NaturalLanguageProcessing-Lecture14

Instructor (Christopher Manning):Okay. Hi, everyone. Okay, so today will the last class doing statistical parsing. And so today, there are three topics I want to cover. The first one is for look at more general and at other ways that you can do the parsing tasks, which are effectively different ways of doing this search task of finding good parsers. Then I want to say something about the pentagrammars and how they contrast with doing the kind of constituent grammar parsing that we have mainly concentrated on. And then just at the end, I want to say a teeny little bit about using discriminative methods in parsing.

I am not going to cover that in very much detail, but just to motivate the idea as to why it’s a useful thing to do, say, affection about how people do it until we have had a couple of results using discriminative parsing. Okay, so right at the end of last time I mentioned these goals of what you might want to do with a PCFG. And how, particularly, if you just want to find the best parse for a sentence, that rather than doing a CKY as we have done up until now, that maybe there are cleverer search techniques that you would like to employ which might work better.

And two methods you could use have beam search, and that’s certainly been used quite a bit for parsing just like MT, but I am not going to talk about that now and then this idea of a gender chart base search. Okay, so just going back a little to the fundamentals, so what do we have? So for what we have built for parsers, we have items, which are commonly in the older literature, also called edges. But that will be confusing, so don’t think edges because these are really going to be our nodes. These are items that are things that we can build over a span. So that’s something like a noun phrase between position zero and two.

And then we have backtraces or traversals, which they represent the way in which you can make something over a span. So the backtrace or traversal is this red bit down on the bottom, and it’s recording the way I was able to make this item a sentence from zero to three was combining a noun phrase from zero to two and a verb from position two to verb phrase from position two to three. Okay, so then you can think of a parse tree that you get out at the end as a collection of the items that you build and then the traversals that they are built from. So a parse tree is a representation of that sort.

And in general, when we are exploring possible parse trees, we are building some kind of packed representation in which we can record a whole lot of partial hypotheses. And the parts of those partial hypotheses are things of that sort as well. And so the main thing that we have done so far is use this idea of a parse triangle or chart representation for CKY parsing. And what we do, as an organizational concept there is that we took all things that were items over the same span and put them in one place.

Over here, if we have both a verb phrase and a verb from position two to three and the word ran from position two to three, that all of these things would be stored in the same cell as things over the same span. But if we think in general about this picture of having items and traversals, we can just sort of draw all of our items, and then we can draw traversals that can construct different items. And what we want to do is search in this space where we can gradually start building different ways in which we can build new items by making a traversal that combines two different things.

And so if you do this in the general case, the data structure you get is a B-directed hypograph, which is a kind of a cool data structure. So there is this general concept of hypographs, where hypographs, unlike the regular graphs that you regularly see in computer science, in hypographs you have nodes just like before, but the edges between nodes can have any number of tails and any number of heads. But the whole assemblage of those things counts as a single edge. And so, for example, this here is a single hyper edge where you are making a sentence payrolls – sentence headed by payrolls out of these two pieces.

So particularly for the case of parsing where we have these binarized trees, the situation that we have is that our hyper edges have at most two tails and always have one head. And hypergraphs with that former referred to as B hypographs because they have this binary nature. And B hypographs have special properties because they can be processed more efficiently than general hypographs. And so they are directed – you can also have undirected hypergraphs.

But these are directed hypergraphs, of course, saying that for a pair of things you can introduce a bigger thing, so that if you built a preposition from three to four and a noun from four to five, then you can traverse this hyper edge and build a prepositional phrase from positions three to five. So in the general case, what we want to do is search, which is sort of like search that you can do in other context in AI. Apart from here, we are doing search in hypergraphs rather than in regular graphs. Okay. So if we think about that, we can then think, well, surely there are other ways in which we can do search inside this hypergraph.

And what do we do if we do CKY parsing? Well, in CKY parsing, this is a hypergraph, right, because I have got these hyper edges here that can connect up to things and rewrite them as another thing. Well, what I have done with CKY parsing is I have kind of organized the hypergraph into a certain shape. I have kind of piled all of the items that cover the same span on top of each other. And then I have imposed by my dynamic programming algorithm a certain order of exploration. So what I do is I, first of all, build everything that I possibly can over spans of size one.

I then build everything I possibly can over the spans of size two, everything I can over spans of size three. And I get up to, eventually, finding parsers of the sentence. Now the good thing about this algorithm is it’s really simple providers of dynamic prograre. And while that’s one good thing about it – I hope you found that when you are doing the assignment. The second good thing about it is it’s kind of trivial that this algorithm is correct. You can’t really kind of mess up the correctness of it because you just build everything small.

And then once you have built smaller things, you build larger things on top of that. But the bad thing about it is what we actually want to build is items that span the entire sentence. And by the construction of the algorithm, we are going to attempt to build absolutely everything possible that has a smaller span before we even attempt to see whether there are constituents that span the whole sentence. And that seems kinda bad because, I mean, probably some of the things that we are building down here, were really, really unlikely.

So surely, there must be some way to do a cleverer search exploration so that we don't build all of those candidates. And there are. People have explored that quite a bit in parsing. And the obvious idea there is that what we are going to use – we are going to use, in some way, the probabilities, the scores of items to decide what a promising hypotheses to pursue and which things aren't promising hypotheses to pursue. And that leads into the idea of a gender-based parsing. So for a gender-based parsing, we have the same kind of table recording scores of things over spans.

So the table looks just like the kind of table that we use for CKY parsing. But in addition to that, we are then also going to introduce an agenda as a way to order our search. Okay, so we start with this table like before, and we put in words, and we give the words their scores of having different parts of speech. Okay. Then on top of this, we are going to introduce a second data structure, the agenda. And the agenda is going to be items of the same sort, actually. It's going to be something like having a noun phrase between two and three. But it's also going to have a score attached to it.

