ANÁLISIS DE PROBLEMAS COMPLEJO

Marco Antonio Cruz Chávez

UAEM Av. Universidad 1001 Col. Chamilpa C.P. 62210 Cuernavaca Morelos, México
Agosto 12 del 2000
00334858@academ01.mor.itesm.mx

Abstract. En este trabajo presentamos una explicación de los puntos básicos mas importantes tratados en la teoría de la complejidad que deben de tenerse como base para el estudio de esta ciencia. También realizamos un análisis de algunos problemas conocidos como muy difíciles de resolver y clasificados como complejos dentro de una clasificación ya muy conocida llamada NP-completos.

1 INTRODUCCIÓN

En la actualidad se ha hecho una necesidad en muchas ciencias la facultad de poder obtener una medida de la complejidad de un problema sobre la base de estudios de su estructura y algoritmo involucrado, quedando claro que para la mayoría de los problema conocidos siempre se involucra un cierto procedimiento matemático que sirve para los cálculos en la obtención de la solución del problema [Gacs and Lobas 1999]. Lo anterior no siempre es cierto, pues existen problemas matemáticos bien definidos para los cuales no se cuenta con algoritmos, Turing demostró que tales problemas sin decisión existen. Un problema típico es el problema de paro (halting): Dado un programa de computadora con su entrada, este siempre para? Turing probo que no hay algoritmos que resuelvan correctamente todas las instancias de este problema [Papadimitriou and Steiglitz, 1982].
Algunas ciencias involucradas en problemas complejos son por ejemplo las ciencias computacionales, las ramas de las matemáticas, física estadística, biología, medicina, ciencias sociales e ingeniería. Debido a los recursos que siempre son restringidos en la búsqueda de la solución de los problemas dentro de las computadoras, se adquiere una clase de complejidad de acuerdo al tiempo y espacio de memoria requerida (las medidas más importantes de la complejidad de los problemas de computo). Se pueden determinar los recursos necesarios para una gran cantidad de problemas importantes conocidos en varias áreas de las matemáticas y obtener algoritmos eficientes para ellos y de esta manera ubicarlos en una de las clases de complejidad existentes [Gacs and Lobas 1999]. Lo anterior es importante en el sentido de que un problema, aunque se derive de un buen algoritmo puede ser inservible para algunos tipos de datos ya que los recursos de memoria pueden sobrepasar la disponible y/o el tiempo requerido para obtener una solución podría ser muy grande.
Los problemas pueden estar dentro de una división de clases de complejidad conocidas en la actualidad, siendo actualmente una de las más importantes

identificadas como P, NP, NP-completo, PSPACE, y NP-hard. Es importante el saber que clase de complejidad tiene un problema, pues de aquí se puede conocer, si bajo ciertos recursos limitados podemos resolverlos de acuerdo al tamaño de entrada dado en el problema. De acuerdo a esta entrada, depende la cantidad de recursos requeridos (tiempo y espacio de almacenamiento) y podemos decidir que tipo de algoritmos utilizar de acuerdo a la clase de complejidad (la complejidad de un algoritmo es una función del tamaño de su entrada) donde el problema se encuentre clasificado, para poder resolver instancias pequeñas, o grandes (si es que se permiten). Estos algoritmos pueden ser deterministicos o no-deterministicos.

Un algoritmo se dice que resuelve un problema A si este algoritmo puede resolver cualquier instancia X de A [Garey and Jonson 1979]. Entendiéndose por instancia de un problema cuando para ese problema se especifican valores particulares para todos sus parámetros. En general en la teoría de la complejidad de problemas, es de suma importancia para la mayoría de los investigadores, el encontrar siempre el algoritmo más eficiente (normalmente significa el mas rápido) que resuelva su problema de estudio.

El hecho de encontrar o tener problemas clasificados en las clases de complejidad más duras de resolver, no quiere decir que se tengan problemas inútiles. Esto puede ser una ventaja en aplicaciones como la generación de números aleatorios, protocolos de comunicación, protección de datos y criptografía (la teoría de la comunicación secreta). La criptografía es muy importante hoy en día en Internet, ya que en el se realizan muchas transacciones financieras que dependen de suposiciones de la teoría de la complejidad, tal como la dificultad de factorización de enteros o del rompimiento DES (encriptamiento estándar de datos)[Cook, 2000].

