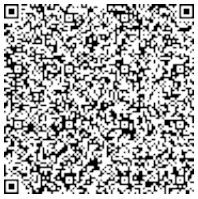
Universidad de La Habana
Facultad de Matemática y Computación

Primeras aproximaciones a la Búsqueda de Vecindad Infinitamente Variable

Autor: **Camila Pérez Mosquera**

Tutores: **MSc. Fernando Raúl Rodríguez Flores**

Trabajo de Diploma
presentado en opción al título de
Licenciado en Ciencia de la Computación



Mayo de 2017

Índice general

Opinión del tutor

El problema de enrutamiento de vehículos (VRP) es un problema NP-duro, por lo que una de las vías de solución más utilizada son las metaheurísticas, en particular las de búsqueda local, entre las que se destaca Búsqueda de Vecindad Variable (VNS). VNS es una metaheurística que se basa en el cambio de criterio de vecindad para escapar de los mínimos locales, por lo que siempre se usan un conjunto finito de estos para encontrar las soluciones. Este trabajo propone una modificación de la metaheurística VNS que permite considerar infinitos criterios de vecindad.

Para lograrlo, la diplomante tuvo que combinar creativamente herramientas y técnicas de varias ramas de la ciencia de computación, estudiar nuevos contenidos por sí misma, revisar bibliografía de temas inicialmente desconocidos para ella, y delimitar cuáles de los resultados que se reportan son relevantes para su trabajo y cuáles no.

La calidad y el volumen del trabajo realizado durante estos años se pueden apreciar en los resultados obtenidos, y en las presentaciones en eventos científicos entre los que se encuentran la Jornada científica de la facultad, pero también eventos internacionales como el Encuentro Cuba México de Métodos Numéricos y Optimización, y el IWOR 2017. Creo que el trabajo realizado y los resultados obtenidos evidencian que estamos en presencia de un excelente trabajo.

MSc. Fernando Raúl Rodríguez Flores
Facultad de Matemática y Computación
Universidad de la Habana

A mis padres, mis abuelos, mi novio, mi tutor, mis amigos
y a todos aquellos que de una forma u otra
me apoyaron para que poder llegar hasta aquí.

Resumen

En este trabajo se presenta una variante de la metaheurística Búsqueda de Vecindad Variable (VNS, por sus siglas en inglés) para el Problema de Enrutamiento de Vehículos con Restricciones de Capacidad (CVRP, por sus siglas en inglés), en la que es posible considerar `infinitos` criterios de vecindad, y no solamente un número finito de estos como es usual.

Esta nueva variante se conoce como Búsqueda de Vecindad Infinitamente Variable (IVNS, Infinitely Variable Neighborghood Search). Con esta idea se mejoran algunos resultados de la Búsqueda de Vecindad Variable.

Para poder considerar infinitos criterios se construye un lenguaje formal que describe los distintos criterios de vecindades que pueden usarse para solucionar un problema de enrutamiento de vehículos. Entre las ventajas de este lenguaje se encuentran: la posibilidad de generar automáticamente el código fuente de los criterios y la creación automática de nuevos criterios.

Abstract

In this thesis we propose a modification for the metaheuristic Variable Neighborhood Search, in which it is possible to consider an infinite number of neighborhood structures. To do this, we create a formal language whose strings are neighborhood criteria. The performance of this Infinitely Variable Neighborhood Search is compared with the classical finite Variable Neighborhood Search, for a Capacitated Vehicle Routing Problema and it shows better results.

As a consequence of representing the neighborhood criteria as strings in a formal language we have the posibility of generate the code of the criteria and the automatic creation of new criteria.

Introducción

En la sociedad existen muchos problemas que requieren utilizar un conjunto de vehículos para satisfacer, de manera óptima, las necesidades de varios clientes. Ejemplos de estas situaciones son la distribución logística de una empresa para entregar mercancías a un grupo de clientes, la recogida eficiente de un grupo de alumnos para llevarlos a su centro de estudio o el recorrido de un grupo de turistas por una ciudad de forma que se pueda visitar la mayor cantidad de lugares posibles. Este tipo de problemas se estudian bajo el nombre genérico de Problema de Enrutamiento de Vehículos.

El Problema de Enrutamiento de Vehículos (VRP, por sus siglas en inglés) es un problema de optimización combinatoria que consiste en planificar recorridos que permitan satisfacer las demandas de un conjunto de clientes geográficamente dispersos. Para ello se cuenta con una flota de vehículos que parten desde uno o varios depósitos centrales. El objetivo del VRP es diseñar rutas para cada vehículo que minimicen o maximicen algún objetivo.

El VRP y sus variantes han sido estudiados durante más de 50 años. La primera referencia al VRP la hace Dantzig [1], donde define el Problema de Enrutamiento de Vehículos con Restricción de Capacidad (CVRP), en el que cada vehículo tiene una capacidad que no debe exceder, y se desea minimizar el costo total de los recorridos.

El VRP es un problema NP-duro [2], por lo que los métodos exactos solo son factibles para resolver problemas de pequeña dimensión. Por ese motivo se han desarrollado diferentes heurísticas y metaheurísticas para solucionarlos, entre las que se encuentran las de búsqueda local [3, 4].

Los algoritmos de búsqueda local son métodos en los que se define una solución actual, una estructura de entorno, y de manera iterativa se generan

soluciones vecinas de la solución actual usando la estructura de entorno (o criterio de vecindad, como también se les conoce). Las soluciones vecinas pueden reemplazar o no a la solución actual en dependencia de su calidad para el problema de optimización y del algoritmo en cuestión. Por ejemplo, en Recocido Simulado se pueden aceptar soluciones peores en base a una probabilidad que depende de un parámetro del algoritmo [5], y en Búsqueda Tabú se prohíben movimientos que empeoren la solución actual [6].

La metaheurística Búsqueda de Vecindad Variable (VNS) se diferencia de otros algoritmos de búsqueda local por tener un conjunto finito de estructuras de entornos que modifican secuencialmente la solución actual, hasta que ninguna de ellas puede encontrar una solución mejor, y en ese caso se asume que la actual es un óptimo del problema. Las ideas fundamentales de este algoritmo son que un óptimo local para una estructura de entorno no tiene por qué serlo para otra y que un óptimo local para todas las estructuras de entorno que se están usando es un óptimo global, al menos, con respecto a ese conjunto de estructuras de entorno.

En los algoritmos de búsqueda local aplicados al CVRP se usan criterios de vecindad que en muchos casos se repiten. Algunos de estos criterios son:

- Mover aleatoriamente un cliente de una ruta a otra.
- Reubicar un cliente en su ruta.
- Intercambiar dos clientes aleatoriamente minimizando el costo del viaje

Un análisis de estos criterios permite identificar que en todos ellos están presentes los mismos elementos: *seleccionar una ruta aleatoria*, *seleccionar un cliente de una ruta*, o *seleccionar una ruta que cumpla determinadas características*. Estos elementos se pueden usar para crear un lenguaje formal que posibilite obtener criterios de vecindades, de esta forma se pueden obtener infinitos criterios que pueden ser usados en VNS.

El objetivo de este trabajo es modificar el algoritmo VNS para considerar **infinitos** criterios de vecindad en el Problema de Enrutamiento de Vehículos con Restricciones de Capacidad. Para poder considerar infinitos criterios de vecindad se construye el lenguaje formal formado por todos los posibles criterios de vecindad que se pueden tener para este problema, y cada vez que se "cambia de estructura de entorno" se genera una nueva cadena del lenguaje, usando una gramática libre del contexto.

Esta variante *infinita* del VNS, que será llamada Búsqueda de Vecindad Infinitamente Variable (IVNS), muestra mejores resultados que la versión *finita* en los experimentos realizados.

Este documento está estructurado de la siguiente forma: en el capítulo 1 se presentan los elementos básicos del trabajo: problemas de enrutamiento de vehículos, algoritmos de búsqueda local, Búsqueda de Vecindad Variable y lenguajes formales. En el capítulo 2 se hace una presentación formal de VNS y se presenta el algoritmo Búsqueda de Vecindad Infinitamente Variable. En el capítulo 3 se presenta un lenguaje formal que describe el conjunto de todos los posibles criterios de vecindad para un problema de enrutamiento de vehículos, y en el capítulo 5 se muestran los resultados obtenidos al aplicar el algoritmo; finalmente se presentan las conclusiones, recomendaciones, trabajos futuros y la bibliografía consultada.

Capítulo 1

Preliminares

En este capítulo se presentan los elementos principales de este trabajo que son: el problema de enrutamiento de vehículos con restricción de capacidad (CVRP) con algunas soluciones exactas y heurísticas utilizadas para su solución; las características fundamentales de los criterios de vecindad utilizados en algoritmos de búsqueda local, y la teoría de lenguajes formales. En la sección 1.1 se introduce el problema de Enrutamiento de Vehículos con restricción de Capacidad. En la sección 1.2 se presentan soluciones exactas y aproximadas para el problema de enrutamiento de vehículos, en particular se presenta la metaheurísitca Búsqueda de Vecindad Variable. En la sección 1.3 se presentan algunos de los criterios de vecindades usados para el VRP. Por último en la sección 1.4 se presentan los elementos de la teoría de lenguajes formales necesarios para describir, en el capítulo 3, todos los posibles criterios de vecindad mediante una gramática libre del contexto.

1.1. Problema de Enrutamiento de Vehículos con restricción de Capacidad

El Problema de Enrutamiento de Vehículos consiste en un conjunto de nodos que deben ser visitados (clientes) y un conjunto de entidades que deben visitar esos clientes (vehículos). Una ruta es el recorrido que realiza un vehículo para visitar a un grupo de clientes en un orden determinado y luego volver al depósito de donde salió. En el VRP se pueden optimizar distintos criterios como: el número de clientes atendidos, el consumo de

combustible o el costo del recorrido.

El VRP y sus variantes han sido estudiados durante más de 50 años. La primera referencia al VRP la hace Dantzig [1], donde define el problema de enrutamiento de vehículos con capacidad limitada (CVRP), en el que cada vehículo tiene una capacidad que no debe exceder, y se desea minimizar el costo total de los recorridos. En [1] se describe una aplicación real de la entrega de gasolina a las estaciones de servicio y se propone una formulación matemática. Este problema surge como una generalización del problema del Viajante (TSP) y es NP-Duro [2], por esto solo se han usado métodos exactos para resolver instancias pequeñas y para el resto de los problemas se han usado heurísticas y metaheurísticas. Algunos de estos métodos se presentan a continuación.

