UNIVERSIDAD DE INGENIERÍA Y TECNOLOGÍA
MAESTRÍA EN CIENCIA DE DATOS & IA

IA FUNDAMENTALS
Práctica 02

Grupo 4:
Fernando Ivan Gutiérrez Delgado
César Eduardo Poma Legua
Julio César Mogollon Vilca
Diego Remigio Perez Borromeo

Docente:
Vicente Machaca Arceda, Ph.D.

Lima, 2025

Índice

Índice de figuras

1. Introducción

Actualmente en la empresa se dispone de vehículos para realizar la movilización del personal técnico quienes realizan las actividades diarias de conexiones eléctricas en los distintos distritos de la ciudad de Lima. El problema radicó en la búsqueda de optimizar el tiempo total de atención mediante la determinación del orden óptimo, en que la cuadrillas deben realizar las visitas a cada punto de conexión. Para este problema planteado se encontró dos soluciones mediante el uso de dos algoritmos: a) Hill Climbing (H.C) y b)Simulated Annealing (S.A); estos algoritmos fueron alimentados con información proporcionada por el software GIS en donde se tiene la asignación de las órdenes de trabajo.

2. Metodología

2.1. Solución

Algorithm 1: Solución para el problema de VRP usando H.C.

Input: positions: lista de coordenadas (x,y) incluyendo el depósito (ubicado en el índice 0,0);
n_vehicles: número de vehículos
Output: current: mejor solución encontrada;
current_fitness: fitness de la mejor solución;
init: solución inicial;
init_fitness: fitness de la solución inicial

Inicialización;
TSP ← build_TSP(positions);
current ← initial solution(len(positions), n_vehicles);
init ← copy(current);

Bucle principal;
while *true* **do**
    neighbors ← get_neighbours(current);
    best ← selecciona el mejor vecino;
    if *fitness(best, TSP) < fitness(current, TSP)* **then**
        current ← best;
    end
    else
        return (current, fitness(current, TSP), init, fitness(init, TSP));
    end
end

Algorithm 2: Solución para el problema de VRP usando S.A.

Input: *positions*: lista de coordenadas (x,y) incluyendo el depósito (ubicado en el índice 0,0);
n_vehicles: número de vehículos;
T_max: temperatura máxima;
T_min: temperatura mínima;
α: ratio de enfriamiento
Output: x: mejor solución encontrada;
E: fitness de la mejor solución;
sol_init: solución inicial;
init_cost_sa: fitness de la solución inicial

Inicialización;
TSP $\leftarrow$ build_TSP(positions);
$x \leftarrow$ initial solution(len(positions), n_vehicles);
$E \leftarrow$ fitness(x, TSP);
sol_init $\leftarrow$ copy(x);
$T \leftarrow T\_max$

Bucle principal;
while *($T > T\_min$)* **do**
    x_new $\leftarrow$ random_neighbour(x);
    E_new $\leftarrow$ fitness(x, TSP);
    $\Delta E \leftarrow E\_new - E$;
    if *accept(ΔE, T)* **then**
        x $\leftarrow$ x_new;
        E $\leftarrow$ E_new;
    end
    $T \leftarrow T\alpha$;
end
return (x, E, sol_init, fitness(sol_init, TSP))

Para ambos casos, se probaron ambos algoritmos para una combinación de número de 10, 25 y 50 localidades y un número de 2,3 y 4 vehículos. Luego, se generaron gráficos para cada algoritmo en cada combinación.

2.2. Obtención de vecinos

Para H.C, la generación de vecinos se realiza mediante un intercambio de nodos entre todas las rutas posibles: el código explora sistemáticamente cada par de rutas (i, j) y cada par de posiciones (a, b) dentro de ellas, creando nuevas soluciones donde los nodos en dichas posiciones son intercambiados, excepto cuando se trata de la misma posición en la misma ruta (i = j y a = b). Este enfoque genera un espacio de vecindario amplio que incluye tanto intercambios entre rutas diferentes (para redistribuir la carga entre vehículos) como dentro de una misma ruta (para optimizar el orden de visita), asegurando una búsqueda local detallada.

```python
def get_neighbors(routes):
    neighbors = []
    for i in range(len(routes)):
        for j in range(i, len(routes)):
            for a in range(len(routes[i])):
                for b in range(len(routes[j])):
                    if i == j and a == b: continue
                    new = copy.deepcopy(routes)
                    new[i][a], new[j][b] = new[j][b], new[i][a]
                    neighbors.append(new)
    return neighbors
```

Figura 1: Código para generar vecinos en H.C.

Para S.A, la generación de vecinos se realiza mediante perturbaciones aleatorias controladas sobre la solución actual: cuando hay un solo vehículo, se permutan aleatoriamente dos nodos dentro de su ruta (si existen al menos dos), mientras que en casos con múltiples vehículos, se intercambia un cliente elegido al azar entre dos rutas distintas (si ambas tienen al menos un cliente), asegurando así una exploración diversificada del espacio de soluciones mientras se mantiene la estructura básica de las rutas. Este enfoque equilibra la exploración de nuevas configuraciones con la explotación de soluciones prometedoras, permitiendo al algoritmo escapar de óptimos locales mediante cambios graduales y aleatorizados durante el proceso de búsqueda.

```python
def random_neighbor(sol):
    new = copy.deepcopy(sol)
    if len(sol) == 1:
        # Un solo vehículo: permutar dentro de la única ruta
        if len(sol[0]) >= 2:
            i, j = random.sample(range(len(sol[0])), 2)
            new[0][i], new[0][j] = new[0][j], new[0][i]
    else:
        # Múltiples vehículos: intercambiar clientes entre dos rutas
        r1, r2 = random.sample(range(len(sol)), 2)
        if sol[r1] and sol[r2]:
            i, j = random.randint(0, len(sol[r1])-1), random.randint(0, len(sol[r2])-1)
            new[r1][i], new[r2][j] = new[r2][j], new[r1][i]
    return new
```

Figura 2: Código para generar vecinos en S.A.

