University of Washington, Tacoma

TCSS 543: Advanced Algorithms

Empirical Study Project

Due: Saturday, November 30, 2019, 1:30pm

Shang Wang

Jin Young Park

**Abstract**

Network flow problems are related to network traffic. And the problem is mostly solved by finding the maximum flow. We learned two algorithms to find the maximum flow in class. One is the Ford-Fulkerson algorithm and the other is the Scaling Ford-Fulkerson algorithm.

The Ford-Fulkerson algorithm pushes the flow on the directed graph G. Then create a residual graph with graph G and pushing flow. And push the flow again on the residual graph. Repeat this until it no longer finds the flow in the residual graph. The total flow thus pushed be the max flow. The running time of this algorithm is . (m is the number of edges in the directed graph G, and C is sum of capacities of edge out of source node).

The Scaling Ford-Fulkerson algorithm is similar to the Ford-Fulkerson algorithm. The difference is that they have a delta. Use the delta to try to increase the flow by a large amount each iteration, instead of increasing it by 1unit. The delta is initially set with the largest power of 2 that is no longer then. Then it is run like Ford-Fulkerson algorithm. However, the residual graph has a different form from the Ford-Fulkerson algorithm. The residual graph in the Scaling Ford-Fulkerson algorithm except all edges must have weight at least delta. And at the end of the iteration, the value of the delta is divided by 2. The loop ends when the value of delta is less than 0. The total flow thus pushed be the max flow. The running time of this algorithm is .

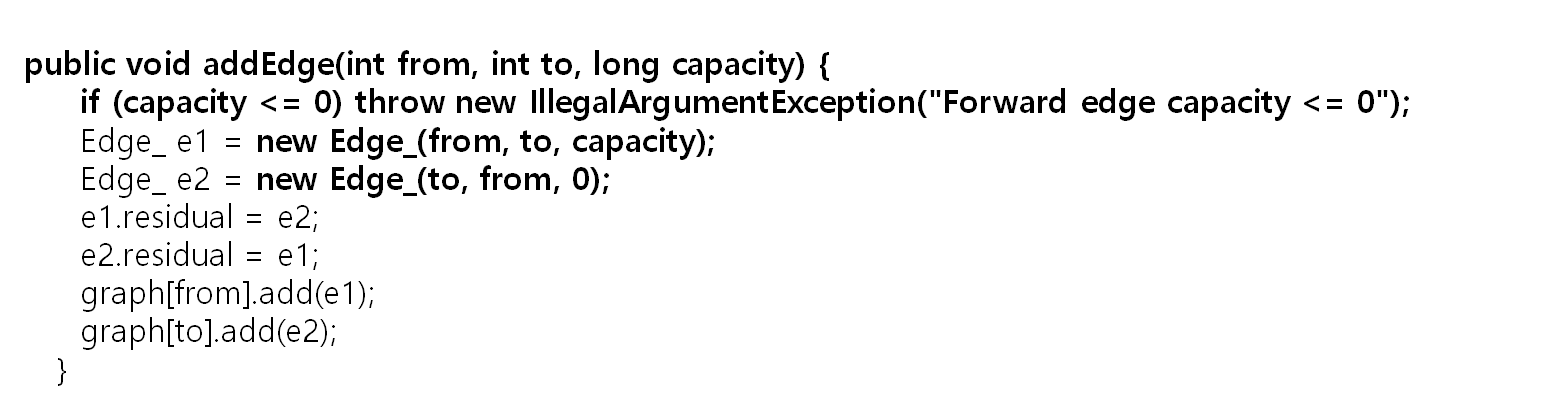
In the description of algorithms, running times of two algorithms are different for the same graph. This project compares and analyzes the running time of two algorithms under the same conditions. Depending in the number of edges and the size of the capacity, the algorithms that run faster vary. The goal of this project is to find out which algorithms are more fast depending on the number of edges and size of the capacity.

