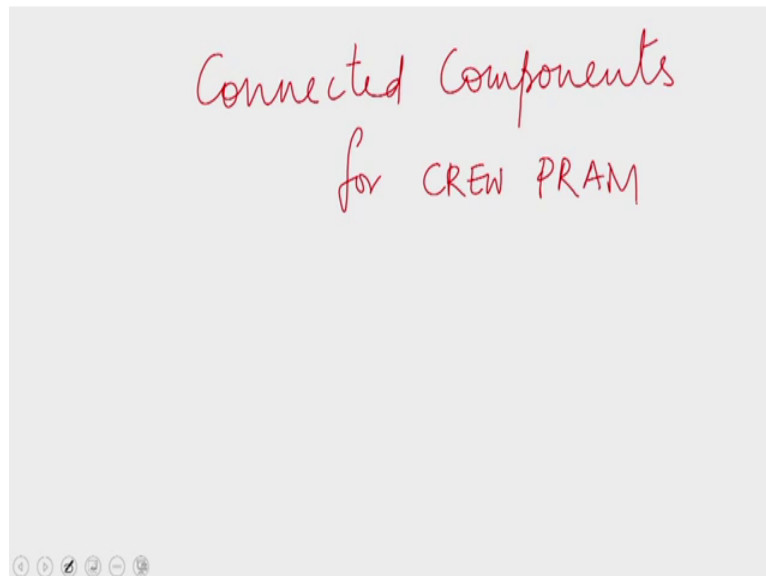
**Parallel Algorithms**
**Prof. Sajith Gopalan**
**Department of Computer Science & Engineering**
**Indian Institute of Technology, Guwahati**

**Lecture – 23**
**Connected Components, Vertex Colouring**

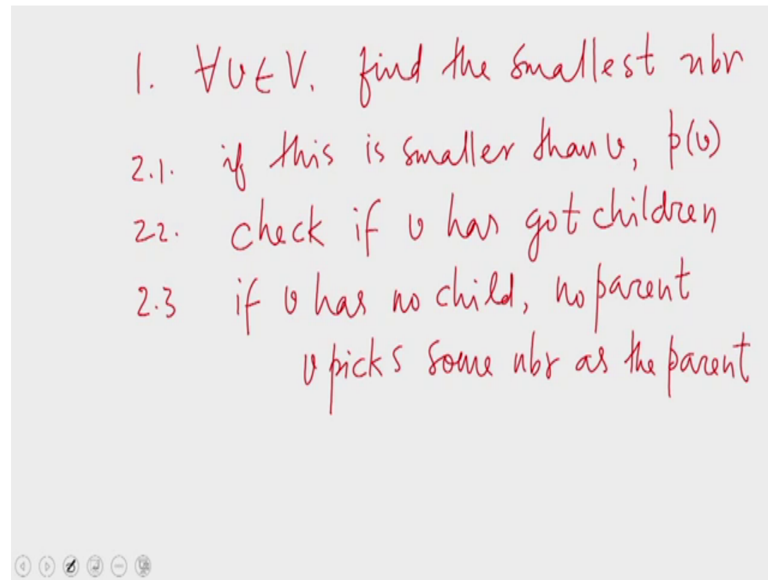
Welcome to the 23rd lecture of the NPTEL MOOC on Parallel Algorithms, in the previous lecture we have been discussing the Connected Components problem for the CREW PRAM.

(Refer Slide Time: 00:41)




Let us do a recap of what we have seen.

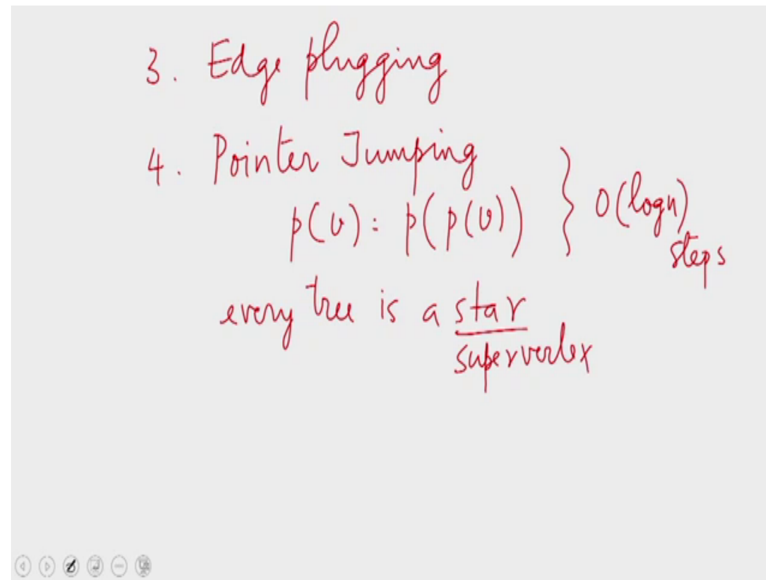
(Refer Slide Time: 00:59)




In the first step of the algorithm for every vertex of the graph we find the smallest neighbor. We do we do this using pointer jumping over the adjacency list of every vertex V. Therefore, this can be done in order of log n time. Then in step 2 1 if this that is the smallest neighbor is smaller than v then we set it as the parent of v.

In step 2 2 we check if v has got children now, that is if somebody has chosen v as the parent in step 2 1, that is checked in step 2 2. In step 2 3 if v has no child and no parent then v picks some neighbor as the parent. So, by now every vertex has got either a parent or a child and the sub graph defined by these parent pointers will be a forest that is each component of it is a tree.

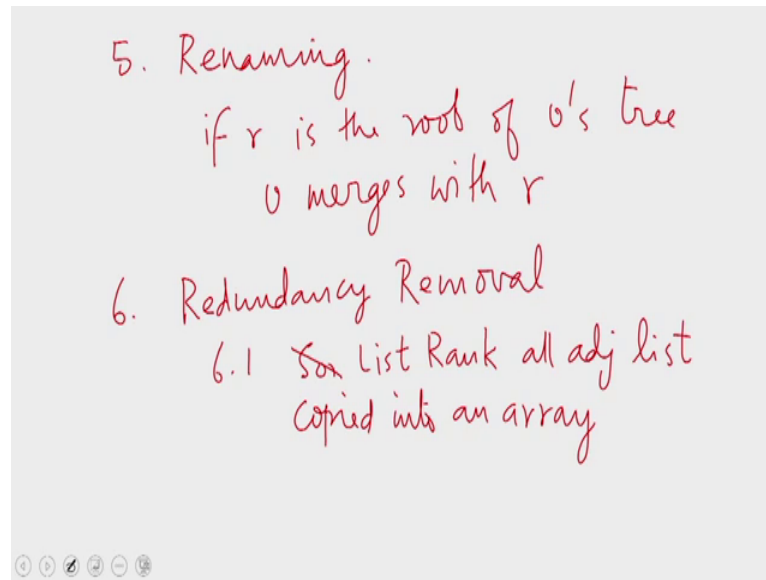
(Refer Slide Time: 02:45)




Then in step 3 we do edge plugging, this involves sending the adjacency list of every vertex into the adjacency list of its parent. The entire adjacency list of the vertex is taken and is plugged into the adjacency list to the parent while the parent does the same thing. Therefore or the adjacency lists of all the vertices in the tree will now be reachable from the root node. So, the root node now has all the adjacency list of all the vertices belonging to its tree.

In step 4 we did pointer jumping that is every vertex defines redefines its parent, it adopts its grandparent as its parent. So, this is done repeatedly until no further change happens. So, this is done for order log n steps. So, by now every tree has become a star, the star is what we call a super vertex.