2.3. Fitness

La función de costo o fitness calcula la distancia total recorrida por todos los vehículos en la solución propuesta: para cada ruta no vacía, suma la distancia desde el depósito (nodo 0) al primer cliente (TSP[0][r[0]]), las distancias entre clientes consecutivos en la ruta (sum(TSP[r[i]][r[i+1]]), y la distancia desde el último cliente de vuelta al depósito (TSP[r[-1]][0]), acumulando estos valores para todas las rutas. Este enfoque penaliza directamente la longitud total de los recorridos.

```python
def fitness(routes, TSP):
    return sum(
        TSP[0][r[0]] + sum(TSP[r[i]][r[i+1]] for i in range(len(r)-1)) + TSP[r[-1]][0]
        for r in routes if r
    )
```

Figura 3: Código para generar la función de costo

3. Resultados

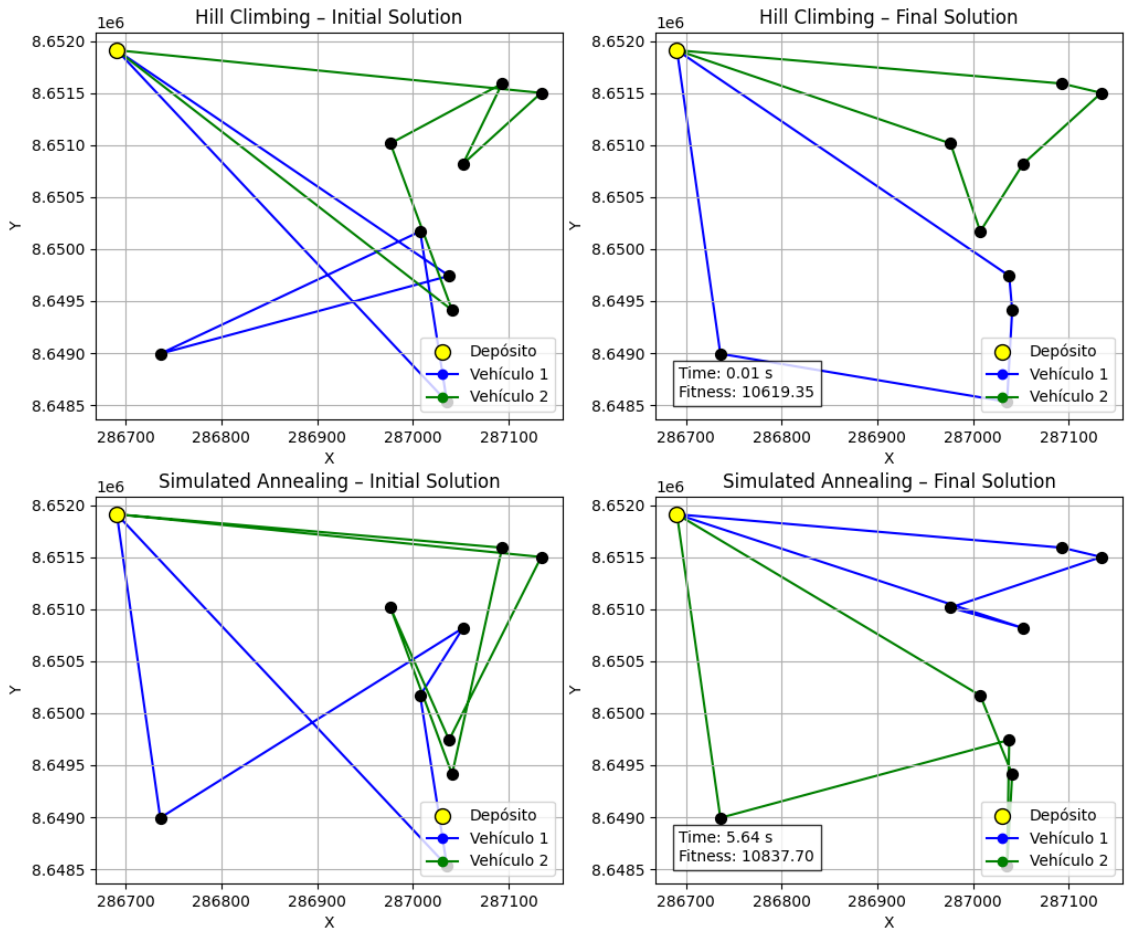

Figura 4: Solución inicial generada aleatoriamente y solución final para los algoritmos de H.C y S.A. Localidades=10 y vehículos=2

