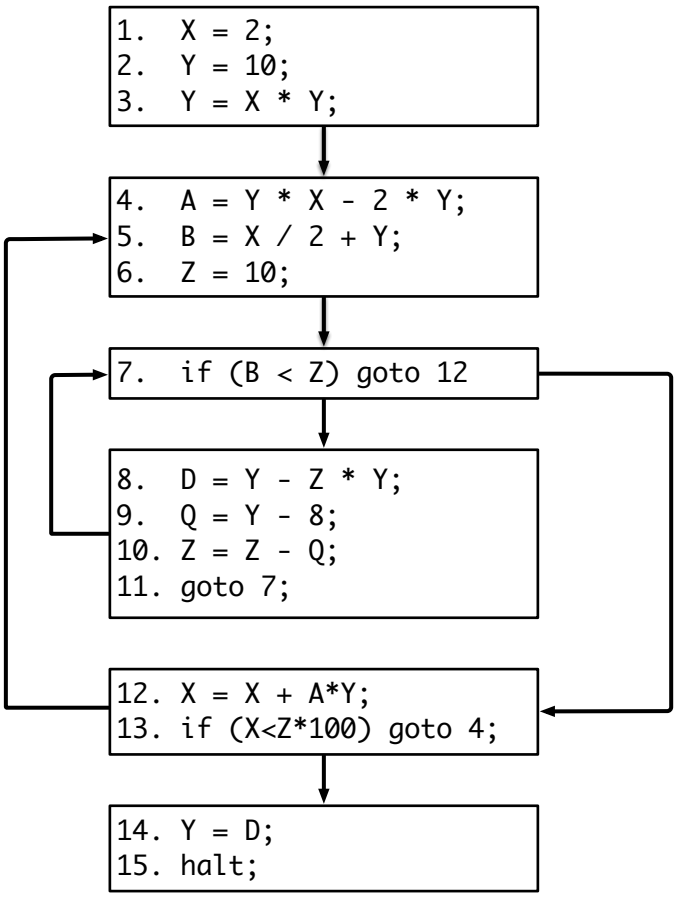
ECE 468/583
Problem Set 6: Loop Optimization and Pointer Analysis

Loop transformations
For the following problems, consider the code below:

```
1.   X = 2;
2.   Y = 10;
3.   Y = X * Y;
4.   A = Y * X - 2 * Y;
5.   B = X / 2 + Y;
6.   Z = 10;
7.   if (B < Z) goto 12
8.   D = Y - Z * Y;
9.   Q = Y - 8;
10.  Z = Z - Q;
11.  goto 7;
12.  X = X + A*Y;
13.  if (X < Z*100) goto 4;
14.  Y = D;
15.  halt;
```

1. Draw the CFG for the code above. Identify the loops in the code.
   Answer:



The loops are: lines 4–13 and lines 7–11.

2. Which statements are loop invariant? Can they be moved outside their enclosing loop? Show the code that results after hoisting any loop invariant code outside the loop.

   Answer: Statements 6 and 9 are loop invariant. Statement 9 can be housted outside of its loop (indeed, it can be hoisted outside of both loops), but Statement 6 cannot, because Z is defined more than once within the loop.

   The resulting code is:

```
1.   X = 2;
2.   Y = 10;
3.   Y = X * Y;
3'.  Q = Y - 8;
4.   A = Y * X - 2 * Y;
```

```
5.   B = X / 2 + Y;
6.   Z = 10;
7.   if (B < Z) goto 12
8.   D = Y - Z * Y;
9.   // Q = Y - 8;
10.  Z = Z - Q;
11.  goto 7;
12.  X = X + A*Y;
13.  if (X < Z*100) goto 4;
14.  Y = D;
15.  halt;
```

