Course Structure

1. Introduction
   Goal, Overview, Motivation, Notions,
   Lab 1: Set implementation
2. Testing
3. O-Calculus
4. Measurements
5. Profiling
6. Application profiles
7. Graphs / `grail`
8. Competition results I
9. Grail results
10. Competition results II

Motivation

- Concluding practical example:
  - Find/remove performance bugs in a larger application (graph library `grail`)
  - Use analytical and measuring techniques
- As side effects
  - Refresh some algorithms
  - Become aware of the `grail` graph library for reuse in own projects

Graphs

Directed graph:

- Let V be a non-empty set and let E: $V \times V$ be a binary relation. The pair $G = (V, E)$ is then called a *directed graph*, where V is the set of *vertices* (or *nodes*) and E is the set of *edges* (or *arcs*).

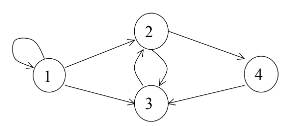
Undirected graph:

- A graph $G = (N, E)$ is an *undirected graph* iff it holds $(v_1, v_2) \in E \Rightarrow (v_2, v_1) \in E$.

We can attach *labels* to nodes and arcs.

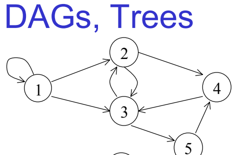
Loops, Paths, and Cycles

- Let (u, v) be an arc in a directed graph. If $u = v$ then the arc (u, v) is called a *loop*.
- A *path* in a graph $G = (V, E)$ is a list of nodes $(v_1, \ldots v_n)$ such that: $\forall i = 1..n\text{-}1$: $(v_i, v_{i+1}) \in E$. The *length* of the path is $n\text{-}1$.
- A path in a graph that begins and ends at the same node is called a *cycle*. The *length of the cycle* is the length of the path. A cycle is *simple* if the only node that appears twice in the path is the first node.
- Example:

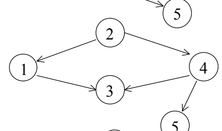


Cyclic graphs, DAGs, Trees

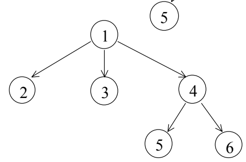
- A graph is *cyclic* if it has at least one cycle. A graph that is not cyclic is called *acyclic*.



- A directed acyclic graph is called a *DAG*.



- A DAG is called a *tree* if $\exists\, n \in V$ (the *root* of the tree) such that for each node n' there is a unique path from n to n'.

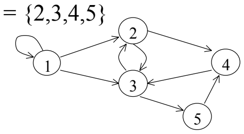


Topological Order

- A *topological order* of a DAG G is an order of all the nodes in G such that for each arc (u, v) of G the source node u precedes the target node v in that order

Strongly Connected Components

- Let n and n' be two nodes of a directed graph G. Node n is *reachable* from node n' iff there is a path from n' to n in G.
- A *strongly connected component* of a directed graph $G = (V, E)$ is a set of nodes $C \subseteq V$ such that for every pair of nodes $(u, v) \in C$, u is reachable from v and v is reachable from u.
- Example: $C = \{2,3,4,5\}$



Depth-first search

Used in many graph algorithms, e.g. for

- Computing a topological order of a DAG
- Computing the strongly connected components of a graph

Depth-first search

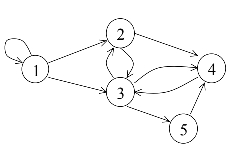
Mark all nodes unvisited.
Loop:
  Choose any unvisited node s, as start node
  Call DFS(s)
Until all nodes are visited

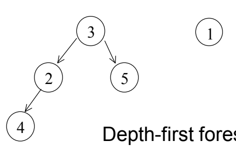
DFS(n):
  Mark n visited
  For all successors m, of n
    If m is unvisited:
     Recursively call DFS(m)

Classification

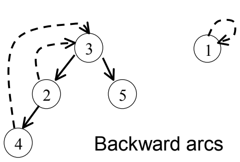
- With respect to a given DFS of the graph G the arcs of G can be divided in four groups:
  - *Tree arcs*: arcs (u, v) such that DFS(v) is called by DFS(v).
  - *Forward arcs*: arcs (u, v) such that v is a proper descendant of u but not a child of u in the tree defined by the tree arcs.
  - *Backward arcs*: arcs (u, v) such that v is an ancestor of u in the tree defined by the tree arcs. A loop (u, u) is a backward arc.
  - *Cross arcs*: arcs (u, v) such that v is neither an ancestor nor a descendant of u.

