APROXIMABILIDAD EN PROBLEMAS NP-DUROS

MARITZA HERRERA FLOREZ
YUDY MARCELA BOLAÑOS RIVERA

UNIVERSIDAD DEL CAUCA
FACULTAD DE CIENCIAS NATURALES, EXACTAS Y DE LA EDUCACIÓN
DEPARTAMENTO DE MATEMÁTICAS
POPAYÁN
2003

APROXIMABILIDAD EN PROBLEMAS NP-DUROS

MARITZA HERRERA FLOREZ
YUDY MARCELA BOLAÑOS RIVERA

Trabajo de grado para optar al título de Matemático

Director
Mg. MAURICIO MACA CHAGÜENDO

UNIVERSIDAD DEL CAUCA
FACULTAD DE CIENCIAS NATURALES, EXACTAS Y DE LA EDUCACIÓN
DEPARTAMENTO DE MATEMÁTICAS
POPAYÁN
2003

Nota de aceptación

____

____

____

Director: Mauricio Maca Chagüendo.

Jurado: Diego Ramiro Correa Cuene.

Jurado: Fredy Amaya Robayo.

Popayán, 25 de abril de 2003.

A nuestros padres
por su constante apoyo.
A nuestros profesores
por su dedicación.

AGRADECIMIENTOS

Las autoras expresan sus agradecimientos a:

Mauricio Maca Chagüendo, Profesor de Matemáticas y Director de esta monografía, por su constante y valiosa colaboración y orientación.
Diego Ramiro Correa Cuene, Profesor de Matemáticas y Jurado de esta monografía por su interés y oportunas sugerencias.
Fredy Amaya Robayo, Profesor de Matemáticas y Jurado de esta monografía por su colaboración y oportunas sugerencias.
Gerardo Loaiza, Profesor de Matemáticas por su valiosa colaboración.
Luis Eduardo Montoya, Profesor de Matemáticas y Jefe del Departamento de Matemáticas por su motivación en la realización de este trabajo.
Alvaro Lopez Tascón, Profesor de Matemáticas y Decano de la Facultad de Educación por su motivación en la realización de este trabajo.
José Ignacio Tellez, Profesor de Matemáticas por su constante colaboración en el transcurso de nuestra carrera.
Nuestros padres, por su apoyo permanente y su constante motivación en el transcurso de todos nuestros estudios.

A la Universidad del Cauca y a todos los profesores del Departamento de Matemáticas y demás personas que de alguna manera colaboraron con la realización de esta monografía.

CONTENIDO

INTRODUCCION **20**

1. PRELIMINARES **24**

1.1. NOTACIÓN ASINTÓTICA 24

1.2. GRAFOS 30

1.3. PROBLEMAS Y ALGORITMOS 43

1.4. ALFABETOS Y LENGUAJES 49

1.5. ESQUEMAS DE CODIFICACIÓN 51

2. MODELOS DE COMPUTACIÓN BÁSICOS **53**

2.1. MÁQUINAS DE TURING DETERMINÍSTICAS 54

2.1.1. Máquina de Turing 1-cinta 54

2.1.2. Máquinas de Turing k-cintas 62

2.2. MÁQUINAS DE TURING NO DETERMINÍSTICAS 67

2.2.1. Modelo estándar 67

2.2.2. Modelo no estándar 69

2.3. MÁQUINA DE TURING UNIVERSAL 71

2.4. EL PROBLEMA DE LA PARADA 73

2.5. MÁQUINAS DE ACCESO ALEATORIO(RAM) 75

3. NP-COMPLETITUD **80**

3.1. LA CLASE P 81

3.2. LA CLASE NP 87

3.2.1. Máquinas de Turing no determinísticas y la clase NP 87

3.2.2. Algoritmos de verificación y la clase NP 89

3.3. RELACIÓN ENTRE P Y NP 93

3.4. TRANSFORMACIÓN POLINOMIAL Y LA CLASE NPC 96

3.5. TEOREMA DE COOK 101

3.6. ALGUNOS PROBLEMAS NP-COMPLETOS 108

3.6.1. El problema del ciclo hamiltoniano (HC) 109

3.6.2. El problema SUBSET-SUM 115

4. ALGORITMOS DE APROXIMACION PARA PROBLEMAS NP-DUROS 119

4.1. FUNDAMENTOS: PROBLEMAS DE OPTIMIZACIÓN Y LAS CLASES NPO y PO 120

4.2. PROBLEMAS DE DECISIÓN SUBYACENTES 123

4.3. CLASES DE APROXIMACIÓN 125

4.3.1. La clase APX 127

4.3.2. La clase PTAS 129

4.3.3. La clase FPTAS 134

4.4. ALGORITMOS DE APROXIMACIÓN PARA ALGUNOS PROBLEMAS NP-DUROS 134

4.4.1. El problema del AGENTE VIAJERO METRICO está en APX . 135

4.4.2. El problema de Cobertura con discos está en PTAS 139

4.4.3. Una mejor aproximación para el problema de Cobertura con discos 143

4.4.4. El problema MAX SUBSET-SUM está en FPTAS 149

5. CONCLUSIONES Y SUGERENCIAS PARA TRABAJOS FUTUROS 158

BIBLIOGRAFIA 160

LISTA DE TABLAS

2.1. Máquina de Turing que multiplica por 4. 57

2.2. Máquina de Turing que acepta las cadenas que terminan en 00 o 11 y máquina de Turing que suma dos naturales escritos en el código "palitos". 58

2.3. Máquina de Turing para palíndromos. 60

2.4. Máquina de Turing que produce computaciones infinitas. 60

2.5. Máquina de Turing 2-cintas para palíndromos. 64

2.6. Máquina de Turing 2-cintas que acepta las cadenas de la forma 0^n1^n. . 66

2.7. Instrucciones RAM y sus significados. 76

2.8. Una computación de una máquina RAM para multiplicar dos enteros. . 78

3.1. Codificación de un grafo bajo tres esquemas de codificación diferentes. . 84

3.2. Variables usadas en la demostración del Teorema de Cook. 104

3.3. Grupos de cláusulas usados en la demostración del Teorema de Cook. . 105

3.4. Los seis grupos de cláusulas usados en la demostración del Teorema de Cook. 106

3.5. Número de variables usadas en la demostración del Teorema de Cook. . 107

3.6. Número de cláusulas usadas en la demostración del Teorema de Cook. . 108

4.1. Algoritmos de aproximación para el problema de Cobertura con discos. 144

LISTA DE FIGURAS

1.1. Significado de la notación O grande. 25

1.2. Interpretación gráfica de $n^2 + 2n + 1 = \mathrm{O}(n^2)$ 26

1.3. Significado de la notación Omega. 29

1.4. Significado de la notación Theta. 30

1.5. Representación pictórica de un grafo. 31

1.6. Representación pictórica de un grafo dirigido y un grafo no dirigido. . . . 33

1.7. Grafo subyacente 33

1.8. Digrafo simple y su grafo subyacente 33

1.9. Un grafo y el grado de sus vértices. 34

1.10. Ejemplos de grafos completos. 35

1.11. Dos grafos bipartitos. 35

1.12. El grafo simple mas pequeño no bipartito. 35

1.13. Grafos con peso. 36

1.14. Representación de un subgrafo. 36

1.15. Recorrido entre dos vértices. 37

1.16. Un recorrido dirigido. 38

1.17. Grafos conexos. 38

1.18. Un grafo no conexo con tres subgrafos conexos. 39

1.19. Conectividad en digrafos. 39

1.20. Camino en un grafo 40

1.21. Digrafo con tres ciclos. 41

1.22. Grafos acíclicos. 41

1.23. Un grafo y uno de sus ciclos hamiltonianos. 42

1.24. Un digrafo y su matriz de adyacencia. 42

1.25. Un grafo y su matriz de adyacencia. 42

1.26. Una instancia del problema del Circuito Hamiltoniano. 45

2.1. Una computación de una máquina de Turing que multiplica por 4. . . . 57

2.2. Computación de una máquina de Turing que calcula la suma de dos números naturales codificados con el código "palitos". 59

2.3. Configuraciones de una máquina de Turing que multiplica por 4. 61

2.4. Computación de una máquina de Turing 2-cintas para palíndromos. . . 65

2.5. Los primeros 3 niveles de un árbol de computación. 68

2.6. Máquina RAM para multiplicar dos enteros (MPLY) 78

2.7. Máquina RAM que calcula el producto punto entre dos vectores. 79

3.1. Camino de computación de aceptación más corto para una MTN. . . . 88

3.2. Una posible visión del mundo de NP. 95

3.3. Ilustración gráfica de una transformación polinomial. 97

3.4. Una visión mas detallada del mundo de NP. 100

3.5. Subgrafo A usado en la transformación de 3SAT a HC. 110

3.6. Subgrafo B usado en la transformación de 3SAT a HC. 111

3.7. Transformación de una instancia de 3SAT a una instancia de HC. . . . 112

3.8. Transformación de una instancia de EXACT COVER BY 3-SETS en una instancia de SUBSET-SUM. 116

4.1. Una aplicación del procedimiento APROX-AVM. 138

4.2. Particiones generadas por una aplicación de la estrategia $shifting$. . . . 140

4.3. Una rejilla y algunos discos *grid*. 145

4.4. Una partición del plano en cuadrados de lado lD. 147

4.5. Cobertura compacta de un cuadrado lD. 147

4.6. Funcionamiento del procedimiento EXACT-SUBSET-SUM. 150

4.7. Una computación del algoritmo de aproximación para MAX SUBSET-SUM. 154

RESUMEN

Esta monografía presenta los fundamentos de la Teoría de la Aproximabilidad en problemas NP-duros. Primero, se presentan algunos modelos de computación como las máquinas de Turing y las máquinas RAM, que son luego usados para definir las clases de complejidad P, NP y NPC, con lo cual se da paso al estudio de la aproximabilidad en problemas NP-duros presentando las clases de aproximación APX, PTAS y FPTAS y describiendo dos estrategias muy aplicables en el desarrollo o mejoramiento de algoritmos de aproximación, las estrategias *shifting* y *grid*. Además, la presentación de todos los conceptos tratados en esta monografía se ha complementado con ejemplos, tablas y gráficos pertinentes para una mayor facilidad en su comprensión.

INTRODUCCION

La Teoría de la Computabilidad desarrollada principalmente por Alan Turing, Alonso Church y Kurt Gödel a comienzos de la década de los 30's fué la que dió origen a la Teoría de la Complejidad Computacional . Dicha teoría se preocupaba por conocer si toda función era computable[*] lo cual llevó a Turing a desarrollar una máquina llamada máquina de Turing, capaz de computar funciones. Turing no se preocupó por la eficiencia de sus máquinas, esto es, por el tiempo usado en sus computaciones, sólo se interesó en que sus máquinas fueran capaces de simular algoritmos arbitrarios; así la máquina de Turing se convirtió en uno de los principales modelos de computación. La noción de computación tiempo polinomial fué introducida en los 60's por Cobham y Edmonds en los inicios de la Teoría de la Complejidad Computacional; Edmonds llamó a los algoritmos tiempo polinomial, esto es, a aquellos algoritmos cuyo número de pasos en su ejecución es acotado por un polinomio, "buenos algoritmos", así la Teoría de la Complejidad Computacional se preocupa por la eficiencia de los algoritmos.

En la práctica, existen muchos problemas en matemáticas y otras áreas para los cuales se han desarrollado procedimientos que encuentran su solución en forma eficiente (algoritmos tiempo polinomial), a estos problemas se les llama usualmente "fáciles de resolver"; sin embargo, para una gran cantidad de problemas hasta el momento no ha sido posible el desarrollo de tales procedimientos, por lo cual se les llama usualmente

[*] Una función es computable si existe un procedimiento que siguiendo un conjunto finito de pasos encuentra el valor de la función para cualquier entrada.

“difíciles de resolver”. En la década de los 70’s Karp introdujo la notación estándar de P y NP para las clases de problemas “fáciles de resolver” (clase P) y “difíciles de resolver” (clase NP). En 1971 Stephen Cook introdujo la noción de NP-completitud, más precisamente la noción de problema NP-completo; estos problemas se caracterizan por ser los “más difíciles” en NP y tienen la propiedad fundamental que si se resuelve uno de ellos eficientemente entonces todos los problemas en NP también se pueden resolver en forma rápida. En el mismo año Cook presentó el primer problema NP-completo, el cual es un problema de lógica booleana, llamado el problema de SATISFABILIDAD, a raíz de este resultado han sido identificados cientos de problemas NP-completos. En este momento, una pregunta de fundamental importancia que surge es ¿P = NP?, esto es, ¿todos los problemas en NP admiten un algoritmo tiempo polinomial que los resuelva?, esta pregunta continúa abierta y es de observar que la comunidad matemática la considera como uno de los problemas abiertos más relevantes de este milenio; así Steven Smale, ganador de la medalla Fields, lo sitúa como el problema número 3 en la lista de Smale de los 18 problemas desafiantes para el siglo XXI, además también aparece en la lista de los siete problemas del milenio publicada por el Instituto Clay de Matemáticas.

Un tipo importante de problemas que son de interés en la actualidad son lo problemas de optimización, en los cuales se busca maximizar o minimizar una cierta medida. Este tipo de problemas ha sido ligado a la Teoría de NP-completitud al asociarles de manera natural un problema de decisión, es decir, un problema en el que la respuesta es SI o NO; infortunadamente muchos problemas de optimización tienen asociado un problema de decisión que es NP-completo, dichos problemas pertenecen a una clase especial llamada la clase de problemas NP-duros y se ha demostrado que en este caso es improbable que el problema de optimización inicial pueda ser resuelto por un algoritmo tiempo polinomial. Por esta razón es natural enfocar los esfuerzos en buscar otras alternativas para resolverlo, una de ellas es desarrollar algoritmos que retornen soluciones aproximadas al problema, llamados algoritmos de aproximación, y el estudio de ellos ha dado origen a la Teoría de la Aproximabilidad.

La Teoría de la aproximabilidad en problemas NP-duros estudia algoritmos de aproximación que permiten obtener una solución que puede no ser óptima, pero que no se aleja de la solución ideal más allá de un cierto factor conocido. Así, los algoritmos de aproximación se han desarrollado en respuesta a la imposibilidad de resolver una gran variedad de problemas de optimización importantes. Al analizar estos algoritmos surgen algunas preguntas tales como: ¿qué tanto pueden ser mejorados en términos de un factor conocido?, ¿cómo lograr una aproximación mejor?, ¿todos los problemas pueden ser aproximados?, estas preguntas dan una visión general de la "dureza"de desarrollar aproximaciones. Además, para muchos problemas, estas preguntas continuan abiertas y encontrar su respuesta sería equivalente a contestar la pregunta ¿P = NP?.

Gran cantidad de problemas que se intentan aproximar han surgido como resultado de situaciones que se presentan en la práctica en diferentes campos; el siguiente ejemplo ilustra uno de tales problemas, el cual da una visión de la importancia de los algoritmos de aproximación: "Imagínese que usted se encuentra a las 9 a.m. en su trabajo, frente a 8 máquinas que estarán listas para trabajar a las 10 a.m. con 147 tareas de varias duraciones para ser procesadas en estas máquinas. Por otro lado, a las 9 p.m. hay un partido de basketball en la televisión que no se quiere perder por nada del mundo, pero tiene que quedarse hasta que todas las tareas estén terminadas. Así que usted desearía asignar tareas a las máquinas de tal forma que pueda llegar a su casa lo mas pronto posible". Este problema en apariencia sencillo es sin embargo una instancia del conocido Problema *Minimum Makespan* el cual es NP-duro; esto significa no solamente que no se conoce un algoritmo tiempo polinomial para resolverlo, sino que es improblable que uno exista. A raíz de esto, se han desarrollado algoritmos de aproximación para este problema, intentando disminuir cada vez más el error en la aproximación y el tiempo de ejecución.

Como ya se ha dicho, el desarrollo de las Teorías de Complejidad y Aproximabilidad son muy recientes, de ahí que la literatura disponible sea escasa y frecuentemente de un nivel muy avanzado. La presente monografía pretende subsanar de alguna manera esta dificultad, proporcionando un texto que recopile en forma amplia y detallada los

fundamentos de la Teoría de la Complejidad Computacional y de la Aproximabilidad en problemas NP-duros. Por otro lado, se espera que este texto sirva de material de consulta y apoyo para muchos estudiantes que desean iniciarse en este interesante campo de la matemática y sea una motivación para continuar su estudio y profundizar en todo lo que en él queda por hacer. De manera particular se pretende que el presente trabajo contribuya al fortalecimiento académico del Grupo de estudio y desarrollo investigativo en Matemática aplicada del departamento de Matemáticas de la Universidad del Cauca, que trabaja en la línea de Matemática Computacional.

1. PRELIMINARES

Este capítulo introductorio contiene algunos conceptos básicos que se requieren en el estudio de la teoría de NP-completitud y de aproximabilidad en problemas NP-duros que se desarrolla en el resto de esta monografía. Primero, se presentan las nociones básicas sobre notación asintótica y teoría de grafos proporcionando varios ejemplos que aclaran el uso que se dará a estos conceptos en el resto del texto. A continuación se describen los conceptos de problema, algoritmo, alfabeto, lenguaje y se introduce la noción de esquema de codificación para un problema.

1.1. NOTACIÓN ASINTÓTICA

En el contexto de la complejidad computacional son de gran interés las *funciones de complejidad*, las cuales expresan el tiempo requerido por un algoritmo para ejecutarse sobre entradas de determinado tamaño en el peor caso; usualmente se denotan por $\mathrm{T}(n)$ donde n es el tamaño de la entrada, estas funciones pueden ser del tipo $\mathrm{T}(n) = 3n^2 + 5n - 2$, $\mathrm{T}(n) = 4\ln n + 7$, $\mathrm{T}(n) = 10\, e^n - 1$ entre otras, pero en ellas, generalmente, no es necesario tener en cuenta sus constantes multiplicativas o aditivas. En este caso, para representar tales funciones se utiliza la notación asintótica.

Se presentarán a continuación, las tres notaciones asintóticas más usuales en complejidad.

En esta sección, se usará el símbolo $\mathbb{R}^*$ para denotar el conjunto de números reales positivos unido con el cero.

Definición 1.1 (Notación O). Sea $g(n)$ una función de $\mathbb{N}$ en $\mathbb{R}^*$.

$$\begin{aligned} \mathrm{O}(g(n)) = \quad & \{f : \mathbb{N} \to \mathbb{R}^* : \text{existen constantes positivas c y } n_0 \text{ tales que} \\ & 0 \leq f(n) \leq \mathrm{c}g(n), \text{ para todo } n \geq n_0\}. \end{aligned}$$

Usualmente, si $f(n) \in \mathrm{O}(g(n))$, se escribe $f(n) = \mathrm{O}(g(n))$ y se lee "$f(n)$ es o grande de $g(n)$"o "$f(n)$ es del orden de $g(n)$".

En otras palabras, $\mathrm{O}(g(n))$ denota el conjunto de funciones asintóticamente acotadas superiormente por $g(n)$. La Figura 1.1 muestra el significado de la notación O.

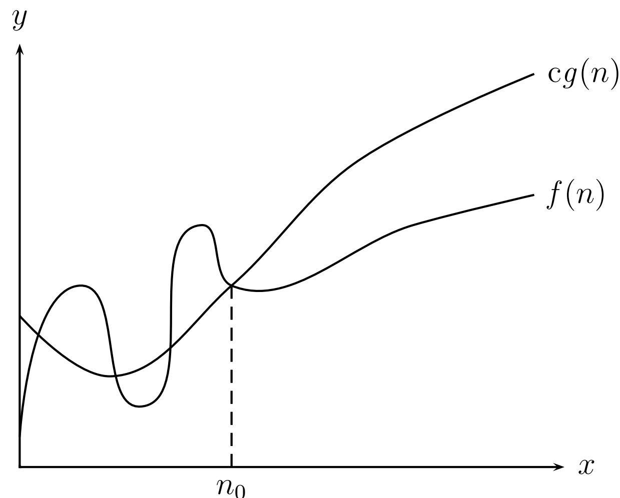


Figura 1.1: $f(n) = \mathrm{O}(g(n))$

Ejemplo 1.1

1. $n^2 + 2n + 1 \in O(n^2)$. En efecto:

$$\begin{aligned} n &\leq n^2 && \text{para todo } n \geq 1 \\ 2n &\leq 2n^2 && \text{para todo } n \geq 1 \\ n^2 + 2n &\leq 2n^2 + n^2 && \text{para todo } n \geq 1 \\ n^2 + 2n + 1 &\leq 2n^2 + n^2 + n^2 && \text{para todo } n \geq 1 \\ n^2 + 2n + 1 &\leq 4n^2 && \text{para todo } n \geq 1. \end{aligned}$$

Haciendo c = 4 y $n_0 = 1$, se obtiene que $n^2 + 2n + 1 \in O(n^2)$, como se ilustra en la Figura 1.2.

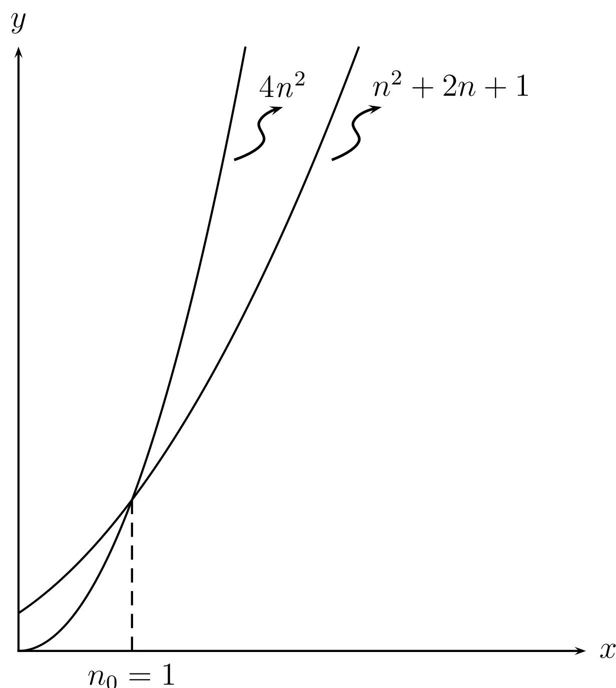


Figura 1.2: $n^2 + 2n + 1 = O(n^2)$.

En general, si $f : \mathbb{N} \rightarrow \mathbb{R}^*$ es un polinomio de grado d, es decir, $f(n) = a_d n^d + a_{d-1} n^{d-1} + \ldots + a_0$, entonces $f(n) \in O(n^d)$.

2. $3^n \notin O(2^n)$. En efecto:

Si $3^n \in O(2^n)$ deben existir constantes positivas c y n_0 tales que $3^n \leq c2^n$ para

todo $n \geq n_0$. Esto es, $(\frac{3}{2})^n \leq$ c, lo cual es imposible pues $(\frac{3}{2})^n$ es una función exponencial.

3. $4n + 100 = O(n)$.

4. $n! = O[(n+1)!]$.

5. $2^n = \mathrm{O}(3^n)$. $\diamondsuit$

Frecuentemente las propiedades o teoremas son herramientas útiles que permiten probar resultados que sólo con el uso de la definición sería más complicado. A continuación, se presentan dos reglas que cumple la notación o grande.

Teorema 1.1 (Regla del Máximo). [BB97] Sean f y g funciones de $\mathbb{N}$ en $\mathbb{R}^*$. Se cumple que:

$$O(f(n) + g(n)) = O(\text{máx}\{f(n), g(n)\}).$$

Ejemplo 1.2

$n^3 + n = \mathrm{O}(n^3)$. En efecto:

Por el Teorema 1.1, $\mathrm{O}(n^3 + n) = \mathrm{O}(\text{máx } \{n^3, n\}) = \mathrm{O}(n^3)$,

luego, $\mathrm{O}(n^3 + n) = \mathrm{O}(n^3)$

en particular, $n^3 + n \in \mathrm{O}(n^3 + n)$

por lo tanto, $n^3 + n \in \mathrm{O}(n^3)$. $\diamondsuit$

Teorema 1.2 (Regla del Límite). [BB97] Sean f y g funciones de $\mathbb{N}$ en $\mathbb{R}$. Se cumplen las siguientes proposiciones:

1. Si $\lim_{n\to\infty} \dfrac{f(n)}{g(n)} = k$ *y* $k \in \mathbb{R}^+$ entonces $f(n) \in O(g(n))$ *y* $g(n) \in O(f(n))$.

2. Si $\lim_{n\to\infty} \dfrac{f(n)}{g(n)} = 0$ entonces $f(n) \in O(g(n))$ *y* $g(n) \notin O(f(n))$.

3. Si $\lim_{n\to\infty} \dfrac{f(n)}{g(n)} = \infty$ entonces $g(n) \in O(f(n))$ *y* $f(n) \notin O(g(n))$.

Ejemplo 1.3

Puesto que $\lim_{n\to\infty} \frac{n!}{(n+1)!} = 0$, por el teorema 1.2 se obtiene que $n! \in O((n+1)!)$ y $(n+1)! \notin O(n!)$.

El límite $\lim_{n\to\infty} \frac{n!}{2^n} = \infty$, entonces por el teorema 1.2 se obtiene que $2^n \in O(n!)$ y $n! \notin O(2^n)$. $\diamondsuit$

Definición 1.2 (Notación Ω). Sea *g(n)* una función de $\mathbb{N}$ en $\mathbb{R}^*$.

$$\begin{aligned}\Omega(\mathrm{g(n)}) = \quad & \{f : \mathbb{N} \to \mathbb{R}^* : \text{existen constantes positivas c y } n_0 \text{ tales que} \\ & 0 \leq \mathrm{c}g(n) \leq f(n), \text{ para todo } n \geq n_0\}.\end{aligned}$$

Usualmente, si $f(n) \in \Omega(g(n))$, se escribe $f(n) = \Omega(g(n))$ y se lee "$f(n)$ es omega de $g(n)$".

En otras palabras, $\Omega(g(n))$ denota el conjunto de funciones asintóticamente acotadas inferiormente por $g(n)$. La Figura 1.3 muestra el significado de la notación Ω.

Teorema 1.3 (Regla de Dualidad). [BB97]

$$f(n) \in O(g(n)) \Longleftrightarrow g(n) \in \Omega(f(n)).$$

Ejemplo 1.4

1. $n^n = \Omega(n!)$. En efecto:

   Como $\lim_{n\to\infty} \frac{n!}{n^n} = 0$, esto implica, por el Teorema 1.2 que $n! \in O(n^n)$. Y por el Teorema 1.3, se obtiene que $n^n = \Omega(n!)$.

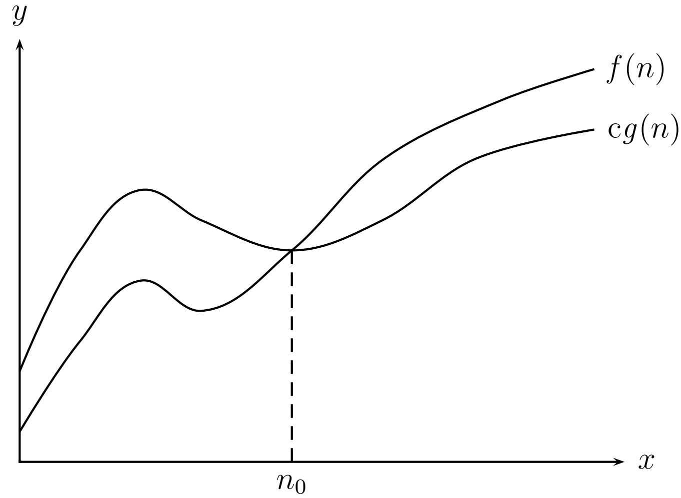


Figura 1.3: $f(n) = \Omega(g(n))$

2. $\sqrt{n} = \Omega(\ln n)$. En efecto:

   Como $\lim\limits_{n\to\infty} \dfrac{\sqrt{n}}{\ln n} = \infty$, esto implica, por el Teorema 1.2, que $\ln n = O(\sqrt{n})$ y por el Teorema 1.3, se obtiene que $\sqrt{n} = \Omega(\ln n)$.

3. Del ejemplo 1.1 y por el Teorema 1.3, se obtiene:

   - $n^2 = \Omega(21n^2 + 15n + 8)$.
   - $3^n = \Omega(2^n)$.
   - $n = \Omega(4n + 100)$.
   - $(n+1)! = \Omega(n!)$. $\diamondsuit$

Definición 1.3 (Notación Θ). Sea g una función de $\mathbb{N}$ en $\mathbb{R}^*$.

$$\Theta(g(n)) = \{f : \mathbb{N} \to \mathbb{R}^* : \text{existen constantes positivas } c_1,\ c_2 \text{ y } n_0 \text{ tales que } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n), \text{ para todo } n \geq n_0\}$$

Se puede observar que $\Theta(g(n)) = \mathrm{O}(g(n)) \bigcap \Omega(g(n))$.

Usualmente, si $f(n) \in \Theta(g(n))$ se escribe $f(n) = \Theta(g(n))$ y se lee "$f(n)$ es theta de

$g(n)$". En otras palabras, $\Theta(g(n))$ denota el conjunto de funciones asintóticamente acotadas por $g(n)$. La Figura 1.4 muestra el significado de la notación Θ.

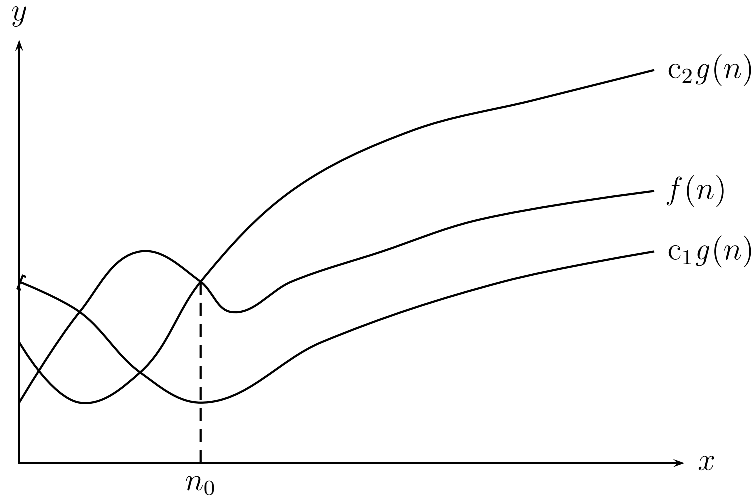


Figura 1.4: $f(n) = \Theta(g(n))$

Ejemplo 1.5

$n + 2 = \Theta(n + 1)$ En efecto:

$n + 1 < n + 2 \leq 2(n + 1)$

Haciendo $c_1 = 1$, $c_2 = 2$ y $n_0 = 1$, se obtiene que $n + 2 = \Theta(n + 1)$. ♢

1.2. GRAFOS

Se puede pensar en un grafo como un conjunto de puntos en un plano o en un espacio y un conjunto de segmentos lineales (o curvas) cada uno de los cuales une dos puntos o une un punto con él mismo.

Ejemplo 1.6

Una representación pictórica de un grafo se muestra en la Figura 1.5. ♢

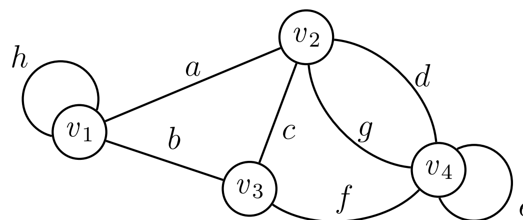


Figura 1.5: Representación de un grafo.

Los grafos son modelos muy útiles para analizar un rango grande de problemas prácticos en los cuales los puntos y las conexiones entre ellos tienen alguna interpretación física o conceptual.

Por ejemplo, la red de calles de una ciudad, los circuitos eléctricos, las moléculas orgánicas, son algunas de las estructuras que pueden ser representadas con grafos.

Definición 1.4 (Grafo). Un grafo G=(V, E, f) es una estructura matemática que consta de un conjunto no vacío V denominado conjunto de ***vértices*** (o nodos), un conjunto E de ***aristas*** y una función inyectiva f del conjunto de aristas E en un conjunto de pares ordenados o desordenados de V. Si una arista se corresponde con un par ordenado, entonces se dice que es una ***arista dirigida***; en caso contrario, se denomina ***arista no dirigida***.
Si una arista $e \in$ E está asociada con un par ordenado (u, v) o con un par desordenado $\{u, v\}$, en donde $u, v \in$ V, entonces se dice que la arista e conecta o une los vértices u y v, los cuales son llamados sus ***vértices finales***.
Los pares de vértices que estén conectados por una arista dentro de un grafo se denominan ***vértices adyacentes***.
Con frecuencia, es conveniente escribir los grafos en la forma (V, E) o también, simplemente como G. En el primer caso, cada arista se representa directamente como el par con el cual se corresponde, lo cual no hace necesaria la especificación de f si f es una correspondencia uno a uno.

Definición 1.5 (Bucle). Un bucle es una arista de un grafo que conecta un vértice consigo mismo.

Definición 1.6 (Arista propia). Una arista propia es una arista que no es bucle.

Definición 1.7 (Multi-arista). Una multi-arista es una colección de dos o más aristas que tienen los mismos vértices finales.

Ejemplo 1.7

El grafo de la Figura 1.5 tiene conjunto de vértices $V = \{v_1, v_2, v_3, v_4\}$ y conjunto de aristas $E = \{a, b, c, d, e, f, g, h\}$. Las aristas e y h son bucles, el vértice v_1 es adyacente a los vértices v_2 y v_3 pero no al vértice v_4, y el conjunto de aristas $\{g,\ d\}$ es una multi-arista. Observese que las aristas a, b, c, d, f y g están asociadas a los pares no ordenados $\{v_1, v_2\}, \{v_1, v_3\}, \{v_3, v_2\}, \{v_2, v_4\}, \{v_3, v_4\}, \{v_2, v_4\}$ respectivamente.

En los diagramas, las aristas representadas por medio de flechas, se denominan *dirigidas*. Si $e \in$ E es una arista dirigida asociada al par ordenado de vértices (u, v) entonces, se dice que la arista e comienza (o sale) del vértice u y llega al (o termina en el) nodo v. $\diamondsuit$

Definición 1.8 (Grafo dirigido). Un grafo dirigido (o digrafo) es un grafo en el cual toda arista es dirigida.

Definición 1.9 (Grafo no dirigido). Un grafo no dirigido es un grafo en el cual todas las aristas son no dirigidas.

Definición 1.10 (Grafo parcialmente dirigido). Un grafo parcialmente dirigido es un grafo que tiene tanto aristas dirigidas como aristas no dirigidas.

Definición 1.11 (Grafo subyacente). Un grafo subyacente de un grafo dirigido o parcialmente dirigido es el grafo que resulta de borrar todas las direcciones en las aristas de G.

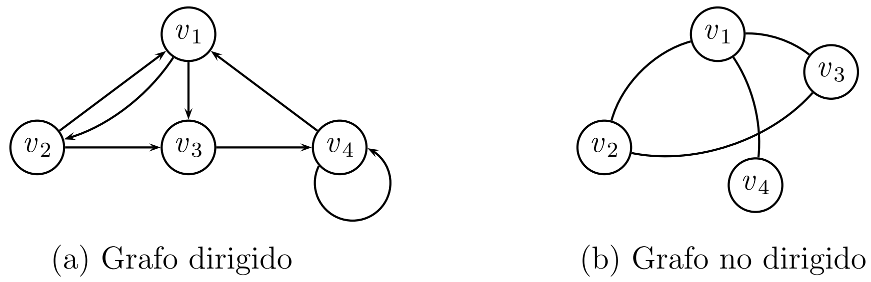


Figura 1.6: Representación pictórica de los grafos

Ejemplo 1.8

El grafo subyacente del grafo de la Figura 1.6(a) es mostrado en la Figura 1.7. $\diamondsuit$

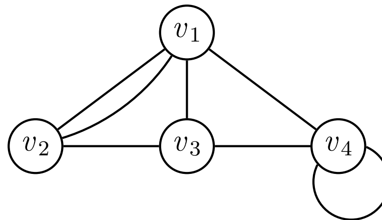


Figura 1.7: Grafo subyacente.

Definición 1.12 (Grafo simple). Un grafo o digrafo es simple si éste no tiene bucles ni multi-aristas.

Ejemplo 1.9

El digrafo de la Figura 1.8 es simple, pero su grafo subyacente no es simple. $\diamondsuit$

Figura 1.8: Digrafo simple y su grafo subyacente

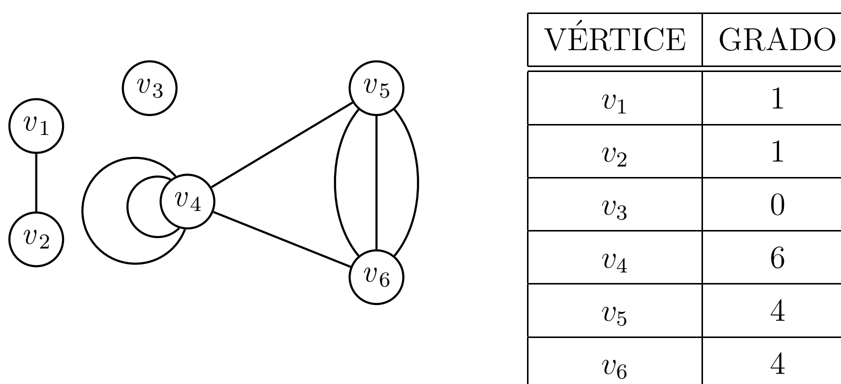
Definición 1.13 (Aristas adyacentes). Dos aristas son adyacentes si tienen un vértice final en común.

