TP 2 Optimización

Sebastián Cherny

3 de Julio de 2019

1 Consigna 1

El problema del Travelling Salesman Problem (TSP) puede pensarse como un grafo completo (aristas entre todo par de nodos distintos) para el cual hay que encontrar un ciclo que pase exactamente una vez por cada vértice (hamiltoniano) exceptuando el primero; pero no sólo esto que ya es suficientemente difícil, sino el de menor longitud.

Dada su dificultad, en este trabajo se propone un algoritmo para encontrar un camino 'bastante' bueno, basado en algoritmos estadísticos que se acercan a buenas soluciones.

1.1 Observación - cota mínima (y máxima)

Algo que podemos ver, es que si tenemos un árbol (grafo sin ciclos que conecta a todos los nodos), luego la suma de las aristas del árbol dan una cota para un ciclo: Utilizando sólo aristas de ese árbol (suponiendo que podemos repetir aristas), no podemos obtener un camino de longitud menor, ya que tendremos que recorrer por lo menos una vez cada aristas para recorrer todos los nodos. Entonces, si encontramos el árbol (o uno de los) que minimice esta suma, tendremos una cota para el mínimo camino que recorre todos los nodos. Y para nuestro ciclo, la cota la podremos pensar como esa suma, más la arista de menor longitud dada (ya que si tuviéramos mucha suerte, podríamos cerrar el ciclo con esta arista pequeña que no está en el árbol).

Es decir, un buen escenario sería que el árbol que minimiza la suma de las distancias (a partir de ahora le llamaré MST por sus siglas en inglés, Minimum Spanning Tree) sea directamente un camino, y que haya una arista pequeña entre el primer y el último nodo que no está en el árbol.

Además, de este razonamiento podemos pensar que si tenemos un árbol, seguro que podremos encontrar un camino cuya longitud sea el doble de la suma de sus aristas. Esto es porque, pensando en el algoritmo DFS para recorrer todos los nodos, podemos desde el primer nodo, movernos por un caminito sin volver hacia atrás hasta llegar al final del camino, luego empezar a volver hasta que haya un nodo con vecinos no visitados, de vuelta ir hasta el final del camino,

y así sucesivamente hasta recorrer todos los nodos, y finalmente desde el último volver al primero. Y se puede ver que esto recorre cada arista a lo sumo 2 veces. Este razonamiento en realidad tiene un detalle que es que no podemos recorrer las aristas más de una vez porque estaríamos pasando por un nodo dos veces. Pero si suponemos que las distancias cumplen la desigualdad triangular (ir directo de un nodo a otro es más corto o igual que ir a través de otros), mirando el camino que nos quedó volviendo en las aristas y repitiendo nodos, si sacamos los repetidos y conservamos el orden de las primeras apariciones de los nodos, el camino será a lo sumo igual al obtenido repitiendo aristas. Entonces la cota sigue valiendo.

Entonces, si nuestro algoritmo da un camino de longitud mayor al doble de la suma del MST, sabemos que es un mal recorrido. Y si está 'cerca' de la suma del MST, sabemos que estaremos cerca del óptimo (siempre sin poder verificarlo).

Además, si el MST fuera por ejemplo un camino y tuviéramos una de las aritas de menor longitud afuera del camino, listo, habríamos encontrado el mejor camino de manera exacta y también súper rápida. Y si el MST formara un camino salvo por una ciudad que se desprende de algún lugar no extremo del camino, podríamos pensar que la solución debe darse metiendo a esa ciudad extra en algún lugar del camino.

En conclusión, el MST podría darnos información de qué ciudades pueden ir consecutivas y se podría implementar una función que separe al MST en caminos lo más largos posibles, e intente ordenar estos caminos. Por más que no asegure una solución exacta probablemente sea información útil, al menos más útil que empezar probando cualquier orden al azar.

1.2 Solución exacta para pocas ciudades

Veamos una solución cuyo algoritmo tiene orden $O(2^n * n^3)$:

La idea es utilizar programación dinámica. Supongamos que tenemos el mejor camino empezando por una todas las ciudades, terminando por todas las ciudades y para todo subconjunto de ciudades en el medio de tamaño c. Entonces, qué pasa si ahora quiero saber el mejor camino empezando empezando por A, terminando por Z y dejando en el medio un subconjunto de tamaño $c+1$? Lo que tengo que hacer es probar dejar como penúltima ciudad a alguna de este subconjunto (porque alguna tiene que ir en ese lugar), y luego entonces, teniendo fijadas la primera (A) y la penúltima (llamémosle X), tengo entre estas dos ciudades un subconjunto de tamaño c, por lo que por nuestra hipótesis ya sé cuál es el mejor camino entre A y X con el subconjunto en el medio que me quedó de tamaño c. Entonces como en el recorrido teniendo estas tres ciudades fijadas no puedo evitar el recorrido de X a Z y de Z a A para la vuelta, lo único que podré alterar es el camino del medio, que fijando A y X ya tenemos la solución óptima.

