Universidad de Oviedo

Escuela de Ingeniería Informática
Universidad de Oviedo

Trabajo Fin de Grado

Introducción a la teoría de complejidad computacional

Josué Fernández Argüelles
Director: Elías Fernández-Combarro Álvarez

Grado, Asturias, 31 de mayo de 2024

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

Uno de los mayores retos de la ciencia y del ser humano, ha sido la respuesta a la pregunta de cómo resolver un problema de la forma más rápida y eficaz posible. ¿Pero qué significa realmente resolver un problema de forma rápida y eficaz?

En nuestro mundo, el de la informática, resolver un problema eficazmente puede significar que este tenga una buena solución respecto al tiempo que tarda en hallar una respuesta, a la cantidad de espacio y memoria que necesita, o a la cantidad de energía que se consume en la búsqueda de esta. Por ello un algoritmo que es capaz de resolver un problema, pero consume una cantidad de tiempo, memoria y energía demasiado elevados, no será de utilidad [1].

En este trabajo, mi tarea será investigar a cerca de aquellos problemas relacionados con la complejidad computacional, comenzando por un análisis de las maneras en las que se puede determinar el coste temporal de un algoritmo mediante el uso de máquinas de Turing de diferentes tipos, seguida por una definición de las clases P y NP, estudiando su relación a través de problemas NP-duros y NP-completos, ejemplificando con algunos de los problemas más destacados como pueden ser el coloreado de grafos, la búsqueda de caminos hamiltonianos, etc. Por último, realizaré un estudio de las formas de mitigar la dificultad de la resolución de estos problemas con algoritmos heurísticos y aproximaciones.

Capítulo 2

Palabras Clave

Algoritmo, problema computacional, complejidad, notación asintótica, máquina de Turing, problemas P, problemas NP, NP-completo, NP-duro, reducciones, no determinismo, verificador, SAT.

Capítulo 3

Fundamentos Teóricos

En este apartado veremos las bases y conceptos más importantes de la complejidad computacional, como qué es una modelo de computación, una máquina de Turing o la notación asintótica, así como ejemplos de todos estos para entenderlos completamente, ya que estos conceptos serán usados a lo largo de todo el trabajo.

3.1. Modelos formales de computación

Un modelo formal de computación es una abstracción, un modelo matemático, en el que una serie finita de instrucciones nos permiten resolver un problema en un número finito de pasos. Estos nos ayudan a comprender el funcionamiento de los sistemas computacionales [7]. Algunos de los ejemplos más conocidos son:

- **Máquina de Turing**: Fue propuesta por Alan Turing en 1936, y se trata de un modelo de computadora que trabaja sobre una cinta infinita dividida en celdas en las que lee, escribe y borra símbolos. Es el modelo que utilizaremos en este trabajo para describir las ideas centrales, por lo que lo explicaré con mucho mayor detalle más adelante.
- **Cálculo Lambda**: Este modelo fue creado por Alonzo Church en la década de 1930, siendo este un sistema que describe la computación mediante la manipulación de funciones, de manera que estas son tratadas como datos y son pasadas como argumentos a otras funciones [10]. Un ejemplo de expresión en cálculo lambda sería:

  $$(\lambda x.x + 1)2$$

  Donde:

  - $\lambda x.$: indica el comienzo de una función con un parámetro x
  - **x+1**: cuerpo de la función, es este caso indica que se le suma 1 al parámetro.
  - $(\lambda x.x+1)$: define la función a la que se le pasa el parámetro x, y devuelve x + 1.
  - **2**: argumento que se da a la función

  Otro ejemplo más complejo sería:

  $$(\lambda x.(\lambda y.x + y)))2\,3$$

En este caso, la primera función recibe como parámetro x, pero en devuelve otra función, la que recibe y como parámetro y devuelve la suma de los parámetros pasados.

- **Máquina de registros**: este modelo consta de un número finito de registros, donde cada uno puede almacenar un valor y realizar operaciones básicas [17]. Un ejemplo de ello sería:

  - **Registros**:
    - R1 = 0
    - R2 = 0
    - R3 = 0
  - **Operaciones**:
    - **Suma** R1 = R2 + R3
    - **Resta** R1 = R2 - R3
    - **Multiplicación** R1 = R2 * R3
    - **División** R1 = R2 / R3

Además de estos ejemplos, podemos encontrar una cantidad enorme de modelos formales de computación, todos ellos equivalentes a los vistos [2].

3.2. Problemas computacionales

Un problema computacional consiste en una serie de preguntas con una serie de respuestas, o dicho de una forma más técnica, es una relación binaria entre un conjunto de instancias y un conjunto de soluciones de ese determinado problema. Otra forma de referirse a él, es como *problema abstracto*. Estos se convierten en *problemas concretos* una vez que tanto las instancias como las soluciones se codifican en lenguajes formales. Por ejemplo, un problema abstracto sería el problema de la ordenación de números enteros, en cuyo caso la instancia sería una sucesión finita de números enteros $a_1, a_2, ..., a_n$ y la solución una permutación de la instancia $(a'_1, a'_2, ..., a'_n$ tal que $a'_1 \leq a'_2 \leq ... \leq a_n)'$, y un problema concreto derivado, sería aquel con la instancia (7, 8, 1, 2) y la solución (1, 2, 7, 8).

Estos a su vez se pueden dividir en tres grandes grupos de problemas:

- **Problemas de decisión**: Son aquellos en los que las dos únicas soluciones posibles son *sí* o *no*, o dicho de una manera más formal, aquellos determinados por el conjunto Y de instancias, asociadas a la solución *sí*. Por ejemplo, el conjunto de todos los grafos que tienen un camino Hamiltoniano. A continuación vamos a ver un ejemplo de este tipo de problema:

  Tenemos el grafo de la Figura 3.1, para el que tenemos que responder a la pregunta de si tiene un camino hamiltoniano, es decir, si existe un camino dentro del grafo que recorra todos los nodos pasando exactamente una vez por cada uno de ellos.

  Para responder a esta pregunta, simplemente tenemos que encontrar un camino que cumpla con esta condición, que en este caso podría ser, como se puede observar en la Figura 3.2, V1, V2, V5, V9, V8, V4, V7, V6, V3.

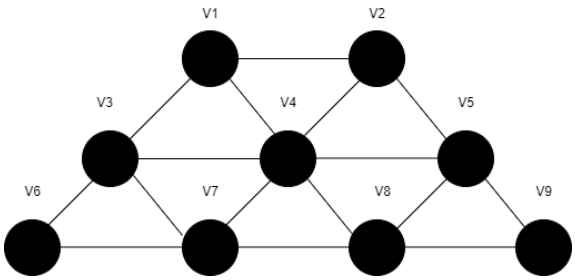


Figura 3.1: Grafo Inicial

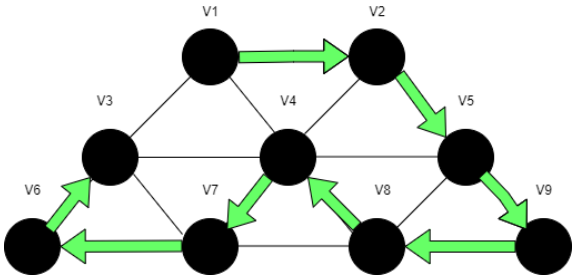


Figura 3.2: Grafo con camino Hamiltoniano

- **Problemas de búsqueda**: En estos problemas lo que se quiere es encontrar una solución en un espacio de respuestas (que suele ser exponencial). En este caso, la relación entre las instancias y soluciones queda determinadas por el llamado predicado lógico $P(i, s)$ que determina si s es solución de i. Podemos encontrar un buen ejemplo de este tipo de problema en el algoritmo de búsqueda en profundidad, más conocido como *Depth-First Search*. Para ver este problema ejemplificado tenemos a continuación en la Figura 3.3 un grafo en el que queremos encontrar el nodo rojo. El algoritmo que vamos a utilizar para ello es *Depth-First Search*.

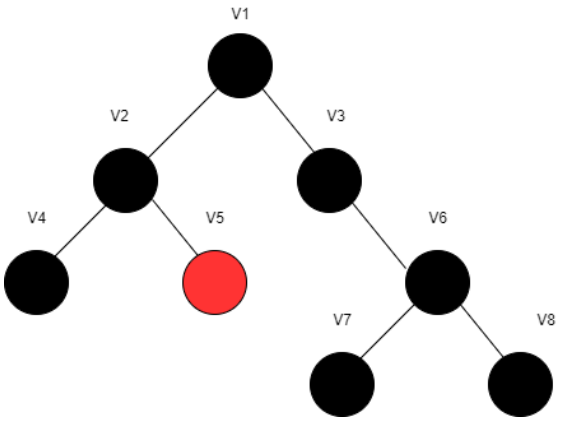


Figura 3.3: Grafo Depth-First Search

Este consiste en recorrer cada una de las ramas del árbol de la manera más profunda posible, es decir, elegir como siguiente nodo de la búsqueda, al primer hijo no visitado del nodo en el que nos encontremos. Cuando ya no queden más hijos sin visitar, se hace backtracking al nodo padre actual y se continúa la aplicación del algoritmo. De esta manera, el recorrido para encontrar el nodo rojo sería V1, V2, V4, V5. Como puede verse en la Figura 3.4

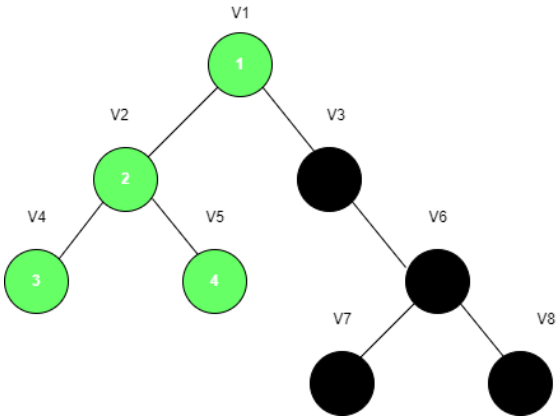


Figura 3.4: Solución Depth-First Search

- **Problemas de optimización**: En estos problemas, no solamente se busca una solución como en los vistos anteriormente, si no que se busca la mejor de las soluciones según alguna métrica. Un ejemplo de estos casos es el problema del viajante de comercio, que describiremos en detalle más adelante en la sección 6.4.3.

3.3. Máquinas de Turing

Una máquina de Turing es un modelo de computación consistente en un dispositivo que tiene una o varias cintas ilimitadas unidimensionales divididas en celdas, cada una de ellas conteniendo un valor de un conjunto finito de símbolos preestablecidos. Este dispositivo también tiene una cabeza que lee y escribe en una única celda a la vez, moviéndose a la izquierda o la derecha. Una máquina de Turing puede ser formalmente definida como una tupla de siete elementos $M =< Q, \Gamma, b, \Sigma, \delta, q_0, F >$ donde:

- Q es un conjunto finito no vacío que contiene los estados que puede tomar la máquina.
- Σ es un conjunto de símbolos que pueden aparecer en el contenido inicial de la cinta. Se le denomina alfabeto de entrada.
- Γ es un conjunto de símbolos finito, no vacío. A diferencia de Σ, este conjunto se denomina alfabeto de cinta, ya que son todos los símbolos que pueden aparecer en la cinta de la máquina, desde los símbolos de entrada hasta aquellos que la máquina utiliza de forma interna para realizar las operaciones que necesite.
- b es el símbolo vacío habitualmente representado por $\_$. Es el valor por defecto que tienen las casillas de la cinta o cintas.
- δ es la función de transición, la que indica cómo debe comportarse la máquina de Turing, es decir, hacia donde tiene que ir, a qué estado cambiar y qué escribir, en función del símbolo que lea y del estado en el que se encuentre.
- q_0 es el estado inicial de la máquina.
- F es el conjunto de estados finales de la máquina.

A continuación, vamos a ver el funcionamiento real de una máquina de Turing de una cinta con uno de los ejemplos más populares: una máquina de Turing que calcula si una secuencia de dígitos binarios es o no un palíndromo.

En primer lugar, vamos a definir la máquina que vamos a utilizar como:

- $Q = \{q_0, q_1, qRight0, qRight1, qSearch0L, qSearch1L, qLeft0, qLeft1, qSearch0R, qSearch1R\}$
- $\Sigma = \{0, 1\}$
- $\Gamma = \{0, 1, \_\}$
- $b = \_$
- $q_0 = q_0$
- $F = \{\text{qAccept, qReject}\}$

Siguiendo la definición de la máquina de Turing, vamos a indicar la tabla de función de transición δ, que se puede ser en el Cuadro 3.1.

Para facilitar la lectura de la tabla, podemos guiarnos por el diagrama de la Figura 3.5, en el que cada nodo representa un posible estado de la máquina, y cada una de las fechas las posibles transiciones entre estos, siendo la notación *lee:escribe/se mueve hacia*

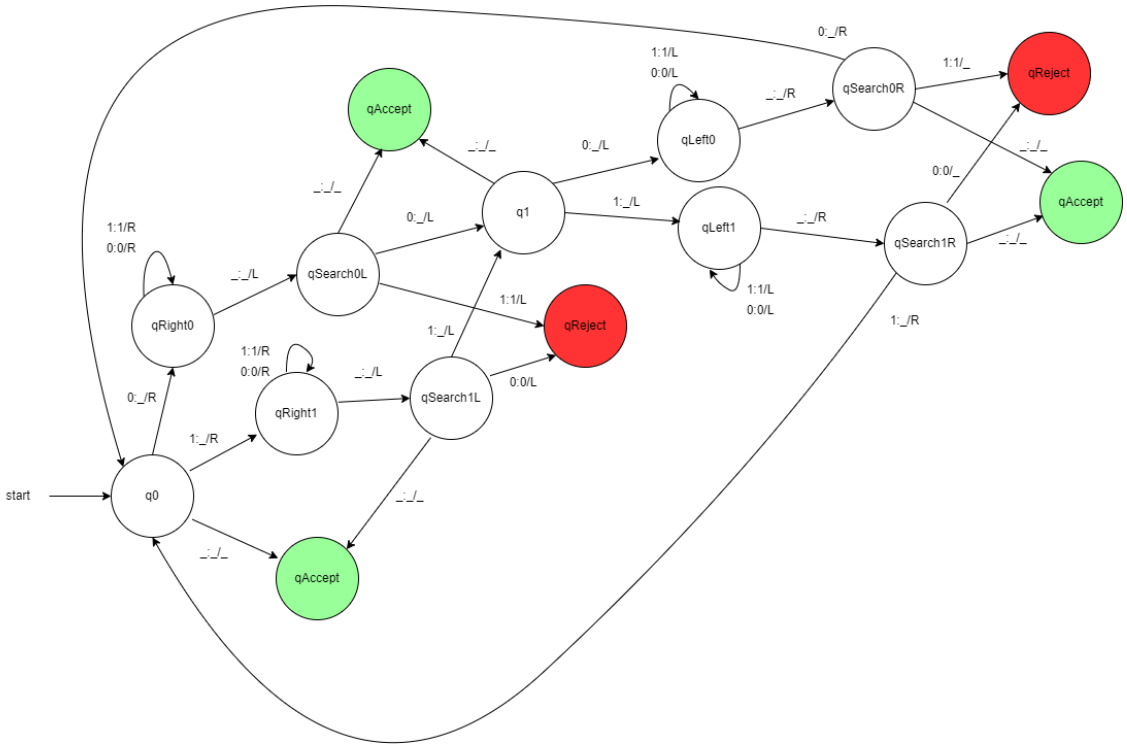


Figura 3.5: Diagrama de la máquina de Turing

Estado	Lee Símbolo	Nuevo Símbolo	Nuevo Estado	Dirección
q_0	0	_	$qRight0$	derecha
q_0	1	_	$qRight1$	derecha
q_0	_	_	$qAccept$	-
$qRight0$	0	0	$qRight0$	derecha
$qRight0$	1	1	$qRight0$	derecha
$qRight0$	_	_	$qSearch0L$	izquierda
$qRight1$	0	0	$qRight1$	derecha
$qRight1$	1	1	$qRight1$	derecha
$qRight1$	_	_	$qSearch1L$	izquierda
$qSearch0L$	0	_	q_1	izquierda
$qSearch0L$	1	1	$qReject$	-
$qSearch0L$	_	_	$qAccept$	-
$qSearch1L$	1	_	q_1	izquierda
$qSearch1L$	0	0	$qReject$	-
$qSearch1L$	_	_	$qAccept$	-
q_1	0	_	$qLeft0$	izquierda
q_1	1	_	$qLeft1$	izquierda
q_1	_	_	$qAccept$	-
$qLeft0$	0	0	$qLeft0$	izquierda
$qLeft0$	1	1	$qLeft0$	izquierda
$qLeft0$	_	_	$qSearch0R$	derecha
$qLeft1$	0	0	$qLeft1$	izquierda
$qLeft1$	1	1	$qLeft1$	izquierda
$qLeft1$	_	_	$qSearch1R$	derecha
$qSearch0R$	0	_	q_0	derecha
$qSearch0R$	1	1	$qReject$	-
$qSearch0R$	_	_	$qAccept$	-
$qSearch1R$	1	_	q_0	derecha
$qSearch1R$	0	0	$qReject$	-
$qSearch1R$	_	_	$qAccept$	-

Cuadro 3.1: Ejemplo de función de estado

Aunque pueda parecer extremadamente complicado, el funcionamiento de esta máquina es bastante sencillo. La máquina comienza en el estado q0, y dependiendo si el primer dígito que lee es 0 o 1, pasa al estado qRight0 o qRight1 respectivamente. En estos estados lo que hará es recorrer toda la cinta en busca del final de la cadena de entrada. Una vez llega a este, es decir, lee el símbolo '_', pasa al estado qSearch0L o qSearch1L. En este punto la máquina leerá el siguiente carácter, y como lo que está buscando es decidir si la entrada es una cadena palíndroma, este deberá ser igual que el leído al principio, por eso si no lo es y por ejemplo lee un 1 en el estado qSearch0L, pasará al estado qReject, indicando que el número no es un palíndromo; en cambio si el símbolo que lee es un 0, continuará calculando si la cadena es o no palíndroma, haciendo un proceso igual al explicado, esta vez hacia la izquierda. Vamos a ver cómo sería la ejecución paso a paso con la entrada 1001.

