Resumen

El problema de enrutamiento de vehículos (VRP) posee gran importancia académica e industrial. La cantidad de variantes que pueden existir es virtualmente infinita, limitada sólo por la imaginación humana. El tiempo necesario para resolver una de estas variantes es, comunmente, entre seis meses y dos años; modificado por las especificaciones involucradas.
Se propone una herramienta para resolver cualquier variante de VRP en un estimado de dos semanas utilizando algoritmos de búsqueda local. Se unen las implementaciones del grafo de evaluación, el árbol de vecindad con generador de soluciones y la generación automática de estrategias de exploración y selección para resolver los problemas de forma eficiente.

Introducción

Los costos por transportación pueden representar más de la mitad de los costos logísticos de una empresa [11]. Esto implica que para empresas con capitales millonarios, una buena planeación de sus rutas de transportación puede representar un ahorro igualmente millonario.

El problema de enrutamiento de vehículos (VRP por sus siglas en inglés) es un problema de optimización combinatoria cuyo objetivo en su forma más simple es, dado un conjunto de clientes, un depósito y una flota de vehículos, encontrar una asignación de rutas que optimice ciertos criterios tales como tiempo y costos de transportación.

En la literatura se reportan múltiples variantes para este problema como el Problema de Enrutamiento de Vehículos con restricciones de capacidad (CVRP) [6], el Problema de Enrutamiento de Vehículos con ventanas de tiempo y el Problema de Enrutamiento de Vehículos con recogida y entrega (VRPPD) [15], por citar sólo algunos de las más estudiadas. Cada variante tiene restricciones propias, por lo que resulta difícil la creación de un método de solución universal para cualquier VRP.

Estos problemas son NP-duros, esto implica que no existen soluciones exactas para conjuntos de datos reales y por tanto, se utilizan técnicas aproximadas como heurísticas y metaheurísticas que han sido objeto de estudio por décadas [5, 3, 8, 14, 15].

En particular, los algoritmos de búsqueda local han mostrado buenos resultados en los últimos tiempos [**?**].

El proceso de solución de un Problema de Enrutamiento de Vehículos específico es complejo y su solución puede tomar tiempo. Por ejemplo, varios meses en el caso de un trabajo de diploma [**?**], hasta un año en caso de una maestría [1] y varios años en caso de una tesis de doctorado [2].

Desde hace varios años, comenzando por [9], en la facultad de Matemática y Computación de La Universidad de La Habana se han desarrollado varias investigaciones con el objetivo de agilizar el tiempo que toma resolver una variante específica de VRP. Para ello se propone utilizar una

metaheurística de búsqueda local (IVNS) que permite considerar infinitos criterios de vecindad para un problema, a partir de gramáticas libres del contexto. Luego, en [4] se propone una forma de obtener automáticamente, a partir de la descripción del problema, la gramática necesitada por IVNS. En [13] se propone una manera de calcular el costo de un vecino en casi cualquier variante de VRP y para ello se utiliza un Grafo de Evaluación. En [12] se plantean formas automáticas para explorar vecindades de distintas maneras mediante un Árbol de Vecindad. Finalmente, en [7] se propone una vía para combinar estrategias de exploración y selección sin tener que programar todas las combinaciones posibles a través de un generador de funciones de exploración. Cada una de estas herramientas resuelve un problema por separado, pero hasta ahora no era posible su integración. Si se lograran integrar sería posible resolver cualquier problema de VRP utilizando cualquier combinación de estrategias de exploración y criterios de vecindad a partir de [12] y [12]. Además usando infinitos criterios de vecindad al combinar lo propuesto en [9] y [4]. La única restricción estaría en que los vecinos fueran calculables por lo propuesto en [13].

En este trabajo se propone una herramienta que permite integrar el Grafo de Evaluación de [13], con el Árbol de Vecindad de [12] y el generador de funciones de exploración de [7]. Por ahora no es objetivo integrar el sistema con IVNS y por tanto se proponen otras metaheurísticas de búsqueda local como VNS [**?**].

Con el sistema propuesto, resolver una variante de VRP se traduce en definir cómo se evalúa una única solución y definir, en un lenguaje casi natural, qué estrategias de exploración y criterios de vecindad se quieren utilizar.

Objetivos

El objetivo general de este trabajo es diseñar e implementar un sistema que permita calcular los costos de cualquier vecino usando el Grafo de Evaluación de [13], que las vecindades se exploren usando el Árbol de Vecindad de [12], que las estrategias de exploración y selección se puedan combinar con lo propuesto en [7] y, por tanto, resolver variantes de VRP a partir únicamente del código de evaluación de una solución.

Para lograr los objetivos generales se han trazado los siguientes objetivos específicos:

1. Consultar literatura especializada sobre el estado del arte de los problemas VRP.

2. Estudiar las ideas y códigos relacionados con el Grafo de Evaluación [13], el Árbol de Vecindad [7], y el generador de funciones de exploración [12].

3. Diseñar e implementar un sistema que combine estas ideas para resolver distintas variantes de VRP a partir de la evaluación de una solución.

4. Analizar los resultados obtenidos. Para ello se utiliza el sistema implementado para resolver, con muy poco esfuerzo humano variantes de VRP.

El presente documento está organizado en 5 capítulos.

TODO: En el capítulo 2 se describen los elementos fundamentales del Problemas de Enrutamiento de Vehículos. En el capítulo 2 se describen las de las tesis que se combinan en este trabajo [13, 7, 12]. El capítulo 3 describe el sistema que se implementó. En el capítulo 4 se describe el método de uso el sistema para describir y resolver variantes de VRP. En el capítulo 5 se muestra cómo extender este sistema. En el capítulo 6 se exponen los resultados alcanzados. En el capítulo 6 se ofrecen las conclusiones y se brindan algunas recomendaciones para trabajos futuros.

Capítulo 1

Problemas de Enrutamiento de Vehículos y métodos de búsqueda local

En este trabajo se propone un sistema que permite resolver de manera Ágil cualquier de VRP utilizando algoritmos de búsqueda local. En este capítulo se presentan los elementos fundamentales de VRP y de las metaheurísticas de búsqueda local.

En la sección 1.1 se describen los Problemas de Enrutamiento de Vehículos. En la sección 1.2 se describen los algoritmos de búsqueda local. En 1.3 se presentan algunas características de Common lisp, lenguaje en que se implementó el sistema.

1.1. Problema de Enrutamiento de Vehículos

La primera referencia al VRP es [6] y data de 1959. En este trabajo se describe una aplicación real de VRP, entrega de gasolina a varias estaciones de servicio. Para resolverlo, se propone un modelo matemático y una heurística para aproximar la solución.

La versión más sencilla recibe el nombre de CVRP (Capacitated Vehicle Routing Problem) [6] consta de una flota de vehículos que salen de un depósito y deben satisfacer las demandas de una serie de clientes. El objetivo es encontrar una distribución de rutas que se asignan a los vehículos de forma que se optimice determinada métrica (tiempo, combustible, costo total, etc). A partir de esta versión inicial se estudian diversas variantes que

pueden ser encontradas en la literatura:

- VRPTW - VRP con ventanas de tiempo. Cada cliente posee un período de tiempo fijo durante el cual puede ser visitado [**?**].
- VRPPD - VRP con recogida y entrega. Los bienes deben ser entregados y recogidos en cantidades fijas [**?**].
- MDVRP - VRP con múltiples depósitos. Se cuenta con múltiples depósitos desde los que pueden salir los vehículos [**?**].

Esta es una familia de problemas NP-Duros[**?**], por lo las soluciones exactas no son factibles para problemas reales. Para buscar aproximaciones a la solución se utilizan heurísticas y metaheurísticas. En la literatura se reporta [**?**] que las metaheurísticas de búsqueda local brindan buenas soluciones en poco tiempo y por eso esta es la estrategia seleccionada para este trabajo.

1.2. Metaheurísticas de Búsqueda Local

Los algoritmos basados en búsqeuda local parten de una solución a la que se llama solución actual y exploran un conjunto de soluciones a las que se les llama vecinos. Las soluciones vecinas se obtienen al aplicar sobre la solución actual un determinado criterio (llamado criterio de vecindad). Cada vecino se analiza y de ellos se escoge uno que puede sustituir o no a la solución actual en dependencia del algoritmo que se esté usando.

En dependencia de factores como cuántos criterios de vecindad se tiene y cuándo se acepta o no sustituir la solución actual, se tienen distintos algoritmos.

Para aplicar cualquier algoritmo de búsqueda local es necesario definir qué es una solución y de esto se trata en la siguiente sección.

1.2.1. Representación de soluciones del VRP

Una solución de un Problema de Enrutamiento de Vehículos describe qué clientes visita cada vehículo y en qué orden. Al recorrido de cada vehículo se le llama ruta. Usualmente las rutas se pueden representar por una lista de clientes y las soluciones como una lista de rutas. Por ejemplo, si se define a P_1 como un problema clásico (VRP en su versión mas simple) que consta de 6 clientes: $[c_1, c_2, c_3, c_4, c_5, c_6]$, entonces una solución s_1 se puede definir como:

$$s_1 = [(c_2, c_3), (c_1, c_4, c_5), (c_6)] \tag{1.1}$$

En s_1 se muestra una solución con tres rutas. El vehículo perteneciente a la primera (r_1) ruta visita a c_2 y c_3, el vehículo de la segunda ruta (r_2) visita a c_1, c_4 y c_5 y el de la tercera (r_3) sólo visita a c_6. En todos los casos los recorridos comienzan y terminan en el depósito.

Una vez definida una solución, es posible definir un criterio de vecindad.

1.2.2. Criterios de vecindad

Un criterio de vecindad es una regla que permite, a partir de una solución actual, obtener nuevas soluciones. En el caso de VRP, estos criterios se pueden formar con modificaciones que se aplica a los elementos de la solución actual. Algunos de estos criterios de vecindad [9][4] pueden ser:

1. Intercambiar dos clientes seleccionados aleatoriamente.
2. Reubicar un cliente dentro de su ruta en la posición que minimice el costo de esa ruta.
3. Mover aleatoriamente un cliente de una ruta a otra.
4. Intercambiar dos vehículos seleccionados aleatoriamente.

Los criterios de vecindad dependen también de la variante del problema sobre la que se trabaje. Por ejemplo, el criterio de *Cambiar vehículos* no tiene sentido para el problema P_1 pues en este todos los vehículos son iguales.

Los criterios de vecindad se pueden describir a través de un conjunto de operaciones llamadas operaciones elementales. Algunas de estas operaciones elementales son:

1. Seleccionar una ruta **r**.
2. seleccionar un cliente **c** de una posición **p** de una ruta **r**.
3. insertar un cliente **c** en una posición **p** de una ruta **r**.

Las operaciones *Seleccionar ruta*, *seleccionar cliente* e *insertar cliente* se representan en este trabajo con los símbolos *r*, *a* y *b* respectivamente.

Se tiene como ejemplo el criterio *Mover un cliente a cualquier ruta, en cualquier posición*. Este puede ser representado por las operaciones:

1. Seleccionar ruta r_x.
2. Seleccionar cliente c_x de la posición p_x de la ruta r_x.
3. Seleccionar ruta r_y.
4. Insertar cliente c_x en la ruta r_y en la posición p_y.

