MA2015: Diseño de Algoritmos Matemáticos Bioinspirados
TSP: Algoritmos Genéticos y de Hormigas

Nombre: Emiliano Ruiz López Matricula: A01659693
Nombre: Alberto Boughton Reyes Matricula: A01178500
Nombre: Valeria García Hernández Matricula: A01742811

Resumen

El presente trabajo aborda el Problema del agente viajero, uno de los problemas clásicos de optimización combinatoria, cuya finalidad es de terminar la ruta más corta que permita visitar un conjunto de nodos o ciudades exactamente una vez y regresar al punto de partida, minimizando la distancia total recorrida.

Para su resolución se generaron redes aleatorias de 40, 100, 150, y 200 nodos, sobre las cuales se aplicaron y compararon tres enfoques distintos:

- Método exacto implementado en GAMS, que garantiza la obtención de la solución óptima, aunque con un alto costo computacional.
- Algoritmo Genético, método bioinspirado basado en los principios de la evolución natural que busca soluciones de alta calidad mediante procesos de selección, cruza, mutación y supervivencia.
- Algoritmo de colonia de hormigas, inspirado en el comportamiento cooperativo de las hormigas en la búsqueda de recursos, que utiliza feromonas para guiar la exploración hacia rutas prometedoras.

Se compararon los resultados obtenidos por cada método en términos de valor objetivo y tiempo de cómputo, con el fin de destacar el equilibrio entre precisión y eficiencia de los enfoques bioinspirados frente al método exacto. Los resultados muestran que aunque los métodos heurísticos no siempre alcanzan una solución óptima, estos ofrecen tiempos de ejecución significativamente menores, lo que los hace más adecuados para instancias de gran tamaño.

Introducción

El Problema del Viajante (Traveling Salesperson Problem - TSP) es un problema clásico y fundamental dentro de la optimización combinatoria y la ciencia de la computación.

El **objetivo principal** del TSP es encontrar el camino más corto posible que visite un conjunto específico de ciudades (nodos) y regrese al punto de partida original, visitando cada ciudad exactamente una vez.

El TSP pertenece a la clase de problemas NP-completo. Esto significa que, si bien es fácil verificar una solución dada, encontrar la solución óptima (la ruta de menor distancia) se vuelve intratable computacionalmente para un gran número de ciudades, ya que la cantidad de posibles rutas crece de manera factorial (O(N!)). .Debido a su alta complejidad, el TSP se utiliza a menudo como un banco de pruebas para evaluar la eficiencia y el rendimiento de algoritmos de optimización, incluyendo los

métodos de solución exacta (como programación matemática con GAMS y los métodos heurísticos y bioinspirados como el Algoritmo Genético y el Algoritmo de Colonia de Hormigas .

El enfoque de esta situación problema es diseñar, implementar y comparar soluciones para el TSP en redes de diferente tamaño (40, 100, 150 y 200 nodos), evaluando la calidad de la solución y el tiempo de cómputo para cada método.

Modelación matemática

Conjunto de índices

- $I = \{1, \cdots, n\}$: Nodos/ciudades; 1 es la ciudad de origen

Parámetros

- $c_{ij} \geq 0$: Distancia o costo para ir del nodo $i \in I$ a $j \in I$

Variables de decisión

- $x_{ij} \in 0, 1$ Vale 1 si viaja de $i \in I$ a $j \in I$ y 0 en caso contrario

Función objetivo

- $Minimizar\ Z = \sum_{i=1}^{n} \sum_{j=1}^{n} = c_{ij} x_{ij}$

Sujeto a

Cada ciudad se visita exactamente una vez:

$$\sum_{i} x_{ij} = 1 \quad \forall i$$

$$\sum_{j} x_{ij} = 1 \quad \forall j$$

Se evitan subrutas (subtours) mediante restricciones adicionales (como las de Miller-Tucker-Zemlin o restricciones de eliminación de subciclos).

Descripción del problema

Se busca minimizar la distancia total recorrida entre las diferentes direcciones proporcionadas, considerando distintas redes generadas aleatoriamente. Cada red representa un conjunto de nodos (ciudades o direcciones) conectados (ciudades o direcciones) conectados entre sí mediante aristas que indican la distancia entre cada par de puntos.

El objetivo del trabajo es comparar el desempeño de distintos metodos de solucion aplicados al TSP:

- Método exacto, resuelto a optimalidad,usamos el software de modelado algebraico GAMS para llevar a cabo la implementación.

- Algoritmo genético y Algoritmo de colonia de hormigas.

Para ello, deberán generarse redes de 40, 100, 150 y 200 nodos, aplicando los tres métodos de resolución.

Complejidad del problema

El Problema del Agente Viajero (TSP) pertenece a la clase de problemas NP-duros (NP-hard) dentro de la teoría de la complejidad computacional.

Esto significa que no existe un algoritmo conocido que pueda resolverlo en tiempo polinomial para todos los casos, y que su tiempo de ejecución crece de forma exponencial conforme aumenta el número de ciudades o nodos

En el TSP, el número de rutas posibles crece factorialmente con el número de nodos:

$Número\ de\ posibles\ rutas = \frac{(n-1)!}{2}$ para el caso simétrico, o simplemente (n−1)! si es asimétrico.

Esto implica que incluso con un número moderado de ciudades, el número de combinaciones posibles se vuelve enorme.

Métodos de Solución

Los algoritmos implementados buscan encontrar una ruta (permutación de ciudades) que minimice la distancia total de viaje, utilizando la Distancia Haversine como métrica de costo.

1. Algoritmo Genético (AG)

El Algoritmo Genético es una heurística de búsqueda inspirada en los principios de la selección natural y la evolución. Mantiene una población de soluciones y las mejora iterativamente mediante operadores genéticos.

