Implementación de IVNS bajo el paradigma de POO

Autores:

- Victor Manuel Cardentey Fundora C511
- Amanda González Borrell C511

Introducción

El Problema de Enrutamiento de Vehículos (VRP) es un problema de optimización combinatoria que consiste en planificar recorridos que permitan satisfacer las demandas de un conjunto de clientes geográficamente dispersos. Para ello se cuenta con una flota de vehículos que parten desde uno o varios depósitos centrales. El objetivo de VRP es diseñar rutas para cada vehículo que minimicen o maximicen algún objetivo. Existen múltiples variantes del VRP aunque este proyecto se centra en el Problema de Enrutamiento de Vehículos con Capacidades (CVRP) en el cual cada vehículo tiene una capacidad de carga limitada.

Estos problemas al ser NP-duros no permiten la utilización de métodos exactos para instancias de grandes dimensiones, por lo que se utilizan algoritmos de aproximación basados en heurísticas y metaheurísticas para solucionarlos. En 2017 se define el algoritmo Búsqueda de Vecindad Infinitamente Variable (IVNS) el cual es una extensión del ampliamente utilizado algoritmo Búsqueda de Vecindad Variable (VNS) [2].

Estos algoritmos forman parte de los denominados algoritmos de búsqueda local en los cuales se define una solución actual, una estructura de entorno y de manera iterativa se generan soluciones vecinas de la solución actual utilizando la estructura de entorno (criterio de vecindad). El VNS realiza este procedimiento utilizando un conjunto finito de estructuras de entorno y el IVNS extiende este proceso utilizando una gramática que permite generar infinitas estructuras de entorno para ser utilizadas en la búsqueda. La implementación de IVNS descrita utiliza técnicas de compilación para generar código ejecutable para cada estructura de entorno, en este trabajo proponemos un enfoque de implementación utilizando programación orientada a objetos que permite evitar la generación de código definiendo una estructura genérica capaz de implementar todas las posibles estructuras de entorno generadas por la gramática.

Algoritmo IVNS

El algoritmo de IVNS se define de la siguiente forma:

```
V(x) = GenerarEstructuraDeEntorno()
s = GenerarSoluciónInicial()
while not CondicionesDeParada do:
    Agitación: Explorar V(s) y obtener s'
    if x' < x then:
        x = x'
    else:
        V(x) = GenerarEstructuraDeEntorno()
```

Generación de criterios de vecindad

La característica propia de IVNS es la generación de estructuras de entorno creadas por las instrucciones `V(x) = GenerarEstructuraDeEntorno()`. Para generar infinitas estructuras nos basamos en la gramática definida en [2] para implementar una jerarquía de clases que representan los símbolos de la gramática.

Primeramente definimos los elementos generales de la gramática:

```python
class Symbol:
    def __init__(self, parent) -> None:
        self.parent = parent
        self.is_terminal = False
        self.is_non_terminal = False

class TerminalSymbol(Symbol):
    def __init__(self, value) -> None:
        self.is_terminal = True
        self.value = value

class NonTerminalSymbol(Symbol):
    def __init__(self) -> None:
        self.is_non_terminal = True
        self.children = []
        self.productions = None

    def add_child(self, child):
        child.parent = self
        self.children.append(child)
```

Los símbolos terminales solo tienen un valor que se corresponden a una letra la cual representa una de las posibles operaciones para modificar una solución, estas son:

- r: seleccionar una ruta.
- a: seleccionar un cliente dada una ruta previamente seleccionada.
- b: insertar un cliente seleccionado en una ruta seleccionada.
- c: intercambiar dos clientes previamente seleccionados.
- f_{ri}: filtros para la operación r
- f_{ai}: filtros para la operación a
- f_{bi}: filtros para la operación b
- f_{oi}: filtros de optimización (operaciones b y c)

```python
# src/symbols.py
# Ejemplo de implementación de un Símbolo Terminal

class aSymbol(TerminalSymbol):
    def __init__(self) -> None:
        super().__init__('a')
```

Los símbolos no terminales almacenan sus hijos en el Árbol de Sintaxis Abstracta (AST) y las posibles producciones de ese símbolo.

```python
# src/symbols.py
# Ejemplo de implementación de un Símbolo No Terminal

class SSymbol(NonTerminalSymbol):
    def __init__(self) -> None:
```

```
        super().__init__()
        self.productions = [self.p1, self.p2, self.p3, self.p4]

    def p1(self):
        return [RSymbol(), ASymbol(), S1Symbol(), BSymbol()]

    def p2(self):
        return [RSymbol(), ASymbol(), S1Symbol(), RSymbol(), BSymbol()]

    def p3(self):
        return [RSymbol(), ASymbol(), ASymbol(), S1Symbol(), CSymbol()]

    def p4(self):
        return [RSymbol(), ASymbol(), RSymbol(), ASymbol(), S1Symbol(),
CSymbol()]

