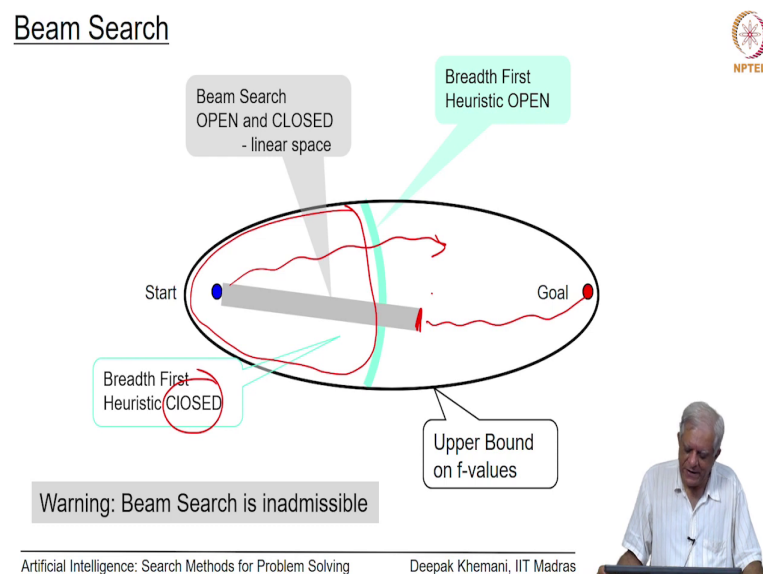


Artificial Intelligence: Search Methods for Problem Solving
Prof. Deepak Khemani
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Chapter – 05
A First Course in Artificial Intelligence
Lecture – 49
A*: Pruning CLOSED and OPEN Beam Stack Search

(Refer Slide Time: 00:14)



Beam search as we have said, our main attraction towards beam search is, because of the fact that it requires linear space together for OPEN and CLOSED. So, OPEN is only at the front ends of this and the rest of this shaded region is a CLOSED essentially and we have seen that it grows linearly w into d where w is the beam width. But of course, it comes with a warning; that means, search is not admissible. It will not give you the optimal path, it will definitely give you a path, but not necessarily an optimal path.

So, if you want an optimal path then you better go to something like A star algorithm and the characteristic of A star like algorithm is that they never preclude a possible solution. So, this is something that we started over saying that arranges search space which does not preclude any solution.

So, in this case, because the solution may have lied around this path, beam search will not find it, because we have said that we will terminate as soon as we find the path to the goal and the path could be somewhere along this slide which may not be the optimal path. So, how can we now adapt or adopt some methods to adapt the beam search into a admissible algorithm, which means that it will find the optimal path for us.

(Refer Slide Time: 01:43)

Divide & Conquer BFHS

Divide and Conquer BFHS keeps three or four layers

The diagram illustrates the Divide & Conquer Beam Search (BFHS) process. It shows a search space bounded by an oval, with a 'Start' point (blue dot) on the left and a 'Goal' point (red dot) on the right. A grey shaded path represents the search space. Three layers are shown: a purple dashed line labeled 'RELAY', a yellow solid line labeled 'BOUNDARY', and a cyan solid line labeled 'OPEN'. A callout box labeled 'OPEN and BOUNDARY' points to the cyan and yellow lines respectively. The NPTEL logo is visible in the top right corner.

Divide and Conquer Beam Search keeps three layers of constant width

Artificial Intelligence: Search Methods for Problem Solving Deepak Khemani, IIT Madras

Before we do that we can combine the methods that we studied for pruning closed. So, remember that all these algorithms were this idea of keeping a relay layer, in a boundary

layer, and an open layer came from pruning closed. The idea of breadth first you receive search came from pruning open. If we combine both the aspects then we do not have to keep the entire closed list for breadth first heuristic search.

We keep only a boundary layer and a relay layer, like we did for the earlier algorithms that we saw, but of course, at the expense of having to reconstruct the path again and again. We could do the same thing with divide and conquer beam search. It will only keep three layers of constant width; one layer for open, one layer for the boundary, and one layer for the delay node.

So, this divide and conquer versions of breadth first heuristic search and beam search essentially try to prune both the open and closed list essentially, but let us get back to first trying to see whether somehow we can make beam search a little bit more explorative in nature so that it will not ignore any paths to the, it will not ignore any path which means that it will find the optimal paths essentially ok.

(Refer Slide Time: 03:31)

Beam Stack Search: Zhou and Hansen, 2005



Beam Search is inadmissible, may terminate with a non optimal path

ANYTIME ALGORITHM.

