IT301: Parallel Computing Parallelization of Floyd-Warshall Algorithm

Aryaman Surya
Department of Information Technology
National Institute of Technology Karnataka, Mangalore, India
aryamansurya.211it011@nitk.edu.in

Verma Ayush
Department of Information Technology
National Institute of Technology Karnataka, Mangalore, India
vermaayush.211it079@nitk.edu.in

Vaidaant Thakur
Department of Information Technology
National Institute of Technology Karnataka, Mangalore, India
vaidaantthakur.211it075@nitk.edu.in

Vivek Nair
Department of Information Technology
National Institute of Technology Karnataka, Mangalore, India
viveknair.211it084@nitk.edu.in

***Abstract*—Before traveling, it is of utmost priority to determine not just any path to the destination, but to find the optimal or shortest path that can be taken from the source to the destination. The Floyd Warshall Algorithm is used to solve the All Pairs Shortest Path problem in a graph. This paper aims at improving the efficiency of the sequential implementation of the algorithm by parallelizing the same. Based upon the results obtained from 3 different programming environments (namely OpenMP, MPI and Cuda) on varying the size of the graph and the number of threads involved, a solid comparison on the performance inn terms of speedup and run time were made.**

***Index Terms*—all-pairs, shortest path, Floyd-Warshall.**

I. Introduction

Shortest Path algorithms are one of the most deeply researched and used algorithms in Graph Theory. A graph in programming is an object with vertices, that are connected to each other via edges. Each of these edges may have some cost associated with it. The internet or networks in general and digital maps are some examples of entities that can be visualized as graphs. Finding the shortest paths between cities in a map, routers in a network, or in a programming sense, between vertices in a graph, therefore becomes necessary for optimal performance. The shortest path from one vertex in the graph to another is the one that minimises the sum of cost of all edges along the path. Several algorithms have been designed to find the shortest paths in graphs, such as the Dijkstra's algorithm, Bellman-Ford algorithm and the Floyd-Warshall algorithm to name a few. Each of these work differently and are used in different scenarios. Out of these, the Floyd-Warshall algorithm happens to be the most expensive algorithm with respect to time taken for execution that makes it an interesting subject for research on the effect of parallelization. The Floyd-Warshall algorithm is a dynamic programming approach at finding the shortest path between all pairs in a weighted graph without negative cycles. It was published in 1962 by Robert Floyd. However, it was the same as the algorithms previously published by Stephen Warshall for finding the transitive closure of graphs. Hence, the algorithm came to be known as the Floyd-Warshall algorithm. The algorithm is immensely used in networks for finding optimal routes between routers. The algorithm works regardless of whether the graph is directed or undirected but fails when the graph has a negative cycle. Thus, this algorithm is especially useful in finding the optimal distance between a source and a destination. Other usages of the algorithm include inverting real matrices, checking if a graph is bipartite or not, finding similarities between graphs, detecting negative cycles in a graph and more. The algorithm consists of three loops, each of constant complexity. Thus, the Floyd Warshall algorithm has a time complexity of O(n3), while the space complexity is O(n2). This turns out to be inefficient considering both performance and power consumed. This inefficiency can be dealt with by paralleling the algorithm using various mechanisms. The core idea of this paper is to parallelize the Floyd- Warshall Algorithm in three different environments using different mechanisms:

1)Shared-Memory Multithreading using OpenMP

2)Distributed Computing using MPI, and

3)Distributing amongst Graphical Processing Units (GPUs) Using CUDA

Further, the paper aims to compare the performance measures obtained from all three mechanisms along with those obtained from the naive algorithm implementation to find the most efficient execution of the algorithm.

II. Literature Survey

The current body of research predominantly focuses on exploring applications of the Floyd-Warshall algorithm in diverse fields such as networks and maps. Investigations have been conducted to assess the algorithm's efficacy by comparing its performance with other similar shortest-path algorithms. Notably, there has been a particular emphasis on the parallelization of the Floyd-Warshall algorithm using Threading Building Blocks (TBBs). However, there exists a gap in research regarding the parallelization of the algorithm through OpenMP, MPI, and CUDA.

