



Universidad  
Andrés Bello®  
Conectar • Innovar • Liderar

**FACULTAD DE INGENIERÍA**

**INGENIERÍA CIVIL INFORMÁTICA**

# **Análisis del Problema del Circuito Hamiltoniano: Una Perspectiva Matemática**

Christian Garrido

Profesor:

Gustavo Gática

Santiago, Chile

2023

# índice general

PORTADA .....	1
índice general .....	2
INTRODUCCIÓN.....	3
EXPLICACIÓN DEL PROBLEMA.....	4
DEFINICIONES.....	4
MODELO MATEMÁTICO .....	6
IMPLEMENTACIÓN DEL MODELO EN PYTHON .....	8
EXPERIMENTOS COMPUTACIONALES .....	14
• Implementación del algoritmo de “Held-Karp”:	13
• Ejecución del algoritmo:	14
• Gráficos de complejidad	15
CONCLUSIONES.....	16
REFERENCIAS BIBLIOGRÁFICAS .....	17

# INTRODUCCIÓN

¿Qué tienen en común un viajero incansable y un problema matemático? El “**TSP**” (Problema del Vendedor Viajero) y los Ciclos Hamiltonianos son dos conceptos que desafían nuestra comprensión y nos llevan a explorar las complejidades de la optimización. Estos conceptos, que surgen en el campo de la teoría de grafos, han capturado la atención de matemáticos, informáticos y científicos de datos durante décadas.

Desde su primera aparición en la literatura matemática, los ciclos Hamiltonianos y el “**TSP**” han sido objeto de estudio y fascinación, presentando desafíos intelectuales y aplicaciones prácticas en diversos campos. A lo largo de la historia, han surgido múltiples enfoques para abordar estos problemas, desde algoritmos exactos hasta técnicas heurísticas y algoritmos de aproximación. A pesar de su aparente simplicidad en su formulación, encontrar una solución eficiente para grandes grafos sigue siendo un desafío en la actualidad. El problema del vendedor viajero es un problema famoso de tipo “**NP-hard**”, lo que significa que a medida que el número de nodos aumenta, tomará más tiempo para poder resolverse debido a la complejidad computacional del problema. A lo largo de este estudio, se explorará el fascinante mundo del “**TSP**” y los Ciclos Hamiltonianos, desde sus fundamentos matemáticos hasta sus aplicaciones modernas, abordando los desafíos en la búsqueda de soluciones eficientes para estos problemas complejos.

## EXPLICACIÓN DEL PROBLEMA

Los ciclos Hamiltonianos y el problema del vendedor viajero son dos conceptos que surgen en el campo de la teoría de grafos. Un ciclo hamiltoniano es un ciclo que visita cada vértice de un grafo exactamente una vez. El problema del ciclo hamiltoniano consiste en determinar si un grafo dado tiene la propiedad de tener un ciclo hamiltoniano, mientras que el problema de vendedor viajero (TSP, por sus siglas en inglés.) consiste en lo siguiente: “Dado un conjunto de ciudades a visitar, un agente viajero debe encontrar la ruta que le permita pasar por todas las ciudades una única vez y volviendo al punto de partida, asegurando que el costo del recorrido sea mínimo” Se está hablando de un costo mínimo, entonces se busca minimizar costos de viajes, tiempo de viaje o distancia dependiendo de la naturaleza del problema. El TSP es un problema de optimización combinatoria, esto quiere decir que se busca encontrar la solución óptima en un conjunto finito de posibles soluciones. La relación entre estos dos problemas radica en encontrar el ciclo hamiltoniano optimo en un grafo ponderado.

## DEFINICIONES

Sea  $G = (V, E)$  un grafo conexo no dirigido, siendo  $V = \{1, \dots, n\}$  el conjunto de vértices y  $E$  el conjunto de aristas. Donde los vértices representan las ciudades que se deben visitar,  $v_1$  la ciudad de origen y final. Cada arista  $(i, j)$  tendrá asociado un valor no negativo  $c_{ij}$  que representa el costo entre las ciudades  $i$  y  $j$ .

- $n!$  = Número de trayectorias posibles
- $\frac{(n-1)!}{2}$  = Número de trayectorias posibles en un grafo simétrico con un nodo de partida.

