El Problema del Viajante de Comercio:
Algoritmos Heurísticos para la Optimización de Rutas

Lee Sang-cheol
Carné: 2024801079

Estructuras de Datos y Algoritmos
Prof. Victor Manuel Garro Abarca

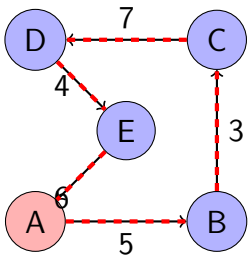
Septiembre 2025

Agenda

¿Qué es el TSP?

Problema del Viajante de Comercio

- Visitar n ciudades exactamente una vez
- Regresar al origen
- Minimizar la distancia total
- Problema NP-Completo



Complejidad:

- $(n-1)!/2$ rutas posibles
- 20 ciudades $= 6 \times 10^{16}$ rutas

Tour: A→B→C→D→E→A
Distancia: 25

Aplicaciones en el Mundo Real

Logística y Transporte

- Rutas de entrega (Amazon, UPS)
- Recolección de basura
- Transporte escolar

Manufactura

- Perforación de PCB
- Corte con láser
- Soldadura robotizada

Otros Campos

- Secuenciación de ADN
- Planificación de telescopios
- Diseño de microchips
- Videojuegos (pathfinding)

Impacto Económico

Una mejora del 1 % en rutas de entrega puede ahorrar millones de dólares anuales

Estructuras de Datos para Grafos Ponderados

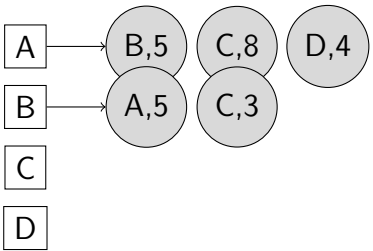
Matriz de Adyacencia

- Acceso $O(1)$ a distancias
- Espacio $O(n^2)$
- Ideal para grafos densos (TSP)

	A	B	C	D
A	0	5	8	4
B	5	0	3	7
C	8	3	0	2
D	4	7	2	0

Lista de Adyacencia

- Espacio $O(|V| + |E|)$
- Iteración eficiente sobre vecinos
- Mejor para grafos dispersos



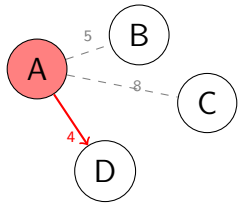
Estrategia

1. Comenzar en ciudad aleatoria
2. Visitar la ciudad no visitada más cercana
3. Repetir hasta visitar todas
4. Regresar al origen

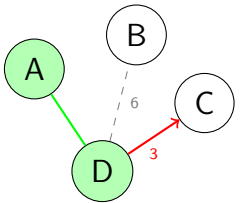
Características

- Complejidad: $O(n^2)$
- Solución: $\sim$25 % del óptimo
- Muy rápido
- Fácil de implementar

Paso a paso:



Elegir D (más cercano)



Siguiente: C

Implementación: Vecino Más Cercano

```python
def vecino_mas_cercano(dist_matrix, inicio=0):
    n = len(dist_matrix)
    visitado = [False] * n
    tour = [inicio]
    visitado[inicio] = True
    ciudad_actual = inicio
    costo_total = 0

    # Visitar n-1 ciudades
    for _ in range(n - 1):
        min_dist = float('inf')
        ciudad_mas_cercana = -1

        # Encontrar la ciudad no visitada mas cercana
        for j in range(n):
            if not visitado[j] and dist_matrix[ciudad_actual][j] < min_dist:
                min_dist = dist_matrix[ciudad_actual][j]
                ciudad_mas_cercana = j

        # Agregar al tour
        tour.append(ciudad_mas_cercana)
        visitado[ciudad_mas_cercana] = True
        costo_total += min_dist
        ciudad_actual = ciudad_mas_cercana

    # Regresar al inicio
    costo_total += dist_matrix[ciudad_actual][inicio]
    tour.append(inicio)

    return tour, costo_total
```

