Universidad Nacional Autónoma de México

Facultad de ciencias

Optimización por enjambre de partículas aplicado al problema del árbol de Steiner

T E S I S

que para obtener el título de:

LICENCIADA EN CIENCIAS DE LA COMPUTACIÓN

presenta:

Marisol Amézcua López

Director de tesis:
Dr. Canek Peláez Valdés

2023

Índice general

Introducción

El problema del árbol de Steiner (*STP*, *Steiner Tree Problem* por sus siglas en inglés) es un problema de optimización combinatoria que también pertenece al área de geometría computacional. Éste consiste en que dado un conjunto $\mathcal{P}$ de puntos en el plano euclidiano, se debe encontrar un conjunto $\mathcal{S}$ de puntos Steiner (definimos un punto de Steiner como un punto en el plano euclidiano el cual optimiza el peso total del árbol obtenido a partir del conjunto $\mathcal{P}$), tales que el Árbol de Peso Mínimo de $\mathcal{P} \cup \mathcal{S}$ sea óptimo, es decir, que la suma del peso de todas las aristas sea la mínima, en la Figura 1 se muestra un ejemplar del problema, en el cual el árbol rosa corresponde al de peso mínimo euclideano de los puntos originales, éste cuenta con un peso de 37. Por otro lado, el árbol morado conecta a los anteriores más un punto Steiner y cuenta con un peso de 35.80, lo cual corresponde a una mejora del 3.22 % sobre el peso original. El problema fue introducido por M. R. Garey y D. S. Johnson [22], el cual plantea una variante del problema de optimización y demostraron que es *NP*-completo, en otras palabras, no es posible resolverlo utilizando algoritmos convencionales. Actualmente el problema cuenta con diversas variantes y aplicaciones, entre estos podríamos destacar las siguientes:

- Problemas de diseño de redes: Se han utilizado en el diseño físico de la integración a escala muy grande de circuitos (*VLSI design*), donde el uso principal es para ayudar a resolver problemas de enrutamiento [7].
- Construcción de árboles de distribución multidifusión (*multicast*) [66].
- Árboles filogenéticos: En el campo de la biología o lingüística un árbol filogenético es un diagrama que representa las relaciones evolutivas dentro de un conjunto de taxones. Dentro de este campo existe el método de máxima parsimonia, el cual busca encontrar un árbol filogenético con el menor número de cambios evolutivos. Este problema se ha demostrado que es equivalente al problema del árbol de Steiner en hipercubos y se conoce como el problema del árbol de Steiner en filogenia (*SPP, Steiner Tree in phylogeny*) [21].

Dentro de la teoría de complejidad computacional se encuentran diversas clases de complejidad (las cuales se verán con más detalle en el Capítulo 2), entre las cuales están:

- *P*: Es la clase en la que se encuentran los problemas tales que se pueden resolver en tiempo polinomial por una máquina de Turing determinística.
- *NP*: Es la clase en la que se encuentran los problemas de decisión que pueden ser resueltos por una máquina de Turing no determinística.
- *NP*-duro: Son un conjunto de problemas de decisión que son como mínimo tan difíciles como un problema en *NP*. De una manera formal, son los problemas H tales que para cualquier problema $L \in NP$, L puede ser reducido en tiempo polinomial a H.

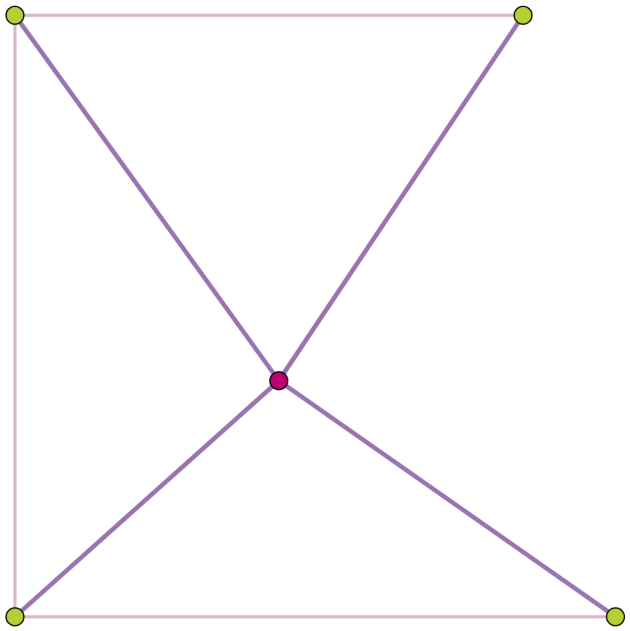


Figura 1: Ejemplo de un árbol de Steiner

- *NP*-completo: Estos problemas se encuentran en la intersección de los problemas *NP* y los problemas *NP*-duro, estos son los problemas más duros de la clase *NP* [23].

Para poder aproximarse a una solución de los problemas de la clase *NP*-duro existen diversos enfoques, entre estos destacan:

- Cómputo evolutivo: Se puede considerar un conjunto de algoritmos que buscan la optimización global. Estos se encuentran inspirados en la evolución biológica y se consideran un subcampo de la inteligencia artificial [10].
- Heurísticas de optimización combinatoria: Son una colección de procedimientos para resolver problemas de optimización bien definidos mediante una aproximación intuitiva en la que la estructura del problema se utiliza para obtener una buena solución [20].
- Investigación de operaciones: Es una disciplina que se preocupa por decidir científicamente cómo diseñar y operar mejor los sistemas hombre-máquina, por lo general bajo condiciones que requieren la asignación de recursos escasos [18].
- Algoritmos de aproximación: Algoritmos en los cuales se garantiza una rápida ejecución (con respecto al tamaño de entrada) y la obtención de una solución que se encuentra cuantificablemente cerca del valor óptimo [47].

El problema del árbol de Steiner se considera un problema de la clase de complejidad *NP*-duro, debido a esto los algoritmos convencionales no lo pueden resolver en un tiempo humanamente razonable. Por lo tanto, se propone utilizar una heurística para aproximarse a lo que sería una solución deseable.

De entre varias heurísticas, para este problema se plantea utilizar Optimización por Enjambre de Partículas (*PSO*, *Particle Swarm Optimization*), la cual fue propuesta originalmente por James

Kennedy en 1995 [44] como un modelo de simulación de organismos sociales, como son bancos de peces o parvadas de aves; la heurística consiste en un conjunto de entidades simples llamadas partículas, éstas se desplazan de forma iterativa sobre un cierto espacio de búsqueda para alguna función. Las partículas evalúan su aptitud (usualmente conocido como *fitness*) utilizando la ubicación en la que se encuentran; posteriormente, se emplea la aptitud actual, la mejor aptitud del enjambre (coeficiente social) y la mejor aptitud de la partícula durante la ejecución (coeficiente individual), además de perturbaciones aleatorias para determinar la posición de cada partícula en la siguiente iteración [69].

Esta heurística usualmente se utiliza en problemas en los que existe más de un óptimo y cuenta con diversas variaciones como son modelos de hormigas, algas, leones, abejas, virus, etcétera. Actualmente se ha utilizado para diversas aplicaciones, como por ejemplo:

- Algoritmos de programación de tareas basados en enjambre de partículas para computación en malla (*Grid Computing*) [79].
- Un algoritmo basado en enjambre de partículas para la segmentación de imágenes [6].
- Uso de enjambre de partículas para el entrenamiento de un sistema de inferencia difusa basado en una red adaptativa (*ANFIS*)[74].
- Un algoritmo híbrido para resolver el problema de enrutamiento de vehículos [56].
- Un método para el descubrimiento informado de recursos en un sistema de computación en malla (*Grid Computing*) utilizando algoritmo de enjambre de partículas [1].

Este trabajo busca implementar la heurística de Optimización por Enjambre de Partículas para tratar de encontrar soluciones aceptables al problema del árbol de Steiner así como mostrar los resultados obtenidos en el proceso. Con este fin se explicará a detalle en el Capítulo 1 el problema a resolver así como la historia de este, en el Capítulo 2 se profundizará más en el concepto de complejidad computacional así como en el concepto de problemas de optimización. En el Capítulo 3 se explicará a fondo la heurística a utilizar, sus modificaciones y optimizaciones, posteriormente se describirá la implementación de la heurística y el problema en el lenguaje de programación Python en el Capítulo 4; y la experimentación realizada con el programa en el Capítulo 5. Finalmente, en la sección de conclusiones se explicarán los resultados obtenidos y las conclusiones del trabajo.

Capítulo 1

El problema del árbol de Steiner

En las matemáticas y ciencias de la computación podemos encontrar el problema del árbol de Steiner (*Steiner Tree Problem*), cuyos orígenes datan desde 1811 como una generalización del problema de Fermat-Torricelli, el cual enuncia lo siguiente:

Definición 1.1 (Problema de Fermat-Torricelli) *Dados tres puntos x_1, x_2 y x_3 (no colineales), encontrar un punto p tal que la suma de los segmentos $p\bar{x}_1 + p\bar{x}_2 + p\bar{x}_3$ sea mínima [19].*

Un ejemplo del problema de Fermat-Torricelli se puede visualizar en la Figura 1.1.

El problema del árbol de Steiner originalmente fue propuesto por el matemático francés Pierre de Fermat al físico italiano Evangelista Torricelli [54].

Actualmente existen distintas variantes del problema del árbol de Steiner, entre las que están:

- Árbol de Steiner euclideano mínimo (*ESTP*[1]), también conocido como el problema del árbol de Steiner geométrico, en cual anuncia lo siguiente:

  Definición 1.2 (ESTP) *Dado $\mathcal{P}$ un conjunto de puntos en el plano euclideano y* EMST[2] *una función tal que $EMST(\mathcal{X})$ encuentra el árbol euclideano de peso mínimo del conjunto de puntos $\mathcal{X}$, el problema* ESTP *consiste en encontrar un conjunto de puntos $\mathcal{S}$ tal que $\mathcal{S} \neq \mathcal{P}$ y el árbol euclideano de $\mathcal{S} \cup \mathcal{P}$ sea mínimo y $EMST(\mathcal{S} \cup \mathcal{P}) \leq EMST(\mathcal{P})$.*

  A los árboles resultantes se les llama árboles de Steiner y a los puntos pertenecientes al conjunto $\mathcal{S}$ se les llama puntos de Steiner.

[1] Siglas en inglés de *Euclidean Steiner Tree Problem*
[2] Siglas en inglés de *Euclidean Minimum Spanning Tree*

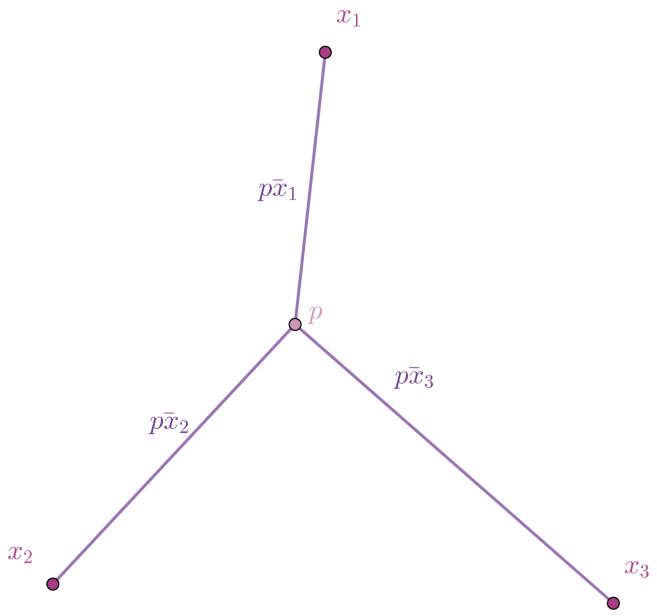


Figura 1.1: Ejemplo del problema de Fermat-Torricelli.

- Árbol de Steiner rectilíneo ($MRST$[3]): es una variante en la que la distancia euclideana es reemplazada por la distancia rectilínea o de Manhattan[4]. En términos matemáticos se enuncia de la siguiente manera:

  Definición 1.3 (MRST) *Sea $\mathcal{A}$ un conjunto de puntos en el plano, un árbol óptimo rectilíneo de Steiner es tal que conecta los puntos de $\mathcal{A}$ usando líneas horizontales y verticales de longitud mínima [22].*

  En 1977 los computólogos Michael Garey y David Johnson demostraron que este problema es NP-completo, un ejemplo de éste problema puede verse en la Figura 1.2, donde los puntos son conectados únicamente por líneas verticales y horizontales.

- Árbol de Steiner completo ($FSTT$[5]): esta variante enuncia lo siguiente:

  Definición 1.4 (FSTT) *Dada una gráfica $G = (V, E)$ con una función de peso $w : E \longrightarrow \mathbb{R}^2$ y un conjunto $R \subset V$, encontrar un árbol de Steiner completo de peso mínimo en G el cual es un tipo de árbol donde todos los vértices de R son hojas.*

  En 2003 los computólogos Chin Lung Lu, Chuan Yi Tang y Richard Chia-Tung Lee demostraron que el problema es NP-completo [53].

[3] Siglas en inglés de *Minimum Rectilinear Steiner Tree Problem*

[4] Dados dos puntos, la distancia rectilínea es la suma de las diferencias absolutas de las coordenadas cartesianas: $d(p, q) = \sum_{i=1}^{n} |p_i - q_i|$.

[5] Siglas en inglés de *Full Steiner Tree Problem*

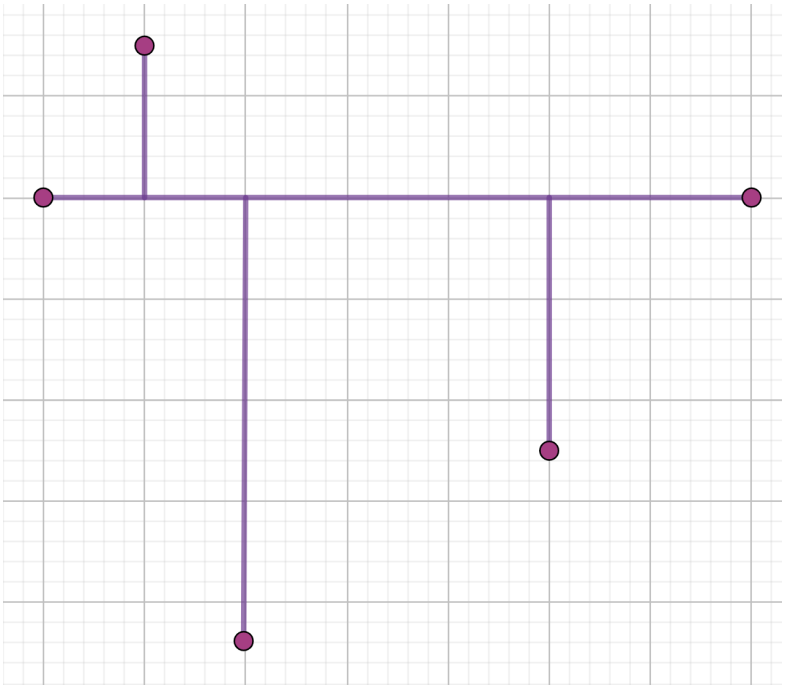


Figura 1.2: Ejemplar de un árbol rectilíneo.

- Árbol de clase Steiner ($CSTP$[6]): Es una generalización del problema del árbol de Steiner (euclideano) donde dada una gráfica con pesos en sus aristas y no dirigida, con los vértices particionados en g clases, encontrar un árbol de peso mínimo que incluya al menos un vértice de cada clase. Este problema se demostró ser NP-duro [35].

1.1. Contexto histórico

El origen del problema del árbol de Steiner está muy relacionado con el problema de Fermat-Torriceli, el cual fue introducido por Pierre de Fermat en 1643; éste se puede ver como el caso más simple no trivial del problema del árbol de Steiner. Consiste en que dados tres puntos en el plano, encontrar un cuarto punto tal que la suma de los tres segmentos que conectan a estos tres puntos con el cuarto, sea mínima.

La mención conocida más antigua del problema la hizo el matemático Joseph Diaz Gergonne en 1811, en el primer volumen de la revista *Annales de Gergonne*, donde se encontraba el problema de Fermat-Torriceli, incluyendo generalizaciones de este problema y una versión de lo que ahora se conoce como el problema del árbol de Steiner [54].

En 1819 un escritor anónimo con el pseudónimo Gallicus volvió a mencionar el problema en *The Mathematical Repository*, una revista de problemas matemáticos publicada en Inglaterra, donde se hace un análisis del trabajo de Gergonne. En 1836 Carl Friedrich Gauss y Heinrich Christian Schumacher discutieron por correspondencia una aparente paradoja relacionada con el problema de Fermat-Torricelli usando cuatro puntos[54].

[6]Siglas en inglés de *Class Steiner Tree Problem*

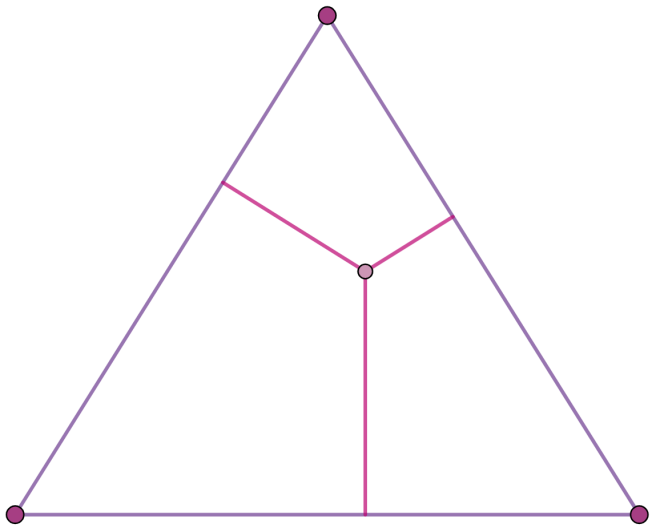


Figura 1.3: Visualización del teorema de Viviani.

En 1829 Karl Bopp retoma la aparente paradoja mencionada por Gauss y Schumacher y enumera todos los posibles sistemas de conexión para cuatro puntos dados. Esto lo hace considerando las rutas entre pares de puntos. Boop concluye que existen tres topologías distintas para cuatro puntos, cada una con dos puntos de ayuda (puntos Steiner) que pueden sobreponerse con otro punto. Dada una topología, Bopp considera la construcción de un árbol de Steiner completo para cada caso, muestra que si x_1 y x_2 pueden ubicarse en el plano de manera que cada una de sus tres aristas tenga un ángulo de 120° entre sí, entonces el árbol resultante es relativamente mínimo. La demostración es principalmente geométrica y es una aplicación del teorema de Viviani el cual enuncia que la suma de los tres segmentos que conectan al punto rosa claro con los tres lados del triángulo es constante, curiosamente Bopp atribuye este teorema a Jakob Steiner [54] (Un ejemplo del Teorema de Viviani puede apreciarse en la Figura 1.3, donde la suma de los segmentos rosados que conectan el punto del centro con los lados del triángulo es constante).

Bopp también considera el problema del árbol de Steiner de n puntos, prueba que el número de sistemas de conexión completos para n puntos es $a_n = 1 \times 3 \times 5 \times \ldots \times (2n-7) \times (2n-5)$. También demuestra por inducción que el número de aristas de estos sistemas es $s_n = 2n-3$; con estos datos Bopp bosqueja un algoritmo de tiempo finito para n puntos, e incluso presenta un ejemplo detallado para cinco puntos. Sin embargo, no aborda situaciones como degeneraciones (la construcción de árboles de Steiner no completos); ante esto Bopp señala que el número de casos diferentes es tan grande que una investigación general es difícilmente concebible.

En 1890 Eduard Hoffmann escribe un artículo con el título *On the shortest connection system for four points in the plane*[32] donde, como Bopp, principalmente ahonda en el caso donde los cuatro puntos forman un cuadrilátero convexo. Sin embargo, Hoffmann utiliza cálculo diferencial para demostrar que las aristas de los puntos Steiner se encuentran a 120°. Usando esto demuestra que si un árbol de Steiner completo con dos puntos Steiner existe, entonces éste será más corto que cualquier solución que tenga uno o cero puntos Steiner. En su artículo Hoffmann también comenta el problema con n puntos para el cual muestra que el sistema de conexión más corto es un árbol con a lo más $n-2$ puntos Steiner. También menciona que las aristas tienen las

mismas propiedades angulares que la instancia de cuatro puntos, y aunque no describe un análisis detallado, Hoffmann señala que la cantidad de gráficas posibles incrementa rápidamente a medida que lo hace el número de puntos iniciales.

En 1934, Vogtěch Jarník y Miloš Kössler tratan el problema no solo en el plano, sino también para espacios euclideanos de dimensiones superiores [37]. En la última sección del artículo, Jarník y Kössler construyeron árboles de Steiner mínimos para polígonos de n vértices donde $n = 2, 3, 4$ y 5, y mostraron que un árbol de Steiner mínimo para $n = 6$ o $n \geq 13$ coincide con el árbol de peso mínimo para el mismo conjunto de vértices (en otras palabras, no hay puntos Steiner).

Dentro del mismo artículo también derivaron algunas de las propiedades estructurales fundamentales de los arboles de Steiner mínimos. Entre estas propiedades, mostraron que el ángulo de cualquier vértice es al menos $2\pi/3$ y cada punto Steiner de la gráfica tiene grado 3; este resultado generaliza el resultado de Bopp para un espacio de 3 dimensiones. Después de probar que el problema en tres puntos tiene una solución única, mencionan

> Para $n > 3$ la situación es muy complicada, por lo tanto nos limitamos al caso donde los puntos base forman parte de los vértices de un polígono regular.

En 1941 el libro *What is Mathematics?: An Elementary Approach to Ideas and Methods* de Richard Courant y Herbert Robbins se discutió el problema. Dentro del libro se encuentra una sección titulada *"Steiner's Problem"*, el cual inicia con un resumen de el problema de Fermat-Torricelli con tres puntos. Los orígenes del problema son atribuidos a Jakob Steiner a inicios del siglo XIX, aunque sus contribuciones al problema de Fermat-Torricelli son mínimas [14]. Kupitz y Martini en 1997 [51] sostuvieron que Courant y Robbins leyeron sobre este problema en un pequeño artículo de Steiner de 1882, la cual era una publicación póstuma privada escrita por Steiner, sin referencias y varios errores. Esto se puede afirmar debido a que reprodujeron uno de estos errores.

