From Human to Machine: the Limit of Computation

Note: This passage is one of my blog posts. Since it is supposed to be a popular science article on computation and Turing Machine, I didn't follow some of the conventions like set of states ***Q****, alphabet* ***Σ****, etc. Hope that won't bother you.*

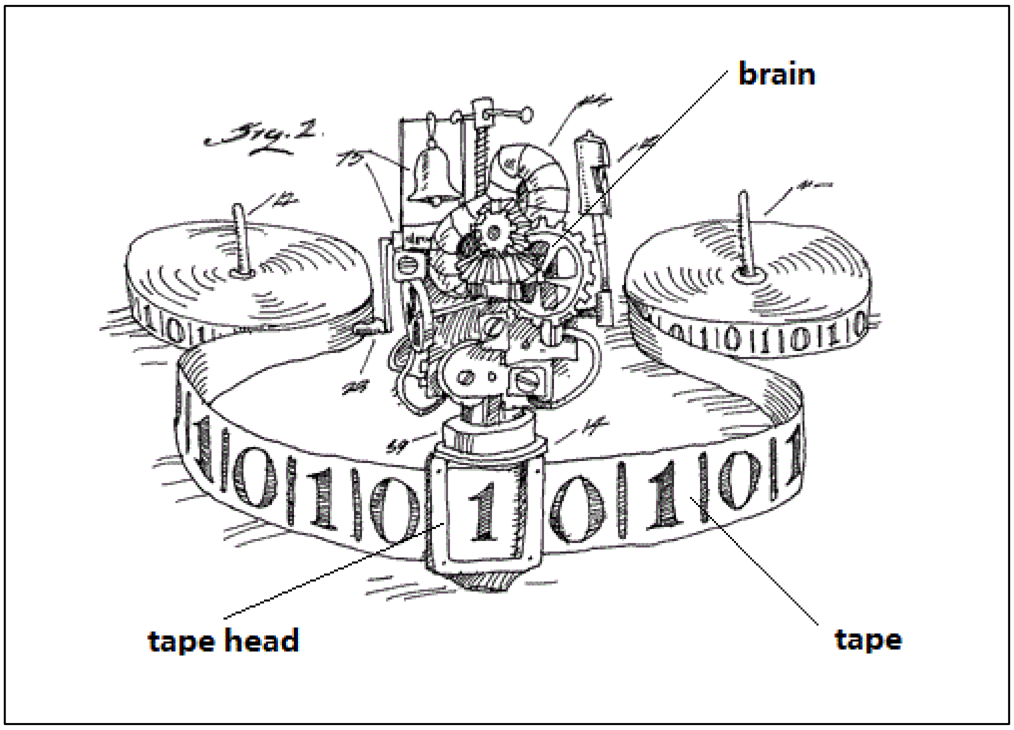
Suppose you are asked a question of ***1+1***, you must be able to answer "***2***" without hesitation. Bingo! Maybe it's too easy for a smart guy like you. Then what if you are asked ***1234567+2345678***? This time it might be difficult to give the correct answer instantaneously. But that doesn't matter. Given a pen and some paper, you can figure it out in less than one minute. But what if you are asked to add 2000 seven-digit numbers up instead of just two?

You probably have to add the first number to the second number, take the result and add it to the third number, doing this on and on till all numbers get added. Each time you get one number added. Since there are 2000 numbers, you have to do 2000 additions. However, for each addition out the 2000, you actually get two seven-digit numbers. This means you have to add ones first, then tens, hundreds, and so on. So you end up with seven additions for every two numbers. Consequently, there are ***2000*7=14000*** additions in total! That is huge amounts of calculation. Note that we did not take carries into consideration, which already made the problem simpler. What is worse, it's very common for modern science to perform calculations like thousands of additions. OMG! Are we supposed to compute till death?

Although it's theoretically feasible to do manual computation, the amount of computation is definitely beyond the capability of human beings. Luckily, we humans are animals that can "manufacture and use tools". Is it possible manufacture one that can help us to handle these computations? Hmm, this is certainly a good idea. But before we draft and buy materials to make the real machine, we have to pin down the underlying theory. Fortunately this work has been done. In the 1930s, several mathematicians from all over world conducted the earliest research on different theoretical computing systems. Alonzo Church invented lambda calculus, Kurt Gödel got recursive functions, Alan Turing ended up with his famous Turing Machine, and so on. Amazingly, although these systems are different from each other, they are at last proved to have the same computing power. They are like different explorers setting off together from Europe. One goes east, another goes west, some go by sea, etc. But they finally arrived at the same new continent.

