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 20, 2025

Abstract

This document describes a Prolog program designed to identify all "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 a specific shortest path criterion to determine simplicity. The implementation utilizes Depth-First Search (DFS) for cycle detection and Breadth-First Search (BFS) for shortest path calculations.

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") exists between u and v using edges outside the cycle, the cycle is not considered simple. 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/1` predicate, which defines different graph structures, and for `generate_edges/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 a directed 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/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, Target)` is dynamically generated from the current `arc/4` facts.

- `generate_edges/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(_, _, N, M), assertz(edge(N, M)))` iterates through all current `arc/4` facts. For each, it extracts the source (`N`) and target (`M`) nodes and asserts a new fact `edge(N, M)`. 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(N, node(N, _), 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_starting_from_nodes/2` with this list.

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

- `find_cycles_starting_from_nodes/2`: Iterates through all nodes from the `Nodes` list and initiates DFS from each.
  - It uses `findall/3`. The template is `Cycle`.
  - The goal is `(member(StartNode, Nodes), edge(StartNode, Neighbor), dfs_find_cycle(Neighbor, StartNode, [Neighbor, StartNode], Cycle))`.
  - `member(StartNode, Nodes)` iterates through each node as a potential starting point.
  - `edge(StartNode, Neighbor)` finds a node directly reachable from `StartNode`.
  - `dfs_find_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 `Cycle` bindings found through backtracking.

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

- 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_simple_cycles/2`: Takes `ElementaryCycles` and returns `NormalizedSimpleCycle`

  - Base Case: `filter_simple_cycles([], []).`
  - Recursive Step 1 (Simple Cycle Found): If `is_simple_cycle(Cycle)` succeeds for the head `Cycle`, it's cut (`!`), normalized via `normalize_cycle(Cycle, NormalizedCycle)`, and prepended to the result of recursively processing `RestCandidates`.
  - Recursive Step 2 (Not Simple): If `is_simple_cycle(Cycle)` fails, the `_Cycle` is ignored, and the predicate recurses on `RestCandidates`.

- `is_simple_cycle/1`: Checks if an elementary cycle is simple.
  - Reverses the DFS cycle: `reverse(Cycle, ForwardCycleWithRepeatEnd)`.
  - 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_pairs_shortest_path(NodesInCycleOrdered, NodesInCycleOrdered)`.

- `check_all_pairs_shortest_path/2`: Iterates through ordered pairs (u, v) in `NodesInCycleOrdered`.
  - Base Case: `check_all_pairs_shortest_path([], _)`.
  - Recursive Step: Takes `N1` from the list, calls `check_pairs_from_node(N1, OriginalCycleNodes, OriginalCycleNodes)`, then recurses on `RestN1`.

- `check_pairs_from_node/3`: For a given `N1`, iterates through other nodes `N2` in `OriginalCycleNodes`.
  - Base Case: `check_pairs_from_node(_, [], _)`.
  - Recursive Step: Takes `N2`.
  - If `N1 == N2`, skip (`true`).
  - Else, calculate `get_cycle_distance(N1, N2, OriginalCycleNodes, CycleDist)` and `shortest_path_length(N1, N2, ShortestDist)`.
  - Check simplicity:
    - If `ShortestDist == -1` (no path in graph), then $CycleDist > 0$ must hold (or `!, fail`).
    - Else (path exists), $ShortestDist \geq CycleDist$ must hold.

- If the check succeeds, cut (`!`) and recurse on `RestN2`.
- Failure Clause: `check_pairs_from_node(_, _, _) :- !, fail.` ensures immediate failure if any pair violates simplicity.

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

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

- `bfs/4`: Standard BFS: `bfs(Queue, Target, Visited, Length)`.
  - Base Case 1 (Queue Empty): `bfs([], _, _, _) :- !, fail.`
  - Base Case 2 (Target Found): `bfs([[Target, Length] | _], Target, _, Length) :- !.`
  - Recursive Step: Dequeues `[Current, Dist]`. Finds unvisited neighbors via `findall(Next, (edge(Current, Next), \+ member(Next, Visited)), Neighbors)`. Calculates $NewDist = Dist + 1$. Adds neighbors to queue via `add_neighbors_to_queue/4`. Updates visited list (using `append/3` and `list_to_set/2`). Recurses.