El primer artículo publicado después del libro de Courant y Robbins que trata estos árboles es publicado por William Miehle [61] en 1958. Miehle propone un método mecánico para resolver el problema de Steiner usando cuerdas que pueden ser apretadas y clavijas móviles (para los puntos Steiner). Miehle no da un nombre al problema de minimización o a las redes mínimas, pero siguiendo a Courant y Robbins, acredita el trabajo inicial del ejemplar con tres puntos a Jakob Steiner.

En 1959 Beardwood, Halton y Hammersley publicaron el artículo *The shortest path through many points* [3]; donde muestran que la longitud del camino cerrado más corto para n puntos en una región del plano acotada v es casi siempre asintóticamente proporcional a $\sqrt{nv}$ donde $n \to \infty$. Los autores muestran que esta cota también aplica para el árbol de Steiner, donde llaman al problema *"Steiner's street network problem"*, haciendo referencia al libro de Courant y Robbin. Ésta parece ser la primera vez que el nombre de Steiner se usa formalmente como el nombre de un problema de redes. En 1961 Hammersley hizo un seguimiento de este trabajo donde se trató el problema con el mismo título, aunque el artículo se llama *"On Steiner's Network Problem"*.

En el mismo año Zdzislaw Melzak publicó un artículo titulado *"On the Problem of Steiner"* [59]. El artículo describe un algoritmo finito para generar un árbol de Steiner mínimo para cualquier conjunto de puntos. La idea es similar al método de Gergonne, pero muestra una mejor apreciación del problema. Melzak utiliza constantemente la terminología *"Steiner's problem"* en todo el artículo, pero se refiere a los árboles de peso mínimo como *"S-trees"* y a los puntos de Steiner como *"S-points"*. El conocimiento de este problema parece provenir de sus contactos dentro de

Bell Telephone Laboratories, ya que reconoce a Robert C. Prim en su artículo y aparentemente también conoció a Gilbert.

Seis años después, uno de los estudiantes de Melzak, Ernest Cockayne, escribió el artículo *"On the Steiner Problem"* [9] que mejora el algoritmo de Melzak y analiza el problema de Steiner usando otras métricas. Cockayne utiliza en este artículo la misma terminología de Melzak.

Unos años antes, en 1966 Maurice Hanan publicó un artículo influyente titulado *"On Steiner's Problem with Rectilinear Distance"* [29]. En este artículo se cubrieron la mayor parte de las propiedades básicas de los árboles de Steiner mínimos en la métrica rectilínea y predice correctamente la importancia de esta teoría para el área de *"printed circuit technology"* (diseño físico de microchips). Hanan cita a Melzak y usa su terminología en el artículo.

Otro artículo importante fue el de Cavalli-Sforza y Edwards publicado en 1967, titulado *"Phylogenetic analysis: Models and estimation procedures"* [5] donde se da una aplicación del problema de Steiner al análisis filogenético (construcción de árboles evolutivos). En el artículo no solo se utiliza la frase *"Steiner problem"*, si no que también se le llama a los árboles resultantes como *"Steiner Trees"*.

En 1968 Gilbert y Pollak publicaron un artículo titulado *"Steiner Minimal Trees"* [26], donde plantearon la siguiente conjetura:

Conjetura 1.1 *Sea P un conjunto de n puntos en el plano euclideano, $L_S(P)$ y $L_M(P)$ los pesos del árbol de Steiner y el árbol de peso mínimo de P, respectivamente. Entonces $L_S(P) \geq (\frac{\sqrt{3}}{2}) \times L_M(P)$.*

En 1990 se proveyó una prueba por Dingzhu Du y Frank Hwang [17], sin embargo en 2010 en el artículo *The Steiner Ratio Conjecture of Gilbert-Pollak May Still Be Open* [36], Innami, Kim, Mashiko y Shiohama ofrecen evidencia para descartar esta demostración, concluyendo que la conjetura de Gilbert y Pollak aún no ha sido demostrada. En el mismo artículo también se examinaron varias propiedades de los árboles de Steiner y se repasó la literatura existente. Este artículo fue importante debido a cómo da el marco geométrico que jugaría un papel clave en las siguientes décadas para el entendimiento de las propiedades de los árboles de Steiner mínimos. Este marco abrió camino a la construcción de algoritmos rápidos para "resolver" el problema de Steiner, entre ellos está el notorio algoritmo GeoSteiner [15] publicado en el año 2000. En el artículo se describen algoritmos exactos basados en la concatenación y generación de árboles de Steiner completos para el problema del árbol de Steiner euclideano y rectilíneo.

Años después en 1976 M. R. Garey y D. S. Johnson retomaron el problema utilizando el árbol de Steiner rectilíneo. El cual consiste en dado un conjunto $\mathcal{A}$ de puntos en el plano, un árbol rectilíneo óptimo es aquel que conecta a $\mathcal{A}$ usando líneas horizontales y verticales tal que la longitud total es la más corta. Este problema en el mismo artículo se demuesta que es *NP*-completo reduciendo el problema al de la cobertura exacta por tres conjuntos (*X3C, Exact Cover by 3-Sets*) [22]. En la Figura 1.4 se puede apreciar un ejemplar de árbol de Steiner rectilíneo, los puntos rosas corresponden al conjunto original de puntos, el punto morado es un punto Steiner. Las aristas moradas corresponden al árbol de peso mínimo sobre el conjunto original de puntos, los segmentos verdes corresponden al árbol de Steiner rectilíneo de peso mínimo ($MRST$).

En 2003 se publicó un artículo llamado "The full Steiner tree problem"[53] donde se demuestra que el problema del árbol de Steiner completo es *NP*-completo.

En 2005 Edmund Ihler publicó el artículo *"The complexity of approximating the class Steiner tree problem"* [35] donde obtiene que para cierta entrada de árboles sin puntos Steiner y aristas unita-

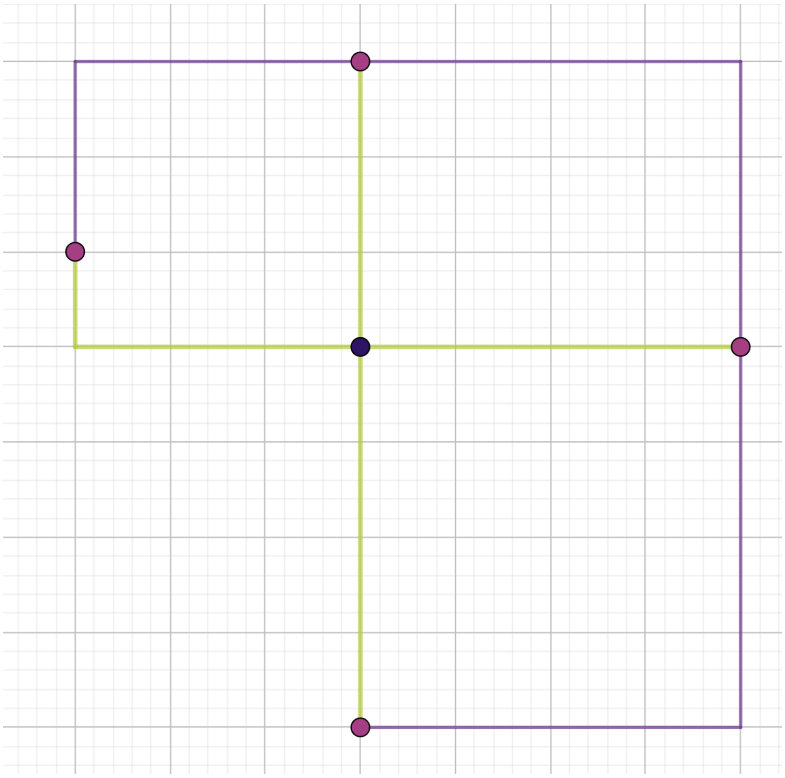


Figura 1.4: Ejemplo de un árbol de Steiner rectilíneo.

rias, el problema de árbol de clase Steiner (*Class Steiner Tree*) es tan difícil de aproximar como el problema de covertura mínima, para la cual tampoco se conoce una aproximación constante.

En el siguiente capítulo se ahondará más en el concepto de complejidad computacional y problemas de optimización.

Capítulo 2

Complejidad Computacional

Para comprender la necesidad de heurísticas en Ciencias de la Computación y por qué el problema del árbol de Steiner necesita usar una, es necesario primero comprender qué es la complejidad computacional y las clases de complejidad. Estos conceptos parten de la definición de computabilidad y para explicarlos se hará uso del modelo de la máquina de Turing.

2.1. Máquinas de Turing

Las máquinas de Turing (TM) son autómatas nombrados en honor de su creador, Alan Turing, inventadas en 1936. Las máquinas de Turing pueden computar cualquier función considerada computable, de hecho, es usual definir *computable* como computable por una máquina de Turing [48].

Fueron inventadas en los 30's mucho antes de que las computadoras reales fueran creadas para contestar una pregunta abierta en lógica matemática:

> Sean $\alpha_1, \alpha_2, .., \alpha_n$ predicados a los que llamaremos axiomas y θ otra fórmula a la que se nombrará como *candidato a teorema*. ¿θ es una consecuencia lógica de $\alpha_1, \alpha_2, .., \alpha_n$? (¿$\alpha_1, \alpha_2, .., \alpha_n \models \theta$?) [48]

A este problema se le conoce como el problema clásico de la decisión o *Entscheidungsproblem*, Church y Turing demostraron que no existe un método para resolver el *Entscheidungsproblem*, pero al resolverlo inventaron dos de los modelos más populares de computación, las máquinas de Turing y el cálculo λ [48].

Estos modelos llegaron en un momento en el que los matemáticos intentaban asimilar la noción de computable (*effective computation*). Se conocían varios algoritmos para calcular soluciones a problemas de forma eficiente, pero no estaban seguros de cómo definir si algo es computable o no. Varios modelos surgieron debido a esta necesidad, cada uno con sus propias características:

- Máquinas de Turing (creadas por Alan Turing).
- Máquina de Post (creado por Emil Post en 1936): Se trata de un autómata determinista que consiste en una cinta infinita en dos direcciones de espacios, fue publicada poco después de la máquina de Turing y se considera una variante de ésta, en la Figura 2.1 puede verse la

cinta infinita en ambas direcciones, cada casilla se encuentra marcada o vacía, y un cabezal o trabajador, opera cada casilla bajo un conjunto de reglas [49].

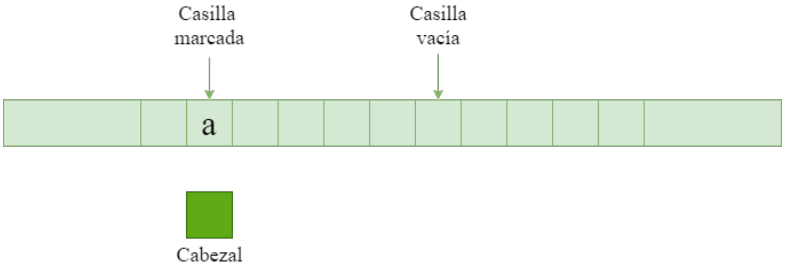


Figura 2.1: Ejemplo de una máquina de Post.

- Funciones recursivas-μ (introducidas por Karl Gödel y Jacques Herbrand en 1934[46]): Las funciones recursivas-μ son otro modelo de computabilidad. Estas son el mínimo conjunto cerrado bajo las operaciones de composición, recursión primitiva y minimización acotada [48]. Contiene a las siguientes funciones básicas:
  - Cero: La constante $c : \mathbb{N}^0 \to \mathbb{N}$, con $c = 0$ es computable.
  - Sucesor: La función $s : \mathbb{N} \to \mathbb{N}$, con $s(x) = x + 1$ es computable.
  - Proyecciones: Las funciones $\pi_k^n : \mathbb{N}^n \to \mathbb{N}$ y $\pi_k^n(x_1, .., x_n) = x_k$, con $1 \leq k \leq n$ son computables.
- Cálculo λ (creadas por Alonzo Church): Es un sistema formal basado en la abstracción de funciones y sustitución, consiste en lo siguiente:
  - Sea Λ el conjunto de términos del cálculo λ: $\Lambda \to V \mid (\lambda x.\Lambda) \mid (\Lambda\Lambda)$, donde V denota a una variable.
  - La regla básica es la reducción β: $((\lambda x.M)N) \to_\beta M_{[x/N]}$.

  Fue introducido como una manera de formalizar el concepto de computable y puede ser utilizado para expresar y evaluar cualquier función computable, también puede ser usado para simular cualquier máquina de Turing [48].
- Lógica combinatoria (introducida por Moses Schönfinkel y Haskell Curry): Es el fundamento del cálculo λ. En la lógica combinatoria las expresiones lambda son sustituidas por un sistema limitado de combinadores (funciones primitivas que no contienen variables libres o ligadas). Fue originalmente trabajado por Moses Schönfinkel en 1924 y posteriormente redescubierto y expandido por Haskell B. Curry [49].

De entre todos los modelos anteriormente listados, el más parecido a las computadoras modernas es la máquina de Turing. Hay que tener en consideración que además de la máquina de Turing —que a continuación se definirá—, hay otras variaciones (no determinista, con múltiples cintas, con cinta infinita en dos direcciones, etcétera) las cuales resultan ser computacionalmente equivalentes, ya que pueden simularse entre sí.

La equivalencia entre los diversos modelos llevó a plantear la siguiente afimación conocida como la tesis de Church-Turing, ya que se deriva del trabajo de ambos:

Toda función es computable (*effectively computability*) si y sólo si es computable por una máquina de Turing [48].

2.1.1. Definición

Una *Máquina de Turing* es un mecanismo simple que consiste en una cinta de lectura y escritura, un cabezal de lectura y escritura que recorre esa cinta (la cual puede estar en diferentes estados) y una función que regula los movimientos de la cinta. En la Figura 2.2 se puede ver un ejemplo de una máquina de Turing donde las casillas de la cinta tienen distintos caracteres y el cabezal va recorriendo los símbolos y siguiendo las reglas para cada estado [49].

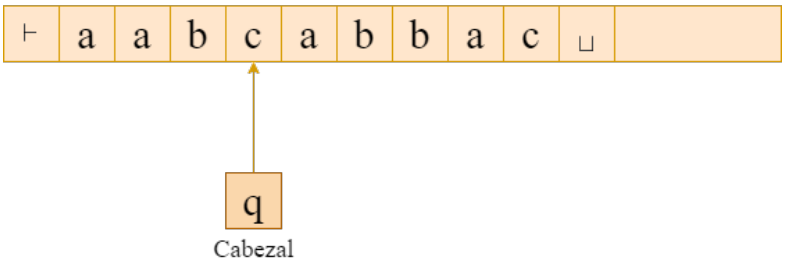


Figura 2.2: Ejemplo de una máquina de Turing.

Está formada por los siguientes elementos:

- Q: Un conjunto de estados.
- Σ: Un alfabeto de entrada.
- Γ: Un alfabeto de cinta tal que $\Sigma \subseteq \Gamma$.
- δ: Un alfabeto de transición $\delta : Q \times \Gamma \to Q \times \Gamma \times \{\rightarrow, \leftarrow\}$.
- Estados inicial, de aceptación y rechazo $s, t, r \in Q$ respectivamente.
- $\sqcup$: El símbolo de espacio en blanco $\sqcup \in \Gamma - \Sigma$.
- $\vdash$: El símbolo de inicio de cinta $\vdash \in \Gamma$.

Intuitivamente, $\delta(p, a) = (q, b, d)$ significa: "Cuando el cabezal se encuentre en el estado p y lea a, escribe b en esa celda, muévete en dirección d y entra en el estado q".

2.2. ¿Qué es la complejidad computacional?

Usualmente medimos la eficiencia de un algoritmo cómo el número de operaciones básicas[1] que realizamos en función del tamaño de la entrada del algoritmo. Una definición equivalente es la cantidad de veces que se mueve la cabeza lectora de una máquina de Turing al procesar el algoritmo en función del tamaño de entrada[24].

En otras palabras, la eficiencia de un algoritmo puede ser obtenida por una función $T : \mathbb{N} \to \mathbb{N}$ de manera que $T(n)$ es la cantidad de operaciones básicas que realiza el algoritmo dadas las entradas de tamaño n[2]. Lo anterior introduce la siguiente definición:

[1] Las operaciones básicas son: asignación, comparación y operaciones aritméticas básicas $(+, -, \times, \div)$.

Sean $f : \mathbb{N} \to \mathbb{N}$ y $g : \mathbb{N} \to \mathbb{N}$ dos funciones, definimos la notación O grande (*Big-O notation*) como:

1. f es $O(g)$ si existe una constante c tal que $f(n) \leq c \cdot g(n)$ para toda n.
2. f es $\Omega(g)$ si g es $O(f)$.
3. f es $\Theta(g)$ si f es $O(g)$ y $\Omega(g)$.

2.3. Reducciones polinomiales

Sean P y Q dos problemas, supóngase que un ejemplar arbitrario e tal que $e \in P$ puede solucionarse convirtiendo a e en un ejemplar $e' \mid e' \in Q$, resolviendo para e' y reduciendo la solución S' en la solución S para E [24], este esquema se puede ver reflejado en la Figura 2.3.

Definición 2.1 (Reducción) *Si existen algoritmos $A_{e \to e'}$ para convertir un ejemplar e con $e \in P$ en e', tal que $e' \in Q$ y $A_{S' \to S}$ para convertir la solución $S' \mid S'$ resuelve Q en $S \mid S$ resuelve P, se dice entonces que existe una **reducción** de P en Q, esta se denota como $P \propto Q$.*

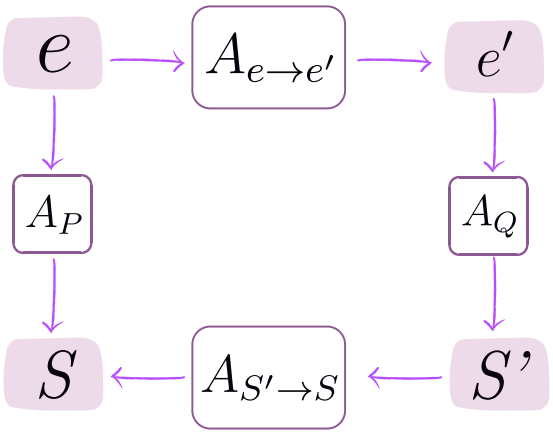


Figura 2.3: Reducción de P en Q: $P \propto Q$.

Generalmente, se realiza una transformación de P en Q, cuando P es muy complejo de resolver de manera directa y Q es más sencillo. A este proceso se le denomina *Algoritmo de Reducción* o *Algoritmo de simplificación* [24].

Teorema 2.1 (TLB (Transformation Lower Bound)) *Supóngase que $P \propto Q$ y que los desempeños computacionales, en el peor de los casos, para los algoritmos que convierten e en e' y S en S' son $f_1(n)$ y $f_2(n)$ respectivamente, donde n es el tamaño del ejemplar E, entonces:*

1. *Por cada algoritmo con desempeño computacional W'(n) que solucione Q, existe un algoritmo para P con desempeño:*

$$W(n) = f_1(n) + W'(n) + f_2(n). \tag{2.1}$$

2. *Si $g(n)$ es una cota mínima sobre la complejidad de P, en el peor de los casos, entonces una cota mínima sobre la complejidad de Q resulta ser:*

$$g'(n) = g(n) - (f_1(n) + f_2(n))\,. \tag{2.2}$$

Para definir qué es una reducción polinomial o reducción de Karp se utilizarán *problemas de decisión*, estos son únicamente los problemas cuya respuesta es sí o no. Trabajar con estos hace la teoría más sencilla, además de que la mayoría de ellos pueden ser convertidos en problemas de decisión [24].

Definición 2.2 (Lenguaje del problema) *Sea U el conjunto de todas las posibles entradas de un problema de decisión y $L_f \mid L_f \subset U$ el conjunto de todas las entradas para las cuales, la respuesta al problema es **sí** (1). L_f entonces es el lenguaje del problema y el problema de decisión consiste en decidir si una entrada x pertence a L_f[24].*

$$L_f = \{x \mid f(x) = 1\} \tag{2.3}$$

Definición 2.3 (Polinomialmente reducible) *Sean L_1 y L_2 dos lenguajes para los cuales los espacios de entradas son U_1 y U_2 respectivamente. El lenguaje L_1 es **polinomialmente reducible** a L_2 si existe un algoritmo A sobre el tamaño de u_1 que en tiempo polinomial convierte cada entrada $u_1 \in U_1$ en $u_2 \in U_2$ tal que $u_1 \in L_1$ si y sólo si $u_2 \in L_2$.*

En general, se escribirá $L_1 \leq_P L_2$, si y sólo si existe una reducción de L_1 a L_2.

Teorema 2.2 *Si $L_1 \leq_P L_2$ y existe un algoritmo polinomial para L_2, entonces existe un algoritmo polinomial para L_1[24]*

La noción de reducción no es simétrica, si $L_1 \leq_P L_2$, no implica que $L_2 \leq_P L_1$. Esta asimetría se da porque la definición de $\leq_P$ requiere que cualquier entrada de L_1 pueda ser convertida en una entrada de L_2, pero no viceversa [24].

Así pues, si $L_1 \leq_P L_2$, entonces se considera que L_2 es un problema más difícil.

2.4. Clases de complejidad

Una *clase de complejidad* es un conjunto de funciones que se pueden calcular dentro de ciertos parámetros [2], el diagrama de los conjuntos de clases de complejidad puede verse en la Figura 2.4, en el primer diagrama podemos observar los conjuntos en caso de que se demostrara que $P \neq NP$, en el segundo puede verse el resultado en caso de que se demuestre que $P = NP$.

2.4.1. Clase *P* y *NP*

Definición 2.4 (*P* o DTIME) *Sea $T : \mathbb{N} \to \mathbb{N}$ una función, definimos como **DTIME**$(T(n))$ el conjunto de todos los problemas que son computables en un tiempo $c \cdot T(n)$ para alguna constante $c > 0$, a esta clase también se le denota por P de tiempo polinomial[24].*

Algunos ejemplos dentro de esta categoría son:

- Decidir si un número n es divisible entre 4.
- Ordenar una colección.
- Dada una gráfica G, decidir si la gráfica es conexa.
- Dada una gráfica G, decidir si la gráfica es un árbol.

