May 1, 2020

Determinant using permutations & permanent in graph theory: a $\mathscr{P}$ vs. $\#\mathscr{P}$ complete problem.

Notes and Coding Exercises.

Rohith Krishna

PGDM in Research and Business Analytics
Madras School of Economics

mailto: pgdm19rohith@mse.ac.in

1 Introduction

The idea of determinant and permanent play an important role in Linear Algebra. This report is a collection of topics that highlight the interconnectedness of linear algebra with the fields of graph theory, combinatorics and computation using determinant and permanent. The definitions of determinant and permanent are given followed by a statement of their properties. Subsequently, determinant is connected to combinatorics by using permutation as a method to calculate determinants. Further, the application of permanent in graph theory is also explored.

1.1 Definitions

The determinant of a matrix is a linear function on rows or columns of that matrix. It has several interpretations; notably the geometric and algebraic ones. The geometric interpretation of the determinant of a matrix is that it gives the volume of the parallelepiped formed by taking the columns of the matrix as vectors. The algebraic interpretation is that the determinant of a matrix is the product of all the eigenvalues of that matrix. The permanent has a combinatorial interpretation in that, if A is a matrix with non-negative numbers, the permanent is the sum of weights of all perfect matchings of the bipartite graph represented by A.

The ***determinant*** for a matrix $X = [x_{i,j}];\ 1 \leq i, j \leq n$ is defined as:

$$\det X = \sum_{\pi \in S_n} \operatorname{sgn}(\pi) \cdot \prod_{i=1}^{n} x_{i,\pi(i)} \tag{1.1}$$

where S_n is a group of all permutations on $[1, n]$ and sgn (π) is the sign of the permutation π, such that $\operatorname{sgn}(\pi) \in [-1, 1]$. The ***permanent*** for a matrix is defined as

$$\operatorname{perm} X = \sum_{\pi \in S_n} \cdot \prod_{i=1}^{n} x_{i,\pi(i)} \tag{1.2}$$

We see that despite the similarity in definition of permanent and determinant, they have very different properties. Particularly, while the computation algorithms of determinant pertains to the $\#\mathscr{P}$ complexity class of problems, the computation of permanent is a much harder problem and falls under the $\mathscr{NP}$ class.

1.2 Permutations

A permutation of the set $1, 2, 3, ..., n$ is defined as list of its n elements where each element appears exactly once in the list. For instance, the six permutations of the set $1, 2, 3$ are $123, 132, 213, 231, 312$, and 321. In general,there are $n!$ permutations of a set of size n. A permutation σ is often written as a list, such as $\sigma = 24153$, but one can also treat it as a function on the set. For example in the set of first five natural numbers, the permutation σ is a function on the set $1, 2, 3, 4, 5$, where σ sends 1 to 2, 2 to 4, 3 to 1, 4 to 5, and 5 to 3; then we write $\sigma_1 1 = 2$, $\sigma_2 = 4$, $\sigma_2 = 1$, $\sigma_2 = 5$, and $\sigma_2 = 3$.

2 Determinants and permutations

2.1 Intuition for computing determinant using permutations

The determinant of a square $n \times n$ matrix is expressed as a sum or difference of $n!$ terms, each term being a product of n elements, with one element chosen out of each row and column. There are n! ways to choose one element out of each row and column. This is because each choice is determined by which column is chosen for each row. So, there are $n!$ choices, and each corresponds to a permutation.

We now form all $n!$ products of n elements, one element chosen out of each row and column. Half of these n! are positive terms and are added while, the other half are negative, with the result being the determinant. Every even transposition is added while an odd transposition is subtracted. A ***transposition*** is a particularly simple permutation which exchanges exchanges exactly two elements and leaves all the others fixed. Some transpositions of the set of 5 elements, for example, are 21345, 12543, 53413 etc.

2.2 Algorithm to generate permutations of a given set

The first algorithm is to generate an array of permutations of the n elements. It was already mentioned that there are $n!$ such permutations for n given elements. The algorithm to obtain permutations of the first n numbers starting from zero is elaborated here.

