Computational Geometry

Prof. Sandeep Sen

Department of Computer Science and Engineering

Indian Institute of Technology, Delhi

Module No. # 12

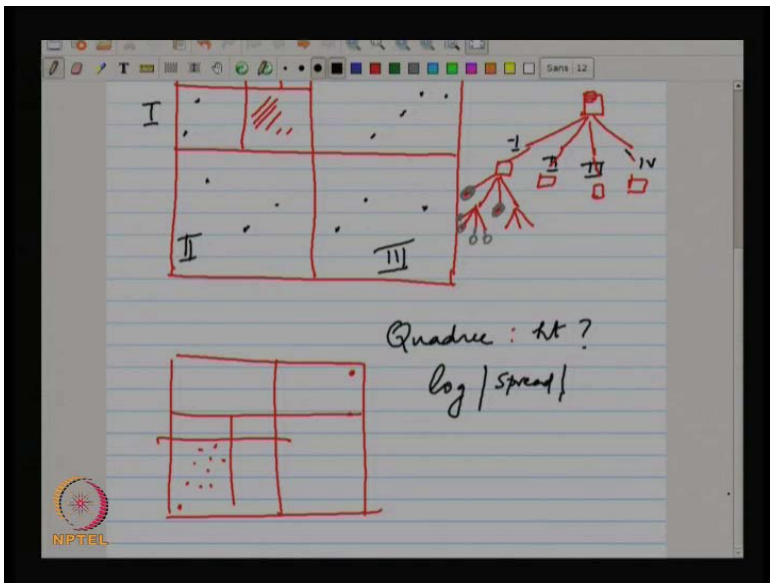
Clustering Point Sets Using Quadtrees and Applications

Lecture No. # 02

Quadtrees: E -WSPD

We continue from our previous lecture, where we discussed you know broadly the applications of a technique point partitioning technique called WSPD or Well Separated Periodic Composition. And today let me try to sketch, you know the algorithms or the kind of algorithms people use to construct the WSPD at to be a more accurate we are looking at epsilon WSPD.

(Refer Slide Time: 00:54)



So, the construction actually proceeds from a very simple structure for quadtrees, we never took up quadtrees in this course not yet because quadtrees usually do do not have any provable properties normally the the kind of quadtrees people use in practice I and it is it is a very popular data structure namely that, you know given a set of points, that is

on a plane then you enclose that in a some kind of a square, in enclosing square right we are you then partitioned the point sets repeatedly by the four sub squares, which are half the size of the original square and you continue this partitioning.

Let me take this one square, so now of course, when you have done this partitioning the second level of nesting these are empty squares right, so you do not need to partition them any further right, so you are going to only partition those squares that contain some point not known, so I am going to continue partitioning here. And then of course, now when you have one point in a in a particular square you can kind of stop partitioning, because your purpose is served that is you want to get it down to a either a constant number of points or a single point, you know whichever, whichever termination condition you prefer.

And then this structure, this partitioning structure can be easily stored in a in a tree, where the root denotes the bounding square, let me use square to denote that and then, that if there are the four children corresponding to the four sub squares right; so then you have this four squares corresponding to the four partitions and then I have sub divided it further and two of them you know basically they are leaf nodes kind of they they terminate. And then the other two you know we we partition and and then again subsequently they terminate right, so we go to the next level again we have two leaf nodes may be I should make them green and then we have only one point in these two and we do not do anything further we kind of again we can see that they have.

So, you continue this partitioning for all the entire four sub squares recursively and this tree is this tree basically corresponds to the the partitioning that we have described which is which is essentially a quadtree, so this tree is called a quadtree, why because analogous to the binary tree it has a maximum of four nodes four children at a any internal node and actually you know this quadtree you know has you know has a I mean if you if you think about one dimensional quadtree, it is going to be an interval where you are going to bisect the interval in terms of its length,

So, here we are always partitioning in terms of the the the length of the square rather than the way the points are distributed; and therefore, ah as I just remarked that the you know the we do not have many desirable provable properties for quadtrees ah in

particular you know you can could have a distribution where you know of course, I can I start with a square.

But then, I may have most points clustered in here, so I am going to basically generate lots of but in fact, this could be much smaller, so I could just I have to go deep and deep to basically somehow split these points; so the whole idea is that I have some point sets and I want to separate them right I want to separate them till my partitions becomes small like one point or just a constant number of points, but there is no guarantee because, I am actually doing the splitting based on the side of the square rather than the way the points are distributed, this is precisely to tell you while you know. So, far in the course we did not talk about quadtree, we are always talking about dividing the points it is in the median line it is very different from the median line, it is the median of the interval it is not the median of the point set.

