Loop analysis

Program representations:

- Control Flow Graph (seen already)
- Spanning trees
- Strongly connected regions
- Identifying loops
- Reducible CFGs

Spanning Tree Algorithm

Span(Root)

procedure Span(v)
ST.addVertex(v)
for w : v → w ∈ CFG do
if w /∈ ST then
ST.addEdge(v → w)
Span(w)
endfor
end Span

For convenience, assume all V are reachable from Entry:

∀v ∈ V : Entry →* v

Entry /∈* v ⇒ add edge Entry → v

Spanning Trees

Given CFG ⟨V, E, Entry, Exit⟩:
- V: vertices v
- E: edges v → w
- Entry ∈ V: unique entry
- Exit ∈ V: unique exit

construct spanning tree ST ⟨VT, ET, RootT, ExitT⟩:
- VT = V
- ET ⊆ E
- RootT = Entry
- ExitT = Exit

Given ST, each edge v → w in the CFG can be partitioned as follows:
1. Tree: v → w in both CFG and ST
2. Advancing: v → w is not in ST, but w is a descendant of v in ST
3. Back: v = w or w is a proper ancestor of v in ST
4. Cross: w neither an ancestor nor descendant of v in ST

Spanning Tree: Example
Spanning Edge Identification

\[ i \leftarrow 0 \]
\[ \text{DFST}(\text{Root}) \]

\[ \text{procedure DFST}(v) \]
\[ \text{ST}.\text{addVertex}(v) \]
\[ v.\text{num} \leftarrow i++ \]
\[ v.\text{inStack} \leftarrow \text{true} \]
\[ \text{for } w : v \rightarrow w \in \text{CFG} \text{ do} \]
\[ \text{if } w \notin \text{ST} \text{ then} \]
\[ v \rightarrow w \text{ is a tree edge} \]
\[ \text{ST}.\text{addEdge}(v \rightarrow w) \]
\[ \text{DFST}(w) \]
\[ \text{else if } w.\text{inStack} \]
\[ v \rightarrow w \text{ is a back edge} \]
\[ \text{else if } w \in \text{ST} \text{ and } w.\text{num} > v.\text{num} \]
\[ v \rightarrow w \text{ is a forward edge} \]
\[ \text{else} \]
\[ v \rightarrow w \text{ is a cross edge} \]
\[ \text{end for} \]
\[ v.\text{inStack} \leftarrow \text{false} \]
\[ \text{end DFST} \]

Cycles: Strongly Connected Regions (SCR)

\[ \forall s_1, s_2 \in S. \text{if } S \text{ is a cycle then } s_1 \rightarrow^* s_2 \text{ and } s_2 \rightarrow^* s_1 \]

Compute maximal SCR on a directed graph.


- Uses a depth-first spanning tree
  - left-to-right pre-order number in *Number*
- Tracks the lowest numbered \( v \) to which each vertex has a path in *lowlink*
- Determines a number for SCR to which \( v \) belongs.

Tarjan’s maximal SCR algorithm

\[ i \leftarrow 0 \]
\[ \text{for } v \in V \text{ do } v.\text{lowlink} \leftarrow 0; v.\text{num} \leftarrow 0 \text{ endfor} \]
\[ \text{scr} \leftarrow 0 \]
\[ \text{stack} \leftarrow \text{empty} \]
\[ \text{for } v \in V \text{ do} \]
\[ \text{if } v.\text{num} = 0 \text{ then Tarjan}(v) \]
\[ \text{endfor} \]
Tarjan’s maximal SCR algorithm (cont.)

procedure Tarjan(v)
  i++; v.num ← i; v.lowlink ← i
  stack.push(v)
  for w : v → w do
    if w.num = 0 then
      Tarjan(w)
      v.lowlink ← min(v.lowlink, w.lowlink)
    else if w ∈ stack then
      v.lowlink ← min(v.lowlink, w.lowlink)
  endfor
  if v.lowlink = v.num then
    scr++
    repeat
      w ← stack.pop()
      w.scr ← scr
    until w = v
  end Tarjan

Identifying Loops and Loop Headers

• DFST does not find a unique header in irreducible graphs
• SCRs do not differentiate inner loops

Natural Loop

• Single entry, header dominates all vertices in loop.
  dominates: v dom w ⇔ every path Entry → ∗ w passes through v.
• There is at least one path from the header to itself.
• All vertices and edges on a path from the header to any back edges to the header are in the loop.
• Two natural loops are either entirely disjoint, or one is a proper subset of the other.
Improperly Nested Loops

One loop or two?
Inner loop?
Outer loop?

Natural Loop Example

Natural Loop Algorithm

Given a back edge $(t \rightarrow h)$:
- loop.addVertex($h$)
- loop.addEdge($t \rightarrow h$)
- insert($t$)

procedure insert($v$)
if $v \notin$ loop then
  loop.addVertex($v$)
for $p \rightarrow v$ do
  loop.addEdge($p \rightarrow v$)
  insert($p$)
endfor
end insert

Reducible Control Flow Graphs

Intuitively, if all loops are single entry, the CFG is reducible.

More formally:

- Given a spanning tree, for every back edge in the CFG, the head dominates the tail (i.e., you cannot execute the tail without executing the head first).
Next Time

Dominator and Control Dependences