Given the algorithm's inherent computational expense, as evidenced by its worst-case time complexity of($O(n^3)$), it becomes imperative to adopt strategies for enhancing its runtime efficiency. One viable approach is parallelization, which has shown promise in optimizing the algorithm's performance. Consequently, this paper seeks to contribute to the existing body of knowledge by conducting a comparative analysis of various parallel implementations of the Floyd-Warshall algorithm using OpenMP, MPI, and CUDA. Through this comparative study, insights can be gained into the strengths and weaknesses of each parallelization method, thereby enabling informed decisions on selecting the most suitable implementation for specific scenarios involving the application of shortest-path algorithms to graphs with a substantial number of nodes.

III. Methodology

A. A. Sequential Implementation:

The Floyd-Warshall algorithm serves the purpose of finding the shortest paths between all pairs of vertices in a weighted graph, whether directed or undirected. Notably, it is important to acknowledge that the algorithm is not effective in the presence of a negative cycle within the graph.

In this implementation, the weighted graph is represented by an adjacency matrix, denoted as $adj_mat[][]$, which is an nxn matrix, with n being the number of vertices. The indices i and j in the matrix correspond to the vertices of the graph, and the value at position $adj_mat[i][j]$ represents the weight of the edge from the ith vertex to the jth vertex. The diagonal values in this matrix are set to 0.

Following the initialization of the adjacency matrix with random values using $rand()10000$, the floydWarshall function is invoked. Within this function, the result is stored in another nxn matrix named $distances[][]$. The initialization of this matrix involves using the values from the adjacency matrix.

The core logic of the algorithm revolves around modifying the 'distances[i][j]' with the shortest distance between vertex i and vertex j. This is accomplished through three loops where k, i, and j are the loop variables. The outermost loop variable, k, denotes the intermediate vertex, while the inner loop variables, i and j, denote the vertices between which the shortest path must be determined. The modification of values in 'distances[i][j]' is based on the condition:

$D_k[i,j] = min(D_k - 1[i,j], D_k - 1[i,k] + D_k - 1[k,j])$

Here, D_k represents the distances matrix after the kth iteration. After n iterations, D_n provides the length of the shortest path between i and j. It is important to note that the algorithm is unsuccessful if the diagonals of the distances matrix hold negative values, indicating the presence of a negative-weighted cycle in the graph.

Considering that the algorithm employs three nested loops in a dynamic programming approach to compute the distances matrix, the time complexity of the Floyd-Warshall algorithm is $O(n^3)$, which is notably inefficient. Consequently, there is a necessity to introduce parallel techniques to enhance performance.

Algorithm 1: The Floyd-Warshall algorithm Sequential

```
for (u, v) ∈ Edges do
    D_{u,v} ← weight(u, v)
end

for v = 1 → n do
    D_{v,v} ← 0
end

for k = 1 → 4n do
    for i = 1 → n do
        for j = 1 → n do
            if D_{i,j} > D_{i,k} + D_{k,j} then
                D_{i,j} ← D_{i,k} + D_{k,j}
            end
        end
    end
end
```

Additionally, as the algorithm utilises a two-dimensional matrix both to represent the graph and store the result, the space complexity of the Floyd-Warshall algorithm is $O(n^2)$.

B. B. Parallel Techniques - OpenMP Overview :

OpenMP, which stands for Open Specification for Multi-Processing, serves as an application program interface designed to support multi-threaded, shared memory parallelism. This versatile interface can be effectively utilised in languages such as C, C++, and Fortran. It encompasses key components like Compiler Directives, Runtime library routines, and Environmental Variables.

To parallelize the sequential algorithm discussed in the preceding section, the code leverages the $omp_parallel$ directive, instructing the compiler to parallelize the specified block of code. In this context, the $omp_parallel$ directive is employed to parallelize the three loops crucial to the algorithm. This directive is complemented by the use of clauses such as $num_threads$, 'shared', and 'private'. The $num_threads$ clause specifies the number of threads to be utilised in the parallel region, the 'shared' clause designates variables to be shared among the threads, and the 'private' clause designates variables as private to each thread.

In the code, the variable 'k', which represents the loop variable of the outermost loop, is declared using the 'private' clause, while the adj_mat and 'distances' matrices are declared using the 'shared' clause. Additionally, the 'omp for' directive is employed for the inner two loops to distribute loop iterations within the team of threads. The 'private' clause is used in conjunction with the 'omp for' directive to declare the inner looping variables 'i' and 'j'.

