Vendedor Viajero - Un problema de Optimización

Trabajo Práctico 2 - Algebra Lineal y Optimización para DS

AUTOR/A
Cinthya Leonor Vergara Silva

FECHA DE PUBLICACIÓN
27 de noviembre de 2024

1 Presentación y descripción del problema

Este trabajo práctico aborda la implementación del algoritmo de *recocido simulado* (Simulated Annealing) (Kirkpatrick, Gelatt Jr, y Vecchi 1983) para resolver el problema del *vendedor viajero* (Traveling Salesman Problem - TSP) (Menger 1928; Schrijver 2005).

El algoritmo de recocido simulado está inspirado en un proceso metalúrgico llamado recocido. En metalurgia, cuando se calienta un metal a altas temperaturas y luego se enfría lentamente, los átomos tienen la oportunidad de reorganizarse en una estructura más estable y con menos defectos. Este proceso físico sirve como metáfora para resolver problemas de optimización computacional.

Cómo método de optimización, el *recocido simulado* se utiliza para resolver problemas sin restricciones y con restricciones limitadas. El método modela el proceso físico de calentar un material y luego reducir lentamente la temperatura para disminuir los defectos, minimizando así la energía del sistema.

En cada iteración del algoritmo de recocido simulado, se genera aleatoriamente un nuevo punto. La distancia del nuevo punto con respecto al punto actual, o la extensión de la búsqueda, se basa en una distribución de probabilidad con una escala proporcional a la temperatura. El algoritmo acepta todos los puntos nuevos que reducen el objetivo, pero también, con cierta probabilidad, los puntos que lo elevan. Al aceptar puntos que lo elevan, el algoritmo evita quedar atrapado en mínimos locales y puede explorar globalmente más soluciones posibles. Se selecciona un programa de recocido para disminuir sistemáticamente la temperatura a medida que avanza el algoritmo. A medida que la temperatura disminuye, el algoritmo reduce la extensión de su búsqueda para converger al mínimo (The MathWorks 2024).

En general se puede estucturar como:

Algoritmo SimulatedAnnealing:

```
Entrada:

    - Solución Inicial (S0)
    - Temperatura Inicial (T0)
    - Función de Evaluación (f(S))
    - Función de Enfriamiento (T(t))
    - Criterio de Parada (criterio)

Salida:
    - Mejor solución encontrada

1. Establecer S ← S0  // Establecer la solución inicial
```

```
2. Establecer T ← T0  // Establecer la temperatura inicial
3. Mientras no cumpla el criterio de parada:
    4. Generar una solución vecina S' de S
    5. Evaluar la solución vecina: f(S')
    6. Si f(S') < f(S):
        7. Aceptar S' como la nueva solución (S ← S')
    8. Sino:
        9. Calcular la probabilidad de aceptación: P = exp((f(S) - f(S')) / T)
        10. Si aleatorio() < P:
            11. Aceptar S' como la nueva solución (S ← S')
    12. Reducir la temperatura T según la función de enfriamiento: T ← T(t)
13. Retornar la mejor solución encontrada
```

Lo que puede expresarse como una función que toma un problema y retorna una solución encontrada.

```python
from random import random
from function_simulated_annealing_base import *
import math

# ejemplo base
distancias = [
    [0, 10, 15, 20],
    [10, 0, 35, 25],
    [15, 35, 0, 30],
    [20, 25, 30, 0]
]

TEMPERATURA_INICIAL = 7
iteraciones = 0
max_iteraciones = 7

def simulated_annealing(distancias):
    n_cities = len(distancias)  # Número de ciudades
    solucion_actual = generar_solucion_inicial(n_cities)
    mejor_solucion = solucion_actual
    temperatura = TEMPERATURA_INICIAL
    TASA_ENFRIAMIENTO = 0.95

    while not no_criterio_parada():
        nueva_solucion = generar_vecino(solucion_actual)
                delta_energia = calcular_energia(nueva_solucion, distancias)
                 - calcular_energia(solucion_actual, distancias)

        if delta_energia < 0 or random.random()
         < probabilidad_aceptacion(delta_energia, temperatura):
            solucion_actual = nueva_solucion

        if calcular_energia(solucion_actual, distancias)
        < calcular_energia(mejor_solucion, distancias):
            mejor_solucion = solucion_actual
```

```
        temperatura *= TASA_ENFRIAMIENTO

    return mejor_solucion

mejor_ruta = simulated_annealing(distancias)
print("Mejor ruta encontrada:", mejor_ruta)
print("Costo total:", calcular_energia(mejor_ruta, distancias))
```

El objetivo principal de este informe es analizar la eficacia del algoritmo de **recocido simulado** en la resolución del **Problema del vendedor viajero**, su relación con los **ciclos Hamiltonianos** y cómo se formula como un **problema de programación entera**. Finalmente se compararán los resultados respecto al algoritmo **Branch&Cut**.

2 Desarrollo del problema planteado

2.1 Ciclo Hamiltoniano y el Problema del Vendedor Viajero (TSP)

El ciclo Hamiltoniano dentro de la teoría de grafos, corresponde a un camino dentro de un grafo donde se visita cada vértice exactamente una vez, regresa al punto de origen y pasa por todos los vértices del grafo sin repetir ninguno. En términos análogos se puede ver como un viaje en el cual un vendedor debe visitar cada ciudad una única vez y regresar a su ciudad de origen, buscando la ruta más corta que cumpla con estas condiciones.

En términos simples, el ciclo Hamiltoniano representa un recorrido eficiente y sin repeticiones que cubre todos los puntos de un grafo, lo que corresponde al objetivo principal en el **Problema del Vendedor Viajero (TSP)**.

El Problema del Vendedor Viajero (Traveling Salesman Problem o TSP) puede modelarse como un problema de búsqueda en un grafo, donde cada ciudad es un vértice y las conexiones entre ciudades son las aristas con pesos representando las distancias. La solución óptima consiste en encontrar un ciclo Hamiltoniano de costo mínimo (Bertsimas y Howell 1993). Este problema se puede formular como un problema de optimización combinatorial y pertenece a la clase de problemas NP-difíciles, lo que implica que no existe un algoritmo de fuerza bruta eficiente para resolverlo en grafos grandes. Existen diversas técnicas para abordar este problema, desde la búsqueda exhaustiva hasta algoritmos heurísticos avanzados como el Recocido Simulado y los Algoritmos Genéticos.

2.2 Modelado del Problema del Vendedor Viajero en un Grafo

El **Problema del Vendedor Viajero (TSP)** es un problema de optimización combinatorial que puede modelarse como una búsqueda en un grafo ponderado. Los elementos para su modelado son:

2.2.1 Elementos del Grafo

1. **Vértices**: Cada vértice del grafo representa una ciudad que el vendedor debe visitar.
2. **Aristas**: Las aristas del grafo representan las conexiones entre las ciudades, y tienen un peso asociado, que en el contexto del TSP corresponde a la distancia (o costo) de viajar entre dos ciudades.

2.2.2 Modelo Matemático

El conjunto de ciudades puede representarse como un conjunto $C = \{c_1, c_2, \ldots, c_n\}$, donde n es el número de ciudades a visitar. Además, se define una matriz de distancias D, donde D_{ij} es la distancia entre las ciudades c_i y c_j.

El objetivo del TSP es encontrar una permutación π de las ciudades tal que el vendedor recorra todas las ciudades exactamente una vez y regrese al punto de origen, minimizando la distancia total recorrida. La distancia total del recorrido está dada por la fórmula:

$$\text{Distancia Total} = \sum_{i=1}^{n-1} D_{c_{\pi(i)}, c_{\pi(i+1)}} + D_{c_{\pi(n)}, c_{\pi(1)}}$$

El problema consiste en encontrar la permutación π que minimice la distancia total.

