Instituto Politécnico Nacional

Centro de Investigación en Computación

“Algoritmo de aproximación aleatorizado para el Problema de Selección de k Centros”

Tesis
que para obtener el grado de
Maestro en Ciencias de la Computación

presenta:

Ing. Jesús García Díaz

Director de tesis:

Dr. Rolando Menchaca Méndez

México D.F. a 13 de mayo del 2013

SIP-14

INSTITUTO POLITÉCNICO NACIONAL
SECRETARÍA DE INVESTIGACIÓN Y POSGRADO

ACTA DE REVISIÓN DE TESIS

En la Ciudad de México, D.F. siendo las 16:00 horas del día 13 del mes de diciembre de 2012 se reunieron los miembros de la Comisión Revisora de la Tesis, designada por el Colegio de Profesores de Estudios de Posgrado e Investigación del:

Centro de Investigación en Computación

para examinar la tesis titulada:

"Algoritmo de aproximación aleatorizado para el Problema de Selección de k Centros"

Presentada por el alumno:

GARCÍA	DÍAZ	JESÚS
Apellido paterno	Apellido materno	Nombre(s)

Con registro: | A | 1 | 1 | 0 | 8 | 6 | 6 |

aspirante de: **MAESTRÍA EN CIENCIAS DE LA COMPUTACIÓN**

Después de intercambiar opiniones los miembros de la Comisión manifestaron ***APROBAR LA TESIS***, en virtud de que satisface los requisitos señalados por las disposiciones reglamentarias vigentes.

LA COMISIÓN REVISORA

Director de Tesis

Dr. Rolando Menchaca Méndez

Dr. Marco Antonio Moreno Armendáriz

Dr. Salvador Godoy Calderón

Dr. Rolando Quintero Téllez

M. en C. Germán Téllez Castillo

M. en C. Sergio Sandoval Reyes

PRESIDENTE DEL COLEGIO DE PROFESORES

INSTITUTO POLITÉCNICO NACIONAL
CENTRO DE INVESTIGACIÓN EN COMPUTACIÓN
DIRECCIÓN

Dr. Luis Alfonso Villa Vargas

INSTITUTO POLITÉCNICO NACIONAL

SECRETARÍA DE INVESTIGACIÓN Y POSGRADO

CARTA CESIÓN DE DERECHOS

En la Ciudad de México el día 13 del mes de mayo del año 2013, el que suscribe Jesús García Díaz alumno del Programa de Maestría en Ciencias de la Computación con número de registro A110866, adscrito al Centro de Investigación en Computación, manifiesta que es autor intelectual del presente trabajo de Tesis bajo la dirección de Rolando Menchaca Méndez y cede los derechos del trabajo intitulado "Algoritmo de aproximación aleatorizado para el Problema de Selección de k Centros" al Instituto Politécnico Nacional para su difusión con fines académicos y de investigación.

Los usuarios de la información no deben reproducir el contenido textual, gráficas o datos del trabajo sin el permiso expreso del autor y/o director del trabajo. Este puede ser obtenido escribiendo a la siguiente dirección: jgarciad1106@alumno.ipn.mx. Si el permiso se otorga, el usuario deberá dar el agradecimiento correspondiente y citar la fuente del mismo.

Jesús García Díaz

Nombre y firma

Dedicatoria

A quien corresponda.

Agradecimientos

Gracias a mis padres.

Gracias a mi hermana.

Gracias a mi novia.

Gracias a mis amigos.

Gracias a mi asesor y a todos mis maestros.

Gracias al CIC, al IPN y al CONACYT.

Gracias a mis mascotas.

La presente tesis está basada en trabajo realizado gracias al financiamiento del Instituto México Estados Unidos de la Universidad de California (UC MEXUS) y el Consejo Nacional de Ciencia y Tecnología de México (CONACYT).

Abstract

The k-center selection problem takes as input a complete non-directed weighted graph $G = (V, E)$ and a positive integer k and consists in finding a set of centers $C \subseteq V$ of cardinality at most k, such that the maximum distance from every node to its nearest center is minimized. This maximum distance is called the coverage distance and is denoted by $r(C)$.

The k-center selection problem has been widely studied in areas like *Clustering* and *Facility Location*. This problem has many practical applications related to the distribution of facilities such as emergency services, computer network services, and many others.

The k-center selection problem is an NP-$Hard$ problem, which means that if $P \neq NP$ it cannot be efficiently solved, namely, in polynomial time. Currently, the only known method for solving this problem to optimality is the *brute force* method, which consists in testing every single possible solution and returning the best. This method's complexity is $O(n^k)$ and hence, it is not efficient for arbitrary instances and in particular for instances where the value of k is large.

It is important to highlight that as well as finding the optimal solution for the k-center selection problem is in the NP-$Hard$ class, finding a $(2-\varepsilon)$-approximation, with $\varepsilon > 0$ is also in NP-$Hard$. This bound is essentially tight since the best known approximation algorithms find solutions with an approximation factor of 2 (i.e., they find a 2-approximation). Since it is not possible to design better polynomial algorithms (unless $P = NP$), this kind of problems are usually addressed with very diverse approaches, whose goal is to get solutions as close as possible to the optimal.

To date, many algorithms for solving the k-center selection problem have been designed. Some of the techniques used by these algorithms are greedy selections, parametric pruning, linear programming, tabu search and scatter search. In practice, the algorithms that tend to find the best solutions are those classified as metaheuristics. Unfortunately, this kind of algorithms do not have, in general, strong theoretical basis and therefore it is not possible to formally characterize their complexity or to demonstrate that they will find *good* solutions on arbitrary instances.

In this thesis we present a new *randomized approximation algorithm* of complexity $O(kn)$ for the k-center selection problem. The proposed randomized algorithm is based on a novel technique which aims at combining the advantages of the deterministic approximation algorithms with those of the amplified randomized algorithms. The proposed algorithm performs a randomized search in the solutions space, guided by the search procedure used by a deterministic approximation algorithm. The salient property of the proposed algorithm is that it finds ρ-approximated solutions with a characterizable probability in polynomial time.

We show that the proposed algorithm (amplified α times) finds a ρ-approximation with a probability dependent on α, where ρ is the approximation factor of the algorithm used as basis and $\alpha \in \mathbb{Z}^+$ is a user-defined parameter of the proposed algorithm. Even though the probability of getting a ρ-approximation is α-dependent, it is possible to demonstrate that this probability will never be smaller than $1 - \frac{1}{e}$.

Additionally, a probabilistic local search variant of the proposed algorithm is presented. Our local search scheme consists in investigating solutions located nearby the current best solution. The proposed neighbor relation is also probabilistic in the sense that the solutions examined by the algorithm are those produced by sampling a probability distribution that is designed to produce the current best known solution with high probability. The experimental results show that no more than $O(n^4)$ steps are needed by the proposed algorithm to get solutions as *good* as those found by the best polynomial heuristics; actually it finds the best known solution for half of the used test instances.

Resumen

El problema de selección de k-centros (*k-center selection problem*) consiste en, dado un grafo completo no dirigido $G = (V, E)$ y un entero positivo k, determinar un conjunto de centros $C \subseteq V$, de cardinalidad menor o igual a k, tal que la máxima distancia de cada nodo hacia su centro más cercano sea mínima. A la máxima distancia obtenida se le denomina *radio de cobertura* $r(C)$.

El problema de selección de k-centros ha sido ampliamente estudiado en áreas como *Agrupamiento* (*Clustering*) y *Ubicación de Facilidades* (*Facility Location*). Este problema cuenta con gran número de aplicaciones prácticas, tales como la distribución de inmuebles, de servicios de emergencia, de servicios en redes de computadoras, etcétera.

El problema de selección de k-centros es un problema que pertenece a la clase de complejidad *NP-Difícil*, lo cual implica que en caso de que $P \neq NP$ no puede ser resuelto de forma eficiente, es decir, en tiempo polinomial. De hecho, el único método conocido para resolver este problema de manera óptima es el de *fuerza bruta*, el cual consiste en comparar el radio de cobertura de todas las posibles soluciones y retornar aquella de tamaño mínimo. Este método se ejecuta en $O(n^k)$ pasos, que es una cantidad de pasos exponencial con respecto a la variable k y por lo tanto no es considerado un método viable para instancias arbitrarias, en particular para valores grandes de k.

Es importante destacar que, de la misma manera que encontrar la solución óptima al problema de selección de k-centros es un problema *NP-Difícil*, encontrar una $(2-\varepsilon)$-aproximación, donde $\varepsilon > 0$, también lo es. Esta cota de aproximación es estrecha, pues los mejores algoritmos polinomiales conocidos para resolver el problema de selección de k-centros encuentran soluciones con un factor de aproximación de 2 (es decir, entregan una 2-aproximación). Dado que no es posible diseñar mejores algoritmos polinomiales (a menos que $P = NP$), este tipo de problemas suelen ser abordados a través de enfoques muy diversos cuya finalidad es aproximarse tanto como sea posible a la solución óptima.

A la fecha se ha diseñado una gran cantidad de algoritmos para resolver el problema de selección de k-centros. Algunas de los técnicas utilizadas por estos algoritmos son las selecciones voraces (*greedy*), poda paramétrica, programación lineal, búsqueda tabú, búsqueda dispersa, etcétera. En la práctica los algoritmos que suelen entregar mejores resultados son aquellos clasificados como metaheurísticas. Desafortunadamente, este tipo de algoritmos, en general, no poseen una base teórica fuerte y por lo tanto no es posible caracterizar formalmente su complejidad o bien demostrar que encuentran soluciones *buenas* para instancias arbitrarias.

En esta tesis proponemos un nuevo *algoritmo de aproximación aleatorizado* de complejidad $O(kn)$ para el problema de selección de k-centros. El algoritmo aleatorizado propuesto está basado en una nueva técnica que tiene por objetivo combinar las ventajas de los algoritmos de aproximación deterministas con las de los algoritmos aleatorizados amplificados. El algoritmo propuesto realiza una búsqueda aleatoria dentro del espacio de soluciones con base en el procedimiento seguido por un algoritmo de aproximación determinista. Una propiedad sobresaliente del algoritmo propuesto es que, en tiempo polinomial, encuentra una ρ-aproximación con una

probabilidad caracterizable.

Demostramos que el algoritmo propuesto amplificado α veces entrega una ρ-aproximación con una probabilidad dependiente del valor de α, donde ρ es el factor de aproximación del algoritmo utilizado como base y $\alpha \in \mathbb{Z}^+$ es un parámetro de entrada del algoritmo propuesto definido por el usuario. Si bien la probabilidad de entregar una ρ-aproximación depende del valor de α, también es posible demostrar que esta probabilidad nunca será menor a $1 - \frac{1}{e}$.

Adicionalmente, se propone una variante de búsqueda local probabilística para el algoritmo propuesto. El esquema de nuestro método de búsqueda local consiste en examinar soluciones cercanas a la mejor solución actual. La relación de vencindad propuesta es también probabilística, en el sentido de que las soluciones examinadas son aquellas que resultan del muestreo de una distribución de probabilidad que es definida de tal manera que la mejor solución actual sea obtenida con una alta probabilidad. Los resultados experimentales obtenidos demuestran que no más de $O(n^4)$ pasos son requeridos por el algoritmo propuesto para encontrar soluciones tan buenas como las entregadas por los mejores algoritmos polinomiales del estado del arte, encontrando incluso la mejor solución conocida en la mitad de las instancias de prueba utilizadas.

Índice general

Abstract xi

Resumen xiii

Índice general xv

Índice de figuras xvii

Índice de algoritmos xix

Índice de tablas xxi

1. Introducción **1**

1.1. Planteamiento del problema . . . 3
1.2. Objetivos . . . 4
1.2.1. Objetivo general . . . 4
1.2.2. Objetivos particulares . . . 4
1.3. Justificación . . . 5
1.4. Organización de la tesis . . . 6

2. Marco teórico **7**

2.1. Intratabilidad computacional . . . 7
2.1.1. Clases P y NP . . . 8
2.1.2. Clase NP-Completo . . . 12
2.1.3. Clase NP-Difícil . . . 15
2.2. Algoritmos de aproximación . . . 19
2.3. Algoritmos aleatorizados . . . 21
2.3.1. Conceptos de probabilidad . . . 22
2.3.2. Algoritmos aleatorizados tipo *Las Vegas* . . . 27
2.3.2.1. Algoritmo *Random QuickSort* . . . 27
2.3.3. Algoritmos aleatorizados tipo *Monte Carlo* . . . 31
2.3.3.1. Algoritmo para encontrar el Corte Mínimo . . . 32
2.3.3.2. Amplificación . . . 33
2.4. Problema de selección de k-centros . . . 34
2.4.1. Complejidad del problema de selección de k-centros . . . 36
2.4.2. Aplicaciones y clasificaciones del problema de selección de k-centros . . . 38
2.5. Discusión . . . 39

3. Trabajos relacionados **41**
3.1. Algoritmos de aproximación para el problema de selección de k-centros 41
3.1.1. Algoritmo de González 41
3.1.2. Algoritmo de Hochbaum y Shmoys 44
3.1.3. Algoritmo de Shmoys, Kleinberg y Tardos 48
3.2. Heurísticas y metaheurísticas para el problema de selección de k-centros 51
3.2.1. Algoritmo de Mihelic y Robic 52
3.2.2. Algoritmo de Daskin 54
3.2.3. Algoritmo de Mladenovic 57
3.2.4. Algoritmo de Pacheco y Casado 62
3.3. Metaheurísticas del estado del arte similares a la técnica propuesta 67
3.3.1. Algoritmos de Estimación de Distribución (EDAs) 68
3.3.1.1. Ejemplo de algoritmo EDA 69
3.3.2. Procedimientos de Búsqueda Voraz Aleatorizada Adaptable (GRASP) . 72
3.4. Análisis experimental de los algoritmos del estado del arte 74
3.5. Discusión 79

4. Propuesta **81**
4.1. Propuesta general 82
4.2. Generalización del algoritmo de González 83
4.3. Algoritmo propuesto 87
4.3.1. Función de masa de probabilidad *Caracterizable* 91
4.3.2. Caracterización teórica 95
4.3.3. Heurísticas de búsqueda local 99
4.3.4. Complejidad 100
4.4. Similitudes entre la técnica propuesta y algunas metaheurísticas del estado del arte 101
4.5. Discusión 102

5. Resultados **105**
5.1. Resultados experimentales 106
5.2. Discusión 107

6. Conclusiones **115**

Trabajo futuro **119**

Bibliografía **121**

Índice de figuras

1.1. Problema de selección de k-centros 4

2.1. Ejemplos gráficos de las notaciones O, Θ y Ω. 8
2.2. Máquina de Turing 9
2.3. Máquina de Turing no determinista con módulo de suposición 11
2.4. Posible relación entre las clases P y NP 13
2.5. Clase NP (si $P \neq NP$) 14
2.6. Máquina de Turing oráculo 17
2.7. Relación entre las clases P, NP, *NP-Completo* y *NP-Difícil* 19
2.8. Elementos principales de un modelo probabilístico 23
2.9. Ejemplo de árbol binario generado por RandQS 30
2.10. Ejemplo de contracción de nodos 32

3.1. La cota de aproximación del algoritmo de González es estrecha 44
3.2. Solución gráfica a un problema de programación lineal con dos variables. 55
3.3. Sustitución de cadena con una y dos listas tabú 61
3.4. Evolución de un conjunto de referencia (búsqueda dispersa) 63
3.5. Re-encadenamiento de trayectoria 64
3.6. Búsqueda GRASP dentro del espacio de soluciones de un problema 64
3.7. Búsqueda aleatoria realizada por el Algoritmo 3.20 y el *algoritmo propuesto* . . 74
3.8. Comparación de la calidad de los algoritmos del estado del arte (grafos 1 al 15) 76
3.9. Comparación de la calidad de los algoritmos del estado del arte (grafos 1 al 40) 78

4.1. Función de masa de probabilidad basada en la razón de dos distancias (BD2) . 85
4.2. Funciones de masa de probabilidad con las que se experimentó sobre la generalización del algoritmo de González (GGon). 85
4.3. Calidad de las soluciones entregadas por el algoritmo GGon con diferentes funciones de masa de probabilidad 86
4.4. Funciones $\left(1-\frac{1}{\alpha}\right)^{\alpha}$ y $1-\left(1-\frac{1}{\alpha}\right)^{\alpha}$ dentro del rango $[1,\infty)$ 89
4.5. Búsqueda probabilística del *algoritmo propuesto* con y sin *memoria* 92
4.6. Distribución de las soluciones utilizando diferentes funciones de masa de probabilidad 94

5.1. Comparación de la calidad de algunos algoritmos del estado del arte y del *algoritmo propuesto* (grafos 1-15) 110
5.2. Comparación de la calidad de algunos algoritmos del estado del arte y del *algoritmo propuesto* (grafos 1 al 40) 111

Índice de algoritmos

2.1. Algoritmo RandQS . . . 29

3.1. Algoritmo de González . . . 42
3.2. Algoritmo de Hochbaum y Shmoys . . . 48
3.3. Algoritmo de Shmoys . . . 49
3.4. Algoritmo de Kleinberg y Tardos . . . 50
3.5. Heurística para resolver el problema del Conjunto Dominante Mínimo . . . 53
3.6. Algoritmo de Mihelic y Robic . . . 54
3.7. Algoritmo de Daskin . . . 57
3.8. Procedimiento *Interchange* . . . 59
3.9. Búsqueda tabú con sustitución de cadena . . . 60
3.10. Procedimiento *Move* . . . 61
3.11. *Chain-substitution* 1 . . . 61
3.12. *Chain-substitution* 2 . . . 61
3.13. Procedimiento *Update* . . . 62
3.14. Algoritmo de Pacheco y Casado (*SS*) . . . 65
3.15. Procedimiento *Ávido-Aleatorio* (*Generador-Diversificador*) . . . 65
3.16. Procedimiento *Alternate* . . . 66
3.17. Procedimiento *Interchange* . . . 67
3.18. Estructura básica de un algoritmo EDA . . . 69
3.19. Metaheurística GRASP . . . 73
3.20. Construcción_Voraz_Aleatorizada(Entrada, α) . . . 73
3.21. Búsqueda_Local(Solución) . . . 73

4.1. Algoritmo de González y Kleinberg-Tardos (Gon) . . . 83
4.2. Generalización del algoritmo de González (GGon) . . . 84
4.3. Algoritmo propuesto . . . 90
4.4. Heurística para utilizar el *algoritmo propuesto con memoria* . . . 90
4.5. Algoritmo propuesto con memoria . . . 91

Índice de tablas

2.1. Comparación de la eficiencia entre funciones polinomiales y exponenciales. . . . 9

3.1. Población inicial, D_0 70
3.2. Individuos seleccionados D_0^{Se}, de la población inicial D_0 70
3.3. Población de la primera generación, D_1 71
3.4. Individuos seleccionados D_1^{Se}, de la primera generación D_1 72
3.5. Resultados de los algoritmos del estado del arte ejecutados sobre los grafos *pmed* de la librería OR-Lib 77
3.6. Factores de aproximación experimentales (grafos 1 al 40) 78
3.7. Factores de aproximación experimentales (grafos 1 al 15) 78
3.8. Complejidad de los algoritmos del estado del arte 79

4.1. Eficacia experimental del algoritmo GGon (generalización del algoritmo de González) utilizando diferentes funciones de masa de probabilidad 87
4.2. El algoritmo González+ tiende a ser mejor que el algoritmo González 97
4.3. El *algoritmo propuesto* amplificado α^2 veces tiende a ser mejor que cuando es amplificado α veces ($\alpha = n$) 99
4.4. Complejidad del algoritmo de González (Gon) 101
4.5. Complejidad del *algoritmo propuesto* (AP) con $\alpha = n$ 101

5.1. Resultados de algunos algoritmos del estado del arte ejecutados sobre los grafos *pmed* de la librería OR-Lib, utilizando las variantes (A) e (I) 108
5.2. Resultados del *algoritmo propuesto* ejecutado sobre los grafos *pmed* de la librería OR-Lib, utilizando las variantes (A) e (I) 109
5.3. Factores de aproximación experimentales (grafos 1 al 40) 112
5.4. Factores de aproximación experimentales (grafos 1 al 15) 113

Capítulo 1

Introducción

Los problemas tratados en las ciencias de la computación suelen ser clasificados en función de su *complejidad* espacial y temporal; es decir, en función de la cantidad de espacio (o memoria) que requieren y de la cantidad de tiempo requerido para su resolución. Las principales clases de complejidad temporal en que se clasifican los problemas computacionales son la *Polinomial* denotada por P, la *Polinomial No Determinista* denotada por NP, la *Polinomial No Determinista Completa* denotada por NP-*Completo* y la *Polinomial No Determinista Difícil* denotada por NP-*Difícil*. Existen más clases de complejidad, como por ejemplo el complemento de cada una de las anteriores; sin embargo, para fines de la presente tesis, la definición de las clases mencionadas es suficiente. En el Capítulo 2 se presenta la definición formal de estas clases de complejidad. La definición manejada en el presente capítulo es sólo un bosquejo introductorio.

Los problemas pertenecientes a la clase P se caracterizan por el hecho de que es conocido al menos un algoritmo que permite resolverlos de forma *eficiente*. Si el tiempo de ejecución de un algoritmo es *polinomial* con respecto al tamaño de la entrada, se dice que el algoritmo es *eficiente*, mientras que cuando el tiempo de ejecución es exponencial se dice que no es *eficiente*. En la Tabla 2.1 se aprecian algunos ejemplos donde el tiempo de ejecución de algoritmos polinomiales es menor al de los exponenciales. La clase P puede ser definida como aquella a la que pertenecen los problemas que pueden ser resueltos en tiempo polinomial. La clase P es un subconjunto de la clase NP, de modo que todo problema en P también es un problema NP. Los problemas NP son aquellos para los cuales no necesariamente se conoce un algoritmo eficiente, pero que sí pueden ser *verificados* en tiempo polinomial. Un problema es verificable en tiempo polinomial si, dada una entrada y una *supuesta* solución (comúnmente llamada *certificado*), es posible determinar en tiempo polinomial si la *supuesta* solución en realidad lo es.

Una pregunta abierta en ciencias de la computación es si $P = NP$ o $P \neq NP$; es decir, si la clase $NP - P$ es igual al vacío o no. Si P es igual a NP, entonces existe un algoritmo polinomial para todos los problemas NP, incluso cuando estos aún no hayan sido descubiertos. Si P es diferente a NP, entonces existen problemas que no pueden ser resueltos en tiempo polinomial. Este problema, denominado como el problema de P *vs* NP, sigue abierto y su importancia es tal que el *Instituto Clay de Matemáticas* lo considera uno de los *problemas del milenio*, ofreciendo un millón de dólares a quien logre darle solución [52].

Los problemas NP-*Completos* son aquellos problemas que pertenecen a la clase NP y que además tienen la propiedad de que todos los problemas NP pueden ser *reducidos* en tiempo polinomial a ellos. La *reducción* de un problema A a un problema B consiste en definir un método, que conste de un número polinomial de pasos, que permita convertir una entrada a para el problema A en una entrada b para el problema B, de manera que el problema A pueda

ser resuelto a través de la resolución del problema B. Dado que todos los problemas de la clase NP pueden ser *reducidos* en tiempo polinomial a un problema NP-*Completo*, bastaría descubrir un algoritmo polinomial que resuelva un problema NP-*Completo* para demostrar que $P = NP$. Hasta ahora nadie ha logrado hacerlo.

Los problemas NP-*Difíciles* comparten una característica con los NP-*Completos*, la cual es que todos los problemas en NP se pueden *reducir* en tiempo polinomial a ellos. La diferencia de los problemas NP-*Difíciles*, con respecto a los NP-*Completos*, es que no necesariamente deben pertenecer a NP. Es por ello que los problemas NP-*Difíciles* suelen ser definidos como aquellos problemas que son al menos tan difíciles como los NP-*Completos*. De hecho, todos los problemas NP-*Completos* son problemas NP-*Difíciles*.

Dado que, en caso de que $P \neq NP$, no existe ningún algoritmo polinomial que permita resolver problemas NP-*Completos* y NP-*Difíciles*, se han propuesto diferentes enfoques para abordarlos. Uno de estos enfoques es el de los algoritmos de aproximación, los cuales entregan soluciones aproximadas (con un determinado factor de aproximación) y cuyo tiempo de ejecución es polinomial. Otro enfoque es el de las heurísticas y metaheurísticas, las cuales son algoritmos que, aunque no proveen garantías sobre la calidad de las soluciones entregadas, tienden a entregar soluciones cercanas a la óptima. El tiempo de ejecución de las heurísticas y metaheurísticas bien puede ser polinomial o no polinomial. En la presente tesis se propone un *algoritmo de aproximación aleatorizado*, por lo cual es importante dedicar un par de secciones al tema de los *algoritmos de aproximación* y los *algoritmos aleatorizados*.

Los algoritmos de aproximación tienen un tiempo de ejecución polinomial y encuentran soluciones que son una ρ-aproximación de la solución óptima, donde una ρ-aproximación es aquella solución cuyo tamaño es a lo más igual a ρ veces el tamaño de la solución óptima, en el caso de problemas de minimización, o bien al menos $1/\rho$ veces el tamaño de la solución óptima cuando el problema es de maximización. La definición formal de este tipo de algoritmos se presenta en el Capítulo 2. Los algoritmos aleatorizados son aquellos que toman decisiones de carácter aleatorio en algún punto de su ejecución y los hay de varios tipos, siendo los algoritmos aleatorizados tipo "Las Vegas" y "Monte Carlo" algunos de los más comunes. Los algoritmos tipo "Las Vegas" son aquellos que siempre encuentran la solución óptima, pero cuyo tiempo de ejecución es variable entre una ejecución y otra. Los algoritmos tipo "Monte Carlo" son aquellos cuyo tiempo de ejecución es polinomial, pero que entregan la solución óptima con una cierta probabilidad estrictamente menor a 1; es decir, no garantizan que la solución encontrada sea la correcta. En el Capítulo 2 se presenta una introducción a este tipo de algoritmos.

El problema abordado en la presente tesis es el problema conocido como problema de selección de k-centros (*k-center selection problem*). Este problema pertenece a la clase de complejidad NP-*Difícil* [2, 4, 19]. El único método conocido para resolver este problema de manera óptima es el de *fuerza bruta*, el cual consiste en probar todas las posibles soluciones y elegir la mejor. El método de fuerza bruta se ejecuta en tiempo exponencial, lo cual significa que no es un método eficiente. Los algoritmos de aproximación son una alternativa común para la resolución de problemas NP-*Difícil*es, pues son eficientes y las soluciones que encuentran no son arbitrariamente malas. Otro tipo de algoritmos utilizados para resolver problemas NP-*Difícil*es son los algoritmos aleatorizados, los cuales han demostrado ser muy útiles e incluso en ocasiones son los mejores conocidos para resolver determinados problemas, como por ejemplo el problema de la k *satisfacibilidad booleana* [30].

El problema de selección de k-centros consiste en, dado un grafo $G = (V, E)$ y un entero positivo k, determinar un subconjunto de *centros* $C \subseteq V$ de cardinalidad menor o igual a k, tal que la distancia máxima de los nodos hacia su centro más cercano sea mínima. A esta versión del problema se le conoce como versión *discreta*, siendo que en la versión *continua* cualquier punto del espacio puede ser considerado como un centro. Por ejemplo, para la instancia de la

Figura 1.1(a) la solución óptima al problema en su versión *discreta* es la de la Figura 1.1(b). Este problema tiene una amplia cantidad de aplicaciones, tales como la ubicación de centros de salud, hospitales, estaciones de bomberos, ubicación de antenas en redes inalámbricas, etcétera. Áreas de estudio como *Investigación de Operaciones* (*Operations Research*) y *Ubicación de Facilidades* (*Facility Location*) prestan particular interés a la resolución de este tipo de problemas.

1.1. Planteamiento del problema

Son muchas las versiones que existen del problema de selección de k-centros, las cuales dependen de las restricciones asociadas a cada variable involucrada. Por ejemplo, la versión *discreta* restringe el contenido de la solución a nodos del grafo de entrada, la versión *continua* permite que un nodo posicionado en cualquier punto del espacio forme parte de la solución, la versión *absoluta* restringe el contenido de la solución a nodos posicionados únicamente sobre cualquier punto de las aristas; a su vez, cada una de estas versiones puede corresponder a la versión *con peso* o *sin peso*, donde en la versión *sin peso* todos los nodos son tratados de la misma manera, mientras que en la versión *con peso* la distancia de todo nodo hacia cada centro es multiplicada por el costo asociado a cada nodo [10]. Otras tantas versiones existen, como aquellas relacionadas a las características de las aristas, tales como la versión *métrica* o *métrica asimétrica*. Las propiedades de un espacio métrico (V, d), donde V es un conjunto de puntos y $d : V \times V \rightarrow \mathbb{R}^+$, son las siguientes [53]:

- $\forall x \in V,\ d(x, x) = 0$
- (*simetría*) $\forall x, y \in V,\ d(x, y) = d(y, x)$
- (*desigualdad del triángulo*) $\forall x, y, z \in V,\ d(x, y) + d(y, z) \geq d(x, z)$

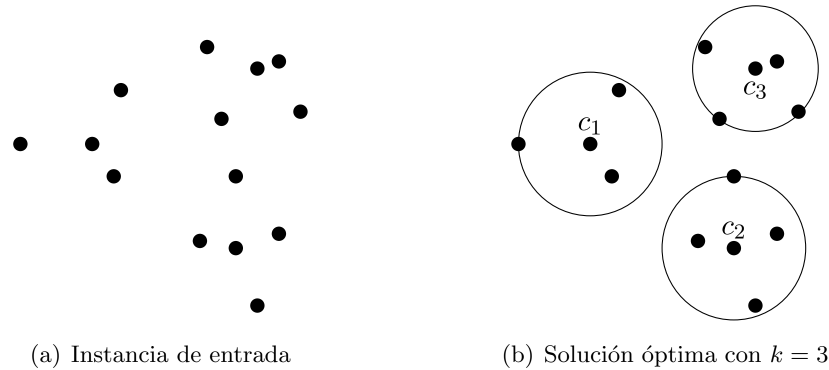
La versión que se aborda en la presente tesis es la más elemental de las mencionadas, es decir, el problema de selección de k-centros en su versión *discreta*, *métrica* y *sin peso*. Dado que se trata de la versión más elemental, es común que en la literatura se omitan estos adjetivos.

El problema de selección de k-centros en su versión *discreta*, *métrica* y *sin peso* (a partir de este punto se omitirán estos adjetivos) consiste en, dado un grafo $G = (V, E)$ y un entero k, encontrar un conjunto de centros $C \subseteq V$ de cardinalidad menor o igual a k, tal que la máxima distancia de todo nodo hacia su centro más cercano sea mínima, es decir, se desea minimizar la expresión:

$$max_{v \in V}\ min_{c \in C}\ distancia(v, c) \tag{1.1}$$

Al valor correspondiente a la expresión 1.1 se le suele llamar *radio de cobertura*, denotado por $r(C)$, pues representa la máxima distancia que existe de cualquier nodo hacia su centro más cercano. Véase la Figura 1.1.

Como se ha mencionado, el problema de selección de k-centros es *NP-Difícil*, por lo cual no existe un algoritmo que lo resuelva de manera óptima en tiempo polinomial (a menos que $P = NP$). Asimismo, encontrar soluciones C con un tamaño $r(C) < 2 \cdot r(C^*)$ (donde C^* es la solución óptima) también es un problema *NP-Difícil*. Es decir, encontrar soluciones que sean una ρ-aproximación de la solución óptima, donde $\rho < 2$, es tan difícil como encontrar la solución óptima [2, 4, 19].



(a) Instancia de entrada (b) Solución óptima con $k = 3$

Figura 1.1: Problema de selección de k-centros

Son varias las técnicas utilizadas por los algoritmos que se han diseñado para resolver el problema de selección de k-centros. Los mejores algoritmos conocidos encuentran una 2-aproximación $[1, 4, 9, 17, 34]$. Además de este tipo de algoritmos, se han diseñado diferentes tipos de metaheurísticas que tienden a entregar soluciones cercanas a la óptima: Búsqueda Tabú [29], Búsqueda de Entorno Variable (*Variable Neighborhood Search*) [29] y Búsqueda Dispersa (*Scatter Search*) [26]. En general, estas metaheurísticas tienen un *buen* desempeño y encuentran soluciones cercanas a la óptima; sin embargo, no garantizan tener un *buen* desempeño sobre instancias arbitrarias. Todos estos algoritmos son de tipo determinista.

Los algoritmos deterministas cuentan con la desventaja de que suelen quedar atrapados en mínimos locales, de los cuales no pueden escapar. En este contexto, otro enfoque interesante para el diseño de algoritmos es el de los algoritmos aleatorizados, los cuales entregan soluciones óptimas en un tiempo de ejecución variable (algoritmos tipo *Las Vegas*) o bien entregan una solución óptima con cierta probabilidad estrictamente menor a 1 (algoritmos tipo *Monte Carlo*). Los algoritmos aleatorizados han demostrado ser muy útiles para resolver gran cantidad de problemas (incluyendo problemas *NP-Difíciles*).

Por lo anterior, se propone desarrollar una nueva técnica de diseño de algoritmos de optimización combinatoria que combine las ventajas de los algoritmos de aproximación deterministas y de los algoritmos aleatorizados tipo Monte Carlo. El objetivo es proveer un algoritmo para el problema de selección de k-centros que, con alta probabilidad, entregue soluciones ρ-aproximadas, pero que al mismo tiempo tenga la capacidad de analizar el vecindario de dichas soluciones para así escapar de máximos o mínimos locales.

1.2. Objetivos

1.2.1. Objetivo general

Diseñar, analizar y caracterizar un algoritmo aleatorizado que resuelva de manera aproximada el problema de selección de k-centros en tiempo polinomial.

1.2.2. Objetivos particulares

1. Realizar un estudio detallado de los diferentes algoritmos de aproximación que se han propuesto en la literatura para resolver el problema de selección de k-centros.

2. Demostrar matemáticamente que el algoritmo propuesto posee complejidad polinomial.

3. Demostrar matemáticamente que el algoritmo propuesto entrega una 2-aproximación con probabilidad estrictamente mayor a cero.

4. Implementar y caracterizar experimentalmente el desempeño del algoritmo propuesto, comparándolo con algoritmos del estado del arte con base en un conjunto estandarizado de instancias de entrada.

1.3. Justificación

El problema de selección de k-centros es muy interesante tanto desde el punto de vista teórico como desde el punto de vista práctico. Desde la perspectiva teórica resulta interesante que los dos primeros algoritmos de aproximación desarrollados para resolver este problema (alrededor de 1985) demostraron ser los mejores posibles, a menos que $P=NP$, pues ambos entregan una 2-aproximación [1,4]. Los algoritmos desarrollados posteriormente fueron diseñados bajo distintos enfoques (algoritmos voraces, búsqueda local, metaheurísticas, programación lineal, etcétera). Uno de los algoritmos polinomiales con mejor desempeño, tanto en tiempo de ejecución como en calidad de las soluciones entregadas, es justamente uno de los primeros algoritmos de aproximación que fue diseñado: el algoritmo de González [1].

Desde un punto de vista práctico resulta evidente la importancia del problema de selección de k-centros; incluso existe un área de estudio llamada *Ubicación de Facilidades* (*Facility Location*), donde se plantean y estudian problemas de selección de centros como el *k-median*, *k-center* (o problema de selección de k-centros) y sus variantes [10,23]. Los resultados reportados por esta área de estudio son de particular interés para la disciplina de la *Investigación de Operaciones* (*Operations Research*), la cual básicamente se encarga de modelar teóricamente sistemas reales para mejorar su funcionamiento [54]. En la sección 2.4.2 se enlistan algunos problemas reales que pueden ser modelados con el problema de selección de k-centros.

Como ya se ha mencionado, el problema planteado ha sido abordado desde diferentes enfoques; sin embargo, no parece haber habido intentos de utilizar la aleatorización como herramienta principal para el diseño de algoritmos que resuelvan el problema en cuestión. Los algoritmos aleatorizados son aquellos que entregan soluciones óptimas con cierta probabilidad (es decir, no es posible garantizar al 100 % que la solución entregada es la óptima) o bien entregan la solución óptima en una cantidad de tiempo variable. Este tipo de algoritmos han demostrado ser muy eficientes e incluso pueden superar a otros algoritmos de tipo determinista. Los algoritmos deterministas son aquellos que no son aleatorios; es decir, son aquellos que tienden a entregar *"siempre la misma solución"* en *"la misma cantidad de tiempo"*. Es por ello que la idea de mezclar ambos enfoques (*aleatorización* de un *algoritmo de aproximación*), con la intención de así lograr un mayor acercamiento a la solución óptima, surge de manera natural, pues la aleatoriedad brinda la posibilidad de escapar de los mínimos locales en que los algoritmos deterministas suelen quedar atrapados. El algoritmo de aproximación utilizado como base para el diseño del *algoritmo propuesto* es el algoritmo de González, el cual, dada su eficiencia y la calidad de las soluciones entregadas, sigue siendo considerado como una buena alternativa.

Cabe señalar que el algoritmo de González es simple, siendo también la simplicidad una característica propia de los algoritmos aleatorizados. Por lo tanto, es de esperar que el *algoritmo propuesto* sea simple y fácil de implementar, pero sin dejar de ser una buena alternativa para resolver el problema planteado.

1.4. Organización de la tesis

Capítulo 1 : Introducción. Se presentan los objetivos, la justificación y el planteamiento del problema.

Capítulo 2: Marco Teórico. Se presenta una introducción a las herramientas del estado del arte que permitieron el desarrollo del *algoritmo propuesto*. El *algoritmo propuesto* es un *algoritmo de aproximación aleatorizado* para el problema de selección de k-centros, el cual es un problema *NP-Difícil*. Por lo tanto, resulta importante tratar los siguientes temas: *intratabilidad computacional*, *algoritmos de aproximación*, *algoritmos aleatorizados* y evidentemente el *problema de selección de k-centros*.

Capítulo 3: Trabajos relacionados. Se describen algunos de los principales algoritmos del estado del arte diseñados para la resolución del problema de selección de k-centros. Este capítulo está dividido en tres partes: *algoritmos de aproximación*, *heurísticas y metaheurísticas* y *metaheurísticas del estado del arte similares a la técnica propuesta*. En la primera parte se describen los algoritmos de aproximación más relevantes, en la segunda sección se describen algunas de las heurísticas y metaheurísticas más eficientes y/o eficaces, en la tercera sección se describe un par de metaheurísticas que presentan similitudes con la técnica de diseño utilizada por el *algoritmo propuesto*.

Capítulo 4: Propuesta. Se presenta el *algoritmo propuesto* y una variante del mismo denominada *algoritmo propuesto con memoria*. Asimismo, se describe la técnica utilizada para el diseño del *algoritmo propuesto*. Se demuestra matemáticamente que el *algoritmo propuesto* encuentra soluciones aproximadas con una determinada probabilidad mayor a cero para instancias arbitrarias.

Capítulo 5: Resultados. Tanto el *algoritmo propuesto*, incluyendo sus variantes, como los algoritmos del estado del arte, son ejecutados sobre un conjunto estandarizado de instancias de entrada. Con base en los resultados obtenidos se demuestra experimentalmente la eficacia del *algoritmo propuesto* e incluso su superioridad sobre la mayoría de los algoritmos del estado del arte.

Capítulo 6: Conclusiones. Se muestra un pequeño resumen de los elementos y conclusiones más destacables del presente trabajo

Capítulo 2

Marco teórico

En este capítulo se introducen los fundamentos teóricos de la presente tesis. En particular, se desarrollan los conceptos de *clases de complejidad*, así como de *algoritmo de aproximación* y *algoritmo aleatorizado*.

En la Sección 2.1 se describen las principales clases de complejidad en las que se clasifican los diversos problemas computacionales. Definir estas clases es importante, pues el problema de selección de k-centros pertenece a la clase NP-*Difícil*, por lo cual es necesario conocer las propiedades que debe poseer un problema para pertenecer a una u otra clase. De igual manera, dado que la solución propuesta está basada en la aleatorización de un algoritmo de aproximación, en la Sección 2.2 se introduce la teoría relacionada con los algoritmos de aproximación, mientras que en la Sección 2.3 se hace lo propio con la teoría de los algoritmos aleatorizados. Finalmente, en la Sección 2.4 se presentan las principales propiedades y aplicaciones del problema de selección de k-centros.

2.1. Intratabilidad computacional

La palabra **problema** puede ser definida de muy distintas maneras. En las ciencias de la computación suele manejarse la siguiente definición:

*Un **problema** es una pregunta general cuya respuesta deseamos conocer. Dicha pregunta suele tener asociados una serie de parámetros cuyos valores no están especificados.* [11]

Para describir un problema es necesario especificar el siguiente par de datos:

1. Una descripción general de todos los parámetros del problema.

2. Un enunciado que defina las propiedades con que debe contar la respuesta.

Otro término importante que es importante definir es el de **instancia** de un problema. Una **instancia** de un problema es el resultado de especificar valores particulares para cada parámetro del problema planteado.

En términos informales, un **algoritmo** es un procedimiento general para la resolución de un problema en particular [11]. Una de las características de los algoritmos es el tiempo de ejecución; siendo en general más deseables los algoritmos cuyo tiempo de ejecución es eficiente.

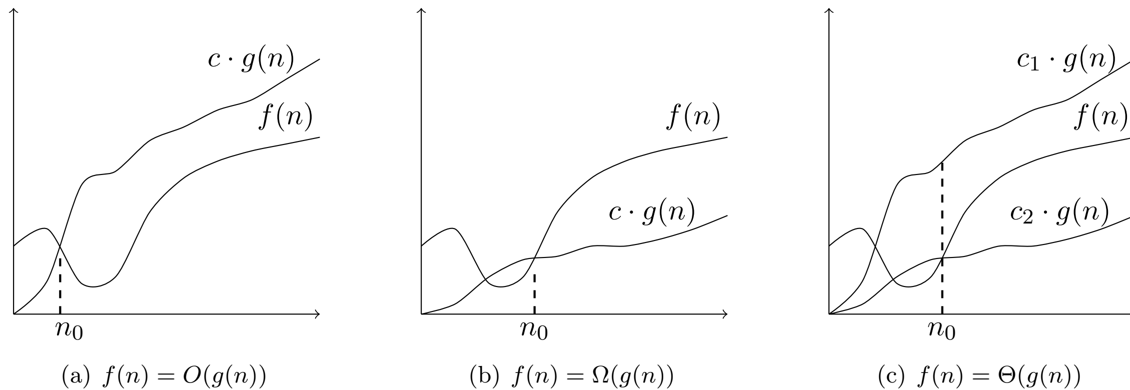
Los requerimientos de tiempo de un algoritmo se expresan en términos del tamaño de la instancia del problema, el cual pretende reflejar la cantidad de datos necesarios para describir a dicha instancia. La eficiencia de un algoritmo se expresa con una función de complejidad temporal, la cual es una función del tamaño de la instancia de entrada.

Para denotar que una función cuenta con una cota asintótica superior se utiliza la notación O (O mayúscula). Se dice que una función $f(n)$ es $O(g(n))$ siempre y cuando exista una

constante c tal que $f(n) \leq c \cdot g(n)$ para todos los valores de $n \geq n_0$, donde n_0 es el valor mínimo de n para el cual $f(n) \leq c \cdot g(n)$ (véase Figura 2.1a).

Para denotar que una función cuenta con una cota asintótica inferior se utiliza la notación Ω. Se dice que una función $f(n)$ es $\Omega(g(n))$ si existe una constante c tal que $f(n) \geq c \cdot g(n)$ para todos los valores de $n \geq n_0$, donde n_0 es el valor mínimo de n para el cual $f(n) \geq c \cdot g(n)$ (véase Figura 2.1b).

La notación Θ es una combinación de las notaciones O y Ω. Se dice que una función $f(n)$ es $\Theta(g(n))$ si existen un par de constantes c_1 y c_2 tales que $c_2 \cdot g(n) \leq f(n) \leq c_1 \cdot g(n)$ para todos los valores de $n \geq n_0$, donde n_0 es el valor mínimo de n para el cual $c_2 \cdot g(n) \leq f(n) \leq c_1 \cdot g(n)$ (véase Figura 2.1c).



(a) $f(n) = O(g(n))$ (b) $f(n) = \Omega(g(n))$ (c) $f(n) = \Theta(g(n))$

Figura 2.1: Ejemplos gráficos de las notaciones O, Θ y Ω.

Un algoritmo de tiempo polinomial se define como aquel cuya función de complejidad temporal es de la forma $O(p(n))$ para alguna función polinomial p, donde n representa el tamaño de la entrada. Los algoritmos que no pueden ser acotados por una función polinomial son denominados algoritmos de tiempo exponencial (esta definición incluye funciones de complejidad no polinomial, como $n^{\log n}$, que no son comúnmente consideradas como exponenciales).

Los algoritmos más eficientes suelen ser aquellos cuya función de complejidad temporal es un polinomio, como se aprecia en la Tabla 2.1.

Resulta claro que los algoritmos con complejidad temporal polinomial (o simplemente algoritmos polinomiales) resuelven de manera más eficiente los problemas para los que fueron diseñados. Lamentablemente no todos los problemas cuentan con algoritmos polinomiales; ya sea porque no han sido descubiertos o bien porque no existen. Sea cual sea el caso, se dice que un problema es intratable cuando no es conocido un solo algoritmo polinomial que lo resuelva.

2.1.1. Clases P y NP

Todo problema computacional es susceptible de ser clasificado dentro de alguna de las clases de complejidad que han sido definidas dentro de las ciencias de la computación. Entre las primeras clases de complejidad definidas se encuentran las clases P y NP.

La definición formal de las clases P y NP se basa en la definición de la *máquina de Turing*. Una máquina de Turing consta de una cinta de escritura de longitud infinita, que hace las veces de memoria, y de una *cabeza* que puede leer y escribir sobre la cinta y que además se puede desplazar hacia cualquier punto de ésta (Figura 2.2).

Inicialmente la cinta contiene únicamente una cadena de entrada, la cual es la representación de la instancia del problema de entrada, y el resto de la cinta se rellena con espacios en blanco. Si la máquina necesita guardar información, puede hacerlo sobre la cinta. Para leer

Tabla 2.1: Comparación de la eficiencia entre funciones polinomiales y exponenciales.

complejidad de la función	tamaño de n 10	20	30	40	50	60
n	.00001 segundos	.00002 segundos	.00003 segundos	.00004 segundos	.00005 segundos	.00006 segundos
n^2	.0001 segundos	.0004 segundos	.0009 segundos	.0016 segundos	.0025 segundos	.0036 segundos
n^3	3.001 segundos	.008 segundos	.027 segundos	.064 segundos	.125 segundos	.216 segundos
n^5	5.1 segundos	3.2 segundos	24.3 segundos	1.7 minutos	5.2 minutos	13 minutos
2^n	.001 segundos	1.0 segundos	17.9 minutos	12.7 días	35.7 años	366 siglos
3^n	0.59 segundos	58 minutos	6.5 años	3855 siglos	2×10^8 siglos	1.3×10^{13} siglos

y escribir información, la *cabeza* puede desplazarse. La máquina se detiene cuando llega a un estado de *aceptación* o *rechazo*, similar a como lo hace un autómata finito [5].

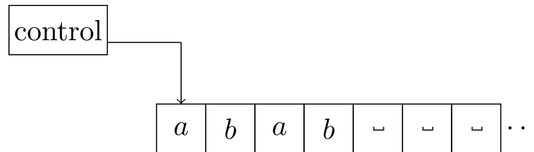


Figura 2.2: Máquina de Turing

Parte primordial de la definición formal de una máquina de Turing es la función de transición δ, ya que ésta es la encargada de determinar el comportamiento de la máquina en cada uno de sus pasos. Para una máquina de Turing, la función δ tiene la forma:

$$Q \times \Gamma \longrightarrow Q \times \Gamma \times \{L, R\} \tag{2.1}$$

De manera que si la máquina se encuentra en un determinado estado $q \in Q$, la cabeza se ecuentra ubicada sobre un espacio de la cinta que contiene el símbolo $a \in \Gamma$, donde Γ representa el alfabeto de la cinta, es decir, los símbolos que pueden ser escritos sobre la cinta, y $\delta(q, a) = (r, b, L)$, entonces la máquina reemplazará el símbolo a con el símbolo b y pasará al estado r. El tercer componente (L o R) indica la dirección hacia la que ha de desplazarse la *cabeza* despúes de realizar la lectura o escritura (L por *left* = izquierda, R por *right* = derecha).

