Bellman-Ford Algorithm
Hybrid implementation using OpenMP and MPI

Pavan Gami
DA-IICT
Gandhinagar, India
201901409@daiict.ac.in

Deep Tank
DA-IICT
Gandhinagar, India
201901410@daiict.ac.in

Yash Mandaviya
DA-IICT
Gandhinagar, India
201901438@daiict.ac.in

Rahi Krishna
DA-IICT
Gandhinagar, India
201901446@daiict.ac.in

Swetanshu Patel
DA-IICT
Gandhinagar, India
201901448@daiict.ac.in

Yash Raja
DA-IICT
Gandhinagar, India
201901461@daiict.ac.in

***Abstract*—This research paper highlights the hybrid implementation of the Bellman-Ford algorithm for solving the single source shortest path problem (SSSP) for an undirected graph. Parallel threading of the algorithm is analyzed as various methods are employed to generate considerable speed up.**

***Index Terms*—Shortest Path Problem, Bellman-Ford Algorithm, Parallelization**

I. Introduction

The single source shortest path problem is used in a wide range of scientific and real-world applications. These algorithms are commonly used in network routing, VLSI design, robotics, and transportation, and they are also utilised in Google Maps for directions between physical sites. Real-time data sharing is based almost entirely on pathfinding algorithms such as Bellman-Ford, Dijkstra and A* to find the quickest route through which information packets can be transferred. All of the applications discussed so far have positive weights, although there are notable exceptions, such as currency exchange arbitrage and other areas where the edge represents something other than the distance between two entities.Main focus of this paper will be bellman-ford algorithm and we have presented 3 variations of the bellman ford algorithm and have also tried to parallelize them . The results are also shown in graphical view for batter understandings

1.1 Bellman-Ford Algorithm

The Bellman-Ford algorithm is one of the single source shortest path algorithms (SSSP) that computes shortest paths from a single source vertex to all of the other vertices in a weighted digraph. The algorithm is a combined contribution of the American applied mathematicians Richard E. Bellman and Lester Randolph Ford.

The Bellman-Ford algorithm uses the concept of relaxing (or relabeling) the edges of a graph, in which approximations to the correct distance are replaced by better ones until they eventually reach the solution. The approximate distance to each vertex is always an overestimate of the true distance, and is replaced by the minimum of its old value and the length of a newly found path. The distance from the source vertex (current location) to every other node is calculated iteratively, and relaxed until the optimal route to the end vertex (destination) is found.

This algorithm works for every weighted digraph (directed graph) which does not contain negatively weighted cycles, and can also be applied for all undirected graphs that don't contain negatively weighted edges (Since the edges are bidirectional, and thus behave as negative weighted cycles).

1.2 Historical Improvements of bellman-ford algorithm

Bellman ford algorithm was introduced by sir bellman in his paper " On a routing problem" in 1958.[12]. The algorithm was introduced as a sssp algorithms which was capable of detecting negative cycles. Lester ford Jr. published his paper "Network flow theory" in 1956, which also described the same algorithm. [13].

In 1957, major improvement on bellman-ford algorithm was presented by Edward F. Moore. He presented his paper at at 1957 conference on theory of switching and showed his variation of bellman ford algorithm, and in 1959 his book "The shortest path through a maze" [16].

In 1993 Andrew V. Goldberg and Tomasz Radzik showed another variation of the algorithm using label correction method and topological scanning. They also stats that in most cases this algorithm will outperform deque and two que algorithms, which are another variations of bellman ford by D'Escopo-Pape and Pallottino, but its asymptotic complexity

will remain as O(edges*vertices). [7]

1.3 Juxtaposition

The first and simplest of all path finding algorithms is Dijkstra's algorithm, widely known in graph theory. Even though both Dijkstra's and Bellman-Ford algorithms use the relaxation procedure, Dijkstra's algorithm greedily chooses the closest unprocessed vertex, and relaxes the outgoing edges. In contrast, the Bellman-Ford algorithm relaxes all the edges $|\mathbf{V}| - \mathbf{1}$ times.

This simple change in the algorithm allows the Bellman-Ford algorithm to be applied to a wider range of inputs, which include negative edge-weights.

For a directed weighted graph having $\mathbf{V}$ vertices and $\mathbf{E}$ edges, Dijkstra's algorithm has a time complexity of $\mathbf{O(E + V \cdot logV)}$, while Bellman-Ford's algorithm takes $\mathbf{O(V \cdot E)}$ time. Both algorithms have the same space complexity $\mathbf{O(E + V)}$.

Each of these algorithms are used situationally, as Dijkstra's algorithm performs better than Bellman-Ford's where only positive-weight graphs are involved, but fails otherwise.

Dijkstra's algorithm provides a work efficient implementation, whereas Bellman-Ford provides scope for easy parallel implementation. Delta Stepping algorithm introduces a trade-off between the two.[17]

II. Related Work

In this paper we have worked on this two variation , one based on observation and second based on moore's Improvement.

The Bellman–Ford algorithm may be improved in practice by the observation that, if an iteration of the main loop of the algorithm terminates without making any changes, the algorithm can be immediately terminated, as subsequent iterations will not make any more changes. With this early termination condition, the main loop may in some cases use many fewer than $|V| - 1$ iterations, even though the worst case of the algorithm remains unchanged. The following improvements all maintain the $O(|V|.|E|)$ worst-case time complexity.

A variation of the Bellman-Ford algorithm known as Shortest Path Faster Algorithm, first described by Moore [16], reduces the number of relaxation steps that need to be performed within each iteration of the algorithm. If a vertex v has a distance value that has not changed since the last time the edges out of v were relaxed, then there is no need to relax the edges out of v a second time. In this way, as the number of vertices with correct distance values grows, the number whose outgoing edges that need to be relaxed in each iteration shrinks, leading to a constant-factor savings in time for dense graphs.

III. Serial Algorithm

In this section, we develop a simple serial code for the Bellman-Ford algorithm. In this approach, each edge will be relaxed —V-1— times . Initial distance of all the vertices will be infinite and source's distance will be zero .At any point of time distance of the vertices will be the upper bound of the original distance from source. Here for batter performance we have taken data structure as list of edges or edge list, cause we have to access each edge in o(1) time .