Definición 1.14 (Arista incidente). Si el vértice v es un vértice final de la arista e entonces se dice que e es incidente en v.

Definición 1.15 (Grado de un vértice). El grado de un vértice v en un grafo G es el número de aristas propias incidentes en v mas dos veces el número de bucles de v.

Ejemplo 1.10

La Figura 1.9 muestra un grafo y el grado de sus vértices. ♢



VÉRTICE	GRADO
v_1	1
v_2	1
v_3	0
v_4	6
v_5	4
v_6	4

Figura 1.9: Un grafo y el grado de sus vértices.

Definición 1.16 (Grafo completo). Un grafo completo es un grafo simple tal que todo par de vértices es unido por una arista. Los grafos completos de n nodos se denotan por K_n.

Ejemplo 1.11

La Figura 1.10 muestra los cinco primeros grafos completos. ♢

Definición 1.17 (Grafo bipartito). Un grafo bipartito G es un grafo cuyo conjunto de vértices V puede ser particionado en dos subconjuntos U y W, tal que cada arista de G tiene un vértice final en U y un vértice final en W.

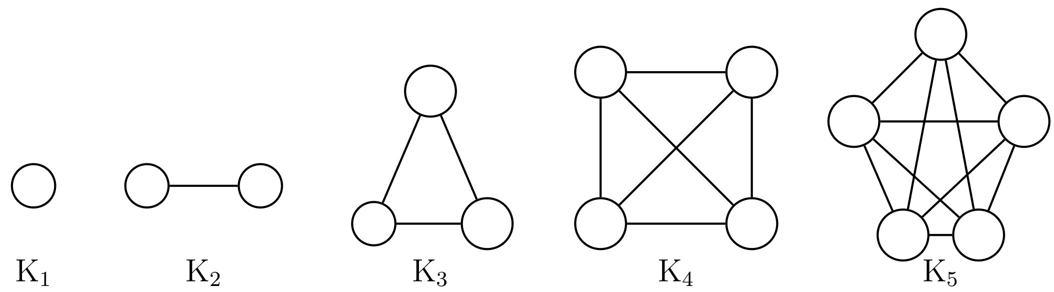


Figura 1.10: Ejemplos de grafos completos.

Ejemplo 1.12

Dos grafos bipartitos se muestran en la Figura 1.11. Los elementos de los conjuntos de vértices U y W son indicados en distinto color. ♢

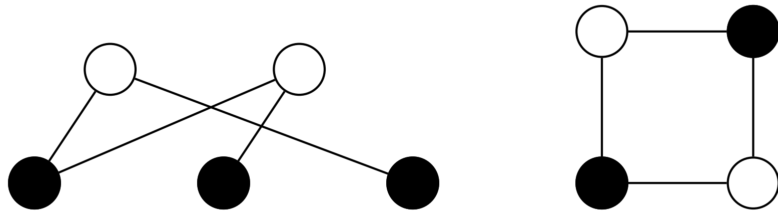


Figura 1.11: Dos grafos bipartitos.

Ejemplo 1.13

El grafo simple mas pequeño que no es bipartito es el grafo completo K_3, que se muestra en la Figura 1.12. ♢

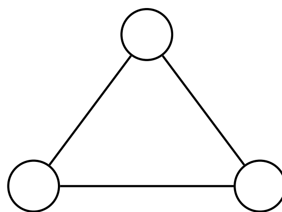
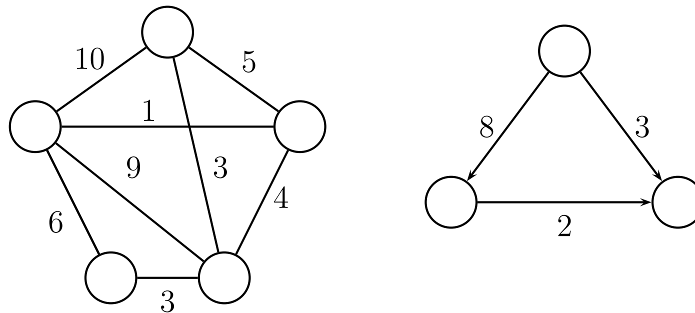


Figura 1.12: El grafo simple mas pequeño no bipartito.

Definición 1.18 (Grafo con peso). Un grafo con peso es un grafo G =(V, E) con una función de E en $\mathbb{R}$ tal que a cada arista le asigna un número, llamado peso.

Ejemplo 1.14

Dos grafos con peso se muestran en la Figura 1.13. $\diamondsuit$



(a) Grafo no dirigido con peso (b) Grafo dirigido con peso

Figura 1.13: Grafos con peso.

Definición 1.19 (Subgrafo). Un subgrafo G' = (V', E') de un grafo G = (V, E), es un grafo tal que V' $\subseteq$ V y E' $\subseteq$ E y para todo $\{a, b\} \in$ E' se cumple que a, $b \in$ V'.

Ejemplo 1.15

Un subgrafo de un grafo G = (V, E) con V = $\{v_1, v_2, v_3, v_4, v_5\}$ se muestra en la figura 1.14. $\diamondsuit$

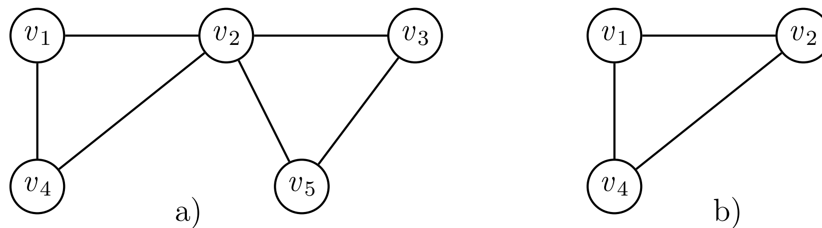


Figura 1.14: (a) Grafo G = (V, E). (b) Un subgrafo G' = (V', E') con V' = $\{v_1, v_2, v_4\}$ del grafo G = (V, E).

Definición 1.20 (Recorrido). En un grafo no dirigido, un recorrido del vértice v_0 al vértice v_n es una sucesión alternante de vértices y aristas,

$$W = \langle v_0, e_1, v_1, e_2, \ldots, v_{n-1}, e_n, v_n \rangle$$

tal que e_i es la arista que une a los vértices v_{i-1} y v_i, para $i = 1,2,\ldots,n$.
Si el grafo es simple, entonces el recorrido del vértice v_0 al vértice v_n se expresa simplemente como la sucesión de vértices, así:

$$W = \langle v_0, v_1, \ldots, v_n \rangle$$

Ejemplo 1.16
La Figura 1.15 muestra un recorrido en un grafo de cuatro vértices. $\diamondsuit$

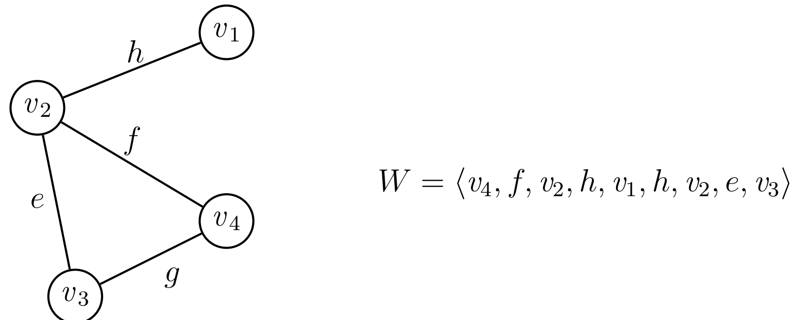


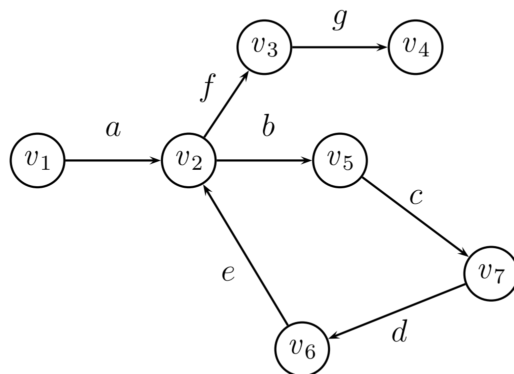
Figura 1.15: Un recorrido de v_4 a v_3

Definición 1.21 (Recorrido dirigido). En un digrafo, un recorrido dirigido de v_0 a v_n es una sucesión alternante de vértices y aristas dirigidas,

$$W = \langle v_0, e_1, v_1, e_2, \ldots, v_{n-1}, e_n, v_n \rangle$$

tal que la arista e_i comienza en el vértice v_{i-1} y termina en el vértice v_i, para $i = 1, 2, \ldots, n$.

Ejemplo 1.17
La Figura 1.16 muestra un recorrido dirigido sin aristas repetidas. $\diamondsuit$



$$W = \langle v_1, a, v_2, b, v_5, c, v_7, d, v_6, e, v_2, f, v_3, g, v_4 \rangle$$

Figura 1.16: Un recorrido dirigido.

Definición 1.22 (Grafo conexo). Un grafo no dirigido es un grafo conexo si para todo par de vértices u y v, existe un recorrido de u a v.

Definición 1.23 (Digrafo conexo). Un digrafo es un digrafo conexo (o débilmente conexo) si su grafo subyacente es conexo.

Los grafos de la Figura 1.17 son grafos conexos.

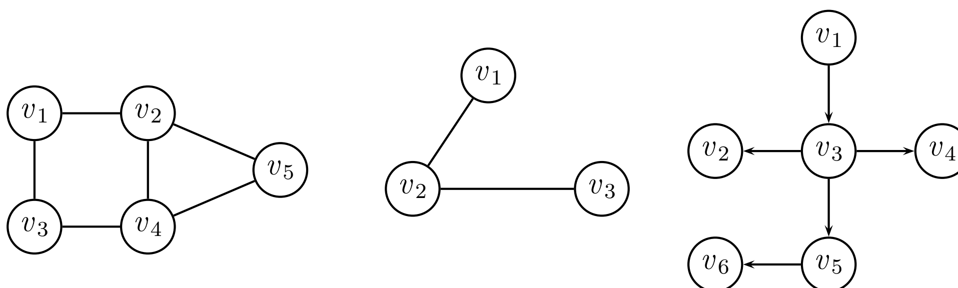


Figura 1.17: Grafos conexos.

Ejemplo 1.18

En la Figura 1.18 se muestra un grafo no conexo formado por subgrafos conexos G_1, G_2, G_3. $\diamondsuit$

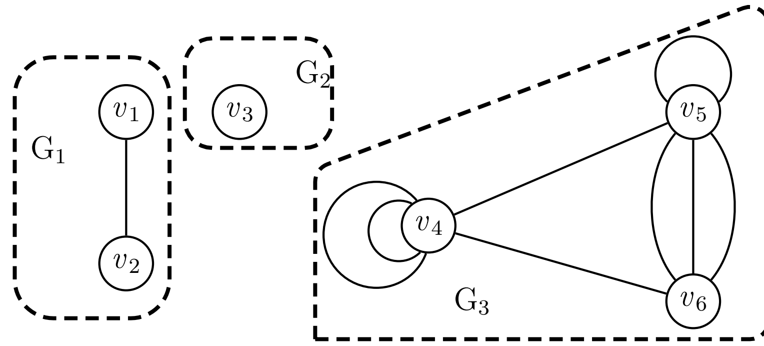
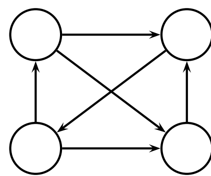


Figura 1.18: Un grafo no conexo con tres subgrafos conexos.

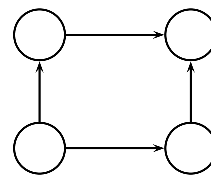
Definición 1.24 (Digrafo fuertemente conexo). Un digrafo es fuertemente conexo si para toda pareja de vértices del grafo los dos vértices de la pareja se pueden alcanzar uno desde el otro.

Ejemplo 1.19

Para los digrafos dados en la Figura 1.19 el grafo de la parte (a) es fuertemente conexo y el de la parte (b) es débilmente conexo pero no fuertemente conexo. ♢



(a) Digrafo fuertemente conexo.



(b) Digrafo débilmente conexo

Figura 1.19: Conectividad en digrafos.

Definición 1.25 (Camino). Un camino es un recorrido sin vértices repetidos, excepto posiblemente los vértices inicial y final. Análogamente, en un digrafo un ***camino dirigido*** es un recorrido dirigido sin vértices repetidos.

Ejemplo 1.20

Para el grafo que se muestra en la Figura 1.20, $W = \langle a,\ e,\ f,\ a,\ d \rangle$ es la sucesión de

aristas de un recorrido que no es camino, $T = \langle a,\ b,\ c,\ d,\ e\rangle$ es la sucesión de aristas de un recorrido que no es camino y $Z = \langle f,\ a,\ b,\ c\rangle$ es la sucesión de aristas de un camino.$\diamondsuit$

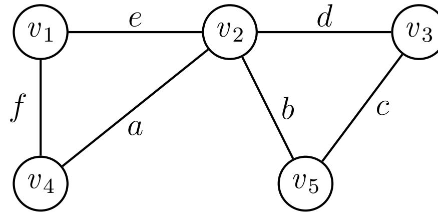


Figura 1.20: Grafo con un camino

Definición 1.26 (Ciclo). Un ciclo (o circuito) es un camino cerrado, esto es, un camino que comienza y termina en el mismo vértice.

Ejemplo 1.21

El grafo de la Figura 1.20 tiene dos ciclos cuyas sucesiones de vértices son:

$$C_1 = \langle\ v_1,\ v_2,\ v_4\rangle \quad \text{y} \quad C_2 = \langle v_2,\ v_3,\ v_5\rangle \qquad \diamondsuit$$

Ejemplo 1.22

La Figura 1.21 muestra un digrafo que tiene tres ciclos, cuyas sucesiones de vértices son:

$$\begin{aligned} C_1 &= \langle v_2,\ v_1,\ v_4,\ v_5,\ v_2\rangle \\ C_2 &= \langle v_2,\ v_3,\ v_6,\ v_2\rangle \\ C_3 &= \langle v_2,\ v_3,\ v_6,\ v_5,\ v_2\rangle \qquad \diamondsuit \end{aligned}$$

Definición 1.27 (Grafo acíclico). Un grafo simple que no tenga ningún ciclo se denomina acíclico.

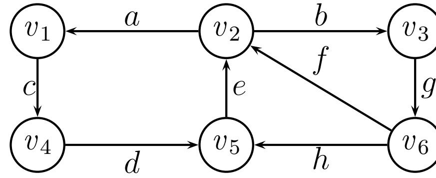


Figura 1.21: Digrafo con tres ciclos.

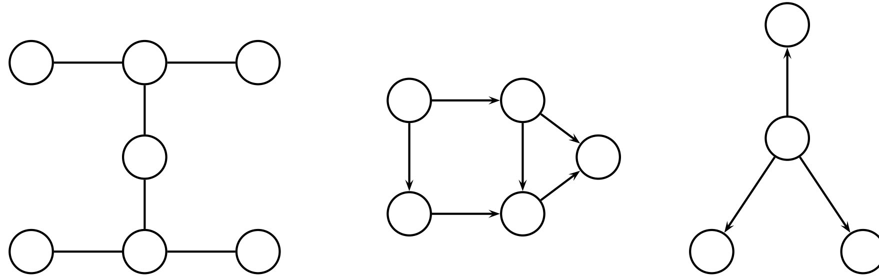


Figura 1.22: Grafos acíclicos.

Ejemplo 1.23
En la Figura 1.22 se dan ejemplos de grafos acíclicos. $\diamondsuit$

Definición 1.28 (Ciclo hamiltoniano). Un ciclo hamiltoniano en un grafo es un ciclo que recorre todos los vértices del grafo.

Ejemplo 1.24
Un grafo con un ciclo hamiltoniano se muestra en la Figura 1.23. $\diamondsuit$

Definición 1.29 (Matriz de adyacencia). Sea G = (V, E) un grafo simple con $V = \{v_1, v_2, \ldots, v_n\}$ y en el que se supone que los nodos están ordenados desde v_1 hasta v_n. La matriz Ad, $n \times n$, cuyos elementos a_{ij} están dados por:

$$a_{ij} = \begin{cases} 1, & \text{si } \{v_i, v_j\} \in \text{E}, \\ 0, & \text{en caso contrario.} \end{cases}$$

se denomina matriz de adyacencia del grafo G.

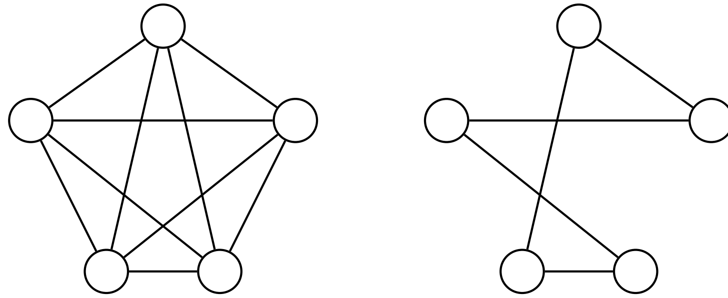


Figura 1.23: Un grafo y uno de sus ciclos hamiltonianos.

Ejemplo 1.25

En la Figura 1.24 se muestra un digrafo y su matriz de adyacencia. $\diamondsuit$

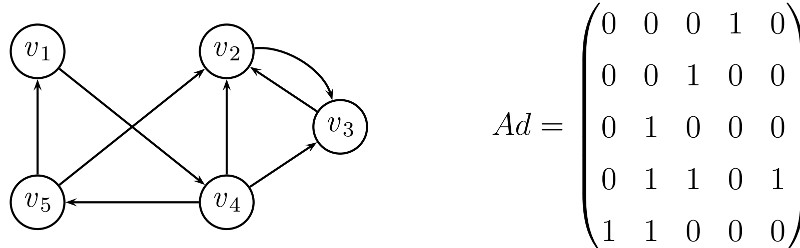


Figura 1.24: Un digrafo y su matriz de adyacencia.

Ejemplo 1.26

Para los grafos no dirigidos, la matriz de adyacencia es simétrica, esto es, $a_{ij} = a_{ji}$ para todo i y para todo j. La Figura 1.25 muestra un grafo no dirigido y su matriz de adyacencia. $\diamondsuit$

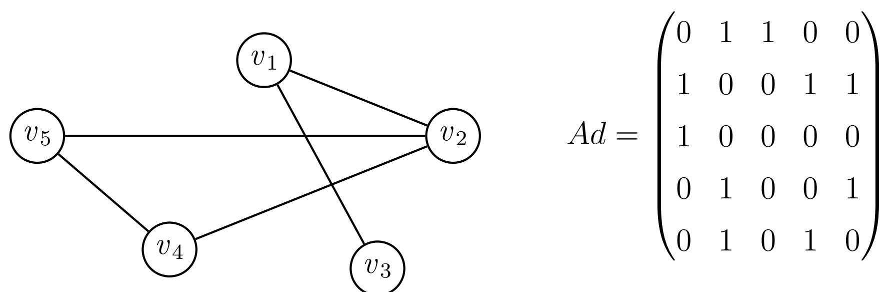


Figura 1.25: Un grafo y su matriz de adyacencia.

1.3. PROBLEMAS Y ALGORITMOS

En matemáticas y en todas las ciencias surgen continuamente problemas para los cuales se quiere encontrar su solución, para esto usualmente se desarrollan procedimientos que son llamados algoritmos. Así, un algoritmo podría verse como un conjunto de pasos a seguir para resolver un problema determinado. Sin embargo, actualmente existen una gran cantidad de problemas para los cuales no se ha encontrado ningún algoritmo que los resuelva o al menos no en forma "satisfactoria". En el contexto de complejidad computacional, un problema usualmente puede ser visto como una situación en la que surge una pregunta que se desea responder, a esta situación generalmente la constituyen ciertas componentes variables o parámetros que pueden ser grafos, vectoress, matrices, conjuntos entre otros y cuya solución puede requerir el cumplimiento de algunas condiciones, por ejemplo, que un determinado conjunto no sobrepase una cota dada. Sin embargo dar una definición precisa del concepto de problema es bastante complejo debido a la gran variedad de problemas que surgen en todas las áreas. No obstante si un problema se puede expresar mediante un modelo formal a menudo esto resulta ser beneficioso ya que permite una mayor facilidad en su manejo. A continuación se presenta la definición de problema que aquí será usada.

Definición 1.30 (Problema). Un problema se define como sigue:

a) Una descripción general de todos los parámetros.

b) Una declaración de las condiciones que debe cumplir la solución.

Una ***instancia*** de un problema se obtiene asignando valores particulares a todos los parámetros del problema.

Ejemplo 1.27
Sea T$[1 \ldots n]$ un vector de n elementos. Se plantea el problema de ordenar los elementos

de T en orden ascendente. Este problema se conoce usualmente como el problema de *ordenación*.

El párametro de este problema es el vector de n elementos y la solución es un vector V[1. . . n] de elementos de T tal que V[1]≤V[2]≤ . . . ≤V[n].

Una instancia del problema de ordenación está dada por:
el vector [2,8,3,6,1,9,4] de 7 elementos y la solución para esta instancia es el vector [1,2,3,4,6,8,9]. ♢

Ejemplo 1.28
Sean A una matriz $m \times r$ y B una matriz $r \times n$. Se plantea el problema de encontrar el producto A·B. A este problema se le conoce con el nombre de *multiplicación de matrices*.

Aquí, los parámetros son las matrices A y B y la solución es una matriz C que es el producto A·B.

Una instancia de este problema está dada por:

$$A = \begin{pmatrix} 2 & 4 \\ 1 & 9 \\ 8 & 1 \end{pmatrix}, \quad B = \begin{pmatrix} 9 & 0 & 2 \\ 5 & 7 & 6 \end{pmatrix}$$

Y la solución para esta instancia es:

$$C = \begin{pmatrix} 38 & 28 & 28 \\ 54 & 63 & 56 \\ 77 & 7 & 22 \end{pmatrix} \qquad \diamondsuit$$

Ejemplo 1.29
Sea G = (V, E) un grafo donde V es el conjunto de vértices y E es el conjunto de aristas. Se plantea el problema de encontrar un ciclo hamiltoniano en G. Este problema es conocido como el problema del *ciclo hamiltoniano*.

El parámetro de este problema es un grafo G =(V, E). Una solución es un ciclo hamiltoniano; esto es, un ordenamiento $\langle v_1, v_2, \ldots, v_k\rangle$ de todos los vértices en V tales que $\{v_i, v_{i+1}\} \in E$ para $1 \leq i < k$ y $\{v_k, v_1\} \in E$.
Una instancia del problema del circuito hamiltoniano ilustrado en la Figura 1.26 esta dado por:
G=(V, E)
V={1, 2, 3, 4, 5}
E={{1, 2}, {1, 5}, {1, 4}, {2, 4}, {2, 3}, {3, 4}, {3, 5}, {4, 5}}.
La sucesión $\langle 1, 2, 3, 5, 4\rangle$ es una solución para esta instancia. ♢

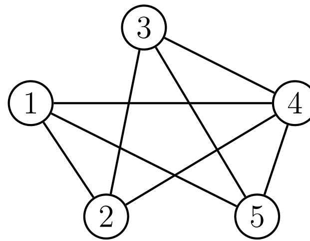


Figura 1.26: Una instancia del problema HC

Definición 1.31 (Problema de decisión). Un problema de decisión es un problema que tiene solo dos posibles soluciones, la respuesta SI o la respuesta NO.

En un problema de decisión Π* se pueden considerar el conjunto D_Π de instancias y el subconjunto $Y_\Pi \subseteq D_\Pi$ de instancias con respuesta SI.

*En adelante, Π denotará un problema.

El formato estándar que se usará para representar problemas de decisión consta de dos partes:

- Una instancia genérica del problema en términos de varios parámetros, los cuales pueden ser conjuntos, grafos, funciones, números, etc...
- Una pregunta con respuesta SI o NO, formulada en términos de la instancia genérica.

Ejemplo 1.30
El problema de decisión del ciclo hamiltoniano puede ser descrito como sigue:

CICLO HAMILTONIANO (HC)

INSTANCIA : Un grafo G=(V, E)

PREGUNTA : ¿ Existe un ordenamiento $\langle v_1, v_2, \ldots, v_n \rangle$ de V, con $n = |V|$ tal que $\{v_i, v_{i+1}\} \in E$ para $1 \leq i < n$ y $\{v_n, v_1\} \in E$?. ♢

Ejemplo 1.31
El problema de decisión del agente viajero puede ser descrito como sigue:

AGENTE VIAJERO

INSTANCIA : Un conjunto finito $C = \{c_1, c_2, \ldots, c_m\}$ de "ciudades", una "distancia" $d(c_i, c_j) \in \mathbb{Z}^+$ para cada par de ciudades c_i, $c_j \in C$ y una cota $k \in \mathbb{Z}^+$.

PREGUNTA : ¿Existe un tour de todas las ciudades en C que tenga una longitud total no mas que k, esto es, un ordenamiento $\langle c_{\Gamma(1)}, c_{\Gamma(2)}, \ldots, c_{\Gamma(m)} \rangle$ de C tal que $\sum_{i=1}^{m-1} d(c_{\Gamma(i)}, c_{\Gamma(i+1)}) + d(c_{\Gamma(m)}, c_{\Gamma(1)}) \leq k$?. ♢

Definición 1.32 (Algoritmo). Un algoritmo es un método para resolver un problema o una clase de problemas mediante una serie de pasos precisos, definidos y finitos.

Cuando se utiliza un algoritmo para encontrar la respuesta de un problema concreto, debe suceder que al realizar la serie de pasos, si se aplican correctamente, se obtendrá la respuesta correcta. Un algoritmo debe ser preciso, esto es, debe indicar el orden de realización de cada paso; además, los pasos que se siguen deben ser definidos en el sentido que si el algoritmo se sigue dos veces (con la misma instancia), se obtiene el mismo resultado cada vez; así, la ejecución de un algoritmo no debe de implicar ninguna decisión subjetiva, ni tampoco debe de hacer uso de la intuición ni de la creatividad. Por tanto se puede considerar que una receta de cocina es un algoritmo si describe precisamente la forma de preparar un cierto plato, proporcionando las cantidades exactas que deben de utilizarse y también instrucciones detalladas acerca del tiempo que debe de guisarse. Por otra parte, si se incluye nociones vagas tales como "salpimentar a su gusto"o "guísese hasta que esté medio hecho "entonces no se podría llamar algoritmo. Por último, un algoritmo debe tener un número determinado de pasos y además producir un resultado en un tiempo finito. [BB97]

Sin embargo, existen otras clases de algoritmos que no cumplen todas las características de esta definición, entre ellos están los *algoritmos probabilísticos* y los algoritmos heurísticos. Los algoritmos probabilísticos se diferencian de los algoritmos descritos arriba, fundamentalmente, en que un mismo algoritmo probabilístico se puede comportar de forma distinta cuando se aplica dos veces a una misma instancia. Como consecuencia de esto, ellos pueden producir algunas veces resultados inexactos, aunque esto debe suceder con probabilidad suficientemente pequeña para cualquier caso dado; de manera que para llegar a un nivel de probabilidad prefijado basta ejecutar el algoritmo varias veces sobre el caso deseado. Lo cual no ocurre en los algoritmos convencionales, que para cada entrada, siempre deben llegar a la respuesta correcta. Por otra parte, un algoritmo heurístico o simplemente heurística, es un procedimiento que utiliza herramientas o "trucos" no convencionales que en ocasiones no tienen un orden lógico y que permiten tomar decisiones a partir de un conocimiento intuitivo del problema a resolver. Además, este tipo de algoritmos puede producir una solución aproximada para el problema asociado, incluso la solución óptima pero también, por otro lado, puede no producir una solución, o dar lugar a una que no sea precisamente óptima.

Ejemplo 1.32
Un algoritmo que resuelve el problema de ordenación presentado en el Ejemplo 1.26 es el siguiente:

Procedimiento INSERTAR (T[1... n])
    para $i \longleftarrow 2$ **hasta** n **hacer**
        $x \longleftarrow \mathrm{T}[i]$
        $j \longleftarrow i - 1$
        mientras $j > 0$ y $x < \mathrm{T}[j]$ **hacer**
            $\mathrm{T}[j+1] \longleftarrow \mathrm{T}[j]$
            $j \longleftarrow j - 1$
        $\mathrm{T}[j+1] \longleftarrow x$

Este procedimiento examina sucesivamente todos los elementos del vector desde el segundo hasta el n-ésimo, compara cada elemento con todos los anteriores y lo inserta en la posición adecuada dentro del mismo vector.

Es de anotar que se han desarrollado otros algoritmos que resuelven este mismo problema mucho más rápido. ♢

Ejemplo 1.33
El algoritmo clásico para el problema de multiplicación de matrices presentado en el Ejemplo 1.27, proviene directamente de su definición. El producto de una matriz A de tamaño mxr por una matriz B de tamaño o rxn es una matriz C de tamaño mxn definida por:

$$c_{ij} = \sum_{k=1}^{r} A_{ik} B_{kj},\ 1 \leq i \leq m,\ 1 \leq j \leq n$$

El algoritmo para realizar este producto es el siguiente:

Para $i \longleftarrow 1$ **hasta** m **hacer**

  Para $j \longleftarrow 1$ **hasta** n **hacer**

    $\mathrm{C}[i, j] \longleftarrow 0$

    Para $k \longleftarrow 1$ **hasta** r **hacer**

      $\mathrm{C}[i, j] \longleftarrow \mathrm{C}[i, j] + \mathrm{A}[i, k]\,\mathrm{B}[k, j]$

Hacia el final de los años 60, Strassen mejoró este algoritmo, para el caso de matrices cuadradas, utilizando otra técnica. $\diamondsuit$

1.4. ALFABETOS Y LENGUAJES

En esta sección se presentan los conceptos de alfabeto y lenguaje que son necesarios para introducir la noción de esquema de codificación. Un ejemplo muy conocido de alfabeto es el utilizado por un computador actual, compuesto por dos elementos, el cero y el uno, los cuales son usados por la máquina para el manejo de la información.

Definición 1.33 (Alfabeto). Un alfabeto es un conjunto enumerable (finito o infinito) de símbolos indivisibles.

Ejemplo 1.34
Los siguientes conjuntos son ejemplos de alfabetos:

$\Sigma_1 = \{\ 0,\ 1\}$
$\Sigma_2 = \{\ \mathrm{a},\ \mathrm{b}, \ldots,\ \mathrm{z},\ \mathrm{A},\ \mathrm{B}, \ldots,\ \mathrm{Z}\}$
$\Sigma_3 = \{\ \mathrm{c},\ [,\ ],\ /,\ 0,\ 1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 8,\ 9\}$
$\Sigma_4 = \{\ \{,\ (\ \} \cup \{\ v_i : i \in \mathbb{N}\ \} \bigcup \{\ ,\ \}$
$\Sigma_5 = \mathbb{N} \cup \{\ (,\ /,\ )\ \}$ $\diamondsuit$

Definición 1.34 (Cadena). Una cadena es una sucesión finita de símbolos de un alfabeto Σ. Como cadena de Σ se incluye la cadena vacía denotada por ε.

Ejemplo 1.35

1. Para $\Sigma_1 = \{ 0, 1\}$ son cadenas:
   01010101, 01, 00, 000, 1111, 1101.
2. Para $\Sigma_2 = \{$ a, b,..., z, A, B,..., Z$\}$ son cadenas:
   abea, casa, BbCc, oso.
3. Para $\Sigma_3 = \{$c, [,], /, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9$\}$ son cadenas:
   c [/ 0 1, c[0], 2 / [3 /, c[1]c[2]c[3]//1/2//8. $\diamondsuit$

Definición 1.35 (Lenguaje universal). El conjunto Σ^* de todas las cadenas que se pueden construir sobre un alfabeto Σ se denomina lenguaje universal para Σ, esto es:

$$\Sigma^* = \{a_1 a_2 \ldots a_n : n \in \mathbb{Z}^+\} \bigcup \{\varepsilon\}, \; a_i \in \Sigma.$$

Ejemplo 1.36
Para $\Sigma_1 = \{0, 1\}$, el conjunto Σ_1^* contiene la cadena vacía ε y todas las posibles cadenas finitas de ceros y unos. $\diamondsuit$

Definición 1.36 (Lenguaje). Sea Σ un alfabeto. Un lenguaje L sobre Σ denotado por L(Σ) es un subconjunto de Σ^*, es decir,
L(Σ) es un lenguaje sobre Σ, si y sólo si, L(Σ) $\subseteq \Sigma^*$.

Ejemplo 1.37

1. Si Σ es un alfabeto, entonces Σ^* y $\emptyset$ son lenguajes sobre Σ.

2. Sea $\Sigma_1 = \{0, 1\}$. Los siguientes conjuntos son lenguajes sobre Σ_1.
   $L_1(\Sigma_1) = \{01, 001, 0101, 111, 11011\}$.
   $L_2(\Sigma_1) = \{a_1 a_2 \ldots a_n : a_i \in \{0, 1\}$ para $i = 1, \ldots n - 1$ y $a_n = 1\}$.

3. Sea $\Sigma_2 = \{a, b, \ldots, z, A, B, \ldots, Z\}$. Los siguientes conjuntos son lenguajes sobre Σ_2.
   $L_1(\Sigma_2) = \{$calle, gato, mesa, lápiz$\}$.
   $L_2(\Sigma_2) = \{$apq, rst, uxz, nbLo$\}$. $\diamondsuit$

1.5. ESQUEMAS DE CODIFICACIÓN

Si un algoritmo resuelve un problema, las instancias del problema deben ser representadas de tal forma que el computador lo entienda. Por ejemplo, en el problema del circuito hamiltoniano, una instancia está compuesta por un grafo. Un algoritmo que resuelva este problema debe tomar como parámetro dicha instancia; esto implica que se debe encontrar un mecanismo adecuado de describir el grafo para ser proporcionado al algoritmo de tal forma que sea comprendido por él.

Definición 1.37 (Esquema de codificación). Sea Π un problema y S el conjunto de todas las instancias de Π. Un esquema de codificación para Π es una transformación e de S en Σ^* para algún alfabeto Σ.

Es decir, e provee una forma de describir cada instancia de Π por una cadena apropiada de Σ.

Definición 1.38 (Lenguaje asociado a un problema de decisión). Sea Π un problema de decisión y e un esquema de codificación para Π. El lenguaje asociado con Π y e, denotado por L[Π,e] está dado por:

$$\text{L}[\Pi,e] = \{x \in \Sigma^* : \Sigma \text{ es el alfabeto usado por } e \text{ y } x \text{ es la codificación bajo } e \text{ de una instancia } I \in Y_\Pi\}$$

Un problema Π y un esquema de codificación e para Π, particiona Σ^* en tres clases de cadenas: aquellas que no codifican instancias de Π, aquellas que codifican instancias de Π para las cuales la respuesta es NO y aquellas que codifican instancias de Π para las cuales la respuesta es SI. Las cadenas de la tercera clase son las que componen el lenguaje L$[\Pi,e]$.

Ejemplo 1.38
Considérese el siguiente problema:

INSTANCIA: Un entero positivo m.
PREGUNTA: ¿m es par?

Esquema de codificación: Representación en binario.
$\Sigma = \{0, 1\}$
Σ^*: Todas las cadenas finitas de ceros y unos.

El problema anterior y su esquema de codificación particionan Σ^* en tres clases:

1. Cadenas que no codifican instancias del problema.
   Por ejemplo, {00, 000, 01, 001, 010, ...}.
2. Cadenas que codifican instancias del problema y cuya respuesta es NO.
   Por ejemplo, {1, 11, 101, 111, 1001, ...}.
3. Cadenas que codifican instancias del problema y cuya respuesta es SI.
   Por ejemplo, {0, 10, 100, 110, 1010, ...}. Este es el conjunto de todos los números en binario que terminan en cero.

Esta última clase es el lenguaje asociado al problema de decidir si un número es par.$\diamondsuit$

2. MODELOS DE COMPUTACIÓN BÁSICOS

Intuitivamente, el significado de computar es ser capaz de especificar una secuencia finita de acciones con el objetivo de obtener algún resultado. El intento de muchos matemáticos de formalizar la noción de algoritmo, los llevó ha desarrollar muchos modelos de computación distintos que finalmente fueron mostrados equivalentes en poder computacional a la máquina de Turing.

En este capítulo se presentan algunos modelos de computación básicos como la máquina de Turing 1-cinta que fué desarrollada por Alan Mathison Turing en 1935 y su generalización a k cintas, la máquina de Turing no determinística, la máquina de Turing Universal y la máquina de acceso aleatorio (RAM), enfocando la atención principalmente a mostrar sus funcionamientos. Además, se describe el problema de la parada y se demuestra su caracter irresoluble. También se mencionan sin demostración los resultados que muestran la equivalencia entre estos modelos de computación y la máquina de Turing. Por esto la máquina de Turing corresponde a la noción formal del concepto de algoritmo y será el modelo de computación formal que se usará en los próximos capítulos.

2.1. MÁQUINAS DE TURING DETERMINÍSTICAS

Una máquina de Turing determinística es un modelo computacional abstracto que captura y formaliza la noción de algoritmo. Estas máquinas pueden ser usadas para computar funciones, resolver problemas de decisión, etc..

