3004 – Complexity Theory*

Mark Herbster
http://www.cs.ucl.ac.uk/staff/M.Herbster/

2014/2015

*Lecture Notes Copyright ©1997 to 2001, Sean Holden

The Course in a Nutshell

1. **Part I (15 Lectures), taught by Mark Herbster:**
   We study the limits of computation using any present-day (and quite possibly any future computer). We ask the question, *What is computation?* and in answering we prove the existence of *uncomputable* problems, i.e., there is *no* algorithm that will solve them.

2. **Part II (15 Lectures), taught by Robin Hirsch:**
   Having looked at what can not be solved, we turn to the question of wht problems can be solved *using a feasible amount of time or memory.*

The First 15 Lectures

The first 15 lectures of the course cover the following:

Lectures 1 to 6:

- Introduction to the course.
- How do we *formally* approach *real* computations?
- The simplest kind of real computation is the *decision problem*.
- The models of computation presented in your earlier courses are not powerful enough to solve many interesting problems.
- The *Turing Machine (TM)* is essentially equivalent to a normal computer with an infinite amount of memory, and can be dealt with formally.
- We look at this and some equivalent models of computation.
- The *Universal Turing Machine (UTM)*.

Lectures 7 to 10:

- Solvable and unsolvable problems,
- Recursive and recursively enumerable (r.e.) languages,
- A quick proof that most problems are unsolvable! This uses countable and uncountable sets and diagonalization.
- There are limits to what Turing Machines (and hence real computers) can do.
- Two problematic problems.
- The Halting Problem.
- Reduction as a technique for proving interesting problems to be unsolvable, and some examples.

Lectures 11 to 15:

- Post's correspondence problem.
- A problem involving tilings.
- Revision of predicate calculus.
- Proof that satisfiability is an unsolvable problem.

3C04 - Complexity Theory

Lectures 1 to 6: Part A

Decision problems and Turing Machines

Aims:

- To introduce the idea of a *decision problem* and to show how such problems can be represented using a *formal language*.
- To introduce the *Turing machine (TM)*, and show how such machines can be used to solve decision problems (amongst other things).
- To provide evidence that a TM is a very powerful model of computation, and to introduce a more familiar, but equivalent model of computation: the *Register Machine*.
- To introduce *Church's thesis*, and to place it and the preceding material in the context of practical computing.

Decision Problems

Very often, a problem that we want to solve using a computer can be expressed as a *decision problem*. For example,

- Given some graph, is it connected?
- Given some set of cities and a description of the distances between them, is it possible to visit them in turn in such a way that the total length of the route is less than d?
- Given a set of simultaneous equations, does a solution exist?
- Given some shapes to be cut from sheet aluminium, can they all be cut from a single rectangle of size $n \times m$?

And so on. In each case, the answer can be either 'yes' or 'no' (and nothing else). Much of this course will consider problems of this kind.

Decision Problems

In general, any decision problem of interest to us can be describing by two things: a generic *instance*, and a *question* to be asked. For example:

Travelling salesman:

Generic instance: A set of cities and a description of the distances between them.

Question: Does a route of length less than d exist?

Simultaneous equations:

Generic instance: A set of simultaneous equations.

Question: Does the set of equations have a solution?

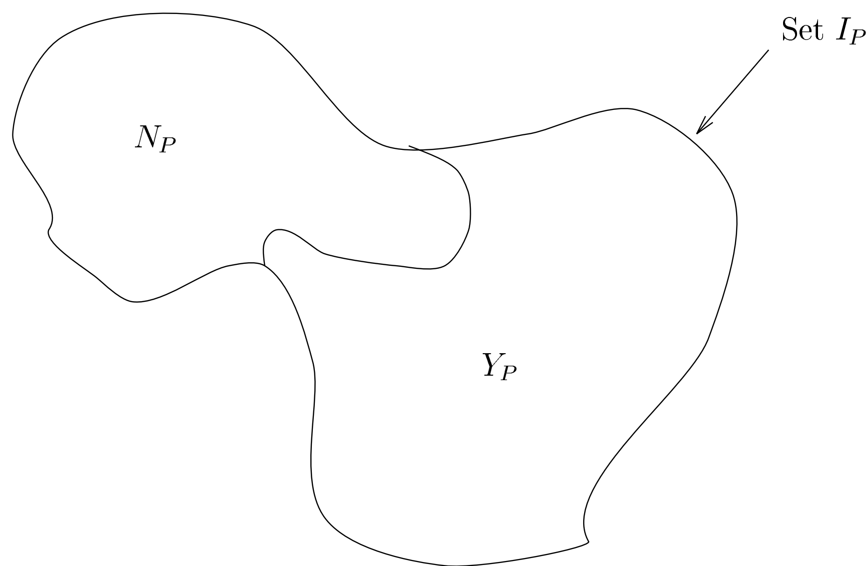
Decision Problems

More formally, any decision problem P consists of a set of instances, I_P which is divided into *yes-instances*, Y_P, and *no-instances*, N_P, in such a way that,

$$I_P = N_P \cup Y_P$$

$$N_P \cap Y_P = \emptyset$$

Diagrammatically,



To solve a decision problem, we need a program that takes any $x \in I_P$ and tells us whether or not $x \in Y_P$. However, to do this we need a way to *encode* instances.

Formal Languages Re-visited

- An *alphabet* Σ is a set of *symbols*. For example $\Sigma = \{1, 0\}$.
- A *string* over Σ is a finite sequence of symbols from Σ, for example,

$$\mathbf{a} = 100101110.$$

- The *length* of a string $\mathbf{a}$ is denoted $|\mathbf{a}|$ and is defined as the number of symbols in $\mathbf{a}$. In the above example $|\mathbf{a}| = 9$.
- The *empty string* has length 0 and is denoted ε.
- If $\mathbf{a} = a_1 a_2 \ldots a_m$ and $\mathbf{b} = b_1 b_2 \ldots b_n$ are strings of finite length, their *concatenation* is the string,

$$\mathbf{ab} = a_1 a_2 \ldots a_m b_1 b_2 \ldots b_n$$

  where $|\mathbf{ab}| = m + n$. For any string $\mathbf{a}$ we have $\mathbf{a}\varepsilon = \varepsilon\mathbf{a} = \mathbf{a}$.
- The set of all finite length strings over Σ is denoted by $\Sigma^\star$.

Formal Languages Re-visited (continued)

- A *(formal) language* L over Σ is a subset of $\Sigma^\star$. For example, using the alphabet given above, we can define the language $L = \{l_1, l_2, l_3, l_4\}$ where,

$$\begin{aligned} l_1 &= 10010111 \\ l_2 &= 1 \\ l_3 &= 00 \\ l_4 &= 0000001 \end{aligned}$$

- The *characteristic function* of a language L is the function $\chi_L : \Sigma^\star \rightarrow \{0, 1\}$ defined as,

$$\chi_L(x) = \begin{cases} 1 & \text{if } x \in L \\ 0 & \text{otherwise.} \end{cases}$$

Turning Decision Problems into Formal Languages

The basic idea is as follows. We construct an *encoding scheme* that allows us to take any instance of a decision problem P and turn it into a string over some alphabet Σ.

$$x \in I_P \xrightarrow{\text{encoding scheme}} \text{code}(x) \in \Sigma^\star$$

In general there will be many different ways of designing the encoding scheme. There are some properties that the scheme should clearly have.

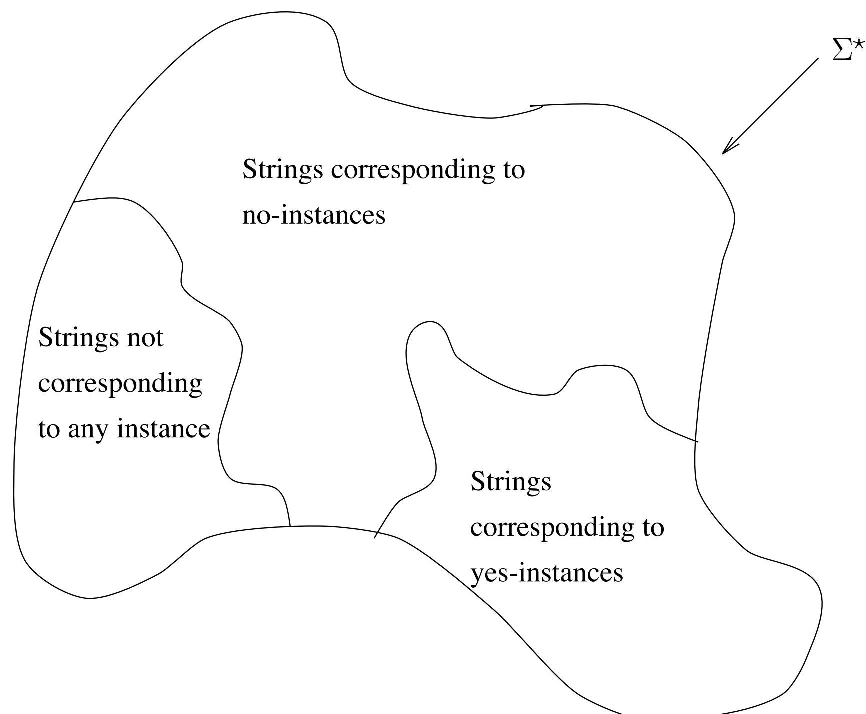
1. If $x_1 \neq x_2$ then we should have code$(x_1) \neq$ code(x_2).
2. Given a string in $\Sigma^\star$ it should be possible to tell whether or not it is the encoded form of an instance.
3. There should be a well-defined process for turning x into code(x) and code(x) into x, for any x.

In the second half of the course, some further properties will also be important. In particular,

1. The strings produced should not be 'padded out' with symbols that are redundant.
2. Numbers should not be represented using unary notation.

Turning Decision Problems into Formal Languages

If we have an encoding scheme using the alphabet Σ, then $\Sigma^\star$ will be divided as follows:



The language L_P associated with a decision problem P is then the set of strings corresponding to yes-instances,

$$L_P = \{x \in \Sigma^\star : x = \mathsf{code}(y) \text{ where } y \in Y_P\}.$$

We then regard a decision problem P as being solvable if a Turing Machine can be designed that tells us whether or not a string $x \in \Sigma^\star$ is a member of L_P.

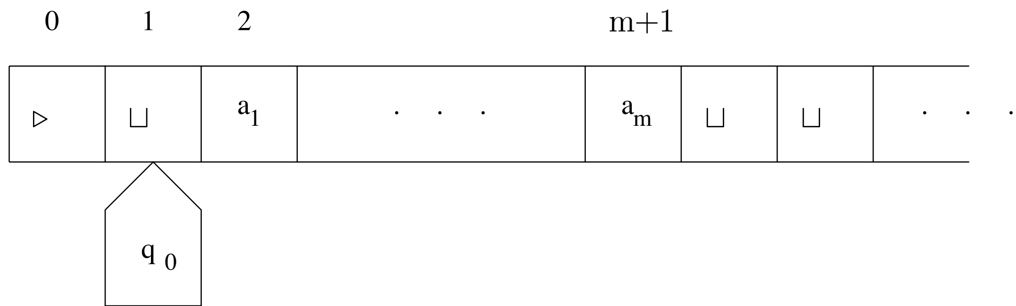
Turing Machines

Henceforth the term 'Turing Machine' will be abbreviated to TM.

Informally, Turing machines can be described as follows:

- A TM has a single *tape*. The tape has a left end and is infinite to the right. It is divided into numbered cells.
- The leftmost cell is numbered 0 and is *always* marked with the symbol $\triangleright$.
- Each cell on the tape can contain a symbol or be blank. Initially the tape is blank, with the exception that cell 0 contains $\triangleright$ and any input string $\mathbf{a} = a_1 a_2 \ldots a_m$ is written in cells 2 to $m + 1$.
- A TM has a *head*, which can be in a finite number of *states*.
- Initially the head is in a distinguished starting state q_0 and scanning cell number 1.

Turing Machines



Turing Machines

- At any time the head is reading the contents of a single cell on the tape. Based on the current contents of the cell and the current state, the TM can either *halt*, or do the following:
  1. Change to a new state.
  2. *Either* Write a new symbol in the current cell, *or* move one cell to the left or right.
- There is an exception: if the head is reading the $\triangleright$ symbol, denoting the left end of the tape, then it can *only* move to the right.
- The actions to be taken are specified by a *program*.

Turing Machines: The Formal Definition

A *Turing Machine* is a 5-tuple (Σ, Q, q_0, H, f) where,

- Σ is a finite alphabet, the symbols of which are the symbols that can appear on the tape. Σ includes the blank symbol, denoted $\sqcup$ and the end of tape symbol $\triangleright$. Σ does *not* contain either of the symbols $\leftarrow$ or $\rightarrow$.
- Q is a finite set of *states*.
- $q_0 \in Q$ is a distinguished state known as the *initial state*.
- $H \subseteq Q$ is a set of states known as *halting states*.
- f is a function $f : (Q \backslash H) \times \Sigma \rightarrow Q \times (\Sigma \cup \{\leftarrow, \rightarrow\})$.

There are two further restrictions on f:

- If $f(q, \triangleright) = (q', a)$ then $a = \rightarrow$. (Translation: you can't fall off the left end of the tape.)
- If $f(q, a) = (q', b)$ then $b \neq \triangleright$. (Translation: the $\triangleright$ symbol is *only* used to mark the left end of the tape.

It is always safe to assume that $|H| = 1$.

Turing Machines: The Formal Definition

- The function f encapsulates the program governing the operation of the TM. In fact, we can write this program down as a finite set of 4-tuples (q, σ, q', σ').
- In each 4-tuple q and σ are the current state and symbol being scanned, q' and σ' are the new state and action (write or move).
- As f is a *function* at most *one* 4-tuple can appear in this list for each possible argument $(q, \sigma) \in (Q \backslash H) \times \Sigma$. We could not, for example, have the two 4-tuples,

$$\begin{array}{l} \vdots \\ (q_1, \sigma_1, q_5, \sqcup) \\ \vdots \\ (q_1, \sigma_1, q_2, \sigma_{10}) \\ \vdots \end{array}$$

  in the program for a TM.
- As f is a *function* it defines an outcome for *every* $(q, \sigma) \in (Q \backslash H) \times \Sigma$.

Turing Machines: The Formal Definition

- Informally, if a TM is in a particular state scanning a particular symbol, there is exactly one course of action it can take. We say that the TM is *deterministic*. In the second half of the course this requirement will be relaxed (!).
- A computation continues on a step by step basis under the control of f. There are two possible outcomes.
  1. If a TM ever reaches a state $h \in H$ then its computation can progress no further. Such a machine is said to *halt*.
  2. A TM *may not halt* when started on a particular input.

Computation, Part 1: Computing a Function

There are two basic ways in which a TM can be used as a computational device. We introduce one now, and the alternative later.

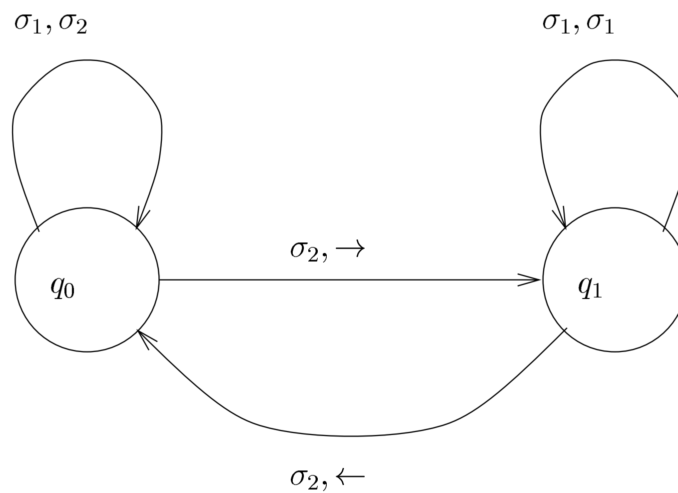
- Define an *input alphabet* $\Sigma_I \subset \Sigma$. This is the set of symbols that can be used to write an input on the tape
- Σ_I does not contain $\triangleright$ or $\sqcup$.
- An input $x \in \Sigma_I^\star$ is written on the tape as described above, using only symbols from Σ_I.
- A machine M computes under the control of f until either:
  1. It halts with the head at position 1, and position 1 contains $\sqcup$. The *output* of the computation is denoted $M(x)$ and is defined to be the string $y \in \Sigma_I^\star$ appearing between position 2 and the next $\sqcup$ to the right. All positions to the right of the output should contain $\sqcup$.
  2. It fails to halt. In this case the output is undefined.
- A given function $F : \Sigma_I^\star \to \Sigma_I^\star$ is *recursive* if there exists a TM M that computes it. That is, for all $x \in \Sigma_I^\star$, $M(x) = F(x)$.

