Functional Programming in Haskell

Prof. Madhavan Mukund and S. P. Suresh

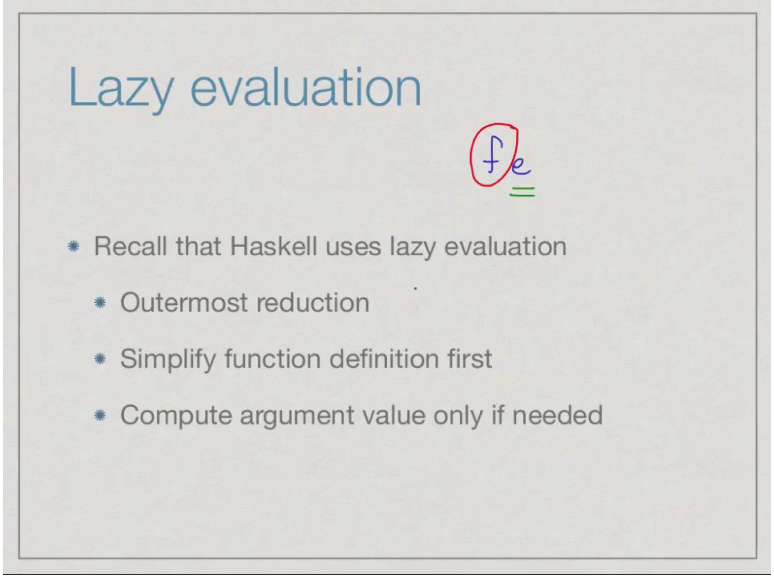
Chennai Mathematical Institute

Module # 04

Lecture – 03

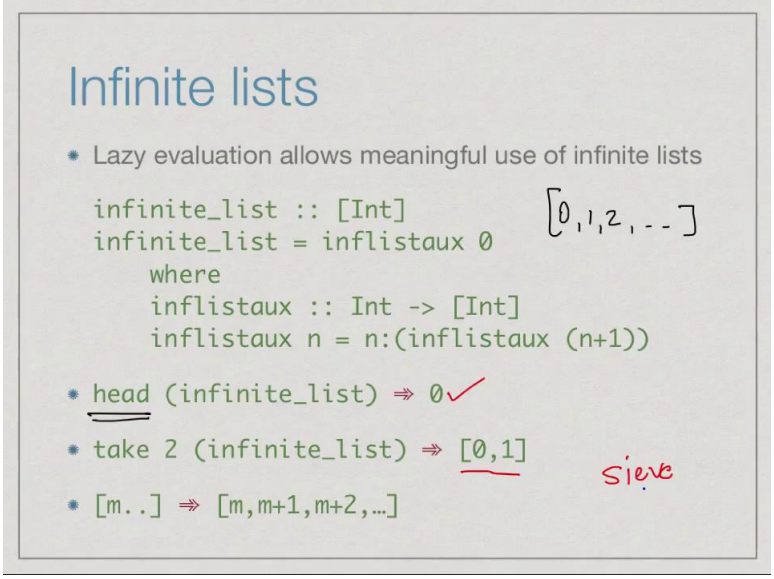
Using Infinite Lists

(Refer Slide Time: 00:01)



We have seen that Haskell uses lazy evaluation. Technically, this is called outermost reduction. So, given an expression of the form f e, Haskell will try to first apply a definition which simplifies the outer function f. So, it will compute this argument e, only if e is required in the expansion of f.

(Refer Slide Time: 00:28)



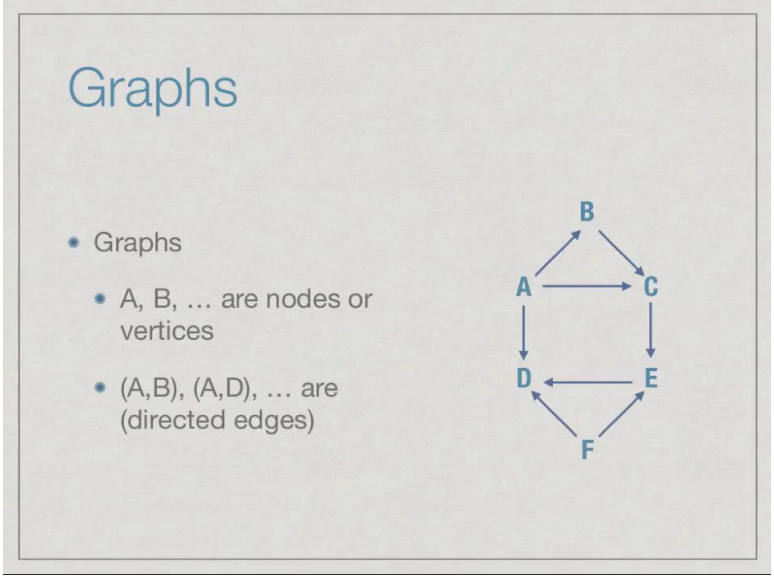
We saw that this allows us to sensibly use infinite list. So, we wrote this function infinite list which generates the list of the form 0, 1, 2 and so on, without stopping. At every point n generates the value n followed by the infinite list, starting at n plus 1. Now, though this is an infinite list and I take an infinite time to compute and therefore, the computation of this infinite list function does not terminate.

We can apply sensible functions to it and get answers in a finite moderate time, for instance if we take the head of an infinite list at this form, it only needs to generate the first element. So, once it has expanded this function once, it finds it has 0 followed by the infinite list starting from 1 and since it has the first element, it can terminate and tell us that the head of this infinite list is 0.

Similarly, if I ask to take the first two elements, it will apply the definition twice and once it has generate two elements, it will return the list 0, 1 without trying to evaluate the entire argument. So, this list range notation that we said from m to n can be used to generate infinite list by leaving out the upper bound. So, if you write m dot, dot, it means the infinite list m, m plus 1, m plus 2 and so on.

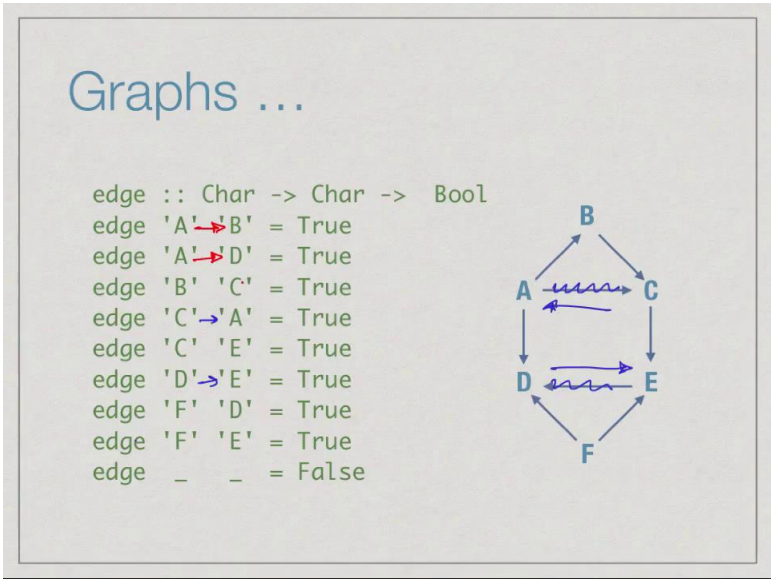
So, so far we saw only one curiosity involving this which is permutation of this sieve algorithm by Eratosthenes for computing all the primes. So, today let us look at a couple of examples which illustrate, why it is sometimes convenient to think in terms of function which generate unbounded list, rather than worry about, how to make them bounded.