```

Definimos las producciones como funciones y almacenando estas como delegados podemos utilizarlas iterando por el arreglos `productions`

Teniendo definida la gramática lo siguiente sería generar cadenas pertenecientes a la gramática, esto se puede hacer con un algoritmo muy básico.

```
Expande(s):
    if s es terminal:
        return
    else:
        p = seleccionar un elemento de Producciones(s) de acuerdo a una
distribución
        c = aplicar p para obtener los nodos hijos de s
        foreach s' in c:
            Expande(s')
```

Una hipótesis es que este algoritmo de expansión pudiese ser mejorado para generar criterios que sean más probables de generar buenas soluciones utilizando una muestra de criterios que obtenga buenos resultados y hallar la distribución de las producciones en dicha muestra. Otra mejora es definiendo funciones de costo para las producciones para definir restricciones sobre que producciones son utilizadas.

Este algoritmo fue implementado utilizando un patrón `visitor` que se encarga de visitar cada nodo y expandir recursivamente hasta llegar a un terminal, una consideración importante a tener en cuenta es la generación de un ciclo infinito ya que la gramática es recursiva por lo que incorporamos un parámetro `h` el cual es la profundidad máxima de generación recursiva, la distribución utilizada fue uniforme por tanto toda producción tiene la misma probabilidad de ser aplicada.

```
# src/rexpand.py

class RExpandVisitor:

    @visitor.on('symbol')
    def visit(self, symbol, h):
        pass

    @visitor.when(TerminalSymbol)
    def visit(self, symbol: TerminalSymbol, h):
```

```
        pass

    @visitor.when(NonTerminalSymbol)
    def visit(self, symbol: NonTerminalSymbol, h):
        production = random.choice(symbol.productions)
        for s in production():
            symbol.add_child(s)
            self.visit(s, h)

    @visitor.when(S1Symbol)
    def visit(self, symbol: S1Symbol, h):
        if h > 0:
            production = random.choice(symbol.productions)
        else:
            production = symbol.p3

        for s in production():
            symbol.add_child(s)
            self.visit(s, h-1)
```

Una vez ejecutada la expansión tenemos un criterio de vecindad válido.

Ejecución de criterios

Para la ejecución de los criterios definimos una clase llamada `Solver`, `Solver` recibe como parámetros estructuras de datos que definen el problema.

```
#src/solver.py
class Solver:
    def __init__(self, clients_loc, clients_demand, routes, max_capacity, depot) -> None:
        pass
```

- `clients_loc`: es un diccionario con las coordenadas de los clientes, este diccionario se puede transformar en una matriz de distancias para realizar el cómputo de una forma más eficiente y para estandarizar la implementación para distintos problemas donde existan modificaciones para calcular la distancia. En este caso utilizamos la distancia euclidiana.
- `clients_loc`: es un diccionario con las demandas de los clientes.
- `routes`: una solución inicial del problema. Para este caso utilizamos una solución golosa para encontrar una solución viable, siempre poniendo un cliente en la ruta de más capacidad.
- `max_capacity`: la capacidad máxima de los vehículos.
- `depot`: las coordenadas del depósito de vehículos.

Luego realizamos distintas implementaciones de las operaciones definidas por la gramática, a continuación brindamos un ejemplo para la operación de seleccionar una ruta.

```
# src/solver.py
# Route selection operation and filters

    def select_route_random(self):
        self.current_route = random.choice(list(self.routes.keys()))

    def select_route_not_empty(self):
```

```
        self.current_route = random.choice(list(rid for rid in self.routes
if self.routes[rid]))

    def select_route_min_cost(self):
        self.current_route = sorted(self.routes_cost.items(), key=lambda x:
x[1])[0][0]

    def select_route_max_cost(self):
        self.current_route = sorted(self.routes_cost.items(), key=lambda x:
x[1], reverse=True)[0][0]

    def select_route_min_capacity(self):
        self.current_route = sorted(self.routes_capacity.items(),
key=lambda x: x[1])[0][0]

    def select_route_max_capacity(self):
        self.current_route = sorted(self.routes_capacity.items(),
key=lambda x: x[1], reverse=True)[0][0]

```

Debido a que todas las operaciones se definen como métodos dentro del cuerpo de la clase la definición de un criterio es simplemente una secuencia de llamadas a estos métodos.

