Finding Simple Cycles in a Directed Graph using Prolog

Cirilli Davide[1] and Fontana Emanuele[2]

[1,2]Department of Computer Science, Università degli Studi di Bari

May 22, 2025

Contents

1 Introduction

A cycle in a directed graph is a path that starts and ends at the same node. An *elementary cycle* is a cycle where no node (except the start/end node) appears more than once. This program aims to find a subset of elementary cycles termed "simple cycles".

A cycle is defined as *simple* if, for any two distinct nodes u and v within the cycle, the shortest path from u to v in the *entire graph* is the path that follows the edges of the cycle itself. If a shorter path (a "shortcut" or "chord") exists between u and v using edges outside the cycle, the cycle is not considered simple.

This document describes our Prolog program designed to identify all such "simple cycles" within a directed graph. The program features an interactive main predicate that allows selection from predefined test cases. For a chosen graph, it first finds all elementary cycles and then filters them based on the specific shortest path criterion described above to determine simplicity. The implementation utilizes Depth-First Search (DFS) for cycle detection and Breadth-First Search (BFS) for shortest path calculations.

This program implements this definition using Prolog, leveraging its backtracking capabilities for graph traversal and dynamic fact manipulation for setting up different graph structures.

2 Implementation Details

The Prolog program (`simpleCycle.pl`) begins with directives:

```
:- dynamic node/2.
:- dynamic arc/4.
:- dynamic edge/2.
```

The `:- dynamic Predicate/Arity` directive declares that facts for `node/2`, `arc/4`, and the helper `edge/2` can be added (`assertz`) or removed (`retractall`) during program execution. This is crucial for the `setup_test_graph/1` predicate, which defines different graph structures, and for `generate_edges_from_arcs/0`, which dynamically creates `edge/2` facts. The program consists of several key components:

2.1 Graph Representation and Setup

The graph is primarily defined by `arc/4` facts, which are dynamically asserted by test case setup predicates.

- `node(NodeID, Type)`: Declares a node with a unique ID and an associated type. While test cases primarily define connectivity through `arc/4` facts, default `node/2` facts are provided in the code. These are used by `find_all_elementary_cycles/1` to gather an initial list of all potential starting nodes for cycle detection.

- `arc(ArcID, Type, SourceNode, TargetNode)`: Declares an arc with a unique ID, type, source node, and target node. These facts define the graph's structure for a given test case.

- `setup_test_graph/1`: A predicate responsible for configuring the graph for a chosen test case (1, 2, 3, or 4). It first calls `retractall(arc(_, _, _, _))` to clear any existing arc definitions and then asserts the specific `arc/4` facts for the selected test case.

For traversal efficiency, a helper predicate `edge(Source, TargetNode)` is dynamically generated from the current `arc/4` facts.

- `generate_edges_from_arcs/0`: This predicate prepares the graph for traversal.
  - It first calls `retractall(edge(_, _))` to remove any existing `edge/2` facts, ensuring a clean state corresponding to the current set of `arc/4` facts.
  - Then, `forall(arc(_, _, SourceNode, DestinationNode), assertz(edge(Source DestinationNode)))` iterates through all current `arc/4` facts. For each, it extracts the source (`SourceNode`) and target (`DestinationNode`) nodes and asserts a new fact `edge(SourceNode, DestinationNode)`. This provides faster lookups for direct connections.

2.2 Finding Elementary Cycles

Elementary cycles are found using a Depth-First Search (DFS) approach:

- `find_all_elementary_cycles/1`: The main predicate for this stage.
  - It uses `findall(NodeID, node(NodeID, _NodeType), Nodes)` to collect all `NodeID`s from the currently asserted `node/2` facts into the list `Nodes`. These nodes serve as potential starting points for cycles.
  - It then calls `find_cycles_from_potential_starts/2` with this list.

  - The argument `ElementaryCyclesList` will be unified with the list of all elementary cycles found.