An elementary pseudo-code for the algorithm is as follows:

3.1 Simple Approach

Algorithm 1 Simple Version (Not compatible with negative cycle)

distance array $dis[v]$
$dis[source] \leftarrow 0$
for range (0 to $v - 1$) **do**
  for all edges in graph **do**
    $relax$ edge
  end for
end for

Explanation:
For all edges, if the distance to the destination can be shortened by taking the edge, the distance is updated to the new lower value. At each iteration i that the edges are scanned, the algorithm finds all shortest paths of at most length i edges .
Since the longest possible path without a cycle can be $|\mathbf{V}| - \mathbf{1}$ edges, the edges must be scanned $|\mathbf{V}| - \mathbf{1}$ times to ensure the shortest path has been found for all nodes.

Relaxation process:
Bellman ford algorithms is mainly based on edge relaxation. Edge relaxation is operation performed on a destination vertices to reduce the distance of the destination if possible . If by a given edge you can reach the destination earlier then before then distance of the destination vertices will update [18].
In pseudo code relaxation will look something like this:

Relax (source, destination)

If $dis[destination] > dis[source] + cost(source, destination)$
Then $dis[destination] = dis[source] + cost(source, destination)$

A simple flowchart that highlights the processes in the Bellman-Ford program is shown in Fig. [1]

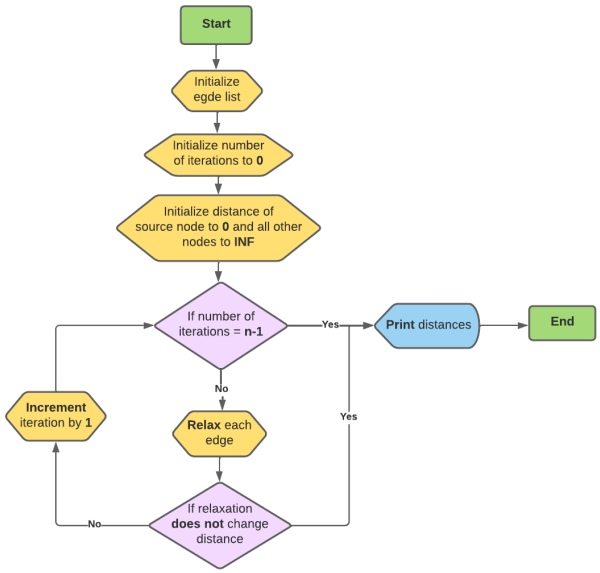

Fig. 1. Flowchart: Simple Bellman-Ford algorithm

3.2 Improved version of bellman-ford based on observation and case of negative cycle

This algorithm shows the optimized version of the original pseudo-code. To achieve this, we use a check variable to check if the cost values have changed from the previous iteration. If they haven't, then we can terminate program to avoid excess computations. Given below is the pseudo code for the optimized Bellman-Ford,

Algorithm 2 Improved version of bellman-ford

```
distance array dis[v]
dis[source] ← 0

for range 0 to v − 1 do
    flag ← false
    for all edges in the graph do
        relax edge and set flag true on relaxation
    end for
    if flag ← false then
        break
    end if
end for
```

Using bellman ford algorithm we can also detect negative cycle , For detection negative cycle we have to run a single iteration and if relaxation functions works for even a single time , there will be a negative cycle according to the algorithm.

But to get effective distances we have to run v-1 iterations again and reason for the same has given by an example.

Algorithm 3 For case negative cycle

```
for range 0 to v − 1 do
    flag ← false
    for all edges in graph do
        if dis[starting_node ← ∞] then
            dis[ending_node] ← ∞
            flag ← true
        else
            set flag to true on relaxation and
            dis[ending_node] ← ∞
        end if
    end for
    if flag ← false then
        break
    end if
end for
```

A flowchart of the optimized version and negative cycle case is shown below. This is built on the previous flowchart shown in Fig. [2]

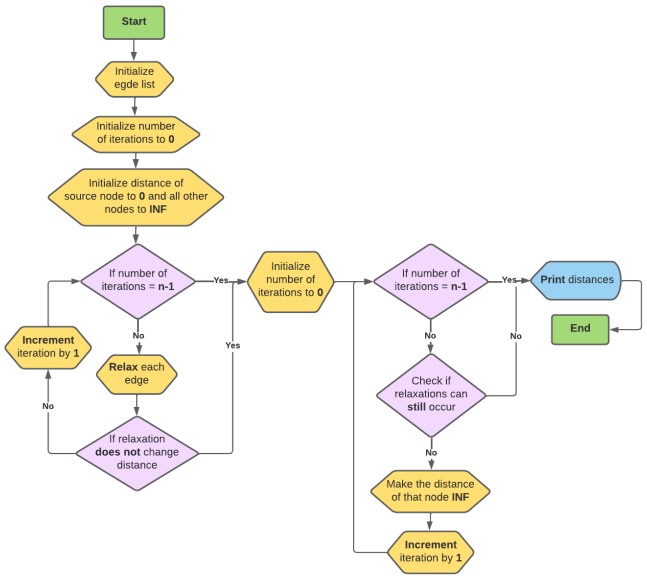

Fig. 2. Flowchart: Optimized Bellman-Ford algorithm

Explanation:

Let us take an example. Consider the graph in Fig. [3].
Let us assume that the source node is Node 1, which makes it's initial cost 0. The cost of all the other nodes would thus be initialized to INF.

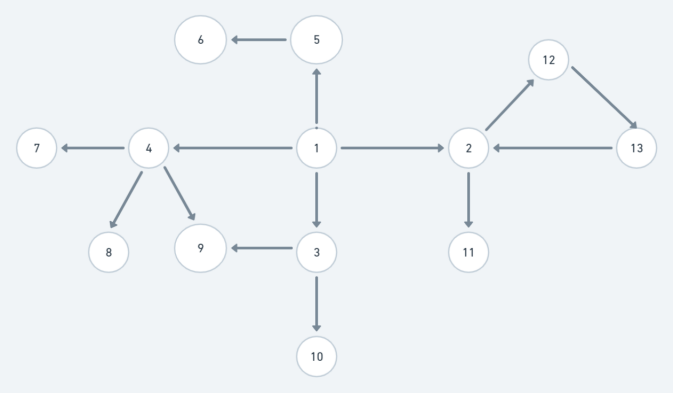

Fig. 3. Example graph with a few cycles.

As per the algorithm, we must perform relaxation on each edge $\mathbf{V-1}$ times where $\mathbf{V}$ is the number of vertices in the graph (13 in this case).

