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.
- Costruire un nuovo grafo orientato in cui i vertici sono le componenti fortemente connesse del grafo iniziale 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.
- Acquisire un numero intero tra i vertici del grafo a rappresentare il vertice di partenza validando che esso sia un vertice presente nel grafo iniziale.
- 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 vertice
    - 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 vertice 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                           #
-- ######################################################################

{-
    Specifica : Scrivere un programma Haskell 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 vertice scelto dall'utente.
-}

module Main where

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

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

{-
    Azione principale del programma.

    1) Legge un grafo orientato dal file "input.txt".
    2) Verifica che il formato e il contenuto del grafo siano validi.
    3) Calcola le Componenti Fortemente Connesse (SCC) con Kosaraju.
    4) Costruisce il grafo compresso (ogni SCC diventa un nodo).
    5) Richiede all'utente un vertice valido di partenza.
    6) Conta quante SCC hanno grado entrante zero,
       escludendo la SCC contenente il vertice scelto.

    Se il file contiene errori il programma termina in modo controllato.
-}
main :: IO ()
main = do
    risultato <- leggiGrafoDaFile "input.txt"

    case risultato of
        Nothing -> do
            putStrLn "\nIl programma è stato terminato a causa di errori nel file di input.\n"
```

```
        Just (vertici, archi) -> do

            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)

            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"

--------------------------------------------------
-- LETTURA E VALIDAZIONE DEL GRAFO DA FILE
--------------------------------------------------

{-
    Azione che legge e valida un grafo orientato da file.

    Parametri:
    - nomeFile : percorso del file da leggere

    Il file deve contenere almeno due righe:
    1) Lista dei vertici  (es: [1,2,3])
    2) Lista degli archi  (es: [(1,2),(2,3)])
```

```
    Restituisce:
    - Just (vertici, archi) se parsing e validazione hanno successo
    - Nothing se si verifica un errore
-}
leggiGrafoDaFile :: FilePath -> IO (Maybe ([Int], [(Int, Int)]))
leggiGrafoDaFile nomeFile = do
    contenuto <- readFile nomeFile
    case lines contenuto of
        vLine:aLine:_ -> parseGrafo vLine aLine
        _ -> errore "Errore: il file deve contenere almeno due righe (lista vertici e lista
archi)."

{-
    Azione che esegue il parsing testuale delle due righe lette dal file.

    Parametri:
    - rVertici : stringa contenente la lista dei vertici
    - rArchi   : stringa contenente la lista degli archi

    Usa 'reads' per convertire le stringhe nei tipi Haskell corretti.
-}
parseGrafo :: String -> String -> IO (Maybe ([Int], [(Int, Int)]))
parseGrafo rVertici rArchi =
    case (reads rVertici, reads rArchi) of
        ([(vertici, "")], [(archi, "")]) ->
            validaGrafo vertici archi
        _ ->
            errore "Errore: formato non valido. Verificare parentesi, virgole e struttura delle
liste."

{-
    Azione che verifica la validità semantica del grafo.

    Parametri:
    - vertici : lista dei vertici del grafo
    - archi   : lista degli archi orientati del grafo

    Controlli effettuati:
    - lista vertici non vuota
    - assenza di vertici duplicati
    - archi che usano solo vertici esistenti
-}
validaGrafo :: [Int] -> [(Int, Int)] -> IO (Maybe ([Int], [(Int, Int)]))
validaGrafo vertici archi
    | null vertici =
        errore "Errore: la lista dei vertici è vuota. Il grafo deve contenere almeno un
vertice."
    | not (verticiSenzaDuplicati vertici) =
        errore "Errore: la lista dei vertici contiene duplicati. Ogni vertice deve comparire una
sola volta."
```

```
    | not (archiValidi vertici archi) =
        errore "Errore: esistono archi che utilizzano vertici non presenti nella lista dei
vertici."
    | otherwise =
        return (Just (vertici, archi))

{-
    Funzione che verifica l'assenza di vertici duplicati.