Diagrammatic Representation of TM Programs

TM programs can be represented diagrammatically. For example, the program:

$$(q_0, \sigma_1, q_0, \sigma_2)$$
$$(q_0, \sigma_2, q_1, \rightarrow)$$
$$(q_1, \sigma_1, q_1, \sigma_1)$$
$$(q_1, \sigma_2, q_0, \leftarrow)$$

can be represented as follows:



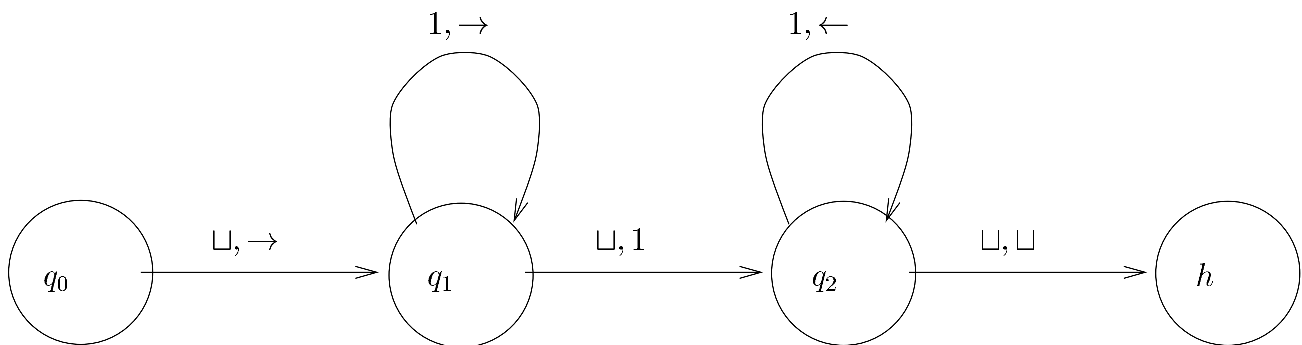
Examples: Computing Functions from $\mathbb{N}^k$ to $\mathbb{N}$

Using the stated approach to computing functions based on strings, we can exhibit some simple computable functions from $\mathbb{N}^k$ to $\mathbb{N}$.

- In general, we can use the input alphabet $\Sigma_I = \{1, \star\}$.
- A natural number $n \in \mathbb{N}$ can be represented using unary notation.
- A k-tuple $x \in \mathbb{N}^k$ can be represented using unary notation numbers separated by the $\star$ symbol.
- For example, the tuple $(4, 5, 1) \in \mathbb{N}^3$ is represented as $1111 \star 11111 \star 1$.

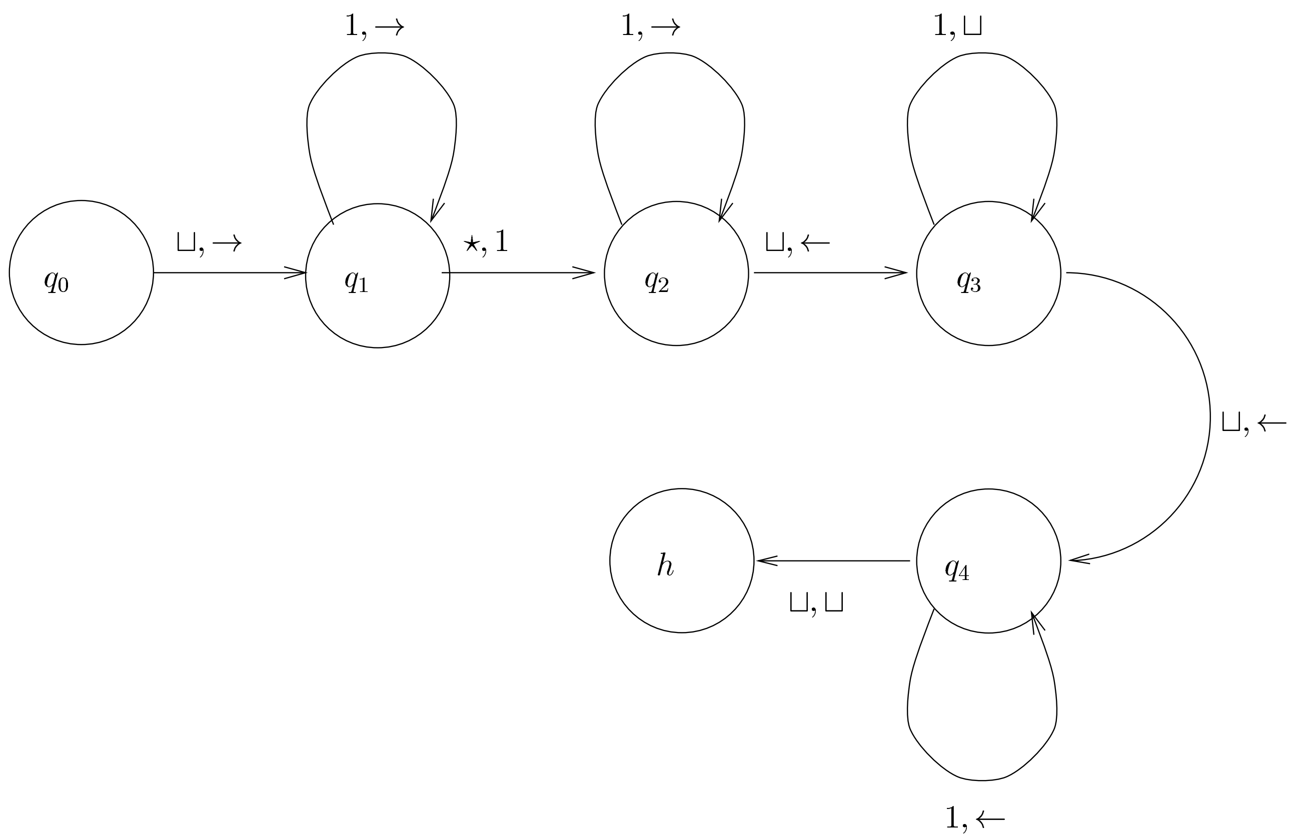
Example 1: Unary addition of 1

$\Sigma = \{1, \sqcup, \triangleright\}$
$\Sigma_I = \{1\}$
$Q = \{q_0, q_1, q_2, h\}$
$H = \{h\}$



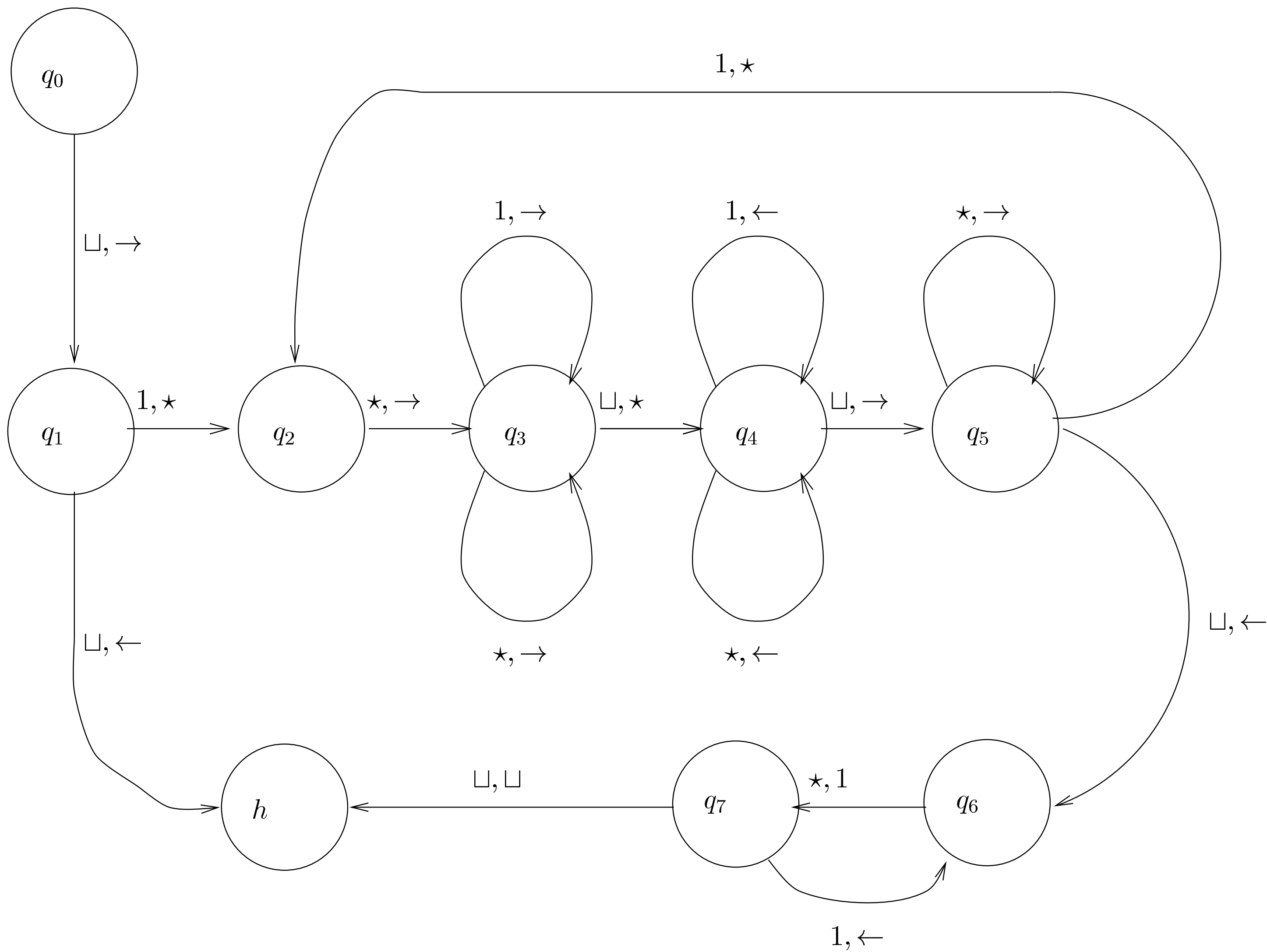
Example 2: General Unary Addition

$\Sigma = \{1, \star, \sqcup, \triangleright\}$
$\Sigma_I = \{1, \star\}$
$Q = \{q_0, q_1, q_2, q_3, q_4, h\}$
$H = \{h\}$



Example 3: Unary Multiplication by 2

$$\Sigma = \{1, \sqcup, \triangleright\}$$
$$\Sigma_I = \{1\}$$
$$Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, h\}$$
$$H = \{h\}$$



Decision Problems I: Recursive Languages

How do we go about using a TM to solve a decision problem?

- Assume we have represented the decision problem using a language over the alphabet Σ'.
- We let $\Sigma_I = \Sigma'$.
- We impose the condition that $H = \{Y, N\}$. Informally Y denotes 'yes' and N denotes 'no'.
- The TM *accepts* an input $x \in \Sigma_I^\star$ if the resulting computation halts in state Y. It *rejects* the input if the resulting computation halts in state N.
- The TM *decides* a language $L \subseteq \Sigma_I^\star$ if:
  1. If $x \in L$ then the TM accepts x.
  2. If $x \notin L$ then the TM rejects x.
- A language L is *recursive* if there is a TM that decides it.

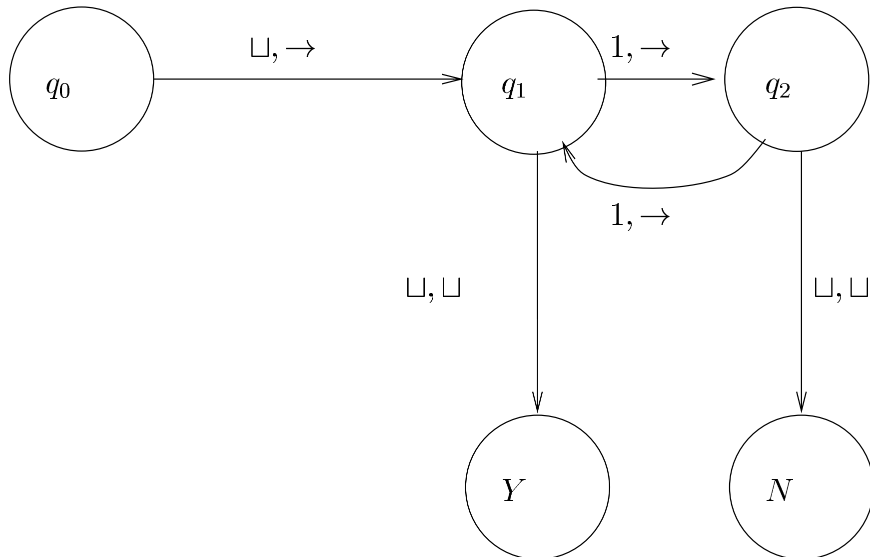
Decision Problems 2: Recursively Enumerable Languages

There is a somewhat more subtle way in which a TM can be associated with a language (decision problem).

- Assume we have represented a decision problem using a language over the alphabet Σ'.
- Let $\Sigma_I = \Sigma'$.
- A TM *semidecides* a language $L \subseteq \Sigma_I^\star$ if:
  1. If $x \in L$ then the TM halts.
  2. If $x \notin L$ then the TM fails to halt.
- A language L is *recursively enumerable (r.e.)* if there is a TM that semidecides it.

Example 1: Strings of even length

The language of even-length strings over the alphabet $\Sigma_I = \{1\}$ is recursive.



Example 2: Palindromes

- Consider the langauge representing palindromes over the alphabet $\{a, b\}$.
- For example, some members of the language are

$$aaabaaa$$

$$abababbababa$$

$$baaabbabbaaab$$

and so on.

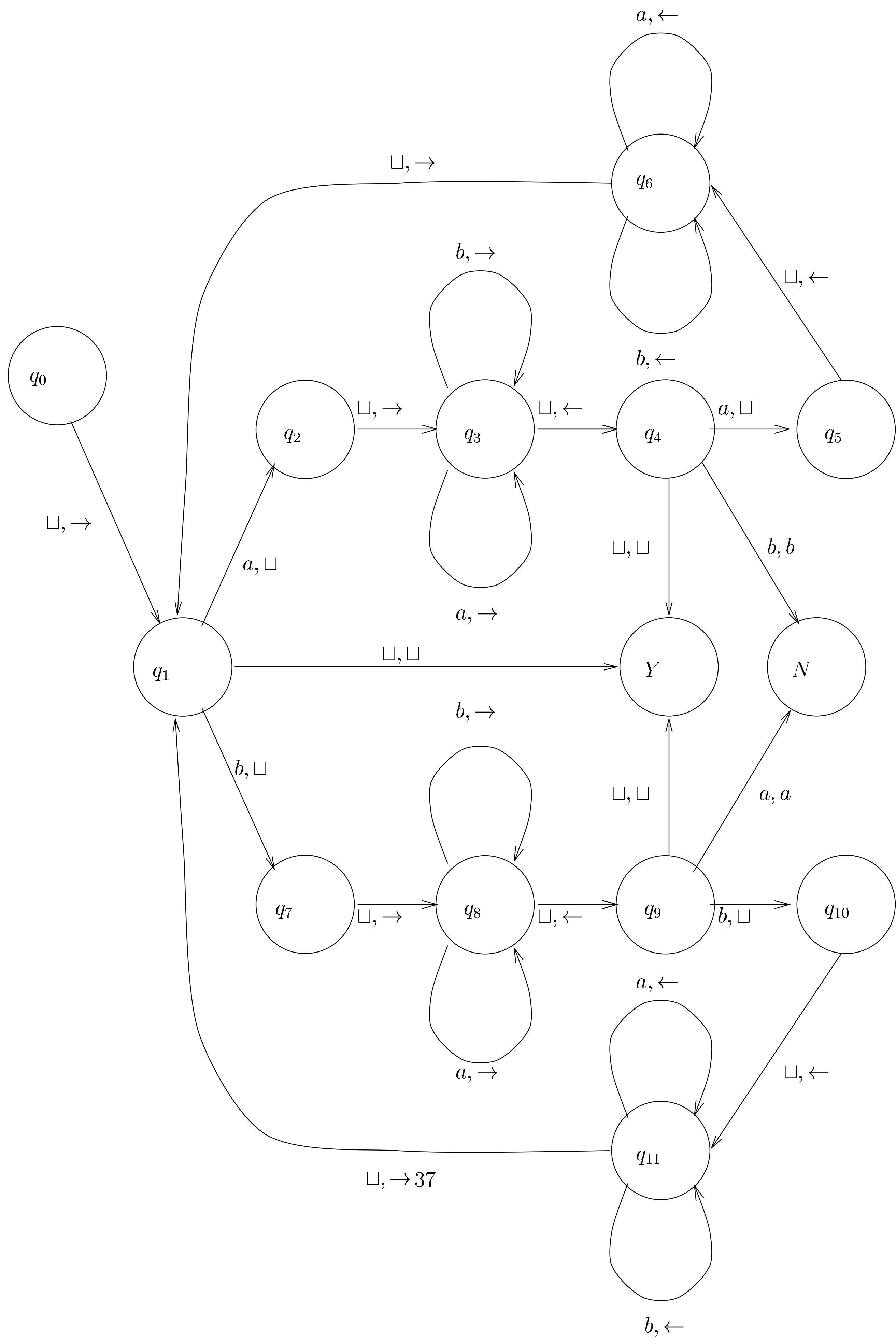
- Some non-members of the language are

$$abb$$

$$babaaaaaaaab$$

and so on.