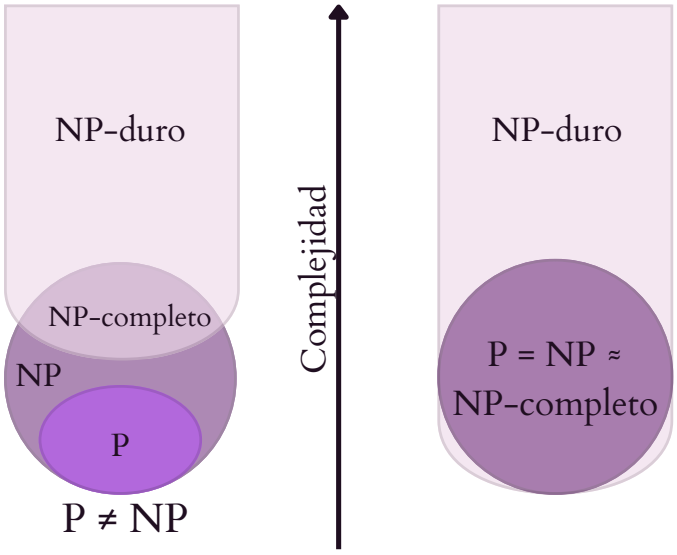


Figura 2.4: Clases de complejidad resultantes si $P = NP$ y si $P \neq NP$.

Para definir la clase *NP* es primero necesario definir una máquina de Turing no determinista.

Definición 2.5 (Maquina de Turing no determinista (NTM)) *Una máquina de Turing no determinista (NTM) es similar a una máquina de Turing tradicional (TM), excepto por el hecho de que la TM produce aceptación si y sólo si existe una secuencia de acciones que lleve al estado de aceptación[24].*

Definición 2.6 *Una* NTM ***reconoce*** *a un lenguaje L si dada una entrada x, existe una secuencia de acciones tal que $NTM(x) = 1$[2].*

En otras palabras la *NTM* debe contar con al menos una posible secuencia de acciones para las entradas de L tales que, las salidas sean aceptables (SÍ / 1).

Definición 2.7 (Clase *NP*) *Para toda función $T : \mathbb{N} \to \mathbb{N}$, $x \in \{0,1\}^*$ y $L \mid L \subseteq \{0,1\}^*$, decimos que $L \in NP$ si existe una constante $c > 0$ y una* NTM *que en tiempo $c \cdot T(n)$ [24]*

$$x \in L \leftrightarrow M(x) = 1. \tag{2.4}$$

Visto de otra manera, es el conjunto de problemas para los cuales existe una *NTM* que los reconoce, cuyo desempeño computacional es polinomial sobre el tamaño de entrada.

Algunos ejemplos de lenguajes en la clase *NP* son:

- *Problema del conjunto independiente*: Dada una gráfica G y un número k, decidir si existe un subconjunto de vértices independientes en G con cardinalidad k.
- *Suma de subconjuntos*: Dada una lista de números $A_1, A_2, A_3, .., A_n$ y un número T, decidir si existe un subconjunto de números cuya suma sea T.
- *Números compuestos*: Dado un número n, decidir si n es un número compuesto (no primo).

2.4.2. Clase *NP-duro* y *NP-completo*

Resulta ser que el problema del conjunto independiente es al menos tan difícil como cualquier otro lenguaje dentro de *NP*, si existe un algoritmo que lo resuelva en tiempo polinomial (en otras palabras, pertenece a P) entonces usando reducciones de Karp, todos los problemas en *NP* también. Esta propiedad es conocida como *NP*-dificultad(*NP-hardness*)[2].

Definición 2.8 (*NP*-duro (*NP*-Hard)) *Un lenguaje L es llamado NP-duro, si cada problema en NP es polinomialmente reducible en L*

$L \in NP-hard$ si $L' \leq_p L$ para todo $L' \in NP$[2]

Definición 2.9 (*NP*-completo) *Un lenguaje L es llamado NP-completo, si:*

- *$L \in NP$.*
- *$L \in NP-hard$[2].*

Usando reducciones de Karp, una definición equivalente es si:

- *$L \in NP$.*
- *$L' \leq_p L$ para algún L' que sea NP-completo[2].*

Teorema 2.3 *Utilizando las anteriores definiciones y propiedades de las reducciones de Karp, podemos decir lo siguiente:*

- *(Transitividad) Si $A \leq_p B$ y $B \leq_p C$ entonces $A \leq_p C$.*
- *Si un lenguaje A es NP-duro y $A \in P$ entonces $P = NP$.*
- *Si un lenguaje A es NP-completo entonces $A \in P$ si y sólo si $P = NP$.*

Algunos problemas que se han demostrado ser *NP*-completos son:

- *Problema 3-SAT*: Dada una expresión lógica en *CNF* (Forma Normal Conjuntiva, *Conjuntive Normal Form*) tal que cada claúsula C contiene exactamente 3 variables, determinar si C se satisface.
- *Ciclo Hamiltoniano*: Dada una gráfica G, determinar si existe un ciclo hamiltoniano en G; es decir, si existe un ciclo que visite todos los vértices de la gráfica una vez sin repetir arista, un ejemplar del problema puede encontrarse en la Figura 2.5 donde las aristas color rosa corresponden a un ciclo hamiltoniano y las restantes en negro son las que no pertenecen al ciclo, pero pertenecen a la gráfica original.
- *Problema de la mochila*: Dado un conjunto C tal que, para cada $c \in C$ se tiene asociado un tamaño $s(c)$ y un valor $v(c)$. Determinar si $\exists B \subseteq C$ tal que la suma de $s(c_i)$ no supere el peso máximo y cuya suma de $v(c_i)$ sea mayor que la cota mínima.
- *Problema del arbol de Steiner en gráficas*: Dada una gráfica $G = (V, A)$, una función de peso para cada arista, $R \subseteq V$ y un número entero positivo B. Determinar si existe un subarbol en G que incluya todos los vértices de R tal que la suma de los pesos en las aristas no sea mayor a B.

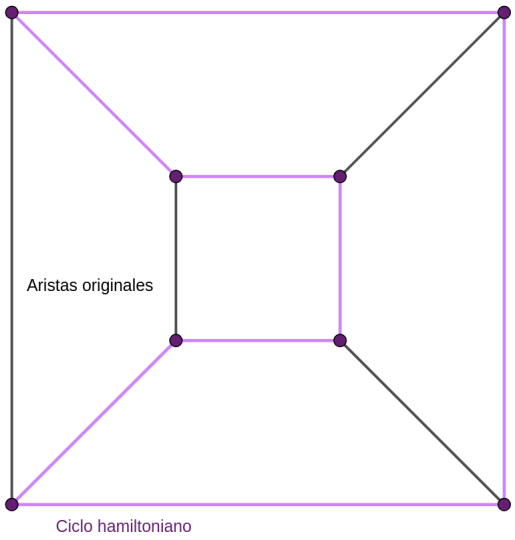


Figura 2.5: Ejemplo de un ciclo hamiltoniano de una gráfica.

2.5. Problemas de optimización

La teoría de *NP*-completitud corrobora que algunos problemas en *NP* no pueden ser resueltos por algoritmos eficientes (polinomiales). Esto también es válido para problemas de optimización, los cuales son problemas *NP*-duros en el sentido de que un algoritmo de optimización eficiente para uno de estos problemas implicaría un algoritmo de decisión eficiente para sus correspondientes versiones de problemas de decisión (*NP*-completo).

Para poder hacer algún progreso dentro de una cantidad razonable de tiempo tenemos que distanciarnos de los requisitos que ponemos a un algoritmo. Existen varias posibilidades, una de ellas, es una rama importante dentro de la teoría de la complejidad computacional la cual estudia el beneficio de la *aleatorización* [39].

En esta rama, los algoritmos ya no son deterministas, sino aleatorizados; el propósito entonces es calcular de forma eficiente soluciones factibles. Sin embargo, la parte perjudicial es que no garantiza el éxito en todos los casos.

Otro enfoque existente es el de los *algoritmos de aproximación*, estos son utilizados para problemas de optimización, la idea principal es no buscar el óptimo global, sino una salida que sea la mejor posible. Con esta filosofía, se espera obtener algoritmos de aproximación eficientes para problemas difíciles. En otras palabras, se trata de ganar eficiencia, sacrificando la propiedad de la solución de ser la óptima a ser aproximadamente óptima.

Dentro de la categoría de problemas de optimización, la que se estudia en este documento, es la categoría de problemas de optimización combinatoria [39].

Definición 2.10 (Problemas de optimización combinatoria) *Un problema de optimización combinatoria es un problema de minimización o maximización que consiste de tres partes:*

- *Un conjunto $\mathcal{I}$ de ejemplares del problema.*

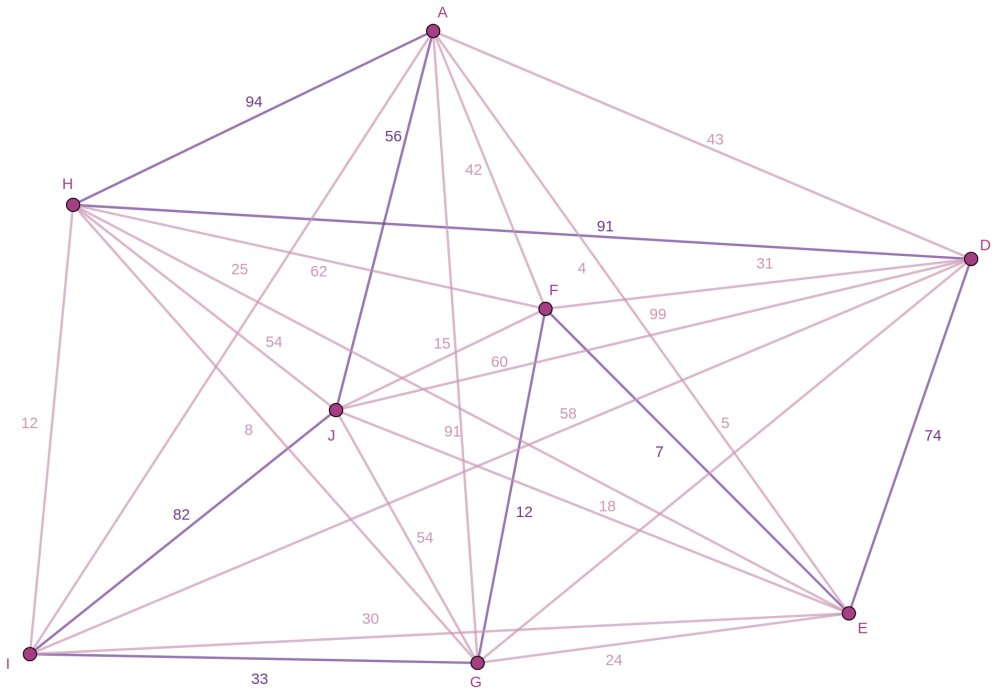


Figura 2.6: Ejemplar del problema de optimización del agente viajero.

- *Para cada ejemplar $I \in \mathcal{I}$ un conjunto finito de soluciones $Sol(I)$ de posibles soluciones. Este conjunto es conocido como* soluciones factibles *para el ejemplar I.*
- *Una función $m : \{(I, \sigma) \mid I \in \mathcal{I}, \sigma \in Sol(I)\} \to \mathbb{Q}_+$ donde $\mathbb{Q}_+$ es el conjunto de todos los números racionales. El valor $m(I, \sigma)$ es llamado* valor de la solución factible σ.

Una solución factible σ^* es llamada solución óptima *para un ejemplar I del problema $\left(\mathcal{I}, \{Sol(I)\}_{I \in \mathcal{I}}, m\right)$ para todas las soluciones $\sigma \in Sol(I)$ se cumple:*

- *$m\left(I, \sigma^*\right) \leq m(I, \sigma)$ en caso de que se esté interesado en la solución mínima, a esto se le llama problema de minimización combinatoria.*
- *$m\left(I, \sigma^*\right) \geq m(I, \sigma)$ en caso de que se esté buscando la solución máxima, a esto se le llama problema de maximización combinatoria.*

Se denota al valor óptimo por:

$$OPT(I) := m(I, \sigma^*). \tag{2.5}$$

Si el conjunto $Sol(I)$ de todas las soluciones factibles es vacío definimos a los valores óptimos como $OPT(I) := 1$ [39].

En campos como *Cómputo evolutivo* la función m también es conocida como función de aptitud (*fitness*).

Los siguientes son algunos ejemplos de problemas de optimización combinatoria.

- *Problema de optimización del agente viajero*: Un ejemplar I de este problema es dada por una gráfica $G = (V, \frac{V^2}{2}), V = \{v_1, v_2, .., v_n\}$ junto con la matriz de adyacencias $[d_{ij}] \in \mathbb{Q}_+^{n \times n}$.

  El conjunto de soluciones factibles $Sol(I)$ es el conjunto de las permutaciones de n nodos en V. La función m que nos da el valor de una solución factible es definida como la suma de las distancias a lo largo del ciclo:

$$m(I, \sigma) := \sum_{i=1}^{n-1} d_{\sigma(i),\sigma(i+1)} + d_{\sigma(n),\sigma(1)}. \tag{2.6}$$

  El problema del agente viajero es un problema de minimización combinatoria [39], puede verse un ejemplar del problema en la Figura 2.6 donde se puede apreciar un conjunto de vértices y aristas, así como un ciclo, el cual corresponde a una solución factible del problema [39].

- *Problema de optimización de Bin Packing*: Un ejemplar I del problema es dada por un conjunto finito S, B un número natural y para cada $s \in S$ existe $c_s \in \mathbb{N}$, c_s es visto como el tamaño del elemento s y B es visto como el tamaño máximo al que se puede llegar.

  El conjunto de todas las soluciones $Sol(I)$ es el conjunto de todas las particiones de S en un número finito de subconjuntos disjuntos $S_1, S_2, ..., S_k$; para alguna $k \leq |S|$, tal que para toda $1 \leq i \leq k$ se cumple $\sum_{s \in S_i} c_s \leq B$.

  La función m que da el valor a una solución factible $S_1, S_2, .., S_k$ está definida como la *cardinalidad del subconjunto*, en otras palabras:

$$m(I, (S_1, S_2, .., S_k)) := k. \tag{2.7}$$

  El problema trata con la cuestión de cuántos contenedores de un tamaño dado hay necesarios para embalar una serie de objetos. Es de nuevo un problema de minimización combinatoria [39].

- *Problema de coloración máxima*: Un ejemplar I del problema es dada por una gráfica $G = (V, A)$ unidireccional, $w : V \to \mathbb{N}$ una función de peso sobre los vértices.

  El conjunto de todas las soluciones $Sol(I)$ es el conjunto de todas las coloraciones válidas $C_1, C_2, .., C_k$ en G, una coloración válida es tal que:

  - $\forall v \in V$ existe $c(v)$.
  - $\forall u, v \in A : c(v) \neq c(u)$.

  Cada clase de color tiene un peso igual al vértice con mayor peso dentro de esta clase. La función m de una coloración está definida como:

$$m(I, (C, C_2, ..., C_k)) := \sum_{i=1}^{k} max_{v \in C_i}(w(v)). \tag{2.8}$$

  El objetivo del problema es producir una coloración tal que se minimice el peso total de las clases de color, es entonces un problema de minimización combinatoria [60], en la Figura

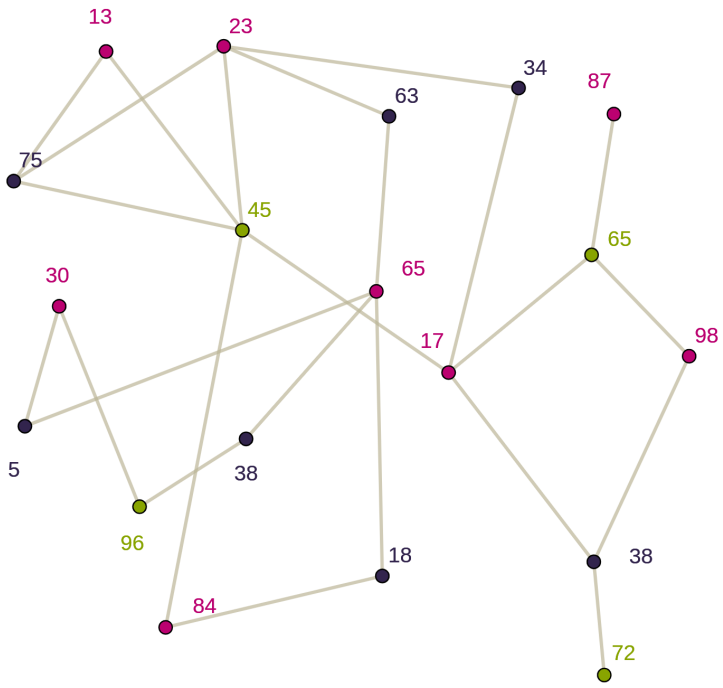


Figura 2.7: Ejemplar del problema de coloración máxima.

2.7 se puede ver un ejemplar del problema de coloración máxima donde cada vértice tiene un color correspondiente a una clase de coloración [24].

- *Problema de maximización de la influencia*: Un ejemplar del problema I es dada por una gráfica de influencia $G(V, A)$ y un número entero k.

  Una *gráfica de influencia* es una gráfica dirigida $G = (V, A)$ con pesos donde los nodos representan individuos y las aristas representan las relaciones sociales entre ellos. Cada arista $u \to v$ cuenta con un peso $p(u, v) \in [0, 1]$ que cuantifica el nivel de influencia de u en v, se dice que el nodo v es vecino del nodo u si $u \to v \in A$ [76](figura 2.8).

  El problema de la maximización de influencia busca un pequeño conjunto de nodos de tamaño k que influyan en el número máximo de nodos dentro de la gráfica de influencia. Para esto se cuenta con dos modelos estocásticos en cascada, también conocidos como modelos de difusión, para estudiar este problema:

  - Cascada independiente (IC, *Independent Cascade*).
  - Umbral lineal (LT, *Linear Threshold*).

  En estos modelos cada nodo de la gráfica está **activo** o **inactivo**, inicialmente ambos modelos especifican el conjunto inicial de nodos activos, denominado *conjunto semilla*, mientras que el resto de nodos se encuentran inactivos. Una vez que un nodo se activa, significa que ha sido influenciado con éxito por sus vecinos activos durante el resto del proceso.

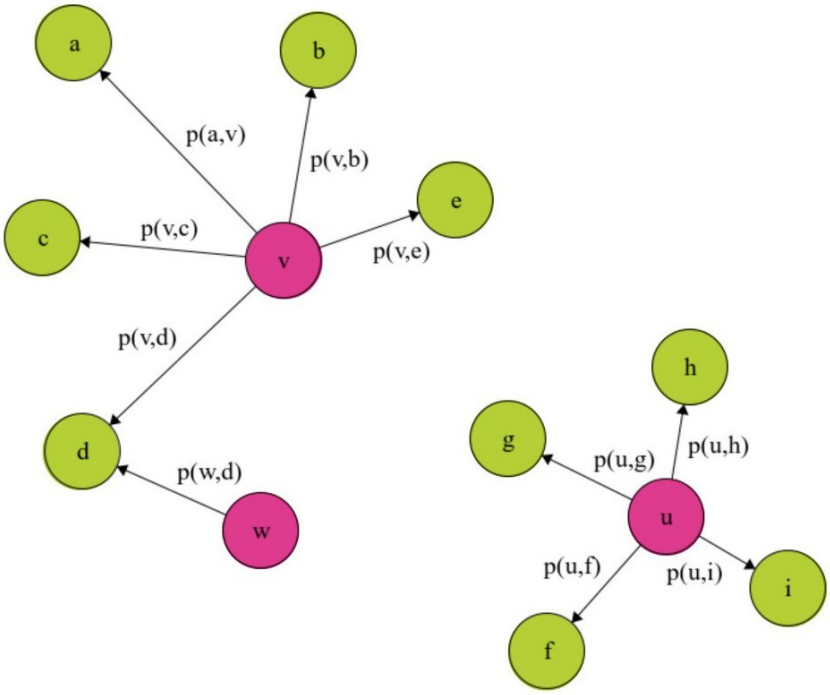


Figura 2.8: Ejemplo de una gráfica de influencia.

Sea S un conjunto semilla de nodos activos en G tal que $|S| \leq k$ y $I_G(S)$ el numero esperado de nodos activos al final del proceso en un modelo de influencia (IC o LT). La función I_G se conoce como *función de influencia* de S y el valor $I_G(S)$ es la *propagación de influencia* de S o simplemente *propagación*.

Dado lo anterior definimos el conjunto de soluciones $Sol(I)$ como todas las particiones $S_1, S_2, ..., S_n$ de V tal que cada partición $|S_i| \leq k$ y $\forall v \in S_i$, v está activo.

La función m que nos da el valor de una solución factible S_i es la propagación de influencia del conjunto S_i, en otras palabras:

$$m(I, S_1) := I_G(S_i). \tag{2.9}$$

El objetivo del problema es obtener un conjunto semilla S de k nodos tal que $I_G(S)$ sea máximo, por ende es un problema de maximización combinatoria [76]. En la Figura 2.8 se puede ver un ejemplar del problema, los nodos verdes son los inactivos y los rosas son los que se encuentran activos.

En el siguiente capítulo se ahondará en el concepto de heurísticas y en específico la heurística de optimización por enjambre de partículas.

Capítulo 3

Enjambre de partículas

Como se explicó en el Capítulo 2, existen problemas denominados como *NP*-duro, los cuales son difíciles de resolver utilizando algoritmos convencionales, sin embargo al ser necesaria una solución para varios de estos problemas, se comenzaron a utilizar técnicas conocidas como heurísticas con el fin de aproximarse a una solución aceptable en un tiempo razonable.

3.1. Heurísticas

Originalmente el término *heurística* fue utilizado en ciencias sociales para referirse a estrategias usadas por las personas para reducir la demanda cognitiva asociada a ciertas tareas demandantes [62]. En Ciencias de la Computación son una clase de métodos utilizados para obtener una solución cuasi-óptima a problemas de optimización complejos, entre ellas se encuentran recocido simulado, algoritmos genéticos, modelado de agentes, evolución diferencial, entre otras. La mayor parte de estas estrategias son algoritmos estocásticos [27], a continuación se describirá brevemente algunas de estas estrategias.

3.1.1. Algoritmos genéticos

Los algorimos genéticos (denominados originalmente *planes reproductivos genéticos*) fueron desarrollados por John H. Holland entre 1960 y 1970, motivado por resolver problemas de aprendizaje de máquina [34, 33], estos algoritmos trabajan principalmente en espacios de búsqueda discretos. En estos algoritmos n agentes son codificados generalmente en arreglos de tamaño m los cuales representan posibles soluciones del problema a resolver [64].