Luego, las principales características del TSP son las siguientes (Laporte 1992):

- El objetivo es minimizar la distancia entre las ciudades visitadas.
- Todas las ciudades deben ser visitadas una sola vez.
- El recorrido es cerrado, es decir, el origen y el destino son el mismo lugar.

Y, esencialmente, se formula como :

$$\text{Minimizar } z = \sum_{i=1}^{n} \sum_{j=1}^{n} d_{ij} x_{ij}$$

Sujeto a:

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

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

$$x_{ij} \in \{0, 1\}$$

donde d_{ij} es la distancia de la ciudad i a la ciudad j, $d_{ij} = \infty$ cuando $i = j$ y $d_{ij} = d_{ji}$.

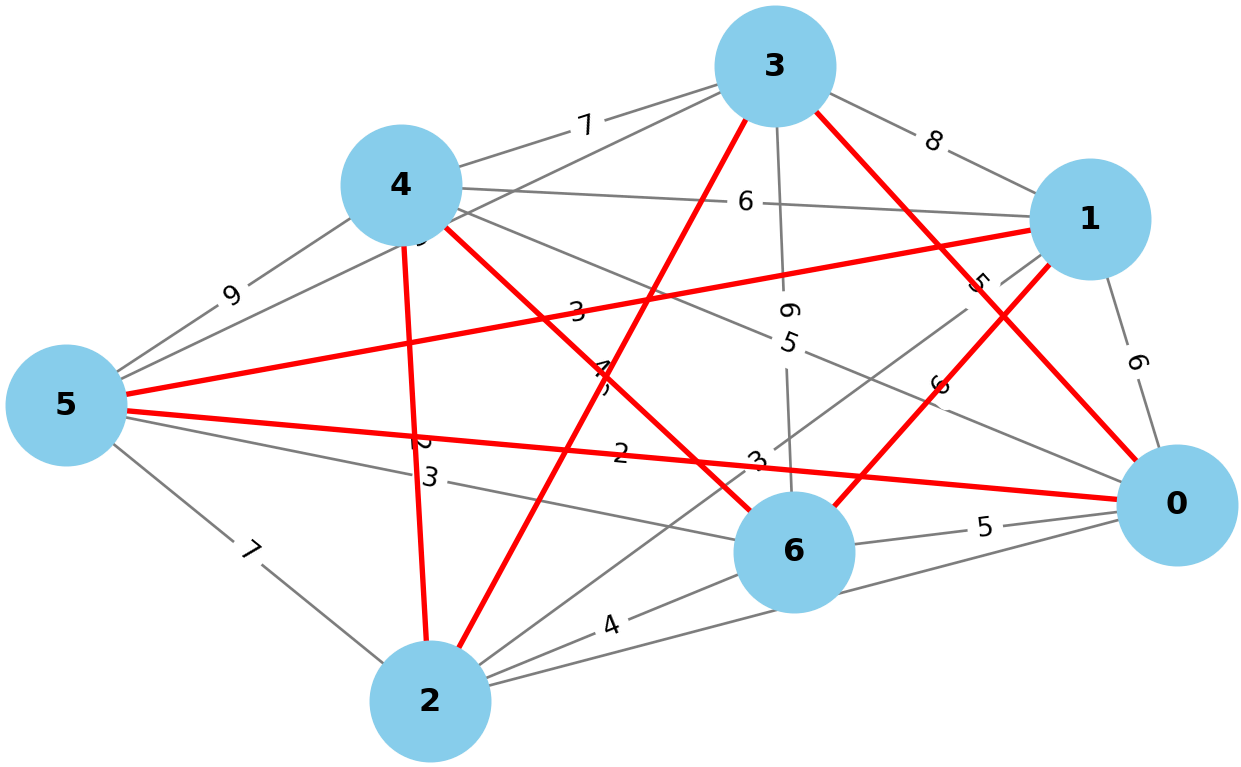
2.2.3 Modelado como en código

```
import itertools
import numpy as np

# 1. Crear la matriz de distancias entre las ciudades
# Número de ciudades
n = 7
ciudades = list(range(n))

# Generar una matriz de distancias aleatorias (simétrica)
np.random.seed(42)  # Para reproducibilidad
distancias = np.random.randint(1, 10, size=(n, n))
```

```
np.fill_diagonal(distancias, 0)  # La distancia a sí mismo es 0
distancias = (distancias + distancias.T) // 2  # Hacer la matriz simétrica

# 2. Función para calcular el costo de un ciclo
def calcular_costo_ciclo(ciclo, distancias):
    costo_total = 0
    for i in range(len(ciclo) - 1):
        costo_total += distancias[ciclo[i], ciclo[i+1]]
    costo_total += distancias[ciclo[-1], ciclo[0]]  # Regresar al punto de inicio
    return costo_total

# 3. Función de búsqueda de todas las permutaciones (fuerza bruta)
def tsp_fuerza_bruta(ciudades, distancias):
    # Generar todas las permutaciones posibles de las ciudades (sin repetir)
    todas_las_permutaciones = itertools.permutations(ciudades)
    costo_minimo = float('inf')
    mejor_ciclo = None

    # Evaluar cada posible recorrido (ciclo Hamiltoniano)
    for perm in todas_las_permutaciones:
        costo = calcular_costo_ciclo(perm, distancias)
        if costo < costo_minimo:
            costo_minimo = costo
            mejor_ciclo = perm

    return mejor_ciclo, costo_minimo

# 4. Ejecutar el algoritmo de fuerza bruta para encontrar la mejor ruta
mejor_ciclo, costo_minimo = tsp_fuerza_bruta(ciudades, distancias)

# Mostrar la solución óptima
print(f"La mejor ruta es: {mejor_ciclo}")
print(f"El costo total de la ruta es: {costo_minimo}")
```

```
La mejor ruta es: (0, 3, 2, 4, 6, 1, 5)
El costo total de la ruta es: 25
```

Grafo del Problema del Vendedor Viajero (TSP) con la Ruta Óptima



2.3 Formulación del TSP como un Problema de Programación Lineal Entera (ILP)

El **Problema del Vendedor Viajero (TSP)** se puede modelar como un problema de **programación lineal entera (ILP)** (Laporte 1992; Bertsimas y Tsitsiklis 1997). En este enfoque, se busca encontrar una **solución óptima** que minimice el **costo total de viajar** entre un conjunto de ciudades, cumpliendo con las siguientes condiciones:

- Cada ciudad se visita exactamente una vez.
- El vendedor regresa a su ciudad de origen.
- No se forman subciclos (es decir, no se crean rutas que no cubren todas las ciudades).

Al igual que en las definiciones previas de este informe, se define la estructura de grafo como:

1. **V**: Es el conjunto de **ciudades** que deben ser visitadas.
2. **E**: Representa las **conexiones** entre las ciudades. Formalmente $E = \{(a, b) : a, b \in V, a \neq b\}$, donde cada par de ciudades está conectado por un arco en el grafo.
3. **c**: Es el **costo** o **distancia** entre las ciudades. Si tenemos una matriz de distancias d, entonces c_{ab} es el costo de viajar de la ciudad (a) a la ciudad (b).
4. **±(S)**: Representa los **arcos cruzando la frontera** de un subconjunto S de ciudades, es decir, los arcos que conectan a las ciudades dentro de S con las ciudades fuera de S.

$$\pm(S) = \{(a, b) \in E : a \in S, b \in V \setminus S\}$$

Con ello, el objetivo del problema es **minimizar** el costo total de las conexiones seleccionadas en el ciclo. Esto se puede expresar como:

$$\text{minimizar } Z = \sum_{(a,b)\in E} c_{ab} x_{ab}$$

donde x_{ab} es una variable binaria que toma el valor de 1 si el arco (a, b) es parte del ciclo y 0 si no lo es.