Descripción del Algoritmo

Inicialización (inicializar_poblacion): Se genera una población inicial de rutas aleatorias (permutaciones de los índices de las ciudades).

```python
def inicializar_poblacion(tamano_poblacion, indices_ciudades):
    poblacion = []
    for _ in range(tamano_poblacion):
        individuo = indices_ciudades.copy()
        random.shuffle(individuo)
        poblacion.append(individuo)
    return poblacion
```

Cálculo de Aptitud (calcular_aptitud): La aptitud de una ruta es inversamente proporcional a su distancia total. El algoritmo busca minimizar esta distancia, por lo que las rutas más cortas tienen mayor aptitud.

```python
def calcular_aptitud(individuo, matriz_distancias):
    distancia_total = 0
    for i in range(len(individuo)):
        ciudad_origen = individuo[i]
        ciudad_destino = individuo[(i + 1) % len(individuo)]
        distancia_total += matriz_distancias[ciudad_origen][ciudad_destino]
    return distancia_total
```

Selección (seleccionar_padres): Se eligen los individuos más aptos para la reproducción. La implementación utiliza la selección elitista, donde la mitad de la población con las menores distancias son seleccionados como padres.

```python
def seleccionar_padres(poblacion, aptitudes, num_padres):
    padres_indices = np.argsort(aptitudes)[:num_padres]
    padres = [poblacion[i] for i in padres_indices]
    return padres
```

Cruce (cruzar_padres): Se combinan las rutas de dos padres para crear descendencia. El método se basa en el Cruce Ordenado (Ordered Crossover - OX), donde se copia un segmento del primer padre y el resto de las ciudades se completa con el orden del segundo padre.

```python
def cruzar_padres(padre1, padre2):
    tamano = len(padre1)
    inicio, fin = sorted(random.sample(range(tamano), 2))
    hijo = [None]*tamano
    hijo[inicio:fin+1] = padre1[inicio:fin+1]

    pointer = 0
    for i in range(tamano):
        if hijo[i] is None:
            while padre2[pointer] in hijo:
                pointer += 1
            hijo[i] = padre2[pointer]
            pointer += 1
    return hijo
```

Mutación (mutar_individuo): Se aplica una mutación de intercambio (swap mutation) con una pequeña probabilidad, intercambiando aleatoriamente la posición de dos ciudades para introducir diversidad y explorar nuevos espacios de solución.

```python
def mutar_individuo(individuo, tasa_mutacion):
    for i in range(len(individuo)):
        if random.random() < tasa_mutacion:
            j = random.randint(0, len(individuo)-1)
            individuo[i], individuo[j] = individuo[j], individuo[i]
    return individuo
```

Nueva Generación: La descendencia mutada reemplaza a la población anterior, y el proceso se repite por un número definido de generaciones, rastreando siempre la mejor ruta global.

Tecnológico de Monterrey

Pseudocódigo Algoritmo genetico

Algoritmo 1 Algoritmo genético para TSP (OX + mutación por intercambio)

Entrada: matriz de distancias D; tamaño de población P; número de generaciones G; tasa de mutación μ
Salida: mejor ruta *MejorRuta* y su distancia *MejorDistancia*
Inicialización: $N \leftarrow |D|$; $IndicesCiudades \leftarrow [0, \ldots, N-1]$
$Poblacion \leftarrow$ INICIALIZARPOBLACION(P, $IndicesCiudades$)
$MejorDistancia \leftarrow +\infty$; $MejorRuta \leftarrow$
Para $Generacion = 1$ **hasta** G **hacer**
  $Aptitudes \leftarrow [\,]$
  Para cada $Individuo \in Poblacion$ **hacer**
    $d \leftarrow$ CALCULARAPTITUD($Individuo$, D) {distancia total de la ruta}
    APPEND($Aptitudes$, d)
    Si $d < MejorDistancia$ **entonces**
      $MejorDistancia \leftarrow d$
      $MejorRuta \leftarrow Individuo$
    Fin Si
  Fin Para
  $k \leftarrow \lfloor P/2 \rfloor$ {elitismo: 50 % mejores (menor distancia)}
  $Padres \leftarrow$ SELECCIONARPADRES($Poblacion$, $Aptitudes$, k)
  $SiguienteGeneracion \leftarrow [\,]$
  Mientras $|SiguienteGeneracion| < P$ **hacer**
    $Padre_1, Padre_2 \leftarrow$ ELEGIRDOSALEATORIOS($Padres$)
    $Hijo \leftarrow$ CRUZARPADRESOX($Padre_1, Padre_2$) {crossover por orden (OX)}
    $Hijo \leftarrow$ MUTARSWAP($Hijo$, μ) {intercambia dos ciudades con prob. μ}
    APPEND($SiguienteGeneracion$, $Hijo$)
  Fin Mientras
  $Poblacion \leftarrow SiguienteGeneracion$
Fin Para
Retornar: ($MejorRuta$, $MejorDistancia$)

Parámetros utilizados (AG)

Parámetro	Función en Python	Valor Utilizado	Descripción
Tamaño de Población	tamano_poblacion	100	Número de rutas (individuos) en cada generación.

Generaciones	num_generaciones	500	Número total de iteraciones del proceso evolutivo.
Tasa de Mutación	tasa_mutacion	0.01	Probabilidad de que se aplique el operador de mutación a un individuo.

2. Algoritmo por Colonia de Hormigas

El Algoritmo por Colonia de Hormigas (ACO) es una metaheurística que simula el comportamiento de las **hormigas reales** para encontrar el camino más corto, utilizando un mecanismo de comunicación basado en **feromonas**.

Descripción del Algoritmo

Inicialización: Se establece una cantidad inicial de feromona en todos los caminos (aristas). La heurística (η) se calcula como el inverso de la distancia entre ciudades ($\frac{1}{d_{ij}}$), dando preferencia a los caminos más cortos.