Definición 1. *Una **máquina de Turing** es una 7-tupla,* $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$*, donde* Q*,* Σ*,* Γ *son conjuntos finitos y:*

1. Q *es el conjunto de estados,*
2. Σ *es el alfabeto de entrada, el cual no contiene al símbolo* $\sqcup$ *(espacio en blanco),*
3. Γ *es el alfabeto de la cinta, donde* $\sqcup \in \Gamma$ *y* $\Sigma \subseteq \Gamma$*,*
4. $\delta : Q \times \Gamma \longrightarrow Q \times \Gamma \times \{L, R\}$ *es la función de transición,*
5. $q_0 \in Q$ *es el estado inicial,*

6. $q_{accept} \in Q$ es el estado de ***aceptación****, y*

7. $q_{reject} \in Q$ es el estado de ***rechazo****, donde $q_{accept} \neq q_{reject}$.*

La función de transición de una máquina de Turing puede definirse no sólo como se indica en la ecuación 2.1, sino también de la siguiente manera:

$$Q \times \Gamma \longrightarrow \mathcal{P}(Q \times \Gamma \times \{L, R\}) \tag{2.2}$$

A las máquinas de Turing con función de transición de la forma 2.1 se les denomina *máquinas de Turing deterministas*, mientras que a aquellas con función de transición de la forma 2.2 se les denomina **máquinas de Turing no deterministas**. La diferencia entre este par de máquinas radica en que el potencial de las no deterministas es mayor, pues son capaces de ejecutar varias *posibilidades* (de ahí el uso del simbolo $\mathcal{P}$ en 2.2) de funciones de transición a la vez en lugar de una sola, el cual es el caso de las máquinas deterministas.

La entrada de una máquina de Turing (ya sea determinista o no determinista) es una cadena de símbolos $x \in \Sigma^*$, donde Σ^* representa al conjunto de cadenas que pueden formarse con los elementos de Σ, que es escrita sobre la cinta de la máquina. Si, al poner en funcionamiento la máquina de Turing, ésta se detiene en un estado aceptor (o de aceptación), se dice que la máquina ha *reconocido* a la cadena x. De modo que el lenguaje que es reconocido por una máquina de Turing se define como:

$$L_M = \{x \in \Sigma^* : M \text{ acepta a } x\} \tag{2.3}$$

El hecho de que una máquina de Turing pueda *reconocer* un lenguaje le permite *resolver* problemas de decisión. Se dice que una máquina de Turing *resuelve* un problema de decisión Π, bajo un esquema de codificación e, si dicha máquina termina, es decir, si llega a un estado de *aceptación* o de *rechazo* para todas la cadenas posibles sobre su alfabeto de entrada y si el lenguaje reconocido por esta máquina es igual a $\mathrm{L}[\Pi, e]$.

Con base en la definición de la máquina de Turing es posible definir de una manera más formal el concepto de complejidad temporal. El *tiempo* utilizado por una máquina de Turing determinista M, sobre una cadena x, es el número de pasos que le toma llegar al estado de *aceptación* o *rechazo*. Una máquina de Turing determinista que termina para todas las entradas $x \in \Sigma^*$, tiene una *función de complejidad temporal*, $T_M : Z^+ \to Z^+$, dada por:

$$T_M(n) = max \left\{ m : \begin{array}{l} \text{existe una } x \in \Sigma^*, \text{ con } |x| = n, \text{ tal que la computación} \\ \text{de } M \text{ sobre la entrada } x \text{ toma una cantidad de tiempo} \\ \text{igual a } m \end{array} \right\} \tag{2.4}$$

Si existe un polinomio p tal que, para todo $n \in \mathbb{Z}^+$, $T_M(n) \leq p(n)$, se dice que la máquina de Turing determinista M es una máquina de tiempo polinomial. La clase de lenguajes P se define de la siguiente manera:

$$P = \left\{ L : \begin{array}{l} \text{existe una máquina de Turing determinista} \\ \text{de tiempo polinomial } M \text{ para la cual } L = L_M \end{array} \right\} \tag{2.5}$$

Todo problema de decisión Π puede ser convertido a un lenguaje $L[\Pi, e]$, donde e es un esquema de codificación. Es por ello que es común referirse a las clases de complejidad como conjuntos de problemas y no de lenguajes; sin embargo, por la relación expuesta, en ambos casos se está hablando de manera correcta. El esquema de codificación empleado no es relevante, pues la relación entre los diferentes esquemas de codificación es polinomial.

La definición de la clase NP depende de la definición de la máquina de Turing no determinista, la cual es más poderosa que la versión determinista, pues puede realizar varias operaciones o cálculos a la vez. Se dice que una máquina de Turing no determinista M reconoce una cadena de entrada $x \in \Sigma^*$ si alguna de los tantos cómputos que puede realizar termina en un estado de *aceptación*. Por lo tanto, el lenguaje que M reconoce es:

$$L_M = \{x \in \Sigma^* : M \text{ acepta a } x\} \tag{2.6}$$

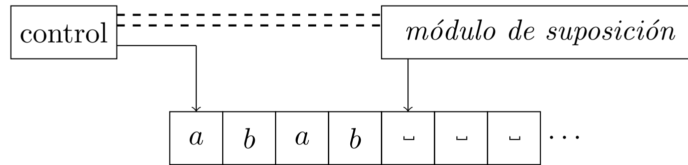


Figura 2.3: Máquina de Turing no determinista con módulo de suposición

El funcionamiento de la máquina de Turing no determinista con *módulo de suposición* M consta de dos etapas: la etapa de *suposición* y la etapa de *verificación*. Inicialmente se escribe la cadena de entrada x en la cinta. En la etapa de *suposición* el módulo de control se encuentra "inactivo" y el *módulo de suposición* escribe sobre los espacios en blanco de la cinta una cadena $\in \Sigma^*$. En la etapa de de *verificación* se "desactiva" el *módulo de suposición* y la máquina continúa funcionando de la misma forma en que lo haría una máquina de Turing determinista. Sin embargo, dado que la máquina M es no determinista, es capaz de realizar una infinidad de veces el par de etapas descrito, todas al mismo tiempo. Para cada una de estas ejecuciones el *módulo de suposición* genera una cadena diferente. Se dice que la máquina M acepta la cadena x si al menos una de estas ejecuciones termina en un estado de *aceptación*. El lenguaje reconocido por M es:

$$L_M = \{x \in \Sigma^* : M \text{ acepta a } x\} \tag{2.7}$$

El tiempo requerido por una máquina de Turing no determinista M para aceptar una cadena $x \in L_M$ se define como el tiempo requerido por la computación más rápida, de entre todas las computaciones realizadas por M sobre la entrada x que terminan en un estado de *aceptación*. La función de complejidad temporal $T_M : Z^+ \rightarrow Z^+$ de M es:

$$T_M(n) = max\left(\{1\} \cup \left\{ m : \begin{array}{l} \text{existe una } x \in L_M \text{ con } |x| = n \text{ tal que el tiempo} \\ \text{que le toma a } M \text{ aceptar a } x \text{ es igual a } m \end{array} \right\}\right) \tag{2.8}$$

Se puede observar que la función de complejidad temporal de M depende únicamente del número de pasos ejecutados por las computaciones que terminan en un estado de *aceptación*; es por ello que, por convención, $T_M(n)$ es igual a 1 cuando ninguna cadena de entrada x, $|x| = n$, es aceptada por M.

Si existe un polinomio p tal que, para todo $n \in \mathbb{Z}^+$, $T_M(n) \leq p(n)$, se dice que la máquina de Turing no determinista M es una máquina de tiempo polinomial. La clase de lenguajes NP se define de la siguiente manera:

$$NP = \left\{ L : \begin{array}{l} \text{existe una máquina de Turing no determinista} \\ \text{de tiempo polinomial } M \text{ para la cual } L = L_M \end{array} \right\} \tag{2.9}$$

El *módulo de suposición* de la máquina de Turing no determinista funciona de manera totalmente independiente a la entrada x, pues toda cadena elemento de Σ^* es una posible *suposición*. Sin embargo, es posible modificar la máquina de Turing de tal manera que, previo a la etapa de *verificación*, exista una etapa en la que se determine si la cadena generada por el *módulo de suposición* es apropiada para la entrada x. Si la cadena generada no es apropiada entonces la máquina entra directamente a un estado de *rechazo*.

Resulta claro que el modelo de máquina de Turing no determista no es realista, pues no es posible realizar una infinidad de computaciones a la vez. Sin embargo, si sustituimos el *módulo de suposición* por una entrada definida por el usuario, ésta terminará siempre en un estado de *aceptación* o de *rechazo*. Es por esto que la clase de complejidad NP suele ser descrita como aquella que contiene problemas (o lenguajes $L[\Pi, e]$, donde Π es un problema y e un esquema de codificación) tales que, dado un *certificado*, que corresponde a la cadena generada por el *módulo de suposición*, y una instancia de entrada, que corresponde a la cadena $x \in \Sigma^*$ que se escribe inicialmente en la cinta, es posible determinar en tiempo polinomial si el *certificado* es una solución al problema Π (lo cual equivale a terminar en un estado de *aceptación*). A la máquina de Turing que recibe como entrada tanto una instancia de un problema como un certificado se le denomina *verificador*.

Todo problema de decisión Π puede ser convertido a un lenguaje $L[\Pi, e]$, donde e es un esquema de codificación. Es por ello que es común referirse a las clases de complejidad como conjuntos de problemas y no de lenguajes; sin embargo, por la relación expuesta, en ambos casos se está hablando de manera correcta. El esquema de codificación empleado no es relevante, pues la relación entre los diferentes esquemas de codificación es polinomial.

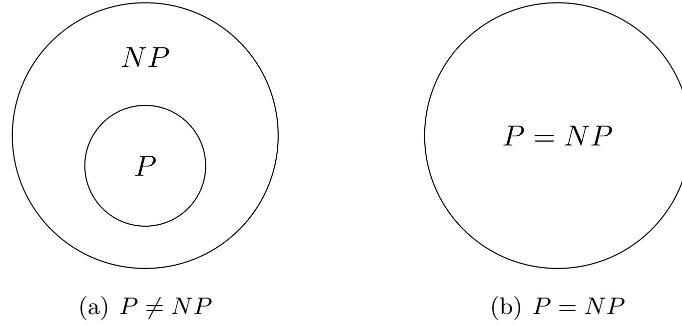
El contenido de esta sección se puede reducir a lo siguiente. Se dice que un problema pertenece a la clase P si es posible resolverlo en tiempo polinomial con una máquina de Turing determinista. La definición de la clase NP señala que un problema pertenece a la clase NP si existe un *verificador* de tiempo polinomial que lo resuelva. Las siglas NP corresponden a las palabras *nondeterministic polynomial time*, ya que los problemas de la clase NP pueden ser resueltos en tiempo polinomial por una máquina de Turing no determinista. Dado que las máquinas de Turing no deterministas tienen una mayor capacidad de cálculo que las deterministas, resulta claro que todo problema que pueda ser resuelto por una máquina de Turing determinista también podrá ser resuelto por una no determinista, lo cual implica que todo problema perteneciente a la clase P también pertenece a la clase NP. Sin embargo a la fecha no ha sido posible demostrar si $P = NP$ o bien si $P \neq NP$ (veáse la Figura 2.4). Dada la falta de progreso sobre esta cuestión (aunque en realidad todo parece apuntar a que $P \neq NP$), los investigadores del tema se han planteado otras preguntas como: ¿Cuáles son los problemas NP más difíciles? [11] La respuesta a esta pregunta ha llevado al desarrollo de la teoría de la *NP-Completitud*, que es tratada en la siguiente sección.

Las definiciones para las clases P y NP se plantearon en términos de máquinas de Turing; sin embargo dichas definiciones no dependen en realidad del modelo de computadora utilizado, siempre y cuando el modelo elegido varíe polinomialmente con respecto al de una máquina de Turing.

2.1.2. Clase NP-Completo

"Un problema NP-Completo es aquel que es tan difícil como un gran número de otros problemas que han confundido a los expertos durante años y que no se han podido resolver (eficientemente)" [11]. Esta es una definición muy intuitiva, pero que resume de una manera muy general a la teoría de la *NP-Completitud*.

Si la clase P no es igual a la clase NP, es decir, si $P \neq NP$, entonces la diferencia entre



(a) $P \neq NP$ (b) $P = NP$

Figura 2.4: Posible relación entre las clases P y NP

la clase P y la clase $NP - P$ es significativa, pues implicaría que todos los problemas en P pueden ser resueltos con algoritmos polinomiales, mientras que los problemas en $NP - P$ serían intratables. Por lo tanto, dado un problema Π, si $P \neq NP$, es deseable saber si $\Pi \in P$ o $\Pi \in NP - P$. Sin embargo, mientras no se demuestre que $P \neq NP$, existen pocas posibilidades de demostrar que un problema dado pertenezca a $NP - P$. Es por este motivo que la teoría de la *NP-Completitud* se enfoca en demostraciones más débiles, de la forma: "Si $P \neq NP$, entonces $\Pi \in NP - P$". La idea básica de este enfoque condicional es la de la *reducción polinomial* (o *transformación polinomial*).

Una *transformación polinomial* (*reducción polinomial*) de un lenguaje $L_1 \in \Sigma_1^*$ a un lenguaje $L_2 \in \Sigma_2^*$ es una función $f : \Sigma_1^* \rightarrow \Sigma_2^*$ que satisface las siguientes condiciones:

1. Existe una máquina de Turing determinista que computa la función f.

2. Para toda $x \in \Sigma_1^*$, $x \in L_1$ si y solo si $f(x) \in L_2$.

Para señalar que existe una reducción polinomial de L_1 a L_2 se utiliza la notación $L_1 \propto L_2$ o bien $L_1 \leq_p L_2$, que se lee "L_1 se transforma (o reduce) a L_2" (se suele omitir el adjetivo *polinomialmente*).

El Lema 2.1.1 hace evidente la importancia de la *reducción polinomial*:

Lema 2.1.1. *Si $L_1 \propto L_2$, entonces $L_2 \in P$ implica que $L_1 \in P$. De manera equivalente $L_1 \notin P$ implica $L_2 \notin P$.*

Demostración. Sean Σ_1 y Σ_2 los alfabetos de los lenguajes L_1 y L_2 respectivamente. Sea $f : \Sigma_1 \rightarrow \Sigma_2$ una transformación polinomial de L_1 a L_2. Sea M_j una máquina de Turing determinista que computa la función f, y sea M_2 una máquina de Turing determinista que reconoce a L_2. Se puede construir una máquina de Turing determinista que reconozca a L_1 al combinar las máquinas M_j y M_2: para una entrada $x \in \Sigma_1^*$, se permite que M_j genere $f(x) \in \Sigma_2^*$ y posteriormente se permite que M_2 determine si $f(x) \in L_2$. Dado que $x \in L_1$ si y solo si $x \in L_2$, esta nueva máquina reconoce a L_1. El tiempo de ejecución es polinomial, pues el tiempo de ejecución de M_j y M_2 es polinomial. De manera más específica, si p_j y p_2 son funciones polinomiales que acotan el tiempo de ejecución de M_j y M_2, entonces $|f(x)| \leq p_j(|x|)$ y el tiempo de ejecución de la máquina de Turing construida es $O(p_f(|x|) + p_2(p_f(|x|)))$, que está acotado por un polinomio de $|x|$. □

Si Π_1 y Π_2 son problemas de decisión, con esquemas de codificación e_1 y e_2 asociados, se escribirá $\Pi_1 \leq_p \Pi_2$ (o $\Pi_1 \propto \Pi_2$) cuando exista una reducción polinomial de $L[\Pi_1, e_1]$ a $L[\Pi_2, e_2]$.

Lema 2.1.2. *Si $L_1 \propto L_2$ y $L_2 \propto L_3$, entonces $L_1 \propto L_3$.*

Demostración. Sean Σ_1, Σ_2 y Σ_3 los alfabetos de los lenguajes L_1, L_2 y L_3, respectivamente. Sea $f_1 : \Sigma_1^* \rightarrow \Sigma_2^*$ una reducción polinomial de L_1 a L_2 y sea $f_2 : \Sigma_2^* \rightarrow \Sigma_3^*$ una reducción polinomial de L_2 a L_3. La función $f : \Sigma_1^* \rightarrow \Sigma_2^*$ definida por $f(x) = f_2(f_1(x))$ para toda $x \in \Sigma_1^*$ representa justamente a la reducción deseada de L_1 a L_3. Claramente, $f(x) \in L_3$ si y solo si $x \in L_1$. La función f puede ser computada en tiempo polinomial por una máquina de Turing determinista por el hecho de que asimismo lo son las máquinas correspondientes a las funciones f_1, f_2 y a las máquinas que reconocen a L_2 y L_3. □

Se dice que dos lenguajes L_1 y L_2 (o dos problemas de decisión Π_1 y Π_2) son *polinomialmente equivalentes* si $L_1 \propto L_2$ y $L_2 \propto L_1$ ($\Pi_1 \propto \Pi_2$ y $\Pi_2 \propto \Pi_1$). La relación "$\propto$" impone un orden parcial en las clases resultantes de equivalencia de lenguajes. De hecho, la clase P conforma a la clase "mínima" de este orden parcial, siendo considerada como la clase que contiene a los lenguajes (problemas de decisión) más "fáciles". La clase de los lenguajes (problemas) *NP-Completos* conforma otra clase de equivalencia, la cual se distingue por contener a los lenguajes (problemas) más "difíciles" de NP.

Formalmente, un lenguaje L es *NP-Completo* si $L \in NP$ y, para todos los lenguajes $L' \in NP$, $L' \propto L$. En términos informales, un problema de decisión Π es *NP-Completo* si $\Pi \in NP$ y, para todos los problemas Π', $\Pi' \propto \Pi$. Si un problema *NP-Completo* es resuelto en tiempo polinomial, entonces todos los problemas en NP pueden ser resueltos en tiempo polinomial. Si un problema en NP es intratable, entonces también lo son los problemas *NP-Completos*. Por lo tanto, todo problema *NP-Completo* Π tiene la siguiente característica: si $P \neq NP$, entonces $\Pi \in NP - P$. De manera más precisa, $\Pi \in P$ si y solo si $P = NP$.

Asumiendo que $P \neq NP$, el *panorama* de la clase NP corresponde al de la Figura 2.5.

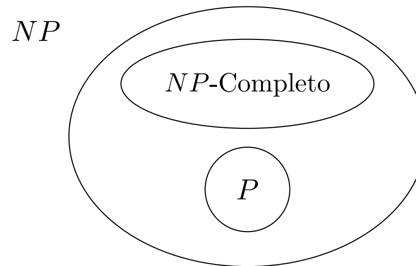


Figura 2.5: Clase NP (si $P \neq NP$)

Dado que los problemas que pertenecen a la clase *NP-Completo* son los más "difíciles" de la clase NP, resulta conveniente saber identificarlos. El siguiente Lema brinda una idea simple acerca de cómo identificar dichos problemas.

Lema 2.1.3. *Si L_1 y L_2 pertencen a NP, L_1 es NP-Completo y $L_1 \propto L_2$, entonces L_2 es NP-Completo.*

Demostración. Dado que $L_2 \in NP$, lo único que es necesario demostrar es que, para todo $L' \in NP$, $L' \propto L_2$. Sea algún $L' \in NP$. Dado que L_1 es *NP-Completo*, entonces se cumple que $L' \propto L_1$. Utilizando la propiedad de transitividad de "$\propto$" y el hecho de que $L_1 \propto L_2$, queda claro que $L' \propto L_2$. □

En términos de problemas de decisión, el lema anterior define un método para identificar a los problemas de la clase *NP-Completo*, para aplicar el cual es necesario conocer al menos un problema *NP-Completo*. Es decir, para que un problema Π sea *NP-Completo* es necesario que se cumplan las siguientes condiciones:

- $\Pi \in NP$ y
- existe una transformación (polinomial) de un problema *NP-Completo* hacia Π.

Para utilizar este método es necesario conocer un problema *NP-Completo*. El primer problema *NP-Completo* es provisto por el teorema fundamental de Cook, en el cual el autor demuestra que el problema de la *satisfacibilidad booleana* es *NP-Completo* [11].

El problema de la *satisfacibilidad booleana* se define de la siguiente manera: sea $U = \{u_1, u_2, ..., u_m\}$ un conjunto de variables booleanas. Una asignación *verdadera* para U es una función $t : U \to \{T, F\}$. Si $t(U) = T$, se dice que u es *verdadera* al ser sometida a t; si $t(u) = F$, se dice que u es *falsa* sometida a t. Si u es una variable en U, entonces u y $\bar{u}$ son literales sobre U. La literal u es verdadera bajo t si y solo si la variable u es verdadera bajo t; la literal $\bar{u}$ es verdadera si y solo si la variable u es falsa. Una *cláusula* sobre U es un conjunto de literales sobre U, tal como $\{u_1, \bar{u_3}, u_8\}$, que representa la disyunción de las literales y se *satisface* con una asignación verdadera si y solo si al menos uno de sus miembros es verdadero bajo esa asignación. La cláusula anterior será satisfecha por t, a menos que $t(u_1) = F$, $t(u_3) = T$ y $t(u_8) = F$. Una colección C de cláusulas bajo U es *satisfacible* si y solo si existe alguna *asignación de verdad satisfactoria* para C. El problema de la *satisfacibilidad booleana* consiste en, dado un conjunto de variables U y una colección de cláusulas C sobre U, determinar si existe alguna asignación de verdad satisfactoria para C.

Teorema 2.1.4. *(Teorema de Cook) El problema de la satisfacibilidad booleana (SAT) es NP-Completo.*

Demostración. Para que el problema de la *satisfacibilidad booleana* (SAT) sea NP-Completo es necesario que cumpla con las siguientes propiedades [28]:

- $L_{SAT} \in NP$ y
- una transformación (polinomial) de un problema NP-Completo a L_{SAT}.

Es fácil observar que el problema SAT pertenece a la clase NP, ya que se puede diseñar un verificador que, dado un certificado, determine si el certificado es la solución óptima, simplemente asignando los valores de verdad a cada literal. De modo que, para terminar la demostración, es necesario demostrar que $\forall L \in NP$, $L \propto L_{SAT}$. Para demostrar esto es necesario regresar al nivel de lenguaje, donde el problema SAT es representado por el lenguaje $L_{SAT} = L[SAT, e]$ bajo un sistema de codificación e. Existen muchos y muy diversos lenguajes en NP (en teoría, el conjunto NP es de cardinalidad infinita), de modo que resulta imposible proveer una transformación para cada uno de ellos. Sin embargo, todos los lenguajes en NP se pueden describir de forma estándar, dando como resultado una máquina de Turing no determinista *general*, de tiempo polinomial, que reconoce a todos los lenguajes en NP. Utilizando esta máquina de Turing no determinista *general* es posible deducir una transformación *general* hacia el lenguaje L_{SAT}. Cuando esta transformación *general* se particulariza para una máquina de Turing no determinista M que reconoce el lenguaje L_M, se obtiene una transformación polinomial de L_M a L_{SAT}. De esta manera Cook logra demostrar que $\forall L \in NP$, $L \propto L_{SAT}$. La demostración completa se puede encontrar en [11]. □

2.1.3. Clase NP-Difícil

Aunque los problemas *NP-Completos* pertenecen a la clase NP, es posible utilizar las técnicas que permiten determinar si un problema es *NP-Completo* para determinar si un problema fuera de NP también es "difícil". Cualquier problema de decisión, sea o no parte de

NP, hacia el cual podamos transformar un problema NP-*Completo*, tendrá la propiedad de que no puede ser resuelto en tiempo polinomial a menos que $P = NP$. Este tipo de problemas son denominados problemas NP-*Difíciles*, en el sentido de que, en el mejor de los casos, son tan "difíciles" como los problemas NP-*Completos*.

Muchos problemas son planteados como problemas de *búsqueda*, donde un problema de búsqueda Π consta de un conjunto de instancias D_Π y de un conjunto de soluciones $S_\Pi[I]$ para cada $I \in D_\Pi$. Se dice que un algoritmo *resuelve* un problema de búsqueda Π si, para cualquier instancia de entrada $I \in D_\Pi$, regresa como respuesta *no* cada vez que $S_\Pi[I]$ es vacío o en caso contrario devuelve alguna solución s que pertenece a $S_\Pi[I]$.

Cualquier problema de decisión Π puede ser formulado como un problema de búsqueda al definir las siguientes implicaciones, donde Y_Π es el conjunto de instancias de *sí* del problema de decisión:

- Si $I \in Y_\Pi$ entonces $S_\Pi[I] = sí$
- Si $I \notin Y_\Pi$ entonces $S_\Pi[I] = \emptyset$

Por lo tanto, asumiendo que los problemas de decisión pueden ser formulados de esta manera, un problema de decisión puede ser considerado como un caso especial de un problema de búsqueda.

La definición formal de un problema de búsqueda implica el uso del concepto de *relación de cadena*. Para un alfabeto finito Σ, una relación de cadena sobre Σ es una relación binaria $R \subseteq \Sigma^+ \times \Sigma^+$, donde $\Sigma^+ = \Sigma^* - \{\epsilon\}$, es decir, el conjunto de todas las cadenas no vacías sobre Σ. Un lenguaje L sobre Σ puede ser definido con la relación de cadena:

$$R = \{(x, s) : x \in \Sigma^+ \text{ y } x \in L\} \tag{2.10}$$

donde s es cualquier símbolo fijo de Σ. Una función $f : \Sigma^* \rightarrow \Sigma^*$ *genera* la relación de cadena R si y solo si, para cada $x \in \Sigma^*$, $f(x) = \epsilon$ si no existe alguna $y \in \Sigma^+$ para la cual $(x, y) \in R$. Una máquina de Turing determinista M *resuelve* la relación de cadena R si la función f_M calculada por M genera a R.

La correspondencia entre los problemas de búsqueda y las relaciones de cadena se logra mediante los sistemas de codificación, sólo que ahora un sistema de codificación para Π debe dar una cadena codificada para cada instancia $I \in D_\Pi$ y una cadena codificada para cada solución $s \in S_\Pi[I]$. Bajo un sistema de codificación e, el problema de búsqueda Π corresponde a la relación de cadena $R[\Pi, e]$ definida como:

$$R[\Pi, e] = \left\{ (x, y) : \begin{array}{l} x \in \Sigma^+ \text{ es la codificación bajo } e \text{ de una instancia } I \in D_\Pi \\ y \in \Sigma^+ \text{ es la codificación bajo } e \text{ de una instancia } s \in S_\Pi[I] \end{array} \right\} \tag{2.11}$$

Se dice que Π se resuelve mediante un algoritmo de tiempo polinomial si existe una máquina de Turing determinista de tiempo polinomial que resuelve $R[\Pi, e]$.

A continuación se describe una generalización del concepto de *transformación polinomial*, la cual procede del hecho de que una transformación polinomial de un problema de decisión Π a un problema de decisión Π' proporciona un algoritmo A para resolver Π mediante el uso de una "subrutina" hipotética para resolver Π'. Dada cualquier instancia I de Π, el algoritmo construye una instancia equivalente I' para Π', sobre la cual aplica la subrutina y finalmente entrega la solución obtenida, la cual es también la solución correcta para I. El algoritmo A se ejecuta en tiempo polinomial, excepto por el tiempo requerido por la subrutina. Por lo

tanto, si la subrutina fuera un algoritmo de tiempo polinomial para resolver Π', entonces el procedimiento en general sería un algoritmo de tiempo polinomial para resolver Π.

Nótese que lo mencionado en el párrafo anterior es verdad sin importar si la subrutina hipotética es ejecutada más de una vez (siempre y cuando esté acotada por un polinomio) o si resuelve un problema de búsqueda en vez de uno de decisión. Por lo tanto, una *reducción de Turing de tiempo polinomial* de un problema de búsqueda Π a un problema de búsqueda Π' es un algoritmo A que resuelve Π usando una subrutina hipotética S para resolver Π' de tal manera que si S fuera un algoritmo de tiempo polinomial para Π', entonces A sería un algoritmo de tiempo polinomial para Π.

La formalización del concepto anterior se realiza en términos de *máquinas oráculo*. Una máquina de Turing oráculo consiste de una máquina de Turing determinista, aumentada con una *cinta oráculo* que tiene los elementos de su cinta etiquetados como $..., -2, -1, 0, 1, 2, ...$ y una *cabeza oráculo* de lectura-escritura para trabajar con esta cinta (véase la Figura 2.6).

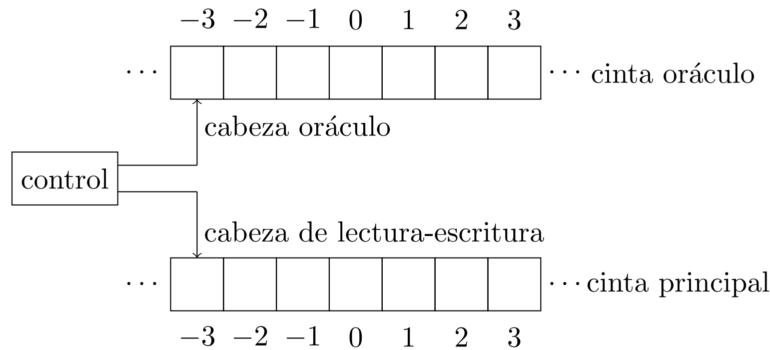


Figura 2.6: Máquina de Turing oráculo

Un programa para una máquina de Turing oráculo es similar al de una máquina de Turing determinista y especifica lo siguiente:

1. Un conjunto finito Γ de *símbolos de cinta* (alfabeto de la cinta), que incluye un subconjunto $\Sigma \subset \Gamma$ de *símbolos de entrada* (alfabeto de entrada) y un *símbolo en blanco* $\llcorner\!\lrcorner \in \Gamma - \Sigma$.

2. Un conjunto finito Q de estados, incluyendo un estado inicial q_0, un estado final q_h, un estado oráculo de consulta q_c y un estado de reanudación del cálculo q_r.

3. Una función de transición $\delta : (Q - \{q_h, q_c\} \times \Gamma \times \Sigma \to Q \times \Gamma \times \Sigma \times \{R, L\} \times \{R, L\})$

El cálculo de un programa para una máquina de Turing oráculo sobre una entrada $x \in \Sigma^*$ es similar al de un programa para una máquina de Turing determinista, con la excepción de que cuando la máquina se encuentra en el estado q_c, lo que ocurra en el siguiente paso dependerá de una *función oráculo* específica $g : \Sigma^* \to \Sigma^*$. El cálculo comienza con los símbolos de x escritos del cuadro 1 al cuadro $|x|$ de la cinta principal, mientras que los cuadros restantes (incluyendo a toda la cinta oráculo) se llenan con el *símbolo en blanco* $\llcorner\!\lrcorner$. La cabeza de lectura-escritura se ubica sobre el cuadro 1 de su cinta y el estado inicial es el q_0. El cálculo se desarrolla paso a paso, siendo cada paso alguno de los siguientes:

1. Si el estado actual es q_h, entonces el cálculo termina y ningún otro paso se lleva a cabo.

2. Si el estado actual es $q \in Q - \{q_h, q_c\}$, entonces la acción a ejecutar dependerá de los símbolos que se estén revisando en las dos cintas y de la función de transición δ.

Sea s_1 el símbolo que está revisando la cabeza de la cinta principal, s_2 el símbolo que está revisando la cabeza de la cinta oráculo y $(q', s_1', s_2', \Delta_1, \Delta_2)$ el valor de $\delta(q, s_1, s_2)$. Con base en estos parámetros el control de estados finito cambia del estado q al estado q', la cabeza de la cinta principal escribe s_1' en el lugar de s_1 y cambia su posición según Δ_1 lo indique (hacia la derecha si $\Delta_1 = R$ o hacia la izquierda si $\Delta_1 = L$) y la cabeza oráculo escribe s_2' en el lugar de s_2 y cambia su posición de acuerdo a Δ_2. De manera que este paso es similar al de una máquina de Turing determinista, con la diferencia de que involucra dos cintas.

3. Si el estado actual es q_c, entonces la acción a ejecutar dependerá del contenido de la cinta oráculo y de la función oráculo g. Sea $y \in \Sigma^*$ la cadena que aparece en los cuadros 1 a $|y|$ de la cinta oráculo, donde el cuadro $|y| + 1$ es el primer cuadro a la derecha del cuadro 0 que contiene un *símbolo en blanco* ($\sqcup$), y sea $z \in \Sigma^*$ el valor de $g(y)$; entonces, en un paso, la cinta oráculo se modifica para contener la cadena de z en los cuadros 1 a $|z|$ y símbolos en blanco en los cuadros restantes, mientras que la cabeza oráculo se coloca sobre el cuadro 1 y el control de estados finito se traslada del estado q_c al estado q_r. Durante este paso no cambia la posición de la cabeza de lectura-escritura ni el contenido de la cinta principal.

La principal diferencia entre una máquina de Turing determinista y una máquina de Turing oráculo radica en este tercer tipo de paso, mediante el cual un programa para un máquina de Turing oráculo puede *consultar* el oráculo. Si una máquina de Turing oráculo escribe una cadena de consulta y entonces entra al estado-oráculo de consulta y la cadena de respuesta $z = g(y)$ será devuelta en un solo paso del cálculo. Por lo tanto, esto corresponde a una subrutina hipotética para calcular la función g. El cálculo de un programa para una máquina de Turing oráculo M sobre una cadena x depende tanto de esta cadena como de la función oráculo g asociada.

Sea M_g un programa para una máquina de Turing oráculo obtenido de la combinación de M con el oráculo g. Si M_g se *detiene* para todas las entradas $x \in \Sigma^*$, entonces también puede ser considerada como la computación de una función $f_M^g : \Sigma^* \to \Gamma^*$, definida exactamente de la misma forma que para una máquina de Turing determinista. Se dice que M_g es un programa para una máquina de Turing oráculo de tiempo polinomial si existe un polinomio p tal que M_g se detiene en $p(|x|)$ pasos para cada entrada $x \in \Sigma^*$.

Sean R y R' dos relaciones de cadena sobre Σ. Una *reducción de Turing en tiempo polinomial* de R a R' es un programa para una máquina de Turing oráculo M con un alfabeto de entrada Σ tal que, para cada función $g : \Sigma^* \to \Sigma^*$ que produce a R', el programa generado M_g es un programa para una máquina de Turing oráculo de tiempo polinomial y la función f_M^g calculada por M_g produce a R. Si dicha reducción de R a R' existe, se escribe como $R \propto_T R'$, que se lee como "*R es Turing-reducible a R'*". La relación $\propto_T$, al igual que la relación $\propto$, es transitiva.

Una relación de cadena R es *NP-Difícil* si existe algún lenguaje *NP-Completo* L tal que $L \propto_T R$. Un problema de búsqueda Π (bajo el sistema de codificación e) es *NP-Difícil* si la relación de cadena $R[\Pi, e]$ es *NP-Difícil*. Informalmente, esto se interpreta de la siguiente manera: un problema de búsqueda Π es *NP-Difícil* si existe algún problema *NP-Completo* Π' que es *Turing-reducible* a Π. Es fácil observar que si una relación de cadena R (o problema de búsqueda Π) es *NP-Difícil*, entonces no puede ser resuelto en tiempo polinomial a menos que $P = NP$.

Por transitividad de $\propto_T$, si R es cualquier relación de cadena *NP-Difícil* y si R es *Turing-reducible* a la relación de cadena R', entonces R' también es *NP-Difícil*. Además, por la asociación de lenguajes con relaciones de cadenas y problemas de decisión con problemas de

búsqueda, se puede afirmar que todos los lenguajes *NP-Completos* y todos los problemas *NP-Completos* son *NP-Difíciles*. En la Figura 2.7 se muestra la relación entre las clases planteadas.

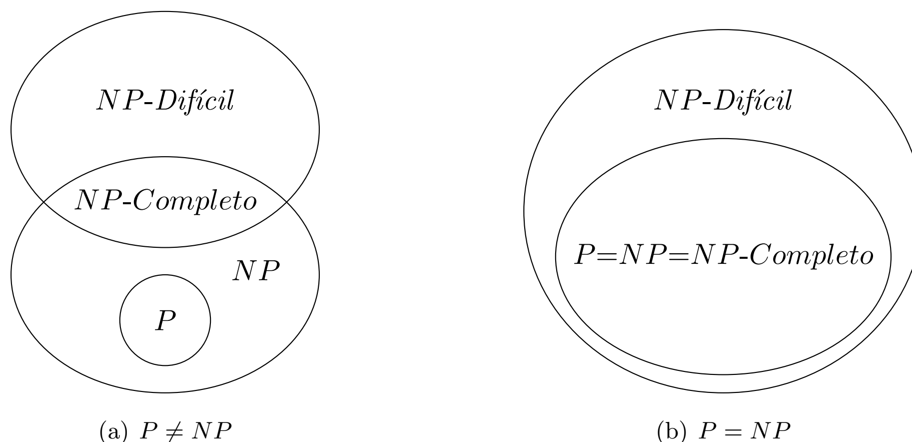


Figura 2.7: Relación entre las clases P, NP, *NP-Completo* y *NP-Difícil*

2.2. Algoritmos de aproximación

Antes de presentar la definición de *algoritmo de aproximación*, es importante definir la clase de problemas de *NP-Optimización* (*NP-Optimization* o simplemente *problemas de optimización combinatoria*).

Un problema de *NP-Optimización*, Π, debe contar con los siguientes elementos y características:

- Un conjunto de *instancias válidas*, D_Π, reconocibles en tiempo polinomial. Se asume que todos los números especificados en la entrada son racionales, para así evitar el problema de tener que manejar precisión aritmética infinita. El tamaño de una instancia $I \in D_\Pi$, denotado por $|I|$, es definido como el número de bits necesarios para representar a I (desde luego que esta definición es propia de un modelo que requiere que la entrada sea especificada en sistema binario).

- Cada instancia $I \in D_\Pi$ tiene asociado un conjunto de *soluciones factibles*, $S_\Pi(I)$. Es necesario que $S_\Pi(I) \neq \emptyset$ y que el tamaño de cada solución $s \in S_\Pi(I)$ esté acotado polinomialmente por $|I|$. Más aún, debe existir un algoritmo de tiempo polinomial que, dado un par (I, s), decida si $s \in S_\Pi(I)$.

- Existe una *función objetivo*, obj_Π, que puede ser calculada en tiempo polinomial. Esta función objetivo asigna un número racional no negativo a cada pareja (I, s), donde I es una instancia y s una solución factible para I. Comúnmente se asocia a la función objetivo una interpretación física, tal como *peso*, *costo*, *distancia*, etc.

- Debe especificarse si el problema Π es de *maximización* o *minimización*.

Una *solución óptima* para una instancia de un problema de minimización (o maximización) es aquella *solución factible* cuyo valor de la función objetivo es mínimo (o máximo). $OPT_\Pi(I)$

denota el valor de la función objetivo de una solución óptima para la instancia I (comúnmente se abrevia OPT).

A cada problema de NP-*Optimización* se le puede asociar un problema de decisión que recibe como entrada una cota para la solución óptima. Es decir, la versión de decisión, π, de un problema de NP-*Optimización* Π recibe como entrada una pareja (I, B), donde I es una instancia de Π y B es un número racional. Si π es un problema de minimización (o maximización), entonces la respuesta al problema π es *"sí"* si y solo si existe una solución factible para I con costo menor o igual a B (o mayor o igual a B). Si es así, se dice que (I, B) es una *instancia de "sí"* (*"yes" instance*); de lo contrario es una *instancia de "no"* (*"no" instance*).

Claramente, un algoritmo de tiempo polinomial para un problema Π permite resolver también su versión de decisión π (bastaría encontrar la solución óptima y compararla con B). De igual manera, la dificultad planteada por el problema de decisión π es heredada al problema Π. De hecho, la dificultad de un problema de NP-*Optimización* se establece al demostrar que su versión de decisión es NP-*Difícil*. Abusando un poco de la notación, se puede decir que la versión de optimización de un problema de decisión NP-*Difícil*, también es un problema NP-*Difícil* [3].

Si $P \neq NP$, la única forma de encontrar soluciones óptimas para problemas NP-*Difíciles* es utilizando el método de *fuerza bruta*, el cual tiene una complejidad de orden exponencial. Cuando encontrar una solución óptima implica invertir una cantidad de tiempo exponencial, resulta razonable considerar técnicas de resolución que garanticen tiempos de ejecución polinomial, aunque a costa del sacrificio de la optimalidad. La idea fundamental del diseño de algoritmos de aproximación es justamente la de integrar, de la mejor manera posible, la eficiencia con la optimalidad; es decir, garantizar la entrega de soluciones tan próximas a la óptima como sea posible en tiempo polinomial.

El diseño de algoritmos (en general) bien podría dividirse en dos partes:

1. Estudiar la estructura del problema y seleccionar los elementos de la misma que servirán de base para el algoritmo.

2. Explotar los elementos seleccionados, dando como resultado el algoritmo deseado.

La dificultad del diseño de algoritmos polinomiales exactos, para la resolución de problemas NP-*Difíciles*, radica en que la estructura de estos problemas es tal que simplemente no es posible aprovecharla (en caso de que $P \neq NP$) o quizá es tan compleja que nadie ha sido capaz de vislumbrar la forma de explotarla (en caso de que $P = NP$). El diseño de algoritmos de aproximación es básicamente igual al del diseño de algoritmos polinomiales exactos, en el sentido de que se requiere del aprovechamiento de la estructura del problema. Los problemas NP-*Difíciles* no tienen (o no se ha descubierto o bien no se ha sabido explotar) la estructura necesaria para diseñar algoritmos polinomiales exactos; sin embargo sí cuentan con la estructura necesaria para resolver versiones más restringidas del problema, donde la restricción consiste en permitir que las soluciones entregadas no sean necesariamente la solución óptima, pero sí cercanas a ella.

Las soluciones entregadas por un algoritmo de aproximación, si bien no son las óptimas, tampoco son arbitrariamente incorrectas. Es decir, el valor de las soluciones entregadas puede ser acotado en función del tamaño de la solución óptima, lo cual nos permite saber cuán cercana es la solución obtenida a la solución óptima, incluso sin necesidad de conocer esta última.

Definición 2. *Se dice que un algoritmo A entrega soluciones $A(I)$ con una ρ-aproximación si para toda instancia I con solución óptima $OPT(I)$ se cumple lo siguiente:*

$$\frac{A(I)}{OPT(I)} \leq \rho \tag{2.12}$$

Para problemas de minimización el valor de ρ debe ser mayor o igual a 1, mientras que para problemas de maximización ρ debe ser menor o igual a 1. Claramente la principal dificultad en el diseño de algoritmos de aproximación consiste en determinar la cota superior o inferior de las soluciones generadas, la cual permite calcular el valor del factor de aproximación, es decir, el valor de ρ.

El valor de la variable ρ suele recibir diferentes nombres, tales como:

- Cota del peor caso (*worst case bound*).
- Desempeño del peor caso (*worst case performance*).
- Factor de aproximación (*approximation factor*).
- Razón de aproximación (*approximation ratio*).
- Cota del desempeño (*performance bound*).
- Razón del desempeño (*performance ratio*).
- Razón del error (*error ratio*).

Así como es importante determinar el factor de aproximación ρ de un algoritmo de aproximación, también es importante garantizar que éste se ejecuta en tiempo polinomial. Si bien la definición formal del término *eficiencia* está dada en términos del tipo de función que describe el tiempo de ejecución (polinomial o exponencial), existen algoritmos polinomiales (en teoría eficientes) que en realidad son ineficientes y poco prácticos. Por lo tanto, resulta deseable contar con algoritmos cuyo tiempo de ejecución sea no solo polinomial, sino también práctico; es decir, algoritmos donde el tiempo de ejecución sea proporcional al factor de aproximación, donde, desde luego, el tiempo de ejecución pueda ser definido a *nivel de usuario*. Este balance entre factor de aproximación y tiempo de ejecución forma parte de los llamados *esquemas de aproximación*.

Definición 3. *Una familia de algoritmos de aproximación para un problema $\mathcal{P}$, $\{\mathcal{A}_\epsilon\}_\epsilon$, es denominada un "esquema de aproximación polinomial" o PAS ("polynomial approximation scheme"), si el algoritmo $\mathcal{A}_\epsilon$ encuentra una $(1+\epsilon)$-aproximación en tiempo polinomial con respecto al tamaño de la entrada para un valor fijo de ϵ.*

Definición 4. *Una familia de algoritmos de aproximación para un problema $\mathcal{P}$, $\{\mathcal{A}_\epsilon\}_\epsilon$, es denominada un "esquema de aproximación completamente polinomial" o FPAS ("fully polynomial approximation scheme"), si el algoritmo $\mathcal{A}_\epsilon$ encuentra una $(1+\epsilon)$-aproximación en tiempo polinomial con respecto al tamaño de la entrada y al valor $1/\epsilon$.*

2.3. Algoritmos aleatorizados

La aleatoriedad suele ser definida como aquella característica propia de los procesos que entregan resultados imposibles de preveer con certeza, por ejemplo, el lanzamiento de un dado. Dada esta definición, pareciera que el estudio y uso de la aleatoriedad en las ciencias de la

computación representa una desventaja o un retroceso, pues aparentemente complica aún más los retos existentes. Sin embargo, a partir del desarrollo de la teoría cuántica (donde se sugiere que la base de las leyes del universo es de carácter probabilístico), la aleatoriedad ha logrado ser aceptada en el ámbito científico, jugando un papel importante en prácticamente todas las ciencias, sin ser las ciencias de la computación la excepción [30].

Un algoritmo aleatorizado es básicamente aquel que toma decisiones aleatorias en algún punto de su ejecución. En la práctica los algoritmos aleatorizados (su implementación) hacen uso de valores entregados por generadores de números aleatorios. Algunos ejemplos de aplicaciones comunes donde la aleatoriedad juega un papel importante son [30]:

- Protocolo de comunicación de la tarjetas de red Ethernet.
- Primalidad de números en criptografía.
- Verificación de igualdad de polinomios.

Dado que los algoritmos aleatorizados toman decisiones aleatorias en algún punto de su ejecución, resulta natural que tiendan a entregar soluciones diferentes entre una ejecución y otra, contrario a lo que ocurre con los algoritmos deterministas. Los algoritmos aleatorizados pueden ser clasificados con base en sus características elementales. Dos de los principales tipos de algoritmos aleatorizados son los algoritmos tipo “Las Vegas” y tipo “Monte Carlo” [40]:

- **Las Vegas**. algoritmos aleatorizados que encuentran la solución óptima, pero con un tiempo de ejecución variable.
- **Monte Carlo**. algoritmos aleatorizados que, con una probabilidad mayor a cero y menor a 1, encuentran la solución óptima. Su tiempo de ejecución es polinomial.

La base de los algoritmos aleatorizados es la teoría de la Probabilidad, por lo cual es necesario presentar un resumen de los conceptos básicos de ésta.

2.3.1. Conceptos de probabilidad

Un *modelo probabilístico* es una descripción matemática de una situación donde existe incertidumbre. Los elementos de un modelo probabilístico son [12] (Figura 2.8) :

1. **Espacio de muestreo** Ω. Es el conjunto de todas las posibles salidas de un experimento.
2. **Función de probabilidad (o ley de probabilidad)** . Es la encargada de asignar a cada conjunto A de posibles salidas (también denominadas **eventos**) un número no negativo $\mathbf{P}(A)$ (el cual se lee como “probabilidad de A”) que representa nuestro conocimiento o creencia acerca del potencial del conjunto A como posible salida. La ley de probabilidad debe satisfacer ciertas propiedades que se enlistarán más adelante.

Todo modelo probabilístico está asociado a un proceso denominado “experimento”, el cual genera una salida que forma parte de un conjunto de posibles salidas. Al conjunto de posibles salidas se le denomina “espacio de muestreo” (denotado por el símbolo Ω) del experimento. A cualquier conjunto de salidas se le denomina “evento”. Es importante resaltar que no existe una restricción en cuanto a lo que constituye un experimento, es decir, cualquier fenómeno puede ser considerado como tal, ya sea el lanzamiento de una moneda o un par de lanzamientos de una misma moneda.

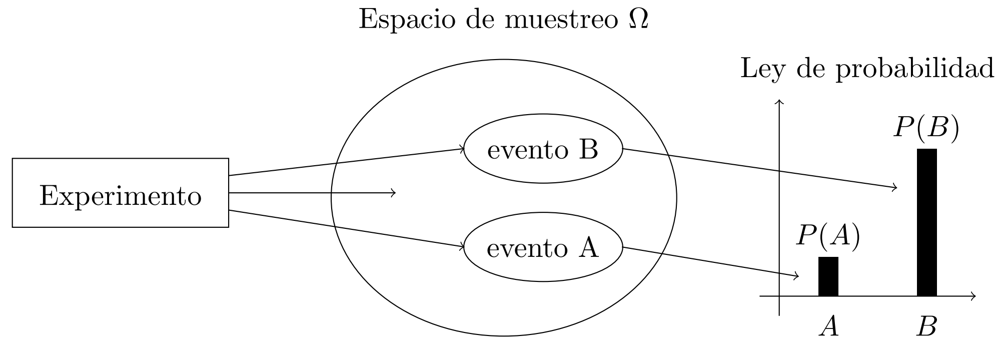


Figura 2.8: Elementos principales de un modelo probabilístico

El espacio de muestreo de un experimento puede consistir de una cantidad finita o infinita de posibles salidas, siendo los espacios de búsqueda finitos los que resultan matemática y conceptualmente más simples.

