Matrix multiplication

We saw in class how to use matrix multiplication to find if a graph has a triangle. However, for the recursive algorithm we saw that enumerates all triangles, we needed a variant of the algorithm, where the sets for each of the vertices are specified as part of the input. In the first question, you will design this variant.

1. Input: graph $G = (V, E)$, three sets of vertices $I, J, K \subset V$ of size $|I| = |J| = |K| = d$.
   Goal: find if $G$ contains a triangle $(i, j, k)$ with $i \in I, j \in J, k \in K$.
   Find an algorithm that solve this which runs in time $O(d^\omega)$.

The $(\min, +)$ matrix product (also called tropical matrix product) is defined as follows. Let $A, B$ be two $n \times n$ matrices of integers. Their $(\min, +)$ product $C$ is defined as

$$C_{i,k} = \min_{1 \leq j \leq n} A_{i,j} + B_{j,k}$$

Next, let $G = (V, E)$ be an undirected graph with $|V| = n$. The All-Pairs-Shortest-Path (APSP) matrix for $G$ is an $n \times n$ matrix $M$ such that $M_{i,j}$ is the distance in $G$ between vertices $i$ and $j$.

2. Assume that you have an oracle that computes the $(\min, +)$ product on two $n \times n$ integer matrices in time $T(n)$. Show how to compute the APSP matrix for a graph $G$ in time $O(T(n) \cdot \log n)$.

The simple algorithm to compute $(\min, +)$ matrix multiplication takes time $O(n^3)$. It is an important open problem whether there exists a significantly better algorithm, that is one that runs in time $O(n^{3-\epsilon})$ for some $\epsilon > 0$ and works for all integer matrices. However, if the matrices have small entries, there are better algorithms.

3. (a) Show that if $A, B$ have only $\{0,1\}$ entries, then their $(\min, +)$ product can be computed in time $O(n^\omega)$.
   (b) What is the best running time you can find when the entries belong to $\{0,1,\ldots,d\}$?
   (c) What bound on $d$ is needed to compute All-Pairs-Shortest-Path? Does this beat the simple $O(n^3)$ algorithm?
Fast Fourier Transform (FFT)

We learned in class fast univariate polynomials multiplication. It was based on FFT, which allows to compute their evaluations on many carefully chosen points in near-linear time. Here, we will develop similar ideas for polynomials on Boolean inputs. Such polynomials arise in various settings, like combinatorial optimization, learning problems, cryptography, etc.

Let $x_1, \ldots, x_n \in \{−1, 1\}$ be Boolean variables. A monomial is a product of a subset of the variables; a polynomial is a linear combination of monomials. So a general polynomial is of the form:

$$P(x) = \sum_{S \subseteq [n]} p_S \prod_{i \in S} x_i$$

The core of the FFT idea is that, given the set of $2^n$ coefficients of $P$, we can compute its evaluation on all $2^n$ Boolean inputs in near-linear time. Concretely, in time $O(n \cdot 2^n)$.

4. First, show a simple algorithm that evaluates $P$ on all the Boolean inputs, that takes time $O(4^n)$.

Define a $2^n \times 2^n$ matrix $H = H_n$ as follows. Its rows are indexed by $x \in \{−1, 1\}^n$. Its columns are indexed by subsets $S$ of $[n]$. For $x \in \{−1, 1\}^n, S \subseteq [n]$ define $H_{x,S} = \prod_{i \in S} x_i$. Note that $H_{x,S} \in \{−1, 1\}$.

5. Show that $(Hp)_x = P(x)$, when we view $p$ (the coefficients of $P$) as a vector of length $2^n$.

So, evaluating $P$ on all Boolean inputs is equivalent to multiplication of $H_n$ by a vector. For general matrices, this can take quadratic time. The FFT matrix is special: it can be multiplied by a vector in near-linear time. The main reason is that it has a recursive structure.

6. Show that:

$$H_n = \begin{pmatrix} H_{n-1} & H_{n-1} \\ H_{n-1} & -H_{n-1} \end{pmatrix}$$

Use this to design an algorithm which multiplies $H_n$ by a vector of length $2^n$ using only $O(n \cdot 2^n)$ additions and multiplications.

Hint: build it recursively.

Finally, we also need to solve the reverse problem sometimes: given the list of evaluations of $P$, compute its coefficients. This corresponds to matrix multiplication with the inverse matrix $(H_n)^{-1}$. Luckily for us, it is the same as $H_n$ (up to scaling), so the same algorithm works.

7. Show that $(H_n)^{-1} = 2^{-n} \cdot H_n$.

Hint: show that $(H_n)^2 = 2^n I$, where $I$ is the $2^n \times 2^n$ identity matrix.