UNIVERSIDAD DE SANTIAGO DE CHILE
FACULTAD DE INGENIERÍA
Departamento Ingeniería Informática

Generación Automática de Algoritmos para el Problema del Vendedor Viajero con Tiempo de Servicio en los Nodos

Franco Patricio Leal Guarda

Profesor guía: Víctor Parada Daza
Profesor co-guía: Carlos Contreras-Bolton

Tesis para optar al grado académico de Magíster de Ingeniería en Informática y al título de Ingeniero Civil en Informática.

Santiago – Chile
2021

RESUMEN

Actualmente, existen múltiples problemas de optimización combinatoria. Tales problemas pueden ser resueltos por métodos exactos, que requieren de un tiempo computacional muy alto. En consecuencia, se hace deseable contar con métodos que puedan determinar buenas soluciones en un tiempo menor. Un ejemplo de esos problemas corresponde al Problema del Vendedor Viaje con Tiempo de Servicio en los Nodos (TSP-TS), el que consiste en recorrer y atender un conjunto de ciudad a partir de un nodo que sirve de origen y destino minimizando el tiempo total. Una forma que se ha encontrado de disminuir el alto tiempo computacional requerido, se consigue generando algoritmos metaheurísticos automáticamente (GAA), utilizando como técnica base la Programación Genética (PG), la que genera diversas poblaciones de algoritmos representados como árboles y los evoluciona mediante operadores de mutación, cruzamiento y selección, simulando las leyes darwinianas Talbi (2009).Este trabajo determina si es posible aplicar la técnica de GAA para el TSP-TS con metaheurísticas clásicas como base, a partir del diseño del experimento computacional para posteriormente evaluar su desempeño frente a diversas instancias o ejemplos de prueba. Después, se debe compara la estructura de dichos algoritmos con las metaheurísticas base a fin de determinar patrones comunes y se compara su rendimiento con el mejor algoritmo metaheurístico de la literatura. Finalmente, se generan algoritmos capaces de resolver diversas instancias del TSP-TS, incluso algunas para los que no estaban diseñados originalmente, presentado mejoras significativas respecto al mejor algoritmo de la literatura.

Palabras Claves: Generación Automática de Algoritmos, Metaheurísticas, Hiper-heurísticas, Problema del Vendedor Viajero con Tiempo de Servicio en los Nodos.

ABSTRACT

Currently, there are many combinatorial optimization problems. These problems may be solved by exact methods, which require a very high computational time. Consequently, it is desirable to have methods that can determine good solutions in less time. One of these problems is the Traveling Salesman Problem with Time-Dependant Service Time (TSP-TS) ,which consists of going through and serving a set of cities from a node that serves as origin and destination, minimizing the total time.One way that has been found to reduce the high computational time required is achieved by automatically generating metaheuristic algorithms (GAA), using Genetic Programming (PG) as a base technique, which generates various populations of algorithms represented as trees and evolves them through operators of mutation crossing and selection, simulating the Darwinian laws Talbi (2009). This work determines if it is possible to apply the GAA technique for the TSP-TS with classical metaheuristics as a basis, from the design of the computational experiment to later evaluate its performance against various instances or test examples. Then, the structure of these algorithms must be compared with the base metaheuristics in order to determine common patterns and their performance is compared with the best metaheuristic algorithm in the literature. Finally, algorithms capable of solving various instances of the TSP-TS are generated, including some for which they were not originally designed, presenting significant improvements with respect to the best algorithm in the literature.

Keywords: Automatic Generation of Algorithms, Metaheuristics, Hyper-heuristics, Traveling Salesman Problem with Time-Dependant Service Time.

Dedicado a cada persona que hizo esto posible

AGRADECIMIENTOS

Agradezco a mi familia, especialmente a mi mamá, por su apoyo incondicional en toda mi vida. Agradezco también a mis compañeros de carrera y a mis amigos de la vida, sin los cuáles habría sido mucho más difícil el proceso.

TABLA DE CONTENIDO

ÍNDICE DE TABLAS

ÍNDICE DE ILUSTRACIONES

ÍNDICE DE ALGORITMOS

CAPÍTULO 1. INTRODUCCIÓN

Este capítulo tiene como objetivo mostrar una introducción al trabajo realizado, en el cual se presentan los antecedentes y la motivación para abordar el problema. Luego, se presentan los objetivos, alcances, metodologías utilizadas y la organización del documento.

1.1 ANTECEDENTES Y MOTIVACIÓN

Resolver problemas de optimización no siempre es una tarea trivial. De hecho, existen algunos problemas para los cuales la determinación de la solución óptima es una tarea altamente demandante en recursos. Específicamente, en tiempo computacional. Tales problemas se pueden clasificar según la teoría de la complejidad. Por una parte, se tienen los problemas para los que se puede encontrar una solución en un tiempo computacional que requiere de un numero polinomial de operaciones en relación con el tamaño de la entrada de datos, lo que se conoce como conjunto P. Por otro lado, existen otros problemas para los cuales la determinación de la solución óptima requiere de un algoritmo que ejecuta un numero exponencial de operaciones elementales, lo que se conoce como conjunto NP (Garey & Johnson, 1979). En la Teoría de la Complejidad, existe un desafío que consiste en mostrar la igualdad de tales conjuntos (Fortnow, 2009). Una manera de abordar estos problemas difíciles es utilizando metaheurísticas, que son técnicas que combinan heurísticas elementales. De esta forma, se puede buscar una solución aproximada para tales problemas, dado que encontrar la solución exacta es un desafío computacional.

El diseño de algoritmos para resolver problemas de optimización combinatoria es una tarea difícil. Existen métodos exactos para resolver problemas NP-Completo, como el Problema del Vendedor Viajero con Tiempo de Servicio en los Nodos (TSP-TS) (Cacchiani et al., 2020). Sin embargo, la ejecución de estos métodos pueden tardar un tiempo excesivamente alto, haciendo inviable su utilización para ejemplos de la vida real. Algunas aplicaciones son: ruteamiento de vehículos de empresas proveedoras de internet o televisión (tiempo de viaje entre hogares y tiempo de servicio en los mismos); ruteamiento de vehículos para mantención de transformadores de cableado eléctrico;disponibilidad de plazas de estacionamiento y accesibilidad al cliente en su ubicación (Taş et al., 2016). Debido a la alta complejidad para resolver estos problemas, se hace necesario diseñar métodos alternativos que puedan resolverlo en un bajo tiempo computacional y con una alta calidad de solución.

La generación automática de dispositivos electrónicos es un campo ampliamente explorado (Koza, 1992, 1994, 1999, 2003). En sus libros, John Koza propone distintas técnicas para generar automáticamente diversos dispositivos o máquinas a través de la Programación

Genética (PG). Se usa un árbol sintáctico para representar las componentes fundamentales de un dispositivo electrónico. Entonces, a partir de una población de tales árboles sintácticos, y combinándolos mediante selección, cruzamiento y mutación, emergen nuevas y novedosas combinaciones. En el año 2002, el autor registraba 12 patentes de nuevos dispositivos generados por el computador.

Siguiendo el mismo procedimiento para generar dispositivos electrónicos, es posible generar algoritmos automáticamente utilizando técnicas de PG. Al dividir diversos algoritmos en partes fundamentales con tareas específicas y combinándolas con instrucciones de alto nivel de los lenguajes de programación, se diseñan nuevos algoritmos capaces de resolver problemas de optimización. Existe evidencia en diversas publicaciones (Parada et al., 2016; Contreras Bolton et al., 2013; Acevedo et al., 2020),etc), en las que a partir de ciertos algoritmos se obtienen nuevos que son competitivos con otros métodos de optimización.

Las metaheurísticas son métodos de optimización aproximados muy potentes (Talbi, 2009). Estos métodos han demostrado tener un rendimiento muy competitivo a la hora de enfrentarlos a resolver problemas altamente complejos, como el TSP-TS, coloración de grafos, etc., lo que se apoya en que (Talbi, 2009) escribe un libro con una recopilación de las metaheurísticas creadas hasta la fecha. Luego, en una búsqueda reciente, se puede incluso encontrar un manual de metaheurísticas (Gendreau & Potvin, 2019), lo que indica la gran utilidad que pueden tener el uso de estos métodos aproximados.

Existe clara evidencia que la combinación de algoritmos metaheurísticos generan mejores resultados numéricos que la utilización de éstas por sí solas. Como descubren Jun et al. (2021), la combinación de algoritmos genéticos con búsqueda local genera mejores resultados que algoritmos genéticos o búsqueda por vecindades por si solos para el problema abordado. Por otro lado, en Martí et al. (2021) obtienen un mejor rendimiento que $GRASP$ utilizando una hibridación de algoritmo evolutivo con búsqueda local. Así mismo se presentan resultados similares en otras publicaciones (Kayé et al., 2021; Vela et al., 2020).

Existe evidencia de que las hiper-heurísticas tienen un buen funcionamiento con los problemas de optimización. En Burke et al. (2013), se presenta un resumen con el estado del arte de las hiper-heurísticas desde fines del año 2010, en el que se demuestran las bondades de tanto la selección de heurísticas como la generación de las mismas (Burke et al., 2013). Además, al realizar una búsqueda en el motor de Web of Science, se puede apreciar que del término $hyper - heuristics$ se desprenden alrededor de 353 resultados(Loyola et al., 2016); (Ryser-Welch et al., 2015) y otros).

Existen múltiples variantes del TSP clásico, en las cuales se introduce el tiempo como una variable importante para el modelamiento de diversos problemas con aplicaciones a la vida real, considerando tiempo con los nodos, ventanas de tiempo, entre otros(Cacchiani et al., 2020).

Una de esas variantes que considera tiempo en los nodos, es útil para representar situaciones en las cuales en cada nodo se deben realizar operaciones de atendimiento y se dispone de un tiempo limitado para llevarlas a cabo. Un ejemplo de tales situaciones ocurre cuando un vehículo debe distribuir productos en distintos lugares de una región y para realizar esta tarea en cada lugar existe un horario disponible y un tiempo de atención. Se trata de una situación común, por ejemplo en los camiones que reparten productos de retail, o los vehículos de mantenimiento que deben inspeccionar las estaciones subeléctricas de una ciudad.

1.2 DESCRIPCIÓN DEL PROBLEMA

El problema del Vendedor Viajero con Tiempo de Servicio en los Nodos (TSP-TS), es un problema difícil de resolver. De hecho, pertenece al conjunto NP-completo. El TSP-TS consiste en determinar un circuito que pasa por todos los vértices de un grafo visitando cada uno una única vez y que comienza y termina en el mismo nodo. Tal circuito debe tener el mínimo costo y debe atender restricciones de tiempo de servicio en los nodos (Taş et al., 2016).

El problema ha sido recientemente presentado en la literatura como una extensión del clásico problema del vendedor viajero (Applegate et al., 2011). Actualmente, existen algoritmos que son capaces de resolver algunas instancias del problema. Sin embargo, se identifica una brecha, debido a que hay una baja cantidad de publicaciones en el tratamiento del TSP-TS, dado que el problema es relativamente nuevo (Taş et al., 2016). Luego, en la publicación de Cacchiani et al. (2020), es posible apreciar que los métodos propuestos permiten encontrar soluciones óptimas para ejemplos de máximo 58 nodos. Debido a que el tamaño de estos ejemplos es pequeño en comparación a otros conjuntos de datos existentes, se hace necesario encontrar nuevas formas de resolver el problema del TSP-TS.

Dado lo anterior, se hace necesario diseñar algoritmos eficientes que resuelvan el TSP-TS en un espectro de instancias mayor . Para esto, es posible utilizar la técnica de GAA mediante la combinación de componentes fundamentales extraídos de las metaheurísticas clásicas existentes, produciendo algoritmos más eficientes.

Se plantean dos preguntas de investigación que esperan ser respondidas: ¿Es posible generar metaheurísticas automáticamente a partir de componentes elementales de metaheurísticas clásicas? Tales metaheurísticas, ¿pueden competir con las metaheurísticas existentes en relación con la calidad de la solución y el tiempo de ejecución?

La hipótesis que se pretende probar con este trabajo es la siguiente: ”Se pueden generar algoritmos metaheurísticos automáticamente utilizando un computador para resolver el TSP-TS y que sean igual o más eficientes que los existentes actualmente en relación a la calidad

de la solución y el tiempo de ejecución”

1.3 SOLUCIÓN PROPUESTA

La solución propuesta para el problema explicado es nuevos algoritmos metaheurísticos automáticamente mediante la PG con la finalidad que sean competitivos con los existentes y producir novedosas combinaciones metaheurísticas. En otras palabras, se busca generar estos algoritmos con una máquina que explora el espacio de los algoritmos que resuelven el problema del TSP-TS. Para generar estos algoritmos, se combinarán componentes fundamentales de metaheurísticas clásicas que han demostrado tener la capacidad de resolver problemas de complejidad combinatoria.

Este hecho tiene impacto sobre la investigación científica en el campo tanto del problema TSP-TS como en el problema P igual NP. En el primer caso, la maquina generadora de algoritmos, (en este caso la PG) puede resultar en una potente herramienta para explorar numerosas combinaciones de componentes elementales de componentes metaheurísticas que no han sido exploradas hasta ahora. En el segundo caso, al combinar gran cantidad de algoritmos y producir por miles, podrían surgir nuevas ideas algorítmicas que alimenten a los investigadores de este campo. En el ámbito tecnológico, contar con algoritmos eficientes para el TSP-TS permitirá producir herramientas de software comerciales que apoyen la gestión de la distribución de vehículos.