Estas definiciones pueden ser probadas en el siguiente ejemplo. (Ver figura 1.1)

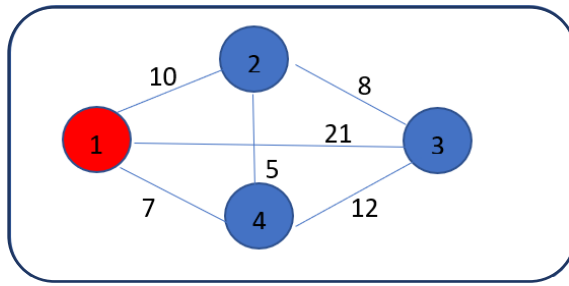


Figura 1.1 elaboración propia

Se observa que  $n = 4 \longrightarrow \frac{(4-1)!}{2} = 3$  ; por lo que se obtiene las siguientes trayectorias:

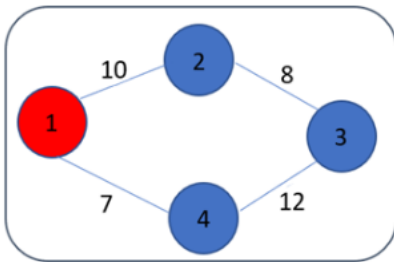


Figura 1.2 Elaboración propia

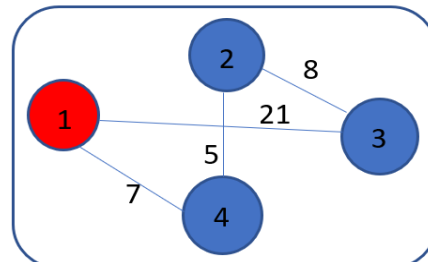


Figura 1.3 Elaboración propia

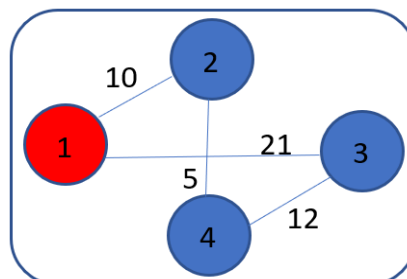


Figura 1.4 Elaboración propia

## MODELO MATEMÁTICO

- **Variables de decisión:**

$$x_{ij} = \begin{cases} 1, & \text{Si la ciudad } i \text{ est\u00e1 conectada a la ciudad } j \\ 0, & \text{en el caso contrario} \end{cases}$$

- **Funci\u00f3n objetivo:**

$$Z = \min \left( \sum_{i=1}^n \sum_{\substack{j=1 \\ j \neq i}}^n c_{ij} \cdot x_{ij} \right) \text{ con } c_{ij} = \infty \text{ para } i = j$$

- **Restricciones:**

- Por cada nodo debe Salir un solo arco, se abandona una ciudad por un solo camino.

$$\sum_{\substack{j=1 \\ i \neq j}}^n x_{ij} = 1, \quad 1 \leq i \leq n \cap i \neq j$$

- Cada nodo debe recibir un solo arco, cada ciudad se llega por un solo camino.

$$\sum_{\substack{i=1 \\ i \neq j}}^n x_{ij} = 1, \quad 1 \leq j \leq n \cap i \neq j$$

- Utilizando solo las 2 restricciones anteriores, puede causar la aparición de un subciclo (subtour), en lugar de los circuitos completos que abarquen los  $n$  nodos. Para evitar lo mencionado se agrega una tercera restricción.

$$c_i + 1 \geq c_j + M(1 - x_{ij})$$

La restricción establece que si el arco  $(i, j)$  está presente en la solución (es decir,  $x_{ij} = 1$ ), entonces el costo acumulado de llegar a la ciudad  $i$  ( $c_i$ ) debe ser al menos 1 unidad mayor que el costo acumulado de llegar a la ciudad  $j$  ( $c_j$ ) más el valor de  $M$ . Esto ayuda a evitar la formación de circuitos o subtours más cortos dentro de la solución del TSP.

- **Parámetros:**

- 1) Número de nodos ( $n$ ): El número total de nodos en el grafo, que representa las ciudades (nodos) que el viajero debe visitar.
- 2) Matriz de costos ( $c_{ij}$ ): Matriz que representa los costos asociados a viajar entre cada par de ciudades (nodos) en el grafo.

