MITOCW | Lec-04

PATRICK WINSTON: Today we're going to be talking about Search.

I know you're going to turn blue with yet another lecture on Search.

Those of you who are taking computer science subjects, you've probably seen in 601.

You'll see it again as theory course.

But we're going to do it for a little different purpose.

I want you to develop some intuition about various kinds of Search work.

And I want to talk a little bit about Search as a model of what goes on in our heads.

And toward the end, if there's time, I'd like to do a demonstration for you of something never before demonstrated to a 603.4 class, because it was only completed last spring.

And some finishing touches were added by me this morning.

Always dangerous, but we'll see what happens.

There's Cambridge.

You all recognize it, of course.

You might want to get from some starting position s to some goal position g.

So, you'll hire a cab and hope for the best.

So, here's what might happen, not too hot.

Let's move the starting position over here.

I've had cab drivers like this New York.

But it's not a very good path.

It's the path of a thief.

Let's change the way that the search is done to that of a beginner, an honest beginner.

Not too bad.

Now, let's have a look at how the Search would happen if the cab driver was a Ph.D. in physics after his third post-doc.

These are not actually traverse.

These are just things that the driver is thinking about, and that is the very best of all possible paths.

So, the thief does a horrible job.

The beginner does a pretty good job, but not an optimal job.

This is the optimal job as produced by the Ph.D. in physics after his third post-doc.

So, would you like to understand how those all work?

The answer, of course, is yes.

I'm going to talk to you about procedures that are different from the way that you just solved this problem.

I imagine that if I said to you, please find a path for s to g, you would, within a few seconds, find a pretty good path-- not the optimal one, but a pretty good one-- using your eyes.

And we're not going to tell you about how that works, because we don't know how that works.

But we do know that problem solving with the eyes is an important part of our total intelligence.

And we'll never have a complete theory of human intelligence until we can understand the contributions of the human visual system to solving everyday problems like finding a pretty good path in that map.

But, alas, we can't talk about that, because we don't know how to do it.

We're working on it.

But we don't know how to do it.

So, I'm not going to use Cambridge in my illustrations.

There's too much there to work through in an hour.

So, we're going to use this map over here which has been designed to illustrate a few important points.

You, too, can find a path through that graph pretty easily with your eyes.

Our programs don't have eyes, and they don't have visually grounded algorithms, so they're going to have to do something else.

And the very first kind of search we want to talk about is called the British Museum approach.

This is a slur against at least the British Museum, if not the entire nation, because the way you do a British Museum search is you find every possible path.

So, it'll be helpful to have a diagram of all possible paths on the board.

We're going to start with a British Museum search.

From the starting position, it's clear, you can go from my s to either a or b.

And already there's an important quiz point.

Whenever we have these kinds of problems on a quiz, we ask you to develop the tree associated with a search in lexical order.

So, the nodes there under s are listed alphabetically, just to have an orderly way of doing it.

So, from a we can go either b or d.

And another convention of the subject, another thing you have to keep in mind in quizzes, is it we don't have these searches bite their own tail.

So, I could have said that if I'm at a, I can also go back to s.

But no path is ever allowed them to bite itself, to go around and enter and get back to a place that's already on the path.

Now if I go on to b first, that means that from b I can go to either a or c.

This is getting fat pretty fast.

But let's see, s, a, b.

The only place I can go is c and then to e.

s, a, d, without biting my own tail and going back to a, the only place I can go is g.

s b, a, I can only go to d and then to g.

And finally, s, b, c, I can only go to e.

So, that is a complete set of paths as produced by any program that you will feel you'd like to write that finds all possible paths.

I haven't been very precise about how to do that, because you don't have to be.

You can't save much work by being clever, because you have to find everything.

So, that's the British Museum expansion of the tree.

So, what have I done?

I've been playing around with a map.

I showed you an example of a map.

And pretty soon you're going to think that Search is about maps.

So, before going even another tiny step, I want to emphasize that Search is not equal to maps.

Search is about choice.

And I happen to illustrate these searches with maps, because they are particularly cogent.

