Course Notes of MIT 6.034 Artificial Intelligence [1]

Yuwei Yin[2]

July 12, 2022

[1]Fall 2020. Instructor: Patrick H. Winston. MIT OpenCourseWare: https://ocw.mit.edu/courses/6-034-artificial-intelligence-fall-2010/

[2]https://yuweiyin.github.io/

Contents

1

Introduction and Scope

https://www.youtube.com/watch?v=TjZBTDzGeGg

1.1 Concepts & Insights

AI models are model of thinking.

MIT approach is to build models that we can use to explain the past, predict the future, understand the subject, and control the world. That's what MIT is about.

MIT is about two things. It's about skill building, and it's about big ideas. So you can build the skill at home, or at Dartmouth, or at Harvard, or at Princeton, or all those kinds of places. But the experience you can only get at MIT. I (Prof. Winston) know everybody there is to know in artificial intelligence. I can tell you about how they think. I can tell you about how I think. And that's something you're not going to get any other place.

Artificial Intelligence: Algorithms/procedures/methods enabled by constraints exposed by representations that support models targeted at thinking, perception, and action (with loops that tie all the thinking, perception, and action together).

Generate and Test method: generate some possible solutions and feed them into a box that tests them. Most of the generated solutions are failure, but every once in a while we get something that succeeds.

Rumpelstiltskin principle: Once you can name something, you get power over it. You can start to talk about it. And that vocabulary gives you power. Symbolic labels give us power over concepts.

The value and power of using personal names and titles is well established in psychology, management, teaching and trial law. It is often referred to as the "Rumpelstiltskin principle".

---

simple $\neq$ trivial: simple but powerful; trivial is not only simple but of little worth. For example, "generate and test" is a trivial idea.

Many MIT people miss the opportunities, because they have a tendency to thing that ideas aren't important unless they are complicated. But the most simple ideas in AI are often the most powerful.

---

representation right –> almost done: If you've got your representation right, you're often almost done.

We solve problems with our eyes, as well as our symbolic apparatus.

If we're engineers, it's for building smarter programs. It's about build a tool kit of representations and methods that make it possible to build smarter programs.

If you're a scientist, you're interested in what it is that enables us to build a computational account of intelligence.

1.2 History of AI

Lady Ada Lovelace, the world's first programmer, who wrote programs about 100 years before there were computers to run them.

In 1842, people were hassling her about whether computers could get really smart. And she said, "The analytical engine has no pretensions to originate anything. It can do whatever we know how to order it to perform."

- 1950 - 1960: Age of Speculation
- 1960 - 1980: Dawn Age
- 1980 - 1990
- 1990 - 2000: Bulldozer age
- 2000 - 2010: Imagination: asking a system to imagine something.

1950 - Alan Turning: Turing Test (The Imitation Game) [1].

1960 - Marvin Minsky: Steps toward Artificial Intelligence [2].

AI program examples in the early Dawn Age: integration program, the Eliza (conversation bot), geometric analogy. (purely symbolic reasoning)

AI program examples in the late Dawn Age: vision (perceptual apparatus), learning from a small number of examples, **rule-based expert systems** (e.g., a program written at Stanford that did diagnosis of bacterial infection of the blood.)

AI program examples in the 1980s: Bizs system (to park aircraft effectively)

AI program examples in the Bulldozer Age (the time when people began to see that we had at our disposal unlimited amounts of computing. And frequently you can substitute computing for intelligence.): **Deep Blue** (It processes data like a bulldozer processes gravel so that beat the world chess champion.)

1.3 AI and Paleoanthropology

It's becoming increasingly clear why we're different from chimpanzees, and how we got to be that way.

The high school idea is that we evolve through a slow, gradual, and continuous improvement. But that doesn't seem to be the way it happened. There are some characteristics in our species that are informative when it comes to guiding the activities of people.

And here's what the story seems to be, from the fossil record.

First of all, we human has been around for maybe 20,000 years in our present anatomical form. If someone walk through the door right now from 200,000 years ago, I imagine they would be dirty, probably naked, too. But other than that, you wouldn't be able to tell the difference.