Beam Stack Search is like Beam Search + Backtracking

Backtracking is **explicit** guided by a Beam Stack

- stack contains pairs of values (f_{\min}, f_{\max}) at each level
- the pair of values identify the current nodes in the beam
- used to guide the regeneration of nodes *while* backtracking

Beam Stack Search orders nodes on f-values.

It slides the f_{\min}, f_{\max} window on backtracking,
but does not go beyond the upper bound U



So, we have already said this repeatedly then beam search is inadmissible. It may terminate with a non optimal path to the goal node. The algorithm given to us by Zhou and Hansen called Beam Stack Search. It is a little bit like beam search, but with backtracking essentially. What does that mean?.

It means that you go forward in the impetuous manner of beam search that keep going towards a goal, but keep the option of backtracking available to us. When will you backtrack?

There are two situations; one is if the search goes beyond the upper bound then you do not want to continue searching and you would backtrack and look for another path, the other situation would be when you have found a path to the goal node, when you have found a path

to the goal node you may want to continue searching to see if there is a better path to the goal node.

In fact, to make things more efficient, every time you find a path to the goal node and its better than the value U of upper bound that you have been using, you can update the value U to that better cost in that sense the algorithm will also converge faster. So, just to repeat, there are two ways when you two points where you might want to backtrack; one is when you have short beyond the upper bound and the other is even when you have found a path to the goal node you can look for another path.

This also gives this algorithm feature which is called anytime algorithm and the notion of an anytime algorithm and it occurs in many places is that you may recall for a solution at any time that you want. So, supposing there is a controlling program which is doing this path finding for some reason, maybe it is a program that is helping you navigate in a city while you are driving.

So, you can afford to keep making for long times before the system tells you what is the path that you want. In an any time algorithm once a path has been found and remember that we have already started with finding a path with beam search. You can say give me a path and whatever the best path it has been found it will be given to you essentially.

This process is also used in game playing algorithms where if you are playing a chess tournament for example, and you have working in the time constraints and your algorithm is searching for a move and you might suddenly say give me the best move that you have found so far and then go ahead with that essentially. So, that is another place where any time algorithms are used and those algorithms were similar to DFID kind of search, that you keep searching and at any time I may say give me a solution.

Now, backtracking in beam stack search is explicit unlike, when we for example, implemented depth first search backtracking was happening implicitly, because the nodes that we stored in open were kept in such an order that they were visited in a depth first fashion and

we could say that the algorithm is backtracking going up to towards the route and then coming down again and that kind of stuff, but it does not implemented explicitly.

In beam stack search the backtracking is implemented explicitly which means you will actually go back to the parent and regenerate nodes essentially and this regeneration process will be guided by something which is called as a beam stack. So, beam stack is a kind of a stack which is associated with the beam and for every level in the beam it stores some information. What is the information it stores? It contains two values; one is called f_{min} and one is called f_{max} .

So, it is an interval as you can see here close on the left hand side and open on the right hand side which means that f_{min} is inside the beam, it is a value of a node inside the beam and f_{max} is a value of a node outside the beam, not included in this beam. The pairs of values f_{min} and f_{max} , they identify the current nodes in the beam. We will see in a moment.

They are used to guide the regeneration of nodes while you are backtracking essentially. So, if you want to backtrack you need to know which path that you have visited and which path you are going to visit again or which new parts you want to visit to make the search systematic.

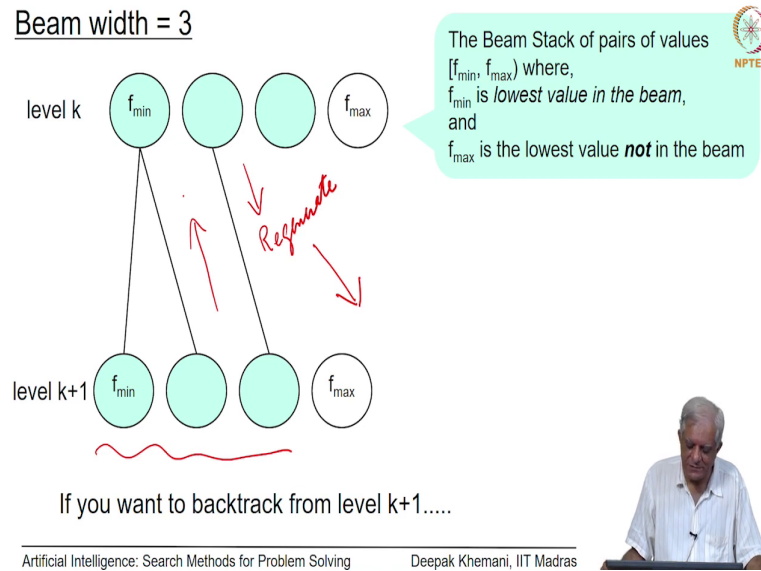
That is done with the help of this beam stack which contains these two values. Since, at any time in the beam we will keep the best w nodes, if w is a beam width. You can think of saying that beam stack search organizes the space into an ordered space in which the cheaper solutions are encountered first.