3.3.1. Ejecución con entrada 1001

Inicialmente la máquina se encuentra en el estado q0 y la cabeza de lectura en el primer dígito de la entrada.

Estado: q0

···	-	1	0	0	1	-	···
		↑					

Cabeza de lectura

Como esta lee un 1, pasa al estado qRight1, escribe un espacio en blanco, y sigue avanzado hasta la derecha en busca del final de la cadena.

Estado: qRight1

···	-	-	0	0	1	-	···
			↑				

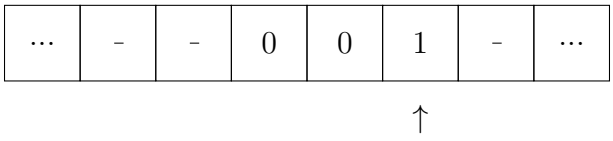
Al llegar al primer espacio en blanco, la máquina sabe que está al final de la cadena, por lo que cambia al estado qSearch1L, en el que como su nombre indica, buscará un 1 a su izquierda

Estado: qRight1

···	-	-	0	0	1	-	···
						↑	

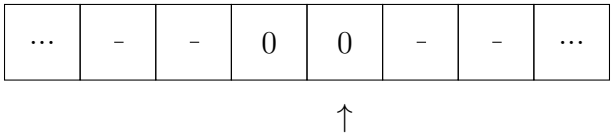
En este punto, la máquina busca un 1, y al encontrarlo, quiere decir que por el momento el número puede ser palíndromo, por lo que cambiará el 1 un por un espacio en blanco, pasará al estado q1 y seguirá comprobando el número

Estado: qSearch1L

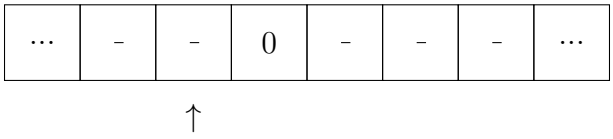


En el estado q1, la máquina realiza una acción similar que en q0, ya que esta lee el número sobre el que se encuentra la cabeza de lectura, lo borra, pasa al estado qLeft0, en el que recorre toda la cadena hasta llegar al primer espacio vacío de su izquierda.

Estado: q1

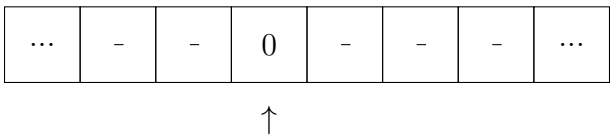


Estado: qLeft0



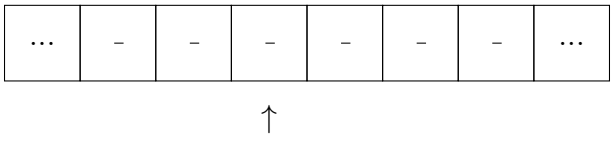
En este punto, como el último digito leído fue un 0, la máquina pasará al estado qSearch0R, en el que buscará un 0 a su derecha.

Estado: qSearch0R



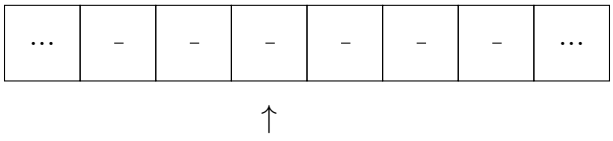
Una vez la máquina encuentra el 0 que estaba buscando vuelve al estado q_0.

Estado: q_0



La máquina, al estar en el estado q_0 y encontrarse con el símbolo vacío pasa a qAccept, indicando que efectivamente el número introducido como parámetro es palíndromo.

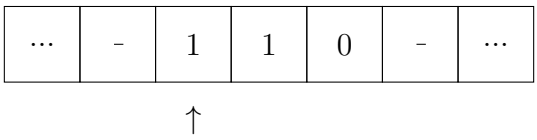
Estado: qAccept



3.3.2. Ejecución con entrada 110

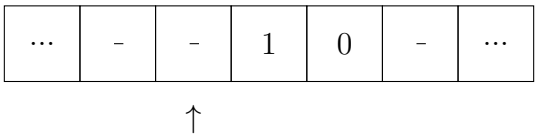
A continuación vamos a ver un nuevo ejemplo con la máquina de Turing anterior, esta vez con el parámetro de entrada 110: La máquina se encuentra en su estado inicial q0, y con la cabeza de lectura sobre el primer dígito de la entrada.

Estado: q0



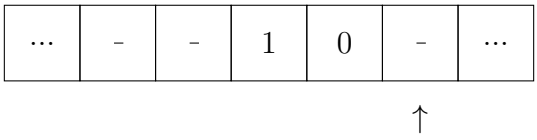
Al leer un 1, el siguiente estado de la máquina es qRight1, que mantendrá hasta llegar al final de la cadena.

Estado: qRight1



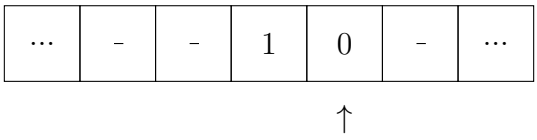
Una vez alcanzado este, se pasará a qSearch1L, y se comprobará si la casilla de la izquierda es un 1.

Estado: qSearch1L



Pero a diferencia del primer ejemplo, la máquina, que estaba buscando un 1, se encuentra con un 0, lo que quiere decir que la cadena proporcionada como entrada no es palíndroma, sin importar ya el resto de este, por lo que sin necesidad de seguir con la ejecución la máquina pasa al estado qReject, no aceptando esta entrada.

Estado: qReject



3.3.3. Otros ejemplos de máquina de Turing

También podemos encontrar otras variantes de la máquina de Turing más complejas, como las denominadas máquinas de Turing multicinta, que en lugar de estar formadas por una única cinta, tienen 2 o más, de manera que la función de estado, se amplía indicando en qué cinta se debe leer y escribir. Puede parecer que estas tienen mucho mayor potencial que una máquina de Turing de una cinta, pero la realidad es que son equivalentes en cuanto a poder de computación, por lo que ambas pueden realizar los mismos cálculos. Sin embargo, no son equivalentes en cuanto a nivel de complejidad, ya que un algoritmo ejecutado en un tiempo t por una máquina de Turing de varias cintas, podría necesitar un tiempo t^2 para ser ejecutado por una máquina de Turing de una sola cinta. Además, cualquier máquina de k cintas ($k > 2$) puede ser simulada en una máquina de dos cintas en un tiempo $t \log t$, siendo t el tiempo de la máquina de k cintas [6]. Vamos a ver a continuación una demostración de por qué esto es cierto: Sea T una máquina de Turing con k cintas y U una con una única cinta. U es capaz de simular el comportamiento de T guardando la información de todas sus cintas en su única cinta, utilizando el símbolo # para separar el contenido de las diferentes cintas. Además de esto, U debe guardar la localización de las cabezas de lectura, utilizando símbolos con un punto encima [13]. En la Figura 3.6 podemos ver cómo se representa en la cinta de U, tanto la separación entre las diferentes cintas de T, cómo las múltiples cabezas de lectura de esta.

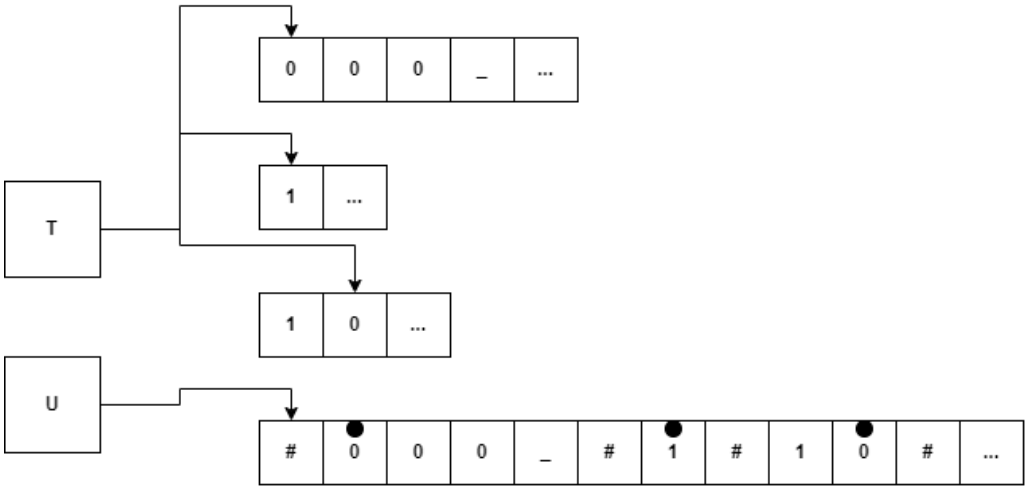


Figura 3.6: Comparativa máquina de Turing múltiple y simple.

Como ejemplo de una máquina de Turing multicinta, podemos pensar en la máquina utilizada anteriormente que calcula si una secuencia de digitos binarios es o no palíndroma. Para ello tendríamos 2 cintas, una con la secuencia de entrada, y la otra vacía. Lo primero que se haría es una copia de la secuencia en la cinta vacía. En una la cabeza de lectura apuntaría al primer dígito, mientras que en la otra, al último. El funcionamiento de esta máquina simplemente sería comprobar que el número al que apunta la cabeza de lectura se una cinta sea el mismo al que apunta la cabeza de la otra, y moverse hacia el centro de la secuencia comprobando en cada celda que el número es el mismo. En este caso, aunque la simulación puede parecer muy ineficiente, su complejidad no difiere de la del algoritmo óptimo. Una máquina de una cinta no puede realizar esta comprobación en un tiempo lineal, ya que se puede demostrar que el mejor algoritmo posible tarda un tiempo cuadrático. Para realizar esta simulación, la máquina de una cinta tendría que tener dos veces la

secuencia que queremos comprobar separada por un #, guardando la localización de las cabezas de lectura al principio de la primera secuencia y al final de la segunda. Después de esto, tendría que recorrer ambos números comprobando donde está cada cabeza de lectura, verificando que los números sean el mismo y moviéndolas hacia el centro de sus respectivos números. Por ello, por cada instrucción de la máquina multicinta, se ejecutarían O(T) instrucciones, siendo T el tiempo total de ejecución de la máquina multicinta.

3.4. No determinismo

El no determinismo es un concepto clave en la teoría de complejidad computacional. Cuando pensamos en un algoritmo, en una máquina de computación, nos imaginamos que cuando esta se encuentra en un estado, el siguiente está determinado en función del símbolo que lea o de la instrucción que se le dé. Esto es lo que llamamos una máquina determinista. Pero al hablar de indeterminismo, nos referimos justo a lo contrario, y aunque pueda parecer contraintuitivo, una máquina indeterminista es aquella en la que desde cualquier estado, podemos acceder a diferentes estados [13]. Definamos dos máquinas de Turing M1 y M2, siendo M1 determinista y M2 no determinista:

M1:

- $Q = \{q_1, q_2, q_f\}$
- $\Sigma = \{0, 1\}$
- $\Gamma = \{0, 1, \_\}$
- $b = \_$
- $q_0 = q_1$
- $F = \{q_f\}$
- $\delta =$ Cuadro 3.2

Estado	Lee Símbolo	Nuevo Símbolo	Nuevo Estado	Dirección
q_1	0	0	q_1	derecha
q_1	1	1	q_2	derecha
q_2	0	0	q_f	derecha
q_2	1	1	q_f	derecha

Cuadro 3.2: Función de estado M1

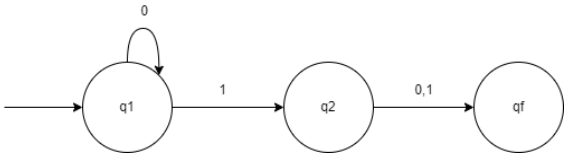


Figura 3.7: Autómata M1

M2:

- $Q = \{q_1, q_2, q_f\}$
- $\Sigma = \{0, 1\}$
- $\Gamma = \{0, 1, \_\}$
- $b = \_$
- $q_0 = q_1$
- $F = \{q_f\}$
- $\delta =$ Cuadro 3.3

Estado	Lee Símbolo	Nuevo Símbolo	Nuevo Estado	Dirección
q_1	0	0	q_1	derecha
q_1	1	1	q_1	derecha
q_1	1	1	q_2	derecha
q_2	0	0	q_f	derecha
q_2	0	0	q_2	derecha
q_2	1	1	q_2	derecha
q_2	1	1	q_f	derecha

Cuadro 3.3: Función de estado M2

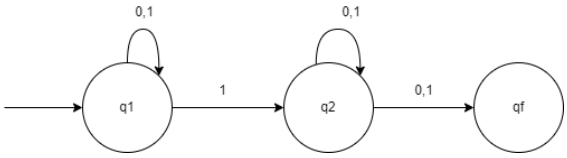


Figura 3.8: Autómata M2

Como podemos ver en los autómatas de las Figuras 3.7 y 3.8, los que consisten en versiones más sencillas de máquinas de Turing, donde no se escribe en la cinta y siempre se realiza un movimiento hacia la derecha, que se corresponden a M1 y M2 respectivamente, la diferencia entre estas dos máquinas es evidente: mientras que en la máquina determinista tiene exactamente una flecha de transición por cada símbolo del alfabeto, la no determinista no, ya que si nos fijamos en el estado q_1, vemos que tiene dos flechas para 1. La pregunta ahora es, que cómo funciona realmente una máquina indeterminista, cómo es posible que pueda leer un único símbolo y acabar en dos estados diferentes. La respuesta, es que cada vez que la máquina se encuentra en la situación de tener varios caminos para un único símbolo, esta se divide en múltiples copias de si misma, representada por una rama en un árbol de posibilidades (Figura 3.9), siguiendo todos estos posibles caminos en paralelo, donde cada una de estas copias seguirá uno de estos caminos, repitiendo este proceso tantas veces como sea necesario. Pero esto tiene un límite, ya que, si un símbolo no aparece en ninguna de las flechas del estado en el que se encuentra una de las copias de la máquina, esta copia se "muere", eliminando consigo la rama que la representa. Finalmente, si una estas ramas consigue aceptar, la máquina acepta la entrada [13].

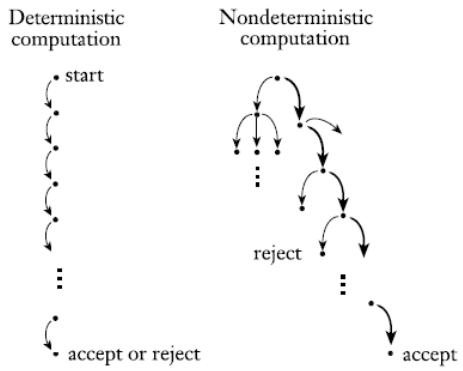


Figura 3.9: Comparación entre máquina determinista y no determinista

Vamos a ver a continuación cómo sería el árbol creado por M2 para la entrada 110: El árbol comienza en el estado q_1 leyendo el símbolo 1. En este punto la máquina tiene dos caminos posibles, mantenerse en q_1 o avanzar a q_2, por lo que la máquina se divide en dos para continuar por cada uno de los caminos posibles. El segundo carácter de la entrada es nuevamente 1, por lo que ambas máquinas tienen dos posibles opciones. La máquina que se encuentra en q_1, se divide una vez más en dos, tomando tanto el camino de q_1, como el de q_2, mientras que la máquina con estado q_2, dará lugar a 2 máquinas con estado q_2 y q_f. El último símbolo de la entrada es 0, así que como podemos ver en la figura 3.10, la primera máquina pasa al único estado posible q_1, la segunda, se divide en q_2 y q_f, al igual que la tercera, mientras que la cuarta no tiene ningún posible camino que tomar, por lo que esta simplemente "muere", indicando que esa máquina no acepta. En este punto con la ejecución terminada, tenemos que comprobar si hay alguna máquina que se encuentre en q_f y haya aceptado. Como podemos ver, existen dos máquinas que están en este estado, por lo que podemos afirmar que la máquina acepta para la entrada 110.