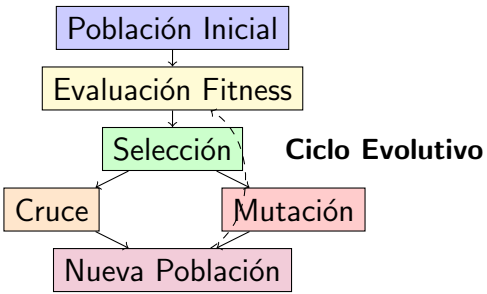
Algoritmos Genéticos: Conceptos

Inspiración Biológica

- **Población**: Conjunto de soluciones
- **Cromosoma**: Tour = [A,C,B,D,A]
- **Fitness**: 1/distancia
- **Selección**: Torneos
- **Cruce**: Combinar padres
- **Mutación**: Cambios aleatorios

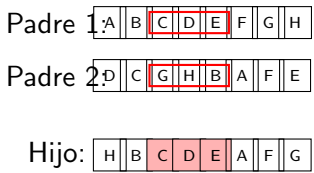
Ventajas

- Exploración global
- Paralelizable
- Evita óptimos locales

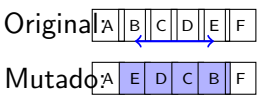


Operadores Genéticos para TSP

Order Crossover (OX)



Mutación 2-opt



Parámetros Clave

- Tamaño población: 100-200
- Generaciones: 1000-5000
- Prob. cruce: 0.8-0.9
- Prob. mutación: 0.01-0.05
- Selección: Torneo (tamaño 3-7)

Complejidad

- Tiempo: $O(g \cdot p \cdot n^2)$
- Espacio: $O(p \cdot n)$

donde g=generaciones, p=población

Recocido Simulado: Fundamentos

Analogía Metalúrgica

- Calentar metal (alta temperatura)
- Enfriar lentamente
- Átomos encuentran configuración óptima

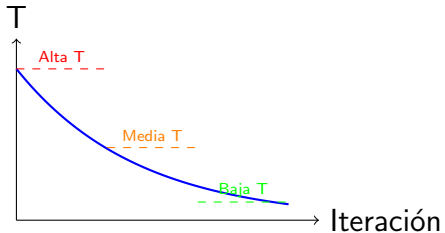
Probabilidad de Aceptación

$$P = \begin{cases} 1 & \text{si } \Delta < 0 \\ e^{-\Delta/T} & \text{si } \Delta \geq 0 \end{cases}$$

Esquema de Enfriamiento

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

donde $\alpha \in (0{,}9, 0{,}999)$



Exploración → Explotación

Ventaja Clave

Acepta soluciones peores probabilísticamente para escapar de óptimos locales

Implementación: Recocido Simulado

```python
def recocido_simulado(dist_matrix, temp_inicial=1000, alpha=0.995):
    n = len(dist_matrix)
    # Solucion inicial aleatoria
    tour_actual = list(range(n))
    random.shuffle(tour_actual)
    costo_actual = calcular_costo(tour_actual, dist_matrix)

    mejor_tour = tour_actual[:]
    mejor_costo = costo_actual
    temperatura = temp_inicial

    while temperatura > 0.1:
        # Generar vecino (intercambio 2-opt)
        i, j = sorted(random.sample(range(n), 2))
        tour_vecino = tour_actual[:]
        tour_vecino[i:j+1] = tour_vecino[i:j+1][::-1]
        costo_vecino = calcular_costo(tour_vecino, dist_matrix)

        # Decidir si aceptar
        delta = costo_vecino - costo_actual
        if delta < 0 or random.random() < math.exp(-delta/temperatura):
            tour_actual = tour_vecino
            costo_actual = costo_vecino

            if costo_actual < mejor_costo:
                mejor_tour = tour_actual[:]
                mejor_costo = costo_actual

        # Enfriar
        temperatura *= alpha
```

Comparación de Rendimiento

Calidad de Solución (Gap %)

Tiempo de Ejecución

Conclusión

- **Voraz**: Muy rápido pero calidad baja
- **GA**: Buena calidad, paralelizable
- **SA**: Mejor balance calidad/tiempo