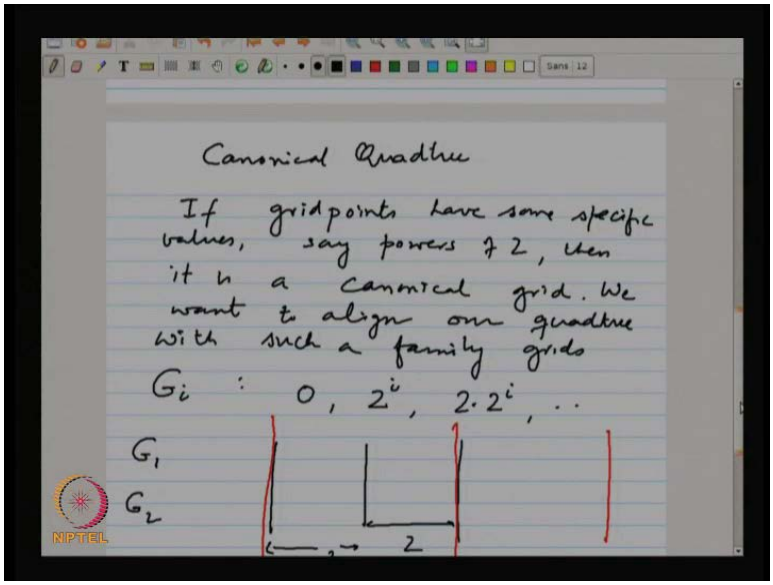
And therefore, you know what can you say even about the most basic parameter of a tree that is what is the height of a quadtree, I could of course you know start with just just to make thing's bad I could have a point in the corner, so that is an of course, to have this bonding square that fine. Thereafter you know I am basically having everything clustered towards one corner, so I have to keep splitting it, keep splitting it you know how how many times did you split it do you have any bound on that (( )). Yeah right yeah what we described in the last lecture is it is a kind of the spread I mean that is going to describe that is going to define the height; so yes I mean that a good thing to observe that log of the spread.

We have to define the spread in the last class, that issue of the maximum distance to the minimum distance, but then what is the bound on that, it is not a bound based on the number of points it is it is something there is going to do with the values associated with the coordinates. Now depending on the module of computation you know you may or may not have a bound you know if you are working with a big module then you can say my input at size n, so no number can exceed, let us say n bits or whatever and then the smallest number can be 2 the power sorry the largest number can be 2 to the power n and the smallest number can be constant.

So, the ratio is at least at least 2 to the power n and log of that is n, so at least in that kind of computation module you have some bound in the module, that we have been

following so far, that is real number no bounds basically, what bounds we have if every input is a real number you know it can be arbitrary, so therefore, you know even if you cannot even get a bound on the height of this tree you know how do we why should we use this tree I mean at least in computer science it is not something that we look for right; so will we will have to do something modify this step not the definition, but the way we look upon upon quadtrees.

(Refer Slide Time: 09:06)



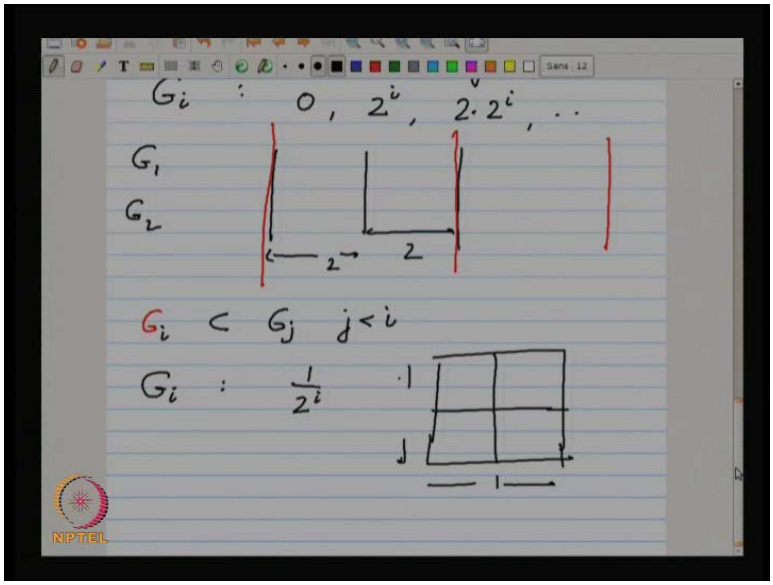
So, the first thing that we do is, we define the motion of a canonical quadtree, what is the canonical quadtree, so in the previous diagram you know we simply just took this this bounding square and the bounding square could be you know some something arbitrary, let us say, some 0 to not 0 you know something, any anything an arbitrary, let us say, some alpha and beta, now if we specify that, the alpha and beta can only take some forms, so for instance we want to always start with alpha is equal to 0 then and we want beta to be let us say power of 2, so what I am defining is something like a grid. So, if you have grid points, so if grid points have some specific value a say powers of 2; why powers of 2? Because, it is also easy for us to divide right it is just exactly division by 2 is very simple, then it is a canonical grid and we want to align our quadtree with such grids such a family of grids

Let us say G's of i is basically looking at the points corresponding to G s of i are 0 2 to the I, 2 times 2 to the i and so on, so forth, so that is what I am calling g's of I, so G's of

1 is 0 2 4 8 and so on, G of G's of 2 will be again 2 to the power yak, so that kind of thing, so G's of 1 these are space, basically you know 2 apart though the coordinates, so everything is space two apart right and G's of 2 will be the 4 apart right this and this every alternate will be the G s of 2 G's of 3 will be again 2 of these will form.