Una vez que es claro en qué consiste el experimento y es conocido su espacio de muestreo, es necesario completar el modelo con una "*función de probabilidad*" (o "*ley de probabilidad*"). Una función de probabilidad se encarga de asignar a todo evento A un valor $P(A)$, llamado "probabilidad de A", que satisfaga las siguientes condiciones, conocidas como axiomas de la probabilidad:

1. **No-negatividad**. $P(A) \geq 0$ para todo evento A.
2. **Aditividad**. Si A y B son dos eventos disjuntos, la probabilidad de su unión debe satisfacer:

$$P(A \cup B) = P(A) + P(B). \tag{2.13}$$

   En general, si el espacio de búsqueda tiene un número infinito de elementos y $A_1, A_2, ...$ es una secuencia de conjuntos disjuntos, entonces la probabilidad de su unión debe satisfacer:

$$P(A_1 \cup A_2 \cup \ldots) = P(A_1) + P(A_2) + \ldots \tag{2.14}$$

3. **Normalización**. La probabilidad del espacio de muestreo completo debe ser igual a 1:

$$P(\Omega) = 1 \tag{2.15}$$

Una manera práctica de visualizar el comportamiento de una función de probabilidad es imaginar una unidad de masa esparcida sobre todo el espacio de muestreo. De esta manera, $P(A)$ es equivalente a la cantidad de masa esparcida sobre los elementos de A. Utilizando esta analogía el axioma de aditividad resulta claro, pues la suma de las masas de eventos disjuntos da como resultado la cantidad de masa que se repartió sobre dichos eventos inicialmente.

Dado que los eventos son conjuntos, suele usarse la notación de la teoría de conjuntos para expresar combinaciones de eventos. De manera que $E_1 \cap E_2$ representa la ocurrencia de los

eventos E_1 y E_2, mientras que $E_1 \cup E_2$ representa la ocurrencia de al menos uno de los eventos E_1 y E_2. De manera similar $E_1 - E_2$ representa la ocurrencia de un evento en E_1 que no está en E_2.

Lema 2.3.1. *Para cualquier par de eventos E_1 y E_2.*

$$P(E_1 \cup E_2) = P(E_1) + P(E_2) - P(E_1 \cap E_2) \tag{2.16}$$

Demostración. Por definición:

$$P(E_1) = P(E_1 - (E_1 \cap E_2)) + P(E_1 \cap E_2).$$
$$P(E_2) = P(E_2 - (E_1 \cap E_2)) + P(E_1 \cap E_2).$$
$$P(E_1 \cup E_2) = P(E_1 - (E_1 \cap E_2)) + P(E_2 - (E_1 \cap E_2)) + P(E_1 \cap E_2).$$

Sustituyendo apropiadamente:

$$P(E_1 \cup E_2) = P(E_1) - P(E_1 \cap E_2) + P(E_2) - P(E_1 \cap E_2) + P(E_1 \cap E_2)$$
$$P(E_1 \cup E_2) = P(E_1) + P(E_2) - P(E_1 \cap E_2)$$

□

El siguiente lema es consecuencia del segundo axioma de la probabilidad (Aditividad) y es conocido como "*union bound*" o "límite de la unión".

Lema 2.3.2. *Para cualquier secuencia finita o infinita contable de eventos E_1, E_2,...,*

$$P\left(\bigcup_{i \geq 1} E_i\right) \leq \sum_{i \geq 1} P(E_i) \tag{2.17}$$

Nótese que el Lema 2.3.2 difiere del segundo axioma de la probabilidad en que en el segundo se trata de una igualdad y además requiere que los eventos involucrados sean disjuntos entre sí.

Definición 5. *Dos eventos E y F son independientes si y solo si*

$$P(E \cap F) = P(E) \cdot P(F) \tag{2.18}$$

De manera más general, los eventos E_1, E_2,..., E_k son mutuamente independientes si y solo si, para cualquier subconjunto $I \subseteq [1, k]$,

$$P\left(\bigcap_{i \in I} E_i\right) = \prod_{i \in I} P(E_i) \tag{2.19}$$

Definición 6. *La probabilidad de que un evento E ocurra dado que un evento F ha ocurrido es representada por $P(E|F)$ y se lee como probabilidad condicional de E dado F.*

$$P(E|F) = \frac{P(E \cap F)}{P(F)} \tag{2.20}$$

La probabilidad condicional solo puede ser bien definida cuando $P(F) > 0$

Intuitivamente, la probabilidad que se busca es la del evento $E \cap F$ dentro del conjunto de eventos definido por F. Dado que F restringe nuestro espacio de muestreo, se normaliza la probabilidad dividiendo por $P(F)$, de manera que la suma de las probabilidades de todos los eventos sea 1. Cuando $P(F) > 0$, la definición anterior puede ser escrita como:

$$P(E|F) \cdot P(F) = P(E \cap F) \tag{2.21}$$

Nótese que, cuando E y F son independientes y $P(F) \neq 0$:

$$P(E|F) = \frac{P(E \cap F)}{P(F)} = \frac{P(E) \cdot P(F)}{P(F)} = P(E) \tag{2.22}$$

Esta es una propiedad que la probabilidad condicional debe poseer pues, intuitivamente, si dos eventos son independientes la información de uno de ellos no tiene por qué afectar a la probabilidad del otro.

Con base en las definiciones de probabilidad condicional es posible generalizar la ecuación 2.19 para el caso en que los eventos no son independientes entre sí:

Definición 7. *Si dos eventos EyF no son independientes entre sí:*

$$P[E \cap F] = P[E|F] \cdot P[F] = P[F|E] \cdot P[E] \tag{2.23}$$

[Regla de la multiplicación]: *La probabilidad de la intersección de un conjunto de i eventos está dada por:*

$$P[\cap_{i=1}^{k} E_i] = P[E_1] \cdot P[E_2|E_1] \cdot P[E_3|E_1 \cap E_2] \cdots P[E_k| \cap_{i=1}^{k-1} E_i] \tag{2.24}$$

Teorema 2.3.3. ***[Ley de la probabilidad total]:*** *Sean $E_1, E_2, ..., E_n$ eventos disjuntos del espacio de muestreo Ω, y sea $\bigcup_{i=1}^{n} E_i = \Omega$. Entonces:*

$$P(B) = \sum_{i=1}^{n} P(B \cap E_i) = \sum_{i=1}^{n} P(B|E_i)P(E_i) \tag{2.25}$$

Demostración. Dado que los eventos $B \cap E_i (i = 1, ..., n)$ son disjuntos y cubren completamente el espacio de muestreo Ω:

$$P(B) = \sum_{i=1}^{n} P(B \cap E_i)$$

Haciendo uso de la definición de probabilidad condicional:

$$\sum_{i=1}^{n} P(B \cap E_i) = \sum_{i=1}^{n} P(B|E_i)P(E_i)$$

□

Intuitivamente el teorema de la probabilidad total consiste en particionar el espacio de muestreo Ω en un cierto número de escenarios (eventos) E_i, para así calcular la probabilidad de B con base en la probabilidad de B dentro de cada escenario. Este teorema resulta útil cuando las probabilidades condicionales $P(B|E_i)$ son conocidas.

Teorema 2.3.4. ***[Regla de Bayes]:*** *Si $E_1, E_2, ..., E_n$ son conjuntos mutuamente disjuntos, tales que $\bigcup_{i=1}^{n} E_i = E$. Entonces:*

$$P(E_j|B) = \frac{P(E_j \cap B)}{P(B)} = \frac{P(B|E_j)P(E_j)}{\sum_{i=1}^{n} P(B|E_i)P(E_i)}$$

La regla de Bayes permite relacionar las probabilidades condicionales de la forma $P(A|B)$ con las de la forma $P(B|A)$. Para verificar la regla de Bayes solo hay que notar que $P(E_j|B) \cdot P(B)$ es igual a $P(B|E_j) \cdot P(E_j)$, pues ambas son iguales a $P(E_j \cap B)$. Posteriormente se utiliza el teorema de la probabilidad total para completar la regla de Bayes.

La regla de Bayes se usa comúnmente para "inferir causas". Es decir, si existe un cierto número de posibles causas (eventos) para un fenómeno (otro evento), es deseable inferir qué causa lo provocó. Los eventos E_j se asocian a las causas, mientras que el evento B representa el fenómeno.

Definición 8. *Una variable aleatoria X en un espacio de muestreo Ω es una función de Ω evaluada en los reales; es decir, $X : \Omega \to \mathbb{R}$. Una variable aleatoria discreta es una variable aleatoria que toma únicamente un número finito o infinito contable de valores.*

Dado que las variables aleatorias son funciones, suelen ser denotadas con letras mayúsculas como X y Y, mientras que los números reales son denotados con letras minúsculas.

Para una variable aleatoria discreta X y un número real a, el evento "$X = a$" incluye a todos los eventos del espacio de muestreo donde la variable X toma el valor de a. Es decir, "$X = a$" representa el conjunto $\{s \in \Omega | X(s) = a\}$. La probabilidad de este evento se denota por:

$$P(X = a) = \sum_{s \in \Omega : X(s) = a} P(s) \tag{2.26}$$

Definición 9. *Dos variables aleatorias X y Y son independientes si y solo si:*

$$P((X = x) \cap (Y = y)) = P(X = x) \cdot P(Y = y) \tag{2.27}$$

para todos los valores "x" y "y". De manera similar, las variables aleatorias $X_1, X_2, ..., X_k$ son mutuamente independientes si y solo si, para cualquier subconjunto $I \subseteq [1, k]$ y cualquier valor de $x_i, i \in I$:

$$P\left(\bigcap_{i \in I} X_i = x_i\right) = \prod_{i \in I} P(X_i = x_i) \tag{2.28}$$

Una característica básica de las variables aleatorias es la *esperanza*. La *esperanza* de una variable aleatoria es el promedio de los valores que puede asumir, donde cada valor es calculado con base en la probabilidad de que la variable aleatoria asuma ese valor.

Definición 10. *La* esperanza *de una variable aleatoria discreta X, denotada por $E[X]$, está dada por:*

$$E[X] = \sum_i i \ P(X = i) \tag{2.29}$$

donde la sumatoria se realiza sobre todos los valores en el rango de X. La esperanza es finita si $\sum_i |i| \ P(X = i)$ converge; de otra manera la esperanza *no puede ser acotada.*

Una propiedad clave de la *esperanza* que simplifica significativamente su cálculo es la *linealidad de la esperanza*. Esta propiedad consiste en que la *esperanza* de la suma de variables aleatorias es igual a la suma de sus respectivas *esperanzas*.

Teorema 2.3.5. *Para cualquier colección finita de variables aleatorias discretas $X_1, X_2, ..., X_n$ con* esperanza *finita:*

$$E\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} E[X_i] \tag{2.30}$$

Demostración. Se puede demostrar este teorema para dos variables aleatorias X y Y; el caso general se demuestra por inducción. Las siguientes sumatorias son realizadas sobre el rango de cada variable aleatoria:

$$
\begin{aligned}
E[X+Y] = & \sum_i \sum_j (i+j) P((X=i) \cap (Y=j)) \\
= & \sum_i \sum_j i \ P((X=i) \cap (Y=j)) + \sum_i \sum_j j \ P((X=i) \cap (Y=j)) \\
= & \sum_i i \sum_j P((X=i) \cap (Y=j)) + \sum_j i \sum_i P((X=i) \cap (Y=j)) \\
= & \sum_i i \ P(X=i) + \sum_j j \ P(Y=j) \\
= & E[X] + E[Y]
\end{aligned}
$$

La primera igualdad se obtiene directamente de los axiomas de la probabilidad. La penúltima igualdad se obtiene utilizando el teorema de la probabilidad total. □

Lema 2.3.6. *Para cualquier valor constante c y para cualquier variable aleatoria X,*

$$E[cX] = c \ E[X]$$

Demostración. Para $c = 0$ el lema es obvio. Para $c \neq 0$,

$$
\begin{aligned}
E[cX] = & \sum j \ P(cX = j) \\
= & c \sum_j (j/c) \ P(X = j/c) \\
= & c \sum_k k \ P(X = k) \\
= & c \ E[X]
\end{aligned}
$$

□

2.3.2. Algoritmos aleatorizados tipo *Las Vegas*

Los algoritmos aleatorizados tipo Las Vegas son aquellos que siempre encuentran la solución óptima, pero en un tiempo variable. Uno de los conceptos elementales para el diseño de este tipo de algoritmos es el de la *linealidad de la esperanza*. Para comprender mejor la naturaleza de los algoritmos aleatorizados tipo Las Vegas se muestra a continuación un ejemplo clásico: el algoritmo *RandQS* (*Random QuickSort*).

2.3.2.1. Algoritmo *Random QuickSort*

Sea S un conjunto de n números que se desea ordenar ascendetemente. Si fuera posible encontrar en un solo paso un elemento y de S tal que la mitad de los elementos restantes sean menores a él, entonces podría seguirse el siguiente enfoque para resolver el problema: particionar $S \setminus \{y\}$ (lo cual es equivalente a $S - \{y\}$) en dos conjuntos S_1 y S_2, donde S_1 consista de aquellos elementos de S que son menores a y, y S_2 de aquellos mayores a y. Recursivamente son ordenados ascendentemente los conjuntos S_1 y S_2, siendo la salida del algoritmo el conjunto S_1 ordenado, seguido de y y del conjunto S_2 ordenado. Si fuera posible encontrar el elemento y en cn pasos, donde c es un valor constante, entonces sería posible particionar $S \setminus \{y\}$ en S_1 y S_2 en $n-1$ pasos al comparar cada elemento de S con y. De manera que el número total de pasos requeridos por este procedimiento estería definido por la siguiente recurrencia:

$$T(n) \leq 2T(n/2) + (c+1)n \tag{2.31}$$

donde $T(n)$ representa el tiempo máximo requerido por este método para ordenar n números.

Utilizando el método de sustitución directa es posible demostrar que la complejidad de esta recurrencia es de $c'n \log n$; es decir $T(n) \leq c'n \log n$ para un valor constante c'.

Demostración. Para corroborar que la complejidad de la recursión es $c'n \log n$ es necesario verificarlo por inducción. El caso base de la recursión se da cuando $n = 2$ (para $n = 1$ la solución es la misma entrada). El elemento y es ubicado en $(c+1)n$ pasos. La partición de los restantes elementos (que es uno solo) se realiza en $T(1) = 0$ pasos, pues al tratarse de un solo elemento la partición consiste justamente de éste. Por lo tanto:

$$\begin{aligned} T(2) = &\quad 2(0) + (c+1)n \\ = &\quad (c+1)n \\ = &\quad (c+1)n = c'n \\ = &\quad c'n \end{aligned}$$

Para el caso base se cumple la hipótesis sobre la complejidad:

$$\begin{aligned} T(2) \leq &\quad c'n \\ \leq &\quad c'2 \\ \leq &\quad c'n \log n \\ \leq &\quad c'2 = c'2 \log 2 \end{aligned}$$

Para demostrar que la hipótesis sobre la complejidad se cumple para cualquier valor de n utilizamos la sustitución directa:

$$\begin{aligned} T(n) \leq &\quad 2T(n/2) + cn \\ \leq &\quad 2c(n/2)\log(n/2) + cn \end{aligned}$$

Dado que:

$$\begin{aligned} 2c(n/2)\log(n/2) + cn = &\quad 2c(n/2)[(\log n) - 1] + cn \\ = &\quad cn[(\log n) - 1] + cn \\ = &\quad cn \log n - cn + cn \\ = &\quad cn \log n \end{aligned}$$

Entonces:

$$T(n) \leq cn \log n$$

□

La dificultad del método descrito radica en encontrar el elemento y que divida $S \setminus \{y\}$ en dos conjuntos S_1 y S_2 del mismo tamaño. Sin embargo, la complejidad de la recurrencia sigue siendo de orden $O(n \log n)$ incluso cuando la cardinalidad de los conjuntos S_1 y S_2 es *aproximadamente* la misma; por ejemplo, cuando S_1 contiene $n/4$ elementos y S_2 contiene $3n/4$ elementos. La recurrencia correspondiente a esta idea es la siguiente:

$$T(n) \leq T(n/4) + T(3n/4) + (c+1)n \tag{2.32}$$

Demostración. La demostración es similar a la realizada para la recurrencia 2.31.

$$\begin{aligned} T(n) \leq &\quad T(n/4) + T(3n/4) + c'n \\ \leq &\quad [d(n/4)\log(n/4)] + [d(3/4)n\log(3n/4)] + c'n \end{aligned}$$

Donde:

$$[d(n/4)\log(n/4)] + [d(3n/4)\log(3n/4)] + c'n$$

es igual a:

$$\begin{aligned} &= (dn\log n) - [d(n/4)\log 4] + [d(3n/4)\log 4] - [d(3n/4)\log(3)] + c'n \\ &= (dn\log n) - (dn\log 4) + [d(3n/4)\log 3] + c'n \\ &= (dn\log n) - [dn((\log 4) - (3/4)\log 3)] + c'n \end{aligned}$$

De modo que para que la recurrencia 2.32 sea de orden $O(n\log n)$ es necesario que la siguiente desigualdad se cumpla:

$$(dn\log n) - [dn((\log 4) - (3/4)\log 3)] + c'n \leq dn\log n$$

Donde d puede ser una constante o una función de n. Si se demuestra que d es una constante queda demostrado que la recurrencia es de orden $O(n\log n)$. Al despejar d observamos que en efecto se trata de una constante:

$$d \geq \frac{c'}{\log 4 - (3/4)\log 3}$$

Dado que d es una constante, queda demostrado que la complejidad de la recurrencia 2.32 es de orden $O(n\log n)$ □

El hecho de que la recurrencia 2.32 sea de orden $O(n\log n)$ es una buena noticia, pues en cualquier instancia del problema (es decir, para cualquier conjunto de n números que se desee ordenar) existen $n/2$ elementos que pueden cumplir con el papel del elemento y. Sin embargo, el problema sigue siendo cómo encontrar dicho elemento y. Una manera rápida de encontrarlo es eligiéndolo de manera aleatoria en un solo paso. Si bien no existe ninguna garantía de seleccionar uno de los $n/2$ elementos en un solo paso, es razonable suponer que en buena parte de las ejecuciones esto sucederá.

El algoritmo RandQS se describe a continuación.

Algoritmo 2.1: Algoritmo RandQS

Entrada: un conjunto de números S

Salida: Los elementos de S ordenados ascendentemente

1. Seleccionar un elemento de S aleatoriamente (todos los elementos de S tienen la misma probabilidad de ser seleccionados) y nombrarlo y ;
2. Crear un conjunto S_1 con todos los elementos de $S \setminus \{y\}$ menores a y. Crear un conjunto S_2 con todos los elementos de $S \setminus \{y\}$ mayores a y. Para formar estos conjuntos es necesario comparar todos los elementos de $S \setminus \{y\}$ contra y ;
3. Ordenar recursivamente los conjuntos S_1 y S_2 ;
4. La salida del algoritmo es el conjunto S_1 ordenado, seguido de y y del conjunto S_2 ordenado ;

Como es usual para los algoritmos de ordenamiento, el tiempo de ejecución del algoritmo RandQS es calculado en términos del número de comparaciones que ejecuta. Dada la naturaleza del algoritmo, el objetivo es más bien calcular el número *esperado* de comparaciones que éste realiza durante su ejecución. Nótese que todas las comparaciones se hacen en el paso 2 del algoritmo. Para $1 \geq i \geq n$, sea $S_{(i)}$ el i-ésimo elemento más pequeño del conjunto S. De manera que $S_{(1)}$ representa al elemento más pequeño de S, y $S_{(n)}$ al más grande. Sea X_{ij} una variable aleatoria que toma el valor de 1 cuando S_i y S_j son comparados durante una ejecución, y el valor 0 cuando no son comparados. De esta manera, la variable aleatoria X_{ij} funciona como un contador de comparaciones entre S_i y S_j; siendo, por lo tanto, el número total de comparaciones igual a $\sum_{i=1}^{n} \sum_{j>i} X_{ij}$. Debido a la naturaleza del algoritmo RandQS no hay manera de saber con certeza cuántas comparaciones se realizan durante una ejecución de éste, por lo cual la información que en realidad nos interesa conocer es el número *esperado* de comparaciones que se realizan durante su ejecución:

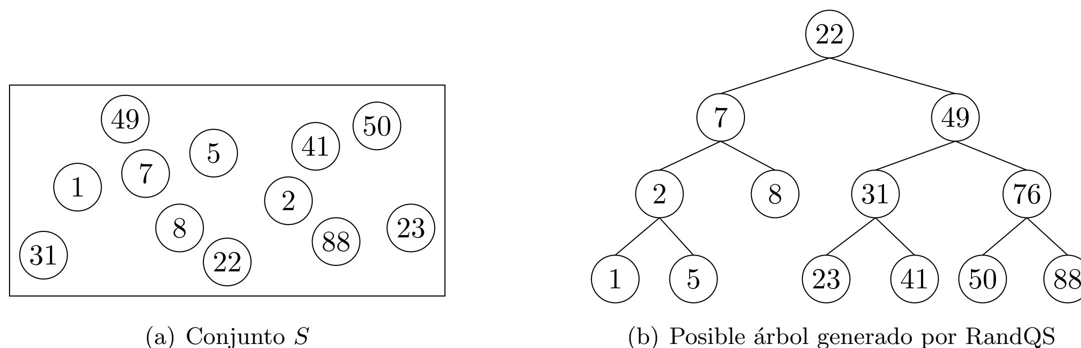
$$E\left[\sum_{i=1}^{n} \sum_{j>i} X_{ij}\right] = \sum_{i=1}^{n} \sum_{j>i} E[X_{ij}] \tag{2.33}$$

Esta ecuación hace uso de la propiedad de la *esperanza* denominada *linealidad de la esperanza.*

Sea P_{ij} la probabilidad de que S_i y S_j sean comparados en una ejecución. Dado que X_{ij} solo toma los valores 0 y 1:

$$E[X_{ij}] = P_{ij} \cdot 1 + (1 - P_{ij}) \cdot 0 = P_{ij} \tag{2.34}$$

Si se visualiza la ejecución del algoritmo RandQS como un árbol binario T, donde cada nodo está etiquetado con un elemento de S, resulta más sencillo calcular el valor de P_{ij}. La raíz del árbol es etiquetada con el elemento y seleccionado al inicio del algoritmo, el sub-árbol izquierdo contiene a los elementos de S_1 y el sub-árbol derecho contiene a los elementos de S_2. La estructura de los sub-árboles es determinada recursivamente por las ejecuciones del algoritmo sobre los conjuntos S_1 y S_2 (véase la Figura 2.9).



(a) Conjunto S

(b) Posible árbol generado por RandQS

Figura 2.9: Ejemplo de árbol binario generado por RandQS

La raíz y del árbol generado por el algoritmo RandQS es comparada con los elementos de los dos sub-árboles, pero ninguna comparación es realizada entre los elementos del sub-árbol izquierdo con los del sub-árbol derecho. Por lo tanto, se realiza una comparación entre los elementos $S_{(i)}$ y $S_{(j)}$ si y solo si alguno de estos elementos es ancestro del otro en el árbol generado.

El recorrido *simétrico* o *in-order* (es decir, el recorrido del subárbol izquierdo, luego la raíz y finalmente el subárbol derecho de manera recursiva) del árbol generado por el algoritmo

RandQS es justamente la salida del algoritmo. Para el análisis del algoritmo es importante observar el recorrido *de nivel* o *level-order* (es decir, el recorrido que inicia en la raíz y continúa de izquierda a derecha incrementando el nivel de profundidad, donde el i-ésimo nivel del árbol es el conjunto de nodos cuya distancia hacia la raíz es i). Este recorrido es una permutación que para motivos del análisis se denomina π.

Para calcular P_{ij} es necesario hacer el siguiente par de observaciones:

1. Se hace una comparación entre $S_{(i)}$ y $S_{(j)}$ si y solo si uno (y solamente uno) de estos elementos aparece en la permutación π antes que cualquier elemento $S_{(l)}$ tal que $i < l < j$. Para visualizar esto más claramente supóngase que $S_{(k)}$ es el elemento, dentro del rango $[i, j]$, que aparece primero en π. Si $k \notin \{i, j\}$, entonces $S_{(i)}$ pertenecerá al subárbol izquierdo de $S_{(k)}$, mientras que $S_{(j)}$ pertenecerá al derecho, lo cual implica que $S_{(i)}$ no es comparado con $S_{(j)}$. Ahora bien, cuando $k \in \{i, j\}$ se da una relación *padre-hijo* entre $S_{(i)}$ y $S_{(j)}$, lo cual implica que $S_{(i)}$ y $S_{(j)}$ son comparados entre sí.

2. Cualquiera de los elementos $S_{(i)}, S_{(i+1)}, ..., S_{(j)}$ puede ser seleccionado como la raíz del árbol generado por RandQS, siendo la probabilidad de ser seleccionado la misma para todos estos elementos. En otras palabras, todos los elementos de S dentro del rango $[i, j]$ tienen la misma probabilidad de aparecer en la primera posición de la permutación π. Por lo tanto, la probabilidad de que el elemento $S_{(i)}$ o $S_{(j)}$ sea el primer elemento de la permutación π es exactamente $2/(j - i + 1)$.

Ahora que P_{ij} ha sido calculado es posible determinar el *valor esperado* del número de comparaciones realizadas por el algoritmo RandQS. Retomando las ecuaciones 2.33 y 2.34:

$$\begin{aligned} \sum_{i=1}^{n} \sum_{j>i} P_{ij} = & \sum_{i=1}^{n} \sum_{j>i} \frac{2}{j-i+1} \\ \leq & \sum_{i=1}^{n} \sum_{k=1}^{n-i+1} \frac{2}{k} \\ \leq & \; 2 \sum_{i=1}^{n} \sum_{k=1}^{n} \frac{1}{k} \end{aligned} \tag{2.35}$$

Teorema 2.3.7. *El tiempo de ejecución esperado del algoritmo RandQS es de orden $O(n \log n)$.*

Demostración. La definición del n-ésimo número armónico es $H_n = \sum_{k=1}^{n} 1/k$. Por lo tanto, el valor *esperado* del número de comparaciones realizadas por el algoritmo RandQS está acotado por $2nH_n$. Asimismo, el n-ésimo número armónico H_n es aproximadamente igual a $\ln n + \Theta(1)$. □

Es importante resaltar que el tiempo de ejecución *esperado* es el mismo para cualquier entrada, pues éste depende únicamente de las selecciones aleatorias tomadas por el algoritmo, y no de la distribución de los elementos de la entrada. El comportamiento de un algoritmo aleatorizado tiende a ser diferente entre una ejecución y otra, incluso sobre una misma entrada. En el caso de los algoritmos aleatorizados de tipo *Las Vegas* la variable aleatoria es el tiempo de ejecución, siendo la solución entregada siempre la óptima.

2.3.3. Algoritmos aleatorizados tipo *Monte Carlo*

La principal característica de los algoritmos aleatorizados tipo Monte Carlo es que su tiempo de ejecución es polinomial y **no siempre** encuentran la solución óptima, a diferencia de los algoritmos tipo Las Vegas, que **siempre** la encuentran. En otras palabras: los algoritmos Las Vegas encuentran la solución óptima con una probabilidad exactamente igual a 1, mientras que los algoritmos Monte Carlo encuentran la solución óptima con una probabilidad estrictamente menor a 1. Aparentemente esto representa una desventaja para los algoritmos tipo

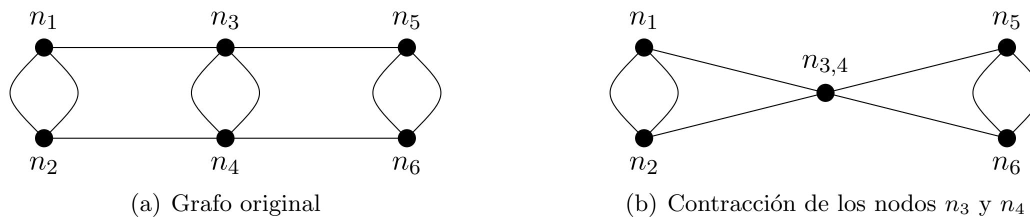
Monte Carlo; sin embargo, considerando que no existen algoritmos polinomiales que resuelvan problemas *NP-Difíciles* (a menos que $P = NP$), los algoritmos aleatorizados tipo Monte Carlo representan un enfoque muy útil (incluso en ocasiones el mejor) para la resolución de este tipo de problemas [40].

A continuación se describe un algoritmo aleatorizado tipo Monte Carlo que resuelve el problema del Corte Mínimo (*min-cut problem*), el cual es un problema que pertenece a la clase de complejidad P. Este algoritmo permite observar que los algoritmos aleatorizados, en comparación con otros algoritmos polinomiales deterministas, tienden a ser muy simples, sobre todo cuando se aprovecha el concepto de *amplificación*, el cual también es definido.

2.3.3.1. Algoritmo para encontrar el Corte Mínimo

Sea G un multigrafo[1] conexo no-dirigido que contiene n nodos. Un *corte* de G es un conjunto de aristas, tales que al removerlas de G el grafo es descompuesto en dos o más componentes. Un *corte-mínimo* es un *corte* de cardinalidad mínima. El problema de encontrar el *corte mínimo* puede ser resuelto en tiempo polinomial con el algoritmo de Ford-Fulkerson [17], el cual aprovecha el concepto de redes de flujo. El algoritmo aleatorizado tipo Monte Carlo que se describe a continuación resuelve con una alta probabilidad el problema del *corte mínimo* en tiempo polinomial [41], siguiendo un enfoque más simple que el del los algoritmos deterministas basados en redes de flujo. El algoritmo consiste en repetir hasta n veces el siguiente paso:

Elegir aleatoriamente una arista (donde todas las aristas tienen la misma probabilidad de ser seleccionadas) y combinar los nodos de la arista en un sólo nodo (véase la Figura 2.10). A este proceso se le denomina *contracción*. Las aristas que hubiera entre los nodos contraídos son eliminadas y las aristas del tipo (n_i, n_x), donde n_i es cualquier arista y n_x es un nodo contraído, son sustituidas por las aristas $(n_i, n_{x'})$, donde $n_{x'}$ es el nodo resultante de la contracción. Con cada contracción el número de nodos del grafo G disminuye en una unidad. Es importante observar que la contracción de alguna arista no disminuye el tamaño del *corte mínimo* de G; lo cual se debe a que todo *corte*, en cualquier etapa intermedia del proceso, es un *corte* del grafo original. El algoritmo continúa contrayendo aristas hasta que quedan solamente 2 nodos en el grafo. El conjunto de aristas que existen entre este par de nodos es un *corte* y es entregado por el algoritmo como un posible *corte mínimo*.



(a) Grafo original

(b) Contracción de los nodos n_3 y n_4

Figura 2.10: Ejemplo de contracción de nodos

Para analizar el comportamiento de este algoritmo es necesario utilizar las siguientes definiciones de la teoría de grafos.

Definición 11. *Para todo nodo v en un multigrafo G, la vecindad de v, denotada $\Gamma(v)$, es el conjunto de nodos de G que son adyacentes a v.*

Definición 12. *El* grado *de v, denotado $d(v)$, es el número de aristas que inciden en v.*

[1]Un multigrafo es un grafo que puede tener más de una arista entre cualquier par de nodos.

Definición 13. *Para un conjunto S de nodos de G, la vecindad de S, denotada $\Gamma(S)$, es la unión de las vecindades de los nodos que lo constituyen.*

Nótese que cuando no hay ciclos de un nodo hacia sí mismo, ni más de una arista entre todo par de nodos, el valor de $d(v)$ es igual a $\Gamma(v)$.

Sea k el tamaño del *corte mínimo* de un grafo G. El mínimo número de aristas que puede tener este grafo es $kn/2$, pues de lo contrario habría al menos un nodo con grado menor a k, siendo sus aristas incidentes justamente un *corte* de tamaño menor a k, lo cual contradice el hecho de que el *corte mínimo* es de tamaño k. Sea C un *corte mínimo* de tamaño k del grafo G. A continuación se calcula la probabilidad de que ninguna arista de C sea contraída durante la ejecución del algoritmo, de manera que al final el algoritmo entregue las aristas de C como solución.

Sea ε_i el evento "no seleccionar una arista de C en el i-ésimo paso", para $1 \leq i \leq n-2$. La probabilidad de que el nodo seleccionado en el primer paso forme parte de C es a lo más $k/(nk/2) = 2/n$, por lo tanto $Pr[\varepsilon_i] \geq 1 - 2/n$. Asumiendo que ε_1 ocurrió en el primer paso, en el segundo paso quedan $k(n-1)/2$ aristas, de modo que la probabilidad de elegir un nodo en C es a lo más $k/(n-1)$, por lo tanto $Pr[\varepsilon_1|\varepsilon_2] \geq 1-2/(n-1)$. En el i-ésimo paso el número de aristas restantes es $n-i+1$. El tamaño del *corte mínimo* sigue siendo aún k, por lo cual el grafo tiene al menos $k(n-i+1)/2$ aristas restantes durante el i-ésimo paso. Por lo tanto:

$$P\left[\varepsilon_i | \bigcap_{j=1}^{i-1} \varepsilon_j\right] \geq 1 - \frac{2}{n-i+1}$$

Con base en la regla de la multiplicación (ecuación 2.24) es posible calcular la probabilidad de no seleccionar ningún nodo en C durante la ejecución del algoritmo:

$$P\left[\bigcap_{i=1}^{n-2} \varepsilon_i\right] \geq \prod_{i=1}^{n-2}\left(1 - \frac{2}{n-i+1}\right) = \frac{2}{n(n-1)} \tag{2.36}$$

La probabilidad de que el algoritmo encuentre el *corte mínimo* es mayor a $2/n^2$. Para valores grandes de n es muy poco probable que el algoritmo encuentre la solución deseada. Sin embargo, es posible incrementar la probabilidad de *éxito* del algoritmo, donde el *éxito* corresponde a encontrar el *corte mínimo*, al repetir muchas veces el algoritmo. A la repetición de un algoritmo, con la intención de incrementar su probabilidad de *éxito* se le denomina *amplificación*. En la siguiente sección se muestra cómo la amplificación permite mejorar el algoritmo descrito en esta sección.

2.3.3.2. Amplificación

La amplificación consiste en repetir un cierto número de veces un algoritmo, con la intención de así incrementar (amplificar) su probabilidad de *éxito*, donde el *éxito* consiste en encontrar la solución óptima.

Supóngase que se desea resolver el problema de "obtener un 1 al lanzar un dado". Aunque este problema quizá no sea muy realista, resulta útil para comprender el concepto de la amplificación. Una manera de resolver este problema consiste en simplemente lanzar el dado. La probabilidad de que el resultado del lanzamiento sea un 1 es de $1/6$ (suponiendo que el dado no está cargado). Una probabilidad de *éxito* de $1/6$ podría no ser muy convincente; afortunadamente puede ser incrementada tan solo con repetir el lanzamiento del dado. Si el dado es lanzado 6 veces, la probabilidad de que en ninguno de los lanzamiento aparezca un 1 es de $(1-1/6)^6 \approx 0.33$. Por lo tanto, la probabilidad de que en alguno de los lanzamiento aparezca

un 1 es de $1 - 0.33 \approx 0.67$. Si el lanzamiento se repite más veces la probabilidad de éxito seguirá incrementándose, de manera que es posible acercarse a una probabilidad tan cercana a 1 como se desee. Por supuesto que el precio que se paga es el de incrementar el número de lanzamientos, lo cual es equivalente a incrementar el tiempo de ejecución.

En la sección anterior se presentó un algoritmo que encuentra el *corte mínimo* con una probabilidad mayor a $2/n^2$. Si este algoritmo es repetido $n^2/2$ veces, la probabilidad de encontrar el *corte mínimo* se incrementa, de modo que es válido decir que el algoritmo ha sido amplificado. Dado que todas las ejecuciones del algoritmo son independientes entre sí, la probabilidad de **no encontrar** el *corte mínimo* en alguna de las $n^2/2$ ejecuciones es de:

$$\left(1 - \frac{2}{n^2}\right)^{n^2/2} < 1/e \approx 0.36 \tag{2.37}$$

De manera que la probabilidad de encontrar el *corte mínimo* usando el algoritmo amplificado es de:

$$P[\acute{e}xito] \geq 1 - 1/e \approx 0.64 \tag{2.38}$$

Al amplificar el algoritmo se logró incrementar su probabilidad de *éxito*, inclusive logrando que esta probabilidad sea independiente del tamaño de la entrada (en este caso n). El algoritmo para encontrar el *corte mínimo* es un claro ejemplo de lo simples que suelen ser los algoritmos aleatorizados con respecto a algunos algoritmos deterministas, pues los algoritmos deterministas conocidos para resolver el problema del *corte mínimo* hacen uso de redes de flujo y son considerablemente más complicados. Asimismo, la simplicidad de los algoritmos aleatorizados facilita su implementación.

2.4. Problema de selección de k-centros

Existen una gran cantidad de problemas que pueden ser catalogados como "problemas de selección de centros", los cuales suelen presentarse de manera cotidiana en las sociedades y en los sistemas en general. Un ejemplo clásico es el de la distribución de estaciones de bomberos en una comunidad, donde el objetivo es minimizar la distancia de las estaciones de bomberos hacia los inmuebles o zonas que pudieran requerir auxilio, con lo cual se garantiza que el servicio brindado es el mejor posible en términos del tiempo de arribo de las unidades. Otro ejemplo es el de la distribución de antenas en el diseño de redes inalámbricas, donde al minimizar la distancia de los clientes hacia el conjunto de antenas omnidireccionales se garantiza que la intensidad de la señal recibida es la mejor posible, lo cual implica brindar un mejor servicio.

Los "problemas de selección de centros" han sido planteados a lo largo de diferentes periodos de la historia, lo cual de alguna manera es un indicador de su importancia. Quizá uno de los problemas de selección de centros más antiguos sea el problema conocido como "*problema de Weber*" (el cual es una versión restringida del problema más general conocido como *k-median*), cuyo planteamiento y resolución ha sido asociado a diferentes matemáticos, entre ellos Battista Cavalieri (1598-1647), Pierre Fermat (1601-1655) y Evarista Torricelli (1608-1647) [10]. Si bien el planteamiento y resolución de los problemas de selección de centros ha estado presente a lo largo de la historia, no es sino hasta el siglo XX, con el surgimiento de las ciencias de la computación y la teoría de la intratabilidad computacional, cuando se destaca la relación entre estos problemas así como la importancia de su resolución óptima, dando lugar a áreas muy específicas dentro de las ciencias de la computación tales como: "Ubicación de Facilidades" (*Facility Location*) [10, 23] o "Agrupamiento" (*Clustering*) [19].

En general un problema de selección de centros puede definirse de la siguiente manera:

- Dado un conjunto de elementos y las relaciones entre sí (lo cual suele modelarse con un grafo), determinar un subconjunto de elementos, denominados centros, tal que se minimice el valor de una función en particular.

Algunos de los problemas básicos atacados por áreas como "Ubicación de Facilidades" son [10]:

- **Problema de Weber** (también conocido como problema: de Fermat, generalizado de Fermat, de Fermat-Torricelli, de Steiner, generalizado de Steiner, de Steiner-Weber, generalizado de Weber, de Fermat-Weber, 1-*median*, *minisum*, del punto de recorrido total mínimo y *median* espacial). Consiste en encontrar un punto (x,y) tal que la suma de las distancias euclidianas de éste hacia un conjunto de n puntos fijos con coordenadas (a,b) sea mínimo.

- ***k-median*** (o *p-median*). Se trata de una generalización del problema de Weber, donde el número de puntos que se desea localizar es igual a k.

- ***k-center*** (también llamado *p-center* o problema de selección de k-centros). Dado un conjunto de nodos y las distancias entre ellos, determinar un subconjunto de nodos de tamaño menor o igual a k, tal que la máxima distancia de todo nodo hacia su centro más cercano sea mínima.

Dado que el problema abordado en la presente tesis es el problema de selección de k-centros, a continuación se le define formalmente y se describen sus características principales.

Las entradas del problema de selección de k-centros son:

1. Un grafo completo no dirigido $G = (V, E)$.

2. Un número entero positivo k.

Son muchas las versiones que existen del problema de selección de k-centros, las cuales dependen de las restricciones asociadas a cada variable involucrada. Por ejemplo, la versión *discreta* restringe el contenido de la solución a nodos del grafo de entrada, la versión *continua* permite que un nodo posicionado en cualquier punto del espacio forme parte de la solución, la versión *absoluta* restringe el contenido de la solución a nodos posicionados únicamente sobre cualquier punto de las aristas; a su vez, cada una de estas versiones puede corresponder a la versión *con peso* o *sin peso*, donde en la versión *sin peso* todos los nodos son tratados de la misma manera, mientras que en la versión *con peso* la distancia de todo nodo hacia cada centro es multiplicada por el costo asociado a cada nodo [10]. Otras tantas versiones existen, como aquellas relacionadas a las características de las aristas, tales como la versión *métrica* o *métrica asimétrica*. Las propiedades de un espacio métrico (V, d), donde V es un conjunto de puntos y $d : V \times V \rightarrow \mathbb{R}^+$, son las siguientes [53]:

- $\forall x \in V,\ d(x, x) = 0$

- (*simetría*) $\forall x, y \in V,\ d(x, y) = d(y, x)$

- (*desigualdad del triángulo*) $\forall x, y, z \in V,\ d(x, y) + d(y, z) \geq d(x, z)$

La versión que se aborda en la presente tesis es la más elemental de las mencionadas, es decir, el problema de selección de k-centros en su versión *discreta*, *métrica* y *sin peso*. Dado que se trata de la versión más elemental, es común que en la literatura se omitan estos adjetivos.

El problema de selección de k-centros en su versión *discreta*, *métrica* y *sin peso* (a lo largo del presente documento se omitirán estos adjetivos) consiste en, dado un grafo $G = (V, E)$ y un entero positivo k, encontrar un conjunto de centros $C \subseteq V$ de cardinalidad menor o igual a k, tal que la máxima distancia de todo nodo hacia su centro más cercano sea mínima, es decir, se desea minimizar la expresión:

$$max_{v \in V} \ min_{c \in C} \ distancia(v, c) \tag{2.39}$$

Al valor correspondiente a la expresión 2.39 se le suele llamar *radio de cobertura*, denotado por $r(C)$, pues representa la máxima distancia que existe de cualquier nodo hacia su centro más cercano.

El problema de selección de k-centros es NP*-Difícil*, por lo cual no existe un algoritmo que lo resuelva de manera óptima en tiempo polinomial (a menos que $P = NP$). Asimismo, encontrar soluciones C con un tamaño $r(C) < 2 \cdot r(C^*)$ (donde C^* es la solución óptima) también es un problema NP*-Difícil*. En la siguiente sección se demuestra lo planteado en este párrafo.

2.4.1. Complejidad del problema de selección de k-centros

Para demostrar que el problema de selección de k-centros es NP*-Difícil* es necesario demostrar antes el Lema 2.4.1.

Lema 2.4.1. *El problema de determinar la existencia en $G = (V, E)$ de un conjunto dominante de cardinalidad menor o igual a k es NP*-Completo *[2]. Este problema es conocido como el problema del Conjunto Dominante (Dominating Set) y consiste en determinar un subconjunto $V' \subseteq V$ (de cardinalidad menor o igual a k), tal que todo nodo en V esté en V' o sea adyacente a dicho conjunto.*

Demostración. La demostración es por reducción polinomial del problema NP*-Completo* conocido como problema de Cobertura de Vértices (*Vertex Cover*) al problema del Conjunto Dominante (*Dominating Set*).

El problema de Cobertura de Vértices consiste en: dado un grafo $G = (V, E)$ y un entero k, $1 < k < n$, encontrar un conjunto S de nodos de cardinalidad menor o igual a k, tal que $\forall (u, v) \in E, u \vee v \in S$. Este problema es NP*-Completo* [6].

La reducción polinomial consiste en: dado un grafo de entrada $G = (V, E)$ para el problema de Cobertura de Vértices, se construye un grafo G' agregando al grafo G un nodo $v_{x,y}$ por cada arista $(x, y) \in E$. Además se agregan las aristas $(x, v_{x,y})$ y $(y, v_{x,y})$ al conjunto E. La construcción del grafo G' es tal que es posible demostrar que:

$$\begin{gathered} G \text{ tiene un } \textit{Vertex Cover}\ ^2\text{de cardinalidad menor o igual a } k \\ \longleftrightarrow \\ G' \text{ tiene un } \textit{Dominating Set}\ ^3\text{de cardinalidad menor o igual a } k \end{gathered} \tag{2.40}$$

A continuación se demuestra cada una de las implicaciones que conforman esta doble implicación.

[2] Un *Vertex Cover* representa una solución al problema del mismo nombre.
[3] Un *Dominating Set* representa una solución al problema del mismo nombre.

1. Si G tiene un *Vertex Cover* de cardinalidad menor o igual a k, entonces G' tiene un *Dominating Set* de cardinalidad menor o igual a k.

   Demostración: Sea C un *Vertex Cover* de cardinalidad k para el grafo G . Dado que toda arista en G tiene un nodo incidente en C, los vértices de G que están en G' son dominados por los vértices en C. Para toda arista (u, v), $u \vee v \in C$, de modo que todos los vértices $v_{x,y}$ agregados al grafo G' son dominados por los vértices de C. Por lo tanto, todos los vértices en G' son dominados, lo que significa que C es un *Domintaing Set* de G'.

2. Si G' tiene un *Dominating Set* de cardinalidad menor o igual a k, entonces G tiene un *Vertex Cover* de cardinalidad menor o igual a k.

   Demostración: Sea D un *Dominating Set* de cardinalidad k para el grafo G' . Existe la posibilidad de que D contenga únicamente vértices del grafo G o bien de que además contenga vértices del grafo G' que no existen en G. En el primer caso todos los vértices $v_{x,y}$ nuevos (es decir, que no existen en G) son dominados por D, lo cual implica que $x \vee y \in D$; dado que existe un nodo $v_{x,y}$ por cada arista en G, el conjunto D es un *Vertex Cover* para G. En el caso de que D contenga nodos que no existen en G es posible cambiar cada nodo $v_{x,y}$ por un nodo $x \vee y \in G$, hecho lo cual D seguirá siendo un *Dominating Set*, pues el nodo $v_{x,y}$ sigue siendo dominado, al igual que los nodos $x \wedge y$; de manera que el conjunto D es un *Vertex Cover* para G.

□

Teorema 2.4.2. *El problema de selección de k-centros es NP-Difícil [2].*

Demostración. La demostración es por reducción polinomial del problema de determinar si existe un Conjunto Dominante (*Dominating Set*) de cardinalidad menor o igual a k, al problema de selección de k-entros.

Dado un grafo $G = (V, E)$ y un entero k, determinar si existe un *Dominating Set* de tamaño menor o igual a k. Para resolver este problema se construye un grafo G' que será la entrada al problema de selección de k-centros; este grafo es idéntico al grafo G, con la excepción de que cada arista de G' tiene asociado un costo de 1. Una vez construido el grafo G' se resuelve el problema de selección de k-centros sobre él. Si la solución óptima al Problema de Selección de k Centros tiene un tamaño de 1 entonces existe un *Dominating Set* de tamaño menor o igual a k en G. Es decir, si el radio de cobertura de la solución óptima al problema de selección de k-centros sobre el grafo G' es exactamente igual a 1, entonces todos los nodos asignados a cada centro son dominados por dicho centro, pues de lo contrario el radio de cobertura tendría que ser mayor a 1. Por lo tanto, la solución al problema de selección de k-centros en G' con radio de cobertura 1 define un *Dominating Set* en G. Si el radio de cobertura sobre G' es mayor a 1, entonces no existe un *Dominating Set* de tamaño menor o igual a k en G. □

Como ya se ha mencionado anteriormente, no solo el problema de selección de k-centros es *NP-Difícil*, sino que también lo es encontrar una ρ-aproximación, donde $\rho < 2$. Esta afirmación se demuestra en el Teorema 2.4.3.

