Universidad Autónoma
Metropolitana

UNIDAD CUAJIMALPA

# GREEN-VRP

Proyecto Terminal

QUE PRESENTA:
ALEJANDRO MARTÍNEZ GUZMÁN

Licenciatura en Ingeniería en Computación

Departamento de Matemáticas Aplicadas e Ingeniería

División de Ciencias Naturales e Ingeniería

Asesor y Responsable de la tesis:
EDWIN MONTES OROZCO

Mes y año (de finalización)

# Índice general

# Índice de figuras

# Capítulo 1

# Resumen

# Capítulo 2

# Introducción

# Capítulo 3

# Conocimientos preliminares

## 3.1. Problema de Optimización

La optimización es una rama de las matemáticas aplicadas que se enfoca en el desarrollo de principios y métodos para resolver problemas cuantitativos en diversas disciplinas como la física, biología, ingeniería o economía, así como en cualquier campo donde sea necesaria la toma de decisiones dentro de un conjunto de opciones, con el objetivo de encontrar la mejor o una de las mejores soluciones de manera eficiente.

En términos formales, consideramos una función $f : S \to \mathbb{R}$, donde $S \subseteq \mathbb{R}^n$. Denominaremos a $f$ como función objetivo, y a $S$ lo llamaremos conjunto factible o conjunto de soluciones posibles.

## 3.2. Tipos de Optimización

Dentro de la optimización, existen dos tipos principales: la optimización discreta y la optimización continua.

La optimización discreta se aplica cuando el dominio $S$ de la función objetivo es un conjunto discreto. En este contexto, el conjunto de soluciones posibles es finito o numerablemente infinito. La optimización discreta a menudo involucra la búsqueda de soluciones en combinaciones específicas y puede in-

volucrar problemas como la programación lineal entera o los problemas de asignación. Las soluciones óptimas pueden ser difíciles de encontrar debido a la naturaleza combinatoria del problema.

Por otro lado, la optimización continua se refiere a situaciones en las que el conjunto $S$ de la función objetivo es un conjunto continuo. En este caso, el dominio es un intervalo o un subconjunto de $\mathbb{R}^n$ que no es discreto. Aquí se buscan soluciones que maximizan o minimizan la función objetivo sobre un espacio continuo, y los métodos comunes incluyen la programación lineal, la programación no lineal y el cálculo de variaciones. La optimización continua suele implicar el uso de técnicas de cálculo y análisis matemático.

## 3.3. Heurística y Metaheurística

La palabra "heurística"proviene del término griego "euriskein", que significa "encontrar". En el contexto de la optimización, se refiere a técnicas diseñadas para mejorar o resolver problemas que, de otra manera, no tendrían una solución eficiente. Los algoritmos heurísticos ayudan a encontrar soluciones aproximadas para problemas cuyas soluciones exactas no son factibles en tiempos polinomiales. Muchos algoritmos de inteligencia artificial son heurísticos o se basan en sus principios para resolver problemas.

El prefijo "meta.ºtorga un sentido de mayor nivel a la palabra heurística, lo que permite abordar problemas de mayor complejidad mediante soluciones factibles.

Tanto las heurísticas como las metaheurísticas se pueden utilizar de manera intercambiable para resolver problemas de optimización combinatoria.

## 3.4. Complejidad P, NP, NP-completo y NP-difícil

En computación, un algoritmo es un procedimiento que resuelve un problema paso a paso, generando una salida a partir de una entrada en un número finito de pasos. La eficiencia de un algoritmo se evalúa a través de su complejidad

temporal, la cual se mide en función del tamaño de la entrada. Esto nos permite clasificar los problemas en diferentes clases de complejidad, como P, NP, NP-completo y NP-difícil.

La clase P contiene problemas que pueden resolverse eficientemente (en tiempo polinomial), mientras que la clase NP contiene aquellos problemas cuyas soluciones pueden verificarse eficientemente. Las clases NP-completo y NP-difícil incluyen problemas más complejos; resolver uno de estos de manera eficiente significaría resolver eficientemente cualquier problema en NP. Esto nos lleva al problema fundamental en teoría de la computación: P vs NP.

