Artificial Intelligence: Search Methods for Problem Solving

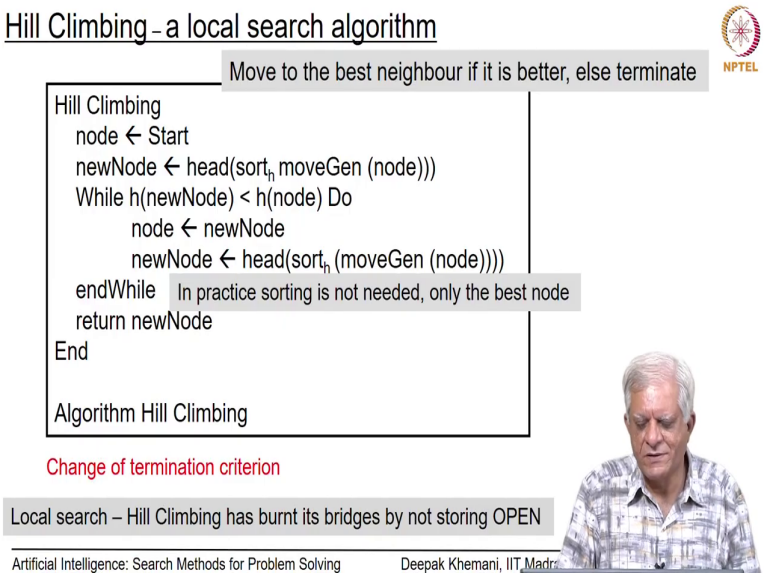
Prof. Deepak Khemani
Department of Computer Science & Engineering
Indian Institute of Technology, Madras

Chapter – 03
A First Course in Artificial Intelligence
Lecture – 25
Solution Space Search: Escaping Local Optima

So, welcome back. In the last session, we saw that one of the problems with Hill Climbing is that it get stuck in a local optima or local maxima or local minima. And as promised we are going to start looking at algorithms which try to escape from such local optima. In the process, we will also look at this notion of state solution space search.

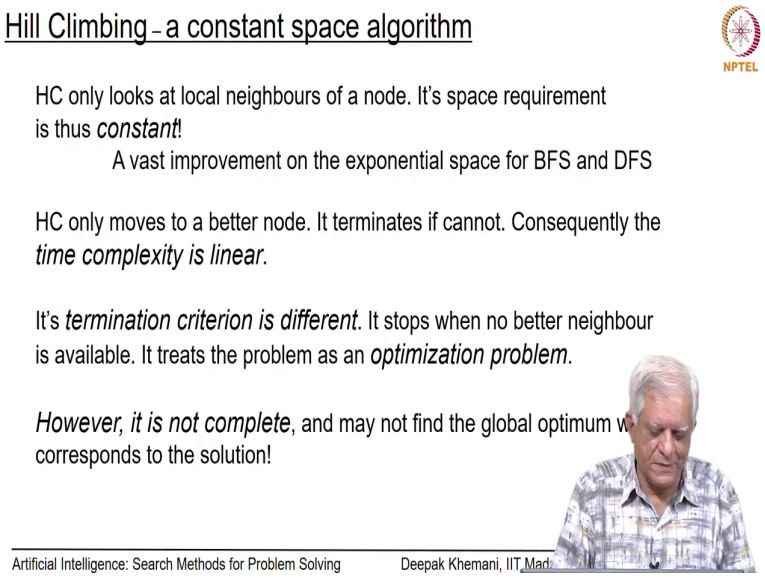
And the difference between state space search and solution space search is that in state space every node in the search space was a state, and you could move from one state to another. Whereas, in solution space search every node in the search space is going to be a candidate solution, and essentially we would be searching through candidate solutions looking for a node which is actually a solution essentially, and such a approach is called solution space or plan space search.

