

GEM 1501 Problem Solving With Computers

Lecture 9:

Universality

Frank Stephan

Summary of Previous Lecture

- Certain things are not only slow but just impossible
- Unbounded Puzzles, Post's Correspondence Problem
- The Halting-Problem
- Recursively Enumerable Sets
- Rice's Theorem

General Picture

- In general, problems are just sets of texts.
- Highly undecidable problems.
Example: Set of all texts T such that T is the text of a Java Script programme which terminates on all possible inputs.
- Recursively enumerable problems.
These are problems which have yes-certificates.
- Complements of recursively enumerable problems.
These are problems which have no-certificates.
- Decidable Problems.
These problems have yes-certificates and no-certificates.
All the intractable and tractable problems from Lecture 7 are decidable.

Recursively Enumerable Sets

- A nonempty set A is recursively enumerable iff A is the range of a computable function F
iff A has yes-certificates.
- Post's Correspondence Problem is a recursively enumerable set.
- The Halting Problem is a recursively enumerable set.
- Sometimes one can translate a recursively enumerable problem B into another problem A :
There is a computable function F with $T \in B \Leftrightarrow F(T) \in A$.
- A set A is complete iff A is recursively enumerable and for every recursively enumerable problem B there exists a computable function F such that $x \in B \Leftrightarrow F(x) \in A$. The function F translates the problem B into the more difficult problem A .

Hierarchy of Recursively Enumerable Sets

- Complete recursively enumerable problems
The Halting Problem is a complete problem; Post's Correspondence Problem is a complete problem.
Most natural undecidable but recursively enumerable problems are complete.
- Incomplete but undecidable problems
Most examples of these are not natural and difficult to construct. Problem of compressible texts is one which is most easy to understand: T is compressible iff there is a Java Script programme P outputting T such that P needs no input and is shorter than T .
- Decidable problems
A problem A is decidable iff there is an algorithm (computable function) which takes as input any text T and outputs "YES" if $T \in A$ and "NO" if $T \notin A$.
Presburger arithmetic is an example of a decidable problem (although it is intractible).

Proof Techniques

- Diagonalization:
Origin of proof method is Set Theory;
Create new problem D such that $D(P)$ differs from $P(P)$ for all total programmes P with input being P itself;
The halting problem is undecidable.
- Translation and Simulation:
Every recursively enumerable set can be translated into the halting problem;
Rice's Theorem: The halting problem can be translated into many problems dealing with programme behaviour;
Most undecidability results are obtained that way.

Today: Are all computers the same?

- Computers differ in their speed, architecture and design.
- Obvious question: Is every problem solvable by computer A also solvable by computer B?
- In practice: no; — In theory: yes.
- Given enough time and memory, any computer can “simulate” any other computer.
- Emphasis will be on easy computers and mechanisms able to simulate all other computing machinery.

Overview

- **Turing Machines**
- Multiple Stack Machines
- Church's Thesis
- Numerical Computation:
Counter Programmes
- Restricted Models:
One Stack Machines and Finite Automata

Simplifications

- Data: All data are strings!
- Control: Tape, gearbox, narrow eye.
- Operations: Change the symbol under the eye and move left or right.
- Result: The Turing Machine.

The Turing Machine

- Finite set of states
- Finite alphabet of symbols
- Infinite tape
- Head that can read and write symbols on the tape
- State-transition table (“Turing Table”)

`http://www.comp.nus.edu.sg/~gem1501/turingmachine.html`

`http://www.comp.nus.edu.sg/~gem1501/interactiveturing.html`

Changing Symbols

Task: Nonempty input, exchange 0 and 1 everywhere.

States: st, ri, ch. Commands: Write #,0,1,2,3,4; Right; Left; Halt.

Turing Table maps old state * symbol to new state and action.