(Refer Slide Time: 02:15)



So, our first example comes from a very important class of algorithm set, we use in computer science. So, these are the algorithms on graphs. So, graph is a structure, picture like the one you see here, where you have nodes or vertices, here the node 6 nodes called A, B, C, D, E, F and these nodes are connected by edges. In this particular example, it is a directed graph, so the edges have a direction, you can go from A to B, but you cannot go from B to A. One can also use undirected graphs in which you say A and B are connected without specifying the direction. So, from A you are connected to B, from B are also connected to A.

(Refer Slide Time: 03:00)

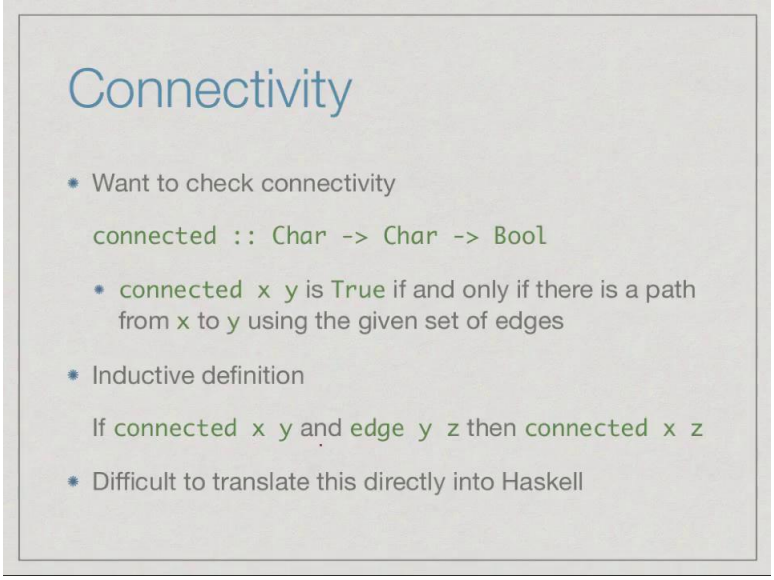


So, continuing with this directed graph, we can represent this directed graph in Haskell

by a function edge, which describes all the edges. So, it says that, there is an edge between A and B for example saying that, this is oriented now, the first argument is starting point of the edge, so this says this is an edge from A to B. Similarly, there is an edge from A to D and so on, so this is probably as known, so these two edges in this should be reverse. So, it should say that, there is an edge from C to A and there is an edge from D to E.

So, this function actually represents this graph note dimension earlier. So, it says just an edge from C to A, but not from A to C, there is an edge from D to E, but not from B to B and so on. So, we exhaustively enumerate all the edges which actually exist in our graph and then in one short, we can say that any other pair of vertices, either shown here or not even shown here. Supposing, I have other vertices called z or w, then any edge involving those vertices also does not exist. So, edge therefore, is a Boolean valued function, which tells us whether a given pair of vertices is connected by an edge or not.

(Refer Slide Time: 04:12)



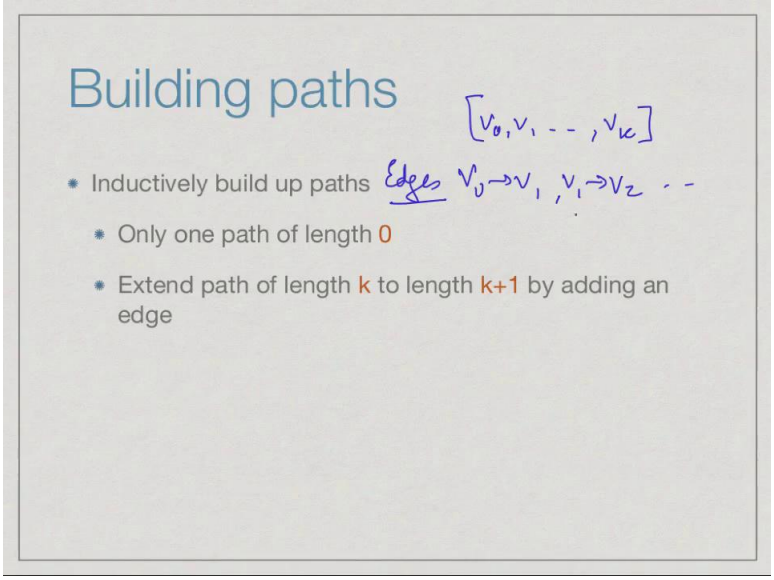
The typical property that you want to evaluate and such a graph is connectivity given two vertices is their path between them. So, using the edges which are there in the graph can be started x and go to y. So, we want to write this kind of the function connected, which takes two input vertices and tells us the answer is true, if it is possible to construct a path from x to y, given the edges specified in the edge function.

So, there is a very nice inductive way of defining connectedness in terms of edges. If we have already a path from x to y and then, we have a single edge from y to z, then

inductively, we have a path from x to z. So, we can define connected x, y in terms connected x, z in terms of the existence of the y, such that x is connected to y and there is an x and y to z.

But, looking for this y, searching for this y is not easy to do in a language like Haskell, this is a natural definition in other forms of programming, such as logic programming. But, in functional programming it is little hard to compute all the y's possible that are needed to check this fact. So, how do we do this?

