

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 – 45
A*: Pruning CLOSED and OPEN Divide & Conquer Frontier Search

So welcome back, we are in the last stages of looking at algorithm A star which we have spent a lot of time and the reason for that is that it is an algorithm which is widely used algorithm and specially as computers become faster and bigger in terms of memory.

You are finding more and more applications where we can afford to run A star essentially. At the same time just as machines are becoming faster the problems that we want to tackle even becoming faster at a greater pace and we still need to worry about how to manage memory and that is what we have been trying to do so far.

So, we have seen various approaches to reducing memory in A star and before that in heuristic search we looked at local search we looked at stochastic methods, then we came to A star because we wanted guarantees of optimal paths and we saw variations of A star.

For example, the blue star which made it faster using less memory but it could become admissible. Then we saw IDA star and recursive best first search which required linear space, but the number of cycles that they had to do was very large and in a phenomena which we call as thrashing; the algorithm would revisit the same path again and again and again essentially hm.

So, today and in the next session we will look at some variations of A star in which we will try to see if we can prune the close list and prune the open list as well essentially.

(Refer Slide Time: 02:06)

The Monotone Condition (recap)



The *monotone property* or the *consistency property* for a heuristic function says that for a node n that is a *successor* to a node m on a path to the goal being constructed by the algorithm A^* using the heuristic function $h(x)$,

$$h(m) - h(n) \leq k(m,n)$$

The heuristic function
underestimates the cost of each edge

For A^* the interesting consequence is that every time it picks a node for expansion, it has found an optimal path to that node.

We can design algorithms that never look back!



We also studied this monotone condition, the monotone condition on the monotone property says essentially that the heuristic function estimates the underestimates the cost of every edge.

And this is reflected or expressed by the inequality that if n is a successor of m on some paths and in particular to the paths to the goal, then h of m minus h of n is less than the actual cost which means the difference in heuristic values is less than the actual cost uh . So, it underestimates the actual cost essentially.

We saw that for A^* the interesting consequence is that if this property is satisfied by the heuristic function, then at the moment that the algorithm picks any node n it has already found the optimal path to that node. Which means that the case 3 in the A^* algorithm

where we updated the cost of node when closed is no longer necessary and we can now focus on algorithms which will never look back and just look ahead.

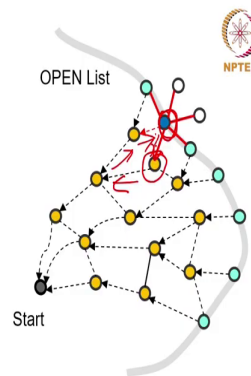
And head towards a goal essentially Dijkstra star algorithm was a particular case of this in the Dijkstra star algorithm h of n for any node is 0. So, you can see that 0 minus 0 is always going to be less than the cost of the h and it is not a surprise that lifestyle algorithm or branch and bound they find an optimal path as well.

(Refer Slide Time: 03:34)

The role of CLOSED in Search

We needed to maintain the CLOSED list* of nodes for two reasons

- One, to avoid getting into infinite loops, which happen because a node on CLOSED may be a neighbour of the node being expanded.
 - We will look at other ways to avoid the search from "leaking back"
- Two, to reconstruct the path once the goal node is picked up by the algorithm.
 - We will look at a new mechanism to do so



* In practice CLOSED should be implemented using a hash table for efficiency



Now, if you want to prune the closed list we should figure out as to what was it doing for us in the search, why did we have this notion of closed list at all essentially. So, there were essentially 2 reasons that we maintain the closed list for incidentally.

We are calling it a closed list but in practice if you look at the application you want to essentially check whether the node has been visited before or not. And secondly you want to retrieve a particular node to reconstruct the path both these operations are done based by constructing something like a hash table rather than a list.

But conceptually speaking we will refer to it as a list or we will call it only closed, but keep in mind that if you are implementing A star or its variations you would want to use a hash map kind of a representation for that. So, one reason why we needed to keep close was to avoid getting into infinite loops and infinite loops happen because a node which is on closed maybe a neighbour of a node being expanded.

