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 hasta el 60 porciento de los costos logísticos de una empresa. 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.

Existen además numerosas variaciones del problema clásico tales como el Problema de Enrutamiento de Vehículos con restricciones de capacidad (CVRP)[**?**], el Problema de Enrutamiento de Vehículos con ventanas de tiempo (VRPTW)[**?**] y el Problema de Enrutamiento de Vehículos con recogida y entrega (VRPPD)[**?**]. Cada variante tiene especificaciones propias, por lo que resulta difícil la creación de un método de solución universal para la familia de problemas VRP.

El hecho de ser problemas NP-Duros[**?**] implica la falta de soluciones exactas óptimas para conjuntos de datos no pequeños con muchos clientes a ser atendidos, por tanto, se utilizan técnicas no exactas como heurísticas y metaheurísticas que han sido objeto de estudio por décadas[**?**].

Una instancia de VRP es un problema con un conjunto de datos definidos. El proceso de solución de una instancia arbitraria de VRP es complejo y necesita de una considerable cantidad de tiempo. Siendo un problema de gran importancia tanto académica como industrial, ha inspirado la creación de numerosas herramientas y artículos científicos. Cabe destacar la biblioteca OR-Tools, un software de código abierto útil para resolver problemas de optimización combinatoria entre los que se encuentra VRP [**?**].

En la facultad de Matemática y Computación de La Universidad de La Habana este ha sido tema de estudio desde hace unos 6 años y se ha logrado resolver diversas problemáticas. En particular, la metaheurística de

búsqueda local infinitamente variable (IVNS) planteada por Camila Pérez en [4], la generación automática de gramáticas para IVNS hecha por Daniela Gonzáles en [1], la exploración de vecindades a partir de la combinación de distintas estrategias de exploración y selección hecha por Heidy Abreu en [3], el Árbol de vecindad con generación de soluciones y la exploración de dos fases fueron planteados por Héctor Massón en [6] y el Grafo de evaluación para la evaluación automática y eficiente de soluciones creado por Jose Jorge Rodríguez en [7]. Cada una resuelve por separado un problema distinto.

Hasta el momento, para resolver una instancia de VRP por búsqueda local se le debe invertir hasta dos años de tiempo a programar aspectos como la forma de evaluar vecinos o de explorar vecindades. A partir de la unión de las ideas anteriores es posible resolver un VRP únicamente programando la evaluación de una solución y eso es, precisamente, lo que se pretende implementar.

Objetivos

El objetivo general de este trabajo es diseñar e implementar un sistema que permita usar los beneficios de años anteriores de investigación para resolver (casi) cualquier instancia de VRP mediante una búsqueda local, a partir del código de evaluación de una solución.

Objetivos específicos

1. Consultar literatura especializada sobre el estado del arte de los problemas VRP.
2. Entender a profundidad las ideas y códigos propuestos en tesis anteriores.
3. Diseñar y programar un sistema que combine estas ideas para resolver cualquier VRP a partir de la evaluación de una solución.
4. Analizar los resultados obtenidos. Se utiliza el sistema para resolver instancias conocidas de problemas de VRP.

La propuesta de creación de generadores de código prefabricados planteada por Heidy Abreu facilita la exploración de vecindades combinando diferentes estrategias, con trabajo humano mínimo. Al agregar el árbol de vecindad desarrollado por Héctor Massón se puede generar soluciones vecinas de una solución una inicial de forma simple y eficiente. Utilizando

además el grafo de evaluación de Jose Jorge Rodríguez, se optimiza la evaluación y obtención de costo de las soluciones. El grafo de evaluación necesita que el usuario ingrese la forma de evaluar una solución y esto, junto con la descripción del problema, es lo único que el sistema requerirá para ejecutarse.

Organización de la tesis

El presente documento está organizado en 4 capítulos.

En el capítulo 1 **Preliminares** se describe a profundidad la familia de Problemas de Enrutamiento de Vehículos, se introducen las vías existentes (bibliotecas) para resolverlos, se describen las ideas desarrolladas en cada una de las tesis anteriores y se hace una breve descripción de Coommon Lisp y algunas de las funcionalidades utilizadas.

El capítulo 2 **Propuesta de Solución** describe el sistema implementado y la forma en que fueron unidas las piezas que lo conforman.

En el capítulo 3 **Método de uso** se describe paso a paso el método de uso el sistema y cómo describir y resolver instancias de VRP.

El capítulo 4 **Experimentos y resultados** comprende los experimentos realizados para validar el modelo, así como las métricas destinadas para su evaluación.

Por último se ofrecen las conclusiones a partir de los objetivos propuestos y los resultados alcanzados. Adicionalmente se brindan algunas ideas y recomendaciones para trabajos futuros.

Capítulo 1

Preliminares

Este trabajo pretende implementar un sistema que resuelva instancias de VRP en cualquiera de sus variantes con el menor trabajo humano posible. En este capítulo se presentan los principales elementos de la investigación realizada para lograrlo.

Primeramente se comienza con una explicación general del problema de enrutamiento de vehículos (VRP) y algunas de sus variantes con la sección 1.1. Se explica cómo describir a partir de código una solución de VRP.

En **??** se muestran las bibliotecas de clases existentes hasta el momento que pueden ser utilizadas para encontrar soluciones a problemas de VRP.

En 1.2 se explica cómo crear un Árbol de vecindad a partir de una solución inicial y un criterio de vecindad. Este árbol es utilizado para obtener la cardinalidad de vecindades y generar soluciones.

En 1.3 se expone el concepto de Grafo de evaluación y cómo es esto utilizado para evaluar soluciones de forma eficiente y automática.

En 1.4 se presenta un mecanismo para explorar vecindades de forma automática a partir de combinaciones de cualesquiera estrategias de exploración y selección.

Finalmente en 1.5 se describe brevemente algunas características y funcionalidades del lenguaje Common Lisp que resultaron especialmente útiles para el desarrollo del sistema.

1.1. Problema de Enrutamiento de Vehículos

La primera referencia al VRP fue hecha por Dantzing y Ramser en [2] en el año 1959. Se propone una formulación matemática, una aproximación

algorítmica y se describe una aplicación real entregando gasolina a varias estaciones de servicio.

En su versión más simple, VRP 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 caminos a asignar a los vehículos de forma que se optimice determinada métrica (tiempo, combustible, etc). Con más de 50 años de estudios se ha ramificado en una inmensa cantidad de variantes entre las que se pueden contar las siguientes:

- CVRP - VRP con restricciones de capacidad. Cada vehículo tiene una capacidad que no debe ser excedida.
- VRPTW - VRP con ventanas de tiempo. Cada cliente posee un período de tiempo fijo durante el cual puede ser atendido.
- 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 instancias con muchas entradas (clientes). Para buscar aproximaciones a la solución se utilizan heurísticas y metaheurísticas. Se destaca la búsqueda local como metaheurística que ha dado muy buenos resultados encontrando óptimos [**?**] y es la seleccionada en el presente trabajo como se explica en 1.1.2.

Para resolver un VRP es necesario definir qué se entiende por **solución**.