Junto con la función objetivo, se definen las siguientes restricciones:

1. **Restricción de visita única por ciudad**:

   Para cada ciudad v, el número de arcos que entran y salen de esa ciudad debe ser igual a 2. Esto garantiza que cada ciudad sea visitada exactamente una vez:

   $$\sum_{(a,b)\in \pm(\{v\})} x_{ab} = 2 \quad \forall v \in V$$

   Es decir, cada ciudad tiene un arco entrante y un arco saliente.

2. **Restricción para evitar subciclos**:

   Para evitar que se formen subciclos (es decir, ciclos que no incluyen todas las ciudades), se utiliza una restricción de subtour. Esto implica que no se puede formar un ciclo cerrado dentro de un subconjunto de ciudades S que no cubra todo el conjunto V. Esta restricción es fundamental para asegurar que el ciclo sea único y cubra todas las ciudades:

   $$\sum_{(a,b)\in \pm(S)} x_{ab} \geq 2 \quad \forall S \subset V, S \neq V$$

   Esta restricción asegura que cualquier subconjunto de ciudades (S) con al menos 2 ciudades no puede formar un ciclo cerrado que no cubra todas las ciudades.

3. **Restricción de variables binarias**:

   Cada variable x_{ab} es binaria, lo que significa que el valor de x_{ab} solo puede ser 0 o 1:

   $$x_{ab} \in \{0, 1\} \quad \forall (a, b) \in E$$

   Esto indica si el arco (a, b) es parte del ciclo (1) o no (0).

4. **Restricciones adicionales (subtour elimination)**:

   A veces, se introducen variables auxiliares $u_v $ para evitar subciclos. Estas variables controlan el orden en que se visitan las ciudades, garantizando que no se formen ciclos pequeños dentro del ciclo global. Si se usan, las restricciones para las variables u_v podrían tener la forma:

   $$u_i - u_j + (n-1)x_{ij} \leq n - 2, \quad \forall i \neq j,\ 2 \leq i, j \leq n$$

   Donde u_i representa el orden en el que se visita la ciudad (i).

2.4 Estrategias para resolver el TSP

Existen diversas estrategias para abordar el problema Singh, Bedi, y Gaur (2020). Una de las opciones más directas es el método de **fuerza bruta**, que consiste en generar todas las permutaciones posibles de las ciudades, calcular la distancia de cada permutación y seleccionar la ruta más corta. Sin embargo, este enfoque tiene una complejidad de tiempo de $O(n!)$, lo que lo hace altamente ineficiente para grafos grandes (tiempo exponencial).

Existen algoritmos de **optimización exactos** como *Branch and Bound* que se basa en la exploración sistemática de todas las soluciones posibles, pero utiliza técnicas de poda para eliminar partes del espacio de búsqueda que no pueden contener la solución óptima. En cuanto al orden, en el peor de los casos, puede ser exponencial, es decir, $O(n!)$ para el TSP, ya que en el peor de los casos se pueden explorar todas las permutaciones de las ciudades. Sin embargo, en la práctica, la eficiencia del algoritmo puede mejorar considerablemente con una buena estrategia de poda, a menudo en el rango de $O(n^2)$ a $O(n^3)$.

Otra estrategia que se emplea con más frecuencia son los **algoritmos de aproximación**. Entre los más conocidos están el algoritmo del vecino más cercano, el algoritmo de inserción y el algoritmo 2-opt, que intentan encontrar una solución cerca de la óptima de manera más eficiente. Estos métodos tienen una complejidad temporal de $O(n^2)$ o $O(n^3)$, lo que los hace más adecuados para grafos de tamaño moderado.

También existen algoritmos de optimización **heurísticos** que pueden aplicarse al TSP, como las **Colonias de Hormigas**, los **Algoritmos Genéticos** y el **Recocido Simulado**. Estos algoritmos buscan soluciones cercanas al óptimo de manera eficiente, aunque no garantizan una solución exacta en todos los casos. En relación al orden podemos identificar:

- **Colonias de Hormigas (Ant Colony Optimization)**: Generalmente, el tiempo de ejecución es $O(n^2)$ a $O(n^3)$ por iteración, donde $n$es el número de ciudades. Sin embargo, el número total de iteraciones puede ser alto, lo que puede hacer que el tiempo total sea considerablemente mayor.
- **Algoritmos Genéticos**: La complejidad puede variar, pero típicamente es $O(n * m)$, donde n es el número de individuos en la población y m es el número de generaciones. La complejidad puede aumentar dependiendo de la cantidad de operaciones de cruce y mutación que se realicen.
- **Recocido Simulado (Simulated Annealing)**: La complejidad es generalmente $O(n * k)$, donde n es el número de ciudades y k es el número de iteraciones o pasos que se realizan en el proceso de enfriamiento. La eficiencia puede depender de la función de enfriamiento y de cómo se generan las soluciones vecinas.

Sin embargo dependiendo dependiendo de la implementación específica y de los parámetros pueden variar significativamente.

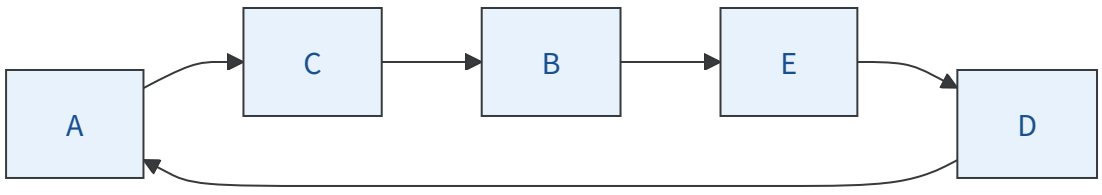
Algoritmo	Tipo	Complejidad Computacional	Características
Fuerza Bruta	Exacto	$O(n!)$	Genera todas las permutaciones posibles. Muy costoso para grandes n.
Algoritmo de Vecino Más Cercano	Heurístico	$O(n^2)$	Heurístico simple; elige siempre el vecino más cercano. No garantiza la mejor solución.

Algoritmo	Tipo	Complejidad Computacional	Características
Algoritmo de Inserción	Heurístico	$O(n^2)$	Construye el ciclo insertando ciudades de forma incremental.
2-opt	Heurístico/Local	$O(n^2)$	Optimiza soluciones existentes al mejorar el recorrido.
Recocido Simulado	Heurístico/Probabilístico	$O(n^2)$ a $O(n^3)$	Algoritmo probabilístico que acepta soluciones subóptimas para evitar mínimos locales.
Colonia de Hormigas	Heurístico/Probabilístico	$O(n^2)$ a $O(n^3)$	Inspirado en el comportamiento de las hormigas; utiliza un sistema de feromonas para guiar la búsqueda.
Programación Dinámica (Held-Karp)	Exacto	$O(n^2 \cdot 2^n)$	Optimización exacta usando programación dinámica; más eficiente que la fuerza bruta.
Algoritmos de Ramificación y Poda	Exacto	$O(n!)$	Método exacto que descarta soluciones no prometedoras. Puede ser más eficiente que la fuerza bruta en ciertos casos.

2.5 Aplicación del Recocido Simulado al TSP

El **recocido simulado** es una técnica heurística inspirada en el proceso físico de enfriamiento de materiales, donde se permite que el sistema explore soluciones subóptimas al principio y, a medida que avanza el proceso, se va enfocando en soluciones de mejor calidad (The MathWorks 2024). En el contexto del TSP, esta técnica se utiliza para encontrar una solución de aproximación para el problema, sin necesidad de realizar una búsqueda exhaustiva, lo que es costoso computacionalmente para grandes instancias del problema (Kirkpatrick, Gelatt Jr, y Vecchi 1983; Aarts, Korst, y Laarhoven 1988; Pepper, Golden, y Wasil 2002).