For the first relaxation iteration, only the edges connected directly to the source node will be relaxed. These values will be updated to the new costs which will always be the minimum cost because there is no other path from Node 1 to the adjacent nodes. This cost will not be reduced further unless there is a negative edge-weight in the graph.

In the 1^{st} iteration, nodes [5, 4, 3, 2] will be relaxed. In the 2^{nd} iteration, edges directly connected to the nodes iterated through in the 1^{st} iteration $\rightarrow$ [6, 7, 8, 11, 12, 9, 10] will be relaxed. For example, the final cost of Node 9 will be either from Node 4 or Node 3 depending on the order of edge relaxation. Similarly, final cost of Node 2 will be either from Node 1 or Node 13.

Let us take the case of a negative weighted cycle in the graph. Assume that the Node 2, 12, 13 cycle has a negative weight. In such cases, we proceed in a similar fashion. Node 2 will be relaxed in the 1^{st} iteration, Node 12 will be relaxed on the 2^{nd} iteration, and Node 13 in the 3^{rd} iteration. Now, here is the bit tricky. If we continue to the 4^{th} iteration, Node 2 will be relaxed yet again, as the edge is negatively weighted.

The problem with continued iteration is - If we continue this cycle, the cost of reaching Node 2 keeps decreasing towards –INF. To avoid this, we try to detect negative cycles in a graph beforehand. We know that for a graph with n edges, the entire graph will relax in n iterations. So, if the nodes keep relaxing after n iterations, the graph very obviously contains a -ve cycle.

From the negative cycle detection method, is is clear that a maximum of $(\mathbf{V-1})$ iterations are required for completely relaxing any graph, as the node furthest from the source node can be at most at a distance of $(\mathbf{V-1})$ edges.

IV. Time Complexity

From the understanding of the working of the Bellman-Ford algorithm and its pseudo-code, we can confidently say that the complexity of this algorithm depends on the number of edge comparisons for all cases.

Even though the time complexity of Bellman-Ford's algorithm exceeds that of other path finding algorithms, it is much simpler and even works with negative weighted graphs and distributed systems. The edge relaxation varies depending on the graph. In some cases the algorithm might need to run through all iterations, while in other cases the result could be acquired in the first few iterations.

1. Worst-case time complexity:
We encounter a worst-case scenario when the graph in focus contains a negative cycle. For these cases, the program has to go through $(\mathbf{V-1})$ iterations. It's time complexity will be:

$$\mathbf{T.C.} = \mathbf{O}(|\mathbf{V}| \cdot |\mathbf{E}|)$$

Another scenario is when we are given a complete graph with edges between every pair of vertices, considering a case where we have found the shortest path in the first few iterations but still proceed with relaxation of edges, we would have to relax $|\mathbf{E}| \cdot \frac{|\mathbf{E}|-\mathbf{1}}{\mathbf{2}}$ edges, $(|\mathbf{V}|-\mathbf{1})$ times. For this case, the T.C. turns out to be:

$$\mathbf{T.C.} = \mathbf{O}(|\mathbf{E}| \cdot |\mathbf{V}|) = \mathbf{O}(|\mathbf{V^2}|) \cdot \mathbf{O}(|\mathbf{V}|) = \mathbf{O}(\mathbf{V^3})$$

2. Average-case time complexity:
The worst-case running time can be improved by simply terminating the algorithm as soon as the iterations stop making any further changes to path costs.
In this case, relaxation still occurs $(|\mathbf{V}|-\mathbf{1})$ times, to exactly $\mathbf{E}$ edges. The T.C. in this case will also be quadratic:

$$\mathbf{T.C.} = \mathbf{O}(|\mathbf{V}| \cdot |\mathbf{E}|)$$

3. Best-case time complexity:
In any graph, if edge relaxation were to happen in order from left $\rightarrow$ right, then the algorithm would have to go through only one relaxation iteration to find the shortest path. This will make the T.C. directly proportional to the number of edges in the graph:

$$\mathbf{T.C.} = \mathbf{O}(|\mathbf{E}|)$$

The **Space Complexity** of the Bellman-Ford algorithm is simply $\mathbf{O}(\mathbf{V})$ (linear space), as we need to store all the nodes in the graph.

V. Shortest Path faster Algorithm

This variation of the Algorithm is Moore's improvement and also called all BFS improvement. The SPFA algorithm was first published by Edward F. Moore in 1959, as a generalization of breadth first search. SPFA is Moore's "Algorithm D." The name, "Shortest Path Faster Algorithm (SPFA)," was given by FanDing Duan, a Chinese researcher who rediscovered the algorithm in 1994.[19]

In this method we will maintain a queue of the vertices whose distance is updated. Then for each vertices in queue we will look into all the edges of that vertex all relax them all. To check whether there is a negative cycle we can keep an count for the number of times distance of the node updated , if it becomes more then —V-1— then there is a negative cycle. We assume, without loss of generality, that all nodes are reachable from s in G and that G has no multiple arcs. This assumption allows us to refer to an arc by its endpoints without ambiguity. In this method we have used adjacency list as a data structure cause we have to find all the edges connected to a node. Here proof of concept will remain same but we will only take the edges which have high chances to get relaxed , as its ancestor's distance is updated.

An elementary pseudocode for Shortest Path Faster Algorithm is given below:

Algorithm 4 Pseudo-Code for Shortest Path Faster Algorithm

distance array $dis[v]$
$dis[source] \leftarrow 0$

push $source$ into $Queue$
while $Queue$ is not empty **do**
  $u :=$ poll $Queue$
  for each edge from $u, (u, v)$ **do**
    if $dis(u) + weight(u, v) < dis(v)$ **then**
      $dis(v) = dis(u) + weight(u, v)$
      v into $Queue$
    end if
  end for
end while

VI. Parallelization

Following are the needs to parallelize the algorithm:

- The search for shortest paths in Bellman-Ford can be independently performed for all the nodes.
- The search for shortest paths beginning in a fixed node ***u*** can also be made in parallel.
- Despite Djikstra's smaller run time complexity, it is practically difficult to parallelize [4], while the Bellman-Ford algorithm has a considerable parallelization resource.

This means that if $\mathbf{O}(|\mathbf{E}|)$ processors are available for computation, then the parallel algorithm will terminate after at most $\mathbf{O}(|\mathbf{V}|)$ steps. Usually, a smaller number of steps are required.

Computer scientists have been successfully able to combine the Bellman-Ford algorithm with Dijkstra's algorithm, to accommodate both negative edge-weights and a smaller time complexity. This is called the Hybrid-Bellman-Ford-Dijkstra algorithm.

