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 millonari.

El problema de enrutamiento de vehículos (VRP por sus siglas en inglñés) es un problema de optimización combinatoria cuyo objetivo en su forma más simple es, dado un conunto 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 instancias no pequeñas, por tanto, se utilizan técnicas no exactas como heurísticas y metaheurísticas que han sido objeto de estudio por décadas.

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 muchísimo 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 visión 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 1.2 se muestran las bibliotecas de clases existentes hasta el momento que pueden ser utilizadas para encontrar soluciones a instancias de VRP.

En 1.3 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 realizar una exploración de dos fases que utiliza técnincas estadísticas.

En 1.4 se expone el concepto de Grafo de evaluación y cómo es esto utilizado para evaluar soluciones de forma eficiente y automática.

En 1.5 se presenta un mecanismo para explorar vecindades de forma automática a partir de combinaciones de cualesquiera estrategias de exploración y selección.

Finalmente en 1.6 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, el problema 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 de grandes tamaños. Para buscar aproximaciones a la solución se utilizan heurísticas y metaheurísticas. Se destaca la búsqueda local como metaheurística que ha dado muy buenos resultados y es la seleccionada en el presente trabajo como se explica en 1.1.2.

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 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.

1.1.2. Metaheurísticas de búsqueda local

Los algoritmos basados en búsquda 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. A continuación se muestran algunos ejemplos de criterios de vecindad:

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.

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.

Estas 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**.

Por ejemplo, **intercambiar dos clientes de posición** puede ser realizado a partir de dos **selección de ruta**, dos **selección de cliente** y dos **inserción de cliente**.

La importancia de las operaciones elementales para la implementación del sistema se explicarán en con más profundidad en **??**.

A partir de un criterio y una solución inicial se pueden generar nuevas soluciones. El proceso de obtención y selección de nuevas soluciones se denomina exploración de la vecindad. Mientras más grande la vecindad es más probable encontrar mejores soluciones, pero esto puede requerir una gran cantidad de tiempo. Por tanto, las estrategias a seguir durante la exploración son un factor vital a tener en cuenta para la implementación del sistema y se explicarán en 1.3 y 1.5. Además, a la hora de comparar el costo entre dos soluciones es necesario evaluarlas, lo cual posee también un costo computacional considerable. Para la evaluación de soluciones se tiene el Grafo de evaluación explicado en 1.4.

1.2. Vías de solución existentes

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.

NOTA PARA FERNANDO: Profe, no sé qué más poner aquí XD.

1.3. Árbol de vecindad y exploración de dos fases

La exploración de vecindades puede ser ineficiente y difícil de programar. La idea que se propone es utilizar técnicas estadísticas para analizar cuáles son las mejores regiones de las vecindades para intensificar la búsqueda en estas.

Buscando aplicar dichas técnicas estadísticas es necesario saber la cardinalidad de las vecindades y separarlas en regiones. Para lograr esto de forma eficiente (sin iterar por todos los elementos de una vecindad) se propone la creación de un Árbol de vecindad. El Árbol de vecindad utiliza oun concepro de *solucion* diferente al explicado en 1.1.1. Previamente se definió una solución de VRP en su forma más trivial como una lista de caminos conformados a su vez por la lista de clientes que se visitan en cada uno, por ejemplo:

$$s_1 = [(c_2, c_3), (c_1, c_4, c_5), (c_6)] \tag{1.2}$$

Con el propósito de contar la cantidad de soluciones que tiene una vecindad, pierde importancia saber qué clientes se visita en cada ruta, en cambio sólo es necesario conocer la cantidad de clientes visitados en cada una. A este tipo de solución se le denomina solución de conteo y aplicado a s_1 daría como resultado:

$$sc_1 = [2, 3, 1] \tag{1.3}$$

Teniendo una solución inicial, todas las soluciones de una vecindad pueden ser obtenidas aplicando sobre esta el criterio de vecindad en cuestión, con todos sus valores posibles. A la asignación de valores de un criterio de vecindad sobre una solución se le llamará una instanciación de dicho criterio. Por ejemplo, dado el criterio **mover cliente** dado por:

- Seleccionar ruta (r).

- Seleccionar cliente en ruta (a).
- Seleccionar ruta (r).
- Insertar cliente en ruta (b).

Las letras r, a y b son los símbolos representativos de cada operación.

