Finding Simple Cycles in a Graph using Prolog

Cirilli Davide[1] and Fontana Emanuele[2]

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

May 5, 2025

Abstract

This document describes a Prolog program designed to identify all "simple cycles" within a directed graph. The program 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.

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 the facts for `node/2`, `arc/4`, and the helper `edge/2` can be added (`assertz`) or removed (`retractall`) during the program's execution. This is necessary because the graph data might be loaded or modified, and the `edge/2` facts are generated dynamically for efficiency.

The program consists of several key components:

2.1 Graph Representation

The graph is defined using dynamic facts:

- `node(NodeID, Type)`: Declares a node with a unique ID and an associated type. The type is not used in the cycle finding logic itself but is part of the data structure.

- `arc(ArcID, Type, SourceNode, TargetNode)`: Declares a directed arc with a unique ID, type, source node, and target node.

For efficiency, a helper predicate `edge(Source, Target)` is dynamically generated.

- `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. The underscores `_` are anonymous variables, matching any term.
  - Then, `forall(arc(_, _, N, M), assertz(edge(N, M)))` iterates through all existing `arc/4` facts. For each `arc` fact, it extracts the source (`N`) and target (`M`) nodes and asserts a new fact `edge(N, M)` into the Prolog database using `assertz/1` (which adds the fact at the end). This provides faster lookups for direct connections during graph traversal compared to querying the `arc/4` facts repeatedly.

2.2 Finding Elementary Cycles

Elementary cycles are found using a Depth-First Search (DFS) approach implemented by the predicates:

- `find_all_elementary_cycles/1`: The main predicate for this stage.

  - It uses `findall(N, node(N, _), Nodes)` to collect all `NodeIDs` from the `node/2` facts into the list `Nodes`.
  - 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 and initiates DFS from each.

  - It uses `findall/3` again. 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 in the graph as a potential starting point.
  - `edge(StartNode, Neighbor)` finds a node directly reachable from the `StartNode`.
  - `dfs_find_cycle/4` is then called to perform the DFS starting from this `Neighbor`, aiming to return to the `StartNode`.

- `findall` collects all successful `Cycle` bindings found through backtracking into the final list `Cycles`.

- `dfs_find_cycle/4`: Performs the recursive DFS. Its arguments are `dfs_find_cycle(CurrentNode, TargetNode, PathSoFar, Cycle)`.
  - It looks for an edge from the `CurrentNode` to a `NextNode` using `edge(CurrentNode, NextNode)`.
  - **Base Case:** If `NextNode == TargetNode`, the starting node has been reached again. The cycle is complete. `Cycle` is unified with the path list, prepending the `TargetNode` (e.g., `[TargetNode | PathSoFar]`). The `==/2` operator checks for literal equality.
  - **Recursive Step:** If `NextNode` is not the `TargetNode`, it checks if `NextNode` is already in the `PathSoFar` using
    `+ memberchk(NextNode, PathSoFar)`.
    `+/1` is the negation operator (logical NOT), and `memberchk/2` efficiently checks for membership without leaving a choice point. If the node is not already visited in the current path (ensuring elementarity), the predicate calls itself recursively: `dfs_find_cycle(NextNode, TargetNode, [NextNode | PathSoFar], Cycle)`. The `NextNode` is added to the front of the path list.

- The cycles found by DFS are returned in reverse order of traversal (e.g., `[a, d, c, b, a]` for a cycle a $\rightarrow$ b $\rightarrow$ c $\rightarrow$ d $\rightarrow$ a) because nodes are prepended to the path list during recursion.

2.3 Filtering for Simple Cycles

The core logic for identifying simple cycles resides in:

- `filter_simple_cycles/2`: Takes a list of elementary cycles (`ElementaryCycles`) and returns only those that satisfy the simplicity condition (`NormalizedSimpleCycles`).
  - **Base Case:** If the input list is empty (`[]`), the result is also an empty list.
  - **Recursive Step 1 (Simple Cycle Found):** If the head of the list, `Cycle`, satisfies `is_simple_cycle(Cycle)`, the predicate proceeds. The cut `!` prevents backtracking into the next clause for this `Cycle`. It then normalizes the cycle using `normalize_cycle(Cycle, NormalizedCycle)` and recursively calls `filter_simple_cycles` on the rest of the list (`RestCandidates`). The result is constructed as `[NormalizedCycle | SimpleRest]`.