2.1.1. Máquina de Turing 1-cinta

La entrada de una máquina de Turing es una cadena finita de símbolos de un determinado alfabeto, el cual, por convención siempre contendrá el símbolo especial denominado "símbolo blanco" denotado por b. Utiliza como medio para la entrada y salida de datos una cinta unidimensional bi-infinita dividida en celdas etiquetadas por $..., -2, -1, 0, 1, 2, ...$ las cuales pueden contener sólo un símbolo cada una.

La máquina consta de una cabeza de lectura-escritura que se puede mover libremente sobre la cinta en ambas direcciones, avanzando o retrocediendo sólo una celda en un paso, y tiene la capacidad de leer o escribir un símbolo en la celda. La máquina posee también un conjunto finito de estados que indican el estado en que se encuentra la máquina en algún instante de tiempo.

Una situación de la máquina se define como una dupla formada por un estado (que pertenece al conjunto de estados) y un símbolo (que pertenece al alfabeto). En un instante de tiempo, la situación actual de la máquina está determinada por el estado actual en el cual se encuentra y por el símbolo que está actualmente leyendo la cabeza de lectura-escritura.

Además, la parte fundamental de la máquina de Turing está dada por un conjunto

de instrucciones que representan el comportamiento de la máquina en cada instante de tiempo. Cada instrucción está compuesta por dos partes: la primera está formada por un estado y un símbolo e indica en qué situación se debe ejecutar la instrucción; la segunda parte indica lo que hace la máquina en esa situación, lo cual consiste en colocar la máquina en un nuevo estado, escribir un símbolo en la posición actual y mover la cabeza de lectura-escritura. De acuerdo a esto pueden suceder dos cosas, que la máquina tenga una sola instrucción para una situación en particular o que la máquina tenga más de una instrucción para una situación en particular. El primer caso caracteriza a las máquinas de Turing determinísticas y el segundo a las máquinas de Turing no determinísticas.

Formalmente, una máquina de Turing se define como sigue:

Definición 2.1 (Máquina de Turing 1-cinta). Una máquina de Turing determinística 1-cinta MT es una tupla $\langle Q, \Sigma, \delta \rangle$ donde:

1. Q es un conjunto finito llamado el *conjunto de estados*, el cual contiene un estado especial q_o llamado el estado inicial y tres estados especiales de parada: el estado de aceptación q_Y, el estado de rechazo q_N y el estado "*stop*".

2. Σ es un conjunto finito de símbolos llamado el *alfabeto* de la máquina, el cual siempre contiene el "símbolo blanco" b.

3. δ es una función, llamada *función de transición* definida de un subconjunto de $(Q - \{q_Y, q_N, stop\}) \times \Sigma$ en $Q \times \Sigma \times$ M donde M={I, D, N} es el conjunto de movimientos (I:izquierda, D:derecha, N:ningún movimiento).
   La función δ puede ser representada como un conjunto finito de instrucciones, donde cada instrucción es una quintupla de la forma (q, s, q', s', m) donde $q, q' \in Q,\ s, s' \in \Sigma$ y m $\in$ M tal que ningún par de quintuplas en el conjunto tienen los dos primeros elementos iguales.

Se debe observar de la definición que el hecho de que δ sea función refleja la característica principal de una máquina determinística, aunque en algunos casos se realizan

asignaciones innecesarias puesto que la máquina puede no necesitarlas todas para su funcionamiento. Por facilidad en la representación de δ se omitirán todas las asignaciones innecesarias.

La cadena de entrada $x \in (\Sigma - \{b\})^*$ es colocada desde el cuadrado 1 de la cinta hasta el cuadrado $|x|$ (longitud de x), un símbolo por cuadrado. Todos los otros cuadrados inicialmente contienen el símbolo blanco. La máquina comienza su operación en el estado q_0, con la cabeza de lectura-escritura leyendo el cuadrado 1, lo cual constituye su *situación inicial*.

Si la situación actual de la máquina en algún instante de tiempo es (q, s) y $\delta(q, s) = (q', s', m)$ entonces la máquina pasa del estado q al estado q', la cabeza de lectura-escritura borra s y escribe s' en su lugar y se mueve a la izquierda si m = I, a la derecha si m = D o no se mueve si m = N. Esto constituye un paso en la ejecución de la máquina y produce una nueva situación actual (q', s') con la cual se procede de igual forma, aplicando la función de transición.

La ejecución de la máquina termina si alcanza uno de los tres estados de parada, q_Y, q_N o *stop*. Si el estado alcanzado es q_Y, se dice que la máquina para aceptando la entrada, si q_N ha sido alcanzado entonces la máquina para rechazando la entrada. En muchos casos es de interés conocer la cadena obtenida al final de la ejecución de la máquina; por ejemplo, cuando se quiere computar una función. Para esto se utiliza el estado *stop*. Es decir, si la máquina de Turing alcanza el estado *stop*, la máquina para y la salida es la cadena contenida en la cinta en el momento de la parada.

Ejemplo 2.1

Considere la máquina de Turing $MT = \langle Q, \Sigma, \delta \rangle$ donde $Q = \{q_0, q_1, q_Y, q_N, stop\}$, $\Sigma = \{0, 1, b\}$ y δ se presenta en la tabla 2.1. Esta máquina toma una entrada escrita en binario y la multiplica por 4.

q	s	$\delta(q,\, s)$
q_0	0	$(q_0, 0, \mathrm{D})$
q_0	1	$(q_0, 1, \mathrm{D})$
q_0	b	$(q_1, 0, \mathrm{D})$
q_1	b	$(stop, 0, \mathrm{N})$

Tabla 2.1: Máquina de Turing que multiplica por 4.

La Figura 2.1 ilustra la computación de MT sobre la entrada 101. La salida de la máquina es la cadena 10100.

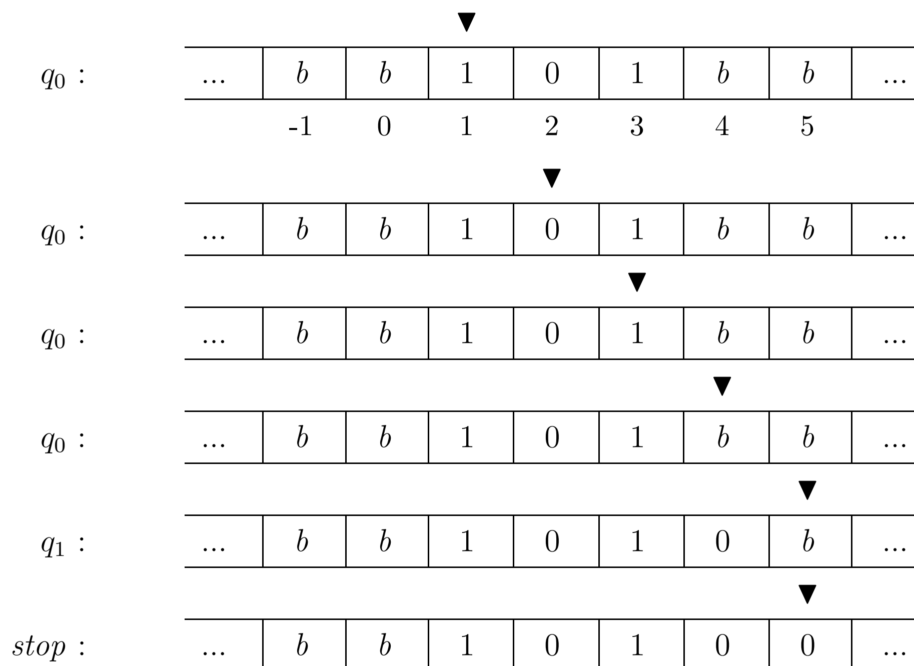


Figura 2.1: Una computación de la máquina MT de la tabla 2.1. ♢

Ejemplo 2.2

Una máquina de Turing que acepta solamente las cadenas en binario que terminan en 00 o 11 es mostrada en la tabla 2.2.

q	s	$\delta(q,\, s)$
q_0	0	$(q_0,\, 0,\, \text{D})$
q_0	1	$(q_0,\, 1,\, \text{D})$
q_0	b	$(q_1,\, b,\, \text{I})$
q_1	0	$(q_2,\, 0,\, \text{I})$
q_1	1	$(q_3,\, 1,\, \text{I})$
q_1	b	$(q_N,\, b,\, \text{N})$
q_2	0	$(q_Y,\, 0,\, \text{N})$
q_2	1	$(q_N,\, 1,\, \text{N})$
q_2	b	$(q_N,\, b,\, \text{N})$
q_3	0	$(q_N,\, 0,\, \text{N})$
q_3	1	$(q_Y,\, 1,\, \text{N})$
q_3	b	$(q_N,\, b,\, \text{N})$

a)

q	s	$\delta(q, s)$
q_0	/	$(q_0,\, /,\, \text{D})$
q_0	b	$(q_1,\, /,\, \text{D})$
q_1	/	$(q_1,\, /,\, \text{D})$
q_1	b	$(q_2,\, b,\, \text{I})$
q_2	/	$(q_3,\, b,\, \text{I})$
q_3	/	$(stop, b,\, \text{N})$

b)

Tabla 2.2: a) Máquina de Turing que acepta las cadenas en binario terminadas en 00 o 11. b) Máquina de Turing que suma dos números naturales codificados con "palitos ". $\diamondsuit$

Ejemplo 2.3

[GS01] Sea $MT=\langle Q, \Sigma, \delta\rangle$ donde $Q = \{q_0,\, q_1,\, q_2,\, q_3,\, q_4,\, stop\}$, $\Sigma = \{/,\, b\}$, una máquina de Turing que calcula la suma de dos números naturales codificados de la siguiente manera:

$0 = /$

$1 = //$

$2 = ///$

$3 = ////$

$\vdots$

$n = \underbrace{///\ldots/}_{n+1 \text{ veces}}$

$\vdots$

La función de transición δ para MT está dada en la tabla 2.2.

A continuación se calcula la suma de 2 y 3. Los datos de entrada pueden ser representados en la cinta de diferentes formas, una de ellas es colocando los números que se van a sumar separados por el símbolo b . La salida en este caso son seis "palitos "que representan el número 5. La figura 2.2 ilustra la computación de MT sobre la entrada 2+3. $\diamondsuit$

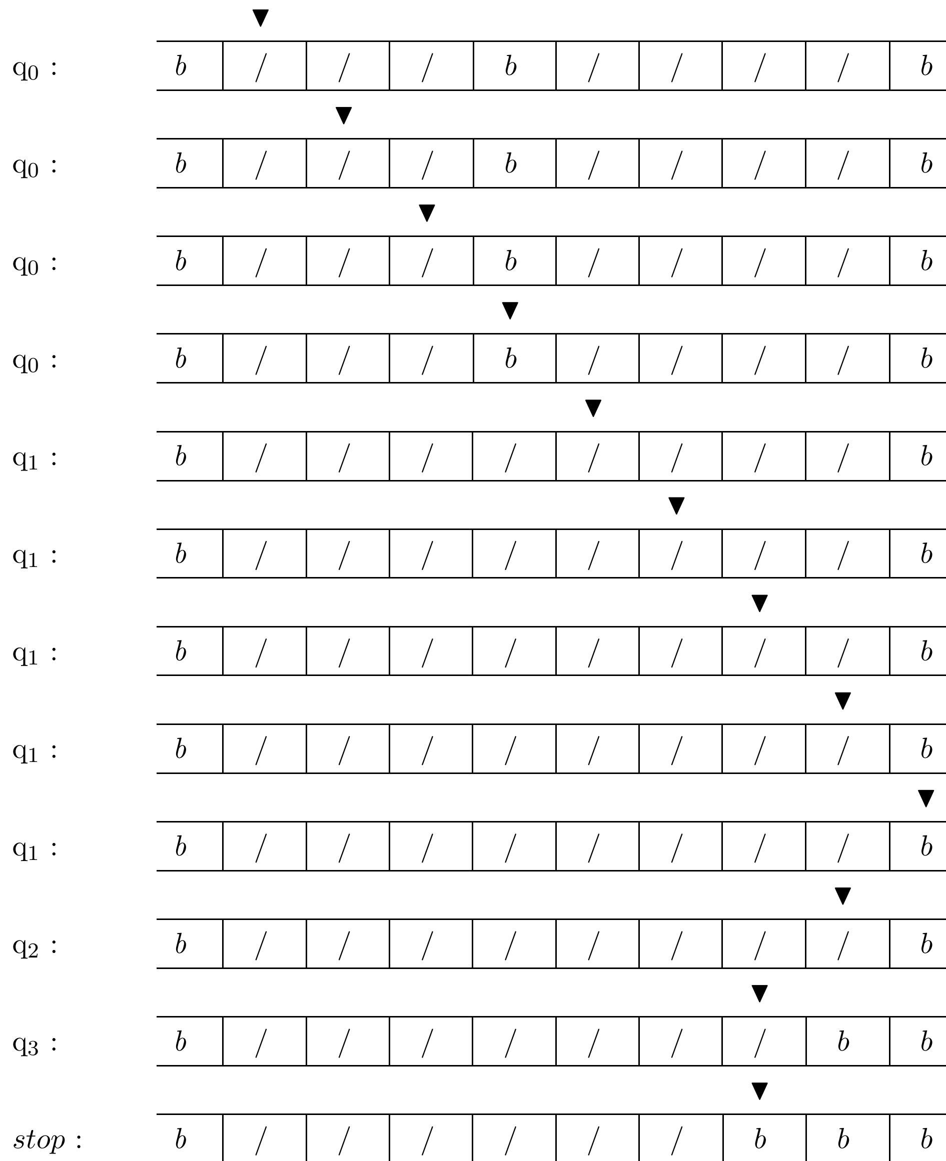


Figura 2.2: La computación de la máquina MT de la tabla 2.2 sobre la entrada 2+3.

Ejemplo 2.4

[Pap94]Una máquina de Turing que acepta solamente las cadenas en binario que son palíndromos es mostrada en la tabla 2.3. El procedimiento usado por esta máquina es recursivo. En el estado q_0, marca el primer símbolo de la entrada con b y mueve la cabeza a la derecha hasta el último símbolo de la entrada; si este símbolo es igual al primero entonces lo marca con b y regresa al estado q_0; en caso contrario, la entrada no es un palíndromo y la rechaza. Si la máquina continua, se ha obtenido una cadena con dos símbolos menos a la cual la máquina aplica el mismo procedimiento. $\diamondsuit$

q	s	$\delta(q, s)$	q	s	$\delta(q, s)$
q_0	0	(q_1, b, D)	q'_1	0	(q, b, I)
q_0	1	(q_2, b, D)	q'_1	1	$(q_N, 1, \text{N})$
q_0	b	(q_Y, b, N)	q'_1	b	(q_Y, b, N)
q_1	0	$(q_1, 0, \text{D})$	q'_2	0	$(q_N, 0, \text{N})$
q_1	1	$(q_1, 1, \text{D})$	q'_2	1	(q, b, I)
q_1	b	(q_1, b, I)	q'_2	b	(q_Y, b, N)
q_2	0	$(q_2, 0, \text{D})$	q	0	$(q, 0, \text{I})$
q_2	1	$(q_2, 1, \text{D})$	q	1	$(q, 1, \text{I})$
q_2	b	(q'_2, b, I)	q	b	(q_0, b, D)

Tabla 2.3: Máquina de Turing para palíndromos.

Ejemplo 2.5

La máquina de Turing mostrada en la tabla 2.4 produce computaciones infinitas. $\diamondsuit$

q	s	δ(q,s)
q_0	0	$(q_0, 0, \text{D})$
q_0	1	$(q_0, 1, \text{D})$
q_0	b	(q_0, b, I)

Tabla 2.4: Máquina de Turing que produce computaciones infinitas.

Definición 2.2 (Configuración). Sea MT una máquina de Turing. Una configuración de MT es una cadena xqy donde $x, y \in \Sigma^*$, y no es la cadena vacía y $q \in Q$. (Se denotará por ϵ la cadena vacía).

Una configuración contiene una descripción completa de la situación actual de la computación. Es decir, la interpretación de la configuración xqy es que MT está en estado q con la cadena xy sobre su cinta y con la cabeza de lectura-escritura examinando el primer símbolo de y.

Ejemplo 2.6
En la Figura 2.3 se muestran las configuraciones de cada paso de la máquina de Turing del ejemplo 2.1. $\diamondsuit$

Paso											Configuración
1	q_0:	$\cdots$	b	1	0	1	b	b	b	$\cdots$	C_0: $q_0\,101$
2	q_0:	$\cdots$	b	1	0	1	b	b	b	$\cdots$	C_1: $1\,q_0\,01$
3	q_0:	$\cdots$	b	1	0	1	b	b	b	$\cdots$	C_2: $10\,q_0\,1$
4	q_0:	$\cdots$	b	1	0	1	b	b	b	$\cdots$	C_3: $101\,q_0\,b$
5	q_1:	$\cdots$	b	1	0	1	0	b	b	$\cdots$	C_4: $1010\,q_1\,b$
6	$stop$:	$\cdots$	b	1	0	1	0	0	b	$\cdots$	C_5: $1010\,stop\,0$

Figura 2.3: Configuraciones de la Máquina de Turing del ejemplo 2.1 sobre la entrada 101.

Definición 2.3 (Configuración obtenida de otra). Sea MT una máquina de Turing dada y C, C' configuraciones. Se dice que C' se obtiene de C en un paso de MT y

se denota C' $\underset{\longrightarrow}{MT}$ C si C $= xqsw$, $\delta(q,\, s) = (q',\, s', \mathrm{m})$ y se cumple una de las siguientes condiciones:

- Si m = D y w es no vacío entonces C' $= xs'q'w$.
- Si m = D y w es vacío entonces C' $= xs'q'b$.
- Si m = I y $x = x'a$ para algún $a \in \Sigma$ entonces C' $= x'q'as'w$.
- Si m = I y w es vacío entonces C' $= q'bs'w$.
- Si m = N entonces C' $= xqs'w$.

Definición 2.4 (Computación). La computación de una máquina de Turing MT sobre una entrada $x \in (\Sigma - \{b\})^*$ es una secuencia única $C_0, C_1, C_2, \ldots$ (finita o infinita) de configuraciones tal que $C_0 = q_0\, x$ (configuración inicial) y $C_i \underset{\longrightarrow}{MT} C_{i+1}$ para cada i con C_{i+1} en la secuencia. Se dice que MT *acepta* x si y sólo si la computación es finita y la configuración final contiene el estado q_Y.

Ejemplo 2.7
La computación de la máquina de Turing mostrada en el ejemplo 2.1 sobre la entrada 101 de acuerdo a las configuraciones del ejemplo 2.6 es: C_0, C_1, C_2, C_3, C_4, C_5. $\diamondsuit$

Definición 2.5 (Lenguaje aceptado por una *MT*). Sea MT una máquina de Turing. El conjunto de entradas aceptadas por MT, denotado por $L(MT)$, es llamado el lenguaje aceptado por MT.

2.1.2. Máquinas de Turing k-cintas

La máquina de Turing k-cintas es una generalización de la máquina de Turing 1-cinta. Como su nombre lo indica, esta opera simultáneamente con k cintas y una cabeza de lectura-escritura para cada una de ellas.

Definición 2.6 (Máquina de Turing k-cintas). Una máquina de Turing k-cintas determinística MT con k ≥ 1 es una tupla $\langle Q, \Sigma, \delta \rangle$, donde:

1. Q y Σ son los mismos conjuntos que para una máquina de Turing 1-cinta.
2. δ es una función definida de $(Q - \{q_Y, q_N, stop\}) \times \Sigma^k$ en $Q \times (\Sigma \times \mathrm{M})^k$ donde $\mathrm{M} = \{\mathrm{I}, \mathrm{D}, \mathrm{N}\}$ es el conjunto de movimientos.

Intuitivamente, δ es una función que debe reflejar el comportamiento de las múltiples cintas. Esto es, $\delta(q, s_1, s_2, \ldots, s_k) = (q', s'_1, \mathrm{m}_1, s'_2, \mathrm{m}_2, \ldots, s'_k, \mathrm{m}_k)$ significa que: si MT está en estado q, la cabeza de lectura-escritura de la primera cinta está leyendo a s_1, la de la segunda a s_2, y así sucesivamente, entonces el próximo estado será q', la cabeza de la primera cinta escribirá s'_1 y se moverá en la dirección indicada por m_1 y así sucesivamente para las otras cintas.

La cadena de entrada de una máquina de Turing k-cintas es colocada desde el cuadrado 1 de la primera cinta. La máquina comienza su ejecución en estado q_0, todas las cintas excepto la primera contienen símbolos b y cada cabeza de lectura-escritura examina el primer cuadrado de su cinta.

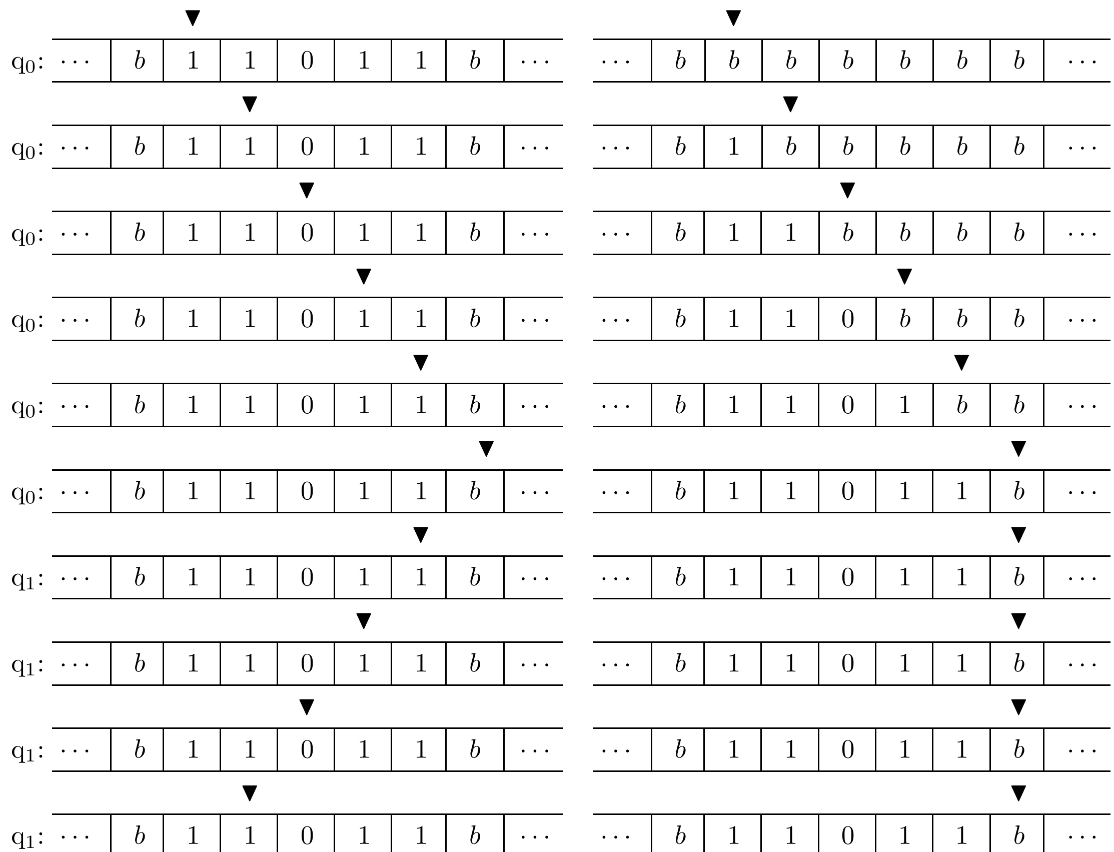
Ejemplo 2.8

En el ejemplo 2.4 se presentó una máquina de Turing 1-cinta para decidir palíndromos. Una forma más eficiente para hacer esto es usando una máquina de Turing 2-cintas. La tabla 2.5 muestra el comportamiento de esta máquina.

La figura 2.4 muestra la computación de la máquina sobre la entrada 11011. $\diamondsuit$

q	s_1	s_2	$\delta(q, s_1, s_2)$
q_0	0	b	$(q_0, 0, D, 0, D)$
q_0	1	b	$(q_0, 1, D, 1, D)$
q_0	b	b	(q_1, b, I, b, N)
q_1	0	b	$(q_1, 0, I, b, N)$
q_1	1	b	$(q_1, 1, I, b, N)$
q_1	b	b	(q_2, b, D, b, I)
q_2	0	0	$(q_2, 0, D, 0, I)$
q_2	1	1	$(q_2, 1, D, 1, I)$
q_2	0	1	$(q_N, 0, N, 1, N)$
q_2	1	0	$(q_N, 1, N, 0, N)$
q_2	b	b	(q_Y, b, N, b, N)

Tabla 2.5: Máquina de Turing 2-cintas para palíndromos.



			▼													▼	
q1:	···	b	1	1	0	1	1	b	···	···	b	1	1	0	1	1	b	···
		▼														▼	
q1:	···	b	1	1	0	1	1	b	···	···	b	1	1	0	1	1	b	···
			▼												▼		
q2:	···	b	1	1	0	1	1	b	···	···	b	1	1	0	1	1	b	···
				▼										▼			
q2:	···	b	1	1	0	1	1	b	···	···	b	1	1	0	1	1	b	···
					▼								▼				
q2:	···	b	1	1	0	1	1	b	···	···	b	1	1	0	1	1	b	···
						▼						▼					
q2:	···	b	1	1	0	1	1	b	···	···	b	1	1	0	1	1	b	···
							▼				▼						
q2:	···	b	1	1	0	1	1	b	···	···	b	1	1	0	1	1	b	···
								▼		▼							
q2:	···	b	1	1	0	1	1	b	···	···	b	1	1	0	1	1	b	···

Figura 2.4: Computación de la MT presentada en el ejemplo 2.5 sobre la entrada 11011.

Ejemplo 2.9

La tabla 2.6 muestra una máquina de Turing 2-cintas que acepta solo las cadenas en binario de la forma 0^n1^n con $n = 0, 1, 2...$; es decir, acepta el lenguaje:

$$L = \{x : x = 0^n1^n \,\text{con}\, n = 0, 1, ...\}$$

La máquina opera de la siguiente forma:

- Cuando lee el primer símbolo de la entrada y este es 0, escribe un cero sobre la cinta 2 para delimitar la secuencia de ceros leídos sobre la cinta 1 y pasa al estado q_1.
- Escribe un uno sobre la cinta 2 por cada cero leído sobre la cinta 1.
- Cuando el primer uno ha sido leído, se mueve hacia la derecha sobre la cinta 1 buscando otros unos y hacia la izquierda sobre la cinta 2 para chequear si un correspondiente cero ha sido encontrado y pasa a q_2.
- La cadena 0^01^0 denota la cadena vacía y pertenece a L.

q	s_1	s_2	$\delta(q, s_1, s_2)$
q_0	0	b	$(q_1, 0, D, 0, D)$
q_0	1	b	$(q_N, 1, N, b, N)$
q_0	b	b	(q_Y, b, N, b, N)
q_1	0	b	$(q_1, 0, D, 1, D)$
q_1	1	b	$(q_2, 1, D, b, I)$
q_1	b	b	(q_N, b, N, b, N)
q_2	0	0	$(q_N, 0, N, 0, N)$
q_2	0	1	$(q_N, 0, N, 1, N)$
q_2	1	1	$(q_2, 1, D, 1, I)$
q_2	1	0	$(q_N, 1, N, 0, N)$
q_2	b	0	$(q_Y, b, N, 0, N)$
q_2	b	1	$(q_N, b, N, 1, N)$

Tabla 2.6: Máquina de Turing 2-cintas que acepta el lenguaje L.

- Si la entrada comienza con 1, entonces la rechaza.
- La entrada de la forma $x = 0^h$, la rechaza.
- La entrada de la forma $x = 0^h1^k0$, la rechaza.
- Las entradas con más ceros que unos las rechaza y las entradas con más unos que ceros las rechaza. $\diamondsuit$

Finalmente, es importante mencionar el siguiente teorema que señala que cualquier máquina de Turing k-cintas puede ser simulada por una máquina de Turing 1-cinta con a lo mas una pérdida cuadrática de tiempo; lo cual deja ver el poder de las máquinas de Turing 1-cinta.

Teorema 2.1. Dada cualquier máquina de Turing k-cintas MT que opera en tiempo $f(n)$, se puede construir una máquina de Turing 1-cinta *MT'* que opera en tiempo $O(f^2(n))$ y tal que, para cualquier entrada x, la salida de *MT'* sobre x corresponde a la salida de MT sobre x. [Pap94]

2.2. MÁQUINAS DE TURING NO DETERMINÍSTICAS

En ésta sección se describirá un modelo de computación denominado Máquina de Turing no determinística, del cual se harán dos presentaciones: el primero es el modelo estándar, que es esencialmente una máquina de Turing que puede, en cada paso, seleccionar de un conjunto finito de posibilidades una acción o instrucción a ejecutar. El segundo, es el modelo no estándar, el cual funciona igual que una máquina de Turing sólo que su computación hace uso de dos etapas, una etapa de suposición y una de chequeo.

2.2.1. Modelo estándar

Definición 2.7 (Máquina de Turing no determinística). Una máquina de Turing no determinística de una cinta MTN es una tupla $\langle Q, \Sigma, \delta\rangle$ donde: Q y Σ son los mismos conjuntos que para una máquina de Turing determinística y δ es una relación definida de $(Q - \{q_Y, q_N, stop\}) \times \Sigma$ en $Q \times \Sigma \times M$, es decir

$$\delta \subseteq (Q - \{q_Y, q_N, stop\}) \times \Sigma \times Q \times \Sigma \times M.$$

δ refleja la principal diferencia entre una máquina determinística y una no determinística. En la primera, δ es una función mientras que en la segunda δ es una relación. Esto significa que para una situación particular de la máquina, no existe solo una instrucción a seguir sino que MNT puede "escoger" entre varias posibilidades, la instrucción a ejecutar.

Definición 2.8 (Computación no determinística). Una computación no determinística sobre una entrada x, $MNT(x)$, puede ser vista como un árbol, llamado *árbol*

de computación; los nodos corresponden a las situaciones de la máquina, las aristas entre los nodos corresponden a una aplicación de δ. Cada camino (finito o infinito) del árbol comenzando desde la raíz se denomina *camino de computación*. La figura 2.5 muestra los primeros 3 pasos de un ejemplo de un árbol de computación.

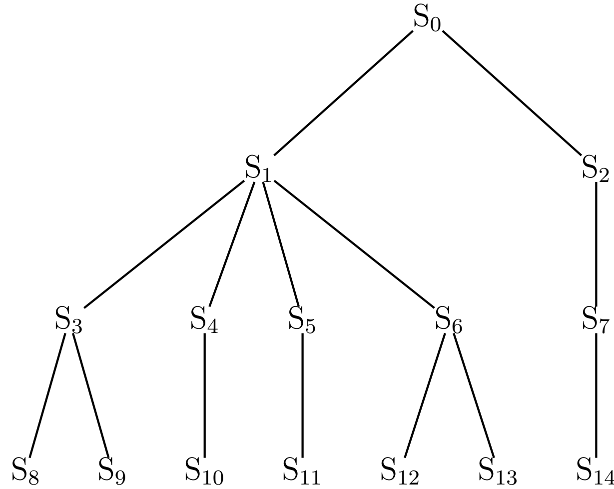


Figura 2.5: Los primeros 3 niveles de un árbol de computación.

Definición 2.9 (Grado de no determinismo). Sea MTN una máquina de Turing no determinística. Para cada $(q, s) \in Q - \{q_Y, q_N, stop\} \times \Sigma$, sea $C_{q,s} = \{(q', s', m) : (q, s, q', s', m) \in \delta\}$. Se define el grado de no determinismo de MNT como:

$$d = \text{máx}\,|C_{q,s}| \text{ para todo } (q, s) \in Q - \{q_Y, q_N, stop\} \times \Sigma.$$

Definición 2.10 (El lenguaje aceptado por *MTN*). Sea MTN una máquina de Turing no determinística. Una entrada *x es aceptada por MTN* si el árbol de computación asociado con $MTN(x)$ incluye por lo menos un *camino de computación de aceptación*, esto es, un camino de computación finito cuya hoja es una situación que contiene el estado q_Y. El conjunto de entradas aceptadas por MTN es llamado el *lenguaje aceptado por MTN* y se denota $L(MTN)$.

De la definición se observa que una cadena de entrada es rechazada solamente si ningún camino de computación es de aceptación.

2.2.2. Modelo no estándar

Una máquina de Turing no determinística 1-cinta MTN es especificada exactamente de la misma forma que una máquina de Turing determinística excepto que es aumentada con un módulo de suposición que tiene su propia cabeza de escritura solamente. La máquina MTN incluye un alfabeto Σ con su símbolo blanco, un conjunto de estados Q con el estado inicial q_0 y los estados de parada q_Y, q_N, *stop* y una función de transición δ de $Q - \{q_Y, q_N, stop\} \times \Sigma$ en $Q \times \Sigma \times M$. La *computación* de una MTN sobre una cadena de entrada $x \in (\Sigma - \{b\})^*$ difiere de una MT en que toma lugar en dos etapas distintas, una etapa de suposición y una etapa de chequeo.

Inicialmente, la cadena de entrada x es escrita en el cuadrado 1 de la cinta hasta el cuadrado $|x|$ (todos los demás cuadrados contienen el símbolo blanco), la cabeza de lectura-escritura está leyendo el cuadrado 1 de la cinta, la cabeza de escritura está apuntando al cuadrado -1 y la máquina no se encuentra en ningún estado. La primera etapa que se ejecuta es la etapa de suposición; en cada paso, el módulo de suposición opera dirigiendo la cabeza de escritura un cuadrado a la izquierda y escribiendo algún símbolo de Σ en el cuadrado de la cinta que está apuntando. Cuando para el módulo de suposición queda inactivo. La decisión de si queda activo o no, es tomada por el módulo de suposición en una manera totalmente arbitraria, al igual que la decisión de qué símbolo de Σ escribir en cada paso.

Inmediatamente la etapa de suposición ha terminado (el módulo de suposición queda inactivo), la máquina toma el estado q_0, y comienza la etapa de chequeo. Desde este momento, la computación procede de acuerdo a exactamente el mismo procedimiento de una máquina de Turing determinística. El módulo de suposición y su cabeza de escritura no se involucran nunca mas en la computación habiendo ya ejecutado su tarea: escribir la cadena supuesta sobre la cinta. La computación termina cuando la máquina alcanza uno de los estados (q_Y, q_N, *stop*). Una *computación* es de *aceptación* si la máquina para en estado q_Y.

Es importante observar que cualquier máquina MTN tiene un número infinito de posibles computaciones para una cadena de entrada x dada, una por cada posible cadena supuesta. Se dice que la máquina *acepta* x si por lo menos una de estas es una computación de aceptación. Por el contrario, si ninguna de las computaciones posibles es de aceptación, la máquina rechaza la entrada. El *lenguaje aceptado por* MTN, L(MTN), es el conjunto de todas las cadenas de entrada aceptadas por la máquina.

Ejemplo 2.10
Una máquina de Turing no determinística que resuelve el problema del Agente Viajero, presentado en el ejemplo 1.31 (pág. 46), se puede construir de la siguiente manera: Inicialmente la cinta contiene la representación de una instancia del problema (un conjunto de ciudades, la distancia entre ellas y un entero positivo B). En la etapa de suposición la cabeza de escritura escribe una cadena arbitraria de símbolos, de longitud no mas grande que su entrada. Cuando acaba de escribir, la cabeza de lectura-escritura se dirige al primer símbolo supuesto y chequea si la cadena (supuesta) corresponde a la representación de una permutación de ciudades, y si es así, chequea si la permutación es un tour de longitud menor o igual que B. Si la cadena supuesta en realidad codifica un tour de costo menor o igual que B, la máquina acepta la entrada.

Verdaderamente, esta máquina decide el problema del agente viajero: acepta la entrada si, y sólo si, ésta codifica una instancia "si"[*] del problema. Además si la entrada es una instancia "si", existirá una computación de ésta máquina que "suponga" precisamente la permutación de las ciudades de costo menor o igual que B, aunque pueden existir computaciones que supongan tours muy costosos o cadenas sin sentido. $\diamondsuit$

Un aspecto para destacar de las máquina no determinísticas y que corroboran la importancia de las máquinas de Turing determinísticas (1-cinta) se muestra en el siguiente teorema.

[*] Si la entrada es una instancia "si", esto significa que existe un tour de ciudades de longitud menor o igual que B.

Teorema 2.2. Dada una máquina de Turing no determinística MTN, siempre es posible derivar máquina de Turing determinística MT equivalente. Además, existe una constante c tal que, para cualquier entrada x, si MTN acepta x en t pasos, MT aceptará x en a lo más c^t pasos. [BC94]

2.3. MÁQUINA DE TURING UNIVERSAL

Una máquina de Turing puede ser vista como un programa fijo en el que se implementa un algoritmo específico, capaz solamente de aplicar ese algoritmo a distintos valores de la entrada. Por otra parte, los computadores actuales, son máquinas programables capaces de ejecutar cualquier programa sobre cualquier conjunto de entradas. En esta sección, se presentará una máquina de Turing que se asemeja de alguna forma a los computadores actuales, esto es, una máquina de Turing en cierto sentido programable; denominada Máquina de Turing Universal.