Of course, in practice that does not happen, but its helps us to visualize how that algorithm is performing. So, in the visualization that we will see in a moment, we will see that at every level the cheapest nodes are on the left and the most expensive nodes are on the right and they are ordered in that fashion.

So, this window of f_{min} f_{max} would be a window at every level which in the process of backtracking. This algorithm will slide from left to right essentially. So, it will allow you to explore new nodes by the process that we will see, but it will stay within the bounds of upper

bound U and as we have said a short while ago, you can keep updating or tightening the upper bound as and when you find better paths to the goal essentially.

(Refer Slide Time: 09:38)



So, what is the process like? So, imagine that your beam width is 3 and at level k you have those 3 nodes that are drawn on the top, shaded inside. Remember, these are ordered from left to right, f_{\min} is the smallest f value, then the next one, then the next one and then the next one.

So, there are 4 nodes here, 3 of them are in the beam and the smallest is f_{\min} and the fourth one is not in the beam and in fact, fourth one is a marker which will tell us as to where to start looking for if you want to backtrack and come back to that layer again.

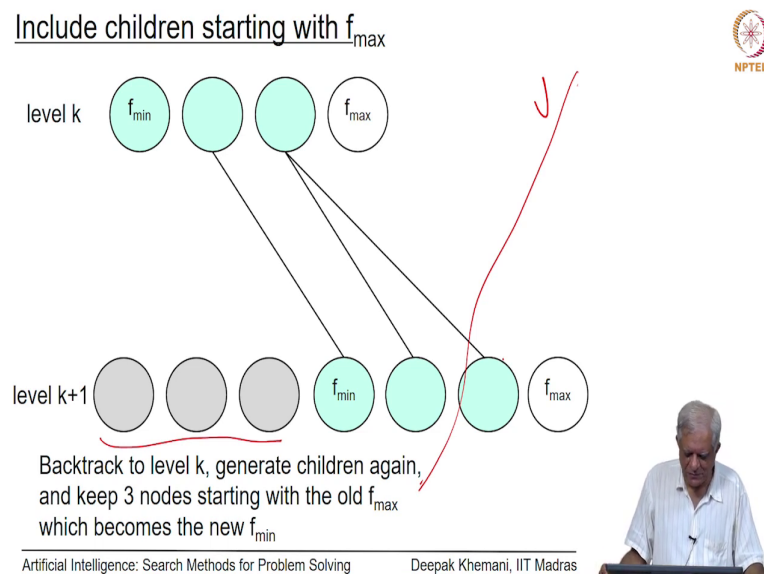
So, we will illustrate this with this level $k + 1$. $k + 1$ also has those 4 nodes; f_{\min} as the cheapest node in the beam, f_{\max} is the cheapest node which is not in the beam. We are not counting, we are not talking about nodes which have been visited.

So, all these nodes are yet to be visited. So, f_{\max} is the cheapest node which is yet to be visited, f_{\min} is the cheapest node which is currently in the beam essentially and if you want to backtrack from here what do you do?

You can go back to the parent of these nodes at $k + 1$ level and you will have 3 nodes in this k th level. You can regenerate the children of those 3 nodes. So, you backtrack up and then regenerate and from these regenerated nodes you know which ones to select next and which ones to ignore.

So, these 3 nodes also will be, these 3 nodes that we have here they will also be regenerated, but we have already visited them and we are in fact back tracking from them. So, we will not use them and in fact, the regenerated nodes will be starting with f_{\max} essentially.