What is a Turing Machine like?

Since all these systems share the same power, let's just take the most popular one as an example, namely the Turing Machine. To be simple and concrete, we can think of a TM (short for Turing Machine) as the picture below. There exists a perfect analogy between this TM and us humans. The TM has a long tape made of cells, which is similar to scrap paper for humans. It also has a tape head, which is similar to our eyes and hands. The head can read information from the tape like our eyes do, or it can take down notes as our hands do. Besides it can change its position by moving left or right. The last and also the most important part of the TM is its brain.



Why do humans know how to compute? It's because we know the rules for computation. Let's take addition as an example. To calculate "***1+2***", we directly apply our rules and get the answer ***3***. To calculate "***12+34***", according to the rules we need to compute the ones first, which is ***2+4=6***, and then the tens, which is ***1+3=4***. Finally we concatenate the two parts to get the answer ***46***. Every addition traces back to our rules. So obviously we have to teach these rules to a TM if we want it to do additions. These rules (almost) constitute the brain of a TM, namely transition table (or more formally, transition function). Intuitively, transition table contains the rules about what should we do each time we come across something from the tape.

It turns out that rules alone are not enough. Human brains apply arithmetic rules with the help of memory. Imagine a teacher is orally telling an addition problem ***1+1*** to you. When he finishes reading the first “one”, you will do nothing but waiting for him to continue. But when he finishes the second “one”, you will start calculation at once. Why do you react differently to the same pronunciation? It’s because you know in your memory that you are now in a different state. This “one” is the last number rather than the first number. The same reasoning applies to a TM. A TM also has to remember states like “finished reading first number”, “finished reading both numbers” and “a carry produced”. Having known the concept of states, we can now refine our explanation of the transition table. Formally, the transition table is composed of entries that are of the following form:

$$(p, a) \rightarrow (q, b, RIGHT)$$

This expression represents a computing rule: when the TM is at state p and tape head reads a character a (the left part to the arrow), do the following three things -- transfer to state q, modify the original a to b, and move the tape head leftward (the right part to the arrow).

A Turing Machine calculating example

Now we know what is a TM, we can turn to real TM's. We will design an addition TM together, who will accompany us throughout this passage.

The first problem we need to handle is language. An expression may have different notations in different languages. For example ,in English we may say “one plus two”, in Chinese we may say “一加二”, while in mathematical language we prefer “1+2”. Even human brains can't handle languages that we don't know, not to mention TM's. So we need to design a language for our addition TM. To make it as simple as possible, let's suppose our TM is similar to ancient people who count with marks. Our TM counts with ⊿. Therefore decimal ***1*** in our TM language is ⊿, ***2***=⊿⊿, ***3***=⊿⊿⊿, etc. Since our simple TM is designed specifically for additions, we don't need an adding sign like “+”. Yet we still need a delimiter since we have two numbers for each addition. Let's use | for this delimiter and append it to every number. As a result, ***1+2*** translated to our TM language is ⊿|⊿⊿|. Similarly, ***6+5*** is ⊿⊿⊿⊿⊿⊿|⊿⊿⊿⊿⊿|. If our TM genuinely knows how to add numbers, it should be able to give the answer ⊿⊿⊿ when fed with ⊿|⊿⊿|, which means ***1+2***. Now let's see what exactly our TM is like under this primitive language.