(Refer Slide Time: 01:21)



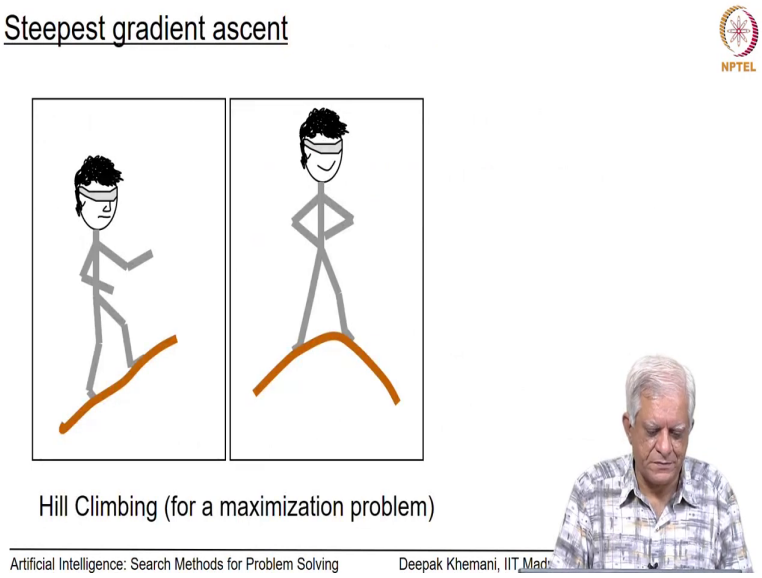
This is the algorithm that we saw in the last sessions. And this is just to recap that what Hill Climbing does is that it moves from a given node to the best neighbor if it is better; else it terminates. And we are observe that we do not need to do sorting we can just simply pick the best neighbour and we had also observed that the termination criteria has changed, and it is a local algorithm which means that it has burnt its bridges from all the nodes that it had generated earlier essentially.

(Refer Slide Time: 02:01)



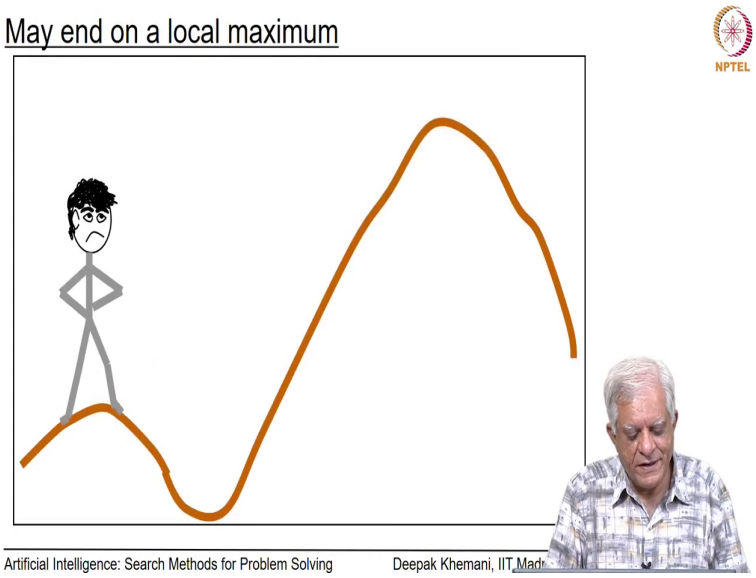
And we had made this observation that it is a constant space algorithm which is good and that it will terminate also reasonably fast because it just goes up the gradient and stops when the gradient becomes 0, and that is a termination criteria that it stops when the termination is gradient is become 0. And in that sense we are treating it as an optimization problem. However, it is not complete because it often get stuck on a local optima.

(Refer Slide Time: 02:33)