1.2. Estrategias de soluciones para el Problema de Enrutamiento de Vehículos

El Problema de Enrutamiento de Vehículos ha sido solucionado de varias formas. Para instancias pequeñas se han usado métodos exactos como ramificación y acotación (Branch and Bound B&B) [7], algoritmos sobre grafos [8] y programación dinámica [9]. Una explicación más detallada de todos los métodos exactos que se han usado para solucionar el CVRP se pueden encontrar en [2, 10, 11], pero como este es un problema NP-Duro [2] para resolver problemas grandes se suelen usar heurísticas y metaheurísticas.

Entre las heurísticas más conocidas están el algoritmo de Clarke y Wrigth [12], el algoritmo de dos-fases de Christofides-Mingozzi-Toth [2] y algoritmos de intercambio (sweep algorithm) [13, 14]. Estas heurísticas son constructivas y en algunos casos se intenta hacer una segunda fase para mejorar la solución construida [10].

En la literatura se reporta el uso de metaheurísticas como algoritmos genéticos [15], colonias de hormigas [16] y metaheurísticas de búsqueda local.

Las metaheurísticas basadas en búsqueda local son métodos en los que se define una solución actual, una estructura de entorno y de manera iterativa exploran las soluciones vecinas de la actual en busca de una nueva solución. Las nuevas soluciones se aceptan o rechazan en dependencia de su calidad para el problema y del algoritmo en cuestión. Por ejemplo, en

Recocido Simulado se aceptan soluciones peores en dependencia de determinada probabilidad que depende de un parámetro del algoritmo [5], y en Búsqueda Tabú se prohíben movimientos que empeoren la solución actual [17].

Algunas metaheurísticas de búsqueda local que se han usado para dar solución al Problema de Enrutamiento de Vehículos son Adaptive Large Neighborhood Search Heuristic (ANLS) [18], algoritmos híbridos de Búsqueda Tabú como [19, 20] y VNS, este último con muy buenos resultados [21, 22].

Los algoritmos de búsqueda local usan estructuras de entorno, o criterios de vecindad para mejorar la solución actual. En muchos casos se usan los mismos criterios de vecindad [5, 23, 24], la mayoría de estos se pueden expresar como combinación de un conjunto de elementos básicos, como se muestra en la siguiente sección.

1.3. Criterios de Vecindad

Los algoritmos de Búsqueda local usan estructuras de entorno o criterios de vecindad para explorar el espacio de soluciones. Estos criterios de vecindad dependen del problema que se quiera resolver y en el caso de los Problemas de Enrutamiento de Vehículos algunos de ellos pueden ser:

Intercambiar dos clientes seleccionados aleatoriamente [5, 6, 24], *Reubicar un cliente dentro de su ruta en la posición que minimice el costo de esa ruta* [23, 24], y *Mover aleatoriamente un cliente de una ruta a otra sin afectar la factibilidad débil de la misma* [5, 24].

En todos estos criterios se pueden encontrar los mismos elementos:

- Seleccionar una ruta.
- Seleccionar un cliente de una ruta.
- Eliminar un cliente de una ruta.
- Insertar un cliente en una ruta.

Las operaciones anteriores también se pueden realizar de acuerdo a algunas condiciones, que en este trabajo serán llamados filtros.

Algunos ejemplos de estos filtros son:

- Ruta con el mayor costo posible.
- Ruta con el menor costo posible.
- Cliente que al insertarlo aumente el costo lo más posible.
- Cliente que al sacarlo disminuya el costo lo más posible.

Algunos criterios en los que se apliquen los filtros son: *Reubicar un cliente dentro de su ruta en la posición que minimice el costo de esa ruta* [23, 24], y *Mover aleatoriamente un cliente de una ruta a otra sin afectar la factibilidad débil de la misma* [5, 24].

En la práctica, casi todos los criterios de vecindad son combinaciones de estos elementos comunes y los filtros. Por eso, uno de los objetivos de este trabajo es usar esos elementos para construir el lenguaje de los criterios de vecindad y de esta forma poder obtener infinitos criterios. En el capítulo 3 se propone un lenguaje formal, cuyas cadenas son combinaciones de estos elementos que representan criterios de vecindad en algoritmos de búsqueda local para el VRP. A continuación se presentan los elementos básicos de la teoría de lenguajes formales.

1.4. Teoría de Lenguajes Formales

Los elementos básicos de la teoría de lenguajes formales son: alfabeto, cadena y lenguaje. Un alfabeto es un conjunto de símbolos, una cadena es la concatenación de símbolos y un lenguaje es un conjunto de cadenas [25].

Una gramática libre del contexto es una tupla $G = (N, T, P, S)$ donde N es un conjunto de símbolos llamados no-terminales, T es un conjunto de símbolos llamados terminales que cumplen: $N \cap T = \emptyset$. $S \in N$ es un no terminal, llamado símbolo inicial y P es el conjunto de producciones de la gramática. En las gramáticas libres del contexto las producciones son de la forma: $A \to \alpha$, $A \in N$, y $\alpha \in (N \cup T)^*$.

A partir de una gramática se pueden generar cadenas partiendo del símbolo inicial, y aplicando aleatoriamente las producciones posibles. Para ello existen distintos algoritmos como seleccionar las producciones de manera uniforme o generar las cadenas uniformes en cuanto a su tamaño [26, 27].

Utilizando una gramática se puede definir un lenguaje formal que posibilite generar los criterios de vecindad para el VRP que se pueden usar en

una Búsqueda de Vecindad Variable. Las cadenas del lenguaje están formada por los elementos presentados en la sección 1.3. Por ejemplo una cadena generada por la gramática y que pertenezca al lenguaje puede ser *seleccionar una ruta y de ella seleccionar un cliente, insertar el cliente en la misma ruta en la posición que minimice el costo del recorrido*. De esta forma se pueden tener tantos criterios como cadenas genere la gramática, que pueden ser infinitas. En el capítulo 3 se presenta un lenguaje que posibilita obtener estos criterios.

Toda cadena que pertenezca a un lenguaje tiene asociado un árbol de sintaxis abstracta (AST). Esto es una estructura arbórea que representa el orden en que se deben aplicar las producciones para obtener la cadena dada.

En este trabajo se utilizan los AST de los criterios generados por la gramática para obtener el código de los mismos en un lenguaje de programación, de forma que se puedan utilizar en la metaheurística IVNS.

En este capítulo se presentaron los elementos necesarios para modificar el VNS para que acepte infinitos criterios de vecindad generados por una gramática. En el próximo capítulo se hace una explicación más detallada de la metaheurística VNS y se presenta su versión *infinita* IVNS.

Capítulo 2

Búsqueda de Vecindad Infinitamente Variable

En este capítulo, en la sección 2.1 se presentan los detalles de la metaheurística Búsqueda de Vecindad Variable y en la sección 2.2 se presentan distintas formas de explorar el espacio de las soluciones y en dependencia de esto se tienen distintas variantes de la VNS. Finalmente en la sección 2.3 se presenta la metaheurística Búsqueda de Vecindad Infinitamente Variable, que a diferencia de VNS considera infinitos criterios de vecindad.

2.1. Búsqueda de Vecindad Variable

Búsqueda de Vecindad Variable es una metaheurística que se basa en el cambio sistemático de entorno dentro de una búsqueda local [28]. Las primeras referencias que se tienen sobre esta metaheurística son: [29] de 1995 y [30] en 1997 donde usan VNS para dar solución al problema del viajante considerando la existencia o no de *backhauls*, (esto significa que los vehículos primero entregan las mercancías a los clientes y después recogen lo que los clientes deben devolver).

VNS está basada en dos principios básicos:

1. Un mínimo local con una estructura de entornos no lo es necesariamente con otra.

2. Un mínimo global es un mínimo local con todas las posibles estruc-

turas de entornos.

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

Para resolver un problema utilizando VNS es necesario definir qué es una solución, cuándo una solución es mejor que otra, qué criterios de vecindad se deben usar, y qué estrategia seguir para explorar la vecindad de la solución actual.

A continuación se formalizarán los elementos presentados anteriormente. Para ello, el conjunto de todas las posibles soluciones se denota como X, y el conjunto de soluciones vecinas de x, para un criterio de vecindad V, se denota como $V(x)$.

Definición 1 *Dada una función f que se desea minimizar, una solución y es un mínimo local de f en, si no existe una solución $x \in V(y)$ tal que $f(x) < f(y)$.*

Definición 2 *Dada una función f que se desea minimizar, una solución $x^* \in X$ es un mínimo global de f, si no existe una solución $x \in X$ tal que $f(x) < f(x^*)$.*

En VNS se usan estructura de entorno para cambiar de soluciones. La siguiente definición formaliza esta idea.

Definición 3 *Cada estructura de entorno (o criterio de vecindad) en el espacio de soluciones X es una aplicación $\eta : X \to 2^X$ que a cada solución $x \in X$ le asigna un conjunto de soluciones $\eta(x) \subset X$, que se dicen **vecinas** de x.*

Definición 4 *Para problemas donde las vecindades se exploran mediante estructuras de entornos, se dice que x' es un mínimo local con respecto a η_k, si no existe una solución $y \in \eta_k(x')$ tal que $f(y) < f(x')$.*

La Búsqueda de Vecindad Variable parte de una solución actual y un conjunto de estructuras de entorno. Utilizando uno de los criterios de vecindad se obtiene una solución vecina de la actual. Si la nueva solución es mejor, se cambia y se regresa a usar el primer criterio. Si no, se generan soluciones vecinas usando los otros criterios hasta que no sea posible mejorar la solución actual con ninguno de ellos.

Tomando en cuenta el principio: *un mínimo global es un mínimo local para todas las posibles estructuras de entornos*, se puede decir que esa solución es un mínimo global, al menos para el conjunto de entornos utilizados.

