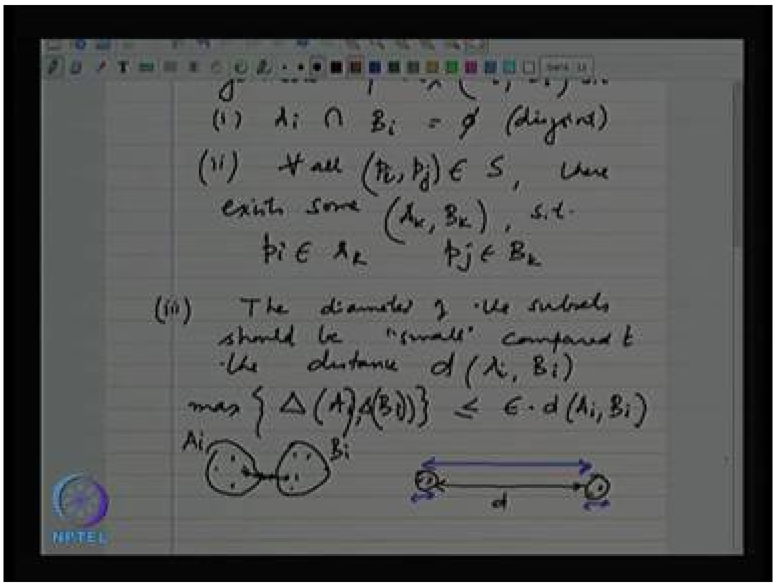
Computational Geometry
Prof. Sandeep Sen
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Module No. # 12
Clustering Point Sets Using Quadtrees and Applications
Lecture No. # 03
Construction of E – WSPD

(Refer Slide Time: 00:37)



(i) $A_i \cap B_i = \phi$ (disjoint)

(ii) $\forall$ all $(p_i, p_j) \in S$, there exists some (A_k, B_k), s.t. $p_i \in A_k$ $p_j \in B_k$

(iii) The diameter of the subsets should be "small" compared to the distance $d(A_i, B_i)$

$\max\{\Delta(A_i), \Delta(B_i)\} \leq \epsilon \cdot d(A_i, B_i)$

So, we resume basically where we left of a lecture 30, I started discussing well separated pair decomposition. So, given a point set S, we want to generate pairs A i B i such that A i intersection B i is that is there disjoint, and for all an adds p 1 sorry p i, p j belonging to S, there exists some A k, B k, such that p i belongs to A k, and p j belongs to B k. So, every pair of points that you can generate from the given point set S, we should have some this pairs of sets A i, B i.

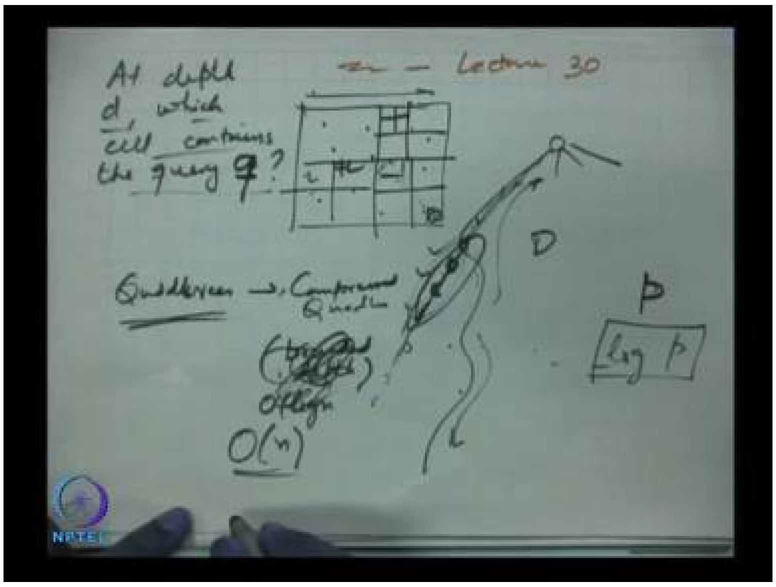
So, we want to generate pairs of subsets such that this properties hold. So, every pair p i, p j should be a present in one of the pair of subsets that we are generating. And the well separated pair part essentially is that the diameter of the subsets should be small compared to the distance d between A i, B i. So, most in a formally what we want is that max of say, let us use a something like a delta for the diameter of A i, B i should be less than equal to epsilon times distance between A i , B i.

The distance between a two subsets of points is the closest pair. So, if you have A i, you have these are disjoint remember that. So, this… So, the pair of… So, there some points that line these two sets, you have to look at that pair of points that are closest, and that defines the distance between the subsets A i, B i. So, what we want essentially is that the diameter. So, it should be 2 scale it should be like this, you know that the distance should be much figure compare to the diameter of the points. So, the speed of the point sets, and what it allows into do is… If you since the distance much larger. So, this is the distance, if you take any pair of points.

So, the distance between any specific pair of points, you should be able to a pros met by d, because this is very small compare to the distance. Maximum of mentally(( )). (No audio from 05:16 to 05:35). So, that is a basic reason, why you are doing all these. So, that even, if you take two specific points, the distance between that pair is not much different from may be another pair of point that you may have chosen. For some representative points that you have chosen the two sets fine. So, this thing this a epsilon well separated pair construction, the generation of the subsets will begin and the algorithm will start with essentially with every compressed quadtree.