    Parametri:
    - vertici : lista dei vertici del grafo

    Restituisce:
    - True se tutti i vertici sono distinti
    - False se esiste almeno un duplicato
-}
verticiSenzaDuplicati :: [Int] -> Bool
verticiSenzaDuplicati vertici =
    length vertici == length (nub vertici)

{-
    Funzione che controlla che ogni arco utilizzi solo vertici esistenti.

    Parametri:
    - vertici : lista dei vertici ammessi
    - archi   : lista degli archi del grafo
-}
archiValidi :: [Int] -> [(Int, Int)] -> Bool
archiValidi vertici =
    all (\(x, y) -> x `elem` vertici && y `elem` vertici)

{-
    Azione che stampa un messaggio di errore e restituisce Nothing.
-}
errore :: String -> IO (Maybe a)
errore msg = putStrLn msg >> return Nothing

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

{-
    Azione che acquisisce da tastiera un vertice valido.

    Parametri:
    - vertici : lista dei vertici del grafo

    Continua a chiedere input finché l'utente non inserisce
    un intero presente nella lista dei vertici.
-}
acquisisciVertice :: [Int] -> IO Int
```

```haskell
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: inserire un numero presente nella lista dei vertici."
            acquisisciVertice vertici

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

{-
    Restituisce tutti i vertici adiacenti ad un vertice dato.

    Parametri:
    - v     : vertice di partenza
    - archi : lista degli archi del grafo
-}
verticiAdiacenti :: Int -> [(Int, Int)] -> [Int]
verticiAdiacenti v archi = [y | (x, y) <- archi, x == v]

{-
    Inverte la direzione di tutti gli archi del grafo.
-}
invertiArchi :: [(Int, Int)] -> [(Int, Int)]
invertiArchi [] = []
invertiArchi ((x, y) : xs) = (y, x) : invertiArchi xs

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

{- Inserisce il vertice in coda (post-order). -}
aggiungiInCoda :: Int -> [Int] -> [Int]
aggiungiInCoda v res = res ++ [v]

{- Inserisce il vertice in testa (pre-order). -}
aggiungiInTesta :: Int -> [Int] -> [Int]
aggiungiInTesta v res = v : res

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

{-
    Implementazione generica della DFS parametrizzata.

    Parametri:
```

```
    - combina  : funzione di combinazione dei risultati
    - v        : vertice corrente
    - archi    : lista archi
    - visitati : lista vertici già visitati
-}
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 SPECIALIZZATE
-------------------------------------------------

{- DFS per calcolo ordine di fine. -}
visitaInProfonditaOrdineFine :: Int -> [(Int, Int)] -> [Int] -> ([Int], [Int])
visitaInProfonditaOrdineFine = visitaInProfondita aggiungiInCoda

{- DFS per costruzione SCC. -}
visitaInProfonditaComponente :: Int -> [(Int, Int)] -> [Int] -> ([Int], [Int])
visitaInProfonditaComponente = visitaInProfondita aggiungiInTesta

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

{-
    Calcola ordine di fine globale del grafo.
-}
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')

-------------------------------------------------
-- KOSARAJU
-------------------------------------------------

{-
```

```haskell
    Implementazione algoritmo di Kosaraju per SCC.
-}
kosaraju :: [Int] -> [(Int, Int)] -> [[Int]]
kosaraju vertici archi =
    componenti
  where
    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
-------------------------------------------------

{- Restituisce indice SCC contenente un vertice. -}
indiceSCC :: Int -> [[Int]] -> Int
indiceSCC v sccs =
    case [i | (i, comp) <- zip [0 ..] sccs , v `elem` comp] of
        (i:_) -> i
        []        -> error ("Errore interno: vertice non trovato in alcuna SCC: " ++ show v)

{- Costruisce grafo compresso delle SCC. -}
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
        ]

