Técnicas Avanzadas de Programación

Tema 1:

Presentación del curso. Computabilidad. P vs NP. Complejidad en tiempo

Prof. Rhadamés Carmona (20/03/2019)

Agenda

- Presentación del curso [done!]
- Computabilidad
  - P versus NP
  - Conjetura del milenio: P=NP
  - NP-completo, NP-hard
- Repaso de Complejidad en Tiempo
  - Complejidad en ciclos
  - Complejidad de algoritmos recursivos

Computabilidad

- Un problema computable es aquella abstracción de la realidad que tiene solución algorítmica.
- Los problemas computables se pueden escribir como un problema de decisión.
- Por ejemplo, existe un camino de mínimo costo entre un par de nodos a,b en un grafo? Para responder a esta pregunta habría que resolver el problema original de camino de costo mínimo.

Computabilidad

- Los problemas los podemos dividir en:
  - La clase de los problemas decidibles o computables, que admiten un algoritmo que los resuelve; es decir, para todo posible valor de las variables de entrada, el algoritmo se detiene eventualmente dando como salida la solución al problema.
  - La clase de los problemas indecidibles o no computables, que no admiten un algoritmo que los resuelva.

Computabilidad

- Un problema indecidible no puede ser resuelto con un algoritmo aún si se dispone de espacio y tiempo ilimitado.
- Ejemplos:
  - Problema de la parada de Turing: “Dada la descripción de un programa y su entrada inicial, determinar si el programa cuando se ejecuta termina en algún momento o se ejecuta para siempre sin parar”

Computabilidad

- Un problema indecidible no puede ser resuelto con un algoritmo aún si se dispone de espacio y tiempo ilimitado.
- Ejemplos:
  - Dados dos programas, ¿calculan lo mismo?

Computabilidad

- ¿De qué nos sirve un algoritmo si tarda en darnos la solución en 100 años?.
- Antes de utilizar un algoritmo podemos estimar el tiempo que tardará en solucionar el problema y ver si este tiempo es razonable. Pero, ¿cuándo un tiempo es razonable?.

Computabilidad

N/ Función	**10**	**50**	**100**	**300**	**1.000**
5*N	50	250	500	1.500	5.000
$N*\log_2 N$	33	282	665	2.469	9.966
N^2	100	2.500	10.000	90.000	Un millón (7 dígitos)
N^3	1.000	125.000	Un millón (7 dígitos)	27 millones (8 dígitos)	Mil millones (10 dígitos)
2^N	1024	Un número de 16 dígitos	Un número de 31 dígitos	Un número de 91 dígitos	Un número de 302 dígitos
N!	36 millones (7 dígitos)	Un número de 65 dígitos	Un número de 161 dígitos	Un número de 623 dígitos	Inimaginablemente grande
N^N	Diez mil millones (11 dígitos)	Un número de 85 dígitos	Un número de 201 dígitos	Un número de 744 dígitos	Inimaginablemente grande

Computabilidad

- P versus NP
  - Los problemas decidibles pueden ser entonces tratables (polinomiales) o intratables (exponenciales)
  - P = problemas decidibles tratables, resueltos en una Máquina de Turing **Determinística** en tiempo polinómico. Ejemplo, N es primo?. Tiempo lineal.
  - NP = problemas decidibles donde dada la solución, se puede **verificar** que retorna “yes” en tiempo polinomial. Se pueden resolver en una Máquina de Turing **no determinística**. Ejemplo Sudoku.

Computabilidad

- Por qué Sudoku está en NP y no en P?.
- Sudoku convertido a un problema de decisión es: dado un tablero “incompleto”, ese tablero tiene solución?
- Note que dado un tablero completado (posible solución), **verificar** si es una solución válida es un problema polinomial. Sin embargo, tratar de llenar el tablero incompleto determinísticamente es exponencial (backtracking).