A partir de un criterio de vecindad y una solución inicial se pueden generar nuevas soluciones vecinas. Un vecino se obtiene dándole valores específicos a los elementos que forman cada una de las operaciones del criterio. Por ejemplo, a partir de la solución: $s_1 = [(c_2, c_3), (c_1, c_4, c_5), (c_6)]$, se puede obtener un vecino si se usa el criterio (*Mover un cliente a cualquier ruta, en cualquier posición*) con los siguientes valores:

1. $r_x = 1$.
2. $c_x = c2$.
3. $p_x = 1$.
4. $r_y = 2$.
5. $p_y = 3$.

Entonces se obtiene la solución vecina

$$s_2 = [(c_3), (c_1, c_4, c_2, c_5), (c_6)]$$

.

Existen varias formas de explorar una vecindad y de seleccionar una solución dentro de una vecindad. en la siguiente sección se habla de estas estrategias.

1.2.3. Estrategias de exploración y selección

En ocasiones no se exploran las vecindades completas. [2][7]. A las formas de explorar una vecindad se le llama estrategia de exploració. Dos formas de explorar vecindades son:

- Exploración exhaustiva: Se analizan todos los vecinos que puedan ser generados.

- Exploración aleatoria: Se genera una cantidad fija de vecinos menor que la cardinalidad de la vecindad. La decisión de qué vecinos generar es aleatoria.

Además de decidir qué vecinos explorar, también es necesario decidir cuál retornar entre aquellos que mejoran la solución. A esta decisión se le denomina estrategia de selección. Se tiene como ejemplos:

- Mejor vecino: Se retorna al mejor vecino entre todos aquellos analizados.
- Primera mejora: Tan pronto se encuentre un vecino con mejor evaluación que la solución actual, se detiene la búsqueda y se devuelve el vecino encontrado.
- Mejora aleatoria: Se retorna un vecino aleatorio entre todos aquellos mejores que la solución inicial.

A partir de distintas combinaciones de estrategias de exploración y selección es posible realizar exploraciones distintas que obtengan diferentes resultados.

Una vez definida una solución del problema, los criterios de vecindad, una estrategia de exploración y una de selección, es posible implementar una metaheurística de búsqueda local. En la siguiente sección se presenta una de ellas, Búsqueda de Vecindad Variable.

1.2.4. Búsqueda de Vecindad Variable

Búsqueda de Vecindad Variable es una metaheurística que se basa en el cambio sistemático de vecindades dentro de una búsqueda local [**?**]. Las primeras referencias que se tienen sobre esta metaheurística son: [**?**] de 1995 y [**?**] en 1997 donde usan VNS para dar solución al problema del viajante considerando la existencia o no de backhauls, (esto significa que los vehículos primero entregan las mercancías a los clientes y después recogen lo que los clientes deben devolver).

VNS está basada en dos principios básicos [9]:

- Un mínimo local de una vecindad no lo es necesariamente de otra.
- Un mínimo global es un mínimo local de todas las vecindades.

Los principios anteriores sugieren el empleo de vecindades en las búsquedas locales para abordar un problema de optimización, pues si se encuentra un óptimo con todas las vecindades, entonces se podría afirmar que es el óptimo global del problema (al menos con los criterios de vecindad que se usaron) y esta es precisamente la idea de VNS [9].

Para implementar el sistema de este trabajo se utilizó Common Lisp. En la siguiente sección se presenta este lenguaje y se justifica su elección.

1.3. Common Lisp

Common Lisp es un lenguaje de programación multi-paradigma (soporta una combinación de paradigmas de programación tales como la programación imperativa, funcional y orientada a objetos). Facilita el desarrollo de software evolutivo e incremental, con la compilación iterativa de programas eficientes en tiempo de ejecución. [**?**]

El sistema de objetos de Common Lisp (CLOS) contiene herencia múltiple. Esto le permite a las clases heredar directamente de una o más clases superiores. Utilizando herencia múltiple se puede definir elementos de VRP describiendo sus características como clases que tienen diferentes propiedades [**?**].

Por ejemplo, se pueden definir las clases siguientes:

```
1 ( defclass has-id ()
2   ((has-id)))
3
4 ( defclass has-demand ()
5   ((demand)))
6
7 ( defclass basic-client (has-id)())
8
9 ( defclass cvrp-client (has-id has-demand)())
```

La clase **has-id** tiene una propiedad **id** para representar un objeto que tiene un identificador. La clase **has-capacity** con una propiedad **demand** para representar que el objeto tiene una demanda. Luego, se definen dos clases diferentes para representar clientes: **basic-client** (cliente básico) y **cvrp-client** (cliente de un Problema de Enrutamiento de Vehículos con restricciones de capacidad). El cliente básico sólo hereda de **has-id** para tener

un identificador, mientras que el cliente de CVRP hereda tanto de **has-id** como de **has-demand** para, además de un identificador, contar con una demanda.

Las funciones genéricas definen una operación abstracta y su ejecución depende de los tipos de sus argumentos. El comportamiento de una función genérica se define mediante un conjunto de métodos en los que se puede especificar el tipo de los argumentos que se recibe [**?**].

De acuerdo a los tipos de los argumentos que se le pase a la función genérica se decide cuál de los métodos ejecutar. Si existe un método definido cuyos argumentos sean exactamente del mismo tipo que de los que se le pasó a la función genérica en ese llamado, ese es el método que se ejecuta. En caso contrario, se determina en cual de los métodos existentes los argumentos son lo más parecido a los que se le pasaron a la función genérica en ese llamado. En Common Lisp se define lo *más parecido posible* ordenando los métodos de más específico a menos específico según el conjunto de argumentos que reciben [**?**].

Por ejemplo, se tiene la siguiente función genérica con dos métodos:

```
(defgeneric visit (c1 c2))

(defmethod visit ((c1 basic-client) (c2 basic-client))
   (format t "visited clients ~a, and ~a" (id c1) (id c2)))

(defmethod visit ((c1 basic-client) (c2 cvrp-client))
   (format t "visited clients ~a, and ~a with demand of ~a" (id c1)
       (id c2) (demand c2)))
```

Además se definen los siguientes parámetros:

```
(defparameter basic-1 (make-instance 'basic-client
                           :id "basic-1"))

(defparameter basic-2 (make-instance 'basic-client
                           :id "basic-1"))

(defparameter cvrp-1 ( make-instance 'cvrp-client
                           :demand 5))
```

Al ejecutar la función **visit** con los parámetros **basic-1** y **basic-2** se obtiene la salida:

```
visited clients basic-1, and basic-2
```

Luego, al ejecutar **visit** con los parámetros **basic-1** y **cvrp-1** se obtiene la salida:

```
visited clients basic-1, and cvrp-1, with demand of 5
```

En lisp cada vez que se define un método con un conjunto de argumentos se le denomina método *primario*. También se tienen métodos auxiliares . Por ejemplo, se tiene los métodos *after* que se ejecutan después del método primario y *before* que se ejecutan antes del método primario. Como ejemplo se define el siguiente método *after*:

```
(defmethod visit :after ((c1 basic-client) (c2 :has-demand))
   (format t "~%I repeat, ~a has a demand of ~a" (id c2) (demand
       c2)))
```

Al ejecutar la función **visit** con los parámetros **basic-1** y **cvrp-1**, esta vez se obtiene la salida:

```
visited clients basic-1, and cvrp-1, with demand of 5
I repeat, cvrp-1 has demand of 5
```

En este capítulo se explicaron las características de los Problemas de Enrutamiento de Vehhículos y los algoritmos de Búsqueda local, junto con algunas ventajas que brinda Common Lisp que se utilizan en la implementación del sistema. El siguiente capítulo muestra algunas características de las herramientas que se utilizaron en este trabajo para automatizar la solución de variantes de VRP. Estas herramientas son son el Grafo de evaluación

[13], el Árbl de Vecindad [12] y el generador de funciones de exploración [7].

Capítulo 2

Herramientas para la automatización de la solución de VRP

Desde el año 2016 [9] en la facultad de Matemática y Computación de la Universidad de La habana se han desarrollado varias tesis que permiten agilizar la solución de un Problema de Enrutamiento de Vehículos. En particular, se desarrollaron tres herramientas que son relevantes para este trabajo y en este capítulo se presentan los elementos fundamentales de cada una de estas herramientas. En la sección 2.1 se explica el Árbol de Vecindad que permite explorar una vecindad con distintas estrategias, en la sección 2.2 se explica el grafo de evaluación que permite obtener el costo de un vecino cualquiera y en la sección 2.3 se explica el combinador de estrategias de exploración y selección. Se comienza por la exploración automática de vecindades.

2.1. Árbol de vecindad

Los algoritmos de búsqueda local parten de una solución y y exploran un conjunto de soluciones que se denominan vecinos. Existen distintas formas de efectuar esta exploración (exhaustiva, aleatoria, etc). En [12] se propone un mecanismo que permite explorar una vecindad de cualquier manera posible. Para ello se le asigna un número a cada solución de la vecindad. De esta forma, se puede acceder a cualquier solución a partir de su número asociado. Esto se logra a través de un Árbol de Vecindad.

Un Árbol de vecindad es una estructura arbórea donde cada nodo representa una operación de un criterio de vecindad y permite determinar la cantidad de vecinos que tiene un criterio, así como establecer la biyección entre ese número de vecinos y los primeros números naturales.

Una vez se tiene una correspondencia entre los n elementos de una vecindad y nos n primeros números naturales, tener distintas maneras de explorar la vecindad se traduce en tener distintas maneras de explorar un conjunto de números.

El árbol de vecindad parte de una solución y un criterio de vecindad definido como se presentó en **??**. Su objetivo es, en primer lugar, determinar la cantidad de soluciones que tiene la vecindad sin tener que enumerarlas y, en segundo lugar, hacer la correspondencia entre cada solución y su índice.

A partir de las operaciones de un criterio y una solución se construye el Árbol de Vecindad donde cada nodo es una operación y los hijos de esos nodos describen los posibles valores que se le puede asignar a esas operaciones. Por ejemplo, dado el criterio **mover cliente a una posición aleatoria en una ruta aleatoria** formado por las operaciones:

- Seleccionar ruta r_x.
- Seleccionar cliente c_x de la posición p_x de la ruta r_x.
- Seleccionar ruta r_y.
- Insertar cliente c_x en la posición p_2 de la ruta r_2 .

Al aplicársele a la solución: $s_1 = [(c_2, c_3), (c_1, c_4, c_5), (c_6)]$, se obtiene el árbol representado en la figura 2.1.

Cada rama del árbolrepresenta un conjunto de soluciones vecinas. Por ejemplo, en la figura 2.1, la rama más a la izquierda representa todas las posibles soluciones en las que el cliente se escoge de la primera ruta y se inserta también en la primera ruta.

A partir del árbol que representa a una vecindad para una solución, es posible contar la cantidad de vecinos y es posible asignarle a cada vecino un número natural [12]. De esta manera, explorar una vecindad se reduce a encontrar una manera de seleccionar elementos de los n primeros números. En una exploración exhaustiva se seleccionan todos los números mientras que en una aleatoria se selecciona sólo un conjunto de ellos. Usando esta idea, las estrategias de exploración se definen como iteradores que generan un número, calculan cuál es la solución le corresponde y retornan esta solución en forma de lista de operaciones.

