ITCS 532:
2. Turing Machine Variants

Rob Egrot

Variants of Turing machines

- Last class we saw the definition of a Turing machine.
- This definition is to some extent arbitrary.
- We could make trivial changes and not affect what can be computed. E.g.
  - No : symbol at start of tape (rules for 'hanging' computations).
  - Allow machines to write and move in the same action.
  - Etc.
- We could also make more serious looking changes. E.g.
  - Allow multiple tapes.
  - Allow two-way infinite tapes.
  - Allow infinite states.
- Some of these changes make a significant difference to the computational power of the machine, but others do not.
- We explore this in this class.

Multiple tape Turing machines

- A standard Turing machine has only one tape.
- We can change the definition so that the machine has two, or even more.
- Each tape has its own tape head, but the machine has one global state.
- Each tape head works on its tape independently of the others, according to the transition function δ.
- Question: Should a 2-tape machine be more powerful than a 1-tape machine?
- I.e. are there decision problems that are solvable by a 2-tape machine that are not solvable by a 1-tape machine (ignoring run time)?

Multiple tape Turing machines

- Intuitively we might expect that 2-tape machines are more powerful than 1-tape machines.
- In other words, that TMs with more than one tape could decide more problems and recognize more languages than their single-tape counterparts.
- It turns out that this is *not* the case.
- Multi-tape TMs are equivalent to single-tape TMs, in the sense that for any multi-tape TM there's a single-tape TM that gets the same result for the same input.
- This is not obvious, but we prove it in theorem 2 later.

Multiple tape Turing machines - formal definition

Definition 1 (k-tape Turing machine)

A k-tape Turing machine M_k is a modified TM with the following additional properties:

- M_k has k one-way infinite tapes and k tape heads.
- At any moment M_k is in a state q. I.e. all tape heads share a single state.
- We formally describe M_k as a 5-tuple $(Q, \Sigma, q_0, H, \delta)$.

$$\delta : (Q \setminus H) \times (\Sigma \cup \{\llcorner\!\lrcorner, :\})^k \to Q \times (\Sigma \cup \{\llcorner\!\lrcorner, \leftarrow, \rightarrow\})^k$$

- One tape (call it tape 1) is the designated input tape, and all other tapes start blank.
- Tape 1 is also the output tape.

Two tapes on one

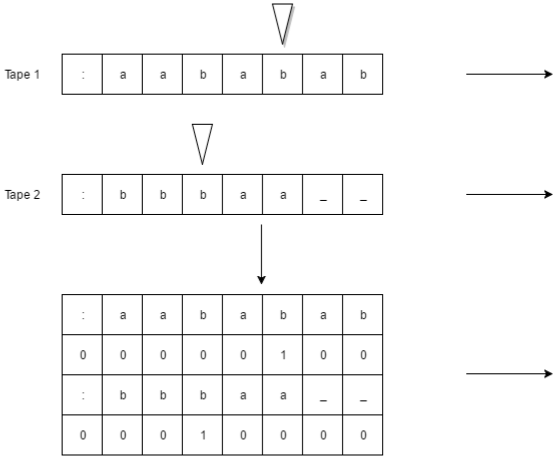
Theorem 2

If Σ is a finite alphabet and M is a k-tape TM using Σ there is a finite alphabet $\Sigma' \supseteq \Sigma$ and a single-tape Turing machine M' using Σ' such that for all $x \in \Sigma^*$ we have $M(x) = M'(x)$.

Proof

- The idea is to represent the state of all the tapes, and the positions of all the tape heads, on one tape.
- To do this we need to expand the alphabet Σ.
- Before we make a formal definition we'll look at an example with two tapes.

Multiple tape Turing machines - equivalence



Extending the alphabet

- We want to define an extended alphabet Σ'.
- In any square other than the first (which is a special case as only : can be written there on each tape), there are $|\Sigma \cup \{\llcorner\!\lrcorner\}|$ possibilities for each tape.
- Also for each tape the tape head may or not be present.
- So we need $(|\Sigma| + 1)^k \times 2^k$ symbols to cover all these possible combinations.
- In addition we need 2^k extra symbols for the first squares of the tapes (where only : can be written but the tape head may or may not be present).
- Finally we also need every symbol from Σ (so that $\Sigma \subset \Sigma'$).
- So we need $((|\Sigma| + 1)^k + 1) \times 2^k + |\Sigma|$ symbols in Σ'.

Defining M'

- Remember we start with multi-tape M, and we want to define single tape M' that does the same thing.
- We describe the operation of M' step by step.
- We assume two tapes, as general idea is the same.

