Turing Machines: Limits of Decidability
COMP1600 / COMP6260

Dirk Pattinson
Australian National University

Semester 2, 2017
Catch Up / Drop in Lab

When    Fridays, 15.00 - 17.00
Where   N335, CSIT Building (bldg 108)
Until   the end of the semester

This is *not* a tutorial, but we’re happy to answer *all* your questions. No registration or anything required – just drop by!
Strings and Machines

TM Encoding
- a TM may have several encodings (re-order states or symbols)
- some strings may not be encodings at all, e.g. 110.

Questions. Given binary string $w$
- Is $w$ an encoding of a TM? \textit{(easy)}
- If $w$ \textit{is} an encoding of a TM $M$, does $M$ accept the string $w$, or reject it?

Q. Why does this make sense?
- it amounts to asking whether a TM accepts itself
- key step to exhibit non-recursively enumerable languages
A language that is not recursively enumerable

**Definition.** $L_d$ is the set of binary strings $w$ such that

\[ w \text{ is a code of a Turing Machine } M \text{ that rejects } w \]

- $d$ for ‘diagonal’
- ‘reject’ means halting in a non-final state, or non-termination.

**Theorem.** $L_d$ is not recursively enumerable, i.e there is no TM that accepts *precisely* $L_d$.

**Proof (Sketch)**
- Suppose for contradiction that TM $M$ exists with $L(M) = L_d$.
- $M$ has a binary encoding $C$ (pick one of them)
- Question: is $C \in L_d$?
A language that is not recursively enumerable ctd.

Two Possibilities.

Option 1. \( C \in L_d \)
- then \( M \) accepts \( C \) because \( M \) accepts all strings in \( L_d \)
- but \( L_d \) contains only those TM (codes) \( w \) that reject \( w \)
- hence \( c \notin L_d \) – contradiction.

Option 2. \( C \notin L_d \).
- then \( M \) rejects \( C \) because \( M \) rejects all strings not in \( L_d \).
- but \( L_d \) contains all the encodings \( w \) for TMs that reject \( w \)
- so \( C \in L_d \) – contradiction!

As we get a contradiction either way, our assumption that \( L_d \) can be recognised by a TM must be false.

In Short. There cannot exist a TM whose language is \( L_d \).
Reflecting on this example

Reflections on the proof.

- the language $L_d$ is artificial, designed for easy contradiction proof
- but it is *easy* to define, like any other language (using basic maths)
- if we believe the Church-Turing thesis, there is *no* program that would be able to answer the question $w \in L_d$.

Questions.

- are all non-recursively enumerable languages “weird” (like $L_d$)?
- are the problems that are not computable but of interest in practice?
The Halting Problem

Halting Problem.

- Given a Turing machine $M$ and input $w$, does $M$ halt (either accepting or rejecting) on $w$?

Blue Screen of Death

- answering this question could lead to auto-testing
- programs that don’t get stuck in infinite loops . . .

Partial Answers.

- can give an answer for some pairs $M, w$
- e.g. if $M$ accepts straight away, or has no loops
- difficulty: answer for all $M, w$. 
The Halting Problem – a First Stab

First Attempt at solving the halting problem

- Feed $M$ the input $w$, sit back, and see if it halts!

Critique.

- this is a *partial* decision procedure
- if $M$ halts on $w$, will get an answer
- will get *no* answer if $M$ doesn’t halt!

Comparison with $L_d$

- this is *better* than $L_d$
- for $L_d$, we cannot guarantee *any* answer at all!
Recursively Enumerable vs. Recursive Languages

Recursively Enumerable Languages.
A language $L$ is *recursively enumerable* if there is a Turing machine $M$ so that $M$ accepts precisely all strings in $L$ ($L = L(M)$)

- if $w \notin L$, the TM may never terminate and give an answer . . .
- also called *semidecidable*
- can enumerate all elements of the language, but cannot be sure whether a string eventually occurs

Recursive Languages.
A language $L$ is *recursive* if there is a Turing machine *that halts on all inputs* and accepts precisely the strings in $L$, $L = L(M)$.

