UNIVERSIDAD SIMÓN BOLÍVAR.
SEDE SARTENEJAS.
ESTUDIO DE PREGRADO
INGENIERÍA EN COMPUTACIÓN.

Informe Proyecto 2:
The Rural Postman Problem

POR:
JESUS BANDEZ 17-10046
ROBERTO GAMBOA 16-10394

ENERO, 2022.

Diseño de la Solución

Se pidió la implementación del algoritmo presentado por Pearn y Wu para la solución del problema del cartero rural en Kotlin, con dos formas de conseguir el emparejamiento entre dos vértices. La primera consiste en usar un algoritmo ávido y la segunda usando el algoritmo Vertex-Scan.

La implementación de estos algoritmos está basada en la librería grafolib. Desde la representación de los grafos, hasta conseguir los caminos de costo mínimo. Sin embargo, la librería carece de algunas funciones que fueron necesarias y tuvieron que implementarse.

La primera de estas implementaciones fue la clase CicloEulerianoGrafoNoDirigido.kt. Esta clase consiste en una modificación de CicloEuleriano.kt para poder conseguir un ciclo Euleriano en un grafo no dirigido con lados repetidos.

También fue necesario crear la clase DijkstraGrafoNoDirigido.kt. Esta consiste en una modificación de Dijkstra.kt. La modificación no es más que, antes de ejecutar el algoritmo de Dijkstra, se consigue el digrafo asociado al grafo no dirigido. Luego, se ejecuta el algoritmo de Dijkstra sobre este grafo conseguido.

A su vez, se tuvo que modificar la clase CFC.kt. Esta clase permite conseguir las componentes fuertemente conexas de un grafo, pero no era capaz de retornar los vértices que pertenecen a una componente específica. El cambio a la clase consiste en agregar un método que permita retornar los vértices que pertenecen a una componente dando como parámetro el identificador de la misma.

Se usaron diccionarios para actuar como una correspondencia entre el grafo dado como parámetro y los grafos creados a partir de agregar vértices al grafo inducido por los lados requeridos. Más adelante se explica este punto con más detalle.

Detalles de la implementación

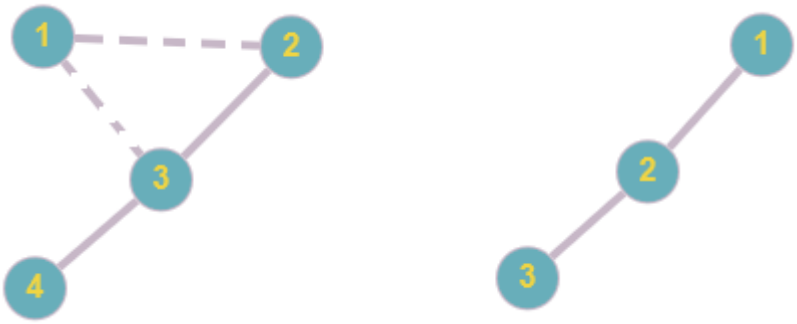
El algoritmo ávido para el emparejamiento y el algoritmo Vertex Scan, no necesitaron de estructuras extras; bastaron los grafos que tiene la librería grafolib. En el caso de la cola de prioridad para el algoritmo ávido, se usó la implementada en la librería de Kotlin.

Al implementar el algoritmo Pearn y Wu se encontraron varias dificultades al momento de agregar vértices a un grafo. Esto ocurrió ya que los grafos de grafolib no permiten agregar vértices a un grafo ya creado. Por esto, se recurrió a usar un grafo temporal. Suponga que se quiere agregar vértices a un grafo G; antes de agregar los vértices, se consigue qué cantidad de vértices se deben agregar. Después, se crea un grafo que tiene como numero de vértices la cantidad de vértices que tiene el grafo G, más la cantidad de nuevos vértices a agregar. Posteriormente, se agregan al grafo temporal los lados de G y los lados que inciden en los nuevos vértices. Finalmente, el grafo temporal pasa a ser el nuevo G.

Por otra parte, al agregar los vértices que estaban en otro grafo. Estos vértices tenían un identificador, un número entero, el cual no era posible garantizar que se conservara al agregarse al otro grafo. Esto sucede porque todo vértice que se agregara tendría como identificador un número entero de 0 a la cantidad de vértices del grafo, más la cantidad de vértices a agregar. Para resolver este problema se recurrió a la siguiente estrategia.