La idea de VNS se expresa en forma de pseudocódigo en el algoritmo 1 de la página 11.

```
1  η1, η2, ..., ηk_max : Conjunto finito de estructuras de entorno.
2  X : Solución Inicial
3  while No se cumplan las condiciones de parada do
4  |  k = 1;
5  |  while k ≠ k_max do
6  |  |  Agitación: Explorar el k-ésimo entorno de x y obtener x'
7  |  |  if x' es mejor que x then
8  |  |  |  x ← x'
9  |  |  |  k ← 1
10 |  |  end
11 |  |  else
12 |  |  |  k ← k + 1
13 |  |  end
14 |  end
15 end
```

Algoritmo 1: Algoritmo VNS

En el algoritmo, en la línea 6, se define **Agitación** como explora el k-ésimo entorno de x y obtener x'. En dependencia de cómo se explore ese entorno se tienen distintas variantes del algoritmo. Estas variantes se explican a continuación.

2.2. Estrategias para explorar el entorno de una solución

Como se explicó en la definición de VNS, existen diferentes estrategias para explorar el entorno de x y en dependencia de estas estrategias se han creado distintas variantes (o esquemas) del algoritmo VNS.

Exploración aleatoria : Se selecciona una solución aleatoria del entorno.

Exploración exhaustiva : Se exploran todas las soluciones del entorno y se devuelve la mejor.

Exploración primera mejora : Se comienza a explorar la vecindad exhaustivamente, hasta que se encuentra una solución mejor que la actual, y esa es la que se devuelve. Si no se encuentra esta solución, entonces esta exploración es equivalente a la exhaustiva.

Exploración primera mejora aleatoria : Se generan N soluciones aleatorias y se devuelve la mejor de ellas, a no ser que antes de generar N, aparezca una mejor que la solución actual.

Otras variantes de exploración de la vecindad se pueden consultar en [31].

Todas estas variantes tiene dos elementos fundamentales: la solución solo se actualiza si la nueva es mejor, y en cada corrida se usa una cantidad finita de estructuras de entorno.

En este trabajo se propone modificar la VNS para considerar infinitas estructuras de entorno usando la gramática que se presenta en el capítulo 3. A esta variante infinita de VNS se le llamará Búsqueda de Vecindad Infinitamente Variable y se presenta a continuación.

2.3. Búsqueda de Vecindad Infinitamente Variable

La idea de la Búsqueda de Vecindad Infinitamente Variable (IVNS) está basada en los principios presentados en la sección 2.1 y en otro que se deduce de ellos: *el óptimo para* ***todos*** *los infinitos posibles criterios de vecindad es el óptimo global*. De forma que si se tiene una solución que no puede ser mejorada con ninguno de los infinitos criterios de vecindad, se puede tener la certeza de que esa solución es un óptimo global.

La idea de IVNS es la misma que la VNS: cambiar sistemáticamente de estructura de entorno, pero considerando un conjunto de estructuras de entornos infinito.

De esta forma, cada vez que es necesario cambiar la estructura de entorno, en lugar de seleccionar otra de un conjunto finito y predefinido al comienzo del algoritmo, se genera un nuevo criterio de vecindad del conjunto (infinito) de todas las posibles estructuras de entorno para el problema que se desea resolver.

En la algoritmo 2 de la página 13 se presenta esta metaheurística en forma de pseudocódigo.

```
1  η(x) ← Generar nueva estructura de entorno.
2  x ← Generar Solución Inicial
3  while No se cumplan las condiciones de parada do
4  |  Agitación: Explorar el k-ésimo entorno de x y obtener x'
5  |  if x' es mejor que x then
6  |  |  x ← x'
7  |  end
8  |  else
9  |  |  η(x) ← Generar nueva estructura de entorno
10 |  end
11 end
```

Algoritmo 2: Algoritmo IVNS

En IVNS las estrategias para explorar un entorno son las mismas que en VNS: puede ser exploración aleatoria, exhaustiva, primera mejora o primera mejora aleatoria. Para los experimentos en este trabajo se usó primera mejora aleatoria.

Para aplicar el IVNS al CVRP, es necesario definir para este problema, cada uno de los elementos básicos planteados en la sección 2.1: qué es una solución, cuando una solución es mejor que otra, qué criterios de vecindad se deben usar, y cuál es la estrategia a seguir. Estos elementos se definen a continuación.

Solución : Para el CVRP, una solución es una partición del conjunto de clientes donde la suma de las demandas en cada clase no excede la capacidad de los vehículos.

Función objetivo : Se desea minimizar la distancia total recorrida por todos los vehículos.

Criterio de mejora : Una solución es mejor que otra si la distancia total recorrida por los vehículos en la primera es menor que en la segunda.

Estructuras de entorno : Son los criterios que se usan para perturbar una solución y obtener otra vecina de ella. Para el CVRP se pueden usar *Intercambiar dos clientes seleccionados aleatoriamente* [5, 24, 6], *Reubicar un cliente dentro de su ruta en la posición que minimice el costo de esa*

ruta [23, 24]. Estos son solo algunos ejemplos, pues en este trabajo se considerarán infinitos criterios.

Estrategia a seguir : Exploración primera mejora aleatoria.

Estos elementos son los mismos que el VNS, excepto por las estructuras de entorno que pueden ser infinitas.

En este capítulo se presentó la Búsqueda de Vecindad Variable, que usa una cantidad finita de criterios de vecindad y se propuso una modificación llamada IVNS que permite considerar infinitas estructuras de entorno. En el capítulo siguiente se presenta un lenguaje formal que permite representar infinitas estructuras de entorno para el CVRP.

Capítulo 3

Infinitos criterios de vecindad para el CVRP

Las referencias al VNS que aparecen en la literatura [22, 31] utilizan una cantidad finita de criterios de vecindad, y que normalmente suele se pequeña (3 o 4 criterios). En este trabajo se propone considerar infinitos criterios, los cuales se obtienen a través de una gramática que genera cadenas del lenguaje formado por los criterios de vecindad para el CVRP. Las cadenas del lenguaje se pueden convertir en código ejecutable para que, dada una solución, se pueda explorar su vecindad utilizando el criterio especificado por la cadena.

En este capítulo, en la sección 3.1 se describen los elementos comunes que están presentes en los criterios de vecindad para el CVRP y que posibilitan diseñar un lenguaje formal. En la sección 3.2 se define el lenguaje de los diferentes criterios y finalmente, en la sección 3.3, se presenta la gramática que genera los criterios de vecindad del CVRP.

3.1. Elementos comunes en criterios de vecindad para el CVRP

Una revisión de la bibliografía sobre criterios de vecindad para algoritmos de búsqueda local para el CVRP [2, 5, 11, 17, 32], permite apreciar que la mayoría de estos criterios están formados por los mismos elementos. Ejemplo de ello son los siguientes criterios, de los que se presenta una des-

cripción con palabras y un desglose de los mismos en términos de "elementos comunes".

Intercambiar dos clientes de rutas distintas:

1. Seleccionar una ruta.
2. Seleccionar un cliente de esa ruta.
3. Seleccionar una ruta diferente de la primera.
4. Seleccionar un cliente de esa nueva ruta.
5. Insertar el cliente de la primera ruta en la segunda.
6. Insertar el cliente de la segunda ruta en la primera.

Mover un cliente de una ruta a otra sin afectar la factibilidad:

1. Seleccionar una ruta.
2. Seleccionar un cliente de esa ruta.
3. Seleccionar una ruta diferente de la primera.
4. Insertar el cliente en esta nueva ruta tal que sea factible.

Reubicar un cliente en su ruta en la posición que minimice el costo:

1. Seleccionar una ruta.
2. Seleccionar un cliente de esa ruta.
3. Insertar el cliente en la ruta tal que se minimice el costo.

En los criterios definidos anteriormente se puede apreciar que existen elementos comunes a todos. Estos elementos son las acciones básicas a partir de las cuales se pueden obtener los distintos criterios. Algunas de estas acciones son:

- Seleccionar una ruta.
- Seleccionar un cliente de una ruta.
- Eliminar un cliente de una ruta.
- Insertar un cliente en una ruta.

Además, estas acciones se pueden realizar de acuerdo a alguna condición. A continuación se resaltan en **negrita** las condiciones en los criterios anteriores :

1. Seleccionar una ruta **diferente de la primera**.
2. Insertar el cliente en esta nueva ruta **tal que sea factible**.
3. Insertar el cliente en la ruta **tal que se minimice el costo**.

La condición 1 se puede considerar como una condición que debe cumplir una ruta para poder ser seleccionada, y las condiciones 2 y 3 son condiciones que se deben cumplir después de haber realizado las inserciones.

En este trabajo estas condiciones serán llamadas **filtros**. Los filtros se pueden definir por cada operación, por lo que pueden existir filtros para seleccionar rutas, para seleccionar clientes, y filtros para insertar. También se consideran filtros para optimizar la solución una vez se hayan intercambiado dos clientes o insertado un cliente en una ruta.

Un ejemplo de este último caso, puede ser que después de insertar un cliente en una ruta, se resuelva el problema del viajante para esa ruta con el fin de minimizar la distancia recorrida por el vehículo.

Algunos ejemplos de estos filtros son:

- Ruta con el mayor costo posible.
- Ruta con el menor costo posible.
- Cliente que al insertarlo aumente el costo lo más posible.
- Cliente que al sacarlo disminuya el costo lo más posible.

Tomando en cuenta que casi cualquier criterio se puede escribir utilizando los elementos comunes y los filtros identificados anteriormente, es posible construir el lenguaje formal, de los infinitos criterios de vecindad para el CVRP.

A continuación se presentan el lenguaje y la gramática definidos para generar los criterios de vecindad

3.2. Lenguaje Formal para criterios de vecindad

Para escribir el lenguaje de los criterios de vecindad del VRP, primero se le asignará a cada una de las acciones definidas anteriormente, una letra que las represente:

Seleccionar una ruta $\rightarrow$ r

Seleccionar un cliente $\rightarrow$ a

Insertar cliente en ruta $\rightarrow$ b

Intercambiar dos clientes $\rightarrow$ c

Como existen filtros para las diferentes acciones, los filtros se van a representar por f_k, donde k es la operación del filtro. Por ejemplo: los filtros para insertar se denotan por f_b, los de seleccionar ruta, por f_r, los de seleccionar un cliente por f_a, y filtros para la optimización de una ruta o solución, por f_o.

A cada filtro, se le agrega un subíndice i para representar el número del filtro que se está usando y así diferenciar dos filtros del mismo tipo. Por ejemplo para los filtros definidos anteriormente se pudiera tener:

Ruta con el mayor costo posible $\rightarrow f_{r_1}$.

Ruta con el menor costo posible $\rightarrow f_{r_2}$.

Cliente que al insertarlo aumente el costo lo más posible $\rightarrow f_{a_1}$.

