

CS 456: Advanced Algorithms

Programming Assignment 03

Bhavana Kurapati (800665827)

bkurapa@siue.edu

Program Execution:

A make file compiles all the algorithms and to run the program we use:

```
./MaxFlow
```

Then it shows:

Please input file name: input.dat (example)

Then the output will be generated in input.out file.

Motivation and Background:

Maximum Flow is the maximum amount of flow that the network would allow to flow from source to sink. All we concern about Max Flow Problem is How to Maximize the flow in a Network from source to sink. Many Algorithms exists to find the Maximum Flow. Some of them are Dinic's Algorithm, Ford-Fulkerson Algorithm, Edmonds-Karp Algorithm, Push Relabel Algorithm, Binary blocking flow Algorithm, MPM Algorithm etc., and the applications include Maximum cardinality bipartite Matching, Maximum Flow with vertex Capacities, Maximum number of paths from source to sink, etc. Real World Applications include Airline Scheduling, Baseball elimination, Circulation demand problem.

Pseudocode and Implementation:

Ford-Fulkerson Algorithm:

Input:

```
CapacityMatrix (n*n);  
Adjacency_list (n) // created by DFS node contains the nodes that are on path  
src //Source_node  
target //destination_node
```

Output:

```
maxflow// Max flow
```

FordFulkerson():

```
ResMatrix= CapacityMatrix
```

```
maxflow=0;
```

```
while(true):
```

```
    shortestPath = DFS (ResMatrix, src, target) // DFS finds shortest path
```

```
    if(shortestPath==NULL):
```

```
        break;
```

```
    else:
```

```
        cur_flow= minflow (shortest_path)
```

```
        flow+= cur_flow;
```

```
        current =t;
```

```
    END IF
```

```
While (current!=s ):
```

```
    p_vertex = adjacency_matrix(current)-> previous ;
```

```
    ResMatrix[p, current] =Res_matrix[p,current]-cur_flow
```

```
    ResMatrix [current, p] =ResMatrix[current, p]+cur_flow
```

```
    current= p;
```

```
END;
```

```
END;
```

```
return maxflow;
```

Dinic's Algorithm:

Step 1: Read the values from file and copy them into a 2D Array of size $n \times n$ where n is total number of vertices.

Step2: First Vertex that is read from file is considered as source and last vertex that is read is our sink.

Step3: Augment the flow until there is a path from source to sink.

```
While(bfs() returns false)
```

```
{
```

```
//.....
```

}

Step4: Find if more flow is possible or send flows until there are no flows traversing from source to sink. In this case, for every non-zero flow keep adding current path flow to maxflow.

Step4: In the Send Flow,

- Traverse all edges,
- Find min flow from adjacent vertex(u) to sink.,
- For a non-zero flows, update the residual graph flow.

Step6: At the end of send flow loop and bfs loop we will have Maximum Flow.

Edmond-Karp Algorithm:

Pseudocode:

Input:

capacityMatrix [n x n]: Capacity Matrix
adjMatrix [n x n]: Adjacency Matrix
src: Source
target: sink or target

Output:

maxFlow: Maximum Flow Rate

The Edmonds-Karp:

```
maxFlow = 0           // Initialize the flow to 0
residualMatrix [n x n] // The residual capacity array

while true:
    // Finding the shortest path
    minimum, augmentPath = BFS(capacityMatrix, adjMatrix, source,
                                target, residualMatrix);

    if m = 0:
        break; //if shortest path is zero

    maxFlow = maxFlow + min

    // Walk through the augmenting path
    v = target
    while v != src:
        u = P[v];
        // Reduce the residual capacity
        residualMatrix [u, v] = residualMatrix [u, v] - min
        // Increase the residual capacity of reverse edges
```

```

        residualMatrix [v, u] = residualMatrix [v ,u] + min
        v = u

    return maxFlow;

```

General Push Re-Label:

Step 1: Read the values from file and copy them into a 2D Array of size $n*n$ where n is total number of vertices.

Step2: First Vertex that is read from file is considered as source and last vertex that is read is our sink.

Step3: Once the values are read into an array keep adding the edges and weights into an capacity array.