Una máquina de Turing Universal U tiene la función de simular el comportamiento de cualquier máquina de Turing MT sobre una entrada x. Como cualquier máquina de Turing, U consiste de un alfabeto, un conjunto de estados y una función de transición. Una entrada de U contiene una descripción de la máquina MT a ser simulada seguida de una entrada para MT.

Por conveniencia, para describir una máquina de Turing Universal, se asumirá que para cualquier máquina de Turing $MT = \langle Q, \Sigma, \delta \rangle$, Q y Σ se representarán como $\Sigma = \{1, 2, \dots |\Sigma|\}$ y $Q = \{|\Sigma| + 1, |\Sigma| + 2, \dots, |\Sigma| + |Q|\}$. El estado inicial q_0 siempre será $|\Sigma| + 1$, los estados de parada q_Y, q_N y *stop* serán $|\Sigma| + 2$, $|\Sigma| + 3$ y $|\Sigma| + 4$ respectivamente y los movimientos de la cabeza I, D, N se codificarán por los números $|\Sigma| + |Q| + 1$, $|\Sigma| + |Q| + 2$, $|\Sigma| + |Q| + 3$ respectivamente. Todos los números serán representados en binario, reservando para cada uno de ellos $|\Sigma| + |Q| + 3$ cuadrados

de la cinta, colocando ceros en los cuadrados sobrantes en cada caso, con el fin de que todos queden con la misma longitud.

Definición 2.11 (Máquina de Turing Universal). Sea MT=$\langle Q, \Sigma, \delta \rangle$ una máquina de Turing y $x \in \{\Sigma - b\}^*$ una entrada para MT. Una máquina de Turing universal U= $\langle Q_U, \Sigma_U, \delta_U \rangle$ es una máquina de Turing que toma como entrada a MT y x y simula MT sobre x. En una máquina de Turing universal no se puede predeterminar el número de estados y símbolos que usará (puesto que ella debe simular cualquier máquina de Turing). La simulación de U es mejor descrita asumiendo que tiene dos cintas. En la primera cinta, se describe la entrada; la descripción de MT puede realizarse comenzando con el número $|Q|$ y seguido por $|\Sigma|$ ambos en binario(éstos dos números son suficientes para describir a Q y Σ por la definición de ellos dada), seguido por una descripción de δ. La descripción de δ será una secuencia de pares de la forma $((q, s), (q', s', m))$ donde los símbolos "(", ")", "," se asume que están en Σ_U. La descripción de MT es seguida por ";" y continúa la representación de x, los símbolos de x son también codificados en binario separados por ",". La máquina U usa su segunda cinta para almacenar la configuración actual de MT. Una configuración wqu es almacenada precisamente como w, q, u, esto es, con la codificación de la cadena w seguida por una "," la cual es seguida por la codificación del estado q y luego otra "," y la codificación de u. La segunda cinta inicialmente contiene la codificación de q_0 y en seguida la codificación de x.

Para simular un paso de MT, U simplemente examina su segunda cinta, hasta encontrar la representación binaria de un entero correspondiente a un estado (esto es, un entero menor o igual que $|Q|$). Luego, busca en la primera cinta una regla de δ que contenga este estado (que corresponde al estado actual). Si tal regla es localizada, la cabeza de lectura-escritura de la segunda cinta se mueve hacia la izquierda para comparar el símbolo leído con el de la regla. Si no son iguales, continúa este proceso examinando otras reglas. Si son iguales, la regla es implementada, lo cual consiste en cambiar el estado actual, el símbolo actual y en mover el estado hacia la izquierda o hacia la derecha en la segunda cinta (el hecho de que los estados y los símbolos son

codificados como enteros binarios con la misma longitud resulta útil aquí). Cuando MT para también lo hace U. Esto completa la descripción de la operación de la máquina universal U sobre la entrada MT, x.

Si la entrada para la máquina U no es una descripción válida de una máquina de Turing junto con su entrada, U se moverá a la derecha por siempre.

2.4. EL PROBLEMA DE LA PARADA

El problema de la parada fué replanteado por Alan Turing de la siguiente forma: ¿Es posible construir una máquina de Turing capaz de responder si una máquina de Turing se detendrá o no sobre una cadena de entrada dada?. Este problema también se puede expresar en los siguientes términos: Sean MT una máquina de Turing y x una entrada para MT. Existe una máquina de Turing P, tal que:

$$P(MT, x) = \begin{cases} 1 & \text{si } MT \text{ se detiene con } x \\ 0 & \text{si } MT \text{ no se detiene con } x. \end{cases}$$

donde $P(MT,\ x)$ significa que la máquina P recibe como entrada la máquina de Turing MT y la cadena x.

Se observa fácilmente que para algunas máquinas de Turing y para algunas de sus entradas es posible determinar si la máquina se detendrá o no. Por ejemplo, para la máquina MT construída en el ejemplo 2.5 (pág. 60) y cualquier cadena de entrada, la máquina no se detiene. Por el contrario, la máquina MT construída en el ejemplo 2.3 (pág. 58) y para una cadena de entrada constituida por dos números naturales, sí se detiene. Pero estas, son soluciones al problema de la parada para un caso particular, y lo que interesa es la solución al problema de la parada para su caso general, es decir: ¿es posible construir una máquina de Turing que resuelva el problema de la parada,

para cualquier máquina de Turing y para cualquier cadena de entrada?

Teorema 2.3. [GS01] Sea MT una máquina de Turing y x una entrada para MT, entonces no existe una máquina de Turing P, tal que:

$$P(MT, x) = \begin{cases} 1 & \text{si } MT \text{ se detiene con } x \\ 0 & \text{si } MT \text{ no se detiene con } x. \end{cases}$$

Demostración. (Por contradicción)
Sea MT una máquina de Turing cualquiera y x una entrada para MT. Supóngase que existe P tal que:

$$P(MT, x) = \begin{cases} 1 & \text{si } MT \text{ se detiene con } x \\ 0 & \text{si } MT \text{ no se detiene con } x. \end{cases}$$

Una nueva máquina de Turing H, puede ser definida de la siguiente manera:

$$H(x) = \begin{cases} 1 & \text{si } P(MT; x) = 0 \\ \text{no se detiene} & \text{si } P(MT; x) = 1 \end{cases}$$

Sea β una entrada para H. Se invocará H con la entrada β. Se pueden presentar 2 casos: $H(\beta) = 1$ o $H(\beta)$ no se detiene.

Si $H(\beta) = 1$, H se detiene con β. Como H es una máquina de Turing, se puede invocar P con la entrada $H; \beta$. Como H se detiene con β entonces por la definición de P, $P(H; \beta) = 1$ y entonces por la definición de H tenemos que $H(\beta)$ no se detiene; lo cual es una contradicción.

Ahora, supóngase que $H(\beta)$ no se detiene. Al igual que en el caso anterior, se invocará P con la entrada $H; \beta$. Como H no se detiene con β entonces por la definición de P, $P(H; \beta) = 0$ y entonces por la definición de H tenemos que $H(\beta)$ se detiene; lo cual es una contradicción.

2.5. MÁQUINAS DE ACCESO ALEATORIO (RAM)

Ya se ha destacado la potencia de las máquinas de Turing cuando se mencionó que las máquinas de Turing multicintas pueden ser simuladas por una máquina de Turing de 1-cinta con solo una pérdida cuadrática de tiempo. En esta sección se presentará otro modelo de computación llamado una máquina de acceso aleatorio (RAM). Como su nombre sugiere, las máquinas RAM hacen uso de una memoria de acceso aleatorio, al contrario de las máquinas de Turing que usan como una componente de memoria una cinta de acceso secuencial. Sin embargo, también estos dos modelos son equivalentes, lo cual hace notar nuevamente la potencia de las máquinas de Turing.

Una máquina RAM consiste en un programa actuando sobre una estructura de datos. Una estructura de datos RAM es un arreglo de registros de memoria cada uno capaz de contener un entero arbitrariamente grande, posiblemente negativo. El programa es un conjunto de instrucciones que se asemeja al conjunto de instrucciones de los computadores actuales (ver Tabla 2.7).

Definición 2.12 (Máquina RAM). Formalmente, una máquina RAM $\Pi = \langle \pi_1, \pi_2, \ldots, \pi_m \rangle$ es una secuencia finita de instrucciones, donde cada instrucción π_i, $i = 1, \ldots, m$ corresponde a una de las mostradas en la Tabla 2.7. Como ya se ha mencionado Π utiliza un arreglo de registros de memoria; en cada paso de una computación RAM, el registro de memoria i, $i \geq 0$, contiene un entero, posiblemente negativo, denotado r_i, y una instrucción π_k es ejecutada; k es el "contador del programa". Inicialmente, todos los registros de memoria son inicializados con ceros y la primera instrucción se ejecuta. La ejecución de una instrucción puede producir un cambio en el contenido de uno de los registros de memoria y un cambio en k.

La entrada de una máquina RAM es una secuencia finita de enteros, contenidos en

un arreglo finito de registros de entrada. Cualquier entero en la entrada puede ser transferido a los registros de memoria.

INSTRUCCION	**OPERANDO**	**SIGNIFICADO**
READ	j	$r_0 := i_j$ (lee el $j-$ésimo elemento de la entrada y lo escribe en el Registro 0)
READ	$\uparrow j$	$r_0 := i_{r_j}$ (lee el elemento de la entrada cuyo índice es el valor contenido en el Registro j y lo escribe en el Registro 0)
STORE	j	$r_j := r_0$ (almacena en el Registro j el valor contenido en el Registro 0)
STORE	$\uparrow j$	$r_{r_j} := r_0$ (almacena en el Registro cuyo índice es el valor contenido en el Registro j, el contenido del Registro 0)
LOAD	x	$r_0 := x$
ADD	x	$r_0 := r_0 + x$
SUB	x	$r_0 := r_0 - x$
HALF		$r_0 := \lfloor r_0/2 \rfloor$
JUMP	j	$k := j$ (hace que la máquina "salte"a la instrucción j)
JPOS	j	Si $r_0 > 0$ entonces $k := j$
JCERO	j	Si $r_0 = 0$ entonces $k := j$
JNEG	j	Si $r_0 < 0$ entonces $k := j$
HALT		$k := 0$ (la máquina para)

Tabla 2.7: Instrucciones RAM y sus significados.

En la Tabla 2.7 se muestran las instrucciones de una máquina RAM junto con los tres tipos de operandos que ellas usan. El operando j es un entero, r_j es el contenido actual del Registro de memoria j, i_j es el $j-$ésimo elemento de la entrada y x representa un operando de la forma j, "$\uparrow j$", o "$= j$". Si x es el entero j entonces x representa el contenido del Registro de memoria j; si x es "$\uparrow j$", entonces x representa el contenido del registro cuyo índice es el valor contenido en el Registro j; y si x es "$= j$", esto significa que x representa al mismo entero j. En otras palabras, el valor del operando j

es r_j, el de "$\uparrow j$" es r_{r_j} y el de "$= j$" es j. El valor así identificado es entonces usado en la ejecución de la instrucción. Todas las instrucciones cambian k a $k+1$ a menos que en la instrucción se declare otra cosa (k es el contador del programa) tal como ocurre en las últimas cinco. Note la función especial del Registro 0: es el acumulador, donde se realizan todas las operaciones aritméticas y lógicas. Finalmente, la instrucción HALT para la computación. Toda instrucción semánticamente incorrecta también se puede considerar como una instrucción HALT.

Ejemplo 2.11

[Pap94] La Figura 2.6 muestra una máquina RAM que multiplica dos enteros (MPLY). El método utilizado por esta máquina es la multiplicación en binario; para esto se usa la instrucción HALF. Los dos enteros que son la entrada a la máquina se colocan en los registros de entrada i_1 e i_2. El programa repite $\lceil i_2 \rceil$ veces las instrucciones 5 a 19. Al comienzo de la n-ésima iteración (el programa comienza en la iteración 0), el registro 3 contiene $\lfloor i_2/2^n \rfloor$, el registro 5 contiene $i_1\, 2^n$, y el registro 4 contiene $i_1\,(i_2\, mod\, 2^n)$. Al final de la iteración el programa verifica si el registro 3 contiene 0. Si es así, el programa termina con la salida contenida en el Registro 4, sino continúa con la próxima iteración.

1. READ 1
2. STORE 1 (El registro 1 contiene i_1, durante la n-ésima iteración
3. STORE 5 el registro 5 contiene $i_1 2^n$. Actualmente n=0)
4. READ 2
5. STORE 2 (El registro 2 contiene i_2)
6. HALF (n se incrementa y la n-ésima iteración comienza)
7. STORE 3 (El registro 3 contiene $\lfloor i_2/2^n \rfloor$)
8. ADD 3
9. SUB 2
10. JCERO 14
11. LOAD 4 (suma el contenido del Registro 5 al del Registro 4
12. ADD 5 sólo si el residuo de dividir por 2 es distinto de 0)
13. STORE 4 (El registro 4 contiene $i_1 \cdot (i_2\, mod\, 2^n)$)
14. LOAD 5
15. ADD 5
16. STORE 5

17. LOAD 3
18. JCERO 20 (Si $\lfloor i_2/2^n \rfloor$ =0 entonces el proceso termina)
19. JUMP 5 (sino, el proceso se repite)
20. LOAD 4 (Este es el resultado)
21. HALT

Figura 2.6: Máquina RAM para multiplicar dos enteros (MPLY)

La Tabla 2.8 muestra la computación de la máquina RAM para la multiplicación sobre la entrada 13×5.

Registros de entrada:

i_1	i_2
13	5

Iteración	Contador	Registros de memoria					
(n)	(k)	r_0	r_1	r_2	r_3	r_4	r_5
0	1	13					
	2		13				
	3						13
	4	5					
1	5			5			
	6	2					
	7				2		
	8	4					
	9	-1					
	11	0					
	12	13					
	13					13	
	14	13					
	15	26					
	16						26
	17	2					
	7				1		
	8	2					
	9	0					
	14	26					
	15	52					
2	5			2			
	6	1					
	7				1		
	8	2					
	9	0					
	14	26					
	15	52					
	16						
	17	1					
3	5			1			
	6	0					
	7				0		
	8	0					
	9	-1					
	11	13					
	12	65					
	13					65	
	14	52					
	15	104					
	16						104
	17	0					
	20	65					

Tabla 2.8: Una computación del programa MPLY sobre la entrada 13×5.

El programa MPLY puede ser usado como la instrucción MPLY x en cualquier otra máquina RAM, donde x puede ser cualquiera de los tres operandos j,"$\uparrow j$"o "$= j$". $\diamondsuit$

Ejemplo 2.12

Una máquina RAM que calcula el producto punto entre dos vectores, se muestra en la Figura 2.7.

1. READ 1	14. LOAD 5
2. STORE 1	15. ADD =1
3. ADD =2	16. STORE 5
4. STORE 4	17. LOAD 1
5. LOAD =2	18. ADD =2
6. STORE 5	19. SUB 5
7. READ ↑ 5	20. JCERO 26
8. STORE 2	21. LOAD 4
9. READ ↑ 4	22. ADD =1
10. STORE 3	23. STORE 4
11. MPLY 2	24. JUMP 7
12. ADD 6	25. LOAD 6
13. STORE 6	26. HALT

Figura 2.7: Máquina RAM que calcula el producto punto entre dos vectores.

Esta máquina utiliza $2n + 1$ registros de entrada, donde n es la longitud de los vectores. El registro de entrada 1 contiene n y en los registros restantes se colocan las componentes del primer vector seguidas de las componentes del segundo vector. El programa utiliza la definición usual de producto punto entre vectores de $\mathbb{R}^n$ y hace uso de 6 registros de memoria. Finalmente, la salida queda contenida en el registro 6. $\diamondsuit$

3. NP-COMPLETITUD

A pesar de los esfuerzos realizados por muchos expertos en matemáticas y otras áreas, existen muchos problemas para los cuales no se ha podido encontrar un algoritmo que los resuelva eficientemente. En este contexto por algoritmo "más eficiente" normalmente se entiende el más rápido. El requisito del tiempo es a menudo un factor dominante para determinar si un algoritmo particular es o no bastante eficiente para ser usado en la práctica. Sin embargo, la teoría de NP-completitud no está encaminada a tratar de encontrar tales algoritmos sino a proveer técnicas para probar que un problema dado es "difícil de resolver", es decir que probablemente no exista un algoritmo eficiente para resolverlo; así que es conveniente buscar otras alternativas para tratar de enfrentar el problema. Estas alternativas podrían ser, por ejemplo, buscar algoritmos rápidos que retornen soluciones aproximadas al problema o también, algoritmos que resuelvan algunos casos especiales del problema general en forma eficiente.

Por conveniencia, la teoría de NP-completitud está diseñada para ser aplicada sólo a problemas de decisión. Tales problemas, como se mencionó en el capítulo 1, tienen sólo dos posibles soluciones: la respuesta "sí" o la respuesta "no".

En este capítulo se presentan las nociones básicas de la teoría de NP-completitud describiendo cada una de las tres clases más importantes de problemas de decisión: la clase P, la clase NP y la clase NPC. La máquina de Turing 1-cinta introducida

en el capítulo anterior es tomada como el modelo de computación básico y es usada para definir la clase P; la clase NP se define usando la versión no determinística de la máquina de Turing. Después, se estudia la relación entre P y NP y se define lo que es una transformación polinomial de un lenguaje a otro, la cual será usada para definir la clase más importante, la clase de los problemas NP-completos. Finalmente, se presenta el teorema de Cook, el cual demuestra que SAT es NP-completo. La importancia de este resultado radica en que proporciona el primer problema NP-completo.

3.1. LA CLASE P

Antes de entrar a definir la primera clase de complejidad, es necesario primero especificar algunos conceptos, uno de ellos es el de eficiencia. Este concepto involucra entre otras cosas, el tiempo empleado por un algoritmo para ejecutarse.

Definición 3.1 (Longitud de la entrada). La longitud de la entrada para una instancia I de un problema Π está definida como el número de símbolos en la descripción de I, obtenida desde el esquema de codificación para Π. Si x es una entrada, su longitud se denotará por $|x|$.

Definición 3.2 (Tiempo requerido por una computación de una MT). El tiempo requerido en la computación de una máquina de Turing MT sobre una entrada x es el número de pasos que ocurren en la computación hasta alcanzar un estado de parada.

La función tiempo de complejidad para un algoritmo recibe como parámetro la longitud de la entrada y devuelve el tiempo requerido por el algoritmo para ejecutarse en el peor caso; es decir, el tiempo máximo que necesita el algoritmo para resolver cualquier

instancia del problema de determinado tamaño. Formalmente la función tiempo de complejidad se define como sigue:

Definición 3.3 (Función tiempo de complejidad). Sea MT una máquina de Turing y Σ su alfabeto de entrada. Si MT se detiene para todas las entradas $x \in \Sigma^*$, su función de complejidad $T_{MT} : \mathbb{Z}^+ \to \mathbb{Z}^+$, está dada por:

$$T_{MT}(n) = \text{máx} \left\{ \begin{array}{l} m\text{: existe un } x \in \Sigma^*\text{, con } |x| = n\text{, tal que el tiempo requerido} \\ \text{en la computación de } MT \text{ sobre } x \text{ es } m. \end{array} \right\}$$

Definición 3.4 (Algoritmo tiempo polinomial). Un algoritmo es tiempo polinomial si su función tiempo de complejidad es $O(p(n))$ para algún polinomio p, donde n denota la longitud de la entrada.
Formalmente, una *máquina de Turing MT es tiempo polinomial* si existe un polinomio p tal que $T_{MT} \in O(p(n))$ para todo $n \in \mathbb{Z}^+$.

Cualquier algoritmo cuya función de complejidad no pueda acotarse mediante un polinomio es llamado usualmente *algoritmo tiempo exponencial*, se debe notar que esta definición incluye ciertas funciones de complejidad no polinomiales tales como $n^{\log n}$ o n^n.

Definición 3.5 (Problema soluble por una máquina de Turing). Una máquina de Turing MT resuelve un problema de decisión Π bajo un esquema de codificación e si:

1. MT se detiene para todas las cadenas de entrada sobre su alfabeto.

2. $\text{L}(MT) = \text{L}[\Pi, e]$.

Definición 3.6 (Clase P). La clase de complejidad P se define como el siguiente conjunto:

$$P = \left\{ \begin{array}{l} \text{L: L es un lenguaje y existe una máquina de Turing determinística } MT \\ \text{que corre en tiempo polinomial y tal que L} = \text{L}(MT). \end{array} \right\}$$

Esto es, P es el conjunto de todos los lenguajes que son aceptados por alguna máquina de Turing determinística tiempo polinomial. Por otra parte, se dirá que un problema de decisión Π pertenece a P bajo el esquema de codificación e si $\text{L}[\Pi, e] \in \text{P}$.

La correspondencia entre problemas de decisión y lenguajes se origina en los esquemas de codificación que se usan para especificar las instancias del problema. Como se mencionó en el Capítulo 1 (sección 1.5), un esquema de codificación e para un problema Π provee una forma de describir cada instancia de Π mediante una cadena apropiada de símbolos sobre un alfateto Σ; de esta manera se obtuvo el lenguaje asociado con Π y e, $\text{L}[\Pi, e]$.

No se puede hablar de resolver un problema sin especificar primero una codificación para sus instancias aunque en realidad las instancias de un problema pueden ser representadas de muchas formas distintas. Sin embargo, no todas las codificaciones son convenientes, los siguientes ejemplos muestran la diferencia que puede haber entre los distintos esquemas de codificación que se usan para codificar un problema dado.

Ejemplo 3.1
[GJ79] Si se considera un problema en el cual cada instancia es un grafo $G = (V, E)$, donde V es el conjunto de vértices y E es el conjunto de aristas, tal instancia podría ser descrita (ver Tabla 3.1) simplemente con un listado de todos los vértices y aristas, o con un listado de filas de la matriz de adyacencia para el grafo, o listando para cada vértice todos los otros vértices que comparten una arista común con él (lista de adyacencia). Cada uno de estos esquemas puede dar una longitud de entrada diferente para el mismo grafo. Sin embargo, se puede observar que las longitudes de entrada que ellos determinan difieren a lo más polinomialmente unas de otras, de modo que algún algoritmo tiempo polinomial bajo uno de estos esquemas de codificación también

será tiempo polinomial bajo todos los otros. $\diamondsuit$

ESQUEMA DE CODIFICACIÓN	CADENA	LONGITUD
Lista de vértices y aristas	$V[1]V[2]V[3]V[4](V[1]V[2])(V[2]V[3])$	36
Lista de adyacencia	(V[2])(V[1]V[3])(V[1])( )	24
Filas de la matriz de adyacencia	0100/1010/0100/0000	19

Tabla 3.1: Descripciones del grafo $G = (V, E)$ donde $V = \{v_1, v_2, v_3, v_4\}$ y $E = \{\{v_1, v_2\}, \{v_2, v_3\}\}$ bajo tres esquemas de codificación diferentes.

Por otra parte, existen esquemas de codificación que determinan, para una misma instancia, longitudes de entrada exponencialmente más grandes que las determinadas por otros esquemas, como es el caso de la representación de enteros en unario, cuya longitud es exponencial respecto a la obtenida con la representación en binario, como lo muestra el ejemplo siguiente.

Ejemplo 3.2

[CLR90] Supóngase que un algoritmo recibe como entrada un entero k y que el tiempo de ejecución del algoritmo es $\theta(k)$. Si el entero k es codificado en unario (una cadena de k-unos) entonces el tiempo de ejecución del algoritmo es $O(n)$ donde n es la longitud de la entrada (aquí $n = k$), luego el algoritmo es tiempo polinomial. Sin embargo, si se usa la representación binaria del entero k entonces la longitud de la entrada es $n = \lceil \log k \rceil$. En este caso, el tiempo de ejecución del algoritmo es $\theta(k) = \theta(2^n)$, el cual es exponencial en el tamaño de la entrada. Así, dependiendo del esquema de codificación, el algoritmo corre en tiempo polinomial o exponencial. $\diamondsuit$

Sin embargo, no es de interés desarrollar algoritmos eficientes que resuelvan problemas bajo un esquema de codificación fijo si la representación de tales problemas bajo este esquema es demasiado grande (esto sería muy costoso computacionalmente). Dichos esquemas, por consiguiente, deben ser descartados. Se puede probar que, en la práctica, los alfabetos unarios, esto es, los alfabetos consistentes de exactamente un símbolo, son los únicos que deben ser descartados. El uso de tales esquemas podría hacer que

la complejidad del problema sea ocultada en la representación del mismo, como lo muestra el ejemplo anterior y el que a continuación se presenta.

Ejemplo 3.3
EL PROBLEMA DEL CAMINO MÁS CORTO

INSTANCIA:	Un grafo $G = (V, E)$, dos nodos $n_1, n_2 \in V$ y un número natural k
PREGUNTA:	¿ Existe un camino entre n_1 y n_2 que tenga una longitud a lo más k ?.

Si una instancia de tal problema es representada listando todos los caminos en el grafo G, entonces es fácil derivar un algoritmo eficiente que lo resuelva; solamente se debe determinar si el camino conecta n_1 y n_2 y chequear si su longitud es mayor o igual que k. Por otra parte, si el grafo G es representado en una forma más natural, por ejemplo por su matriz de adyacencia, ningún algoritmo tiempo polinomial que resuelva el problema del camino mas corto es conocido. $\diamondsuit$

Los ejemplos anteriores muestran que se debe ser cuidadoso al seleccionar la codificación de un problema de decisión; dicho esquema debe ser "razonable". Aunque es difícil definir formalmente la noción de "esquema de codificación razonable", este siempre debe satisfacer las siguientes condiciones:

1. La codificación de una instancia I debe ser concisa y no contener información o símbolos innecesarios, y
2. Los números incluidos en I son representados en base k, con $k > 1$, los conjuntos son representados como secuencias de elementos delimitados por símbolos apropiados.

Si se hace uso de tales esquemas de codificación estándar, la complejidad del problema no dependerá del esquema usado y además se puede cambiar de un esquema a otro rápidamente. En conclusión, se puede evitar especificar el esquema de codificación

para un problema asumiendo implícitamente que éste es razonable, como se hará en adelante.

Por otro lado, desde el punto de vista de los modelos de computación, la definición de la clase P puede ser reformulada utilizando otro modelo tal como la máquina de Turing multi-cintas o la máquina RAM, debido a que estos son equivalentes con respecto al tiempo de complejidad polinomial, como se mencionó en el capítulo anterior. Por esto y dado que los modelos computacionales formalizan la noción de algoritmo, se puede reemplazar el término "máquina de Turing tiempo polinomial " por "algoritmo tiempo polinomial".

Todo lo anterior, permite también considerar la clase P como el conjunto de todos los problemas solubles por un algoritmo tiempo polinomial.

Ejemplo 3.4
[AKS02] El problema de distinguir números primos de números compuestos es uno de los más importantes y útiles que se ha conocido en Aritmética. Tres científicos Hindúes anunciaron recientemente haber resuelto este problema que por varias décadas ha desafiado a investigadores de todo el mundo: decidir primalidad eficientemente. Ellos desarrollaron un algoritmo determinístico tiempo polinomial para determinar si un número entero n es primo o compuesto; así, el problema de decidir primalidad está en P. Los autores del descubrimiento son Manindra Agrawal, Neeraj Kayal y Nitin Saxena del Indian Institute of Technology de Kanpur. $\diamondsuit$

Ejemplo 3.5
[CLR90] El problema del CAMINO MÁS CORTO está en P. Un algoritmo determinístico tiempo polinomial para él computa el camino más corto de n_1 a n_2 en G usando búsqueda primero en anchura, y luego compara la distancia obtenida con k; si la distancia es a lo más k, el algoritmo responde "si" y para, en otro caso, el algoritmo no para. $\diamondsuit$

3.2. LA CLASE NP

En ésta sección se presenta otra clase de complejidad, denominada la clase NP [*]. A esta clase pertenecen una gran cantidad de problemas que hasta el momento no se han podido resolver determinísticamente en tiempo polinomial. Primero, se hará una presentación formal de la clase NP mediante máquinas de Turing no determinísticas y luego, haciendo uso de algoritmos de verificación, se mostrará otra caracterización de esta misma clase.

3.2.1. Máquinas de Turing no determinísticas y la clase NP

Definición 3.7 (Tiempo requerido en una computación por una $\boldsymbol{MTN}$). El tiempo requerido por una máquina de Turing no-determinística MTN para aceptar la cadena $x \in \mathrm{L}(MTN)$ se define como el mínimo número de pasos que ocurren en las etapas de suposición y chequeo, sobre todas las computaciones de aceptación de MTN sobre x, hasta que para en estado q_Y.

Considerando el árbol de computación de una máquina MTN, también se puede ver el tiempo requerido para aceptar la cadena $x \in \mathrm{L}(MTN)$ como la longitud del camino de computación de aceptación mas corto entre todos estos caminos (Ver figura 3.1).

Definición 3.8 (Función tiempo de complejidad). Sea MTN una máquina de Turing no-determinística. La función tiempo de complejidad $T_{MTN} : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$ para MTN está definida como sigue:

[*] La notación NP representa la expresión “no determinístico tiempo polinomial” y no significa “no polinomial”.

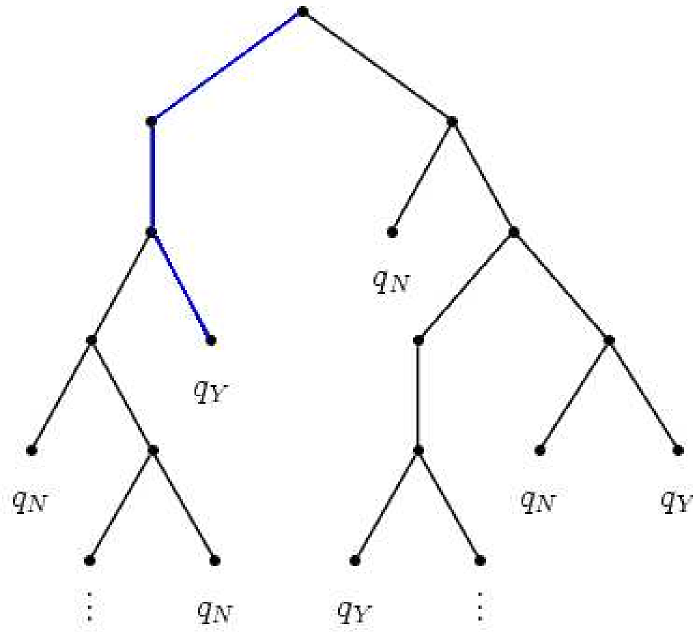


Figura 3.1: Arbol de computación para una cadena x en una MTN, cuyo camino de computación de aceptación más corto tiene longitud 3.

$$T_{MNT}(n) = \text{máx}\left\{ \{1\} \cup \left\{ \begin{array}{l} \text{m: existe un } x \in \text{L}(MTN) \text{ con } |x| = n \text{ tal que} \\ \text{el tiempo requerido por } MTN \text{ para aceptar } x \text{ es m.} \end{array} \right\} \right\}$$

Se puede observar que la función tiempo de complejidad para MTN depende solamente de el número de pasos que ocurren en las computaciones de aceptación, y que, por convención, T_{MNT} es igual a 1 si ninguna entrada de longitud n es aceptada por MTN. Este valor es asignado para que en éste caso la función esté bien definida.

Definición 3.9 (Máquina de Turing no determinística tiempo polinomial). Una máquina de Turing no determinística MTN es tiempo polinomial si existe un polinomio p tal que $T_{MTN}(n) \in O(p(n))$, para todo $n \in \mathbb{Z}^+$.

Definición 3.10 (Clase NP). La clase de complejidad NP se define como el siguiente conjunto:

$$\text{NP} = \left\{ \begin{array}{l} \text{L: L es un lenguaje y existe una máquina de Turing no-determinística} \\ MTN \text{ tiempo polinomial para la cual L} = \text{L}(MTN). \end{array} \right\}$$

Un problema de decisión Π se dice que pertenece a NP bajo un esquema de codificación e si el lenguaje $L[\Pi, e] \in$ NP. Al igual que en la clase P, solamente se dirá que Π está en NP sin especificar un esquema de codificación para Π, ya que se supondrá siempre que se tiene implícitamente un esquema de codificación razonable.

3.2.2. Algoritmos de verificación y la clase NP

A continuación se presentará la clase NP haciendo uso de algoritmos de verificación. Las nociones de "verificación" y "verificación tiempo polinomial" se muestran en los ejemplos siguientes.

Ejemplo 3.6
Sea HC el problema del ciclo hamiltoniano descrito en el ejemplo 1.30 (pág. 46), el cual se definió como sigue:

EL PROBLEMA DEL CICLO HAMILTONIANO (HC)

INSTANCIA :	Un grafo $G = (V, E)$
PREGUNTA :	¿ Existe un ordenamiento $\langle v_1, v_2, \ldots, v_n \rangle$ de V, con $n = \|V\|$ tal que $\{v_i, v_{i+1}\} \in E$ para $1 \leq i < n$ y $\{v_n, v_1\} \in E$?.

¿ Existe un algoritmo determinístico para HC ?

Dada una instancia $\langle G \rangle$ del problema, un posible algoritmo que lo resuelve debe listar todas las permutaciones de los vértices de G y luego chequear cada permutación para

determinar si es un ciclo hamiltoniano. Si se usa la matriz de adyacencia Ad del grafo G como su codificación y si m es el número de vértices de G, entonces $m = \Omega(\sqrt{n})$, donde $n = |\langle G \rangle|$ es la longitud de la codificación de G, puesto que $n \leq m^2$ por ser Ad simétrica (G es no dirigido). Existen $m!$ posibles permutaciones de los vértices, y por consiguiente el tiempo de ejecución es $\Omega(m!) = \Omega(\sqrt{n}\,!) = \Omega(2^{\sqrt{n}})$. Por lo tanto, este algoritmo no corre en tiempo polinomial y hasta el momento no se conoce algoritmo tiempo polinomial que resuelva HC.

Aunque HC es difícil de resolver, sin embargo, es fácil de verificar; es decir, dada una instancia arbitraria $\langle G \rangle$ de este problema, si la respuesta para $\langle G \rangle$ es "si", entonces, esto puede ser probado, proporcionando el ciclo hamiltoniano C requerido. La validez de esta afirmación puede ser verificada fácilmente chequeando que C es realmente un ciclo que recorre todos los vértices de G. De esta forma, es posible construir un algoritmo tiempo polinomial que verifique que la respuesta para $\langle G \rangle$ es "si". Por otra parte, si se considera una instancia $\langle G \rangle$ cuya respuesta es "no", esto es, si $\langle G \rangle$ no es hamiltoniano, ninguna lista de vértices debe llevar a que el algoritmo responda que $\langle G \rangle$ es hamiltoniano. $\diamondsuit$

Note que verificabilidad tiempo polinomial no implica solubilidad tiempo polinomial. En el ejemplo anterior, cuando se dice que se puede verificar una respuesta "si" para una instancia de HC en tiempo polinomial, no se está considerando el tiempo necesitado para buscar el circuito deseado entre la cantidad exponencial de posibles ciclos; lo que se afirma es que dado cualquier ciclo para una instancia $\langle G \rangle$, se puede verificar en tiempo polinomial si ese ciclo prueba o no que la respuesta para $\langle G \rangle$ es "si".

Formalmente un algoritmo de verificación se puede definir como sigue:

Definición 3.11 (Algoritmo de verificación). Un algoritmo de verificación A es un algoritmo que tiene dos argumentos, donde cada uno de ellos es una cadena y el segundo argumento es llamado un certificado. Un algoritmo A con dos argumentos, *verifica* una cadena de entrada x si existe un certificado y tal que $A(x, y) = 1$, donde

A(x, y) es el valor retornado por A con las entradas x y y.

Definición 3.12 (Algoritmo de verificación tiempo polinomial para un problema de decisión). Un algoritmo de verificación A(x, y) tiempo polinomial para un problema de decisión Π debe cumplir que:

- Si x representa una instancia "si"de Π entonces existe un certificado y tal que A(x, y)=1 y $|y| \leq O(|x|^c)$ para alguna constante c.
- Si x representa una instancia "no"de Π entonces no existe un certificado y tal que A(x, y)=1.

En otras palabras, un algoritmo de verificación A para un problema Π debe cumplir que para cualquier instancia "si" de Π, existe un certificado y que se puede usar para probar que la instancia es "si", y además, para cualquier instancia "no", no debe existir ningún certificado que lleve a que la instancia es "si".

Definición 3.13 (Clase NP en términos de verificadores). La clase de complejidad NP se define como la clase de problemas de decisión que pueden ser verificados por un algoritmo tiempo polinomial.

Ejemplo 3.7
El ejemplo 3.6 muestra que el problema del CICLO HAMILTONIANO está en NP. $\diamondsuit$

Ejemplo 3.8
El problema de determinar si un número entero es compuesto está en NP. El algoritmo de verificación recibe como entrada un entero a y un certificado b y verifica si b es menor que a y si b divide a a. Además de acuerdo al ejemplo 3.4 este problema tambien está en P. $\diamondsuit$

Ejemplo 3.9

Considérese el problema de decisión del AGENTE VIAJERO (AV) presentado en el ejemplo 1.31 (pág. 46), el cual se definió como sigue:

EL PROBLEMA DE DECISIÓN DEL AGENTE VIAJERO (AV)

INSTANCIA : Un conjunto finito $C = \{c_1, c_2, \ldots, c_m\}$ de “ciudades”, una “distancia” $d(c_i, c_j) \in \mathbb{Z}^+$ para cada par de ciudades c_i, $c_j \in C$ y una cota $k \in \mathbb{Z}^+$.