Cliente que al sacarlo disminuya el costo lo más posible $\rightarrow f_{a_2}$.

A continuación se define un lenguaje formal cuyos símbolos son las notaciones definidas anteriormente, por lo que el alfabeto es:

$$\{a, b, c, r, f_{r1}, f_{r2} \ldots, f_{n_r}, f_{a1}, f_{a2} \ldots, f_{n_a}, f_{b1}, f_{b2} \ldots, f_{n_b}, f_{o1}, f_{o2} \ldots, f_{n_o}, [,]\},$$

donde:

- los corchetes se usan para declarar los filtros de cada posible operación.

- n_a, n_b, n_r y n_o son la cantidad de filtros para la selección de clientes, inserción de clientes y selección de rutas, y optimizar de la solución respectivamente.

Usando los elementos del lenguaje se pueden escribir los distintos criterios de vecindad. Por ejemplo:

- Intercambiar dos clientes $\rightarrow$ **rarac**.
- Mover un cliente a otra ruta $\rightarrow$ **rar[**$f_{r\neq}$**]b**. Donde $f_{r\neq}$ representa que la ruta sea distinta de una específica.
- Reubicar un cliente en su ruta en la posición que minimice el costo $\rightarrow$ **rab[**f_{b_l}**]**. Donde f_{b_l} representa que se inserte en la posición que minimice el costo.

Algunos filtros pueden recibir parámetros. Eso no se refleja directamente en la cadena, pero sí se tiene en cuenta en el momento de la generación de código, como se describe en el capítulo 4.

Tomando en cuenta que las cadenas que pertenecen al lenguaje son criterios de vecindad, tienen que cumplir una serie de propiedades que se señalan en 3.2.1. Pero antes se presentan un conjunto de notaciones necesarias para describir mejor estas propiedades.

Clientes Disponibles: Se considera que los clientes disponibles son aquellos que han sido seleccionados de las distintas rutas y que aún no han sido insertados en ninguna ruta, ni intercambiados.

Consumir un cliente: Es eliminar un cliente del conjunto de **clientes disponibles**. Esto sucede cuando se selecciona un cliente para insertar en una ruta, o cuando se seleccionan dos clientes para intercambiarlos.

Por tanto, se puede decir que la operación insertar **consume** un cliente, y la operación intercambiar **consume** dos clientes. Dado que las acciones están definidas por letras, se puede decir que una **b consume** una **a** y que una **c consume** dos **a**.

Paso *i*-ésimo o posición *i*-ésima de la cadena : Como un criterio de vecindad es un conjunto de pasos o acciones, cuando se habla de la posición *i*-ésima de la cadena, se está haciendo referencia a la acción o paso *i*-ésimo del criterio.

Cantidad de ocurrencias de la letra l: Aquí se definen tres notaciones:

$|l|$: Para una cadena cualquiera, $|l|$ representa la cantidad de ocurrencias del terminal l en la cadena.

Por ejemplo para la cadena ababracb, se tiene: $|a| = 3$, $|b| = 3$, $|c| = 1$, $|r| = 1$.

$|l|_{<i}$: Para una cadena cualquiera, $|l|_{<i}$ representa la cantidad de ocurrencias del terminal l antes de la posición i.

Por ejemplo para la cadena aba, se tiene:

	$l = a$	$l = b$
$\|l\|_{<1}$	0	0
$\|l\|_{<2}$	1	0
$\|l\|_{<3}$	1	1
$\|l\|_{<4}$	2	1

Para una cadena de longitud n, se tiene que $|l|_{<(n+1)}$ representa la cantidad de ocurrencias de la letra l en toda la cadena, esto es lo mismo que calcular $|l|$.

$|l|_i$: Para una cadena cualquiera, $|l|_i$ representa la cantidad de ocurrencias del terminal l hasta la posición i.

Con estas notaciones se pueden expresar las propiedades que debe cumplir el lenguaje. Estas propiedades se presentan a continuación.

3.2.1. Propiedades del lenguaje

Tomando en cuenta que las cadenas representan criterios de vecindad se pueden deducir algunas propiedades que deben cumplir las cadenas del lenguaje:

1. Las cadenas tienen que empezar con **r**, porque la primera operación siempre será seleccionar una ruta.

2. Si es la posición i de la cadena hay una **b**, (esto significa que la acción que corresponde hacer es insertar un cliente en una ruta), tiene que quedar algún cliente disponible al llegar a esta posición. Esto se traduce en que si hay una **b** en la posición i, se tiene que cumplir que $|\mathbf{a}|_{<i} - |\mathbf{b}|_{<i} - 2|\mathbf{c}|_{<i} \geq 1$.

La idea es que en la posición i de la cadena, después de haber realizado $i-1$ pasos del criterio de vecindad, tiene que existir al menos un cliente disponible. Como cada **b** consume una **a**, y cada **c** consume 2 **a**, la cantidad de **a** antes de la posición i tiene que ser mayor que la cantidad de **b** anteriores a la posición i sumada con 2 veces la cantidad de **c** que aparecen antes de la posición i.

Siguiendo un razonamiento similar, para el caso en que en la posición i hay una **c** se tiene la siguiente propiedad.

3. Si en la posición i de la cadena hay una **c**, esto significa que la acción i-ésima es intercambiar dos clientes, tienen que existir al menos dos clientes disponibles. Haciendo un razonamiento similar al caso anterior se tiene que si aparece una **c** en la posición i, se tiene que cumplir que $|\mathbf{a}|_{<i} - |\mathbf{b}|_{<i} - 2|\mathbf{c}|_{<i} \geq 2$.

4. No se deben seleccionar rutas si no se van a usar. Las rutas se usan cuando se extrae un cliente o se inserta un cliente. Esto se traduce en que la cantidad de **r** en la cadena tiene que ser menor o igual que la cantidad de selecciones e inserciones de clientes. En otras palabras $|\mathbf{r}| \leq |\mathbf{a}| + |\mathbf{b}|$. En la cantidad de **r** no influye la cantidad de **c**, porque en la operación intercambiar dos clientes no interviene ninguna ruta.

5. Por otra parte no se deben seleccionar clientes si no se van a usar. Con un análisis similar al anterior, se tiene que la cantidad de **a** en la cadena tiene que ser igual que la cantidad clientes que se consumen cuando se hace una operación de inserción y los que se consumen cuando se hace un intercambio. En otras palabras, $|\mathbf{a}| = 2|\mathbf{c}| + |\mathbf{b}|$.

6. Por último y no menos importante, las cadenas no pueden ser vacías.

Una vez definidas las propiedades que deben cumplir las cadenas que pertenecen al lenguaje, se presenta una gramática libre del contexto que permite generar las infinitas cadenas.

3.3. Gramática para el lenguaje

Para definir la gramática primero hay que declarar los elementos noterminales:

S : Es el no-terminal principal, el *elemento distinguido* de la gramática, con él se asegura que la cadena siempre empiece por una **r** y en dependencia de la producción que se aplique pueden tener operaciones de intercambio o inserción.

S_1 : Posibilita seleccionar más clientes de una ruta, y pueden seleccionarse más rutas también, en dependencia de la producción que se escoja en R_1. Garantiza que si selecciona un cliente se agregue una inserción y si se seleccionan dos clientes se agregue un intercambio.

R_1 : Seleccione una ruta o no.

A, B, C, R : Permiten que se agreguen las acciones de seleccionar un cliente, insertar un cliente en una ruta, intercambiar dos clientes y seleccionar una ruta respectivamente con y sin filtros.

F_a, F_b, F_o, F_r : Permite seleccionar los filtros de las acciones anteriores.

Una vez definidos los elementos no-terminales se presenta la gramática

1. $S \rightarrow R\ A\ S_1\ B$
2. $S \rightarrow R\ A\ S_1\ R\ B$
3. $S \rightarrow R\ A\ A\ S_1\ C$
4. $S \rightarrow R\ A\ R\ A\ S_1\ C$
5. $S_1 \rightarrow R_1\ A\ S_1\ R_1\ B$
6. $S_1 \rightarrow R_1\ A\ R_1\ A\ S_1\ C$
7. $S_1 \rightarrow \varepsilon$
8. $R_1 \rightarrow R \mid \varepsilon$
9. $A \rightarrow a \mid a[F_a]$
10. $B \rightarrow b \mid b[F_b] \mid b[F_o]$
11. $R \rightarrow r \mid r[F_r]$
12. $C \rightarrow c \mid c[F_o]$
13. $F_a \rightarrow f_{ai}\ , F_a \mid f_{ai}, \forall\ i = 1, \ldots, n_a$

14. $F_b \rightarrow f_{bi}\ , F_b \mid f_{bi}, \forall\ i = 1, \ldots, n_b$
15. $F_r \rightarrow f_{ri}\ , F_r \mid f_{ri}, \forall\ i = 1, \ldots, n_r$
16. $F_o \rightarrow f_{oi}\ , F_o \mid f_{oi}, \forall\ i = 1, \ldots, n_o$

Cada una de las posibles acciones y filtros que se deseen incluir en los criterios de vecindad, se deben agregar como un elemento terminal. De esta forma, se pueden agregar tantos filtros como se deseen y así extender los criterios de vecindad que se generan a otros tipos de problema de la familia VRP.

A continuación se muestra que las cadenas generadas por la gramática cumplen las propiedades definidas en 3.2.1.

3.3.1. Propiedades de las cadenas generadas por la gramática

En esta sección se brindan elementos que permiten argumetar por qué las cadenas generadas por la gramática cumplen las condiciones indicadas en la sección 3.2.1.

Las propiedades 1 y 6 se cumplen, pues las producciones 1, 2, 3 y 4 obligan a que la gramática empiece con **r** y que no se genere la cadena vacía.

Con las producciones 1 y 2 se generan operaciones de seleccionar un cliente para insertar en una ruta, con todas las posibles combinaciones de selecciones de ruta: solo una ruta para las dos operaciones, o una ruta para cada operación (una para seleccionar, y otra, que puede ser la misma o no, para insertar).

Las producciones 3 y 4 son similares a las anteriores, pero ahora para el intercambio. Las opciones de rutas son: seleccionar los dos elementos de la misma ruta (3), o seleccionar cada elemento después de una selección de ruta (4).