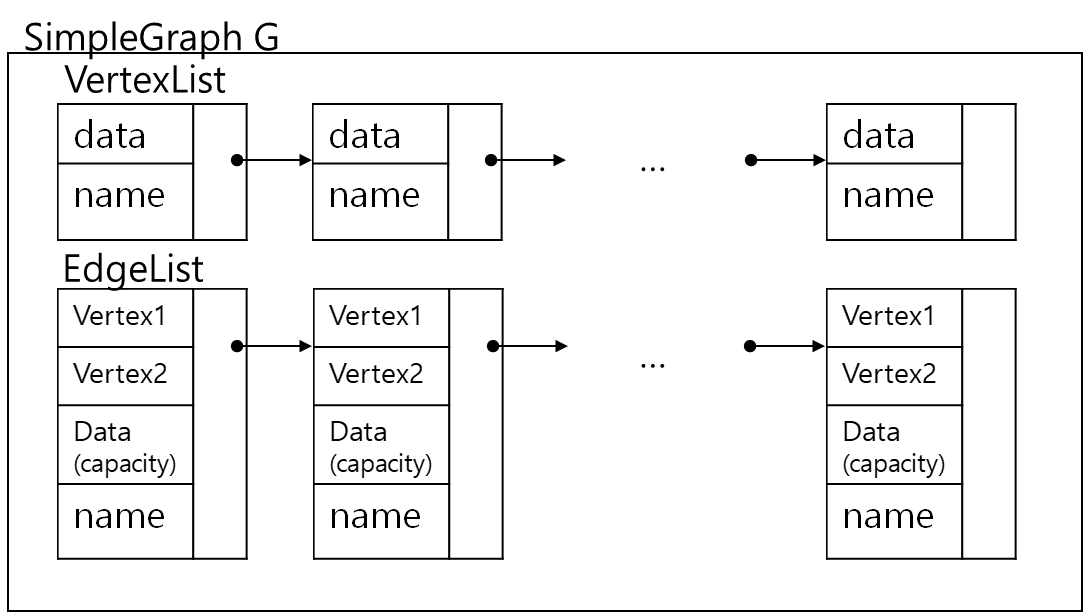
**Methodology**

1. Code

We borrowed the Ford-Fulkerson algorithm, and the scaling Ford-Fulkerson algorithm. This code uses an array list as a graph. In our opinion, when creating a graph, SimpleGraph class in the type of linked list is faster, but linked list is inconvenient and slow when searching the list while executing an algorithm. So, we changed SimpleGraph class to an array list and ran it.

Create a function called LinkedlistToArrayList() in the class NetworkFlowSolverBase and change the SimpleGraph class to an array list graph[]. The array list graph[] is a list created from the start node of the edges. Edge is used by creating a class called Edge\_. In the Edge\_ class, ‘vertex1’ of the SimpleGraph class is ‘from’ and ‘vertex2’ is ‘to’. There are new parameters ‘flow’ and ‘capacity’. We store the input capacity value of the Edge class in SimpleGraph. We put this capacity in the capacity of the Edge\_ class. In the case of flow, it indicates the flow of this edge. Also, there is the residual graph of the Edge\_ class type as a parameter, ‘residual’. Class Edge\_ uses addEdge() function to create edges. The addEdge() function creates and adds a residual edge when adding an edge to graph[].





* 1. Ford-Fulkerson algorithm

The NetworkFlowSolverBase class in this code creates a graph and executes the Ford-Fulkerson algorithm. They have some parameters. The following is a description of the parameters in the code. “'visited' and 'visitedToken' are variables used in graph sub-routines to track whether a node has been visited or not. In particular, node 'i' was recently visited if visited[i] == visitedToken is true. This is handy because to mark all nodes as unvisited simply increment the visitedToken.”, ‘solved’ is“Indicates whether the network flow algorithm has ran. The solver only needs to run once because it always yields the same result.”, ‘maxFlow’ is “The maximum flow. Calculated by calling the {@link #solve} method.”, and ‘graph[]’ is “The adjacency list representing the flow graph.” The initializeEmptyFlowGraph is gather the Edge\_ classes representing the edges in list type and make it as a graph. The LinkedlistToArrayList function in this class is a function we created that converts the graph from the SimpleGraph class to the appropriate type because the graph of the borrowed code is not in the SimpleGraph class. And also the class has the addEdge() function.

The FordFulkersonDfsSolver is a class that implements the Ford-Fulkerson algorithm using a depth first search, we learned in class. The solve() function of this class, executed by the NetworkFlowSolverBase class, calls the dfs function. The dfs() function finds the flow using the residual graph and changes the residual graph with the bottleneck of this flow. Then add the found flow to the parameter maxflow. If no more flows are found in the graph, the maxflow is returned by returning 0 is the for statement of the solve function. Then we can get maxflow by this code.

* 1. Scaling Ford Fulkerson Algorithm

The NetworkFlowSolverBase class in this code is same as theNetworkFlowSolver Base in the Ford-Fulkerson Algorithm.

The difference between the Ford-Fulkerson algorithm and Scaling Ford Fulkerson algorithm is the CapacityScalingSolver class. The CapacityScailinglSolver class creates a parameter delta. When the addEdge function is executed, it adds an edge and stores the larger of the capacity of the edge and the existing delta value in delta, so the delta is the largest value of the capacity. And when the solve function is executed, delta saves the largest bit of the existing delta value as the value. The CapacityScailinglSolver is a class that implements the Scaling Ford-Fulkerson algorithm in the Dfs format we learned in class. The solve function of this class, executed by the NetworkFlowSolverBase class, calls the dfs function through the for loop. The dfs() function finds flows in a residual graph consisting of edges above the delta value and changes the residual graph with the bottleneck of this flow. Then add the found flow to the parameter maxflow. At the end of each loop iteration, the delta is divided by two. If delta is less than 0, the loop ends and the maxflow is returned. Then we can get maxflow by this code.

1. Input Graph Code

To execute the Ford-Fulkerson algorithm and Scaling Ford-Fulkerson algorithm, we must have a graph of type SimpeGraph class. So, we created graph generating codes. Generating random graph, bipartite graph and mesh graph takes input values and create files and graphs of SimpleGraph class type.

The given class SimpleGraph consists of linked list vertexList and edgeList. The vertexList consists of Name and Data. The edgeList consists of vertex1, vertex2, Data, and Name. We stored the start node of the edge in vertex1, the end node of the edge in vertex2, and the capacity in the data.

