Permutation Detection Algorithm with Linear Complexity

Ezra A. Singh

April 29, 2017

Abstract

The algorithm will use a product of indexed primes to generate permutation invariant value. This method allows for $\mathcal{O}(N)$ possibly one of the fastest methods to date for this type of problem.

The Problem

Given two sets A and B of symbols with finite length, how can we determine if set A permutes set B. We will use the base 10 radix to define our set of symbols. Such that A^N defines set of length N where

$$A^N = \{a_0, a_1, a_2, ..., a_{N-2}, a_{N-1}\} \tag{1}$$

$$a_j \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}; \qquad j \in [0, N) \tag{2}$$

Next we define a function P, the permutation function, which takes in two sets A^N and B^N and returns 1 if the two sets permute each other else returns some other number. Our goal is to explicitly define an algorithm that resolves $P(A, B)$ in linear time.

The Permutation Algorithm

Let us first map our symbol set to the set of prime numbers

$$B_{10} \longrightarrow \mathcal{P}^{10} \tag{3}$$

$$\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \longrightarrow \{2, 3, 5, 7, 11, 13, 17, 19, 23, 29\} \tag{4}$$

We will then introduce an indexing notation such that $\mathcal{P}_0 = 2$ and $\mathcal{P}_9 = 29$. Now given two sets A and B [1] we can apply this mapping to each symbol within our sets creating what I call a projection into prime number space.

$$A = \{a_1, a_2, ...a_N\} \longrightarrow \{\mathcal{P}_{a_1}, \mathcal{P}_{a_2}, ...\mathcal{P}_{a_N}\} = \mathcal{P}(A) \tag{5}$$

[1] We will omit the superscript because the length of A and B are implied to be the same, otherwise the sets trivially do not permute each other.

We can define our permutation function as'

$$P(A,B) = \prod_{j=1}^{N} \frac{\mathcal{P}(a_j)}{\mathcal{P}(b_j)} = \frac{\mathcal{P}(a_1) \times \mathcal{P}(a_2) \times \cdots \times \mathcal{P}(a_N)}{\mathcal{P}(b_1) \times \mathcal{P}(b_2) \times \cdots \times \mathcal{P}(b_N)} \tag{6}$$

Lets analyze this algorithm with two examples, the first example will demonstrate two sets that permute each other and the second will demonstrate two sets that do not.

Example 1

$$A = \{3,3,4,1\} \tag{7}$$

$$B = \{3,1,3,4\} \tag{8}$$

$$P(A,B) = \frac{\mathcal{P}(3) \times \mathcal{P}(3) \times \mathcal{P}(4) \times \mathcal{P}(1)}{\mathcal{P}(3) \times \mathcal{P}(1) \times \mathcal{P}(3) \times \mathcal{P}(4)} = \frac{7 \times 7 \times 11 \times 3}{7 \times 3 \times \times 7 \times 11} = \frac{7^2 \times 11 \times 3}{7^2 \times 11 \times 3} = 1 \tag{9}$$

Example 2

$$A = \{0,2,0,2\} \tag{10}$$

$$B = \{2,3,0,0\} \tag{11}$$

$$P(A,B) = \frac{\mathcal{P}(0) \times \mathcal{P}(2) \times \mathcal{P}(0) \times \mathcal{P}(2)}{\mathcal{P}(2) \times \mathcal{P}(3) \times \mathcal{P}(0) \times \mathcal{P}(0)} = \frac{2 \times 3 \times 2 \times 3}{3 \times 7 \times \times 2 \times 2} = \frac{2^2 \times 3^2}{2^2 \times 3 \times 7} = \frac{3}{7} \tag{12}$$

Implementation

Python

Complexity[2]: $\mathcal{O}(K * (2^K) * N) \ \alpha \ \mathcal{O}(N); \quad K = [16, 24]$

```python
def isPermutation(A, B):
  primes = [2.0, 3.0, 5.0, 7.0, 11.0, 13.0, 17.0, 19.0, 23.0, 29.0]
  if not (len(A) - len(B)):
    product = 1.0
    N = len(A)
    for j in range(N):
      product *= primes[int(A[j])]
      product /= primes[int(B[j])]
    return product
  else:
    # trivial case
    return 0.0
```

With this implementation it is possible to detect permutations from an arbitrary amount of sets through daisy chaining

```python
# P(A, B, C)
isPermutation(A,B)*isPermutation(B,C)

# P(A, B, ...Y, Z)
isPermutation(A, B)*...*isPermutation(Y, Z)
```

[2] K represents the bit width of a floating point number in python which is implementation specific. For further information refer to the architecture of your system and the IEEE 754 standard