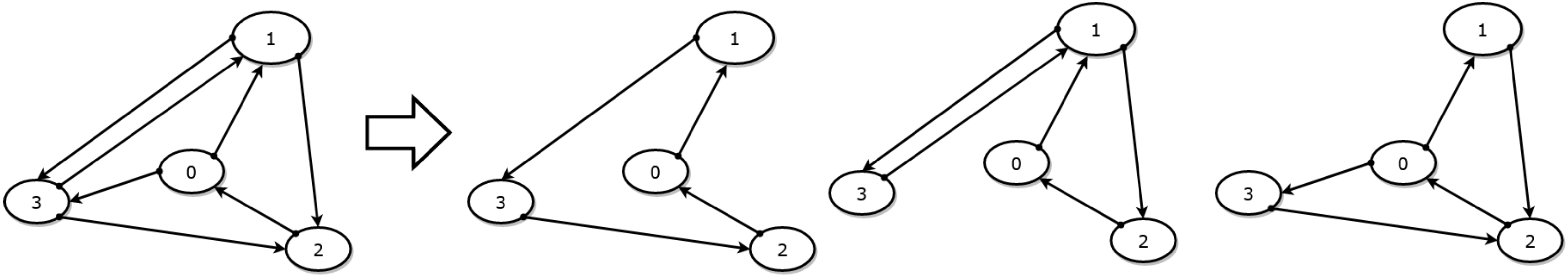
$P_5^{(4)}$

Motivation of Prime digraphs

- The prime (edge-critical strongly-connected) digraphs are basic unit of construction for any strongly-connected graph.
- If even a single prime digraph is available for a given number of nodes, then one can add edges randomly with some probability p adhering to $G(n, p)$ Erdős–Rényi model[11]. This results into a digraph that is assured to be strongly-connected.
- In order to choose a random prime digraph for reduced bias in digraph selection, it is best to have list of all prime digraphs for a given node count n.

Algorithm I – Reduce to Primes

- An algorithm “reduce_to_primes” is described below that finds all prime digraphs formed by removing one or more edges of the input digraph G.



G $\qquad$ $P_4^{(4)}$ $\qquad$ $P_3^{(4)}$ $\qquad$ $P_5^{(4)}$

- Code present at: https://github.com/oyeluckydps/directed_graph/blob/master/prime_DiGraph_Generator.py in `find_primes(DG, already_checked_cdg = None)` function.

Algorithm I – Reduce to Primes

```
def reduce_to_primes(digraph, already_checked_digraphs):
        if not is_strongly_connected(digraph):
                return ([], already_checked_digraphs)
        if isomorphic_graph_exists(already_checked_digraphs, digraph):
                return ([], already_checked_digraphs)
        else:
                already_checked_digraphs.append(digraph)
        if is_prime(digraph):
                return ([digraph], already_checked_digraphs)
        reduced_primes_on_this_digraph = []
        for edge in digraph.edge_list:
                digraph_temp = digraph.remove_edge(edge)
                primes_for_this_edge_removal, already_checked_digraphs =
                        reduce_to_primes(DG_temp, already_checked_digraphs)
                reduced_primes_on_this_digraph.add_to_list(primes_for_this_edge_removal)
        return (reduced_primes_on_this_digraph, already_checked_digraphs)
```

List of already checked digraphs

Return empty list if digraph is not strongly connected.

True if digraph (or its isomorph) is already checked, so present in list

Returns True if digraph is prime.

Algorithm I – Explanation

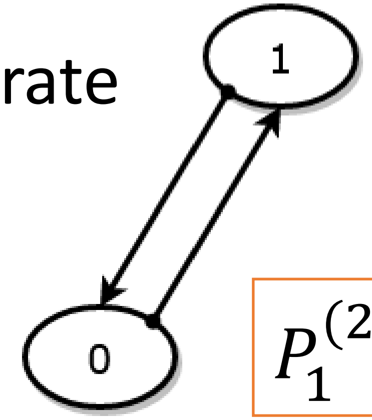
- If the digraph is not strongly-connected or it has already been checked, an empty list is returned.
- If the digraph is prime, return the digraph in a list.
- The algorithm used Depth First Search to reduce input digraph into its prime if none of the above criteria is met.
- It removes one edge at a time and calls itself (Algorithm I) with the digraph after edge removal. The output is concatenated to a list for all edges that are removed in iteration. The list is returned in output.
- already_checked_digraphs may be implemented as a dict with key as the Weisfeiler Lehman hash and value being list of all graphs with given hash.

