CSC165H1: Problem Set 5

Due April **6**, 2017 before 10pm

General instructions

Please read the following instructions carefully before starting the problem set. They contain important information about general problem set expectations, problem set submission instructions, and reminders of course policies.

- Your problem sets are graded on both correctness and clarity of communication. Solutions which are technically correct but poorly written will not receive full marks. Please read over your solutions carefully before submitting them.
- Each problem set may be completed in groups of up to three. If you are working in a group for this problem set, please consult `https://github.com/MarkUsProject/Markus/wiki/Student_Groups` for a brief explanation of how to create a group on MarkUs.

  Exception: Problem Set 0 must be completed individually.
- Solutions must be typeset electronically, and submitted as a PDF with the correct filename. **Hand-written submissions will receive a grade of ZERO.**

  The required filename for this problem set is **problem_set5.pdf**.
- Problem sets must be submitted online through MarkUs. If you haven't used MarkUs before, give yourself plenty of time to figure it out, and ask for help if you need it! If you are working with a partner, you must form a group on MarkUs, and make one submission per group. "I didn't know how to use MarkUs" is not a valid excuse for submitting late work.
- Your submitted file should not be larger than 9MB. This may happen if you are using a word processing software like Microsoft Word; if it does, you should look into PDF compression tools to make your PDF smaller, although please make sure that your PDF is still legible before submitting!
- Submissions must be made *before* the due date on MarkUs. You may use *grace tokens* to extend the deadline; please see the Problem Set page for details on using grace tokens.
- The work you submit must be that of your group; you may not refer to or copy from the work of other groups, or external sources like websites or textbooks. You may, however, refer to any text from the Course Notes (or posted lecture notes), except when explicitly asked not to.

Recall that we say a graph $G = (V, E)$ is **(fully) connected** if and only if for every pair of vertices $u, v \in V$, u and v are connected in G.

Also recall (from Problem Session 11) that if the vertex set is $V = \{0, 1, \ldots, n-1\}$, then we can represent a graph in a program as a two-dimensional n-by-n array A, where the entry $A[i][j]$ is set to 1 if vertices i and j are adjacent (the edge), and 0 if they aren't.

The following algorithm takes as input such an array, and returns `True` if the algorithm is connected, and `False` otherwise.

UPDATE: in the original version of the algorithm, the matrix A was being updated too early in the Main Loop. We've updated the algorithm so that now a new matrix $A1$ stores all the updates, and then the updates are transferred over to A.

```
def connected(A):
  n = len(A)  # This is the number of vertices in the graph.

  # First, set diagonal to 1, since every vertex is connected to itself.
  for i in range(n):
    A[i][i] = 1

  for q in range(1, ceil(log(n)) + 1):          # Main Loop: q = 1,2,...,ceil(log(n))
    A1 = new n by n array containing all zeros  # UPDATE: Assume this takes n*n steps.
    # Find new connectedness information
    for i in range(n):
      for j in range(n):
        for k in range(n):
          if A[i][k] == 1 and A[k][j] == 1:
            A1[i][j] = 1                        # UPDATE: A1 stores new connectivity
                                                # information.

    for i in range(n):            # UPDATE: Change A to store the new connections.
      for j in range(n):
        if A[i][j] == 0:          # If A[i][j] is already 1, don't need to update.
          A[i][j] = A1[i][j]

    # Check if the graph is connected at this point.
    all_conn = True
    for i in range(n):
      if A[0][i] == 0:
        all_conn = False
        break

    if all_conn:
      return True

  # The loop has ended and *not* returned early.
  return False
```

This algorithm starts with an adjacency matrix A and continually updates the entries of A to represent new connectedness relationships between pairs of vertices. Essentially, at each iteration of the innermost loop, the algorithm checks whether vertex k is connected to both i and j; and if so, it records the fact that i and j are also connected.

In this problem set, you'll formally prove the correctness of this algorithm, and analyse its running time.

1. **[12 marks] Proving correctness.** To formally prove the correctness of this algorithm, we need the following predicate, where G is a graph, u and v are vertices in the graph, and $d \in \mathbb{N}$.

$$PathLength(G, u, v, d): \text{ “there is a path in } G \text{ between } u \text{ and } v \text{ of length at most } d\text{”}$$

Using this predicate, we can state the key property of graphs that makes the `connected` algorithm work:

$$\forall G = (V, E),\ \forall u, v \in V,\ \forall d \in \mathbb{N},\ PathLength(G, u, v, 2d) \Leftrightarrow \big(\exists w \in V,\ PathLength(G, u, w, d) \wedge PathLength(G, w, v, d)\big)$$

(a) Let $n \in \mathbb{Z}^+$ **with** $n > 1$, and consider running `connected` on an n-by-n adjacency matrix representing a graph $G = (V, E)$, where $V = \{0, 1, \dots, n-1\}$.

Prove by induction on q that at the end of iteration q of the Main Loop, the following is true:

$$\forall i, j \in V,\ A[i][j] = 1 \Rightarrow PathLength(G, i, j, 2^q)$$

Note: $q \in \{1, 2, \dots, \lceil \log n \rceil\}$, but you can still use induction: pick a base case of 1, and for the induction step assume $1 \leq q \leq \lceil \log n \rceil - 1$ in addition to the induction hypothesis.

(b) Now in the same context as part (a), prove by induction on q that at the end of the iteration q of the Main Loop, the following is true:

$$\forall i, j \in V,\ A[i][j] = 0 \Rightarrow \neg PathLength(G, i, j, 2^q)$$

(You might recognize that parts (a) and (b) together prove an “if and only if.”)

(c) Use parts (a) and (b) to prove that this algorithm is correct. That is, prove that if `connected` returns `True`, the input graph is connected, and if it returns `False` then the input graph is not connected. **UPDATE**: you can assume that the graph has at least two vertices.[1]

2. **[9 marks] Runtime analysis.** Let $WC(n)$ and $BC(n)$ be the worst-case and best-case running times of the `connected` algorithm, where n represents the *number of vertices* of the input graph. (Note that the input is a two-dimensional array with n^2 elements.)

(a) Find and prove a good asymptotic upper bound on $WC(n)$. By “good” we mean that if you prove $WC(n) \in \mathcal{O}(f)$, it should also be the case that $WC(n) \in \Omega(f)$, although you do not need to prove that here.

(b) Find and prove a good asymptotic lower bound on $BC(n)$. The term “good” here means the analogous thing as in part (a).

(c) Prove the asymptotic upper bound on $BC(n)$ that matches the lower bound you proved in part (b). For example, if in part (b) you proved that $BC(n) \in \Omega(n)$, then in this part you should prove that $BC(n) \in \mathcal{O}(n)$.

[1]Someone correctly pointed out the algorithm is incorrect on a graph with a single vertex, because in this case $\log n = 0$, and the Main loop doesn't run at all. This is straightforward to fix in the algorithm, but to make it easiest for you we're adding an additional assumption for this question.