- **Recursive Step 2 (Not a Simple Cycle):** If `is_simple_cycle(Cycle)` fails, the second clause is tried. It simply ignores the current `_Cycle` (using `_` to indicate the variable is not used) and recursively calls `filter_simple_cycles` on the rest of the list.

- `is_simple_cycle/1`: Checks if a single elementary cycle is simple.
  - It first reverses the DFS cycle (e.g., `[a, d, c, b, a]`) to get the forward path (`[a, b, c, d, a]`) using `reverse/2`.
  - It decomposes the forward path `ForwardCycle` into the start node and the rest using pattern matching: `ForwardCycle = [Start | PathNodes]`.
  - It uses `append(PathNodesWithoutEnd, [Start], PathNodes)` to get the list of nodes in the cycle path excluding the repeated start/end node (e.g., `[b, c, d]`). `append/3` joins or splits lists.
  - It reconstructs the list of unique nodes in the cycle in forward order: `NodesInCycle = [Start | PathNodesWithoutEnd]` (e.g., `[a, b, c, d]`).
  - Finally, it calls `check_all_pairs_shortest_path/2` with the list of unique cycle nodes.

- `check_all_pairs_shortest_path/2`: Iterates through all ordered pairs of distinct nodes (u, v) within the cycle (`NodesInCycle`).
  - **Base Case:** If the first list of nodes is empty (`[]`), all pairs starting from previous nodes have been checked, so it succeeds.
  - **Recursive Step:** Takes the first node `N1` from the list. It calls `check_pairs_from_node(N1, OriginalCycleNodes, OriginalCycleNodes)` to check all pairs starting with `N1`. If that succeeds, it recursively calls itself with the rest of the list (`RestN1`).

- `check_pairs_from_node/3`: For a given node `N1`, iterates through all other nodes `N2` in the cycle (`OriginalCycleNodes`).
  - **Base Case:** If the list of potential `N2` nodes is empty (`[]`), all pairs for `N1` have been checked successfully.
  - **Recursive Step:** Takes the head node `N2`.
  - If `N1 == N2`, it's the same node, so this pair is skipped (`true`).
  - Otherwise (`;`), it calculates the distance along the cycle path from `N1` to `N2` using `get_cycle_distance/4`, storing it in `CycleDist`.

- It calculates the shortest path distance in the overall graph from `N1` to `N2` using `shortest_path_length/3`, storing it in `ShortestDist`.
- It checks the simplicity condition:
- If `ShortestDist == -1` (meaning no path exists in the graph between `N1` and `N2` other than potentially the cycle path itself), the condition holds if `CycleDist > 0` (it's a valid forward path along the cycle). If `CycleDist` is not positive (shouldn't happen for distinct nodes in a cycle but handles edge cases), it fails using `!, fail`.
- Otherwise (a shortest path exists), the condition holds only if `ShortestDist >= CycleDist`. If a shorter path exists (`ShortestDist < CycleDist`), the cycle is not simple, and this check will fail.
- If the check succeeds, a cut `!` is used to prevent backtracking for the current pair `(N1, N2)`, and the predicate recursively calls itself for `N1` and the rest of the nodes (`RestN2`).
- **Failure Clause:** The final clause `check_pairs_from_node(_, _, _) :- !, fail.` ensures that if any pair fails the simplicity check, the entire predicate fails immediately due to the cut.

• `get_cycle_distance/4`: Calculates the number of edges traversed when moving from node `NodeA` to node `NodeB` along the cycle path (`CycleNodes`).

  - It finds the 0-based indices of `NodeA` and `NodeB` in the `CycleNodes` list using `nth0/3`.
  - It gets the total number of nodes (which equals the number of edges) in the cycle using `length/2`.
  - If `IndexB >= IndexA`, the distance is simply `IndexB - IndexA`.
  - Otherwise (the path wraps around), the distance is `Len - IndexA + IndexB`.
  - The result is unified with the `Distance` argument using the `is/2` operator for arithmetic evaluation.