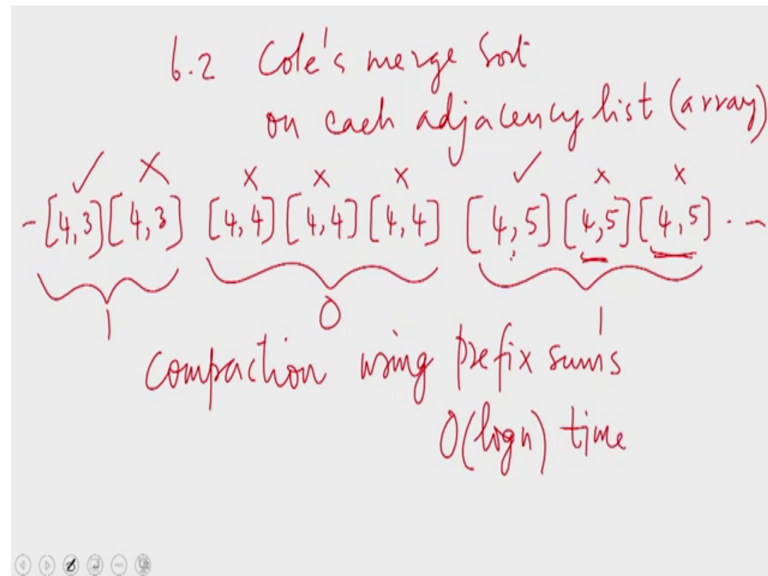
(Refer Slide Time: 04:12)




Then in step 5 we did a renaming. So, if r is the root of vertex v's tree v merges with r; that is v is now losing its identity, v is going to be a part of the super vertex that is named r. Therefore, every edge which is originating at v and going somewhere will have to be renamed appropriately. So, this renaming is done by informing every element within v's adjacency list that its new name is r and then these elements these adjacency list entries will inform their twins of the same. Therefore, the presence of v in the twins also will be replaced with r's therefore, now at this point in time all the adjacency list entries in the whole graph is renamed appropriately.

So, we will find only the names of super vertices in all these adjacency list entries and all of them are present with the root of the tree, the root of the respect the respective trees. And in step 6 this is what we were discussing at the end of the last lecture; we have redundancy removal. For this in step 6 1 we list ranked all adjacency list, the adjacency lists which are now with the root of a tree that is a super vertices will be all list ranked and then they are copied into an array. So, the elements will be appearing in this edge array in the same order in which they appear in the adjacency list.

(Refer Slide Time: 06:15)



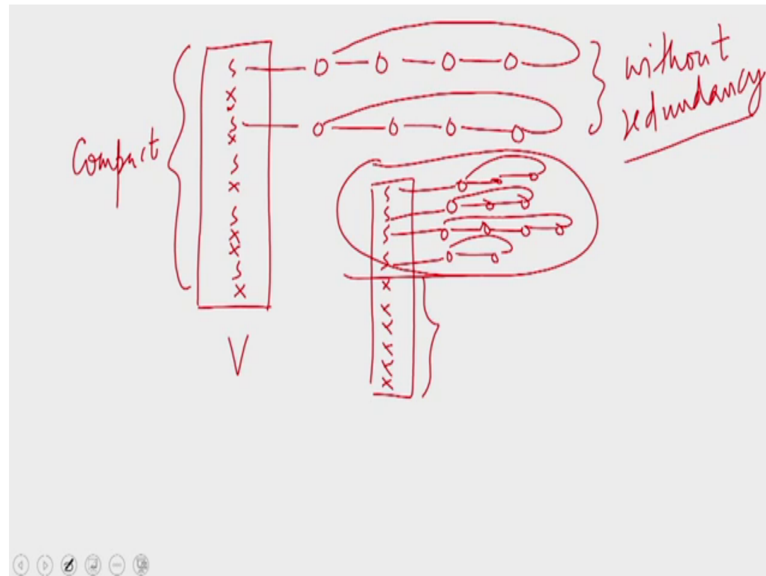

Then in 6 2 we apply Cole's merge sort on each adjacency list which is now in the form of an array. Therefore, we are able to apply Cole's merge sort on it. So now, we will have all the copies of the same element appearing together. For example, if we have a self loops of the form 4 4 we will have many of them appearing together; let us say we have entries of this sort with the super vertex 4. So, super vertex 4 has multiple entries of the form 4 3 where, 3 is also a super vertex. Then there are multiple entries of the form 4 4 which are self-loops and then there are multiple entries of the form 4 5 which are again edges into the super vertex 5.

So, these are all redundant edges, we need to keep only 1 copy out of this set and out of this set we need not keep anything; from the multiple copies of 4 3 again we only need to keep 1 copy. How exactly do we manage it? For this what we have to remember is that these elements are now available in an array and when we know which elements are to be picked out of an array, we have a way of getting them together which is compaction. So, first we have to mark all the elements that are necessary; all self loops can cross themselves out. Out of multiple entries of the same value we pick out the first one and cross out the remaining that is the processor which is sitting on the second 4 5 looks to the left and finds that entry is also a 4 5.

Therefore, this 4 5 is a duplicate and it is not necessary so, that processor marks itself. Similarly, this also marks itself, but the process sitting on this 4 5 when it looks to the

left finds that it is a 4 4 to the left therefore, this is the first 4 5. Therefore, this is chosen for retention. Similarly, the first 4 3 is also chosen for retention and the second 4 3 is crossed out. So, in this way we can mark all the elements that are necessary to be that are to be retained into the next step. So, after marking all of them we will do a compaction using prefix sums. This is an algorithm that we have seen before, this also can be done in order of log n time.

(Refer Slide Time: 09:34)

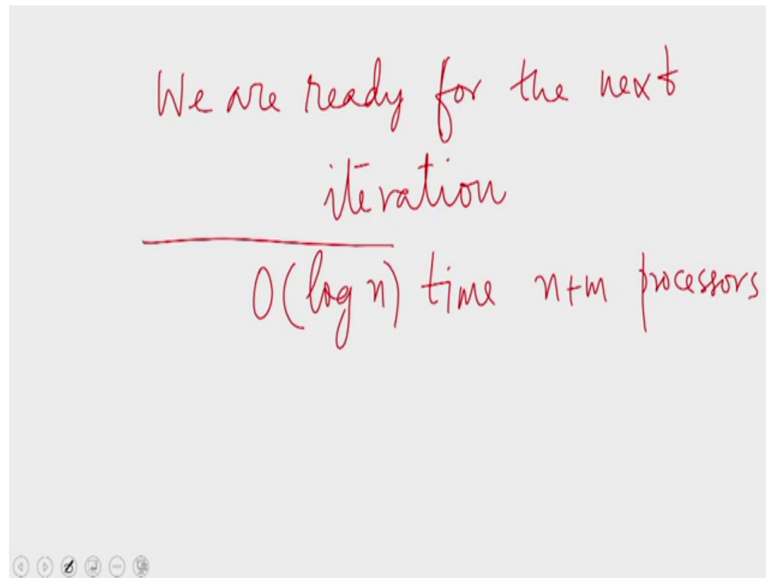



So, now let us see how the graph looks like, now we have the super vertices in the array V we have some vertices marked out as super vertices. The remaining are all crossed out, they have all joined some super vertex or the other. So, we have some super vertices and some non-super vertices and every adjacency list entry has been transferred to the super vertices. So, these super vertices have adjacency list entries. The crossed out elements are without adjacency list entries. Now, what we have ensured is that these adjacency lists are without redundancy.

Now, in this array of vertices we can again apply compaction to get all super vertices consecutive and the crossed out vertices afterwards. So, this array with all of them having their corresponding adjacency list is what we are going to start the next iteration with. So, this is how the modified graph looks like. So, we can ignore all these vertices now and the next iteration will continue with only this part of the adjacency list representation. So, every edge that is remaining here is between super vertices and they

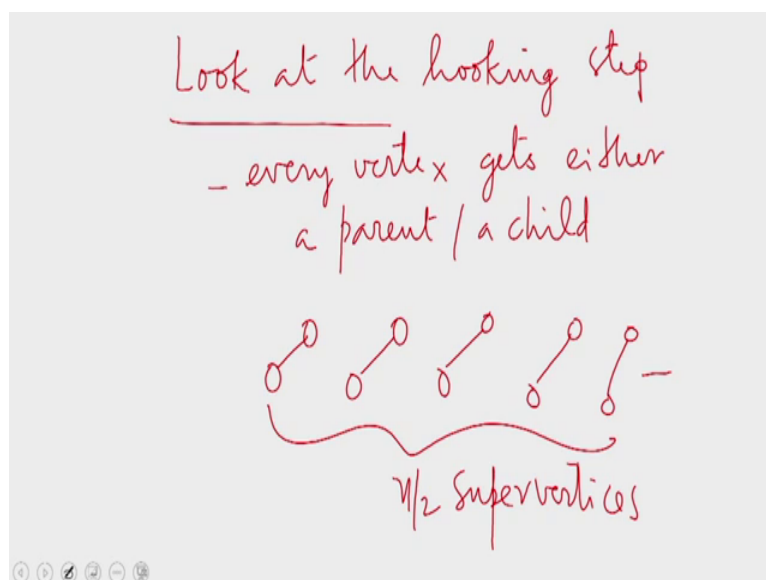
have been named appropriately and we have only the super vertices together in a compacted form. Now we have got the graph in exactly the same form that we had at the beginning of the iteration therefore, we are ready for the second iteration.

