MA50B Complejidad Computacional

Random Access Machine

por Marcos Kiwi

Una *Random Access Machine* (RAM) consiste de un programa que actúa sobre una estructura de datos. El *programa* $\Pi = (\pi_1, \pi_2, \ldots, \pi_m)$ es una secuencia finita de *instrucciones*, donde cada instrucción π_i es del tipo que se muestra en la Fig. 1. La estructura de datos sobre la que actúa Π es una secuencia infinita de *registros*. Cada registro contiene un entero, posiblemente negativo. El contenido del i-ésimo registro se denota r_i. El registro 0 es especial y se llama *acumulador*. En el acumulador es donde se ejecuta toda la aritmética y el cálculo lógico.

Instrucción	Operando	Interpretación
READ	j	$r_0 \leftarrow \rho_j$
READ	$\uparrow j$	$r_0 \leftarrow \rho_{r_j}$
STORE	j	$r_j \leftarrow r_0$
STORE	$\uparrow j$	$r_{r_j} \leftarrow r_0$
LOAD	$x \in \{j, \uparrow j, = j\}$	$r_0 \leftarrow val(x)$
ADD	$x \in \{j, \uparrow j, = j\}$	$r_0 \leftarrow r_0 + val(x)$
SUB	$x \in \{j, \uparrow j, = j\}$	$r_0 \leftarrow r_0 - val(x)$
HALF		$r_0 \leftarrow \lfloor r_0/2 \rfloor$
JUMP	j	$\kappa \leftarrow j$
JPOS	j	Si $r_0 > 0$ entonces $\kappa \leftarrow j$
JZERO	j	Si $r_0 = 0$ entonces $\kappa \leftarrow j$
JNEG	j	Si $r_0 < 0$ entonces $\kappa \leftarrow j$
HALT		$\kappa \leftarrow 0$

Figura 1: *Instrucciones de una RAM: j es un entero, r_j es el contenido del registro j, ρ_j es la j-ésima entrada, si x es j entonces $val(x) = r_j$, si x es $\uparrow j$ entonces $val(x) = r_{r_j}$, si x es $= j$ entonces $val(x) = j$, κ es el contador de programa (salvo que se señale lo contrario todas las instrucciones cambian κ a $\kappa + 1$).*

Durante cada paso del cálculo de una RAM, se ejecuta una instrucción π_κ, donde κ es un entero no–negativo denominado *contador de programa*. El contador de programa es otro parámetro que evoluciona mientras la RAM ejecuta. Al comienzo de la ejecución de la RAM $\kappa = 1$ y al final de la ejecución $\kappa = 0$.

La entrada de un programa RAM es una tupla de la forma $I = (\rho_1, \ldots, \rho_k)$ donde ρ_j es un entero. Todos los registros están en 0 inicialmente. El resultado del cálculo de una RAM es el contenido del acumulador en el momento que la RAM se detiene (o indefinido si esta no se detiene). Cada instrucción puede causar un cambio en los contenidos de un registro y del contador de programa.

Una RAM parte ejecutando la primera instrucción π_1, implementa los cambios dictados por la misma, y luego ejecuta las instrucción π_κ, donde κ es el nuevo valor del contador de programa, y así sigue ejecutando salvo que se detenga.

Example 1 *Notar que en la Fig. 1 está ausente una instrucción para multiplicar. Esto no es una gran pérdida pues el programa que se incluye más abajo calcula el producto de dos enteros no–negativos. De hecho, la inclusión de la operación multiplicación entre las instrucciones de una RAM tiene consecuencias indeseables (este punto lo discutiremos más extensamente en la Observación 1).*

El siguiente programa RAM calcula el producto de dos números no negativos i_1 *y* i_2*. La entrada del programa es* $I = (i_1, i_2)$*.*

1	*READ 1*	*Asigna al acumulador el valor de* i_1
2	*STORE 1*	*Asigna al registro 1 el valor del acumulador, i.e.,* i_1
3	*STORE 5*	*Asigna al registro 5 el valor del acumulador, i.e.,* i_1
4	*READ 2*	*Asigna al acumulador el valor de* i_2
5	*STORE 2*	*Asigna al registro 2 el valor del acumulador, i.e.,* i_2
6	*HALF*	*Comienza* k*-ésima iteración,* $k = 1, 2, \ldots$
7	*STORE 3*	*Asigna al registro 3 el valor* $\lfloor i_2/2^k \rfloor$
8	*ADD 3*	*Se incrementa el acumulador en* $\lfloor i_2/2^k \rfloor$
9	*SUB 2*	*Se decrementa el acumulador en* i_2
10	*JZERO 14*	*Si el* k*-ésimo bit más significativo de* i_2 *es* 0 *ir a la instrucción 14*
11	*LOAD 4*	*Instrucciones 11–13, asignan al registro 4 el valor* $i_1 \cdot (i_2 \bmod 2^{k+1})$
12	*ADD 5*	
13	*STORE 4*	
14	*LOAD 5*	*Instrucciones 14–16, asignan al registro 5 el valor* $i_1 2^{k+1}$
15	*ADD 5*	
16	*STORE 5*	
17	*LOAD 3*	
18	*JZERO 20*	*Si* $\lfloor i_2/2^k \rfloor = 0$ *(estamos listos) saltar a la instrucción 20*
19	*JUMP 5*	*Continuar iterando*
20	*LOAD 4*	*Asignar al acumulador el resultado (que está en el registro 4)*
21	*HALT*	