PREGUNTA : ¿Existe un tour de todas las ciudades en C que tenga una longitud total no mas que k, esto es, un ordenamiento $\langle c_{\Gamma(1)}, c_{\Gamma(2)}, \ldots, c_{\Gamma(m)} \rangle$ de C tal que: $\sum_{i=1}^{m-1} d(c_{\Gamma(i)}, c_{\Gamma(i+1)}) + d(c_{\Gamma(m)}, c_{\Gamma(1)}) \leq k$?.

El problema AV pertenece a NP. Un algoritmo de verificación para AV recibe como entrada un conjunto de ciudades, la distancia entre ellas, una cota dada B y un certificado. El algoritmo chequea si el certificado corresponde a un tour de todas las ciudades y si es así, computa su longitud y la compara con B. $\diamondsuit$

Ejemplo 3.10

Considérese el siguiente problema:

EL PROBLEMA SUBSET-SUM

INSTANCIA: Un conjunto S de enteros positivos y $t \in \mathbb{Z}^+$.

PREGUNTA: ¿Existe un subconjunto $S' \subseteq S$ tal que $t = \sum_{s \in S'} s$?.

El problema SUBSET-SUM $\in$ NP. Un algoritmo de verificación para él recibe como entrada un conjunto S de enteros positivos, un valor $t \in \mathbb{Z}^+$ y un certificado $S' \subseteq S$ y chequea si la suma de los elementos de S' es t. No es difícil ver que el algoritmo trabaja en tiempo polinomial. $\diamondsuit$

Ejemplo 3.11

Considere el siguiente problema:

EL PROBLEMA DE LA MOCHILA

INSTANCIA:	Un conjunto finito U, para cada $u \in U$ un valor $v(u) \in \mathbb{Z}^+$ y un peso $w(u) \in \mathbb{Z}^+$, un tamaño fijo $W \in \mathbb{Z}^+$ y un valor objetivo $K \in \mathbb{Z}^+$.
PREGUNTA:	¿Existe un subconjunto $U' \subseteq U$ tal que $\sum_{u \in U'} s(u) \leq W$ y $\sum_{u \in U'} v(u) \geq K$?.

El problema de la MOCHILA está en NP. Un algoritmo de verificación para este problema recibe como entrada un conjunto U, los valores y pesos para cada elemento de U, dos enteros positivos K y W y un certificado $U' \subseteq U$. El algoritmo chequea que la suma de los pesos de los elementos de U' sea menor que W y la suma de los valores sea mayor o igual que K. No es difícil ver que el algoritmo trabaja en tiempo polinomial. $\diamondsuit$

3.3. RELACIÓN ENTRE P Y NP

Se podría pensar en la clase P como aquella que contiene los problemas "fáciles de resolver", en el sentido de que existe para cada uno de ellos un algoritmo determinístico que los resuelve en tiempo polinomial; además se puede ver que todo problema de decisión soluble por un algoritmo determinístico tiempo polinomial es también soluble por un algoritmo no determinístico tiempo polinomial (Teorema 3.1). Muchos investigadores han tratado durante décadas de establecer la relación entre las clases P y NP, han puesto sus esfuerzos en tratar de responder una de las preguntas más importantes de este siglo: ¿P = NP?, pero aunque alrededor de estas clases se han establecido muchas relaciones y se ha desarrollado una extensa teoría, la pregunta aún continua abierta. Realmente, la respuesta a esta pregunta traería grandes consecuencias no sólo para las matemáticas sino tambien para otras áreas. En esta sección se presentan dos

resultados fundamentales al respecto.

Teorema 3.1. P $\subseteq$ NP.

Demostración. Sea Π un problema y supóngase que $\Pi \in$ P, entonces existe un algoritmo determinístico tiempo polinomial A que lo resuelve. Se puede obtener un algoritmo no determinístico tiempo polinomial para Π, solamente usando A como la etapa de chequeo del nuevo algoritmo e ignorando la cadena supuesta en la etapa de suposición. Así $\Pi \in$ NP.

Existen muchas razones para creer que esta inclusión es propia, esto es, que P no es igual a NP. Una de ellas es que hasta el momento no se conocen métodos eficientes para convertir un algoritmo no determinístico tiempo polinomial en uno determinístico tiempo polinomial. El siguiente resultado muestra un procedimiento que obtiene un algoritmo determinístico a partir de uno no determinístico pero que sin embargo no cumple con las condiciones de eficiencia deseadas.

Teorema 3.2. Si $\Pi \in$ NP, entonces existe un polinomio p tal que Π puede ser resuelto por un algoritmo determinístico que tiene tiempo de complejidad $O(2^{p(n)})$. [GJ79]

Demostración. Como $\Pi \in$ NP, supóngase que A es un algoritmo no determinístico tiempo polinomial que resuelve Π, y sea $q(n)$ una cota polinomial para el tiempo de complejidad de A. Entonces, para toda entrada aceptada por A de longitud n, debe existir alguna cadena supuesta (sobre el alfabeto Σ) de longitud a lo más $q(n)$ que lleve la etapa de chequeo de A a responder "si" para esta entrada en a lo más $q(n)$ pasos. El número de posibles supuestos que necesitan ser considerados es a lo más $k^{q(n)}$, donde $k = |\Sigma|$, ya que los supuestos más cortos que $q(n)$ pueden ser considerados como supuestos de longitud exactamente $q(n)$ completándolos con símbolos blanco. Se puede averiguar determinísticamente si A tiene una computación de aceptación para una entrada dada de longitud n, aplicando la etapa de chequeo determinística de A hasta que pare o haga $q(n)$ pasos sobre cada uno de los $k^{q(n)}$ posibles supuestos. La simulación responde "si" si encuentra una cadena supuesta que lleva a una computación de

aceptación en la cota de tiempo; en otro caso responde "no". Por lo tanto, esto produce un algoritmo determinístico que resuelve Π; además, su tiempo de complejidad es $q(n)k^{q(n)}$, el cual es $\mathrm{O}(2^{p(n)})$ para una escogencia apropiada del polinomio p. En efecto,

$$
\begin{aligned}
k &\leq 2^k && \text{para todo } k \geq 1,\\
k^{q(n)} &\leq 2^{kq(n)} && \text{para todo } k \geq 1,\\
q(n)k^{q(n)} &\leq q(n)2^{kq(n)} \leq 2^{q(n)}2^{kq(n)} && \text{para todo } k \geq 1,\\
&= 2^{q(n)+kq(n)} && \text{para todo } k \geq 1.
\end{aligned}
$$

Haciendo $p(n) = q(n) + kq(n)$, se obtiene que $q(n)\Delta k^{q(n)} = O(2^{p(n)})$.

En efecto, la dificultad en encontrar algoritmos determinísticos tiempo polinomial para cualquier problema en NP, es un argumento fuerte para pensar que P $\neq$ NP. Sin embargo, al mismo tiempo, el hecho de que ninguna prueba a esta conjetura haya surgido, a pesar de los esfuerzos de muchos investigadores, podría llevar a creer que P = NP. Por otro lado, es importante señalar que el teorema 3.2 parece mostrar que los algoritmos no determinísticos son estrictamente mas poderosos que los determinísticos. En la Figura 3.2 se muestra una posible representación de las clases P y NP, quizá la más esperada, en la cual la región NP-P no es vacía.

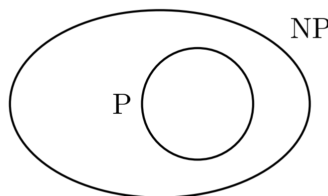


Figura 3.2: Una posible visión del mundo de NP.

3.4. TRANSFORMACIÓN POLINOMIAL Y LA CLASE NPC

En ésta sección se presenta una clase importante de problemas de decisión, la clase de problemas NP-completos. Esta clase tiene la propiedad fundamental que si cualquier problema NP-completo puede ser resuelto determinísticamente en tiempo polinomial, entonces todo problema en NP tiene solución determinista en tiempo polinomial, lo cual implicaría que P = NP. Además, es importante señalar que hasta el momento, para ningún problema NP-completo ha sido encontrado un algoritmo determinístico tiempo polinomial que lo resuelva. Por esta razón, los problemas NP-completos son considerados como los problemas "más difíciles" en NP. Antes de presentar la definición de estos problemas, se introducirá la noción de transformación polinomial que juega un papel importante en la teoría de NP-completitud, como se observará a lo largo del capítulo.

Definición 3.14 (Transformación polinomial aplicada a lenguajes). Sean Σ_1 y Σ_2 alfabetos. Una transformación polinomial de un lenguaje $L_1 \subseteq \Sigma_1^*$ a un lenguaje $L_2 \subseteq \Sigma_2^*$ es una función $f : \Sigma_1^* \to \Sigma_2^*$ que satisface las dos condiciones siguientes:

1. Existe una máquina de Turing tiempo polinomial que computa f.
2. Para todo $x \in \Sigma_1^*$, $x \in L_1$ si, y sólo si $f(x) \in L_2$.

En este caso se escribe $L_1 \leq_p L_2$ y se lee "L_1 se transforma polinomialmente a L_2". La Figura 3.3 ilustra la noción de transformación polinomial.

Teorema 3.3. Sean L_1 y L_2 lenguajes tales que $L_1 \leq_p L_2$. Si $L_2 \in P$ entonces $L_1 \in P$. [GJ79]

***Demostración*.** Sean Σ_1^* y Σ_2^* los alfabetos de L_1 y L_2 respectivamente. Por hipótesis, $L_1 \leq_p L_2$, es decir, existe una transformación polinomial $f : \Sigma_1^* \to \Sigma_2^*$ de L_1 a L_2. Sea

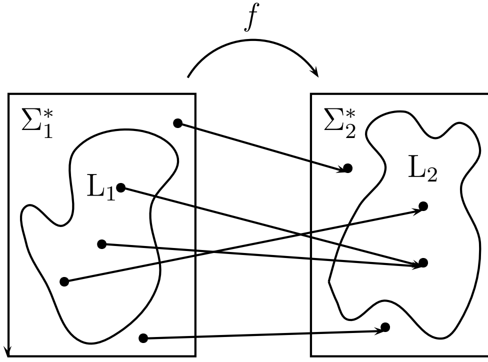


Figura 3.3: Una ilustración de una transformación polinomial f de L_1 a L_2

MT_f la máquina de Turing tiempo polinomial que computa f. Por otra parte, como $L_2 \in P$, existe una máquina de Turing MT_2 tiempo polinomial que acepta L_2. Se debe probar que $L_1 \in P$, esto es, que existe una máquina de Turing MT_1 tiempo polinomial que acepta L_1.

El programa MT_1 puede ser construido de la siguiente forma: Para cada entrada $x \in \Sigma_1^*$, MT_1 usa MT_f para transformar x en $f(x) \in \Sigma_2^*$ y luego, usa MT_2 para determinar si $f(x) \in L_2$. Si es así, MT_1 acepta x, en caso contrario MT_1 rechaza x. Puesto que $x \in L_1$ si, y sólo si $f(x) \in L_2$, se obtiene que MT_1 acepta L_1. Además, MT_1 opera en tiempo polinomial pues MT_f y MT_2 operan en tiempo polinomial. Mas precisamente, si p_f y p_2 son funciones polinomiales que acotan el tiempo de ejecución de MT_f y MT_2 respectivamente, entonces $|f(x)| \leq p_f(|x|)$ y el tiempo de ejecución de MT_1 es $O(p_f(|x|) + p_2(p_f(|x|)))$, el cual es polinomial en $|x|$.

Teorema 3.4. Sean L_1, L_2 y L_3 lenguajes. Si $L_1 \leq_p L_2$ y $L_2 \leq_p L_3$ entonces $L_1 \leq_p L_3$. [GJ79]

Demostración. Sean Σ_1, Σ_2 y Σ_3 los alfabetos asociados a L_1, L_2 y L_3 respectivamente.

Como $L_1 \leq_p L_2$ y $L_2 \leq_p L_3$, existen transformaciones polinomiales $f_1 : \Sigma_1^* \rightarrow \Sigma_2^*$ de L_1 a L_2 y $f_2 : \Sigma_2^* \rightarrow \Sigma_3^*$ de L_2 a L_3.

Se debe probar que $L_1 \leq_p L_3$, esto es que existe una transformación polinomial $f : \Sigma_1^* \rightarrow \Sigma_3^*$ de L_1 a L_3.

Sea $f(x) = f_2(f_1(x))$ para todo $x \in \Sigma_1^*$.
f es una transformación polinomial de L_1 a L_3:

- Como f_1 y f_2 son transformaciones polinomiales se tiene que:
  Para todo $x \in \Sigma_1^*,\ x \in \mathrm{L}_1 \Longleftrightarrow f_1(x) \in \mathrm{L}_2$ y
  para todo $y \in \Sigma_2^*,\ y \in \mathrm{L}_2 \Longleftrightarrow f_2(y) \in \mathrm{L}_3$
  Por lo tanto, tomando $y = f_1(x)$ se cumple que:

  $$x \in L_1 \Longleftrightarrow f_1(x) \in L_2 \Longleftrightarrow f_2(f_1(x)) \in L_3,$$

  Luego, $x \in \mathrm{L}_1 \Longleftrightarrow f(x) \in \mathrm{L}_3$, para todo $x \in \Sigma_1^*$.

- Por un razonamiento similar al usado en el teorema 3.3, se puede ver que f opera en tiempo polinomial.

La noción de transformación polinomial también puede ser planteada solamente en términos de problemas, de la siguiente manera.

Definición 3.15 (Transformación polinomial aplicada a problemas de decisión). Sean Π_1 y Π_2 problemas de decisión con esquemas de codificación asociados e_1 y e_2 respectivamente. Se dice que Π_1 se transforma polinomialmente a Π_2 ($\Pi_1 \leq_p \Pi_2$) si existe una transformación polinomial de $\mathrm{L}[\Pi_1, e_1]$ a $\mathrm{L}[\Pi_2, e_2]$. Es decir, en términos de problemas, se puede considerar una transformación polinomial de Π_1 a Π_2 como una función $f : D_{\Pi_1} \rightarrow D_{\Pi_2}$ que satisface las siguientes condiciones:

1. f es computable por un algoritmo tiempo polinomial.
2. Para todo $I \in D_{\Pi_1}$, $I \in Y_{\Pi_1}$ si, y sólo si, $f(I) \in Y_{\Pi_2}$.

en donde D_{Π_1} y D_{Π_2} denota el conjunto de todas las instancias del problema Π_1 y Π_2 respectivamente, y Y_{Π_1}, Y_{Π_2} el conjunto de instancias con respuesta “si”.

En la sección 3.6 se muestran algunos ejemplos de transformación polinomial.

Informalmente, un problema Π puede ser transformado a otro problema $\Pi^{'}$ si cualquier instancia de Π puede ser "fácilmente expresada" como una instancia de $\Pi^{'}$, y una solución para esta instancia provee una solución para la instancia correspondiente de Π. Así, si un problema Π se reduce a otro problema $\Pi^{'}$, entonces Π es en cierto sentido no más duro para resolver que Π.

Definición 3.16 (Lenguaje NP-completo). Un lenguaje L es NP-completo si:

1. $L \in NP$ y
2. $L^{'} \leq_p L$ para todo $L^{'} \in NP$.

Si un lenguaje L satisface la propiedad 2, pero no necesariamente la propiedad 1 entonces se dice que L es ***NP-duro***.

En forma análoga a las definiciones anteriores, la definición de problema NP-completo también es expresada en términos de problemas.

Definición 3.17 (Problema NP-completo). Un problema de decisión Π es NP-completo si:

1. $\Pi \in NP$ y
2. $\Pi^{'} \leq_p \Pi$ para todo $\Pi^{'} \in NP$.

Definición 3.18 (Clase NPC). La clase NPC es el conjunto de todos los lenguajes NP-completos.

Si $P \neq NP$ entonces un problema Π NP-completo cumple que $\Pi \in$ NP-P. Es decir, $\Pi \in P$ si, y sólo si $P = NP$. Asumiendo que $P \neq NP$, se puede dar una visión mas detallada del "mundo de NP", como se muestra en la Figura 3.4.

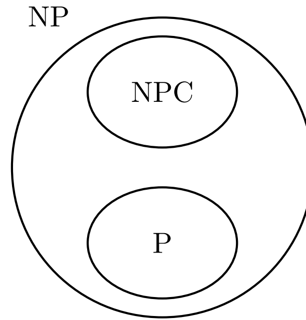


Figura 3.4: Una visión mas detallada del mundo de NP.

Mostrar que un problema es NP-completo implicaría mostrar que todo problema en NP se transforma a él, lo cual evidentemente no es una tarea fácil, más aún cuando no se conoce cuántos problemas NP-completos existen en realidad. El siguiente teorema proporciona una herramienta más práctica para probar NP-completitud, si se tiene un problema que ya es NP-completo.

Teorema 3.5. Sean L_1 y L_2 lenguajes. Si L_1 y L_2 están en NP, L_1 es NP-completo y $L_1 \leq_p L_2$ entonces L_2 es NP-completo. [GJ79]

Demostración. Como $L_2 \in$ NP, sólo falta probar que, para todo $L' \in$ NP, $L' \leq_p L_2$. Sea L' cualquier lenguaje en NP. Como L_1 es NP-completo, debe cumplirse que $L' \leq_p L_1$. Por la transitividad de $\leq_p$ mostrada en el teorema 3.4 y puesto que $L_1 \leq_p L_2$ se obtiene que $L' \leq_p L_2$.

Teorema 3.6. [CLR90]

1. Si cualquier problema NP-completo es soluble por un algoritmo determinístico tiempo polinomial, entonces P = NP.

2. Si algún problema en NP no es soluble por un algoritmo deteminístico tiempo polinomial, entonces ningún problema NP-completo es soluble por un algoritmo determinístico tiempo polinomial.

Demostración.

1. Sea Π un problema NP-completo soluble por un algoritmo determinístico tiempo polinomial, es decir, $\Pi \in$ P. Por la condición 2 de la definición 3.17, para todo problema $\Pi' \in$ NP, se cumple que $\Pi' \leq_p \Pi$ y por el Teorema 3.3, se tiene que $\Pi' \in$ P.

2. Supóngase que existe un problema $\Pi \in$ NP tal que $\Pi \notin$ P. Sea Π' cualquier problema NP-completo y supóngase por contradicción que $\Pi' \in$ P. Entonces por definición de NP-completitud, $\Pi \leq_p \Pi'$, y por el Teorema 3.3, $\Pi \in$ P, lo cual es una contradicción.

3.5. TEOREMA DE COOK

Hasta el momento se ha presentado la noción de problema NP-completo, pero no se ha probado que un problema particular es NP-completo.

En 1971, el matemático Stephen Cook demostró el primer problema NP-completo, el cual es un problema de decisión de Lógica Booleana llamado *el problema de satisfabilidad* (SAT). Los términos que se usan para describirlo son definidos como sigue:

Sea U = $\{u_1, u_2, u_3, \ldots, u_m,\}$ un conjunto de variables booleanas. Una *asignación de verdad* para U es una función $t : U \rightarrow \{V, F\}$ tal que:

- Si $t(u)$ = V entonces u es verdadera.
- Si $t(u)$ = F entonces u es falsa.

Si u es una variable en U, entonces u y $\overline{u}$ son *literales* sobre U.
El literal u es verdadero bajo t si, y solo si la variable u es verdadera bajo t.
El literal $\overline{u}$ es verdadero bajo t si, y solo si la variable u es falsa bajo t.

Una *cláusula* sobre U es un conjunto de literales sobre U, tal como $\{u_1, \overline{u_3}, u_5\}$. Esto representa la disyunción de estos literales, es decir, una asignación de verdad satisface la cláusula cuando al menos uno de sus miembros es verdadero bajo esta asignación.

Una colección C de cláusulas es *satisfactible* si, y solo si, existe alguna asignación de verdad para U que simultáneamente satisface todas las cláusulas en C.

El problema se describe como sigue:
INSTANCIA: Un conjunto U de variables y una colección C de cláusulas sobre U.
PREGUNTA: ¿Existe una asignación de verdad que satisface C?

Teorema 3.7. (TEOREMA DE COOK) El problema de satisfabilidad (SAT) es NP-completo. [GJ79]

Demostración. Se debe probar que:

1. SAT está en NP.

2. L$\leq_p$ L$_{SAT}$ para todo lenguaje L$\in$ NP, donde L$_{SAT}$ = L[SAT, e] para algún esquema de codificación e.

1. La máquina de Turing no determinística tiempo polinomial que resuelve SAT supone una asignación de verdad para las variables dadas (en la etapa de suposición) y en la etapa de chequeo verifica si la asignación satisface todas las cláusulas en la colección dada C. Además, se puede ver que la ejecución de esta máquina es polinomial.

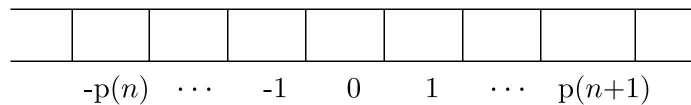
2. Sea L un lenguaje en NP y sea $M = \langle Q, \Sigma, \delta \rangle$ una MTN tiempo polinomial que acepta el lenguaje L (L = L$_M$).

   Se debe construir una función $f_L : \Sigma^* \rightarrow$ D$_{SAT}$. Por simplicidad, f_L es construida directamente a instancias de SAT y no al conjunto de cadenas sobre el alfabeto

de SAT. Así, $f_L(x)$ representará un conjunto U de variables y una colección C de cláusulas para alguna cadena $x \in \Sigma^*$.

Se probará que f_L es una transformación polinomial, mostrando primero que para todo $x \in \Sigma^*$, $x \in L$ si, y solo si $f_L(x)$ tiene una asignación de verdad satisfaciente, y después que f_L se computa en tiempo polinomial.

Dada una entrada $x \in \Sigma^*$, si x es aceptada por M existe una computación de aceptación de M sobre x tal que el número de pasos en la etapa de chequeo y el número de símbolos en la cadena supuesta son acotados por un polinomio p(n), $n = |x|$, esto ocurre puesto que M es tiempo polinomial y por lo tanto el tiempo de complejidad (que se puede ver como el máximo número de pasos en la etapa de suposición y de chequeo requeridos por M para aceptar x) es acotado por p(n). Tal computación involucra solamente los cuadrados de la cinta enumerados desde -p(n) hasta p(n)+1, puesto que la cabeza de lectura-escritura comienza en el cuadrado 1 y se mueve a lo más un cuadrado en cualquier paso simple.



La situación de la etapa de chequeo en cualquier momento puede ser especificada completamente, dando el contenido del cuadrado, el estado actual, y la posición de la cabeza de lectura-escritura. Además, puesto que no hay más que p(n) pasos en la computación de chequeo, hay a lo más p(n)+1 tiempos distintos que pueden ser considerados.

Los elementos de Q se etiquetarán como sigue:
$q_0,\ q_1 = q_Y,\ q_2 = q_N,\ q_3,\ \ldots,\ q_r$, donde $r = |Q| - 1$.
Y los elementos de Σ como: $s_0 = b,\ s_1,\ s_2,\ \ldots,\ s_v$, donde $v = |\Sigma| - 1$.

Existirán tres tipos de variables, cada una de las cuales tiene un significado particular como se muestra en la Tabla 3.2.

Una computación de M induce una asignación de verdad sobre estas variables bajo la convención que, si el programa para antes del tiempo p(n), la configuración

VARIABLE	RANGO	SIGNIFICADO
Q$[i,k]$	$0 \leq i \leq$ p(n) $0 \leq k \leq r$	En tiempo i, M está en estado q_k.
H$[i,j]$	$0 \leq i \leq$p(n) -p$(n) \leq j \leq$ p(n)+1	En tiempo i, la cabeza de lectura-escritura, está examinando el cuadrado j de la cinta.
S$[i,j,k]$	$0 \leq i \leq$ p(n) -p$(n) \leq j \leq$ p(n)+1 $0 \leq k \leq v$	En tiempo i, el contenido del cuadrado j de la cinta es el símbolo s_k.

Tabla 3.2: Variables en $f_L(x)$ y sus significados.

permanece estática en todos los tiempos posteriores, manteniendo el mismo estado de parada, posición de la cabeza y contenido de la cinta. Por otro lado, una asignación de verdad arbitraria para ellas no necesariamente corresponde a una computación, mucho menos a una computación de aceptación, puesto que una asignación de verdad arbitraria para ellas puede implicar por ejemplo que en un mismo tiempo, un cuadrado dado de la cinta contenga varios símbolos o que la máquina pueda estar simultáneamente en varios estados diferentes o que la cabeza de lectura-escritura pueda estar examinando varios cuadrados a la vez.

La transformación f_L trabaja por construir una colección de cláusulas que involucre estas variables tal que una asignación de verdad es una asignación de verdad satisfaciente si, y solo si esta es la asignación de verdad inducida por una computación de aceptación para x.

Las cláusulas en $f_L(x)$ pueden ser divididas en seis grupos, los cuales son creados para cumplir una misión específica, que son mostrados en la Tabla 3.3.

Se puede observar que si todos los seis grupos de cláusulas realizan sus misiones, entonces una asignación de verdad satisfaciente para las variables conducirá a la computación de aceptación deseada para x. Así, todo lo que se necesita mostrar es cómo construir los grupos de cláusulas para que cumplan sus misiones.

GRUPO DE CLAUSULAS	MISION
G_1	En cada tiempo i, M está en exactamente un estado.
G_2	En cada tiempo i, la cabeza de lectura-escritura está escaniando exactamente un cuadrado de la cinta.
G_3	En cada tiempo i, cada cuadrado de la cinta contiene exactamente un símbolo de Σ.
G_4	En tiempo 0, la computación está en la configuración inicial de su etapa de chequeo para la entrada x.
G_5	En tiempo p(n), M ha entrado al estado q_y, y por lo tanto ha aceptado x.
G_6	Para cada tiempo i, $0 \leq i \leq$ p(n), la configuración de M en tiempo $i+1$ se sigue de una única aplicación de la función de transición δ desde la configuración en tiempo i.

Tabla 3.3: Grupos de cláusulas en $f_L(x)$ y sus misiones.

La Tabla 3.4 muestra las cláusulas que conforman a cada grupo.

El primer subgrupo de cláusulas de G_1 pueden ser satisfechas simultáneamente si, y sólo si, para cada tiempo i, M está en al menos un estado. Las cláusulas del segundo subgrupo pueden ser satisfechas simultáneamente si y sólo si, en cada tiempo i, M no está en dos estados a la vez. Los grupos G_2 y G_3 son construidos de la misma forma.

El grupo G_4 está formado por cláusulas de un solo literal y garantiza que en tiempo 0, la máquina está en su configuración inicial. Las cláusulas del primer subgrupo son satisfechas si y sólo si en tiempo 0, el estado es q_0, la cabeza de lectura-escritura está examinando el cuadrado 1 de la cinta y el símbolo en el cuadrado 0 es $s_0 = b$. Las cláusulas del segundo subgrupo son satisfechas si y sólo si en tiempo 0, la entrada x está escrita desde el cuadrado 1 hasta el cuadrado n y todos los demás cuadrados contienen el símbolo blanco.

GRUPOS DE CLAUSULAS	SUBGRUPO	CLAUSULAS EN EL GRUPO
G_1	1	$\{Q[i, 0], Q[i, 1], \ldots, Q[i, r]\}$
	2	$\{Q[i, j], Q[i, j']\}$
		$0 \leq i \leq \mathrm{p}(n), \ 0 \leq j < j' \leq r$
G_2	1	$\{H[i, -p(n)], H[i, -p(n)+1], \ldots, H[i, p(n)+1]\}$
	2	$\{H[i, j], H[i, j']\}$
		$0 \leq i \leq \mathrm{p}(n)$, $-\mathrm{p}(n) \leq j < j' \leq \mathrm{p}(n)+1$
G_3	1	$\{S[i, j, 0], S[i, j, 1], \ldots, S[i, j, v]\}$
	2	$\{S[i, j, k], S[i, j, k']\}$
		$0 \leq i \leq \mathrm{p}(n)$, $-\mathrm{p}(n) \leq j \leq \mathrm{p}(n)+1$, $0 \leq k < k' \leq v$
G_4	1	$\{Q[0, 0]\}, \{H[0, 1]\}, \{S[0, 0, 0]\}$,
	2	$\{S[0, 1, k_1]\}, \{S[0, 2, k_2]\}, \ldots, \{S[0, n, k_n]\}$,
		$\{S[0, n+1, 0]\}, \{S[0, n+2, 0]\}, \ldots, \{S[0, p(n)+1, 0]\}$,
		donde $x = s_{k_1} s_{k_2} s_{k_3} \ldots s_{k_n}$
G_5	1	$\{Q[p(n), 1]\}$
G_6	1	$\{S[i, j, l], H[i, j], S[i+1, j, l]\}$
	2	$\{H[i, j], Q[i, k], S[i, j, l], H[i+1, j+\Delta]\}$
		$\{H[i, j], Q[i, k], S[i, j, l], Q[i+1, k']\}$
		$\{H[i, j], Q[i, k], S[i, j, l], S[i+1, j, l']\}$
		$0 \leq i < \mathrm{p}(n)$, $-\mathrm{p}(n) \leq j \leq \mathrm{p}(n)+1$, $0 \leq k \leq r$, $0 \leq l \leq v$
		donde si $q_k \in Q - \{q_Y, q_N\}$, entonces los valores
		de Δ, k' y l' son tales que $\delta(q_k, s_l) = (q_{k'}, s_{l'}, \Delta)$
		y si $q_k \in \{q_Y, q_N\}$, entonces $\Delta = N$, $k' = k$, $l' = l$.

Tabla 3.4: Los seis grupos de cláusulas de $f_L(x)$.

La cláusula en G_5 es satisfecha cuando en tiempo p(n) la máquina alcanza el estado de parada q_Y.

El primer subgrupo de cláusulas de G_6 garantiza que si la cabeza de lectura-escritura no está examinando el cuadrado j en el tiempo i, entonces el símbolo en el cuadrado j no es cambiado entre los tiempos i y $i+1$, de lo contrario las cláusulas no serán satisfechas. Las cláusulas del segundo subgrupo son satisfechas si y sólo si los cambios de una configuración a la próxima son de acuerdo a la función de transición δ para M.

Así, se ha mostrado cómo construir los grupos de cláusulas $G_1,\ldots,G_6$ que cumplen la misión establecida previamente. Luego:

1. Si $x \in$ L, entonces existe una computación de aceptación de M sobre x de longitud p(n) o menos, y esta computación impone una asignación de verdad que satisface todas las cláusulas en $C = G_1 \cup G_2 \cup G_3 \cup G_4 \cup G_5 \cup G_6$.
2. Recíprocamente, la construcción de C es tal que cualquier asignación de verdad satisfaciente para C corresponde a una computación de aceptación de M sobre x.

De (1) y (2) se sigue que $f_L(x)$ tiene una asignación de verdad satisfaciente si y sólo si $x \in L$.

En este momento, sólo falta probar que f_L se computa en tiempo polinomial, es decir, que $f_L(x)$ puede ser construido de x en tiempo acotado por una función polinomial de $n = |x|$. Para esto, basta con verificar que el tamaño de $f_L(x)$ es acotado por arriba por una función polinomial de n. Para esto, puesto que $f_L(x)$ es una instancia de SAT, determinada por el conjunto U de variables y la colección C de cláusulas, es necesario observar el número de variables (Tabla 3.5) y el número de cláusulas (Tabla 3.6) que componen la instancia.

VARIABLE	No. DE VARIABLES
Q	$[\text{p}(n+1)](r+1)$
H	$2[\text{p}(n+1)]^2$
S	$2[\text{p}(n+1)]^2(v+1)$

Tabla 3.5: Número de variables.

Además, se puede observar que $|U| = O(p^2(n))$ y $|C| = O(p^3(n))$. Por otra parte, la longitud de la instancia obtenida, $|f_L(x)|$, puede considerarse como $|f_L(x)| = |U||C|$,

GRUPO DE CLAUSULAS	No. DE CLAUSULAS
G_1	$[p(n)+1]+[p(n)+1](r+1)r/2$
G_2	$[p(n)+1]+2p(n)[p(n)+1]^2$
G_3	$2[p(n)+1]^2+2[p(n)+1]^2(v+1)v/2$
G_4	$3+[p(n+1)]$
G_5	1
G_6	$6p(n)[p(n)+1](r+1)(v+1)$

Tabla 3.6: Número de cláusulas en cada grupo.

esto es, cuando en cada cláusula de C se encuentran todas las varibles de U. Por consiguiente, $|f_L(x)| = O(p^5(n))$. De lo cual se obtiene que $f_L(x)$ se puede construir en tiempo polinomial para cualquier entrada x y por lo tanto f_L se computa en tiempo polinomial.

De lo anterior se obtiene que f_L es una transformación polinomial de L a SAT. Finalmente, se puede concluir que SAT es NP-completo.

3.6. ALGUNOS PROBLEMAS NP-COMPLETOS

En esta sección se presentan algunas pruebas de NP-completitud, haciendo uso de uno de los resultados mas útiles que se han mencionado, el Teorema 3.5. Así, se mostrará básicamente cómo se puede transformar un problema a otro para deducir que si uno de ellos es NP-completo, el otro también lo es.

3.6.1. El problema del ciclo hamiltoniano (HC)

Para probar que el problema del ciclo hamiltoniano es NP-completo se utilizará el hecho que el problema 3-SATISFABILIDAD es NP-completo [GJ79]. Este problema se define como sigue:

EL PROBLEMA 3-SATISFABILIDAD (3SAT)

INSTANCIA	Una colección $C = \{c_1, c_2, \ldots, c_m\}$ de cláusulas sobre un conjunto finito U de variables tales que $\|c_i\| = 3$ para $1 \leq i \leq m$.
PREGUNTA	¿Existe una asignación de verdad para U que satisfaga todas las cláusulas en C?

Teorema 3.8. HC es NP-completo. [CLR90]

Demostración. Por el ejemplo 3.6 se tiene que HC $\in$ NP.
Ahora se mostrará que 3SAT se transforma polinomialmente a HC.
Dada una colección de cláusulas $C = \{c_1, c_2, \ldots, c_k\}$ sobre un conjunto finito de variables $x_1, x_2, \ldots, x_n$, con $|c_i| = 3$ para $i = 1, \ldots, k$, se debe construir un grafo $G = (V, E)$ en tiempo polinomial tal que G contiene un ciclo hamiltoniano si y sólo si C es satisfactible.

La construcción del grafo G está basada en dos estructuras especiales, que son partes de grafos que imponen ciertas propiedades.

La primer estructura es el subgrafo A mostrado en la Figura 3.5. Supóngase que A es un subgrafo de algún grafo G' y que las únicas conexiones entre A y el resto de G' son por medio de los vértices a, a', b y b'. Además, si G' tiene un ciclo hamiltoniano, éste debe pasar por los vértices z_1, z_2, z_3 y z_4 en una de las formas mostradas en la Figura 3.5(b) y (c).

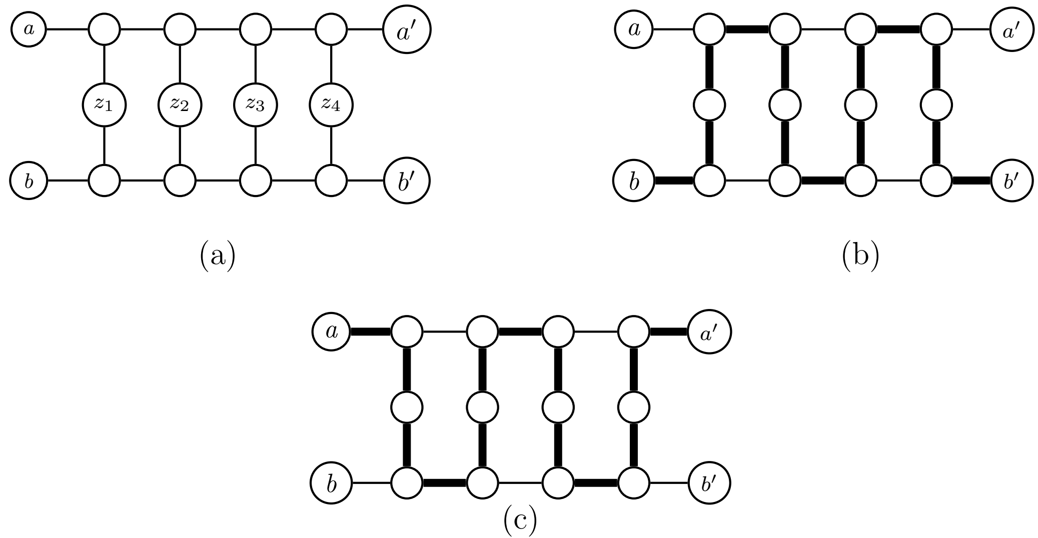


Figura 3.5: (a) Subgrafo A. (b)-(c) Los dos recorridos posibles en el cual el subgrafo A de un grafo G' puede ser atravesado por un ciclo hamiltoniano.

La segunda estructura es el subgrafo B mostrado en la Figura 3.6. Supóngase que B es un subgrafo de algún grafo G' y que las únicas conexiones de B al resto del G' son a través de los vértices b_1, b_2, b_3 y b_4. Un ciclo hamiltoniano del grafo G' no puede atravesar todas las aristas $\{b_1, b_2\}$, $\{b_2, b_3\}$ y $\{b_3, b_4\}$, ya que entonces los vértices en B distintos de b_1, b_2, b_3 y b_4 no serían atravesados por el ciclo. Luego, un ciclo hamiltoniano de G' debe recorrer B en una de las formas mostradas en la Figura 3.6.