(Refer Slide Time: 11:28)



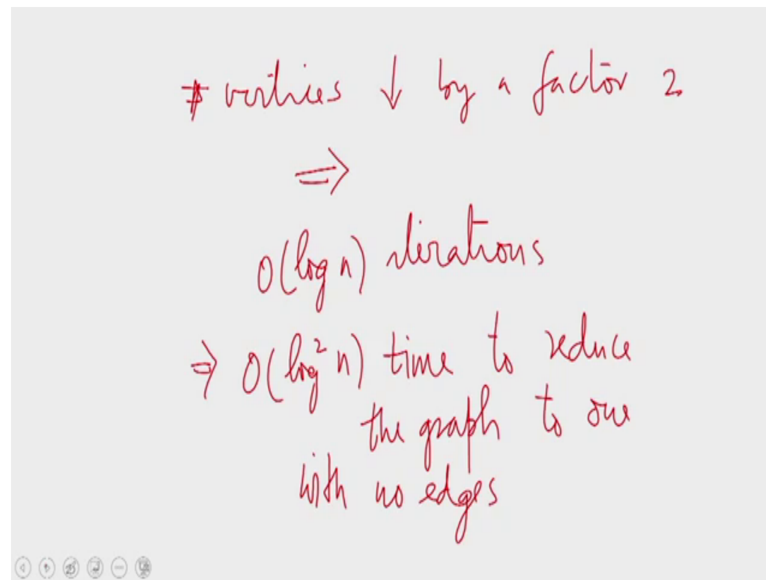

So, as you can see that this required order of log n time with n plus m processors on a CREW PRAM. Now how many iterations are required? We get a clue when we look at the trees.

(Refer Slide Time: 12:05)



Look at the hooking step, in the original graph we ensure that every vertex gets either a parent or a child; many of them of course, will get both. Therefore, what we ensure is that every tree that forms has at least 2 vertices. This is the worst that can happen, vertices is pairing off.

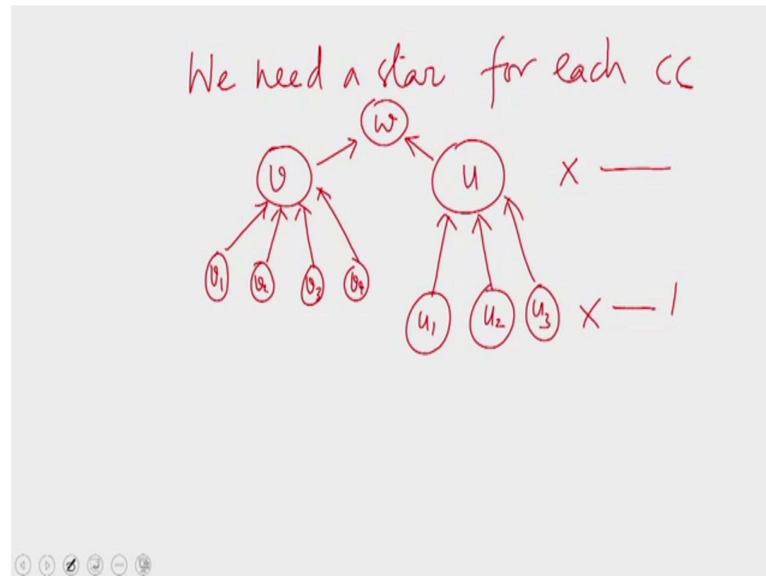
(Refer Slide Time: 13:02)




If this is the case when we have n by 2 super vertices, this is the worst that can happen to us which means the number of vertices reduces by a factor of 2. That implies that we require order of log n iterations, with each iteration taking order log n time this implies order of log square n time to reduce the graph to one that contains a number of isolated super vertices.

So, we have several super vertices, but no edges which means there is no further reduction possible. At this point we have one vertex corresponding to every connected component. Are we done? Not quite finally, for the problem of connected components to be completely solved, what we require is that for each connected component we need a star, we do not have a star yet.

(Refer Slide Time: 14:16)

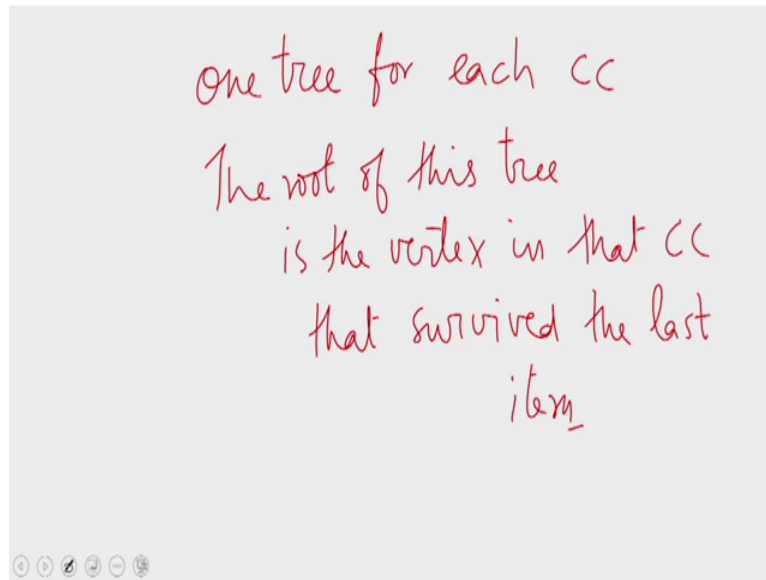



For example in the first iteration when several vertices join their super vertices and bow out of the algorithm, they all end up pointing to their chosen super vertex. So, v is the super vertex which survives into the second iteration whereas, v 1 v 2 v 3 v 4 all going out. Similarly, you have another vertex u that is surviving into the second iteration as a super vertex; several vertices have joined u. Now, suppose v and u belong to the same component and they join hands to form a tree in the second iteration; in that case it could be that one of them or some other vertex belonging to their component survives into the next iteration.

So, I am considering the case where neither v nor u survives into the third iteration, but some w survives where, w belongs to the same component as v and u. These vertices had bowed out in iteration 1 therefore, they did not participate in the second iteration. Therefore, at the end of the algorithm they continued to point to u and v, but u and v participated in the second iteration and they bow out in the second iteration.

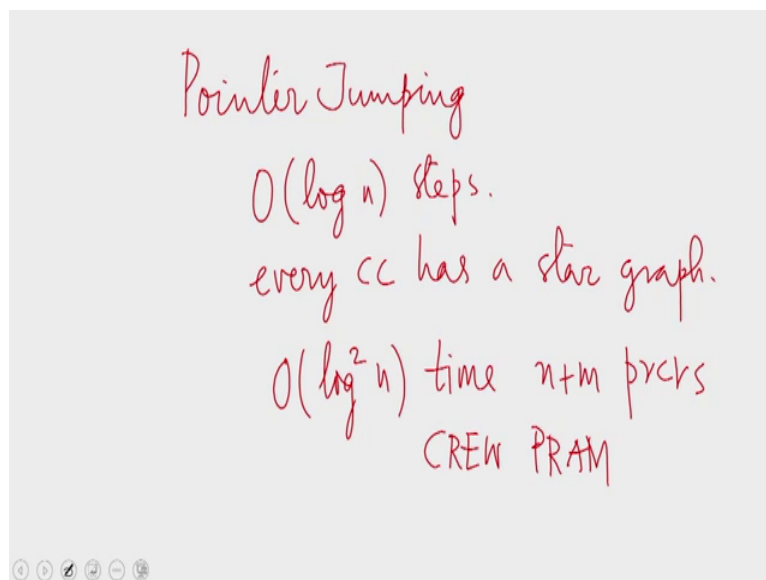
Therefore, at the end of the algorithm they end up pointing to w and w at the end of the third iteration might point to some other vertex and so on. Therefore, what we find us that when the algorithm terminates we have one single tree for each connected component.

(Refer Slide Time: 16:05)




The root of this tree is the vertex in that connected component that survived the last iteration. Now, what do we want? We want to ensure the every component has one single star graph. How do we ensure this now?