- *always* gives a yes/no answer
- also called *decidable*

Example.
- the language $L_d$ is not recursively enumerable
- the halting problem is recursively enumerable
- . . . but not recursive (as we will see next)
Universal TM

TM that simulate other TMs

- given a TM $M$, it’s easy to work out what $M$ does, given some input
- it is an *algorithm*. If we believe the Church-Turing thesis, this can be accomplished by (another) TM.

Universal TM

- is a TM that accepts two inputs: the *coding* of a TM $M_s$ and a string
- it *simulates* the execution of $M_s$ on $w$
- and accepts if and only if $M_s$ accepts $w$.

Construction of a universal TM

- keep track of current state and head position of $M_s$
- scan the TM instructions of $M_s$ and follow them
- (this requires lots of coding but is possible)
The Halting Problem as a Language Problem

Slight Modification of universal TM:

- $U_1$ is a universal TM with all states accepting
- hence if $U_1$ halts, then $U_1$ accepts.

Halting Problem formally

- Is $L(U_1)$ recursive?

Observation.

- all problems can be expressed as language problems
- we know that $L(U_1)$ is recursively enumerable – by definition

Q. Is $L(U_1)$ even recursive?
- can we design a “better” TM for $L(U_1)$ that always halts?
The Halting Problem is Undecidable

Theorem. The halting problem is undecidable.

Proof (Sketch).

- Suppose we had a TM $H$ that always terminates so that $L(H) = L(U_1)$ ($H$ for halt)
- Construct a new TM $P$ (for paradox)

Construction of $P$: $P$ takes one input, an encoding of a TM

- If $H$ accepts $(M, M)$ (i.e. if $M$ halts on its own encoding), loop forever.
- If $H$ rejects $(M, M)$, halt.

Q. does $P$ halt on input (an encoding of) $P$?

- **No** – then $H$ accepted $(P, P)$, so $P$ should have halted on input $P$.
- **Yes** – then $H$ rejected $(P, P)$, so $P$ should not have halted on input $P$.

Contradiction in both cases, so $H$ cannot exist.
Reflections on the proof

Positive Information.
- to show that a language is (semi-) decidable, one usually needs to exhibit an algorithm. This generates information (the algorithm)

Negative Information.
- to show that a language is not decidable, assume that there is a TM for it, and show a contradiction. This (just) shows impossibility.

Reduction.
- standard proof technique
- assume that a TM exists for a language $L$
- reduce $L$ to a known undecidable language
- so that a solution for $L$ would give a solution to a known undecidable problem

Example.
- if a TM for language $L$ existed, we could solve the halting problem!
- many other undecidable problems ...
Total Turing Machines

Question. Is there a TM $T$ (for total) that
- always terminates
- takes an encoding of a TM $M$ as input
- accepts if $M$ terminates \textit{on all inputs}?

Reduction Strategy.
- Suppose we had such a TM $T$
- for a TM $M$ and string $w$ define a new TM $M_w$ that ignores its input and runs like $M$ would on $w$
- running $T$ on $M_w$ tells us whether $M$ halts on $w$
- so we would have solved the halting problem
- since the halting problem cannot be solved, $T$ cannot exist.
The Chomsky Hierachy

Recall. Classification of language according to complexity of grammars
- regular languages – FSA’s
- context-free languages – PDA’s
- context-sensitive languages
- recursively enumerable languages – TM’s

Q. Where do *recursive* languages sit in this hierachy? Are the automata for them?
- they sit between context sensitive and recursively enumerable
- and are recognised by *total* TMs that halt on every input.