So, assume that we are already constructed a compressed quadtree on the given set of point p. So, in the last lecture, we discussed how to a efficiently construct a compressed quadtree of a given set of points, and I will actually retract something said on that particular lecture that is... So, if you recall the construction made some like this. So, we define something called this canonical grids, which are basically powers of two grids, and then we define our cells based on the canonical grids and when you do this quadtree.

(Refer Slide Time: 06:56)


At depth d, which cell contains the query q?
Lecture 30
D
p
log p
Quadtrees → Compressed Quadtrees
O(n)

So, first you know this is the enclosing square of the given set of points and then you generate these quadtrees. So, this is the real root node, and you generate this 4 cells and you keep generating them recursively till you know something is empty, and then you do not generate any more. So, that was the initial thing, but that make that you know, we have to go very deep depending on the dimensional of the unit square to begin with. So, the depth of this tree would not be bounded, if you just follow this construction, then what we did was recompress that. So, compress these pass you know, if you have paths having just one child you decided to compress the whole thing, and by compressing the whole thing, we could argue that the depth would not be more than logarithm, because at every point there will be branching of at least two and every root a leaf node corresponds to exactly one point of the exactly one point. So, this we stop basically when the sub square contains exactly one point that become solution.

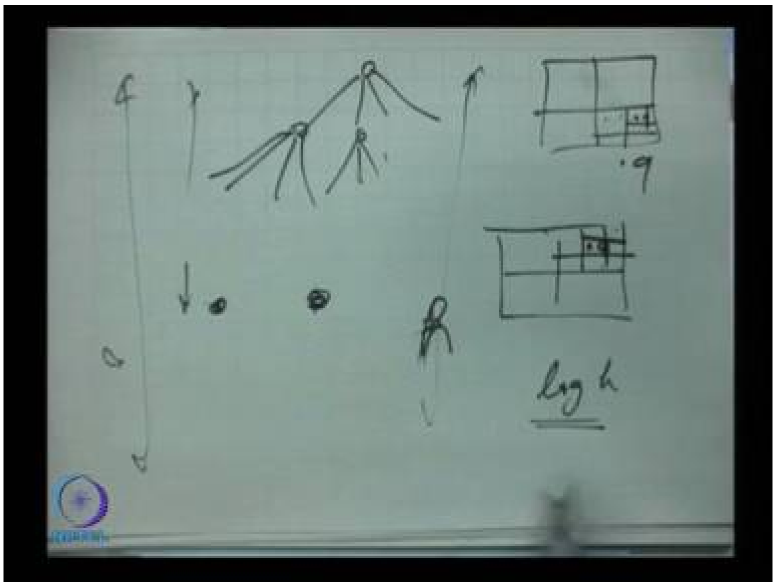
So, going from quadtrees to compressed quadtrees, we manage to basically bound the depth. So, this is a bounded depth pairs about log n, but then … We also made use of some property, even we were discussing just the original quadtree that when you want to search for a point, you could do something like a binary search to find out, which is the basically the path taking by window original quadtree, which is the path taken by a point. So, that we could do find out by doing a binary search on the path itself. So the path length was p in log p time one could actually do a binary search by figure, because you could use the hashing. So, we could find out exactly at depth d its at depth d, which cell contains the query point p. So, this one we can simply you know, just have a function that defines it you know that at, because this is a you know, we are sub dividing the square into exactly into 4 parts.

So, at depth d we do exactly which cell is a suppose to contain that point. Now, so, if that square is actually present then we hash it, if it is not present it is that you know that, the point must be present you know higher up in the tree and, if that square is present then you have to go lower down in the tree. So, that is how we are doing the binary search. See when you have the full quadtree defined you know exactly, because of the way we are sub dividing the square which square would occur. So, in the given tree, where so, this square should occur may be a level one you know, some other squares at depth d, we know exactly where is square is suppose to occur, if that square is present.

Now that may not be present at all, because the construction stops when we have empty cells are basically, we stop when there is an exactly one point. So, when we trying to find out do some kind of query for a point p find out which cell contains this point p, among the cells that are actually stored, we only store the non-empty cells. So, to find out which cell contains this point p, we could do a binary search on the path itself, because we at depth d I know which cell is suppose to point contain that very point p. If that cell is and we are storing all these cells we hashing, because we know exactly the dimensions of this square or address of the square, we can actually hash it. So, if that square is present we know that, we need to go further down, deeper down into the tree. So, find out which cell contains a point, if that square is not presented it means that path stops much before that, because a node is become empty.

So, by doing a binary search of this form is that self present that is basically given a depth d is the point occurring the before the depth d or if the point is below the depth d. So, we can actually look at that cell that is all point. So, we could do a binary search on the path itself. So, if the path has a... So, p is not a good notation may be actually used p q a. So, if that path is p, the length is p then we could do this kind of searching for a query point q in log of p time using this whether, it is present or not is it deeper than d or is it occurs before d. So, where does the path basically end, at some point see this you can infinitely divide, but you talk when you have an empty cell (( )).