- This langauge is also recursive...



$a, \leftarrow$
$\sqcup, \rightarrow$
q_6
$b, \rightarrow$
$\sqcup, \leftarrow$
$b, \leftarrow$
q_0
$\sqcup, \rightarrow$
q_2
$\sqcup, \leftarrow$
q_3
$a, \sqcup$
q_4
q_5
$\sqcup, \rightarrow$
$\sqcup, \sqcup$
b, b
$a, \sqcup$
$a, \rightarrow$
q_1
$\sqcup, \sqcup$
Y
N
$b, \rightarrow$
$b, \sqcup$
$\sqcup, \sqcup$
a, a
q_7
q_8
q_9
q_{10}
$\sqcup, \rightarrow$
$\sqcup, \leftarrow$
$b, \sqcup$
$a, \leftarrow$
$a, \rightarrow$
$\sqcup, \leftarrow$
q_{11}
$\sqcup, \rightarrow$
$b, \leftarrow$

3C04 - Complexity Theory

Lectures 1 to 6: Part B

Further Models of Computation

Copyright ©1997 to 2002, by Sean Holden

TMs with more than one tape

Do we get a more powerful model of computation by trying to expand the capabilities of the basic TM?

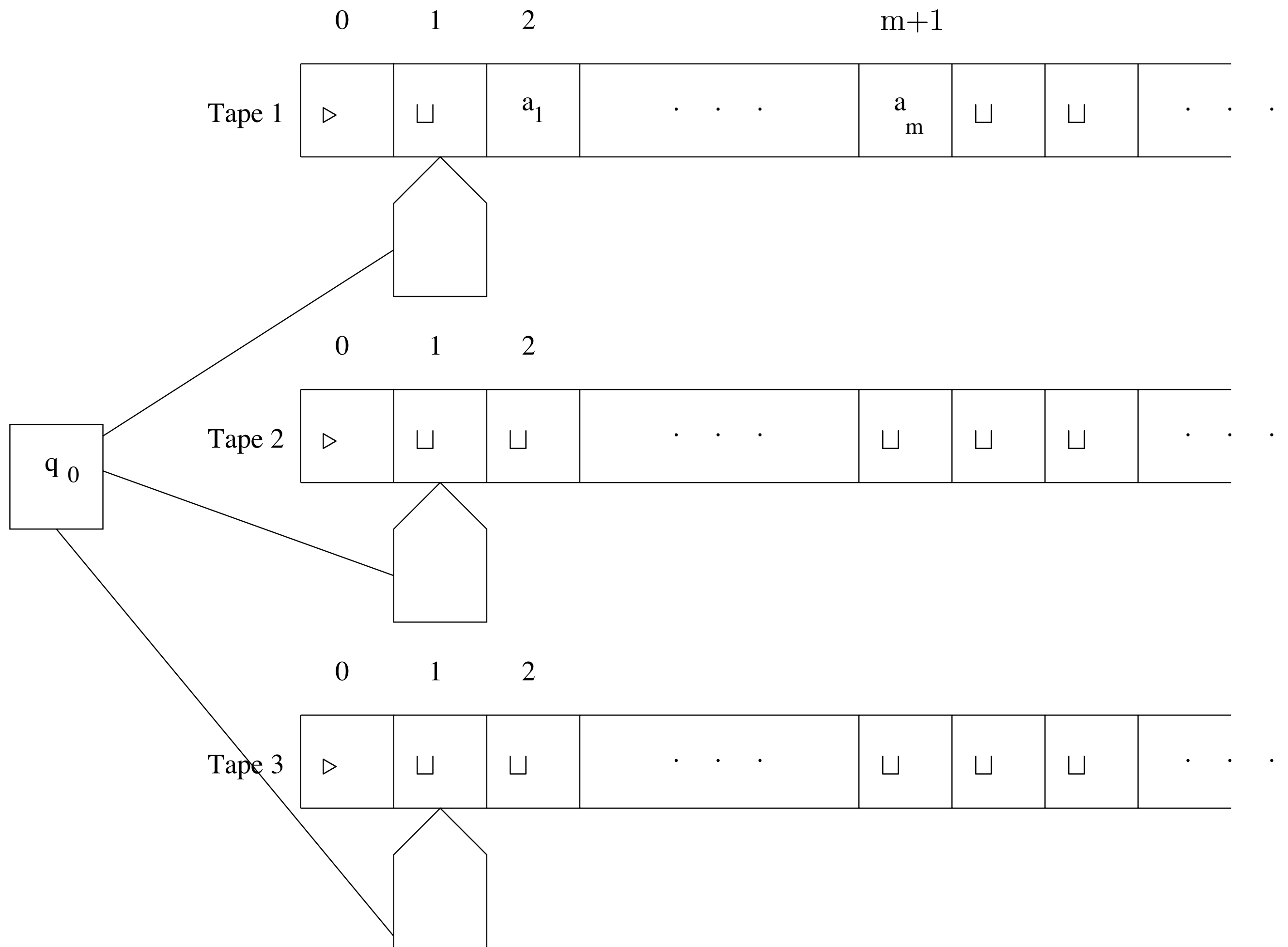
- A k-tape Turing machine (kTM) has k tapes of the kind used by a standard TM.
- Each tape has it's own head.
- The machine as a whole exists in a single state at any given time.
- Formally a kTM is a 5-tuple (Σ, Q, q_0, H, f) where,
  1. Σ, Q, q_0 and H are as previously defined.
  2. f is now a function
  $$f : (Q \backslash H) \times \Sigma^k \to Q \times (\Sigma \cup \{\leftarrow, \rightarrow\})^k.$$
  3. f has the usual conditions regarding the $\triangleright$ symbol: any head reading $\triangleright$ must move to the right; and no head ever writes $\triangleright$.

TMs with more than one tape

- Input to a kTM is placed on tape 1 in the same format as for a standard TM.
- Computation begins with the input on tape 1 and all other tapes blank. The heads are all at position 1.
- At each step, the machine reads the tapes at each head. It can then perform an independent operation at each head and move to a new state.
- Output is produced on tape 1 in the same format as previously, with the other tapes being ignored.
- When $k = 1$ we recover the standard TM.

TMs with more than one tape

For example, if $k = 3$ and the input is $a_1a_2 \ldots a_m$:



TMs with more than one tape

Theorem Let $M = (\Sigma, Q, q_0, H, f)$ be a kTM. Then there exists a standard TM $M' = (\Sigma', Q', q_0, H, f')$ with the following properties:

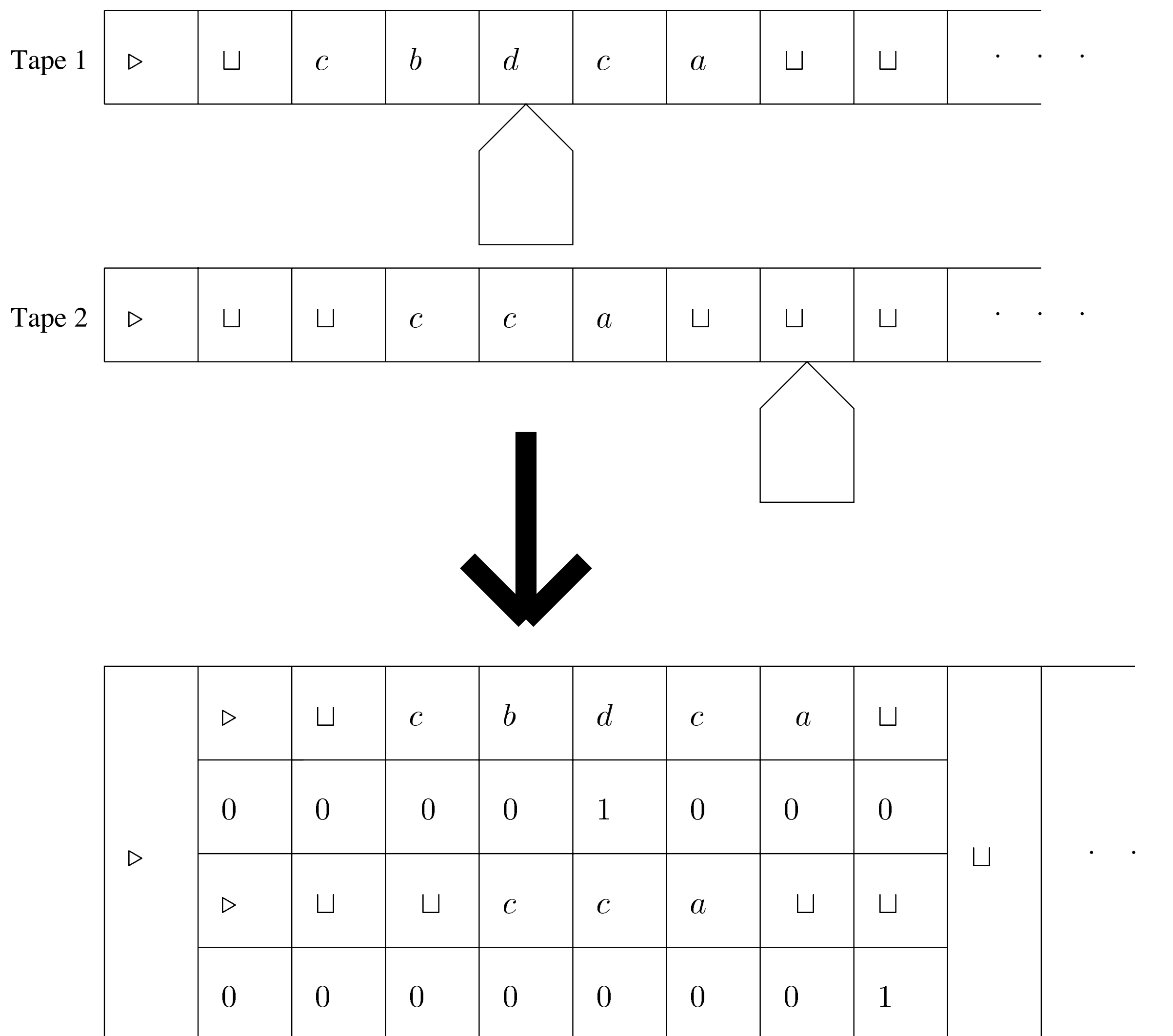
- $\Sigma \subseteq \Sigma'$.
- If M is started with input $x \in \Sigma^\star$ then:
  1. If M fails to halt then M' fails to halt when started with input x.
  2. If M halts with output y then M' halts in the same state and with the same output.

In other words, any kTM can be simulated by a standard TM.

(The converse is also obviously the case.)

TMs with more than one tape

How do we prove this? The central idea is to represent all k tapes *and* the positions of the heads on a single tape.



- Σ' then in effect contains new symbols which encode $2k$-tuples of symbols from $\Sigma \cup \{0, 1\}$.
- Formally,
$$\Sigma' = \Sigma \cup (\Sigma \times \{0, 1\})^k$$
- Σ is a subset of Σ' so that the kTM and the standard TM can take the same inputs and produce the same outputs.
- In the following, we will only outline the way in which the simulation can be achieved. Exercise: convince yourself that it's possible!

M' can simulate M as follows:

- **Stage 1:** It re-writes the initial input in encoded k-tape format as follows:
  1. It shifts the input string one space to the right.
  2. It moves to cell 1 and writes the symbol
  $$(\triangleright, 0, \triangleright, 0, \ldots, \triangleright, 0)$$
  3. It moves to cell 2 and writes the symbol
  $$(\sqcup, 1, \sqcup, 1, \ldots, \sqcup, 1)$$
  4. It moves one cell at a time to the right. At each step, of the tape contains a non-blank symbol a it replaces it with the symbol
  $$(a, 0, \sqcup, 0, \ldots, \sqcup, 0)$$
  5. The process stops when the head reaches the first $\sqcup$ symbol.

TMs with more than one tape

The initial state of the k tapes has been encoded on a single tape. We now need to simulate the k tape computation one step at a time. At the start of the simulation of each step the head of the standard TM is at the first 'real' blank symbol - that is, the first cell of the tape that does not yet encode k tapes:

- **Stage 2:** We need to find which symbol each of the k heads is scanning:
  1. Move the head of the standard TM leftward along the tape.
  2. At each cell, we can change the *state* of the standard TM to reflect the contents of the current cell, each of which encodes k tapes, and possibly tells us a head position.
  3. Once the head of the standard TM reaches the $(\triangleright, 0, \triangleright, 0, \ldots, \triangleright, 0)$ symbol, it moves back to the first 'real' blank.
  4. At this point the *state* of the standard TM has changed to reflect the symbols being scanned by the heads of the kTM.
- Nothing is written on the tape during this stage.

We now need to update the tape according to the current state of the kTM being simulated and the symbols being scanned by its heads.

- **Stage 3:**

    1. Move the head of the standard TM left then right along the tape.
    2. At each cell where a head of the kTM is at present update the tape to reflect the action of that head.

This overall process is iterated to simulate the computation of the kTM. If at any point this computation halts we need to tidy up:

- **Stage 4:** If the computation halts, the standard TM moves back along its tape converting to single tape format:

    1. Only the contents of tape 1 of the kTM are retained.
    2. The standard TM then places its head where head 1 of the kTM would be.
    3. Finally, the standard TM halts in the same state as the kTM would halt in.

Other 'more powerful' TMs

It turns out that any attempt to increase the power of a TM fails in this way.

1. **Two-way infinite tapes:** We can allow the TM tape to be infinite in both directions. Such a machine can be simulated by a standard TM:

   - Start with a kTM where $k = 2$.
   - Tape 1 contains the part of the two-way infinite tape from the beginning of the input to the right.
   - Tape 2 represents the remainder of the two-way infinite tape, but in reverse.
   - This machine can simulate the two-way infinite tape TM.
   - A standard TM can simulate the two-tape TM.

Other ‘more powerful’ TMs

2. **Multiple head TMs:** We could attempt to increase the power of a standard TM by placing multiple heads on the single available tape.

   - At each step of the computation each head reads a symbol and then either writes or moves independently.
   - (We need a way of dealing with what happens when two heads try to write in the same place.)

   Such a machine can again be simulated by a standard TM:

   - Assume there are h heads.
   - As with the simulation of a kTM, the single tape can be divided into a $h + 1$ tracks, h of which represent the head positions and one of which represents the contents of the tape.
   - The details are similar to those for the simulation of a kTM.

Other ‘more powerful’ TMs

In fact, any attempt to increase the power of the basic TM leads to the same conclusion: the augmented TM can be simulated by a standard TM:

- Multiple, two-way infinite tapes.
- TMs operating on a two-dimensional grid.
- and so on...

Unlimited Register Machines

TMs seem (on the surface) far removed from familiar computer hardware. Unlimited register machines (URMs) provide an equivalent model of computation that looks similar to a RISC processor.

- A URM has an infinite number of registers, numbered $1, 2, 3, \ldots$. Register n is denoted R_n.
- Each register can contain a natural number $n \in \mathbb{N}$. The number stored in R_n is denoted r_n.
- URMs compute functions from $\mathbb{N}^k$ to $\mathbb{N}$.
- The input $x \in \mathbb{N}^k$ to a URM is written in registers 1 to k in the obvious way. All other registers initially contain 0.
- If a computation halts (and as usual it may or may not) then the output is the value r_1.

