Great Ideas in Computing

University of Toronto CSC196
Fall 2025

Class 8: September 29 (2025)

Announcements and Agenda

Announcements

- Our first guest lecture will be given on October 8 by Professor Nathane Wiebe on the subject of quantum computers.
- Thanksgiving Monday October 13. No classes.
- Assignment 2 is due October 15. The first two questions have now been posted on the course web page. .
- Our first (of two) quizzes will be held October 17 in the usual time and place for the tutorial.

Today's Agenda

- Continuation of discussion of Turing machines and what is computable. The Church-Turing thesis. Universal Turing machines. Reductions.
- We should decide what topic we want to do next. Two possible choices:
  1. Extend the topic of "what is computable" to the topic of "what is *efficiently* computable".
  2. Search engines, how they work, and why has search been so profitable for Google.

A pictorial representation of a Turing machine

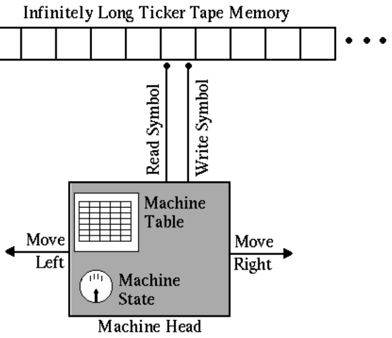

Figure: Figure taken from Michael Dawson “Understanding Cognitive Science”

Comments on Turing's model

- Formally, a Turing machine algorithm is described by the following function $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$
  Q is a *finite* set of *states*. Γ is a *finite* set of symbols.
  Note: Some might say this is the definition of a single Turing machine
- We can assume there is a halting state q_{halt} such that the machine halts if it enters state q_{halt}. There is also an initial state q_0.
- A Turing machine is allowed to read and write symbols from some *finie* alphabet Γ. We view a Turing machine P as computing a function $f_P : \Sigma^* \rightarrow \Sigma^*$ where $\Sigma \subseteq \Gamma$ where $y = f(x)$ is the string that remains if (and when) the machine halts. There can be other conventions as to interpreting the resulting output y.
- Note that the model is precisely defined and the definition of a computation step is also precise.
- For decision problems, we can have two halting states, a YES and NO state.

Turing machine simulators

There are many examples of Turing machine (TM) simulators on the web. Here are two links to sites for simulating Turing machines. They all have examples, some easy to usderstand; some not so easy. Some sites have bigger librariers and some mainly want you to construct your own program.

- https://morphett.info/turing/turing.html
- https://turingmachinesimulator.com/

NOTE: Writing TM programs is tedious except for some very simple functions. I would not spend much if any time constructing TM programs.

The take-away message is that the allowable TM operations are remakably basic, and yet, any computable function can be constructed from these basic operations.

More about Turing's seminal results

- Turing showed that there is a Universal Turing machine (UTM) call it U. That is, given an input $p\#x$ the machine interprets the string p as a Turing machine description (i.e. as a state transition function δ) and $x \in \Sigma^*$ is interpreted as the input to the machine P described by p and $f_U(p\#x) = f_P(x)$.
- In modern terms, a UTM is an *interpreter*.
- Turing showed that the **halting problem** is undecidable. That is, there does not exist a fixed TM F such that F when executed on an input string $(p\#x)$, where p encodes a TM P, whether or not P will halt and correctly decide if P halts on the input string x. It is also undecidable if a TM P will halt on all inputs.
- As a consequence, this means that you cannot have a compiler which will check for the algorithm $\mathcal{A}$ you have written whether or not $\mathcal{A}$ will halt on every input.

The Church-Turing hypothesis

As already discussed, there is a wide consensus on the acceptance of the Church-Turing thesis. That is, the equating of *computable* with Turing computable.

I want to emphasize however, here we are talking about discrete computation. There are some different formulations of what we mean by say computing a function $f : \mathbb{R} \rightarrow \mathbb{R}$.

For computability and discrete computation, it doesn't matter if we consider functions $f : \mathbb{N} \rightarrow \mathbb{N}$ or $f : \Sigma^* \rightarrow \Sigma^*$ for any for alphabet Σ with $|\Sigma| \geq 2$. Why?
When $|\Sigma| = 1$, there are some issues regarding how to encode things and when we consider complexity, it is best to assume $|\Sigma| \geq 2$ when representing integers.

