COMP 182: Homework 4

Quan Le

Friday, February 28, 2020

Problem 1

Divide-and-conquer algorithms, greedy algorithms, and proofs [45 pts]

1. Consider the following problem: **Input:** set S of n real numbers, and a real number x. **Question:** Does S contain two elements whose sum equals x? Design an $O(n \log n)$ algorithm for the problem. Give the pseudo-code of the algorithm and show that your algorithm indeed runs in the specified time.

Algorithm 1: SetSum

Input: set S of n real numbers; a real number x
Output: True, if there exist two elements in S that sum to c, False otherwise

```
S' ← an arbitrarily ordered sequence of the elements in S.
L ← MergeSort(S')
a ← 1
b ← n
for i ← 1 to n − 1 do
    y ← L_a + L_b
    if y < x then
        a ← a + 1
    else if y > x then
        b ← b − 1
    else
        return True
return False
```

As discussed in class, **MergeSort** operates in $O(n \log n)$. The additional for-loop only runs for $n-1$ iterations. Also in class, we noted $n = O(n \log n)$. $n = O(n \log n) \implies n + n \log n = O(n \log n)$, and therefore, the algorithm for **SetSum** is $O(n \log n)$.

2. Consider a sequence $x_1, x_2, ..., x_n$ of n real numbers (assume n is even). We are interested in partitioning the n numbers into $n/2$ pairs (that is, each of the n numbers appears in exactly one of the $n/2$ pairs) in the following way. For each pair in the partition, compute the sum of its numbers. Denote by $s_1, s_2, ..., s_{n/2}$ the $n/2$ sums. The goal is to find the partition that minimizes the maximum sum. (a) Describe concisely in English the idea of a brute-force algorithm for solving this problem, and provide, using big-O notation, the worst-case running time of your algorithm. (b) Establish a mathematical result that will allow you to design a worst-case polynomial-time algorithm for the problem, and prove the result. (c) Describe the polynomial-time algorithm and provide, using big-O notation, its worst-case running time.

- A brute force algorithm for the above problem is to calculate the maximum sum, for each of the possible partitions. Then, the worst-case running time involves finding the number of possible partitions, of which there are $n!/(2^{n/2}(n/2)!)$ for a set of size n. $n!$ describes the number of

possible permutations for the set, the division by $2^{n/2}$ eliminates the duplicate case for flipping, and the division by $(n/2)!$ eliminates the duplicate permutations of the pairs. Finally, for each partition, to determine the maximum sum is $O(n)$. Therefore, the brute force algorithm described operates in $O(n * n!/(2^{n/2}(n/2)!))$ time.

- We seek to prove that the minimum maximum sum is obtained by pairing each element x_i with x_{n+1-i}.

  First, it is clear that in order to minimize the sum of the pair containing x_n, we must pair it with x_1. Since the sequence is ordered

  Now consider an arbitrary finite ordered sequence of reals with an even number of elements, and let $x_n + x_1$ be the minimized sum of the pair containing x_n. Consider the remaining set, each element of which must also be paired with another element in the set. The sum of the pair containing x_{n-1} is thus minimized if x_{n-1} is paired with the new minimum of the set, x_2.

  Then, we have shown that, given a set of n real numbers, the partition that minimizes the maximum sum is given by the set $S = \{s_i | s_i = x_i + x_{n+i-1}, 1 \leq i \leq n/2\}$.

  Algorithm 2: MinMaxSumPartition

  Input: Set X of n real numbers
  Output: Set S

- ```
  1 S ← ∅
  2 X′ ← MergeSort(X)
  3 for i ← 1 to n/2 do
  4     S ← S ∪ {(x_i + x_{n+1−i})}
  5 return S
  ```

  Then, since **MergeSort** is $O(n \log n)$, and $n/2 = O(n \log n)$ (the construction of S), then the worst case running time of **MinMaxSumPartition** is $O(n \log n)$.

3. Prove that any weighted connected graph with distinct weights to its edges (that is, no two edges have the same weight) has exactly one MST.

   Consider the contradiction, that there exists at least 2 MSTs $g_1 = (V, E_1), g_2 = (V, E_2)$ of a distinctly weighted connected graph.

   First, let us define $\textbf{SumWeights}(g = (V, E, w)) = \sum_{e \in E} w(e)$.

   Then, the addition of an arbitrary $e_1 \in E_1$, $e_1 \notin E_2$ to g_2 results in a cycle in g_2.

   In that cycle, there exists an edge $e_2 \neq e_1$, where $e_2 \in E_2 \setminus E_1$.

   Then, $g_1' = (V, E_1 \cup \{e_2\} - \{e_1\})$ and $g_2' = (V, E_2 \cup \{e_1\} - \{e_2\})$ are both spanning trees.