### 3.4.1. Complejidad P

En la teoría de la complejidad, nos enfocamos principalmente en problemas de decisión, aquellos donde la respuesta es un simple "sí.º "no". La clase P está compuesta por problemas que pueden ser resueltos en tiempo polinomial. Esto significa que existe un algoritmo que, dado un problema de tamaño $n$, puede resolverlo en un tiempo que crece a lo sumo como un polinomio de $n$.

#### Ejemplo: Algoritmo de Ordenamiento por Selección

El algoritmo de ordenamiento por selección es un método simple para ordenar una lista de elementos, aunque no es el más eficiente para listas grandes. A continuación, se presenta una explicación detallada del algoritmo, su funcionamiento, complejidad y características.

**Descripción del algoritmo** El algoritmo selecciona repetidamente el elemento mínimo de la lista y lo coloca en su posición correcta. Consideremos la lista (3 1 6 8 2). El proceso de ordenamiento sería:

1. (3 1 6 8 2) - Estado inicial
2. (1 3 6 8 2) - Se intercambia 1 (mínimo) con 3
3. (1 2 6 8 3) - Se intercambia 2 con 3
4. (1 2 3 8 6) - Se intercambia 3 con 6

5. (1 2 3 6 8) - Se intercambia 6 con 8
6. (1 2 3 6 8) - La lista está ordenada

**Representación matemática:** Para una lista de $n$ elementos:

- Primera pasada: $n - 1$ comparaciones
- Segunda pasada: $n - 2$ comparaciones
- $\vdots$
- Última pasada: 1 comparación

**Complejidad:** El número total de comparaciones es $(n-1) + (n-2) + \cdots + 2 + 1 = \frac{n(n-1)}{2}$, lo que da una complejidad de $O(n^2)$.

El algoritmo de ordenamiento por selección, aunque simple de entender e implementar, no es eficiente para listas grandes debido a su complejidad cuadrática. Sin embargo, su tiempo de ejecución es polinomial, por lo que pertenece a la clase P.

### 3.4.2. Clase de complejidad NP

La clase NP abarca problemas donde una solución afirmativa ("sı") puede verificarse en tiempo polinomial. En estos problemas, existe un testigo o certificado que sirve como prueba de la validez de la respuesta, y un algoritmo verificador que puede validar este testigo eficientemente.

Un ejemplo clásico es el problema de la suma de subconjuntos: dado un conjunto de $n$ números, se busca si existe un subconjunto cuya suma sea $S$. Para verificar una solución afirmativa, bastaría con que nos dieran el subconjunto en cuestión, y en tiempo lineal podríamos sumar sus elementos para comprobar que la suma es correcta.

Al igual que en P, los problemas en NP pueden incluir aquellos que ya mencionamos, pero con una importante distinción: aunque podemos verificar

soluciones en tiempo polinomial, no necesariamente podemos encontrarlas rápidamente. De hecho, uno de los grandes interrogantes en teoría de la complejidad es si todos los problemas en NP pueden ser también resueltos en tiempo polinomial.

### Ejemplo de problema NP: El Problema del Agente Viajero

El Problema del Agente Viajero (TSP, por sus siglas en inglés) es un ejemplo claro de un problema NP. El objetivo es encontrar una trayectoria que visite todas las $n$ ciudades exactamente una vez y regrese a la ciudad de origen, minimizando el costo total del recorrido.

**Datos del problema:**

- Un entero $n > 0$, que representa el número de ciudades.
- Una matriz de distancias $(d_{ij})$ de dimensión $n \times n$, donde $d_{ij}$ es un entero mayor o igual a cero que representa la distancia entre la ciudad $i$ y la ciudad $j$.

**Representación matemática:** Una permutación $\pi$ cíclica representa un recorrido, donde $\pi(j)$ se interpreta como la ciudad que sigue a la ciudad $j$ para $j = 1, 2, \ldots, n$.

**Costo del recorrido:** El costo del recorrido se calcula mediante la siguiente fórmula:

$$\sum_{j=1}^{n} d_{j,\pi(j)}$$

Donde:

- $j$ recorre todas las ciudades de 1 a $n$.
- $d_{j,\pi(j)}$ representa la distancia entre la ciudad $j$ y la ciudad que le sigue en el recorrido según la permutación $\pi$.

