CMPT 477 Project: Graph Connectivity

By: Sahba Hajihoseini, Layan Barrieshee, Liza Awwad

Fall 2024

December 2, 2024

Introduction

This program checks the connectivity of undirected and directed graphs in Z3. The graphs are given as input from the user through text files and the program checks whether the graph is connected or not. The input files contain a U or D to specify the type of graph, the number of nodes (indexed from 1), and the edges between the pair of nodes. An undirected graph is connected if each vertex is reachable from every other node in the graph. To establish this property for a given graph, the depth first search algorithm and Z3 constraints were implemented. A directed graph is strongly connected if every vertex is reachable to every other vertex in the graph. For directed graphs, strong connectivity is determined using depth first search and Tarjan's algorithm to identify strongly connected components. All code was written by our members Sahba and Liza with references to websites such as Geek for Geeks and online GitHub repositories. Test cases were created by all our members.

Properties Being Verified:

The properties that the program ensures for undirected graphs are connectivity and reachability. Connectivity ensures that every node reaches all other nodes and reachability confirms that there exists a path between any two nodes. That means that if we start at node u_1 from the set of nodes U, we can reach the rest of the nodes from u_2 to u_n (with n being the number of nodes in the set).

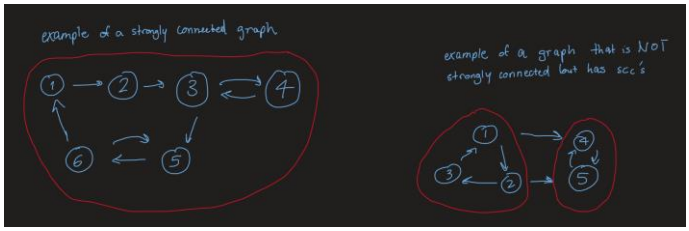

Figure 1: A strongly connected graph vs. a graph that is NOT strongly connected which has connected components

The properties that the program ensures for directed graphs are weak connectivity, strong connectivity, and reachability. A directed graph is weakly connected if there is a path between a pair of nodes no matter the direction of the edge. A graph is strongly connected if there exists a path in both directions between every pair of nodes in the graph. The reachability property, similar to the undirected graph, ensures that for any node in the set, there is a path that can be taken to any other node in the graph.

Explanation of the code structure and properties:

This program has two classes: a smaller class for storing the graph data and a larger class for the main function where the graph data is processed and checked for connectivity.

GraphInput Class:

The smaller class, GraphInput, is for creating a GraphInput object where the graph information is stored. It stores the type of graph as a Boolean (a directed graph as true and undirected as false), the number of nodes, and the edges. This GraphInput object is later called and used in the main function for informing the user of the graph connectivity.

Connectivity Class:

The main class processes the graph data from a text file. The file specifies the graph type (U or D), the number of nodes, and the edges. The data is parsed into a GraphInput object. For undirected graphs, the program uses a Depth-First Search (DFS) and Z3 constraints to verify connectivity. For directed graphs, the program checks strong and weak connectivity using DFS and Tarjan's algorithm.

The following functions are used in the main function in the Connectivity class:

- parseInputFile: This function is used to read and store the information of the graph. It identifies the graph type (U for undirected and D for directed), counts the nodes, and records edges as integer pairs by first storing them into a list. The parsed data is then stored in a GraphInput object for later use. Specifically, it is used to compare the number of elements visited versus the total number of nodes.

- DFS: We used a recursive implementation of the Depth-First Search (DFS) to traverse the graph using the following steps:

  1) For the base case, starting from a node, if it is already part of the visited set, we return it because it means that we have already reached it.
  2) Otherwise, we mark it as visited and add it to the set of visited nodes. We then recursively explore all reachable nodes marking them as visited.

  The runtime of this algorithm is O (|V| + |E|). Here, |V| represents the number of vertices in the graph and |E| represents the number of edges in the graph that we are visiting and traversing.

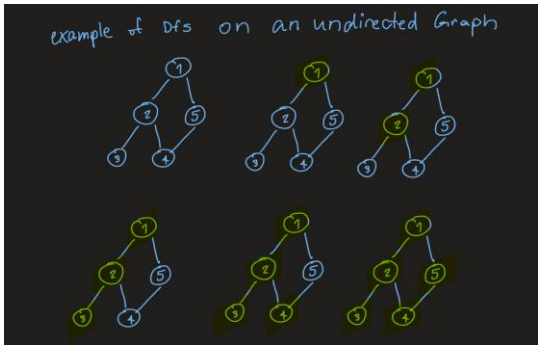

Figure 2: Depth first search on an undirected graph

- checkConnectivityUndirected: This function is used to check the connectivity of undirected graphs with the following steps:

  1) A context is initialized to manage the constraints.
  2) It builds an adjacency list to traverse the graph using a DFS (starting at the first node). The DFS determines the reachable nodes and marks the visited nodes by building a second adjacency list.

