**Second Level Algorithms**

**Prof. Palash Dey**

**Department of Computer Science and Engineering**

**Indian Institute of Technology, Kharagpur**

**Week – 04**

**Lecture 20**

Welcome to the 20th lecture of the second-level algorithms course. In the last lecture, we finished our analysis of the Edmund-Karp algorithm, and we saw that the worst-case running time of that algorithm is big O. So, all these algorithms are based on flow augmentation or augmenting path-based algorithms. Although we have intuitively explained what we mean by flow augmentation, let us begin this lecture by formalizing that notion. So, let us begin.

Flow augmentation. Given a flow $f_1$ in a network—so given an s-t flow, a flow from source to sink— and an s-t flow $f_2$ in the corresponding So, the original network, let us call it G, and so the residual network is $G_f^1$. The augmented flow in G, denoted by $f_1$ augmentation $f_2$, is defined as follows. So, the value of this augmented flow in an edge uv is if I take an edge u, v (u to v) in E[G], then a couple of cases: if $f_2$ of uv is positive, then that means there is a forward edge u to v in the residual graph, and that edge carries some amount of flow. Then the resulting flow is $f_1(uv)+f_2(uv)$.

If there is an opposite edge, or actually we can define it together, or if there is a reverse edge. is greater than 0, then it is $f_1(uv)-f_2(uv)$. So, the first condition is there is a forward edge, but the reverse edge carries no flow. or the reverse edge does not exist; vu does not belong to $E[G_f^1]$, and in the second case, there is no U to V[H]. So if both uv and vu edges are there in the residual graph, that means if both the Then the resultant flow on the edge uv is $f_1(uv)+f_2(uv)-f_2(vu)$. This also should be vu. So, this is the formalization of flow augmentation. You can see that this flow augmentation can be done in $O(m)$ time. Not only that, if there are k edges that carry a non-zero amount of flow, if there are at most k edges that carry a non-zero amount of flow in $f_2$, then only at most 2k edges the flow value in $f_1$ changes, so this flow augmentation can be done in big O iterations. For both Ford-Fulkerson and Edmund-Karp algorithms, $f_2$ is a flow along an s to t path. So That means that there are at most n-1 edges in $G_f^1$ which can carry a positive

amount of flow in $f_2$ because any path from S to T can have at most n-1 edges. So, in both the algorithms, Ford-Fulkerson and Edmund-Karp, we can perform flow augmentation in $O(n)$ time next we will see another algorithm for computing a maximum sd flow and that algorithm is called Dinic's algorithm which uses the same augmentation technique but there the flow augmentation is not along a path And thereby, we will improve the running time of the Edmonds-Karp algorithm from $O(m^2n)$ of Edmonds-Karp to $O(mn^2)$ in the Dinic algorithm. So, let us begin with the Dinic's algorithm. So instead of performing flow augmentation along some path in the residual graph we compute what is called a blocking flow, a blocking flow in the Let us call this blocking flow F' in the residual graph and augment it with the current flow in the input graph. Okay, so what is a blocking flow? A blocking flow is a flow such that there is no s-t path where every edge has positive residual flow or residual capacity.

So, a blocking flow is a flow F such that there is no s-t path whose every edge e has positive residual capacity, that is c(e) - f(e) is greater than 0. So, let us see an example of a blocking flow.

Consider this flow network. the flow of one unit along the path S, A, B, T is a blocking flow. This F is a blocking flow. Observe that the output of the naive greedy algorithm that we have tried as a first approach for designing an algorithm for finding a maximum SD flow indeed outputs a blocking flow. Of course, a blocking flow need not be a maximum flow.

okay how much time does the naive greedy algorithm take to find a blocking flow so it can in every iteration it can perform a breadth first search to find one path from s to t and along that path send as much flow as you as it can so in every iteration at least one edge gets saturated so the time complexity of naive greedy algorithm is of m times $m^2$ because there could be at most m iterations and there are at most m edges which can get saturated. greedy algorithm makes at most m iterations since at least one edge gets saturated in every iteration.

And in every iteration we can find the s to t path by performing a breadth first search in $O(m+n)$ time. ok we will prove that the Dinic's algorithm makes at most n iterations. So, what is the runtime of naive greedy algorithm? So, the overall runtime is $O(m(m+n))$ which is for any connected graph or if there is a path from s to t and it is a connected graph then it is O of Observe that we can assume without loss of generality that every vertex is reachable from s and from that vertex t is also reachable. That means for every

vertex there is a path from s to t via that vertex. why we can assume that without loss of generality because if there exist any such vertex for which there is no s to t path s to that vertex path We can assume without loss of generality that m is greater than equal to n or $n-1$.

So, every iteration of Linux algorithm takes $O(m^2)$ time and there are n iterations at most that we will show. So, the overall runtime of Dinic's algorithm using the naive greedy algorithm to compute a blocking flow in every iteration is $O(m^2)$ time and there are n iterations at most so the runtime is $O(m^2 n)$ but this is not the improvement from Edmund-Karp algorithm the runtime of this is the same as the Edmund-Karp algorithm so what we will do we will use more efficient algorithm for computing a blocking flow in every iteration in $O(mn)$ time thereby improving this running time to $O(mn^2)$. We will use more efficient big O of a main time algorithm to compute a blocking flow in every iteration thereby making the overall runtime of Dinic's algorithm $O(mn^2)$. So, we will see these two things in next lectures. So, let us stop here. Thank you.