Construcción del Camino (construir_camino): En cada iteración, un conjunto de hormigas construye una ruta completa de manera probabilística.

```python
def construir_camino(feromonas, heuristica, costos, alpha, beta):
    num_nodos = costos.shape[0]
    nodo_actual = random.randint(0, num_nodos - 1)
    camino = [nodo_actual]
    costo_total = 0
    visitados = set(camino)

    while len(camino) < num_nodos:
        probabilidades = calcular_probabilidades(feromonas, heuristica, nodo_actual, visitados, alpha, beta)
        if np.sum(probabilidades) == 0:
            nodos_no_visitados = list(set(range(num_nodos)) - visitados)
            siguiente_nodo = random.choice(nodos_no_visitados)
        else:
            siguiente_nodo = random.choices(range(num_nodos), weights=probabilidades, k=1)[0]
        costo_total += costos[nodo_actual, siguiente_nodo]
        camino.append(siguiente_nodo)
        visitados.add(siguiente_nodo)
        nodo_actual = siguiente_nodo

    costo_total += costos[camino[-1], camino[0]]
    camino.append(camino[0])

    return camino, costo_total
```

La probabilidad de que una hormiga (k) se mueva de la ciudad i a la ciudad j está determinada por la combinación de la feromona (τ) en ese camino y la heurística (η):

$$P_{ij}^{k} = \frac{[\tau_{ij}]^{\alpha}[\eta_{ij}]^{\beta}}{\Sigma[\tau_{ij}]^{\alpha}[\eta_{ij}]^{\beta}}$$

donde:

τ_{ij}: $cantidad\ de\ feromona\ en\ la\ arista\ (i,j)$

$\eta_{ij} = \frac{1}{d_{ij}}$: $heuristica\ inversa\ a\ la\ distancia$

α, β: $parámetros\ que\ ponderan\ la\ importancia\ relativa\ de\ feromona\ y\ heuristica$

Actualización de Feromonas (actualizar_feromonas):

```
def actualizar_feromonas(feromonas, soluciones, rho):
    feromonas *= (1 - rho)
    for camino, costo in soluciones:
        for i in range(len(camino) - 1):
            feromonas[camino[i], camino[i + 1]] += 1 / costo
```

Se realiza en dos fases:

Evaporación: La feromona en todos los caminos disminuye por una tasa ρ para simular el olvido y evitar la convergencia prematura.

Deposición: Se deposita nueva feromona en los caminos recorridos. Las rutas más cortas depositan una mayor cantidad de feromona (inversamente proporcional al costo), reforzando así las soluciones prometedoras.

Después de que todas las hormigas completan su recorrido, las feromonas se actualizan:

$$\tau_{ij} \leftarrow (1 - \rho)\tau_{ij} + \sum_{k=1}^{m} \Delta\tau_{ij}^{k}$$

donde:

ρ: $tasa\ de\ evaporación\ de\ feromonas\ (0 < \rho < 10\ )$

$\Delta\tau_{ij}^{k} = \frac{Q}{L_k}\ si\ la\ hormiga\ k\ recorrió\ la\ arista\ (i,j)$

$siendo\ L_k\ la\ longitud\ total\ de\ la\ ruta\ y\ Q\ una\ constante\ de\ depósito$

Tecnológico de Monterrey

Pseudocódigo Colonia de hormigas

Algoritmo 2 Optimización por Colonia de Hormigas (ACO) para TSP

Entrada: matriz de costos C; número de hormigas m; número de iteraciones T; parámetros α, β, ρ
Salida: mejor camino *MejorCamino* y su costo *MejorCosto*
$N \leftarrow |C|$ {número de ciudades}
MejorCamino $\leftarrow$; *MejorCosto* $\leftarrow +\infty$
$\tau \leftarrow$ matriz $N \times N$ inicializada a 0,1 {feromonas}
$\eta \leftarrow$ CalcularHeuristica(C) {$\eta_{ij} = 1/C_{ij}$ para $i \neq j$}
Para *Iteracion* = 1 **hasta** T **hacer**
  Soluciones $\leftarrow$ []
  Para $k = 1$ **hasta** m **hacer**
    NodoActual $\leftarrow$ EligirNodoInicialAleatorio()
    Camino $\leftarrow$ [*NodoActual*]; *Visitados* $\leftarrow$ {*NodoActual*}; *CostoTotal* $\leftarrow 0$
    Mientras |*Camino*| $< N$ **hacer**
      Prob $\leftarrow$ CalcularProbabilidades(τ, η, *NodoActual*, *Visitados*, α, β)
      Siguiente $\leftarrow$ SeleccionarNodo(*Prob*)
      CostoTotal $\leftarrow$ *CostoTotal* + C[*NodoActual*, *Siguiente*]
      Append(*Camino*, *Siguiente*); Insert(*Visitados*, *Siguiente*)
      NodoActual $\leftarrow$ *Siguiente*
    Fin Mientras
    {Cerrar el ciclo (volver al nodo inicial)}
    $u \leftarrow$ Ultimo(*Camino*)
    CostoTotal $\leftarrow$ *CostoTotal* + $C[u, Camino[1]]$
    Append(*Camino*, *Camino*[1])
    Append(*Soluciones*, (*Camino*, *CostoTotal*))
    Si *CostoTotal* $<$ *MejorCosto* **entonces**
      MejorCosto $\leftarrow$ *CostoTotal*;
      MejorCamino $\leftarrow$ *Camino*
    Fin Si
  Fin Para
  {3) Actualización de feromonas}
  Evaporación: $\tau \leftarrow (1 - \rho) \cdot \tau$
  Depósito:
  Para cada (*Cam*, *Cost*) $\in$ *Soluciones* **hacer**
    $\Delta\tau \leftarrow 1/Cost$
    Para cada arista (i, j) **en** *Cam* **hacer**
      $\tau[i, j] \leftarrow \tau[i, j] + \Delta\tau$
    Fin Para
  Fin Para
Fin Para
Retornar: (*MejorCamino*, *MejorCosto*)

Parámetros utilizados en ACO

Parámetro	Función en Python	Valor Utilizado

Número de Hormigas	num_hormigas	500
Iteraciones	num_iteraciones	100
Factor Alfa (α)	alpha	1.0
Factor Beta (β)	beta	2.0
Tasa de Evaporación	rho	0.5

Resultados:

Nodos (n)	Tiempo Promedio GAMS (s)	Valor Promedio GAMS	Tiempo Promedio AG (s)	Valor Promedio AG	Tiempo Promedio ACO (s)	Valor Promedio ACO
40	33.10	131.73	2.58	156.27	72.17	35.37
100	N/A	N/A	9.74	746.83	379.33	204.80

150	N/A	N/A	16.99	222.26	776.45	263.81
200	N/A	N/A	28.75	1693.18	1245.03	313.93

a) Redes pequeñas (n=40) - Precisión vs Tiempo

Método	Valor de Solución (Distancia)	Tiempo de Cómputo (s)	Desviación del Óptimo
GAMS	131.73	33.10	0% (Óptimo)
ACO	135.37	72.17	2.77%
AG	156.27	2.58	18.63%

GAMS (Método Exacto): Logra el valor óptimo promedio (distancia más corta), pero tiene el mayor tiempo de cómputo promedio (33.10 s) para esta instancia.