Las producciones 5 y 6 son similares a las del terminal S, con la diferencia, de que como S_1 siempre aparece después de alguna de las producciones de S, ya existe al menos una **r** en la cadena, por lo que las opciones de extraer/insertar/intercambiar se pueden realizar con o sin selección de rutas previas. Por eso es que, con las producciones señaladas en 8, el no-terminal R_1 puede derivar en el noterminal R (que sí inserta una selección de ruta con o sin filtros como se indica en la producción 11), o en la cadena vacía.

Y para terminar algún momento la recursividad de agregar más y más S_1, se tiene la producción 7 con la que S_1 deja de agregar operaciones.

Las propiedades 2, 3, 4, 5 se cumplen por las producciones 5, 6 , 7, 8 respectivamente pues estas posibilitan que no se seleccionen clientes ni rutas de más y garantizan junto con las producciones de S que siempre existan los clientes necesarios para realizar las inserciones y los intercambios.

Las producciones señaladas en 9, 11, 10, 12 agregan las operaciones: *seleccionar cliente de ruta* , *seleccionar ruta*, *insertar cliente en ruta*, *intercambiar dos clientes* con o sin filtros.

Finalmente en las producciones 13, 14, 15 y 16, se agregan los filtros que se hayan definidos como terminales.

Una vez definida la gramática y demostrado que las cadenas generadas cumplen las propiedades del lenguaje, se presentan algunas de las cadenas que genera el lenguaje.

3.3.2. Ejemplos de criterios generados

Antes de presentar algunos criterios generados por la gramática, se presentan los filtros que se están usando.

Para seleccionar más de un cliente se seleccionan de la última ruta que se definió.

Filtros de selección de ruta :

- con-al-menos-1-elemento: La ruta que se selecciona deben tener al menos un cliente.
- mayor-costo: La ruta que se selecciona es la que tenga mayor costo.
- menor-costo: La ruta que se selecciona es la que tenga menor costo.
- menos-factible: La ruta que se selecciona es la que tenga menor factibilidad; su carga está muy cerca de la capacidad real del vehículo, o el vehículo está lleno.
- mas-factible: La ruta que se selecciona es la que tenga mayor factibilidad; es la que su vehículo está menos cargado, por tanto pudiera añadir más clientes a su recorrido.

Filtros para seleccionar cliente :

- minimice-costo-extraccion: Cliente que al quitarlo de una ruta el costo de esta disminuye, disminuyendo así el costo total de los recorridos.

Filtros para optimizar la solución :

- 2-opt: Después de intercambiar los clientes, se realiza el algoritmo 2-opt para optimizar las rutas. Este algoritmo se puede consultar en [28].

Filtros para la inserción :

- factible: Cuando el cliente se inserte en la ruta, esta tiene que seguir siendo factible, o sea no se debe exceder la capacidad máxima de la ruta.
- minimice-costo-insercion: Cuando se inserta el cliente, se debe insertar en la posición que menos aumente el costo de la ruta.
- rutas-distintas: Solo se realiza la inserción si la ruta en que debe ser insertado es diferente de la ruta de la que fue seleccionado.
- en-la-posicion-de *cliente*: Este filtro recibe como parámetro un cliente y significa que cuando se inserte en la ruta se pone en la posición del *cliente* que se pasa como argumento.

Una vez presentados los filtros, se presentan algunas cadenas generadas.

1. **rarac**: Seleccionar una ruta, y de esa extraer un cliente. Seleccionar otra ruta (que puede ser la misma o no) y de ella seleccionar otro cliente. Una vez que se tengan los dos clientes, se intercambian.
2. **rar[**f_{c_1}**]b**: Seleccionar una ruta, y de ella un cliente. Seleccionar una ruta para insertar clientes con al menos un elemento, y en ella insertar el cliente seleccionado.
3. **rab[**f_{b_1}**]**: Seleccionar una ruta, y de ella un cliente. Insertar el cliente en la ruta en la posición que minimice el costo de la ruta.
4. **r[**f_{r_2}**]a[**f_{c_1}**]r[**f_{r_3}**]ac**: Seleccionar la ruta de mayor costo, y de esa extraer un cliente que minimice el costo de la extracción. Seleccionar otra ruta que sea la de menor costo y de ella seleccionar un cliente. Una vez que se tengan los dos clientes, se insertan en la ruta del otro.

5. **r[**f_{r_5}**]arb**: Seleccionar la ruta de menor factibilidad (la que su capacidad actual está más cerca de su capacidad real), y de esa extraer un cliente. Seleccionar otra ruta para insertar, e insertar el cliente en ella.

En este capítulo se presentó cómo generar infinitos criterios dado una gramática. Estas cadenas serán usadas para tener infinitos criterios en el algoritmo 2.

Los criterios generados son solo cadenas del lenguaje que representan las acciones que se deben realizar para modificar la solución. Para poder usar estos criterios en un algoritmo, es necesario poder ejecutar el código que ellos indican. Esto se puede hacer generando el código de los criterios en distintos lenguajes, como se indica en el siguiente capítulo.

Capítulo 4

Generación de Código de los criterios

En este capítulo se describe cómo obtener el código de los criterios de vecindad representados por cadenas de la gramática mostrada en el capítulo anterior. Para ello, en la sección 4.1 se explica cómo se le aplica un procesamiento a las gramáticas para poder convertirlas en código ejecutable y en la sección 4.2 se presentan algunas cadenas convertidas a código ejecutable y cómo se realiza la generación.

4.1. Procesamiento de las cadenas de la gramática

Las cadenas que se pueden generar con la gramática en realidad representan familias de criterios. Por ejemplo, si la solución inicial está formada por dos rutas, el criterio **raraabc** puede representar varios criterios, aunque en todos ellos los primeros cinco pasos son siempre los siguientes:

1. Seleccionar una ruta r_1
2. seleccionar un cliente de la ruta r_1 y guárdalo en c_1
3. Seleccionar una ruta r_2
4. seleccionar un cliente de la ruta r_2 y guárdalo en c_2
5. seleccionar cliente de la ruta r_2 y guárdalo en c_3

Esto ocurre porque en la selección de una ruta y la de un cliente, se agregan variables r_i y c_j, donde los índices i y j dependen de la cantidad de rutas y clientes que se hayan seleccionado hasta ese momento. En el caso de la cadena **raraabc**, cuando se procesa para agregar estas variables, solo hay una forma de asignar estas variables r_1, r_2, c_1, c_2, y c_3. Sin embargo, las últimas dos operaciones que son la inserción de uno de los clientes en una de las rutas (**b**) e intercambiar los otros dos clientes (**c**) se pueden realizar de 6 formas diferentes, en dependencia de qué clientes y qué rutas se seleccionan para cada operación. A continuación se muestras estas posibilidades:

1. **b**: Insertar el cliente c_1 en la ruta r_1.
   c: Intercambiar los clientes c_2 y c_3.
2. **b**: Insertar el cliente c_1 en la ruta r_2.
   c: Intercambiar los clientes c_2 y c_3.
3. **b**: Insertar el cliente c_2 en la ruta r_1.
   c: Intercambiar los clientes c_1 y c_3.
4. **b**: Insertar el cliente c_2 en la ruta r_2.
   c: Intercambiar los clientes c_1 y c_3.
5. **b**: Insertar el cliente c_3 en la ruta r_1.
   c: Intercambiar los clientes c_1 y c_2.
6. **b**: Insertar el cliente c_3 en la ruta r_2.
   c: Intercambiar los clientes c_1 y c_2.

En estos casos no todos los criterios son iguales. Por ejemplo, en 2 se inserta c_1 en la ruta r_2, por tanto se mueve c_1 hacia otra ruta pero se intercambian c_2 y c_3 que son de la misma ruta. El caso 1 es parecido, pero c_1 se queda en su ruta lo que puede ser que se inserte en otra posición.

En 3, 4, 5 y 6, se intercambian clientes de rutas que pueden ser distintas. En 3 y 5 se insertan los clientes en una ruta distinta de la que se seleccionaron, mientras que en 4 y 6 se inserta en su misma ruta.

Para que cada cadena se convierta en un criterio específico (determinado por los nombres de las variables r_i y c_j, que intervienen en el mismo) y

no en una familia de criterios como sucede hasta ahora, las cadenas generadas se procesan antes de hacer la generación de código.

Este procesamiento es el siguiente: Cada vez que viene una **r** se agrega r_i con el i correspondiente a la lista de rutas, i empieza en 1 y se incrementa de uno en uno hasta llegar a la cantidad de rutas total que hay en la cadena. Cada vez que aparece una **a** se agrega un r_j con el j correspondientes a la lista de clientes, j empieza en 1 y se incrementa de uno en uno hasta la cantidad total de clientes en la cadena. Si aparecen mas de una **a** consecutivas entonces se agrega a la lista clientes el último r_i por cada **a** que aparezca sin contar la primera. Si hay una **b** se seleccionan de manera aleatoria un cliente y una ruta, se eliminan de las listas correspondientes. Si aparece una **c** entonces se seleccionan de manera aleatoria dos clientes de la lista de clientes y se eliminan. Siguiendo estos pasos se asegura que todas las rutas que han sudo creadas se usen al menos una vez, ya que se pueden seleccionar para extraer y/o insertar cliente. También se garantiza que todos los clientes se usen.

Este procesamiento se describe en el algoritmo 3 de la página 30.

A continuación se muestra, para varias cadenas del lenguaje, uno de los criterios específicos que resulta de aplicar este algoritmo.

1. **rarac:** Seleccionar una ruta r_1. Seleccionar un cliente de r_1 y guardarlo en c_1. Seleccionar una ruta r_2. Seleccionar un cliente de r_2 y guardarlo en c_2. Intercambiar los clientes c_1 y c_2.

2. **raraabc:** Seleccionar una ruta r_1. Seleccionar un cliente y guardarlo en c_1. Seleccionar una ruta r_2. Seleccionar un cliente de r_2 y guardarlo en c_2. Seleccionar un cliente de r_2 y guardarlo en c_3. Insertar el cliente c_3 en la ruta r_1. Intercambiar los clientes c_1 y c_2.

3. **rarb[f_{b_1}]:** Seleccionar una ruta r_1. Seleccionar un cliente de r_1 y guardarlo en c_1. Seleccionar una ruta r_2. Insertar c_1 en la ruta r_2 en la posición que minimice el costo de la ruta.

La selección de las rutas de la cual se van a extraer y/o insertar los clientes se hace de manera aleatoria, al igual que la selección de clientes que van a ser intercambiados.