So, if you look at this diagram here and you see this node in blue which is on the open list and let us say this is a node that is about to be expanded. Then there are 2 nodes in closed which are neighbours of this node, 2 nodes on open shown in cyan which are also neighbours of this node and 2 nodes which will be generated when we expand this node these are new nodes essentially.

The matter of concern for us was a node like this one ah, because it was enclosed then it is possible that if A star went from generated this as a child of this open node then it could it could possibly go along this path then maybe along this path, then maybe along this path maybe along this path this may be an extreme example that I am giving.

But in practice it could happen that it could get into a closed loop in our case, of course since we are trying to keep track of g of n which is a cost of the path found it is not likely to happen. But that was the motivation for keeping things in closed list.

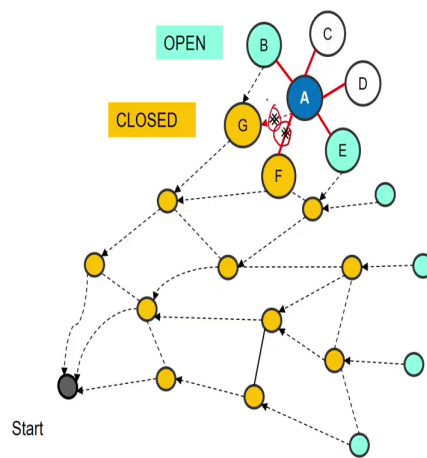
So, avoiding this infinite loop may require other solutions and that is one of the things that we will look at today as to how can we make sure that the search does not go back to the nodes that it is already visited essentially. And the researchers who work on this area we call it that the search will not leak back into the visited nodes, because that is what can lead to infinite loops.

The second reason why we wanted closed list first to reconstruct the path from the goal node to the start node, because we always maintain parent pointers and the parents were always in closed. And essentially by tracing those parent pointers you could go back to along the paths that is been discovered for reaching the goal node.

So, we will need to do another mechanism for this as well. So, what is our goal now our goal is to look at an algorithm which prunes the closed list, but which also maintains a functionality that you do not get into infinite loops and you can find the path or report back the path or return the paths that we have found essentially for which we had also needed close.

(Refer Slide Time: 07:25)

Korf and Zhang, 2000: Frontier Search



Let us say node A is about to be expanded.



Its *active* neighbours are C,D (new nodes)
B,E (on OPEN)

Nodes G and F (on CLOSED) are barred (see next slide)



So, this algorithm was given by Korf and Zhang in 2000 half of it is name is Frontier search. So, let us focus first on the Frontier part of the search. So, this is the same graphs with some

labels attached here the node that is about to be expanded is node A and its neighbours are shown by edges which are shown coloured in red.

As you can see there are 2 neighbours B and E which are already on open there are 2 neighbours G and F which are already on closed and there are 2 neighbours C and D which are new nodes that are going to be generated.

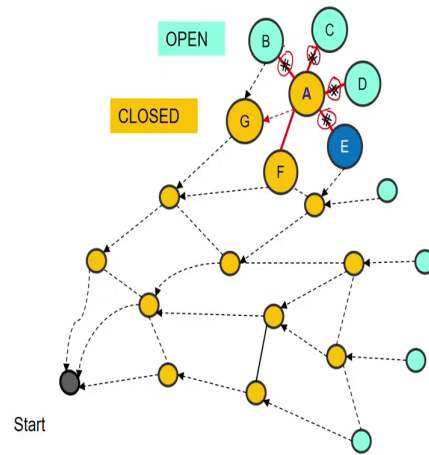
So, what does Frontier search to the idea Frontier search is to maintain only a frontier and we will see how it does that. So, let us say that A is about to be expanded by this algorithm which is a variation of A star. So, basically it picks nodes on the same criteria as A star which is the lowest F values, its active neighbours as we can call them are C and D which are new nodes and B and E which are already on open essentially.

