3515ICT: Theory of Computation

Turing machines and undecidability

(IALC, Chapters 8 and 9)

Introduction to Turing's life, Turing machines, universal machines, unsolvable problems.

An undecidable problem (Section 8.1)

No program H can test whether a given program will print `"hello, world"` as its first output on a given input.

Detailed proof by self-reference leading to a contradiction.

Proofs that other problems are undecldable (or unsolvable) by reduction from a known undecidable problem.

Example: Proof that the *halting problem* is undecidable (by reduction from the *hello-world* problem).

Alternative proof that there exist undecidable problems (languages): Let Σ be a finite alphabet. Then Σ^* has countably many elements. Every language over Σ is a subset of Σ^* and is thus countable. By diagonalisation, the number of subsets of Σ^* is *not* countable. But the number of computer programs *is* countable. Hence there exist undecidable problems.

Turing machines (Section 8.2)

A Turing machine (TM) is a finite-state machine with an infinite tape (for input and working storage) and a tape head that can read or write one tape cell and move left or right. It normally accepts the input string, or completes its computation, by entering a final or accepting state.

Example of a TM to recognise the language $\{ 0^n 1^n \mid n \geq 1 \}$ (Exx. 8.2 and 8.3, Figs. 8.9 and 8.10). Machine structure:

```
while 0 {
    // ID is X*q0*Y*1*
    change 0 to X & move R
    while 0 or Y move R
    change 1 to Y & move L  // fail if not 1 (too many 0s)
    while Y or 0 move L
    move R
}
while Y move R  // fail if 1 (too many 1s)
halt
```

We can test this machine on inputs 000111, 00011, 00111, 000110.

Formal definition of a TM $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$: Q is a finite set of states, Σ is a finite input alphabet, Γ is a finite tape alphabet ($\Sigma \subset \Gamma$), $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is a transition function, $q_0 \in Q$ is the initial state, $Bin\Gamma - \Sigma$ is the blank symbol, and $F \subseteq Q$ is the set of final states.

Instantaneous description (ID) $X_1X_2 \dots X_{i-1}qX_i \dots X_n$ of a TM. A move of a TM is denoted $\vdash$; a sequence of zero or more moves is denoted $\vdash^*$.

Turing machines to recognise languages accept by entering a final state (and halting); Turing machines to perform computation may simply halt when the computation is complete.

Example of a TM to recognise palindromes (M, Ex. 16.2, Fig. 16.4). Machine structure:

```
while true {
    if a {
        change a to B & move R
        while a or b move R
        move L
        if B halt                // odd-length palindrome
        if a {
            while a or b move L
            move R
        }
    } else if b {
        ... similarly ...
    } else {
        break
    }
}
if B halt                         // even-length palindrome
```

Test on inputs $abba$, $abbba$, $abaa$.

Example of a TM to recognise $L_{ww} = \{ \, ww \mid w \in \{a, b\}^* \, \}$ (M, Ex. 16.3, Fig. 16.5). Machine structure:

```
// Stage 1: Check length is even, change as / bs to As / Bs
while a or b {
    change a to A or b to B & move R
    while a or b move R
    move L
    change a to A or b to B & move L
    while a or b move L
    if A or B move R
}
// Stage 2: Change As and Bs in first half to as and bs
if Blank halt
if A or B move L
while A or B change to a or b & move L
if Blank move R
// Stage 3: Compare first and second halves
while true {
    if a {
        change a to A & move R
        while a or b or Blank move R
        change A to Blank & move L   // fail if B
        while a or b or Blank move L
        if A or B move R
    } else if b {
        ... similarly ...
    }
}
if Blank halt
```

Test on inputs aba, $abaa$, $abab$.

The language $L(M)$ accepted by a TM M is the set of strings w in Σ^* such that $q_0 w \vdash \alpha q \beta$ for some state q in F and any tape strings α and β.

The set of languages accepted by TMs is called the set of *computably enumerable* languages. The set of languages accepted by TMs that *always halt* is called the set of *computable* languages. Significance of this distinction.

Example of a TM to perform integer subtraction (Ex. 8.4, Fig. 8.12). The initial tape is $0^m 1 0^n$. (Who knows why they reversed the normal roles of 0s and 1s?) The TM halts with 0^{m-n} on its tape. Machine structure:

```
while 0 {
    // ID is q0^(m-k)11^k0(n-k)
    change 0 to B & move R
    while 0 move R
    while 1 move R
    if B { // m >= n, so return 0^(m-n)
        move L
        while 1 change to B & move L
        while 0 move L
        if B change to 0 & halt
    }
    change 0 to 1 & move L
    while 1 or 0 move L
    if B move R
}
// m < n, so return zero (blank tape)
change 1 to B & move R
while 1 change to B & move R
while 0 change to B & move R
if B halt
```

Test on 0000100, 00100, 01000.