Step4: Initialize height, flow to zero and flow of every edge to 0, Also initialize all vertices adjacent to source to capacity.

```

for (int i = 0; i < n; i++) {
    f[s][i] = cap[s][i];
    f[i][s] = -f[s][i];
    e[i] = cap[s][i];
}

```

Step5: while you have a chance to push or relabel do the loop, until excess flow do push or relabel.

While(vertex has over flow)

```

{
    if( you cannot push)
    {
        //Relabel it;
    }
}

```

Step6: In order to relabel, you have to find the adjacent vertex with min. height

While(you find min height)

```

{
    If(edge flow==edge capacity)

```

```

{
//update min height
}
}

```

Step7: At the end of excess flow loop, we will have max flow.

Proof of correctness:

Ford Fulkerson Algorithm:

Proof by induction:

Base Case: if there is only one source and one sink, max flow is trivially the capacity between source and sink.

Induction Step:

Let K be an augmenting path, where K is a simple path between source and sink.

Loop invariant:

Let AP be an Augmenting Path such that AP being an Simple Path in the residual graph.

Initialization:

For an single source and single sink, there is trivially a single path and a simple path with Max Flow

Maintenance:

At an iteration i,

Augmenting Path AP should leave from source s with no incoming edges and AP is a simple Path.

Also, AP should never go to source because if it reaches the source it would form a cycle.

Termination:

At the end of the Augmenting path loop we will have our Max Flow.

Dinic's Algorithm:

Proof by induction:

Base Case: if there is only one source and one sink, max flow is trivially the capacity between source and sink.

Induction Step: Let K be the total number of blocking ways, i.e loop terminates atmost at $(k-1)$ th iteration.

Loop Invariant:

Dinic's Algorithm ends in at most $n-1$ blocking steps (less than n)

Initialization:

For one source and one sink (where $n=2$) Algorithm ends in one step $(n-1)$. ($1 < 2$)

Maintenance:

At each iterations distance won't decrease from s to u . shortest Path SP has its length and lets say it would be L , Now Graph at $(i+1)$ iteration contains edges and back edges from iteration I which means no back edges. Now length at iteration $i+1$ is greater than i (since SP belongs to Graph at i th iteration).

Termination:

At the end, in less than $n-1$ blocking steps we would have Max Flow.

General Push Re-Label:

Proof by induction:

Base Case: if there is only one source and one sink, max flow is trivially the capacity between source and sink.

Induction Step:

Let K be pre flow v has excess (over flow) then vertex has a path to source is true.

Loop Invariant:

The pre flow at the end of the loop is the Maximum flow

Initialization:

For one source and a sink the pre flow is the capacity between source and sink which is Max Flow at end of loop

Maintenance:

At an iteration i , which is considered as final iteration, Lets say it has flow f , which means the nodes have no overflow and Lets assume its not Max Flow. Then we have to prove there a exists an augmenting path between source and sink.(a path in residual graph). Since the final labelling L must be valid, we can say $L(\text{source}) = L(\text{adj.vertex to } s) \leq L(\text{vertex next to it 'p'})$. now $p < n-1$ which contradicts $L(\text{source}) = n$.

Termination:

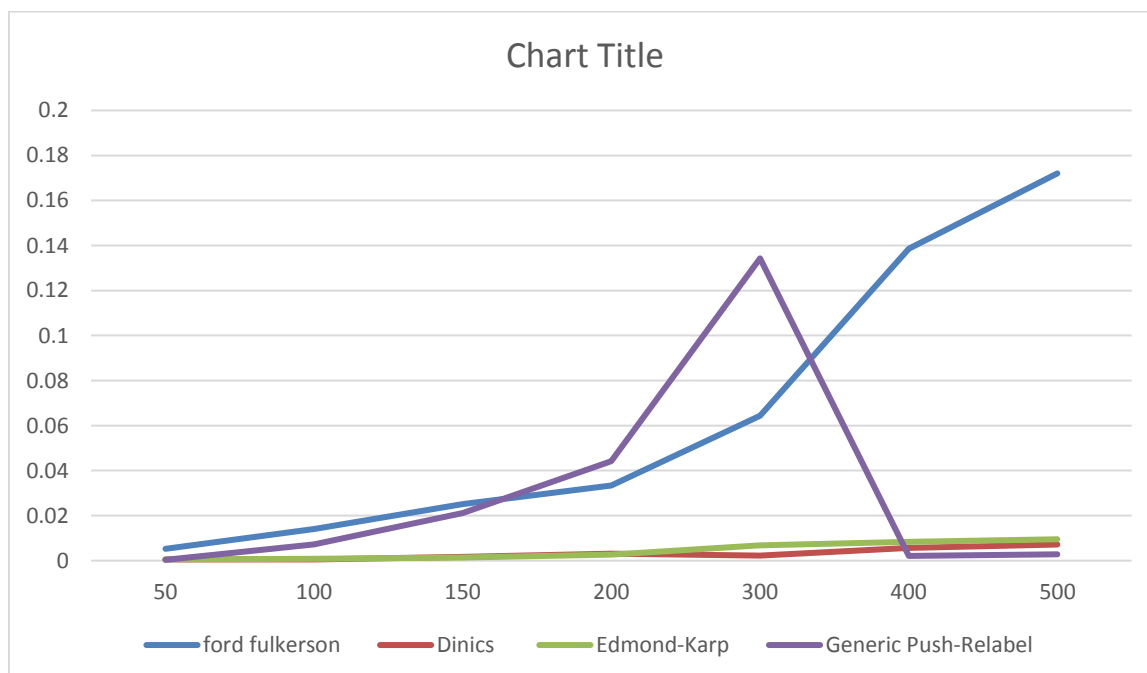
At the end of the iteration, we will have final pre flow which is our Max Flow.

Testing Plan:

The time complexities can be determined using the number of operations that the algorithm can be able to perform in given length. To get worst case analysis, we need to calculate upper bound on running time of an algorithm (i.e case that causes maximum number of operations to be executed). To know average case analysis, we need to consider all possible inputs and calculate computing time for all of the inputs & do average. For best case analysis, calculate lower bound on running time of an algorithm (i.e case that causes minimum number of operations to be executed).

The four algorithms run as follows:

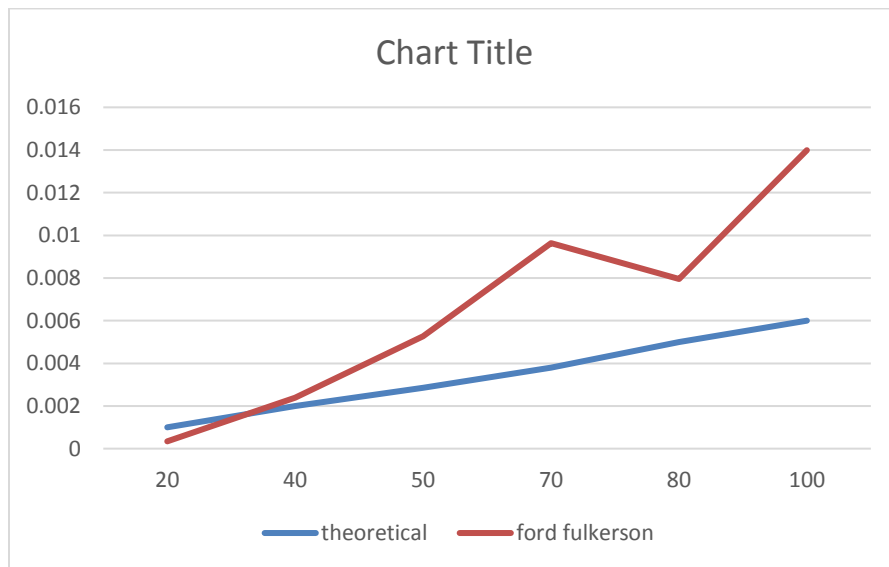
input size	ford fulkerson	Dinic's	Edmond-Karp	Generic Push-Relabel
50	0.0052624	0.0005333	0.0005921	0.000375
100	0.0139916	0.0006203	0.0008559	0.007166
150	0.0250815	0.0016433	0.0014512	0.021126
200	0.033344	0.0031024	0.0027329	0.044186
300	0.0643793	0.0022361	0.0067932	0.134284
400	0.138571	0.0056131	0.0083102	0.002111
500	0.172028	0.0071768	0.0095093	0.002843



