Great Ideas in Computing

University of Toronto CSC196
Fall 2023

Week 4: October 9 - October 13 (2023)

Week 5 slides

Announcements:

- I have posted Assignment 2 which is due October 27, 9AM.
- The first quiz will take place Friday, October 20 during the tutorial class.

This weeks agenda

- Formalizing the Turing machine computational model.
- The Church-Turing thesis.
- Universal Turing machines
- Computable reductions and transformations.
- The halting problem.

A pictorial representation of a Turing machine

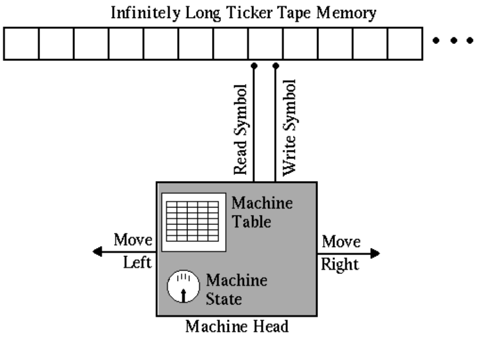

Figure: Figure taken from Michael Dawson “Understanding Cognitive Science”

Comments on Turing's model

- Formally, a Turing machine algorithm is described by the following function $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$
  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 $f_P : \Sigma^* \to \Sigma^*$ 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. (See slides 26 and 27.)
- For decision problems, we can have two halting states, a YES and NO state.

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.

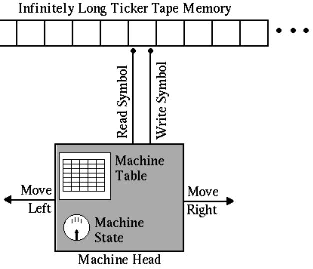

Figure: Figure taken from Michael Dawson "Understanding Cognitive Science"

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 $A \implies B$.)

Decidability and undecidability

When we discuss decidability and undecidability, we usually formulate this in tems of language recognition. Let's consder Turing machines with input strings $w \in \Sigma^*$ for some finite alphabet Σ.

Notation: If M is a TM, we let $< M >$ be an encoding of M and for a string $w \in \Sigma^*$ we let $|w|$ be the length of the string. We also use "iff" for "if and only if".

A language is a subset of strings; that is, $\mathcal{L} \subseteq \Sigma^*$ satisfying some property. We say that $\mathcal{L}$ is decidable (also called recognizable) if there is a TM M that will halt and accept a string w if $w \in \mathcal{L}$ and it will halt and reject w if $w \notin \mathcal{L}$. Note, in particular, M halts (i.e. stops) on every input. Otherwise we say that $\mathcal{L}$ is undecidable.

A simple example of using a reduction to prove undecidability

For example consider the following languages:
$\mathcal{L}_1 = \{< M, w > | M \text{ is a TM that halts on input string } w\}$.
Note: The question as to whether or not $\mathcal{L}_1$ is decidable is what is usually called the halting problem.
$\mathcal{L}_2 = \{< M > | M \text{ is a TM that halts on all inputs } w\}$.
In fact, we can let w be a fixed input (e,g, $w = 010$) or we can even let w be $w =< M >$. That is, we can consider the languages
$\mathcal{L}_3 = \{< M > | M \text{ is a TM that halts on the input string } w = 010\}$ or
$\mathcal{L}_4 = \{< M, < M >> | M \text{ is a TM that halts on the input string } w =< M >\}$

Suppose $\mathcal{L}_1$ (or $\mathcal{L}_3$, or $\mathcal{L}_4$ is undecidable (which they are). We want to show that $\mathcal{L}_2$ is undecidable.

Let's show that $\mathcal{L}_3 \leq_T \mathcal{L}_2$

Tranformations as a special case of Turing reductions

‘

We ended the previous slide claiming that we can show that $\mathcal{L}_3 \leq_T \mathcal{L}_2$. Thus if $\mathcal{L}_3$ is undecidable then $\mathcal{L}_2$ mjust also be undecidable.

In fact, we will use a very simple type of reduction which I will call a *transformation* and which we can denote by $\leq_\tau$. We say that $A \leq_\tau B$ if there is a computable function f such that $w \in A$ iff $f(w) \in B$.

It should be easy to see that $A \leq_\tau B$ is a special case of $\leq_T$. Is this obvious?

Showing the desired transformtion for the languages on the last slide

Here is a description of the transformation of $< M >$ to $f(< M >)$ such that $< M_3 > \in \mathcal{L}_3$ iff $f(< M_2 >) \in \mathcal{L}_2$.

Given an encoding of a Turing machine TM M_3, we are going to construct a TM M_2. That is, $< M_2 >= f(< M_1 >$. M_2 operates on input string w as follows:
If $w \neq 010$, M_2 halts. (It isn't important if M_2 accepts or rejects.)

Claim: M_2 accepts w iff M_3 accepts w. That is, $< M_3 > \in \mathcal{L}_3$ iff $< M_2 > \in \mathcal{L}_2$

Given that the halting problem is undecidable, many other problems can be proved to be undecidable using reductions. Most of these problems do not mention Turing machines but undecidability comes from the problem "being able to encode a Turing computation".

Expanding on the previous slide

A halting computation is a composition of Turing machine *configurations* $C_1, C_2, \ldots, C_t$ such that C_{i+1} is the configuration of the Turing machine that follows from executing one step of the Turing machine when it is in configuration C_i and C_t is in a halting state.

In the transformation we described, we used the fact that a Turing machine can simulate the computation of another Turing machine. This is what Turing called a *universal Turing machine*i (UTM). In modern terms, a UTM is an interpreter.

A universal Turing machine (UTM) $\mathcal{U}$ is a T.M. such that

$$\mathcal{U}(< \mathcal{M} >, w) = \mathcal{M}(w)$$

.

That is, $\mathcal{U}$ simulate exactly what $\mathcal{M}$ does on input w. Turing showed how to design a UTM.

The Entscheidungsproblem

In his seminal paper "On Computable Numbers With an Application to the Entscheidungsproblem (i.e. decision problem), Turing uses his model and the undecidability of the halting problem, to prove the undecidabiliy of the "Entscheidungsproblem" posed by Hilbert in 1928. (Church provided an independent proof within his formalism.)

Sometimes this is informally stated as "can mathematics be decided ?"

The Entscheidungsproblem question refers to the decidability of predicate logic which Church and Turing independently resolved in 1936-1937. It would take a little while to formally define this "Entscheidungsproblem" but here is an example of the kind of question that one wants to answer:
Given a formula such as $\forall x \exists y : y < x$
can we determine if such a formula is always true no matter what what ordered domain x, y and $<$ refer to?
For example, $x < y$ and $y < z$ implies $x \neq z$ is always true.
But $x < y$ implies $\exists z : x < z < y$ is not true of all ordered domains (e.g., consider the integers) but is true of the rationals.