There have been many other improvements and variations of the Bellman-Ford algorithm in the past few decades. A few of them include Yen's improvement, Randomized Bellman-Ford and the Shortest Path Faster Algorithm (SPFA).

We parallelize our serial code and run it on the LAB-207 PC. Hardware details are as follows:

Type	**Lab207**
CPU Model	Intel(R) Core(TM) i5-4590 CPU Z
Architecture	x86_64
Byte Order	Little Endian
CPU(s)	4
Sockets	1
Cores per Socket	4
Clock (GHz)	3.30
L1 cache	64 KB
L2 cache	256 kB
L3 cache	6144 kB

VII. Parallel Algorithm

In this section, we try to analyse and develop parallel code for the Bellman-Ford algorithm. Here the parallization of inner loop is done using for closure. An elementary pseudo-code for the algorithm is as follows:

Algorithm 5 Pseudo-Code for Parallel Algorithm

for k in range (1 to $n-1$) **do**
  for all v in V such that (u, v) belongs to E in parallel **do**
    relax (u, v)
  end for
end for

Parallel time complexity:

In the implementation we have parallelize the inner loop, so time complexity of outer loop will remain same as serial code. And time complexity of inner loop will be p folds, where p is number of threads.

So we can write worst case complexity $O(|V-1| * |E|/p)$

The relaxation part of the program is the most time-consuming, and we can try to run different iterations on different threads to reduce the overall runtime.

Proof of parallelization There is inherent parallelism in standard Bellman Ford algorithm which lies in Relax() procedure [11].In some kth iteration we can write relaxation procedure as follows:

$$Dis^k[dest] = min(dis^{k-1}[dest], dis^{k-1}[src] + cost[src\_to\_dest]$$

Here, the value of distance (k^{th} value) depends on the k-1^{th} iteration, so we cannot parallelize the outer loop. We can still parallelize the inner loop upto two levels [12].

Level 1: Distance[v_1] does not depend on Distance[v_2], so we can parallelize it simply.
Level 2: For all u in set v, $Dist_{k-1}$[u] + cost[u, v] can be calculated in parallel as they are independent of each other.

When we increase the problem's data size, memory access times also increase, as an increase in vertices and edges will increase more each time we have to access new data cause every time we have to access new node and its weight every time we relax edge.

We run our parallel code on Lab-207 PC and get the following walltimes, speedup and efficiency for different data-sizes. Our dataset was generated online using a testcase generator.[15]

VIII. Results for Naive approach

V × E	Serial wall-time (ms)	Parallel wall-time (ms) (Thread-2)	Parallel wall-time (ms) (Thread-3)	Parallel wall-time (ms) (Thread-4)
2.5×10^2	0.03072	0.012023	0.03294	0.034156
2×10^3	0.076741	0.036387	0.032849	0.033087
6.15×10^4	1.09827	0.602186	0.509286	0.34205
1.3×10^5	2.17433	1.20718	0.864472	0.665018
3×10^5	9.74696	6.31964	1.35198	3.32789
1.65×10^6	63.3555	25.9878	10.0566	8.81199
1.32×10^7	208.786	105.173	73.7584	60.2976
5×10^7	790.977	398.132	272.886	211.234
1.7×10^8	2732.77	1370.29	948.27	734.015
2.25×10^8	3602.9	1807.46	1243.86	958.571
4×10^8	19204	9664.1	6616.64	5101.71
1.2×10^9	19213	9641.05	6614.54	5120.3

TABLE I
WallTime(sec) Analysis for Parallel algorithm

V × E	Speedup (Thread-2)	Speedup (Thread-3)	Speedup (Thread-4)
2.5×10^2	2.55360	0.9320582878	0.8988757466
2×10^3	2.109022453	2.336174617	2.319370145
6.15×10^4	1.823805269	2.156489674	3.210846367
1.3×10^5	1.801164698	2.515211597	3.210846367
3×10^5	1.54232836	7.209396589	2.928870846
1.65×10^6	2.437893935	6.299892608	7.18969268
1.32×10^7	1.985167296	2.830674201	3.462592209
5×10^7	1.986720485	2.898562037	3.744553434
1.7×10^8	1.994300477	2.88184799	3.723043807
2.25×10^8	1.993349784	2.896547843	3.758615689
4×10^8	1.987148312	2.902379455	3.758615689
1.2×10^9	1.992832731	2.904661549	3.7523192

TABLE II
Speedup Analysis for Parallel algorithm

V × E	Efficiency (Thread-2)	Efficiency (Thread-3)	Efficiency (Thread-4)
2.5×10^2	1.2768027	0.310686	0.224718
2×10^3	1.054511	0.7787248	0.5798425
6.1×10^4	0.911907	0.7188298	0.802711
1.3×10^5	0.900582	0.838403	0.81739
3×10^5	0.771164	2.40313	0.73221
1.7×10^6	1.21894	2.09996	1.7974
1.3×10^7	0.992583	0.943558	0.86564
5×10^7	0.99336	0.96618	0.93613
1.7×10^8	0.99715	0.96061	0.93078
2.25×10^8	0.996674	0.9655	0.939653
4×10^8	0.993574	0.967455	0.94101
1.2×10^9	0.99641	0.96822	0.938079

TABLE III
Efficiency Analysis for Parallel algorithm

From Table(I) we can state that when we increase the number of threads, walltime decreases.
From Table(II) we can state that upon increasing the number of threads, speedup increases.
From Table(III) we can say that upon increasing the number of threads, efficiency decreases.

we have also shown the graphical results for the above tables as well.