Por tanto la clase utiliza una lista para almacenar la secuencia de métodos a llamar en el criterio que está explorando.

```
# src/solver.py

class Solver:
    def __init__(self, clients_loc, clients_demand, routes, max_capacity,
depot) -> None:
        # some other dark code

        self.operations = []

    def register_operation(self, op):
        self.operations.append(op)

    def reset(self):
        self.operations = []
```

Entonces podemos utilizar el método `register_operation` para crear el criterio a nuestra conveniencia, de nuevo utilizando un patrón `visitor` recorremos el AST y utilizamos `register_operation` para construir el criterio, a continuación mostramos su uso para construir el filtro de selección de rutas.

```
# src/build_criteria.py

    @visitor.when(frSymbol)
    def visit(self, symbol : frSymbol):
        if symbol.filter == 1:

 self.solver.register_operation(self.solver.select_route_min_cost)
        elif symbol.filter == 2:

 self.solver.register_operation(self.solver.select_route_max_cost)
```

```
        elif symbol.filter == 3:
            self.solver.register_operation(self.solver.select_route_min_capacity)
        elif symbol.filter == 4:
            self.solver.register_operation(self.solver.select_route_max_capacity)
        elif symbol.filter == 5:
            self.solver.register_operation(self.solver.select_route_not_empty)
```

Una vez terminamos de construir el criterio solo queda recorrer la lista `operations` la cual contiene métodos e invocarlos uno a uno, esta es la estrategia más básica de exploración de la vecindad ya que solo genera una aleatoria.

Resultados y consideraciones

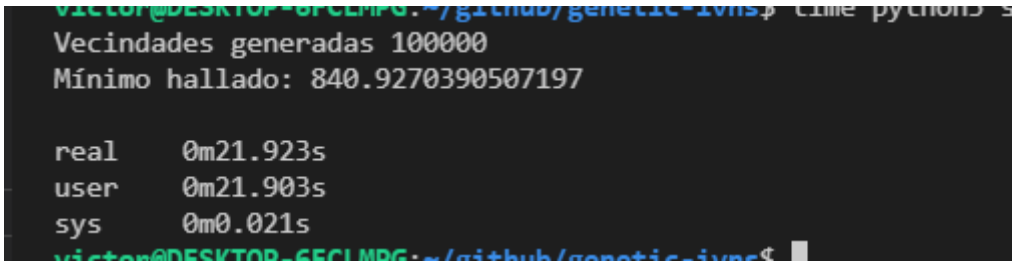
El mejor resultado obtenido fue de 795.3896893727342 luego de generar 100000 soluciones.

Entre los factores que consideramos influyen en este resultado están:

1. Utilización de exploración aleatoria como estrategia para explorar las vecindades.
2. Generación aleatoria de los criterios.

El punto 1 se puede solucionar con la implementación de nuevos métodos de exploración dentro de la clase `Solver` y el punto 2 bien se pudiese utilizar algunas de las estrategias sugeridas en la sección de Generación de Criterios.

Esta implementación es una prueba de concepto y no es una implementación optimizada por lo que a continuación planteamos ciertas líneas generales para una mejor implementación.



1. Utilizar un lenguaje de alto rendimiento orientado a objetos como C++ u Objective-C dado que Python es un lenguaje poco eficiente.
2. Realizar un estudio de las distintas variaciones de VRP para definir una interfaz y jerarquía de clases entre los `Solvers` de los distintos problemas ya que mediante herencia se pueden reutilizar soluciones a otros problemas.
3. Utilizar estructuras de datos para filtrar dinámicamente, por ejemplo mantener el costo y capacidad de las rutas en un heap para poder utilizar los filtros en $\mathrm{O}(\log n)$

Entre las ventajas de este enfoque para la implementación podemos destacar que:

- Evita generación de código, el cual es un proceso que implica generar el código y luego compilarlo para ejecutarlo haciendo la ejecución del algoritmo más lenta.
- Un generador de código es más difícil de mantener y debuggear que una solución sin generación.
- Nuestra solución permite un aprovechamiento al máximo de las características de la POO para reutilizar código entre distintas clases de VRP

Bibliografía

1. J. H. Dantzig, G. B.; Ramser. The truck dispatching problem. Management Science, 6, 10 1959.
2. Camila Pérez Mosquera. Primeras aproximaciones a la Búsqueda de Vecindad Infinitamente Variable. Trabajo de diploma. MATCOM, Universidad de La Habana. 2017.