Data Science for Fintech Regtech and Suptech: Methodological Foundations and Key Applications 2024-2025

Imperial College London

Department of Natural Sciences, Mathematics with Finance MSc

Algorithm to Find the Minimum Number of Edges to Strongly Connect a Weakly Connected Graph

Authors:
Waner Chen, Thomas Pink (CID: 01854827, 06003339)

Date: December 15, 2024

1 Executive Summary

Firstly we consider some actual applications to the real world, with a focus on resilience of payment networks, or other finance based applications.

Having established the usefulness of such an algorithm, we will develop and define some basic mathematical machinery, make our specific statement, and then prove it, specifically proving that, for G not strongly connected, that it can be strongly connected with:

$$\max(SOURCE(C), SINK(C)) \tag{1}$$

where C is the condensation of the graph G, and $SOURCE(C)$ and $SINK(C)$ are the number of sources and sinks in the condensation respectively.

As the argument we develop to prove our statement is in fact constructive, this serves as the basis for coding the algorithmic implementation. However, there are a couple of subtleties in the algorithm that require further explanation.

Having done this, we show that the complexity of our implementation is

$$O(V + E + H^2) \tag{2}$$

With V the number of vertices, E the number of edges, and H the number of sinks in the condensation.

Finally we apply the algorithm, demonstrating the complexity curves on both synthetic and real world graphs.

2 Introduction

We start off by motivating with a couple of non-finance-based examples.

2.1 Wikipedia ‘Game’

There is a well-known Wikipedia ‘Game’, of attempting to navigate from any page to any other page in as few clicks as possible, using only links in the article. This being possible for any two pages, is essentially the statement that the graph, with vertices as pages, and directed edges as links from one page to another, is strongly connected.

Unfortunately, Wikipedia is not strongly connected, but it would be nice if the game were possible for all pages. To avoid being punished by the Wikipedia editors, we may ask the question, what’s the minimal number of links (edges) we need to add in order for the game to be possible?

2.2 Spy Ring

Suppose a spymaster has a communication network between various agents out in the field. Unfortunately, a particular method of communication becomes compromised, and safe communication between the agents becomes impossible.

Every connection made to agents, and between agents is costly, as there is a chance it could be discovered by the enemy. With an unconnected communication network (with potentially one-way communication, e.g. radios that can receive, or carrier pigeons etc.), the spymaster wishes to ensure he can send and receive a message from any agent, while not adding too many risky connections. I.e.to strongly connect the network while adding as few edges as possible.

We now consider finance-based examples.

2.3 Settlement Networks

Suppose we have a series of transactions, all of which need to be settled through trading assets through certain exchanges and participants (as a directed network). Due to possible downtime issues at certain exchanges or should a participant fail, it would be desirable to ensure some additional resiliency in the payment network.

Therefore, by removing that participant, and all associated edges, the graph may no longer be strongly connected or could even become disconnected. Therefore, adding minimal edges to ensure that the remaining graph is strongly connected would enable a more resilient settlement network.

2.4 High Frequency Trading Networks

It is now common for high-frequency traders to use sometimes unique methods of transporting data from one exchange to another, in order to gain informational advantage in reflecting prices at both exchanges. For example, the use of microwave towers in the US, rather than fiber optic cables, to reduce latency times between New York and Chicago.

However, there is a certain vulnerability with such networks in that, despite their high speeds, rerouting may be required in the event of a faulty repeater. These disruptions may prevent two-way communication for a significant amount of time. Identifying how we can minimally strongly connect the disconnected communication network may allow the identification of alternative communication methods for repair or improving resiliency.

3 Mathematical Statement and Proof

Notation and lemmas used in the proof are explored in more detail in the appendix.

3.1 Generalised Statement

We originally developed our method according to the problem statement, i.e., just for a weakly connected graph. But in fact we can drop the need for the graph to be weakly connected, and give a restatement of the problem on a general directed graph.

Theorem 1
Let G be a directed graph that is not strongly connected, with finite condensation C. Then the minimum number of edges required to strongly connect G is $\max(SINK(C), SOURCE(C))$*, and this can always be achieved.*

The condensation of Figure 1 as seen in Figure 2 has sources $\{1,2,3\}$ and sinks $\{4,5\}$, so it should take at least 3 edges to strongly connect it. We achieve this in Figure 3.

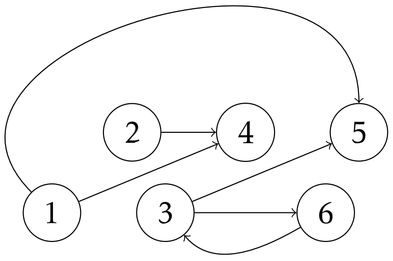

Figure 1: An example of a weakly connected graph

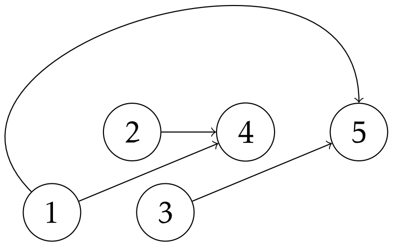

Figure 2: Condensation of Figure 1

3.2 Comment and Ideas of Proof

We begin by providing motivation for the ideas behind our approach.

Firstly, we do not need to add edges between any vertices in a component that is already strongly connected. Therefore, it is natural to consider these as a single

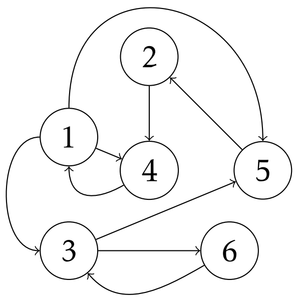

Figure 3: Showing that we can strongly connect Figure 1 with 3 additional edges, (4,1), (5,2) and (1,3)

vertex in some sense, and add edges to this graph. Intuitively, adding edges to this graph, where we ignore the connections between strongly connected components, is the same as strongly connecting the original graph.