Structure vs Property
- all other automata had a clear cut definition
- total TMs have a *condition* attached

Problem.
- cannot *test* whether this condition is fulfilled
- so the definition is mathematical, not computational
Back to the Entscheidungsproblem

Q. Can we design an algorithm that *always terminates* and checks whether a mathematical formula is a theorem?
   - this is the *Entscheidungsproblem* posed by Hilbert
   - and what Alan Turing’s original paper was about

More detail.
   - mathematical formula means statement of first-order logic
   - proof means proof in natural deduction (or similar)

Ramifications.
   - all mathematicians could be replaced by machines

Turing’s Result.
   - the set of first-order formulae that are provable is *not* recursive.
   - the existence of a TM that computes the Entscheidungsproblem leads to a contradiction

Other Approaches.
   - Church showed the same (using the λ-calculus) in 1932
   - was not widely accepted as λ-calculus is less intuitive
Each of the inner sets is a tiny proportion of the set that contains it.
Time Complexity

Q. It’s good about solving problems in general. How about efficiency?

Inverting booleans. Easy: constant time.

Multiplication of integers. Easy: polynomial time
- takes time that is proportional to the square of the total number of digits
- if we double the digits, it takes $4$ times as long.
- if $n$ are the total number of digits, multiplication has complexity of the order $n^2$.

Matrix Multiplication. Polynomial, of order $n^{2.376}$

Theorem Proving. Hard, sometimes of order $2^{2^n}$ (or even undecidable)

Feasible Problems.
- can be solved in polynomial time, i.e. in time of the order $n^k$, for some $k$
- $n$ is the length of (the representation of) the input.
P vs. NP: Signpost

Polynomial Time. The complexity class \( P \)
- problems (languages) that can be decided in the order of \( n^k \) steps
- usually considered feasible

Non-deterministic polynomial time. The complexity class \( NP \)
- problems that can be decided \textit{with guessing} in polynomial time
- alternatively, problems whose solutions can be verified in polytime

Example. Boolean satisfiability is in \( NP \)
- given boolean formula \( A \), can \( A \) evaluate to \( T \)?
- can guess solution (assignment)
- alternatively, can verify correctness of assignment in polynomial time

As a slogan. Coming up with a solution within a time bound seems intuitively harder than checking someone else’s solution in that time.

Big Open Problem. Is \( P = NP \)?
- most important open problem in our discipline
- 1,000,000 prize by the Clay maths foundation
Computational Problem.

- given by a *Language* $L$, say
- Question: for a string $w$, is $w \in L$?

Solution. A Turing machine $M$ that

- always halts
- and accepts $w$ if and only if $w \in L$.

Time Complexity.

- Given $w$, can count the number of steps of $M$ to termination
- this defines a function $f(w)$ *dependent on the input*
Time Complexity – Abstraction

**Problem.** Number of steps function usually *very complicated*
- for example, \( n^{17} + 23n^2 - 5 \)
- and hard to find in the first place.

**Solution.** Consider *approximate* number of steps
- focus on *asymptotic* behaviour
- as we are only interested in *large* problems

**Landau Symbols.** for \( f \) and \( g \) functions on natural numbers
- \( f \in \mathcal{O}(g) \) if \( \exists c.\exists n_0.\forall n \geq n_0 (f(n) \leq c \cdot g(n)) \)
- “for large \( n \), \( g \) is an upper bound to \( f \) up to a constant.”

**Idea.** Abstract details away by just focussing on upper bounds
- e.g. \( n^{17} + 23n^2 - 5 \in \mathcal{O}(n^{17}) \)
Landau Symbols: Examples

Examples.

- **Polynomials**: leading exponent dominates
  - e.g. \( x^n + \text{lower powers of } x \in \mathcal{O}(x^n) \)

- **Exponentials**: dominate polynomials
  - e.g. \( 2^n + \text{polynomial} \in \mathcal{O}(2^n) \)

**Important Special Cases.**

- **linear.** \( f \) is linear if \( f \in \mathcal{O}(n) \)
- **polynomial.** \( f \) is polynomial if \( f \in \mathcal{O}(n^k) \), for some \( k \)
- **exponential.** \( f \) is exponential if \( f \in \mathcal{O}(2^n) \)
Application to Computational Problems

Definition.
A computational problem (given by a language $L$) is in $O(f)$ if there is a Turing machine $M$ that
- always terminates, and accepts precisely all strings in $L$
- on every input string of length $n$, terminates in $g(n)$ steps and $g \in O(f)$

Example: Regular Languages
- Given: regular language $L$
- Question: what’s the complexity of deciding whether $w \in L$?

