Instituto Tecnológico Autónomo de México

Resolviendo el problema del agente viajero usando distintas heurísticas

Investigación de Operaciones
Profesora: Mayra Nuñez López

Juan Pablo Macías Watson CU: 192984
Erick Martínez Henández CU: 191821

Índice

1. Introducción

El Problema del Agente Viajero (TSP, por sus siglas en inglés, *Traveling Salesman Problem*) es uno de los problemas más estudiados en la teoría de la optimización, combinatoria e investigación de operaciones. A pesar de esto, es un problema muy fácil de entender; encontrar su solución, esa es otra historia.

Considere un grupo de n ciudades, el objetivo es lograr visitar cada una de las ciudades, sin repetir (cada ciudad se visita solamente una vez), y regresar a la ciudad donde se inició el recorrido. Es decir, se busca encontrar la ruta cerrada óptima, en términos de menor distancia, que recorra todas las ciudades. Por esto, el problema solo tiene 2 grupos de datos:

- El número de ciudades, n
- Una matriz de distancias D donde d_{ij} es la distancia para ir de la ciudad i a la j:
  - En el caso en el que el problema contenga algún par de ciudades que no sea posible ir de i a j se dice que $d_{ij} = \infty$.
  - El número máximo de recorridos en una situación de n ciudades es $(n-1)!$.
  - Para fines prácticos en nuestro problema, es lo mismo ir de i a j que de j a i, es decir: $d_{ij} = d_{ji}$

Este problema fue formulado en el siglo XIX por los matemáticos William Rowan Hamilton y Thomas Kirkman, ellos estaban intentando encontrar ciclos hamiltonianos en grafos. Sin embargo, la versión moderna del problema, de la cual hablamos al principio, fue popularizada a mediados del siglo XX, época cuando se empezó a buscar posibles soluciones a este problema.

El TSP tiene una amplia gama de aplicaciones prácticas, incluyendo la planificación de rutas de distribución, la organización de circuitos de fabricación, la secuenciación de ADN, el diseño de redes de comunicación, entre muchas otras. Además, existen otros problemas de viajeros "derivados" de TSP como: problema del viajero comprador, TSP generalizado, problema de ruta de vehículos, problema de estrella del anillo, entre otros. Cada uno de estos problemas, tiene otra gama de posibles aplicaciones. Por lo que, los problemas tipo TSP, resultan ser muy importantes.

1.1. Definición matemática

Sean d_{ij} la distancia entre las ciudades i y j y x_{ij} una variable binaria que indica si la ruta incluye un viaje de i a j y sea sea u_i una variable entera que guarda el numero de ciudades visitada hasta la ciudad i. La formulación del problema como un problema de programación lineal entera sería:

$$
\begin{aligned}
\text{Minimizar:} \quad & \sum_{i=1}^{n}\sum_{j=1}^{n} d_{ij}x_{ij}, \\
\text{Sujeto a:} \quad & \sum_{j=1}^{n} x_{ij} = 1, \quad \forall i, \\
& \sum_{i=1}^{n} x_{ij} = 1, \quad \forall j, \\
& u_i - u_j + 1 \leq (n-1)(1-x_{ij}), \quad \forall i, j = \{2,3,...,n\}, i \neq j \\
& x_{ij} \in \{0,1\}, \quad u_i \in \mathbb{Z}.
\end{aligned}
$$

La primera restricción nos indica que el viajero debe pasar por todas las ciudades del problema una vez; la segunda, nos indica que el viajero debe salir de las ciudades que visita unz vez; por último, la

tercera restricción nos indica que nuestra ruta sea cerrada y no se generen subciclos en la trayectoria del viajero, nótese que, para un solo ciclo, las restricciones siempre se cumpliran dado que si $x_{ij} = 0$, el lado derecho será $(n-1)$ lo cual es la diferencia máxima entre cuando visitamos 2 ciudades. Si $x_{ij} = 1$, implicaría que en la ruta vas de i a j, por lo que, $u_i + 1 = u_j$, así, $u_i - u_j + 1 = 0 = (n-1)(1 - x_{ij})$ cumpliendo esta última restricción.