Computabilidad

- Otro problema NP es ordenar un arreglo, puesto que “dada una permutación del arreglo”, en O(N) podemos decir si está ordenado o no.
- Está claro que este problema además de ser NP, también es P, pues para ordenar un arreglo “determinísticamente” existen soluciones desde O(N), O(Nlog) y $O(N^2)$. Sin embargo, el problema del Sudoku solo es NP (y no pertenece a P).
- Note entonces que $P \subseteq NP$.

Computabilidad

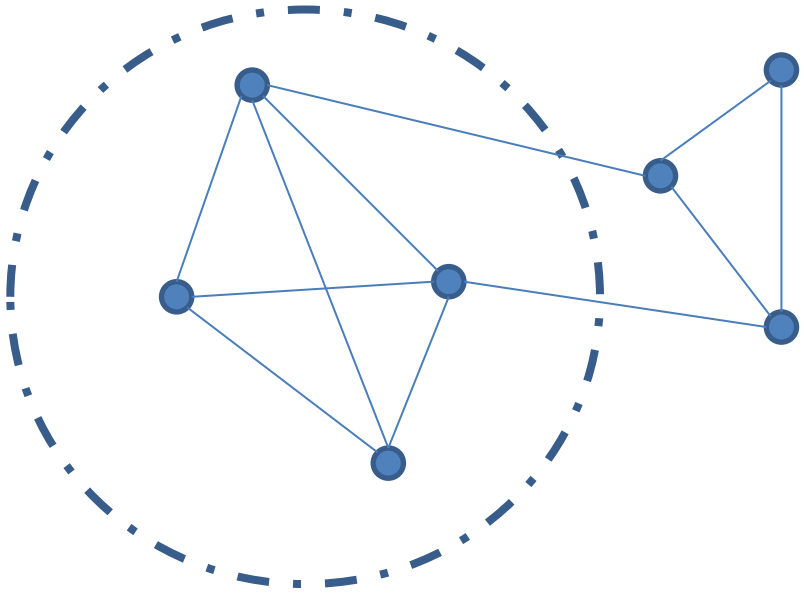
- Dentro de NP hay un subconjunto de problemas (NPC o NP completo) donde todos son difíciles (ninguno polinomial), donde resolver un problema $A \in NP$ es tan difícil como resolver otro problema $B \in NP$. Además, cualquier problema NPC se puede transformar en otro problema NPC en tiempo polinomial, lo cual lo mantiene igual de difícil.

Computabilidad

- Para que un problema sea catalogado como NPC, el problema debe poder ser transformado a algún problema NPC conocido en tiempo polinomial.
- El primer problema que se demostró que era NPC fue Circuit-SAT. Cualquier problema NPC debe poderse transformar a cualquier otro problema NPC (y en particular en este Circuit-SAT) directa o indirectamente en orden polinómico.
- Ejemplos: Knapsack (mochila), coloración de grafo, Maximum Cut (corte en grafo), Set Packing (dada una lista de N subconjuntos, hay K de ellos disjuntos entre sí).
- Ver “Karp's 21 NP-complete”.

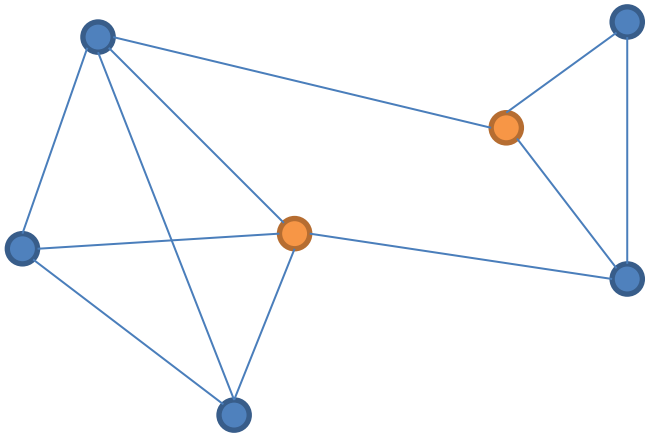
Computabilidad

- **Clique**: El problema del clique consiste en encontrar el subgrafo completo más grande de un grafo no dirigido G(V,E).