En el TSP, una **solución** se representa por un orden específico en que las ciudades deben ser visitadas. Por ejemplo, si se tienen 5 ciudades etiquetadas como A, B, C, D y E, una posible solución sería:



Este es un ciclo cerrado, lo que significa que cada ciudad es visitada exactamente una vez y la ruta retorna al punto de inicio, es decir, un ciclo hamiltoniano. Con ello, el algoritmo de recocido simulado aplicado al TSP sigue los siguientes pasos:

1. Solución Inicial

El proceso comienza generando una ruta aleatoria entre todas las ciudades. Esta ruta es utilizada como la **solución inicial**. Luego, se calcula la longitud total de esta ruta, que servirá como base para las iteraciones sucesivas del algoritmo.

2. Generación de Soluciones Vecinas

Una vez que se tiene una solución inicial, el algoritmo genera nuevas soluciones **vecinas** mediante pequeñas modificaciones de la ruta actual. Estas modificaciones pueden ser simples, como el intercambio de dos ciudades en la ruta, la inversión de un segmento de la ruta o la rotación de un subconjunto de ciudades. Estas soluciones vecinas se evalúan en cada iteración del algoritmo.

3. Función de Costo

La **función de costo** corresponde a la longitud total de la ruta generada y algoritmo buscará **minimizar** esta longitud para encontrar la ruta más corta que pase por todas las ciudades. En cada iteración, el algoritmo evalúa si la nueva ruta (vecina) es mejor o peor que la anterior en función de la distancia total recorrida.

2.5.1 Formulación Matemática

La formulación matemática general se puede expresar como:

1. **Función de Probabilidad de Transición**

El recocido simulado utiliza una función de probabilidad de aceptación, que se define matemáticamente como:

$$P(\text{aceptar}) = \min\left\{1, \exp\left(-\frac{\Delta E}{T}\right)\right\}$$

Donde:

- ΔE: Incremento en la energía (en nuestro caso, incremento en la distancia de la ruta)
- T: Temperatura actual

2. **Espacio de Soluciones** Sea S el conjunto de todas las posibles rutas del vendedor viajero.

$$S = \{\text{ruta}_1, \text{ruta}_2, \ldots, \text{ruta}_n\}$$

3. **Función de Energía** La función de energía $E(s)$ es la longitud total de la ruta:

$$E(s) = \sum_{i=1}^{n-1} d(s_i, s_{i+1}) + d(s_n, s_1)$$

Donde $d(s_i, s_{i+1})$ es la distancia entre ciudades consecutivas.

4. **Probabilidad de Transición**

$$P(s' \leftarrow s) = \begin{cases} 1 & \text{si } E(s') < E(s) \\ \exp\left(-\frac{E(s')-E(s)}{T}\right) & \text{si } E(s') \geq E(s) \end{cases}$$

Para el esquema de enfriamiento, el factor asociada al concepto de *temperatura* se reduce según una función de enfriamiento dada por:

$$T_{k+1} = \alpha \cdot T_k$$

Donde:

- $\alpha \in (0, 1)$ es el factor de enfriamiento
- Típicamente $\alpha \approx 0.9$ a 0.99

Luego, la *probabilidad de aceptación* se guía de acuerdo a las siguientes reglas:

- Cuando ΔE es negativo (mejora la solución), $P(\text{aceptar}) = 1$
- Cuando ΔE es positivo (empeora la solución):
  - Alta temperatura: Mayor probabilidad de aceptar
  - Baja temperatura: Menor probabilidad de aceptar

y el *comportamiento probabilístico* esta dado por:

$$\lim_{T \to 0} P(\text{aceptar}) = \begin{cases} 1 & \text{si } \Delta E < 0 \\ 0 & \text{si } \Delta E > 0 \end{cases}$$

En resumen, el algoritmo de recocido simulado presenta una **convergencia probabilística**, lo que significa que puede converger a la solución óptima con probabilidad 1 bajo dos condiciones clave dadas por si la temperatura se reduce lo suficientemente lento y si el número de iteraciones tiende a infinito. Este proceso se basa en una **función de probabilidad** que permite al algoritmo escapar de los mínimos locales mediante una exploración probabilística del espacio de soluciones. En términos teóricos, el algoritmo debe cumplir con la **ergodicidad**, lo que asegura que es capaz de explorar todo el espacio de soluciones, y con el **balance detallado**, una condición que garantiza la convergencia al equilibrio termodinámico. Sin embargo, aunque el algoritmo tiene una alta probabilidad de encontrar la solución óptima, no garantiza que lo haga, y los resultados dependen en gran medida de los parámetros iniciales.

2.5.2 Ejemplo de Implementación en Pseudocódigo

A continuación, se presenta un ejemplo básico de cómo implementar el algoritmo de **recocido simulado** en Python para el TSP. Este código calcula la ruta óptima utilizando el algoritmo descrito previamente.

```
import math
import random

def calcular_distancia_total(ruta, distancias):
    """Calcula la longitud total de una ruta."""
    distancia_total = 0
    for i in range(len(ruta) - 1):
```

```
        distancia_total += distancias[ruta[i]][ruta[i+1]]
    # Añadir distancia de vuelta al punto de origen
    distancia_total += distancias[ruta[-1]][ruta[0]]
    return distancia_total

def generar_solucion_vecina(ruta):
    """Genera una solución ligeramente modificada."""
    nueva_ruta = ruta.copy()
    # Intercambiar dos ciudades al azar
    i, j = random.sample(range(len(ruta)), 2)
    nueva_ruta[i], nueva_ruta[j] = nueva_ruta[j], nueva_ruta[i]
    return nueva_ruta

def recocido_simulado_tsp(ciudades, distancias,
         temperatura_inicial=10,
         factor_enfriamiento=0.995,
         iteraciones=100):
    # Generar ruta inicial aleatoria
    ruta_actual = ciudades.copy()
    random.shuffle(ruta_actual)

    mejor_ruta = ruta_actual.copy()
    temperatura = temperatura_inicial

    for _ in range(iteraciones):
        # Generar solución vecina
        nueva_ruta = generar_solucion_vecina(ruta_actual)

        # Calcular diferencia de distancias
        distancia_actual = calcular_distancia_total(ruta_actual, distancias)
        nueva_distancia = calcular_distancia_total(nueva_ruta, distancias)
        delta_distancia = nueva_distancia - distancia_actual

        # Criterio de aceptación
        if delta_distancia < 0 or random.random() < math.exp(-delta_distancia
        / temperatura):
            ruta_actual = nueva_ruta

            # Actualizar mejor ruta si es necesario
            if nueva_distancia < calcular_distancia_total(mejor_ruta, distancias):
                mejor_ruta = nueva_ruta

        # Reducir temperatura
        temperatura *= factor_enfriamiento

    return mejor_ruta, calcular_distancia_total(mejor_ruta, distancias)
```

En cuanto a la complejidad computacional, para **n** ciudades, hay **(n-1)!** rutas posibles. La complejidad crece conforme se aumenta el número de ciudades. Por ejemplo, con 5 ciudades hay 24 posibles rutas, pero con 10 ciudades el número de rutas posibles se eleva a 362,880, y con 20 ciudades, el número de soluciones supera los 2×10^{18}.