El TSP es notoriamente difícil de resolver, al menos de manera exacta, debido a su naturaleza combinatoria. Es un problema NP-difícil, lo que significa que no se conoce un algoritmo eficiente y único que pueda resolver todas sus instancias en un tiempo polinómico. A medida que el número de ciudades aumenta, la cantidad de posibles soluciones crece factorialmente.

Actualmente, una de las únicas maneras de resolver un problema tipo TSP de manera exacta es encontrar y comparar todas las permutaciones de posibles rutas entre las ciudades, la cual es de el orden computacional de $O(n!)$, haciendo que la búsqueda exhaustiva sea imposible para instancias grandes o hasta medianas.

2. Metodología e implementación

Implementamos y comparamos cuatro heurísticas principales para resolver el TSP vecino más cercano, nearest insertion, 2-opt y búsqueda tabú (aunque en nuestro código hay una quinta que es recocido simulado, solo la dejamos en nuestro código para compararla con las otras, pero no se va a entrar en detalle sobre esta heurística en este reporte), buscaremos la ruta de menor distancia para visitar todas las ciudades en Qatar. Nuestros datos son en total 194 ciudades, es decir, sobre todo en comparación a otros países, es un problema con pocos datos. Por lo que, estos algoritmos, a pesar de no ser los más robustos, darán buenas aproximaciones a este problema particular.

El objetivo de este proyecto es comparar los 4 métodos, evaluando las aproximaciones al problema, considerando el tiempo que tomó en llegar a ellas y concluir sobre las ventajas y desventajas de cada uno de ellos. Además, buscaremos encontrar la mejor aproximación a la solución real del problema.

2.1. Vecino más cercano

La heurística del vecino más cercano es una de las más simples y conocidas para abordar problemas como el TSP en escenarios de baja complejidad. Su objetivo es construir una ruta comenzando en una ciudad inicial y seleccionando, en cada paso, la ciudad más cercana que aún no ha sido visitada. Aunque es rápida y sencilla, no garantiza soluciones óptimas.

A continuación se presenta el pseudocódigo de esta heuristica:

```
Inicializar:
    current_city = 0 (empezar desde la primera ciudad)
    route = [current_city]
    total_distance = 0
    visited = [False] * número_de_ciudades
    Marcar current_city como visitada

Mientras no se hayan visitado todas las ciudades:
    nearest_city = None
    min_distance = infinito
    Para cada ciudad no visitada:
        Si la distancia a la ciudad actual es menor que min_distance:
            Actualizar nearest_city y min_distance
```

```
    Agregar nearest_city a route
    Sumar min_distance a total_distance
    Marcar nearest_city como visitada
    current_city = nearest_city

Agregar la distancia para regresar a la ciudad inicial
Agregar la ciudad inicial al final de la ruta
Regresar route y total_distance
```

La complejidad del algoritmo es $O(n^2)$, ya que para cada ciudad, se realiza una búsqueda entre $n-1$ ciudades restantes para determinar el vecino más cercano.

Las ventajas y desventajas de este algoritmo son las siguientes:

Ventajas	**Desventajas**
Implementación sencilla.	Propenso a quedarse atrapado en soluciones subóptimas.
Baja complejidad computacional comparado con otras heurísticas.	No garantiza una solución cercana al óptimo.
Proporciona una solución inicial útil para otros métodos más avanzados.	Sensible al punto de partida.
Rápido en escenarios con pocas ciudades.	Limitado por su enfoque *greedy*, que no considera decisiones a futuro.

Cuadro 1: Ventajas y desventajas del algoritmo del vecino más cercano.

2.2. Nearest insertion

La heurística de *Nearest Insertion* se basa en añadir nuevos puntos (o ciudades) a un grupo ya construido y visitado. Usualmente es implementado empezando con dos ciudades (pueden ser la primera y la más cercana a la primera o solamente 2 al azar). Ahora, iterativamente, hasta que se visiten todas las ciudades, se busca la ciudad más cercana a la ruta, es decir, a alguna de las ciudades en la ruta. Este algoritmo busca minimizar el incremento en la distancia total al decidir en qué posición insertar la nueva ciudad. Aunque es más sofisticado que el vecino más cercano, sigue siendo una heurística subóptima.

