Theory of Computation
Professor Subrahmanyam Kalyanasundaram
Department of Computer Science and Engineering
Indian Institute of Technology, Hyderabad

W12L66_Savitch's theorem

Hello and welcome to lecture 59 of the course Theory of Computation. In the previous two lectures, we saw an introduction to space complexity. We saw what space complexity is. We saw the space complexity model. We saw the relationships between space complexity and the time bound complexity classes. Then in the previous lecture, we saw NL completeness.

And in this lecture, we are going to see an important theorem in space complexity, which is Savitch's theorem. Savitch's theorem relates the space requirement of the non-deterministic space bounded Turing machine and the deterministic space bounded Turing machine. To be more precise, it states that if a language can be computed by a non-deterministic space bounded Turing machine that uses f(n) space or O(f(n)) space, the same language can be decided by a deterministic space bounded Turing machine by f(n)^2 space. So, a non-deterministic space bounded Turing machine using f(n) space can be converted to a deterministic space bounded Turing machine in f(n)^2 space.

So basically, we can let go of the non-determinism if we just use up the square of the space. We can leave the non-determinism by just squaring the space usage. It's a kind of a surprising result, especially when we look at it from the perspective of what we saw in time complexity. For instance, SAT was in non-deterministic linear time. So, if you have a non-deterministic Turing machine that runs in linear time, you could decide, let us say, 3-SAT.

If we had a similar result, n time f(n) is contained in d time f(n)^2, such a result does not exist. If it was there in time complexity, this would imply that SAT is decidable in order n^2 time, which we know would be a huge thing, but that is not the case, right. In fact, we still do not know if there is any polynomial time algorithm for SAT. So, that way this is surprising. And in fact, there are a couple of other results also in space bounded complexity classes.

The results are very surprising, especially when we contrast them with the similar results in time complexity or similar classes in time complexity. So, this is one of such results. And as we will see, the proof is going to be extremely simple and straightforward. It builds on some of the things that we have already seen in the previous lectures. So, we saw that to decide whether a space bounded non-deterministic Turing machine accepts a certain input string, we just have to look at the configuration graph and see whether there is a path from the starting configuration to the accepting configuration.

Of course, we need to assume that there is a unique accepting configuration, but we have already discussed how to get to that. This is a relatively easy thing to accomplish. So now the focus is going to be, so you have a language A, right? Suppose you have a language A, which is an arbitrary language in N space f(n), right? We will show that this A belongs to D space f(n)^2, right? So suppose A is decided by a non-deterministic Turing machine called N, which decides it in O(f(n)) space, right? A decides, sorry, N decides A in O(f(n)) space, right? Now, in the previous lectures, we have seen that such a machine has 2^(O(f(n))) configurations. And since it is better to be specific than asymptotics, so instead of saying things like 2^(O(f(n))), it is better to have a specific number at hand. So, let us say that the constant hidden by this O is d.

So, let us say that there are 2^(d f(n)) configurations for the Turing machine N. So, maybe I will just highlight this. So, the Turing machine N that decides A, the non-deterministic Turing machine N, it has 2^(d f(n)) configurations. So this is an upper bound. So we will just choose d to be the appropriate constant so that this is satisfied.

We know such a constant exists because of the calculations that we did in the previous lectures. We just didn't really, the exact constant will depend on the exact Turing machine that we are looking at. Now, for this Turing machine and amongst these configurations, there is an obvious graph, right. So, each configuration will have some number of successor configurations, right, because it is a non-deterministic Turing machine, right. So, we will have some, if you remember, the picture was something like this, the C start may have some successor configurations, right.

So, and it could be, like there could be multiple successor configurations because it is a non-deterministic Turing machine, right. There could be various directions it goes, and we want to check whether there is a path from C start to C exit. So, we want to see whether there is some computation path that goes from, let us say, C start to C accept. If it is there, then the input string is accepted. So, C start is also specific to the input string for the Turing machine N on a certain input string.

So, the starting configuration will change based on the input string. In fact, this entire configuration graph is drawn for that particular input tape content. So now we have to decide this on this graph. We have traversal algorithms all that, but right now the goal is to decide this in space bound O(f(n)^2), right, where the idea that we will use is very, very simple. So all we are saying is that if there is a path from, let us say, V1 to V2 that takes T steps. If there is a path from V1 to V2, let us say, that takes T steps.

So, please focus on this part. If there is a path from V1 to V2 that takes at most T steps, then there should be a center vertex, or there could be more than one, but there should be one vertex such that V1 to W, there is a path. So, the one central vertex that I am calling it W, there should be a path from V1 to W of length at most T divided by 2 and there should

be a path from W to V2 of length at most T divided by 2. So, this is the basic idea. There should be a, if there is a path from V1 to V2, using at most T vertices, then there should be a vertex W such that there should be a path from V1 to W using at most T divided by 2 and W to V2 using at most T divided by 2.

So, maybe I will just do one thing. I will just put this thing in a box. So, this is right. So, this is the main idea here. So, now we want to check whether there is a path from C start to C accept.