The Church-Turing hypothesis continued

For a number of proposed alternative computational models either it has been shown that the model is either equivalent in representational power or weaker.

Moreover, let $\mathcal{M}$ an alternative model, then one can exhibit a mapping of any $\mathcal{M}$ computation into a Turing computation. This implies that the Turing model is at least as powerful as the $\mathcal{M}$ model. To prove equivalence of the models, one needs to show the converse is also true. To show that $\mathcal{M}$ is a weaker model, one has to exhibit some function that can be computed by a T.M. but not by $\mathcal{M}$.

We can extend the defiintion of the Turing model (and other models) to the computation of the computation of higher order *operators*. For example let F be the set of one variable differentiable functions. Then we would want an operator that takes a function to its derivative. Similarly we would want an operator for integrating a function. And, of course, we would have to say how we are representing the function being differentiated or integrated.

Repeating the pictorial representation of a Turing machine

Notice the remarkable simplicity of this model.

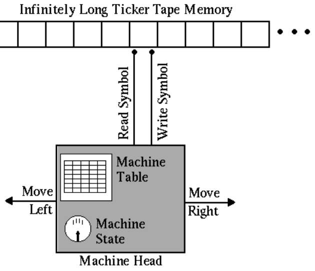

Figure: Figure taken from Michael Dawson "Understanding Cognitive Science"

Turing reducibility

Alan Turing also introduced the idea of an *oracle Turing machine*. While one can formalize this concept, we will just use some examples. In today's terminology it is easy to understand the concept if one is familiar with subroutines.

Intuitively, when we say that solving a problem A (say computing a function) by reducing the problem to being able to solve a problem B, we mean that in a program P_A for A, we can ask ask (perhaps many times) for a different program P_B to solve B given some input z. That is, P_A temporarily turns control over to P_B (specifying some input z and P_B returns with the solution $B(z)$ and P_A continues its computation.

Let's just consider this concept in terms of $\{$YES,No$\}$ decision problems. Here is an example in terms of graphs.

Reducing strong connectivity to s-t connectivity

In a directed graph, $G = (V, E)$, a directed path from a node (also called a vertex) $u \in V$ to a node $v \in V$ is a sequence $u = u_0, u_1, u_2, \ldots, u_r = v$ such that $(u_i, u_{i+1}) \in E$. E is the set of edges in the directed graph.

The strong connectivity decision problem is : "Given a directed graph G, output YES if for every $u, v \in V$, there is a directed path from u to v, output NO otherwise.

For a directed graph the s-t connectivity problem is : Given a directed graph G and two vertices $s, t \in V$, is there a path from s to t.

How would you reduce the strong connectivity problem to the s-t connectivity problem?

Reducing strong connectivity to s-t connectivity

In a directed graph, $G = (V, E)$, a directed path from a node (also called a vertex) $u \in V$ to a node $v \in V$ is a sequence $u = u_0, u_1, u_2, \ldots, u_r = v$ such that $(u_i, u_{i+1}) \in E$. E is the set of edges in the directed graph.

The strong connectivity decision problem is : "Given a directed graph G, output YES if for every $u, v \in V$, there is a directed path from u to v, output NO otherwise.

For a directed graph the s-t connectivity problem is : Given a directed graph G and two vertices $s, t \in V$, is there a path from s to t.

How would you reduce the strong connectivity problem to the s-t connectivity problem?
We could simply ask "the oracle" for s-t connectivity if the answer is YES for ever pair of nodes $s, t \in V$.

Two consequences of a reduction

We denote a reduction of problem A to problem B by the notation $A \leq_T B$ where the T stands for Turing.

Suppose we can reduce a decision problem A to a decision problem B. Then if we can decide problem B, we can decide problem A.

This is how we normally think of reductions if we are an algorithm designer.

If $A \leq_T B$, there is another consequence, namely the *contrapositive*: if problem A is undecidable, then problem B is undecidable.

In propositional logic we say $B \implies A$ is equivalent to $\neg A \implies \neg B$. That is, $\neg A \implies \neg B$ is the contrapositive of $B \implies A$. (Do not confuse with the *converse* where the converse of $B \implies A$ is $A \implies B$.)