CS243 Midterm Examination Solution

Winter 2023

February 15, 2023

Write your answers in the space provided on the exam. If you use additional scratch paper, please turn that in as well.

Your Name: _____ SUNet ID: _____

The following is a statement of the Stanford University Honor Code:

1. The Honor Code is an undertaking of the students, individually and collectively:
   (a) that they will not give or receive aid in examinations; that they will not give or receive unpermitted aid in class work, in the preparation of reports, or in any other work that is to be used by the instructor as the basis of grading;
   (b) that they will do their share and take an active part in seeing to it that others as well as themselves uphold the spirit and letter of the Honor Code.
2. The faculty on its part manifests its confidence in the honor of its students by refraining from proctoring examinations and from taking unusual and unreasonable precautions to prevent the forms of dishonesty mentioned above. The faculty will also avoid, as far as practicable, academic procedures that create temptations to violate the Honor Code.
3. While the faculty alone has the right and obligation to set academic requirements, the students and faculty will work together to establish optimal conditions for honorable academic work.

Signature: _____

Problem	#1	#2	#3	#4	**Total**
Score					
Max	15	10	15	30	70

Problem 1. Short questions. You must **justify your answer** to each question in one to two sentences. [15 points]

1. True or false. Given a monotone dataflow analysis with only finite descending chains, if we initialize the interior points to the bottom element, running the analysis on an acyclic control flow graph (CFG) may not yield the maximum fixed point solution.

   False. For acyclic CFGs, the initial interior point values have no bearing on the fixed-point solution, and hence we will still arrive at the MFP solution. Notice that even using a worst-case node ordering, it is guaranteed that at least one node will change its value during every iteration. This will keep occurring until all initial interior point values are entirely overwritten, after at most n iterations where n is the number of blocks in the program.

   For some intuition, imagine doing liveness analysis on a straight-line program. Using the best node ordering (i.e., from EXIT to ENTRY), the algorithm converges in one iteration, and the interior point values are never used at all. On the other hand, using the worst node ordering (from ENTRY to EXIT) means that the live variables will slowly propagate from EXIT to ENTRY one block at a time, but still it will reach the same conclusion after n iterations. This finding can be generalized to any acyclic CFGs, using topological order rather than straight-line order.

   (This is essentially the same problem as Problem 1.3 of the released 2021 midterm.)

2. True or false. A monotone dataflow analysis with finite descending chains that uses reverse postorder will converge within (maximum loop depth + 2) passes.

   False. As discussed in Lecture 4, this bound only applies to analyses that don't add information on cycles. Constant propagation is an example of an analysis for which this bound does not hold. See also textbook section 9.6.7 (pp. 667–9).

3. Recall that in Homework 1, the *Available Expression* analysis is defined with the transfer function $\text{OUT}[b] = (\text{IN}[b] \cup e\_gen_b) - e\_kill_b$, where e_gen_b is the set of expressions generated by block b and e_kill_b is the set of expressions killed by block b.

   The second pass in partial redundancy elimination (PRE) computes *Availability*. Its transfer function is defined as: $\text{OUT}[b] = (\text{Anticipated.IN}[b] \cup \text{IN}[b]) - e\_kill_b$, where Anticipated.IN$[b]$ is the result from the anticipation analysis in the first pass in PRE.

   What is the relationship between the IN/OUT sets resulting from *Available Expression* and those resulting from *Availability*? Express your answer using equality or subset relations.

   For both IN/OUT sets, *Available Expressions* $\subseteq$ *Availability*. This is since Anticipated.IN$[b]$ is always a superset of e_gen_b.

4. True or false. Given a program with an interference graph G, and the maximum node degree is n (meaning every node has at most n adjacent nodes). If the target machine has n registers, we do not need to spill to memory in register allocation.