So, what we, so let us first explain a generic thing. So, we accept if we just check whether T1 equals T2, sorry, V1 equals V2, sorry, this is an error. We accept if V1 equals V2 or if there is an edge from V1 to V2, right. Basically, that is the only way that we could have a path of length at most one. Otherwise, if T is not 1, so there has to be such a vertex W that we just explained.

So, what is the W? We do not know that W. So, because we do not know, we just try out all possible such Ws. We just go through the list of vertices one by one and see whether for any W, is it the case that there is a path from V1 to W of length at most T divided by 2 and W to V2 of length at most T divided by 2? And if both of them accept, then we accept. So, once again, the algorithm is the following. If T equals 1, we accept with the simple conditions.

If T is not 1, we check whether there is a path from V1 to W of length at most T divided by 2 and W to V2 of length at most T divided by 2 and accept if both accept. And this W, we do not know which W it is. So, we just try out all possible W's. And if any W works out, that is fine. We have a path from V1 to V2 of length at most T.

So, now, this is the algorithm. Now, we will have to just what should be V1, what should be V2, what should be T, we have to just say. The space usage is at most d f(n)^2. These are the two things that we have to show. So, V1 is C start, V2 is C exit, and T, we set T to be the total number of vertices.

So, because this is a graph and it has 2^(o(f(n))) vertices. That's my assumption. We said that it has 2^(o(f(n))) vertices. So, if there is a path from C* to C_accept, there has to be a path of at most 2^(o(f(n))) vertices because that is the total number of vertices. If there is a path, there is a path without reusing any vertex.

So, the maximum, like if you have a graph with 10 vertices or 100 vertices, the farthest apart two vertices can be is of length 100. You can have a path from A to B of length 100, but not more than 100 because the total number of vertices itself is 100. So, we just need to check whether there is a path from C* to C_accept that uses at most 2^(o(f(n))) vertices. So, now let us see the space usage. So, initially, we have V1 equal to C*, V2 equal to

C_accept, and T equal to 2^(o(f(n))). Sorry, maybe I should also mention initial call has, I'm sorry, C* and V2 equal to C_exit, right? And then what we do is we use recursion, right? So, what happens is we use recursion in this, right?

So, because it is, this is what is happening here, the same algorithm, the same function calls itself. So, it is kind of like a recursion. So, if initially the parameter was T, I am referring to this parameter, the inside calls, it makes two calls of where the parameter is T/2, right? So, initially we have one call with T, then two calls of T/2, and then each one of them makes two further calls of T/4, and each one of them makes further calls of T/8, and so on. So, we need to look at the total space usage.

So, let's see what is space usage. So, here in this case, we have to store V1, V2, and then a counter for W, and then we also need to store. But then the thing is that when we make these recursive calls, at some point we will get the answer back, and then we have to continue the computation. So, when you have multiple nested recursive calls, we have to keep remembering all the data that was relevant of all the recursive calls that we have made in memory. So, if we are, let us say, the third deep or the third nested recursive call, we should remember where we were in the first time, the second time, and so on, right? So, for each recursive call, we need to keep this memory, right? So, it is not just having to remember V1, V2, W, etc. V1, V2, W, etc. We actually need to remember V1, V2, W, T for all the nested recursive calls, right? So, in fact, I think I need to mention W also, right? So maybe W also. I think I'll just edit it, right? And I'll say, so now let's see how much we need to remember.

So, we need to remember V1, V2, T, and W at each recursive call. And let's see, so V1, V2, and W are vertices. We know there are 2^(o(f(n))) vertices. So, the label associated with each vertex has o(f(n)) bits or o(f(n)) space is required. So, log of that many bits, that much space is required.

There are 2^(o(f(n))) vertices, so if there are, let's say, 64 vertices or 63 vertices, we need six bits to remember which, or to, we need a six-bit label to separately name each vertex, right? Similarly, there are 2^(o(f(n))) vertices. So, we need o(f(n)) bits to separately name each vertex. So, each one of V1, V2, W requires o(f(n)) bits, right? We have also said that T is 2^(o(f(n))). So, even to keep track of T, we need o(f(n)) bits.

So, all of this requires o(f(n)) bits. So, that is the space required at each recursive call. So, o(f(n)) times 4 of these. So, it is 4 times o(f(n)). And then the thing is I mentioned that we have to make this, we have to keep this thing in mind or in memory whenever we make a recursive call. So, the next question is how deep is this recursive tree? How deep is this recursion tree? So, T will become T/2, T/4, and so on.

But how many levels can it go? So, we know that the base case is when T equals 1. So, how many levels does it require for it to reach 1? So, every time it halves. So, we need log

T levels for it to reach 1. So, this tree has log T levels. So, the number of levels is o(f(n)), which is just log T.

