UNIVERSIDAD
DE GRANADA

Facultad de Ciencias
Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

Doble Grado en Ingeniería Informática y Matemáticas

TRABAJO DE FIN DE GRADO

Complejidad Algorítmica de Resolución Aproximada de Problemas de Optimización

Presentado por:
Alberto Luque Infante

Tutor:
Serafín Moral Callejón
Ciencias de la computación e Inteligencia Artificial

Curso académico 2020-2021

Complejidad Algorítmica de Resolución Aproximada de Problemas de Optimización

Alberto Luque Infante

Alberto Luque Infante *Complejidad Algorítmica de Resolución Aproximada de Problemas de Optimización*.

Trabajo de fin de Grado. Curso académico 2020-2021.

Responsable de tutorización

Serafín Moral Callejón
Ciencias de la computación e Inteligencia Artificial

Doble Grado en Ingeniería Informática y Matemáticas

Facultad de Ciencias
Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

Universidad de Granada

Declaración de originalidad

D./Dña. Alberto Luque Infante

Declaro explícitamente que el trabajo presentado como Trabajo de Fin de Grado (TFG), correspondiente al curso académico 2020-2021, es original, entendida esta, en el sentido de que no ha utilizado para la elaboración del trabajo fuentes sin citarlas debidamente.

En Granada a 9 de julio de 2021

Fdo: Alberto Luque Infante

Índice general

Índice de figuras

Resumen

En este trabajo se estudia la complejidad computacional de los algoritmos aproximados que resuelven problemas de optimización. El estudio se centra en la clasificación de los problemas de optimización combinatorios en distintas clases de complejidad (NPO, PO, APX, PTAS, FPTAS), así como en el estudio de la completitud de las mismas. Para esto último, inmersos en el modelo PCP (Probabilistic Checkeable Proofs), a partir del Teorema de caracterización PCP, se prueban algunos resultados de no aproximabilidad y de completitud para problemas concretos. Finalmente, se desarrolla una biblioteca en Python que incorpora algunos de los algoritmos aproximados estudiados a lo largo del trabajo para ciertos problemas de optimización NP-difíciles.

Palabras clave: Aproximación, Optimización, Complejidad Computacional, Clases de Complejidad, NPO, APX, PTAS, FPTAS, Completitud.

Summary

On this work we study the computational complexity in the approximate resolution of optimization problems. It is focused on the classification of combinatorial optimization problems based on their approximabilities properties.

The work is divided into four main parts. The first one (chapter 1) is where we introduce some of the main concepts about computational complexity theory and where we present the notation that is going to be followed throughout the work.

The second part involve the chapters 2 and 3. In this part, we study the NPO class and the classes that are contained in NPO (APX, FPTAS, PTAS).

In the third part (chapters 4 and 5) the PCP model is used in order to prove some non-approximability and completeness results.

The last part (chapters 6 and 7) is the most practical one. We discuss about the use of metaheuristics against the studied polynomial algorithms with guaranteed performance, and we develop a Python library where most of the studied algorithms are implemented.

Now we proceed with an individual description of the chapters:

- In the **first chapter**, we present some of the central notions of computational complexity. We start introducing the concept of Turing Machine, decision problems and their classification into the NP and P complexity classes, and we present the NP-completeness theory. Finally, we describe some of the problems (considering both decision and optimization versions) that are mentioned throughout the work. All of these are problems whose decision and optimization versions are a NP-complete and a NP-hard problem, respectively.
- In the **second chapter**, we deepen in the notion of optimization problem and we define the NPO and PO classes. Then, we study the relation of these concepts with the NP-hardness.

- We start the **third chapter** describing the different ways of measuring the quality of an approximation. After that, we focus on the type of approximation that we are most interested in: performance guarantee approximation. We continue defining the main approximation classes included in *NPO*: APX, PTAS and FPTAS. Finally, we study a special case of approximation problems: the pseudo-polynomial problems.
- In the **fourth chapter**, we introduce the PCP model. Using a new computation model, the probabilistic Turing machine, we develop the notion of probabilistically checkable proof (in short, PCP). Then, we see how probabilistically checkable proofs can be used in a rather surprising way to show non-approximability results for NP-hard optimization problems. Particularly, we prove that the problem MAX3-SAT is not in PTAS.
- In the **fifth chapter**, we use the AP-reducibility to identify the hardest problems within NPO and APX, respectively: that is, we prove several completeness results in both APX and NPO. We focus on proving that MAX3-SAT is APX-complete. We also introduce a new family of approximation classes, with a special interest in exp-APX, where we can find the Traveling Salesman Problem as a complete problem for this class.
- The **sixth** chapter is a short reflection about the comparison between metaheuristics and the studied polynomial algorithms with guaranteed performance. We also take a brief look at the most relevant metaheuristics.
- Finally, in the **seventh** chapter we develop a Python library where we implement most of the approximation algorithm studied throughout the work.

Keywords: Approximation, Optimization, Computational Complexity, Complexity Classes, NPO, APX, PTAS, FPTAS, NP-hardness, Completeness.

Introducción

La noción de aproximación siempre ha estado presente en numerosos campos de la física, las matemáticas y la computación. Por ejemplo, en análisis numérico o en geometría computacional se trabaja con aproximaciones, puesto que computacionalmente no podemos trabajar con precisión arbitraria y debemos truncar la representación de los números reales. Por otra parte, en numerosas disciplinas se utiliza este concepto para construir modelos u objetos matemáticos simples que aproximen a otros mucho más complejos.

A principios de la década de los 70, Cook formalizó el concepto de NP-completitud, en un famoso artículo de 1971 titulado "The Complexity of Theorem Proving Procedures"[Coo71], donde también formuló el problema de la relación entre las clases de complejidad P y NP.

La creencia de que los problemas NP-completos no pueden ser resueltos por algoritmos en tiempo polinómico hizo que los investigadores se plantearan buscar otro tipo de estrategias para lidiar con estos problemas. Como la mayoría de los problemas NP-completos son las versiones de decisión de problemas de optimización, surgió de forma natural la siguiente pregunta: ¿se pueden conseguir algoritmos polinómicos en tiempo que, aunque no encuentren la solución óptima exacta, consigan soluciones muy cercanas a ellas?

Para abordar esta pregunta, se definió una clase de problemas de optimización, que cumplen la propiedad de que su versión de decisión es un problema de NP, denominada NPO (NP optimization problems), cuyos problemas fueron objeto de estudio para tratar de encontrar algoritmos que proporcionaran soluciones de calidad en tiempo polinómico.

Uno de los primeros papers en los cuales se se analizaba el comportamiento de un algoritmo aproximado es el de Graham en 1966. En la década de los 70, el estudio de los algoritmos aproximados comenzó a ser más sistemático y se publicaron los trabajos que se considera que han establecido los primeros conceptos básicos relativos a los algoritmos aproximados, como son

el de Garey, Graham y Ullman en 1972, Sahni en 1973, Johnson en 1974 y Nigmatullin en 1976.

Dentro de este tipo de algoritmos, particularmente tienen interés aquellos que ofrecen una garantía de aproximación, que nos aseguran encontrar soluciones cercanas a la óptima para todas las instancias del problema.

En este trabajo se van a estudiar estos algoritmos, que permitirán clasificar los problemas de NPO en una jerarquía de clases, en base a la calidad de las soluciones encontradas por los algoritmos que los aproximan.

A finales de los 70, ya se podía encontrar una amplia literatura acerca de los algoritmos aproximados y sus clases de aproximación. En particular, Horowitz y Sahni en 1978 y Garey y Johnson en 1979 publicaron en sus libros los conceptos básicos y la terminología (algoritmo aproximado, esquema de aproximación polinómico, esquema de aproximación polinómico total) que ha sido la base a partir de la cual se ha desarrollado todo el trabajo en este campo.

Los **objetivos iniciales** del trabajo establecidos antes de comenzar el estudio eran los siguientes:

1. Estudio teórico de los problemas de optimización combinatorios.
2. Estudio de las clases de complejidad para la resolución aproximada de estos problemas. Concretamente, las clases NPO, APX, PTAS y FPTAS.
3. Estudio de la completitud de las clases anteriores.
4. Construcción de una biblioteca para la resolución de problemas de optimización aproximada en Python. Centrada fundamentalmente en problemas de grafos, como el mínimo cubrimiento por vértices o el corte máximo.

Los primeros dos objetivos se han conseguido de forma satisfactoria. El tercer objetivo se ha conseguido de forma parcial. El estudio de la completitud para estas clases de aproximación ha resultado ser un tema un poco más extenso y complejo de lo esperado, por lo que se ha tratado de sintetizar los aspectos más relevantes. Se ha estudiado con detalle sobre todo la APX-completitud. También se ha incluido contenido acerca de la completitud en *NPO* y en exp-APX. El estudio de los problemas completos para las demás clases se deja como trabajo futuro.

El cuarto objetivo también se ha conseguido. Se ha desarrollado una biblioteca donde se han implementado los algoritmos aproximados estudiados a lo largo del trabajo.

Por otra parte, en este trabajo no encontramos una división clara entre las actividades desarrolladas en el ámbito de las matemáticas y la informática. Este trabajo lo podemos describir como un proyecto que complementa y profundiza en la asignatura **Modelos Avanzados de Computación**, impartida en el grado de Ingeniería Informática, pero cuyo contenido es puramente matemático en su mayor parte. Por tanto, todo este trabajo se encuentra enmarcado en el campo de la ciencia de la computación teórica, que la podemos ver como un subconjunto de las matemáticas y la ciencia de la computación que se centra en los aspectos más abstractos y formales de la informática.

Las principales fuentes consultadas han sido [APMS+99], [AB09] y [GJ90a]. Las demás fuentes consultadas se encuentran debidamente citadas a lo largo de trabajo.

1 Conceptos básicos

Antes de comenzar con el contenido principal del trabajo, debemos conocer algunos conceptos básicos que vamos a utilizar a lo largo del estudio. Comenzamos con el concepto más básico y esencial de todos, originalmente definido por Alan Turing y que da comienzo a esta teoría de la Complejidad Computacional.

1.1. Máquina de Turing

Una Máquina de Turing modela matemáticamente a una máquina que opera mecánicamente sobre una cinta. La cinta, de longitud ilimitada, está dividida en celdas, y es similar a la representación de un array unidimensional. Cada celda contiene un símbolo de un alfabeto finito (entre los que se encuentra el símbolo blanco #).

Además de la cinta, la Máquina de Turing incorpora un cabezal de lectura, que siempre apunta a una de las celdas de la cinta, que puede leer y escribir sobre la celda, y que se puede desplazar, de celda en celda, a la derecha o a la izquierda. Formalmente:

Definición 1.1. Máquina de Turing. Una Máquina de Turing (MT) es una séptupla $(Q, A, B, \delta, q_0, \#, F)$ en la que

- Q es un conjunto finito de estados.
- A es un alfabeto de entrada.
- B es el alfabeto de símbolos de la cinta que incluye a A
- δ es la función de transición que asigna a cada estado $q \in Q$ y símbolo $b \in B$, el valor $\delta(q, b)$ que puede ser vacío (no definido) o una tripleta (p, c, M) donde $p \in Q$, $c \in B$, $M \in I, D$ donde I indica izquierda y D indica derecha.
- q_0 es el estado inicial.

- # es un símbolo de $B \setminus A$ llamado símbolo blanco.
- F es el conjunto de estados finales.

Definición 1.2. Una **configuración** de una Máquina de Turing es una tripleta (q, w_1, w_2) donde $q \in Q$, $w_1, w_2 \in B^*$.

Una configuración ofrece una descripción acerca de la situación en la que se encuentra la máquina en un momento determinado. Indica el estado q en el que se encuentra, la palabra w_1 que aparece a la izquierda del cabezal de lectura (eliminando la sucesión infinita de blancos de su izquierda de las casillas que son distinto de blanco) y la palabra w_2 que se obtiene empezando en el cabezal de lectura hacia la derecha (eliminando la sucesión infinita de blancos a la derecha de las casillas que son distinto de blanco).

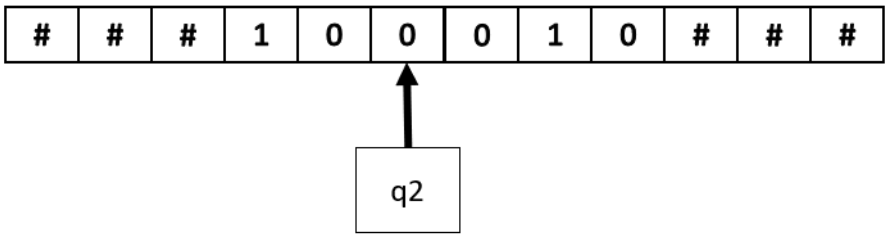


Figura 1.1: Ejemplo de MT cuya configuración en este momento es $(q_2, 10, 0010)$

Si definimos ϵ como la palabra vacía, para un $u \in A^*$, la configuración inicial de una MT es (q_0, ϵ, u), o $(q_0, \epsilon, \#)$ si $u = \epsilon$. Según nos movamos por la cinta a la derecha o a al izquierda, la MT va cambiando su configuración. El cambio desde una configuración a la siguiente se conoce como **paso de cálculo**. Formalmente:

Definición 1.3. Paso de cálculo (movimiento a la izquierda). Sea $M = (Q, A, B, \delta, q_0, \#, F)$ una Máquina de Turing y supongamos que $\delta(q, a) = (p, b, I)$ es una transición. Decimos que de la configuración $(q, c_1...c_n, ad_2...d_m)$ llegamos en un **paso de cálculo** a la configuración $(p, c_1...c_{n-1}, c_nbd_2...d_m)$, lo que se denota como $(q, c_1...c_n, ad_2...d_m) \vdash (p, c_1...c_{n-1}, c_nbd_3...d_m)$ donde se supone que:

- Si $c_1...c_n = \epsilon$, entonces $c_1...c_{n-1} = \epsilon$ y $c_n = \#$.
- Se eliminan los blancos a la derecha de la palabra $c_n b d_3...d_m$ excepto el primero si toda la palabra está formada por blancos.

Definición 1.4. Paso de cálculo (movimiento a la derecha). Sea $M = (Q, A, B, \delta, q_0, \#, F)$ una Máquina de Turing y supongamos que $\delta(q, a) = (p, b, D)$ es una transición. Decimos que de la configuración $(q, c_1...c_n, ad_2...d_m)$ llegamos en un **paso de cálculo** a la configuración $(p, c_1...c_n b, d_2...d_m)$, lo que se denota como $(q, c_1...c_n, ad_2...d_m) \vdash (p, c_1...c_n b, d_3...d_m)$ donde se considera que:

- Si $m = 1$ entonces $d_2 d_3...d_m = \#$.
- Se eliminan todos los blancos a la izquierda en $c_1...c_n b$.

Definición 1.5. Relación de pasos de cálculo. Si R y R' son configuraciones de una Máquina de Turing $M = (Q, A, B, \delta, q_0, \#, F)$, se dice que desde R se llega en una sucesión de pasos de cálculo a R', lo que se denota como $R \overset{*}{\vdash} R'$, si y solo si existe una sucesión finita de configuraciones $R_1, ..., R_n$ tal que $R = R_1, R' = R_n$ y $R_i \vdash R_{i+1}, \forall i < n$.

Definición 1.6. Lenguaje aceptado por una Máquina de Turing. Si M es una Máquina de Turing, entonces el lenguaje aceptado es el conjunto de palabras $L(M)$ tales que $u \in L(M)$ si y solo si existen $w_1, w_2 \in A^*$ y $q \in F$ tales que $(q_0, \epsilon, u) \overset{*}{\vdash} (q, w_1, w_2)$.

Es decir, desde la configuración inicial asociada a u se puede llegar mediante una sucesión de pasos de cálculo a una configuración en la que estamos en un estado final.

Definición 1.7. Dada una Máquina de Turing $M = (Q, A, B, \delta, q_0, \#, F)$, la **función f calculada** por esta MT es una función $f : D \longrightarrow B^*$ tal que $D \subseteq A^*$ es el conjunto de entradas para los que la MT termina y si $u \in D$, entonces $f(u)$ es el contenido de la cinta cuando la MT termina excluyendo los símbolos en blanco.

Definición 1.8. Función Parcialmente Calculable. Una función f se dice que es parcialmente calculable cuando existe una MT que la calcula.

Definición 1.9. Función Calculable Total. Si una función es parcialmente calculable y $D = A^*$ (la MT termina en todas las entradas) se dice que es calculable total.

Otro concepto interesante es el de computación no determinista. Este modelo de computación permite actuar de forma diferente ante una misma entrada. Este concepto se formaliza mediante la Máquina de Turing no determinista. Intuitivamente, es una Máquina de Turing en la que, para ciertas configuraciones, puede comportarse de más de una forma distinta.

Definición 1.10. Máquina de Turing no determinista (MTND). MT en la que puede haber más de una transición posible para una configuración dada. Es decir, $\delta(q, a)$ puede ser un conjunto finito de tripletas $\{(q_1, b_1, M_1), ..., (q_k, b_k, M_k)\}$.

La definición de lenguaje aceptado por una MTND es la misma que la vista en la definición 1.6, con la diferencia de que, al no existir una única computación posible, dada la configuración inicial asociada a la palabra de entrada u, puede que haya algunas computaciones posibles en las que no se llegue a un estado final. La palabra u pertenece al lenguaje con que exista al menos una en la que sí.

Se puede comprobar fácilmente que tiene la misma potencia de cálculo que la MT determinista. Este resultado se puede consultar en el teorema 8.11 de [HMU07].

A la hora de medir los recursos consumidos (complejidad) por una MT tenemos que tener en cuenta lo siguiente:

- En una MT, el número de pasos (tiempo) para una entrada u es el número de pasos de cálculo entre la configuración de entrada y la última configuración.
- Una MT tiene complejidad $t(n)$ en tiempo si para toda entrada de longitud n, la MT termina en $t(n)$ o menos pasos.
- En una MTND, el número de pasos (tiempo) para una entrada u es el número de pasos para el cálculo más largo posible para esa entrada. Si hay una secuencia de cálculos que no termina, entonces el tiempo es infinito.
- Si decimos que una MTND tiene complejidad $t(n)$ en tiempo, quiere decir que todos los posibles cálculos de la MTND terminan en $t(n)$ o menos pasos donde n es la longitud de la entrada.

- Si una MTND tiene complejidad $t(n)$, la MT que la simula tiene complejidad $O(d^{t(n)})$ donde d es una constante mayor que uno.

1.2. Problemas de Decisión

Ahora pasamos a concretar formalmente otro concepto: el concepto de problema.

Definición 1.11. Un problema Π está compuesto por:

- Un conjunto X de entradas. Un elemento $x \in X$ se llama una entrada o instancia.
- Un conjunto Y de soluciones. Un elemento $y \in Y$ se llama una solución.
- Una aplicación $R : X \times Y \longrightarrow \{0, 1\}$ que representa la relación que debe de existir entre las entradas y las soluciones. Si $R(x, y) = 1$, simplemente escribiremos $R(x, y)$, y si $R(x, y) = 0$, escribiremos $\neg R(x, y)$. Es decir, x e y están relacionados cuando y es una solución asociada a la entrada x para el problema Π

Los problemas se resuelven mediante algoritmos. En informática, el concepto que se suele tener de algoritmo es el de un programa escrito en un lenguaje de programación, pero es totalmente equivalente al concepto de Máquina de Turing, que es más apropiado para razonamientos teórico-matemáticos.

Un algoritmo ALG resuelve un problema Π cuando el argumento de dicho algoritmo es un elemento $x \in X$ y $ALG(x)$ es un $y \in Y$ tal que $R(x, y) = 1$.

Los problemas se pueden dividir en diferentes categorías, pero nosotros nos vamos a centrar en dos tipos concretos: los Problemas de Decisión y los Problemas de Optimización.

Los **Problemas de Decisión** son aquellos en los que las soluciones son $Y = \{SI, NO\}$ y cada entrada x tiene una única solución. Por ejemplo, dado un grafo no dirigido, determinar si tiene un circuito hamiltoniano.

Para una entrada $x \in X$, se suele notar como $\Pi(x)$ a la única solución $y \in \{SI, NO\}$ de forma que $R(x, y) = 1$. Por ejemplo, para el problema del circuito hamiltoniano, una entrada x será un grafo no dirigido y $\Pi(x)$ será SI o NO dependiendo de si el grafo x tiene un circuito hamiltoniano o no.

Los problemas de decisión son especialmente importantes ya que son simples y es fácil razonar sobre ellos, además de que cualquier otro problema tiene asociado un problema de decisión.

1.2.1. Problemas de Decisión y Lenguajes

En muchos casos, la teoría está desarrollada utilizando lenguajes. Las entradas se codifican como palabras del alfabeto A. El lenguaje asociado a un problema de decisión lo podemos definir como $L(\Pi) = \{x \in X : \Pi(x) = \text{`}SI\text{`}\}$. Es decir, el lenguaje está formado por aquellas entradas que tienen respuesta afirmativa. Podemos plantear el problema de decisión como: dado el lenguaje L y una entrada $x \in A^*$, determinar si $x \in A$.

Para medir la complejidad de un problema de decisión podemos medir la complejidad de su lenguaje asociado. Un lenguaje se dice de complejidad $f(n)$ si existe una Máquina de Turing que acepta el lenguaje y tiene complejidad $f(n)$.

1.3. Problemas de optimización

En los problemas de optimización, el objetivo principal es optimizar (minimizar o maximizar) una función definida sobre un conjunto de soluciones factibles asociadas a la entrada. Por ejemplo, el problema del viajante de comercio.

La caracterización formal para los problemas de optimización que vamos a utilizar durante todo el trabajo es la siguiente.

Definición 1.12. Un problema de optimización $\mathcal{P}$ está caracterizado por una cuaterna de objetos $\{I_{\mathcal{P}}, SOL_{\mathcal{P}}, m_{\mathcal{P}}, goal_{\mathcal{P}}\}$, donde:

1. $I_{\mathcal{P}}$ es el conjunto de instancias del problema $\mathcal{P}$.
2. $SOL_{\mathcal{P}}$ es una función que asocia a cada $x \in I_{\mathcal{P}}$ el conjunto de soluciones factibles de x.

3. $m_{\mathcal{P}}$ es la función de medida, definida para cada pareja (x, y) tal que $x \in I_{\mathcal{P}}$ y $y \in SOL_{\mathcal{P}}(x)$. Es decir, para (x, y), $m_{\mathcal{P}}(x, y)$ es el valor numérico entero (coste o beneficio) que tiene la solución factible y para la instancia x del problema.
4. $goal_{\mathcal{P}} \in \{MIN, MAX\}$ es una variable que nos indica si el problema es de maximización o de minimización.

Según $goal_{\mathcal{P}}$ sea MIN o MAX, buscamos, para cada instancia $x \in I_{\mathcal{P}}$, el elemento $y^* \in SOL_{\mathcal{P}}(x)$ tal que su beneficio o coste $m_{\mathcal{P}}(x, y^*)$ sea máximo o mínimo respectivamente, entre los elementos de $SOL_{\mathcal{P}}(x)$. El conjunto de soluciones óptimas para una instancia x lo denotamos por $SOL^*_{\mathcal{P}}(x)$.

Esto se conoce como la versión constructiva del problema de optimización. Si solo buscamos obtener el valor del coste o beneficio óptimo sin importar cuál es el elemento y^*, se trata de la versión de evaluación.

El **problema de decisión asociado a un problema de optimización** se conoce como un problema de umbral: partimos de los mismos datos de un problema de optimización más un umbral K, y ahora nos preguntamos si existe una solución de valor mayor o menor que el valor K según sea un problema de máximo o mínimo. Ejemplo: dado un caso del problema del viajante de comercio y un valor K, determinar si existe un circuito de coste menor o igual que K.

Este tema se abordará con más detalle en el capítulo 2.

1.4. Clases de Complejidad

Vamos a medir y clasificar la complejidad de un problema según los recursos que se consumen al encontrar la solución. Concretamente, estos recursos son el tiempo y el espacio, que intuitivamente se traduce en el tiempo que se tarda y la memoria que se necesita para encontrar la solución. Nosotros nos vamos a centrar sobre todo en el tiempo.

Definición 1.13. La clase **TIEMPO(f)** está formada por todos los lenguajes aceptados por una Máquina de Turing determinista en tiempo $O(f(n))$.

La clase **NTIEMPO(f)** está formada por todos los lenguajes aceptados por una Máquina de Turing no determinista en tiempo $O(f(n))$.

Las Clases de Complejidad más importantes relacionadas con la medición del tiempo son la siguientes:

- Clase polinómica $\mathbf{P} = \bigcup_{j>0} TIEMPO(n^j)$. Es decir, está formada por todos los lenguajes reconocidos en tiempo polinómico por una MT determinista.
- Clase polinómica no determinista $\mathbf{NP} = \bigcup_{j>0} NTIEMPO(n^j)$. Es decir, está formada por todos los lenguajes reconocidos en tiempo polinómico por una MT no determinista.

Existen diferentes definiciones equivalentes para las distintas clases de complejidad. Por ejemplo, otra definición alternativa para la clase NP que será similar a la que usaremos para definir las clases de problemas de optimización es la siguiente:

La clase NP es la clase de todos los problemas de decisión cuyas soluciones constructivas pueden ser verificadas en tiempo polinómico sobre el tamaño de la entrada. Es decir, si dada una entrada x para un problema, podemos comprobar si un objeto $y(x)$, que depende de x, pertenece al conjunto de las instancias positivas del problema en tiempo polinómico.

1.5. Reducciones

Las reducciones entre problemas se utilizan para relacionar la complejidad de dos problemas. Destacan dos tipos de reducciones:

- **Reducciones Karp**: Un problema de decisión Π_1 es **reducible** a un problema de decisión Π_2

  $$\Pi_1 \propto \Pi_2$$

  si y solo si existe un algoritmo (una MT) que en tiempo polinómico calcula una función $R : D_{\Pi_1} \longrightarrow D_{\Pi_2}$ de tal manera que

  $$\Pi_1(x) = \Pi_2(R(x)) \;\; \forall x \in D_{\Pi_1}$$

  La reducción de Π_1 a Π_2 indica que si obtuviésemos una solución sencilla para Π_2, entonces componiéndola con la reducción, podríamos obtener una solución para Π_1. Lo contrario no tiene por qué verificarse.

En definitiva, la reducción nos indica que, en términos de complejidad, Π_2 es al menos tan difícil como Π_1.

Se dice que un problema es **NP-Completo** si y solo si es un problema de NP y cualquier otro problema de NP se reduce a él.

El **Teorema de Cook-Levine** demuestra que el problema SAT (definido en la siguiente sección) es NP-Completo construyendo una reducción entre un problema cualquiera de NP y SAT. A partir de conocer el hecho de que SAT es NP-Completo, se pueden construir reducciones desde SAT a otros problemas y de esta manera demostrar, de forma encadenada, que otros problemas de NP son NP-completos sin tener que hacer la reducción desde un problema genérico de NP.

- **Reducciones Turing**: Un problema Π_1 se reduce Turing a Π_2, lo que se representa como

$$\Pi_1 \propto_T \Pi_2$$

si y solo si Π_1 se puede resolver en tiempo polinómico mediante un algoritmo que puede llamar a una función que resuelve Π_2, contando cada llamada como un paso de cálculo.

La reducibilidad Turing es un concepto más débil que el que hemos visto de reducibilidad Karp: Si Π_1 se reduce a Π_2 entonces se puede construir una reducibilidad Turing, que simplemente consistiría en una sola llamada a la composición de la función de reducción entre Π_1 y Π_2 y el algoritmo que resuelve Π_2.