(Refer Slide Time: 16:51)

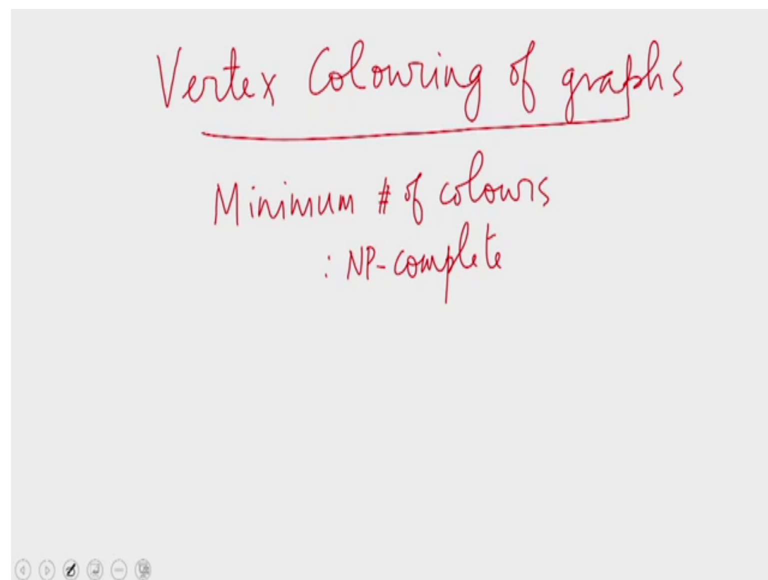



It is a simple solution; pointer jumping, do pointer jumping for order log n steps. We ensure that every CC has a star graph. So now, the algorithm has run in order of log squared n time using n plus m processors on a CREW PRAM. You can obtain an analog result for the EREW PRAM as well. And later on this solution has been improved to an

order log n time algorithm again on a EREW PRAM, but those algorithms are much more involved and are outside the scope of this course.

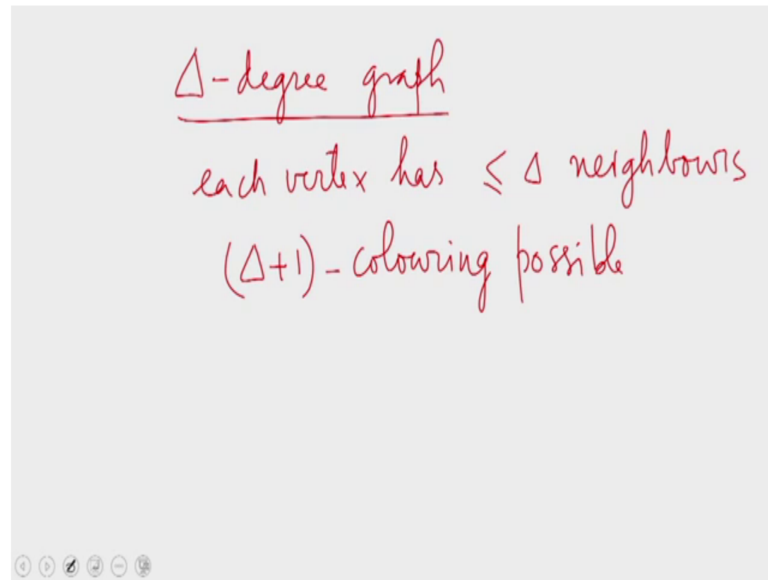
Therefore, we will not be doing into them. So, for our purpose now we can conclude this, the connected components of the graph can be found in order of log square n time using n plus m processes on a CREW PRAM.

(Refer Slide Time: 18:16)



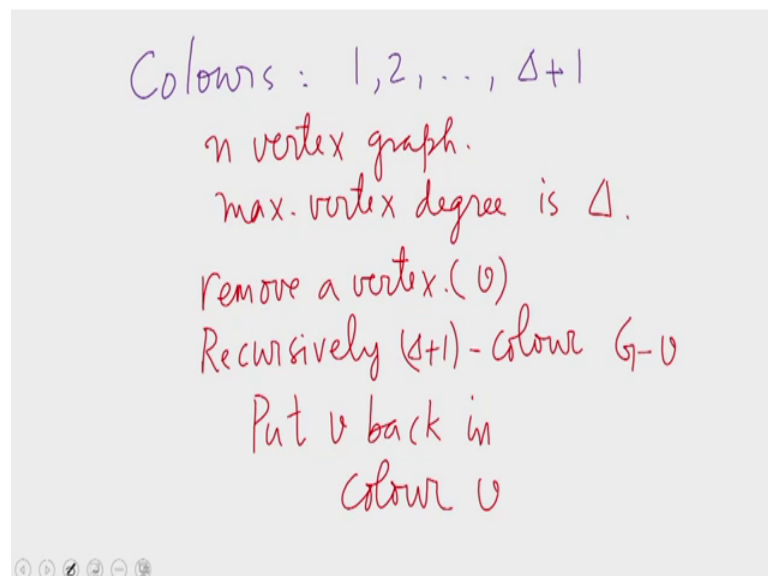

Now, we will take up another problem, this is the problem of vertex coloring of graphs. There are various use of this problem of course, the most restrictive of the problems is very hard to solve. For example, if you want to vertex color a graph with a minimum number of colors then the problem is NP complete. So, we do not expect to find an efficient algorithm for vertex coloring graphs with the minimum possible number of colors. We will use far more number of colors and color the graph more efficiently.

(Refer Slide Time: 19:12)




For example let us say we are given a delta degree graph, what we mean by a delta degree graph is one where each vertex has at most delta neighbours; that is a vertex degree of each node is at most delta such a graph is called a delta degree graph. It can be easily shown that such a graph can be vertex colored with the delta plus 1 colors. So, let us see an easy sequential algorithm for this problem.

(Refer Slide Time: 19:59)

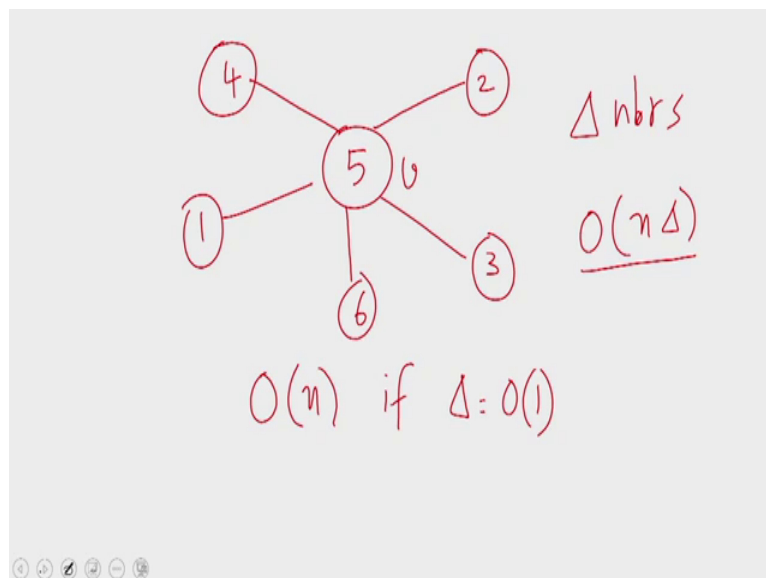



Let us say the colors are 1 2 delta plus 1 and we have an n vertex graph. The maximum vertex degree is delta; the sequential algorithm for this problem is very easy; repeatedly

you remove a vertex. And then suppose the vertex that we removes v then recursively delta plus 1 color the remaining graph G minus v. G minus v is the graph that we obtained by removing vertex v along with all its incident edges.

So, this is a graph with one less number of vertices, by removing a vertex you cannot of course, increase the degree of any vertex. So, this is also a delta degree graph. So, inductively we assume that this can be delta plus 1 color. So, G minus v is delta plus 1 colored and then we put v back into the graph.

(Refer Slide Time: 21:31)




But then when we reconstruct G we find that v is the only uncolored vertex; all its neighbours have already been colored, all its neighbours have already been colored. But, then v has at most delta neighbours each of these neighbours in the worst case can occupy one color of course, in it usually its much better because many neighbours could share a color. In which case v could have multiple free colors. But, in any case we can guarantee that vertex v has at least one free color; this is because there are delta plus 1 colors available to us in total and v has at most delta neighbours.

