CS 630, Fall 2024, Homework 7 Solutions
Due *Tuesday*, December 10, 2024, 11:59 pm EST, via Gradescope

Homework Guidelines

Collaboration policy Collaboration on homework problems, with the exception of programming assignments and reading quizzes, is permitted, but not encouraged. If you choose to collaborate on some problems, you are allowed to discuss each problem with at most 5 other students currently enrolled in the class. Before working with others on a problem, you should think about it yourself for at least 45 minutes. Finding answers to problems on the Web or from other outside sources (including generative AI tools or anyone not enrolled in the class) is strictly forbidden.

You must write up each problem solution by yourself without assistance, even if you collaborate with others to solve the problem. You must also identify your collaborators. If you did not work with anyone, you should write "Collaborators: none." It is a violation of this policy to submit a problem solution that you cannot orally explain to an instructor or TA.

Typesetting Solutions should be typed and submitted as a PDF file on Gradescope. You may use any program you like to type your solutions. LaTeX, or "Latex", is commonly used for technical writing (`overleaf.com` is a free web-based platform for writing in Latex) since it handles math very well. Word, Google Docs, Markdown or other software are also fine.

Solution guidelines For problems that require you to provide an algorithm, you must provide:

1. pseudocode and, if helpful, a precise description of the algorithm in English. As always, pseudocode should include
   - A clear description of the inputs and outputs
   - Any assumptions you are making about the input (format, for example)
   - Instructions that are clear enough that a classmate who hasn't thought about the problem yet would understand how to turn them into working code. Inputs and outputs of any subroutines should be clear, data structures should be explained, etc.

   If the algorithm is not clear enough for graders to understand easily, it may not be graded.
2. a proof of correctness,
3. an analysis of running time and space.

You may use algorithms from class as subroutines. You may also use facts that we proved in class.

You should be as clear and concise as possible in your write-up of solutions. A simple, direct analysis is worth more points than a convoluted one, both because it is simpler and less prone to error and because it is easier to read and understand.

Problem 1 *Graph centrality (10 points)*

Recall from class the definition of betweenness centrality. Let $G(V, E)$ be an undirected unweighted graph, you may assume that G is connected.

We denote the *number of shortest paths* between two vertices s and t with $p(s, t)$, note that this is a symmetric notation as G is undirected. We also use the notation $p(s, t|v)$ to denote the number of shortest paths between s and t that contain vertex v. The betweenness of v is the fraction of shortest paths it covers for each pair of vertices

$$B(v) = \sum_{s \neq t \in V} \frac{p(s, t|v)}{p(s, t)}$$

In this problem you will develop an algorithm to efficiently compute the betweenness centrality of vertices.

1. Use Breadth First Search (BFS) to find the distance from a source s to each node. You may treat BFS as a blackbox when calling it as a subroutine in your algorithm. The running time of BFS is $O(|V|+|E|)$. Use the syntax $d_s \leftarrow BFS(G, s)$. Here d_s is an array of lists, such that the list in $d_s[i]$ contains the vertices at distance i from s. With this notation $d_s[0]$ consists of only s itself. (Think about it for yourself: what is the largest index in d_s?)

   Compute $p(s, t)$ for each pair of vertices $s \neq t$. For ease of notation you may want to store these values in an $O(|V|^2)$ hash table P, with $P[(s, t)] = p(s, t)$. *(Write pseudo code* [1] *for your algorithm and analyze its running time. No space complexity is needed. Prove the correctness of your algorithm, you may reference the correctness of BFS without proof.)*

   Solution. *syntax:* We assume that G is given in the format of an adjacency list implemented as a nested hash table. (The syntax is `For u in G` iterates over vertices in G, `For v in G[u]` iterates over neighbors of u.)*(You may use any meaningful implementation of G.)*

   In ComputeP( ) we compute the number of shortest paths from s to v as the sum along the directed edges (u, v) to v, such that the distance of v is distance u+1.

[1] In case you need a reminder, at the end of this document you can find the tex syntax for an algorithm template.

