CS 170 Homework 3

Due **Friday 9/20/2024, at 10:00 pm (grace period until 11:59pm)**

1 Study Group

List the names and SIDs of the members in your study group. If you have no collaborators, explicitly write "none".

2 Hadamard matrices

The Hadamard matrices $H_0, H_1, H_2, \ldots$ are defined as follows:

- H_0 is the 1×1 matrix $[1]$
- For $k > 0, H_k$ is recursively defined as the $2^k \times 2^k$ matrix

$$H_k = \left[\begin{array}{c|c} H_{k-1} & H_{k-1} \\ \hline H_{k-1} & -H_{k-1} \end{array}\right]$$

For instance, the first three Hadamard matrices H_0, H_1, and H_2 are:

$$H_0 = \begin{bmatrix}1\end{bmatrix} \quad H_1 = \begin{bmatrix}1 & 1 \\ 1 & -1\end{bmatrix} \quad H_2 = \begin{bmatrix}1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1\end{bmatrix}$$

(a) Suppose that

$$v = \begin{bmatrix}v_1 \\ v_2\end{bmatrix}$$

is a column vector of length $n = 2^k$. v_1 and v_2 are the top and bottom half of the vector, respectively. Therefore, they are each vectors of length $\frac{n}{2} = 2^{k-1}$. Write the matrix-vector product $H_k v$ in terms of H_{k-1}, v_1, and v_2 (note that H_{k-1} is a matrix of dimension $\frac{n}{2} \times \frac{n}{2}$, or $2^{k-1} \times 2^{k-1}$). Since H_k is a $n \times n$ matrix, and v is a vector of length n, the result will be a vector of length n.

Solution: $H_k v = \begin{bmatrix}H_{k-1}v_1 + H_{k-1}v_2 \\ H_{k-1}v_1 - H_{k-1}v_2\end{bmatrix} = \begin{bmatrix}H_{k-1}(v_1 + v_2) \\ H_{k-1}(v_1 - v_2)\end{bmatrix}$

(b) Use your result from the previous part to come up with a divide-and-conquer algorithm to calculate the matrix-vector product $H_k v$, and show that it can be calculated using $O(n \log n)$ operations. Assume that all the numbers involved are small enough that basic arithmetic operations like addition and multiplication take unit time. You do not need to prove correctness.

Solution: Compute the 2 subproblems described in part (d), and combine them as described in part (c). Let $T(n)$ represent the number of operations taken to find $H_k v$.

We need to find the vectors $v_1 + v_2$ and $v_1 - v_2$, which takes $O(n)$ operations. And we need to find the matrix-vector products $H_{k-1}(v_1 + v_2)$ and $H_{k-1}(v_1 - v_2)$, which take $T(\frac{n}{2})$ number of operations. So, the recurrence relation for the runtime is:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

Using the master theorem, this give us $T(n) = O(n \log n)$.

3 Distant Descendants

You are given a tree $T = (V, E)$ with a designated root node r and a positive integer K. For each vertex v, let $d[v]$ be the number of descendants of v that are a distance of at least K from v. Describe an $O(|V|)$ algorithm to output $d[v]$ for every v. **Please give a 3-part solution; for the proof of correctness, only a brief justification is needed.**

Hint 1: write an equation to compute $d[v]$ given the d-values of v's children (and potentially another value).

Hint 2: to implement what you derived in hint 1 for all vertices v, we recommend using a graph traversal algorithm from lecture and keeping track of a running list of ancestors.

Solution:
Observe that for a vertex v,

$$d[v] = \sum_{c \text{ is a child of } v} d[c] + (\#\text{ descendants of } v \text{ at distance } K)$$

The algorithm proceeds as follows. Initialize all the $d[v]$ values to 0 and start a dfs at the root node. At every step of the algorithm, we will maintain the ancestors of the current node in a separate array. To ensure that our array only contains vertices on our current path down the DFS tree, we'll only add a vertex to our array (at index equal to the current depth) when we've actually visited it. Since a path can have at most n vertices, the length of this array is at most n.