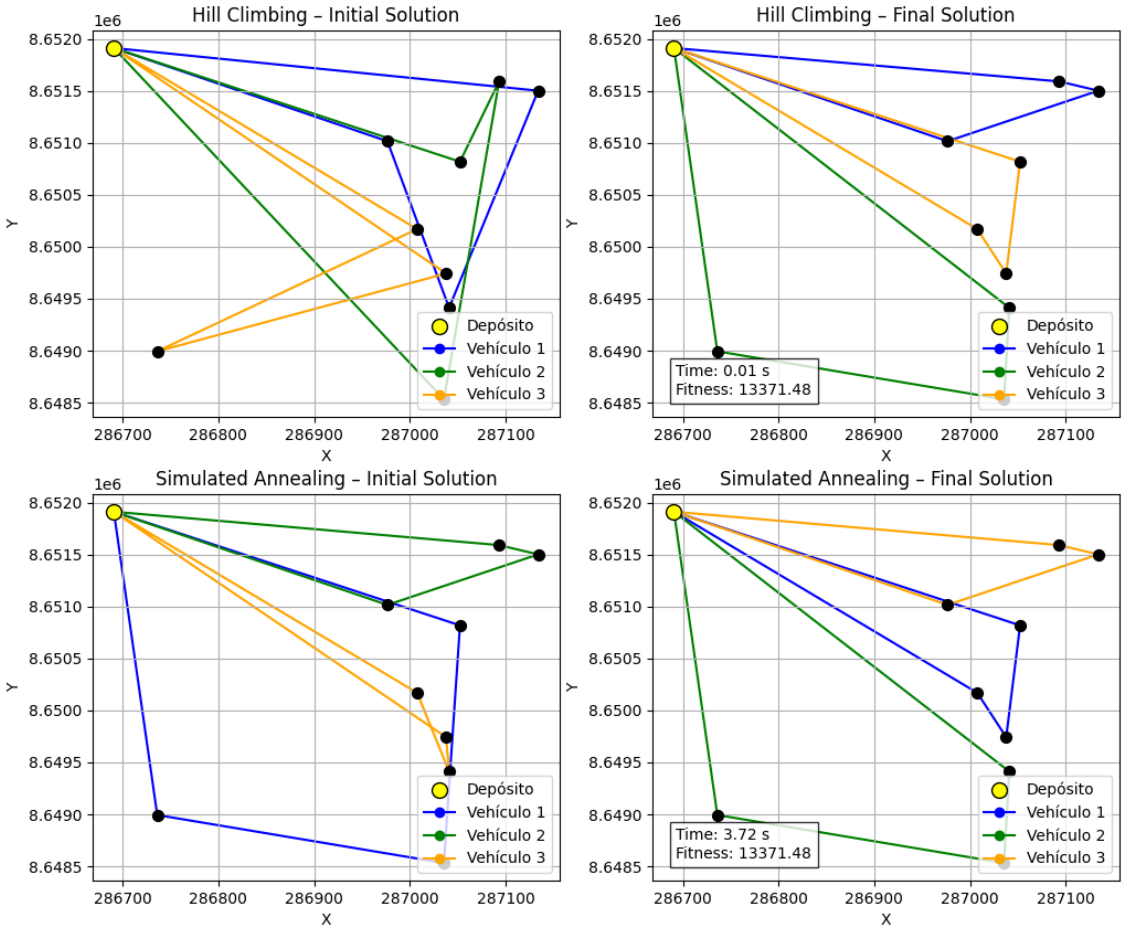

Figura 5: Solución inicial generada aleatoriamente y solución final para los algoritmos de H.C y S.A. Localidades=10 y vehículos=3

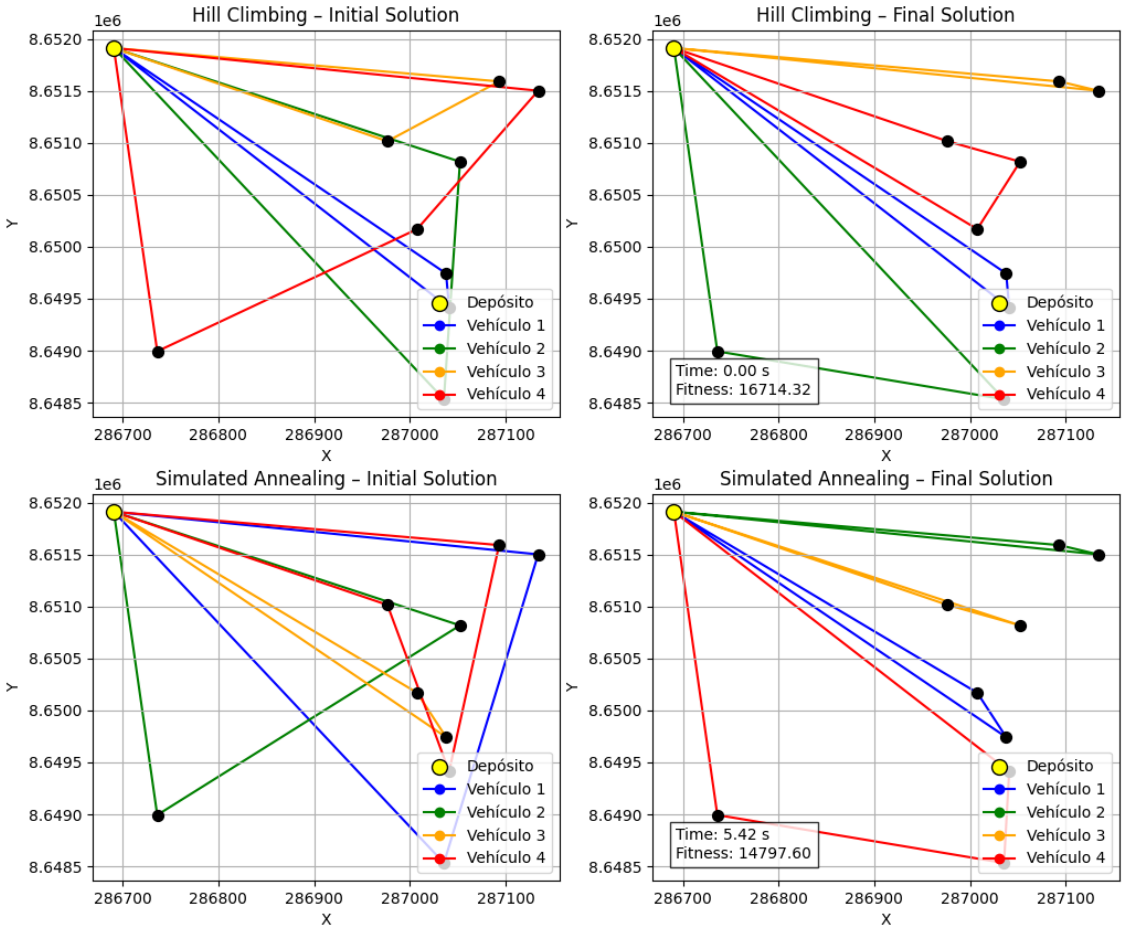

Figura 6: Solución inicial generada aleatoriamente y solución final para los algoritmos de H.C y S.A. Localidades=10 y vehículos=4

