Università degli studi di Urbino Carlo Bo

Corso di laurea in Informatica Scienza e Tecnologia

Progetto d’esame di Programmazione Logica e Funzionale

Anno accademico 2025/2026

Andrea Pedini, N. 322918

Matteo Fraternali, N. 316637

1 Specifica del Problema

Scrivere un programma Haskell e un programma Prolog che leggano da file e stampino a schermo i seguenti dati di input:

- Una lista di numeri interi
- Una lista di coppie di numeri interi

Successivamente il programma dovrà:

- Costruire un grafo orientato i cui vertici sono costituiti dalla lista di interi letta da file e gli archi sono rappresentati dalle coppie di interi letti da file.
- Acquisire un numero intero tra i vertici del grafo a rappresentare il vertice di partenza validando che esso sia un vertice presente nel grafo.
- Costruire un nuovo grafo orientato in cui i vertici sono le componenti fortemente connesse rappresentate da un singolo vertice scelto arbitrariamente tra i vertici di una componente fortemente connessa e gli archi sono i collegamenti tra le componenti fortemente connesse distinte.
- Stampare a schermo il numero di componenti fortemente connesse diverse dalla componente contenente il vertice di partenza che hanno grado entrante uguale a 0.

2 Analisi del problema

2.1 Dati di ingresso del problema

I dati di ingresso sono i seguenti:

- Una lista di numeri interi
- Una lista di coppie di numeri interi
- Un numero intero

2.2 Dati di uscita del problema

I dati di uscita del problema sono costituiti solamente dal numero di componenti fortemente connesse diverse dalla componente contenente il vertice di partenza che hanno grado entrante uguale a 0.

2.3 Relazioni intercorrenti tra i dati del problema

Le relazioni intercorrenti tra i dati di ingresso e i dati di uscita del problema riguardano la struttura del grafo e la sua trasformazione attraverso il concetto di componenti fortemente connesse (SCC - Strongly Connected Components).
Una componente fortemente connessa C è un insieme di vertici tale per cui per ogni $v1,\ v2$ appartenenti a C esiste un percorso da $v1,\ v2$ e uno da $v2,\ v1$.

Dopo la creazione del grafo orientato iniziale, l'individuazione delle componenti fortemente connesse, la determinazione della SCC contenente il vertice di partenza, il dato di uscita sarà dato semplicemente dal numero delle SCC con grado entrante 0, escludendo la SCC del vertice di partenza.
Adottiamo questo metodo perchè siamo interessati a ottenere il numero minimo di archi da aggiungere per rendere tutto il grafo raggiungibile dal vertice scelto, mentre se ci limitassimo ad aggiungere tanti archi quanti sono i vertici non raggiungibili non otterremmo il numero minimo.

3 Progettazione dell’Algoritmo

3.1 Scelte di Progetto

Le scelte di progetto che sono state effettuate riguardano:

- la rappresentazione dei dati del grafo come semplici liste di interi per i vertici e liste di coppie di interi per gli archi.
- La definizione di una funzione di visita in profondità che consenta tramite un parametro formale la scelta di una strategia di accumulazione dei vertici per il risultato da ritornare.

3.2 Passi dell’Algoritmo

- Acquisizione del grafo
    - Leggere da un file una lista di vertici e una lista di archi, costruendo un grafo diretto
- Visualizzazione iniziale
    - Stampa a schermo dei vertici e degli archi del grafo così come sono stati letti
- Calcolo delle componenti fortemente connesse (SCC)
    - Esaminare il grafo per identificare insiemi di vertici tali che ciascun vertice del gruppo sia raggiungibile da ogni altro vertice dello stesso gruppo
    - Generare una lista di questi insiemi (componenti)
    - Visualizzare le componenti trovate
- Costruzione del grafo compresso
    - Considerare ogni componente come un singolo nodo
    - Creare archi tra componenti se esiste almeno un arco nel grafo originale che collega vertici appartenenti a componenti diverse
    - Mostrare il grafo risultante, dove ogni nodo rappresenta una componente
- Calcolo del grado entrante delle componenti
    - Per ogni componente, contare quanti archi entrano in essa da altre componenti
    - Visualizzare questi valori
- Scelta del vertice di partenza
    - Richiedere all’utente di indicare un vertice specifico validando che esso sia presente nel grafo iniziale
    - Identificare la componente che contiene questo vertice
- Conteggio delle componenti indipendenti
    - Contare quante componenti nel grafo compresso non hanno archi in ingresso escludendo la componente contenente il vertice di partenza
    - Visualizzare il numero risultante.