So, these are nodes to which it may have found a new path or the first path in the case of C and D here. The nodes G and F which are also neighbours of this node A which are on closed are barred and we will depict this by saying that we will bar A from generating those 2 nodes.

So, effectively what Korf and Zhang algorithm frontier search does is that when it expands node A it does not regenerate G and F, because they are barred and it may regenerate the other active nodes which are B C D and E essentially.

(Refer Slide Time: 09:21)

Korf and Zhang, 2000: Frontier Search

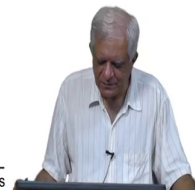


A is added to CLOSED and its active neighbours C, D, B, and E are barred from generating A when they are expanded



Remember A* has already found the optimal path to A

Let E be the next node visited



So, this is what happens we have put A into close A is added to the CLOSED and it is active neighbours which will B C D and E they are now on open, some of them are B and A were already on open they continue to remain on open. Maybe A has found a better path to them, so that will be worked out as per the algorithm A star that we have studied.

What the different thing that this algorithm does is that the active neighbours C B C D and E are barred from generating node A again and as I said that bar is depicted by this hash like sign, this on the edge which means that when you expand those nodes you are not allowed to generate A as a neighbour of those nodes. So, A will not be generated as a neighbour of B or of C or of D or of E.

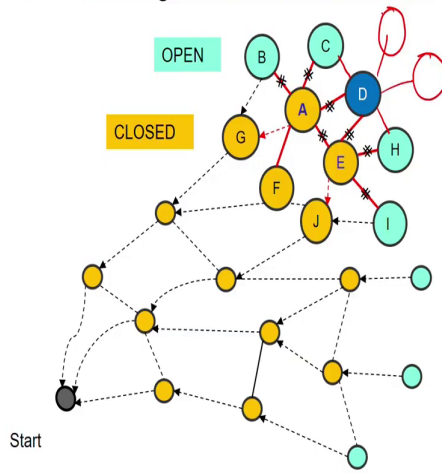
Now, that is fine because you remember that we are working with the monotone criteria and under the monotone criteria or the monotone condition when A was picked A star I had

already found optimal path to that. So, there is no case for trying to revise that path because you cannot find any better path essentially.

And let us say node E is the next one to be picked up, remember that the node is picked up from one of the open nodes and I have depicted in blue that node in open which is likely to be picked up next essentially or which is going to be picked up next.

(Refer Slide Time: 10:59)

Korf and Zhang, 2000: Frontier Search



When E is added to CLOSED its active neighbours D, H, and I are barred from generating E when they are expanded

So D will not generate either of A and E



So, what does E do? E also goes into CLOSED as before and as before it generates some active neighbours. Now, notice that the active neighbours of E D H I A is not an active neighbour because E was generated as a child of A and we have barred A from becoming a child of E. So, even any 2 nodes only one can be the child of another node. So, the active nodes are D H and I and when they are expanded in their turn they will not generate the node E.

Now, notice that both A and E are on the barred list for the node D which presumably is about to be expanded next. So, when you expand node D it will not generate A and E it may generate H. If there is an edge to node C it may generate H, it may generate some new nodes as before and then the same process will repeat essentially.

(Refer Slide Time: 12:13)

Korf and Zhang, 2000: Frontier Search

Frontier Search:
Keep only the OPEN nodes,
along with list of barred (tabu)
nodes with each node in OPEN.

Search only moves forward,
without "leaking back"

Start

Deleted CLOSED

OPEN

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

So, having done this having done this altered this OPEN list in a way that when nodes on OPEN are picked and expanded, they do not generate the parents which exist for them in the tree. The parents may be the parent that you came from, but it could also be a parent that had also generated node. So for example, in this case you can see that for both node C and D is a parent and also for nodes D and H E is a parent.

But the OPEN list has been modified such that if you were to generate the neighbours of any node or expand any node, the parents that had already been seen they will not be active or

they will not be generated as part of the new set. So, what Frontier does it maintains only the open list and it deletes the closed list it does not keep a copy of that closed list, though those parents in the closed which have been barred have to be maintained with every open node.