1. M' scans the input x and rewrites x using composite symbols. E.g. $aba\textvisiblespace$ will become

$$\begin{array}{cccc} : & a & b & a \\ 1 & 0 & 0 & 0 \\ : & \textvisiblespace & \textvisiblespace & \textvisiblespace \\ 1 & 0 & 0 & 0 \end{array}$$

The tape of M' now represents the initial configuration of M with input x.

Defining M'

2. - M' scans the tape till it finds the 1 representing the position of the first tape head.
   - M' changes state to record the symbol the 1st tape head would be reading.

3. - M' scans the tape for the 1 representing the position of the 2nd tape head.
   - M' changes state to record what symbols the tape heads of M are reading.

(M' needs many more states that M, as it must record the configuration of M in its state).

Defining the M'

4. Based on the recorded information about the current symbols being read and the simulated state of M (which M' stores via its own state), M' rewrites the tape to represent M acting on all its tapes.

5. Steps 2, 3, and 4 repeat till M would enter a halting state (if this ever happens!), at which point we proceed to either 6 or 7, depending on the kind of halting state:

6. (Acceptance) M' rewrites the tape so that the output as written on the top row of the combined tape is now written using symbols from Σ (so it matches the output of M).

7. (Rejection) M' enters a rejection state.

Defining M'

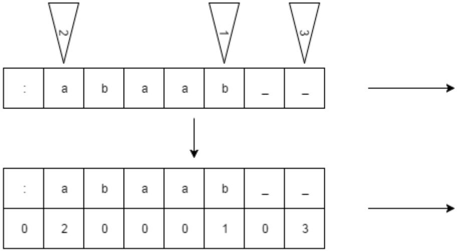
- M' is equivalent to M.
- I.e. Given input x, M' accepts/rejects iff M accepts/rejects, and the output is the same.
- This completes the proof.
- Question: Why is it easy to model a single-tape TM using a multi-tape machine?

Two-way infinite tapes

- The tape of a regular TM is only infinite in one direction.
- What if we modified the definition to allow the tape to be infinite in both directions?
- Would this be a more powerful model of computation? It turns out no.
- Think about the multi-tape example.
- A two-way infinite tape with only one head is at most as powerful as a TM with two tapes and two tape heads.
- But we saw that this has the same power as a regular TM.
- Since a TM with a two-way infinite tape is certainly not *less* powerful than a regular TM, they must be equivalent.

More than one tape head

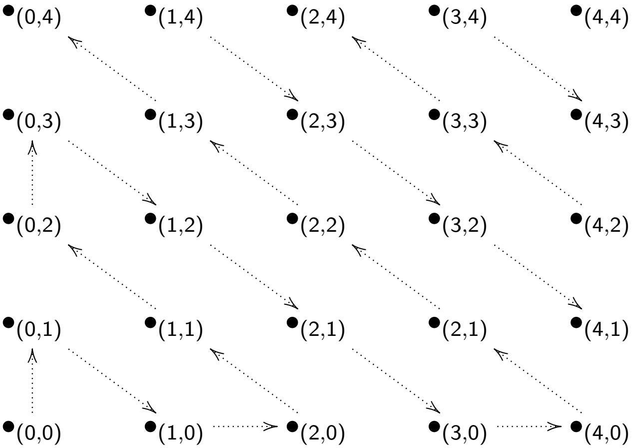
- We could modify the definition of a Turing machine so that there are two (or more) tape heads working on the same tape.
- This is slightly tricky to formally define because we have to deal with the possibility of 'simultaneous edits'.
- It turns out we can simulate a multi-head TM using a regular TM like how we simulated the multi-tape TM.
- I.e. add symbols to the language to model the tape and the positions of the tape heads on a single tape:



Grid instead of tape

- How about instead of a tape we have an infinite grid that the tape head moves around on in two dimensions?
- Again this is not more powerful.
- To see this we first model the 2D grid as a sequence of squares in one dimension.
- This is similar to how we showed the rational numbers are countable.
- Then we have to create enough states and design the code well enough to deal with the fact that moving the tape head is more complicated now.
- It's tricky but it can be done.
- Question: Why are the rational numbers countable?

$\mathbb{Q}$ is countable



Infinite states

- Regular Turing machines can only have a finite number of states.
- How about if we allow the machine to have an infinite number of states?
- This is more powerful!
- In fact it's too powerful.
- If we're allowed to use an infinite number of states then every language over a countable alphabet can be decided by a TM.
- This is because with an infinite number of states we can have one unique state for every possible input.
- So all we have to do to 'decide' a language is assign acceptance or rejection to each state appropriately.
- Having an infinite number of states is more like magic than computation.