Theoretical Behaviour:

Ford- Fulkerson Algorithm:

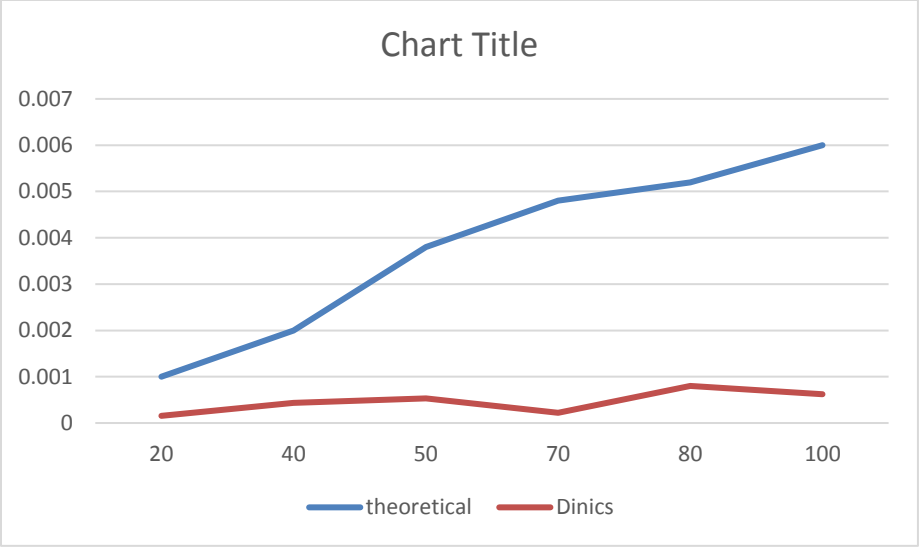
input size	theoretical	ford fulkerson
20	0.001	0.000342364
40	0.002	0.00239423
50	0.00285	0.00526237
70	0.0038	0.00963507
80	0.005	0.00795882
100	0.006	0.0139916



At $n=70$, implementation start to exhibit asymptotic complexity for Ford-Fulkerson Algorithm.

Dinic's Algorithm:

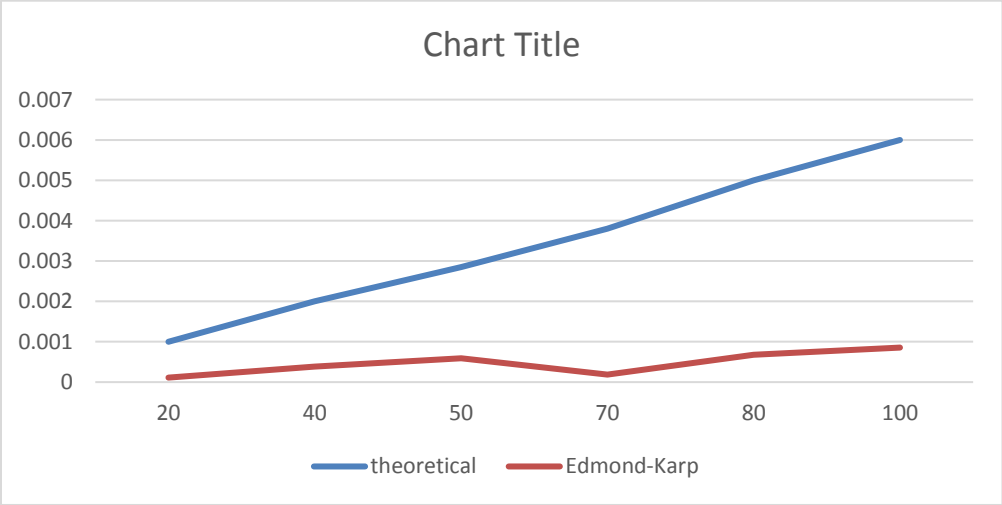
input size	theoretical	Dinic's
20	0.001	0.000154496
40	0.002	0.000432122
50	0.0038	0.000533325
70	0.0048	0.000223209
80	0.0052	0.000800343
100	0.006	0.000620299



At n0=70, implementation start to exhibit asymptotic complexity for Dinic’s algorithm.