So, you can imagine that if the open list is of size K and if in general the branching factor is let us say B then something of the order of B times K extra nodes would be stored along with open. So far with C for example A would be stored with D in this example A and E would be stored and so on essentially.

So, this is the first objective is to make sure that the search does not leak back and this has been achieved by ensuring that nodes which have been already seen and which may have generated loads on open whether or not that was optimal path or not they will never be visited again.

So, this stops the search from leaking back essentially. So, this is a aspect of the Frontier search that you only maintain the Frontier nodes or only maintain the open nodes, keep only the open nodes along with the list of barred we can call it tabu in the style that we had been talking about earlier. Nodes with which with each node in the open and search only moves forward without leaking back at all.

(Refer Slide Time: 14:56)

When Frontier Search picks the goal node... *...it has no means of reconstructing the optimal path it has found*

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

Now, the problem is this that when the algorithm terminates this is how the set of nodes look, you see a host of nodes on the open list you see the start node which we always have and you see the goal node that you have picked up, but the algorithm has no means of reconstructing the path the optimal path that it is bound to the goal node.

So, it would have found optimal path because the criteria is still the same that it picks node based on their F value and we have shown that at the point when it picks the goal it could have found optimal path to the goal.

So we have no reservations on that count, but we do not have a closed list that we used to have from which you could trace back the path. So, remember that what you would have done is that you would have gone to its parent and then from there you would have gone to it is

parent and then you would have gone to it is parent and so on and then you would have reconstructed the path to the start node.

Now, unfortunately because we have thrown away closed we do not have this facility available to us. So, what did Korf and Zhang suggest they said the following that.

(Refer Slide Time: 16:10)

Relay nodes: at roughly the half way mark

When a node on OPEN has $g(n) \approx h(n)$ mark it as relay and keep pointers from its descendants on OPEN

Start Goal

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

You want to delete the closed list go ahead and do so but maintain one extra layer of nodes which is shown here in pink and these nodes are called relay nodes which will be at most the size of OPEN. But typically less and maintain a pointer from every node on OPEN to a relay node essentially.

So, when the relay layer is created, so for example when this node is created and when it is children are generated they may be somewhere here we maintain a pointer to those children.

But when these children are deleted and their children are deleted again we maintain a pointer back to the relay here.

So, in that sense all the descendants of a relay node will point back to that particular relay node and different nodes on open may have different relay nodes essentially. Where do we place a relay node Korf and Zhang said that place them at roughly the halfway mark essentially.


And the idea was to divide the problem into 2 smaller problems and like we did. For example, you must have studied binary search in binary search you would like to divide the problem into 2 halves either you search in the left half or on the right half and so on. And that makes an optimal case for the divide and conquers approach to problem solving that you convert a problem into smaller problems and solve them essentially.

How do we know that a node is at the halfway mark well we do not know for sure, but you can use some criteria which says that at the point when a node has A G value roughly equal to H value, remember that H values actually an underestimate.

So, at the point where G value is roughly equal to H value it will be a little bit closer towards the goal; it will not be exactly at the halfway mark. But it will do for us because you know our goal is to basically break up the problem into 2 parts and make it a relay node.

So, at the point when a node on open becomes such that its G value is roughly equal to its H value convert that node into relay node do not delete it and use it as an ancestor for all its descendants, key pointers from all the descendants which are on the OPEN list. So, this is how the algorithm looks like at the point where it terminates, you can see that there is one pointer from the goal node now the pointer goes to the relay node which I have circled which is an ancestor of the goal node.