Example

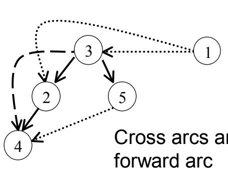




Depth-first forest



Backward arcs



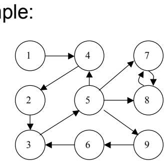
Cross arcs and forward arc

Strongly Connected Components
[Tarjan's and Gabow's Algorithm]

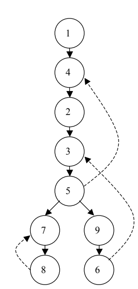
Idea: Each backward arc in the DFS -forest of graph G indicates a cycle in G. All nodes of a cycle are strongly connected.

Example:

G



DFS - forest of graph G



Tarjan's and Gabow's Algorithm

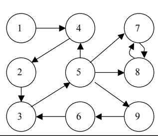
Algorithm:

```
scc_number := 0
mark all nodes in G as 'not visited'
for each node v in G that is not visited, do
    scc( v )
end for
```


Tarjan's and Gabow's Algorithm (cont'd)

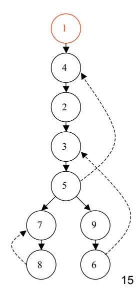
```
scc( v ) :
   lowlink(v) := number(v) := ++scc_number
   push( v )
   for all successors w of v do
        if w is not visited then        // v->w is a tree arc
                scc( w )
                lowlink(v) := min( lowlink(v), lowlink(w) )
        elsif number(w) < number(v) then    // v->w is backwards arc
                if in_stack(w) then
                        lowlink(v) := min( lowlink(v), number(w) )
                end if
        end if
   end for
   if lowlink(v) = number(v) then       // next scc found
        while w := top_of_stack_node; number(w) >= number(v) do
                pop(w)
        end while
   end if
```


Example

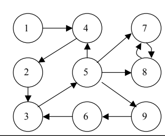


	1	2	3	4	5	6	7	8	9
num	1								
low	1								
id									

Stack: 1

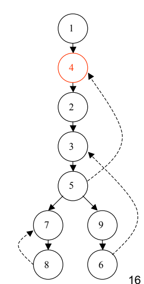


Example

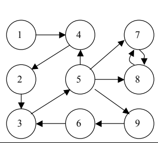


	1	2	3	4	5	6	7	8	9
num	1			2					
low	1			2					
id									

Stack: 1 4

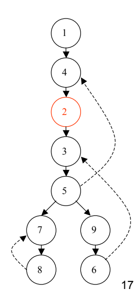


Example

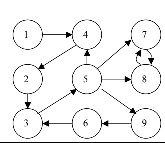


	1	2	3	4	5	6	7	8	9
num	1	3		2					
low	1	3		2					
id									

Stack: 1 4 2

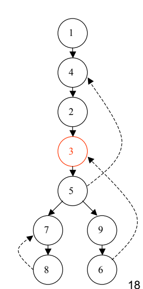


Example

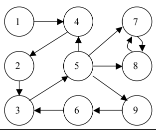


	1	2	3	4	5	6	7	8	9
num	1	3	4	2					
low	1	3	4	2					
id									

Stack: 1 4 2 3

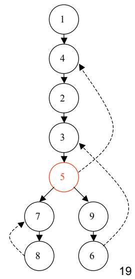


Example



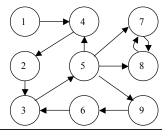
	1	2	3	4	5	6	7	8	9
num	1	3	4	2	5				
low	1	3	4	2	5				
id									

Stack: 1 4 2 3 5



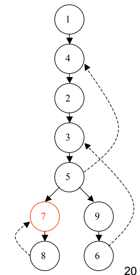
19

Example



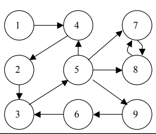
	1	2	3	4	5	6	7	8	9
num	1	3	4	2	5		6		
low	1	3	4	2	5		6		
id									

Stack: 1 4 2 3 5 7



20

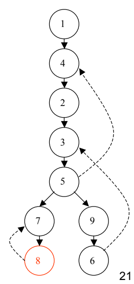
Example



min

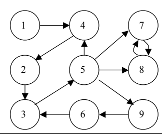
	1	2	3	4	5	6	7	8	9
num	1	3	4	2	5		6	7	
low	1	3	4	2	5		6	7	
id									