4 Implementazione dell’Algoritmo

4.1 Implementazione in Haskell

```haskell
-- ######################################################################
-- #         Corso di Programmazione Logica e Funzionale                #
-- #        Progetto per la sessione invernale A.A 2025/2026            #
-- #                          di Andrea Pedini                          #
-- #                        Matricola: 322918                           #
-- #                        e Matteo Fraternali                         #
-- #                        Matricola: 316637                           #
-- ######################################################################

module Main where

-- Caricamento della funzione per eliminare duplicati da una lista
import Data.List ( nub )

-------------------------------------------------
-- FUNZIONE PRINCIPALE
-------------------------------------------------

{-
    Funzione principale del programma.
    - legge un grafo da file
    - calcola le componenti fortemente connesse
    - costruisce il grafo compresso
    - calcola quante SCC hanno grado entrante zero
-}
main :: IO ()
main = do
    contenuto <- readFile "input.txt"
    let righe   = lines contenuto
        vertici = read ( head righe ) :: [ Int ]
        archi   = read ( righe !! 1 ) :: [ ( Int , Int ) ]

    putStrLn "\n======================================"
    putStrLn "            GRAFO LETTO DA FILE       "
    putStrLn "--------------------------------------"
    putStrLn $ "Vertici: " ++ show vertici
    putStrLn $ "Archi:   " ++ show archi

    let componenti = kosaraju vertici archi

    putStrLn "\n======================================"
    putStrLn "       COMPONENTI FORTEMENTE CONNESSE "
    putStrLn "--------------------------------------"
    mapM_
        ( \( i , c ) -> putStrLn $ "SCC " ++ show i ++ ": " ++ show c )
        ( zip [0 ..] componenti )
```

```haskell
    let grafoCompresso = comprimiGrafo componenti archi

    putStrLn "\n======================================"
    putStrLn "           GRAFO COMPRESSO            "
    putStrLn "--------------------------------------"
    mapM_
        ( \( i , j ) -> putStrLn $ "SCC_" ++ show i ++ " -> SCC_" ++ show j )
        grafoCompresso

    verticePartenza <- acquisisciVertice vertici

    let numeroZero =
            contaSCCConGradoZero verticePartenza componenti grafoCompresso

    putStrLn "\n======================================"
    putStrLn $
        "Numero di SCC con grado entrante 0 (esclusa la partenza): "
        ++ show numeroZero
    putStrLn "======================================\n"

-------------------------------------------------
-- FUNZIONE DI ACQUISIZIONE DEL VERTICE
-------------------------------------------------

{-
    Funzione che acquisisce da tastiera un vertice valido.
    - l'argomento è la lista dei vertici del grafo
-}
acquisisciVertice :: [Int] -> IO Int
acquisisciVertice vertici = do
    putStrLn $
        "Inserisci il vertice di partenza (tra " ++ show vertici ++ "):"
    input <- getLine
    case reads input :: [(Int, String)] of
        [(v, _)] | v `elem` vertici -> return v
        _ -> do
            putStrLn "Vertice non valido! Riprova."
            acquisisciVertice vertici

-------------------------------------------------
-- FUNZIONI DI BASE SUI GRAFI
-------------------------------------------------

{-
    Funzione che restituisce tutti i vertici adiacenti a un vertice.
    - il primo argomento è il vertice di partenza
    - il secondo argomento è la lista degli archi
-}
verticiAdiacenti :: Int -> [(Int, Int)] -> [Int]
verticiAdiacenti v archi = [ y | ( x , y ) <- archi , x == v ]
```

```haskell
{-
    Funzione che inverte la direzione di tutti gli archi del grafo.
    - l’argomento è la lista degli archi
-}
invertiArchi :: [(Int, Int)] -> [(Int, Int)]
invertiArchi [] = []
invertiArchi ( ( x , y ) : xs ) = ( y , x ) : invertiArchi xs

-------------------------------------------------------
-- FUNZIONI DI COMBINAZIONE PER LA VISITA IN PROFONDITA'
-------------------------------------------------------

{-
    Funzione di combinazione che inserisce il vertice
    alla fine della lista dei risultati (post-order).
-}
aggiungiInCoda :: Int -> [Int] -> [Int]
aggiungiInCoda v res = res ++ [v]

{-
    Funzione di combinazione che inserisce il vertice
    all'inizio della lista dei risultati (pre-order).
-}
aggiungiInTesta :: Int -> [Int] -> [Int]
aggiungiInTesta v res = v : res

-------------------------------------------------
-- VISITA IN PROFONDITA' GENERICA
-------------------------------------------------

{-