Teorema 2.4.3. *Encontrar una solución con un factor de aproximación estrictamente menor a 2 para el problema seleccion de k-centros es NP-Difícil [3, 19].*

Demostración. La demostración es por reducción del problema de determinar si existe un Conjunto Dominante (*Dominating Set*) de tamaño menor o igual a k hacia el problema de selección de k-centros. La reducción consiste en, dado un grafo $G = (V, E)$ y un entero k para

el problema del Conjunto Dominante, construir un grafo completo G', el cual consta de los mismos nodos y aristas que G, a excepción de las nuevas aristas que permiten que G' sea un grafo completo. Se asocia a cada arista de G' un peso, de la siguiente manera:

$$w(e) = \begin{cases} 1 & si\ e \in E \\ 2 & si\ e \notin E \end{cases} \tag{2.41}$$

Los pesos de las aristas del grafo G' satisfacen la desigualdad del triángulo, pues si el peso de una arista (u, v) es igual a 1, entonces este peso será menor o igual a la suma de los pesos de las aristas de cualquier otra trayectoria entre u y v, ya que el peso mínimo de cualquier otra trayectoria es igual a 2. Si el peso de una arista (u, v) es igual a 2, resulta claro que este peso también es menor o igual que la suma de los pesos de las aristas de cualquier otra trayectoria entre u y v, que como ya se mencionó tiene un costo al menos igual a 2.

La reducción planteada satisface las siguientes condiciones:

- Si $dom(G) \leq k$ entonces G' tiene un *k-center* [4] de costo 1.
- Si $dom(G) > k$ entonces el costo óptimo de un *k-center* en G' es igual 2.

donde $dom(G)$ es un *Dominating Set* de G.

Para el primer caso, donde existe un *Dominating Set* de cardinalidad menor o igual a k en G ($dom(G) \leq k$), resulta claro que una ρ-aproximación (donde $\rho < 2$) para el problema de selección de k-centros resuelve el problema de decisión del Conjunto Dominante, pues cualquier ρ-aproximación (donde $\rho < 2$) sólo podrá tener un tamaño de 1. Por lo tanto, utilizando este hipotético algoritmo que resuelve el problema de selección de k-centros con un factor de aproximación de ρ, donde $\rho < 2$, se puede distinguir entre las dos posibilidades, resolviendo así el problema del Conjunto Dominante. □

2.4.2. Aplicaciones y clasificaciones del problema de selección de k-centros

Algunas de las aplicaciones potenciales del problema de selección de k-centros son:

- Ubicación de servicios de emergencia [35] como hospitales, estaciones de bomberos, estaciones de policía,etc.
- Ubicación de servicios para redes de computadoras [23].
- Ubicación de centros de distribución [23].
- Ubicación de facilidades como parques u hoteles [23].
- Ubicación y asignación de códigos postales y estaciones de transporte [23].
- Ubicación de servicios de salud como unidades de atención a diabéticos [26].
- Ubicación de antenas en redes inalámbricas.
- Diseño de servicios de emergencia de gran escala [32] para atención a eventualidades como ataques terroristas, epidemias.
- Ubicación de escuelas [31].

Una forma interesante de clasificar los problemas de selección (o ubicación) de centros es la propuesta por Marianov y Serra [10]:

[4] Un *k-center* es una solución al problema del mismo nombre.

- **Sector público**. En este tipo de aplicaciones el objetivo es brindar servicios de manera universal, es decir, se pretende que toda la población tenga acceso a los servicios (servicios de salud, acceso a la educación, etcétera). Además de buscar la universalidad, también es importante reducir al máximo los costos, ya que estos corren a cargo de la población misma, a través del pago de impuestos. Una manera de reducir los costos consiste en minimizar el número de centros (ya sean hospitales, estaciones de bomberos, escuelas, etcétera), pues así la inversión es mínima y por lo tanto también lo son los costos de mantenimiento (a reserva de las consideraciones y variables que el modelo utilizado no contemple). Otros objetivos de las aplicaciones del sector público son la eficiencia y la igualdad, es decir, brindar servicios adecuados y de manera indiscrimada a toda la población.

- **Sector privado**. En las aplicaciones del sector privado los objetivos son diferentes a los del sector público. Los objetivos del sector privado son básicamente la maximización de los beneficios y la captura del mercado de la competencia.

Claramente cualquier problema planteado puede corresponder a cualquiera de estas clasificaciones, pues básicamente esto depende del sector que plantee el problema y de las funciones que desee maximizar o minimizar. Sin embargo, el problema de selección de k-centros pareciera corresponder de una manera más natural al grupo de problemas del sector público.

2.5. Discusión

- Algunas de las principales clases de complejidad manejadas en las ciencias de la computación son: P, NP, *NP-Completo* y *NP-Difícil*. Los problemas en P se pueden resolver en tiempo polinomial, mientras que los problemas en NP se pueden *verificar* en tiempo polinomial, lo cual implica que la clase P es un subconjunto de la clase NP (es decir, todo problema en P forma parte también de la clase NP). Un problema abierto es el de determinar si $P = NP$ o $P \neq NP$. Los problemas *NP-Completos* son aquellos problemas que forman parte de NP y cuya resolución permite resolver cualquier otro problema en NP a través de una reducción (o transformación) polinomial. Los problemas *NP-Difíciles* son aquellos que, en el mejor de los casos, son *NP-Completos*. Se dice que un problema es intratable cuando no se conoce ningún método eficiente (es decir, cuyo tiempo de ejecución sea polinomial) para su resolución óptima; los problemas *NP-Difíciles* (lo cual incluye a los *NP-Completos*) son intratables.

- Dada la imposibilidad de resolver eficientemente un problema *NP-Difícil* (en caso de que $P \neq NP$), surge el diseño de algoritmos de aproximación, los cuales se ejecutan en tiempo polinomial pero a costa del sacrificio de la optimalidad.

- Una alternativa interesante para el diseño de algoritmos es la de los algoritmos aleatorizados, los cuales se caracterizan por tomar decisiones aleatorias durante su ejecución. Si bien el uso de la aleatoriedad pareciera ser una desventaja, en realidad permite resolver eficientemente (con una determinada probabilidad) gran cantidad de problemas, incluyendo problemas *NP-Difíciles*.

Capítulo 3

Trabajos relacionados

3.1. Algoritmos de aproximación para el problema de selección de k-centros

Como respuesta ante la imposibilidad de resolver gran cantidad de problemas de manera óptima en tiempo polinomial (en caso de que $P \neq NP$), surge el desarrollo de los algoritmos de aproximación. Los algoritmos de aproximación permiten resolver problemas NP-*Difíciles* en tiempo polinomial, aunque no necesariamente de manera óptima. Es decir, se sacrifica la optimalidad con la intención de incrementar la eficiencia.

El problema de selección de k-centros pertenece a la clase NP-*Difícil*, lo cual implica que no es posible resolverlo de manera óptima en tiempo polinomial. Más aún, encontrar una ρ-aproximación, para $\rho < 2$, también es un problema NP-*Difícil* (a menos que $P = NP$). Dada la naturaleza del problema de selección de k-centros, el mejor algoritmo de aproximación que se puede diseñar (si $P \neq NP$) es aquel que entregue una 2-aproximación. Los algoritmos presentados en esta sección encuentran justamente una 2-aproximación en tiempo polinomial, lo cual los convierte en los *mejores algoritmos (polinomiales) posibles*.

Aunque en teoría los algoritmos de aproximación presentados en esta sección son los *mejores posibles*, existen otros algoritmos (clasificados como heurísticas y metaheurísticas) que, a pesar de no dar ninguna garantía teórica de su *buen* desempeño, tienden a entregar *buenas* aproximaciones en una cantidad de tiempo *razonable*.

3.1.1. Algoritmo de González

En 1985 T. González propone un algoritmo que permite resolver, con un factor de aproximación de 2, el problema de la minimización de la máxima distancia entre parejas de nodos dentro de agrupamientos (*pairwise clustering*) [1]. Este problema es planteado por González de la siguiente manera:

Dado un grafo $G = (V, E, W)$ y un valor entero k, encontrar un k-split de V tal que se minimice el valor de $max\{M_1, M_2, \ldots, M_k\}$, donde $G = (V, E, W)$ es un grafo no dirigido definido por un conjunto de nodos V, un conjunto de aristas E y una función de peso $W : E \to R_0^+$ (conjunto de los reales no negativos) que respeta la desigualdad del triángulo. Un k-split $(B_1, B_2, \ldots, B_k)$ es una partición de tamaño k del conjunto de vértices V. Los conjuntos B_i de un k-split son denominados clusters o agrupamientos. La función objetivo $f : B_1, B_2, \ldots, B_k \to R_0^+$ de tipo $max\{M_1, M_2, \ldots, M_k\}$ es aquella donde M_i es el peso máximo de una arista cuyos vértices pertenecen al cluster B_i.

González se refiere a este problema como un problema de tipo k-tMM, donde k indica que se desean encontrar k agrupamientos, t representa un grafo que satisface la desigualdad del

triángulo y MM indica que el problema es de tipo MinMax (minimización del máximo valor de una función objetivo).

El problema planteado por González no pareciera correponder al problema de selección de k-centros; sin embargo son el mismo problema. De hecho, otro nombre con el cual se conoce al problema de selección de k-centros en su version *continua* es el de *pairwise clustering* [5,9].

Algoritmo 3.1: Algoritmo de González

```
    Entrada: un grafo G = (V, E) y un entero k
    Salida: una partición (B_1, B_2, ..., B_k) de V
 1  Sea {v_1, ..., v_n} el conjunto de elementos a agrupar ;
 2  head_1 ← v_1 ;
 3  B_1 ← {v_1, ..., v_n} ;
 4  for l = 1 to k − 1 do
 5  |   h ← max{W(head_j, v_i) | v_i ∈ B_j ∧ 1 ≤ j ≤ l};
 6  |   Sea v_i uno de los nodos cuya distancia al head del cluster al que pertenece es h ;
 7  |   Movemos v_i a B_{l+1};
 8  |   head_{l+1} ← v_i;
 9  |   foreach v_t ∈ (B_1 ∪ B_2 ∪ ... ∪ B_l) do
10  |   |   Sea j tal que v_t ∈ B_j;
11  |   |   if W(v_t, head_j) ≥ W(v_t, v_i) then
12  |   |   |   Movemos v_i de B_j a B_l;
13  |   |   end
14  |   end
15  end
```

El algoritmo de González consta básicamente de dos etapas:

1. Fase de inicialización.

2. $k-1$ fases de expansión.

En la fase de inicialización se asignan todos los elementos de S al conjunto único B_1 (primer *cluster* formado). Se elige arbitrariamente a alguno de sus elementos como *cabeza* del *cluster* ($head_1$).

En las siguientes j-ésimas fases de expansión algunos elementos de los *clusters* $(B_1, ..., B_j)$ son movidos al *cluster* B_{j+1}. Alguno de los elementos del nuevo *cluster* será etiquetado como *cabeza* de dicho *cluster* ($head_{j+1}$) de la manera que se describe en el siguiente párrafo.

El nuevo *cluster* que se forma en cada j-ésima fase de expansión se construye identificando primero un nodo v_i en alguno de los *clusters* $(B_1, ...B_j)$ cuya distancia al $head_i$ del *cluster* B_i al que pertenece sea máxima. Dicho nodo es cambiado de su *cluster* actual al nuevo *cluster* B_{j+1} y es etiquetado como $head_{j+1}$. Una vez creado el *cluster* B_{j+1} se trasladan a dicho *cluster* todos los nodos en $(B_1, ..., B_j)$ tales que su distancia a $head_{j+1}$ no sea mayor que su distancia al $head_i$ del conjunto B_i al que pertenecen.

Para demostrar que este algoritmo entrega una 2-aproximación es necesario hacer referencia al Lema 3.1.1, el cual hace uso de la definición de *clique* que se muestra a continuación.

Sea V un conjunto de puntos que se desea agrupar, y sea C un subconjunto de V. Se asume que $|V| > k$, pues de lo contrario la solución es trivial. Se dice que C forma un $(k+1)$-*clique* de peso h si la cardinalidad del conjunto C es $k+1$ y cada par de elementos distintos en C están a una distancia de al menos h entre sí.

Sea C^* la solución óptima a una instancia V del problema *k-tMM* que, como ya se mencionó, es otra manera de referirse al problema de selección de k-centros. Sea $t(C^*)$ el tamaño de la solución C^*, es decir, el tamaño de la arista más pesada dentro del *k-split* óptimo.

Lema 3.1.1. *Si existe un* $(k+1)$-clique *de peso* h *para* V*, entonces* $t(C^*) \geq h$

Demostración. La cardinalidad del $(k+1)$-*clique* de peso h es $k+1$, donde k es el número de *clusters* que se desea formar. Si al menos un par de elementos del $(k+1)$-*clique* de peso h están en uno de los k *clusters*, entonces la arista de mayor peso en dicho *cluster* tendrá un valor mayor o igual a h. Resulta evidente que, por el principio del nido de paloma, existirá al menos un *cluster* conteniendo al menos dos vértices del $(k+1)$-*clique* de peso h. □

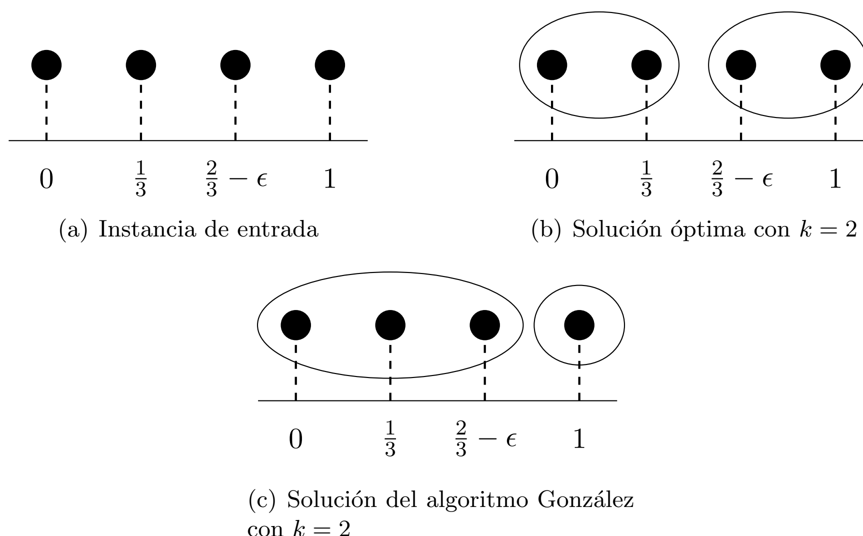
Teorema 3.1.2. *El algoritmo de González genera una solución con un valor de la función objetivo menor o igual a* $2 \cdot t(C^*)$.

Demostración. Sea $v_i \in B_j$, $1 \leq j \leq k$, un nodo cuya distancia a $head_j$ sea máxima. Llámese h a dicha distancia. Resulta claro que los pesos de todas las aristas en B_j son menores o iguales a h, por lo tanto, el valor de la función objetivo (que es el peso mayor de alguna arista en B_j) sólo podrá tener un valor menor o igual a $(2 \cdot h)$ ya que las instancias del problema *k-tMM* respetan la desigualdad del triángulo. Dada la naturaleza del algoritmo, podemos notar que durante el tiempo de ejecución la distancia del nodo v_i hacia el *head* del *cluster* al cual pertenecía anteriormente debe ser al menos h. Esto, aunado al hecho de que v_i nunca es etiquetado como *head* de un nuevo *cluster*, implica que, cada vez que un nuevo cluster se crea, la distancia de su *head* a los *heads* de los clusters creados anteriormente es mayor o igual a h, i.e. $W(head_p, head_q) \geq h$ para $p \neq q$. Sea $T = \{head_1, ..., head_k, v_i\}$. Resulta claro que T es un $(k+1)$-*clique* de peso h, lo cual implica (por el lema 3.1.1) que $t(C^*) \geq h$. El *k-split* entregado por el algoritmo (llámese C a esta solución) consta de conjuntos donde las aristas tienen pesos menores o iguales a $(2 \cdot h)$, es decir, $t(C) \leq 2 \cdot h$. Uniendo este último par de resultados ($t(C^*) \geq h$ y $t(C) \leq 2 \cdot h$) queda demostrado el teorema: $t(C) \leq 2 \cdot h \leq 2 \cdot t(C^*)$, por lo tanto $t(C) \leq 2 \cdot t(C^*)$ □

El factor de aproximación de este algoritmo no puede ser disminuido, pues existen instancias donde el factor de aproximación obtenido es exactamente igual a 2. En la Figura 3.1 se aprecia justamente una instancia para la cual la solución óptima con $k = 2$ tiene un valor de $\frac{1}{3}$; sin embargo el algoritmo González encuentra una solución de tamaño $\frac{2}{3} - \varepsilon$. Dado que el problema *k-tMM* (o de la *minimizacion de la máxima distancia inter-cluster*) no es más que un nombre diferente para el problema de selección de k-centros continuo, el algoritmo de González encuentra soluciones que son una 2-aproximación para el problema de seleccion de k-centros en sus versiones tanto *continua* como *discreta*.

Como se puede observar, el papel de los nodos *head* dentro de cada agrupamiento es el de ayudar a determinar los elementos que conforman cada agrupamiento, mas no son parte de la solución, pues al finalizar el algoritmo éste entrega un conjunto de k agrupamientos y no un conjunto de k centros. Sin embargo, es posible considerar a cada nodo *head* como un centro, lo cual comúnmente es asumido por los investigadores que han caracterizado experimentalmente el desempeño de este algorimo [15, 16].

El algoritmo de González entrega a la salida una partición, por lo tanto es necesario construir un conjunto de centros con base en dicha partición. Como ya se mencionó, normalmente se considera a los nodos *head* como centros. Sin embargo, es posible construir el conjunto de centros utilizando algún otro método, por ejemplo, resolviendo el problema de selección de 1-centro sobre cada agrupamiento (en el Capítulo 5 se utiliza esta idea, a la cual se le nombra variante A).



(a) Instancia de entrada

(b) Solución óptima con $k = 2$

(c) Solución del algoritmo González con $k = 2$

Figura 3.1: La cota de aproximación del algoritmo de González es estrecha

Teorema 3.1.3. *Una solución al problema de selección de k-centros, con un factor de aproximación igual a 2, puede construirse al resolver el problema de selección de 1-centro sobre cada cluster definido por el algoritmo de González.*

Demostración. Por el Teorema 3.1.2: $t(C) \leq 2 \cdot t(C^*)$, donde C es la solución entregada por el algoritmo de González y C^* es la solución óptima (recuérdese que estas soluciones son particiones). Si se resuelve el problema de selección de 1-centro sobre cada *cluster* se tendrá que hacer una reasignación de nodos a centros (cada nodo será asignado hacia su centro más cercano), dando lugar a una nueva partición. Todo nodo v_i dentro de un *cluster* B_i podrá permanecer en el mismo *cluster* o bien será asignado a un nuevo *cluster*. Si permanece en el mismo *cluster*, claramente su distancia hacia el *centro* del mismo será menor o igual a $(2 \cdot h)$. Un nodo v_i será asignado a un nuevo *cluster* cuando su distancia hacia el *centro* de éste sea menor que su distancia hacia el *centro* de su *cluster* original. Por lo tanto, ya sea que los nodos cambien o no de conjunto, la distancia que hay de ellos hacia su centro más cercano será menor o igual a $(2 \cdot h)$, que es menor o igual a $2 \cdot t(C^*)$. □

La complejidad de este algoritmo es $O(kn)$, lo cual se puede confirmar observando directamente el Algoritmo 3.1, donde la iteración más externa se ejecuta k veces y la iteración interna se ejecuta hasta $|\{B_1 \cup B_2 \cup \ldots \cup B_t\}|$ veces, donde $\{B_1 \cup B_2 \cup \ldots \cup B_t\}$ es una partición de V y por lo tanto tiene cardinalidad igual a $|V| = n$.

3.1.2. Algoritmo de Hochbaum y Shmoys

En 1985 Hochbaum y Shmoys proponen un algoritmo para resolver el problema de selección de k-centros discreto con un factor de aproximación igual al del algoritmo de González, es decir con un factor de 2. Este algoritmo hace uso de una serie de conceptos que posteriormente se convertirían en elementos básicos para el diseño de algoritmos, tales como la *poda paramétrica*, el *cuadrado de un grafo* (G^2) y *grafo de cuello de botella* (*bottleneck graph* o *grafo-W*). La poda paramétrica consiste en eliminar las partes irrelevantes de la entrada, para lo cual se debe conocer el tamaño de la solución óptima; sin embargo, encontrar el costo (o tamaño) de una solución óptima es justamente la dificultad principal de los problemas *NP-Difíciles*

(para encontrar este costo, tendríamos que encontrar la solución misma). La técnica de poda paramétrica sortea esta dificultad al hacer suposiciones sobre el tamaño de la solución óptima.

Definición 14. *El método de **poda paramétrica** consiste en: dado un problema y una entrada G, un parámetro t es elegido, el cual puede ser considerado como una suposición del costo de la solución óptima. Para cada valor de t, la instancia de entrada G es* podada *al remover las partes de la misma que no serán utilizadas en una solución de costo menor o igual a t.*

Para aprovechar la técnica de poda paramétrica, el algoritmo de Hochbaum-Shmoys requiere de la construcción de grafos-W.

Definición 15. *Un **grafo-W** o **grafo de cuello de botella** de $G = (V, E)$ se define como: $G(W) = (V, E_W)$, donde $e \in E_W$ si y solo si $w_e \leq W$, donde w_e es el peso de la arista e.*

Otro concepto importante que es presentado por Hochbaum y Shmoys es el de **cuadrado de un grafo** (G^2).

Definición 16. *Dado un grafo*

$$G = (V, E) \tag{3.1}$$

el grafo G^2 se construye de la siguiente manera:

$$G^2 = (V, E^2) \tag{3.2}$$

donde $e \in E^2$ si y solo si $e = (u, v) \in E \vee \exists t \in V$ tal que $(u, t) \in E$ y $(t, v) \in E$. De manera alternativa, G^2 se define como un grafo que contiene aristas de la forma (u, t) si y solo si existe una trayectoria de tamaño a lo más 2 entre u y t en G.

El algoritmo propuesto por Hochbaum y Shmoys hace uso de la poda paramétrica, de los grafos-W y de la construcción de grafos al cuadrado, de modo que resulta necesario presentar la siguiente información.

Lema 3.1.4. *Sea un grafo $G(W) = (V, E_W)$, correspondiente a un grafo completo $G = (V, E)$, y un valor específico de W. Sea $G(W)^2 = (V, E_W^2)$ el cuadrado de $G(W)$. Si $e \in E_W^2$ entonces $e \in E_{2W}$, donde $G(2W) = (V, E_{2W})$.*

Demostración. Dado que los pesos de las aristas satisfacen la desigualdad del triángulo, resulta claro que para toda arista $e \in E_w^2$, $w_e \leq 2W$, lo cual implica que $e \in E_{2W}$. □

La relación entre el problema de selección de k-centros y el problema del Conjunto Dominante (*Dominating Set*) es muy estrecha (véase el Capítulo 2). De hecho, resulta fácil observar que resolver el problema de selección de k-centros es equivalente al de encontrar el valor mínimo de W para el cual $G(W)$ tiene un Conjunto Dominante de cardinalidad menor o igual a k. A este Conjunto Dominante se le denomina C^* y al tamaño de la correspondiente solución óptima al problema de selección de k-centros se le denomina $w(C^*)$ (nótese que $w(C^*)$ representa el radio de cobertura $r(C^*)$; para evitar confusión con la notación se trabajará en esta sección únicamente con el término $w(C^*)$). Al grafo correspondiente a este valor mínimo de W se le conoce como *grafo de cuello de botella*, $G_B = G(w(C^*))$. Sin embargo, como ya se mencionó anteriormente, el problema del Conjunto Dominante es *NP-Completo*, de manera que esta reducción no es del todo útil.

El Lema 3.1.5 es particularmente interesante, pues muestra la relación entre el problema de selección de k-centros y el problema del Conjunto Dominante con la definición de grafo al cuadrado. Nótese que, dado un Conjunto Dominante C_W de cardinalidad k del grafo $G(W)$ (si dicho Conjunto Dominante existe), puede considerarse a C_W como una solución al problema de selección de k-centros cuyo radio de cobertura es a lo más W.

Lema 3.1.5. *Sea $W \leq w(C^*)$. Si C es un conjunto de cardinalidad menor o igual a k y además es un Conjunto Dominante en el grafo $G(W)^2$, entonces C es una solución para el problema de selección de k-centros con costo $w(C) \leq 2W \leq 2 \cdot w(C^*)$.*

Demostración. La demostración se deriva directamente del lema 3.1.4 y de la observación hecha en el párrafo anterior. □

Ahora bien, sea G^2 el cuadrado de un grafo $G = (V, E)$. Sea C^{2^*} un Conjunto Dominante de tamaño mínimo para G^2. Dado que todo conjunto dominante de G es un conjunto dominante de G^2, $|C^{2^*}| \leq |C^*|$. La estrategia utilizada por el algoritmo de Hochbaum y Shmoys consiste en determinar una solución C que sea un conjunto dominante de G^2 y que cumpla la siguiente condición:

$$|C^{2^*}| \leq |C| \leq |C^*| \tag{3.3}$$

Para cumplir con la primera condición ($|C^{2*}| \leq |C|$) el algoritmo de Hochbaum y Shmoys entrega una solución C justamente igual a C^{2*} ($C^{2*} = C$); es decir, la solución C entregada es un conjunto dominante de G^2. La siguiente condición a cumplir es $|C| \leq |C^*|$. Para cumplir esta condición es necesario introducir el concepto de Conjunto Fuertemente Estable (*Strong Stable Set*), tambien llamado Conjunto Fuertemente Independiente (*Strong Independent Set*), el cual se define como: conjunto de vértices C tal que para todo $u \in V$, $|N_g(u) \cap C| \leq 1$, donde $N_g(u)$ es el conjunto de vecinos del vértice u. En otras palabras, un Conjunto Fuertemente Estable C es un Conjunto Independiente (es decir, $\forall a, b \in C, (a, b) \notin E$) tal que todo nodo en $V - C$ está conectado a lo más a un nodo de C. El problema de encontrar el Conjunto Fuertemente Estable Máximo, es decir el de mayor cardinalidad posible, es equivalente al de encontrar un Conjunto Dominante Mínimo. De hecho el problema del Conjunto Fuertemente Estable Máximo es el dual del problema del Conjunto Dominante Mínimo [7, 8], donde ambos problemas son, respectivamente, un problema de programación lineal de la forma:

$$\begin{aligned} \textit{minimizar} \quad & z = cx \\ \textit{sujeto a} \quad & Ax \geq b \\ & x \geq 0 \end{aligned} \tag{3.4}$$

$$\begin{aligned} \textit{maximizar} \quad & w = yb \\ \textit{sujeto a} \quad & yA \leq c \\ & y \geq 0 \end{aligned} \tag{3.5}$$

El teorema de la dualidad débil en programación lineal establece que para toda x factible para 3.4 y para toda y factible para 3.5, $cx \geq yb$ [33]. Con base en este teorema, y dado que los problemas del Conjunto Fuertemente Estable Máximo y el del Conjunto Dominante Mínimo son duales, queda demostrado el siguiente lema:

Lema 3.1.6. *Dado un grafo $G = (V, E)$. Sea $C_E \subseteq V$ un Conjunto Fuertemente Estable factible, y sea C^* un Conjunto Dominante Mínimo de G. Entonces $|C_E| \leq |C^*|$.*

Como se mencionó antes, el algoritmo de Hochbaum y Shmoys busca entregar soluciones que satisfagan la desigualdad 3.3. La primera condición $|C^{2*}| \leq |C|$ se logra al encontrar un conjunto dominante de G^2. Ahora bien, para cumplir con la segunda condición se hace uso

del lema 3.1.6. Es decir, el algoritmo de Hochbaum y Shmoys garantiza que la desigualdad 3.3 se cumple al entregar una solución que es un conjunto dominante de G^2 y que además es un conjunto fuertemente estable de G. Sin embargo los problemas del conjunto dominante y del conjunto estable son *NP-Difícil*, es por ello que en realidad no se trabaja sobre los grafos G y G^2, sino sobre grafos-W, $G(W)$, que se construyen con el método de poda paramétrica.

En síntesis, cualquier conjunto C que sea un conjunto fuertemente estable de $G(W)$ (y que además sea un conjunto dominante de $G(W)^2$, cumpliendo así con la condición 3.3), cuya cardinalidad sea menor o igual a k, $|C| \leq k$, y $W \leq w(C*)$ es una 2-aproximación por el lema 3.1.5.

El algoritmo propuesto por Hochbaum y Shmoys cumple con todas las condiciones impuestas por el Lema 3.1.5, de manera que entrega soluciones que son una 2-aproximación al problema de selección de k-centros. Antes de mostrar el algoritmo, y para facilitar la demostración de lo afirmado en este párrafo, es necesario presentar el siguiente lema:

Lema 3.1.7. *Sea C un conjunto fuertemente estable de $G = (V, E)$. Si $x \in V$ no es dominado por C en G^2 entonces $C \cup x$ es un conjunto fuertemente estable de G.*

Demostración. Supóngase que $C \cup x$ no es un conjunto fuertemente estable, lo cual significa que existe un vértice u tal que $U = N_G(u) \cap (C \cup \{x\})$ contiene al menos dos vértices, uno de los cuales es x. Sin embargo, esto implica que x está dominado en G^2 por el resto de vértices en U (los cuales son elementos de C), lo cual contradice el hecho de que $x \in V$ no es dominado por C en G^2. □

Las entradas al algoritmo de Hochbaum y Shmoys (Algoritmo 3.2) son un grafo completo no dirigido $G = (V, E)$ y un entero k, donde $E = \{e_1, e_2, ..., e_m\}$ y cada arista tiene asociado un peso w_e. Se asume que las aristas están ordenadas de tal manera que $w_{e_1} \leq w_{e_2} \leq \cdots \leq w_{e_m}$ y el grafo es representado con una lista de adyacencias, donde los vértices adyacentes están ordenados de forma creciente con respecto a su peso [1].

Teorema 3.1.8. *El algoritmo de Hochbaum y Shmoys entrega una solución C tal que $w(C) \leq 2w(C^*)$ en $O(n^2 \log n)$.*

Demostración. Sea C el conjunto producido por el algoritmo al término de cada iteración *repeat* (líneas 6-23). Para cualquier valor de mid, al inicio de la iteración *while* (líneas 11-16) C es un Conjunto Fuertemente Estable y T es el conjunto de nodos que no son dominados por C, lo cual resulta obvio al inicio de la iteración *while*, ya que C es el conjunto vacío y T es igual a V. A continuación se demuestra que, incluso al final de cada iteración *while*, C continúa siendo un Conjunto Fuertemente Estable y T un conjunto de nodos que no son dominados por C. Por el Lema 3.1.7, al término de cada iteración *while* el conjunto C debe ser un Conjunto Fuertemente Estable, ya que C se construye justamente agregando nodos x que no son dominados por C en G^2_{mid}. Más aún, los nodos que son eliminados de T son aquellos que se encuentran dentro del 2-vecindario de x, y por lo tanto son dominados por C en G^2_{mid}, de manera que los nodos que permanecen en T son aquellos que no son dominados por C en G^2_{mid}. Por lo tanto, al término de cada iteracion *while* el conjunto C es un Conjunto Fuertemente Estable en G_{mid} y además es un Conjunto Dominante en G^2_{mid}, con lo cual se cumple la desigualdad 3.3.

Durante toda la ejecución del algoritmo se cumple la siguiente desigualdad: $w_{e_{low}} \leq w(C^*)$. Esto se debe al hecho de que G_{low} tiene un Conjunto Fuertemente Estable de cardinalidad mayor a k y por lo tanto su Conjunto Dominante mínimo debe tener cardinalidad mayor a

[1] En realidad no importa si el conjunto de aristas del grafo de entrada no está ordenado, ya que puede ordenarse en una fase inicial del algoritmo sin que esto incremente el orden de la complejidad del mismo

k (tómese en cuenta que para toda $W \geq w(C^*)$, $G(W)$ tiene un Conjunto Dominante de cardinalidad al menos igual a k). Al término del algoritmo los valores de *high* y *low* son tales que $high = low + 1$, lo cual implica que $w_{e_{high}} \leq w(C^*)$. Dado que C además cumple con la desigualdad 3.3, por el Lema 3.1.5 la solución C' (que es igual a C) entregada por el algoritmo es una 2-aproximación.

La iteracion más externa *repeat* (líneas 6-23) ejecuta una búsqueda binaria, de modo que se repite a lo más $\log m$ veces. La iteración interna *while* (líneas 11-16) se ejecuta a lo más n veces, de modo que el orden de la complejidad del algoritmo es de $O(n^2 \log m) = O(n^2 \log n)$. □

Algoritmo 3.2: Algoritmo de Hochbaum y Shmoys

Entrada: un grafo $G = (V, E)$ y un entero k
Salida: un conjunto C, donde $|C| \leq k$

1. **if** $k = |V|$ **then**
2. $\quad C = V$ y termina el algoritmo.;
3. **end**
4. $low = 1$ (C puede ser el conjunto V) ;
5. $high = m$ (C puede ser cualquier nodo en V) ;
6. **repeat**
7. $\quad mid = \lfloor high + low/2 \rfloor$;
8. $\quad$ {Sea ADJ_{mid} la lista de adyacencia de G_{mid}. No es necesario construirla, pues al tener $w_{e_{mid}}$ podemos simularla sobre la lista de adyacencia de G.} ;
9. $\quad C = \emptyset$;
10. $\quad T = V$;
11. $\quad$ **while** $\exists x \in T$ **do**
12. $\quad\quad C = C \cup \{x\}$;
13. $\quad\quad$ **forall the** $v \in ADJ_{mid}(x)$ **do**
14. $\quad\quad\quad T = T - ADJ_{mid}(v) - \{v\}$;
15. $\quad\quad$ **end**
16. $\quad$ **end**
17. $\quad$ **if** $|C| \leq k$ **then**
18. $\quad\quad high = mid$;
19. $\quad\quad C' = C$;
20. $\quad$ **else**
21. $\quad\quad low = mid$;
22. $\quad$ **end**
23. **until** $high = low + 1$;
24. **return** C';

3.1.3. Algoritmo de Shmoys, Kleinberg y Tardos

El algoritmo de Kleinberg y Tardos [17] es prácticamente idéntico al algoritmo de González; de hecho, el mismo algoritmo fue propuesto también por Dyer y Frieze, pero para el caso más general en que cada nodo tiene un peso asignado [21].

El hecho de que este algoritmo haya sido propuesto por diferentes personas resulta muy interesante, pues permite contrastar la manera en que cada autor concibió y diseñó su algoritmo. Por un lado, González partió de la idea de construir *clusters*, mientras que Kleinberg y

Tardos (y de manera independiente Dyer y Frieze) partieron directamente de la idea de seleccionar centros. De hecho, González construye un conjunto de nodos a los que llama *heads* (los cuales en realidad no tienen otra función que la de servir de ayuda para la construcción de los *clusters*) y que es justamente el conjunto que el algoritmo de Kleinberg y Tardos entrega como solución. Otro motivo por el cual resulta interesante contrastar ambos algoritmos es por la manera tan diferente en que cada autor demuestra que su algoritmo entrega una 2-aproximación. De hecho, tanto la definición del algoritmo como la demostracion de Kleinberg y Tardos son más simples que las de González.

Antes de mostrar el algoritmo de Kleinberg y Tardos resulta útil analizar antes otro algoritmo, conocido como algoritmo de Shmoys, que permite resolver el problema de selección de k-centros con un factor de aproximacion de 2, pero para lo cual requiere conocer el radio de cobertura de la solución óptima $r(C^*)$. Desde luego que para conocer $r(C^*)$ es necesario construir primero el conjunto C^*, el cual es justamente el problema planteado. Por lo tanto, este algoritmo no resulta muy útil en la práctica, ya que para ejecutarlo es necesario proponer posibles valores de $r(C^*)$ y, dado que $r(C^*)$ es un número real, las posibilidades son infinitas (para determinadas instancias); sin embargo, el análisis de este algoritmo permite comprender más fácilmente el algoritmo de Kleinberg y Tardos. Este algoritmo es descrito por Shmoys en [34] y por Kleinberg y Tardos en [17].

Algoritmo 3.3: Algoritmo de Shmoys

Entrada: un grafo $G = (V, E)$, un entero k y un supuesto $r(C^*)$, llamado r
Salida: un conjunto C, donde $|C| \leq k$

```
1  V' es el conjunto de nodos que aún no son cubiertos ;
2  V' = V ;
3  C = ∅ ;
4  while V' ≠ ∅ do
5     Selecciona cualquier nodo v ∈ V' ;
6     C = C ∪ {v} ;
7     Elimina todos los nodos de V' cuya distancia hacia v sea a lo más 2r ;
8  end
9  if |C| ≤ k then
10    return C ;
11 else
12    No existe un conjunto de cardinalidad k con radio de cobertura menor o igual a r ;
13    Lo cual implica que el valor r ingresado como entrada es menor a r(C*) ;
14 end
```

Si el algoritmo de Shmoys (Algoritmo 3.3) entrega al término de su ejecución un conjunto C de cardinalidad menor o igual a k, claramente $r(C) \leq 2r$, ya que el hecho de eliminar los nodos de V', cuya distancia hacia $v \in C$ es a lo más $2r$ (línea 7), se debe a que dichos nodos se encuentran dentro del radio de cobertura de v, el cual es menor o igual a $2r$. En las líneas 12 y 13 se afirma que si $C > k$, entonces no existe una solución C de cardinalidad k con radio de cobertura menor o igual a r.

Teorema 3.1.9. *Si el algoritmo de Shmoys entrega una solución C de cardinalidad mayor a k, entonces, para cualquier conjunto C^* de cardinalidad menor o igual a k, el radio de cobertura $r(C^*) > r$.*

Demostración. Supóngase lo contrario, es decir, que existe un conjunto C^* de cardinalidad mayor a k con radio de cobertura $r(C^*) \leq r$. Cada centro $c \in C$ seleccionado por el algoritmo de Shmoys es un nodo que existe en el grafo de entrada $G = (V, E)$ y el conjunto C^* tiene un radio de cobertura de a lo más r, por lo tanto debe existir un centro $c^* \in C^*$ cuya distancia hacia el centro c sea a lo más r (i.e. $dist(c, c^*) \leq r$); se dice que dicho centro c^* es *cercano* a c. Para demostrar el teorema es necesario observar que ningún centro $c^* \in C^*$ puede ser *cercano* a dos centros de la solución C entregada por el algoritmo de Shmoys. La demostración de lo anterior implicaría que, dado que a cada centro $c \in C$ le corresponde un solo centro $c^* \in C^*$, entonces $|C^*| \geq |C|$ y, dado que $|C| > k$, entonces $|C^*| > k$, lo cual contradice nuestra suposición inicial de que $|C^*| \leq k$.

Para terminar la demostración es necesario demostrar que ningún centro $c^* \in C^*$ puede ser *cercano* a dos centros $c, c' \in C$. La demostración es simple: cada par de centros $c, c' \in C$ se encuentran separados por una distancia de al menos $2r$ (porque así lo obliga el algoritmo), por lo tanto, si $c^* \in C^*$ se encontrara a una distancia menor o igual a r de ambos centros $c, c' \in C$ estaría violando la desigualdad del triángulo, ya que $dist(c, c^*) + dist(c^*, c') \geq dist(c, c') > 2r$. □

El algoritmo de Shmoys entrega soluciones que son una 2-aproximación siguiendo la estrategia de seleccionar como centros a aquellos nodos que se encuentran a una distancia mayor o igual a $2r$ de los nodos previamente elegidos. El algoritmo de Kleinberg y Tardos utiliza esta misma estrategia, pero con la ventaja de que puede hacerlo sin necesidad de conocer el valor de r. La observación que permite eliminar el uso de valores hipotéticos de r es la siguiente: Si existe un nodo v, cuya distancia hacia los nodos previamente seleccionados como centros sea mayor o igual a $2r$, entonces el nodo más alejado debe ser uno de ellos. El Algoritmo 3.4 define al algoritmo de Kleinberg y Tardos.

Algoritmo 3.4: Algoritmo de Kleinberg y Tardos

Entrada: un grafo $G = (V, E)$ y un entero k
Salida: un conjunto C, donde $|C| \leq k$

```
1  if k = |V| then
2  |   C = V y termina el algoritmo ;
3  end
4  Selecciona cualquier nodo v ∈ V ;
5  C = {v} ;
6  while |C| ≤ k do
7  |   Selecciona un nodo v ∈ V tal que dist(v, C) sea máxima ;
8  |   C = C ∪ {v} ;
9  end
10 return C;
```

Teorema 3.1.10. *El algoritmo de Kleinberg y Tardos devuelve un conjunto C de k nodos tal que $r(C) \leq 2 \cdot r(C^*)$, donde C^* es la solucion óptima al problema de seleccion de k-centros.*

Demostración. Sea $r(C^*)$ el menor radio de cobertura posible que se puede obtener al distribuir k centros. Sea c un nodo que se encuentra a una distancia mayor a $2 \cdot r(C^*)$ de cada nodo en C. Considérese ahora una iteración intermedia en la ejecución del algoritmo, donde anteriormente

ya se ha conformado un conjunto de centros C'. Supóngase que el nodo agregado a C' en esta iteración es el nodo c'. Dado que el nodo c se encuentra a una distancia mayor a $2 \cdot r(C^*)$ de cada nodo en C, y dado que se ha seleccionado un nodo c' que es el más alejado de C', se observa que el nodo c' debe encontrarse a una distancia mayor a $2 \cdot r(C^*)$ de todos los centros en C':

$$dist(c', C') \geq dist(c, C') \geq dist(c, C) > 2 \cdot r(C^*) \tag{3.6}$$

Claramente el algoritmo de Kleinberg y Tardos es una correcta implementación de las primeras k iteraciones del algoritmo de Shmoys. Por lo tanto, al igual que sucede con el algoritmo de Shmoys, la solución C entregada por el algoritmo de Kleinberg y Tardos es tal que ningún centro $c^* \in C^*$ es *cercano* a dos o más centros $c \in C$. Lo cual permite establecer la siguiente desigualdad:

$$dist(v, C) \leq dist(v, c_i) \leq dist(v, c_i^*) + dist(c_i^*, c_i) \leq 2 \cdot r(C^*) \tag{3.7}$$

$dist(v, C)$ es la distancia de un nodo $v \in V$ cualquiera hacia la solución C entregada por el algoritmo de Kleinberg y Tardos; dicha distancia se define como la distancia de v hacia su centro $c \in C$ más cercano, por lo cual para cualquier valor de i (es decir, para todo $c \in C$) se cumple la desigualdad $dist(v, C) \leq dist(v, c_i)$. La desigualdad $dist(v, c_i) \leq dist(v, c_i^*) + dist(c_i^*, c_i)$ se cumple gracias a que los pesos de las aristas respetan la desigualdad del triángulo, donde $c_i \in C$ es el nodo de C más cercano al nodo $c_i^* \in C^*$ (recuérdese que a cada nodo $c_i \in C$ le corresponde únicamente un nodo $c_i^* \in C^*$). El valor máximo que puede tener $dist(v, c_i^*)$ es de $r(C^*)$ (pues de lo contrario C^* no sería la solución óptima) y el valor máximo que puede tener $dist(c_i^*, c_i)$ es de $r(C^*)$ por el mismo motivo, por lo tanto $dist(v, c_i^*) + dist(c_i^*, c_i) \leq 2 \cdot r(C^*)$. El tamaño de una solución C (es decir, su radio de cobertura) es representado por $r(C)$ y es igual al máximo valor de $dist(v, C)$ para todo $v \in V$. Dado que la desigualdad 3.7 se cumple para todo $v \in V$, entonces se cumple también para el caso en que $dist(v, C) = r(C)$. Por lo tanto:

$$r(C) \leq 2 \cdot r(C^*) \tag{3.8}$$

□

3.2. Heurísticas y metaheurísticas para el problema de selección de k-centros

Heurística o *"ars inveniendi"* solía ser el nombre de un área de estudio (no muy claramente circunscrita) perteneciente a la lógica, a la filosofía o a la psicología [45]. La finalidad de esta área es (o era) la de estudiar los métodos y reglas del *descubrimiento e invensión*. Por tanto, una heurística puede ser definida como *un procedimiento surgido más del sentido común y la experiencia que de una afirmación matemática* [44].

Muchos procedimiento heurísticos (o simplemente heurísticas) son básicamente algoritmos de búsqueda local que tienden a *quedar atrapados* en mínimos locales [46]. Las metaheurísticas son procedimientos más generales que las heurísticas, pues pueden ser aplicados sobre familias enteras de heurísticas de búsqueda [46].

El problema de selección de k-centros ha sido abordado a través de diferentes enfoques, siendo uno de ellos el de los algoritmos de aproximación (en la Sección 3.1 se describieron los algoritmos de aproximación más representativos para este problema). En la presente sección se describen algunas de las mejores heurísticas y metaheurísticas diseñadas para resolver el

problema de selección de k-centros; estas heurísticas y metaheurísticas cuentan con al menos una de las siguientes desventajas:

- No garantizan que las soluciones entregadas sean *cercanas* a la óptima (es decir, no garantizan que éstas sean una ρ-aproximación para un valor fijo de ρ).
- No es posible determinar la cota superior de su tiempo de ejecución.

Si bien, en teoría, las heurísticas y metaheurísticas cuentan con las desventajas mencionadas, también es verdad que en la práctica tienden a encontrar las mejores soluciones conocidas e incluso su tiempo de ejecución suele ser eficiente. A continuación se describen algunas de las heurísticas y metaheurísticas, aplicadas al problema de selección de k-centros, más relevantes del estado del arte. Una introducción más profunda a las metaheurísticas descritas se puede encontrar en [48].

3.2.1. Algoritmo de Mihelic y Robic

El algoritmo propuesto por Mihelic y Robic se basa en la relación que existe entre el problema de selección de k-centros y el problema del Conjunto Dominante (*Dominating Set*). Este algoritmo consiste básicamente en la resolución de una serie de subproblemas de Conjunto Dominante Mínimo; sin embargo, el problema del Conjunto Dominante Mínimo es *NP-Difícil*, de modo que Mihelic y Robic recurren al uso de una heurística eficiente para la resolución de este problema, utilizándola como base de su algoritmo. La heurística que sirve de base al algoritmo de Mihelic y Robic se aprecia en el Algoritmo 3.5:

La heurística utilizada para resolver el problema del Conjunto Dominante Mínimo funciona de la siguiente manera:

Inicialmente el conjunto dominante es el conjunto vacío, $D = \emptyset$ (línea 5). Durante la ejecución del algoritmo el conjunto D crece lo más lentamente posible (Mihelic y Robic se refieren a este comportamiento como el "principio perezoso"). Para cada nodo $v \in V$ se define una variable $CovCnt[v]$, la cual representa el número de veces que el nodo v es cubierto por el resto de los nodos. Esta variable es inicializada con el valor $deg(v) + 1$, donde $deg(v)$ es el grado de v (se considera que cada nodo es adyacente a sí mismo) (líneas 1-3). Otra variable que se define para cada nodo es la variable $Score[v]$, la cual representa el potencial (*score*) de cada nodo de ser considerado como un centro de la solución, siendo los valores más pequeños los que representan un potencial mayor. Al inicio el *score* de cada nodo es igual al valor de $CovCnt[v]$ (línea 4).

Las líneas 6-20 definen el ciclo más externo del algoritmo. En cada ejecución de este ciclo se elige un nodo x de *score* mínimo (línea 7); posteriormente se verifica si x tiene algún vecino y cuyo contador de cobertura ($CovCnt[y]$) sea igual a 1 (línea 8). Si es así, significa que el único nodo capaz de cubrir a y es el nodo x; por lo tanto, x es agregado al Conjunto Dominante D. Una vez agregado x al conjunto D, el contador de cobertura de los vecinos de x es igualado a 0, indicando así que estos nodos ya están cubiertos (líneas 10-12). Si x no tiene vecinos cuyo contador de cobertura sea igual a 1, entonces es utilizado para "mejorar" el *score* de sus vecinos aún no cubiertos (líneas 14-19); esto se logra decrementando en una unidad el contador de cobertura de cada vecino de x (ya que x no será elegido como centro y por lo tanto no los cubrirá) e incrementando en una unidad el *score* de sus vecinos. Al término de la iteración principal, el *score* del nodo x es igualado a infinito, para de esta manera evitar que pueda ser elegido una vez más como posible centro.