3. Identify the induction variables in this code. Show the code that results after performing any possible strength reduction.
   Answer: There is only one induction variable: Z, in line 10. Note that X is *not* an induction variable: A is not loop invariant, so X does not increment by a fixed amount each iteration. The rewritten code looks like (note the signs! Z decrements by Q each time, so we multiply -Q by -Y):

```
1.   X = 2;
2.   Y = 10;
3.   Y = X * Y;
3'.  Q = Y - 8;
4.   A = Y * X - 2 * Y;
5.   B = X / 2 + Y;
6.   Z = 10;
6'.  D' = Y - Z * Y;
7.   if (B < Z) goto 12
8.   D = D'
9.   // Q = Y - 8;
10.  Z = Z - Q;
10'. D' = D' + Q * Y;
11.  goto 7;
12.  X = X + A*Y;
13.  if (X < Z*100) goto 4;
14.  Y = D;
15.  halt;
```

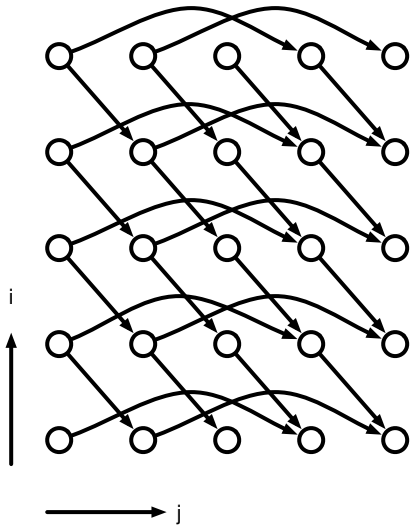
4. Show the code after performing any possible linear test replacement.
   Answer: Note that the comparison operation switched directions! To do linear test replacement, we had to multiply both sides by -Y, which flips the inequality.

```
1.   X = 2;
2.   Y = 10;
3.   Y = X * Y;
3'.  Q = Y - 8;
4.   A = Y * X - 2 * Y;
5.   B = X / 2 + Y;
6.   Z = 10;
6'.  D' = Y - Z * Y;
7.   if (Y - B * Y > D') goto 12
8.   D = D'
9.   // Q = Y - 8;
10.  Z = Z - Q;
10'. D' = D' + Q * Y;
11.  goto 7;
12.  X = X + A*Y;
13.  if (X < Z*100) goto 4;
14.  Y = D;
15.  halt;
```

5. Draw the iteration space graph for the following piece of code (be careful about the index expressions and the loop order!):

```
for (j = 0; j < 5; j++)
  for (i = 0; i < 5; i++)
    A[j+2][i+1] = A[j-1][i+1] + A[j+1][i+2];
```

Answer:



6. What are the distance vectors? The direction vectors?
   Answer: Distance vectors: (3, 0), (1, -1) Direction vectors: (+, 0), (+, -)

7. Can the loops be interchanged? Why or why not?
   Answer:

   The loops cannot be interchanged, because the (+, -) dependence would break.

8. Can the following two loops be fused? Why or why not? Explain your answer in terms of dependences between the loops.

```
for (i = 1; i < 10; i++)
  A[i - 1] = B[i + 1]

for (i = 1; i < 10; i++)
  A[i + 2] = A[i]
```

   Answer: They cannot be interchanged. There is a dependence from the i=3 iteration of the first loop (which writes to A[2]) and the i=1 iteration of the second loop (which writes to A[3]). In the transformed code, the second write will happen before the first write, generating incorrect code.

