Artificial Intelligence: Constraint Satisfaction Problems
Module 6: Search Methods: Look-Back
Lecture 3: Gaschnig's Backjumping, Graph-Based Backjumping
Professor: Deepak Khemani
Department of Computer Science and Engineering, IIT Madras

Keywords: Gaschnig's backjumping, Graph-based backjumping.

In the last class we started looking at this Gaschnig's backjumping algorithm. And we saw that the key feature of this algorithm was a select value function, which when it is trying to find a value for this i^{th} variable x_i, keeps track of the partial solution which was inconsistent with this i^{th} variable. And then essentially it needs to know what is the longest partial solution that is inconsistent, starting to construct it from x_1, x_2 up to x_i so on. And then it can identify the culprit variable, by keeping track of this longest variable. And we saw in a small example with this n queen’s problem how when in this example, when the sixth queen we were trying to find the value we could figure out that the jump back should happen to Queen 4 and not to Queen 5, because the latest value for this queen which is $latest_6$ its value was 4 and we could jump back to 4. So, let us formalize this into this algorithm. So, the algorithm is a slight variation on the backtracking algorithm that we have done. The basic idea is still the same, that you keep moving forward to the next variable trying to get a value for it, it is except that when you do not find a value you jump back to something. And what you jump back to is determined by the select value function essentially. So, everything else is similar, you start with $i = 1$ you make a copy of the i^{th} domain and you say that the latest value is 0. And then like before while i is between 1 and n, you do the following: you instantiate x_i to a value that you get from a function which we will now call as **select-value-GBJ**, which is a acronym we will use for this algorithm Gaschnig's backjumping and as before if $x_i = null$, then we will need to backtrack. But, here we will backtrack to... instead of saying $i = i - 1$ you will say $i = latest_i$ and that tells us for example queen 6 the $latest_6$ was 4 and we could jump back to Queen 4. Otherwise, we progress forward. Rest is similar, we make a copy, and we set the latest value to zero. And then, if $i = 0$ return inconsistent else return... I will just write it as else return the solution. which is basically the set of values x_1 is equal to this value x_2 is equal to this value and so on and so forth. Let us write this function here. So, again like in all our select value functions while the domain of $D_i' \neq \{\}$ we select some value and remove it from there. And now we want to test whether this value is consistent with the partial solution that we have for the $i - 1$ variables before that. Now instead of just testing the partial solution a_{i-1}, we will incrementally look at larger and larger portions of $i - 1$. So, with basically we say that we will start with by setting $k = 1$, and while we are still looking at partial solutions remember that when you reach i then we have already come to the current variable and consistent. Somewhere we should say consistent, so we will start by looking at... so remember that we have this we have this partial solution that we have a_{i-1} and so we set $k = 1$. So we initially point $k = 1$ and we look at only this part of the partial solution then we will look at a slightly longer partial solutions and slightly longer solution and so on essentially. If this is a part that we have written earlier, remember that when we enter this module or this subprogram, $latest_i$ was instantiated to 0 essentially. But now of course once we move to the first variable, we will set it to 1, and then as we move this k, so this k will be incremented gradually and then we will keep incrementing it like a ratchet mechanism. If $k > latest_i$ then $latest_i$ will get the value k. So, as k moves forward we will keep incrementing $latest_i$. Remember we want to find the maximum inconsistent partial solution essentially so that we know where to jump back to essentially. And if not consistent, remember that when we write this in upper case it's a function which checks the consistency of this a_k with $x_i = a$. a is the value that we are investigating. Then you simply say so this is a variable, and you set it to false so you can see what is happening here, we started with $k = 1$ then we go to $k = 2$. Look at incrementally larger and larger partial solutions that we have from for the previous variables. And stop at the point where it becomes inconsistent. At that point the $latest_i$ variable would have got incremented to wherever we reached essentially. And of course if if it is not consistent then we go on and look for the next value essentially. So, if not consistent we just say consistent = false and then because while has a consistent check, it will exit from the loop else it will look at a larger part of the partial solution. So, essentially we look at longer and longer prefixes, if you want to call it of the partial solution. So this is one part of the while loop, and how will you exit from this while loop, you will either exit when $k = i$, which means it is no longer smaller than i, which means if that you have found a consistent value or if consistent has been set to be false, which will happen if the partial solution is not consistent with this value a for this D_i and at that point you have already incremented $latest_i$ to the maximum that you could reach essentially. Now, in the next cycle it will go back to the next cycle which means it will take the next value from the domain of this variable and try the whole thing all over again. It will again set $k = 1$ and then keep incrementing till it becomes inconsistent and