Algorithm II – Find all prime digraphs

- Given an exhaustive list of all prime digraphs CDG_n with n nodes, we employ Algorithm II to build CDG_{n+1}, a list of all prime digraphs with $n + 1$ nodes.
- Code present at: https://github.com/oyeluckydps/directed_graph/blob/master/prime_DiGraph_Generator.py
- A digraph is chosen from CDG_n iteratively. It has nodes labelled from $0\ to\ n - 1$.
- A new digraph is generated by adding a new node (n). It is connected to original digraph through an in-edge and an out-edge on $node\ n$, i.e. $(a, n)\ and\ (n, b)$ edges are added to the digraph, $where\ a, b\ \in V(digraph)$.
- There are n^2 possibilities for (a, b) and all of them are explored for new prime generation.

Algorithm II – Find all prime digraphs

- Given the newly generated digraph, it is reduced to all possible prime digraphs using Algorithm I – Reduce to Primes.
- Newly generated primes for all possible values of a, b and for all CDG_n digraphs are concatenated in a list after checking for existing isomorphic digraphs.
- The correctness of Algorithm II to generate exhaustive list of all prime digraphs of $n + 1$ is provable.
- One may start with single prime digraph of 2 nodes and generate prime digraphs for other node values iteratively.



$P_1^{(2)}$

Algorithm II – Find all prime digraphs

```
def compute_next_primes(cdg_n):
        all_primes = []; already_checked_digraphs = {}
        for DG_n in cdg_n:
                for in_edge, out_edge in new_edge_possibilities:
                        DG_n_temp = DG_n.add_edges_from_list([in_edge, out_edge])
                        reduced_DGs_list, already_checked_digraphs =
                        reduce_to_primes(DG_n_temp, already_checked_digraphs)
                        all_primes.add(reduced_DGs_list)
        return all_primes
```

- **`new_edge_possibilities`** is list of all possibilities for in_edge and out_edge in following form:

$$[\ \big((0,n),(n,0)\big),\big((1,n),(n,0)\big),\big((2,n),(n,0)\big),\ldots\big((n-1,n),(n,0)\big),$$
$$\big((0,n),(n,1)\big),\big((1,n),(n,1)\big),\big((2,n),(n,1)\big),\ldots\big((n-1,n),(n,1)\big),$$
$$\ldots$$
$$\big((0,n),(n,n-1)\big),\big((1,n),(n,n-1)\big),\ldots\ldots\ \big((n-1,n),(n,n-1)\big)]$$

- **`reduce_to_primes`** is described in previous slides.

Result: Find all prime digraphs

- Algorithm II was used to generate all prime digraphs for 2 to 10 nodes. It can be found at: https://github.com/oyeluckydps/directed_graph/blob/master/using_isomorphic_hashCDG.csv
- Count of prime digraphs for given number of nodes is:

Number of nodes	Total Prime Digraphs
2	1
3	2
4	5
5	15
6	63
7	288
8	1526
9	8627
10	52021

Correctness of reduce_to_primes

- Every graph that is a member of reduce_to_primes' output tests positive on is_prime algorithm. Thus, showing correctness of is_prime is sufficient.
- Re: For a graph to be prime (edge-critical strongly connected digraph) it must be –
  - Strongly connected: Any node is reachable from any other or itself.
  - edge-critical: Removing any edge must result in a non-strongly connected graph.
- The algorithm first checks if graph is strongly connected[9,10].
- The algorithm then removes edges iteratively and checks them for non-strongly connectedness.
- These steps of algorithm follow directly from the definition of prime graphs. Thus, every graph that is a member of reduce_to_primes' output set is prime.

Completeness of reduce_to_primes

- Now we shall prove that any graph
  - whose nodes are same as that or original graph (the graph being reduced)
  - whose set of edges are a subset of the set of original graph's edges
  - is prime.

  will be found using BFS (or DFS) on original graph.
- The algorithm removes edges iteratively in BFS mechanism and terminates on a leaf graph if either
  - The leaf graph found by removing edges is not strongly connected.
  - The leaf graph is prime.
- To prove the correctness, we assume the contradictory, i.e. let us assume that a graph adhering to the three conditions mentioned above is not a member of the output set.