2 COMPLEJIDAD ESPACIAL Y TEMPORAL

Las preguntas más importantes que podemos hacernos acerca de un algoritmo es; si hace correctamente lo que tiene que hacer cuando recibe una entrada de datos validos; si su estructura es la adecuada; si tardara poco o demasiado en arrojar un resultado; si requiere una cierta cantidad de espacio de memoria para almacenar información como datos, arreglos y variables de programación. Los científicos de las ciencias de la computación han medido la complejidad de los algoritmos a través de estos dos últimos conceptos (tiempo y espacio), los cuales ofrecen una medida clara de la complejidad algorítmica.

Por muy bueno que sea un algoritmo, este podría ser inútil para algunos tipos de datos de entrada. Debido a que podría requerir una gran cantidad de memoria para almacenamiento no siendo suficiente la que se tenga, o bien podría tardar un tiempo considerablemente grande (años) para obtener un resultado. Debido a esto es de suma importancia el poder estimar las necesidades de estos recursos en cualquier algoritmo. Los requisitos de tiempo y espacio de un algoritmo son función de la cantidad de datos de entrada. De aquí que el problema de complejidad se convierta solo en: Dados un algoritmo y sus datos de entrada, determinar el tiempo y el espacio requeridos para su ejecución.

3 EL PROBLEMA SAT

Una fórmula booleana se crea a partir de un conjunto $\{x_i\}$ de variables booleanas y de conectivos lógicos (or ($\vee$), and ($\wedge$) y not ($\sim$)). Por ejemplo, $\sim x_1 \wedge (x_2 \vee x_3)$.
Dado un valor de verdad f(x) para cada variable es posible evaluar la fórmula booleana. Por ejemplo, si en la fórmula anterior $f(x_1) = V$, $f(x_2) = F$ y $f(x_3) = V$, esta tomará el valor F (falso), pero si en cambio $f(x_1) = F$, $f(x_2) = F$ y $f(x_3) = V$, la fórmula tomará el valor V (verdadero). Cuando una fórmula booleana resulta verdadera para una asignación de valores de verdad se dice que es satisfactible o consistente. Debe señalarse que no todas las fórmulas son satisfactibles, hay algunas que no pueden ser verdaderas para ninguna asignación de verdad, básicamente porque en ellas existen contradicciones.
Una forma muy común de escribir una fórmula booleana, también llamada fórmula proposicional, es la denominada Forma Normal Conjuntiva (CNF, por sus siglas en inglés). La formula CNF es una conjunción de cláusulas, esto es, cláusulas unidas a través del conector lógico “and”; Una cláusula es una disyunción de literales unidas mediante el conector lógico “or” y una literal es una variable booleana negada o no-negada que podrá ser interpretada como verdadera o falsa. Una cláusula que contiene sólo una literal es llamada cláusula unitaria. Una cláusula que no contiene literales es llamada cláusula vacía y es interpretada como falsa. De la estructura del CNF y los operadores lógicos contenidos se tiene que para que una fórmula escrita en CNF sea verdadera, cada cláusula en la fórmula debe ser verdadera. De lo mencionado, se puede definir el problema de satisfactibilidad (SAT) como [Gent, et al., 1999; Gu, et al., 1997; De Ita y Morales, 1996]:
Definición: Dadas m cláusulas C_1, C_2, ..., C_m, las cuales emplean las variables booleanas x_1, x_2, ..., x_n, ¿es satisfactible la fórmula $C_1 \wedge C_2 \wedge ... \wedge C_m$?
En el problema de satisfactibilidad (SAT) cada restricción se expresa mediante una ecuación booleana (aquella compuesta por variables que pueden asumir únicamente los valores de V o F) llamada cláusula proposicional, y el conjunto de cláusulas establece una fórmula booleana. De esta forma, el problema consiste en determinar una asignación de valores de verdad para las variables que satisfaga la fórmula proposicional (que la haga verdadera), esto es, que satisfaga todas las restricciones. De lo mencionado se establece que SAT es un caso especial de CSP (Problema de Satisfacción de Restricciones).
El problema de satisfactibilidad (SAT) es un problema muy simple de enunciarse, y su solución puede verse de manera combinatoria como el tratar todas las posibles asignaciones de valores de verdad para ver si alguna satisface la fórmula. Sin embargo, este no es un algoritmo eficiente, ya que hay 2^m asignaciones de verdad que probar, con una elección binaria para cada variable, por lo que el tiempo de ejecución del algoritmo crece exponencialmente con el tamaño de la entrada del problema.
SAT fue el primer problema que se demostró ser NP-completo [Cook, 1971], por lo que constituyó la piedra angular de la teoría de la complejidad computacional [Papadimitriou, 1994; Papadimitriou and Steiglitz, 1982]. El hecho de que SAT sea NP-completo implica que cualquier problema puede ser transformado a una instancia de él, por lo que cualquier mejora a los algoritmos que los resuelven puede transformarse en mejoras a los algoritmos que resuelvan la clase NP-completo. Aun más, el hallar un algoritmo eficiente para uno de los problemas en la clase NP-

