Implementación y Optimización del método heurístico Hill Climbing para resolver el Traveling Salesman Problem (TSP)

Equipo 5: José Luis Zárate Cortés, Carlos Román López Sierra y Miguel López Cruz

1/23/2021

Introducción

Existe un problema importante conocido como el Traveling Salesman Problem (TSP), el cual plantea, de manera inforaml, la siguiente interrogante:

"¿Cuál es la ruta mínima posible que parte de una ciudad inicial arbitraria X_0, que pasa por todos las ciudades del conjunto una y sólamente una vez, y retorna a la ciudad de origen X_0?"

Descripción de TSP

Se tiene un número de nodos (ciudades, localidades, tiendas, empresas, etc.) que deben ser visitados por una entidad (persona, agente viajero, automotor, avión, autobús, etc.), sin visitar 2 veces el mismo nodo. Si tenemos 3 nodos (a, b y c) por visitar, entonces tendríamos una función de combinaciones sin repetición c(3,2), es decir, tendríamos 6 posibles soluciones: abc, acb, bac, bca, cab, cba, para el caso de 4 nodos tendríamos 12 combinaciones, para 10 nodos tendríamos 90 combinaciones, para 100 ciudades tendríamos 9,900 combinaciones y así sucesivamente. Como ejemplo en el problema del Ulises de Homero que intenta visitar las ciudades descritas en la Odisea exactamente una vez (16 ciudades) donde existen múltiples conexiones entre las diferentes ciudades, Grötschel y Padberg (1993) llegó a la conclusión de que existen 653,837'184,000 rutas distintas para la solución de este problema.

Origen

La primera solución reportada para resolver el problema del Agente Viajero fue en 1954, cuando George Dantzig, Ray Fulkerson, y Selmer Johnson publicaron la descripción de un método de solución del PAV (Problema del Agente Viaje o sus siglas en inglés TSP – Travel Sailsman Problem) titulado "Solutions of a large scale traveling salesman problem" (Soluciones de gran escala para el problema del agente viajero) para resolver una instancia de 49 ciudades donde un agente viajero desea visitar un conjunto de ciudades, asignándoles un costo por visitar ciudades contiguas (distancia de traslado entre dos ciudades). Para esta solución se propusieron 2 condiciones: regresar a la misma ciudad de la cual partió y no repetir ciudades con el objetivo de encontrar una ruta o un camino con el menor costo posible.

Solución “trivial” e ingénua

La solución más directa puede ser, intentar todas las permutaciones (combinaciones ordenadas) y ver cuál de estas es la menor (usando una búsqueda de fuerza bruta). El tiempo de ejecución es un factor polinómico de orden $O(n!)$, el factorial del número de ciudades, esta solución es impráctica para dado solamente 20 ciudades.

Otras aproximaciones

Otras aproximaciones incluyen:

Varios algoritmos de ramificación y acotación, los cuales pueden ser usados para procesar TSP que contienen entre 40 y 60 ciudades.

Solución de un TSP con 7 ciudades usando un simple algoritmo de ramificación y acotación. Nota: el número de permutaciones es mucho menor que el de la búsqueda fuerza bruta.

Algoritmos de mejoras progresivas (iterativas) los cuales utilizan técnicas de programación lineal. Trabajan bien para más de 200 ciudades.

Implementaciones de ramificación y acotación y un problema específico de generación de cortes (ramificación y poda); este es el método elegido para resolver grandes instancias. Esta aproximación retiene el récord vigente, resolviendo una instancia con 85,900 ciudades, vea Applegate et al. (2006).

Aplicación del TSP

TSP se puede emplear en cualquier situación que requiere seleccionar nodos en cierto orden que reduzca los costos:

Reparto de productos. Mejorar una ruta de entrega para seguir la más corta.

Transporte. Mejorar el recorrido de caminos buscando la menor longitud (figura 1).

Robótica. Resolver problemas de fabricación para minimizar el número de desplazamientos al realizar una serie de perforaciones en un circuito impreso.

Turismo y agencias de viajes. Aun cuando los agentes de viajes no tienen un conocimiento explícito del Problema del Agente Viajero, las compañías dedicadas a este giro utilizan un software que hace todo el trabajo. Estos paquetes son capaces de resolver instancias pequeñas del TSP.

Aplicación del TSP

Horarios de transportes laborales y/o escolares. Estandarizar los horarios de los transportes es claramente una de sus aplicaciones, tanto que existen empresas que se especializan en ayudar a las escuelas a programarlos para optimizarlos en base a una solución del TSP.

Inspecciones a sitios remotos. Ordenar los lugares que deberá visitar un inspector en el menor tiempo.