1. Initialize the permutation list. This shall be the original permutation with which other permutations are compared for even or odd transpositions.
2. While the count of the total permutations is less than $n!$, use the swapping method with variable indices to obtain different permutations.
3. Add the newly obtained permutations to the existing list with a non-repetition condition.
4. Any left out permutations can by obtained by redoing the same process with a different starting position of iteration and by taking a different permutation as the primer permutation,
5. Return the list of permutations and end the program.

2.3 Program to generate permutations of a given set

Note: `pNum` is the number of given elements in the set. The set consists of numbers $0, 1, 2, \cdots$, `pNum`. The above algorithm is implemented and the resulting output is displayed for `pNum` = 3. `originalPermutation` contains [0,1,2] in this case. All other permutations, 6 in total number, are obtained as shown below.

```
# Generating permutations
import math
```

```
def generatePermutations(pNum):

    iStartPos = 0 # Starting position index
    iList = 0     # Indexing in a list of permutations

    # Initialize the original permutation as numbers
    # starting from 0 to pNum
    originalPermutation = tuple(range(pNum))
    print("The original permutation is", originalPermutation)
    # permutationList contains all permutations generated
    permutationList = [originalPermutation]

    # A total of pNum! permutations can be made
    while (len(permutationList) < math.factorial(pNum)):
        for index in range(pNum-iStartPos):
            # Generating permutations by swapping
            generatedPerm = permutationList[iList]
            generatedPerm = list(generatedPerm)
            tempPerm = generatedPerm[iStartPos] #swap variable
            generatedPerm[iStartPos]=generatedPerm[iStartPos+index]
            generatedPerm[iStartPos+index] = tempPerm

            # Add generated permuatation to the list
            if (tuple(generatedPerm) not in permutationList):
            permutationList.append(tuple(generatedPerm))

        # To derive the other permutations using swap method
        # Change starting position and the list taken
        if (iStartPos >= pNum):
            iStartPos = 0
            iList = iList + 1
        else:
            iStartPos = iStartPos + 1
    return permutationList

print("List of permutations: \n",generatePermutations(3))
```

Output

```
The original permutation is (0, 1, 2)
List of permutations:
[(0, 1, 2), (1, 0, 2), (2, 1, 0), (0, 2, 1), (2, 0, 1), (1, 2, 0)]
```

2.4 Algorithm to generate the number of the transpositions

The number of the transpositions is generated here. The first element in the list of all $n!$ permutations is kept as the base permutation. From this, several swaps are considered between elements in the set. The number of swaps it takes to reach the original permutation is counted and stored in an array. For each permutation in `permutationList` there

exists a corresponding element in the `transpositionList` that contains the number of transpositions of that particular permutation.

2.5 Program to generate the number of transpositions

```
pNum = 3
count = 0
transpositionList =[]
tempPermutation = []
permutationList=generatePermutations(pNum)

# obtaining the number of transpositions per permutation
for permutation in permutationList:
    tempPermutation = list(permutation)
    for iPerm in range(len(tempPermutation)):
        while (tempPermutation[iPerm] != iPerm):
            iSwap = tempPermutation[iPerm]
            tempPerm=tempPermutation[iPerm]
            tempPermutation[iPerm]= tempPermutation[iSwap]
            tempPermutation[iSwap] = tempPerm
            count += 1

transpositionList.append(count)
count = 0

print("List of permutations: \n",permutationList)
print("List of transpositions (odd/even): \n",transpositionList)
```

Output

```
The original permutation is (0, 1, 2)
List of permutations:
[(0, 1, 2), (1, 0, 2), (2, 1, 0), (0, 2, 1), (2, 0, 1), (1, 2, 0)]
List of transpositions (odd/even):
[0, 1, 1, 1, 2, 2]
```

We see from the above that it takes only one transposition to go to (0,2,1) while it takes two transpositions to reach (1,2,0).