Edmond-Karp Algorithm:

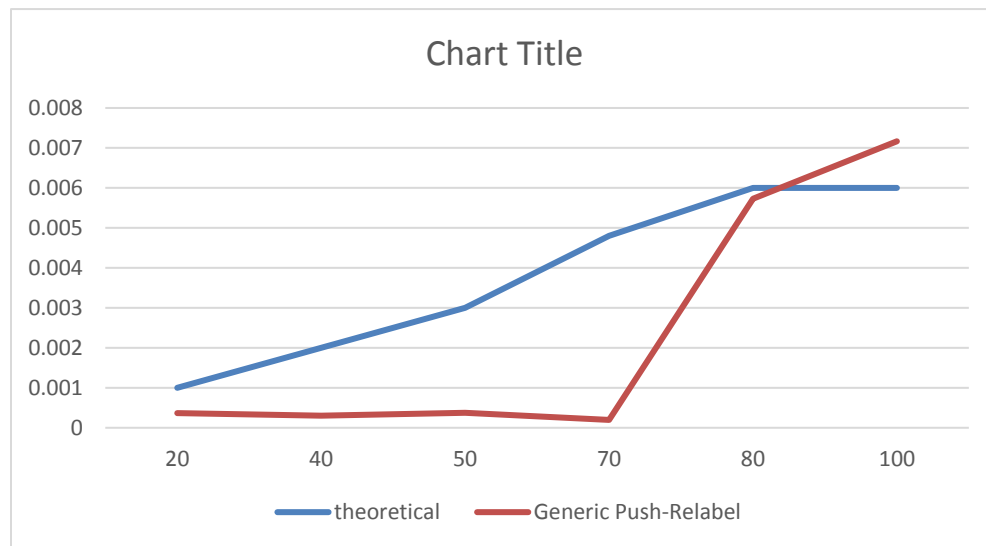
input size	theoretical	Edmond-Karp
20	0.001	0.00011894
40	0.002	0.000382138
50	0.00285	0.000592139
70	0.0038	0.000188578
80	0.005	0.000678205
100	0.006	0.000855929



At n0= 50 , implementation start to exhibit asymptotic complexity for Dinic’s Algorithm.

Generic Push Re – Label:

input size	theoretical	Generic Push-Relabel
20	0.001	0.000369664
40	0.002	0.000304209
50	0.003	0.00037476
70	0.0048	0.000196799
80	0.006	0.00573051
100	0.006	0.00716567



At $n=70$, implementation start to exhibit asymptotic complexity for Generic Push Re-label algorithm.

Observations:

From the above graphs,

- **Time Complexity** for Ford Fulkerson Algorithm is $O(E|\text{maxflow}|^*)$ where time to find a path in a residual network is $O(V+E') = O(E)$ if we either use DFS or BFS. Each iteration of the while loop thus takes $O(E)$, and in the while loop, our algorithm executes at most $|\text{maxflow}|^*$ times since the flow value increases by at least one unit in each iteration, where $|\text{maxflow}|^*$ means the maximum flow transformed in the network.
- **Time Complexity** for Dinic's Algorithm depends on total number of blocking steps and steps to find the blocking flow.
Blocking steps run from 1 to $V-1$.

Blocking flow can be reduced to $O(E \log V)$.

At Best to worst from $O(VE \log V)$ to $O(V^2 E)$.

- **Time Complexity** for Edmond-Karp Algorithm is $O(V|E|^2)$ is found by showing that each augmenting path can be found in $O(E)$ time using BFS. Any edge can become critical at most $O(V)$ times and finding the number of augmented paths i.e, no of iterations the Edmond Karp should go through is $O(V * E)$. Then overall time complexity becomes $O(V|E|^2)$.
- **Time Complexity** for General Push Re label Algorithm runs at most at $O(V^2 E)$. Best, average and worst run times are based on number of push and re label operations it performed. At Best to worst it can perform $O(V^3)$ to $O(V^2 E)$.

