PROBLEMA DE ENRUTAMIENTO DE VEHÍCULOS

Sandra del Mar Soto Corderi
No. cuenta: 315707267

11 de febrero de 2021

1. Introducción

El problema del que se hablará en este reporte es el de enrutamiento de vehículos mejor conocido como **VRP**. El VRP implica enrutar una flota de vehículos, cada uno de ellos visitando un conjunto de nodos de modo que cada nodo sea visitado exactamente por un vehículo solo una vez. Entonces, el objetivo es minimizar la distancia total recorrida por todos los vehículos. Así mismo, hay que considerar que los vehículos no consuman más de su capacidad dada.

Este es un problema considerado NP-duro, podemos ver que el número de posibles soluciones del orden de $n!$, donde n es el número de nodos o clientes. Hay muchas variantes del VRP, pero en este proyecto se usó la versión más simple que no utiliza ventanas de tiempo, sino que solo considera las capacidades, esta variante es mejor conocida como CVRP (Capacitated Vehicle Routing Problem).

Tenemos a una gráfica conexa no dirigida $G(V, A)$ con $V = \{0, 1, ..., n\}$ y $E = \{(i, j) : i, j \in V, i \neq j\}$.
El vértice 0 representa el depósito y cada uno de los otros vértices $i \in V - \{0\}$ son clientes con pedidos no negativos q_i y na distancia no negativa $c_{i,j}$ que está asociada con cada arco $(i, j) \in E$ y representa el costo de viaje del nodo i al nodo j.
Tenemos que la matriz de costo es simétrica, es decir $\forall_{i,j} \in V c_{ij} = c_{ji}$. Así mismo, no se permite el uso del arco (i, i), por lo que $c_{i,i} = 0 \forall i \in V$.
Así que las soluciones son un conjunto de rutas de envío que satisfacen los puntos de distribución y obtienen el mínimo costo total para todos los vehículos. En práctica, minimizar el costo total es equivalente a minimizar la distancia visitada por $m > 1$ vehículos.

Matemáticamente se puede ver como las siguientes fórmulas donde:

n - número de nodos para cada instancia.
m - número de vehículos usados para cada instancia.
x_{ij} es igual a 1 si el vehículo viaja del nodo i a j o es igual a 0 en otro caso.
y_{ik} es igual a 1 si el nodo i es visitado por el vehículo k o es igual a 0 en otro caso.

1. $\min \sum_{i=0}^{n} \sum_{j=0}^{n} c_{ij} x_{ij}$
2. $\sum_{i=0}^{n} x_{ij} = 1 \ (j = 1, ..., n)$
3. $\sum_{i=0}^{n} x_{i0} = m$
4. $\sum_{j=0}^{n} x_{ij} = 1 \ (i = 1, ..., n)$
5. $\sum_{j=0}^{n} x_{0j} = m$
6. $\sum_{i=0}^{n} q_i y_{ik} \leq Q (k = 1, .., m)$
7. $\sum_{i \in S} \sum_{j \in N-S} x_{ij} \geq 1 \ (\phi \neq S \subseteq \{1, .., n\}, |S| \geq 2)$
8. $\sum_{i \in N-S} \sum_{j \in S} x_{ij} \geq 1 \ (\phi \neq S \subseteq \{1, .., n\}, |S| \geq 2)$
9. $x_{ij} \in \{0, 1\} \ (i = 0, ..., n; j = 0, ..., n)$
10. $y_{ik} \in \{0, 1\} \ (i = 0, ..., n; k = 1, ..., m)$

La fórmula 1 minimiza la distancia total viajada en la ruta.
Las fórmulas 2 y 3 aseguran que el vehículo llega una vez a cada nodo de la ruta y m veces al depósito.
Las fórmulas 4 y 5 aseguran que el vehículo sale de cada nodo y deja el depósito m veces.
La fórmula 6 sirve para evitar que los vehículos carguen más de su capacidad.
Las fórmulas 7 y 8 evitan la presencia de subrutas para cada vehículo.
Las fórmula 9 y 10 definen condiciones binarias en las variables.