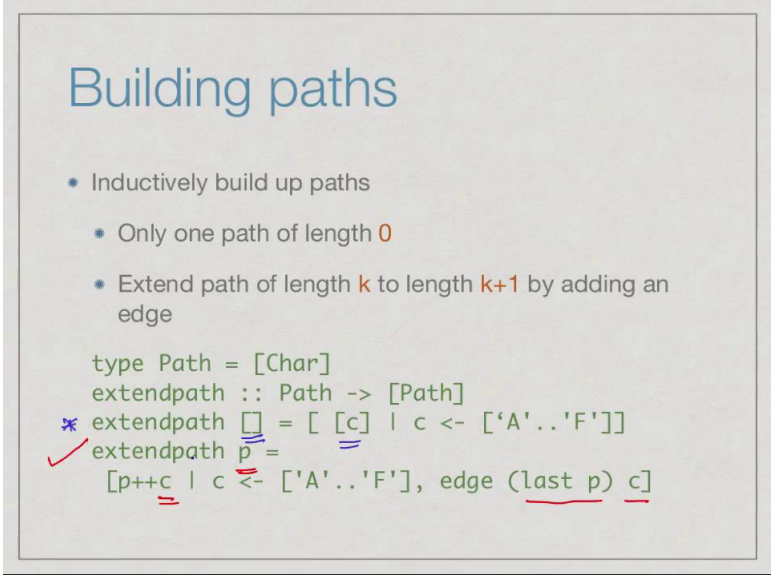
(Refer Slide Time: 05:30)



So, we will follow a different strategy, we will try to inductively build up all the paths that can be constructed in the graph. So, there is only one empty path of length 0, so path is nothing but, a sequence of vertices, a path is can be thought of is the list V 0, V 1 up to V k. Such that V 0 to V 1 is an edge V 1 to V 2 is an edge and so on, so these are all edges.

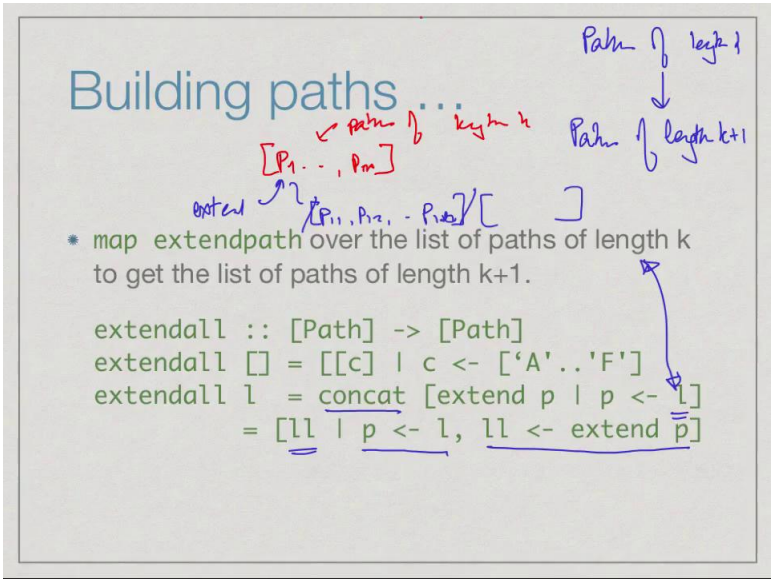
So, path is just a list of vertices in which every vertex has an edge from it is previous element and 2 it is next element. So, we can now take a path of length k. So, notice in the path of length k has k plus 1 vertices, because start somewhere and followed k edges. So, we have a starting vertex and k new vertices. So, you can extend a path of length k to k plus 1, adding an edge.

(Refer Slide Time: 06:28)



So, here is a simple version of this. So, if you have an empty path, then you get paths where you just start with starting vertex. So, this is the kind of a trivial base case and we just put it there convenience, it is not really important. The really important once is the second one, we says that if I have already a path, which is a sequences of vertices, then I can add to it, the new edge provided the last element in p and the new vertex are connected by an edge. So, I can add a new vertex to this path and make it a path of length k plus 1 by adding anything, which is connected to the Haskell. So, this is my extend path function.

(Refer Slide Time: 07:11)

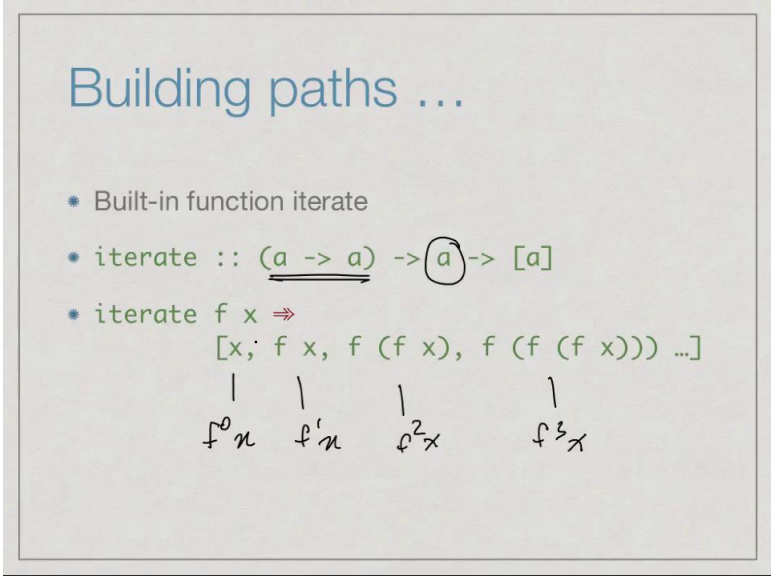


And now, I can build up paths of longer and longer length by just repeatedly extending

this. So, if I first take all paths of length x or length k, so I have P 1 to P m, so these are all paths of length k, then if I map extend on this, then P 1 will generate a list of new paths P 1 1, P 1 2, P 1 n, P 1 l say. So, these are all the extension the P 1, of the similarly P 2 will generate all exemption of P 2. So, these are all new path of length k plus 1. Now, I have this extra level of bracket, so I want to remove them, so I can apply concat.

So, extend all, so l is a list of all paths of length k, for every p and l, I extended it, this gives me a list of extensions, then I dissolve concept of brackets by using concat or I can directly use this comprehension. I can say take every path of length k, take every list that extends that path and now, collect all those list into a new list. So, this gives me a function which takes paths of length k, which is a list and generates paths of length k plus 1.

(Refer Slide Time: 08:42)

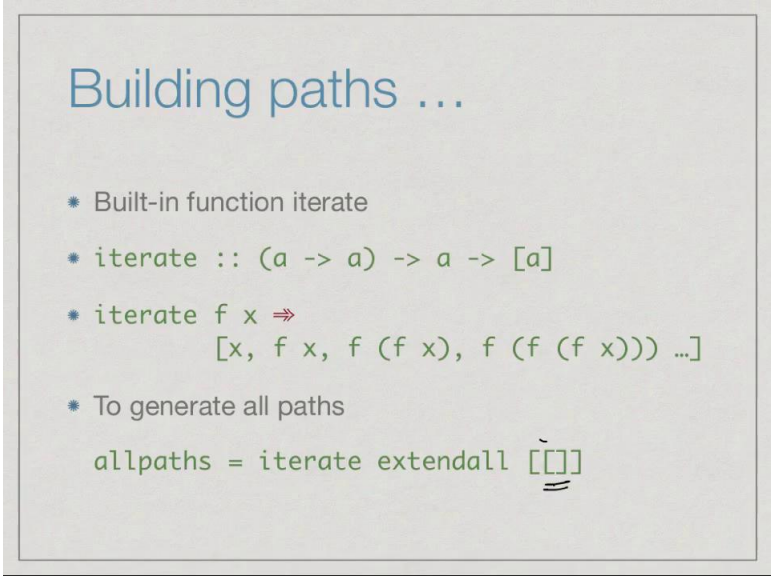


So, now if we want to do is, we want to done this from 0 generate paths of length 1 generate paths of length 2 and so on. And at each point, we have generating using these functions extend all, which takes every path of the previous length and generates paths of one length more. So, we can use a very nice built in function in Haskell call iterate. So, iterate takes a function which has input type equal to the output type.