Decimos que esta reducción es más débil en el siguiente sentido. Sea R la función de reducción entre las entradas de Π_1 y Π_2. Para una entrada x de Π_1, el algoritmo que resuelve Π_1 a través de uno que resuelva Π_2 podría ser:

$\Pi_1(x)$:
$y \leftarrow R(x)$
return $\Pi_2(y)$

Si permitimos manipular la salida y hacer más de una llamada a $\Pi_1(y)$ obtenemos una reducción Turing. Por tanto, siempre que se dé una reducción Karp, se dará una reducción Turing.

El siguiente concepto se aplica para cualquier tipo de problema, no es exclusivo para los problemas de decisión, como sí lo es la NP-completitud:

Se dice que un problema Π es **NP-difícil** si y solo si existe un problema NP-completo Π_c que se puede reducir (Turing) a Π ($\Pi_c \propto_T \Pi$) .

Un ejemplo de una clase de problemas que está dentro de la clase NP-difícil es la clase *Co-NP*, formada por los problemas cuyo problema contrario está en NP. El **problema contrario** de un problema de decisión Π es el problema $\overline{\Pi}$, que intercambia las salidas SI y NO. De forma más precisa, $\Pi(x) = SI \iff \overline{\Pi}(x) = NO$.

Un problema $\overline{\Pi} \in$ Co-NP está en la clase NP-difícil porque podemos construir una reducción Turing desde Π, que es un problema NP-completo, a $\overline{\Pi}$ ($\Pi \propto_T \overline{\Pi}$). El algoritmo que resuelve $\overline{\Pi}$, para una entrada x podría ser:

$\Pi(x)$:
$y \leftarrow \overline{\Pi}(x)$
return $\overline{y}$

Por el argumento usado se puede ver claramente que también se da la reducción recíproca ($\overline{\Pi} \propto_T \Pi$).

Un ejemplo visual de cómo están organizadas las clases es el siguiente:

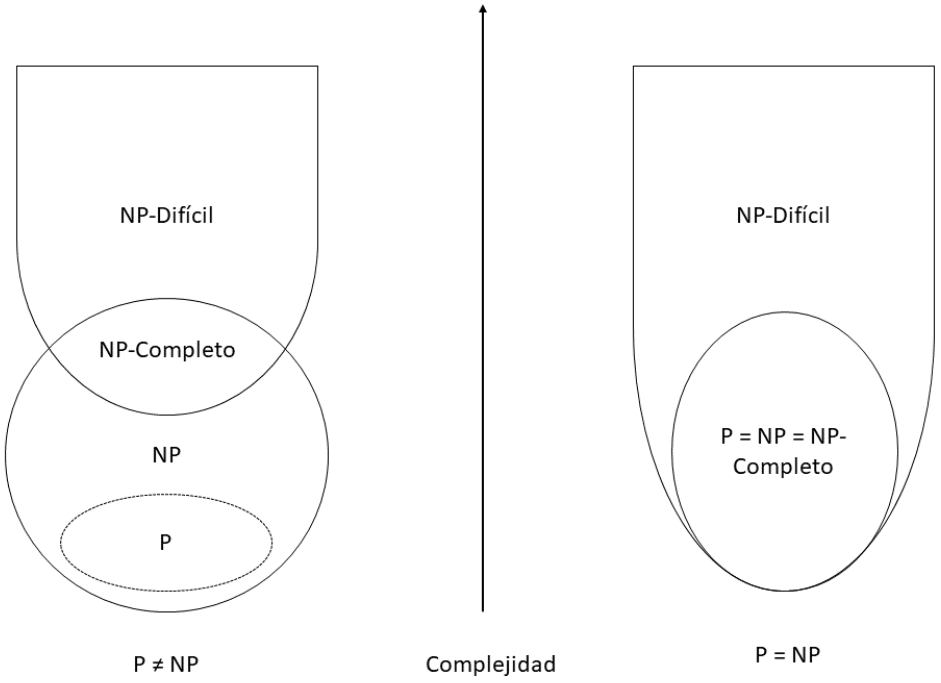


Figura 1.2: Diagrama de clases de complejidad para el caso en el que $P \neq NP$ y $P = NP$

1.6. Lista de problemas

En esta sección vamos a presentar los principales problemas que vamos a estudiar o mencionar a lo largo del trabajo. Sobre todo nos vamos a centrar en problemas de grafos. Se tratan de problemas que normalmente tienen dos versiones: una de decisión, que es un problema NP-Completo, y una de optimización.

1.6.1. Problema del Clique Máximo (CM)

Dado un grafo no dirigido $G = (V, E)$, un **clique** es un subconjunto maximal totalmente conectado. Es decir, un subconjunto $V_t \subseteq V$ tal que $\forall v_1, v_2 \in$

V_t, $\{v_1, v_2\} \in E$, y que no está estrictamente incluido en otro conjunto que cumpla esta propiedad.

- **Versión de optimización:** Dado un grafo no dirigido $G = (V, E)$, encontrar un conjunto totalmente conectado de tamaño máximo (clique).
- **Versión de decisión:** : Dado un grafo no dirigido $G = (V, E)$ y un número natural $J \leq |V|$, determinar si existe un clique de tamaño mayor o igual que J (o también es equivalente a comprobar la existencia de un conjunto totalmente conectado de tamaño mayor o igual que J.

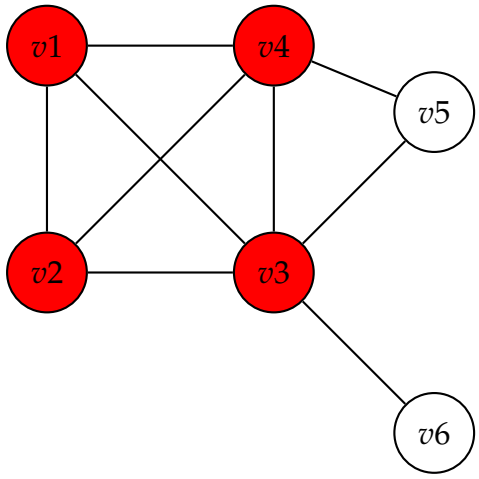


Figura 1.3: Ejemplo de clique de tamaño 4

1.6.2. Problema del Cubrimiento por Vértices (CV)

Dado un grafo no dirigido $G = (V, E)$ y un subconjunto $V_c \subseteq V$, se dice que V_c es un **cubrimiento por vértices** de G, si y solo si toda arista del grafo tiene un extremo en V_c :

- **Versión de optimización:** Dado un grafo no dirigido $G = (V, E)$, encontrar el cubrimiento por vértices V_c de G de menor número de vértices.
- **Versión de decisión**: Dado un grafo $G = (V, E)$ y un número natural $K \leq |V|$, determinar si existe un cubrimiento por vértices de tamaño menor o igual que K.

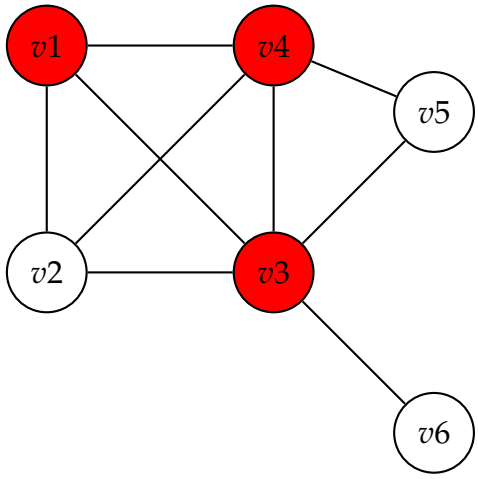


Figura 1.4: Ejemplo de cubrimiento de tamaño 3

1.6.3. Problema del Conjunto Independiente (CI)

Dado un grafo no dirigido $G = (V, E)$ y un subconjunto $V_i \subseteq V$, se dice que V_i es un conjunto independiente de G, si y solo si no hay ninguna arista que una vértices de V_i :

$$\forall u, v \in V_i, \quad \{u, v\} \notin E$$

- **Versión de optimización:** Dado un grafo no dirigido $G = (V, E)$, encontrar un conjunto independiente V_c de G de mayor número de vértices.

- **Versión de decisión:** : Dado un grafo no dirigido $G = (V, E)$ y un número natural $J \leq |V|$, determinar si existe un conjunto independiente de tamaño mayor o igual que J.

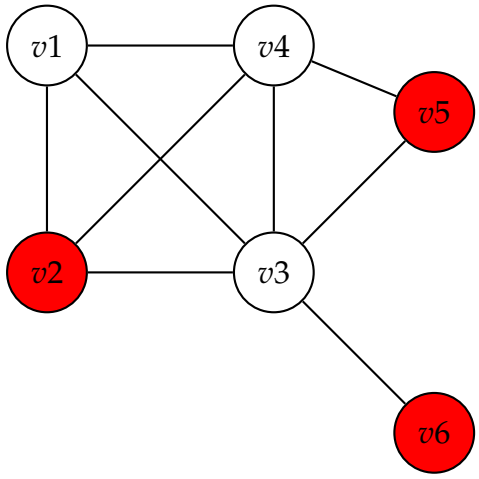


Figura 1.5: Ejemplo de conjunto independiente de tamaño 3

1.6.4. Relación entre los problemas CM, CV y CI

Si $G = (V, E)$ es un grafo no dirigido y $V^* \subseteq V$, entonces las siguientes condiciones son equivalentes:

(i) V^* es un cubrimiento por vértices de G.

(ii) $V \setminus V^*$ es un conjunto independiente de G.

(iii) $V \setminus V^*$ es un subgrafo totalmente conectado del grafo complementario $\overline{G} = (V, \overline{E})$, donde $\overline{E} = V \times V \setminus E$.

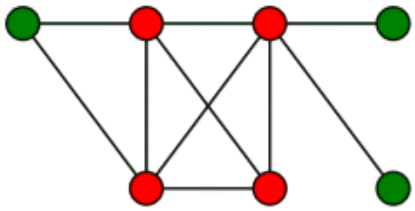


Figura 1.6: Conjunto totalmente conectado (en rojo)

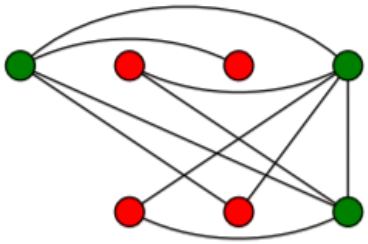


Figura 1.7: Conjunto independiente (en rojo)

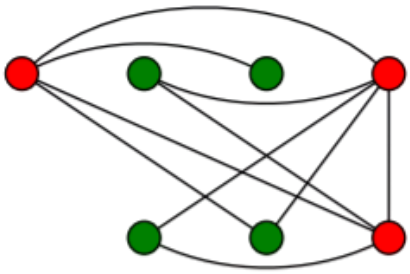


Figura 1.8: Cubrimiento por vértices (en rojo)

1.6.5. Problema del Corte Máximo

Consiste en el siguiente problema de optimización:

- **Versión de optimización:** Dado un grafo no dirigido $G = (V, E)$, partir V en dos conjuntos S y $V \setminus S$ de forma que haya un número máximo de arcos entre S y $V \setminus S$.

- **Versión de decisión:** Dado un grafo no dirigido $G = (V, E)$ y un número $K \leq |E|$, determinar si es posible partir V en dos conjuntos S y $V \setminus S$ de forma que haya un número de arcos entre S y $V \setminus S$ mayor o igual que K. Es un problema NP-Completo.

En el siguiente grafo, tomando S como el conjunto formado por los vértices v4 y v5, obtenemos un corte donde el número de arcos que cruzan es 3.

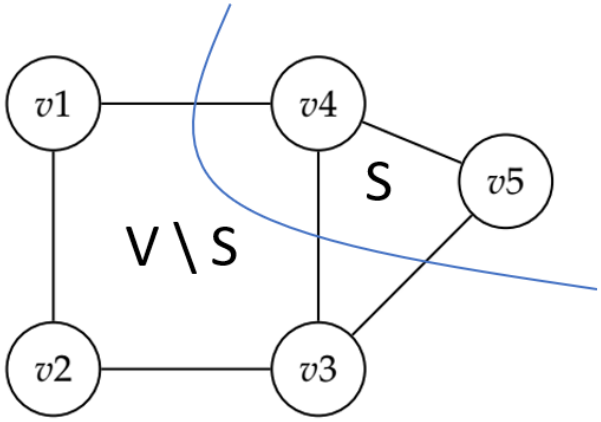


Figura 1.9: Ejemplo de un corte de tamaño 3

1.6.6. Problema de la Mochila

Tenemos un conjunto finito de objetos A. Sea n el número de objetos en A. Cada objeto $a_i \in A$ tiene un tamaño/peso, $w_i \in \mathbb{N}$, y un valor $v_i \in \mathbb{N}$, con $i \in \{1, .., n\}$. Tenemos, además, dos números naturales: W (el tamaño/peso máximo) y K (el valor mínimo).

- **Versión de optimización:** Encontrar un subconjunto $S \in \{1, .., n\}$ tal que $\sum_{i \in S} w_i \leq W$ que tenga valor máximo (que $\sum_{i \in S} v_i$ sea máximo).
- **Versión de decisión:** Dado un umbral K, determinar si existe un subconjunto de objetos $S \subseteq \{1, .., n\}$ tal que $\sum_{i \in S} w_i \leq W$ y $\sum_{i \in S} v_i \geq K$.

1.6.7. Problema del Circuito Hamiltoniano

Un **circuito hamiltoniano** es un camino que parte de un nodo para llegar a él mismo, visitando todos los nodos del grafo una y solo una vez.

Dado un grafo no dirigido $G = (V, E)$, determinar si existe un circuito hamiltoniano.

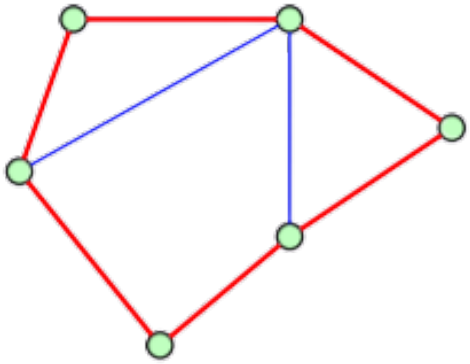


Figura 1.10: Ejemplo de circuito hamiltoniano

1.6.8. Problema del Viajante de Comercio

Dado un conjunto finito de ciudades $C = \{c_1, .., c_m\}$, una función de distancia $d : C \times C \longrightarrow \mathbb{N}$ y una cota $B \in \mathbb{N}$:

- **Versión de optimización:** Encontrar un circuito que visite todas las ciudades una sola vez y de coste mínimo. Suponiendo que las distancias entre ciudades son simétricas, se puede formular el problema utilizando un grafo totalmente conectado donde cada nodo se corresponde con una ciudad, con el objetivo de encontrar el circuito hamiltoniano de menor coste.

  Es decir, determinar un orden de las ciudades $(c_{\pi(1)}, .., c_{\pi(m)})$ tal que

  $$(\sum_{i=1}^{m-1} d(c_{\pi(i)}, c_{\pi(i+1)}) + d(c_{\pi(m)}, c_{\pi(1)}))$$

  sea mínimo.

- **Versión de decisión:** Formulado de la misma forma que en la versión de optimización, determinar si existe un circuito hamiltoniano de coste no superior a B.

  Es decir, determinar si existe un orden de las ciudades $(c_{\pi(1)}, .., c_{\pi(m)})$ tal que

  $$(\sum_{i=1}^{m-1} d(c_{\pi(i)}, c_{\pi(i+1)}) + d(c_{\pi(m)}, c_{\pi(1)})) \leq B$$

1.6.9. Problema de satisfacibilidad booleana (SAT)

Se trata de un problema de decisión. Sea $U = \{u_1, ..., u_m\}$ un conjunto de variables booleanas. Una *asignación de valores de verdad* es una función $f : U \longrightarrow \{TRUE, FALSE\}$. Si u es una variable en U, entonces u y $\neg u$ se llaman *literales* asociados a la variable u. El literal u es verdad bajo f si y solo si, para la variable u, $f(u) = TRUE$. En cambio, el literal $\neg u$ es verdad bajo f si y solo si $f(u) = FALSE$.

Una *cláusula* es una disyunción de literales, donde un literal y su opuesto no pueden aparecer en la misma cláusula (por ejemplo, $\neg u_1 \vee u_2 \vee u_3$). Se dice que una cláusula se satisface bajo una asignación de valores de verdad f cuando al menos uno de sus literales es cierto bajo f. Por ejemplo, una asignación de valores de verdad que satisface la cláusula anterior es $f(u_1) = FALSE$, $f(u_2) = FALSE$, $f(u_3) = FALSE$.

El problema **SAT** consiste en lo siguiente. Dado un conjunto $U = \{u_1, ..., u_m\}$ de variables y una colección C de cláusulas sobre estos símbolos, determinar si las cláusulas son consistentes. Es decir, si existe una asignación de valores de verdad que haga que se satisfagan todas las cláusulas. (En cada cláusula cada variable u_i solo puede aparecer una vez como máximo (como el literal u o el literal $\neg u$)).

Una instancia de SAT también se puede ver como una una fórmula Booleana que está en forma normal conjuntiva (CNF). Es decir, una fórmula Booleana compuesta por una conjunción de cláusulas.

1.6.10. Variantes de SAT

- **3-SAT.** Es un caso particular de SAT donde cada cláusula está formada por 3 literales como máximo.
- **MAX-SAT**. Es una versión de optimización del problema SAT, donde se busca encontrar una asignación de valores de verdad que haga que se satisfagan el mayor número de cláusulas.
- **MAX3-SAT**.Es una versión de optimización del problema 3-SAT. En este caso también se busca encontrar una asignación de valores de verdad que haga que se satisfagan el mayor número de cláusulas, pero dentro del problema 3-SAT, donde cada cláusula está formada por 3 literales como máximo.

2 Complejidad de los Problemas de Optimización

2.1. Problemas de Optimización

Definición 2.1. Un problema de optimización $\mathcal{P}$ está caracterizado por una cuaterna de objetos $\{I_{\mathcal{P}}, SOL_{\mathcal{P}}, m_{\mathcal{P}}, goal_{\mathcal{P}}\}$, donde:

1. $I_{\mathcal{P}}$ es el conjunto de instancias del problema $\mathcal{P}$.
2. $SOL_{\mathcal{P}}$ es una función que asocia a cada $x \in I_{\mathcal{P}}$ el conjunto de soluciones factibles de x.
3. $m_{\mathcal{P}}$ es la función de medida, definida para cada pareja (x, y) tal que $x \in I_{\mathcal{P}}$ y $y \in SOL_{\mathcal{P}}(x)$. Es decir, para (x, y), $m_{\mathcal{P}}(x, y)$ es el valor numérico entero (coste o beneficio) que tiene la solución factible y para la instancia x del problema.
4. $goal_{\mathcal{P}} \in \{MIN, MAX\}$ es una variable que nos indica si el problema es de maximización o de minimización.

Según $goal_{\mathcal{P}}$ sea MIN o MAX, buscamos, para cada instancia $x \in I_{\mathcal{P}}$, el elemento $y^* \in SOL_{\mathcal{P}}(x)$ tal que su beneficio o coste $m_{\mathcal{P}}(x, y^*)$ sea máximo o mínimo respectivamente, entre los elementos de $SOL_{\mathcal{P}}(x)$. El conjunto de soluciones óptimas para una instancia x lo denotamos por $SOL^*_{\mathcal{P}}(x)$.

Esto se conoce como la versión constructiva del problema de optimización. Si solo buscamos obtener el valor del coste o beneficio óptimo sin importar cuál es el elemento y^*, se trata de la versión de evaluación. En general, tienen una dificultad similar.

Ejemplo 2.1. En el problema del cubrimiento por vértices $\mathcal{P}$:

1. Cada $G \in I_{\mathcal{P}}$ es un grafo no dirigido.

2. $SOL_{\mathcal{P}}(G)$ es el conjunto de los cubrimientos por vértices de G (el conjunto de todos los subconjuntos de vértices A tales que toda arista tiene, al menos, un extremo en A).

3. $m_{\mathcal{P}}$ es el número de vértices del cubrimiento.

4. $goal_{\mathcal{P}}$ es MIN, ya que buscamos encontrar el cubrimiento por vértices con el menor número de vértices.

El problema constructivo consiste en, para un $G \in I_{\mathcal{P}}$, encontrar un cubrimiento por vértices óptimo $y^* \in SOL_{\mathcal{P}}(G)$ y su medida $m^*(G)$ (número de vértices del recubrimiento óptimo). En cambio, para el problema de evaluación, nos basta con encontrar el número de vértices del recubrimiento mínimo, sin importar qué vértices forman dicho recubrimiento.

Estos problemas suelen ser equivalentes bajo reducción Turing a los problemas de decisión asociados cuando el problema de decisión es NP-completo.

En estos casos, por lo general, encontrar una solución óptima no es nada sencillo, puesto que los algoritmos que los resuelven suelen ser muy ineficientes. De ahí la importancia de los algoritmos aproximados, ya que muchas veces podemos encontrar una solución muy cercana a la óptima de forma mucho más rápida.

Vamos a ver que, problemas que son de dificultad similar cuando se resuelven de forma exacta, son bastante diferentes cuando intentamos aproximarlos con algoritmos polinómicos: unos no se pueden aproximar con ningún error; otros se pueden aproximar con algún error, pero no con errores muy pequeños; y otros se pueden aproximar con errores arbitrariamente pequeños.

Antes de abordar este tema, vamos a definir las clases de complejidad asociadas a los problemas de optimización.

2.2. Clases de complejidad NPO y PO

Definición 2.2. Un problema de optimización $\mathcal{P} = \{I, SOL, m, goal\}$ pertenece a la clase NPO si:

1. El conjunto de instancias I es reconocible en tiempo polinómico.

2. Existe un polinomio q tal que, dada una instancia $x \in I$, para todo $y \in SOL(x)$ se cumple que $|y| \leq q(|x|)$ y, además, para todo y tal que $|y| \leq q(|x|)$, es decidible en tiempo polinómico si $y \in SOL(x)$.
3. La función de medida m es calculable en tiempo polinómico.

Ejemplo 2.2. El problema del cubrimiento mínimo por vértices pertenece a la clase NPO puesto que:

1. El conjunto de instancias (los grafos no dirigidos) podemos reconocerlos en tiempo polinómico.
2. Una solución factible está formada por un subconjunto de vértices del grafo, por lo que el tamaño de la solución es menor o igual que el tamaño de la entrada (pues esta contiene a todos los vértices del grafo). Por tanto, el polinomio q puede ser la identidad. Comprobar si un subconjunto de vértices es solución, se puede hacer comprobando que cada arista del grafo sea incidente con al menos uno de los vértices del subconjunto, que se puede hacer en tiempo polinómico.
3. La función de medida (el número de vértices del subconjunto) es calculable trivialmente en tiempo polinómico.

Teorema 2.1. *Para todo problema de optimización $\mathcal{P}$ en NPO, el problema de decisión asociado $\mathcal{P}_\mathcal{D}$ pertenece a NP.*

Demostración. Suponemos que $\mathcal{P}$ es un problema de maximización (si fuera de minimización es análogo). Para que $\mathcal{P}_\mathcal{D}$ pertenezca a NP debemos encontrar un algoritmo polinómico no determinista que, dada una instancia $x \in I$ y un entero K, nos resuelva el problema de decisión.

El algoritmo no determinista podría ser de la siguiente forma:

1. Se selecciona de forma no determinista una cadena y tal que $|y| \leq q(|x|)$.
2. Se comprueba si y es solución de x, que se puede comprobar en tiempo polinómico.
3. Si sí es solución, se computa $m(x, y)$, que también se puede hacer en tiempo polinómico.
4. Si $m(x, y) \geq K$ la respuesta es SÍ, y en otro caso la respuesta es NO.

□

Definición 2.3. Un problema de optimización $\mathcal{P}$ pertenece a la clase PO si está en NPO y existe un algoritmo polinómico que, para toda instancia $x \in I$, devuelve una solución óptima $y \in SOL^*(x)$, junto con su valor $m^*(x)$.

Por ejemplo, el problema de obtener el camino más corto entre dos nodos de una grafo está en PO, ya que sabemos que se puede calcular la solución óptima y el coste en $O(n^2)$ donde n es el número de vértices, utilizando un algoritmo de búsqueda en anchura.

Los problemas más interesantes de optimización, y sobre todo, los que más nos interesan en relación al estudio de los algoritmos aproximados, son aquellos que están en NPO pero no en PO, ya que en estos casos ya existe un algoritmo eficiente que resuelve de forma exacta el problema.

La relación entre NPO y PO es parecida a la que hay entre NP y P. La pregunta de si "$PO = NPO$" está estrechamente relacionada con la pregunta "$P = NP$".

Sin embargo, aunque existen pocos problemas entre los más difíciles de NP (los NP-completos) y los de P, sí hay una graduación entre los problemas más difíciles de NPO (los NPO-completos) y los de PO. Problemas que son comparables cuando se resuelven de forma exacta, resultan de distinta dificultad cuando se intentan resolver de forma aproximada.

2.3. Problemas de optimización NP-Difícil

La definición es la misma que dimos en el apartado inicial de conceptos básicos, solo que concretando para el caso de problemas de optimización.

Definición 2.4. Un problema de optimización $\mathcal{P}$ es NP-Difícil si y solo si existe un problema de decisión NP-Completo Π_c tal que $\Pi_c \propto_T \mathcal{P}$

Teorema 2.2. *Sea $\mathcal{P} \in NPO$. Si su problema de decisión asociado P_D es NP-Completo, entonces $\mathcal{P}$ es NP-Difícil.*

Demostración. Tenemos que ver que $P_D \propto_T \mathcal{P}$. Es obvio que con un algoritmo que calcule la solución óptima de $\mathcal{P}$ podemos construir la reducción llamando una vez al algoritmo y comparando el coste de la solución con el valor de K para dar la respuesta al problema de decisión. □

Gracias a este resultado podemos encontrar muchos ejemplos de problemas que son NP-difíciles, como por ejemplo todas las versiones de optimización de los problemas NP-completos presentados anteriormente en la sección 1.6 (lista de problemas).

De este resultado también podemos deducir el siguiente corolario que nos relaciona a las clases P-NP con PO-NPO.

Corolario 2.1. *Si $P \neq NP$, entonces $PO \neq NPO$.*

Demostración. Es equivalente que demostremos que si $PO = NPO$, entonces $P = NP$. Sea P_D un problema NP-Completo que sea el problema de decisión asociado a un problema $\mathcal{P}$ en NPO. Para que $P = NP$, debemos resolver P_D de forma polinómica, ya que al ser P_D NP-Completo, todos los problemas en NP se reducen a él, y este mismo algoritmo los resolvería en tiempo polinómico.

Como $P_D \propto_T \mathcal{P}$, usando que $PO = NPO$, podemos resolver $\mathcal{P}$ en tiempo polinómico, y por tanto el algoritmo utilizado para la reducción Turing nos permite resolver P_D en tiempo polinómico. □

3 Algoritmos Aproximados y Clases de Aproximación

3.1. Aproximación con garantía de comportamiento

Definición 3.1. Dado un problema de optimización $\mathcal{P} = \{I, SOL, m, goal\}$, un algoritmo $\mathcal{A}$ es un algoritmo aproximado para $\mathcal{P}$ si para toda instancia $x \in I$ devuelve una solución aproximada, es decir, una solución factible $\mathcal{A}(x) \in SOL(x)$.

Esta definición es muy general y no es muy práctica, puesto que lo que buscamos como solución aproximada es una solución factible cuyo valor no esté muy "lejos" del óptimo.

Hace falta determinar cómo de "lejos" está la solución obtenida de la óptima, y para ello surgen las siguientes opciones.

3.1.1. Aproximación Absoluta

Definición 3.2. Dado un problema de optimización $\mathcal{P}$, para toda instancia x y para toda solución y de x, el error absoluto de y respecto a x está definido como

$$D(x, y) = |m^*(x) - m(x, y)|$$

donde $m^*(x)$ denota la medida de una solución óptima de la instancia x y $m(x, y)$ denota la medida de la solución y.