Dentro de este proceso, al arreglo binario se le denomina *cromosoma*, a cada posición de la cadena se le denomina *gen* y al valor dentro de esta posición se le llama *alelo* [10].

El algoritmo genético cuenta con las siguientes operaciones:

1. Generación de la población inicial de tamaño n.
2. Cálculo de la aptitud de cada individuo, en este paso se hace uso de una función de aptitud.
3. Aplicación de operadores genéticos:

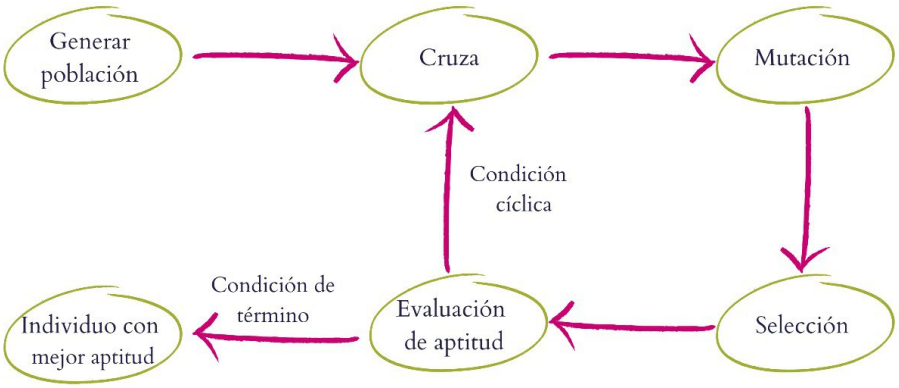


Figura 3.1: Ciclo de un algoritmo genético básico.

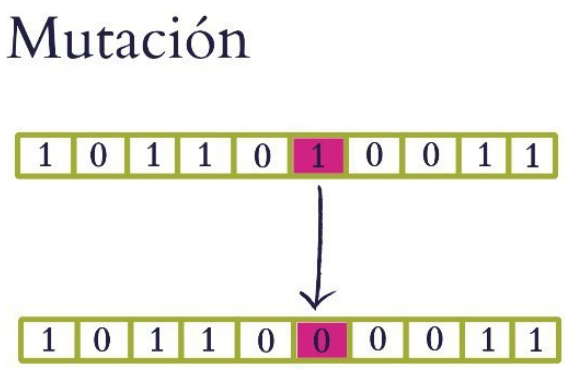


Figura 3.2: Operador de mutación en el algoritmo genético.

- Cruza: En esta operación se toman dos individuos p_1 y p_2 y se generan dos soluciones nuevas h_1 y h_2 combinando partes del arreglo p_1 y completando el restante del arreglo con los datos de p_2, a los arreglos h_1 y h_2 se les denomina como los hijos de p_1 y p_2. La operación se puede visualizar más claramente en la Figura 3.3, donde h_1 es creado uniendo la mitad izquierda del arreglo p_1 y la mitad derecha del arreglo p_2, por otro lado h_2 es creado uniendo las mitades restantes de cada padre.

- Mutación: Para cada individuo recién obtenido de la cruza, cada bit del arreglo es invertido con probabilidad p, esta probabilidad generalmente es $p = 1/n$, [64]. Un ejemplo del operador de mutación se puede ver en la Figura 3.2, donde a una de las casillas dentro del arreglo se les aplica la operación `not`.

4. Selección de la población: Del total de la población obtenida de padres e hijos, se hace una selección para contar de nuevo con una población de n individuos, existen distintas estrategias para realizar esto entre las cuales la más común es selección por aptitud proporcional, donde los individuos con mejor aptitud tienen mas probabilidad de ser seleccionados para

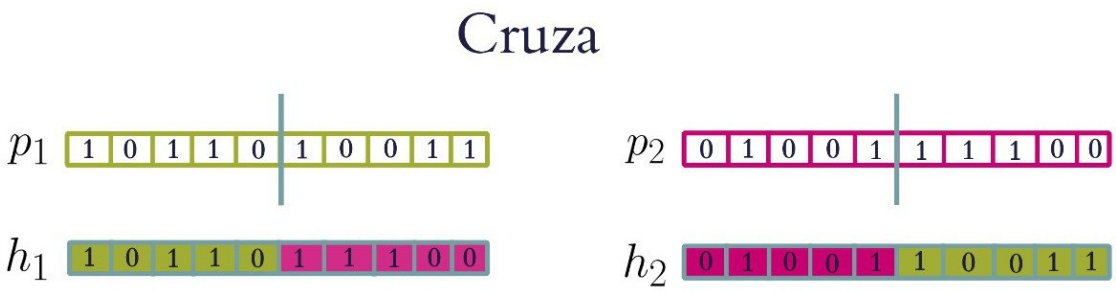


Figura 3.3: Operador de cruza en el algoritmo genético.

la siguiente generación [64].

Las operaciones de cálculo de aptitud, cruza, mutación y selección se repiten hasta cumplir el criterio de paro del algoritmo, el ciclo se puede visualizar más claramente en la Figura 3.1 donde se ilustra cómo se comporta el algoritmo desde la generación de la población, hasta que se cumple el criterio de paro y se regresa al individuo con mejor aptitud.

Algunas aplicaciones de los algoritmos genéticos son:

- Optimización en planificaciones de pagos: Dentro de una organización pequeña es importante enfatizar en la planeación de pagos como lo son impuestos, materia prima, rentas, entre otros. En 2018 el computólogo Mamta Madan propuso solucionar el problema utilizando algoritmos genéticos en su artículo *Bio-Inspired Computation for Optimizing Scheduling* [55].
- Optimización en el problema de las 8 reinas: El problema de las 8 reinas consiste en colocar 8 reinas en un tablero de ajedrez de tal manera que ninguna reina ataque a otra, en el libro *Artificial intelligence—A Modern Approach* [73] se describe como utilizar el algoritmo genético para resolver este problema.
- Problema de asignación de horarios: En 1992 en el artículo *Genetic Algorithms: A New Approach to the Timetable Problem* [12], Colorni, Doriego y Maniezzo propusieron resolver el problema de la asignación de horarios en una escuela italiana utilizando algoritmos genéticos.

3.1.2. Recocido simulado

La técnica de recocido simulado es un algoritmo de búsqueda local no voraz que define una búsqueda mas sofisticada desde su punto de partida a sus vecinos, aceptando con cierta probabilidad soluciones que no mejoren la aptitud de la solución actual [11].

El algoritmo surgió alrededor del año 1953 cuando fue utilizado para simular en una computadora un tratamiento térmico en metales [11]. El tratamiento térmico o recocido en metalurgia es el proceso utilizado para templar metales y vidrio, el cual consiste en calentar el material a una temperatura alta y gradualmente disminuirla para así poder liberar esfuerzos en los bordes de la estructura molecular [73].

La idea de aplicar la metodología de la simulación a problemas de optimización combinatoria se planteó en 1983 por los científicos Kirkpatrick, Gelatt y Vecchi en su artículo *Optimization by Simulated Annealing* [45].

El método posee los siguientes parámetros:

- S el conjunto de soluciones factibles para el problema.
- $i \in S$ una solución factible.
- $\mathcal{N}(i) \subseteq S$ el conjunto de soluciones factibles alcanzables desde i (vecinos de i).
- f la función de aptitud a minimizar.
- Δ la diferencia $f(j) - f(i)$.
- t la temperatura actual, este parámetro va ir decreciendo durante la ejecución del algoritmo.
- g la función que actualiza la temperatura, usualmente $g(t) = at$ para alguna constante $a < 1$.
- p la probabilidad de aceptación de un vecino $j \in \mathcal{N}(i)$

$$p = \begin{cases} 1 \; si \; \Delta < 0 \\ e^{-\Delta/t} \; si \; \Delta \geq 0. \end{cases} \tag{3.1}$$

- q una función aleatoria con distribución uniforme en el intervalo $[0, 1]$.

Dados los parámetros anteriores, el algoritmo en cada iteración cuenta con la temperatura actual (calculada con la función g, la cual arrojará resultados menores conforme avanzan las iteraciones) y una solución actual (S_i). Al inicio de cada iteración, calcula una siguiente solución S_j (solución alcanzable desde S_i) y dada la temperatura actual y la función p decide si acepta o no la solución; si la acepta, entonces S_j se vuelve la solución actual del sistema, el pseudocódigo se ve de la siguiente manera:

```
solucion_actual = i
for iteracion in range(iteracion_maxima):
  t = g(t)
  j = N(solucion_actual)
  if p(solucion_actual,j,t) >= q:
    solucion_actual = j
return solucion_actual
```

Código 3.1: Pseudocódigo del algoritmo de recocido simulado [11].

Como se puede ver, mientras la temperatura es alta, el algoritmo acepta con una mayor probabilidad soluciones que no mejoran la aptitud actual, conforme la temperatura va descendiendo (o bien, llegando a un estado de equilibrio), el algoritmo más dificilmente acepta este tipo de soluciones, por lo que va convergiendo a un óptimo en la función [11].

Algunas aplicaciones para el algoritmo de recocido simulado son:

- Métodos en econometría: Varios métodos en econometría dependen de la optimización para estimar los parámetros del modelo, desgraciadamente esta optimización puede ser difícil porque estos algoritmos pueden fácilmente fallar o no encontrar valores favorables. William Goffe, Gary Ferrier y John Rogers propusieron en 1992 en su artículo *Simulated annealing: An initial application in econometrics* [28] la utilización del recocido simulado para optimizar estos métodos.

- Reconocimiento de huellas digitales: En 2011 Xiaofang Peo, Nan Li y Shuiping Wang propusieron en su artículo *Application of Simulated Annealing Algorithm in Fingerprint Matching* [67] la utilización de recocido simulado para el emparejamiento de las características de una huella dactilar extraídas usando filtros de Gabor contra un conjunto de imágenes de huellas dactilares.
- Modelado y agrupamiento de glaucoma: En 2019 Mohd Jilani, Allan Tucker y Stephen Swift propusieron en su artículo *An application of generalised simulated annealing towards the simultaneous modelling and clustering of glaucoma* [38] aplicar el algoritmo de recocido simulado al modelado y agrupación simultáneos de datos del campo visual, los cuales son comúnmente usados para controlar el glaucoma.

3.2. Optimización por enjambre de partículas

El algoritmo de optimización por enjambre de partículas fue introducido en 1995 por Kennedy y Eberhart [80] como un método de optimización de funciones no lineales continuas, el método fue descubierto a través de la simulación de un modelo social simplificado [42].

El algoritmo tiene raíces en dos metodologías de componentes principales, la vida artificial (A-life) y en la teoría de parvada de aves, cardúmenes de peces y particularmente en la teoría de enjambres. Sin embargo, también está relacionado con el cómputo evolutivo, tiene vínculos con algoritmos genéticos y programación evolutiva [42].

3.2.1. Inspiración en la conducta social

Varios científicos crearon simulaciones computacionales de diversas interpretaciones del movimiento de organismos en una parvada de aves o cardúmen de peces. Entre ellos, Reynolds en 1987 [72] y Heppner y Grenander en 1990 [31].

Por un lado Craig Reynolds estaba intrigado por la estética de la coreografía en las bandadas de pájaros, en su artículo *Flocks, Herds and Schools: A Distributed Behavioral Model* diseñó un simulador de parvadas de aves, donde cada ave se implementa como un ente independiente que navega de acuerdo a su percepción local del entorno [72]. Para el simulador se establecieron las reglas:

1. Evadir colisiones: Los individuos deben evitar colisionar con vecinos cercanos.
2. Coordinar velocidad: Los individuos deben intentar coordinar su velocidad con la de sus vecinos cercanos.
3. Enfocado en la cercanía con los vecinos (*flock centering*): Los individuos buscan estar cerca de los vecinos más cercanos.

Por otro lado, el zoólogo Heppner estaba interesado en descubrir las reglas que permitían a un gran número de pájaros aglomerarse, cambiar de dirección, dispersarse y reagruparse de manera coordinada [42]. En su artículo *A Stochastic Nonlinear Model for Coordinate Bird Flocks* creó un modelo estocástico de ecuaciones diferenciales en el cual muestra una conducta *realista* de una bandada, las funciones consideraban la atracción a una zona de descanso, atracción no lineal a los compañeros en la parvada, preservación de la velocidad de vuelo y un proceso estócastico n-dimensional de Poisson.

El núcleo de la simulación fue inicialmente inspirado en el juego de la vida de John Conway en 1970 [31], posteriormente para especializar el modelo para bandadas, se tomaron en cuenta las

siguientes reglas para diseñar las ecuaciones del modelo [31]:

1. Se exhibe un modelo colectivo en el que no se considera a un líder dentro del colectivo.
2. Mostrar la conducta errática, no sistemática vista en parvadas provocada por ráfagas de viento, perturbaciones externas, entre otras.
3. Deseo de la parvada por una zona de descanso.
4. La parvada puede dar vueltas por un tiempo indefinido sobre la zona de descanso.
5. Los individuos pueden cambiar de posición dentro de la parvada.
6. Cada cierto tiempo la parvada puede dividirse en dos o más subparvadas.
7. Las trayectorias de vuelo de los individuos pueden ser aproximadamente paralelos y aleatorios hasta cierto punto.
8. El modelo no agrega otra influencia externa que actúe sobre la bandada para dirigirlo a comportarse como una bandada.

Ambos modelos dependían en gran medida de la manipulación de distancias entre individuos, es decir, la sincronía de la bandada era pensada como una función del esfuerzo de las aves para mantener una distancia óptima entre los miembros y sus vecinos [42].

No es exagerado suponer que las mismas reglas aplican en la conducta social de los animales incluyendo manadas, bancos, parvadas y humanos. El sociólogo E. O. Wilson dijo en su libro *Sociobiology: The New Synthesis* [77] en referencia a la formación de bancos de peces:

> "Al menos en teoría, los miembros del banco pueden beneficiarse de los descubrimientos y la experiencia previa del resto de los miembros al momento de buscar comida. Esta ventaja puede volverse decisiva, superando las desventajas durante la competición por comida, siempre que el recurso se distribuya de manera impredecible (p. 209)."

Lo cual sugiere que el intercambio de información entre los miembros del banco ofrece una ventaja evolutiva, esta hipótesis fue fundamental para el desarrollo del algoritmo por optimización de enjambre de partículas [42].

Uno de los motivos principales para Eberhart y Kennedy fue modelar la conducta social humana, la cual no es idéntica a los bancos de peces o parvadas de aves. Una diferencia importante es el carácter abstracto. Los pájaros y peces ajustan físicamente su movimiento para evitar depredadores, buscar comida y pareja, optimizar el entorno, etcétera. Los humanos no ajustan solamente los movimientos físicos, si no también variables cognitivas. Por lo general, no caminamos al mismo paso y giramos al mismo tiempo, más bien, tendemos a ajustar nuestras creencias y actitudes para coordinarlas con las de nuestros compañeros sociales [42].

3.2.2. Algoritmo original

A diferencia de los algoritmos evolutivos, el algoritmo de optimización por enjambre de partículas no utiliza un mecanismo de selección, todos los miembros del enjambre sobreviven desde el inicio de la ejecución hasta el final y las interacciones entre los miembros mejoran la calidad de las soluciones encontradas en el tiempo [44].

Un vector numérico de D dimensiones, generalmente creado de manera aleatoria dentro de un espacio de búsqueda es la abstracción de un punto de d-dimensiones en el plano cartesiano de d-

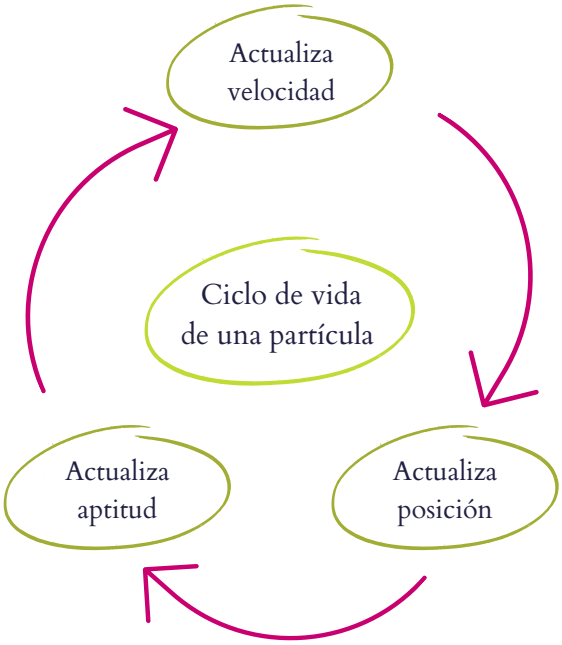


Figura 3.4: Ciclo de vida de una partícula en el modelo original.

dimensiones. Debido a que mueve su posición en el plano cartesiano probando nuevos parámetros, el punto se considera una partícula.

Puesto que las partículas realizan esto de manera simultánea y tienden a converger en zonas óptimas dentro del espacio de búsqueda, se les conoce como enjambre de partículas [44].

Además de este movimiento (normalmente en un problema sobre un espacio euclidiano), a cada partícula se le asigna un número de vecinos con los que están enlazados bidireccionalmente. Originalmente la implementación del algoritmo define el comportamiento de las partículas en dos formulas, la primera modela la velocidad o tamaño del cambio en la posición de la partícula y la segunda mueve la posición de la partícula sumando la velocidad a su anterior posición [44].

$$v_{id}^{(t+1)} \leftarrow \alpha v_{id}^{(t)} + U(0,\beta)\left(\rho_{id} - x_{id}^{(t)}\right) + U(0,\beta)\left(p_{gd} - x_{id}^{(t)}\right). \tag{3.2}$$

$$x_{id}^{(t+1)} \leftarrow x_{id}^{(t)} + v_{id}^{(t+1)}. \tag{3.3}$$

Donde:

- i: Es el índice de la partícula.
- d: Es la dimensión del espacio de búsqueda.

- $\vec{x_i}$: Es la posición de la partícula i.
- $\vec{v_i}$: Es la velocidad de la partícula i.
- $\vec{p_i}$: Es la mejor posición de la partícula i encontrada hasta ese momento
- g: es el índice del mejor vecino de la partícula i
- α y β son constantes, aunque éstas pueden variar dependiendo de la implementación y el problema a resolver, la versión estándar usa $\alpha = 0.7298$ y $\beta = \frac{\psi}{2}$, donde $\psi = 2.9922$, esto siguiendo un análisis publicado por Clerc y Kennedy en 2002 [8], la constante α se suele conocer como peso de inercia y β como la aceleración constante.
- $U(0, \beta)$ es un generador de números aleatorios uniforme.

El programa evalúa el vector de posición de la partícula p_i en la función $f(\vec{x})$ y compara éste resultado con el mejor obtenido por p_i hasta ahora (p_{best_i}). Si el resultado es mejor que p_{best_i}, entonces p_{best_i} pasa a tener el valor actual de p_i, en la Figura 3.4 puede verse con mejor claridad el ciclo de vida de cada partícula dentro del enjambre.

Cuando se ejecuta el sistema, cada partícula se mueve alrededor de una región cercana a los mejores anteriores $\vec{p_i}$ y $\vec{p_g}$, a medida que estas variables se actualizan, la trayectoria de las partículas cambia a nuevas regiones del espacio de búsqueda, estas comienzan a agruparse alrededor de óptimos locales y se obtienen mejores valores de la función de aptitud [44].

3.2.3. Variantes del algoritmo

Al ser el algoritmo de optimización por enjambre de partículas general y fácilmente modificable, existen distintos tipos de variaciones para ajustarse al problema a resolver entre los cuales se encuentran los siguientes.

3.2.3.1. Algoritmo STP-MSO

En 2015 Decroos, De Causmaecker y Demoen describieron en *Solving Euclidean Steiner Tree Problems with Multi Swarm Optimization* [16] una variación del algoritmo de optimización por enjambre de partículas para resolver el problema del árbol de Steiner euclideano, el cual consiste en una aplicación del algoritmo de optimización de enjambres múltiples (*Multi Swarm Optimization*, MSO por sus siglas en inglés), donde se cuenta con un conjunto de subenjambres, cada uno busca su propia ubicación óptima local de un punto Steiner al ser conscientes de la posición de los puntos Steiner de otros subenjambres, una visualización de los subenjambres puede encontrarse en la Figura 3.5, donde el conjunto de segmentos morados corresponden al árbol euclideano original y cada grupo de puntos (o *cluster*) corresponden a un subenjambre, los cuales en el mejor de los casos resultaran en un nuevo punto Steiner.

El algoritmo descrito realiza una serie de iteraciones, ya sea predefinidas o hasta cumplir algún criterio de paro. Cada iteración mueve todas las partículas de un subenjambre en dirección de su mejor partícula, teniendo en cuenta la mejor partícula de otros subenjambres, y con cierto factor aleatorio. En cada iteración, un nuevo subenjambre (en este caso, un nuevo punto Steiner) es añadido al sistema, esto sujeto a condiciones que limitan el número de subenjambres o aptitud, y con cierto componente aleatorio al generarlo [16].

La evolución de las partículas está basada en su posición actual y velocidad, ya que se considera la ubicación de los puntos de Steiner en el plano euclideano. Los subenjambres se eliminan cuando no contribuyen a mejorar la solución actual [16].

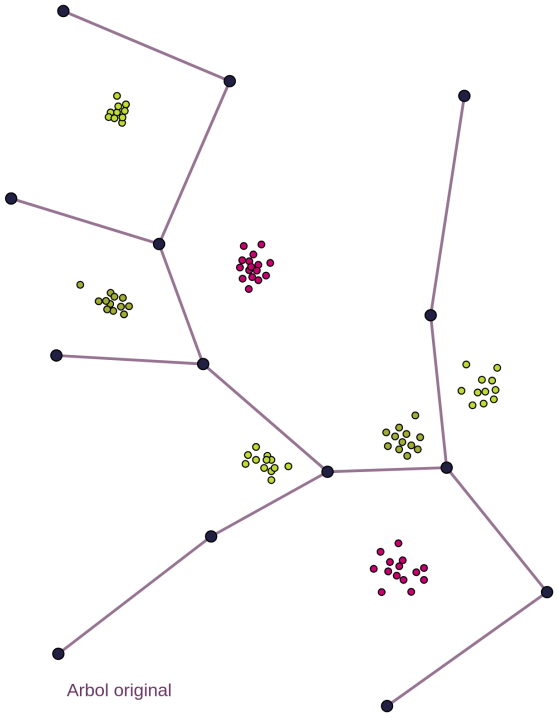


Figura 3.5: Enjambres interactuando en el algoritmo STP-MSO.