completo, repercute en la resolución del problema abierto ¿P = NP? O de, si existe un algoritmo que pueda resolver los problemas de la clase NP-completo en el peor de los casos en tiempo polinomial.
La NP-completes es generalmente considerada para marcar el límite entre tratabilidad e intratabilidad de los problemas [Garey and Johnson, 1979]. Debido a que SAT es NP-completo, es poco probable que exista un algoritmo que lo resuelva en el peor de los casos en tiempo polinomial. Sin embargo, una gran cantidad de investigaciones en los últimos años a mostrado que muchos problemas de satisfactibilidad de interés práctico pueden ser resueltos eficientemente. Por ejemplo, k-SAT es la clase de problemas de decisión en los cuales todas las cláusulas son de longitud k. El problema k-SAT es NP-completo para cualquier $k \geq 3$, pero es polinomial para $k = 2$ [Garey and Johnson, 1979].
Los términos P, NP, NP-Completo dan una de tantas formas de clasificar a los problemas por su grado de complejidad, en esto recae el objeto de la teoría de la complejidad computacional. De acuerdo a problemas de decisión:

P. Son problemas que se pueden resolver en tiempo polinomial con algoritmos deterministicos.

NP. Son problemas que solo pueden resolverse en tiempo polinomial con algoritmos no-deterministicos.

PSPACE. Son problemas que pueden resolverse usando una cantidad razonable de memoria. Sin tomar en cuenta el tiempo en que la solución se encuentra.

Actualmente se acepta que P es un subconjunto de NP [Papadimitriou and Steiglitz, 1982] y se ha demostrado de forma trivial [Cook, 2000]. La pregunta que aun queda en el aire sin respuesta es si, P = NP?. En la figura 1 se muestra una posible asociación de la clase de problemas de acuerdo a su grado de complejidad con respecto al tiempo, el tamaño de los conjuntos en la figura no es una medida real de comparación..

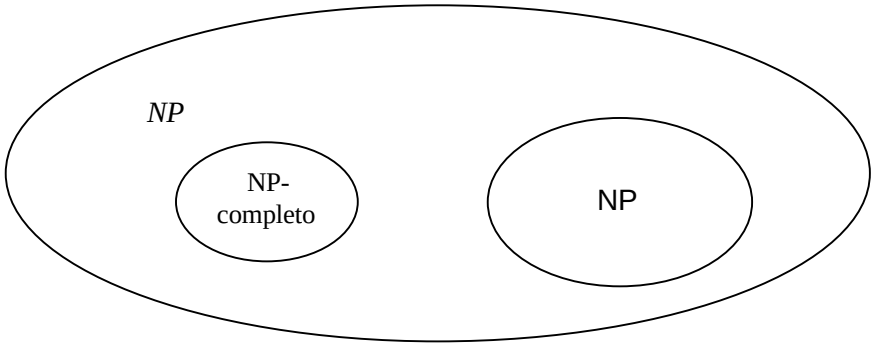

Fig. 1. Posible asociación de tipos de problemas en la teoría de la complejidad computacional.