Claramente el algoritmo termina después de $|V| = n$ iteraciones del ciclo principal. Para demostrar que el algoritmo entrega un Conjunto Dominante es necesario considerar dos casos: el caso de los nodos aislados (es decir, aquellos nodos v que sólo pueden ser cubiertos

por sí mismos, ya que $CovCnt[v] = 1$) y el de los nodos no aislados (aquellos nodos v con $CovCnt[v] > 1$). En el caso de los nodos aislados, estos son siempre agregados al Conjunto Dominante D, pues la condición de la línea 8 siempre se cumple para ellos. Después de verificar todos los nodos aislados, el algoritmo continúa verificando a los nodos no aislados. Considérese un nodo no aislado y; éste nodo puede ser cubierto ya sea por un vecino x, previamente agregado a D (debido a la existencia de algún vecino de x, diferente de y, cuyo contador de cobertura fuera igual a 1), o bien porque el contador de cobertura de y es igual a 1 después de $deg(y)$ decrementos. Por lo tanto, todos los nodos no aislados también son cubiertos al término del algoritmo, de manera que el conjunto D entregado por el algoritmo es un Conjunto Dominante del grafo de entrada.

Algoritmo 3.5: Heurística para resolver el problema del Conjunto Dominante Mínimo

```
   Entrada: un grafo G = (V, E)
   Salida: un conjunto dominante D
 1 foreach v ∈ V do
 2 |   CovCnt[v] = grado(v) + 1 ;
 3 end
 4 Score = CovCnt ;
 5 D = ∅ ;
 6 for i ← 1 to |V| do
 7 |   x=selecciona un nodo con Score mínimo ;
 8 |   if ∃y ∈ V : ((x, y) ∈ E ∩ CovCnt[y] = 1) then
 9 |   |   D = D ∪ {x} ;
10 |   |   foreach y : (x, y) ∈ E do
11 |   |   |   CovCnt[y] = 0 ;
12 |   |   end
13 |   else
14 |   |   foreach y : (x, y) ∈ E do
15 |   |   |   if CovCnt[y] > 0 then
16 |   |   |   |   CovCnt[y] − − ;
17 |   |   |   |   Score[y] + + ;
18 |   |   |   end
19 |   |   end
20 |   end
21 |   Score[x] = ∞ ;
22 |   i + + ;
23 end
24 return D ;
```

Para resolver el problema de selección de k-centros, Mihelic y Robic hacen uso de la heurística descrita y de los conceptos de *poda paramétrica* y *grafo de cuello de botella* propuestos por Hochbaum y Shmoys. El algoritmo resultante es el Algoritmo 3.6:

Algoritmo 3.6: Algoritmo de Mihelic y Robic

Entrada: un grafo $G = (V, E)$
Salida: un conjunto C
1 Ordenar las aristas de forma creciente con base en su peso: $w_1, w_2, ..., w_m$, donde $m = |E|$;
2 **for** *i=1* **to** m **do**
3 $\quad G_i = GrafoDeCuelloDeBotella(G, w_i)$;
4 $\quad C =$Aplicar el Algoritmo del Conjunto Dominante sobre G_i (Algoritmo 3.5) ;
5 $\quad$ **if** $|C| \leq k$ **then**
6 $\quad\quad$ **return** C ;
7 $\quad$ **end**
8 **end**

3.2.2. Algoritmo de Daskin

Actualmente una parte considerable de la teoría de los algoritmos de aproximación gira alrededor de la Programación Lineal (*Linear Programming*) [3]. Si bien muchos algoritmos de programación lineal entregan ρ-aproximaciones, el algoritmo de Daskin no da ninguna garantía a este respecto, ni tampoco con respecto a que su tiempo de ejecución sea eficiente. La programación lineal consiste básicamente en optimizar una función lineal sujeta a restricciones en forma de igualdades y/o desigualdades lineales. La función que se desea optimizar es llamada *función objetivo*. Un problema de programación lineal en forma estandarizada consiste en encontrar los valores $x_1 \geq 0, x_2 \geq 0, ..., x_n \geq 0$ y el valor mínimo de z que satisfagan las siguientes igualdades:

$$\begin{array}{ccccccccc} c_1x_1 & + & c_2x_2 & + & \ldots & + & c_nx_n & = & z(Min) \\ a_{11}x_1 & + & a_{12}x_2 & + & \ldots & + & a_{1n}x_n & = & b_1 \\ a_{21}x_1 & + & a_{22}x_2 & + & \ldots & + & a_{2n}x_n & = & b_2 \\ \vdots & & \vdots & & \vdots & & \vdots & & \vdots \\ a_{m1}x_1 & + & a_{m2}x_2 & + & \ldots & + & a_{mn}x_n & = & b_m \end{array} \tag{3.9}$$

Con notación matricial se pueden reescribir las desigualdades anteriores como:

$$\begin{array}{rl} \text{Minimizar} & c^Tx = z \\ \text{sujeto a} & Ax = b, \quad A : m \times n, \\ & x \geq 0. \end{array} \tag{3.10}$$

Por ejemplo, sea el siguiente problema de programación lineal de dos variables:

$$\begin{array}{lrcrcr} \text{Minimizar} & -2x_1 & - & x_2 & = & z \\ \text{sujeto a} & x_1 & + & x_2 & \leq & 5 \\ & 2x_1 & + & 3x_2 & \leq & 12 \\ & x_1 & & & \leq & 4 \\ \text{donde} & \multicolumn{5}{l}{x_1 \geq 0, x_2 \geq 0.} \end{array} \tag{3.11}$$

La solución a este problema se puede obtener gráficamente al *sombrear* la región que contiene todas las soluciones *factibles* para después desplazar la función objetivo z sobre la misma hasta encontrar el valor mínimo que z puede tomar. La región de soluciones *factibles* contiene al conjunto de soluciones con coordenadas (x_1, x_2) que satisfacen todas las restricciones. En la Figura 3.2 se aprecia la región sombreada y la solución óptima al problema planteado.

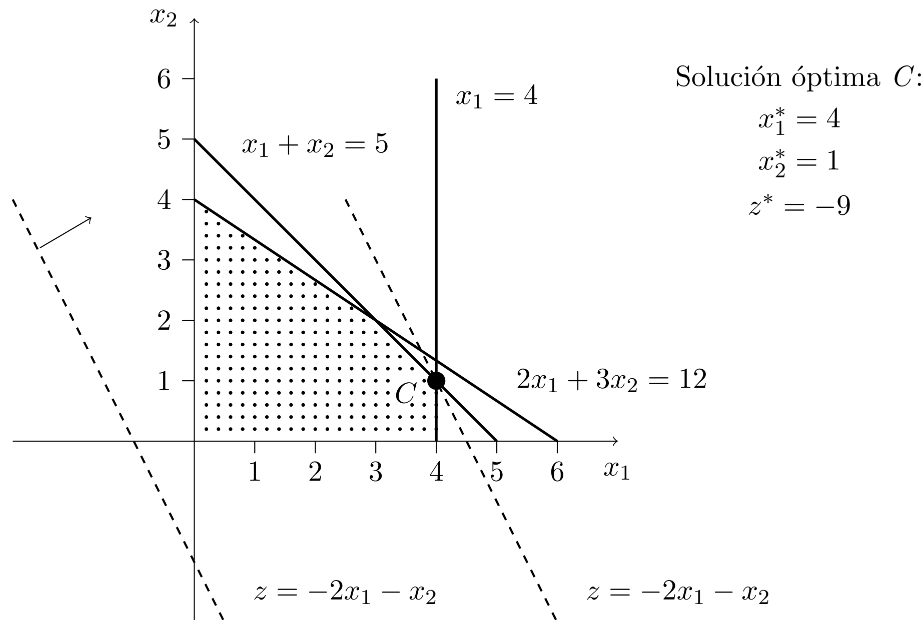


Figura 3.2: Solución gráfica a un problema de programación lineal con dos variables.

El problema de selección de k-centros puede ser expresado como un problema de Programación Entera (*Integer Programming*), la cual es un caso particular de la programación lineal. En los problemas de programación lineal los valores que pueden tomar las variables forman parte de un espacio continuo (por ejemplo, los números reales), mientras que en Programación Entera los valores que pueden tomar las variables son de tipo entero. Los problemas de programación lineal son resueltos de manera óptima y eficiente, mientras que los de programación entera sólo pueden ser resueltos de manera óptima en tiempo exponencial.

Para formular el problema de selección de k-centros como un problema de programación entera es necesario definir las siguientes variables:

$$I = \text{Conjunto de nodos } \textit{cliente}, \ I = \{1, ..., N\}$$

$$J = \text{Conjunto de nodos que pueden ser considerados como un centro, } J = \{1, ..., M\}$$

$$d_{ij} = \text{Distancia del nodo } i \in I \text{ hacia el nodo } j \in J$$

$$K = \text{Número de centros}$$

$$w_j = \begin{cases} 1 & \text{si hay un centro en } j \in J \\ 0 & \text{cualquier otro caso} \end{cases}$$

$$Y_{ij} = \begin{cases} 1 & \text{si el nodo } i \in I \text{ es asignado al centro } j \in J \\ 0 & \text{cualquier otro caso} \end{cases}$$

$$D = \text{Máxima distancia entre un nodo y su centro más cercano}$$

Utilizando estas variables, el problema de selección de k-centros puede ser expresado como un problema de programación entera:

(A) Minimizar D

sujeto a:

(B) $\sum_{j \in J} Y_{ij} = 1$ $\forall i \in I$

(C) $Y_{ij} \leq w_j$ $\forall i \in I, j \in j$

(D) $\sum_{j \in J} w_j = K$

(E) $D \geq \sum_{j \in J} d_{ij}\ Y_{ij} = 1$ $\forall i \in I$

(F) $w_j, Y_{ij} \in \{0, 1\}$ $\forall i \in I, j \in J$

La función objetivo (A) minimiza la máxima distancia entre cada nodo y su centro más cercano. La restricción (B) asegura que cada nodo es asignado a un solo centro, mientras que la restricción (C) impide que los nodos sean asigandos a nodos que no son un centro. La restricción (D) define el número de centros que han de seleccionarse. La restricción (E) define la máxima distancia que hay entre todo nodo y su centro más cercano. Las restricciones (F) señalan que los valores de las variables w_j y Y_{ij} deben ser 0 o 1.

La resolución al problema de programación entera planteado consta de una cantidad exponencial de pasos, por lo cual suele recurrirse al uso de alternativas como la de plantear el problema en términos de programación lineal relajada (*LP-Relaxation*). El algoritmo de Daskin hace uso justamente de esta idea, pues reduce el problema de selección de k-centros a una serie de subproblemas de Cobertura de Conjuntos (*Set Cover*) que son resueltos con la técnica de *relajación lagrangeana* de programación lineal. El algoritmo de Daskin (Algoritmo 3.7) explota la idea de *poda paramétrica* y la relación que existe entre el problema de Cobertura de Conjuntos y el problema de selección de k-centros. Esta relación es fácil de observar, pues intuitivamente es claro que el conjunto de *agrupamientos* que definen la solución óptima del problema de selección de k-centros es también (o tiende a ser) la solución al problema de Cobertura de Conjuntos, donde cada conjunto es construido con base en un *supuesto* tamaño de la solución (la *poda paramétrica* permite sortear la dificultad planteada por el desconocimiento del tamaño de la solución óptima).

Es importante señalar que el algoritmo de Daskin no es propiamente un algoritmo de aproximación, pues su tiempo de ejecución no necesariamente es polinomial y las soluciones que entrega no están acotadas por un factor ρ. En realidad, las soluciones entregadas por este algoritmo tienden a ser una aproximación muy cercana a la óptima del problema de Cobertura de Conjuntos, lo cual, por la relación expuesta entre el problema de Cobertura de Conjuntos y el problema de selección de k-centros, permite encontrar *buenas* soluciones para el problema de selección de k-centros.

El problema de Cobertura de Conjuntos es *NP-Completo*, por lo cual no aparenta representar una ventaja la *reducción* propuesta por Daskin; sin embargo, Daskin aprovecha el hecho de que existen métodos de Relajación de programación lineal (*LP-Relaxation*), en específico la técnica de relajación lagrangeana, que tienden a encontrar *buenas* soluciones (cercanas a la óptima y con asignaciones de valores enteros a las variables involucradas) para el problema de Cobertura de Conjuntos.

El algoritmo de Daskin no garantiza teóricamente un buen desempeño para instancias arbitrarias (tanto en términos del tiempo de ejecución como en términos de la calidad de las soluciones encontradas); sin embargo es quizá el algoritmo más eficaz y eficiente del estado del arte.

Algoritmo 3.7: Algoritmo de Daskin

```
   Entrada: un grafo G = (V, E), un entero k
   Salida: un conjunto C, donde |C| ≤ k
1  L = 0;
2  U = max{distancia(i, j) | ∀i, j ∈ V};
3  repeat
4  |  D = ⌊(L+U)/2⌋;
5  |  Construir una instancia para el problema de Cobertura de Conjuntos, generando un
   |  conjunto por cada nodo;
6  |  ∀i ∈ V, C_i = {j | j ∈ V, distancia(j, i) ≤ D};
7  |  Resolver el problema de Cobertura de Conjuntos sobre la instancia creada;
8  |  La solución al problema de selección de k-centros es el conjunto de nodos i
   |  correspondientes al conjunto de conjuntos encontrado;
9  |  if el número de conjuntos (nodos) es menor o igual a k then
10 |  |  U = D
11 |  else
12 |  |  L = D + 1
13 |  end
14 until L = U;
15 C ← la última solución correspondiente a D;
16 return C;
```

3.2.3. Algoritmo de Mladenovic

En [29] Nenad Mladenovic propone las heurísticas *Alternate* e *Interchange*, así como una metaheurística de Búsqueda Tabú (*Tabu Search*), para resolver el problema de seleccion de k-centros. La heurística *Alternate* consiste en resolver el problema de seleccion de 1-centro sobre cada *agrupamiento* de una solución, repitiendo el procedimiento hasta que deja de haber mejora en la misma (Algoritmo 3.16). La heurística *Interchange* consiste en intercambiar un centro de la solución por el nodo fuera de ella que reduzca lo más posible el valor de la función objetivo, repitiendo el procedimiento hasta que deja de haber mejora en la solucion (Algoritmo 3.8 y 3.17). La solución inicial utilizada por las heurísticas *Alternate* e *Interchange* se construye aleatoriamente. La metaheurística de búsqueda tabú consiste básicamente en realizar búsquedas locales sobre soluciones que son construidas con ayuda de *listas tabú* (se pueden utilizar una o dos listas tabú). En el Algoritmo 3.9 se muestra el pseudocódigo del algoritmo de Mladenovic, denominado *búsqueda tabú con sustitución de cadena* (*Chain-substitution Tabu Search*).

El algoritmo de búsqueda tabú con sustitución de cadena (Algoritmo 3.9) consta de 5 etapas:

- **(0) Inicialización**. En esta etapa se construye la solución inicial de la misma manera que en el algoritmo *Interchange* (Algoritmo 3.8), donde se construyen los arreglos x_{opt}, c_1, c_2 y se calcula el valor de las variables i^* y f. El arreglo x_{opt} contiene a todos los nodos del grafo, donde los primeros k elementos son la solución inicial; cada elemento $c_1(j)$ contiene al centro más cercano al nodo j; cada elemento $c_2(j)$ contiene al segundo centro más cercano al nodo j; i^* es el nodo *crítico*, es decir, aquel que define el valor de f, pues es el más alejado de la solución; f es el tamaño de la solución; el arreglo x_{opt} se construye aleatoriamente. El arreglo x_{opt} contendrá en todo momento a la mejor solución encontrada y los arreglo c_1 y c_2 contendrán el primer y segundo centro (en x_{opt})

más cercano a cada nodo. Es necesario copiar los valores de x_{opt}, c_1, c_2, i^* y f en los arreglos y variables x_{cur}, $c1_{cur}$, $c2_{cur}$, i^*_{cur} y f_{cur}, pues sobre estas copias se realizará la búsqueda. También es necesario inicializar las listas tabú $V^t_{in} = V^t_{out} = \emptyset$, lo cual se logra al definir dos contadores $cn_1 = k$ y $cn_2 = n$, ya que las listas tabú son secciones del arreglo x_{opt} (véase Figura 3.3, donde la línea horizontal representa al arreglo x_{opt} o x_{cur}); se definen los valores iniciales t_1 y t_2 para el tamaño de las listas tabú (los valores de t_1 y t_2 pueden ser definidos con base en la experiencia del usuario o bien de forma aleatoria) y finalmente se asigna el valor de *verdadero* a la variable *mejora*, la cual se encarga de determinar la manera en que se genera la lista de nodos candidatos.

- **(1) Reducción del vecindario**. En esta etapa se construye el conjunto de nodos candidatos para ingresar en la solución. Se considera como candidato a todo aquel nodo que se encuentre a una distancia menor o igual a f_{cur} del nodo *crítico* i^*_{cur}, siempre y cuando no esté en la lista tabú (el objetivo de la lista tabú es guardar a los nodos que han demostrado ser malos candidatos); si el conjunto de candidatos resulta ser igual al conjunto vacío, entonces se sustituye por el conjunto de todos los nodos que no están en la lista tabú.

- **(2) Encontrar el mejor nodo en el vecindario**. En esta etapa se aplica el procedimento $Move$ (Algoritmo 3.10) por cada nodo candidato y se guarda el mejor intercambio con su correspondiente valor de la función objetivo.

- **(3) Actualización**. Independientemente de que se haya encontrado una mejor solución, se actualizan los valores de x_{cur}, f_{cur}, c_1 y c_2, con los valores correspondientes a la mejor solución encontrada (la cual puede ser igual a la misma solución de entrada x_{opt}). Se actualizan los contadores de las listas tabú, de manera que en la siguiente iteración estas decrementen su tamaño en una unidad, dando así lugar a una búsqueda menos restringida, pues llegará el punto en que su tamaño sea cero. Cabe señalar que el algoritmo puede ejecutarse para funcionar con una o dos listas tabú; el método *Chain-substitution* 1 corresponde al caso en que se trabaja con una lista tabú, mientras que el método *Chain-substitution* 2 corresponde al caso en que se trabaja con dos.

- **(4) Mejora**. Si se encuentra una mejor solución (es decir, si x_{cur} es mejor que x_{opt}), se guarda en x_{opt} con su correspondiente valor de la función objetivo. Si no hubo mejora se asigna el valor de $false$ a la variable $mejora$, lo cual dará como resultado que en la siguiente iteración la lista de candidatos sea más grande, permitiendo que la búsqueda sea más extensa.

El tiempo de ejecución del algoritmo de búsqueda tabú con sustitución de cadena (Algoritmo 3.9) depende de una *condición de paro* definida por el usuario, la cual comúnmente es una cantidad de tiempo máximo, por ejemplo $2 \cdot n$ segundos. Aunque la cota superior del tiempo de ejecución es parte de la entrada del algoritmo, es posible conocer la cota inferior, la cual es de $\Omega(n^2)$, pues el algoritmo $Move$ (Algoritmo 3.10) es aplicado hasta n veces cuando el tamaño de la lista tabú es cero.

El algoritmo $Move$ (Algoritmo 3.10) permite resolver el problema de determinar, dada una solución y un candidato (a ser parte de la solución), el centro por el cual debe ser intercambiado el candidato, de tal manera que la nueva solución tenga un tamaño lo más pequeño posible. El tiempo de ejecución de este algoritmo es $O(n)$.

El algoritmo $Interchange$ (Algoritmo 3.8) consiste básicamente en aplicar el algoritmo $Move$ (Algoritmo 3.10) para cada nodo (es decir, todos los nodos son considerados como candidatos), repitiendo este procedimiento hasta que deja de haber una mejora en la solución.

Dado que la condición de paro depende de la instancia de entrada, no es posible determinar la cota superior del tiempo de ejecución de este algoritmo; sin embargo, sí es posible determinar su cota inferior, la cual es de $\Omega(n^2)$, pues el algoritmo *Move* se aplica hasta n veces por iteración.

El algoritmo *Update* (Algoritmo 3.13) permite actualizar eficientemente los arreglos c_1 y c_2. Si se utiliza una *pila* (*heap data structure*) la complejidad de este algoritmo es de $O(n \log n)$.

Algoritmo 3.8: Procedimiento *Interchange*

Entrada: un grafo $G = (V, E)$ y un entero k
Salida: un arreglo x_{opt} cuyos primero k elementos son la solución

```
1  (1) Inicialización.
2  Guardar una permutación aleatoria de nodos {1, ..., n} en el arreglo x_opt(j), j = 1, ..., n.
   Los k primeros elementos de x_opt son la solución inicial. Encontrar los dos centros más
   cercanos a cada nodo, es decir, llenar los arreglos c1(i), c2(i), i = 1, ..., m. Encontrar el
   valor de la función objetivo f_opt y el índice i* de un nodo, tal que f_opt = d(i*, c1(i*))
3  (2) Iteraciones.
4  f* ← ∞;
5  for in = x_opt(k + 1) hasta x_opt(n) do
6      (Agregar un nodo in en la solución y determinar el mejor centro a eliminar)
7      if (d(i*, in) < d(i*, c1(i*))) then
8          Ejecutar el procedimiento Move(c1, c2, d, x_opt, in, m, k, f, out) (Algoritmo 3.10);
9          (Guardar el mejor par de nodos a intercambiar)
10         if f < f* then
11             f* ← f;
12             in* ← in;
13             out* ← out;
14         end
15     end
16 end
17 (3) Finalización.
18 if f* ≥ f_opt then
19     (Si no hubo mejora termina el programa)
20     Termina el programa
21 end
22 (4) Actualización.
23 Actualizar el valor de la función objetivo: f_opt ← f* y encontrar el nuevo nodo crítico i*;
24 Actualizar x_opt; intercambiar la posición de x_opt(out*) con la de x_opt(in*);
25 Actualizar los dos nodos más cercanos: Update(d, in*, out*, m, k, c1, c2) (Algoritmo
   3.13);
26 Regresar al paso: (2) Iteraciones;
```

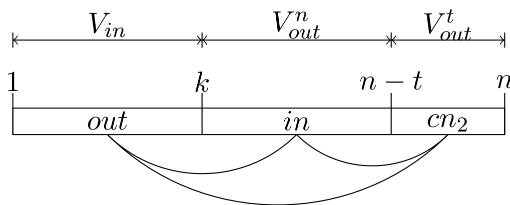
Algoritmo 3.9: Búsqueda tabú con sustitución de cadena

Entrada: un grafo $G = (V, E)$ y un entero k
Salida: un arreglo x_{opt} cuyos primero k elementos son la solución

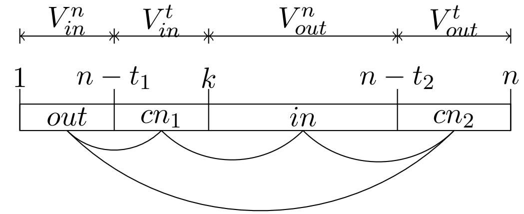
1 **(0) Inicialización**. **(1)** Construir los arreglos x_{opt}, c_1, c_2, f_{opt}, i^* de la misma manera que en la Inicialización del Algoritmo 3.8 (*Interchange*). **(2)** Copiar la solución inicial dentro de la *actual*, es decir, copiar los arreglos x_{opt}, c_1, c_2, f_{opt}, i^* en los arreglos x_{cur}, $c1_{cur}$, $c2_{cur}$, f_{cur}, i^*_{cur} **(3)** Inicializar las listas tabú ($V^t_{in} = V^t_{out} = \emptyset$) utilizando dos contadores $cn_1 = k$ y $cn_2 = n$; definir los valores iniciales t_1 y t_2 para el tamaño de las listas tabú; dar el valor de *verdadero* a la variable *mejora* ($mejora \leftarrow true$);
2 **repeat**
3   **(1) Reducir el vecindario**;
4   Construir el conjunto J de nodos que pueden ser insertados en la solución, $J = \{j \mid d(i^*_{cur} < f_{cur})\} \setminus V^t_{out}$;
5   **if** $J = \emptyset$ *o* $mejora = false$ **then**
6         $J = \{x_{cur}(i) \mid i = k + 1, ..., n\} \setminus V^t_{out}$;
7   **end**
8   **(2) Encontrar el mejor nodo en el vecindario**;
9     $f^* \leftarrow \infty$;
10   **for** $in \in J$ **do**
11   Encontrar el mejor centro a ser eliminado (*out*) usando el procedimiento $Move(c1_{cur}, c2_{cur}, d, x_{cur}, in, m, k, f_{cur}, out)$ (Algoritmo 3.10);
12   Guardar el mejor par de nodos a ser intercambiados;
13   **if** $f < f^*$ **then**
14             $f^* \leftarrow f, in^* \leftarrow in, out^* \leftarrow out$;
15   **end**
16   **end**
17   **(3) Actualización**;
18   Actualizar el valor de la función objetivo: $f_{cur} \leftarrow f^*$ y encontrar el nuevo nodo crítico i^*;
19   Actualizar x_{cur}: $Chain\text{-}substitution(in, out, cn_1, cn_2, x_{cur})$ (Algoritmos 3.11 y 3.12);
20   Actualizar los contadores de las listas tabú: $cn_1 = cn_1 - 1$; $cn_2 = cn_2 - 1$; si son iguales a $k - t_1$ o $n - t_2$, actualizarlos a k y n respectivamente;
21   Actualizar los dos nodos más cercanos: $Update(d, in^*, out^*, m, k, c_1, c_2)$ (Algoritmo 3.13);
22   **(4) Mejora**;
23   **if** $f^* < f_{opt}$ **then**
24         $f_{opt} = f^*$;
25         $mejora = true$;
26         $x_{opt}(j) = x_{cur}(j)$, $j = 1, ..., n$;
27   **else**
28         $mejora = false$;
29   **end**
30 **until** *condición de paro*;

Algoritmo 3.10: Procedimiento *Move*

Entrada: $c1, c2, d, x_{cur}, in, m, k, f, out$
Salida: out
1 **(1) Inicialización**.
2 $f \leftarrow 0$;
3 $r(x_{cur}(l)) \leftarrow 0,\ l = 1, ..., k$;
4 $z(x_{cur}(l)) \leftarrow 0,\ l = 1, ..., k$;
5 **(2) Agregar nodo**.
6 **for** $i = 1$ *hasta* $i = m$ **do**
7   **if** $d(i, in) < d(i, c_1(i))$ **then**
8         $f \leftarrow max\{f, d(i, in)\}$
9   **else**
10   (Actualizar los valores correspondientes al eliminar $c_1(i)$)
11         $r(c_1(i)) \leftarrow max\{r(c_1(i)), d(i, c_1(i))\}$;
12         $z(c_1(i)) \leftarrow max\{z(c_1(i)), min\{d(i, in), d(i, c_2(i))\}\}$;
13   **end**
14 **end**
15 **(3) Mejor eliminación**.
16 Encontrar la primera y segunda distancia más grande obtenida al no eliminar cada centro
17 $g_1 = max\{r(x_{cur}(l)), l = 1, ..., k\}$; sea l^* el índice correspondiente;
18 $g_2 = max_{l \neq l^*}\{r(x_{cur}(l))\},\ l = 1, ..., k$;
19 Encontrar el nodo out y el valor de f
20 $f = min\{g(l), l = 1, ..., k\}$; sea out el índice correspondiente;
21 $g(l) = \begin{cases} max\{f, z(x_{cur}(l)), g_1\} & \text{si } l \neq l^* \\ max\{f, z(x_{cur}(l)), g_2\} & \text{si } l = l^* \end{cases}$



(a) Sustitución de cadena con una lista tabú



(b) Sustitución de cadena con dos listas tabú

Figura 3.3: Sustitución de cadena con una y dos listas tabú

Algoritmo 3.11: *Chain-substitution* 1

Entrada: in, out, cn_2, x_{cur}
Salida: x_{cur}
1 $a \leftarrow x_{cur}(out)$;
2 $x_{cur}(out) \leftarrow x_{cur}(cn_2)$;
3 $x_{cur}(in) \leftarrow x_{cur}(cn_2)$;
4 $x_{cur}(cn_2) \leftarrow a$;

Algoritmo 3.12: *Chain-substitution* 2

Entrada: $in, out, cn_1, cn_2, x_{cur}$
Salida: x_{cur}
1 $a \leftarrow x_{cur}(out)$;
2 $x_{cur}(out) \leftarrow x_{cur}(cn_1)$;
3 $x_{cur}(cn_1) \leftarrow x_{cur}(in)$;
4 $x_{cur}(in) \leftarrow x_{cur}(cn_2)$;
5 $x_{cur}(cn_2) \leftarrow a$;

Algoritmo 3.13: Procedimiento *Update*

```
    Entrada: c1, c2, d, goin, goout, n, k
    Salida: c1, c2
 1  for i = 1 hasta i = n do
 2      (Si se eliminó el centro del nodo, se debe encontrar su nuevo centro)
 3      if c1(i) = goout then
 4          if d(i, goin) ≤ d(i, c2(i)) then
 5              c1(i) = goin
 6          else
 7              c1(i) = c2(i);
 8              (Encontrar el segundo centro más cercano del nodo i)
 9              Buscar el centro l*, tal que d(i, l) sea mínima (para l = 1, ..., k, l ≠ c1(i));
10              c2(i) ← l*;
11          end
12      else
13          if d(i, c1(i)) > d(i, goin) then
14              c2(i) ← c1(i);
15              c1(i) ← goin;
16          else
17              if d(i, goin) < d(i, c2(i) then
18                  c2(i) ← goin
19              else
20                  if c2(i) = goout then
21                      Encontrar el centro l*, tal que d(i, l) sea mínima (para
                        l = 1, ..., k, l ≠ c1(i));
22                      c2(i) ← l*;
23                  end
24              end
25          end
26      end
27  end
```

3.2.4. Algoritmo de Pacheco y Casado

En [26] Pacheco y Casado presentan un algoritmo basado en la estrategia de Búsqueda Dispersa (*Scatter Search*). El algoritmo de Pacheco y Casado, denominado *SS* (por *Scatter Search*), incorpora el uso de diferentes estrategias, tales como búsqueda local, GRASP (*Greedy Randomized Adaptive Search Procedure*) y Re-encadenamiento de Trayectoria (*Path Relinking*), siendo la base del algoritmo el método de búsqueda dispersa. Antes de presentar el algoritmo y su descripción, se describen (a grandes rasgos) en los siguientes párrafos las características básicas de las técnicas de búsqueda dispersa, re-encadenamiento de trayectoria y GRASP. Una más extensa y apropiada introducción a las técnicas mencionadas se puede encontrar en las siguientes referencias: búsqueda dispersa y re-encadenamiento de trayectoria [42], GRASP [18].

Tanto la técnica de búsqueda dispersa como la de re-encadenamiento de trayectoria trabajan sobre una población (conjunto) de soluciones y, empleando determinados procedimientos, las combinan dando lugar a nuevas soluciones. La idea de trabajar sobre poblaciones de solu-

ciones es similar a la idea básica del método de búsqueda tabú, donde se generan listas tabú y no-tabú, siendo precisamente estas últimas la población sobre la cual está permitido trabajar. La diferencia más clara entre las técnicas de búqueda tabú y búsqueda dispersa se encuentra en que esta última integra un marco de trabajo para la combinación de soluciones.

La técnica de búsqueda dispersa consta básicamente de dos partes: (1) captura de información que no existe en las soluciones cuando éstas están separadas y (2) uso de heurísticas auxiliares tanto para seleccionar los elementos a ser combinados como para generar las nuevas soluciones [42]. La técnica de búsqueda dispersa trabaja sobre un conjunto de soluciones, denominado *conjunto referencia*, el cual contiene soluciones que al ser combinadas permiten generar otras diferentes. Cuando el mecanismo de combinación utilizado consiste en la combinación lineal de dos soluciones, el *conjunto referencia* puede *evolucionar* de la forma ilustrada en la Figura 3.4, donde se asume que el *conjunto referencia* original consta de los círculos A, B y C; después de una combinación *no-convexa* de las soluciones A y B, la solución 1 es creada (es decir, una cierta cantidad de soluciones, que se encuentran sobre el segmento de línea definido por A y B, son creadas, siendo en este caso solo la solución 1 la que es integrada al *conjunto referencia*); las soluciones 2, 3 y 4 son generadas realizando combinaciones *convexas* y *no-convexas* entre las soluciones originales y las nuevas soluciones integradas. La solución 2 es un ejemplo de combinación *convexa* entre la solución C y B. La combinación de soluciones permite realizar la primera parte (captura de información que no existe en las soluciones cuando éstas están separadas) del método de búsqueda dispersa. La segunda parte se logra al aplicar determinadas heurísticas sobre el *conjunto referencia*, para posteriormente seleccionar las mejores soluciones y con ellas actualizar el *conjunto referencia*. El procedimiento se repite hasta que el *conjunto referencia* deja de cambiar, es decir, hasta que ninguna nueva solución es creada.

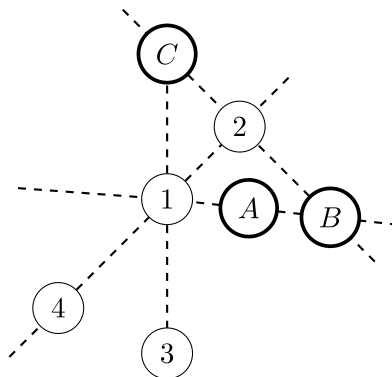


Figura 3.4: Evolución de un conjunto de referencia (búsqueda dispersa)

La técnica de re-encadenamiento de trayectoria es considerada una generalización de la técnica de búsqueda dispersa, pues la idea de combinar soluciones puede ser vista, desde una perspectiva espacial o geométrica, como la generación de trayectorias entre soluciones, donde las soluciones dentro de dichas trayectorias pueden servir también como fuentes de generación de nuevas trayectorias. Justamente la técnica de re-encadenamiento de trayectoria consiste en la generación de trayectorias entre soluciones. El método para generar estas trayectorias consiste en partir de una *solución inicial* y realizar movimientos que incorporen gradualmente atributos de una *solución guía*, de tal manera que llegue un punto en el que la *solución inicial* contenga todos los atributos de la *solución guía*, convirtiéndose en esta última; desde luego las soluciones que resultan de interés son las *mejores* (con base en la función objetivo establecida) dentro de la trayectoria generada. En la Figura 3.5 se muestra un ejemplo gráfico de re-

encadenamiento de trayectoria, donde la línea continua representa una trayectoria conocida previamente y la línea punteada es el resultado del re-encadenamiento entre la *solución inicial* x' y la *solución guía* x''.

La técnica GRASP (*Greedy Randomized Adaptive Search Procedures*) o Procedimientos de Búsqueda Voraz Aleatorizada Adaptable, consta básicamente de dos etapas: (1) generación de soluciones con base en la voracidad aleatorizada y (2) mejora de las soluciones con base en técnicas de búsqueda local. En la primera etapa el usuario interviene al definir el nivel de aleatoriedad del algoritmo (por ello el procedimiento incluye la palabra *adaptable*), de manera que la contrucción de soluciones puede ser totalmente aleatoria o totalmente definida por el algoritmo voraz utilizado. En la Figura 3.6 se ejemplifica gráficamente la manera en que la técnica GRASP realiza la búsqueda de soluciones, donde el conjunto *Greedy* contiene a todas las soluciones que un algoritmo voraz dado puede encontrar y el conjunto *Opt* contiene a las soluciones óptimas.

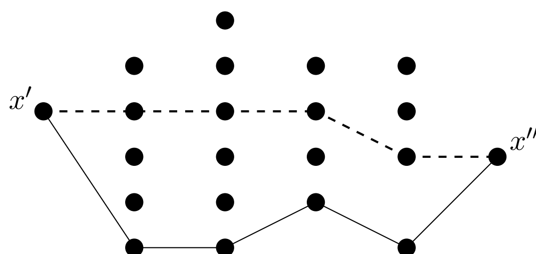
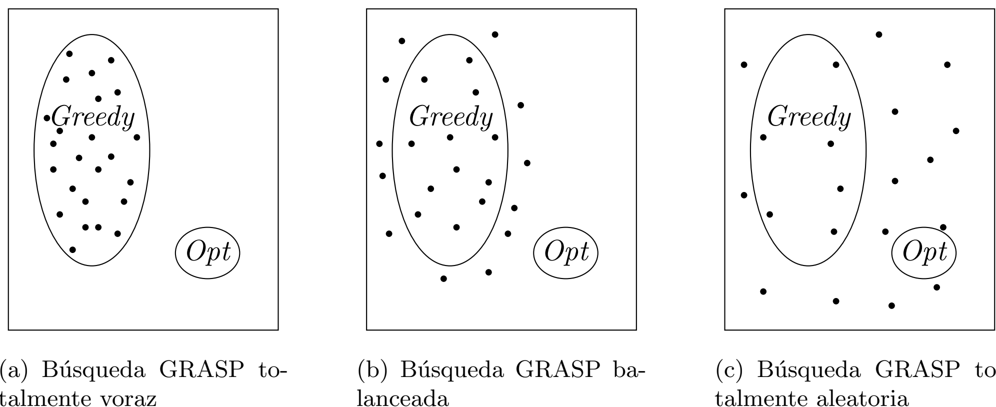


Figura 3.5: Re-encadenamiento de trayectoria



(a) Búsqueda GRASP totalmente voraz

(b) Búsqueda GRASP balanceada

(c) Búsqueda GRASP totalmente aleatoria

Figura 3.6: Búsqueda GRASP dentro del espacio de soluciones de un problema

El algoritmo de Pacheco y Casado se describe en términos generales en el Algoritmo 3.14.

Como ya fue mencionado, la técnica de búsqueda dispersa se caracteriza por el uso de un conjunto de referencia (*RefSet*) de soluciones. El algoritmo de Pacheco y Casado hace uso de un *RefSet* que a su vez consta de dos subconjuntos, el subconjunto *RefSet1* y el subconjunto *RefSet2*. El propósito del subconjunto *RefSet1* es el de guardar las mejores soluciones, por lo cual es denominado *subconjunto de calidad*, mientas que el subconjunto *RefSet2* se encarga de guardar la soluciones más diferentes dentro del conjunto *RefSet*, por lo cual es denominado *conjunto diversificador*. El tamaño del conjunto *RefSet* y de los subconjuntos *RefSet1* y *RefSet2* es definido por el usuario utilizando las variables *n_pob*, *Tam_Ref1* y *Tam_Ref2*.

Algoritmo 3.14: Algoritmo de Pacheco y Casado (*SS*)

Entrada: $G = (V, E)$, k, max_iter, n_pob, Tam_Ref1, Tam_Ref2, α y β
Salida: un conjunto C, donde $|C| \leq k$
1 (1) Generar un conjunto inicial de soluciones con un método *Generador-Diversificador*;
2 (2) Mejorar estas soluciones con un método de mejora;
3 (3) Con estas soluciones construirr un *RefSet* inicial;
4 (4) **repeat**
5 | (4.1) Obtener todos los subconjuntos de pares de *RefSet*;
6 | (4.2) Combinar estos subconjuntos para obtener nuevas soluciones;
7 | (4.3) Mejorar estas nuevas soluciones con el método de mejora;
8 | (4.4) Actualizar *RefSet* con estas nuevas soluciones;
9 **until** *RefSet se estabilice (i.e. no se incluyan nuevas soluciones)*;5
10 Si han transcurrido max_iter iteraciones de los pasos 1 al 4, finalizar; de lo contrario volver al paso 1 (reinicializar);
11 $C \leftarrow$ la mejor solución de *RefSet*;
12 **return** C;

En el paso (1) del Algoritmo 3.14 se genera un conjunto de soluciones candidatas a ingresar al *RefSet*, el cual es hasta este punto igual al conjunto vacío. El método *Generador-Diversificador* utilizado por el algoritmo de Pacheco y Casado (véase Algoritmo 3.15) recibe como entrada un parámetro α, el cual determina el nivel de aleatoriedad en la búsqueda de soluciones.

Algoritmo 3.15: Procedimiento *Ávido-Aleatorio* (*Generador-Diversificador*)

Entrada: un grafo $G = (V, E)$, un entero k y un valor α, $0 \geq \alpha \leq 1$
Salida: un conjunto C, donde $|C| \leq k$
1 $C \leftarrow \emptyset$;
2 $L \leftarrow \emptyset$;
3 **while** $|C| < k$ **do**
4 | Determinar, $\forall j \in V - C$, el valor Δ_j (es decir, el valor de f al añadir j a C);
5 | Determinar $\Delta_{max} = max\{\Delta_j \mid j \in V - C\}$ y $\Delta_{min} = min\{\Delta_j \mid j \in V - C\}$;
6 | Construir $L = \{j \in V - C \mid \Delta_j \leq \alpha \cdot \Delta_{min} + (1 - \alpha) \cdot \Delta_{max}\}$;
7 | Elegir $j^* \in L$ aleatoriamente;
8 | Hacer $C = C \cup \{j^*\}$;
9 **end**
10 **return** C;

La primera vez que se emplea el método *Generador-Diversificador* (paso 1 del Algoritmo 3.14) no hay "historia" acerca de cuántas veces un elemento ha formado parte de las soluciones del *RefSet*; sin embargo, esta información puede resultar útil cada vez que se reinicializa el procedimiento, por ello es importante almacenarla, conforme se ejecuta el algoritmo, dentro de un arreglo $freq$, donde $freq(j)$ contiene el número de veces que el elemento j de V ha pertenecido al *RefSet*. La información registrada en $freq(j)$ se puede aprovechar en la reinicialización del proceso, modificando los valores Δ_j dentro del método *Generador-Diversificador* de la siguiente manera:

$$\Delta'_j = \Delta_j - \beta\Delta_{max}\frac{freq(j)}{freq_{max}} \tag{3.12}$$

donde $freq_{max} = max\{freq(j) \mid \forall j\}$. Con los valores modificados Δ'_j se calculan los valores Δ'_{min} y Δ'_{max}, para con base en ellos ejecutar el método *Generador-Diversificador*, generando la lista de candidatos L. Nótese que cuando β es igual a cero, la ecuación 3.12 no modifica los valores originales de Δ_j, mientras que altos valores de β fuerzan a la selección de los elementos que menos han aparecido en *RefSet*; es por lo anterior que, cuando se aplica la ecuación 3.12 al método *Generador-Diversificador* (Algoritmo 3.15), éste es denominado simplemente método *Diversificador*, pues su misión principal es la de diversificar las soluciones dentro de *RefSet*.

Los métodos de mejora utilizados en el algoritmo de Pacheco y Casado (paso 2 y 4.3 del Algoritmo 3.14) son las heurísticas *Altenate* (Algoritmo 3.16) e *Interchange* (Algoritmo 3.17) ejecutadas en este mismo orden.

Algoritmo 3.16: Procedimiento *Alternate*

Entrada: un grafo $G = (V, E)$, una solución C'
Salida: un conjunto C, donde $|C| = |C'|$

1 $C \leftarrow \emptyset$;
2 **repeat**
3 Para cada centro j en C', determinar el subconjunto de nodos que tienen a j como centro más cercano;
4 Para cada uno de los subconjuntos generados resolver el Problema de Selección de 1 Centro;
5 Hacer C' igual a la nueva solución generada;
6 **until** *no hay cambios en* C';
7 $C \leftarrow C'$;
8 **return** C;

Para formar el *conjunto de referencia RefSet* (paso 3 del Algoritmo 3.14) se comienza por *RefSet1*, es decir, por los elementos de mayor calidad según la función objetivo. Para formar el conjunto *RefSet2* se utiliza la siguiente función o criterio que permite determinar la *diversidad* de una solución X con respecto a las soluciones que ya están en *RefSet*:

$$Difmin(X, RefSet) = min\{dif(X, X') \mid X' \in RefSet\} \tag{3.13}$$

donde:

$$dif(X, X') = |X - X'| \tag{3.14}$$

La actualización de *RefSet* (paso 4.4 del Algoritmo 3.14) se realiza considerando la calidad de las soluciones; es decir, se incorporan aquellas nuevas soluciones que mejoran la función objetivo de alguna de las soluciones existentes en *RefSet*.

Algoritmo 3.17: Procedimiento *Interchange*

Entrada: un grafo $G = (V, E)$, una solución C'
Salida: un conjunto C, donde $|C| = |C'|$

```
1  f ← tamaño de la solución C';
2  C ← ∅;
3  repeat
4      Para cada j ∈ V − C' y k ∈ C' determinar el valor de la función objetivo v_jk si se
       cambiara el nodo j por el centro k;
5      Determinar v_j*k* = min{v_jk | j ∈ V − C' y k ∈ C'};
6      if v_j*k* < f then
7          C' = {C' − {k*}} ∪ {j*};
8          f = v_j*k*
9      end
10 until no hay cambios en C';
11 C ← C';
12 return C;
```

Para combinar las soluciones (paso 4.2 del Algoritmo 3.14) se utiliza la técnica de reencadenamiento de trayectoria. El número de soluciones generadas por cada par de soluciones en *RefSet* depende de la relativa calidad de las soluciones que son combinadas. Considérese que x^p y x^q son las soluciones de *RefSet* que se desea combinar, donde $p < q$ (asumiendo que el *RefSet* está ordenado de manera que x^1 es la mejor solución y x^{Tam_Ref} la peor); el número de soluciones generadas por cada combinación es: 3, si $p \leq Tam_Ref1$ y $q \leq Tam_Ref1$; 2, si $p \leq Tam_Ref1$ y $q > Tam_Ref1$; 1, si $p > Tam_Ref1$ y $q > Tam_Ref1$. Para construir la trayectoria entre la solución x^p y x^q se siguen los siguientes pasos: se define una variable x que es inicializada como $x = x^p$; en los siguientes pasos se añade a x un elemento de x^q que x no tenga y se elimina un elemento de x que no esté en x^q. De esta forma la solución intermedia x en cada paso tiene un elemento más en común con x^q (en cada paso se elige el mejor de todos los posibles cambios).

La condición de paro del algoritmo es definida por el usuario usando la variable max_iter. Si transcurren más de max_iter iteraciones de los pasos 1 al 4 sin encontrar una mejor solución, entonces el algoritmo termina, devolviendo la mejor solución dentro de *RefSet*.

Si bien el algoritmo de Pacheco y Casado (Algoritmo 3.14) no da ninguna garantía teórica de su buen desempeño (ya sea en cuanto a su tiempo de ejecución o a la calidad de las soluciones encontradas), es un hecho que tiende a encontrar soluciones muy cercanas a las solución óptima. Una de las desventajas de este algoritmo radica en que, dada la gran cantidad de procedimientos que incorpora, tiende a tener un tiempo de ejecución *elevado*; incluso sus autores recomiendan su ejecución sobre instancias donde el valor de k sea menor o igual a 10.

3.3. Metaheurísticas del estado del arte similares a la técnica propuesta

En el Capítulo 4 se describe tanto el *algoritmo propuesto* como la técnica utilizada para su diseño. Esta técnia comparte algunas características con otras técnicas del estado del arte, en particular con las metaheurísticas EDA (*Estimation of Distribution Algorithms* o Algoritmos de Estimación de Distribución) y GRASP (*Greedy Randomized Adaptive Search Procedures* o Procedimientos de Búsqueda Voraz Aleatorizada Adaptable). A continuación se describe

brevemente cada una de estas metaheurísticas.

3.3.1. Algoritmos de Estimación de Distribución (EDAs)

El *Cómputo Evolutivo* es un área de estudio de las ciencias de la computación inspirada por el proceso de la evolución natural. El hecho de que muchos investigadores hayan visto a la selección natural como fuente de inspiración no resulta sorpendente, pues la naturaleza misma es un ejemplo de la evidente eficacia de este proceso, donde los individuos que sobreviven a su entorno son los más aptos para desarrollarse en el mismo [50].

Los *Algoritmos Evolutivos* o EAs (*Evolutionary Algorithms*) son un conjunto de técnicas inspiradas en la evolución natural de las especies y son los productos generados por el *Cómputo Evolutivo*. En el mundo natural, la evolución de las especies se lleva a cabo a través de la selección natural y de cambios aleatorios; procesos que pueden ser modelados a través de un algoritmo evolutivo y simulados con una computadora [49].

Los algoritmos evolutivos han sido aplicados a diferentes problemas de optimización exitosamente, superando casi siempre a los métodos clásicos. Si bien este tipo de algoritmos tienden a tener un buen desempeño en la práctica, tambien es cierto que no existe ninguna garantía de que el mínimo global sea encontrado ni de que la condición de paro sea la más conveniente; es decir, aún no cuentan con un soporte teórico firme.

Los algoritmos evolutivos bien pueden ser divididos en tres ramas: *Algoritmos Genéticos* o GAs (*Genetic Algorithms*), *Estrategias de Evolución* o ES (*Evolution Strategies*) y *Programación Evolutiva* o EP (*Evolutionary Programming*). Una apropiada introducción a cada uno de estos tipos de algoritmos evolutivos se puede encontrar en la siguiente referencia [49]. Si bien cada tipo de algoritmo evolutivo posee propiedades particulares, en general todos cuentan con los siguientes componentes básicos:

- Una población de individuos, donde cada individuo es una solución al problema particular planteado.
- Un conjunto de operadores aleatorios que modifican a los individuos.
- Un procedimiento de selección de individuos.