Secuencias. Se refiere al orden en el cual n trabajos tienen que ser procesados de tal forma que se minimice el costo total de producción.

Hill Climbing

El algoritmo de Hill Climbing realiza un seguimiento de un estado actual y en cada iteración se mueve al estado vecino con el valor más alto, es decir, se dirige en la dirección que proporciona el ascenso más "empinado". Termina cuando alcanza un "pico" donde ningún vecino tiene un valor más alto. Hill Climbing no mira hacia el futuro más allá de los vecinos inmediatos del estado actual. Esto se asemeja a tratar de encontrar la cima del Monte Everest en una espesa niebla mientras sufres de amnesia.

Por otro lado, otra forma de utilizar Hill Climbing consiste en utilizar el negativo de una función de coste heurística como función objetivo; que ascenderá localmente al estado con menor distancia heurística a la meta.

Implementación

Nuestro método se compone de cinco funciones incluyendo `best_solution` que efectúa el algoritmo *Hill climbing*. A continuación describimos las funciones que componen nuestro código:

- `distance_matrix`: Esta función calcula la matriz de distancias de los puntos que componen el *dataset* de las ciudades a visitar. Esta genera una matriz simétrica de tamaño nxn.
- `random_solution`: Genera soluciones aleatorias para el *dataset* propuesto.Esta función retorna una lista con el orden en el que se visitaran las ciudades.
- `calculate_distance`: Calcula la distancia euclidiana que se recorre dada una solución propuesta. La salida de esta función es un número real correspondiente a la distancia recorrida en dicha solución propuesta.

Implementación

- `neighbors`: *Hill climbing* funciona en parte generando todas las soluciones vecinas a la solución actual. Una solución vecina es una solución que es solo ligeramente diferente de la solución actual, esto cuidando que cada ciudad se visite una sola vez. Crear soluciones ligeramente diferentes lo logramos intercambiando ciudades de lugar a través de ciclos `for`. Después con un conjunto de este tipo de soluciones, el algoritmo busca la que recorre la menor distancia, encontrado así el vecino más óptimo.

- `best_solution`: Primero, crea una solución aleatoria y calcula la longitud de su ruta. Después a partir de esta soluciones crea el conjunto de soluciones vecinas y con ello encuentra la mejor. A partir de ahí, siempre que el mejor vecino sea mejor que la solución actual se repite el mismo patrón con la solución actual cada vez que se actualiza con el mejor vecino. Para no quedar estancado en mínimos locales implementamos un random-restart, generando un nueva ruta aleatoria. Si esta nueva solución es mejor que la

Implemetación

El dataset elegido para resolver TSP se extrajo de la pagina National Traveling Salesman Problem de concord, el cual contiene un total de 4,663 ciudades canadienses cada una con su latitud y longitud.

Resultados

Ejemplo 20 ciudades

[28]:
```
tsp_cities_dos = dat1[0:20,:]
```

[30]:
```
%%time
parallel_hc(tsp_cities_dos, 0, 1e-1, 100)
```

```
CPU times: user 4.67 ms, sys: 36 ms, total: 40.7 ms
Wall time: 9.82 s
```

[30]:
```
(3329.1623916962017,
 [0, 1, 5, 6, 12, 13, 11, 15, 17, 18, 19, 16, 14, 7, 2, 4, 9, 10, 8, 3, 0],
 8.481815576553345)
```