So, it make sense to call the function on his result, it takes a starting value and produces the list of all values of that function applied 0 or no tags. So, if I say iterate f x, so this is f to the power of 0 of x, this is the f to the 1 of x, this is f squared of x, this is f cubed of x and so on. So, iterate f x gives me a list, f of 0 of x comma f 1 of x comma x square

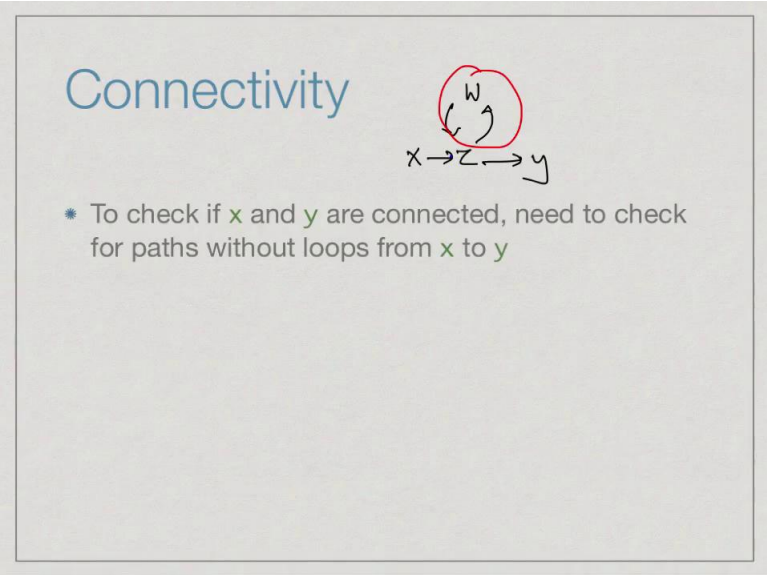
comma f to the power 3 of x and so on.

(Refer Slide Time: 09:38)



So, if you want to generate all the paths that we could possibly get, then what we do is, we start with a list consisting of the unique path of length 0 and we iterate this extend all function. So, extend all will take this unique path length 0 and generate all paths of length 1. So, length 1, actually consist of just a starting vertex, then it will do something useful, it generate from that all paths which have two vertices and therefore, one edge and then all paths three vertices and therefore, two edges and so on.

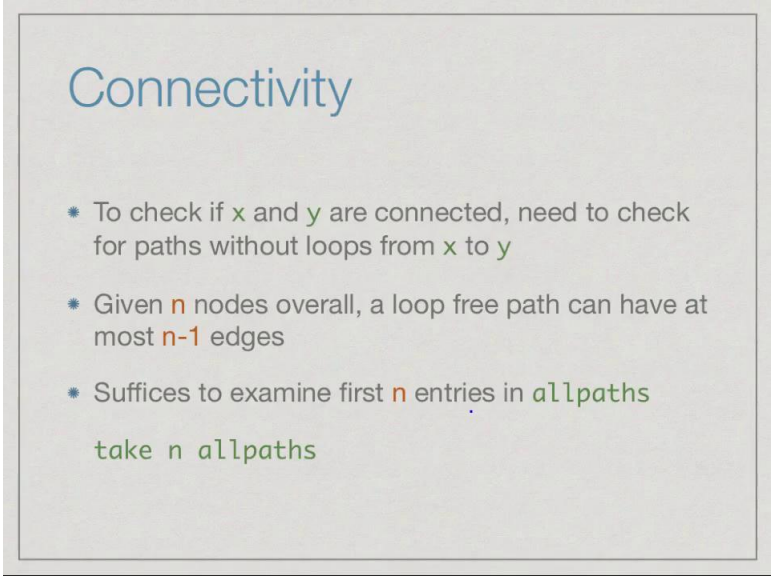
(Refer Slide Time: 10:12)



So, a little bit of thought tells us that to check, if x and y are connected, we need to check

for a path from x to y, but there is no need to for this path to loop. So, if we have a path which goes through z, then go through w, comes back to z and then, goes to y, then this loop is irrelevant, we could directly go from x through z to y.

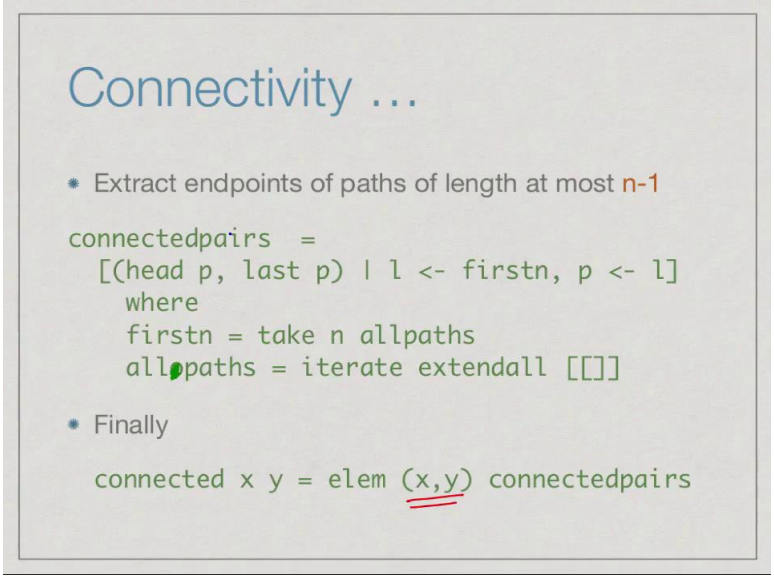
(Refer Slide Time: 10:41)



Now, by a simple counting argument, if we are n nodes, a loop free path can have at most n minus 1 edges, because remember that for n minus 1 edges, I have a numerated n nodes. So, therefore, all the nodes have been used, if I add another edge, I will repeat a node, if I repeat a node, I automatically have a loop. And therefore, that path is not optimal.