El recocido simulado presenta varias ventajas clave para la resolución del Problema del Vendedor Viajero (TSP). Una de sus principales ventajas es que escapa de óptimos locales, lo que lo distingue de otros algoritmos de búsqueda local. Mientras que estos últimos pueden quedar atrapados en soluciones subóptimas, el recocido simulado permite explorar soluciones menos prometedoras al principio del proceso, aumentando así las posibilidades de encontrar una solución globalmente mejor. Además, el recocido simulado se basa en una exploración probabilística, lo que significa que el algoritmo puede evaluar diferentes caminos de manera aleatoria, evitando quedar atrapado en un óptimo local y permitiendo una búsqueda más amplia del espacio de soluciones. Finalmente, una de las mayores ventajas de este enfoque es su capacidad para ser utilizado en grandes instancias del problema, donde los métodos exactos resultan computacionalmente impracticables. Esto lo convierte en una herramienta especialmente útil cuando el número de ciudades a considerar es elevado, permitiendo obtener soluciones razonablemente buenas en un tiempo de cómputo razonable.

Sin embargo, aunque el recocido simulado es una técnica eficiente, es importante tener en cuenta algunas limitaciones:

- **No garantiza la solución óptima**: Aunque es eficaz en encontrar soluciones cercanas a la óptima, no siempre garantiza que se encontrará la mejor solución global.
- **Dependencia de parámetros**: El rendimiento del algoritmo depende en gran medida de los parámetros, como la **temperatura inicial** y el **factor de enfriamiento**. Estos parámetros deben ser ajustados cuidadosamente para cada instancia del problema.
- **Ajuste experimental**: Es recomendable realizar un ajuste experimental de los parámetros para obtener los mejores resultados posibles.

2.5.3 Variantes y mejoras

Existen varias maneras de mejorar el recocido simulado para resolver el TSP de manera más eficiente:

- Se pueden emplear **estrategias de generación de vecinos** más sofisticadas que exploren el espacio de soluciones de forma más eficiente.
- Se pueden implementar **esquemas de enfriamiento adaptativos**, que ajusten la temperatura de manera dinámica durante la ejecución del algoritmo.
- También es posible **combinar el recocido simulado con otros algoritmos** de optimización, como los algoritmos genéticos o de colonia de hormigas, para mejorar aún más los resultados.

2.6 Extra: Pequeña explicación de Brunch & Cut

Branch and Cut es una técnica híbrida utilizada en la optimización combinatoria (Dantzig, Fulkerson, y Johnson 1954; Applegate 2006). Combina dos enfoques principales:

1. **Branch and Bound**: Divide el espacio de soluciones en subproblemas más pequeños para encontrar soluciones óptimas.

2. **Cortes (Cuts)**: Añade restricciones adicionales al modelo para reducir el espacio factible y acelerar la convergencia.

En el contexto del **Problema del Vendedor Viajero (TSP)**, el algoritmo busca encontrar la ruta de menor distancia que visite todas las ciudades exactamente una vez y regrese a la ciudad inicial.

2.6.1 Descripción del algoritmo

1. **Formulación inicial**:
   - Se crea un modelo de programación lineal con las restricciones básicas (visitas únicas y conectividad).
   - Inicialmente, se permite la formación de subtours.
2. **Relajación lineal**:
   - El modelo se resuelve relajando las variables $x_{i,j}$ a valores continuos en $[0, 1]$.
3. **Generación de cortes**:
   - Si la solución contiene subtours, se añaden restricciones adicionales para eliminarlos, conocidas como **cortes de subtours**:

$$\sum_{i \in S} \sum_{j \in S, i \neq j} x_{i,j} \leq |S| - 1, \quad \forall S \subseteq \{1, \ldots, n\}, |S| \geq 2$$

Con S un subconjunto de ciudades que forman un ciclo inválido.

4. **Búsqueda en el árbol**:
   - Si la solución actual aún no es entera, se divide el problema en subproblemas (rama izquierda y derecha del árbol) asignando $x_{i,j} = 0$ o $x_{i,j} = 1$ para ciertas variables.
   - Se resuelve cada subproblema iterativamente.
5. **Criterio de parada**:
   - El algoritmo termina cuando encuentra una solución entera factible y no hay más ramas o cortes posibles.

2.6.2 Ejemplo de aplicación con solver

A continuación, se muestra cómo se implementa el algoritmo Branch and Cut en Python utilizando la librería PuLP (Mitchell, OSullivan, y Dunning 2011):

```
import pulp as pl

def branch_and_cut(problema):
    n = len(problema.ciudades)
    modelo = pl.LpProblem("TSP_Branch_and_Cut", pl.LpMinimize)

    # Variables de decisión
    x = pl.LpVariable.dicts("x", ((i, j) for i in range(n)
     for j in range(n) if i != j), cat="Binary")

    # Función objetivo
    modelo += pl.lpSum(x[i, j] * problema.matriz_distancias[i, j]
     for i in range(n) for j in range(n) if i != j)

    # Restricciones de entrada/salida
    for i in range(n):
```

```
        modelo += pl.lpSum(x[i, j] for j in range(n) if i != j) == 1
        modelo += pl.lpSum(x[j, i] for j in range(n) if i != j) == 1

    # Variables auxiliares para subtours
    u = pl.LpVariable.dicts("u", range(n), lowBound=0, cat="Continuous")
    for i in range(1, n):
        for j in range(1, n):
            if i != j:
                modelo += u[i] - u[j] + n * x[i, j] <= n - 1

    # Resolver
    modelo.solve()
    ruta = [0]
    actual = 0

    while len(ruta) < n:
        for j in range(n):
            if j not in ruta and pl.value(x[actual, j]) > 0.5:
                ruta.append(j)
                actual = j
                break

    return ruta, problema.distancia_ruta(ruta)
```

En cuanto al orden, el algoritmo Branch and Cut aplicado al Problema del Vendedor Viajero (TSP) tiene una complejidad computacional en el peor caso exponencial, similar a otros métodos exactos como Branch and Bound, con un orden de complejidad que puede ser $O(n!)$. Sin embargo, en la práctica, la eficiencia mejora significativamente gracias a la eliminación de soluciones inviables mediante cutting planes, lo que reduce el número de nodos a explorar. Esto puede hacer que la complejidad efectiva sea mucho más baja, en algunos casos acercándose a $O(n^2)$ o mejor, dependiendo de la calidad de los cortes generados. Aunque Branch and Cut es eficaz para instancias medianas a grandes, sigue siendo un algoritmo exacto, lo que lo hace poco práctico para problemas de muy gran escala, donde es más realista utilizar enfoques heurísticos o aproximados.

Entre las ventajas del algoritmo, se destaca que puede encontrar la solución óptima global del problema, además, reduce el espacio de búsqueda mediante cortes efectivos, lo que mejora la eficiencia del proceso. Sin embargo, también tiene limitaciones importantes como el tiempo de cómputo y requisitos de memoria cuando se enfrenta a problemas grandes, lo que puede dificultar su aplicación en instancias de gran tamaño. Además, su rendimiento depende en gran medida de la implementación del solver y de la calidad de los cortes generados, lo que puede afectar la efectividad del algoritmo en ciertos casos.

3 Desarrollo simulación, implementación y resultados

Para probar la resolución del problema del vendedor viajero se desarrollará una clase sintética que simule grafos con rutas para luego encontrar la ruta más corta probando 3 estrategias de optimización. La implementación será realizada utilizando el lenguaje de programación *python* y solvers especializados para problemas de optimización.

3.1 Estructura del proyecto de simulación

El simulador se organizó de acuerdo a la siguiente estructura y la documentación quedó disponible en GitHub - Cinthya Leonor Vergara Silva.

Dado que la complejidad del problema recae sobre la estructura, es decir, sobre la combinación ruta-destinos y no sobre el tipo de datos o características de los datos, el problema se enfocará en revisar el funcionamiento de los algoritmo en base a datos sintéticos con la estructura general de un problema de identificación de ruta más corta en dentro de un grafo conexo y no dirigido.

Clases Principales

- **TSPSintetico**:
  - Genera instancias aleatorias del problema del vendedor viajero.
  - Calcula la matriz de distancias entre ciudades y la distancia de rutas específicas.