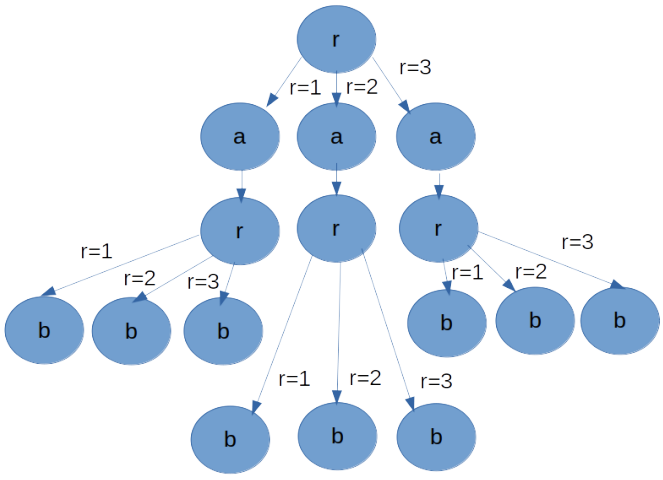

Figura 2.1: Árbol de vecindad asociado a *rarb*

El Árbol de Vecindad permite explorar vecinos. Para cada vecino se necesita conocer su costo y en la siguiente sección se explica cómo se obtienen estos costos mediante un Grafo de Evaluación.

2.2. Grafo de Evaluación

Para resolver un VRP utilizando búsqueda local, es necesario durante la exploración de las vecindades obtener el costo de cada vecino. Los algoritmos de búsqueda local calculan un vecino a partir de pequeñas modificaciones de la solución actual. Calcular el costo del vecino evaluando la solución en su totalidad suele ser menos eficiente que calcular la variación del costo debido a las operaciones. Por eso, al programar estos algoritmos, es conveniente determinar la manera más eficiente de evaluar vecinos con la menor cantidad de operaciones posibles.

Por ejemplo, en CVRP, al eliminar un cliente de una ruta sólo es necesario calcular la nueva distancia a partir de la eliminación, y modificar la carga que debe tener el vehículo a lo largo de la ruta. Algo similar ocurre cuando se inserta un cliente.

Si se desea obtener el costo de un vecino con la menor cantidad de operaciones posible, para cada problema y cada operación del criterio de ve-

cindad, se debe programar sus diferencias. Esto puede variar mucho de un problema a otro. Por tanto, calcular el costo de un vecino, para cada variante de VRP es un trabajo distinto y necesita una implementación diferente.

Por ejemplo, en CVRP sólo se debe actualizar la distancia recorrida y la capacidad restante del vehículo, pero en un problema con *backhaul* hay que verificar también si no se incumple la restricción de que todos los clientes de tipo A se visiten antes de los de tipo B.

Este es, precisamente, uno de los motivos por el que resolver un Problema de Enrutamiento de Vehículos requiere tiempo de desarrollo humano, al tener que programar cada una de las combinaciones posibles. En [13] se propone un mecanismo que permite calcular el costo de cualquier vecino, a partir, únicamente, del código que calcula el costo de la solución inicial del problema. Esto se hace a través de un Grafo de Evaluación que se expone a continuación.

El Grafo de Evaluación es una estructura que permite almacenar las operaciones realizadas durante la evaluación de una solución. en este grafo hay tres tipos de nodos:

- Nodos que representan los elementos de la solución (clientes, depósitos).
- Nodos que representan variables y mantienen sus valores (como el costo total). Estos se denominan nodos acumuladores.
- Nodos que representan las operaciones involucradas en una evaluación (aumentar valor, disminuir valor).

Para calcular el costo de un vecino a partir de las operaciones necesarias, se hace al grafo la menor cantidad de modificaciones posible para que este represente el estado de la solución luego de efectuadas las operaciones. El Grafo de Evaluación permite obtener costo de soluciones vecinas de una solución actual de forma eficiente, ya que garantiza que sólo se recalculen los fragmentos de las nuevas soluciones en los que estas se diferencien de la solución actual.

Cómo construir el grafo y cómo ejecutar operaciones sobre un grafo construido se explicará en las siguientes secciones.

2.2.1. Creación del grafo

El grafo que representa la solución inicial comienza con los nodos representativos de esta solución. Por ejemplo, en la figura 2.2 se muestran los

nodos que representan los elementos de la solución: $s = [(c_1, c_2), (c_3, c_4)]$. Los nodos d representan al depósito y los nodos (c_i) a cada cliente. Nótese que todas las rutas en el grafo comienzan y terminan con nodos depósito.

Luego, el usuario debe ejecutar el código de evaluación de la variante específica que se quiere resolver y, a partir de este código, se construyen los nodos acumuladores y los nodos operacionales. Por ejemplo, se muestra a continuación el código de evaluación de VRP básico:

```
1 def-var total-distance as 0
2 loop for r in solution-routes do
3   loop for c in r-clients do
4     increment total-distance with distance between
          r-previous-client and c
5   set previous-client as c
6 return-value total-distance graph
```

En la línea 1 se crea un nodo para mantener el valor del costo total y en la línea 4 se crean los nodos que aumentan el valor de la distancia total. Cómo programar códigos de evaluación se explicará detalladamente en la sección 3.3.

Al aplicar el código de evaluación al grafo recién inicializado que se representa en la figura 2.2, se obtiene el grafo representado en la figura 3.1. El nodo *cost* almacena el costo total de la solución y los nodos con símbolo de incremento son nodos operacionales que toman como entrada dos nodos clientes (o depósito) y adicionan al nodo de costo total la distancia entre ellos.

En la figura 2.4 se transforma el problema en CVRP utilizando la misma solución. En este caso se agregan los nodos *cap* que tienen como valor inicial la capacidad del vehículo perteneciente a cada ruta. Los nodos de decremento reciben un cliente como entrada y disminuyen la capacidad del vehículo en una cantidad igual a la demanda del cliente. Luego, los nodos *pen* reciben como entrada los nodos de capacidad y, en caso de tener estos valor negativo, penalizan el costo total de la solución.

2.2.2. Modificación del grafo

Los nodos operacionales reciben y modifican nodos representativos de elementos de la solución y nodos acumuladores. Estos nodos tienen asocia-

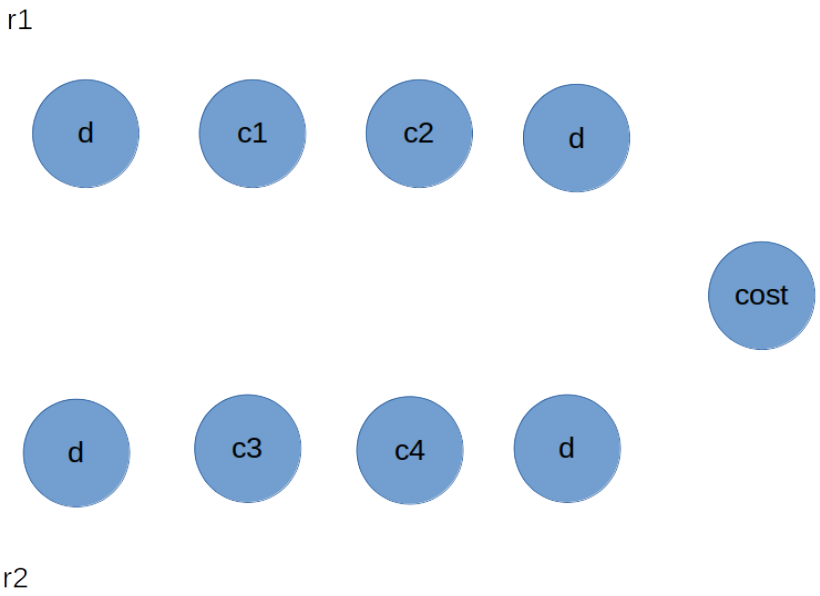

Figura 2.2: Nodos que representan elementos de la solución s_1 de VRP básico.

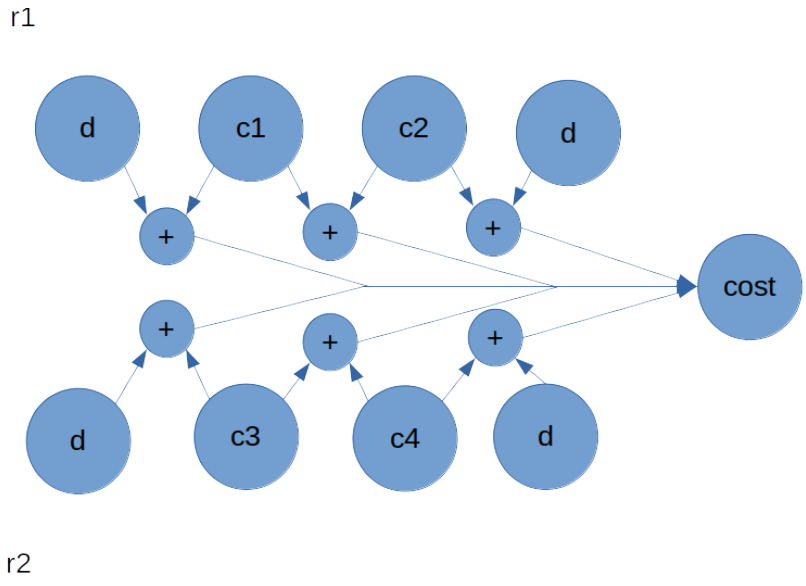

Figura 2.3: Grafo de evaluación que representa la solución $s1$ de VRP clásico.

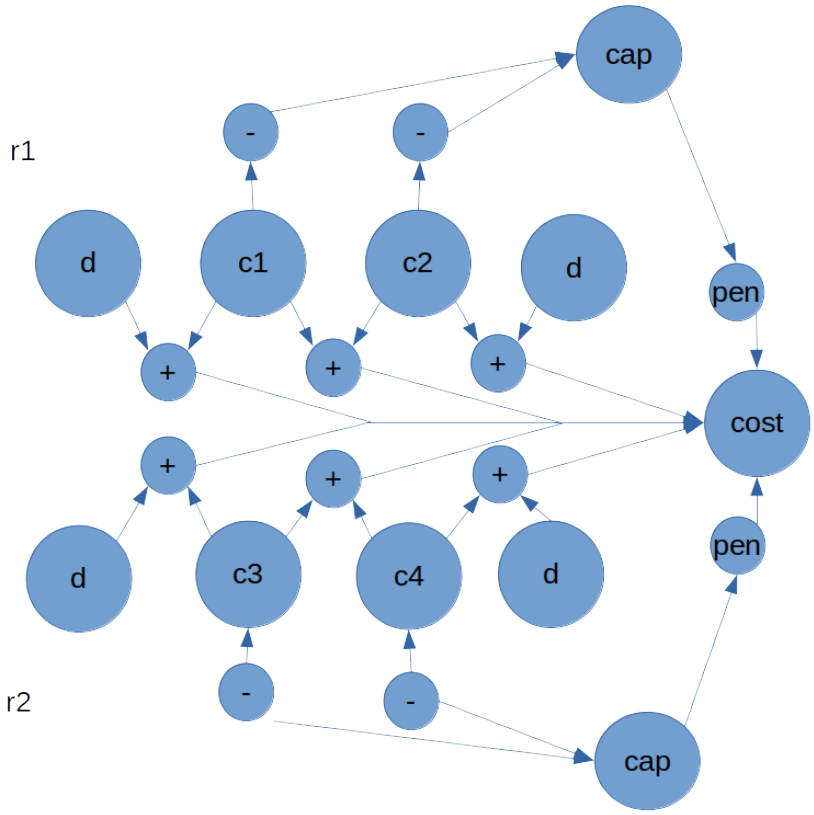

Figura 2.4: Grafo de evaluación que representa la solución $s1$ de VRP con restricciones de capacidad.

das funciones para ejecutar o deshacer sus operaciones. Esto se usa cuando se agregan o eliminan nodos que representan elementos de la solución que afecten a estos nodos operacionales.