2.6 Program to compute the terms of the determinant

Combining the above to sections, the number of transpositions and the permutations list are fed into the determinant terms generating function. Here, the permutation elements are used to index the terms of the determinant, while the number of transpositions gives the parity of the terms: +1 for even parity and -1 for odd parity transpositions.

```
import math
def genPermutesAndTranspositions(pNum):

    iStartPos = 0
```

```
    iList = 0
    count = 0
    originalPermutation = tuple(range(pNum))
    permutationList = [originalPermutation]

    # A total of pNum! permutations can be made
    while (len(permutationList) < math.factorial(pNum)):
        for index in range(pNum-iStartPos):
            # Generating permutations by swapping
            generatedPerm = permutationList[iList]
            generatedPerm = list(generatedPerm)
            tempPerm = generatedPerm[iStartPos]
            generatedPerm[iStartPos] = generatedPerm[iStartPos+index
    ]
            generatedPerm[iStartPos+index] = tempPerm
            if (tuple(generatedPerm) not in permutationList):
            permutationList.append(tuple(generatedPerm))

        # To derive the other permutations using swap method
        # Change starting position and the list taken
        if (iStartPos >= pNum):
            iStartPos = 0
            iList = iList + 1
        else:
            iStartPos = iStartPos + 1

    transpositionList =[]
    tempPermutation = []

    # obtaining the number of transpositions per permutation
    for permutation in permutationList:
        tempPermutation = list(permutation)
        for iPerm in range(len(tempPermutation)):
            while (tempPermutation[iPerm] != iPerm):
                iSwap = tempPermutation[iPerm]
                tempPerm=tempPermutation[iPerm]
                tempPermutation[iPerm]= tempPermutation[iSwap]
                tempPermutation[iSwap] = tempPerm
                count += 1

        transpositionList.append(count)
        count = 0

    # Return list of permutations and swaps
    return permutationList, transpositionList

def determTerms(matrix):
    term = 0     #term variable
```

```
    determinant = 0      #determinant
    factor = 1
    termList = []

    dim = len(matrix)
    # Generate permutation list
    permList, swapList = genPermutesAndTranspositions(dim)
    for iPermutation,permutation in enumerate(permList):
        for iOne,iTwo in enumerate(permutation):
            factor *= matrix[iOne][iTwo]

        term = (-1)**swapList[iPermutation] * factor
        termList.append(term)
        determinant += term
        factor = 1
    return determinant, termList

Matrix = [[2,3,4],[-1,5,6],[1,1,2]]
print(determTerms(Matrix))
```

Output

```
(8, [20, 6, -20, -12, -4, 18])
```

From the above, we see that the determinant for the 3×3 matrix is correctly computed as the addition of the six terms displayed above. Further examples elucidate why this method of computation of determinant is highly inefficient.

2.7 A 6×6 integer matrix

For a 6×6 integer matrix, the determinant terms are calculated. As is predicted, there are $6! = 720$ terms in the expansion.

```
Matrix = [[2,3,4,5,6,7],
[-1,5,6,2,3,4],
[1,2,-5,0,1,1],
[0,3,-3,1,2,7],
[2,2,2,-1,-1,-1],
[1,0,3,-6,1,1]]

terms = determTerms(Matrix)
print(terms)
print(math.factorial(6))
print(len(terms[1]))
```