3.2.3.2. Enjambres de partículas binarias

Un enjambre de partículas binarias se crea fácilmente tratando el vector de la velocidad como umbral de probabilidad. Los elementos del vector de velocidad son calculados dentro de una función del estilo $S(v) = \frac{1}{(1+exp(-v))}$, produciendo un resultado entre 0 y 1. Un número aleatorio es generado y comparado con $S(v_{id})$ para determinar si x_{id} será 0 o 1. A pesar de que se han propuesto sistemas discretos de mayor cardinalidad, es difícil definir conceptos tales como distancia y dirección [44].

Kennedy y Spears compararon en 1998 este enjambre de partículas binarias con varios tipos de algoritmos genéticos utilizando un generador de problemas binarios aleatorio. En ese estudio, el enjambre de partículas binarias fue el único algoritmo que encontró el óptimo global en cada prueba individual, independientemente de las características del problema. Resultó ser más rápido que los algoritmos genéticos con cruce, mutación o ambas operaciones en todas las pruebas exeptuando las más simples, con dimensiones pequeñas y pocos óptimos locales, el algoritmo genético con la operación de mutación fue el más rápido [43].

3.2.3.3. PSO con control de diversidad

Algunos investigadores han notado una tendencia del enjambre a converger prematuramente en óptimos locales. Con el fin de disminuir este comportamiento, se han implementado diversas aproximaciones para corregir la disminución de diversidad y por ende evitar la convergencia prematura en un único óptimo [68].

En 2002 Løvbjerg y Krink utilizaron la criticidad autoorganizada para ayudar al algoritmo a contar con una mayor diversidad. En su propuesta, cuando dos partículas están demasiado cerca una de la otra, una variable llamada *Valor crítico* se incrementa, cuando alcanza cierto umbral de criticidad la partícula dispersa su criticidad a otras partículas cercanas y posteriormente se reubica [52], un diagrama de estas partículas puede encontrarse en la Figura 3.6.

Otros investigadores han intentado diversificar el enjambre de partículas evitando que las partículas se agrupen demasiado en cierta región del espacio de búsqueda. En 2002 Blackwell y Bentley en su artículo *Don't push me! Collision-avoiding swarms* propusieron enjambres que evitan colisiones gracias a un sistema de reducción de atracción al centro del enjambre [4]. Krink y Vesterstromen propusieron en 2002 un modelo con partículas *espacialmente extendidas* donde cada partícula se conceptualiza como si estuviera rodeada por una esfera de cierto radio. Cuando la partícula espacialmente extendida se encuentra con otra, rebotan cambiando su posición [50].

En 2002 Xie, Zhang y Yang agregaron entropía negativa al enjambre de partículas para evitar la convergencia prematura (convergencia excesivamente rápida hacia un óptimo local de mala calidad). En algunas ocasiones, se agregaba aleatoriedad a la velocidad de la partícula y en otras, era agregada a la ubicación de la partícula, obteniendo así una especie de *enjambre de partículas disipativas* [78].

En el siguiente Capítulo se verá con detalle la implementación de un programa realizado en `Python` modelando la heurística de optimización por enjambre de partículas y aplicándola al problema del árbol de Steiner euclideano.

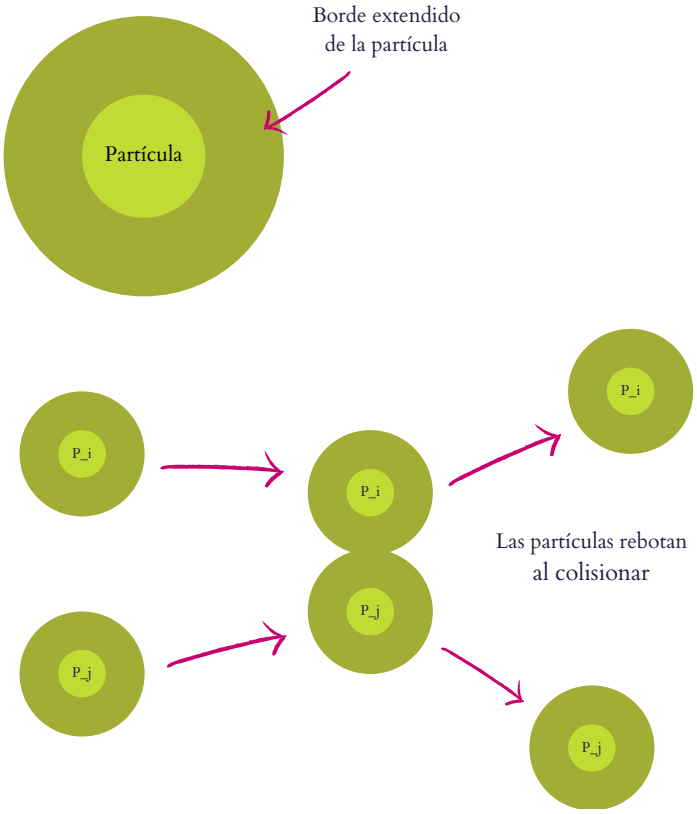


Figura 3.6: Partículas propuestas por Krink y Vesterstromen.

Capítulo 4

Implementación

Con el fin de encontrar soluciones a ejemplares del problema del árbol de Steiner euclideano mínimo haciendo uso de una versión del algoritmo de optimización por enjambre de partículas, fue necesaria la creación de un programa computacional que modele los ejemplares del problema, el algoritmo y una manera para visualizar las gráficas generadas. El programa en cuestión se realizó en Python 3 y para el control de versiones se utilizó Git.

De igual manera, para la realización del código se contó con un módulo con funciones auxiliares y tres clases (`Partícula`, `Enjambre` y `Steiner`), éstas se pueden apreciar en el diagrama 4.1.

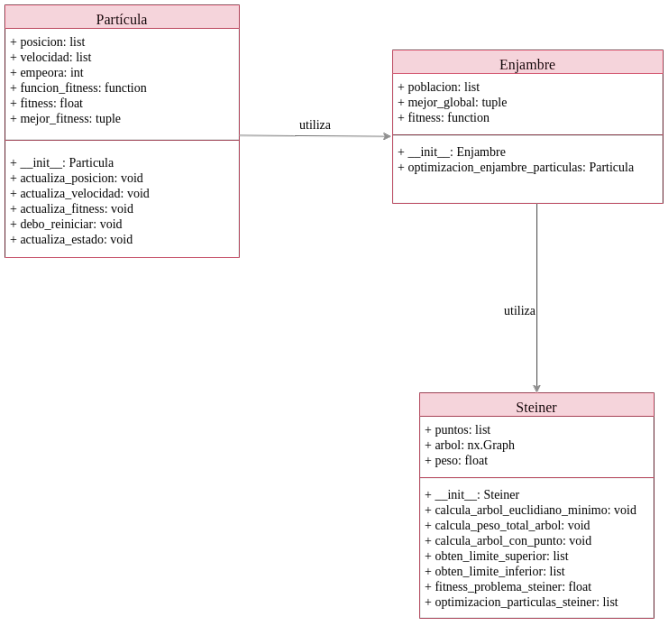


Figura 4.1: Diagrama UML de las clases principales.

4.1. Clases y módulos

Para la creación del programa se separaron las responsabilidades en:

- Clase partícula: La clase modela una partícula dentro de un enjambre, cuenta con las funciones necesarias para el modelado del ciclo de vida de una partícula.
- Clase Enjambre: Clase que modela un conjunto de partículas, ésta se encarga del manejo del líder del enjambre.
- Clase Steiner: La clase modela el problema del árbol de Steiner euclideano mínimo, maneja el cálculo del peso de los árboles Steiner.
- Módulo `util.py`: Módulo con funciones de ayuda para las clases.
- Pruebas unitarias: Con la finalidad de comprobar la correctitud del código se crearon diversas pruebas unitarias.

Además de las clases y módulos anteriores, se creó un módulo `main.py` el cual lee los archivos de entrada para el programa, los procesa utilizando las clases definidas para obtener un conjunto de puntos Steiner y regresa el conjunto de puntos resultantes, así como las imágenes de los árboles obtenidos.

4.1.1. Clase partícula

La clase `particula.py` modela una partícula dentro de un enjambre, se encarga de la creación de nuevas partículas, actualización de velocidad, posición y aptitud, así como el reinicio de la partícula en caso de ser necesario, el diagrama UML de la clase se puede ver en la Figura 4.2.

Atributos

Los atributos con los que cuenta la clase son los siguientes:

- `posición`: Tupla con la posición donde se encuentra la partícula, cada elemento x_i dentro de la tupla representa la posición de la partícula en la dimensión i.
- `velocidad`: Lista con la velocidad de la partícula, cada elemento x_i dentro de la tupla representa la velocidad de la partícula en la dimensión i.
- `empeora`: Veces que ha empeorado la aptitud de la partícula.
- `función_fitness`: Función que evalúa la aptitud de una partícula.
- `fitness`: Valor de la aptitud de la partícula.
- `mejor_fitness`: Tupla con la posición y aptitud donde se encontró la mejor aptitud dentro del ciclo de vida de la partícula.

Métodos

La clase cuenta con los siguientes métodos.

- Constructor:

  El constructor solicita una `posicion_inicial` y una `funcion_fitness`, a partir de la `posicion_inicial` se obtiene la dimensión del plano donde la partícula se moverá y con estos datos se inicializan los valores de posición, velocidad, veces en las que la aptitud ha empeorado, la aptitud inicial y la mejor aptitud dentro del ciclo de vida de la partícula.

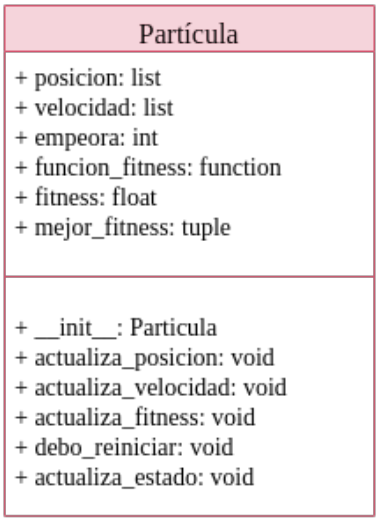


Figura 4.2: Diagrama UML de la clase partícula.

Para obtener la velocidad inicial, por cada dimensión se obtiene un valor aleatorio entre 0 y 1. La primera posición se calcula de la siguiente manera:

$$X_{t_0}[i] = X_{inicial}[i] + random\,[0,1]\,. \tag{4.1}$$

- `actualiza_posicion`:

  Para calcular la posición se hace uso de lo siguiente:

  $$X_{t+1}[i] = X_t[i] + Vt + 1[i]. \tag{4.2}$$

  Para realizar el cálculo, el método `actualiza_posicion` recibe los valores `limite_superior` y `limite_inferior` los cuales son los límites del espacio de búsqueda, con estos valores actualiza la posición de la partícula utilizando el vector de velocidades, si la posición en la dimensión i (p_i) excede algún límite en la dimensión i, entonces p_i toma el valor del límite que haya excedido en i.

- `actualiza_velocidad`:

  El método recibe el parámetro `posicion_mejor_global` el cual es la posición donde el enjambre ha encontrado la mejor evaluación de la aptitud y con ello se calcula la velocidad de la siguiente manera:

  $$V_{t+1}[i] = w \cdot V_t[i] + c_1 \cdot r_1 \cdot (P_t[i] - X_t[i]) + c_2 \cdot r_2 \cdot (G_t[i] - X_t[i])\,. \tag{4.3}$$

  Donde:

  - w es la constante de inercia, dentro del programa se utilizó de manera arbitraria el valor 0.5.

- c_1 es la constante cognitiva, arbitrariamente se tomó el valor 1.25.
- c_2 es la constante social, dentro del programa tiene un valor de 1.75.
- r_1 y r_2 son valores aleatorios uniformes escogidos del 0 al 1.
- $P_{i,t}$ es la posición en la dimensión i donde la partícula ha tenido la mejor aptitud dentro de su ciclo de vida.
- $G_{i,t}$ es la posición con la mejor aptitud evaluada de todo el enjambre.
- $w \cdot V_{i,t}$ sirve como el término de momento para evitar el incremento descontrolado de la velocidad.
- $c_1 \cdot r_1 \cdot (P_{i,t} - X_{i,t})$ representa el componente cognitivo, es decir, la tendencia natural de los individuos a regresar al ambiente donde experimentaron una mejor experiencia.
- $c_2 \cdot r_2 \cdot (G_{i,t} - X_{i,t})$ representa el componente social, en otras palabras, la tendencia de los individuos a seguir al líder, o bien el éxito de sus compañeros.

Es importante mencionar que los valores para las constantes w, c_1 y c_2 se tomaron de forma arbitraria, para determinar los resultandos cambiando los valores de las constantes harían falta más experimentos.

- `actualiza_fitness`:

  El método actualiza la aptitud de una partícula utilizando su posición y la función de aptitud (*fitness*) que tiene guardada.

  Una vez evaluada la nueva aptitud se verifica lo siguiente:

  - Si la nueva aptitud es mejor que la aptitud anterior o se mantiene, `empeora` se reestablece en 0, hay que tomar en cuenta que cuando la partícula se crea este valor es 0.
  - Si la nueva aptitud es mejor que el valor almacenado en `mejor_fitness`, entonces el valor se actualiza guardando la nueva posición.
  - Si la nueva aptitud es peor que la evaluación anterior, el valor `empeora` incrementa en 1.

- `debo_reiniciar`:

  A las partículas se les permite empeorar su aptitud un máximo 20 veces consecutivas (valor arbitrario), esto con el fin de evitar que una partícula se quede detenida en valores no satisfactorios. Si esto ocurre se reinicia la partícula utilizando la posición actual. El propósito de esta estrategia es evitar que una partícula se quede estancada y poder agregar algo de diversidad al enjambre.

  Dentro de los valores reiniciados se encuentra la posición, velocidad, veces que ha empeorado, evaluación actual y mejor evaluación dentro de su ciclo de vida.

- `actualiza_estado`:

  El método recibe todos los parámetros necesarios para poder actualizar el estado en una partícula en la siguiente generación:

  - `limite_inferior`

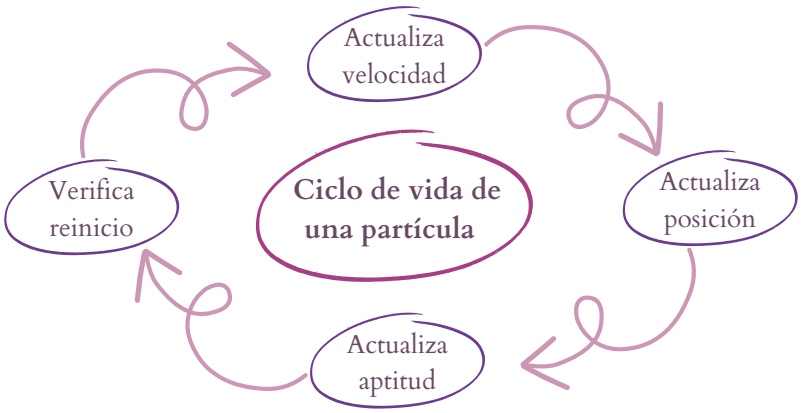


Figura 4.3: Ciclo de vida de una partícula dentro de un enjambre.

- `limite_superior`
- `posicion_mejor_global`

El código de funcionamiento consiste en la llamada de las funciones anteriormente descritas; en otras palabras, esta función se encarga de simular una iteración del ciclo de vida de una partícula, el ciclo de vida puede verse en la Figura 4.3 donde se actualiza la velocidad, se actualiza la posición, la aptitud, se verifica si se debe reiniciar la partícula y se repite el proceso.

4.1.2. Clase enjambre

La clase `enjambre.py` modela un enjambre (conjunto de partículas), se encarga de la creación del enjambre, generación de las partículas, así como del control de asignación del líder del enjambre (partícula con mejor aptitud durante la iteración). Dentro de esta clase se lleva a cabo la ejecución del algoritmo de *optimización por enjambre de partículas* de una manera genérica, es decir, dada una función de aptitud a optimizar, el algoritmo optimiza esa función, el diagrama UML de la clase puede encontrarse en la Figura 4.4.

Atributos

Los atributos con los que cuenta la clase son los siguientes:

- `poblacion`: Lista de partículas dentro del enjambre.
- `mejor_global`: Partícula con la mejor aptitud en el enjambre durante todas las iteraciones.
- `fitness`: Función que evalúa la aptitud de una partícula.

Métodos

A continuación describimos los métodos de la clase.

- Constructor:

  Con los siguientes parámetros el constructor crea todas las partículas del enjambre, desde la iteración inicial se obtiene el agente con la mejor aptitud de esa iteración.

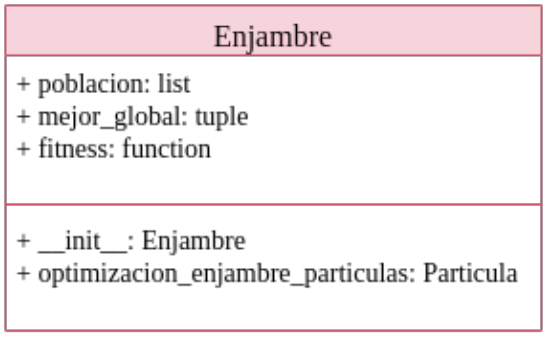


Figura 4.4: Diagrama UML de la clase enjambre.

- `tam_poblacion`: Número entero que representa la cantidad de partículas dentro del enjambre.
- `posicion_inicial`: Lista con la posición donde se inicializarán las partículas, esta posición es utilizada para inicializar todas las partículas.
- `funcion_fitness`: Función que se quiere optimizar con el enjambre.

- `optimizacion_enjambre_particulas`:

  El método recibe los siguientes parámetros:

  - `limite_inferior`: Lista con las coordenadas del límite inferior del problema.
  - `limite_superior`: Lista con las coordenadas del límite superior del problema.
  - `cantidad_iteraciones`: Número entero con la cantidad de iteraciones máximas que se realizarán.

  El método realiza una optimización por enjambre de partículas para la función que tiene guardada en el atributo `fitness`, para esto realiza a lo más n iteraciones, donde n es `cantidad_iteraciones`.

  En cada iteración asigna al líder del enjambre y posteriormente actualiza el estado de todas las partículas utilizando `limite_inferior` y `limite_superior`. Una vez terminada la última iteración, el método regresa la partícula con la mejor evaluación de todo el enjambre.

  Para evitar seguir iterando un enjambre cuando este ya no encuentra mejores valores se agregó un criterio de paro, el cual es, si durante 35 iteraciones consecutivas no se encuentra un mejor valor que el que tiene el líder, entonces se regresa el líder actual.

  Un detalle a tener en cuenta dentro de este algoritmo es que fue necesario utilizar la función `copy.copy()` de la biblioteca `copy` para evitar el paso por referencia de Python al cambiar el valor de las variables que guardaban la partícula con mejor evaluación actual y la partícula con mejor evaluación en la iteración anterior, los detalles de esta parte de la implementación se pueden encontrar en la sección 4.2.

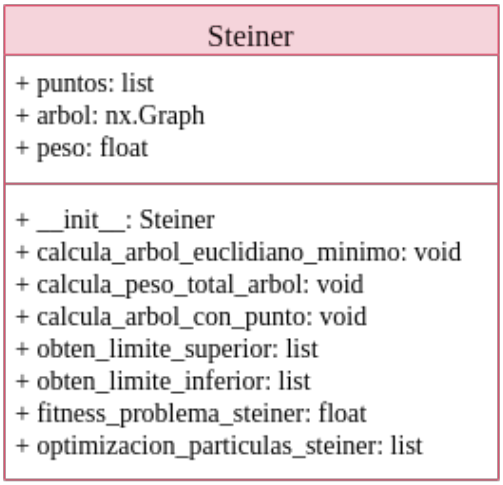


Figura 4.5: Diagrama UML de la clase Steiner.

4.1.3. Clase Steiner

La clase `steiner.py` modela el problema del arbol de steiner euclideano mínimo (*ESTP*), se encarga de a partir de los puntos del problema obtener los árboles de peso mínimo, peso de los árboles y calcular un nuevo arbol añadiendo puntos a la gráfica. En esta clase es donde se define la función de aptitud a optimizar con el algoritmo de optimización por enjambre de partículas, en la Figura 4.5 puede verse el diagrama UML de la clase.

Atributos

Los atributos con los que cuenta la clase son los siguientes:

- `puntos`: Lista de puntos pertenecientes al conjunto inicial del problema.
- `arbol`: Lista de aristas pertenecientes al árbol euclideano de peso mínimo.
- `peso`: Valor numérico del peso total del árbol.

Métodos

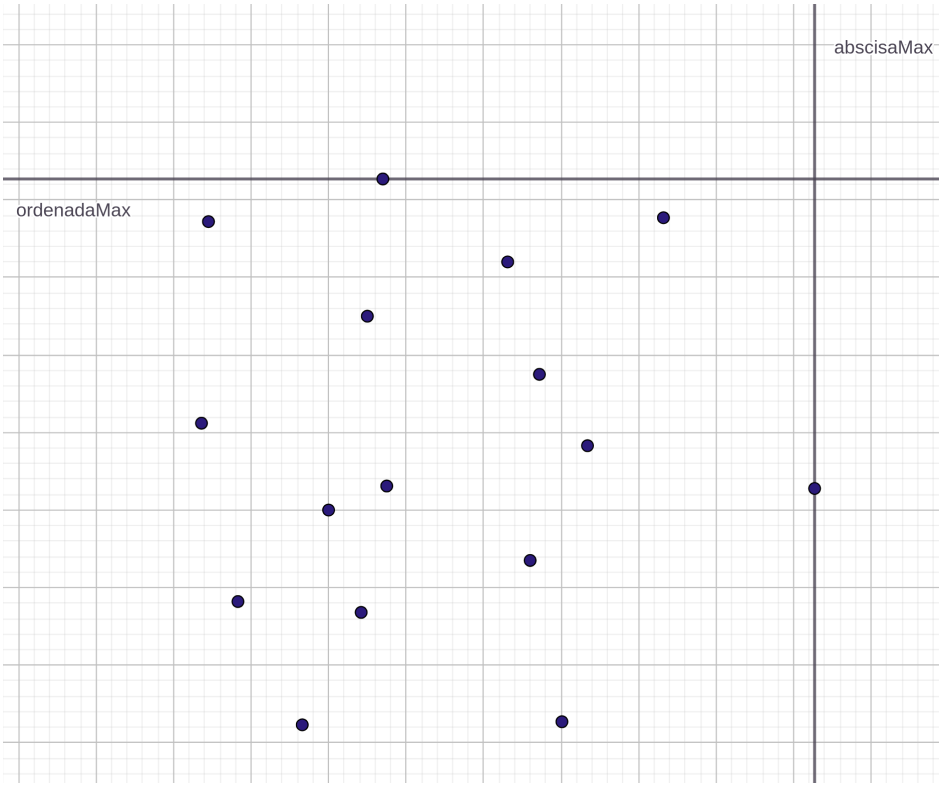
La clase cuenta con los siguientes métodos para modelar los árboles, función de aptitud y ejecución de la heurística.