Por que es importante SAT? Este es de suma importancia pues a partir de este se puede identificar la NP-completes de otros problemas NP que se sospechen puedan ser NP-completos. Esto de acuerdo al Teorema de Cook:
Se dice que un problema A en NP es NP-completo cuando, para cada otro problema B en NP, A es más difícil de resolver que B.
En la práctica se demuestra que un problema B que esta en NP es NP-completo de la siguiente forma [ICS 161, 1996]: Se comienza con un problema A que se conoce es NP-completo (como por ejemplo SAT) y cuya característica es que todos los problemas en NP son reducibles a A. Si A es NP-completo y B esta en NP, y A es reducible a B (se puede utilizar el algoritmo que resuelva B para resolver A, esto es, A es más fácil que B), entonces B es NP-completo. Y por lo tanto todos los problemas en NP son reducibles también a B. De aquí la importancia del problema SAT en la prueba de NP-completes para problemas NP.

4 MEDIDAS DE EFICIENCIA

El crecimiento de la complejidad de un algoritmo depende del tamaño de la entrada, principalmente para entradas muy grandes (n >> 1000), donde en este tipo de entradas se encuentra el principal interés de la complejidad de los algoritmos. Con el propósito de poder obtener una medida de la eficiencia de un algoritmo (optimización de recursos como tiempo y memoria) es de utilidad evaluar cotas superiores para los parámetros de ejecución en lugar de precisar los valores de estos parámetros. Tales estimaciones se pueden expresar bajo ciertas notaciones. Para tratar con velocidades de crecimiento de complejidad en las funciones, los teóricos ocupan la siguiente nomenclatura [Papadimitriou 1982; Gács and Lovász 1999]:

Sean f(n) y g(n) funciones reales definidas sobre los números naturales.

1. Se dice escribe f(n) = O(g(n)) si existe una constante c > 0 tal que, para grandes entradas de n, f(n) <= cg(n).
2. Se escribe f(n) = Ω(g(n)) si existe una constante c > 0 tal que, para grandes entradas de n, f(n) >= cg(n). Siendo esto el inverso de (a).
3. Se escribe f(n) = Θ(g(n)) si existen las constantes c_1, c_2 > 0 tal que, para grandes entradas de n, c_1g(n) <= f(n) <= c_2g(n). Esto es para ambos f(n) = O(g(n)) y g(n) = O(g(n)) se mantienen.
4. Se escribe f(n) = o(g(n)) si f(n) = 0 en un número finito de sitios y f(n)/g(n) -> 0 si n -> α.

Mediante esta notación, la rapidez de crecimiento de la complejidad de un algoritmo puede ser acotada por frases como “toma un tiempo f(n) = O(n^3)”, esto significa que la función f(n) es de orden temporal a lo sumo n^3, donde n es una entrada grande (n >> 1000).

A continuación se listan ejemplos de notación O [Johnsonbaugh, 1988].

Ejemplo 1. Dado que $f(n) = n^2 + n + 3 <= 3n^2 + 3n^2 + 3n^2 = 9n^2$

Se podría tomar $c = 9$ en la definición de (a) para obtener

$f(n) = n^2 + n + 3 = O(n^2)$

Ejemplo 2. Si se reemplaza cada entero 1,2,…,n por n en la suma 1 + 2 + … + n, la suma no decrece y se tiene

$1 + 2 + \ldots + n <= n + n + \ldots + n = n \cdot n = n^2$

Se deduce que $1 + 2 + \ldots + n = O(n^2)$.

En general podemos clasificar a los algoritmos por su grado de complejidad como: polinomiales y exponenciales. Las características de los algoritmos exponenciales, son que su rapidez de crecimiento (en función de su entrada n) en cuanto a requerimiento de tiempo y/o memoria son muy grandes, mientras que en los polinomiales los recursos requeridos se incrementan mucho mas lentamente (ver fig 2). También se puede observar en la figura 2 que cuando el tamaño de la entrada crece, cualquier algoritmo polinomial será más eficiente que cualquier exponencial.