And so the ensuing 150,000 years was a period in which we **humans didn't actually amount to much**. But somehow, shortly before 50,000 years ago, **some small group of us developed a capability that separated us from all other species**.

It was an accident of evolution. And these accidents may or may not happen, but it happened to produce us. It's also the case that we probably necked down as a species to a few thousand, or maybe even a few hundred individuals, something which made these accidental changes, accidental evolutionary products, more capable of sticking.

This leads us to speculate on what it was that happened 50,000 years ago. And paleoanthropologists, Noam Chomsky, a lot of people reached similar conclusions – "It seems that shortly before 50,00 years ago, some small group of us acquired the ability to **take two concepts**, and **combine them to make a third concept**, without disturbing the original two concepts, without limit." (quote from Noam Chomsky).

And from a perspective of an AI person, what Chomsky seems to be saying is, we learned how to begin to **describe things**, in a way that was intimately **connected with language**. And that, in the end, is what separates us from the chimpanzees.

So you might say, well, let's just study language. No, you can't do that, because we also think with our eyes (vision).

Language is at the center of things because it enables storytelling going up, and marshalling the resources of the perceptual apparatus, going down.

Specifically, language does two things:

1. it enables us to make descriptions, and descriptions enable us to tell stories. And storytelling and story understanding is what all of education is about. That's going up.

2. And going down, it enables us to marshal the resources of our perceptual systems, and even command our perceptual systems to imagine things we've never seen.

1.4 Courses Structure

1. **Lectures** are supposed to be an hour about introducing the material and the big picture. They're about powerful ideas, and the experience side of the course.

2. **Recitations** are for buttressing and expanding on the material, and providing a venue that's small enough for discussion.

3. **Mega recitations** are usual components of the course. They're taught at the same hour on Fridays. And the TA will show you how to solve past quiz problems.

4. **Tutorials** are about helping you with the homework.

2

Reasoning: Goal Trees and Problem Solving

https://www.youtube.com/watch?v=PNKj529yY5c

2.1 Introduction: An Integration Problem

What we're going to talk about today, is goals. Let's look at an integration problem:

$$\mathcal{I} = \int \frac{-5x^4}{(1-x^2)^{5/2}} dx \tag{2.1}$$

The question is, if a program can do that, is this program, in any sense of the word, intelligent?

Today we're going to be modeling a little bit of human problem solving, the kind of that is required when you do symbolic integration.

How do we human do it, and is the problem solving technique, that we are trying to model by building a program that does symbolic integration, a common kind of description of what people do when they solve problems?

The answer to this question is yes. It is a very common problem solving that we all engage in without thinking about it and without having a name for it. But once we get a name for it, we'll get power over it (Rumpelstiltskin principle). And then we'll be able to deploy it, and it will become a skill. We'll not just witness it, we'll not just understand it, we'll use it instinctively, as a skill.

You try to apply a transform, and make it into a different problem that's easier to solve. And eventually, what you hope is that you'll simplify it sufficiently, that the pieces that you've simplified to will be found in some small table of integrals.

One of the interesting questions is, how many elements have to be in that table to get an A in the integral calculus course?

What we're going to do is very simple. we're going to convert the problem we're given into another problem that's simpler. This process is named as "**problem reduction**".

Some of these transformations are extremely simple and always safe. Some of them are not so. Now I'm going into some grungy detail. But why do I need to do this?

Here's the educational philosophy. At one level, you want to have a **skill**. But if you're going to have a skill, you have to **understand** it one level down. If you're going to understand it, you have to have **witnessed** it on a level lower than understanding.

So I'm not just going to talk about the idea of problem reduction. Instead, I'm going to show you a particular example of it, so you understand it better. And I'm going to show you the detail at an even lower level than that.

So you will witness the stuff that makes it possible to understand, and understand the stuff that makes it possible to build a skill. So that's why I'm going through the grungy detail.

For that integration problem, what are some simple and safe transformations? There are a few examples.