But Search is not about maps.

It's about the choices you make when you're trying to make decisions.

These things I'm going to be talking to you about today are choices you make when you explore the map.

You can make other kinds of choices when you're exploring other kinds of things.

And, in fact, at the end, if there's time, I'll show you how you do searches when you're solving problems in a humanities class.

That's the British Museum algorithm.

Search is not about maps.

Our first gold star idea, Search is about choice.

But for our illustration, Search is about maps.

So, the first kind of Search we want to talk about that's real is Depth-first Search.

And the idea of Depth-first Search is that you barrel ahead in a single-minded way.

So, from s, your choices are a or b.

And you always go down the left branch by convention.

So, from s, we go to a.

From a we have two choices.

We can go to either b or d following our lexical convention.

After that, we can go to c.

And after that we can go to e.

And too bad for us, we're stuck.

What are we going to do.

We've got into a dead end, all is lost.

But of course, all isn't lost.

Because we have the choice of backing up to the place where we last made a decision and choosing another branch.

So, that process is called variously back-up or backtracking.

At this point, we would say, ah, dead end.

The first place we find when we back up the tree where we made a choice is when we chose b instead of d.

So, we go back up there and take the other route.

s, a, d now goes to g.

And we're done.

We're going to make up a little table here of things that we can embellish our basic searches with.

And one of the things we can embellish our basic searches with is this backtracking idea.

Now, backtrack is not relevant to the British Museum algorithm, because you've got to find everything.

You can't quit when you've found one path.

But you'd always want to use backtracking with Depth-first Search, because you may plunge on down and miss the path that gets to the goal.

Now, you might ask me, is backtracking, therefore, always part of Depth-first Search?

And you can read textbooks that do it either way.

Count on it.

If we give you a Search problem on a quiz, we'll tell you whether or not your Search is supposed to use backtracking.

We consider it to be an optional thing.

You'd be pretty stupid not to use this optional thing when you're doing Depth-first Search.

But we'll separate these ideas out and call it an optional add-on.

so, that's Depth-first Search, very simple.

Now, the natural companion to Depth-first Search will be Breadth-first Search, Breadth-first.

And the way it works is you build up this tree level by level, and at some point, when you scan across a level, you'll find that you've completed a path that goes to the goal.

So, level by level, s can go to either a or b.

a can go either to b or d.

And b can go to either a or c.

So, you see what we're doing.

We're going level by level.

And we haven't hit a level with a goal in it yet, so we've got to keep going.

Note that we're building up quite a bit of stuff here, quite a lot of growth in the size of the path set that we're keeping in mind.

At the next level, we have b going to c, d going to g, a going to d, and c going to e.

And now, when we scan across, we do hit g.

So, we found a path with Breadth-first Search, just as we found a path with Depth-first Search.

Now, you might say, well, why didn't you just quit when you hit g?

Implementation detail.

We'll talk about a sample implementation.

You can write it in any way you want.

But now that we know what these searches are, let's speed things up a little bit here and do a couple searches that now have names.

The first type will be Depth-first, boom.

That's the one that produces the thief path.

And then we can also do a Breadth-first Search, which we haven't tried yet.

What do you suppose is going to happen?

Is it going to be fast, slow, produce a good path, produce a bad path?

I don't know, let's try it.

I had to speed it up, you see, because it's doing an awful lot of Search.

It's generating an awful lot of paths.

Finally, you got a path.

Is it the best path?

I don't think so.

But we're not going to talk about optimal paths today.

We're just going to talk about pretty good paths, heuristic paths.

Let's move the starting position here in the middle.

Do you think Breadth-first Search is going to be stupid?

I think it's going to be pretty stupid.

Let's see what happens.

This Search is a lot to the left, which you would never do with you eye.

Let me slow that down just to demonstrate it.

It finds a shorter path, because it's right there in the middle.

But it spends a lot of its time looking off to the left.

It's pretty stupid.

But that's how it works.

So, now that we've got two examples of searches on the table, I'd like to just write a little flow chart for how the search might work.