(Refer Slide Time: 13:07)

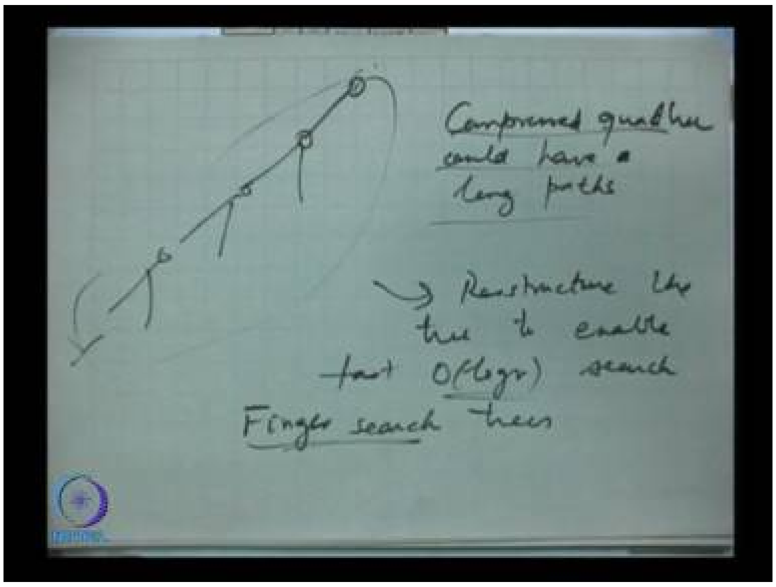


So, the original see… So, what is the originally the quadtree, if I do not compress it, it is like a complete binary tree right, it is like a complete binary tree, and you can go to infinite depth. (( )) So, given any cell you know exactly, where it should be present whether, you complete binary tree. Now, instead of storing it in a tree fashion, we are actually hashing this thing that is all, but I can compute exactly at depth d, given any node at depth d what should be the address of this. So, I am going to hash it. So, if I am trying to find out for a query point q, q is here at what point basically we stop subdividing, I am doing essentially a binary search on the depth, that I have to go still before basically the cells become empty.

If q is you know, if there is some sub square, where q is present, after that you divide n over your divide any further. So, it should stop here, I am not going to divide this any further, but I do not know initially what is the depth at which this q is 2. So, I will do a binary search on the depth of that, where the point q is likely to be stored and that I can cut down my search time to. So, if the quadtree has depth maximum depth p or h let us say height then I can do it in log of h queries. Now, where a either is slide over site, when we went from the… This is the uncompressed quadtree, the moment I compress the quadtree then I loose the ability to do this binary search, because I do not know compress quadtree basically means at I compress this path and different paths, even compressed in different ways. So, I do not know which exactly, which of these may not be consecuting nodes, because I have already compress them. So, I loose my ability to do the binary search this is, where you know there is on over site that, how do you actually do the searching in...

So, we found well when you do the… I think we have to go back and see that what we did actually, why is just height in log in why you are just a... So, there is a problem with the height (( )). So, why did you not only the logging logout the maximum in. So, that is know, when you were the depth that is not bounded, the size of the trees bounded, because at every node there is at least a branching of size 2, you can only partition it n minus one times in the therefore, the size of the quadtree compressed quadtree bounded by order n. So, this does not give you a bound in the height. So, (( )) log of next by mean something like the way the next (( )). So, that is when we assume that we have a bounded spreads. So, I am not assuming bounded spread. So, what happens when is that you tree has a log n nodes, but it can be very squid.

(Refer Slide Time: 16:43)


Compressed quadtree could have a long paths

→ Restructure the tree to enable fast $O(\log n)$ search

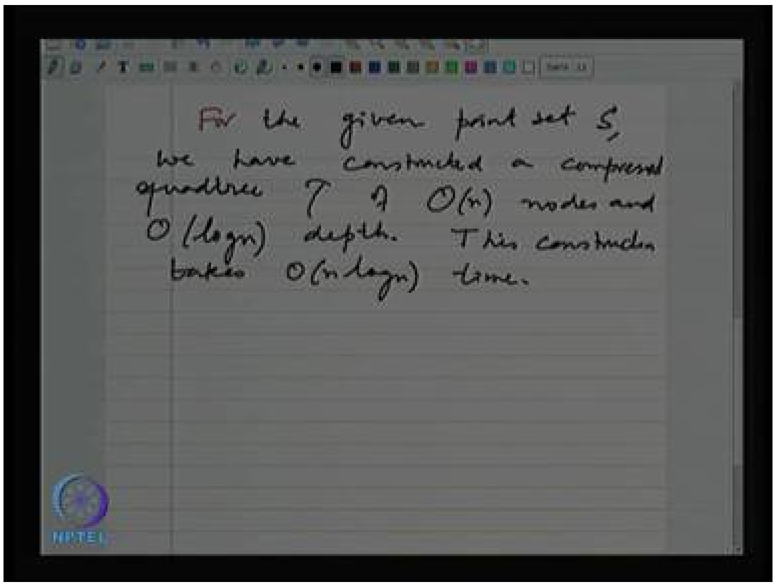
Finger search trees

So, you could have a situation, where you know you are just removing a long something like this you know it could have branching of 2, but it could have a long path. So, compressed quadtree could have a long path. So, this is where we have last ability to do searching quickly in log n time. We could construct this tree in n log n time, that is what we discussed last time. So, now going from a tree of this kind a compressed quadtree to restructuring the trees, you want to restructure the tree to enable fast order log n search. So, there is some data structures tricks that we required, and which something that I do not want to get into today. So, if your familiar with things called finger search trees.