And we are going to – and the score, it's, in some way, connected with the probability of things. I'll explain the details of that later. And what the agenda is going to be is some sort of priority queue so that we can decide what to work on next based on its score on the agenda. Okay. And so we are gradually going to build up things in the chart. And if we find better ways of making things in the chart, we'll then lower the score of the corresponding theme in the agenda. And so the idea is that the agenda will tell us what to work on next that's the most promising thing.

Okay, so we have this kind of two-way communication between the agenda and the chart. So what we do is we will build things in the chart, and when we have found a new way of building something, like I have found a new way of building a noun phrase from words two to five, that will then go as a thing to work on in the agenda. And the agenda will be ordered. And if it gets to the top of the agenda, saying that one of the most promising things I found is noun phrases from two to five, it will then come off the top of the agenda and then will explore ways to search further.

Such as, you know if there is a noun phrase from two to five, maybe – and we have a role of noun phrase – noun prepositional phrase. So therefore, we could build a new edge of noun phrase from two to ten, say, because we know there is a prepositional phrase further to the right. So that will then generate a new edge that will go into the chart, and then immediately that will leads to a new thing being put on the agenda, which is, at some point, you should then explore that new noun phrase from two to ten and work out how you could build even bigger things from that.

And so the distinction that we get from these two data structures, which if you paid attention in 161, and you have seen Dykstra's Algorithm, effectively this distinction is the same distinction that you have in Dykstra's Algorithm between things that you have discovered and things that you have finished, that you have done further exploration from them. So things in the table or chart are things that you know exist, that you know that there is a possible constituent there. Things – those things, then, are put in the agenda.

And when you take them off the agenda, you then work out what you can do with them, like how can you explore further from this thing? Okay. And so, in general, something can appear – you can find multiple ways of building some things. So you can put something in the table once, and then you can find a different way of making a noun phrase from two to five. And at that point, you can look at its current score in the agenda. If the new way of making it has a better score, you update its score in the agenda. And [inaudible] was the priority queue. If the new way of making it is worse than the way of making it before, you don't need to make any change.

Okay, so that the general algorithm for a gender-based parsing is – you have to initialize things in some way, so you need to put some items into your chart such as the words and their part of speech tags, and you have to have something on the agenda to start off with. And so you might start off with initializing the agenda with rules that can use parts of speech tags. And then the general algorithm is you take the best thing off the top of the agenda. You find always that you can locally explore around it. So how can you take that item from the agenda and sort of combine it into bigger things on either the left or the right?

And that will lead you to build new, even bigger, things which you stick in the chart. And that will give new task to the agenda. And this includes the unary roles. And then you kind of repeat over this loop of keeping on, taking off the best thing from the agenda and exploring it. And then at some point you stop. And when you stop, isn't when you have found a way of parsing the whole sentence over on this side because you may yet find a better way of parsing the whole sentence.

The right time to stop is when you pop from the agenda a way of parsing the whole sentence because that means that you have found, at that point, what your system thinks is the best way of parsing the whole sentence. Okay. So that's the general framework of a gender-based parsing. If you think about it – if I don't specify anything more, it's just the general family of ways of doing parsing. And in fact, you know, you can code up CKY parsing like this. I mean I don't recommend you do so because it's just taking a fairly simple thing and making it more complex.

But if you wanted to, you could code up CKY parsing as this kind of a gender-based parsing. The way that you do it is say that the score of an item is its size and have the priority queue ordered so that small size is good. And so, then the first things that came off the priority queue would be over size one items, and then all of the size two items, and then all of the size three items, and you simulate doing CKY parsing. Okay, so what we want to do now, though, is do less work than before. And so the idea is, well, maybe what we could do is use some of our probability ideas to work out how to do less work.

And so the very simplest idea is to say, okay, when we build any item, we know what its score is, at least for that way of building it, right. That we have built a noun phrase in some way; we know the probability of that way of building a noun phrase. So let's use that probability as a goodness measure and so explore forward from that. Okay. So we stick on the probability of an edge, and then what we are going to do is use that probability to order our agenda. And so that gives us a Dykstra's algorithm, or uniform cost search in AI terms, where you are just ordering things by the probability of the part that you have already built. Okay.

And so uniform cost parsing now, rather than being like CKY where you order things by their span, now you are ordering things by what is their probability so that you do all of the highest probability things, and then you start working out to building with lower probability things. And so this is, again – also, easily you can show that this is a provably correct algorithm which will necessarily find the best parse of a sentence. And the essence of that proof, which I'll just give extremely briefly, is that because we have rules that are probabilities and under the assumption that the probability of each rule was less than one, at least there are a couple of rules that rewrite each nonterminal.

Then whenever you build something bigger, it has a smaller probability than something it's made out of. And therefore, because we put things in the agenda and take out the highest probability one first, we have to have built this before we built that. So if this is the in the agenda, that must be in the agenda. And, well, by assumption, this has to have higher probability than that; therefore, this has to come off the agenda first and be expanded. And so therefore, we can guarantee that when we pop a parse – when we pop a constituent that spans the entire sentence of the agenda, it has to be the highest probability way of parsing the entire sentence.

And at that point, we can stop and guarantee that we have found the best parse of the sentence. Okay. So here is just a little example of the difference. So here is a tiny little grammar of the kind of that I – well, it's different from the one that I have used before, since this one is using a noun-verb ambiguity. So this is people, fish, tanks. And it has an ambiguity because you can either have people, fish as a noun and tanks as a verb, this is some new start-up called people fish, or you can have fish as a verb and have people, fish, tanks. Yeah, it's another of my tortured attempts to make ambiguous sentences with very few words.

And so that's a grammar that will find both of those parsers. And so then here, in very small print that you can't read, but this will just give you the idea; this is the order of things that you build if you do CKY parsing. So you, first of all, build all of things of span one in each cell, zero to one, one to two, two to three; then the things of span two in each cell. And then, eventually, you build the things of span zero to three in each cell.

Here on this side is the order of things in which you would explore doing uniform cost search. And the important points and notices here that the order in which things explore are mixed up because it does things in different orders.