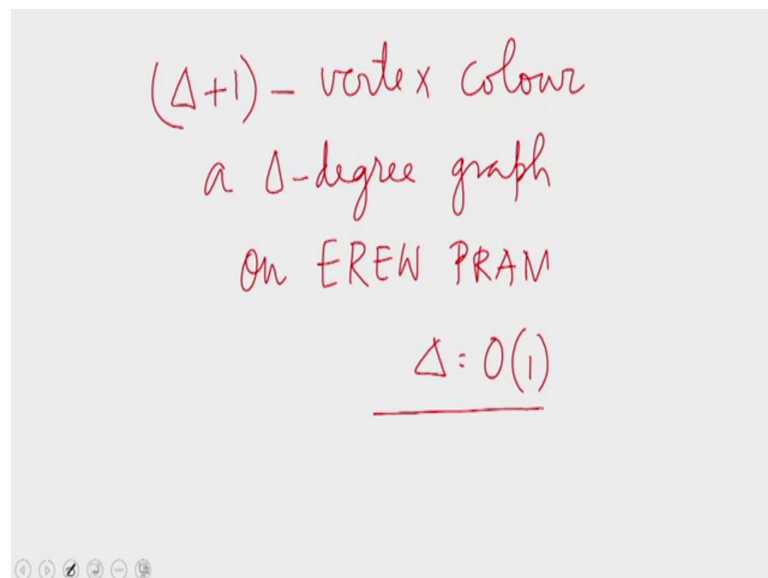
So, there is at least one free color. So, if delta happens to be 5 in this case we have used 6 colors and when we look around in the neighborhood of v we find that one color is missing which is color 5. So, v can be given color 5. So, that is what our algorithm is; we remove a vertex the rest of the graph is still delta degree graph. It can be recursively delta plus 1 color you put v back in and then color v. So, this is the inductive algorithm,

the basis is when all vertices are removed except for one. So, there is only one vertex remaining and when there is only one vertex remaining we can give it any color that we want; out of the delta plus 1 colors, we can give it any color that we want.

So, this algorithm all this does is to remove the vertices one by one and then put them back in the reverse order. So, the time taken by this algorithm is order of n if delta is order 1, if delta is bounded then the algorithm runs in order of n time. Of course, in terms of delta you can say it takes order of n delta time that is because, when a vertex is removed and is put back in it has to scan its neighborhood.

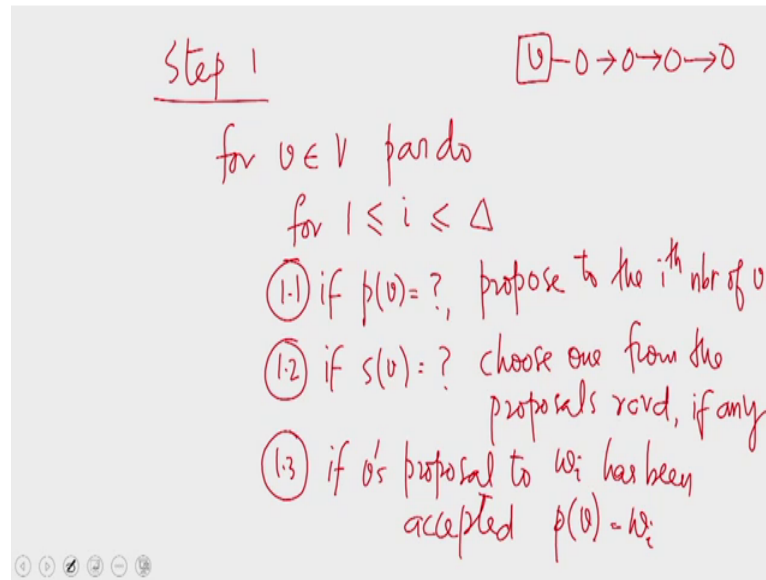
It can have an adjacency list of size delta it has to go through this adjacency list to find the smallest color which is not in its neighborhood, this might take order delta time. So, on the whole the algorithm runs in order of n delta time, if delta is order 1 this is order n. So, let us look at a parallel algorithm which does the which solves this problem for boundary degree graphs.

(Refer Slide Time: 24:15)




Let us say we want to delta plus 1 vertex color, delta degree graph on an EREW PRAM we assume that delta is order 1.

(Refer Slide Time: 24:49)




So, the algorithm proceeds like this; in step 1 for each V of the graph we do the following in parallel. There are delta iterations for each vertex, for each vertex we attempt to define a predecessor as well as a successor. If the predecessor of vertex v is undefined then vertex v proposes to the i th neighbour of v. What we assume is that vertex v has an adjacency list, the first element in the adjacency list is its first neighbour and the second element it is the second neighbour and so on. So, in the ith iteration what vertex v does is to propose to its i th neighbour.

So, it is asking its i th neighbour to be its predecessor. So, the wave vertex v is proposing to its i th neighbour vertex v might be the i th neighbour of some other vertex. So, vertex v might get some proposal. So, if the successor of v has not been defined yet, it chooses one from the proposals received if any. Of course, it is not necessary for a vertex to receive proposals; in case it does not receive a proposal there is nothing to choose from. It will not be able to choose the successor in this step; let us say if v's proposal to its i th neighbour w i has been accepted then a point w i as a parent.

So, once again what is happening is this we are attempting to define a predecessor as well as a successor for every single vertex. For every vertex we have delta iterations, if the predecessor of vertex v is not defined yet then it will propose to i th neighbour of v, v will propose to its i th neighbour. So, this is happening synchronously so, every vertex is

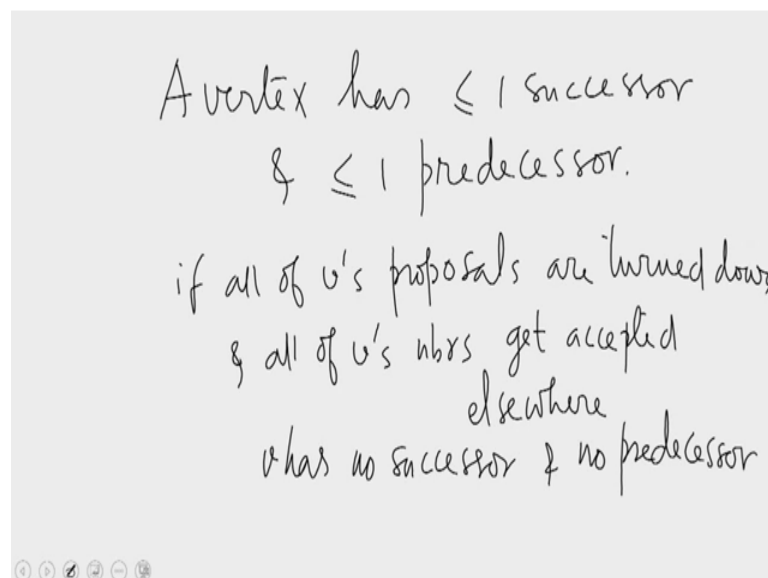
proposing to its neighbours in this one step. So, this is let us say step 1 1. So, in step 1 1 every vertex is proposing to its i'th neighbour.

In step 1 2 of the i th iteration the vertices are mulling over the proposals it received in step 1 1; its if for a vertex v if its successor is not defined yet it chooses one from the proposals that it received in step 1 1. And then in step 1 3 it goes back to the proposals it made, if one of them has been accepted, if it has been accepted it has made exactly one proposal in this step, if it has been accepted by w i which happens to be the i th neighbour of v.

Then p of v is said to w i, it made the proposal only because its predecessor was not defined. So, once the predecessor is defined it will not be making any more proposal in the coming steps. So, this is what step 1 involves: a vertex makes multiple proposals until it gets accepted as a successor. So, a vertex is appealing to its neighbours to take it on as a successor. So, if somebody agrees then that vertex will be chosen as v's predecessor.

Similarly, v receives multiple proposals if it receives a proposal it will choose one among them as its successor and once the successor is chosen, the successor will not be redefined. So, vertex will get at most one successor and at most one predecessor.