Parece razonable encontrar un algoritmo polinómico que, para cada instancia x, sea capaz de garantizar una solución con un error absoluto acotado por una constante independiente de x.

Dado un problema de optimización $\mathcal{P}$ y un algoritmo aproximado $\mathcal{A}$ para $\mathcal{P}$, se dice que $\mathcal{A}$ es un algoritmo absoluto de aproximación si existe una constante k tal que, para toda instancia x de $\mathcal{P}$, $D(x, \mathcal{A}(x)) \leq k$.

Sin embargo, no siempre es posible asegurar que exista un algoritmo de este tipo:

Ejemplo 3.1. Si $P \neq NP$, no existe ningún algoritmo absoluto de aproximación polinómico para el *Problema de la Mochila*.

Sea X un conjunto con n items con valor $p_1, ..., p_n$ y tamaños $a_1, ..., a_n$, y sea b la capacidad de la mochila. Sabemos que este problema es NP-Completo, por lo que no existe un algoritmo polinómico que lo resuelva. Vamos a ver que podríamos encontrar uno en el caso de que existiese un algoritmo de aproximación con error absoluto k.

En ese caso, podemos crear una nueva instancia multiplicando los valores de todos los items p_i por $k + 1$. Con esto, no modificamos el conjunto de soluciones factibles, solo que la suma total será mayor.

Como todos los valores son múltiplos de $k + 1$, la suma total de cualquier solución lo será también. Entonces, la única solución con un error acotado por k es la solución óptima, y por tanto, hemos encontrado la solución óptima del problema en tiempo polinómico, lo que es imposible si $P \neq NP$, y hace que no pueda existir tal algoritmo aproximado.

3.1.2. Aproximación Relativa

Es más interesante y usual, a la hora de mejorar la calidad de la solución aproximada, utilizar otros indicadores como el error relativo y sobre todo, la razón de eficacia.

Definición 3.3. Dado un problema de optimización $\mathcal{P}$, para toda instancia x y para toda solución factible y de x, el error relativo de y respecto a x está definido como

$$E(x, y) = \frac{|m^*(x) - m(x, y)|}{max\{m^*(x), m(x, y)\}}$$

Tanto en los problemas de maximización como en los de minimización, el error relativo es 0 para la solución óptima, y más cercano a 1 cuando la solución es más pobre.

Al igual que con la aproximación absoluta, se dice que un algoritmo $\mathcal{A}$ para un problema de optimización $\mathcal{P}$ es un algoritmo ϵ-aproximado si

$$E(x, \mathcal{A}(x)) \leq \epsilon$$

para toda instancia x de $\mathcal{P}$.

3.1.2.1. Algoritmos δ-aproximados

La herramienta más utilizada, y la que vamos a usar para clasificar los distintos algoritmos aproximados, y que es también una medida relativa es la siguiente.

Definición 3.4. Razón de Eficacia. Dado un problema de optimización $\mathcal{P}$, para toda instancia x de $\mathcal{P}$ y para toda solución factible y de x, la Razón de Eficacia de la solución y respecto a x está definida como

$$R(x,y) = max(\frac{m^*(x)}{m(x,y)}, \frac{m(x,y)}{m^*(x)}) = \begin{cases} \frac{m^*(x)}{m(x,y)} & \text{si } \mathcal{P} \text{ es un problema de maximización} \\ \\ \frac{m(x,y)}{m^*(x)} & \text{si } \mathcal{P} \text{ es un problema de minimización} \end{cases}$$

En ambos casos, cuando la Razón de Eficacia es 1, se trata de una solución óptima, y a medida que va aumentando el valor, la calidad de la solución va disminuyendo.

El Error Relativo y la Razón de Eficacia están claramente relacionados, pues

$$E(x,y) = 1 - \frac{1}{R(x,y)}$$

Definición 3.5. Algoritmos δ-aproximados. Dado un problema de optimización $\mathcal{P}$ y un algoritmo aproximado $\mathcal{A}$ para $\mathcal{P}$, se dice que $\mathcal{A}$ es un algoritmo δ-aproximado si, para toda instancia x de $\mathcal{P}$

$$R(x, \mathcal{A}(x)) \leq \delta$$

Si el coste/beneficio obtenido por el algoritmo lo denotamos como $m(x) = m(x, \mathcal{A}(x))$, tendremos que $\frac{m^*(x)}{m(x)} \leq \delta$ si $\mathcal{P}$ es un problema de maximización y $\frac{m(x)}{m^*(x)} \leq \delta$ para el caso de minimización.

Un problema de optimización $\mathcal{P}$ se dice que es δ-aproximable si existe un algoritmo δ-aproximado polinómico para $\mathcal{P}$.

Como observación, podemos deducir que el valor de δ será un valor racional mayor o igual que uno, siendo uno solo en el caso de obtener la solución óptima con el algoritmo.

Definición 3.6. Umbral de Aproximación. El Umbral de Aproximación de un problema de optimización $\mathcal{P}$ es el ínfimo de los δ tal que $\mathcal{P}$ tiene un algoritmo δ-aproximado polinómico.

Que el umbral de un problema de optimización sea, por ejemplo, 2 no significa que exista un algoritmo 2-aproximado para el problema, porque podría ocurrir que el intervalo para δ donde exista un algoritmo δ-aproximado polinómico para el problema sea $(2, \infty)$.

Por tanto, es posible que haya problemas con un umbral de aproximación igual a 1, sin que eso signifique que exista un algoritmo polinómico exacto que los resuelva. También es posible que haya problemas cuyo umbral de aproximación sea infinito, que indica que no hay algoritmos polinómicos δ-aproximados para ningún δ.

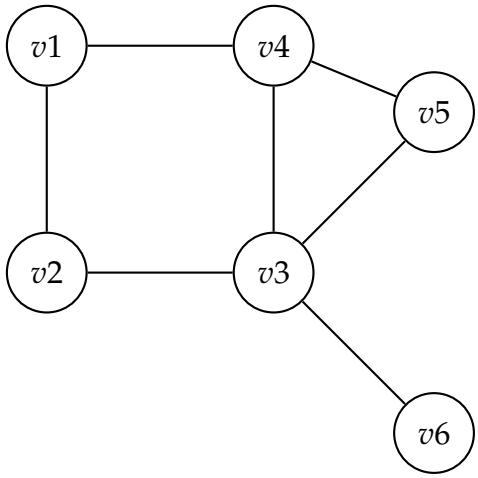
Vamos a ver ahora dos ejemplos de algoritmos δ-aproximados. Un primer algoritmo 2-aproximado para el problema del *cubrimiento mínimo por vértices* y otro segundo algoritmo $2-$aproximado para el problema del *corte máximo.* Para cada uno de ellos, primero vamos a describir el funcionamiento del algoritmo y seguidamente demostraremos que el algoritmo ofrece la garantía de comportamiento indicada.

Ejemplo 3.2. Algoritmo 2-aproximado para el problema del cubrimiento por vértices.

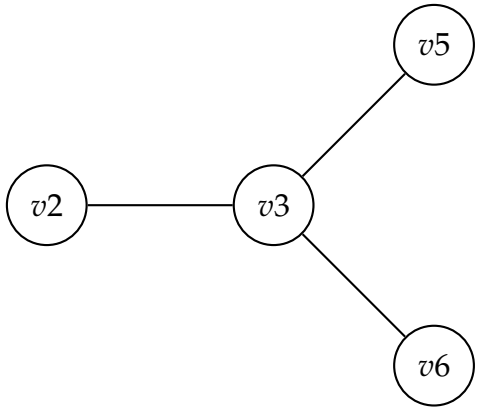
Sea G un grafo no dirigido y C un conjunto donde vamos a almacenar vértices (inicialmente vacío). El algoritmo es el siguiente:

- C = $\emptyset$
- Mientras haya aristas en G:
    - Elegir una arista cualquiera de G.
    - Añadir sus dos extremos a C.
    - Borrar de G los dos nodos y todas sus aristas.

Ejemplo 3.3. Veamos un ejemplo práctico del algoritmo anterior. Suponemos el siguiente grafo:



1. Elegimos por ejemplo la arista v1-v4. $C = \{v1, v4\}$. Eliminamos los vértices y todas sus aristas:



2. Elegimos ahora, por ejemplo, v2-v3. $C = \{v1, v4, v2, v3\}$.
3. Ya no hay más aristas en G, luego los vértices que forman el cubrimiento son v1, v2, v3, v4.

Este algoritmo es efectivamente 2**-aproximado**. Elegimos aristas que no tienen dos vértices en común y para cada una de ellas, al menos uno de sus dos vértices tiene que estar en el cubrimiento, ya que por definición toda arista debe tener un vértice en el cubrimiento.

Como añadimos dos vértices por arista, como mucho habremos obtenido un cubrimiento que tenga el doble de vértices que el cubrimiento óptimo.

En el caso del ejemplo, el cubrimiento óptimo está formado por los vértices v2, v3, v4. Lo hubiéramos obtenido en el caso de elegir como primera arista la v3-v4. No obstante, hemos obtenido un recubrimiento de tamaño menor que el doble del óptimo ($4 < 6$).

Actualmente es el mejor algoritmo δ-aproximado que se conoce para este problema (no existe un algoritmo aproximado para un δ constante menor que 2). También hay un resultado que nos dice que, si $P \neq NP$, no puede existir un algoritmo aproximado para este problema para $\delta = 1.1659$ [Kan].

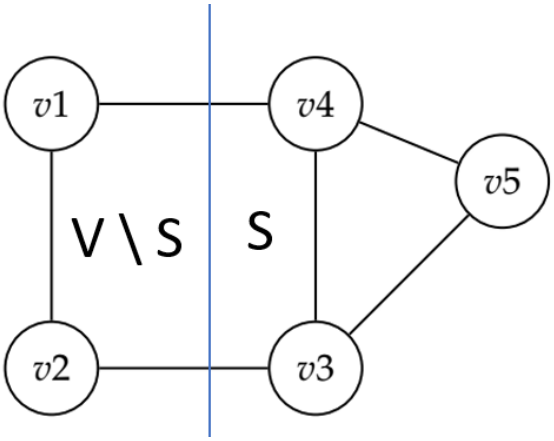
Ejemplo 3.4. Algoritmo 2-aproximado para el problema del corte máximo.

Sea G un grafo no dirigido. Suponemos que no existen arcos que unen un nodo con él mismo: estos arcos se pueden quitar ya que no están nunca en el corte.

Si v es un nodo y A un subconjunto de nodos, definimos $arc(v, A)$ como el número de arcos que unen v con A.

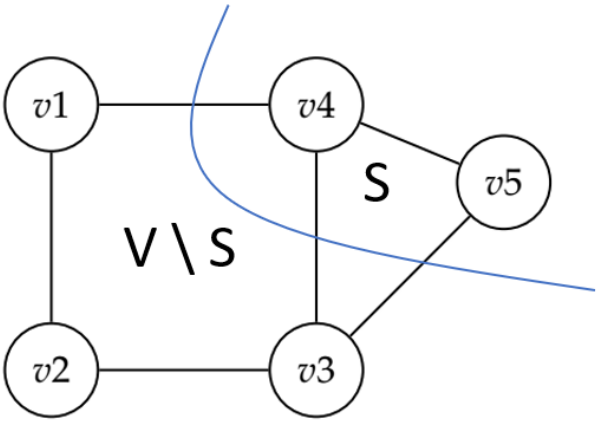
- Comenzamos con un conjunto S arbitrario
- Mientras S cambie:
    - Para cada vértice v del grafo
        - Si el vértice está en S
            - Si $arc(v, V \setminus S) < arc(v, S)$, eliminar v de S.
        - Si el vértice no está en S
            - Si $arc(v, V \setminus S) > arc(v, S)$, añadir v a S.

Ejemplo 3.5. Dado el siguiente grafo, tomamos como conjunto S uno arbitrario:

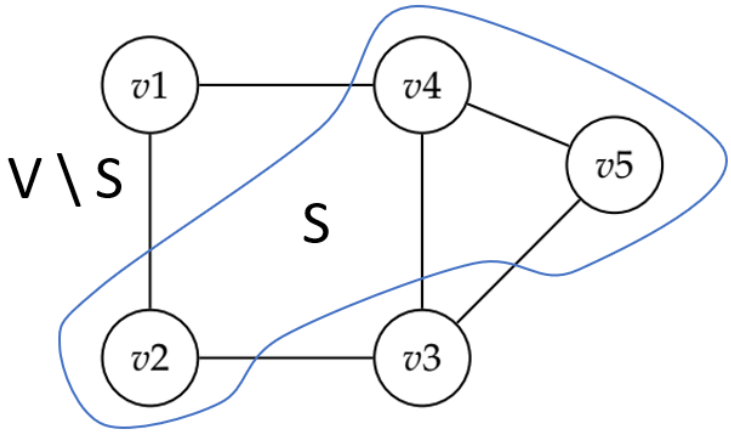


Comenzamos a estudiar cada vértice del grafo:

1. $v1$: No está en S, $arc(v1, V \setminus S) = 1$ y $arc(v1, S) = 1$, luego mantenemos $v1$ en $V \setminus S$.
2. $v2$: Es idéntico a $v1$.
3. $v3$: Está en S, $arc(v3, V \setminus S) = 1$ y $arc(v3, S) = 2$, luego eliminamos $v3$ de S.



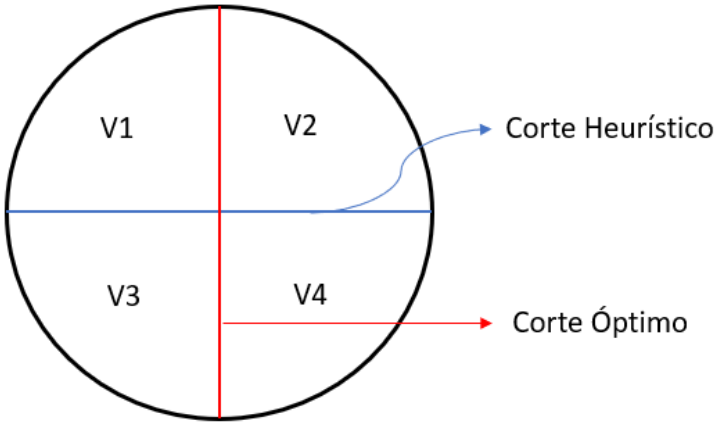
4. De igual forma vemos que $v4$ y $v5$ se mantienen en S.
5. S ha cambiado durante el recorrido por los vértices, luego repetimos el proceso.
6. En una segunda iteración por los vértices, solo se modifica $v2$, que pasa a estar en S.



Como S ha vuelto a cambiar en estas iteraciones por los vértices, volvemos a repetir de nuevo el proceso. En este caso, S ya no se modifica y el algoritmo finaliza con un valor de corte igual a 5.

Proposición 3.1. *El algoritmo descrito en el ejemplo 3.4 es 2-aproximado.*

Demostración. Suponemos un grafo G no dirigido, el corte obtenido con el algoritmo y el corte óptimo, obteniendo así el grafo dividido en cuatro zonas.



Sea e_{ij} el número de arcos entre V_i y V_j, para i, j tal que $1 \leq i \leq j \leq 4$. El valor del corte óptimo será igual a $OPT = e_{12} + e_{14} + e_{23} + e_{34}$ y el valor del corte con el algoritmo aproximado será $APROX = e_{13} + e_{14} + e_{23} + e_{24}$. Buscamos una relación entre ambas cantidades, concretamente que $OPT \leq 2APROX$ y habremos demostrado que el algoritmo es 2-aproximado.

Para todo $x_1 \in V_1$, $arc(x_1, V_1) + arc(x_1, V_2) \leq arc(x_1, V_3) + arc(x_1, V_4)$ ya que si no fuese así, nuestro algoritmo no hubiera acabado y hubiera cambiado de lugar a dicho nodo.

Sumamos la desigualdad anterior para todo nodo de V_1. Vamos a fijarnos en cada término. El primero será $\sum_{x_1 \in V_1} arc(x_1, V_1) = 2e_{11}$, ya que para dos nodos conectados $y, z \in V_1$, al computar $arc(y, V_1)$ estamos contando el arco que une y con z, y con $arc(z, V_1)$ lo volvemos a sumar.

El resto de sumandos serán e_{12}, e_{13} y e_{14} respectivamente, luego la desigualdad obtenida es $2e_{11} + e_{12} \leq e_{13} + e_{14}$. Por tanto, también se cumple que $e_{12} \leq e_{13} + e_{14}$.

Repitiendo lo anterior para V_2, V_3 y V_4, obtenemos

$$e_{12} \leq e_{13} + e_{14}, \quad e_{12} \leq e_{23} + e_{24}$$

$$e_{34} \leq e_{23} + e_{13}, \quad e_{34} \leq e_{14} + e_{24}$$

Sumando las cuatro desigualdades y dividiendo entre dos se obtiene

$$e_{12} + e_{34} \leq e_{14} + e_{23} + e_{13} + e_{24}$$

También es obvio que $e_{14} + e_{23} \leq e_{14} + e_{23} + e_{13} + e_{24}$.

Sumando las dos últimas desigualdades:

$$e_{12} + e_{34} + e_{14} + e_{24} \leq 2(e_{14} + e_{23} + e_{13} + e_{24})$$

Luego $OPT \leq 2APROX$, que nos dice que nuestro algoritmo heurístico obtiene más de la mitad de arcos del óptimo: la razón de aproximación es 2.

□

3.2. Clases de Aproximación

3.2.1. APX

Definición 3.7. La clase **APX** es la clase de todos los problemas de NPO que admiten un algoritmo δ -aproximado polinómico para $\delta < \infty$

Por definición, se tiene que $APX \subseteq NPO$, pero se cumple que la inclusión es estricta:

Proposición 3.2. *Si $N \neq NP$, entonces $APX \neq NPO$*

Demostración. Para probar este resultado tenemos que encontrar un problema de NPO para el que no exista ningún algoritmo aproximado polinómico para ningún δ (suponiendo que $N \neq NP$) y de esta forma existirá un problema que está en NPO pero no en APX.

Concretamente vamos a ver que **si $P \neq NP$, el problema del viajante de comercio no tiene ningún algoritmo aproximado polinómico para ningún δ.**

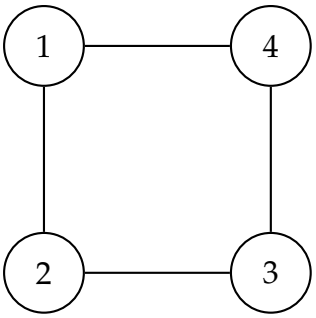
Vamos a probar que si existiese un algoritmo δ-aproximado polinómico para el viajante de comercio, entonces podríamos construir un algoritmo polinómico para el circuito hamiltoniano, que al ser un problema NP-completo, solo sería posible si $P = NP$.

Sea $G = (V, E)$ un grafo no dirigido. Entonces, construimos una instancia del problema del viajante de comercio con los mismos nodos y en el que

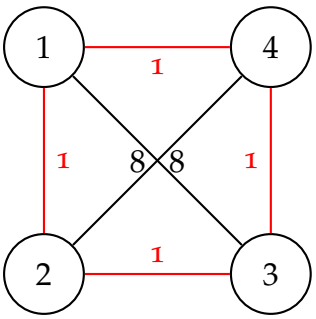
$$d(i,j) = \begin{cases} 1 & \text{si } (i,j) \in E \\ |V|\delta & \text{si } (i,j) \notin E \end{cases}$$

Es decir, si una arista está en el grafo original, le asociamos una distancia igual a 1, y si no, la distancia será igual a $|V|\delta$. De esta forma, el grafo original tiene un circuito hamiltoniano si y solo si el circuito óptimo tiene un coste $|V|$. Si el grafo original tiene un circuito hamiltoniano, elegimos como circuito para el viajante de comercio el propio circuito hamiltoniano, cuyas aristas tienen coste 1, por lo que el coste total será $|V|$. Recíprocamente, si el coste óptimo es $|V|$, tiene que ocurrir que todas las aristas del circuito tengan coste 1, y eso significa que esas aristas pertenecían al grafo original y constituyen un circuito hamiltoniano.

Por ejemplo, dado el siguiente grafo:



Construimos el siguiente problema del viajante de comercio ($|V| = 4$ y tomamos por ejemplo $\delta = 2$:



Una vez obtenido el problema del viajante de comercio asociado al grafo, aplicamos el hipotético algoritmo aproximado polinómico, obteniendo como solución un circuito. Puede ocurrir una de estas dos opciones:

- El coste del circuito es $|V|$: en este caso existe un circuito hamiltoniano en G.
- El coste del circuito es mayor que $|V|$: en este caso hay al menos una arista en el circuito que tiene coste $|V|\delta$, por lo que el coste total será mayor que $|V|\delta$. Sabemos que al ser un algoritmo δ-aproximado, se cumple que

$$\frac{m(x)}{m^*(x)} = \frac{CosteCircuito}{CosteOptimo} \leq \delta$$

o equivalentemente

$$CosteOptimo \geq \frac{CosteCircuito}{\delta}$$

Como acabamos de ver que $CosteCircuito > |V|\delta$:

$$CosteOptimo \geq \frac{CosteCircuito}{\delta} > |V|$$

Luego acabamos de ver que en este caso el grafo original no tiene un circuito hamiltoniano, ya que el circuito óptimo tiene un coste mayor que $|V|$.

Aplicando el algoritmo aproximado polinómico para el viajante de comercio hemos podido resolver el problema del circuito hamiltoniano de forma polinómica. Al ser un problema NP-completo, esto no puede ocurrir, ya que hemos supuesto que $P \neq NP$. Esta contradicción nos indica que no puede existir el supuesto algoritmo δ-aproximado polinómico para el problema del viajante de comercio, y por tanto este problema no está en la clase APX pero sí en NPO. □

3.2.2. PTAS y FPTAS

Definición 3.8. Esquema de Aproximación Polinómico. Un problema de optimización $\mathcal{P}$ tiene un **esquema de aproximación polinómico** si existe un algoritmo $\mathcal{A}$ que, para cada $\delta > 1$ y para cada instancia x de $\mathcal{P}$, devuelva una aproximación de grado δ del óptimo de x ($R(x, \mathcal{A}(x, \delta)) \leq \delta$), en tiempo polinómico en función de $|x|$ (el polinomio puede depender de δ). Si la dependencia de δ se puede expresar como un polinomio en $\frac{1}{\delta-1}$, se dice que es un **esquema de aproximación polinómico total** .

Ejemplo 3.6. Vamos a ver que podemos encontrar, para cada instancia x del Problema de la Mochila, un algoritmo δ-aproximado polinómico de complejidad $O(n^3 \frac{1}{\delta-1})$. Es decir, el Problema de la Mochila tiene un esquema de aproximación polinómico que además es de aproximación total. [Vaz10]

El Umbral de Aproximación para este problema es igual a 1, es decir, podemos conseguir un algoritmo aproximado polinómico para δ tan cerca de 1 como queramos, sin llegar a 1 (a medida que nos acercamos, aunque el algoritmo es polinómico, el coeficiente del polinomio va tendiendo a infinito).

Recordamos que en el Problema de la Mochila teníamos n objetos, donde cada objeto $i \in \{1, .., n\}$ tenía asociado un peso w_i y un valor v_i. Dada una constante W, el objetivo del problema es encontrar un subconjunto $S \in \{1, .., n\}$, con $\sum_{i \in S} w_i \leq W$, que tenga valor máximo (que $\sum_{i \in S} v_i$ sea máximo).

Vamos a ver primero un algoritmo **pseudo-polinómico** para el problema. Un algoritmo se dice que es *pseudo-polinómico* si resuelve el problema en un tiempo que depende polinómicamente del tamaño de la entrada y del entero mayor que aparezca en la entrada. Comentaremos este concepto más adelante (3.2.3).

Usando este algoritmo pseudo-polinómico vamos a construir el algoritmo δ-aproximado que hace que este problema tenga un esquema de aproximación polinómico total.

Algoritmo pseudo-polinómico para el problema de la mochila

Podemos suponer que todo objeto tiene un peso $w_i \leq W$, es decir, que todos los objetos caben en la mochila. No es ninguna restricción ya que, si un objeto no cabe en la mochila, nunca va a formar parte de la solución y podemos eliminarlo sin que afecte a ella.

Sea $V = max\{v_1, .., v_n\}$. Por tanto, el valor máximo del problema ($\sum_{i \in S} v_i$) está acotado por nV, pues en el peor de los casos introduciremos los n objetos en la mochila y cada uno tendrá que tener un valor menor o igual que V.

Para $i = 0, 1, .., n$ y $0 \leq v \leq nV$ calculamos:

$W(i, v)$: Mínimo peso que se puede conseguir eligiendo ítems entre los i primeros de valor v exactamente. En caso de que no se pueda conseguir el valor v de forma exacta, le asignamos el valor $+\infty$. Podemos calcularlos de forma recursiva en i de la siguiente forma:

$$W(i+1, v) = min\{W(i, v), W(i, v - v_{i+1}) + w_{i+1}\}$$

Supongamos que tenemos los $i+1$ objetos $a_1, a_2, .., a_i, a_{i+1}$ con sus respectivos pesos y valores $w_1, w_2, .., w_i, w_{i+1}$, $v_1, v_2, .., v_i, v_{i+1}$. Puede ocurrir dos alternativas:

- El objeto a_{i+1} no es elegido. En este caso, debemos seguir buscando entre los i objetos anteriores para conseguir un valor v: $W(i, v)$.
- El objeto a_{i+1} sí es elegido. Entonces debemos buscar entre los i objetos restantes el valor $v - v_{i+1}$. Además, como hemos elegido el objeto $i+1$, debemos sumar su peso: $W(i, v - v_{i+1}) + w_{i+1}$.

Se tiene que $W(0, v) = +\infty$, con $v \neq 0$, pues con cero ítems no podemos alcanzar ningún valor distinto de cero, y $W(i, 0) = 0$ para todo i.

Una vez hemos calculado estos valores ya podemos calcular el óptimo fácilmente. Basta con recorrer, desde $i = nV$ hasta $i = 0$ los valores $W(n, i)$. Comenzando por $W(n, nV)$, este valor será igual al mínimo peso que se puede conseguir eligiendo entre todos los objetos cuya suma total tenga valor nV. Si este valor es menor o igual que W, ya hemos encontrado el valor máximo: nV. Si no, comprobamos para $nV - 1$ y así sucesivamente.

Como tenemos que calcular estos valores para $i = 1, .., n$ y $1 \leq v \leq nV$, la complejidad del algoritmo es $O(n^2V)$. El valor de V es exponencial en función del tamaño que se necesita para almacenar V, luego este algoritmo es exponencial en función del tamaño de la entrada.

Algoritmo δ-aproximado para el problema de la mochila (esquema de aproximación polinómico total)

Dada una instancia del problema $I = (w_1, .., w_n, W, v_1, .., v_n)$, consideramos $I' = (w_1, .., w_n, W, v'_1, .., v'_n)$ donde $v'_i = 2^b[\frac{v_i}{2^b}]$ para $i = 1, .., n$ donde $[x]$ se refiere a la parte entera de x. Es decir, los b bits menos significativos se reemplazan por 0. Por ejemplo, para $b = 3$, si tenemos que para algún i, $v_i = 101011$, tendremos que $v'_i = 101000$.

Aplicamos ahora el algoritmo pseudo-polinómico, pero eliminando los b ceros de la derecha de los números y añadiéndolos al resultado (en el ejemplo nos quedaríamos solo con $v'_i = 101$ por tanto).

Como hemos quitado b cifras significativas, ahora el valor máximo V pasará a ser $\frac{V}{2^b}$ y, por tanto, la complejidad del algoritmo será $O(\frac{n^2V}{2^b})$.

Ahora vamos a demostrar que tomando $b = [log(\frac{(\delta-1)V}{n})]$ obtenemos un algoritmo δ-aproximado.

