1 Algorithm

Algorithm 1: Shortest Distance using Matrix Arithmetic

```
$distance\_matrix \leftarrow matrixofNxNwithvalues - 1$;
$D \leftarrow IdentityMatrix$;
for k from 1 to n do
    $D = D * A$;
    for i from 0 to n-1 do
        for j from 0 to n-1 do
            if D[i,j] != 0 and distance_matrix[i,j] != -1 then
                $distance\_matrix[i, j] = k$;
            end
        end
    end
end
```

2 Proof

- Let us start by proving that there is a walk of distance k if A^k is not 0 for any two vertices u and v. We will try to prove this via induction.
- For the initial case $k = 1$, the proof is trivial. If there exists an edge between the two vertices, then $A(u, v)$ is not 0, and hence there exists a walk of length 1. Now, let us assume that there exists a walk between two vertices u, v of length k if $A^k(u, v)$ is not 0. Now, to find a walk of length k + 1 between vertices (u, v) we have $sum(A^k(u, i) * A(i, v))$ which is equivalent to $A^{k+1}(u, v)$.
- Hence, we can see that the kth power of the adjacency matrix represents the length of the walk between two vertices u, v.
- To check for the validity of the shortest path, let us assume that the shortest path between the two vertices (u, v) is d. Now, there must be a walk of length d between u, v. Now if for k¡d we get a walk, then there is a way to get from u to v with a distance of k which contradicts our assumption. Hence, there must be a walk of distance k $\geq$ d. And since after the first time of assigning the value of the distance matrix, we do not reassign it, we can safely say that the algorithm gives us the shortest distance.

- It is quite clear that since there are n vertices, the maximum length of the walk cannot be greater than n-1. Hence, the algorithm will have an upper bound of n-1.

3 Time Complexity

The time complexity of the above algorithm is $O(n^4)$. Since the outer loop iterates n times and the complexity of matrix multiplication is n^3, we have the overall complexity of the algorithm $O(n^4)$.

4 Parallel Algorithm

Since the next iteration of outer loop depends on the value of D from the inner loop, parallelization of outer loop would not be possible. Instead, what we do is parallelize the inner loop - the matrix multiplication part and the value assignment.

Algorithm 2: Parallel algorithm for matrix multiplication f(A, B)

```
// A and B are the input parameters.
// Both loop can be parallelized. We can use OpenMP library to
 parallelize. $C \leftarrow NxNmatrixof0s$. for i from 0 to n-1 do
    for j from 0 to n-1 do
        #pragma omp parallel for schedule(static) for k from 0 to
         n-1 do
            C[i, j] += A[i, k] * B[k, j];
        end
    end
end
Return: C;
```

We can easily see that if the number of threads available are p, then we can achieve a speedup of p (excluding the scheduling overhead). We have used static because here each thread would have equal load so allocating the work load initially would prove to be much more beneficial.

Similarily we can parallelize the distance allocation. Here, we parallelize the outer loop.

Algorithm 3: Distance matrix allocation (Parallel)

```
// A and B are the input parameters.
// Both loop can be parallelized. We can use OpenMP library to
  parallelize. $C \leftarrow NxNmatrixof0s$. #pragma omp parallel for
  schedule(static) for i from 0 to n-1 do
    for j from 0 to n-1 do
        if D[i,j] != 0 and distance_matrix[i, j] != -1 then
            distance_matrix[i, j] = D[i, j];
        end
    end
end
```

We can see that the entire inner loop is parallelizable with a theoretical speedup of p. Hence, we have the new theoretical complexity to be $O(n^4/p)$ where p is the number of threads.