The scheduling clause, along with the 'omp for' directive, is employed to specify how different iterations of the 'for' loop are distributed among the available threads. The code explores four types of scheduling—static, dynamic, guided,

Algorithm 2: The Floyd-Warshall algorithm in OpenMP

```
for (u, v) ∈ Edges do
    D_{u,v} ← weight(u, v)
end

for v = 1 → n do
    D_{v,v} ← 0
end

#pragma omp num_threads(8) private(k) shared(adj_mat,D)
for k = 1 → n do
    #pragma omp for private(i,j) schedule(dynamic,20)
    for i = 1 → n do
        for j = 1 → n do
            if D_{i,j} > D_{i,k} + D_{k,j} then
                D_{i,j} ← D_{i,k} + D_{k,j}
            end
        end
    end
end
```

and runtime—while varying the number of nodes and threads. Execution times for each scheduling type are recorded, and a graph is generated to compare their performances under different configurations.

The first scheduling type is dynamic scheduling with a specified chunk size of 20. In this approach, iterations are distributed among the threads as they request them, with each thread receiving a chunk of iterations upon request. Static scheduling, akin to dynamic scheduling, provides chunks of iterations to threads in a round-robin fashion based on thread number.

Guided scheduling, similar to dynamic scheduling, assigns iterations to threads upon request, but with varying chunk sizes. The chunk assigned to each thread is determined by the ratio of the remaining number of iterations to the total number of threads, and the specified chunk size establishes the minimum size of the chunk of iterations.

Finally, run-time scheduling is determined dynamically during run-time using the OMP_SCHEDULE environmental variable.

C. C. Parallel Techniques - MPI Overview :

MPI, or Message Passing Interface, serves as a standard for message passing libraries, enabling communication between processes. It facilitates the development of message-passing programs in languages such as C, C++, and Fortran, particularly in the context of parallel computing. Adopting the Single Program, Multiple Data (SPMD) technique, this implementation utilises OpenMPI, an open-source implementation of the MPI standard.

In the MPI-based parallelization, three fundamental routines from Collective Communication are employed. One of these routines, MPI_Scatter, is utilised to distribute chunks of an array to all processes in a communicator. To facilitate the scattering of the adjacency matrix among all processes, the matrix is initialised as a one-dimensional array of size n^2 (where n is the number of vertices). Each process receives a portion of the adjacency matrix with a size of n^2/p, where p represents the number of processes.

Within the code, the outer loop, with k as the looping variable, iterates until k reaches the number of vertices n. Since the adjacency matrix has been scattered, each process possesses the adjacency matrix of n/p vertices. However, the k value is crucial for identifying an intermediate vertex, and thus, each process needs access to the entire adjacency matrix containing all vertices. Therefore, the values of the adjacency matrix corresponding to the k-th vertex are stored in a separate array by the process containing the block with the k-th vertex. Subsequently, this array is broadcasted to all other processes using MPI_Bcast, a collective communication routine that broadcasts a message from one process to all others in the group. This broadcasting occurs each time the value of k changes, resulting in a total of k MPI_Bcast operations. These operations serve as synchronisation points among the processes.

The remaining two loops within the outer loop operate similarly to the original Floyd Warshall Algorithm. However, as each process has its own chunk of the adjacency matrix, the second loop runs n/p times, while the third loop runs n times.

Since each process writes updated values of the shortest distances in its local matrix, all these local matrices are gathered to the root process into one resultant matrix at the end of the $k-th$ for-loop. This aggregation is achieved using MPI_Gather, which essentially reverses the operation performed by MPI_Scatter. MPI_Gather collects chunks of data (an array) from all processes into a single process.

D. D. Parallel Techniques - CUDA Overview:

CUDA, an abbreviation for Compute Unified Device Architecture, serves as an API that empowers software programs to harness the processing capabilities of GPUs for general computing tasks. The distinguishing feature of a GPU, compared to a CPU, lies in its multitude of cores—often numbering in the hundreds. CUDA leverages these cores to concurrently execute multiple processes within the same program, adhering to the Single Program Multiple Data (SPMD) processing model.

In the CUDA paradigm, blocks can be defined in one, two, or three dimensions, and threads within blocks can also be specified in up to three dimensions. However, a block is limited to a maximum of 1024 threads, and a grid supports up to 65536 blocks in each dimension. This paper proposes an algorithm that utilises a 2D grid of blocks, with each block containing threads in two dimensions.