Output

```
(19422, [50, 15, 20, 0, 300, -175, 24, -60, 60, 0, 0, -20, 30, -100, 2100,
 -50, 8, -75, 60, 0, -18, 0, -90, 60, 0, -6, 9, -30, 630, -15, 75, 60, -105,
  -24, 24, 0, 0, 40, -20, -40, 840, -20, 0, 0, 0, 0, 0, 0, 0, -500, 875, 0,
```

```
  -350, 144, -360, 0, 0, 180, 12600, 150, -84, 210, -210, 0, 70, 350, 24, 24,
  -48, 0, 24, 0, -48, 1008, -24, 90, -720, 24, -36, -80, 0, 60, -80, 0, 36,
  2520, 0, 0, 0, 0, 30, -180, 0, 0, 0, 20, -840, -30, -60, 350, -600, 30, 24,
  -42, 0, -48, 32, 0, 8, 0, -16, 336, -8, 90, -630, 90, 450, -300, 30, -45,
  -100, 0, 75, -70, -72, 0, 240, 0, 36, 2520, 0, 0, 0, 0, 0, 0, 0, -18, -18, 36,
  0, -36, 18, 36, -756, 18, 0, 0, 0, 0, 120, -210, 0, 120, 0, -54, -3780, -45,
  0, -24, -120, 9, -54, 0, 0, 0, 6, -252, -9, -18, 105, -180, -90, 630, 90, 0,
  150, -75, -100, 525, -75, -70, -72, 0, 0, -60, 2520, 90, 126, -126, 0, -210,
  210, 36, -288, -48, 24, -32, 0, 24, -32, 0, -24, 1008, 0, 0, 0, 0, 0, 0, 0, 0,
  0, -40, 1680, 20, 40, 140, -240, 0, 0, 0, 0, -1050, 0, 0, 0, 0, -4200, 0, 0,
  0, 0, -240, 420, 0, 0, 0, 1050, 0, 0, 0, 0, -350, 0, -300, 1750, -250, -168,
  420, 0, 144, -288, 0, 0, 6048, 72, -4320, -216, 0, -180, 0, -1080, 0, 0,
  -84, -84, 0, -84, 168, -315, -84, 280, 0, -105, 0, -36, 288, 24, 0, -32,
  168, -24, -32, 0, 0, 1008, 36, 0, -48, 96, 36, -216, 0, 0, 0, -24, 1008, 0,
  0, 168, -288, -120, 54, 0, 540, 288, -960, -24, 0, 36, -48, 280, 0, 0,
  -216, 0, 0, 0, 0, -30, 210, 180, 240, 0, 0, -36, 252, -180, 120, 30, 0,
  -40,  210, -30, -28, 0, 96, 0, 0, 1008, 36, 0, 252, 0, -42, 84, 0, 0, 0, 0,
  64, -112, 0, 64, 0, 0, -2016, -24, 0, 32, -64, 12, -72, 0, 0, 0, -8, 336, 0, 0,
  56, -96, -105, -108, 360, 54, 0, 540, 756, 3780, 252, -840, 90, -180,
  180, -90, 120, 0, -90, -600, 270, 0, 225, 120, -400, -30, 0, 45, -60, 350,
  0, 84, 0, 0, -72, 144, 0, 72, -3024, 0, 0, 0, -840, 0, 0, -216, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 27, -216, -36, 18, 24, -126, 18, 24, 0, 18,-756, -27,
  0, 72, -72, 0, 0, 0, 0, 0, 36, -1512, -18, -36, -126, 216, 0, 0, 315, 0, 0,
  1440,  0, 84, 0, 72, -420, 60, 0, 324, 0, 0, 36, 0, -9, 63, 54, 72, 0, 0,
  105, 0, 90,  -630, -540, 756, 1260, -840, -120, -90, 630, 0, 0, 0, -150,
  1050, 75, 100, 350, -150, 84, 0, -864, 72, 0, -108, 0, 0, 0, 0, -189, 252,
  168, 0, 0, 0, -48, -36, 0, 216, -576, -384, 48, 0, -24, 32, 112, 0, 0, 0, 0,
  0, 0, 0, 0, -140, 0, -480, 0, 0, 0, 0, 0, -504, 0, 0, 0, -2016, 0, 0, 420, 0,
  0, 480, 0,  420, 0, 0, 840, -120, -168, 0, 1728, 0, 1008, 72, 0, -1296, 0,
  0, 126, -84,  112, 0, -126, 0, 48, 0, -252, -216, 288, -384, -24, 168, 0,
  0, 112, -48, 0,  -216, 0, 0, -72, 0, -36, 0, 216, -288, 0, 0, 42, -144, 0,
 -252, -216, -1512, 252, -336, 240, 0, -1260, -90, 120, -160, -30, 210, 0, 0,
 140, -60, 0, 0, 0, 0, -336, 0, 0, -216, 0, 0, 0, 0, -336, 0, -63, 0, 0, 0, 168,
 0, 0, 768, 0,  -112, 0, 0, -224, 32, 0, 432, 0, 0, 48, 0, -12, 0, 72, -96, 0, 0,
 126, 216,  108, 0, -648, 756, 1512, 1008, -120, -90, 0, 135, -360, -240,
 -180, 0, 90, -120, -420, 0, 84, 0, -864, 72, -504, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, -36,  -27, 189, 162, -432, 288, 36, -252, -18, -24, -84, 36, 0, 0, 0, 0,
 0, 0, 0, 126, 0, 432, 0, 0])
720
720
```