1.4 OBJETIVO GENERAL Y ESPECÍFICOS

1.4.1 Objetivo General

Generar algoritmos metaheurísticos automáticamente, utilizando técnicas de Programación Genética con componentes derivados de metaheurísticas clásicas para el Problema del Vendedor Viajero con Tiempo de Servicio en los nodos.

1.4.2 Objetivos Específicos

- Diseñar un experimento computacional para la generación automática de algoritmos metaheurísticos.
- Generar nuevos algoritmos metaheurísticos para la resolución del TSP-TS.
- Evaluar el desempeño computacional de los algoritmos utilizando distintos conjuntos de datos respecto a calidad de la solución y tiempo de ejecución.
- Comparar las nuevas estructuras algorítmicas construidas con las metaheurísticas existentes, con el fin de determinar patrones comunes.

1.4.3 Alcances

El trabajo tiene como objetivo generar algoritmos metaheurísticos automáticamente utilizando técnicas de PG. Se deben tener en consideración algunos puntos:

- Se utiliza un conjunto acotado de instancias para realizar el experimento, debido a que lo costosa que es computacionalmente la experimentación. Estas se obtienen de las utilizadas por Taş et al. (2016) y Cacchiani et al. (2020).
- Se espera generar una gran familia de algoritmos. sin embargo, solo se evaluarán con mayor detalle los mejores algoritmos (elegidos según criterios explicados en el trabajo).
- Es importante aclarar que se busca resolver el problema científico, más no su aplicación a problemas del mundo real.

1.5 METODOLOGÍA Y HERRAMIENTAS UTILIZADAS

1.5.1 Método de trabajo

La metodología de investigación se basa en cuatro partes, según los objetivos que debe cumplir:

- Preparación del proceso evolutivo de los algoritmos metaheurísticos

- Efectuar el experimento (evolucionar algoritmos)
- Obtener métricas de los resultados (obtener datos de la evolución)
- Evaluar resultados (analizar los resultados obtenidos)

1.5.2 Procedimiento experimental

Se debe determinar el procedimiento necesario para poder verificar si la hipótesis planteada se aprueba o se rechaza. Para lo anterior, se deben ejecutar distintas etapas:

- Determinar funciones y terminales: Las funciones elegidas corresponden a operadores básicos transversales a los lenguajes de programación y sirven de nodos internos en las estructuras de árbol de los algoritmos, tales como operaciones lógicas (Or, And, NOT), recursos iterativos (While, Do While) y recursos condicionales (If, If Then). Por otro lado, se definen los terminales que corresponden a las hojas de los árboles y ejecutan operaciones sobre las soluciones y otros. Los terminales emergen desde las metaheurísticas clásicas.
- Elección de instancias: las instancais elegidas surgen de la revisión de la literatura y se presentan posteriormente en la sección de Conjunto de ejemplos (3.1.6).
- Función de calidad o $fitness$: Función que permite evaluar la calidad con que un algoritmo resuelve una instancia.
- Parámetros del proceso evolutivo: Corresponden a valores numéricos que deben ser determinamos previa la realización de la experimentación.

1.5.3 Programación Genética

Corresponde a una metaheurística de población perteneciente a la Computación Evolutiva (CE), inspirada en la teoría de la evolución darwiniana. El proceso para realizar la evolución se observa gráficamente en la figura 1.1. En primer lugar, se genera una población de soluciones iniciales para el problema planteado (para la PG, corresponden a árboles). Luego, se seleccionan algunos de estos individuos según un puntaje o $fitness$ que se determina por la calidad de la solución. Se aplica luego un operador de reproducción sobre esta población para generar nuevas soluciones. Estas nuevas soluciones son sometidas probabilísticamente a un operador de mutación, el que realiza una pequeña alteración en la solución con el fin

aumentar la exploración del espacio de búsqueda. Finalmente, se calcula el $fitness$ de estas nuevas soluciones, la que el proceso de selección indicarán cuales son las que tienen una mayor probabilidad de sobrevivir y ser parte de la nueva población. Este proceso se hace iterativamente, haciendo evolucionar las soluciones hasta que se cumple algún criterio de detención definido con anterioridad.

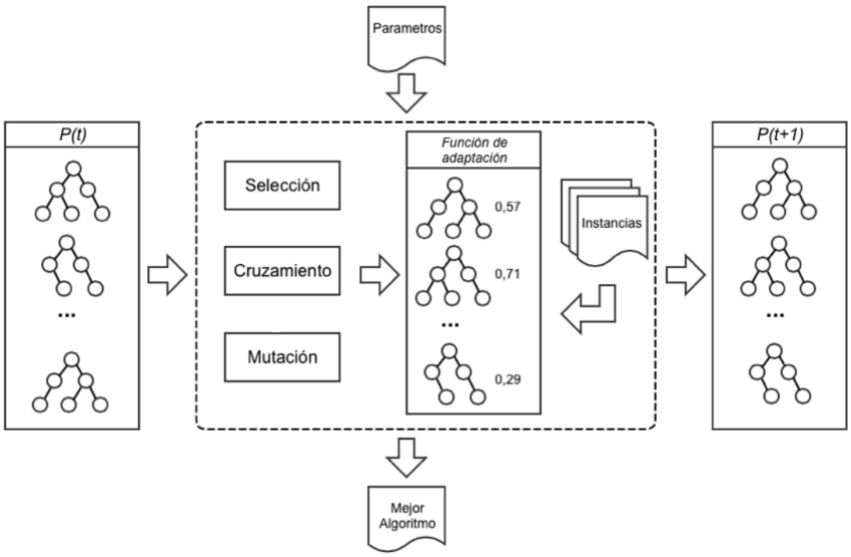

Figura 1.1: Flujo evolución PG. Fuente:

1.5.4 Plataforma de evolución

Para realizar el proceso evolutivo, se utiliza la herramienta de computación evolutiva ECJ 22 (Evolutionary Computation Journal (Luke, 2010)). Esta herramienta se encuentra desarrollada en el lenguaje Java y provee de un entorno para realizar experimentos basados en Computación Evolutiva (ej: algoritmos genéticos, programación genética, etc). Permite definir metaparámetros para el proceso evolutivo, modificar estructuras del sistema sencillamente y generando procesos evolutivos eficientemente. Además, es una herramienta que se utiliza ampliamente y cuenta con un manual disponible en su sitio web y con una gran cantidad de ejemplos de su uso (Luke, 2010).

1.5.5 Herramientas de desarrollo

Para realizar el desarrollo de la tesis, se debe utilizar un computador. Ene ste caso, se utiliza el equipo personal del autor.en esta sección se describen sus características.

Hardware

- Procesador: i7-7500U
- Almacenamiento: 250 GB SSD y 1TB HDD
- RAM: 16GB

Software

- Sistema Operativo: Ubuntu 18.04
- Plataforma de evolución: ECJ versión 22.
- Software gráfico: Graphviz, para representacion de árboles
- Agrupamiento K-Means: módulo kmeans de SKLearn. Python.
- Generación de gráficos: programa personal desarrollado en Python. Se utilizan módulos externos como Matplotlib y Seaborn.

1.6 ORGANIZACIÓN DEL DOCUMENTO

El documento cuenta con 5 capítulos principales, descritos a continuación:

1. Introducción: contiene información preliminar para un mejor entendimiento del la tesis. Su contenido incluye contexto y motivación, descripción del problema, descripción de la solución y herramientas y métodos.

2. Revisión bibliográfica: incluye una revisión de las publicaciones asociadas tanto al problema de optimización que se busca resolver, como del método que se utiliza para realizar la evolución de los algoritmos.

3. Diseño experimental: Se explica todo lo necesario para preparar el ambiente para la realización del experimento, tales como valores de los parámetros de la evolución, forma de almacenamiento de datos, función de evaluación, entre otros.

4. Análisis de resultados: en este capítulo se muestra y analizan todos los resultados obtenidos de la ejecución y evaluación del experimento, tales como curvas de convergencia de los errores, evaluación de rendimiento de algoritmos generados, análisis de estructura de algoritmos y comparación con la literatura.

5. Conclusiones: incluye las conclusiones derivadas del análisis de los resultados obtenidos, cumplimiento de objetivos y se expone información referente a trabajo futuro.

CAPÍTULO 2. REVISIÓN DE LA LITERATURA

2.1 TSP-TS

En la literatura del Problema del Vendedor Viajero, los trabajos que consideran la dependencia del tiempo se centran en la variabilidad de los tiempos de viaje, llamados Problema del Vendedor Viajero dependientes del tiempo (TDTSP) (Cacchiani et al., 2020). En general, el costo del arco depende de la posición de este en la posición que tiene la ruta en la solución. La finalidad es determinar el costo mínimo de la planificación de una secuencia de tareas, en que la transición de un nodo i a un nodo j en un tiempo t tiene un costo c_{ijt}, en el que se procesa la tarea i (Cacchiani et al., 2020). En Picard & Queyranne (1978), se proponen tres formulaciones de Programación Entera: una generalización del Problema del Vendedor Viajero (TSP) en que los costos dependen del tiempo, una formulación basada en el Problema de Asignación Cuadrática (QAP) y una basada en grafos en capas. Cuando todos los nodos tienen la misma función de tiempo, el TSP-TS se puede formular como un TDTSP en el modelo de grafo por capas propuesto en Picard & Queyranne (1978).

Por otra parte, en (Vander Wiel & Sahinidis, 1996), se aplica a un modelo de Programación Entera Mixta (MIP) con una descomposición de Benders, el que corresponde a una linealización del QAP propuesto en Picard & Queyranne (1978), combinado con cortes de Benders óptimos de Pareto. Dos de las formulaciones propuestas en Picard & Queyranne (1978) son mejoradas en Bigras et al. (2008) generando un algoritmo de Branch-and-Cut-and-Price. Luego, en la publicación de Abeledo et al. (2013) se estudia la formulación propuesta en Picard & Queyranne (1978), generando un nuevo algoritmo de Branch-and-Cut-and-Price. Dos de los algoritmos propuestos en Picard & Queyranne (1978) y Vander Wiel & Sahinidis (1996) son comparados en Miranda-Bront et al. (2014)miranda, proponiendo además desigualdades y caras del polítopo, incluyéndolos en una mejora al Branch-and-Cut. Finalmente, en Kinable et al. (2017) se propone un método para resolver los problemas de secuenciación dependientes del tiempo, en el que el tiempo entre dos tareas dependen de la posición de estos en el orden de las tareas. Por otra parte, existe otro grupo de trabajos que consideran que el costo de viaje del arco (i, j) depende del tiempo en que el vehículo deja el nodo i. En Cordeau et al. (2014), el tiempo se divide en intervalos y se asume que la velocidad promedio en cada intervalo es conocida. Luego, el tiempo de viaje se calcula según la función definida en Ichoua et al. (2003). Además, en Cordeau et al. (2014) se define que la velocidad en un arco durante un intervalo de tiempo depende de la velocidad máxima de viaje del arco, el menor factor de congestión y el factor de descenso de la congestión. Luego, se propone un modelo de Programación Entera Lineal (ILP)

con restricciones de eliminación de subrutas. En Arigliano et al. (2018) se propone un algoritmo de Branch-and-Bound para el mismo problema y se prueba en los mismos ejemplos y otras más grandes.

Luego, al problema propuesto en Cordeau et al. (2014), se le agregan restricciones de ventanas de tiempo. Esta extensión se propone primero en Arigliano et al. (2019), en el que un modelo ILP con desigualdades, desarrollando un algoritmo de Branch-and-Cut. En Montero et al. (2017) se propone una nueva formulación basada en la de Sun et al. (2018), para el TDTSP rentable con ventanas de tiempo y recogida y entrega (Profitable Time- Dependent TSP with Time Windows and Pickup and Delivery), junto a un algoritmo de Branch-and-Cut que lo resuelve. En Minh Vu et al. (2018) se propone un nuevo algoritmo para el TSP con ventanas de tiempo, basado en la representación de red de tiempo expandido, el que confía en un marco de trabajo de Descubrimiento de Discretización Dinámica, propuesto en Boland et al. (2017), y probado posteriormente con el benchmark de Arigliano et al. (2019), resolviendo todos los ejemplos en un corto período de cómputo.

Finalmente, el TSP-TS es un problema de optimización combinatoria que pertenece a la clase NP-Completo, que fue introducido recientemente en Taş et al. (2016), donde se propone un modelo de Programación Entera Mixta (MIP), junto a cotas superiores e inferiores. Se define sobre un grafo completo dirigido $G = (V, A)$. El conjunto de nodos $N = 0, 1, ..., n, n + 1$ contiene un conjunto de clientes (que corresponden a los nodos 1,...,n) y el nodo de origen (corresponde al 0, n+1. Se duplica por conveniencia). Por otro lado, el conjunto A contiene un arco (i, j) para conectar cada par de nodos que tienen un tiempo de viaje asociado i, j. Además, cada cliente $i \in N \setminus \{0, n + 1\}$ posee un tiempo de servicio definido por una función continua $s_i(b_i$, en el que b_i representa el tiempo en el que comienza del servicio del nodo i. Dado esto, $s_0(b_0) = 0$ y $s_{n+1}(b_{n+1}) = 0$, debido a que el nodo de origen no requiere de un tiempo de servicio (Traducido de (Cacchiani et al., 2020).