Algorithm 1: ComputeP(G)

```
   /* G is an adjacency list in a hash table                                    */
1  D ← hash table /* For ease of use later we store the distance between each pairs of
       nodes s and t.                                                          */
2  P ← hash table;
3  for s in G do
       /* initialize the pairs in P                                           */
4      P[(s, s)] ← 1;
5      for t in G do
           /* We'll take care of duplicates in part (c) of this problem, you can do it
              there or here.                                                   */
6          if s ≠ t then
7              P[(s, t)] ← 0;
8  for s in G do
9      d_s ← BFS(G, s);
10     for i = 0 to len(d_s) do
           /* update number of paths from s based on the parents of each vertex */
11         for u in d[i] do
12             D[(s, u)] ← i/* distance from s to u is i                       */
13             for v in G[u] do
                   /* directed edge (u,v) s.t.  v is at distance i+1 from s    */
14                 if v in d_s[i + 1] then
15                     P[(s, v)] ← P[(s, v)] + P[(s, u)]/* add the s- v paths through u to the
                           count                                               */
16 return P, D
```

Proof: We have to prove that $P(s,t)$ contains the number of shortest paths from s to t. We will do this by induction on the distance i from s.

base case i=0: Only s is at distance 0 from itself, and clearly there is just one way to get to itself.

inductive hypothesis: For any $k = 0 \dots i$ the value $P(s,u)$ is correct for vertices u in $d_s[k]$.

Proof for i+1: Suppose v is a node in $d_s[i+1]$. In our computation of $p(s,v)$ we consider the edges on the path form s to v. Specifically, we look at the final edge (u,v) in this path. Since the total length of the path is $i+1$, this means that u is in $d_s[i]$. Further, any shortest path from s to u can be extended to a path rom s to v by appending the edge (u,v). Thus the number of path from s to v *through* u is $p(s,u)$. We can make the argument for any incoming neighbor in $d_s[i]$ of v, which gives us the total value of $p(s,v)$.

Running time: We run BFS once for each vertex which takes $O(|V|(|V|+|E|)) = O(|V||E|)$ total. The computations to get $P[(s,v)]$ also takes $O(|V|+|E|)$ as we perform one condition check and one potential update for each edge. Note that in fact the BFS and the updates are performed consecutively, so their running time doesn't influence each other. Since updates are done for each s the second part also takes $O(|V|(|V|+|E|)) = O(|V||E|)$. This yields the overall running time.

2. Let *path* be a shortest path from s to t. Let x be a vertex on *path*. Then we us the notation

$path_{sx}$ and $path_{xt}$ to denote the two sections of $path$ from s to x and t to x.

Prove the following statement: If a vertex v is on $path$ then $path_{sv}$ and $path_{vt}$ are shortest path between the respective vertices.

Solution. *(This is an almost trivial statement which was included to suggest the approach you should take in part c.)*

Proof: Suppose that $path_{sx}$ is in fact not a shortest path from s to x and there is an alternative $path'_{sx}$ that is shorter. But in this case the combination of $path'_{sx} + path_{xt}$ would yield a shorter path than $path_{sx} + path_{xt}$ contradicting the fact that $path$ was the shortest. We can make the same argument for $path_{xt}$ being the shortest.

3. Design an algorithm to compute the table B, such that $B[v] = B(v)$. You should use your algorithm from part (a.) as a subroutine. For full credit your algorithm should run in time $O(|V|^3)$. *(Write pseudo code for your algorithm and analyze its running time, no space complexity analysis is needed. The proof from part (b.) serves as the proof of correctness for this algorithm.)*

Solution.