   $\textbf{SumWeights}(g_1) = \textbf{SumWeights}(g_2)$

   $w(e_2) \neq w(e_1)$

   $\textbf{SumWeights}(g_1') = \textbf{SumWeights}(g_1) + w(e_2) - w(e_1)$

   $\textbf{SumWeights}(g_2') = \textbf{SumWeights}(g_2) + w(e_1) - w(e_2)$

   However, we have a contradiction since $\textbf{SumWeights}(g_1')$ or $\textbf{SumWeights}(g_2')$ must be less than $\textbf{SumWeights}(g_1)$.

   Our assumption is false, and therefore, there is a unique minimum spanning tree for a connected graph with distinct weights.

4. Upon termination of Dijkstra's algorithm on a connected graph g, if we keep only the nodes and the edges that connect nodes to their parents, we obtain a tree. For an arbitrary connected graph g and node i in it, is the tree produced by the algorithm on g, starting from node i, a spanning tree? Is it a minimum spanning tree? Prove your answer.
  ```

Given a connected graph g, the tree produced has to have reached each node, and there are no cycles, as that would imply that Dijkstra's algorithm finds more than one shortest path, which is false. Therefore, it can be concluded that the tree produced is a spanning tree.

However, it is not necessarily the minimum spanning tree. Consider the graph $g = (V, E, w), V = \{1, 2, 3\}, E = \{\{1, 2\}, \{2, 3\}, \{1, 3\}\}$

$$w(x) = \begin{cases} 2 \text{ if } x = \{1, 2\}, \{1, 3\} \\ 1 \text{ if } x = \{2, 3\} \end{cases}$$

Then, the tree formed by running the algorithm from node 1 is not the minimum spanning tree.

5. Show that Dijkstra's algorithm may not work on directed graphs if edges have negative weights (notice that Dijkstra's algorithm, as given on the slides, works for directed graphs).

   If a directed graph has negative weights, then when the algorithm considers the neighbors h of a node k, using $d_k + w((k, h)) < d_h$ may result in choosing a longer path, since the distance values no longer strictly increase.

   Consider the following example: $g = (V, E, w), V = \{1, 2, 3\}, E = \{(1, 2), (2, 3), (1, 3)\}$

$$w(x) = \begin{cases} 1 \text{ if } x = (1, 3) \\ -2 \text{ if } x = (2, 3) \\ 2 \text{ if } x = (1, 2) \end{cases}$$

   Then, it is clear that starting from node 1, the output becomes $\{1 : null, 2 : 1, 3 : 1\}$, and that it fails in determining the shortest path from 1 to 3. More specifically, Dijkstra's algorithm will begin at node 1, then add nodes 2 and 3. It will pop node 3 off of the queue, but does so without updating d. The algorithm is then unable to update the path to node 3 when considering the neighbors of node 2.

6. Given n open intervals $(a_0, b_0), (a_1, b_1), ..., (a_{n-1}, b_{n-1})$ on the real line (that is, each a_i and b_i is a real number), each representing the start and end times of some activity requiring the same resource, the task is to find the largest number of these intervals so that no two of them overlap. Consider three greedy algorithms for this problem based on (a) earliest start first; (b) shortest duration first; and, (c) earliest finish first. For each of the three algorithms, state whether it always yields an optimal solution or not, and prove your answer.

   - The greedy algorithm considering earliest start first does not find the optimal solution. All one has to do is consider the intervals (0,3), (1,2), (2,3). Such a greedy algorithm would select (0,3), and terminate, as there are no more non-overlapping sets. (1,2), (2,3) do not overlap, and thus this algorithm is evidently non-optimal.