So it, first of all, does something from zero to one, but the next thing it does is from one to two because there is a high-probability lexical role. And so that's the next thing that comes off the agenda. And when it starts to see something promising, it starts here building a sentence from zero to two long before it's built over one-span constituents that have lower probabilities. And by the time it's got to here, about two-thirds of the way down, it's built from sentence zero to three with the highest score, and it can stop. And so we have saved some work. So if you can see that green, that green is then stuff that we don't have to do that we did with a CKY parser.

Sorry, I think I said that wrong. The green is the stuff that you did have to do, and the rest of the stuff is the stuff you avoided. Okay, so that seemed reasonably good in that example because I avoided about a third of the work. In practice, doing uniform cost search just isn't much dice for doing parsing. And in general, people don't do it. And the reason for that is if you think about it more for a moment, well, what do you explore in uniform cost search? You explore everything that's got high probably but just because of the way probabilities work when you kinda multiply probabilities. And the more that you multiply them, the smaller that they get.

But it turns out that in these parse trees, almost everything small has higher probability than something that is big. That if you have got a two-word constituent, it doesn't matter how bizarre it is, a PCFG is still going to give it higher probability than any ten-word constituent. I mean that's not strictly a theorem, but in practice that's what happens, right. Almost everything small gets built. And so you can actually run this with a broad coverage tree bank, PCFG of the kind that you have built for the assignment. And it turns out that if you do uniform cost search, you save about 10 percent of the work.

And somehow, saving 10 percent of the work doesn't really seem enough motivation to work harder, in particular because the 10 percent of the work is normally at the cost of actually things running slower simply because of the fact that you can write these tight four loops for CKY parsing. In practice, that tends to run much faster than the uniform cost search when you have to fiddle around with priority queues and data structures like that which take longer. Okay, so if you want to pursue this gain, you have to do something more than uniform cost search, and so you want to find some way of searching where you satisfy the goal of speed, that you are building the promising edges and not all of the unpromising edges.

And so clearly, the uniform cost search heuristic isn't enough to do that. And so, preferably, you then also like to build the edges in a good order. So for uniform cost search and some other forms of search, it has the property of correctness, that you can guarantee that you have found the best solution according to your model. And that seems to be in principle a nice property to have, though it's a property that actually quite a bit of work and probabilistic parsing foregoes and instead settles with being able to find good structures most of the time. Okay. So for speeding up agenda-based parsers, one way of doing it is to say do A-star parsing.

So it's like A-star search, apart from [inaudible] over hypergraphs. And so that depends on having a good heuristic for the work still to be done. And so the idea is that we can fix the problems with uniform cost search where every little constituent is being built by in some way saying, yeah, you have built a little constituent, and its probably looks okay, but you have still got a lot of work to do after that before you have a parse with the sentence. And that will cost you, and so it's not actually such a promising thing to build. And that's something I explored with Dan Klein. But in practice, this hasn't been used much.

Most of the time what people actually use is heuristic searches. One of the well-known methods that actually won by Eugene Charniak; we just used his parser, which he calls best first search. But the important thing to remember is best search – best first search isn't. Its best first search, in Eugene Charniak sense, means think it looks best for a search, and that there is absolutely no guarantee that actually that the best thing comes out at the end. Okay. I am going to skip past, and I'll actually explain all of the details of the A-search. But I'll just show you this graph which gives you a sense as to why it's hard to win doing A-star search, and why it's not that widely used in practice.

So what this is showing is – we were wanting to come A-star estimates for the work still to be done. So the idea is that you have built a noun phrase from positions five to ten inside a 20-word sentence. And what you want to do is come up with an estimate of how much work that you'll have to do parse the rest of the sentence, i.e. the stuff from zero to five, and the stuff from ten to twenty, and the cost of a rule that connects it together with what your noun phrase from five to ten. So somehow you have to estimate the cost still to be done to finish parsing the sentence.

And so that's what we tried to do, and we came up with a range of estimates. And so the simplest estimate here, S, was just saying all we know is that there are five words to the left and ten words to the right which you have to incorporate into the parsing sentence. And then we came up with successively more refined estimates. So for SX you also know O, and the constituent that you need to combine it with is a noun phrase. For SXI, you know – oh, and the constituent that you – and the thing to the right of the constituent that you have built is a preposition. And then for B it adds in several more pieces of information.

And what you can see is that as we kind of add in more information and come up with a better estimate, our estimates improve. We get a tighter estimate, and so that the minimum possible cost – sorry, I should explain first. The access here is in low probabilities, and that's why it's negative scores. So the maximum probability completion is going down because we are getting a tighter estimate because we are incorporating more information about the context.

And that means on average the probability cost of completing the rest of the sentence – well, the probability of completing the rest of the sentence will be lower, which is being shown here as a low probability as a lower number. So our estimates are getting better, but there is a huge problem which is – looking at this black line here; this black line is – let's take the actual sentences that we are meant to parse, and the actual constituents that we build, and what estimate we came up for, for the outside work is here. And what – and the actual probability cost of parsing the rest of the sentence – the average probability cost is this black line.

And so what you see is that although our estimates were getting better, and because this is on a large scale, they are getting better by several orders of magnitude better. But the crucial thing to note is that all of these lines are basically parallel at this slope; whereas, the real cost of completing the rest of the sentence is going down here at this fundamentally different slope. And if you think about it, it's kind of obvious why that is. But if you are doing A-star search, all of those theorems you go through in 221 is that your heuristic has to be optimistic in that the heuristic has to always underestimate the true cost of doing the rest of the work.

And so that means necessarily, in every place, rather than saying, well, on average the parse [inaudible] five words like that will cost me probability ten to the minus seven. You always have to say, well, if they are best possible words and the best possible configuration, the cost of including them would only be ten to the minus three and use that as your estimate. So you are always making, in every possible case, the most optimistic possible assumptions about what else remains. And as we come up with better estimates up here, our estimates get closer.

