PROBLEMA DE ENRUTAMIENTO DE VEHÍCULOS

Sandra del Mar Soto Corderi
No. cuenta: 315707267

10 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.

Viéndolo matemáticamente tenemos a una gráfica conexa no dirigida $G(V, A)$ con $V = \{0, 1, ..., n\}$ y $E = \{(i, j) : i, j \in V, i \neq j\}$

1.1. Búsqueda Tabú *(Tabu Search)*

Ahora hablaremos de la heurística que se utilizó:

Buscando artículos de investigación sobre el problema de asignación generaliza, me encontré con varios que usaban heurística híbridas, en estos híbridos siempre mencionaban a la búsqueda tabú.

Este artículo describe como resolver el problema usando búsqueda adaptativa. La heurística de búsqueda adaptativa propuesta para resolver el GAP puede ser descrita en una estructura general incluyendo tres fases, que son aplicados repetidamente hasta que algún criterio de parada sea verificado:

Fase 1: Generar una solución usando una heurística aleatoria tipo greedy.
Fase 2: Aplicar un método de búsqueda local.
Fase 3: Actualizar los parámetros (si hay).
Más adelante en el artículo explican como la búsqueda tabú entra en estos.

De todo lo anterior decidí usar un método greedy para obtener mi solución inicial, quise implementar el algoritmo de Martello y Toth, pero consideré era demasiado

Ahora 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 [**?**]. 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}

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
- **Graficación:** Gnuplot 5.0
- **Insumos:** Los insumos (datos de entrada) fueron proporcionados por el profesor Victor mediante una base de datos relacional *SQL*. El sistema manejador de la base de datos utilizado es SQLite 3.16.2. El controlador del *SMBD* es una biblioteca para Kotlin SQLite-JDBC 3.28.0.
- **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.

El proyecto se dividió en tres paquetes:

- modelo
  En este paquete se crean los objetos que estaremos manejando en el proyecto y cuya información no cambiará, como son los valores de las tareas, los trabajadores, el costo de sus asignaciones y las capacidades necesarias.
- gap

En este paquete tenemos todos los métodos o funciones necesarios para resolver el problema de gap sin aplicar la heurística

- tabu

  En este paquete tenemos los métodos o funciones necesarios para aplicar la heurística de búsqueda tabú al problema anterior.

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

- Tarea.kt

  Esta clase almacena la información de las tareas de nuestro problema, es decir el nombre y el id de cada tarea que se 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.

- Trabajador.kt

  Esta clase es análoga a la de Tarea en la cuestión que es una *data class* donde guardamos la información de los trabajadores.

- 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 una asignación de tareas y verificamos si son factibles. En esta clase se generan todos los métodos que puedan incluir verificar conexiones entre los trabajadores y las tareas. Así mismo aquí es donde se crea la solución inicial de un forma greedy, el código viene explicado, pero en pocas palabras, lo que se hace es tomar las tareas de forma aleatoria y buscar el trabajador de menor costo que siga manteniendo la solución factible, si esto no es posible se da una solución inicial aleatoria.

- Solucion.kt

  Esta clase es bastante sencilla, ya que solo crea soluciones para el gap, es decir invierte aleatoriamiente vecinos de la asignación para crear nuevas con la esperanza de que mejore. Esta clase originalmente iba a ir integrada con la clase de Grafica, pero era más limpio y fácil de comprender manejar los métodos de la clase heurística con un objeto sencillo Solucion que con el objeto de una clase llena de métodos como es Grafica.

- 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 recocido simulado con aceptación por umbrales. 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 la lista de ciudades 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

Referencias

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

[2] Mazzeo, S., & Loiseau, I., 2004. An ant Colony algorithm for The capacitated vehicle routing. Electronic Notes in Discrete Mathematics 18. 181–186.

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

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