(Refer Slide Time: 11:30)



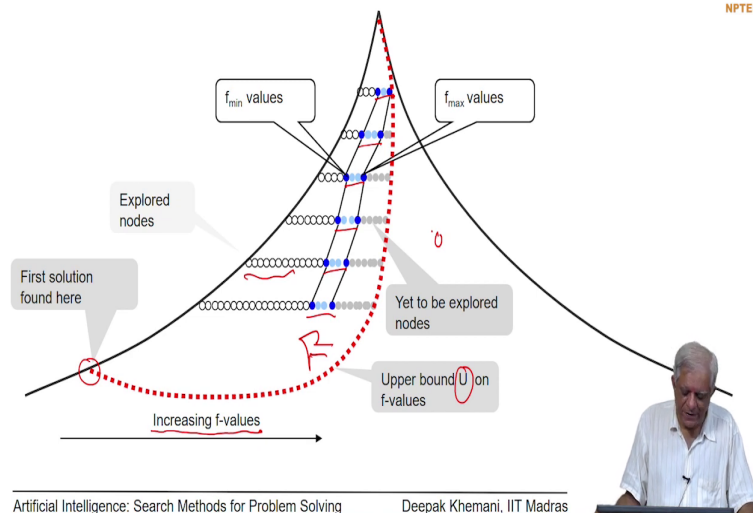
So, this is how it would look. Those 3 nodes that we backtrack from have gone into the set of nodes that we have visited and 3 more children of their parents have been added to the beam and the old f_{\max} value has become the new f_{\min} value and we have got a new f_{\max} value at some point.

Now; obviously, if we have somewhere along which that upper bound boundary ah, then we would not go beyond that boundary essentially. So, this is the basic idea of backtracking.

Use this f_{\min} and f_{\max} values to go back to the parent and regenerate the children and select the appropriate ones as dictated to you by f_{\min} and the f_{\max} values. In fact, as dictated to by the f_{\max} value.

(Refer Slide Time: 12:33)

Beam Stack Search: Zhou and Hansen, 2005



So, this is how the search for beam stack search looks like. As I said we are visualizing a search tree in which we are arranging nodes at every level as increasing f values. So, for every level in this search tree we are organizing the nodes in an increasing value. So, in some sense our search algorithm will sweep from left to the right. Left is the cheapest nodes; right of the more expensive node.

When the first node is found, it would be found in some at some point by the beam. You have an upper bound and then you can draw this boundary which is shown here in red which marks the upper bound U essentially on f values and you can make sure that your algorithm does not go searching beyond this upper bound.

So, in this you can see that I have kind of drawn some nodes in the beam at different level. So, these are the nodes which are actually stored in the memory, the rest of the nodes are have

either been visited and deleted. For example, the nodes which are on the left hand side and the nodes which are on the gray on the right hand side have yet to be generated and explored essentially.

So, beam search beam stack search is like beam search in the sense that it will require space which is linear with depth, but it also has this feature that it will be systematic and complete the term that we had used, that it will explore the space all the space in which a solution could possibly lie essentially and that space is defined to us by the upper bound that we have created.


So, if we keep searching this systematically and we do not exclude any path which is lying within this region which is on the left of the upper bound then we can be sure that we will find the optimal solution, because it has not precluded any solution essentially. There maybe solutions for example, here, but we know that their f values will be higher than that of the node path that we have already found. So, we do not need to explore them any further.

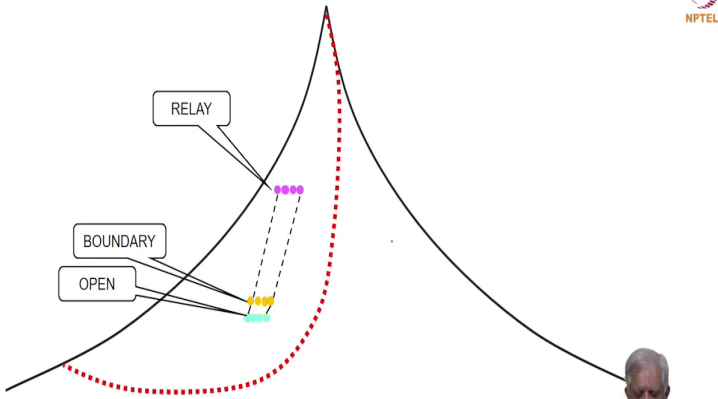
So, here it is an algorithm which is linear space algorithm which does not do repeated searches of the kind we did when you did divide and conquer. It requires linear space and it guarantees to an optimal path. On top of it has this nature of being an any time algorithm which means that if you are happy getting a path which is not necessarily optimum you can still call for it at any point and the algorithm will return the best path that it has found for you essentially.

So, the last thing we want to see is that can we somehow cut down on the space even further from here. We have already brought down this whole exponential space to a linear space and we can clearly solve fairly large problems here, but can we do even better and that algorithm was called divide and conquer beam stack search.


As you can see it is basically just like we converted beam search into divide and conquer beam search, we can convert beam stack search into divide and conquer beam stack search, which means that we do not keep all the nodes which are there in the beam.

(Refer Slide Time: 16:12)

Divide & Conquer Beam Stack Search: Zhou & Hansen, 2005 



DCBSS stores three layers of width w .