    Visita in profondità' generica parametrizzata da una funzione di combinazione.
    - il primo argomento decide come aggiungere il vertice corrente
    - il secondo argomento è il vertice di partenza
    - il terzo argomento è la lista degli archi
    - il quarto argomento è la lista dei vertici visitati

    Restituisce:
    - la lista aggiornata dei vertici visitati
    - la lista dei risultati accumulati secondo la strategia scelta
-}
visitaInProfondita :: (Int -> [Int] -> [Int]) -> Int -> [(Int, Int)] -> [Int] -> ([Int], [Int])
visitaInProfondita combina v archi visitati
    | v `elem` visitati = ( visitati , [] )
    | otherwise =
        let visitati' = v : visitati
            (visitatiFinali, risultatiFigli) =
                foldl visita (visitati', []) (verticiAdiacenti v archi)
        in (visitatiFinali, combina v risultatiFigli)
  where
    visita (vis, res) u =
```

```
        let (vis', res') = visitaInProfondita combina u archi vis
        in (vis', res ++ res')

-------------------------------------------------
-- VISITA IN PROFONDITA' SPECIALIZZATE
-------------------------------------------------

{-
    Visita in profondità che calcola l'ordine di fine dei vertici.
-}
visitaInProfonditaOrdineFine :: Int -> [(Int, Int)] -> [Int] -> ([Int], [Int])
visitaInProfonditaOrdineFine = visitaInProfondita aggiungiInCoda

{-
    Visita in profondità che costruisce una singola componente fortemente connessa.
-}
visitaInProfonditaComponente :: Int -> [(Int, Int)] -> [Int] -> ([Int], [Int])
visitaInProfonditaComponente = visitaInProfondita aggiungiInTesta

-------------------------------------------------
-- ORDINE DI FINE GLOBALE
-------------------------------------------------

{-
    Funzione che calcola l'ordine di fine globale del grafo.
    - il primo argomento è la lista dei vertici
    - il secondo argomento è la lista degli archi
-}
ordineDiFine :: [ Int ] -> [ ( Int , Int ) ] -> [ Int ]
ordineDiFine vertici archi =
    snd $
        foldl visitaGlobale ( [] , [] ) vertici
  where
    visitaGlobale ( vis , ord ) v =
        let ( vis' , ord' ) = visitaInProfonditaOrdineFine v archi vis
        in ( vis' , ord ++ ord' )

-------------------------------------------------
-- ALGORITMO DI KOSARAJU
-------------------------------------------------

{-
    Funzione che calcola tutte le componenti fortemente connesse del grafo
    utilizzando l'algoritmo di Kosaraju.
    - il primo argomento è la lista dei vertici
    - il secondo argomento è la lista degli archi
-}
kosaraju :: [Int] -> [(Int, Int)] -> [[Int]]
kosaraju vertici archi =
    componenti
  where
```

```haskell
    ordine         = reverse (ordineDiFine vertici archi)
    archiInvertiti = invertiArchi archi
    componenti =
        snd $
            foldl costruisci ([], []) ordine

    costruisci (vis, comps) v
        | v `elem` vis = (vis, comps)
        | otherwise =
            let (vis', comp) = visitaInProfonditaComponente v archiInvertiti vis
            in (vis', comps ++ [comp])

------------------------------------------------
-- GRAFO COMPRESSO
------------------------------------------------

{-
    Funzione che restituisce l'indice della SCC contenente un vertice.
    - il primo argomento è il vertice
    - il secondo argomento è la lista delle SCC
-}
indiceSCC :: Int -> [[Int]] -> Int
indiceSCC v sccs =
    case [i | (i, comp) <- zip [0 ..] sccs , v `elem` comp] of
        (i:_) -> i
        []        -> error ("Vertice non trovato: " ++ show v)

{-
    Funzione che costruisce il grafo compresso.
    - il primo argomento sono le SCC
    - il secondo argomento è la lista degli archi originali
-}
comprimiGrafo :: [[Int]] -> [(Int, Int)] -> [(Int, Int)]
comprimiGrafo sccs archi =
    nub
        [ (i, j)
        | (x, y) <- archi
        , let i = indiceSCC x sccs
        , let j = indiceSCC y sccs
        , i /= j
        ]

{-
    Funzione che calcola il grado entrante di una SCC.
    - il primo argomento è l'indice della SCC
    - il secondo argomento è la lista degli archi del grafo compresso
-}
gradoEntrante :: Int -> [(Int, Int)] -> Int
gradoEntrante c archi =
    length [() | (_, y) <- archi , y == c]
```

```haskell
{-
    Funzione che conta quante SCC hanno grado entrante zero,
    escludendo quella contenente il vertice di partenza.
    - il primo argomento è il vertice di partenza
    - il secondo argomento sono le SCC
    - il terzo argomento è il grafo compresso
-}
contaSCCConGradoZero :: Int -> [[Int]] -> [(Int, Int)] -> Int
contaSCCConGradoZero v sccs archi =
    length
        [ i
        | i <- [0 .. length sccs - 1]
        , i /= indiceSCC v sccs
        , gradoEntrante i archi == 0
        ]
```