This crude TM has six states. They are ***{start, num1, num2, bar1, bar2, done}***, among which ***start*** is the initial state and ***done*** means the calculation is finished. The transition table has the following seven rules, each commented with the intuition of the corresponding transfer:

(start, ⊿) → (num1, ⊿ , RIGHT)   // begin to read first number
(num1, ⊿) → (num1, ⊿ , RIGHT)   // reading first number
(num1, |) → (bar1, ⊿ , RIGHT)   // meet first bar, update it to ⊿
(bar1, ⊿) → (num2, ⊿ , RIGHT)   // begin to read second number
(num2, ⊿) → (num2, ⊿ , RIGHT)   // reading second number
(num2, |) → (bar2, |, LEFT)   // meet second bar, move back left
(bar2, ⊿) → (done, |, LEFT)   // update last ⊿ to |, calculation finished

As said above, the tape for ***1+2*** should be ⊿ |⊿ ⊿ |. Now let's put the tape head at the first character, which is "⊿ ". Then we turn on our TM and let it run till the calculation is done. Can you see the results? Our TM ends up with a tape of ⊿ ⊿ ⊿ ||. If we get rid of the two trailing bars, it is exactly the answer we want! If you hold no tolerance for the two trailing bars, just introduce a new blank character (like an eraser) and add several rules to clean up. Whichever style you favor, we now have a rather simple but working TM for additions. I should probably clarify that this TM actually only works for positive integers. But that's no big deal. The key is it indeed can add.

Some may argue that this TM is merely merging two bunches of ⊿ 's. It's not addition at all. As a response let's imagine a mother giving her child two bunches of chocolate bars. The first bunch contains eight bars and the second contains nine. The mother then asks the child how many in total he gets. The child may answer by simply merging the two bunches together and saying "That's it!" But we will never count that as computation. Only when he answers 17, will we consider him to know the correct answer. The same goes for our TM. It does more than merging because it explicitly gives an answer in its own △ language. And that answer is correct; it's neither ⊿ ⊿ nor ⊿ ⊿ ⊿ ⊿ for input ⊿ |⊿ ⊿ |, but the correct answer ⊿ ⊿ ⊿ .

If you are still puzzled about the adding power of our TM, this link provides an example for a multiplication TM. Since multiplication is much more involved, hopefully it will convince you that our TM genuinely knows how to compute.

Beyond addition: the Universal Turing Machine

We've got an addition TM. Yet we might not be satisfied because we still need to design subtraction TM's, multiplication TM's, and so on. We will need millions of different TM's for different functionalities. But for human beings, you can teach both addition and subtraction to a single person. He would be able to handle all cases as long as he can learn them. If only a single TM was also that versatile!

This turns out to be totally possible. The Universal Turing Machine (UTM) is designed specifically for this purpose. It can simulate whatever other TM's you teach it. Clearly the key is how to teach.

We can tell from our previous analysis that, TM's are essentially transition tables because all other elements (e.g. states) can be derived from the table. So knowing a transition table is equivalent to knowing a TM. According to this assertion, if we can teach the transition table of our addition TM to the UTM in a proper way, hopefully the UTM would be able to perform additions.

Once again we have to design a language to communicate with the UTM. There are multiple approaches to do this, but the underlying ideas are similar. Here I'm going to introduce a simple way. Our UTM language will contain only two characters, ***0*** and ***1***.

Recall that every entry of the transition table is of the form ***(p, a) –> (q, b, LEFT)***. The five elements in this expression fall into three categories: states of TM (both current and next), characters of tape (both original and updated), and directions to move. The first thing we're going to do is numbering them. As an example, our addition TM is numbered as follows:

> ***{start, num1, num2, …, done} → {state1, state2, state3, …, state6}***
> ***{⊿ , |} → {character1, character2}***
> ***{LEFT, RIGHT} → {direction1, direction2}***

As a result, ***1+2*** is no longer represented as ***⊿*** |***⊿ ⊿*** |. It's:

character1 character2 character1 character1 character2

With the help of this numbering scheme, we can now represent transition rules in a relatively universal way. For example the fifth rule above:

(num2, ⊿) --> (num2, ⊿ , RIGHT)

is represented as:

$$\textit{(state3, character1)} \rightarrow \textit{(state3, character1, direction2)}$$

Since the five elements in an expression always stay in their fixed positions respectively, we can infer the type of an element even without the English words. This brings a further simplification:

$$(3, 1) \rightarrow (3, 1, 2)$$

Next is the last step, translating the expression above to the UTM language:

$$\overbrace{000}^{3\text{ zeros}}1\ \overbrace{0}^{1\text{ zero}}\ 1\ \overbrace{000}^{3\text{ zeros}}1\ \overbrace{0}^{1\text{ zero}}\ 1\ \overbrace{00}^{2\text{ zeros}}$$

All ***1***'s here are delimiters, functioning just like commas in English or | in our addition TM language. Counts of ***0***'s correspond to the numbers in the previous expression. Thus we manage to represent a transition rule in our UTM language.