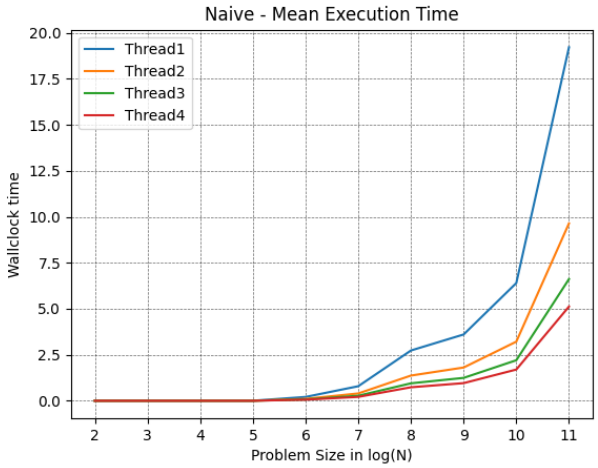

Fig. 4. Walltime vs log(N)

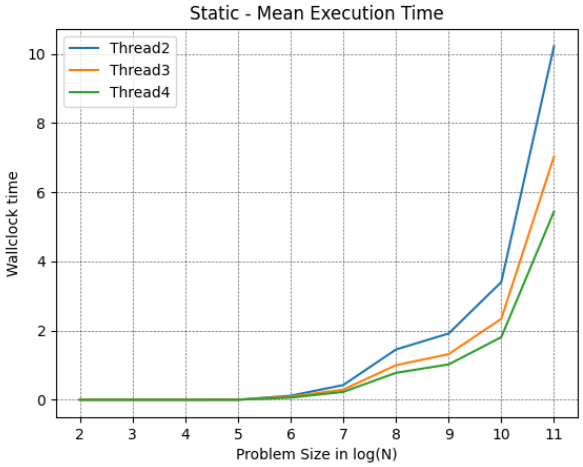

Fig. 5. Walltime vs log(N)

Figures no. 4, 5 and 6 shows the behaviour of the time taken by the first implementation of the parallel bellman algorithm,we can see that Guided gives the minimal Mean Execution Time, later this will also be reflected in the speedup analysis.
Figures no. 7, 8 and 9 shows the speedup analysis for all three Naive, Static and Guided Implementations.

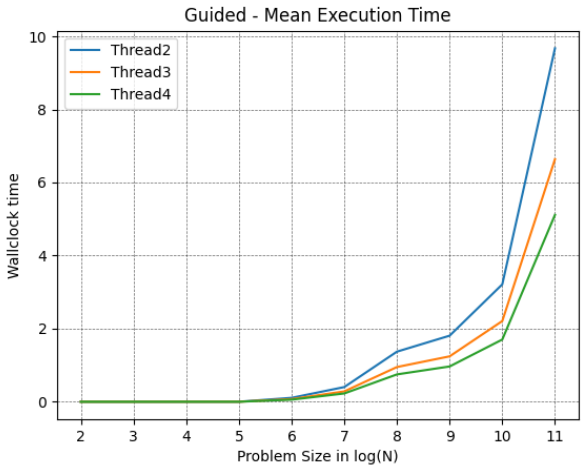

Fig. 6. Walltime vs log(N)

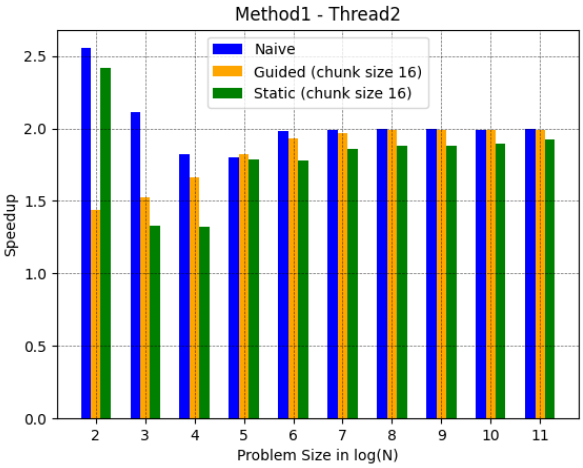

Fig. 7. SpeedUp vs log(N)

We can clearly see that in all three plots(separate for number of threads) the speedup for the Naive approach is Higher or almost equal to the theoretical speedup. This is because the serial algorithm for this approach is not very good.
If we keep the Naive parallelization out of the discussion then the Guided is better than Static in term of speedup.
speedup is higher for naive approach because of poor execution time for serial algorithm. but speedup of guided is higher than static.

Here is the efficiency graph for all the threads for guided scheduling.
The graph above shows that despite the simplicity of the algorithm in terms of the implementation, the method used for the serial implementation is very efficient since the efficiency approaches theoretical efficiency (1).

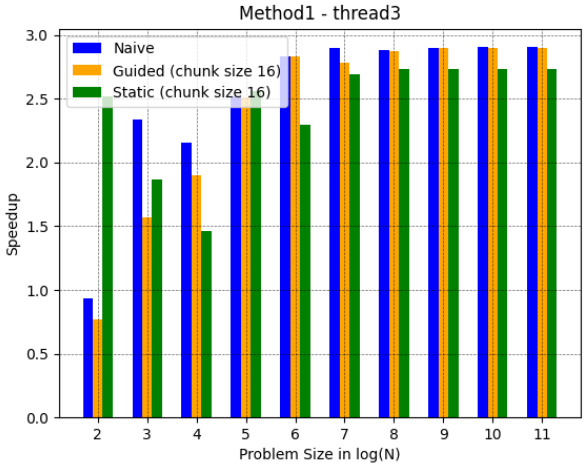

Fig. 8. SpeedUp vs log(N)

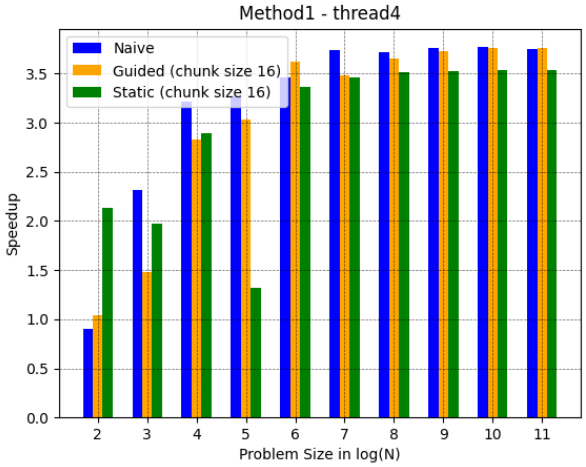

Fig. 9. SpeedUp vs log(N)

IX. Analysis of the Parallelization

Below is the graphs of walltime for different problem size. We use three methods to parallelize our code: Naive approach, Static Scheduling (for chunk size 16) and Guided Scheduling (for chunk size 16).

We generate the output in the form of the bar graph for different problem size. The graphs are represented in Fig. 7, Fig. 8 and Fig. 9. We generate the Speedup graphs for Thread 2, thread 3 and thread 4.

X. Parallel Algorithm Shortest Path Faster Algorithm

Bellman-ford faster algorithm can also be run in parallel as the inner 'for' loop has no loop carried dependencies. Below we can see the loop we are going to parallelize.