So, you are now getting basically G's of this way what what is the advantage your g's of i basically, is a subset of G's of j where j is less than i right, so that is, when it is going to save some combustion computations if we if we have this grids always that, so when we have them it falls, so we have some predefined kind of grid points.

(Refer Slide Time: 13:08)

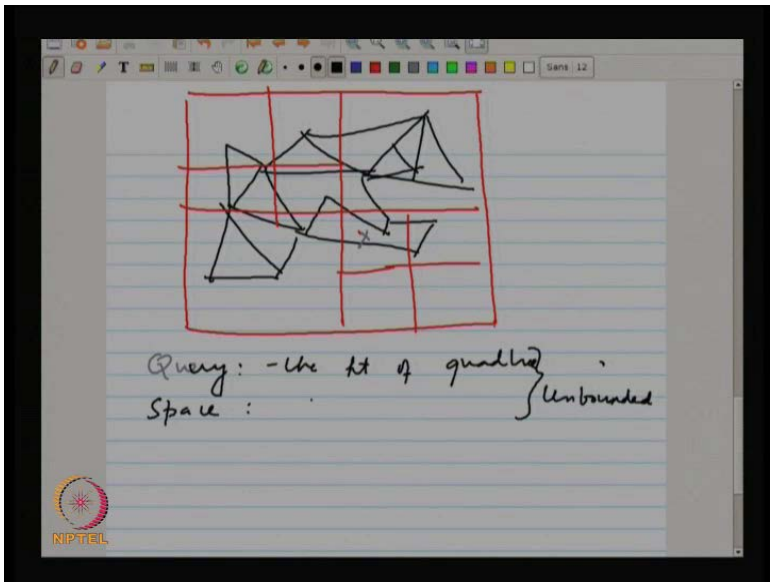


Still hasn't solved our problem that we are facing in terms of you know bounding the height etcetera, but this this is a kind of beginning of that, so in fact, what we will work with the most standing thing to work with, some what is prefer that, instead looking at G i we look at actually G's of i is defined as the grid points; you know 1 over 2 to the power i rather than 2 to the power i. So, you will take the maximum square to be like unit square and then just half it, so instead of looking at 2 to the power i we look at 1 over the 2 to the power i pair, so so the overall square will be, let us say, the unit square bounds everything and then we partition it into G half and then the G 1 fourth and so on. ok.

So, how how what what what is a very simple application of these well, this I mean just the the quadtrees not a canonical quadtrees, but quadtrees we could have, we could do let us say, some kind of point location based on that, right I mean we have studied, so many

things about point location, where we have strict bounds you know linear space or a log n time, so and so forth, but just the trivial idea, where you have a map.

(Refer Slide Time: 14:31)



So, let us say the vertical ray shooting query or something like that, so I again have the map, this is a map they are overlay it with the grid and I do the something basically I keep subdividing it and what do I keep track of, I keep track of which segments intersects which which sub squares right, we have this family of grids and when we do have to do the point location. So, well again it fits very well with our with our notion of plane searching or you know point location, that we will be basically some kind of refinement, if my point is like, suppose I get my point here, if my point lies in in this sub square I only need to look at those line segments, that intersects this sub square and and we keep defining it till you know you know we get come to a point where in that sub square, so as we keep refining it of course, this is a bad this is not a nice example, so let us say somewhere here.

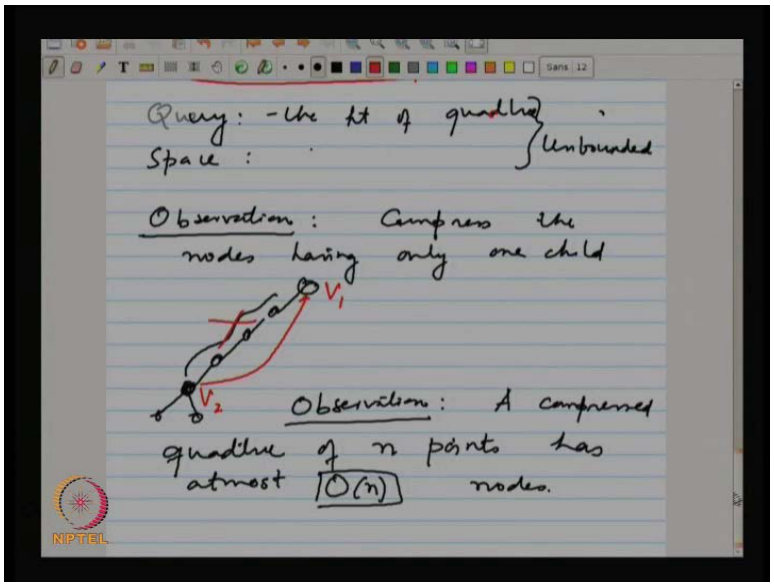
So, we keep refining it till we have only constant number of segments intersecting a square, then we can do the the straight forward vertically shooting and find out what it is, but what is the drawback of this scheme, yeah we do not really have a bound on the either the height of the quadtree which is which is a query, so query is what, the query is the height of quadtree right on which we do not have a bound and the space also we do not have this is, we do not have a bound on the height of the quadtree you know a line

segment can basically, you know keep intersecting the sub squares you know over you know the entire depth of the tree, so again this is also a kind of unbound. So, both of them at this point although the the scheme is a kind of tempting to use, but provable bounds you know we have thought to obtain, just with this kind of limitation, so yes I mean this kind of quadtrees you know have been in practice for about may be 20, 30 years you know even probably before the computational geometry sort of emerged as a rigorous discipline. So, quadtrees have been used if you do use quadtrees, so I cannot wish it away, no this does not work, certainly works in practice and therefore, people keep using it and of course, you know it is a kind of easy to implement.