1.1.1. Representación de soluciones del VRP

Las soluciones son representadas (en su versión más simple) como una serie de listas de clientes denominadas rutas. 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 representa 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.

Encontrar la solución óptima exacta no es factible para instancias grandes. Para resolver VRP computacionalmente en un tiempo factible se utilizan metaheurísticas [**?**]. Específicamente, el presente sistema utiliza Metaheurísticas de Búsqueda Local [**?**].

1.1.2. Metaheurística de búsqueda local

Los algoritmos basados en búsqeuda local son aquellos en que se define una solución inicial y a partir de determinado criterio de vecindad se busca la solución óptima iterando por los vecinos de la vecindad formada por dicho criterio.

Por ejemplo, en VRP los criterios de vecindad consisten en conjuntos de movimientos de los elementos que conforman las rutas, ya sea dentro de una misma ruta o de unas hacia otras. A continuación se muestran algunos ejemplos de criterios de vecindad de VRP [**?**]:

1. Cambiar de posición a un cliente dentro de su ruta.
2. Mover a un cliente de ruta.
3. Intercambiar dos clientes de posición.
4. Cambiar vehículo de ruta.
5. intercambiar dos subrutas entre sí.
6. invertir orden de una subruta (porción de una ruta).

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

A los movimientos que conforman los criterios se les denomina operaciones de vecindad. Las operaciones pueden ser obtenidas a partir de un subconjunto de operaciones más simples a las que se denomina operaciones elementales. Entre las operaciones elementales se encuentran: **selección de ruta**, **selección de cliente** e **inserción de cliente** que comunmente se representan con los símbolos *r*, *a* y *b* respectivamente.

Por ejemplo *intercambiar dos clientes de posición* es un criterio de vecindad conformado por las operaciones:

1. Seleccionar ruta *r1*.

2. Seleccionar cliente **c1** de la ruta *r1*.

3. Seleccionar ruta *r2*.

4. Seleccionar cliente *c2* de la ruta *r2*

5. Intercambiar clientes *c1* y *c2* de posición.

Pero también por las operaciones:

1. Seleccionar ruta *r1*.

2. Seleccionar cliente **c1** de la ruta *r1*.

3. Seleccionar ruta *r2*.

4. Seleccionar cliente *c2* de la ruta *r2*

5. Insertar cliente *c1* en la ruta *r2* en la posición de *c2*

6. Insertar cliente *c2* en la ruta *r1* en la posición de *c1*

A partir de un criterio de vecindad y una solución inicial se pueden generar nuevas soluciones. El proceso de obtención y selección de nuevas soluciones se denomina exploración de la vecindad. La cardinalidad de una vecindad está dada por el número de vecinos que la forman. Mientras mayor la cardinalidad de la vecindad es más probable encontrar mejores soluciones [**?**], pero esto puede ser ineficiente computacionalmente [**?**]. Por tanto, las estrategias (de exploración de vecinos y selección de mejoras) a seguir durante la exploración de una vecindad son un factor vital a tener en cuenta para la creación del sistema implementado en el presente trabajo y se explicarán en 1.2 y 1.4. Además, a la hora de comparar el costo entre dos soluciones es necesario evaluarlas, lo cual puede poseer también costo computacional alto. Para la evaluación de soluciones se tiene el Grafo de evaluación explicado en 1.3.

Una buena herramienta para resolver problemas de VRP en la actualidad es la biblioteca OR-Tools (Google Optimization Tools)[**?**]. Un software de código abierto útil para problemas de optimización combinatoria entre los que se encuentra el Problema de Enrutamiento.

Un paso importante durante la exploración de una vecindad es la generación de los vecinos que conforman dicha vecindad. En el sistema implementado los vecinos se generan a partir de un Árbol de Vecindad.

1.2. Árbol de vecindad

La exploración de vecindades puede ser ineficiente y difícil de programar. En [6] se propone la representación de una vecindad mediante un árbol al que se denomina Árbol de Vecindad.

Todas las soluciones de una vecindad pueden ser obtenidas aplicando sobre la solución inicial el criterio de vecindad en cuestión de todas las formas posibles. A la asignación de valores (qué ruta se selecciona, qué índice tiene el cliente a seleccionar dentro de la ruta, etc) a los elementos que forman las operaciones de un criterio de vecindad se le llamará una instanciación de dicho criterio. Por ejemplo, dado el criterio **mover cliente** dado por las operaciones:

- Seleccionar ruta (r1).
- Seleccionar cliente (c1) de ruta (r1).
- Seleccionar ruta (r2).
- Insertar cliente (c1) en ruta (r2).

Un ejemplo de instanciación de este criterio sobre la solución s_1 se presenta como:

- $r_1 = 1 \rightarrow r_1$ es la ruta de la que el cliente c_1 es extraído.
- $c_1 = 1 \rightarrow c_1$ es el primer cliente de la ruta.
- $r_2 = 2 \rightarrow r_2$ es la ruta en la que el cliente c_1 será insertado.
- $c_1 = 2 \rightarrow c_1$ es insertado en la posición 2 de la ruta seleccionada.

A cada vecino de un criterio se le puede hacer corresponder una instanciación del mismo y por tanto, encontrar la cardinalidad de una vecindad es similar a encontrar el número de instanciaciones posibles del criterio asociado.

Al computar la cardinalidad de una vecindad del VRP, se utiliza una estrategia recursiva que consiste en contar para una operación el número de secuencias de operaciones instanciadas que se pueden formar con el resto de las operaciones del criterio, así como combinar las mismas con las posibles instanciaciones de la operación actual. Lo que se propone en [6] es utilizar dicha estrategia para almacenar toda la información necesaria para instanciar las operaciones que conforman la vecindad en una estructura

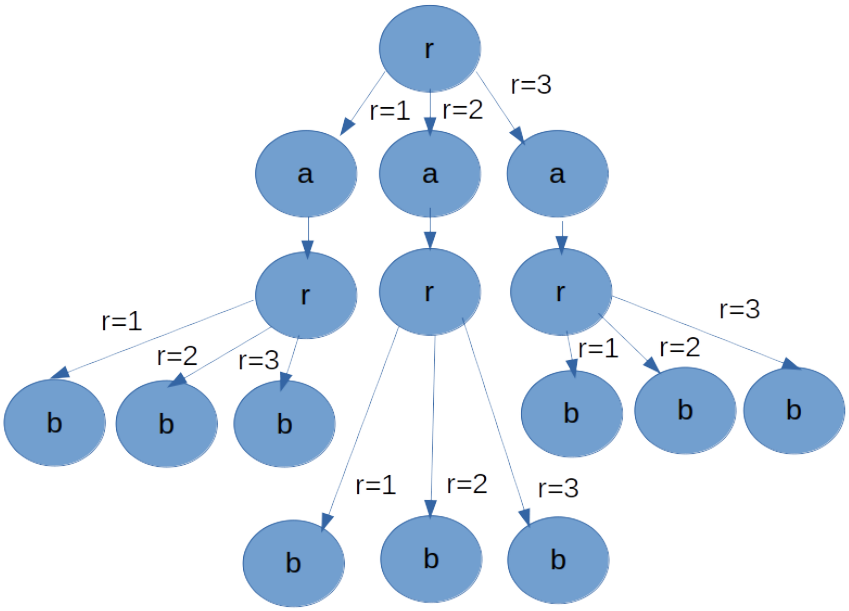

Figura 1.1: Árbol de vecindad asociado a *rarb*

arbórea que será llamada Árbol de Vecindad y que constituye una representación de la vecindad en cuestión mediante la que pueden ser generados todos los vecinos.