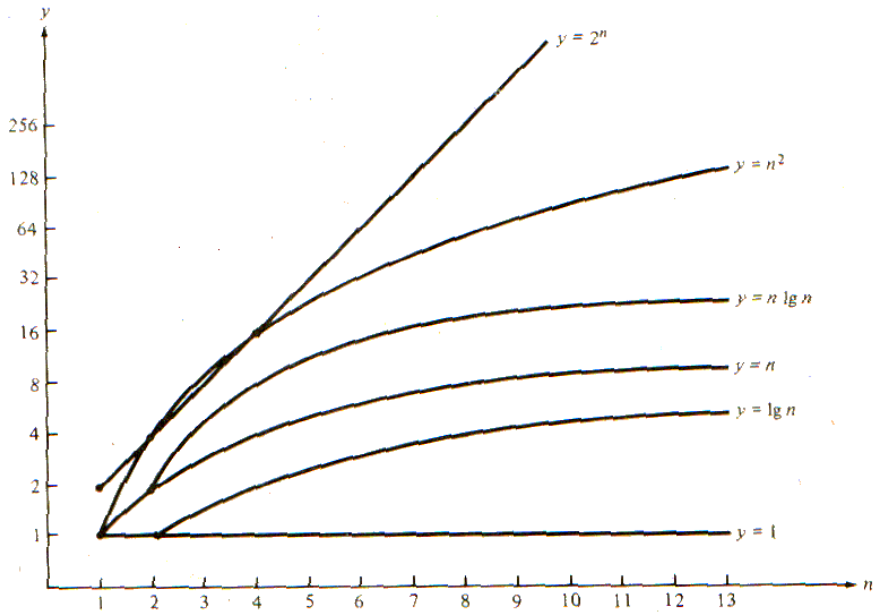

Fig. 2. Tiempos de ejecución de funciones polinomiales y exponenciales para varios tamaños de entradas.

Por otro lado, los algoritmos exponenciales toman mayor ventaja de los avances tecnológicos presentados en las computadoras. Como ejemplo, cada determinado tiempo las computadoras incrementan su velocidad de procesamiento, el tamaño de la más grande instancia resuelta por un algoritmo polinomial en una hora, se multiplicara por una constante entre 1 y 10. En contraparte un algoritmo exponencial solo incrementara muy poco el tamaño de la instancia máxima que pueda resolver [Papadimitriou and Steiglitz, 1982]. Esto se observa con claridad en la tabla 1.

FUNCION	TAMAÑO DE LA INSTANCIA A RESOLVER EN UN DIA	TAMAÑO DE LA INSTANCIA A RESOLVER EN UN DIA EN UNA COMPUTADORA 10 VECES MAS RAPIDA.
N	10^{12}	10^{13}
n log n	0.948×10^{11}	0.87×10^{12}
n^2	10^6	3.16×10^6
$0n^3$	10^4	2.15×10^4
10^8n^4	10	18
2^n	40	43
10^n	12	13
$n^{\log n}$	79	95
n!	14	15

Table 1. Los algoritmos de tiempo polinomial toman mayor ventaja de la tecnología.

5 ORDEN DE CRECIMIENTO DE UNA FUNCION

Como se vio anteriormente el orden de crecimiento de una función puede ser polinomial o exponencial en tiempo y/o memoria. Para ilustrar esto, la tabla 2 muestra el orden de crecimiento para funciones de tipo polinomial y exponencial, en esta se da el tamaño de la entrada para la función, y de acuerdo a este tamaño, se puede notar que las funciones exponenciales crecen de una manera muy rápida. A partir de esta observación se puede notar que algunas funciones, la obtención de una solución podrían ser vista tal vez por futuras generaciones, pero no por nosotros, de aquí que entendamos que estos tipos de algoritmos no son del nada eficientes y que es preferible el contar siempre con algoritmos acotados de tiempo polinomial, los cuales por lo que llegamos a observar son los mas eficientes en la administración de los recursos limitados que generalmente uno cuenta para la solución de un problema en computadora.

Número de instrucciones hasta la terminación para una entrada de datos de tamaño n	Tiempo de ejecución si n =								
	3	6	9	12	50	100	1000	10^5	10^6
1	10^{-6} s	10^{-6} s	10^{-6} s	10^{-6} s	10^{-6} s	10^{-6} s	10^{-6} s	10^{-6} s	10^{-6} s
lg lg n	10^{-6} s	10^{-6} s	2×10^{-6} s	2×10^{-6} s	2×10^{-6} s	3×10^{-6} s	3×10^{-6} s	4×10^{-6} s	4×10^{-6} s
lg n	2×10^{-6} s	3×10^{-6} s	3×10^{-6} s	4×10^{-6} s	6×10^{-6} s	7×10^{-6} s	10^{-5} s	2×10^{-5} s	2×10^{-5} s
n	3×10^{-6} s	6×10^{-6} s	9×10^{-6} s	10^{-5} s	5×10^{-5} s	10^{-4} s	10^{-3} s	0.1 s	1 s
n lg n	5×10^{-6} s	2×10^{-5} s	3×10^{-5} s	4×10^{-5} s	3×10^{-4} s	7×10^{-4} s	10^{-2} s	2 s	20 s
n^2	9×10^{-6} s	4×10^{-5} s	8×10^{-5} s	10^{-4} s	3×10^{-3} s	0.01 s	1 s	3 h	12 d
n^3	3×10^{-5} s	2×10^{-4} s	7×10^{-4} s	2×10^{-3} s	0.13 s	1 s	16.7 min	32 a	31,710 a
2^n	8×10^{-6} s	6×10^{-5} s	5×10^{-4} s	4×10^{-3} s	36 a	4×10^{16} a	3×10^{287} a	3×10^{30089} a	3×10^{301016} a