So, you know this this could be able to a simple application of a quadtree. Now, one way we can try to somehow have some kind of control on the height and in this kind of pathological case where points lie in one corner, is by actually doing something to the structure of the quadtree and what is that, so if you have all points let us say concentrated in one corner most points and only a few points are here and there, so you know you are forced to start with the large bounding square and you you are going down the tree, now most of the let us say, the sub squares are going to be empty.

So, what it essentially means is that, in the tree you are falling this sub square contains something, only one of the sub square contain something and then one of its sub square contain something, so you have a long skinny tree essentially right you have a long skinny tree and in that long skinny tree may be you know you do not basically, have anything I mean just the sub squares are you know contain one sub square of the child. So, this entire part you may want to compress, so if you want child of a node you may just want to compress, so that, when you compress you know you do not have to you do not have to go through a long path to reach you know some meaningful stage, where the points are actually going to be partitioned, before that, no points are actually partitioned we are simply just following the sub square, of a sub square, of a sub square; so you can try to compress that path, so that, is one easy observation, so to do something to the to the to the structure of the quadtree.

(Refer Slide Time: 19:06)



So, so one observation is, compress the nodes having only one child, so what looks like this and then maybe you have some you know there is some bifurcation there, so this entire path I can compress, to say suppose this node is V 1 and this node is V 2, I can directly hang V 2 on V and forget about this. Now, in that process I have done 1 dimension what is the dimension that I may have done when I go to the sub node of the original quadtree I go to the node of the quadtree I know that it is exactly a sort of a square. Let us say it is an aligned square, it is exactly half of the size of the of the, now here, we now compressed you you know let us say path of a you know 5, 6, 7 tree there is no longer that case, so I have to somehow find a way to to identify the, that you know V 2 and V 1 are related, suppose the pointers will give us that; now besides, the pointer there is 1 more property of this aligned quadtree, so I identified spell out everything, so let let me write just how.

So, what what was one of the advantages of an aligned quadtree is that, I can store the quadtree not as a tree, but actually, can you say something what kind of data structure, we are we are searching right, so what what are the kind of structures, that can be used trees are used because, you know we always have a link from the parent to the child and you can follow those links, what other kind of structure can we use, skip less fill will also be some kind of point falling pointers, yeah some kind of arrays, but you know what more specifically, what see a complete binary tree is never stored as a tree, right I mean it is stored in an array because, you know implicitly that, you know what is the child of

which one just you do not have to store the pointers, so here I have, suppose I have an aligned quadtree where I know precisely you know at depth i; so the advantage of aligned quadtree, is that let me just write it out.

So, one advantage is that, at depth i from root, we know exactly which sub squares can be present, so corresponding, so what I basically mean is like a complete binary structure I give you two nodes, you can actually tell me whether or not it has the relation of an ancestry ancestry child, Right the advantage of complete binary tree is that, because, you know you have full levels and so on and so forth you can map it; basically, to an array where given two nodes, I can you you know in constant amount some whatever some calculation I can immediately tell you that, you know this child this is actually ancestor child of this node and vice versa. So, here also I claim the same property, if they aligned and I give you this two nodes, you can quickly tell me because, I know that for the ancestor child relation it has to be a sub square, of the sub square, of the sub square, I know exactly how the numbers look and how the grids look right and if I, so if I can, so then what would be what would be a structure to store this, yeah hash tree, good see the reason why you know you do not think about hash table for a complete binary tree is because, you know you have, your array of size n and basically, it has about n nodes and n power two nodes, but you know in this particular case of a quadtree you know you may have some sub squares that are empty.