El programa ejecuta $\lceil \log i_2 \rceil$ *veces las instrucciones 5 a 19. Al comienzo de la* k*-ésima iteración (comenzando con* $k = 1$*) el tercer registro contiene* $\lfloor i_2/2^k \rfloor$*, el quinto registro contiene* $i_1 2^k$ *y el cuarto registro contiene* $i_1 \cdot (i_2 \bmod 2^k)$*. Cada iteración mantiene este invariante. Al final de cada iteración se determina si el tercer registro contiene* 0*. Si este es el caso, la ejecución termina cargando el resultado del cálculo en el acumulador. Observar que el anterior programa calcula el producto de dos enteros no-negativos* m *y* n *en* $O(\log(\max\{m, n\}))$ *pasos.*

La característica más relevante que posee una RAM (y que no posee una mT), es que en una RAM se puede accesar en un sólo paso cualquier unidad de memoria (registro), mientras que en una mT cualquier unidad de memoria (una celda de una cinta) no se puede accesar tan fácilmente.

A pesar que las operaciones aritméticas de las RAM pueden involucrar enteros arbitrariamente grandes contaremos cada instrucción ejecutada por una RAM como un solo paso. Lo anterior podría parecer poco realista, puesto que una operación como ADD que involucre enteros grandes tiene un costo igual a una operación ADD que sólo involucre números pequeños. De hecho pareciera que un costo más realista para una operación ADD sería el logaritmo del valor absoluto del sumando más grande. Sin embargo, el conjunto de instrucciones que hemos adoptado para una RAM nos asegura que asumir que cada instrucción tiene costo unitario no tiene consecuencias demasiado indeseables. Esta última aseveración es una consecuencia del resultado que enunciamos en el Theorem 1.

Necesitamos de logaritmos para medir el tamaño de la entrada. Sea n un entero denotamos por $l(n)$ el número de bits que se requieren para representar el número en binario. Luego, $l(n) \stackrel{\text{def}}{=} \lceil \log(|n|+1) \rceil + 1$ (el logaritmo es base 2 y la adición de 1 se debe a que necesitamos un bit adicional para representar el signo de n). El *largo* o *tamaño* de la entrada $I = (\rho_1, \ldots, \rho_k)$ lo definimos como $l(I) \stackrel{\text{def}}{=} \sum_{j=1}^{k} l(\rho_j)$.

Como es usual medimos la complejidad de tiempo de una RAM en función del largo de la entrada. En particular, sea f una función eventualmente positiva. Decimos que una RAM con programa Π es a tiempo $f(n)$ si cualquiera sea la entrada de largo n el programa Π se detiene en a lo más $f(n)$ pasos.

Observación 1 *Observar que si incluyésemos la instrucción para multiplicar enteros entre las instrucciones permitidas en un programa RAM podríamos calcular en la entrada $n = 2^k$ el número 2^n en tiempo $O(k) = O(l(n)) = O(\log(n))$ (calculando iterativamente 2^{2^0}, 2^{2^1}, $2^{2^2}, \ldots, 2^{2^k}$). Luego podríamos calcular 2^n en tiempo polinomial en el tamaño de la entrada. Pero $l(2^n) = O(n)$, i.e., es exponencial en el tamaño de la entrada. Un modelo de computación que calcule una salida de tamaño exponencial en el tamaño de la entrada en tiempo polinomial en el tamaño de la entrada es irrealista. Es por esto que omitimos la instrucción para multiplicar entre las instrucciones permitidas en un programa RAM.*

Observar que todo programa Π define una función ϕ_Π que toma valores enteros y cuyo dominio es el conjunto de posibles entradas. El valor de ϕ_Π en una entrada I se denota por $\phi_\Pi(I)$ y es igual al contenido del acumulador una vez que el programa para o ∞ en caso que el programa no se detenga en la entrada I.

Analogamente podemos ver una RAM como una máquina que reconoce lenguajes. En efecto, consideremos una RAM con programa Π. Sea $\Sigma = \{\sigma_1, \ldots, \sigma_k\}$ un alfabeto. Decimos que una RAM con programa Π acepta $\sigma = \sigma_{i_1} \cdots \sigma_{i_s} \in \Sigma^*$, si $\phi_\Pi(I) = 1$ donde $I = (i_1, \ldots, i_s, 0)$ (el 0 le permite al programa RAM identificar el fin de la entrada). El conjunto de palabras en Σ^* aceptadas por una RAM con programa Π lo denotamos por L_Π y si $L = L_\Pi$ para algún programa RAM Π, decimos que L es reconocido por una RAM. Decimos que un lenguaje L es decidido por una RAM con programa Π si $L = L_\Pi$ y cualquiera sea la entrada, Π se detiene.

Teorema 1 *Si $L \in DTIEMPO(f(n))$, entonces existe un programa para una RAM que decide L en tiempo $O(f(n))$.*

El converso del Theorem 1 también se verifica. En particular se tiene que,

Teorema 2 *Si una RAM con programa Π calcula la función ϕ_Π en tiempo $f(n)$, entonces existe una mT que calcula ϕ_Π en tiempo $O(f(n)^3)$.*

Los dos teoremas enunciados nos dicen que los modelos de mT y RAM son polinomialmente equivalentes. Lo anterior implica que para establecer que un algoritmo puede implementarse en tiempo polinomial en una mT basta dar un programa RAM que implemente dicho algoritmo en tiempo polinomial. Lo anterior valida usar pseudo–código para describir algoritmos.

Para mayor información acerca de RAM's ver [Pap94, 36–45].

Referencias

[Pap94] C. H. Papadimitriou. *Computational complexity.* Addison-Wesley Publishing Company, first edition, 1994.