Then it is clear that the sources and sinks are what we should focus on, since a graph cannot be strongly connected (except for a trivial vertex) if a source or sink exists, as we cannot leave or enter the vertex, respectively. It is also natural to connect the sink to the source, since this necessarily means we can leave what was previously a sink and enter what was previously a source. This argument establishes the minimum bound.

Therefore, our goal is to reduce the number of sources and sinks both by 1 each time we add an edge as far as possible.

The tricky part of the argument, dealt with in Step 1, is ensuring that when we add our edges, we do not accidentally create a new source or sink. To illustrate why this is a problem, consider the condensed graph shown in Figure 4

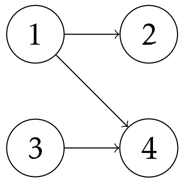

Figure 4: Adding an edge from 2 to 1, and 4 to 3 does not strongly connect the graph, even though each time we add an edge from sink to source. Adding 2 to 1 creates a source, and 4 to 3 creates a sink.

3.2.1 Proof Outline

Before jumping into the proof, we illustrate the proof on an example graph, Figure 5

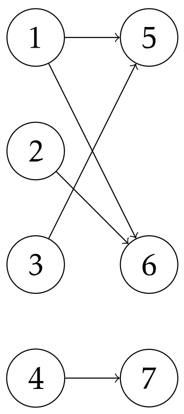

Figure 5: As we consider only the visions of sources, we can think of the condensed graph in this way.

In Step 1, we aim to ensure that a source can reach all sinks by connecting sinks it cannot yet reach to it. In doing so, we ensure there is always a path from a source to a sink, thereby avoiding the creation of a new sink or source. See Figure 6

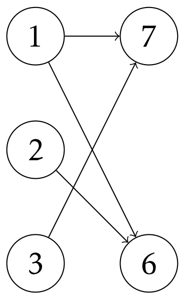

Figure 6: After step 1 in the algorithm above, we connect 5 to 4, and after condensing are left with the following graph, in which 1 can see all remaining sinks.

In Step 2, once there is a source that can reach all sinks, we repeatedly replace this source with another by connecting a sink in the new sources vision to the original source. We repeat this until one source or one sink remains. Since we always ensure that one source or sink remains, we know there is a path from the source to the sink each time, and thus we do not create a new sink or source when adding an edge. See Figure 7

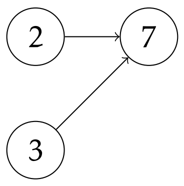

Figure 7: After step 2 in the algorithm, we connect 6 to 1, so that 2 can see all the remaining sinks. Since there is only 1 sink remaining we stop and move on.

In Step 3, we connect the remaining source to the remaining sinks, or the remaining sink to the remaining sources, as appropriate. In the examples, as seen in 7 this connects 7 to 2 and 3.

We now move on to the proof itself, which also serves as an algorithm for computing a set of possible edges to add to achieve the goal.

3.3 Proof

Proof 1

We begin with $R_1 = C$ the condensation of G, a weakly connected graph.
Denote the set of sources in R_i as S_i and likewise the set of sinks H_i. Our goal is to connect the sinks to the sources in such a way that we can traverse all the points in the graph. We consider only the visions, since all transitive vertices by definition can be reached on some path from a source to a sink.

Step 1:
Begin by taking a source $s \in R_i$, if $Vis(s) = H_i$ move onto the next step. If not by the union of the visions of the sources being the set of sinks, there must be a source $\hat{s}$ which can see a sink not seen by s. Add an edge between an element $v \in Vis(s)$ to $\hat{s}$. In this new graph s can see $Vis(s) \cup Vis(\hat{s}) \setminus \{v\}$.

This process could not have condensed down to a create another additional source/sink (which would not have decreased the number of sources and sinks by 1), since we have a path from the source we just relabelled, to the sink we can now see in the vision of the source, that goes through the edge we just added, (and in particular we know the vertices in the condensation representing the components of each of the items we just added an edge to are transitive).

Therefore we have reduced the number of sinks and number of sources by 1, by v and $\hat{s}$, respectively.

Condense this graph to obtain R_{i+1}. We note that except for the vertices we added an edge to if a source was a source it must remain a source, and a sink must remain a sink. This condensation is still weakly connected as we know we are still connected, and not strongly connected since we have at least one source, s, and therefore also at least one sink since $Vis(s) \neq H_i$. Relabel in the obvious way so that sources and sinks retain their labels.

Repeat this step (including the check at the beginning).

Step 2:
If we only have one source or one sink left move onto the next step. Now take another source $\hat{s}$ that is not s. Pick a sink $h \in Vis(\hat{s})$ and connect h to s. Now in this graph

$Vis(\hat{s}) = Vis(\hat{s}) \cup Vis(s) = H_i \setminus \{h\}$, i.e. all the remaining sinks. So condense again, relabel $\hat{s}$ to be s in the new graph, and we then again have $Vis(s) = H_{i+1} = H_i \setminus \{h\}$.

Again through similar argument to last time, we could not have resulted in the source/sink condensing down to a source or a sink, since we know we have a path from the new primary source to all the sinks through the edge we added. Therefore again we have reduced the number of sinks and number of sources by 1, with all other sources remaining sources, and all other sinks remaining sinks as in step 1. Then repeat this step (including the check at the beginning).

Step 3:
We are now left in one of three cases.

Case 1:
We have one sink and one source left. Connect them, sink to source. We reduce the number of sinks and sources by 1. Since at each stage we reduced the number of sinks sources by 1, we must have used $SOURCE(C) = SINK(C)$ edges.

Case 2:
We have one source and a number of sinks. Connect each sink to the source. In this case, for each edge we add, we reduce the number of sinks by 1. Since at each stage, we reduced the number of sinks by 1, we must have used $SINK(C)$ edges. It is clear that there were more sinks than sources, as we reduced the number of sources by 1 each time we added an edge before this step.

Case 3:
We have one sink and a number of sources. Connect the sink to each source. In this case for each edge we add we reduce the number of sources by 1. Since at each stage we reduced the number of sources by 1 we must have used $SOURCE(C)$ edges, and it is clear that there were more sources than sinks, as we reduced the number of sinks by 1 each time we added an edge before this step.