So if the outside span is extremely small, like there are only kind of two or four words left to parse, our estimates can start getting very close to the truth because we can essentially build our estimates, taking account of the words that are outside of what we build. But if the span is large, we can't do that because, well, the only way we could build an accurate outside estimate is actually by parsing the rest of the sentence and finding out how much it cost to parse the rest of the sentence. And, well, it doesn't work if your A-star heurist costs you as much time to compute as actually doing the parsing. Then you don't get any gains.

You have to have a way of computing the A-star heuristic that's relatively cheap. And the way that we did that was we pre-computed A-star heuristics exhaustively for configurations that were conditioned on only a few bits of information, the kind of things that I talked about. Okay. So I guess the – so A-star search is kind of cool, but the – what to take away from this is it's never really going to get you a fast parser. And I mean there is kind of there an interesting conflict, if you are kind of thinking in statistical terms.

But A-star search is always taking the very best possible outcome that could possibly happen, where it's really much more statistical thinking to actually be wondering, well, what's the average behavior that should be happening. And so that's exactly what Charniak's best-first parsing does. So Charniak's best-first parsing is that you estimate for an item about how much you think it will cost you to parse the rest of the sentence. Whether you – about how much it will cost you to parse the rest of the sentence is trying to come up with an accurate estimate of this black line. And Charniak does a good job.

He comes up with an accurate estimate of that black line. And so most of the time his stuff works really great. Now if he messes up, if he explores the wrong thing first, so he finds a way of building a noun phrase from five to ten and then starts building things on top of that. But then somehow he messed up and then later discovers that there is a better way to build that noun phrase from five to ten, then the algorithm's got a problem. You then have to go back and redo all of the work that you have built on top of that noun phrase from five to ten because all of that work is now wrong because you have found a better way of building that noun phrase.

And so you lose the nice property of Dykstra's algorithm or A-star algorithms where you are guaranteed to find the optimal method of doing things without repeat work. But in practice, the algorithm works extremely well because it's really good at finding the right way to make edges and relatively and frequently needs to do that kind of fix-up. Okay. Yeah, so here is just sort of a summary of a few parsers. So people have done different things. So we use this A-star search, which I have now been knocking. Cullins and his parser effectively used a mix between a gender-based parsing and beam parsing.

So it was a gender based on what to explore, and then simultaneously you'll see these beams that limited the amount of things that you explored over any particular span. And then Charniak used this idea of this inadmissible heuristic that I have just talked about. And actually, of the three, his way of doing it is the faster because his inadmissible heuristics are actually really good in practice with coming up with good estimates of the cost of things. And one other idea that's been explored lately in parsing that's given people a lot of speedup goes in a different direction.

And the idea here is that you can guide the work for the parser to do by parsing first with one or more different grammars that are coarser grammars but can, nevertheless, tell you what's useful to explore. So the idea here is, well, for your grammars for the assignment, you were asked to do parent annotation, so you had categories like VP under an S or VP under an NP and things like that. Well, that was good for improving parse accuracy, but the problem is doing that actually causes your parsers to go much slower because you have not got a lot more symbols in your grammar.

And remember, there is also a grammar-constant term in the parsing time, and so you are slowing down based on the fact that it's roughly G-cubed in the number of grammar symbols. So what if you, first of all, parse with a very small grammar that has few symbols in it, so like a basic PCFG where you just throw away all of these refinements if lexicalization, sub-categorization and work out the things that you can build that are kind of likely in that basic PCFG. They'll still probably be good things to explore in the bigger space. And so this has been used very successfully by a couple of recent parsers, Charniak and Johnson's, and Petrov and Klein's.

And then here is a little animation of this from Petrov and Klein which is showing through successive refinements of the grammar which things have been explored. So for the coarsest grammar – let’s try that again. For the coarsest grammar they explore everything. Then they continue to explore in successive phases only the things that have substantial probability in the early phase. So for their most refined grammar, they are actually exploring very few cells in the chart because they have very good idea of which ones have – are likely to be in the final parse.

Okay, so that’s me having rushed through parsing a search. And so now on to the final two topics which are dependency parsing and then discriminative parsing. Pause for a bit. [Inaudible] parsing a search. Okay. Yeah, dependency grammar – so this is a change of tact, though it then does connect up again. So what we have mainly worked on and what modern American linguistics and modern American computational linguistics has mainly worked on is constituency parsing, things like noun phrases, verb phrases, building these constituents, putting them together with context-free grammar rules.

It turns out that idea of context-free grammars and doing things with constituency is actually just a really new idea. So the idea of constituency was basically invented in the 1920s by a British guy, I think his name might have been Wells. I forget it now, but about 1920 it started to be used in some early-American linguistics. And then it was only in the 1950s that Chomsky came up with the Chomsky Hierarchy and the idea of context-free grammar. So overall the idea only has sort of about 80 years of – well 90 now, I guess, 90 years of history and about 60 in a formalized fashion.

So in terms of kind of a whole history of grammar, what people have used for millennia to think of the structure of sentences isn’t that. It’s, instead, the idea of dependency structure. And I’ll just show a picture first. The idea of dependency structure is what you are doing is for words you are saying what they are dependents are. So here is acquired, and so this is a dependent of the subject. This is a dependent of it, 30 percent of American city the object, and here is another dependent of it in March. And then for each of these sure publishing, so this is the kind of headword of the dependent, and it’s got its own dependent which is sure.

So that’s the idea of a dependency grammar representation. It used to be used in American schools, so up until, I don't know, it was the ’50s or ‘60s, all American school children were taught diagramming sentences. I don’t – probably none of you – has anyone ever seen diagramming? Yes. Okay. Some people still got taught diagramming sentences, at least a little. All right, so diagramming sentences was a notation that – I don’t actually know the full history of it, but I mean it got ingrained in American education, and it was kind of uniformly taught in the first half of the 20th Century as a way of sort of understanding sentence structure, though.