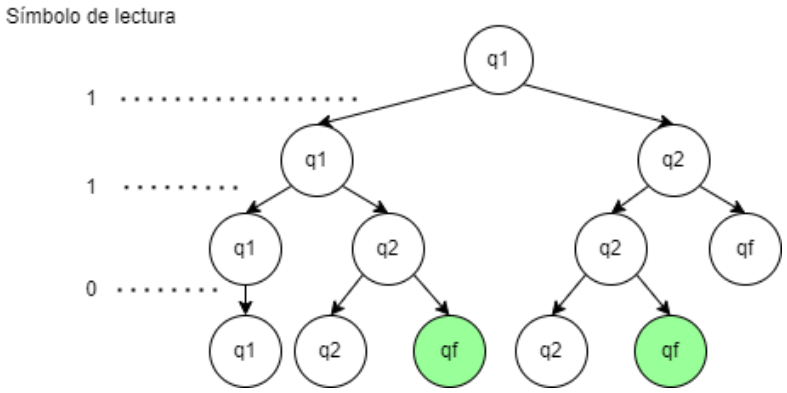


Figura 3.10: Computación realizada por una máquina no determinista

3.5. Notación asintótica

La notación asintótica es una herramienta que nos permite analizar el crecimiento de una función. En la teoría de la complejidad computacional, se usa habitualmente para medir recursos usados por un algoritmo, normalmente espacio (memoria) y tiempo (pasos o número de instrucciones ejecutadas). Una de las más comunes y que más veremos en este trabajo será la llamada notación O grande (O), o *big O notation* en inglés. Esta es utilizada para describir el límite superior asintótico de una función.

Definición 3.5.1. Sean f y g funciones $f, g : N \to R$. Se dice que una función $f(n)$ está en el conjunto $O(g(n))$ si existe una constante positiva c y un valor n_0 tal que para todos los valores n mayores que n_0 el valor de $f(n)$ está acotado por $c \cdot g(n)$ [13]. Es decir, para todo n mayor que n_0 se cumple que:

$$f(n) \leq cg(n)$$

En términos más simples, $f(n)$ está en $O(g(n))$ si a partir de cierto tamaño, crece como mucho, tan rápido como $g(n)$, ignorando términos constantes y de orden inferior.

Algunos ejemplos de diferentes notaciones de O grande ordenados de menor a mayor complejidad son:

- O(1) - Contante: La complejidad no depende del tamaño de la entrada, como en el acceso a un elemento en una lista.
- O(log n) - Logarítmica: La complejidad crece de manera logarítmica, en función del tamaño de la entrada. Un ejemplo sería buscar un elemento en una lista ordenada.
- O(n) - Lineal: La complejidad crece de manera lineal. Al recorrer una lista de n elementos, obtenemos esta complejidad.
- O(nlogn) - Linearítmica: Esta complejidad está entre la lineal y la logarítmica. La obtenemos en algunos algoritmos de ordenación como el Merge Sort.
- O(n^2) - Cuadrática: Al igual que las anteriores, como su nombre indica esta complejidad mantiene una relación cuadrática con el tamaño de la entrada. Podemos encontrarla en algoritmos como Bubble Sort
- O(2^n) - Exponencial: La complejidad crece de forma exponencial en base al tamaño de la entrada, como en un algoritmo de generación de subconjuntos.
- O($n!$) - Factorial: La complejidad crece de forma factorial en base al tamaño de la entrada.

Para calcular cuál es la complejidad final tanto de una función como de un programa, tenemos que tomar el término con mayor exponente, ignorando todas las constantes. Vamos a ejemplificar cómo es el uso de esta notación en la determinación de la complejidad de funciones:

- O(2n + 1): El mayor término en este caso es 2n, pero ignorando las constantes, tanto 2 como 1, la complejidad final sería O(n).

- O($n^3 + n^2 + 10n$): En este caso la complejidad total es O(n^3).
- O($2n^2$+$n!$+2^n): Aquí tenemos varias elementos con complejidad muy alta, pero siguiendo el orden de complejidades previamente definido, podemos afirmar que la complejidad final es O($n!$).

A continuación vamos a ver cómo sería el uso de esta notación, este vez en uno de los algoritmos de ordenamiento recientemente mencionados, el *Bubble Sort*, que consiste en recorrer una lista varias veces, comparando elementos adyacentes y moviendo al final de esta al elemento más grande en cada iteración [8]. Al ser la complejidad de este algoritmo cuadrática en el peor escenario, sería representada como $O(n^2)$, siendo n el número de elementos a ordenar. Aquí podemos ver un ejemplo con código java y pseudocódigo, de cómo sería este algoritmo:

```
public void bubbleSort(int[] array) {
    int n = array.length;
    for (int i = 0; i < n-1; i++) {
        for (int j = 0; j < n-i-1; j++) {
            if (array[j] > array[j+1]) {
                int temp = array[j];
                array[j] = array[j+1];
                array[j+1] = temp;
            }
        }
    }
}
```

3.6. Midiendo complejidad de algoritmos

Existen varias formas de medir la complejidad de un algoritmo, dependiendo del contexto y los aspectos específicos del modelo en el que nos basemos:

- **Complejidad temporal:** este el tiempo que le lleva al modelo realizar una tarea, pudiéndose medir en términos de operaciones básicas como comparaciones o accesos a memoria. Para calcular esta complejidad debemos de tener en cuenta la complejidad de cada una de las partes que forman el algoritmo. Si utilizamos el algoritmo del apartado anterior, tenemos que tener en cuenta todas sus partes:
  - **Calcular la longitud de un array (array.length):** Asumiendo que en este caso el cálculo de la longitud de un array consiste en acceder a una variable con ese valor, y no la llamada a un método que la calcule, podemos decir que es una operación de tiempo constante, lo que quiere decir que independientemente del tamaño de esta, se calculará en un tiempo fijo $O(1)$.
  - **Recorrer una list (for loop):** En el caso de que un bucle "for" tenga una longitud fija, por ejemplo 10, su complejidad será lineal, es decir $O(1)$. Pero si el número de iteraciones de este depende del tamaño de la entrada, como es este caso, su complejidad irá en función del tamaño de la entrada, siendo $O(n)$.

- **Acceder a un elemento de un array (array[i]):** Al igual que calcular la longitud, acceder a un elemento de un array tiene una complejidad constante de $O(1)$.
- **Comparar dos elementos (x>y):** Esta operación tiene una complejidad temporal de $O(1)$, debido a que se realizan utilizando instrucciones de comparación de bajo nivel en el hardware de la computadora, y estas instrucciones generalmente tienen un tiempo de ejecución constante y muy rápido.
- **Asignación (int x = y):** La operación de asignar un valor a otro tiene una complejidad de $O(1)$.

Sabiendo esto, podemos asignar la complejidad a cada una de las líneas del algoritmo:

```
        public void bubbleSort(int[] array) {
            int n = array.length;
// O(1)
            for (int i = 0; i < n-1; i++) {
// O(n)
                for (int j = 0; j < n-i-1; j++) {
// O(n)
                    if (array[j] > array[j+1]) {
// O(1)
                        int temp = array[j];
// O(1)
                        array[j] = array[j+1];
// O(1)
                        array[j+1] = temp;
// O(1)
                    }
                }
            }
        }
```

Para calcular la complejidad total de este algoritmo, debemos tomar la complejidad más alta de los elementos que se encuentran al mismo nivel y multiplicarla por la de los elementos de menos nivel. Empezando por el nivel más bajo, vemos que todas las complejidades son $O(1)$, por lo que esa será la complejidad del nivel. Subiendo un nivel, tenemos $O(n)$, debido al segundo bucle "for". Si le añadimos la complejidad del nivel inferior, nos quedamos con $O(n)$. Por último, la complejidad mayor del primer nivel es $O(n)$. En este caso, al multiplicarla por la de su nivel inmediatamente inferior obtenemos la complejidad total del algoritmo: $O(n^2)$.

A continuación vamos a ver otro ejemplo de cómo calcular la complejidad temporal, en este caso con la máquina de Turing del apartado 3.3. Como podemos ver en la Figura 3.11, la relación entre el tamaño de la entrada y la cantidad de pasos que la máquina necesita para determinar si la entrada es o no un número palíndromo es cuadrática, por lo tanto podemos decir que la máquina de Turing tiene una complejidad temporal $O(n^2)$. Esta complejidad se mide con el número de pasos que tiene que realizar la máquina de Turing con una entrada de tamaño n.

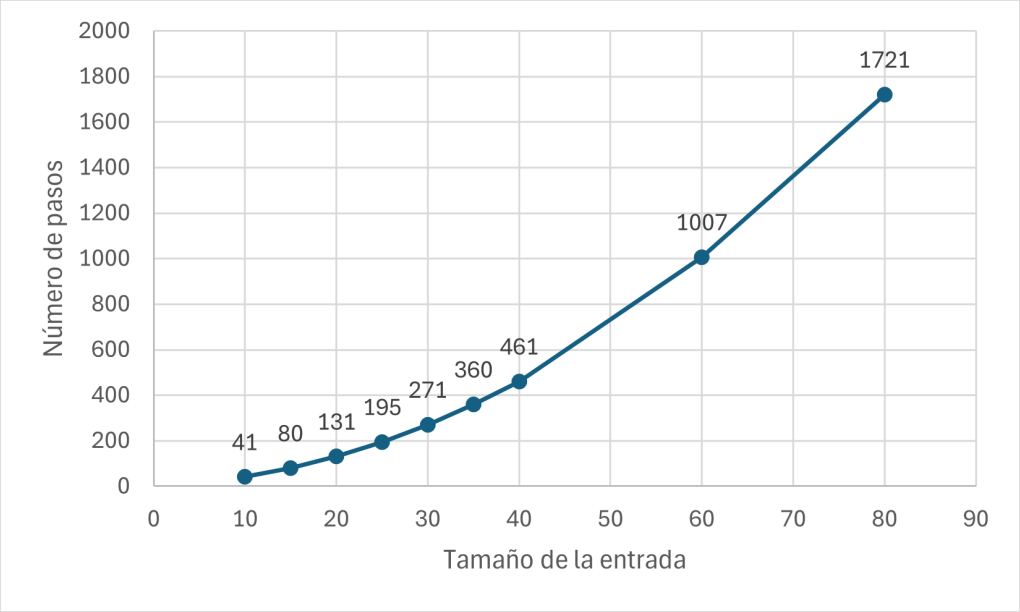


Figura 3.11: Análisis de la complejidad de máquina de Turing

- **Complejidad espacial:** la complejidad espacial es aquella que determina la cantidad de memoria o espacio de almacenamiento que requiere el modelo para realizar una tarea determinada. Veamos un ejemplo de esto:

  El siguiente algoritmo realiza la suma de todos los elementos de la lista pasada como parámetro:

```
public int sumaLista(List<int> lista){
    int suma = 0;
    for( int i : lista ){
        suma += i;
    }
    return suma;
}
```

  Para saber cuál es la complejidad espacial de este algoritmo, debemos de fijarnos en cuánta memoria necesita tomar para realizar la operación. En este caso, requiere una cantidad constante de memoria, ya que la única variable que tiene que crear es suma, donde almacena la suma de todos los elementos de la lista, por lo que la complejidad espacial es de $O(1)$

  Pero pueden existir algoritmos con una complejidad espacial mucho mayor, como por ejemplo el siguiente:

```
public ArrayList<ArrayList<Integer>> generarCombinaciones(
int[] elementos) {
    int n = elementos.length;
    ArrayList<ArrayList<Integer>> combinaciones;
    combinaciones = = new ArrayList<>();

    for (int i = 0; i < (1 << n); i++) {
        ArrayList<Integer> combinacion = new ArrayList<>();
        for (int j = 0; j < n; j++) {
```

```
                if ((i >> j & 1) == 1) {
                    combinacion.add(elementos[j]);
                }
            }
            combinaciones.add(combinacion);
        }
        return combinaciones;
    }
```

Este algoritmo genera todas la posibles permutaciones de los elementos de la lista pasada como parámetro, guardando cada una de ellas como una lista, en la lista de listas combinaciones, por lo que la complejidad espacial de este algoritmo crece de manera mucho mayor que exponencial con cada nuevo elemento de la lista, es decir, con una complejidad $O(n! * n)$

Habiendo definido estas formas de calcular la complejidad de un problema computacional, queremos ahora definir la complejidad computacional de un problema. Uno de los conceptos principales de toda la teoría de complejidad computacional es que en última instancia la complejidad de un problema siempre será la complejidad del mejor algoritmo que lo resuelve. Además todo este análisis de complejidad en funciones, puede ser trasladado a una máquina de Turing, ya que como hemos visto anteriormente, en el caso de la complejidad temporal podemos establecer una relación entre el tamaño de la entrada y la complejidad de esta. Por otro lado, para la complejidad espacial de una máquina de Turing solamente tenemos que tener en cuenta el número de celdas que se usan durante la computación.

Capítulo 4

Clase de complejidad P

Uno de los puntos clave de la teoría de la complejidad computacional es el estudio de qué problemas se pueden resolver en un tiempo eficiente. Si agrupamos todos estos problemas que pueden ser resueltos de manera eficiente, es decir, que invertir tiempo y recursos en ello sea algo con sentido, obtenemos la clase de complejidad P.

4.1. Descripción de la clase de complejidad P

La clase de complejidad P consiste en todos los problemas de decisión que pueden ser resueltos por un máquina de Turing determinista de una cinta en un tiempo polinomial, dicho de una manera formal [13]:

$$P = \bigcup_k Tiempo(n^k)$$

donde $Tiempo(n^k)$ son todos aquellos problema que se pueden solucionar en $O(n^k)$.

P juega un papel fundamental en la teoría de complejidad computacional debido a que se identifica con todos aquellos problemas que pueden ser resueltos eficientemente, mientras que aquellos que no están en P se considera que no se pueden resolver eficientemente. Además de esto, conocer la constante k, nos permite saber si la resolución de dicho problema se puede lograr en un tiempo práctico, para el que merezca la pena gastar tiempo y recursos, o si simplemente, aunque el problema se pueda resolver, tardaría demasiado tiempo y consumiría una cantidad demasiado grande de recursos.

4.2. Ejemplos de problemas en P

Un ejemplo de problema de la clase P es el utilizado anteriormente en el apartado 3.3, decidir si un número es o no palíndromo, ya que como hemos visto, su complejidad es $O(n^2)$, por lo que cumple con la definición de P para $k = 2$. A continuación vamos a ver en detalle las demostraciones de por qué varios problemas pertenecen a la clase P:

4.2.1. PATH

Este problema consiste en decidir si un grafo tiene un camino que una dos vértices seleccionados. Para la entrada $\langle G, s, t \rangle$ donde G es un grafo dirigido G(V, E) y

$s, t \in V$, ¿existe un camino en el grafo en el que saliendo de s podamos llegar a t? Pongamos que G tiene un número n de vértices, entre los que se encuentran s y t, en este caso el número máximo de caminos que puede haber es de n^n, mientras que la complejidad que se necesita para comprobar si un camino es el que buscamos, es de $O(n)$, ya que solamente tendría que recorrer la lista de vértices una vez. Por ello si aplicamos un algoritmo de fuerza bruta, no podríamos decir que $PATH \in P$. Pero esto no quiere decir que $PATH \notin P$, sino que simplemente este algoritmo de fuerza bruta no demuestra $PATH \in P$. Vamos a ver qué pasaría con un algoritmo más eficiente: Breath Fisrt Search. Para ello, vamos a ver cómo sería el pseudocódigo de dicho algoritmo, y estudiar tanto la complejidad de sus partes como la total.

```
Crear una lista vacia B
Marcar s como visitado y meter en B todos sus adyacentes //O(n)
Mientras B no sea vacio                                  //O(n)
    Sacar un nodo a de B                                 //O(1)
    Si a no esta marcado                                 //O(1)
        Marcar                                           //O(1)
        Meter en B todos sus adyacentes no marcados      //O(n)
Si t esta marcado                                        //O(1)
    existe camino
```

Debido a que con este algoritmo solamente tenemos que visitar cada vértice una única vez, la complejidad del algoritmo será de $O(n^2)$, por lo que podemos afirmar $PATH \in P$.

4.2.2. Dijsktra

En algunos problemas de optimización también hay casos en los que se puede evitar la fuerza bruta. El algoritmo de Dijkstra, aplicado para la resolución del problema del camino mínimo, consiste en un algoritmo que calcula el recorrido más corto desde un vértice de un grafo a todos los demás vértices de este. Debe su nombre Edsger Dijkstra, quien lo publicó en 1959. La implementación de este algoritmo sería:

```
Nodo inicial A                                           //O(1)
Vector de costes D                                       //O(1)
Vector de nodos no visitados NV                          //O(1)

Mientras NV no este vacio {                              //O(n)
Seleccionar el nodo B mas cercano a A y quitarlo de NV   //O(1)
Para cada nodo C adyacente a B                           //O(n)
    Actualizar D, si la suma de la distancia de A a B,
        y la de B a C, es menor que A a C.               //O(1)
}
```

En este algoritmo, solo tenemos que visitar cada nodo una vez, y dentro de ese nodo, como mucho tendremos que visitar el resto de nodos una única vez, por lo que la complejidad del algoritmo de Dijkstra es $O(n^2)$.

La existencia de ejemplos como el que acabamos de ver nos motiva a hacernos la pregunta si siempre podemos evitar la fuerza bruta. Veamos en el siguiente apartado si esto es posible.

Capítulo 5

Clase de complejidad NP

Como hemos visto en el apartado anterior, para algunos problemas podemos evitar algoritmos de fuerza bruta y utilizar soluciones polinomiales, pero esto no siempre es posible. Hoy en día no conocemos por qué no se ha podido encontrar una solución polinomial a todos los problemas computacionales , si bien tienen una que aún no ha sido encontrada, o si simplemente esta no existe.