Caso de Estudio: Optimización de Entregas

Empresa de Logística - San José

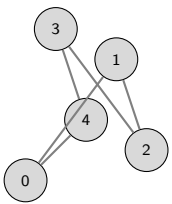
- 50 puntos de entrega diarios
- Implementación: SA + 2-opt
- Tiempo de cálculo: 5 segundos

Resultados Obtenidos

- $\downarrow$ 22 % distancia total
- $\downarrow$ \$3,500/mes combustible
- $\downarrow$ 2 horas $\rightarrow$ 5 min planificación
- $\uparrow$ 15 % entregas/día

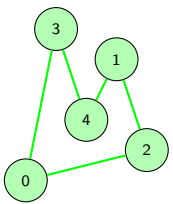
Antes vs Después

Ruta Manual



145 km

Ruta Optimizada



113 km

ROI

Retorno de inversión en 2 meses

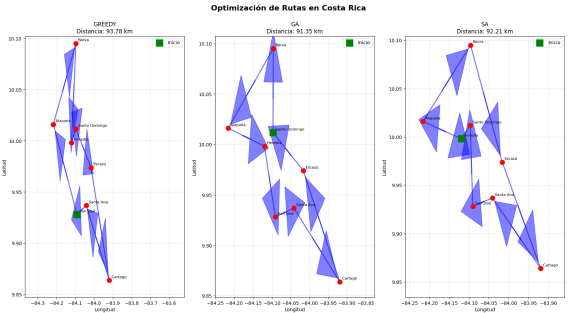
Caso Real: Optimización de Rutas en Costa Rica

Datos GPS Reales

- 27 ciudades de Costa Rica
- Coordenadas GPS precisas
- Distancias reales (Haversine)

Resultados Valle Central

- 8 ciudades principales
- 25.5 % reducción
- 72.7 km ahorrados



Recomendaciones por Escenario

Escenario	Algoritmo	Razón
Tiempo real (¡1s)	Vecino Cercano	$O(n^2)$, muy rápido
Calidad media	Inserción + 2-opt	Balance calidad/tiempo
Alta calidad	Híbrido GA+SA	Exploración + explotación
n ¡ 20 ciudades	Branch & Bound	Solución exacta posible
n ¿ 500 ciudades	SA paralelo	Mejor escalabilidad
Múltiples objetivos	GA multiobjetivo	Pareto optimal

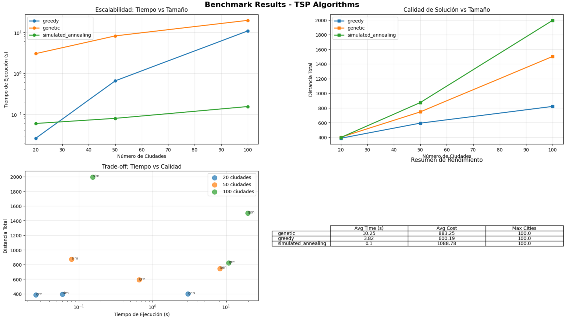
Para Implementación

- Comenzar con algoritmo voraz
- Aplicar mejora 2-opt
- Si necesario, usar SA o GA
- Considerar hibridación

Optimización Avanzada

- Paralelización
- Estructuras de datos eficientes
- Cache de distancias
- GPU computing para GA

Análisis de Rendimiento: Benchmark



	Avg Time (s)	Avg Cost	Max Cities
genetic	10.25	883.25	100.0
greedy	3.82	600.19	100.0
simulated_annealing	0.1	1088.78	100.0

Trade-off Tiempo vs Calidad:

- Greedy: Rápido pero subóptimo
- GA: Mejor calidad, más tiempo
- SA: Balance óptimo

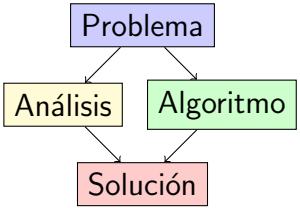
Conclusiones

Puntos Clave

- TSP es NP-Completo pero tiene soluciones prácticas
- No existe "mejor.ªlgoritmo universal
- La elección depende del contexto:
  - Tamaño del problema
  - Tiempo disponible
  - Calidad requerida

Tendencias Futuras

- Machine Learning para TSP
- Computación Cuántica
- TSP dinámico y en tiempo real
- Optimización multi-objetivo



Proceso de Decisión

¿Preguntas?