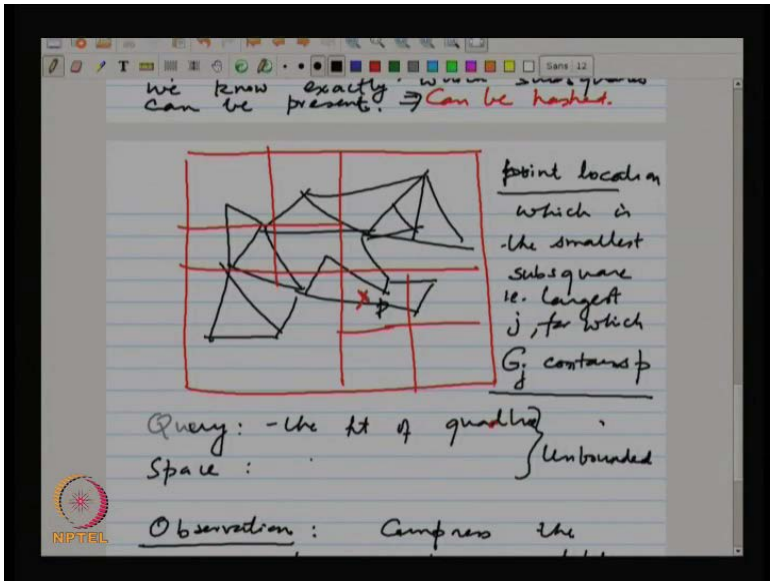
You may not want to store it after a while you know you you have only 1 point in a in a sub square or an empty sub square you do not want to store the children at all. So, prior you have no knowledge about, how the things are going to be distributed, but you know that what you can claim about. So, once I have eliminated these long paths what can you claim claim about the number of nodes of a quadtree ok..

So, observation A, so this is called a compressed quadtree of n points, has at most where what is that quantity, log n no no how can we log n, I mean there are n points right order n why is it order n whatever you know log n is cannot do anything on the height we are not able to do much I mean not yet right. So, every every level point is getting partitioned, so it has to be partitioned at at least it can be partitioned at most n minus 1 times right before it becomes trivial point sets and you know these kind of nodes where you you you hang it to this, one may be some extra another n nodes So, it certainly no more than order n nodes.

So, we have at least gained something you can now say that, you know this this kind of a quadtree compressed quadtree has only order n nodes and moreover this compressed quadtree or in fact, not just a compressed quadtree, that the the aligned quadtree one advantage of let me see from this advantage.

We can we can claim just as equal to remark, this can be stored in a hash table I can hash every square because, I know that, every square now is not an arbitrary square it is an aligned square; so it can only have certain kind of coordinates right. So, I can hash it, can be stored in a hash table, no we do not know, we do not mention that, but suppose I am now let us let us try to implement a search algorithm. So, if I am trying to do basically whatever I trying to do I am trying to find you know what kind of. So, if you look at this application, whatever I trying to do we are trying to find that, which is the smallest sub square that contains this point because, the smallest sub square means, that we do not set the sub divided any further because there is no useful information after that right.

(Refer Slide Time: 27:17)



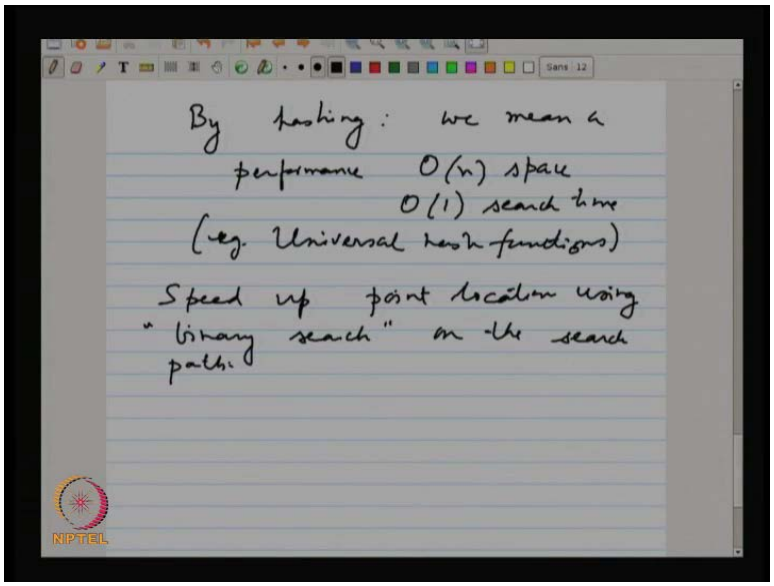
Once the sub square contains only one, let us say object or one point, we are not going to sub divided any further, so point location query actually is of the following kind, that which sub point location query, which is the smallest square or let us say, a sub square which is basically, which grid level you know sub square largest j, for which G j contains p, so this is called p.

So, a search in this quadtree essentially amounts to, which is the smallest square, that contains this point and how are you going to find this, one is that you know I just, do you know from the original bounding box, I go to the child and if the path length is h, then this could give me a performance of h right height of the quadtrees certainly a bound on the search time, but now, that we have been able to hash I claim, that we can do it much faster yes.

(( )).

No no when we hash it to a table you know we are only hashing those squares that are relevant to the quadtree, we are not going to hash anything, that is empty or basically, after the leaf node we do not have any children of that we have certain number of nodes in the hash tree as a in the in the quadtree and those instead of storing in a tree explicitly a tree with pointers we just hashing it them. So, that you know I can find out given a node whether its child exist or not by simply looking at the whether, I will search for that every square has a label let us say, right every square has a label I can label every square. So, every square has a unique address or whatever and that can be hashed. So, I can always find out if there is a child of a certain node by just checking whether whether you know it is present in the hash table, and let us assume the another thing, maybe I should I should want or you know point out is that, we have a very good hash function, where we can do constant time hashing, like; essentially, we was in hash tree, so we are talking about some you know some you know some some you know broadly low kind of hash form, we are talking about a proper hash function of A, so, by hashing we may the performance of order n space and order one search time example universal hash functions.