5.1. Descripción de la clase de complejidad NP

NP es la clase de problemas de decisión que pueden ser verificados en un tiempo polinomial, lo que quiere decir que tiene un verificador que se ejecuta en un tiempo polinomial. Este verificador consiste en un algoritmo que toma una instancia del problema original junto con una posible solución como información adicional llamada prueba o certificado. Este algoritmo verifica si la solución es correcta o no [13]. Es decir, un verificador para un problema P es un algoritmo V tal que, si la respuesta de una instancia w de P es positiva, entonces existe una palabra c tal que V acepta cuando se le da la entrada w y c. Además, si la respuesta de w es negativa, entonces V no aceptará ninguna entrada ⟨ w,c ⟩. Podemos ver un ejemplo menos abstracto de qué es un verificador en el apartado 5.3.

El nombre NP viene del término ***tiempo Polinomial No-determinista***, ya que otra definición de NP es aquellos problemas que pueden ser decididos por una máquina de Turing no determinista en un tiempo polinomial. Vamos a demostrar que ambas definiciones de NP son equivalentes:

Sea A un problema de NP. Vamos a demostrar que, a su vez, A es decidible en tiempo polinomial por una máquina de Turing no determinista N. V es el verificador polinomial de A que existe por definición de NP. Asumamos ahora que V es una máquina de Turing que se ejecuta en un tiempo n^k y construyamos N de la siguiente manera:

Cuando N se ejecuta con una entrada w de tamaño n, hace lo siguiente:

1. De manera no determinista selecciona la palabra c de longitud como mucho n^k.
2. Ejecuta V para la entrada $\langle w, c \rangle$.
3. Si V acepta, acepta, si no, rechaza.

Lo que esto quiere decir, es que, si tenemos un verificador, podemos construir una máquina de Turing no determinista que compruebe todos los posibles certificados o pruebas en paralelo y acepta si al menos una de las ramas acepta.

Para demostrar la otra dirección del teorema, asumamos que A es decidido en un tiempo polinomial por una máquina de Turing no determinista N y construyamos el verificador en tiempo polinomial V de la siguiente manera:

Cuando V recibe la entrada $\langle w, c \rangle$, donde w y c son palabras, hace lo siguiente:

1. Simula la ejecución de N con entrada w. Cada vez que tiene que tomar una decisión no determinista, usa un bit de c para elegir una de las opciones. Esto creará una única rama de la computación de N para cada posible c.

2. Si esta rama de N acepta, acepta; si no, rechaza [13].

Lo que quiere decir que, si tenemos una máquina no determinista, sabemos que para que esta acepte tenemos que tener una secuencia de elecciones (rama) que llevan hasta ese estado en el que acepta. En este caso el verificador es una máquina de Turing que simula la máquina previamente nombrada, mientras que el certificado no es más que la secuencia de decisiones que ha tenido que tomar la máquina para llegar a aceptar.

La clase NP es muy importante porque en ella se encuentran problemas de una gran relevancia práctica y teórica, como los que veremos en el apartado 6.4.

5.2. Significado e implicaciones de la conjetura P != NP

Existen un montón de problemas como los que veremos en los siguientes apartados, como el problema de la satisfabilidad (SAT) que son miembros de NP, pero no sabemos si son o no miembros de P. El poder y la cantidad de problemas que alberga NP puede parecer mucho mayor que el de P, pero aunque parezca difícil de entender, estos dos podrían ser iguales, ya que hasta ahora ha sido imposible demostrar la existencia de un problema que esté en NP pero no en P.
La pregunta de si P = NP es uno de los mayores problemas sin respuesta en toda la rama teórica de la computación. Si estas clases fuesen iguales supondría que:

1. Todos los problemas que se pueden verificar en tiempo polinomial, podrían ser resueltos en un tiempo polinomial, por lo que problemas que a día de hoy se consideran irresolubles debido a la complejidad y tiempo de cálculo, podrían ser resueltos, revolucionando tanto la informática como toda la ciencia en general.

2. Los sistemas de seguridad informática que tenemos a día de hoy dejarían de ser seguros, ya que estos se basan en la dificultad de resolver algunos problemas de NP que no sabemos si están en P, llamados problemas NP-completos (que veremos en el apartado 5.4), suponiendo esto un cambio de paradigma y un replanteamiento de todo lo establecido en este campo.

Sabemos que todos los problemas de P están en NP, ya que si se puede resolver un problema en tiempo polinomial con una máquina determinista también se puede hacer con una no determinista. Por ello las dos únicas posibilidades que existen son que P y NP sean iguales, o que P sea un subconjunto de NP, como muestra la Figura 5.1.

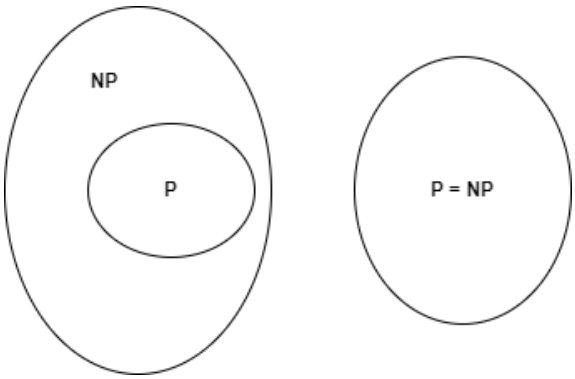


Figura 5.1: Conjetura P != NP

5.3. Ejemplos de verificadores

Vamos a ver con un ejemplo para entenderlo de una forma menos abstracta cómo actúa un verificador como los descritos al hablar de la clase NP.

Pongamos que nos dan un número como el 231 y nos preguntan si es factorizable. Sin ningún tipo de información adicional, una posible estrategia podría ser hacer la raíz cuadrada de 231 ($\pm 15, 19868$), redondear al número entero menor en valor absoluto (15) y dividir 231 entre todos los números desde 15 hasta 2 para comprobar si alguno nos da resto 0. Pero si en un primer momento nos dan como certificado adicional el número 11, simplemente tendremos que hacer la operación $231/11 = 21$ para así verificar que 231 es un número factorizable. En este caso la prueba es el número 11. Por otro lado, el verificador no es más que una máquina de Turing que divide 231 entre 11, por lo que ahora podemos entender que la verificación de un problema en casos como este es un proceso sencillo con muy poca complejidad.

Ahora vamos a ver otro ejemplo de un verificador, en este caso para un problema de un camino hamiltoniano.

Tomemos como ejemplo la Figura 3.1, donde podemos ver un grafo compuesto por nueve nodos desde V1 hasta V9. Debido a que en este problema existen $n!$ secuencias de nodos que podrían ser el camino que buscamos, la cantidad total de secuencias que deberíamos calcular si empleamos una aproximación de fuerza bruta es más de 300000. Pero si al igual que en el ejemplo anterior nos dan una información adicional, en este caso, la lista de nodos que debemos seguir (V1, V2, V5, V9, V8, V4, V7, V6, V3), solamente tendremos que comprobar que efectivamente ese camino es hamiltoniano, lo que se puede realizar en un tiempo polinomial.

5.4. Problemas NP-completos y NP-duros

En la década de 1970 Stephen Cook y Leonid Levin hicieron un gran avance en la pregunta P vs NP. Este consistió en el descubrimiento de ciertos problemas en NP cuya complejidad está relacionada con toda la clase NP, de manera que, si un algoritmo capaz de resolver en tiempo polinomial existía para uno de esos problemas, todos los problemas de NP podrían ser resueltos en tiempo polinomial. Estos problemas son los llamados **NP-completos**. Para un problema A:

$$\text{Si A es NP-completo y } A \in P, \text{ entonces P} = \text{NP}.$$

Desde un punto de vista teórico, estos problemas ayudan a que una persona que investiga acerca de P vs NP pueda centrarse en los problemas NP-completos, mientras que desde un punto de vista práctico, puede prevenir el desperdicio de recursos en la búsqueda de un algoritmo polinomial para resolver un determinado problema [13].

Las características que tiene que cumplir un problema B para ser considerado NP-completo son:

1. **Ser un problema de decisión:** La respuesta a B tiene que ser binaria, sí o no.
2. Está en NP, es decir, la solución puede ser **verificada** en un tiempo polinomial.
3. Cualquier otro problema A de NP se puede reducir en un tiempo polinomial a B (hablaremos de qué es una reducción enseguida).

Por otro lado, tenemos los problemas NP-duros (NP-hard), que son aquellos que tienen por lo menos el mismo nivel de dificultad que los problemas de NP. Estos no tienen por qué cumplir la condición de ser verificados en tiempo polinomial, pero sí la condición de que cualquier problema de NP pueda reducirse a un problema NP-duro. Algunos de los problemas que se encuentran en esta clase de complejidad son el *Problema de la mochila o el Problema del viajante de comercio* que veremos con mayor detalle en el apartado 6.4.

La relación entre estos dos tipos de problemas es que aquellos problemas de decisión NP-duros que sí que pueden ser verificados en un tiempo polinomial son los problemas NP-completos. Como podemos ver en la Figura 5.2 de una manera más visual, estas son las 2 posibilidades que existen en cuanto a los problemas NP-duros y NP-completos. Por un lado, si $P = NP$ significaría que $NP = NP - completo$, siendo todos ellos una parte de los problemas NP-duros. Por otro lado, si $P \neq NP$, los problemas NP-completos no representarían todo NP, sino que solamente serían los problemas NP-duros pertenecientes a NP

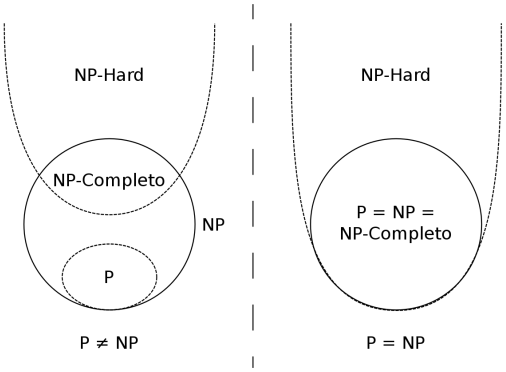


Figura 5.2: P, NP, NP-hard y NP-completo. [16]

Capítulo 6

Métodos para la clasificación de problemas

6.1. Reducciones polinómicas

Uno de los métodos más usados para clasificar la dificultad relativa de varios problemas es transformar uno en una instancia de otro, de manera que podamos aplicar algoritmos que funcionan sobre el segundo para resolver también el primero. A esto se le conoce como reducciones. Cuando tenemos en cuenta la eficiencia de una reducción, siendo un problema A reducible en un tiempo eficiente a B, una solución en un tiempo eficiente de B puede ser utilizada para solucionar A en un tiempo eficiente. Si cambiamos eficiente por polinomial, nos encontramos por las llamadas reducciones polinómicas. Se dice que un problema A es reducible en tiempo polinomial a un problema B, escrito como $A \leq_p B$, si existe una función f tal que para todo w se cumple que

$$w \in A \iff f(w) \in B,$$

y f se puede calcular en tiempo polinomial. Lo que estamos haciendo es transformar mediante f instancias w del problema A en instancias f(w) del problema B. Además, la instancia w de A tiene solución positiva (respuesta "sí") si y sólo si la instancia f(w) de B tiene respuesta positiva.

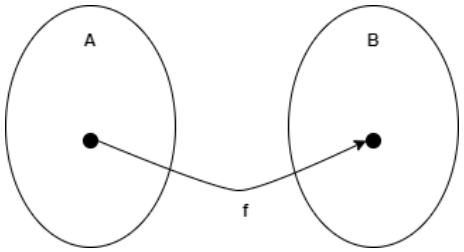


Figura 6.1: Reducción polinomial de A a B

6.2. Ejemplos de reducciones comunes

6.2.1. SAT

El problema de la satisfabilidad booleana es uno de los problemas más importante de toda la teoría de complejidad, siendo este el primero en ser identificado como parte

de la clase de problemas NP-completos. SAT consiste en el problema de saber si en una expresión booleana con n variables, hay algún valor de estas que haga verdadera a la expresión. El teorema Cook-Levin, que veremos en el apartado 6.3, demuestra que SAT pertenece a la clase de problemas NP-completos. A continuación vamos a ver diferentes variantes de SAT que nos ayudarán a entender mejor de qué se trata este problema.

6.2.2. CIRCUIT-SAT

Pongamos como ejemplo a seguir durante todas las instancias de SAT la expresión $((x_1 \wedge x_2) \wedge (x_2 \vee x_3) \vee (\neg(x_1 \wedge x_2)))$. Esta puede representarse como un circuito booleano C, siendo este un grafo acíclico dirigido, con un número n de entradas $x_1, ..., x_n$, en este caso x_1, x_2, x_3, como podemos ver en la Figura 6.2. C tiene una única salida, representada por el nodo que carece de flecha de salida.

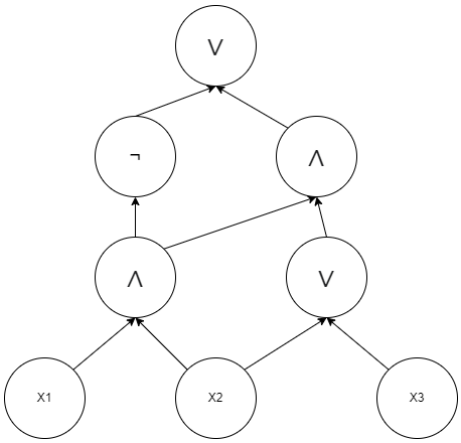


Figura 6.2: circuito C

C representa una función booleana f_c, que mapea n entradas, en este caso 3, en una única salida.

$$f_c : \{0,1\}^n \rightarrow \{0,1\}$$

El problema CIRCUIT-SAT consiste en, dado un circuito C, determinar si hay alguna combinación de entradas que hagan que la salida sea TRUE. Para resolver este problema la mejor opción que tenemos a día de hoy es usar un algoritmo de fuerza bruta que compruebe todas las posibles combinaciones de valores, con una complejidad que puede ser exponencial en la entrada. En caso de que CIRCUIT-SAT $\in P$, podríamos afirmar que $P = NP$ ya que se sabe que este es un problema NP-completo.

6.2.3. FORMULA-SAT

Otra de las variantes de SAT es FORMULA-SAT, problema que consiste en un tipo de CIRCUIT-SAT en el que dada una fórmula en la que todos los nodos tiene una única salida, se busca determinar una asignación de variables que haga la fórmula verdadera. Debido a esto, no podemos reutilizar expresiones como en CIRCUIT-SAT, por lo que la representación del ejemplo del problema anterior sería:

$$((x_1 \wedge x_2) \wedge (x_2 \vee x_3) \vee (\neg(x_1 \wedge x_2)))$$

6.2.4. CNF-SAT

El problema Forma Normal Conjuntiva - SAT (Conjuntive Normal Form - SAT) es una simplificación de FORMULA-SAT, donde la estructura de la fórmula es una secuencia de cláusulas o sus negaciones, que consisten en secuencias de variables unidas por OR, todas ellas unidas por AND, donde al igual que en los ejemplos anteriores se busca una asignación de las variables que haga la fórmula verdadera. Una fórmula en CNF sería, por ejemplo:

$$(x_1 \vee x_2) \wedge (x_2 \vee x_3 \vee x_4)$$

6.2.5. K-SAT

K-SAT es un caso especial de CNF-SAT, donde todas las cláusulas tienen como mucho un número k de variables. Por ejemplo, una instancia de 2-SAT sería:

$$(x_1 \vee x_2) \wedge (x_2 \vee x_3)$$

6.2.6. 3-SAT

Este es un caso especial de K-SAT, donde todas las cláusulas tienen como mucho 3 variables. Por ejemplo, podríamos tener:

$$(x_1 \vee x_2 \vee x_3) \wedge (x_4 \vee x_5 \vee x_6)$$

De la misma forma que hemos visto en instancias anteriores de SAT, intentar resolver este problema con un algoritmo de fuerza bruta tendría una complejidad de $O(2^n)$. Sin embargo, existe un algoritmo capaz de resolver 3-SAT en un tiempo $O(1{,}34^n)$, descubierto por Uwe Schöning [12]. Este sigue sin ser un tiempo polinomial pero es mucho mejor que el tiempo $O(2^n)$ que se tardaría en resolverlo con fuerza bruta. El algoritmo de Schöning consiste en:

```
Asignar de manera aleatoria valores a las variables
Mientras no hayamos encontrado una asignacion que satisfaga
la formula{

    Evaluar la formula con los valores actuales

    Si la formula es correcta{
        Devolver los valores actuales
    }

    Si no es correcta{
        Elegir aleatoriamente una de las clausulas erroneas
        Elegir aleatoriamente un literal de esa clausula y
        cambiar su valor
    }
}

Si no se encuentra solucion despues de un numero determinado
de iteraciones se asume que la formula no es satisfacible
```

Este es un algoritmo aleatorio, por lo que no puede ser implementado por una máquina determinista. Además este tiene una cierta probabilidad de error que se puede hacer tan baja como se desee aumentando el número de iteraciones.

6.3. Teorema de Cook-Levin

Como ya habíamos mencionado anteriormente, el teorema Cook-Levin demuestra que SAT es NP-completo. Para probarlo, necesitamos demostrar que todo problema de NP es reducible en tiempo polinómico a SAT. Obviamente hay una cantidad infinita de problemas en NP, por lo que no se puede mostrar haciendo una reducción polinómica de todos ellos por separado. La única información que poseemos es que $B \in NP$, por lo que sabemos que B tiene una máquina de Turing no-determinista capaz de resolverlo en un tiempo polinomial $O(n^k)$. Sabiendo también que la reducción tiene que ser $w \in B \iff f(w) \in SAT$, significa que tenemos que demostrar la existencia de una fórmula $f(w)$ que, de alguna forma, simula a la máquina de Turing. Vamos a representar la reducción con la tabla de la Figura 6.3 [13].