Se debe encontrar la distancia Hamiltoniana para encontrar la solución al TSP-TS. Esto quiere decir que se debe encontrar un camino desde el nodo 0 al $n + 1$ que visite todos los nodos del grafo una vez, de manera tal que se minimice el tiempo total del viaje. Este tiempo se encuentra dado por la suma de viajar entre cada par de nodos y el tiempo de servicio de cada uno de ellos.

En la literatura, se proponen diversos métodos tanto exactos como heurísticos para resolver el TSP-TS. Dentro de la familia de los métodos exactos, en (Cacchiani et al., 2020) se proponen Branch-and-Cut (BC) y Dynamic Branch-and-Cut(D-BC). A la hora de probar D-BC, los tiempos para resolver ejemplos de TSP-TS son menores que los propuestos en (Taş et al., 2016). Dentro de los algoritmos metaheurísticos, en (Cacchiani et al., 2020) se propone el uso de Algoritmos Genéticos (GA), obteniendo tiempos computacionales mucho menores respecto a los

métodos exactos, a cambio de no obtener el óptimo en todos los casos (Cacchiani et al., 2020).

En (Cacchiani et al., 2020) se expone el modelo matemático básico, descrito a continuación:

$$min \sum_{i \in N} \sum_{j \in N} t_{ij} x_{ij} + \sum_{i \in N \setminus \{0, n+1\}} s_i(b_i) \tag{2.1}$$

$$\sum_{j \in N \setminus \{i\}} x_{ij} = 1, i \in N \setminus \{n+1\} \tag{2.2}$$

$$\sum_{i \in N \setminus \{j\}} x_{ij} = 1, j \in N \setminus \{0\} \tag{2.3}$$

$$b_i + s_i(b_i) + t_{ij} - M(1 - x_{ij}) \leq b_j, \quad i \in N, j \in N, \tag{2.4}$$

$$b_i \geq 0, \quad i \in N, \tag{2.5}$$

$$x_{ij} \in \{0, 1\}, i \in N, j \in N \tag{2.6}$$

Se tiene un nuevo modelo propuesto por (Cacchiani et al., 2020), implementando diversas mejoras al modelo base:

$$min \sum_{i \epsilon N} \sum_{j \epsilon N} t_{ij} x_{ij} \tag{2.7}$$

$$\sum_{j \epsilon N \setminus \{i\}} x_{ij} = 1, i \epsilon N \setminus \{n+1\} \tag{2.8}$$

$$\sum_{i \epsilon N \setminus \{j\}} x_{ij} = 1, j \epsilon N \setminus \{0\} \tag{2.9}$$

$$r_i x_{ij} \leq y_{ij} \leq d_i x_{ij}, i \epsilon N \setminus \{0, n+1\}, j \epsilon N \setminus \{0\}, i \neq j \tag{2.10}$$

$$\sum_{i \epsilon N \setminus \{0, n+1\}, i \neq j} y_{ij} + \sum_{i \epsilon N \setminus \{n+1\}, i \neq j} t_{ij} x_{ij} \leq \sum_{k \epsilon N \setminus \{0\}, k \neq j} y_{jk}, j \epsilon N, \{0, n+1\} \tag{2.11}$$

$$x_{i,j} \epsilon \{0, 1\}, i \epsilon N, j \epsilon N \tag{2.12}$$

$$y_{ij} \geq 0, i \epsilon \setminus \{0, n+1\}, j \epsilon N \setminus \{0\}, i \neq j \tag{2.13}$$

Para efectos de este trabajo, se decide trabajar con el modelo matemático compuesto

por las ecuaciones 2.1, 2.2, 2.3, 2.4, 2.5, 2.6. Es importante aclarar que el modelo matemático solo contempla el problema científico y no su aplicación a problemas de la vida real, los que podrían incluir más restricciones.

2.2 GENERACIÓN AUTOMÁTICA DE ALGORITMOS

La Generación automática de Algoritmos (GAA) es una técnica que permite ensamblar automáticamente partes de algoritmos que pueden resolver un problema dado.(Loyola et al., 2016; Ryser-Welch et al., 2015). El problema de encontrar el mejor algoritmo que resuelve un problema puede ser formulado como un metaproblema de optimización (Acevedo et al., 2020). Se puede realizar el proceso de búsqueda de este mejor algoritmo optimizando el problema descrito en 2.14 , donde A_π es un algoritmo que resuelve un problema de optimización π, dentro de un espacio de todos los posibles algoritmos que lo resuelvenω_π:

$$A_I : MinimizarRMSE(y_i, y_{Aj}) \tag{2.14}$$

Donde:

- A_I: Mejor algoritmo para un conjunto de ejemplos I
- y_i: Óptimo del ejemplo i
- y_{Ai}: Óptimo del ejemplo i encontrado por A_I

2.3 EJEMPLO DE ALGORITMO

A partir de la revisión de la literatura, se puede resumir que existe una área que busca generar algoritmos automáticamente (Loyola et al., 2016);(Ryser-Welch et al., 2015). Debido a que los algoritmos se pueden representar como árboles sintácticos, se puede utilizar la PG para aplicar la evolución de éstos algoritmos. Al aplicar ésta técnica, se trabaja en el campo de las hiperheurísticas, el que propone ser un método que permite generar algoritmos a partir de heurísticas o metaheurísticas base (Burke et al., 2013).

Un algoritmo obtenido desde la literatura, es el presentado en la figura 2.1, en el cual se presenta como árbol sintáctico. Este algoritmo resuelve el TSP, y representa el algoritmo descrito en Algoritmo 2.1. Este algoritmo utiliza el terminal $nearest - insertion()$, el que agrega

el nodo en la ruta que es mas cercano a alguno de las otros nodos de la ruta. También utiliza $best - neighbor()$, el que agrega el nodo en la ruta que está más cerca al último nodo visitado. Luego, utiliza $near - center()$, el que agrega el nodo más cercano al nodo central en la ruta. Después, si puede agregar el nodo más lejano al nodo central ($far-center()$), aumenta el contador que indica la cantidad de nodos visitados. Luego, si puede agregar el $worst - neighbor()$, lo agrega y aumenta el contador de los nodos. Después, vuelve a verificar si puede agregar el $worst - neighbor()$ y, mientras queden nodos por agregar, utiliza $nearest - insertion()$ junto a $2 - Opt$, el que toma 2 arcos de la solución parcial y los reconecta (intercambiados). Si la solución mejora, se actualiza.

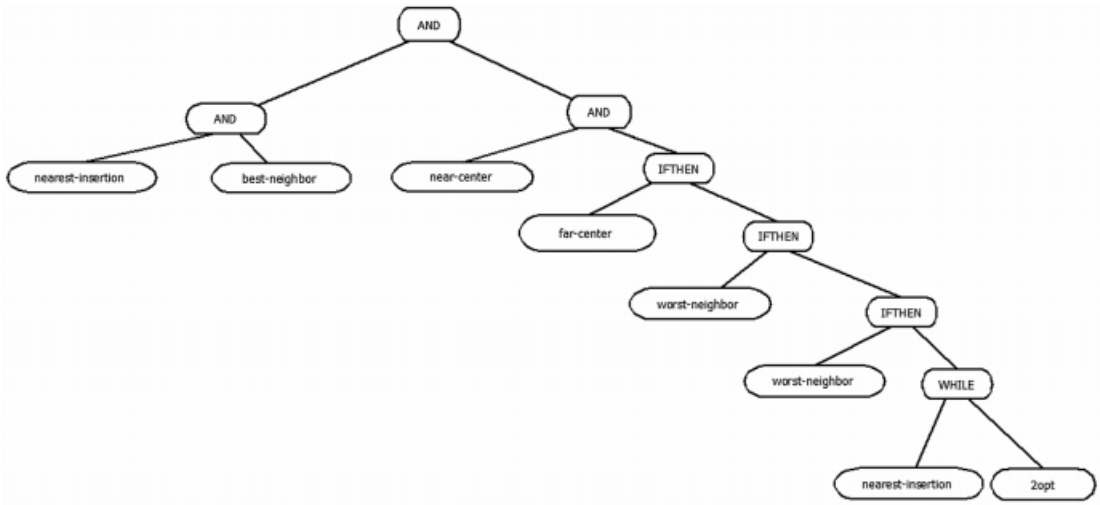

Figura 2.1: Árbol del algoritmo TSP11. Fuente: Loyola et al. (2016).

Algoritmo 2.1: TSP11. Fuente Loyola et al. (2016)

```
1:  set i = 0
2:  nearest-insertion ();
3:  best-neighbor ();
4:  near-center ();
5:  i ← 3;
6:  if far-center() = 1 then
7:     i ← i + 1;
8:     if worst-neighbor () = 1 then
9:        i ← i + 1;
10:       if worst-neighbor () = 1 then
11:          i ← i + 1;
12:          while i < c do
13:             nearest-insertion ();
14:             i ← i + 1;
15:             2-Opt ();
16:          end while
17:       end if
18:    end if
19: end if
```

CAPÍTULO 3. DISEÑO EXPERIMENTAL

Este capítulo explica el procedimiento experimental llevado a cabo para la investigación. Se presenta el entorno utilizado para la evolución, la metodología, consideraciones propias de la experimentación y definición de componentes necesarios.

3.1 METODOLOGÍA DEL EXPERIMENTO

En esta sección, se describen las dos etapas que deben ser desarrolladas para llevar a cabo el experimento propuesto.

1. Preparación del proceso evolutivo de los algoritmos metaheurísticos:Corresponde al diseño, implementación y ejecución del proceso evolutivo que permite generar automáticamente algoritmos. Para esto, se deben realizar múltiples tareas:

    - Definir plataforma de evolución.
    - Definición del contenedor de la solución.
    - Definición de funciones
    - Definición de terminales a partir de las metaheurísticas seleccionadas y su descomposición.
    - Determinar función de $fitness$ o de evaluación para medir calidad de los algoritmos generados.
    - Determinar ejemplos propios del problema
    - Determinar meta-parámetros del proceso evolutivo

2. Evaluación de rendimiento de los algoritmos: En esta etapa se debe verificar el desempeño de los algoritmos, enfrentándolos a ejemplos que no se utilizan en el proceso evolutivo para generar algoritmos. Los ejemplos utilizadas tanto para el proceso evolutivo como para la evaluación corresponden a problemas resueltos del TSP-TS obtenidos de la literatura. Para comparar los resultados obtenidos, se utiliza el Error Relativo Promedio (ERP).

    ERP: Calcula el valor absoluto de las diferencias entre el costo óptimo real de un ejemplo i (c_{Ri}) y el costo óptimo promedio obtenido por el algoritmo (c_{ai}) dividido en el costo real. El costo promedio se obtiene ejecutando 10 veces el algoritmo sobre un ejemplo, y promediando los $fitness$, debido a que los algoritmos pueden presentar terminales probabilísticos (Cacchiani et al., 2020). Luego, se suman los errores obtenidos al evaluar

sobre todos los ejemplos de evolución dividido en la cantidad de ejemplos. Este cálculo se especifica en la fórmula 3.1.

$$ERP_a : \frac{1}{|IE|} \sum_{i \epsilon IE} \left| \frac{c_{ai} - c_{Ri}}{c_{Ri}} \right| \tag{3.1}$$

Donde:

- IE: Conjunto de ejemplos de evaluación.
- c_{ai}: Costo promedio encontrado por el algoritmo para el ejemplo i.
- c_{Ri}: Costo real óptimo del ejemplo i.

Además, se espera que los algoritmos generados sean competitivos en calidad de la solución respecto a los métodos existentes.

3.1.1 Contenedores de la solución

Para representar la solución del TSP-TS se elige un conjunto de enteros almacenados en una estructura del tipo $ArrayList$ de Java. Cada uno de los enteros representa el orden en el que se visitan las ciudades, desde la primera a la penúltima (dado que la última corresponde a la primera). Para cada algoritmo, se definen 3 distintos contenedores, a fin de utilizarlos en el proceso de construcción de la mejor solución:

- Solución propuesta: Contenedor que es utilizado por los terminales de perturbación de la solución. Es una solución que puede ser promovida a solución actual.
- Solución actual: Si un terminal de aceptación cumple las condiciones, la solución propuesta es promovida a solución actual. Idealmente, el contenedor de solución actual tiene la mejor solución. Sin embargo, esto puede no ser así debido a los terminales de aceptación probabilísticos.
- Solución auxiliar: Contenedor utilizado por los terminales de búsqueda local para comparar su solución con la propuesta por iteración.

3.1.2 Funciones

Corresponden a operaciones elementales que actúan sobre las diversas estructuras de almacenamiento de información. Las funciones a utilizarse son las que describen a

continuación:

- While(A,B): Mientras A retorne verdadero, se ejecuta B. Esta función devuelve verdadero si se ejecuta al menos una iteración. Devuelve falso en caso contrario.
- IfThen(A,B): Si A devuelve verdadero, se ejecuta el terminal B. Devuelve verdadero si se ejecuta al menos una vez B y falso en caso contrario.
- IfThenElse(A,B,C): Si A devuelve verdadero, ejecuta B. Si A devuelve falso, se ejecuta C. Devuelve el valor de verdad de B o C según sea el caso.
- DoWhile(X,Y): Se ejecuta una vez Y. Mientras X retorne verdadero, Y se ejecuta un máximo de n veces. Retorna verdadero si X retorno verdadero al menos una vez.
- Not(A): Función que implementa la operación lógica de la negación. Si A es verdadero devuelve falso, en caso contrario devuelve verdadero.
- And(A,B): Función que implementa la operación lógica de la conjunción. Si A y B son verdaderos devuelve verdadero. en todos los otros casos devuelve falso.
- Or(A,B): Función que implementa la operación lógica de la disyunción. Si A y B son falsos, devuelve falso. En todos los otro casos devuelve verdadero.