(Refer Slide Time: 30:13)



So, I hope everyone knows about this or otherwise you know I could give you some background material, so our hashing works very well, you know its constant space and its constant time for search. So, if I have to search where I do not explicitly have the pointers, you know I first of all there is a some unit square, every point is present now I want to see if there is a certain child, I will find out if the child exists or not.

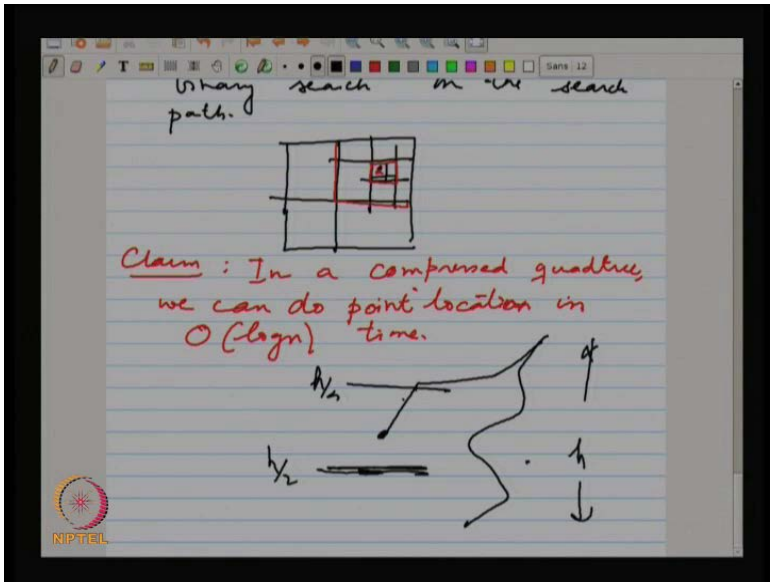
I can always find out a child will have a particular label, I will just check whether it is present in the hash table or not, in fact, I can find out if a child, of a child, of a child is present or not because, again everything has a unique label. The quadtree all the children of a quadtree have some distinct unique labels right and I can always find out if a certain node is present or not.

Now, given that, given this property I claim that we can search much faster than actually, trying to just go sequentially from the root to the child, to the child, to the child, again using the hash table because, now we are instead of following the pointers, we are using the hash table to find out the child, of a child yeah right great, so you do basically a binary search on the path basically, right.

So, and speed of search, speed of point location, using binary search on the search path right eventually what we need, we need to find out which is the smallest square, that contains this point right and I and I know precisely, if the point is you know in a certain. So, this unit square as I subdivide further, it is the second level I know where it is, the

third level I know where it is, the fourth level I know where it is, so it is a question of you know as I sub divide, I know exactly the nested sequence of squares, where the point is going to lie you give me a level or a depth and I know exactly which square, what is the label of the square, that is supposed to contain that point and the question is does, that square actually exists in the data structure in the quadtree.

(Refer Slide Time: 32:26)



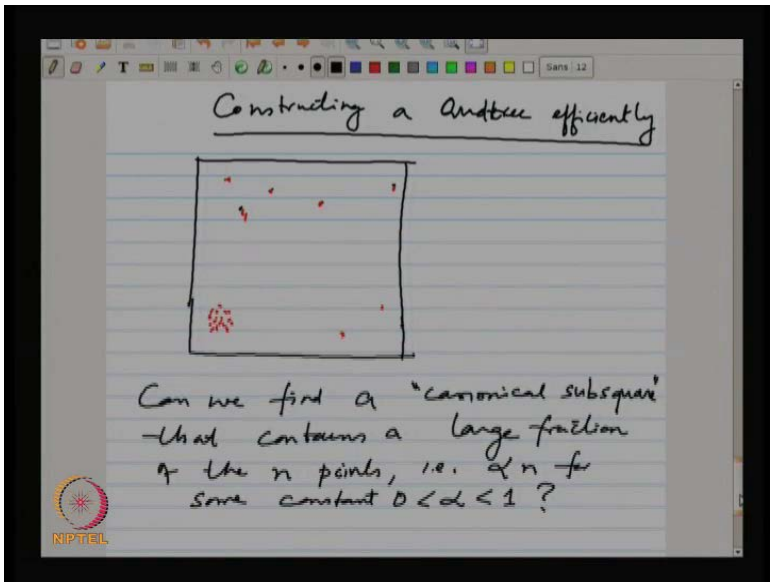
So, it strictly speaking it is what is called as unbounded search, but the unbounded search can be basically done by using the binary search, what is the smallest sub square that exists in the hash table containing that point, because all the family of the nested squares have distinct labels right so, so now, we immediately even if we have let us said a path length of n suppose now see what is happened. So, from the compressed quadtree we have got it down to a situation where a length of a path is no more than n, how do we actually construct a quadtree, it is this still does not tell us that a quadtree can be constructed very quickly, we are compressing the quadtree as a as a it after this thing after processing right here is the quadtree with someone built somehow and now we have is a post processing the compressed quadtree the actual quadtree, that give rise to that particular point, we we we can shrink the paths and we can compress the paths put them in a hash table full structure, but how do we actually build a quadtree in some feasible amount of time (( ))

Well for that we have to define some algorithms, you want to do that, you want to do that, how we do it.