Stack: 1 4 2 3 5 7 8



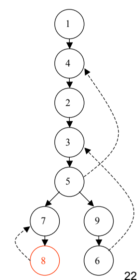
21

Example



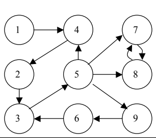
	1	2	3	4	5	6	7	8	9
num	1	3	4	2	5		6	7	
low	1	3	4	2	5		6	6	
id									

Stack: 1 4 2 3 5 7 8



22

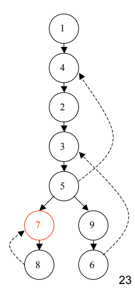
Example



min

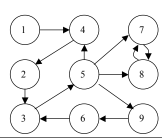
	1	2	3	4	5	6	7	8	9
num	1	3	4	2	5		6	7	
low	1	3	4	2	5		6	6	
id									

Stack: 1 4 2 3 5 7 8



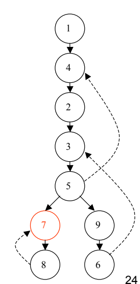
23

Example



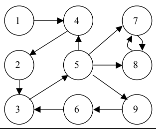
	1	2	3	4	5	6	7	8	9
num	1	3	4	2	5		6	7	
low	1	3	4	2	5		6	6	
id									

Stack: 1 4 2 3 5 7 8



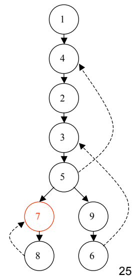
24

Example



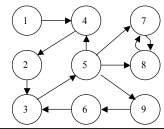
	1	2	3	4	5	6	7	8	9
num	1	3	4	2	5		6	7	
low	1	3	4	2	5		6	6	
id								1	

Stack: 1 4 2 3 5 7



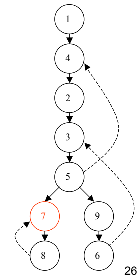
25

Example



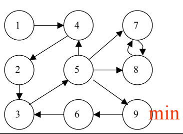
	1	2	3	4	5	6	7	8	9
num	1	3	4	2	5		6	7	
low	1	3	4	2	5		6	6	
id							1	1	

Stack: 1 4 2 3 5



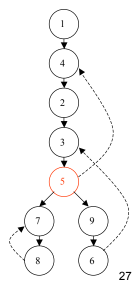
26

Example



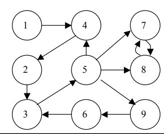
	1	2	3	4	5	6	7	8	9
num	1	3	4	2	5		6	7	
low	1	3	4	2	5		6	6	
id							1	1	

Stack: 1 4 2 3 5



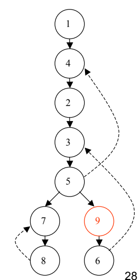
27

Example



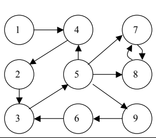
	1	2	3	4	5	6	7	8	9
num	1	3	4	2	5		6	7	8
low	1	3	4	2	5		6	6	8
id							1	1	

Stack: 1 4 2 3 5 9



28

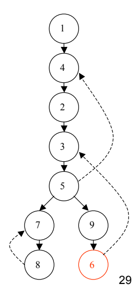
Example



min

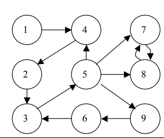
	1	2	3	4	5	6	7	8	9
num	1	3	4	2	5	9	6	7	8
low	1	3	4	2	5	9	6	6	8
id							1	1	

Stack: 1 4 2 3 5 9 6



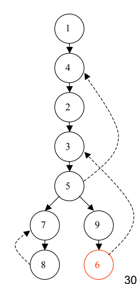
29

Example



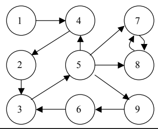
	1	2	3	4	5	6	7	8	9
num	1	3	4	2	5	9	6	7	8
low	1	3	4	2	5	4	6	6	8
id							1	1	

Stack: 1 4 2 3 5 9 6



30

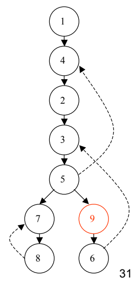
Example



min

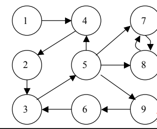
	1	2	3	4	5	6	7	8	9
num	1	3	4	2	5	9	6	7	8
low	1	3	4	2	5	4	6	6	8
id							1	1	