**Complejidad:** El Problema del Agente Viajero pertenece a la clase NP. Si nos dieran una solución (un recorrido), podríamos verificar si es válida y calcular su costo en tiempo polinomial. Sin embargo, encontrar la solución óptima no se conoce como un problema que pueda resolverse en tiempo polinomial para todos los casos.

### 3.4.3. Reducciones y las clases NP-completo y NP-difícil

Para profundizar en las clases de NP-completo y NP-difícil, es esencial introducir el concepto de reducción. Una reducción entre problemas significa que si podemos resolver un problema de manera eficiente, entonces también podemos resolver el otro. Esto es clave para entender la relación entre los problemas de estas clases.

#### Definición de Reducción Polinomial

Dado un problema A y un problema B, diremos que existe una reducción polinomial de A a B si cualquier instancia de A puede ser transformada en una instancia de B en tiempo polinomial, de manera que resolver B implica resolver A.

Esto nos lleva a la clase de complejidad NP-difícil, que está compuesta por problemas a los cuales cualquier problema en NP puede ser reducido. NP-completo, por otro lado, incluye aquellos problemas que son tanto NP-difíciles como pertenecientes a NP. Si alguna vez encontráramos un algoritmo eficiente para resolver un problema NP-completo, entonces podríamos resolver todos los problemas en NP de manera eficiente.

En resumen las clases de complejidad P, NP, NP-completo y NP-difícil son fundamentales para comprender la eficiencia de los algoritmos. El problema abierto más importante de la teoría de la computación, P vs NP, sigue siendo una pregunta sin respuesta: ¿Podemos resolver todos los problemas que podemos verificar eficientemente? Resolver esta pregunta tendría implicaciones profundas no solo en la teoría, sino en muchas áreas prácticas de la computación.

## 3.5. Computación Evolutiva

La computación evolutiva es una rama de las Ciencias de la Computación que se inspira en la evolución natural para resolver problemas mediante un enfoque de prueba y error. Este proceso se basa en una población de individuos (soluciones candidatas) que compiten por recursos, donde su calidad (aptitud) determina su capacidad para sobrevivir y reproducirse.

### 3.5.1. Esquema General de un Algoritmo Evolutivo

1. **Inicio**
2. INICIALIZAR la población con soluciones candidatas aleatorias.
3. EVALUAR cada candidato.
4. Mientras no se cumpla la condición de paro:
   *a*) SELECCIONAR padres.
   *b*) RECOMBINAR padres.
   *c*) MUTAR la descendencia.
   *d*) SELECCIONAR individuos para la siguiente generación.
5. **Fin**

## 3.6. Evolución Diferencial

El algoritmo de Evolución Diferencial (DE) fue desarrollado por Price y Storn en 1995 como un método de optimización versátil y fácil de usar. DE utiliza una población de vectores (soluciones) y genera nuevas soluciones mediante mutaciones y recombinaciones.

### 3.6.1. Inicialización

Se inicia con una población $P_x$ de $N_P$ vectores $D$-dimensionales. Los vectores se crean aleatoriamente dentro de límites predefinidos.

### 3.6.2. Mutación

La mutación se realiza sumando una diferencia escalada de dos vectores aleatorios a un tercer vector:

$$v_{i,g} = x_{r_0,g} + F(x_{r_1,g} - x_{r_2,g})$$

Donde $F$ es un factor de escala, y $r_0$, $r_1$, y $r_2$ son índices aleatorios de la población.

### 3.6.3. Recombinación

Los vectores trial se generan al intercambiar elementos entre el vector mutante y el vector padre:

$$u_{i,g} = \begin{cases} v_{j,g} & \text{si } r_j \leq CR \\ x_{i,g} & \text{si } r_j > CR \end{cases}$$

Donde $CR$ es la tasa de recombinación.

### 3.6.4. Selección

Finalmente, se elige el vector que tenga el menor valor de la función objetivo:

$$x_{i,g+1} = \begin{cases} u_{i,g} & \text{si } f(u_{i,g}) < f(x_{i,g}) \\ x_{i,g} & \text{en caso contrario} \end{cases}$$

Este proceso se repite hasta que se alcanza un criterio de parada, como encontrar una solución aceptable o alcanzar un límite computacional.

