CSE 756, Project 6: Control-Flow Analysis

You need to implement a builder for a control-flow graph (CFG) as well as an analyzer of loops in this graph. The input will be a C program that is equivalent to three-address code. This program is the output from Project 5. The three-address instructions will be represented by Sage AST nodes. The output of your project will be some statistics about the CFG and its loops. The project is due by **May 29 (Tuesday)**, 11:59 pm.

Input Language

Assume that the input program is produced by a correct implementation of Project 5, which itself was given some input C program that satisfies all restrictions described in Project 5. Recall that a correct implementation of Project 5 must produce a valid C program in which only the following statements are allowed:

- Assignments: `x = y op z; x = op y; x = y; x = y[z]; x[y] = z;` as discussed in class
- Conditional jumps: `if (x relop y) goto L;`
- Unconditional jumps: `goto L;`
- Returns: `return x;` where `x` is a program variable, a temporary variable, or a constant
- Null statements (Section 6.8.3): just a semicolon, as in `_l2: ;` from the earlier example
- Each of the above statements could be labeled or unlabeled (but null statements should always be labeled). Note that a labeled statement could itself be labeled with another label.

When an input program is parsed by ROSE, the entire function body will be a single SgBasicBlock, without any nested SgBasicBlocks inside it. You should process this SgBasicBlock as if it were a sequence of three-address instructions, and should generate a CFG for it. For example, consider the following body of the function:

```
int _t1;
int x = -1, y;
_t1 = x + 1;
if (_t1 < 0) goto _l3;
goto _l2;
_l3: _l1: y = 1;
_l2: ;
_l11: _l12: _l13: _l14: x = 1;
if (x > 1) goto _l12;
```

The web page contains a visual representation of the Sage AST for this example, in file ast.jpg.

Control-Flow Graph

The CFG should be represented by some internal data structure – choose whatever you feel like. There should be an artificial ENTRY node which should be the predecessor of the first "real" CFG node. There should also be an artificial EXIT node which should be the successor of all `return` instructions. This means that the ENTRY node is a separate CFG node – it is not included in any basic block. Similarly, the EXIT node is not included in any basic block (even if there is only one `return`, and it is the only predecessor of the EXIT). When counting CFG nodes and edges, you should include the ENTRY and EXIT nodes, as well as all edges connecting them with the rest of the graph.

With the exception of the ENTRY and EXIT nodes, each other CFG node should conceptually represent a *basic block*, as defined in class. Specifically, inside each CFG node, you should store an ordered sequence of SgStatement* (i.e., pointers to statements in the Sage AST) representing the instructions in this basic block. You should ignore all declarations – they are not executable code and are not part of the CFG. For example, the declarations of **`_t1`**, **`x`**, and **`y`** from above should be completely ignored. Note that there is an initialization of **`x`** as part of its declaration; in a real compiler this will be represented by a three-address assignment instruction, but for the purposes of this project we will ignore the initializations that occur as part of declarations.

The sequence of StStatement* should also exclude any SgLabelStatement (see the AST in file ast.jpg on the web page for examples of these statements). Thus, the **only** elements of the SgStatement* sequence inside a basic block should be SgExprStatement (for assignment instructions), SgIfStmt (for conditional jump instructions), SgGotoStatement (for unconditional jump instructions), and SgReturnStmt (for return instructions). Note that null statements (**`;`**) are not part of basic blocks; in fact, they are not even part of the AST. To construct the basic blocks, use the algorithm described in class (lecture notes, slides 5-6). Note that, as part of this process, a **`return`** instruction is similar to an unconditional branch, and should be handled in a similar manner. After determining the basic blocks, construct all edges between basic blocks. The target of a SgGotoStatement can be obtained using get_label(). Do **not** try to use get_statement() for SgLabelStatement.

For the constructed CFG, you need to collect the following measurements: (1) number of CFG nodes including the two artificial nodes; (2) number of CFG edges; (3) total number of instructions inside all basic blocks.

Loops in the CFG

The CFG should be analyzed to find all loops. First, find all *back edges* (you can assume that the CFG is reducible). Next, for each back edge, find its *natural loop*. For simplicity, you can assume that every natural loop will be a separate CFG loop. For each loop, find its *nesting depth*: depth zero means that the loop is not nested inside any other loops, depth one means that the loop is surrounded by one outer loop, etc.

After the analysis of loops, you need to collect the following measurements: (1) number of loops; (2) total number of instructions inside all loops – but make sure that instructions inside an inner loop are counted only once, and are not double-counted due to an outer loop; (3) histogram of the loop nesting depth: number of loops that have nesting depth zero, number of loops that have nesting depth one, number of loops that have nesting depth two, etc.

Output

The result should be a single string value that is returned back to the **`main`** function provided for Project 5 (do **not** change this **`main`**; it will be used by the grader). This string should be **exactly** of the following form:

CFG NODES: …
CFG EDGES: …
TOTAL INSTRUCTIONS: …
LOOPS: …
INSTRUCTIONS IN LOOPS: …
LOOPS WITH DEPTH 0: …
LOOPS WITH DEPTH 1: …
LOOPS WITH DEPTH 2: …
….

The first three measurements are described above under "Control-Flow Graph". The remaining measurements are described above under "Loops in the CFG". The string should be **exactly** of this form, it order to allow the grader to run "diff" on your output.

Details

- Your submission must compile and run on **`stdlinux`**, using the ROSE installation in **`/class/cse756/…`**
- During the analysis of the AST, do **not** print directly to **`cout`**. The only printing to **`cout`** must be done by the provided **`main`**.
- modfft1.c: Your submission must work correctly on the three-address code (represented as a C program) that may be generated when any correct implementation of Project 5 is applied to modfft1.c.
- Depending on the input three-address code, it may be possible that some nodes in the CFG are not reachable from ENTRY. This is OK – you should still create such CFG nodes, any edges related to them, and count these nodes/edges in any measurements you report. Normally, the CFG would be "cleaned up" of such unreachable nodes before any CFG analyses, but for simplicity we will not do this in this project.

Project Submission

On or before 11:59 pm on the due date, you should submit a single file cfg.cpp containing all of your code; it should work with **`main`** from Project 5. Submit your project using "**`submit c756aa lab6 cgf.cpp`**" on **`stdsun`**.

If the time stamp on your electronic submission is **12:00 am on the next day or later**, you will receive 10% reduction per day, for up to three days. If your submission is later than 3 days after the deadline, it will not be accepted and you will receive zero points for this project. If you resubmit your project, this will override any previous submissions and only **the latest** submission will be considered – **resubmit at your own risk**.

Academic Integrity

The project you submit must be your own work. Minor consultations with others in the class are OK. The work on the project should be your own: all the design, programming, and testing should be done independently. Submissions that show excessive similarities will be taken as evidence of cheating and dealt with accordingly.