   - A counterexample demonstrating the sub-optimal performance of the shortest duration greedy algorithm is also simple: (2,4), (0,3), (3,6). The greedy algorithm would select (2,4), the interval of shortest duration, and terminate. Since the output (0,3), (3,6) is larger, it is evident that this algorithm does not always yield an optimal solution.
   - This third algorithm always yields an optimal solution.

     Consider the contradiction, that there is a subset of intervals, U' that has a cardinality greater than the subset found by the algorithm, U_a. Then, there must be a single interval i in U_a that conflicts with at least 2 intervals j, k in U'. However, this implies that either j or k has an earlier finish than i, and thus, would be included in U_a. We have a contradiction, and thus our assumption is false and the third algorithm always yields an optimal solution.

Problem 2

Shortest-path betweenness of edges: BFS revisited [26 pts]

1. Running bfs from Part 1 on graph g_1 in Figure 2, starting from node 0, report the values d_j and n_j for every node j in g_1. What quantity does the value of n_j, for a node j, reflect?

   Running bfs on the provided graph g1 yields the results:

   $$d = \{0:0, 1:1, 2:1, 3:2, 4:2, 5:3, 6:3\}$$
   $$n = \{0:1, 1:1, 2:1, 3:2, 4:1, 5:3, 6:1\}$$

   For a node j in g_1, n_j represents the number of paths that go through node j. More specifically, it is the number of shortest paths from the start node to node j.

2. The function is written in the provided code, *autograder.py*.

3. **Table 1** demonstrates the betweenness values for each edge in the graph depicted in Figure 3.20 in *Networks, Crowds, and Markets*. In the given graph, the edges between ('I','F') and ('A', 'D') have the highest betweenness values, 23.79047619047619 and 21.504761904761907, respectively. This signifies that the graph is highly reliant on those two edges for efficient paths in the graph. I.e, the removal of those edges would increase the average shortest path length much more than the removal of any other nodes.

edge in graph	betweenness of edge
'A', 'C':	14.771428571428569
'A', 'D':	21.504761904761907
'A', 'B':	14.771428571428569
'A', 'E':	16.03809523809524
'C', 'B':	2.0
'F', 'B':	12.371428571428572
'C', 'F':	12.371428571428572
'D', 'G':	17.752380952380953
'D', 'H':	12.99047619047619
'E', 'H':	14.704761904761904
'I', 'F':	23.79047619047619
'I', 'G':	17.038095238095238
'J', 'G':	12.219047619047618
'J', 'H':	17.504761904761903
'I', 'K':	14.752380952380953
'J', 'K':	13.419047619047621

Table 1: betweenness values for each edge in the graph

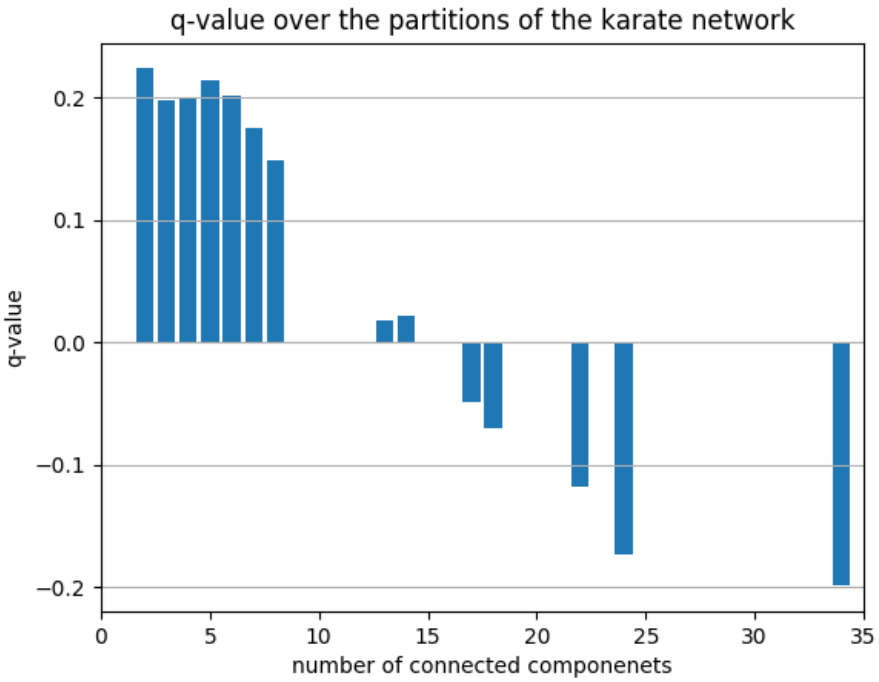
Problem 3:

Community Structure in Social Networks [29 pts]

1. *compute_q(g, c)* is given in both *autograder.py* and *analysis.py*

2. The question is: does the network structure itself contain enough information to predict the fault line, along which the club split? To answer this question, run gn_graph_partition on the karate club graph, and plot the values of Q you find. How many peaks do you observe? Report the community structure that achieves the highest modularity value Q. Does this community structure correspond to the community structure (i.e., two small clubs) observed by Zachary?

   Yes, the network structure itself indeed does include enough information to predict the fault line. There are 3 local maxima, at (2, 0.22353714661406987), (5, 0.21433267587113736), and (14, 0.0208744247205786). There is clear global peak at the first local maxima. The community structure with the highest modularity value Q is at 2 subgroups. It does indeed strongly correspond to the two small clubs observed by Zachary, except for nodes 3 and 9, which were mis-categorized. However, this is to be expected as the algorithm described does not always seek to find the best partition, and merely eliminates edges of highest betweenness.