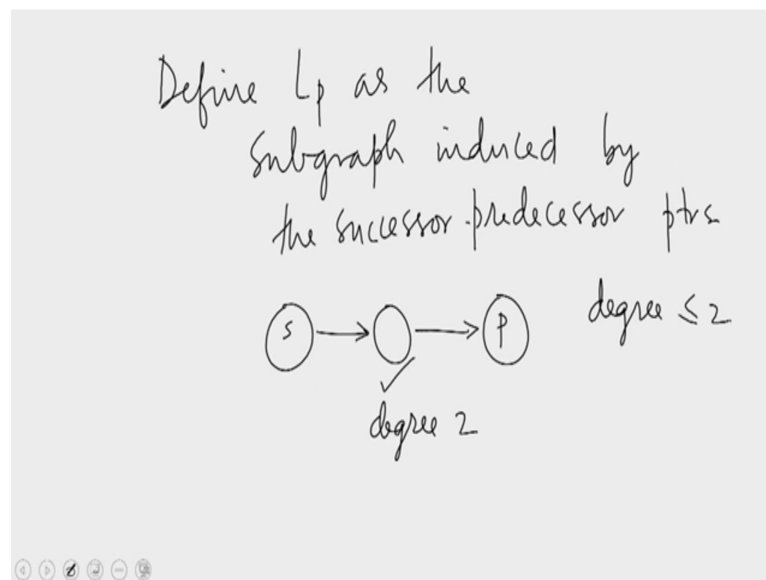
(Refer Slide Time: 29:14)



It is possible that a vertex receives neither a successor nor a predecessor in this step. When does that happen? If all of v's proposals are turned down, this can happen because in the first step v appeals to its first neighbour. It is turned down because, this neighbour has chosen somebody else as its successor, v has a appeal to the second neighbour w 2 in the second step. Again this was turned down because, w 2 turned elsewhere, likewise it has gone through all its neighbours and all of them rejected the proposals of v. In that case v becomes unable to define a predecessor. So, v might end up without a predecessor.

Similarly, v might end up without the successor as well that happens when all of v's neighbours get accepted elsewhere, that is in the adjacency list of these neighbours v happens to be rather way down and all the neighbours of v get accepted before they come to v. For example, if v happens to be the tenth neighbour of all its neighbours then and all of them get accepted let us say by a ninth round of proposals then v will never get to choose a successor. So, it is possible that v has neither a predecessor nor a successor. So, in this case v has no successor and no predecessor.

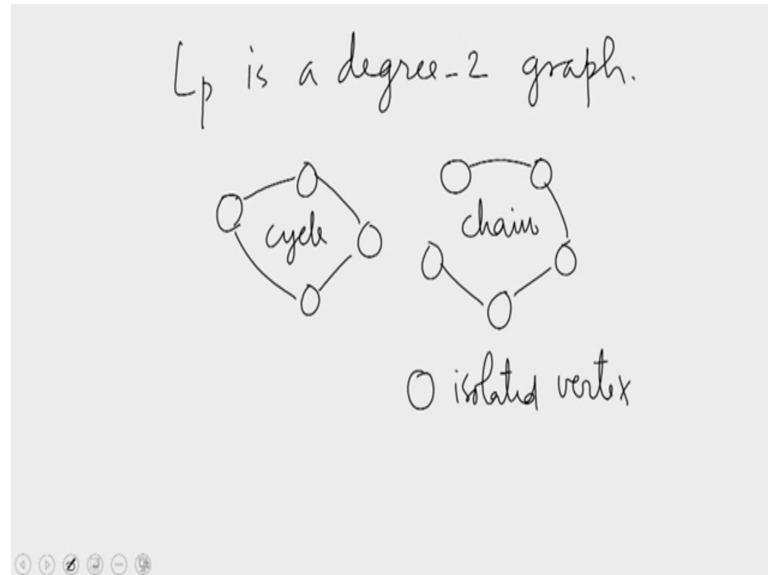
(Refer Slide Time: 31:33)




Now, let me define L p as the sub graph induced by the successor predecessor pointers. So, here each vertex gets one successor and one predecessor at the most, a vertex gets at most one successor and at most one predecessor. So, if a vertex v happens to get a successor as well as a predecessor then its vertex degree is 2. If it either misses a successor or misses a predecessor then its degree is 1, if it is isolated that if it is neither

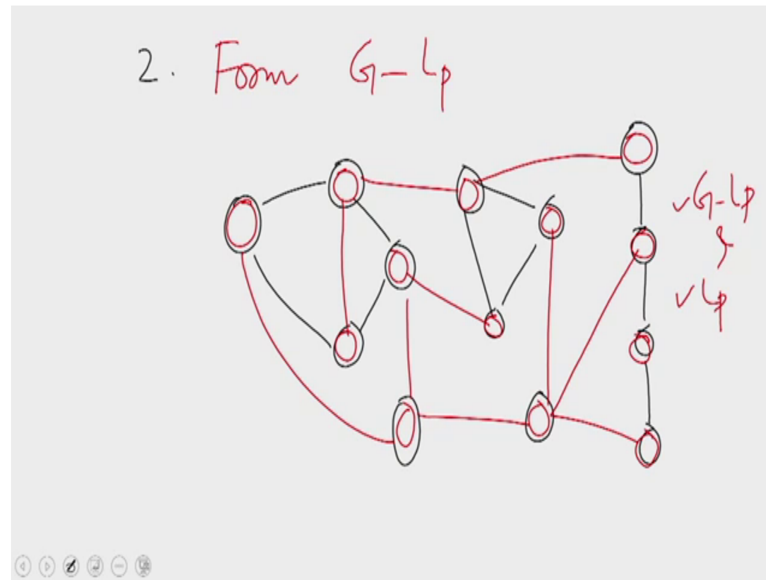
with a successor nor with the predecessor then its degree is 0. Therefore, in this graph L p the vertex degree of any node is at most 2.

(Refer Slide Time: 32:58)




So, L p is a degree 2 graph, the vertex degree of any node in L p is 0 1 or 2 that is why we call L p a degree 2 graph. So, what would L p look like? A graph in which every vertex has a degree of at most 2 will have this form. If in a component every vertex has a degree of exactly 2 then it will have to be a cycle. If a component has degree 1 vertices along with some degree 2 vertices then that will necessarily have to be a degree a chain. So, this is a cycle, this is a chain if a vertex has no degree it is an isolated vertex. So, L p is a graph of this form with cycles and chains and isolated vertices.

(Refer Slide Time: 34:06)



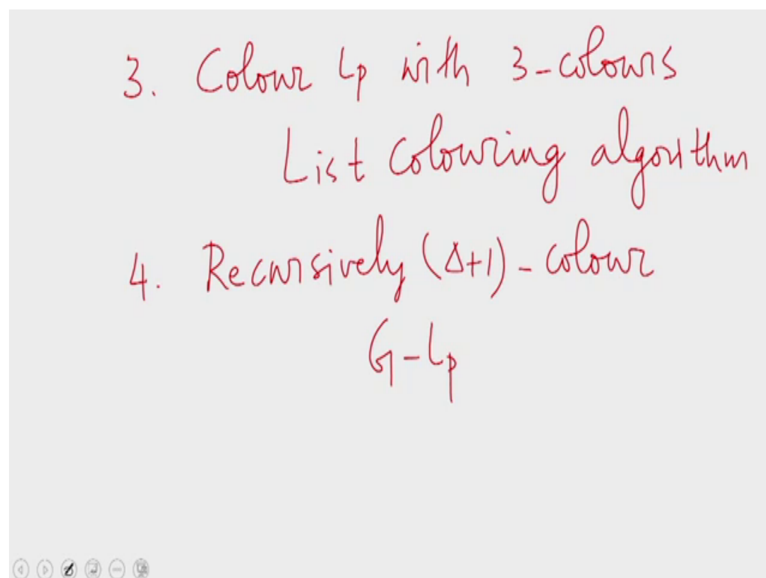

Now, what we do is this in step 2 we form G minus L p, G is our original graph and L p is the graph that has been formed just now and then we form G minus L p. So, once again let us consider L p, L p consists of cycles and chains and isolated vertices; so, this is what L p is. So, these are all edges belonging to G. So, some edges of G have been picked out to be in L p, these edges define the sub graph L p. So, L p consists of chains and cycles; if you add the remaining edges we will get G minus L p.

So, G minus L p has the same set of vertices, but a different set of edges. To signify that it has the same set of vertices, I am superimposing the 2 vertices the vertex set of G as well as the vertex set of L p are identical. Now, the edges of G minus L p I will draw in red, these could be the edges of G minus L p. The original graph G could be like this, here the black edges are the edges of L p and the red edges are the edges of G minus L p; when you put them together we have the original graph G.

(Refer Slide Time: 35:48)

So, both L p and G minus L p have the same vertex sets; G minus L p and L p have the same vertex sets, but disjoint edge sets. L p is a 2 minus degree graph, G minus L p is a graph with delta plus 1 degree again; that is because some of the vertices of G minus L p could be isolated in L p, that is if a vertex is isolated in L p then in G minus L p it has not lost any degree from G. Therefore, it retains its vertex degree of delta plus 1.