Funciones Implementadas

1. `recocido_simulado`:
   - Resuelve el TSP usando el algoritmo de Recocido Simulado.
   - Permite ajustar la temperatura inicial, el factor de enfriamiento y el número de iteraciones.
   - Devuelve la mejor ruta encontrada y la distancia total.
   - Almacena el historial de temperaturas y valores de la función objetivo para análisis iterativo.
2. `comparar_esquemas_reduccion`:
   - Esquemas de Reducción de Temperatura
     - **Reducción Exponencial**: Enfriamiento gradual, geométrico
     - **Reducción Logarítmica**: Enfriamiento más lento para más exploración
     - **Reducción Lineal**: Disminución de temperatura consistente y predecible
   - Genera visualizaciones
3. `branch_and_cut_iterativo`:
   - Utiliza PuLP para resolver el TSP mediante programación lineal.
   - Encuentra la solución óptima global para el problema dado.
4. `comparar_rendimiento`
   - Compara el progreso iterativo del Recocido Simulado
   - Contrasta con la solución óptima de Branch and Cut
   - Genera visualizaciones de convergencia

3.2 Esquema de simulaciones

Para realizar la simulación llamamos a la función *comparar_soluciones* y le damos instancias sintetica a través de la clase *TSPSintetico*. El código queda anexo al documento y se deja a modo de referencia el llamdo a las funciones:

```
from simulacion_recocido_branch_and_cut_plots import *

if __name__ == "__main__":
    # Crear instancia del problema
    problema = TSPSintetico(num_ciudades=15, semilla=42)
```

```
mostrar_resultados_esquemas(problema) # Resultados Recocido
comparar_esquemas_reduccion(problema) # Resultados Recocido Plots

optimo_bc, tiempo_bc = branch_and_cut_iterativo(problema)
print(f"Branch and Cut - Distancia óptima: {optimo_bc}, Tiempo: {tiempo_bc:.2f} segundos")

comparar_rendimiento(42, 15)
```

Se prueba con 15 ciudades ya que luego de 15 Brunch and Cut comienza a requerir una gran cantidad de tiempo de computo. Para efectos de este informe en la ciudad 15 ya es posible ver las diferencias de orden entre los dos algoritmos.

3.3 Análisis recocido simulado para distintas configuraciones

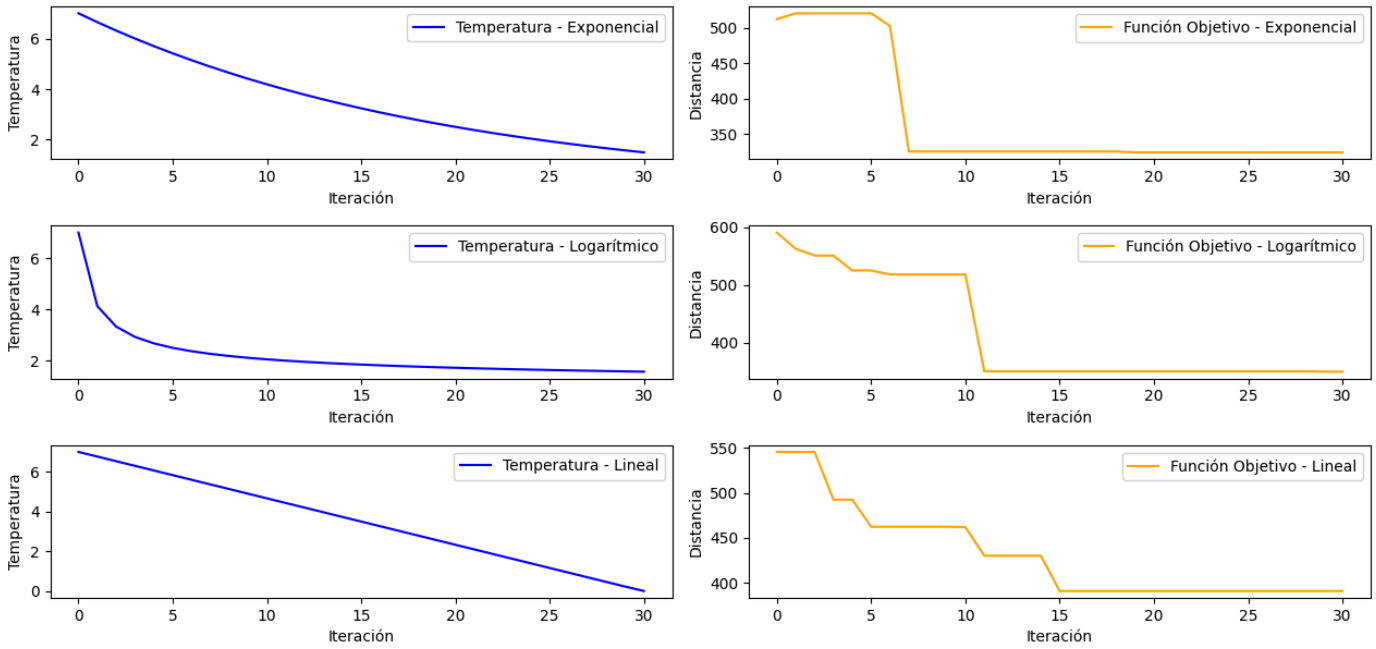
Se prabaron 3 distintas configuraciones de reducción tomando las variantes exponencial, logarítmica y lineal a través de la medición de los parámetros *temperatura* y el resultado de la *función objetivo* (busncado la mínima distancia).

En el **esquema de reducción exponencial**, el rendimiento comienza con una distancia relativamente alta de **624.54**, pero a lo largo de las primeras iteraciones se observa una mejora notable. La distancia se reduce rápidamente a **558.78** y se estabiliza en **454.78** entre las iteraciones 19 y 25. Finalmente, en la iteración 30, el algoritmo alcanza la mejor distancia de **435.22**. Este esquema muestra una reducción significativa en las primeras iteraciones, lo que sugiere que es eficaz al principio del proceso de optimización, aunque su capacidad de mejora se desacelera al final.

El **esquema logarítmico** tiene una tasa de reducción más lenta en las primeras iteraciones en comparación con el exponencial. Comienza con una distancia inicial de **687.42** y disminuye gradualmente en las primeras iteraciones, estabilizándose en **391.12** entre las iteraciones 11 y 23. A partir de la iteración 27, se observa una mejora importante, alcanzando la mejor distancia de **365.89** en la iteración 30. Aunque la mejora inicial es más gradual, este esquema es efectivo a largo plazo, con una notable mejora hacia el final del proceso.

El **esquema lineal** muestra una reducción constante en la distancia, comenzando con un valor de **518.07** y estabilizándose en **484.22** hasta la iteración 10. A partir de la iteración 11, la distancia comienza a reducirse de manera continua y significativa, alcanzando la mejor distancia de **314.75** en la iteración 30. Este esquema ofrece una mejora constante a lo largo del tiempo, con una reducción progresiva que finalmente resulta en el mejor valor encontrado.

El resulado muestra que el **esquema lineal** alcanza la mejor distancia final de **314.75**, superando tanto al esquema exponencial (435.22) como al logarítmico (365.89). Este rendimiento estable lo convierte en el esquema más efectivo para este problema en particular. En cambio, el **esquema exponencial** destaca por su capacidad para realizar mejoras rápidas en las primeras iteraciones, pero con una estabilización más temprana que limita su rendimiento final. Por otro lado, el **esquema logarítmico**, aunque muestra una reducción más lenta en sus primeras fases, experimenta una mejora significativa al final del proceso, lo que puede ser útil en casos donde se busque una optimización más gradual.