En algunos casos, como la instancia “9_40”, el tiempo se dispara a 218.02 segundos.

ACO (Colonia de Hormigas): Obtiene una solución de muy alta calidad, con una desviación promedio de solo 2.77% respecto al óptimo de GAMS, lo que demuestra su buena capacidad de búsqueda local.

Sin embargo, su tiempo de ejecución (72.17 segundos) es el más alto, superando incluso al promedio de GAMS.

AG (Algoritmo Genético): Es, con mucho, el más rápido (2.58 segundos), pero su solución es la de menor calidad de los tres métodos, desviándose en promedio un 18.63% del óptimo de GAMS.

b) Redes Grandes (n => 100) - Eficiencia vs. Calidad Heurística

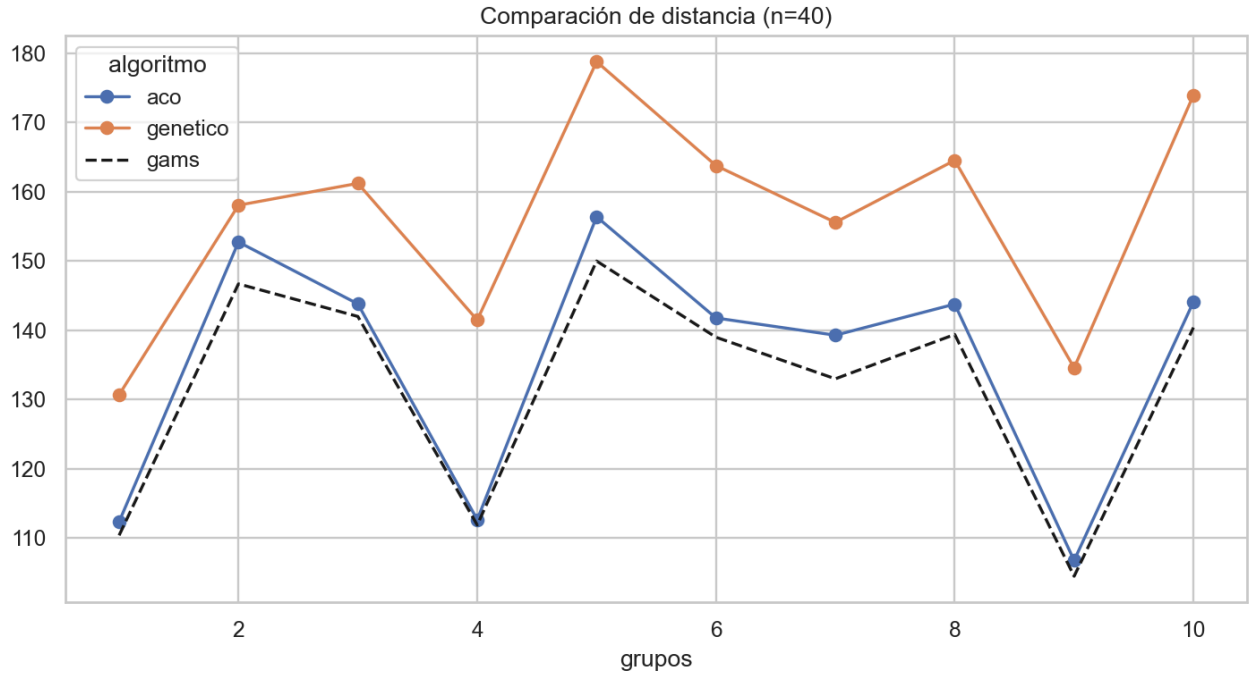
Nodos (n)	Mejor Valor Heurístico	Mejor Tiempo Heurístico
100	**ACO** (204.80)	**AG** (9.74 segundos)
150	**ACO** (263.81)	**AG** (16.99 segundos)
200	**ACO** (313.93)	**AG** (28.75 segundos)

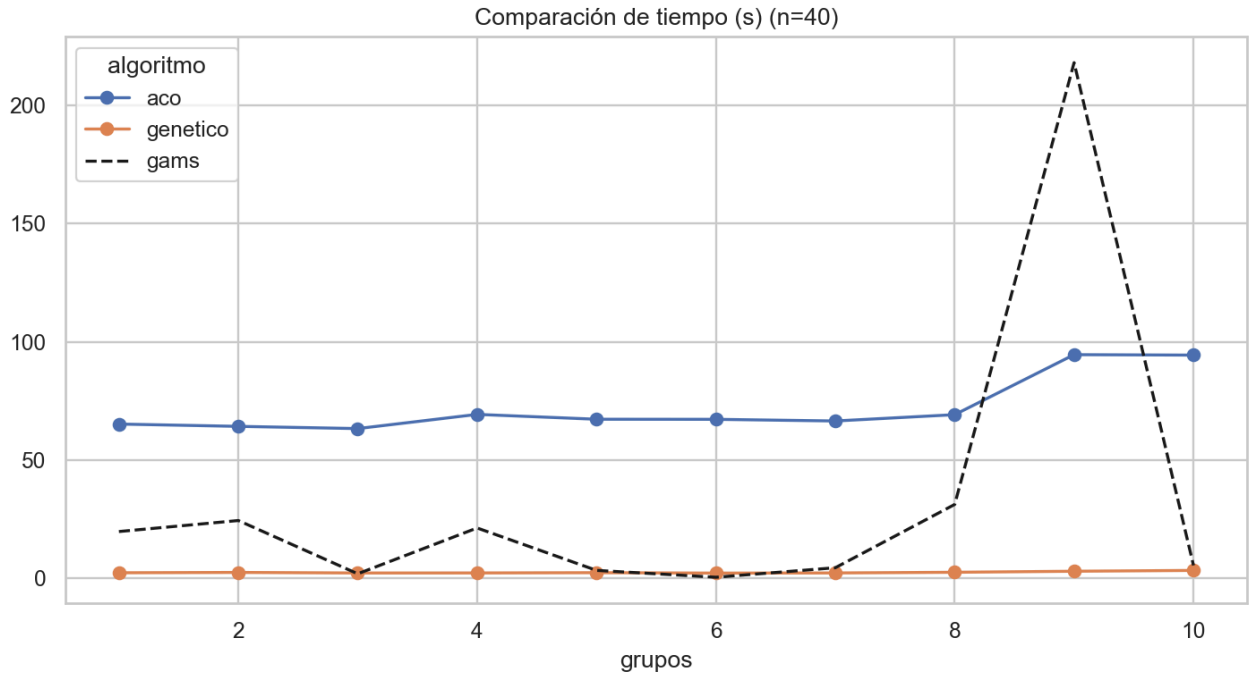
AG (Algoritmo Genético): Mantiene su ventaja como el método más rápido de todos para todas las instancias, con tiempos de cómputo que apenas aumentan linealmente con n (de 2.58 segundos a 28.75 segundos). Esto lo convierte en la opción ideal para escenarios con limitaciones estrictas de tiempo.