Stack: 1 4 2 3 5 9 6



31

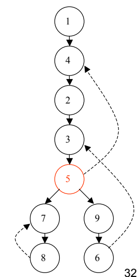
Example



min

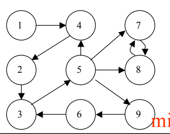
	1	2	3	4	5	6	7	8	9
num	1	3	4	2	5	9	6	7	8
low	1	3	4	2	5	4	6	6	4
id							1	1	

Stack: 1 4 2 3 5 9 6



32

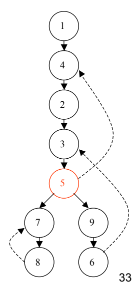
Example



min

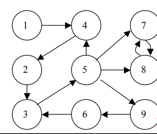
	1	2	3	4	5	6	7	8	9
num	1	3	4	2	5	9	6	7	8
low	1	3	4	2	5	4	6	6	4
id							1	1	

Stack: 1 4 2 3 5 9 6



33

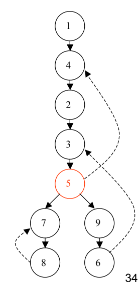
Example



min

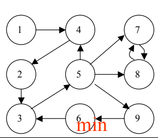
	1	2	3	4	5	6	7	8	9
num	1	3	4	2	5	9	6	7	8
low	1	3	4	2	4	4	6	6	4
id							1	1	

Stack: 1 4 2 3 5 9 6



34

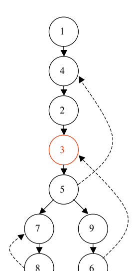
Example



min

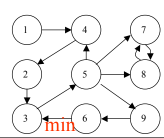
	1	2	3	4	5	6	7	8	9
num	1	3	4	2	5	9	6	7	8
low	1	3	4	2	2	4	6	6	4
id							1	1	

Stack: 1 4 2 3 5 9 6



35

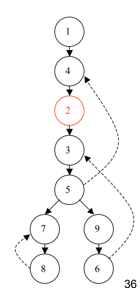
Example



min

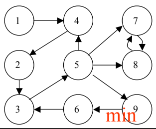
	1	2	3	4	5	6	7	8	9
num	1	3	4	2	5	9	6	7	8
low	1	3	2	2	2	4	6	6	4
id							1	1	

Stack: 1 4 2 3 5 9 6



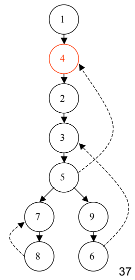
36

Example



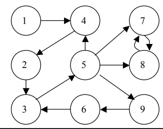
	1	2	3	4	5	6	7	8	9
num	1	3	4	2	5	9	6	7	8
low	1	2	2	2	2	4	6	6	4
id							1	1	

Stack: 1 4 2 3 5 9 6



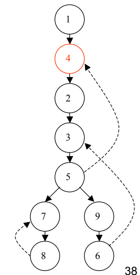
37

Example



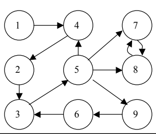
	1	2	3	4	5	6	7	8	9
num	1	3	4	2	5	9	6	7	8
low	1	2	2	2	2	4	6	6	4
id						2	1	1	

Stack: 1 4 2 3 5 9



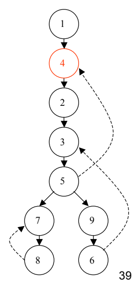
38

Example



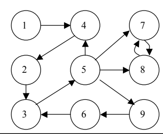
	1	2	3	4	5	6	7	8	9
num	1	3	4	2	5	9	6	7	8
low	1	2	2	2	2	4	6	6	4
id						2	1	1	2

Stack: 1 4 2 3 5



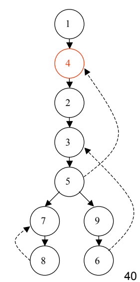
39

Example



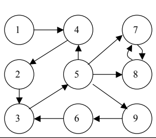
	1	2	3	4	5	6	7	8	9
num	1	3	4	2	5	9	6	7	8
low	1	2	2	2	2	4	6	6	4
id					2	2	1	1	2