So with a lot of the modern reforms, a lot of people never see it these days. But diagramming sentences, it’s a dependency representation. It’s got its own funny ways of drawing the lines at different angles to represent different things. But it’s a dependency representation. This kind of way of kind of drawing kind of arrows is what you most commonly see as dependencies. But the one caution I should immediately give is half of people like draw the arrows one way, and the other half draw the arrows the other way.

So you kind of have to just look and see where the person is drawing the arrows from heads to dependents or dependents to heads because both of them are actually quite commonly used. But hopefully, they are at least consistent in any one diagram. Okay, yeah, so huge history of dependency grammar. So normally the person recognizes first reducing a precise grammatical description of a language is Parnini, who worked on Sanskrit, and he worked all the way from phonology up through syntax. But his syntax is a dependency grammar. Arabic Rumerians working around the 5th to the 9th Century A.D., they all used dependency grammar.

I mean if you look at the Chinese, Japanese grammatical tradition, dependency grammar, and early 20th Century American schools dependency grammar. So basically, everything is dependency grammar. So this sort of modern formalized dependency grammar is often traced to Tezniare's work, which is then from the late '50s around the same time as Chomsky's work. Because of this, actually, dependency grammar was some of the first work those explored in computational linguistics to even in the U.S., so all of those kind of early Soviet work was definitely dependency grammar.

But also in the U.S., David Hayes was one of the sort of first computational linguists in the United States and one of the founders of the Association of Computational Linguistics. And he wrote this 1962 dependency parser for English, which is sort of around the years when some of these other algorithms like CKY were being invented, right. We had those dates of 1960 for John Calkins, '65 for Kassaming Younger. Okay, so that's the history.

Oh yeah, the other footnote is normally when you do dependency parsing, you sort of put this fake extra thing here, which can be either on the left or the right, which has some word like called something like the root or the wall or something. And it just kinda makes the algorithm more straightforward because if you do that, you have the property that every real word is the dependent of something. And so you have this definite place to start from. And so what you want to do is find what is the dependent of this? That's the head of the whole sentence. And then you want to work down, and then recursively find heads downward. Okay.

So how do these two represent – how do these two things relate to each other? And there is sort of a straightforward answer to that, though with a few little gotchas. So officially a CFG has no notion of a head. So you just have A rewrites as BCD, no notion of a head. So if you are in that space, there isn't any real equivalents that you can draw between dependency grammars and CFGs. But in practice, for linguistic purposes, that's kind of not what anybody uses. That people use, in all of our algorithms, [inaudible] parsing, for example, use a notion of heathered CFGs that more or less, informally, people know what the head is.

So that if you have a verb phrase going to a verb, a noun phrase, and a prepositional phrase, in people’s heads and in their imagination from the symbol names, the verb is the head of the role. And so you can formalize that and say, well, let’s have a formalism of a heathered CFG in which in each rewrite role you are making one of the things on the right-hand side as the head of that role. And so that kind of gets you a little bit closer to a dependency grammar. But there is still one other crucial difference, so that in a dependency grammar, like in the picture back here, you have a head and then all its dependents.

And all of its dependents are equivalent, that you can’t have the notion that some of the dependents are kind of privileged in grouping with it before others. In particular, very standardly in constituency representations you have this idea of a verb phrase acquired 30 percent of American city, which is made as a phrase before you can combine it with sure publishing. So you have these intermediate constituents like verb phrase where a word has taking some of its dependents but not all of its dependents. So that’s an idea that you can’t represent in dependency grammar.

So the isomorphism that you get is that if you have headed free structure roles where there is kind of only one level of projection, so that you go from a word to its phrase, like verb-to-verb phrase, and all of the dependents of the verb are contained in the verb phrase. So there is a head marked on level of projection. Then the represent – then what you get is actually isomorphic toward dependency grammar. And it takes a moment to see that because it’s represented very differently, but it ends up isomorphic.

And the idea of how to see that is that here is a word, and here is the headword of its dependent, and to find out what the size of this phrase is what you can then do is just follow dependencies recursively. So studying from the headword of this dependent, you kind of follow recursively all dependents and get the closure with here is just sure publishing, and that’s equivalent to that phrase being something that the verb phrase acquired rewrites as this phrase and then some other phrases over here. So similarly, over here, from acquired you to go here, which at least in this example is marked as the head, and then you can recursively follow along and find the other words.

You have this implicit phrase of 30 percent of American city. And so the kind of – the constituents are – the constituents are implicitly represented. And if they are done like this so that none of these dependency lines cross, they are mappable onto a phrase structure tree. Okay. And before we talked about propagating headwords. Okay, well, now where was I? I was going to say here. Okay, so one way of getting dependencies is that if what you have here is a pin tree, pin trees don’t directly show dependency relations. But you can kind of reconstruct a form of dependency relations where what you do is you have some rules that say what the head of a phrase is, so the VBD is the head of the verb phrase.

And then you propagate the heads just as we did to lexicalize parsing, so this is VP and nouns. And then what you do is you flatten things a little and turn them into dependency relations. So here we have announced yesterday, and announced his resignation, and announced for John – John Smith announced, so that you are taking the pairs of things that you found in the phrase structure tree here of announced resignation, announced yesterday, and turning those into the dependencies.

And if you do that in the most obvious way, you just get dependencies between words, though some work that's exploited the pin tree bank and tried to dependency parsing has then actually typed these dependencies by naming them after the phrase structure rule that they are associated with. So announced his resignation, the type of this dependency is named a VP rewriting as a VBD and P dependency. And so that gives it kind of a name for the dependency based on phrase structure. Okay. But if we have those kind of dependency representations, we can then directly say, well, let's go about doing dependency parsing.