{- Calcola grado entrante di una SCC. -}
gradoEntrante :: Int -> [(Int, Int)] -> Int
gradoEntrante c archi =
    length [() | (_, y) <- archi , y == c]

{- Conta SCC con grado entrante zero (esclusa quella di partenza). -}
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 invernale A.A 2025/2026            #
% #                          di Andrea Pedini                          #
% #                        Matricola: 322918                           #
% #                        e Matteo Fraternali                         #
% #                        Matricola: 316637                           #
% ######################################################################

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

/*
    Predicato di avvio del programma.
    Legge il grafo dal file 'input.txt', ne valida il contenuto
    e avvia l'esecuzione del programma principale.
*/
main :-
    leggi_grafo_sicuro('input.txt', Risultato),
    gestisci_risultato(Risultato).

/* Gestisce il risultato della lettura e validazione del file. */
gestisci_risultato(errore) :-
    nl, stampa_separatore,
    write('Errore nel file di input.'), nl,
    write('Controllare formato, duplicati e contenuto del grafo.'), nl,
    stampa_separatore, nl, !, fail.

gestisci_risultato(ok(Grafo)) :-
    esegui_programma(Grafo).

/* Predicato principale che coordina l'esecuzione. */
esegui_programma(Grafo) :-
    vertici(Grafo, Vertici),
    archi(Grafo, Archi),
    kosaraju(Grafo, SCCs),

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

    leggi_vertice_valido(Vertici, VerticeScelto),
    scc_di_vertice(VerticeScelto, SCCs, SCCPartenza),

    findall(S, (membro(S, SCCs), S \= SCCPartenza,
                grado_entrante(Grafo, SCCs, S, 0)), SCCZeroIn),
    length(SCCZeroIn, Conteggio),

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

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% LETTURA E VALIDAZIONE DEL GRAFO
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

/*
    Legge il grafo da file in modo sicuro.
    Effettua controlli su formato, duplicati e coerenza archi.
*/
leggi_grafo_sicuro(File, ok(grafo(Vertici, Archi))) :-
    catch(open(File, read, Stream), _, fail),

    leggi_termine_sicuro(Stream, Vertici),
    is_list(Vertici),
    Vertici \= [],
    lista_vertici_valida(Vertici),
    vertici_senza_duplicati(Vertici),

    leggi_termine_sicuro(Stream, Archi),
    is_list(Archi),
    lista_archi_valida(Archi),

    close(Stream),
    archi_validi(Archi, Vertici), !.

leggi_grafo_sicuro(_, errore).

/* Verifica che la lista dei vertici non contenga duplicati. */
vertici_senza_duplicati([]).
vertici_senza_duplicati([H|T]) :-
    \+ membro(H, T),
    vertici_senza_duplicati(T).

/* Legge un termine Prolog da stream intercettando eccezioni. */
leggi_termine_sicuro(Stream, Termine) :-
    leggi_linea(Stream, Line),
```

```
    atom_concat(Line, '.', LineConPunto),
    catch(read_from_atom(LineConPunto, Termine), _, fail).

/* Verifica che una lista contenga solo interi. */
lista_vertici_valida([]).
lista_vertici_valida([H|T]) :-
    integer(H),
    lista_vertici_valida(T).

/* Verifica che una lista contenga solo coppie (X,Y) di interi. */
lista_archi_valida([]).
lista_archi_valida([(X,Y)|T]) :-
    integer(X),
    integer(Y),
    lista_archi_valida(T).

/* Verifica che tutti gli archi usino solo vertici esistenti. */
archi_validi([], _).
archi_validi([(X,Y)|Resto], Vertici) :-
    membro(X, Vertici),
    membro(Y, Vertici),
    archi_validi(Resto, Vertici).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% LETTURA DA STREAM
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

/*
    Legge una riga da uno stream.
    Parametri:
    - Stream: stream di input
    - Linea: atomo contenente la riga letta
*/
leggi_linea(Stream, Linea) :-
    get_char(Stream, Char),
    leggi_linea_aux(Stream, Char, Caratteri),
    atom_chars(Linea, Caratteri).