Para el proceso de generación de los algoritmos representados por árboles sintácticos, las funciones representan los nodos internos de los árboles.

3.1.3 Metaheurísticas

Para el desarrollo del proceso evolutivo, es necesario determinar operaciones que modifiquen los contenedores de modo que se pueda generar una convergencia a la hora de encontrar soluciones de buena calidad. En este caso, se utilizan componentes fundamentales derivadas de metaheurísticas existentes en la literatura. Las metaheurísticas elegidas son las siguientes:

Simulated Annealing

Simulated annealing (SA) corresponde a un algoritmo metaheurístico que permite explorar el espacio de soluciones, permitiendo escapar de óptimos locales.Esto se logra aceptando

de manera probabilística y contralada soluciones de poer calidad a la actual, según en que etapa iterativa de la ejecución se genere. Primero se genera una solución inicial al problema. Luego, se determina una temperatura inicial (máxima) desde la cuál comienza el proceso iterativo. Luego, mientras no se alcance un criterio de término (por ejemplo, que la temperatura actual sea menor o igual a la temperatura final) y mientras no se alcance una condición de equilibrio (a temperatura constante), genera un vecino aleatorio de s' a partir de s. Luego, se calcula la diferencia entre el valor objetivo de s' y s y, si esta diferencia es menor que 0 (es decir que la solución nueva sea mejor que la actual), se acepta la nueva solución. En cambio, si la nueva solución es de peor calidad, es posible aceptarla con una probabilidad $e^{\frac{-\Delta E}{T}}$. Esta probabilidad es más grande mientras mayor sea la temperatura actual, por lo que al comienzo del proceso es más fácil aceptar soluciones de peor calidad, y también se hace grande mientras menor sea la diferencia entre ambas soluciones (Talbi, 2009). Después, se procede a realizar un decrecimiento de la temperatura según un patrón de enfriamiento. Finalmente, se devuelve la mejor solución encontrada.

Algoritmo 3.1: Simulated Annealing (SA)

Input: Patrón de enfriamiento
$s = s_0$; /* Generación de solución inicial*/
$T = T_{max}$; /* Temperatura de inicio*/
while No Criterio_término() /*ej: $T < T_{min}$*/ **do**
  while No Condición_equilibrio() /*A temperatura constante*/ **do**
    Genera un vecino aleatorio;
    Δ E = $f(s') - f(s)$;
    if $\Delta E \leq 0$ **then**
      $s = s'$; /*Acepta solución vecina*/
    else
      Acepta s' con probabilidad $e^{\frac{-\Delta E}{T}}$;
    end if
  end while
  T = actualizar_temperatura(T); /*Según patrón*/
end while
Return: Mejor solución encontrada

Tabú Search

La búsqueda tabú fue propuesta en el año 1986, como una forma determinista de implementar el control de la aleatoriedad del SA para escapar de óptimos locales (Talbi, 2009). Tabú Search (TS) se comporta como un algoritmo de búsqueda local, pero acepta soluciones de peor calidad para escapar de los óptimos locales cuando todos los vecinos son soluciones que no mejoran la solución actual. Por lo general, todo el vecindario se explora de una manera determinista, mientras que en SA se selecciona un vecino probabilísticamente (Talbi, 2009). Para su funcionamiento, se deben considerar distintas componentes. En primer lugar, se define una memoria de corto plazo o lista tabú la que sirve para evitar volver a visitar soluciones ya visitadas anteriormente. Dado que almacenar soluciones completas y su posterior revisión se vuelve muy costoso, es que en esta lista se almacenan movimientos prohibidos. Luego, se define un criterio de aspiración, el que permite aceptar soluciones generadas a partir de movimientos tabú bajo ciertos criterios, dentro de los que podemos encontrar, por ejemplo, un movimiento tabú que mejora la calidad de la mejor solución encontrada hasta el momento (Talbi, 2009). Algunos mecanismos más avanzados para aplicar intensificación y diversificación de las soluciones son:

- Memoria de mediano plazo (Intensificación): Almacena las mejores soluciones encontradas durante la búsqueda. De esta forma, se permite explorar en mayor detalle los vecindarios de estas soluciones.
- Memoria de largo plazo (Diversificación): Almacena información de las soluciones visitadas de manera de se encuentren características de las mejores soluciones para explorar zonas inexplorados del espacio de soluciones.

Variable Neighborhood Search

La búsqueda en vecindarios variable (VNS) es un algoritmo metaheurístico estocástico que explora el espacio de soluciones utilizando el concepto de estructuras generadores de vecindad (Talbi, 2009). Estas estructuras se encargan de generar un conjunto de soluciones vecinas a partir de una solución dada. Primero, se definen k estructuras de vecindad N_k ($k = 1, ..., n$). Luego, cada iteración del algoritmo se compone de 3 etapas: elección, búsqueda local y movimiento. En cada iteración, una solución inicial s' es elegida desde el vecindario actual (N_k). Luego, se aplica una búsqueda local sobre la solución s' para generar una solución s''. Sí y

Algoritmo 3.2: Tabu Search (TS)

1: $s = s_0$; /* Generación de solución inicial*/
2: Inicializar la lista tabú y las memorias de mediano y largo plazo;
3: **while** No Criterio_término() **do**
4: Encontrar el mejor vecino admisible s'; /*no tabú o aceptado por el criterio de aspiración*/
5: $s = s'$
6: Actualizar lista tabú, criterios de aspiración y memorias de mediano y largo plazo
7: **if** criterio_intensificación se cumple **then**
8: Intensificar;
9: **end if**
10: **if** criterio_diversificación se cumple **then**
11: Diversificar;
12: **end if**
13: **end while**
14: **Return:** Mejor solución encontrada

solo sí la solución s'' es mejor que la mejor solución actual, se actualiza la mejor solución por s'' y se vuelve a aplicar el procedimiento desde el vecindario N_1. En el momento en que no se encuentre ninguna solución que la actual, el algoritmo se cambia al vecindario N_{k+1}, aleatoriamente genera una nueva solución en este vecindario e intenta mejorarla (Talbi, 2009).

Greedy Randomized Adaptative Search Procedure

El procedimiento de búsqueda adaptativa golosa aleatoria (GRASP) es una metaheurística iterativa para resolver problemas de optimización combinatoria , introducido en el año 1989 (Talbi, 2009). Cada iteración se compone de dos pasos: construcción y búsqueda local. En la etapa de construcción, se genera una solución factible usando un algoritmo goloso aleatorio (por ejemplo, de un conjunto de n soluciones, obtener un subconjunto aleatorio y elegir la mejor). Luego, en la siguiente etapa se aplica una búsqueda local a la solución construida. Este proceso se repite hasta que se cumpla un criterio de detención (por ejemplo, un número dado de iteraciones).

Algoritmo 3.3: Variable Neighborhood Search (VNS)

Input Conjunto de estructuras generadores de vecindad $N_k fork = 1, ..., k_{max}$;
$s = s_0$; /* Generación de solución inicial*/
while No Criterio_término() **do**
  k=1;
  while $k \neq k_{max}$ **do**
    Perturbar: Elegir una solución aleatoria s' del k-ésimo vecindario $N_k(s)$ de s;
    s" = búsqueda_local(s');
    if $f(s'') < f(s)$ **then**
      s = s";
      Continuar la búsqueda con N_1; k = 1;
    else
      k = k +1;
    end if
  end while
end while
Return: Mejor solución encontrada

Algoritmo 3.4: Greedy Randomized Adaptive Search Procedure (GRASP)

Input Número de iteraciones;
$s = s_0$; /* Generación de solución inicial*/
while No Criterio_término() /*ej: número de iteraciones*/ **do**
  s = $Random - Greedy$; /*aplicar una búsqueda heurística golosa aleatoria*/
  s' = búsqueda_local(s);
end while
Return: Mejor solución encontrada

Iterated local Search

La búsqueda local iterada (ILS) es un algoritmo metaheurístico que permite mejorar la calidad de las soluciones obtenidas a partir de realizar múltiples búsquedas locales (Talbi, 2009). En la primera etapa, ILS aplica una búsqueda local a una solución inicial dada, generando una nueva solución s_*. Luego, iterativamente perturba la solución s_* para generar una solución s'; aplica una búsqueda local a s' para obtener s'_* y según un criterio de aceptación, puede o no actualizar s_*. Este proceso se repite hasta que se cumpla algún criterio de detención.

Algoritmo 3.5: Iterated Local Search (ILS)

1: s_0 = Generar solución inicial;
2: s_* = búsqueda_local(s_0);
3: **while** No Criterio_término() **do**
4: s' = Perturbar(s_*,historial de búsqueda); /*Perturba óptimo local*/
5: s'_* = Búsqueda_local(s');
6: s_* Aceptar(s_*,s'_*,memoria de búsqueda); /*Criterio de aceptación*/
7: **end while**
8: **Return:** Mejor solución encontrada

3.1.4 Terminales

Corresponden a las hojas del árbol. Son componentes pequeñas de las metaheurísticas base que se utilizan para realizar el proceso de generación de nuevos algoritmos. A partir de las metaheurísticas elegidas, se realiza una descomposición y se definen diversos terminales que pueden ser agrupados por sus características. Estos terminales se utilizan en el proceso evolutivo para formar los algoritmos y corresponden a las hojas de los árboles.

a) Terminales de Aceptación

Los terminales de aceptación evalúan la solución propuesta y, si cumple con un criterio, intercambian la solución actual con la solución propuesta. Devuelven verdadero si la solución propuesta es aceptada y falso en caso contrario.

1. NA-Deterministic_ChooseIfBetter: Terminal de aceptación determinista. Se acepta la solución propuesta solo si el costo es igual o mejor que la solución actual.

2. NA-Boltzman_NoCooling: Terminal de aceptación probabilístico. utiliza la distribución de probabilidad de Boltzman al igual que usa SA. La probabilidad se usa para aceptar candidatos que no mejoren la solución actual. Mientras más alta la temperatura, mayor probabilidad de aceptación. No maneja enfriamiento interno.

3. NA-Boltzman_NonMonotonous: Terminal de aceptación probabilístico. Funciona igual que $NA\_Boltzman\_NoCooling$ para aceptar las soluciones que no mejoran la calidad y manejar la temperatura, con la diferencia de que aumenta la temperatura en caso que no se acepte la solución propuesta. Maneja enfriamiento interno.

b) Terminales de Enfriamiento

Su finalidad es cambiar el valor del parámetro de temperatura una vez se ha ejecutado un terminal de aceptación una cantidad de veces determinada. Retorna verdadero si al temperatura cambia y falso en caso contrario.

1. CSch_LinearCooling: Calcula la temperatura en base a la ecuación 3.2 para la iteración i donde T_0 es la temperatura inicial y β una constante.

2. CSch_LogCooling: Calcula temperatura en base a la ecuación 3.3 para la iteración i, donde T_0 es la temperatura inicial.

3. CSch_GeomCooling: Calcula la temperatura en base a la ecuación 3.4, donde T es la temperatura actual y α es un coeficiente para decrementar la temperatura. $\alpha \in]0, 1[$

$$T_i = T_0 - i\beta \tag{3.2}$$

$$T_i = \frac{T_0}{logi} \tag{3.3}$$

$$T = \alpha T \tag{3.4}$$

c) Terminales de Perturbación

Estos terminales tienen por finalidad alterar la solución propuesta y retornan verdadero siempre.

1. NR1_ShiftCity: Elige un nodo aleatorio y lo inserta en una nueva posición aleatoria en la solución propuesta.
2. NR2_SwapCities: Se eligen dos nodos aleatorios de la solución propuesta y se invierten de posición.
3. NRn_BlockReverse: Se elige un bloque aleatorio de la solución propuesta y se invierten los nodos.
4. NR_Movimiento2-OPT: Se ejecuta el algoritmo conocido como 2-opt en la solución propuesta (Anexo B.2).
5. SM: Elige un bloque de tamaño aleatorio de la solución propuesta y mezcla los nodos del bloque.

d) Terminales de generación de vecindad

Estos terminales generan una vecindad de soluciones a partir de una solución propuesta y se elige el mejor individuo de la vecindad. La solución propuesta se actualiza si el mejor vecino mejora la calidad de la solución. Retorna verdadero si la solución se mejora y falso en caso contrario.

1. NG_BestNeighboor: Genera tantos vecinos como el tamaño de la solución utilizando $swap$ de dos ciudades. Se elige un tercio de las soluciones generadas aleatoriamente y se elige el mejor de ellos.
2. NG_BestRandomNeighboor: Genera tantos vecinos como el tamaño de la solución utilizando $swap$ de dos ciudades. Elige el mejor de los vecinos.

e) Terminales de Búsqueda Local

Los terminales de búsqueda local intentan mejorar la solución propuesta modificándola según un criterio determinado. Al finalizar todas las iteraciones y guarda la solución generada en el contenedor de solución propuesta. Devuelve verdadero si se produce una mejora en la solución y falso en caso contrario. Inicialmente se copia la solución propuesta a la solución auxiliar y se modifica la segunda.

1. LS_BlockRule: Método de perturbación que iterativamente invierte bloques de la solución auxiliar.
2. LS_SwapCitiesRule: Método de perturbación que iterativamente invierte las posiciones de dos nodos elegidos aleatoriamente.

f) Terminales Híbridos

Son terminales que combinan algunos de los terminales propuestos anteriormente. Se utilizan terminales de Aceptación con terminales de Perturbación.

