Problem 1. Initial Values in Dataflow Analysis

This question asks you to think about how changes to initial values in a data flow analysis can affect the result. Recall that an answer to a data flow problem is considered “safe” if it is no bigger than the ideal solution.

Suppose you have defined a forward dataflow algorithm that is distributive, has a monotonic transfer function and has finite descending chains. You accidentally initialized \( \text{OUT}[b] \) for all basic block to \( \bot \).

1. Will your algorithm give a safe answer for all flow graphs?
   Yes, because \( \bot \) is lower than the correct initialization so the answer will be lower (by monotonicity of the transfer function).

2. If not, will it give a safe answer for some flow graphs? If it will, give an example.

3. Will your algorithm give the MOP solution for all flow graphs?
   No, because it will in general give a solution smaller than MFP, which is itself lower or equal than MOP.

4. If not, will it give the MOP solution for some flow graphs? If it will, give an example.
   Yes, for example it will give the MOP solution for all acyclic graphs.
5. What if instead you had initialized $\text{IN}[b]$ to $\perp$. Would the algorithm produce the correct solution?

Yes, because the IN values are always recomputed in a forward pass.

**Rubric**: 1 point for each answer (0.5 correct answer, 0.5 correct explanation). Question 2 had no answer, so it was considered correct by default if you answer the previous one correctly.

**Problem 2.** Consider the following flow graph:

1. As seen in class, a flow graph is *reducible* if all retreating edges are back edges. Is this flow graph reducible? Justify your answer.

There are two retreating edges in any DFS visit you make: $x = 0 \rightarrow y > 0$ and $x = 2; y = 0 \rightarrow y > 0$. These are also back edges, because $y > 0$ dominates all nodes except $y = 0; z = 1; x = 1;$. Therefore the graph is reducible.

Observe that $x < 1 \rightarrow \text{return } z$ is either cross or advancing depending on the DFS visit order.

2. Assuming a language with *goto*, *if*, *while*, *break* and *continue* statements, what code could generate that flow graph? Use the minimum number of gotos.

There are multiple solutions, depending on how you write your if statements and which side you take to be the true or false branch. The model solution was:

```plaintext
y = 0;
z = 1;
x = 1;
```
while (y > 0) {
    if (z > 1) {
        x = 0;
    } else {
        y = 1;
        z = 1;
        if (x < 1)
            break;
        x = 2;
        y = 0;
    }
}
return z;

Note that in fact you do not need continue at all, you can always achieve the same effect with an if-else.

3. Given the previous language, are there flow graphs that cannot be expressed using those control statements? What if we remove goto?

The previous language has if and goto. Once you have that, you have everything you need for a flow graph where each block has at most two successors. Most IRs satisfy that requirement, because they use conditional branch statements to terminate a basic block with multiple successors, so it is acceptable to say that yes, if and goto are enough for any flow graph.

At the same time, if your IR has table switches or other multi-branch statements (like in joeq), then you cannot represent it in code unless you have a switch statement. This was an acceptable answer if you mentioned the requirement of at most two successors.

On the other hand, if we remove gotos, then non-reducible flow graphs cannot be represented anymore, because structured code only produces reducible flow graphs. Additionally, and this was a tricky case, there are reducible flow graphs that cannot be represented either: namely, all those where a break jumps out of two nested loops (a “break to label” in Java or JavaScript). This was not the answer we expected, so we did not took points if you did not consider it, but it is a valid answer, so if you mentioned as an example of some flow graph that cannot be represented, you did not lose points either.

Rubric: 2 point for answer 1 if correct and including the list of retreating/back edges; 2 points for correct code in answer 2 (1 point for a solution with gotos); 2 points for answer 3.

Problem 3. Apply Partial Redundancy Elimination to the following program. Assume that w, v, x, and u are used in other portions of the code (not shown), possibly in the same basic block. You do not need to show the intermediate steps, just show the optimized code. You may add basic blocks to the flow graph, but only show those that are not empty in your solution (existing basic blocks are not empty, even if they appear to be).
Solution:
Observe that “u = a + b” is not moved out of the loop. This is because it is not anticipated on the block immediately before the loop: the expression will be killed by “a = read()”. Additionally, there is no temporary and no copies of “a+b” going into “u” because at point the expression becomes postponable (it has seen the earliest but not a use yet).

**Rubric:** 2 point for each correctly placed temporary leading to “x = a + b”. 4 points for considering the “u = a + b” case correctly.
Problem 4. You are given the task of optimizing the code given below. You are only allowed to run the following four optimization techniques:

- PRE (as discussed in class)
- Constant Propagation (as discussed in class)
- Copy Propagation (as discussed in Section 9.1.5 of the textbook)
- Dead Code Elimination (liveness analysis, as discussed in class and in Homework 2)

in any order and multiple time if necessary. You can move and duplicate input(), but it must be executed exactly the same number of times as the original graph. You cannot modify the control flow graph or eliminate empty basic blocks, except to preprocess it for PRE. As in joes, assume that expression can take both registers and constants.

1. What is the order in which you should execute them to produce the best optimized code by running a minimum number of analyses?

2. What is the final optimized program?
Note that the question asks what is the order in which you should execute them. In other words, before you see the flow graph, what do you expect will yield the most advantage? The answer we expected was something along at the lines of constant prop first, because constants will not change later; then DCE to remove spurious uses that would confuse PRE into putting expressions too early; then copy propagation to maximize the number of identical expressions; then PRE, followed by copy propagation again to get rid of copies introduced by PRE; dead code elimination should be executed last, because all other passes introduce dead code.

At the same time, in this specific example, DCE will get rid of almost all code. If you observed that, and run DCE only, you received full credit. If you run DCE followed by anything else, and did not run DCE at the end, you received no credit. If you did not run DCE in the middle, and run PRE on the original graph, you received partial credit because it is not the best optimized code.

Additionally, the DCE in this problem was meant to remove all dead code in one pass (ie include multiple liveness passes inside, or use faintness analysis like in HW2). You did not lose points if you run multiple DCE passes to remove dead variables incrementally.

The final solution removes most of the code. The inputs survive in the same place, as nothing can ever move or delete an input (PRE cannot move the input outside the loop, because it would change the number of times it is executed), as do $v = q + m$ and $m = 2$ because the variables they set are live; everything else is dead code and is removed.

**Rubric:** 2 points for the correct order of executing the passes, and 4 points for correct final code.
Problem 5. For the following control flow graph, perform register allocation. Assign each definition and use of a variable to a live range. Draw the interference graph and the assignment of registers (i.e. colors) to live ranges.

1. What is the number of registers used by the heuristic algorithm?

2. What is the minimum number of registers needed by this program? Justify your answer.

Solution:
These are the live ranges:
The interference graph looks like this:

A possible assignment, with 4 registers is:

- R1 to A1, A4
- R2 to B2
• R3 to C5

• R4 to D3

With $n < 4$, the heuristic algorithm removes A4 first, then gets stuck. No matter what heuristic choice you make, you cannot color it. With $n = 4$, the heuristic algorithm removes A4 first, then all nodes have degree 3 and the algorithm runs to completion.

You cannot color this graph with fewer than 4 registers because you have a clique (complete subgraph) of size 4.

Rubric: 4 points for the interference graph; 2 points for the assignment; 1 point for the heuristic algorithm answer and 1 point for the minimum number answer.