# IMPLEMENTACIÓN DEL MODELO EN PYTHON

Previo a la implementación del modelo en Python, se debe trabajar sobre un conjunto de datos. Se creará un conjunto de 10 ciudades que corresponderán a los nodos del grafo (ver figura 1.5).

```
Los conjuntos de datos

Primero inventaremos datos aleatorios

[25]: n = 10 #10 ciudades
      ciudades = [i for i in range(n)]
      aristas = [(i,j) for i in ciudades for j in ciudades if i!=j]
      ciudades, print(aristas)

[(0, 1), (0, 2), (0, 3), (0, 4), (0, 5), (0, 6), (0, 7), (0, 8), (0, 9), (1, 0), (1, 2), (1, 3), (1, 4), (1, 5), (1, 6), (1, 7), (1, 8), (1, 9), (2, 0), (2, 1), (2, 3), (2, 4), (2, 5), (2, 6), (2, 7), (2, 8), (2, 9), (3, 0), (3, 1), (3, 2), (3, 4), (3, 5), (3, 6), (3, 7), (3, 8), (3, 9), (4, 0), (4, 1), (4, 2), (4, 3), (4, 5), (4, 6), (4, 7), (4, 8), (4, 9), (5, 0), (5, 1), (5, 2), (5, 3), (5, 4), (5, 6), (5, 7), (5, 8), (5, 9), (6, 0), (6, 1), (6, 2), (6, 3), (6, 4), (6, 5), (6, 7), (6, 8), (6, 9), (7, 0), (7, 1), (7, 2), (7, 3), (7, 4), (7, 5), (7, 6), (7, 8), (7, 9), (8, 0), (8, 1), (8, 2), (8, 3), (8, 4), (8, 5), (8, 6), (8, 7), (8, 9), (9, 0), (9, 1), (9, 2), (9, 3), (9, 4), (9, 5), (9, 6), (9, 7), (9, 8)]

[25]: ([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], None)
```

Figura 1.5 Elaboración Propia

Las ciudades se ubicarán en un mapa de  $100 \times 100 [u^2]$  con coordenadas generadas de manera aleatoria. (ver figura 1.6)

```
Crear coordenadas aleatorias para cada ciudad, en un mapa de 100x100

[4]: random=np.random
      random.seed(1)

      coordenada_x = random.rand(n)*100
      coordenada_y = random.rand(n)*100

      print(f"coordenadas X para todas las ciudades: \n {coordenada_x}")
      print(f"\nCoordenadas Y para todas las ciudades\n {coordenada_y}")

coordenadas X para todas las ciudades:
[4.17022005e+01 7.20324493e+01 1.14374817e-02 3.02332573e+01
 1.46755891e+01 9.23385948e+00 1.86260211e+01 3.45560727e+01
 3.96767474e+01 5.38816734e+01]

Coordenadas Y para todas las ciudades
[41.91945144 68.52195004 20.44522497 87.81174364 2.73875932 67.04675102
 41.73048024 55.86898284 14.03869386 19.81014891]
```

Figura 1.6 Elaboración Propia



Para observar las ciudades orientadas en el mapa de  $100 \times 100 [u^2]$  se realizó la siguiente representación para tener una visión más clara (ver figura 1.7).