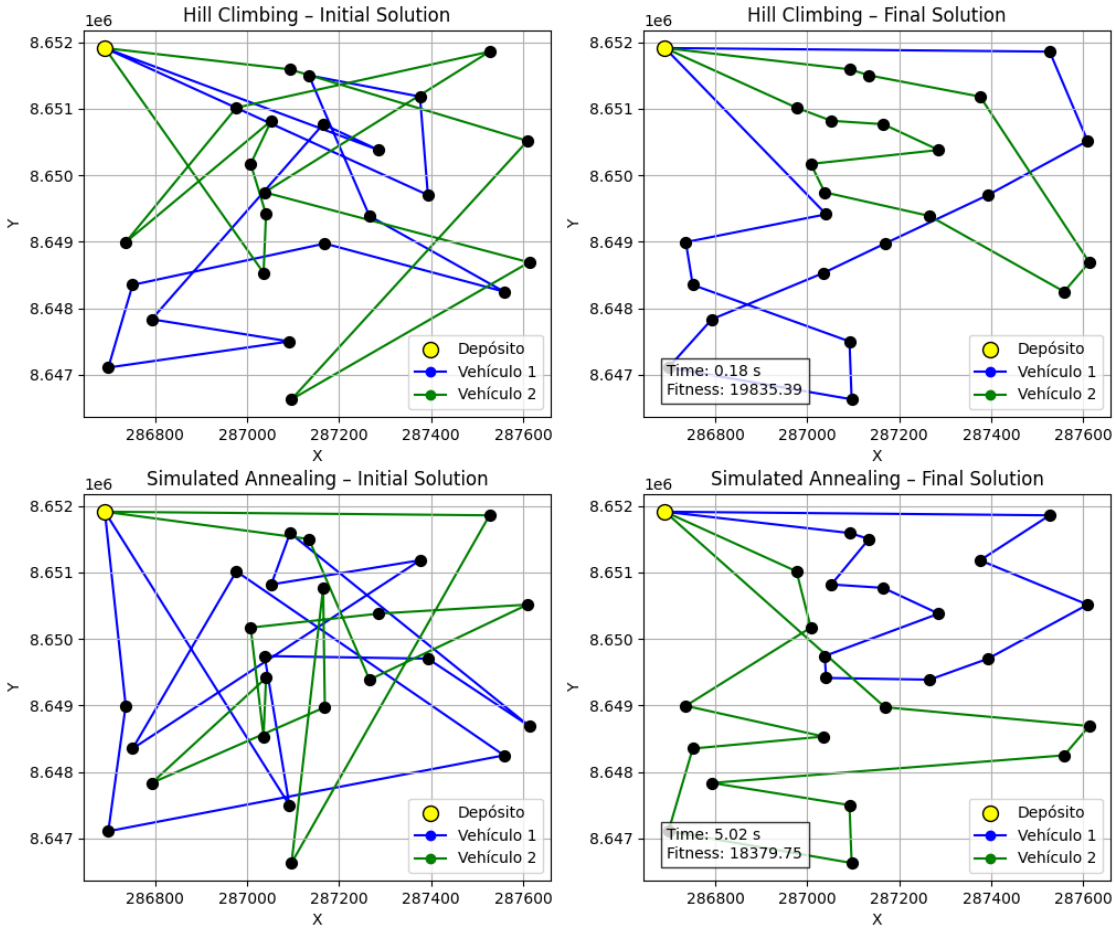

Figura 7: Solución inicial generada aleatoriamente y solución final para los algoritmos de H.C y S.A. Localidades=25 y vehículos=2