ACO (Colonia de Hormigas): Logra consistentemente una calidad de solución mucho mejor (menor distancia) que el AG para instancias de n>100.

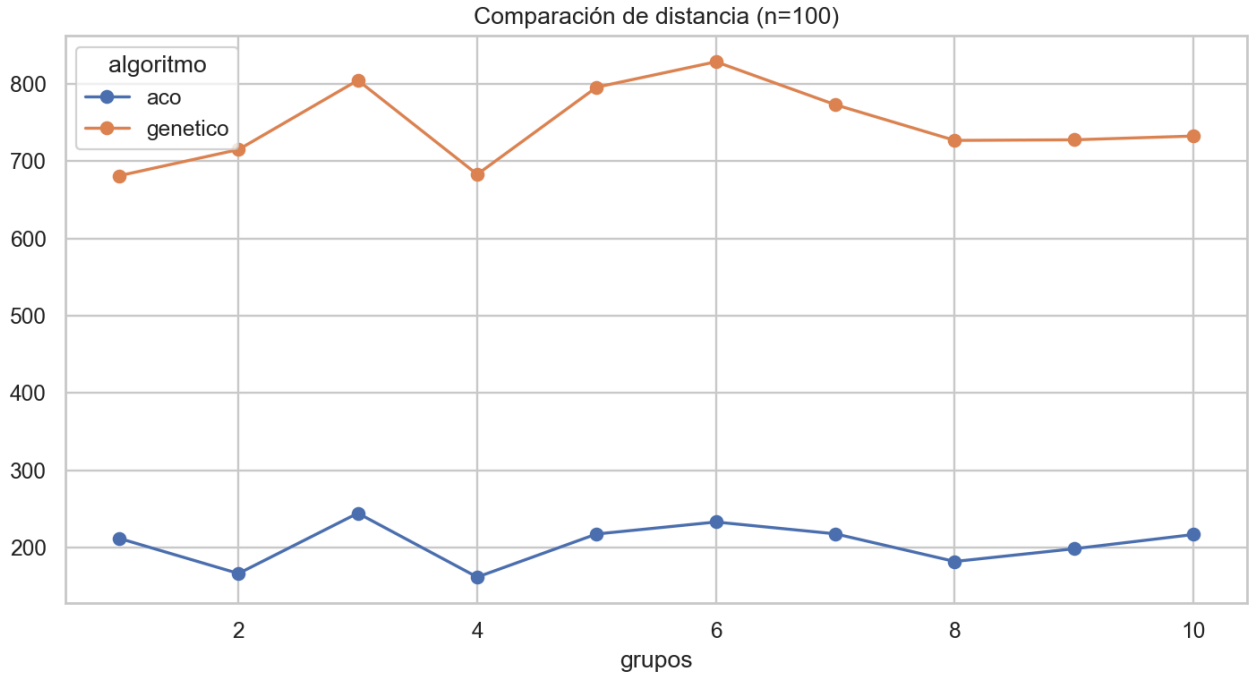
Su tiempo de ejecución crece considerablemente más que el AG (llegando a 1245.03 segundos para n=200), pero el valor de la solución es significativamente menor.

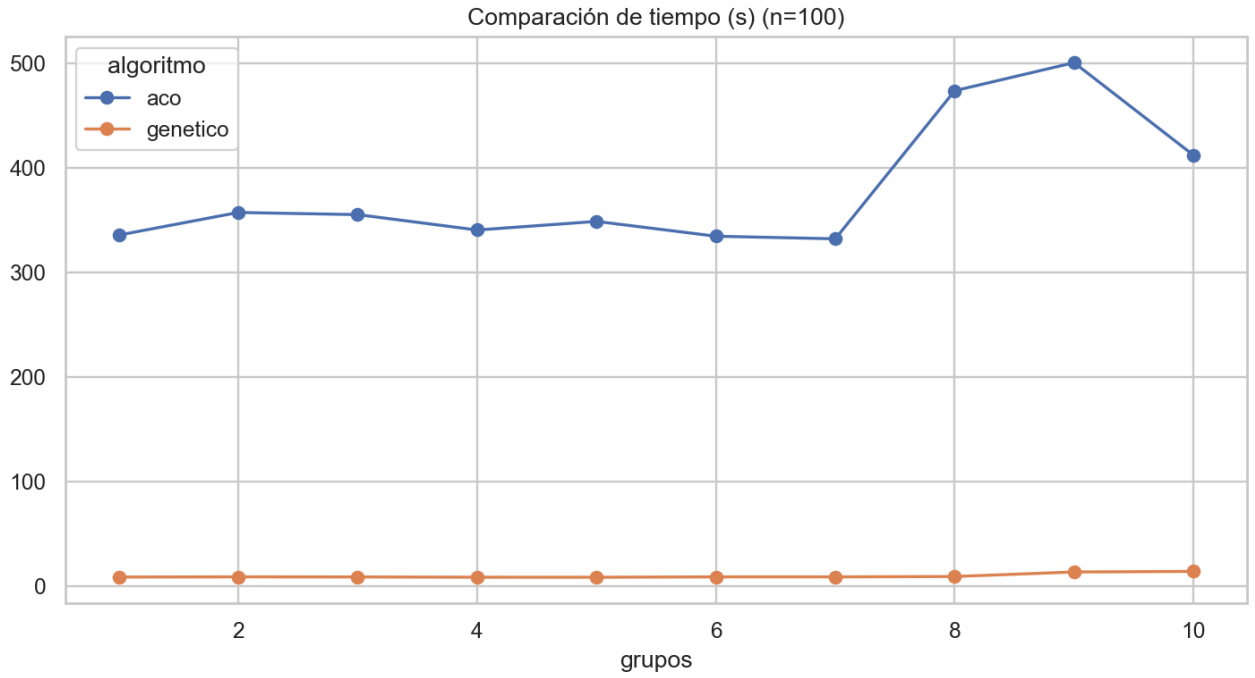
d) Comparativa de métodos (Soluciones, tiempos y mediciones de complejidad experimental)