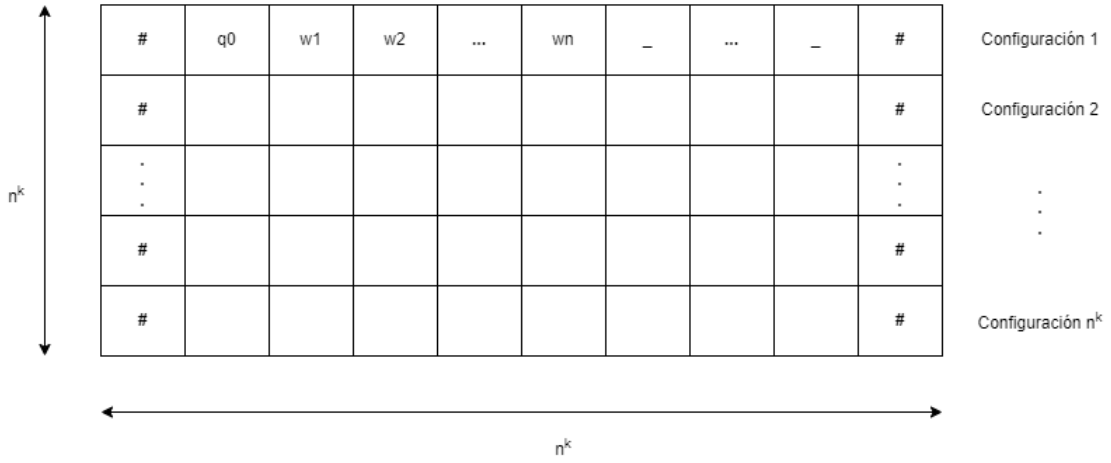


Figura 6.3: Reducción

En este cuadro, cada fila representa una posible configuración de la máquina de Turing. En ellas, se representan los símbolos que se encuentran en las celdas de la máquina junto con el estado en el que se encuentra la máquina, que se coloca justo a la izquierda del símbolo que está leyendo. La primera fila representa el estado inicial de la máquina, mientras que la última, representa un estado en el que la máquina acepta. Como este cuadro es una representación de la fórmula que queremos encontrar, vamos a definir las características de este como la fórmula que queremos:

- ϕ_1: Cada casilla de la tabla tiene exactamente un valor.
- ϕ_2: La primera fila es el estado inicial de la máquina.
- ϕ_3: La última fila es un estado en el que acepta.
- ϕ_4: Cada fila cede el paso a la siguiente.

De manera que la fórmula que buscamos sería:

$$\phi = \phi_1 \wedge \phi_2 \wedge \phi_3 \wedge \phi_4$$

En primer lugar vamos a codificar las variables que representan los valores que pueden aparecer en el cuadro como: $x_{i,j,s}$, donde i representa la fila, j la columna, y s el valor en esa determinada posición. Vamos ahora a completar cada una de las partes de esta fórmula [14]:

- ϕ_1: Para codificar que cada casilla tiene un único valor, tenemos que comprobar todas las filas y columnas entre 1 y n^k, haciendo un *AND* para todas ellas ($\bigwedge_{1\leq i,j\leq n^k}$). Para cada una de estas casillas, tenemos que comprobar que tiene exactamente un valor. Para ello vamos a comprobar que tiene por lo menos un valor con un *OR* que compruebe todos los valores posibles que puede tomar la casilla correspondiente (C), ya que con que una de estos posibles valores coincida, esta parte de la fórmula se cumplirá ($\bigvee_{s\in C} x_{i,j,s}$). Por otro lado tenemos que comprobar que no tiene 2 valores, iterando todos los posibles pares de valores diferentes, asegurando que estos no están asignados a la vez ($\bigwedge_{s,t\in Cs\neq t}(\neg x_{i,j,s} \vee \neg x_{i,j,t})$). Juntando todo esto, obtenemos la siguiente fórmula:

$$\phi_1 = \bigwedge_{1\leq i,j\leq nk} [(\bigvee_{s\in C} x_{i,j,s}) \wedge (\bigwedge_{s,t\in Cs\neq t} (\neg x_{i,j,s} \vee \neg x_{i,j,t}))] \tag{6.1}$$

- ϕ_2: Para comprobar que la primera fila es el estado inicial de la máquina, tenemos que asegurarnos que cumple con la forma que hemos determinado en la Figura 6.3. Esta tiene que comenzar por # ($x_{1,1,\#}$), seguida de q_0 ($x_{1,2,q_0}$), seguida de la entrada (w) de la máquina ($x_{1,3,w_1} \wedge ... \wedge x_{1,n+2,w_n}$), seguida de espacios en blanco ($x_{1,n+3,\_} \wedge ... \wedge x_{1,n^k-1,\_}$), y terminar con # ($x_{1,n^k,\#}$).

$$\phi_2 = x_{1,1,\#} \wedge x_{1,2,q_0} \wedge x_{1,3,w_1} \wedge ... \wedge x_{1,n+2,w_n} \wedge x_{1,n+3,\_} \wedge ... \wedge x_{1,n^k-1,\_} \wedge x_{1,n^k,\#} \tag{6.2}$$

- ϕ_3: En este caso, la única condición que se debe cumplir es que en algún punto de la fila haya un estado q_{accept}. Esto no quiere decir que tenga que ser la última fila de la tabla, sino la última fila que representa a la máquina de Turing, pudiendo ser esta cualquier fila de la tabla.

$$\phi_3 = \bigvee_{1\leq i,j\leq n^k} x_{i,j,q_{accept}} \tag{6.3}$$

- ϕ_4: Que cada fila ceda paso a la siguiente, significa que la fila n y la fila n+1, tienen que ser iguales, salvo por exactamente una ventana, la que codifica una transición valida de la máquina de Turing. Podemos ver el ejemplo de una ventana en la Figura 6.4, donde se produce una transición en la que la máquina se mueve a la izquierda, cambia de q_1 a q_2 y escribe una c. Para comprobar que esto es verdadero en toda la tabla, tenemos que recorrer esta ($\bigwedge_{1\leq i,j\leq n^k}$), y comprobar cada una de las posible ventanas de 2x3 ($\bigvee_{a_1,...,a_6}(x_{i,j-1,a_1} \wedge x_{i,j,a_2} \wedge x_{i,j+1,a_3} \wedge x_{i+1,j-1,a_4} \wedge x_{i+1,j,a_5} \wedge x_{i+1,j+1,a_6})$).

$$\phi_4 = \bigwedge_{1\leq i,j\leq n^k} (\bigvee_{a_1,...,a_6} (x_{i,j-1,a_1} \wedge x_{i,j,a_2} \wedge x_{i,j+1,a_3} \wedge x_{i+1,j-1,a_4} \wedge x_{i+1,j,a_5} \wedge x_{i+1,j+1,a_6})) \tag{6.4}$$

a	q_1	b
q_2	a	c

Figura 6.4: Ventana 2x3

La fórmula final que obtendríamos y que simularía la máquina de Turing no-determinista, sería:

$$\phi = (\bigwedge_{1 \leq i,j \leq nk} [(\bigvee_{s \in C} x_{i,j,s}) \wedge (\bigwedge_{s,t \in C s \neq t} (\neg x_{i,j,s} \vee \neg x_{i,j,t}))]) \bigwedge$$

$$(x_{1,1,\#} \wedge x_{1,2,q_0} \wedge x_{1,3,w_1} \wedge ... \wedge x_{1,n+2,w_n} \wedge x_{1,n+3,\_} \wedge ... \wedge x_{1,n^k-1,\_} \wedge x_{1,n^k,\#}) \bigwedge$$

$$(\bigvee_{1 \leq i,j \leq n^k} x_{i,j,q_{accept}}) \bigwedge$$

$$(\bigwedge_{1 \leq i,j \leq n^k} (\bigvee_{a_1,...,a_6} (x_{i,j-1,a_1} \wedge x_{i,j,a_2} \wedge x_{i,j+1,a_3} \wedge x_{i+1,j-1,a_4} \wedge x_{i+1,j,a_5} \wedge x_{i+1,j+1,a_6})))$$

Habiendo demostrado la existencia de la fórmula que se puede hallar en tiempo polinomial que simula la máquina de Turing, y que demuestra la veracidad de que para todo $B \in NP$, se verifica que $B \leq_p SAT$, podemos afirmar que SAT es $NP - completo$.

6.4. Problemas NP-completos y NP-duros clásicos

En este apartado vamos a ver a fondo algunos de los ejemplos más comunes de los problemas NP-completos y NP-duros.

6.4.1. Problema del cubrimiento de conjuntos

El problema del cubrimiento de conjuntos, también conocido como SCP (Set Covering Problem) es uno de los problemas clásicos del campo de la complejidad computacional que ha permitido el desarrollo de técnicas fundamentales para todo lo relacionado con los algoritmos de aproximación. Este problema se formula de la siguiente manera:

Dado un conjunto de elementos $\{1, ..., m\}$ (universo) y n conjuntos cuya unión comprende el universo, ¿cuál es el menor número de conjuntos cuya unión aún contiene todos los elementos del universo? En el universo $U = \{6, 7, 8, 9\}$ y los conjuntos $S = \{\{6, 7\}, \{7, 8, 9\}, \{8, 9\}\}$, la unión de todos los conjuntos de S contiene a todo U, pero podemos cubrir todos los elementos de U, simplemente con los conjuntos $\{6, 7\}$ y $\{8, 9\}$. Formulando el problema de una manera formal tenemos que para un universo $\mathcal{U}$ y la familia $\mathcal{S}$ de subconjuntos de $\mathcal{U}$ una cobertura es una subfamilia $\mathcal{C} \subseteq \mathcal{S}$ de subconjuntos cuya unión es $\mathcal{U}$, $c(S)$ es el costo asociado al subconjunto S y x_s una variable binaria que indica si S está o no en la solución, donde buscamos:

- Minimizar el coste total

$$\sum_{S \in \mathcal{S}} c(S) x_S \tag{6.5}$$

- Cumpliendo que se cubran todos los elementos del universo:

$$\sum_{S:e \in S} x_S \geq 1 \forall e \in \mathcal{U} \tag{6.6}$$

6.4.2. Problema de la mochila

El problema de la mochila, también conocido como KP (Knapsack Problem), es un problema de optimización combinatoria, en el que se modela una situación en la que al llenar una mochila capaz de soportar un peso determinado se debe maximizar el valor total de los objetos introducidos en esta, sin exceder el peso máximo. Definamos el problema en función de los siguientes parámetros:

- Conjunto de **n** artículos.
- 1, 2, . . . , n
- El peso de cada artículo viene dado por un número w_j
- El valor de cada artículo viene dado por un número p_j
- LA capacidad de la mochila viene dada por un número c

Para resolver este problema lo que necesitamos es:

- Representaremos la decisión de si incluir el artículo j dentro de la mochila mediante la variable booleana x_j. Si $x_j = 1$, meteremos el artículo y lo dejaremos fuera si $x_j = 0$.
- Maximizar el valor de todos los artículos seleccionados:

$$\sum_{j=1}^{n} p_j x_j \tag{6.7}$$

- De manera que la suma de los pesos no exceda el peso de la mochila:

$$\sum_{j=1}^{n} w_j x_j \leq c \tag{6.8}$$

Este problema tiene una gran cantidad de aplicaciones como el uso de contenedores en aduanas, donde se envían objetos de diferentes tamaños y pesos que aportan diferentes beneficios. También puede ser muy útil en todo lo relacionado con la planificación de recursos, donde aquellos bienes limitados como la mano de obra, equipamiento o fondos, deben ser utilizados de la manera más eficiente posible [4].

6.4.3. Problema del viajante de comercio

El problema del viajante de comercio, *Travelling Salesman Problem (TSP)* en inglés, consiste en el problema de un comerciante que desea recorrer una serie de ciudades, visitar cada una exactamente una vez, volver al punto de partida, y minimizar la distancia recorrida.

La cantidad de rutas que se pueden tomar en un grafo con n elementos es de n!. Teniendo en cuenta que como teniendo una ruta no nos importa el punto de partida, y tampoco la dirección de esta, el problema queda reducido a (n-1)!/2. Aun así, el número de caminos posibles para un problema con 9 ciudades sería de 20160 y crece desorbitadamente con el número de ciudades.

Vamos a formular este problema de manera algebraica. Sea $G = (N, A)$ un grafo completo, donde N es el conjunto de nodos y A es conjunto de arcos. Consideremos al nodo 1 como el punto de partida, mientras que los nodos $2, ..., n$, son todos los que tendremos que visitar. A cada arco (i, j) se le asocia un valor no negativo $d_{i,j}$. Definiendo las variables binarias de decisión $x_{i,j}$, que toman valor 1 o 0 si es arco (i, j) está o no en la solución, tenemos la función de coste que representa al problema:

$$\sum_i \sum_j d_{i,i} x_{i,j} \tag{6.9}$$

Tenemos que imponer la restricción de que de cada vértice i solo puede salir un arco que llegue al vértice j:

$$\sum_i x_{i,j} = 1 \tag{6.10}$$

Y también a la restricción de que de cada vértice j solo puede salir un arco que llegue al vértice i:

$$\sum_j x_{i,j} = 1 \tag{6.11}$$

Pero estas restricciones no son suficientes, ya que no impiden la existencia de subcircuitos, de los que podemos ver un ejemplo en la Figura 6.5. Para evitar la formación de estos podemos añadir una restricción que obligue a que haya al menos un arco saliente de cada subtour [11]:

$$\sum_{i \in S, j \notin S} x_{i,j} \geq 1, \forall S \subset N / S \neq \phi, S \neq N \tag{6.12}$$

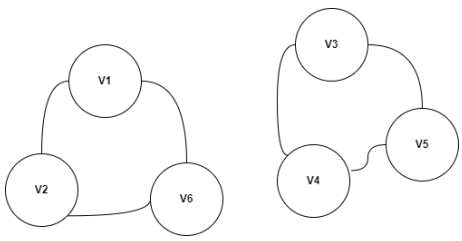


Figura 6.5: Formación de subcircuitos

Como es de suponer, este problema tiene aplicaciones en todo lo relacionado con:

- **Logística y distribución**, donde la buena planificación de rutas puede significar un ahorro de millones de euros.
- **Rutas de transporte público**, el TSP se utiliza para minimizar gasto, tanto de tiempo como de recursos en las rutas de metros, trenes o autobuses.
- **Circuitos electrónicos**, en el diseño de estos circuitos, así como en la disposición de los diferentes componente, el TSP se puede utilizar para mejorar su eficiencia.
- **ADN y proteínas**, también nos encontramos con ejemplos de este problema en el mundo la biología, donde problemas relacionados con la búsqueda de secuencias de aminoácidos, pueden ser resueltos gracias al TSP.

6.4.4. Problema de coloreado de grafos

El coloreado de grafos es un caso especial del problema denominado como etiquetado de grafos. Este consiste en colorear todos los vértices de un grafo, de manera que ningún par de vértices adyacentes tenga el mismo color. Esto tiene su origen en la coloración de mapas, concretamente cuando el matemático inglés Francis Guthrie postuló la conjetura de los cuatro colores al intentar colorear un mapa de Inglaterra, dándose cuenta que cuatro colores eran suficientes para colorear todas las regiones sin que dos adyacente compartiesen color. En este problema, saber si un grafo puede ser coloreado por 2 colores es un problema de P, ya que para ello solo necesitamos comprobar si tiene algún ciclo de longitud impar. El algoritmo para resolver este problema no es más que:

1. Dar un color 1 al nodo inicial.
2. Dar un color 2 a sus nodos adyacentes
3. Para cada nivel de nodo vecino, cambiar de color entre 1 y 2.
4. Si a un nodo se le asignan ambos colores, el grafo no se puede colorear con dos colores.

En cambio, comprobar si un grafo puede ser coloreado por 3 colores es un problema NP-completo.

Para saber cómo un grafo puede ser o no coloreado, podemos usar conceptos cómo el polinomio cromático, que cuenta las diferentes formas en que puede ser coleado un grafo dado un número de colores. El polinomio cromático consiste en una función $p(G, t)$ que varía en base al grafo que intentemos colorear, donde t es el número de colores que vamos a usar para el coloreado del grafo G. Algunos ejemplos de polinomios cromáticos son los siguientes:

- **Grafo completo de n vértices**: $P(G, t) = t(t-1)(t-2)...(t-(n-1))$
  En este ejemplo tomamos un grado en el que todos los vértices están conectados entre sí. Para un grafo de 4 vértices, si intentamos colorearlo con 3 colores tenemos que $P(G, 3) = 3(3-1)(3-2)(3-3) = 0$, por lo que no sería posible colorearlo. En cambio si en lugar de 3 intentamos utilizar 4 colores, obtenemos $P(G, 4) = 4(4-1)(4-2)(4-3) = 24$, lo que quiere decir que existen 24 formas diferentes de colorear este grafo con 4 colores. La Figura 6.6 muestra uno de los posibles coloreados.