More Detail.
- need to construct a Turing machine that decides whether $w \in L$ or not.
- how many steps (depending on length of input string) does $M$ take?
**Application to Computational Problems**

**Definition.**
A computational problem (given by a language \( L \)) is \( \text{in} \ \mathcal{O}(f) \) if there is a Turing machine \( M \) that

- always terminates, and accepts precisely all strings in \( L \)
- on every input string of length \( n \), terminates in \( g(n) \) steps and \( g \in \mathcal{O}(f) \)

**Example: Regular Languages**
- Given: regular language \( L \)
- Question: what’s the complexity of deciding whether \( w \in L \)?

**More Detail.**
- need to construct a Turing machine that decides whether \( w \in L \) or not.
- how many steps (depending on length of input string) does \( M \) take?

A. This is *linear*. Think of finite automata.
Example: Graph Connectedness

Reminder. A graph is a pair \( G = (V, E) \) where
- \( V \) is the set of vertices of the graph
- \( E \) is the set of edges, a collection of two-element subsets of \( V \)

Example.

Formally. \( G = (V, E) \) with
- \( V = \{0, 1, 2, 3, 4\} \)
- \( E \) consisting of \( \{0, 2\}, \{0, 1\}, \{0, 3\}, \{1, 2\}, \{1, 3\}, \) and \( \{3, 4\} \).
- Note: Edges are not directional.
Connected and non-connected Graphs

Definition. A graph is *connected* if there is a path between any two nodes.

Connected Graph Example.

Non-Connected Graph Example
The Connected Graph Problem

Problem. Given a graph $G = (V, E)$

- is $G$ connected?
- what is the complexity of deciding whether $G$ is connected?

Algorithm.

- Pick a vertex $v$ in $G$ as starting vertex
- do a breadth-first search and remember vertices encountered
- if the total number of vertices found equals the number of vertices in the graph, we know that $G$ is connected.

By Church-Turing Thesis. The problem “Is $G$ connected?” is clearly computable.

Q. How many steps does an algorithm take to figure this out?

- number of steps refers to “on a Turing machine”
- but input to TMs are strings, not graphs . . .
Coding of Graphs as Strings

Recall. We have coded TM transition tables as strings.

Coding of graphs:

- vertices: numbered 0 to \( n \), can be coded by \( n \) in binary
- single edge: pair \((n, k)\), can be coded by \( 01 \ldots 0 \# 11 \ldots 1 \).
- set of edges: can be coded by \( e_1 \# \# e_2 \ldots e_{l-1} \# \# e_l \)

Complete Graph

\[
10 \ldots 11 \# \# 11 \ldots 1 \# 10 \ldots 0 \# \# \ldots \# \# 11 \ldots 0 \# 11 \ldots 0
\]

- **green** number of vertices
- **blue** set of edges
- **black** separators
Question, reloaded.

**Computational Problem.** Given a graph $G = (V, E)$, how many steps does a TM take to determine whether the encoding of $G$ is a connected graph?

- the encoded graph is now a string
- number of steps relative to the length of encoding

**Difficulty.** Exact answers require *way too much* bookeeping.

- Landau symbols $O(\cdot)$ allow us to be “generous”
- required answer of the form “in $O(f)$”.
Complexity Analysis

Algorithm in Turing machine form.

- pick vertex 0 initially.
- designate vertex 0 as “to explore” (e.g. by writing its binary encoding to the right of the input)
- for every vertex $v$ that is (still) to explore:
  - search through the edges to find other vertices it connects with
  - if a connected vertex is neither explored nor marked “to explore”, mark it as “to explore” (e.g. by writing its binary encoding to the right of the input, with a special separator)
  - mark the vertex $v$ as “explored”
- check that the number of vertices found is equal to the number of vertices in the graph.
Worst Case Analysis

Worst Case for a given graph $G$ with $n$ vertices
- When exploring a vertex, need to check $n^2$ edges
- For every edge checked, one more vertex to explore
- this needs to be done for every vertex