Old State	#	0	1	2	3	4
st	st R	ri 1	ri 0	ch R	ch R	ch R
ri	----	ch R	ch R	----	----	----
ch	ch H	ri 1	ri 0	ch R	ch R	ch R

Algorithm:

```
st: if symbol == # then right and goto st;
    if symbol == 0 or symbol == 1, then change symbol and goto ri;
    if symbol == 2 or symbol == 3 or symbol == 4, then right and goto ch;
ri: right and goto ch;
ch: if symbol == # then halt;
    if symbol == 0 or symbol == 1, then change symbol and goto ri;
    if symbol == 2 or symbol == 3 or symbol == 4, then right and goto ch.
```

Runtime Example

Old State	#	0	1	2	3	4
st	st R	ri 1	ri 0	ch R	ch R	ch R
ri	----	ch R	ch R	----	----	----
ch	ch H	ri 1	ri 0	ch R	ch R	ch R

State	Position	Symbol	String	Action	New State
st	0	#	#00231#	Right	st
st	1	0	#00231#	Write 1	ri
ri	1	1	#10231#	Right	ch
ch	2	0	#10231#	Write 1	ri
ri	2	1	#11231#	Right	ch
ch	3	2	#11231#	Right	ch
ch	4	3	#11231#	Right	ch
ch	5	1	#11231#	Write 0	ri
ri	5	0	#11230#	Right	ch
ch	6	#	#11230#	Halt	ch

Appending Symbol

Task: Nonempty input, append first symbol at end.

States: st,q0,...,q4,ha. Commands: Write #,0,1,2,3,4; Right; Left; Halt.

Turing Table maps old state * symbol to new state and action.

Old State	#	0	1	2	3	4
st	st R	q0 R	q1 R	q2 R	q3 R	q4 R
q0	ha 0	q0 R	q0 R	q0 R	q0 R	q0 R
q1	ha 1	q1 R	q1 R	q1 R	q1 R	q1 R
q2	ha 2	q2 R	q2 R	q2 R	q2 R	q2 R
q3	ha 3	q3 R	q3 R	q3 R	q3 R	q3 R
q4	ha 4	q4 R	q4 R	q4 R	q4 R	q4 R
ha	ha H	ha H	ha H	ha H	ha H	ha H

Algorithm (with k in {0,1,2,3,4}):

```
st: while (symbol == #) { right; }
```

```
    k = symbol; right; goto qk;
```

```
qk: while (symbol != #) { right; } write k; goto ha;
```

```
ha: halt;
```

Runtime Example

Algorithm

```
st: while (symbol == #) { right; }  
    k = symbol; right; goto qk;  
qk: while (symbol != #) { right; } write k; goto ha;  
ha: halt;
```

Machine uses state to remember first symbol k in $\{0,1,2,3,4\}$.

State	Position	Symbol	String	Action	New State
st	0	#	#400231##	Right	st
st	1	4	#400231##	Right	q4
q4	2	0	#400231##	Right	q4
q4	3	0	#400231##	Right	q4
q4	4	2	#400231##	Right	q4
q4	5	3	#400231##	Right	q4
q4	6	1	#400231##	Right	q4
q4	7	#	#400231##	Write 4	ha
ha	7	4	#4002314#	Halt	ha

Detecting Palindromes

- Input tape: `...##010##...`
- Ideas:
 - For each digit at the beginning, find the corresponding digit at the end.
 - If equal, replace corresponding digits by #.
 - Write 1 if the entire string has been replaced by #.
 - Erase string and write 0 if at some time corresponding digits are different.

Turing Table for 2 digits

Situations which cannot come up are marked by "-----"

st: start, br: branch, zk: zero, ok: one,

bk: back, ek: erase, ha: halt.