File: connessioniGrafi.hs

4.2 Implementazione in Prolog

```
/* ##############################################################
# Corso di Programmazione Logica e Funzionale                  #
# Progetto per la sessione autunnale A.A. 2024/2025            #
# Versione GNU Prolog                                         #
############################################################## */

/*
Specifica : Scrivere un programma Prolog che legga un grafo orientato
da file e calcoli le sue Componenti Fortemente Connesse (SCC)
utilizzando l'algoritmo di Kosaraju.
Successivamente il programma costruisce il grafo compresso
e determina il numero di SCC con grado entrante zero,
escludendo la SCC contenente un nodo scelto dall'utente.
*/

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% MAIN
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

/* Predicato principale del programma .
- Legge il grafo da file
- Calcola le componenti fortemente connesse
- Stampa i risultati ottenuti */
main :-
    leggi_grafo_da_file('input.txt', G),
    nodi(G, Nodi),
    archi(G, Archi),
    kosaraju(G, SCCs),

    stampa_separatore,
    write('            GRAFO LETTO DA FILE       '), nl,
    stampa_riga,
    write('Vertici: '), write(Nodi), nl,
    write('Archi:   '), write(Archi), nl,

    nl,
    stampa_separatore,
    write('       COMPONENTI FORTEMENTE CONNESSE '), nl,
    stampa_riga,
    stampa_scc_numerate(SCCs, 0),

    nl,
    stampa_separatore,
    write('           GRAFO COMPRESSO'), nl,
    stampa_riga,
    write('Inserisci il vertice di partenza (tra '),
    write(Nodi), write('):'), nl,

    leggi_numero(Nodo),
```

```
    scc_di_nodo(Nodo, SCCs, SCCpartenza),

    findall(
        S,
        ( membro(S,SCCs),
          S \= SCCpartenza,
          grado_entrante(G,SCCs,S,0)
        ),
        ZeroIn
    ),
    length(ZeroIn, Conteggio),

    nl,
    stampa_separatore,
    write('Numero di SCC con indegree 0 (esclusa partenza): '),
    write(Conteggio), nl,
    stampa_separatore.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% STAMPA
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

/* Predicato che stampa una linea separatrice */
stampa_separatore :-
    write('======================================'), nl.

/* Predicato che stampa una riga separatrice */
stampa_riga :-
    write('--------------------------------------'), nl.

/* Predicato che stampa le SCC numerate .
- Il primo parametro la lista delle SCC
- Il secondo parametro l'indice corrente */
stampa_scc_numerate([], _).
stampa_scc_numerate([S|T], N) :-
    write('SCC '), write(N), write(': '),
    write(S), nl,
    N1 is N + 1,
    stampa_scc_numerate(T, N1).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% LETTURA GRAFO DA FILE
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

/* Predicato che legge una linea da uno stream .
- Il primo parametro lo stream di input
- Il secondo parametro la linea letta */
leggi_linea(Stream, Line) :-
    get_char(Stream, Char),
    leggi_linea_aux(Stream, Char, Chars),
```

```
    atom_chars(Line, Chars).

/* Predicato ausiliario per la lettura delle linee .
- Il primo parametro lo stream
- Il secondo parametro il carattere corrente
- Il terzo parametro la lista dei caratteri letti */
leggi_linea_aux(_, end_of_file, []) :- !.
leggi_linea_aux(_, '\n', []) :- !.
leggi_linea_aux(Stream, Char, [Char|Chars]) :-
    get_char(Stream, NextChar),
    leggi_linea_aux(Stream, NextChar, Chars).

/* Predicato che legge un termine Prolog da file .
- Il primo parametro lo stream
- Il secondo parametro il termine letto */
leggi_termine(Stream, Termine) :-
    leggi_linea(Stream, Line),
    atom_concat(Line, '.', LineConPunto),
    read_from_atom(LineConPunto, Termine).

/* Predicato che legge il grafo da file .
- Il primo parametro il nome del file
- Il secondo parametro il grafo letto */
leggi_grafo_da_file(File, grafo(Nodi, Archi)) :-
    open(File, read, Stream),
    leggi_termine(Stream, Nodi),
    leggi_termine(Stream, Archi),
    close(Stream).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% INPUT NUMERICO
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

/* Predicato che legge un numero da input standard .
- Il parametro il numero letto */
leggi_numero(N) :-
    leggi_linea(user_input, Line),
    atom_codes(Line, Codes),
    number_codes(N, Codes).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% PREDICATI BASE
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

/* Predicato che verifica l'appartenenza di un elemento a una lista .
- Il primo parametro l'elemento
- Il secondo parametro la lista */
membro(X,[X|_]).
membro(X,[_|T]) :- membro(X,T).

/* Predicato che restituisce i nodi del grafo .
```

```
- Il primo parametro il grafo