Sea S el conjunto óptimo del problema original y S' el que obtenemos en el algoritmo aproximado donde hemos eliminado b cifras significativas. Vamos denotar como $V(S) = \sum_{i \in S} v_i$ el valor conseguido con el algoritmo óptimo y $V(S') = \sum_{i \in S'} v_i$ el valor conseguido con el algoritmo aproximado (utilizando ya el valor real v_i en lugar de v'_i).

Podemos suponer que $V(S') \geq V$, ya que en el caso en el que no se cumpla, podemos considerar la solución que mete el objeto de mayor valor (es

una solución válida porque supusimos que todos los objetos cabían en la mochila).

Tenemos que:

$$V(S) = \sum_{i \in S} v_i \overset{(i)}{\geq} V(S') = \sum_{i \in S'} v_i \overset{(ii)}{\geq} \sum_{i \in S'} v'_i \overset{(iii)}{\geq} \sum_{i \in S} v'_i \overset{(iv)}{\geq}$$

$$\sum_{i \in S} (v_i - 2^b) \overset{(v)}{\geq} (\sum_{i \in S} v'_i) - n2^b = V(S) - n2^b$$

(i) Como S es el óptimo para los valores originales, cualquier otra solución tiene que ser menor o igual.

(ii) v_i y v'_i se diferencian en que v'_i tiene los b bits menos significativos a cero, por lo que $v_i \geq v'_i$

(iii) Para el problema con los valores v'_i el óptimo es S', por lo que cualquier otra solución será menor o igual.

(iv) Como hacemos cero las b últimas cifras significativas, al restar 2^b siempre será menor que v'_i. En el ejemplo anterior, donde $b = 3$ y $v_i = 101011$, tenemos que $v'_i = 101000$ y $v_i - 2^b = 101011 - 1000 = 100011$.

(v) Podemos dividir la sumatoria en dos términos y usar que en S hay como mucho n objetos.

Por tanto, hemos probado que

$$V(S) \geq V(S') \geq V(S) - n2^b$$

que podemos escribir como

$$V(S) \leq V(S') + n2^b$$

Para que sea un algoritmo aproximado nos queda probar que, para $b = [log(\frac{(\delta-1)V}{n})]$, el cociente entre la solución óptima y la aproximada debe ser menor o igual que δ, es decir, $\frac{V(S)}{V(S')} \leq \delta$

Para ello, usamos la desigualdad que acabamos de obtener y sustituimos el valor de b:

$$\frac{V(S)}{V(S')} \leq \frac{V(S') + n2^b}{V(S')} \overset{(*)}{\leq} \frac{V(S') + n\frac{(\delta-1)V}{n}}{V(S')} = \frac{V(S') + (\delta - 1)V}{V(S')} \leq$$

$$\overset{(**)}{\leq} \frac{V(S') + (\delta - 1)V(S')}{V(S')} = \delta$$

En (*) usamos que $2^{[log(x)]} \leq 2^{log(x)}$ y en (**) lo que hemos comentado antes de que $V \leq V(S')$.

La complejidad del algoritmo era $O(\frac{n^2 V}{2^b})$, que sustituyendo el valor de b nos queda $O(\frac{n^3}{\delta - 1})$. Para cada valor de δ fijo mayor que uno obtenemos un algoritmo δ-aproximado polinómico, que además depende de δ como un polinomio en $\frac{1}{\delta - 1}$. Por tanto, el problema de la mochila tiene un esquema de aproximación polinómico total.

Los problemas que tengan un esquema de aproximación polinómico total, como el Problema de la Mochila, son los que mejor se pueden aproximar, dentro de los problemas de optimización no polinómicos.

Definición 3.9. La clase **PTAS** es la clase de problemas con un esquema de aproximación polinómico.

Por definición, se tiene que PTAS $\subseteq$ APX, y de nuevo la inclusión es estricta, ya que existen problemas cuyo umbral de aproximación es mayor que 1, como el problema del cubrimiento mínimo por vértices. Como hemos comentado antes, hay un resultado que nos dice que, si $P \neq NP$, no puede existir un algoritmo aproximado para este problema para $\delta = 1.1659$ [Kan].

Definición 3.10. La clase **FPTAS** es la clase de problemas con un esquema de aproximación polinómico total.

Ya hemos visto que un problema de la clase FPTAS es el problema de la mochila. Sin embargo, existen muchos problemas que no pertenecen a esta clase. Vamos a ver un ejemplo de algunos problemas que no admiten un esquema de aproximación polinómico total.

Para ello vamos a definir un concepto antes:

Definición 3.11. Un problema de optimización está **acotado polinómicamente** si y solo si existe un polinomio p tal que para toda instancia x y para toda solución factible y de x ($y \in SOL(x)$), entonces

$$m(x, y) \leq p(|x|)$$

donde $m(x, y)$ era el coste que tiene la solución factible y para la instancia x del problema.

Proposición 3.3. *No existe ningún problema NP-difícil acotado polinómicamente con un esquema de aproximación polinómico total.*

Demostración. Sea $\mathcal{P}$ un problema NP-difícil de maximización acotado polinómicamente (el caso de minimización es análogo). Suponemos que tenemos un esquema de aproximación polinómico total para $\mathcal{P}$ tal que, para cada instancia x y para cada $\delta > 1$, su tiempo de ejecución está acotado por $q(|x|, \frac{1}{(r-1)})$ para un cierto polinomio q.

Como $\mathcal{P}$ es acotado polinómicamente, para cada instancia x y para toda solución óptima y^* de x se tiene que $CosteOptimo = m^*(x) = m(x, y^*) \leq p(|x|)$ para un cierto polinomio p.

Tomando $\delta = \frac{p(|x|)+1}{p(|x|)}$, por ser un algoritmo δ-aproximado se cumple que

$$\frac{CosteOptimo}{CosteAlgoritmo} \leq \delta = \frac{p(|x|)+1}{p(|x|)}$$

que es equivalente a

$$CosteAlgoritmo \geq CosteOptimo \frac{p(|x|)}{p(|x|)+1} = CosteOptimo(1 - \frac{1}{p(|x|)+1}) =$$

$$= CosteOptimo - \frac{CosteOptimo}{p(|x|)+1}) \overset{(i)}{>} CosteOptimo - \frac{p(|x|)}{p(|x|)+1} >$$

$$> CosteOptimo - 1$$

En (i) usamos la desigualdad anterior obtenida por ser acotado polinómicamente.

Como la función de coste devuelve siempre valores enteros por definición, no queda más remedio que se de que $CosteAlgoritmo = CosteOptimo$. Como el algoritmo estaba acotado en tiempo por $q(|x|, \frac{1}{(r-1)})$, podríamos, por tanto, conseguir la solución óptima de $\mathcal{P}$ en tiempo polinómico, que es una contradicción si $P \neq NP$, ya que $\mathcal{P}$ está en la clase NP-difícil. □

Corolario 3.1. *Si $P \neq NP$ entonces $FPTAS \subset PTAS$.*

El problema del máximo conjunto independiente para grafos planares está en PTAS (sección 3.2.1 de [APMS+99]). Este problema está acotado polinómicamente. La función de coste mide el número de vértices que componen el conjunto independiente, que es menor o igual que el número total de vértices, que a su vez es menor o igual que $|x|$ (basta tomar $p(|x|) = |x|$), luego aplicando este último resultado, no pertenece a FPTAS.

Una vez estudiadas las distintas clases de complejidad, podemos representarlas gráficamente de la siguiente forma.

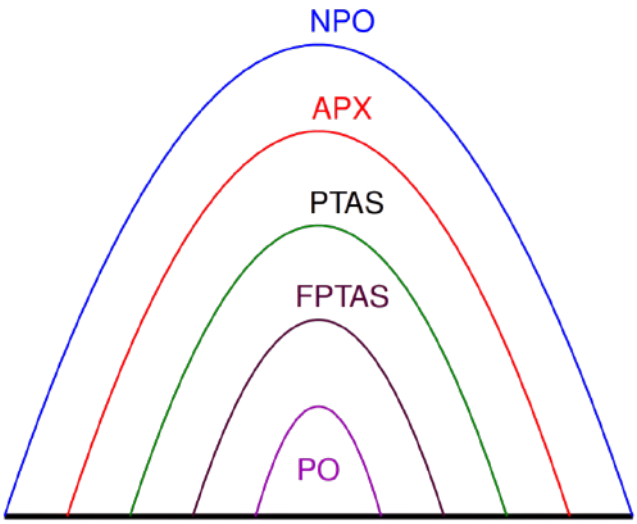


Figura 3.1: Diagrama de clases de aproximación para el caso en el que $PO \neq NPO$

3.2.3. Problemas Pseudo-polinómicos

Hemos visto que el problema de la mochila puede resolverse por programación dinámica en tiempo $O(n^2 p_{max})$ donde p_{max} es el entero de tamaño máximo que aparece en el problema. Eso no implica que el problema sea polinómico, pero su complejidad se debe al tamaño de los números que aparecen en el mismo. Es decir, si en estos casos fijamos el valor de p_{max} como parámetro, entonces el algoritmo sería polinómico.

Definición 3.12. Se dice que un problema es **pseudo-polinómico** si tiene un algoritmo que, para cada instancia del problema x, encuentra la respuesta

correcta en un tiempo que depende polinómicamente de $|x|$ y del entero mayor que aparezca en la especificación de x.

Teorema 3.1. *Si $\mathcal{P}$ es un problema que está en FPTAS , y si existe un polinomio p tal que para cada instancia x, el coste óptimo del problema $m^*(x)$ verifica:*

$$m^*(x) \leq p(|x|, max(x))$$

donde $max(x)$ es el entero mayor que aparece en x, entonces el problema es pseudo-polinómico.

Demostración. Como $\mathcal{P}$ está en FPTAS, tiene un esquema de aproximación polinómico total $\mathcal{A}$ (para cada par (x, δ) $\mathcal{A}(x, \delta)$ es una solución δ-aproximada que es polinómica en $|x|$ y $\frac{1}{\delta - 1}$) . A partir de $\mathcal{A}$ vamos a construir un algoritmo $\mathcal{A}'$ que resuelva cualquier instancia x de $\mathcal{P}$ en un tiempo que depende polinómicamente de $|x|$ y de $max(x)$.

Sea $M = p(|x|, max(x))$. Definimos $\mathcal{A}'$ como:

$$\mathcal{A}'(x) = \mathcal{A}(x, \frac{M+2}{M+1})$$

Suponiendo que $\mathcal{P}$ es un problema de minimización, se debe cumplir que

$$R(x, \mathcal{A}'(x)) = \frac{m(x)}{m^*(x)} \leq \delta = \frac{M+2}{M+1}$$

donde $m(x)$ es el coste de la solución del algoritmo $\mathcal{A}'$.

Por tanto,

$$m(x) \leq \frac{M+2}{M+1} m^*(x)$$

Por otro lado, tenemos que

$$\frac{M+2}{M+1} m^*(x) - m^*(x) = \frac{1}{M+1} m^*(x) \leq \frac{M}{M+1} < 1$$

$$\Longrightarrow \frac{M+2}{M+1} m^*(x) < m^*(x) + 1$$

Retomando la primera desigualdad

$$m(x) \leq \frac{M+2}{M+1} m^*(x) \leq m^*(x) + 1 \Longrightarrow m(x) - m^*(x) < 1$$

Como el valor del coste/beneficio en ambos casos son valores enteros, solo puede ocurrir que $m(x) = m^*(x)$, por lo que el nuevo algoritmo $\mathcal{A}'$ obtiene el óptimo.

Además, como $\mathcal{A}$ se ejecuta en tiempo $q(|x|, \frac{1}{\delta - 1})$ para un cierto polinomio q, para $\mathcal{A}'$ sustituimos el valor de δ y tenemos que se ejecuta en tiempo $q(|x|, p(|x|, max(x)) + 1)$, por lo que hemos encontrado un algoritmo que, para cada instancia x, obtiene la solución óptima en un tiempo que depende polinómicamente de $|x|$ y de $max(x)$.

□

4 PCP y resultados de no aproximabilidad

En este capítulo vamos a definir el concepto de PCP (probabilistically checkeable proofs), que nos va a permitir dar una nueva caracterización de la clase NP. Además, gracias a esta caracterización, vamos a demostrar resultados de no aproximabilidad, como por ejemplo, que el problema MAX 3-SAT no pertenece a la clase PTAS.

4.1. El modelo PCP

Sabemos que un problema pertenece a la clase NP si se puede reconocer su lenguaje asociado en tiempo polinómico por una MT no determinista. Es decir, una MT no determinista que acepta todas las instancias positivas en tiempo polinómico. Para ello, para cada instancia, la MT elige, de forma no determinista, una cadena de símbolos, de tamaño polinómico, y a continuación hace una comprobación determinista en tiempo polinómico sobre la instancia y la cadena.

Esto lo podemos ver de otra forma: la cadena elegida de forma no determinista la podemos considerar una candidata a **demostración**, es decir, una demostración de que la instancia asociada es un caso positivo del problema, y existe un algoritmo polinómico determinista que lo comprueba.

Ejemplo 4.1. Para el problema SAT, dada una instancia formada por un conjunto U de símbolos, y una colección C de cláusulas, una demostración π para dicha instancia sería una asignación de valores de verdad que satisfaga todas las cláusulas. Es decir, π es la demostración de que la instancia es un caso positivo del problema.

Podemos caracterizar, por tanto, la clase NP como el conjunto de problemas de decisión que verifica lo siguiente: para toda instancia x, x es un caso positivo del problema si y solo si existe una demostración de longitud polinómica que puede ser verificada en tiempo polinómico.

De manera más formal, un lenguaje $L \subseteq A^*$ pertenece a la clase NP si y solo si existe un polinomio $p(n)$ y un verificador V tal que

$$L = \{x \in A^* : \exists \pi \in A^* \text{ con } |\pi| \leq p(|x|), V(x, \pi) = SI\}$$

donde el verificador V es una MT que, dada una palabra x y una demostración π, comprueba en tiempo polinómico si π es una demostración válida para x. Es decir, responde SI si π demuestra que x es una palabra del lenguaje. Desde la perspectiva de los problemas, el verificador responde SI si π demuestra que x es una instancia positiva del problema. En caso contrario, responde NO.

Esta otra caracterización de NP no es la que anunciábamos que íbamos a encontrar, pero este tipo de razonamiento nos va a servir más adelante para definir el concepto de la verificación probabilística de demostraciones.

Antes de abordar este concepto, necesitamos introducir un modelo de computación más adecuado.

4.1.1. Máquinas de Turing Probabilísticas

Definición 4.1. Una **Máquina de Turing Probabilística** (MTP) es una máquina de Turing con una cinta adicional de solo lectura, denominada como *cinta aleatoria*, y con un estado adicional q_r. Mientras que no esté en el estado q_r, la máquina se comporta como una máquina de Turing ordinaria (ignorando la cinta aleatoria). En caso contrario, la máquina hace un paso de cálculo aleatorio considerando solo el símbolo actual de la cinta aleatoria: cambia de estado de acuerdo a alguna transición especificada y se mueve a la derecha una posición en la cinta aleatoria.

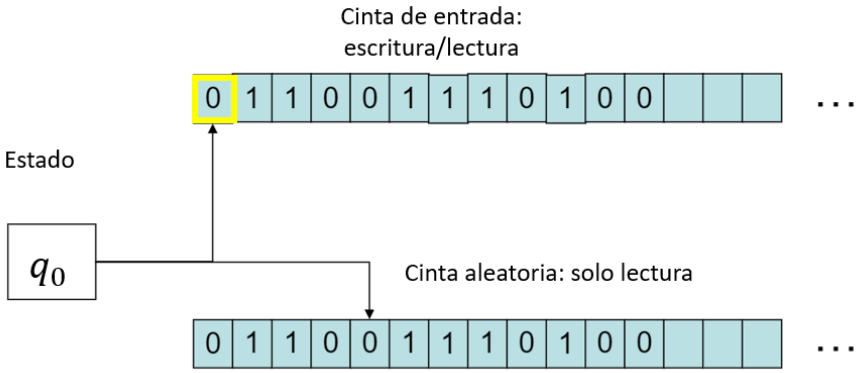


Figura 4.1: MTP en un estado distinto de q_r

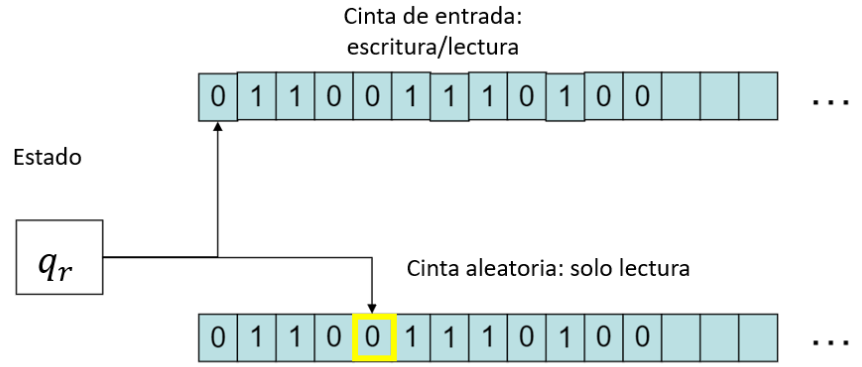


Figura 4.2: MTP en el estado q_r

La ejecución de una MTP para una cadena de entrada x depende de la cadena aleatoria contenida inicialmente en la cinta aleatoria. Es decir, podemos tener diferentes resultados tras ejecutar varias veces una misma cadena de entrada, que contrasta con lo que ocurre en una MT tradicional.

Definición 4.2. Se dice que una MTP es **r(n)-restringida** si, para cualquier entrada x de longitud n, entra en el estado q_r como mucho $r(n)$ veces.

Podemos definir por tanto la probabilidad de que una MTP $r(n)$-restringida acepte una entrada x como la probabilidad, sobre todas las cadenas aleatorias de longitud $r(n)$, que, comenzando con la entrada x, la MTP se detenga en un estado de aceptación.

Existen distintos criterios para definir la noción de lenguaje aceptado por una MTP. Por ejemplo, la clase $\mathcal{R}$P se define como el conjunto de lenguajes L para los cuales existe una MTP $\mathcal{M}$ tal que, para toda cadena x:

1. Si $x \in L$, entonces $\mathcal{M}$ acepta x con probabilidad como mínimo $1/2$.
2. Si $x \notin L$, entonces $\mathcal{M}$ rechaza x con probabilidad 1.

Ejemplo 4.2. Consideramos el siguiente problema: dado un polinomio en n variables $p(x_1, .., x_n)$ de grado d, determinar si el polinomio NO es idénticamente igual a cero. Vamos a probar que este problema está en la clase $\mathcal{R}P$.

Si p viene dado en la forma estándar simplificada es fácil conocer si es idénticamente cero comprobando si todos los coeficientes son nulos. Sin embargo, si p viene dado como una expresión aritmética arbitraria, no se conoce un algoritmo polinómico que lo resuelva.

Se puede probar que si un polinomio $p(x_1, .., x_n)$ de grado d no es idénticamente cero, el número de n-tuplas $(a_1, .., a_n)$ de enteros entre $-nd$ y nd tal que $p(a_1, .., a_n) = 0$ tiene que ser como mucho igual a $nd(2nd + 1)^{n-1}$. Además, el número de n-tuplas de enteros entre $-nd$ y nd es igual a $(2nd + 1)^n$

A partir de este resultado podemos construir una MTP polinómica $\mathcal{M}$ que verifique que, si el polinomio es un caso positivo del problema (es decir, si es distinto de cero en algún punto), $\mathcal{M}$ lo acepte con probabilidad mínima $1/2$ y si el polinomio es un caso negativo del problema (es decir, si es idénticamente cero), $\mathcal{M}$ lo rechace con probabilidad 1.

La MTP funciona de la siguiente manera: dado un polinomio de entrada p genera de forma aleatoria una n-tupla $(a_1, .., a_n)$ de enteros entre $-nd$ y nd y evalúa $p(a_1, .., a_n)$ de forma que si es cero rechaza la entrada y si es distinto de cero la acepta. Por tanto, se cumple lo siguiente:

- Si p es idénticamente cero ($p \notin L$), entonces $p(a_1, .., a_n) = 0$ para toda n-tupla y siempre se va a rechazar la entrada (rechaza con probabilidad uno).
- Si p no es idénticamente cero ($p \in L$) entonces la probabilidad de que $\mathcal{M}$ lo acepte será igual a la probabilidad de que $p(a_1, .., a_n) \neq 0$:

$$P[p(a_1, .., a_n) \neq 0] = 1 - P[p(a_1, .., a_n) = 0] = 1 - \frac{\text{casos favorables}}{\text{casos posibles}} =$$

$$= 1 - \frac{nd(2nd+1)^{n-1}}{(2nd+1)^n} = 1 - \frac{1}{2+1/nd} > 1 - 1/2 = 1/2$$

Por tanto, se verifica la condición necesaria para que el problema pertenezca a la clase $\mathcal{RP}$.

4.1.2. Verificadores y PCP

Retomamos la nueva visión que hemos dado a la clase NP a principio de la sección. Supongamos que, dado un lenguaje L y una cadena x, tenemos una demostración π que asegura que $x \in L$ (podemos suponer sin pérdida de generalidad que π es una cadena binaria). A la hora de hacer la comprobación de que π sea una demostración válida, hasta el momento la hemos hecho de forma determinista, donde era necesario leer toda la demostración completa.

Por ejemplo, en el ejemplo [4.1], una demostración para una instancia de SAT era una asignación de valores de verdad, donde todos los valores de todas las variables debían ser leídos para poder comprobar la validez de la demostración.

En el caso probabilístico, en lugar de leer la demostración completa, que nos llevaría a un cálculo determinista, se eligen de forma aleatoria un conjunto de bits en la demostración y solo se considera el contenido de esos bits. A continuación, solo a partir del subconjunto de bits elegidos de la demostración, se decide de forma determinista si la demostración es correcta o no. Por tanto, de nuevo ocurre que una misma demostración puede ser correcta y incorrecta tras varias ejecuciones, según los bits elegidos sean unos u otros.

Definición 4.3. Un **verificador** es una MTP polinómica que accede a posiciones aleatorias de una demostración π y determina la validez de la misma, donde cada acceso a π se computa como un paso de cálculo. El verificador funciona en dos pasos:

1. EL verificador utiliza la cinta aleatoria para elegir qué bits de la demostración π va a leer.
2. El verificador, de forma determinista, lee esos bits y acepta (responde SI) o rechaza (responde NO) dependiendo de los valores leídos.

Sobre la definición anterior, resaltar que el verificador tiene que decidir las direcciones de los bits que va a usar de la demostración antes de leer ningún bit de la misma. El único componente aleatorio en el verificador está en la elección del valor de la cadena aleatoria que lee de la cinta aleatoria. Lo demás es un proceso algorítmico determinista.

Dado un verificador V, una entrada x, una cadena aleatoria ρ y una demostración π, $V(x, \rho, \pi)$ denota la salida del cálculo realizado por V, que puede ser *SI* o *NO*.

Definición 4.4. Se dice que un verificador V es **(r(n), q(n))-restringido** si V es una MTP $r(n)$-restringida y para cualquier entrada x de tamaño n, V accede a $q(n)$ bits de la demostración.

Lo podemos ver de la siguiente forma. En la cinta aleatoria utilizamos una cadena ρ de $r(n)$ bits como máximo, y para cada valor de ρ, mediante un proceso algorítmico, seleccionamos $q(n)$ posiciones de π. Dependiendo del valor aleatorio de la cadena ρ se elegirán unos bits de la demostración u otros.

Como hay $2^{r(n)}$ valores posibles para ρ, como máximo consultaremos $q(n)2^{r(n)}$ bits de la demostración, y por tanto, nos basta con considerar demostraciones de longitud $q(n)2^{r(n)}$ como máximo.

Para cada par $(r(n), q(n))$ vamos a definir una nueva clase de complejidad: $PCP[r(n), q(n)]$.

Definición 4.5. Un lenguaje L pertenece a la clase $PCP[r(n), q(n)]$ si y solo si existe un verificador V $(r(n), q(n))$-restringido tal que:

1. Para cualquier $x \in L$, existe una demostración π tal que

$$P[V(x, \rho, \pi) = SI] = 1$$

2. Para cualquier $x \notin L$, y para cada demostración π

$$P[V(x, \rho, \pi) = NO] \geq 1/2$$

En ambos casos, tomamos la probabilidad sobre todas las cadenas binarias aleatorias ρ de longitud $r(|x|)$, elegidas uniformemente.

A partir de estos verificadores probabilísticos y las clases *PCP*, podemos dar una nueva caracterización de la clase NP, lo que se conoce como teorema PCP.

Antes, vamos a extender la definición de las clases PCP para permitir el uso de clases de funciones. Dadas dos clases de funciones $\mathcal{R}$ y $\mathcal{Q}$,

$$PCP[\mathcal{R}, \mathcal{Q}] = \bigcup_{r \in \mathcal{R}, q \in \mathcal{Q}} PCP[r(n), q(n)]$$

Un resultado inmediato es que

$$NP = PCP[0, \text{poly}]$$

donde poly$= \bigcup_{k \in \mathbb{N}} n^k$, ya que, como se ha comentado antes, podíamos ver la clase NP como aquellos problemas de decisión cuyas soluciones positivas podían ser verificadas en tiempo polinómico. En este caso, no utilizamos nada de la cinta aleatoria ($r(n) = 0$) y utilizamos todos los bits de la demostración para verificarla, que tendrá un tamaño polinómico.

Vamos a ver un ejemplo no trivial de cómo se puede utilizar el modelo PCP. (Ejemplo 18.3 de [ABo9] y [Aha] [Gar])

Ejemplo 4.3. Sea GNI (Graph Non-Isomorphism problem) el lenguaje formado por las parejas de grafos que no son isomorfos entre sí. Vamos a probar que $GNI \in PCP[poly(n), 1]$.

Consideramos que una entrada para GNI es una pareja $\langle G_0, G_1 \rangle$ donde G_0 y G_1 son grafos con n vértices. Tenemos que encontrar un verificador $(poly(n), 1)$-restringido que verifique las condiciones de la definición 4.5.

La idea general del funcionamiento del verificador es la siguiente. Se elige aleatoriamente uno de los dos grafos (G_b con $b = 0$ ó $b = 1$) y se hace una permutación aleatoria de sus vértices. El grafo resultante es isomorfo al grafo original seleccionado (solo se ha cambiado el orden de los vértices).

Una demostración π para este problema consiste en un array de bits indexado por todos los posibles grafos de n vértices. Para cada grafo H de n vértices, $\pi[H] \in \{0, 1\}$ es un bit que indica si H es un grafo isomorfo a G_0 o a G_1. En caso de que sea isomorfo a los dos o a ninguno, el bit tiene un valor arbitrario.

Una vez tenemos el grafo resultante H de la permutación aleatoria, se consulta el único bit permitido de la demostración ($\pi[H]$) y se acepta si y solo si $\pi[H] = b$.

Veamos que efectivamente el verificador es $(poly(n), 1)$-restringido y que verifica las dos condiciones de la definición 4.5.

En primer lugar, el verificador tiene que leer de la cinta aleatoria un primer bit (b) para elegir uno de los dos grafos y además debe leer una permutación de n elementos, por lo que utiliza un número polinómico en n de bits de la cinta. A continuación, se aplica la permutación al grafo G_b (en tiempo polinómico) obteniendo el grafo H y se consulta un solo bit de la demostración, el bit $\pi[H]$.