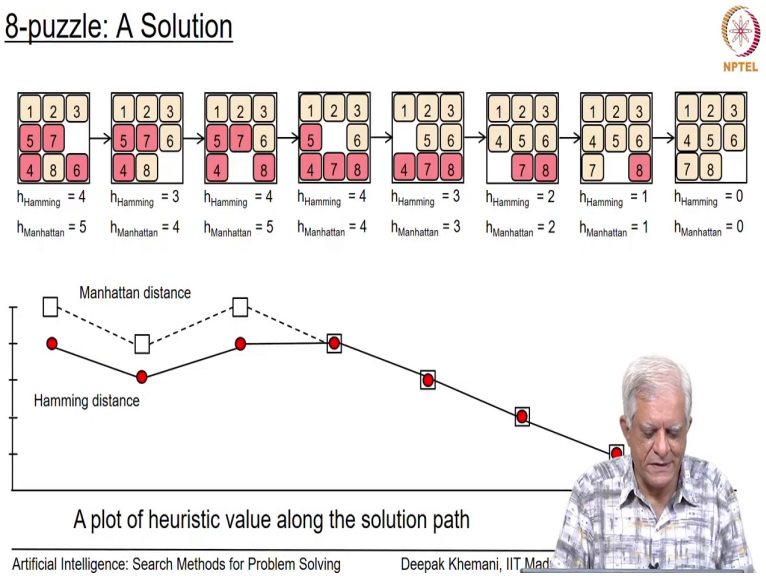


And this is a figure that we had seen a few thinking that you have reached an optima in this case a maximum.

(Refer Slide Time: 02:41)

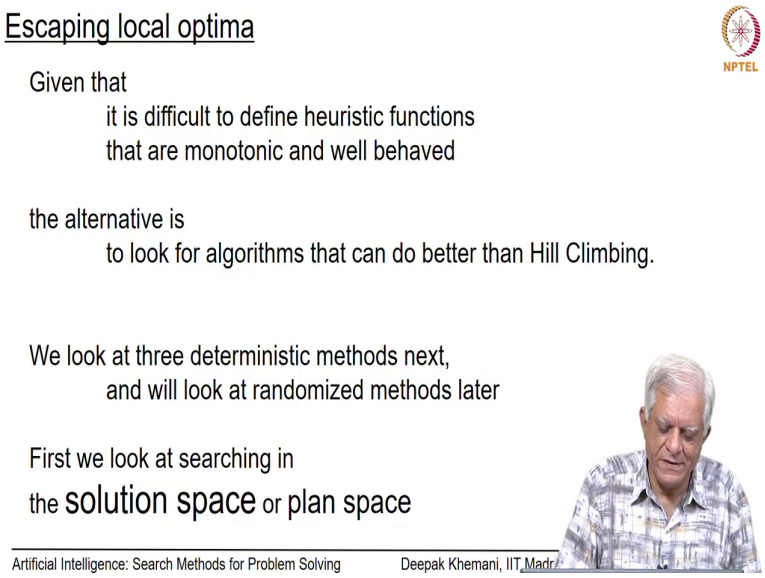


(Refer Slide Time: 02:45)



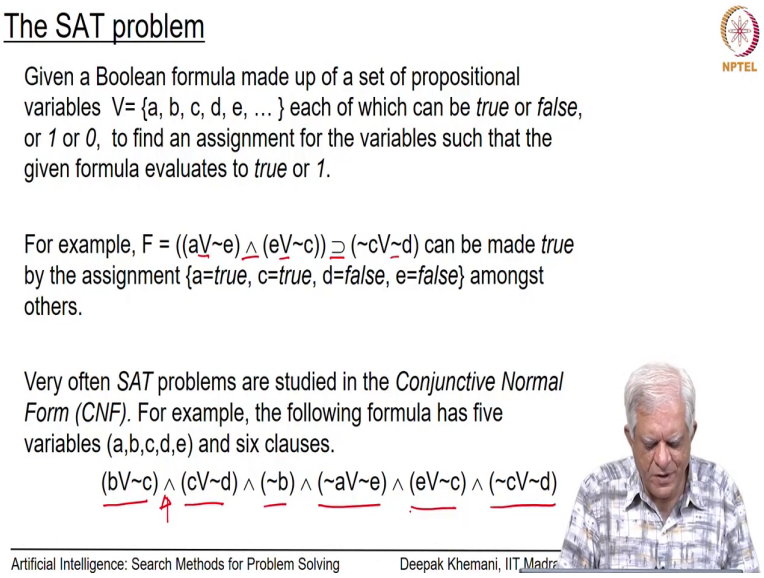
And finding that it is not the global, but only a local maximum essentially. And this is an example that we had seen in the state space where we saw that the surface that the algorithm is traversing over is not monotonic, and therefore, there is a possibility of getting stuck at a local minima in this case.