Por ejemplo, retirar el cliente $c1$ del grafo representado en 3.1 (VRP clásico) provoca que los dos nodos de incremento que utilizan dicho cliente como entrada deshagan sus operaciones, y al mismo tiempo, se crea y ejecuta un nuevo nodo incremento que recibe como entrada d y $c2$. Luego, insertar a $c1$ al final de la ruta $r2$ se deshace la operación el nodo incremento que recibe como entrada a $c4$ y d mientras se crean y ejecutan dos nodos incrementos nuevos, uno recibiendo de entrada a $c4$ y $c1$ mientras que el otro a $c1$ y d. El resultado de estas dos operaciones se muestra en 3.2 y es, precisamente, el grafo resultante de aplicar al grafo de la figura 3.1, las siguientes operaciones del criterio **mover cliente a una posición aleatoria en una ruta aleatoria**:

- Seleccionar ruta **r1**.
- Seleccionar cliente **c1** de la posición **2** de la ruta **r1**.
- Seleccionar ruta **r2**.
- Insertar cliente **c1** en la posición textbf3 de la ruta **r2**.

Nótese que luego de aplicar los métodos **evaluate** y **undo** correspondientes el nodo *cost* tiene almacenado el costo de la solución resultante luego de aplicar una el criterio **mover cliente a una posición aleatoria en una ruta aleatoria**. Para encontrar el costo de la nueva solución sólo fue necesario analizar y modificar los nodos en que el grafo de la solución nueva se diferencia de la solución anterior y no todo el grafo. En esto se basa la evaluación .^{ef}iciente"del Grafo de Evaluación.

Para explorar una vecindad a partir de un criterio es necesario (además de generar y evaluar soluciones) decidir e implementar estrategias de exploración y de selección. A partir de combinaciones de diferentes estrategias se pueden realizar exploraciones con distintos resultados.

2.3. Combinación de estrategias de exploración y selección.

Las exploraciones de las vecindades obtienen diferentes resultados en dependencia de las estrategias de exploración y de selección que se utilice.

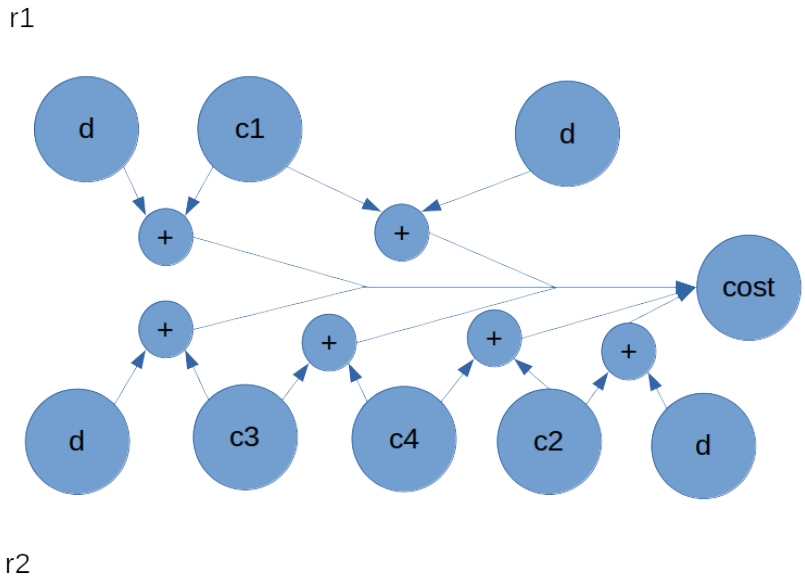

Figura 2.5: Grafo de evaluación que representa la solución $s1$ de VRP clásico luego de aplicada una instancia de *rarb*

Por ejemplo, a partir de una solución actual y un criterio, se puede explorar una vecindad con una búsqueda exhaustiva y mejora aleatoria o con una búsqueda aleatoria y primera mejora. En ambos casos se puede encontrar soluciones distintas. Programar cada combinación de estas estrategias demanda tiempo de trabajo humano.

En [7] se propone una herramienta que permite generar funciones de exploración a partir de un criterio, una estrategia de selección y una de exploración. Las posibles estrategias se definen como clases y usando instancias de estas clases se puede generar una función de exploración que se comporte de acuerdo a las instancias recibidas. La generación de cada porción del código se logra utilizando funciones genéricas, en particular los métodos *:after* [**?**].

La función de exploración resultante recibe el problema que se quiere resolver junto con una solución actual, ejecuta la exploración y retorna un vecino de acuerdo al criterio de selección. Si ninguna solución explorada cumple con la estrategia de selección, se retorna **nil**.

En [7], para realizar la exploración, se utilizaban ciclos que hacían poco flexible incorporar nuevas estrategias. Con el Árbol de vecindad es posible crear cualquier exploración y diseñar estrategias de búsqueda de mayor

complejidad. Además, en [7] tampoco se usa el Grafo de Evaluación por lo que la cantidad de problemas que podían ser evaluados era limitada. En este trabajo se combinan el Árbol de Vecindad y el Grafo de Evaluación junto con el generador de funciones de exloración para resolver cualquier problema que se pueda evaluar en el grafo, con cualesquiera combinaciones de estrategias de exploración y selección. El siguiente capítulo expone cómo se unen todas estas herramientas.

Capítulo 3

Propuesta de solución

La implementación del sistema está dividida en cuatro módulos: Núcleo, trabajo con vecindades, evaluación de soluciones y generador de funciones de exploración.

- Núcleo (**Core**): En este módulo se implementan las clases a partir de las cuales se pueden describir las soluciones y criterios de vecindad para Problemas de Enrutamiento de Vehículos.
- Trabajo con vecindades (**Neigh**): Se implementa el Árbol de Vecindad a partir del cual se genera soluciones con distintas formas de exploración de vecindades.
- Evaluación de soluciones (**Eval**): Se implementa el grafo de evaluación con el que se obtiene el costo de soluciones vecinas a partir de cómo se evalúa una solución.
- Generador de funciones de exploración (**Generator**): Este módulo contiene el código que permite generar funciones de exploración para combinar estrategias de exploración y selección.

Estos cuatro módulos ya existían antes de que se desarrollara el presente trabajo. En las secciones siguientes se describen los detalles de cada uno de estos módulos, cómo se combinaron y qué modificaciones se le hizo a cada uno para implementar esta tesis. En ela sección 3.1 se describe el módulo **core**. En la sección 3.2 se describe el módulo **neigh**. En la sección 3.3 se describe el módulo **eval**. Por último, en la sección **??** se describe el módulo **generator**

3.1. Vrp-Core

En el módulo *core* se definen funciones para inicializar el sistema, así como implementaciones de algoritmos de búsqueda local que utilizan funciones del sistema para explorar vecindades.

También se define un conjunto de clases que permiten describir soluciones para un Problema de Enrutamiento de Vehículos. A partir de esas soluciones se construye el grafo de evaluación.

También se definen clases que permiten definir los datos necesarios para resolver problemas específicos. Por ejemplo, para CVRP es necesaria la demanda de cada cliente y la matriz que guarda la distancia entre cada par de clientes. Para el Problema con backhaul, se define también una lista que guarda los tipos de cada cliente. En el capítulo 4 se discute cómo ampliar el sistema para definir otras variantes de VRP.

Además, en este módulo se tiene clases que representan las operaciones que conforman los criterios de vecindad (como seleccionar ruta o insertar cliente).

Para el desarrollo de esta tesis, no se añadió código al módulo *core*. Por el contrario, se desecharon secciones de código que quedaron obsoletas. Por ejemplo, en la versión anterior se utilizaba algo llamado *solución de trabajo*, que era copia de la solución actual sobre la que se realizaban las modificaciones para evaluar vecinos eficientemente. Como esta tesis las evaluaciones se hacen mediante el Grafo de Evaluación, las *soluciones de trabajo* quedan obsoletas. En la versión anterior se tenía también unos mecanismos con los que se exploraba vecindades, que al incorporar el Árbol de vecindad fueron desechados.

Una vez definidas las clases que permiten crear una solución, definir características de un problema, y definir un problema, es posible construir un Árbol de Vecindad. Esto se hace en el módulo **neigh**.

3.2. Vrp-neigh

En este módulo se define el Árbol de Vecindad como una clase que se construye a partir de un problema, una solución y un criterio de vecindad.

El Árbol de Vecindad permite crear iteradores (o generadores) de soluciones. La función que se usa para crear el iterador depende del tipo de exploración que se desee (como **exhaustive-exploration** o **random-exploration**). Cada vez que se ejecuta un iterador, se obtiene la lista de operaciones que se aplican a la solución actual para generar un vecino, o **nil** en caso de que

se no queden vecinos por analizar. Esta condición varía según el tipo de exploración. Por ejemplo, para la exploración exhaustiva se generan todas las soluciones de la vecindad, mientras que en una aleatoria se genera una cantidad predefinida de vecinos.

En el trabajo original donde se desarrolló este módulo, las clases se definían de manera distinta a como se definen en el módulo **core**. Por tanto, en el presente trabajo se cambió las definiciones de clases para, en todos los casos, utilizar **def-vrp-class**. De esta forma, todas las clases de **neigh** se volvieron consistentes con el resto del sistema. También se agregó la función **random-exploration** para obtener el iterador de exploración aleatoria.

A partir del Árbol de Vecindad se obtienen iteradores que generan vecinos en forma de lista de operaciones de vecindad. Al aplicar estas operaciones sobre la solución actual se forma una solución cuyo costo se puede calcular utilizando el Grafo de Evaluación.

3.3. Vrp-eval

En esta sección se implementa el Grafo de Evaluación, que como se explicó en la sección 3.3, tiene tres tipos de nodos: Nodos que representan los elementos de la solución (clientes, depósitos), nodos acumuladores (como el costo total), y nodos operacionales (aumentar valor, disminuir valor).

El grafo contiene los siguientes elementos:

- Una copia de la solución actual.
- Una lista con los nodos del grafo.
- Una correspondencia entre cada elemento de la solución y el nodo del grafo que lo representa.
- Una correspondencia entre los nodos acumuladores y las variables a las que representan.
- El valor del costo total de la solución.

Para incorporar este módulo al sistema hubo que reimplementarlo prácticamente desde cero. En su versión anterior, el código del grafo de evaluación estaba aislado del módulo **core** y, por tanto, este grafo no se podía construir a partir de las clases para describir soluciones que utiliza el sistema. Con el desarrollo de este trabajo se crearon métodos y clases de inicialización, construcción y modificación de grafo compatibles con las clases de **core**.

Para inicializar un grafo sólo es necesaria la solución actual. La inicialización consiste en convertir los elementos de la solución (clientes, depósitos, vehículos) en los nodos que los representan.