1. NR&Ac_ShiftCityAndBoltzman: Concatena los terminales (c.1) y (a.2)
2. NR&Ac_ShiftCityAndBoltzmanNoMono: Concatena los terminales (c.1) y (a.3)
3. NR&Ac_ShiftCityAndDetermAccept: Concatena los terminales (c.1) y (a.1)
4. NR&Ac_Swap2CitiesAndBoltzman: Concatena los terminales (c.2) y (a.2)
5. NR&Ac_Swap2CitiesAndBoltzmanNoMono: Concatena los terminales (c.2) y (a.3)
6. NR&Ac_Swap2CitiesAndDetermAccept: Concatena los terminales (c.2) y (a.1)
7. NR&Ac_BlockReverseAndBoltzman: Concatena los terminales (c.3) y (a.2)
8. NR&Ac_BlockReverseAndBoltzmanNoMono: Concatena los terminales (c.3) y (a.3)
9. NR&Ac_BlockReverseAndDetermAccept: Concatena los terminales (c.3) y (a.1).

g) Terminales de ejecución probabilística

Corresponden a terminales que condicionan la ejecución de otros nodos del árbol según una probabilidad interna decida en cada terminal.

1. Prob10: Terminal que retorna verdadero un 10% de sus ejecuciones y falso en el resto.
2. Prob30: Terminal que retorna verdadero un 30% de sus ejecuciones y falso en el resto.
3. Prob50: Terminal que retorna verdadero un 50% de sus ejecuciones y falso en el resto.
4. Prob70: Terminal que retorna verdadero un 70% de sus ejecuciones y falso en el resto.
5. Prob90: Terminal que retorna verdadero un 90% de sus ejecuciones y falso en el resto.

3.1.5 Metaparámetros de la evolución

El proceso evolutivo requiere determinar diversos metaparámetros. La probabilidad de mutación, cruzamiento y selección se obtienen de trabajos existentes en la literatura y que son ampliamente utilizados (Loyola et al. (2016); Acevedo et al. (2020)). Finalmente, se define un valor de error mínimo al evaluar las instancias en los algoritmos. Si el error obtenido es menor que el umbral definido, se considera como un hit. Las probabilidades y parámetros se presentan en la tabla 3.1.

Metaparámetro	Valor
Cruzamiento	85%
Reproducción	10%
Mutación	5%
Individuos	500
Generaciones	100
Profundidad de los árboles	5
Error mínimo	0,0001

Tabla 3.1: Metaparámetros del proceso evolutivo. Fuente: Elaboración propia, (2022).

3.1.6 Conjunto de ejemplos

Los ejemplos seleccionadas para evolucionar se presentan originalmente en Taş et al. (2016). Corresponde a un conjunto de 22 ejemplos con distintas cantidades de nodos (entre 14 y

45 nodos), pertenecientes a ejemplos clásicos del TSP tradicional, obtenidos de la librería TSPLIB. Para el TSP-TS es necesario definir una función de tiempo que indique el costo de atender un nodo dado un tiempo transcurrido dado. En Taş et al. (2016) se proponen cuatro funciones de tiempo distintas, de las que se propone utilizar la función de tiempo pequeño para realizar la evolución de los algoritmos y su correspondiente evaluación.

1. Tiempo de servicio pequeño: $s_i = 5(10^{-3})b_i + 3(10^{-2})$
2. Tiempo de servicio medio: $s_i = 10^{-2}b_i + 6(10^{-2})$
3. Tiempo de servicio grande: $s_i = 2(10^{-2})b_i + 1.2(10^{-1})$
4. Tiempo de servicio cuadrático: $s_i = 4(10^{-5}) - 4(10^{-3})b_i + 3(10^{-1})$

Finalmente, los tiempos de viaje originales se ajustan de manera que cada ejemplo tenga el mismo costo promedio en los arcos para todos los ejemplos (Taş et al., 2016).

CAPÍTULO 4. RESULTADOS Y ANÁLISIS

En este capítulo se presentan los resultados obtenidos del proceso evolutivo. Se analiza el comportamiento de las poblaciones a lo largo de distintos experimentos evolutivos, los que tienen como producto los mejores algoritmos generados. Luego, se evalúa el desempeño de los algoritmos obtenidos frente a diversas instancias y se eligen los mejores de ellos para ser analizada su estructura. Finalmente, se comparan con el mejor algoritmo metaheurístico de la literatura y se realizan pruebas estadísticas para verificar si hay evidencia suficiente para determinar que los algoritmos propuestos presentan una mejora o no para resolver el problema.

4.1 CLUSTERING DE EJEMPLOS

Para seleccionar los grupos de ejemplos para la evolución y la evaluación, se propone utilizar un método de agrupamiento llamado $K - Means$. Este algoritmo tiene como objetivo la partición de un conjunto de n datos en k grupos, en el que cada uno de los n datos se asocia a un grupo según su distancia a los promedios. Se parametriza de manera tal que que entregue 2 grupos de instancias, con la finalidad de que un grupo sirva como conjunto de evolución y otro como conjunto de evaluación. Para generar los grupos, se deben determinar distintas métricas de las instancias (características). Las características elegidas son: cantidad de nodos de cada ejemplo, el costo promedio entre nodos, costo mayor entre los nodos y costo menor entre nodos (Smith-Miles & Lopes, 2012), resultando la siguiente distribución:

- Grupo 1
  - gr24
  - bayg29
  - bays29
  - eil30
  - eil35
  - eil40
  - eil45
- Grupo 2
  - burma14
  - ulysses16

- gr17
- gr21
- ulysses22
- fri26
- dantzig30
- att30
- gr30
- swiss30
- hk30
- gr35
- swiss35
- swiss42
- dantzig42

4.2 RESULTADOS DEL PROCESO EVOLUTIVO

4.2.1 Convergencia de la GAA para el TSP-TS

La GAA aplicada al TSP-TS converge consistentemente para las distintas ejecuciones y para ambos conjuntos de datos. En la figura 4.1, se muestra el comportamiento de la evolucion de ambos procesos evolutivos. Cada curva representa al $fitness$ promedio de todos los algoritmos de la correspondiente generación, al evaluar sobre los ejemplos tanto del grupo 1 como del grupo 2. Para el grupo 1 (4.1), es posible apreciar que a partir de la generación 23 aproximadamente, la calidad de los algoritmos generados comienza a estabilizarse. Para el grupo 2, se aprecia un comportamiento similar a partir de la generación 25 aproximadamente. También, es posible observar que en ambos gráficos se presenta una convergencia acelerada en las primeras generaciones, lo que indica que los algoritmos rápidamente mejoran respecto a la calidad de las soluciones encontradas. Debido a la convergencia consistente, se posee una primera evidencia que los algoritmos generados resuelven cada vez mejor el problema.

Los mejores algoritmos de cada evolución se obtienen en etapas tempranas del proceso. En la figura 4.2, se visualiza un gráfico para la evolución con el grupo 1 y para el grupo 2, en que cada curva representa el $fitness$ del mejor algoritmo encontrado para cada proceso

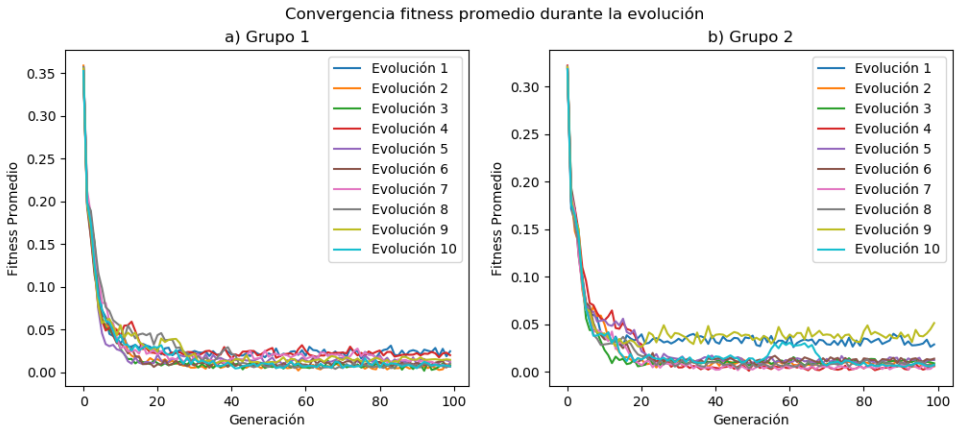

Figura 4.1: Curva convergencia procesos evolutivos. Fuente: Elaboración propia, (2022)

evolutivo. Para el grupo 1 (4.2.a), se observa que en la generación 0 existe un algoritmo (Mejor algoritmo evolución 7) que posee un $fitness$ inicial peor respecto a los otros algoritmos. Si se sigue su proceso evolutivo es posible visualizar que en las generaciones subsiguientes demora más en alcanzar valores de $fitness$ pequeños, comparado a la convergencia de las otras evoluciones. Lo anterior se podría mejorar aumento la capacidad de diversificación e intensificación de búsqueda de soluciones. Luego, es posible encontrar un pequeño grupo de 4 que comienzan con un $fitness$ similar (cercanos a 0.045). Después, se encuentra otro grupo de 3 algoritmos que comienzan con un $fitness$ que bordea el 0.02. Finalmente, se distinguen 2 algoritmos más que comienzan con $fitness$ 0.019 y 0.004 aproximadamente. Sobre el grupo 2, (4.2.b), existe un algoritmo que comienza con un $fitness$ claramente mayor que en los otros procesos evolutivos (Mejor algoritmo evolución 4) el que corresponde a 0.1. Luego, se observa un grupo de aproximadamente 7 algoritmos que comienzan con un $fitness$ cercano a 0.04. Finalmente, un último grupo de aproximadamente 2 algoritmos comienzan con un $fitness$ carcano a 0.01. Si se miran ambos gráficos (4.2), se observa que a partir de la generación 20 aproximadamente se produce una estabilización de la mejora de los mejores algoritmos generados. Sin embargo, se obtienen errores del orden del 0.1%, lo que indica que los algoritmos están obteniendo resultados muy cercanos al óptimo real de los ejemplos de evolución. Finalmente, el mejor algoritmo tanto del grupo 1 como del grupo 2(4.2) alcanzan un $fitness$ muy cercano a 0, lo que indica que está resolviendo los ejemplos en su totalidad, o muy cercano al total. Además, este comportamiento no se ve solo en el mejor de los mejores, sino que es una tendencia general de los mejores algoritmos de cada grupo.

El $fitness$ de los algoritmos generados converge para toda la población para cada generación del proceso evolutivo. En la figura 4.3 se muestran 4 evoluciones distintas con el grupo 1 (4.3.a,4.3.b,4.3.c,4.3.d), que describen la distribución de los $fitness$ de los algoritmos durante el proceso evolutivo, representado por el área sombreada, mientras que la línea azul

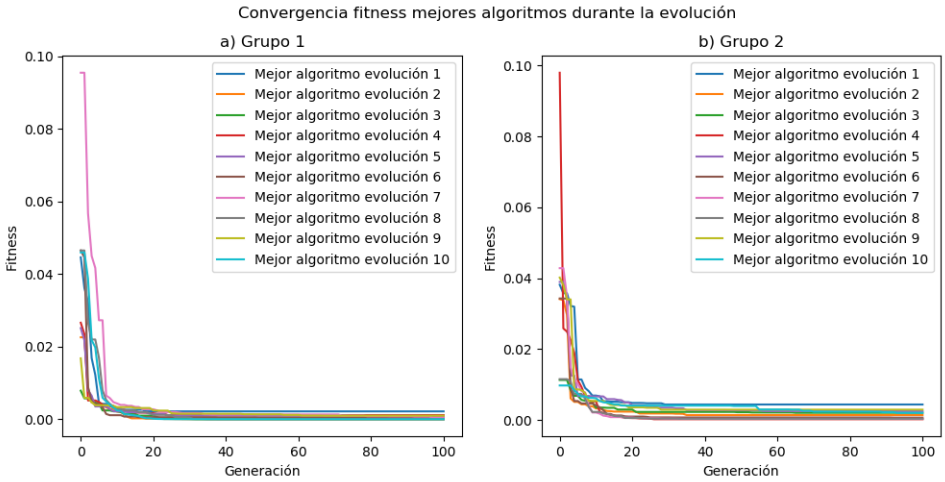

Figura 4.2: Curva convergencia mejores algoritmos. Fuente: Elaboración propia, (2022)

representa el $fitness$ promedio de cada generación. Es posible apreciar que el $fitness$ converge de manera permanente, como ya se menciona anteriormente, y se puede observar también que la población entera converge durante el proceso evolutivo. Ocurre un fenómeno similar en los 4 gráficos restantes (4.3.e, 4.3.f, 4.3.g, 4.3.h), que muestra evoluciones distintas pero con el grupo 2, en las cuales toda la población de algoritmos converge en conjunto.

Los algoritmos son muy similares inicialmente. La distribución de los $fitness$ al comienzo del proceso evolutivo es muy pequeña. Esto se puede observar debido a que al comienzo el área sombreada es casi imperceptible para todos los casos mostrados. Esto puede ser provocado por falta de diversificación en la generación de soluciones iniciales, pudiendo generar convergencia hacia un óptimo local. Lo anterior se puede mejorar aumentando el porcentaje de cruzamiento de la población, aumentando la mutación, o disminuyendo el elitismo del proceso (en otras palabras, disminuir el porcentaje de reproducción).