En la figura 1.1 se muestra una representación del árbol de vecindad asociado al criterio **mover cliente** (*rarb*).

Para obtener una instancia del criterio representado por un árbol y por tanto generar el vecino que se le corresponde basta con dar valores a (las operaciones representadas por) los nodos que conforman una rama del árbol. Por ejemplo, en el caso de la figura 1.1 qué ruta o cliente seleccionar o dónde insertar un cliente

Cada conjunto de asignación de valores a los nodos de una rama se asocia a un índice i con $1 <= i <= k$ siendo k la cardinalidad de la vecindad. Al indexar i en el árbol, este genera una solución.

Cada rama del árbol representa un conjunto de soluciones, por tanto, el conjunto de ramas representa una partición de la vecindad a la que está asociado dicho árbol. A los conjuntos hechos por esta partición se les de-

nomina regiones y resultan útiles para encontrar características compartidas en grupos de vecindades. Por ejemplo, es conveniente analizar cuáles son las regiones con soluciones de menor costo para intensificar en estas la búsqueda de soluciones óptimas.

Luego de generadas las soluciones a partir del Árbol de Vecindad, es necesario conocer sus costos. Para esto se utiliza un Grafo de Evaluación.

1.3. Grafo de Evaluación

Durante la exploración de las vecindades es generalmente necesario evaluar las soluciones que se van analizando (es posible, por ejemplo, querer contar los vecinos repetidos de una vecindad, en este caso no hay necesidad de evaluarlos). Ya sea para retornar inmediatamente una solución mejor a la inicial (estrategia de selección de primera mejora), para devolver la mejor solución (estrategia de selección de mejor vecino) o un vecino aleatorio entre todos los que mejoren la solución (estrategia de selección de mejor vecino aleatorio), en todos los casos hay que determinar el costo de las soluciones exploradas para analizar lo que es "mejor". Encontrar el costo de una solución es lo que se denomina como evaluar.

La evaluación de soluciones es potencialmente costosa. En el caso más simple de VRP se debe sumar las distancias entre cada cliente de cada ruta y el depósito. Agregar restricciones al problema implica análisis extra como la aplicación de penalizaciones a rutas con vehículos sobrecargados en el caso de CVRP [**?**].

El Grafo de Evaluación, propuesto por Jose Jorge Rodríguez en 1.3, permite evaluar soluciones vecinas a una solución inicial de forma eficiente ya que garantiza que sólo se recalculan los fragmentos de las nuevas soluciones en los que estas se diferencien de la solución inicial.

La estructura propuesta para evaluar una solución es una representación en forma de grafo de la evaluación dicha solución. Sus nodos están divididos en dos tipos: Nodos de alto nivel y nodos de bajo nivel. Los nodos de bajo nivel representan operaciones que reciben como entradas ciertos nodos de alto nivel y modifican con sus salidas otros nodos de alto nivel.

Por ejemplo, en la figura 1.2 se muestran los nodos de alto nivel del grafo que representa la solución: $s = [(c_1, c_2), (c_3, c_4)]$. Los nodos d representan al depósito y el nodo *cost* almacena el valor del costo total de la solución. Nótese que todas las rutas en el grafo comienzan y terminan con nodos depósito.

En 1.3 se muestra una representación del grafo completo para esta so-

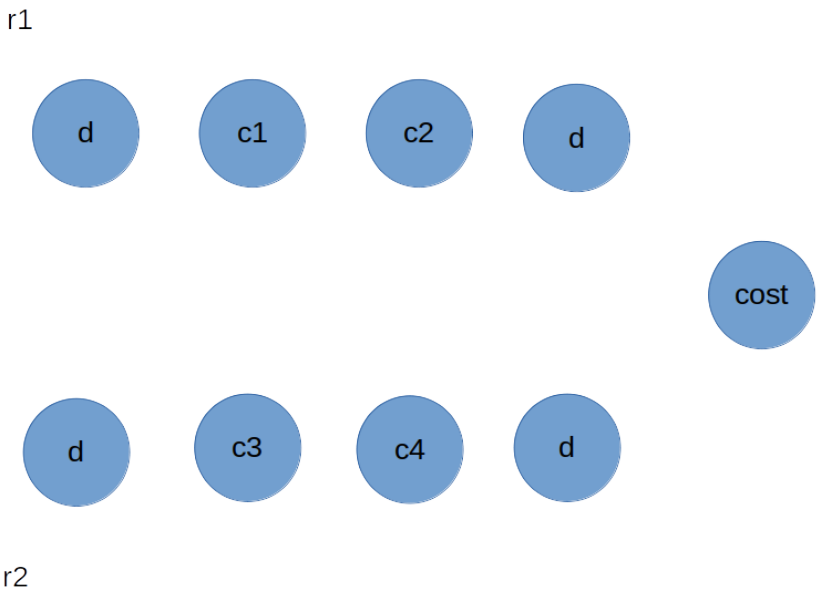

Figura 1.2: Nodos de alto nivel en un grafo de evaluación que representa la solución $s1$ de VRP clásico.

lución. Los nodos con símbolo de incremento son nodos de bajo nivel que toman como entrada dos nodos clientes (o depósito) y como salida adicionan al nodo de costo total la distancia entre ellos.

En 1.4 se transforma el problema en CVRP utilizando la misma solución. En este caso se agregan los nodos de alto nivel cap que tienen como valor inicial la capacidad del vehículo perteneciente a cada ruta. Los nodos de decremento reciben un cliente como entrada y, como salida, disminuyen la capacidad del vehículo en una cantidad igual a la demanda del cliente. Luego, los nodos *pen* (también de bajo nivel) reciben como entrada los nodos de capacidad y, en caso de tener estos valor negativo, como salida penalizan el costo total de la solución.