Computabilidad

- **Vertex Cover**: el vertex cover de un grafo no dirigido G(E,V) es un conjunto de vértices que permite “cubrir” todos los vértices de V. Un vértice “cubre” a todos los vértices adyacentes a él. El problema del vertex cover consiste en encontrar el vertex cover mínimo de un grafo.



Computabilidad

- Si un problema NPC puede convertirse en tiempo polinomial en un problema P, se resuelve uno de los **problemas del milenio** (demostrar P=NP).
- Tarea 1: buscar un ejemplo de mapeo de un problema NPC en otro NPC. Explicar en a lo sumo 3 páginas tamaño carta ambos problemas, y cómo efectivamente resolviendo un problema se resuelve el otro. No hace falta escribir los algoritmos. Basta la explicación y un ejemplo!.

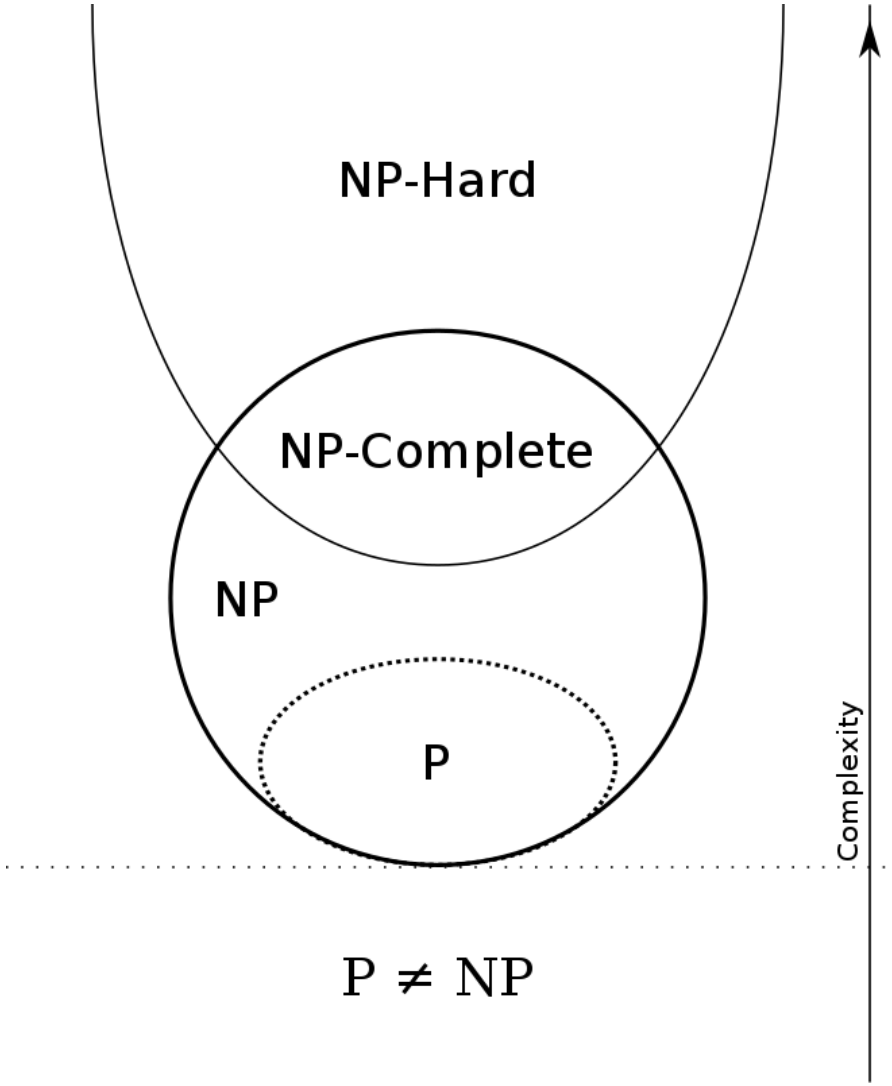
Computabilidad

- NP-hard: informalmente se dice que estos problemas al menos son tan difíciles como el problema más difícil en NP. No pueden ser resueltos en tiempo polinomial. Cada problema NP-hard puede ser reducido a otro problema NP-hard en tiempo polinomial. Aunque parezca lo mismo que NP-completo, NP-hard es más grande que NP-completo porque incluye problemas no decidibles o no computables. De hecho, NP-completo es la intersección entre NP y NP-hard.

Computabilidad

- Ejemplo de un problema no computable (no existe algoritmo que lo resuelve): Décimo problema de Hilbert. “Una ecuación diofántica es la ecuación de los ceros enteros de un polinomio con coeficientes enteros. El décimo problema de Hilbert trata de encontrar un algoritmo que determine si una ecuación diofántica polinómica dada con coeficientes enteros tiene solución entera”. Matiyasevich demostró que no existe dicho algoritmo.

Computabilidad



Repaso de Complejidad en Tiempo

- Sean f y g dos funciones, f,g: +-{0}. Se dice que f=$\mathcal{O}$(g) (f es de orden g) si y solo si existe c$\in\Re^+$ y $n_0 \in$N tal que $\forall$n>n_0 se cumpla f(n)$\leq$ c.g(n).

  (N no incluye el 0).