Old State	#	0	1
st	br R	-----	-----
br	ha 1	z1 R	o1 R
z1	z2 L	z1 R	z1 R
z2	-----	b1 #	e1 1
o1	o2 L	o1 R	o1 R
o2	-----	e1 0	b1 #
b1	b2 L	-----	-----
b2	b3 R	b2 L	b2 L
b3	ha 1	st #	st #
e1	ha 0	e2 #	e2 #
e2	e1 L	-----	-----
ha	-----	ha H	ha H

Runtime Example 010 - Part 1

State	Position	Symbol	String	Action	New State
st	0	#	#010#	Right	br
br	1	0	#010#	Right	z1
z1	2	1	#010#	Right	z1
z1	3	0	#010#	Right	z1
z1	4	#	#010#	Left	z2
z2	3	0	#010#	Write #	b1
b1	3	#	#01##	Left	b2
b2	2	1	#01##	Left	b2
b2	1	0	#01##	Left	b2
b2	0	#	#01##	Right	b3
b3	1	0	#01##	Write #	st

Runtime Example 010 - Part 2

State	Position	Symbol	String	Action	New State
st	1	#	##1##	Right	br
br	2	1	##1##	Right	o1
o1	3	#	##1##	Left	o2
o2	2	1	##1##	Write #	b1
b1	2	#	#####	Left	b2
b2	1	#	#####	Right	b3
b3	2	#	##1##	Write 1	ha
ha	2	1	##1##	Halt	ha

There is no requirement where exactly the output appears on the tape.
It must only be unique in the sense that no other digits are on the tape.

Runtime Examples 011

State	Position	Symbol	String	Action	New State
st	0	#	#011#	Right	br
br	1	0	#011#	Right	z1
z1	2	1	#011#	Right	z1
z1	3	1	#011#	Right	z1
z1	4	#	#011#	Left	z2
z2	3	1	#011#	Write 1	e1
e1	3	1	#011#	Write #	e2
e2	3	#	#01##	Left	e1
e1	2	1	#01##	Write #	e2
e2	2	#	#0###	Left	e1
e1	1	0	#0###	Write #	e2
e2	1	#	#####	Left	e1
e1	0	#	#####	Write 0	ha
ha	0	0	0#####	Halt	ha

<http://www.comp.nus.edu.sg/~gem1501/assignment14.html>

Variants of Turing Machines

- Turing Machines might use additional symbols on the tape.
- Turing Machines might by the same command write a symbol and then move the head (as in Harel's book).
- Turing Machines might compute $\{0, 1\}$ -valued functions by having states for "Halt with output 1" and a "Halt with output 0" (as in Harel's book).
- All these models are equivalent.

Universal Turing Machines

There is a Turing machine M with two inputs interpreting the first input e as a program. M has the following properties:

- For every two inputs e and x , the function f_e given by e is computable where $f_e(x) = y$ if M with input e, x halts with output y and $f_e(x)$ be undefined if M with input e, x does not halt.
- Whenever g is a computable partial function then there is an e with $g = f_e$;
- Whenever N is a Turing machine with one input then one can compute from the description of N an index e such that f_e is the function computed by N .

The power of Turing Machines

- Turing machines can detect palindromes, add, multiply and so on.
- What other problems can they solve?
- Answer: Turing machines can solve any effectively solvable algorithmic problem!
- A function is computable iff there is a Turing machine computing it.

Characterizing Recursively Enumerable Sets

- A nonempty set A is recursively enumerable iff there is a Turing machine which computes a total function f with range A .
- A set A is recursively enumerable iff there is a Turing machine which halts iff the input is from A .
- The same could be done with Java Script programmes in place of Turing machines.
- Are there other simple machine models?

Overview

- Turing Machines
- **Multiple Stack Machines**
- Church's Thesis
- Numerical Computation:
Counter Programmes
- Restricted Models:
One Stack Machines and Finite Automata

Example: Multiple Stack Machines

- Machine with k stacks, k a constant.
- Machine can pop top element of some stack.
- Machine can push an element onto some stack.
- Besides that, machine has finitely many states to store current information.
- Input either read step by step from a source or in one stack.

Detecting Palindromes

Machine with three stacks u, v, w and two character variables a, b .
Initially input in stack u , stacks v and w are empty.