High Level Analysis. Of complexity $O(n^3)$ – *polynomial*
- need to do $n^2$ checks, at most $n$ times

Overhead of a Turing implementation.
- checking whether two vertices match: *polynomial*
  - at most $n$ (in fact, $\log n$) bitwise comparisons
  - and going back and forth over the tape, at most $n^2 \cdot n$ times
- adding another vertex to the list: *polynomial*
  - at most $n$ bits to add
  - and going back and forth over the tape, at most $n^2 \cdot n$ steps

Summary. Polynomial Complexity
- polynomially many “high level” steps
- each of which takes polynomial time
Other Problems: Propositional Satisfiability

**Given.** A propositional formula, constructed from $\land$, $\lor$, $\rightarrow$, $\neg$, $T$, $F$ and variables.

**Question.** Is there a truth value assignment to the propositional variables such that the formula evaluates to $T$?

**Naive Algorithm.** Truth tables
- loop through all possible assignments and evaluate the formula

**Questions.**
1. How many truth assignments do we need to check?
2. How do we measure the size of the input?
3. What is the worst case complexity of this algorithm?
Complexity Class: Polynomial Time

Definition. The class \( \mathbf{P} \) of \textit{polynomial time} decision problems consists of all problems that can be answered in time \textit{polynomial} in the input

- of order \( \mathcal{O}(n), \mathcal{O}(n^2), \mathcal{O}(n^3), \ldots \)

Examples.

- check whether a graph is connected
- check whether a list is sorted
- check whether a propositional formula is true for a given valuation

Last Problem. Have \textit{two} inputs

- need only \textit{one line} of the truth table
- according to the valuation given
Other Problems: Propositional Satisfiability

Given. A propositional formula, constructed from $\land$, $\lor$, $\rightarrow$, $\neg$, $T$, $F$ and variables.

Coding.
- have new tape symbols for $\land$, $\rightarrow$, etc.
- assume that variables are numbered, encode in binary

Worst Case.
- variables proportional to length of formula (e.g. $p_1 \land p_2 \land p_3 \land \ldots$)
- exponentially many valuations

This Algorithm
- at least exponential, e.g. $O(2^n)$
- and in fact exponential

Q. Can we do better?
Other Problems: Propositional Satisfiability

Given. A propositional formula, constructed from $\land$, $\lor$, $\rightarrow$, $\neg$, $T$, $F$ and variables.

Coding.
- have new tape symbols for $\land$, $\rightarrow$, etc.
- assume that variables are numbered, encode in binary

Worst Case.
- variables proportional to length of formula (e.g. $p_1 \land p_2 \land p_3 \land \ldots$)
- exponentially many valuations

This Algorithm
- at least exponential, e.g. $O(2^n)$
- and in fact exponential

Q. Can we do better?

A. Probably not . . . this is the $1,000,000$ Clayton math problem
Propositional Satisfiability

Verifying whether a formula evaluates to $T$ for an assignment
- takes polynomial time

Determining whether a satisfying assignment exists
- takes exponential time
- but if we could guess an assignment, it would be fast (polynomial)

Observation. The (coding of) a valuation is polynomially large
- in fact, shorter than the formula, a sequence of 0s and 1s

Non-Deterministic Machines (informally)
- like non-deterministic finite automata: more than one transition possible
- propositional satisfiability: guess a valuation, then check
Complexity Class: Nondeterministic Polynomial Time

Definition. The class \( \text{NP} \) of \textit{non-deterministic polynomial time} decision problems consists of all decision problems \( L \) that can be solved by a \textit{non-deterministic} Turing machine in polynomial time:

- we don’t define this type of machine formally
- idea: can make \textit{guesses} at every stage
- accepts if the machine \textit{can} move into final state

Alternative Characterisation \( L \) is in \( \text{NP} \) if, for every string \( w \in L \) there exists a \textit{certificate} \( c \) such that:

- \( c \) is of polynomial length (in the length of \( w \))
- determining whether \( c \) is a certificate for \( w \in L \) is in \( \text{P} \)

Example. Propositional Satisfiability

- certificates are valuations
- checking the formula under a valuation is polynomial
More Problems

The Independent Set Problem Assume you want to throw a party. But you know that some of your friends don’t get along. You only want to invite people that do get along.