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.position = $1 \rightarrow c_1$ es el primer cliente de la ruta.
- $r_2 = 1 \rightarrow r_2$ es la ruta de la que el cliente c_1 será insertado.
- c_1.position = $2 \rightarrow c_1$ 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 generar las distintas instanciaciones de un criterio de vecindad para una solución dada se cumple que muchas de estas comparten una secuencia común de operaciones instanciadas. La estrategia propuesta se basa en agrupar aquellos criterios instanciados para los cuales dicha secuencia común comience en la primera operación de los mismos, pues de esta forma el resto de las operaciones de tales criterios instanciados no se ven afectadas, y contar para cada una de estas el número de posibles secuencias de operaciones instanciadas que unidas con la secuencia común forman un criterio instanciado.

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 aprovechando que para las operaciones modificadoras estas secuencias son comunes a todas las posibles instanciaciones de las mismas.

Es posible utilizar este algoritmo, para almacenar toda la información necesaria para cualquier procesamiento posterior sobre dicha vecindad en

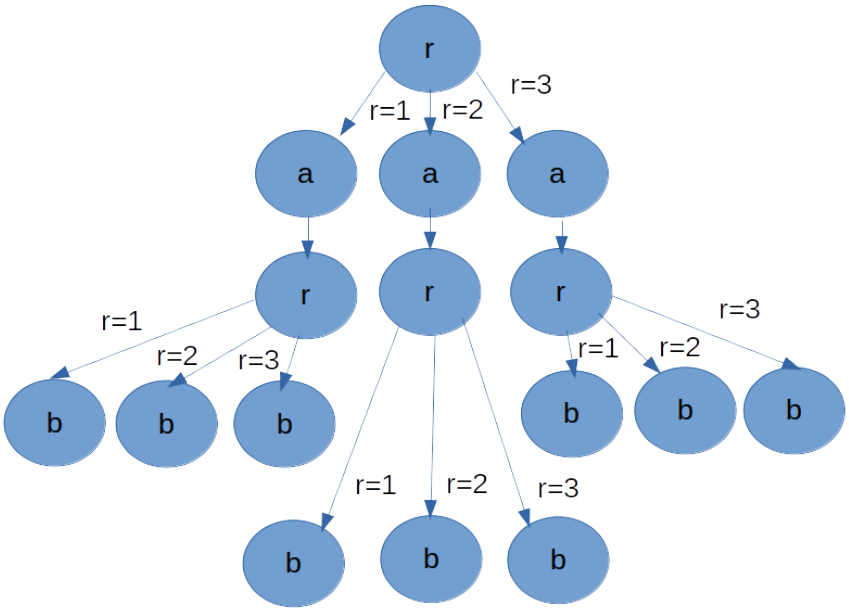

Figura 1.1: Árbol de vecindad asociado a *rarb*

una estructura arbórea que será llamada árbol de vecindad y que constituye una representación de la vecindad en cuestión. En 1.1 se muestra una representación del árbol de vecindad asociado a l criterio **mover cliente** (*rarb*). Cada nodo del árbol, representa una operación de vecindad y almacena toda la información necesaria para instanciar dicha operación y de esta forma, a partir del proceso que computa la cardinalidad de la vecindad, se puede pasar a generar todas las soluciones de la misma.

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 denomina regiones y resultan útiles para encontrar características compartidas en grupos de vecindades. Por ejemplo, es conveniente analizar cuáles son las regiones con mejores soluciones 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.

1.4. Grafo de Evaluación

Durante la exploración de las vecindades es (generalmente) necesario evaluar las soluciones que se van analizando. 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 se debe sumar las distancias entre cada cliente de cada ruta y el depósito. Agregar restricciones 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 1.4, 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 es una representación en forma de grafo de la evaluación de la función objetivo en una determinada solución. Sus nodos están divididos en dos tipos: Nodos de alto nivel y nodos de bajo nivel. Los nodos de bajo nivel representan operaciones (incremento, decremento) que reciben como entradas ciertos nodos de alto nivel y modifican con sus salidas otros nodos de alto nivel.

Por ejemplo, en 1.2 se muestran los nodos de alto nivel del grafo que representa la solución:

$$s = [(c_1, c_2), (c_3, c_4)] \tag{1.4}$$

Nótese que todas las rutas en el grafo comienzan y terminan con el nodo que representa al depósito. El nodo *cost* almacena el valor del costo total de la solución.