Unlimited Register Machines

- A computation proceeds according to a program P. URM programs look similar to assembler code.
- P contains s instruction. $I_1, I_2, \ldots, I_s$. There are four types of instruction.
  1. **Zero:** An instruction $Z(n)$ sets r_n to be equal to 0.
  2. **Successor:** An instruction $S(n)$ adds 1 to the contents of R_n.
  3. **Move:** An instruction $M(n, m)$ sets the contents of R_m to be equal to r_n. No other registers are affected.
  4. **Jump:** An instruction $J(n, m, p)$ has the following effect: if $r_n = r_m$ move to instruction p, otherwise continue.

Unlimited Register Machines

- A computation begins with instruction I_1.
- At each step, if we have executed instruction I_i and it is not a jump instruction then we move to instruction I_{i+1}.
- At each step, if we have executed instruction I_i and it is a jump instruction $J(n, m, p)$, we move to instruction I_p if $r_n = r_m$ and instruction I_{i+1} otherwise.
- A computation halts if we move beyond instruction I_s.

Unlimited Register Machines

Example 1: adding two numbers.

$I_1 : J(2,3,6)$
$I_2 : S(1)$
$I_3 : S(3)$
$I_4 : J(2,3,6)$
$I_5 : J(1,1,2)$

Example 2: multiplying two numbers.

$I_1 : J(1,3,16)$
$I_2 : J(2,3,16)$
$I_3 : S(3)$
$I_4 : J(2,3,17)$
$I_5 : M(1,3)$
$I_6 : S(5)$
$I_7 : Z(4)$
$I_8 : S(1)$
$I_9 : S(4)$
$I_{10} : J(3,4,12)$
$I_{11} : J(1,1,8)$
$I_{12} : Z(4)$
$I_{13} : S(5)$
$I_{14} : J(2,5,17)$
$I_{15} : J(1,1,8)$
$I_{16} : Z(1)$

Unlimited Register Machines

- **Definition:** A *partial* function is one whose value is not defined for all possible arguments.
- Both TMs and URMs may compute partial functions on $\mathbb{N}^k$ as for some inputs they may fail to halt.
- **Theorem:** A (possibly partial) function $f : \mathbb{N}^k \to \mathbb{N}$ is computatble by a URM if and only if it is computable by a standard TM.

3C04 - Complexity Theory

Lectures 1 to 6: Part C

Church's thesis, encoding TMs and the Universal TM

Copyright ©1997 to 2002, by Sean Holden

Church's Thesis

You will be able to find many different statements of *Church's Thesis*—sometimes referred to as the *Church-Turing Thesis*:

> *Any problem that can be solved by a well-defined step-by-step procedure can be solved by a Turing machine.*

- Although this assertion is not susceptible to absolute *proof*, it could be disproved by offering a suitable counterexample.
- No such counterexample has been found.
- There is a huge body of evidence that supports the thesis. For example, the class of functions that you can compute using a TM is *exactly the same* as the class of functions computed by:
  1. TMs with multiple tapes.
  2. TMs with a tape infinite in both directions.
  3. Recursive functions.
  4. Unlimited register machines.
  5. and others ...

Church's Thesis

Why is Church's thesis important? In a slightly more symbolic form it says:

(Solvable by a well-defined step-by-step procedure)
$\rightarrow$ (Solvable by a TM)

which is the same as saying:

$\neg$(Solvable by a TM)
$\rightarrow$ $\neg$(Solvable by a well-defined step-by-step procedure)

Conclusion: Any problem that can *not* be solved using a TM is of significant practical importance:

- You can not solve it using any well-defined step-by-step procedure.
- In particular, if we accept Church's Thesis, it is not solvable by a program in C++, Java, or anything else!

Of course, this assertion depends on Church's Thesis, for which we have ample evidence.

Encoding Turing Machines

The further development of the theory is based on the ability to encode a TM, so that we can address decision problems concerning TMs.

Let's start by taking a step back, for the moment, from TMs and thinking about some of the things it's possible to do in Java or any other high-level language.

1. Imagine you've written a text editor. You can use this to edit it's own source code.
2. You could in fact write a program in Java that executes other programs written in Java.
3. There is nothing to stop you from using this program to simulate *itself*. You'd then have a program in Java simulating another program in Java, the latter being capable of simulating any program in Java.

So far, all the TMs examined are fixed—we've designed them for only one purpose.

It's quite possible to write a Java program that simulates *any* TM. Church's thesis therefore tells us that we can produce a TM that can simulate *any* TM. (Including itself!) This is the *Universal Turing machine*.

Encoding TMs and their Inputs

To allow a TM to use another TM as part of its input, we need a way of encoding TMs.

- There are many ways of doing this—we will look at only one.
- *Any* TM, and any input to a TM, can be encoded as a string over the alphabet $\{0, 1\}$.
- The encoding procedure will be presented in several stages.

Encoding TMs and their Inputs

- We can standardise the symbols used to represent states and the contents of the tape by introducing the infinite sets,

$$\begin{aligned}\hat{Q} &= \{q_0, q_1, q_2, \ldots\} \\ \hat{\Sigma} &= \{\sigma_0, \sigma_1, \sigma_2, \ldots\}\end{aligned}$$

  and insisting that for any TM, $\Sigma \subseteq \hat{\Sigma}$ and $Q \subseteq \hat{Q}$.

- By convention, we insist that,

$$\sigma_0 = \sqcup$$

$$\sigma_1 = \triangleright$$

$$\sigma_2 = \rightarrow$$

$$\sigma_3 = \leftarrow$$

  and,

$$\text{start state} = q_0.$$

  (We effectively treat $\rightarrow$ and $\leftarrow$ as special tape symbols, although they will never appear on the tape.)

Encoding TMs and their Inputs

- Each of the symbols used to specify a TM can be encoded as a string of 1's as follows:
  - code$(q_i) = \underbrace{11\ldots1}_{i+1, 1\text{'s}}$
  - code$(\sigma_i) = \underbrace{11\ldots1}_{i+1, 1\text{'s}}$
- To specify a TM all that remains is to specify its *halt states* and its *program*. Each tuple in the program can be written,

$$(q_{i_1}, \sigma_{i_2}, q_{i_3}, \sigma_{i_4})$$

  where the final σ_{i_4} can specify $\rightarrow$ or $\leftarrow$ in the obvious way.
- Each tuple t is encoded by the string,

  code(t)
  $= \text{code}(q_{i_1})0\text{code}(\sigma_{i_2})0\text{code}(q_{i_3})0\text{code}(\sigma_{i_4})0$

- Halt states can be inferred, as they never appear on the left-hand half of any tuple in the program.

Encoding TMs and their Inputs

- Finally, if a TM has a program consisting of the tuples $t_1, t_2, \ldots, t_T$ then we encode it using the string,

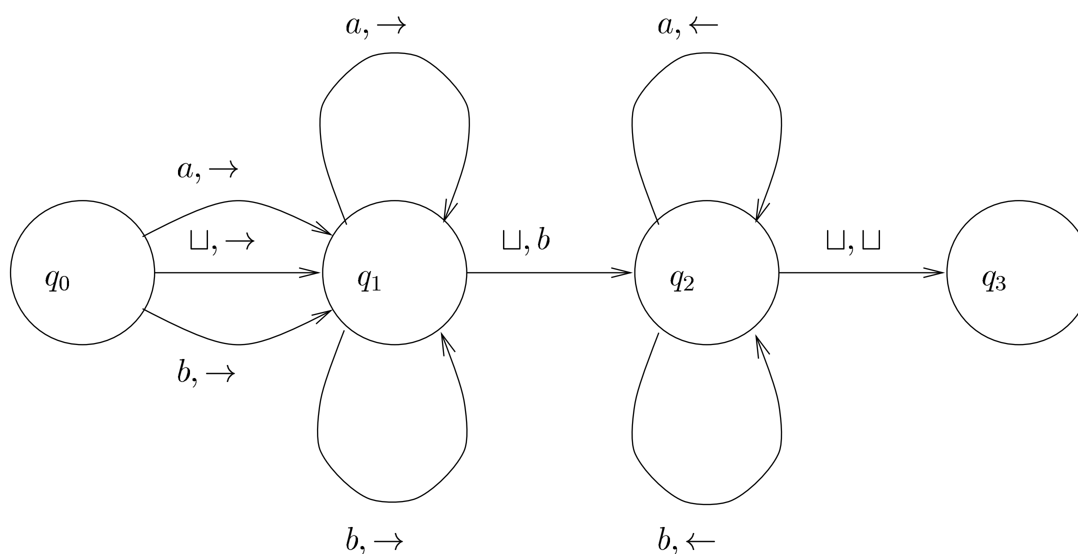
$$\mathsf{code}(t_1)0\mathsf{code}(t_2)0\cdots0\mathsf{code}(t_T)0$$

- Inputs to TMs can also be encoded in this manner. The input $\sigma_{i_1}\sigma_{i_2}\cdots\sigma_{i_N}$ is encoded as the string,

$$00\mathsf{code}(\sigma_{i_1})0\mathsf{code}(\sigma_{i_2})0\cdots0\mathsf{code}(\sigma_{i_N})0$$

Example

Here is a TM with $\Sigma = \{a, b, \sqcup, \triangleright\}$, $Q = \{q_0, q_1, q_2, q_3\}$, and $H = \{q_3\}$. It appends a b to its input string. Its program is as follows:



(Actions to be taken when scanning $\triangleright$ are not shown on the diagram.)

We first alter the TM to use the conventions introduced earlier. So now Σ and Q are,

$$\Sigma = \{\sigma_0, \sigma_1, \sigma_4, \sigma_5\}$$
$$Q = \{q_0, q_1, q_2, q_3\}$$

Example

Writing the program as a set of tuples we have,

$$
\begin{aligned}
t_1 &= (q_0, \sigma_0, q_1, \sigma_2) \\
t_2 &= (q_0, \sigma_1, q_0, \sigma_2) \\
t_3 &= (q_0, \sigma_4, q_1, \sigma_2) \\
t_4 &= (q_0, \sigma_5, q_1, \sigma_2) \\
t_5 &= (q_1, \sigma_0, q_2, \sigma_5) \\
t_6 &= (q_1, \sigma_1, q_1, \sigma_2) \\
t_7 &= (q_1, \sigma_4, q_1, \sigma_2) \\
t_8 &= (q_1, \sigma_5, q_1, \sigma_2) \\
t_9 &= (q_2, \sigma_0, q_3, \sigma_0) \\
t_{10} &= (q_2, \sigma_1, q_2, \sigma_2) \\
t_{11} &= (q_2, \sigma_4, q_2, \sigma_3) \\
t_{12} &= (q_2, \sigma_5, q_2, \sigma_3)
\end{aligned}
$$

Example

The tuples are coded as:

$$
\begin{aligned}
\mathsf{code}(t_1) &= 10101101110 \\
\mathsf{code}(t_2) &= 10110101110 \\
\mathsf{code}(t_3) &= 101111101101110 \\
\mathsf{code}(t_4) &= 1011111101101110 \\
\mathsf{code}(t_5) &= 1101011101111110 \\
\mathsf{code}(t_6) &= 1101101101110 \\
\mathsf{code}(t_7) &= 1101111101101110 \\
\mathsf{code}(t_8) &= 11011111101101110 \\
\mathsf{code}(t_9) &= 1110101111010 \\
\mathsf{code}(t_{10}) &= 111011011101110 \\
\mathsf{code}(t_{11}) &= 1110111110111011110 \\
\mathsf{code}(t_{12}) &= 11101111110111011110
\end{aligned}
$$

Finally, the TM is coded as,

10101101110010110101110010111110110
11100101111110110111001101011101111
11001101101101110011011111011011100
11011111101101110011101011110100111
01101110111001110111110111011110011
1011111101110111100

Some things to note

There are some things that should be noted regarding this technique.

1. By using different *states*, it is possible to have two or more machines that do the same job, and these can have different codes.

2. The coding is 1 to 1. That is, two *different* machines will have *different* codes.

3. Given a string over $\{0, 1\}$, it's possible to tell whether or not this string is the code for some TM. In fact, we can produce a TM that takes a string over $\{0, 1\}$ and tells us whether it is such a code.

Universal Turing Machines

There exists a TM with input alphabet $\{0, 1\}$ that can simulate the action of *any* TM. This is a *Universal Turing Machine* (UTM), which we will call M_U. Let M be any TM and let I be an input string for M.

M_U, when given the input code(M)code(I), simulates the action of M for input I as follows:

1. if M fails to halt for input I then M_U fails to halt for input code(M)code(I).
2. if M halts for input I and produces output O then M_U halts for input code(M)code(I) producing output code(O).

Universal Turing Machines

The UTM operates as follows. (Exercise: design one!)

- The UTM is a three-tape TM (which, as we know, can be simulated by a single-tape TM). In general:
  1. The first tape contains the *encoded form* of the tape of M. Head 1 is at the start of the encoded form of the symbol currently scanned by M.
  2. The second tape contains a copy of code(M).
  3. The third tape represents the *encoded form* of the state of M.
- To start a simulation the UTM:
  1. Copies code(M) from tape 1 to tape 2.
  2. Removes code(M) from tape 1.
  3. Shifts code(I) on tape 1 to the left and preceeds it with code$(\triangleright)0$code$(\sqcup)0$.
  4. Writes code(q_0) on tape 3.
  5. Places head 1 at encoded $\sqcup$ before code(I), head 2 at the start of code(M), and head three at the start of code(q_0).

Universal Turing Machines

To simulate one step of the computation of M the UTM then proceeds as follows:

- It searches in code(M) on tape 2 for a tuple where the first element matches the state stored on tape 3 and the second part matches the symbol currently scanned by M.
- It uses the fourth element of the tuple to update tape 1 and the position (if necessary) of head 1 to reflect the action to be performed on the tape of M.
- It uses the third element of the tuple to write the new state that M will move to on tape 3; or, if M has reached a halt state, it halts.

3C04 - Complexity Theory

Lectures 7 to 9

In the last few lectures we looked at decision problems, languages, and Turing machines. We also looked at the way in which you can encode Turing machines and their inputs, and the fact that there is a universal Turing machine.

In the next three lectures we demonstrate that there are many more possible problems than there are Turing machines to solve them. The conclusion we can draw from this is that the vast majority of possible problems can not be solved by a Turing machine, and hence by any existing computer!

Aims:

1. To introduce the concept of an unsolvable problem.
2. To review the concept of a *countably infinite set*, and of an *uncountable set*.
3. To review the method of *diagonalization* for proving that certain sets are uncountable.
4. To apply these concepts to Turing machines, and decision problems.
5. To begin the study of specific unsolvable problems.

Copyright ©1997 to 2002, by Sean Holden

Solvable and Unsolvable Problems

We've seen that a *decision problem* can be transformed into a corresponding *language*. We'll therefore concentrate, for the time being, on languages. Let L be a language on the alphabet Σ_I. Recall the earlier definition:

Definition:

- The set of halting states is $H = \{Y, N\}$. Informally Y denotes 'yes' and N denotes 'no'.
- The TM *accepts* an input $x \in \Sigma_I^\star$ if the resulting computation halts in state Y. It *rejects* the input if the resulting computation halts in state N.
- The TM *decides* a language $L \subseteq \Sigma_I^\star$ if:
  1. If $x \in L$ then the TM accepts x.
  2. If $x \notin L$ then the TM rejects x.
- A language L is *recursive* if there is a TM that decides it.

Suppose L is the language corresponding to a decision problem. We say that the problem is *solvable* if L is recursive. The reason for this should be clear: there is a Turing machine, namely M, that tells us whether or not any instance is a yes-instance. If there is *no* TM that decides L then we say that the problem is *unsolvable*.

Solvable and Unsolvable Problems

A couple of questions should present themselves:

1. Are there any unsolvable problems? If so, what are they like?
2. Are there any problems that are 'partly' solvable in some sense?

Earlier, we introduced a further definition.

Definition:

- A TM *semidecides* a language $L \subseteq \Sigma_I^\star$ if:
    1. If $x \in L$ then the TM halts.
    2. If $x \notin L$ then the TM fails to halt.
- A language L is *recursively enumerable (r.e.)* if there is a TM that semidecides it.

We will abbreviate 'recursively enumerable' to 'r.e.'.

Solvable and Unsolvable Problems

Are there any languages that are r.e. but not recursive?