As a Graph.

- vertices are your mates
- draw an edge between two vertices if people don’t get along

Problem. Given $k \geq 0$, is there an independent set, i.e. a subset $I$ of at least $k$ vertices so that
- no two elements of $I$ are connected with an edge.
- i.e. everybody in $I$ gets along

Example of an independent set of size 2

![Graph Example](image-url)
Independent Set: Naive Algorithm

- loop through all subsets of size $\geq k$
- and check whether they are independent sets

Alternative Formulation using guessing:
- *guess* a subset of vertices of size $\geq k$
- *check* whether it is an independent set

Complexity. Independent Set is in **NP**
- represent subsets as bit-vectors (certificates)
- checking is polynomial
Vertex Cover

Vertex Cover. Given a graph $G = (V, E)$, a vertex cover is a set $C$ of vertices such that every edge in $G$ has at least one vertex in $C$.

Example.

![Graph](image)

Vertex Cover Problem. Given a graph $G = (V, E)$ and $k \geq 0$, is there a vertex cover of size $\leq k$?

Naive Algorithm.

- search through all subsets of size $\leq k$
- check whether it's a vertex cover
From Independent Set to Vertex Cover

Reductions. Use solutions of one problem to solve another problem.

Observation. Let $G$ be a graph with $n$ vertices and $k \geq 0$.
- $G$ has a v. c. of size $\leq k$ iff $G$ has an i. s. of size $n - k$.

Reduction. A *polynomial reduction* from decision problem $A$ to decision problem $B$ is a function $f$ that transforms $A$-instances to $B$-instances and
- $w \in A \iff f(w) \in B$ and $f$ is computable in polynomial time.

Example. Vertex cover to independent set

$$(G, k) \mapsto (G, n - k)$$

where $n$ is the number of vertices of $G$. 

![Diagram of reduction from Independent Set to Vertex Cover](attachment:image.png)
Recall. A reduction from decision problem $A$ to $B$ is a polytime function $f$ such that $w \in A \iff f(w) \in B$.

Informally. If we can solve $B$, then we can also solve $A$

- Given $w$, is $w \in A$?
- Compute $f(w)$, and decide whether $f(w) \in B$

Q. If $A$ is reducible to $B$, which of $A$ and $B$ is more “difficult”? 
NP-Completeness

Q. What is the “hardest” or most difficult NP-problem?

A. It’s a problem that all other NP-problems can be reduced to
   - a solution would yield solutions to all NP-problems
   - recall that B is more difficult than A if A can be reduced to B

NP-Hardness and completeness.
   - A decision problem \( L \) is NP-hard if all other NP-problems can be reduced to it.
   - A decision problem is NP-complete if it is NP-hard and in NP.

Hard Theorem. (Stephen Cook 1974) The propositional satisfiability problem is NP-complete
   - have seen that satisfiability is in NP
   - hard part: reduce all NP-problems to satisfiability.
The P vs NP Problem

Big Open Question. is $P = NP$ or not?

- Given that propositional satisfiability is NP-complete
- “all” it takes is to solve one problem efficiently!

Ramifications If $P = NP$ . . .

- could break public-key cryptography
- this includes https-protocol!
- could solve optimisation problems efficiently
- lots of AI (learning) problems have fast solutions
Summary.

Undecidable Problems.
- Problems for which we cannot find an algorithmic answer
- Most famous: halting problem – determine whether a computation terminates

Efficiently Solvable Problems.
- Usually identified with polynomial time, i.e. $\mathbf{P}$

More difficult Problems. Polynomial time with guessing
- Complexity class $\mathbf{NP}$, not considered feasible
- $\mathbf{NP}$-complete problems, like propositional satisfiability

Open Problem. Is $\mathbf{P} = \mathbf{NP}$ or not?
- Neither have proof nor counter-example
- Most important open problem in the discipline