2.8 A 7×7 Hadamard matrix

For a 7×7 integer matrix, the determinant terms are calculated. As is predicted, there are $7! = 5040$ terms in the expansion. At this point, the time complexity is only a matter of a fraction of a second. However, this blows up very soon.

```
import time
start_time = time.time()
Matrix = [[-1, 1, 1, 1, -1, 1, 1],
                [ 1, -1, 1, 1, 1, -1, 1],
                [ 1, 1, -1, 1, 1, 1, -1],
```

```
                [ 1,  1,  1, -1,  1,  1,  1],
                [-1,  1,  1,  1,  1, -1, -1],
                [1, -1,  1,  1, -1,  1, -1],
                [1,  1, -1,  1, -1, -1,  1]]

terms = determTerms(Matrix)
print(terms)
print(math.factorial(7))
print(len(terms[1]))
    print("--- %s seconds ---" % (time.time() - start_time))
```

Output

```
(512, [1, -1, -1, -1, 1, 1, 1, -1, ..........,  -1, 1, -1, 1, -1, 1, 1, 1])
5040
5040
--- 0.07550477981567383 seconds ---
```

2.9 An 8×8 Hadamard matrix

For a 8×8 integer matrix, the determinant terms are calculated. As is predicted, there are $8! = 40320$ terms in the expansion. At this point, the time complexity is severe - about 380 seconds. The exponential blow up therefore makes this a very inefficient method to compute determinants.

```
import time
start_time = time.time()
Matrix = [[-1,  1,  1,  1, -1,  1,  1,  1],
                [ 1, -1,  1,  1,  1, -1,  1,  1],
                [ 1,  1, -1,  1,  1,  1, -1,  1],
                [ 1,  1,  1, -1,  1,  1,  1, -1],
                [-1,  1,  1,  1,  1, -1, -1, -1],
                [1, -1,  1,  1, -1,  1, -1, -1],
                [1,  1, -1,  1, -1, -1,  1, -1],
                [1,  1,  1, -1, -1, -1, -1,  1]]

terms = determTerms(Matrix)
print(terms[0])
print(math.factorial(8))
print(len(terms[1]))
print("--- %s seconds ---" % (time.time() - start_time))
```

Output

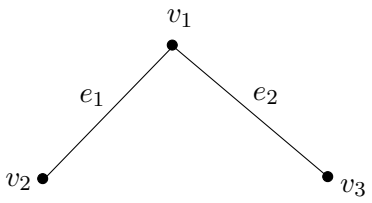
```
4096
40320
40320
--- 378.8883421421051 seconds ---
```

3 Review of graph theory

Definition. A graph G is a set of vertices V and edges E. $\implies G = \{V, E\}$.
Here, the set of vertices is given by $V = \{v_1, v_2, \ldots, v_n\}$ where, $|V| = n$ and the set of edges is given by $E = \{e_1, e_2, \ldots, e_n\}$ where, $|E| = m$.

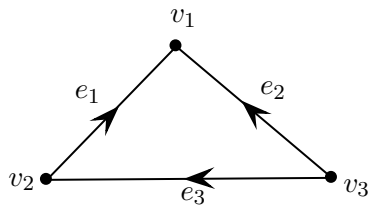
Definition. The degree of a vertex v in a graph is the number of edges incident on $v \in V$.

Example 1. Consider the following undirected graph. The degrees of the vertices of the graph have been calculated.



$$\begin{aligned} d_1 &= \text{Deg}\{v_1\} = 2 \\ d_2 &= \text{Deg}\{v_2\} = 1 \\ d_3 &= \text{Deg}\{v_3\} = 1 \end{aligned}$$

Example 2. Consider the following directed graph. There are two types of degrees for these vertices in directed graphs, namely, d_{out} and d_{in}.



	v_1	v_2	v_3
d_{in}	2	1	0
d_{out}	0	1	2

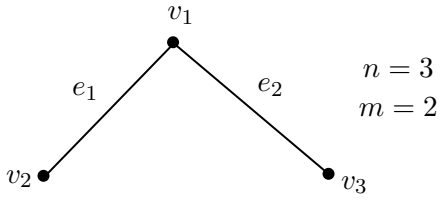
The sum of degrees for a graph can be generalised as:

$$\boldsymbol{d^T} \cdot \mathbf{1} = \sum_{i=1}^{n} d_i = 2m = 2|E| \tag{3.1}$$

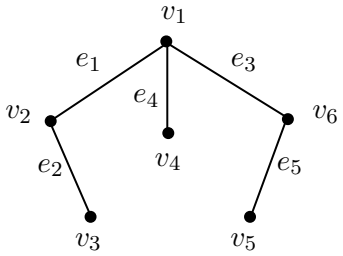
This shows that the degree of a graph is given by double counting the edges as every edge will be included from both the vertices.