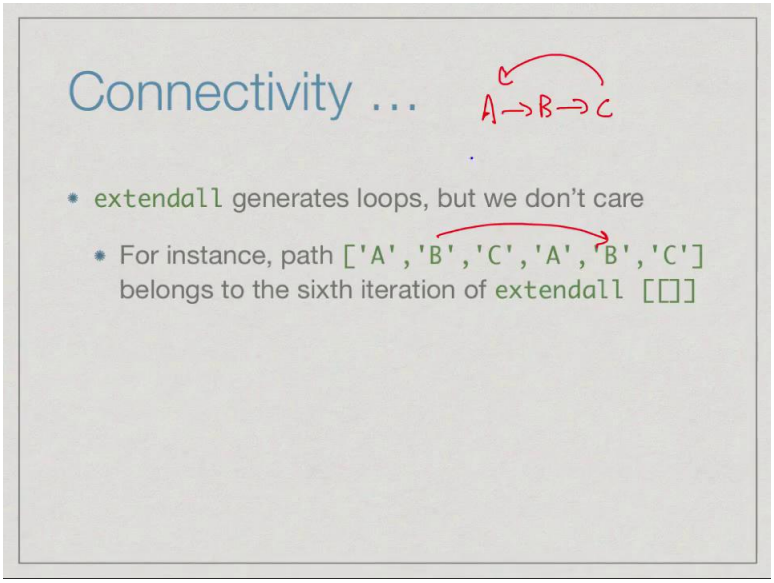
So, I only need to look at paths which have at most n minus 1 edges, so this such that certain order to check connectivity, though my function all paths, which we defined earlier. Generates paths of arbitrary length itself, infinite list of paths of lengths 1, 2, 3, 4; for any length, if we know that, we are only looking at that graph with n vertices, it is surprises to take n entries in this pats, in this list.

(Refer Slide Time: 11:34)



And now, having taken these n entries, so these n entries are generated by we treating are extend all and then, take in the first elements. We just need to take for each path in the final list that we get, the first value and the last value, because this path connects the head to the last value. And finally, to check whether two values are connected, we just have to check whether the pair starting point and ending point is one of the pairs numerated in this list of connected pairs.

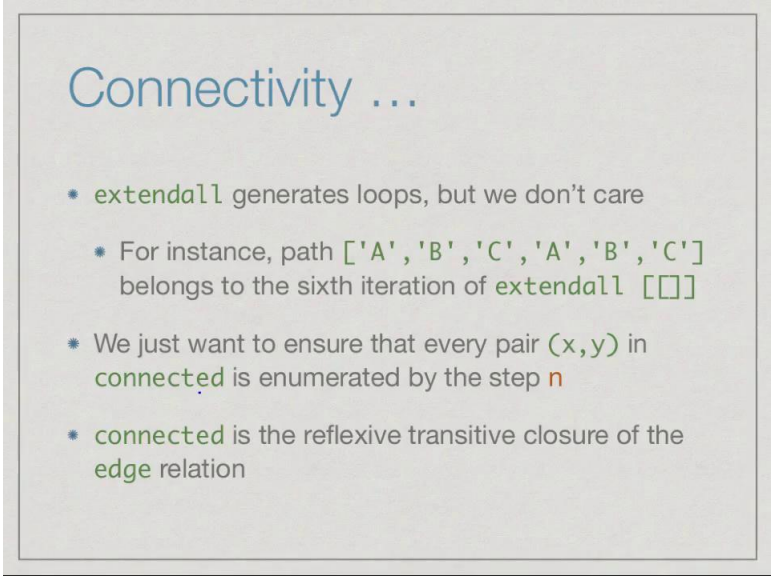
(Refer Slide Time: 12:07)



So, the point to note is that although we worried about, the fact that all not worried about we analyze the fact that loops are not relevant to us. We do not only care of path has loops are not, from the fact that loops are not relevant to us, what we reduce this that we

never need to look at path longer than length l. So, in other words, even though we could have paths of length 6 for example, which have a loop, so this has a loop, which take me back from B to B. So, this is actually the path A, B, C, but it goes A to B to C, a back to A, B and then to C again. So, though we have such paths, it does not really matter to us.

(Refer Slide Time: 12:51)

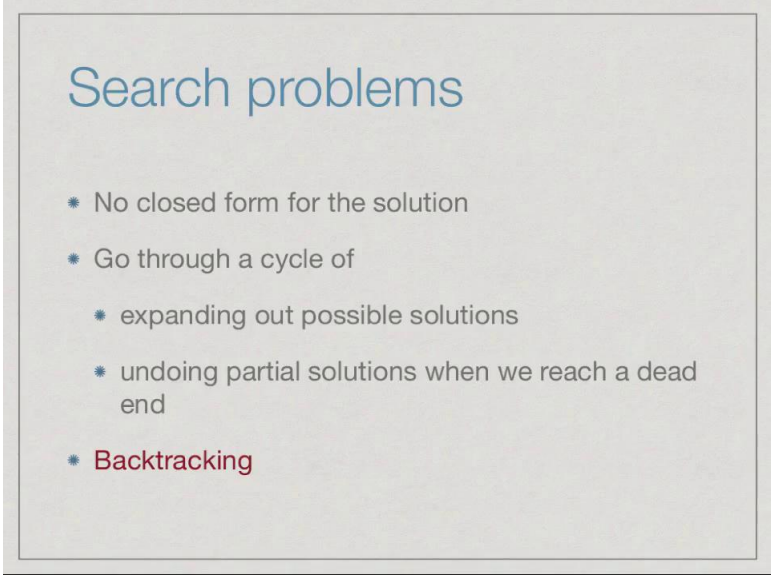


Always saying is that, every path that we really need to look at this finally, present by step n, always looking for is a piece of evidence that I give an x and y are connectivity, it does not matter they connected with multiple ways. On the other hand, we do know that, if there are not a numerate by step n, there is no way to numerate them, because no loop free path in exist.

On more technical way of saying this is that the connected relation is the reflexive and transitive closure of the edge relation. So, in general, if we have any relation which is given to us by Haskell function like we wrote edge which tells us which pairs are in the relation, which pairs or not. If we find want to compute a relation which computes sequences of such pairs, which are connected one of to the other, this is the transitive reflexive transitive closure.

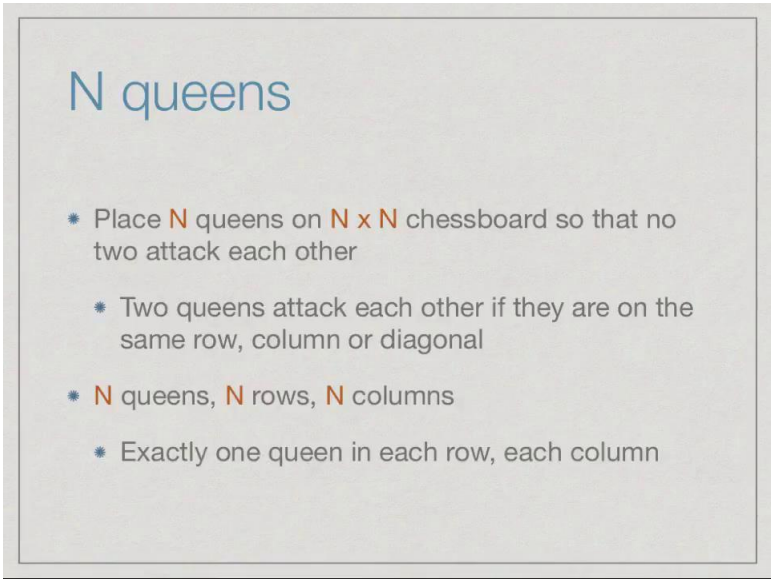
So, every edge is connected and everything which is connected follow by an edge is also connected. Then, we can use this trick that we have of generating all possible length paths and then, using some external criteria to decide on a maximum length path that one needs in order to capture whether or not something is connected.