Resultado simulación variantes recocido

Al observar los gráficos, podemos interpretar lo siguiente:

1. **Temperatura vs Iteración (Exponencial, Logarítmica, Lineal)**:

Estos gráficos muestran cómo varía el parámetro de temperatura a lo largo de las iteraciones del algoritmo de recocido simulado. Los gráficos con esquemas exponencial y logarítmico muestran una disminución más rápida de la temperatura, mientras que el gráfico con esquema lineal presenta una disminución más gradual. La elección de la función de actualización de temperatura tiene un impacto significativo en el rendimiento del algoritmo. Los esquemas de enfriamiento exponencial y logarítmico son más comunes, ya que permiten una exploración más amplia al principio y luego se enfocan en las mejores soluciones.

2. **Función Objetivo vs Iteración (Exponencial, Logarítmico, Lineal)**:

Estos gráficos muestran cómo cambia la función objetivo (el valor que estamos tratando de minimizar, como la distancia total del recorrido en el TSP) a lo largo de las iteraciones. En los esquemas exponencial y logarítmico, la función objetivo disminuye más rápidamente, lo que indica que estos esquemas de enfriamiento son capaces de encontrar soluciones de mejor calidad en menos tiempo. En el esquema lineal, la disminución de la función objetivo es más gradual, lo que sugiere que este enfoque es menos eficiente para converger rápidamente hacia soluciones de alta calidad.

Con ello podemos ver que los esquemas de enfriamiento exponencial y logarítmico muestran ser más efectivos que el esquema lineal en el algoritmo de recocido simulado, ya que permiten una exploración más eficiente del espacio de soluciones. Los gráficos de la función objetivo indican que las variantes exponencial y logarítmica encuentran soluciones de menor costo más rápidamente que el esquema lineal.

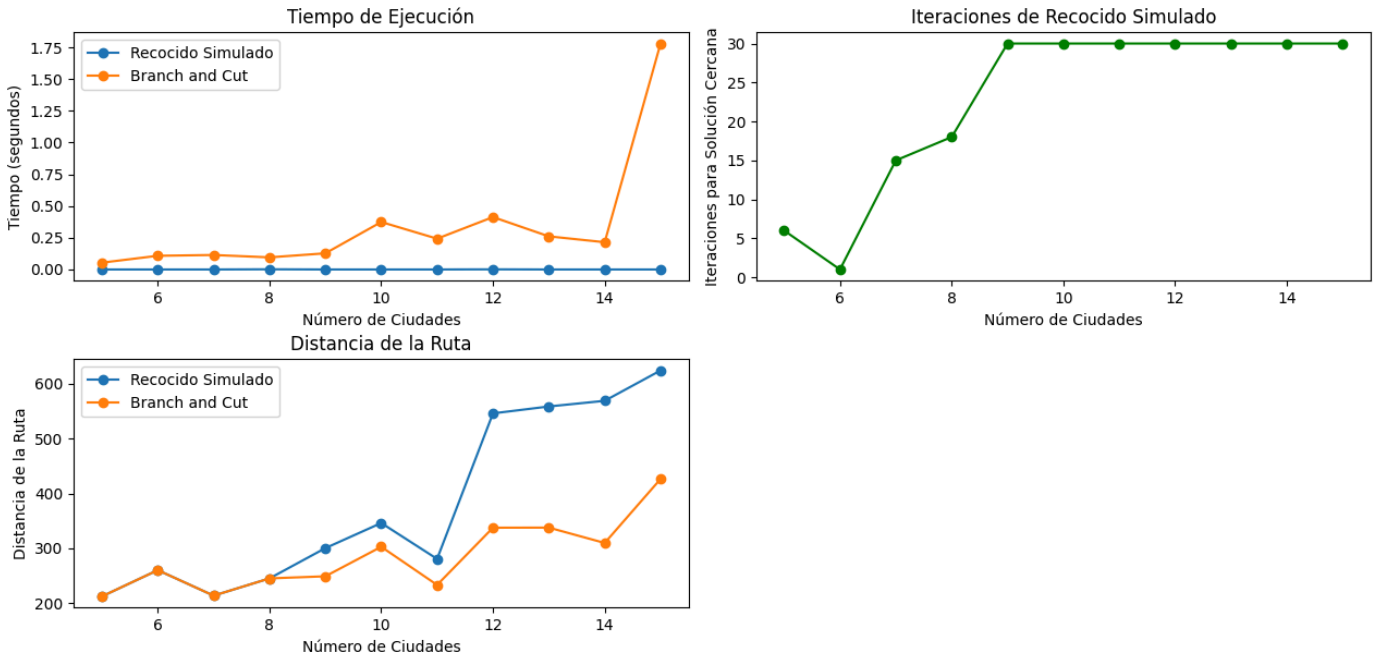
3.4 Comparación con Branch and Cut

Luego se realiza la comparación entre 5 y 15 ciudades entre recocido simulado (**SA**) y branch and cut (**BC**) obteniendo los siguientes resultados:

Num_Ciudades	Distancia_SA	Distancia_BC	Tiempo_SA	Tiempo_BC	Iteracion_Optimo_SA
5	212.61	212.61	0.00	0.05	6
6	260.41	260.41	0.00	0.11	1
7	214.00	214.00	0.00	0.11	15
8	245.40	245.40	0.00	0.10	18
9	300.47	249.23	0.00	0.13	30
10	346.17	302.93	0.00	0.37	30
11	280.84	233.49	0.00	0.24	30
12	545.68	337.64	0.00	0.41	30
13	558.26	337.82	0.00	0.26	30
14	568.56	309.84	0.00	0.21	30
15	623.76	426.36	0.00	1.78	30

Los resultados muestran que el algoritmo de recocido gana en tiempo notablemente pero con mínimos mayores a los de branch and cut, además, dependiendo de la iteración, al tomar instancias aleatorias, los resultados del recocido son variables.

Graficamente podemos ver:



Resultado simulación para comparación de modelos

1. Gráfico de Tiempo de Ejecución:

Este gráfico muestra el tiempo que tardan los dos algoritmos, *Recocido Simulado* (Simulated Annealing) y *Branch and Cut*, en ejecutar el problema del Vendedor Viajero (TSP) a medida que aumenta el número de ciudades. La línea azul representa el tiempo de ejecución del algoritmo de *Recocido Simulado*, mientras que la línea naranja muestra el tiempo de ejecución del algoritmo *Branch and Cut*. Se observa que, a medida que aumenta el número de ciudades, el tiempo de ejecución de ambos algoritmos también crece. Sin embargo, el algoritmo *Branch and Cut* presenta un aumento mucho más pronunciado en el tiempo de ejecución en comparación con el algoritmo *Recocido Simulado*. Esto indica que el algoritmo *Recocido Simulado* es más eficiente y escalable cuando se enfrenta a tamaños de problema más grandes (más ciudades), mientras que el algoritmo *Branch and Cut* se vuelve significativamente más costoso computacionalmente a medida que crece el tamaño del problema. La curva relativamente plana del algoritmo *Recocido Simulado* sugiere que este puede manejar un número creciente de ciudades sin un aumento dramático en el tiempo de ejecución, lo que lo convierte en una opción más adecuada para instancias grandes del TSP.

2. Gráfico de Iteraciones del RecocidoSimulado:

Este gráfico muestra el número de iteraciones realizadas por el algoritmo de *Recocido Simulado* a medida que aumenta el número de ciudades. La línea verde representa el número de iteraciones del algoritmo *Recocido Simulado*. Se observa que el número de iteraciones aumenta de forma constante a medida que crece el número de ciudades, pero la tasa de aumento es relativamente moderada en comparación con el crecimiento exponencial que podría esperarse para problemas de optimización más complejos. El aumento consistente y gradual en el número de iteraciones sugiere que el algoritmo *Recocido Simulado* es capaz de explorar eficientemente el espacio de soluciones y converger a una solución casi óptima, incluso para instancias más grandes del problema. Esto indica que el algoritmo *Recocido Simulado* está bien adaptado para resolver el problema del Vendedor Viajero, ya que puede encontrar soluciones buenas de manera consistente sin un aumento abrumador en la complejidad computacional a medida que el tamaño del problema crece.