3. In this problem, we will analyze a network of Rice University Facebook users (all of whom are students) and investigate whether the students form communities based on certain attributes. We have provided you with a subset of the Rice University Facebook graph in rice-facebook.repr that includes a sample of undergraduate students. The file rice-facebook-undergrads.txt provides data mapping the anonymized students with their college, major, and year. Remember that even though it has been anonymized, this data is not to be shared or made public.

   (a) For the provided subgraph of Rice undergraduates, plot the Q values when using the Girvan-Newman partitioning method on the graph.

   (b) Identify the community structure that corresponds to the highest value of Q. What is the highest value of Q? How many communities are there?

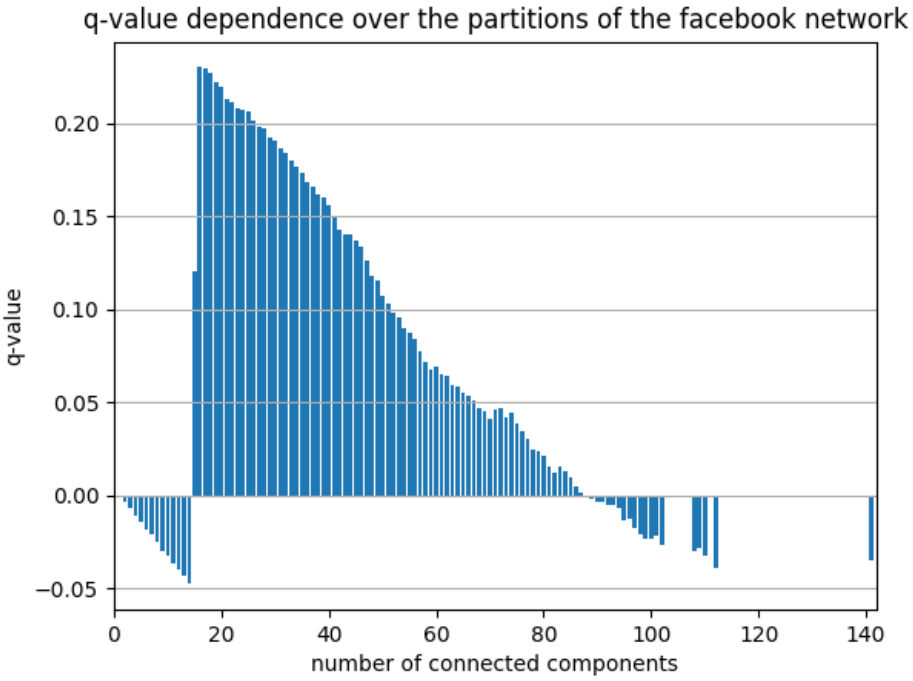
   The community structure that corresponds to the highest Q is that of 16 connected components, clocking in with a Q-value of 0.23004630790974534.

(c) What attribute(s) have the strongest influence on the community structure?

Given the following modularity values:

- q-value for college divisions: 0.2456681188549914
- q-value for year divisions: -0.24428582443465308
- q-value for major divisions: -0.0899314748997732

It is evident, then, that division among college lines provides the strongest communities, and the highest q-value. Note that the college communities are even higher than that of the maximum value for the Girvan-Newman method.



Problem 2

Shortest-path betweenness of edges: BFS revisited [26 pts]

1. Running bfs from Part 1 on graph g_1 in Figure 2, starting from node 0, report the values d_j and n_j for every node j in g_1. What quantity does the value of n_j, for a node j, reflect?

   Running bfs on the provided graph g1 yields the results:

   $$d = \{0:0, 1:1, 2:1, 3:2, 4:2, 5:3, 6:3\}$$
   $$n = \{0:1, 1:1, 2:1, 3:2, 4:1, 5:3, 6:1\}$$

   For a node j in g_1, n_j represents the number of paths that go through node j. More specifically, it is the number of shortest paths from the start node to node j.

2. The function is written in the provided code, *autograder.py*.

3. **Table 1** demonstrates the betweenness values for each edge in the graph depicted in Figure 3.20 in *Networks, Crowds, and Markets*. In the given graph, the edges between ('I','F') and ('A', 'D') have the highest betweenness values, 23.79047619047619 and 21.504761904761907, respectively. This signifies that the graph is highly reliant on those two edges for efficient paths in the graph. I.e, the removal of those edges would increase the average shortest path length much more than the removal of any other nodes.

edge in graph	betweenness of edge
'A', 'C':	14.771428571428569
'A', 'D':	21.504761904761907
'A', 'B':	14.771428571428569
'A', 'E':	16.03809523809524
'C', 'B':	2.0
'F', 'B':	12.371428571428572
'C', 'F':	12.371428571428572
'D', 'G':	17.752380952380953
'D', 'H':	12.99047619047619
'E', 'H':	14.704761904761904
'I', 'F':	23.79047619047619
'I', 'G':	17.038095238095238
'J', 'G':	12.219047619047618
'J', 'H':	17.504761904761903
'I', 'K':	14.752380952380953
'J', 'K':	13.419047619047621

Table 1: betweenness values for each edge in the graph