Para terminar de construir el grafo, se debe definir y ejecutar el código de evaluación de la solución inicial a partir de funciones definidas en el sistema. Algunos ejemplos de estas funciones son:

- **def-var**: Recibe un nombre y un valor inicial. Inicializa un nuevo nodo acumulador y se lo hace corresponder con el nombre recibido.
- **increment-distance**: Recibe dos clientes, una variable asociada a un nodo acumulador y la matriz de distancia. Crea y evalúa un nodo operacional que incrementa el valor almacenado en el nodo correspondiente a la variable en una cantidad igual a la distancia entre los clientes.
- **decrement-demand**: Recibe un cliente y una variable asociada a un nodo acumulador. Crea y evalúa un nodo operacional que recibe el cliente y decrementa el valor del nodo correspondiente a la variable en una cantidad igual a la demanda del cliente.
- **increment-value**: recibe dos variables. Incrementa el valor del nodo asociado a la segunda variable en una cantidad igual al valor del nodo asociado a la primera variable.
- **apply-penalty**: Recibe dos variables y un factor de penalización. Crea y evalúa un nodo operacional que recibe el nodo asociado a la primera variable y, en caso de que este tenga valor negativo, aumenta el valor del nodo asociado a la segunda variable en una cantida que depende del factor de penalización.
- **return-value**: Recibe una variable y marca su nodo asociado como el nodo que contiene el costo total de la solución. A partir del punto en que se invoca, la propiedad *output* (salida o costo total) del grafo referencia a este nodo.

Todas las funciones de construcción de grafo reciben también como parámetro la instancia del grafo sobre el que se está trabajando. En el capítulo 4 se discutirá la ampliación del sistema con nuevos métodos.

A continuación se muestra el pseudo-código que evalúa una solución de CVRP.

```
1 def-var total-distance (initial-value = 0)
2 loop for r in solution-routes do
3   def-var route-distance (initial-value = 0)
4   def-var route-capacity (initial-value = vehicles-capacity)
5   loop for c in r-clients do
6     increment route-distance in distance between r-previous-client
          c
7     decrement route-demand in client-capacity
8   set r-previous as c
9   increment total-distance in route-distance
10  apply-penalty in total-distance with route-capacity
11 return-value total-distance
```

En las líneas 1 se usa def-var para inicializar, y asociar a su respectivo nodos, la variable que almacenan el costo total de la solución. Luego se analizan las rutas de la solución. Por cada ruta se definen, y asocian a sus respectivos nodos, variables que almacena el costo (línea 3) y la capacidad restante de la ruta (líneas 4). A continuación se analizan los clientes de la ruta actual. Se incrementa el valor del nodo asociado al costo de la ruta en una cantidad igual a la distancia del cliente actual y el cliente previo (el cliente previo inicial de la ruta es el depósito)(línea 6). También se decrementa el valor del nodo asociado a la capacidad de la ruta en una cantidad igual a la demanda del cliente actual (línea 7) Una vez se analiza una ruta, se aumenta el valor del nodo asociado al costo total de la solución en una cantidad igual al costo de esta ruta (línea 8) y se penaliza el nodo de costo total en caso de ser necesario (línea 9). Finalmente se declara como nodo salida al nodo asociado a la variable de distancia total (línea 10).

En su versión anterior, el módulo **eval** tampoco era compatible con las clases de **core** que describen operaciones de vecindad. Para aplicar y deshacer en el grafo las operaciones de vecindad que utiliza el sistema se implementaron las funciones **do-core-operations** y **undo-core-operations**.

Do-core-operations recibe una lista de operaciones y el grafo sobre el cual estas se ejecutan. Las operaciones se hacen en orden. Los nodos que representan elementos de la solución se insertan o eliminan del grafo y los nodos operacionales que dependan de ellos se ejecutan o deshacen. Por ejemplo, dado el grafo representado en la figura 3.1, al ejecutar el siguiente código:

```
1 do-core-operations (graph ((op-select-client 1 from route 1)
```

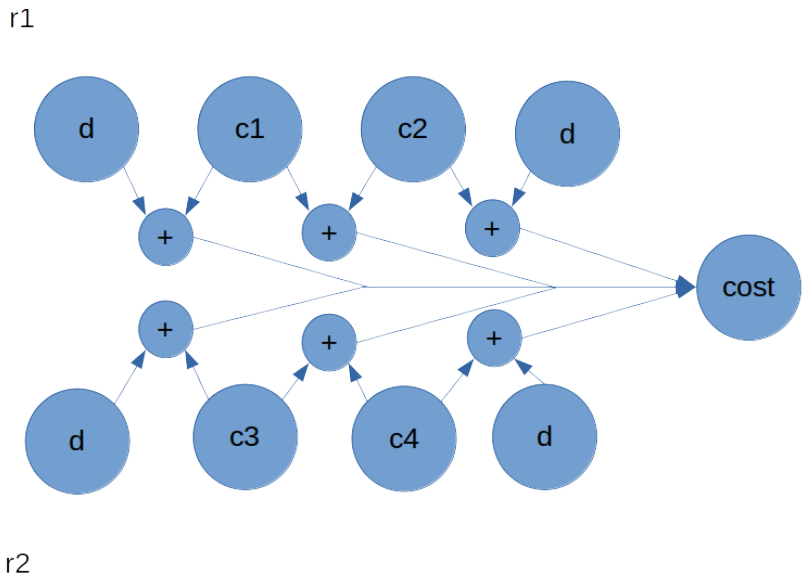

Figura 3.1: Grafo de evaluación que representa la solución $s1$ de VRP clásico.

```
2                              (op-insert-client 1 to route 3 in 2))
```

Se obtiene el grafo representado en 3.2. En este punto, la propiedad *output* del grafo (Que en este caso hace referencia al nodo *cost*) tiene el costo total de la nueva solución.

Para deshacer operaciones que se hicieron previamente en un grafo se ejecuta la función **undo-core-operations**. Por ejemplo, al ejecutar el siguiente código sobre el grafo representado en la figura 3.2:

```
1 undo-core-operations (graph ((op-select-client 1 from route 1)
2                              (op-insert-client 1 in route 3 in 2))
```

Se obtiene nuevamente el grafo representado por 3.1 y el valor almacenado en *cost* vuelve a ser el costo total de la solución original.

Una vez que se definen funciones que generan y evalúan soluciones, sólo resta utilizarlas en exploraciones de vecindades. De eso se habla en la

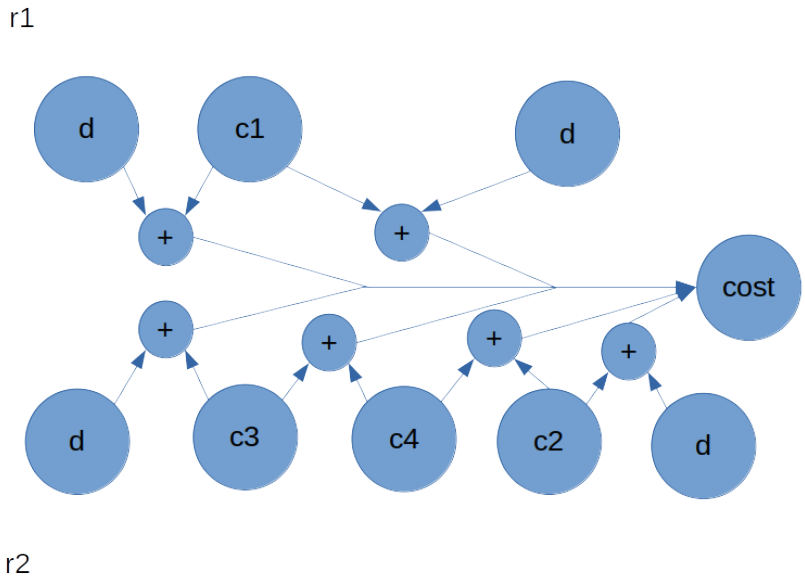

Figura 3.2: Grafo de evaluación que representa la solución $s1$ de VRP clásico luego de aplicada una instancia de *rarb*

siguiente sección.

3.4. Vrp-generator

Una vecindad se puede analizar con distintas estrategias de exploración y selección. Cada vez que se combina una estrategia de exploración con una de selección se obtiene una manera diferente de analizar la vecindad. A cada una de estas combinaciones, en este trabajo se les llama función exploradora de la vecindad. Programar estas funciones exploradoras es un proceso con alto consumo de tiempo de trabajo humano. En [7] se propone un mecanismo para automatizar la creación de funciones exploradoras a partir de cualesquiera estrategias de exploración y selección, con un esfuerzo mínimo. Esto significa que ya no es necesario programar todas las funciones de exploración diferentes, lo cual agiliza el tiempo de desarrollo. Para esto, se utilizan las ventajas del Sistema de Objetos de Common Lisp (CLOS), en particular las combinaciones de métodos y la facilidad que brinda el lenguaje para generar código fuente en tiempo de ejecución.

Las posibles estrategias de exploración y selección se representan como

clases y los criterios de vecindad mediante una lista de operaciones. En [7] se tiene una un mecanismo que recibe una estrategia de exploración, una de selección y un criterio, y devuelve una función que dada una solución explora la vecindad correspondiente a esas entradas.

La versión del módulo **generator** que se hizo en [7] utilizaba en las funciones de exploración que se generaban, elementos de **core** que con este trabajo quedaron obsoletos. Por tanto, parte de los mecanismos desarrollados en [7] también quedaron obsoletos.

En la versión anterior las vecindades se exploraban con cadenas de ciclos que mezclaban las operaciones. En el presente trabajo, esto se hace con un único ciclo que genera soluciones a partir del iterador del Árbol de Vecindad. Por otro lado, los vecinos se evaluaban mediante *soluciones de trabajo* y esto ahora se hace a mediante el Grafo de Evaluación.

En este trabajo se define un nuevo generador de funciones que utiliza un Árbol de Vecindad y que, además, en lugar de ceribir una solución, ahora reciven un Grafo de Evaluación.

En [7] el código de una función de exploración (y por tanto, el código que se genera a partir de la función generadora) está dividido en 5 regiones comunes para cualquier exploración:

- Inicializaciones de variables.
- Ciclos que definen cómo realizar la exploración (dependen del criterio).
- Instrucciones que se ejecutan dentro de los ciclos.
- Instrucciones que se ejecutan fuera de los ciclos.
- Valores que se retornan.

El siguiente pseudo-código muestra un ejemplo de función exploradora para el criterio *mover cliente en cualquier posición de posición dentro de su ruta* con estrategia de exploración exhaustiva y con selección de mejor vecino. Esta función recibe una solución (**solution**) como argumento.

```
1 set wc as working-copy of solution
2 set ops-list as nil
3 set current-cost as 0
4 set best-cost as 0
5 set best-neighbor as nil
```

```
6  set result-solution as nil
7
8  loop foreach route r:
9     loop foreach client c:
10       loop foreach pos p:
11
12          set current-ops-list as (select-route-r,
13                                   select-client-c,
14                                   insert-client-c-in-route-r-in-position-p)
15          set current-cost as apply-operations(wc current-ops-list)
16          if (current-cost < best-cost) then:
17             set best-neighbor as current-ops-list
18             set best-cost as current-cost
19
20 if (best-neighbor exists) then
21    set result-solution as operations best-neighbor applied over
          solution
22
23 return result-solution
```

En las líneas 1-6 seinicializan las variables variables necesarias. En las líneas 8-10 se crean los ciclos que definen cómo realizar la exploración, por cada operación del criterio se hace un ciclo. Las líneas 12-18 contienen el código dentro de los ciclos, se obtiene el costo actual a partir de la *solución de trabajo* y en caso de ser menor que el actual mejor costo, se actualiza **best-cost** y *best-neighbor*. Las líneas 20-21 tienen el código fuera de los ciclos, en caso de haberse encontrado un mejor vecino, se aplican sus operaciones sobre la solución actual para obtener la nueva solución. La línea 23 retorna la solución encontrada.