4.2.2 Evaluación de los algoritmos

Los algoritmos generados resuelven el problema para diversos ejemplos del TSP-TS. En cada generación, se evalúan todos los algortmos generados. Luego, el mejor de la generación se elige en base a la calidad con que resuelven los diversos ejemplos (menor $fitness$). El mejor de cada proceso evolutivo corresponde al algoritmo haya tenido mejor desempeño en todo el proceso. Para evaluar y comparar los mejores algoritmos de cada proceso evolutivo, se organizan en la tabla 4.1, en que las primeras 10 filas corresponden a los 10 algoritmos generados a partir de evolucionar con el grupo 1 de ejemplos, y los 10 siguientes generados en la evolución con el grupo

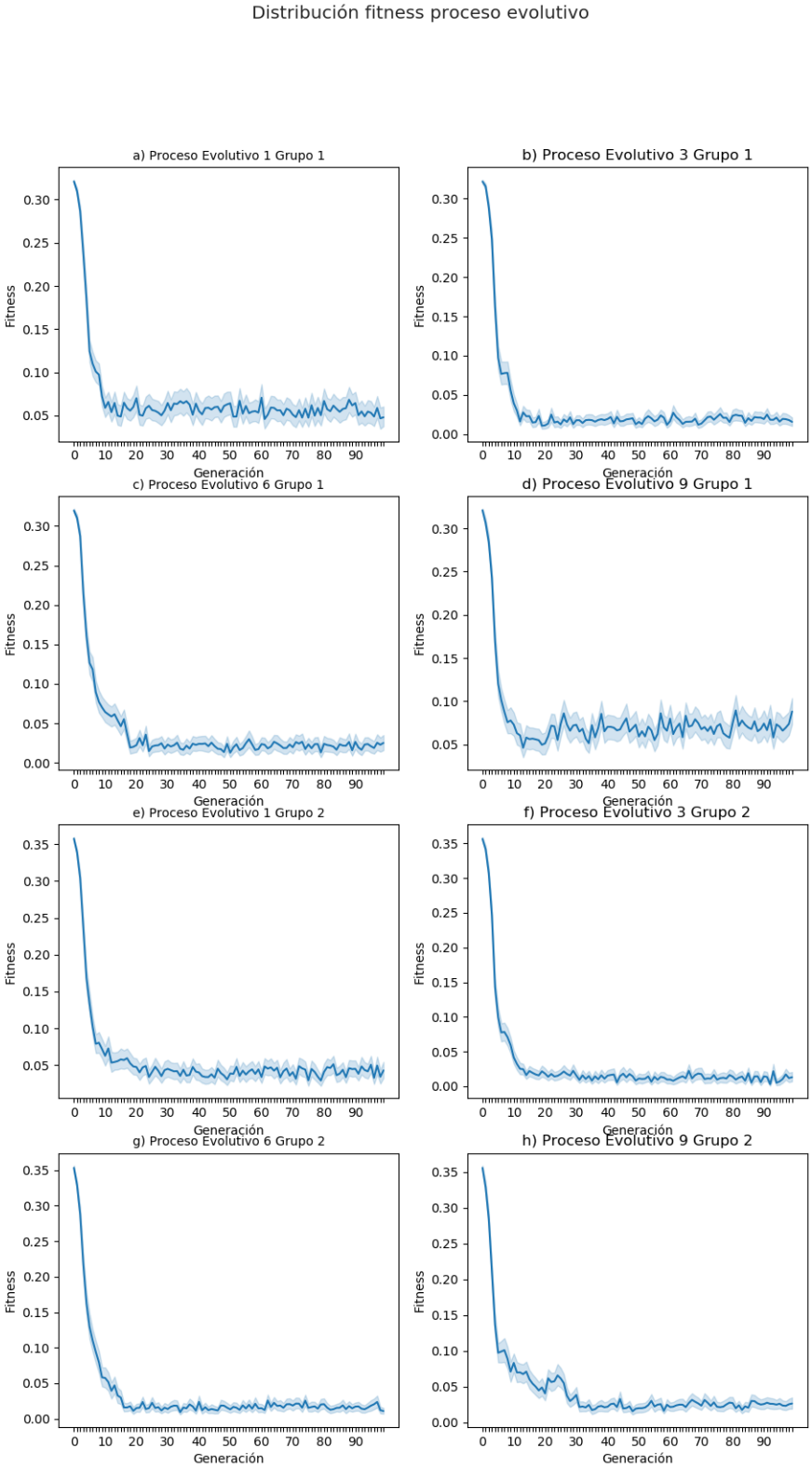

Figura 4.3: Curva distribución $fitness$ algoritmos evolución. Fuente: Elaboración propia, (2022)

2. Luego, se escribe el $fitness$ promedio de cada algoritmo, los $hits$ promedio y el tiempo promedio en resolver cada ejemplo. El promedio de $fitness$ se obtiene al evaluar un algoritmo contra todas los ejemplos, para luego sumar estos $fitness$ y dividir por la cantidad de ejemplos. Para el tiempo y los $hits$ se aplica el mismo método. Se recuerda que el valor de $fitness$ corresponde al ERP y los $hits$ corresponde a la cantidad de veces promedio que acierta el algoritmo en encontrar un óptimo. Para identificar los algoritmos se hace necesario definir una nomenclatura, en la cual se utiliza una combinación de subíndices y superíndices. Los superíndices indican el grupo de ejemplos con el que se evolucionaron y los subíndices indican el proceso evolutivo del que se obtienen. Por ejemplo, A_5^2 se obtuvo evolucionando con el grupo 2 de ejemplos, en el proceso evolutivo 5.

Al evaluar los algoritmos del grupo 1 con los ejemplos del grupo 1, es posible apreciar que los algoritmos resuelven en la mayoría de los casos todas los ejemplos las que se enfrentan, obteniendo $fitness$ muy bajos, cercanos o iguales a 0 y con $hits$ cercanos a 1 (4.1). Por ejemplo, A_7^1 obtiene un $fitness$ de 0.0013 y un valor de $hits$ igual a 0.93, lo que quiere decir que la diferencia entre los óptimos reales y encontrados por el algoritmo es muy pequeña ($fitness$ bajo), hasta el punto que resuelve los ejemplos un 93% de las veces. Además, el tiempo computacional en resolverlos bordean el orden de los 0.01 segundos en promedio. Luego, al evaluar los algoritmos del grupo frente a los ejemplos que no se usaron para su evolución (grupo 2), también se obtienen valores de $fitness$ cercanos a 0 y $hits$ cercanos a 1, indicando que resuelven también este conjunto de ejemplos, en tiempos de cómputo muy pequeños.

La mayoría de los algoritmos tienen un $fitness$ muy pequeño. Existen 5 algoritmos (A_2^1, A_3^1, A_5^1, A_8^1, A_{10}^1) evolucionados con el grupo 1 que obtienen un $fitness$ de 0 al evaluarlos con el mismo grupo. Esto indica que los algoritmos se están especializados en ese conjunto de ejemplos. Luego, esos mismos 5 algoritmos tienen un $fitness$ muy cercano a 0 al evaluarlos sobre el grupo 2 de ejemplos, indicando que también son capaces de resolver problemas que no han visto en la evolución. Luego, al mirar los algoritmos evolucionados con el grupo 2, se identifican 3 algoritmos (A_4^2, A_7^2, A_8^2) que tienen un $fitness$ igual a 0 al evaluarlos con el grupo 1, indicando que son capaces de encontrar los óptimos a ejemplos que no se usaron para evolucionar, pero su desempeño en cuanto a $fitness$ sobre los ejemplos del grupo 2 no es 0. Sin embargo, los valores son tan pequeños (0.0001, 0.0004 y 0.0008), que la diferencia llega a ser despreciable (referenciar error en capítulo de metodología). Después, si se miran los promedio de $fitness$ de evaluar los algoritmos evolucionados con el grupo 1, se verifica que se obtiene un error menor en el grupo 1 que en el grupo 2 (0,0018 < 0,0032), lo que es esperable, dado que el grupo 1 es el utilizado para evolucionar esos algoritmos. Si se analiza el $fitness$ promedio de evaluar los algoritmos generados con el grupo 2, ocurre un fenómeno distinto. Se obtiene un error promedio mayor al evaluar con los mismas ejemplos que se usaron para evolucionar. Esto se debe a que

existe un algoritmo (A_1^2) que tiene un error de evaluación muy superior respecto a los otros ($fitness$ de 0.0182), lo que es un caso aislado. Finalmente, resalta también que solo se presentan valores de $fitness$ igual a 0 en el grupo1 y no el grupo 2, lo que podría indicar que los ejemplos que se encuentran en el grupo 1 son problemas menos complejos.

Los algoritmos resuelven ambos grupos de ejemplos. Si se miran la cantidad de $hits$ de los algoritmos evolucionados con el grupo 1, destaca A_9^1 con 0,8 al evaluarlo sobre el grupo 1. Esto quiere decir que ese algoritmo resuelve los ejemplos en un 80% de las veces, lo que entrega un desempeño mucho peor respecto a algoritmos como A_2^1, A_3^1, A_8^1 o A_{10}^1, que poseen $hits$ igual a 1, es decir, resuelven todas los ejemplos del grupo 1. Más aún, A_2^1 o A_3^1 también resuelven en la totalidad de las ejecuciones los ejemplos del grupo 2, las que no se utilizaron para el proceso evolutivo que los genera, indicando que estos algoritmos pueden resolver problemas para los que no fueron generados. Al mirar los $hits$ de los algoritmos del grupo 2, destacan 3 principalmente (A_4^2, A_7^2 y A_8^2) que son capaces de resolver en todas las ejecuciones ambos conjuntos de ejemplos. Luego, al observar el valor de $hits$ promedios de los algoritmos generados con el grupo 1, se obtiene la resolución de los ejemplos en promedio una mayor cantidad de veces al evaluarlos con el grupo 1 que con el grupo 2. Finalmente, al mirar los $hits$ promedio de los algoritmos del grupo 2, se obtiene un resultado inverso. Es decir, resuelven una mayor cantidad de veces los ejemplos del grupo 1 que las del grupo 2.

Los algoritmos terminan su ejecución en tiempos acotados. Al mirar la cantidad de segundos de ejecución de los algoritmos es posible ver que ninguno supera 1 segundo de ejecución, lo que es deseable al momento de resolver problemas tan complejos computacionalmente. Luego, no se encuentra una diferencia significativa al evaluar el mismo algoritmo frente a ambos conjuntos de ejemplos. En otras palabras, si se ven los tiempos en orden de magnitud de A_7^1, demoran alrededor de 0,002 segundos en ejecutarse para ambos grupos, comportamiento que se repite en todos los algoritmos, excepto A_9^1. Además, se puede notar que A_9^1 demora en promedio mucho menos en resolver el grupo 2 que el grupo 1 y los resuelve con una calidad inferior respecto a otros algoritmos que demoran tiempos similares (por ejemplo, A_1^5). Después, se encuentra A_1^2 que demora 0.0032 y 0.0028 segundos en promedio en ejecutarse para intentar resolver los grupos 1 y 2 respectivamente, pero no es capaz de resolver en todas las ejecuciones. Finalmente, se encuentran A_2^1 y A_4^2 que son los algoritmos que más demoran en ejecutarse, pero resuelven los ejemplos en todas sus ejecuciones, con valores de $fitness$ muy cercanos a 0.

Finalmente, es posible generar algoritmos para el TSP-TS y son capaces de resolver distintos ejemplos del problema. Se identifican diversos algoritmos (A_2^1,A_3^1,A_8^1,A_{10}^1,A_4^2,A_7^2 y A_8^2) que son capaces de resolver tanto ejemplos que se usaron para generarlos como otras que no vieron en su generación.

Tabla 4.1: Evaluación algoritmos generados frente a grupos de ejemplos con tiempos pequeños (función 1 de la sección 3.1.6)

	Grupo 1			Grupo 2		
	$Fitness$	$Hits$	Tiempo (seg)	$Fitness$	$Hits$	Tiempo (seg)
A_1^1	0,0012	0,98	0,0289	0,0039	0,92	0,0333
A_2^1	**0,0000**	**1,00**	0,0390	0,0001	**1,00**	0,0568
A_3^1	**0,0000**	**1,00**	0,0268	0,0001	**1,00**	0,0329
A_4^1	0,0002	**1,00**	0,0085	0,0020	0,96	0,0085
A_5^1	**0,0000**	**1,00**	0,0077	0,0012	0,99	0,0060
A_6^1	0,0004	0,98	0,0159	0,0011	0,99	0,0168
A_7^1	0,0013	0,93	0,0027	0,0035	**1,00**	0,0022
A_8^1	**0,0000**	**1,00**	0,0313	0,0002	**1,00**	0,0351
A_9^1	0,0044	0,80	0,0023	0,0040	0,86	0,0006
A_{10}^1	**0,0000**	**1,00**	0,0234	0,0003	**1,00**	0,0262
Avg	0,0018	0,96	0,0180	0,0032	0,91	0,0204
A_1^2	0,0012	0,95	0,0032	0,0182	0,80	0,0028
A_2^2	0,0006	0,98	0,0196	0,0005	**1,00**	0,0193
A_3^2	0,0008	0,97	0,0122	0,0013	0,99	0,0126
A_4^2	**0,0000**	**1,00**	0,0367	0,0001	**1,00**	0,0429
A_5^2	0,0004	0,99	0,0068	0,0014	0,99	0,0059
A_6^2	0,0001	**1,00**	0,0113	0,0008	0,99	0,0117
A_7^2	**0,0000**	**1,00**	0,0262	0,0004	**1,00**	0,0291
A_8^2	**0,0000**	**1,00**	0,0282	0,0004	**1,00**	0,0322
A_9^2	0,0008	0,97	0,0033	0,0013	0,91	0,0031
A_{10}^2	0,0005	0,97	0,0052	0,0008	0,99	0,0033
Avg	0,0009	0,96	0,0165	0,0057	0,95	0,0181

Fuente: Elaboración Propia, (2022)

4.2.3 Desempeño de los algoritmos en otros grupos de ejemplos

Los algoritmos generados resuelven problemas distintos. Esto quiere decir que, si modificamos la función objetivo usando polinomios de tiempo de servicio distintos, los algoritmos generados también los resuelven. En esta tesis, se trabaja con un conjunto de ejemplos particular y con una función de tiempo particular para el TSP-TS. Sin embargo, cuando se enfrentan los algoritmos a ejemplos con funciones de tiempo diferentes, también logran resolverlas. Las tablas 4.2 y 4.3 tienen la misma estructura que la tabla 4.2 se muestran los resultados obtenidos a partir de evaluar los algoritmos frente a ejemplos que usan distintos polinomios para calcular la función objetivo del problema (funciones 2 y 3 expuestas en la sección 3.1.6 obtenidas de Cacchiani et al. (2020)). Es decir, los algoritmos generados a partir de un conjunto de ejemplos en particular también son capaces de resolver problemas nuevos.