If L is r.e. but not recursive then the problem having language L is 'partly' solvable, because,

1. If an input x is in L, the TM will eventually tell us this.
2. For an input x that is *not* in L, the TM will never halt. Therefore, we'll never find out whether x is in L or not.

Solvable and Unsolvable Problems

Theorem 1: If L is recursive then it is r.e.

Proof As L is recursive there is a TM, M that decides L. Modify M as follows:

- Convert state N from a halt state to an extra, standard state.
- Add to the program for M such that whenever this state is entered the machine enters a loop.
- (Take care to deal with the possibility that M ends with its head scanning $\triangleright$.)

What we don't yet know is whether or not the converse is true...

Where are we heading?

The rationale for the next step is simple:

- Assume a Turing Machine is used to semidecide a language.
- Our aim is to show that there are many more languages than there are TMs. Conclusion: most languages are not r.e. (and consequently, not recursive).
- Consequently, most decision problems are not solvable!

Later in the course we will also see that there exist languages that are r.e. but not recursive, and we will also explore a somewhat more useful way of proving that particular decision problems are unsolvable.

Countable Sets

A set S is countably infinite if we can write down a function $f : \mathbb{N} \to S$ such that:

1. f maps each natural number to a member of S.
2. Each member of S is a value $f(x)$ for some x.

$\mathbb{N}$ is therefore countably infinite as we can use the function $f(x) = x$.

The set of odd natural numbers,

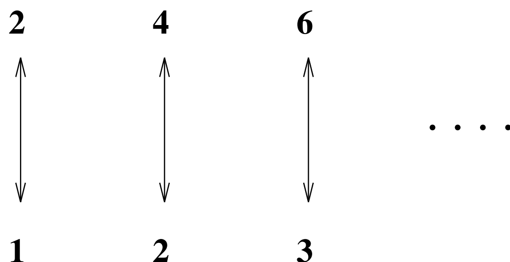
$$\mathbb{O} = \{1, 3, 5, \ldots\}$$

is therefore countably infinite as we can use the function $f(x) = 2x - 1$.

Also, the set of even natural numbers,

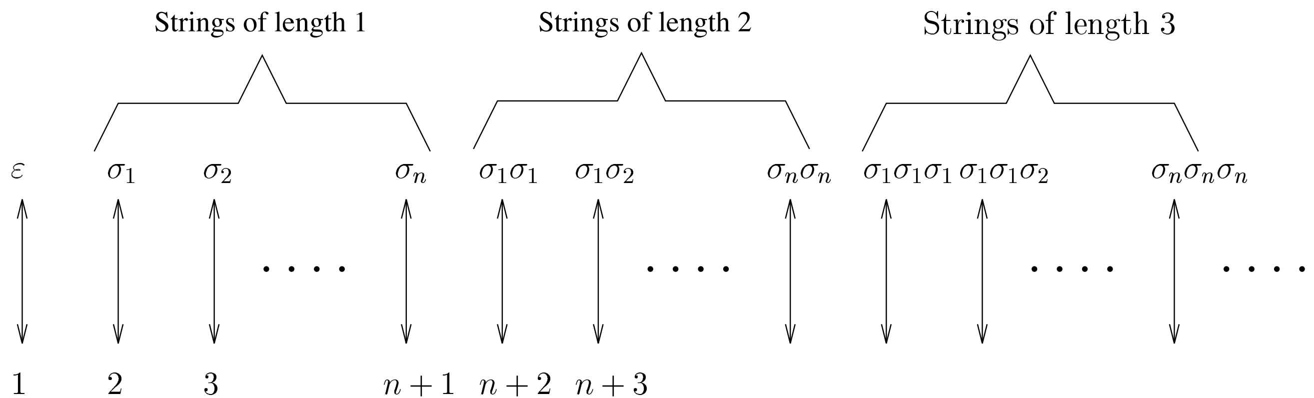
$$\mathbb{E} = \{2, 4, 6, \ldots\}$$

is countably infinite as we can use the function $f(x) = 2x$.

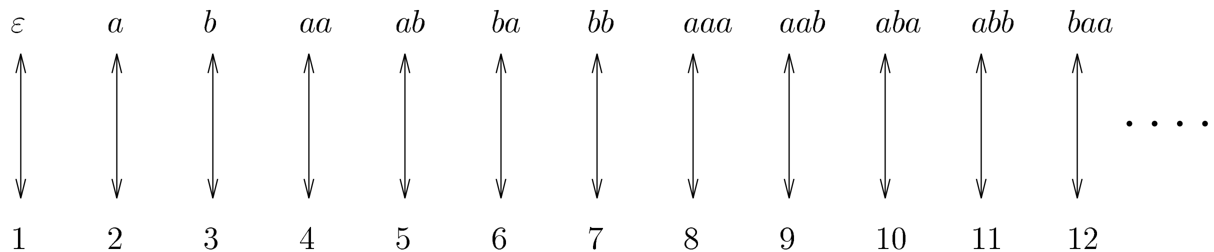


Countable Sets

Another example of a countable set is the set of all finite-length strings for a finite alphabet Σ. To see this, assume $|\Sigma| = n$.



Example: $\Sigma = \{a, b\}$.



Exercise: What happens if Σ is countably infinite?

How many Turing Machines are there?

We've already seen that *any* Turing machine can be encoded as a string of finite length for an alphabet Σ with $|\Sigma| = 2$.

Some important conclusions can now be drawn from this fact:

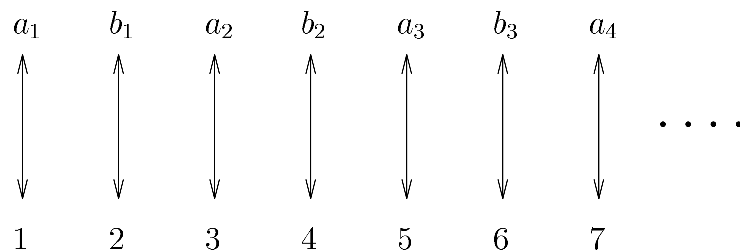
1. The set of all Turing machines is countably infinite.
2. For any finite alphabet Σ the set of r.e. languages over Σ is countably infinite. To see this, just note that if a language is r.e. then there's a Turing machine that semidecides it.
3. The set of computable functions from $\mathbb{N}$ to $\mathbb{N}$ is countably infinite, using the same argument.

Countable Sets

Let's imagine we have two countably infinite sets A and B. As they are both countably infinite we can write them as follows:

$$A = \{a_1, a_2, a_3, \dots\}$$
$$B = \{b_1, b_2, b_3, \dots\}$$

Is the set $A \cup B$ countably infinite? The answer is yes:



This will be needed later so let's state it as a theorem:

Theorem 2: *If A and B are countably infinite sets then $A \cup B$ is a countably infinite set.*

Uncountable Sets

We now need to consider the set of all *languages* for a finite alphabet Σ, which we will call S_Σ.

Is S_Σ countable?

Recall that a language L is a subset of $\Sigma^\star$. We've already seen that $\Sigma^\star$ itself must be countably infinite, so we can write it as,

$$\Sigma^\star = \{\sigma_1, \sigma_2, \sigma_3, \ldots\}$$

Let's say that we have a language,

$$L_1 = \{\sigma_1, \sigma_4\}$$

Then L_1 can be represented like this:

	σ_1	σ_2	σ_3	σ_4	σ_5	. . . .
L_1	1	0	0	1	0	. . . .

Uncountable Sets

Any language can be represented in this way, so if S_Σ is countably infinite it can be represented like this:

	σ_1	σ_2	σ_3	σ_4	σ_5	$\cdots$
L_1	(1)	0	0	1	0	$\cdots$
L_2	0	(1)	1	0	1	$\cdots$
L_3	0	0	(0)	0	0	$\cdots$
L_4	1	1	1	(0)	1	$\cdots$
L_5	1	1	1	1	(1)	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

so that any $L \in S_\Sigma$ appears as a row. However, we can use the diagonalization technique to construct the language L' defined as follows:

- If $\sigma_i \in L_i$ then $\sigma_i \notin L'$
- If $\sigma_i \notin L_i$ then $\sigma_i \in L'$

If L' appears in the table then $L' = L_n$ for some n. Then we would have,

$$\sigma_n \in L_n \rightarrow \sigma_n \notin L' = L_n$$

$$\sigma_n \notin L_n \rightarrow \sigma_n \in L' = L_n$$

The language L' clearly does not appear in the table. Therefore S_Σ is *not* countably infinite.

What can we conclude?

We've now seen that:

1. For any finite alphabet Σ the set of r.e. languages over Σ is countably infinite.
2. For any finite alphabet Σ, the set of all languages over Σ is uncountable.

We can therefore conclude that there exist languages that are not r.e.

Removing a countably infinite subset from an uncountable set

We saw earlier that if we take two countably infinite sets A and B, then $A \cup B$ is also countably infinite (theorem 2).

We can now show the following.

Theorem 3: If S is an infinite set that is *not* countably infinite, and S' is a countably infinite subset of S, then $S \backslash S'$ is *not* countably infinite.

Proof Assume $S \backslash S'$ is countably infinite. Then by theorem 2 $S' \cup (S \backslash S')$ must be countably infinite. But $S' \cup (S \backslash S') = S$ - a contradiction!

In other words, you can remove a countably infinite number of things from an an uncountable set and what remains will still be uncountable! The conclusion we can draw from this is that there are more languages that are not r.e. than languages that are.

Where do we go next?

Next, we're going to look at the following questions:

1. What are the basic properties of recursive and r.e. languages.
2. Are there languages that are r.e. but not recursive?
3. Can we find a specific language that is not r.e.?

The answer to questions 2 and 3 is, in both cases, yes. To demonstrate this we have to start with question 1.

Theorem 4: If L is recursive then so is $\overline{L}$. Recall that $\overline{L}$ is the language,

$$\overline{L} = \{x \in \Sigma^\star : x \notin L\}$$

Proof As L is recursive there is a TM, M that decides L. Amend M such that, instead of halting in state Y it halts in state N and vice versa.

Properties of recursive and r.e. languages

Theorem 5: If L is r.e. and $\overline{L}$ is r.e. then L is recursive.

Proof

As L and $\overline{L}$ are both r.e. there are TMs, M_1 and M_2 that semidecide L and $\overline{L}$ respectively.

Construct a 3-tape TM, M as follows.

1. M copies its input from tape 1 to tapes 2 and 3 and simulates M_1 on tape 2 and M_2 on tape 3.
2. If at any point M_1 halts then M halts in state Y.
3. If at any point M_2 halts then M halts in state N.

Properties of recursive and r.e. languages

Theorem 6:

1. If L_1 and L_2 are r.e. then $L_1 \cup L_2$ and $L_1 \cap L_2$ are r.e.
2. If L_1 and L_2 are recursive then $L_1 \cup L_2$ and $L_1 \cap L_2$ are recursive.

Proof We'll look at the proof that L_1, L_2 recursive implies $L_1 \cup L_2$ recursive. The other proofs are similar.

- As L_1 and L_2 are recursive there are TMs, M_1 and M_2 that decide L_1 and L_2 respectively.
- Construct a 3-tape TM, M that copies its input from tape 1 to tapes 2 and 3 and simulates M_1 on tape 2 and M_2 on tape 3.
- If M_1 and M_2 both halt in state N then M halts in state N, otherwise M halts in state Y.

Two Specific Languages

We've seen that TMs and their inputs can be encoded using the alphabet $\Sigma = \{0, 1\}$. We'll use this fact to exhibit the languages NSA (not self accepting) and SA (self accepting) on $\Sigma = \{0, 1\}$ which have the following properties:

1. NSA is not r.e.
2. SA is r.e. but not recursive.

Definition

1. NSA is the language defined as follows:

   $NSA = \{x \in \Sigma^\star \ : \ x \neq \text{code}(M)$ for *any* TM, M or,
   $x = \text{code}(M)$ and M does *not*
   halt for input $x\}$.

2. SA is the language,

   $$SA = \{x \in \Sigma^\star : x \notin NSA\}$$

   or equivalently,

   $SA = \{x \in T^\star : x = \text{code}(M)$ and M halts for input $x\}$.

Theorem 7: NSA is not r.e.

Proof Assume that NSA is r.e. This means that there is a TM, M that semidecides NSA. We show that this leads to a contradiction.

First, note that as M is assumed to halt for inputs from NSA, it can take strings of 0s and 1s as inputs. Also, it will itself have the code $\mathbb{M} = \text{code}(M)$. There are two possibilities:

1. If M halts for input $\mathbb{M}$, then $\mathbb{M} \notin$ NSA. But we *assumed* above that M semidecides NSA, so if M halts for input $\mathbb{M}$ then it must be the case that $\mathbb{M} \in$ NSA. This is obviously a contradiction.

2. If M does not halt for input $\mathbb{M}$, then $\mathbb{M} \in$ NSA. But we assumed above that M semidecides NSA, so if M does not halt for input $\mathbb{M}$ then $\mathbb{M} \notin$ NSA. Again, a contradiction.

We have therefore shown, by reductio ad absurdum, that NSA is not r.e.

SA is r.e. but not recursive

Theorem 8: SA is not recursive.

Proof Let L be the language SA. Assume that L is recursive. From theorem 4 we know that, if this is the case, then $\overline{L}$ is also recursive. Also, from theorem 1 we know that if $\overline{L}$ is recursive then it is r.e.

Now simply note that $\overline{L}$ is the language NSA. We have just proved that NSA (hence $\overline{L}$) is *not* r.e.—a contradiction.

SA is r.e. but not recursive

Theorem 9: SA is r.e.

Proof (sketch) To see that this is the case we simply outline the design of a TM, M that semidecides SA. Given an input $x \in \{0,1\}^\star$, M checks to see whether x is the description of a TM, then,

- If x is *not* a description of a TM, then M enters an infinite loop.
- If x is the description of a TM then
  1. M turns it into $x$code(x) and uses the UTM to simulate the operation of the machine with code x for the input x.
  2. If the UTM indicates that the machine with code x halts for input x, then M halts.
  3. If the machine with code x does not halt for input x then the UTM simulation does not halt, and so then M does not halt.

3C04 - Complexity Theory

Lecture 10 and 11

The Halting problem and Reduction

We've now seen that there must be many languages that are not r.e. We've also seen a specific example of such a language, and an example of a language that is r.e. but not recursive.

We now introduce the Halting problem as a further example of a decision problem that is unsolvable, and we introduce a more useful method for proving that decision problems are unsolvable. Finally, we use this method to show that some further decision problems involving TM's are unsolvable.

Aims:

- To introduce the Halting problem and to prove that it is unsolvable.
- To introduce the technique of *reduction*, and to give some examples of how it can be used.

Copyright ©1997 to 2002, by Sean Holden

The Halting Problem - Version I

Consider the following problem:

Given a TM, M, and an input I, does M halt for input I?

Slightly more formally, is the language,

$$H = \{\text{code}(M)\text{code}(I) : M \text{ halts for } I\}$$

recursive? We can prove that it is not by arguing that, if it were, then the language SA would also be recursive—as the latter statement is false we conclude that H is not recursive.

This is basically an example of a technique called *reduction*, which will take a central rôle in the remainder of the course.

The Halting Problem - Version I

Theorem 1: H is not recursive.

Proof Assume that H is recursive, so that a TM, M_1 exists that decides it. We would then be able to construct a TM, M_2 that does the following:

1. Turns any input $x \in \{0, 1\}^\star$ into $x$code(x).
2. Applies the machine M_1 to $x$code(x).

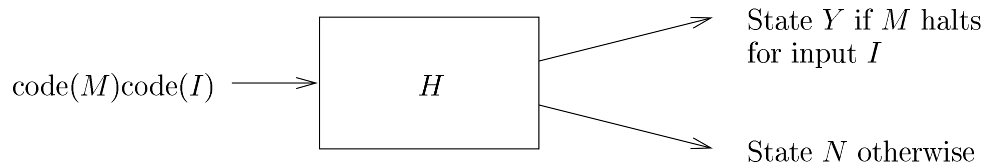
So:

1. If x is in SA then M_1 halts in state Y for input $x$code(x).
2. If x is not in SA then M_1 halts in state N for input $x$code(x).