1.1. Búsqueda Tabú *(Tabu Search)*

Ahora hablaremos de la heurística que se utilizó:

Buscando artículos de investigación sobre el problema de enrutamiento de vehículos, me encontré con varios que usaban heurística híbridas, en estos híbridos siempre mencionaban a la búsqueda tabú como una parte del algoritmo.

Algo que observé igualmente fue que en muchos casos creaban la solución inicial de forma inteligente, es decir no solamente aleatoria y a partir de esa solución optimizaban. Para obtener mi solución inicial seguí el método usado en el artículo [2], donde aplican el algoritmo de barrido (sweep algorithm). El algoritmo de barrido ordena los puntos de demanda por su coordenada polar con la fórmula: $An(i) = arctan\{(y(i) - y(0))/(x(i) - x(0)\}$. Y después se inicia el "barrido"del cliente i, quien no ha sido visitado, desde el ángulo más pequeño hasta el más largo, hasta que todos los clientes están asignados en varias rutas.

Este artículo[1] describe como resolver el problema usando búsqueda tabú. Vamos a explicar de una forma muy sencilla en que consiste la búsqueda tabú: La búsqueda tabú es una heurística para búsqueda local que fue declarada por Fred Glover [3] y su objetivo es salirse de óptimos locales para alcanzar óptimos globales, por ello maneja una lista tabú, para evitar seguir en vecindades cerca de un óptimo local y poder recorrer el espacio de búsqueda de forma más inteligente y rápida. Hay dos conceptos importantes en la búsqueda tabú para lograr mejores soluciones, esos son la diversificación (se busca recorrer lo más posible del espacio de búsqueda) y la intensificación (concentrarse en una región local sabiendo que hay una buena solución por ahí). Estos dos conceptos se usan de forma implícita en la implementación dada durante la obtención del mejor vecino, ya que diversifica al usar la lista tabú e intensifica al buscar en la vecindad del mejor vecino posible. La búsqueda tabú utiliza memorias, estas son a corto, mediano y largo plazo. La de corto plazo es nuestra lista tabú, las de mediano y largo se utilizan para hacer una búsqueda aún más inteligente, donde vamos guardando los movimientos posibles al obtener vecinos y se buscan las similitudes de estos movimientos para aceptar o no vecinos, esto no se implementé tan explícitamente, pero se mantuvo la idea-

En este proyecto se uso la versión simplificada, donde no se hace manejo de la memoria a mediano y largo plazo, es decir se siguió un algoritmo muy parecido al que presentan en [4]. A continuación se muestra el algoritmo usado en el proyecto:

Algorithm 1: Búsqueda Tabú simplificada

Output: S_{mejor} mejor solución encontrada
Input: La condición de paro SC, tamaño máximo de lista tabú TM, $S_{inicial}$ una solución inicial
$S_{mejor} \leftarrow S_{inicial}$ $ListaTabu \leftarrow S_{mejor}$ **while** $\neg SC$ **do**
  for $S_{candidato} \in Vecindad_s$ **do**
    if $\neg SeEncuentra(S_{candidato}, ListaTabu)$ **then**
      $ListaCandidatos \leftarrow S_{candidato}$
    end
  end
  $S_{candidato} \leftarrow BuscaMejorCandidato(ListaCandidatos)$
  if $Costo(S_{candidato}) \leq Costo(S_{mejor})$ **then**
    $S_{mejor} \leftarrow S_{candidato}$
    $ListaTabu \leftarrow S_{candidato}$
  end
  while $ListaTabu > TM$ **do**
    $EliminaPrimer(ListaTabu)$
  end
end
return S_{mejor}

Para obtener los vecinos que se utilizaran como candidatos se utilizo el método swap, es decir se tomaron aleatoriamente dos ciudades entre vehículos y se intercambiaron.