And if we think of those kind of words with arrows between them, representations, that it seems like there are three basic ideas that we can use for dependency parsing. So the first one is the kind of obvious one that we have talked about before; it's this bilexical information. So is dependents a reasonable subject of the verb tend? Is tend for [inaudible] reasonable verb complement of tend? In string, is that a reasonable bilexical dependency? So bilexical dependency is very directly represented in dependency relations. And there probability is a very obvious thing to use for dependency parsing.

It seems like there two other key ideas that you want to represent for dependency parsing. One is the distance of dependencies. So what this picture here is meant to be showing in a comical way is that it turns out that most dependencies are very local, so that here is the subject dependents, and it's got its own dependents, and they appear right next to it. Tend to [inaudible], tend as this dependent, and then the dependent here, and there are various other words. The dependencies are all close by. They are all close by.

You have to have occasionally longer distance dependencies, so from dependents to tend is slightly longer distance because there is this stuff coming in the middle. But the vast majority of dependents are local, so you get an enormous amount of information in dependency parsing by having a strong preference to local dependencies. And then the third feature is valents, where valents is like Collins's sub-categorization features, that you want to have an idea of how many dependents a word should have. So here we have the verb tend, and it's got two dependents. That's good for a lot of verbs.

A lot of verbs, it's bad news if they have zero dependents; it's good news if they have one or two dependents. Some verbs, particular verbs, will take three or four sometimes, but some verbs won't. Here we have got the word the. The doesn't have any dependents, and that's what you'd expect for the word the, no dependents. Here you have got string with one dependent; that's typical. Preposition in, it's – you know that you have messed up pretty much if a preposition doesn't have two dependents. It should have one dependent for its prepositional object, and it should have another dependent for what it's modifying.

So there is a lot of information counting the number of dependencies things have. And so it’s effectively those three sources of information that, by and large, dependency parsers are built around. Okay. So this is the kind of basic generative probabilistic model that people use for dependency grammars. You start at the wall. You generate the headword of the whole sentence based on a probability distribution. You then gradually generate its children to the left side, so you have some kind of mark-off model, star-generation process where you generate children to the left using a probability distribution.

And then you have a chance of stopping, and at some point you stop. You then generate children to the right with some kind of mark-off model with a stopping probability when you have found all of the right-side children. Then for each of those you recurse, and you say for each of those words let me generate its left children; let me generate its right children; and that will stop. And you will effectively bottom out when you get down to words or parts of speech. You bottom out when you get to parts of speech, which necessarily have no left children and no right children, and then you are done.

So this is straightforward parameterization of a probabilistic dependency grammar. And this is showing the second round of the recursion. Okay, and then you return. Okay. So that’s how you generate a sentence. How do you parse a sentence? I mean the first idea that might pop into your head is, okay, well, we have got the words, and so the words are things. And then what we can do is we can combine words. So let’s say that to is the object of take, so we can combine these things, pointing that arrow there and building this constituent takes to, which is headed by takes. And then we’ll kind of repeat over to tango, we can combine them as a dependency and build a constituent headed by to.

And then we can combine those, and we can say to tango is a kind of another infinitive dependent of takes. So we have built a constituent like that. And we continue up to the top of the sentence. But the problem if you do this, and this is the same thing that I mentioned with lexicalized parsing, this algorithm looks very obvious, and it seems like it can’t do any harm. But if you do this algorithm, you have ended up with an into-the-fifth algorithm, again, for doing parsing.

And the reason for that is that in general, once you get into bigger sentences – it isn’t actually true in this example – but in general, when you get to bigger sentences, the head of the constituent that you have built will be somewhere in the middle of the constituent. And that will be true for both this one here, and that one there. You have both the left extent, the right extent, and you have a head that’s in the middle of them. And so you need to keep track of the extents and where the head is in the middle, which in general means that your rule for building things looks like this.

You have got a left extent, a head, a right extent, which has to be the left extent of the next thing, the head of the right thing, and a right extent which you are combining together to produce a new thing. And because you are keeping these five indexes into a sentence that you end up into with an O into-the-fifth algorithm. So that’s bad news. And it seems like dependency parsing shouldn’t actually be that complex because we are not actually making inherent use of phrasal constituents anymore. And it turns out that it isn’t that complex. And you can do dependency parsing in cubic time in the length of the sentence.

And that's something that's been worked on recently, and this is shown in the work of Jason Eisner and Georgia Sutter. And so they present a clever algorithm as to how you can go about thinking about dependency parsing to make it a cubic algorithm. And these same ideas can be extended into lexicalized parsing, as I mentioned earlier. And so I'll just explain this briefly. I'll probably explain it so quickly – I think this is something that you have to stare at for half an hour to actually work it out in your own head before it can possibly make sense. So I think there is probably little chance I could explain it well enough it will make sense and certainly won't today.

But here, very quickly, is the idea of it. The crucial idea of it is you are going to make these different kind of parse items, the trapezoids. And so what the trapezoids are is that the trapezoids are going to be, say, that in your parse you can make something where, on this side, the tall side is the head; the low side is the dependent. And this trapezoid has the property that it contains all of the right dependents of the dependent and all of the left dependents of the prior left dependents of the head before this one in such a way as they touch in the middle. So what's the intuition here?

Intuition here was on the last diagram the reason why we were messed up was that the heads were in the middle of things. And so if the heads are in the middle of things, we kind of have these extra two indices, and it gets turned into an into-the-fifth algorithm. So what if we did some freaky stuff, and instead what we did is we kind of made this piece here into a trapezoid where the trapezoid is saying it's got all of the dependents – I'll say this is the child. If I'll call this parent-child, it's saying it's got all of the left dependents on this side.

It's got some right dependents on this side, and these right dependents stretch into where left dependents of this word begins. And if I cut out that piece and call it a trapezoid, then I will have constructed something where the headword and the dependent were on the edges of it rather than stuck in the middle. And that way I am going to get rid of needing to have these two extra indices. Okay. And so that's what the algorithm does. You have got these two data structures, okay. All right, so you have triangles.