(Refer Slide Time: 03:03)



And we had said that given that we cannot change the heuristic function we will look at that approach also a little bit later. But the alternative is to look for algorithms that can do better than Hill Climbing, and we had promised to look at deterministic methods and that is what we will start off by doing in this session ok.

(Refer Slide Time: 03:32)



So, what is the solution space or the plan space? Let us begin by looking at that. And we will look at this famous SAT or satisfiability problem in which this is this comes naturally.

So, observe that we have talked about SAT earlier and we had classified it as a configuration problem. In a sense that we are looking for a particular node which is a solution, and we are not really looking at a path of going from some given start node to the goal node. So, we are only interested in a node which satisfies certain properties and SAT and SAT is a typical example of that essentially ok.

So, just to give a quick definition. Given a Boolean formula, a Boolean formula is made up of variables which can take value 0 or 1 made up of a set of propositional variables we will call them a, b, c, d, e, each of which can take a value of 0 or 1, or false and true as the case may be. Both are equivalent. We often treat 0 as false, and 1 as true. So, we want to look for an

assignment for the variable such that the given formula evaluates to true or to 1 essentially, so that is a satisfiability problem.

It is a very famous problem. We will spend a little bit of time on this. And it is very interesting because it was shown to be one of the harder problems by Garey and Johnson in 1983 I think. And they showed that the SAT problem can be used to define a class complexity class of problems which we now called as np complete set. And this is a canonical problem in the NP complete set. We will come back to this later again.

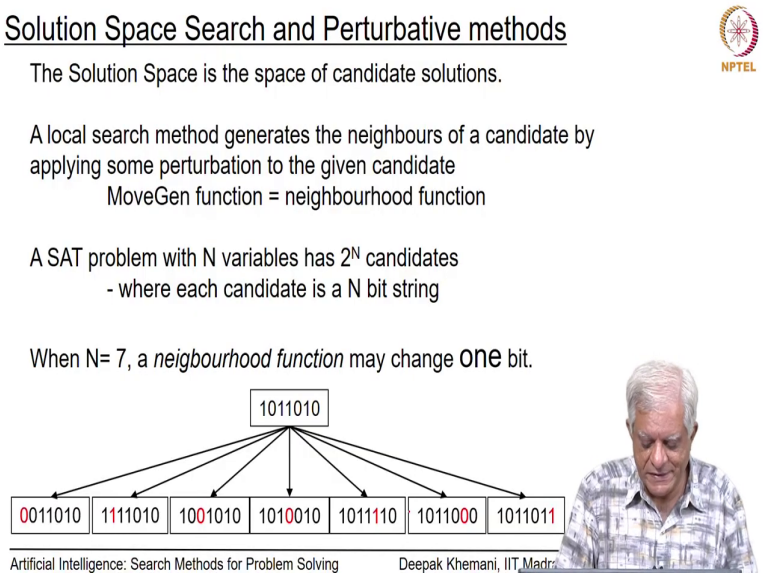
So, here is a small example of a formula on the variables a, b, c, d and e. So, we have a OR not e and then we have e OR not c implies not c OR not d. And what you see other the other things are the logical connective.

So, the OR, AND, the and the implication sign and the negation sign essentially. Now, this formula can be made true by the assignment a is equal to true, c is equal to true, d is equal to false and e is equal to false amongst other assignments essentially.

Very often we work with SAT in the conjunctive normal form which is the CNF form. And the CNF form is a set of clauses which I am underlining now. And the clauses are joined together by the AND connective as shown here. Within each clause we have the OR connector. And so each clause is a disjunct of some expressions, and each expression is either the proposition or the negation of a proposition.

So, in this CNF form the negation sign is pushed inner most, and those of you who have studied logic or Boolean circuits you know that we can always convert any formula into a CNF form. And very often when doing experiments we tend to work with CNF forms essentially. It may be also interesting to note that when you convert a formula into a CNF formula, we often you grow the size of the formula, sometimes exponentially. And after that we have to talk about the complexity of the algorithms.

(Refer Slide Time: 07:16)



So, in solution space what we do is that you work with methods which are called perturbative methods which means you look at a candidate solution and perturb the solution a little bit. So, the solution space is a space of candidate solutions. A local search method generates a neighbours of a candidate by applying some perturbation to the given candidate.