Algunos de los algoritmos generados obtienen errores cercanos a 0 en promedio para ambos grupos. Existen algoritmos que obtienen en promedio valores de $fitness$ iguales o cercanos a 0 para todos los ejemplos. Por ejemplo A_2^1, A_3^1 y A_{10}^1 tienen valores de $fitness$ cercanos a 0 al evaluarlos tanto el grupo 1 como el grupo 2 para las funciones de tiempo medio y grande. Respecto a los algoritmos generados con el grupo 2 están A_4^2 y A_7^2 que tienen valores de $fitness$ muy cercanos a 0. Por otro lado, existen algoritmos que tienen un despeño peor respecto a los otros algoritmos. Por ejemplo, A_4^1 con $fitness$ igual a 0,0143 con función de tiempo medio en el grupo 2 y 0,0103 con función de tiempo grande en el grupo 1. Finalmente, se obtienen en promedio valores de $fitness$ del orden de 0,0005 al evaluar los algoritmos con el grupo 1 para ambas funciones de tiempo, mientras que al evaluar los algoritmos en el grupo 2 se obtienen errores del orden de 0,0014.Esto aporta evidencia a que las instancias del grupo 2 son más complejas de resolver.

Existen un gran número de algoritmos que resuelven todas las instancias. De los 20 algoritmos generados, se identifican 7 algoritmos (A_2^1,A_3^1,A_8^1,A_{10}^1,A_4^2,A_7^2,A_8^2) que obtienen valores de $hits$ igual 1al evaluarlos en el grupo 1 y el grupo 2 de ejemplos tanto para la función de tiempo medio como la grande, lo que indica que tienen una gran capacidad de generalización. por otro lado, A_9^1 obtiene un valor de $hits$ de 1 o cercano al evaluar sobre el grupo 1 tanto para la función de tiempo medio como grande, pero no logra resolver las instancias del grupo 2 para las mismas funciones, por lo que no es capaz de generalizar tanto como otros algoritmos.

Los algoritmos resuelven ejemplos diferentes en tiempos acotados. Los algoritmos en promedio se ejecutan en un tiempo del oren de los 0,01 segundos, tanto para el grupo 1 como el grupo 2 de ejemplos. Existen algoritmos (por ejemplo A_5^1, A_9^2 y A_{10}^2) cuya ejecución es del orden de los 0,002 segundos y logran resolver casi en todas las ejecuciones los ejemplos del grupo 1 y el grupo 2 para la función de tiempo medio y grande. También está A_7^1 que puede

resolver correctamente los ejemplos del grupo 1 para la función de tiempo medio, pero no supera el 87% al resolver el grupo 2 para el tiempo medio y ninguno de los dos grupos para la función de tiempo grande, a pesar de tener un tiempo computacional menor que el promedio. Existen otros algoritmos (A_3^1, A_8^1 y A_4^2) que demoran más que el promedio en ejecutarse, pero logran resolver ambos grupos de instancias para las funciones de tiempo medio y grande. Después, se puede estudiar A_9^1, que logra resolver sobre un 90% de las veces los ejemplos del grupo 1 para las funciones de tiempo medio y grande, pero no supera el 57% para las instancias del grupo 2 con ninguna de las 2 funciones de tiempo. Finalmente, se ve que los algoritmos solo obtienen valores de $fitness$ igual a 0 sobre las instancias del grupo 1 para ambas funciones de tiempo, mientras que para el grupo 2 nunca se encuentra un valor de $fitness$ igual a cero, indicando que el grupo 2 puede ser un conjunto de instancias más complejas.

La mayoría de los algoritmos que logran resolver correctamente las instancias con tiempos pequeños (tabla 4.1) también resuelven las instancias con tiempos medios y grandes (tabla 4.2 y tabla 4.3). Los algoritmos A_2^1, A_3^1, A_8^1, A_{10}^1, A_4^2 y A_7^2 tienen valores de $hits$ igual 1 para todos los grupos de instancias, con las 3 funciones de tiempo evaluadas, indicando que dichos algoritmos son capaces de resolver distintos ejemplos del TSP-TS encontrando el óptimo. Por otra parte, algoritmos como A_9^1 o A_1^2 tienen un desempeño más bajo en todas las evaluaciones respecto a los algoritmos de otros procesos evolutivos.

No hay evidencia para asegurar que las instancias con funciones de tiempo más grandes presenten mayor dificultad para los algoritmos generados. En general, los algoritmos que resuelven las instancias con tiempos pequeños, resuelven también las instancias con tiempos medios o grandes. Por otro lado, los algoritmos que tienen un desempeño peor con las instancias con tiempo pequeño también tienen problemas al resolver instancias con tiempos medios o grandes. A pesar de que se hayan generados algoritmos que tengan desempeños peores que otros, si es posible generar algoritmos que resuelvan distintos ejemplos del problema aunque estos ejemplos no se hayan usado para generarlos. Es decir, se pueden generar algoritmos con capacidad de generalizar.

4.2.4 Estructura de mejores algoritmos generados

Existen algoritmos con mejor desempeño según la calidad con que resuelven los ejemplos. En el experimento mostrado en la tabla 4.1, se identifican algoritmos que resuelven los ejemplos con mayor exactitud que otros. Estos se pueden elegir minimizando el valor de $fitness$ promedio en que resuelven los ejemplos que no se usaron para su evolución. En este trabajo se eligen A_3^1, A_4^2 y A_7^2. Se presentan sus pseudocódigos en Anexo A y sus algoritmos como árboles

Tabla 4.2: Evaluación algoritmos generados frente a grupos de ejemplos con tiempos medianos (función 2 de la sección 3.1.6)

	Grupo 1			Grupo 2		
	$Fitness$	$Hits$	Tiempo (seg)	$Fitness$	$Hits$	Tiempo (seg)
A_1^1	0,0011	0,95	0,0156	0,0041	0,81	0,0184
A_2^1	0,0001	**1,00**	0,0359	0,0003	**1,00**	0,0420
A_3^1	**0,0000**	**1,00**	0,0295	0,0003	**1,00**	0,0328
A_4^1	0,0008	0,97	0,0098	0,0143	0,94	0,0085
A_5^1	0,0008	0,97	0,0060	0,0019	0,99	0,0057
A_6^1	0,0001	**1,00**	0,0156	0,0014	**1,00**	0,0164
A_7^1	0,0008	**1,00**	0,0019	0,0058	0,71	0,0021
A_8^1	**0,0000**	**1,00**	0,0322	0,0002	**1,00**	0,0367
A_9^1	0,0012	**1,00**	0,0013	0,0079	0,57	0,0005
A_{10}^1	**0,0000**	**1,00**	0,0245	0,0007	**1,00**	0,0269
Avg	0,0005	0,99	0,0172	0,0037	0,90	0,0190
A_1^2	0,0039	0,83	0,0031	0,0052	0,81	0,0037
A_2^2	0,0001	**1,00**	0,0175	0,0005	0,99	0,0209
A_3^2	0,0004	0,99	0,0118	0,0016	**1,00**	0,0122
A_4^2	**0,0000**	**1,00**	0,0449	0,0004	**1,00**	0,0534
A_5^2	0,0013	0,99	0,0066	0,0018	**1,00**	0,0056
A_6^2	0,0003	0,99	0,0122	0,0008	0,97	0,0118
A_7^2	**0,0000**	**1,00**	0,0294	0,0005	**1,00**	0,0341
A_8^2	0,0001	**1,00**	0,0311	0,0006	**1,00**	0,0334
A_9^2	0,0007	0,99	0,0032	0,0019	0,93	0,0032
A_{10}^2	0,0007	0,97	0,0049	0,0008	**1,00**	0,0041
Avg	0,0007	0,98	0,0165	0,0014	0,97	0,0182

Fuente: Elaboración Propia, (2022)

Tabla 4.3: Evaluación algoritmos generados frente a grupos de ejemplos con tiempos grandes (función 3 de la sección 3.1.6)

	Grupo 1			Grupo 2		
	$Fitness$	$Hits$	Tiempo (seg)	$Fitness$	$Hits$	Tiempo (seg)
A_1^1	0,0028	0,91	0,0201	0,0043	0,87	0,0235
A_2^1	**0,0000**	**1,00**	0,0382	0,0007	**1,00**	0,0452
A_3^1	**0,0000**	**1,00**	0,0291	0,0006	**1,00**	0,0361
A_4^1	0,0103	0,92	0,0088	0,0044	0,87	0,0075
A_5^1	0,0008	0,98	0,0071	0,0020	0,96	0,0061
A_6^1	0,0003	0,99	0,0166	0,0022	0,99	0,0164
A_7^1	0,0028	0,87	0,0020	0,0056	0,86	0,0020
A_8^1	**0,0000**	**1,00**	0,0323	0,0011	**1,00**	0,0382
A_9^1	0,0012	0,93	0,0013	0,0102	0,57	0,0006
A_{10}^1	0,0001	**1,00**	0,0248	0,0006	**1,00**	0,0285
Avg	0,0018	0,96	0,0180	0,0032	0,91	0,0204
A_1^2	0,0049	0,76	0,0033	0,0055	0,86	0,0035
A_2^2	0,0003	0,99	0,0204	0,0022	0,99	0,0208
A_3^2	0,0004	0,99	0,0121	0,0024	0,96	0,0135
A_4^2	**0,0000**	**1,00**	0,0420	0,0006	**1,00**	0,0505
A_5^2	0,0013	0,93	0,0070	0,0378	0,83	0,0054
A_6^2	0,0006	0,97	0,0118	0,0014	0,99	0,0120
A_7^2	**0,0000**	**1,00**	0,0306	0,0007	**1,00**	0,0346
A_8^2	0,0001	**1,00**	0,0293	0,0004	**1,00**	0,0324
A_9^2	0,0009	0,98	0,0034	0,0032	0,93	0,0035
A_{10}^2	0,0003	**1,00**	0,0049	0,0027	0,94	0,0045
Avg	0,0009	0,96	0,0165	0,0057	0,95	0,0181

Fuente: Elaboración Propia, (2022)

en Anexo C.

En primer lugar, A_3^1 resuelve con un $fitness$ de 0.0001 los ejemplos del Grupo 2 y con un $fitness$ de 0 sobre el Grupo 1, al evaluar sobre las instancias con tiempos pequeños, siendo uno de los mejores algoritmos creado por el proceso evolutivo. Además, resuelve en un 100% de las ejecuciones todas los 2 grupos de ejemplos. Al comprobar su comportamiento contra las instancias con tiempos medios, obtiene un error promedio de 0 para el grupo 1 y de 0,0003 para el grupo 2, con un 100% de $hits$. Para las instancias con tiempos grandes obtiene un error de 0 para el grupo 1 y 0,0006 para el grupo 2, también con 100% de hits. Dado lo anterior, este algoritmo posee una gran capacidad de generalización a la hora de resolver diversos ejemplos.

Al observar A_4^2 sobre las instancias con tiempos pequeños, se ve que tiene un $fitness$ de 0 al evaluarlo con los ejemplos del Grupo 1 y un $fitness$ 0.0001 sobre el Grupo 2. Además, resuelve en un 100% de las ejecuciones los dos conjuntos de ejemplos. Cuando se revisa su comportamiento frente a las instancias con tiempos medios y grandes, obtiene valores de hits de 100%, con errores del orden de 0,0001.

Luego se describe A_7^2, el que resuelve las instancias con tiempos pequeños con un $fitness$ de 0 los ejemplos del grupo 1 y con $fitness$ de 0.0004 los ejemplos del Grupo 2, con un 100% de veces resueltos correctamente ambos grupos. Al revisar su comportamiento con las instancias con tiempo medio, obtiene un 100% de tasa de $hits$, con un error de 0,0005. Con las instancias con tiempo grande, también obtiene 100% de $hits$ con errores de 0 y 0,0007.