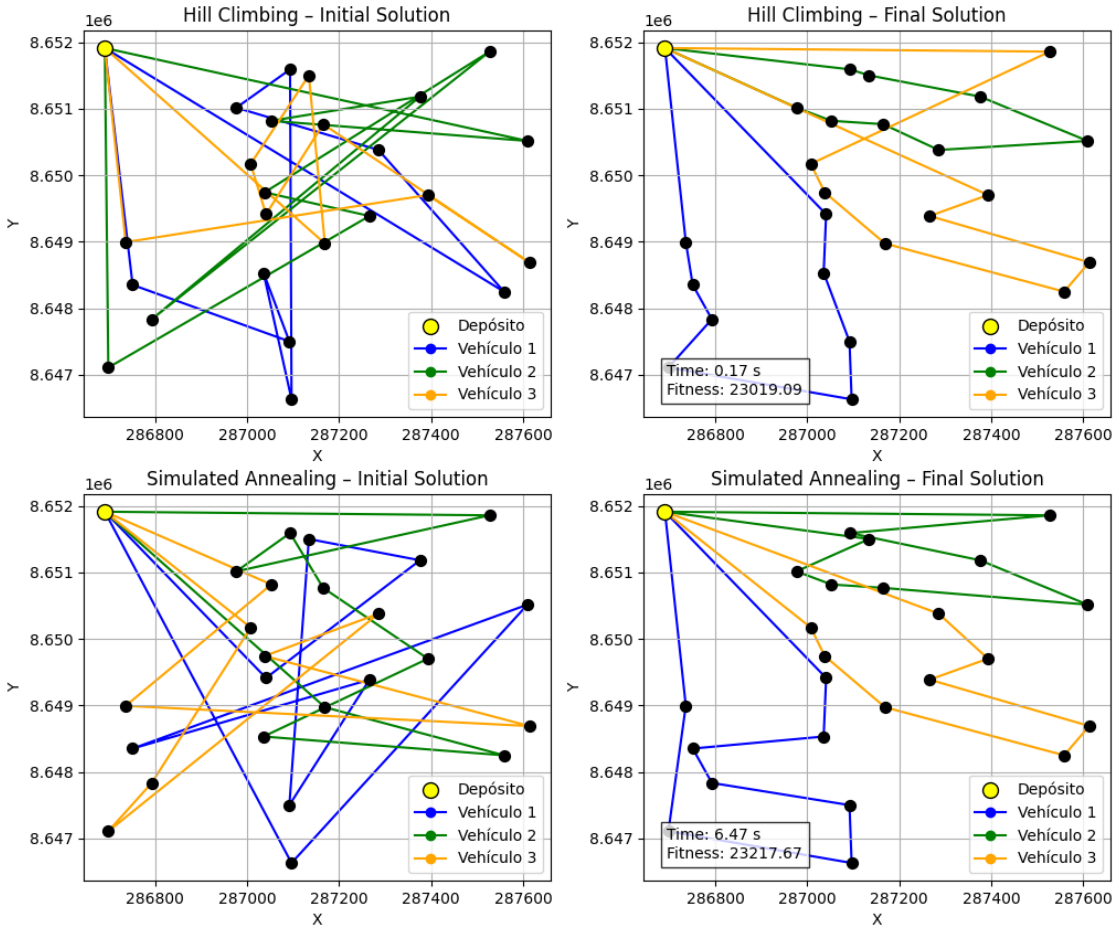

Figura 8: Solución inicial generada aleatoriamente y solución final para los algoritmos de H.C y S.A. Localidades=25 y vehículos=3

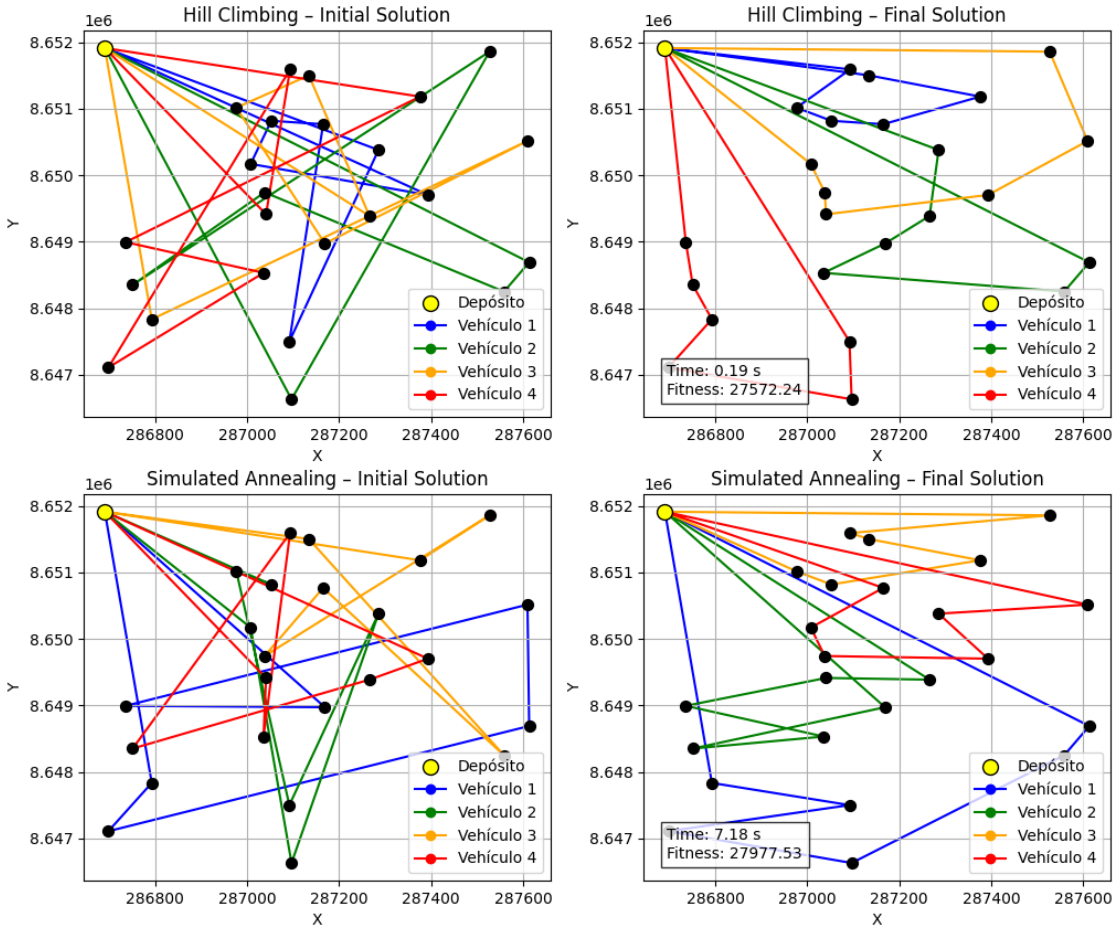

Figura 9: Solución inicial generada aleatoriamente y solución final para los algoritmos de H.C y S.A. Localidades=25 y vehículos=4