El comportamiento de un algoritmo evolutivo depende de los parámetros asociados a las variables involucradas, tales como: operadores de cruce (*crossover*) y mutación, probabilidad de cruce y mutación, tamaño de la población, tasa de reproducción generacional, número de generaciones, etc. Una de las desventajas de los algoritmos evolutivos radica en que determinar los mejores valores para cada una de las variables depende de la experiencia del usuario e incluso es por sí mismo un problema de optimización [51]. Esta desventaja, aunada al hecho de que predecir el movimiento de las poblaciones dentro del espacio de búsqueda es complicado, motivó el surgimiento y desarrollo de un nuevo tipo de algoritmos denominados Algoritmos de Estimación de Distribución o EDAs (*Estimation of Distribution Algorithms*).

En los algoritmos de estimación de distribución no son necesarios los operadores de cruce y mutación, pues cada nueva población de individuos es muestreada con base en una distribución de probabilidad, la cual es estimada sobre el conjunto de individuos seleccionados en la generación previa. La estimación de la distribución de probabilidad de empalme (*joint probability distribution*) del conjunto de individuos representa el principal problema para el diseño de algoritmos de estimación de distribución. Esta distribución de probabilidad es denominada de *empalme* (*joint*) por ser la función que determina la relación entre los individuos de la población.

Un algoritmo EDA consiste básicamente de la repetición de los siguientes pasos (hasta que determinada condición de paro se cumple):

1. Seleccionar ciertos individuos (soluciones) de una población (subconjunto de soluciones del espacio de búsqueda).
2. *Aprender* la distribución de probabilidad de empalme de los individuos seleccionados.
3. Realizar un muestreo de individuos con base en la distribución de probabilidad de empalme *aprendida*.

En el Algoritmo 3.18 se aprecia la estructura básica de un algoritmo EDA.

Algoritmo 3.18: Estructura básica de un algoritmo EDA

```
1 D_0 ← Generar M individuos (la población inicial) aleatoriamente;
2 l = 1;
3 repeat
4     D^{Se}_{l-1} ← Seleccionar N ≤ M indiviuos de D_{l-1} de acuerdo al método de selección;
5     p_1(x) = p(x|D^{Se}_{l-1}) ← Estimar la distribución de probabilidad de que un individuo
      pertenezca al conjunto de individuos seleccionados;
6     D_l ← Muestrear M individuos (la nueva población) con base en p_l(x);
7     l = l + 1;
8 until condición de paro;
```

El conjunto de algoritmos EDAs que realizan el muestreo con base en la factorización de distribuciones de probabilidad marginales univariables son denominados algoritmos UMDA (*Univariate Marginal Distribution Algorithm*). Existen muchos más tipos de algoritmos EDA, como los PBIL (*Population Based Incremental Learning*), cGA (*compact Genetic Algorithm*), MIMIC (*Mutual Information Maximization for Input Clustering*), COMIT (*Combining Optimizers with Mutual Information Trees*), BMDA (*Bivariate Marginal Distribution Algorithms*), EcGA (*Extended compact Genetic Algorithm*), etcétera [49]. Con la intención de conocer y comprender los elementos básicos de un algoritmo EDA, en la siguiente sección se presenta un algoritmo EDA simple (tipo UMDA) aplicado a un problema de optimización.

3.3.1.1. Ejemplo de algoritmo EDA

Sea *OneMax* una función, definida en un espacio de 6 dimensiones, que se desea maximizar. La función *OneMax* está definida por:

$$h(x) = \sum_{i=1}^{6} x_i \qquad \text{, donde } \; x_i = 0, 1 \tag{3.15}$$

La población inicial se obtiene aleatoriamente utilizando la siguiente distribución de probabilidad: $p_0(x) = \prod_{i=1}^{6} p_0(x_i)$, donde $p_0(X_i = 1) = 0.5$ para $i = 1, ..., 6$. Es decir, la distribución de probabilidad de empalme (*joint probability distribution*) utilizada para realizar el muestreo es factorizada como el producto de seis distribuciones de probabilidad marginal univariable (*univariate marginal distribution*), cada una de las cuales respeta una distribución de Bernoulli con el valor 0.5. Sea D_0 la población inicial generada por este muestreo (véase la Tabla 3.1).

Una vez formada la población inicial D_0, algunos de los individuos de la misma son seleccionados. Esta selección se puede realizar utilizando cualquier criterio, como por ejemplo el de

Tabla 3.1: Población inicial, D_0

	X_1	X_2	X_3	X_4	X_5	X_6	$h(x)$
1	1	0	1	0	1	0	3
2	0	1	0	0	1	0	2
3	0	0	0	1	0	0	1
4	1	1	1	0	0	1	4
5	0	0	0	0	0	1	1
6	1	1	0	0	1	1	4
7	0	1	1	1	1	1	5
8	0	0	0	1	0	0	1
9	1	1	0	1	0	0	3
10	1	0	1	0	0	0	2
11	1	0	0	1	1	1	4
12	1	1	0	0	0	1	3
13	1	0	1	0	0	0	2
14	0	0	0	0	1	1	2
15	0	1	1	1	1	1	5
16	0	0	0	1	0	0	1
17	1	1	1	1	1	0	5
18	0	1	0	1	1	0	3
19	1	0	1	1	1	1	5
20	1	0	1	1	0	0	3

truncamiento, que consiste en reducir la población a la mitad seleccionando los mejores individuos. Sea D_0^{Se} el conjunto de individuos seleccionados de la primera población D_0. Si existen empates entre las funciones de evaluación de algunos individuos (por ejemplo los individuos 1, 9, 12, 18 y 20), los individuos pueden seleccionarse aleatoriamente.

Tabla 3.2: Individuos seleccionados D_0^{Se}, de la población inicial D_0

	X_1	X_2	X_3	X_4	X_5	X_6
1	1	0	1	0	1	0
4	1	1	1	0	0	1
6	1	1	0	0	1	1
7	0	1	1	1	1	1
11	1	0	0	1	1	1
12	1	1	0	0	0	1
15	0	1	1	1	1	1
17	1	1	1	1	1	0
18	0	1	0	1	1	0
19	1	0	1	1	1	1

Una vez seleccionados los individuos (véase Tabla 3.2), resulta deseable expresar de forma explícita las características de los mismos. Esto se logra al calcular la distribución de probabilidad de empalme, la cual, idealmente, debería considerar todas las dependencias entre todas las variables. Con la intención de mantener la simplicidad, en este ejemplo cada variable es considerada independiente de las demás. La distribución de probabilidad de empalme para este ejemplo se expresa de la siguiente manera:

$$p_1(x) = p_1(x_1, ..., x_6) = \prod_{i=1}^{6} p(x_i \mid D_0^{Se}) \tag{3.16}$$

Se puede observar que sólo 6 parámetros son necesarios para especificar el modelo. Cada uno de estos parámetros, $p(x_i|D_0^{Se})$ con $i = 1, ..., 6$, puede ser estimado al calcular la frecuencia relativa, $\hat{p}(X_i = 1|D_0^{Se})$, correspondiente a cada una de las variables dentro del conjunto D_0^{Se}. En este caso, los valores de los parámetros son:

$$\begin{array}{lll} \hat{p}(X_1 = 1|D_0^{Se}) = 0.7 & \hat{p}(X_2 = 1|D_0^{Se}) = 0.7 & \hat{p}(X_3 = 1|D_0^{Se}) = 0.6 \\ \hat{p}(X_4 = 1|D_0^{Se}) = 0.6 & \hat{p}(X_5 = 1|D_0^{Se}) = 0.8 & \hat{p}(X_6 = 1|D_0^{Se}) = 0.7 \end{array} \tag{3.17}$$

Al realizar un muestreo con base en la distribución de probabilidad de empalme, $p_1(x)$, se obtiene una nueva población D_1. En la Tabla 3.3 se muestra el conjunto de individuos D_1^{Se} obtenidos al muestrear con base en $p_1(x)$.

Tabla 3.3: Población de la primera generación, D_1

	X_1	X_2	X_3	X_4	X_5	X_6	$h(x)$
1	1	1	1	1	1	1	6
2	1	0	1	0	1	1	4
3	1	1	1	1	1	0	5
4	0	1	0	1	1	1	4
5	1	1	1	1	0	1	5
6	1	0	0	1	1	1	4
7	0	1	0	1	1	0	3
8	1	1	1	0	1	0	4
9	1	1	1	0	0	1	4
10	1	0	0	1	1	1	4
11	1	1	0	0	1	1	4
12	1	0	1	1	1	0	4
13	0	1	1	0	1	1	4
14	0	1	1	1	1	0	4
15	0	1	1	1	1	1	5
16	0	1	1	0	1	1	4
17	1	1	1	1	1	0	5
18	0	1	0	0	1	0	2
19	0	0	1	1	0	1	3
20	1	1	0	1	1	1	5

De la primera generación, D_1, se seleccionan los mejores individuos utilizando algún criterio específico; en este ejemplo se utiliza el criterio de truncamiento (seleccionar los mejores individuos, reduciendo a la mitad la población). El resultado de esta selección de individuos da como resultado el conjunto de la Tabla 3.4.

Al igual que se hizo con el conjunto D_0^{Se}, se calcula la distribución de probabilidad de empalme del conjunto D_1^{Se}, es decir:

$$p_2(x) = p_2(x_1, ..., x_6) = \prod_{i=1}^{6} p(x_i \mid D_1^{Se}) \tag{3.18}$$

Tabla 3.4: Individuos seleccionados D_1^{Se}, de la primera generación D_1

	X_1	X_2	X_3	X_4	X_5	X_6
1	1	1	1	1	1	1
2	1	0	1	0	1	1
3	1	1	1	1	1	0
5	1	1	1	1	0	1
6	1	0	0	1	1	1
8	1	1	1	0	1	0
9	1	1	1	0	0	1
15	0	1	1	1	1	1
17	1	1	1	1	1	0
20	1	1	0	1	1	1

donde $p(x_i|D_1^{Se})$, con $i = 1, ..., 6$, es estimada al calcular la frecuencia relativa, $\hat{p}(X_i = 1|D_1^{Se})$, correspondiente a cada una de las variables dentro del conjunto D_1^{Se}. En este caso, los valores de los parámetros son:

$$\begin{array}{lll} \hat{p}(X_1 = 1|D_1^{Se}) = 0.9 & \hat{p}(X_2 = 1|D_1^{Se}) = 0.8 & \hat{p}(X_3 = 1|D_1^{Se}) = 0.8 \\ \hat{p}(X_4 = 1|D_1^{Se}) = 0.7 & \hat{p}(X_5 = 1|D_1^{Se}) = 0.8 & \hat{p}(X_6 = 1|D_1^{Se}) = 0.7 \end{array} \tag{3.19}$$

Con base en esta nueva distribución de probabilidad de empalme, $p_2(x)$, se puede generar una nueva generación de individuos sobre la cual se puede repetir el proceso hasta que se cumpla una determinada condición de paro. Sin duda este algoritmo es un algoritmo EDA, pues respeta la estructura esencial definida en el Algoritmo 3.18, que consiste básicamente en la repetición de los siguientes pasos:

1. Seleccionar ciertos individuos de una población.

2. *Aprender* la distribución de probabilidad de empalme de los individuos seleccionados.

3. Realizar un muestreo de individuos con base en la distribución de probabilidad de empalme *aprendida*.

3.3.2. Procedimientos de Búsqueda Voraz Aleatorizada Adaptable (GRASP)

La técnica GRASP (*Greedy Randomized Adaptive search Procedures*) o Procedimientos de Búsqueda Voraz Aleatorizada Adaptable, es una metaheurística de multi-inicio (*multi-start*) utilizada para resolver problemas de optimización combinatoria; esta técnica consiste básicamente en seleccionar soluciones (etapa de *construcción*) y aplicar algún procedimiento de búsqueda local sobre las mismas (etapa de búsqueda local). Este par de etapas es repetido hasta que se cumple la condición de paro, guardando siempre la mejor solución encontrada [18]. Las soluciones construidas por los algoritmos tipo GRASP son formadas utilizando *Voracidad Aleatorizada*, donde los elementos candidatos para ingresar en la solución son definidos vorazmente al calcular el tamaño de la solución si estos fueran incluídos (aquí es donde la parte de *Voracidad* es aplicada) y el elemento que ingresa a la solución es elegido aleatoriamente (aquí es donde la parte de *Aleatorización* es aplicada). En el Algoritmo 3.19 se muestra la estructura básica de un algoritmo GRASP.

Algoritmo 3.19: Metaheurística GRASP

Entrada: Entrada, condición de paro
Salida: Solución
1 Mejor_Solución = $\emptyset$;
2 **repeat**
3 Solución $\leftarrow$ Construcción_Voraz_Aleatorizada(Entrada, α);
4 Solución $\leftarrow$ Búsqueda_Local(Solución);
5 **if** *Solución es mejor que Mejor_Solución* **then**
6 Mejor_Solución $\leftarrow$ Solución;
7 **end**
8 **until** *condición de paro*;
9 **return** Mejor_Solución;

Algoritmo 3.20: Construcción_Voraz_Aleatorizada(Entrada, α)

Entrada: Entrada, α
Salida: Solución
1 Solución = $\emptyset$;
2 Sea E el conjunto de elementos de la Entrada que pueden formar parte de una Solución;
3 Inicializar el conjunto de candidatos: $C \leftarrow E$;
4 Evaluar el costo, $c(e)$, de considerar cada $e \in C$ en la Solución;
5 **while** $C \neq \emptyset$ **do**
6 $c_{min} \leftarrow min\{c(e) \mid e \in C\}$;
7 $c_{max} \leftarrow max\{c(e) \mid e \in C\}$;
8 Se construye una Lista de Candidatos Restringida (LCR);
9 $LCR \leftarrow \{e \in C \mid c(e) \leq c_{min} + \alpha(c_{max} - c_{min})\}$;
10 Seleccionar un elemento s de LCR aleatoriamente;
11 Solución $\leftarrow$ Solución $\cup\{s\}$;
12 $C \leftarrow C - \{s\}$;
13 Reevaluar el costo, $c(e)$, de considerar cada $e \in C$ en la Solución;
14 **end**
15 **return** Solución;

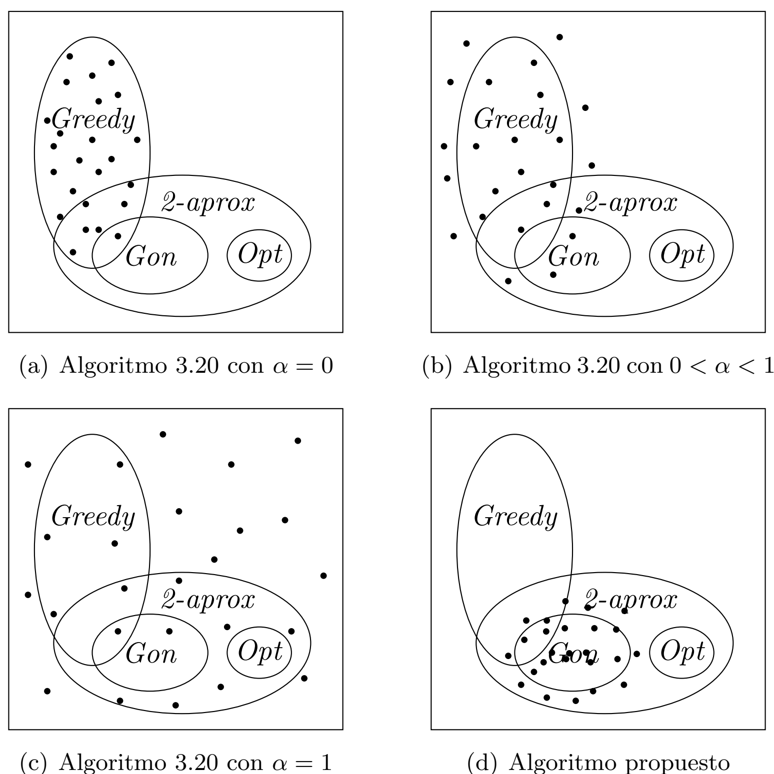
Algoritmo 3.21: Búsqueda_Local(Solución)

Entrada: Solución
Salida: Solución
1 **while** *Solución no es un óptimo local* **do**
2 Encontrar una solución $s' \in Vecindario(Solución)$ con $f(s') < f(Solución)$;
3 $Solución \leftarrow s'$;
4 **end**
5 **return** Solución;

El Algoritmo 3.20 permite construir soluciones aleatoriamente (línea 10), pero con base en una selección voraz previa (línea 9), donde el valor de la variable α está dentro del intervalo [0,1]. Si α es igual a 0, el elemento seleccionado será siempre el que minimiza la función, es decir, el algoritmo es totalmente voraz; mientras que cuando α es igual a 1, todos los elementos tendrán la misma probabilidad de ingresar a la solución. El valor de α es introducido por el

usuario y permite cierta flexibilidad para realizar la búsqueda de soluciones (de ahí el uso del término *Adaptable* en las siglas GRASP).

El Algoritmo 3.20 permite realizar una búsqueda aleatoria con base en el determinismo de un algoritmo voraz base. La variable α es definida por el usuario y permite dar cierta flexibilidad en la búsqueda de soluciones al determinar el nivel de aleatoriedad y determinismo. En la Figura 3.7 (incisos a, b y c) se aprecia una descripción gráfica de la manera en que el Algoritmo 3.20 realiza la búsqueda dentro del espacio de soluciones, donde el conjunto *Greedy* contiene todas las soluciones que dicho algoritmo (algoritmo voraz base) puede encontrar y el conjunto *Opt* es el conjunto de las soluciones óptimas; desde luego que la intersección del conjunto *Greedy* con el conjunto *Opt* no necesariamente es diferente al conjunto vacío.



(a) Algoritmo 3.20 con $\alpha = 0$

(b) Algoritmo 3.20 con $0 < \alpha < 1$

(c) Algoritmo 3.20 con $\alpha = 1$

(d) Algoritmo propuesto

Figura 3.7: Búsqueda aleatoria realizada por el Algoritmo 3.20 y el *algoritmo propuesto*

El principal problema asociado con los algoritmos GRASP es el de determinar el valor de α, los procedimientos de búsquea local y la condición de paro. Además, este tipo de algoritmos no garantizan que la solución encontrada sea un mínimo global. A pesar de lo anterior, esta metaheurística tiende a tener un buen desempeño para gran cantidad de problemas de optimización. Por ejemplo, en la Sección 3 se describe el algoritmo de Pacheco y Casado para el Problema de Selección de k Centros, el cual utiliza, entre otras técnicas, un algoritmo GRASP y es uno de los más eficaces del estado el arte.

3.4. Análisis experimental de los algoritmos del estado del arte

Los algoritmos de aproximación descritos en este capítulo son los mejores posibles (a menos que P=NP), pues encuentran en tiempo polinomial una 2-aproximación. Las heurísticas y metaheurísticas descritas, si bien no garantizan teóricamente un *buen* desempeño, sí han

demostrado experimentalmente ser tanto eficaces como eficientes, a excepción del algoritmo de Pacheco y Casado, cuyo tiempo de ejecución tiende a ser *elevado*).

En esta sección se muestra un análisis comparativo de los algoritmos descritos. El método de comparación consiste en ejecutar los algoritmos estudiados sobre un conjunto estándar de instancias de entrada. Las instancias de entrada elegidas son un conjunto de 40 grafos presentados en [24], los cuales fueron inicialmente propuestos para la comparación experimental de algoritmos diseñados para resolver el problema del *k-median* (o *p-median*); sin embargo, son muchos los autores que los han utilizado para realizar la comparación de algoritmos diseñados para resolver el problema de selección de k-centros [13, 14, 20, 22, 27, 29, 38], de manera que actualmente es muy común su uso con este fin, pues se cree que el conjunto de mejores soluciones conocidas que se ha obtenido corresponde precisamente a las soluciones óptimas. El tamaño de estos grafos varía desde los 100 nodos hasta los 900 nodos; el valor de k varía desde 5 hasta 200.

Los algoritmos comparados son los siguientes:

- **Método voraz puro : Gr** (*pure greedy method*). Consiste en elegir los centros de manera consecutiva, tratando de reducir en cada iteración el valor de la función objetivo; es decir, el primer nodo elegido como centro es aquel que minimiza el tamaño de la solución con $k = 1$ sobre el grafo de entrada G, el segundo nodo elegido como centro es aquel que minimiza el tamaño de la solución, pero sin cambiar de posición el (los) nodo(s) previamente elegido(s) como centro(s), y así sucesivamente hasta tener k centros.
- **Algoritmo de González : Gon**.
- **Algoritmo de Hochbaum y Shmoys : HS**.
- **Algoritmo de Mihelic y Robic : Scr** (*algoritmo de scoring*).
- **Algoritmo de Daskin : Das**.
- **Algoritmo de Mladenovic : TS-1 y TS-2** (*búsqueda tabú con 1 y 2 listas*).
- **Algoritmo de Pacheco y Casado : SS** (*Scatter Search*).

Las siguiente son algunas variantes del estado del arte [16] que definen la manera en que se selecciona el primer centro:

- **Aleatoriamente : R** *(random)*. El primer nodo se elige de manera aleatoria.
- **1-center : 1**. El primer nodo seleccionado como centro es aquel que resuelve de manera óptima el Problema de Selección de k Centros con k=1.
- **n-repeticiones : + o Plus**. Se ejecuta n veces el algoritmo, eligiendo en cada iteración un nodo diferente.

Los resultados obtenidos al ejecutar cada uno de los algoritmos, con cada una de la variantes mencionadas, se aprecian en la Tabla 3.5, donde los resultados reportados son los valores del radio de cobertura de la solución encontrada por cada algoritmo. Las mejores soluciones conocidas se encuentran resaltadas.

Para obtener los resultados del algoritmo de Daskin se utilizó la herramienta SITATION [43], la cual es un programa diseñado por M. Daskin y distribuido gratuitamente. Esta herramienta cuenta con la desventaja de que sólo permite resolver el problema de selección de k-centros sobre instancias de hasta 300 nodos; es por esto que en la Tabla 3.5 sólo se muestran resultados para el algoritmo de Daskin sobre los primeros 15 grafos.

El algoritmo SS (*Scatter Search*) se ejecutó con los valores $max_iter = 5$, $n_pob = 12$, $\alpha = 0.8$, $\beta = 0.8$, $ref1_size = 5$ y $ref2_size = 5$. Como ya se mencionó, el tiempo de ejecución del algoritmo SS puede ser muy alto; de hecho Pacheco y Casado recomiendan su uso para valores *pequeños* de k, específicamente para $k \leq 10$. En la Tabla 3.5 se muestran los resultados del algoritmo SS sobre las instancias *pmed1* hasta *pmed15* y sobre las instancias donde k es menor o igual a 10 (el tiempo de ejecución de este algoritmo sobre las otras instancias puede tomar semanas o meses). La condición de paro recomendada para los algoritmos TS-1 y TS-2 es de $2n$ segundos [29], donde n es el número de nodos; sin embargo estos algoritmos fueron ejecutados durante $3n$ segundos en el presente trabajo.

Con base en los resultados de la Tabla 3.5 es posible determinar el promedio del porcentaje de aproximación de cada algoritmo sobre el conjunto de grafos. Este método de comparacion es completamente experimental, por lo cual no entrega resultados contundentes sobre la superioridad de un algoritmo sobre los otros; sin embargo, resulta útil para determinar qué algoritmos tienden a ser mejores en la práctica.

El promedio del porcentaje de aproximación entregado por cada algoritmo (con sus respectivas variantes) sobre el conjunto de grafos se aprecia en las Tablas 5.3 y 5.4, donde la Tabla 5.4 considera únicamente los primeros 15 grafos y la Tabla 5.3 considera los 40 grafos.

Las gráficas de barras de las Figuras 3.9 y 3.8 permiten apreciar que el algoritmo de aproximación que tiende a entregar mejores soluciones es el Gon+, pues encuentra soluciones con un factor de aproximación experimental promedio de 1.28. Los algoritmos que tienden a encontrar las mejores soluciones conocidas son los algoritmos Scr, SS, TS-2 y Das con un factor de aproximación experimental promedio de 1.057, 1.078, 1.30 y 1.002 respectivamente.

En la Tabla 3.8 se muestra la cota superior del tiempo de ejecución de cada uno de los algoritmos analizados. Los algoritmos Das, TS-1, TS-2 y SS no son polinomiales, por lo cual no cuentan con una cota superior para su tiempo de ejecución.

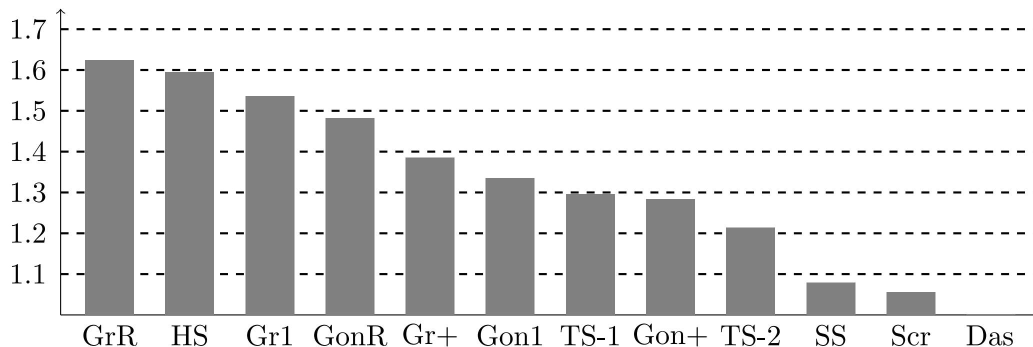


Figura 3.8: Comparación de la calidad de los algoritmos del estado del arte (grafos 1 al 15)

Tabla 3.5: Resultados de los algoritmos del estado del arte ejecutados sobre los grafos *pmed* de la librería OR-Lib

grafo	n	k	Mejor solución conocida	GrR	Gr1	Gr+	Gon	Gon1	Gon+	HS	Scr	Das	TS-1	TS-2	SS
1	100	5	127	143	133	133	191	162	155	174	133	**127**	**127**	**127**	**127**
2	100	10	98	112	117	111	143	124	117	155	109	**98**	100	100	**98**
3	100	10	93	116	116	106	133	133	124	148	99	**93**	121	94	**93**
4	100	20	74	108	108	92	101	99	93	109	83	**74**	79	77	**74**
5	100	33	48	100	87	78	66	64	62	76	**48**	**48**	70	67	**48**
6	200	5	84	97	94	89	138	99	98	121	90	**84**	**84**	**84**	**84**
7	200	10	64	79	79	70	102	83	83	96	70	**64**	**64**	**64**	**64**
8	200	20	55	79	79	70	82	71	68	85	60	**55**	57	**55**	**55**
9	200	40	37	73	73	64	57	51	49	61	38	**37**	63	56	40
10	200	67	20	56	44	38	31	29	29	32	**20**	**20**	25	**20**	24
11	300	5	59	68	67	60	80	68	68	80	60	**59**	60	60	**59**
12	300	10	51	61	72	56	72	70	66	76	53	**51**	52	52	**51**
13	300	30	35	60	58	52	54	51	48	72	38	36	58	55	40
14	300	60	26	60	46	46	41	37	36	45	27	**26**	48	42	34
15	300	100	18	42	42	39	25	25	23	35	**18**	**18**	37	36	26
16	400	5	47	55	51	**47**	74	55	52	59	48	-	**47**	**47**	**47**
17	400	10	39	50	50	44	57	49	49	71	41	-	41	**39**	**39**
18	400	40	28	62	62	42	44	41	39	48	32	-	30	38	-
19	400	80	18	38	43	35	29	28	26	32	20	-	23	31	-
20	400	133	13	32	32	32	19	18	17	25	14	-	27	32	-
21	500	5	40	44	48	42	58	51	45	64	**40**	-	**40**	**40**	**40**
22	500	10	38	48	48	43	61	54	47	58	41	-	39	39	**38**
23	500	50	22	38	38	33	32	32	30	45	24	-	37	37	-
24	500	100	15	42	35	32	22	22	21	38	17	-	22	19	-
25	500	167	11	27	27	27	16	16	15	20	**11**	-	27	27	-
26	600	5	38	44	43	39	59	47	43	55	41	-	**38**	**38**	**38**
27	600	10	32	39	39	35	49	42	38	44	33	-	**32**	**32**	**32**
28	600	60	18	28	29	27	29	28	25	31	20	-	24	22	-
29	600	120	13	36	36	34	19	19	18	23	**13**	-	24	18	-
30	600	200	9	29	29	29	13	14	13	18	10	-	29	20	-
31	700	5	30	36	36	31	41	41	36	45	**30**	-	31	**30**	**30**
32	700	10	29	34	37	32	40	43	36	45	31	-	30	30	30
33	700	70	15	26	26	25	26	24	23	27	17	-	21	20	-
34	700	140	11	30	30	27	17	16	16	21	11	-	20	18	-
35	800	5	30	30	33	31	42	38	34	41	32	-	**30**	31	**30**
36	800	10	27	34	34	30	45	41	34	46	28	-	28	28	28
37	800	80	15	28	26	25	24	24	23	28	16	-	22	19	-
38	900	5	29	39	35	30	42	38	31	42	**29**	-	**29**	**29**	**29**
39	900	10	23	29	29	25	32	35	28	38	24	-	25	24	24
40	900	90	13	23	22	22	21	20	20	23	14	-	22	17	-
Comp.				$O(kn)$	$O(n^2)$	$O(kn^2)$	$O(kn)$	$O(n^2)$	$O(kn^2)$	$O(n^2 \log n)$	$O(n^3)$	-	-	-	-

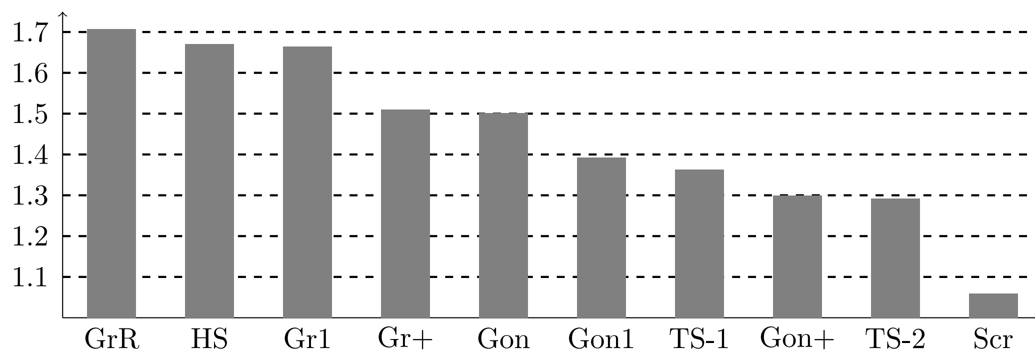


Figura 3.9: Comparación de la calidad de los algoritmos del estado del arte (grafos 1 al 40)

Tabla 3.6: Factores de aproximación experimentales (grafos 1 al 40)

Algoritmo	Factor de aproximación	
	Promedio	Desviación
GrR	1.7075	0.6105
HS	1.6702	0.2472
Gr1	1.6645	0.5649
Gr+	1.5106	0.5541
GonR	1.5021	0.0950
Gon1	1.3928	0.1199
TS-1	1.3635	0.4853
Gon+	1.2991	0.1231
TS-2	1.2924	0.4090
Scr	1.0591	0.0443

Tabla 3.7: Factores de aproximación experimentales (grafos 1 al 15)

Algoritmo	Factor de aproximación	
	Promedio	Desviación
GrR	1.6240	0.6105
HS	1.5941	0.2472
Gr1	1.5353	0.5649
GonR	1.4818	0.0950
Gr+	1.3853	0.5541
Gon1	1.3354	0.1199
TS-1	1.2954	0.4853
Gon+	1.2833	0.1231
TS-2	1.2136	0.4090
Scr	1.0555	0.0443
SS	1.0784	0.6522
Das	1.0019	0.4912

Tabla 3.8: Complejidad de los algoritmos del estado del arte

Algoritmo	Complejidad
GrR	$O(kn)$
Gr1	$O(n^2)$
HS	$O(n^2 \log n)$
Gr+	$O(kn^2)$
GonR	$O(kn)$
Gon1	$O(n^2)$
Gon+	$O(kn^2)$
Scr	$O(n^3)$
Das	–
TS-1	–
TS-2	–
SS	–

3.5. Discusión

- El problema de selección de k-centros pertenece a la clase *NP-Difícil*, por lo cual solo es posible resolverlo de manera óptima en $O(n^k)$ pasos, donde n es el número de nodos. De igual manera, encontrar una ρ-aproximación, donde $\rho < 2$, es también un problema *NP-Difícil*. Debido a esto se han diseñado tanto algoritmos de aproximación como heurísticas y metaheurísticas para su resolución.

- Los mejores algoritmos de aproximación diseñados para el problema de selección de k-centros encuentran una 2-aproximación (algoritmo de González [1], de Hochbaum-Shmoys [4] y de Kleinberg-Tardos [17]). Dado que encontrar una mejor aproximación (en tiempo polinomial) demostraría que $P = NP$, se dice que estos algoritmos de aproximación son los mejores posibles (si $P \neq NP$). Si bien estos algoritmos son, en teoría, los mejores posibles, también es cierto que las soluciones que entregan pueden resultar poco satisfactorias en la práctica, sobre todo cuando lo que se desea es encontrar la solución óptima. En la Tabla 5.3 se observa que el mejor factor de aproximación experimental (sin considerar a las heurísticas y metaheurísticas) se atribuye al algoritmo Gon+, el cual consiste en repetir el algoritmo de González n veces, seleccionando como centro inicial un nodo diferente en cada ejecución. El factor de aproximación experimental de este algoritmo es de 1.29.

- Una alternativa a los algoritmos de aproximación es la de las heurísticas y metaheurísticas, las cuales son *"procedimientos surgidos más del sentido común y la experiencia que de una afirmación matemática"* [44]. Dado que estos procedimientos no tienen un soporte teórico fuerte, no es posible garantizar su *buen* desempeño para instancias arbitrarias; a pesar de lo cual tienden a entregar las mejores soluciones conocidas. Los algoritmos, dentro de esta categoría, que tienden a encontrar las mejores soluciones conocidas para el problema de selección de k-centros son el algoritmo Das (algoritmo de Daskin [37]), el algoritmo TS-2 (búsqueda tabú con 2 listas tabú [29]), el algoritmo SS (*Scatter Search* [26]) y el algoritmo Scr (*Scoring* [16]).

Capítulo 4

Propuesta

Con la intención de hacer más clara la exposición de la propuesta, el presente capítulo cuenta con la siguiente estructura:

1. **Propuesta general**: Se describe, en términos generales, la técnica utilizada para el diseño del *algoritmo propuesto*. Cabe señalar que esta técnica no parece corresponder de manera precisa a alguna otra técnica del estado del arte, es decir, se trata de una aportación original. Además, esta técnica de diseño no es exclusiva del problema de selección de k-centros, sino que puede ser utilizada para resolver otros problemas NP-*Difíciles*.

2. **Generalización del algoritmo de González (GGon)**: Se presenta una versión *relajada* del algoritmo de González, donde en vez de seleccionar siempre como centro al nodo más alejado (i.e. seleccionar con una probabilidad de 1 al nodo más alejado) se selecciona un nodo con base en una función de masa de probabilidad cualquiera (siempre y cuando se trate de una función válida).

3. **Algoritmo propuesto**: El *algoritmo propuesto* consiste de la Generalización del algoritmo de González (GGon) que hace uso de una función de masa de probabilidad específica, denominada función *Caracterizable* (C), que recibe como entrada una variable α definida por el usuario.

   a) **Función de masa de probabilidad *Caracterizable* (C)**: La función *Caracterizable* es definida. Esta función se construye con base en un parámetro α definido por el usuario.
   b) **Caracterización teórica**: Se demuestra que el *algoritmo propuesto*, que no es mas que la Generalización del algoritmo de González que hace uso de la función *Caracterizable* y que recibe como entrada una variable α, entrega una 2-aproximación con una probabilidad de al menos 0.63 al ser amplificado por lo menos α veces.
   c) **Variantes de búsqueda local**: Se muestran un par de variantes de búsqueda local que pueden ser incorporadas al *algoritmo propuesto* y que tienden a incrementar su eficacia. Estas variantes son un caso particular de las heurísticas *Alternate* e *Interchange* del estado del arte.
   d) **Complejidad**: Se calcula la complejidad del *algoritmo propuesto* utilizando las diferentes variantes propuestas.

4. **Similitudes entre la técnica propuesta y algunas metaheurísticas del estado del arte**: Se mencionan algunas de las similitudes y diferencias que existen entre la

técnica utilizada para el diseño del *algoritmo propuesto* y algunas metaheurísticas del estado del arte. Esta técnica, al igual que el *algoritmo propuesto*, es una aportación original.

4.1. Propuesta general

Como ya se ha mencionado, los algoritmos del estado del arte diseñados para resolver el problema de selección de k-centros suelen contar con alguna de las siguientes características:

- Encuentran soluciones aproximadas (2-aproximaciones) para instancias arbitrarias [1, 4, 17] y su tiempo de ejecución es polinomial; es decir, se trata de algoritmos de aproximación.

- A pesar de no garantizar que las soluciones encontradas sean una ρ-aproximación, ni que su tiempo de ejecución sea eficiente, tienden a entregar las mejores soluciones conocidas (estos algoritmos son clasificados como heurísticas y metaheurísticas).

El *algoritmo propuesto* está basado en una técnica de *aleatorización de algoritmos de aproximación*, la cual permite diseñar algoritmos aleatorios que encuentran soluciones aproximadas. En el Capítulo 5 se muestran algunos resultados que demuestran que el *algoritmo propuesto* es tan eficaz como la mejores heurísticas y metaheurísticas; es decir, el *algoritmo propuesto* cuenta con las ventajas de los dos tipos de algoritmos antes descritos: encuentra soluciones aproximadas para instancias arbitrarias y, además, estas soluciones tienden a ser las mejores conocidas.

La técnica específica de *aleatorización de algoritmos de aproximación*, propuesta y utilizada en la presente tesis, consiste en realizar una búsqueda probabilística, dentro del espacio de soluciones, aprovechando el conocimiento previo con que se cuenta; es decir, aprovechando la estructura que es explotada por el mejor algoritmo de aproximación conocido para el problema que se esté abordando (esta técnica no es de uso exclusivo del problema de selección de k-centros).

El *algoritmo propuesto*, que será definido más adelante, realiza una búsqueda aleatoria con base en el determinismo del algoritmo de González y con base en una variable de entrada α definida por el usuario, la cual brinda cierta flexibilidad en la búsqueda de soluciones, tratando de conservar un *equilibrio* entre aleatoriedad y determinismo. En la Figura 3.7(d) se aprecia una descripción gráfica de la manera en que el *algoritmo propuesto* realiza la búsqueda dentro del espacio de soluciones, donde la intersección entre el conjunto Opt y el conjunto Gon no necesariamente es igual al conjunto vacío.

El *algoritmo propuesto* tiende a encontrar soluciones cercanas a la óptima, pues explota la estrucutra utilizada por el algoritmo de González, el cual además es uno de los mejores algoritmos posibles (si $P \neq NP$), pues encuentra una 2-aproximación en tiempo polinomial (encontrar una ρ-aproximación, para valores de ρ menores a 2, es un problema NP-*Difícil*). De hecho, de acuerdo con los resultados obtenidos (Capítulo 5) y lo planteado por la Figura 3.7, es válido afirmar que el *algoritmo propuesto* tiende a encontrar soluciones cercanas al vecindario del conjunto de soluciones entregadas por el algoritmo de González (o González+, como se verá más adelante). Si bien el *algoritmo propuesto* tiende a encontrar soluciones cercanas a la óptima, su eficacia puede ser incrementada al utilizar una variante denominada *algoritmo propuesto con memoria*, la cual recibe como entrada la mejor solución encontrada previamente (más adelante son definidos este par de algoritmos). En el Capítulo 5 se muestran algunos resultados que demuestran lo afirmado en este párrafo.

4.2. Generalización del algoritmo de González

El *algoritmo propuesto* es un algoritmo aleatorizado que tiene como base al algoritmo determinista propuesto por González. Como se demostró en el Capítulo 3, el algoritmo de González encuentra una 2-aproximación para el problema de selección de k-centros en sus versiones tanto *continua* como *discreta*. El algoritmo de González funciona de manera similar al algoritmo de Kleinberg y Tardos, siendo la diferencia que el primero encuentra una partición de tamaño k y el segundo un conjunto de k nodos. Sin embargo, los nodos *head* de cada *cluster* entregado por el algoritmo de González conforman justamente el conjunto de nodos que entrega el algoritmo de Kleinberg y Tardos. Es por ello que comúnmente los algoritmo de González y de Kleinberg-Tardos son implementados de la misma manera.

La idea básica del *algoritmo propuesto* es la siguiente: construir el conjunto de centros C seleccionando k nodos de manera aleatoria, con base en una función de masa de probabilidad que es definida por una variable α ingresada por el usuario. El algoritmo de González elige siempre al nodo más alejado o, de manera equivalente, elige al nodo más alejado con probabilidad de 1, mientras que el *algoritmo propuesto* elige al nodo más alejado con *cierta probabilidad* (definida por la función de masa de probabilidad utilizada), la cual no necesariamente será igual a 1. Más aún, todos los nodos tendrán asignado un valor de probabilidad estrictamente mayor a cero, de manera que cualquier nodo es susceptible de ser considerado como parte de la solución C, lo cual es algo que no ocurre con el algoritmo de González.

El algoritmo denominado *generalización del algoritmo de González* (GGon) es una versión más general, tanto del algoritmo de González como del *algoritmo propuesto*, pues permite utilizar cualquier función de masa de probabilidad, mientras que los dos primeros utilizan funciones específicas.

A continuación se muestra nuevamente el algoritmo de González y Kleinberg-Tardos, así como la *generalización del algoritmo de González* (GGon), el cual es un algoritmo que permite relajar el determinismo del algoritmo de González (Gon). Al observar ambos algoritmos resulta evidente su similitud, pues la estructura general de ambos es idéntica. El *algoritmo propuesto* es básicamente el algoritmo GGon que hace uso de una función de masa de probabilidad específica que será descrita más adelante.

Algoritmo 4.1: Algoritmo de González y Kleinberg-Tardos (Gon)

Entrada: un grafo $G = (V, E)$ y un entero k
Salida: un conjunto C, donde $|C| \leq k$
1 **if** $k = |V|$ **then**
2 | $C = V$ y termina el algoritmo ;
3 **end**
4 Asignar a cada nodo $v \in V$ una probabilidad $P(v) = \frac{1}{n}$;
5 ($P(v)$ representa la probabilidad de elegir al nodo v) ;
6 Seleccionar un nodo $v \in V$ con base en $P(v)$;
7 $C = \{v\}$;
8 **while** $|C| < k$ **do**
9 | Seleccionar un nodo $v \in (V - C)$ tal que $dist(v, C)$ sea máxima ;
10 | $C = C \cup \{v\}$;
11 **end**
12 **return** C;

Algoritmo 4.2: Generalización del algoritmo de González (GGon)

Entrada: un grafo $G = (V, E)$ y un entero k
Salida: un conjunto C, donde $|C| \leq k$
1 **if** $k = |V|$ **then**
2 | $C = V$ y termina el algoritmo ;
3 **end**
4 Asignar a cada nodo $v \in V$ una probabilidad $P(v) = \frac{1}{n}$;
5 ($P(v)$ representa la probabilidad de elegir al nodo v) ;
6 Seleccionar un nodo $v \in V$ con base en $P(v)$;
7 $C = \{v\}$;
8 **while** $|C| < k$ **do**
9 | Asignar a cada nodo $v \in V$ una probabilidad $P(v)$ con base en una Función de Masa de Probabilidad determinada ;
10 | Seleccionar un nodo $v \in V$ con base en $P(v)$;
11 | $C = C \cup \{v\}$;
12 **end**
13 **return** C;

La *generalización del algoritmo de González* (Algoritmo 4.2) está definida de tal manera que es posible utilizar cualquier función de masa de probabilidad (F.M.P.) para definir el valor $P(v)$ correspondiente a cada nodo $v \in V$. Basta observar el algoritmo de González y en GGon para notar que su estructura es similar, siendo la *generalización del algoritmo de González* precisamente una versión más general del algoritmo de González. De hecho, el algoritmo de González consiste de la *generalización del algoritmo de González* que hace uso de una función de masa de probabilidad que asigna siempre una probabilidad de 1 al nodo más alejado.

La generalización del algoritmo de González puede utilizar cualquier función de masa de probabilidad, siempre y cuando se trate de una función válida. Una función válida es aquella que cumple con los axiomas de la Probabilidad (aditividad, no-negatividad y normalización). Para corroborar que la estructura explotada por el algoritmo de González es (o tiende a ser) una buena alternativa, se ejecutó el algoritmo GGon con las siguientes funciones de masa de probabilidad válidas:

- *Distribución uniforme* (DU). Para todas las iteraciones, todos los nodos tienen la misma probabilidad de ser seleccionados como centros (a excepción de los nodos que ya forman parte de la solución, a los cuales se les asigna una probabilidad de 0).

- *Distribución basada en la distancia hacia la solución* (BD1). Para todas las iteraciones, la probabilidad asignada a cada nodo depende directamente de su distancia a la solución parcial construida en la iteración anterior (en la primera iteración la distribución de probabilidad es uniforme).

- *Distribución basada en la razón de dos distancias* (BD2). Para todas las iteraciones, la probabilidad asignada a cada nodo depende directamente de la razón entre su distancia a la solución parcial construida en la iteración anterior y su distancia al nodo más alejado de la solución parcial construida en la iteración anterior (en la primera iteración la distribución de probabilidad es uniforme). Si el valor de esta razón es mayor a 1 (o un valor indeterminado) se sustituye por su inversa. En la Figura 4.1 se muestra un ejemplo de cómo determinar la probabilidad de cada nodo, donde y es una variable de normalización y el centro seleccionado en la primera iteración fue el nodo b.

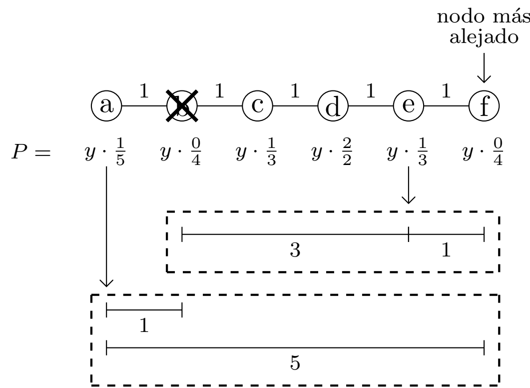


Figura 4.1: Función de masa de probabilidad basada en la razón de dos distancias (BD2)

En la Figura 4.2 se muestra un ejemplo donde se utilizan las tres funciones de masa de probabilidad descritas. En dicha figura se puede apreciar que el nodo más alejado tiene mayor probabilidad de ser seleccionado cuando se utiliza la función BD1, mientras que al utilizar la función BD2 el nodo más cercano al centro (entre la solución y el nodo más alejado) tiene mayor probabilidad de ser seleccionado.

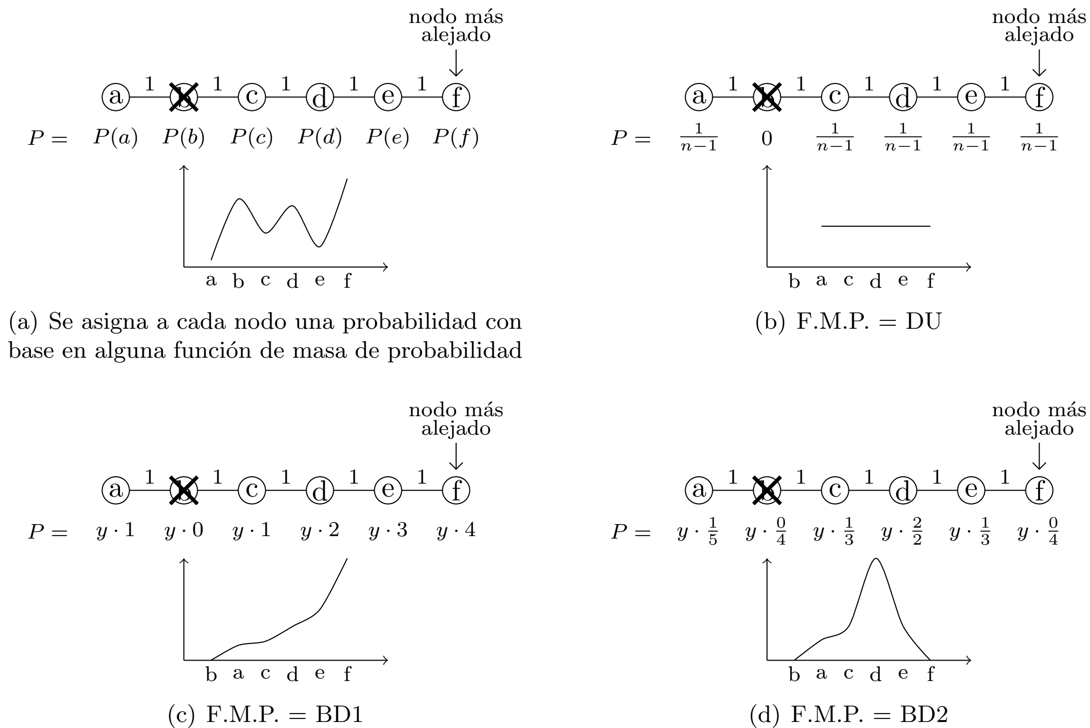


Figura 4.2: Funciones de masa de probabilidad con las que se experimentó sobre la generalización del algoritmo de González (GGon).