- `find_cycles_from_potential_starts/2`: Iterates through all nodes from the `Nodes` list and initiates DFS from each.
  - It uses `findall/3`. The template is `RawCycle`.
  - The goal is `(member(StartNode, PotentialStartNodes), edge(StartNode, FirstNeighbor), dfs_for_cycle(FirstNeighbor, StartNode, [FirstNeighbor, StartNode], RawCycle))`.
  - `member(StartNode, PotentialStartNodes)` iterates through each node as a potential starting point.
  - `edge(StartNode, FirstNeighbor)` finds a node directly reachable from `StartNode`.
  - `dfs_for_cycle/4` is called to perform DFS starting from this `Neighbor`, aiming to return to `StartNode`. The path is initialized with `[Neighbor, StartNode]`.
  - `findall/3` collects all `RawCycle` bindings found through backtracking.

- `dfs_for_cycle/4`: Performs the recursive DFS: `dfs_for_cycle(CurrentNode, TargetNode, PathBackToStart, FoundCycleInReverse)`.
  - It looks for an edge from `CurrentNode` to `NextNode` using `edge(CurrentNode, NextNode)`.
  - Base Case: If `NextNode == TargetNode`, the starting node is reached. `FoundCycleInReverse` is unified with `[TargetNode | PathBackToStart]`.
  - Recursive Step: If `NextNode` is not `TargetNode`, it checks `\+ memberchk(NextNode, PathBackToStart)` to ensure elementarity. If not visited in the current path, it recursively calls `dfs_for_cycle(NextNode, TargetNode, [NextNode | PathBackToStart], FoundCycleInReverse)`.

- Cycles from DFS are returned in reverse traversal order (e.g., `[a, d, c, b, a]` for a cycle $a \to b \to c \to d \to a$).

2.3 Filtering for Simple Cycles

The logic for identifying simple cycles:

- `filter_to_simple_cycles/2`: Takes `ElementaryCyclesRaw` and returns `NormalizedSimpleCyclesCandidates`.
  - Base Case: `filter_to_simple_cycles([], [])`.
  - Recursive Step 1 (Simple Cycle Found): If `is_simple_cycle_candidate(CandidateCy` succeeds for the head `Cycle`, it's cut (`!`), normalized via `normalize_cycle_representa` `NormalizedCycle)`, and prepended to the result of recursively processing `RestCandidates`.
  - Recursive Step 2 (Not Simple): If `is_simple_cycle_candidate(CandidateCycle)` fails, the `_Cycle` is ignored, and the predicate recurses on `RestCandidates`.
- `is_simple_cycle_candidate/1`: Checks if an elementary cycle is simple.
  - Reverses the DFS cycle: `reverse(ReversedCycleFromDFS, ForwardCycleWithRepea`
  - Deconstructs to get ordered unique nodes: `ForwardCycleWithRepeatEnd = [StartNode | PathNodesWithRepeatEnd]`, `append(PathNodesUnique, [StartNode], PathNodesWithRepeatEnd)`, `NodesInCycleOrdered = [StartNode | PathNodesUnique]`. E.g., for DFS output `[a,c,b,a]`, this yields `[a,b,c]`.
  - Calls `check_all_node_pairs_for_chords(CycleNodesInOrder, CycleNodesInOrder)`.
- `check_all_node_pairs_for_chords/2`: Iterates through ordered pairs (u, v) in `CycleNodesInOrder`.
  - Base Case: `check_all_node_pairs_for_chords([], _)`.
  - Recursive Step: Takes `Node1` from the list, calls `check_one_node_against_all_others(Node1, OriginalCycleNodesOrdered, OriginalCycleNodesOrdered)`, then recurses on `RestN1`.
- `check_one_node_against_all_others/3`: For a given `Node1`, iterates through other nodes `Node2` in `OriginalCycleNodesOrdered`.
  - Base Case: `check_one_node_against_all_others(_, [], _)`.
  - Recursive Step: Takes `Node2`.
  - If `Node1 == Node2`, skip (`true`).
  - Else, calculate `get_distance_along_cycle(Node1, Node2, OriginalCycleNodesOr` `CycleDist)` and `find_shortest_path_length(Node1, Node2, GraphShortestDistance)`.