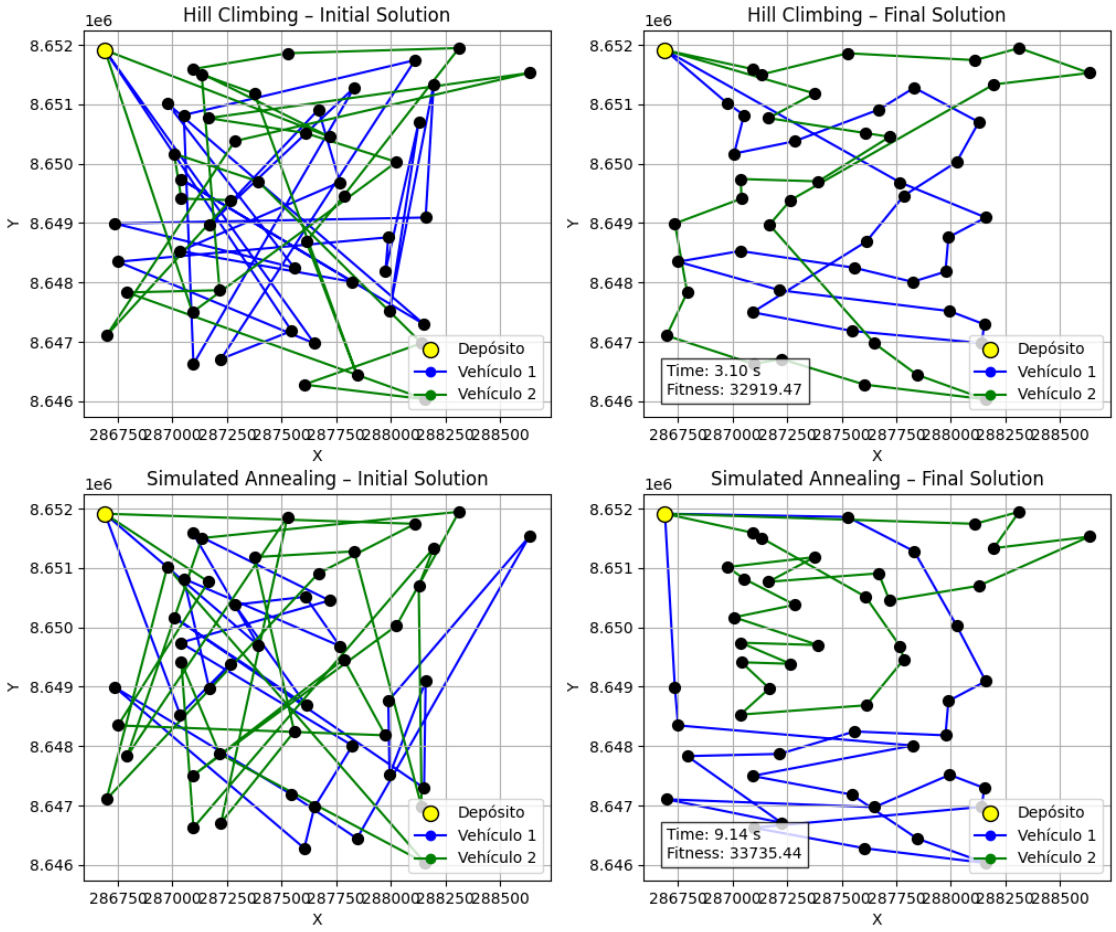

Figura 10: Solución inicial generada aleatoriamente y solución final para los algoritmos de H.C y S.A. Localidades=50 y vehículos=2

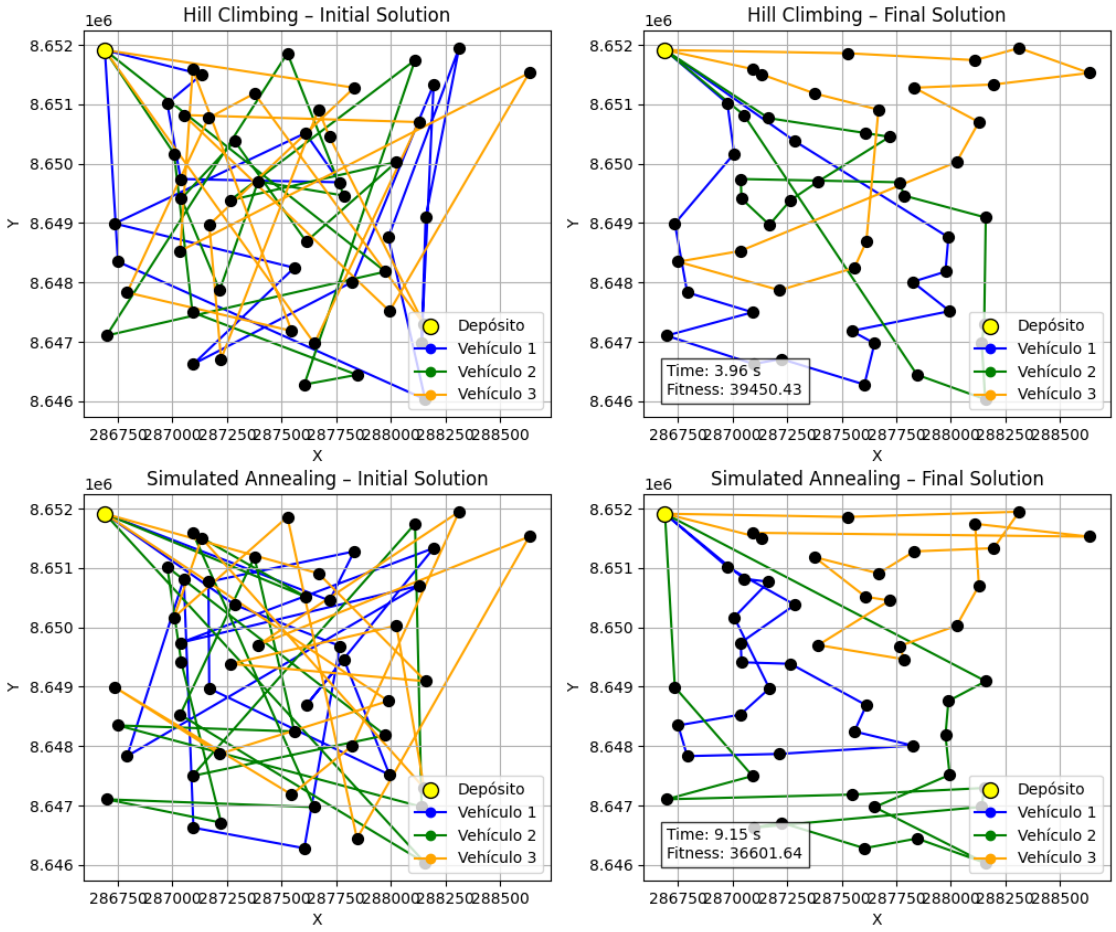

Figura 11: Solución inicial generada aleatoriamente y solución final para los algoritmos de H.C y S.A. Localidades=50 y vehículos=3