1) Whether you use a separate transformation or not, of course depends on how you represent the knowledge. And all of this knowledge was written in an early form of Lisp. As a consequence, the way in which minus "$-$" was represented is different from the way -1 is represented. So we need the following transformation:

$$\int -f(x)dx = -\int f(x)dx \tag{2.2}$$

2) Take the constants out:

$$\int cf(x)dx = c\int f(x)dx \tag{2.3}$$

3) The sum of integrals is the integral of the sum.

$$\int \sum f(x)dx = \sum \int f(x)dx \tag{2.4}$$

4) If the degree of the numerator is greater than the degree of the denominator, then it's a knee-jerk always win, you can divide it out.

$$\int \frac{P(x)}{Q(x)}dx = \text{DIVIDE} \tag{2.5}$$

The above four transformations forms the core of an integration program designed by Jim R. Slagle in 1960. So far we have:

$$\mathcal{I} = \int \frac{-5x^4}{(1-x^2)^{5/2}} dx = -5 \int \frac{x^4}{(1-x^2)^{5/2}} dx \tag{2.6}$$

Now we need some heuristic transformations. "Heuristic" is a funny word, meaning a method that often works isn't guaranteed to work. It's not an algorithm in the usual sense that we talk about algorithms. But rather, it's an attempt. For example, trig substitution, such as:

$$\begin{aligned} &f(\sin x, \cos x, \tan x, \cot x, \sec x, \csc x) = \\ &g_1(\sin x, \cos x) = g_2(\tan x, \csc x) = g_3(\cot x, \sec x) \end{aligned} \tag{2.7}$$

And there is transformation from a trigonometric form into a polynomial form, such as:

$$\int f(\tan x) dx = \int \frac{f(y)}{1+y^2} dy \tag{2.8}$$

And the knee-jerk reaction of $(1-x^2)$ is to make a transformation that involves $x = \sin y$ because we know $\sin^2 x + \cos^2 x = 1$. Similarly, when encountering $(1+x^2)$, we make a transformation that involves $x = \tan y$ because we know $1 + \tan^2 x = 1/\cos^2 x$. These two transformations can get trigonometric form from the polynomial form. So now we can make progress on the integration problem. Let $x = siny$:

$$\mathcal{I} = -5 \int \frac{\sin^4 y}{(1-\sin^2 y)^{5/2}} d(siny) = -5 \int \frac{\sin^4 y}{\cos^4 y} dy \tag{2.9}$$

And $\sin^4 y / \cos^4 y = \tan^4 y = 1/\cot^4 y$. After measuring the depth of the functional composition, $\tan^4 y$ is the winner. Now we apply Equation 2.8 to the integration problem. Let $x = \tan y$:

$$\mathcal{I} = -5 \int \tan^4 y dy = -5 \int \frac{x^4}{1+x^2} dx \tag{2.10}$$

Now, $x^4/(1+x^2)$ is a rational function and the degree of the numerator is greater than the degree of the denominator, so we can just divide it out.

$$\mathcal{I} = -5 \int \frac{x^2(1+x^2) - (1+x^2) + 1}{1+x^2} dx = -5 \int (x^2 - 1 + \frac{1}{1+x^2}) dx \tag{2.11}$$

Then we deal with different components, i.e. x^2, -1, and $1/(1+x^2)$:

$$\mathcal{I} = -5\int x^2 dx + 5\int dx - 5\int \frac{1}{1+x^2}dx \tag{2.12}$$

$$= -\frac{5}{3}x^3 + 5x - 5\int \frac{1}{1+x^2}dx \tag{2.13}$$

Let the third integral component be $\mathcal{I}_3$ and $x = \tan y$. Since $1+\tan^2 y = 1/\cos^2 y$ and $d(\tan y) = 1/\cos^2 y$:

$$\mathcal{I}_3 = -5\int \frac{1}{1+x^2}dx = -5\int \frac{1}{1+\tan^2 y}d(\tan y) = -5\int dy = -5y \tag{2.14}$$

Therefore, the result of the integration problem is as follows, where $C \in \mathbb{R}$ is a real constant:

$$\mathcal{I} = \int \frac{-5x^4}{(1-x^2)^{5/2}}dx = -\frac{5}{3}x^3 + C \tag{2.15}$$

2.2 Goal Tree

The whole problem solving process can be represented as a **problem reduction tree**, or an "**and/or tree**", because for each node (current state), we have one and/or more following choices. It can also be called a "**goal tree**", because this tree shows how our goals are related to one another.