Definition. A tree T is a connected acyclic graph.

Example 3. Consider the following trees. We observe an interesting relationship between the number of vertices and edges, $m = n - 1$ or $|E| = |V| - 1$.



$n = 3$
$m = 2$



$n = 6$
$m = 5$

Therefore, the sum of degrees for a tree can be generalised as:

$$\sum_{i=1}^{n} d(v_i) = 2m = 2(n-1) \tag{3.2}$$

4 Relation between Linear Algebra and Graph Theory

We know that the dimension of Null space of a matrix is related to the rank of the matrix. Recall that,

$$\text{Dim}[\boldsymbol{N}(A^T)] = m - r \tag{4.1}$$

where m is the number of edges and r is the number of nodes minus one. The $\text{Dim}[\boldsymbol{N}(A^T)]$ gives the number of independent loops L in the graph. The number of rows in the incidence matrix A is equal to the number of edges $|E|$ in the loop. The rank of the incidence matrix r (which is invariant under the transpose operation) is one less than the number of nodes $|V| - 1$. Therefore,

$$|E| + |V| - 1 = L \tag{4.2}$$

$$|V| - |E| + L = 1 \tag{4.3}$$

The above formula can be interpreted as a variant of Euler's formula in topology or the Gibb's Phase Rule in thermodynamics. Note that the first term represents zero-dimensional vertices, the second denotes one-dimensional lines and the third denotes two-dimensional loops.

Note.

- A graph G is a set of vertices V and edges E. $\implies G = \{V, E\}$.
  Here, the set of vertices is given by $V = \{v_1, v_2, \ldots, v_n\}$ where, $|V| = n$ and the set of edges is given by $E = \{e_1, e_2, \ldots, e_n\}$ where, $|E| = m$.
- The degree of a vertex v in a graph is the number of edges incident on $v \in V$.
- A tree T is a connected acyclic graph.
- Therefore, the sum of degrees for a tree can be generalised as: $\sum_{i=1}^{n} d(v_i) = 2m = 2(n-1)$
- The Euler's formula in topology, given by $|V| - |E| + L = 1$ is linked to graph theory. Here $|V|$ is the number of vertices, $|E|$ is the number of edges and L is the number of independent loops.

5 Determinant vs. Permanent is $\mathscr{P}$ vs. $\mathscr{NP}$

Definition. Let $G = (V, E)$ be an undirected graph. A subset $M \subset E$such that no two edges in M share an endpoint is called a matching. If M *matches* all vertices, i.e., $|M| = |V|/2$ then it is defined that M is a *perfect matching*. A popular graph theoretic problem in relation to permanent is as follows: Given a bipartite graph G, count the number

of perfect matchings in G. This problem is denoted by BMCount.

The computing of permanent of a non-negative matrix is related closely to the problem of counting perfect matchings BMCount. Recall from the first section the definition for the permanent of a matrix:

$$\text{perm } X = \sum_{\pi \in S_n} \cdot \prod_{i=1}^{n} x_{i,\pi(i)} \tag{5.1}$$

where S_n is a group of all permutations on $[1, n]$ and sgn (π) is the sign of the permutation π, such that $\text{sgn}(\pi) \in [-1, 1]$. The ***determinant*** for matrix A was defined as:

$$\det X = \sum_{\pi \in S_n} \text{sgn}(\pi) \cdot \prod_{i=1}^{n} x_{i,\pi(i)} \tag{5.2}$$

The Permanent problem Perm is stated as follows: given a matrix A with non-negative entries, compute perm(A). We know that computing determinant using permutations is an inefficient method. However, there exists the Gaussian elimination method by reducing the matrix into the lower triangular form and thereby multiplying the diagonal terms to obtain the determinant. Since this can be done in polynomial time, it is tempting to think that a permanent can also be computed much the same way. However, it turns out that this problem is $\#\mathscr{P}$ complete - and hence no polynomial algorithm would be able to solve this.