Our final condensed graph is now clearly strongly connected, and in total we added $\max(SOURCE(C), SINK(C))$ edges. Finally using lemma 1 and remark 3, we can go backwards along the chain of condensations, undoing each one while retaining the graph being strongly connected, showing we can strongly connect G with the theoretical minimum number of edges.

Finally we show the case where we don't assume G weakly connected.

If G is not weakly connected, condense it, and then take each weakly connected component $C_1, ..., C_n$. Each component will have at least one source and at least one sink. Since G is finite we can connect the condensation by adding edges between C_i and C_{i+1} from a sink in C_i to a source in C_{i+1}. Again we note this reduces the number of sources and sinks by 1 each time. We then apply the algorithm to this graph, now weakly con-

nected, which again reduces the number of sources and sinks by 1 each time (until the last step, where it will reduce in the same way as before), and therefore also achieves the minimum.

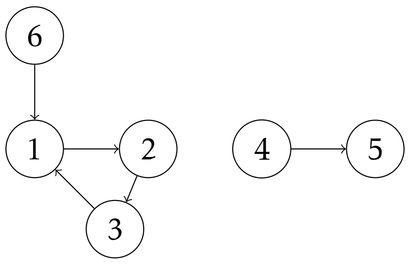

Figure 8: This graph is not even weakly connected

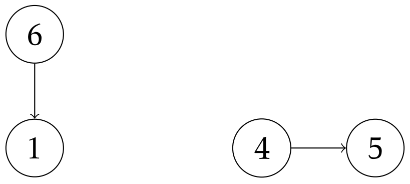

Figure 9: We can condense it

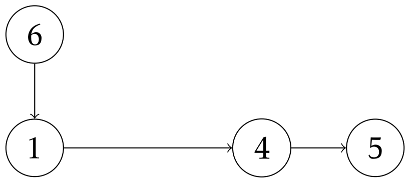

Figure 10: And then weakly connect it, reducing the number of sources and sinks by 1 each. The condensation of this new graph is identical to itself.

□

4 Algorithmic Implementation

To begin with, we work with visions and condensations throughout our algorithm. We start by explaining how to compute the visions of all the sources of the condensation, and then move on to computing the condensation itself.

Once we have these two objects, the algorithm proceeds as outlined in the proof of the theorem. (Although in our coded implementation, we do not need to recondense each time as the sources sinks and visions remain identical once the joined sink and source are removed.)

In step 1, we have to identify which sources and sinks to connect to expand the vision of the first source to everything. This has to be done so that we always add a new sink, and this means we must give some thought on how to find such a list of sinks and sources to connect efficiently.

Steps 2 and 3 just iterate until we have connected all the remaining sources and sinks, so is at most $O(V)$ in the worst case scenario where the original graph G was already acylic.

4.1 Computing Visions

We considered two methods, and implement both for comparison.

4.1.1 Depth First Search of Successors of Each Source

Depth First Search begins at a given node and explores each possible path deeply into the graph until no further nodes can be visited, at which point it backtracks. Initializing an empty set to track visited nodes and recursively visiting all reachable neighbors from the starting node. For each unvisited neighbor, the algorithm marks it as visited and continues the search using that neighbor as the new starting node. Once the search is complete, the set contains all the successor nodes that can be reached from the initial node.

As we only wish to record those vertices that are sinks, these are recorded separately.

Assuming the graph has V vertices and E edges, the time complexity is $O(V+E)$, as each vertex and edge is visited at most once. For each pass through to identify the successors of each source.

However, we must do this for every source in order to compute the visions, and so the complexity of this algorithm in total is $O(S(V+E))$ where S are the number of sources.

4.1.2 Back Propagation from Sinks

We perform a back propagation working backwards from the sinks.

The algorithm is fairly simply illustrated with Figures 11, 12, 13, 14.
In more formal terms. At each step, from each sink we label all immediate predecessors with the relevant labels (being the set of sinks that that element could see) and then remove the sinks at the end, obtaining a new graph.

Repeating this until we are left with only the sources provides us with a way to compute the visions of each source efficiently, as we only had to check each vertex and edge once, and therefore this algorithm is $O(V+E)$.

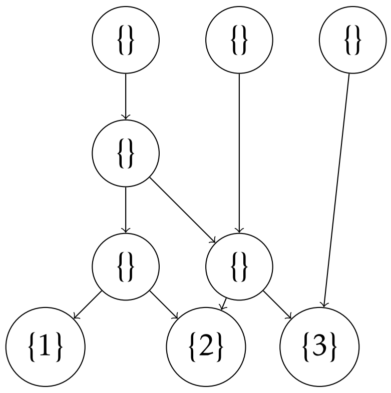

Figure 11: Step 1 of Back Propagation

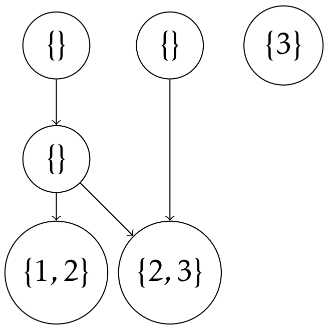

Figure 12: Step 2 of Back Propagation

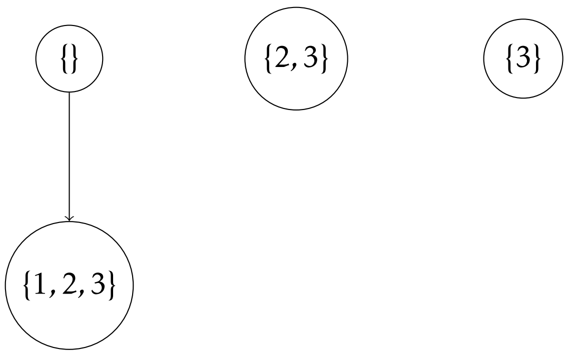

Figure 13: Step 3 of Back Propagation

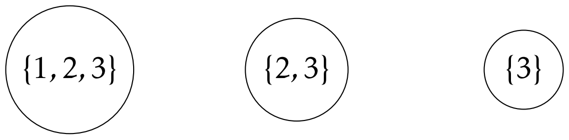

Figure 14: Step 4 of Back Propagation

4.2 Condensation

Condensing the graph essentially comes down to identifying the strongly connected components.