- Ejemplo: si el tiempo de un algoritmo es T(n)=3n+2, es claro que T(n) $\leq$ 4n, con n_0=1, c=4, por lo cual T(n)=$\mathcal{O}$(n), y f(n)=n. La función f(n) es la tasa de crecimiento. Valores típicos para f(n):
- 1 < logn < n < nlogn < n^2 < n^3 < 2^n < N! < n^n

Repaso de Complejidad en Tiempo

- Usaremos la notación big O para referirnos al peor de los casos.
- Regla de la suma:

  $T1(n)=O(f(n))$ y $T2(n)=O(g(n))$ ➔ $T1+T2 = O(\max\{f,g\})$
- Regla del producto

  $T1(n)=O(f(n))$ y $T2(n)=O(g(n))$ ➔ $T1*T2 = O(f*g)$

Repaso de Complejidad en Tiempo

- La complejidad en tiempo de una instrucción simple es una constante, $T(n)=c=O(1)$, incluye asignación, operación aritmética, lectura o escritura de una palabra.
- Para un condicional simple, se toma el peor caso entre **(a)** el tiempo de evaluar la condición (siendo falsa) o **(b)** el tiempo de evaluar la condición (siendo verdadera) más el tiempo de ejecutar el cuerpo del condicional.

Repaso de Complejidad en Tiempo

- Ejemplo:

  If (a == b || **factorial(n) < a**) cout << a;

  El peor caso es que a≠b para que se tenga que evaluar el factorial(n), y el tiempo sea T(n)=Tfact(N)+4

  Existirá un caso donde sea más eficiente ejecutar el cuerpo del condicional a no ejecutarlo?. Dar ejemplos. También es posible que el cuerpo del condicional simplifique etapas siguientes de un problema (e.g. colocar N=0 o retornar).

Repaso de Complejidad en Tiempo

- Para el caso del condicional múltiple, por lo general el peor caso es tener que evaluar todas las condiciones, y que solo la última sea verdadera. Pero esto depende del cuerpo de cada caso.

If <Cond1>

<Inst1>

else If <Cond2>

<Inst2> …

Repaso de Complejidad en Tiempo

- Para los ciclos, hay que sumar el tiempo de cada iteración. Y este tiempo puede variar drásticamente entre iteraciones cuando hay condicionales que evaluar.
- Por lo general, se puede acotar por la peor de todas las interaciones, multiplicada por el número de iteraciones efectuadas. Pero esta cota podría ser muy superior.