Algorithm:

```
function ispalindrome(u)
  { while (u not empty)                                // copying and
    { a = pop(u); push(v,a); push(w,a); }             // reverting u to v,w
  while (v not empty)                                  // copying and
    { a = pop(v); push(u,a); }                       // reverting v to u
  b = a;                                               // copying a to b
  while ((u not empty) and (a == b))                 // comparing u,w
    { a = pop(u); b = pop(w); }
  if (a == b) { return(true); } else { return(false); } }
```

Runtime Example

stacks	u	v	w	variables	a	b	
Input 1	10201	-	-	-	-	-	
u -> v,w	-	10201	10201	1	-		
v -> u	10201	-	10201	1	-		
a -> b	10201	-	10201	1	1		
comparing	1020	-	1020	1	1		
	102	-	102	0	0		
	10	-	10	2	2		
	1	-	1	0	0		
	-	-	-	1	1		>> a palindrome
Input 2	1011201	-	-	-	-	-	
u -> v,w	-	1021101	1021101	1	-		
v -> u	1011201	-	1021101	1	-		
a -> b	1011201	-	1021101	1	1		
comparing	101120	-	102110	1	1		
	10112	-	10211	0	0		
	1011	-	1021	2	1		>> not a palindrome

Detecting Palindromes, Summary

- Multi Stack Machines can also detect palindromes.
- They are even faster and need time $O(n)$.
- Turing Machine cannot detect palindromes in time $O(n)$.
- Can Multi Stack Machines do the same operations as Turing machines?

Simulating Turing Machines

2 Stack Machines can simulate Turing machines.

Popping an empty stack gives the symbol #.

stack v	var a	stack w	var b
left part of tape	head	right part of tape	state

Initialization

v is empty, a = #, b = st, w = input word.

Operations simulated as follows:

Left Push a onto stack w, Pop a from stack v

Right Push a onto stack v, Pop a from stack w

Write c Let a = c

Halt Halt the stack machine

Update of state:

b = Turing Table Entry for (old value of b, old value of a)

Equivalence

- Just seen: Multi Stack machines can simulate Turing machines
- Turing machines can also simulate Multi Stack machines.
- The models for computation given by Turing machines and Multi Stack machines are equivalent.

Overview

- Turing Machines
- Multiple Stack Machines
- **Church's Thesis**
- Numerical Computation:
Counter Programmes
- Restricted Models:
One Stack Machines and Finite Automata

Church's Thesis

- Named after Alonzo Church (and sometimes also after Alan Turing).
- **Thesis:** A function is computable in the intuitive sense iff it is computable by a Turing machine.
- More precisely, any reasonable formalization of the notion “computable function” will lead to a concept equivalent to being computable by a Turing machine.
- The same applies for the notions of “decidable (or recursive) set” and “recursively enumerable set”.
- Furthermore, computation can be formalized in the numerical framework as well as in the symbolic one (by manipulating strings). Both frameworks are transformable into each other.

Some Evidence for Church's Thesis

- Multi Stack machines and Turing machines can simulate each other.
- Furthermore, a function is polynomial time computable iff this holds for one of the following models:
 - Turing machines;
 - Multi Stack machines;
 - Java Script programmes.
- The notions of computation and even of polynomial time computation are independent of the chosen machine model.

Why “Thesis” ?

- **Theorem**

A function is computable by a Turing machine iff it is computable by a Multi Stack machine.

- **Thesis**

A function is computable by a Turing machine iff it is computable by any reasonable model of computation.

- **Remark**

The words “reasonable model” or “intuition” used in Church’s Thesis are not formally defined.

Thus “Thesis” and not “Theorem”.