So, our MoveGen function now is becomes a neighborhood function that what can you do to change the candidate a little bit essentially. And a lot of what we will do in the next couple of weeks is going to look at this neighborhood function which says that you take a candidate and change it in some way and hope that leads you to a solution candidates essentially.

Now, if you look at the SAT problem, if you have N variables, then you will have 2 raise to N different candidates because for each of those variables you can give a value of 0 or 1, and there are two there are N variables.

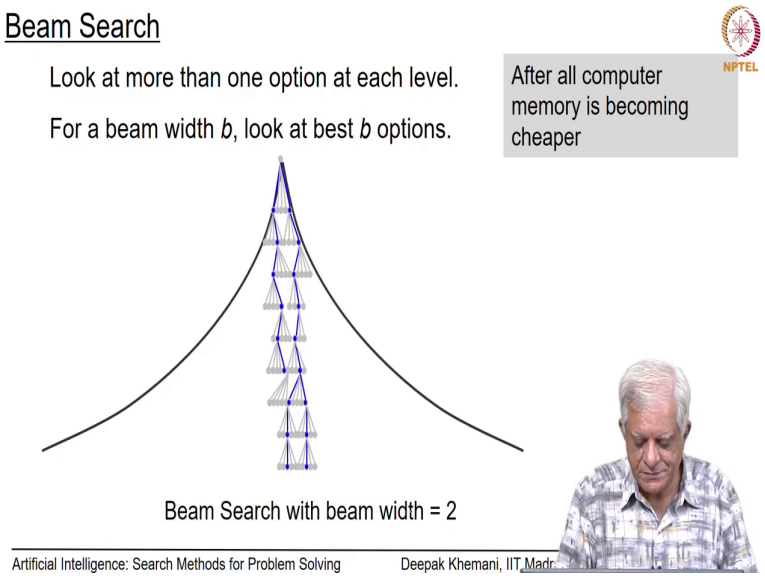
So, the number of candidates is 2 raise to N which is quite a large number. We will kind of try to get a sense of this larger number a little bit later again, but just imagine that if you have 100 variables, then you have 2 raise to 100 possible candidates which is a significant number.

Now, each candidate is a N bit string, because there are N variables and you want to either give a value of 1 or 0. So, what you see here is a an example of a candidate which has got 7 bits, and therefore, this is a problem with seven variables. And we what we are saying in this candidate is that the first variable is 1; the second variable is 0; the third variable is 1; the fourth variable is 1; the fifth one is 0; the sixth one is 1; and the last one is 0 essentially.

So, what is a, what do we mean by perturbation? What is the kind of change we can make? So, one of the things we can do is to say that you can change one bit to generate a new candidate, either you can change the first bit, or the second bit, or the third bit, or the forth bit and so on. And therefore, we have a set of 7 neighbours for this particular example. And what you see in red in each of those candidates is the bit that has been changed coming back coming from the given candidate.

Observe of course, that this is a reversible process that you could have gone back from this to the first one. But since we are doing Hill Climbing, we will not be interested in going back because you can only want to go back to moves which are to states which are better or to candidates which are better than the current candidate.

(Refer Slide Time: 10:08)



So, one thing that you could do is to look at more than one option at each level. Now, why did we move to Hill Climbing? We move to Hill Climbing to save on space essentially. We said that we cannot afford to have exponential space and exponential time algorithms which was best first search for example.