Pueden ocurrir dos casos:

- $G_0 \not\simeq G_1$. Si no son isomorfos entonces la palabra pertenece al lenguaje. Consideramos una demostración π tal y como hemos comentado antes, de forma que para cada grafo H isomorfo a G_0 o G_1, $\pi[H]$ indique si lo es a G_0 o a G_1. En este caso, el grafo H solo será isomorfo a uno de los dos, y por tanto, se cumplirá que $\pi[H] = b$ para cualquier permutación y valor de b, y el verificador siempre aceptará.
- $G_0 \simeq G_1$. Es decir, la palabra de entrada no pertenece al lenguaje. En este caso, el grafo H resultante de la permutación es isomorfo a ambos grafos, luego hay una probabilidad de $1/2$ de que el verificador acierte el grafo a partir del cuál se ha hecho la permutación. Por tanto, el verificador rechazará la entrada con una probabilidad de $1/2$

4.1.2.1. Nueva caracterización de NP

El teorema que nos va a permitir obtener resultados de no aproximabilidad es el siguiente.

Teorema 4.1. ***Teorema de caracterización PCP.*** $NP = PCP[O(logn), O(1)]$

La demostración es extensa y compleja, donde se utilizan técnicas sofisticadas de ECC (Error-Correcting Code) y álgebra de polinomios para campos finitos. Algunas demostraciones completas de este teorema se pueden consultar en [Dino6], [Gol] o en [APMS+99].

El teorema es sorprendente por el hecho de que para todos los problemas de NP, existe un verificador que cumple las condiciones de la definición 4.5 pero con la peculiaridad de que el verificador solo utiliza un número constante de bits de las demostraciones, que no depende de la longitud de la entrada.

4.2. Resultados de no aproximabilidad

En primer lugar, vamos a utilizar el teorema anterior para probar que el problema MAX3-SAT no está en la clase PTAS. En la demostración vamos usar los siguientes dos lemas.

Lema 4.1. *Sea L un lenguaje NP-completo. Si para $x \in A^*$ podemos construir en tiempo polinómico una instancia (U, C) de MAX3-SAT de forma que:*

- *si $x \in L$, (U, C) es satisfactible (el óptimo es $|C|$)*
- *si $x \notin L$, una fracción ϵ de cláusulas no se satisfacen (el óptimo es, a lo más, $(1 - \epsilon)|C|$)*

entonces MAX3-SAT no admite un esquema de aproximación polinómico.

Demostración. Vamos a demostrarlo probando que, en las condiciones del lema, no puede existir un algoritmo δ-aproximado con $\delta < 1/(1 - \epsilon)$ para MAX3-SAT.

En el caso de que existiese, tendríamos un algoritmo polinómico $\mathcal{A}$ que, para cada instancia x' de MAX3-SAT, verifica:

$$\frac{m^*(x')}{m(x, \mathcal{A}(x'))} < \frac{1}{1 - \epsilon}$$

Por lo tanto, para cada x':

$$m^*(x') < \frac{m(x', \mathcal{A}(x'))}{1 - \epsilon}$$

Entonces, podríamos obtener un algoritmo polinómico para L con solo ejecutar el algoritmo aproximado $\mathcal{A}$. Dado un $x \in A^*$ construimos en tiempo polinómico una instancia $x' = (U, C)$ de MAX3-SAT según las hipótesis del lema. A continuación, ejecutamos el algoritmo aproximado con x':

- Si $m(x', \mathcal{A}(x)) > (1 - \epsilon)|C|$, entonces $m^*(x') \geq m(x', \mathcal{A}(x')) > (1 - \epsilon)|C|$, y esto solo puede ocurrir si $x \in L$, luego deducimos que la palabra pertenece al lenguaje.

- Si $m(x', \mathcal{A}(x')) \leq (1-\epsilon)|C|$, entonces

$$m^*(x') < \frac{m(x', \mathcal{A}(x'))}{1-\epsilon} \leq \frac{(1-\epsilon)|C|}{1-\epsilon} = |C|$$

que solo ocurre si $x \notin L$, luego en este caso deducimos que la palabra no pertenece al lenguaje.

Hemos encontrado un algoritmo exacto polinómico para un problema NP-completo, por lo que llegamos a una contradicción. □

Lema 4.2. *Dado un conjunto U de q variables, con $q > 2$, y un conjunto C' de cláusulas de exactamente q literales por cláusula. Entonces, se puede construir, en tiempo polinómico, un conjunto de cláusulas C con exactamente 3 literales por cláusula tal que C es satisfactible si y solo si lo es C'. Además, $|C| = (q-2)|C'|$. [GJ90b]*

Teorema 4.2. *Si $P \neq NP$, el problema MAX3-SAT no admite un esquema de aproximación polinómico.*

Demostración. La demostración consiste en construir una instancia (U, C) de MAX3-SAT que verifique las condiciones del lema 4.1.

Sea L un lenguaje NP-completo. El teorema de caracterización PCP nos asegura que existe un verificador (r(n),q)-restringido para L, donde r(n) es $O(logn)$ y q es una constante, que suponemos mayor que 2.

Sin pérdida de generalidad, podemos también asumir que el verificador siempre selecciona exactamente q bits de la demostración (aunque no necesite utilizarlos todos).

Construimos una instancia (U, C) de la siguiente forma. Sea ρ un posible valor para la cadena aleatoria. Para cada ρ, el verificador selecciona q bits de la demostración. El bit número i de los q seleccionados se corresponde con el bit número $\rho[i]$ de la demostración. Es decir, para cada ρ se seleccionan los bits $\rho[1], ..., \rho[q]$ de la demostración.

Para cada posible ρ, añadimos a U las variables $v_{\rho[1]}, ..., v_{\rho[q]}$, que se corresponden con los q bits que el verificador lee de la demostración.

Es decir, para cada bit número i de los q bits que selecciona el verificador de la demostración, se añade una variable $v_{\rho[i]}$, que contendrá el valor del bit (0 o 1) que leemos de la demostración en la posición $\rho[i]$.

Como hay $2^{r(n)}$ posibles valores de ρ, y para cada uno de ellos añadimos q variables, tenemos $q2^{r(n)}$ variables en total. Como $r(n)$ es $O(logn)$, se tiene que $r(n) \leq clogn$ para una cierta constante c. Por tanto, $q2^{r(n)} \leq qn^c$, luego el número de variables en U está acotado por qn^c (polinómico en n).

Para un cierto ρ, habrá algunas asignaciones de valores para $v_{\rho[1]}, ..., v_{\rho[q]}$, en las que el verificador aceptará y otras en las que no.

Por ejemplo, para $q = 3$, tendríamos tres variables $v_{\rho[1]}, v_{\rho[2]}, v_{\rho[3]}$, que para cada demostración π tendrán unos valores distintos.

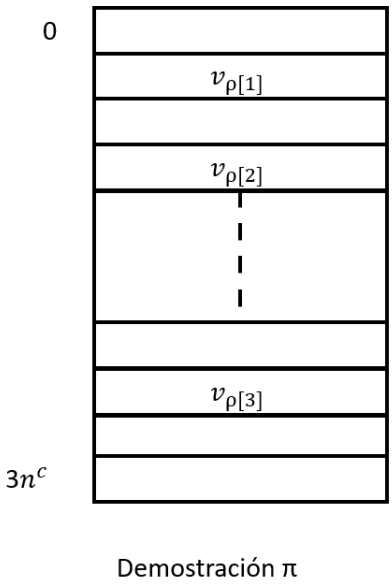


Figura 4.3: Ejemplo para $q = 3$

A modo de ejemplo, suponemos que el verificador acepta las siguientes configuraciones de valores de las variables, donde asumimos que la rama izquierda significa un 0 y la derecha un 1.

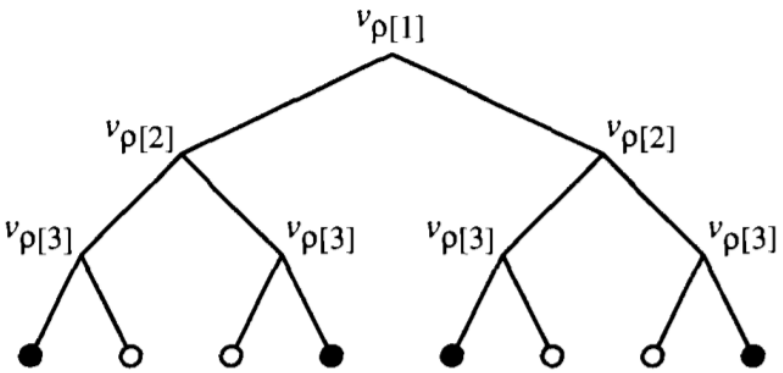


Figura 4.4: Árbol de posibles valores para las variables

Definimos el conjunto A_ρ como el conjunto de q-tuplas para las cuales el verificador rechaza. El número total de tuplas distintas con q valores es 2^q, $|A_\rho| \leq 2^q$, que es un número constante que no depende de n. Para el ejemplo anterior tenemos que $A_\rho = \{(0,0,0),(0,1,1),(1,0,0),(1,1,1)\}$.

A continuación, para cada tupla $(a_1,..,a_q) \in A_\rho$, construimos una cláusula de q literales de forma que solo sea cierta cuando los bits $v_{\rho[1]},...,v_{\rho[q]}$ no toman los valores $a_1,..,a_q$ respectivamente. Por ejemplo, en el caso anterior, añadiríamos 4 cláusulas, una por cada tupla de A_ρ:

$$(0,0,0) \quad \longrightarrow \quad v_{\rho[1]} \vee v_{\rho[2]} \vee v_{\rho[3]}$$

$$(0,1,1) \quad \longrightarrow \quad v_{\rho[1]} \vee \overline{v}_{\rho[2]} \vee \overline{v}_{\rho[3]}$$

$$(1,0,0) \quad \longrightarrow \quad \overline{v}_{\rho[1]} \vee v_{\rho[2]} \vee v_{\rho[3]}$$

$$(1,1,1) \quad \longrightarrow \quad \overline{v}_{\rho[1]} \vee \overline{v}_{\rho[2]} \vee \overline{v}_{\rho[3]}$$

Para cada ρ, en A_ρ hay como máximo 2^q q-tuplas y por cada tupla hay una cláusula. Como hay como mucho $2^{r(n)} \leq n^c$ valores distintos de ρ, obtenemos un conjunto de cláusulas C' con $2^q n^c$ cláusulas como máximo (polinómico en n).

Utilizando el lema 4.2 obtenemos un conjunto C de cláusulas de exactamente 3 literales por cláusula tal que C es satisfactible si y solo si lo es C' y con $|C| = (q-2)|C'|$.

Una vez ya tenemos construida la instancia (U, C) de MAX3-SAT, únicamente nos falta ver que efectivamente se verifican las hipótesis del lema 4.1 y ya habríamos acabado la demostración.

- Si $x \in L$, tenemos que probar que el óptimo es $|C|$ (que todas las cláusulas se satisfacen). Como $L \in PCP[O(logn), O(1)]$, existe una demostración $\pi(x)$ tal que $P[V(x, \rho, \pi(x)) = SI] = 1$. Es decir, la demostración es aceptada para cualquier valor de ρ. Por tanto, tomando una asignación de las variables $v_{\rho[1]}, ..., v_{\rho[q]}$ igual al valor de los bits asociados a la demostración $\pi(x)$ se satisfacen todas las cláusulas, ya que la q-tupla nunca va a estar en A_ρ para ningún ρ al ser siempre aceptada.

- Si $x \notin L$, para cada demostración π, $P[V(x, \rho, \pi) = NO] \geq 1/2$, luego como mínimo el verificador siempre rechazará la mitad de valores de ρ ($2^{r(n)}/2$). No podemos encontrar ninguna asignación de valores para las que se satisfagan todas las cláusulas ya que para cualquiera de ellas siempre va a haber alguna cláusula que no se satisfaga, puesto que, de hecho, debe haber más de la mitad cláusulas que no se satisfagan, que son las asociadas a los valores de ρ donde el verificador rechaza (la tupla correspondiente está en A_ρ).

  Es decir, suponemos una cierta asignación. Como x $\notin L$, el verificador rechaza para más de la mitad de los ρ. Si para un ρ se rechaza significa que se ha añadido una cláusula que no se satisface para esta asignación concreta. Por tanto, más de $2^{r(n)}/2$ cláusulas no se satisfacen para esta asignación. Esto mismo ocurre con cualquier otra asignación que tomemos, luego siempre hay más de más de $2^{r(n)}/2$ cláusulas que no se satisfacen.

  La fracción de cláusulas que no se satisfacen es como mínimo

$$\frac{2^{r(n)}/2}{(q-2)2^{q+r(n)}} = \frac{1}{(q-2)2^{q+1}} \leq 2^{-(q+1)}$$

  que es una fracción constante.

□

También se pueden obtener resultados de no aproximabilidad para el problema del *clique máximo* (CM).

En la sección siguiente vamos a estudiar un concepto de reducción entre problemas que preserva la calidad de aproximación. Utilizando este tipo de reducción, de forma sencilla se puede comprobar que MAX3-SAT es reducible a CM, de forma que cualquier solución aproximada para el segundo problema puede ser transformada en tiempo polinómico en una solución para el primero, con al menos la misma garantía de comportamiento.

Utilizando el teorema que acabamos de probar, como MAX3-SAT no está en la clase PTAS, CM no puede estar tampoco en PTAS, ya que en caso contrario, a partir de un esquema de aproximación polinómico para CM podríamos obtener uno para MAX3-SAT, llegando a una contradicción.

Además, usando el teorema de caracterización PCP se puede probar que el problema del **clique máximo no está en APX si** $P \neq NP$.

Estos dos resultados se pueden consultar con más detalle en la sección 6.4.2 de [APMS+99].

En cambio, en la sección que viene a continuación vamos a probar que MAX3-SAT sí está en APX y que de hecho es un problema completo para esta clase.

5 Completitud dentro de NPO

Buscamos hacer un estudio sobre la existencia de problemas completos para las clases de complejidad que hemos definido para los algoritmos aproximados. Si queremos hacerlo de forma análoga al que se hizo para los problemas de decisión en la clase NP, lo primero que nos debemos fijar es que el concepto de reducción Karp no nos sirve.

En el caso de NP, un problema era NP-completo si estaba en NP y si cualquier otro problema de NP se podía reducir a él mediante una reducción Karp.

Una reducción Karp entre los problemas $\mathcal{P}_1$ y $\mathcal{P}_2$ establece una correspondencia entre instancias del primer problema y el segundo, de forma que si tenemos un algoritmo que resuelva el segundo, nos permitirá resolver el primero mediante la composición con la reducción. Sin embargo, no te asegura que una solución buena del segundo se corresponda con una buena del primero.

Por ejemplo, hemos visto que en los problemas del cubrimiento mínimo por vértices (CV) y del conjunto máximo independiente (CI) ocurría que si $G = (V, E)$ era un grafo y $V^* \subseteq V$, era equivalente que V^* fuese un cubrimiento por vértices de G a que $V \setminus V^*$ fuese un conjunto independiente de G.

Supongamos que para un cierto grafo $G = (V, E)$ de 120 vértices la solución óptima para CV es 100 vértices. Por tanto, para CI será 20 vértices.

Supongamos también que tenemos un algoritmo aproximado que nos ha dado una solución aproximada para CV de 119 vértices, es decir, una solución bastante buena, puesto que $\frac{OPT}{APROX} = \frac{119}{100} = 1.19$. Sin embargo, para CI tenemos que la solución aproximada es de $200 - 119 = 1$ vértice, y por tanto, $\frac{OPT}{APROX} = \frac{20}{1} = 20$ obteniendo así una solución mucho peor.

Por tanto, debemos buscar otro concepto más fuerte de reducción que nos permita mantener la calidad de la aproximación. Si un problema $\mathcal{P}_1$ se reduce a un problema $\mathcal{P}_2$, la reducción debería garantizarnos que una solución

aproximada de $\mathcal{P}_2$ pueda ser usada para obtener una solución aproximada de $\mathcal{P}_1$ con una garantía de comportamiento similar.

5.1. Reducibilidad AP

A la reducción Karp polinómica que utilizábamos para problemas de decisión le vamos a añadir una modificación. Si un problema $\mathcal{P}_1$ se reduce a un problema $\mathcal{P}_2$, necesitamos, además de la función f que transforme instancias de $\mathcal{P}_1$ en instancias de $\mathcal{P}_2$, una función g que transforme de vuelta soluciones de $\mathcal{P}_2$ en soluciones de $\mathcal{P}_1$.

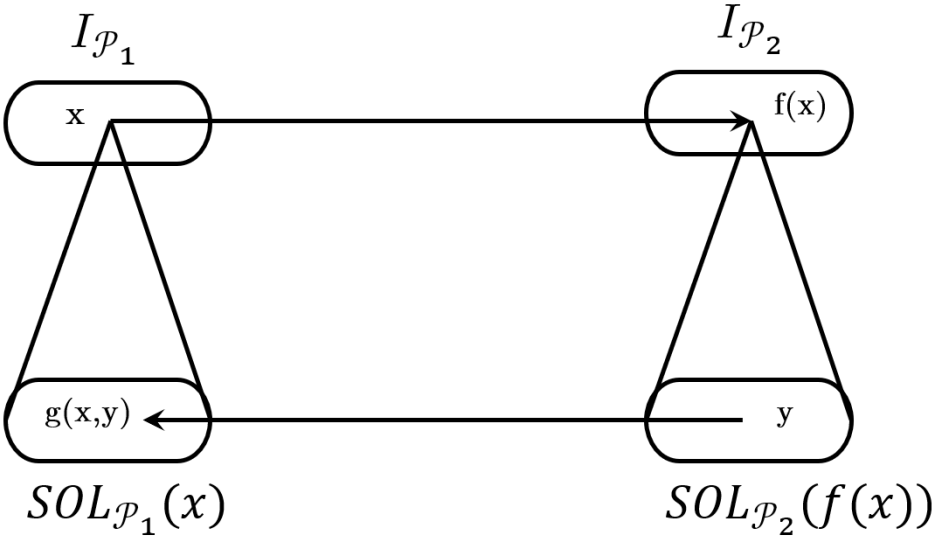


Figura 5.1: Reducción entre dos problemas de optimización

Vamos a definir el concepto que se conoce como reducibilidad AP (Approximation Preserving reduction), que va a tener la propiedad de establecer una relación lineal entre las razones de eficacia.

Definición 5.1. Sean $\mathcal{P}_1$ y $\mathcal{P}_2$ dos problemas de optimización en NPO. Se dice que $\mathcal{P}_1$ es AP-reducible a $\mathcal{P}_2$ ($\mathcal{P}_1 \leq_{AP} \mathcal{P}_2$) si existen dos funciones f y g, y una constante $\alpha \geq 1$ tal que:

1. Para cualquier instancia $x \in I_{\mathcal{P}_1}$ y para cualquier racional $\delta > 1$, $f(x, \delta) \in I_{\mathcal{P}_2}$.

2. Para cualquier instancia $x \in I_{\mathcal{P}_1}$ y para cualquier racional $\delta > 1$, si $SOL_{\mathcal{P}_1}(x) \neq \emptyset$ entonces $SOL_{\mathcal{P}_2}(f(x,\delta)) \neq \emptyset$.

3. Para cualquier instancia $x \in I_{\mathcal{P}_1}$, para cualquier racional $\delta > 1$, y para cualquier $y \in SOL_{\mathcal{P}_2}(f(x,\delta))$, $g(x,y,r) \in SOL_{\mathcal{P}_1}(x)$

4. f y g son calculables por dos algoritmos $\mathcal{A}_f$ y $\mathcal{A}_g$ respectivamente, que son polinómicos en tiempo para cada racional δ fijo.

5. Para cualquier instancia $x \in I_{\mathcal{P}_1}$, para cualquier racional $\delta > 1$ y para cualquier $y \in SOL_{\mathcal{P}_2}(f(x,\delta))$,

$$\text{Si } R_{\mathcal{P}_2}(f(x,\delta),y) \leq \delta \Longrightarrow R_{\mathcal{P}_1}(x,g(x,y,\delta)) \leq 1 + \alpha(\delta - 1)$$

   Esta condición se conoce como *AP-condition*, ya que es la que hace que las buenas soluciones encontradas para $\mathcal{P}_2$ lo sean también para $\mathcal{P}_1$ cuando aplicamos g. Para $\alpha = 1$ tenemos de hecho que

$$\text{Si } R_{\mathcal{P}_2}(f(x,\delta),y) \leq \delta \Longrightarrow R_{\mathcal{P}_1}(x,g(x,y,\delta)) \leq \delta$$

La tripleta (f,g,α) es lo que se conoce como *reducción-AP* de $\mathcal{P}_1$ a $\mathcal{P}_2$.

Cómo usar una reducción-AP

En la siguiente figura podemos ver la utilidad de este tipo de reducción y cómo se debe usar para sacarle partido. Supongamos que tenemos una reducción-AP de un problema $\mathcal{P}_1$ a $\mathcal{P}_2$, ambos problemas en NPO. Supongamos también que tenemos un algoritmo aproximado que nos resuelve $\mathcal{P}_2$. Entonces, para cada instancia x de $\mathcal{P}_1$ podemos conseguir una solución aproximada y de x de la siguiente forma:

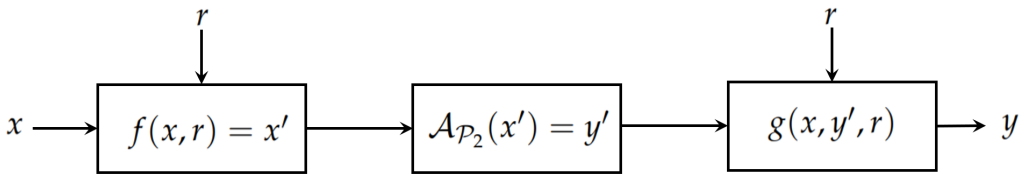


Figura 5.2: Cómo usar una reducción-AP

Dado un α y un δ, consiste en lo siguiente. Primero aplicar f a x para obtener una instancia x' de $\mathcal{P}_2$. A continuación, obtener una solución aproximada y' de x' usando el algoritmo, y a partir de la solución aproximada obtenida, conseguir una solución aproximada y para x aplicando g a la solución y'.

Proposición 5.1. *Si $\mathcal{P}_1 \leq_{AP} \mathcal{P}_2$ y $\mathcal{P}_2 \in$ APX (respectivamente, PTAS), entonces $\mathcal{P}_1 \in$ APX (respectivamente, $\mathcal{P}_1 \in$ PTAS).*

Demostración. Sea (f, g, α) una reducción-AP de $\mathcal{P}_1$ a $\mathcal{P}_2$. Si $\mathcal{P}_2 \in$ APX y $\mathcal{A}_{\mathcal{P}_2}$ es un algoritmo δ-aproximado polinómico para $\mathcal{P}_2$, entonces podemos construir un algoritmo $\mathcal{A}_{\mathcal{P}_1}$ tal y como indica la figura 5.2:

$$\mathcal{A}_{\mathcal{P}_1}(x) = g(x, \mathcal{A}_{\mathcal{P}_2}(f(x, \delta)), \delta)$$

que, por la definición de reducción-AP es un algoritmo δ'-aproximado polinómico para $\mathcal{P}_1$ con $\delta' = 1 + \alpha(\delta - 1)$, por lo que $\mathcal{P}_1 \in$ APX.

Si $\mathcal{P}_2 \in$ PTAS, entonces tenemos que $\mathcal{A}_{\mathcal{P}_2}$ es un esquema de aproximación polinómico, es decir, para cada $\delta > 1$, $\mathcal{A}_{\mathcal{P}_2}$ es un algoritmo δ-aproximado polinómico , y por el mismo argumento que antes, $\mathcal{A}_{\mathcal{P}_1}$ también es un esquema de aproximación polinómico, luego $\mathcal{P}_1 \in$ PTAS. □

5.2. Completitud

Se puede ver fácilmente que la reducibilidad-AP es **transitiva**, es decir, si $\mathcal{P}_1 \leq_{AP} \mathcal{P}_2$ y $\mathcal{P}_2 \leq_{AP} \mathcal{P}_3$, entonces $\mathcal{P}_1 \leq_{AP} \mathcal{P}_3$. Esto hace que esta reducibilidad induzca un orden parcial entre problemas de la misma clase de aproximación. Al igual que ocurría en NP con la reducción Karp, surge la noción de problema completo, como el problema máximo respecto al orden inducido por la reducción.

Definición 5.2. Dada una clase C de NPO, un problema es **C-difícil** (respecto a la reducibilidad-AP) si para cualquier $\mathcal{P}' \in C$, $\mathcal{P}' \leq_{AP} \mathcal{P}$. Un problema C-difícil es **C-completo** (respecto a la reducibilidad-AP) si pertenece a C.

5.2.1. APX-completitud

En este apartado vamos a ver cómo el Teorema de Caracterización PCP no solo se puede utilizar para obtener resultados de no aproximabilidad, sino que también puede ser utilizado para probar resultados de APX-completitud.

Hemos visto en el teorema 4.2 que el problema MAX3-SAT no está en PTAS. En este apartado vamos a ver que este problema es APX-completo, es decir, que está en la clase APX y que además, cualquier problema de esta clase se puede reducir a él mediante una reduccion-AP.

Veamos primero que MAX3-SAT está en APX. Este problema es un versión particular del problema MAX-SAT, donde cada cláusula contiene como mucho tres literales. Vamos a ver que existe un algoritmo 2-aproximado para MAX-SAT, y por tanto, también será un algoritmo 2-aproximado para MAX3-SAT, como caso particular.

Proposición 5.2. *Algorithm 1 es un algoritmo polinómico* 2*-aproximado para MAX-SAT*

Demostración. Dada una instancia de MAX-SAT con c cláusulas, vamos a probar, usando una inducción sobre el número de variables, que la asignación de valores de verdad que obtiene el algoritmo satisface como mínimo $c/2$ cláusulas. Como el óptimo es, como máximo, c, ya tendríamos probado el resultado.

Para el caso de una única variable, es trivial. Supongamos que es cierto para el caso en el que hay $n-1$ variables ($n \geq 1$) y consideremos el caso en el que tenemos n variables. Sea v la variable que se corresponde con el literal que aparece en el mayor número de cláusulas. Sea c_1 el número de cláusulas en las que v aparece positiva y c_2 el número de cláusulas en las que aparece negada. Sin pérdida de generalidad, suponemos que $c_1 \geq c_2$, por lo que el algoritmo asigna el valor $TRUE$ a v. Después de esta asignación, nos quedan como mínimo $c - c_1 - c_2$ cláusulas en $n-1$ variables. Por la hipótesis de inducción, aplicando el algoritmo sobre estas cláusulas y variables se obtiene una asignación de valores de verdad que satisface como mínimo a $(c - c_1 - c_2)/2$ cláusulas. Por tanto, el número total de cláusulas que se satisfacen es $c_1 + (c - c_1 - c_2)/2 = c/2 + (c_1 - c_2)/2 \geq c/2$

□

Algorithm 1: Greedy MAX-SAT 2-aproximado

Data: Conjunto C de cláusulas en un conjunto de variables V
Result: Asignación de verdad $f : V \mapsto \{TRUE, FALSE\}$
begin
  forall $v \in V$ **do**
    $f(v) := TRUE$
  repeat
    Sea l el literal que aparece en el mayor número de cláusulas (si hay empate elegimos de forma arbitraria);
    Sea C_l el conjunto de cláusulas en las que aparece l;
    Sea $C_{\bar{l}}$ el conjunto de cláusulas en las que aparece $\bar{l}$;
    Sea v_l la variable asociada al literal l;
    if *l es un literal positivo* **then**
      begin
        $C := C - C_l$; (si $f(v_l) = TRUE$, todas las cláusulas de C_l se satisfacen)
        Eliminar $\bar{l}$ de todas las cláusulas de $C_{\bar{l}}$; ($\bar{l}$ nunca es cierto y podemos eliminarlo)
        Eliminar todas las cláusulas vacías en C; (por si al eliminar $\bar{l}$ de las cláusulas alguna se queda vacía)
    else
      begin
        (análogo al caso anterior cuando el literal l es una variable negada)
        $f(v_l) := FALSE$;
        $C := C - C_l$;
        Eliminar $\bar{l}$ de todas las cláusulas de $C_{\bar{l}}$;
        Eliminar todas las cláusulas vacías en C;
  until $C = \emptyset$;
  return f

Corolario 5.1. *MAX3-SAT está en APX.*

Para demostrar que un problema $\mathcal{P}$ es APX-completo hace falta construir una reducción-AP entre un problema genérico $\mathcal{P}'$ de APX y $\mathcal{P}$. Sin embargo, vamos a ver que basta con que $\mathcal{P}'$ sea un problema cualquiera de maximización.