El grafo G que se construirá está compuesto principalmente de subgrafos A y B. En la Figura 3.7 se ilustra la construcción del grafo G para la instancia de 3SAT compuesta por $U = \{x_1, x_2, x_3\}$ y $C = \{\{x_1, x_2, \overline{x}_3\}, \{\overline{x}_1, \overline{x}_2, x_3\}, \{x_2, \overline{x}_2, x_3\}, \{x_2, x_3, \overline{x}_3\}\}$.
La construcción del grafo G es como sigue:
Para cada una de las k cláusulas en C, se incluye un subgrafo B, y se unen estos subgrafos entre sí en serie conectando el vértice $b_{i,4}$ al vértice $b_{i+1,1}$ para $i = 1, 2, \ldots, k-1$.

Para cada variable x_m se incluyen dos vértices x'_m y x''_m. Se conectan estos dos vértices por medio de dos copias de la arista $\{x'_m, x''_m\}$, las cuales son denotadas por e_m y $\overline{e}_m$ para distinguirlas.

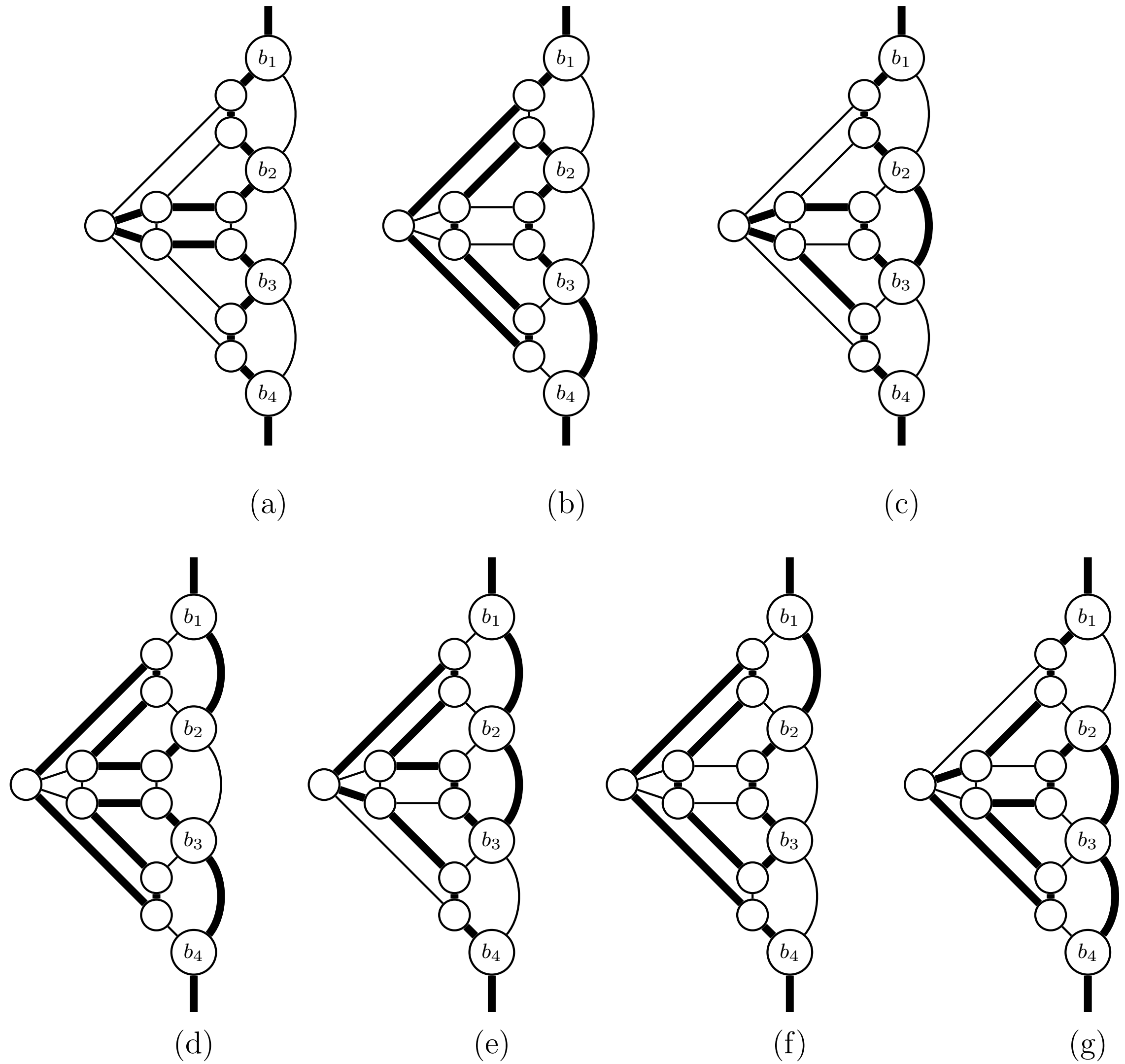


Figura 3.6: Subgrafo B. (a)-(g) Todos los posibles recorridos por donde puede pasar un ciclo hamiltoniano de un grafo G'.

Cada par de estas aristas forma un bucle; se conectan estos bucles en serie adicionando las aristas $\{x''_m, x'_{m+1}\}$ para $m = 1, \ldots, n-1$. El lado izquierdo del grafo (cláusulas) se conecta al lado derecho (variables) por medio de dos aristas $\{b_{1,1},\ x'_1\}$ y $\{b_{k,4},\ x''_n\}$, las cuales corresponden a la arista más alta y más baja de la Figura 3.7.

Si el $j-$ésimo literal de la cláusula c_i es x_m, entonces se conecta la arista $\{b_{i,j},\ b_{i,j+1}\}$ con la arista e_m mediante un subgrafo A. Si el j-ésimo literal de la cláusula c_i es $\overline{x}_m$, entonces se conecta la arista $\{b_{i,j},\ b_{i,j+1}\}$ con la arista $\overline{e}_m$ mediante un subgrafo A.

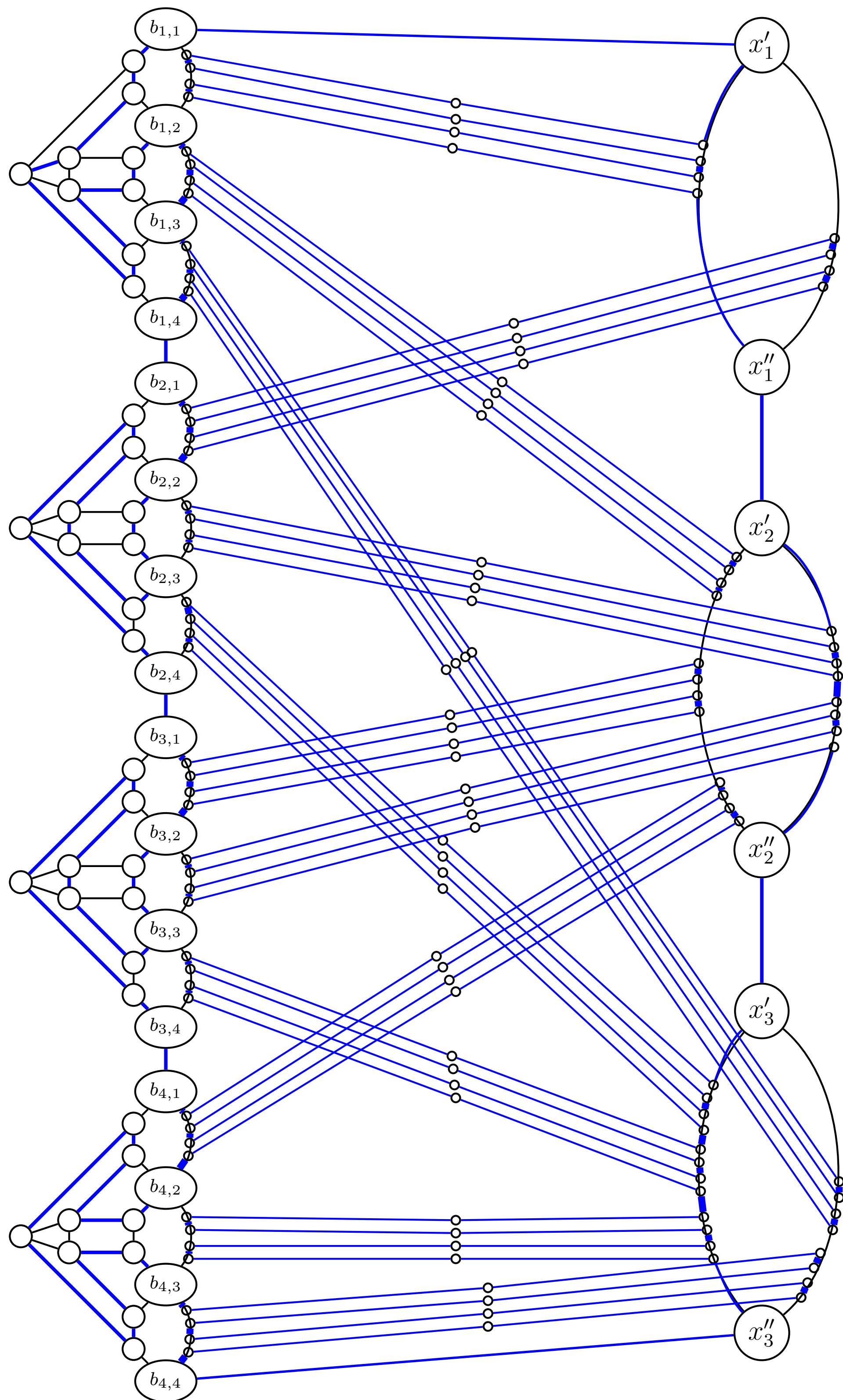


Figura 3.7: Transformación de una instancia de 3SAT a una instancia de HC.

En la Figura 3.7, por ejemplo, se colocan tres subgrafos A de la siguiente manera:

- entre $\{b_{1,1},\, b_{1,2}\}$ y e_1.
- entre $\{b_{1,2},\, b_{1,3}\}$ y e_2.
- entre $\{b_{1,3},\, b_{1,4}\}$ y $\overline{e}_3$.

puesto que la cláusula c_1 es $\{x_1,\, x_2,\, \overline{x}_3\}$.

Ahora se probará que C es satisfactible si y sólo si el grafo G contiene un ciclo hamiltoniano.

- Supóngase que G tiene un ciclo hamiltoniano h. El ciclo h debe tener la siguiente forma:

  - Primero, este atraviesa la arista $\{b_{1,1},\, x'_1\}$ para ir del lado superior izquierdo al lado superior derecho.
  - Luego, h pasa por todos los vértices x'_m y x''_m de arriba a abajo, escogiendo la arista e_m o $\overline{e}_m$, pero no ambas.
  - Después atraviesa la arista $\{b_{k,4},\, x''_n\}$ para ir del lado inferior derecho al lado inferior izquierdo.
  - Finalmente, atraviesa los subgrafos B de abajo hacia arriba.

  Dado el ciclo hamiltoniano h, se define una asignación de verdad para C como sigue:
  Si la arista e_m pertenece a h, entonces a x_m se le asigna V. En caso contrario, la arista $\overline{e}_m$ pertenece a h, y entonces a x_m se le asigna F. Esta asignación de verdad satisface C. En efecto, se debe probar que todas las cláusulas en C son verdaderas. Considérese una cláusula c_i y su correspondiente subgrafo B en G. Cada arista $\{b_{i,j},\, b_{i,j+1}\}$ está conectada por un subgrafo A a una arista e_m o $\overline{e}_m$, dependiendo si x_m o $\overline{x}_m$ es el j-ésimo literal en la cláusula. Además, la arista $\{b_{i,j},\, b_{i,j+1}\}$ es atravesada por h si y sólo si el correspondiente literal es F, esto se puede ver de la siguiente forma: Si la arista $\{b_{i,j},\, b_{i,j+1}\}$ es atravesada por

h entonces h no atraviesa la arista (e_m o $\overline{e}_m$) con la que ella está conectada por medio de un subgrafo A. Si h no atraviesa a e_m (en este caso, el literal correspondiente es x_m), h debe pasar por $\overline{e}_m$ y de acuerdo con la asignación de verdad x_m es F. Si h no atraviesa $\overline{e}_m$ (en este caso, el literal correspondiente es $\overline{x}_m$), h debe pasar por e_m y de acuerdo con la asignación de verdad x_m es V y $\overline{x}_m$ es F.
Recíprocamente, si el literal correspondiente a la arista $\{b_{i,j}, b_{i,j+1}\}$ es F y éste es x_m entonces por la asignación de verdad, h pasa por $\overline{e}_m$ y como $\{b_{i,j}, b_{i,j+1}\}$ está conectada con e_m se obtiene que h atraviesa $\{b_{i,j}, b_{i,j+1}\}$.
Por otra parte, ya que las tres aristas $\{b_{i,1}, b_{i,2}\}$, $\{b_{i,2}, b_{i,3}\}$ y $\{b_{i,3}, b_{i,4}\}$ en la cláusula c_i no pueden ser todas atravesadas por h, una de las tres aristas debe tener un literal correspondiente cuyo valor asignado es V (por la equivalencia que se acaba de probar) y por lo tanto la cláusula c_i se satisface. Esta propiedad se cumple para todas las cláusulas en C y así se obtiene que C es satisfactible.

- Supóngase que C es satisfactible. Se puede construir un ciclo hamiltoniano para el grafo G así: El ciclo atraviesa la arista e_m si x_m es V, atraviesa la arista $\overline{e}_m$ si x_m es F, y atraviesa la arista $\{b_{i,j}, b_{i,j+1}\}$ si y sólo si el j-ésimo literal de la cláusula c_i es F bajo la asignación de verdad.

Finalmente, el grafo G puede ser construido en tiempo polinomial, este contiene un subgrafo B por cada una de las k cláusulas en C. Existe un subgrafo A por cada literal en cada cláusula en C, luego hay $3k$ subgrafos A. Puesto que los subgrafos A y B son de tamaño fijo, el grafo G tiene $\mathrm{O}(k)$ vértices y aristas. Así, G se puede construir en tiempo polinomial.
Por lo tanto, $3SAT \leq_p HC$.

3.6.2. El problema SUBSET-SUM

Para probar que el problema SUBSET-SUM es NP-completo se usará el hecho que problema EXACT COVER BY 3-SETS es NP-completo [GJ79]. Este problema se define como sigue:

EL PROBLEMA EXACT COVER BY 3-SETS

INSTANCIA:	Una familia $F = \{S_1, \ldots, S_n\}$ de subconjuntos de un conjunto U tal que $\lvert U \rvert = 3m$ para algún entero m y $\lvert S_i \rvert = 3$ para todo $i = 1, \ldots, n$.
PREGUNTA:	¿Existen m conjuntos en F disyuntos y que su unión sea U?

Teorema 3.9. El problema SUBSET-SUM es NP-completo. [Pap94]

Demostración. Por el ejemplo 3.10, SUBSET-SUM $\in$ NP.
Se probará que EXACT COVER BY 3-SETS $\leq_P$ SUBSET-SUM. Sea $\langle U, F \rangle$ una instancia de EXACT COVER BY 3-SETS donde $U = \{x_1, x_2, \ldots, x_{3m}\}$ para algún $m \in \mathbb{Z}^+$ y $F = \{S_1, S_2, \ldots, S_n\}$ es una familia de subconjuntos de U tal que $\lvert S_i \rvert = 3$, $i = 1, \ldots, n$. Se debe construir un conjunto $S = \{w_1, w_2, \ldots, w_n\}$ de enteros positivos y $t \in \mathbb{Z}^+$ tal que S tiene un subconjunto cuya suma es t si y sólo si existen m subconjuntos disyuntos cuya unión es U. A cada S_i se le asigna un vector de ceros y unos de longitud $3m$ de la siguiente manera:
Si $x_j \in S_i$ entonces se asigna 1 a la j-ésima componente del vector.
Si $x_j \notin S_i$ entonces se asigna 0 a la j-ésima componente del vector.
para todo $j = 1, \ldots, 3m$. Así, se obtiene un conjunto de n vectores donde cada vector es la representación de un entero en base $n+1$. Luego, cada conjunto S_i se transforma en el entero

$$w_i = \sum_{j \in S_i} (n+1)^{3m-j}.$$

Para completar la transformación se define

$$S = \{w_1, \ldots, w_n\} \quad y \quad t = \sum_{j=0}^{3m-1} (n+1)^j$$

donde t corresponde al entero de longitud $3m$ que contiene sólo unos, escrito en base $n+1$. Además, es fácil ver que esta transformación tiene tiempo de complejidad lineal en el tamaño de la entrada.

La Figura 3.8 ilustra la transformación de una instancia particular de EXACT COVER BY 3-SETS compuesta por $U = \{2, 4, 5, 7, 10, 11, 12, 15, 18, 19, 21, 22\}$ y $F = \{\{4, 7, 10\}, \{19, 21, 22\}, \{4, 12, 21\}, \{11, 12, 19\}, \{7, 10, 22\}, \{2, 5, 21\}, \{2, 7, 18\}, \{15, 18, 22\}\}$ en una instancia de SUBSET-SUM.

$$\begin{array}{lc cccccccccccc}
S_1 = \{4, 7, 10\} & \longrightarrow & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\
S_2 = \{19, 21, 22\} & \longrightarrow & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\
S_3 = \{4, 12, 21\} & \longrightarrow & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
S_4 = \{11, 12, 19\} & \longrightarrow & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} \\
S_5 = \{7, 10, 22\} & \longrightarrow & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
S_6 = \{2, 5, 21\} & \longrightarrow & \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} \\
S_7 = \{2, 7, 18\} & \longrightarrow & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
S_8 = \{15, 18, 22\} & \longrightarrow & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{1} \\
\hline
 & & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1
\end{array}$$

Figura 3.8: Transformación de una instancia particular de EXACT COVER BY 3-SETS en una instancia de SUBSET-SUM.

Ahora, sólo falta probar que $\langle U, F \rangle$ es una instancia "si" si y sólo si $\langle S, t \rangle$ es también una instancia "si":

1. Suponga que $\langle U, F \rangle$ es una instancia con respuesta SI, es decir, existen m subconjuntos de U (elementos de F) cuya unión es U. Se puede obtener una instancia de SUBSET-SUM con respuesta SI, tomando los m enteros correspondientes a los m elementos de F y sumándolos. Puesto que estos subconjuntos son disyuntos, los enteros correspondientes no tendrán 1 en la misma posición y, además,

ya que su unión es U, en todas las $3m$ posiciones habrá un 1 puesto que cada columna en la suma representa a un elemento de U. Así, se obtiene que la suma de estos m números es un vector de $3m$ unos, es decir, la suma es t.

2. Suponga que $\langle S, t\rangle$ es una instancia con respuesta SI, es decir, existe un subconjunto A de S tal que la suma de sus elementos es t. Se debe notar lo siguiente:

   a) Puesto que la representación de t en base $n+1$, es un vector de n unos, debe suceder que en cada columna de la suma de los elementos de A haya exactamente un 1 (ver Figura 3.8) ya que de lo contrario, si en una columna hay dos o mas unos, el vector suma tendría el número 2 o mayor en la posición correspondiente a dicha columna, lo cual implicaría que la suma no es t; o si en un columna no aparece ningún 1, el vector suma tendría en esa posición 0.

   b) Cada vector tiene exactamente tres unos repartidos en sus $3m$ posiciones.

   Finalmente, de (a) y (b) se obtiene que $|A| = m$ y los conjuntos correspondientes a los elementos de A son disyuntos y tienen a U como unión.

En esta sección se han demostrado algunos problemas NP-completos haciendo uso de la noción de transformación polinomial y de otros problemas que ya se conoce son NP-completos. A continuación se describirá otra técnica para probar NP-completitud denominada ***restricción***. La prueba por restricción es una de las técnicas más sencillas y más aplicable.

Probar por restricción que un problema dado $\Pi \in$ NP es NP-completo, consiste en mostrar que Π contiene como un caso particular un problema Π' que ya se conoce es NP-completo. Se debe observar que el problema restringido y el problema NP-completo conocido no requieren ser dos duplicados exactos sino que exista una correspondencia entre instancias SI y NO de tal forma que se preserven las respuestas.

Teorema 3.10. El problema de la Mochila (KNAPSACK) es NP-completo.

Demostración.

- Por el ejemplo 3.11, KNAPSACK $\in$ NP.
- KNAPSACK se puede restringir a SUBSET-SUM de la siguiente forma: Sea $v(u) = w(u)$ para todo $u \in U$ y K = W. Luego, el problema se reduce a tener un conjunto de enteros positivos $w_1, w_2, \ldots, w_n$ y otro entero K y pregunta si existe un subconjunto de los enteros dados que sumen exactamente K. Este problema obtenido coincide con el problema SUBSET-SUM, el cual por el Teorema 3.9, es NP-completo. Por lo anterior, el problema de la Mochila es NP-completo.

Considérese el problema:

EL PROBLEMA MINIMUM COVER (MC)

INSTANCIA:	Una colección C de subconjuntos de un conjunto S y un entero positivo K.
PREGUNTA:	¿C contiene una *cubierta* para S de tamaño K o menos, esto es, un subconjunto $C' \subseteq C$ con $\lvert C'\rvert \leq K$ y tal que $\bigcup_{c \in C'} c = S$?

Teorema 3.11. El problema MINIMUM COVER es NP-Completo.

Demostración. El problema MINIMUM COVER se puede restringir a EXACT COVER BY 3-SETS, considerando solamente las instancias de MC en las cuales $\lvert S\rvert = 3m$ para algún $m \in \mathbb{Z}^+$, $\lvert c\rvert = 3$ para todo $c \in C$ y $K = m$.

4. ALGORITMOS DE APROXIMACION PARA PROBLEMAS NP-DUROS

Muchos problemas de optimización se caracterizan porque su problema de decisión asociado es NP-completo y, como se ha visto, es improbable encontrar un algoritmo tiempo polinomial que lo resuelva; estos problemas de optimización se denominan NP-duros y conservan la dureza de sus correspondientes problemas de decisión. Esta dificultad para resolverlos eficientemente condujo a la búsqueda de algoritmos que retornen soluciones aproximadas a dichos problemas; estos algoritmos se denominan algoritmos de aproximación. Si la solución óptima es difícil de encontrar entonces usar un algoritmo de aproximación es una buena opción aunque se sacrifique precisión para obtener una solución aproximada que pueda ser computada eficientemente; más aún, lo que se busca es sacrificar tan poca precisión como sea posible y al mismo tiempo ganar eficiencia.

En este capítulo se presentan los conceptos básicos sobre problemas de optimización, las clases de aproximabilidad APX, PTAS y FPTAS. Finalmente se presentan algoritmos de aproximación para algunos problemas NP-duros.

4.1. FUNDAMENTOS: PROBLEMAS DE OPTIMIZACIÓN Y LAS CLASES NPO y PO

La Teoría de Aproximabilidad en problemas NP-duros estudia fundamentalmente aquellos problemas de optimización que cumplen ciertas propiedades, más precisamente, aquellos pertenecientes a la clase de problemas NPO.

Definición 4.1 (Clase NPO). Un *problema de optimización* $A = \langle I, \, sol, \, m, \, obj \rangle$ pertenece a la clase NPO si:

1. I es el conjunto de instancias de A y es reconocible en tiempo polinomial, esto es, existe un algoritmo tiempo polinomial que decide para cualquier entrada, si ésta corresponde a una instancia del problema.
2. Dada una instancia x de I, $sol(x)$ denota el conjunto de soluciones posibles de x. Estas soluciones deben ser cortas, esto es, existe un polinomio p tal que, para todo $y \in sol(x)$, $|y| \leq p(|x|)$. Además, para todo x y para todo y tal que $|y| \leq p(|x|)$, es decidible en tiempo polinomial si $y \in sol(x)$.
3. m es una función tal que para cualquier instancia x en I, $m(x, \, y) \in \mathbb{Z}^+$ representa el costo o valor de la solución $y \in sol(x)$. La función m es computable en tiempo polinomial y es llamada *función objetivo.*
4. $obj \in \{\text{máx}, \, \text{mín}\}$ determina si el problema es de maximización o minimización.

Resolver un problema de optimización $A = \langle I, \, sol, \, m, \, obj \rangle$ consiste en encontrar para cada $x \in I$, una solución $y \in sol(x)$ tal que

$$\begin{aligned} m(x, \, y) &= obj\{m(x, \, z) : z \in sol(x)\} \\ &= opt(x) \end{aligned}$$

Definición 4.2 (Clase PO). La clase PO es el conjunto de todos los problemas de optimización en NPO que tienen un algoritmo de tiempo polinomial que lo resuelve.

Ejemplo 4.1
EL PROBLEMA DEL AGENTE VIAJERO (MIN-AV)

INSTANCIA:	Un conjunto $C = \{c_1, c_2, \ldots, c_m\}$ de ciudades y una distancia $d(c_i, c_j) \in \mathbb{Z}^+$ para cada par de ciudades $c_i, c_j \in C$.
SOLUCION:	Un *tour* de todas las ciudades en C, esto es, un ordenamiento $\langle c_{\pi(1)}, c_{\pi(2)}, \ldots, c_{\pi(m)} \rangle$ donde π es una permutación $\pi : \{1, \ldots, m\} \to \{1, \ldots, m\}$.
MEDIDA:	La longitud del *tour*, esto es, $d(c_{\pi(m)}, c_{\pi(1)}) + \sum_{i=1}^{m-1} d(c_{\pi(i)}, c_{\pi(i+1)})$.
OBJETIVO:	Minimizar.

El problema MIN-AV está en NPO.

1. I es reconocible en tiempo polinomial puesto que dada una entrada para AV es suficiente chequear si C es un conjunto finito, si están dadas todas las distancias entre ciudades y si estas son enteros positivos. Esto se hace en tiempo cuadrático en el tamaño de la entrada.

2. Dada una instancia $\langle C, d \rangle$ de AV compuesta por un conjunto C de ciudades y la distancia entre ellas, $sol(\langle C, d \rangle)$ es el conjunto de todos los ordenamientos de la forma $\langle c_{\pi(1)}, c_{\pi(2)}, \ldots, c_{\pi(m)} \rangle$. Si $\langle c_{\pi(1)}, c_{\pi(2)}, \ldots, c_{\pi(m)} \rangle \in sol(\langle C, d \rangle)$, se tiene que $|\langle c_{\pi(1)}, c_{\pi(2)}, \ldots, c_{\pi(m)} \rangle| = |C| \leq |\langle C, d \rangle|$.
   Además, para todo $x \in I$ y y tal que $|y| \leq p(|x|) = |x|$, es decidible en tiempo polinomial si $y \in sol(x)$, chequeando si $|y| = |C|$ y si corresponde a un ordenamiento de la forma $\langle c_{\pi(1)}, c_{\pi(2)}, \ldots, c_{\pi(m)} \rangle$.

3. Para cualquier instancia $\langle C, d \rangle$ de AV, la medida $m(\langle C, d \rangle, y)$ es la longitud del *tour* y, la cual es computable en tiempo polinomial.

4. $obj = \text{mín}$ $\diamondsuit$

Ejemplo 4.2

EL PROBLEMA DEL AGENTE VIAJERO MÉTRICO (MIN-AVM)

INSTANCIA:	Un conjunto $C = \{c_1, c_2, \ldots, c_m\}$ de ciudades y una distancia $d(c_i, c_j) \in \mathbb{Z}^+$ para cada par de ciudades $c_i, c_j \in C$ que satisface la desigualdad triangular.
SOLUCIÓN:	Un *tour* de C.
MEDIDA:	La longitud del *tour*.
OBJETIVO:	Minimizar.

De igual forma que en el ejemplo 4.1, se obtiene que el problema MIN-AVM está en NPO. $\diamondsuit$

Ejemplo 4.3

EL PROBLEMA MAX SUBSET-SUM

INSTANCIA:	Un conjunto $S = \{x_1, x_2, \ldots, x_n\}$ donde $x_i \in \mathbb{Z}^+$ para todo $i = 1, \ldots, n$ y un valor $t \in \mathbb{Z}^+$.
SOLUCIÓN:	Un subconjunto A de S tal que la suma de sus elementos sea menor o igual que t.
MEDIDA:	$\sum_{a \in A} a$.
OBJETIVO:	Maximizar.

El problema MAX SUBSET-SUM está en NPO:

1. I es reconocible en tiempo polinomial puesto que dada una instancia $\langle S, t\rangle$ de I, es suficiente chequear si S es un conjunto finito de valores positivos y si $t \in \mathbb{Z}^+$; esto se hace en tiempo lineal en la longitud de la entrada.

2. Dada una instancia $\langle S, t\rangle$ de I, $sol(\langle S, t\rangle)$ es el conjunto de todos los subconjuntos de S que cumplen que la suma de sus elementos es menor o igual que t. Si $A \in sol(\langle S, t\rangle)$, entonces $|A| \leq |S| \leq |\langle S, t\rangle|$, luego $|A| \leq |\langle S, t\rangle|$ $(p(x) = x)$. Además, para todo $\langle S, t\rangle$ y para todo A que cumplan $|A| \leq p(|\langle S, t\rangle|) = \langle S, t\rangle$ es decidible en tiempo polinomial si $A \in sol(\langle S, t\rangle)$ ya que sólo se necesita chequear

que $A \subseteq S$ y si la suma de sus elementos es menor o igual que t.

3. Puesto que la medida es la suma de los elementos de $A \subseteq S$ se tiene que $m(\langle S, t\rangle, y)$ es calculable en tiempo polinomial en el tamaño de la entrada.

4. $obj = \text{máx}$. $\diamondsuit$

Ejemplo 4.4

EL PROBLEMA MAX3SAT

INSTANCIA:	Una colección $C = \{c_1, c_2, \ldots, c_m\}$ de cláusulas sobre un conjunto finito U de variables tales que $\lvert c_i\rvert = 3$ para $1 \leq i \leq m$.
SOLUCIÓN:	Una asignación de verdad para U.
MEDIDA:	Número de cláusulas que se satisfacen con la asignación de verdad.
OBJETIVO:	Maximizar.

4.2. PROBLEMAS DE DECISIÓN SUBYACENTES

La teoría desarrollada en los capítulos anteriores, como se observó, sólo se aplica a problemas de decisión y por lo tanto no puede ser aplicada directamente a problemas de optimización. Sin embargo, a cada problema de optimización se le puede asociar un problema de decisión como se indica a continuación.

Definición 4.3 (Problema de decisión subyacente). El problema de decisión subyacente de un problema de optimización $A = \langle I, sol, m, obj\rangle$ en NPO es el siguiente:

INSTANCIA:	$x \in I$, $k \in \mathbb{Z}$.
PREGUNTA:	¿Existe $y \in sol(x)$, tal que $m(x, y) \leq k$? (si $obj = \text{mín}$).
	¿Existe $y \in sol(x)$, tal que $m(x, y) \geq k$? (si $obj = \text{máx}$).

Ejemplo 4.5
El problema de decisión subyacente de MIN-AV es AV (ejemplo 3.9).

Ejemplo 4.6
El problema de decisión subyacente de MAX SUBSET-SUM es SUBSET-SUM (ejemplo 3.10).

Ejemplo 4.7
El problema de decisión subyacente de MAX3SAT es 3SAT (subsección 3.6.1). $\diamondsuit$

Lema 4.1. Si un problema de optimización pertenece a NPO, entonces el problema de decisión subyacente pertenece a NP. [Mac99]

Demostración. Sea $A = \langle I,\, sol,\, m,\, obj \rangle$ un problema de optimización en NPO y sea $\langle x, k \rangle$ una instacia del problema de decisión subyacente. Se mostrará un algoritmo tiempo polinomial V que verifica el problema de decisión subyacente. El algoritmo V recibe como parámetros a $\langle x, k \rangle$ y un posible certificado c. Para que V acepte a $\langle x, k \rangle$ se debe dar lo siguiente:
$|c| \leq p\,(\,|x|\,)$, $c \in sol(x)$ y $m(x, c) \leq k$ (si $obj =$ mín) o $m(x, c) \geq k$ (si $obj =$ máx).
El algoritmo trabaja como sigue:
Si $|c| \leq p(\,|x|\,)$ entonces es decidible en tiempo polinomial si $c \in sol(x)$ puesto que A pertenece a NPO. A continuación, el algoritmo calcula $m(x, c)$, lo compara con k y decide si c es una solución del problema de decisión subyacente. Todo esto se hace en tiempo polinomial. En cualquier otro caso el algoritmo rechaza la entrada.

Lema 4.2. Si un problema de optimización pertenece a PO entonces el problema subyacente pertenece a P. [Mac99]

Demostración. Sea $A = \langle I,\, sol,\, m,\, obj \rangle$ un problema de optimización en PO. Por hipótesis, para toda instancia x, el costo óptimo $opt(x)$ del problema puede calcularse

en tiempo polinomial. Si $\langle x, k \rangle$ es una entrada para el problema de decisión subyacente, entonces es posible comparar $opt(x)$ con k, lo cual tambien se puede hacer en tiempo polinomial. Por lo anterior, el problema subyacente pertenece a P.

Lema 4.3. Si P $\neq$ NP, entonces cualquier problema $A = \langle I, sol, m, obj \rangle$ en NPO cuyo problema subyacente es NP-completo no pertenece a PO. [Mac99]

Demostración. Sea L el problema de decisión subyacente del problema de optimización A. Puesto que L es NP-completo y P $\neq$ NP, entonces L no pertenece a P y por el lema anterior se tiene que A no pertenece a PO.

Corolario 4.1. Si P $\neq$ NP entonces PO $\neq$ NPO.

4.3. CLASES DE APROXIMACIÓN

Cuando se intenta aproximar la solución de un problema de optimización, se encuentran diversos comportamientos en dichos problemas; en algunos, por ejemplo, la solución puede aproximarse a la óptima tanto como se desee en tiempo polinomial aunque cuando la solución se acerca a la óptima el tiempo de ejecución del algoritmo aumenta rápidamente, estos algoritmos se llaman esquemas de aproximación tiempo polinomial. En otros problemas, cuando el error que se comete disminuye, el tiempo del algoritmo crece pero en forma más lenta, a estos se les llama esquemas de aproximación totalmente polinomiales. Existen problemas cuya solución se ha aproximado en un factor constante pero hasta el momento no se ha podido mejorar dicho factor, puesto que se ha probado que reducir esta constante es en sí un problema NP-duro. En casos peores, se puede estar enfrentando un problema que simplemente no es aproximable lo que ha levado a buscar criterios que indiquen si la aproximación es o no posible. Para probar

que un problema no es aproximable se ha desarrollado una extensa teoría y varios resultados importantes llamados resultados de no aproximabilidad.

Definición 4.4 (Factor de aproximación). Sea $A = \langle I, \, sol, \, m, \, obj \rangle$ un problema en NPO. Dada una instancia $x \in I$ y una posible solución $y \in sol(x)$, se define el factor de aproximación de y con respecto a x como:

$$R(x, y) = \text{máx}\left\{\frac{m(x,\, y)}{opt(x)}, \frac{opt(x)}{m(x,\, y)}\right\}$$

Para problemas de minimización, $0 < opt(x) \leq m(x, \, y)$, lo que implica que

$$R(x,\, y) = \frac{m(x,\, y)}{opt(x)} \geq 1.$$

y para problemas de maximización, $0 < m(x, \, y) \leq opt(x)$, lo que implica que

$$R(x,\, y) = \frac{opt(x)}{m(x,\, y)} \geq 1.$$

Este factor de aproximación siempre es un número mayor o igual que 1 y es cercano a 1 cuando y es cercano a la solución óptima.

Definición 4.5 (Algoritmo *r(n)*-aproximado). Sea $A = \langle I, \, sol, \, m, \, obj \rangle$ un problema en NPO y sea T un algoritmo tal que, para toda instancia $x \in I$, retorna una solución factible $T(x) \in sol(x)$. Dada una función arbitraria $r : \mathbb{N} \to \mathbb{Q}^+$, se dice que T es un algoritmo $r(n)$-aproximado para A, si para cualquier instancia x, el factor de aproximación de la solución $T(x)$ con respecto a x verifica la siguiente desigualdad:

$$R(x,\, T(x)) \leq r(|x|).$$

Si un problema $A \in$ NPO admite un algoritmo $r(n)$-aproximado y que corre en tiempo polinomial se dice que A es $r(n)$-*aproximable.*

4.3.1. La clase APX

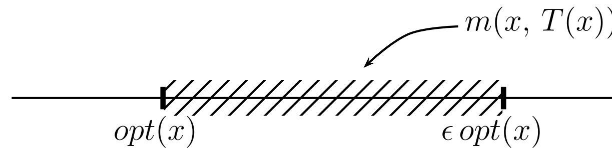
Definición 4.6 (Clase APX). Sea A un problema en NPO. A pertenece a la clase APX si es ϵ-aproximable para alguna constante $\epsilon > 1$.

