Lecture 41:
All Computers Are Created Equal

Aims:

- To discuss Universal Turing Machines;
- To discuss Church's thesis, and
- To discuss the Sequential Computation Thesis.

41.1 Universal Turing Machines

- Computers are programmable but Turing machines, as we have described them, are not.
- Each of our Turing machines can be viewed as a computer with a single fixed program.
- We have to design a particular Turing machine for each problem, whereas real computers are 'general-purpose'.
- But this is misleading because, in fact, even Turing machines are programmable.
- The idea is
  - just like we found a way to encode a Turing machine as integers which could be the input to a Counter Program that simulates the behaviour of the Turing machine
  - we can encode a Turing machine as a string that can be placed onto the tape of another 'general-purpose' Turing machine that then simulates the behaviour of the original Turing machine.
- Such a general-purpose Turing machine is called a *Universal Turing Machine*.
- The main challenge is to encode the transition table.
- Using only one symbol (e.g. '1'), we set up the following correspondences:

h	:	1	L	:	1
q_0	:	11	R	:	11
q_1	:	111	$\sqcup$	:	111
q_2	:	1111	a	:	1111
q_3	:	11111	b	:	11111
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

- Then a transition such as $\delta(q_1, a) = \langle L, q_2 \rangle$ becomes a 4-tuple:

$$*111*1111*1*1111*$$

- We'll have something like this for each entry in the transition table.
- The Universal Turing Machine that we will describe has three tapes (but, as we have discussed, multi-tape machines can be simulated by single-tape machines):
  1. a tape that contains a string that represents a Turing machine TM (using the encoding from above);
  2. a tape that contains a string that represents the input to TM (also encoded using the correspondences from above); and
  3. a tape that can be used to keep track of the current state that TM would be in (initially containing q_0, i.e. 11).
- The Universal Turing Machine knows
  - the state that TM would be in (tape 3), and
  - the symbol that TM would be scanning (tape 2).
- At each step, the Universal Turing Machine
  - searches tape 1, to find the 4-tuple that represents the transition for this state and symbol;
  - it reads the 3rd part of the 4-tuple from tape 1:
    - if 1 (or 11), it moves tape 2's head left (or right) a suitable number of symbols;
    - else, it writes onto tape 2 (maybe shifting to make room first);
  - then it reads the 4th part of the 4-tuple from tape 1 (the new state) and overwrites tape 3 with this.

41.2 Church's Thesis

41.2.1 The Evidence

- We have seen that Turing machines can be designed to solve quite complicated problems.
- We have also seen that there are some problems that Turing machines cannot solve.
- We have also seen that certain extensions to Turing machines do not increase the power (no additional problems become solvable).
- This suggests that Turing machines represent a natural upper limit on what a computing machine can be designed to do.
- We have also seen that there are other approaches to computation, e.g. Counter Programs (and their variants).
- But, we have seen that these too are of equivalent power to Turing machines (i.e. the set of problems that Turing machines and Counter Program can solve are the same).
- Over the last 60+ years, numerous other formal models of computation have been proposed. There have been models based on string processing, models based on stored programs, models based on the composition of basic functions, etc. E.g.:
  - Unrestricted grammars
  - μ-recursive functions

  - Markov algorithms
  - Post machines
  - Random access machines
  - λ-calculus
- None of these models is more powerful than Turing machines. The best that has been achieved is equivalent power.
- This adds support to the idea that we have reached a natural upper limit on what a computing machine can be designed to do.
- Of course, we only consider extensions and alternative models that are *in the same spirit* as Turing machines.
- Let's call such models *'reasonable'* models of computation. What do we mean by 'reasonable'? That's hard to say, but perhaps it's something along these lines...
  - There is a finite set of operations.
  - Each operation is simple, precise and mechanical (e.g. not requiring the use of 'insight').
  - Each operation must be of a type which could be carried out with a finite amount of effort at each step. E.g. a machine which could answer an infinite set of questions in one step would not be reasonable.
  - The machines (or programs) must be finite.
  - The machines (or programs) must be uniform. This means that, for a particular problem, one machine (or program) is used for inputs of all sizes. We don't have a different machine (or program) for different input sizes. E.g. we don't have a sequence of machines (or programs) where the member of the sequence used to deal with inputs of size 100, say, need not be the one used to solve inputs of size 101.
  - Each machine (or program) will, if operated correctly, produce its result in a finite number of steps.