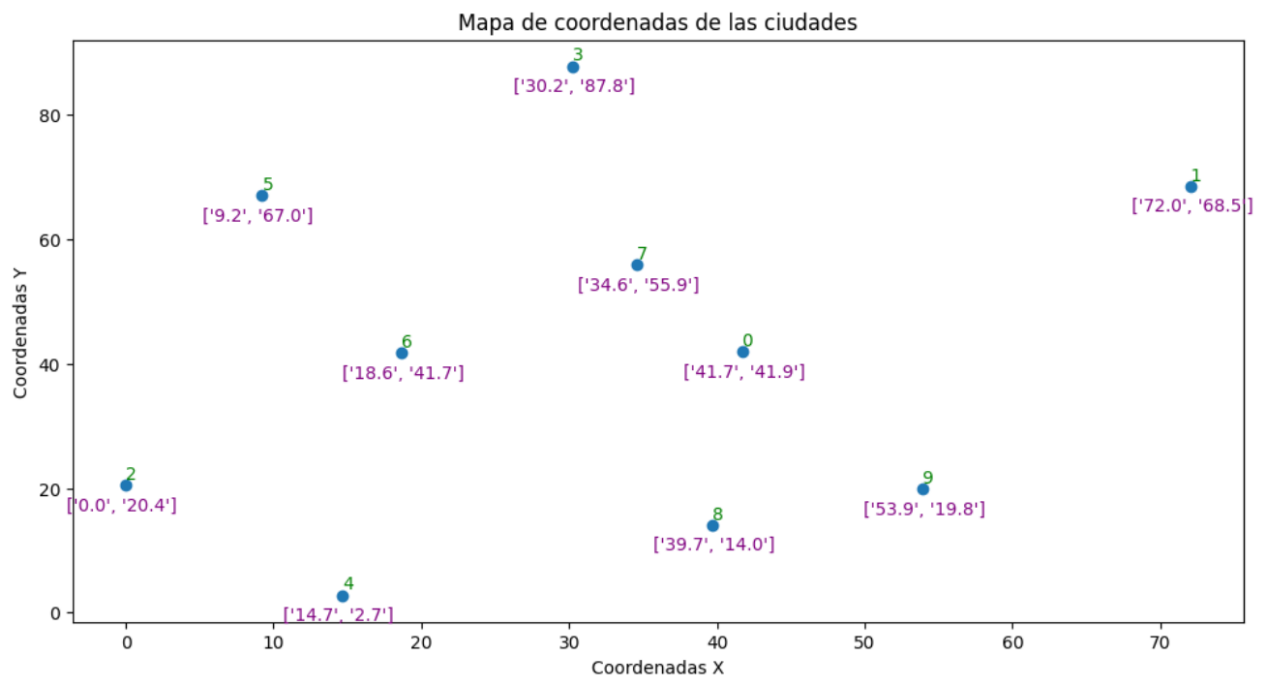


Figura 1.7 Elaboración Propia

Una vez ya visualizado el problema de manera gráfica se necesita determinar el ciclo que recorre todas las ciudades minimizando la distancia total a recorrer. Por ende, se calculará las distancias euclidianas en el espacio bidimensional entre cada ciudad mediante la función ***“numpy.hypot()”*** y será almacenada dentro de un diccionario con las respectivas aristas que hacen referencia a los viajes de una ciudad  $i$  a una ciudad  $j$  (ver figura 1.8).

```
costos = {}
for i,j in aristas:
    costos[(i,j)] = np.hypot(coordenada_x[i] - coordenada_x[j], coordenada_y[i] - coordenada_y[j])
costos
```

Figura 1.8 Elaboración Propia

Teniendo los conjuntos de datos listos, ya es posible implementar el modelo en Python, para su implementación se utilizará la biblioteca de optimización matemática **“Docplex”** desarrollada por IBM para Python.

Mediante la imagen adjunta se muestra la creación del modelo, declarando sus respectivas variables de decisión, función objetivo, y restricciones (ver figura 1.9).

### Creación del modelo

```
6]: #Creamos la instancia del modelo
modelo = Model('Vendedor viajero')

#Creamos las variables de decision
x=modelo.binary_var_dict(aristas, name='x')
d=modelo.continuous_var_dict(ciudades, name='d')

#Funcion objetivo
modelo.minimize(modelo.sum(costos[i]*x[i] for i in aristas))

#Restricciones

#restriccion de salida
for c in ciudades:
    modelo.add_constraint(modelo.sum(x[(i,j)] for i,j in aristas if i==c)==1, cname='Salida_%d' %c)
#La suma de todas las posibles salidas de una ciudad c debe ser 1, se debe salir por un camino de una ciudad.

#restriccion de entrada
for c in ciudades:
    modelo.add_constraint(modelo.sum(x[(i,j)] for i,j in aristas if j==c)==1, cname='Entrada_%d' %c)
#La suma de todas las posibles entradas de una ciudad c deben ser 1, se debe entrar por un camino a la ciudad.

# Eliminacion de subciclos (subtours)
for i,j in aristas:
    if j!=0:
        modelo.add_indicator(x[(i,j)], d[i]+1==d[j], name='order_(%d,%d)'%(i,j))
```

Figura 1.9 Elaboración propia

Ya creado el modelo se puede obtener la solución (ver imagen 2.0 y 2.1).

## Resolver el modelo

```
solucion = modelo.solve(log_output=True)
```

```
Version identifier: 22.1.1.0 | 2022-11-26 | 9160aff4d
CPXPARAM_Read_DataCheck          1
Tried aggregator 2 times.
MIP Presolve modified 36 coefficients.
Aggregator did 36 substitutions.
Reduced MIP has 65 rows, 145 columns, and 315 nonzeros.
Reduced MIP has 90 binaries, 0 generals, 0 SOSs, and 81 indicators.
Presolve time = 0.00 sec. (0.25 ticks)
Probing time = 0.00 sec. (0.14 ticks)
Tried aggregator 1 time.
Detecting symmetries...
Reduced MIP has 65 rows, 145 columns, and 315 nonzeros.
Reduced MIP has 90 binaries, 0 generals, 0 SOSs, and 81 indicators.
Presolve time = 0.02 sec. (0.21 ticks)
Probing time = 0.00 sec. (0.14 ticks)
Clique table members: 56.
MIP emphasis: balance optimality and feasibility.
MIP search method: dynamic search.
Parallel mode: deterministic, using up to 8 threads.
Root relaxation solution time = 0.00 sec. (0.11 ticks)
```

