UNIVERSIDAD SIMÓN BOLÍVAR
Dpto. de Computación y Tecnología de la Información
CI-2693 - Lab. de Algoritmos y Estructuras de Datos III

Proyecto II

Un algoritmo heurístico para resolver el Problema del Cartero Rural

Autor:
Christopher Gómez (18-10892)
Ka Fung (18-10492)

Profesor:
Guillermo Palma

17 de enero de 2022

1. Introducción

El siguiente estudio experimental consiste en implementar un algoritmo heurístico para obtener soluciones aproximadas al Problema del Cartero Rural (RPP), modelado mediante el uso de grafos no dirigidos. Para ello, se empleará el algoritmo presentado por Pearn y Wu, el cual es una modificación del algoritmo de Christofides et al. El RPP consiste en encontrar un ciclo de costo mínimo en un grafo no dirigido, tal que atraviese un conjunto de lados requeridos al menos una vez.

En el algoritmo de Pearn y Wu requiere de la implementación de un algoritmo de apareamiento perfecto de costo mínimo. Por ello, se implementará dos algoritmos heurísticos para obtener un apareamiento perfecto de un grafo completo (un algoritmo ávido y el Vertex-Scan presentado por David Avis [1]), que proporcionan una solución aproximada y que sirven como una alternativa mucho más eficiente y menos compleja que el algoritmo de Edmonds, el cual proporciona una solución exacta al problema.

Luego, el objetivo de este estudio consiste en comparar la eficacia y eficiencia de estos dos algoritmos aplicados a solucionar 36 distintas del problema RPP.

2. Diseño de la solución

Para resolver el *problema del cartero rural* se usó el algoritmo constructivo proveído por el profesor del curso, basado en el presentado por Pearn y Wu en [2].

La idea principal de dicho algoritmo consiste en modelar la instancia del problema con un grafo no dirigido $G = (V, E)$ y un subconjunto $R \subseteq E$ que representa las aristas por las que el cartero debe pasar en el ciclo buscado. Luego, construir a partir del conjunto R un grafo $G' = (V_R, R)$, al cual se le añaden lados y vértices en varias ocasiones hasta obtener finalmente un grafo par y conexo del cual obtener un ciclo euleriano que servirá como solución aproximada del problema.

Los procedimientos para añadir lados al grafo G' en dos ocasiones se pueden resumir y simplificar como los siguientes:

- Se construye un grafo G_t completo donde cada vértice es una componente conexa de G' y el costo de cada lado corresponde al del camino de costo mínimo entre dichas componentes en G.
- Se obtiene el conjunto E_{MST} de los lados del arbol mínimo cobertor de G_t.
- Se construye E_t como el conjunto de lados de los caminos de costo mínimo asociados a cada lado de E_{MST}.
- Se agrega al conjunto de vértices de G' los vértices que aparecen E_t, y luego los lados admitiendo duplicados.

En este punto se ha convertido G' en un grafo conexo, pues hemos añadido a cada componente conexa del grafo G' inicial los caminos más cortos que las unen, por lo que solo resta hacer que G' sea par para poder obtener de él un ciclo euleriano:

- Se obtiene el conjunto V_0 de los vértices de grado impar de G' y se construye de él un grafo completo G_0, donde el costo de cado lado corresponde al del camino de costo mínimo entre tales vértices en G.

- Se determina M como el conjunto de lados que conforman un **apareamiento perfecto** de G_0
- Para cada lado de M, se obtiene el camino de costo mínimo asociado al mismo en G, y se añaden a G' sus vértices y luego sus lados, admitiendo nuevamente los duplicados para estos últimos.

Se tiene que finalmente G' es conexo y par, debido a que este último procedimiento, al tratarse de los caminos obtenidos de un apareamiento perfecto, añadirá un lado a los vértices inicial y final, inicialmente de grado impar, y dos lados a todos los demás vértices del camino, manteniéndolos pares.

Ya teniendo un grafo G' conexo y par, se puede obtener de él un ciclo euleriano que será la solución aproximada al problema modelado por G.

Cabe destacar que el algoritmo propuesto busca obtener un **apareamiento perfecto de costo mínimo** para hacer el grafo G' par, sin embargo, para simplificar la implementación se usaron algoritmos que obtienen un apareamiento perfecto aproximado al de costo mínimo. Además, es importante mencionar que si luego de añadir los primeros lados a G' el grafo resultante es par, se obtiene directamente el ciclo euleriano, y análogamente si al comienzo del algoritmo G' era conexo y par.

3. Detalles de la implementacion

La implementación de todos los algoritmos se realizó en el lenguaje de programación `Kotlin`, mediante el uso y la modificación de la librería `grafoLib`, construida a lo largo del curso.

Uno de los primeros aspectos a tomar en cuenta consistió en modificar las implementaciones de la clase `GrafoNoDirigido` para permitir tener lados duplicados, pues es un requerimiento del algoritmo en dos ocasiones.