Due to the constraint on the number of threads per block, the number of threads is predefined in each program, and the workload is distributed across blocks. This implementation defines an n/t x n/t grid of blocks, with each block comprising

Algorithm 3: The Floyd-Warshall algorithm in MPI

```
for (u, v) ∈ Edges do
    D_{(u*n)+v} ← weight(u, v)
end

for v = 1 → n do
    D_{(v*n)+v} ← 0
end

div ← (n * n)/p

MPI_Scatter(D, div, MPI_INT, M, div, MPI_INT, 0,
  MPI_COMM_WORLD)

floydWarshall(M, rank, p)

MPI_Gather(M, div, MPI_INT, Ans, div, MPI_INT, 0,
  MPI_COMM_WORLD)

Function floydWarshall(M, r, p):
    d ← n/p
    for k = 1 → n do
        pos ← k%d
        rank ← k/d
        if r == rank then
            for x = 1 → n do
                kthRow_x ← M_{(pos*n)+x}
            end
        end
        MPI_Bcast(kthRow, n, MPI_INT, rank,
          MPI_COMM_WORLD)
        for i = 1 → d do
            for j = 1 → n do
                dist ← M_{(i*n)+k} + kthRow_j
                if M_{(i*n)+j} > dist then
                    M_{(i*n)+j} ← dist
                end
            end
        end
    end
    return
```

Algorithm 4: The Floyd-Warshall algorithm in CUDA

```
for (u, v) ∈ Edges do
    D_{(u*n)+v} ← weight(u, v)
end

for v = 1 → n do
    D_{(v*n)+v} ← 0
end

block ← n/t × n/t

threads ← t × t

for k = 1 → n do
    floydWarshallGPU<<< block, threads >>>(D)
end

Function floydWarshallGPU(D):
    i ← (blockIdx.x × blockDim.x) + threadIdx.x
    j ← (blockIdx.y × blockDim.y) + threadIdx.y
    syncthreads()
    if D_{i,j} > D_{i,k} + D_{k,j} then
        D_{i,j} ← D_{i,k} + D_{k,j}
    end
    syncthreads()
    return
```

t x t threads. Similar to previous parallelization techniques, CUDA is employed to parallelize the two inner loops. The kernel function, 'floydWarshallGPU', is executed n times, and the inner variables i and j are calculated as follows:

$i = blockIdx.x * blockDim.x + threadIdx.x$

$j = blockIdx.y * blockDim.y + threadIdx.y$

This ensures that i and j vary from 0 to n as required.

CUDA mandates the copying of arrays between CPU and GPU memory. To facilitate this, one-dimensional arrays of size n^2 are defined, and the element $arr[i][j]$ is accessed as $arr[i*n+j]$. Initially, the distance array is initialised in the CPU with values from the adjacency matrix. After allocating space in the GPU for computation, values are transferred from the CPU to the GPU. Subsequently, the kernel function is executed. Within this function, after computing the values of i and j as mentioned above, threads are synchronised before and after applying the core part of the algorithm:

$D_k[i,j] = min(D_k - 1[i,j], D_k - 1[i,k] + D_k - 1[k,j])$

Thread synchronisation is achieved using the 'syncthreads' function, ensuring that all threads in the block possess the same version of the array. After GPU computation, values are copied back to the CPU memory following a 'cudaDeviceSynchronize' directive, which guarantees synchronisation across all blocks within the GPU, indicating the completion of their execution.

IV. Result

The sequential Floyd-Warshall algorithm was compared with the parallelized versions implemented using OpenMP, MPI, and CUDA. A comprehensive analysis was performed by varying the data size and number of threads for all implementations.

A. OpenMP

The sequential algorithm was analyzed by varying the number of nodes to see how it performs compared to the parallel versions. As expected, it took longer to execute than the parallel versions of the code. The results have been plotted along with the OpenMP parallelized version of the algorithm tabulated in Table I and Table II. Further, the OpenMP algo-

TABLE I
EXECUTION TIMES FOR THE SEQUENTIAL ALGORITHM

	No. of Nodes							
Algorithm	500	750	1000	1250	2500	3750	5000	7500
Sequential	0.422094	1.384741	3.222286	6.381931	49.762170	167.688373	390.662876	1307.492269

rithm was analyzed extensively by varying various parameters like:

1) Number of nodes
2) Scheduling type
3) Number of threads

It was found that the parallel versions execute much faster than the sequential when using 8 threads. While static scheduling takes longer than its parallel counter-parts when the number of nodes is large, it's comparable to the other scheduling types for smaller graphs. Changing scheduling types have little effect on the time taken to run the algorithm. It is a similar situation on varying the number of threads. The code was executed with dynamic scheduling, and the number of threads was varied to analyze the effect. When only one thread is used, it takes a long time to execute. It was found that the effect of the number of threads on the runtime is negligible when the number of threads were varied between 2 and 16. The results have also been tabulated in Table III

B. MPI

The MPI implementation was executed on a system with 4 cores, and an Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz processor. It was then analyzed by varying the number of processes and the number of nodes within the graph. As expected, the runtime increases with the number of nodes. When compared with the number of processes, it can be seen that the performance is best when the number of processes is 4. In general, however, performance isn't affected much when the number of processes is varied between 2 and 8 for nodes less than 3750. The difference is much more significant for graphs with 5000 or more nodes. Another point to be noted, by comparing values in Table I and values for 1 process from Table IV, is that the MPI algorithm performs worse than the sequential algorithm, with significant differences for graphs with more than 5000 nodes, negligible otherwise. This can be attributed to the additional processing done for the distribution of data across processes. A deeper analysis on the effect of varying the number of processes, showed that the performance worsens when the number of processes increases from 4 to 5. It again gets better as the number of processes is reduced to 8, which is comparable with the performance of 4 processes. The variation is significant when the number of nodes in the graph is large. This happens because the program is run on a computer with 4 cores. With 4 processes equally distributed across the 4 cores, all cores perform efficiently and give the best performance. When the number of processes increases to 5, after distribution across the 4 cores, the 5th process runs on one of the cores, reducing its efficiency and affecting overall performance. With more processes, however, the workload is more evenly distributed. Even though the number of processes is greater than the number of cores, the performance isn't affected as much, and it gets better. For 8 processes, the distribution is similar to that of 4, so performance is comparable. A detailed set of observations is given in Table IV. Overall, when the best performances of MPI and OpenMP are considered, MPI performs better than OpenMP.

C. CUDA

The CUDA code was executed on a Google Colab environment with CUDA-9.2. Due to the large number of nodes, the number of threads per block was controlled, and number of blocks varied depending upon the size of the graph. The performance was analyzed by varying the size of the graph and the number of threads per block. As expected, runtime increased with the number of nodes. While 1 thread per block performed similar to the sequential algorithm on the Google Colab environment, it improved with more threads per block. It was found that 64 threads per block performed best, much better than 4 threads or lesser per block. As the number of threads per block in the grid increases, the runtime also steadily, although negligibly, increases. This is due to the synchronization of threads performed before and after the main computation of distances. With more number of threads, each thread waits for all threads to be in sync before proceeding, which causes a bigger delay for a larger number of threads. However, CUDA performs much better than the other algorithms overall. While the sequential algorithm takes ap- proximately 20 minutes to execute for 7500 nodes, CUDA only takes about 1 minute in the best case. This is a massive improvement, even compared to MPI and OpenMP, which take 8 and 11 minutes respectively, for the same graph size. A detailed set of observations is given in Table V.

V. Conclusion

The sequential implementation has been parallelized in three different programming environments: OpenMP, MPI and CUDA. The results have been analyzed considering the varying number of vertices in every environment, along with varying number of threads in OpenMP, varying number of processes in MPI and varying number of threads per block in CUDA. The analysis of sequential versus parallel implementations revealed all of the parallel implementations to have much lesser execution times for varying number of vertices. However, when OpenMP is run with one thread, MPI for one process and CUDA for one thread per block, the time taken is similar to that of sequential execution as the parallelization aspect of these three environments is not being used in these scenarios. From the experimental observations, it is clear that OpenMP performed best when dynamic scheduling was

TABLE II
EXECUTION TIMES FOR THE OPENMP ALGORITHM WITH 8 THREADS DIFFERENT SCHEDULES

	No. of Nodes							
Schedule	500	750	1000	1250	2500	3750	5000	7500
Static	0.256139	0.763251	1.753842	3.306170	25.882340	87.366389	203.911654	1279.759253
Runtime	0.288553	0.907263	2.059180	3.945450	30.440848	101.550698	238.506913	797.356790
Guided	0.233105	0.748041	1.700897	3.283199	25.413071	86.269530	199.419663	665.432118
Dynamic	0.237393	0.740559	1.682285	3.231955	25.085005	84.068729	198.316902	664.064237