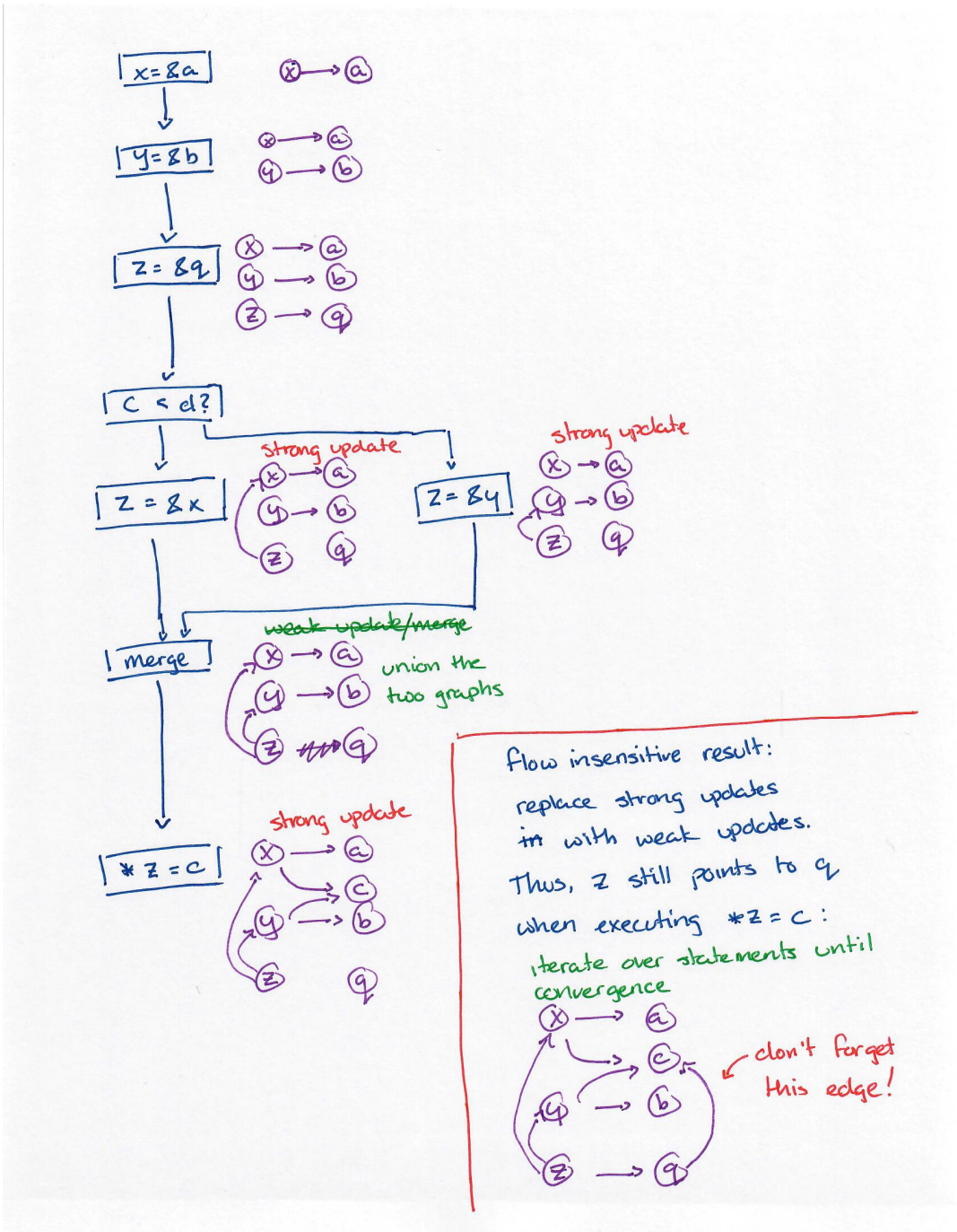
9. Draw the points-to-graph *at the end* of this piece of code for a flow-sensitive pointer analysis (assume the variables have all been declared appropriately beforehand)

```
x = &a;
y = &b;
z = &q;
if (c < d) {
  z = &x;
else {
  z = &y;
}
*z = &c;
```

Answers below. I have shown the points-to graph after each statement is executed.

10. Draw the points-to-graph you would get if you ran a flow-*insensitive* pointer analysis on the same code.

    Answers below, in inset.


x = &a
y = &b
z = &q
c < d?
z = &x
z = &y
strong update
strong update
merge
weak update/merge
union the two graphs
*z = c
strong update
flow insensitive result:
replace strong updates in with weak updates.
Thus, z still points to q
when executing *z = c:
iterate over statements until convergence
don't forget this edge!