It is more common to represent the unary integer $n \geq 0$ by 1^{n+1}, so we can represent 0 by 1 (and not by a blank tape!).

Programming techniques (Section 8.3)

- Storage in the state.

- Multiple tracks.
- Subroutines.

Example of a TM to perform multiplication using a "submachine" (Ex. 8.8, Figs. 8.14 and 8.15).

Exercise: Construct a TM to recognise language $\{ 0^n \mid n \text{ a perfect square} \}$.

Exercise: Construct a TM to compute the factorial function.

Exercise (difficult): Construct a TM to compute Ackermann's function: $A(0, n) = n + 1$, $A(m + 1, 0) = A(m, 1)$, $A(m + 1, n + 1) = A(m, A(m + 1, n))$. This function grows *very* rapidly. (See Turing's World for a solution.)

Extensions of Turing machines (Section 8.4)

A **multitape TM** differs from a TM in that each transition depends on each of the tape symbols and can independently change each tape symbol and move the head on each tape.

Every language accepted by some multitape TM is also accepted by some (single-tape) TM (Theorem 8.9).

The construction (not required) depends on simulating the multiple tapes on a single (multi-track) tape.

The time taken by the single-tape TM to simulate n steps of the multitape TM is $O(n^2)$ (Theorem 8.10).

A **nondeterministic TM** differs from a TM in the same way that an NFA differs from a FA: the value of the transition function $\delta(p, X)$ is a *set* of triples (q, X, D). It accepts an input if *some* sequence of transitions reaches a final state.

If M_N is a nondeterministic TM, then there is a deterministic TM M_D such that $L(M_N) = L(M_D)$ (Theorem 8.11).

The construction of the equivalent (multitape) TM M_D depends on maintaining a `queue` of pending IDs of M_N (p. 341).

Suppose that M_N has at most m choices from any ID. Then the time taken by M_D to simulate n steps of M_N *by this construction* is $O(nm^n)$. It is unknown whether a subexponential simulation is possible.

Summary: Adding features to a Turing Machine does **not** give more computing power.

Restricted Turing machines (Section 8.5)

Multistack Machines A multistack machine is like a PDA with $k \geq 1$ independent stacks. If language L is accepted by some TM, then it is also accepted by some 2-stack machine (Theorem 8.13).

Construction: Let $\alpha q \beta$ be an ID of the TM. Represent this ID by storing α^R on the first stack (i.e., *last*(α) on top), β on the second stack (*first*(β) on top), and performing transitions to maintain this representation.

Exercise: Define a "queue machine". Prove that if a language is accepted by some TM, and hence by some 2-stack machine (see above), then it is also accepted by some queue machine.

Counter machines (not required). Very informally, the class of functions that can be computed by finite state machines with just *two* non-negative integer counters (variables). Transitions depend on the state, the input symbol and which if any of the counters is zero. each transition changes the state, changes the input symbol, and may add or subtract one from either or both of the counters. The resulting set of functions that can be computed is equivalent to the the class of functions that can be computed by TMs.

Turing machines and recursive functions (not required). Very informally, the class of functions over the natural numbers that can be computed by TMs is equivalent to the class of functions that can be expressed by a set of recursive definitions (cf., Ackermann's function above) together with a minimisation (while-statement) operator. (See Martin or Minsky for details.) This latter characterisation of the computable functions, due to Kleene, is a very important one.

Summary: Some restrictions of TMs still allow the same set of functions to be computed (equivalently, allow the same set of languages to be recognised).

Turing machines and computers (Section 8.6)

Clearly, a TM can be simulated by a computer. It is only necessary to have an infinite supply of disks (to simulate) the infinite tape and to manage them appropriately.

More interestingly, a computer can also be simulated by a multitape TM. The construction is described in Section 8.6.2.

The time taken by a multitape TM to simulate n steps of a computer is $O(n^3)$, so the time taken by a single-tape TM to simulate n steps is $O(n^6)$.

A language that is not c.e. (Section 9.1)

A language is *computable* if it is accepted by some Turing machine that always halts. Problems correspond to languages; problem instances correspond to strings. A problem is *decidable* (resp., *undecidable*) if the corresponding language is computable (resp., not computable).

A language is *computably enumerable* (*c.e.*) if it is accepted by some Turing machine. Clearly every computable language is c.e.

Exercise: Prove that every context-free language is computable.

Establish a 1-1 mapping between (positive) integers and binary strings, so we can refer to the ith string, w_i.