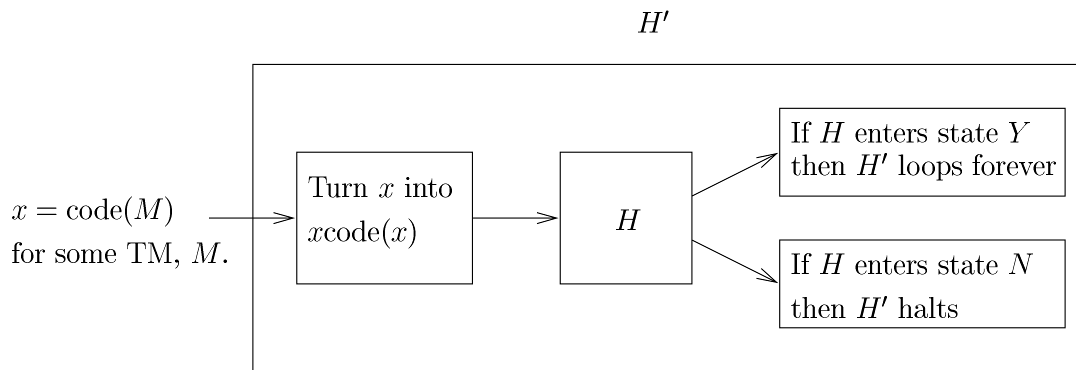
Consequently, M_2 decides SA—a contradiction.

The Halting Problem - Version II

There is another way of proving that the Halting problem is unsolvable, which does not rely on knowing that SA is not recursive. Assume that the Halting problem is solvable, so that we can design a TM, H, that does the following:



Now we can construct a TM, call it H', that works as follows:



The Halting Problem - Version II

Now, H' itself is a TM, so what happens if we give it the input $x = \text{code}(H')$? There are two possibilities:

1. H' halts. This means that the TM, H, enters state Y when its input is $x\text{code}(x)$, so H' loops forever.
2. H' does not halt. This means that the TM, H, enters state N when its input is $x\text{code}(x)$, so H' halts.

In both cases we have a contradiction.

Reduction

The initial proof that the Halting problem is unsolvable worked as follows:

1. Start with a problem, call it A, *that you know is unsolvable*.
2. Show that *if* you could solve the Halting problem *then you could solve* A.
3. Conclude that you can not solve the Halting problem.

We formalise this technique by introducing the idea of *reduction*.

Definition: Let A be a language over some alphabet Σ, and let B be a language. We say that A reduces to B and write $A \leq B$ if there is a TM, M that computes a total function f such that for any $x \in \Sigma^{\star}$,

$$x \in A \leftrightarrow f(x) \in B$$

In essence, we are saying that a TM that decides B can be used to make another that decides A.

Reduction

You can think of the notation $A \leq B$ as saying 'B is at least as difficult as A'. The reason for this is given by the following result.

Theorem 2: If $A \leq B$ and A is not recursive, then B is not recursive.

Proof Assume B were recursive, so there is a TM, M that decides it, and assume A is a language over Σ. Then we could design a TM to decide A. For any $x \in \Sigma^{\star}$ the TM would:

1. use f to turn x into $f(x)$.
2. use M with the input $f(x)$.

If $x \in A$ then $f(x) \in B$ so the overall TM halts in state Y. If $x \notin A$ then $f(x) \notin B$ so the overall TM halts in state N. This means that the overall TM decides A, which is a contradiction, and so B is not recursive.

Reduction is less useful when A is recursive

If A *is* recursive, then the fact that it reduces to another language B is not very useful to us.

Theorem 3: Let B be a language over Σ and assume that $B \neq \Sigma^\star$ and $B \neq \emptyset$. If A is *any* recursive language then $A \leq B$.

Proof As $B \neq \Sigma^\star$ and $B \neq \emptyset$ we can choose a $b_1 \in \Sigma^\star$ such that $b_1 \in B$ and a $b_2 \in \Sigma^\star$ such that $b_2 \notin B$. The function f is defined as follows:

$$f(x) = \begin{cases} b_1 & \text{if } x \in A \\ b_2 & \text{otherwise} \end{cases}$$

Clearly f is total, and can be implemented by a TM as A is recursive. Also, $x \in A \leftrightarrow f(x) \in B$ as required.

Reduction is not (quite) an equivalence relation

It should be clear that:

1. Reduction is *reflexive*: for any language A it is true that $A \leq A$.
2. Reduction is *transitive*: for any languages A, B and C, if $A \leq B$ and $B \leq C$ then $A \leq C$.

If it were also true that reduction were *symmetric*, that is, for any languages A and B, $A \leq B \rightarrow B \leq A$, then $\leq$ would be an *equivalence relation*. However, this is not the case.

Theorem 4: $\leq$ is not symmetric.

Proof From theorem 3 we know that if B is a language over Σ where $B \neq \Sigma^\star$ and $B \neq \emptyset$ and A is a recursive language then $A \leq B$. However, assume B is not recursive. It can not be the case that $B \leq A$, as this would mean that B is recursive.

Example: the Halting problem for the empty tape

The empty tape halting problem (ETHP) is defined as follows:

Instance: A TM, M.

Question: Does M halt when given ε as input?

Theorem 5: The empty tape halting problem is unsolvable.

Proof We construct the reduction HP $\leq$ ETHP. If we have any instance of HP, say code(M)code(I), then we can clearly construct a TM, M' that does the following:

1. M' writes I on its tape.
2. M' then simulates M.

Now, M' is the required instance of ETHP. Note the following:

1. If M halts for input I, then M' halts for input ε.
2. If M does not halt for input I, then M' does not halt for input ε.

<u>Example: does a TM accept all possible inputs?</u>

Let's look at another simple example: HAI (Halts for All Inputs).

Instance: A TM, M.

Question: Does M halt for all possible inputs?

Theorem 6: HAI is unsolvable.

Proof Again, we show that HP $\leq$ HAI. Given any instance code(M)code(I) of HP we can construct a TM, M' that does the following:

1. M' erases it's input.
2. M' writes I on its tape.
3. M' simulates M.

If M halts for input I, then M' halts for all possible inputs. If M fails to halt for input I, then M' halts for *no* inputs.

Example: given two TM's, do they do the same thing?

A slightly more complicated decision problem is ETM (Equivalence of Turing Machines).

Instance: Two TMs, M_1 and M_2, having the same input alphabet Σ_I.

Question: Is it the case that M_1 and M_2 compute the same function?

Example: given two TM's, do they do the same thing?

Theorem 7: ETM is unsolvable.

Proof We show that HAI $\leq$ ETM. Assume we have an instance $M = (\Sigma, Q, q_0, H, f)$ of HAI, and M uses input alphabet Σ_I.

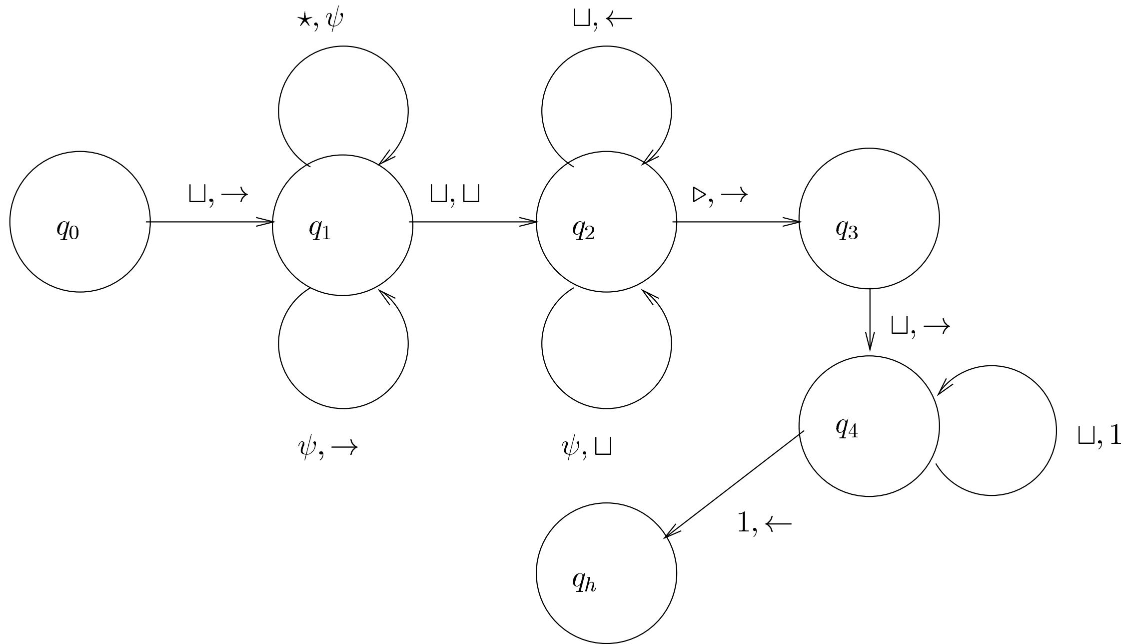
We need to construct M_1 and M_2. A two tape TM, M' can be constructed from M as follows:

1. M' copies its input to tape 2.
2. M' simulates M on tape 2.
3. If M would halt for this input then M' outputs 1, otherwise it fails to halt.

M_1 is now the single tape TM obtained from M'. Clearly, if M halts for all inputs then M_1 outputs 1 for all inputs, and if M does not halt for all inputs then the output of M_1 is undefined for at least one input.

Example: given two TM's, do they do the same thing?

M_2 is now simply a TM that outputs 1 for all inputs. For example,



where $\star$ denotes *any* symbol in Σ_I and $\psi \notin \Sigma_I$.

Now note that:

1. If M halts for all inputs then M_1 outputs 1 for all inputs so M_1 and M_2 compute the same function.
2. If M does not halt for all inputs then M_1 has undefined output for at least one input so M_1 and M_2 compute different functions.

An Exercise

Consider the following decision problem.

Instance: A TM, M and a symbol σ in the input alphabet of M.

Question: Does M ever write σ on its tape after being started with the input ε?

Exercise: Prove that this problem is unsolvable.

3C04 - Complexity Theory

Lecture 12

A problem involving tilings.

So far, all the unsolvable decision problems that we've looked at have basically involved Turing machines. We now look at a problem that does not.

Aims:

- To introduce the idea of a *Tiling System*.
- To prove that the problem of deciding whether, given a tiling system, a tiling exists is unsolvable.

Copyright ©1997 to 2002, by Sean Holden

Tiling Problems

A *tiling system* can be described as follows. We have a set of square tiles, for example,

$$\mathcal{T} = \{ \quad \blacksquare, \quad \blacksquare, \quad \square \quad \}$$

We also have:

- A particular tile $t_0 \in \mathcal{T}$ known as the *origin tile*.
- A set of *adjacency rules*, which specify which tiles may be placed next to each other.

Consider the positive quadrant of the plane. Given a tiling system, a *tiling* is an arrangement of the tiles in $\mathcal{T}$ with the following properties:

- t_0 is placed in the lower left corner.
- The tiles are arranged such that the entire quadrant is covered with no gaps.
- The adjacency rules are all obeyed.

Tiling Problems

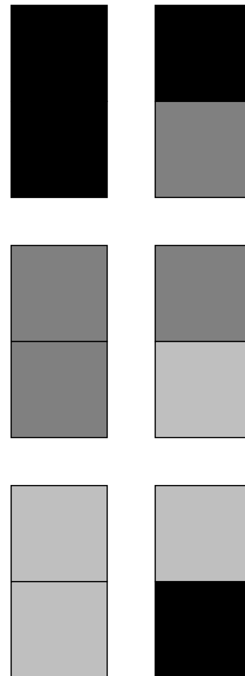
Example:

$\mathcal{T} = \{$, , $\}$

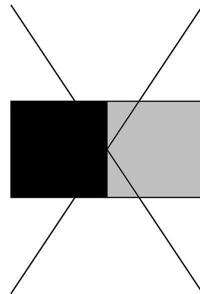
$t_0 =$

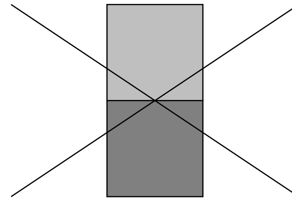
The adjacency rules are:

Horizontal **Vertical**



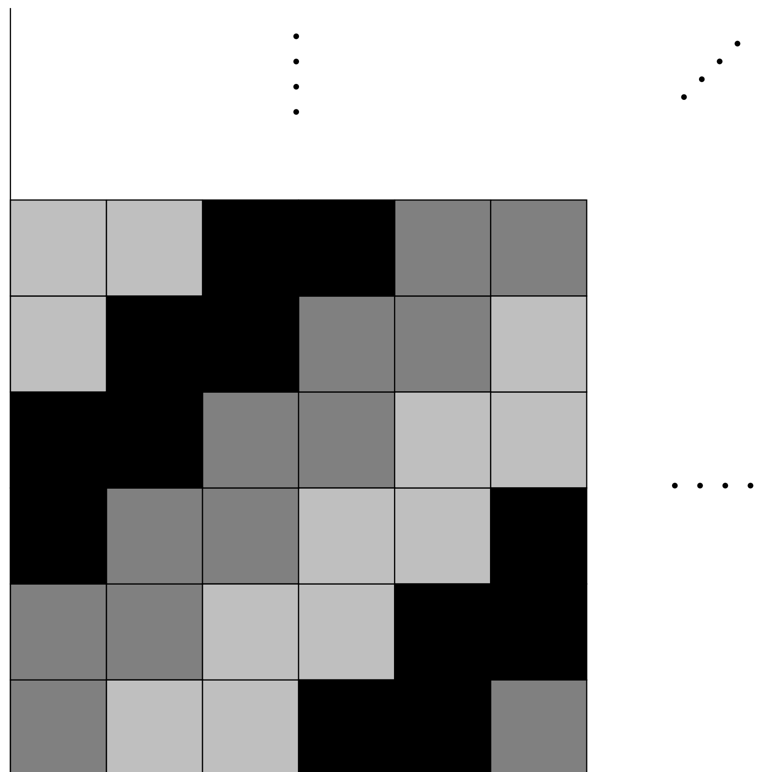
Note that the adjacency rules explicitly do *not* allow tiles to be placed, for example, as follows:





Not allowed　　Not allowed

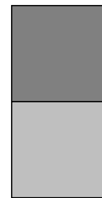
A tiling for the tiling system given in this example is:



Tiling Problems

Clearly, it will not always be possible to find a tiling for a given tiling system. For example, with $\mathcal{T}$ and t_0 as above, if the adjacency rules are changed to:

Horizontal **Vertical**



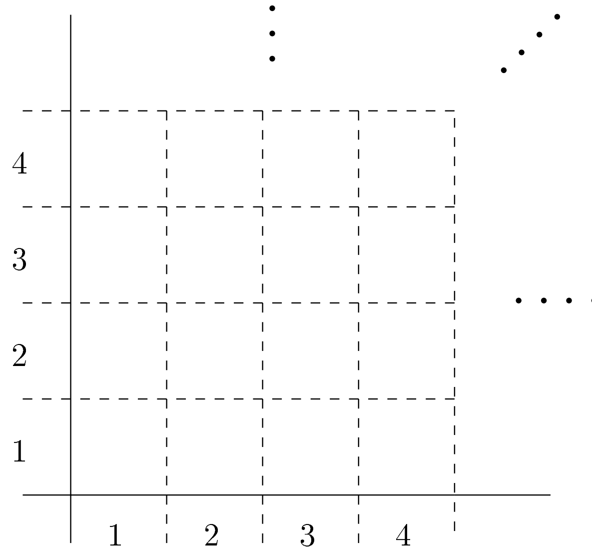
then no tiling exists.

Tiling Problems

More formally, a tiling system consists of:

1. A set $\mathcal{T}$ of tiles.
2. An origin tile $t_0 \in \mathcal{T}$.
3. A set $H \subseteq \mathcal{T}^2$ of horizontal adjacency rules and a set $V \subseteq \mathcal{T}^2$ of vertical adjacency rules.

We can consider the positive quadrant of the plane to be divided into cells identified by their co-ordinates:



A tiling can then be defined as a function $t : \mathbb{N} \times \mathbb{N} \to \mathcal{T}$ with the following properties:

1. $t(1, 1) = t_0$.
2. $(t(n, m), t(n, m + 1)) \in V$ for all $n, m \in \mathbb{N}$.
3. $(t(n, m), t(n + 1, m)) \in H$ for all $n, m \in \mathbb{N}$.