How many of you heard about finger search trees. So, I see most heads moving this way. So, finger search tree actually allow you to a start searching not only from the root node, but at some selected points for a fingers is that you have some extra information, instead of starting for the root you can start at a more convenient face is at least place called fingers now. So, finger search trees you know in many situations allows it to do fast searching and this tree can be restructured, using the same notion is finger search trees and that requires a pair amount to discussion which I cannot get into today. Let us only just assume that this tree can be restructured in a way which will support order log n query time for when we doing a point location to find out which cell contains this point a query point q we can do this whole thing in log n time after again we restructure the compressed quadtree into a finger search tree. Finger search tree predetermine nodes on the tree (( )) we will be discussing something I do not think not in this course we discuss in out off line separately.

So, I will… So, this is, because on a last in the end of last lecture know, I realize that you know I till quiet complete discussion about how do you do the order log n bounded a compressed quadtree right now, we will begin our discussion assume that we have a compressed quadtree that is height log n, after we have restructure the compressed quadtree using some finger search tree techniques. We already proved that finger search tree is already bounded… Well finger search tree is the generic family of trees for this particular restructuring what we will do is, we take a tree and you know will use the notion of what is called as tree separators. You can separate our trees in a two more and equal parts and restructure. So, that the finally, the higher becomes just what logarithmic. So, it is not very relevant to what we discussing not certainly, not geometry it is a data structure tree. So, I am skipping it from the time being, and actually I do not intend to (( )). So, let us go back. (No audio from 20:25 to 20:37). So, what we have now we initially.

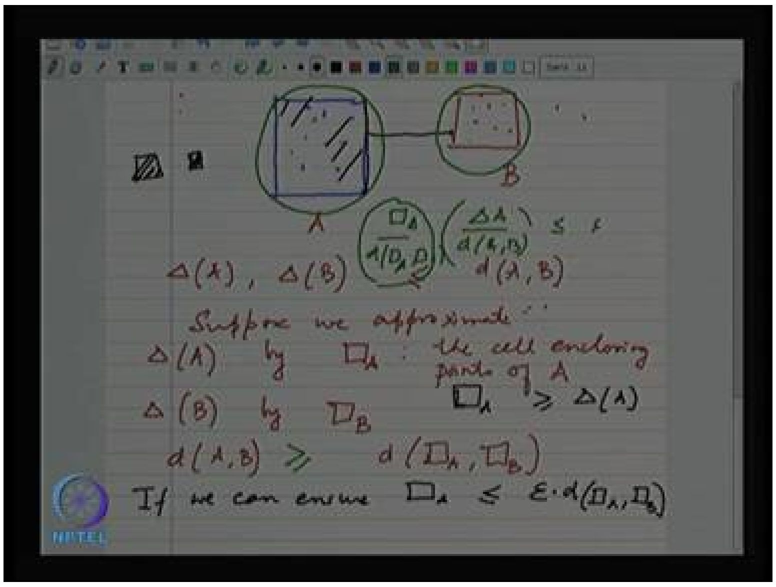
(Refer Slide Time: 20:43)


For the given point set S, we have constructed a compressed quadtree T of O(n) nodes and O(log n) depth. This construction takes O(n log n) time.

So, will assume that this you know, we have given for the given point set S, we have constructed a compressed quadtree tau of order n nodes and log n and this construction takes, if you recall this construction takes n log n time. So, starting point for the well separated pair decomposition is this compressed quadtree tau. Now, let me just first right of the algorithm (No audio from 22:02 to 22:42) Let me step back, because this just think of it clarify one thing before it will do this. So, when we defines actually the well separated pair decomposition, we are talking about diameters of point sets A and diameter of points sets B. So, (( )) and also distances between 2 subsets of points. Now, if you somehow we construct this decomposition, it appears that we need to know how to compute the diameter of a point set else how would I even verify whether, this something is a well separated pair decomposition.

So, I should know how to compute distances between 2 subsets of points, I should also known how to compute the diameter of a given set up points reasonably efficiently otherwise, I will be stop just to that. So, to circumstance to this problem there is a clever tree that is used. So, we actually will never compute the precise diameter or the precise distance between two subsets of points, we will assume see we started with this quadtree what is quadtree do, quadtree actually, if you know partitions this space into cells and this cells are basically orthogonal kind of cells, the discussion that we had previously you know goes through any dimensions. So, it is not specific to this 2 dimensional, I will drawing my pictures into two dimensions. So, you know the squares and sub cells and sub squares, but actually the same thing goes through in any dimensions. In fact, our well separated pair decomposition will also goes through any dimension.