Gracias por su atención

Lee Sang-cheol
Carné: 2024801079
s.lee.1@estudiantec.cr

Referencias

1. Applegate, D. L., et al. (2006). *The Traveling Salesman Problem: A Computational Study*. Princeton University Press.
2. Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley.
3. Kirkpatrick, S., Gelatt, C. D., & Vecchi, M. P. (1983). "Optimization by Simulated Annealing". *Science*, 220(4598), 671-680.
4. Helsgaun, K. (2000). "An effective implementation of the Lin-Kernighan traveling salesman heuristic". *European Journal of Operational Research*, 126(1), 106-130.
5. Dorigo, M., & Gambardella, L. M. (1997). "Ant colony system: a cooperative learning approach to the traveling salesman problem". *IEEE Transactions on Evolutionary Computation*, 1(1), 53-66.
6. TSPLIB: `http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/`

Código fuente disponible en:

`https://github.com/usuario/tsp-heuristics`

[Backup] Análisis de Complejidad Detallado

Algoritmo	**Mejor**	**Promedio**	**Peor**
Fuerza Bruta	$O(n!)$	$O(n!)$	$O(n!)$
Vecino Cercano	$O(n^2)$	$O(n^2)$	$O(n^2)$
Inserción	$O(n^2)$	$O(n^2)$	$O(n^3)$
2-opt	$O(n^2)$	$O(n^3)$	$O(n^3)$
GA (g generaciones)	$O(g \cdot p \cdot n)$	$O(g \cdot p \cdot n^2)$	$O(g \cdot p \cdot n^2)$
SA (k iteraciones)	$O(k \cdot n)$	$O(k \cdot n)$	$O(k \cdot n)$

Espacio

- Matriz de distancias: $O(n^2)$
- Tour: $O(n)$
- GA población: $O(p \cdot n)$

[Backup] Pseudocódigo Completo - Algoritmo Genético

```
ALGORITMO GENETICO_TSP:
    ENTRADA: matriz_distancias, tam_poblacion, num_generaciones
    SALIDA: mejor_tour, mejor_distancia

    poblacion = GENERAR_POBLACION_INICIAL(tam_poblacion)
    mejor_tour = NULL
    mejor_fitness = 0

    PARA gen = 1 HASTA num_generaciones:
        fitness = EVALUAR_POBLACION(poblacion)

        SI max(fitness) > mejor_fitness:
            mejor_fitness = max(fitness)
            mejor_tour = tour_con_mejor_fitness

        nueva_poblacion = []
        MIENTRAS tam(nueva_poblacion) < tam_poblacion:
            padre1 = SELECCION_TORNEO(poblacion)
            padre2 = SELECCION_TORNEO(poblacion)
            hijo = CRUCE_OX(padre1, padre2)
            hijo = MUTACION_2OPT(hijo, prob_mutacion)
            nueva_poblacion.agregar(hijo)

        poblacion = nueva_poblacion

    RETORNAR mejor_tour, 1/mejor_fitness
```

[Backup] Variantes del Problema TSP

TSP Simétrico vs Asimétrico

- Simétrico: $d(i,j) = d(j,i)$
- Asimétrico: $d(i,j) \neq d(j,i)$
- Ejemplo: calles de un solo sentido

TSP con Ventanas de Tiempo

- Cada ciudad tiene horario
- Penalización por llegar tarde/temprano
- Aplicación: entregas programadas

Multiple TSP (mTSP)

- Múltiples viajantes
- Minimizar distancia total
- Balancear cargas de trabajo

TSP Dinámico

- Ciudades aparecen/desaparecen
- Distancias cambian (tráfico)
- Requiere reoptimización online

Nota

Cada variante requiere adaptaciones específicas de los algoritmos base