Now, while processing the node v, we first index into the array equal to the index of its kth ancestor, call this ancestor a. Then we increment $d[a]$ to account for v. Once we finish processing a child c of v, we increment $d[v]$ by $d[c]$.

We provide pseudocode for this algorithm below:

```
def find_distant_descendants(G=(V, E), r, K):
    d = [0 for v in V]
    visited = [False for v in V]
    ancestors = []

    def explore(v):
        visited[v] = True

        # if possible, increment the d-value of the Kth ancestor by 1
        if len(ancestors) >= K:
            d[ancestors[-K]] += 1

        # add v to the list of ancestors and recurse on children
        ancestors.append(v)
        for each edge(v, u) in E:
            if not visited[u]:
                explore(u)
        ancestors.poplast()
        # if possible, increment the d-value of the parent
        if len(ancestors) >= 1:
            d[ancestors[-1]] += d[v]

    explore(r)
    return d
```

Runtime Analysis: Since we perform a constant number of extra operations at each step of DFS, the algorithm is still $O(|V|)$.

Correctness For a vertex v and a child c, every node counted in $d[c]$ should be included in $d[v]$ because their distance to v can only increase. Furthermore, nodes that are exactly K away from v will not be counted for any of its children, since they will be closer than K to the corresponding child. So, these get accounted for whenever we visit a kth descendant of v and increment $d[v]$. Notice that when we finish processing a node v, its $d[v]$ value will be correct and so it can be used by its parent.

4 Depth First Search

Depth first search is a useful and often efficient way to organize computations on a graph.

Let G be an undirected connected tree, and let $wt : E \to \mathbb{R}^+$ be positive weights on its edges. We show a template for depth-first search based computations below.

```
Input: Undirected connected tree $G = (V, E)$ and positive weights $wt(u, v)$ for each edge $(u, v) \in E$

Initialization:
$visited[v] \leftarrow False$ for all vertices $v$.
$L[v] \leftarrow 0$ and $M[v] \leftarrow 0$ for all vertices $v$.

function Explore(Vertex $u$)
    $visited[u] \leftarrow True$
    for each edge $(u, v)$ in $E$ do
        if NOT $visited[v]$ then
            PreVisit(u,v)
            Explore($v$)
            PostVisit(u,v)
```

DFS can be used for different purposes by defining the procedures PreVisit and PostVisit appropriately. In each of the following cases, write down pseudocode for PreVisit and PostVisit routines to perform the computation needed.

(a) For each vertex v, compute the maximum weight of an edge along the path from root r to vertex v and store it in array $L[v]$.

Solution:

```
procedure PreVisit($u, v$)
    $L[v] \leftarrow \max(L[u], wt[u, v])$

procedure PostVisit($u, v$)
    return
```

(b) For each vertex v, compute the maximum weight of any edge in the subtree rooted at vertex v and store it in array $L[v]$.

Solution:

```
procedure PreVisit($u, v$)
    return

procedure PostVisit($u, v$)
    $L[u] \leftarrow \max(L[u], L[v], wt[u, v])$
```

(c) For each vertex v, compute the depth of the tree rooted at vertex v, i.e., the length of the longest path from v to a leaf in its subtree, and store it in $L[v]$.

Solution:

```
procedure PreVisit($u, v$)
    return

procedure PostVisit($u, v$)
    $L[u] \leftarrow \max(L[u], L[v] + 1)$
```

(d) For each vertex v, compute the maximum degree among all the children of v and store it in $L[v]$

Solution:

```
procedure PreVisit($u, v$)
    $L[u] \leftarrow L[u] + 1$

procedure PostVisit($u, v$)
    $M[u] \leftarrow \max(M[u], L[v] + 1)$
```

5 Topological Sort Proofs

(a) A directed acyclic graph G is *semiconnected* if for any two vertices A and B, there is a path from A to B or a path from B to A. Show that G is semiconnected if and only if there is a directed path that visits all of the vertices of G. Make sure to prove both sides of the "if and only if" condition.

Hint: Is there a specific arrangement of the vertices that can help us solve this problem?