Establish a coding for Turing machines as binary strings, so that a TM that is coded as w_i is called the ith TM. Note that the same TM may occur several times in this enumeration.

See Example 9.1 for the coding of a particular TM.

Establish a coding for pairs (M, w), where M is a TM and w is a binary string.

Define the *diagonalisation language*, L_d, as follows:

$$L_d = \{ w_i \mid w_i \notin L(M_i) \} \tag{1}$$

I.e., L_d is the set of binary strings w such that the TM whose code is w does not accept when given w as input.

Theorem: L_d is not c.e., i.e., no TM accepts L_d.

Proof: Suppose $L_d = L(M)$ for some TM M, i.e.,

$$L_d = L(M_i), \text{ for some } i. \tag{2}$$

Now, consider whether $w_i \in L_d$.

- If $w_i \in L_d$, then $w_i \in L(M_i)$ by (2), so $w_i \notin L_d$ by (1), which is a contradiction.
- If $w_i \notin L_d$, i.e., $w_i \notin L(M_i)$ by (2), it follows that $w_i \in L_d$ by (1), which is again a contradiction.

Since both possibilities lead to a contradiction, the assumption that L_d is c.e. is false.

A language that is c.e. but not computable (Section 9.2)

Theorem: If L is computable, then so is its complement, L'.

Proof by modifying the TM that accepts L to become a TM that accepts L'.

Theorem: If L and L' are both c.e., then L is computable.

Proof by combining the TMs that accept L and L' to become a TM that accepts L *and always halts*.

Corollary: L'_d is not computable (else L_d would be computable and hence c.e.)

Define the *universal language*, L_u, as follows:

$$L_u = \{\, s \mid s \text{ is the code for } (M, w), M \text{ is a TM, and } M \text{ accepts } w \,\} \tag{3}$$

We define a TM U that simulates the execution of a given TM M on a given input w. (Details omitted, see pp. 378–379). I.e., U accepts the coded pair (M, w) if and only if M accepts w.

The TM U is hence called a *universal Turing machine*.

Theorem: L_u is c.e. but not computable.

Proof. L_u is clearly c.e., as $L_u = L(U)$. Suppose L_u were computable. Then L'_u would also be computable. Let M be a TM that accepts L'_u, i.e., $L'_u = L(M)$, where M always halts. We can then construct a TM M' that accepts L_d as follows.

- Given input w, M' changes its input to $w111w$.
- M' simulates (behaves as) M on the new input. Suppose w is the ith string. Then M' determines whether or not M_i accepts w_i. Since M accepts L'_u, it accepts if and only if M_i does *not* accept w_i, i.e., if and only if $w_i \in L_d$.

But no TM that accepts L_d exists, so L_u is not computable.

The class of computable languages is closed under complementation (above), union, intersection, concatenation, difference and iteration (star). (I think.)

The class of c.e. languages is closed under union, intersection, concatenation, and iteration. (I think.)

Undecidable problems about Turing machines (Section 9.3)

Reduction between problems (languages). A reduction is an algorithm that transforms an instance of problem P_1 to an instance of problem P_2 such that the two instances have the same answer ("yes" or "no"). If P_1 reduces to P_2, then P_2 is *at least as hard* as P_1. Formally, a reduction from P_1 to P_2 is a TM that takes an instance of P_1 as input and halts with an instance of P_2 as output.

Theorem. If there is a reduction from P_1 to P_2, then:

- If P_1 is undecidable (not computable), then so is P_2.
- If P_1 is not c.e., then neither is P_2.

Proof. See Section 9.3.1.

Examples. Let $L_e = \{\, M \mid L(M) = \emptyset \,\}$ be the set of (codings of) TMs whose language is empty. Let $L_{ne} = L_e' = \{\, M \mid L(M) \neq \emptyset \,\}$ be the set of (codings of) TMs whose language is nonempty. Then L_{ne} is c.e. but not computable, and hence L_e is not c.e. (Section 9.3.2)

A *property* of the c.e. languages is a subset of the c.e. languages. For example, the property of being empty is the set consisting of the empty language. The property of being a regular language is the set consisting of all regular languages. A property is *trivial* if it is either empty (satisfied by no c.e. language) or is the set of *all* c.e. languages. Otherwise, it is *nontrivial*.

Rice's Theorem: Every nontrivial property of the c.e. languages is undecidable. (Proof not required.)

By Rice's Theorem, the following properties are all undecidable:

1. Whether the language accepted by a TM is empty.
2. Whether the language accepted by a TM is finite.
3. Whether the language accepted by a TM is regular.
4. Whether the language accepted by a TM is context-free.

That is, for example, the set of codes of TMs that accept a regular language is not computable.