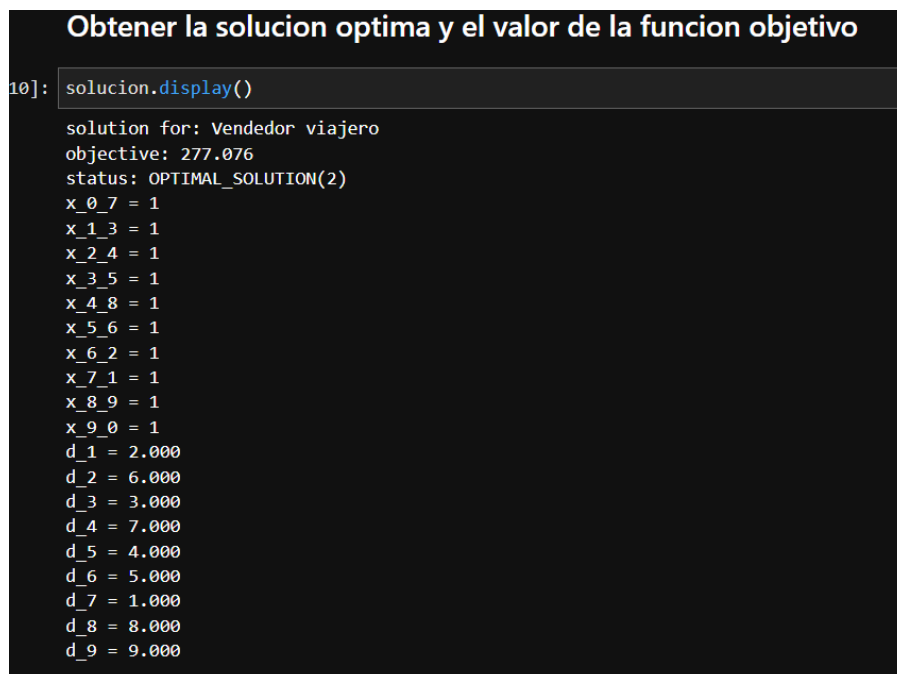
Figura 2.0 Elaboración Propia

Nodes		Objective	IInf	Best Integer	Cuts/	ItCnt	Gap
Node	Left				Best Bound		
	0	0	254.0685	9	254.0685	6	
*	0+	0		277.0762	254.0685		8.30%
	0	0	cutoff	277.0762		17	0.00%
Elapsed time = 0.05 sec. (1.72 ticks, tree = 0.01 MB, solutions = 1)							
Clique cuts applied: 4							
Implied bound cuts applied: 2							
Root node processing (before b&c):							
Real time		= 0.05 sec. (1.73 ticks)					
Parallel b&c, 8 threads:							
Real time		= 0.00 sec. (0.00 ticks)					
Sync time (average)		= 0.00 sec.					
Wait time (average)		= 0.00 sec.					
-----							
Total (root+branch&cut)		= 0.05 sec. (1.73 ticks)					

Figura 2.1 Elaboración Propia

A partir de las figuras adjuntas se puede observar el tiempo que demoró el modelo en encontrar la solución óptima que optimiza la función objetivo y cumple con todas las restricciones, el cual se demoró un total de 0.05[seg] utilizando el método de resolución “ramificación y Poda” (“**Branch&cut**”).

Mediante este modelo se puede obtener la solución óptima y el valor de la función objetivo con el siguiente comando. (ver figura 2.2)



```
Obtener la solución óptima y el valor de la función objetivo

10]: solucion.display()

solution for: Vendedor viajero
objective: 277.076
status: OPTIMAL_SOLUTION(2)
x_0_7 = 1
x_1_3 = 1
x_2_4 = 1
x_3_5 = 1
x_4_8 = 1
x_5_6 = 1
x_6_2 = 1
x_7_1 = 1
x_8_9 = 1
x_9_0 = 1
d_1 = 2.000
d_2 = 6.000
d_3 = 3.000
d_4 = 7.000
d_5 = 4.000
d_6 = 5.000
d_7 = 1.000
d_8 = 8.000
d_9 = 9.000
```

Figura 2.2 Elaboración propia

Se puede graficar la solución obtenida, pero primero se deben obtener las aristas solución y posteriormente se pueden graficar. (ver figura 2.3 y 2.4)

## Graficar la solución

```
11]: aristas_activas = [i for i in aristas if x[i].solution_value>0.9]
aristas_activas
```

```
11]: [(0, 7),
      (1, 3),
      (2, 4),
      (3, 5),
      (4, 8),
      (5, 6),
      (6, 2),
      (7, 1),
      (8, 9),
      (9, 0)]
```