2.1. Random graphs (each edge is generated with probability p)

We used the given BuildGraph code. The input of this code is filename, directory, vertices, dense, minCapacity, and maxCapacity.

vertices: the number of vertices.

dense: the probability to create of edges.

minCapacity: the lower bound on the capacity value of each edge.

maxCapacity: the upper bound on the capacity value of each edge

Vertices are added to SimpleGraph G by the number contained in the parameter vertices. The source node *s* is the first vertex added to SimpleGraph G, and the sink node *t* is the last vertex added to SimpleGraph G. LinkedlList vertexList is created, this list stored vertexList in the SimpleGraph G. The graph is built on the array type, Graph[vertices][vertices]. And Graph[n][m] represents the edge(n,m).

Two numbers are randomly generated, randomInt (minCapacity <= randomInt <= maxCapcity) and k, and if k is greater than dense, the Graph[n][m] has capacity is 0, and else the Graph[n][m] has capacity is randomInt. Repeat this to fill the array Graph[][].

In for(int x = 0; x < Graph.length; x++) loop, edges with a capacity greater than 0 are printed to file and added to SimpleGraph G.

2.2. Bipartite graph

We used the given Bipartite Graph code. BipartiteGraph have input parameters n, m, maxProbability, minCapacity, and maxCapacity.

n: the number of sink side nodes

m: the number of source side nodes

maxProbability: the probability between the minCapacity and maxCapacity.

minCapcity: the lower bound on the capacity value of each edge.

maxCapacity: the upper bound on the capacity value of each edge

the code makes SimpleGraph g and firstly source node s is added to the SimpleGraph G, then n+m vertices are added, and sink node t is added. In other words, the first vertex of the vertexList in SimpleGraph G is the source node *s* and the last vertex is the sink node *t*. Then, LinkedList vertexList is created and is stored SimpleGraph's VertexList. Also array edge is created, edge[n][m].

In the first for loop, vertex v is represented the source side node and vertex w is represented the sink side node. edge(v,w) is created and the capacity of this edge is randomly generated but capacity must always be less than or equal to maxProbability. So, if randomly generated number is greater than maxProbability, the capacity is just 0. If the value is edge is not 0, add an edge whose capacity is the number into SimpleGraph G.

The edge, the start node is source and the end node is source side node, is added to SimpleGraph G, and this edge’s capacity is randomly generated between minCapacity and maxCapacity.

The edge, the start node is sink and the end node is sink side node, is added to the SimpleGraph G, and this edge’s capacity is randomly generated between minCapacity and maxCapacity.

2.3 Mesh graph

The mesh graph is like the matrix, so we used the way to generate matrix to generate the mesh graph. Mesh graph has input parameters col, row, lower, upper.

Col: the number of column

Row: the number of row

Lower: the lower bound on the capacity value of each edge.

Upper: the upper bound on the capacity value of each edge.

The code makes SimpleGraph g and firstly source node s is added to the SimpleGraph G, then n\*m vertices are added, and sink node t is added. In other words, the first vertex of the vertexList in SimpleGraph G is the source node *s* and the last vertex is the sink node *t*. Then, LinkedList vertexList is created and is stored SimpleGraph's VertexList. Also the edge is correspondingly added.

In the first for loop, vertex v is represented the source side node and vertex w is represented the sink side node. edge(v,w) is created and the capacity of this edge is randomly generated but capacity must always be less than or equal to maxProbability. So, if randomly generated number is greater than maxProbability, the capacity is just 0. If the value is edge is not 0, add an edge whose capacity is the number into SimpleGraph G.

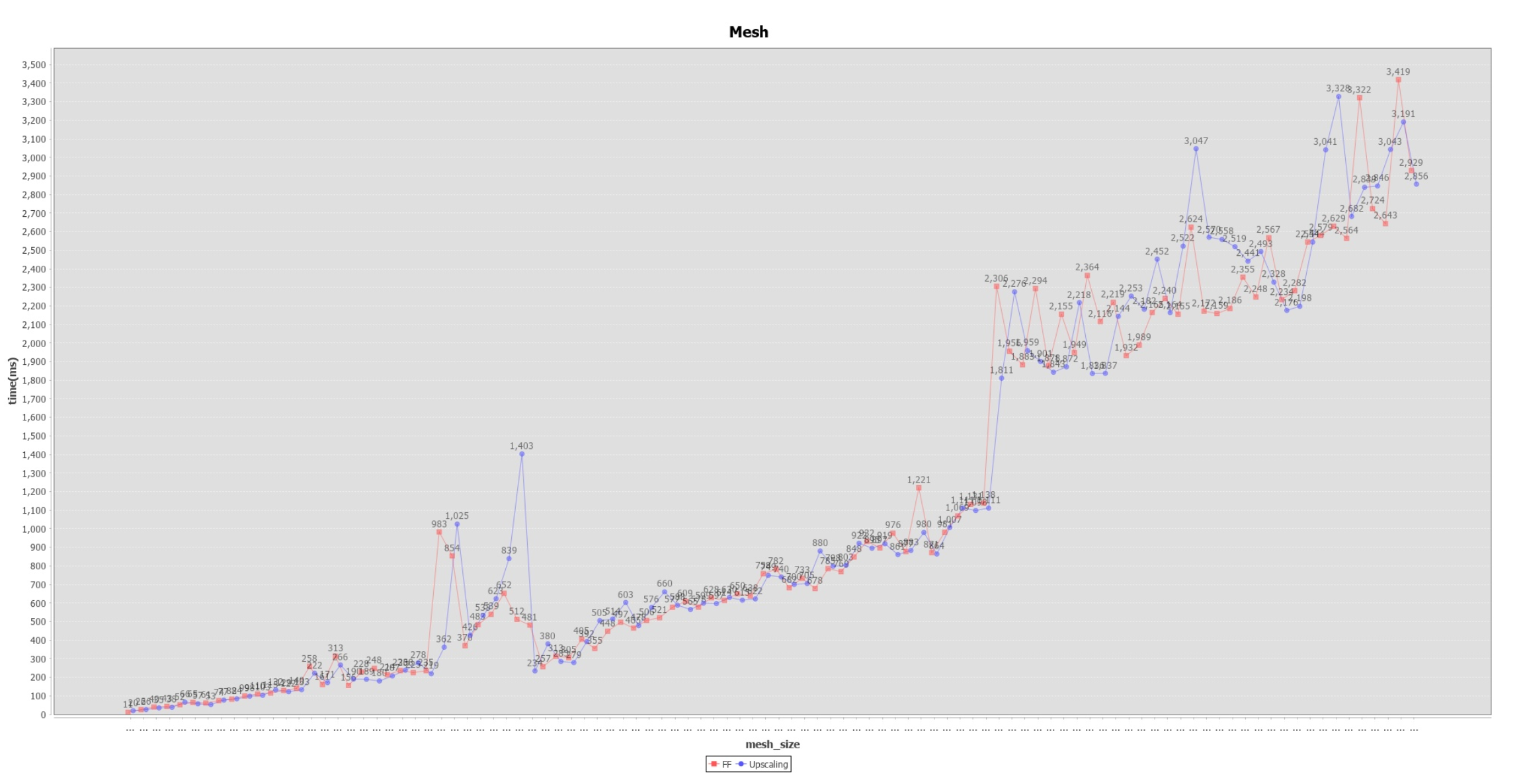
The edge, the start node is source and the end node is source side node, is added to SimpleGraph G, and this edge’s capacity is randomly generated between minCapacity and maxCapacity.