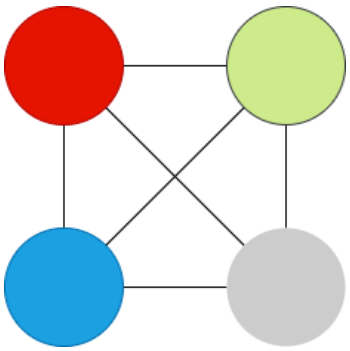


Figura 6.6: Coloreado de grafo completo

- **Árbol de n vértices**: $P(G,t) = t(t-1)^{n-1}$
  En este caso, vamos a tomar un árbol de 5 vértices. En primer lugar si intentamos colorearlo con un color, tenemos que $P(G,1) = 1(1-1)^{5-1} = 0$, por lo que claramente no es posible, pero si tan solo añadimos un color más, el resultado es de $P(G,2) = 2(2-1)^{5-1} = 2$, ya que en este caso solamente necesitamos 2 colores para colorear el grafo como podemos ver en la Figura 6.7.

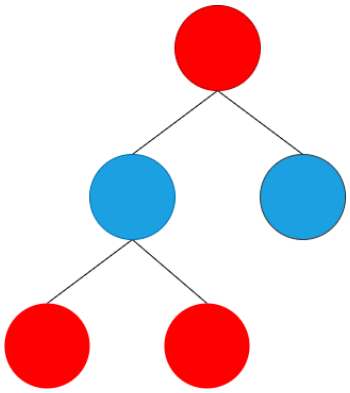


Figura 6.7: Coloreado de árbol

- **Ciclo**: $P(G,t) = (t-1)^n + (-1)^n(t-1)$

  En este último caso, la única diferencia que encontramos es que si un ciclo tiene un número par de nodos, 2 colores serán suficientes, ya que para un ciclo de 6 nodos tenemos que $P(G,2) = (2-1)^6 + (-1)^6(2-1) = 2$ como podemos ver en la Figura 6.8. En cambio si el ciclo tiene un número impar de nodos, no será posible colorearlo con únicamente 2 colores. Si cambiamos el ejemplo anterior por un grafo con 5 nodos vemos que $P(G,2) = (2-1)^5+(-1)^5(2-1) = 0$, por lo que este grafo necesita al menos 3 colores $P(G,3) = (3-1)^5+(-1)^5(3-1) = 33$, como nos muestra la Figura 6.9.

Como en el resto de los problemas que hemos visto, este no solo representa un problema abstracto, sino que tienen un montón de aplicaciones prácticas:

- **Distribución de asientos**: desde la planificación del posicionamiento de los invitados en el banquete de una boda, hasta la distribución de alimentos en una comida, son problemas que se pueden modelar y solucionar con la ayuda del problema de coloreado de grafos o derivados.

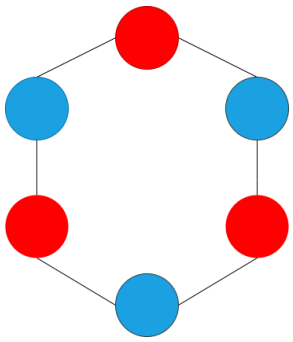


Figura 6.8: Coloreado de ciclo par

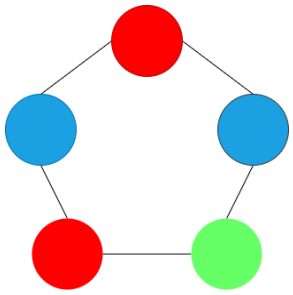


Figura 6.9: Coloreado de ciclo impar

- **Distribución de horarios**: el coloreado de grafos puede ser aplicado a la organización de horarios, tanto en centros educativos como universidades, institutos o colegios, como en cualquier tipo de ámbito que necesite una organización similar.

6.4.5. Problemas de decisión asociados

Aparte de ser problemas NP-duros, todos estos ejemplos que hemos visto tienen algo más en común, que todos ellos son problemas de optimización, ya que en todos se busca resolver un problema de la mejor manera posible. Pero lo interesante es que todos tienen un problema de decisión asociado, de manera que si fuésemos capaces de demostrar que este problema está en P, seguramente el problema de optimización tenga un algoritmo capaz de resolverlo en un tiempo polinomial. Veamos cuales serían los problemas de decisión equivalentes para cada uno de los vistos:

- **Cubrimiento de conjuntos**: ¿Existe algún cubrimiento que tenga X conjuntos o menos?
- **Mochila**: ¿Puedo conseguir un valor mayor que X en la mochila sin exceder el peso máximo?
- **Viajante de comercio**: ¿En este grafo hay algún camino con un coste menor que X?
- **Coloreado de grafos**: ¿Es posible colorear un determinado grafo con X colores?

De esta manera, tenemos una relación entre problemas de decisión y problemas de optimización y podemos estudiar estos últimos mediante técnicas desarrolladas para trabajar con problemas de decisión y con las clases P y NP en particular.

Capítulo 7

Algoritmos aproximados

Retomando la pregunta que nos hacíamos al final del apartado 4, a día de hoy no tenemos una forma de siempre evitar la fuerza bruta en la resolución de ciertos problemas. De hecho, aunque no sabemos si se puede demostrar, creemos que ni siquiera existe una forma de evitarla. Es por eso que la mejor opción que tenemos a día de hoy son los algoritmos de aproximación.

7.1. Definición

Un algoritmo de aproximación es aquel que consigue una solución no necesariamente óptima en un tiempo polinomial para problemas de la clase de complejidad NP. Estos algoritmos de aproximación nos proporcionan, en muchos casos, soluciones suficientemente buenas para ser utilizados de forma práctica, aunque no se pueda garantizar que nos estén dando la solución óptima.

7.2. Ejemplos

A continuación vamos a ver algunos ejemplos de algoritmos aproximados.

7.2.1. Viajante de comercio

Vamos a tomar el diagrama de la Figura 7.1 como base para ver como se comporta un algoritmo aproximado para el problema del viajante de comercio.

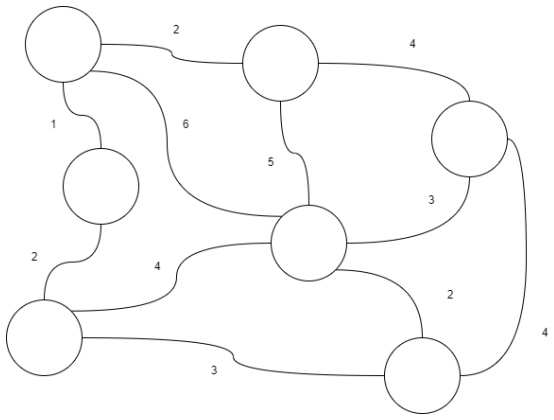


Figura 7.1: Ejemplo de problema del viajante de comercio

Un posible algoritmo aproximado sería el siguiente, conocido como el algoritmo del vecino más cercano:

1. Tomar un nodo S al azar del grafo
2. A partir de S, seleccionar el nodo más cercano y movernos a él
3. Continuar seleccionando el nodo mas cercano no visitado, hasta que todos estén seleccionados
4. Una vez que hayas visitado todos los nodos, vuelve al nodo inicial para completar el ciclo

Con este algoritmo el camino que nos quedaría sería el de la Figura 7.2, donde se ha marcado en los nodos el orden en que se visita cada uno de ellos. Pese a que no se puede garantizar que la solución obtenida con este algoritmo sea la óptima, podemos decir que será una solución práctica, fácil de implementar y rápida.

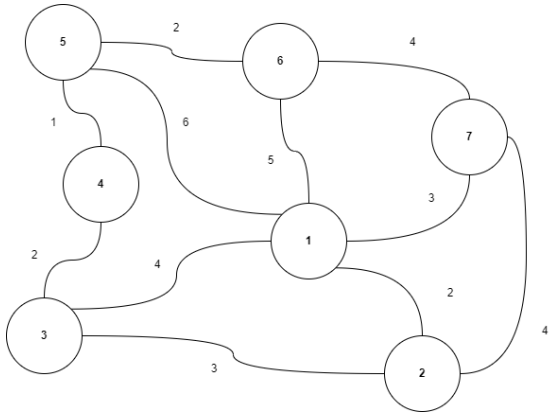


Figura 7.2: Viajante de comercio resuelto

7.2.2. K-SAT

Para el problema ya visto en el apartado 6.2.5 podemos utilizar el algoritmo de búsqueda aleatoria:

1. Seleccionar una asignación aleatoria para las n variables booleanas de la fórmula.
2. Evaluar las clausulas que se satisfacen con los valores actuales.
3. Durante un número X de iteraciones
   a) Realizar un cambio aleatorio en una variable.
   b) Si el número de clausulas que se satisfacen aumenta, guardar el cambio.
   c) Si no, borrar el cambio
4. Si no se encuentra una asignación de valores en X iteraciones, asumir que la fórmula no es satisfacible.

7.2.3. Max-SAT

Este problema consiste en encontrar una asignación de valores que maximice el número de cláusulas satisfacibles en una fórmula en CNF ϕ. Si $M(\phi)$ es el número de cláusulas satisfacibles, el problema de decisión con el que nos encontramos es el de determinar si $M(\phi) \geq K$ para una K dada. Como este problema es NP-completo, ya que podemos hacer una reducción a SAT igualando K al número total de cláusulas. Vamos a buscar un algoritmo de aproximación que lo resuelva. Este consiste en:

1. Asignar falso a todas las variables.
2. Contar el número de clausulas verdaderas como m_0
3. Asignar verdadero a todas las variables.
4. Contar el número de clausulas verdaderas como m_1
5. Devolver la mejor de las 2 soluciones

Este algoritmo es uno de los denominados de 2-aproximación, ya que nos garantiza que la solución será como mucho 2 veces peor que la óptima. Esto se debe a que cualquier cláusula que no sea verdadera al asignar todos los valores a verdadero, lo será al asignarlos a falso. Para demostrarlo, digamos que ϕ tiene un total de a clausulas. Debido a esto $M(\phi) \leq a$. Por otro lado $m_0 + m_1 \geq a$, ya que puede existir alguna cláusula que siempre se cumpla. Al mismo tiempo, sabiendo esto tenemos que $\text{máx}_{m_0,m_1} \geq a/2$, ya que por lo menos una tiene que cubrir la mitad de las cláusulas. Juntando esto con la primera parte de la demostración, obtenemos que $a/2 \geq M(\phi)/2$ [3].

Este es un algoritmo determinista, pero también existe un algoritmo probabilístico para Max-SAT, que simplemente consiste en seleccionar al azar el valor de cada variable. Se puede demostrar que, en media, se satisfarán al menos $(2^k - 1)/2^k$ cláusulas del número total. Por ejemplo, para 3 clausulas se satisfacen de media 7/8 del total de cláusulas que se pueden satisfacer.

7.3. Garantías de comportamiento

La pregunta que surge ahora es la de qué tan lejos de la solución óptima están estos algoritmos aproximados. Para estudiarlo, existen las denominadas garantías de comportamiento, que no son más que una promesa o una manera de asegurar que algoritmo tendrá un rendimiento mínimo. En un algoritmo de aproximación P, el valor a de la solución obtenida por P no será mayor que un factor p, conocido como garantía de comportamiento relativo, multiplicado por la solución óptima s:

$$s \leq a \leq ps \tag{7.1}$$

Por otro lado, el radio de comportamiento absoluto P_A, no es más que una relación entre la solución obtenida y la optima, de un algoritmo de aproximación A. Siendo I una instancia del problema, $R_A(I)$ la garantía de comportamiento de A en I, y r el radio de aproximación, tenemos que:

$$P_A = inf\{r \geq 1 | R_A(I) \leq r, \forall I\} \tag{7.2}$$

De una manera visual, podemos apreciar en la Figura 7.3 si una solución es sufientemente buena, si esta está lo necesariamente cerca de la solución óptima.

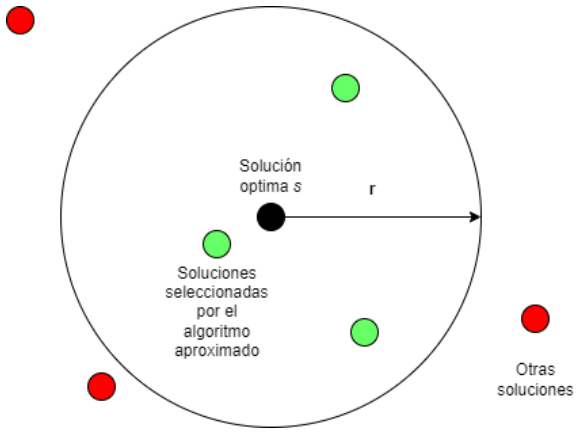


Figura 7.3: Garantía de comportamiento

Para el algoritmo determinista visto en el apartado 7.2.3, el radio de comportamiento sería de 2 para el algoritmo determinista, ya que nos asegura que la solución será como mucho 2 veces peor que la óptima.

Capítulo 8

Temas avanzados y problemas abiertos

La teoría de complejidad computacional no es algo estático que lleve años sin tener ningún cambio, y muchos menos es un campo del que ya se sepa todo. Por ello los avances y las incógnitas siguen estando a la orden del día. Además de problemas como P vs NP, la teoría de complejidad computacional tiene muchas de preguntas a las que hoy en día no se ha podido dar una respuesta. Algunos de los ejemplos de avances y problemas que aún no se han resuelto son:

8.1. Complejidad cuántica

La conexión entre el mundo de la física y el de la complejidad computacional siempre ha sido muy notoria. Por eso no es de extrañar que la mecánica cuántica comience a abrirse un hueco en este campo. Veamos algunos de las temas más punteros e interesantes dentro de la complejidad cuántica [15].

8.1.1. Computación cuántica en tiempo polinomial

De modo análogo a cómo hemos estudiado la clase P, que representa los problemas resolubles eficientemente con ordenadores clásicos, podríamos estudiar qué problemas se pueden resolver en tiempo polinomial con ordenadores cuánticos. Aquí se define la clase de complejidad BQP, que significa tiempo polinomial cuántico con error acotado (*Bounded-error Quantum Polynomial time*). Un problema de decisión pertenece a BQP si existe un algoritmo cuántico que lo resuelve en tiempo polinomial con una probabilidad de al menos 2/3. De una manera formal, un lenguaje L parte de BQP si solo existe una familia de circuitos cuánticos uniformemente generada en tiempo polinomial $\{Q_n : n \in \mathbb{N}\}$, tal que:

1. Para todo $n \in \mathbb{N}, Q_n$ toma n qubits como entrada y produce 1 bit como salida.

2. Para cada x en L, $Pr(Q_{|x|}(x) = 1) \geq 2/3$ (La probabilidad de que la respuesta obtenida sea correcta para una entrada en L es mayor o igual que 2/3.)

3. Para cada x que no esté en L, $Pr(Q_{|x|}(x) = 0) \geq 2/3$ (La probabilidad de que la respuesta obtenida sea correcta para una entrada que no está en L es mayor o igual a 2/3.)

8.1.2. Verificaciones de demostraciones cuánticas

Otra de las clases de complejidad cuántica es QMA (*quantum Merlin-Arthur*), la que es el análogo cuántico de NP. Esta clase se basa en la idea de *prueba cuántica*, un estado cuántico que hace las veces del certificado que conocemos de la definición de NP. La definición de QMA es similar a NP, incluyendo los estados cuánticos:

Sea $A = (A_{si}, A_{no})$ un problema y p un polinomio . En ese caso $A \in QMA_p$ si y solo si existe una familia de circuitos generada en tiempo polinomial $Q = \{Q_n : n \in \mathbb{N}\}$, donde cada circuito Q_n toma n qubits como input y produce una salida de un qubit, con las siguientes propiedades:

1. Para todo $x \in A_{si}$ existe un estado cuántico r de longitud $p(|x|)$ que $Pr[Q_{acepta}(x, r) \geq 2/3]$.
2. Para todo $x \in A_{no}$ existe un estado cuántico r de longitud $p(|x|)$ que $Pr[Q_{acepta}(x, r) \leq 2/3]$

8.2. Conjetura de la jerarquía

Para entender esta conjetura vamos a empezar hablando de *El Gran Colapso*, llamado así porque en caso de $P = NP$, muchas otras clases de complejidad serían igual a P. Una de ellas es $coNP$, que es la clase de complejidad de los problemas cuyos complementarios tienen una solución que puede ser verificada en un tiempo polinomial. Por encima de estas existe una infinidad de clases de complejidad de las que P y NP son solo el comienzo.

Por un lado tenemos que los problemas en NP tienen la forma $\exists w : B(x, w)$, donde w es el certificado y B está en P. Por otro lado, $coNP$ tiene la forma $\forall w : B(x, w)$. Esto se puede ver también como que NP es P con un $\exists$ delante, y $coNP$, P con un $\forall$. Con esto podemos definir clases de complejidad más elevadas en esta jerarquía añadiendo estos cuantificadores. Para ejemplificar esto podemos usar el *problema del circuito booleano más pequeño*, que dado un circuito C que calcula una función f_c, pregunta si C es el circuito más pequeño que resuelva f_c. Podemos ver este problema como

$$\forall C' < C : \exists x : f_{c'}(x) \neq f_c(x) \tag{8.1}$$

donde queremos ver si para todo circuito C' que sea menor que C, existe al menos una entrada x para la que $f_{c'}$ y f_c se comporten diferente. Como el *problema del circuito booleano más pequeño* puede ser expresado con 2 cuantificadores, un $\exists$ dentro de un $\forall$, decimos que está en la clase de complejidad $\prod_2 P$. Pero para entender de donde viene esto, tenemos que definir primero como se pueden representar todas las clases de complejidad sin importar el número de cuantificadores:

En primer lugar tenemos a $\sum_k P$, que es la clase de complejidad de los problemas de la forma

$$A(x) = \exists y_1 : \forall y_2 : \exists y_3 : ...Qy_k : B(x, y_1, ..., y_k), \tag{8.2}$$

donde $B \in P$, cada $|y_i|$ es polinomial en $|x|$, y $Q = \forall$ o $\exists$. De manera similar, $\prod_k P$ son aquellas con la forma

$$A(x) = \forall y_1 : \exists y_2 : \forall y_3 : ...Qy_k : B(x, y_1, ..., y_k), \tag{8.3}$$

Los problemas en estas clases se asemejan a los juegos de 2 jugadores que duran k movimientos. Por ejemplo en ajedrez, si el jugador de blancas tiene mate en 2, significa que existe un moviendo, que para cualquier respuesta de las negras, existe un movimiento con el que blanco da mate.

