Práctico Algoritmos evolutivos 2020

Ignacio Camiruaga, Franco Donnangelo - Facultad de Ingenieria UDELAR
30 de Septiembre 2020, Montevideo, Uruguay

I. Descripción del problema

El práctico consiste en construir un algoritmo capaz de resolver una variante del problema del Capacitated Routing Problem With TIme Window (CVRPTW) . Es un problema NP-Completo para el cual intentaremos buscar una solución óptima por medio de algoritmos evolutivos. El problema consiste en distribuir productos a clientes en un día, satisfaciendo toda la demanda. Para ello se cuenta con vehículos que deben comenzar y terminar su recorrido en un deposito, el objetivo es calcular las rutas y las asignaciones de clientes a vehículos con el fin de optimizar el costo total de distribución. Para la optimización debemos contemplar las siguientes variables: los salarios de los conductores, la capacidad de cada vehículo, la cantidad de vehículos, la cantidad de clientes, el tiempo que toma el vehículo en ir de un cliente a otro, el tiempo de descarga de los productos en cada cliente, los horarios de atención de cada cliente (lo cual implica penalidades por llegar tarde y esperas con el conductor ocioso si se llega antes). A continuación explicamos como resolvimos este problema en el contexto del curso de algoritmos evolutivos.

II. Biblioteca utilizada

Se utilizó la biblioteca Malva implementada en C++. Para construir la solución nos inspiramos fuertemente en la resolución del problema de coloreo de barrios presentado en clase ([1]). Como experiencia nos gustaría destacar que se nos hizo cuesta arriba el manejo de memoria ya que no estamos muy acostumbrados a C++. Pero por otro lado el tiempo de ejecución en las pruebas es muy bajo y eso nos permitió poder analizar rápidamente los resultados obtenidos al modificar diferentes variables como se presenta en la sección de resultados.

III. Representación de soluciones candidatas

Decidimos representar las soluciones con un array de enteros. El array contiene el recorrido que hace cada vehículo secuencialmente, empezando y terminando con un 0 para cada uno, donde el 0 representa el depósito. En caso de que ese vehículo no se utilice, tendremos dos 0 consecutivos.

Ejemplo de solución para tres vehículos y 6 clientes:
0 1 2 0 | 0 4 3 0 | 0 6 5 0

Ejemplo de solución para tres vehículos donde usamos solo los primeros dos:
0 1 2 0 | 0 4 3 5 6 0 | 0 0

Ejemplo de solución para tres vehículos donde usamos solo el primero y el último:
0 1 2 0 | 0 0 | 0 4 3 5 6 0

IV. Inicialización de la población

Para inicializar la solución, lo que hacemos es recorrer la lista de clientes y para cada uno elegimos un vehículo al azar. Si este vehículo ya tiene asignada su carga máxima, elegimos otro al azar hasta encontrar uno que cumpla con los requisitos de capacidad del vehículo.

Este método nos permite darle un carácter aleatorio a la inicialización y comenzar con una población que consideramos lo suficientemente diversa.

V. Operadores evolutivos utilizados

V-A. Check de factibilidad

Chequeamos que los vehículos no tengan asignados clientes con demanda total que sobrepasa su capacidad. Si esto ocurre descartamos el individuo.

V-B. Operador de mutación

Se implementaron dos tipos de operadores de mutación para representaciones de permutaciones: de intercambio y de inserción. El depósito de valor 0 en la solución nunca será mutado. En el intercambio se seleccionan dos clientes al azar y los cambiamos de posición. En la inserción se selecciona un cliente y se lo mueve a otra posición al azar. La operación de inserción le permite al algoritmo variar el tamaño del recorrido de los camiones, lo cual no ocurre en el intercambio o en el operador de cruzamiento. Luego de aplicar la mutación chequeamos la factibilidad de la solución y se descarta si no es factible.

V-C. Operador de cruzamiento

El operador de cruzamiento utilizado es el algoritmo de Partially Mapped Crossover (PMX). Esto se debe a que necesitamos que la solución sea una permutación de los clientes. Para nuestro caso particular, no intercambiamos los valores del array de solución que contuviera ceros (osea los depósitos). Como consecuencia, la función de cruzamiento mantiene la cantidad de clientes por los cuales tiene que pasar un vehículo. Para lograr esto modificamos un poco la función de crossover, pasamos a verificar que en el índice a intercambiar, no haya un cero en el valor de ninguno de los array de soluciones. Luego de hacer el cross, chequeamos que las soluciones sean factibles, la que no sea factible la descartamos (posiblemente descartando las dos).

VI. Función de fitness

El fitness se calcula con la suma de los siguientes valores calculados para cada vehículo y multiplicados por el salario del conductor del mismo:

- Distancia euclidiana entre todos los clientes a recorrer, incluido el depósito tanto al inicio como al final.
- Tiempo de descarga en cada cliente.
- En caso de que se de, el tiempo que tiene que esperar un conductor a que comience el horario de atención del cliente.

Además, se agrega, en caso de que exista, una penalidad por llegar tarde a un cliente, calculada como el tiempo entre el fin del horario de atención del cliente y el horario de llegada del vehículo.

VII. Interpretación de la salida

En la primer línea imprimimos la solución tal cual se describe en la sesión de representación de la misma.

En la segunda línea, los tipos de vehículo van a quedar siempre ordenados, ya que de la forma que representamos nuestra solución, ordenamos los vehículos por tipo. En caso de que la instancia no sea heterogénea, esta línea queda vacía. Se imprime solo el tipo de los vehiculos utilizados, por lo que la cantidad de caracteres no sera estrictamente igual a la cantidad de vehiculos que ofrece la instancia.