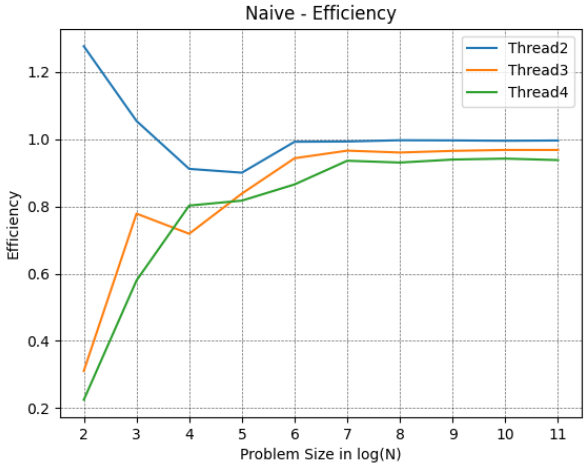

Fig. 10. Efficiency vs log(N)

Algorithm 6 Pseudo-Code for Shortest Path Faster Algorithm

distance array $dis[v]$
$dis[source] \leftarrow 0$

push $source$ into $Queue$
while $Queue$ is not empty **do**
  $u :=$ poll $Queue$
  for each edge from $u, (u, v)$ **do in parallel**
    if $dis(u) + weight(u, v) < dis(v)$ **then**
      $dis(v) = dis(u) + weight(u, v)$
      v into $Queue$
    end if
  end for
end while

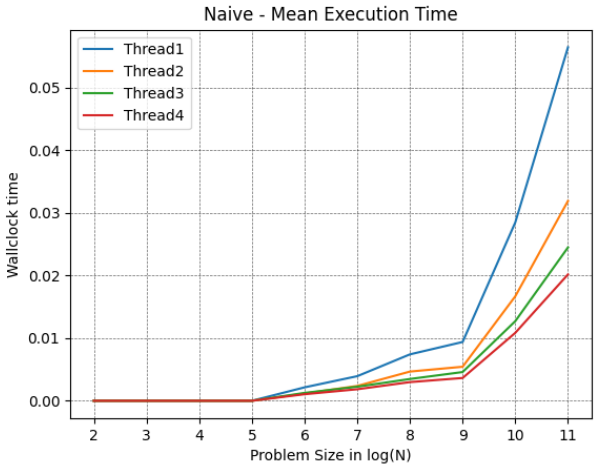

Fig. 11. Walltime vs log(N)

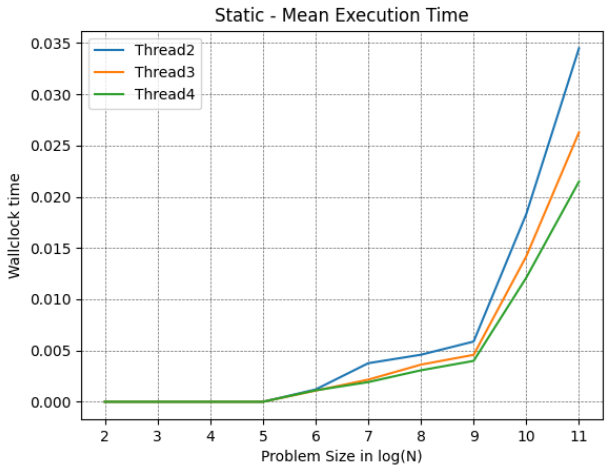

Fig. 12. Walltime vs log(N)

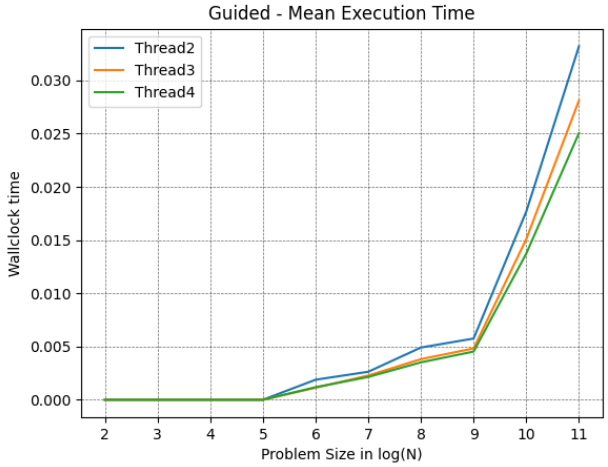

Fig. 13. Walltime vs log(N)

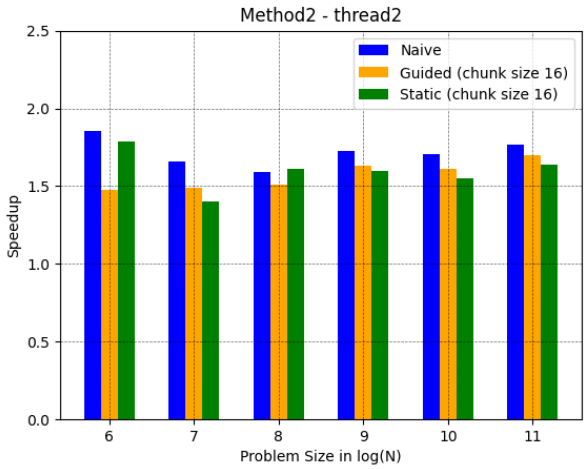

Fig. 14. SpeedUp vs log(N)

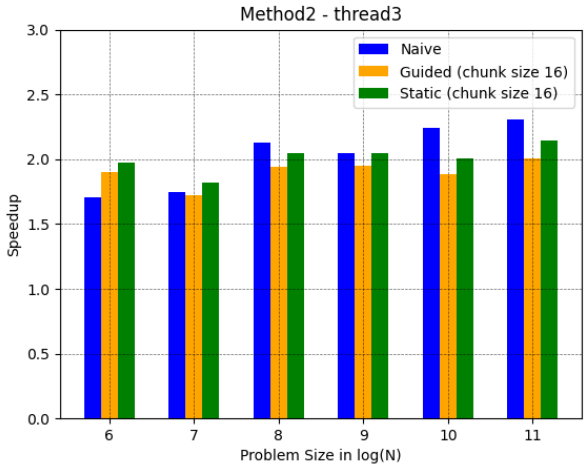

Fig. 15. SpeedUp vs log(N)

XI. Results for faster approach

Figure No. 11, 12 and 13 shows the nature of the run-time over size of the problem size by implementation of the parallel bellman algorithm for the fast-bellman method(approach-2). In this approach Guided and Static takes almost same amount of the time for complete execution.