Luego, dado que los grafos de la librería `grafoLib` tienen un número fijo n de vértices en el intervalo $[0..n)$, surgió el problema de cómo crear el grafo inicial G', teniendo que sus vértices consistirán en los vértices que aparecen en R, que a pesar de ser un subconjunto de V, puede estar conformado, por ejemplo, por los dos primeros y los dos últimos vértices de G, generando un grafo de a lo sumo 4 vértices en el intervalo $[0..n)$, imposible en `grafoLib` si $n > 4$.

Para resolver este problema, en la implementación se construye una función biyectiva $f : V \to V_R$ que es usada para modelar G' mediante un grafo isomorfo que permita relacionar sus vértices con los vértices correspondientes de G cuando sea necesario, como también relacionar G con el grafo isomorfo de G' por medio de f^{-1}. Este detalle no afecta gravemente la eficiencia de la implementación debido a que el modelado de f y f^{-1} se basa en arreglos dinámicos y diccionarios como tablas de hash, por lo que cada mapeo que se hace con estas funciones ocurre en un tiempo amortizado constante,

Análogamente, se presentó el mismo problema más adelante con el grafo G_0 y el conjunto de vértices de grado impar, solucionado de forma similar modelando G_0 mediante un grafo isomorfo inducido por la función $h : V_R \to V_0$ que permite relacionar sus vértices con los vértices del isomorfismo de G'. En el código fuente solo se construye h^{-1} por ser la única necesaria.

En seguida, como el algoritmo requiere de la búsqueda de caminos de costo mínimo (al hallar caminos entre pares de componentes conexas y de pares de vértices de V_0), se requirió modificar el algoritmo de Dijkstra, originalmente pensado para dígrafos, para hallar caminos de costo mínimo en grafos no dirigidos. La modificación, realizada en la clase `DijsktraGrafoNoDirigido`, consistió en dar

"orientación" a las relajaciones de las aristas, a pesar de que se tiene que $(u, v) = (v, u)$, con el fin de obtener resultados correctos, además de modificaciones mínimas de la misma índole al momento de hacer *backtracking* para hallar caminos de costo mínimo.

De igual manera, se implementó una modificación del algoritmo para obtener ciclos eulerianos en la clase `CicloEulerianoGrafoNoDirigido`, con la misma idea de dar orientación a las aristas y que estas estén orientadas de forma correcta al obtener el ciclo, lo cual agrega al algoritmo un trabajo de tiempo lineal con respecto a los lados del ciclo, sin afectar la complejidad temporal del algoritmo original.

De esta forma, se tiene que el algoritmo implementado halla los caminos de costo mínimo entre componentes de G', paso en el que se optó por usar el `DijkstraGrafoNoDirigido` mencionado para, de forma similar al algoritmo de Johnson, construir una matriz $matCCM$ en la que $matCCM_{ij}$ representa el costo del camino de costo mínimo entre los vértices i y j de G; se usa el hecho de que G es no dirigido para disminuir constantes ocultas y calcular solamente la triangular superior de $matCCM$, ya que como el camino de costo mínimo de i a j será el mismo que el de j a i, la matriz será simétrica. La complejidad temporal de construir $matCCM$ es, de igual forma, $O(|V||E|log|V|)$.

Consecuentemente, se usa esta matriz para obtener el costo de los lados e_t que conformarán G_t, dado por la función

$$c_{et}(e_t) = \text{mín}\{matCCM_{ij} | i \in C_i \wedge j \in C_j \wedge i \neq j\}$$

sean C_i, C_j componentes conexas distintas de G.

Prosiguiendo, para el paso de obtener el E_{MST} se usa sin más detalles el algoritmo de Prim previamente implementado en `grafoLib`, porque demostró en estudios anteriores ser más eficiente en la práctica que el algoritmo de Kruskal basado en conjuntos disjuntos. Inmediatamente, se obtienen los caminos de costo mínimo asociados para obtener E_t usando instancias de la clase `DijkstraGrafoNoDirigido` guardadas anteriormente.

Una vez agregados a G' los lados de E_t y modelado G_0 mediante el isomorfismo explicado, solo resta obtener el apareamiento perfecto del mismo, para el cual se implementaron un algoritmo ávido y *Vertex-Scan*.

Para el algoritmo ávido, se implementó usando una lista enlazada que almacena todos los lados del grafo y luego los ordena por costos en orden ascendente, para desencolarlos en cada iteración, y se implementa el conjunto V' del algoritmo mediante (i) un arreglo de valores booleanos que indican si un vértice está en V', y una variable que mantiene la cantidad de pares de vértices en V'. La complejidad temporal de este algoritmo resulta de $O(|V|^2 log|V|)$, obteniendo este tiempo por el paso que ordena los lados en orden ascendente, y dado que se tiene que $|E| = O(|V|^2)$ al ser el grafo de entrada completo.