Solution: First, we show that the existence of a directed path p that visits all vertices implies that G is semiconnected. For any two vertices A and B, consider the subpath of p between A and B. If A appears before B in p, then this subpath will go from A to B. Otherwise, it will go from B to A. In either case, A and B are semiconnected for all pairs of vertices (A, B) in G.
Now we show that if G is semiconnected, then there is a directed path that visits all of the vertices. Consider a topological ordering $v_1, v_2, \ldots, v_n$ of the vertices in G. For any pair of consecutive vertices v_i, v_{i+1}, we know that there is a path from v_i to v_{i+1} or from v_{i+1} to v_i by semiconnectedness. But topological orderings do not have any edges from later vertices to earlier vertices. Therefore, there is a path from v_i to v_{i+1} in G. This path cannot visit any other vertices in G because the path cannot travel from later vertices to earlier vertices in the topological ordering. Therefore, the path from v_i to v_{i+1} must be a single edge from v_i to v_{i+1}. This edge exists for any consecutive pair of vertices in the topological ordering, so there is a path from v_1 to v_n that visits all vertices of G.

(b) Show that a DAG has a unique topological ordering if and only if it has a directed path that visits all of its vertices.

Remark: This means that a semiconnected DAG always has a unique topological ordering.

Solution: If a DAG has a directed path that visits all of its vertices, then arranging the vertices in the order they appear in the path will yield a topological ordering, as no backward edges can exist in this ordering since the graph has no cycles.

To prove the other direction, we'll proceed by contraposition. If a DAG does not have a directed path that visits all of its vertices, then by the previous part there exist two vertices A and B with no path between them. Then A and B can be interchanged and the resulting ordering will still be a valid topological ordering. Therefore, there exist at least two topological orderings so the topological ordering is not unique.

(c) This subpart is unrelated to the notion of semiconnectness. Consider what would happen if we ran the topological sorting algorithm from class on a directed graph that had cycles.

Prove or disprove the following: The algorithm would output an ordering with the least number of edges pointing backwards.

Solution: The statement is false. Consider the possible DFS traversal on this graph yielding the following pre- and post-numbers:

$$\begin{tikzcd}
A_1^6 \arrow[r] & B_2^5 \arrow[r] \arrow[l, bend right] & C_3^4 \arrow[l, bend right] \arrow[ll, bend right]
\end{tikzcd}$$

The SCC-finding algorithm would output the ordering specified in the graph, which has 3 edges pointing backwards. However, ordering the vertices in the reverse order would yield an ordering with only 2 edges pointing backwards.

6 [Coding] DFS & Edge Classification

For this week's homework, you'll implement implement DFS and use DFS to classify edges in a graph as forward/tree, backward, or cross edges. There are two ways that you can access the notebook and complete the problems:

1. **On Datahub**: click here and navigate to the `hw03` folder.
2. **On Local Machine**: `git clone` (or if you already cloned it, `git pull`) from the coding homework repo,

   `https://github.com/Berkeley-CS170/cs170-fa24-coding`

   and navigate to the `hw03` folder. Refer to the `README.md` for local setup instructions.

Notes:

- *Submission Instructions:* Please download your completed submission `.zip` file and submit it to the Gradescope assignment titled "Homework 3 Coding Portion".
- *Getting Help:* Conceptual questions are always welcome on Edstem and office hours; *note that support for debugging help during OH will be limited.* If you need debugging help first try asking on the public Edstem threads. To ensure others can help you, make sure to:

  1. Describe the steps you've taken to debug the issue prior to posting on Ed.
  2. Describe the specific error you're running into.
  3. Include a few small but nontrivial test cases, alongside both the output you expected to receive and your function's actual output.

  If staff tells you to make a private Ed post, make sure to include *all of the above items* plus your full function implementation. If you don't provide them, we will ask you to provide them.
- *Academic Honesty Guideline:* We realize that code for some of the algorithms we ask you to implement may be readily available online, but we strongly encourage you to not directly copy code from these sources. Instead, try to refer to the resources mentioned in the notebook and come up with code yourself. That being said, we **do acknowledge** that there may not be many different ways to code up particular algorithms and that your solution may be similar to other solutions available online.