/*
    Predicato ausiliario che legge carattere per carattere
    fino a newline o fine file.
*/
leggi_linea_aux(_, end_of_file, []) :- !.
leggi_linea_aux(_, '\n', []) :- !.
leggi_linea_aux(Stream, Char, [Char|Resto]) :-
    get_char(Stream, Next),
    leggi_linea_aux(Stream, Next, Resto).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% INPUT UTENTE
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
/*
    Richiede all'utente un vertice valido.
    Garantisce che sia intero e presente nel grafo.
*/
leggi_vertice_valido(Vertici, Vertice) :-
    write('Inserisci il vertice di partenza (tra '),
    write(Vertici), write('):'), nl,
    leggi_numero_valido(N),
    (membro(N, Vertici) -> Vertice = N
    ; write('Vertice non valido! Riprova.'), nl,
      leggi_vertice_valido(Vertici, Vertice)).

/* Legge un numero intero valido da input standard. */
leggi_numero_valido(N) :-
    leggi_linea(user_input, Line),
    (leggi_e_valida_numero(Line, N) -> true
    ; write('Input non valido! Inserisci un numero intero.'), nl,
      leggi_numero_valido(N)).

/* Verifica che una stringa rappresenti esattamente un intero. */
leggi_e_valida_numero(Line, N) :-
    atom_concat(Line, '.', LineConPunto),
    catch(read_from_atom(LineConPunto, Termine), _, fail),
    numero_valido(Termine, N).

/* Controlla che il termine sia un numero intero puro. */
numero_valido(N, N) :- integer(N).
numero_valido(Termine, _) :-
    \+ integer(Termine),
    write('Errore: "'), write(Termine), write('" non e'' un numero intero.'), nl,
    fail.

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

/* Stampa una linea di separazione grafica. */
stampa_separatore :- write('======================================'), nl.

/* Stampa una riga separatrice corta. */
stampa_riga :- write('--------------------------------------'), nl.

/* Stampa tutte le SCC numerandole. */
stampa_scc_numerate([], _).
stampa_scc_numerate([S|Resto], Indice) :-
    write('SCC '), write(Indice), write(': '), write(S), nl,
    Next is Indice + 1,
    stampa_scc_numerate(Resto, Next).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%% PREDICATI DI BASE SUL GRAFO
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

/* Verifica l'appartenenza di un elemento a una lista. */
membro(X, [X|_]).
membro(X, [_|Resto]) :- membro(X, Resto).

/* Estrae i vertici dal termine grafo. */
vertici(grafo(N,_), N).

/* Estrae gli archi dal termine grafo. */
archi(grafo(_,A), A).

/* Verifica se esiste arco orientato X -> Y. */
adiacente(grafo(_,A), X, Y) :-
    membro((X,Y), A).

/* Restituisce tutti i vertici raggiungibili con un arco. */
adiacenti(Grafo, Vertice, Adiacenti) :-
    findall(Y, adiacente(Grafo, Vertice, Y), Adiacenti).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% VISITA IN PROFONDITÀ
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

/* Implementazione generica DFS parametrizzata. */
visitaInProfondita(_, _, Vertice, Visitati, Visitati, []) :-
    membro(Vertice, Visitati), !.

visitaInProfondita(Combina, Grafo, Vertice, Visitati, VisitatiFinali, Risultato) :-
    \+ membro(Vertice, Visitati),
    adiacenti(Grafo, Vertice, Vicini),
    visitaInProfondita_lista(Combina, Grafo, Vicini,
                             [Vertice|Visitati], VisitatiParziali, RisultatiFigli),
    call(Combina, Vertice, RisultatiFigli, Risultato),
    VisitatiFinali = VisitatiParziali.