Unlike speedup analysis for the first method, the second method does not show the theoretical behaviour reason being the good implementation of the serial algorithm. As we have improved the algorithm to a level where it takes lesser serial execution time, we expected that the speedup will be lesser than the first method.

Figure No. 14,15 and 16 shows the bar graph for the speedup of the second method.

We can see that for the thread2 the speedup of 2 is not achieved,because the algorithm for this method took lesser time to run serially.

The Efficiency plot (Figure No. 17) says alot about the algorithm, we can see that the efficiency increases as we increase the problem size. Now for the problem size of 10^{11}, respective efficiencies are 0.78 and 0.72,which shows that algorithm has largely the parallelizable part and lesser serial part.Which indicates that method used to implement the serial algorithm is very good.

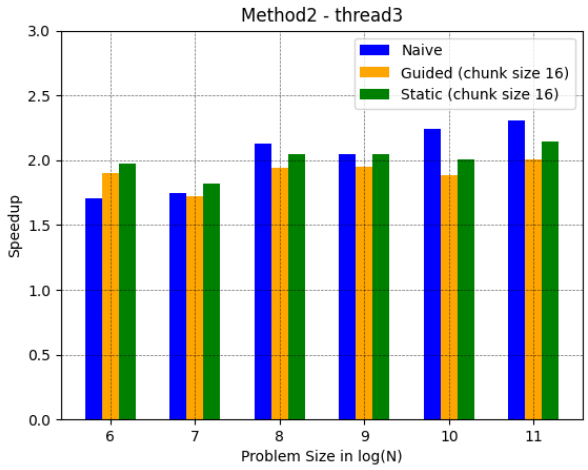

Fig. 16. SpeedUp vs log(N)

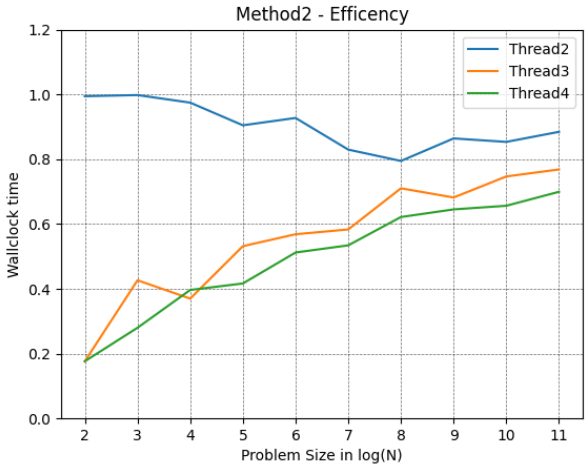

Fig. 17. Efficiency vs log(N)

XII. Profiling using valgrind

For the profiling purpose we have used memcheck and cachegrind tools of valgrind. using memcheck we corrected our code for memory access errors and cachegrind for measuring cache miss-hit .cache grind measures the L1mises and LLmises which in our case is L3. for cachegrind profiling we have used a testcase of size more than 16kb which is our L1 cache size. And also to see effects of how density of graph affects of cache hit-miss we used two testcase of same numbers of edges , but one is relative sparse and one is dense.

For naive approach:

noindent textcolorredFor naive approach we are getting D1 miss rate is 0.2 for sparse and 0.4 for dense graphs. In sparse graphs total instruction references are 948,063,448 and 1,988,276 are miss.In dense graph total Instruction references are 50,961,918 and 213,558 are misses.

For faster approach:

In sparse graphs we are getting D1 miss rate is 0.2 and for dense graph 0.6 .In sparse graphs total instruction references are 33,102,680 and 58,536 are miss.In dense graph total Instruction references are 166,666,033 and 939,874 are misses.

Improvements for cache missrate

We seen that in dense graph cache missrate is much higher so to improve that we implemented adjacency matrix as a data-structure to store the distance between two edges. And results for dense graphs improved a lot .In dense graphs we are getting D1 miss rate is 0.1 and for sparse graph 0.6.
In dense graphs total instruction references are 22,158,234 and 22,535 are miss.But in sparse graph results were not good. total Instruction references for sparse graph are 165,825,726 and 941,407 are misses.

Explanation

This shows how different data-structures affects the cache miss-hit factor on the different testing conditions. In adjacency matrix all distance connected to a nodes are adjacent elements of array so in dense graph this will make use of cache very well but in sparse graph most of this values will be infinite.

So in conclusion for dense graph adjacency matrix will be more cache efficient and for sparse graphs adjacency list will be more cache efficient in faster version of bellman-ford.

XIII. MPI Based Implementation and Analysis

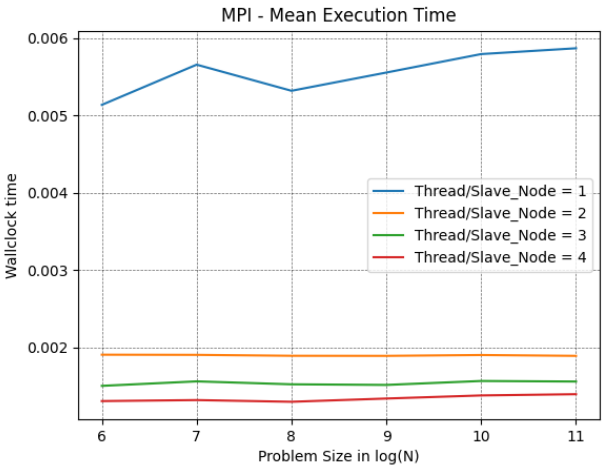

Fig. 18. Walltime vs log(N)

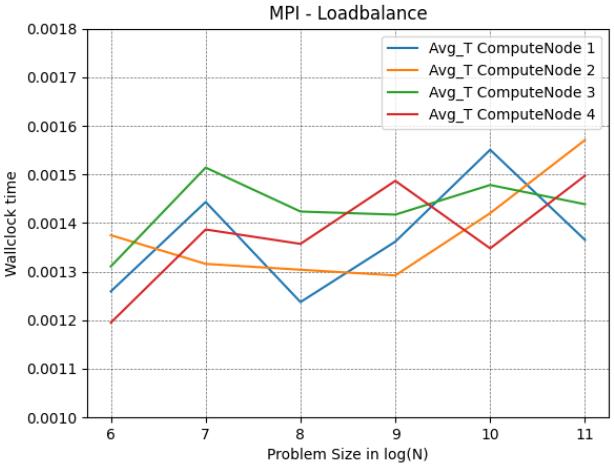

Fig. 19. Average Load-balance vs log(N)