A continuación se presenta el pseudocódigo de esta heuristica:

```
Inicializar:
    route = [0, 1, 0] (iniciar con dos ciudades y vuelta al inicio).
    visited = {0, 1} (marcar las dos primeras ciudades como visitadas).

Mientras no se hayan visitado todas las ciudades:
    nearest_city = None, min_distance = infinito.
    Para cada ciudad no visitada:
        Encontrar la más cercana a cualquier ciudad en route.
    best_position = None, min_increase = infinito.
    Para cada par consecutivo en route:
        Calcular el costo incremental de insertar nearest_city.
        Actualizar best_position si el costo es menor.
    Insertar nearest_city en route en best_position.
    Marcar nearest_city como visitada.
```

```
Calcular distancia total.
Retornar final_route y total_distance.
```

El tiempo computacional del algoritmo es $O(n^2)$ ya que se cumplen las siguientes caracteristicas:

- La búsqueda de la ciudad más cercana toma $O(n)$ por iteración.
- La evaluación del mejor punto de inserción también es $O(n)$.
- Se realizan $n-2$ iteraciones para las ciudades restantes.

Las ventajas y desventajas de este algoritmo son las siguientes:

Ventajas	**Desventajas**
Fácil de implementar, intuitivo y directo de programar.	Algoritmo subóptimo para soluciones globales, especialmente en instancias grandes.
Produce rutas generalmente mejores que las heurísticas más simples, como el vecino más cercano.	Sensible al punto de inicio, lo que puede afectar la calidad de la solución.
Permite expandir iterativamente la ruta, ideal en escenarios dinámicos donde se agregan nuevas ciudades.	Depende mucho de las distribuciones de las ciudades, funcionando peor en distribuciones complejas.
Útil como solución inicial para optimizaciones avanzadas.	Incrementa la distancia total en cada paso al no considerar el problema globalmente.

Cuadro 2: Ventajas y desventajas de la heurística *Nearest Insertion*.

2.3. 2-opt

La heurística de 2-Opt es un método de búsqueda local propuesto en 1958 por Croes, basado en la idea de que en una ruta, dos aristas que se cruzan son subóptimas. Al intercambiarlas, se puede obtener una ruta más corta. Esta heurística se utiliza principalmente para refinar soluciones iniciales mediante la eliminación de aristas y la reconexión de subrutas para reducir la distancia total. Es parte de la familia de heurísticas k-Opt, siendo $k = 2$ en este caso.

El pseudocódigo de esta heurística es el siguiente:

```
Inicializar:
    Crear una ruta inicial en orden natural: route = [0, 1, 2, ..., n-1]
    Calcular la distancia total de la ruta inicial: best_distance
    improved = True

Mientras improved sea True:
    improved = False
    Para cada par de indices (i, j) tal que 1 <= i < j < n:
        Generar una nueva ruta invirtiendo la seccion route[i:j+1]
        Calcular la distancia de la nueva ruta: new_distance
        Si new_distance < best_distance:
            Actualizar route y best_distance
            improved = True

Convertir los indices de la ruta a IDs de las ciudades.
Retornar route y best_distance.
```

La complejidad del algoritmo 2-Opt en el peor de los casos es de $O(n^3)$ y en el mejor de los casos $O(n^2)$ (siendo este caso un muy difícil de conseguir),debido a que:

- Se evalúan $O(n^2)$ combinaciones de pares (i, j).
- Cada evaluación implica calcular la distancia de una ruta en $O(n)$.

Las ventajas y desventajas de este algoritmo son las siguientes:

Ventajas	**Desventajas**
Mejora rutas iniciales al reducir distancias mediante el intercambio de segmentos.	No garantiza encontrar el óptimo global.
Puede ser combinado con otros métodos para refinar soluciones iniciales.	Peor tiempo computacional ($O(n^3)$) que heurísticas como vecino más cercano o *nearest insertion*.
Relativamente fácil de implementar e intuitivo.	Requiere una solución inicial razonable para ser efectivo.
Produce soluciones de buena calidad en tiempos razonables para problemas medianos.	Puede ser ineficiente para problemas muy grandes debido a su alta complejidad.

Cuadro 3: Ventajas y desventajas de la heurística 2-Opt.

2.4. Búsqueda tabú

La búsqueda tabú es una metaheurística que explora el espacio de soluciones permitiendo movimientos que empeoran temporalmente la solución, pero evita ciclos mediante una lista tabú que restringe ciertos movimientos recientes. Es una técnica efectiva para aproximar soluciones de alta calidad en problemas, como el TSP, aunque con una mayor demanda computacional que los métodos anteriores.