2. Tecnologías usadas en el programa

- **Lenguaje de programación:** Kotlin 1.4.10.
  Por presiones de tiempo, decidí usar un lenguaje que ya conociera y tuviera cosas parecidas implementadas, así que para reciclar código y por facilidad se usó el mismo lenguaje de programación del proyecto pasado
- **IDE:** IntelliJ IDEA
  Es la IDE más popular para Java y es compatible con Kotlin, por lo que se usó
- **Sistema de construcción:** Gradle 6.7
  Al investigar la documentación de Kotlin, se mencionaba a Gradle y a Maven como las mejores opciones para usar como sistema de construcción. Gradle tenía el tutorial más corto, así como manejaba las dependencias más fácilmente que Maven, por ello se escogió.
- **Documentación:** dokkaHtml 1.4.10.2
  Es el sistema de documentación oficial de Kotlin
- **Insumos:** Los insumos (datos de entrada) fueron obtenidos de la siguiente liga: http://vrp.atd-lab.inf.puc-rio.br/index.php/en/ La razón por la que se escogieron estos insumos fue porque son muy sencillos de leer y están muy completos, así mismo incluyen soluciones que sirven para comparar el rendimiento del proyecto.
- **Control de versiones:** Para mantener el control de versiones se utilizó Git 2.17.1 y el repositorio en línea se encuentra alojado en GitHub.

3. Diseño del programa

El proyecto se hizo con un enfoque orientado a objetos, por la naturaleza del lenguaje escogido.

A continuación se van a enlistar las clases usadas y lo que hace cada una:

- Vehiculo.kt
  Esta clase almacena la información de los vehículos de nuestro problema, es decir la capacidad que tiene así como la ruta a la que se le asignará. Se declaró como una *data class* de Kotlin, ya que este tipo de clases se dedican únicamente a almacenar información, lo que hace que al construir el objeto, hayan varios métodos ya implementados. Igualmente es necesario comentar que no se incluyó la implementación de getters y setters debido a que en Kotlin en la documentación mencionan que no son necesarios.
- Grafica.kt
  Esta clase es de las que cuentan con más métodos y es donde se realizan todas las funciones de la gráfica. Aquí se obtienen las funciones de costo, que en este caso es el costo de las distancias entre los nodos de la ruta más la distancia del depósito al primer elemento de la ruta, más la distancia del último elemento de la ruta al depósito. Así mismo aquí es donde se crea la solución inicial usando el algoritmo de barrido.
- Solucion.kt
  Esta clase es bastante sencilla, ya que solo crea soluciones para el vrp, es decir invierte aleatoriamiente vecinos de la asignación para crear nuevas con la esperanza de que mejore.
- Heuristica.kt
  Esta podría considerarse como la clase más ilustrativa del objetivo de este proyecto ya que es la que realiza el procedimiento de la heurística de búsqueda tabú. En pocas palabras el procedimiento que se sigue es generar la vecindad de una solución, ver en esa vecindad cual es el vecino con mejor costo que no sea tabú y compararlo con la mejor solución hasta el momento. La lista tabú se va llenando con los mejor mejores vecinos y si sobrepasa su límite de tamaño se eliminan los primeros obtenidos.
- DAO.kt
  Esta clase funciona como nuestro Data Access Object, es la clase que se conecta con la base de datos directamente. Usamos un DAO por nuestro diseño basado en orientación a objetos y para mantener el patrón Modelo-Controlador. Normalmente los DAO no se aconsejan para aplicaciones donde el tiempo de ejecución importa pero es mucho más eficiente tener que hacer una o dos consultas a la base de datos mediante el DAO que hacer varias consultas dentro del modelo.
- Main.kt
  Este archivo no es una clase como tal, sino es el main donde se ejecuta el sistema. Lo que hace es tomar el archivo del problema dada por el usuario con las cuales crea un objeto gráfica, este objeto gráfica manda a

crear soluciones usando el rango de semillas proporcionado y manda a llamar a la heurística para mejorar la solución en los parámetros que se den. Al final se imprimen los costos de todas las mejores soluciones y se regresa la asignación de la mejor solución de todas.

Para ver más detalladamente la implementación, favor de generar la documentación con dokkahtml.