at that point may be $latest_i$ would have become a new value, new larger value. So over all these sequence of while loops, the $latest_i$ will always maintain the maximum value that we have essentially. So, if at the end of this while loop, if this is true, if consistent, which means that we have found a value, for we have found that this value a that we were looking at is consistent so just return that value a, else return null. will assume that this $latest_i$ variable is a global variable which will be seen by the calling algorithm. So if you return null then that x_i would have got this latest value, and this algorithm will simply jump back to whatever is the index stored in the latest value which is the index of the culprit variable essentially. So like I said earlier, there are two cases that you must consider. One case is when you do not find a value for this variable x_i. At that point you jump back to the latest once. On the other hand if you do find a value for this variable x_i in one of the while loops inside which means for one of the values for this from this domain D_i, then you will return that value a and $latest_i$ would have been set to x_{i-1}. So, that is something that you must think about, because at the moment when you find the value of a... by the time you find the value by the time you exit this while loop, and consistent has not changed that means k would have gone up to $i - 1$ essentially. So, that is only place where you can exit... this this internal while loop essentially. So, maybe I should make a note here, here $k = i - 1$. If you are returning a value of a, or rather, $latest_i$, and we have said that this is a global variable essentially. So if you find a value for the i^{th} variable, then you will not have anything to jump back to. You can only jump back to the previous variable because that is what $latest_i$ stores essentially. But if you do not find a value ,then this $latest_i$ would be somewhere higher up as we saw in the example that we were trying to see... in this here. So, this is the maximum that this arrows have reached forward essentially. And you know that you can jump back from this to this and not any of these intermediate variables essentially. So, this is the variable that you should jump back to. But once we jump back to an internal dead end for example in this six Queens problem you jump back from Queen 6 to Queen 4, and the Queen 4 turned out to be an internal dead end, but its latest value is 3 and therefore you can only backtrack one step back essentially. So that is the problem with in some sense a problem with Gaschnig's back jumping algorithm is that it can jump back from leaf dead ends, but it cannot jump back from internal dead ends, because the latest value is only the previous variable essentially.

The next algorithm we want to look at and we just start looking at it informally today and then we will move to the formal algorithm essentially, is called Graph back jumping, or Graph-based back jumping. The basic idea behind graph based backjumping, is what we started with when we started looking at the notion of back jumping with the example that we saw in this map coloring example. In this example we saw that when you cannot find a value for x_6 in this case, then it makes sense to jump back to one of its ancestors which is related to x_6 which means either x_1 or x_2. And if you think about this a little bit you will see that x_1 would not be safe but x_2 would be safe in the general case so you should jump back to the latest ancestor or what we will call as a parent in this case, and a parent is a connected node. Remember we already have defined the notion of parents when we talked about minimum width ordering, and so on and that is the notion we will carry forward here. So let me look at a sample graph, and discuss some of the issues that we would be considering. So, let us say that this is a given ordering for a graph: x_1, x_2, x_3, x_4, x_5, x_6. So, one feature about graph-based backjumping is that it only looks at the topology of the graph. It is not concerned with values inside the domains essentially. So, it simply says jump back to the parent that is the simplest notion for leaf dead ends we jump back to the parent.

But that parent may or may not be conflicting with that essentially. So, it is kind of extra conservative in that sense essentially. So if this is a graph that we have x_1 is connected to x_2 and x_2 is connected to x_6 and x_1 is connected to x_3. So I am just taking this example from Dechter's book. Okay so let us say this is the constraint graph for whatever the constraint satisfaction problem that we are solving and this is the order. So, if you remember the notion of a parent is the latest connected variable in the constraint graph. So, the parent of x_2 is x_1 the parent of x_3 is also x_1 the parent of x_4 is also x_1, the parent of x_6 is x_5 and so on and so forth. So, we have this notion of parents essentially. And let us say we are doing this backtracking algorithm so you are trying a value for x_1 then trying a value for x_2 and x_3 and x_4 and x_5 and so on. So, the first thing we want to do is for a leaf dead end, jump back to a parent. So, let us look at some examples here. If x_3, then jump to, remember I am talking of leaf dead ends here, x_1. Because the reason why you cannot find a value for x_3, so remember that when when if x_3 is a leaf dead end, I must emphasize this, we are talking about leaf dead ends here. If x_3 is a leaf dead end, it means that we found a value for x_1 and x_2 but we are not able to find a value for x_3. Why are we not able to find a value for x_3, because it is violating some constraint and what is the constraint it can violate, the only constraint it can violate is which involves x_1. So, look for a new value for x_1 