Stack: 1 4 2 3



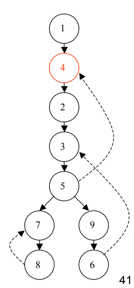
40

Example



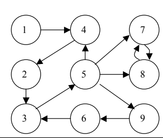
	1	2	3	4	5	6	7	8	9
num	1	3	4	2	5	9	6	7	8
low	1	2	2	2	2	4	6	6	4
id			2		2	2	1	1	2

Stack: 1 4 2



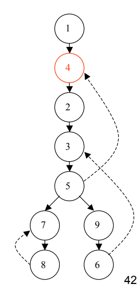
41

Example



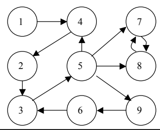
	1	2	3	4	5	6	7	8	9
num	1	3	4	2	5	9	6	7	8
low	1	2	2	2	2	4	6	6	4
id		2	2		2	2	1	1	2

Stack: 1 4



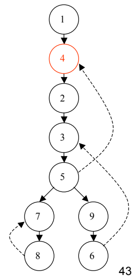
42

Example

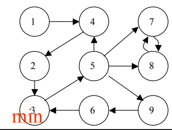


	1	2	3	4	5	6	7	8	9
num	1	3	4	2	5	9	6	7	8
low	1	2	2	2	2	4	6	6	4
id		2	2	4	2	2	1	1	2

Stack: 1

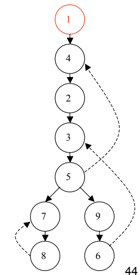


Example

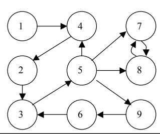


	1	2	3	4	5	6	7	8	9
num	1	3	4	2	5	9	6	7	8
low	1	2	2	2	2	4	6	6	4
id		2	2	4	2	2	1	1	2

Stack: 1

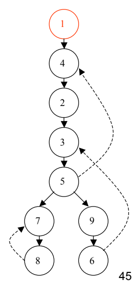


Example



	1	2	3	4	5	6	7	8	9
num	1	3	4	2	5	9	6	7	8
low	1	2	2	2	2	4	6	6	4
id	3	2	2	4	2	2	1	1	2

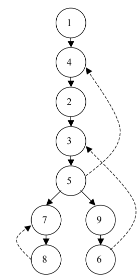
Stack:



Loop Tree

- Given a Control Flow Graph (CFG) of a program, with node being basic blocks and edges control flow dependencies
- Problem in Data Flow Analysis: find a traversal order of the nodes corresponding to the execution order of the basic blocks
- Solution: *compute a loop tree*.

CFG



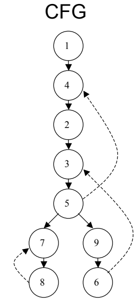
Loop Tree Computation of $CFG=(V, E)$

Create a root node r
Call LoopTree(CFG , r)

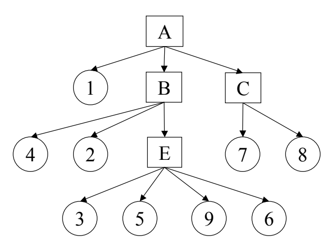
LoopTree(G,n):
- $C = \{C_1 \dots C_k\} = \mathrm{SCC}(G)$
- $G' = (C, E')$ with $(C_i, C_j) \in E'$ iff $(v_i, v_j) \in E \wedge v_i \in C_i \wedge v_j \in C_j$
- For all nodes C_i in topologic order of nodes in G'
  - If $|C_i| = 1$ (single node SCC) add $v_i \in C_i$ as i-th child of n
  - Otherwise
    - Create a new root node r_i,
    - add r_i as i-th child of n
    - $G_i = (C_i, E_i)$ with $(u, v) \in E_i$ iff $(u, v) \in E$
    - Remove a single back-edge from G_i
    - Recursively call LoopTree(G_i ,r_i)

Example

CFG



Loop Tree



Assignment 4: Improving `LoopTree`

- Improve the `LoopTree` algorithm and all parts of the `grail` package upon which it depends.
  - Graphs with a large depth results in stack-overflow exception due to the usage of recursive algorithms (e.g. `depth-first` and `scc`). Your task is to re-implement all those algorithms without using recursion.
  - We can detect very poor performance when applying the loop tree algorithm on larger graphs. Your task is to identify (and repair) the bottlenecks causing this poor performance.
  - The result of this assignment is:
    1. An updated version of the Grail package
    2. A test program that verifies the changes.
    3. A written documentation of all your changes in the graph package with a brief motivation.

49