Por el otro lado, el algoritmo Vertex-Scan tiene una implementación más sencilla que consiste solamente en tener el conjunto V' como un `MutableSet` de Kotlin y, al obtener un elemento aleatorio de este, iterar sobre todos sus adyacentes y escoger el lado que tenga costo mínimo tal que el otro vértice esté en V; esto se asemeja a escoger aleatoriamente una fila de una matriz y escanear toda la columna buscando el lado de costo mínimo, de donde obtiene su nombre el algoritmo, y dando como resultado una implementación que ocurre en tiempo asintótico $O(|V|^2)$, como se señala en [1].

Programa cliente para la solución del problema RPP.

4. Detalles de la plataforma

Los siguientes son los detalles de la máquina y el entorno donde se ejecutaron las implementaciones:

- **Sistema operativo**: GNU/Linux (Debian 11 Bullseye 64-bit).
- **Procesador**: Intel(R) Core(TM) i3-2120 CPU @3.30GHz.
- **Memoria RAM**: 4,00 GB (3,88 GB usables).
- **Compilador**: kotlinc-jvm 1.6.10 (JRE 11.0.12+7-post-Debian-2).
- **Entorno de ejecución**:OpenJDK Runtime Environment (build 11.0.12+7-post-Debian-2).

5. Resultados experimentales

Para realizar una comparación entre la solución obtenida y la solución óptima de una instancias del problema de RPP, se calculó el porcentaje de desviación con la fórmula:

$$\%desv = \frac{\text{valor obtenido} - \text{valor óptimo}}{\text{valor óptimo}} * 100 \tag{1}$$

6. Análisis de los resultados

7. Conclusiones

-
-

8. Referencias (en caso de tenerlas)

ñema.

Nombre de la instancia	Valor óptimo	Desviación (%)		Promedio Resultado V-S
		Alg. Ávido	Vertex-Scan	
UR132	**23913**	16,271	20,655	28852
UR135	**33088**	14,335	16,814	38651
UR137	**42797**	10,347	13,594	48615
UR142	**25548**	15,935	19,035	30411
UR145	**39008**	7,342	12,603	43924
UR147	**55959**	6,346	7,604	60214
UR152	**28975**	11,365	17,336	33998
UR155	**49156**	4,669	7,078	52635
UR157	**70231**	3,91	4,595	73458
UR162	**32341**	10,998	12,462	36371
UR165	**58800**	4,978	6,2	62446
UR167	**82481**	2,917	3,572	85427
UR532	**17277**	16,438	22,261	21123
UR535	**23635**	16,306	16,839	27615
UR537	**30098**	9,253	12,324	33807
UR542	**17830**	14,173	20,269	21444
UR545	**29648**	6,365	8,835	32267
UR547	**38692**	4,784	7,877	41740
UR552	**20097**	12,579	16,875	23488
UR555	**34488**	4,126	8,069	37271
UR557	**48307**	4,651	3,97	50225
UR562	**24556**	7,188	10,541	27144
UR565	**42828**	4,056	5,17	45042
UR567	**58971**	4,087	4,015	61339
UR732	**21114**	18,751	21,313	25614
UR735	**28663**	10,623	15,26	33037
UR737	**36588**	6,308	11,45	40777
UR742	**22557**	13,078	17,916	26598
UR745	**32493**	10,19	10,96	36054
UR747	**47764**	5,481	6,385	50814
UR752	**25131**	12,614	15,796	29101
UR755	**41774**	4,517	8,357	45265
UR757	**58416**	4,002	5,876	61849
UR762	**27880**	8,691	13,155	31548
UR765	**50492**	4,565	5,696	53368
UR767	**72950**	3,413	3,911	75803

Cuadro 1: Desviación de los resultados obtenidos para cada algoritmo

Nombre de la instancia	Promedio del tiempo (seg.)	
	Alg. Ávido	Vertex-Scan
UR132	0,673	0,637
UR135	1,096	0,965
UR137	1,217	1,05
UR142	1,025	0,81
UR145	1,164	1,058
UR147	1,373	1,192
UR152	1,169	0,929
UR155	1,312	1,223
UR157	1,352	1,169
UR162	1,224	1,313
UR165	1,375	1,215
UR167	1,416	1,148
UR532	0,319	0,296
UR535	0,45	0,416
UR537	0,47	0,467
UR542	0,334	0,327
UR545	0,529	0,457
UR547	0,449	0,443
UR552	0,388	0,391
UR555	0,505	0,464
UR557	0,516	0,445
UR562	0,487	0,431
UR565	0,516	0,501
UR567	0,693	0,511
UR732	0,492	0,479
UR735	0,737	0,666
UR737	0,822	0,746
UR742	0,596	0,525
UR745	0,802	0,688
UR747	0,807	0,75
UR752	0,647	0,693
UR755	1,047	0,815
UR757	0,909	0,866
UR762	0,873	0,732
UR765	1,001	0,968
UR767	0,907	0,858
Total	29,692	26,644
Promedio	0,825	0,74

Cuadro 2: Tiempo de ejecución de la implementación para cada instancia