Markov Algorithms

Other Notions of Computability

- Many other notions of computability have been proposed, e.g.
  - *(Type 0 a.k.a. Unrestricted)* Grammars
  - Partial Recursive Functions
  - Lambda calculus
  - ***Markov Algorithms***
  - Post Algorithms
  - Post Canonical Systems,
- • All have been shown equivalent to Turing machines by simulation proofs

Markov Algorithms

- A Markov Algorithm over an alphabet A is a finite ordered sequence of productions x→y, where x, y ∈ A*. Some productions may be "Halt" productions. e.g.

abc → b

ba → x (halt)

Execution proceeds as follows:

1. Let the input string be w

2. The productions are scanned in sequence, looking for a production x → y where x is a substring of w

3. The left-most x in w is replaced by y

4. If the production is a halt production, we halt

5. If no matching production is found, the process halts

6. If a replacement was made, we repeat from step 2.

- Note that a production $\Lambda \rightarrow$ a inserts a at the start of the string.
- What does this Markov algorithm do?

aba → b

ba → b

b → a

aabaaa

abaa

ba

b

a

Example – Binary to Unary

1. "|0" -> "0||"
2. "1" -> "0|"
3. "0" -> ""

"0|01"
"00||1"
"00||0|"
"00|0|||"
"000|||||"
"00|||||"
"0|||||"
"|||||"

Input "101"

- Example from wikipedia http://en.wikipedia.org/wiki/Markov_algorithm

Other Notions of Computability

- Many other notions of computability have been proposed, e.g.
  - *(Type 0 a.k.a. Unrestricted)* ***Grammars***
  - Partial Recursive Functions
  - Lambda calculus
  - Markov Algorithms
  - Post Algorithms
  - Post Canonical Systems,
- • All have been shown equivalent to Turing machines by simulation proofs

Grammars

- We can extend the notion of context-free grammars to a more general mechanism
- An (unrestricted) grammar G = (V,Σ,R,S) is just like a CFG except that rules in R can take the more general form $\alpha \rightarrow \beta$ where α,β are **arbitrary strings of terminals and variables.** α must contain at least one variable (or nontermial).
- If $\alpha \rightarrow \beta$ then $u\alpha v \Rightarrow u\beta v$ ("yields") in one step
- Define $\Rightarrow^*$ ("derives") as reflexive transitive closure of $\Rightarrow$.

Example - Counting

- Grammar generating {w ∈ {a,b,c}* | w has equal numbers of a's, b's, and c's }

- G = ({S,A,B,C},{a,b,c},R,S) where R is

S → Λ

S → ABCS

AB → BA AC → CA BC → CB

BA → AB CA → AC CB → BC

A → a B → b C → c

Try generating ccbaba

Example: $\{a^{2^n}, n \geq 0\}$

- Here's a set of grammar rules

1. S → a
2. S→ ACaB
3. Ca → aaC
4. CB → DB
5. CB → E
6. aD → Da
7. AD →AC
8. aE → Ea
9. AE → Λ

Try generating 2^3 a's

S
ACaB
AaaCB
AaaDB
AaDaB
ADaaB
ACaaB
AaaCaB
AaaaaCB
AaaaaDB

(Unrestricted) Grammars and Turing machines have equivalent power

- For any grammar G we can find a TM M such that L(M) = L(G).
- For any TM M, we can find a grammar G such that L(G) = L(M).

Other Notions of Computability

- Many other notions of computability have been proposed, e.g.
  - (Type 0 a.k.a. Unrestricted) Grammars
  - ***Partial Recursive Functions***
  - Lambda calculus
  - Markov Algorithms
  - Post Algorithms
  - Post Canonical Systems,
- • All have been shown equivalent to Turing machines by simulation proofs

Computation using Numerical Functions

- We're used to thinking about computation as something we do with **numbers (e.g.** on the naturals)

- What kinds of functions from numbers to numbers can we actually compute?

