DCC888 – Worklist Algorithms

Nome: _____ Matrícula: _____

1. Consider the algorithm on the left, which, when applied upon the constraints seen on the right, gives us the table below:

$x_1 = \bot, x_2 = \bot, \ldots, x_n = \bot$
w = $[v_1, \ldots, v_n]$
while (w ≠ [])
  v_i = extract(w)
  $y = F_i(x_1, \ldots, x_n)$
  if $y \neq x_i$
    for $v \in dep(v_i)$
      w = insert(w, v)
    $x_i = y$

x1 = []

x2 = x1 ∪ (x3 \ [3, 5, 6]) ∪ {3}

x3 = x2

x4 = x1 ∪ (x5 \ [3, 5, 6]) ∪ {5}

x5 = x4

x6 = x2 ∪ x4

w	x1	x2	x3	x4	x5	x6
[x1, x2, x3, x4, x5, x6]	⊥	⊥	⊥	⊥	⊥	⊥
[x2, x4, x2, x3, x4, x5, x6]	[]	⊥	⊥	⊥	⊥	⊥
[x3, x6, x4, x2, x3, x4, x5, x6]	[]	[3]	⊥	⊥	⊥	⊥
[x2, x6, x4, x2, x3, x4, x5, x6]	[]	[3]	[3]	⊥	⊥	⊥
[x6, x4, x2, x3, x4, x5, x6]	[]	[3]	[3]	⊥	⊥	⊥
[x4, x2, x3, x4, x5, x6]	[]	[3]	[3]	⊥	⊥	[3]
[x5, x6, x2, x3, x4, x5, x6]	[]	[3]	[3]	[5]	⊥	[3]
[x4, x6, x2, x3, x4, x5, x6]	[]	[3]	[3]	[5]	[5]	[3]
[x6, x2, x3, x4, x5, x6]	[]	[3]	[3]	[5]	[5]	[3]
[x2, x3, x4, x5, x6]	[]	[3]	[3]	[5]	[5]	[3,5]
[x3, x4, x5, x6]	[]	[3]	[3]	[5]	[5]	[3,5]
[x4, x5, x6]	[]	[3]	[3]	[5]	[5]	[3,5]
[x5, x6]	[]	[3]	[3]	[5]	[5]	[3,5]
[x6]	[]	[3]	[3]	[5]	[5]	[3,5]
[]	[]	[3]	[3]	[5]	[5]	[3,5]

   (a) If, instead of starting with the list $[x_1, x_2, x_3, x_4, x_5, x_6]$, we had used the list $[x_6, x_5, x_4, x_3, x_2, x_1]$, how many iterations would be necessary so that our algorithm could reach a fixed point?

(b) The table on the previous page assumes that the last element inserted in the list is the first element to be removed by *extract*. This ordering is called LIFO (Last-in, First-out). How many iterations would be necessary to reach a fixed point, if the ordering were FIFO (First-in, First-out)? Notice that in this case we would have a queue, instead of a stack of nodes to be processed.

(c) Our algorithm does not check if there is already an element in the worklist, before inserting it there. Hence, we might have several versions of the same element in the worklist. Even though that is a very simple approach, it can be found even in actual implementations of algorithms based on worklists. What are the advantages in allowing repeated elements in the worklist? Explain your answer in terms of computational complexity.

(d) Suppose now that an element is inserted in the list only if it is not already there. In this case, how many iterations our algorithm would do, until we could reach a fixed point in this example?

2. Once we study loop optimizations, we will see that a node n_1 dominates a node n_2 in a directed graph $G = (V, E)$, with root $H, H \in V$, if every path from H to n_2 goes across n_1. The set of dominators of a node can be approximated by the set of equations:

$$Dom(n) = \begin{cases} \{n\} & if n \in H, \\ \{n\} \cup \bigcap_{(n',n)\in E} Dom(n') & \text{otherwise} \end{cases}$$

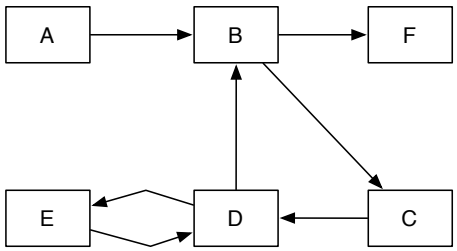
Write an algorithm, based on worklists, to compute the set of dominators of the nodes of the directed graph.

3. Below we have four programs, and their control flow graphs. In each case, you must:

   - Relate the names of basic blocks with lines in the code. To make it easier, I have filled up the first line for you.
   - Produce the post-order tree for each control flow graph. The root of the tree should start always in block A. You can add the tree edges onto the CFG itself.
   - Number the nodes in each tree in reverse post-order.