(Refer Slide Time: 24:53)



A
B
$\frac{\square_A}{d(\square_A,\square_B)} \le \left(\frac{\Delta A}{d(A,B)}\right) \le \varepsilon$
$\Delta(A), \Delta(B)$
$d(A,B)$
Suppose we approximate
$\Delta(A)$ by $\square_A$: the cell enclosing points of A
$\Delta(B)$ by $\square_B$
$\square_A \geq \Delta(A)$
$d(A,B) \geq d(\square_A, \square_B)$
If we can ensure $\square_A \le \varepsilon \cdot d(\square_A,\square_B)$

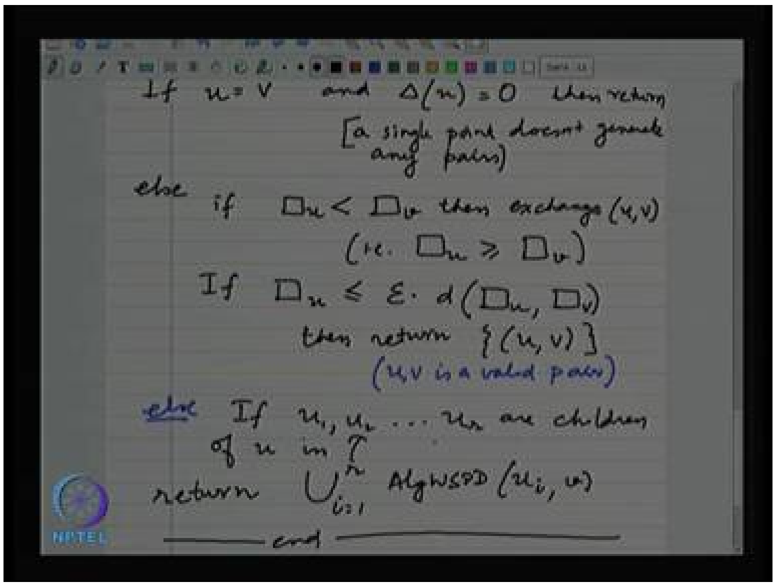
So, what we doing we are given some set of points, we are enclosing the points in some cell which is basically nothing but a square of set in size, suppose I have these two sets of points let us, call it A, and let us call this B. Now, if I have to actually verify whether A and B, satisfies the definition of the epsilon well separated pair and then I should verify the diameter of A let us, now max of diameter of A, and diameter of B, both should be less than the distance between the point sets A and B. So, only this then I will be convinces that S you know this, this constitutes A legal pair of the well separated pair decomposition. Now, instead of that, suppose I do the following I just assume that my the diameter of this point set is a diameter of the cell enclosing the point set A. So, suppose we approximate the diameters of delta A by... So, let me use this notation. So, this is nothing but the cell enclosing points of A.

Similarly you know, I can approximate the diameter of B by the cell enclosing the set of points B, of course, the diameter of A strictly speaking the diameter of A square is probably the distance between the 2 diagonal points, but it is something that you know this the formulas essentially, if I tell you the size n square you know exactly the what is the dimension is the way I do not have to actually d n, a computation for that other than just applying the formula. Similarly, I can approximate the distance between the point sets A, and B by the distance between I can say distance between A and B. I can certainly less than or equal to the distance between the cells and even here maybe I should point out that the diameter of the cell contain the point sets A, actually is greater than equal to the actual diameter of A, because it encloses the set of points. Now, with these approximations you can ensure that. So, if we can ensure that this is less than or equal to epsilon times distance between the cells, this will guarantee that the points are actually well separated. In this general probability that distance between the bounding as A B and it is nearest (( )). Distance between a this point sets is you know some is a point in the boundary, how can be closer than that a way is really the point in the boundary. So, this cell is entire you know, you should think about is the cell is entire I think.

There are at least of one point of A there is no point of A or B, when we talking about cells is basically, the entire square, the area of the square clear (( )). So, if I give you two squares like this, what is the closest point you know it whatever, it has to be point in the boundary of the square, but does not have to be any point of A on the boundary, what is the difference between the two squares, 2 disks. (( )) some point will be. So, I could have also enclose the same thing I could have enclosed it, using a you know some enclosing like this then it would have been the distance between the two disks. So, distance between the two disks has to be less than or equal to the distance between the point sets A and B, because disk actually encloses the point set A and the point set B will be (( )). You absolutely right, first we write.

So, if you can ensure that this …The diameter of this condition holds then I clean that there actually well separated this condition is nothing but delta A over d (A, B). So, we are taking A lower bound on the …And one distance less than or equal to epsilon. So, if this by less than epsilon in this quantity should also be that, because we are chosen A higher numerator, and the larger denominator. (No audio from 31:25 to 31:35), if anything we are actually more strictly imposing this we are proximating it, but approximating it for one end. So, if this term is less than let us, say ours parameter epsilon let say point one then the actual ratio should be certainly less than point one. So, this is how we will avoid actually doing explicit computation of the diameter which is not such an well it is easy in the sense that you know I can take every pair of points and find the distance, but that is n choose to kind of computation and, we do not want to do this n choose to do n kind of computation at many steps of algorithm, you will actually get a much faster algorithm.