    - Check simplicity:
        * If `GraphShortestDistance == -1` (no path in graph), then $CycleDist > 0$ must hold (or `!, fail`).
        * Else (path exists), $GraphShortestDistance \geq CycleDist$ must hold.
    - If the check succeeds, cut (`!`) and recurse on `RestN2`.
    - Failure Clause: `check_one_node_against_all_others(_, _, _) :- !, fail.` ensures immediate failure if any pair violates simplicity.

- `get_distance_along_cycle/4`: Calculates distance from `NodeA` to `NodeB` along `CycleNodesOrdered`.
    - Finds indices `IndexA`, `IndexB` using `nth0/3`. Gets `length(CycleNodesOrdered, PathLength)`.
    - If $IndexB \geq IndexA$, $Distance = IndexB - IndexA$.
    - Else, $Distance = PathLength - IndexA + IndexB$.

- `find_shortest_path_length/3`: Finds shortest path length between `StartNode` and `EndNode` using BFS.
    - Calls `bfs_shortest_path([[StartNode, 0]], EndNode, [StartNode], Length)`.
    - Cuts (`!`) on success. If BFS fails, the second clause `find_shortest_path_length(_, _, -1)` sets `Length` to -1.

- `bfs_shortest_path/4`: Standard BFS: `bfs_shortest_path(Queue, TargetNode, VisitedNodes, PathLength)`.
    - Base Case 1 (Queue Empty): `bfs_shortest_path([], _, _, _) :- !, fail.`
    - Base Case 2 (TargetNode Found): `bfs_shortest_path([[TargetNode, PathLength] | _], TargetNode, _, PathLength) :- !.`
    - Recursive Step: Dequeues `[CurrentNode, CurrentDist]`. Finds unvisited neighbors via `findall(NextNode, (edge(CurrentNode, NextNode), \+ member(NextNode, VisitedNodes)), UnvisitedNeighbors)`. Calculates $NewDistToNeighbors = CurrentDist + 1$. Adds neighbors to queue via `add_unvisited_neighbors_to_queue/4`. Updates visited list (using `append/3` and `list_to_set/2`). Recurses.

- add_unvisited_neighbors_to_queue/4: Formats neighbors as [Node, Distance] and appends to queue using findall/3 and append/3.

2.4 Cycle Normalization

Ensures unique representation for identical cycles starting at different nodes.

- normalize_cycle_representation/2: Converts raw DFS cycle RawReversedCycle (e.g., [a,d,c,b,a]) to NormalizedNodeList.
- Process:
  1. Deconstruct: RawReversedCycle = [StartNode | PathReversedWithStartNodeAtE
  2. Get forward path nodes: reverse(PathReversedWithStartNodeAtEnd, [_StartNodeAgain | PathForwardWithoutStartNode]).
  3. Reconstruct forward cycle nodes: CycleNodesInForwardOrder = [StartNode | PathForwardWithoutStartNode] (e.g., [a,b,c,d]).
  4. Find minimum node: find_lexicographically_smallest_node(CycleNodesInForw SmallestNode).
  5. Rotate: rotate_list_to_start_with_element(CycleNodesInForwardOrder, SmallestNode, NormalizedNodeList).
- find_lexicographically_smallest_node/2: Finds minimum node in a list using @<.
  - Base Case: find_lexicographically_smallest_node([Min], Min) :- !.
  - Recursive Step: Compares head Head with minimum of tail MinOfTail.
- rotate_list_to_start_with_element/3: Rotates OriginalList so StartElement is first.
  - Uses append(BeforeElement, [StartElement | AfterElement], OriginalList), !, append([StartElement | AfterElement], BeforeElement, RotatedList).
  - Fallback clause: rotate_list_to_start_with_element(OriginalList, _, OriginalList) if element already first or not found (latter shouldn't occur in this program's logic).