Artificial Intelligence: Search Methods for Problem Solving Deepak Khemani, IIT Madras 

We keep deleting them, but we keep a relay layer at roughly the halfway mark or if you want to the strategy of smart memory graph search, you could do that in a more smarter fashion, but irrespective of that we keep a constant number of nodes in the beam. If the beam is this 3 for example, we will keep 4 nodes, 3 which are in the beam and fourth one which is the f max value likewise, a boundary would have 4 nodes and the open would have 4 node essentially.

So, you can see that this algorithm requires constant space, at least as far as storing the state nodes is concerned essentially. So, it will store only three layers, but of course, there is a problem, we wanted it to be complete as well. Remember that beam stack would go back to its parent.

So, if you look at this diagram, you go back to your parent here and then regenerate the node. So, if you have deleted this parent then how can you do this process of backtracking? That is

a next question to address and that is the last thing we will do for our exploration of A star algorithm.

(Refer Slide Time: 17:49)

Divide & Conquer Beam Stack Search backtracking



DCBSS has deleted the parents of nodes in the beam

How does it backtrack then?

Answer:

- Regenerate nodes from the start state
- Guided at each level by the values in the beam stack

Space complexity of DCBSS is $O(1)$

- three layers of constant width
- relay, boundary and open
- *if you ignore* the memory needed by the beam stack



So, how does divide and conquer beam stack search do backtracking? Divide and conquer beam stack search as deleted parents of nodes in the beam and how does it do how does it back track name? That is the question that I will ask you to ponder over for a few minutes. Think about how this can be done and come back in a few minutes if you have some suggestion. I will give you the answer when you come back. So, pause the video here and come back after 5 minutes ok.

So, there you are. You have thought about hopefully the problem of how can this divide and conquer beam stack search actually do backtracking. The answer is not so complicated.

The answer simply says that instead of going back to the parent, you regenerate the nodes from the start state again essentially and when you are regenerating the nodes you have the beam stack to guide you. It will tell you that of all the children that you generated every layer which are the nodes in the beam and which are the nodes where you have to start exploring next.

So, you can regenerate the nodes up to the parent level, the appropriate level, then generate the children of the parents and go to the newer set of nodes which is what you wanted to do in backtracking.

So, this of course, means a little bit of extra work not too much extra work and the result of this is that the space complexity of divide and conquer beam stack search is of order 1 or it is constant, because as we said it only keeps three layers of w nodes, where w is the beam width and which is a basically a constant amount.

The nice thing about all these algorithms that we have been talking about beam stack search in particular and divide and conquer beam stack search if need be is that nowadays with the memory being cheaper and abundant we can actually afford to keep the beam width to be very large indeed and while theoretically this idea that you may have to backtrack very often. In practice you could get away by doing only a few backtracks that is a impact of machines becoming faster and bigger essentially.

The complexity of divide and conquer beam stack we have said it is order 1, because it keeps three layers of constant width and the three layers are relay boundary and open, but of course, we are ignoring the fact that we also keeping the beam stack. The beam stack is growing linearly.

At every layer it keeps two values, but if you kind of ignore that those two values and if you only consider the nodes in the state space that you are storing then it is a constant number and it appears to me that you cannot possibly do better than that. So, with that we will end our long journey of A star algorithm.

(Refer Slide Time: 21:07)



The End

A* and its variations



Artificial Intelligence: Search Methods for Problem Solving

Deepak Khemani, IIT Madras

We started by devising A star algorithm or looking A star algorithm in the quest for finding optimal paths and we showed that if you use the g values and the h values.

The g values are the known cost up to a given node and the h value are the estimated cost from that node to the goal node and if the h value satisfies certain properties which is that it underestimates the heuristic function and we also specify that the branching must be finite and h cost must be greater than some epsilon under those conditions we showed that A star is admissible, that it will always find the path to the goal if there exists one and it will always find an optimal path to the goal.

Then we argued that if the heuristic function is more informed better which means higher even though less than the h star value. It will search less and less of space and then we looked at the host of algorithms to reduce space further we started with wA star, we looked at IDA

star, RBFs, then we looked at algorithm to prune the closed list which is defined divide and conquer frontier search and smart memory graph search.

Then we looked at algorithms to prune the open list which was breadth first heuristic search and beam search and then we try to combine all these pruning close and open together by combining the divide and conquer features with the beam search.

So, we got beam stack search on the way which was a complete algorithm admissible algorithm and we said that we can even go further than beam stack which is the divide and conquer beam stack search. So, we will stop with our study of A star algorithm here and move on to the next topic see you then.