Because if I do that, then it'll be easier for us to see what kind of small differences there are between the implementations of these various searches.

So, what we're going to do is we're going to develop a waiting list, a queue, a line, whatever you'd like to call it.

Let's call if a queue.

We're going to develop a queue of paths that are under consideration.

So, the first step in our algorithm will be to initialize our queue.

And I think what I'll do is I'll simulate Depth-first Search on this problem up there on the left using this algorithm.

I need to have some way of representing my paths.

And what I want to do is I'm going to betray my heritage as a list programmer, because I'm just going to put these up as if there were lisp s-expressions.

To begin with, I just have one path.

And it has only one node in it, s.

That's the whole path.

The next thing I do after I initialize the queue is I extend first path on the queue.

OK, when I extend s, I get two paths.

I get s goes to a, and I get s goes to b.

I take the first one off the front of the queue.

And I put back the two that are produced by extending that path.

Now, after I've extended the first path on the queue, I have to but those extended paths on to the queue.

In here there's an explicit step where I've checked to see if that first path is a winner.

If it's not, I extend it.

And I have to put those paths onto the queue.

So, I'll say that what I do is I end queue.

Now, I've done one step.

And let's let me do another step.

I'm going to take this first path off.

I'm going to extend that path.

And where do I put these new paths on the queue if I'm doing Depth-first Search?

Well, I want to work with the path that I've just generated.

I'm taking this plunge down deep into the search tree.

So, since I want to keep going down into the stuff that I just generated, where then do I want to put these two paths?

At the end of the queue?

I don't think so, because it'll be a long time getting there.

I want to put them on the front of the queue.

For Depth-first Search, I want to put them on the front of the queue.

And that's why s, a, b goes here, and s, a, d, and then that's s, b.

So, s, b is still there.

That's still a valid possibility.

But now I've stuck two paths in front of it, both of the ones I generated by taking a path off the front of the queue, discovering that it doesn't go to the goal, extending it and putting those back on the queue.

I might as well complete this illustration here.

While I'm at it, I take the s, a, b off, s, a, b, and I can go only there to c.

But, of course, I keep s, a, d and s, b on the queue.

Now, I take the front off the queue again, and I get s, a, b, c, e, and not to forget s, a, d and s, b.

I take the first one off the queue.

It doesn't go to the goal.

I try to extend it, but there's nothing there.

I've reached a dead end.

So, in this operation, all I'm doing is taking the front one off the queue and shortening the queue.

We're almost home.

I take s, a,d off of queue.

And I get s, a, d, c.

And, of course, I still have s, b.

Now, the next time I visit the situation, buried in that first step, I discover a path that actually does get to goal, and I'm done.

So, each time around I visualize the queue.

I check to see if I'm done.

If not, I take the extensions and put them somewhere on the queue.

And then I go back in.

And then here there's a varied test which checks to see if we're done.

That's how the Depth-first Search algorithm works.

And now, would we have to start all over again if we did Breadth-first Search?

Nope.

Same algorithm.

All the code we've got needs one line replaced, one line changed.

What do I have to do different in order to get a Breadth-first Search out of this instead of a Depth-first Search?

Tanya?

TANYA: Change [INAUDIBLE] on the queue.

PATRICK WINSTON: And where do I put it on the queue?

She says to change it.

TANYA: On the back?

PATRICK WINSTON: Put it on the back.

So, with Breadth-first Search all I have to do is put on the back.

Now, if we were content with a inefficient search, and didn't care much about how good our path was, we'd be done.

And we could go home.

But we are a little concerned about the efficiency of our search.

And we would like a pretty good path.

So, we're going to have to stick around for a little while.

Now, you may have noticed, up there in that the development of the Breadth-first Search, that the algorithm is incredibly stupid.

Why is the algorithm incredibly stupid?

Ty, what do you think?

TY: It can't tell whether it's getting closer or further away from the goal.

PATRICK WINSTON: It certainly can't tell whether it's getting closer or further away from the goal.

And we're going to deal with that in a minute.

But it's even stupider than that.