(Refer Slide Time: 32:33)



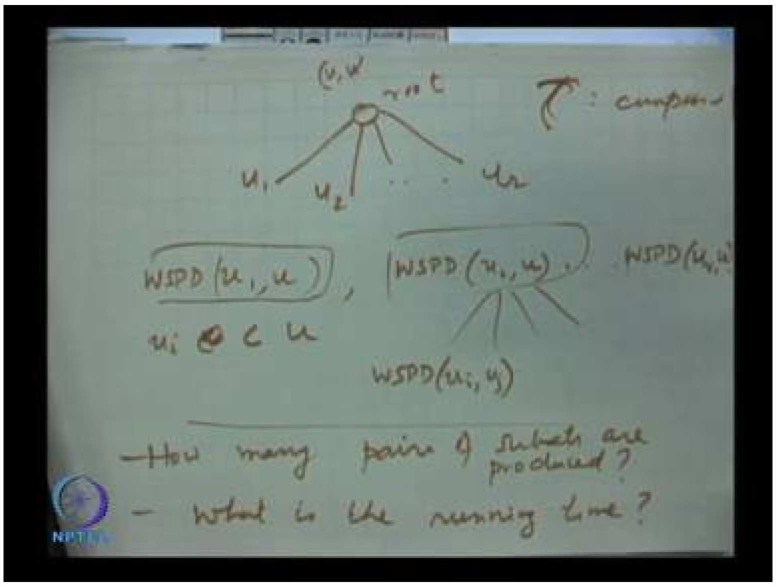
```
If $u = v$ and $\Delta(u) = 0$ then return
        [a single point doesn't generate any pairs)
else if $\square_u < \square_v$ then exchange $(u,v)$
        (i.e. $\square_u \geq \square_v$)
    If $\square_u \leq \varepsilon \cdot d(\square_u, \square_v)$
        then return $\{(u,v)\}$
            ($u,v$ is a valid pair)
else If $u_1, u_2 \ldots u_n$ are children
    of $u$ in $T$
return $\bigcup_{i=1}^{n}$ AlgWSPD$(u_i, v)$
—— end ——
```

So, with this background then I can write down the algorithm. So, you call that tau is our compressed quadtree (No audio from 32:40 to 33:29). So, this algorithm WSPD takes two parameters u and v, these are suppose to be some point sets now, if u equal to v and delta u equal to 0, delta u basically means that the diameter it is a way of writing it is a single singleton point diameter can be 0 only it is singleton point. So, if that is a case then we do not proceed any further. So, it is recursive value. So, it is the termination case basically else (No audio from 34:02 to 34:56) this is just to get the ordering. So, I want make first sub set to have A larger diameter of the second subset that is all, you know anything (No audio from 35:06 to 35:47). So, u, v now becomes one of the pairs of the decomposition pairs satisfies that.

The condition of well separateness (No audio from 36:05 to 36:58). So, we call it as recursively. So, this is the only pairs that we call it recursively. So, if the condition… So, we are… So, starts with two subsets of points you read, initially actually u equal to v equal to S. So, this is my algorithm, the algorithm ends here. So, initially u equal to v equal to these points set S and, we always referring to the compressed quadtree tau. So, what happens? So, you call this algorithm initially and u u the root of the tree, which is the root of the tree.

So, you note that the one singleton point case you know, there nothing really happening this singleton point otherwise, if this two we initially it is u u. So, both of them are same now, clearly this condition cannot be satisfied, just look at the first step this condition cannot be satisfied they cannot be well separated, because u and v are the same set of points; obviously, they cannot well separated. So, what we going to do, we going to call the algorithm recursively on u, all the children of u. Let us look at this.

(Refer Slide Time: 38:32)



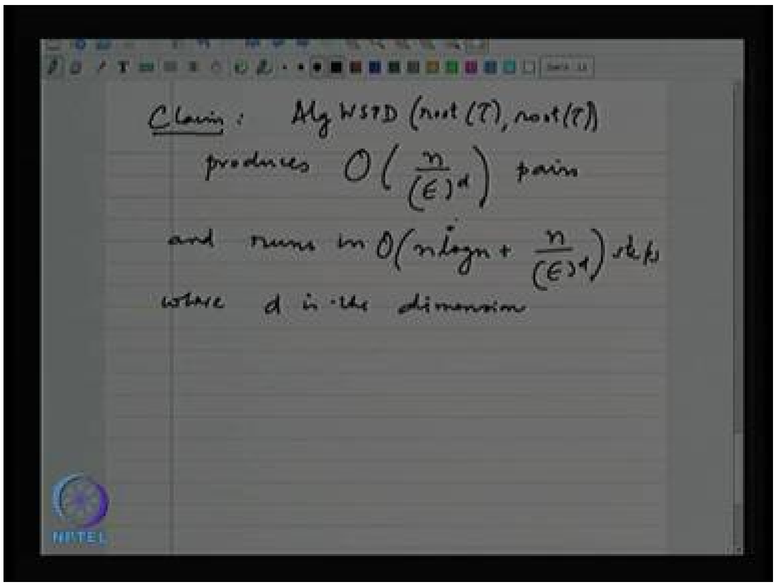
So, here is the root of the tau this is my tau, this is u 1 u 2 u r actually enough the root. So, when we called it initially, we are calling this as u u now that can be well separated. So, we are going to call it recursively on all the pairs u 1 u (No audio from 39:00 to 39:14) this how it is going to practice, after the first level. Now, you can see that again u 1 and u they are not going to be disjoint. So, nothing is happen we even the second level u and u 1 are not disjoint and, because u i is actually a subset of u. So, nothing is really going to happen there, but the next time again you are going to see this one will give raise to let say look at any arbitrary things. So, this one again give raise to things like to WSPD u i u j until pairs like that.