De esta manera, vamos a tener la memorización de la dinámica guardando el

mejor camino empezando desde cada ciudad, teniendo en el medio cada subconjunto, y terminando en cada ciudad no usada antes, y como hay 2^n subconjuntos en total, el orden será de $O(2^n * n^2)$, pero para ir de una solución ya calculada a una no calculada deberemos fijarla penúltima ciudad, quedando así la complejidad temporal mencionada al principio.

Ahora, otra complicación de esta solución es la memoria, ya que para cada una de estos $O(2^n * n^2)$ lugares en memoria deberíamos guardar el camino exacto para al final poder reproducirlo. Esto se puede evitar guardando simplemente los valores de las longitudes, y luego al obtener el mejor camino y sabiendo cuáles son las ciudades por las que debo empezar y terminar, busco cuál debe ser la penúltima para que se cumpla que la suma del mejor camino desde la primera a la penúltima más la distancia desde la penúltima a la última y de la última a la primera, den como resultado la mejor longitud obtenida. Y así sucesivamente se puede reconstruir el camino sin necesidad de guardar más que las longitudes en memoria.

Para nuestro caso, se podría usar esta idea dividiendo primero a las 30 ciudades en dos grupos de 15, encontrando los mejores caminos de la primera mitad y la segunda mitad, y luego encontrando la mejor manera de unir estos dos grupos de ciudades. Lo que pasa es que la manera de dividir a las ciudades afecta mucho el resultado ya que se está forzando a muchos pares de ciudades a ir siempre separadas. Se podría iterar sobre los grupos de ciudades de la división, pero esto queda para otra posible implementación.

1.3 Algoritmo

Para encontrar una solución, usaré y combinaré distintos algoritmos:

1. El primero lo que hará será, dado un camino (que empezará siendo uno al azar), mirar todos los posibles caminos que se obtienen de intercambiar de lugar a dos ciudades en nuestro camino incial, y para cada uno de esos caminos posibles, quedarse con el que produzca menor longitud total. El problema de este algoritmo es que puede caer en mínimos locales, es decir un camino a partir del cual cambiando dos ciudades no se puede mejorar la longitud, pero que quizás cambiando 3 sí. Para arreglar esto, introduciremos una variación al estar en un mínimo local: Se generará un camino parecido al que tenemos (ya que por ser mínimo local suponemos que es más o menos bueno y no queremos perder la información), pero que agarre una posición al azar e invierta las primeras ciudades hasta esta posición.

2. Cada uno de los mínimos obtenidos con el algoritmo anterior pasará por un algoritmo que lo que hace es mirar para cada par de posiciones, qué pasa si invertimos las ciudades entre esas dos posiciones. Si con alguno de estos cambios obtenemos un camino mejor, nos quedamos con el cambio

que produzca el mejor camino y volvemos a mirar qué pasa si invertimos subarreglos de ciudades nuevamente. Así hasta que ningún cambio de estos produzca un camino mejor o se llegue a una cierta cantidad de iteraciones. Este algoritmo está inspirado en la idea del algoritmo de Lin-Kernighan según lo que entendí.

3. Por último, los caminos obtenidos con los algoritmos anteriores se meten en una lista que será nuestra población, e inspirado en el algoritmo genético, a partir de esa población itera varias veces haciendo lo siguiente: Se queda con los mejores caminos (descarta la mitad), para cada camino, obtiene otro como su 'descendiente' que lo obtiene invirtiendo el orden de algunas de las primeras ciudades. Luego para cada descendiente, según una función de probabilidad lo deja como está o le cambia de lugar a dos ciudades. Y luego agrega a estos descendientes a la población para volver a iterar descartando la mitad.

4. Luego de ejecutar el algoritmo genético, según si se quiere una corrida rápida o no (ver explicación de corrida) se vuelve a ejecutar el primer algoritmo combinado con el segundo.