El pseudocódigo de esta heuristica se presenta a continuación:

```
Inicializar:
    current_solution = lista de ciudades en orden natural
    current_distance = calcular_distancia_total(current_solution)
    best_solution = current_solution
    best_distance = current_distance
    tabu_list = lista vacía con capacidad máxima tabu_tenure

Para cada iteración en el rango [0, max_iterations):
    Generar vecinos de current_solution usando intercambios 2-opt
    Evaluar la distancia de cada vecino
    Filtrar vecinos que están en tabu_list
    Seleccionar el mejor vecino no tabú
    Actualizar current_solution y current_distance con el mejor vecino
    Si current_distance < best_distance:
        Actualizar best_solution y best_distance
    Agregar el movimiento usado (índices del intercambio) a tabu_list

Convertir índices de la mejor solución a IDs de ciudades.
Retornar best_solution y best_distance.
```

A diferencia de las otras heurísticas este necesita ciertos parametros clave, para el correcto funcionamiento de este método los cuales se mencionan a continuación:

- **Número máximo de iteraciones (*max_iterations*)**: Determina cuántas iteraciones realiza el algoritmo. En nuestro caso, $max\_iterations = 300$ fue el valor que produjo los mejores resultados, ya que con más el tiempo de ejecución era demasiado elevado sin una mejora considerable.
- **Tamaño de la lista tabú (*tabu_tenure*)**: Define la cantidad de movimientos recientes que se restringen. $tabu\_tenure = 100$ fue el valor óptimo en nuestro caso.

La complejidad del algoritmo se calculo de la siguiente manera:

- Tamaño del vecindario $V = O(n^2)$.
- Cálculo de distancias = $O(n)$.
- Complejidad por iteración = $O(n^2)$.
- Complejidad total = $O(IteMax \cdot n^2)$, donde $IteMax$ es el número máximo de iteraciones.

Este algoritmo, como los demas vistos presentan ciertas ventajas y desventajas, las cuales algunas se mencionan en la siguiente tabla.

Ventajas	**Desventajas**
Evita ciclos y caer en óptimos locales mediante la lista tabú.	Alta demanda computacional con complejidad $O(IteMax \cdot n^3)$.
Aproxima la solución global mejor, dependiendo menos de la solución inicial.	Requiere ajustar parámetros como *tabu_tenure* para lograr un rendimiento óptimo.
Es flexible y adaptable a diferentes problemas mediante la configuración de parámetros.	Menos eficiente en tiempo para problemas grandes comparado con heurísticas más simples.
Selecciona la mejor solución no tabú, incluso si es peor que la actual, permitiendo explorar soluciones más amplias.	Implementación más compleja que métodos como vecino más cercano o *nearest insertion*.

Cuadro 4: Ventajas y desventajas de la búsqueda tabú.

3. Resultados

Los resultados obtenidos se resumen en la Tabla 5.

Heurística	**Distancia Total**	**Tiempo Erick (s)**	**Tiempo Juan Pablo (s)**
Vecino más cercano	11,640	0.0028	0.0059
Nearest Insertion	11,267	0.1256	0.2732
2-Opt	10,446	2.2285	4.7472
Búsqueda Tabú	9,691	162.1926	310.4544

Cuadro 5: Comparación de heurísticas para el TSP usando distintos equipos.

Los tiempos dependen del equipo que se está utilizando, ya que como se ve en la tabla estos cambian considerablemente, si se quiere replicar dejamos el link del repositorio de nuestro proyecto elaborado en python, junto con los resultados obtenidos en el Jupyter notebook que se encuentra en el repositorio (los tiempos que aparecen son los de Juan Pablo). GitHub

Como era de esperarse, Vecino Más Cercano (VMC) obtuvo el peor valor óptimo, 11,640, y entre las heurísticas más "simples" superado por Nearest Insertion (NI) con 11,267. Esto se debe principalmente a que VMC es más dependiente del punto de partida (el cual es el primer dato del archivo .txt), mientras que NI tiene un enfoque más equilibrado al construir la ruta.