Why is it stupid?

What's your name?

DYLAN: Dylan.

It [? hits ?] the same nodes twice.

PATRICK WINSTON: Dylan said it's extending paths that go to the same node more than once.

Let's see what Dylan's talking about.

Down here, it extends a.

But it's already extended a up there.

Down here, it extends a path that goes to b.

And it's already extended a path that goes to d.

Over here, it could extend a path that went through c, but it's already got a path that goes through c.

So, all of these paths are duplicated.

And we're still going through them.

That's incredibly stupid.

What we're going to do is we're going to amend our algorithm just a little bit.

And we're not going to extend the first path on the queue unless final node never before extended.

What we're going to do is we're going to look to see if there-- we've got this path.

And we're going to extend it.

And it's got a final note.

If we've ever extended a path that goes to that final node, and it was a final node on that path, then we're not going to do it again.

We got to keep a list of places that have already been the last piece of a path that was extended.

Everybody got that?

It's a little awkward to say it, because it's the last node we care about.

If a path terminates in a node, and if some other path previously terminated in that node and got extended-- we're not going to do it again.

Because it's a waste of time.

Now, let's see if this actually helps.

Now, use the extended list.

Let's see, well, gee, we got that place in the center there.

Let's just repeat the previous search.

Wow, it's taking a long time.

But notice it put 103 paths back on the queue.

Now, let's add a filter and try again.

A lot less.

So, let's speed this up, and we'll start way over here.

You remember how tedious that search was.

And now we'll repeat it with this list, boom, there it is.

That's all because we didn't do that silly thing of going back through the final node that's already been gone through.

So, you would never not want to do this.

We better list this as another option.

It doesn't help with a British Museum algorithm, because nothing helps with the British Museum algorithm.

Does it help with Depth-first?

Yes.

Does it help with Breadth-first?

Yes.

Do we do backtracking with Breadth-first?

No, because backtracking can't do us any good.

OK, we're almost, except that search that's starting in the middle is still pretty stupid.

Both the Breadth-first version and the Depth-first version are going off to the left.

And we would never do that with our eyes in any case.

The next thing we want to do is we want to have ourselves a slightly more informed search by taking into consideration whether we seem to be getting anywhere.

So, in general, it's a good thing to get closer to where we want to go.

In general, if we've got a choice of going to a node that's close to the goal or a node that's not so close to the goal, we'll always want to go to the one that's close to the goal.

And as soon as we add that to what we're doing, we have another kind of Search, which goes by the name of Hill Climbing.

And it's just like Depth-first Search, except instead of using lexical order to break ties, we're going to break ties according to which node is closer to the goal.

I went to some trouble to talk to you about this enqueued list.

And having gone to that trouble, I'm now going to ignore it.

Not because it isn't a good idea, but because trying to keep track of everything in the example is confusing the example.

It won't work out right in the small example and all that.

Put the queueing thing aside, queued list aside, and think instead just about the value of going in the direction that's getting us closer to the goal.

In Hill Climbing Search, just like a Depth-first Search, we have a and b.

And we're still going to list them lexically on underneath the parent node.

But now which one is so closer to the goal?

Now, this time b is closer to the goal than a.

So, instead of following the Depth-first course, which would take us down through a, we're going to go to the one that's closest which goes through b.

And b can either go to a or c.

b is six units away from the goal. a is about seven plus, not drawn exactly to scale.

Use the numbers not your eyes.

Now where are we?

It's symmetric, so a and c are both equally far from the goal.

Now we're going to use the lexical order to break the tie.

Now from s, b, a, we'll go to d.

And now, which is closest to the goal?

That's the only choice we have.

So, now we have no choice but to go down to the goal.

That's the Hill Climbing way of doing the search.

And notice that this time there's no backtracking.

It's not the optimal path.

It's not the best path.

But at least there's no backtracking.

That's not always true.

That's just an artifact of this particular example.

Do you think Hill Climbing would produce a faster search?

I think so.

Let's see what happens when we add these things at one at a time.

First, let's turn off our extended list.

We turned off our extended list.