Justifications for observations:

Inputs that affect the algorithms:

- For Ford-Fulkerson ,Edmond-Karp and Dinic's Algorithm if the input graph is in such a way that solution is far away which means if it has to traverse until last path it consumes a lot of time. (While checking for Augmented Path).
- For General Push Relabel, if input graph contains non saturated push operations, it runs almost worst case amount of time.
- **Ford- Fulkerson Algorithm** is bounded by $O(E|\text{maxflow}|^*)$, where maxflow =Maximum flow in the graph.Finding an augmenting path requires a depth-first search of the graph, which takes $O(E)$ time. Since we can do at most $|\text{maxflow}|$ iterations, and each iteration takes $O(E+V)$ time, the worst-case run time is $O((E+V)\text{maxflow})$ which is $O(Ef)$.
- **Edmon-Karp Algorithm** is bounded by $O(V|E|^2)$,this algorithm makes some important improvement on the Ford-Fulkerson algorithm. Ford-Fulkerson simply states that the flow increases by at least 1 in every iteration. Each iteration takes $O(|E|)$ time. as we saw above. So, all Ford-Fulkerson can promise is that the maximum flow is found in $O(|E| \cdot f^*)$, where f^* is the maximum flow itself.
- Whereas, **Edmonds-Karp** removes the dependency on maximum flow for complexity, making it much better for graphs that have a large maximum flow. By

showing that Edmonds-Karp runs each iteration in $O(|E|)$ time and that there are at most $|V| \cdot |E|$ iterations, Edmonds-Karp is bounded by $O(|V| \cdot |E|^2)$.

- **Dinic's Algorithm** has from 1 to at most $V-1$ blocking steps. In each blocking step Graph can be constructed using $O(E)$ time and flow can be found in $O(VE)$ time. So time taken for this Algorithm is (time taken for blocking steps) * (time taken for bfs + time taken to find flow) i.e. $O(V-1) * O(E + VE) \sim O(V) * O(VE) \sim O(V^2E)$.
- **General Push Re-Label Algorithm** runs re label operation at most $O(V^2)$ time., It runs pushes from $O(VE)$ to $O(V^2E)$ amount of time. Therefore the overall run time is $O(V^2E)$.

Key Insights:

- We can improve the time complexities of Dinic's and push relabel Algorithms.
- Using dynamic trees in Dinic's Algorithm time taken by blocking flow can be reduced to $O(E \log V)$.
- By improving re label operations we might reduce the whole run time because it alone runs in quadratic amount of time.

Performance Comparisons:

If we run same input against four algorithms, We Can conclude that:

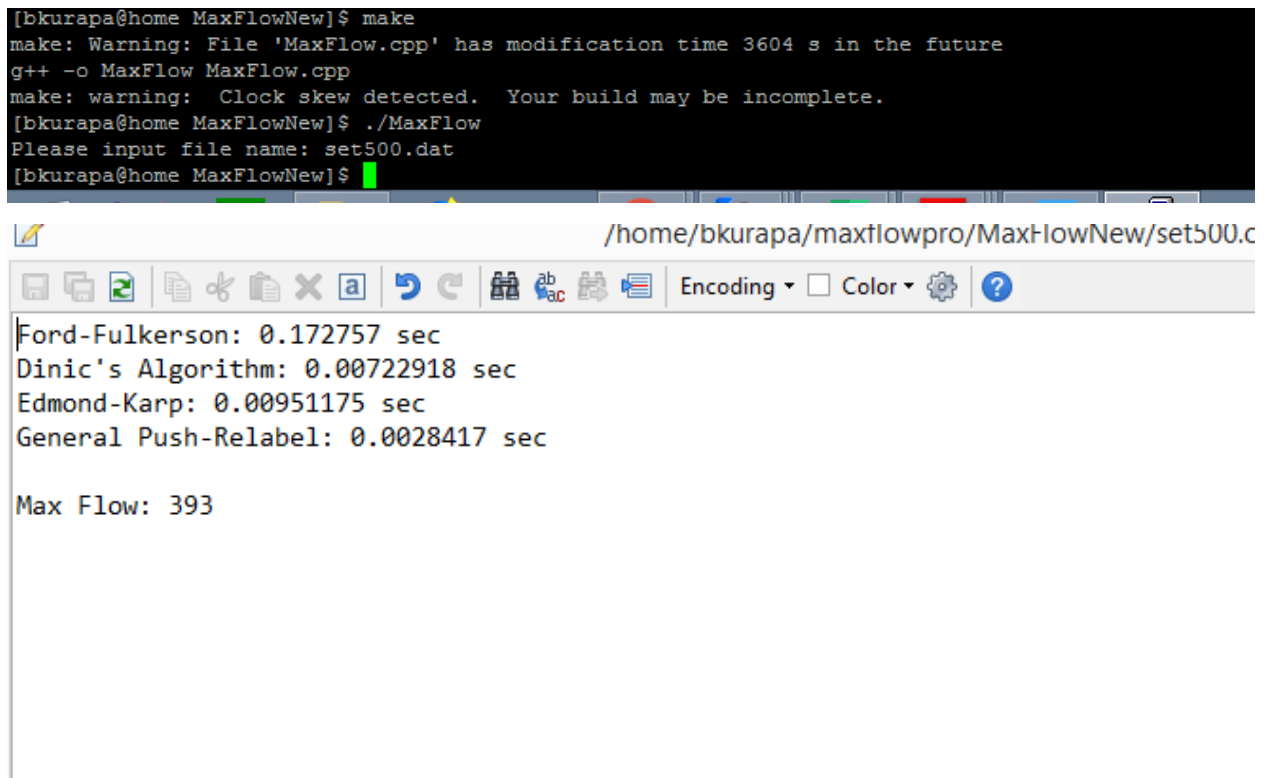
- Dinic Algorithm is working best than all algorithms.
- Edmond-Karp Algorithm is also working best but not better than Dinic's
- Ford Fulkerson Algorithm runs worst and General push re label gives a fluctuating but good run time but worst than Dinic's and Edmond algorithms

Conclusion:

In this Project, We have discussed and implemented Maximum Flow Algorithm Using Ford-Fulkerson Algorithm, Dinic's Algorithm, Edmond-Karp and General push Re-Label Algorithm. We have compared the results and found the asymptotic behavior of the algorithm. We have discussed about the time complexities and reasons for those run times and evaluated their performance.

Proof of work:

```
[bkurapa@home MaxFlowNew]$ make
make: Warning: File 'MaxFlow.cpp' has modification time 3604 s in the future
g++ -o MaxFlow MaxFlow.cpp
make: warning: Clock skew detected. Your build may be incomplete.
[bkurapa@home MaxFlowNew]$ ./MaxFlow
Please input file name: set500.dat
[bkurapa@home MaxFlowNew]$
```



```
Ford-Fulkerson: 0.172757 sec
Dinic's Algorithm: 0.00722918 sec
Edmond-Karp: 0.00951175 sec
General Push-Relabel: 0.0028417 sec

Max Flow: 393
```

The below screenshot is just to show that all the algorithms are generating the same result:

```
[bkurapa@home MaxFlowNew]$ make
make: Warning: File 'MaxFlow.cpp' has modification time 3553 s in the future
g++ -o MaxFlow MaxFlow.cpp
make: warning: Clock skew detected. Your build may be incomplete.
[bkurapa@home MaxFlowNew]$ ./MaxFlow
Please input file name: input.dat
Ford-Fulkerson Max flow:12
Dinic Max flow:12
Edmond-Karp Max flow:12
PushRelabel Max flow:12
[bkurapa@home MaxFlowNew]$ ./MaxFlow
Please input file name: set100.dat
Ford-Fulkerson Max flow:476
Dinic Max flow:476
Edmond-Karp Max flow:476
PushRelabel Max flow:476
[bkurapa@home MaxFlowNew]$
```