4. Resultados

Se realizaron pruebas usando la solución inicial dada por el algoritmo de barrido y con una solución inicial aleatoria. A partir de la solución inicial aleatoria no se obtenían muy buenos valores y a veces se tenían resultados no factibles. Mientras que con la solución inicial hecha por el algoritmo de barrido, siempre devolvíamos soluciones factibles, pero nos atorábamos muy fácilmente en mínimos locales, lo que implicaba que el resultado no mejoraba, aunque era un buen resultado.

Así que fue una decisión de diseño escoger que la solución inicial siempre fuera hecha por el algoritmo de barrido, ya que considero preferible devolver siempre soluciones factibles aunque no sean las mejores.

A continuación se muestran las mejores soluciones obtenidas para dos archivos distintos:

```
Archivo: A-n32-k5.vrp
Semilla: 12
Vehiculo 1
Ruta: [11, 6, 26, 21, 13, 2, 22, ]
Costo 334.36
Vehiculo 2
Ruta: [17, 32, 31, 20, 18, 8, ]
Costo 263.805
Vehiculo 3
Ruta: [14, 27, 3, 4, 7, 24, 29, 5, ]
Costo 323.259
Vehiculo 4
Ruta: [12, 15, 9, 19, 25, 10, 23, 28, ]
Costo 447.595
Vehiculo 5
Ruta: [16, 30, ]
Costo 163.68
Costo Total: 1532.699
```

Costo Ideal de A-n32-k5.vrp: 784

```
Archivo: A-n63-k9
Semilla: 43
Vehiculo 1
Ruta: [20, 31, 46, 30, 55, 6, 27, ]
Costo 205.372
Vehiculo 2
Ruta: [12, 34, 22, 41, 15, 56, 59, ]
Costo 197.026
Vehiculo 3
Ruta: [33, 54, 45, 51, 23, ]
Costo 260.384
Vehiculo 4
Ruta: [18, 36, 29, 52, 58, 13, ]
Costo 234.409
Vehiculo 5
Ruta: [38, 16, 7, 63, 53, 11, 48, 19, 40, ]
Costo 391.697
Vehiculo 6
Ruta: [57, 5, 24, 47, 44, 2, 32, ]
Costo 300.462
Vehiculo 7
```

```
Ruta: [39, 17, 10, 43, 62, 42, ]
Costo 350.36
Vehiculo 8
Ruta: [4, 8, 49, 61, 37, 50, ]
Costo 327.811
Vehiculo 9
Ruta: [25, 26, 14, 9, 60, 35, ]
Costo 302.417
Vehiculo 10
Ruta: [21, 28, 3, ]
Costo 174.022
Costo Total: 2743.96
```

Costo Ideal de A-n63-k9: 1616

Podemos observar que hay una diferencia algo notable con el costo ideal, pero al compararlo con los resultados obtenidos de usar una solución inicial aleatoria, consideramos que estos costos son muy buenos.

5. Comentarios extras sobre el proyecto

Originalmente iba a realizar la implementación dada en [2] donde se usa colonia de hormigas combinada con búsqueda tabú, pero al final no integré al proyecto la implementación de la colonia de hormigas, el código correspondiente a esta implementación aún se encuentra en el git como referencia. En un futuro me gustaría poder retomar el proyecto e integrar esa implementación. Fuera de eso tengo soluciones bastante buenas y aseguro siempre son factibles.

Referencias

[1] Huang, F., & Liu, C. (2010). An improved tabu search for open vehicle routing problem. 2010 International Conference on Management and Service Science. doi:10.1109/icmss.2010.5576368

[2] Yousefikhoshbakht, M. and Khorram, E., 2012. Solving the vehicle routing problem by a hybrid meta-heuristic algorithm. Journal of Industrial Engineering International, 8(1).

[3] Glover F, Laguna M. "Tabu search". S.L.: Kluwer Academic; 1993.

[4] Brownlee J., Clever Algorithms Nature-Inspired Programming Recipes. 2011; 76-90.

Si hay duda de alguna fuente favor de contactarme para proporcionarla.