And we're going to do Depth-first again just for the sake of comparison.

It produces a very roundabout path with 48 enqueueings.

Now, let's switch over to Hill Climbing.

And what do think?

Do you think it will produce a straighter path, fewer enqueueings?

Boom.

You wouldn't not want to do that, would you?

If you've got some kind of heuristic that tells you that you're getting close to the goal, you should use it.

Now, it's easy to modify my example over there so that getting close to the goal gets you trapped in a blind alley on e.

That's easy to do.

But that's just an artifact of the example.

In general, you want to go along a path that gets you closer to the goal.

So, that's 23.

I don't know, let's see if using the extended list filter does any good.

Yeah, still 23.

So, in that particular case the extension list didn't actually do us any good, because we're driving so directly toward the goal.

OK, that's that.

Now, let's see, is there any analog to-- well, we might say that this is yet another way of distinguishing the searches.

And that is, is it an informed search?

Is it making use of any kind of heuristic information?

Certainly, a British Museum is not, Depth is not, Breadth is not.

And now let's consider what we got for Hill Climbing.

Do we want to use backtracking?

Sure.

Do we want to use an enqueued list?

Sure.

And it is informed, because it's taking advantage of this extra information.

It may not be in your problem.

It's not often the case you've got this information in a map.

Your problem may not have any heuristic measurement of distance to the goal.

In which case, you can't do it.

But if you've got it, you should use it.

Oh, yeah, there's one more.

And I've already given it away by having it on my chart.

It's called Beam Search.

And just as Hill Climbing is an analog of Depth-first Search, Beam Search is a complement or addition of an informing heuristic to Breadth-first Search.

What you do is you start off just like Breadth-first Search.

But you say I'm going to limit the number of paths I'm going to consider at any level to some small, fixed number, like, in this case, how about two.

So, I'm going to say that I have a Beam of two for my Beam Search.

Otherwise, I proceed just like Breadth-first Search, b, d, a, g.

And now I've got that stupid thing where I'm duplicating my nodes, because I'm forgetting about the enqueued list.

But to illustrate Beam Search, what about I'm going to do now is I'm going to take all these paths I've got at the second level, and I'm only going to keep the best two.

That's my beam width.

And the best two are the two that get closest to the goal.

So, those four, b, c, a, and d, which two get closest to the goal?

Now, b and d.

These guys are trimmed off.

I'm only keeping two at every level.

Now, going down from b and d, I have, at the next level, c and g.

And now I've found the goal.

So, I'm done.

We could do that here, too.

We could choose a Beam Search, not bad.

Let's see, let's try this thing from the middle.

Let's slow my speed down a little bit.

Now, are we going to see anything going off to the left like we did with ordinary Breadth-first Search?

No, because it's smart.

It doesn't say, I want to go to a place that's further away from my goal.

Now, let's see, maybe we can go back to our algorithm now and talk about that enqueueing mechanism and talk about Hill Climbing.

Can I use the same basic search mechanism, just change that one line again?

Yes.

How do I add new paths to the queue this time?

Well, it's very much like Hill Climbing, right?

I want to add them to the front but with one little flourish.

What's the flourish?

[? Krishna, ?] what do you think?

Remember, I want to use my heuristic information.

So, I not only add them to the front, but amongst the ones I'm adding to the front, what do I do?

AUDIENCE:Check the distance?

PATRICK WINSTON: Check the distance.

And how do you arrange them?

AUDIENCE:[? You ?] [? keep the ?] minimum [? first. ?] PATRICK WINSTON: Yeah, you can put the minimum first if you like.

But let's sort them.

We'll sort them, that will keep everything straight.

So Hill Climbing is front-sorted.

And, finally, how about Beam?

What do we do with Beam Search to add them to the queue?

Well, it doesn't matter where we add them, because all we're going to do is we're going to keep the w best.

So, with Beam, we'll just abbreviate that by saying keep w best.

Now, you have some of the basic searches in you're toolkit.

There's one more that's sometimes talked about.

We've got Depth, Breadth, Best, and Beam, one more is Best, Best-first Search.