- To study this, we make a very careful selection of building blocks

Primitive Recursive Functions

- The primitive recursive functions from ℕ x ℕ x ... x ℕ → ℕ are those built from these primitives:
  - zero(x) = 0
  - succ(x) = x+1
  - π k,j (x1,x2,...,xk) = xj for 0 < j ≤ k
- using these mechanisms:
  - Function composition, and
  - Primitive recursion

Function Composition

- Define a new function f in terms of functions h and g1, g2, ..., gm as follows:

  f(x1,...xn) = h(g1(x1,...,xn),...gm(x1,...,xn))

Example: f(x) = x + 3 can be expressed using two compositions as f (x) = succ(succ(succ(x)))

Primitive Recursion

- Primitive recursion defines a new function f in terms of functions h and g as follows:

  f(x1, ..., xk, 0) = h(x1,...,xk)

  f(x1, ..., xk, succ(n)) = g(x1,...,xk, n, f(x1,...,xk,n))

Many ordinary functions can be defined using primitive recursion, e.g.

  add(x,0) = π1,1(x)

  add(x, succ(y)) = succ(π3,3(x, y, add(x,y)))

More P.R. Functions

- For simplicity, we omit projection functions and write 0 for zero(_) and 1 for succ(0)

  - add(x,0) = x
    add(x,succ(y)) = succ(add(x,y))
  - mult(x,0) = 0
    mult(x,succ(y)) = add(x,mult(x,y))
  - factorial(0) = 1
    factorial(succ(n)) = mult(succ(n),factorial(n))
  - exp(n,0) = 1
    exp(n, succ(n)) = mult(n,exp(n,m))
  - pred(0) = 0
    pred(succ(n)) = n

- Essentially all practically **useful arithmetic** functions are primitive recursive, but...

Ackermann's Function is not Primitive Recursive

- A famous example of a function that is clearly well-defined but not primitive recursive

```
A(m, n)=
   if m0 then n+1
   else if n=0 then A(m−1, 1)
   else A(m−1, A(m,n−1))
```

This function grows extremely fast!

Values of *A*(*m*, *n*)

*m**n*	0	1	2	3	4	n
0	1	2	3	4	5	$n + 1$
1	2	3	4	5	6	$n + 2 = 2 + (n + 3) - 3$
2	3	5	7	9	11	$2n + 3 = 2 \cdot (n + 3) - 3$
3	5	13	29	61	125	$2^{(n+3)} - 3$
4	13	65533	$2^{65536} - 3$	$2^{2^{65536}} - 3$	*A*(3, *A*(4, 3))	$\underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}_{n+3 \text{ twos}} - 3$
5	65533	$\underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}_{65536 \text{ twos}} - 3$	*A*(4, *A*(5, 1))	*A*(4, *A*(5, 2))	*A*(4, *A*(5, 3))	*A*(4, *A*(5, n-1))
6	*A*(5, 1)	*A*(5, *A*(6, 0))	*A*(5, A(6, 1))	*A*(5, *A*(6, 2))	*A*(5, *A*(6, 3))	*A*(5, *A*(6, n-1))

A is not primitive recursive

- Ackermann's function grows faster than any primitive recursive function, that is:

- for *any primitive recursive function f, there is* an *n such that*

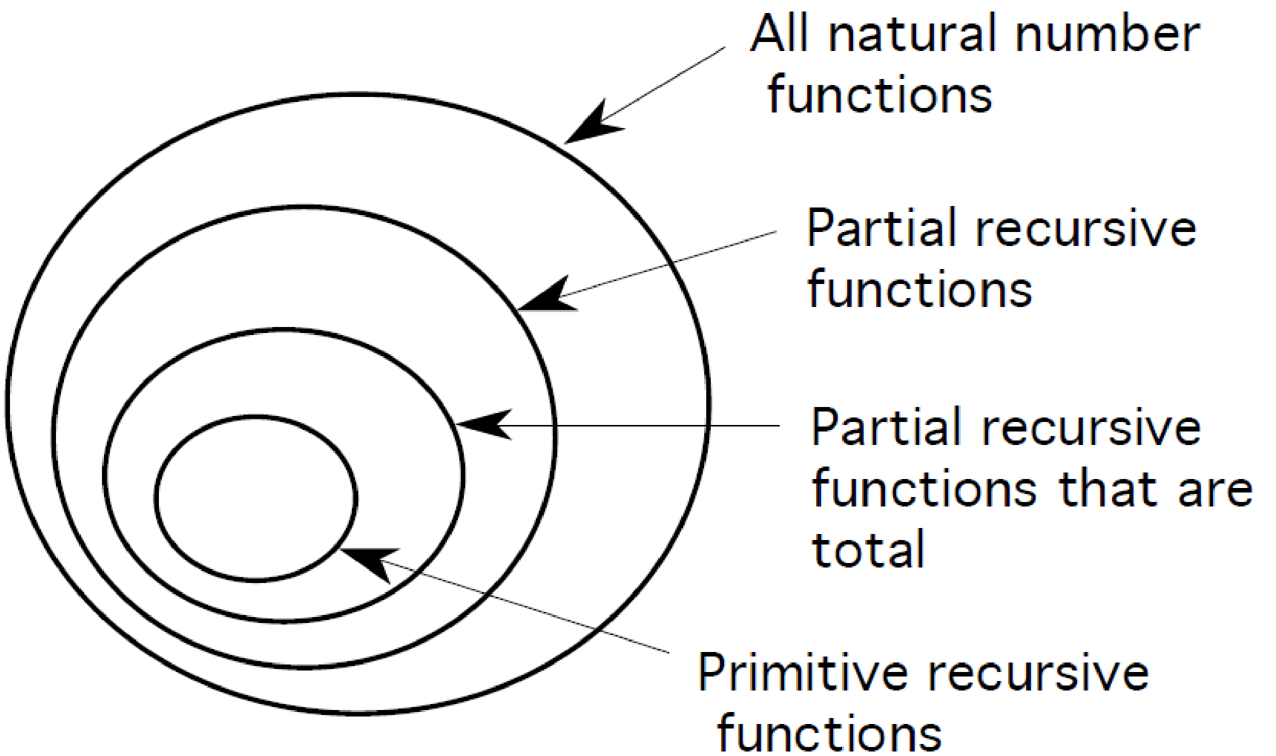
- *A(n, x) > f x*

- So *A can't be primitive recursive*

Partial Recursive Functions

- *A belongs to class of* ***partial recursive*** **functions, a superset of the primitive recursive** functions.
- Can be built from primitive recursive operators & new **minimization operator**
  – Let *g be a (k+1)-argument function.*
  – Define *f(x1,…,xk) as the* ***smallest m such that g(x1,…,xk,m)*** = 0 (if such an m exists)
  – Otherwise*, f(x1,…,xn) is undefined*
  – We write *f(x1,…,xk) = μm.[g(x1,…,xk,m) = 0]*
  – Example: *μm.[mult(n,m) = 0] = zero(_)*

Hierarchy of Numeric Functions



Turing-computable functions

- To formalize the connection between partial recursive functions and Turing machines, we need to describe how to use TM's to compute functions on $\mathbb{N}$.
- We say a function f : $\mathbb{N} \times \mathbb{N} \times \ldots \times \mathbb{N} \rightarrow \mathbb{N}$ is **Turing-computable** if there exists a TM that, when started in configuration $q_0 1^{n1} \sqcup 1^{n2} \sqcup \ldots \sqcup 1^{nk}$, halts with just $1^{f(n1,n2,\ldots nk)}$ on the tape.
- **Fact: f is Turing-computable iff it is partial recursive.**