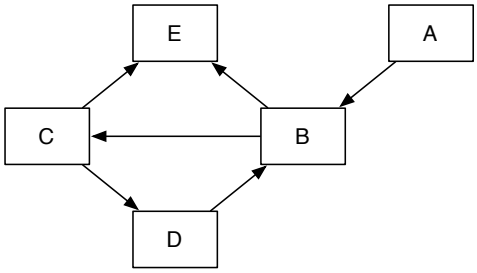
   (a) First program:

```
A     int bbLoop(int n, int m) {
        int sum = 0;
        int c0;
___     for (c0 = n; c0 > 0; c0--) {
___       int c1 = m;
___       for (; c1 > 0; c1--) {
___         sum += c0 > c1 ? 1 : 0;
          }
        }
___     return sum;
      }
```



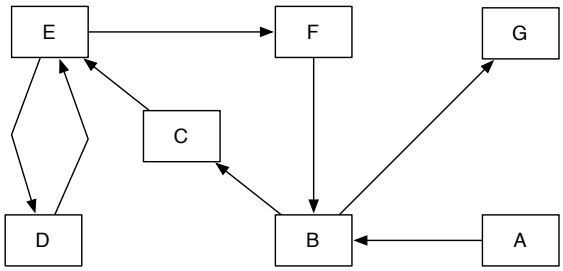
(b) Second program:

```
    #define SIZE 80
    int main(int argc, char** argv) {
A     int i = 0;
      char* buf = malloc(SIZE);
      while (
___            i < SIZE &&
___            argv[0][i] != '\0') {
___       buf[i] = argv[0][i];
          i++;
      };
___   printf("%s\n", buf);
    }
```



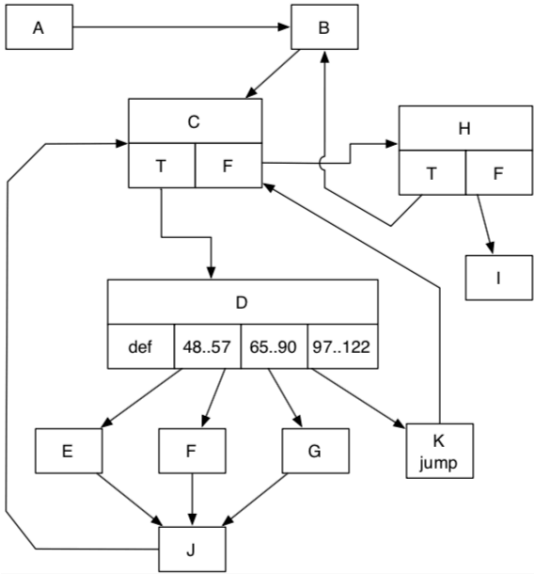
(c) Third program:

```
    int main(int argc, char **argv) {
A     unsigned k = 0;
___   while (k < 100) {
___     int i = 0;
        int j = k;
___     while (i < j) {
___       i = i + 1;
          j = j - 1;
        }
___     k = k + 1;
      }
___   return k;
    }
```



(d) Fourth program. Notice that this program is not ANSI C compliant, but it can be compiled with current versions of gcc and clang:

```
    int main(int argc, char** argv) {
A     int freq_dig = 0;
      int freq_lower = 0;
      int freq_upper = 0;
      int i = 0;
___   do {
        char* p = argv[0];
___     while (*p != '\0') {
___       switch (*p) {
___         case '0' ... '9':
              freq_dig++;
              break;
___         case 'A' ... 'Z':
              freq_upper++;
              break;
___         case 'a' ... 'z':
              freq_lower++;
              break;
___         default:
              continue;
          }
___     }
___   } while (i < argc);
___   printf("Digits = %d\n", freq_dig);
      printf("Lcase = %d\n", freq_lower);
      printf("Ucase = %d\n", freq_upper);
    }
```



4. The code snippet below is a skeleton of a data-flow solver, implemented in Python. Implement the method `eval()` in the `Constraint` class, so that this program computes the dominance relations seen in Question 2.

```
class Constraint:
    def __init__(self, name, init):
        self.name = name
        self.solution = set(init)
        self.deps = []
    def eval(self):
        # TODO

def evaluate(constraints):
    change = False
    for c in constraints:
        if c.eval():
            change = True
    return change

def fixPoint(constraints):
    change = evaluate(constraints)
    if change:
        fixPoint(constraints)

def solve(constraints):
    fixPoint(constraints)
    for c in constraints:
        print c.solution
```

```
def test1():
    universe = ['n0', 'n1', 'n2', 'n3']
    c0 = Constraint('n0', ['n0'])
    c1 = Constraint('n1', universe)
    c2 = Constraint('n2', universe)
    c3 = Constraint('n3', universe)
    c1.deps = [c0]
    c2.deps = [c0]
    c3.deps = [c1, c2]
    solve([c0, c1, c2, c3])
```

```
def test2():
    universe = ['n0', 'n1', 'n2', 'n3', 'n4', 'n5']
    c0 = Constraint('n0', ['n0'])
    c1 = Constraint('n1', universe)
    c2 = Constraint('n2', universe)
    c3 = Constraint('n3', universe)
    c4 = Constraint('n4', universe)
    c5 = Constraint('n5', universe)
    c1.deps = [c0, c4]
    c2.deps = [c1]
    c3.deps = [c1]
    c4.deps = [c2, c3]
    c5.deps = [c4]
    solve([c0, c1, c2, c3, c4, c5])
```

Your implementation must compute the dominance relations correctly for the two tests left as examples.