CS170Hw3

Trustin Nguyen

September 20, 2024

Study Group

Rohan Kudchadker: 3038065006

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

$$H_k = \begin{bmatrix} H_{k-1} & H_{k-1} \\ H_{k-1} & -H_{k-1} \end{bmatrix}$$

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

$$H_0 = [1] \qquad H_1 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \qquad \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix}$$

– 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.

Answer. This can be more easily visualized by breaking down the Hadamard matrix as:

$$H_k = \begin{bmatrix} H_{k-1} & 0 \\ 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & H_{k-1} \\ 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ H_{k-1} & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & -H_{k-1} \end{bmatrix}$$

Then we multiply:

$$\begin{aligned} H_k v &= \left(\begin{bmatrix} H_{k-1} & 0 \\ 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & H_{k-1} \\ 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ H_{k-1} & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & -H_{k-1} \end{bmatrix} \right) \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} \\ &= \begin{bmatrix} H_{k-1} & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} + \begin{bmatrix} 0 & H_{k-1} \\ 0 & 0 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ H_{k-1} & 0 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & -H_{k-1} \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} \\ &= \begin{bmatrix} H_{k-1} v_1 + H_{k-1} v_2 \\ H_{k-1} v_1 - H_{k-1} v_2 \end{bmatrix} \end{aligned}$$

– 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.

Answer. We see that we can break the product down into a smaller product $H_{k-1} v_1$ and $H_{k-1} v_2$. So we are first given an $n \times n$ matrix. The dimension is scaled down by 2 each time. We compute two products $H_{k-1} v_1, H_{k-1} v_2$, so we have $T(n) = 2T(n/2)$ so far. We also have to take the sum and difference of the new vectors $H_{k-1} v_1, H_{k-1} v_2$. This means there are n additions at each level. The recursive equation is therefore, $T(n) = 2T(n/2) + O(n)$. There is n work at each level, $\log n$ levels. So the total number of operations is $O(n \log n)$.

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**.

Answer. The psuedocode:

```
d ← [0] * |V|

function distDescendants(node, k)
    res ← (1 if k ⩽ 0 else 0)
    for children in node.children:
        res += distDescendants(children, k)
    return res

for v in V:
    d[v] = distDescendants(v, K)
```

Proof. We sill show that for a vertex v, the algorithm runs as desired. In the base case, $K \leqslant 0$.

We will prove that at every level the number of descendants is the output for the node that distDescendants was called on. At the leaf level, the output will be 1 because there are no children. Suppose we are at a depth $d > 0$ and assume that the output for the previous levels were correct. Then at the current level, suppose that the input node was node n. The output of distDescendants run for each of n's children c_i will be the number of nodes below each c_i including c_i. We also add 1 to include n because $K \leqslant 0$. So it is clear that the output at the level $d > 0$ is correct.

Now we see that if $K > 0$, then res does not get incremented, the output above a certain level will be the same at the level when $K = 0$. Therefore, this is exactly what we want: the number of descendants that are at least distance d away from our start node. □

At each node, we do constant work, checking if res is 1 or 0. This means that

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.

```
1   Input: Undirected connected tree G and positive weights wt(u,v) for
    each edge (u,v) ∈ E
2
3   Initialization:
4   visited[v] ← False for all vertices v.
5   L[v] ← 0 and M[v] ← 0 for all vertices v.
6
7   function EXPLORE(Vertex u)
8   visited[u] ← True
9   for each edge (u,v) in E do
10      if NOT visited[v] then
11          PREVISIT(u, v)
12          EXPLORE(v)
13          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 $L[v]$.

Answer. We will use PREVISIT only:

```
1       function PREVISIT(u, v)
2           L[v] = max(L[u], wt(u, v))
```

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

Answer. We will use POSTVISIT only:

```
1       function POSTVISIT(u, v)
2           L[u] = max(L[u], max(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 $L[v]$.

Answer. We will use POSTVIST only:

```
1       function POSTVISIT(u, v)
2           L[u] = 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 $L[v]$.

Answer. We will use both PREVISIT and POSTVISIT. PREVISIT will set the degree for each vertex while POSTVISIT will calculate the max.

```
1       function PREVISIT(u, v)
2           M[u] = M[u] + 1
3           M[v] = 1
```

and for POSTVISIT:

```
1        function POSTVISIT(u, v)
2            L[u] = max(L[u], max(L[v], M[v]))
```

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.

Proof. ($\leftarrow$) To show that if there is a directed path that visits all vertices, then the graph is semiconnected. Let A, B be two arbitrary vertices. Then when we traverse our directed path, we will either run into A first or B first. Wlog, assume that we reach A first, then B. This means that there is a path to B from A. So the graph is semiconnected.

($\rightarrow$) We will now show that if a graph G is semiconnected, there exists a directed path that passes through all vertices. Suppose that our graph has a finite number of vertices. Then we can write a vertices set $[v_0, v_1, v_2, \ldots]$. We start by removing vertices out of our vertex set and reconstructing the graph. First, we remove v_0. There is no edges to fill in. Then we pull out v_1. We might need to pull out more vertices to fill in all the edges connecting v_0, v_1, but we know that there exists a path from v_0 to v_1 or v_1 to v_0. Wlog, assume $v_0 \rightarrow v_1$. Now consider the current graph that we have G_c. If there is path that can get to all vertices in G_c from v_0, then we can move on, otherwise, there exists a vertex that we cannot reach from v_0. Let this be v_i. Then since G is semiconnected, we can find a path from v_i to v_0, by pulling out more vertices/edges to add to G_c. Then v_i is our new root that can reach all other vertices, otherwise, we repeat. Since we have a finite number of vertices, this process must terminate. Then it is clear that there exists a root vertex from which we can reach all other vertices. □

(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.

Proof. ($\rightarrow$) If a DAG has a unique topological ordering, then wlog, suppose that we can write that ordering as a finite sequence:

$$x_1 x_2 x_3 \ldots x_n$$

We begin to reconstruct our graph one vertex at a time. Since this ordering is unique, we see that x_1 must be first. Then there cannnot be any edges from x_i to x_1. Now we see that x_2 is second and since it is unique, x_2 must lie after x_1. Since all that is in our graph so far is x_1, we know that there must be an edge from x_1 going to x_2. In general, we repeat. x_i must be after $x_1, x_2, \ldots, x_{i-1}$ and before x_{i+1}. So there is an edge from x_{i-1} to x_i. This shows that there is a directed path that visits all vertices:

$$x_1 \rightarrow x_2 \rightarrow x_3 \rightarrow \cdots \rightarrow x_n$$

($\leftarrow$) Suppose that there is a directed graph visits all vertices. Then we can write it down as:

$$x_1 \rightarrow x_2 \rightarrow x_3 \rightarrow \cdots \rightarrow x_n$$

Suppose that there is a different topological ordering lets call T. Then we see that any other permutation of the x_i must have an instance where x_j is before x_i in the ordering but $j > i$. Furthermore, there is a j such that x_{j-1} is after x_j.

Suppose that this does not happen. Then for every i, x_i goes after x_{i-1} in the topological ordering. But then we are at a contradiction because if we start at $i = n$, and work backwards, we see that x_{n-1} is before x_n, x_{n-2} must be before x_{n-1} and so on. We see that by the transitive property, our ordering is reduced to

$$x_1 x_2 \cdots x_n$$

as x_1 must go before all $x_{j>1}$ and so on.

This means that there exists a j such that x_j is before x_{j-1} in our topological ordering. But this is impossible and not a valid ordering because x_{j-1} must go before x_j. □

(c) This subpart is unrelated to the notion of semiconnectedness. 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.