41.2.2 The Thesis

- All these equivalences between such a diversity of (reasonable) models lead to the idea that we have found some quite profound, intuitive concept that has many different-looking but equivalent precise definitions. This belief is encapsulated in Church's Thesis, also known as Turing's Thesis (especially when we state it in terms of Turing machines), and sometimes called the Church-Turing Thesis.
- There are many ways of stating this thesis and we choose to give two informal ways.

  Church's Thesis: *Any problem that is computable by some 'reasonable' model of computation is computable by a Turing machine.*

  Church's Thesis: *Nothing will be considered an algorithm if it cannot be rendered as a Turing machine.*
- This is a thesis, not a theorem. If it were a theorem, we would have a proof. But a proof of this is not possible. It equates informal concepts ('reasonable models of computation', 'algorithms') with a formal one (Turing machines).
- It is possible it could be overthrown if someone ever proposed an alternative model of computation that people agreed was 'reasonable' and yet was provably capable of solving problems that cannot be solved by any Turing machine. Faced with the evidence of the equivalences mentioned earlier, no-one considers this likely.

41.2.3 Ill-Conceived Attacks

1. Every now and again someone proposes a model of computation that they claim is capable of solving problems that Turing machines cannot solve. It might be massively parallel computers, hypercomputers, quantum computers, molecular computers, ...

   Usually they are wrong. E.g. sober writings on quantum computers usually say that there are no problems that quantum computers can solve that Turing machines cannot solve.

   Sometimes, however, they are right! Their model does solve problems that Turing machines cannot, hypercomputers being an example. So why doesn't this overthrow Church's thesis? The reason is that, in each case where the model can solve problems that Turing machines cannot, the model has always been found to be 'unreasonable'. "Failure to appreciate this subtlety has resulted in 'models which disprove the Church-Turing hypothesis' being announced in some of the (fringe or popular) literature. In all such cases the error is usually found to be that of comparing a non-uniform model with a uniform one [i.e. Turing machines]." [Dun91], p.138 Or they are comparing a model that allows an infinite amount of labour in each step (e.g. hypercomputers) with one that does not (i.e. Turing machines).
2. Sometimes people claim to have found ways of using Turing machines to solve problems that have been proven to be non-computable. So they think they too have overthrown Church's thesis in some way. All they have actually ever done is solve special cases of the problem. No-one ever said that special cases were unsolvable. For example, in our lecture on the Halting Problem, we carefully noted that, just because the Halting Problem is non-computable *in general*, does not mean that you can't write a program that goes some way to solving it.

41.2.4 Consequences

- A problem that is not computable by a Turing machine is not computable by any algorithm. Given an unlimited amount of time and memory, all general-purpose reasonable models of computation can solve precisely the same problems. The non-computable problems are not solvable on any of them; the computable problems are solvable on all of them.

  This is why it didn't matter what programming language we used in the proof of the non-computability of the Halting Problem. Turing used Turing machines in his original proof. We used MQCQCA programs. But we could have used any fully general model of computation or fully general programming language.
- When you want to prove whether a certain problem is computable, you can use whichever of the equivalent models suits you best (whichever gives the easiest proof), knowing that, by Church's thesis, your result will apply to all of the equivalent models of computation.
- No programming language is more powerful than Turing machines; most are of the same power as Turing machines. They differ only in such things as programmer convenience and efficiency.

  There are specialised languages, however, that are of lower power. E.g. SQL is just a database query language and is of lower power than Turing machines.

41.3 The Sequential Computation Thesis

41.3.1 Background

- A primitive formal model of computation, such as Turing machines, is also essential for obtaining robust results in Complexity Theory.