Overview

- Turing Machines
- Multiple Stack Machines
- Church's Thesis
- **Numerical Computation:**
Counter Programmes
- Restricted Models:
One Stack Machines and Finite Automata

A numerical model: Counter programmes

- Variables can hold any natural number
- Operations: $x = 0$, $x = y$, $x = y + 1$, $x = y - 1$
- Control statement: `if $x == 0$ goto line g`
Also unconditional Goto Statements permitted

Adding numbers

```
function add(x,y) // counter machine format
```

```
  1: z = 0;
  2: if x == 0 goto 6;
  3: x = x-1;
  4: z = z+1;
  5: goto 2;
  6: if y == 0 goto 10;
  7: y = y-1;
  8: z = z+1;
  9: goto 6;
10: return(z);
```

```
function add(x,y) // normal Java Script format for same programme
```

```
{ var z; z = 0;
  while (x>0) { x = x-1; z = z+1; }
  while (y>0) { y = y-1; z = z+1; }
  return(z); }
```

Multiplying numbers

```
function mult(x,y)
```

```
  1: z = 0;  
  2: v = x;  
  3: if v == 0 then goto 11;  
  4: w = y;  
  5: if w == 0 then goto 9;  
  6: w = w-1;  
  7: z = z+1;  
  8: goto 5;  
  9: v = v-1;  
10: goto 3;  
11: return(z);
```

```
function mult(x,y)
```

```
  { var z; var v; var w; z = 0; v = x;  
    while (v>0) { w = y; v = v-1;  
                  while (w>0) { z = z+1; w = w-1; } }  
    return(z); }
```

Counter Programmes

- Counter programmes and Turing machines are equivalent.
- One can always translate one to the other, they can simulate each other.
- Counter programmes are slower than Turing machines: adding n -digit numbers is polynomial in n for Turing machines and exponential in n for counter programmes.
- As very simple models, Turing machines are used in lower-bound proofs on computation speed.
- Counter programmes which can also add, subtract and compare variables give the same notions of polynomial time computable functions as Turing machines.

Overview

- Turing Machines
- Multiple Stack Machines
- Church's Thesis
- Numerical Computation:
Counter Programmes
- **Restricted Models:**
One Stack Machines and Finite Automata

One Stack machines

- Exactly one stack permitted;
- Input is read from separate source from left to right;
- Furthermore finite control (finitely many variables for one symbol, finitely many states)
- Some states are accepting and some rejecting.
- Input word in set A iff last state of computation is “accepting” .

Brackets

Algorithm to check correctness of bracket-expression

```
var stack = new Array("#"); var a; var b;
loop { a = next input symbol;
      switch(a)
        { case "(", case "[": stack.push(a); break;
          case ")": b = stack.pop();
                    if (b != "(") { reject; abort; } break;
          case "]": b = stack.pop();
                    if (b != "[") { reject; abort; } break;
          default: break; }
      if (stack == "#") then accept else reject; }
```

Algorithm accepts iff it is not aborted and the state is accepting after processing all input symbols.

Runtime Example

Input	Stack after current input	Accept / Reject / Reject forever (current) (aborted case)
(2+3*(5+8)+[10	#([Reject
-25*3]-18)*12	#	Accept
*x*y+23*(235+	#(Reject
[[[(12+3)*5+7*	#([[Reject
234567]-13]+2*(#([Reject
12-17)]*3+5)+8*	#	Accept
[[*17-3))+((5*2	#[Aborted, Reject forever
.....	#[Aborted, Reject forever

A more sophisticated version of this algorithm is implemented in syntax-checkers of compilers and browsers.

Counting 0's and 1's

Algorithm to check whether 0's and 1's are same quantity

```
var stack = new Array(); var a; var b;
loop { a = next input symbol;
      if ((a == "0") or (a == "1"))
        { if (stack.length == 0) { stack.push(a); }
          else
            { b = stack.pop();
              if (a == b) { stack.push(b); stack.push(a); } } }
      if (stack.length == 0) { accept; } else { reject; } }
```

Assume that n zeroes and m ones have seen:

If $n > m$ then stack has $n-m$ zeroes and state rejecting;

If $n < m$ then stack has $m-n$ ones and state rejecting;