So on triangles, the tall side is the head and these – down on the other side you have got dependents. And the dependents that you have identified are guaranteed to be complete in that they have found all of their own dependents. And then you have trapezoids where trapezoids have the head on the tall side, a dependent on the low side, all of the left children of the dependent have been found. And they can span out to touch some right dependents of the head. But it's still unspecified what is out on this side. Out on this side there may be other stuff.

And this other stuff might be further dependents of the dependent, or it might be further dependents of the head. Okay. So then this is the algorithm when we do that. Okay. We start with the same sentence. We start off, because these are just words, and saying these are completed triangles for the various parts of speech that we can build, things that don't have any more dependents on the left and don't have any more dependents on the right. So then we can start building trapezoids. Okay. So we can say, okay, I can make a trapezoid by combining these two triangles.

This is the head; this is the dependent; and these two – since these two triangles touch – there are no words in between them – I am able to make a trapezoid here. Okay. I can make another trapezoid there. Okay. Because these two triangles touch; head, dependent make a trapezoid. And I can then keep on going. Here is a trapezoid I can make to tango. At this point, I can say, okay, this thing has no left dependents, so I can build a complete triangle here. Here is the head, no more dependents to be found on the left-hand side.

On the right hand – I can also make that triangle on the right-hand side, but then I can also make more trapezoids by combining up. And so I do this ordination where I build triangles and trapezoids. And the crucial thing to notice is that because both of the triangles and the trapezoids have their stuff at the edges of them, that the number of things that I can build by combining trapezoids or combining triangles. That that's all OM-cubed, again, because I am – when I am putting things together, I am putting together two triangles to make a trapezoid, N-cubed, or else I am putting together a trapezoid and a triangle N-cubed.

And so I am doing these N-cubed operations, and so I end up with an N-cubed parser. Hopefully, that is just enough to leave you confused but thinking that there might be a neat algorithm here if you spent enough time trying to understand it. Okay. Yeah, so very quickly then for the rest of this, so when people do dependency parsing, the way people normally score dependency parsing is just with dependency accuracy. So if these are the dependents, and they are labels, you just sort of have the gold answers, the guessed answers, and you sort of say what percentage of them are right as dependents.

So that's the head position, dependent position, and are optionally a label, and you get an accuracy measure. And the reason that seems fine to do is that the number of dependencies is just the number of words in the sentence. So if you put in a root so every word is a dependent, it's a dependent of one thing. So you are just making number of words in this sentence dependency decisions. And so it's sort of fine to score by accuracy. There is no reason to do F-measure. Okay. So there has been a lot of work recently in dependency parsing again, on particulars this Connell competition for a couple of years doing a dependency parser.

One of the big attributes of dependency parsers is that they can be really fast. They can largely be really fast because they avoid the large grammar constant of PCFG parsers. But current results is that dependency parsers are typically in order of magnitude faster than good PCFG-based parsers. So if essentially what you want is dependency-like representation, and you just want to get it quickly, you're really better off using dependency parser. And so lots of work has used a dependency parser.

And so from a kind of meaning extraction applications, dependency representations actually seem much easier and more like the structure that you want to deal with for getting the meaning of things out. So one well-known recent dependency parser is Ryan McDonald's dependency parser, which there are various papers on. I mean an interesting question to ask is how good are these dependency parsers? Are they as good as constituency parsers? And the answer to that seems to be almost, but not quite. So while it is kinda hard to compare the two kinds of parsers based on getting constituency right because the dependency parser doesn't label constituents.

It doesn't say this is a noun phrase; this is a verb phrase. You could measure unlabeled constituency and just say where did it propose constituents? But the easiest way to evaluate them seems to be to say since our PC – lexicalized PCFGs are implicitly headed anyway, why don't we just evaluate their dependency scores and see how they compare on dependency scores. And so if you do that, Ryan McDonald's dependency parser had 90.9 percent dependency accuracy. The Collins 2000 parser had 91.4 percent dependency accuracy, so it is actually slightly better.

I think there is something about the constituency parameterization, which actually is a little bit useful to parsers. But that's not very big difference when it comes down to it. And the next thing that Ryan McDonald immediately did was use scores from Collins's parser as an extra feature inside his dependency parser. And then he was able to get 92.2 percent by taking advantage of it as a feature in his parser. Okay. Maybe I'll skip that. Okay. So for the last few minutes, I then just wanted to say a little bit about the discriminative of parsing, just to give a flavor of that.

So we talked for several lectures in the kind of information extraction sequence model part about how let's build discriminative models of the CMN, EMNN. It's much better than building generative models. And kind of the obvious question is to ask is, well, why not do that for parsing as well? Shouldn't it be better for exactly the same reasons that it's better for doing sequence models and information extraction? And the answer to that is yes, but – but the, yes, it is better, is certainly the case. On the other hand, it's kind of a complex enough thing to do that it's still not yet really kind of real.

That most of the parsers that most people are using most of the time are still generative lexicalized PCFG parsers. So that kind of contrasts sharply with the space of sequence models where for these days most tasks people don't use generative models like HMNs, they use discriminative models, whether SVM based or maximum-entry based like we did; that's just not the case for parsing. Okay. So if you want to do parsing, in the abstract, discriminative parsing it's exactly the same problem, all right. We have got a set of IID samples where we have the input X, which is a sentence; and our class Y that we want to assign using our discriminative function.

The problem is that the class of a sentence is a parse tree, which means that you need to build a model over an enlarged, well actually infinite a number of classes. We have no longer just got – let me label this as personal location or other. Right? You now are trying to label over an infinite number of parse trees. That might appear as if it's just impossible, but it's not impossible. And the reason it's not impossible is we have features, right. The way I defined things that the features were F of XY, they were a feature of the observe data and the class that you wanted to assign.