## 3.7. El Problema del Agente Viajero (TSP)

El problema del agente viajero (Travelling Salesman Problem - TSP) es uno de los problemas más estudiados en el campo de la optimización y la planificación de rutas. El TSP consiste en encontrar un recorrido de longitud mínima que permita a un agente visitar un conjunto de $n$ ciudades, pasando por cada una exactamente una vez y regresando a la ciudad de origen. Este problema es de gran importancia en aplicaciones como la logística, la distribución de productos y el diseño de circuitos.

### 3.7.1. Función Objetivo

La función objetivo del TSP tiene como meta minimizar la distancia total recorrida entre las ciudades. Matemáticamente, se expresa de la siguiente forma:

$$\text{mín} \sum_{i=1}^{n} \sum_{j=1}^{n} d_{ij} X_{ij}$$

Donde:

- $d_{ij}$ representa la distancia entre la ciudad $i$ y la ciudad $j$.
- $X_{ij}$ es una variable binaria que toma el valor 1 si la ciudad $i$ es visitada justo antes de la ciudad $j$, y 0 en caso contrario.

El objetivo es minimizar la suma total de las distancias $d_{ij}$ de todas las transiciones $i \to j$, asegurando que cada ciudad sea visitada solo una vez.

### 3.7.2. Restricciones

Para garantizar que el recorrido cumpla con las condiciones del problema, se deben cumplir las siguientes restricciones:

1. **Restricción de unicidad de salida:** Cada ciudad $i$ debe tener exactamente una ciudad sucesora (una ciudad $j$ a la que viajar):

$$\sum_{j=1}^{n} X_{ij} = 1, \quad \forall i = 1, 2, \ldots, n$$

   Esta restricción asegura que cada ciudad tenga exactamente una ciudad siguiente en el recorrido.

2. **Restricción de unicidad de entrada:** Cada ciudad $j$ debe ser visitada exactamente una vez por alguna ciudad anterior $i$:

$$\sum_{i=1}^{n} X_{ij} = 1, \quad \forall j = 1, 2, \ldots, n$$

   Esta restricción asegura que cada ciudad sea visitada por exactamente una ciudad anterior.

3. **Restricciones de subciclo:** Para evitar subciclos (recorridos más cortos que no incluyan todas las ciudades), se introducen variables auxiliares $u_i$ que controlan el orden en el que se visitan las ciudades. Estas variables deben cumplir la siguiente restricción:

$$u_i - u_j + nX_{ij} \leq n - 1, \quad \forall i, j = 2, 3, \ldots, n, \; i \neq j$$

   Donde $u_i$ y $u_j$ son variables auxiliares que ayudan a eliminar los subciclos. Estas restricciones aseguran que no se generen ciclos más pequeños que el recorrido completo.

4. **Restricción binaria:** Las variables $X_{ij}$ son binarias, es decir, pueden tomar solo los valores 0 o 1, indicando si una ciudad $i$ es seguida por la ciudad $j$ en el recorrido:

$$X_{ij} \in \{0, 1\}, \quad \forall i, j = 1, 2, \ldots, n$$

### 3.7.3. Ejemplo

Consideremos un problema con $n = 4$ ciudades. La matriz de distancias $d_{ij}$ podría ser:

$$d_{ij} = \begin{pmatrix} 0 & 10 & 15 & 20 \\ 10 & 0 & 35 & 25 \\ 15 & 35 & 0 & 30 \\ 20 & 25 & 30 & 0 \end{pmatrix}$$

El objetivo es encontrar un recorrido que minimice la suma de las distancias entre las ciudades, cumpliendo con las restricciones mencionadas anteriormente.

### 3.7.4. Complejidad Computacional: TSP es NP-completo

Una de las características más importantes del TSP es que pertenece a la clase de problemas NP-completos. Esto significa que no existe un algoritmo eficiente conocido que pueda resolver el TSP de manera óptima en tiempo polinomial para todos los casos posibles. Dado que el número de posibles recorridos crece factorialmente con el número de ciudades ($n!$), resolver el TSP para grandes instancias no es computacionalmente eficiente.

# Capítulo 4

# Desarrollo del proyecto

# Capítulo 5

# Conclusiones y trabajo futuro

# Bibliografía