- There are advantages to studying the running time and memory requirements of Turing machines:
  - The time a Turing machine takes is unambiguously obtained by counting the number of transitions it makes. We can ignore actual timings and treat each transition as something that can be executed in one time unit. This seems quite sensible — much more sensible than the assumptions we were making in our lectures on Complexity (e.g. that multiplication should cost the same as addition; that a complex assignment should cost the same as a simple one; that an assignment should cost the same as a return command; etc.)
  - The memory requirements of a Turing machine can also be defined unambiguously. We simply count the number of cells the Turing machine ever visits during a computation.
  - A Turing machine will operate in the same way on all sizes of inputs. E.g. it uses the same transitions to multiply two large numbers as it does to multiply two small numbers. This isn't true of real hardware. Multiplying small numbers that fit into a single word is different from multiplying two much larger integers that exceed the capacity of a single word.
- Of course, time and space complexities *are sensitive* to what kind of Turing machine you are using. E.g.:
  - On a one-tape machine, the problem of copying a string is at least quadratic in the length of the string.
  - But on a two-tape machine, there is a linear solution.
- While the time and space complexities of problems are sensitive to the kind of machine under consideration, the nice thing about *classes* of problems, such as **P**, is how *insensitive* they are to the kind of machine.
- In particular, problems that take polynomial-time on a one-tape machine also take polynomial time on all *sequential* models of computation (whether they be variations of Turing machines or other models of computation).
- So, for example, **P** will contain the same problems irrespective of the model of computation (provided it is sequential).
- What do we mean by *'sequential"*?
- It is easier to say what we don't mean by by sequential than what we do mean.
  - Any model that allows *unlimited* amounts of concurrency is not sequential.
  - Non-deterministic Turing machines and ND-DECAFF are not sequential: they use 'magic dice' to do the equivalent of *unlimited* amounts of work in a single step.

41.3.2 The Thesis

- This leads to the Sequential Computation Thesis, for which we give two informal versions. Note again that it is a thesis, not a theorem.

  The Sequential Computation Thesis: *All sequential models of computation have polynomially related time behaviour.*

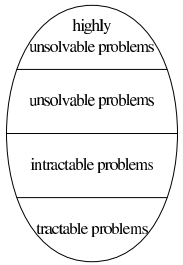
  The Sequential Computation Thesis: *The class of problems solvable in polynomial time is the same for all sequential models of computation.*
- I am simplifying a bit to make things easier to understand, but not in any drastic way. There are some sequential models of computation whose actions are exponentially weaker than Turing machine transitions and only if we strengthen these models does the thesis still hold. But this is a nicety that you can ignore.

41.3.3 Consequences

- Turing machines may strike you as very inefficient. For even the simplest problems, they do large amounts of scanning, shifting, etc. However, the point is that they are only polynomially less efficient than even the fastest sequential models of computation. E.g. a Turing machine might take twice as long, or a thousand times as long, or even that amount squared or cubed, but it will not take exponentially more than the fastest models.
- When you want to prove whether a certain problem is tractable, you can use whichever of the sequential models suits you best (whichever gives the easiest proof), knowing that, by the Sequential Computation Thesis, your result will apply to all of the sequential models of computation.
- Many other classes are insensitive to the model of computation in the same way (e.g. **NP** is).
- Practical parallel computers (where you have 2 CPUs, or 10 or 1000000 or any number fixed in advance) is, under this definition, still a sequential model of computation. (This is a bit counter-intuitive because, outside of the Theory of Computer Science, people use the word 'parallel' in opposition to 'sequential', whereas we are saying that 'sequential' includes 'normal' parallelism.) The reason, of course, is that such parallelism just changes the constant hidden under the big-Oh: with your parallel machine you might be able to do things twice as fast or 10 times as fast or 1000000 as fast. 'Normal' parallelism cannot change the growth rate, e.g., from 2^n to n or from n^2 to n or from n to $\log n$. To make these kinds of changes to the growth rate, you need a *growing* number of processors, the exact number depending on n. This, of course, is (probably) not practical and, while researchers may investigate it, it doesn't qualify as a sequential model of computation, so it doesn't invalidate the Sequential Computation Thesis.

41.4 Summary

- In this module, we have explored problems and their solutions. The distinctions we have drawn are quite robust, largely unaffected by using different models of computation, different computers and different programming languages.



Acknowledgements

I brought together some material from [Dun91], [LP81], [Jun] and [Har92] when writing this lecture. I took the title of the lecture from a chapter in [Dew93].

References

[Dew93] A. K. Dewdney. *The (New) Turing Omnibus*. W.H. Freeman, 1993.

[Dun91] P.E. Dunne. *Computability Theory: Concepts and Applications*. Ellis Horwood, 1991.

[Har92] D. Harel. *Algorithmics: The Spirit of Computing*. Addison-Wesley, 2nd edition, 1992.

[Jun] A. Jung. Models of Computation (Course Notes).

[LP81] H.R. Lewis and C.H. Papadimitriou. *Elements of the Theory of Computation*. Prentice Hall, 1981.