Creating the condensation from the components is simple. After first choosing representative vertices for each component, we need to add the edges. We do so by iterating over all the edges, adding the edge to the condensation if we haven't already, so long as it's between two strongly connected components. This is O(V+E) since we have to check every edge and vertex at most once.

There are algorithms that are able to do a single depth first search of the graph to identify strongly connected components, and therefore have complexity O(V+E). Specifically Tarjan's algorithm. [9]

For efficiency purposes, in our implementation we use *NetworkX*'s [3] implementation of the algorithm. However, for completeness we have also created our own method for identifying strongly connected components.

4.2.1 Identifying Strongly Connected Components Using Successors

First, through Depth First Search of the Successors of Each Source, the successors of each node are calculated one by one are stored in a dictionary. This step aims to construct a mapping that contains reachability information for each node. The complexity of this step is $O(V(V+E))$ since we have to a depth first search on each node.

Before identifying the strongly connected components (SCC), two structures are prepared:
A container for storing the SCCs, where each SCC is represented by a set of nodes belonging to that component.
A set to track processed nodes, avoiding redundant computations and improving efficiency.

Each node and its successor set are traversed sequentially. For the current node, it is checked whether it is mutually reachable with certain nodes in its successor set. If the current node can reach another node and that node can also reach the current node, these two nodes belong to the same SCC. These nodes are grouped into the same set and marked as processed.

During the traversal, if a node is found that does not have mutual reachability with any other node and has not been marked as processed, it is treated as an isolated weakly connected component and added to the result.

After the traversal, the container storing the SCCs will contain all the strongly connected components in the graph, with each component represented by a node. These components are the results identified based on the mutual reachability relationships between nodes.

In the most ideal scenario, the graph is strongly connected, and the time complexity is $O(V)$. In the worst-case scenario, where there are no strongly connected components and all components are weakly connected, each node and its descendants will need to be looped through, resulting in a time complexity of $O(V^2)$.

Therefore the overall complexity of this step is dominated by the creation of the dictionary in the first place, which is $O(V(V+E))$.

4.3 Step 1 of the Algorithm

Our goal is to identify a sequence of sources to add, such that the union of their visions is strictly increasing in size. This ensures that whenever we connect our sink to source, since we had a new sink in the vision each time, we must have a path from source to sink, preventing the problem identified in Figure 4

To accomplish this, we make use of the algorithm to compute visions twice. Once on the original condensation, and once on the condensation with edges reversed.

This provides us with for each source, the sinks it can see, and also for each sink which sources can reach it.

To identify our list we iterate along a list of all sinks, marking whether they are visible or not.

If the sink is already in the vision of the set of sources we have found, we do nothing, but if it is not, we identify a source that can see it, and then mark all sinks in that source's vision as visible in our list.

In this way we obtain a list of sources, where each source we add contains a sink that the previous set of sources could not see.

Furthermore recording the sinks and sources identified at each stage in this process provides us with exactly the list of sinks and sources we can connect to obtain a source that can see everything.

Since we update the list of visible sinks for every sink in the vision of the source, that step could be as bad as if we add back the same sinks we already knew were visible. I.e. we add the first sink, the first and then the second, and so on. $O((1+2+...+H)) = O(H^2)$.

As we required two breadth first searches on the condensation, which has at most V vertices and E edges, it was potentially $O(V+E)$. Therefore this step is potentially

$O(V + E + H^2)$ but is likely better in practice.

In the case of single vertex pointing to sinks, then $H = V - 1$, giving the algorithm a theoretical worst complexity of $O(V^2 + E)$

However the current implementation of this step may be made more efficient, as the current algorithm's requirement for the list of sources to be strictly increasing in total vision is quite strict.

4.4 Theoretical Algorithmic Complexity

Apart from step 1 we achieve $O(V + E)$ complexity. This is in some sense optimal, since we must iterate over each vertex and edge to even ensure that graph is fully connected, by adding an edge whenever there isn't one, let alone an algorithm that will ensure it is minimally strongly connected.

However as a result of the inefficiency of Step 1 our algorithm is potentially as poor as $O(V^2 + E)$.

5 Algorithmic Complexity Results

We run the algorithm on a series of synthetic and real world data graphs. The details of how this data was obtained and created is included in the appendix. We produce figures of the time the algorithm took to complete vs the number of edges and vertices, and the number of sources and sinks.

We first look at synthetic data

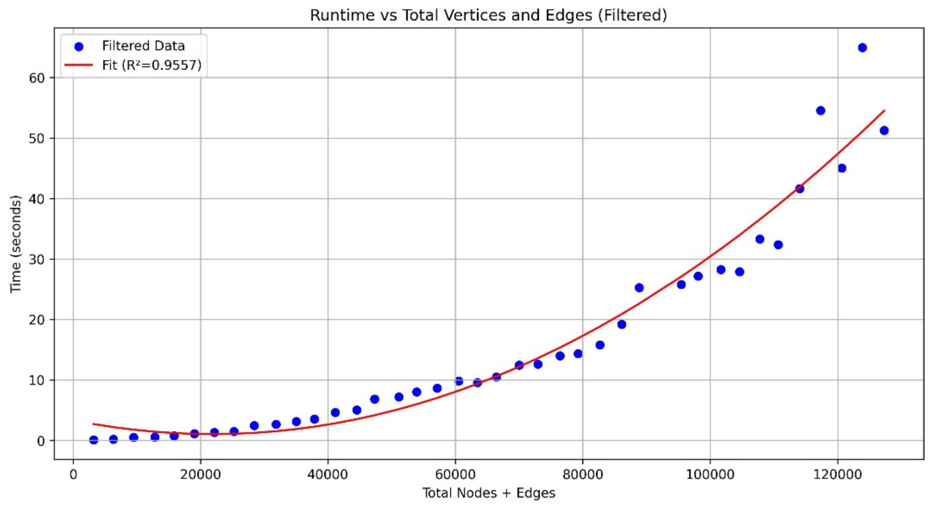

Figure 15: Runtime on Scaled Graphs against total Vertices + Edges with Square Fit Regression Line