/* Visita in profondità su lista di vertici. */
visitaInProfondita_lista(_, _, [], Visitati, Visitati, []).
visitaInProfondita_lista(Combina, Grafo, [H|T], Visitati, VisitatiFinali, Risultato) :-
    visitaInProfondita(Combina, Grafo, H, Visitati, Visitati1, R1),
    visitaInProfondita_lista(Combina, Grafo, T, Visitati1, VisitatiFinali, R2),
    append(R1, R2, Risultato).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% VISITE SPECIALIZZATE
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

/* Visita in profondità che costruisce ordine di completamento. */
visitaInProfondita_ordine(Grafo, Vertice, Visitati, VisitatiFinali, Ordine) :-
    visitaInProfondita(combina_fine, Grafo, Vertice, Visitati, VisitatiFinali, Ordine).
```

```
/* Visita in profondità che costruisce una SCC. */
visitaInProfondita_scc(Grafo, Vertice, Visitati, VisitatiFinali, Componente) :-
    visitaInProfondita(combina_testa, Grafo, Vertice, Visitati, VisitatiFinali, Componente).

/* Visita in profondità globale su tutto il grafo. */
visitaInProfondita_grafo(Grafo, Ordine) :-
    vertici(Grafo, Vertici),
    visitaInProfondita_lista(combina_fine, Grafo, Vertici, [], _, Ordine).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% STRATEGIE DI COMBINAZIONE
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

/* Inserisce un vertice in coda alla lista. */
combina_fine(N, Lista, Risultato) :-
    append(Lista, [N], Risultato).

/* Inserisce un vertice in testa alla lista. */
combina_testa(N, Lista, [N|Lista]).

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

/* Costruisce il grafo trasposto invertendo tutti gli archi. */
trasposto(grafo(Vertici, Archi), grafo(Vertici, ArchiTrasposti)) :-
    trasponi_archi(Archi, ArchiTrasposti).

/* Inverte direzione di ogni arco. */
trasponi_archi([], []).
trasponi_archi([(X,Y)|Resto], [(Y,X)|Trasposti]) :-
    trasponi_archi(Resto, Trasposti).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% ALGORITMO DI KOSARAJU
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

/* Implementazione algoritmo di Kosaraju. */
kosaraju(Grafo, SCCs) :-
    visitaInProfondita_grafo(Grafo, Ordine),
    reverse(Ordine, OrdineInverso),
    trasposto(Grafo, GrafoTrasposto),
    kosaraju_visita(GrafoTrasposto, OrdineInverso, [], SCCs).

/* Costruzione progressiva delle SCC. */
kosaraju_visita(_, [], _, []).
kosaraju_visita(Grafo, [Vertice|Resto], Visitati, SCCs) :-
    membro(Vertice, Visitati), !,
    kosaraju_visita(Grafo, Resto, Visitati, SCCs).
```

```
kosaraju_visita(Grafo, [Vertice|Resto], Visitati, [SCC|Altre]) :-
    visitaInProfondita_scc(Grafo, Vertice, Visitati, Visitati1, SCC),
    kosaraju_visita(Grafo, Resto, Visitati1, Altre).

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

/* Trova la SCC che contiene un vertice. */
scc_di_vertice(Vertice, [S|_], S) :-
    membro(Vertice, S), !.
scc_di_vertice(Vertice, [_|Resto], S) :-
    scc_di_vertice(Vertice, Resto, S).

/* Verifica esistenza arco tra SCC diverse. */
arco_scc(Grafo, SCCs, S1, S2) :-
    archi(Grafo, Archi),
    membro((X,Y), Archi),
    scc_di_vertice(X, SCCs, S1),
    scc_di_vertice(Y, SCCs, S2),
    S1 \= S2.

/* Calcola grado entrante di una SCC.*/
grado_entrante(Grafo, SCCs, SCC, Grado) :-
    findall(1, arco_scc(Grafo, SCCs, _, SCC), Lista),
    length(Lista, 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