- Constructor:

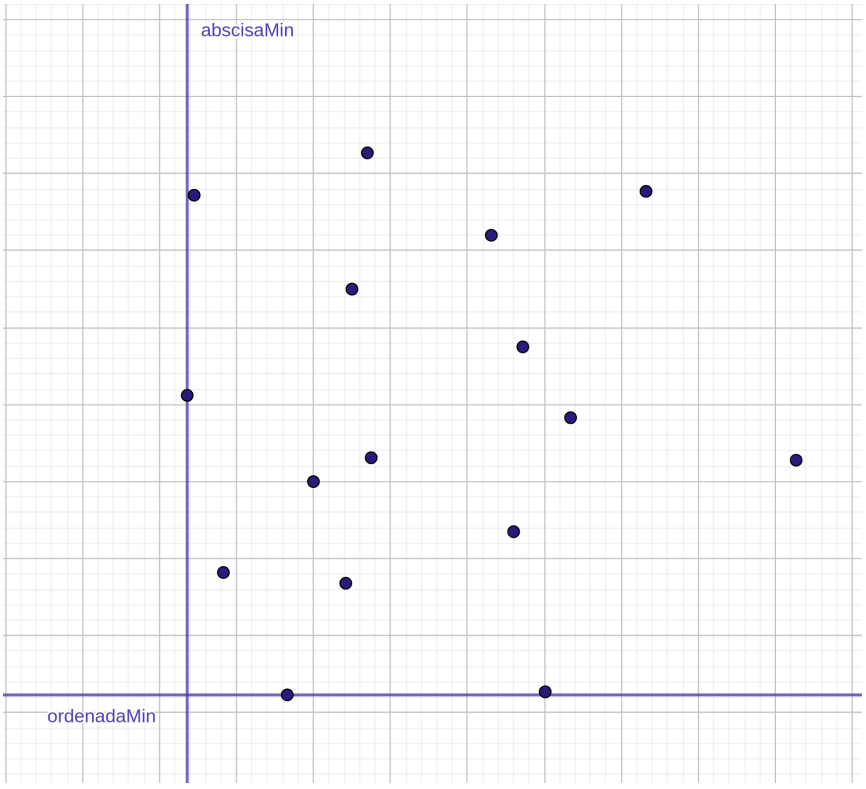
  El constructor solicita un conjunto de puntos, los cuales serán considerados como los puntos del problema. Utilizando estos datos el constructor instancia el atributo `puntos` con el valor del parámetro recibido y los valores de `peso` y `arbol` con `None`.

- `calcula_arbol_euclidiano_minimo`:

  El método calcula el árbol euclideano mínimo de la gráfica con los puntos almacenados en el atributo `puntos`, para lograr esto se utiliza la función `minimum_spanning_tree` para calcular árboles euclideanos mínimos de la biblioteca `NetworkX` [63], la cual internamente utiliza el algoritmo de Kruskal.



(a) Obtención de límites superiores.



(b) Obtención de límites inferiores.

Figura 4.6: Obtención de los límites del espacio de búsqueda.

Debido a que la función recibe una gráfica con aristas sobre la cual calcula el árbol, es necesario ingresar como parámetro una gráfica completa. Por lo tanto, la función `calcula_arbol_euclidiano_minimo` primero calcula la gráfica completa con los puntos originales como vértices, agrega todas las aristas y posteriormente calcula el árbol mínimo utilizando como peso de las aristas la distancia entre los puntos. Al final del método el árbol resultante es asignado como valor del atributo `arbol`.

- `calcula_peso_total_arbol`:

  El método calcula el peso total del árbol guardado en el atributo `arbol` utilizando la función auxiliar `calcula_peso_total_grafica` definida en `util.py`, este resultado es posteriormente guardado en el atributo `peso`.

- `calcula_arbol_con_punto`:

  El método recibe como parámetro un punto —`punto`— y calcula el árbol de peso mínimo de los puntos originales agregando al conjunto original el punto recibido como parámetro, una vez calculado regresa el nuevo árbol obtenido.

- `obten_limite_superior`:

  Método que calcula las coordenadas máximas del conjunto de puntos original, el método regresa una lista con una ordenada y abscisa. Para hacer el cálculo se toman las coordenadas máximas del conjunto, el cálculo se puede observar en la Figura 4.6a.

- `obten_limite_inferior`:

  Método que calcula las coordenadas mínimas del conjunto de puntos original, el método regresa una lista con el valor de la abscisa y ordenada mínimas, el cálculo se puede observar en la Figura 4.6b.

- `fitness_problema_steiner`:

  Función utilizada para evaluar la aptitud de las partículas dentro del algoritmo de optimización por enjambre de partículas, el método recibe como parámetro un nuevo punto p_i, calcula el árbol euclideano mínimo a partir del conjunto original de puntos y p_i, posteriormente devuelve el peso del árbol obtenido el cual, dentro del sistema, corresponde a la aptitud de la partícula con posición p_i, en la Figura 4.7.

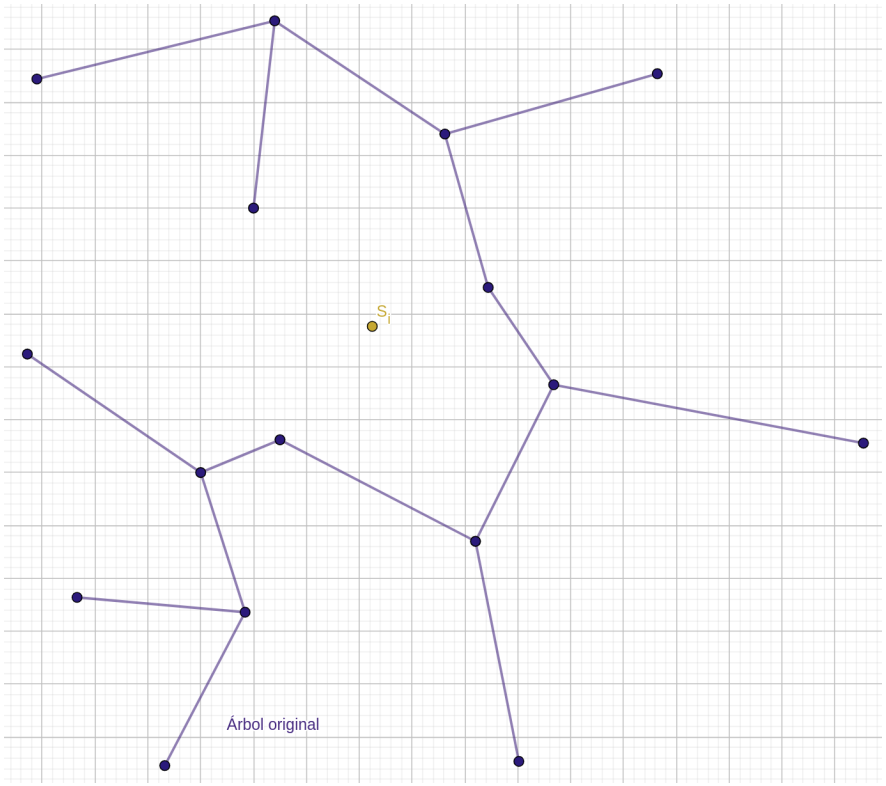
- `optimizacion_particulas_steiner`:

  El método recibe los siguientes parámetros:

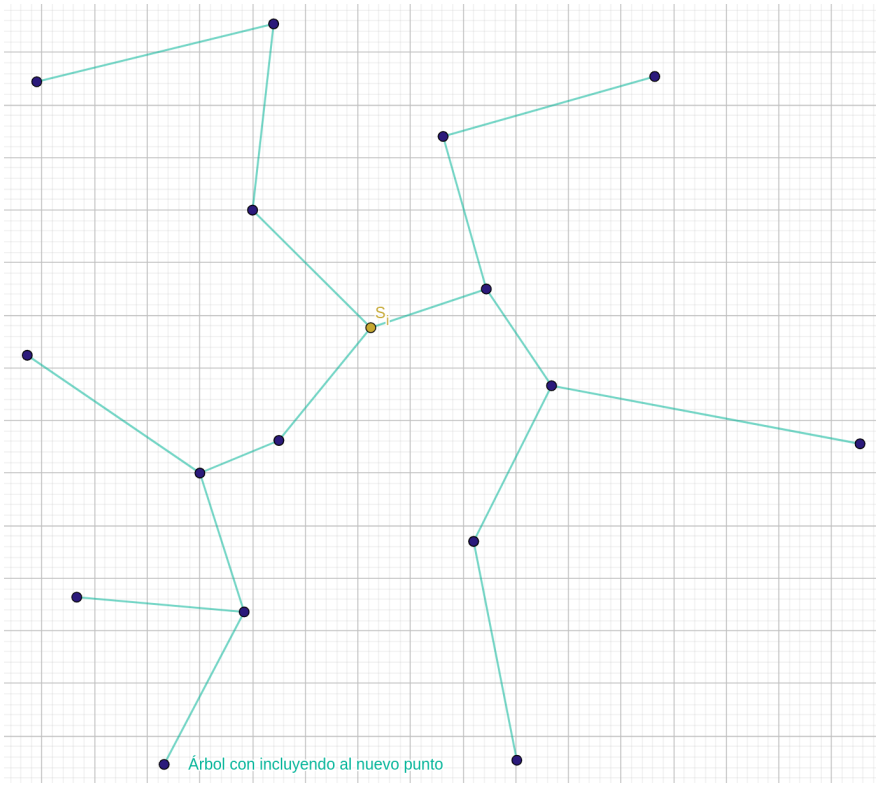
  - `iteracion_max`: Iteraciones máximas deseadas para cada enjambre.
  - `cantidad_enjambres`: Cantidad de enjambres que se crearán para ejecutarlos de uno en uno.
  - `tam_poblacion`: Tamaño de población para los enjambres, todos los enjambres tendrán el mismo tamaño.
  - `no_max_puntos`: Número máximo de puntos Steiner a encontrar, por omisión no hay una cantidad máxima.

  Con los parámetros anteriores el método realiza lo siguiente:

  1. Calcula los límites superiores e inferiores del problema utilizando los puntos que tiene en el atributo `puntos`, el cálculo puede verse más claramente en la figura 4.8.



(a) Árbol original.



(b) Árbol con el nuevo punto agregado.

Figura 4.7: Cálculo de la función de aptitud para una partícula dada su posición.

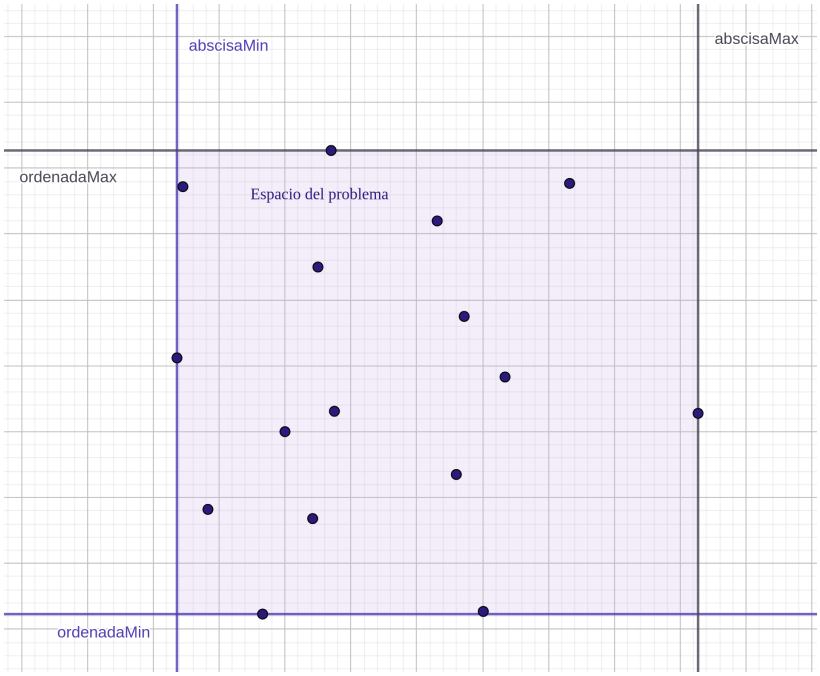


Figura 4.8: Espacio del problema sobre el que los enjambres buscan.

2. Inicializa enjambres de uno en uno mientras la cantidad de puntos Steiner agregados sea menor a `no_max_puntos` y la cantidad de enjambres creados sea menor a `cantidad_enjambres`
3. Para generar cada enjambre, obtiene las coordenadas iniciales del enjambre de manera aleatoria utilizando los límites previamente calculados. Posteriormente ejecuta el enjambre utilizando los límites, las coordenadas y el parámetro `iteracion_max` para obtener un punto de Steiner.
4. Si el punto mejora estrictamente el peso del árbol, entonces lo agrega al conjunto de puntos del problema y ejecuta el siguiente enjambre utilizando este nuevo punto. Si no mejora el peso, entonces el punto se descarta.
5. Una vez se termina de encontrar los puntos solicitados o se han ejecutado todos los enjambres, el método regresa la lista de puntos originales, puntos Steiner encontrado y el peso del árbol con los puntos Steiner.

4.1.4. Módulo `util.py` y pruebas unitarias

Para apoyar al funcionamiento de las clases y comprobar el correcto funcionamiento del código, dentro del programa también se encuentra el módulo `util.py` con funciones de ayuda y un conjunto de pruebas unitarias.

Módulo `util.py`

Dentro del módulo `util.py` se codificaron las siguientes funciones:

- `distancia_entre_dos_puntos`: Calcula la distancia entre dos puntos con dimensión n.

- `calcula_peso_total_grafica`: Dada una gráfica con aristas, calcula el peso total de la misma.

El módulo también cuenta con pruebas unitarias para verificar que las funciones sean correctas.

Pruebas unitarias

Para cada clase y funciones auxiliares se crearon clases con pruebas unitarias utilizando la biblioteca `unittest` [75]:

`utilTest`:

La clase cuenta con tres pruebas para verificar el funcionamiento de las funciones auxiliares:

- `test_distancia_dos_puntos_caso_base`: La prueba verifica el caso base de distancia entre dos puntos, el cual es que los puntos sean el mismo.
- `test_distancia_dos_puntos`: La prueba verifica un caso general para dos puntos de tres dimensiones.
- `test_calcula_peso_total_grafica`: La prueba verifica que dada una gráfica con aristas, el peso de la gráfica (suma de todas las aristas) sea correcto.

`particulaTest`:

La clase cuenta con una única prueba `test_constructor_particula` para probar que el constructor sea correcto, dentro de la misma se comprueban los siguientes hechos:

- Que al instanciar una partícula, el vector de velocidades tenga en cada posición un valor entre 0 y 1.
- Que la posición inicial es igual a la posición guardada en el atributo `mejor_fitness`.
- Que la cantidad de veces que la partícula ha empeorado es 0.
- Que el atributo `mejor_fitness` guarda correctamente el valor de la evaluación de la función de aptitud.

`enjambreTest`:

La clase cuenta con una única prueba `test_constructor_enjambre` para probar el constructor de la clase `Enjambre`, dentro de la prueba se verifica lo siguiente:

- La cantidad de partículas en el enjambre es igual al número ingresado como parámetro.
- Que para cada ente en el atributo `poblacion`, pertenece a la clase `Particula`.
- Que la función guardada en el atributo `fitness` es la ingresada como parámetro.

`steinerTest`:

La clase cuenta con varias pruebas para verificar distintas secciones de la funcionalidad de la clase `Steiner`.

- `test_constructor_steiner`:

  La prueba verifica que el constructor guarde correctamente todos los puntos y que los atributos `arbol` y `peso` sean instanciados como `None`.

- `test_calcula_arbol`:

  La prueba verifica que dado un objeto de la clase `Steiner`, el método `calcula_arbol_euclidiano_minimo` regrese una gráfica con una lista de aristas correspondientes a las de un árbol.

- `test_peso_total`:

  La prueba verifica que dado un arbol obtenido a partir de una instancia de la clase `Steiner`, se calcula correctamente el peso del árbol.

- `test_calcula_arbol_con_punto`:

  La prueba verifica que dado un árbol obtenido a partir de una instancia de la clase `Steiner`, al calcular un nuevo árbol agregando un punto, éste incluya una arista extra y el peso sea el correcto.

- `test_limite_superior`:

  La prueba verifica que dada una instancia de la clase `Steiner`, al obtener el límite superior del espacio de búsqueda, las coordenadas seas las correctas.

- `test_limite_inferior`:

  La prueba verifica que dada una instancia de la clase `Steiner`, al obtener el límite inferior del espacio de búsqueda, las coordenadas seas las correctas.

- `test_optimizacion_steiner`:

  La prueba verifica que dada una instancia de la clase `Steiner`, calcular su árbol y peso orginal y posteriormente hacer una ejecución de optimización por enjambre de partículas se obtienen correctamente una lista con los puntos originales, los puntos Steiner obtenidos y el peso resultante. Además de esto verifica que el peso resultante es menor o igual al peso del árbol original.

4.2. Entrada y salida del programa

El programa cuenta con la clase `main.py` la cual se encarga de la administración de la ejecución del algoritmo y la graficación de los árboles obtenidos. Como entrada y salida utiliza archivos en formato `JSON`.

`JSON` (*JavaScript Object Notation* o Notación de Objetos de JavaScript) es un formato ligero de intercambio de datos, está basado en un subconjunto del Lenguaje de Programación JavaScript. `JSON` es un formato de texto completamente independiente del lenguaje pero utiliza convenciones que son ampliamente conocidas haciendo que `JSON` sea un lenguaje ideal para el intercambio de datos [40]. Dada la cantidad de datos necesarios para ejecutar el algoritmo se optó por utilizar el formato `JSON`.

Los archivos de entrada necesitan las siguientes llaves:

- `puntos_originales`: Lista con los puntos originales del problema.
- `iteración_max`: Iteración máxima que ejecutará el algoritmo de optimización por cada enjambre creado.
- `cantidad_enjambres`: Cantidad máxima de enjambres que se desean crear.

- `tam_poblacion`: Cantidad de partículas con las que contará cada enjambre.
- `ejecuciones`: Cantidad de ejecuciones que se harán del método `steiner.optimizacion_particulas_steiner`, cada ejecución es independiente de la anterior.

Opcionalmente también se puede contar con los siguientes parámetros:

- `puntos_encontrados`: Puntos Steiner encontrados en la bibliografía donde se describe el ejemplar.
- `peso_steiner`: Peso obtenido usando los puntos steiner `puntos_encontrados` sobre los puntos del problema original.
- `peso_original`: Peso del árbol original (sin puntos Steiner).

Estos parámetros opcionales son utilizados para la graficación del arbol Steiner encontrado en la bibliografía donde se define el ejemplar de donde se obtuvieron los datos.

Para la salida del programa se cuenta con dos opciones, una es la gráfica obtenida a partir de los puntos pasados como parámetro, la otra es un archivo `JSON` con los puntos Steiner encontrados y su peso. El archivo `JSON` de salida cuenta con las siguientes llaves:

- `peso_original`: Peso original del árbol sin puntos Steiner.
- `peso_mejorado`: Peso correspondiente al árbol original y los puntos Steiner agregados.
- `porcentaje_mejora`: Porcentaje de mejora del nuevo peso con respecto al peso del árbol original.
- `puntos_encontrados`: Lista con los puntos Steiner encontrados por el programa en su mejor ejecución.

4.3. Bibliotecas auxiliares utilizadas

En el código se utilizaron diversas bibliotecas auxiliares con distintos propósitos.

- `NetworkX`

  La biblioteca `NetworkX` [63] es un paquete de Python para la creación, manipulación y visualización de gráficas. Entre sus funcionalidades se encuentran las siguientes:

  - Estructuras de datos para gráficas, digráficas y multigráficas
  - Implementación de diversos algoritmos de gráficas como lo son Kruskal, cálculo de ciclos eulerianos, generación de gráficas aleatorias, entre otras.
  - Integración con la biblioteca `matplotlib` para la visualización de gráficas.

  Dentro del programa se utilizó para la creación, manipulación, cálculo de árboles de peso mínimo y para la visualización de gráficas.

- `random`

  La biblioteca `random` [71] cuenta con generadores de números pseudoaleatorios usando varias distribuciones.

Entre los generadores se encuentran generadores de enteros con distribución uniforme, generadores de secuencias, generadores para números de punto flotante con las distribuciones de Gauss, Gamma, uniforme, entre otras.

En la implementación fue utilizada para la obtención de números con distribución uniforme en la creación de coordenadas.

- `numpy`

  La biblioteca `numpy` [65] es la principal biblioteca utilizada para el cómputo científico en Python, la biblioteca cuenta con diversas utilidades entre las cuales están:

  - Manejo de arreglos multidimensionales.
  - Objetos derivados de arreglos como son las matrices.
  - Funciones matemáticas como son transformaciones de Furier, álgebra lineal básica, operaciones estadísticas básicas, entre otras.
  - Generadores de números pseudoaleatorios.

  En la implementación se utilizó para la obtención de números pseudoaleatorios con distribución normal para el cálculo de la velocidad de una partícula.

- `unittest`

  La biblioteca `unittest` [75] es un marco para hacer pruebas inspirado en JUnit. Permite la ejecución de pruebas automáticas, agregación de pruebas en colecciones entre otras características.