La generalización del algoritmo de González (GGon) fue ejecutada utilizando las tres F.M.P. definidas (DU, BD1 y BD2) sobre un conjunto de 40 grafos estándar[1] definidos en [24]. Estos grafos contienen desde 100 hasta 900 nodos, con valores de k desde 5 hasta 200. En la

[1] La información detallada sobre este conjunto de grafos se puede consultar en el Capítulo 5

Tabla 5.1 se muestran los resultados obtenidos al ejecutar cada algoritmo sobre cada grafo, incluyendo la variante (n), la cual consiste en ejecutar n veces el algoritmo, tomando como salida la mejor de las n soluciones.

Es importante señalar que los grafos utilizados cuentan con la peculiaridad de que una arista puede tener asignado más de un peso (probablemente se trate de un error por parte de J. E. Beasly, quien es el autor de estos grafos), por lo cual es necesario determinar cuál peso será seleccionado para definir el grafo. En [25] J. E. Beasly recomienda utilizar el último peso que aparece en el archivo de texto que define al grafo. En los experimentos realizados en esta tesis se siguió esta recomendación, la cual también es respetada por la mayoría de los artículos del estado del arte [13–16, 20, 27, 29, 36–38], siendo el artículo de Pacheco y Casado [26] la excepción. Es importante destacar esto, ya que de lo contrario se corre el riesgo de realizar comparaciones con base en grafos que en realidad son diferentes; justamente en la referencia [26] se aprecia esta confusión.

Los resultados entregados al utilizar la función BD1 tienden a ser mejores que los entregados por las funciones DU y BD2, lo cual era de esperarse, pues la función BD1 aprovecha de una manera más inteligente la estructura que es explotada por el algoritmo de González. La desventaja de utilizar estas tres funciones consiste en que resulta imposible determinar la probabilidad de encontrar soluciones con ciertas características para instancias arbitrarias, pues su comportamiento depende más de la estructura de la instancia de entrada que de la estructura misma del algoritmo o del problema. Es por lo anterior que se definió una cuarta función de masa de probabilidad, la cual es definida y caracterizada en la siguiente sección. La ventaja de utilizar esta cuarta función de masa de probabilidad, denominada función *Caracterizable* (C), consiste en que permite que el *algoritmo propuesto* encuentre soluciones que son una 2-aproximación con una alta probabilidad (al menos 0.63) para instancias arbitrarias. Parte de esta función de masa de probabilidad es construida de la misma manera que la función de masa BD1, pues esta función aprovecha más eficazmente la estructura del problema que es explotada por el algoritmo de González (lo cual se puede verificar al observar los resultados de la Tabla 4.1 y la Figura 4.3).

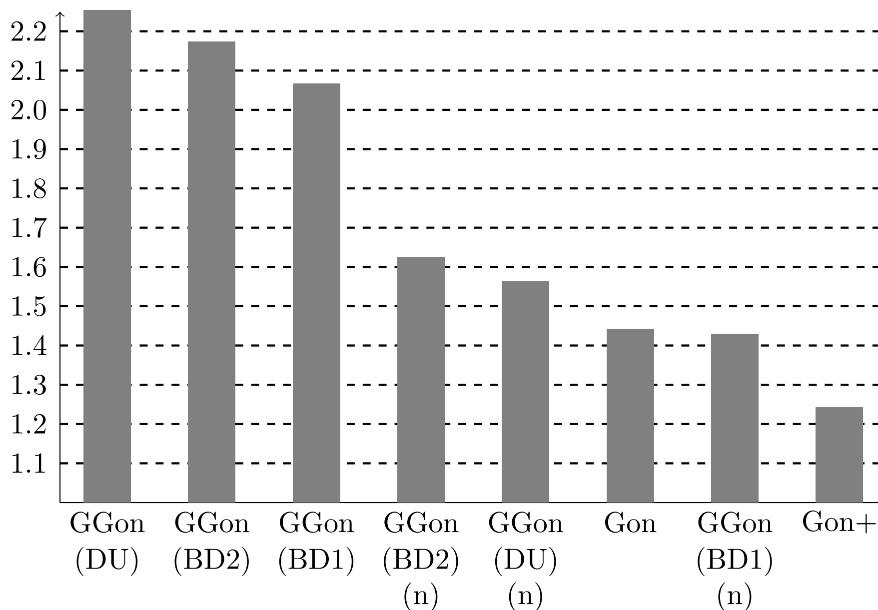


Figura 4.3: Calidad de las soluciones entregadas por el algoritmo GGon con diferentes funciones de masa de probabilidad

Tabla 4.1: Eficacia experimental del algoritmo GGon (generalización del algoritmo de González) utilizando diferentes funciones de masa de probabilidad

grafo	n	k	Mejor solución conocida	Gon	Gon+	GGon (DU)	GGon (BD1)	GGon (BD2)	GGon (DU) (n)	GGon (BD1) (n)	GGon (BD2) (n)
1	100	5	127	191	155	156	178	168	149	148	140
2	100	10	98	143	117	169	139	163	131	127	130
3	100	10	93	133	124	217	177	162	140	124	131
4	100	20	74	101	93	152	155	140	115	101	102
5	100	33	48	66	62	94	95	91	80	66	70
6	200	5	84	138	98	119	116	102	99	99	100
7	200	10	64	102	83	92	105	108	83	84	76
8	200	20	55	82	68	101	102	102	89	70	83
9	200	40	37	57	49	103	98	76	66	60	65
10	200	67	20	31	29	76	61	72	48	35	47
11	300	5	59	80	68	88	76	81	68	68	68
12	300	10	51	72	66	98	77	96	71	68	73
13	300	30	35	54	48	60	66	67	55	52	55
14	300	60	26	41	36	81	81	81	56	47	59
15	300	100	18	25	23	62	43	54	40	33	39
16	400	5	47	74	52	73	70	73	53	55	55
17	400	10	39	57	49	51	67	54	49	48	48
18	400	40	28	44	39	67	66	61	45	45	48
19	400	80	18	29	26	42	44	43	35	32	31
20	400	133	13	19	17	44	39	51	34	29	32
21	500	5	40	58	45	55	55	57	47	46	45
22	500	10	38	61	47	79	57	75	50	47	49
23	500	50	22	32	30	45	48	45	36	35	35
24	500	100	15	22	21	48	37	37	31	26	28
25	500	167	11	16	15	35	39	50	31	23	33
26	600	5	38	59	43	50	53	56	44	45	45
27	600	10	32	49	38	63	64	58	40	40	39
28	600	60	18	29	25	68	64	57	31	29	36
29	600	120	13	19	18	37	36	40	31	25	28
30	600	200	9	13	13	42	42	43	25	21	26
31	700	5	30	41	36	44	47	40	34	35	34
32	700	10	29	40	36	88	94	90	37	37	72
33	700	70	15	26	23	39	33	33	27	27	26
34	700	140	11	17	16	35	32	41	26	22	29
35	800	5	30	42	34	45	44	38	34	36	35
36	800	10	27	45	34	61	60	61	36	36	42
37	800	80	15	24	23	43	28	33	27	25	26
38	900	5	29	42	31	56	58	48	36	39	40
39	900	10	23	32	28	93	38	89	31	30	74
40	900	90	13	21	20	35	30	29	23	23	23
Comp.				$O(kn)$	$O(kn^2)$	$O(kn)$	$O(kn)$	$O(kn)$	$O(kn^2)$	$O(kn^2)$	$O(kn^2)$

4.3. Algoritmo propuesto

El *algoritmo propuesto* consiste básicamente de la *generalización del algoritmo de González* que hace uso de una función de masa de probabilidad específica, la cual es denominada función *Caracterizable* (C). La función *Caracterizable*, a diferencia de las funciones BD1 y BD2, no se construye con base en la estructura propia de cada instancia, sino que se construye con base en la estructura que es explotada por el algoritmo de González (estructura que existe para cualquier instancia) y con base en una variable α establecida por el usuario.

La función *Caracterizable* es construida de tal manera que, al repetir varias veces el algoritmo GGon, este tiende a encontrar soluciones *cercanas* a las que encuentra el algoritmo de González (es decir, con radios de cobertura cercanos a los de las soluciones que entrega el algoritmo de González).

La función *Caracterizable* consiste básicamente en: para cada iteración, asignar al nodo más alejado (de la solución parcial construida en la iteración previa) una probabilidad que garantice que, al repetirse el algoritmo una cierta cantidad de veces (justamente esta cantidad de veces es definida por la variable α), la solución encontrada será igual a la que encuentra el algoritmo de González con una probabilidad de al menos $1 - \frac{1}{e} \approx 0.63$. La estrategia seguida para determinar este valor está basada en el diseño de algoritmos aleatorios tipo Monte Carlo, en particular en algoritmos como el del Corte Mínimo [41]. A continuación se describe este método:

1. Se establece el mínimo número de veces (α) que se desea repetir el *algoritmo propuesto*. Para evitar que la complejidad del algoritmo amplificado resultante sea muy alta, se sugiere elegir valores polinomiales con respecto al tamaño de la entrada. En el presente trabajo se realizaron experimentos con $\alpha = n$.

2. Para lograr que la probabilidad de *éxito* (donde el evento *éxito* es equivalente al evento *encontrar la misma solución que el algoritmo de González*) del algoritmo amplificado sea de al menos $1-\frac{1}{e}$ es necesario observar que cada ejecución del algoritmo es independiente de las demás; por lo tanto, la probabilidad de *error* del algoritmo amplificado α veces es igual al producto de las probabilidades de *error* de cada ejecución:

$$P[error]_{\alpha_repeticiones} = \prod_{i=1,..,\alpha} P[error]_i \tag{4.1}$$

Dado que la probabilidad de *error* será la misma para cada ejecución, la ecuación 4.1 se puede reescribir de la siguiente manera:

$$P[error]_{\alpha_repeticiones} = (P[error])^{\alpha} \tag{4.2}$$

Por lo tanto, la probabilidad de *éxito* del algoritmo amplificado α veces es:

$$P[\acute{e}xito]_{\alpha_repeticiones} = 1 - P[error]_{\alpha_repeticiones} = 1 - (P[error])^{\alpha} \tag{4.3}$$

Para que la probabilidad de *éxito* sea al menos igual a $1-\frac{1}{e}$ es necesario que $(P[error])^{\alpha}$ sea igual a $\frac{1}{e}$. Para lograr esto último se puede aprovechar la siguiente propiedad:

$$\left(1 - \frac{1}{\alpha}\right)^{\alpha} < \frac{1}{e} \quad , \quad \alpha \geq 1 \tag{4.4}$$

Lo cual se debe a que el valor de $(1-\frac{1}{\alpha})^{\alpha}$, donde $\alpha \geq 1$, se encuentra siempre dentro del rango $(0, \frac{1}{e})$ (véase la Figura 4.4):

$$\lim_{\alpha \to \infty} \left(1 - \frac{1}{\alpha}\right)^{\alpha} = \frac{1}{e} \tag{4.5}$$

Es decir, es necesario construir la F.M.P. de tal manera que $P[error]$ sea menor o igual a $1 - \frac{1}{\alpha}$.

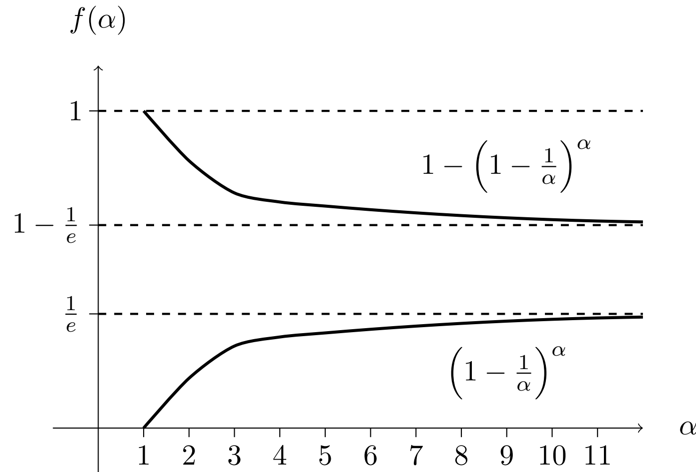


Figura 4.4: Funciones $\left(1-\frac{1}{\alpha}\right)^{\alpha}$ y $1-\left(1-\frac{1}{\alpha}\right)^{\alpha}$ dentro del rango $[1,\infty)$

3. Sea ε_i el evento: *seleccionar como centro en la i-ésima iteración un nodo que pertenezca a una 2-aproximación, donde los centros previamente seleccionados pertenecen a esa misma 2-aproximación.* Dado que cada evento ε_i es dependiente de los demás, la probabilidad de *éxito* del *algoritmo propuesto* está dada por la *regla de la multiplicación*:

$$P[\acute{e}xito] = P[\varepsilon_1]P[\varepsilon_2|\varepsilon_1]\ldots P\left[\varepsilon_k \mid \bigcap_{j=1}^{k-1}\varepsilon_j\right] \tag{4.6}$$

   En el algoritmo de González los valores de $P\left[\varepsilon_i \mid \bigcap_{j=1}^{i-1}\varepsilon_j\right]$, para $i = 0,1,...,k-1$, son iguales a 1. En el *algoritmo propuesto* se utiliza una función específica (dependiente de α) que, aprovechando la propiedad señalada en la ecuación (4.4), da como resultado una probabilidad de error, $P[error]$, menor o igual a $1-\frac{1}{e}$. En el caso del problema de selección de k-centros la F.M.P. que cumple con lo requerido es cualquiera que asigne al nodo más alejado una probabilidad de al menos $\frac{1}{k-\sqrt[1]{\alpha}}$. En la sección 4.3.2 se puede verificar esta última afirmación.

A continuación se muestran el *algoritmo propuesto* (Algoritmo 4.3) y el *algoritmo propuesto con memoria* (Algoritmo 4.5), el cual es una subrutina de la heurística 4.4. La descripción de la función de masa de probabilidad utilizada (líneas 9-10 del Algoritmo 4.3 y líneas 6-7 del Algoritmo 4.5), denominada función *Caracterizable*, se encuentra en la siguiente sección.

Algoritmo 4.3: Algoritmo propuesto

Entrada: un grafo $G = (V, E)$, un entero k y un entero α
Salida: un conjunto C, donde $|C| \leq k$
1 **if** $k = |V|$ **then**
2 $\quad C = V$ y termina el algoritmo ;
3 **end**
4 Asignar a cada nodo $v \in V$ una probabilidad $P(v) = \frac{1}{n}$;
5 ($P(v)$ representa la probabilidad de elegir al nodo v) ;
6 Seleccionar un nodo $v \in V$ con base en $P(v)$;
7 $C = \{v\}$;
8 $i = 1$;
9 **while** $|C| < k$ **do**
10 $\quad$ Asignar al nodo más alejado de C (llámese u) una probabilidad de al menos $\frac{1}{k - \sqrt[1]{\alpha}}$;
11 $\quad$ Asignar a cada nodo $v \in V - C$ una probabilidad $dist(v, C) \cdot y$;
12 $\quad$ donde $y = \frac{1 - P(u)}{\sum_{v \in V - C, v \neq u} dist(v, C)}$;
13 $\quad$ Seleccionar un nodo $v \in V - C$ con base en $P(v)$;
14 $\quad C = C \cup \{v\}$;
15 $\quad i = i + 1$;
16 **end**
17 **return** C;

Algoritmo 4.4: Heurística para utilizar el *algoritmo propuesto con memoria*

Entrada: un grafo $G = (V, E)$, un entero k y un entero α
Salida: un conjunto C, donde $|C| \leq k$
1 Se ejecuta el algoritmo propuesto (Algoritmo 4.3) amplificado sobre el grafo G con la variable α;
2 $C' \leftarrow$ Se guarda la mejor solución obtenida ;
3 **repeat**
4 $\quad$ Se ejecuta el algoritmo propuesto con memoria (Algoritmo 4.5) amplificado con la solución C' y la variable α como entrada ;
5 $\quad$ **if** *la nueva mejor solución obtenida es mejor que* C' **then**
6 $\quad\quad C' \leftarrow$ Se guarda la mejor solución obtenida ;
7 $\quad$ **end**
8 **until** *determinada condición de paro*;
9 $C \leftarrow C'$

Algoritmo 4.5: Algoritmo propuesto con memoria

Entrada: un grafo $G = (V, E)$, un entero k, una solución C' y un entero α
Salida: un conjunto C, donde $|C| \leq k$

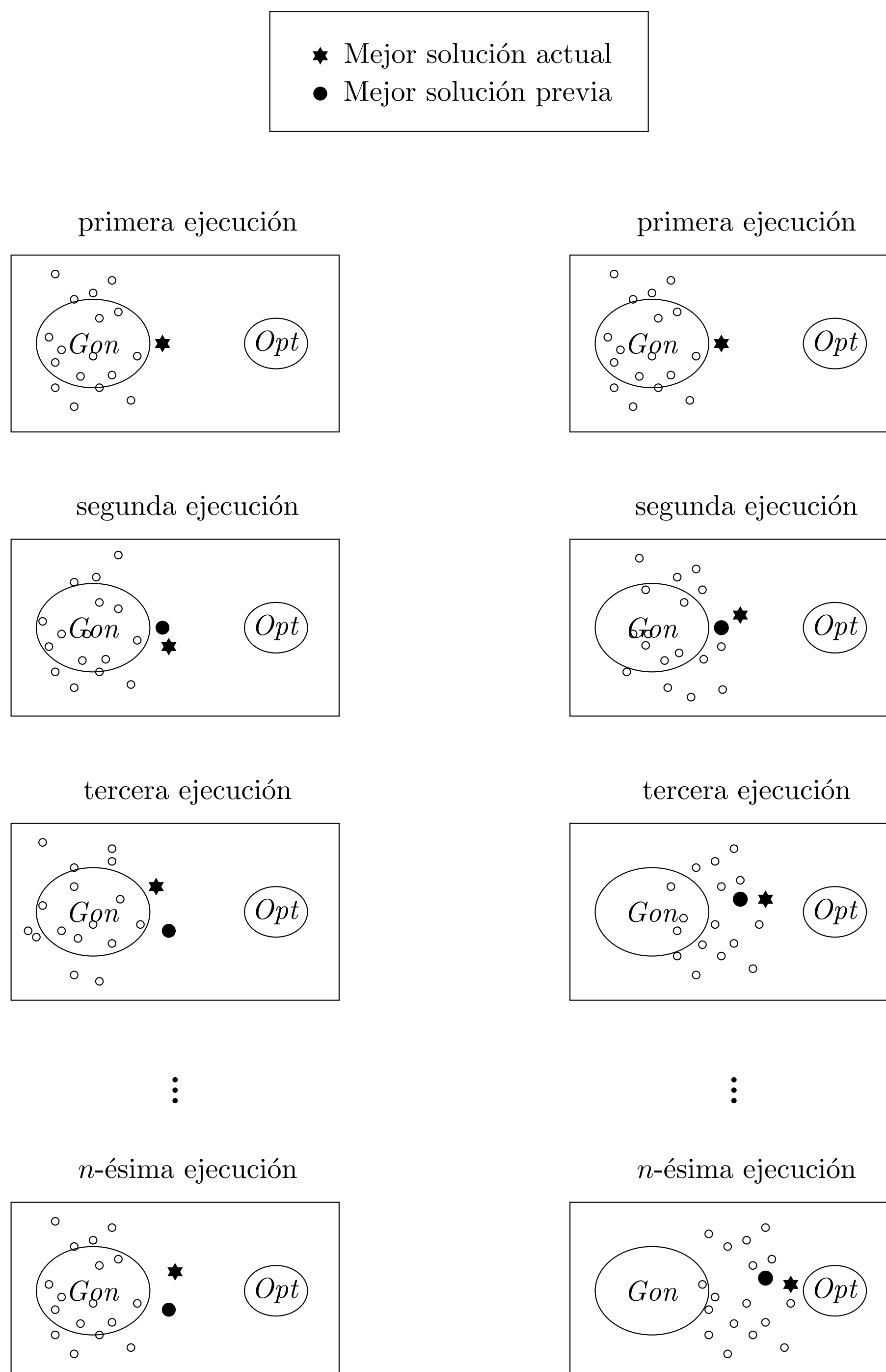
1. Asignar a cada nodo $v \in C'$ una probabilidad $P(v) = \frac{1}{|C'|}$;
2. Seleccionar un nodo $v \in C'$ con base en $P(v)$;
3. $C = \{v\}$;
4. $i = 1$;
5. **while** $|C| < k$ **do**
6. Asignar a cada nodo en $C' - C$ una probabilidad $P(C' - C)$ de al menos $\left(\frac{1}{k - \sqrt[i]{\alpha}}\right) / |C' - C|$;
7. Asignar a cada nodo $v \in V - C' - C$ restante una probabilidad $dist(v, C) \cdot y$;
8. donde $y = \frac{1 - P(C' - C)}{\sum_{v \in V - C' - C} dist(v,C)}$;
9. Seleccionar un nodo $v \in V - C$ con base en $P(v)$;
10. $C = C \cup \{v\}$;
11. $i = i + 1$;
12. **end**
13. **return** C;

La ventaja de utilizar el *algoritmo propuesto con memoria*, junto con la heurística 4.4, radica en que permite que iteración tras iteración se acumule un *mejor* conocimiento de la estructura que es conveniente explotar. En la Figura 4.5 se aprecia un ejemplo gráfico de cómo el *algoritmo propuesto* y el *algoritmo propuesto con memoria* (como subrutina de la heurística 4.4) realizan la búsqueda de soluciones dentro del espacio de búsqueda del problema. La Figura 4.5(a) permite observar que cada vez que el *algoritmo propuesto* amplificado es ejecutado se genera un conjunto de soluciones, de las cuales solo la mejor es conservada; ésta mejor solución bien puede ser mejor, igual o peor que la mejor solución obtenida en las ejecuciones anteriores, pues no existe ninguna dependencia entre cada ejecución del algoritmo amplificado. En cambio, la Figura 4.5(b) permite observar que cada ejecución del *algoritmo propuesto con memoria* amplificado es dependiente de las anteriores, pues realiza la búsqueda de soluciones alrededor de la mejor solución encontrada previamente, lo cual le permite aproximarse más *rápidamente* a la solución óptima. El *algoritmo propuesto con memoria* requiere una solución como parte de la entrada; es por ello que el texto de la Figura 4.5(b) señala que se trata de $n - 1$ repeticiones del *algoritmo propuesto con memoria* amplificado, pues la primera ejecución corresponde a algún otro algoritmo, el cual se encarga de generar la primera solución que sirve de entrada para el *algoritmo propuesto con memoria* amplificado.

4.3.1. Función de masa de probabilidad *Caracterizable*

La función de masa de probabilidad utilizada en el *algoritmo propuesto* garantiza que en cada iteración se selecciona exactamente un nodo, pues de lo contrario el conjunto solución C entregado por el algoritmo podría tener cardinalidad mayor a k. Para garantizar esto se asigna a cada nodo $v \in V - C$ un valor $P(v)$, el cual representa la probabilidad de que v sea agregado al conjunto de centros. La probabilidad de seleccionar más de un nodo en cada iteración es de 0, lo cual resulta evidente por observación directa del pseudocódigo del *algoritmo propuesto*.

Al inicio del *algoritmo propuesto*, el conjunto solución es igual al conjunto vacío, de modo que se puede considerar que todos los nodos de V están a la misma distancia de C, y es por



(b) n repeticiones del *algoritmo propuesto* amplificado

(c) $n-1$ repeticiones del *algoritmo propuesto con memoria* amplificado

Figura 4.5: Búsqueda probabilística del *algoritmo propuesto* con y sin *memoria*

ello que se asigna a todos los nodos de V la misma probabilidad de ser seleccionados: $\frac{1}{n}$. En las siguientes iteraciones el conjunto C es diferente al conjunto vacío, de modo que la probabilidad que se asigna a cada nodo depende de su distancia al conjunto C, exceptuando al nodo más alejado, al cual se le asigna una probabilidad de al menos $\frac{1}{\sqrt[k-1]{\alpha}}$. Es necesario asignar esta probabilidad al nodo más alejado para poder asegurar que el algoritmo *amplificado* α veces encontrará una 2-aproximación con *cierta probabilidad* (más adelante se calcula el valor de esta probabilidad).

Es decir, la función de densidad de probabilidad utilizada en la primera iteración ($i = 0$) es la siguiente:

$$P(x) = \begin{cases} \frac{1}{n} & , \quad x = v_1 \\ \frac{1}{n} & , \quad x = v_2 \\ \frac{1}{n} & , \quad x = v_3 \\ \vdots & \\ \frac{1}{n} & , \quad x = v_n \end{cases} \tag{4.7}$$

donde los subíndices de cada nodo $v \in V$ son solamente una etiqueta de los nodos que forman parte de $V - C$ (nodo 1, nodo 2, etcétera).

La función definida en 4.7 es una función de masa de probabilidad válida, pues cumple con las condiciones necesarias para ser una Ley de Probabilidad (aditividad, no-negatividad y normalización):

$$\sum_{v \in V} P(v) = n\left(\frac{1}{n}\right) = 1 \tag{4.8}$$

Para las iteraciones donde $C \neq \emptyset$, es decir para valores de $i \geq 1$, la función de masa de probabilidad utilizada es la siguiente:

$$P(x) = \begin{cases} dist(v_1, C) \cdot y_i & , \quad x = v_1 \\ dist(v_2, C) \cdot y_i & , \quad x = v_2 \\ \vdots & \\ \geq \frac{1}{\sqrt[k-1]{\alpha}} & , \quad x = v_j \\ \vdots & \\ dist(v_{n-i}, C) \cdot y_i & , \quad x = v_{n-i} \end{cases} \tag{4.9}$$

donde:

$$y_i = \frac{1 - P(v_j)}{\sum_{v \in V - C, v \neq v_j} dist(v, C)} \tag{4.10}$$

donde los subíndices de cada nodo $v \in V - C$ son solamente etiquetas, y el nodo v_j representa al nodo más alejado del conjunto C (si hubiera más de un nodo más alejado se elige arbitrariamente cualquiera de ellos). La variable y_i permite normalizar los valores de probabilidad asignados, de manera que su suma sea igual a 1, y se calcula para cada una de la $k - 1$ iteraciones (en la primera iteración no es necesario calcular dicho valor, pues la función de masa de probabilidad utilizada es la 4.7). Sin duda, para que la función de masa de probabilidad sea válida, debe considerarse únicamente la raíz positiva de $\frac{1}{\sqrt[k-1]{\alpha}}$. Nótese que para valores de α mayores a k es posible sustituir la función (4.9) por la siguiente:

$$P(x) = \begin{cases} dist(v_1, C) \cdot y_i & , \quad x = v_1 \\ dist(v_2, C) \cdot y_i & , \quad x = v_2 \\ \vdots & \\ \frac{1}{k-1\sqrt[]{\alpha - i}} & , \quad x = v_j \\ \vdots & \\ dist(v_{n-i}, C) \cdot y_i & , \quad x = v_{n-i} \end{cases} \tag{4.11}$$

Para utilizar la función 4.11 es necesario que α sea mayor a k, pues de lo contrario el valor de $\alpha - i$ podria ser negativo, dando lugar a que $P(v_j)$ tome valores complejos en alguna(s) de las iteraciones del algoritmo, con lo cual la función de probabilidad dejaría de ser válida.

Al asignar al nodo más alejado una probabilidad de al menos $\frac{1}{k-1\sqrt[]{\alpha}}$ se garantiza que, si bien el algoritmo es susceptible de encontrar cualquier solución dentro del espacio de búsqueda del problema, al ser *amplificado* α veces será mayor la probabilidad de que ésta sea una 2-aproximación (esto se verá a detalle en la sección de caracterización del algoritmo). En la Figura 4.6 se muestra un ejemplo de cómo se distribuyen n soluciones entregadas por el *algoritmo propuesto* (donde cada ejecución consiste a su vez de n ejecuciones y la solución entregada es la mejor de éstas) con $\alpha = n$, así como la distribución de n soluciones entregadas por el algoritmo *GGon* (donde cada ejecución también consiste a su vez de n ejecuciones y la solución entregada es la mejor de éstas) utilizando las funciones de masa DU y BD1 sobre una instancia específica[2]. Cada solución es representada con un punto y los valores de la recta numérica son los tamaños de cada solución. Se puede apreciar cómo la función de masa 4.9 (que define a la función*Caracterizable*) tiende a entregar soluciones más próximas a la solución óptima. Cabe señalar que la complejidad del algoritmo GGon con DU, GGon con BD1 y del *algoritmo propuesto* amplificado α veces (con α igual a n) es la misma, $O(kn^2)$.

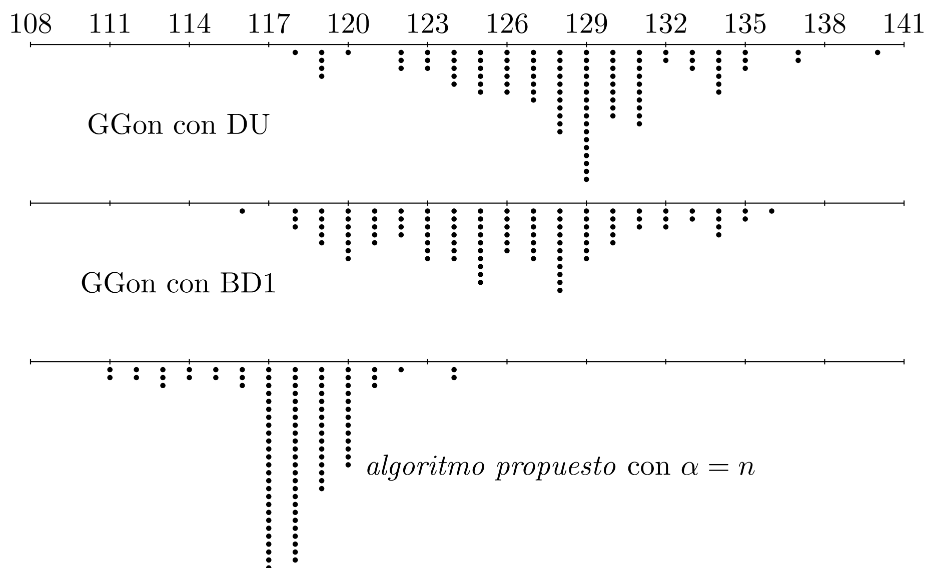


Figura 4.6: Distribución de las soluciones utilizando diferentes funciones de masa de probabilidad

[2]La instancia es el grafo *pmed2* del conjunto de 40 grafos de la librería OR-Lib. En la sección de Resultados se da toda la información relativa a este conjunto de grafos. El valor de n para el grafo *pmed2* es de 100.

La función 4.9 es una función de masa de probabilidad válida pues cumple con las condiciones de normalización, aditividad y no-negatividad:

$$\sum_{v\in V-C} P(v) = \left(\sum_{v\in V-C, v\neq v_j} dist(v,C)\cdot y\right) + \frac{1}{\sqrt[k-1]{\alpha}}$$
$$\sum_{v\in V-C} P(v) = y\left(\sum_{v\in V-C, v\neq v_j} dist(v,C)\right) + \frac{1}{\sqrt[k-1]{\alpha}} \tag{4.12}$$

Al sustituir la ecuación 4.10 en la 4.12 se obtiene lo siguiente:

$$\sum_{v\in V-C} P(v) = \frac{1-\frac{1}{\sqrt[k-1]{\alpha}}}{\sum_{v\in V-C, v\neq v_j} dist(v,C)}\left(\sum_{v\in V-C, v\neq v_j} dist(v,C)\right) + \frac{1}{\sqrt[k-1]{\alpha}}$$
$$\sum_{v\in V-C} P(v) = \left(1-\frac{1}{\sqrt[k-1]{\alpha}}\right) + \frac{1}{\sqrt[k-1]{\alpha}} \tag{4.13}$$
$$\sum_{v\in V-C} P(v) = 1$$

La no-negatividad se deriva del hecho de que el mínimo valor que puede tomar $dist(v,C)$ es cero (el cual sería el caso en que $v \in C$), los valores que puede tomar $\frac{1}{\sqrt[k-1]{\alpha}}$ son tales que $0 < \frac{1}{\sqrt[k-1]{\alpha}} \leq 1$ y, debido a lo anterior, el valor que puede tomar y_i es siempre positivo.

4.3.2. Caracterización teórica

El algoritmo de González entrega soluciones que son una 2-aproximación (Teorema 3.1.2), incluso sin importar qué nodo haya sido elegido al inicio del algoritmo como parte de la solución. Por lo tanto, al ejecutar el algoritmo de González, la probabilidad de que en la primera iteración se seleccione un nodo que forma parte de una 2-aproximación es de 1.

$$P[\varepsilon_1] = 1 \tag{4.14}$$

donde ε_1 representa al evento "*seleccionar un nodo que forme parte de una 2-aproximación en la primera iteración, donde los nodos previamente seleccionados forman parte de la misma 2-aproximación*" (para la primera iteración el conjunto solución generado previamente es el conjunto vacío).

Ahora bien, al ejecutar el algoritmo de González la probabilidad de elegir en cada iteración un nodo que forma parte de una 2-aproximación es siempre igual a 1; sin embargo, en el *algoritmo propuesto* la probabilidad de elegir un nodo que forma parte de una 2-aproximación (nótese que el nodo más alejado forma parte de una 2-aproximación) es mayor o igual a $\frac{1}{\sqrt[k-1]{\alpha}}$, siempre y cuando en las iteraciones anteriores se hayan elegido nodos que forman parte de la misma 2-aproximación, es decir:

$$P\left[\varepsilon_i \mid \bigcap_{j=1}^{i-1} \varepsilon_j\right] \geq \frac{1}{\sqrt[k-1]{\alpha}} \tag{4.15}$$

Teorema 4.3.1. *El algoritmo propuesto,* ***amplificado*** α ***veces****, entrega soluciones con un tamaño menor o igual a* $2\cdot r(OPT)$ *(iguales o mejores a las que entrega el algoritmo* ***González****) con una probabilidad de al menos* $1-\frac{1}{e}\approx 0.63$

Demostración. Con base en la regla de la multiplicación:

$$P\left[\bigcap_{i=1}^{n} A_i\right] = P(A_1) \cdot P(A_2 \mid A_1) \cdot P(A_3 \mid A_1 \cap A_2) \ldots P\left(A_n \mid \bigcap_{j=1}^{n-1} A_j\right) \tag{4.16}$$

es posible determinar la probabilidad de que el *algoritmo propuesto*, sin amplificar (es decir, en una sola ejecución), entregue la misma solución que entregaría el algoritmo de González:

$$\begin{aligned} P\left[\bigcap_{i=1}^{k} \varepsilon_i\right] &= P[\varepsilon_1] \cdot P[\varepsilon_2|\varepsilon_1] \ldots P\left[\varepsilon_k \mid \bigcap_{j=1}^{k-1} \varepsilon_j\right] \\ P\left[\bigcap_{i=1}^{k} \varepsilon_i\right] &\geq 1 \cdot \left(\prod_{j=1}^{k-1} \frac{1}{\sqrt[k-1]{\alpha}}\right) \\ P\left[\bigcap_{i=1}^{k} \varepsilon_i\right] &\geq \left(\prod_{j=1}^{k-1} \frac{1}{\sqrt[k-1]{\alpha}}\right) \end{aligned} \tag{4.17}$$

Por lo tanto, la probabilidad de que el *algoritmo propuesto* entregue la misma solución que entregaría el algoritmo de González es:

$$\begin{aligned} P\left[\bigcap_{i=1}^{k} \varepsilon_i\right] &\geq \left(\prod_{j=1}^{k-1} \frac{1}{\sqrt[k-1]{\alpha}}\right) = \frac{1}{\alpha} \\ P\left[\bigcap_{i=1}^{k} \varepsilon_i\right] &\geq \frac{1}{\alpha} \end{aligned} \tag{4.18}$$

Dado que todas las soluciones entregadas por el algoritmo González son una 2-aproximación, es válido afirmar que el *algoritmo propuesto* entrega soluciones que son una 2-aproximación con una probabilidad mayor o igual a $1/\alpha$ (para simplificar la notación se puede renombrar el evento "*encontrar una 2-aproximación*" con el nombre *"éxito"*). Por lo tanto, la probabilidad de que el *algoritmo propuesto* entregue soluciones con un tamaño superior a $2 \cdot r(OPT)$, es decir, que sean ρ-aproximaciones, donde $\rho > 2$ es:

$$\begin{aligned} P[error] &< 1 - P[éxito] \\ P[error] &< 1 - P\left[\bigcap_{i=1}^{k} \varepsilon_i\right] \\ P[error] &< 1 - \frac{1}{\alpha} \end{aligned} \tag{4.19}$$

Puesto que cada ejecución del *algoritmo propuesto* es independiente entre sí, es posible incrementar la probabilidad de *éxito* (donde el evento *éxito* es equivalente al evento "*encontrar soluciones que son una 2-aproximación*") al repetir varias veces el algoritmo. Si se repite varias veces el algoritmo, la probabilidad de *error* será igual al producto de las probabilidades de *error* de cada experimento. De manera que, al repetir α veces el *algoritmo propuesto*, la probabilidad de error es:

$$P[error]_\alpha < \left(1 - \frac{1}{\alpha}\right)^{\alpha} \tag{4.20}$$

dado que:

$$\left(1 - \frac{1}{\alpha}\right)^{\alpha} < \frac{1}{e}, \qquad , \qquad \alpha \geq 1 \tag{4.21}$$

la probabilidad de error del *algoritmo propuesto* amplificado α veces es:

$$P[error]_{\alpha} < \left(1 - \frac{1}{\alpha}\right)^{\alpha} < \frac{1}{e} \tag{4.22}$$

Por transitividad:

$$\begin{aligned} P[error]_{\alpha} &< \frac{1}{e} \\ P[error]_{\alpha} &\lesssim 0.3678 \end{aligned} \tag{4.23}$$

Por lo tanto, la probabilidad de *éxito* es:

$$\begin{aligned} P[\acute{e}xito]_{\alpha} &\gtrsim 1 - 0.3678 \\ P[\acute{e}xito]_{\alpha} &\gtrsim 0.6322 \end{aligned} \tag{4.24}$$

Dado que al repetir α veces el algoritmo se incrementa (*amplifica*) la probabilidad de *éxito*, es válido afirmar que el algoritmo ha sido amplificado, pues no se está repitiendo de manera injustificada.

□

Como se puede apreciar en la Tabla 4.2, el algoritmo González+ (o bien González *Plus*) tiende a entregar mejores soluciones que el algoritmo González, lo cual significa que la elección del primer nodo sí es relevante (para algunas instancias) y, en el peor de los casos, la selección de solamente uno de los α nodos como nodo inicial implica encontrar una mejor solución. Es por esto que resulta conveniente repetir más de α veces el algoritmo propuesto, con la intención de obtener soluciones iguales o mejores a las que encontraría el algoritmo González+.

Tabla 4.2: El algoritmo González+ tiende a ser mejor que el algoritmo González

grafo	tamaño de la solución	
	González	González+
pmed1	162	155
pmed2	134	117
pmed3	135	124
pmed4	107	93
pmed5	64	62

Teorema 4.3.2. *El* algoritmo propuesto, ***amplificado*** α^2 ***veces*** *(donde α es mayor o igual a n), entrega soluciones con un tamaño menor o igual a $2 \cdot r(OPT)$ (iguales o mejores a las que entrega el algoritmo* ***Gonzalez+****) con una probabilidad de al menos $1 - \frac{1}{e} \approx 0.63$*

Demostración. En la demostración del Teorema 4.3.1 se partió del hecho de que la probabilidad de elegir un nodo miembro de una 2-aproximación en la primera iteración es igual a 1 ($P[\varepsilon_1] = 1$). Sin embargo ahora se desea demostrar que la solución entregada por el *algoritmo propuesto* (amplificado α^2 veces, donde α es mayor o igual a n) es menor o igual a la que entrega el algoritmo González+. Es sabido que, dada una instancia de entrada, el algoritmo de González tiende a entregar una mejor solución si se elige como primer nodo uno de ellos en particular, por lo tanto la probabilidad de seleccionar este nodo en la primera iteración es de $P[\varepsilon_1] = \frac{1}{n}$. Repitiendo un proceso similar al de la demostración del teorema 4.3.1:

$$P\left[\bigcap_{i=1}^{k} \varepsilon_i\right] \geq \frac{1}{n} \cdot \left(\prod_{j=1}^{k-1} \frac{1}{\sqrt[k-1]{\alpha}}\right) \tag{4.25}$$

Claramente esta expresión puede reducirse a:

$$\begin{aligned} P\left[\bigcap_{i=1}^{k} \varepsilon_i\right] &\geq \frac{1}{n} \cdot \left(\prod_{j=1}^{k-1} \frac{1}{\sqrt[k-1]{\alpha}}\right) = \frac{1}{n} \cdot \frac{1}{\alpha} \\ P\left[\bigcap_{i=1}^{k} \varepsilon_i\right] &\geq \frac{1}{\alpha n} \end{aligned} \tag{4.26}$$

La probabilidad de que el *algoritmo propuesto* (sin amplificar y donde $\alpha \geq n$) entregue soluciones diferentes a las que entregaría el algoritmo González+ es:

$$\begin{aligned} P[error] &< 1 - P[éxito] \\ P[error] &< 1 - P\left[\bigcap_{i=1}^{k} \varepsilon_i\right] \\ P[error] &< 1 - \frac{1}{\alpha n} \end{aligned} \tag{4.27}$$

Por lo tanto, la probabilidad de *error* del algoritmo amplificado α^2 veces es:

$$P[error]_{\alpha^2} < \left(1 - \frac{1}{\alpha n}\right)^{\alpha^2} \tag{4.28}$$

Dado que $\alpha \geq n$ y $0 < 1 - \frac{1}{\alpha n} < 1$:

$$P[error]_{\alpha^2} < \left(1 - \frac{1}{\alpha n}\right)^{\alpha^2} \leq \left(1 - \frac{1}{\alpha n}\right)^{\alpha n} \tag{4.29}$$

Utilizando la ecuación 4.29 y la propiedad señalada en la ecuación 4.4:

$$\begin{aligned} P[error]_{\alpha^2} &< \left(1 - \frac{1}{\alpha n}\right)^{\alpha^2} \leq \left(1 - \frac{1}{\alpha n}\right)^{\alpha n} < \frac{1}{e} \\ P[error]_{\alpha^2} &< \frac{1}{e} \\ P[error]_{\alpha^2} &\lesssim 0.3678 \end{aligned} \tag{4.30}$$

De modo que la probabilidad de encontrar soluciones iguales o mejores a las que encuentra el algoritmo González+ es de:

$$\begin{aligned} P[\acute{e}xito]_{\alpha^2} &\gtrsim 1 - 0.3678 \\ P[\acute{e}xito]_{\alpha^2} &\gtrsim 0.6322 \end{aligned} \tag{4.31}$$

□

En la Tabla 4.3 se pueden apreciar algunos ejemplos que demuestran que el *algoritmo propuesto* (con $\alpha = n$) amplificado α^2 veces entrega mejores soluciones que cuando se le amplifica α veces, lo cual es de esperarse, pues al repetir más veces el algoritmo se realiza una exploración más extensa dentro del espacio de soluciones.

Tabla 4.3: El *algoritmo propuesto* amplificado α^2 veces tiende a ser mejor que cuando es amplificado α veces ($\alpha = n$)

grafo	amplificación α	amplificación α^2
pmed1	137	128
pmed2	119	111
pmed3	114	102
pmed4	89	81
pmed5	59	53

Resulta interesante observar que la garantía de que el *algoritmo propuesto* entrega una 2-aproximación depende únicamente de la probabilidad $\frac{1}{k-\sqrt[1]{\alpha}}$ asignada en cada iteración al nodo más alejado; de modo que la probabilidad asignada al resto de los nodos es irrelevante (siempre y cuando la función de masa de probabilidad sea válida); es decir, la probabilidad asignada al resto de los nodos bien puede ser la misma para todos (distribución uniforme) o puede depender de alguna propiedad específica (con base en la distancia u otra variable). En el *algoritmo propuesto* la probabilidad asignada a cada nodo se definió en términos de su distancia hacia la solución, pues se realizaron experimentos que demostraron que la solución encontrada con esta probabilidad (BD1) es mejor que al utilizar otras funciones (véase la Tabla 4.1).

4.3.3. Heurísticas de búsqueda local

Las heurísticas *Alternate* e *Interchange*, propuestas por Mladenovic [29], son un par de algoritmos de búsqueda local para el problema de selección de k-centros que pueden ejecutarse eficientemente. En el presente trabajo se propone utilizar este par de heurísticas modificadas para mejorar las soluciones entregadas por el algoritmo propuesto (ya sea amplificado o no), al aplicarlas sobre cada ejecución del mismo.

- Búsqueda local **(A)** . Consiste en resolver el problema de seleccion de 1-centro sobre cada agrupamiento de la partición generada por el algoritmo (dicha partición no se genera explícitamente en los Algoritmos 4.3 y 4.5; sin embargo, puede construirse sin incrementar el orden de su complejidad). En [29] se define el método *Alternate*, el cual consiste en resolver el problema de seleccion de 1-centro reiteradamente sobre una solución hasta que

deja de haber una mejora en la misma. El método (A) definido en este documento consiste en resolver el problema de seleccion de 1-centro, sobre cada agrupamiento, solamente una vez. La idea de utilizar la variante (A) surge de manera natural al observar que en la definición original del algoritmo de González [1] la solución entregada es una partición (un conjunto de agrupamientos) y no un conjunto de nodos. La complejidad de la variante (A) es $O(n^2)$. Cabe señalar que, cuando la distancia es euclidiana, la complejidad de la variante (A) se reduce a $O(n)$ [39].

- Búsqueda local **(A-I)** . Consiste en ejecutar la variante A, seguida de la variante I, la cual consiste en realizar el mejor intercambio entre un centro de la solución y un nodo fuera de la misma, de tal manera que la nueva solución sea mejor que la previa. Este método, denominado *Interchange*, es definido en [29]. La complejidad de la variante (I) es $O(n^2)$. En [29] se demuestra experimentalmente que las soluciones entregadas por la permutación (*Alternate-Interchange*) son mejores que las entregadas por la permutación (*Interchange-Alternate*), razón por la cual en la presente tesis se experimentó con la permutación (A-I).

- Repeticiones $(\boldsymbol{n})$. Esta variante no es una técnica de búsqueda local, pues consiste únicamente en ejecutar $\boldsymbol{n}$ veces el algoritmo, tomando como salida la mejor de las $\boldsymbol{n}$ soluciones.

- Repeticiones $(\boldsymbol{n^2})$. Esta variante no es una técnica de búsqueda local, pues consiste únicamente en ejecutar $\boldsymbol{n^2}$ veces el algoritmo, tomando como salida la mejor de las $\boldsymbol{n^2}$ soluciones.

En el Capítulo 5 se muestran algunos resultados que demuestran que el uso de las variantes (A) y (A-I) tiende a mejorar las soluciones entregadas tanto por los algoritmos del estado del arte como por el *algoritmo propuesto*.

4.3.4. Complejidad

La estructura del *algoritmo propuesto* es similar a la del algoritmo de González. La diferencia entre estos algoritmos radica en que el *algoritmo propuesto* asigna probabilidades a los nodos con base en la función de masa de probabilidad denominada *Caracterizable*. Para asignar a cada nodo un valor $P(v)$ es necesario conocer su distancia hacia la solución parcial C construida en la iteración anterior, la cual se puede calcular en un solo paso utilizando una matriz de adyacencias o alguna otra estructura eficiente. Por lo tanto, la complejidad del *algoritmo propuesto*, sin amplificar, es igual a la del algoritmo de González, es decir, $O(kn)$.

Si el *algoritmo propuesto* es amplificado α veces, su complejidad se incrementa en un factor de α. De manera que el *algoritmo propuesto*, amplificado α veces, tiene una complejidad de orden $O(\alpha kn)$, mientras que el *algoritmo propuesto*, amplificado α^2 veces, tiene una complejidad de orden $O(\alpha^2 kn)$.