Tiling Problems

- The problem we're going to look at is this:

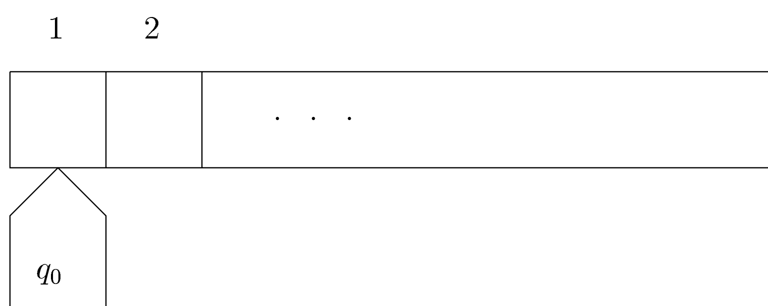
Is there an algorithm that, given a tiling system, decides whether a tiling exists?

- The answer to this question turns out to be *no*. We prove this by reducing to it a version of the empty tape halting problem.

A Variant of the Standard Turing Machine

In order to prove this result, we'll start with a slightly modified form of the TM. The machine is specified as follows:

1. The tape is still infinite in only one direction:

   As previously, the machine is initially in state q_0 scanning cell number 1, which is now the *leftmost* cell.

2. The machine has only a single halting state.

3. If at any point the head tries to move left from cell 1 we say that the machine *hangs*. This is *not* the same as saying that the machine halts.

Unsurprisingly, this version of the TM is equivalent to the standard version in terms of what it can compute.

A Variant of the Standard Turing Machine

Any TM, M will be specified by:

1. Its alphabet Σ.
2. Its set of states Q.
3. Its initial state $q_0 \in Q$.
4. Its halting state $h \in Q$.
5. Its program f, which is a total function of the form:

$$f : (Q \backslash \{h\}) \times \Sigma \to Q \times (\Sigma \cup \{\leftarrow, \rightarrow\})$$

As usual, we assume that Σ does not contain $\leftarrow$ or $\rightarrow$.

A Variant of the Empty Tape Halting Problem

- Consider the following problem:

  Given a TM, M of the kind described, does M halt when started with an empty tape?

- Unsurprisingly, this problem is undecidable. To show that the tiling problem is also undecidable we must show how to take any TM, M and turn it into a tiling system in such a way that M does not halt when started on an empty tape if and only if a tiling exists.

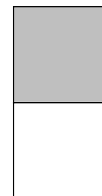
Representing Adjacency Rules

- We can specify the set of tiles and the adjacency rules simultaneously by marking the edges of the tiles, and specifying that they can not be rotated or turned over.
- For example, the tiling system

$\mathcal{T} = \{$, $\}$

Horizontal

Vertical



can be represented as,

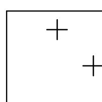
$\mathcal{T} = \{$ b a , b a a b $\}$

- Tiles can be placed next to each other if their edges match.

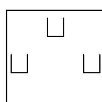
Constructing a Tiling System from a TM

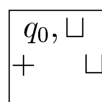
We now show how, given a TM, M, an appropriate tiling system can be constructed. The basic idea is that successive rows of the tiling will represent the tape of M at successive steps. Special tiles will be included to keep track of the current state and current head position.

We begin with the origin tile,

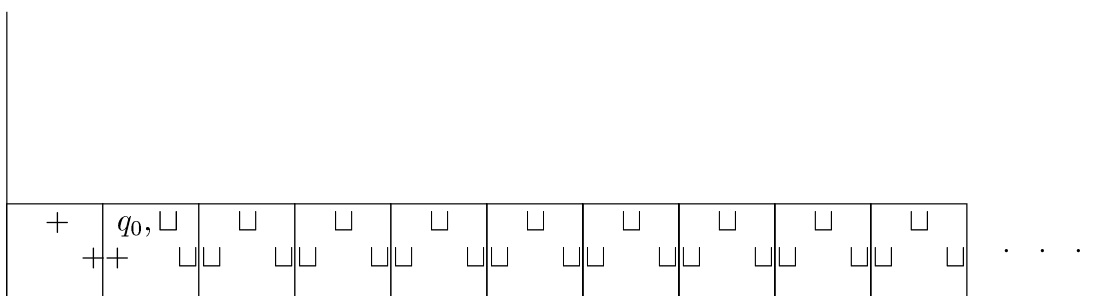


and in addition we include the tiles,





The remaining tiles will be produced such that this forces the first row of any tiling to represent M at the start of a computation.



Constructing a Tiling System from a TM

- For each $\sigma \in \Sigma$, the tiling system includes a tile,

σ
σ

  These tiles represent the fact that only *one* cell can be altered at any step in the computation.

- The next set of tiles to be included represents the fact that at each step in the computation M can change the contents of the cell at the current position, and change its state. (We deal with the movement of the head in a moment.)

- For each $q \in Q \backslash \{h\}$ and $\sigma \in \Sigma$ for which $f(q, \sigma) = (q', \sigma')$ and σ' is not $\leftarrow$ or $\rightarrow$ the tiling system includes a tile,

q', σ'
q, σ

Constructing a Tiling System from a TM

Finally we deal with movement of the head.

- For each $q \in Q\backslash\{h\}$ and $\sigma \in \Sigma$ for which $f(q,\sigma) = (q', \leftarrow)$ and for each $\sigma' \in \Sigma$ the tiling system includes the tiles:

$$\boxed{\begin{array}{c} q',\sigma' \\ \sigma' \end{array}} \leftarrow q' \qquad\qquad q' \leftarrow \boxed{\begin{array}{c} \sigma \\ q,\sigma \end{array}}$$

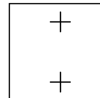
- For each $q \in Q\backslash\{h\}$ and $\sigma \in \Sigma$ for which $f(q,\sigma) = (q', \rightarrow)$ and for each $\sigma' \in \Sigma$ the tiling system includes the tiles:

$$\boxed{\begin{array}{c} \sigma \\ q,\sigma \end{array}} \rightarrow q' \qquad\qquad q' \rightarrow \boxed{\begin{array}{c} q',\sigma' \\ \sigma' \end{array}}$$

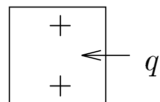
Constructing a Tiling System from a TM

Finally, the tiling system includes,

- The tile,

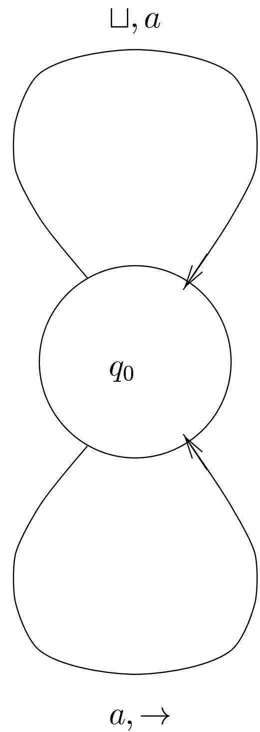


- For each $q \in Q$, the tile,



Constructing a Tiling System from a TM

Example: Consider the TM,



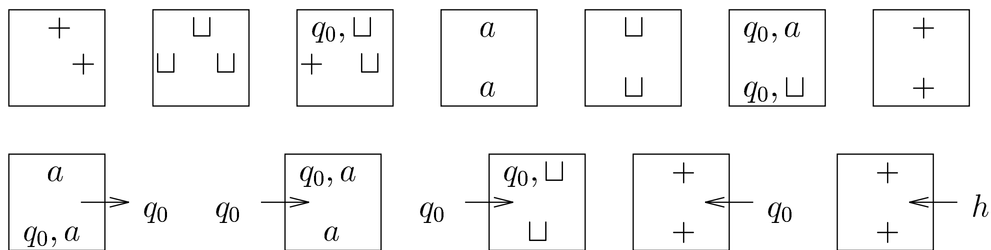
$\Sigma = \{a, \sqcup\}$

$Q = \{q_0, h\}$

$f(q_0, \sqcup) = (q_0, a)$

$f(q_0, a) = (q_0, \rightarrow)$

The tiling system is,



Constructing a Tiling System from a TM

In this case, the tiling is as follows:

$+$	a	a	a $\xrightarrow{q_0}$	$q_0, \sqcup$	$\sqcup$	$\sqcup$
$+$	a	a	q_0, a	$\sqcup$	$\sqcup$	$\sqcup$
$+$	a	a	q_0, a	$\sqcup$	$\sqcup$	$\sqcup$
$+$	a	a	$q_0, \sqcup$	$\sqcup$	$\sqcup$	$\sqcup$
$+$	a	a $\xrightarrow{q_0}$	$q_0, \sqcup$	$\sqcup$	$\sqcup$	$\sqcup$
$+$	a	q_0, a	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$
$+$	a	q_0, a	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$
$+$	a	$q_0, \sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$
$+$	a $\xrightarrow{q_0}$	$q_0, \sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$
$+$	q_0, a	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$
$+$	q_0, a	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$
$+$	$q_0, \sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$
$+$	$q_0, \sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$
$+$	$+$ $\sqcup$	$\sqcup$ $\sqcup$	$\sqcup$ $\sqcup$	$\sqcup$ $\sqcup$	$\sqcup$ $\sqcup$	$\sqcup$ $\sqcup$

Clearly, if M does *not* halt, a tiling exists.

Constructing a Tiling System from a TM

Also, note that,

1. If the machine tries to move left from the end of the tape, then a tiling exists.

2. The first row can only be constructed in one way.

3. After n rows have been constructed either:

   - The TM has moved left from the end of the tape, and so a tiling can be produced.
   - The TM is still computing. In this case, row $n+1$ can be constructed *in only one way*.
   - The TM has halted. In this case row $n+1$ can not be constructed.

As a consequence of points 2 and 3, if the TM halts, then *no* tiling exists.

Conclusion: we have reduced the empty tape halting problem to the tiling problem. Therefore the tiling problem is unsolvable.

3C04 - Complexity Theory

Lectures 14 and 15

Aims:

- To review some material on the *predicate calculus*.
- To prove that the problem of deciding whether a formula is satisfiable is unsolvable.

Copyright ©1997 to 2002, by Sean Holden

Predicate calculus - some revision

We have a supply of,

1. *Variables* x, y, z, ...
2. *Predicate signs* P, Q, R, ... each of which has a *number of places*.
3. *Function signs* f, g, h, ... each of which has a *number of places*.

We refer to a predicate sign having n places as an n-place predicate sign, and similarly in the case of function signs. A 0-place function sign will also be called a *constant*. We will usually use the letters c, d, ... to refer to constants.

<u>Predicate calculus - some revision</u>

Formulas of the predicate calculus can be constructed as follows:

Terms:

1. Each variable is a term.
2. Let f be any n-place function sign and let $t_1, t_2, \ldots, t_n$ be terms. Then $f(t_1 t_2 \cdots t_n)$ is also a term.
3. Nothing else is a term.

Atomic Formulae:

1. Let P be any n-place predicate sign and let $t_1, t_2, \ldots, t_n$ be terms. Then $Pt_1 t_2 \cdots t_n$ is an atomic formula.
2. Nothing else is an atomic formula.

Predicate calculus - some revision

Formulae:

1. Any atomic formula is a formula.
2. If A and B are both formulae then the following are also formulae.
   - $(A \vee B)$ - 'A or B', the *disjunction* of A and B.
   - $(A \wedge B)$ - 'A and B', the *conjunction* of A and B.
   - $\neg A$ - 'not A', the *negation* of A.
3. If A is a formula and x is a variable then the following are also formulae.
   - $\forall x A$ - 'for all x, A'.
   - $\exists x A$ - 'there exists an x, A'.
4. Nothing else is a formula.

Predicate calculus - some revision

The symbol $\forall$ is the *universal quantifier* and the symbol $\exists$ is the *existential quantifier*.

We can also use the *conditional* and the *biconditional*.

Conditional: $(A \rightarrow B)$ is an abbreviation for,

$$(\neg A \vee B)$$

Biconditional: $(A \leftrightarrow B)$ is an abbreviation for,

$$((A \rightarrow B) \wedge (B \rightarrow A))$$

Predicate calculus - some revision

Given any formula F we can decompose it into *subformulae* in the obvious way. The subformulae of F are obtained as follows.

1. F itself is a subformula of F.
2. If $F = (A \vee B)$ or $F = (A \wedge B)$ for formulae A and B, the subformulae of A and B are subformulae of F.
3. If $F = \neg A$ for a formula A, or $F = \exists x A$ or $F = \forall x A$ where x is a variable and A is a formula, then the subformulae of A are subformulae of F.

The *matrix* of a formula F is the formula F' obtained by removing all quantifiers and the variables immediately following them from F,

Example: If,

$$F = \forall x(\exists y Pxy \wedge \neg \forall x Q f(x) y)$$

then the subformulae are,

$$\forall x(\exists y Pxy \wedge \neg \forall x Q f(x) y)$$

$$(\exists y Pxy \wedge \neg \forall x Q f(x) y)$$

$$\exists y Pxy$$

$$\neg \forall x Q f(x) y$$

$$Pxy$$

$$\forall x Q f(x) y$$

$$Q f(x) y$$

and the matrix is,

$$F' = (Pxy \wedge \neg Q f(x) y)$$

Predicate calculus - some revision

Scope:

1. The *scope* of an occurrence of $\neg$ in a formula F is the subformula G of F such that the $\neg$ in question is the leftmost symbol of $\neg G$.
2. The scope of a quantifier Q ($Q = \forall$ or $Q = \exists$) is the subformula G of F such that the Q in question is the leftmost symbol of QvG where v is the relevant variable.

Example: if $F = \forall x(\exists yPxy \wedge \neg\forall xQf(x)y)$.

1. The scope of the $\neg$ is,
$$\forall xQf(x)y$$
2. The scope of the first $\forall$ is,
$$(\exists yPxy \wedge \neg\forall xQf(x)y)$$
3. The scope of the first $\exists$ is,
$$Pxy$$
4. The scope of the second $\forall$ is,
$$Qf(x)y$$

Structures

Everything introduced so far has basically involved the manipulation of strings of symbols—we do not yet have any way of assigning a value of *true* or *false* to a formula. In order to do this we need some more concepts.

A *structure* $\mathbb{A} = ([\mathbb{A}], \phi_{\mathbb{A}})$ consists of a set $[\mathbb{A}]$ called the *universe* of $\mathbb{A}$ and a function $\phi_{\mathbb{A}}$ having as its domain a set of function and predicate signs. The function $\phi_{\mathbb{A}}$ has the following properties.

- Assume P is an n-place predicate sign, and is in the domain of $\phi_{\mathbb{A}}$. Then $\phi_{\mathbb{A}}(P)$ is an n-ary relation on $[\mathbb{A}]$ (a subset of $[\mathbb{A}]^n$).
- Assume f is an n-place function sign, and is in the domain of $\phi_{\mathbb{A}}$. Then $\phi_{\mathbb{A}}(f)$ is a function from $[\mathbb{A}]^n$ to $[\mathbb{A}]$.

We will write $P^{\mathbb{A}}$ instead of $\phi_{\mathbb{A}}(P)$, and $f^{\mathbb{A}}$ instead of $\phi_{\mathbb{A}}(f)$.

Structures

If F is a formula and $\mathbb{A}$ is a structure where each function and predicate sign in F is given a value by $\phi_{\mathbb{A}}$, then $\mathbb{A}$ is *appropriate* to F.

Let F be a formula and let $\mathbb{A}$ be a structure appropriate to F. Let ξ be a function where:

1. The domain of ξ includes all the variables in F.
2. The range of ξ is a subset of $[\mathbb{A}]$.

Then ξ is *appropriate* to F and $\mathbb{A}$.

If x is a variable in F and $a \in [\mathbb{A}]$ then $\xi[x/a]$ is a function ξ' identical to ξ apart from the fact that $\xi'(x) = a$.