  Dentro del programa se utilizó la biblioteca para hacer todas las pruebas unitarias necesarias así como utilizar sus funciones auxiliares (`assertEqual`, `assertLessEqual`, `assertIsInstance`, `assertGreaterEqual`, etcétera) para establecer los predicados.

- `math`

  La biblioteca `math` [57] provee acceso a funciones matemáticas definidas en el lenguaje de programación `C`. Entre estas funciones se encuentran:

  - `ceil`
  - `factorial`
  - `isfinite`
  - `nextafter`
  - `trunc`

  Dentro del programa se hizo uso de la función `math.sqrt` y de la variable `math.inf`.

- `matplotlib`

  La biblioteca `matplotlib` [58] es la principal biblioteca para la creación de visualizaciones estáticas o animadas en `Python`. Entre sus funcionalidades se encuentran:

  - Creación de gráficas.
  - Creación de figuras interactivas donde es posible cambiar la escala de la vista, actualizar, etcétera.

- Personalizar características visuales de la gráfica.
- Exportar figuras en diversos formatos.

En el programa se utilizó la biblioteca para modificar parámetros de personalización de las gráficas de los árboles generados por el programa, entre estos parámetros se encuentran la leyenda de la gráfica, cambio de la fuente del título y el posicionamiento de las gráficas.

- `json`

  La biblioteca `json` [41] consta de una API familiar para los usuarios al hacer uso de archivos en formato `JSON`, entre sus utilidades se encuentran:

  - Decodifica archivos en formato `JSON`.
  - Codifica datos a formato `JSON`.
  - Imprime con un formato legible datos provenientes de un `JSON` (*Pretty printing*).

  En el programa fue utilizada para lectura y escritura de archivos `JSON`, ya que los ejemplares ejecutados por el programa se codificaron en ese formato.

- `copy`

  Las declaraciones de asignación en Python no copian objetos, si no que se utiliza un paso por referencia. Para colecciones que son mutables o contienen objetos mutables, si se desea poder cambiar una copia sin cambiar la otra, es necesario crear una copia del objeto original y asignar esta a la nueva variable.

  La biblioteca `copy` [13] proporciona operaciones genéricas de copia superficial (`shallow`) y profunda (`deep`).

  - `copy`: La función regresa una copia superficial del objeto recibido, esta copia construye un nuevo objeto compuesto y posteriormente (en la medida de lo posible) inserta referencias a los objetos que se encuentran en el original.
  - `deepcopy`: La función regresa una copia profunda del objeto recibido, esta copia construye un nuevo objeto compuesto y posteriormente, recursivamente, inserta copias en el de los objetos encontrados en el original.

  En el programa fue utilizado para guardar correctamente referencias a objetos como los mejores valores encontrados o los anteriormente encontrados.

```
# Usando la asignacion usual
anterior_mejor_global = self.mejor_global
# Esta llamada hace que ahora anterior_mejor_global
# tenga el valor de nuevo_mejor_global
self.mejor_global = nuevo_mejor_global

# Usando copy
anterior_mejor_global = copy.copy(self.mejor_global)
# En esta llamada anterior_mejor_global y
# self.mejor_global cuentan con valores distintos,
# la asignacion no modifica el valor de
# anterior_mejor_global
```

```
13 self.mejor_global = nuevo_mejor_global
```

Código 4.1: Ejemplo de uso de la biblioteca `copy`.

Capítulo 5

Resultados

El programa para cálculo de puntos de Steiner fue probado con distintos ejemplares del problema propuestos en diversas fuentes ([54, 16, 30]). Para cada ejemplar se obtuvo un conjunto de puntos Steiner y sus gráficas correspondientes, los puntos Steiner aquí descritos son los que obtuvieron los mejores resultados después de varias iteraciones del programa.

5.1. Proceso de recopilación de datos

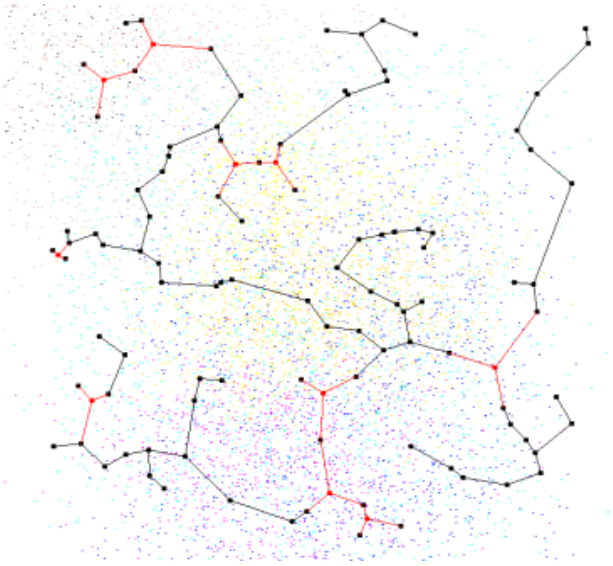
Debido a que en algunos casos los trabajos académicos utilizados solo documentan las gráficas originales y las gráficas obtenidas con los puntos Steiner de forma únicamente visual, fue necesario hacer una interpolación de los datos para lo cual fue utilizado el software de *Geogebra* [25].

Geogebra es un software matemático dinámico de código abierto y libre que reúne geometría, álgebra, hojas de cálculo, gráficas, estadística y cálculo en un solo motor. Es ampliamente utilizado por estudiantes y científicos por herramientas como:

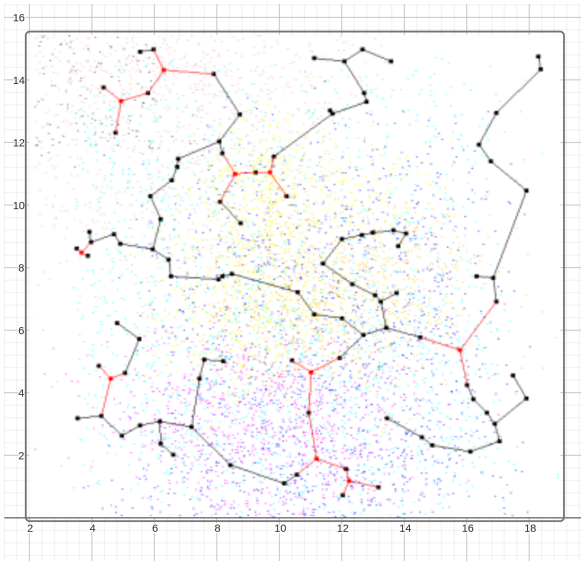
- Calculadora gráfica.
- Calculadora 3D.
- Calculadora científica.
- Calculadora CAS (*Computer Algebra System*, sistema de álgebra computacional).

Para realizar el proceso se realizaron los siguientes pasos:

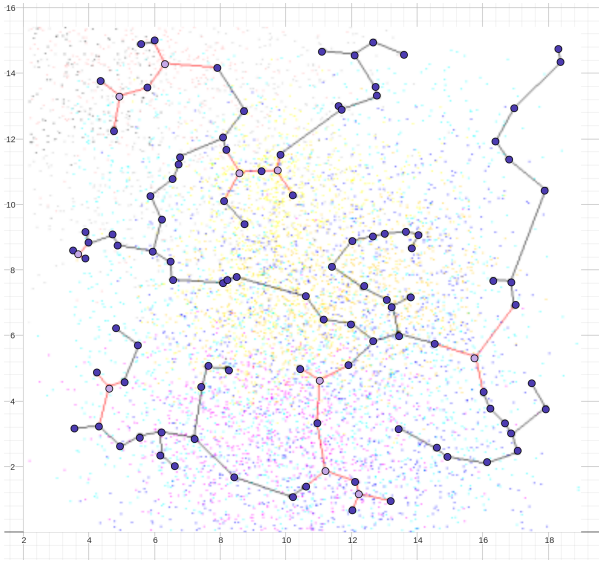
1. Se extrajo la imagen correspondiente a los árboles (veáse Figura 5.1a).
2. Se importó el archivo a Geogebra (visible en la Figura 5.1b).
3. Una vez contando con la imagen como fondo, se agregaron puntos a la imagen con la herramienta `Punto` sobre los puntos marcados en la imagen, al hacer esto se diferenciaron entre los puntos Steiner y los puntos originales (veáse Figura 5.1c).
4. Para cada punto agregado, se recopilaron las coordenadas para utilizarlas como parámetro del programa.



(a) Gráfica original mostrada en bibliografía [16].



(b) Imagen importada en *Geogebra*.



(c) Imagen importada con los puntos agregados.

Figura 5.1: Obtención de los puntos a partir de la gráfica documentada en bibliografía.

5. En el caso de los puntos obtenidos para el ejemplo descrito en la sección 5.4, además de este proceso, las coordenadas fueron pasadas a números enteros recorriendo el punto decimal y truncando las últimas décimas.

Es importante tomar en cuenta que este proceso no es exacto, por lo que los resultados descritos en las secciones 5.3 y 5.4 deben tomarse con reserva, en el sentido de que no se usaron exactamente los mismos datos y el método de interpolación deja espacio a inexactitudes, no obstante, el proceso nos otorga una noción de la realidad.

5.2. Ejemplo de optimización 1

El primer ejemplo fue obtenido del libro *Handbook of Combinatorial Optimization* [30], éste contaba con la definición de los puntos utilizados, por lo que no fue necesario el proceso de interpolación descrito en la sección 5.1 para obtener los puntos originales (Figura 5.2a), pero sí fue necesario para obtener los puntos Steiner que se pueden ver en la Figura 5.2b.

Estos puntos nos dan los siguientes resultados:

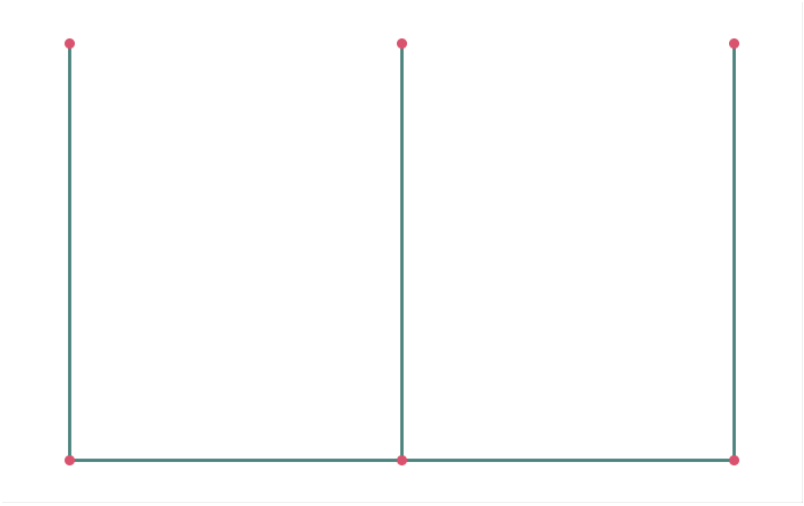
- Peso original: 7.0.
- Peso con puntos Steiner agregados: 6.62062814049094.

Para la ejecución del programa se utilizaron de manera arbitraria los siguientes parámetros:

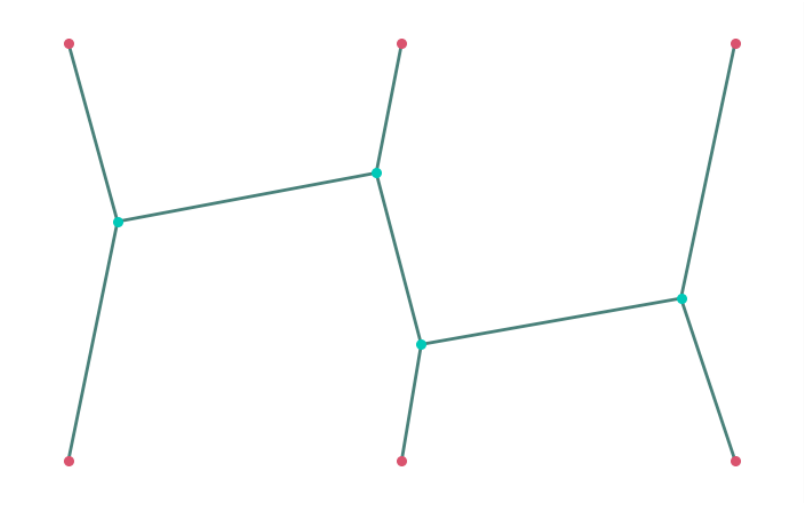
- Iteraciones máximas (`iteración_max`): 50.
- Cantidad de enjambres (`cantidad_enjambres`): 12.
- Tamaño de población (`tam_poblacion`): 80.
- Cantidad de ejecuciones (`ejecuciones`): 50.

La mejor ejecución obtuvo un peso mínimo de **6.621679219281745** y un porcentaje de mejora de **5.40 %** con respecto al peso del árbol original.

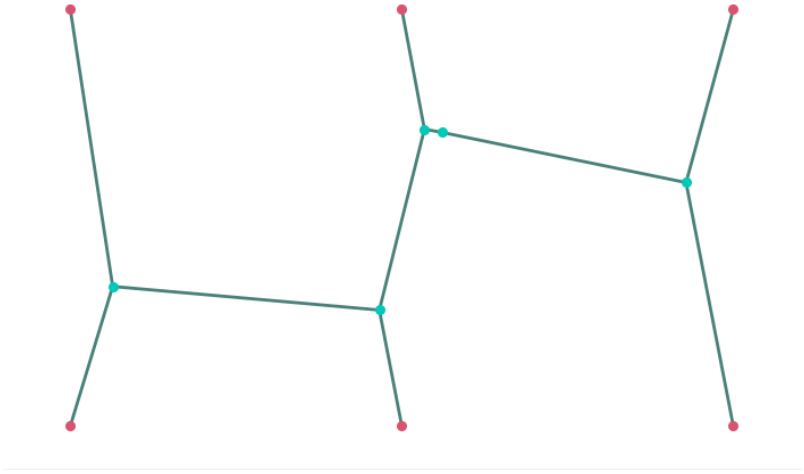
El árbol obtenido se puede visualizar en la Figura 5.2c. Como se puede percibir, si invertimos horizontalmente la gráfica obtenida por el programa, la única diferencia es un punto Steiner adicional, sin embargo éste punto es intrascendente, ya que al encontrarse dentro de una de las aristas, éste no afecta de ninguna manera el peso resultante.



(a) Árbol original.



(b) Árbol Steiner obtenido con la interpolación de puntos.



(c) Árbol obtenido con los puntos originales y los puntos Steiner encontrados por el programa.

Figura 5.2: Resultados obtenidos a partir del ejemplo descrito en *Handbook of Combinatorial Optimization* [30].

5.3. Ejemplo de optimización 2

El segundo ejemplo fue obtenido del artículo *On the history of the Euclidean Steiner tree problem* [54], este ejemplo no contaba con la descripción de los puntos originales o los puntos Steiner, por lo que el proceso de interpolación tuvo que realizarse con ambos conjuntos, el árbol original puede verse en la Figura 5.3a y el árbol de Steiner en la Figura 5.3b.

Estos puntos nos dan los siguientes resultados:

- Peso original: 145.31619976710812.
- Peso con puntos Steiner agregados: 141.85556141735944.

Para la ejecución del programa se utilizaron los siguientes parámetros de manera arbitraria:

- Iteraciones máximas (`iteración_max`): 50.
- Cantidad de enjambres (`cantidad_enjambres`): 15.
- Tamaño de población (`tam_poblacion`): 150.
- Cantidad de ejecuciones (`ejecuciones`): 150.

La mejor ejecución obtuvo un peso mínimo de **141.9124195765307** lo cual corresponde a un porcentaje de mejora de **2.34 %** con respecto al peso del árbol original.

- Peso mínimo encontrado: .

La gráfica del árbol obtenido por el programa puede observarse en la Figura 5.3c, como se puede observar, las gráficas son casi iguales debido a que los puntos que hacen una diferencia significativa se encuentran en la misma área, sin embargo, un detalle a denotar, es que el programa agrega puntos que no mejoran significativamente el árbol resultante, esto se puede percibir debido a que algunos puntos Steiner se encuentran casi dentro de los segmentos que conforman las aristas del árbol.

5.4. Ejemplo de optimización 3

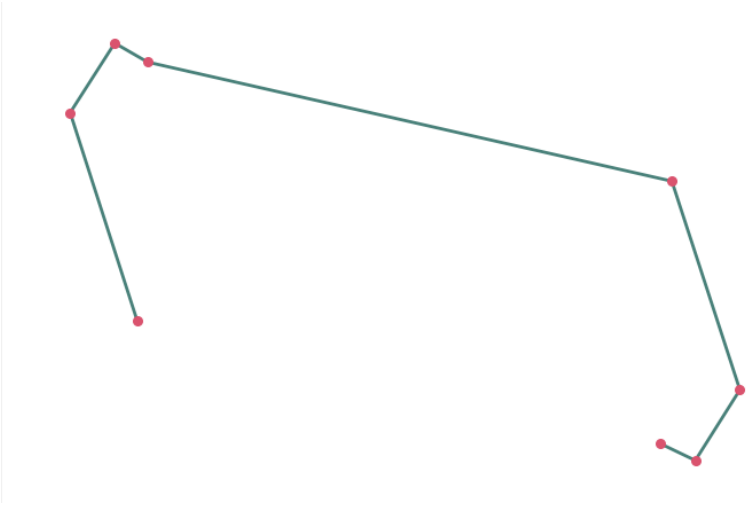
Para el tercer ejemplo se consultó el artículo *Solving Euclidean Steiner Tree Problems with Multi Swarm Optimization* [16], este ejemplo tampoco contaba con la descripción de los puntos originales o de los puntos Steiner, por lo que el proceso de interpolación se realizó en ambos casos, el árbol resultante se puede encontrar en la Figura 5.4. Además de esto, como fue mencionado en la sección 5.1, se modificaron las coordenadas resultantes para hacerlas enteras.

Estos puntos nos dan el árbol de la Figura 5.5a y los siguientes resultados:

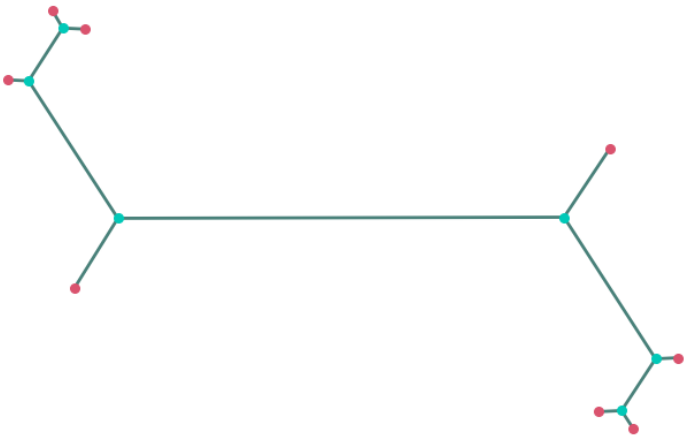
- Peso original: 210572.85375362588.
- Peso con puntos Steiner agregados: 206973.8455339465.

Para la ejecución del programa se utilizaron de manera arbitraria los siguientes parámetros:

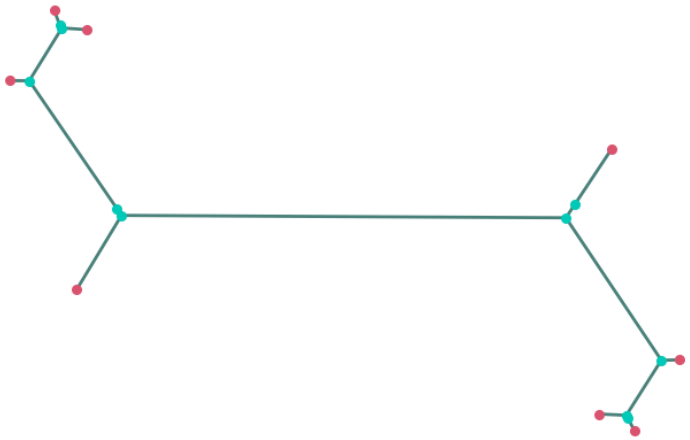
- Iteraciones máximas (`iteración_max`): 60.
- Cantidad de enjambres (`cantidad_enjambres`): 20.
- Tamaño de población (`tam_poblacion`): 70.
- Cantidad de ejecuciones (`ejecuciones`): 50.



(a) Arbol original obtenido con la interpolación de puntos.



(b) Árbol Steiner obtenido con la interpolación de puntos.



(c) Árbol obtenido con los puntos originales y los puntos Steiner encontrados por el programa.

Figura 5.3: Resultados obtenidos a partir del ejemplo descrito en *On the history of the Euclidean Steiner tree problem* [54].

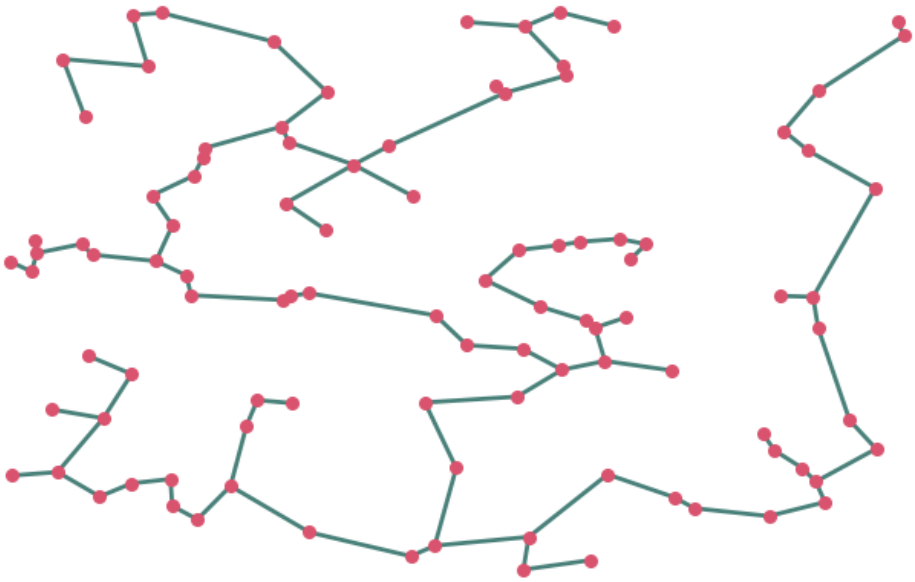


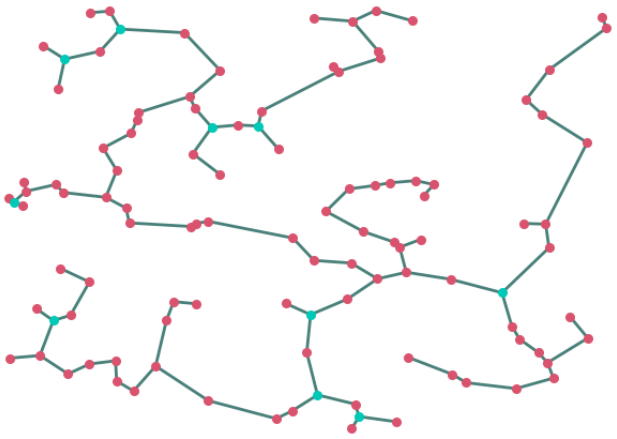
Figura 5.4: Árbol original descrito en *Solving Euclidean Steiner Tree Problems with Multi Swarm Optimization* [16].