- Il secondo parametro la lista dei nodi */
nodi(grafo(N,_), N).

/* Predicato che restituisce gli archi del grafo .
- Il primo parametro il grafo
- Il secondo parametro la lista degli archi */
archi(grafo(_,A), A).

/* Predicato che verifica l'adiacenza tra due nodi .
- Il primo parametro il grafo
- Il secondo parametro il nodo sorgente
- Il terzo parametro il nodo destinazione */
adiacente(grafo(_,A), X, Y) :- membro((X,Y), A).

/* Predicato che restituisce i nodi adiacenti a un nodo .
- Il primo parametro il grafo
- Il secondo parametro il nodo
- Il terzo parametro la lista dei nodi adiacenti */
adiacenti(G, X, L) :- findall(Y, adiacente(G,X,Y), L).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% STRATEGIE DI COMBINAZIONE PER LA VISITA IN PROFONDITA'
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

/* Predicato che inserisce un nodo in coda alla lista .
Usato per l'ordine di completamento */
combina_fine(N, Lista, Risultato) :-
    append(Lista, [N], Risultato).

/* Predicato che inserisce un nodo in testa alla lista .
Usato per la costruzione delle SCC */
combina_testa(N, Lista, [N|Lista]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% VISITA IN PROFONDITA' GENERICA
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

/* Predicato di visita in profondità generico .
- Il primo parametro la strategia di combinazione
- Il secondo parametro il grafo
- Il terzo parametro il nodo corrente
- Il quarto parametro i nodi visitati
- Il quinto parametro i nodi visitati aggiornati
- Il sesto parametro il risultato della visita */
visitaInProfondita(_, _, N, V, V, []) :-
    membro(N, V), !.

visitaInProfondita(Combina, G, N, V, V2, Risultato) :-
    \+ membro(N, V),
    adiacenti(G, N, Vicini),
```

```
    visitaInProfondita_lista(Combina, G, Vicini, [N|V], V1, RisFigli),
    call(Combina, N, RisFigli, Risultato),
    V2 = V1.

/* Predicato ausiliario per visitare una lista di nodi .
- Parametri analoghi alla visita in profondità principale */
visitaInProfondita_lista(_, _, [], V, V, []).
visitaInProfondita_lista(Combina, G, [H|T], V, V2, Risultato) :-
    visitaInProfondita(Combina, G, H, V, V1, R1),
    visitaInProfondita_lista(Combina, G, T, V1, V2, R2),
    append(R1, R2, Risultato).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% VISITA IN PROFONDITA' SPECIALIZZATE
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

/* Visita in profondità per il calcolo dell'ordine di completamento */
visitaInProfondita_ordine(G, N, V, V2, Ordine) :-
    visitaInProfondita(combina_fine, G, N, V, V2, Ordine).

/* Visita in profondità per la costruzione di una componente fortemente connessa */
visitaInProfondita_scc(G, N, V, V2, Comp) :-
    visitaInProfondita(combina_testa, G, N, V, V2, Comp).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% VISITA IN PROFONDITA' ORDINE DI COMPLETAMENTO (GLOBALE)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

/* Predicato che calcola l'ordine di completamento globale del grafo */
visitaInProfondita_grafo(G, Ordine) :-
    nodi(G, Nodi),
    visitaInProfondita_lista(combina_fine, G, Nodi, [], _, Ordine).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% GRAFO TRASPOSTO
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

/* Predicato che costruisce il grafo trasposto */
trasposto(grafo(N,A), grafo(N,AT)) :-
    trasponi_archi(A, AT).

/* Predicato che inverte tutti gli archi del grafo */
trasponi_archi([], []).
trasponi_archi([(X,Y)|T], [(Y,X)|R]) :-
    trasponi_archi(T,R).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% KOSARAJU
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

/* Predicato che implementa l'algoritmo di Kosaraju */
kosaraju(G, SCCs) :-
    visitaInProfondita_grafo(G, Ordine),
    reverse(Ordine, OrdInv),
    trasposto(G, GT),
    kosaraju_visita(GT, OrdInv, [], SCCs).

/* Predicato ausiliario di visita per Kosaraju */
kosaraju_visita(_, [], _, []).
kosaraju_visita(G, [N|T], V, SCCs) :-
    membro(N, V), !,
    kosaraju_visita(G, T, V, SCCs).
kosaraju_visita(G, [N|T], V, [SCC|R]) :-
    visitaInProfondita_scc(G, N, V, V1, SCC),
    kosaraju_visita(G, T, V1, R).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% GRAFO DELLE SCC
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

/* Predicato che individua la SCC di un nodo */
scc_di_nodo(N, [S|_], S) :- membro(N,S), !.
scc_di_nodo(N, [_|T], S) :- scc_di_nodo(N,T,S).

/* Predicato che verifica l'esistenza di un arco tra SCC */
arco_scc(G, SCCs, S1, S2) :-
    archi(G,A),
    membro((X,Y),A),
    scc_di_nodo(X,SCCs,S1),
    scc_di_nodo(Y,SCCs,S2),
    S1 \= S2.

/* Predicato che calcola il grado entrante di una SCC */
grado_entrante(G, SCCs, S, Grado) :-
    findall(1, arco_scc(G,SCCs,_,S), L),
    length(L, Grado).
```