En otras palabras, que A pertenezca a APX significa, con respecto a las soluciones que genera el algoritmo T, que:

Si A es un problema de minimización entonces

$$R(x, T(x)) = \frac{m(x, T(x))}{opt(x)} \leq \epsilon$$

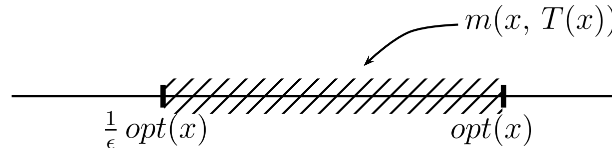
y en consecuencia, $m(x, T(x)) \leq \epsilon\, opt(x)$. Esto significa que la solución dada por T es a lo más ϵ veces la óptima, con $\epsilon > 1$.



y, si A es un problema de maximización, entonces

$$R(x, T(x)) = \frac{opt(x)}{m(x, T(x))} \leq \epsilon$$

y por lo tanto, $\frac{1}{\epsilon}\, opt(x) \leq m(x, T(x))$, con $\epsilon > 1$. Esto significa que la solución dada por T es por lo menos $\frac{1}{e}$ veces la óptima.



Sin embargo, no todo problema en NPO es ϵ-aproximable para alguna constante ϵ. El siguiente teorema demuestra esta afirmación condicionada a que P $\neq$ NP.

Teorema 4.1. Si P $\neq$ NP entonces existe un problema en NPO que no es aproximable. [CLR90]

Demostración. Se probará que:
Si P $\neq$ NP y $\epsilon > 1$, no existe un algoritmo ϵ-aproximado para el problema del Agente Viajero (AV).

Supóngase por contradicción que, para algún $\epsilon > 1$, existe un algoritmo ϵ-aproximado A tiempo polinomial para AV y que ϵ es un entero (aproximando por arriba en caso de que no lo sea). Se mostrará como usar A para resolver instancias del problema del ciclo hamiltoniano (HC) (presentado en el ejemplo 3.6) en tiempo polinomial; lo cual lleva a una contradicción ya que HC es NP-completo según el Teorema 3.8, y resolverlo en tiempo polinomial implicaría que P = NP.

Sea $G = (V, E)$ una instancia de HC. Se desea determinar eficientemente si G contiene un ciclo hamiltoniano usando el algoritmo de aproximación A. Se transforma G en una instancia del AV como sigue:
Sea $G' = (V, E')$ el grafo completo con conjunto de vértices V; esto es, $E' = \{\{u, v\} : u, v \in V \text{ y } u \neq v\}$. Se asigna un costo entero a cada arista en E' como sigue:

$$c(u, v) = \begin{cases} 1 & \text{si } \{u, v\} \in E \\ \epsilon\,|V| + 1 & \text{en otro caso} \end{cases}$$

Observese que G' y c se obtienen de G en tiempo polinomial en $|V|$ y $|E|$.

Ahora, se considera la instancia del problema del Agente Viajero (G', c). Si el grafo original G tiene un ciclo hamiltoniano H, entonces la función de costo c asigna a cada arista de H un costo igual a 1, y así (G', c) contiene un *tour* de costo $|V|$. Por otra parte, si G no contiene un ciclo hamiltoniano, entonces cualquier *tour* de G' debe usar alguna arista que no está en E. Pero cualquier *tour* que use alguna arista que no está en E tiene un costo de al menos $(\epsilon\,|V| + 1) + (|V| - 1) > \epsilon|V|$.

Por lo tanto, existe una gran diferencia entre el costo de un *tour* que es ciclo hamiltoniano en G (su costo es $|V|$) y el costo de cualquier otro *tour* (su costo es mayor que $\epsilon\,|V|$).

Se aplica el algoritmo de aproximación A al problema del Agente Viajero (G', c). Puesto que el costo del *tour* retornado por A es a lo más ϵ veces el costo del *tour* óptimo, si G contiene un ciclo hamiltoniano entonces A debe retornar un *tour* de costo $|V|$ que es el ciclo hamiltoniano en G. En efecto: si esto no fuera así entonces el *tour* de menor costo que puede retornar A que no es ciclo hamiltoniano en G es $(\epsilon\,|V| + 1) + (|V| - 1)$ y esto es mayor que $\epsilon\,|V|$, lo cual contradice el hecho que A es un algoritmo ϵ-aproximado para AV.

Si G no contiene un ciclo hamiltoniano, entonces A retorna un *tour* de costo mayor que $\epsilon\,|V|$. Por lo tanto, se puede usar A para resolver el problema del ciclo hamiltoniano en tiempo polinomial.

Corolario 4.2. Si P $\neq$ NP entonces APX $\neq$ NPO.

4.3.2. La clase PTAS

Definición 4.7 (Esquema de aproximación). Sea A un problema en NPO. Un algoritmo T es un esquema de aproximación para A si, para toda instancia x de A y para cualquier racional $\epsilon > 1$, T(x, ϵ) retorna una solución posible de x cuyo factor de aproximación es a lo más ϵ.

Definición 4.8 (Clase PTAS). Un problema A en NPO pertenece a la clase PTAS si admite un esquema de aproximación tiempo polinomial, esto es, un esquema de apro-

ximación cuyo tiempo de complejidad es acotado por $q(|x|)$ donde q es un polinomio en el tamaño de x.

Sin embargo, se observa que el tiempo de complejidad de un esquema de aproximación en la definición anterior puede ser del tipo $2^{1/(\epsilon-1)}\,p\,(|x|)$ o $|x|^{1/(\epsilon-1)}$ donde p es un polinomio. Así, las computaciones con valores de ϵ muy cercanos a 1 podrían no generar soluciones en forma eficiente ya que el tiempo se hace muy grande para dichos valores.

Teorema 4.2. Si P $\neq$ NP entonces existe un problema en APX que no pertenece a PTAS. [Hou97]

Antes de presentar la prueba de este teorema, es necesario proporcionar algunos nuevos elementos. Uno de ellos es la definición de un nuevo verificador que al igual que el verificador presentado en la sección 3.2 toma como entrada una cadena x y un certificado, que para el nuevo verificador se llamará prueba, y además una cadena de *bits* aleatorios denominada cadena de *bits random*. También se presentará un teorema reciente que ha tenido muchas aplicaciones, sobre todo ha sido utilizado para probar muchos resultados de no aproximabilidad, tal como el teorema 4.2.

Definición 4.9. (Verificador (log n, 1)-restringido) Un verificador $(\log n, 1)-$ restringido V es una máquina de Turing (probabilística) tiempo polinomial que tiene acceso a una entrada x, una cadena r compuesta de $O(\log|x|)$ *bits random* y una *prueba* Π. Dependiendo de la entrada x, la cadena *random* r y la *prueba* Π, el verificador V aceptará o rechazará la entrada x. Aunque el verificador tiene acceso completo a la entrada x, el acceso a la *prueba* Π es muy limitado, V solamente examina un número constante de *bits* de Π. Esto lo hace de la siguiente forma: sobre la entrada x y con la cadena $r \in \{0,\, 1\}^{\lceil c \log|x| \rceil}$ para algún $c > 0$, el verificador computa un conjunto finito de enteros $Q(x,\, r) = \{i_1, \ldots, i_k\}$ donde k es un entero fijo y $|i_j| \leq |\Pi| \leq p(|x|)$ para $j = 1, \ldots, k$. Luego, el i_j-ésimo símbolo de Π, Π_{i_j}, es leído y escrito sobre una cinta del verificador, el resto de la *prueba* Π no es necesaria. El proceso de leer un *bit* de Π

es llamado una *pregunta*. Finalmente, el verificador ejecuta una computación tiempo polinomial sobre x, r y $\Pi_{i_1}, \ldots, \Pi_{i_k}$.

Así, un verificador $(\log n, 1)$-restringido es un verificador que para entradas de longitud n usa a lo más $O(\log n)$ *bits random* y *pregunta* a lo más $O(1)$ *bits* de la *prueba* Π.

Un verificador $(\log n, 1)$-restringido decide un lenguaje L si:

- Si $x \in L$ entonces existe una *prueba* Π tal que para todas las cadenas *random* el verificador finaliza con "si".
- Si $x \notin L$ entonces para toda *prueba* Π se cumple que al menos $1/2$ de las cadenas *random* llevan al verificador a responder "no".

El siguiente teorema denominado Teorema PCP (*Probabilistically Checkable Proofs*) proporciona una nueva caracterización de la clase NP.

Teorema 4.3 (Teorema PCP). Un lenguaje L está en NP si y sólo si L es decidible por un verificador $(\log n, 1)$-restringido. [Pap94]

Demostración del Teorema 4.2. Se probará que, si $\mathrm{P} \neq \mathrm{NP}$ entonces MAX3SAT $\in$ APX [CK95] no admite un PTAS. La prueba se realizará por contradicción mostrando que la existencia de un PTAS para MAX3SAT implica $\mathrm{P} = \mathrm{NP}$.

Sea L un lenguaje arbitrario de NP y sea V el verificador $(\log n, 1)$-restringido para L cuya existencia está garantizada por el Teorema PCP.

Para cualquier entrada x se usará el verificador V para construir una instancia $\mathcal{S}_x$ de 3SAT tal que $\mathcal{S}_x$ es satisfactible si y sólo si x es un elemento de L. Además si

x no pertenece a L entonces a lo más una fracción constante de las cláusulas en $\mathcal{S}_x$ se pueden satisfacer simultáneamente. Por consiguiente, un PTAS para MAX3SAT podría ser usado para decidir si x está en L o no y el lenguaje L puede ser reconocido en tiempo polinomial, es decir P = NP.

Ahora, se describirá cómo construir la instancia $\mathcal{S}_x$ de 3SAT para una entrada dada x usando el verificador V. La *prueba* requerida por V se denota como una secuencia x_1, x_2, $x_3, \ldots$ de *bits*, donde el i-ésimo *bit* de la prueba es representado por la variable x_i. Para una cadena *random* r el verificador V requerirá $q = O(1)$ *bits* de la *prueba* los cuales se denotarán b_{r_1}, $b_{r_2}, \ldots, b_{r_q}$. El verificador V aceptará la entrada x para la cadena *random* r si los *bits* b_{r_1}, $b_{r_2}, \ldots, b_{r_q}$ tienen los valores correctos. Tomando subconjuntos de a lo más q de estos *bits* se puede construir una fórmula q-SAT $\mathcal{F}_r$ que contiene a lo más 2^q cláusulas y de modo que $\mathcal{F}_r$ es satisfactible si y sólo si existe una *prueba* Π tal que el verificador V acepta la entrada x sobre la cadena *random* r. Esta fórmula q-SAT $\mathcal{F}_r$ puede ser transformada en una fórmula 3SAT $\mathcal{F}'_r$ equivalente que contiene a lo más $q\,2^q$ cláusulas; esto debido a que en la transformación el número de cláusulas de tres literales que aparecen depende del número de literales que hay en cada cláusula de SAT y el máximo número de literales en una cláusula es q y, puesto que hay 2^q cláusulas, en el peor caso resultarán $q\,2^q$ cláusulas de tres literales y posiblemente algunas nuevas variables. Se define $\mathcal{S}_x$ como la conjunción de todas las fórmulas $\mathcal{F}'_r$ para todas las posibles cadenas *random* r.

1. Si x es un elemento de L entonces por la definición de verificador restringido existe una *prueba* Π tal que V acepta x para cualquier cadena *random* r. Así, la fórmula $\mathcal{S}_x$ es satisfactible.

2. Si x no es un elemento de L entonces para cada *prueba* Π el verificador V acepta x para a lo más $1/2$ de todas las posibles cadenas *random* r. Esto significa que a lo más la mitad de todas las fórmulas $\mathcal{F}'_r$ se satisfacen simultáneamente. El número de cláusulas en $\mathcal{S}_x$ es a lo más $N_f\,q\,2^q$, donde N_f es el número de fórmulas $\mathcal{F}'_r$. Luego, al menos $\frac{1}{2}\,N_f$ fórmulas $\mathcal{F}'_r$ no se satisfacen y por lo tanto el menor número de cláusulas en $\mathcal{S}_x$ que no se satisfacen es $\frac{1}{2}\,N_f$ puesto que basta que una cláusula sea falsa para que la fórmula no sea satisfactible. Así, la fracción

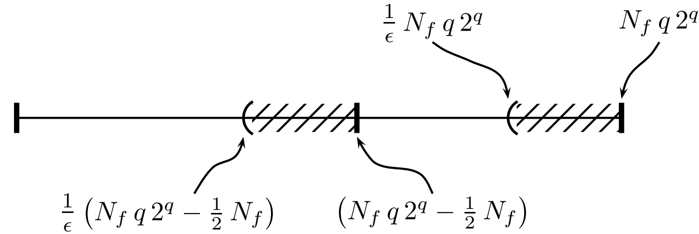
de cláusulas que no se satisfacen de $\mathcal{S}_x$ es al menos:

$$\frac{\frac{1}{2}\,N_f}{N_f\,q\,2^q} = \frac{1}{2\,q\,2^q} \tag{4.1}$$

Sea A_ϵ un PTAS para MAX3SAT, el cual recibe como entrada a $\mathcal{S}_x$ y un $\epsilon > 1$ apropiado tal que:

$$\frac{1}{\epsilon}\,N_f\,q\,2^q > N_f\,q\,2^q - \frac{1}{2}\,N_f \tag{4.2}$$

Esta desigualdad garantiza que los dos intervalos que se muestran en el siguiente gráfico no se intersecten, lo cual es necesario para que A_ϵ pueda decidir el lenguaje L.



Por definición de PTAS se tiene que:

$$A_\epsilon(S_{\,x}) > \frac{1}{\epsilon}\,OPT(x).$$

Además:

- Si $OPT(x) = N_f q 2^q$ entonces el mayor número de cláusulas que se satisfacen son todas las cláusulas de $\mathcal{S}_x$, es decir, $\mathcal{S}_x$ es satisfactible y por (2) se obtiene que $x \in L$.

- Si $OPT(x) = N_f\,q\,2^q - \frac{1}{2}\,N_f$ entonces el mayor número de cláusulas que se satisfacen en $\mathcal{S}_x$ es $N_f\,q\,2^q - \frac{1}{2}\,N_f$, es decir, $\mathcal{S}_x$ no es satisfactible y por (1) se obtiene que $x \notin L$.

Por consiguiente, A_ϵ puede ser usado para decidir L de la siguiente forma: si $A_\epsilon(S_{\,x}) > \frac{1}{\epsilon}N_f q 2^q$ entonces $x \in L$, en caso contrario $x \notin L$.

Finalmente se ha construido un algoritmo que decide L en tiempo polinomial, lo cual implica que P = NP.

Corolario 4.3. Si P $\neq$ NP entonces PTAS $\neq$ APX.

4.3.3. La clase FPTAS

Definición 4.10 (Clase FPTAS). Un problema A en NPO pertenece a la clase FPTAS si admite un esquema de aproximación totalmente polinomial, esto es, un esquema de aproximación cuyo tiempo de complejidad es acotado por $q\left(|x|, \frac{1}{(\epsilon-1)}\right)$ donde q es un polinomio tanto en $|x|$ como en $\frac{1}{(\epsilon-1)}$.

Teorema 4.4. Si P $\neq$ NP entonces existe un problema en PTAS que no pertenece a FPTAS.[BC94]

Corolario 4.4.

$$FPTAS \subseteq PTAS \subseteq APX \subseteq NPO$$

y las inclusiones son estrictas si y sólo si P $\neq$ NP.

4.4. ALGORITMOS DE APROXIMACIÓN PARA ALGUNOS PROBLEMAS NP-DUROS

En esta sección se presentan algunos problemas para los cuales se han desarrollado algoritmos de aproximación tiempo polinomial. Dependiendo de las características de

estos algoritmos, estos problemas se clasifican en las clases APX, PTAS o FPTAS. También, se describen la estrategia *shifting* y la estrategia *grid* cuya aplicación proporcionan un PTAS para el problema de Cobertura con discos en tiempo lineal.

4.4.1. El problema del AGENTE VIAJERO METRICO está en APX

En el Teorema 4.1 se demostró que el problema del Agente Viajero no puede ser aproximado con un factor de aproximación constante. Sin embargo, restringiendo el problema a que la función de costo c satisfaga la desigualdad triangular, es decir, si para todo vértice u, v, w en V: $c(u,w) \leq c(u,v) + c(v,w)$, entonces el problema puede ser aproximado con un factor de aproximación 2. Esto significa que el problema del Agente Viajero Métrico pertenece a APX.

Teorema 4.5. El problema del AGENTE VIAJERO MÉTRICO $\in$ APX. [CLR90]

Demostración. Para desarrollar un algoritmo de aproximación para el Agente Viajero Métrico se modelará el problema como un grafo completo no dirigido $G = (V, E)$, donde el conjunto V de vértices representa el conjunto de ciudades y un costo entero no negativo asociado a cada arista $\{u, v\} \in E$ que representa la distancia entre cada par de ciudades.

El algoritmo APROX-AVM computa un *tour* aproximado de un grafo no dirigido G, usando el algoritmo PRIM que retorna el árbol de recubrimiento mínimo para G. Estos algoritmos se presentan a continuación.

función PRIM$(G = (V, E), c, r)$: conjunto de aristas

$T \leftarrow \emptyset$

$B \leftarrow r$

mientras $B \neq V$ **hacer**

  buscar $e = \{u, v\}$ de longitud mínima tal que $u \in B$ y $v \in V - B$

  $T \leftarrow T \cup \{e\}$

  $B \leftarrow B \cup \{v\}$

devolver T

APPROX-AVM (G, c)

1. Selecciona un vértice $r \in V$ para ser el vértice "raiz".
2. Desarrolla un árbol de recubrimiento mínimo T para G desde la raiz r, usando PRIM(G, c, r).
3. L es la lista de vértices visitados en un recorrido ordenado del árbol T.
4. Retorna el ciclo hamiltoniano H que visita los vértices en el orden L.

Se mostrará que cuando la función de costo c satisface la desigualdad triangular, el *tour* que este algoritmo retorna no es mayor que dos veces la longitud de un *tour* óptimo. En el ejemplo 4.6 se muestra una aplicación del algoritmo APROX-AVM.

Un *recorrido ordenado* de un árbol visita todos los vértices en el árbol, listando un vértice cuando es encontrado por primera vez, antes de que cualquiera de sus ramas sea visitada.

Ahora se probará que APPROX-AVM es un algoritmo 2-aproximado para el problema del Agente Viajero Métrico.

Sea H^* un *tour* óptimo para el grafo $G = (V, E)$ dado. Se debe probar que $m(G, H) \leq 2\, m(G, H^*)$, donde H es el *tour* retornado por APPROX-AVM. Puesto que se puede obtener un árbol de recubrimiento borrando cualquier arista de un *tour*,

si T es un árbol de recubrimiento mínimo para el conjunto de vértices dado, entonces

$$m(G,\, T) \leq m(G,\, H^*) \tag{4.3}$$

Un *recorrido completo* de T lista los vértices cuando ellos son visitados por primera vez y tambien cada vez que se regresa a ellos después de haber visitado un sub-árbol. Sea W este recorrido. Puesto que el recorrido completo visita cada arista de T exactamente dos veces, entonces

$$m(G,\, W) = 2\, m(G,\, T) \tag{4.4}$$

De (4.3) y (4.4) se tiene

$$m(G,\, W) \leq 2\, m(G,\, H^*) \tag{4.5}$$

y así, la longitud de W es menor o igual que dos veces la longitud de un *tour* óptimo.

Desafortunadamente, W no es generalmente un *tour* ya que éste visita algunos vértices más de una vez. Sin embargo, por la desigualdad triangular se puede borrar una visita a cualquier vértice y la longitud no se incrementa. Repitiendo este proceso, se pueden remover de W todas, menos las primeras visitas a cada vértice.

El ordenamiento de vértices obtenido haciendo uso de un recorrido completo y quitando de él todas las visitas repetidas es exactamente el mismo que se obtiene si se halla un camino ordenado del arbol T, que es el calculado por APROX-AVM. Sea H el ordenamiento correspondiente a este recorrido ordenado. H es en realidad un *tour* puesto que todos los vértices son visitados exactamente una vez.

Por la desigualdad triangular y puesto que H es obtenido borrando vértices del recorrido W se tiene que:

$$m(G,\, H) \leq m(G,\, W) \tag{4.6}$$

Finalmente de (4.5) y (4.6) se obtiene que

$$m(G,\, H) \leq 2\, m(G,\, H^*)$$

Ejemplo 4.8

Una instancia del problema del Agente Viajero Métrico puede ser representada mediante un grafo completo G donde el conjunto de vértices está formado por los puntos a, b, c, d, e, f, g los cuales están ubicados sobre una grilla entera donde la distancia entre los vértices es la distancia euclidea. En la Figura 4.1 se muestra una operación del procedimiento APROX-AVM. Primero, se computa un árbol de recubrimiento mínimo T de estos puntos usando el algoritmo de PRIM donde a es el vértice raíz (Figura 4.1(b)). Luego, se calcula un recorrido completo de T, comenzando en a, el cual visita los vértices en el orden $a, d, e, g, e, b, c, f, c, b, e, d, a$ (Figura 4.1(c)). Usando la desigualdad triangular se obtiene un recorrido ordenado de T: a, d, e, g, b, c, f. Finalmente, un *tour* de los vértices es obtenido visitando los vértices en el orden dado por el recorrido ordenado. Este es el *tour* H retornado por APROX-AVM y su costo total es aproximadamente 28.61 (Figura 4.1(d)). Un *tour* óptimo H^* para el conjunto de puntos dados es mostrado en la Figura 4.1(e), su costo total es aproximadamente 22.93. $\diamondsuit$

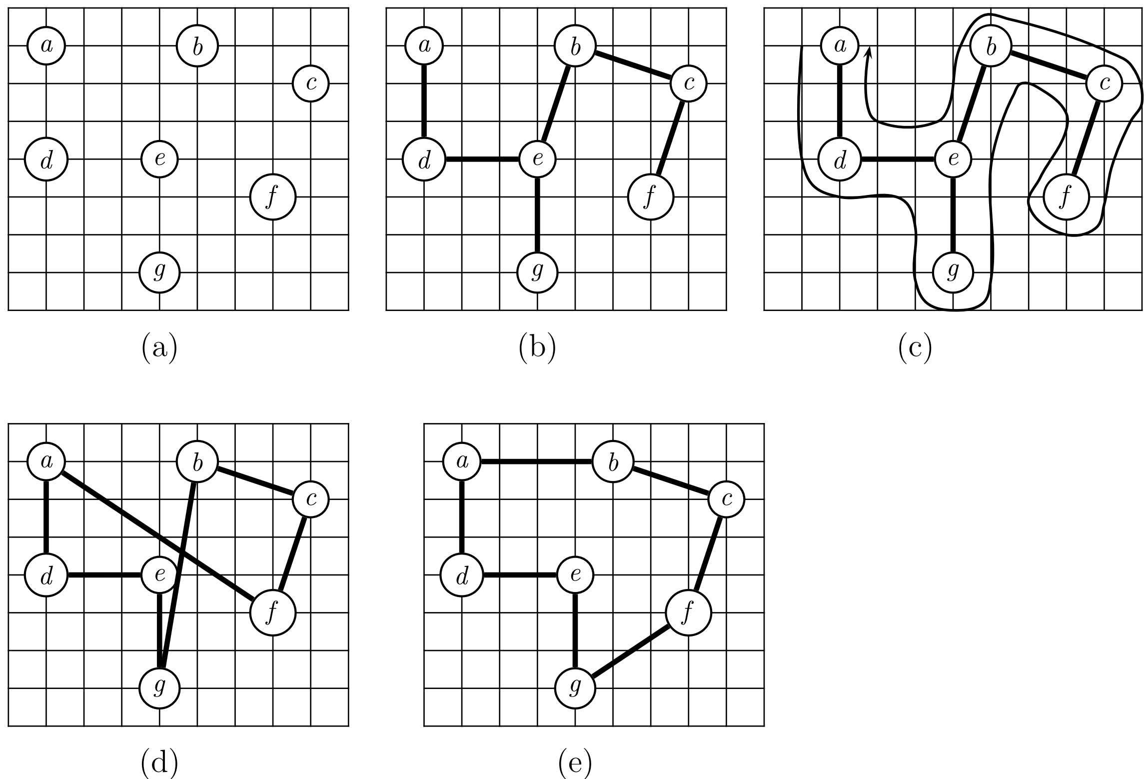


Figura 4.1: Una aplicación de APROX-AVM.

4.4.2. El problema de Cobertura con discos está en PTAS

El problema de Cobertura con discos es un problema geométrico que se define como sigue:

INSTANCIA:	Un conjunto de puntos en el plano y un diámetro $D \in \mathbb{Z}^+$.
SOLUCION:	Una *cobertura de discos*, esto es, un conjunto de discos de radio $r = D/2$ que cubran todos los puntos dados.
MEDIDA:	La cardinalidad de la cobertura de discos.
OBJETIVO:	Minimizar.

Para desarrollar un esquema de aproximación tiempo polinomial para este problema se utiliza fundamentalmente una aplicación simple de la técnica "*divide y vencerás*", denominada la estrategia $shifting$.

La estrategia *shifting*

A continuación se ilustra la aplicación de la estrategia $shifting$ para cubrir puntos en el plano con discos de radio fijo. Sin embargo, el método puede ser aplicado para cubrir puntos en el espacio d-dimensional con cualquier tipo de objetos de una forma arbitraria fija. Sea n el número de puntos dados en el plano, el objetivo es cubrir estos puntos con un número mínimo de discos de diámetro D. Sea l el parámetro $shifting$. El algoritmo $Shift$ $\mathcal{S}$ (A), definido para un algoritmo local dado A, trabaja como sigue: primero, se subdivide el plano en bandas verticales de ancho D y se consideran grupos de l bandas consecutivas, obteniendo así bandas de ancho lD (Figura 4.2). Sea A un algoritmo que retorna una solución en cualquier banda de ancho lD. Aplicando el algoritmo A a cada una de las bandas de ancho lD y luego considerando la unión de todos los discos usados, se encuentra una solución factible. Después se desplazan todas

las bandas de ancho lD una longitud D y se repite el mismo procedimiento. Puesto que cada banda considerada tiene ancho lD, se pueden realizar en total l desplazamientos, resultando l soluciones factibles de las cuales el algoritmo $shift$ escoge la de mínima cardinalidad.

Sean $S_1, \ldots, S_l$ las particiones del plano en bandas de ancho lD obtenidas en cada uno de los l desplazamientos respectivamente (Figura 4.2). Para una partición dada S_i, se denotará por $A(S_i)$ el algoritmo que aplica el algoritmo A a cada banda en la partición S_i y retorna la unión de todos los discos usados. Así, el algoritmo $shift$ $\mathcal{S}(A)$ retorna el conjunto de discos de cardinalidad mínima entre los l conjuntos retornados por $A(S_1)$,$\ldots$, $A(S_l)$.

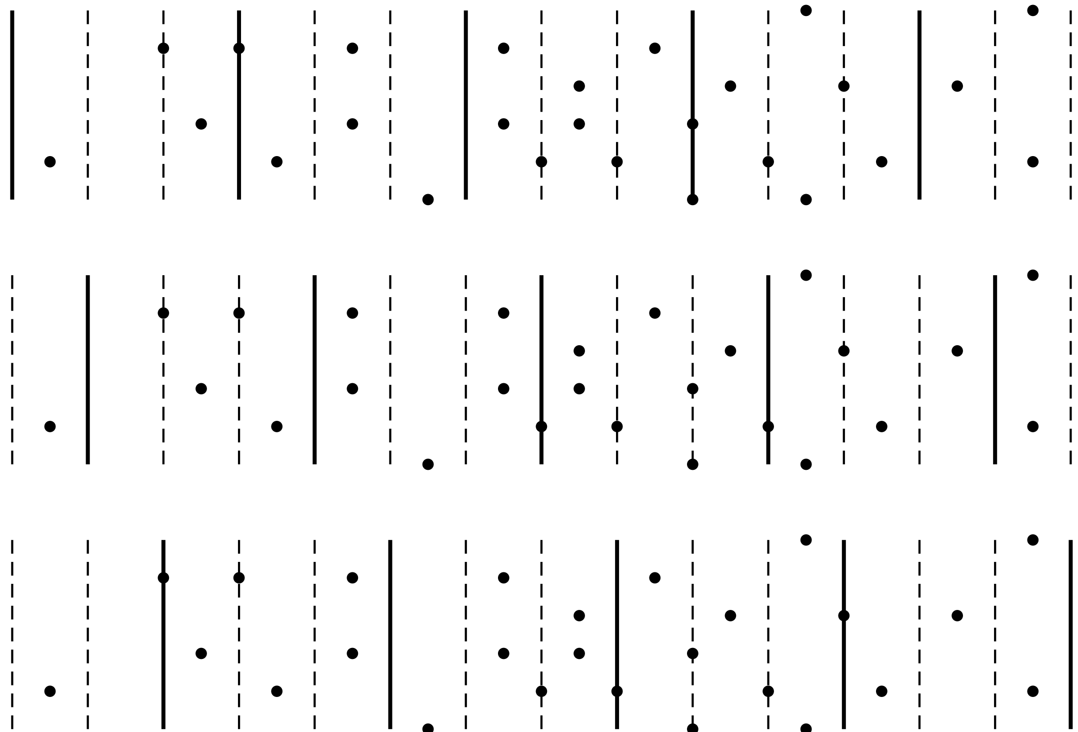


Figura 4.2: Particiones S_1, S_2 y S_3 generadas por la estrategia $shifting$ para $l = 3$ y $D = 2$.

Sea R_B el factor de aproximación de un algoritmo B definido como el supremo de $\frac{m(B)}{OPT}$ sobre todas las instancias del problema, donde $m(B)$ denota el valor de la solución retornada por el algoritmo B.

Lema 4.4. (Lema *Shifting*)

$$R_{\mathcal{S}(A)} \leq R_A \left(1 + \frac{1}{l}\right) \tag{4.7}$$

donde A es un algoritmo local y l es el parámetro $shifting$. [Hoc97]

Demostración. Puesto que

$$R_A = \sup \frac{m(A_j)}{OPT_j}$$

se tiene que $m(A_j) \leq R_A \mathrm{OPT}_j$ donde j corre sobre todas las bandas en la partición S_i, OPT_j es el número de discos en una cobertura óptima de los puntos en la banda j y $m(A_j)$ es el valor de la solución retornada por el algoritmo A sobre la banda j.

Luego,

$$m(A_1) + \ldots + m(A_{|S_i|}) \leq R_A \sum_{j \in S_i} OPT_j \tag{4.8}$$

Además, es fácil ver que

$$m(A(S_i)) \leq m(A_1) + \ldots + m(A_{|S_i|}) \tag{4.9}$$

De (4.8) y (4.9) se sigue que

$$m(A(S_i)) \leq r_A \sum_{j \in S_i} OPT_j \tag{4.10}$$

Sea OPT el conjunto de discos en una solución óptima y $\mathrm{OPT}^{(i)}$ el número de discos en OPT que cubren puntos en dos bandas lD adyacentes en la i-ésima partición, i=1,...,l. Puesto que OPT puede escribirse de la forma:

$$OPT = |B_1| + |B_2| + \ldots + |B_{|S_i|}| + |I_1| + |I_2| + \ldots + |I_{|S_i|-1}| \tag{4.11}$$

donde B_j denota el conjunto de discos en OPT que cubren puntos de la banda j-ésima solamente, esto es, que no cubren puntos en ningún par de bandas adyacentes

y I_j el conjunto de discos que cubren puntos en las bandas j y $j+1$ simultáneamente. Además,

$$\begin{aligned}
OPT_1 &\leq |B_1| + |I_1| \\
OPT_2 &\leq |I_1| + |B_2| + |I_2| \\
OPT_3 &\leq |I_2| + |B_3| + |I_3| \\
&\vdots \\
OPT_{|S_i|} &\leq |I_{|S_i|-1}| + |B_{|S_i|}|
\end{aligned}$$

Sumando estas desigualdades y por (4.11) se tiene que:

$$\begin{aligned}
\sum_{j\in S_i} OPT_j &\leq OPT + (|I_1| + |I_2| + \ldots + |I_{|S_i|-1}|) = \\
&= OPT + OPT^{(i)}
\end{aligned}$$

Esto es,

$$\sum_{j\in S_i} OPT_j \leq OPT + OPT^{(i)} \tag{4.12}$$

No pueden existir discos en OPT que cubran puntos en dos bandas adyacentes en una partición y que también cubran puntos en dos bandas adyacentes de otra partición. Por consiguiente, los conjuntos $OPT^1, \ldots, OPT^l$ son disyuntos y pueden sumar a lo más OPT. Es decir,

$$\sum_{i=1}^{l} OPT^{(i)} \leq OPT$$

De esto se sigue que:

$$\sum_{i=1}^{l} (OPT + OPT^{(i)}) \leq (1+l)OPT \tag{4.13}$$

Además, usando el hecho que el elemento mínimo de cualquier conjunto de números es menor o igual que el valor medio de ellos, se obtiene que:

$$\min_{i=1,\ldots,l} \sum_{j\in S_i} OPT_j \leq \frac{1}{l} \sum_{i=1}^{l} \left(\sum_{j\in S_i} OPT_j \right) \tag{4.14}$$

Y por las expresiones (4.12) y (4.13)

$$\frac{1}{l}\sum_{i=1}^{l}\left(\sum_{j\in S_i} OPT_j\right) \leq \left(1+\frac{1}{l}\right)OPT \tag{4.15}$$

Combinando las desigualdades (4.10), (4.14) y (4.15) se obtiene que:

$$m(\mathcal{S}(A)) = \min_{i=1,\ldots,l} m(A(S_i)) \leq R_A\left(1+\frac{1}{l}\right)OPT$$

de lo cual se establece (4.7).

El siguiente teorema muestra el PTAS para el problema de Cobertura con discos, que fué desarrollado por Hochbaum y Maass en 1985.

Teorema 4.6. Existe un esquema de aproximación tiempo polinomial $\mathcal{H}_l$ tal que para todo número natural dado $l \geq 1$, el algoritmo $\mathcal{H}_l$ retorna una cobertura de los n puntos dados en un espacio Euclidiano bidimensional con discos bidimensionales de diámetro dado D en $O(l^2\,\lceil l\sqrt{2}\rceil^2\, n^{2\lceil l\sqrt{2}\rceil^2+1})$ pasos con factor de aproximación menor o igual que $\left(1+\frac{1}{l}\right)^2$.

La prueba de este teorema se encuentra en [Hoc97], en ella se usan dos aplicaciones anidadas de la estrategia shifting.

4.4.3. Una mejor aproximación para el problema de Cobertura con discos

La Tabla 4.1 muestra varios algoritmos de aproximación que se han desarrollado para el problema de Cobertura con discos. En la sección 4.4.2 se presentó el esquema de aproximación tiempo polinomial desarrollado por Hochbaum y Maass en 1985. Haciendo uso de la estrategia $shifting$, ellos obtuvieron una familia de algoritmos cuyos

factores de aproximación son a lo más ϵ, con $1 < \epsilon \leq 4$ y tiempo de ejecución que es polinomial cuando ϵ es fijo. Si $\epsilon = 4$ el grado del polinomio es 9 y crece cuando ϵ se aproxima a uno. Posteriormente, Feder y Greene, e independientemente Gonzalez obtuvieron resultados similares para este problema. Ellos encontraron una familia de algoritmos con factores de aproximación de a lo más ϵ con $1 < \epsilon \leq 2$ y tiempo de ejecución que es polinomial cuando ϵ es fijo. El grado del polinomio es 13 si $\epsilon = 2$ y crece cuando ϵ se aproxima a uno. Ellos tambien obtuvieron un algoritmo más rápido, con un factor de aproximación de 8 y un tiempo de ejecución de $O(n + n \log S)$, donde $S \leq n$ es el número de discos en la solución óptima. Por otro lado, Brönnimann y Goodrich desarrollaron un algoritmo $O(n^3 \log n)$ que produce un factor de aproximación constante, todavía no determinado. Finalmente, Franceschetti, Cook y Bruck desarrollaron otro esquema de aproximación tiempo polinomial para este problema cuyo tiempo de ejecución es lineal en el número de puntos a cubrir y el factor de aproximación es a lo más ϵ con $3 < \epsilon \leq 24$. Ellos obtuvieron este resultado combinando la estrategia *shifting* de Hochbaum y Maass con una estrategia *grid* que consiste en encontrar una cobertura de los puntos colocando discos solamente en los vértices de una malla.

	Autor	Aproximación	Tiempo de ejecución	Año
1	Hochbaum y Maass	$\left(1 + \frac{1}{l}\right)^2$	$O\left(l^2 \lceil l\sqrt{2} \rceil^2 n^{2\lceil l\sqrt{2} \rceil^2 + 1}\right)$	1985
2	Gonzalez - Feder y Greene	$\left(1 + \frac{1}{l}\right)$	$O\left(6l \lceil l\sqrt{2} \rceil n^{6\lceil l\sqrt{2} \rceil + 1}\right)$	1991
3	Gonzalez - Feder y Greene	8	$O(n + n \log S)$	1991
4	Brönnimann y Goodrich	$O(1)$	$O(n^3 \log n)$	1995
5	Franceschetti, Cook y Bruck	$\alpha\left(1 + \frac{1}{l}\right)^2$	$O(Kn)$	2000

Tabla 4.1: Algoritmos de aproximación para Cobertura con discos. l es un parámetro entero, n es el número de puntos a cubrir, $S \leq n$ es el número de discos en la solución óptima, $\alpha \in \{3, 4, 5, 6\}$, K es un valor constante que depende de α y l.