Repaso de Complejidad en Tiempo

- Para algoritmos recursivos por partición (se particiona un problema de tamaño n en a problemas de tamaño n/b, con un costo de fusión de soluciones parciales en tiempo f(n)), la función del tiempo de ejecución T(n) es una función recurrente
- T(1) = 1 { se toma T(1)=1 por conveniencia }
- T(n) = a.T(n/b)+f(n), si n>1

Repaso de Complejidad en Tiempo

$T(1) = 1$

$T(n) = aT(n/b)+f(n)$, si $n>1$

$$a < f(b) \Rightarrow T(n) = \mathrm{O}(n^{\log_b^{f(n)}})$$

$$a = f(b) \Rightarrow T(n) = \mathrm{O}(n^{\log_b^a} . \log_b^n)$$

$$a > f(b) \Rightarrow T(n) = \mathrm{O}(n^{\log_b^a})$$

Repaso de Complejidad en Tiempo

```
void MergeSort(list L) -> list
{
        If  (L.size() <= 1)
                return L;
        else
        {
                list L1 = Lista de los primeros n/2 elementos de L;
                list L2 = Lista de los últimos n/2 elementos de L;
                return(Mezcla(MergeSort(L1),MergeSort(L2)));
        }
}
```

$$T(n) = aT(n/b)+f(n)$$

Es claro que merge sort divide un problema de tamaño n en dos sub problemas de tamaño n/2, con un costo de dividir y mezclar f(n)=n. Por lo tanto a=b=2, f(n)=n, a=f(b), quedando así

$$a = f(b) \Rightarrow T(n) = \mathrm{O}(n^{\log_b^a} . \log_b^n) \equiv \mathrm{O}(n . \log_2^n)$$

Análisis amortizado

- Considera más de una operación en el análisis del tiempo, y no simplemente el peor caso.
- La razón es que en ciertos problemas, el peor de los casos cuando sucede, no vuelve a suceder sucesivamente, por lo cual, su costo queda amortizado.
- Por ejemplo, en un “vector”, si duplicamos su tamaño al hacer un “push_back”, la operación queda O(n) para el peor caso, pero hacer n inserciones seguidas en conjunto es O(n), pues se duplica el arreglo solo una vez. Así que una inserción es O(1) “amortizado”, pero es O(n) como peor caso.

Análisis amortizado

- El análisis amortizado considera el peor caso de un conjunto operaciones seguidas, por lo que no son opciones al azar. Esto difiere del análisis en el “caso promedio” (tiempo promedio de todas las posibles entradas), y del análisis probabilístico (promedio de todas las posibles opciones tomadas al azar).
- Se sigue considerando el peor de los casos, pero para una serie de operaciones, y no para una individual.
- Desde este punto de vista, es más justo que simplemente considerar el peor caso.

Análisis amortizado

- Con el análisis amortizado optimizamos el diseño de las estructuras de datos, produciendo estructuras avanzadas.
- Estudiaremos más adelante las estructuras de datos para conjuntos disjuntos y Heaps de Fibonacci.
- Veremos que el costo de unas operaciones son amortizadas con otras.

Análisis amortizado

- Existen diversas técnicas. Todas son aplicables y dan resultados equivalentes.

$$\textit{Ténicas} \begin{cases} \text{Método del agregado} \\ \text{Método contable} \\ \text{Método del potencial} \end{cases}$$

Análisis amortizado

- En este curso, consideraremos el método del agregado y el método del potencia.
- Método del agregado: calcula una cota superior del tiempo para n operaciones, y luego obtiene un promedio dividiendo entre n.
- Método del potencial: considera la energía potencial como las tareas realizadas en llamadas anteriores que se libera para pagar operaciones futuras.

Ideas finales

- Retomaremos el análisis amortizado al estudiar ciertas estructuras de datos avanzadas, como conjuntos disjuntos y Heaps de Fibonacci.