The final list of simple cycles is produced using setof/3 on the normalized cycles to ensure uniqueness and canonical order.

2.5 Main Predicate and Output Control

- `main/0`: The primary entry point for execution.

  1. Prompts the user to select a test case (1-4) using `write/1` and `read/1`.
  2. Calls `setup_test_graph(testCaseNumber)` with the chosen number `N`.
  3. If `setup_test_graph_graph/1` succeeds, it calls `find_simple_cycles(SimpleCycles`
  4. Prints the resulting `SimpleCyclesList` list.
  5. If an invalid test case number is entered, an error message is shown.
  6. Calls `halt/0` to terminate the Prolog session.

- `find_simple_cycles/1`: Orchestrates the cycle finding and filtering process for the currently loaded graph.

  1. Calls `generate_edges_from_arcs/0`.
  2. Calls `find_all_elementary_cycles/1` to get `ElementaryCyclesRaw`.
  3. Prints the count and list of elementary cycles (using `print_cycles_list_reversed/2` as DFS cycles are reversed).
  4. If elementary cycles exist:
     - Calls `filter_to_simple_cycles/2` to get `NormalizedSimpleCyclesCandidates.`
     - Uses `setof(NormCycle, Member^(member(Member, NormalizedSimpleCyclesC` `NormCycle = Member), SimpleCyclesList)` to get the final unique, sorted list.
     - Prints the simple cycles (using `print_cycles_list/2`) and their count.
     - If `setof/3` fails (no simple cycles), prints a message and sets `SimpleCyclesList` to `[]`.
  5. If no elementary cycles, prints a message and sets `SimpleCyclesList` to `[]`.
  6. The argument `SimpleCyclesList` is unified with the final list.

- Helper Printing Predicates:

  - `print_cycles_list/2`: `print_cycles_list(Header, ListOfCycles)`. Prints a header and then each cycle in the list, indented.

- `print_cycles_list_reversed/2`: Similar to `print_cycles_list/2`, but it first reverses each cycle in the list before printing. This is used for displaying elementary cycles as found by DFS in their natural traversal order.

3 Test Cases and Usage

3.1 Usage

Ensure a Prolog interpreter (e.g., SWI-Prolog) is installed. Load the program file: `?- [simpleCycle].` (or your filename). Run the main interactive predicate: `?- main.` The program will prompt: `Select test case (1, 2, 3, or 4):` Enter a number from 1 to 4 and press Enter. The program will then execute for the chosen test case, printing intermediate elementary cycles and the final list of unique, normalized simple cycles. The output will also show `SimpleCycles = [[...], ...]`. The program will then halt.

3.2 Predefined Test Cases

The program includes four predefined test cases, set up by `setup_test_graph/1`.

3.2.1 Test Case 1: Simple Triangle

- **Description:** A basic directed triangle: $a \to b \to c \to a$.
- **Arcs Defined:**

```
assertz(arc(t1_ab, type_edge, a, b)).
assertz(arc(t1_bc, type_edge, b, c)).
assertz(arc(t1_ca, type_edge, c, a)).
```

- **Graph Visualization:**

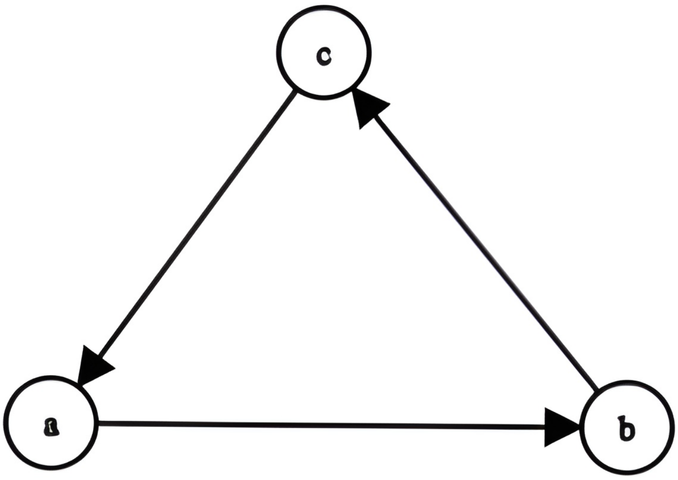

Figure 1: Graphical representation of Test Case 1.