How well does the program performed? It was given 56 of the hardest problems, and it got 54 right. What happened when it didn't get the other two? It was not because of running out of memory (only 32 KB). In fact, it just was lacking 2 transformations that were needed in order to solve the whole entire set of final quiz problems.

What was the depth of the tree in the maximal case? The answer was 7. So in the worst case, this program has to get down seven levels. And what was the average depth? The answer was approximately 3.

So now we're going to say something about the nature of the domain. In the domain of calculus problems, integrals expression that are given to freshman, the average depth of problem reduction needed to solve the problem was 3. So that's not very complicated.

Another question of even greater interest is, how many branches were unused? For example, in the above integration problem, the $\tan^4 y$ branch is used while the $1/\cot^4 y$ branch is not used. The number of unused branches is about 1, which means this tree keeps itself together, and doesn't run down to a very large, bushy, useless tree. So this means that the depth of functional composition, which someone may suggest as a technique for recognizing the right problem work on, was a choice that didn't actually matter. Because the tree doesn't grow deep not go broad.

2.3 Knowledge

One of those things as a catechism having to do with **knowledge**.

What kind of knowledge is involved in solving these integral problems? Well, knowledge about transformation. Knowledge about how goal trees work and when we're done with a problem. Knowledge about what things don't need to be transformed, because you can look them up in a table. That's the kind of knowledge that is involved in doing MIT 18.01.

And if you do 18.0 circuit theory, 6.0 circuit theory or 6.0 Maxwell's equations, this is the same thing. You have to ask questions of this sort, about the nature of the knowledge involved. And **question number one** is always **what kind of knowledge is involved?** Is it Kirchhoff's laws, Maxwell's equations, what is it?

The **next question** is, **how is the knowledge represented?** All this stuff, ultimately was represented in list, best expressions. Some of the knowledge was recorded in a table of best expressions to show what transformations there are. There was a similar table of integrals. Knowledge about goal trees was embedded in the procedure. So it was procedurally represented. So for each of the categories of knowledge, there's a way it gets represented.

How is it used? Straightforward. Transformations are used to make the problem simpler. The table is used to trim off and to serve as the bottom of the tree. Those are the ways in which the knowledge is used.

And then there's the question of **how much knowledge is required**. For the integral solving program, the following knowledge is required: 1) the table of integrals containing only 26 elements; 2) about 12 safe transformations and about 12 heuristic ones. That was surprisingly few.

Another surprise of a similar kind, also about knowledge, is that the relationship between the method to be used, and the characteristics of the problem, was almost a diagonal table. That means you could, in this domain, make the right transformation almost all the time if you're a little bit smart, and never back up.

That was an observation made by Joel Moses, who became subsequently a provost at MIT for a while. And he wrote a program that could solve anything. It would beat the most dedicated mathematicians at integration. And its descendants are in MATLAB today.

The catechism is the kind of stuff you should ask any time you're dealing with a new domain. It will make you smarter. This is meta knowledge, this is knowledge about knowledge. So the tired aphorism "knowledge is power" should be "knowledge about knowledge is power", because meta knowledge is where the real power is.

Now there's one final thing that this program does for us. It tells us something about our appreciation of what it means to be intelligent. Back

to the question, whether a program that could do symbolic integration would be, in any way, or should be considered to any degree, intelligent.

Even in these days of MATLAB, and whatnot, many of you said, yes. I learned how to do that at MIT, or late in high school, so I must be smart. But now that we've completed this discussion, I also expect that your feeling of intelligence in this program is somewhat diminished. Because what happens is that, **when we understand how something works, it's intelligence seems to vanish**. You've seen this in your friends, right? They solve some problems, they seem super smart. Then they tell you how they did it, and they don't seem so smart anymore.