En el Capítulo 5 se muestran los resultados obtenidos al ejecutar el *algoritmo propuesto* y el *algoritmo propuesto con memoria* sobre un conjunto de instancias de prueba, donde el valor de α utilizado fue igual a n. La complejidad del *algoritmo propuesto* amplificado α veces, donde $\alpha = n$, es de $O(kn^2)$. La complejidad del *algoritmo propuesto* amplificado α^2 veces, donde $\alpha = n$, es de $O(kn^3)$.

Al término de las ejecuciones del *algoritmo propuesto*, amplificado α veces, se cuenta con un conjunto de α soluciones. Por el Teorema 4.3.1, con probabilidad $1 - (1 - \frac{1}{\alpha})^\alpha \gtrsim 0.63$, al menos una de estas α soluciones es igual a la que entregaría el algoritmo González. Dado que

el objetivo es encontrar una solución C con un radio de cobertura $r(C)$ mínimo, se toma del conjunto de α soluciones aquella que tenga el radio de cobertura menor.

Si se aplican las técnicas de búsqueda local (A) y (A-I) sobre algún algoritmo, la complejidad de éste puede incrementar. En las Tablas 4.4 y 4.5 se muestra la complejidad del *algoritmo propuesto* y del agoritmo de González utilizando diferentes variantes.

Tabla 4.4: Complejidad del algoritmo de González (Gon)

Algoritmo	Complejidad
Gon	$O(kn)$
Gon (A)	$O(n^2)$
Gon (A-I)	$O(n^2)$
Gon+	$O(kn^2)$
Gon (A) +	$O(n^3)$
Gon (A-I) +	$O(n^3)$

Tabla 4.5: Complejidad del *algoritmo propuesto* (AP) con $\alpha = n$

Algoritmo	Complejidad
AP	$O(kn)$
AP (A)	$O(n^2)$
AP (A-I)	$O(n^2)$
AP (n)	$O(kn^2)$
AP (A) (n)	$O(n^3)$
AP (A-I) (n)	$O(n^3)$
AP (n^2)	$O(kn^3)$
AP (A) (n^2)	$O(n^4)$
AP (A-I) (n^2)	$O(n^4)$

4.4. Similitudes entre la técnica propuesta y algunas metaheurísticas del estado del arte

La técnica propuesta es similar a la técnica GRASP y a la del diseño de algoritmos EDAs en el sentido de que todas ellas giran en torno a la generación de soluciones con base en selecciones aleatorias; sin embargo, cada técnica lo hace de manera diferente:

- Los algoritmos EDAs requieren de una población inicial y, dado que la distribución de probabilidad que define a dicha población es desconocida, ésta es estimada. Con base en la estimación de la distribución de probabilidad se realiza un muestreo que da como resultado una nueva población, sobre la cual se puede repetir el procedimiento. La condición de paro se define con base en la experiencia del usuario o bien de manera arbitraria. Además, no es posible garantizar que la ejecución del algoritmo será eficiente, ni que el mínimo global o una ρ-aproximacion será encontrada.
- La técnica GRASP realiza una búsqueda aleatoria con base en el determinismo de un algoritmo voraz base y con base en una variable α definida arbitrariamente por el usuario. La función de la variable α es la de determinar el nivel de determinismo y aleatoriedad

del algoritmo (si $\alpha = 0$ el algoritmo es totalmente determinista, mientras que cuando $\alpha = 1$ el algoritmo es totalmente aleatorio). La condición de paro, así como el valor de la variable α, son definidas con base en la experiencia del usuario o bien de manera arbitraria. Además, no es posible garantizar que la ejecución del algoritmo será eficiente, ni que el mínimo global o una ρ-aproximacion será encontrada.

- La técnica propuesta define una función de masa de probabilidad (a diferencia de los algoritmos EDAs, no es necesario estimar esta función de masa de probabilidad, pues ésta se define desde un principio de manera explícita), con base en la cual se genera un conjunto de soluciones, las cuales no conforman una población, pues de hecho no es necesario almacenarlas, sino simplemente conservar la mejor de ellas. La función de masa de probabilidad con base en la cual se generan las soluciones depende de la estructura aprovechada por el algoritmo determinista base y del valor de una variable α definida por el usuario. Si bien la variable α se define arbitrariamente ($\alpha \in \mathbb{Z}^+$), la garantía teórica de la eficacia del algoritmo resultante se mantiene constante (siempre y cuando el algoritmo determinista base sea un algoritmo de aproximación). La complejidad de los algoritmos diseñados con la técnica propuesta es polinomial con respecto al tamaño de la entrada y al valor de la variable α. La garantía teórica de la eficacia del algoritmo resultante es heredada de la garantía teórica del algoritmo de aproximación base (i.e. si el algoritmo de aproximación base garantiza que la solucion entregada es una 2-aproximación, entonces el algoritmo diseñado con la técnica propuesta garantizará que la solución entregada es una 2-aproximación con una determinada probabilidad menor a 1).

4.5. Discusión

- El *algoritmo propuesto* fue diseñado siguiendo un esquema de diseño que, si bien presenta ciertas similitudes con algunas metaheurísticas del estado del arte (por ejemplo, con los algoritmos EDAs y la técnica GRASP), no parece corresponder de forma precisa a alguna de ellas. Es decir, tanto el *algoritmo propuesto* como la técnica utilizada para su diseño son una aportación original

- El *algoritmo propuesto* tiene como base al algoritmo de González, el cual es uno de los mejores algoritmos de aproximación, ya que encuentra una 2-aproximación en tiempo polinomial (encontrar mejores soluciones en tiempo polinomial demostraría que $P = NP$).

- El *algoritmo propuesto* realiza búsquedas aleatorias dentro del espacio de soluciones con base en la estructura que es explotada por el algoritmo de González y con base en una variable $\alpha \in \mathbb{Z}^+$ definida arbitrariamente por el usuario. La estructura explotada por el algoritmo de González es la siguiente: integrar a la solución al nodo más alejado de la solución parcial construida. La variable α define la probabilidad (de ser seleccionado) asignada al nodo más alejado en cada iteración del algoritmo; esta probabilidad es definida considerando que el objetivo deseado es que al menos α repeticiones del algoritmo garanticen que al menos una de las α soluciones será una 2-aproximación con una probabilidad constante para cualquier instancia. A este proceso de repetición se le denomina *amplificación*, pues permite incrementar la probabilidad de *éxito* de un evento específico.

- La probabilidad de *éxito* (es decir, de encontrar una 2-aproximación) del *algoritmo propuesto* amplificado α veces es mayor o igual a 0.63. Este valor es alcanzado gracias a que se aprovecha la siguiente propiedad:

$$\left(1 - \frac{1}{\alpha}\right)^{\alpha} < \frac{1}{e} \quad , \quad \alpha \geq 1 \tag{4.32}$$

La propiedad 4.32 es fácilmente adaptable al problema de selección de k-centros y es por ello que se utilizó para definir la función de masa, denominada *Caracterizable*, que es utilizada por el *algoritmo propuesto*.

Capítulo 5

Resultados

Los experimentos se realizaron sobre un conjunto de 40 grafos estándar, los cuales reciben el nombre *pmed1* hasta *pmed40*. Estos grafos fueron propuestos por J.E.Beasley [24] para caracterizar experimentalmente el desempeño de algoritmos diseñados para resolver el problema *k-median* (o *p-median*). A pesar de no haber sido específicamente diseñados para el problema de selección de k-centros, su uso como herramienta de comparación de algoritmos para este problema es muy amplio en la literatura [13, 14, 20, 22, 27, 29, 38].

Cada uno de los grafos está definido en un archivo de texto. Los elementos de cada archivo de texto son:

- En la primera línea:
  - Número de nodos.
  - Número de aristas.
  - Valor de k.
- Cada una de las siguientes líneas:
  - Nodo A.
  - Nodo B.
  - Costo de la arista (A,B)

Cada uno de los grafos definidos por J. E. Beasly deben ser grafos simétricos no dirigidos; sin embargo, aparentemente algunos de ellos fueron definidos erróneamente, pues una misma arista llega a tener más de un costo asociado. Para evitar confusiones J. E. Beasly sugiere considerar únicamente el último costo que aparezca en el archivo de texto [25]; sin embargo, diferentes autores suelen generar los grafos de manera distinta, ya sea considerando el primer peso que aparece en el archivo de texto o bien el último. Los experimentos mostrados en la presente tesis se llevaron a cabo respetando la sugerencia de J. E. Beasly. Es importante señalar la manera en que se construyen los grafos, pues pueden existir artículos del estado del arte donde no se aclare este detalle (un ejemplo es la referencia [26]), dando lugar a la posibilidad de realizar comparaciones inválidas entre algoritmos.

La entrada al problema de selección de k-centros es un grafo completo, por lo cual es necesario aplicar el algoritmo de Floyd-Marshall sobre cada grafo *pmed*. El algoritmo de Floyd-Marshall se encarga de calcular la distancia más corta entre todas las parejas de nodos [47], de tal manera que el grafo resultante sea un grafo completo cuyas aristas respetan la desigualdad del triángulo.

5.1. Resultados experimentales

En el Capítulo 3 se muestran algunos resultados obtenidos por los algoritmos del estado del arte (Tabla 3.5 y Figuras 3.8 y 3.9) al ser ejecutados sobre el conjunto de 40 grafos *pmed*. De acuerdo con dichos resultados, los mejores algoritmos polinomiales del estado del arte son el algoritmo Gon+ (González+ o González *Plus*) y el algoritmo Scr (*Scoring*); mientras que los mejores algoritmos no polinomiales son el algoritmo SS, TS-2 y Das.

En esta sección se muestran los resultados obtenidos por el *algoritmo propuesto* (para fines prácticos este algoritmo es abreviado como AP) y por los algoritmos Gon, Gon+, Scr, SS, TS-2 y Das. Sobre el *algoritmo propuesto* y los algoritmo polinomiales Gon, Gon+ y Scr se aplican diferentes variantes; de modo que los algoritmos cuyo desempeño fue analizado son los siguientes:

- **Gon**. Algoritmo de González.
- **Gon (A)**. Algoritmo de González utilizando la variante (A) [1].
- **Gon (A-I)**. Algoritmo de González utilizando la variante (A) seguida de la variante (I) [2].
- **Gon +**. Algoritmo de González repetido n veces, seleccionando en cada ejecución un nodo diferente como centro inicial.
- **Gon (A) +**. Algoritmo de González repetido n veces, seleccionando en cada ejecución un nodo diferente como centro inicial y utilizando la variante (A) en las n ejecuciones.
- **Gon (A-I) +**. Algoritmo de González repetido n veces, seleccionando en cada ejecución un nodo diferente como centro inicial y utilizando la variante (A), seguida de la variante (I), en las n ejecuciones.
- **Scr**. Algoritmo de *scoring* de Mihelic y Robic.
- **Scr (A)**. Algoritmo de *scoring* de Mihelic y Robic utilizando la variante (A).
- **SS**. Algoritmo de Pacheco y Casado (*Scatter Search*).
- **TS-2**. Algoritmo de Mladenovic (*Tabu Search* con dos listas tabú).
- **Das**. Algoritmo de Daskin.
- **AP**. *algoritmo propuesto* sin amplificar; es decir, se ejecuta sólo una vez.
- **AP (A)**. *algoritmo propuesto* sin amplificar y utilizando la variante (A).
- **AP (A-I)**. *algoritmo propuesto* sin amplificar y utilizando la variante (A) seguida de la variante (I).
- **AP (n)**. *algoritmo propuesto* amplificado n veces.
- **AP (A) (n)**. *algoritmo propuesto* amplificado n veces, aplicando la variante (A) sobre cada ejecución.
- **AP (A-I) (n)**. *algoritmo propuesto* amplificado n veces, aplicando la variante (A), seguida de la variante (I), sobre cada ejecución.

[1] La variante (A) fue definida en la Sección 4.3.3 y consiste en resolver el Problema de Selección de 1 Centro sobre cada conjunto de una partición.

[2] La variante (I) fue definida en la Sección 4.3.3 anterior y consiste en realizar el mejor cambio entre un nodo dentro de la solución y un nodo fuera de ella.

- **AP (n^2).** *algoritmo propuesto* amplificado n^2 veces.
- **AP (A) (n^2).** *algoritmo propuesto* amplificado n^2 veces, aplicando la variante (A) sobre cada ejecución.
- **AP (A-I) (n^2).** *algoritmo propuesto* amplificado n^2 veces, aplicando la variante (A), seguida de la variante (I), sobre cada ejecución.
- **APM (A) (n^2) (nM).** Algoritmo 4.4 repetido n veces, donde cada repetición consta del *algoritmo propuesto con memoria* amplificado n^2 veces aplicando la variante (A).
- **APM (A-I) (n^2) (sc).** Algoritmo 4.4 repetido hasta que deja de haber mejora en la solución (sc = sin cambio), donde cada repetición consta del *algoritmo propuesto con memoria* amplificado n^2 veces aplicando la variante (A) seguida de la variante (I).
- **APM (A-I) (n^2) (10M).** Algoritmo 4.4 repetido 10 veces, donde cada repetición consta del *algoritmo propuesto con memoria* amplificado n^2 veces aplicando la variante (A) seguida de la variante (I).

El *algoritmo propuesto* (AP), junto con todas sus variante, fue ejecutado con un valor de $\alpha = n$. Además, el valor $P(v)$ que se asignó al nodo $v \in V$ más alejado en cada iteración i fue de $\frac{1}{k - \sqrt[k-1]{\alpha - i}}$ (recuérdese que la única condición requerida es que $P(v)$ sea mayor o igual a $\frac{1}{k - \sqrt[k-1]{\alpha}}$ y, dado que $n > k$ y $\alpha = n$, esta condición se cumple).

En la Tablas 5.1 y 5.2 se muestran los resultados obtenidos al ejecutar cada uno de los algoritmos descritos. El promedio del factor de aproximacion, así como la deviación estándar, se aprecian en las Tablas 5.3 y 5.4. Las Figuras 5.1 y 5.2 muestran gráficamente la eficacia de cada algoritmo.

5.2. Discusión

Los algoritmos del estado del arte que tienden a encontrar las mejores soluciones conocidas son:

- Das (algoritmo de Daskin).
- SS (*Scatter Search* o Búsqueda Dispersa).
- TS-2 (*Tabu Search* o Búsqueda Tabú con dos listas).

Si bien estos algoritmos tienden a ser eficaces, no existe ninguna garantía teórica de que esta eficacia sea constante para instancias arbitrarias.

El *algoritmo propuesto* (con o sin memoria) tiende a superar en eficacia a la mayoría de los algoritmos del estado del arte, siendo el algoritmo Das la excepción, mientras que prácticamente iguala en eficacia al algoritmo Scr. Es importante señalar que el *algoritmo propuesto* garantiza teóricamente que la solución entregada es una 2-aproximación, para instancias arbitrarias, con una probabilidad de al menos 0.63.

La complejidad de la versión del *algoritmo propuesto* que iguala en eficacia al algoritmo Scr, el cual es mejor algoritmo polinomial del estado del arte con una complejidad de $O(n^3)$, es de $O(n^4)$. Incluso no más de $O(n^4)$ pasos fueron necesarios para encontrar las mejores soluciones conocidas en la mitad de las instancias de prueba.

Tabla 5.1: Resultados de algunos algoritmos del estado del arte ejecutados sobre los grafos *pmed* de la librería OR-Lib, utilizando las variantes (A) e (I)

grafo	n	k	Mejor solución conocida	Gon	Gon (A)	Gon (A-I)	Gon+	Gon (A) +	Gon (A-I) +	Scr	Scr (A)	Das	TS-2	SS
1	100	5	127	183	153	148	155	136	131	133	133	**127**	**127**	**127**
2	100	10	98	135	123	121	117	117	104	109	109	**98**	100	**98**
3	100	10	93	145	132	114	124	114	103	99	99	**93**	94	**93**
4	100	20	74	121	97	92	92	84	80	83	82	**74**	77	**74**
5	100	33	48	66	55	66	62	52	52	**48**	**48**	**48**	67	**48**
6	200	5	84	135	106	98	98	91	88	90	90	**84**	**84**	**84**
7	200	10	64	102	88	81	85	75	71	70	70	**64**	**64**	**64**
8	200	20	55	82	82	71	71	66	61	67	67	**55**	**55**	**55**
9	200	40	37	52	49	49	49	42	42	38	38	**37**	56	40
10	200	67	20	31	31	28	29	25	24	**20**	**20**	**20**	**20**	24
11	300	5	59	88	68	66	68	60	60	60	60	**59**	60	**59**
12	300	10	51	82	67	63	66	58	54	53	53	**51**	52	**51**
13	300	30	35	57	43	46	49	43	42	38	38	36	55	40
14	300	60	26	40	38	36	36	33	32	27	27	**26**	42	34
15	300	100	18	25	25	24	23	22	22	**18**	18	**18**	36	26
16	400	5	47	57	56	54	52	49	**47**	48	48	-	**47**	**47**
17	400	10	39	55	58	48	48	44	42	41	41	-	**39**	**39**
18	400	40	28	43	40	38	39	37	36	32	32	-	38	-
19	400	80	18	28	28	26	27	24	23	20	20	-	31	-
20	400	133	13	19	18	19	17	17	16	14	14	-	32	-
21	500	5	40	63	48	47	45	43	41	**40**	**40**	-	**40**	**40**
22	500	10	38	59	47	45	47	44	41	41	41	-	39	**38**
23	500	50	22	35	31	32	32	30	28	24	24	-	37	-
24	500	100	15	23	23	22	21	21	20	17	17	-	19	-
25	500	167	11	15	16	14	15	14	14	**11**	**11**	-	27	-
26	600	5	38	47	43	42	43	40	**38**	41	41	-	**38**	**38**
27	600	10	32	47	39	35	38	34	33	33	33	-	**32**	**32**
28	600	60	18	28	29	28	25	24	23	20	20	-	22	-
29	600	120	13	20	19	19	18	17	17	**13**	**13**	-	18	-
30	600	200	9	14	14	14	13	13	13	11	11	-	20	-
31	700	5	30	42	37	33	36	32	30	**30**	**30**	-	**30**	**30**
32	700	10	29	43	36	35	37	32	30	31	31	-	30	30
33	700	70	15	26	25	23	23	22	21	17	17	-	20	-
34	700	140	11	17	17	16	16	16	16	**11**	**11**	-	18	-
35	800	5	30	37	35	35	34	32	31	32	32	-	31	**30**
36	800	10	27	41	37	31	34	31	29	28	28	-	28	28
37	800	80	15	24	22	22	23	21	20	16	16	-	19	-
38	900	5	29	39	31	30	31	**29**	**29**	**29**	**29**	-	**29**	**29**
39	900	10	23	33	29	28	28	25	25	24	24	-	24	24
40	900	90	13	20	21	20	19	19	18	14	14	-	17	-
Comp.				$O(kn)$	$O(n^2)$	$O(n^2)$	$O(kn^2)$	$O(n^3)$	$O(n^3)$	$O(n^3)$	$O(n^3)$	-	-	-

Tabla 5.2: Resultados del *algoritmo propuesto* ejecutado sobre los grafos *pmed* de la librería OR-Lib, utilizando las variantes (A) e (I)

grafo	n	k	Mejor solución conocida	AP	AP (A)	AP (A-I)	AP (n)	AP (A) (n)	AP (A-I) (n)	AP (n^2)	AP (A) (n^2)	AP (A-I) (n^2)	APM (A-I) (n^2) (sc)	APM (A-I) (n^2) (10M)
1	100	5	127	181	150	133	140	129	**127**	132	**127**	**127**	**127**	**127**
2	100	10	98	123	118	125	117	117	102	108	102	101	**98**	**98**
3	100	10	93	140	133	115	113	101	102	104	96	95	94	94
4	100	20	74	109	100	97	92	84	79	82	79	78	**74**	**74**
5	100	33	48	73	69	66	59	53	49	52	**48**	**48**	**48**	**48**
6	200	5	84	108	104	91	94	92	88	91	85	**84**	**84**	**84**
7	200	10	64	91	89	80	78	73	71	73	68	67	**64**	**64**
8	200	20	55	70	69	68	67	63	61	64	58	57	56	56
9	200	40	37	52	52	46	45	42	42	44	41	40	40	**37**
10	200	67	20	31	32	25	27	24	24	25	22	21	21	**20**
11	300	5	59	80	66	69	68	63	60	62	60	**59**	**59**	**59**
12	300	10	51	73	67	62	63	60	55	58	54	52	52	**51**
13	300	30	35	56	48	51	46	44	40	43	41	40	39	39
14	300	60	26	41	34	36	35	32	32	33	31	29	29	27
15	300	100	18	24	24	23	22	22	21	21	20	19	19	19
16	400	5	47	62	52	50	52	49	**47**	49	**47**	**47**	**47**	**47**
17	400	10	39	51	49	47	46	45	43	44	43	41	**39**	**39**
18	400	40	28	41	39	41	37	35	34	35	34	33	31	31
19	400	80	18	27	28	27	25	24	23	24	23	22	20	19
20	400	133	13	19	18	17	17	17	17	17	16	16	16	16
21	500	5	40	56	49	44	45	43	**40**	42	**40**	**40**	**40**	**40**
22	500	10	38	51	49	44	44	43	40	43	41	39	39	39
23	500	50	22	34	33	31	30	28	28	29	28	27	24	24
24	500	100	15	22	21	21	21	21	19	20	19	18	17	17
25	500	167	11	16	15	15	15	14	14	14	14	14	14	14
26	600	5	38	49	46	41	43	39	39	40	**38**	**38**	**38**	**38**
27	600	10	32	42	42	41	37	36	34	36	34	33	**32**	**32**
28	600	60	18	28	26	25	24	24	23	23	23	22	20	20
29	600	120	13	18	19	19	18	17	17	17	16	16	15	15
30	600	200	9	14	14	13	13	12	12	12	12	12	12	12
31	700	5	30	40	34	32	35	32	**30**	32	**30**	**30**	**30**	**30**
32	700	10	29	40	37	34	34	32	32	33	31	30	30	**29**
33	700	70	15	24	22	22	22	21	21	21	20	20	19	18
34	700	140	11	17	16	17	16	15	15	15	15	14	14	13
35	800	5	30	38	32	32	35	32	**30**	32	32	**30**	**30**	**30**
36	800	10	27	35	35	34	33	31	30	31	31	28	28	28
37	800	80	15	22	23	22	21	20	20	20	20	18	18	18
38	900	5	29	41	33	36	33	30	**29**	30	30	**29**	**29**	**29**
39	900	10	23	29	29	27	27	25	25	26	26	24	24	24
40	900	90	13	20	19	19	19	18	18	18	17	17	17	15
Comp.				$O(kn)$	$O(n^2)$	$O(n^2)$	$O(kn^2)$	$O(n^3)$	$O(n^3)$	$O(kn^3)$	$O(n^4)$	$O(n^4)$	-	$O(n^4)$

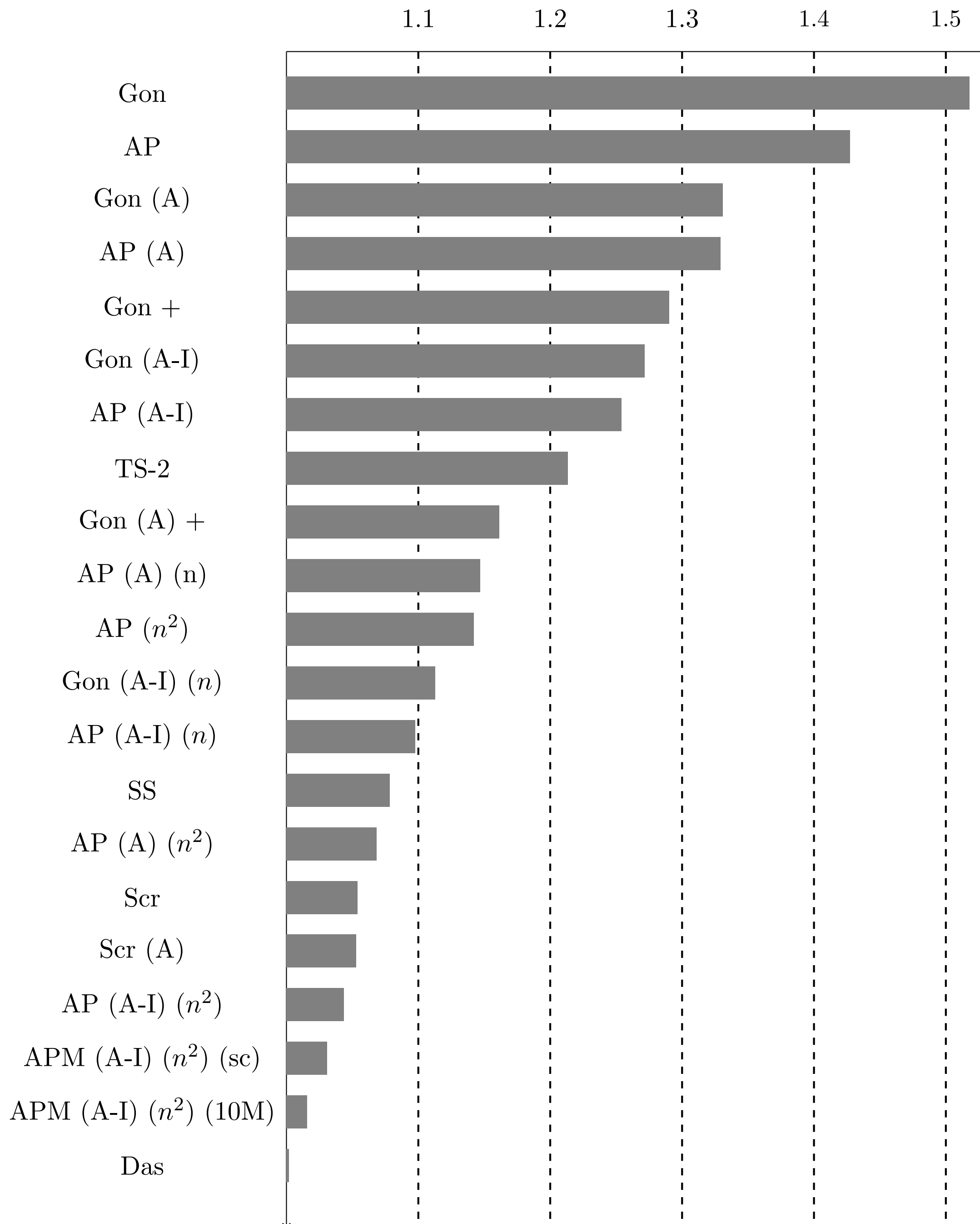


Figura 5.1: Comparación de la calidad de algunos algoritmos del estado del arte y del *algoritmo propuesto* (grafos 1-15)

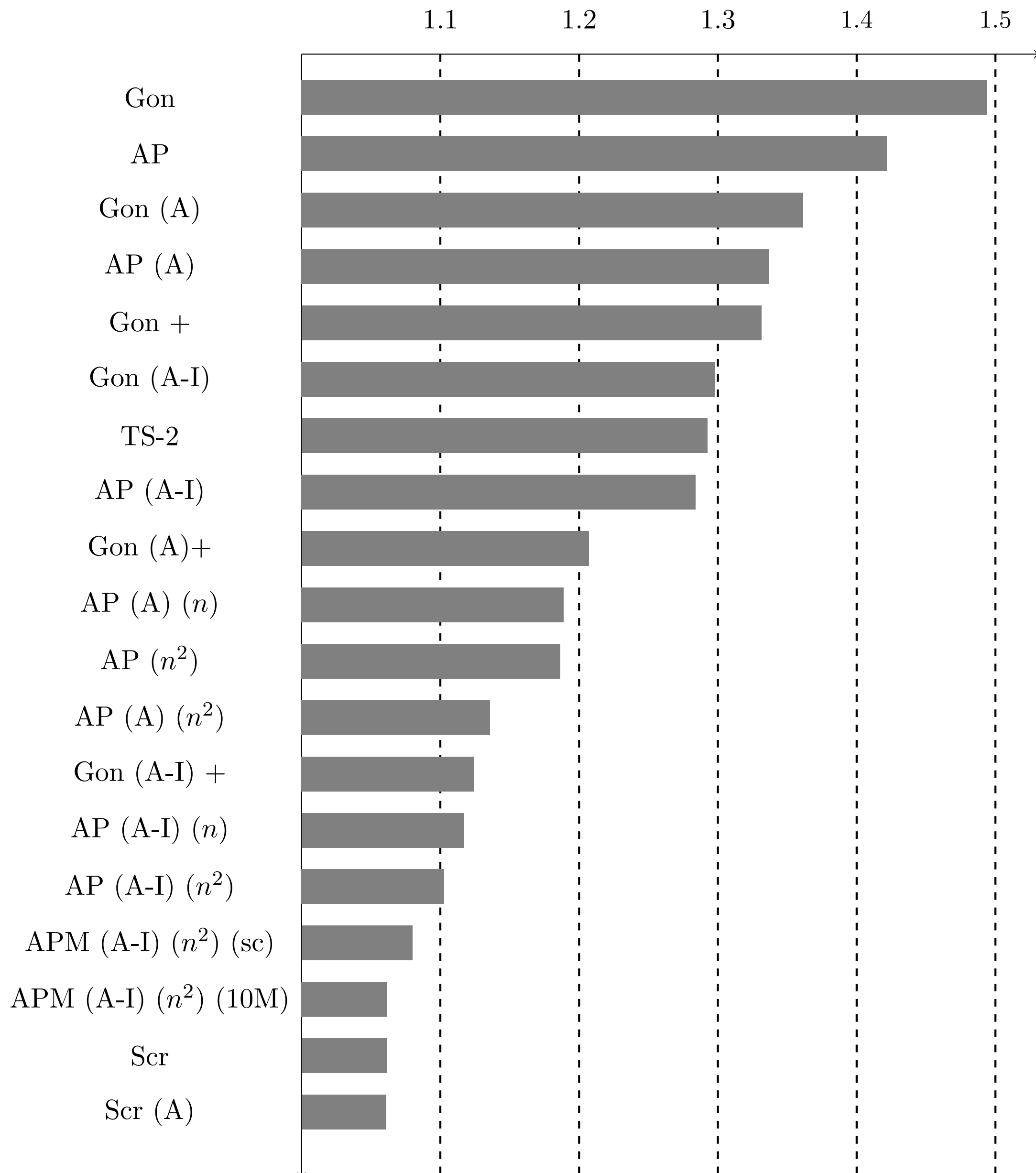


Figura 5.2: Comparación de la calidad de algunos algoritmos del estado del arte y del *algoritmo propuesto* (grafos 1 al 40)

Tabla 5.3: Factores de aproximación experimentales (grafos 1 al 40)

Algoritmo	Factor de aproximación	
	Promedio	Desviación
Gon	1.4937	0.1141
AP	1.4218	0.1057
Gon (A)	1.3615	0.1558
AP (A)	1.3370	0.1344
Gon+	1.3313	0.2330
Gon (A-I)	1.2978	0.1465
TS-2	1.2924	0.4090
AP (A-I)	1.2840	0.1423
Gon (A) +	1.2070	0.1373
AP (A) (n)	1.1889	0.1153
AP (n^2)	1.1865	0.1123
AP (A) (n^2)	1.1358	0.1172
Gon (A-I) +	1.1241	0.2231
AP (A-I) (n)	1.1171	0.2188
AP (A-I) (n^2)	1.1027	0.1103
APM (A-I) (n^2) (sc)	1.0799	0.1004
APM (A-I) (n^2) (10M)	1.0614	0.0818
Scr	1.0613	0.0507
Scr (A)	1.0609	0.0504

Tabla 5.4: Factores de aproximación experimentales (grafos 1 al 15)

Algoritmo	Factor de aproximación	
	Promedio	Desviación
Gon	1.5181	0.0910
AP	1.4275	0.11109
Gon (A)	1.3309	0.1233
AP (A)	1.3291	0.1206
Gon+	1.2901	0.0852
Gon (A-I)	1.2718	0.0845
AP (A-I)	1.2541	0.1063
TS-2	1.2136	0.3211
Gon (A) +	1.1615	0.0745
AP (A) (n)	1.1470	0.0669
AP (n^2)	1.1420	0.0695
Gon (A-I) +	1.1129	0.0702
AP (A-I) (n)	1.0975	0.0684
SS	1.0784	0.1380
AP (A) (n^2)	1.0685	0.0585
Scr	1.0539	0.0411
Scr (A)	1.0530	0.0396
AP (A-I) (n^2)	1.0435	0.0427
APM (A-I) (n^2) (sc)	1.0309	0.0422
APM (A-I) (n^2) (10M)	1.0158	0.0319
Das	1.0019	0.0073

Capítulo 6

Conclusiones

1. Los algoritmos del estado del arte diseñados para resolver el problema de selección de k-centros pueden ser clasificados dentro de alguna de las siguientes clases:

    - Algoritmos de aproximación.
    - Heurísticas o metaheurísticas.

    Los algoritmos de aproximación se ejecutan en tiempo polinomial y entregan una 2-aproximación; además, garantizan teóricamente que no es posible encontrar mejores soluciones en tiempo polinomial, a menos que $P = NP$. Sin embargo, las soluciones que entregan pueden resultar poco satisfactorias en la práctica, sobre todo cuando lo que se desea es encontrar la solución óptima. Por otro lado, las heurísticas y metaheurísticas no garantizan teóricamente que su tiempo de ejecución sea eficiente ni que la solución entregada sea cercana a la óptima; sin embargo, para la mayoría de las instancias tienden a ejecutarse rápidamente e incluso encuentran las mejores soluciones conocidas.

2. El *algoritmo propuesto* cuenta con las ventajas de los algoritmos del estado del arte:

    - Se ejecuta en tiempo polinomial con base en el tamaño de la entrada y un parámetro $\alpha \in \mathbb{Z}^+$ definido por el usuario.
    - Para la mayoría de las instancias encuentra las mejores soluciones conocidas e incluso existen instancias para las cuales supera a los mejores algoritmos del estado del arte.
    - Es importante resaltar la importancia del diseño de algoritmos exactos polinomiales para problemas *NP-Difíciles*, pues tal resultado implicaría que $P = NP$. El *algoritmo propuesto* no es un algoritmo exacto, sin embargo tiende a encontrar las mejores soluciones conocidas en tiempo polinomial.

3. El *algoritmo propuesto* fue diseñado siguiendo una estrategia de diseño que presenta ciertas similitudes con otras técnicas del estado del arte, tales como los algoritmos EDAs y la técnica GRASP; sin embargo, la técnica utilizada no parece corresponder de manera precisa a ninguna de ellas. Esta técnica de diseño, aplicada a un problema Π, requiere básicamente del siguiente par de elementos:

    - Un algoritmo de aproximación determinista para el problema Π que entregue una ρ-aproximación.

- Una función de masa de probabilidad que garantice que el algoritmo resultante amplificado entregará una ρ-aproximación con una probabilidad mayor o igual a una constante.

El *algoritmo propuesto* tiene como base al algoritmo de González y la función de masa de probabilidad que utiliza es construida en función de una variable $\alpha \in \mathbb{Z}^+$ introducida por el usuario. Como se demostró en el Teorema 4.3.1, el *algoritmo propuesto* amplificado α veces entrega una 2-aproximación con una probabilidad de al menos 0.63:

$$P[\acute{e}xito]_\alpha \geq 1 - \left(1 - \frac{1}{\alpha}\right)^\alpha \geq 1 - \frac{1}{e} \approx 0.63 \tag{6.1}$$

La demostración del Teorema 4.3.1 hace uso de la regla de la multiplicación (Ecuación 6.3), pues la probabilidad del evento "*seleccionar como centro un nodo que forme parte de una 2-aproximación, donde los centros seleccionados previamente forman parte de esa misma 2-aproximación*" es conocida y corresponde a la probabilidad asignada al nodo más alejado en cada iteración del algoritmo, donde la función de masa de probabilidad utilizada, denominada función *Caracterizable*, es la siguiente:

$$P(x) = \begin{cases} dist(v_1, C) \cdot y_i & , \quad x = v_1 \\ dist(v_2, C) \cdot y_i & , \quad x = v_2 \\ \vdots & \\ \geq \frac{1}{k - \sqrt[1]{\alpha}} & , \quad x = v_j \\ \vdots & \\ dist(v_{n-i}, C) \cdot y_i & , \quad x = v_{n-i} \end{cases} \tag{6.2}$$

El nodo v_j representa al nodo más alejado de la solución parcial construida en la iteración anterior y y_i es un factor de normalización. La función de masa de probabilidad 6.2 es definida en cada iteración del algoritmo, donde en cada una de ellas se selecciona uno de los k centros que conformarán la solución.

$$P[\cap_{i=1}^k E_i] = P[E_1] \cdot P[E_2|E_1] \cdot P[E_3|E_1 \cap E_2] \cdots P[E_k| \cap_{i=1}^{k-1} E_i] \tag{6.3}$$

El hecho de que el uso de la función de masa de probabilidad 6.2 permita obtener el resultado de la desigualdad 6.1 se debe a que la función 6.2 es construida de manera tal que así sucediera. Es decir, la función de masa de probabilidad 6.2 es construida de tal manera que la probabilidad asignada al evento "*seleccionar como centro un nodo que forme parte de una 2-aproximación, donde los centros seleccionados previamente forman parte de esa misma 2-aproximación*" fuera mayor o igual a la mínima necesaria para garantizar el resultado de la desigualdad 6.1. Nótese que el nodo más alejado v_j, de cada solución parcial construida en la iteración anterior, representa justamente a este evento, cuya probabilidad es mayor o igual a $\frac{1}{k - \sqrt[1]{\alpha}}$.

4. Es interesante observar que, al aplicar las variantes (A) y (A-I) sobre el algoritmo de González+ (Gon+), se logra incrementar considerablemente su eficacia, llegando incluso a encontrar las mejores soluciones conocidas, con la ventaja de que al ejecutar el algoritmo de González se cuenta con la garantía de que las soluciones entregadas son una 2-aproximación.

5. El *algoritmo propuesto* depende fuertemente del concepto de *amplificación*, el cual le permite incrementar su probabilidad de *éxito*, es decir, su probabilidad de encontrar soluciones cercanas a la óptima. Si bien la complejidad del *algoritmo propuesto* depende del valor α introducido por el usuario, se demostró experimentalmente que un valor de α igual a n permite encontrar las mejores soluciones conocidas para la mayoría de las instancias de prueba.

6. Se desarrolló una variante del *algoritmo propuesto* denominada *algoritmo propuesto con memoria*. Ésta variante consiste en realizar la búsqueda de soluciones con base en la mejor solución obtenida previamente, de tal manera que se observa un mayor progreso del valor de la función objetivo en menor tiempo.

7. Se demostró experimentalmente que no más de $O(n^4)$ pasos del *algoritmo propuesto* son necesarios para igualar en eficacia a los algoritmos polinomiales del estado del arte, donde el mejor algoritmo polinomial del estado del arte tiene una complejidad de $O(n^3)$. De hecho, en la mitad de las instancias de prueba el algoritmo propuesto encontró las mejores soluciones conocidas, las cuales son entregadas normalmente por las mejores metaheurísticas del estado del arte.

8. Como queda demostrado, a través del diseño del *algoritmo propuesto*, la técnica propuesta puede ser una herramienta eficiente y eficaz para la resolución de problemas de *Optimización*.

Trabajo futuro

1. Estudiar y/o proponer nuevas funciones de masa que incrementen la eficiencia y eficacia del *algoritmo propuesto*.

2. Estudiar versiones particulares del problema de selección de k-centros (por ejemplo, la versión con capacidades) y diseñar algoritmos para éstas.

3. Analizar y caracterizar la forma en que el esquema de búsqueda local se mueve dentro del espacio de soluciones.

4. Generalizar la técnica propuesta para otros problemas de optimización combinatoria. El esquema informal de esta generalización, aplicada a un problema Π, requeriría básicamente de:

    - Un algoritmo de aproximación determinista para el problema Π que entregue una ρ-aproximación.
    - Una función de masa de probabilidad que garantice que el algoritmo resultante amplificado entregará una ρ-aproximación con una probabilidad mayor o igual a una constante.

    La dificultad del diseño de algoritmos basados en esta técnica radica en la construcción de la función de masa de probabilidad, la cual debe ser construida de manera tal que la probabilidad de *éxito* del algoritmo amplificado sea mayor o igual a un valor constante para instancias arbitrarias.

Bibliografía

[1] T. Gonzalez, "Clustering to minimize the maximum inter-cluster distance", *Theoretical Computer Science*, Vol 38:293-306, (1985).

[2] O. Kariv, S. L. Hakimi, "An algorithmic approach to network location problems. I: the p-centers", *Journal of Applied Mathematics*, Vol. 37, No. 3, 1979.

[3] Vijay V. Vazirani, "Approximation Algorithms", *Springer*, 2001.

[4] D. Hochbaum and D. B. Shmoys, "A best possible heuristic for the k-center problem", *Mathematics of Operations Research*, Vol 10:180-184, (1985).

[5] Michael Sipser, "Introduction to the Theory of Computation", Second Edition, *Thomson*, 2006.

[6] T. Cormen, C. Leiserson and R. Rivest, "Introduction to Algorithms", *The MIT Press*, 1989.

[7] B. D. Acharya, "Domination in Hypergraphs", *AKCE Journal of Graphs and Combinatorics*, Vol 4 ,No. 2, 117-126, 2007.

[8] C. Berge, "Graphs and Hypergraphs", North-Holland, Amsterdan, 1989.

[9] T. Feder and D. Greene, "Optimal algorithms for approximate clustering", *Proc. of the* 20th *ACM Symposium on the Theory of Computing*, pages 434-444, (1988).

[10] Zvi Drezner, Horst W. Hamacher, "Facility Location: Applications and Theory", *Springer*, Berlin, 2001.

[11] M. R. Garey and D.S. Johnson, "Computers and Intractability: A guide to the theory of NP-completeness", *Freeman, San Francisco* (1978).

[12] Dimitri P. Bertsekas, John N. Tsitsiklis, "Introduction to Probability", *Athena Scientific* (2000).

[13] Rattan Rana, Deepak Garg, "The Analytical Study of K-Center Problem Solving Techniques", Internationañ Journal of Information Technology and Knowledge Management, Julio-Diciembte 2008, Volumen 1, No. 2, pp. 527-535.

[14] Al-khedhairi A., Salhi S., "Enhancements to two exact algorithms for solving the vertex P-center Problem", Journal Math. Model. Algorithms, Volumen 4, No. 2, 129-147, 2005.

[15] Jurij Mihelic, Borut Robic, "Approximation Algorithms for the k-centre Problem: an experimental evaluation", *Proc. OR 2002*, Klagenfurt, Austria, 2002.

[16] Jurij Mihelic, Borut Robic, "Solving the k-center Problem Efficiently with a Dominating Set Algorithm", *Journal of Computing and Information Technology*, CIT 13(3): 225-234 (2005)

[17] Jon Kleinberg, Éva Tardos, "Algorithm Design", *Pearson, Addison Wesley*, Cornell University, 2005.

[18] Mauricio G. C. Resende, Celso C. Ribeiro, "Greedy Randomized Adaptive Search Procedures", *Optimization Online* `http://www.optimization-online.org/DB_HTML/2001/09/371.html`.

[19] Dorit S. Hochbaum, "Approximation Algorithms for NP-Hard problems", *PWS Publishing Company*, Vol 19:1363-1373, (1971).

[20] Rajani, Rattan Rana, Anil Kapil, "Comparative Study of Heuristics Approaches for K-Center Problem".

[21] M. E. Dyer, A. M. Frieze, "A simple heuristic for the p-centre problem", *Operations Research Letters*, Volume 3, Number 6, 285-288, (1984).

[22] Doron Chen, Reuven Chen, "New relaxation-based algorithms for the optimal solution of the continuous and discrete p-center problems", Computers & Operations Research, 36 (2009) 1646-1655.

[23] Reza Zanjirani Farahani, Masoud Hekmatfar, "Facility Location: Concepts, Models, Algorithms and Case Studies", *Physica Verlag Heidelberg*, Germany, 2009.

[24] J.E.Beasley, "OR-Library: distributing test problems by electronic mail", Journal of the Operational Research Society 41(11) (1990) pp1069-1072.

[25] J.E.Beasley, OR-Lib, `http://people.brunel.ac.uk/~mastjjb/jeb/orlib/pmedinfo.html`

[26] Joaquín A. Pacheco, Silvia Casado, "Solving two location models with few facilities by using a hybrid heuristic: a real health resources case", *Computers and Operations Research*, Volume 32, Issue 12 (2005) 3075-3091.

[27] Sourour Elloumi, Martine Labbé, Yves Pochet, "New Formulation and Resolution Method for the p-Vertex Problem", INFORMS Journal of Computing 2004, 16 : 84-94.

[28] Ariana Cruz Trejo, "Algoritmos de aproximación distribuidos para la diseminación de información en MANETs 3D", Tesis de Maestría, Instituto Politécnico Nacional, Centro de Investigación en Computación, 2012.

[29] Nenad Mladenović, Martine Labbé, Pierre Hansen, "Solving the p-Center Problem with Tabu Search and Variable Neighborhood Search", Networks, 42:48-64.

[30] Michael Mitzenmacher, Eli Upfal, "Probability and Computing: Randomized Algorithms and Probabilistic Analysis", *Cambridge University Press*, (2005).

[31] N. Pizzolato y H. Silva, "The location of public schools: Evaluations of practical experiences", *International Transactions In Operational Research*, vol. 4, no. 1, pp. 13-22, 1997.

[32] Hongzhong Jia, Fernando Ordóñez, Maged Dessouky, "A modeling framework for facility location of medical services for large-scale emergencies", *HE Transactions*, 39 (2007), 41-55.

[33] Saul I. Gass, "Linear Programming: Methods and Applications", *Mc Graw-Hill*, 1975.

[34] D. B. Shmoys, "Computing near-optimal solutions to combinatorial optimization problems", Technical Report, Ithaca, NY 14853, 1995, `http://people.orie.cornell.edu/shmoys/dimacs.ps`.

[35] C. Toregas, R. Swain, C. Revelle and L. Bergman, "The location of emergency service facilities", *Operations Research*, Vol 19:1363-1373, (1971).

[36] Al-khedhairi A., "Enhancement of Daskin's Algorithm for Solving P-centre Problem", *Journal of Approximation Theory and Applications*, 2(2), 121-134, (2007).

[37] Daskin M., "A new approach to solve the vertex P-center problem to optimality: algorithm and computational results", *Communications of the Operations Research Society of Japan*, 45(9), 428-436, (2000).

[38] Taylan Ilhan, Mustafa C. Pinar, "An Efficient Exact Algorithm for the Vertex p-Center Problem", *http://www.optimization-online.org/DB_HTML/2001/09/376.html*

[39] M. E. Dyer, "On a multidimensional search technique and its application to the Euclidian one-centre problem", Mathematics Department, Teesside Polytechnic, 1984.

[40] Rajeev Motwani, Pradhakar Raghavan, "Randomized Algorithms", *Cambridge University Press*, 1995.

[41] D.R. Karger and C. Stein. "A new approach to the min-cut problem", J. ACM, 43(4):601–640, 1996.

[42] F. Glover, M. Laguna, R. Martí, "Fundamentals of Scatter Search and Path Relinking", *Control and Cybernetics*, Volume 29, Number 3, 2000.

[43] M. Daskin, SITATION Software, `http://sitemaker.umich.edu/msdaskin/software`

[44] Anany Levitin, "Introduction to the design and analysis of algorithms", *Addison Wesley*, 2003.

[45] G. Polya, "How to solve it", *Princeton University Press*, Second edition, 1971.

[46] Ken McAloon, Carol Tretkoff, "Optimization and Computational Logic", *Wiley-Interscience*, 1996.

[47] Steven S. Skiena, "The Algorithm Design Manual", *Springer*, Second edition, 2008.

[48] Fred Glover, Gary A. Kochenberger, "Handbook of metaheuristics", *Kluwer's International Series*, 2003.

[49] Pedro Larrañaga, José A. Lozano, "Estimation of Distribution Algorithm", *Kluwer Academic Publishers*, 2002.

[50] A. E. Eiben, J. E. Smith, "Introduction to Evolutionary Computing", *Springer*, 2007.

[51] J. J. Grefenstette, "Optimization of control parameters for genetic algorithms", *IEEE Transactions on Systems, Man and Cybernetics*, Volume 16, Number 1, 1986.

[52] Clay Mathematics Institute, Millenium Problems, `http://www.claymath.org/millennium/`

[53] Satish Shirali, Harkrishan L. Vasudeva, "Metric Spaces", *Springer-Verlag*, 2006.

[54] Hamdy A. Taha, "Investigación de Operaciones", *Alfaomega*, Segunda edición, 1991.