Al ser un ejemplar más complejo se optó por hacer la experimentación dos veces:

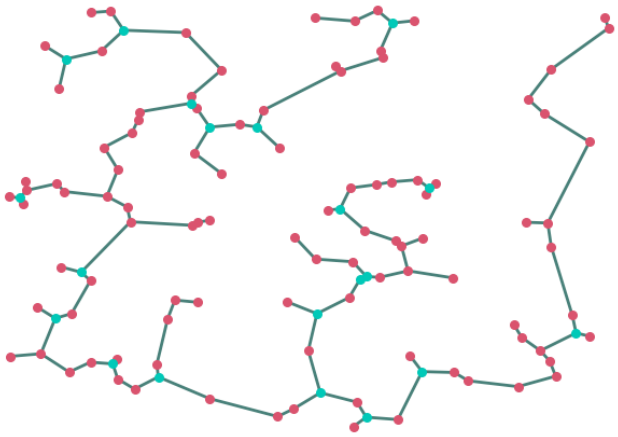
- La primera vez el programa encontró un total de 20 puntos, obteniendo así un peso mínimo de **205099.38677192564** y un porcentaje de mejora de **2.59 %** con respecto al peso del árbol original, el árbol resultante puede verse en la Figura 5.5b.
- La segunda vez se hizo la ejecución con la misma cantidad de puntos Steiner que fueron reportados en el artículo [16] (10 puntos), estos generan un árbol de meso mínimo de **206357.629407742** con un porcentaje de mejora de **2.00 %** con respecto al peso del árbol original, el árbol resultante se encuentra en la Figura 5.5c.

Como se puede observar en la Figura 5.5, los resultados son similares a los reportados en la bibliografía, en el caso de la misma cantidad de puntos el peso encontrado es mejor al de la bibliografía, sin embargo considerando la interpolación de puntos realizada, es posible que este resultado no sea del todo preciso.

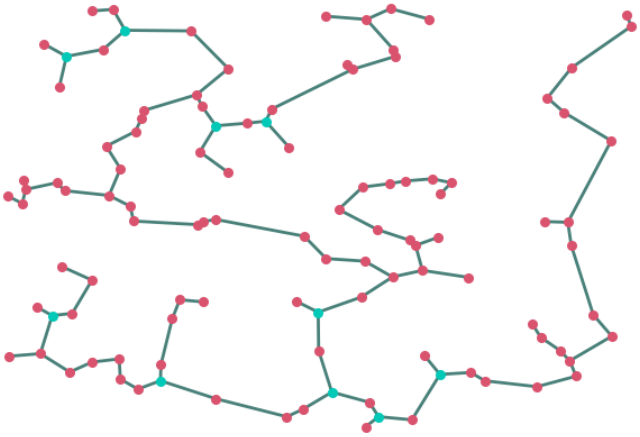
Asimismo, cuando se agregan más puntos al conjunto, el peso del árbol mejora, por lo que es posible que aún utilizando el conjunto de puntos original para hacer la comparación, los resultados del programa sean mejores a los reportados en la bibliografía.



(a) Árbol Steiner obtenido con la interpolación de puntos.



(b) Árbol obtenido con todos los puntos Steiner que el programa podía encontrar durante la ejecución.



(c) Árbol obtenido con la misma cantidad de puntos Steiner reportados en la bibliografía.

Figura 5.5: Resultados obtenidos a partir del ejemplo descrito en *Solving Euclidean Steiner Tree Problems with Multi Swarm Optimization* [16].

Conclusiones

Como se trató en el Capítulo 2, los problemas *NP*-duros son problemas en los que aún si la conjetura $P = NP$ fuera demostrada como válida, estos seguirían siendo de los problemas computables más difíciles de resolver.

Sin embargo, aún cuando teóricamente, los problemas *NP*-duros, no son posibles de resolver de manera exacta, en la práctica es posible obtener soluciones aceptables con la aplicación de heurísticas u algoritmo de optimización como algoritmos genéticos, recocido simulado, descenso por gradiente o la misma optimización por enjambre de partículas.

En cuanto al caso particular del problema del árbol de Steiner, encontré particularmente interesante como es posible optimizar el peso de un árbol generador de peso mínimo agregando puntos, en un inicio me pareció contraintuitivo, no obstante, al estudiar más el problema entendí el funcionamiento del mismo y el porqué geométricamente funciona esta estrategia para optimizar árboles euclidianos.

Otro punto que encontré atrayente fue leer sobre las aplicaciones, particularmente en problemas relacionados con árboles filogenéticos, considero interesante cómo ciencias que podrían considerarse lejanas de las Matemáticas como lo son la Biología o Lingüística buscan resolver un mismo problema.

Por otro lado, al ser un problema *NP*-duro en general no es posible encontrar la mejor solución o una solución aceptable en tiempo humanamente razonable usando algoritmos convencionales, pero es posible obtener soluciones aceptables que puedan optimizar el árbol mínimo euclideano original.

Además del funcionamiento del problema y sus características, de manera histórica y a través de mi investigación pude establecer el porqué actualmente a este problema se le denomina como *Problema del árbol de Steiner*, lo cual inicialmente (resumiendo la sección 1.1) fue debido a un error de Karl Bopp al equivocadamente atribuir el teorema de Viviani a Jakob Steiner. Al notar esto y trabajar con diversas referencias noté también que es común comenter errores al atribuir ciertos avances, teoremas o afimaciones a autores, así como cometer errores al escribir el nombre o año en el que se llevo a cabo dicho avance.

Por otro lado, durante el proceso de escritura del programa descrito en el Capítulo 4, pude reafirmar la importancia de tener un código ordenado, bien documentado y con nombres claros para las variables, así como el saber la diferencia entre paso por referencia y paso por valor al programar en Python.

El programa finalizado fue capaz de optimizar algunas intancias del problema como se pudo ver en el Capítulo 5 llegando a una mejora de hasta el 5.40 % del peso del árbol original (sección 5.2),

sin embargo, como cualquier otro sistema, puede mejorarse en código o sobre el modelo del algoritmo, para que de esta manera sea posible encontrar mejores resultados o encontrarlos en menor tiempo; para esto, se podría realizar lo siguiente:

- Hacer uso de cómputo concurrente para optimizar la ejecución del ciclo de vida de las partículas en lugar de ejecutar el conjunto de partículas de manera secuencial.
- Parametrizar mejor las áreas de búsqueda dentro del árbol, por ejemplo, utilizando el *cierre convexo*[1] del conjunto original de puntos.
- Afinar las constantes social, cognitiva y de inercia para verificar si es posible encontrar mejores resultados.
- En lugar de generar el enjambre en una posición aleatoria, utilizar alguna técnica para acercar el origen del enjambre a un posible punto de Steiner, por ejemplo, usar en centro de los triángulos obtenidos a partir de la *gráfica de Delaunay* del conjunto original.

Este trabajo es resultado de años de estudio en la Facultad de Ciencias, iniciando como un proyecto para Inteligencia Artificial. Por medio de conocimientos de Cómputo Evolutivo, optimizando los resultados con lo aprendido en el Seminario de Heurísticas de Optimización Combinatoria y entendiendo la teoría detrás gracias a materias como Análisis de Algoritmos, Complejidad Computacional y Geometría Computacional se logró perfeccionar la heurística y el desempeño del programa. Cada materia me dejó algo durante la etapa de estudiante y gracias a ello, fue posible la realización de este trabajo de tesis.

[1] Un *cierre convexo* se define como el *conjunto convexo* más pequeño que contiene a todos los puntos de un conjunto, donde un *conjunto convexo*, a su vez se define como un dominio D en un espacio euclidiano tal que para cualesquiera dos puntos $p_1, p_2 \in D$ el segmento $p_1\bar{p}_2 \in D$ por completo [70].

Bibliografía

[1] Abdeyazdan, Marjan: *A new method for the informed discovery of resources in the grid system using particle swarm optimization algorithm.* Volumen 73, Junio 2017.

[2] Arora, Sanjeev y Boaz Barak: *Computational Complexity: A Modern Approach.* Cambridge University Press, USA, 2009.

[3] Beardwood, Jillian, J. H. Halton y J. M. Hammersley: *The shortest path through many points.* Volumen 55, páginas 299–327. Cambridge University Press, 1959.

[4] Blackwell, T.M. y P. Bentley: *Don't push me! Collision-avoiding swarms.* En *Proceedings of the 2002 Congress on Evolutionary Computation.*, volumen 2, páginas 1691–1696, 2002.

[5] Cavalli-Sforza, L. L. y A. W. F. Edwards: *Phylogenetic Analysis: Models And Estimation Procedures.* Volumen 21, páginas 550–570, 1967.

[6] Chander, Akhilesh, Amitava Chatterjee y Patrick Siarry: *A new social and momentum component adaptive PSO algorithm for image segmentation.* Volumen 38, páginas 4998–5004, 2011.

[7] Cho, Jun Dong: *Steiner Tree Problems in VLSI Layout Designs*, páginas 101–173. Springer US, Boston, MA, 2001.

[8] Clerc, M. y J. Kennedy: *The particle swarm - explosion, stability, and convergence in a multidimensional complex space.* Volumen 6, páginas 58–73, 2002.

[9] Cockayne, Ernest J.: *On the Steiner Problem.* Volumen 10, página 431–450. Cambridge University Press, 1967.

[10] Coello Coello, Carlos Artemio: *Computación Evolutiva.* Academia Mexicana de Computación, A, C., México, 2019.

[11] Colorni, A., M. Dorigo, F. Maffioli, V. Maniezzo, G. Righini y M. Trubian: *Heuristics from nature for hard combinatorial optimization problems.* Volumen 3, páginas 1–21, 1996.

[12] Colorni, Alberto, Marco Dorigo y Vittorio Maniezzo: *Genetic Algorithms: A New Approach to the Timetable Problem.* En *Combinatorial Optimization*, páginas 235–239, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg.

[13] *Documentación oficial de la biblioteca copy.* Acceso: 2023-03-10. `https://docs.python.org/3/library/copy.html`.

[14] Courant, Richard y Herbert Robbins: *What is Mathematics?: An Elementary Approach to Ideas and Methods.* Oxford University Press, 1941.

[15] David M. Warme, Pawel Winter y Martin Zachariasen: *Exact Algorithms for Plane Steiner Tree Problems: A Computational Study*, páginas 81–116. Springer US, 2000.

[16] Decroos, Tom, Patrick De Causmaecker y Bart Demoen: *Solving Euclidean Steiner Tree Problems with Multi Swarm Optimization*. En *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*, páginas 1379–1380, New York, NY, USA, 2015. Association for Computing Machinery.

[17] Du, D Z y F K Hwang: *The Steiner ratio conjecture of Gilbert and Pollak is true*. Volumen 87, páginas 9464–9466. National Academy of Sciences, 1990.

[18] Eiselt, H. A. y C. L. Sandblom: *Introduction to Operations Research*, páginas 1–12. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[19] Eriksson, Folke: *The Fermat-Torricelli Problem Once More*. Volumen 81, páginas 37–44. Mathematical Association, 1997.

[20] Fernandez, A.D., H.M.A. Ghaziri, A.A. Diaz y F. Glover: *Optimización heurística y redes neuronales*. PARANINFO, 1996.

[21] Foulds, L.R. y R.L. Graham: *The steiner problem in phylogeny is NP-complete*. Volumen 3, páginas 43–49, 1982.

[22] Garey, M. R. y D. S. Johnson: *The Rectilinear Steiner Tree Problem is NP-Complete*. Volumen 32, páginas 826–834. Society for Industrial and Applied Mathematics, 1977.

[23] Garey, M.R. y D.S. Johnson: *Computers and Intractability: A Guide to the Theory of NP-completeness*. Mathematical Sciences Series. W. H. Freeman & Co., 1979.

[24] Gasca Soto, María de la Luz: *Introducción a los problemas NP-completos*. Vínculos matemáticos, 19, 2003.

[25] *Página oficial del software Geogebra*. Acceso: 2023-03-17. `https://www.geogebra.org`.

[26] Gilbert, E. N. y H. O. Pollak: *Steiner Minimal Trees*. Volumen 16, páginas 1–29. Society for Industrial and Applied Mathematics, 1968.

[27] Gilli, Manfred, Dietmar Maringer y Enrico Schumann: *Chapter 12 - Heuristic methods in a nutshell*. En Press, Academic (editor): *Numerical Methods and Optimization in Finance*, páginas 273–318. Academic Press, 2ª edición, 2019.

[28] Goffe, William L., Gardy D. Ferrier y John Rogers: *Simulated Annealing: An Initial Application in Econometrics*. Volumen 5, USA, Mayo 1992. Kluwer Academic Publishers.

[29] Hanan, Maurice: *On Steiner's Problem with Rectilinear Distance*. Volumen 14, páginas 255–265. Society for Industrial and Applied Mathematics, 1966.

[30] Harris, Frederick C. y Rakhi Motwani: *Steiner Minimal Trees: An Introduction, Parallel Computation, and Future Work*, páginas 3133–3177. Springer New York, New York, NY, 2013.

[31] Heppner, Frank y Ulf Grenander: *A stochastic nonlinear model for coordinated bird flocks*. The ubiquity of chaos, 1990.

[32] Hoffmann, Eduard: *Ueber das Kürzeste Verbindungssystem zwischen 4 Punkten der Ebene*. 1890.

[33] Holland, John H.: *Outline for a Logical Theory of Adaptive Systems.* Volumen 9, página 297–314, New York, NY, USA, Julio 1962. Association for Computing Machinery.

[34] Holland, John H.: *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence.* MIT Press, Cambridge, MA, USA, 1992.

[35] Ihler, Edmund: *The Complexity of Approximating the Class Steiner Tree Problem.* En *Proceedings of the 17th International Workshop*, WG 91, páginas 85–96. Springer-Verlag, 1992.

[36] Innami, Nobuhiro, B. Kim, Y. Mashiko y K. Shiohama: *The Steiner Ratio Conjecture of Gilbert-Pollak May Still Be Open.* Volumen 57, páginas 869–872, Agosto 2008.

[37] Jarník, Vojtěch y Miloš Kössler: *O minimálních grafech, obsahujících n daných bodů.* Volumen 63, páginas 223–235, 1934.

[38] Jilani, Mohd Zairul Mazwan Bin, Allan Tucker y Stephen Swift: *An Application of Generalised Simulated Annealing towards the Simultaneous Modelling and Clustering of Glaucoma.* Volumen 25, páginas 933–957, USA, Diciembre 2019. Kluwer Academic Publishers.

[39] Jongen, Hubertus Th, Klaus Meer y Eberhard Triesch: *Approximating NP-hard Problems*, páginas 353–363. Springer US, Boston, MA, 2004.

[40] *Introducción al formato JSON.* Acceso: 2023-03-15. `https://www.json.org/json-es.html`.

[41] *Documentación oficial de la biblioteca json.* Acceso: 2023-03-10. `https://docs.python.org/3/library/json.html`.

[42] Kennedy, J. y R. Eberhart: *Particle swarm optimization.* En *Proceedings of ICNN'95 - International Conference on Neural Networks*, volumen 4, páginas 1942–1948 vol.4, 1995.

[43] Kennedy, J. y W.M. Spears: *Matching algorithms to problems: an experimental test of the particle swarm and some genetic algorithms on the multimodal problem generator.* En *1998 IEEE International Conference on Evolutionary Computation Proceedings. IEEE World Congress on Computational Intelligence*, páginas 78–83, 1998.

[44] Kennedy, James: *Particle Swarm Optimization*, páginas 760–766. Springer US, Boston, MA, 2010.

[45] Kirkpatrick, S., C. D. Gelatt y M. P. Vecchi: *Optimization by Simulated Annealing.* Volumen 220, 1983.

[46] Kleene, Stephen C.: *The theory of recursive functions, approaching its centennial.* Volumen 5, páginas 43–61. American Mathematical Society, Julio 1981.

[47] Klein, Philip y Neal Young: *Approximation algorithms for NP-hard optimization problems.* Febrero 1999.

[48] Kozen, Dexter C.: *Automata and Computability.* Springer-Verlag, Berlin, Heidelberg, 1997.

[49] Kozen, Dexter C.: *The Complexity of Computations*, páginas 3–10. Springer London, London, 2006.

[50] Krink, T., J.S. Vesterstrom y J. Riget: *Particle swarm optimisation with spatial particle extension.* En *Proceedings of the 2002 Congress on Evolutionary Computation.*, volumen 2, páginas 1474–1479, 2002.

[51] Kupitz, Y y Horst Martini: *Geometric aspects of the generalized Fermat-Torricelli problem.* Volumen 6, páginas 55–129, 1997.

[52] Lovbjerg, M. y Krink T.: *Extending particle swarm optimisers with self-organized criticality.* En *Proceedings of the 2002 Congress on Evolutionary Computation.*, volumen 2, páginas 1588–1593, 2002.

[53] Lu, Chin Lung, Chuan Yi Tang y Richard Chia Tung Lee: *The Full Steiner Tree Problem.* Volumen 306, páginas 55–67. Elsevier Science Publishers Ltd., Septiembre 2003.

[54] M. Brazil, R. Graham, D. Thomas y M. Zachariasen: *On the history of the Euclidean Steiner tree problem.* Volumen 68, páginas 327–354. Springer, 2014.

[55] Madan, Mamta: *Bio-Inspired Computation for Optimizing Scheduling.* En *Nature Inspired Computing: Proceedings of CSI 2015*, páginas 69–74. Springer, 2018.

[56] Marinakis, Yannis y Magdalene Marinaki: *A hybrid genetic – Particle Swarm Optimization Algorithm for the vehicle routing problem.* Volumen 37, páginas 1446–1455, Marzo 2010.

[57] *Documentación oficial de la biblioteca math.* Acceso: 2023-03-10. `https://docs.python.org/3/library/math.html`.

[58] *Documentación oficial de la biblioteca matplotlib.* Acceso: 2023-03-10. `https://matplotlib.org`.

[59] Melzak, Zdzislaw Alexander: *On the problem of Steiner.* Volumen 4, páginas 143–148. Cambridge University Press, 1961.

[60] Mestre, Julián y Rajiv Raman: *Max-Coloring*, páginas 1871–1911. Springer New York, New York, NY, 2013.

[61] Miehle, William: *Link-length minimization in networks.* Volumen 6, páginas 232–243. INFORMS, 1958.

[62] Mumford, Michael D. y Lyle E. Leritz: *Heuristics.* En *Encyclopedia of Social Measurement*, páginas 203–208. Elsevier, New York, 2005.

[63] *Documentación oficial de la biblioteca NetworkX.* Acceso: 2023-03-10. `https://networkx.org/documentation/stable/index.html`.

[64] Neumann, Frank y Carsten Witt: *Stochastic Search Algorithms*, páginas 21–32. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[65] *Documentación oficial de la biblioteca numpy.* Acceso: 2023-03-10. `https://numpy.org`.

[66] Oliveira, Carlos A. S. y Panos M. Pardalos: *Steiner Trees and Multicast*, páginas 29–45. Springer New York, New York, NY, 2011.

[67] Pei, Xiaofang, Nan Li y Shuiping Wang: *Application of Simulated Annealing Algorithm in Fingerprint Matching.* En *Applied Informatics and Communication*, páginas 33–40, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[68] Poli, Riccardo, James Kennedy y Tim Blackwell: *Particle Swarm Optimization: An Overview.* Volumen 1, Octubre 2007.

[69] Pornsing, C.: *A Particle Swarm Optimization for the Vehicle Routing Problem.* University of Rhode Island, 2014.

[70] Preparata, Franco P. y Michael Ian Shamos: *Convex Hulls: Basic Algorithms*, páginas 95–149. Springer New York, New York, NY, 1985.

[71] *Documentación oficial de la biblioteca random.* Acceso: 2023-03-10. `https://docs.python.org/3/library/random.html`.

[72] Reynolds, Craig W.: *Flocks, Herds and Schools: A Distributed Behavioral Model.* Volumen 21, páginas 25–34, New York, NY, USA, Agosto 1987. Association for Computing Machinery.

[73] Russell, Stuart y Peter Norvig: *Artificial intelligence - A Modern Approach.* Pearson Education, Inc., New Jersey, USA, 2010.

[74] Seydi Ghomsheh, V., M. Aliyari Shoorehdeli y M. Teshnehlab: *Training ANFIS structure with modified PSO algorithm.* En *2007 Mediterranean Conference on Control Automation*, páginas 1–6, Junio 2007.

[75] *Documentación oficial de la biblioteca unittest.* Acceso: 2023-03-10. `https://docs.python.org/3/library/unittest.html`.

[76] Wang, Jie, You Li, Benyuan Liu, Guanling Chen y Jun Hong Cui: *Optimization Problems in Online Social Networks*, páginas 2455–2501. Springer New York, New York, NY, 2013.

[77] Wilson, Edward O.: *Sociobiology: The New Synthesis, Twenty-Fifth Anniversary Edition.* Harvard University Press, 2000.

[78] Xiao-Feng Xie, Wen Jun Zhang y Zhi Lian Yang: *Dissipative particle swarm optimization.* En *Proceedings of the 2002 Congress on Evolutionary Computation.*, volumen 2, páginas 1456–1461, 2002.

[79] Zhang, Lei, Yuehui Chen y Bo Yang: *Task Scheduling Based on PSO Algorithm in Computational Grid.* Volumen 2, páginas 696–704, Octubre 2006.

[80] Zhong, Wen Liang, Jian Huang y Jun Zhang: *A novel particle swarm optimization for the Steiner tree problem in graphs.* En *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)*, páginas 2460–2467, 2008.