Por lo anterior y de la Tabla 4.1 se observa que los algoritmos número tres y número cinco presentan los mejores tiempos de ejecución aunque este último, a diferencia del anterior, permite controlar la aproximación ya que es un PTAS. Sin embargo, los algoritmos número uno y número dos presentan los mejores factores de aproximación.

A continuación se explicará en qué consiste la estrategia *grid* y cómo se aplica para obtener un PTAS para el problema de Cobertura con discos. Este es el resultado de Franceschetti, Cook y Bruck que se muestra en el algoritmo número cinco en la tabla.

La estrategia *Grid*

Considere un cuadrado que contiene m puntos que se desean cubrir con discos de radio r fijo y sobre él una rejilla. La estrategia *grid* consiste en encontrar una cobertura de los puntos colocando los discos centrados solamente en los vértices de la rejilla. Un disco de radio fijo r, centrado en un vértice de la malla, se llamará un *disco grid* (Figura 4.3). Usando esta estrategia $R_A \in \{3, 4, 5, 6\}$, como se observa en el corolario 4.5 del siguiente teorema.

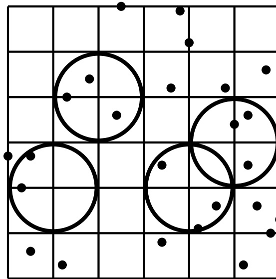


Figura 4.3: Una rejilla y algunos discos *grid*.

Teorema 4.7. [FCB00] Considérese una rejilla cuadrada, donde la distancia entre dos de sus vértices consecutivos es uno. El número N de discos *grid* que son necesarios y suficientes para cubrir cualquier disco de radio r colocado sobre el plano, es dado por:

1. N no existe, si $r < \frac{\sqrt{2}}{2}$.

2. $N = 6$, si $\frac{\sqrt{2}}{2} \leq r < \frac{\sqrt{10}}{4}$.

3. $N = 5$, si $\frac{\sqrt{10}}{4} \leq r < 1$.

4. $N = 4$, si $1 \leq r < \frac{5\sqrt{2}}{4}$.

5. $N = 3$, si $r \geq \frac{5\sqrt{2}}{4}$.

Corolario 4.5. Considérese una rejilla cuadrada donde la distancia entre dos de sus vértices consecutivos es uno. El factor de aproximación resultante de cubrir un conjunto de n puntos en el plano usando discos *grid* es:

$$R \leq \begin{cases} 3 & \text{si } r \geq \frac{5\sqrt{2}}{4} \\ 4 & \text{si } 1 \leq r < \frac{5\sqrt{2}}{4} \\ 5 & \text{si } \frac{\sqrt{10}}{4} \leq r < 1 \\ 6 & \text{si } \frac{\sqrt{2}}{2} \leq r < \frac{\sqrt{10}}{4} \end{cases}$$

y es indefinido para $r < \frac{\sqrt{2}}{2}$.

Demostración. Sea OPT la cobertura óptima de los n puntos dados. Una cobertura mínima obtenida usando discos *grid* es llamada una *cobertura grid*. Puesto que para cubrir un disco de OPT se necesitan un número constante $(3, 4, 5$ o $6)$ de discos *grid*, el factor de aproximación es el número de discos en la cobertura *grid* que son requeridos, en el peor caso, para cubrir todos los puntos cubiertos por un disco del conjunto OPT. Este número es dado por el Teorema 4.7.

El siguiente teorema muestra el algoritmo desarrollado por Franceschetti, Cook y Bruck.

Teorema 4.8. Existe un algoritmo de aproximación tiempo lineal $\mathcal{B}_l$ tal que para todo número natural dado $l \geq 1$, el algoritmo $\mathcal{B}_l$ retorna una cobertura de los n puntos dados con discos de diámetro dado D en $O(K_{(l,p)}\, n)$ pasos, con factor de aproximación menor o igual que $\alpha\left(1 + \frac{1}{l}\right)^2$. [FCB00]

Demostración. Inicialmente, se usan dos aplicaciones anidadas de la estrategia *shifting*. Primero, se corta el plano en bandas verticales de ancho lD; luego, con el objetivo de cubrir los puntos en una de estas bandas, se aplica la estrategia

shifting a la otra dimensión. Así, se corta la banda considerada en cuadrados de lado lD (Figura 4.4).

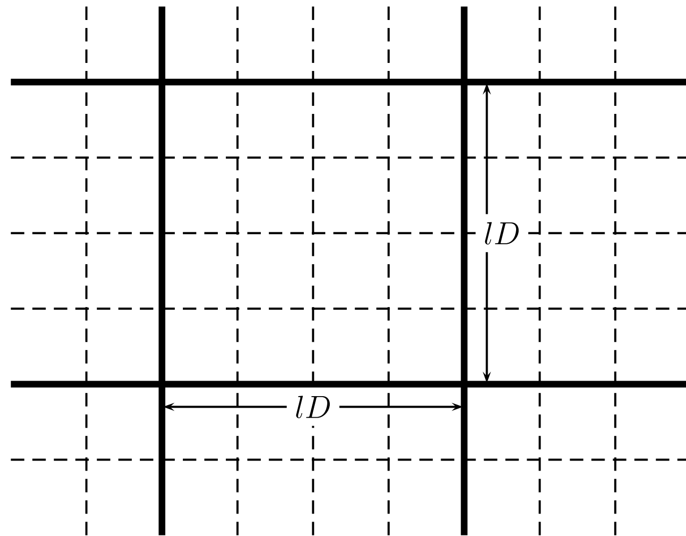


Figura 4.4: Una partición del plano en cuadrados de lado lD con $l = 4$ y $D = 1$.

En el algoritmo desarrollado por Hochbaum y Maass (Teorema 4.6) se muestra que con $\lceil l\sqrt{2}\rceil^2$ discos de diámetro D, se puede cubrir un cuadrado lD compactamente (Figura 4.5).

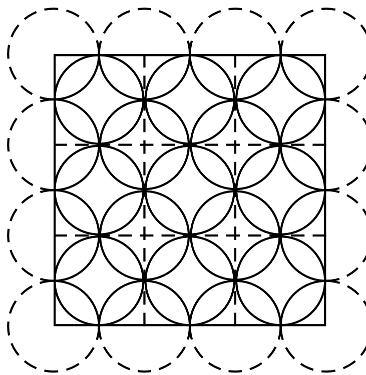


Figura 4.5: Cobertura compacta de un cuadrado lD con $3^2 \times 2$ discos para $l = 3$ y $D = 2$.

Ahora el algoritmo procede aplicando la estrategia *grid*. Considérese una rejilla finita colocada en el tope de un cuadrado de lado lD, donde la distancia entre dos de los

vértices consecutivos de la rejilla es uno. Sea p el número de vértices de esta rejilla y α el factor de aproximación resultante de cubrir los puntos en este cuadrado usando discos centrados en vértices de esta rejilla. Sea $K_{(l,\,p)} = l^2 \sum_{i=1}^{\lceil l\sqrt{2}\rceil^2-1} \binom{p}{i} i$.

Sea $\overline{n}$ el número de puntos dados para cubrir, contenidos en un cuadrado de lado lD. Ya que con $\lceil l\sqrt{2}\rceil^2$ discos se puede cubrir el cuadrado compactamente, restringiendo la búsqueda a una cobertura óptima con discos $grid$, se tienen que considerar arreglos de a lo más $\lceil l\sqrt{2}\rceil^2 - 1$ discos $grid$, sobre las p posibles posiciones de discos. Esto se debe a que si no se puede cubrir todos los $\overline{n}$ puntos usando $\lceil l\sqrt{2}\rceil^2 - 1$ discos $grid$, se puede abandonar la rejilla y usar la cobertura compacta. Con el objetivo de chequear si un arreglo de discos es una cubierta factible de los $\overline{n}$ puntos en el cuadrado, se necesita determinar para cada punto si está a una distancia a lo más de $r = D/2$ de uno de los centros de los discos. Esta operación requiere a los más $i\overline{n}$ pasos, donde i es el número de discos en el arreglo, asumiendo que la distancia entre dos puntos en el plano se puede determinar con la precisión necesaria en un paso. Por consiguiente, el número de pasos necesarios para encontrar una cubierta mínima en un cuadrado es a lo más:

$$\sum_{i=1}^{\lceil l\sqrt{2}\rceil^2-1} \binom{p}{i} i\,\overline{n}$$

Sumando sobre todos los cuadrados, se tiene un número de pasos de:

$$\begin{aligned}\sum_{i=1}^{\lceil l\sqrt{2}\rceil^2-1} \binom{p}{i} i\,\overline{n}_1 + \sum_{i=1}^{\lceil l\sqrt{2}\rceil^2-1} \binom{p}{i} i\,\overline{n}_2 + \ldots + \sum_{i=1}^{\lceil l\sqrt{2}\rceil^2-1} \binom{p}{i} i\,\overline{n}_t \\ = \sum_{i=1}^{\lceil l\sqrt{2}\rceil^2-1} \binom{p}{i} i\ [\overline{n}_1 + \overline{n}_2 + \ldots + \overline{n}_t] \\ = \sum_{i=1}^{\lceil l\sqrt{2}\rceil^2-1} \binom{p}{i} i\,n\end{aligned}$$

donde $\overline{n}_j$ es el número de puntos en el cuadrado j-ésimo. Las dos aplicaciones anidadas de la estrategia $shifting$ adicionan otro factor l^2. Así, la cota de tiempo global es:

$$l^2 \sum_{i=1}^{\lceil l\sqrt{2}\rceil^2-1} \binom{p}{i} i\,n = K_{(l,p)}\,n \quad \text{pasos.}$$

Para cada aplicación de la estrategia $shifting$, de acuerdo al lema 4.4 (Lema $Shifting$) se obtiene:

$$R_{\mathcal{B}_l} = R_{S(S(B))} \leq R_{S(B)}\left(1 + \frac{1}{l}\right) \leq R_B\left(1 + \frac{1}{l}\right)^2$$

donde B es un algoritmo local que calcula una solución en un cuadrado lD usando discos *grid* y por el corolario 4.5, el factor de aproximación del algoritmo local B es α, donde $\alpha \in \{3,\ 4,\ 5,\ 6\}$. Por lo tanto, $R_{\mathcal{B}_l} \leq \alpha\left(1 + \frac{1}{l}\right)^2$.

Así, para cualquier ϵ tal que $\alpha < \epsilon \leq 4\alpha$, el algoritmo $\mathcal{B}_l$ con $l = \frac{\alpha\left(1+\sqrt{\epsilon/\alpha}\right)}{\epsilon-\alpha}$ retorna una solución cuyo factor de aproximación es a lo más ϵ.

4.4.4. El problema MAX SUBSET-SUM está en FPTAS

Teorema 4.9. El problema MAX SUBSET-SUM $\in$ FPTAS. [CLR90]

Demostración. Por el ejemplo 4.3, MAX SUBSET-SUM $\in$ NPO. Se debe probar que MAX SUBSET-SUM admite un esquema de aproximación completamente polinomial. Primero se mostrará un algoritmo de tiempo exponencial, EXACT-SUBSET-SUM, para este problema.

Sea L una lista de enteros positivos y $x \in \mathbb{Z}^+$. Sea $L + x$ una lista de enteros obtenida de L incrementando en x cada elemento de L. Por ejemplo, si $L = \langle\, 5,\ 7,\ 9\,\rangle$ entonces $L + 3 = \langle\, 8,\ 10\,12\,\rangle$. También, se usará esta misma notación para conjuntos. Así, si A es un conjunto, $A + x = \{a + x :\ a \in A\}$.

El algoritmo EXACT-SUBSET-SUM utiliza un procedimiento auxiliar, MEZCLAR-LISTAS(L, L') el cual retorna una lista que es el resultado de unir las listas de entrada L y L' ordenadas. MEZCLAR-LISTAS corre en tiempo $\mathrm{O}(|L| + |L'|)$.

EXACT-SUBSET-SUM toma como entrada un conjunto $S = \{x_1,\ x_2,\ \ldots,\ x_n\}$ con $x_i \in \mathbb{Z}^+$, $i = 1, \ldots, n$ y un entero positivo t.

EXACT-SUBSET-SUM $(S,\, t)$

1. $n \leftarrow |S|$
2. $L_0 \leftarrow \langle 0 \rangle$
3. **para** $i \leftarrow 1$ **hasta** n **hacer**
   $L_i \leftarrow$ MEZCLAR-LISTAS$(L_{i-1},\, L_{i-1} + x_i)$
   quitar de L_i todos los enteros mayores que t
4. Retornar el entero más grande en L_n

La Figura 4.6 (a) ilustra el funcionamiento de EXACT-SUBSET-SUM para la instancia $S = \{1,\, 3,\, 5,\, 8\}$ y $t = 7$.

$n \leftarrow 4$

$L_0 \leftarrow \langle 0 \rangle$

Iteración(i)	L_i
1	$L_1 =$ MEZCLAR-LISTAS$(L_0,\, L_0 + 1) = \langle\, 0,\, 1\, \rangle$
2	$L_2 =$ MEZCLAR-LISTAS$(L_1,\, L_1 + 3) = \langle\, 0,\, 1,\, 3,\, 4\, \rangle$
3	$L_3 =$ MEZCLAR-LISTAS$(L_2,\, L_2 + 5) = \langle\, 0,\, 1,\, 3,\, 4,\, 5,\, 6,\, 8,\, 9 \rangle$
	$L_3 = \langle\, 0,\, 1,\, 3,\, 4,\, 5,\, 6\, \rangle$
4	$L_4 =$ MEZCLAR-LISTAS$(L_3,\, L_3 + 8) = \langle\, 0,\, 1,\, 3,\, 4,\, 5,\, 6,\, 8,\, 9,\, 11,\, 12,\, 13,\, 14\, \rangle$
	$L_4 = \langle\, 0,\, 1,\, 3,\, 4,\, 5,\, 6\, \rangle$

a)

$P_1 = \{0,\, 1\}$

$P_2 = \{0,\, 1,\, 3,\, 4\}$

$P_3 = \{0,\, 1,\, 3,\, 4,\, 5,\, 6,\, 8,\, 9\}$

$P_4 = \{0,\, 1,\, 3,\, 4,\, 5,\, 6,\, 8,\, 9,\, 11\, 12,\, 13,\, 14,\, 16,\, 17\}$

b)

Figura 4.6: a) Funcionamiento del procedimiento EXACT-SUBSET-SUM aplicado a la instancia $S = \{1,\, 3,\, 5,\, 8\}$ y $t = 7$. Retorna 6 que corresponde a la suma de los elementos del subconjunto $\{1, 5\}$. b) Conjuntos P_i para S, t.

Sea P_i el conjunto de todos los enteros que son obtenidos seleccionando un subconjunto (posiblemente vacío) de $\{x_1,\, \ldots, x_i\}$ y sumando sus miembros (Figura 4.6 (b)).

Observe que $|P_i| \leq 2^i$ y además que $P_i = P_{i-1} \cup (P_{i-1} + x_i)$, $i = 1, \ldots, n$.
Utilizando inducción matemática se probará que cada lista L_i es una lista ordenada que contiene todos los elementos de P_i que son menores o iguales que t.

- Si $i = 1, P_1 = \{0, x_1\}$. Además, por las líneas 4 y 5 del algoritmo se tiene que: MEZCLAR-LISTAS(L_0, $L_0 + x_1$) $= \langle 0, x_1 \rangle$ y

$$L_1 = \langle 0, x_1 \rangle \quad \text{si} \quad x_1 \leq t$$

o

$$L_1 = \langle 0 \rangle \quad \text{si} \quad x_1 > t.$$

  Luego, L_1 es una lista ordenada que contiene todos los elementos de P_1 que son menores o iguales que t.

- Suponga que la afirmación se cumple para $i = k$ $(k = 1, \ldots, n-1)$, esto es, si $P_k = \{y_1, \ldots, y_r\}$ con $r \leq 2^k$ entonces $L_k = \langle \overline{y}_1, \overline{y}_2, \ldots, \overline{y}_m \rangle$ tal que $\{\overline{y}_1, \overline{y}_2, \ldots, \overline{y}_m\} \subseteq P_k$ con $\overline{y}_1 \leq \overline{y}_2 \leq \ldots \leq \overline{y}_m \leq t$.

- Finalmente, se mostrará que la afirmación se cumple para $i = k+1$. Puesto que $P_i = P_{i-1} \cup (P_{i-1} + x_i)$ se tiene que $P_{k+1} = \{y_1, \ldots, y_r, y_1 + x_{k+1}, \ldots, y_r + x_{k+1}\}$. Ahora, de las líneas 4 y 5 del algoritmo se obtiene que MEZCLAR-LISTAS(L_k, $L_k + x_{k+1}$) es una lista ordenada compuesta por $\overline{y}_1, \ldots, \overline{y}_m, \overline{y}_1 + x_{k+1}, \ldots, \overline{y}_m + x_{k+1}$ y puesto que $\{\overline{y}_1, \overline{y}_2, \ldots, \overline{y}_m\} \subseteq P_k \subseteq P_{k+1}$ y $\{\overline{y}_1 + x_{k+1}, \ldots, \overline{y}_m + x_{k+1}\} \subseteq P_k + x_{k+1} \subseteq P_{k+1}$.
  Además, por la línea 5, L_{k+1} se obtiene quitando de $\overline{y}_1, \ldots, \overline{y}_m, \overline{y}_1 + x_{k+1}, \ldots, \overline{y}_m + x_{k+1}$ todos los valores que sean mayores que t. Por lo tanto, L_{k+1} es una lista que contiene todos los elementos de P_{k+1} menores o iguales que t.

Así, por lo anterior, $|L_i| \leq |P_i| \leq 2^i$, $i = 1, \ldots, n$. Luego, EXACT-SUBSET-SUM es un algoritmo de tiempo exponencial.

Se puede derivar un esquema de aproximación completamente polinomial para SUBSET-SUM "recortando" cada lista L_i después de creada. Sea δ un "parámetro de recorte"

tal que $0 < \delta < 1$. "Recortar" una lista L por δ significa quitar de L tantos elementos como sea posible, de tal forma que si L' es el resultado de "recortar" L, entonces para cada elemento y que fué quitado de L, existe un elemento $z \leq y$ en L' tal que

$$\frac{y - z}{y} \leq \delta$$

o equivalentemente

$$(1 - \delta)\, y \leq z \leq y.$$

Se puede pensar del elemento z como un "representante" de y en la lista L'. Cada y es representado por un z tal que el error relativo de z con respecto a y es a lo más δ. Por ejemplo, si $\delta = 0.2$ y $L = \langle 3, 5, 10, 11, 13, 14, 16, 19, 20 \rangle$, entonces el "recorte" de L es $L' = \langle 3, 5, 10, 13, 19 \rangle$ donde el valor borrado 11 es representado por 10, los valores borrados 14 y 16 son representados por 13 y el valor borrado 20 es representado por 19. Así, cada elemento borrado de la lista tiene un representante de la misma lista. A continuación se presenta un procedimiento que "recorta" una lista dada $L = \langle y_1, \ldots, y_m \rangle$ en tiempo $O(m)$ asumiendo que L está ordenada en forma no decreciente.

TRIM(L, δ)

1. $m \leftarrow |L|$
2. $L' \leftarrow \langle y_1 \rangle$
3. ultimo $\leftarrow y_1$
4. **para** $i \leftarrow 2$ **hasta** m **hacer**
5.   **si** último $< (1 - \delta)\, y_i$ **entonces**
6.   colocar y_i al final de L'
7.   último $\leftarrow y_i$.
8. **Retornar** L'

Los elementos de L son examinados en forma creciente y un número es colocado en la lista L' si, y sólo si es el primer elemento de L o si no puede ser representado por el último número acabado de colocar en L'.

Finalmente, se mostrará un esquema de aproximación totalmente polinomial para MAX SUBSET-SUM. Este procedimiento toma como entrada un conjunto $S = \{x_1, x_2, \ldots, x_n\}$ de enteros y un "parámetro de aproximación" ϵ donde $\epsilon > 1$.

APROX-SUBSET-SUM(S, t, ϵ)

1. $n \leftarrow |S|$
2. $L_0 \leftarrow \langle 0 \rangle$
3. **para** $i \leftarrow 1$ **hasta** n **hacer**
4. $\quad L_i \leftarrow$ MEZCLAR-LISTAS$(L_{i-1},\, L_{i-1} + x_i)$
5. $\quad L_i \leftarrow \text{TRIM}\left(L_i, \frac{1}{n}\left(1 - \frac{1}{\epsilon}\right)\right)$
6. $\quad$ quitar de L_i todos los elementos que sean mayores que t
7. Retornar el elemento mayor de L_n

La Figura 4.7 muestra una computación de APROX-SUBSET-SUM aplicado a la instancia $S = \{52,\, 48,\, 75,\, 80,\, 77\}$, $t = 208$ y $\epsilon = 1{,}25$.

La solución óptima de esta instancia es $207 = 80 + 75 + 52$, el algoritmo retorna $z = 200$ que corresponde a la suma $77 + 75 + 48$. Nótese que el factor de aproximación $R(x,\, T(x)) = 207/200 < 1.25$.

A continuación se probará que APROX-SUBSET-SUM es en realidad un esquema de aproximación totalmente polinomial para MAX SUBSET-SUM.

Como se observó anteriormente, todos los elementos de L_i son también elementos de P_i, además, después de recortar L_i y quitar de ella todo elemento más grande que t (líneas 5 y 6) esta propiedad se sigue manteniendo. Por consiguiente, el valor z retornado en la línea 8 es realmente la suma de algún elemento de S. Se debe probar que z no es mas pequeño que $1/\epsilon$ veces una solución óptima, es decir,

$$\frac{1}{\epsilon} z^* \leq z$$

$n \leftarrow 5$

$L_0 \leftarrow \langle 0 \rangle$

Iteración(i)	Línea	L_i
1	4	$L_1 = \langle 0,\, 52 \rangle$
	5	$L_1 = \langle 0,\, 52 \rangle$
2	4	$L_2 = \langle 0,\, 48,\, 52,\, 100 \rangle$
	5	$L_2 = \langle 0,\, 48,\, 52,\, 100 \rangle$
3	4	$L_3 = \langle 0,\, 48,\, 52,\, 75,\, 100,\, 123,\, 127,\, 175 \rangle$
	5	$L_3 = \langle 0,\, 48,\, 52,\, 75,\, 100,\, 123,\, 175 \rangle$
4	4	$L_4 = \langle 0,\, 48,\, 52,\, 75,\, 80,\, 100,\, 123,\, 128,\, 132,\, 155,\, 175,\, 180,\, 203,\, 255 \rangle$
	5	$L_4 = \langle 0,\, 48,\, 52,\, 75,\, 80,\, 100,\, 123,\, 132,\, 155,\, 175,\, 203,\, 255 \rangle$
	6	$L_4 = \langle 0,\, 48,\, 52,\, 75,\, 80,\, 100,\, 123,\, 132,\, 155,\, 175,\, 203 \rangle$
5	4	$L_5 = \langle 0,\, 48,\, 52,\, 75,\, 77,\, 80,\, 100,\, 123,\, 125,\, 129,\, 132,\, 152,\, 155,\, 157,\, 175,\, 177,\, 200,\, 203,\, 209,\, 232,\, 252,\, 280 \rangle$
	5	$L_5 = \langle 0,\, 48,\, 52,\, 75,\, 80,\, 100,\, 123,\, 129,\, 152,\, 175,\, 200,\, 209,\, 232,\, 252,\, 280 \rangle$
	6	$L_5 = \langle 0,\, 48,\, 52,\, 75,\, 80,\, 100,\, 123,\, 129,\, 152,\, 175,\, 200 \rangle$

Figura 4.7: Una computación del algoritmo de aproximación para MAX SUBSET-SUM aplicado a la instancia $S = \{52,\, 48,\, 75,\, 80,\, 77\}$, $t = 208$ y $\epsilon = 1{,}25$

donde z^* es una solución óptima, y tambien que el algoritmo corre en tiempo polinomial.

Por inducción sobre i, se mostrará que para todo elemento y en P_i que sea menor que t, existe un $z \in L_i$ tal que:

$$\left(1 - \frac{\epsilon - 1}{n\,\epsilon}\right)^i y \leq z \leq y. \tag{4.16}$$

- Primero, obsérvese que para $i = 1$ la afirmación 4.16 se cumple:
  Si $S = \{x_1,\, x_2, \ldots, x_n\}$, entonces $P_1 = \{0,\, x_1\}$ y

$$L_1 = \langle 0,\, x_1 \rangle \quad \text{si} \quad x_1 \leq t$$

$$\text{o}$$

$$L_1 = \langle 0 \rangle \quad \text{si} \quad x_1 > t.$$

Luego, L_1 contiene todos los enteros de P_1 no mayores que t.

- Ahora, supóngase que para $k = i - 1$, la afirmación 4.16 se cumple (hipótesis inductiva), es decir, para todo y en P_{i-1} que es menor que t, existe un $z' \in L_{i-1}$ tal que:

$$\left(1 - \frac{\epsilon - 1}{n\,\epsilon}\right)^{i-1} y \leq z' \leq y. \tag{4.17}$$

- Sea $y_i \in P_i$ con $y \leq t$. Puesto que $P_i = P_{i-1} \cup (P_{i-1} + x_i)$ pueden suceder dos casos: $y_i \in P_{i-1}$ o $y_i \in P_{i-1} + x_i$.

  1. Si $y_i \in P_{i-1}$, aplicando la hipótesis inductiva sobre y_i se tiene que existe $z' \in L_{i-1}$:

  $$(1 - \delta)^{i-1} y_i \leq z' \leq y_i \tag{4.18}$$

  Del algoritmo APROX-SUBSET-SUM se puede observar que todo elemento en L_{i-1} o en $L_{i-1} + x_i$ tiene representante, el mismo o un elemento distinto, en la iteración i-ésima. Por consiguiente existe $z'' \in L_i$ tal que

  $$(1 - \delta) z' \leq z'' \leq z' \tag{4.19}$$

  Multiplicando (4.18) por $(1 - \delta)$, se tiene que

  $$(1 - \delta)^i y_i \leq (1 - \delta) z' \leq z'' \leq z' \leq y_i \tag{4.20}$$

  (por(4.18) y (4.19)).
  Luego, $(1 - \delta)^i y_i \leq z'' \leq y_i$.

  2. Si $y_i \in P_{i-1} + x_i$, implica que existe $w \in P_{i-1}$ tal que $y_i = w + x_i$. Aplicando la hipótesis inductiva sobre w, se tiene que existe $z' \in L_{i-1}$ tal que

  $$(1 - \delta)^{i-1} w \leq z' \leq w. \tag{4.21}$$

  Además, puesto que $z' \in L_{i-1}$, $z' + x_i \in L_{i-1} + x_i$. Luego, existe $z'' \in L_i$ tal que

  $$(1 - \delta)(z' + x_i) \leq z'' \leq (z' + x_i) \tag{4.22}$$

De (4.21) y (4.22) se tiene que

$$\begin{aligned}
(1-\delta)^{i-1}(y_i - x_i) &\leq z' \\
(1-\delta)^{i}(y_i - x_i) &\leq z'(1-\delta) \\
(1-\delta)^{i} y_i - (1-\delta)^{i} x_i &\leq z'(1-\delta) \\
(1-\delta)^{i} y_i &\leq (1-\delta)^{i} x_i + z'(1-\delta) \\
&\leq z'(1-\delta) + (1-\delta) x_i \\
&\leq (1-\delta)(z' + x_i) \\
&\leq z'' \leq z' + x_i \leq w + x_i = y_i
\end{aligned}$$

Por lo tanto, $(1-\delta)^i y_i \leq z'' \leq y_i$.

Si $z^* \in P_n$ es una solución óptima de MAX SUBSET-SUM entonces existe un $z \in L_n$ tal que

$$\left(1 - \frac{\epsilon - 1}{n\,\epsilon}\right)^n z^* \leq z \leq z^*$$

donde z es el valor retornado por APROX-SUBSET-SUM.

Puesto que

$$\frac{d}{dn}\left(1 - \frac{\epsilon - 1}{n\,\epsilon}\right)^n > 0$$

entonces la función $\left(1 - \frac{\epsilon-1}{n\,\epsilon}\right)^n$ es creciente, de modo que $n > 1$ implica que:

$$\frac{1}{\epsilon} < \left(1 - \frac{\epsilon - 1}{n\,\epsilon}\right)^n.$$

Luego, $\frac{1}{\epsilon}\, z^* \leq z$.

Por consiguiente, el valor z retornado por APROX-SUBSET-SUM es por lo menos $1/\epsilon$ veces la solución óptima.

Ahora sólo falta probar que APROX-SUBSET-SUM es tiempo polinomial en la longitud de la entrada y en $\frac{1}{\epsilon - 1}$. Para esto, se acotará la longitud de cada lista L_i. Sea

$L_i = \langle 0, y_1, \ldots, y_k \rangle$ después de aplicar el algoritmo TRIM y quitar los números más grandes que t. Por definición del algoritmo TRIM para todo y_j y y_{j-1} en L_i se tiene que $y_{j-1} < \left(1 - \frac{\epsilon-1}{n\,\epsilon}\right) y_j$ puesto que cuando $last = y_{j-1}$ esa es la condición que se debe cumplir para incluir a y_j en L_i. Como $y_1 \geq 1$ y $y_k \leq t$ se obtiene que

$$1 \leq y_1 < \left(1 - \frac{\epsilon - 1}{n\,\epsilon}\right) y_2 < \left(1 - \frac{\epsilon - 1}{n\,\epsilon}\right)^2 y_3 < \ldots < \left(1 - \frac{\epsilon - 1}{n\,\epsilon}\right)^{k-1} y_k \leq \left(1 - \frac{\epsilon - 1}{n\,\epsilon}\right)^{k-1} t.$$

Haciendo uso de la desigualdad $(1 - \alpha) < e^{-\alpha}$ con $\alpha = \frac{\epsilon-1}{n\,\epsilon}$ se tiene que

$$1 \leq t\left(1 - \frac{\epsilon - 1}{n}\right)^{k-1} < te^{-(k-1)\,\epsilon-1/n\epsilon}$$

Luego,

$$\begin{aligned}
te^{-(k-1)\,\epsilon-1/n\epsilon} &\geq 1 \\
\ln t - (k-1)\frac{\epsilon - 1}{n\,\epsilon} &\geq 0 \\
\ln t &\geq (k-1)\frac{\epsilon - 1}{n\,\epsilon} \\
\frac{n\,\epsilon}{\epsilon - 1} \ln t &\geq k - 1 \\
\frac{n\,\epsilon}{\epsilon - 1} \ln t + 2 &\geq k + 1
\end{aligned}$$

Por lo tanto, $|L_i|$ es acotada en n y $\frac{1}{\epsilon-1}$. Puesto que se realizan n iteraciones y en cada iteración, el número de operaciones es lineal en n y $\frac{1}{\epsilon-1}$ y polinomial en $\sum \log x_i$ (correspondiente a los *bits* utilizados por los números x_i), se obtiene que el tiempo de ejecución del algoritmo es polinomial en la longitud de la entrada y en $\frac{1}{\epsilon-1}$, como se quería probar. En consecuencia, APROX-SUBSET-SUM es un FPTAS para el problema MAX SUBSET-SUM.

5. CONCLUSIONES Y SUGERENCIAS PARA TRABAJOS FUTUROS

- Cuando una persona se enfrenta por primera vez a la Teoria de NP-completitud y de aproximabilidad en problemas de optimización frecuentemente se encuentra con la dificultad de que gran parte de la literatura disponible alrededor de estas teorías tienen un nivel avanzado. En esta monografia se hizo una presentación amplia, detallada y clara usando ilustraciones gráficas, ejemplos y modificando la forma de presentación de las demostraciones con los detalles y justificaciones pertinentes de tal manera que sea accesible a toda persona interesada en el tema con un nivel básico de preparación matemática.
- Se realizó un estudio minucioso e ilustrativo de los modelos de computación específicamente de las máquinas de Turing y las máquinas RAM con el propósito de alcanzar una buena comprensión de sus funcionamientos y de proporcionar una visión general de su aplicabilidad y su importancia como elementos que sustentan la Teoría de la Complejidad Computacional.
- En el estudio realizado de la Teoría de NP-completitud se identificó el papel central que tienen los esquemas de codificación dentro de esta teoría puesto que el uso de un esquema de codificación no apropiado puede llevar a disminuir la dureza de un problema hasta tal punto que pueda resolverse en tiempo polinomial, cuando en realidad utilizando una codificación apropiada, el problema sea incluso NP-completo.

- Se hizo una presentación amplia y didáctica de dos técnicas recientes, denominadas la estrategia $shifting$ y la estrategia $grid$, que están siendo muy aplicadas en el desarrollo de esquemas de aproximación tiempo polinomial por lo que han alcanzado gran importancia dentro de la teoría; esta aplicabilidad se ilustró con el desarrollo de un PTAS para el problema de Cobertura con discos.

- Dentro de la Teoría de la Aproximabilidad se identificó la importancia de la relación entre las clases P y NP. Esta teoría se fundamenta en la suposición de que P $\neq$ NP puesto que si se prueba lo contrario entonces todo problema en NPO cuyo problema de decisión subyacente es NP-completo está en PO.

- La teoría desarrollada alrededor de NP-completitud y aproximabilidad en problemas NP-duros es muy extensa y lo que se ha presentado en este trabajo es sólo una pequeña parte que puede servir de motivación y base para futuros trabajos. Algunos aspectos que se podrían tener en cuenta se presentan a continuación.

  - Es de interés buscar entre los problemas relaciones cada vez mas específicas de acuerdo a sus características y su dureza de tal forma que estas relaciones los clasifican y generan clases cada vez mas excluyentes, esto es conveniente puesto que los resultados que se obtienen para los problemas mas "duros" benefician a toda la clase. Así, análogamente a la noción de transformación polinomial utilizada en la teoría de la complejidad, también se definen transformaciones o reducciones entre problemas de optimización en NPO, algunas de ellas son las L-reducciones, PTAS-reducciones y A-reducciones que introducen el concepto de completitud en las clases APX y PTAS, similar a la noción de completitud en la clase NP. Algunas referencias: [Mac99], [Hoc97], [Pap94].

  - Realizar un estudio de la no-aproximabilidad de problemas NP duros ya que éste es un campo muy amplio en el que recientemente se han obtenido avances teóricos importantes, muchos de los cuales se han generado a raíz de la aparición del Teorema PCP (*Probabilistically Checkable Proofs*). Algunas referencias: [Hoc97], [Hou97], [Pap94].

BIBLIOGRAFIA

[AKS02] AGRAWAL, Manindra, KAYAL Neeraj y SAXENA, Nitin. PRIMOS is in P. Indian Institute of Technology Kanpur. Department of Computer Science & Engineering. Kanpur, 2002.
<http://www.cse.iitk.ac.in/>

[AHU98] AHO, Alfred V.; HOPCROFT, Jhon E. y ULLMAN,Jeffrey D. Estructuras de datos y Algoritmos. México: Adisson Wesley Longman de México, S.A de C.V, 1998.

[BB97] BRASSARD, G. y BRATLEY, P. Fundamentos de Algoritmia. Madrid: Prentice Hall, 1997.

[BC94] BOVET, Daniel Pierre y CRESCENZI, Pierluigi. Introduction to the theory of complexity. Gran Bretaña: Prentice Hall International, 1994.

[Coo00] COOK, Stephen. The P versus NP Problem. University of Toronto, 2000.
<http://www.claymath.org/Millenium_ Prize_ Problems/P_ Vs_ NP/_ objects /Official_ Problem_ Description.pdf>

[CLR90] CORMEN, Thomas; LEISERSON, Charles y RIVEST, Ronald. Introduction to algorithms. New York: MIT PRESS, 1990.

[CK95] CRESCENZI,Pierluigi y KANN, Viggo. A Compendium of NP Optimization problems. Dipartimento di Scienze dell'Informazione Universitá di Roma, La Sapienza, 1998.
<http://www.nada.kth.se/theory/compendium.>

[FCB00] FRANCESCHETTI, Massimo; COOK, Matthew y BRUCK, Jehoshua. A Geometric Theorem for Approximate Disk Covering Algorithms. California Institute of Technology, 2000.
<http://www.paradise.caltech.edu/papers/etr035.ps.>

[GJ79] GAREY, Michael R. y JOHNSON, David S. Computers and Intractibility: A Guide to the Theory of NP-Completeness. New York: W.H. Freeman, 1979.

[GS01] GOMEZ M., Raul y SICARD, Andres. Informática Teórica: Elementos propedéuticos. Medellin: Fondo Editorial Universidad EAFIT, 2001.

[GY99] GROSS, Jonathan y YELLEN, Jay. Graph Theory and its applications. Florida: The CRC Press series on discrete mathematics and its applications,1999.

[Hoc97] HOCHBAUM, Dorit S. Approximation Algoritms for NP-hard problems. PWS Publishing Company, 1997.

[Hou97] HOUGARDY, Stefan. Proof Checking and Non-Approximability. 1997
<www.informatik.hu-berlin.de/∼ hougardy/paper/chapter4.ps>.

[Mac99] MACA CHAGÜENDO, Mauricio. Estudio sobre la aproximabilidad del problema del reducto minimal. Santiago de Cali, 1999. Universidad del Valle. Facultad de Ciencias, Posgrado en Matemáticas.

[Pap94] PAPADIMITRIOU, Christos H. Computational complexity. University of California. San Diego: Addison-Wesley Publishing Company, 1994.