En 1.3 se muestra una representación del grafo completo para esta solución. 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

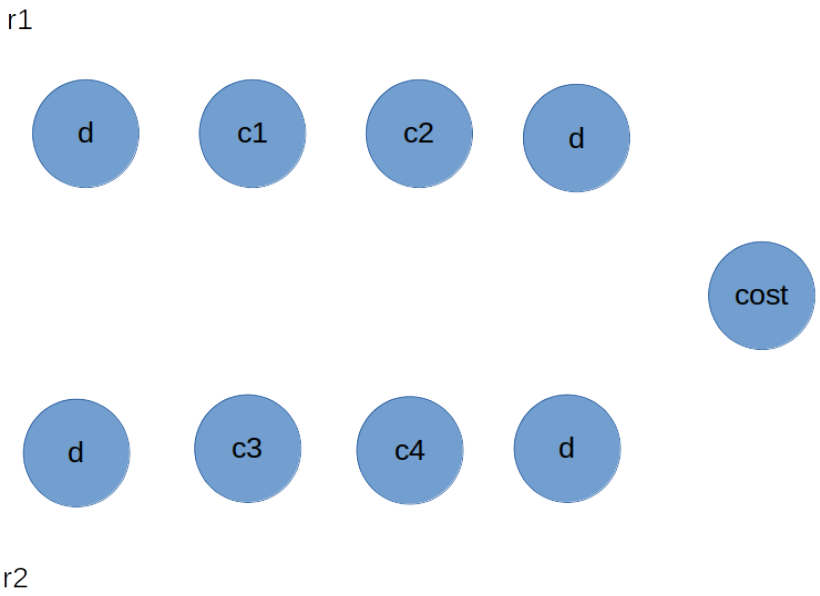

Figura 1.2: Nodos de alto nivel en un grafo de evaluación que representa la solución $s1$ de VRP clásico.

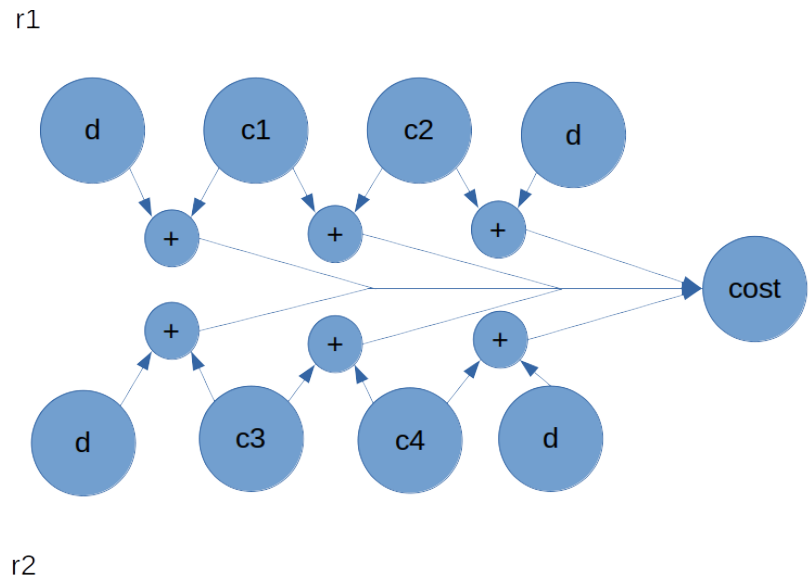

Figura 1.3: Grafo de evaluación que representa la solución $s1$ de VRP clásico.

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 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 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 montones de exploraciones distintas.