Símbolos: s = segundo; min = minuto; h = hora; d = día; a = año.

Table 2. Tiempo para ejecutar un algoritmo si una instrucción toma 1 microsegundo [μs] en ejecutarse.

6 ANÁLISIS DE CASOS PRACTICOS

Criptografía [ICS 161, 1996]

Suponga que tenemos una función de encripción:

code=RSA(key,text)

La encripción RSA trabaja realizando un poco de aritmética simple en el código y la llave, la cual consiste de un par (p,q) de números primos grandes. Uno puede realizar la encripción solo conociendo el producto pq; pero para des-encriptar el código se necesita conocer el producto de diferencias, (p-1)(q-1).

Una suposición común en criptografía es la “Ataque al texto plano conocido”: se tiene el código para un mensaje, y se conoce (o puede suponerse) el texto de este mensaje. Se quiere utilizar esta información para encontrar la llave, y así poder des-encriptar otros mensajes utilizando la misma llave.

Formalizado como un problema NP, simplemente se quiere encontrar una llave para el cual code=RSA(key,text). Si se da una llave, puede probarse realizando la encripción, esto es en NP.

La pregunta es, como encontrar la llave?. Para que el código sea fuerte, se espera que no sea posible hacerlo mucho mejor que una búsqueda de fuerza bruta.

Otro uso común de RSA involucra “la criptografía de llave pública”: un usuario del sistema publica el producto pq, pero no publica p, q, o (p-1)(q-1). De este modo cualquiera puede enviar un mensaje a este usuario utilizando la encripción RSA, pero sólo el usuario puede des-encriptarlo.

Fraccionando este esquema puede pensarse cono un problema NP diferente: dado un número compuesto pq, encontrar la factorización en números pequeños.

Puede probarse la factorización rápidamente, así el problema es NP. Encontrar la factorización puede parecer complicado, y puede pensarse que no lo es en tipo P. Sin embargo, existe fuerte evidencia de que tampoco es NP-completo.

Problema Halt [ICS 161, 1996]

Suponga que está trabajando en un laboratorio para una clase de programación, ha escrito sus programas y empieza a córrelos. Después de cinco minutos, estos aun no terminan de correr, esto significa que tienen un loop infinito, o son demasiado lentos? Puede ser conveniente que el compilador informe que el programa tiene un loop infinito. Sin embargo, este es un problema: no existe un programa que pueda detectar correctamente loops infinitos.

Algunas personas han usado esta idea para demostrar que las personas son intrínsecamente mas listas que las computadoras, desde entonces, se demostró que existen problemas que las computadoras no pueden resolver. Sin embargo, tampoco está claro que las personas puedan resolverlas. Aquí un ejemplo:

```
main()
{
int x = 3;
for (;;) {
   for (int a = 1; a <= x; a++)
   for (int b = 1; b <= x; b++)
      for (int c = 1; c <= x; c++)
      for (int i = 3; i <= x; i++)
         if(pow(a,i) + pow(b,i) == pow(c,i))
         exit;
   x++;
}
}
```

Este programa busca soluciones para el ultimo teorema de Fermat. Tarminará? (Puede asumirse que se está utilizando un paquete de enteros de precisión múltiple en lugar de construcciones en enteros, no preocuparse por las complicaciones aritméticas de overflow). Para poder contestar esto, debe entenderse la demostración del último teorema de Fermat. Existen problemas similares para los cuales no hay demostración conocida, así, no hay solución al problema si el programa correspondiente para.