Figura 2.3 Elaboración propia

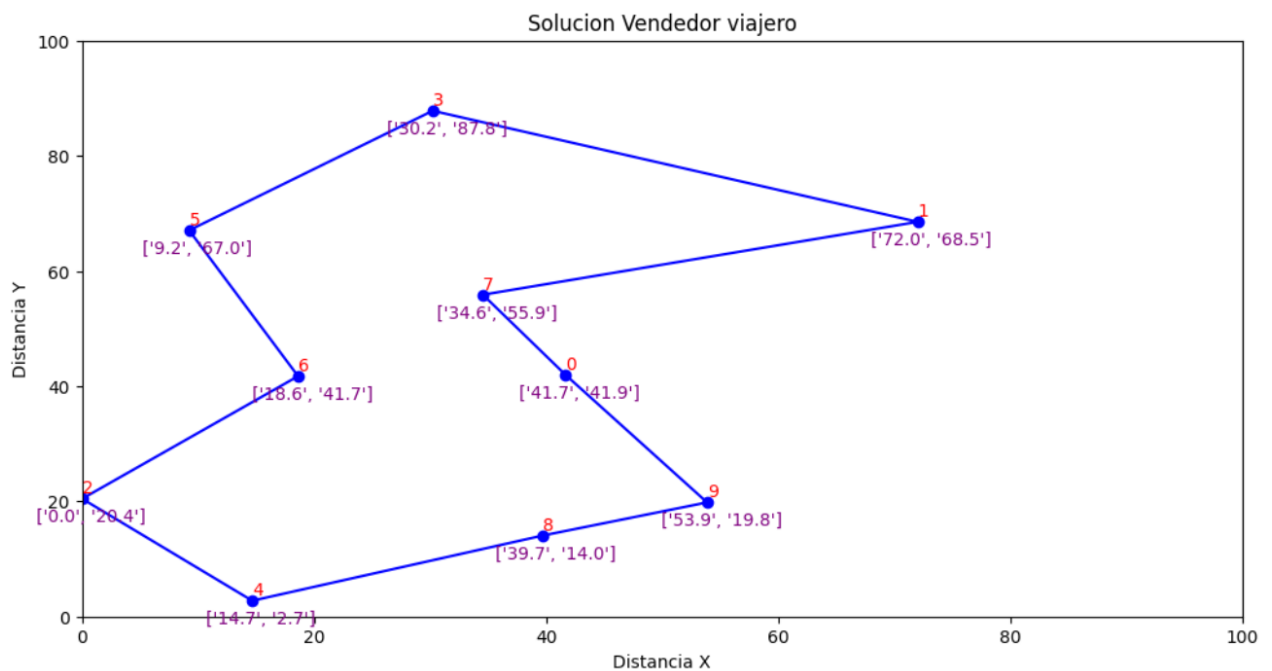


Figura 2.4 Elaboración propia

## EXPERIMENTOS COMPUTACIONALES

Los experimentos computacionales se llevarán a cabo con el objetivo de evaluar el desempeño del algoritmo de Held-Karp, un algoritmo de programación dinámica utilizado para resolver el problema del agente viajero. Se obtendrán los tiempos de ejecución para diferentes grafos que contienen 5, 10, 15 y 20 nodos. Estos tiempos de ejecución serán graficados junto con la cota asintótica superior e inferior del algoritmo. El objetivo de estos experimentos es poder apreciar cómo crece el tiempo de ejecución del algoritmo a medida que aumenta la cantidad de nodos en un grafo. La realización de estos experimentos permitirá evaluar la eficiencia del algoritmo de Held-Karp y su capacidad para resolver el problema del agente viajero en grafos de diferentes tamaños. La evaluación de la complejidad del algoritmo permitirá comprender mejor su comportamiento y su aplicabilidad en diferentes situaciones prácticas.

- Implementación del algoritmo de “Held-Karp”:

```
import random
import time
import matplotlib.pyplot as plt
import numpy as np

def generate_graph(num_nodes):
    graph = [[0]*num_nodes for _ in range(num_nodes)]
    for i in range(num_nodes):
        for j in range(i+1, num_nodes):
            graph[i][j] = random.randint(1, 100)
            graph[j][i] = graph[i][j]
    return graph

def tsp_dp(graph):
    n = len(graph)
    dp = [[float('inf')] * (1 << n) for _ in range(n)]
    dp[0][1] = 0

    for mask in range(1 << n):
        for last in range(n):
            if not (mask & (1 << last)):
                continue
            for curr in range(n):
                if mask & (1 << curr):
                    continue
                nxt = mask | (1 << curr)
                dp[curr][nxt] = min(dp[curr][nxt], dp[last][mask] + graph[last][curr])

    return min(dp[i][(1<<n)-1] + graph[i][0] for i in range(n))
```

Figura 2.5 Elaboración Propia

Este código es una implementación del “*TSP*” utilizando el algoritmo de programación dinámica de “*Held-Karp*”. El objetivo es encontrar el camino más corto que visita cada ciudad una sola vez y vuelve al punto de partida en un grafo completo. La función “*generate\_graph*” genera un grafo aleatorio con un número dado de nodos. La función “*tsp\_dp*” utiliza el algoritmo de Held-Karp para encontrar el camino más corto en el grafo dado. Los tiempos de ejecución del algoritmo se miden para graficarlos junto a las cotas superior e inferior de la complejidad del algoritmo.