1.4 Respuesta obtenida

El mejor camino obtenido fue el siguiente (con seed 10 por ejemplo en la corrida rápida y la 'ni rápida ni lenta'): 19, 22, 29, 1, 6, 20, 16, 2, 17, 21, 0, 18, 7, 8, 9, 14, 4, 23, 5, 3, 24, 11, 10, 13, 12, 26, 27, 25, 15, 28
Cuya longitud es 6146.64584622 .

Se puede ver que corriendo el archivo con –amount-of-seeds=10 que el promedio de corrida de este algoritmo 'ni rápido ni lento' da una solución en un promedio de 2 minutos y medio que en todos estos casos es la solución óptima encontrada incluso en corridas con más iteraciones (la mencionada arriba). La salida al final imprime:
Tardando un promedio de 150.23 (total 1502.35), el promedio de los caminos es de 6146.65. El mejor camino fue de 6146.64584622 y el peor de 6146.64584622

1.5 Cómo correr el algoritmo

El algoritmo tiene distintos parámetros:

1. –**rapido** si se quiere una corrida con pocas iteraciones y bien rápidas (si necesitáramos saber ahora qué recorrido conviene para empezarlo lo más temprano posible).

2. –**lento** si se quiere encontrar una muy buena solución realizando muchas iteraciones.

3. –**seed** para pasar el número de seed y poder reproducir la misma corrida en otro momento.

4. –**amount-of-seeds** si se quiere ver la solución del algoritmo con varios seeds (itera desde el 0 hasta la cantidad deseada).

5. –**archivo** para pasarle la dirección del archivo de la matriz de distancias, 'distancias.npy' o cualquier otro que tenga el formato que pueda ser interpretado por numpy.

Para fines prácticos la versión 'ni rápida ni lenta' parece ser la más útil, ya que podemos permitirnos unos minutos de espera con tal de mejorar ampliamente la longitud del camino (salvo que vayamos a una velocidad muy rápida, en donde poca diferencia de distancia no afecta mucho el tiempo del recorrido y nos podría afectar más la espera de 2 minutos).

1.6 Investigación posterior

Después de obtener un resultado que según las cotas del MST parecía bueno, investigué un poco y encontré un solucionador del problema TSP, llamado localsolver (https://www.localsolver.com/) que para pocas ciudades aparentemente lo resuelve de manera exacta usando un poco de programación lineal, en poco tiempo. Pedí una licencia que me dieron para probar su código durante un mes, y ejecutándolo para la matriz de distancias dada obtiene el mismo camino que mi algoritmo (en realidad da el camino comenzando por el 0, siguiendo por el 18, 7, y así, creando el mismo ciclo que mi respuesta empezando desde otro lugar).

2 Consigna 2

En esta consigna simplemente había que encontrar puntos en donde las funciones dadas (segunda de Schaffer y Holder table) evaluaran a valores lo más chicos posibles.

Para esta consigna se utilizó el algoritmo de simmulated annealing (SA), que arrancando en un punto al azar se mueve a un vecino y si el valor de la función se achicó, o si una función de probabilidad nos lo indica, el punto que se considera pasa a ser el vecino.

Con el algoritmo como el visto en clase con los parámetros estándares, la solución no era lo suficientemene buena (acá podemos saberlo porque tenemos los valores óptimos). De todas maneras, al combinarse el algoritmo con la técnica de 'random restart' (probar varios puntos iniciales), se mejoró exponencialmente el resultado, con el costo de aumentar el tiempo. Sin embargo, el algoritmo tarda poco en correr (10 segundos para ambas funciones probando 2000 puntos iniciales al azar para cada función), por lo que podemos pensar que la solución es eficiente y cumple los estándares de tiempo. Si se quisiera algo más rápido se podría jugar quizás con los parámetros o aplicar alguna otra técnica además de SA.

2.1 Resultados

Corriendo el algoritmo con el seed 0 ('python consigna2.py 0' en Linux, o simplemente con un '0' en los argumentos), el punto obtenido para la función de Holder table es el (8.03736161, 9.64460137), donde la función evalúa a -19.2013371227.

Y con el mismo seed, para la función segunda de Schaffer se obtiene el punto (0.02797762, -0.02685234), donde la función evalúa a 4.93750512698e-09 .

Sabiendo que los menores valores son -19.2085 y 0 respectivamente, vemos que las diferencias entre los valores obtenidos y los óptimos son muy pequeñas (el hecho de que un valor sea 0 nos impide hablar de porcentajes).