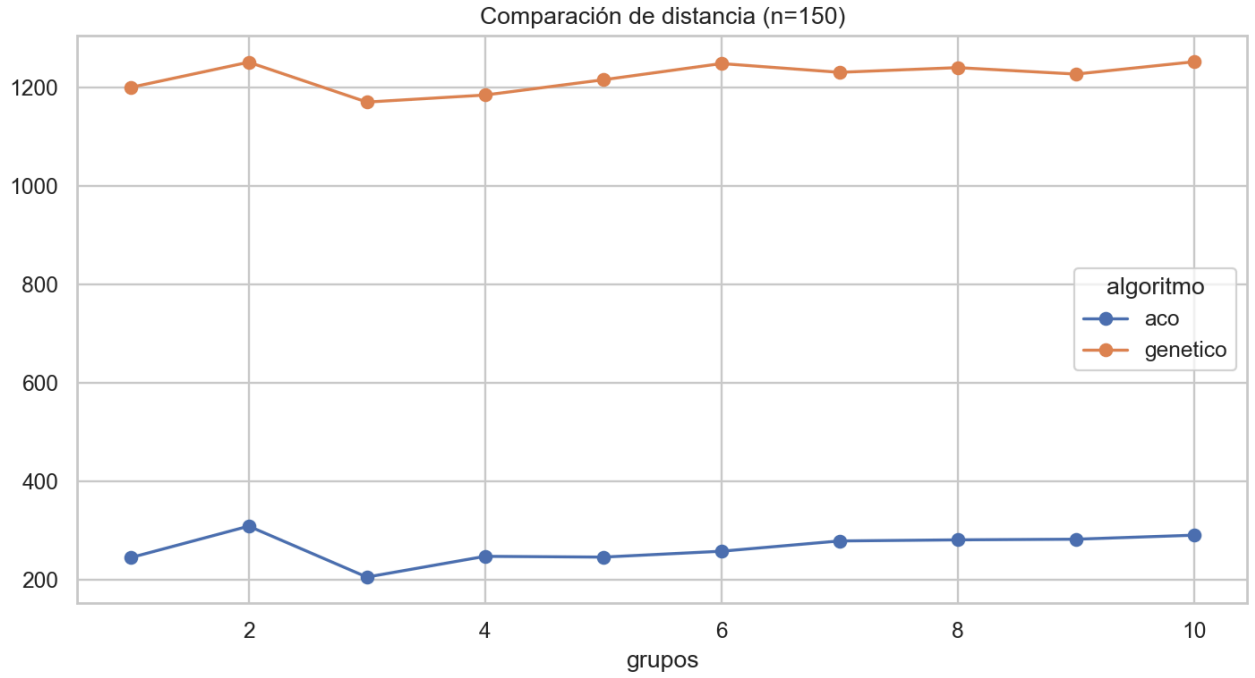


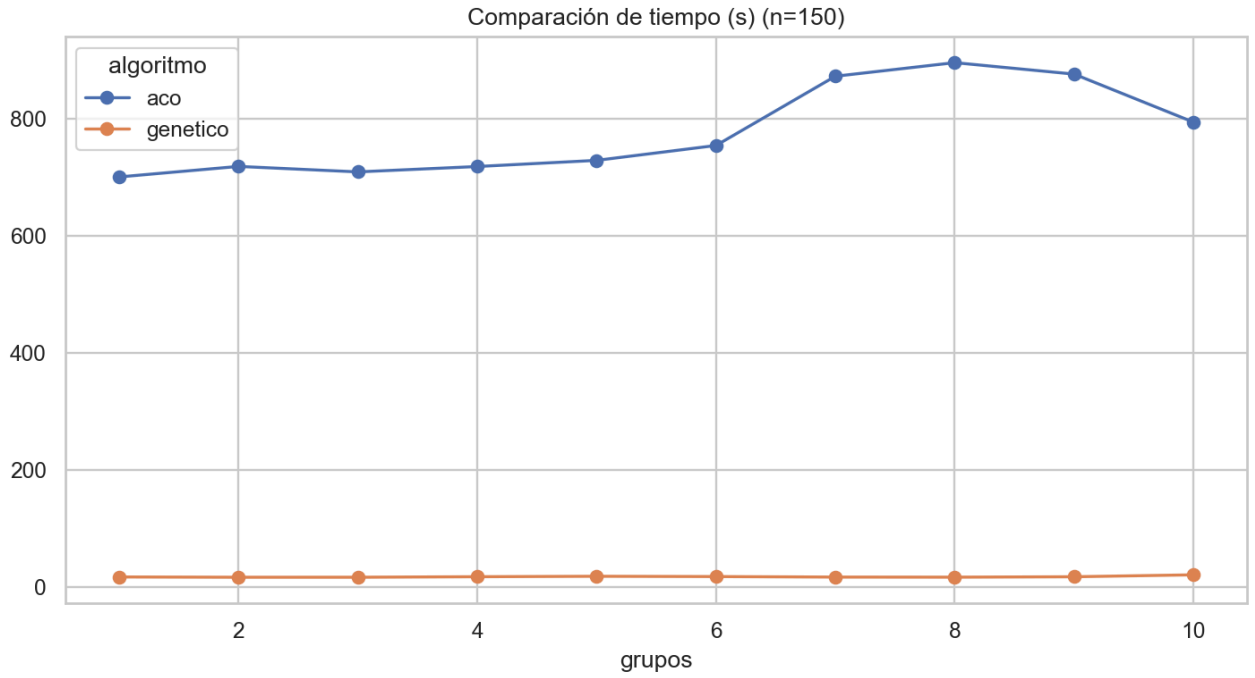
Tecnológico de Monterrey

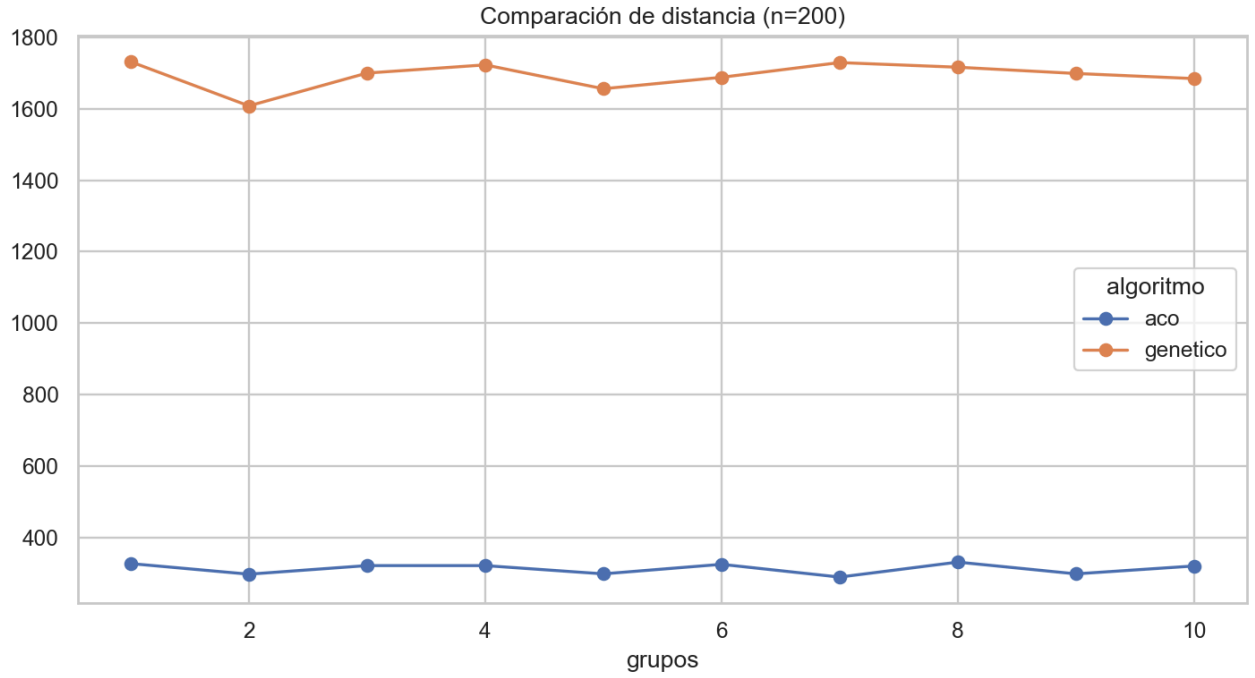