Si aparecen los filtros en la cadena, estos se procesan de forma similar, algunos filtros pueden recibir parámetros. Por ejemplo el filtro de inserción

```
1  clients = ()
2  route = ()
3  J = 1
4  foreach símbolo s en la cadena do
5      if s == r then
6          r_J ← crear-route
7          J ← J + 1
8          route ← route + {r_J}
9      end
10     if s == a then
11         c_I ← crear-client
12         I ← I + 1
13         clients ← clients + {c_I}
14     end
15     if s == b then
16         r ← random-element-routes
17         c ← random-element-clients
18     end
19     if s == c then
20         c_1 ← random-element-routes
21         c_2 ← random-element-clients
22     end
23 end
```

Algoritmo 3: Algoritmo para procesar las cadenas.

de cliente *en-la-posicion-de cliente* recibe como parámetro el cliente en cuya posición se desea insertar el nuevo.

Si se tiene la cadena **rarar[**$f_{\neq}$**]bb** , la tercera ruta seleccionada puede ser distinta de la primera o distinta de la segunda en dependencia de cual sea seleccionada aleatoriamente entre las posibles rutas.

Una vez procesado las cadenas se presenta la generación de código. En la siguiente sección se explica cómo se convierte las cadenas específicas de cada criterio de vecindad en código ejecutable.

4.2. Generación de código ejecutable

Para hacer la generación del código ejecutable se crea una clase por cada operación básica. Por ejemplo

1. Seleccionar una ruta: Se convierte en una clase que tiene como parámetros los filtros que se usarán (puede ser que no se use ninguno) y el nombre de la variable donde guarda la ruta seleccionada (por ejemplo r_1).
2. Seleccionar un cliente: Tiene como parámetros la ruta de la que se va a seleccionar el cliente, los filtros y el nombre de la variable.
3. Insertar un cliente en la ruta: Tiene el cliente y la ruta (el nombre la variable del cliente, por ejemplo c_1, y el nombre de la variable de la ruta, por ejemplo r_1).
4. Intercambiar dos clientes: Tiene los nombres de las variables de los clientes.

Tanto 2, como 3 y 4 guardan en una variable la solución que resulta después de aplicar la operación correspondiente, pues con cada una se modifica la solución original.

Las distintas clases que se crean forman parte de los nodos del árbol de sintaxis abstracta, que en cierta forma indican el orden en que se realizan las acciones. Como primero se realiza las selecciones de rutas y clientes, cuando corresponde generar el código de una operación de inserción o intercambio, ya las distintas rutas y clientes que intervienen en ella tienen un valor asignado.

Aplicando la generación de código a un criterio específico quedaría:

1. Generar una cadena del lenguaje → **raraabc**.
2. Procesar la cadena generada. Queda como resultado:
   - r → seleccionar-ruta r_1.
   - a → seleccionar-cliente c_1 de r_1.
   - r → seleccionar-ruta r_2.
   - a → seleccionar-cliente c_2 de r_2.
   - a → seleccionar-cliente c_3 de r_2.
   - b → insertar c_2 en r_1.
   - c → intercambiar c_1 c_3.
3. Generar el AST. Luego de aplicar las producciones para obtener la cadena, se crean las instancias de las clases que representan los nodos del AST correspondientes a la cadena generada. Si las funciones que generan las instancias de las clases tienen el mismo nombre de la clase que generan y reciben como argumento los parámetros del nodo, para el ejemplo anterior se tiene que el AST se puede generar con las siguientes instrucciones:
   - (seleccionar-ruta r_1)
   - (seleccionar-cliente r_1 c_1)
   - (seleccionar-ruta r_2)
   - (seleccionar-cliente r_2 c_2)
   - (seleccionar-cliente r_2 c_3)
   - (insertar c_2 r_1)
   - (intercambiar c_1 c_3)

   El AST resultante para la cadena **raraabc** se puede ver en la figura 4.1 de la página 33.
4. Una vez creadas las distintas clases del AST, y después de haber definido cómo se genera cada nodo en cada lenguaje en el que se desee el código, se puede recorrer el AST generando el código del criterio.

   A continuación se muestra cómo quedaría el código de este criterio generado en una descripción cercana al lenguaje natural, y en Lisp, que fue el lenguaje seleccionado para el desarrollo de este trabajo:

Lenguaje Natural :

Seleccionar una ruta aleatoria (route1)
Seleccionar un cliente aleatorio de la ruta (route1), y guardarlo en (client1)
Seleccionar una ruta aleatoria (route2)
Seleccionar cliente aleatorio de la ruta (route2), y guardarlo en (client2)
Seleccionar cliente aleatorio de la ruta (route2), y guardarlo en (client3)
Intercambia los clientes (client1) (client3)
Insertar el cliente (client2) en la ruta (route1)

Lisp :

```
(defun Neighbooorhood (x)
        (let*((route1 nil)
              (client1 nil)
              (route2 nil)
              (client2 nil)
              (client3 nil)
              (solution (clone x)))
    (setf (values route1 solution) (seleccionar-ruta solution))
    (setf (values client1 solution) (seleccionar-cliente route1 solution))
    (setf (values route2 solution) (seleccionar-ruta solution))
    (setf (values client2 solution) (seleccionar-cliente route2 solution))
    (setf (values client3 solution) (seleccionar-cliente route2 solution))
    (setf solution (insertar-cliente-en-ruta client2 route1))
    (setf solution (intercambiar-dos-clientes client1 client3))
solution))
```

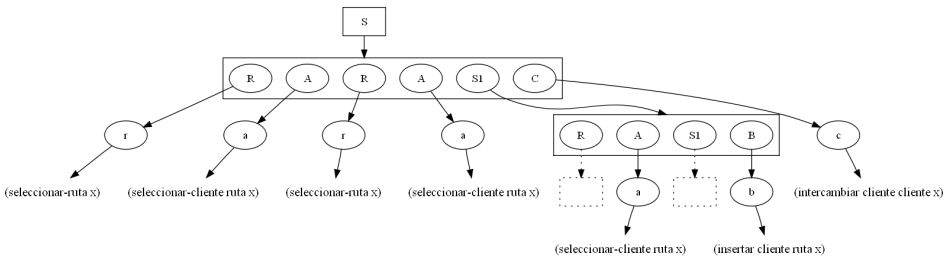


Figura 4.1: AST para la cadena **raraabc**.

En este capítulo se presentó cómo convertir las cadenas generadas por la gramática en código ejecutable, de forma que dada una solución, se pueda modificar para obtener otra en su vecindad. De esta forma se pueden usar infinitos criterios en el algoritmo IVNS.

En el siguiente capítulo se presentan algunos de los resultados obtenidos con una implementación de IVNS.

Capítulo 5

Resultados

En este capítulo se presentan algunos de los resultados obtenidos con el algoritmo IVNS para el CVRP, se comparan con el desempeño de un VNS clásico en la sección 5.1, y se ilustran algunos problemas identificados en las corridas realizadas en 5.2.

5.1. Comparación entre VNS e IVNS

Para probar estos algoritmos se han usado algunos de los ejemplos que se pueden encontrar en [33]. En particular se muestran resultados para el problema A-n33-k6. Los experimentos se corrieron en una computadora Intel i3 con 4 GB de RAM. El criterio de parada utilizado fue un número máximo de soluciones evaluadas.

Tanto en la implementación de IVNS como en la de VNS se usa primera mejora aleatoria como estrategia para explorar el entorno.

Las primeras pruebas que se realizaron tuvieron como objetivo comparar el comportamiento de IVNS con el de un VNS clásico, y esto se muestra en las figuras 5.1, 5.2 y 5.3.

En la figura 5.1 se puede ver el desempeño del IVNS (en azul) y el VNS (en rojo). En el eje x están la cantidad de iteraciones realizadas , y en el eje y, el valor mínimo obtenido después de 30 corridas con cada número de iteraciones.

En la gráfica se define perfectamente la diferencia entre ambos: el IVNS (azul) obtiene mejores resultados, llegando a ser los más altos cercanos a

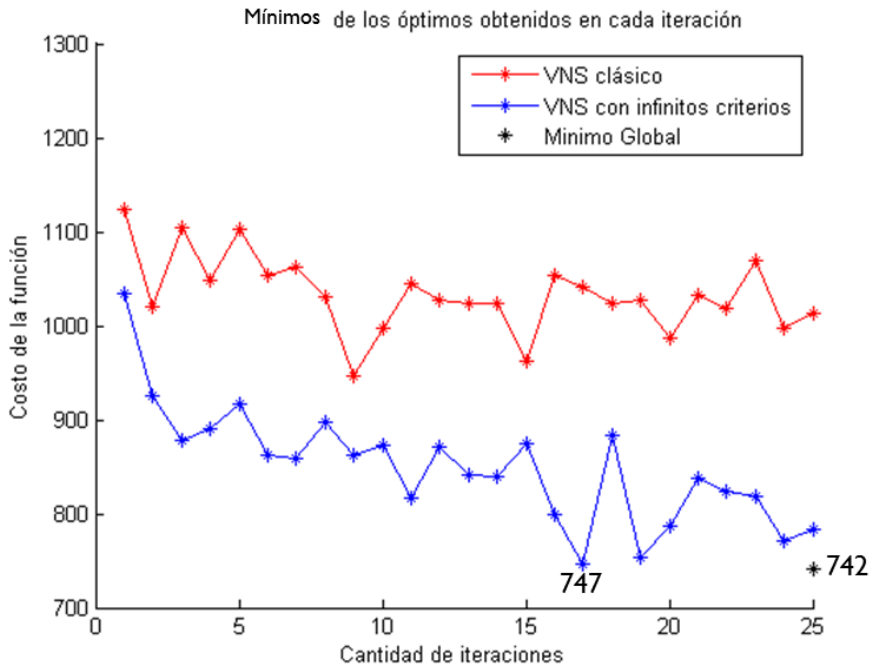


Figura 5.1: Óptimos considerando infinitos criterios para el problema A-n33-k6.

900, y el VNS (rojo) obtiene valores que no disminuyen de 1000. En color negro hay un punto que es 742, que fue el valor que se obtuvo mediante un algoritmo exacto en [33]. También se puede apreciar que con 17000 iteraciones, el IVNS encontró una solución con costo 747, bastante cercano al óptimo real, y que mejora otros resultados de la literatura como el reportado en [21], que es de 785.