It's a variant where you say, I've got this tree.

It's got a bunch of paths that terminate in leaves.

Let me just always work on the leaf node that's closest to the goal.

It can skip around a little bit from one place to another.

Because as it pursues one path, it may not do very well in some other path quite distant.

And the tree will become the best one.

We've actually seen an instance of that in then integration program.

It's capable of skipping all over the place, because it's always taking the easiest problem in the search tree, in the and/or tree, working on that.

That's Best-first Search.

You can do these sorts of things in continuous spaces, too.

And you've done the mathematics of that in 1802 or something.

But in continuous spaces, the Hill Climbing sometimes leads to problems or doesn't do very well.

What kind of a problem can you encounter in a continuous space with Hill Climbing?

Well, how would you do Hill Climbing in a continuous space?

Let's say we're in the mountains, and a big fog has come up.

We're trying to get to the top of the hill before we freeze to death.

And we take a few steps north, a few steps east, west, and south using our compass.

And we check to see which direction seems to be doing the best job of getting us moving upward.

And that's our Hill Climbing approach, right?

We have explored four directions we can go and pick the best one.

And from there, we pick four, try all those, pick the best one, and away we go.

We've got ourselves a Hill Climbing algorithm.

What's wrong with it?

Or what can be wrong with it?

Sometimes it works just fine.

Yes.

SPEAKER 1: You might get stuck in a local maximum.

PATRICK WINSTON: We might get stuck in a local maximum.

So, problem letter a is that if this is your space, it may look like that.

And you may get stuck on a local maximum.

Is there any other kind of problem that can come up?

Well, it all depends on what the space is like.

Here's a problem where the space has local maxima.

Now, a lot of people have been killed on Mt.

Washington when the fog comes up.

And they do freeze to death, why?

The reason they freeze to death is the Hill Climbing fails them, and they can't get to the top to the ranger station.

And the reason is that there are large lawns on the shoulders of Mt.

Washington.

It's quite flat.

So, it's the telephone pole problem.

That space looks like this.

Well, this isn't what Mt.

Washington looks like.

But it's the telephone pole problem.

So, when you're wandering around here, the idea of trying a few directions and picking the one that's best doesn't help any, because it's flat.

That can be a problem with Hill Climbing.

Now, there's one more problem with Hill Climbing that most people don't know about.

But it works like this.

This is a particularly acute problem in high dimensional spaces.

I'll illustrate it here just in two.

And I'm going to switch from a regular kind of view to a contour map.

So, my contour map is going to betray the presence of a sharp bridge along the 45 degree line.

Now you see how you can get in trouble there.

You get in trouble, because if you take a step in each direction, every direction takes you downhill.

And you think you're at the top.

So, suppose you're right here and you go north.

That takes you down over a contour line.

If you go south, that also takes you down over contour lines.

Likewise, going west and east all appear to be taking you down, whereas, in fact, you're climbing a ridge.

And that contour line is the highest that I've shown.

So, sometimes you can get fooled-- not stuck, but fooled-- into thinking you're at the top when you're actually not.

Now, this is a model something.

This subject is about modeling intelligence.

And this is a kind of algorithm you frequently need in order to build an intelligent system.

But do we have any kind of Search happening in our heads?

If we're going to model what goes on inside our heads, do we have to model any kind of searching in order to do the kinds of things that we humans do?

I suppose so.

Anytime we make a plan, we're actually evaluating a bunch of choices and seeing how they work.

Let me see if I can illustrate it another way.

This is a system that I showed you a little bit of last time.

And, shoot, I might as well review one or two things here.

I showed you a Macbeth story.

This is the story I showed you.

And if you had this in a humanities class, the simplest questions that might be asked is why did Macduff kill Macbeth down there at the bottom?

Did I demonstrate the answering of questions last time, or just the development of the graph?

I can't remember.

But we'll do it again, anyway.

This is somewhat stylized English.

Just so you'll know, it doesn't have to be stylized English.

This is English that's made available to the Genesis system by way of something called Story Workbench.