Todos los nodos de bajo nivel tienen asociado una función **evaluate** (para evaluar) y una función **undo** (para desevaluar) que son ejecutados cuando se agregan o remueven nodos de alto nivel. Por ejemplo, retirar un cliente $C1$ del grafo representado en 1.3 (VRP clásico) provoca que los dos nodos de incremento que utilizan dicho cliente como entrada se desevalúen y al mismo tiempo se crea un nuevo nodo incremento que recibe como entrada d y $c2$. Luego, insertar a $c1$ al final de la ruta $r2$ implicaría desevaluar el nodo incremento que recibe como entrada a $c4$ y d mientras se crean dos

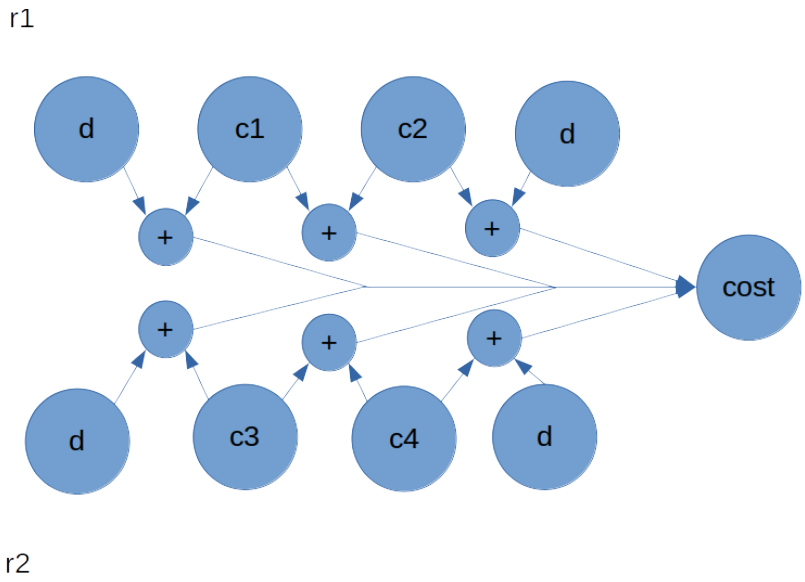

Figura 1.3: Grafo de evaluación que representa la solución $s1$ de VRP clásico.

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 1.5 y es, precisamente, el grafo resultante de aplicar la siguiente instanciación del criterio *rarb*:

- Seleccionar ruta (r1).
- Seleccionar cliente (c1) en ruta (r1).
- Seleccionar ruta (r2).
- Insertar cliente (c1) en posición (3) en 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 instancia del criterio *rarb*. 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 con la solución anterior y no todo el grafo. En esto se basa la evaluación .^{ef}iciente"del Grafo de Evaluación.

Para construir el grafo que representa la solución inicial el usuario debe ingresar el códio de evaluación de esa primera solución. Luego, los costos

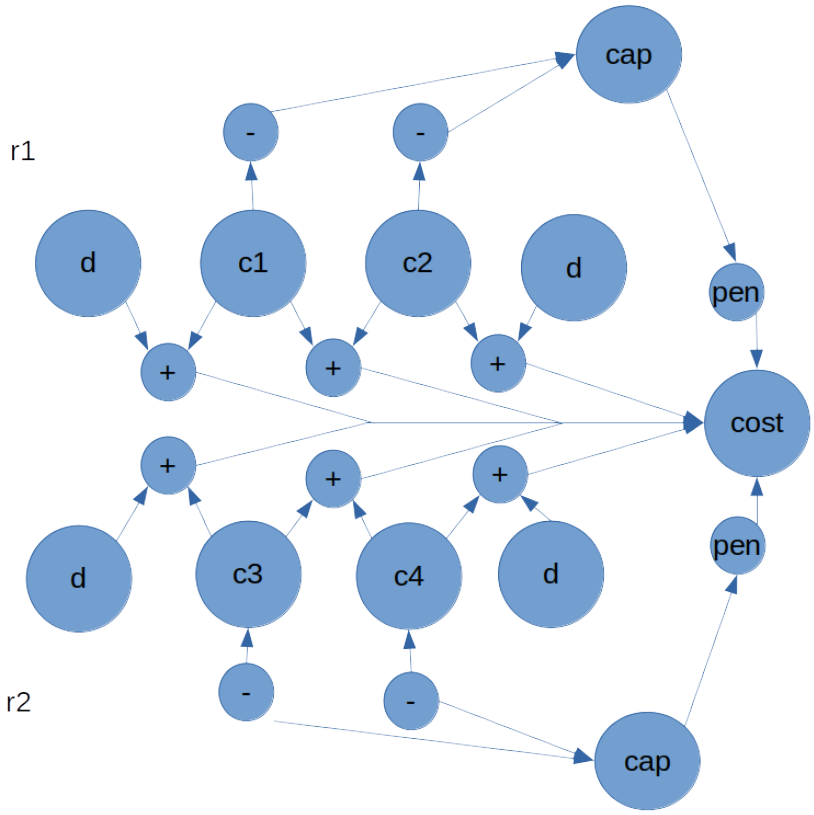

Figura 1.4: Grafo de evaluación que representa la solución $s1$ de VRP con restricciones de capacidad.

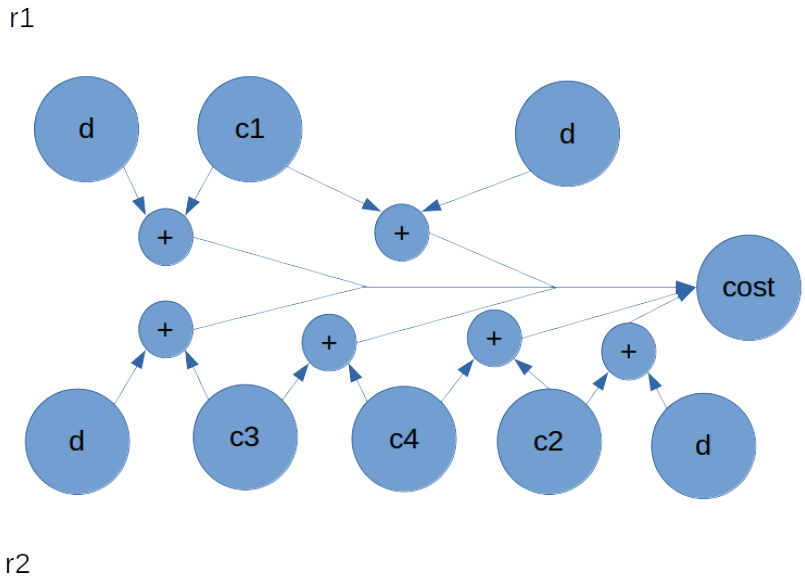

Figura 1.5: Grafo de evaluación que representa la solución $s1$ de VRP clásico luego de aplicada una instancia de *rarb*

el resto de las soluciones generadas son obtenidos al aplicar operaciones sobre el grafo inicializado.

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.

1.4. Combinación de estrategias de exploración y selección.

En el proceso de exploración de una vecindad se parte de una solución inicial, se generan soluciones vecinas a esta (con el Árbol de vecindad) y se evalúan para comparar y obtener mejores soluciones (Grafo de Evaluación). Al explorar, deben también tenerse en cuenta problemas tales como vecindades con cardinalidades tan grandes que provoquen exploraciones ineficientes o el retorno de mínimos locales[**?**]. Para una vecindad con muchas soluciones tal vez dé mejor resultado explorar no todos sus vecinos, sino una porción aleatoria de estos. Tal vez retornar siempre al mejor vecino

de cada vecindad pueda provocar el encuentro de un mínimo local que se hubiera evitado seleccionando aleatoriamente cualquier solución que mejorara la inicial [**?**].

Al espectro de búsqueda de vecinos en una vecindad se le denomina estrategia de exploración. Algunos ejemplos 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: En el momento en que se encuentra un vecino mejor que el inicial, este es retornado.
- 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.