Teorema 5.1. *Para cualquier problema de minimización $\mathcal{P}$ en APX, existe un problema de maximización $\mathcal{P}'$ en APX tal que $\mathcal{P} \leq_{AP} \mathcal{P}'$.*

Demostración. Sea $\mathcal{A}$ un algoritmo δ-aproximado para $\mathcal{P}$. Para toda instancia x de $\mathcal{P}$, sea $t(x) = m_{\mathcal{P}}(x, \mathcal{A}(x))$. Esto significa que $t(x) \leq \delta m^*_{\mathcal{P}}(x)$. Vamos a construir el problema $\mathcal{P}'$ de manera que sea idéntico a $\mathcal{P}$ salvo en la función de medida, que la definimos como:

$$m_{\mathcal{P}'}(x, y) = \begin{cases} (k+1)t(x) - k m_{\mathcal{P}}(x, y) & \text{si } m_{\mathcal{P}}(x, y) \leq t(x) \\ t(x) & \text{en otro caso} \end{cases} \tag{5.1}$$

donde $k = \lceil \delta \rceil$.

Como $\mathcal{P}'$ es un problema de maximización, para toda instancia x, y solución factible y, $m^*_{\mathcal{P}'}(x) \geq m_{\mathcal{P}'}(x, y)$ luego, tomando $y = \mathcal{A}(x)$ ($m_{\mathcal{P}}(x, \mathcal{A}(x)) = t(x)$)

$$m^*_{\mathcal{P}'}(x) \geq m_{\mathcal{P}'}(x, \mathcal{A}(x)) = (k+1)t(x) - kt(x) = t(x)$$

Por otro lado, tomando $y = y'^*(x)$ (el óptimo para $\mathcal{P}'$):

$$m^*_{\mathcal{P}'}(x) = m_{\mathcal{P}'}(x, y'^*(x)) = \begin{cases} (k+1)t(x) - k m_{\mathcal{P}}(x, y'^*(x)) & \text{si } m_{\mathcal{P}}(x, y'^*(x)) \leq t(x) \\ t(x) & \text{en otro caso} \end{cases} \leq$$

$$\leq \begin{cases} (k+1)t(x) & \text{si } m_{\mathcal{P}}(x, y'^*(x)) \leq t(x) \\ t(x) & \text{en otro caso} \end{cases} \leq$$

$$\leq (k+1)t(x)$$

En definitiva, tenemos que $t(x) \leq m^*_{\mathcal{P}'}(x) \leq (k+1)t(x)$, que demuestra que $\mathcal{A}$ es un algoritmo $(k+1)$-aproximado para $\mathcal{P}'$, es decir, $\mathcal{P}' \in APX$. Nos falta construir la reducción-AP entre $\mathcal{P}$ y $\mathcal{P}'$. La definimos de la siguiente forma (f y g no dependen de δ):

1. Para cada instancia x, $f(x) = x$.
2. Para cada instancia x y para cada solución y de $f((x)$,

$$g(x,y) = \begin{cases} y & \text{si } m_{\mathcal{P}}(x,y) \leq t(x) \\ \mathcal{A}(x) & \text{en otro caso} \end{cases}$$

3. $\alpha = k + 1$.

Tenemos que demostrar que se verifica la *AP-condition*. Si y es una solución tal que $R_{\mathcal{P}'}(x,y) = m^*_{\mathcal{P}'}(x)/m_{\mathcal{P}'}(x,y) \leq \delta'$, debemos probar que $R_{\mathcal{P}}(x, g(x,y)) \leq 1 + \alpha(\delta' - 1)$. Para ello, distinguimos dos casos:

1. $m_{\mathcal{P}}(x,y) \leq t(x)$. En este caso, $g(x,y) = y$. Utilizando las siguientes desigualdades que hemos obtenido:

$$m^*_{\mathcal{P}'}(x)/\delta' \leq m_{\mathcal{P}}(x,y) \tag{5.2}$$

$$t(x) \leq m^*_{\mathcal{P}'}(x) \leq (k+1)t(x) \tag{5.3}$$

$$k \geq \delta \tag{5.4}$$

$$t(x) \leq \delta m^*_{\mathcal{P}}(x) \tag{5.5}$$

tenemos que:

$$\begin{aligned} m_{\mathcal{P}}(x,y) &\overset{(i)}{=} \frac{(k+1)t(x) - m_{\mathcal{P}'}(x,y)}{k} \\ &\overset{(ii)}{\leq} \frac{(k+1)t(x) - m^*_{\mathcal{P}'}(x)/\delta'}{k} = \frac{(k+1)t(x) - m^*_{\mathcal{P}'}(x)/(1+(\delta'-1))}{k} \\ &\overset{(iii)}{\leq} \frac{(k+1)t(x) - (1-(\delta'-1))m^*_{\mathcal{P}'}(x)}{k} \\ &= \frac{(k+1)t(x) - m^*_{\mathcal{P}'}(x) + (\delta'-1)m^*_{\mathcal{P}'}(x)}{k} \\ &\overset{(iv)}{\leq} m^*_{\mathcal{P}}(x) + \frac{\delta'-1}{k} m^*_{\mathcal{P}'}(x) \\ &\overset{(v)}{\leq} m^*_{\mathcal{P}}(x) + \frac{\delta'-1}{k}(k+1)t(x) \\ &\overset{(vi)}{\leq} m^*_{\mathcal{P}}(x) + (\delta'-1)(k+1)\frac{t(x)}{\delta} \\ &\overset{(vii)}{\leq} (1+\alpha(\delta'-1))m^*_{\mathcal{P}}(x) \end{aligned}$$

En cada paso se ha usado lo siguiente:

(i) Despejamos de 5.1 (estamos en el primer caso).

(ii) Utilizando 5.3.

(iii) Utilizamos que

$$\frac{1}{1+(\delta'-1)} \geq 1-(\delta'-1)$$

Sea $s = \delta' - 1 \geq 0$. Entonces, tenemos que ver que

$$\frac{1}{1+s} \geq 1-s$$

Podemos escribir

$$1-s = \frac{1+s}{1+s} - s = \frac{1}{1+s} + \frac{s}{1+s} - s \leq \frac{1}{1+s}$$

ya que

$$\frac{s}{1+s} - s = \frac{s-s(1+s)}{1+s} = \frac{-s^2}{1+s} \leq 0$$

(iv) Dividiendo la fracción en dos sumandos, el segundo sumando se queda igual, y para el primero utilizamos 5.3:

$$\frac{(k+1)t(x) - m^*_{\mathcal{P}'}(x)}{k} \leq \frac{(k+1)m^*_{\mathcal{P}'}(x) - m^*_{\mathcal{P}'}(x)}{k} = m^*_{\mathcal{P}'}(x)$$

(v) Usando de forma directa 5.3

(vi) Por 5.4

(vii) Sustituyendo $\alpha = k+1$ y usando 5.5

En definitiva, obtenemos que $R_{\mathcal{P}}(x, g(x,y)) = R_{\mathcal{P}}(x,y) \leq 1 + \alpha(\delta'+1)$

2. $m_{\mathcal{P}}(x,y) > t(x)$. En este caso, de 5.1 obtenemos que $m^*_{\mathcal{P}'}(x) = (k+1)t(x) - km^*_{\mathcal{P}}(x)$. Ahora, distinguimos dos posibilidades:

Si $\delta' \geq \delta$ es inmediato, por ser δ-aproximado, y que $\alpha \geq 1$, que

$$R(x, g(x,y)) = R(x, \mathcal{A}(x)) = t(x)/m^*_{\mathcal{P}}(x) \leq \delta \leq \delta' \leq 1 + \alpha(\delta'-1)$$

Si $\delta' < \delta$, entonces $\delta' < k$ y $(k+1) - \delta' > 1$.

Si $m^*_{P'}(x)/t(x) \leq \delta'$, teniendo en cuenta la primera expresión para el óptimo de $\mathcal{P}'$ se tiene que

$$\frac{(k+1)t(x) - km^*_{\mathcal{P}}(x)}{t(x)} \leq \delta'$$

y se llega entonces a que

$$t(x) \leq \frac{km^*_P(x)}{k+1-\delta'}$$

Por lo tanto

$$R(x, \mathcal{A}(x)) = t(x)/m^*_P(x) \leq \frac{k}{k+1-\delta'} =$$
$$1 + \frac{\delta' - 1}{k+1-\delta'} < 1 + \delta' - 1 = \delta'$$

lo último, teniendo en cuenta que $(k+1) - \delta' > 1$.

□

Utilizando este teorema, podemos demostrar que MAX3-SAT es APX-completo probando que es completo para los problemas de maximización de APX. Para ello, vamos a utilizar el resultado que obtuvimos en la demostración del teorema 4.2, donde se usaba el Teorema de Caracterización PCP.

En dicha demostración, dado un lenguaje L NP-completo, para cada $x \in A^*$ construíamos una instancia (U, C) de MAX3-SAT en las condiciones del lema 4.1.

Particularmente, tomando $L = SAT$, el teorema nos asegura que existe una constante ϵ y dos funciones f_s y g_s tal que, para cualquier fórmula Booleana ϕ en forma normal conjuntiva (CNF), $\psi = f_s(\phi)$ es una formula Booleana en forma normal conjuntiva, con a lo más tres literales por cláusula, que verifica la siguiente afirmación: para cualquier asignación de valores de verdad τ que satisfaga una fracción de al menos de $1 - \epsilon$ cláusulas de ψ, $g_s(\phi, \tau)$ satisface ϕ si y solo si ϕ es satisfactible.

Teorema 5.2. *MAX3-SAT es completo para la clase de problemas de maximización en APX.*

Demostración. Sea $\mathcal{P}$ un problema de maximización en APX y sea $\mathcal{A}_{\mathcal{P}}$ un algoritmo $\delta_{\mathcal{P}}$-aproximado para $\mathcal{P}$. Nuestro objetivo es definir una reducción-AP (f, g, α) de $\mathcal{P}$ a MAX3-SAT. En primer lugar, definimos

$$\alpha = 2(\delta_{\mathcal{P}}\ log\delta_{\mathcal{P}} + \delta_{\mathcal{P}} - 1)\frac{1+\epsilon}{\epsilon}$$

donde ϵ es la constante que hemos comentado antes que obtenemos con el Teorema 4.2.

Para definir las funciones f y g, para cualquier δ dado, distinguimos dos casos.

1. $1 + \alpha(\delta - 1) \geq \delta_{\mathcal{P}}$. En este caso, para cualquier instancia x de $\mathcal{P}$ y para cualquier solución y' de MAX3-SAT, definimos

   $$g(x, y', \delta) = \mathcal{A}_{\mathcal{P}}(x)$$

   que está bien definida ya que $\mathcal{A}_{\mathcal{P}}(x)$ es una solución factible de $\mathcal{P}$.

   En este caso, $R(x, g(x, y', \delta)) = R(x, \mathcal{A}_{\mathcal{P}}(x)) \leq \delta_{\mathcal{P}} \leq 1 + \alpha(\delta - 1)$, y tenemos que se verifica la AP-condition sin necesidad de definir ni utilizar f.

2. $1 + \alpha(\delta - 1) < \delta_{\mathcal{P}}$. En este caso, sea $\delta_n = 1 + \alpha(\delta - 1)$. Entonces, tenemos que

   $$\delta = \frac{\delta_n - 1}{\alpha} + 1 = \frac{\epsilon}{2(1+\epsilon)} \frac{\delta_n - 1}{\delta_{\mathcal{P}}\ log\delta_{\mathcal{P}} + \delta_{\mathcal{P}} - 1} + 1 < \frac{\epsilon}{2k(1+\epsilon)} + 1 \quad (5.6)$$

   donde $k = \lceil log_{\delta_n}\delta_{\mathcal{P}} \rceil$, y la última desigualdad debido a que

   $$k \leq \frac{log\delta_{\mathcal{P}}}{log\delta_n} + 1 < \frac{\delta_n log\delta_{\mathcal{P}}}{\delta_n - 1} < \frac{\delta_{\mathcal{P}} log\delta_{\mathcal{P}} + \delta_{\mathcal{P}} - 1}{\delta_n - 1}$$

   En la primera desigualdad estricta usamos que $logx \geq (x-1)/x$, y en la segunda que $\delta_n < \delta_{\mathcal{P}}$

   Denotamos como $t_{\mathcal{P}}(x)$ a la medida de la solución $\mathcal{A}_{\mathcal{P}}(x)$. Particionamos el intervalo $[t_{\mathcal{P}}(x), \delta_{\mathcal{P}} t_{\mathcal{P}}(x)]$ en los siguientes k subintervalos ($\delta_n > 1$):

   $$[t_{\mathcal{P}}(x), \delta_n t_{\mathcal{P}}(x)], [\delta_n t_{\mathcal{P}}(x), \delta_n^2 t_{\mathcal{P}}(x)], ..., [\delta_n^{k-1} t_{\mathcal{P}}(x), \delta_{\mathcal{P}} t_{\mathcal{P}}(x)]$$

El último intervalo está bien definido, ya que, de $k = \lceil log_{\delta_n} \delta_{\mathcal{P}} \rceil$ deducimos que $\delta_n^{k-1} \leq \delta_{\mathcal{P}} \leq \delta_n^k$

Como $\mathcal{P}$ es un problema de maximización, tenemos que $t_{\mathcal{P}}(x) \leq m^*_{\mathcal{P}}(x) \leq \delta_{\mathcal{P}} t_{\mathcal{P}}(x) \leq \delta_n^k t_{\mathcal{P}}(x)$. Por tanto, esto nos asegura que el valor del óptimo para la entrada x se encuentra en uno de los k intervalos.

Para cada $i = 0, ..., k-1$ vamos a considerar el algoritmo no determinista Algorithm 2.

Algorithm 2: APX-no determinista

Data: Instancia x de un problema $\mathcal{P}$ de APX, un entero $i \geq 0$
Result: *SI* si existe una solución factible en el intervalo i-ésimo
begin
  Seleccionar una palabra y de manera no determinista con $|y| \leq p(|x|)$, para un polinomio p;
  if $y \in SOL_{\mathcal{P}}(x) \wedge m_{\mathcal{P}}(x, y) \geq \delta_n^i t_{\mathcal{P}}(x) \wedge m_{\mathcal{P}}(x, y) \leq \delta_n^{i+1} t_{\mathcal{P}}(x)$ **then**
    return *SI*
  else
    return *NO*

Aplicamos la técnica de la demostración del teorema Cook-Levin [GJ90a] a cada algoritmo no determinista (para cada $i = 0, ..., k-1$) . La técnica consiste en lo siguiente. Se parte de un problema perteneciente a NP, que es resuelto por una MTND M. Para cada instancia I del problema se construye una expresión Booleana que es satisfactible si y sólo si la máquina M acepta a I.

En nuestro caso, obtenemos k fórmulas Booleanas $\phi_0, ..., \phi_{k-1}$, de forma que, dada una asignación de valores de verdad τ_i que satisface ϕ_i, se puede calcular una solución y de x en tiempo polinómico con $m_{\mathcal{P}}(x, y) \geq \delta_n^i t_{\mathcal{P}}(x)$.

A continuación, aplicamos la función f_s derivada del teorema 4.2 (comentada en la observación que precede al teorema) para obtener una instancia ψ de MAX3-SAT:

$$\psi = f(x, \delta) = f_s(\phi_0) \wedge ... \wedge f_s(\phi_{k-1})$$

Sin pérdida de generalidad podemos suponer que cada $f_s(\phi_i)$ contiene exactamente m cláusulas.

Sea τ cualquier asignación de valores de verdad para ψ, cuya razón de eficacia sea como máximo δ. Entonces,

$$\frac{m^*(\psi)}{m(\psi,\tau)} \leq \delta \Rightarrow -m(\psi,\tau) \leq -\frac{m^*(\psi)}{\delta} \Rightarrow$$

$$\Rightarrow m^*(\psi) - m(\psi,\tau) \leq m^*(\psi) - \frac{m^*(\psi)}{\delta} = m^*(\psi)\frac{\delta-1}{\delta} \leq km\frac{\delta-1}{\delta}$$

Por otra parte, como

$$m^*(\psi) = m^*(f_s(\phi_0)) + ... + m^*(f_s(\phi_i)) + ... + m^*(f_s(\phi_{k-1}))$$

$$m(\psi,\tau) = m(f_s(\phi_0),\tau) + ... + m(f_s(\phi_i),\tau) + ... + m(f_s(\phi_{k-1}),\tau)$$

y para todo i

$$m^*(f_s(\phi_i)) - m(f_s(\phi_i),\tau) \geq 0$$

tenemos entonces que

$$m^*(\psi) - m(\psi,\tau) = m^*(f_s(\phi_0)) - m(f_s(\phi_0),\tau) + ...+$$
$$+... + m^*(f_s(\phi_i)) - m(f_s(\phi_i),\tau) + ... + m^*(f_s(\phi_{k-1})) - m(f_s(\phi_{k-1}),\tau) \geq$$
$$\geq m^*(f_s(\phi_i)) - m(f_s(\phi_i),\tau) = m^*(f_s(\phi_i))\frac{\delta_i-1}{\delta_i} \geq \frac{m}{2}\frac{\delta_i-1}{\delta_i}$$

donde δ_i denota la razón de eficacia de τ respecto a $f_s(\phi_i)$, y la última desigualdad es debido a que al menos la mitad las cláusulas de una fórmula Booleana en forma normal conjuntiva siempre son satisfactibles.

Por tanto, combinando las dos desigualdades, para $i = 0, ..., k-1$,

$$\frac{m}{2}\frac{\delta_i-1}{\delta_i} \leq km\frac{\delta-1}{\delta} \iff \frac{\delta_i-1}{\delta_i} \leq 2k\frac{\delta-1}{\delta} \iff$$

$$\iff 1 - 2k\frac{\delta-1}{\delta} \leq \frac{1}{\delta_i}$$

Usando la desigualdad (5.6): $\delta < 1 + \epsilon/(2k(1+\epsilon))$, llegamos a que

$$\delta_i \leq 1 + \epsilon$$

Para ello, es equivalente probar que $1/\delta_i \geq 1/(1+\epsilon)$, que se puede hacer sustituyendo la desigualdad (5.6) en la desigualdad obtenida, y usando que $2k\epsilon/(2k+\epsilon) \leq \epsilon$

Es decir, se verifica que

$$\frac{m^*(f_s(\phi_i))}{m(f_s(\phi_i),\tau)} \leq \delta_i \leq 1+\epsilon$$

Luego

$$m(f_s(\phi_i),\tau) \geq \frac{m^*(f_s(\phi_i))}{1+\epsilon} \geq (1-\epsilon)m^*(f_s(\phi_i))$$

Esto nos indica que se satisfacen como mínimo, una fracción $(1-\epsilon)$ de cláusulas. De la observación que precede al teorema deducimos que para $i = 0, ..., k-1$, $\tau_i = g_s(\phi_i, \tau)$ satisface ϕ_i si y solo si ϕ_i es satisfactible.

Sea j el máximo i tal que τ_i satisface ϕ_i. Por la observación anterior, que τ_i satisfaga ϕ_i es equivalente a que ϕ_i sea satisfactible. Por la técnica de la demostración del Teorema de Cook-Levin, que ϕ_i sea satisfactible es también equivalente a que la MTND del programa Algorithm 2 para i acepte la entrada x.

Hemos visto antes que el óptimo $m^*_{\mathcal{P}}(x)$ tiene que estar en alguno de los subintervalos. Supongamos que está en el intervalo i^* ($\delta_n^{i^*} t_{\mathcal{P}}(x) \leq m^*_{\mathcal{P}}(x) \leq \delta_n^{i^*+1} t_{\mathcal{P}}(x)$). Consideramos por tanto la MTND Algorithm 2 para $i = i^*$. En este caso, tomando $y = y^*(x)$ se verifica la condición del algoritmo, la instancia x se acepta, y por lo que acabamos de comentar, τ_{i^*} satisface ϕ_{i^*}. Como no puede haber un valor que sea más grande que el óptimo, el máximo i tal que τ_i satisface ϕ_i es i^*, luego $j = i^*$. Tenemos por tanto que

$$\delta_n^j t_{\mathcal{P}}(x) \leq m^*_{\mathcal{P}}(x) \leq \delta_n^{j+1} t_{\mathcal{P}}(x)$$

La técnica de la demostración del Teorema de Cook-Levin nos permite obtener, a partir de τ_j, una solución y de x, de forma que la MTND del Algorithm 2 para $i = j$ acepte la entrada x, verificando que $\delta_n^j t_{\mathcal{P}}(x) \leq m_{\mathcal{P}}(x,y) \leq \delta_n^{j+1} t_{\mathcal{P}}(x)$ (y no tiene por qué se la solución óptima).

Finalmente,

$$R(x,y) = \frac{m^*_{\mathcal{P}}(x)}{m_{\mathcal{P}}(x,y)} \leq \frac{\delta_n^{j+1}}{\delta_n^j} = \delta_n = 1 + \alpha(\delta - 1)$$

y se cumple la AP-condition.

Como $\mathcal{P}$ era un problema de maximización arbitrario en APX, hemos demostrado que MAX3-SAT es completo para la clase de problemas de maximización en APX.

□

Utilizando el resultado anterior a este teorema, como cada problema de minimización en APX puede ser reducido a uno de maximización en APX, tenemos lo siguiente:

Corolario 5.2. *MAX3-SAT es APX-completo.*

5.2.2. Completitud en exp-APX

Ya hemos estudiado con detalle la completitud en la clase APX. En este apartado, vamos a introducir una serie de clases, contenidas en NPO, y que contienen a la clase APX, entre las que encuentra la clase exp-APX, que particularmente tiene cierto interés.

Para poder definir estas clases, necesitamos presentar el concepto de **algoritmo $\delta(n)$-aproximado.**

Definición 5.3. Dado un problema de optimización $\mathcal{P}$ en NPO, un algoritmo aproximado $\mathcal{A}$ para $\mathcal{P}$, y una función $\delta : \mathbb{N} \mapsto (1, \infty)$, decimos que $\mathcal{A}$ es una algoritmo $\delta(n)$-aproximado para $\mathcal{P}$ si, para cualquier instancia x tal que $SOL(x) \neq \emptyset$, la razón de eficacia de la solución factible $\mathcal{A}(x)$ respecto a x verifica la siguiente desigualdad:

$$R(x, \mathcal{A}(x)) \leq \delta(|x|)$$

Definición 5.4. Clase F-APX. Dada una clase de funciones F, la clase F-APX es la clase de todos los problemas $\mathcal{P}$ de NPO tal que, para alguna función $\delta \in F$, existe un algoritmo $\delta(n)$-aproximado polinómico para $\mathcal{P}$.

Tomando F como el conjunto de funciones constantes, $O(logn)$, $\bigcup_{k>0} O(n^k)$ y $\bigcup_{k>0} O(2^{n^k})$ obtenemos las clases APX, log-APX, poly-APX y exp-APX respectivamente, que están relacionadas de la siguiente forma:

$$\text{PTAS} \subseteq \text{APX} \subseteq \text{log-APX} \subseteq \text{poly-APX} \subseteq \text{exp-APX} \subseteq NPO$$

Se puede probar que, en realidad, todas estas inclusiones son estrictas si $P \neq NP$.

Particularmente, nos interesa la clase exp-APX. Esta clase se diferencia de la clase NPO en que los problemas de NPO pueden tener instancias en las que decidir si el conjunto de soluciones factibles es vacío sea una tarea complicada. Para los problemas de exp-APX, la existencia de un algoritmo aproximado exponencialmente permite decidir en tiempo polinómico si el conjunto de soluciones factibles es vacío.

Para abordar el tema de la completitud, vamos a presentar antes un problema (en su versión de maximización y de minimización) que va a ser relevante, tanto para la completitud en exp-APX como en NPO en la sección siguiente.

El problema en cuestión es *Maximum Weighted SAT (MAX-W-SAT)*. Dada una fórmula Booleana ϕ, con variables $x_1, \ldots, x_n$ y unos pesos no negativos $w_1, \ldots, w_n$, encontrar una asignación de valores de verdad τ que satisfaga ϕ y que maximice la función de medida $max(1, \sum_{i=1}^{n} w_i \tau(x_i))$, donde los valores Booleanos $TRUE$ y $FALSE$ se identifican con un 1 y un 0 respectivamente.

La versión análoga de minimización se nota como *MIN-W-SAT*. Además, se tiene que:

Teorema 5.3. *MAX-W-SAT es AP-reducible a MIN-W-SAT, y viceversa. [APMS+99](theorem 8.4)*

Para probar los resultados de completitud en exp-APX debemos modificar ligeramente la definición del problema MIN-W-SAT, añadiendo artificialmente al conjunto de soluciones factibles la solución trivial, es decir, la que asigna el valor de $TRUE$ a todas las variables. Probablemente esta asignación trivial no satisfaga la fórmula, pero no importa, ya que esta solución tendrá el valor máximo posible de la función de medida. De esta forma, nos aseguramos de que el conjunto de soluciones factibles no es un conjunto vacío para ninguna instancia.

Se puede demostrar que esta variante de MIN-W-SAT es un **problema completo para la clase de problemas de minimización en exp-APX**. Además, a partir de este resultado, se prueba que el **problema del viajante de comercio es completo para la misma clase.**

Este último resultado nos encaja con lo que habíamos probado en la sección 3.2: si $P \neq NP$, el problema del viajante de comercio no tiene ningún algoritmo aproximado polinómico para ningún δ.

5.2.3. NPO-completitud

Para cerrar el capítulo de completitud, nos queda estudiar aquellos problemas que son completos en la clase NPO. Estos problemas son los más difíciles de aproximar.

Retomamos de nuevo el problema MAX/MIN-W-SAT, con el siguiente resultado.

Teorema 5.4. *MAX-W-SAT y MIN-W-SAT son problemas NPO-completos.*

Demostración. Un esquema de la demostración podría ser el siguiente. Comenzamos con MAX-W-SAT. En primer lugar, debemos asociar a cualquier problema en NPO una máquina de Turing no determinista. A continuación, aplicamos una variación del teorema de Cook-Levin y demostramos que MAX-W-SAT es completo para la clase de problemas de maximización en NPO.

Consideramos ahora el problema MIN-W-SAT. Podemos probar que este problema es completo para la clase de problemas de minimización en NPO.

Finalmente, utilizando el resultado del teorema 5.3, que nos decía que estos dos problemas son AP-reducibles el uno al otro, obtenemos el desenlace buscado. □

A partir de este resultado, se pueden encontrar más problemas que son NPO-completos. MAX-W-SAT es AP-reducible a MAX-W-3-SAT, que es la variación de MAX-W-SAT en la que la fórmula Booleana de entrada contiene como mucho 3 literales por cláusula. Aplicando el teorema anterior, obtenemos que MAX-W-3-SAT es NPO-completo. De forma similar, también podemos ver que MIN-W-3-SAT es NPO-completo.