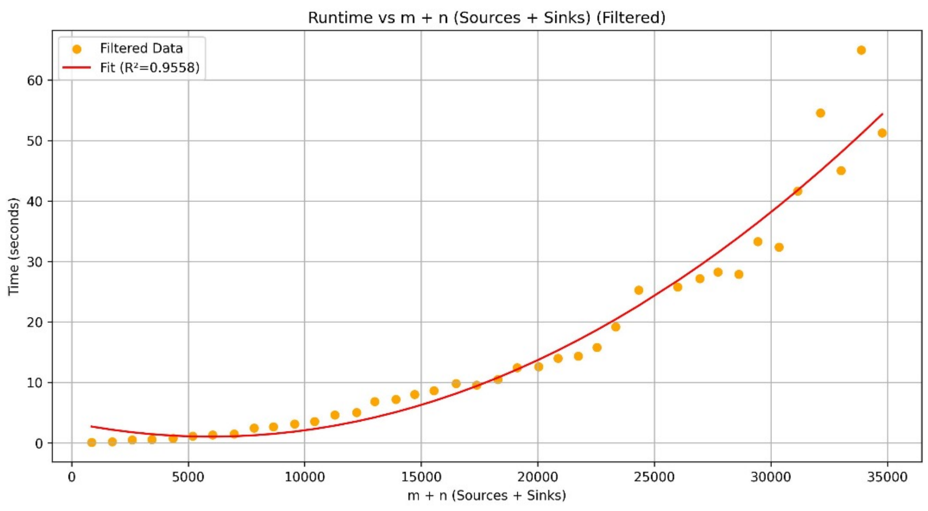

Figure 16: Runtime on Scaled Graphs against total Sources + Sinks with Square Fit Regression Line

On the synthetic data we note that we appear to have a very good square fit, which supports the $O(V+E+H^2)$ hypothesis. In fact the results on sources and sinks for the synthetic data in Figure 16 are a scaled down version of Figure 15, which is likely as a result of how scale free graphs are generated (but is not explored further here). Secondly we examine complexity on the real data.

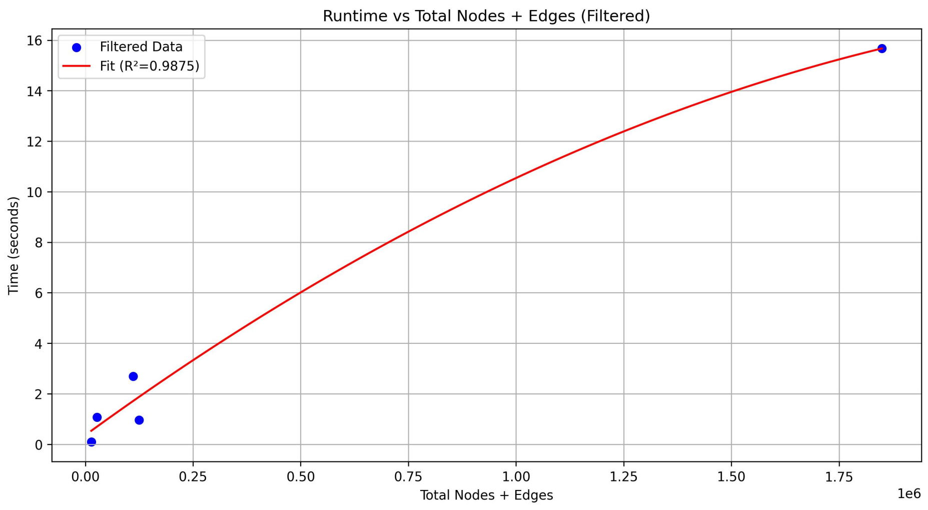

Figure 17: Runtime on a selection of Real Data Graphs against total Vertices + Edges with Square Fit Regression Line

In Figures 17 and in 18 the number of sources and sinks are significantly smaller than the number of vertices and edges. In this case, therefore the time complexity in the vertices and edges in Figure 17 appears to be almost linear with a square fit, as the graphs are dominated by the original total number of vertices and edges rather than the number of sinks.

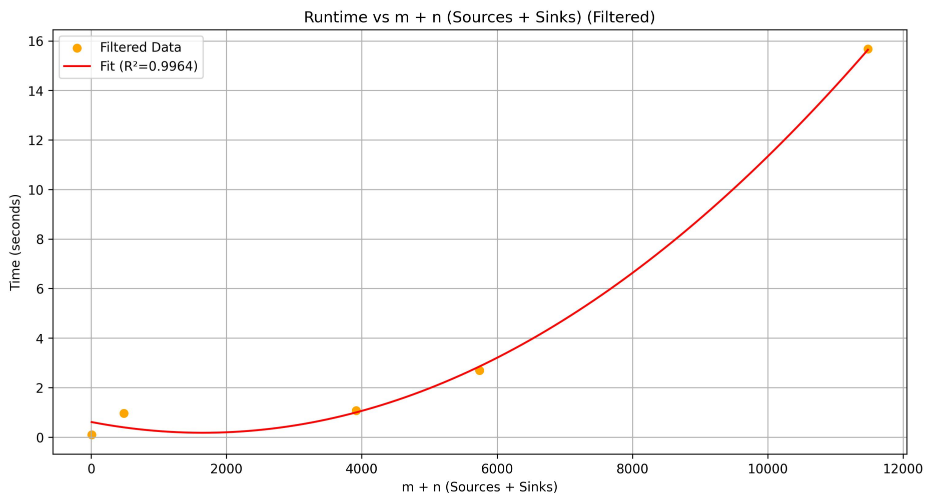

Figure 18: Runtime on a selection of Real Data Graphs against total Sources + Sinks with Square Fit Regression Line

In contrast Figure 18 shows that we return to the square fit, which supports again our hypothesis that the complexity of the current algorithm's implementation is $O(V + E + H^2)$

6 Discussion

Our current implementation in python is a prototype, and relies on set functions to perform a lot of the identification of elements of visions and current visible sinks to add, which is slow and likely unnecessary.