- Ejecución del algoritmo:

```
# Generar grafos aleatorios y medir el tiempo que tarda el algoritmo en encontrar la solución óptima
num_graphs = 10
graph_sizes = [5, 10, 15, 20]
times = []
for n in graph_sizes:
    print(f"Generando {num_graphs} grafos aleatorios de tamaño {n}...")
    avg_time = 0
    for i in range(num_graphs):
        graph = generate_graph(n)
        start_time = time.time()
        tsp_dp(graph)
        elapsed_time = time.time() - start_time
        avg_time += elapsed_time
    times.append(avg_time/num_graphs)

# Graficar los tiempos en función del tamaño del grafo
plt.plot(graph_sizes, times)
plt.title("Tiempo de ejecución para TSP con programación dinámica")
plt.xlabel("Tamaño del grafo")
plt.ylabel("Tiempo (segundos)")
plt.show()

Generando 10 grafos aleatorios de tamaño 5...
Generando 10 grafos aleatorios de tamaño 10...
Generando 10 grafos aleatorios de tamaño 15...
Generando 10 grafos aleatorios de tamaño 20...
```

Figura 2.6 Elaboración Propia

La imagen de la figura 2.6 muestra la ejecución del algoritmo de ***Held-Karp*** para medir el tiempo de ejecución del algoritmo y resolver el problema del ***TSP*** para diferentes tamaños de grafos ( $\forall n \in \{5,10,15,20\}$ ). Genera gráficos para comparar los tiempos de ejecución obtenidos con las cotas asintóticas del algoritmo. Se generan varios grafos aleatorios para cada tamaño de grafo y se toma el tiempo promedio que tarda el algoritmo en resolver el problema para cada uno de ellos. Los tiempos promedio se guardan en una lista para su posterior análisis. Este proceso se repite para diferentes tamaños de grafos para poder observar cómo cambia el tiempo de ejecución del algoritmo en función del tamaño del grafo.