Now, u i and u j may be actually separate I mean disjoint pairs and then you will check whether, or not the condition is satisfied, if not satisfied it will go further into the you know, it will go down the tree basically. So, the algorithm is picking up the recursive calls from this structure of tau, the algorithm is not really proceeding down the tree on that. Not strictly down the tree it is a sequence of recursive calls, and the next set of recursive calls is picked up from the structure of the tree. You can do it in a well structured, well it is not specified you know whether it just that the finally, the WSPD is going to be union of the pairs written by this and this and this and all that in whatever sequence you call, it is not specified in any ordering search.

So, actually it is a very simply stated algorithm, but it may not do simple to visualize what is really going on. Are we computing each node WSPD of u i comma u j? How will going to compute each node. This tree given to us we already done the compressed quadtree. So, tau is a compressed quadtree is a… We are first constructed a tau, that is how p processing after you got tau then this algorithm looking at the structure of tau will you know, it is a recursive algorithm, where the recursive calls are generated depending on the structure of this tree does it terminate really it must every time you know, it is not a well separated pair, we are actually looking at a sub cell of the that.

So, some point that cell is going to become an empty or cell is going to become a singleton eliminated must terminated that point. So, termination is not problem, will it reduce well separated pairs, because it is not going to print out anything till it is well separated. So, only in the step 2 is print out something otherwise, just call it recursively. So, the interesting thing about this algorithm really is how many pairs of subsets are produced and what is the running time. (No audio from 42:16 to 42:30). So, how we are going to argue about these things, we are already done n log n work to construct the tau, I will first write down the claim and then will see we can justify that. (No audio from 42:54 to 43:12). So, we start with the root of this unit root of tau (No audio from 43:18 to 44:32).

(Refer Slide Time: 43:38)



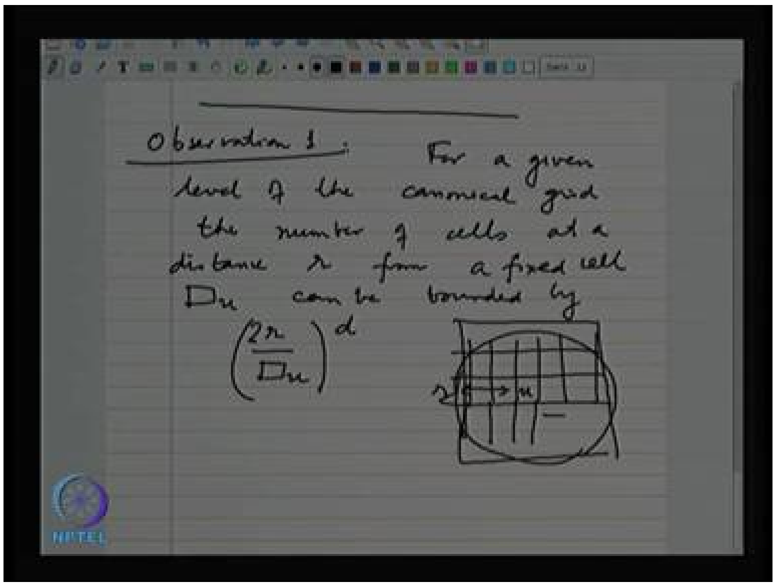
Claim : Alg WSPD $(root(\tau), root(\tau))$ produces $O\left(\frac{n}{(\epsilon)^d}\right)$ pairs and runs in $O\left(n\log n + \frac{n}{(\epsilon)^d}\right)$ steps where d is the dimension

So, if you consider that d is fixed, like you know 2 or 3 or some fixed constant, what it is producing is actually linear number of pairs, and the most in the frequent use of let us say now WSPD we are actually Euclidean spaces of you know dimensions 2 and 3. So, d is a constant in all such applications. So, this actually producing a linear number of pairs, but you know it also depends on epsilon. So, if you want epsilon to be smaller, it is going to basically work harder I mean it will require more time to converge. So, why is that? So, if you want you know points to be set of well separated says that there is really set of small and a further apart there you have to work harder intuitively that should be the case.

I want my diabetes to be small I mean like to you know emission right in the beginning, that you know it is useful applications, where trying to compute some pair wise interactions and those approximations become only better, if you know the points are actually the sets are further apart, because then the approximations you know whatever Taylor's series you do you come out to the better, if the point sets are actually further apart. So, the smaller is specify the epsilon to be in terms of any kind of you know calculations that you do in physics, basically physics or chemistry those the forces, the terms with the lower order terms with the forces etcetera even be smaller.