essentially. So, you can jump back to x_1 essentially. Then if x_5 then jump to x_4. Now of course x_5 is kind of unique, it has only one ancestor. In some sense it's a it's a misleading kind of a case but we should first go through this and then realize what's happening essentially. And then in fact that ancestor is the parent which is x_4 essentially. Now because of this, and because if it has jumped to x_4, if x_4 is a dead end, what does this mean that we found a value for x_1 then x_2 then x_3 and x_4 and then x_5 was a leaf dead end, and we jump back to x_4 and we find that there is no other value left for x_4. So, it is a dead end and it is an internal dead end. Then in this case, you can jump back to x_1. That is fine. There is nothing which is wrong here essentially. So this is two cases that we have seen. This case number 3 is more complicated, let me write it here. So if okay so I used this thing twice. If x_7 is leaf dead end, then jump to parent, to x_5. Now if this is a dead end then jump to x_4 but if this is a dead end, is a question mark essentially. So, what is happening here, we find that x_7 is a dead end, and then we jump back to x_5 then we find this x_5 is a dead end and you jump back to x_4 okay. So I am trying to distinguish between these two cases where x_5 was a dead end. In the first case which is actually the second case that we discussed here, x_5 was a leaf dead end. When x_5 was a leaf dead end we could jump back to x_4 and then we made a claim that we could jump back from x_4 to x_1, if x_4 was an internal dead end. That was one case. The second case or the third case that we are looking at is when x_7 is a leaf dead end. When x_7 is the leaf dead end and we jump back to x_5 because x_5 is a parent of x_7 that is fine. And we find that x_5 is a leaf dead end. Now, we jump back to x_4 because x_4 is a parent of x_5, it is fine but what if x_4 is also a, is an internal dead end essentially. So, in the earlier case when x_5 was a leaf dead end, we could jump back to x_4 and then to x_1, but when x_7 is a leaf dead end, we can jump back to x_5 which becomes an internal dead end now, and then to x_4 which is fine. But where do we jump from x_4. Is it okay to jump to x_1 is a question like before essentially. And the answer to that is no, you must jump to x_3, and why is that? It is a lower level, you are backtracking, but why x_3, why not x_2? Another parent of... well not apparent it is an ancestor of x_7 essentially, because it is an ancestor of x_7. And why is that important? You might miss out a solution. Because the inconsistency that you saw at x_7 could have been caused by x_3 essentially. You know that the value that we were looking at could have been such that, supposing this was a map coloring algorithm, and the, and we could not find a value for x_7 it could be because if you have a different value for x_3 we might have found a value for x_4. And which means that when we want to jump back, so just let us go back over this process. We found x_7 was a leaf dead end. You could jump back to x_5 that was fine, we could jump back to x_4 that was also fine. But we from x_4 we have to jump to someone some node which is not a parent of x_4. But it is a ancestor of x_7 essentially. So, we have to somehow formalize this notion essentially, as to what should we jump back to. And that is an idea that we will pursue in the next class but informally speaking the idea is this. That when you are retreating to an internal dead end, then you must look at not only its ancestors but also the ancestors of the node from where you are retreating essentially. So, if you have come to x_4 as a internal dead end. You have come to x_4 from x_5 so you must look at the ancestors of x_5. But in this example x_5 has only x_4 as an ancestor, so it does not matter. But also ancestors of x_3 and then ancestor of x_3 includes x sorry ancestor of x_7 includes x_3. So x_3 is later than x_1 essentially. So, in some sense we will see and we will define this notion in the next class that x_3 is the induced parent of x_4. When retreating from, so we do not want to jump back to a parent only but to a notion of an induced parent that we will try to define in the next class. So you were saying something. If x_4 and x_5 were not connected then in the second case where x_5 was the leaf dead end, we would jump to x_3. So that is not a, that would not cause a problem essentially. See if the only change you are making is that your instead of connecting x_4 and x_5, you're connecting x_3 and x_4, x_3 and x_5. Then by the same definition of the induced parent we will discover that x_3 is an induced parent of x_4. So, basically the notion of an induced parent is based on the notion of induced ancestors which means your own ancestors plus the ancestors of the nodes that you are retreating from, and this whole set of ancestors from that the latest node is going to be the induced parents. So, you collect all the ancestors into a set, called the induced ancestor set and from that the latest is a parent. So, which kind of extends the notion of a parent. The parent is the latest ancestor that you are connected to. An induced parent is a latest node that is an ancestor of either the one that you are currently, or the one that you have retreated from essentially. And this notion of induced ancestor, induced parent is only for internal dead ends, and it is motivated to define what is the safe jump. So like he said if you jump back from x_4 to x_1 you might actually miss out on a solution because changing x_3 would have been the answer essentially. So we will, we have run out of time so but we will formalize this notion. We will start from here in the next class and we will formalize the notion of induced parents, and once we do that the algorithm is straightforward essentially.