We can generalize this methodology to all transition rules for any TM so that every rule can be numbered like:

$$(i, j) \rightarrow (k, l, m)$$

then converted to:

$$\overbrace{0\cdots0}^{i\text{ zeros}}1\overbrace{0\cdots0}^{j\text{ zeros}}1\overbrace{0\cdots0}^{k\text{ zeros}}1\overbrace{0\cdots0}^{l\text{ zeros}}1\overbrace{0\cdots0}^{m\text{ zeros}}$$

We've known how to represent a single rule. To incorporate the whole table on a long tape, we just concatenate all the rules. Note we need one more delimiter to separate the rules. Let's use ***11*** this time. So the whole table looks like:

$$rule_1 11 rule_2 11 \cdots$$

We are done! The entire transition table has been translated into UTM language. In other words, we've successfully told the structure of our original addition TM to the UTM in a proper way. The next step is to ensure that the UTM knows how to learn it, namely to design states and transition table for the UTM. This step is a little bit complicated so we are going to skip it. If you are interested, follow this link to see a detailed (but different) example of a UTM in Scheme.

So far we've got a very powerful TM, namely the UTM. If we want it to calculate ***5+3***, we can feed to it the transition table of an addition TM along with ***5*** and ***3***. If we would like it to calculate ***5*3***, just replace addition with multiplication. In fact, this UTM is able to do whatever a modern computing device can do. Even if you want it to make an omelette for you, all you need is to tell it in the UTM language the exact steps to make an omelette plus offer it required materials. Then you can enjoy your TV programs while waiting for the omelette. That is literally cool. It feels like “impossible is nothing”, right?

The limit of computation: what we can't do

Unfortunately, it is wrong. Although our UTM is very powerful, there do exist problems that it cannot solve. Some of the problems even appear easy since they have only two possible answers – yes and no.

Why? To be as simple as possible, recall that TM's are essentially transition tables. Following the method above, we can denote a transition table as a ***0/1*** string. So TM's are equivalent to (at least some) ***0/1*** strings. Then what about the problems we ask TM's to solve? They are also ***0/1*** strings in UTM language. Since both are binary strings, can we write down a TM on a tape as input, and then feed it to another TM? Of course, we've actually done that in the UTM example.

However, if we feed a TM to this TM itself, things may change. It is a very similar situation to the famous liar paradox:

This statement is false.

If we assume the sentence above is true, then according its meaning, the sentence itself should be false. But that contradicts our assumption. However, if we assume it is false, we also end up in contradiction. As a result, the statement is neither true nor false.

The TM may face a similar dilemma when referencing itself as the liar paradox does. To prove it, we need to first convert TM's to numbers (like translating TM to 0/1 strings), and then apply a special technique called diagnolization, which was initially published by Cantor. The exact proof is very complicated, but the conclusion instead is simple, elegant, yet somewhat frustrating: there exist some problems that TM's can never solve.

I mentioned in the beginning of this passage that initially mathematicians designed multiple computing systems, but all were proved to be equally powerful at last. Consequently people later came up with a conjecture, that the TM model has precisely described the computing capability of human beings. Anything that we humans can compute is also computable for these computation models. Anything that they can't compute is also beyond our ability, just like the liar paradox. This conjecture is called the Church-Turing thesis, named after Alonzo Church and Alan Turing. 80 years have passed since 1930s, yet no one has found a more powerful computing model thus invalidates the Church-Turing thesis. God says in the bible "the wisdom of the wise will perish, the intelligence of the intelligent will vanish." Could it be that this is already the limit that God set for human intelligence on computing?

References

1. https://class.coursera.org/automata-003
2. http://en.wikipedia.org/wiki/Tool_use_by_animals
3. http://3.bp.blogspot.com/-O8pU1eLknqg/Uf_2LlOsGWI/AAAAAAAAHoU/COz3NOtX-Q8/s1600/turingmachine.gif
4. http://en.wikipedia.org/wiki/Computable_function
5. http://plato.stanford.edu/entries/computability/
6. https://www.biblegateway.com/passage/?search=Isaiah+29&version=NIV