File: connessioniGrafi.pl

5 Testing del Programma

Testing del programma Haskell

Test 1
Vertici: 1,2,3,4,5
Archi: (1,2) - (2,3) - (3,4) - (4,5) - (5,1)
SCC: [1,5,4,3,2]
Vertice di partenza: 4
Risultato: 0

Test 2
Vertici: 1,2,3,4,5,6
Archi: (1,2) - (2,3) - (3,1) - (4,5)
SCC: [6] - [4] - [5] - [1,3,2]
Vertice di partenza: 4
Risultato: 2

Test 3
Vertici: 1,2,3,4,5,6
Archi: (1,2) - (1,3) - (2,4) - (3,4) - (4,5) - (4,6)
SCC: [1] - [3] - [2] - [4] - [6] - [5]
Vertice di partenza: 2
Risultato: 1

Test 4
Vertici: 1,2,3,4,5,6,7
Archi: (1,2) - (2,3) - (3,1) - (3,4) - (4,5) - (5,3) - (5,6) - (6,7)
SCC: [1,3,2,5,4] - [6] - [7]
Vertice di partenza: 5
Risultato: 0

Test 5
Vertici: 1,2,3,4,5,6
Archi: (1,2) - (2,3) - (3,3) - (4,5)
SCC: [6] - [4] - [5] - [1] - [2] - [3]
Vertice di partenza: 6
Risultato: 2