- `add_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/2`: Converts raw DFS cycle `RawCycle` (e.g., `[a,d,c,b,a]`) to `NormalizedNodeList`.

- Process:

1. Deconstruct: `RawCycle = [StartNode | ReversedPathWithStart]`.
2. Get forward path nodes: `reverse(ReversedPathWithStart, [_ | ForwardPathNodes])`.
3. Reconstruct forward cycle nodes: `ForwardCycleNodes = [StartNode | ForwardPathNodes]` (e.g., `[a,b,c,d]`).
4. Find minimum node: `find_min_node(ForwardCycleNodes, MinNode)`.
5. Rotate: `rotate_list_to_start_with(ForwardCycleNodes, MinNode, NormalizedNodeList)`.

- `find_min_node/2`: Finds minimum node in a list using `@<`.
  - Base Case: `find_min_node([M], M) :- !.`
  - Recursive Step: Compares head `H` with minimum of tail `MinTail`.
- `rotate_list_to_start_with/3`: Rotates `List` so `Element` is first.
  - Uses `append(BeforeElement, [Element | AfterElement], List), !, append([Element | AfterElement], BeforeElement, RotatedList)`.
  - Fallback clause: `rotate_list_to_start_with(List, _, List)` 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(N)` with the chosen number `N`.
  3. If `setup_test/1` succeeds, it calls `find_simple_cycles(SimpleCycles)`.
  4. Prints the resulting `SimpleCycles` 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/0`.
2. Calls `find_all_elementary_cycles/1` to get `ElementaryCycles`.
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_simple_cycles/2` to get `NormalizedSimpleCycles`.
   - Uses `setof(NormCycle, Member^(member(Member, NormalizedSimpleCycles), NormCycle = Member), SimpleCycles)` 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 `SimpleCycles` to `[]`.
5. If no elementary cycles, prints a message and sets `SimpleCycles` to `[]`.
6. The argument `SimpleCycles` 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/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(1, t, a, b)).
assertz(arc(2, t, b, c)).
assertz(arc(3, t, c, a)).
```

- **Graph Visualization:**

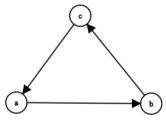

Figure 1: Graphical representation of Test Case 1.

- **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(1, t, a, b)).
assertz(arc(2, t, b, c)).
assertz(arc(3, t, c, d)).
assertz(arc(4, t, d, a)).
assertz(arc(5, t, b, d)). % The chord
```

- **Graph Visualization:**

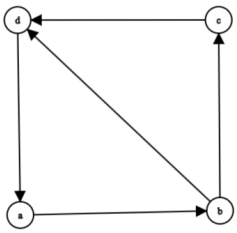

Figure 2: Graphical representation of Test Case 2.

- **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(1, t, a, b)). % 2-cycle
assertz(arc(2, t, b, a)).
assertz(arc(3, t, c, d)). % Triangle 1
assertz(arc(4, t, d, e)).
assertz(arc(5, t, e, c)).
assertz(arc(6, t, d, f)). % Triangle 2
assertz(arc(7, t, f, g)).
assertz(arc(8, t, g, d)).
```

- **Graph Visualization:**

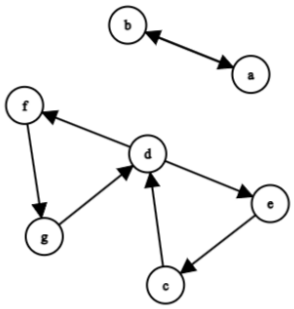

Figure 3: Graphical representation of Test Case 3.

- **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(1, t, a, b)).
assertz(arc(2, t, b, c)).
assertz(arc(3, t, c, d)).
assertz(arc(4, t, d, a)).
assertz(arc(5, t, b, e)).
assertz(arc(6, t, e, d)).
assertz(arc(7, t, d, c)). % Edge for Cycle B, opposite
    of c->d in Cycle A
assertz(arc(8, t, c, b)). % Edge for Cycle B, opposite
    of b->c in Cycle A
```

```
assertz(arc(9, t, a, d)). % Chord for Cycle A
```