En este trabajo, la sección de los ciclos que definen cómo realizar la exploración se cambia por un único ciclo cuya condicional depende de la estrategia. El sigiente pseudo-código muestra la implementación del ejemplo anterior generado por el sistema desarrollado en esta tesis. Esta función recibe un grafo (**graph**) como parámetro.

```
1 set neigh-tree as build-neigh-tree with criteria and graph-solution
2 set generator as exhaustive-exploration(neigh-tree)
3 current-ops-list as funcall(generator)
4
5 set best-cost as 0
```

```
6  set best-neighbor as nil
7
8  cicle while (current-ops-list exists)
9
10         do-core-operations (current-solution graph)
11         set current-cost as output of graph
12         undo-core-operations (current-solution graph)
13         if (current-cost < best-cost) then:
14            set best-neighbor as current-ops-list
15            set best-cost as current-cost
16         set current-ops-list as funcall(generator)
17
18 if (best-neighbor exists) then
19    do-core-operations best-neighbor over graph
20
21 return graph-current-solution
```

En la línea 8 se sustituye el código de la región de los ciclos por un único ciclo que se ejecutará mientras se cumpla su condición (en este caso, que el generador retorne un vecino). Además, la función ahora genera los vecinos a partir de un Árbol de Vecindad y en lugar de recibir una solución como argumento, recibe el grafo asociado a esa solución mediante el cual se obtiene el costo de los vecinos.

En este capítulo se exlicaron las características de cada módulo de este trabajo. El siguiente capítulo explica paso a paso cómo utilizar el sistema para resolver variantes de VRP.

Capítulo 4

Método de uso

Este capítulo explica los pasos que debe seguir un usuario para resolver variantes de VRP utilizando el sistema que se implementó en este trabajo. Esto se hará a través de un ejemplo de resolución de un problema CVRP arbitrario. Para esto se define el problema en 4.1, se define la solución y se construye el grafo de evaluación en 4.2 y se generan las funciones de exploración en 5.3. En 4.4 se presenta el código completo.

4.1. Definición del problema

El primer paso para resolver un VRP usando este sistema, es definirlo usando las clases que lo representan a él y a sus características. Esto se mostrará a través de un ejemplo de resolución de un CVRP con seis clientes. En este caso, se debe crear un objeto de tipo **basic-cvrp-client** (cliente con demanda) por cada cliente, uno de tipo **basic-depot** (depósito) y una matriz de distancias de tamaño $6x6$.

El siguiente código se usa para definir el problema.

```
1 (defparameter c1 (basic-cvrp-client 1 1))
2 (defparameter c2 (basic-cvrp-client 2 1))
3 (defparameter c3 (basic-cvrp-client 3 4))
4 (defparameter c4 (basic-cvrp-client 4 3))
5 (defparameter c5 (basic-cvrp-client 5 2))
6 (defparameter c6 (basic-cvrp-client 6 1))
7
8 (defparameter d0 (basic-depot))
9
```

```
10 (defparameter dist-mat #2A((0 1 2 3 4 5 6)
11                            (1 0 5 2 1 3 2)
12                            (2 5 0 2 2 2 2)
13                            (3 2 2 0 1 2 1)
14                            (4 1 2 1 0 2 3)
15                            (5 3 2 2 2 0 1)
16                            (6 2 2 1 3 1 0)))
17
18 (defparameter problem (basic-cvrp-problem
19                            :id 1
20                            :clients (list c1 c2 c3 c4 c5 c6)
21                            :depot d0
22                            :distance-matrix dist-mat
23                            :capacity 20))
```

En las líneas 1 a 6 se definen los clientes, estos reciben dos parámetros, *id* y *demanda* respectivamente. Los vehículos se definen en las líneas 8 y 9, reciben también dos parámetros *id* y *capacidad* respectivamente. Luego se declara el depósito (línea 11), la matriz de distancias (línea 13) y se crea la instancia del problema (línea 21).

Una vez se definió el problema, se debe crear una solución inicial como se muestra en la siguiente sección.

4.2. Solución inicial y Grafo de Evaluación

La solución inicial se utiliza para construir e inicializar el grafo. En el caso de CVRP una solución está formada por una lista de rutas que son instancias de la clase **route-for-simulation**. El siguiente código muestra cómo se crea una solución:

```
1 (defparameter r1 (route-for-simulation
2                    :id 1
3                    :vehicle (cvrp-vehicle 1 20)
4                    :depot d0
5                    :clients (list c1 c2 c3 (clone d0))
6                    :previous-client (clone d0)))
7 (defparameter r2 (route-for-simulation
8                    :id 2
9                    :vehicle (cvrp-vehicle 2 20)
10                   :depot d0
```

```
11                    :clients (list c4 c5 c6 (clone d0))
12                    :previous-client (clone d0)))
13
14 (defparameter s1 (basic-solution
15                    :id 1
16                    :routes (list r1 r2)))
```

Las rutas se definen como instancias de la clase **rute-for-simulation** (líneas 1 y 7). Esta clase hereda de la clase **basic-route** y la extiende con la propiedad *previous-client*. Esta propiedad se inicializa con una referencia al depósito a partir de la cual se crea el primer nodo de la ruta en el grafo de evaluación y simplifica la implementación del código de evaluación de la solución. Nótese que las rutas del grafo de evaluación comienzan y terminan en el depósito que se debe clonar para tener, en cada posición, nodos distintos.

Luego se debe inicializar el grafo con la función **init-graph**.

```
1 (defparameter graph (init-graph s1))
```

Una vez que el grafo está inicializado, se le deben agregar las operaciones y nodos que mantienen valores. Esto se consigue mediante el código de evaluación de la solución inicial.

```
1 (def-var total-distance 0 graph)
2 (loop for r in (routes s1) do
3   (def-var route-distance 0 graph)
4   (def-var route-demand (capacity (vehicle r)) graph)
5   (loop for c in (clients r) do
6     (increment-distance (previous-client r) c route-distance
          dist-mat graph)
7     (decrement-demand c route-demand graph)
8     (setf (previous-client r) c)
9   (increment-value total-distance route-distance graph)
10  (apply-penalty route-demand total-distance 10 graph)
11 (return-value total-distance graph))
```

En la línea 1 se usa **def-var** para inicializar la variable que almacena el costo total de la solución. Luego se analizan las rutas de la solución. Por cada ruta se define una variable nueva que almacena su costo (línea 3) y otra que almacena la capacidad restante de su vehículo (línea 4). Entonces se analizan los clientes de la ruta actual. Se incrementa el costo de la ruta en una cantidad igual a la distancia del cliente actual y el cliente previo (el cliente previo inicial de la ruta es el depósito), y se disminuye la capacidad del vehículo en una cantidad igual a la demanda del cliente (líneas 6 y 7 respectivamente). Después de analizar cada ruta se aumenta el costo total de la solución en una cantidad igual a los costos de las mismas y se aplica penalización en caso de que un vehículo haya alcanzado una capacidad restante negativa (líneas 9 y 10). Finalmente se retorna la distancia total.

La siguiente sección ejemplifica cómo se generan las funciones de exploración a partir de criterios de vecindad y estrategias de exploración y selección.

4.3. Generación de funciones de exploración

Una vez que se tiene el grafo de evaluación de una solución inicial, para resolver el problema con una búsqueda local se debe definir cómo explorar la vecindad.

Las estrategias de exploración y selección se definen como clases cuyas instancias se pasan como parámetros a la función generadora. Se utilizan métodos que reciben estas instancias y generan código a partir de especializaciones de los tipos de los argumentos recibidos.

A continuación se presentan algunas clases que representan estrategias de exploración y selección:

Exploración:

- **exhaustive-neighborhood-search-strategy**: Exploración exhaustiva.
- **random-neighborhood-search-strategy**: Exploración aleatoria.

Selección:

- **best-improvement-search-strategy**: Selección de mejor solución.
- **first-improvement-search-strategy**: Selección de primera mejora.
- **random-improvement-with-candidates-selection-strategy**: Selección aleatoria de una mejora entre los elementos de una lista con las mejoras encontradas.

- **random-improvement-selection-strategy**: Selección aleatoria de una mejora en base a una probabilidad.

Durante la definición de cada clase se crea un parámetro global que almacena una instancia de su respectiva clase. Por ejemplo, el parámetro **+exhaustive-search-strategy+** tiene como valor asociado una instancia de **exhaustive-neighborhood-search-strategy**.

A continuación se define una lista de funciones de exploración para los criterios *mover un cliente dentro de su ruta*, *mover un cliente a cualquier ruta en cualquier posición* e *intercambiar dos clientes de posición*. En todos los casos se hace una búsqueda exhaustiva con selección de mejor vecino.

```
(setf rab (make-neighborhood-criterion
                ‘((select-route r1)
                (select-client c1 from r1)
                (insert-client c1 to r1))
                +exhaustive-search-strategy+
                +best-improvement+))

(setf rarb (make-neighborhood-criterion
                ‘((select-route r1)
                (select-client c1 from r1)
                (select-route r2)
                (insert-client c1 to r2))
                +exhaustive-search-strategy+
                +best-improvement+))

(setf rarac (make-neighborhood-criterion
                ‘((select-route r1)
                (select-client c1 from r1)
                (select-route r2)
                (select-client c2 from r2)
                (swap-clients c1 c2))
                +exhaustive-search-strategy+
                +best-improvement+))

(setf criteria (list rab rarb rarac))
```

En este punto es posible invocar la función de metaheurística de Búsqueda de Vecindad Variable y obtener una solución.

```
(setf result (vns-vrp-system problem criteria graph :max-iter 1000))
```

La siguiente sección muestra el código completo que soluciona un CVRP con datos ficticios.

4.4. Código completo

```
(defparameter c1 (basic-cvrp-client 1 1))
(defparameter c2 (basic-cvrp-client 2 1))
(defparameter c3 (basic-cvrp-client 3 4))
(defparameter c4 (basic-cvrp-client 4 3))
(defparameter c5 (basic-cvrp-client 5 2))
(defparameter c6 (basic-cvrp-client 6 1))


(defparameter d0 (basic-depot))

(defparameter dist-mat #2A((0 1 2 3 4 5 6)
                           (1 0 5 2 1 3 2)
                           (2 5 0 2 2 2 2)
                           (3 2 2 0 1 2 1)
                           (4 1 2 1 0 2 3)
                           (5 3 2 2 2 0 1)
                           (6 2 2 1 3 1 0)))

(defparameter problem (finite-fleet-cvrp-problem
                           :id 1
                           :clients (list c1 c2 c3 c4 c5 c6)
                           :depot d0
                           :distance-matrix dist-mat
                           :capacity 20))

(defparameter r1 (route-for-simulation
                       :id 1
                       :vehicle (cvrp-vehicle 1 20)
                       :depot d0
                       :clients (list c1 c2 c3 (clone d0))
                       :previous-client (clone d0)))
(defparameter r2 (route-for-simulation
                       :id 2
```

```
                        :vehicle (cvrp-vehicle 2 20)
                        :depot d0
                        :clients (list c4 c5 c6 (clone d0))
                        :previous-client (clone d0)))

(defparameter s1 (basic-solution
                        :id 1
                        :routes (list r1 r2)))

(defparameter graph (init-graph s1))


(def-var total-distance 0 graph)
(loop for r in (routes s1) do
   (def-var route-distance 0 graph)
   (def-var route-demand (capacity (vehicle r)) graph)
   (loop for c in (clients r) do
      (increment-distance (previous-client r) c route-distance
          dist-mat graph)
      (decrement-demand c route-demand graph)
      (setf (previous-client r) c))
   (increment-value total-distance route-distance graph)
   (apply-penalty route-demand total-distance 10 graph)
(return-value total-distance graph))


(setf rab (make-neighborhood-criterion
                      '((select-route r1)
                      (select-client c1 from r1)
                      (insert-client c1 to r1))
                      +exhaustive-search-strategy+
                      +best-improvement+))

(setf rarb (make-neighborhood-criterion
                      '((select-route r1)
                      (select-client c1 from r1)
                      (select-route r2)
                      (insert-client c1 to r2))
                      +exhaustive-search-strategy+
                      +best-improvement+))

(setf rarac (make-neighborhood-criterion
                      '((select-route r1)
                      (select-client c1 from r1)
                      (select-route r2)
```

```
78                      (select-client c2 from r2)
79                      (swap-clients c1 c2))
80                      +exhaustive-search-strategy+
81                      +best-improvement+))
82
83 (setf criteria (list rab rarb rarac))
84
85 (setf result (vns-vrp-system problem criteria graph :max-iter
       10000000000))
```

Para resolver otra variante de VRP utilizando el sistema que se implementó en el presente trabajo basta con definir las características del problema y el código de evaluación de la solución inicial. Además, cualquier problema que se defina puede ser explorado con cualquier criterio de vecindad sin importar la cantidad de operaciones que tenga y con cualesquiera combinaciones de estrategias de exploración y selección.