Unfortunately, finding suitable real world graphs to test the algorithm on is difficult, and so we did not test on many, and so the actual complexity may become more apparent after testing on additional graphs.

We achieve a reasonably fast algorithm complexity wise. To improve speed in practice, if achieving the exact minimum is not required, we can make use of parallelisation. In particular, the graph can be split into components, and then in parallel the algorithm can be applied to each of the component's components, and then finally to the network of now strongly connected components.

Our algorithm finds a set of edges that could be added, but in practice any vertex to any other within each strongly connected component could have been chosen, and a different sequence of connecting source and sink components to each other could have been chosen. A possible extension to the problem could be providing explicit penalties to adding each edge, and then minimising the overall penalty required to strongly connect the graph. (Recovering our solution where the penalties are equal weight)

Finally, we note that we found a theoretical minimum for strongly connecting the graph, and then proved we could always achieve it, as well as identifying an algorithm for computing the edges required explicitly!

7 Conclusion

We proved a hard result and non-obvious fact, to show the exact minimum to strongly connect any finite directed graph, and that it is always attainable. In particular, we found that exactly

$$\max(SINK(C), SOURCE(C)) \tag{3}$$

edges are required.

We wrote an explicit algorithm to compute the specific edges required much more efficiently than a brute force approach.

In particular, the complexity is mostly dependent on the number of sources and sinks in the condensation of the graph, and we have hypothesised and evidenced a complexity of:

$$O(V + E + H^2) \tag{4}$$

References

[1] Christian G Fink et al. "A centrality measure for quantifying spread on weighted, directed networks". In: *Physica A* (2023).

[2] Jerry Grossman. *The Erdos Number Project*. As at 09/12/2024. 2024. URL: `https://sites.google.com/oakland.edu/grossman/home/the-erdoes-number-project`.

[3] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. "Exploring Network Structure, Dynamics, and Function using NetworkX". In: *Proceedings of the 7th Python in Science Conference*. Ed. by Gaël Varoquaux, Travis Vaught, and Jarrod Millman. Pasadena, CA USA, 2008, pp. 11–15.

[4] J. Kleinberg J. Leskovec and C. Faloutsos. "Graph Evolution: Densification and Shrinking Diameters. ACM Transactions on Knowledge Discovery from Data". In: *ACM TKDD* (2007).

[5] Jure Leskovec and Andrej Krevl. *SNAP Datasets: Stanford Large Network Dataset Collection*. `http://snap.stanford.edu/data`. June 2014.

[6] J. McAuley and J. Leskovec. "Learning to Discover Social Circles in Ego Networks". In: *NIPS* (2012).

[7] M. Ripeanu, I. Foster, and A. Iamnitchi. “Mapping the Gnutella Network: Properties of Large-Scale Peer-to-Peer Systems and Implications for System Design”. In: *IEEE Internet Computing Journal* (2002).

[8] Joelle Pineau Robert West and Doina Precup. “Wikispeedia: An Online Game for Inferring Semantic Distances between Concepts”. In: *21st International Joint Conference on Artificial Intelligence (IJCAI)* (2009).

[9] Robert Tarjan. “Depth First Search and Linear Graph Algorithms”. In: *SIAM J Comput.* (June 1972).

[10] Robert West and Jure Leskovec. “Human Wayfinding in Information Networks. 21st International World Wide Web Conference”. In: *WWW* (2012).

8 Appendix

8.1 Conventions

We begin by listing definitions and conventions which will be used throughout the report where we don't follow conventions used elsewhere in the course. Definitions are given for any new concepts not covered in the course.

Definitions such as strongly connected and weakly connected are also included given the importance they have in relation to the problem.

8.2 Some Initial Definitions

Definition 1 (Graph)
A graph $G(V,E)$ is a set of vertices V, and a set of pairs of vertices E, with elements (u,v) with $u,v \in V$. A directed graph has the elements of E being ordered, i.e. $(u,v) \neq (v,u)$ whereas an undirected graph has the elements of E unordered.

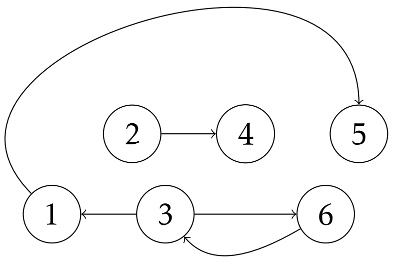

Figure 19: An example of a directed graph, with directed edges represented as arrows, and vertices represented as circles

Definition 2 (Strongly/Weakly Connected)
A directed graph is strongly connected if there is a directed path from any vertex to any other. A graph is weakly connected if there is a path joining any pair of vertices without considering the direction of edges.

Definition 3 (Sinks, Sources and Transient Vertices)
A vertex v in a directed graph G is a source if $v_{indegree} = 0$, and is a sink if $v_{outdegree} = 0$. We denote the number of sinks as SINK(G) and number of sources as SOURCE(G). The set of sinks will usually be denoted by H and the set of sources by S. A vertex is called transient if it is neither a sink or a source.

In Figure 1 vertices 1 and 2 are sources, 4 and 5 are sinks, 3 and 6 are transient.

Definition 4 (Component)
A component $C(U,F)$ of a graph $G(V,E)$, is a set of vertices U, where $U \subseteq V$ the set of vertices of G, and all associated edges $F \subseteq E$ between vertices of U in G. If a component of a directed graph is strongly/weakly connected then we say it is strongly/weakly connected respectively.
We say that a component is trivial if it is a single vertex.