Todas estas clases de complejidad que hemos visto, forman la llamada *jerarquía polinomial*, PH. Podemos definir esta como:

$$PH = \bigcup_{k=0}^{\infty} \sum_{k} P = \bigcup_{k=0}^{\infty} \prod_{k} P \tag{8.4}$$

De la misma manera que creemos que $P \neq NP$, también creemos que las clases $\sum_k$ y $\prod_k$ son todas diferentes. Esto es lo que defiende la conjetura de la jerarquía, que todas estas clases son diferentes y por lo tanto esta jerarquía no colapsa, formando la clase PH. Juntando esta idea con la pregunta de $P = NP$, podemos sacar la conclusión que si esta igualdad fuese correcta, se produciría el ya mencionado *Gran Colapso*, en el que toda la jerarquía colapsaría siendo todas las clases iguales a P [9]. Podemos ver una representación grafica de esta idea en la Figura 8.1.

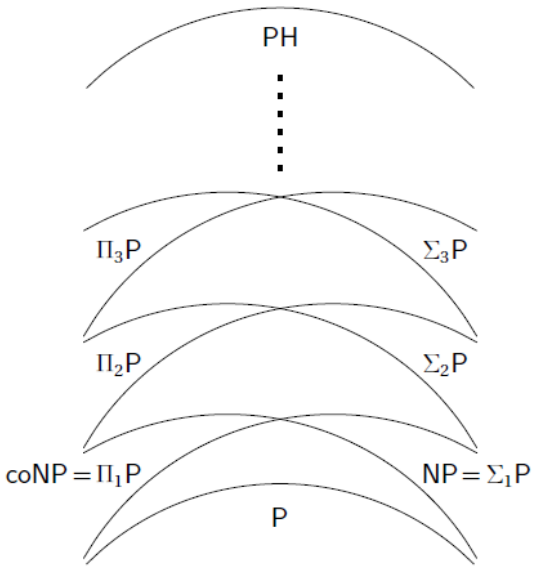


Figura 8.1: Conjetura de la jerarquía

Capítulo 9

Caso experimiental

Para ver de una forma más tangible y con datos reales la importancia del estudio de la complejidad computacional, particularmente de los algoritmos aproximados, vamos a hacer un pequeño estudio para ver las diferencias que hay entre estos algoritmos y los de fuerza bruta, así como la calidad de las soluciones que estos nos pueden aportar, y la relación de estas con el tiempo consumido para calcularlas.

9.1. Problema del viajante de comercio

Retomando el problema visto en el apartado 6.4.3, el problema del viajante de comercio consistía en un caso hipotético en el que un comerciante quiere visitar una serie de ciudades recorriendo la menor distancia posible. Como también vimos en el apartado 7.2.1 existe un algoritmo aproximado para realizar este cálculo, que como ya hemos visto consiste en tomar un nodo del grafo, seleccionar el nodo no visitado más cercano y movernos a él, repitiendo esto hasta visitar todos los nodos. Vamos a ver a continuación una comparativa entre este algoritmo y uno de fuerza bruta, en el que simplemente calcularemos el camino más corto de entre todos los posibles que visitan todos los nodos del grafo.

Para este problema utilizamos un grafo no dirigido representado por una matriz de dos dimensiones, en la que la celda ij representa la distancia entre los nodos i y j. Como podemos ver en el código del apéndice A.3, el grafo se genera de manera aleatoria, con una cantidad de nodos dada en el constructor. Utilizando como base una interfaz (mostrada en el apéndice A.5) y una clase abstracta del apéndice A.6 donde declaramos una lista con los nodos visitados y una variable con la distancia total, creamos tanto la clase de fuerza bruta del apéndice A.8 como la clase del algoritmo aproximado del apéndice A.7. Para la obtención de los datos ejecutamos la clase *Main* del apéndice A.9 donde se generarán grafos desde los 5 hasta los 13 nodos, cada uno de ellos 50 veces, ejecutando tanto el algoritmo de fuerza bruta como el aproximado, guardando en un archivo el tiempo del cálculo de cada uno de ellos, así como el camino calculado. Una vez se ha ejecutado todo obtenemos los resultados que podemos ver en la tabla 9.1

Algoritmo	Nodos	Tiempo(ms)	Desviación Típica	Ratio Aproximación	Desviación Típica Ratio Aproximación
Aproximado	5	0,008	0,002	1,022	0,034
	6	0,007	0,002	1,034	0,045
	7	0,005	0,002	1,039	0,046
	8	0,005	0,002	1,045	0,043
	9	0,005	0,002	1,052	0,040
	10	0,005	0,002	1,060	0,039
	11	0,007	0,002	1,075	0,047
	12	0,008	0,002	1,076	0,034
	13	0,010	0,004	1,077	0,039
Fuerza Bruta	5	0,022	0,007	1,000	
	6	0,057	0,043	1,000	
	7	0,166	0,014	1,000	
	8	0,949	0,124	1,000	
	9	13,429	28,110	1,000	
	10	78,498	4,211	1,000	
	11	1021,885	86,079	1,000	
	12	12150,292	1167,802	1,000	
	13	216474,871	74963,767	1,000	

Cuadro 9.1: Datos ejecución TSP

Como podemos ver tanto en la tabla 9.1 como en las Figuras 9.1 y 9.2, mientras que el tiempo medio de ejecución del algoritmo de fuerza bruta se dispara a partir de los 9 nodos, la media de los tiempos del algoritmo aproximado se mantiene estable sin importar el número de nodos. Por otro lado, podemos apreciar que el ratio de aproximación medio, que como hemos visto en apartados anteriores consiste en la relación entre la solución que nos da el algoritmo aproximado y la óptima, es decir, la del algoritmo de fuerza bruta, se mantiene prácticamente en 1, tan solo a unas centésimas, lo que indica que la solución media encontrada por este algoritmo es casi la óptima. Por ello podemos decir que en este caso, para grafos mayores de 12 o 13 nodos donde la diferencia de los tiempos de espera es muchísimo mayor, recurrir a un algoritmo aproximado podría ser una opción mucho mejor que a una de fuerza bruta.

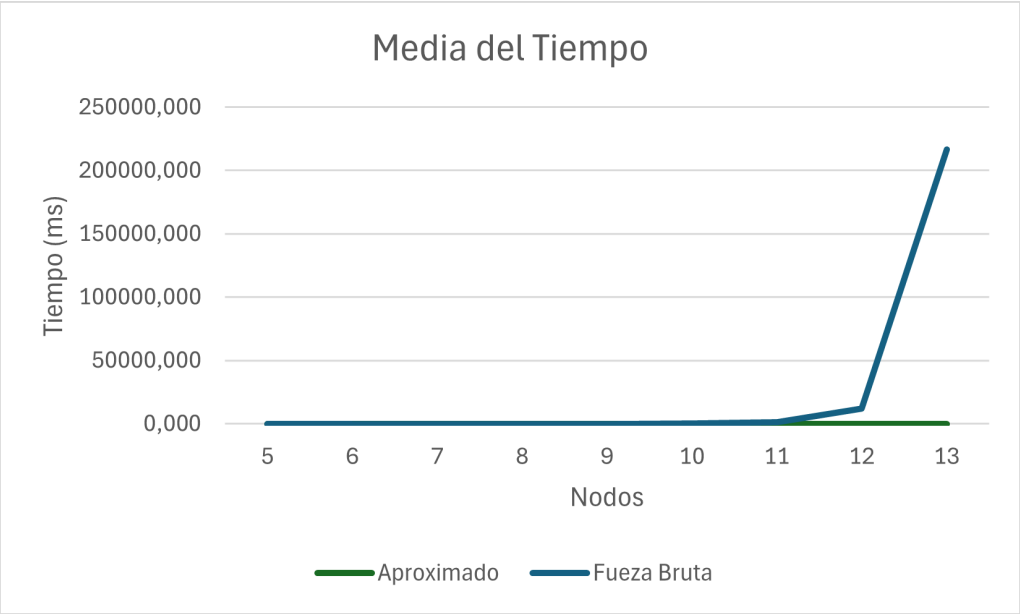


Figura 9.1: Tiempo medio TSP

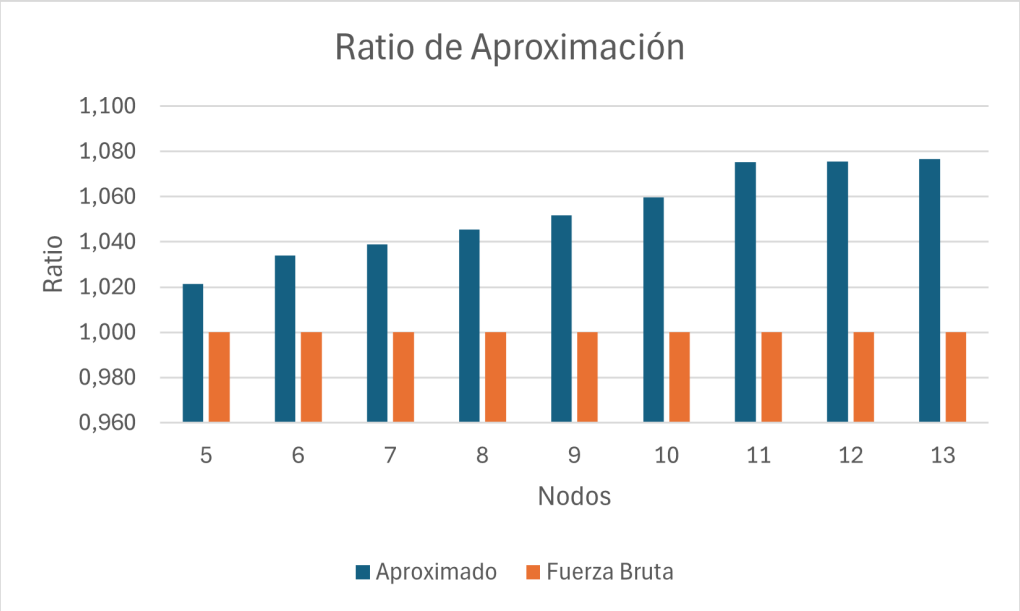


Figura 9.2: Ratio Aproximación TSP

9.2. K-SAT

Pasamos ahora a K-SAT, problema ya estudiado en el apartado 6.2.5, que consistía en una instancia del problema de la satisfabilidad booleana en CNF, donde se intenta encontrar una asignación de valores que satisfaga una fórmula booleana con un número K de variables en cada una de sus cláusulas. Siguiendo el algoritmo aproximado de la sección 7.2.2, vamos a hacer una comparativa entre este y el algoritmo de fuerza bruta.

En este caso utilizamos un grafo parecido al apartado anterior, pero ahora como se puede ver en el código A.4, el grafo está representado por una matriz en la que en el eje x aparece en número de clausulas de la fórmula, mientras que en el eje y se tiene el número de variables. El algoritmo aproximado de la sección A.7 contiene el método KSAT que aplica el algoritmo previamente explicado, mientras que el de fuerza bruta A.8 calcula todas las posibles combinaciones de variables. Para la ejecución y obtención de los datos la clase Main A.10 ejecuta fórmulas desde 5 hasta 16 variables, 100 veces cada una, donde para todas ellas tendremos un número K = 3 de variables en cada clausula, y un número *4.24* número de variables* de clausulas, ya que existen evidencias empíricas de que este es el número de clásusulas en las que el problema K-SAT resulta más difícil de resolver. Para menor número de cláusulas, las fórmulas tienden a ser siempre satisfacibles y para mayor número, siempre insatisfacibles [5].

Tanto en la tabla 9.2 como en las figura 9.3 y 9.4, vemos que en este caso el algoritmo aproximado no se acerca tanto a la solución óptima como si lo hacía el algoritmo del problema de viajante de comercio, ya que el porcentaje de acierto o relación entre el número de fórmulas satisfacibles y fórmulas que el algoritmo aproximado encuentra como satisfacibles, es demasiado bajo para todos los números de variables, especialmente para aquellos mayores a 7. Por ello, aunque para un número grande de variables el tiempo el tiempo de ejecución del algoritmo de fuerza bruta se dispara, no podríamos confiar en este algoritmo aproximado para determinar si una fórmula es satisfacible o no.

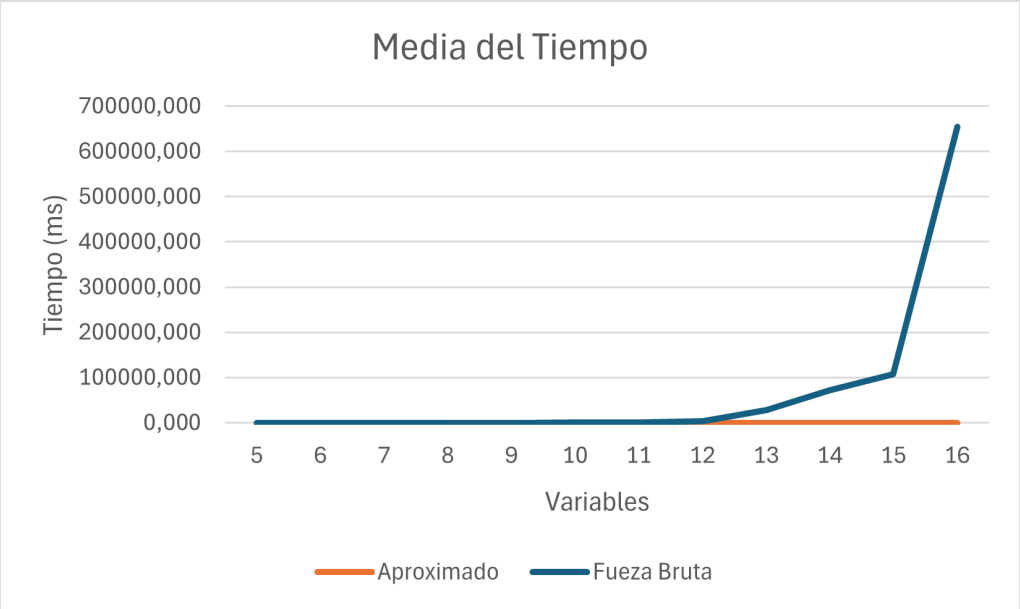


Figura 9.3: Tiempo de ejecución KSAT

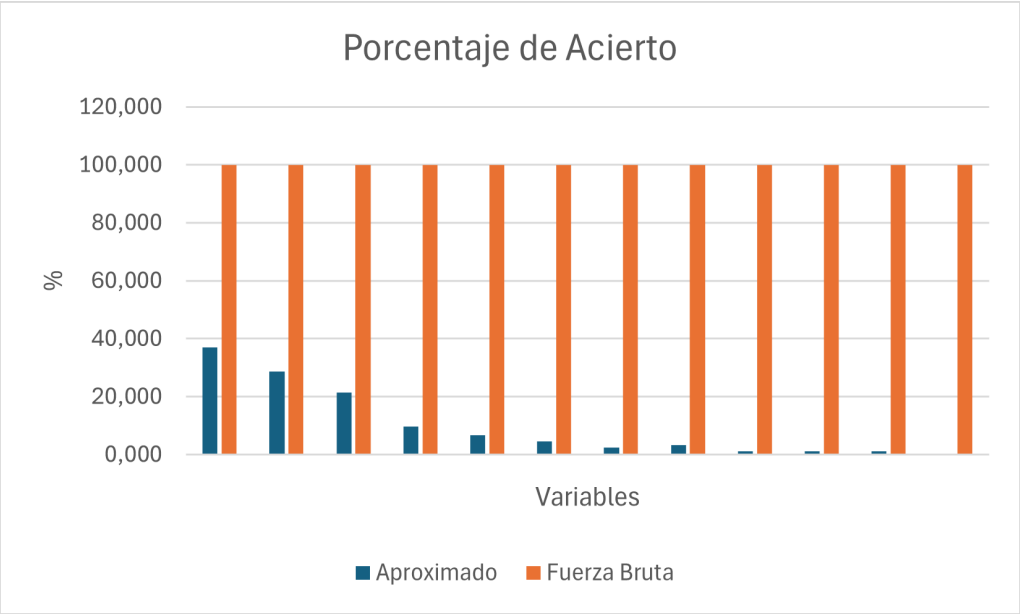


Figura 9.4: Porcentaje de acierto KSAT

Algoritmo	Variables	Tiempo(ms)	Desviación Típica	Porcentaje Acierto
Aproximado	5	0,041	0,032	37,079
	6	0,019	0,007	28,736
	7	0,022	0,006	21,348
	8	0,027	0,007	9,677
	9	0,029	0,007	6,667
	10	0,031	0,013	4,598
	11	0,035	0,012	2,353
	12	0,037	0,005	3,297
	13	0,045	0,011	1,149
	14	0,056	0,009	1,098
	15	0,058	0,008	1,190
	16	0,063	0,004	0
Fuerza Bruta	5	0,052	0,043	100,000
	6	0,053	0,041	100,000
	7	0,071	0,076	100,000
	8	0,145	1884,786	100,000
	9	0,766	4294,596	100,000
	10	44,074	11185,187	100,000
	11	435,441	23447,334	100,000
	12	3421,344	34598,983	100,000
	13	28080,862	467583,656	100,000
	14	71604,077	1175935,588	100,000
	15	107725,335	2414537,726	100,000
	16	653913,806	2397156,803	100,000

Cuadro 9.2: Datos ejecución KSAT

Capítulo 10

Conclusiones

Como hemos comprobado y estudiado a lo largo de este trabajo, la teoría de complejidad computacional estudia una de las preguntas más importantes de toda la humanidad: cómo resolver un problema de la forma más rápida y eficaz posible. Todavía no sabemos si esto es siempre posible, y seguramente haya muchos casos en los que no lo sea. El simple hecho de hacernos estas preguntas, investigar y acercarnos cada vez más a esta idea, es lo que nos ha permitido avanzar y encontrar soluciones a muchos de estos problemas. Y aunque estas no sean perfectas, el descubrimiento de soluciones más eficientes es lo que nos dice que estamos en el camino correcto.