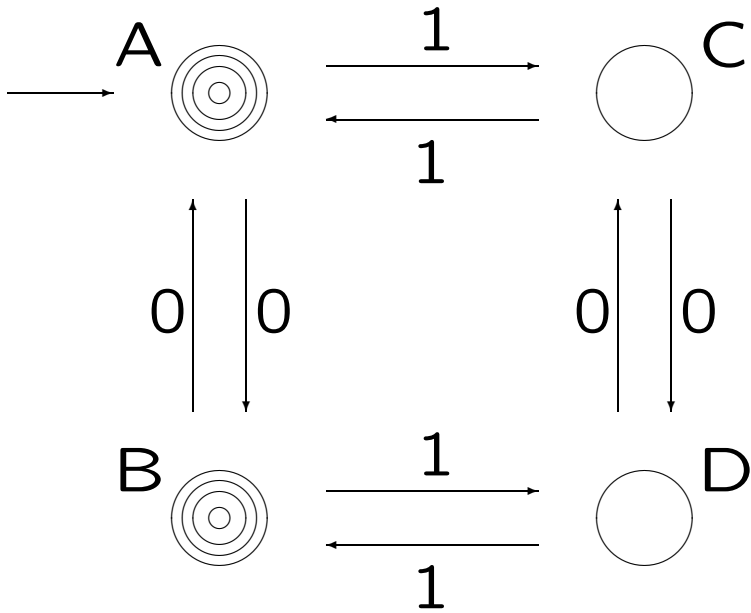
If $n == m$ then stack is empty and state accepting.

So stack is 000 after input 01112020001200.

Limitations

- One stack machines cannot count independently three things: Set of all numbers containing as many 0 as 1 as 2 cannot be accepted.
- One stack machines cannot check in programme-texts whether the names of declared and used variables are the same. This part of a syntax check needs more enhanced methods.

Finite Automata



A word is accepted iff it has an even number of 1s.

State:	A	B	C	D	Starting: A
Successor when 0:	B	A	D	C	Accepting: A, B
Successor when 1:	C	D	A	B	Rejecting: C, D

Runtime Example

Input read so far	State
-	A (accepting)
0	B (accepting)
00	A (accepting)
000	B (accepting)
0001	D (rejecting)
00010	C (rejecting)
000101	A (accepting)
0001011	C (rejecting)
00010111	A (accepting)
000101110	A (accepting)

See also <http://www.comp.nus.edu.sg/~gem1501/assignment16.html>

Power of Finite Automata

- Cannot count
- Only finite control
- Can easily be modified and analyzed
- Every task has a minimal finite automaton

Finite Automata Cannot Count!

No finite automaton can decide whether its input sequence contains precisely the same number of 0's and 1's.

- There are two different numbers n and m such that automaton is in same state after seeing 0^n and 0^m .
- For any k , automaton is then also in the same state after seeing $0^n 1^k$ and $0^m 1^k$.
- Automaton either accepts both $0^n 1^n$ and $0^m 1^n$ or rejects both.
- Thus automaton does not count 0's and 1's properly, even if all 0's come first and all 1's second.

Application of Finite Automata

- Describing Processes with finitely many states.
- Often used as basic algorithms: In each state, a certain subprogramme is activated.
- Example: Drawing Money from Automated Teller Machine
 1. Read Card.
 2. Read PIN.
 3. If PIN incorrect goto 8.
 4. Read amount of money to withdraw.
 5. If money not in account goto 8.
 6. If money not in machine goto 9.
 7. Hand out card, hand out money and goto 1.
 8. Print excuse, hand out card and goto 1.
 9. Print big excuse, hand out card and goto 1.

Summary

- Turing Machines
- Multiple Stack Machines
- Church's Thesis
- Numerical Computation:
Counter Programmes
- Restricted Models:
One Stack Machines and Finite Automata

Next Week: Parallelism

- Parallel Computing
- Parallel Sorting
- Weighted Averages
- Sequential Polynomial Space, Parallel Polynomial Time
- Nick's Class
- Coordinating Parallelism
- New Models of Computing