**Results**

1. Mesh graph

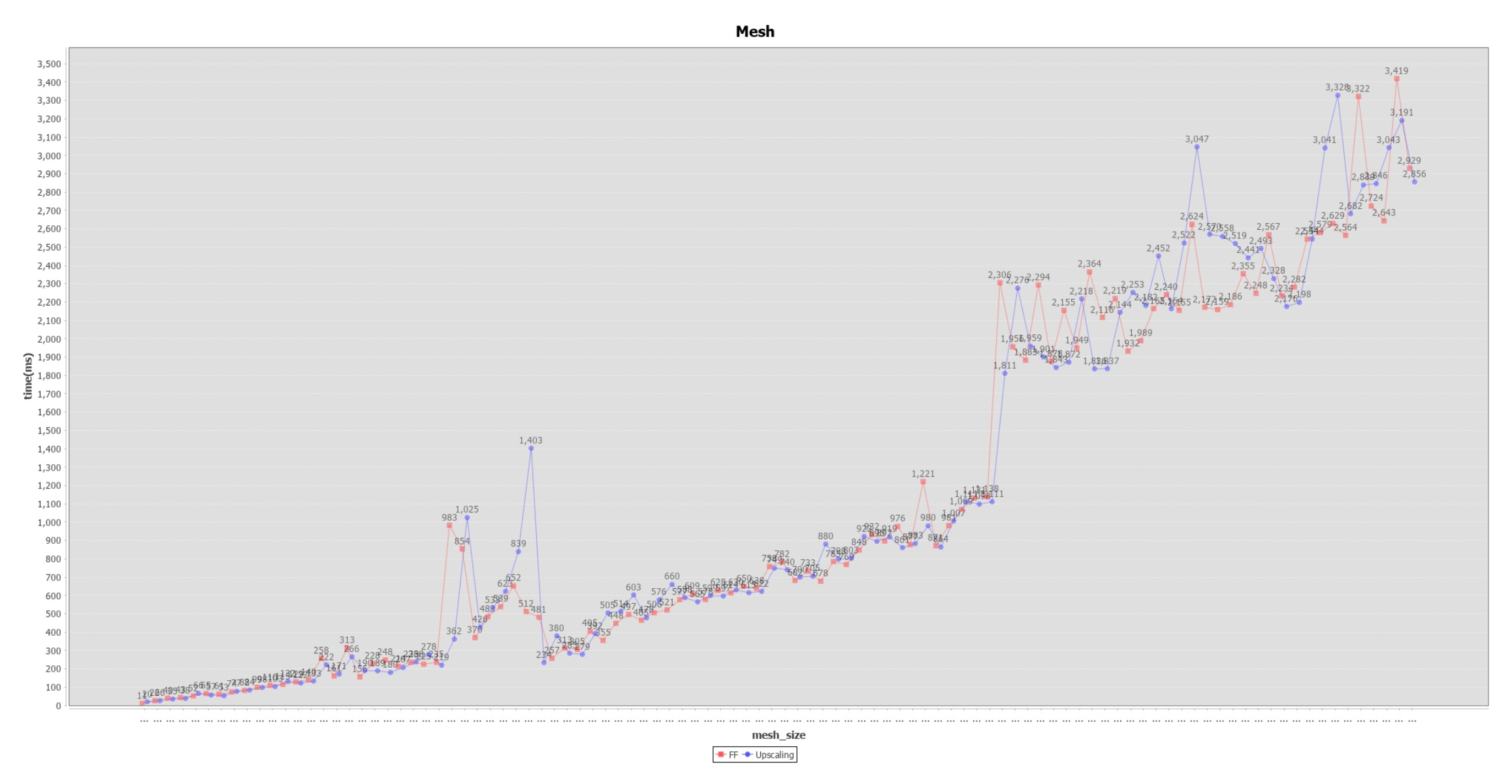
Fixed the col, every time the number of row would be increased by 10,