- **Graph Visualization:**

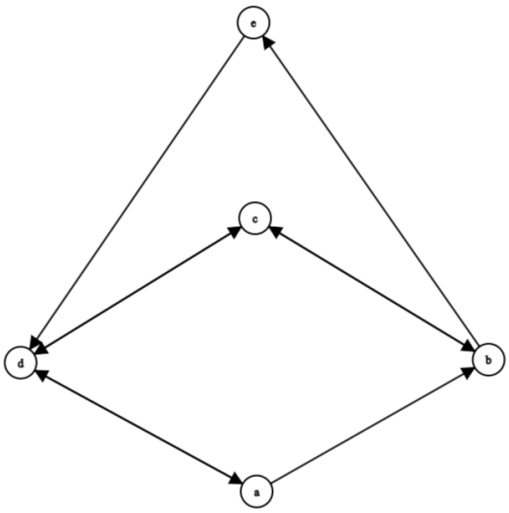

Figure 4: Graphical representation of Test Case 4.

- **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 Implementazione per Grafi non Diretti

Oltre all'algoritmo per grafi diretti descritto nelle sezioni precedenti, è stata sviluppata una variante per grafi non diretti nel file `simpleCycleUndirected.pl`. Di seguito sono descritte le principali differenze tra le due implementazioni:

4.1 Generazione degli Archi nei Grafi non Diretti

La principale differenza nell'implementazione per i grafi non diretti è nella generazione degli archi `edge/2` a partire dagli archi definiti tramite `arc/4`:

- `generate_edges_from_arcs/0`: Mentre nella versione diretta ogni arco `arc(ID, Tipo, Origine, Destinazione)` genera un singolo fatto `edge(Origine, Destinazione)`, nella versione non diretta ogni arco genera due fatti: `edge(Origine, Destinazione)` e `edge(Destinazione, Origine)`, assicurandosi di evitare duplicati:

```
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 Ricerca di Cicli Elementari nei Grafi non Diretti

La procedura DFS è stata modificata per evitare cicli spuri che possono derivare dal tornare immediatamente al nodo precedente in un grafo non diretto:

- `dfs_for_cycle/4`: Nella versione per grafi non diretti, il predicate tiene traccia del predecessore immediato nel percorso e garantisce che il prossimo nodo da visitare non sia tale predecessore:

```
dfs\_for\_cycle(CurrentNode, TargetNode,
   PathBackToStart, FoundCycleInReverse) :-
    PathBackToStart = [CurrentNode, Prev | \_],  %
       Estrae il predecessore immediato
    edge(CurrentNode, NextNode),                 %
       Esplora un arco da CurrentNode a NextNode
    (   % Se NextNode è il target e non è il
       predecessore immediato, un ciclo è trovato
        NextNode == TargetNode,
```

```
        NextNode \== Prev
    ->  FoundCycleInReverse = [TargetNode |
        PathBackToStart]
    ;   % Altrimenti, se NextNode non è il target,
        assicura che non sia il predecessore immediato
        NextNode \== Prev,
        \+ memberchk(NextNode, PathBackToStart)  %
            Assicura che NextNode non sia già nel
            percorso
    ->  dfs\_for\_cycle(NextNode, TargetNode, [NextNode
        | PathBackToStart], FoundCycleInReverse)
    ).
```

Questo contrasta con la versione per grafi diretti, dove non è necessario escludere esplicitamente il predecessore, poiché se esiste un arco da A a B, non necessariamente esiste un arco da B a A:

```
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 Rilevamento delle Corde nei Grafi non Diretti

La versione per grafi non diretti deve gestire in modo particolare il controllo delle corde per evitare falsi positivi:

- `check_one_node_against_all_others/3`: Nella versione non diretta, il controllo salta le coppie di nodi adiacenti nel ciclo (in entrambe le direzioni) per evitare di confondere gli archi del ciclo stesso con possibili corde:

```
check\_one\_node\_against\_all\_others(Node1, [Node2 |
   Rest], CycleNodes) :-
    (   Node1 == Node2
    ;   (get\_distance\_along\_cycle(Node1, Node2,
        CycleNodes, D1), D1 =:= 1)
    ;   (get\_distance\_along\_cycle(Node2, Node1,
        CycleNodes, D2), D2 =:= 1)
    )
```

```
    ->  true  % trivial: stesso nodo o adiacente nel
        ciclo non diretto
    ;   % Non adiacente: assicura che non esista un
        percorso più breve nel grafo (no corde)
        get\_distance\_along\_cycle(Node1, Node2,
            CycleNodes, CyclePathDistance),
        find\_shortest\_path\_length(Node1, Node2,
            GraphShortestDistance),
        (   GraphShortestDistance == -1
        ->  CyclePathDistance > 0  % disconnesso fuori
            dal ciclo
        ;   GraphShortestDistance >= CyclePathDistance
        )
    ,
    !,
    check\_one\_node\_against\_all\_others(Node1, Rest,
        CycleNodes).
```

4.4 Normalizzazione dei Cicli per Grafi non Diretti

Un ciclo in un grafo non diretto può essere percorso in entrambe le direzioni. La versione non diretta normalizza i cicli considerando entrambe le direzioni possibili:

- `normalize_cycle_representation/2`: Genera due rappresentazioni canoniche (una per ciascuna direzione di percorrenza) e sceglie la lessicograficamente minore come rappresentazione canonica:

```
normalize\_cycle\_representation(RawReversedCycle,
    NormalizedNodeList) :-
    % Passo 1: Converte l'output DFS in una lista
        ordinata di nodi unici (CycleForward)
    RawReversedCycle = [StartNode | PathRevEnd],
    reverse(PathRevEnd, [\_StartAgain |
        PathForwardREST]),
    CycleForward = [StartNode | PathForwardREST],

    % Passo 2: Normalizza CycleForward iniziando dal
        nodo lessicograficamente minore
    find\_lexicographically\_smallest\_node(CycleForward,
        SmallestF),
    rotate\_list\_to\_start\_with\_element(CycleForward,
        SmallestF, RotFwd),

    % Passo 3: Genera e normalizza il ciclo nella
        direzione opposta
    reverse(CycleForward, CycleBackwardTemp),
```

```
    find\_lexicographically\_smallest\_node(CycleBackwardTemp,
        SmallestB),
    rotate\_list\_to\_start\_with\_element(CycleBackwardTemp,
        SmallestB, RotBwd),

    % Passo 4: Sceglie la lista lessicograficamente
        minore come rappresentazione canonica
    (  RotFwd @=< RotBwd
    -> NormalizedNodeList = RotFwd
    ;  NormalizedNodeList = RotBwd
    ).
```

Questa normalizzazione migliorata garantisce che i cicli equivalenti (percorsi in direzioni diverse) siano rappresentati in modo coerente, facilitando l'identificazione di cicli unici nei grafi non diretti.

5 Conclusione

Il programma Prolog implementa con successo un algoritmo per trovare i cicli semplici sia in grafi diretti che non diretti, utilizzando l'approccio descritto nelle sezioni precedenti. La versione per grafi non diretti (`simpleCycleUndirected.pl`) estende l'implementazione base (`simpleCycle.pl`) con modifiche specializzate per gestire la bidirezionalità degli archi. Entrambe le implementazioni dimostrano l'efficace uso di DFS per la rilevazione dei cicli, BFS per il calcolo dei percorsi minimi, e le capacità di Prolog per la manipolazione dinamica dei fatti e delle liste. La normalizzazione dei cicli assicura che i cicli equivalenti siano rappresentati in modo coerente e `setof/3` fornisce un elenco finale e ordinato di questi cicli semplici unici.