La propuesta de Heidy Abreu en [3] permite generar funciones de exploración fabricadas a partir de un criterio, una estrategia de selección y una de exploración. Las estrategias se definen como clases que se pasan como instancias a la función generadora (función que genera funciones de exploración). La función de exploraciñon resultante recibe el problema que se quiere resolver junto con una solución inicial, ejecuta la exploración y retorna una solución mejor en caso de existir dentro de la vecindad definida por el criterio.

En el presente trabajo, el árbol de vecindad y el grafo de evaluación forman parte también de las funciones de exploración creadas. Un árbol de vecindad genera las soluciones y un grafo de evaluación se usa para evaluarlas. El grafo debe ser pasado como entrada también a la función de exploración.

La creación automática de funciones de exploración con distintas combinaciones de estrategias de exploración y selección se logra aprovechando sus características comunes y estructuras similares. El código generado en todas las funciones se reparte en cinco regiones que se unen para generar una función completa. Las regiones y el tipo de código que pertenece a cada una se explicarán en 2.5.

Para generar código crear funciones a partir de estrategias diferentes se utilizan características del lenguaje Common Lisp [**?**] tales como la herencia múltiple y su sistema de objetos [**?**].

1.5. Common Lisp y sus funcionalidades

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 lenguaje acepta herencia múltiple. Esta característica permite crear jerarquías con clases que implementan funcionalidades de varias clases superiores. Por ejemplo, la clase que representa la estrategia de selección de mejor vecino (**best-improvement**) hereda de una clase que indica retorno de mejor solución (**return-best-solution**) y de otra que indica el uso de un grafo de evaluación (**use-eval-graph**).

La herencia resulta especialmente útil para el presente trabajo cuando se combina con el sistema CLOS (Common Lisp Object System). Un mismo método puede tener numerosas implementaciones (especificaciones) que se ejecutan de acuerdo a los tipos de los parámetros proveídos como entradas. Además, también se ejecutan todas las especificaciones cuyos parámetros coincidan con los ancestros de los tipos de las entradas. Todas las especificaciones se combinan alrededor de un método primario conformando un único método con la unión de todos los códigos. El orden en que se combinan los métodos depende del orden de herencia y de los parámetros :before, :around y :after con que se definen.

Como ejemplo se tiene que para generar el código de la función de exploración para la estrategia de selección mejor vecino se une el código de métodos cuyo parámetro de search-strategy (estrategia de exploración) tenga tipo **best-improvement**, **return-best-solution**, **use-eval-graph** y cualquier otra clase que herede de alguna de estas.

En el siguiente capítulo se explica cómo está orgainzado el sistema im-

plementado, sus características y cómo fueron ensambladas sus piezas.

Capítulo 2

Propuesta de solución

En este capítulo se describe el sistema implementado y cómo fueron unidas todas las piezas aisladas.

En 2.1 se explica las ventajas del formato .org utilizado para desarrollar los códigos y exportarlos a archivos .lisp. También se explica la estructura general del sistema.

El sistema está separado en secciones.

En 2.2 se presenta la sección central que representa los cimientos sobre los que se construyó el sistema.

En 2.3 se presenta el Árbol de Vecindad y el generador de soluciones.

En 2.4 se presenta el Grafo de Evaluación.

En 2.5 se presenta la función generadora de código fuente para explorar vecindades utilizando combinaciones de estrategias.

2.1. Formato org

El sistema está desarrollado sobre una serie de archivos con formato org (Lotus Organizer File). Org es un modo Emacs para guardar notas, mantener listas TODO, y hacer planificación de proyectos con un rápido y efectivo sistema de texto plano. Es también un sistema de publicación y autoría, que soporta trabajar con código fuente para programación literal e investigación reproducible [**?**].

Org facilita la organización de los archivos facilitando la separación de los datos en regiones. Los títulos de cada región comienzan con una cierta cantidad de símbolos "$*$". Las regiones se ramifican en subregiones de acuerdo a la cantidad de asteriscos.

El texto plano en los archivos contiene información y explicaciones de los elementos implementados. Además, Org-mode proporciona un número de funcionalidades para trabajar con código fuente, incluyendo la edición, evaluación y exportación de bloques código.

La estructura de bloques de código es como sigue:
#+NAME: <name>
#+BEGIN_SRC <language> <switches> <header arguments>
<body>
#+END_SRC

Donde <name> es una cadena usada para nombrar el bloque de código <language> que especifica el lenguaje del bloque de código (lisp en este caso) <switches> puede usarse para controlar la exportación del bloque de código, <header arguments> puede usarse para controlar muchos aspectos del comportamiento de bloques de código, y el <body> contiene el código fuente actual.

El argumento de cabezera : *tangle* $<$ *archivo* $>$ permite exportar los bloques de código al archivo seleccionado. en este caso, los códigos se exportan a archivos con formato *.lisp* que son creados en la carpeta *src* ubicada en la raíz del sistema. Los archivos de *src* contienen todas las funciones que deben ser importadas para utilizar el sistema. El proceso de inicialización del sistema se explica en 3.

La implementación sistema está dividida en cuatro secciones principales.

- *Core*: Contiene funciones para inicializar el sistema, datos de instancias conocidas de VRP y criterios clásicos, implementaciones de algoritmos de búsqueda local y definiciones de clases a partir de las cuales es posible definir un problema de VRP en lenguaje Common Lisp. También se definen las operaciones a partir de las que se forman criterios de vecindad.

- *Neigh*: En los archivos de *neigh* se implementa el árbol de vecindad a partir del cual se generan soluciones.

- *Eval*: En *eval* se implementa el grafo de evaluación con el que se evalúan soluciones.

- *Blueprint*: Esta sección contiene código para generar funciones de exploración a partir de combinaciones de estrategias de exploración y selección.

Para resolver computacionalmente un problema, el primer paso es definir una forma para describirlo a partir de código. La sección **core** es la base del sistema.

2.2. Vrp-Core

En *core* se definen funciones útiles para inicializar fácilmente el sistema, datos de instancias conocidas de VRP y criterios clásicos útiles para la investigación, así como la implementaciones de algoritmos de búsqueda local que utilizan funciones del sistema para la exploración de vecindades. Estos elementos serán analizados a profundidad en 3.

Se definen además una serie de clases a partir de las cuales es posible definir problemas de VRP con sus especificaciones y restricciones en lenguaje Common Lisp. También se crean clases para representar operaciones que conforman los criterios de vecindad (como seleccionar ruta o insertar cliente).

Los elementos que forman un problema están definidos en clases. Los elementos que presenta un VRP en sus versiones más simples son:

1. Clientes
2. Vehículos
3. Depósitos

Siendo las variaciones de VRP sólo limitadas por la imaginación, las características específicas de estas variaciones son también virtualmente infinitas. Por ejemplo, para el Problema de Enrutamiento de Vehículos con Múltiples productos [**?**], se ha definido también *Producto como una característica*. Cómo ampliar el sistema para definir otras variantes de VRP se discute en 3. También se tienen clases para representar rutas y soluciones.