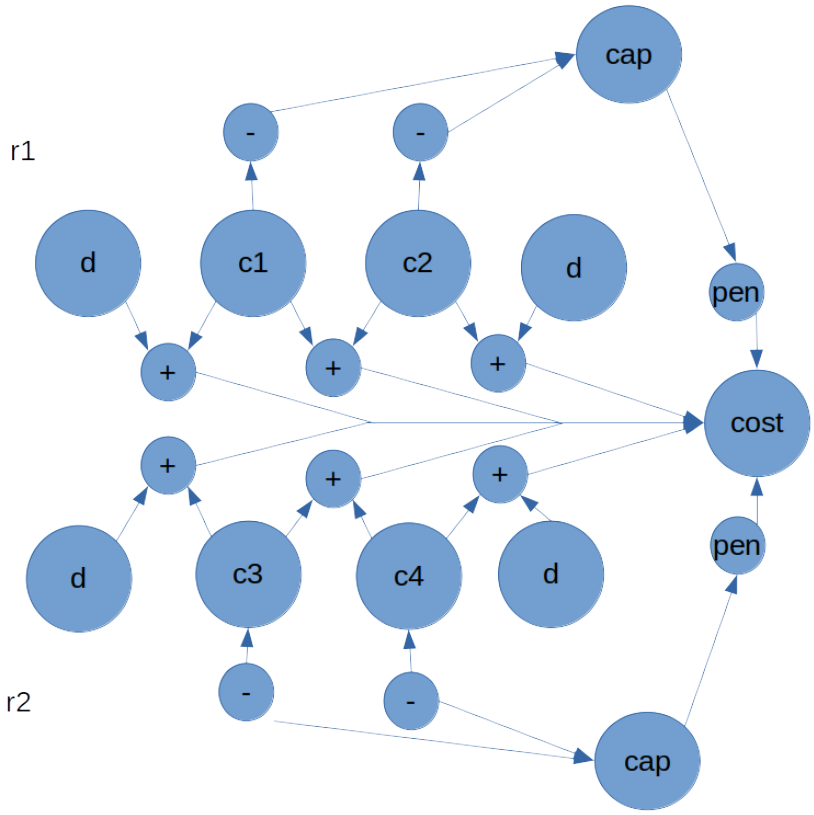

Figura 1.4: Grafo de evaluación que representa la solución $s1$ de VRP con restricciones de capacidad.

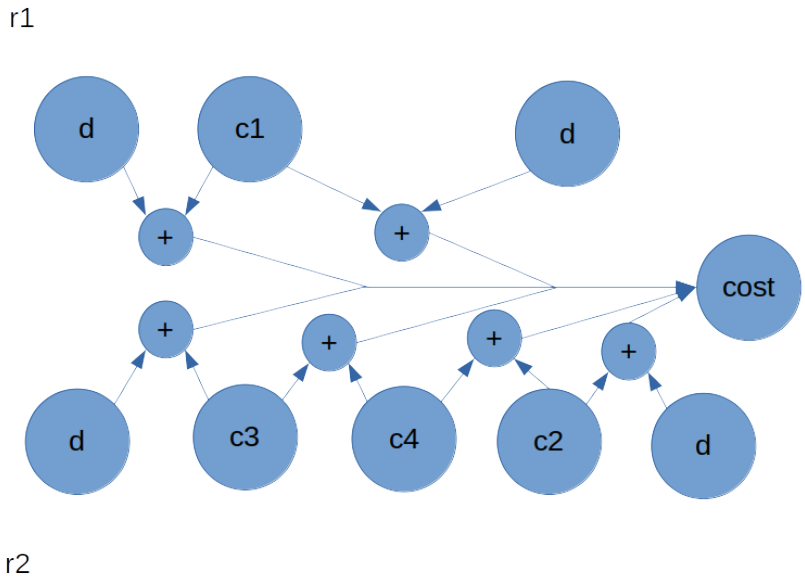

Figura 1.5: Grafo de evaluación que representa la solución $s1$ de VRP clásico luego de aplicada una instancia de *rarb*

1.5. 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 la cardinalidad potencialmente enorme de los vecinos o el retorno de mínimos locales. Para una vecindad muy poblada 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 numerosas 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. La función resultante recibe el problema 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 este trabajo, el árbol de vecindad y el grafo de evaluación forman parte también de las funciones de exploración creadas. Un árbol de vecinad genera las soluciones y un grafo de evaluación de usa para evaluarlas. El grafo debe ser pasado como entrada.

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 **??**.

El trabajo de generación de código y la creación de funciones a partir de estrategias diferentes se facilita mucho gracias a las características del lenguaje Common Lisp.

1.6. 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 útil 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 de menor vecino se une el código de métodos cuyo parámetro de search-strategy tenga tipo **best-improvement**, **return-best-solution**, **use-eval-graph** y cualquier otra clase que herede de alguna de estas.

Capítulo 2

Propuesta de solución

En este capítulo se describe el sistema implementado.

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 blo-

que 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.
- *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. 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).

Capítulo 3

Método de uso

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

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.