- **Expected Simple Cycles:** `[[a,b,c]]`

3.2.2 Test Case 2: Square with a Chord

- **Description:** A square cycle $a \to b \to c \to d \to a$, with an additional "chord" arc $b \to d$.
- **Arcs Defined:**

```
assertz(arc(t2_ab, type_edge, a, b)).
assertz(arc(t2_bc, type_edge, b, c)).
assertz(arc(t2_cd, type_edge, c, d)).
assertz(arc(t2_da, type_edge, d, a)).
assertz(arc(t2_bd_chord, type_edge, b, d)). % The chord
```

- **Graph Visualization:**

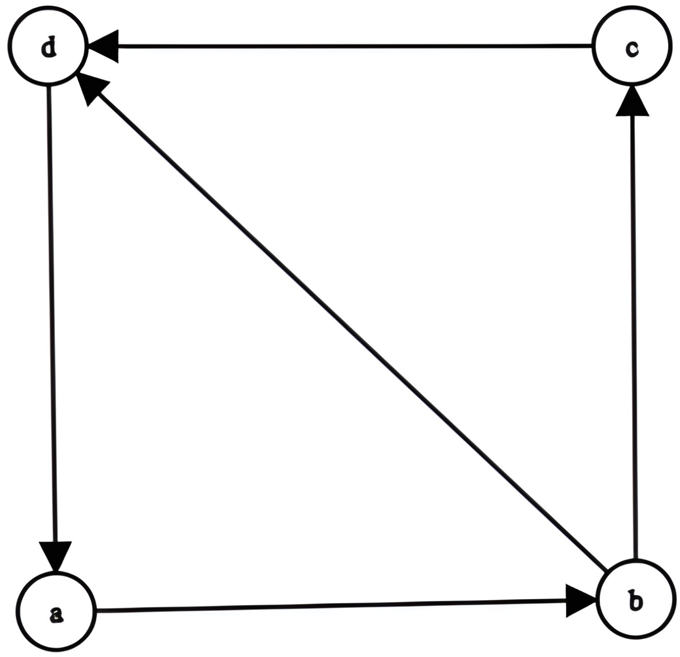

Figure 2: Graphical representation of Test Case 2.

- **Expected Simple Cycles:** `[[a,b,d], [b,c,d]]`. The cycle `[a,b,c,d]` (from $a \to b \to c \to d \to a$) is elementary but not simple due to the shortcut $b \to d$ (path $b \to c \to d$ has length 2, path $b \to d$ has length 1).

3.2.3 Test Case 3: Disjoint Cycles

- **Description:** A graph containing a 2-cycle ($a \leftrightarrow b$) and two separate, non-overlapping triangles ($c \to d \to e \to c$ and $d \to f \to g \to d$).

- **Arcs Defined:**

```
assertz(arc(t3_ab, type_edge, a, b)). % 2-cycle
assertz(arc(t3_ba, type_edge, b, a)).
assertz(arc(t3_cd, type_edge, c, d)). % Triangle 1
assertz(arc(t3_de, type_edge, d, e)).
assertz(arc(t3_ec, type_edge, e, c)).
assertz(arc(t3_df, type_edge, d, f)). % Triangle 2
assertz(arc(t3_fg, type_edge, f, g)).
assertz(arc(t3_gd, type_edge, g, d)).
```