Para crear una instancia de un problema de VRP deben instanciarse elementos acordes a sus especificidades. Por ejemplo, una instancia de VRP clásico utiliza clientes básicos, mientras que una de CVRP trabaja con clientes con demanda.

Se utiliza la herencia múltiple de Common Lisp para definir las características de cada elemento mediante clases abstractas. Por ejemplo, un vehículo de CVRP (**cvrp-vehicle**) se define como una clase que hereda de las clases abstractas: **basic-vehicle** (vehículo básico), **capacity-vehicle** (vehículo con capacidad) y **cargo-vehícle** (vehículo que lleva carga).

Una vez definidas las clases que representan un problema y sus elementos, estas pueden ser utilizadas en el resto de las secciones del sistema. En la sección d *neigh* se define el Árbol de Vecindad con generador de soluciones.

2.3. Vrp-neigh

En esta sección se define el Árbol de vecindad como una clase que almacena sus propiedades, tales como el problema, la solución y el criterio de vecindad que este representa. Se define también una clase para las soluciones de conteo (explicada en 1.2) distinta a la clase solución de *vrp-core*.

La función **build-neighborhood-tree** recibe una solución y una lista de operaciones que representa el criterio de vecindad que se quiere explorar. Esta función instancia, construye y retorna un Árbol de vecindad

Dado un Árbol de Vecindad T con cardinalidad k, a cada elemento perteneciente a la vecindad representada se le hace corresponder un número entero *index*, con:

$$1 <= index <= k$$

Además, la función **from-index-to-coordinate-list** permite obtener el vecino correspondiente a cada índice. Los vecinos son retornados como una lista que contiene las operaciones a realizar sobre la solución inicial. La forma de generación de soluciones durante una exploración varía de acuerdo a la estrategia de exploración seleccionada. Por ejemplo, se puede generar todos los vecinos de forma exhaustiva o, en cambio generar una cantidad fija c de vecinos aleatorios.

Las funciones de exploración (como **exhaustive-exploration** o **random-exploration**) devuelven, a partir de un árbol de vecindad una función *lambda* generadora de soluciones. Cada vez que se invoque esta función, se recibirá la lista de operaciones que representan una solución, o **nil** en caso de haber llegado a la condición para terminar la exploración. Esta condición varía según el tipo de exploración, por ejemplo, para la exploración exhaustiva es generar k soluciones mientras que en la exploración aleatoria es generar c.

A partir del árbol de operaciones se hizo generadores de soluciones que construyen soluciones vecinas de una operación inicial en forma de lista de operaciones. Al aplicar las operaciones de una lista sobre la solución inicial se puede formar una nueva solución que, al ser evaluada retorna un costo. El costo de la nueva solución se utiliza durante la exploración para buscar optimizaciones. Evaluar una solución es un proceso potencialmente costoso y hacerlo para soluciones de vecindades con cardinalidades grandes

puede ser computacionalmente intractable. Para evaluar eficientemente las soluciones generadas se utiliza un Grafo de Evaluación.

2.4. Vrp-eval

En esta sección se implementa el Grafo de Evaluación. La clase **eval-graph** almacena el estado del grafo y lleva constancia de sus nodos. También, mediante tablas de hash hacen corresponder cada nodo con su respectivo elemento del problema. Se define también una clase para cada tipo de nodo existente, de alto y bajo nivel.

Para inicializar un grafo se necesita sólo de una solución inicial. La inicialización consiste en convertir los elementos de una solución (clientes, depósitos, vehículos) en nodos de alto nivel que están inicialmente aislados (un grafo de evaluación es bipartito [6], sólo se conectan nodos de diferentes tipos). Los métodos **convert-to-node** utilizan el sistema CLOS para crear nodos de alto nivel correspondientes a los elementos pasados como parámetros. Por ejemplo, al llamar a conver-to-node con un objeto de tipo **basic-client** como argumento, se crea un nuevo nodo que se agrega al grafo y, en una tabla de hash, se hace corresponder al objeto con el nuevo nodo.

Los nodos creados durante la inicialización del grafo se clasifican en nodos de tipo *entrada*. Terminar la construcción del grafo requiere de la ejecución del código de evaluación de la solución inicial. A partir de este código se crean los nodos de alto nivel restantes (nodos de tipo *salida*). También se crean y evalúan los nodos de bajo nivel que unen a los nodos de alto nivel. Nótese que los nodos de tipo *salida* pueden constituir salidas parciales que son al mismo tiempo *entradas* de otros nodos de bajo nivel.

El código de evaluación se hace utilizando los métodos de construcción de grafo definidos en el sistema tales como:

- **defvar**: Recibe un nombre y un valor inicial. Inicializa un nuevo nodo de alto nivel de tipo *salida* y lo asocia (mediante una tabla de hash) al nombre recibido.
- **increment-distance**: Recibe dos clientes, una variable asociada a un nodo tipo *salida* y la matriz de distancia. Crea y evalúa un nodo de bajo nivel que recibe los nodos asociados a los clientes como entrada e incrementa el valor almacenado en el nodo salida un una cantidad igual a la distancia entre ellos.
- **decrement-demand**: Recibe un cliente y una variable asociada a un nodo tipo *salida*. Crea y evalúa un nuevo nodo de bajo nivel que recibe

como entrada el cliente y decrementa el valor del nodo *salida* en una cantidad igual su demanda

- **increment-value**: recibe una variable y un valor. Incrementa el valor almacenado en el nodo asociado al nombre en una cantidad igual al valor recibido.
- **apply-penalty**: Recibe dos variables asociadas a nodos *salida* y un factor de penalización. Crea y evalúa un nuevo nodo de bajo nivel que recibe como entrada el nodo asociado a la primera variable y, en caso de este almacenar un valor negativo, aumenta el valor almacenado en el nodo asociado a la segunda variable en una cantidad dependiente del factor de penalización.
- **return-value**: Recibe el nombre de una variable y marca su nodo asociado en el grafo como el nodo contenedor del valor de salida. A partir del punto en que se invoca, el slot *output* del grafo referenciará a este nodo.

Todas las funciones de construcción de grafo reciben también como parámetro la instancia del grafo sobre el que se está trabajando.

La ampliación del sistema con nuevos métodos se discutirá en 3. A continuación se muestra el código de evaluación de una solución de VRP clásico.

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

En la línea *1* se usa def-var para inicializar la variable que almacenará 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 *5*). 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)(líneas *9*). Después de analizada cada ruta se aumenta el costo total de la solución en una cantidad igual a los costos de las mismas (líneas *12*). Finalmente se retorna la distancia total.

Una vez construido el grafo a partir de la solución inicial y el código de evaluación, es posible evaluar eficientemente soluciones vecinas ejecutando la función **do-core-operations**. Esta función recibe una lista de operaciones y el grafo sobre el cual estas serán realizadas. Las operaciones se realizan en orden, los nodos de alto nivel se insertan o retiran y los nodos de bajo nivel correspondientes son evaluados o desevaluados. Por ejemplo, dado el grafo representado por 1.3, al ejecutar el siguiente código:

```
1 (do-core-operations graph (list (op-select-client 1 1 0)
2                                 (op-insert-client 2 3 0))
```

Se obtiene el grafo representado en 1.5. En este punto, todos los nodos necesarios fueron evaluados y desevaluados, y, por tanto, en el slot *output* del grafo (Que en este caso hace referencia al nodo *cost*) está almacenado el costo total de la nueva solución.