- Gráficos de complejidad

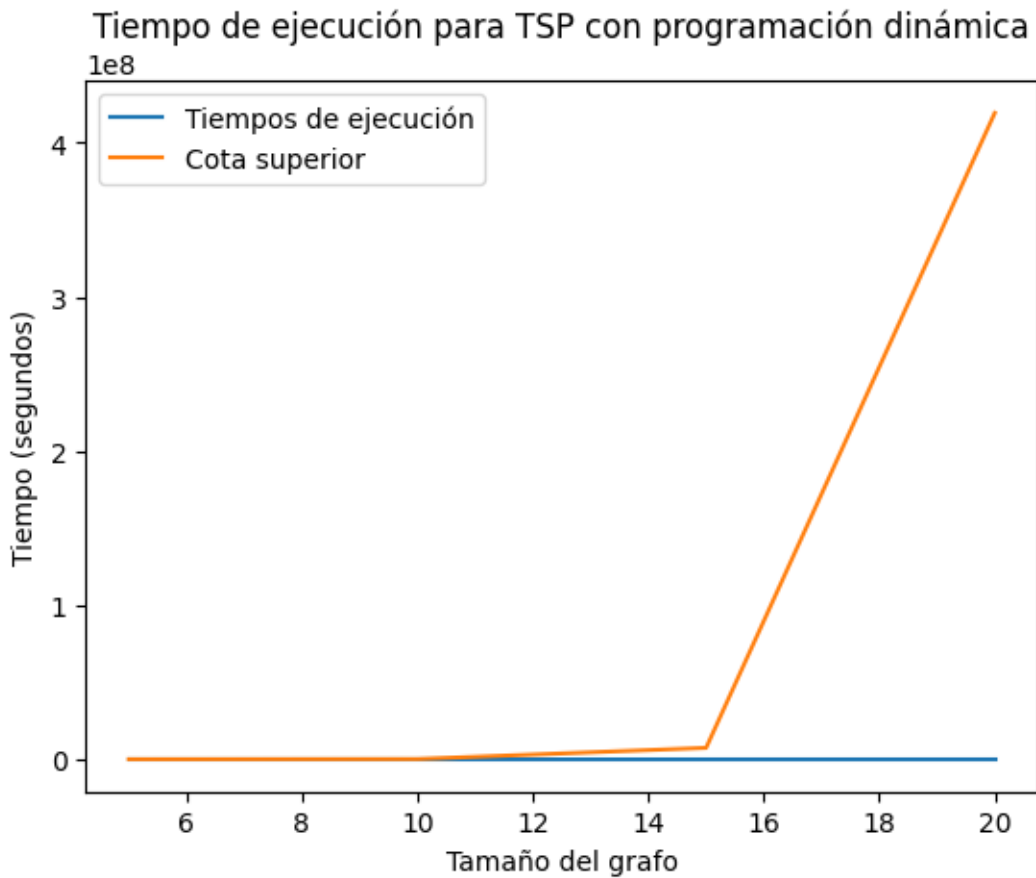


Figura 2.7 Elaboración Propia

El gráfico de la figura 2.7 muestra el tiempo de ejecución del algoritmo de **“Held-Karp”** para diferentes tamaños de grafo, junto con una cota superior teórica en función del tamaño del grafo. La línea azul representa el tiempo de ejecución real del algoritmo, mientras que la línea naranja representa la cota superior teórica, que se calculó como una función exponencial de la forma  $O(n^2 * 2^n)$ . Se puede ver que, a medida que aumenta el tamaño del grafo, el tiempo de ejecución aumenta significativamente y se acerca cada vez más a la cota superior teórica. Esto indica que el algoritmo de **“Held-Karp”** tiene una complejidad exponencial y no es eficiente para grafos grandes.

## CONCLUSIONES

En este estudio se exploró el mundo del Problema del Vendedor Viajero (TSP) y los Ciclos Hamiltonianos, desde sus fundamentos matemáticos hasta sus aplicaciones modernas, abordando los desafíos en la búsqueda de soluciones eficientes para estos problemas complejos.

Se presentó una introducción donde se explicó la importancia del TSP y los ciclos Hamiltonianos, su historia y las diversas aplicaciones prácticas en diferentes campos. Se enfatizó en la complejidad computacional del problema y cómo aumenta el tiempo de resolución a medida que se incrementa el número de nodos.

Se definió el problema y se presentó un modelo matemático, incluyendo las variables de decisión, la función objetivo y las restricciones que permiten resolver el problema de manera óptima. Además, se incluyeron los parámetros necesarios para definir el problema y sus soluciones.

Se explicó la implementación del algoritmo de Held-Karp en Python, que es una técnica de programación dinámica para resolver el TSP. Se presentaron gráficos que muestran el desempeño del algoritmo para diferentes tamaños de grafos, y se concluyó que la complejidad del algoritmo crece exponencialmente con el número de nodos, lo que lo hace impracticable para grafos grandes.

Para finalizar este estudio se logra entender que la lógica de los modelos de optimización consiste en buscar la mejor solución posible para un problema dado, considerando ciertas restricciones y objetivos. Para lograr esto, utilizan técnicas matemáticas que permiten identificar la solución óptima o la que se acerca más a ella.

En contraste, los algoritmos de búsqueda, como el de Held-Karp, pueden ser más eficientes para resolver problemas específicos, como el problema del vendedor viajero. Sin embargo, estos algoritmos no siempre garantizan encontrar la solución óptima en todos los casos.

La ventaja de los modelos de optimización es que pueden manejar problemas más complejos con múltiples restricciones y objetivos, y en algunos casos pueden ser más rápidos que los algoritmos de búsqueda. Esto se debe a que los modelos de optimización aprovechan la estructura del problema y las técnicas matemáticas para simplificarlo y reducir la cantidad de soluciones que deben ser evaluadas, lo que los hace más eficientes en términos de tiempo y recursos.

## REFERENCIAS BIBLIOGRÁFICAS

- Lin, S. and Kernighan, B. W. (1973). An Effective Heuristic Algorithm for the Traveling-Salesman Problem. *Operations Research*, 21(2), 498-516. doi: 10.1287/opre.21.2.498
- Steen, M. van. (2010). *Graph Theory and Complex Networks: An Introduction*. Courier Corporation.
- Chartrand, G. and Zhang, P. (2012). *A First Course in Graph Theory*. Dover Publications.
- Youtube. (n.d.). Sergio Correa - Ingeniero Civil Industrial. Retrieved from <https://www.youtube.com/@sergiocorrea1179>
- ChatGPT. (2021). OpenAI's GPT-3.5 language model. Retrieved from <https://openai.com/>