In Figure 1 the strongly connected components are $\{1\},\{2\},\{3,6\},\{4\},\{5\}$. They all form a single weakly connected component. The component below in Figure 20 is formed from $\{1,5\}$

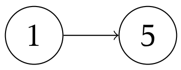

Figure 20: The component formed from vertices 1 and 5

Definition 5 (Condensation)
A condensation (also known as a reduction) $R(U,F)$ of a directed graph $G(V,E)$ is the graph obtained by replacing each maximal strongly connected component C (i.e. the one containing the most vertices possible) with a vertex c such that for each $c \in U$ we have that for $d \in U,\ c \neq d$ which represent the maximal strongly connected components (C,D) with$(C \neq D)$ respectively: $(c,d) \in F \iff \exists u,v \in C,D\ st\ (u,v) \in E$. We do not include self directed edges, and allow at most one edge between any two vertices in the condensation.

Definition 6 (Acyclic)
A directed graph G is acyclic if there are no paths from a vertex to itself.

Remark 1
The condensation of a graph is acyclic. If a cycle existed in the condensation, then those vertices are strongly connected, and so their strongly connected components would have been strongly connected in the original graph. Therefore they would be replaced by a single vertex in the condensation. This also shows that a graph being acyclic is equivalent to it having no none-trivial strongly connected components.

8.3 Required Lemmas

The problem is much simpler to think about after condensing the graph, as the condensed graph is acyclic. And so to make this proof simpler we require some lemmas that allows us to translate between the condensed graph and the original, and shows how the structure is preserved under condensation.

Lemma 1
Let R be the condensation of G. Then R is strongly/(weakly) connected $\iff$ G is strongly/(weakly) connected.

In particular if there is a path between u and v in G, with u and v being in components C_u and C_v respectively, then there is a path between c_u and c_v the corresponding vertices in R. Likewise if there is a path between c_u and c_v then for any vertices $\tilde{u} \in C_u, \tilde{v} \in C_v$, there is a path from $\tilde{u}$ to $\tilde{v}$.

This is relatively clear from looking at a picture, but we give a proof to show explicitly that it is true.

Proof 2
If G is strongly connected, condensing it leaves a single vertex, which is trivially strongly connected. So from now on we assume G is weakly connected but not strongly connected.

Take two vertices u and v in G that are weakly connected. Then there exists a path $v_1 v_2 ... v_n$ with $v_1 = v, v_n = u$.
Each vertex v_i lies in a strongly connected component C_{v_i} of G, and we know that either v_i, v_{i+1} are in the same strongly connected component, $C_{v_i} = C_{v_{i+1}}$ or they are in separate components. If in separate components we know there is an edge between the components, and so there is an edge between c_i and c_{i+1}.

So we obtain a sequence of vertices, either where they are connected (directed), or it is the same vertex. Advancing along this series in the logical way gives us a path from c_u to c_v.

Now instead consider a path $c_{v_1} ... c_{v_n}$ between $c_u = c_{v_1}$ and $c_v = c_{v_n}$. Within each of the respective strongly connected components C_{v_i} of the vertices c_{v_i} in the path we can reach any other vertex in the component. We also know that there must be an edge from a vertex in C_{v_i} to a vertex in $C_{v_{i+1}}$ by definition of the condensation. So we have a path from any vertex in C_u to any vertex in C_v by going to the component getting to the vertex with the connecting edge to the next component, advancing along that edge and so on.

Finally to show that the graph remains weakly connected we can repeat the above arguments between any two vertices, but instead disregarding the directions of the edges in our paths (i.e. we allow moves backward along directed edges.)

□

Lemma 2
For a directed graph G with condensation C, denoting a vertex $v \in G$ that is sent to corresponding vertex $c_v \in C$ in the natural way. Adding an edge (u, v) to G, and then condensing, or adding an edge (c_u, c_v) in C and then condensing obtains the same graph.

This is demonstrated by Figures 21 and 22.

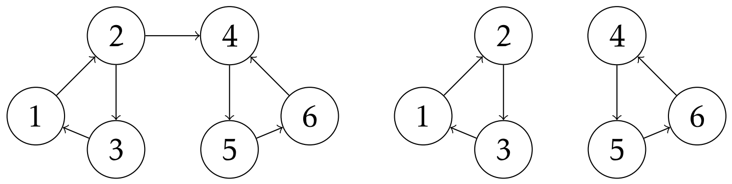

Figure 21: Condensing the right graph and then adding the edge, is the same as adding the edge between the strongly connected components and then condensing

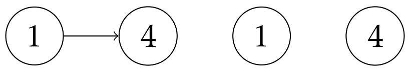

Figure 22: Results after condensing each graph, showing that it is clear that adding the edge before or after condensing does not matter

Proof 3
Add an edge in the original graph G, to obtain G', and add an edge in the relevant place of C to obtain C'. If we don't have a cycle in C', it is irreducible and we cannot condense it. So we did not alter any of the strongly connected components in G. To show G' condenses to C' we proceed by reducing all strongly connected components as before in G (the order we do this in condensing G is irrelevant), until we obtain C with the additional edge, making it C', which is irreducible, showing we have condensed G' to C'.
Suppose we have a cycle now in C', then again, proceed by reducing the elements in G, until we obtain C with the additional edge, which is C'. We then condense this as before. So it is clear we obtain the same condensation from this process. □

Remark 2
This shows us that by repeatedly adding edges to the condensation and repeatedly condensing it until we can tell it is strongly connected, will necessarily mean we have strongly connected the original graph. The new, now strongly connected, graph can be reconstructed by working backwards.

We can now prove formally that the minimum number of edges needed is as stated.