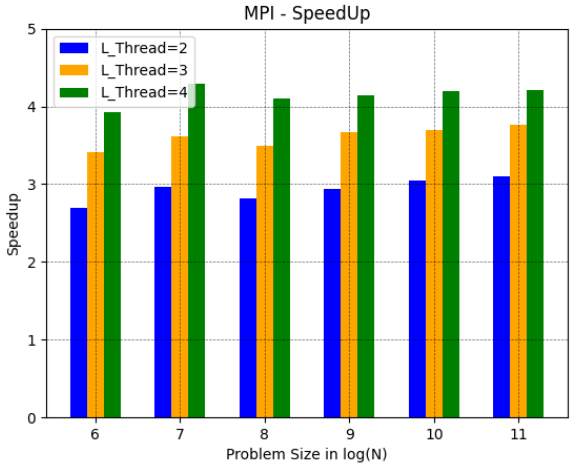

Fig. 20. SpeedUp vs log(N)

For the MPI based analysis of the bellman ford algorithm we were provided with the HPC cluster in which there are total 4 slave nodes under 1 master node. In Message Passing Interface,the Master node distributes the work amongst all four slave nodes, these slave nodes may or may not run with multiple threads.

Figure No. 18 is the plot for time taken by the cluster to finish the execution of the algorithm for different size of input and different slave node configuration.For the MPI based analysis we have subsequently spawned 1,2,3 and 4 threads in each slave node. We can clearly see that when number of threads spawned in each slave node is lesser then it will take the significant amount of time to finish,but in case of 4 threads in each time taken to finsh is nothing comparable to the first case.

Figure No. 20 shows the speedup analysis for different numbers of threads.We can clearly see that the speedup in case of maximum number of threads in slave nodes is highest.

Now we know that before the execution starts Master node assigns the work to each and every slave nodes, to take most out of the hardware of the cluster the work allocation must be done optimally and equally.To check this equality of work sharing ,there is a term called load balancing. Now during the run time if any slave node is not doing anything and any other slave node is doing most of the work then overall time to finish the job will increase ,thus resulting in the bad load-balancing.

Figure No. 19 shows the load balancing for every slave node(or compute node). we can clearly see that, for every problem size each and every slave nodes are doing approximately same work ,so the algorithm will take the least possible time to finish and load balancing is done well enough.Which again proves that the algorithm is very good and well implemented.

XIV. Current State of Art and Contribution

To the best of out knowledge no one has published results for parallel version of faster algorithm which we have implemented here. So this is our contribution in the field. There are many papers with delta stepping algorithm but delta stepping algorithm gives less execution time and is highly parallelizable so there is no means by comparing our results with them.

XV. Acknowledgment

We would like to thank prof. Bhaskar Chaudhury for giving us this great opportunity.As field of HPC was new for all of us this wouldn't be possible without support and lessons of Professor.Also we would like to thank out teaching assistants Mr. Mihir Desai and Ms. Shivani Nandani for constant support.

References

[1] SimpliLearn, Bellman-Ford Algorithm: Pseudocode, Time Complexity and Examples. https://www.simplilearn.com/tutorials/data-structure-tutorial/bellman-ford-algorithm

[2] Programiz, Bellman-Ford's Algorithm. https://www.programiz.com/dsa/bellman-ford-algorithm.

[3] GeeksForGeeks, Bellman-Ford Algorithm, DP-23. https://www.geeksforgeeks.org/bellman-ford-algorithm-dp-23/

[4] Cornell University, More Parallelism in Dijkstra's Single-Source Shortest Path Algorithm, Michael Kainer, Jesper Larsson Träff. https://doi.org/10.48550/arXiv.1903.12085

[5] Algorithms for Competitive Programming, Bellman-Ford Algorithm, Single Source Shortest Path with negative weighted edges. https://cp-algorithms.com/graph/bellman_ford.html

[6] Randomized Speedup of the Bellman-Ford Algorithm, Michael J. Bannister and David Eppstein, Computer Science Dept. University of California, Irvine.

[7] A Heuristic Improvement of the Bellman-Ford Algorithm, Andrew V. Goldberg, Computer Science Department, Stanford University.

[8] Hybrid Bellman-Ford-Dijktra algorithm, Yefim Dinitz, Rotem Itzhak, Dept. of Computer Science, Ben-Gurion University of the Negev.

[9] The Bellman-Ford Algorithm and "Distributed Bellman-Ford", David Walden, E. Sandwich, MA.

[10] Scalable Single Source Shortest Path Algorithms for Massively Parallel Systems, Venkatesan T. Chakarvarthy, Fabio Checconi, Prakash Murali, Fabrizio Petrini, Yogish Sabharwal: Combinatorial Optimization and Parallel Computing.

[11] https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.676.8450&rep=rep1&type=pdf

[12] On a Routing Problem, Quarterly of Applied Mathmatics, Brown University. https://www.ams.org/journals/qam/1958-16-01/S0033-569X-1958-0102435-2/
[13] Network Flow Theory, L.R. Ford https://www.rand.org/pubs/papers/P923.html
[14] The Bellman-Ford algorithm and distributed Bellman-Ford. https://www.researchgate.net/publication/250014977_THE_BELLMAN-FORD_ALGORITHM_AND_DISTRIBUTED_BELLMAN-FORD
[15] Testcase Generator. https://test-case-gen.herokuapp.com
[16] Moore, Edward F. (1959). The shortest path through a maze. Proc. Internat. Sympos. Switching Theory 1957, Part II. Cambridge, Massachusetts: Harvard Univ. Press. pp. 285–292.
[17] Bellman-Ford Single Source Shortest Path Algorithm on GPU using CUDA by Raj Sengo and Sthephan Garland https://towardsdatascience.com/bellman-ford-single-source-shortest-path-algorithm-on-gpu-using-cuda-a358da20144b
[18] A Fine Tuned Hybrid Implementation for Solving Shortest Path Problems using Bellman Ford by Gaurav Hajela and Manish Pandey
[19] Shortest Path Faster Algorithm https://en.wikipedia.org/wiki/Shortest_Path_Faster_Algorithm#cite_note-6