En este capítulo se explicó paso a paso el método de uso del sistema. Sin embargo, este sistema no es capaz de resolver todas las variantes de VRP existentes. El siguiente capítulo explica cómo extenderlo para resolver problemas que actualmente no se pueden representar.

Capítulo 5

Extensibilidad

Con las funcionalidades implementadas hasta el momento, el sistema que se propone puede resolver diversas variantes de VRP. Sin embargo, en algún momento, un usuario necesitará resolver un problema que aún no se pueda representar. En este capítulo se explica cómo extender el sistema para otras variantes.

Para extender el sistema a nuevos problemas se debe crear las clases que lo describen y las funciones que permitan construir su grafo de evaluación a partir de una de sus soluciones. También es posible definir nuevas estrategias de exploración y selección.

En 5.1 se explica cómo extender el módulo **core**. En 5.2 se explica cómo extender el módulo **eval**. En el capítulo 5.3 se explica cómo extender el módulo **generator**.

5.1. Descripción del problema

Para extender el sistema a nuevas variantes de VRP puede ser necesario agregar nuevas clases para representar las características del problema. Las características de cada característica del problema depende de las clases de las que heredan. Además, todas las clases que representan características de problemas son descendientes de su clase base correspondiente. Por ejemplo, un cliente de CVRP (clase **basic-cvrp-client**) hereda de **demand-client**, clase que indica que el cliente posee una demanda y de **basic-client**, la clase base para los clientes.

Definir una clase nueva implica identificar de qué clases debe heredar para satisfacer sus especifiaciones y, en caso de ser necesario, crear nuevas clases. Definir un problema nuevo puede implicar la creación de varias

clases para representar sus características.

A continuación se ejemplificará cómo crear la clase **basic-time-windows-problem** que representa un Problema de Enrutamiento de Vehículos con Vetanas de tiempo (TWVRP) [**?**]. En este problema los clientes, además de sus demandas, tienen un período de tiempo en que pueden se pueden visitar, de lo contrario la solución es penalizada. Además, los vehículos deben esperar ciertas cantidades de tiempo mientras atienden a cada cliente.

Para definir el TWVRP es necesario crear clientes que conozcan sus ventanas de tiempo y el tiempo que debe consumir el vehículo atendiéndolos. Se definen las clases estructurales:

- **time-windows-client**: Tiene ventana de tiempo.
- **service-time-client**: Consume tiempo al ser atendido.

Una vez definidas estas dos clases se crea **basic-tw-client** que representa un cliente de TWVRP y hereda de:

- **basic-client**
- **demand-client**
- **time-windows-client**
- **service-time-client**

Las rutas que existen en el sistema no pueden determinar el momento en que se encuentra su vehículo, por eso se deben definir nuevas rutas que conozcan el tiempo actualmente consumido. Se crea la clase **route-with-time** y la clase **basic-tw-route** que hereda de **route-with-time** y **basic-route** (o **route-for-simulation** si se quiere utilizar en el Grafo de Evaluación).

También debe definirse la clase **time-problem** para indicar que el problema tiene una matriz de nxn cuyas posiciones guardan los tiempos necesarios para viajar entre clientes. Finalmente es posible definir la clase **basic-time-windows-problem** para el problema con ventanas de tiempo que hereda de las siguientes clases abstractas:

- **basic-problem** (tiene clientes, depósitos e identificador)
- **distance-problem** (tiene matriz de distancias)
- **capacity-problem** (tiene una capacidad por cada cliente)

- **time-problem** (tiene matriz de tiempos)

Junto con las nuevas características, definir un nuevo problema implica también definir cómo evaluar sus soluciones. Al evaluar una solución para un problema determinado puede hacer falta operaciones que aún no existen. El siguiente capítulo explica cómo definir nuevas operaciones

5.2. Evaluación

Al definir nuevos problemas, también se debe crear la forma en que sus soluciones se evalúan y, en caso de ser necesario, extender el Grafo de Evaluación para satisfacer nuevas características y restricciones. Extender el grafo se traduce en agregar los nuevos nodos y las funciones que se usarán en el código de evaluación.

Por ejemplo, a continuación se muestra cómo extender el Grafo de Evaluación para evaluar soluciones del de CVRP con una restricción extra. Se restringe el problema de capacidad para, además de penalizar la solución por exceso de carga en los vehículos de sus rutas, tambíen se penaliza si la ruta excede un número definido de vecinos k. El código completo que extiende el sistema para resolver CVRP con limitación de tamaño de rutas se muestra en el capítulo 6.

Para evaluar una solución de este poroblema se define una variable por cada ruta que se inicializa con valor k y se decrementa en uno por cada cliente de la ruta. Luego, en caso de tener esta variable valor negativo, se penaliza el costo total de la solución. El código de evaluación de CVRP con rutas limitadas se muestra a continuación:

```
1
2  (progn
3     (def-var total-distance 0 graph)
4     (loop for r in (routes s1) do
5        (progn
6           (def-var route-distance 0 graph)
7           (def-var route-demand (capacity (vehicle r)) graph)
8           (def-var route-limit k graph)
9           (loop for c in (clients r) do
10             (progn
11                (increment-distance (previous-client r) c
                      route-distance A-n33-k5-distance-matrix graph)
12                (decrement-demand c route-demand graph)
```

```
13              (decrement-size c route-limit graph)
14              (setf (previous-client r) c)))
15      (increment-value total-distance route-distance graph)
16      (apply-penalty route-demand total-distance 100 graph)
17      (apply-penalty route-limit total-distance 1000 graph)))
18    (return-value total-distance graph)))
```

En la línea 5 se crea la variable **route-limit**. En la línea 13 se decrementa en uno el valor de **route-limit** por cada cliente **c**. En la línea 17 se penaliza el costo total si **route-limit** tiene valor negativo. Para que este código funcione debe definirse la función **decrement-size** que cree un nodo operacional (que también debe definirse) **decrement-size-node** que recibe un cliente y, al ejecutarse, decrementa en uno el valor del nodo asociado a route-limit. Para definir cómo se ejecuta y cómo se deshace este nodo operacional deben programarse los métodos **evaluat** y **undo** correspondientes.

En este punto es posible construir el grafo con su solución inicial evaluada. Sin embargo, para evaluar soluciones vecinas al los clientes se remueven e insertan. Remover un cliente implica deshacer su **decrement-size-nodo** y por tanto, cada cliente debe tener una propiedad que referencie este **decrement-size-node**. Los nodos que se usan para evaluar CVRP no tienen una propiedad para referenciar su **decrement-size-node** y por tanto deben crearse nuevos tipos de nodo cliente que sí la tengan. Se define la clase **input-limited-distance-demand-node** que además de la referencia a su nodo operacional de que penzaliza el tamaño de la ruta, mantiene referencias a sus nodos operacionales de penalización por capacidad y de aumento de distancia.

Los nodos que representan elementos de la solución en el grafo se obtienen transformando clases de *core* en clases nodos de *eval* métodos especializaciones de la función genérica **convert-to-node**. Debe implementarse una especialización de **convert-to-node** que cree nodos de tipo **input-limited-distance-demand-node**.

Por último se deben implementar los métodos que inserten y remuevan los nuevos nodos definidos para este problema y ejecuten o deshagan todos los nodos operacionales que dependan de estos nuevos nodos.

El sistema también se puede extender implementando nueas estrategias de exploración y selección.

5.3. Estrategias

Actualmente, el presente sistema cuenta con implementaciones de varias estrategias de exploración y selección, mencionadas en 3.4 que al ser combinadas generan exploraciones distintas. Es posible también extender la generación de funciones de exploración a partir de nuevas estrategias.

La definición de estrategias lleva dos pasos. Primero debe crearse la clase que representa esta estrategia. En dependencia del comportamiento que se espera, la nueva clase puede heredar de clases auxiliares ya creadas o de nuevas clases auxiliares que se definen. El segundo paso consiste en implementar las especializaciones de métodos de cada tipo (inicializaciones de variables, código dentro del ciclo, código de retorno, etc) que reciban como parámetros las nuevas clases que se crearon.

Opcionalmente se puede asociar una instancia cada estrategia que se implemente a un parámetro global con el siguiente formato: **+*name-type*-strategy+**, donde ***name*** es el nombre de la estrategia y ***type*** es *search* o *selection*.

Por ejemplo, a continuación se crea una estrategia de selección en la que se retorna vecino aleatorio de entre una lista de mejoras, pero esa lista sólo tendrá soluciones cuya mejora con respecto a la inicial supere cierto margen. se define la clase **random-improvement-with-restricted-candidates-selection-strategy** y una instancia se asocia al parámetro global **+random-improvement-with-restricted-candidates-selection-strategy+**.

La nueva clase tiene una propiedad **aceptance** para el margen de aceptación con valor entre 0 y 1. Se aceptan soluciones que cumplen:

$$cost_s < cost_{init-s} - cost_{init-s} * \textbf{aceptance}$$

Random-improvement-with-restricted-candidates-selection-strategy debe heredar de las siguientes clases auxiliares que ya están en el sistema:

- **use-eval-graf**: Para generar soluciones con Árbol de vecindad y evaluarlas con grafo de evaluación.
- **return-best-solution**: Para crear y retornar una variable **best-solution**

Además, se debe crear la clase auxiliar:

- **has-restricted-candidates-for-best-neighbor**: Tiene un comportamiento parecido a **has-restricted-candidates-best-neighbor** (una clase que ya está en el sistema). Indica que se tiene lista de candidatos con vecinos que cumplen la restricción de aceptación.

Luego deben implementarse las especializaciones de los métodos **generate-code-inside-let**, **generate-code-inside-loop** y **generate-code-outside-loop** tal que reciban como parámetro la nueva clase **random-improvement-with-restricted-candidates-selection-strategy**.

En las inicialicaciones dentro del **let** se inicializa una variable con nombre **candidates-for-best-neighbor**.

Dentro del ciclo se verifica si el costo de la solución inicial cumple con la restricción establecida y en caso positivo, se agrega esta a la lista **candidates-for-best-neighbor**.