Rutas largas simples. [ICS 161, 1996]

Una ruta simple en un gráfico es sólo uno sin bordes repetidos o vértices. Para describir el problema de encontrar las rutas largas en términos de la teoría de complejidad, es necesario formalizarlo como una pregunta sí-o-no: dado un gráfico G, vértices s y t, y un número k, existe una ruta simple de s a t con al menos k bordes? Una solución a este problema entonces consistiría en tal ruta.
Por qué está en NP? Si se da una ruta, puede mirarse rápidamente y agregar la longitud, comprobando con minuciosidad que realmente es una ruta con al menos longitud k. Esto se puede hacerse en tiempo lineal, así se puede hacer en tiempo polinómico.
Sin embargo no se sabe si este problema está en P; No se ha mencionado una buena manera para encontrar tal camino (con polinomio de tiempo en m,n, y K). Y el hecho de que este problema es NP-completo, así que se cree que no existe tal algoritmo.
Existen algoritmos que solucionan el problema; por ejemplo, enumerando los subconjuntos de los 2^m bordes y controlando cualesquiera de ellos solucionan el problema. Pero por lo que sabemos no hay algoritmo que se ejecuta en tiempo polinómico.

7 CONCLUSIONES

Cuando se tiene un nuevo problema en mano, en el que se desea obtener la solución de una o varias instancias de este, siempre será recomendable antes que nada, analizar su complejidad en tiempo y espacio, para de este modo saber las características que debe de tener el algoritmo que necesitemos aplicar. De esta manera podemos ahorrarnos muchos golpes de cabeza al tratar de resolver el problema de una forma totalmente ciega. Para que el trabajo de analizar las características del problema que se tiene sea mas fácil, nos podemos ayudar de problemas cuya complejidad haya sido clasificada con anterioridad. Con esta clasificación de problemas, podemos realizar comparaciones de semejanza con el nuestro y tal vez encontrar uno que se asemeje lo suficiente para conocer de esta forma en que grado de complejidad nos encontramos.

REFERENCIAS.

1. [Cook, 1971] S.A. Cook, "The complexity of theorem proving procedures", Proc. Of 3rd. Annual ACM Symposium on the Theory of Computing, 151 - 158 pp-, 1971.
2. [Cook, 2000] The P versus NP problem, Stephen Cook, April 2000. Manuscript prepared for the clay Mathematics Institute for the Millenium Prize Problems.
3. [De Ita y Morales, 1996] G. De Ita y G. Morales, "Heurísticas para mejorar la búsqueda local en el tratamiento de problemas de máxima satisfactibilidad", Memorias Iberamia-96, Puebla, México, 1996.
4. [Gács and Lovász, 1999] P. Gács and L. Lovász, Complexity of Algorithms, Lecture Notes for Spring 1999. http://www.cs.yale.edu/homes/lovasz/notes.html
5. [Garey and Jonson, 1979] M.R. Garey and D.S. Johnson, "Computers and intractability: a guide to the theory of NP-completeness", W.H. Freeman and Company, USA, 340 pp., 1979.
6. [Gent, et al., 1999] I.P. Gent and T. Walsh, " The Search for Satisfaction", http://dream.dai.ed.ac.uk/group/tw/sat., {ipg, tw}@cs.strath.ac.uk., 1999.
7. [Gu, et al., 1997] J.Gu, P.W. Purdom, John Franco and B. W. Wah, "Algortthms for the satisfiability (SAT) problem: a survey", DIMACS series in Discrete Mathematics and Theoretical Computer Science, 1997.
8. [ICS 161, 1996] ICS 161: Design and análisis of algorithms, Lecture notes for March 12, 1996. http://www.ics.ucl.edu/~oppstein/161/960312.html
9. [Johnsonbaugh, 1988] R. Johnsonbaugh, Matemáticas discretas, Grupo Editorial Iberoamerica S. A. de C. V., México ISBN 968-7270-46-2, 506 PP., 1988.
10. [Papadimitriou, 1994] C.H. Papadimitriou, "Computational complexity", Addison Wesley Pub. Co. , USA. ISBN 0-201-53082-1, 523 pp., 1994.
11. [Papadimitriou and Steiglitz, 1982] C.H. Papadimitriou and K. Steiglitz, "Combinatorial optimization: algorithms and complexity", Prentice Hall Inc., USA. ISBN 0-13-152462-3, 496 pp., 1982.