- **Graph Visualization:**

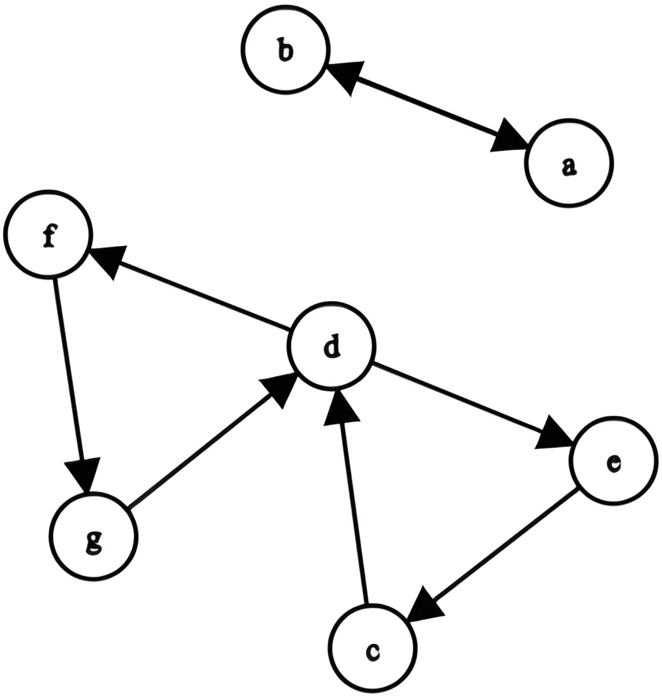

Figure 3: Graphical representation of Test Case 3.

- **Expected Simple Cycles:** `[[a,b], [c,d,e], [d,f,g]]` (order by `setof` might vary but content is these three).

3.2.4 Test Case 4: Complex Overlapping Cycles and Chord

- **Description:** A more complex graph with two larger, overlapping cycles and a shortcut arc. Cycle A: $a \to b \to c \to d \to a$. Cycle B: $b \to e \to d \to c \to b$. Chord: $a \to d$. This also creates smaller 2-cycles like $b \leftrightarrow c$ and $c \leftrightarrow d$ due to edges from both Cycle A and Cycle B.

- **Arcs Defined:**

```
assertz(arc(t4_ab, type_edge, a, b)).
assertz(arc(t4_bc, type_edge, b, c)).
assertz(arc(t4_cd, type_edge, c, d)).
assertz(arc(t4_da, type_edge, d, a)).
assertz(arc(t4_be, type_edge, b, e)).
assertz(arc(t4_ed, type_edge, e, d)).
```

```
assertz(arc(t4_dc, type_edge, d, c)). % Edge for Cycle
    B, opposite of c->d in Cycle A
assertz(arc(t4_cb, type_edge, c, b)). % Edge for Cycle
    B, opposite of b->c in Cycle A
assertz(arc(t4_ad_chord, type_edge, a, d)). % Chord for
    Cycle A
```

- **Graph Visualization:**

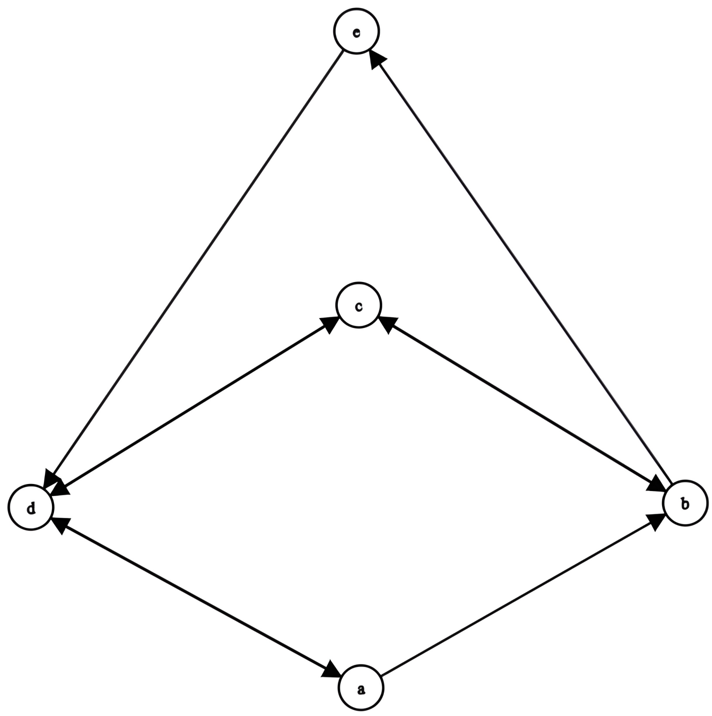

Figure 4: Graphical representation of Test Case 4.