(Refer Slide Time: 14:00)



So, in general a class of problems for which this infinite list technology is useful or they so call search problems. So, we have a space of solutions that we can generate in some form and we are looking for a particular solution with the property that we require among this space of solutions. So, what in general we would do is you keep expanding solutions and then, when we reach us, we are looking for a certain pattern, either we find the pattern or we may find the solution. We are currently expanding has no expansions possible, so we go back and try another one and this is a general strategy call backtracking.

(Refer Slide Time: 14:44)



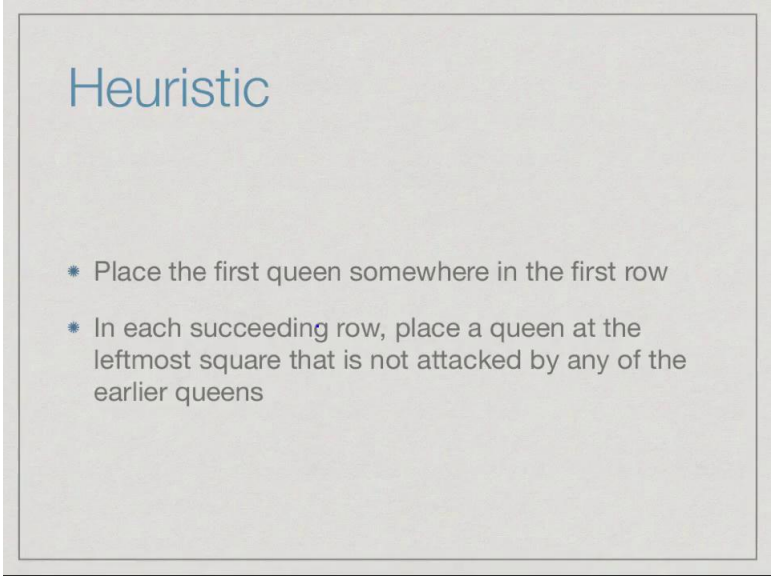
So, one of the most famous problem is which is use to illustrate backtracking is that of

placing N queens on N by N chessboard. So, normally we work with 8 by 8 chessboard, just this paid on 8 by 8 chessboard and of course, you are running one queen. The rule for queen is that a queen can attack any other piece which is on the same diagonal, same row or same column.

So, now we imagine that we are eight queens place on 8 by 8 chessboards, such that no to queens attack each other. So, in particular because any two queens on the same row and the same column will attack each other, each queen must be a different row and different column, but we have exactly as many queens and columns. So, we have N queens and we have N rows and N columns, a simple argument tells us that we must have exactly one queen at each row and an each column.

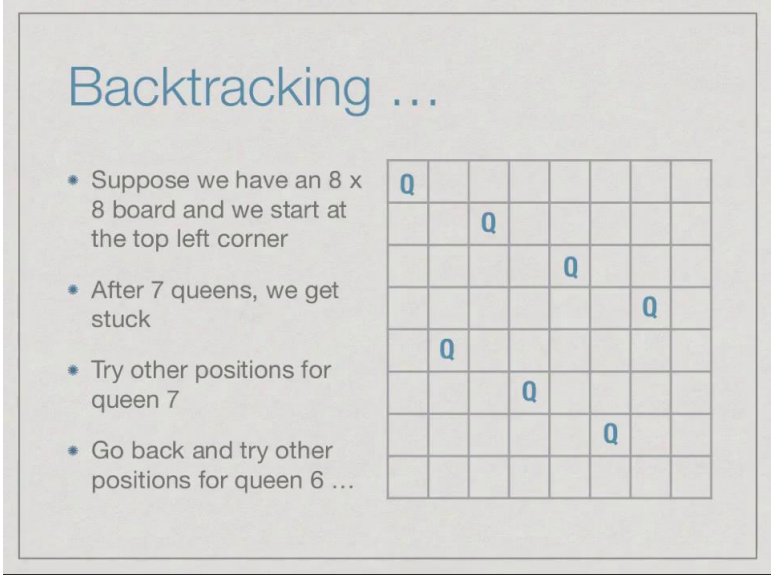
Because, we have two queens in many rows, then we have a problem. If we have no queens in some row or column, then we must have two queens in some other row or column and we have a problem.

(Refer Slide Time: 15:46)



So, here is a heuristic to generate all possible solutions, you place the first queen somewhere in the first row and an each succeeding row, you place a queen at the left most square; that is not attack by any of the earlier queens.

(Refer Slide Time: 16:02)

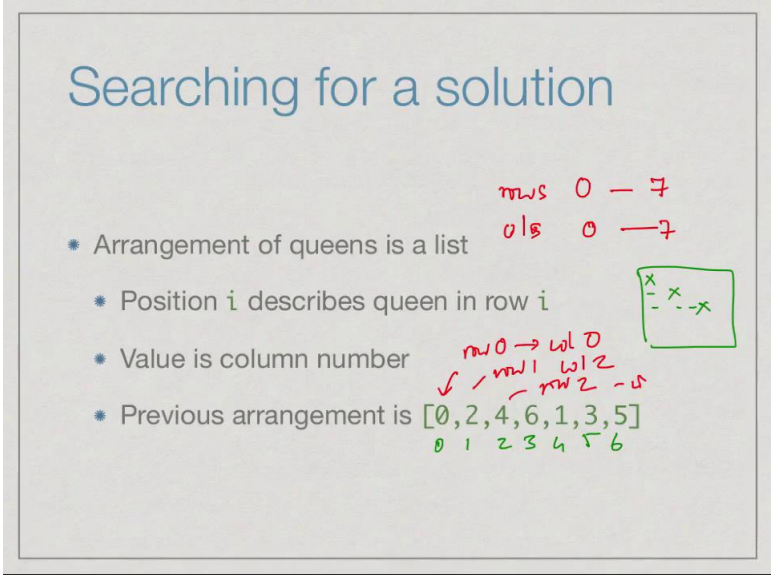


So, suppose if we have a 8 by 8 chessboard and we start with the queen of the top left corner. Now, we are to places second queen and this row, now this is in the same column as the previous queen, this in the same diagonal. So, the first place that we can put a queen is here, so we place a queen there. Now, we will find that these squares are all ruled out, so this is attack by the first queen, this attack by the second queen.