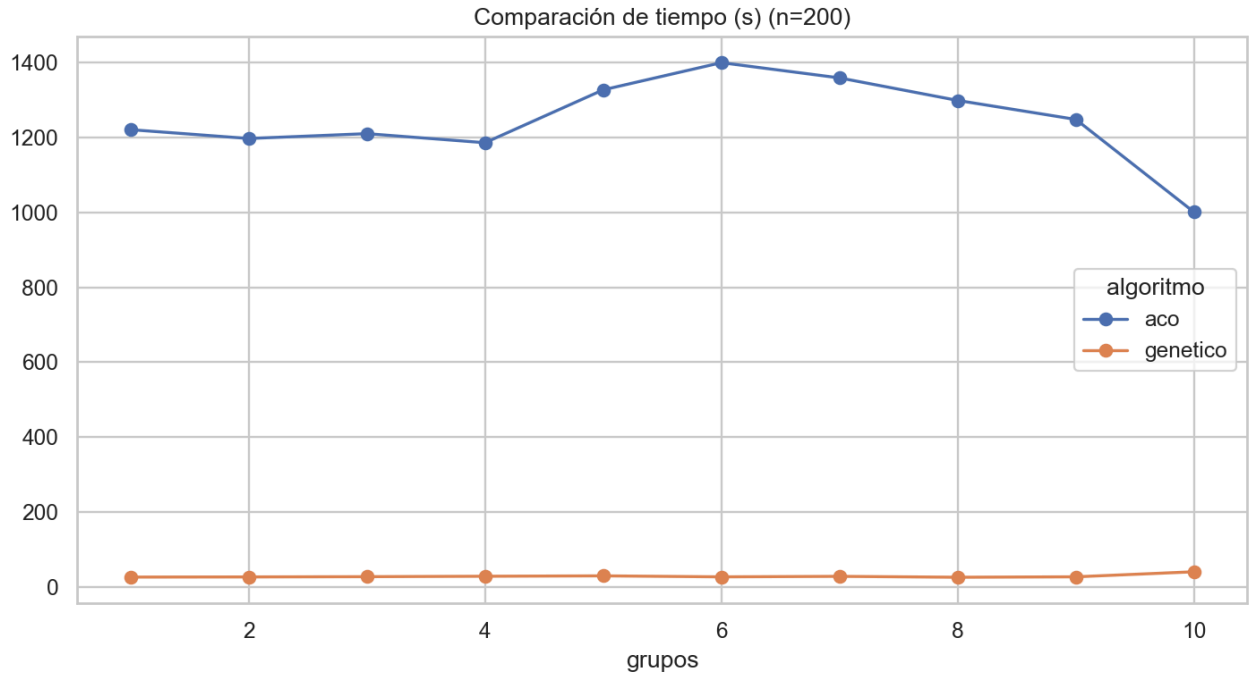


Tecnológico de Monterrey






Comparación de distancia (n=200)
1800
1600
1400
1200
1000
800
600
400
2
4
6
8
10
grupos
algoritmo
aco
genetico