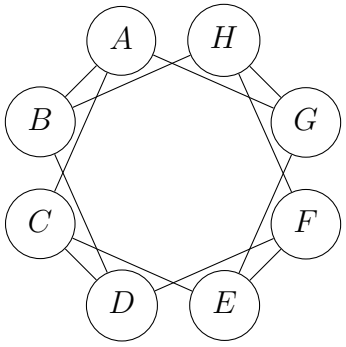
   False. A counterexample is a fully connected graph of 3 nodes (here, $n = 2$).

5. In register allocation, live ranges for the same program variable are only merged if they overlap. Suppose when we apply this technique to build the interference graph for program P, n_1 is the minimum number of registers needed to avoid spilling to memory. Suppose, for simplicity, the compiler instead merges *all* live ranges for the same variable, regardless of whether they overlap. Let n_2 be the new minimum number of registers required for the new interference graph for P to avoid spilling. What is the relationship between n_1 and n_2? Express your answer as an equality or inequality.

   $n_1 \leq n_2$. Merging more live ranges is equivalent to coloring the same interference graph as before, except with the additional constraint that several of the live ranges (corresponding to the same variable) need to have the same color. This invalidates some graph colorings that previously worked, but does not allow for additional colorings that previously did not work.

Problem 2. Register Allocation [10 points]

Consider the following interference graph:



1. Suppose we have a machine with 2 registers. Is it possible to allocate this interference graph? If yes, demonstrate a valid coloring. [2 points]

   (A, R0), (B, R1), (C, R1), (D, R0), (E, R0), (F, R1), (G, R1), (H, R0).

2. If the above answer is yes, what is the maximum number of edges you can add to the interference graph for it to remain 2-colorable? If the above answer is no, how many registers do you need to allocate the interference graph without spilling? [2 points]

   4 (AF, DG, CH, BE). Then it is a complete bipartite graph.

3. In graph theory, a clique is a subset of vertices of an undirected graph where every two distinct vertices in the clique are adjacent (i.e., an edge exists between every pair of vertices of the clique). Prove the following statement: if there exists a clique containing more than n nodes in an interference graph, the graph is not n-colorable. [3 points]

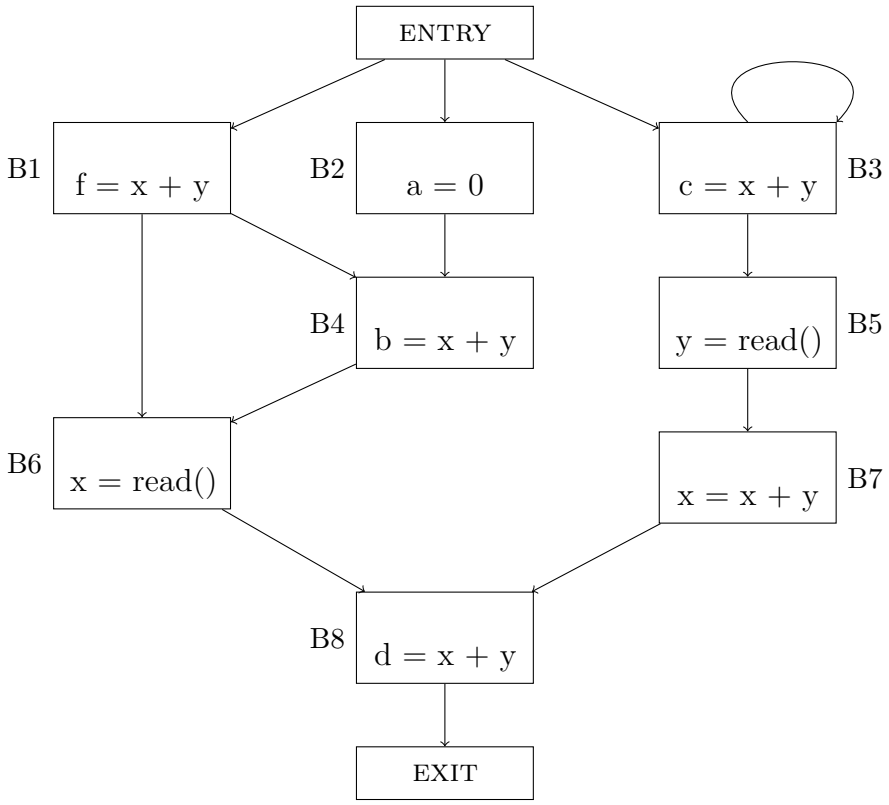
   Let C be the clique with more than n nodes. Since all nodes in a clique are adjacent, all elements in C must have different colors. But since C has more than n nodes, we need more than n colors to color C. Hence, the graph is not n-colorable.