So let's conclude our discussion today with a little story. A long time ago, I was talking with a student who said, computers cannot be intelligent. And I said, OK, maybe you're right, but let me show you this program. So I showed him the integration program. And after I showed him a couple of those examples, he said, well, all right, I guess maybe they can be intelligent. Learning how to do that is not always easy. Then I made a fatal mistake. I said let me show you how it works, and we spent an hour going through it. And at the end of that time, he turned to me and said, I take it back, it's not intelligent after all. It does integration the same way I do.

3

Reasoning: Goal Trees and Rule-Based Expert Systems

https://www.youtube.com/watch?v=leXa7EKUPFk

3.1 The Complexity of The Problem

The example program looks like it's kind of smart, and it somehow can answer questions about its own behavior.

We can draw the trace of the program as it works on the problem. So we can answer **"why" questions** by backtrace (**bottom-up**) the problem-solving route (in the goal tree). And how about **"how" questions**? How would it go about answering a question about how you did something? We can answer "how" questions by going down (**top-down**) in the action trace of the program so as to see how things are put together.

When you got a goal tree, you can answer certain kinds of questions about your own behavior.

Imagine that you're looking at the path of a ant on a beach. The path of the ant looks extremely complicated. And you're tempted to think the ant is some kind of genius or monster brain ant. But in fact, when you take a closer look, what you discover is that there are a bunch of pebbles on the beach, and all the ant is doing is avoiding those pebbles on his way home. So in this case, the complexity of the behavior is a consequence of the complexity of the environment, not the complexity of the program. This is the metaphoric **Simon's ant** (Herbert A. Simon): **the complexity of the behavior is the max of the complexity of the program and the complexity of the environment**.

$$C(\text{behavior}) = \max(C(\text{program}), C(\text{environment})) \tag{3.1}$$

3.2 Rule-Based Expert Systems

The rule-based expert systems were developed in a burst of enthusiasm about the prospects for commercial applications of artificial intelligence in the mid-1980s. At that time, it was supposed that you could account for useful aspects of human intelligence by writing all the knowledge in the form of simple rules. So if this is true, then that's true; if you want to achieve this, then do that. But all the knowledge had to be encapsulated in the form of simple rules.

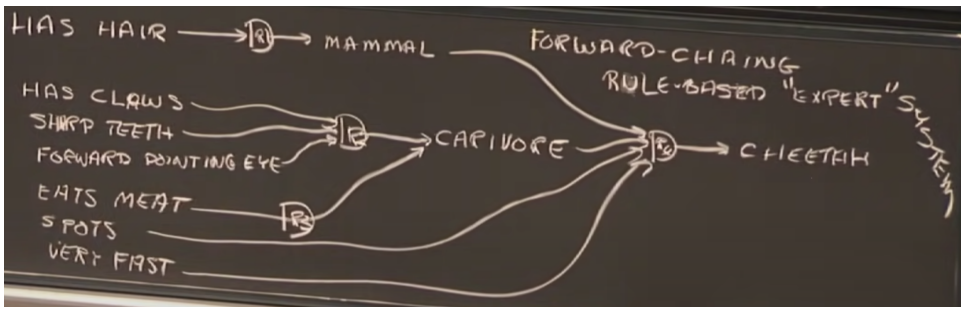

Figure 3.1: Example of the forward-chaining rule-based expert system.

Figure 3.1 shows an example of the **forward-chaining rule-based expert system**, which works forward from the facts we're given to the conclusion off on the right. For marketing reasons, they are called "expert" systems instead of novice systems. But are they really experts in a human sense? Not really, because they have these knee-jerk rules. They're not equipped with anything you might want to call **common sense**. They don't have the ability to deal with previous cases, like we do when we go to medical school. So they really ought to be called rule-based novice systems because they reason like novices on the basis of rules.