So, the space usage is, now the space usage is 4 times o(f(n)) per recursive call and the number of calls is going to be at most o(f(n)) because that is the number of levels. So, the total space usage is, we replace 3 by 4, so maybe I will make it 4 here, but that would not really make a difference. So, total space is 4 times o(f(n))^2. And which is again like 4 and d are a constant, right? So d is some constant based on the configuration tree, right? Which is based on the non-deterministic Turing machine and 4 is of course a constant, so it is o(f(n))^2 and that's it. So just by a simple algorithm, we have shown that a non-deterministic Turing machine that uses o(f(n)) space can be simulated by a deterministic Turing machine that uses o(f(n))^2 space.

And the algorithm is very simple. We want to check whether there's a path from starting configuration to the accepting configuration in a graph with T vertices, where T is 2^(o(f(n))). In other words, if there is a graph with 2^(o(f(n))) vertices, we want to check whether there is a path from one vertex to another vertex. So, we just check whether that uses at most T vertices, the path that uses at most T vertices. So, this is a recursive algorithm.

If T equals 1, it is straightforward. Otherwise, we just make recursive calls by trying out all possible intermediate vertices. Notice that here it is not a non-deterministic guessing. It is just that we are going to try out each possible W. This is okay. It is okay to just spend time on this because we are not paying by time.

We are just paying by space. So, and the space usage is that each level we need to store V1, V2, W, T. So, that requires o(f(n)) space for each of these. So, it is o(f(n)) and the number of levels is the height of this tree, that is also o(f(n)), like log T. So it's o(f(n)) multiplied by o(f(n)) multiplied by some constant. So, it's order o(f(n))^2, which is like d is a constant.

So it's O(o(f(n))^2). Sorry, O(f(n))^2. So that is the total space requirement. So this shows how you can simulate a non-deterministic space-bounded Turing machine using a deterministic space-bounded Turing machine just by squaring the space usage, right? So this is a very surprising result. And one more point is that we don't, so here we are actually using, we need to know what is the space usage of N. What is the space usage of the Turing machine that decides A? That is one thing that we are using here. We are using the space usage of N because 2^(o(f(n))) and 2^(o(f(n))) is being used in the initial call of path.

So, we do not even need to do that. If we do not know f(n), we could just try out f(n) equal to 1, 2, 3, and so on till we get some f(n) for which we know we get an answer. Either we get that it is reachable, like the accept configuration is reachable from the starting

configuration. Or at some point, we'll get that no, nothing more is reachable. At some point, trying out f(n) equal to some value and the next value, it is not going to improve things.

So when that happens, we know it is not reachable. So we can just try out one by one. That will also, even if we do not know f(n). So, that is another small point. Even if we do not know f(n), we can just try out f(n) one by one till we hit an f(n) where we get to know for sure.

So, that is Savitch's theorem. As I said, it is an extremely simple theorem, simple proof, and the result is very powerful. n space f(n) is contained in d space f(n)^2 and it is surprising. Let me just summarize by stating some consequences of it. So, we saw NL which is non-deterministic logarithmic space.

So, it is n space of log n. This is contained in d space of log^2. So, it is log n squared. So, if we use logarithmic space, then the deterministic machine needs only logarithmic space squared. So, for example, we know the algorithm for the graph reachability problem in logarithmic space, as I mentioned before. But in deterministic logarithmic space squared, you need to solve this. So, these are all consequences of this theorem.

Why? Because, so consider, suppose, let us say, just for the sake of understanding, let us assume that there is such a space, NSPACE, which is a union of all NSPACE(n^k), right, all polynomial space-bounded languages of n^k ranging from k equal to 1 to infinity. But then by Savitch's theorem, right, by Savitch's theorem, NSPACE(n^k) is contained in DSPACE(n^(2k)). So, each polynomial space, so whatever was contained in NSPACE(n) is contained in DSPACE(n^2), whatever was contained in NSPACE(n^2) is contained in DSPACE(n^4), NSPACE(n^3) is contained in DSPACE(n^6), and so on. But then once we have this, all of this is contained in DSPACE(n^k) when k ranges from 1 to infinity. So, basically what this is saying is that non-deterministic polynomial space is contained in deterministic polynomial space.

And obviously, the other direction inclusion is trivial. Anything that is deterministic is obviously contained in non-deterministic. So, this shows that non-deterministic polynomial space and deterministic polynomial space are one and the same. So, this is the reason why I did not define it then. So, because nobody studies NPSPACE because it is the same as PSPACE.

So, this is yet another contrast as a consequence of Savitch's theorem. While P versus NP (polynomial time versus non-deterministic polynomial time) or deterministic polynomial time versus non-deterministic polynomial time is probably the biggest open problem in theoretical computer science, or perhaps all of computer science, when we move to space complexity, deterministic polynomial space is the same as non-deterministic polynomial

space. There is no open question, and the proof is just what we just saw. It's extremely simple.

So, these are some of the consequences of Savitch's theorem. Savitch's theorem states that non-determinism can be traded off by squaring the space, and the proof is extremely simple by using an algorithm that decides whether there is a path from one point to another point, one vertex to another vertex, using at most this many intermediate vertices. And the idea is just recursion.

And that completes this lecture, lecture number 59.

That completes this lecture. See you in the next lecture, lecture number 60. Thank you.