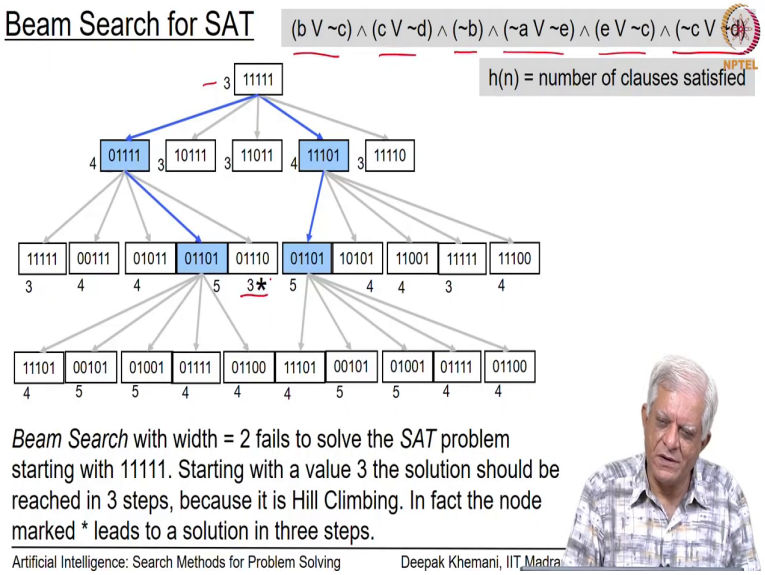
And we said that Hill Climbing is very nice, because it gives takes a constant amount of space essentially. But nowadays given that memory is becoming cheaper, and it is available in abundance, we can generalize from there and say that instead of keeping only one best candidate why not look at b best options. So, we define the term called beam width b, and we look at b best options at every stage.

So, the search tree would look some look at something like this. So, here we have assumed that the beam search is 2. And as at each level as we go down we have taken the best 2 nodes

and generated their children, then amongst all the children that are generated we again take the best two nodes and so on. So, you can try and think of it as a beam being shown through the exponentially growing search space, and all our beam which is kind of like a pillar in the search space. And the algorithm has a better chance of finding a solution.

As I said all this becomes makes more sense because computer memory is becoming cheaper and cheaper, and you can afford to keep more candidates in the free essentially. Observe that the complexity of the algorithm is still constant times except that if you are got a beam width b, and the branching factor is let say k, then you would be generating b into k candidates at every stage, but it is still a constant number.

(Refer Slide Time: 12:13)



So, here is an example of beam SAT beam search for the SAT problem that we were looking at. We need to define a heuristic function. And one advantage of working with the CNF form

is that we can define a heuristic function which says simply count the number of clauses that are satisfied.

So, as you can see in this example there are 3 4, 5, 6 clauses. So, the heuristic value can go from 0 to 6 essentially. And remember that this also we are viewing it as an optimization problem, because in the solution all 6 clauses should be satisfied and only then the formula will become true.

There are variations of this heuristic function. For example, you could multiply do a weighted sum of the clauses, because some clauses have one proposition inside some have two and so on.

So, you could weight it by the size of each clause as well essentially. But here this search algorithm shows that if you started with the candidate solution where every each of the five variables a, b, c, d, e is one then that has a heuristic value of 3 as you can see here, and then you generate the 5 children and compute the heuristic values and the node shown in blue are the best 2 children.

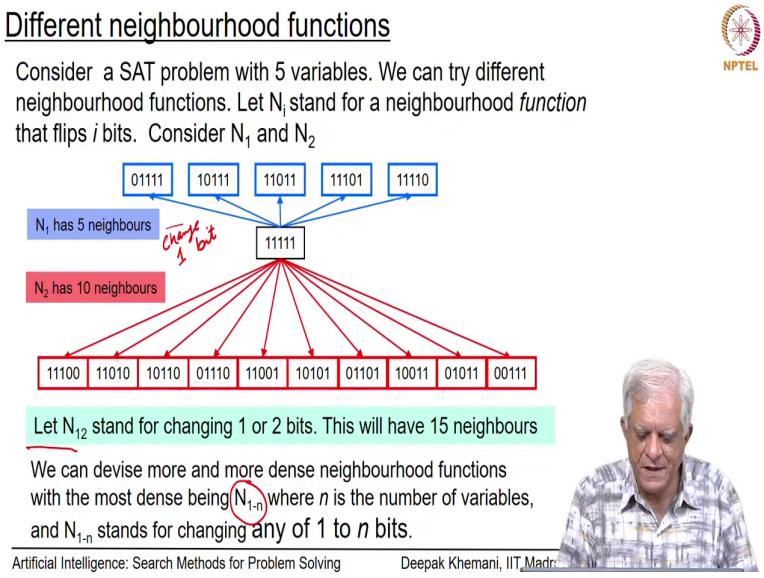
They have heuristic value of 4. Then you generate all the children of all the all those two nodes and you get 10 children, and then you move to the next 2 which are heuristic value of 5. So, you can see that like Hill Climbing as you go down deeper into the search tree, the heuristic value is improving essentially.