So, now, we come to the third row in the only place we can put a queen starting from the left is here the first place within input. So, if put a queen there and continue this we come to put a queen on the 4th row, a queen on the 5th row and queen on the 6th row, queen on the 7th row. And now, unfortunately we cannot put a queen here which is the normal place, because it is a same diagonal as the very first queen we put. So, we are stuck. So, we cannot generate a potential solution.

So, what we have to do at this point is backtrack, we have to take this and say that, at this point the last thing, we did was put a 7th queen, so can we put it somewhere else. So, we backtrack and then, if we cannot do anything as we backtrack one more we change this value and so on. So, this is what backtracking means you generate resolution as for as you can go, if you are lucky generate a good one, if you are unlucky you get stuck. If so, you go back and undo the last thing you did retry that value, if you cannot get anything for all possible choice, you that second last value go back to the third last value and so on.

(Refer Slide Time: 17:42)

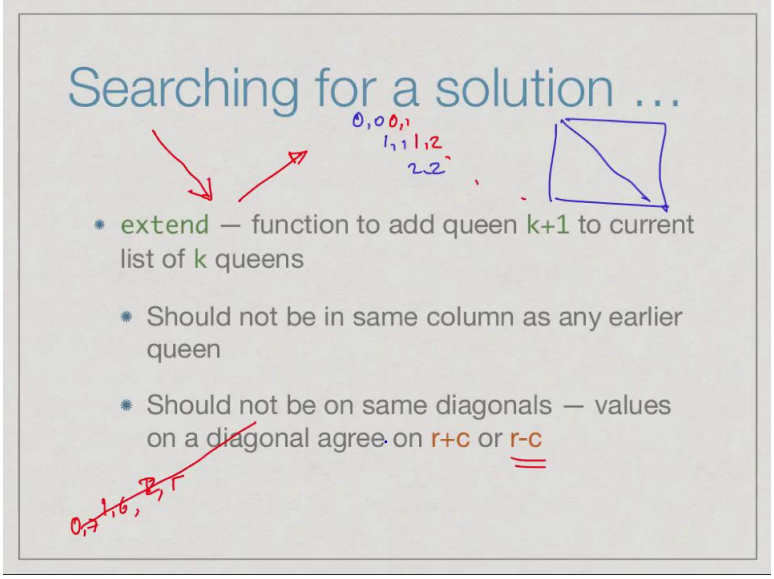


So, what we want to do is use infinite list in a way that we do not have to explicitly worry about backtracking. So, the first thing we need is where of somewhere representing the state of a board with some queens on it. since, we are going from the first row to the last row, we can always describe the current queens on the board in terms of a list.

So, let us because of the Haskell terminology, number this case the eight queens, the rows as 0 to 7 and the columns as also 0 to 7. So, there are 8 rows and 8 columns which are number 0 to 7. So, the list position 0 for first 2 row 0, where is the queen in row 0, so, it will be a number between 0 and 7 again, because it is the column of that. So, this says that the queen and row 0 is in column 0.

This says that queen and row 1, because is a position, so these are the positions are 0, 1, 2, 3, 4, 5, 6. So, row 1 is a column 2, row 2 is a column 4 and so on. So, this is saying exactly what we said before which is that we have on the top left corner we have a queen and then, we skip 2 and then, we have a queen that we skip 2 more and we have a queen and so on. So, this is a representation of the seven queen that we manage to place before we got stuck.

(Refer Slide Time: 19:09)



And now, when we want to place the k plus 1 queen, we have to take a list of the first k queens and add 1 new queen, a new position for the new element, we have to extend the list by one element. The new number is the column position of the new queen, now the new queen should not be, it is by simplicity, because that the new position, it is a new rows, it is not in the same row as any of the earlier column are queens. But, we should not put it in the same column, because if it is same column, it will be attack by a previous queen.

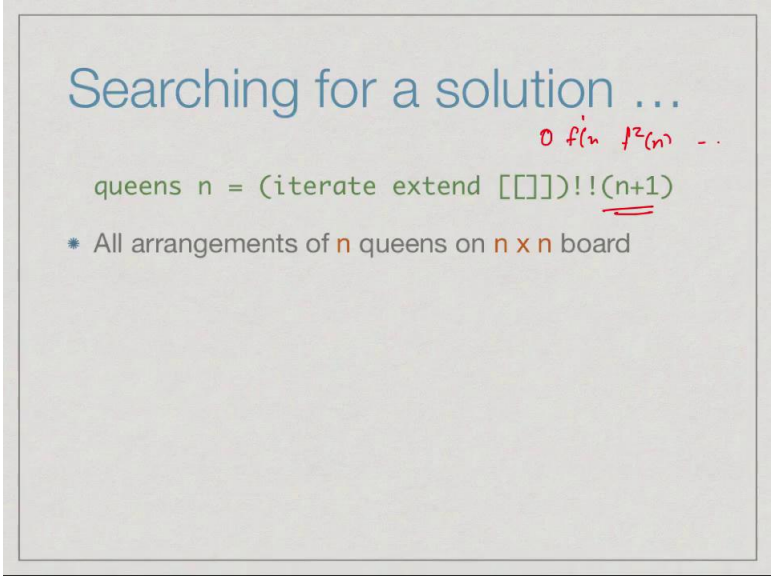
And we also need to check, it is not on the same diagonal. Now, it terms out the diagonals, you can do a little bit an arithmetic. So, if we have a diagonal of this form, so for example, we are the diagonal 0 0, 1 1, 2 2 and so on, all of these are the common difference of 0. If I look other diagonal starting at say 0 1, 1 2 and so on, then all of these are common difference of 1.

So, if we have a diagonal of this form going from top right left to bottom right, then all empties on the diagonal, we have a same r minus c value. Symmetrically, if we do all diagonals are look like this, then you will have the same sum. So, for instance, we have 0 7, then 1 6, 5, 2 5 and so on, so all these are on one diagonal and they are added to 7. So, it is quite easy to check, whether it queen using the same column as an earlier queen, because of the check the number is not present in their list you have so for.