Structures

Let F be a formula, let $\mathbb{A}$ be a structure appropriate to F, and let ξ be a function appropriate to F and $\mathbb{A}$. For terms and formulae constructed using the function signs, predicate signs, and variables in F we define the following:

1. If x is a variable then $\mathbb{A}(x)_\xi = \xi(x)$.
2. If $t_1, \ldots, t_n$ are terms and f is an n-place function sign, then

$$\mathbb{A}(f(t_1 \cdots t_n))_\xi = f^{\mathbb{A}}(\mathbb{A}(t_1)_\xi, \ldots, \mathbb{A}(t_n)_\xi)$$

3. If $t_1, \ldots, t_n$ are terms and P is an n-place predicate sign then

$$\mathbb{A}(Pt_1 \cdots t_n)_\xi = \begin{cases} T & \text{if } (\mathbb{A}(t_1)_\xi, \ldots, \mathbb{A}(t_n)_\xi) \in P^{\mathbb{A}} \\ F & \text{otherwise} \end{cases}$$

Structures

4.

$$\mathbb{A}((A \vee B))_\xi = \begin{cases} T & \text{if } \mathbb{A}(A)_\xi = T \text{ or } \mathbb{A}(B)_\xi = T \\ F & \text{otherwise} \end{cases}$$

5.

$$\mathbb{A}((A \wedge B))_\xi = \begin{cases} T & \text{if } \mathbb{A}(A)_\xi = T \text{ and } \mathbb{A}(B)_\xi = T \\ F & \text{otherwise} \end{cases}$$

6.

$$\mathbb{A}(\neg A)_\xi = \begin{cases} T & \text{if } \mathbb{A}(A)_\xi = F \\ F & \text{otherwise} \end{cases}$$

7.

$$\mathbb{A}(\forall x A)_\xi = \begin{cases} T & \text{if for each } a \in [\mathbb{A}],\ \mathbb{A}(A)_{\xi[x/a]} = T \\ F & \text{otherwise} \end{cases}$$

8.

$$\mathbb{A}(\exists x A)_\xi = \begin{cases} T & \text{if for some } a \in [\mathbb{A}],\ \mathbb{A}(A)_{\xi[x/a]} = T \\ F & \text{otherwise} \end{cases}$$

We write,

$$\mathbb{A} \models_\xi A$$

if and only if $\mathbb{A}(A)_\xi = T$.

Free and bound variables

The *free* variables of a formula can be defined as follows.

1. All the variables in an atomic formula are free.
2. The free variables of a formula $\neg A$ are the free variables of A.
3. The free variables of formulas $(A \vee B)$ or $(A \wedge B)$ are the free variables of A and the free variables of B.
4. The free variables of formulas $\forall x A$ or $\exists x A$ are the free variables of A with the exception of x.

Say x is a variable that appears in a formula A. An occurrence of x in A is a *bound* occurrence if it is in a subformula of A of the form $\forall x B$ or $\exists x B$. Any occurrence of x in A that is not a bound occurrence is a *free* occurrence.

Closed formulae, models, and satisfiability

A formula A is called *closed* if it contains no free occurrences of any variables.

1. Assume A is a closed formula. Then it is the case that $\mathbb{A} \models_\xi A$ for some appropriate function ξ if and only if $\mathbb{A} \models_\xi A$ for *all* appropriate functions ξ.
   - We therefore simply write $\mathbb{A} \models A$.
   - We call $\mathbb{A}$ a *model* for A.
2. If A is a closed formula and $\mathbb{A} \models A$ for all appropriate structures $\mathbb{A}$ then A is said to be *valid*.
3. If A is a closed formula having at least one model then A is said to be *satisfiable*. If A is not satisfiable it is said to be *unsatisfiable*.
4. A closed formula A is valid if and only if $\neg A$ is unsatisfiable.

Our aim is to prove that there is no algorithm that can solve the following decision problem:

Instance: A formula, A.

Question: Is A satisfiable?

Equivalence

Two formulae A and B are said to be *equivalent* if and only if for every appropriate $\mathbb{A}$ and ξ we have,

$$\mathbb{A}(A)_\xi = \mathbb{A}(B)_\xi$$

If A and B are equivalent we write,

$$A \equiv B.$$

For example, if A is a formula then,

$$A \equiv \neg\neg A$$

$$\exists x A \equiv \neg \forall x \neg A.$$

If $A \equiv B$ and F' is obtained from F by replacing an occurrence of A with B, then $F' \equiv F$.

Rectifying a formula

It is quite possible for a variable to have both free and bound occurrences in the same formula. For example, in,

$$F = (\forall x Px \vee Qx)$$

the variable x has a free occurrence and a bound occurrence.

If x is a variable and t is a term, then $F[x/t]$ is obtained by replacing the free occurrences of x in F by t. For example,

$$F[x/y] = (\forall x Px \vee Qy)$$

If A is a formula, x is a variable, and y is a variable that does not occur in A, then,

$$\forall x A \equiv \forall y A[x/y].$$

Using this fact it is possible to take any formula and obtain an equivalent formula in which,

- No variable has both free and bound occurrences.
- Each variable occurs with at most one quantifier.

The formula obtained is said to be *rectified*.

Rectifying a formula

Example: In the formula,

$$F = ((\forall x(Px \wedge \neg Qx) \wedge \forall yQy) \vee (Qx \wedge \exists yPy))$$

x has both free and bound occurrences, and y occurs with two quantifiers. We can obtain the rectified formula,

$$F' = ((\forall w(Pw \wedge \neg Qw) \wedge \forall zQz) \vee (Qx \wedge \exists yPy))$$

Functional Form

We classify quantifiers as being *positive* or *negative* as follows:

- A universal quantifier is positive if it is in the scope of an even number of negations, and negative otherwise.
- An existential quantifier is positive if it is in the scope of an odd number of negations, and negative otherwise.

Example: In the formula,

$$F = (\neg \forall x (Px \wedge \neg \exists y Qxy) \vee \forall x Qxx)$$

the first $\forall$ is negative, the second $\forall$ is positive, and the $\exists$ is negative.

Functional Form

Given a closed formula F we obtain its *functional form* as follows:

1. Rectify F.
2. For every negative quantifier Q let:
   - x be the variable it binds
   - A be its scope
   - $y_1, \ldots, y_n$ be, in order, the variables bound by the positive quantifiers in whose scope QxA occurs

   Choose a new n-place function sign f_x and replace QxA with $A[x/f_x(y_1 \cdots y_n)]$.
3. Repeat step 2 until all negative quantifiers have been removed.

Functional Form

Example: to obtain the functional form of,

$$\forall x(\neg\forall y(Py \wedge \neg\exists z Qyz) \vee Qxx).$$

The formula does not need to be rectified. The first $\forall$ is positive, and the second $\forall$ and the $\exists$ are negative. Dealing with the $\forall$ first we obtain,

$$\forall x(\neg(Pf_y(x) \wedge \neg\exists z Qf_y(x)z) \vee Qxx)$$

and dealing with the $\exists$ gives the functional form,

$$\forall x(\neg(Pf_y(x) \wedge \neg Qf_y(x)f_z(x)) \vee Qxx)$$

The Herbrand Universe and the Herbrand Expansion

If F is a closed formula, let F^* denote the matrix of its functional form. In the last example, we had,

$$F = \forall x(\neg\forall y(Py \wedge \neg\exists z Qyz) \vee Qxx)$$

and F^* will be,

$$F^* = (\neg(Pf_y(x) \wedge \neg Qf_y(x)f_z(x)) \vee Qxx)$$

The *Herbrand Universe* $U(F)$ is defined as follows:

1. If F^* contains no constants then $\bullet \in U(F)$, where $\bullet$ is a special constant used to begin the process if F^* contains no constants.
2. If $t_1, \ldots, t_n \in U(F)$ and f is an n-place function sign in F^* then $f(t_1 \cdots t_n) \in U(F)$.

The Herbrand Universe and the Herbrand Expansion

We can now define the *Herbrand Expansion* $H(F)$ of F.

Let $v_1, \ldots, v_n$ be the variables of F^*. Then $H(F)$ is the set,

$$H(F) = \{F^*[v_1/t_1, v_2/t_2, \ldots, v_n/t_n] : t_1, \ldots, t_n \in U(F)\}$$

Using out previous example with,

$$F^* = (\neg(Pf_y(x) \wedge \neg Qf_y(x)f_z(x)) \vee Qxx)$$

we obtain,

$$\begin{aligned}
U(F) &= \{\bullet, f_y(\bullet), f_z(\bullet), f_z(f_y(\bullet)), f_y(f_y(\bullet)), \ldots\} \\
H(F) &= \{(\neg(Pf_y(\bullet) \wedge \neg Qf_y(\bullet)f_z(\bullet)) \vee Q\bullet\bullet), \\
&\quad (\neg(Pf_y(f_y(\bullet)) \wedge \neg Qf_y(f_y(\bullet))f_z(f_y(\bullet))) \\
&\quad \vee Qf_y(\bullet)f_y(\bullet)), \\
&\quad \ldots\}
\end{aligned}$$

The Expansion Theorem

None of the formulae in the Herbrand expansion will contain any quantifiers or variables. We now treat the *atomic formulae* in $H(F)$ as *propositions*, and the overall formulae in $H(F)$ as formulae of the *propositional* calculus.

For example, using the first formula in $H(F)$ from the example, we have,

$$(\neg(\underbrace{Pf_y(\bullet)}_{\text{proposition}} \wedge \neg \underbrace{Qf_y(\bullet)f_z(\bullet)}_{\text{proposition}}) \vee \underbrace{Q\bullet\bullet}_{\text{proposition}})$$

which, as a formula in the propositional calculus, might be,

$$(\neg(R \wedge \neg S) \vee T)$$

Theorem (The Expansion Theorem) A closed formula F is satisfiable if and only if $H(F)$ is satisfiable.

Another Example Consider the closed formula,

$$F = \forall x \exists y Pxy$$

which is satisfiable. The functional form of F is,

$$\forall x P x f_y(x)$$

and so,

$$\begin{aligned}
F^* &= P x f_y(x) \\
U(F) &= \{\bullet, f_y(\bullet), f_y(f_y(\bullet)), f_y(f_y(f_y(\bullet))), \ldots\} \\
H(F) &= \{P \bullet f_y(\bullet), \\
&\quad\; P f_y(\bullet) f_y(f_y(\bullet)), \\
&\quad\; P f_y(f_y(\bullet)) f_y(f_y(f_y(\bullet))), \ldots\}
\end{aligned}$$

The tiling problem revisited

Recall that a *tiling system* consists of the following:

- a set of tiles,

$$\mathcal{T} = \{t_1, t_2, \ldots, t_n\}$$

- an origin tile $t_0 \in \mathcal{T}$
- a set of horizontal adjacency rules $H \subseteq \mathcal{T}^2$
- a set of vertical adjacency rules $V \subseteq \mathcal{T}^2$

Also, recall that, given a tiling system, a *tiling* is a function $T : \mathbb{N} \times \mathbb{N} \to \mathcal{T}$ with the following properties:

- $T(1, 1) = t_0$
- $(T(n, m), T(n, m + 1)) \in V$ for all $n, m \in \mathbb{N}$
- $(T(n, m), T(n + 1, m)) \in H$ for all $n, m \in \mathbb{N}$

We've already proved that there is no algorithm that, given a tiling system, decides whether a tiling exists.

A further unsolvable problem

Theorem The tiling problem reduces to the following problem:

Given a closed formula A, is A satisfiable?

Consequently, the latter problem is unsolvable.

Proof Let $(\mathcal{T}, t_0, H, V)$ be a tiling system. We need to construct a formula such that a tiling exists if and only if the formula is satisfiable.

The formula will make use of:

1. A single 1-place function sign f.
2. A single constant c.
3. For each $t \in \mathcal{T}$, a 2-place predicate sign P_t.

A further unsolvable problem

The formula constructed for a given tiling system will then be,

$$\begin{aligned}
A \;=\; \forall x \forall y \Bigg(& \bigwedge_{t_1, t_2 \in \mathcal{T}, t_1 \neq t_2} \neg (P_{t_1} xy \wedge P_{t_2} xy) \\
& \wedge P_{t_0} cc \\
& \wedge \bigvee_{(t_1, t_2) \in H} (P_{t_1} xy \wedge P_{t_2} f(x) y) \\
& \wedge \bigvee_{(t_1, t_2) \in V} (P_{t_1} xy \wedge P_{t_2} x f(y)) \Bigg)
\end{aligned}$$

If a tiling exists, then A is satisfiable. Informally, we use a structure $\mathbb{A} = ([\mathbb{A}], \phi_{\mathbb{A}})$ for which:

1. The universe is $\mathbb{N} = \{1, 2, 3, \ldots\}$.
2. f is the successor function and c is 1.
3. $P_t xy$ indicates whether or not tile t is located at position (x, y).

A further unsolvable problem

More formally, if a tiling $T : \mathbb{N} \times \mathbb{N} \to \mathcal{T}$ exists, then A is satisfiable using the structure $\mathbb{A} = ([\mathbb{A}], \phi_{\mathbb{A}})$ where,

- $[\mathbb{A}] = \mathbb{N}$
- $c^{\mathbb{A}} = 1$
- $f^{\mathbb{A}}(x) = x + 1$ where $x \in \mathbb{N}$
- for each $t \in \mathcal{T}$, $P_t^{\mathbb{A}} = \{(x, y) : T(x, y) = t\}$

Now, $\mathbb{A}$ is a model for A because,

- The first part of A asserts that no position contains two different tiles.
- The second part of A asserts that the origin tile is at position $(1, 1)$.
- The third part of A asserts that horizontally adjacent positions contain tiles obeying the horizontal adjacency rules.
- The fourth part of A asserts that vertically adjacent positions contain tiles obeying the vertical adjacency rules.

A further unsolvable problem

We now need to show that if A is satisfiable then a tiling exists. A has no negative quantifiers, and is in fact already in functional form. The Herbrand universe is therefore easily obtained:

$$U(A) = \{c, f(c), f(f(c)), f(f(f(c))), \ldots\}$$

Also, the Herbrand expansion $H(A)$ of A is easily obtained by substituting elements of $U(A)$ for x and y in the matrix of A.

We will use the notation,

$$\begin{aligned} f^0(c) &= c \\ f^1(c) &= f(c) \\ f^2(c) &= f(f(c)) \\ f^3(c) &= f(f(f(c))) \\ \vdots & \;\vdots\; \vdots \end{aligned}$$

Note that all of the propositions we'll need to consider in $H(A)$ will have the form,

$$P_t f^n(c) f^m(c) \text{ for } t \in \mathcal{T} \text{ and } n, m \in \{0, 1, 2, \ldots\}$$

A further unsolvable problem

Assume A is satisfiable. Then by the expansion theorem so is $H(A)$. This means that we can assign a value of true or false to each of the propositions $P_t f^n(c) f^m(c)$ in such a way that all the formulae in $H(A)$ are true.

Consider the following function:

$$T(n,m) = t \text{ if and only if } P_t f^{n-1}(c) f^{m-1}(c) \text{ is true.}$$

We now show that this is a tiling. To do this we need to show that:

1. $T(n,m)$ is defined for all $n, m \in \mathbb{N}$ and is a properly defined function.
2. $T(1,1)$ is the origin tile.
3. T obeys the adjacency rules.

<u>A further unsolvable problem</u>

Part 1

- If $t_1 \neq t_2$ then it is not possible to have both
$$P_{t_1} f^{n-1}(c) f^{m-1}(c)$$
true and
$$P_{t_2} f^{n-1}(c) f^{m-1}(c)$$
true for any $m, n \in \mathbb{N}$.
This is because there is a formula in $H(A)$ which has a part of the form,
$$\bigwedge_{t_1, t_2 \in \mathcal{T}, t_1 \neq t_2} \neg (P_{t_1} f^{n-1}(c) f^{m-1}(c) \wedge P_{t_2} f^{n-1}(c) f^{m-1}(c))$$
and this part has to be true.
- For any $n, m \in \mathbb{N}$ there must be at least one
$$P_t f^{n-1}(c) f^{m-1}(c)$$
that is true.
This is because there is a formula in $H(A)$ which has a part of the form,
$$\bigvee_{(t_1, t_2) \in H} (P_{t_1} f^{n-1}(c) f^{m-1}(c) \wedge P_{t_2} f^{n}(c) f^{m-1}(c))$$
and this part has to be true.

Part 2

Every formula in $H(A)$ has a part,

$$P_{t_0} f^0(c) f^0(c)$$

which must be true. Therefore $T(1,1) = t_0$.

Part 3

We demonstrate this for the horizontal adjacency rules; the vertical adjacency rules are dealt with in a similar way. We need to show that if T is as defined then for each $n, m \in \mathbb{N}$,

$$(T(n,m), T(n+1,m)) \in H.$$

For each $n, m \in \mathbb{N}$ there is a formula in $H(A)$ with a section,

$$\bigvee_{(t_1,t_2)\in H} (P_{t_1} f^{n-1}(c) f^{m-1}(c) \wedge P_{t_2} f^n(c) f^{m-1}(c))$$

that must be true. Consequently for some pair $(t_1, t_2) \in H$ we must have $P_{t_1} f^{n-1}(c) f^{m-1}(c)$ true and $P_{t_2} f^n(c) f^{m-1}(c)$ true. Consequently $T(n,m) = t_1$ and $T(n+1,m) = t_2$ and $(T(n,m), T(n+1,m)) \in H$ as required.