(Refer Slide Time: 19:06)

When Frontier Search picks the goal node...  ...it has a pointer to its relay node Relay

- Solve two recursive problems
 - 1. From Start to Relay
 - 2. From Relay to Goal
- Divide and Conquer Frontier Search (DCFS)



If $T(d)$ is the time complexity needed to find the goal at d steps then time complexity of DCFS is

$$T(\text{DCFS}) = T(d) + 2 \times T(d/2) + 4 \times T(d/4) \dots$$

If T is exponential, then $T(\text{DCFS}) = T(d) \times d$



And next what Korf and Zhang said was that at the point when the goal node has been picked up, the goal node has a pointer to the relay node which we call relay and then you recursively solve 2 problems. One problem is to find a path from start node to the relay node and the second problem is to find a path from the relay node to the goal node essentially.

And you can see that essentially what we have done is that we have managed so throw away the closed list, but we have said that to reconstruct the path we will search again. But in the next cycle of search or in the next round of search instead of solving the problem from start to goal, because that would be meaningless we will solve 2 smaller problems one from start to relay and one from relay to the goal node.

Now clearly this will require less time, but we will have to do it twice essentially keep that in mind. So, supposing you made this relay, you solve one of the problems or let us say you

solve the problem from start to the relay node what will happen at the end of that particular recursive call another relay node would be found on the optimal path essentially.

Then you solve the second half which is depending on what order you solve them ah, let us say you continue solving recursively the path from start to relay. So, we have found one another relay on the path if you call them R1.

Then you solve the left hand side and you will find another relay R 2 and this process this recursive call will continue till the point. When you cannot break up the point in the node any further and that could happen. If for example, the next relay node it was directly connected to the start node and connected to this node.

So, in that manner we will gradually fill in the nodes in the path. So, we will fill up this then we will fill up this and then we will recursively solve this then we solve this then we fill up this then we fill up this and so on essentially. As we happen we gradually reconstruct the path, but at the expense of making many recursive calls to solve a sub problem again and again.

So, what are we doing here like we did in IDA star and RBA FS we are creating of space versus time we are saving on space we are saving on closed and in particular we are interested in saving closed because of the kind of problems we saw like sequence alignment. But at the expense of doing extra work you cannot have a free lunch and this is what the algorithm does.

So, what is extra work that you have to do? So, let T_d be the time that the search algorithm uses to find a path of these steps. Remember that this T_d is dependent upon the heuristic function; if the heuristic function is perfect then this T_d would be a linear function of depth. But in practice we have said that T_d will be exponential in nature.

So, that is one of the reasons why we are doing this whole process repeatedly and because it is exponential which means that the space requirement would be exponential as well. If it was

a perfect heuristic function in this case requirement would be linear and something like hill climbing would have happily taken you to the goal.

But heuristic functions are never perfect and not only are they not perfect in general they require exponential time and space. And in that scenario we are looking at this divide and conquer frontier search algorithm as Korf and Zhang mean that. And you can see that the total time complexity of divide and conquer frontier search here is this is a time spent for the first search.

When you found a paths to the goal or when you found that you could reach the goal rather and then you solve 2 recursive sub problems of size d by 2. So, that you have to add to that for each of those 2 recursive problems you solve 4 recursive problems of size d by 4. So, you have to add to that and till the point where recursive stops and you need the base case you have to generate you have to solve this equation to determine how much is the time.

It turns out and you should work this out, that if time is exponential in nature then the amount of time needed by divide and conquer Frontier search is d times the time that it used to find the first instance of a path to the goal when it found that a part to the goal exists and it found the cost of the path. But if you take that as a complexity T^d then you have to do d times more work essentially and that is the extra cost you have to pay.

So, this is a divide and conquer frontier search algorithm given by Korf and Zhang, what it says is a throw away the close list but at the halfway point maintain a layer of relay nodes. When you find the goal node solve 2 recursive problems 1 from the start node to the relay node that the goal node points. So, remember that is important this relay node in this diagram is a relay for the goal node essentially.

So, we know that the path to the goal goes through that relay node. So, you recursively solve the path from start to the relay node and from relate to goal node and then that itself is solve recursively and so on and we have this algorithm which does repeated searches to solve the problem.

So, what we have achieved here is that we managed to cut down on the closed space requirement. But at the expenses of this extra cost which may be at most d time the original cost where d is the depth of the goal node.