There's no free lunch.

Either you can use your human resources to rewrite the plot in third grade English.

Or you can use your human resources to take a more natural, adult-type version of the story and decorate it with annotations that make it possible to absorb it.

Just this summer, in a miracle of summer [? Europe, ?] [? Brit ?] [? van ?] [? Zijp-- ?] one of you-- connected these two systems together.

So, we can now work with stories that are expressed in pretty natural English.

Everything in our system is expressed in English, including common sense knowledge-- like if somebody kills you, you're dead-- but more importantly, for today's illustration, that reflective level knowledge, that knowledge about what revenge is.

Here you are.

You're in the humanities class, and someone says, what's really going on in the story?

Not the details of who kills whom, but is there a Pyrrhic victory?

Does somebody have a success?

Is there an act of revenge?

These are all kinds of things you might be asked about in some kind of humanities class.

So, let me fire up the genesis system.

Pray for internet connectivity.

Launch the system on a read of that Macbeth story that I showed you just a moment ago.

At the moment, it's absorbing information about background knowledge, and about reflective level knowledge, and all that sort of thing.

It's building itself this thing we call an elaboration graph.

It's not quite there yet.

It's still reading background knowledge.

Now it's reading Macbeth.

It's building it's elaboration graph, the same thing you saw last time, except not quite.

Do you see that stuff down at the bottom?

Those are higher level concepts that it's managed to find in the Macbeth story.

So, its found a revenge.

How did it do that?

It searched.

It had a description of what a revenge is, and it looked to see if that pattern was exhibited in the elaboration graph.

So, in a combination of things that were said explicitly and things that were produced by knee-jerk if/then rules, the elaboration graph was sufficiently instantiated that the revenge pattern could be found.

That's interesting, Pyrrhic victory is a little harder.

You'd probably get an a if you said, oh, there's a Pyrrhic victory in here.

There it is.

So, I'll blow that up a little bit so you can see what that is.

You know what a Pyrrhic victory is.

It's a situation where everything seems to be going good at first, and then not so hot.

So, Macbeth wants to be King down here.

And eventually that leads to becoming King.

But too bad for Macbeth, because eventually he gets killed in consequence.

So, it's a Pyrrhic victory.

All that produced by Search programs who are looking through this graph.

Now once you've got the capability of doing that, of course, then you can find all sorts of things.

And you can report them in English.

But, more interestingly, you can answer questions.

Why did Macbeth-- it cares not a hoot about capitalization.

ARTIFICIAL INTELLIGENCE: On a common sense level, it looks like Dr. Jekyll thinks Macduff killed Macbeth because Macbeth angered Macduff on a reflective level.

It looks like Dr. Jekyll thinks Macduff killed Macbeth as part of acts of mistake, Pyrrhic victory, and revenge.

PATRICK WINSTON: Pretty corny speech output.

But you see the point.

How did it get the stuff on the common sense level?

The same way all those programs that build goal trees report, answers the questions.

It's just looking locally around in the connections in the goal tree.

How did it get the stuff on the reflective level?

By reporting on the searches that produced information-- it does that by looking for higher level thoughts about its own thoughts and reporting in which of those higher level thoughts the incident we asked about actually occurs.

So, let's see, just for fun, we might be interested in why Macbeth murdered Duncan.

Wouldn't this be handy if you hadn't actually read the play, and here it is, you've got to write that paper?

ARTIFICIAL INTELLIGENCE: On a common sense level, it looks like-- PATRICK WINSTON: I'll pull the plug on that, because that's just annoying.

Yeah, pretty good, Macbeth wants to be King, and Duncan is the King.

Let's see, why did Macbeth become King?

Oh, it won't answer the question unless I spell it right.

I wouldn't be able to show that to you until last spring.

In fact, I wouldn't have been able to show you this today until last week with a tweak this morning.

Because we've just now connected the language output to, of course, [? Cass's ?] parser system, which is running in reverse, in order to generate that English.

So, that's something that has never before been seen by any eyes but me.

So, that will conclude what we have to do today.