Completeness of reduce_to_primes

- To prove the correctness, we assume the contradictory, i.e. let us assume that a graph G adhering to the three conditions mentioned above is not a member of the output set of reduce_to_prime algorithm with input graph I.
- We know that adding one or more edges to a prime graph will result in a strongly connected graph that is not prime graph.
- First we find the difference set of edge $D = \mathcal{E}(I) \setminus \mathcal{E}(G)$; here $\mathcal{E}(I)$ denotes the set of edges of graph I.
- Then we form a path from $G\ to\ I$ in the search space by adding one member of D (an edge found in original graph but not in exception graph) at each step.
- Since all the graphs on this path are strongly connected and none of them are prime so there exists a path using BFS (or DFS) from $I\ to\ G$ i.e. from original graph to the exception graph. This is a contradiction.

Correctness of prime generator

- The prime generator algorithm output a set of prime graphs.
- In the final step of algorithm for prime generation, the reduce_to_primes algorithm is employed.
- Since, the correctness of reduce_to_primes is established, we can be sure that any prime in the output of prime generator is also a prime.

Completeness of prime generator

- To prove that all

Conclusion

- An algorithm to find the set of all prime (edge-critical strongly-connected) digraphs for any given number of nodes is found. The algorithm is iterative in nature.
- An algorithm to generate one random prime digraph for any given number of nodes is found which is computationally more efficient than the above-mentioned algorithm.
- A random strongly-connected directed graph can be generated by using any of the above method to generate a prime digraph and then employing the $G(n, p)$ Erdős–Rényi model[11] to add edges.
- A theoretical proof of correctness for Alorithm II to generate exhaustive list of all prime digraphs of $n + 1$ nodes is found.

Further study

- This is the first step in study of random strongly-connected directed graphs and further study may be conducted in following direction:
  1. Study of time/space complexity of the mentioned algorithms.
  2. Optimization of the mentioned algorithms.
  3. Optimization of the reduce_to_primes and is_prime algorithms.
  4. Peer review of correctness and completeness of Algorithm II - to generate exhaustive list of all prime digraphs of $n + 1$ is provable.
  5. Deployment of strongly-connected digraphs in network studies.
  6. Resilience test of AI agents on games with underlying strongly-connected digraphs as playground.

Reference

1. Douglas B West (2001). 1.1.2 Introduction to Graph Theory. Pearson Education, Inc
2. Douglas B West (2001). 1.4.2 Introduction to Graph Theory. Pearson Education, Inc
3. Douglas B West (2001). 1.4.12 Introduction to Graph Theory. Pearson Education, Inc
4. Wikipedia contributors (2022a, June 25). K-edge-connected graph. https://en.wikipedia.org/wiki/K-edge-connected_graph
5. Verma, A. Kumar (2022, December 11). Directed Graph and analysis. GitHub. https://github.com/oyeluckydps/directed_graph
6. Wikipedia contributors (2022c, November 28). Graph isomorphism. Wikipedia. https://en.wikipedia.org/wiki/Graph_isomorphism
7. L. P. Cordella, P. Foggia, C. Sansone, M. Vento (2001). "An Improved Algorithm for Matching Large Graphs", 3rd IAPR-TC15 Workshop on Graph-based Representations in Pattern Recognition, Cuen, pp. 149-159. https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.101.5342
8. Shervashidze, Nino, Pascal Schweitzer, Erik Jan Van Leeuwen, Kurt Mehlhorn, and Karsten M. Borgwardt (2011). Weisfeiler Lehman Graph Kernels. Journal of Machine Learning Research. http://www.jmlr.org/papers/volume12/shervashidze11a/shervashidze11a.pdf

Reference

9. R. Tarjan (1972). Depth-first search and linear graph algorithms. SIAM Journal of Computing 1(2):146-160.
10. E. Nuutila and E. Soisalon-Soinen (1994). On finding the strongly connected components in a directed graph. Information Processing Letters 49(1): 9-14
11. Erdős, P.; Rényi, A. (1959). "On Random Graphs. I" (PDF). Publications Mathematicae. **6**: 290–297.