Comparación de tiempo (s) (n=200)
1400
1200
1000
800
600
400
200
0
2
4
6
8
10
grupos
algoritmo
aco
genetico

Tecnológico de Monterrey

Comparacion de metodos

1. **Distancia de la Solución (Calidad)Redes Pequeñas (n=40)**:
El método exacto (GAMS) generalmente encuentra la ruta más corta, ya que está diseñado para obtener la solución óptima. Para n=40, las distancias de GAMS (línea discontinua negra) son consistentemente las más bajas o están muy cerca de las más bajas en el gráfico de comparación de distancia.
Los algoritmos heurísticos (AG y ACO) alcanzan soluciones que están cercanas al óptimo.

Redes Medianas y Grandes (n=100, 150, 200): Para estas instancias, el Algoritmo de Colonia de Hormigas (ACO) tiende a encontrar rutas con menor distancia (mejor calidad de solución) que el Algoritmo Genético (AG).

En las gráficas de n=100, 150, y 200, la línea de ACO (azul) se mantiene significativamente por debajo de la línea de AG (naranja) en la comparación de distancia.

Tiempo de Cómputo

Método Exacto (GAMS): Es el método menos eficiente en términos de tiempo de ejecución, especialmente a medida que n aumenta. Para n=40, aunque la distancia es mejor, el tiempo de GAMS muestra un pico mucho mayor que los métodos heurísticos en algunas instancias (grupo 9, por ejemplo), evidenciando su alto costo computacional.

Para instancias de gran tamaño, la complejidad factorial del TSP hace que la solución exacta sea intratable computacionalmente o extremadamente costosa, lo que puede llevar a reportar solo la mejor solución factible hasta un límite de tiempo.

Algoritmos Bioinspirados (AG y ACO): Ambos demuestran ser significativamente más rápidos y eficientes que el método exacto, lo que los hace adecuados para instancias de gran tamaño.

Para n=100, 150, y 200, el AG exhibe consistentemente el tiempo de cómputo más bajo (línea naranja cerca de cero), mientras que el ACO consume mucho más tiempo (línea azul, que va desde unos 300 segundos para n=100 hasta más de 1200 segundos para n=200.

Conclusiones

El análisis del Problema del Agente Viajero (TSP) confirma su clasificación como un problema NP-duro, cuyo costo computacional crece exponencialmente con el número de nodos, volviendo a la obtención de soluciones exactas extremadamente costosa o irrealizable en instancias de gran tamaño.

El estudio comparativo de los métodos de solución arrojó las siguientes conclusiones principales:

Método Exacto (GAMS): Garantiza la solución óptima (la ruta más corta) para el TSP.

Sin embargo, su alto costo computacional y sus extensos tiempos de ejecución lo hacen impráctico para redes grandes, como se observa en los casos donde no concluye dentro del tiempo límite o se reporta solo la mejor solución factible.

Algoritmos Bioinspirados (AG y ACO): Demostraron ser alternativas altamente efectivas para encontrar soluciones de alta calidad en un tiempo de ejecución considerablemente menor.

El Algoritmo Genético (AG) destaca por su eficiencia y rapidez, obteniendo soluciones aceptables en el menor tiempo de cómputo para todas las instancias.

Es ideal cuando la restricción principal es el tiempo.El Algoritmo de Colonia de Hormigas (ACO), aunque consume más tiempo que el AG, consistentemente encontró soluciones de mayor calidad (rutas más cortas) que el AG en las instancias grandes ($n > 100$), lo que sugiere una mejor capacidad de convergencia cercana al óptimo.

En conclusión, el estudio demuestra la eficiencia y adaptabilidad de los algoritmos bioinspirados (AG y ACO) como herramientas robustas para resolver problemas de optimización combinatoria complejos como el TSP. Estos métodos ofrecen un equilibrio adecuado entre una solución satisfactoria y la eficiencia computacional, lo que los hace la opción más práctica para la secuenciación de tareas o rutas en redes de gran escala.

Anexo

Github
En el siguiente link se puede revisar la implementación y resultados de cada una de las instancias.

https://github.com/Valg234/Bioinspirados

Infografía
Nuestra infografía
https://www.canva.com/design/DAG3CFxvKWw/vwKQMP7-KFwDxfvDdTxJuA/edit?utm_content=DAG3CFxvKWw&utm_campaign=designshare&utm_medium=link2&utm_source=sharebutton

Tabla de tiempos de ejecución y distancias obtenidas
https://docs.google.com/spreadsheets/d/1tqiBBA8XCRQWotWb2fmdCaTqKbF2P63TbvYArQmD0os/edit?usp=sharing

Bibliografía

Geeks for geeks. (2025, July 23). *Travelling Salesman Problem using Dynamic Programming*. GeeksforGeeks. Retrieved October 28, 2025, from https://www.geeksforgeeks.org/dsa/travelling-salesman-problem-using-dynamic-programming/

Papadimitriou, C. H., & Steiglitz, K. (1976). Some complexity results for the traveling salesman problem. In Proceedings of the eighth annual ACM symposium on Theory of computing (pp. 1–9). ACM. https://doi.org/10.1145/800113.803625

W3School. (n.d.). *DSA The Traveling Salesman Problem*. W3Schools. Retrieved October 28, 2025, from https://www.w3schools.com/dsa/dsa_ref_traveling_salesman.php

Zeybek, S. (2024, April 19). *Ant Colony Algorithm*. Medium. https://serdarzeybek.medium.com/ant-colony-algorithm-15a37c3a1671