En la figura 5.2 de la página 36 y en la figura 5.3 de la página 37 se presentan otras comparaciones de VNS e IVNS con respecto a los problemas A-n65-k9 y A-n80-k10.

A partir de estas imágenes parece ser que mientras mayor sea la cantidad de clientes en el problema mayor es la diferencia entre los resultados obtenidos por el IVNS y los de VNS. De todas formas, para poder justificar esta afirmación sería necesario realizar más experimentos.

Aunque los resultados de IVNS son mejores que los de VNS, no se alcanza el óptimo global del problema, y los resultados obtenidos están le-

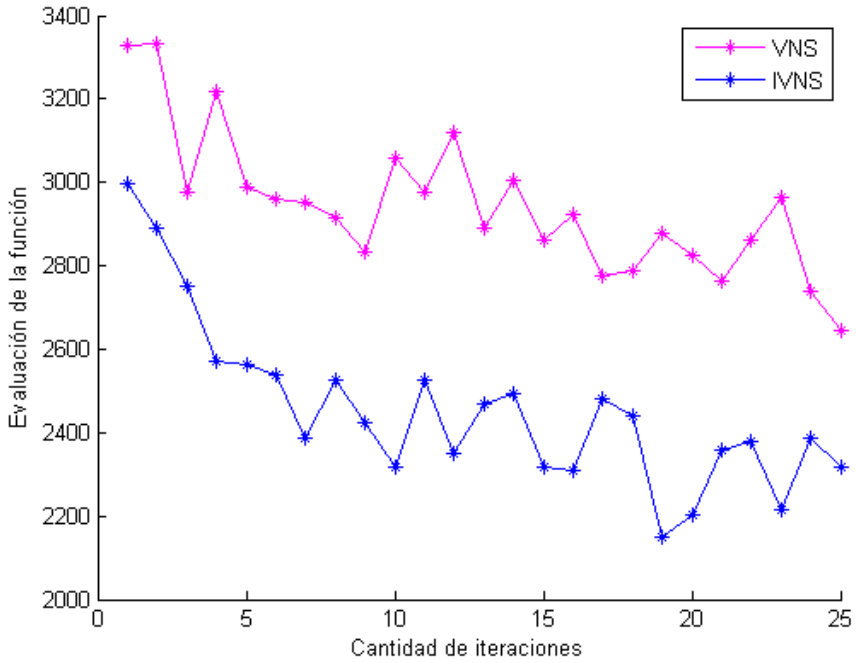


Figura 5.2: Óptimos considerando infinitos criterios para el problema A-n65-k9.

janos del mejor óptimo encontrado. En la siguiente sección se presentan algunas estrategias para tratar de mejorar los resultados de IVNS con respecto al mejor óptimo reportado en la literatura.

5.2. Mejorando IVNS

Una de las características del IVNS es que al aplicar muchos de los criterios de vecindad se obtienen soluciones no factibles. Por ejemplo, para una corrida de 17 000 iteraciones para el problema A-n33-k6 [33], se construyeron un total de 10 563 estructuras de entornos diferentes, de las cuales solamente 6 080 generaron soluciones vecinas factibles. El resto de los criterios, generaron soluciones vecinas no factibles, o no fue posible aplicarlos a la solución actual porque esta no tenía suficientes clientes en una ruta, o por-

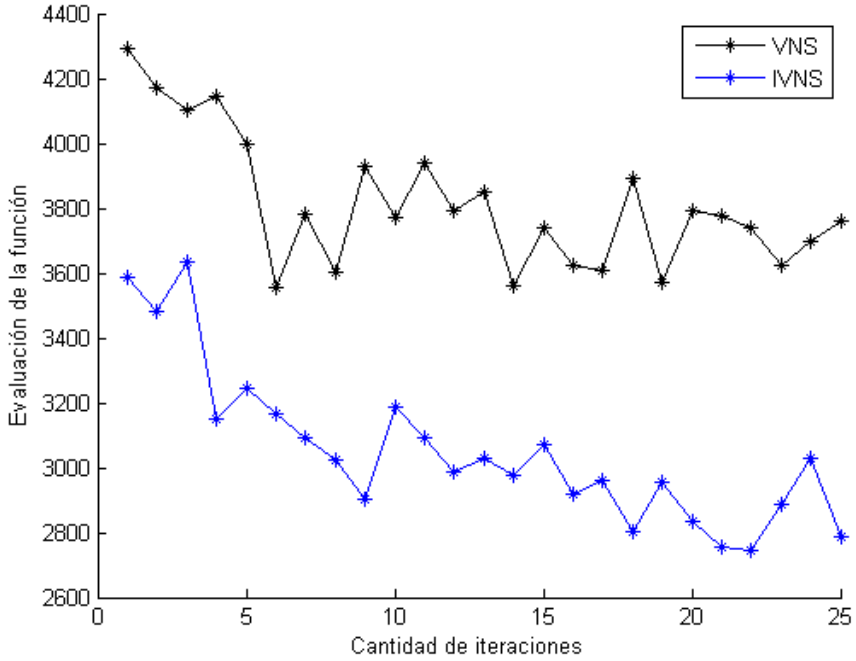


Figura 5.3: Óptimos considerando infinitos criterios para el problema A-n80-k10.

que ninguna ruta cumplía con los requisitos de los filtros. Esto hace que se desperdicie mucho tiempo y poder de cómputo que pudiera aprovecharse mejor.

La estrategia que se siguió para evitar lo anterior fue utilizar una estrategia de penalización. De esta forma, ninguna solución se rechaza por no ser factible y se permite una mayor exploración del espacio de búsqueda. Además, el algoritmo IVNS se modificó para aceptar soluciones peores que la actual cada cierto número de iteraciones, y comenzar a explorar las soluciones vecinas de estas soluciones peores.

Con esta modificación se pueden explorar muchas más soluciones, esto se puede apreciar en la gráfica de la figura 5.4. Por ejemplo para una corrida de 17 000 iteraciones se obtuvieron 12 500 soluciones vecinas, y solamente en 4500 casos fue imposible aplicar el criterio de vecindad. En este caso el criterio no se pudo aplicar porque hacía falta seleccionar más rutas de las

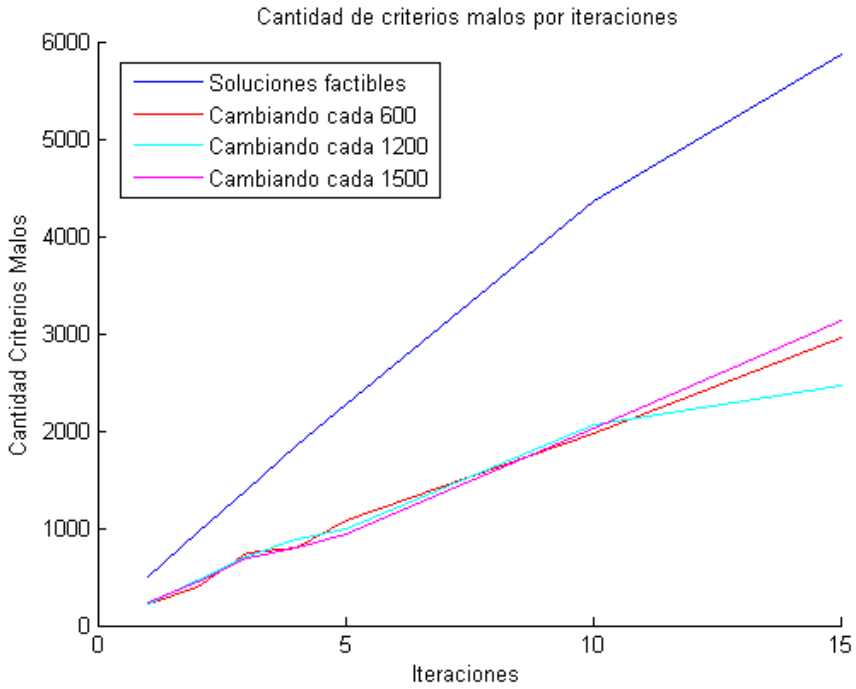


Figura 5.4: Óptimo para el problema A-n33-k6 con el algoritmo 2 y IVNS modificado.

existentes en la solución o extraer más clientes de una ruta que los que esta tenía.

Sin embargo, aunque la cantidad de criterios que no se pueden aplicar disminuye considerablemente (ver figura 5.4), y por tanto, se explora un mayor número de soluciones, los óptimos obtenidos por el algoritmo modificado son peores que los obtenidos por el algoritmo IVNS, como se puede apreciar en la figura 5.5 donde se muestra el comportamiento del algoritmo IVNS tradicional (en azul) y el resto de las curvas muestran los óptimos encontrado al aceptar soluciones peores cada 600, 1200, y 1500 iteraciones.

Una vez más, en el eje x están la cantidad de iteraciones realizadas por el algoritmo, y en el eje y, el valor del mejor óptimo obtenido en 30 iteraciones. Como se puede apreciar, el algoritmo original tiene un comportamiento mucho mejor, a pesar de evaluar muchas menos soluciones.

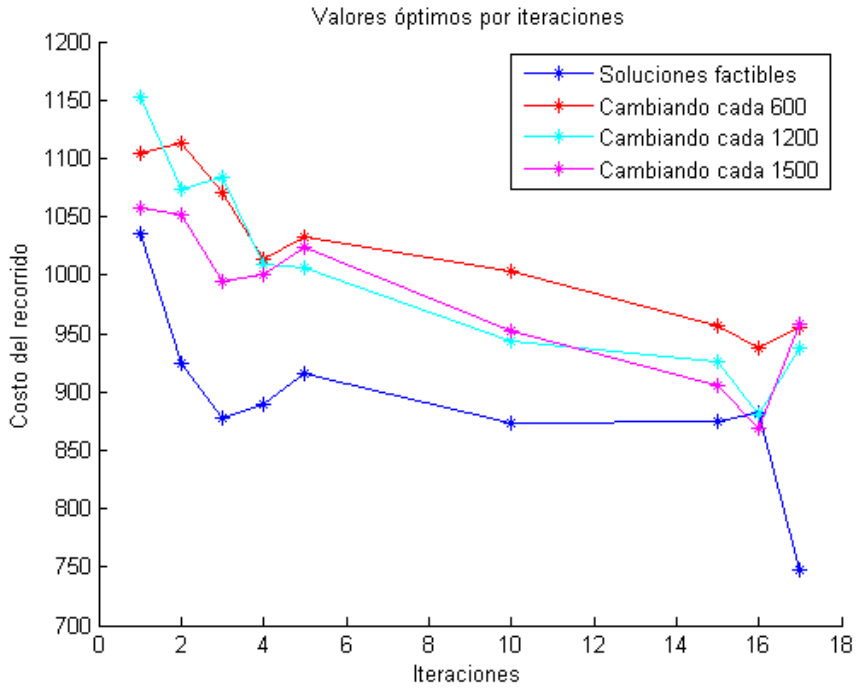


Figura 5.5: Óptimo para el problema A-n33-k6 con el algoritmo 2 y IVNS modificado.