So the graph would be narrower and narrower. The graph col length is 10, and row length is increased from 10 to 1000. The capacity of sides is random chose from 2 to 10

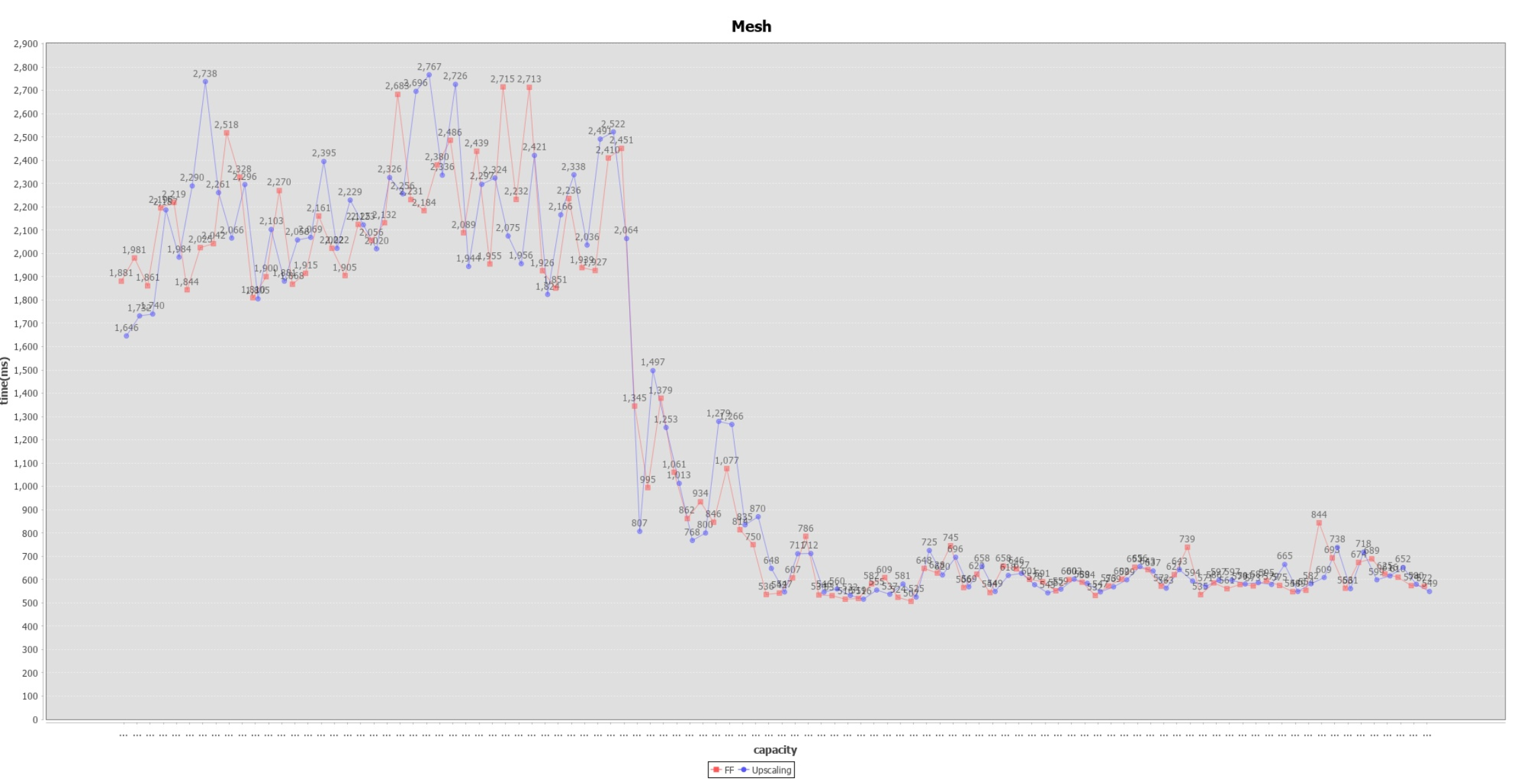


Fixed the row, every time the number of col would be increase by 10,

So the graph would be wider and wider. The graph row length is 10, and row length is increased from 10 to 1000. The capacity of sides is random chose from 2 to 10