So what you can do is have features that are senses of to bits of the parse tree, like the feature can say the sentence has the word in, in it, and somewhere in the parse tree there is a prepositional phrase. That's a possible feature. And so the features can check bits of parsers. And so therefore, you can give a score to a parse by having features that look at bits of the parse. And so, effectively, you can do discriminative techniques over this infinite category space of parse trees. Okay. And here is my motivating example which I have borrowed from a similar example from Mark Johnson as to where generative parsers go wrong, and why being able to do discriminative parsing is actually a really good idea.

So let's just look at this example for a minute. So here is my – this is my training tree bank. So I have got a training corpus of 108 imperative sentences; 100 of which are eat, and 6 of which are eating rice with chopsticks, and 2 of which are eat rice with chopsticks where right of the chopsticks is now an NP. So these six have the VP attachment, and these two have the noun phrase attachment of rice with chopsticks. I guess this is really bad in China, all right. You put chopsticks into rice for dead people. So you don't want to get that parse. Okay. Now here is the question.

If this is my training tree bank, and I train a PCFG, and then I give it the sentence to parse of eat rice with chopsticks, what parse will my PCFG return? I mean if you just look at it for a fraction of a moment, you think, well, of course it should return this one because we got this six times, whereas we only got that one twice. But the fact of the matter is that's not what you get out. And the reason that you don't get that out is because of this. And this is sort of a really kind of a weird thing that it takes a minute to wrap your brain around as to how PCFGs end up acting. That you get this really weird effect that because of these trees in the grammar, in a PCFG that actually affects your choice over here.

Whereas, it seems intuitively like this should just have nothing to do with your choice over here. Your choice over here should be based on – this happens three times as often as that, therefore you should choose this parse to eat rice with chopsticks. But why this is wrong is if you look at these sentences, half of the constituents are the same, right. So both of them has a VP goes to VNP; VP goes to VNP. Both of them have a PP that goes to PNP here. And so the differences between the parsers are this one has a VP that goes to VPPP, and this one has an NP that goes to NPPP. Now what happens when you make a PCFG?

Well, for PCFG it says, well, let's look at ways to rewrite a verb phrase. There is a hundred of – sorry, there is 108 VP goes to V's. There are six VP – wait, now let me get this right. There are eight VP goes to VNPs; six here and two there. And there are precisely six VP goes to VPPP. So if you add that all up, you had 108 plus 8 plus 6, 122. The probability of this rule is 6 over 122, small. Whereas, for NP, right, we have these NP rewrites here, which there are six each. We have two more of each of them here, so there are eight each of them.

And then we have two instances NP goes to NPPP so that there are 12 and 4 – there are 18 rewrites of NP and 2 of them up to NPPP. So the probability of this rule is one-ninth. And so the only difference between these PCFG structures is that this one has the VP goes to VPPP role, and this one has the NP goes to NPPP role. And the probability of this role, one-ninth, comes out as higher then probability of this role because of all these constituents over here. And so the PCFG parser will actually return this structure if given that sentence to parse with the construction of PCFG. That seems wonky.

And so that's precisely what you can fix if you do discriminative parsing. The discriminative parser can effectively learn that what it needs to do is be down-weighting the NP goes to NPPP rule, and up-weighting the VP goes to VPPP rule so that it will get this parse choice correct. And you'll still get the choice of every other parse correct at the same time. So it's just kind of like my example with traffic lights that you can change the scores of things in such a way that you can make better discrimination decisions; whereas, a PCFG will get it wrong. Okay.

So staring at that example and convincing yourself that is true, that is my main concept on understanding that there is something to this discriminative parsing idea. I mean the practice – there is then all of the usual stuff I talked about before, you know, that you can put in general kinds of features and stuff like that. Let me skip right to – okay, so how can you do discriminative parsing? There are at least a couple of ways that you can do it, well, at least three. I mean one way is that you can say, well, if I have got something like a parser, I am making a sequence of choices: shift reducers, building constituents.

I can make each of those choices discriminatively; or else you can say, well, surely I can write a dynamic program which will use parsing and use discriminative scoring everywhere; or a third choice is you can say, well, let me parse with a generative parser and get out an invest list. And then I'll re-rank it using a discriminative algorithm because then I'll only have 15 parse candidates to deal with rather than having to deal with this infinite category space. And all of those methods have been used. So an early parser was [inaudible], which was a shift-reduced parser that did discriminative estimation of which decisions to make.

It didn't actually work so well. It actually – it worked okay, but it worked a little bit worse than the best generative parsers around the same year, around '97, '98. And I think the reason for that is that if you just try and do discrimination of the level of individual parse decisions, you make, effectively, very bad – you can make very bad decisions because of the structure of the model that you are using if you think about it as a model class. Because you are kind of making decisions that are independent of the future if you make discriminative decisions. You can do dynamic program-generative discriminative parsers.

This is something that people are working actively on at the moment. I think Ben Tasker's here, well, actually, it wasn't quite the first; it was one of the first that did that. The problem with Ben Tasker's one was it took him kind of three months to train a model that could parse 15-word sentences, discriminative estimation really, really extensive.

People have continued working on that in the sort of [inaudible] to be with better algorithms you can – better algorithms and more distribution out of multiple machines.

It’s starting to get to the point where you can do it more sensibly. The kind of – last one – the kind of current state of the art is – nevertheless, is the Charniak-Johnson parser where it’s then using a maximum entropy discriminative re-ranker on the 50 best that comes out of a Charniak parser. And the other advantage of doing re-ranking is if you do re-ranking, it’s very easy to write global features that look at anything of interest and say whether it was a good or bad parse. And so you can have features that look at big things, like do these two conjoined ten-word phrases look sensible? Whereas, if you are going to stick to dynamic programming, you kind of have to have local features.

So his results – or their results now – are that they can get 91 percent F1 on sentences of all lengths, which is equivalent to 92 point something looking just at sentences up to 40 words. So that’s the current state of the art of parsing is this sort of mixed generative discriminative re-ranking model. Okay. And I’ll stop there. And so that’s the end of parsing, and then go on to semantics and meaning next time.

[End of Audio]

Duration: 78 minutes