Para deshacer operaciones realizadas sobre un grafo se ejecuta el método **undo-core-operations**. Por ejemplo, al ejecutar el siguiente código sobre el grafo representado en 1.5:

```
1 (un-core-operations graph (list (op-select-client 1 1 0)
2                                 (op-insert-client 2 3 0))
```

Se obtiene nuevamente el grafo representado por 1.3 y el valor almacenado en *cost* vuelve a ser el costo total de la solución inicial.

Una vez definidas las funciones que generan y evalúan soluciones, sólo resta utilizarlas para la exploración de vecindades.

2.5. Vrp-blueprint

Combinar diferentes estrategias de exploración y selección puede dar como resultado numerosas exploraciones con resultados distintos. Programar cada una de estas exploraciones es un proceso con alto consumo de tiempo. En [3] se propone una forma de automatizar el proceso de construcción de funciones exploradoras. Se aprovecha las ventajas del lenguaje Common lisp para generar código fuente y el sistema CLOS, y se genera el código de una función lambda. El código generado depende de los tipos de estrategias seleccionados.

La función **make-neighborhood-criterion** (función generadora) recibe un criterio de vecindad (en forma de lista de operaciones), una estrategia de exploración y una estrategia de selección. Se retorna una función lambda que al ser invocada retorna la solución encontrada durante la exploración en forma de objeto de tipo **solution** o *nil* en caso de no encontrarse ninguna.

La función lambda (función de exploración) recibe como entrada la solución inicial, el problema y el Grafo de Evaluación que representa a la solución inicial. Luego de invocada, la instancia del grafo tendrá aplicadas las operaciones que dan como resultado la solución encontrada, que pasará a ser la solución inicial de próximas operaciones realizadas sobre el grafo. En caso de no encontrarse solución, el grafo no cambiaría.

Las estrategias se definen como clases cuyas instancias son pasadas como argumentos a la función generadora. Se utilizan métodos que reciben estas instancias para generar código. El código que se genera depende de los métodos especializaciones que se ejecutan y estos métodos dependen de los tipos de los argumentos recibidos, como se explicó en 1.5.

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.

La extensión del sistema a partir de nuevas estrategias se discutirá en 3.

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

El siguiente código de ejemplo genera una función lambda que explora el criterio *rarb* de forma exhaustiva y retornando mejor solución:

```
1       (setf rarb (make-neighborhood-criterion
2                       `((select-route r1)
3                         (select-client c1 from r1)
4                         (select-route r2)
5                         (insert-client c1 in r2))
6                       +exhaustive-search-strategy+
7                       +best-improvement+)))
```

Luego, la función generada **rarb** puede ser invocada para ejecutar la exploración y obtener el mejor vecino encontrado.

```
1     (setf best-neighbor (funcall rarb solution problem graph))
```

Capítulo 3

Método de uso

Este capítulo explica los pasos que debe seguir un cliente para resolver variantes de VRP utilizando el sistema desarrollado.

3.1. Inicialización

El presente sistema está diseñado para ser ejecutado en Emacs, un editor de texto con una gran cantidad de funciones. Desde Emacs se carga el archivo *vrp-emacs.el* ubicado en la raíz del sistema. Este archivo define varias funciones útiles.

Ejecutar la función **tangle-all-files** exportará todos los bloques de código de los archivos formato .org ubicados en *vrp* a sus respectivos archivos formato .lisp dentro de *src*. Ambas direcciones se ubican en la raíz del sistema.

Teniendo los archivos exportados, la función **load-system** inicializará un buffer de slime, importará todas las funciones del sistema y establecerá :vrp como el package actual.

Una vez levantado el sistema es posible ejecutar todas las funciones del sistema.

3.2. Ejecución

Para definir el problema que se quiere resolver se debe instanciar las clases que representan sus características. Estas clases fueron definidas en **vrp-core** y explicadas en 2.2. Por ejemplo, para definir una instancia de CVRP se necesita crear un objeto de tipo **basic-cvrp-client** (cliente con demanda)

por cada cliente del problema, un objeto de tipo **cvrp-vehicle** (vehículo con capacidad), uno de tipo **basic-depot** (depósito) y una matriz de distancias de tamaño nxn siendo n el número de clientes.

En el ejemplo siguiente muestra la creación de una instancia de CVR con 6 clientes y dos vehículos.

```
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 v1 (cvrp-vehicle 1 20))
9  (defparameter v2 (cvrp-vehicle 2 20))
10
11 (defparameter d0 (basic-depot))
12
13 (defparameter dist-mat #2A((0 1 2 3 4 5 6)
14                            (1 0 5 2 1 3 2)
15                            (2 5 0 2 2 2 2)
16                            (3 2 2 0 1 2 1)
17                            (4 1 2 1 0 2 3)
18                            (5 3 2 2 2 0 1)
19                            (6 2 2 1 3 1 0)))
20
21 (defparameter problem (finite-fleet-cvrp-problem
22                            :id 1
23                            :clients (list c1 c2 c3 c4 c5 c6)
24                            :depot d0
25                            :distance-matrix dist-mat
26                            :fleet (list v1 v2)))
```

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 depísito (línea 11), matriz de distancias (línea 13) y se crea la instancia del problema (línea 21).

Una vez definido el problema, debe crearse también una solución inicial. Esta será utilizada para construir e inicializar el grafo. También será la

solución inicial del algoritmo de búsqueda local a partir de la cual se generarán vecinos en busca de mejoras. En el caso de CVRP una solución está formada por una lista de rutas. El siguiente código muestra un ejemplo de solución inicial.

```
1  (defparameter r1 (route-for-simulation
2                    :id 1
3                    :vehicle v1
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 v2
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 son definidas como instancias de la clase **rute-for-simulation** (líneas 1 y 7). Esta clase es descendiente de la clase **basic-route** y la extiende con el slot *previous-client*. Este slot comienza guardando una referencia al depósito a partir de la cual se crea el primer nodo de la ruta en el grafo de evaluación y es útil para facilitar la escritura del código de evaluación de la solución. Nótese que las rutas del grafo de evaluación comienzan y terminan en el depósito, este debe ser clonado en cada posición de la ruta para crear un nodo distinto en cada una.

Una vez construida la solución inicial de debe inicializar el grafo ejecutando la función **init-graph**.

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

En este punto se dispone de un grafo inicializado que debe terminar de ser construido ejecutando el código de evluación. El siguiente es el código

de evaluación de una solución de CVRP:

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

En la línea *1* se usa def-var para inicializar la variable que almacenará 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 *5*) y otra que almacena la capacidad restante de su vehículo. 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 *9* y *10*). Después de analizada 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 *12* y *13*). Finalmente se retorna la distancia total.

En **vrp-core** están implementados algunos algoritmos de búsqueda local. Estos utilizan funciones de exploración con árbol de vecindad y grafo de evaluación para realizar sus búsquedas. Por ejemplo, la función de Búsqueda de Vecindad Variable [**?**] (VNS) recibe la instancia del problema, una lista de funciones de exploración (una por cada criterio a aplicar), el Grafo de Evaluación y el número máximo de iteraciones a realizar.

Antes de ejecutar la función de VNS se debe generar las funciones de exploración de acuerdo a los criterios que vayan a ser utilizados. Por ejemplo, a continuación se define una lista de funciones de exploración para los

criterios *rab*, *rarb* y *rarac*. 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
    1000000))
```