Proof 4
We work in the condensed graph, as noted by the above remark.
To be strongly connected, we note that every vertex must have in degree and out degree of at least 1. Adding an edge will increase the in degree of a vertex by 1, and the out degree of a vertex by 1. There are SINK(C) vertices of out degree 0 and SOURCE(C) vertices of in degree 0. So we must add at least $\max(SINK(C), SOURCE(C))$ *edges.*

□

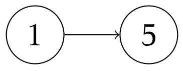

Figure 23: This condensation means that the original graph cannot be strongly connected, since we cannot re-enter the condensed component 1 once we have left it, as it is a source, or re-enter the component that became 1 once we have entered the component that became 5. This is true no matter how complicated the strongly connected components that formed 1 and 5 were.

Remark 3
This minimum remains true in infinite graphs! But as shown by the counterexample to the infinite case of theorem 1 it may actually be impossible to strongly connect an infinite weakly connected graph with finite edges unless its condensation is finite.

We now consider the vision of a vertex, which is a key concept in the remainder of the proof.

Definition 7 (Vision)
The vision, $Vis(v)$, of a vertex v in an acyclic graph G is the set of sinks that can be reached from that vertex. We say that v can see an element $h \in Vis(v)$

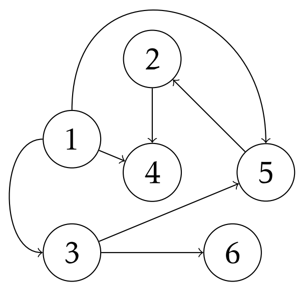

Figure 24: In the above graph, the vision of 3 is {4,6}

It will also be necessary to say that we can reach every vertex in C from a source. We require here for G to be finite.

Lemma 3
Every vertex in a condensed graph C generated from a finite weakly connected graph G can be reached from a source.

Proof 5
Take any vertex in C. Go backwards along paths until we cannot proceed any further. Since C is condensed it is acyclic. C is also finite since it has at most $|V|$ vertices. Therefore this must terminate at some point. (If not, either the graph is infinite, or we got to the same vertex twice, meaning we found a cycle.) The fact we cannot proceed any further shows we must be at a source. □

Remark 4
This shows that the union of the visions of the sources is the set of sinks for a finite weakly connected graph. I.e. for sources $s \in S$ sinks $h \in H$ then $\bigcup_{s \in S} Vis(s) = H$.

8.4 Results and Commentary of the Proof

We now include commentary and other supplementary results following the proof of the main theorem.

Remark 5
Within the theorem, we did not require G to be weakly connected, but we do require the condensation of G to be finite. A counterexample would be the infinite graph of vertices pointing one after another as seen in Figure 25. It is not strongly connected, since we cannot go backwards. It is already acyclic, so condensing it does not change the graph, and it clearly has zero sources and zero sinks. But so long as we add a finite number

of edges we cannot strongly connect it. There will always be a vertex v after the vertex from which the final edge we add goes from, and a vertex u before the first vertex to which an edge we added points, and there is clearly no path from v to u.

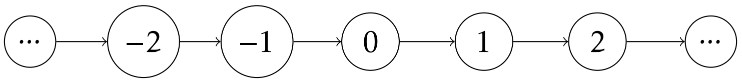

Figure 25: Example of a weakly connected infinite graph that has no sources or sinks in its condensation, but would require an infinite number of edges to strongly connect it.

The proof of our main theorem gives us a quite roundabout way of showing that G, and its condensation, must be finite.

Theorem 2
For a directed graph G with condensation C.

$$\text{G cannot be strongly connected with finite edges} \implies \text{C is infinite} \implies \text{G is infinite} \tag{5}$$

Proof 6
Apply the theorem noting that if C was finite, we could strongly connect G. □

Interestingly, we can't generalise this nicely to say

$$\text{C is infinite} \implies \text{G cannot be strongly connected with finite edges} \tag{6}$$

The following graph on $\mathbb{Z}^+$ with edges $(0, x),\ x \in \mathbb{N},\ (x+1, x)\ x \in \mathbb{N}$ is a counterexample

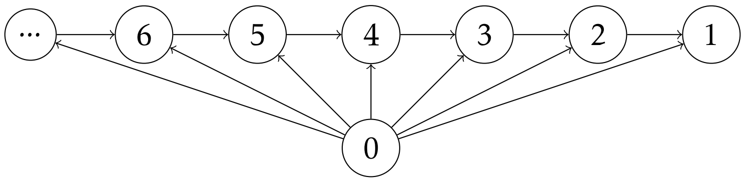

Figure 26: This graph is acyclic, and so its condensation is identical, and therefore infinite. But adding an edge from 1 to 0 will strongly connect it.

8.5 Methodology and Data

8.5.1 Import of Data

One of the websites for obtaining real-world data is the Stanford Large Network Dataset Collection (SNAP) [5]. This website provides a variety of networks, including the directed networks we are interested in. For example, in the social networks section, we can access the 'Social Circles from Twitter' [6] dataset.

The data can then be read and processed using *NetworkX* [3] to perform relevant calculations. The files will usually be available as a list of edges, in a `text`, `csv` or `edgelist` file. These are imported into *NetworkX* using the inbuilt read_edgelist method, which will construct the relevant graph, upon which the algorithm is applied.

8.5.2 Import of Realworld Data

We make use of the following data sets:

Wikispeed - A subgraph of the graph of wikipedia created from taking a directed edge between two pages whenever there is a hyperlink between them. [10] [8].

Wikivote - A directed graph created by recording the votes of users on Wikipedia to each other in elections to assign administrator roles to users. An edge exists where a user voted for another. [10] [8].

Twitter - A directed graph created by recording edges whenever a user has followed another, from webscraped data. [6]

Congress Network - A directed graph created by recording whenever two members of the 117th United States Congress interacted on Twitter (Liked, Retweeted, Followed etc). [1]

P2P Gnutella - A directed graph, with an edge recorded whenever a file was shared from one host to another on the P2P Gnutella File Sharing Platform on August 8th 2002. [4] [7]

8.5.3 Creation of Synthetic Data

Otherwise for synthetic data, we use the *NetworkX* scale_free_graph method in order to generate synthetic data upon which to test our algorithm upon. The graph of mathematician's co-authoring of papers is said to be scale free (as is used to compute the Erdos Number [2]