Can this system answer questions about its own behavior? Yes, because it is, in fact, a **goal tree**. Each of these rules are require several things to be true is creating an "and" node or "or" node. For example, if we ask the system that why were you interested in the animal's claws? It can answer that because I was trying to see if it was a carnivore. How did you know that the animal is a mammal? Because it has hair. Why did you think it was a cheetah? Because it's a mammal, carnivore, has spots, and very fast.

So by working forward and backward in this goal tree, this too can answer questions about its own behavior. So now you know how, going forward, you can write programs that answer questions about their behavior. Either you write the subroutines so that each one is wrapped around a goal, so you've got **goal-centered programming**, or you build a so-called **expert system** using **rules**, in which case it's easy to make it leave behind a trace of a goal tree, which makes it possible to answer questions about its own

behavior, just as this zoo analysis program did.

Instead of working forward from facts, we can also work backward from a hypothesis. For example, if it's going to be a cheetah, then it must be the case that it's carnivore and mammal, and it must be case that it has spots and is very fast. This is called the **backward-chaining rule-based expert system**.

Overall, the whole system (forward or backward) is a **deduction system**. That's because it's working with facts to produce new facts. When you have a deduction system, you can never take anything away. But in fact world, in deduction world, you're talking about proving things. And once you prove something is true, it can't be false. Otherwise, you've got a contradiction in your system. But if you think of using rules as a programming language, then you can think of arranging it so that these rules add or subtract from the database.

3.3 Three Heuristic Principles of Knowledge Engineering

One tendency of MIT students, of course, is that we all tend to **generalize**. In order to really make progress on tasks, you have to know about some principles of **knowledge engineering**. Principle number one (or heuristic number one) is to **deal with specific cases**. By looking at specific cases, you elicit knowledge from people they otherwise would not thought to give you. Principle number two is to **ask questions about things that appear to be the same, but are actually handled differently**. There's something you can't tell the difference, but the experts are handling these objects differently. If you ask the experts why these things are different, they can give you specific nuances. Now you've got some new words in your vocabulary. And those new vocabulary words are going to give you power over the domain because you can now use those words in your own rules. And you can write rules to distinguish different objects in that domain. Principle number three is to **build a system and see when it cracks**. And when it cracks is when you don't have one of the rules you need in order to get the program to execute as you want.

3.4 Lecture Conclusion

The bottom line is that if you build a rule-based expert system, it can **answer questions about its own behavior**. If you build a program that's centered on goals, it can answer questions about its own behavior. If you build an integration program, it can also answer questions about its own behavior. And if you want to build one of these systems, you need to extract knowledge from an expert. You need to approach it with these kinds of

heuristics, because the experts won't think what to tell you unless you elicit that information by specific cases, by asking questions about differences, and by ultimately doing some experiments to see where your programs crack.

Is this all we need to know about human intelligence? Are these things really smart? The traditional answer is no, they're not really smart, because **their intelligence is this sort of thin veneer**. And when you try to get underneath it, as written, they tend to crack. For example, we could talk about a rule that knows that you should put the potato chips on the top of the grocery bag. But a program that knows that would have no idea why you would want to put the potato chips on the top of the bag. They wouldn't know that if you put them on the bottom of the bag, they'll get crushed. And it wouldn't know that if they get crushed, the customer will get angry, because people normally don't like to eat crushed potato chips. So that's why the knowledge of these things tends to be a veneer.

Do rules have anything to do with **common sense**? The professor is becoming a little bit agnostic on this subject. There are certain indications and situations, in which rules could be said to play a role in our ordinary understanding of things.

When we tell a story, it's mostly a matter of controlled hallucination. I know what rules are in your head, so I could take advantage of that in telling the story and not have to tell you anything I'm sure you're going to know. And that's why that storytelling is largely a matter of just controlling how you're going along, a kind of controlled hallucination.

Bibliography

[1] Alan M Turing. Computing machinery and intelligence. In *Parsing the turing test*, pages 23–65. Springer, 1950.

[2] Marvin Minsky. Steps toward artificial intelligence. *Proceedings of the IRE*, 49(1):8–30, 1961.