(( ))

Yeah we do not get a bond that is a thing we have to very quickly somehow get a bound on the height we able to compress, we need to know that, you know all the thing's in between node lead us to anything, right I mean if the paths of where to you know which is basically, a linear path is basically, the trouble spot right; if we are splitting then, you know we can charge it every time we split, we cannot we cannot have more than n minus 1 splits, we cannot have more than n minus one splits, but then you know this long you know skinny paths are the one, that are that are troubling. So, we somehow need to figure out that quickly in the construction, every time you know we have two branches we can charge right, So, that turns to be a somewhat tricky thing, but not that much.

(Refer Slide Time: 41:27)



So, what you do is, so again, what is the problem? The problem is that, so constructing a quadtree efficiently, so again the the basic problem is that, you may have it is a unit square and you know some of these points may be reasonably distributed. But let us say you know just that you know we have this trouble spot, which is really really a sort of small, so the times, number of times as I divide it into sub squares is going to be quite large and that that is the path, those are the paths basically I am worried about, not just

one such cluster, but that be many such small clusters, those are things that I am worried about.

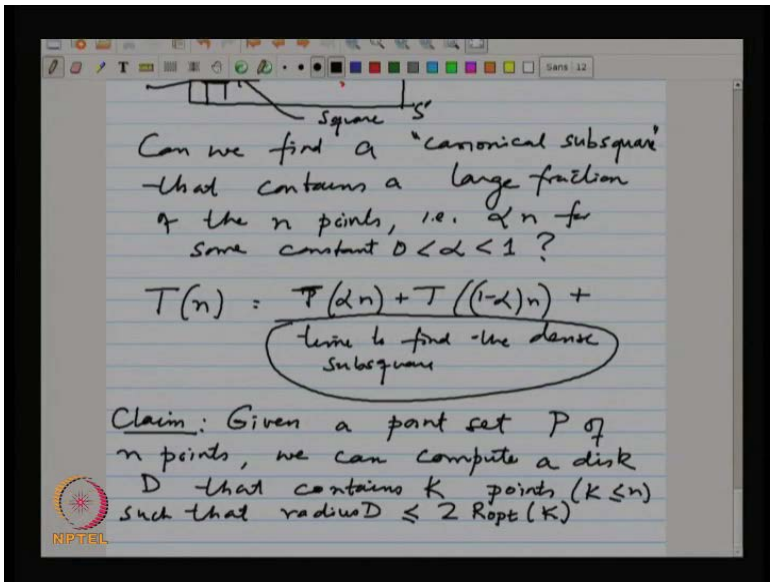
So, the idea is the following, that can we find a sub square, so as in canonical basically, means aligned with the grids, a canonical sub square whether, whatever I am saying actually goes through any dimension, it is just that instead of. So, in two dimensions it is a quadtree in a three dimension it is a 2 by 2 by 2 right, so a it is a knock tree and in d dimension is basically you know 2 to the power d, basically that number of children will be 2 to the power d.

So, but but the same and the same arguments you know everything goes through its just, but you know it becomes more and more expensive because, we have more children, that is all, but the qualitative arguments all go through. So, can we find a canonical sub square, that contains a large fraction of the points, where large fractions are constant fraction points of the n points let us say, that is let us say, some alpha n for some constant alpha, if we could then, we are in good shelf, why? Can you tell me.

(( ))

Well we do not need, well cannot start from this, you also have to see it is only a fraction of the points, that you know we are found that, there is a there is a canonical sub square, that contains a fraction of the points, but there also points distributed everywhere else in a square I claim that, with this we can devise you know a a recursive algorithm. So, we can we can look at the points, we can look at the ah we can we can look at the sub square the canonical sub square and the remaining thing and then, what I what what we can look at the recursively built the the quadtree for this canonical sub square and for the remaining sub square minus these points and place them together; so we recursively call the algorithm that is all.

(Refer Slide Time: 45:28)



So, in this particular case let us say, you know here is of course, if the entire cluster may not be may not able to be cover, but let us say, at some level, so may be this is the dense sub square you know this is the dense, you know that alpha n points you know this particular thing contains the you know some substantial number of points, then what we can do is, we can take out those points construct the so just call this some you know the this is the sub square the square s prime ok. Then what we do is we define so, so initially we have to this, is the unit square and now we devise a quadtree for the original square minus these points, so p is the original set of points. So, p minus s prime let us say a some quadtree and for the s prime, we have another quadtree and the s prime is going to hang somewhere from here, from the original one, that we can figure out, because we know exactly which square this is, so we can attach it.