La tercer línea imprime el fitness de la mejor solución obtenida. El resultado se guarda en un archivo con el nombre que le indiquemos. Damos más detalles en la sección de cómo ejecutar el algoritmo.

VIII. Resultados

VIII-A. Configuración paramétrica

Se obtiene que el algoritmo ejecuta en menos de 10 segundos para cualquiera tipo de instancia, por lo tanto decidimos establecer la cantidad de individuos en 100 y las generaciones en 1000.

Probamos varias opciones para las probabilidades de mutación y cruzamiento, ejecutando varias instancias 10 veces. Con probabilidad de cruzamiento 0.6 probamos las siguientes variantes de mutación, obteniendo los siguientes valores de fitness (Tabla I):

TABLE I

	R101	HR201	RC101	HRC201	C201	HC101
0.001	4953.79	8975.09	4792.31	7103.32	**39575.3**	21478.2
0.01	6244.39	8726.93	6106.5	7144.86	46563.1	26046.3
0.005	**4718.17**	**8099.31**	**4545.61**	**5821.13**	40485.3	**21038.8**

Observamos que 0.005 es la mejor probabilidad de mutación para la mayoría de las instancias. Luego con probabilidad de mutación en 0.005, probamos diferentes valores para la probabilidad de cruzamiento, obteniendo los siguientes resultados (Tabla II):

Vemos que entre 0.4 y 0.6 no hay gran diferencia pero en promedio, obtenemos los mejores resultados con probabilidad de cruzamiento de 0.4. En la Tabla III se muestran los resultados de todos los tipos de instancia con los parámetros 0.005 y 0.4.

TABLE II

	R101	HR201	RC101	HRC201	C201	HC101
0.2	4956.37	8481.41	4927.28	6589.17	40093.8	**20680.5**
0.4	**4654.9**	**7960.28**	**4527.11**	6301.54	**39861.7**	20817.5
0.6	4718.17	8099.31	4545.61	**5821.13**	40485.3	21038.8

TABLE III

R101	R201	HR101	HR201	RC101	RC201	HRC101
4654.9	10529.5	6003.9	7960.28	4527.11	9663.09	6095.29

HRC201	C101	C201	HC101	HC201
6301.54	20781.5	39861.7	20817.5	41595.9

IX. Posibles mejoras

Implementación de función de corrección de soluciones no factibles. Esto podría ayudar a acelerar la convergencia del algoritmo. Ya que en caso de que un operador evolutivo genere una solución no factible, los cambios se descartan y la generación siguiente no recibe cambios para ese individuo.

X. Como ejecutarlo

Antes de empezar a testear se debe ejecutar el comando *make all* desde el directorio principal de la carpeta entregada. Luego si podemos ponernos a evaluar el algoritmo. Los tipos de instancias provistas son: C1, C2, HC1, HC2, HR1, HR2, HRC1, HRC2, R1, R2, RC1, RC2. Para cada una proporcionamos tres ejemplos. Además, como se especifica en la letra, cada instancia está asociada a cierta información sobre los vehículos y clientes. Proveemos configuraciones para cada una de las instancias ubicada en los archivos contenidos en la carpeta rep/GA/data/instances con el nombre de cada instancia.

Si se quiere ejecutar una de estas, es cuestión de ir a la careta rep/GA y ejecutar el comando:
./MainSeq newGA.cfg data/instances/<instance_name>.txt res/<instance_name>.txt

Por ejemplo, el siguiente comando ejecuta el algoritmo para la instancia C101:
./MainSeq newGA.cfg data/instances/C101.txt res/C101.txt

En caso de querer probar con otra instancia, hay que subir un csv con el nombre del ejemplo, luego subir un archivo .txt también con el nombre del ejemplo a la carpeta *rep/GA/data/instances*, y en este escribir en cuatro lineas: cantidad de clientes, cantidad de vehiculos, nombre del ejemplo (el recién creado, sin la extensión .csv), nombre de la instancia con la información de los vehiculos (ubicadas en *rep/GA/data/vehiculos*) .

La salida se guarda en un archivo con el mismo nombre de la instancia y queda guardada en la carpeta *rep/GA/res/*. También proveemos un script en bash para ejecutar varias instancias en bulk, este lo que hace es lanzar un thread para cada ejemplo que se quiera testear, esta ubicado en la carpeta *rep/GA/* con el nombre *script.sh*.

En el archivo newGA.cfg se puede cambiar la cantidad de ejecuciones, de generaciones, de individuos y la probabilidad de cruzamiento y mutación además de otras variables. El archivo newGA.cfg conserva el formato que provee la

librería, lo único que modificamos para la entrega fueron las generaciones, la cantidad de individuos y las independent runs y, dejándolas en 1000, 100 y 10 respectivamente.
El archivo *environment* deberá ser adaptado a cada configuración. En el archivo entregado, en la carpeta *rep/GA/res/*, incluimos los resultados de un ejemplo por instancia, los resultados de los comandos que están por defecto en el script.
A modo informativo, se quitaron las implementaciones de los algoritmos SA y CHC de la carpeta original povista por la libreria Malva.

XI. Referencias

[1]: MapColouring:
https://github.com/hpc-cecal-uy/cursoAE/blob/master/MapColouring/malva/src/netstream.cc
[2]: Malva Project https://github.com/themalvaproject/malva
[3] : Repo de la implementación.
https://gitlab.fing.edu.uy/franco.donnangelo/practicoae