Test 6
Vertici: 1,2,3,4,5,6
Archi: (1,2) - (1,3) - (1,4) - (1,5) - (1,6)
SCC: [1] - [6] - [5] - [4] - [3] - [2]
Vertice di partenza: 3
Risultato: 1

Test 7
Vertici: 1,2,3,4,5,6,7,8
Archi: (1,2) - (2,3) - (3,1) - (4,5) - (5,6) - (6,4) - (6,7) - (7,8)
SCC: [4,6,5] - [7] - [8] - [1,3,2]
Vertice di partenza: 8
Risultato: 2

Test 8
Vertici: 1,2,3,4,5,6,7
Archi: (1,2) - (2,3) - (3,4) - (4,5) - (5,6) - (6,7) - (7,4)
SCC: [1] - [2] - [3] - [4,7,6,5]
Vertice di partenza: 7
Risultato: 1

Test 9
Vertici: 1,2,3,4,5,6,7,8
Archi: (1,2) - (2,3) - (3,1) - (3,4) - (4,5) - (5,3) - (5,6) - (6,7) - (7,6) - (7,8)
SCC: [1,3,2,5,4] - [6,7] - [8]
Vertice di partenza: 8
Risultato: 1

Test 10
Vertici: 1,2,3,4,5,6,7
Archi: (1,2) - (2,3) - (3,1) - (2,4) - (4,5) - (5,6) - (6,4) - (6,7)
SCC: [1,3,2] - [4,6,5] - [7]
Vertice di partenza: 6
Risultato: 1

Testing del programma Prolog

Test 1
Vertici: 1,2,3,4,5
Archi: (1,2) - (2,3) - (3,4) - (4,5) - (5,1)
SCC: [1,5,4,3,2]
Vertice di partenza: 4
Risultato: 0

Test 2
Vertici: 1,2,3,4,5,6
Archi: (1,2) - (2,3) - (3,1) - (4,5)
SCC: [6] - [4] - [5] - [1,3,2]
Vertice di partenza: 4
Risultato: 2

Test 3
Vertici: 1,2,3,4,5,6
Archi: (1,2) - (1,3) - (2,4) - (3,4) - (4,5) - (4,6)
SCC: [1] - [3] - [2] - [4] - [6] - [5]
Vertice di partenza: 2
Risultato: 1

Test 4
Vertici: 1,2,3,4,5,6,7
Archi: (1,2) - (2,3) - (3,1) - (3,4) - (4,5) - (5,3) - (5,6) - (6,7)
SCC: [1,3,2,5,4] - [6] - [7]
Vertice di partenza: 5
Risultato: 0

Test 5
Vertici: 1,2,3,4,5,6
Archi: (1,2) - (2,3) - (3,3) - (4,5)
SCC: [6] - [4] - [5] - [1] - [2] - [3]
Vertice di partenza: 6
Risultato: 2

Test 6
Vertici: 1,2,3,4,5,6
Archi: (1,2) - (1,3) - (1,4) - (1,5) - (1,6)
SCC: [1] - [6] - [5] - [4] - [3] - [2]
Vertice di partenza: 3
Risultato: 1

Test 7
Vertici: 1,2,3,4,5,6,7,8
Archi: (1,2) - (2,3) - (3,1) - (4,5) - (5,6) - (6,4) - (6,7) - (7,8)
SCC: [4,6,5] - [7] - [8] - [1,3,2]
Vertice di partenza: 8
Risultato: 2

Test 8
Vertici: 1,2,3,4,5,6,7
Archi: (1,2) - (2,3) - (3,4) - (4,5) - (5,6) - (6,7) - (7,4)
SCC: [1] - [2] - [3] - [4,7,6,5]
Vertice di partenza: 7
Risultato: 1

Test 9
Vertici: 1,2,3,4,5,6,7,8
Archi: (1,2) - (2,3) - (3,1) - (3,4) - (4,5) - (5,3) - (5,6) - (6,7) - (7,6) - (7,8)
SCC: [1,3,2,5,4] - [6,7] - [8]
Vertice di partenza: 8
Risultato: 1

Test 10
Vertici: 1,2,3,4,5,6,7
Archi: (1,2) - (2,3) - (3,1) - (2,4) - (4,5) - (5,6) - (6,4) - (6,7)
SCC: [1,3,2] - [4,6,5] - [7]
Vertice di partenza: 6
Risultato: 1