6 Note on the $\mathscr{P}$ vs. $\mathscr{NP}$ problem

The P vs. NP problem is one of the seven millennium prize problems selected by the Clay Mathematical Institute. In its multiple forms we see the P vs. $\mathscr{N}P$ problem arising in several diverse fields of natural sciences - physics, biology; finance etc, and is therefore of vitality. A quick attempt to review the $\mathscr{P}$ vs. $\mathscr{NP}$ problem is made here, beyond which the connection of $\mathscr{P}$ vs. $\mathscr{NP}$ problem and ***Determinant vs. Permanent*** problem is presented.

Polynomial time: An algorithm is said to be solvable in polynomial time if the number of steps required to complete the algorithm for a given input is $O(n^k)$ for some non-negative integer k, where n is the input size in units of bits needed to represent the input.

$\mathscr{P}$ vs $\mathscr{NP}$ problem: There are two major classes of problems that theoretical computer scientists have identified according to their complexity. Those problems which can be solved using some algorithm in deterministic polynomial time fall under the complexity class $\mathscr{P}$. For example, multiplication or sorting are class $\mathscr{P}$ problems. For some problems, there is no known way to solve it quickly, but if one is provided with information showing the answer, it is possible to verify the answer quickly. The class of problems which can be verified in polynomial time, but cannot be solved in polynomial time are called $\mathscr{NP}$ problems, short for non-deterministic polynomial time.

One can quickly assert that all problems that can be solved in polynomial time can also be checked for, in polynomial time. Thus it can be said that all P problems form a subset of $\mathcal{NP}$ problems.

$\mathcal{NP}$ **completeness:** A set of problems are $\mathcal{NP}$ complete when each of them can be reduced to another $\mathcal{NP}$ problem in polynomial time. Thus informally it can be said that an $\mathcal{NP}$ complete problem is at least as tough as any other $\mathcal{NP}$ problem. Further solving one $\mathcal{NP}$ problem would imply solving all $\mathcal{NP}$ complete problems in polynomial time.

Cook-Levin theorem: This theorem states that the satisfiability problem is $\mathcal{NP}$ complete. This means that any problem in $\mathcal{NP}$ can be reduced in polynomial time by a deterministic Turing machine to the problem of determining whether a Boolean formula is satisfiable.

If $\mathcal{P} = \mathcal{NP}$, then that would mean that all problems which can be verified in polynomial time, can also be solved in polynomial time. This has widespread ramifications to the way of life as we know it. Many problems in operations research such as integer linear programming and the travelling salesman problem can be efficiently solved, leading to widespread impact on logistics. Further, solving the $\mathcal{NP}$ complete protein folding problem would lead to significant advances in life sciences and medicine.

Negative consequences would also arise, for $\mathcal{NP}$ complete problems are fundamental to several disciplines. For instance cryptography relies on certain problems being hard to solve. Finding successful and efficient ways to solve $\mathcal{NP}$ complete problems would lead to breakage of many cryptosystems, thereby throwing individuals' privacy and security at risk.

Declaration. The notes on the concepts of permanent and determinant were learnt from the book **Combinatorial Matrix theory** by Brualdi et al. Permutation generation by swapping algorithm was understood by the method followed in Rosetta Code. Further, the section on graph theory was discussed with Ishita Gupta while being co-written for lecture notes.

Some references

1. W.M. Gentleman and S.C. Johnson. The evaluation of determinant by expansion of minors and the general problem of substitution. Mathematics of computation. 1974.
2. Mahindra Agrawal, Determinant Versus Permanent, IIT Kanpur, 2000.
3. Nisheeth Vishnoi, Permanent and Counting Perfect Matchings in Bipartite Graphs, IISc Bangalore, 2017
4. Jayalal Sarma, A Combinatorial Interpretation of Permanent, 2012
5. Uriel Fiege, Lecture Notes on Permanent and Determinant. The Weizman Institute. 2014