The Church-Turing Thesis

The Church-Turing Thesis

Any problem expressible in a formal language that is solvable by a well-defined step by step procedure can be solved by a Turing machine

- 'Thesis' here is used to mean "A statement or theory".
- It's impossible to formally prove the Church-Turing thesis, but it would be theoretically possible to disprove it by producing a counterexample.
  - Though how would you check that your step by step procedure for this hypothetical problem was correct?
- Most computer scientists believe the Church-Turing thesis.
- So we use the theory of Turing machines to put hard theoretical limits on computation.

Enumerators

- Recall that a formal language L is defined to be recursively enumerable if there is a Turing machine that accepts when given words from L as input, and does not halt for other inputs.
- *Enumerable* in English means 'can be counted'.
- So a *recursively enumerable* language should, logically, be one that can be counted (put into a list) by a recursive procedure.
- But the standard definition of an r.e. language doesn't say anything about lists or counting.
- What is going on?
- It turns out that these definitions are equivalent, in a way we make precise now.

Enumerators - formal definition

Definition 3 (Enumerator)

An enumerator is a special Turing machine with the following properties:

1. There is no halt state.
2. There is a special state *print*.
3. The machine starts by erasing the input (or we just assume the input is always empty).
4. When the machine enters the *print* state the current contents of the tape between : and the first ␣ is 'printed' (added to the end of an abstract list that starts empty).

Enumerators and formal languages

- Enumerators capture the intuitive idea of generating a list using an automated procedure.
- Given a finite alphabet Σ, and an enumerator E using Σ, the set of words that will eventually be printed by E form a subset of Σ^*.
- I.e. it's a formal language.
- It turns out that every language created using an enumerator is r.e., and every r.e. language has an enumerator that generates it.
- This is not obvious, so we prove it as theorem 4 now.

Enumerators and formal languages

Theorem 4

Let Σ be a finite alphabet and let $L \subseteq \Sigma^*$. Then L is recursively enumerable if and only if there is an enumerator E with the following properties:

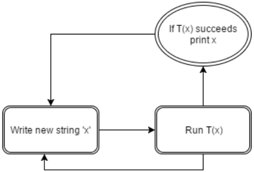
1. *If $s \in L$ then E will print s after a finite number of operations.*
2. *If E prints s then $s \in L$.*

Proof: has enumerator implies r.e.

- We start by proving that if an enumerator E exists for L then L is r.e. as this is relatively easy.
- We use enumerator E to build a Turing machine T_E that accepts for all words in L, and fails to halt for all other words.
- By theorem 2 we can assume that T_E has multiple tapes.
- T_E stores the input string on one tape, and on another tape it outputs strings as produced by E.
- Every time a new string is produced, T_E compares it to the input string.
- If they are the same it accepts.
- Since E is an enumerator for L, if the input string is in L it will eventually be produced by E.
- So T_E semidecides L as required.

Proof: r.e. implies has enumerator

- Harder. Suppose T semidecides L.
- We want to use this to create an enumerator E_T that lists the elements of L.
- Idea: E_T will write the strings from Σ^* one by one on a tape (again we can safely assume multiple tapes).
- Every time a new string is written E_T runs T on that string. If T accepts then E_T 'prints' the string. So the flow would be something like this:



Proof: r.e. implies has enumerator

- Problem. $T(x)$ may not halt! So as soon as we get a string not in L our machine E_T will get stuck and will run forever without printing again.
- We use *dovetailing* to solve this. The modified action of E_T is as follows
  1. Generate a new string x_1.
  2. Simulate $T(x_1)$ for one step only.
  3. Generate a new string x_2.
  4. Simulate $T(x_1)$ for one more step, then simulate $T(x_2)$ for one step.
  5. Generate a new string x_3.
  6. Simulate $T(x_1)$ for one more step, then simulate $T(x_2)$ for one more step, then simulate $T(x_3)$ for one step.
  7. And so on...
  8. Whenever $T(x_n)$ succeeds print x_n.

Proof: r.e. implies has enumerator

- This works because even if one or more of the computations never halts it doesn't matter.
- Because we add an extra string every cycle, the strings that T doesn't accept don't stop the other strings from getting attention.
- Again this is similar to the argument proving $\mathbb{Q}$ is countable.
- We know that, for all values of x, we will eventually have simulated $T(x)$ for any given number of steps.
- It might take a long time, but if T accepts x then E_T will notice and print x.
- We have to manage all this using Turing machine architecture, but we have lots of tapes at our disposal.