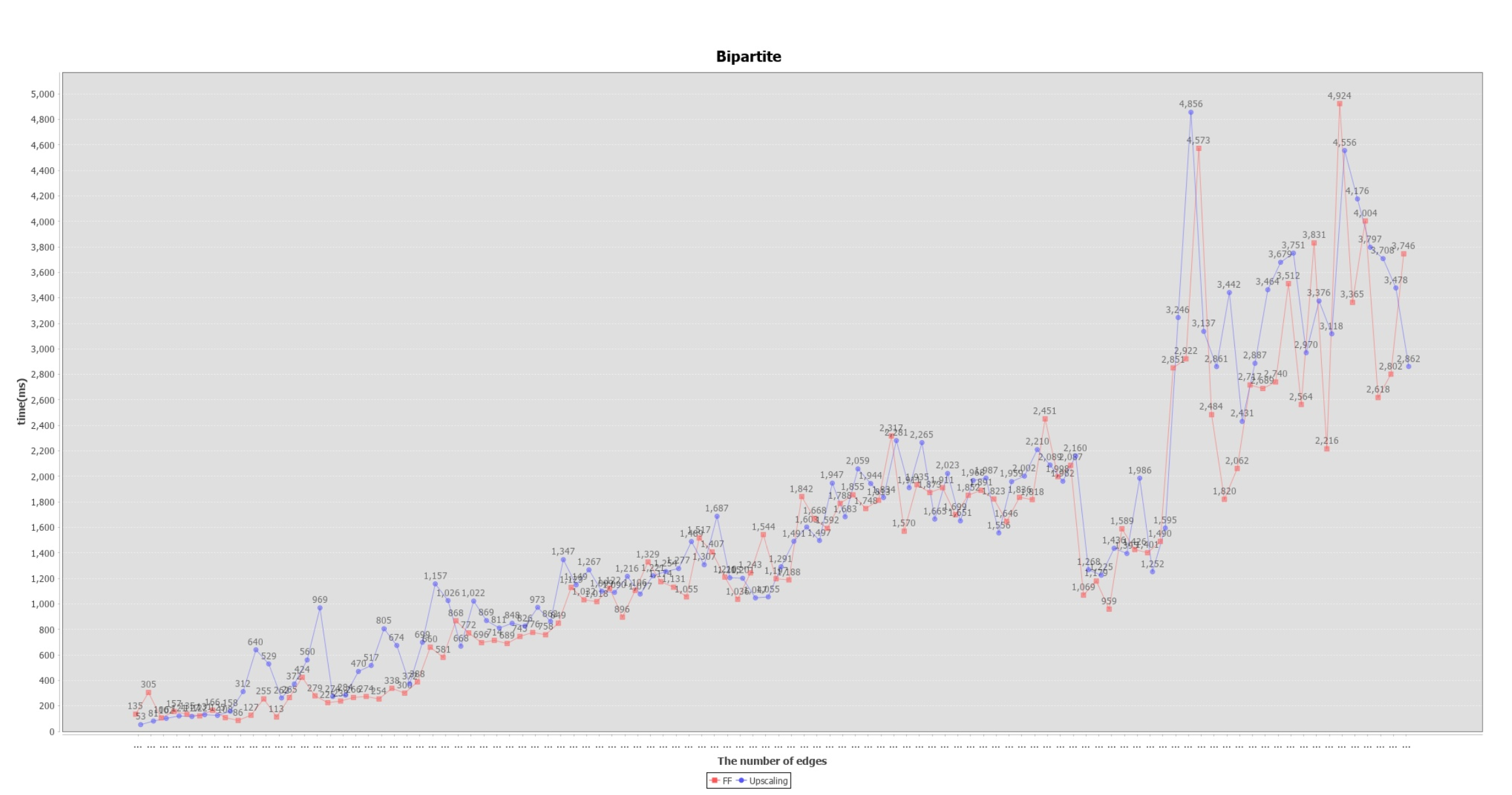


Fixed the number of edges, every time the capacity of edges from source node would be increased by 10. The size of mesh graph is 100\*100 and the capacity would increase from 10 to 1000.