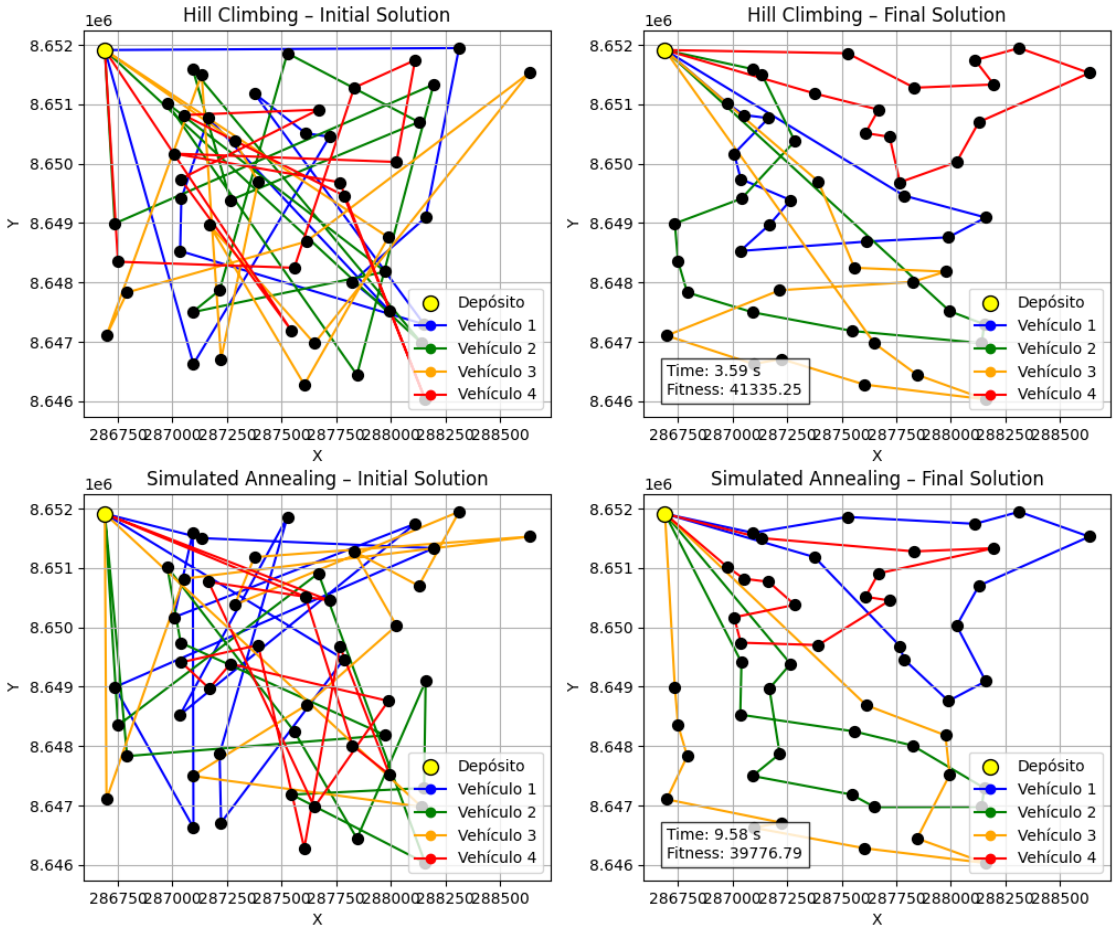

Figura 12: Solución inicial generada aleatoriamente y solución final para los algoritmos de H.C y S.A. Localidades=50 y vehículos=4