La teoría de la complejidad es un marco teórico que nos permite clasificar y entender problemas y algoritmos, de manera que seamos capaces de comprender las limitaciones que hoy en día tiene nuestra tecnología y las dificultades inherentes en la resolución de ciertos problemas, pudiendo cambiar nuestros esfuerzos a nuevos enfoques y estrategias. Pero de igual manera, visto que el estudio de la complejidad computacional no es algo alejado de la realidad, sino que tiene innumerables aplicaciones a nuestras vidas, con el potencial de crear nuevas tecnologías y cambiar la forma que tenemos de interactuar con el mundo.

Apéndice A

Código del caso experimental

A.1. Interfaz Grafo

```
package graph;
public interface Graph {
    int [][] getGraph();
}
```

A.2. Clase Abstracta Grafo

```
package graph;

public abstract class AbstractGraph implements Graph{
    protected int [][] graph;

    public void printGraph(){
        System.out.println("GRAPH:");
        for(int i = 0; i < graph.length; i++){
            for(int j = 0; j < graph[0].length; j++){
                System.out.print(graph[i][j]);
            }
            System.out.println();
        }
    }

    public int[][] getGraph(){
        return this.graph;
    }
}
```

A.3. Grafo TSP

```
package graph;

import java.util.*;
```

```
public class GraphTSP extends AbstractGraph{
    private final int MIN_LINK_VALUE = 3;
    private final int MAX_LINK_VALUE = 6;

    public GraphTSP(int nodes){
        graph = new int[nodes][nodes];
        generateRandomGraph();
    }

    protected void generateRandomGraph(){
        for(int i = 0; i < graph.length; i++){
            for(int j = 0; j < graph[0].length; j++){
                if(graph[i][j] == 0){
                    int value = new Random().nextInt(
                        MIN_LINK_VALUE, MAX_LINK_VALUE);
                    graph[i][j] = value;
                    graph[j][i] = value;
                }
            }
        }
    }
}
```

A.4. Grafo KSAT

```
package graph;

import java.util.HashSet;
import java.util.Random;
import java.util.Set;

public class GraphKSAT extends AbstractGraph{

private int K;

public GraphKSAT(int clauses, int variables, int K){
    this.graph = new int[clauses][variables];
    this.K = K;
    generateRandomGraph();
}

private void generateRandomGraph(){
    for(int i = 0; i < graph.length; i++){
        Set<Integer> selectedIndices = new HashSet<>();
        while (selectedIndices.size() < K) {
            selectedIndices.add(
                new Random().nextInt(graph[0].length));
        }
```

```
            for (int j = 0; j < graph[0].length; j++) {
                graph[i][j] = selectedIndices.contains(j) ?
                new Random().nextInt(2) < 0.5 ? -1 : 1 : 0;
            }
        }
    }
}
```

A.5. Interfaz Algoritmo

```
package algorithms;

public interface Algorithm {

    void TSP();
    int getTotalDistance();
    void KSAT();
    boolean isSatisfied();
}
```

A.6. Clase Abstracta Algoritmo

```
package algorithms;
import graph.Graph;
import java.util.ArrayList;
import java.util.List;

public abstract class AbstractAlgorithm implements Algorithm{

protected int[][] graph;
protected List<Integer> visited = new ArrayList<>();

protected int totalDistance = 0;
protected boolean satisfied = false;

public AbstractAlgorithm(Graph g){
    this.graph = g.getGraph();
}

@Override
public int getTotalDistance(){
    return this.totalDistance;
}

@Override
public boolean isSatisfied() {
    return this.satisfied;
}
```

```
protected boolean[] generateClauses(int[][] values){
    boolean[] clauses = new boolean[values.length];
    for(int i = 0; i < graph.length; i++){
        for(int j = 0; j < graph[0].length; j++) {
            if (graph[i][j] != 0 && graph[i][j] == values[i][j]) {
                clauses[i] = true;
                break;
            }
        }
    }
    return clauses;
}

protected int countTrueClauses(boolean [] clauses){
    int count = 0;
    for (boolean b : clauses){
        if (b){
            count++;
        }
    }
    return count;
}
}
```

A.7. Algoritmo Aproximado

```
package algorithms;
import graph.Graph;
import java.util.ArrayList;
import java.util.List;
import java.util.Random;

public class ApproximationAlgorithm extends AbstractAlgorithm{

private final Random rand = new Random();

public ApproximationAlgorithm(Graph g){
    super(g);
}

@Override
public void TSP(){
    int startNode = 0;
    int currentNode = 0;
    int nextNode = 0;

    while(visited.size() != graph.length){
        int shortest = Integer.MAX_VALUE;
```

```
            for(int j = 0; j < graph[0].length; j++){
                if( currentNode != j && !visited.contains(j) &&
                graph[currentNode][j] < shortest && j != startNode ) {
                    shortest = graph[currentNode][j];
                    nextNode = j;
                } else if (j == startNode && visited.size() ==
                graph.length - 1) {
                    shortest = graph[currentNode][startNode];
                    nextNode = startNode;
                }
            }
            visited.add(nextNode);
            totalDistance += shortest != Integer.MAX_VALUE ?
            shortest : 0;
            currentNode = nextNode;
        }
    }

    @Override
    public void KSAT() {
        int repetitions = graph[0].length;
        int [][] values = assignRandomValues();

        //Evaluar la formula
        boolean [] clauses = generateClauses(values);
        boolean satisfied = evaluateFormula(clauses);
        int trueClauses = countTrueClauses(clauses);
        List<Integer> changedVariables = new ArrayList<>();

        while (repetitions-- != 0){
            if(satisfied || changedVariables.size() ==
            graph[0].length){ break; }

            //Cambio aletorio en una variable
            int variableToBeChanged =rand.nextInt(0, values[0].length);
            while (changedVariables.contains(variableToBeChanged)){
                variableToBeChanged = rand.nextInt(0, values[0].length);
            }
            changedVariables.add(variableToBeChanged);
            int [][] possibleNewValues = randomChange(
            values, variableToBeChanged);
            boolean [] possibleNewClauses = generateClauses(
            possibleNewValues);
            int numberNewTrueClauses = countTrueClauses(
            possibleNewClauses);

            if(numberNewTrueClauses > trueClauses){
                values = possibleNewValues;
                clauses = possibleNewClauses;
```

```
                trueClauses = numberNewTrueClauses;
                satisfied = evaluateFormula(clauses);
                repetitions++;
            }
        }

        this.satisfied = satisfied;
    }

    private boolean evaluateFormula(boolean[] clauses) {
        for(boolean b : clauses){
            if(!b){
                return false;
            }
        }
        return true;
    }

    private int [][] randomChange(int[][] values,
    int variableToBeChanged) {
        int[][] ret = values.clone();

        for(int i = 0; i < values.length; i++){
            int value = ret[i][variableToBeChanged];
            if(value != 0){
                ret[i][variableToBeChanged] =
                ret[i][variableToBeChanged] == 1 ? -1 : 1;
            }
        }
        return ret;
    }

    /**
     * Genera un valor aleatorio, -1 0 1, para cada variable.
     */
     private int [][] assignRandomValues(){
         int [][] values = new int[graph.length][graph[0].length];
         for(int j = 0; j < graph[0].length; j++){
             //Valor igual para cada variable
             int value = rand.nextInt(2) < 0.5 ? -1 : 1;
             for(int i = 0; i < graph.length; i++){
                 if(graph[i][j] != 0){
                     values[i][j] = value;
                 }
             }
         }
         return values;
     }
}
```

A.8. Fuerza Bruta

```
package algorithms;
import graph.Graph;

public class BruteForce extends AbstractAlgorithm{
    public BruteForce(Graph g){
        super(g);
    }

    //TSP

    @Override
    public void TSP() {
        int n = graph.length;
        int[] bestPath = null;
        int bestCost = Integer.MAX_VALUE;
        int[] nodes = new int[n];
        for (int i = 0; i < n; i++) {
            nodes[i] = i;
        }

        int[] indexes = new int[n];

        int i = 0;
        while (i < n) {
            if (indexes[i] < i) {
                swap(nodes, i % 2 == 0 ? 0 : indexes[i], i);

                int currentCost = calculateCost(nodes);
                if (currentCost < bestCost) {
                    bestCost = currentCost;
                    bestPath = nodes.clone();
                }

                indexes[i]++;
                i = 0;
            } else {
                indexes[i] = 0;
                i++;
            }
        }

        this.totalDistance = bestCost;
        assert bestPath != null;
        for (int j : bestPath) {
            this.visited.add(j);
        }
    }
```

```
private void swap(int[] array, int a, int b) {
    int temp = array[a];
    array[a] = array[b];
    array[b] = temp;
}

private int calculateCost(int[] path) {
    int totalCost = 0;
    for (int i = 0; i < path.length - 1; i++) {
        totalCost += graph[path[i]][path[i + 1]];
    }
    // Suma el costo de retorno al punto inicial
    totalCost += graph[path[path.length - 1]][path[0]];
    return totalCost;
}

//KSAT

@Override
public void KSAT() {
    int numVariables = graph[0].length;
    int numCombinations = 1 << numVariables; // 2^numVariables
    int bestTrueClauses = 0;

    // Try all possible combinations
    for (int i = 0; i < numCombinations; i++) {
        int[][] values = generateValuesFromCombination(
        i, numVariables);
        boolean[] clauses = generateClauses(values);
        int trueClauses = countTrueClauses(clauses);

        if (trueClauses > bestTrueClauses) {
            bestTrueClauses = trueClauses;
        }

        if (trueClauses == graph.length) {
            this.satisfied = true;
            return;
        }
    }
}

private int[][] generateValuesFromCombination(int combination,
int numVariables) {
    int[][] values = new int[graph.length][numVariables];
    for (int j = 0; j < numVariables; j++) {
        int value = (combination & (1 << j)) != 0 ? 1 : -1;
        for (int i = 0; i < graph.length; i++) {
            if (graph[i][j] != 0) {
                values[i][j] = value;
```

```
                }
            }
        }
        return values;
    }
}
```

A.9. Código Main TSP

```
import algorithms.ApproximationAlgorithm;
import algorithms.BruteForce;
import graph.GraphTSP;
import java.io.FileWriter;
import java.io.IOException;

public class MainTSP {
    public static void main(String[] args){

        String resultAP = "";
        String resultBF = "";

        for (int i = 5; i < 14; i++){ // Numero de nodos [5-13]
            resultAP += "NUMERO-DE-NODOS:-" + i + "\n";
            resultBF += "NUMERO-DE-NODOS:-" + i + "\n";
            for(int j = 0; j < 50; j++){
                GraphTSP graph = new GraphTSP(i);
                //Approximation

                double start = 0.000;
                double end = 0.000;

                ApproximationAlgorithm ap = new
                ApproximationAlgorithm(graph);
                start = System.nanoTime();
                ap.TSP();
                end = (System.nanoTime() - start)/1000000;
                resultAP += String.format("%s-%s-\n",
                ap.getTotalDistance(), end);

                //BruteForce

                BruteForce bf = new BruteForce(graph);
                start =  System.nanoTime();
                bf.TSP();
                end = (System.nanoTime() - start)/1000000;
                resultBF += String.format("%s-%s-\n",
                bf.getTotalDistance(), end);

                System.out.println("Numero-de-nodos:-" + i);
```

```
                System.out.println("Ejecucion: " + j);
            }
        }

        try {
            FileWriter fwap = new FileWriter("resultsAP.txt", true);
            fwap.write(resultAP);
            fwap.close();
            FileWriter fwbf = new FileWriter("resultsBF.txt", true);
            fwbf.write(resultBF);
            fwbf.close();
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
}
```

A.10. Código Main KSAT

```
import algorithms.ApproximationAlgorithm;
import algorithms.BruteForce;
import graph.GraphKSAT;
import java.io.FileWriter;
import java.io.IOException;

public class MainKSAT {
    public static void main(String[] args){
        String resultAP = "";
        String resultBF = "";

        for (int i = 5; i < 17; i++){
            resultAP += "NUMERO DE VARIABLES: " + i + "\n";
            resultBF += "NUMERO DE VARIABLES: " + i + "\n";
            for(int j = 0; j < 100; j++){
                int K = 3;
                GraphKSAT graphKSAT = new GraphKSAT(
                (int) 4.24*i, i, K);

                double start = 0.000;
                double end = 0.000;

                //Approximation
                ApproximationAlgorithm ap = new
                ApproximationAlgorithm(graphKSAT);
                start = System.nanoTime();
                ap.KSAT();
                System.out.println(ap.isSatisfied());
                end = (System.nanoTime() - start)/1000000;
                resultAP += String.format("%s %s \n",
```

```
                ap.isSatisfied(), end);

                //Brute Force

                BruteForce bf = new BruteForce(graphKSAT);
                start =  System.nanoTime();
                bf.KSAT();
                System.out.println(bf.isSatisfied());
                end = (System.nanoTime() - start)/1000000;
                resultBF += String.format("%s-%s-\n",
                bf.isSatisfied(), end);

                System.out.println("Numero de variables:" + i);
                System.out.println("Ejecucion:" + j);
            }
        }

        try {
            FileWriter fwap = new FileWriter(
            "resultsKSATAP.txt", true);
            fwap.write(resultAP);
            fwap.close();
            FileWriter fwbf = new FileWriter(
            "resultsKSATBF.txt", true);
            fwbf.write(resultBF);
            fwbf.close();
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
}
```

Apéndice B

Bibliografía

[1] Augusto Cortéz. Teoría de la complejidad computacional y teoría de la computabilidad. *RISI*, 1(1):102–105, 2004.

[2] M.D. Davis, E.J. Weyuker, and W. Rheinboldt. *Computability, Complexity, and Languages: Fundamentals of Theoretical Computer Science*. Elsevier Science, 2014.

[3] Computer Science Theory Explained. An approximation algorithms for maxsat. https://www.youtube.com/watch?v=v9b_s7_Cp4Q, 2021. Accedido en: 2024-04-26.

[4] A Fuentes-Penna, D Vélez-Díaz, S Moreno-Gutiérrez, M Martínez-Cervantes, O Sánchez-Muñoz, et al. Problema de la mochila (knapsack problem). *XIKUA Boletín Científico de la Escuela Superior de Tlahuelilpan*, 3(6), 2015.

[5] Ian P Gent and Toby Walsh. The SAT phase transition. In *ECAI*, volume 94, pages 105–109. PITMAN, 1994.

[6] F. C. Hennie and R. E. Stearns. Two-tape simulation of multitape Turing machines. *Journal of the ACM*, 13:533–546, OCT 1966.

[7] Robin K Hill. What an algorithm is. *Philosophy & Technology*, 29:35–59, 2016.

[8] Wang Min. Analysis on bubble sort algorithm optimization. In *2010 International forum on information technology and applications*, volume 1, pages 208–211. IEEE, 2010.

[9] Cristopher Moore and Stephan Mertens. *The Nature of Computation*. Oxford University Press, 2011.

[10] Jonatan Gómez Perdomo, Wilson Castro Rojas, and Alexander Cardona López. Programacion funcional y lambda cálculo. *Ingeniería e Investigación*, 1(40):72–82, 1998.

[11] Beatriz Pérez de Vargas Moreno et al. Resolución del problema del viajante de comercio (TSP) y su variante con ventanas de tiempo (TSPTW) usando métodos heurísticos de búsqueda local. *Universidad de Vallodolid*, 2015.

[12] Schöning. Schöning's algorithm. https://www.cs.yale.edu/homes/spielman/366/schoening.pdf, 2024. Accessed: 2024-03-25.

[13] Michael Sipser. *Introduction to the Theory of Computation*. Course Technology, 1997.

[14] Easy Theory. Cook-levin theorem: Full proof (SAT is NP-complete) - youtube. https://www.youtube.com/watch?v=LW_37i96htQ, 2024. Accessed: 2024-03-26.

[15] John Watrous. Quantum computational complexity. *arXiv preprint arXiv:0804.3401*, 2008.

[16] Wikipedia. NP-hard — wikipedia, la enciclopedia libre, 2024. [Internet; descargado 24-febrero-2024].

[17] Wikipedia contributors. Register machine — Wikipedia, the free encyclopedia, 2024. [Online; accessed 28-May-2024].