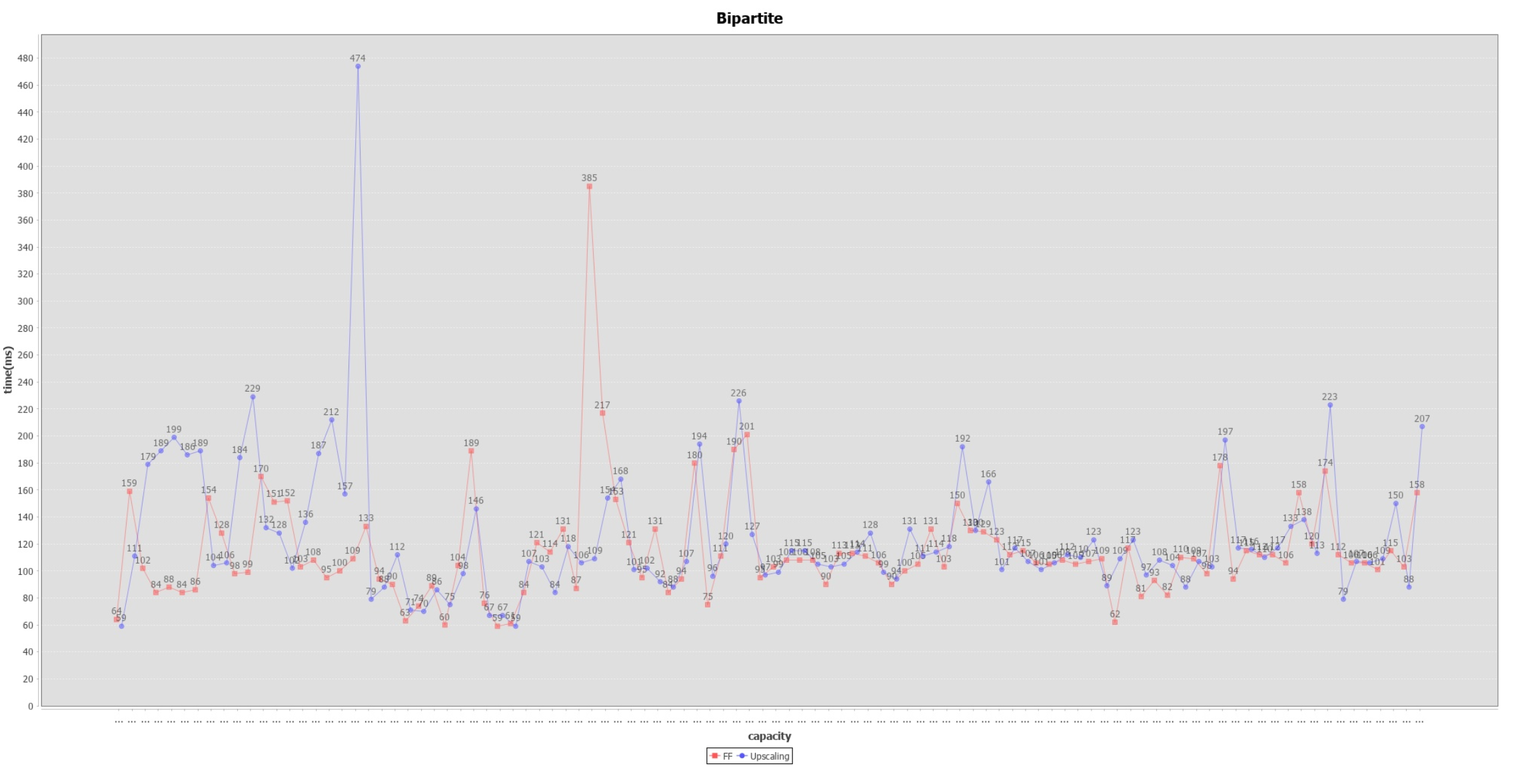


1. Bipartite graph

Fixed capacity, every time the number of sink side vertexes would be increasing by 10. The number of source side nodes is 40 and the number of sink side nodes is increased from 10 to 1000. The capacity of sink and source side nodes are same with 6.



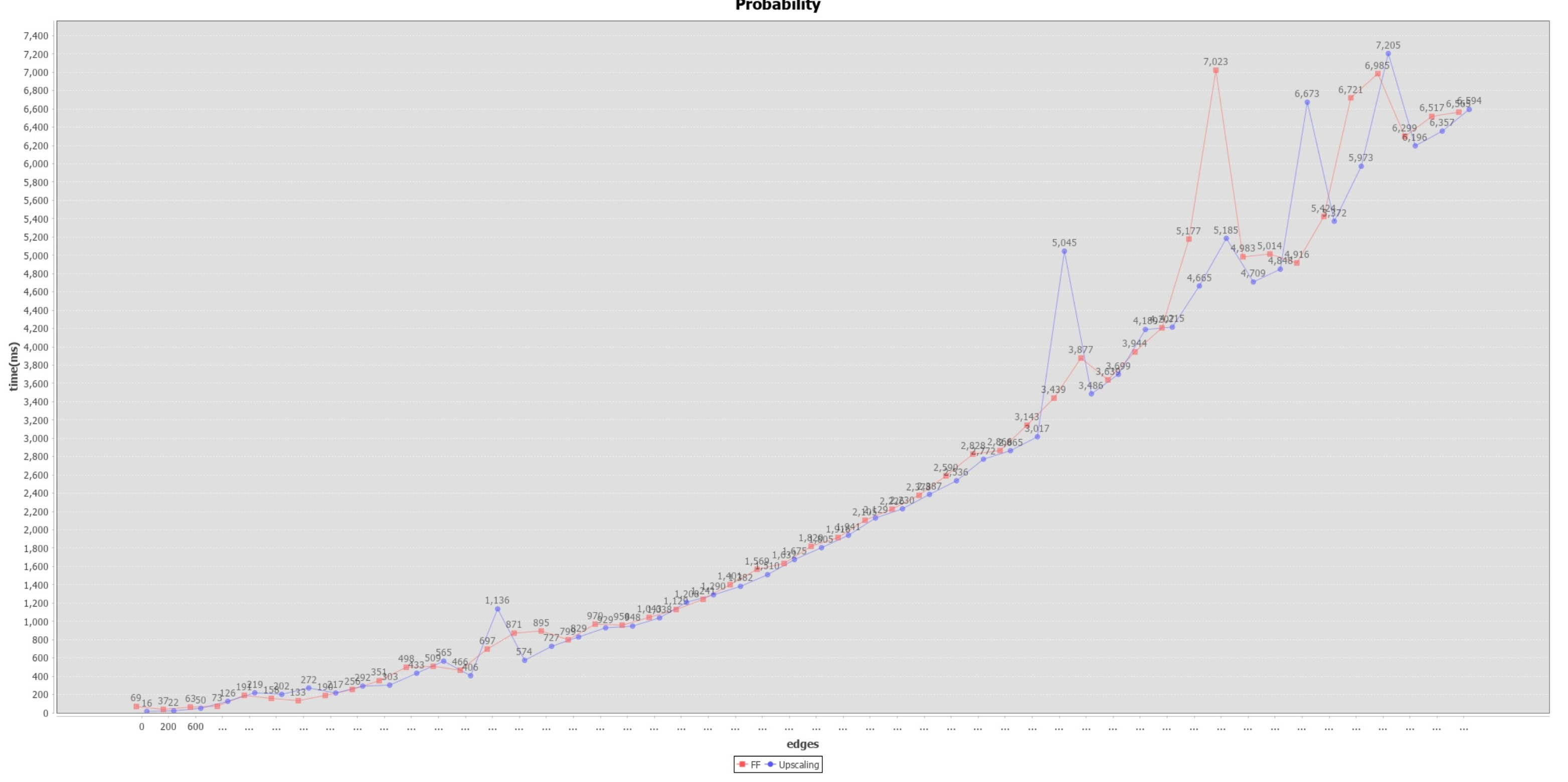
Fixed edges, every time the capacity of source side vertexes would be increased by 10. The number of source side nodes is 40 and the number of sink side nodes is 40. The capacity increased from 10 to 1000