• `shortest_path_length/3`: Finds the length (`Length`) of the shortest path between `Start` and `End` nodes using Breadth-First Search (BFS).

  - It calls `bfs/4` to perform the search. The initial call is `bfs([[Start, 0]], End, [Start], Length)`, starting the queue with the `Start` node at distance 0 and marking `Start` as visited.

- A cut `!` is used after the `bfs/4` call. If BFS succeeds and finds a path, the cut prevents backtracking to the failure clause.
- If `bfs/4` fails to find a path, the second clause `shortest_path_length(_, _, -1)` is executed, unifying `Length` with -1 to indicate no path exists.

- `bfs/4`: The standard BFS implementation: `bfs(Queue, Target, Visited, Length)`.
  - **Base Case 1 (Queue Empty):** `bfs([], _, _, _) :- !, fail.` If the queue is empty, the target was not reachable. The cut `!` prevents backtracking, and the predicate fails.
  - **Base Case 2 (Target Found):** `bfs([[Target, Length] | _], Target, _, Length) :- !.` If the node at the front of the queue is the `Target`, the shortest path is found. Its `Length` is unified with the result, and the cut `!` stops the search.
  - **Recursive Step:** `bfs([[Current, Dist] | RestQueue], Target, Visited, Length) :- ...`
  - Dequeues the current node `Current` and its distance `Dist`.
  - Finds all unvisited neighbors: `findall(Next, (edge(Current, Next),`
    `+ member(Next, Visited)), Neighbors)`.
  - Calculates the distance for neighbors: `NewDist is Dist + 1`.
  - Adds neighbors to the back of the queue: `add_neighbors_to_queue(Neighbors, NewDist, RestQueue, NewQueue)`.
  - Updates the visited list: `append(Visited, Neighbors, NewVisited)`, `list_to_set(NewVisited, UniqueVisited)`. Using `list_to_set/2` (from SWI-Prolog's `library(lists)`) efficiently removes duplicates from the visited list.
  - Recursively calls `bfs` with the new queue and visited list.

- `add_neighbors_to_queue/4`: Helper to format neighbors as `[Node, Distance]` pairs and add them to the queue.
  - Uses `findall([N, Distance], member(N, Neighbors), NeighborEntries)` to create the list of pairs.
  - Uses `append(CurrentQueue, NeighborEntries, NewQueue)` to add the new entries to the end of the existing queue, maintaining the BFS order.

2.4 Cycle Normalization

To ensure that cycles representing the same sequence of nodes but starting at different points are treated as identical, cycles are normalized:

- `normalize_cycle/2`: Takes a raw cycle (`RawCycle`) as found by DFS (e.g., `[a, d, c, b, a]`) and converts it into a standard representation (`NormalizedNodeList`).
- The normalization process involves: 1. Deconstructing the raw cycle: `RawCycle = [Start | RevPathNodes]`. 2. Reversing the path part and removing the duplicate start/end node: `reverse(RevPathNodes, [_ | ForwardPathNodes])`. 3. Reconstructing the forward cycle node list: `ForwardCycleNodes = [Start | ForwardPathNodes]` (e.g., `[a, b, c, d]`). 4. Finding the node with the "minimum" value using standard term comparison (`@<`) via `find_min_node/2`. 5. Rotating the list so that it starts with this minimum node using `rotate_list_to_start_with/3`. The result is unified with `NormalizedNodeList`.
- Example: `[a, d, c, b, a]` → `[a, b, c, d]`. `[b, e, d, c, b]` → `[b, c, d, e]`.
- `find_min_node/2`: Finds the minimum node in a list based on Prolog's standard term comparison (`@<`).
  - **Base Case:** If the list has one element `[M]`, that element is the minimum.
  - **Recursive Step:** Compares the head `H` with the minimum of the tail `MinTail`. Uses conditional `( Condition -> Then ; Else )` syntax. If `H @< MinTail`, `Min` is `H`; otherwise, `Min` is `MinTail`.
- `rotate_list_to_start_with/3`: Rotates `List` so that `Element` becomes the first element.
  - Uses `append(Before, [Element | After], List)` to split the list into the part `Before` the `Element` and the part `After` (including the element itself). The cut `!` commits to the first successful split found.
  - Uses `append([Element | After], Before, RotatedList)` to rejoin the parts in the rotated order.