Algorithm 2: *ComputeB*(G)

```
   /* G is a graph adjacency list                                          */
 1 P, D ← ComputeP(G);
 2 B ← hash table;
 3 for v in G do
 4 |   B[v] ← 0;
 5 |   for s in G do
 6 |   |   for t in G do
 7 |   |   |   if D[(s,v)] + D[(v,t)] == D[(s,t)] then
   |   |   |   |   /* check that v is indeed on a shortest path between s and t  */
 8 |   |   |   |   B[v] ← B[v] + (P([s,v])·[P(v,t)])/P[(s,t)] /* compute dependency of s,t on v  */
 9 B[v] ← B[v]/2/* eliminate double count due to the symmetric role of s and t.  */
10 return B
```

Running time. ComputeP( ) contributes $O(|V||E|)$ to the total running time. The second part is a triple nested for loop of length $|V|$ each. the update to $B[v]$ takes constant time. The total running time of the algorithm is hence $O(|V|^3 + |V||E|) = O(|V|^3)$.

Text about Brandes' paper

Problem 2 *Centrality using an adjacency matrix (10 points)*

Let A be the adjacency matrix of an undirected graph with $|V|$ vertices.

write (short) pseudo code, analyze its running time - use k in your analysis - and give a short proof for both parts (a) and (b).

1. Compute the number of shortest paths between every pair of vertices using matrix multiplication. Compute running time if we know that the longest shortest paths in G consists of k edges.

Solution. From class, we know that $A^l[i,j]$ is the number of paths between the ith and jth vertices of length exactly l. (Those paths do not need to be simple paths and may repeat vertices if l is greater than the shortest path between the ith and jth vertices.) If l is less than the length of the shortest path between the ith and jth vertices, then $A^l[i,j]$ will be zero. For a given i and j, if l is the minimum value such that $A^l[i,j]$ is not zero, then l is the length of the shortest path between the ith and jth vertices, and $A^l[i,j]$ is the number of shortest paths between the ith and jth vertices. If k is the length of the longest shortest paths in G, then all the A^l up to A^k can be computed with $k-1$ matrix multiplications by A, and each matrix multiplication will take $O\left(|V|^3\right)$ time with the basic algorithm, so the total time will be $O\left(k|V|^3\right)$.

If more sophisticated matrix multiplication algorithms are used with exponent ω, then the total time will be $O\left(k|V|^\omega\right)$.

Algorithm 3: *ComputeNumSP*(A)

```
   /* A is a graph adjacency matrix                                          */
 1 NumSP ← |V| × |V| matrix./* mtx to store number of shortest paths between vertices */
 2 Dist ← |V| × |V| matrix./* mtx to store the distance of pairs of vertices */
 3 for i in 1..|V| do
 4 |   for j in 1..|V| do
 5 |   |   NumSP[i, j] ← 0;
 6 B ← A;
 7 l ←= 1;
 8 for l in 1..|V| − 1 do
   |   /* B = A^l                                                           */
   |   /* update NumSP where shortest path length is exactly l             */
 9 |   for i in 1..|V| do
10 |   |   for j in 1..|V| do
11 |   |   |   if NumSP[i, j] = 0 ∧ B[i, j] > 0 then
12 |   |   |   |   NumSP[i, j] ← B[i, j];
13 |   |   |   |   Dist[i, j] ← l/* the distance is the iteration in which the value is set
   |   |   |   |       */
14 |   if all(NumSP ≠ 0) then
15 |   |   return NumSP ;
16 |   B ← B × A /* matrix multiplication                                   */
   |   /* B = A^{l+1}                                                      */
   /* will only reach here if graph is not connected.                        */
17 return NumSP, Dist ;
```

2. Compute the betweenness centrality of the vertices using the adjacency matrix. You may call part (a) as a subroutine.

Solution. We will use the same approach as in problem 1, we can implement it as a nested for loop.

Algorithm 4: *ComputeBetweenness*(A)

```
   /* A is a graph adjacency matrix                                          */
1  NumSP, Dist ← ComputeNumSP(A);
2  B ← hash table/* betweenness of each vertex                              */
3  for i = 1 to |V| do
4      B[i] ← 0;
5      for j = 1 to |V| − 1 do
6          for k = j to |V| do
7              if D[j, i] + D[i, k] == D[j, k] then
                   /* i is on the shortest paths from j to k               */
8                  B[v] ← B[v] + NumSP[j,i]×NumSP[i,k] / NumSP[j,k]
9  return B
```