Compute SCCs in Flix Functional Lanugage

Manoj Kumar (manoj18.iitd@gmail.com)

January 3, 2021

Abstract

We consider the problem of finding all *strongly connected components* (SCCs) in a dependency graph of predicate symbols.
I have presented two implementations for this problem.

1. Programming with Datalog Constraints in Flix[1].
2. Kosaraju's Algorithm[2] in recursive functional language.

Keywords: Programming Languages, First-Class Datalog Constraints, Fixpoints, Datalog, Kosaraju's Algorithm for computing SCCs in a directed Graph, Functional Programming, Logic Programming

Problem Statement

For a Datalog program, we need to compute the strongly connected components of dependencies between predicate symbols.
For example: given the Datalog program:

```
A(x) :- B(x).
B(x) :- A(x), D(x).
C(x) :- A(x), B(x).
```

The dependency graph is the directed graph with edges :

```
A <- B
B <- A
B <- D
C <- A
C <- B
```

The SCC of this graph is {A, B}, {D} and {C}. This means that we can compute the two rules first and then the third rule last.
If a predicate does not depend on any other predicate, it is considered as a separated SCC.

[1] flix: Developed at Aarhus University and the University of Waterloo. https://flix.dev/
[2] Kosaraju's algorithm: Wikipedia: The free encyclopedia: https://en.wikipedia.org/

Implementation with Datalog Constraints

1. The following two predicate symbols have been used in implementation:
   (a) Edge(x, y) : For each pair (x,y) of the edges list, we created the facts Edge(x,y).
   (b) Reachable(x,y) : This rule gives true if

```
Reachable(x, y) :- Edge(x, y).
Reachable(x, y) :- Reachable(x, z), Edge(z, y).
```

2. Further, a list allNodes is created that contains all nodes that present in the given list edges.
3. We maintain a set *visited* to keep track of all visited nodes.
4. For each unvisited node x of the *allNodes*, we find a set of all such previous unvisited nodes y, for which *Reachable(x, y)* and *Reachable(y, x)*, both are true. We mark all these visited and repeat until we visit all nodes.

we keep storing all the SCCs and finally return a list of all SCCs.

Running Time Complexity:
Worst case time complexity : $O(V^2)$ where V is total no. of nodes in the graph.

Implementation using Kosaraju's Algorithm

Algorithmic Approach:

1. Create an empty stack 'S' and do DFS traversal of a graph. In DFS traversal, after calling recursive DFS for adjacent vertices of a vertex, push the vertex to stack.
2. Reverse directions of all edges to obtain the transpose graph.
3. One by one pop a symbol from S while S is not empty. Let the popped symbol be 'v'. Take v as source and do DFS. The DFS starting from v will visit all symbols of the strongly connected component of v.

Implementation Approach:
I implemented a function *scc* and all of its helper functions:

```
def scc(edges: List[(String, String)] : List[Set[String]]
```

It takes all the edges (directed from second to first) as input and returns a list of sets where each set is a separate SCC for the dependency graph.
Whole implementation is divided in two phases :

1. **Phase One:**
   (a) Create a map *adjList* with *key*:String and *value*:List of Strings.
   (b) initialize a stack *'stack'* with is basically a list of Strings.
   (c) initialize an empty set *visited*.
   (d) extract all nodes of the graph from the list *edges* to a list of String: *nodes*.
   (e) DFS traversal in entire graph and fill the *stack*.
2. **Phase Two:**
   (a) Create the transpose graph by inverting all edges. (List[(String, String)])
   (b) Create a adjacency list for the transpose edges.
   (c) initialize visited as empty set.
   (d) pop elements from the stack, if current element is not yet visited, run dfs and get the SCC corresponding to the current element, else move to the next element of the stack.
   (e) When the stack is empty, we have got a list of SCCs.

Finally *scc* function returns a list of all *Sets* of SCCs.

Running Time Complexity:

The algorithm follows Kosaraju's Approach, Hence it is $O(V + E)$. Where V is total number of nodes and E is total number of edges.

Testing

Input Format: In both implementations, function *scc* takes a list of (String, String) in argument.
Example:

```
edges = ("A", "B")::("B", "A")::("B", "D")::("C", "A")::("C", "B")::Nil
then the dependency edges will be : A <- B, B <- A, B <- D, C <- A, C <- B
```

Output Format: function scc returns a List of Sets where each set represents a separate SCC.
For the above example, program will return: [{D}, {A, B}, {C}]
Following Steps has been followed for testing of the implementation:

Extensive and Intensive Testing : Program tested on various extensive and corner cases.

```
Input : [(A, A)]
Output : [{A}]

Input : [(A, B)]
Output : [{A},{B}]
```

```
Input : [(A, B), (B, C)]
Output : [{A}, {B}, {C}]

Input : [(A, B), (B, A)]
Output: [{A, B}]

Input: [(A, B), (B, A), (B, D), (C, A), (C, B)]
Output: [{D}, {A, B}, {C}]
```

Thanks!

Assignment given by:

Prof. Magnus Madsen, Aarhus University

Implemented by:

Manoj Kumar, IIT Delhi