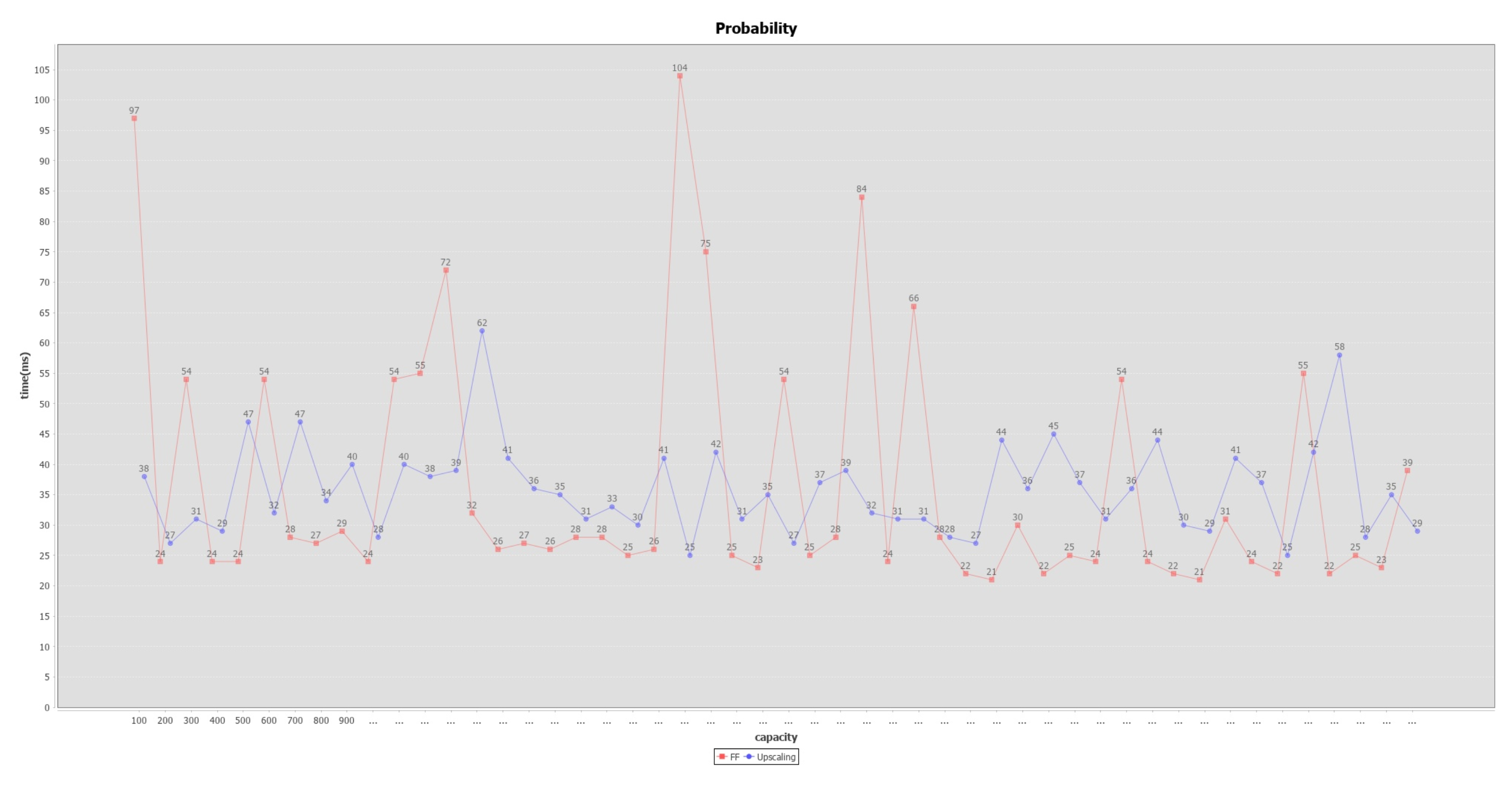
Obviously, at first, the capacity is not big enough and the time complexityof Upscalling algorithm is m^2logC, so the advantage of this algorithm is tiny(logC) and the disadvantage of this algorithm is highlighted(m^2).

1. Probability graph

Fixed capacity, every time the number of vertexes would be increased by 10, correspondingly, the number of edges is increased.



Fixed edges, every time the capacity of edges connected with source node is increased by 10.



**Future work**

Due to the limit of time and computation power, the result is not easy to analyze. We would try much more data(expand data size) and analyze the result. We need to encapsulate the code and create more interface in order to make it easier to use. We also should make a change on drawing graph function.