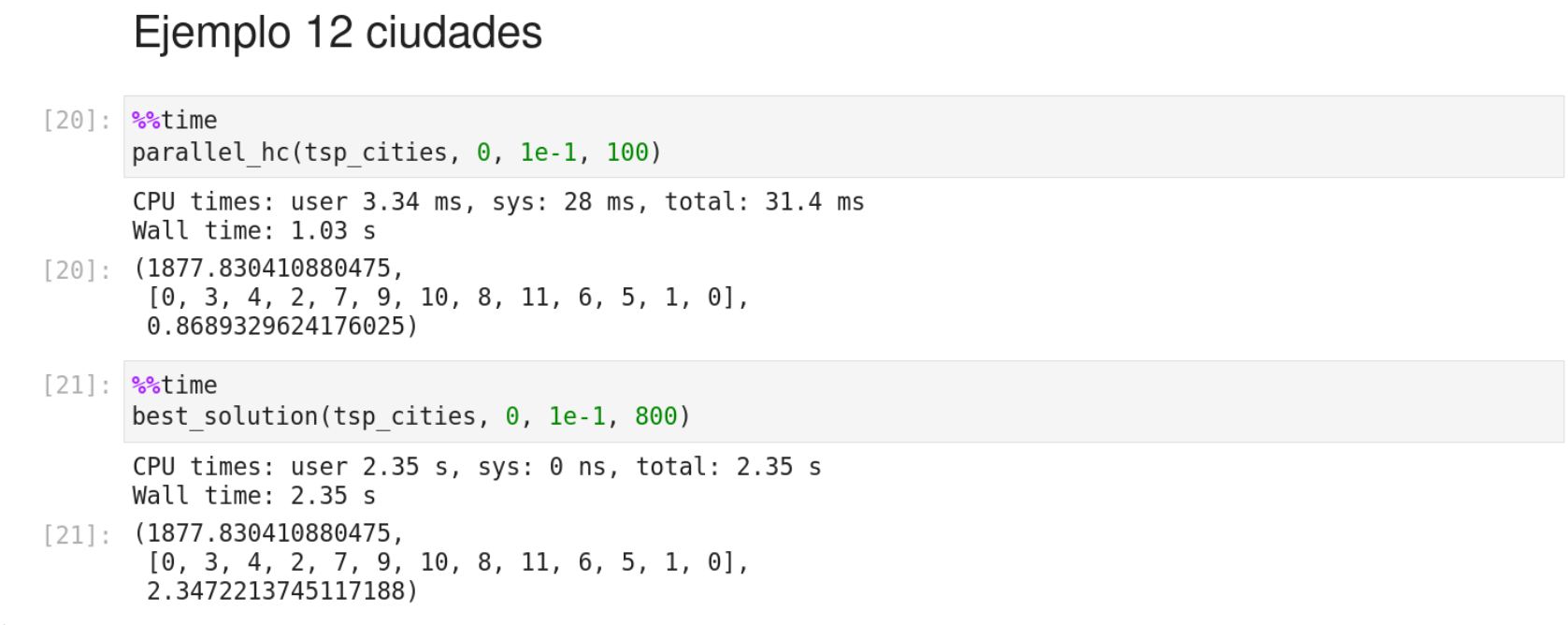
[32]:
```
%%time
best_solution(tsp_cities_dos, 0, 1e-1, 800)
```

```
CPU times: user 20.8 s, sys: 3.99 ms, total: 20.8 s
Wall time: 20.8 s
```

[32]:
```
(3329.162391696201,
 [0, 3, 8, 10, 9, 4, 2, 7, 14, 16, 19, 18, 17, 15, 11, 13, 12, 6, 5, 1, 0],
 20.783585786819458)
```

Figure 1: 20 ciudades

Resultados



Ejemplo 12 ciudades

```
[20]: %%time
      parallel_hc(tsp_cities, 0, 1e-1, 100)

      CPU times: user 3.34 ms, sys: 28 ms, total: 31.4 ms
      Wall time: 1.03 s
[20]: (1877.830410880475,
       [0, 3, 4, 2, 7, 9, 10, 8, 11, 6, 5, 1, 0],
       0.8689329624176025)

[21]: %%time
      best_solution(tsp_cities, 0, 1e-1, 800)

      CPU times: user 2.35 s, sys: 0 ns, total: 2.35 s
      Wall time: 2.35 s
[21]: (1877.830410880475,
       [0, 3, 4, 2, 7, 9, 10, 8, 11, 6, 5, 1, 0],
       2.3472213745117188)
```

Figure 2: 12 ciudades

Resultados

```
[13]: tsp_sol = tsp.main(tsp_cities)

Objective: 3316
Route:
 0 -> 3 -> 8 -> 10 -> 4 -> 2 -> 9 -> 7 -> 14 -> 16 -> 15 -> 11 -> 13 -> 12 -> 6 -> 5 -> 1 ->
0


[14]: hc_sol = parallel_hc(tsp_cities)

[15]: hc_sol

[15]: (3266.456452336076,
 [0, 1, 5, 6, 12, 13, 11, 15, 8, 10, 9, 16, 14, 7, 2, 4, 3, 0],
 4.648738622665405)
```

Figure 3: 17 ciudades

Conclusiones

Uno de los más aspectos importantes a mencionar del problema TSP que buscamos resolver con nuestra implementación del método de Hill Climbing, es que el espacio de posibles soluciones tiene un crecimiento factorial, esto es, para un conjunto de n ciudades se tiene un total de $n!$ rutas posibles. Esto hace que no sea factible explorar todo el espacio de soluciones y sea necesario utilizar métodos que lo hagan de forma aleatoria pero suficientemente amplia, como lo hace Hill Climbing.
Invariablemente, conforme crece el número de ciudades, es necesario que crezca también el espacio a explorar para que se garantice que la mejor ruta es suficientemente cercana al mínimo global.

¡Gracias por su atención!