En la siguiente figura se tienen dos grafos, se quiere agregar el vértice 4 del grafo de la izquierda al grafo de la derecha:

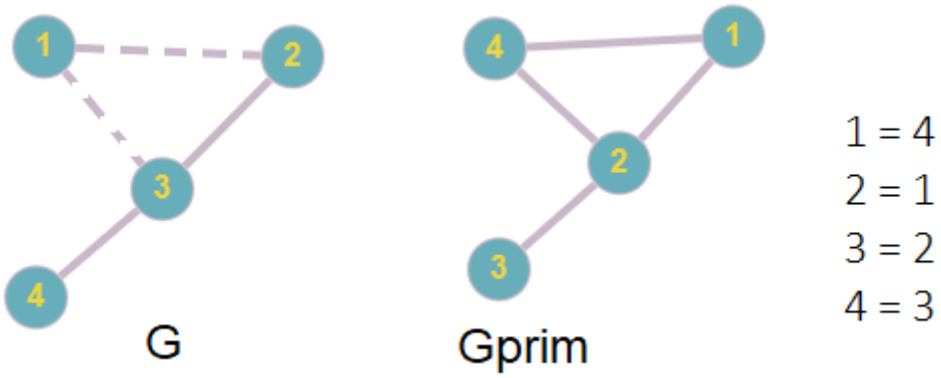
A primeras, es imposible hacer esto. No se puede establecer una correspondencia entre ambos grafos porque no se sabe qué vértice representa a cuál. Si se tuviera una correspondencia ya creada, se podría agregar el vértice. Ahora bien, considere la siguiente imagen. Se tiene que el grafo de la izquierda (llamémoslo G) tiene los lados requeridos (lados seguidos) y los no requeridos (lados punteados), el grafo de la derecha (llamémoslo Gprim) es el grafo inducido por los lados requeridos de G:



En este caso se establece la siguiente correspondencia: el vértice 1 de Gprim es el vértice 2 de G, el vértice 2 de Gprim es el vértice 3 de G, y el vértice 3 de Gprim es el vértice 4 de G.

En la implementación del algoritmo de Pearn y Wu, esta correspondencia se genera al momento de crear el grafo Gprim. La correspondencia se genera como dos diccionarios donde tanto la clave como el valor son números enteros. Uno de los diccionarios tiene como clave un vértice de G y como valor un vértice de Gprim. El otro diccionario tiene como clave un vértice de Gprim y como valor un vértice de G. De manera que, si se tiene un vértice de un grafo es posible conseguir su representación en el otro grafo.

Así, suponga que por alguna razón se quiere agregar el vértice 1 de G a Gprim con sus lados. Primero, se consigue la cantidad de vértices a agregar, en este caso es 1. Luego, se agrega el vértice 1 a la correspondencia entre G y Gprim de la siguiente forma: El vértice 1 de G, será representado por el vértice 4 en Gprim. De tal modo, se tiene que la cantidad de vértices a agregar es 1, por tanto, se crea un nuevo grafo con la cantidad de vértices que tiene Gprim (3) más 1. Prospectivamente, se agregan los lados de G a Gprim usando la nueva correspondencia: agregar el lado (1, 2) de G se convierte en agregar el lado (4, 1) a Gprim y agregar el lado (1, 3) de G se convierte en agregar el lado (4, 2) a Gprim. Finalmente, el grafo G, Gprim y la correspondencia quedarían de la siguiente forma:



Esta estrategia fue usada en todos los casos en los que era necesario agregar vértices a un grafo. A lo largo del algoritmo es necesario insertar vértices en un grafo dos veces. Uno en el caso de que el grafo inducido por los lados requeridos de G no sea conexo, y la segunda vez en el caso de que el grafo no sea de grado par. Si el grafo inducido es conexo, pero no par, entonces sólo se debe insertar vértices en él una vez.

Con respecto al resto de la implementación, se usó las estructuras Array y MutableList de Kotlin para representar las matrices de caminos de costo mínimo. Al momento de conseguir los mencionados caminos de costo mínimo, se usó el algoritmo de Dijkstra ya que los costos de las aristas eran no negativos. Se pudo haber usado uno de los algoritmos Floyd Warshal o Jhonson para encontrar estos caminos. Empero, Jonhson habría sido más lento debido a la ejecución Bellman Ford, y Floyd Warshal habría necesitado conseguir la matriz de costos del grafo. Por otra parte, para conseguir el árbol mínimo recubridor, se usó el algoritmo de Prim debido a ser ligeramente más rápido que el de Kruskal.

Exceptuando las modificaciones y la necesidad de usar correspondencias que asimilan (pero no son) isomorfismos entre los grafos generados durante el algoritmo, bastó usar la librería grafolib y las librerías de Kotlin para implementar el algoritmo de Pearn y Wu.

Resultados experimentales

Tabla 1. Porcentaje de desviación de la solución óptima usando cada heurística.