3. Gráfico de Distancia de la Ruta:

Este gráfico muestra la distancia total de la ruta encontrada por los dos algoritmos, *Recocido Simulado* y *Branch and Cut*, a medida que aumenta el número de ciudades. La línea azul representa la distancia total de la ruta encontrada por el algoritmo de *Recocido Simulado*, mientras que la línea naranja representa la distancia total de la ruta encontrada por el algoritmo *Branch and Cut*. Se observa que la distancia total de la ruta aumenta para ambos algoritmos a medida que el número de ciudades crece, lo cual es esperado, ya que el problema se vuelve más complejo con más ciudades para visitar. Las dos líneas siguen un patrón muy similar, lo que indica que ambos algoritmos son capaces de encontrar rutas de calidad similar, con una ligera ventaja para el algoritmo *Branch and Cut* en términos de la distancia total de la ruta. La similitud en el rendimiento de ambos algoritmos en cuanto a la distancia total sugiere que ambos son efectivos para encontrar soluciones cercanas al óptimo para el problema del TSP, y la elección entre uno u otro puede depender de otros factores, como el tiempo de ejecución y la complejidad computacional, como se discutió anteriormente.

En resumen, los gráficos proporcionan una comparación completa del rendimiento de los algoritmos *Recocido Simulado* y *Branch and Cut* para resolver el Problema del Vendedor Viajero. El algoritmo *Recocido Simulado* parece ser más eficiente y escalable, con un aumento moderado en el tiempo de ejecución y un número

consistente de iteraciones a medida que el tamaño del problema crece. Ambos algoritmos son capaces de encontrar rutas de calidad similar, pero el algoritmo *Recocido Simulado* podría ser la opción preferida para instancias más grandes del TSP debido a su mejor escalabilidad y eficiencia.

4 Conclusiones

El problema del vendedor viajero presenta varios desafíos clave. En primer lugar, la **naturaleza combinatorial** que produce que a medida que aumenta el número de ciudades hace que la búsqueda de soluciones óptimas sea inviable en un tiempo razonable para grafos grandes. Además, el TSP es un **problema NP-completo**, lo que significa que no se conoce un algoritmo polinomial que pueda resolverlo de manera eficiente en todos los casos. A pesar de estos desafíos, el problema tiene importantes **aplicaciones del mundo real**, especialmente en áreas como la logística, la planificación de rutas, el diseño de circuitos y la secuenciación de ADN, donde encontrar la ruta más eficiente tiene implicaciones significativas.

Desde su origen, el ciclo Hamiltoniano y el problema del vendedor viajero representan un desafío en el campo de la optimización combinatoria dada su naturaleza combinatorial. A pesar de su complejidad, estos problemas tienen aplicaciones prácticas y son un área activa de investigación importante para el desarrollo de actividades en muchos casos cotidianas, como los problemas vinculados al transporte y descpacho de bienes o servicios. A pesar de su complejidad computacional, es posible abordar el problema desde el uso de fuerza bruta para grafos pequeños, algoritmos de aproximación para grafos medianos hasta el uso algoritmos heurísticos para grandes instancias. Es importante siempre analizar el compromiso entre precisión y tiempo de cómputo, adaptando la estrategia en función de las características del problema específico.

En particular, el algoritmo de recocido simulado es una herramienta que puede encontrar soluciones cercanas a la óptima para instancias grandes del problema. Sin embargo, el algoritmo no garantiza la mejor solución y depende de parámetros que deben ser ajustados adecuadamente para cada caso, además al tener una componente aletoría, los resultados pueden ir variando en cada ejecución o prueba. En general, es una buena opción para problemas donde las soluciones exactas son inviables debido al tamaño del problema.

Por su parte, algoritmos exactos como Brunch and Cut, si bien aseguran llegar a un óptimo, son poco factibles de implementar en aplicaciones reales ya que el tiempo de cómputo crece exponencialmente y se vuelve muy grande en pocas instancias. En el caso de este ejercicio, con 30 instancias en 1 hora no lograba converger, hasta 15 ciudades tomaba un tiempo razonable para evaluar rendimiento y poder entregar el informe dentro del plazo.

La elección del esquema de enfriamiento es un factor a tener en consideración que puede influir significativamente en el rendimiento del algoritmo, por lo que es importante ajustarlo adecuadamente según las características del problema a resolver. Además es importante considerar la necesidad de encontra una solución exacta tiene el valor suficiente respecto al esfuerzo en tiempo de computo necesario, si una solución sub-optima permite tomar mejores decisiones de manera oportuna, una solución heurística se convierte en una alternativa factible y conveniente.

Referencias

Aarts, Emile HL, Jan HM Korst, y Peter JM van Laarhoven. 1988. «A quantitative analysis of the simulated annealing algorithm: A case study for the traveling salesman problem». *Journal of Statistical Physics* 50: 187-206.

Applegate, David L. 2006. *The traveling salesman problem: a computational study*. Vol. 17. Princeton university press.

Bektas, Tolga. 2006. «The multiple traveling salesman problem: an overview of formulations and solution procedures». *omega* 34 (3): 209-19.

Bertsimas, Dimitris, y Louis H Howell. 1993. «Further results on the probabilistic traveling salesman problem». *European Journal of Operational Research* 65 (1): 68-95.

Bertsimas, Dimitris, y John N Tsitsiklis. 1997. *Introduction to linear optimization*. Vol. 6. Athena Scientific Belmont, MA.

Dantzig, George, Ray Fulkerson, y Selmer Johnson. 1954. «Solution of a large-scale traveling-salesman problem». *Journal of the operations research society of America* 2 (4): 393-410.

Gendreau, Michel, Jean-Yves Potvin, et al. 2010. *Handbook of metaheuristics*. Vol. 2. Springer.

Kirkpatrick, Scott, C Daniel Gelatt Jr, y Mario P Vecchi. 1983. «Optimization by simulated annealing». *science* 220 (4598): 671-80.

Laporte, Gilbert. 1992. «The traveling salesman problem: An overview of exact and approximate algorithms». *European Journal of Operational Research* 59 (2): 231-47.

Menger, K. 1928. «Ein theorem über die bogenlänge». *Anzeiger—Akademie der Wissenschaften in Wien—Mathematisch-naturwissenschaftliche Klasse* 65: 264-66.

Mitchell, Stuart, Michael OSullivan, y Iain Dunning. 2011. «Pulp: a linear programming toolkit for python». *The University of Auckland, Auckland, New Zealand* 65: 25.

Pepper, Joshua W, Bruce L Golden, y Edward A Wasil. 2002. «Solving the traveling salesman problem with annealing-based heuristics: a computational study». *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans* 32 (1): 72-77.

Schrijver, Alexander. 2005. «On the History of Combinatorial Optimization (Till 1960)». En *Discrete Optimization*, editado por K. Aardal, G. L. Nemhauser, y R. Weismantel, 12:1-68. Handbooks en Operations Research y Management Science. Elsevier. https://doi.org/https://doi.org/10.1016/S0927-0507(05)12001-5.

Singh, Karanjot, Simran Kaur Bedi, y Prerna Gaur. 2020. «Identification of the most efficient algorithm to find Hamiltonian Path in practical conditions». En *2020 10th International Conference on Cloud Computing, Data Science & Engineering (Confluence)*, 38-44. https://doi.org/10.1109/Confluence47617.2020.9058283.

The MathWorks, Inc. 2024. «What Is Simulated Annealing?» 2024. https://es.mathworks.com/help/gads/what-is-simulated-annealing.html.