Estos resultados parecen indicar que aumentar la búsqueda explorando soluciones infactibles no mejoran necesariamente la calidad del óptimo encontrado. Por ese motivo, quizás resulte más conveniente modificar la gramática para que todos los elementos del lenguaje generen soluciones vecinas factibles.

Conclusiones

En este trabajo se le realizó una modificación a la metaheurística VNS para considerar infinitos criterios de vecindad. Para ello se construyó una gramática libre del contexto cuyas oraciones son criterios de vecindad. Esto tiene como ventajas que permite implementar la generación automática del código fuente de los criterios a partir de las cadenas del lenguaje y agiliza el proceso de diseño y experimentación con nuevas propuestas de solución para el CVRP.

La creación del lenguaje fue posible porque todos los criterios de vecindad tienen elementos comunes como *seleccionar una ruta*, *seleccionar un cliente*, *insertar un cliente en una ruta*.

Luego de los experimentos realizados los resultados de IVNS parecen ser mejores que los del VNS tradicional, a pesar de que en las corridas realizadas se ha identificado un gasto de tiempo y esfuerzo computacional en soluciones que no podían ser usadas. Cuando se le modificó IVNS para que esto no sucediera, disminuyó considerablemente la cantidad criterios que no podían usarse, pero esto no mejora los óptimos alcanzados. Una idea que se pudiera explorar es modificar la gramática para obtener criterios de vecindad que solo generen soluciones factibles.

Recomendaciones

Como recomendaciones y trabajos futuros se propone aplicar IVNS a otros Problemas de Enrutamiento de Vehículos, para ello se debe modificar la gramática para que los criterios generados estén acordes a los problemas. Una forma de hacerlo es adicionando nuevos filtros a la gramática.

Además, se recomienda, a partir de los criterios generados por la gramática, generar el código de funciones que realicen una exploración exhaustiva de la vecindad, para incorporar estrategias de exploración de la vecindad al estilo de primera mejora, mejora aleatoria o búsqueda exhaustiva, que en la literatura reportan buenos resultados.

También se pudiera experimentar con otras estrategias de solución para el CVRP como algoritmos evolutivos, donde se puedan considerar infinitos criterios de mutación y cruzamientos, que puedan ser generados a partir de una gramática.

En estas direcciones deben estar dirigidos los próximos trabajos.

Bibliografía

[1] J. H. Dantzig, G. B.; Ramser. The truck dispatching problem. *Management Science*, 6, 10 1959. (Citado en las páginas 1 y 5).

[2] Paolo Toth; Daniele Vigo. *The Vehicle Routing Problem (Monographs on Discrete Mathematics and Applications)*. Monographs on Discrete Mathematics and Applications. SIAM, 2001. (Citado en las páginas 1, 5 y 15).

[3] Anand; Ochi Luiz Satoru Penna, Puca Huachi Vaz; Subramanian. An iterated local search heuristic for the heterogeneous fleet vehicle routing problem. *Journal of Heuristics*, 19, 04 2013. (Citado en la página 1).

[4] Leonora Bianchi; Ann Melissa Campbell. Extension of the 2-p-opt and 1-shift algorithms to the heterogeneous probabilistic traveling salesman problem. *European Journal of Operational Research*, 176, 2007. (Citado en la página 1).

[5] Alina Fernández Arias. El problema de enrutamiento de vehículos con recogida y entrega simultánea considerando una flota heterogénea. Master's thesis, Facultad de Matemática y Computación. Universidad de La Habana, La Habana, Cuba., 7 2010. (Citado en las páginas 2, 6, 7, 13 y 15).

[6] et al M. Gendreau. A tabu search heuristic for the heterogeneous vehicle routing problem. *International Journal of Computational Engineering Research*, 1991. (Citado en las páginas 2, 6 y 13).

[7] Jens Lysgaard · Adam N. Letchford · RichardW. Eglese. A new branch-and-cut algorithm for the capacitated vehicle routing problem. 2003. (Citado en la página 5).

[8] N. Christofides; A. Mingozzi; P. Toth. Exact algorithms for the vehicle routing problem, based on spanning tree and shortest path relaxations. *Mathematical Programming*, 20, 1981. (Citado en la página 5).

[9] C. D. T.; Christofides Nicos; de Neufville Richard Eilon, Samuel; Watson-Gandy. Distribution management-mathematical modelling and practical analysis. *IEEE Transactions on Systems Man and Cybernetics*, SMC-4, 1974. (Citado en la página 5).

[10] Gilbert Laporte. The vehicle routing problem: An overview of exact and approximate algorithms. *European Journal of Operational Research*, 59(3):345 – 358, 1992. (Citado en la página 5).

[11] Carlos Martínez Verónica Villanueva Raúl Pino, Jesús Lozano. Estado del arte para la resolución de enrutamiento de vehículos con restricciones de capacidad. 09 2011. (Citado en las páginas 5 y 15).

[12] G. Clarke and J. W. Wright. Scheduling of vehicles from a central depot to a number of delivery points. *Operations Research*, 12, 07-08 1964. (Citado en la página 5).

[13] Alan Wren, Anthony; Holliday. Computer scheduling of vehicles from one or more depots to a number of delivery points. *Journal of the Operational Research Society*, 23, 9 1972. (Citado en la página 5).

[14] Leland R. Gillett, Billy E.; Miller. A heuristic algorithm for the vehicle-dispatch problem. *Operations Research*, 22, 04 1974. (Citado en la página 5).

[15] J. Berger and M. Barkaoui. A new hybrid genetic algorithm for the capacitated vehicle routing problem. *Journal of the Operational Research Society*, 54(12):1254–1262, 2003. (Citado en la página 5).

[16] P. Tian X.-Y. Li and S. C. H. Leung. An ant colony optimization metaheuristic hybridized with tabu search for open vehicle routing problems. *Journal of the Operational Research Society*, 60, 07 2009. (Citado en la página 5).

[17] Ying; Hu Xiangpei Wang, Zheng; Li. A heuristic approach and a tabu search for the heterogeneous multi-type fleet vehicle routing problem with time windows and an incompatible loading constraint. *Computers and Industrial Engineering*, 11 2014. (Citado en las páginas 6 y 15).

[18] David Pisinger and Stefan Ropke. A general heuristic for vehicle routing problems. *Computers and Operations Research*, 34(8):2403 – 2435, 2007. (Citado en la página 6).

[19] James P. Osman, Ibrahim H.; Kelly. *Meta-Heuristics | | A Parallel Tabu Search Algorithm Using Ejection Chains for the Vehicle Routing Problem*, volume 10.1007/978-1-4613-1361-8. 1996. (Citado en la página 6).

[20] S.C. Ho; D. Haugland. A tabu search heuristic for the vehicle routing problem with time windows and split deliveries. *Computers and Operations Research*, 31, 2004. (Citado en la página 6).

[21] John H. Drake, Nikolaos Kililis, and Ender Özcan. *Generation of VNS Components with Grammatical Evolution for Vehicle Routing*, pages 25–36. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. (Citado en las páginas 6 y 35).

[22] Said; Jarboui Bassem; Eddaly Mansour Amous, Marwa; Toumi. A variable neighborhood search algorithm for the capacitated vehicle routing problem. *Electronic Notes in Discrete Mathematics*, 58, 04 2017. (Citado en las páginas 6 y 15).

[23] Dafne García de Armas y Alina Fernández Arias. Una estrategia de penalización aplicada al problema de enrutamiento de vehículos considerando una flota heterogénea, 2013. (Citado en las páginas 6, 7 y 14).

[24] Alina Fernández Arias y Sira María Allende Alonso. Modelos y métodos para el problema de enrutamiento de vehículos con recogida y entrega simultánea, 9 2013. (Citado en las páginas 6, 7, 13 y 14).

[25] Jeffrey D. Ullman John E. Hopcroft, Rajeev Motwani. *Introduction to automata theory, languages, and computation*. Addison-Wesley, 2nd ed edition, 2001. (Citado en la página 7).

[26] Ravichandhran Madhavan, Mikael Mayer, Sumit Gulwani, and Viktor Kuncak. Towards Automating Grammar Equivalence Checking. Technical report, 2015. (Citado en la página 7).

[27] Bruce McKenzie. Generating strings at random from a context free grammar, 1997. (Citado en la página 7).

[28] Pierre Hansen and Nenad Mladenović. First vs. best improvement: An empirical study. *Discrete Applied Mathematics*, 154(5):802 –

817, 2006. {IV} ALIO/EURO Workshop on Applied Combinatorial OptimizationIV ALIO/EURO Workshop on Applied Combinatorial Optimization. (Citado en las páginas 9 y 25).

[29] N Mladenovic. A variable neighborhood algorithm–a new metaheuristic for optimization combinatorial. In *Abstract of papers presented at Optimization Days, Montreal*, volume 12, 1995. (Citado en la página 9).

[30] Nenad Mladenović and Pierre Hansen. Variable neighborhood search. *Computers & operations research*, 24(11):1097–1100, 1997. (Citado en la página 9).

[31] José Andrés Moreno Pérez Pierre Hansen, Nenad Mladenovic. Variable neighbourhood search. inteligencia artificial, revista iberoamericana de inteligencia artificial. 19:77–92, 2003. (Citado en las páginas 12 y 15).

[32] Ingrid Morejón García. El problema de enrutamiento de vehículos con recogida y entrega simultánea. Master's thesis, Facultad de Matemática y Computación. Universidad de La Habana, La Habana, Cuba., 7 2014. (Citado en la página 15).

[33] NEO Networking and Emerging Optimization. Instancias de pruebas para vrp. http://neo.lcc.uma.es/vrp/vrpinstances/capacitated-vrp-instances/, 1 2013. (Citado en las páginas 34, 35 y 36).