Una vez finalizado el estudio de las clases de complejidad aproximadas y sus problemas completos, podemos sintetizar lo estudiado con el siguiente gráfico:

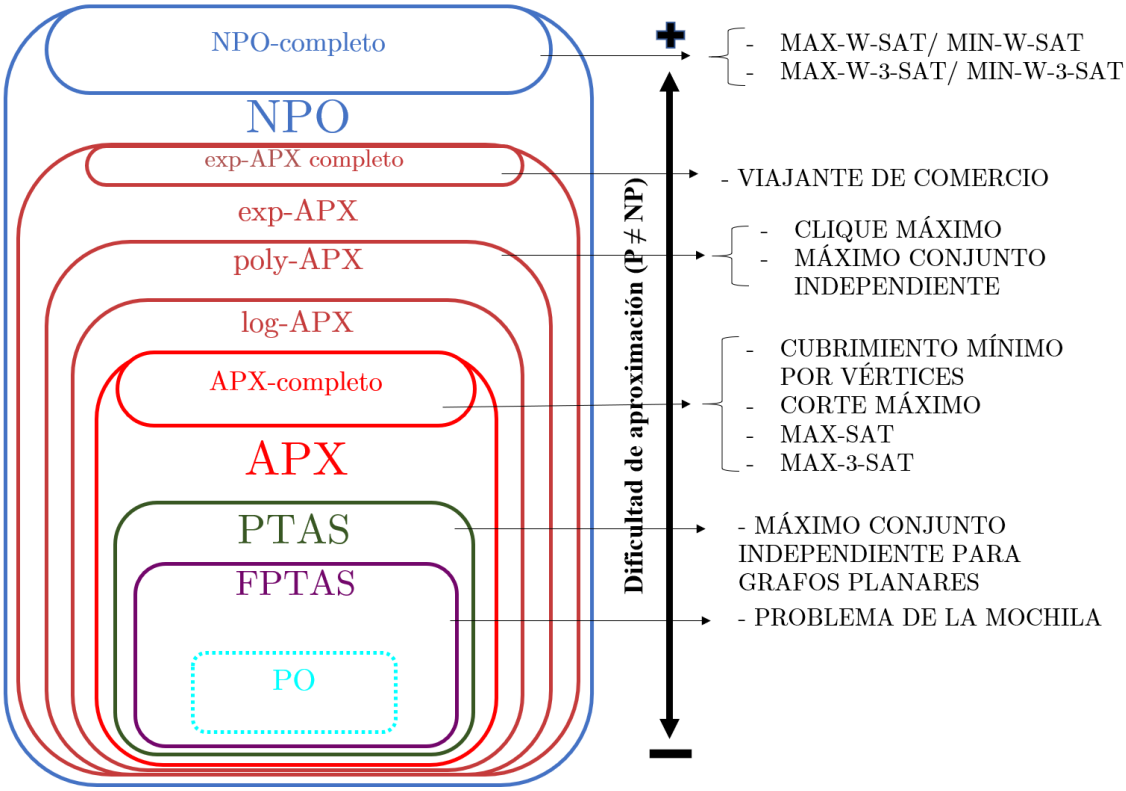


Figura 5.3: Diagrama de clases aproximadas, problemas que contienen y problemas completos.

6 Metaheurísticas

En los capítulos anteriores, el problema de la obtención de buenas soluciones para los problemas de optimización NP-difícil ha estado siempre enfocado en términos de algoritmos que ofrecían un comportamiento garantizado. Esta garantía se refiere a la calidad de la solución obtenida (cómo de cerca está la solución aproximada de la óptima en el peor caso) y al tiempo que necesita el algoritmo para conseguirla (tiempo polinómico en el peor caso).

Sin embargo, existen otro tipo de algoritmos que no ofrecen ninguna garantía teórica, ni en términos de calidad de solución en el peor caso, ni en eficiencia de ejecución, pero que en muchos casos consiguen soluciones muy buenas y en poco tiempo. Estos algoritmos son conocidos como metaheurísticas.

Las **metaheurísticas** las podemos describir como una familia de algoritmos aproximados de propósito general, que suelen ser procedimientos iterativos que guían una heurística subordinada de búsqueda, combinando de forma inteligente distintos conceptos para explorar y explotar adecuadamente el espacio de búsqueda.

6.1. Metaheurísticas frente a la aproximación polinómica de comportamiento garantizado

Las metaheurísticas suelen tener un mal comportamiento si solo nos fijamos en el peor caso, cuando las aplicamos a "malas instancias", ya que nos pueden devolver soluciones muy lejanas respecto al óptimo y en tiempo exponencial. En cambio, en la mayoría de los casos y para la mayoría de las instancias, los resultados obtenidos superan a los algoritmos aproximados polinómicos con garantía de comportamiento, y esto hace que en la práctica sean los más utilizados, siendo uno de los campos de estudio más populares dentro de la computación e inteligencia artificial.

Otro problema de los algoritmos aproximados polinómicos con garantía de aproximación vistos hasta el momento es la dependencia que tienen respecto al problema. Los algoritmos vistos suelen ser muy poco generales y solo aplicables a un determinado problema concreto. La mayoría de las metaheurísticas son de carácter general, siendo bastante flexibles y pudiendo ser aplicadas a numerosos problemas distintos.

Por tanto, además de que son algoritmos de propósito general, las metaheurísticas nos ofrecen numerosas **ventajas**, ya que son fácilmente implementables, fácilmente paralelizables y obtienen un gran éxito en la práctica y en la resolución de problemas del mundo real.

Por otra parte, también tienen sus **inconvenientes**. Son algoritmos no deterministas, es decir, incluyen ciertos componentes aleatorios que hace que el algoritmo encuentre soluciones distintas ante una misma instancia. Finalmente, como ya hemos comentado, en la mayoría de los casos no existe una base teórica establecida. Esta característica es la que hace que este tipo de algoritmos aproximados no sean estudiados en profundidad en este trabajo.

6.2. Metaheurísticas más relevantes

El número de metaheurísticas existentes en la actualidad es muy elevado. Una doble taxonomía de todas las metaheurísticas que se conocen hasta el momento se puede encontrar en [MPS+20]. Algunos de los tipos fundamentales de metaheurísticas son los siguientes: [MMPMV03]

- **Metaheurísticas constructivas.** Construyen soluciones del problema por medio de un procedimiento que incorpora iterativamente elementos a una estructura, inicialmente vacía, que representa a la solución. Las metaheurísticas constructivas establecen estrategias para seleccionar las componentes con las que se construye una buena solución del problema. Entre las metaheurísticas primitivas en este contexto se encuentra la popular estrategia *voraz o greedy*, que ya hemos utilizado a lo largo del trabajo, y que implica la elección que da mejores resultados inmediatos, sin tener en cuenta una perspectiva más amplia. Dentro de este tipo de metaheurística, destaca la aportación de la metaheurística *GRASP* que, en la primera de sus dos fases, incorpora a la estrategia greedy pasos aleatorios con criterios adaptativos para la selección de los elementos a incluir en la solución.

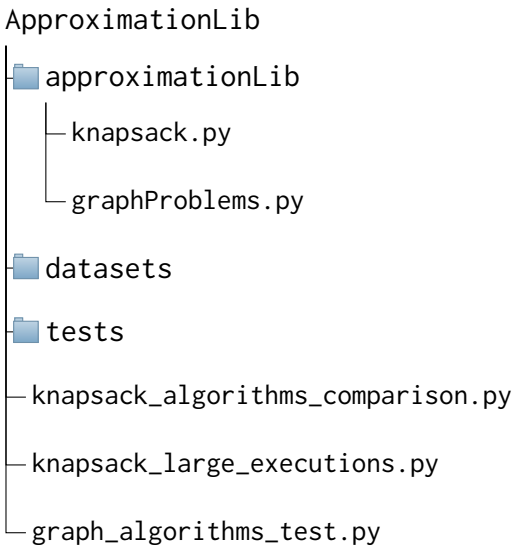
- **Metaheurísticas de búsqueda.** Este tipo de metaheurísticas establecen estrategias para recorrer el espacio de soluciones del problema transformando de forma iterativa soluciones de partida. La metaheurística más popular de este tipo es la *búsqueda local*. Ésta basa su estrategia en el estudio de soluciones del vecindario o entorno de la solución actual. Las metaheurísticas de búsqueda estocásticas establecen pautas para regular la probabilidad de aceptar transformaciones que no mejoren la solución. El *enfriamiento simulado* es el exponente más importante de este tipo de metaheurísticas, donde la probabilidad de aceptación es una función exponencial del empeoramiento producido. Las metaheurísticas de búsqueda con memoria utilizan información sobre el recorrido realizado para evitar que la búsqueda se concentre en una misma zona del espacio. Fundamentalmente se trata de la *búsqueda tabú*, cuya propuesta original prohíbe temporalmente soluciones muy parecidas a las últimas soluciones del recorrido.
- **Metaheurísticas evolutivas.** Las metaheurísticas evolutivas establecen estrategias para conducir la evolución en el espacio de búsqueda de conjuntos de soluciones (usualmente llamados poblaciones) con la intención de acercarse a la solución óptima con sus elementos. El aspecto fundamental de las heurísticas evolutivas consiste en la interacción entre los miembros de la población frente a las búsquedas que se guían por la información de soluciones individuales. Dentro de este tipo destacan los *algoritmos genéticos*, que repiten de forma cíclica las etapas de selección, cruce, mutación y reemplazamiento. Los *algoritmos meméticos* son algoritmos genéticos que incorporan una etapa de mejora local de la población.

Como ya hemos señalado, estas metaheurísticas no ofrecen garantías respecto a la calidad de la solución. Sin embargo, en algunas de ellas, podemos intentar añadir alguna modificación que nos permita obtener esta garantía. Por ejemplo, consideramos un algoritmo genético elitista, donde siempre se mantiene la mejor solución encontrada hasta el momento de todas las generaciones. Modificamos este algoritmo incorporando en la población inicial alguna solución proveniente de un algoritmo aproximado con garantía de comportamiento. Como siempre se mantiene la mejor solución, en el peor de los casos la mejor solución encontrada siempre será alguna de las aproximadas introducidas en la población inicial y por tanto, el genético ofrecerá la misma garantía de comportamiento que el algoritmo aproximado.

7 Aproximación de Problemas

En este capítulo vamos a estudiar algunos algoritmos aproximados para diferentes problemas con más profundidad, con el objetivo de programarlos para confeccionar una biblioteca en **Python** que permita resolver de forma aproximada algunos de los problemas estudiados en el trabajo. El código se puede encontrar en `https://github.com/albertoluin/TFG`

La estructura principal del software implementado es la siguiente:



Los algoritmos implementados se encuentran en los ficheros **`knapsack.py`** y **`graphProblems.py`**. El primer fichero contiene a los cuatro algoritmos estudiados para el problema de la mochila (greedy, greedy 2-aproximado, FPTAS y pseudopolinómico) y el segundo agrupa los algoritmos estudiados asociados a los problemas de grafos (corte máximo y cubrimiento mínimo por vértices).

Además de estos dos ficheros principales, también se ha incluido lo siguiente:

- **knapsack_algorithms_comparison.py**. Este script de Python es un fichero que prueba y compara los distintos algoritmos para el problema de la mochila, utilizando un dataset de un tamaño pequeño para que la ejecución sea rápida.
- **knapsack_large_executions.py**. Este es el script utilizado para ejecutar los algoritmos con instancias más grandes.
- **graph_algorithms_test.py**. Es un script simple que muestra un ejemplo de prueba para cada uno de los algoritmos de grafos implementados.
- En el directorio `dataset` se encuentran los conjuntos de datos utilizados para las ejecuciones, extraídos de [Dat].

Pasamos a estudiar con más detalle los algoritmos implementados. Para el problema de la mochila, al haber estudiado distintas aproximaciones, se ha llevado acabo un pequeño estudio comparativo experimental, donde también se ha incorporado un algoritmo genético y un algoritmo genético 2-aproximado.

7.1. Problema de la Mochila

Como hemos comentado, para este problema vamos a ver varios algoritmos distintos. La primera aproximación es un algoritmo greedy bastante simple e intuitivo, pero que no ofrece ninguna garantía de comportamiento. Veremos que esto hace que en ocasiones puedan encontrarse soluciones realmente alejadas del óptimo.

A continuación, veremos una segunda versión de este algoritmo greedy. Al añadir una pequeña modificación, obtenemos la garantía deseada, convirtiéndose en un algoritmo 2-aproximado.

Finalmente, hemos visto en el ejemplo 3.6 un algoritmo pseudo-polinómico y un FPTAS basado en este algoritmo.

Al igual que hicimos en dicho ejemplo, podemos suponer sin pérdida de generalidad que todos los objetos caben en la mochila. Además, suponemos también que los pesos son mayores que cero.

7.1.1. Greedy sin garantía de comportamiento

Este algoritmo consiste en ordenar los ítems de forma no creciente respecto al ratio valor/peso, e introducir los objetos en ese orden hasta que no quepan más.

Como hemos comentado antes, puede ocurrir que, al no tener ninguna garantía acerca de la bondad de la solución respecto a la óptima, haya algunos casos para los que el algoritmo falle estrepitosamente. Un ejemplo es el siguiente:

Ejemplo 7.1. Sea x una instancia del problema para una mochila de capacidad b con n objetos cuyos pesos y valores son: $w_i = v_i = 1$ para $i = 1, .., n-1$ y $v_n = b - 1$ $w_n = b = kn$, donde k es un número grande arbitrario.

La solución óptima es claramente la que solo introduce el último objeto en la mochila, con $m^*(x) = b - 1$. En cambio, como $\frac{v_i}{w_i} = 1$ para $i = 1, .., n-1$ y $\frac{v_n}{w_n} = \frac{b-1}{b} < 1$, el algoritmo greedy obtendrá como solución una en la que no introduzca el último objeto, donde $m_{Gr}(x) = n - 1$. Por tanto, $\frac{m^*(x)}{m_{Gr}(x)} > k$ donde k era un valor arbitrario que puede ser tan grande como se desee.

Complejidad del algoritmo: Como tenemos que hacer una ordenación de los objetos por el ratio, el algoritmo es $O(nlog(n))$.

El problema en el ejemplo reside en que el algoritmo no incluye el objeto de mayor valor, cuando la solución óptima es precisamente la formada por este objeto. Este ejemplo nos sirve para sugerirnos una posible mejora del algoritmo, que vamos a ver a continuación.

7.1.2. Greedy 2-aproximado

Sea v_{max} el máximo de los valores de los objetos. Partiendo del algoritmo anterior, antes de devolver la solución, comprobamos si la solución obtenida es mejor que la solución consistente en introducir únicamente el objeto de mayor valor. En caso de que sea peor, devolvemos la solución con el único objeto.

Teorema 7.1. *Este algoritmo es* 2*-aproximado*

Demostración. Suponemos una instancia x, con los objetos ordenados ya por el ratio valor/peso. Sea j el primer objeto que el algoritmo no introduce en la mochila. En este punto, el valor total de la mochila vendrá dado por

$$\overline{v}_j = \sum_{i=1}^{j-1} v_i \leq m_{Gr}(x)$$

El peso en este punto será

$$\overline{w}_j = \sum_{i=1}^{j-1} w_i \leq b$$

Vamos a probar que en estas condiciones se cumple la desigualdad

$$m^*(x) < \overline{v}_j + v_j \tag{7.1}$$

Suponiendo que esta desigualdad es cierta ya podemos demostrar que el algoritmo es 2-aproximado. Sea m_{Gr2} la función de beneficio para la versión mejorada del algoritmo. Puede ocurrir dos casos:

- Si $v_j \leq \overline{v}_j$, entonces

$$m^*(x) < \overline{v}_j + v_j \leq 2\overline{v}_j \leq 2m_{Gr}(x) \leq 2m_{Gr2}(x)$$

- Si $\overline{v}_j < v_j$, entonces

$$m^*(x) < \overline{v}_j + v_j \leq 2v_i \leq 2v_{max} \leq 2m_{Gr2}(x)$$

En ambos casos llegamos a que el algoritmo es 2-aproximado.

Nos falta ver que efectivamente se cumple la desigualdad 7.1. Esta desigualdad se ve con facilidad si nos fijamos en el siguiente hecho. En el caso de que se pudieran fraccionar los objetos, la solución óptima estaría formada por los $j-1$ objetos que ya hemos introducido más la fracción del objeto j que quepa en la mochila. Como el objeto j no lo podemos introducir completo porque no cabe, la fracción será menor estricta que 1. Denotamos a la fracción del objeto j que cabe en la mochila como α ($\alpha < 1$).

Sea m_f^* la función de medida para el problema que admite fracciones. Como nuestro problema de la mochila es un caso particular del problema que admite fracciones, se cumple que $m^*(x) \leq m_f^*(x)$ para toda instancia x.

Por tanto, finalmente

$$m^*(x) \leq m_f^*(x) = \overline{v}_j + \alpha v_j < \overline{v}_j + v_j$$

□

Complejidad del algoritmo: La modificación que hemos hecho no afecta en la complejidad, luego sigue siendo $O(nlogn)$.

7.1.3. Algoritmo Pseudo-polinómico

Este algoritmo basado en programación dinámica está explicado con detalle en el Ejemplo 3.4. No obstante, recordamos que consistía en calcular una tabla de valores $W(i,v)$ para $i = 0,..,n$ y $j = 0,..,nV$ donde V es el valor máximo entre los objetos.

$W(i,v)$ indica el mínimo peso que se puede conseguir eligiendo ítems entre los i primeros de valor v exactamente. En caso de que no se pueda conseguir el valor v de forma exacta, le asignamos el valor $+\infty$. Podemos calcularlos de forma recursiva en i de la siguiente forma:

$$W(i+1,v) = min\{W(i,v), W(i,v-v_{i+1}) + w_{i+1}\}$$

Se tiene además que $W(0, v) = +\infty$, con $v \neq 0$, pues con cero ítems no podemos alcanzar ningún valor distinto de cero, y $W(i, 0) = 0$ para todo i.

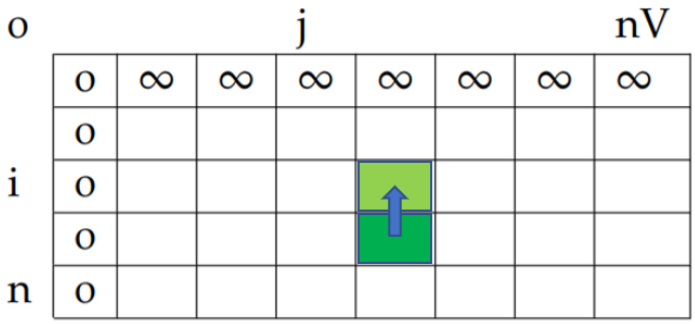
A la hora de la implementación, vamos a tratar de encontrar la manera de poder obtener todos los valores mediante una tabla.

Como resumen, podemos escribir cada $W(i, j)$ como:

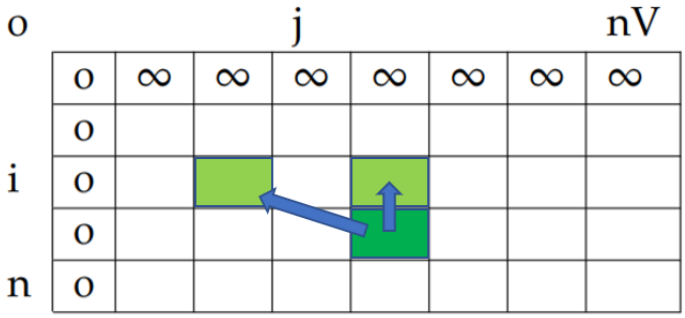
$$W(i,j) = \begin{cases} 0 & \text{si } j = 0 \\ \infty & \text{si } j > 0 \text{ y } i = 0 \\ W(i-1,j) & \text{si } i, j > 0 \text{ y } v_i > j \\ \min\left(W(i-1,j), W(i-1,j-v_i) + w_i\right) & \text{en otro caso} \end{cases}$$

Si lo visualizamos como una tabla, dado un (i, j) no trivial, podemos obtener el valor $W(i, j)$ a partir de:

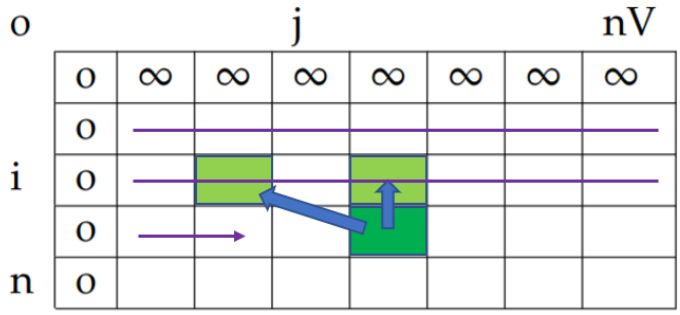
- $W(i-1, j)$. Es justo el elemento en la misma columna y la fila anterior.



- $W(i-1, j)$ y $W(i-1, j-v_j)$. Además del valor anterior, necesitamos otro valor que se encuentra en la fila anterior y en alguna columna inferior.



Por tanto, podemos calcular todos los valores de la tabla si la recorremos en el siguiente sentido:



Una vez obtenida la tabla completa, ya podemos calcular el óptimo fácilmente. Basta con recorrer, desde $i = nV$ hasta $i = 0$ los valores $W(n, i)$. Comenzando por $W(n, nV)$, este valor será igual al mínimo peso que se puede conseguir eligiendo entre todos los objetos cuya suma total tenga valor nV. Si este valor es menor o igual que la capacidad de la mochila, ya hemos encontrado el valor máximo: nV. Si no, comprobamos para $nV - 1$ y así sucesivamente.

Complejidad del algoritmo: Como tenemos que calcular estos valores de la tabla para $i = 1, .., n$ y $1 \leq v \leq nV$, la complejidad del algoritmo es $O(n^2V)$. El valor de V es exponencial en función del tamaño que se necesita para almacenar V, luego este algoritmo es exponencial en función del tamaño de la entrada.

7.1.4. FPTAS

Tal y como se comentó en el Ejemplo 3.4, para cada instancia del problema $I = (w_1, .., w_n, W, v_1, .., v_n)$, construimos una nueva instancia $I' = (w_1, .., w_n, W, v'_1, .., v'_n)$ donde $v'_i = 2^b[\frac{v_i}{2^b}]$ para $i = 1, .., n$ donde $[x]$ se refiere a la parte entera de x. Es decir, los b bits menos significativos se reemplazan por 0.

Concretamente, tomando $b = [log(\frac{(\delta-1)V}{n})]$ y aplicando el algoritmo pseudo-polinómico a la nueva instancia, vimos que se obtenía un esquema de aproximación polinómico total.

Estos algoritmos se encuentran implementados, según lo explicado en esta sección, en el fichero **knapsack.py** de la biblioteca.

7.1.5. Algoritmo genético

Para poder comparar los algoritmos anteriores con otro tipo de algoritmo aproximado, he utilizado la biblioteca **pyeasyga**, que te permite obtener una implementación simple y sencilla de algoritmos genéticos para varios problemas [pyeb] [pyea].

Concretamente, se ha utilizado un modelo elitista de algoritmo genético, donde siempre se mantiene la mejor solución de las generaciones. El algoritmo se ejecuta durante 100 generaciones, con una probabilidad de cruce de 0.8 y una probabilidad de mutación de 0.2.

7.1.6. Algoritmo genético 2-aproximado

Tal y como se describió al final del capítulo 6, podemos modificar el algoritmo genético anterior para que sea un algoritmo δ-aproximado.

Para ello, incluimos la solución 2-aproximada del algoritmo greedy en la población inicial. Como el algoritmo sigue un procedimiento elitista, siempre se mantiene al mejor cromosoma de un generación a la siguiente, y por tanto, la solución final encontrada por el algoritmo deberá ser mejor que la greedy 2-aproximada, o en el peor de los casos, la propia solución greedy.

7.1.7. Comparativa experimental de los algoritmos

Vamos a ejecutar los algoritmos con tres datasets distintos. Para la comparativa, se ha ejecutado el FPTAS con $\delta = 1.5$. Para cada uno de los conjuntos de datos vamos a medir el rendimiento de cada algoritmo comparando la calidad de la solución encontrada respecto a la óptima (óptimo/aproximación) y el tiempo que ha consumido para encontrarla.

En primer lugar, utilizamos un conjunto de instancias sencillas (**`datasets/low-dimensional`**), de tamaño pequeño (alrededor de 20 ítems por instancia).

Al ser un conjunto de datos con instancias de un tamaño pequeño, el algoritmo exacto (pseudo-polinómico) es la mejor opción ya que obtiene la solución en muy poco tiempo y no tiene sentido buscar una aproximación.

Los resultados obtenidos son los siguientes:

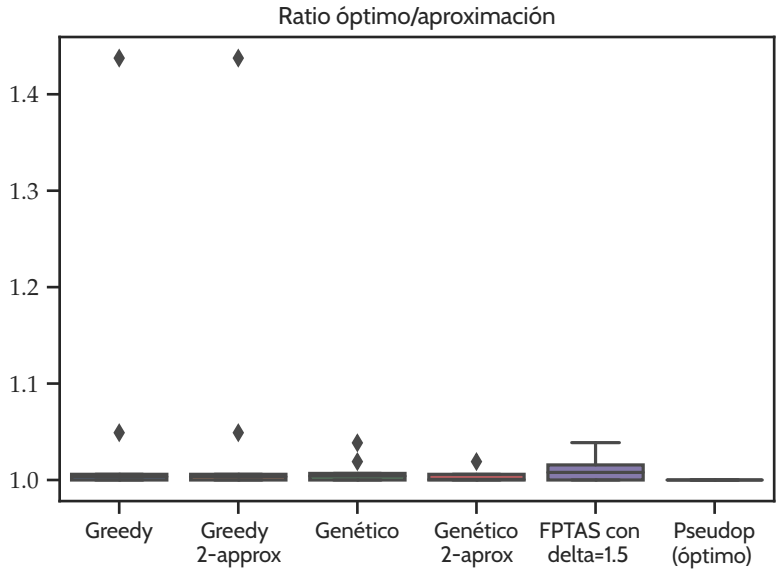


Figura 7.1: Razón de eficacia para `low-dimensional`.

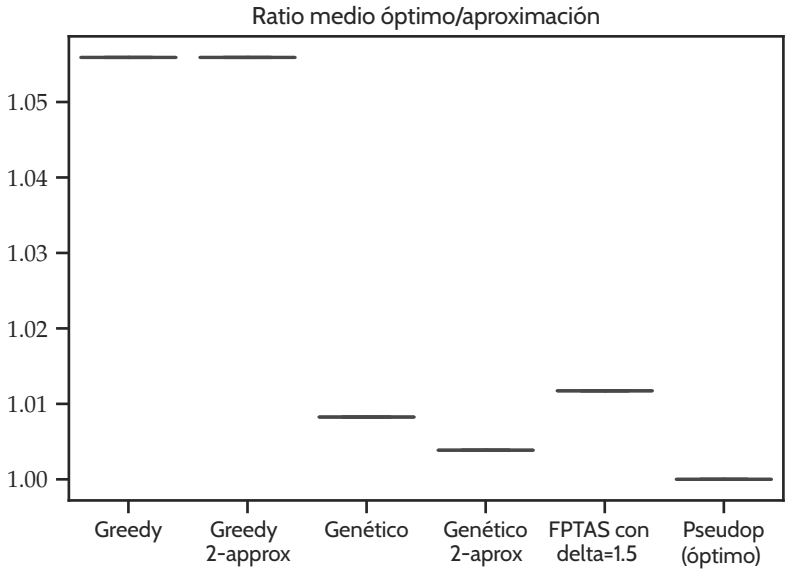


Figura 7.2: Razón de eficacia media para `low-dimensional`.