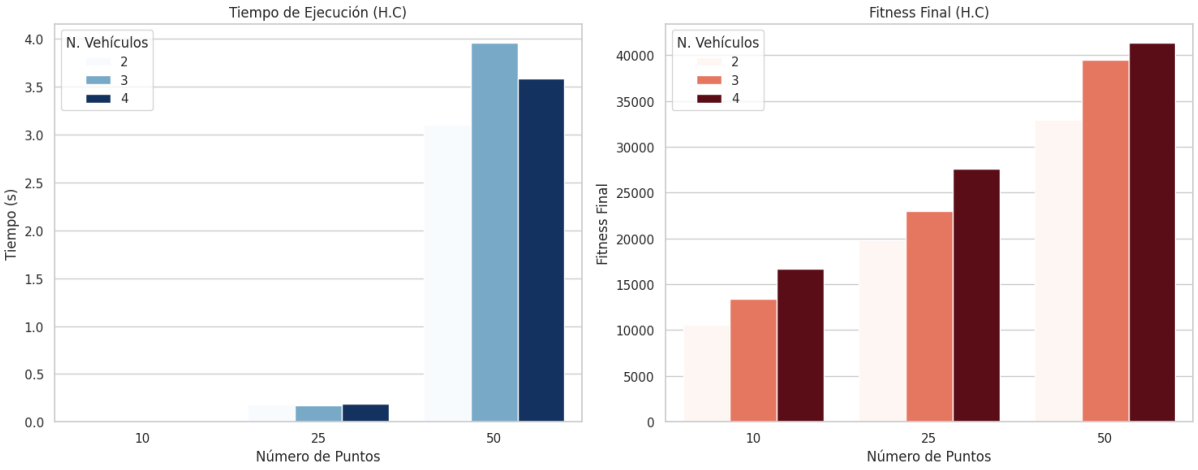

Figura 13: Tiempo de ejecución y Fitness final para H.C

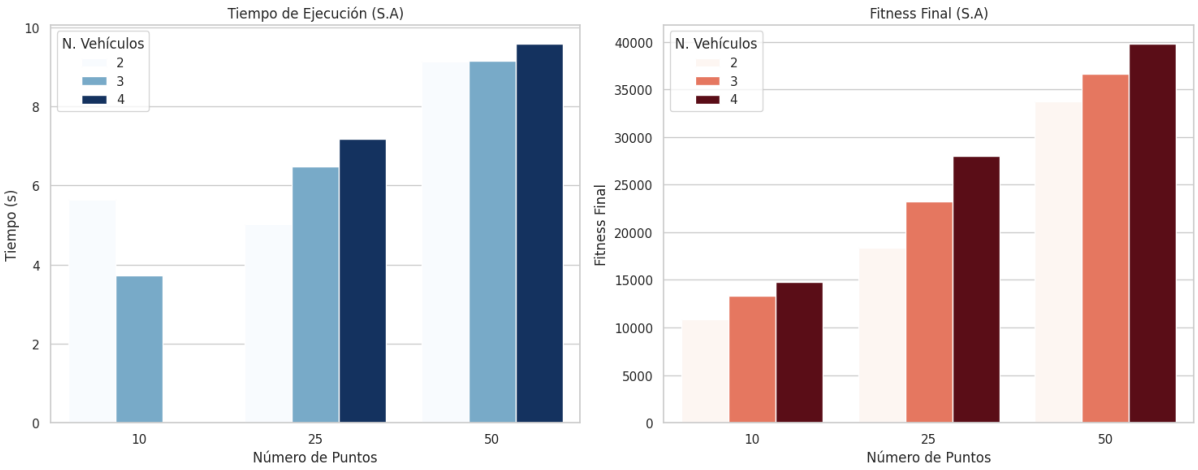

Figura 14: Tiempo de ejecución y Fitness final para S.A

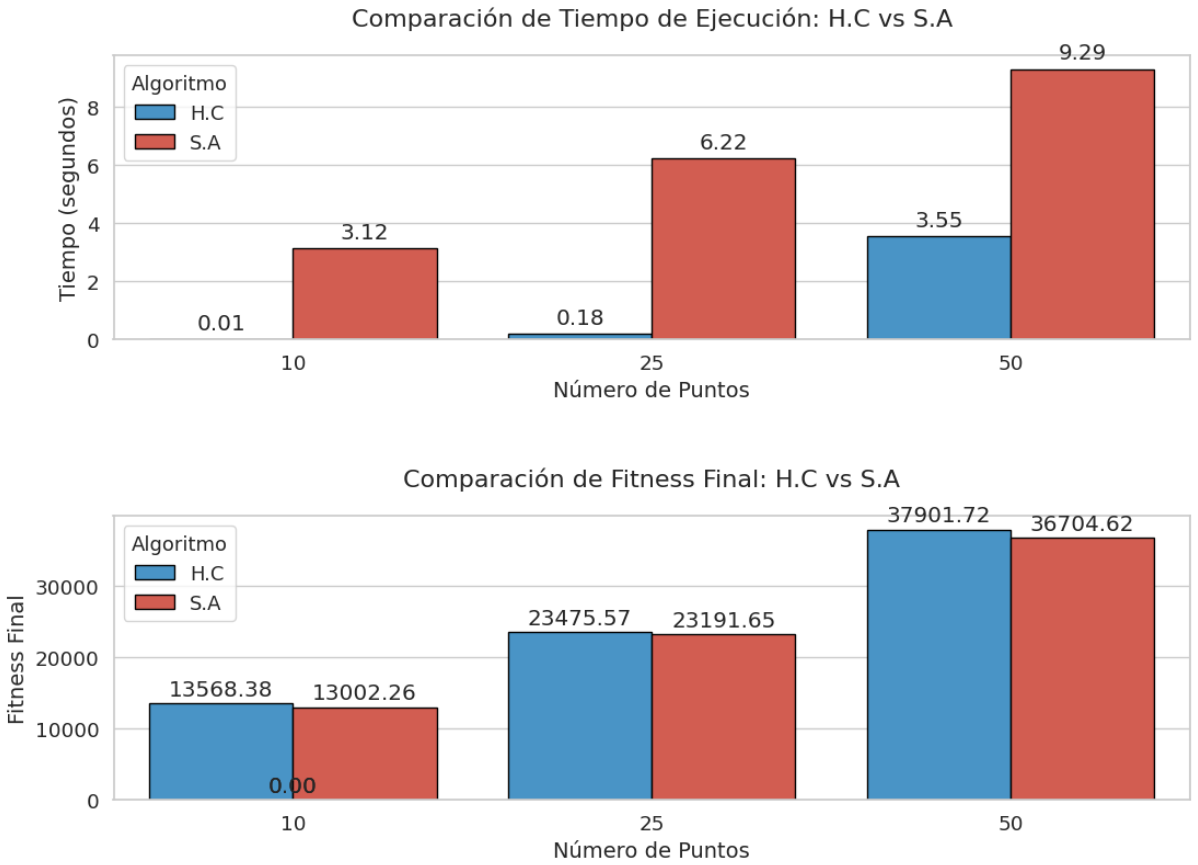

Figura 15: Comparación entre H.C y S.A.

4. Conclusiones

- H.C es óptimo para aplicaciones que requieren baja latencia, mientras que S.A, aunque más lento, podría justificarse si ofrece mejores resultados en fitness.
- En la mayoría de casos, H.C domina en la relación costo-beneficio (fitness similar o mejor con tiempos drásticamente menores). Aunque en algunos casos S.A ofrece una mejora en términos de fitness, es marginal comparada con H.C.
- Normalmente se espera que S.A tenga un performance mejor que H.C, en el análisis realizado esto no se cumple. Una de las causas puede ser que el aumento en el número de vehículos influye significativamente cuando el número de locales a recorrer es relativamente pequeño. Además, se esperaría que la tendencia se revierta al aumentar aun más el número de locales (ej. 100, 500 o 10000).
- H.C es la mejor opción para este análisis en particular, dado su equilibrio superior entre fitness y tiempo. S.A podría explorarse con ajustes (e.g., modificar la tasa de enfriamiento o la temperatura máxima) si se requieren soluciones más diversas, pero con el costo actual no resulta óptimo. Esto no significa que H.C sea mejor para el problema general de VRP, ya que se deben considerar restricciones adicionales.