(Refer Slide Time: 36:40)



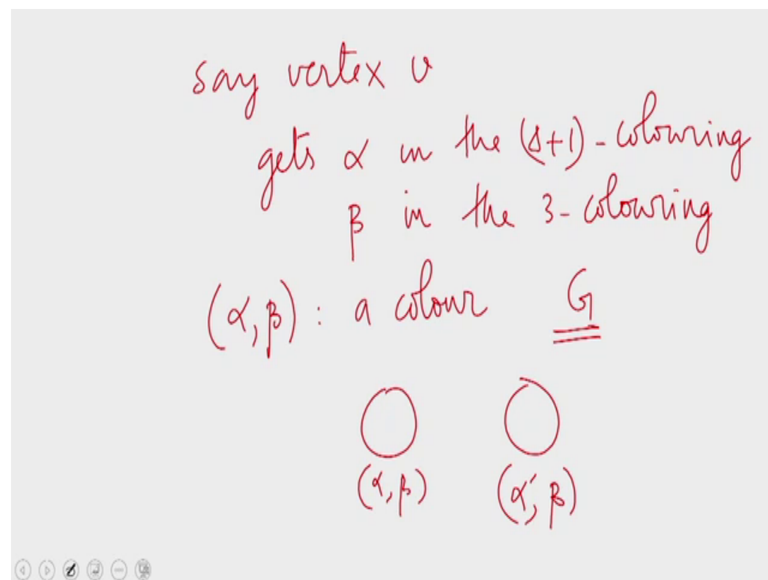

So, here what we do is to in step 3 we color L p with 3 colors using the list coloring algorithm. And let us say we recursively delta plus 1 color G minus L p; G minus L p is a

delta plus 1 color delta degree graph, but it has fewer number of edges. So, it should be possible to delta plus 1 color it from the sequential algorithm we know that it is indeed possible to delta plus 1 color it. So, we recursively delta plus 1 color it; we shall see how far the recursion will go.

First let me describe the algorithm then we will do an analysis and see how far the recursion will go. So, we have recursively delta plus 1 colored G minus L p and we have colored L p. So, when we put G minus L p and L p together we get the original graph, but then what we find is that the graph has the vertices of the graph has got 2 colors each.

One from the coloring of G minus L p and one from the coloring of L p. G minus L p has been delta plus 1 colored and L p has been 3 color, but the 2 graphs has exactly the same set of vertices. Therefore, these vertices are going to get 2 colors each; one from the coloring of the delta plus 1 coloring of G minus L p and the other from a 3 coloring of L p.

(Refer Slide Time: 38:35)



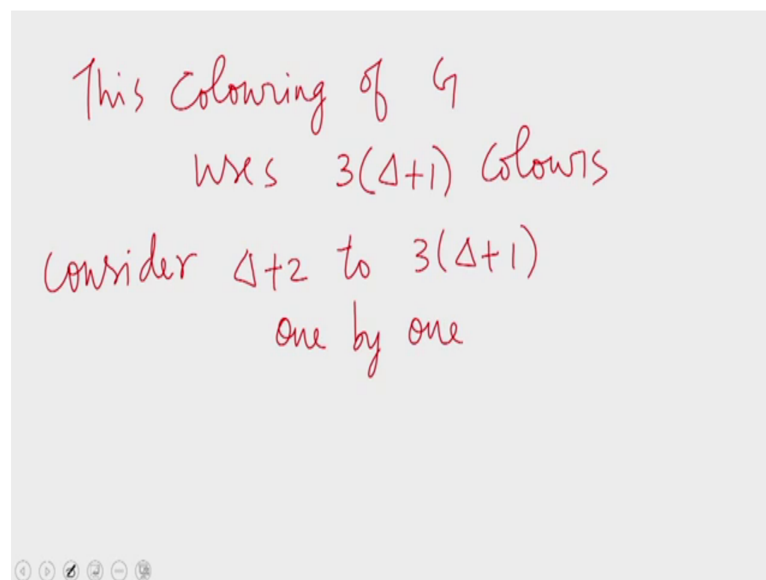

So, we can take an ordered pair of these colors; let us say vertex v gets color alpha in the delta plus 1 coloring of G minus L p and gets beta in the 3 coloring of L p. Let me take the ordered pair alpha beta and call this a color. Now, this becomes a color for G, this is a valid color for G that is because when you take 2 adjacent vertices of G they cannot have the same ordered pair as its color. When you take 2 vertices 2 adjacent vertices, if they are not adjacent in L p it could be that L p gives in the same color. Therefore, its second

component could be identical that is both of them could be beta. But, then since they are adjacent the edge between them falls in G minus L p therefore, in G minus L p they got different colors.

So, if the first component here was alpha, here the first component will have to be some other alpha prime. Therefore, if the edge between them falls in G minus L p then their eventual colors are different. Similarly, if it falls in L p then again their colors are different. If they are not adjacent in G minus L p they might get the same color alpha in that, but then it will they will get different colors beta and beta prime in the 3 coloring. Therefore, no 2 adjacent vertices will get a valid will get the same coloring under this.

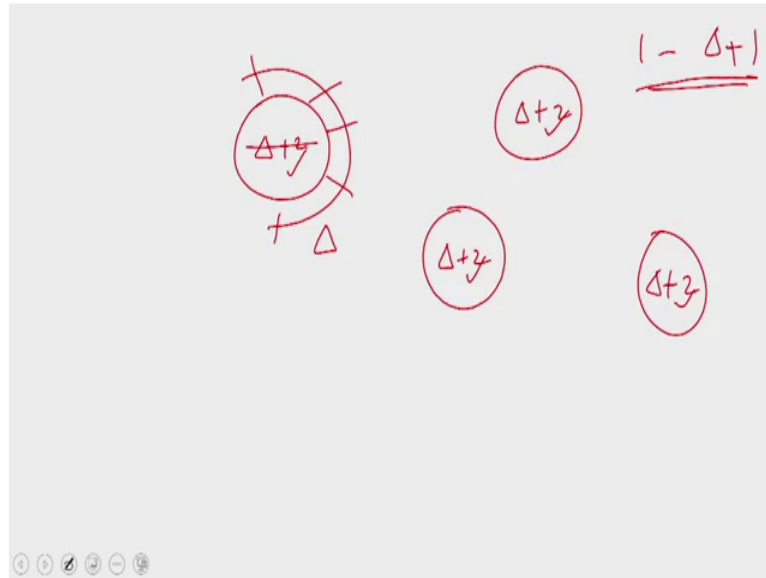
Therefore, if you form colors in this fashion by taking ordered pairs of the 2 colors we get a valid coloring, but then we would be using far more colors then we are prepared to use. How many such ordered pairs could there be? The first component of an ordered pair is a is one of the delta plus 1 colorings. So, there are delta plus 1 possible values for alpha, beta is one of the 3 colors. Therefore, there are 3 possible values for beta.

(Refer Slide Time: 40:51)




Therefore, the coloring of G, this coloring of G uses 3 into delta plus 1 colors whereas, we are prepared to use only delta plus 1 colors on the whole. But, then you can reduce the 3 into delta plus 1 color in into a delta plus 1 coloring. What we do is to consider all the colors from delta plus 2 to 3 into delta plus 1 one by one. First I consider all vertices of color delta minus 2 delta plus 2 we consider all vertices of color delta plus 2.

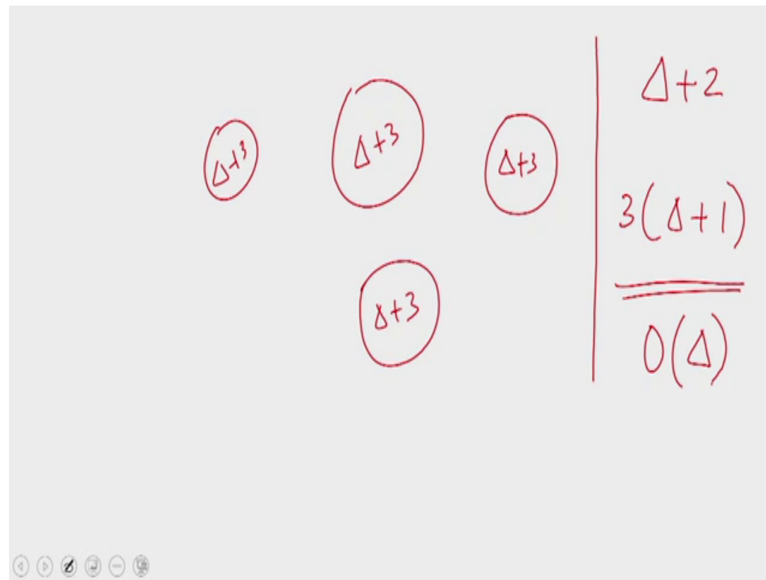
(Refer Slide Time: 41:40)




There cannot be an edge between them in G, because the coloring is valid therefore, no two adjacent vertices could get the same color. Therefore, this will form an independent set. Let us assume that we have one process sitting on all these vertices, this processor will look around and replace this with the smallest color not in its neighborhood. Its neighborhood has a size of delta at the most and we have delta plus 1 colors to use.