It check on the same diagonal just have to add the position to column number and make sure or subtract to position from the column number and make sure that back difference

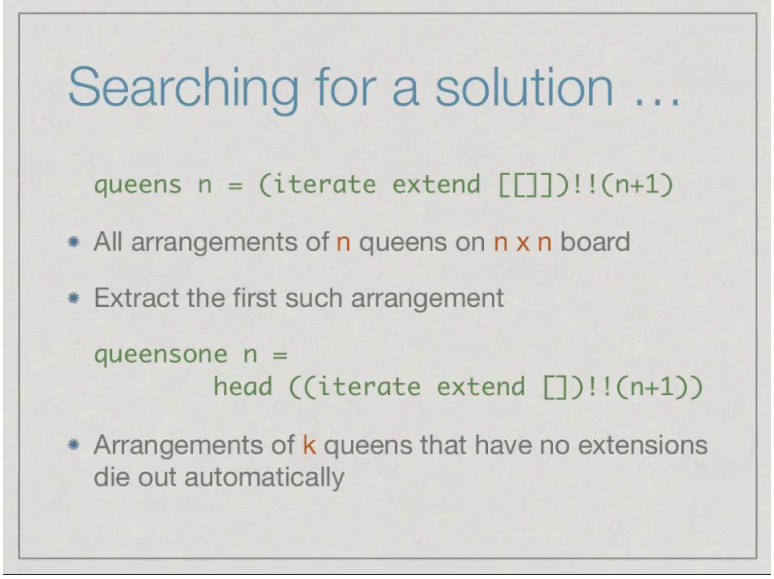
are that sum is not seeing before. So, this extend function we will not write it explicitly that it is easy to right such an extend function which takes a given list of k queens and returns if possible a k plus 1 position for the new queen.

(Refer Slide Time: 20:56)



So, now as we did with graph, we can start with that queen a position which has no queens on it and iteratively extend it and we know that, we have to place n queens. So, if we extend this n times and we end up with the first one is the empty board, the second is all possible configurations with one queen where it is all possible configuration is two queens and so on. And so the n plus 1th entry will be the list of all configuration and which we have placed all n queens. So, n plus 1 because I start from 0, so this is f, f of n, f 2 of n and so on. So, at the end, I will have extended n times are place n queens.

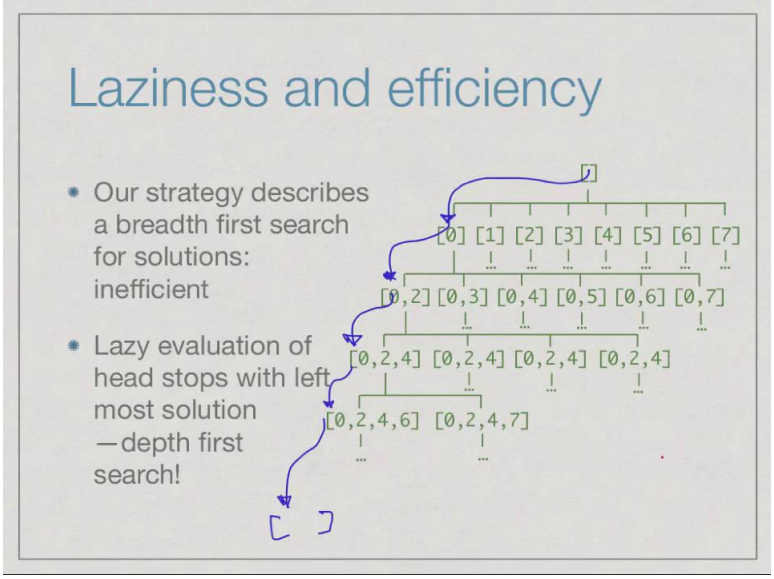
(Refer Slide Time: 21:45)



And now, we are not said anything about which solution we want. So, it is enough to compute say the first element of that. So, you want at least one solution of the n queens. So, you iteratively extend the empty board, until you get all n queens are it and just take the head of that list. So, that will be a list of all possible arrangements of n queens, which do not attack each other and we just take the first one.

So, notice that in the backtracking problem is avoided implicitly, if we come up with the arrangement of k queens, where k is smaller than n and there is no way to extended. Then, the extend function will just reduce that one empty list, so that can we get the solution is just disappear. So, what we doing is rather than going back and generating a new solution, we are in one short generating all possible candidate solutions, those which dialogue die out, those which do not die out or die and we check which all solutions that survive in n plus 1 steps.

(Refer Slide Time: 22:42)

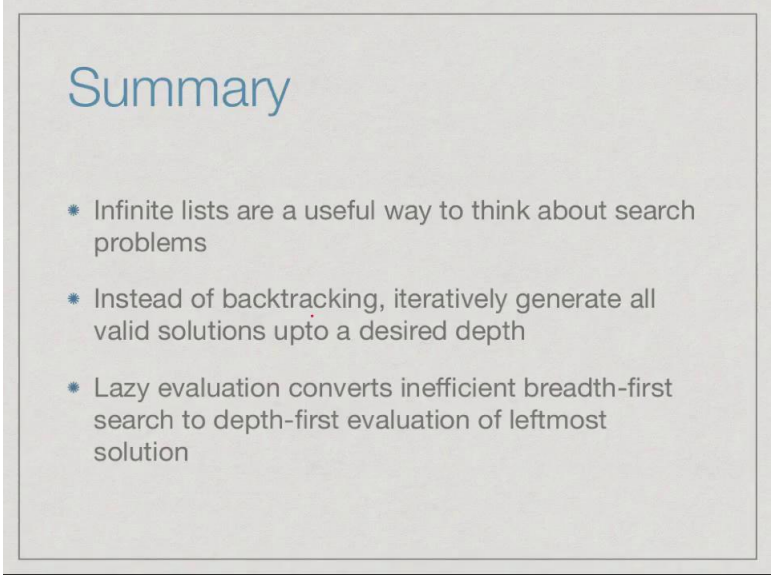


So, in a sense all solution consists of exploring the graph of all possible arrangements in a breadth first way. So, we generate all possible solutions, where we are place one queen in the first row, then for each of these we expanded. So, we get a second level of the this tree, with generate all possible solutions of two queens in the first 2 rows, three queens in the first 3 rows and so on.

So, this looks like a fairly inefficient strategy, because we have going to first compute all the solutions are 1 row, then all the solutions 2 rows and all the solutions 3 rows in order to reach finally, the first solution of the 8h row. But, this is where now lazy evaluation will actually do the job first, because what lazy evaluation will do is, I will say, if I want the first solution, let me see the first solution comes from the first solution in the first row. Then, let me see that, solutions comes on the first expansion of that, then let me see the first come from that.

So, actually though we are programming a breadth first search over this three of all possible solutions, lazy evaluation actually does it first search and finds the first solution without evaluating off. Of course, if I all the solutions I have no choice that evaluate everything, but I am only interested in one solution any one solution according to a numeration I have. Then, lazy evaluation does depth first search and effectively finds as the first solution without trying to compute all the solutions we get.

(Refer Slide Time: 24:12)



So, we have seen that infinite list are more than just curiosity of lazy evaluation, there actually useful way to think about search problems. So, instead of warning about backtracking and deciding, when something has a particular property, we can hetaeristically generate all solutions up to a decide depth and then, ask how many search solutions are there or pick 1.

And in particular, we saw that lazy evaluation converts what seemingly is inefficient breadth first search over the tree of all solutions into a depth first evaluation of the left most solution. So, actually lazy evaluation buys us and efficient to way to execute or rather inefficient strategy.