You started with a value of 3, at the next level we had a value 4, and at the third level we had a value 5. But at this stage when you take the best 2 candidates which are value 5, again shown in blue, and generate all their children.

You can see that all the children have either a value of 4 or 5 essentially which means that none of them is actually better than that. And like Hill climbing the beam search which is basically you can think of it as you know parallel Hill Climbing with multiple nodes being in contention. We will come to a halt, because it has reached a local maxima again.

Now, there is the node here with the heuristic value of 3. And if you just try it yourself, you will see that moving from this node in a manner of 3 in a matter of 3, steps you would actually go to the goal node. But beam search is not able to find this, because it is it thinks that those other nodes are better the nodes which have heuristic value of 5.

(Refer Slide Time: 15:17)



But they are in fact local maxima where the algorithm gets stuck essentially. Why did we choose the neighbourhood function or the perturbation function which said that change 1 bit? Now, this neighbourhood function that we saw has been shown here in blue that you are going up from the state which is change 1 bit, and we saw that it has 5 neighbours. We could have changed, we could have set change 2 bits and that is the neighbourhood function that you see in red.

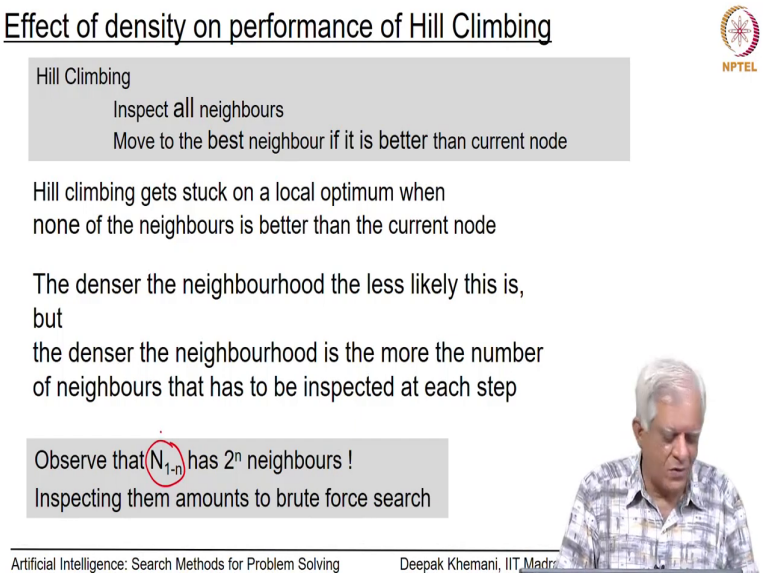
And you can see that if you do that then you have 5 C 2 neighbours, in this case we have 10 neighbours. And therefore, it is a different function. And you can also if you look at the two sets of neighbours, the neighbours generated by N 1 function which change, change one bits or by the N 2 function which is change 2 bits, then they are actually a disjoint set essentially they generate different neighbours.

So, there is also the possibility of saying that you can change 1 or 2 bits, and that we have here mentioned as a neighborhood function N 1, 2 which says that you can change either 1 bit or 2 bit. So, the neighbours would be the union of the neighbours of the neighbourhood function N 1 and N 2. So, in that manner, we can keep increasing you can say 2 bits or 3 bits or 1 to 3 bits and so on and so forth.

So, we can divine device more and more dense neighborhood functions. So, what do we mean by dense neighbourhood function? That these are neighborhood functions which have more number of neighbours essentially. And the most dense being what we can call as N 1-n which means you can change any number of bits you can change 1 bit or 2 bits or 3 bits or 4 bits up to N number of bits.

You should probably think about is that a good idea or is that not a good idea, we will see that in a moment. So, N 1-n stands for changing any of 1 to n bits essentially. Now, the first thing you will observe is that generates a graph where every node is reachable from every other node in exactly one step essentially. So, it is like a very densely connected graph essentially.