El código completo se muestra a continuación:

```
(progn
(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 v1 (cvrp-vehicle 1 20))
(defparameter v2 (cvrp-vehicle 2 20))

(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
                       :fleet (list v1 v2)))

(defparameter r1 (route-for-simulation
                  :id 1
                  :vehicle v1
                  :depot d0
                  :clients (list c1 c2 c3 (clone d0))
                  :previous-client (clone d0)))
(defparameter r2 (route-for-simulation
                  :id 2
                  :vehicle v2
                  :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))

(progn
   (def-var total-distance 0 graph)
   (loop for r in (routes s1) do
      (progn
         (def-var route-distance 0 graph)
         (def-var route-demand (capacity (vehicle r)) graph)
         (loop for c in (clients r) do
            (progn
               (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)))



(progn
(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+))

```

```
90 (setf criteria (list rab rarb rarac))
91
92 (setf result (vns-vrp-system problem criteria graph :max-iter
       10000000000)))
```

Se ha logrado resolver una variante de VRP utilizando las funciones creadas al unir generación de funciones de exploración con Árbol de Vecindad y Grafo de Evaluación, sin embargo, la cantidad de variaciones que pueden ser creadas es practicamente ilimitada. El presente sistema puede ser fácilmente extendido para acomodarse a las necesidades del usuario y resolver (casi) cualquier variante de VRP.

Capítulo 4

Extensibilidad

El sistema de solución de VRP desarrollado puede resolver de forma nativa varias de las variantes de VRP más conocidas. Sin embargo, en algún momento, un usuario potencialmente deberá resolver un problema no soportado nativamente. En este capítulo se explica cómo extender el sistema para otras variantes.

El primer paso a realizar es la extensión de las clases que describen el problema.

4.1. Descripción del problema

Para extender el sistema a nuevas variantes de VRP puede ser necesario definir nuevas clases mediante las cuales sea posible su descripción. Las especificaciones de cada característica del problema depende de las clases estructurales de las que estas hereden. 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 estructurales debe heredar para satisfacer sus especifiaciones y, en caso de ser necesario, crear nuevas clases estructurales. Definir un problema nuevo puede implicar la creación de varias clases para representar las características de este.

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 ser visitados,

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 definir primero clientes que conozcan sus ventanas de tiempo y el tiempo que debe consumir el vehículo atendiéndolos. Se definen las clases estructurales siguientes:

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

Entonces se crea la clase **basic-tw-client** que representa un cliente de TWVRP y hereda de:

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

Además, se deben definir nuevas rutas que conozcan el tiempo actualmente consumido. Se crea la clase estructural **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 abstracta **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**
- **distance-problem**
- **capacity-problem**
- **time-problem**

Junto con las nuevas características, definir un nuevo problema implica también definir cómo son evaluadas sus soluciones. Diferentes evaluaciones pueden necesitar del agrego de nuevos tipos de nodos y funciones de construcción para el Grafo de Evaluación.

4.2. Evaluación

TODO: CAMBIAR EL EJEMPLO

Al definir nuevos problemas, también se debe definir la forma en que sus soluciones son evaluadas. Esto implica tener que extender el Grafo de Evaluación para satisfacer nuevas características y restricciones añadidas. Extender el grafo se traduce en agregar los nuevos nodos y funciones de construcción de grafo que sean necesarios.

Por ejemplo, el código de evaluación de una solución de TWVRP se muestra a continuación:

```
(progn
   (def-var total-distance 0 graph)
   (loop for r in (routes s1) do
      (progn
      (def-var route-distance 0 graph)
      (def-var route-demand (capacity (vehicle r)) graph)

      (def-var route-time 0 graph)
      (def-var route-time-penalizer 0 graph)

      (loop for c in (clients r) do
         (progn
         (increment-distance (previous-client r) c route-distance
             dist-mat graph)
         (decrement-demand c route-demand graph)

         (increment-time (previous-client r) c route-time time-mat
             graph)
         (increment-time-penalizer c route-time route-time-penalizer)

      (setf (previous-client r) c)))
      (increment-value total-distance route-distance graph)
      (apply-penalty route-demand total-distance 10 graph))
   (apply-penalty route-time-penalizer total-distance 10 graph)
   (return-value total-distance graph)))
```

La diferencia con el código de evaluación de CVRP está en las nuevas variables que almacenan el tiempo que ha consimido la ruta por y la penalización de la ruta por atender clientes fuera de sus ventanas. Estas variables

son actualizadas durante la exploración de las rutas y finalmente se penaliza el costo total en caso de haber atendido clientes fuera de tiempo.

4.3. Estrategias

El sistema cuenta de forma nativa con varias estrategias de exploración y selección, mencionadas en 2.5 que al ser combinadas generan muchos tipos de exploraciones distintas. Es posible también extender la generación de funciones de exploración al definir nuevos tipos de estrategias.

La definición de nuevas estrategias lleva dos pasos. Primero debe crearse la clase que representa la estrategia siendo definida. En dependencia del comportamiento esperado, la nueva clase puede heredar de clases auxiliares ya creadas o incluso de nuevas clases auxiliares que deben ser definidas. El segundo paso es implementar las especializaciones de métodos de cada tipo (inicializaciones dentro del *let*, código dentro del ciclo, código de retorno, etc) necesarias que reciban como parámetros las nuevas clases creadas. Opcionalmente se puede asociar una instancia de la nueva clase 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, se creará una estrategia de selección nueva tal que se retorne un vecino aleatorio de una lista de vecinos encontrados, pero esa lista sólo tendrá soluciones cuya mejora superen cierto margen con respecto a la inicial. 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 un slot **aceptance** para el margen de aceptación cuyo valor estará entre 0 y 1. Se aceptarán soluciones que cumplan:

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

Debe heredar de las siguientes clases auxiliares existentes:

- **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 nueva clase auxiliar:

- **has-restricted-candidates-for-best-neighbor**: Tiene un comportamiento parecido a **has-restricted-candidates-best-neighbor** pero los candidatos sólo serán agregados si superan 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 de estrategia de selección 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** son creadas 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.

Conclusiones

Recomendaciones

Bibliografía

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

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

[3] Heidy Abreu Fumero. Exploración de una vecindad para una solución de vrp combinando diferentes estrategias de exploración y de selección. 2019.

[4] Camila Pérez Mosquera. Primeras aproximaciones a la búsqueda de vecindad infinitamente variable. 2017.

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

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

[7] 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.