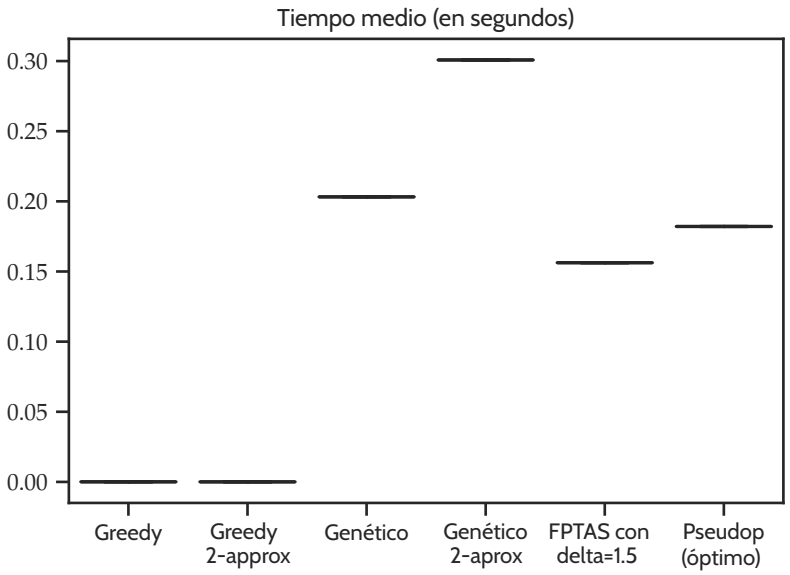


Figura 7.3: Tiempo medio de ejecución para `low-dimensional`.

Ejecutamos ahora los algoritmos con el conjunto de datos ***datasets/different_sizes*** que incorpora instancias de diferentes tamaños, de entre 20 y 1000 ítems. Como los dos algoritmos greedy y los dos genéticos son muy similares, man-

tengo solo las versiones δ-aproximadas para mayor claridad.

En este caso, sí que se empieza a apreciar la valía de los algoritmos aproximados, ya que la diferencia de tiempo cada vez es mayor a medida que aumenta el número de ítems de la instancia.

Cabe remarcar sobre todo el algoritmo greedy, que obtiene soluciones muy cercanas al óptimo y en un tiempo siempre inferior a una décima de segundo

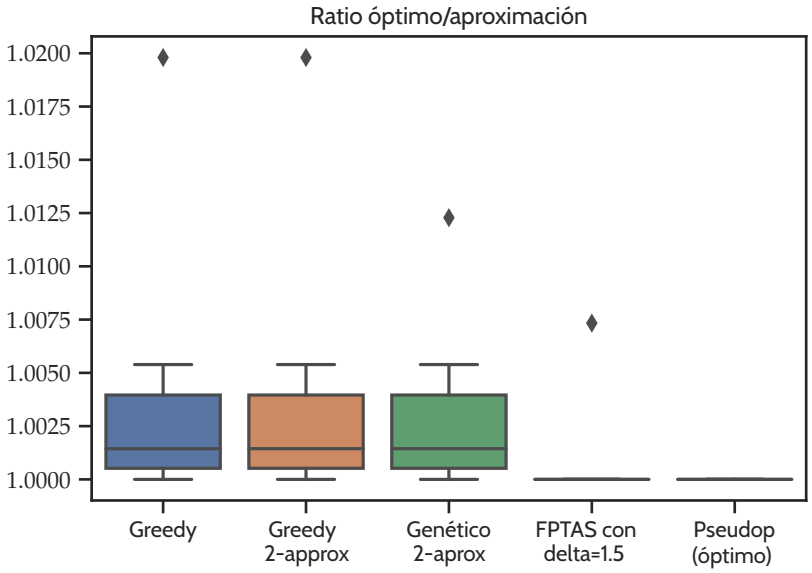


Figura 7.4: Razón de eficacia para `different_sizes`

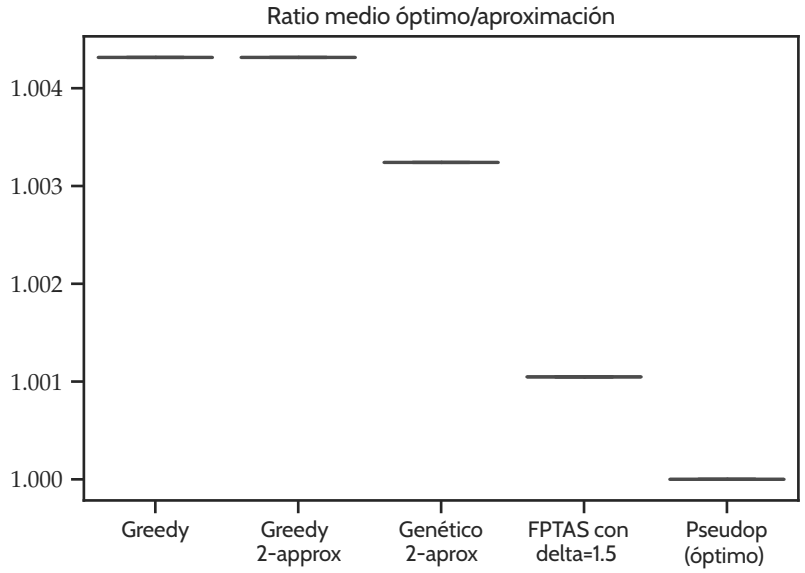


Figura 7.5: Razón de eficacia media para `different_sizes`

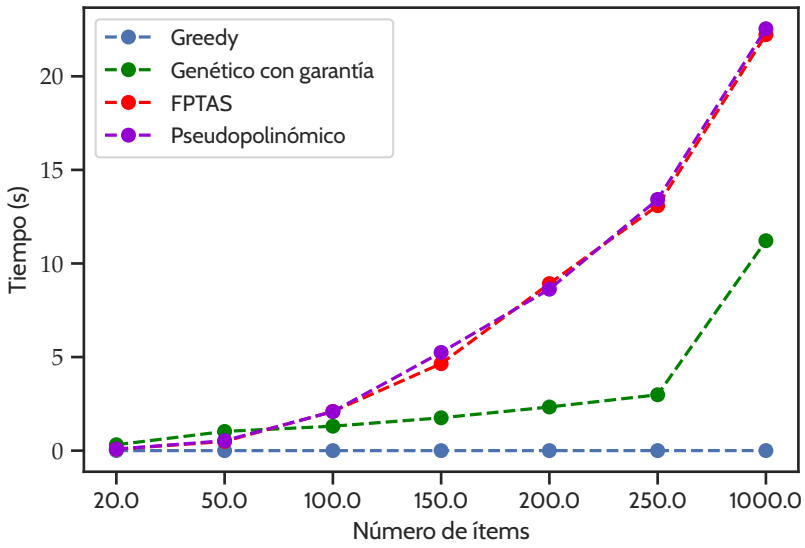


Figura 7.6: Tiempo de ejecución para `different_sizes`

Finalmente, ejecutamos los algoritmos con un tercer conjunto de datos: ***datasets/large_scale***, que consiste en una serie de instancias intencionadamente complicadas y de un tamaño considerable. Concretamente, contiene en total 21 ejemplos del problema, con tres instancias de 100, 200, 500, 1000, 2000, 5000 y 10000 ítems. En este dataset, los tiempos de ejecución son bastante más altos y la ejecución del algoritmo pseudopolinómico es inviable. Para el FPTAS, solo es razonable ejecutar hasta el tamaño 1000 en este dataset.

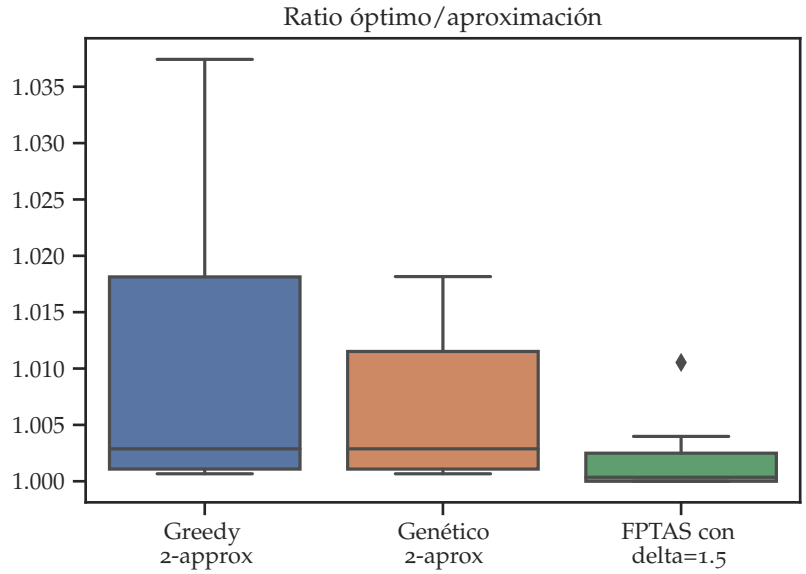


Figura 7.7: Razón de eficacia para `large_scale`

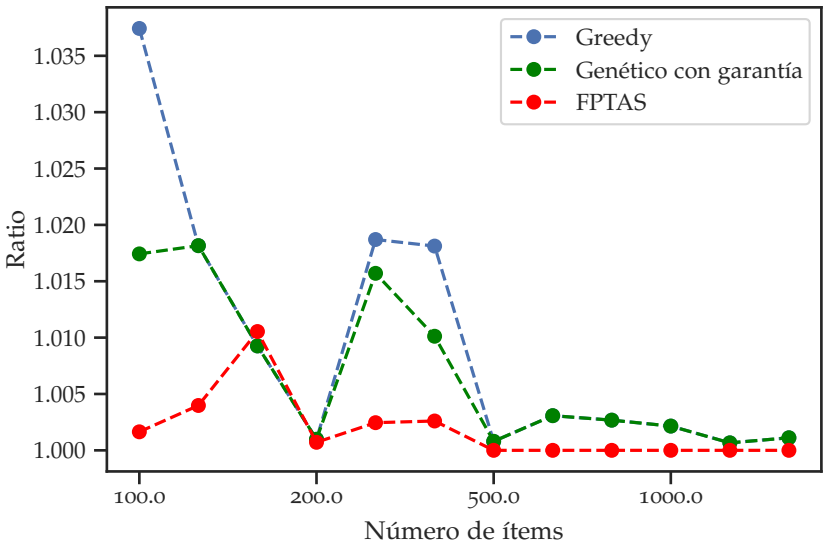


Figura 7.8: Razón de eficacia frente al número de ítems para `large_scale`

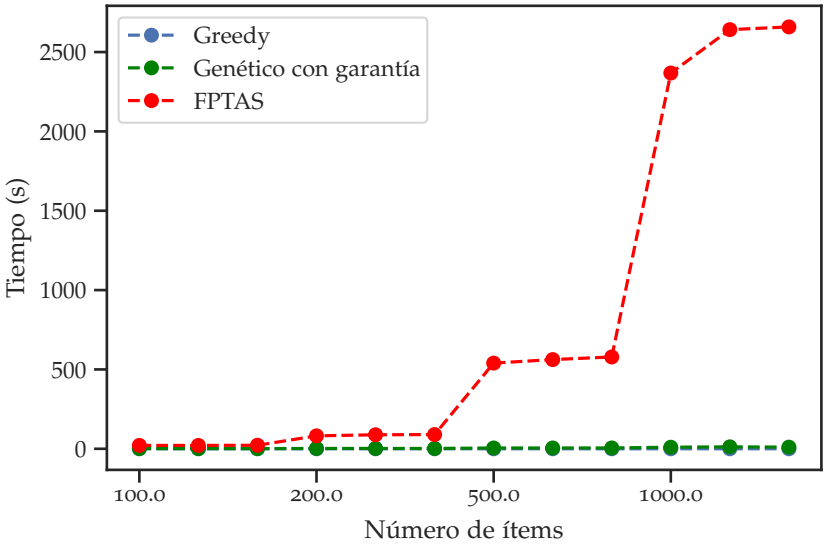


Figura 7.9: Tiempo de ejecución para `large_scale`

En este conjunto de datos la diferencia en tiempo es abismal. Tanto es así

que mientras que los algoritmos greedy y genético tardan del orden de 15 segundos en el peor de los casos, el FPTAS ronda los 40 minutos de ejecución para las instancias de 1000 ítems.

Este dataset incluye instancias de 2000, 5000 y hasta 10000 ítems pero que no pueden ser ejecutadas con el FPTAS por el tiempo y la memoria que necesita. Sin embargo, podemos ejecutar el conjunto de datos completo para el greedy y el genético para poder contrastar la ventaja del greedy respecto al segundo en cuanto al tiempo.

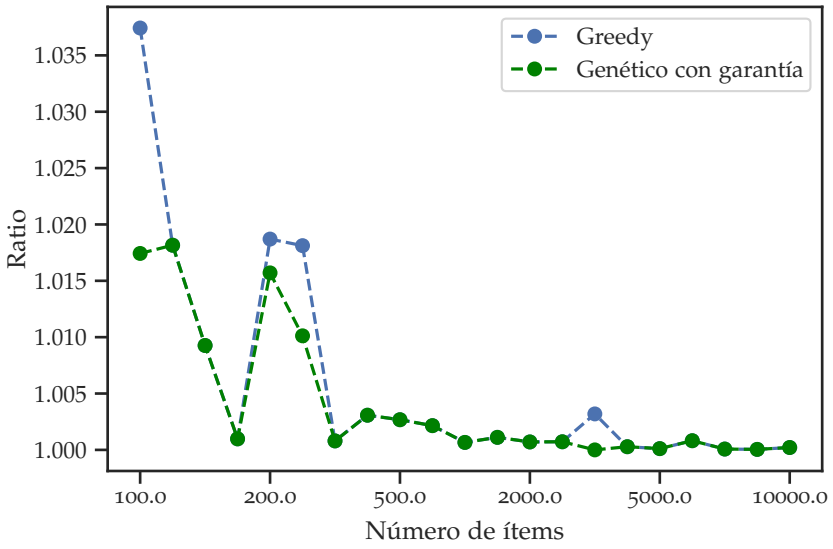


Figura 7.10: Razón de eficacia greedy vs genético `large_scale`

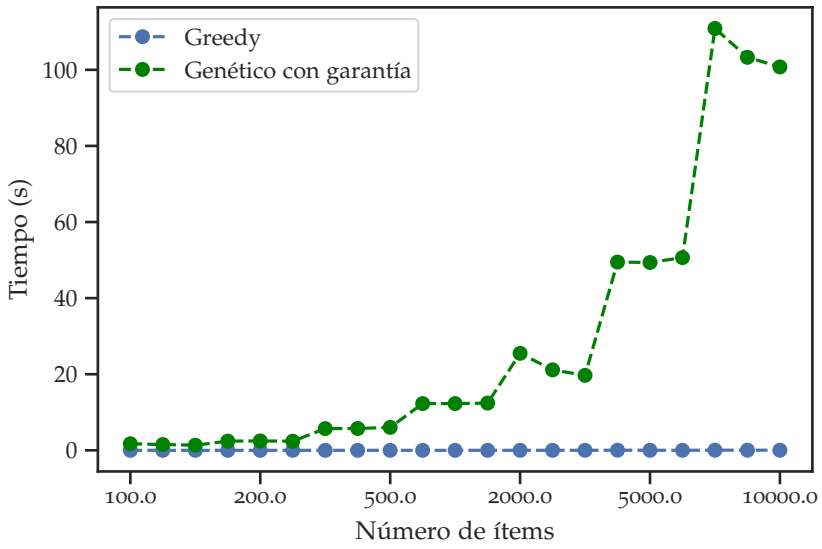


Figura 7.11: Tiempo de ejecución greedy vs genético `large_scale`

7.2. Problemas de grafos

Se han implementado los algoritmos 2-aproximados estudiados en 3.2 y 3.4 para el problema del **cubrimiento mínimo por vértices** y el problema del **corte máximo.**

Estos algoritmos se encuentran en el fichero **`graphProblems.py`** de la biblioteca. Se ha creado una clase *Graph*, partiendo de la presentada en [Gra], que utiliza un diccionario para representar a un grafo.

Se ha creado el script de Python **`graph_algorithms_test.py`** para mostrar un ejemplo de prueba de los algoritmos implementados. Utilizando el paquete **NetworkX** de Python podemos dibujar los grafos para que el resultado sea más visual:

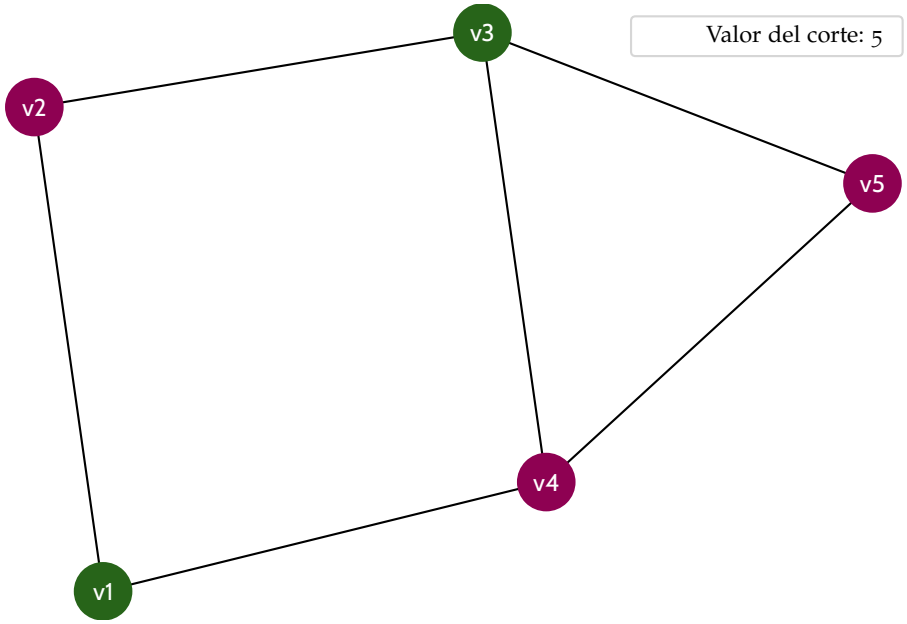


Figura 7.12: Ejemplo de resultado del algoritmo 2-aproximado para el corte máximo

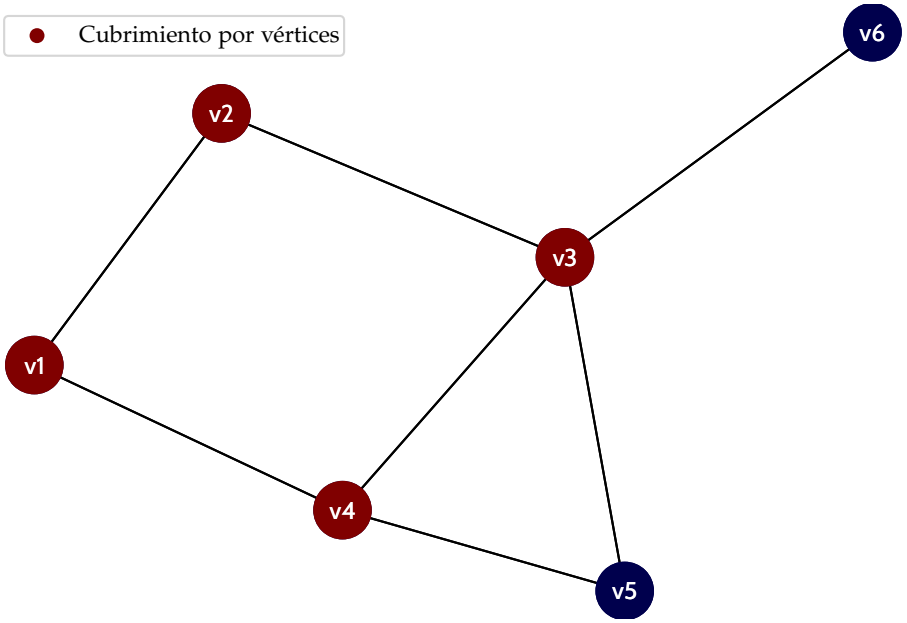


Figura 7.13: Ejemplo de resultado del algoritmo 2-aproximado para el cubrimiento mínimo por vértices

Conclusiones y trabajo futuro

En este proyecto se ha estudiado la complejidad algorítmica en la resolución aproximada de problemas de optimización de tipo combinatorio. Para ello, centrados en los problemas de la clase *NPO*, se ha profundizado en los algoritmos aproximados polinómicos que ofrecen una garantía respecto a la calidad de la solución encontrada en el peor caso. Hemos podido comprobar que esta condición de garantía permite dar más robustez al algoritmo para evitar casos como el visto en 7.1. Se han clasificado los problemas de *NPO* en distintas clases de aproximación en base a la calidad con la que pueden ser aproximados por estos algoritmos. Esta clasificación tiene bastante utilidad ya que, a la hora de enfrentarse a un problema, permite conocer a priori la dificultad de aproximación del mismo.

Se presentó el concepto de *Probabilistically Checkable Proof* (PCP) que, gracias al Teorema PCP, permitió demostrar algunos resultados interesantes de no aproximabilidad, como que MAX3-SAT no puede estar en la clase PTAS. Además, se presentó un nuevo tipo de reducción, la reducción AP, que preserva las propiedades de aproximación entre problemas, y que fue fundamental para estudiar diversos resultados de completitud en las clases APX, exp-APX y NPO.

Finalmente, se construyó una librería en Python implementando algunos de los algoritmos aproximados estudiados en el trabajo. Esto permitió resolver de forma aproximada algunos problemas de optimización, como el problema de la mochila, el cubrimiento mínimo por vértices o el corte máximo. También sirvió para comprobar de forma experimental algunas de las propiedades teóricas estudiadas en los capítulos anteriores.

En definitiva, concluimos que, a la hora de afrontar la resolución de un nuevo problema de optimización, deberíamos comenzar estudiando la clase aproximada a la que pertenece. No es suficiente conocer que el problema de decisión asociado es *NP*-difícil, sino que conocer información sobre su aproximabilidad nos permite seleccionar la forma más adecuada para resolverlo.

El resultado final de trabajo ha sido satisfactorio y ha cumplido la mayoría de los objetivos propuestos inicialmente. No obstante, han quedado muchos aspectos que no han sido abordados o profundizados lo suficiente, que se plantean como **trabajo futuro**:

- El teorema PCP ha resultado ser muy relevante para probar los resultados de no aproximabilidad y de completitud, pero no se ha visto la demostración debido a su extensión y a que requiere de conocimientos que se alejan del contenido del trabajo. Al tratarse de un teorema tan trascendente como sorprendente, se podría estudiar la demostración para entender con detalle el razonamiento seguido para llegar a dicho resultado que, a priori, no parece nada intuitivo.

- Se ha presentado la reducibilidad-AP para el estudio de la completitud de las clases aproximadas. Sin embargo, existen otro tipo de reducciones que podría ser interesante estudiar, como la reducción-L o la reducción-P. También se podría llevar a cabo el estudio de la PTAS-completitud.

- A lo largo del trabajo siempre se han analizado los algoritmos utilizando el punto de vista del peor caso. Esta condición se puede relajar y llevar a cabo un análisis probabilístico de los algoritmos considerando el *caso promedio* o el comportamiento en *casi todos los casos*. Este enfoque también parece tener sentido ya que, desde el punto de vista práctico, tal y como hemos podido comprobar en 7.1.7, en la mayoría de los casos el valor de la solución encontrada está mucho más cerca del óptimo que el valor sugerido por el peor caso. De hecho, hay algunos problemas, como el problema del clique máximo, para los que no se encuentran algoritmos con buenas propiedades de aproximación en el peor caso si $P \neq NP$, y que, en cambio, sí se pueden aproximar de forma más satisfactoria con este otro enfoque.

- En relación a las metaheurísticas, hemos adaptado un tipo de algoritmo genético para que nos asegure que una garantía de comportamiento. Se podrían estudiar más tipos de metaheurísticas y tratar de adaptarlas para que también nos ofrezcan esta garantía teórica.

- La biblioteca confeccionada que recoge los algoritmos analizados en el trabajo está bien como una pequeña muestra práctica de lo estudiado, pero no resulta funcional para utilizarla como una biblioteca que resuelva una gran variedad de problemas de optimización. Para ello, se deberían añadir algoritmos que resuelvan más problemas y además,

tratando de implementar aquellas aproximaciones que sean de mayor calidad en cada caso.

Bibliografía

[AB09] Sanjeev Arora and Boaz Barak. *Computational complexity: a modern approach*. Cambridge University Press, 2009. [Citado en págs. xvii and 51]

[Aha] Dorit Aharonov. Theory of computer science to msc students, spring 2007, lecture 10. https://www.cs.huji.ac.il/course/2006/tcsg/scribes/lecture10.pdf. Courses on Theoretical Computer Science, University of Jerusalem. [Citado en pág. 51]

[APMS+99] G. Ausiello, M. Protasi, A. Marchetti-Spaccamela, G. Gambosi, P. Crescenzi, and V. Kann. *Complexity and Approximation: Combinatorial Optimization Problems and Their Approximability Properties*. Springer-Verlag New York, Inc., 1999. [Citado en págs. xvii, 42, 52, 59, and 76]

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. *Proceedings of the third annual ACM symposium on Theory of computing - STOC 71*, 1971. [Citado en pág. xv]

[CT18] Bastien Chopard and Marco Tomassini. *An Introduction to Metaheuristics for Optimization*. Springer Publishing Company, Incorporated, 1st edition, 2018. [No citado]

[Dat] Datasets para el problema de la mochila. http://artemisa.unicauca.edu.co/~johnyortega/instances_01_KP/. Recurso online. Accedido en junio de 2021. [Citado en pág. 84]

[Din06] Irit Dinur. The pcp theorem by gap amplification. In *Proceedings of the Thirty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '06, page 241–250, New York, NY, USA, 2006. Association for Computing Machinery. [Citado en pág. 52]

[Gar] Sanjam Garg. Lecture 7: Interactive proofs and zero knowledge. https://people.eecs.berkeley.edu/~sanjamg/classes/cs276-fall14/scribe/lec07.pdf. CS 276: Cryptography at UC Berkeley 2014. [Citado en pág. 51]

[GJ90a] Michael R. Garey and David S. Johnson. Computers and intractability; a guide to the theory of np-completeness. 1990. [Citado en págs. xvii and 72]

[GJ90b] Michael R. Garey and David S. Johnson. Computers and intractability; a guide to the theory of np-completeness. pages 48–50, 1990. [Citado en pág. 55]

[Gol] Oded Goldreich. Probabilistic proof systems. *Computational Complexity*, page 349–415. [Citado en pág. 52]

[Gra] Representing graphs (data structure) in python. https://stackoverflow.com/questions/19472530/representing-graphs-data-structure-in-python. Recurso online. Accedido en mayo de 2021. [Citado en pág. 98]

[HMU07] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. *Introduction to automata theory, languages, and computation*. Pearson, 2007. [Citado en pág. 4]

[Kan] Viggo Kann. A compendium of np optimization problems: Vertex cover. http://www.csc.kth.se/~viggo/wwwcompendium/node10.html. Recurso online. Accedido en noviembre de 2020. [Citado en págs. 30 and 40]

[MMPMV03] Belen Melian, José Moreno-Pérez, and J. Moreno-Vega. Metaheuristics: A global view. *Inteligencia Artificial,revista Iberoamericana De Inteligencia Artificial - AEPIA*, 7, 07 2003. [Citado en pág. 80]

[MPS+20] Daniel Molina, Javier Poyatos, Javier Del Ser, Salvador García, Amir Hussain, and Francisco Herrera. Comprehensive taxonomies of nature- and bio-inspired optimization: Inspiration versus algorithmic behavior, critical analysis recommendations. *Cognitive Computation*, 12(5):897–939, 2020. [Citado en pág. 80]

[pyea] Documentación pyeasyga. https://pyeasyga.readthedocs.io/en/latest/#. Recurso online. Accedido en mayo de 2021. [Citado en pág. 90]

[pyeb] Proyecto pyeasyga. https://pypi.org/project/pyeasyga/. Recurso online. Accedido en mayo de 2021. [Citado en pág. 90]

[Vaz10] Vijay V. Vazirani. *Approximation Algorithms*. Springer Publishing Company, Incorporated, 2010. [Citado en pág. 36]