- The second clause `rotate_list_to_start_with(List, _, List)` handles the case where the element is already first (the first clause's `append` fails if `Before` is empty).

The final list of simple cycles is produced using `setof/3` on the normalized cycles. `setof(Template, Goal, Set)` finds all unique instances of `Template` for which `Goal` is true, sorts them, and collects them into `Set`. This guarantees both uniqueness and a canonical order.

2.5 Main Predicate and Output

- `find_simple_cycles/1`: The top-level predicate. It orchestrates the entire process: 1. Prints initialization messages using `write/1` and `nl/0` (newline). 2. Calls `generate_edges/0`. 3. Calls `find_all_elementary_cycles/1` to get `ElementaryCycles`. 4. Calculates and prints the number of elementary cycles using `length/2`. 5. Uses conditional execution (`Condition -> Then ; Else`). 6. If `NumElementary > 0`:
  - Prints the found elementary cycles using `print_cycles_list/2`.
  - Calls `filter_simple_cycles/2` to get potentially non-unique `NormalizedSimpleCycl`
  - Uses `setof(NormCycle, Member^(member(Member, NormalizedSimpleCycles), NormCycle = Member), SimpleCycles)` to get the final unique, sorted list of `SimpleCycles`. The `Member^` syntax indicates that `Member` is an existentially quantified variable within the goal.
  - Prints the final simple cycles and their count.
  - If `setof/3` fails (no simple cycles found after filtering), it prints a message and sets `SimpleCycles` to `[]`.

  7. If `NumElementary =< 0`:
  - Prints a message indicating no elementary cycles were found.
  - Sets `SimpleCycles` to `[]`.

  8. The argument `SimpleCycles` is unified with the final list.

- `print_cycles_list/2`: A helper predicate `print_cycles_list(Header, ListOfCycles)`.
  - **Base Case:** If `ListOfCycles` is empty (`[]`), it does nothing (due to the cut `!`).

– **General Case:** Prints the `Header` string using `writeln/1`. Then, uses `forall(member(Cycle, ListOfCycles), (write(' '), writeln(Cycle)))` to iterate through each `Cycle` in the list and print it indented on a new line. `forall(Condition, Action)` succeeds if `Action` is true for all possible solutions of `Condition`.

3 Usage

1. Ensure a Prolog interpreter (like SWI-Prolog) is installed. 2. Load the program file: `?- [simpleCycle].` 3. Run the main predicate: `?- find_simple_cycles(Cycles).` 4. The program will print the intermediate elementary cycles found and the final list of unique, normalized simple cycles. The variable `Cycles` will be unified with the list of simple cycles.

4 Example Graph

The code includes an example graph defined by `node/2` and `arc/4` facts:

- Nodes: a, b, c, d, e
- Arcs forming cycle 1: a $\rightarrow$ b, b $\rightarrow$ c, c $\rightarrow$ d, d $\rightarrow$ a
- Arcs forming cycle 2: b $\rightarrow$ e, e $\rightarrow$ d, d $\rightarrow$ c, c $\rightarrow$ b
- A "shortcut" arc: a $\rightarrow$ d

In this example:

- Elementary cycles found are (normalized): `[a, b, c, d]` and `[b, c, d, e]`.
- The cycle `[a, b, c, d]` is **not simple** because the path a $\rightarrow$ d along the cycle has length 3, but a direct arc a $\rightarrow$ d exists (length 1).
- The cycle `[b, c, d, e]` **is simple** as there are no shorter paths between its constituent nodes outside the cycle edges.

Therefore, the expected output for `Cycles` is `[[b, c, d, e]]`.

5 Conclusion

The Prolog program successfully implements an algorithm to find simple cycles in a directed graph based on a shortest path criterion. It demonstrates the use of DFS for cycle detection, BFS for shortest path calculation, and Prolog's features for list manipulation and backtracking. The normalization step ensures that unique cycles are reported consistently.