3) If all nodes are reachable (meaning the number of nodes in the graph is equal to the visited nodes), a Z3 context is initialized to apply constraints.
4) The following Z3 constraints are applied to validate mutual reachability:
   -Direction connection constraints ensure edges connect specified nodes bidirectionally.
   -Transitive closure constraints verify paths between nodes, accounting for intermediate nodes
5) The solver checks satisfiability, confirming connectivity if constraints hold.

- checkConnectivityDirected: This algorithm is used to check the strong connectivity of directed graphs, this algorithm with the following steps:

1) It builds the first adjacency list using the input graph. It builds the second adjacency list but this time with the reversed edges.
2) Runs the DFS algorithm twice: once on the input graph and once on the reversed graph. This is to verify reachability in both directions. If the number of nodes visited is less than the number of nodes in the graph, then some nodes could not have been reached. If any node is unreachable in either traversal, the graph is not strongly connected (it can either be disconnected or weakly connected).
3) Then, Tarjan's algorithm is applied to identify strongly connected components.

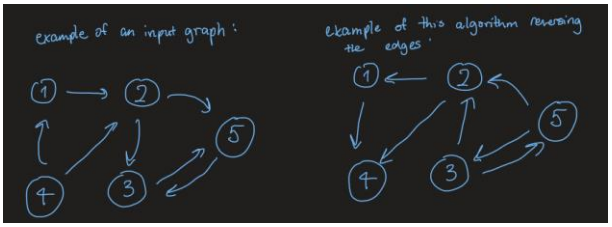

Figure 3: The input graph vs. the graph with the edges reversed

- TarjanSCC: This algorithm is used to create the strongly connected components with the following steps:

1) An adjacency list is built to store the input graph for the traversal.
2) The discovery IDs and low-link values for nodes are initialized. A discovery ID is the order that the node is visited in, and the low-link value is the smallest discovery time reachable from that node. These are calculated when doing a DFS and are kept track of using a stack.
3) For all unvisited nodes, their discovery ID is set to -1. Once they are visited, each node is assigned a discovery ID using a mutable variable so that they are unique. This is done using the Tarjandfs algorithm which visits each node and forms the components based on their discovery ID's and low link values.
4) The Strongly Connected components (SCCs) are formed by grouping nodes with matching discovery IDs and low-link values. They are stored in an array.
5) Once the traversal is finished and these components have been formed, they are returned.

The runtime of this algorithm is O (|V| + |E|) which is linear. Here, |V| represents the number of vertices in the graph and |E| represents the number of edges in the graph that we are visiting and traversing.

- Tarjandfs: This recursive algorithm is a helper function used to traverse the graph and form the components using the following steps:
    1) For the base case, the discovery ID and low link values are set for the current node
    2) Then, all neighbours of the current node are visited recursively.
    3) If a neighbour has not been visited yet, its low-link value is updated, is recursively visited, and pushed onto the stack. Otherwise, its low-link value is updated based on its discovery ID.
    4) After all the nodes that belong to a strongly connected component have been identified, they are all are popped off the stack until reaching the root node of the scc.
    5) Then, the strongly connected component that has been formed is added to the list of strongly connected components.

Conclusion and limitations:

To conclude, the purpose of this program is to ensure graph connectivity for undirected and directed graphs. The depth first search algorithm and the constraints that every pair of nodes that shares and edge must be connected directly, transitive closure, and mutual reachability were implemented for the undirected graphs. For the directed graphs, the DFS algorithm on the given and reverse graph were implemented along with Tarjan's algorithm for creating strongly connected components. These were implemented in a robust and effective manner. While this program works accurately, there are limitations with the implementation. For example, the running time of depth first search and Tarjan are linear. This is suitable for smaller graphs but for graphs with hundreds of nodes and vertices, the time is inefficient. Moreover, the program relies on knowing the type of graph to run the algorithms since without it, it is unable to perform. Creating a component in the program that detects the type of graph and then runs the algorithms would make the program more versatile. For the future, these limitations will be taken into consideration to create a more robust and advanced program.

Division of Work:

Sahba Hajihoseini: Coding the undirected graphs and generating test cases. 33.3% of the workload

Layan Barrieshee: Writing the report + drawings and generating test cases. 33.3% of the workload

Liza Awwad: Coding the directed graphs and generating test cases. 33.3% of the workload

References:

Used to create and explain the algorithms for the undirected graphs:

https://www.geeksforgeeks.org/connected-components-in-an-undirected-graph/

Used to create and explain the algorithms for the directed graphs:

https://www.geeksforgeeks.org/strongly-connected-components/

https://github.com/krlvi/cyclist/blob/master/java/com/videlov/cyclist/graph/StronglyConnectedComponents.java

https://github.com/cmatuszak/StronglyConnectedComponents/blob/master/src/Graph.java