So, this 1 this sub tree sorry this this quadtree corresponds to this this corresponds to s prime right this is for alpha n points sorry and this is also actually we have to, so this this is not enough, so we we we have to ensure that, this alpha is you know strictly between some two bounds you know it cannot be just all the points are here then it does not help. So, we need to have bounds both ways, so this is a sub tree corresponding to 1 minus alpha n points and this is a tree corresponding to alpha n points and then we built them whole recursively, again using the same same idea, that again we will look for a sub square, that contains a substantial number of points and if we, so what happens is, we can write this recurrence to analyze the time bound if T's of n is the time to build the this is

in this algorithm then it is, we are building basically, for well T of alpha n plus T's of 1 minus alpha n plus the time, that we require to find this dense sub square.

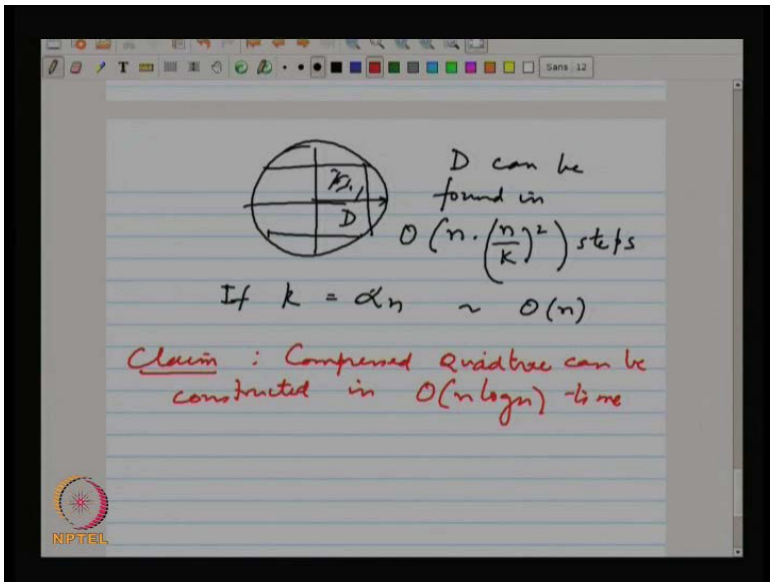
It turns out that this one, so this becomes a problem on its own it is an it is a you know a problem of you know it is it is not a completely trivial, so how do you find a sub square that contains a large fraction of the points. First of all, does it does such a square in sub square exists or not that, is all real for that I will pose this problem and leave it as a kind of an exercise for you.

So, I will I will make a claim and the proof of the claim is left as an exercise, so given a point set P of n points, we can compute a disk D that contains n points, sorry K points K is less than or equal to n and k is a specified parameter contains k points, such that radius of D is less than or equal to two times the R opt n D and I will just R of let say K lets say K let I will just define what is R opt K, so R of K is the smallest disk, that contains K points exactly K points.

So, well I mean exactly means you know you do not have, let us say, some singularity case that all points are on the on the same quadrant or something like that, so without singularities, so think about the smallest disk, that contains K points and this itself is a you know this is not such a simple problem, but do not think do not conclude that it is an empty hard problem.

It is not an easy problem, it is not an empty hard problem either, but what is being said is that we we do not even look, we do not even need that optimal smallest radius, we we will be find even even if it is a factor of two of that and that will be much easier to find and and 1 if we can find that disk which is twice that I claim that we have solved the earlier problem that we have, so from the disk you can actually carve out some squares, the idea is that from the disk you carve out a few squares and 1 of the squares will have a large fraction of points, so it is like this.

(Refer Slide Time: 51:34)



So, I have a disk which has a these points are may be with certain you know this has a certain radius let us D, So, you can carve out you know some fixed number of squares. So, that one of the squares will certainly have that, because the entire if all the squares are inside you know one of the squares will have a large fraction of the points

And how do you find this and how quickly can you find this approximate disk ok can be D can be found in order, with the using a very simple algorithm there are many ways of doing it, but I am just stating one of the simplest form. So, if K is omega n or let us say alpha n, so this is order n, so this k is the fixed fraction of n then the running time is just linear and if that is, so this 1 is linear and then the whole thing is if alpha is some kind of constant it will be n log n, so you can actually build using this method, so the compressed quadtree.

So, claim compressed quadtree making two instances can be constructed, so let me conclude here today; so still no no sign of the epsilon, yet we are still not done the epsilon we are suppose to go from quadtree to the epsilon well separated periodic compositions. So, epsilon has not made any appearance here, so let me just tell you in one line what we are going to do next time we we will start with a set of points will build will build a compressed quadtree and from the compressed quadtree we have a very simple algorithm.

Simply simple means, simple in terms of describing it is going to be very simple from where we will will this will build a w epsilon WPSD in time again, that is proportional to the number of the pairs that will output. So, WPSD outputs some well separated pairs right epsilon WPSD outputs a set of well separated pairs of subsets of points and this algorithm starting from the set of points will first build a quadtree and then from there we will build a epsilon WPSD will output those and it will run in time proportional to the number of pairs.

So, the total running time will be the number of pairs, that it outputs plus the time to build the quadtree is n log n, which means that it is going to be fairly you know a sort of efficient and there we will find some kind of dependence on the epsilon, when we actually build the epsilon WPSD, we will have a dependence like one over epsilon to the power d where d is the dimension.