So, there must be one color which is not in the neighborhood so, this vertex will adopt one such color. Similarly, will this vertex, this vertex and this vertex. So, all vertices of color delta plus 1 delta plus 2 has have now colored themselves with one of the valid colors; the valid colors are ranging from 1 to delta plus 1. So now, we are done with all vertices of color delta plus 2, they have been assimilated into the coloring that we want.

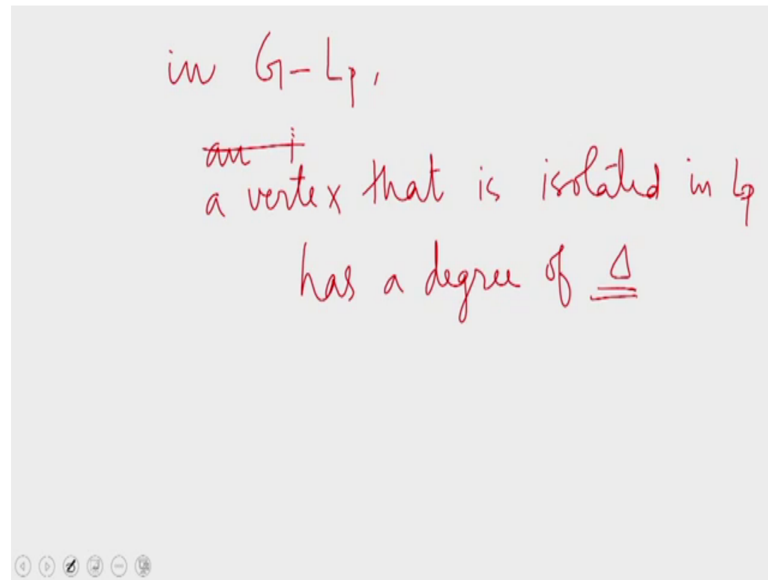
(Refer Slide Time: 42:52)




Now, we consider all vertices of color delta plus 3 they will also form an independent set and we can do likewise; the process of sitting on them will look around in order delta time and pick the minimum color which is not in the neighborhood for these vertices. So, these vertices are also colored likewise when you run through all the excess colors, the excess colors are from delta plus 2 to 3 into delta plus 1. So, we have order delta extra colors, but we have assumed that delta is order 1.

So, when we run through all of them then all the colors have been assimilated, every vertex in the graph would have been colored with one of the valid delta plus 1 colors. So, the algorithm appears complete except that we have to argue that the recursion works correctly, that is after forming the graph L p we removed L p from G and then from G minus L p. Of course, we indeed know that G minus L p is a delta plus 1 a delta degree graph and therefore, is delta plus 1 colorable. But, then will the recursive call color this appropriately is the question.

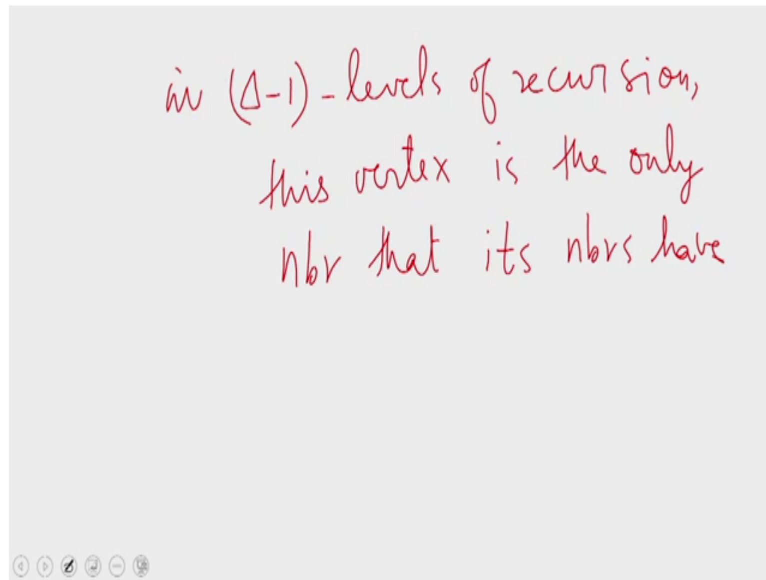
(Refer Slide Time: 44:08)




We can see that this will work correctly that is because in G minus L p a vertex v that is isolated in L p has a degree of delta plus delta, that is this vertex has not lost any of its degree in G minus L p. That is because, it failed to get a predecessor or a successor in L p. But then why did it fail to get a predecessor or a successor? Because, all its proposals were rejected by its neighbours and none of its neighbours made a proposal to it. And why did the neighbours refuse all its proposals?

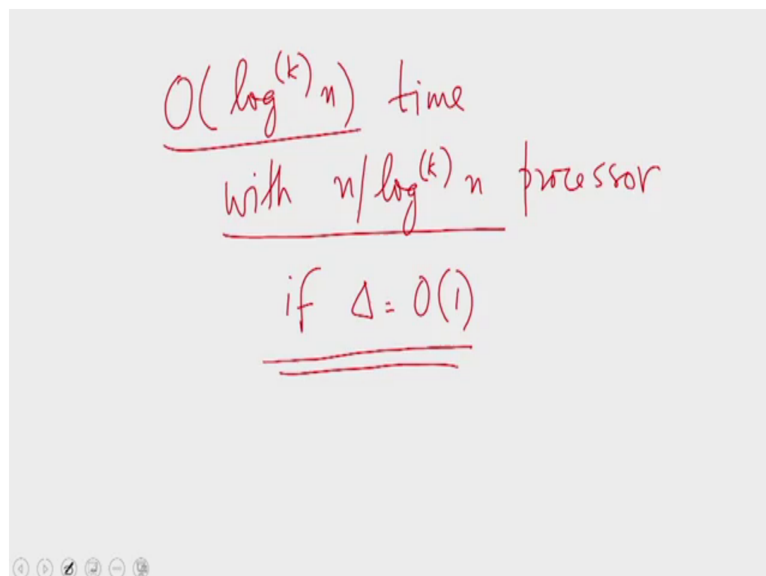
Because, they had accepted somebody else proposal. So, all of them got a predecessor similarly, all of them got a successor as well. Therefore, what we know is that if a vertex is isolated in L p then; that means, all of its neighbours has lost at least one vertex degree. Therefore, this can continue for at most delta minus 1 steps therefore, when the recursion goes down to delta minus 1 steps.

(Refer Slide Time: 45:22)




In delta minus 1 levels of recursion this vertex is the only neighbor that its neighbours have, its neighbours have no other neighbor after delta minus 1 levels of recursion. At this point it will have to necessarily accept its proposals and therefore, its degree will start decreasing. Therefore, by that time we are ready to go down to 1 degree therefore, the graphs that you get at that point can be colored with one fewer colors and therefore, the recursion works correctly.

(Refer Slide Time: 46:24)



So, this establishes that the total time taken for coloring the graph with delta plus 1 colors is order of log k n with n by log k n processors. That is because, in this algorithm the running time is dominated by the cost of coloring a list. A linked list can be 3 colored using our optimal algorithm that runs in order of log k n time using n by log k n processors, we have studied this algorithm before.

So, using that algorithm we can color the linked list. The rest of the cost of the algorithm is all in the recursive calls and the recursion runs only a constant levels deeper. Therefore, what we have established is that the total time taken by the algorithm is order of log k n with n by log k n processors if delta is order 1. So, this is an algorithm for vertex coloring of graphs. So, that is it from this lecture, hope to see you in the next lecture.

Thank you.