Sobre las heurísticas más avanzadas: Búsqueda Tabú (BT) logró el mejor valor óptimo por bastante, bajando aún de los 10,000, llegando hasta 9,691, siendo el más cercano a la solución óptima que es 9,356 según en el ariculo de Menezes [3] con un error del 3,4 % . Una gran mejora sobre los métodos previos, aunque a costa de un tiempo considerablemente mayor (162.19 segundos en total en el equipo de Erick). Por otro lado, 2-opt mostró un balance interesante, con una solución razonablemente buena de 10,446 en un tiempo más reducido (2.22 segundos) sobre búsqueda tabú.

4. Conclusiones y observaciones

Si se tiene poco tiempo disponible, recomendamos utilizar Vecino Más Cercano o Nearest Insertion, ya que son rápidos y simples, aun dando soluciones que aproximan a la real. NI es preferible porque produce mejores soluciones que VMC, pero VMC es aún más rápido, por lo que, para problemas extremadamente grandes, donde la diferencia entre el tiempo en uno y otro es significativa, puede valer más la pena solamente encontrar la solución más rápida.

Para poco poder computacional, aún se puede utilizar, VMC o NI, pero, en general, recomendamos optar por 2-opt, que ofrece una buena solución en tiempos razonables sin requerir demasiados recursos. Usar búsqueda tabú con pocos recursos puede ser muy tardado.

En el mejor de los casos, cuando lo único que nos importa es la calidad del resultado, sin duda, hay que utilizar Búsqueda Tabú, ya que logra las soluciones más cercanas al óptimo global, aunque el costo en tiempo sea elevado.

Ahora, cuando tengamos muchos datos, consideramos que lo mejor es utilizar 2-opt, aunque, incluso este método puede no ser muy efectivo. Para estos casos, se puede considerar implementar otros algoritmos, como Recocido Simulado, Lin-Kernighan, Christofides o hasta versiones optimizadas de 2-opt (como 3-opt), ya que son más escalables que los métodos que tenemos aquí.

A pesar de las posibles mejoras que siempre existen en un proyecto, consideramos que logramos obtener una solución óptima muy buena para nuestro problema particular. Los resultados obtenidos muestran un análisis claro y detallado de cómo las diferentes heurísticas pueden abordar el problema del agente viajero, evaluando tanto la calidad de las soluciones como los tiempos de ejecución. Además, pudimos comparar los métodos seleccionados, entendiendo sus ventajas y desventajas en diferentes contextos. Por lo cual, esto nos proporciona una base sólida para futuras investigaciones y aplicaciones prácticas en escenarios reales.

5. Referencias

Referencias

[1] Davis, A. (2022, mayo 18). Traveling Salesman Problem With the 2-opt Algorithm. *Medium*. Recuperado de https://slowandsteadybrain.medium.com/traveling-salesman-problem-ce78187cf1f3

[2] Kuo, M. (2020, enero 2). Algorithms for the travelling salesman problem. *Routific.com*. Recuperado de https://www.routific.com/blog/travelling-salesman-problem

[3] Menezes, R. (s.f.). TSP Techniques. *UPENN* https://www.cis.upenn.edu/~cis1890/files/Lecture11.pdf

[4] ¿Qué es el Travelling Salesman Problem (TSP) y cómo solucionarlo? (s/f). *Simpliroute*. Recuperado el 13 de diciembre de 2024, de https://simpliroute.com/es/blog/que-es-el-travelling-salesman-problem-tsp-y-como-solucionarlo

[5] Weru, L. (2019, diciembre 28). 11 animated algorithms for the traveling salesman problem. *STEM Lounge*. Recuperado de https://stemlounge.com/animated-algorithms-for-the-traveling-salesman-problem/

[6] Wikipedia contributors. (2024a, diciembre 5). Travelling salesman problem. *Wikipedia, The Free Encyclopedia*. Recuperado de https://en.wikipedia.org/w/index.php?title=Travelling_salesman_problem&oldid=1261381073

[7] Wikipedia contributors. (2024b, diciembre 9). Nearest neighbour algorithm. *Wikipedia, The Free Encyclopedia*. Recuperado de https://en.wikipedia.org/w/index.php?title=Nearest_neighbour_algorithm&oldid=1262140707

[8] (S/f). Recuperado el 13 de diciembre de 2024, de *Gatech.edu*. https://www2.isye.gatech.edu/~mgoetsch/cali/VEHICLE/TSP/TSP009__.HTM