- **Expected Simple Cycles:** `[[a,d], [b,c], [c,d]]`.
  - `[a,b,c,d]` (from $a \to b \to c \to d \to a$) is not simple because path $a \to b \to c \to d$ (length 3) is longer than direct shortcut $a \to d$ (length 1).
  - `[b,e,d,c]` (from $b \to e \to d \to c \to b$) is not simple because path $b \to e \to d \to c$ (length 3) is longer than direct shortcut $b \to c$ (length 1, via arc 2).
  - The 2-cycles `[a,d]` ($a \to d$, $d \to a$), `[b,c]` ($b \to c$, $c \to b$), and `[c,d]` ($c \to d$, $d \to c$) are simple as their constituent 1-edge paths are inherently the shortest.

4 Implementation for Undirected Graphs

In addition to the algorithm for directed graphs described in the previous sections, a variant for undirected graphs has been developed in the file `simpleCycleUndirected.pl`. The main differences between the two implementations are described below:

4.1 Edge Generation in Undirected Graphs

The main difference in the implementation for undirected graphs lies in the generation of `edge/2` facts from the arcs defined via `arc/4`:

- `generate_edges_from_arcs/0`: While in the directed version each arc `arc(ID, Type, Source, Destination)` generates a single fact `edge(Source, Destination)`, in the undirected version each arc generates two facts: `edge(Source, Destination)` and `edge(Destination, Source)`, ensuring duplicates are avoided:

```
generate_edges_from_arcs :-
    retractall(edge(_, _)),
    forall(arc(_ArcID, _ArcType, SourceNode,
        DestinationNode),
           (   (   \+ edge(SourceNode, DestinationNode)
               ->  assertz(edge(SourceNode,
                  DestinationNode))
               ;   true
               ),
               (   \+ edge(DestinationNode, SourceNode)
               ->  assertz(edge(DestinationNode,
                  SourceNode))
               ;   true
               )
           )).
```

4.2 Finding Elementary Cycles in Undirected Graphs

The DFS procedure has been modified to avoid spurious cycles that can arise from immediately returning to the previous node in an undirected graph:

- `dfs_for_cycle/4`: In the version for undirected graphs, the predicate keeps track of the immediate predecessor in the path and ensures that the next node to visit is not this predecessor:

```
dfs_for_cycle(CurrentNode, TargetNode, PathBackToStart,
   FoundCycleInReverse) :-
    PathBackToStart = [CurrentNode, Prev | _],  %
       Extracts the immediate predecessor
    edge(CurrentNode, NextNode),                %
       Explores an edge from CurrentNode to NextNode
    (   % If NextNode is the target and not the
       immediate predecessor, a cycle is found
        NextNode == TargetNode,
        NextNode \== Prev
    ->  FoundCycleInReverse = [TargetNode |
       PathBackToStart]
    ;   % Otherwise, if NextNode is not the target,
       ensure it's not the immediate predecessor
        NextNode \== Prev,
        \+ memberchk(NextNode, PathBackToStart)  %
           Ensures NextNode is not already in the path
    ->  dfs_for_cycle(NextNode, TargetNode, [NextNode |
       PathBackToStart], FoundCycleInReverse)
    ).
```