So, with smaller epsilon actually you get better approximations, and you also have to algorithmically you have to work harder to produce kind of decomposition and this n log n is a… This is coming from the construction of the quadtree. So, after the construction of the quadtree, the time that it takes is only linear in n and of course, you know it is also depends on the epsilon. That is epsilon is not that is small, suppose that epsilon is point 4.5 that kind of thing then you essentially looking at linear time starting from the quadtree, how does one prove this part. So, there is a force you know a very elaborate proof, and I do not want to write out of everything in the class probably give you some pointer may be I will put it on the course with itself that the proof, but let me just discuss the idea of the proof.

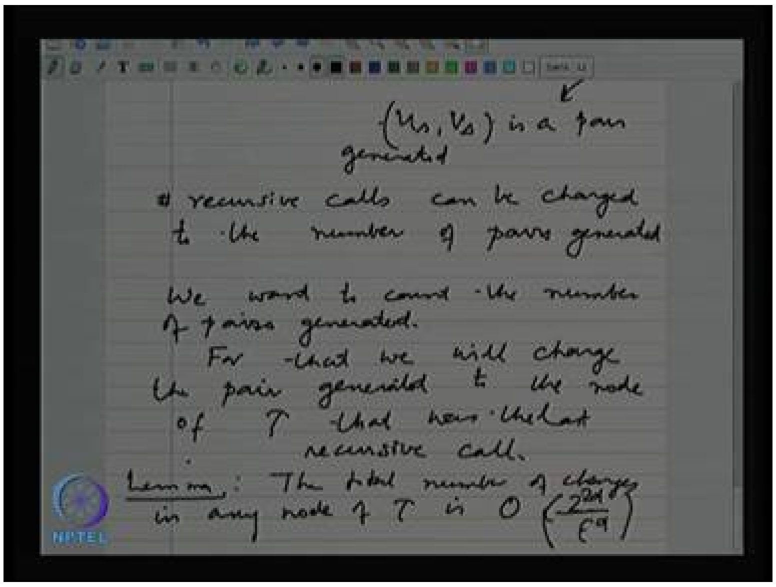
(Refer Slide Time: 47:33)


Observation 1: For a given level of the canonical grid the number of cells at a distance r from a fixed cell D_u can be bounded by

$$\left(\frac{2r}{D_u}\right)^d$$

So, the couple observations that is crucial (No audio from 47:32 to 48:12). So, may the canonical grid is that you know, I take a grid space one and then I take a grid space half, and taking grid space one-forth and so on. So, forth. So, at any fixed level of the grid, the number of cells at a distance d from a fixed cell and p bounded by well d is not a good, I am using d for a dimensions distance let us say r. So, essentially you know, if you look at a 2 dimensional thing always saying is that you know, I fix suppose this is u is the cell containing u, if you go distance deep r from here whatever direction of course, this is nothing but a disk and the number of cells that intersect the space is no more than what is given here, you feeling that I may have to multiply this by 2. So, all I am doing I just bounding it by this thing. So, this distance r. So, it should be only 2r. So, what is to be noted is essentially distance over the size, the cell to be power d in dimension d, where is the observation going to be useful.

(Refer Slide Time: 50:54)



(u_s, v_s) is a pair generated

recursive calls can be charged to the number of pairs generated

We want to count the number of pairs generated.
For that we will charge the pair generated to the node of τ that was the last recursive call.

Lemma: The total number of charges in any node of τ is $O\left(\frac{2^{2d}}{\epsilon^d}\right)$

So, way you want to analyze this is that you look at any sequence of recursive calls, suppose I have (u 1, v 1), (u 2, v 2), (u 3, v 3)… (u s, v s). So, when you go through this, because the sequence recursive calls, the basically means that and we stop here. So, no pairs are generated pair, because of if some pair generated, when you stop it terminates you go further you can desire with the… So, you go further only when you know you do not this 2 pairs of points are not well separated only, then you go deeper into the recursion hence forth you call it on all the children of a certain mode. So, you do not actually generate anything in the middle, at some point you are generating some pairs. So, here you have generated some pairs essentially the (u s, v s) the pair.

(No audio from 52:26 to 52:42) So, what it means is that the… Can I see the following that, the number of recursive calls (No audio from 53:00 to 53:13) can be charged to the number of pairs generated (No audio from 53:28 to 53:44) and you want also somehow. So, we want to basically count the number of pairs generated. We want to count (No audio from 53:56 to 54:10) and to count the number of pairs generated charge the pair generated to the node of tau that was the last called the last recursive call. So, we have the recursive calls are basically structured on the way, the tau is basically structure. So, we… If you do not produce appear, in the next set of recursive calls basically go according to the children of that particular node, where that recursive call occurred. So, whenever we produce the pair, we want to charge it to that node of tau, because of which recursive call this pair was generated and then what will do, we will count the total number of charges accumulated at each node of tau.

So, that will give us the total number of pairs generated, and the key lemma would be that is again stated with the proof here, the total number of charges in any node of tau is order one over well actually to be more precise turns out to be more like 2 to the power 2 times d divided by epsilon to the power d. Now, the 2 to the power 2d is actually ignored in the final bound, because d is consider to be a constant, if you consider d to be fixed, then you can ignore it otherwise this actually the quantity 2 to the power 2d. So, this explanation is another explanation also. So, this has to be argued. So, if you can argue this, then we get the final bound. So, time is up today. So, I will continue this discussion tomorrow.