Finalmente, fuera del ciclo se escoge aleatoriamente un vecino de **candidates-for-best-neighbor**, se asocia este a la variable **best-neighbor** y se asocia su costo a **best-cost**. Las variables **best-neighbor** y **best-cost** se crean dentro del let en la especialización de **return-best-solution**. En caso de **candidates-for-best-neighbor** estar vacía, entonces no se hace nada y tanto **best-neighbor** como **best-cost** permanecen iguales.

En este capítulo se explicó cómo extender el sistema para resolver variantes de VRP que no estén soportadas actualmente. El siguiente capítulo muestra los resultados que se obtuvieron al utilizar el sistema de este trabajo con datos reales.

Capítulo 6

Experimentos y resultados

En este capítulo se presentan las pruebas que se hicieron para evaluar el desempeño del presente sistema. Para esto pruebas se utilizaron datos de CVRP reales conocidos como *A-n33-k5* [**?**]. Se utilizó el algoritmo de búsqueda local *Búsqueda de Vecindad Variable* [**?**] con funciones de exploración generadas a partir **+exhaustive-search-strategy+**, **+best-solution-selection-strategy+** y los siguientes criterios de vecindad:

- rab.
- rarb.
- rarac.
- ref.
- rerf.
- rereg.

A-n33-k5 tiene 32 clientes y se conoce una solución óptima de costo 661.

Luego de aplicar la metaheurística se obtuvo la siguiente solución con costo 675, muy cercano al costo óptimo.

```
S1: (675)
routes:
<r5: <cv:1. 0/100> (<d:0>: (<c2: 23> <c24: 13> <c6: 18> <c23: 14>))>
<r4: <cv:1. 0/100> (<d:0>: (<c20: 8> <c32: 3> <c13: 23> <c8: 10>
    <c7: 19>
```

```
<c26: 2> <c4: 13> <c22: 19>))>
<r3: <cv:1. 0/100> (<d:0>: (<c15: 18> <c17: 24> <c9: 18> <c3: 14>
    <c16: 10>
<c29: 8>))>
<r2: <cv:1. 0/100> (<d:0>: (<c28: 15> <c18: 13> <c31: 24> <c1: 5>
    <c21: 10>
<c14: 9> <c19: 14> <c11: 5>))>
<r1: <cv:1. 0/100> (<d:0>: (<c12: 9> <c5: 8> <c27: 23> <c25: 14>
    <c30: 20>
<c10: 20>))>
```

Luego, para comprobar las capacidades del sistema para resolver distintas variantes con trabajo humano mínimo se le agregó una restricción al problema anterior. Según la nueva restricción, cada una de las rutas de la solución tiene un límite máximo de 7 clientes.

El código que se necesitó para resolver el nuevo problema se muestra a continuación. Primero se definen nuevas clases para los nodos que se agregan al grafo.

```
(def-vrp-class limited-client (demand-client)())

(def-vrp-class size-limited-cvrp-client (limited-client) ()
   :constructor (size-limited-cvrp-client (id demand)))

(def-vrp-class input-limited-node (input-node)
   ((limit-calculator :initform nil))
   :constructor (new-limited-node (&key content)))

 (def-vrp-class input-limited-distance-demand-node
     (input-distance-node input-demand-node input-limited-node) ()
   :constructor (new-input-limited-distance-demand-node (&key
       content)))

 (def-vrp-class decrement-size-node (low-level-node)
   ((input-with-limit))
   :constructor (new-decrement-size-node (&key previous-node )
```

Luego se implementan los métodos de construcción y modificación del grafo.

```
(defmethod convert-to-node :around ((target limited-client) graph)
  (let ((new-c (new-input-limited-distance-demand-node :content
      target)))
    (progn
      (setf (inputs graph) (append (inputs graph) `(,new-c)))
      (setf (gethash target (class-to-io graph)) new-c))))


(defmethod evaluate-low-level-node ((ll-node decrement-size-node))
  (progn
    (if (not (typep (content (input-with-limit ll-node))
        'basic-depot))
      (decf (output-value (output-action ll-node)) 1))
    (if (updater (output-action ll-node))
      (undo-low-level-node (updater (output-action ll-node))))))


(defmethod undo-low-level-node ((ll-node decrement-size-node))
  (progn
    (if (not (typep (content (input-with-limit ll-node))
        'basic-depot))
      (incf (output-value (output-action ll-node)) 1))
      (undo-low-level-node (updater (output-action ll-node)))))


(defmethod remove-node append ((t-node input-limited-node))
  (if (not (typep t-node 'input-depot-node))
    (progn
      (undo-low-level-node (limit-calculator t-node))))))


(defmethod insert-node append ((t-node input-limited-node)
(i-node input-limited-node))
  (let* ((new-inc (new-decrement-size-node
                    :output-action (output-action
                        (limit-calculator t-node))
                    :input-with-limit i-node)))
  (progn
    (setf (limit-calculator i-node) new-inc)
    (evaluate-low-level-node new-inc))))


(defmethod decrement-size (c capacity-slot-accumulator graph)
  (let* ((l-node (gethash c (class-to-io graph)))
```

```
42            (acc (gethash capacity-slot-accumulator (slot-to-output
                  graph)))
43            (l-calc (new-decrement-size-node :output-action acc
                  :input-with-limit l-node)))
44       (progn
45          (setf (limit-calculator l-node) l-calc)
46          (evaluate-low-level-node l-calc))))
```

En este punto sólo resta definir el código de evaluación de una solución para el nuevo problema.

```
1
2  (progn
3     (def-var total-distance 0 graph)
4     (loop for r in (routes s1) do
5        (progn
6           (def-var route-distance 0 graph)
7           (def-var route-demand (capacity (vehicle r)) graph)
8           (def-var route-limit 7 graph)
9           (loop for c in (clients r) do
10             (progn
11                (increment-distance (previous-client r) c
                     route-distance A-n33-k5-distance-matrix graph)
12                (decrement-demand c route-demand graph)
13                (decrement-size c 'route-limit graph)
14                (setf (previous-client r) c)))
15          (increment-value total-distance route-distance graph)
16          (apply-penalty route-demand total-distance 100 graph)
17          (apply-penalty route-limit total-distance 1000 graph)))
18    (return-value total-distance graph)))
```

Luego de ejecutar el sistema con los mismos datos de *A-n33-k5* para el nuevo problema, se obtuvo la siguiente solución de costo 727:

```
1 S1: (727)
2 routes:
3 <r5: <cv:1. 0/100> (<d:0>: (<c22: 19> <c15: 18> <c16: 10> <c3: 14>
      <c9: 18>
4 <c4: 13> <c20: 8>))>
```

```
<r4: <cv:1. 0/100> (<d:0>: (<c2: 23> <c32: 3> <c13: 23> <c8: 10>
    <c7: 19>
<c26: 2> <c10: 20>))>
<r3: <cv:1. 0/100> (<d:0>: (<c23: 14> <c28: 15> <c18: 13> <c11: 5>
    <c6: 18>
<c24: 13>))>
<r2: <cv:1. 0/100> (<d:0>: (<c29: 8> <c31: 24> <c1: 5> <c21: 10>
    <c14: 9>
<c19: 14>))>
<r1: <cv:1. 0/100> (<d:0>: (<c12: 9> <c5: 8> <c27: 23> <c25: 14>
    <c30: 20>
```

El siguiente capítulo presenta las conclusiones y recomendaciones.

Conclusiones y Recomendaciones

En este trabajo se implementó un sistema mediante el cual un usuario es capaz de resolver variantes de VRP sólo definiendo los datos del problema y programando cómo se evalúa una solución inicial. Para lograr esto se unieron los resultados de tesis anteriores [12][13][7] que resolvían problemas aislados. En este trabajo se combinan las ventajas de cada una de estas tesis.

Con el sistema propuesto es posible evaluar cualquier solución vecina de forma automática usando el Grafo de Evaluación, se puede explorar vecindades de cualquier forma con el Árbol de Vecindad y se puede generar funciones de exploración a partir de cualquier criterio de vecindad y cualquier combinación de estrategias de exploración y selección. Por tanto, es posible resolver Problemas de Enrutamiento de Vehículos con muy poco tiempo de trabajo humano. Sólo debe definir los datos del problema y cómo se evalúa una solución.

Siendo esta una primera aproximación, queda como recomendación hacer las pruebas pertinentes para comprobar la eficacia del sistema resolviendo problemas reales. Además, recomienda incorporar la extensión necesaria para resolver nuevas variantes.

Se recomienda también agregar al sistema nuevas características tales como la metaheurística de Búsqueda de vecindad infinitamente variable [7].

Por último, se recomienda modificar la implementación del Árbol de Vecindad en aras de aumentar su extensibilidad y también resolver las limitaciones del Grafo de Evaluación.

Bibliografía

[1] Alina Fernández Arias. Problema de enrutamiento de vehículos con recogida y entrega simultánea considerando una flota heterogénea. Master's thesis, Facultad de Matemática y Computación, Universidad de La Habana, La Habana, Cuba, July 2010.

[2] Alina Fernández Arias. *Modelos y métodos para el problema de enrutamiento de vehículos con recogida y entrega simultánea*. PhD thesis, Facultad de Matemática y Computación, Universidad de La Habana, 2017.

[3] Edward J Beltrami and Lawrence D Bodin. Networks and vehicle routing for municipal waste collection. *Networks*, 4(1):65–94, 1974.

[4] Daniela González Beltrán. Generación automática de gramáticas para la obtención de infinitos criterios de vecindad en el problema de enrutamiento de vehículos. 2019.

[5] Geoff Clarke and John W Wright. Scheduling of vehicles from a central depot to a number of delivery points. *Operations research*, 12(4):568–581, 1964.

[6] George B Dantzig and John H Ramser. The truck dispatching problem. *Management science*, 6(1):80–91, 1959.

[7] Heidy Abreu Fumero. Exploración de una vecindad para una solución de vrp combinando diferentes estrategias de exploración y de selección. Master's thesis, Facultad de Matemática y Computación, Universidad de La Habana, La Habana, Cuba, May 2019.

[8] Jan Karel Lenstra and AHG Rinnooy Kan. Complexity of vehicle routing and scheduling problems. *Networks*, 11(2):221–227, 1981.

[9] Camila Pérez Mosquera. Primeras aproximaciones a la búsqueda de vecindad infinitamente variable. Master's thesis, Facultad de Matemática y Computación, Universidad de La Habana, La Habana, Cuba, May 2017.

[10] P Parthanadee. A multi-product. multi-depot periodic distribution problem. 2002.

[11] B.E. Prentice and D. Prokop. Concepts of transport economics. *World Scientific Publishing*, 2016.

[12] Héctor Felipe Massón Rosquete. Exploración de vecindades grandes en el problema de enrutamiento de vehículos usando técnicas estadísticas. 2020.

[13] José Jorge Rodríguez Salgado. Una propuesta para la evaluación automática de soluciones vecinas en un problema de enrutamiento de vehículos a partir del grafo de evaluación de una solución. 2020.

[14] Éric Taillard, Philippe Badeau, Michel Gendreau, François Guertin, and Jean-Yves Potvin. A tabu search heuristic for the vehicle routing problem with soft time windows. *Transportation science*, 31(2):170–186, 1997.

[15] P Toth and D Vigo. The vehicle routing problem. monographs. *Discrete Mathematics and Applications, Society for Industrial and Applied Mathematics, Philadelphia*, 2002.