This contrasts with the version for directed graphs, where it is not necessary to explicitly exclude the predecessor, because if an arc exists from A to B, an arc from B to A does not necessarily exist:

```
dfs_for_cycle(CurrentNode, TargetNode, PathBackToStart,
   FoundCycleInReverse) :-
    edge(CurrentNode, NextNode),
    (   NextNode == TargetNode ->
        FoundCycleInReverse = [TargetNode |
           PathBackToStart]
    ;   \+ memberchk(NextNode, PathBackToStart),
        dfs_for_cycle(NextNode, TargetNode, [NextNode |
           PathBackToStart], FoundCycleInReverse)
    ).
```

4.3 Chord Detection in Undirected Graphs

The version for undirected graphs must handle chord checking in a particular way to avoid false positives:

- `check_one_node_against_all_others/3`: In the undirected version, the check skips pairs of adjacent nodes in the cycle (in both directions) to avoid confusing the cycle's own edges with possible chords:

```
check_one_node_against_all_others(Node1, [Node2 | Rest],
    CycleNodesOrdered) :-
    (   Node1 == Node2
    ;   (get_distance_along_cycle(Node1, Node2,
        CycleNodesOrdered, D1), D1 =:= 1)
    ;   (get_distance_along_cycle(Node2, Node1,
        CycleNodesOrdered, D2), D2 =:= 1)
    )
    ->  true  % trivial: same node or adjacent in the
        undirected cycle
    ;   % Not adjacent: ensure no shorter path exists in
        the graph (no chords)
        get_distance_along_cycle(Node1, Node2,
            CycleNodesOrdered, CyclePathDistance),
        find_shortest_path_length(Node1, Node2,
            GraphShortestDistance),
        (   GraphShortestDistance == -1
        ->  CyclePathDistance > 0  % disconnected
            outside the cycle
        ;   GraphShortestDistance >= CyclePathDistance
        )
    ,
    !,
    check_one_node_against_all_others(Node1, Rest,
        CycleNodesOrdered).
```

4.4 Cycle Normalization for Undirected Graphs

A cycle in an undirected graph can be traversed in both directions. The undirected version normalizes cycles by considering both possible directions:

- `normalize_cycle_representation_representation/2`: Generates two canonical representations (one for each traversal direction) and chooses the lexicographically smaller one as the canonical representation:

```
normalize_cycle_representation_representation(RawReversedCycle,
    NormalizedNodeList) :-
    % Step 1: Convert DFS output to an ordered list of
        unique nodes (CycleForward)
    RawReversedCycle = [StartNode | PathRevEnd],
    reverse(PathRevEnd, [_StartAgain | PathForwardREST]),
    CycleForward = [StartNode | PathForwardREST],

    % Step 2: Normalize CycleForward starting from the
        lexicographically smallest node
    find_lexicographically_smallest_node(CycleForward,
        SmallestF),
```

```
    rotate_list_to_start_with_element(CycleForward,
        SmallestF, RotFwd),

    % Step 3: Generate and normalize the cycle in the
        opposite direction
    reverse(CycleForward, CycleBackwardTemp),
    find_lexicographically_smallest_node(CycleBackwardTemp,
        SmallestB),
    rotate_list_to_start_with_element(CycleBackwardTemp,
        SmallestB, RotBwd),

    % Step 4: Choose the lexicographically smaller list
        as the canonical representation
    (   RotFwd @=< RotBwd
    ->  NormalizedNodeList = RotFwd
    ;   NormalizedNodeList = RotBwd
    ).
```

This improved normalization ensures that equivalent cycles (traversed in different directions) are represented consistently, facilitating the identification of unique cycles in undirected graphs.

5 Conclusion

The Prolog program successfully implements an algorithm to find simple cycles in both directed and undirected graphs, using the approach described in the previous sections. The version for undirected graphs (`simpleCycleUndirected.pl`) extends the base implementation (`simpleCycle.pl`) with specialized modifications to handle the bidirectionality of edges. Both implementations demonstrate the effective use of DFS for cycle detection, BFS for calculating shortest paths, and Prolog's capabilities for dynamic fact and list manipulation. Cycle normalization ensures that equivalent cycles are represented consistently, and `setof/3` provides a final, ordered list of these unique simple cycles.