(Refer Slide Time: 17:49)



What is the effect of density on the performance of Hill Climbing essentially? Now, if you recall what the algorithm says, Hill Climbing says is that inspect all the neighbours, and move to the best neighbour if it is better than the current node essentially.

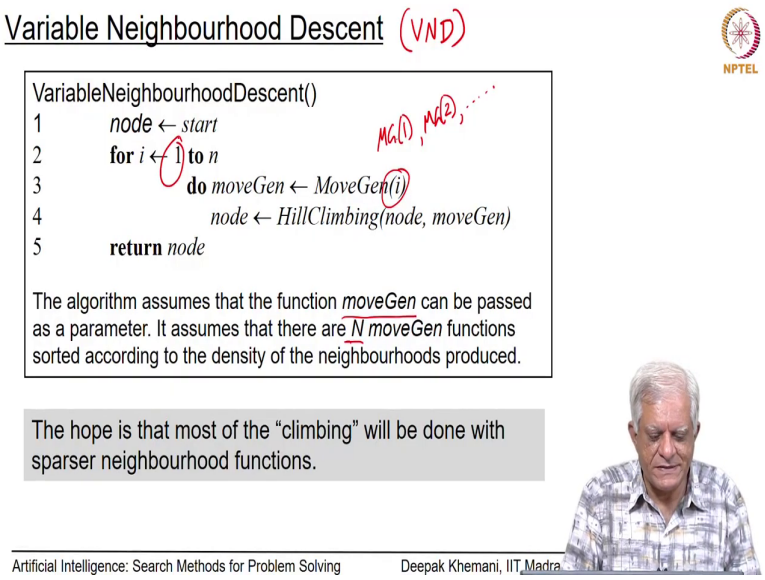
Now, Hill Climbing gets stuck on a local optimum when none of the neighbours is better than the current node that we have seen essentially. So, it might seem that it is a good idea to have denser neighbourhood functions, because the denser the neighbourhood the less likely that it will get stuck on a local optima essentially.

But there is a catch that if you have a dense denser neighbourhood function then you need more computation to generate all the neighbour. So, the denser the neighbourhood is the more the number of neighbours that it has to be inspected at each step essentially. So, that is the

trade off essentially how much how many neighbours you will generate and how many would you inspect.

Now, this function that we talked about in the just now which we called as N 1-n, this has two raise to N neighbours which means every possible candidate is a neighbour. And if you are going to inspect 2 raise to n neighbours, then it is like doing brute force search. So, obviously, we cannot afford to do that. And we said that if n is 100, then 2 raise to 100 neighbours is not something that we would want our machine to spend time in just try to compute how long that will take to do.

(Refer Slide Time: 19:33)



So, instead what we do is we adopt a algorithm called variable neighbourhood descent often abbreviated as VND. And this assumes that the MoveGen function or the neighbourhood function can be passed as a parameter to the search Hill Climbing algorithm. Many

programming languages do allow functions we passed and they treat functions as first class objects, and you can pick one of those languages – especially many functional programming languages they allow you to do that.

So, we will assume that there are some N moveGen functions and they are sorted in the order of increasing density. What this algorithm says is that you look at the first moveGen functions i. So, we assume that you know we have things like let say we use MG for all this. So, MG 1 is as most powers function then MG 2, MG stands for move gen is the next and so on essentially.

What this algorithm says is that start with i is equal to 1, take the most parse function, and simply do Hill Climbing. So, what happens with that, it is a sparse function. So, you can generate the neighbours very quickly and you rapidly climb the hill. At some point you get stuck on a local maxima and that time you change the moveGen function. So, just imagine that if you are doing this SAT problem, we had N 1. So, first you use N 1, then when you get stuck you move to N 2. Then if you get stuck then you move to N 3, or you know something like that.

And so what you do is it that first you use the most sparse function, then the then denser function, then the denser function and so on, and keep doing that till the computation power allows you to do that. Obviously, we cannot do a brute force SAT, because it is a very large problem essentially. The hope is that most of the climbing or descending as the case maybe, we will be done by the sparse neighbourhood functions. And eventually very little work will be done by the denser functions.

(Refer Slide Time: 22:10)