4. Consider the inverse statement. If there does *not* exist a clique containing more than n nodes in an interference graph, is the graph always n-colorable? If true, prove it. If not, provide a counterexample. [3 points]

   False. Counterexample: any ring with n of nodes where n is odd and at least 5. There are no 3-cliques, but the graph is not 2-colorable (i.e., the graph is not bipartite).

Problem 3. Partial Redundancy Elimination [15 points]

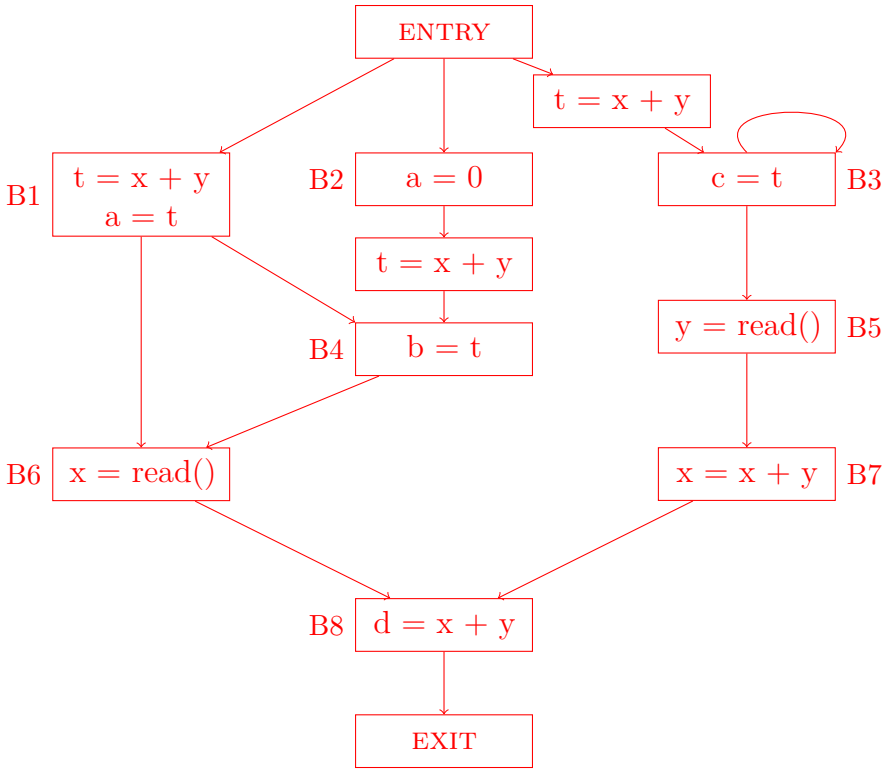
Answer the following questions.



1. Show the result of running partial redundancy elimination. What's the final optimized flow graph? You don't need to show the intermediate steps. [12 points]

2. To model the fact that `read()` may throw an exception, we add two edges to the original control flow graph (CFG): one from B5 to EXIT and one from B6 to EXIT. Compare the optimized flow graphs after running PRE on the original CFG, and running PRE on the new CFG with the two new edges. Will the resulting CFGs be different (other than the two added edges)? Briefly explain your answer. [3 points]

1. Optimized CFG:



2. The resulting CFG will not change except for the added two edges. Adding these two edges will neither affect the redundancy on any existing path in the CFG nor introduce any new redundancy on new paths.