Nombre de la Instancia	Valor óptimo de la instancia	% de desviación usando la heurística ávida	% de desviación usando la heurística Vertex-Scan (Promedio de las soluciones de 3 corridas)
UR532	17277	16,44%	21,57%
UR535	23635	16,23%	19,03%
UR537	30098	9,25%	14,01%
UR542	17830	14,74%	19,86%
UR545	29648	6,36%	6,78%
UR547	38692	4,11%	5,33%
UR552	20097	12,58%	18,88%
UR555	34488	3,48%	8,60%
UR557	48307	4,65%	4,65%
UR562	24556	7,19%	8,90%
UR565	42828	4,12%	3,76%
UR567	58971	4,09%	5,77%
UR732	21114	19,53%	21,30%
UR735	28663	11,65%	20,70%
UR737	36588	6,31%	9,43%
UR742	22557	13,15%	15,81%
UR745	32493	8,70%	12,70%
UR747	47764	5,48%	6,57%
UR752	25131	12,61%	11,27%
UR755	41774	3,77%	6,66%
UR757	58416	3,25%	5,83%
UR762	27880	9,67%	12,89%
UR765	50492	3,96%	6,36%
UR767	72950	3,59%	4,04%
UR132	23913	16,27%	17,69%
UR135	33088	14,41%	14,39%
UR137	42797	10,21%	9,54%
UR142	25548	15,45%	22,48%
UR145	39008	7,16%	11,00%
UR147	55959	6,32%	8,48%
UR152	28975	13,95%	15,63%
UR155	49156	4,69%	10,33%
UR157	70231	2,69%	5,23%
UR162	32341	9,26%	13,09%

UR165	58800	5,00%	6,95%
UR167	82481	3,00%	4,02%

Tabla 2. Tiempo que toma cada heurística en hallar la solución de las instancias.

Nombre de la Instancia	Tiempo promedio heurística ávida	Tiempo promedio heurística Vertex-Scan (Promedio de las soluciones de 3 corridas)
UR532	15,35	15,62
UR535	36,09	35,17
UR537	18,14	16,83
UR542	22,41	22,70
UR545	17,96	17,99
UR547	3,47	3,61
UR552	32,77	31,90
UR555	6,75	6,70
UR557	2,30	2,49
UR562	27,02	27,89
UR565	3,69	3,47
UR567	1,93	2,21
UR732	70,42	69,36
UR735	165,13	166,88
UR737	43,97	44,50
UR742	107,25	109,25
UR745	126,30	127,79
UR747	11,52	11,81
UR752	124,50	123,97
UR755	46,98	47,58
UR757	4,87	5,00
UR762	132,43	133,27
UR765	7,74	8,35
UR767	4,63	5,33
UR132	220,37	217,09
UR135	469,00	473,62
UR137	162,55	159,50
UR142	290,29	293,61
UR145	362,85	366,64
UR147	30,62	32,22
UR152	346,30	348,21

UR155	39,40	39,89
UR157	15,50	17,32
UR162	399,99	400,17
UR165	68,69	70,22
UR167	7,27	9,01

Los resultados presentados en ambas tablas se obtuvieron usando un computador con las siguientes características:

- Sistema operativo: Ubuntu 21.10
- Versión del compilador de Kotlin: 1.6.10
- Versión de la máquina virtual de Java: 11.0.13
- Modelo de CPU: Intel Core I5-560M
- Cantidad de memoria del computador: 4GB (DDR3)

Análisis de los resultados

En todos los casos se obtuvieron valores dentro del rango de las soluciones del algoritmo propuesto por Christofides tanto usando la heurística ávida como usando la heurística Vertex-Scan. Como la heurística Vertex-Scan escoge el vértice desde el que inicia su operación al azar, esto impacta en el costo y el tiempo que se obtienen al usarla y los valores obtenidos para cada instancia con la misma presentados en la Tabla1 y Tabla2 representan el promedio de soluciones de 3 corridas. En la mayoría de las instancias ambas heurísticas presentan soluciones de costo similar, aunque la heurística ávida siempre presenta menor desviación respecto al valor óptimo de la instancia, es decir, soluciones más cercanas a la óptima. A su vez, al hallar la solución de cada instancia, la misma se obtiene en tiempo similar con ambas heurísticas. No obstante, cuando la cantidad de vértices es menor la heurística Vertex-Scan puede presentar tiempos ligeramente menores, aunque de igual manera con una desviación mayor que la obtenida con la heurística ávida. Esta diferencia en los tiempos se incrementa a medida que aumenta el número de vértices y sus componentes conexas, donde en general la heurística ávida puede hallar soluciones de menor costo más rápido que la heurística Vertex-Scan. En los casos donde el grafo G_r ya es conexo en la entrada del algoritmo, se obtienen soluciones de manera mucha más rápida con ambas heurísticas que en los casos cuando G_r no es conexo.

Conclusiones

En la mayoría de los casos cuando el grafo G_r es conexo (par o impar) o no conexo en el inicio del algoritmo, se obtienen soluciones de menor costo con la eucarística ávida. En los casos donde la eucarística Vertex-Scan obtiene la solución en un menor tiempo se debe a que escoge una arista de inicio optima. Esto ocurre en pocas ocasiones, por tanto, es mejor utilizar la eucarística ávida para resolver el problema de RPP, sobre todo cuando se quiere obtener una solución usando un gran número de vértices y de aristas.