TABLE III
EXECUTION TIMES FOR THE OPENMP ALGORITHM WITH DIFFERENT NUMBER OF THREADS

	No. of Threads					
No. of Nodes	1	2	4	8	16	32
500	0.485256	0.254166	0.278963	0.285666	0.276869	0.300628
1250	7.038478	3.748890	3.709332	3.727504	3.777829	3.801500
2500	56.020662	28.844804	29.145761	28.788539	29.296373	29.588597

TABLE IV
EXECUTION TIMES FOR THE MPI ALGORITHM WITH DIFFERENT NUMBER OF PROCESSES

	No. of Nodes							
No. of Processes	500	750	1000	1250	2500	3750	5000	7500
1	0.431461	1.375540	3.182001	6.188979	50.150933	166.003417	412.136310	1325.932049
2	0.212534	0.736809	1.602658	3.171841	24.969531	92.337896	216.585473	738.390621
3	0.163430	0.584663	1.377823	2.574914	19.922221	85.763126	148.464114	488.099018
4	0.560581	0.833428	2.044864	3.726466	28.022778	97.398225	229.157220	769.443911
5	0.225698	0.619068	1.574791	2.950205	27.479623	83.829576	199.947368	648.187426
6	0.174054	0.668151	1.346150	3.154860	25.506249	83.671826	185.290698	620.251866
7	0.097505	0.345942	0.825535	2.130025	19.046290	68.002339	148.380683	499.123987

TABLE V
EXECUTION TIMES FOR THE CUDA ALGORITHM WITH DIFFERENT NUMBER OF THREADS PER BLOCK

	No. of Nodes							
No. of Processes	500	750	1000	1250	2500	3750	5000	7500
1	0.431461	1.375540	3.182001	6.188979	50.150933	166.003417	412.136310	1325.932049
2	0.212534	0.736809	1.602658	3.171841	24.969531	92.337896	216.585473	738.390621
3	0.163430	0.584663	1.377823	2.574914	19.922221	85.763126	148.464114	488.099018
4	0.560581	0.833428	2.044864	3.726466	28.022778	97.398225	229.157220	769.443911
5	0.225698	0.619068	1.574791	2.950205	27.479623	83.829576	199.947368	648.187426
6	0.174054	0.668151	1.346150	3.154860	25.506249	83.671826	185.290698	620.251866
7	0.097505	0.345942	0.825535	2.130025	19.046290	68.002339	148.380683	499.123987

used, MPI when the load was equally distributed among the machine's cores (number of processes was a multiple of the number of cores) and CUDA when the number of threads per block were 8 x 8, with 9 x 9 and 10 x 10 also having comparable performance times. Overall, CUDA performed much better than the other two environments, followed by MPI and finally OpenMP. OpenMP is relatively easier to implement and thus is better suited for most general purposes. In situations involving memory intensive calculations, OpenMP struggles with efficiency. In suc situations, the distributed computation with MPI works well. CUDA is best suited in cases of GPU computing and in using Nvidia GPUs to parallelize tasks. In the future, implementation and analysis of hybrid implementations of these parallel environments can be done to see the performance efficacy as compared to the traditional implementations.

REFERENCES

[1] Jian Ma, Ke-ping Li, Li-yan Zhang., A Parallel Floyd-Warshall algorithm based on TBB [2]

[2] Azis, H., Mallongi, R. dg., Lantara, D., Salim, Comparison of Floyd-Warshall Algorithm and Greedy Algorithm in Determining the Shortest Route[1]

[3] Magzhan Kairanbay, Hajar Mat Jani, A Review and Evaluations of Shortest Path Algorithms [3]

[4] Dehal, R. S., Munjal, C., Ansari, A. A., Kushwaha, A. S., GPU Computing Revolution: CUDA [4]

[5] Novandi, Raden AD. 2017 Perbandingan Algoritma Dijkstra dan Algoritma Floyd-Warshall dalam Penentuan Lintasan Terpendek (Single Pair Shortest Path). Makalah IF2251 Strategi Algoritmik, 2, pp. 1-5