   Another explanation is to reason about how the anticipated expressions/used expressions/... change after we add the two edges.

Problem 4. Optimizing dynamically typed languages [30 points]

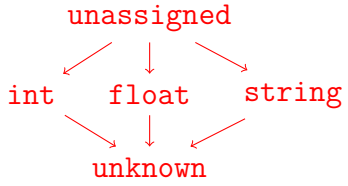
Consider a dynamically-typed language similar to JavaScript and Python. The language has three kinds of instructions:

- $x = T(\ldots)$, where T is one of $\{\texttt{int}, \texttt{float}, \texttt{string}\}$. This sets the variable x to a value of type T.
- $x = y$, where x and y are variables.
- print(x), where x is a variable.

Unlike statically-typed languages like C and Java, the type of a program variable is allowed to change in this language. As an example, the program `x = int(1); x = float(3.14)` will result in variable `x` having an `int` value at first, but afterwards a `float` value.

Suppose we are writing an optimizing compiler for this language. Because there are special instructions for printing an integer (`iPrint`), printing a float (`fPrint`), and printing a string (`sPrint`), your task is to identify those print(...) instructions whose operand type can be determined at compile time. This problem has four parts: (a) Define your dataflow analysis by filling in the table below. (b) State clearly how you perform the optimization. (c) Is your dataflow analysis monotone? Explain your answer. (d) Is your dataflow analysis distributive? Explain your answer.

Part (a). Dataflow analysis specification. You may assume that each instruction is in its own basic block. [18 points]

Direction of your analysis (forward/backward)	Forward
Lattice elements and meaning	Map from each variable to one of the following: `unassigned` representing an unassigned variable; `int`, `float`, `string` representing a variable of definitely that type; `unknown` representing more than one possible types
Meet operator	For each variable: 
Is there a top element? If yes, what is it?	Mapping each variable to `unassigned`
Is there a bottom element? If yes, what is it?	Mapping each variable to `unknown`
Transfer function	$\text{OUT}[b, x] = \begin{cases} T & \text{if } b\text{: } x = T(\ldots), \\ \text{IN}[b, y] & \text{if } b\text{: } x = y, \\ \text{IN}[b, x] & \text{otherwise.} \end{cases}$
Boundary condition	Mapping each variable to `unassigned`
Interior points	Mapping each variable to `unassigned`

Part (b). How do you optimize the program? [4 points]

For each instruction of form b: print(x) where $\text{IN}[b, y] = T$ for some type T, replace it with the specialized instruction for printing out a value of type T.

Part (c). Is your dataflow analysis monotone? Sketch a proof for it if so, or provide a counterexample if not. [4 points]

Yes, since it is distributive. See below.

Part (d). Is your dataflow analysis distributive? Sketch a proof for it if so, or provide a counterexample if not. [4 points]

Yes. The print case is trivial since the transfer function is the identity. In the case of b: $x = T(\ldots)$, for any two mappings V, W and all $v \neq x$:

$$f_b(V \wedge W)[x] = T = T \wedge T = f_b(V)[x] \wedge f_b(W)[x] = (f_b(V) \wedge f_b(W))[x],$$
$$f_b(V \wedge W)[v] = (V \wedge W)[v] = V[v] \wedge W[v] = f_b(V)[v] \wedge f_b(W)[v] = (f_b(V) \wedge f_b(W))[v].$$

In the case of b: $x = y$, for any two mappings V, W and all $v \neq x$:

$$f_b(V \wedge W)[x] = (V \wedge W)[y] = V[y] \wedge W[y] = f_b(V)[x] \wedge f_b(W)[x] = (f_b(V) \wedge f_b(W))[x],$$
$$f_b(V \wedge W)[v] = (V \wedge W)[v] = V[v] \wedge W[v] = f_b(V)[v] \wedge f_b(W)[v] = (f_b(V) \wedge f_b(W))[v].$$

In conclusion, we have that $f_b(V \wedge W) = f_b(V) \wedge f_b(W)$.