Los mejores algoritmos comparten estructuras con algunas de las metaheurísticas seleccionadas. En primer lugar, en A_3^1 se presentan estructuras clásicas de ILS como la generación de vecindad con el $NR\_Movimiento2-OPT$ y perturbación de la solución con $ShiftCity$. Sin embargo, el terminal $ShiftCity$ se presenta en conjunto con $Boltzman$ en un terminal híbrido. Esto indica que se genera una metaheurística híbrida entre ILS y SA, que iterativamente realiza búsquedas locales y acepta soluciones en base al parámetro de temperatura. Luego, se siguen presentando las mismas estructuras que combinan los mismo terminales, inclusive $ShiftCity$ y $MonotonaBoltzman$ como terminales independientes. En el algoritmo A_4^2 se presentan solo terminales característicos de ILS, debido a que se combinan un terminal que explora vecindad ($NR\_Movimiento2-OPT$) y un terminal de perturbación ($ShiftCity$). A pesar de que existe un terminal de aceptación probabilística ($NA-Boltzman\_NoCooling$), este es inocuo debido a que no se presenta ninguna estructura que realice un cambio en la temperatura, por lo que la probabilidad de aceptación siempre se calcula con temperatura constante (la temperatura inicial). Finalmente, en A_7^2 se presentan terminales característicos de GRASP, que corresponden a terminales de búsqueda heurística ($NR\_Movimiento2-OPT$) con búsqueda local ($LS\_SwapCitiesrule$). Luego, se presentan estructuras similares a ILS con $NR\_Movimiento2-OPT$ y $NR\_BlockReverse$ o $NR\_Movimiento2-OPT$ y $NR1\_ShiftCity$.

Es importante destacar que A_7^2 es un algoritmo más pequeño que los anteriores y logra resultados de calidad similar.

4.2.5 Comparación con los algoritmos de la literatura

Los algoritmos seleccionados son competitivos con los encontrados en la literatura. En cada columna de las tablas 4.4, 4.6 y 4.8 se puede encontrar el valor del $fitness$ porcentual (es porcentual dado que se usa el ERP multiplicado por 100, para poder comparar con los valores publicados) al evaluar el mejor algoritmo metaheurístico (un algoritmo genético GA publicado en (Cacchiani et al., 2020) y los algoritmos generados frente a los ejemplos de los grupos 1 y 2. En las dos últimas filas se muestra el valor promedio y la desviación estándar de los $fitness$. Luego, en las tablas 4.5, 4.7 y 4.9 se presentan los tiempos que demoran ejecutarse los algoritmos luego de 10 ejecuciones. En las dos últimas filas de estas tablas se pueden encontrar los promedios y las desviaciones estándar de los tiempos.

Los algoritmos propuestos tienen un menor porcentaje de error que el algoritmo genético. En la tabla 4.4 se ve que los algoritmos propuestos presentan un valor de $fitness$ porcentual igual a 0 en la mayoría de las evaluaciones, exceptuando en la instancia eil40, en la que A_3^1 y A_7^2 tienen un error mayor, y en swiss42 A_4^2 y A_7^2 tienen error mayor. Por otro lado, los algoritmos propuestos tienen un desempeño mejor al resolver dantzig42. Es importante destacar que los algoritmos propuestos tienen un error porcentual igual a 0 en más instancias que GA (GA tiene 12 $fitness$ igual a 0, A_3^1 tiene 19 iguales a 0, A_4^2 tiene 20 iguales a 0 y A_7^2 tiene 16 igual a 0). Luego, el valor promedio de los $fitness$ de los algoritmos propuestos es menor que el promedio de GA, al igual que las desviaciones estándar. Después, al revisar la tabla 4.6 se encuentran resultados similares a los encontrados con las funciones de tiempo pequeño, dado que los algoritmos propuestos tienen un desempeño mejor al resolver dantzig42 pero no al resolver swiss42. Sin embargo, al resolver eil40 los algoritmos propuestos tienen mejor desempeño que GA (al contrario que con la función de tiempo pequeño). Finalmente, al observar la tabla 4.8 no se presenta un comportamiento común para los algoritmos generados. Por ejemplo, para bayg29 A_3^1 tiene un $fitness$ porcentual de 0,62 mientras que A_4^2 y A_7^2 tienen un $fitness$ porcentual de 0,002 y 0,03 respectivamente. Por otro lado, al evaluar dantzig30, GA obtiene un $fitness$ porcentual de 0,57, mientras que A_3^1, A_4^2 y A_7^2 tienen 0,19, 0,00 y 0,00 respectivamente. otro caso se ve al evaluar dantzig42, en que GA obtiene 0,00, A_3^1 obtiene 0,09, A_4^2 obtiene 0,45 y A_7^2 obtiene 0,37. Si se observa la última fila, GA es el que obtiene mejores resultados en promedio y con una menor dispersión.

Los tiempos de ejecución de los algoritmos propuestos son menores en promedio

que el algoritmo genético. En las últimas filas de la tabla 4.5 se observa que, en promedio, los algoritmos propuestos demoran menos tiempo que el algoritmo genético luego de 10 ejecuciones y con una dispersión de valores menor. Además, GA demora más tiempo mientras más grande sea la instancia, mientras que los algoritmos propuestos no muestran esta tendencia. Sin embargo, los algoritmos propuestos si tienen una tendencia a demorar valores similares sobre la mismas instancias. Por ejemplo, al evaluar A_3^1, A_4^2 y A_7^2 sobre la instancia swiss30, demoran 0,15, 0,20 y 0,13 segundos respectivamente. Por otro lado, al evaluarlos sobre eil30, se obtienen valores mucho mayores (0,62, 0,95 y 0,61 respectivamente). Por otro lado, al mirar los tiempos de las tablas 4.7 y 4.9 nos e ve esta tendencia de los algoritmos propuestos, pero si una tendencia de GA a demorar más mientras más grande sea la instancia. Además, en ambas tablas se observa que los tiempos promedios de ejecución de los algoritmos propuestos son menores respecto a GA. Finalmente, se debe considerar que los tiempos medidos para GA no están bajo las mismas condiciones que los algoritmos propuestos, dado que se ejecutan sobre máquinas distintas.

Tabla 4.4: Porcentaje error promedio de evaluación para función de tiempo de servicio pequeño (función 1 de la sección 3.1.6)

	$fitness(GA)$	$fitness(A_3^1)$	$fitness(A_4^2)$	$fitness(A_7^2)$
burma14	0,00	0,00	0,00	0,00
ulysses16	0,06	0,00	0,00	0,00
gr17	0,00	0,00	0,00	0,00
gr21	0,00	0,00	0,00	0,03
ulysses22	0,00	0,00	0,00	0,00
gr24	0,00	0,00	0,00	0,00
fri26	0,00	0,00	0,00	0,00
bayg29	0,05	0,00	0,00	0,01
bays29	0,16	0,00	0,00	0,00
att30	0,00	0,00	0,00	0,00
dantzig30	0,21	0,00	0,00	0,00
eil30	0,25	0,00	0,00	0,00
gr30	0,00	0,01	0,00	0,00
hk30	0,00	0,00	0,00	0,00
swiss30	0,00	0,00	0,00	0,00
eil35	0,11	0,00	0,00	0,08
gr35	0,00	0,00	0,00	0,00
swiss35	0,00	0,00	0,00	0,00
eil40	0,02	0,05	0,00	0,07
dantzig42	0,59	0,00	0,01	0,01
swiss42	0,03	0,03	0,05	0,11
eil45	0,07	0,00	0,00	0,00
Avg	0,07	0,00	0,00	0,01
SD	0,14	0,01	0,01	0,03

Fuente: Elaboración Propia, (2022)

Tabla 4.5: Tiempo total en segundos evaluación para función de tiempo de servicio pequeño

	$Tiempo(GA)$	$Tiempo(A_3^1)$	$Tiempo(A_4^2)$	$Tiempo(A_7^2)$
burma14	0,13	0,25	0,33	0,22
ulysses16	0,13	0,07	0,10	0,09
gr17	0,15	0,25	0,34	0,22
gr21	0,22	0,46	0,63	0,40
ulysses22	0,23	0,20	0,28	0,19
gr24	0,30	0,10	0,14	0,09
fri26	0,33	0,14	0,20	0,14
bayg29	0,43	0,27	0,36	0,24
bays29	0,42	0,35	0,48	0,34
att30	0,46	0,30	0,36	0,25
dantzig30	0,45	0,26	0,36	0,24
eil30	0,46	0,62	0,95	0,61
gr30	0,48	0,46	0,64	0,42
hk30	0,48	0,14	0,13	0,36
swiss30	0,47	0,15	0,20	0,13
eil35	0,68	0,26	0,33	0,23
gr35	0,74	0,25	0,32	0,22
swiss35	0,73	0,25	0,35	0,25
eil40	0,96	0,35	0,46	0,31
dantzig42	1,08	0,42	0,59	0,40
swiss42	1,07	0,59	0,72	0,48
eil45	1,30	0,18	0,24	0,16
Avg	0,53	0,29	0,39	0,27
SD	0,32	0,15	0,21	0,13

Fuente: Elaboración Propia, (2022)

Tabla 4.6: Porcentaje error promedio de evaluación para función de tiempo de servicio medio

	$fitness(GA)$	$fitness(A_3^1)$	$fitness(A_4^2)$	$fitness(A_7^2)$
burma14	0,00	0,00	0,00	0,00
ulysses16	0,02	0,00	0,00	0,00
gr17	0,00	0,00	0,00	0,00
gr21	0,00	0,00	0,00	0,00
ulysses22	0,38	0,00	0,00	0,00
gr24	0,00	0,00	0,00	0,00
fri26	0,00	0,00	0,00	0,00
bayg29	0,00	0,01	0,00	0,01
bays29	0,15	0,00	0,00	0,00
att30	0,00	0,00	0,00	0,00
dantzig30	0,09	0,00	0,00	0,00
eil30	0,17	0,00	0,00	0,00
gr30	0,00	0,02	0,00	0,00
hk30	0,00	0,00	0,00	0,00
swiss30	0,00	0,00	0,00	0,00
eil35	0,16	0,00	0,00	0,00
gr35	0,00	0,00	0,00	0,00
swiss35	0,00	0,00	0,00	0,00
eil40	0,16	0,01	0,02	0,02
dantzig42	0,97	0,07	0,17	0,20
swiss42	0,07	0,14	0,09	0,17
eil45	0,25	0,00	0,00	0,00
Avg	0,11	0,01	0,01	0,02
SD	0,22	0,03	0,04	0,05

Fuente: Elaboración Propia, (2022)

Tabla 4.7: Tiempo total en segundos evaluación para función de tiempo de servicio medio

	$Tiempo(GA)$	$Tiempo(A_3^1)$	$Tiempo(A_4^2)$	$Tiempo(A_7^2)$
burma14	0,11	0,26	0,42	0,28
ulysses16	0,13	0,08	0,12	0,08
gr17	0,15	0,29	0,41	0,27
gr21	0,22	0,50	0,77	0,49
ulysses22	0,23	0,22	0,33	0,26
gr24	0,29	0,11	0,15	0,13
fri26	0,34	0,15	0,30	0,19
bayg29	0,43	0,28	0,47	0,31
bays29	0,42	0,39	0,59	0,43
att30	0,45	0,34	0,46	0,29
dantzig30	0,45	0,28	0,45	0,28
eil30	0,45	0,73	0,94	0,49
gr30	0,46	0,49	0,86	0,54
hk30	0,47	0,15	0,19	0,18
swiss30	0,47	0,16	0,26	0,17
eil35	0,68	0,26	0,43	0,26
gr35	0,75	0,25	0,42	0,25
swiss35	0,73	0,27	0,41	0,29
eil40	0,95	0,36	0,56	0,37
dantzig42	1,06	0,44	0,72	0,45
swiss42	1,05	0,54	0,91	0,57
eil45	1,30	0,19	0,28	0,18
Avg	0,53	0,31	0,48	0,31
SD	0,33	0,16	0,24	0,14

Fuente: Elaboración Propia, (2022)

Tabla 4.8: Porcentaje error promedio de evaluación para función de tiempo de servicio grande

	$fitness(GA)$	$fitness(A_3^1)$	$fitness(A_4^2)$	$fitness(A_7^2)$
burma14	0,00	0,23	0,00	0,00
ulysses16	0,00	0,00	0,00	0,00
gr17	0,00	0,18	0,19	0,08
gr21	0,00	1,24	0,37	0,04
ulysses22	0,00	0,17	0,00	0,00
gr24	0,00	0,26	0,00	0,00
fri26	0,00	0,43	0,00	0,00
bayg29	0,05	0,62	0,02	0,03
bays29	0,16	0,02	0,02	0,00
att30	0,00	0,11	0,01	0,00
dantzig30	0,57	0,19	0,00	0,00
eil30	0,15	0,00	0,00	0,00
gr30	0,00	0,10	0,00	0,74
hk30	0,00	0,00	0,00	0,00
swiss30	0,00	0,60	0,00	0,00
eil35	0,24	0,77	0,07	0,15
gr35	0,00	0,32	0,00	0,00
swiss35	0,00	0,18	0,09	0,10
eil40	0,07	0,45	0,38	0,24
dantzig42	0,00	0,09	0,45	0,37
swiss42	0,05	1,18	0,67	0,09
eil45	0,07	0,00	0,01	0,00
Avg	0,06	0,32	0,10	0,08
SD	0,13	0,36	0,19	0,17

Fuente: Elaboración Propia, (2022)

Tabla 4.9: Tiempo total en segundos evaluación para función de tiempo de servicio grande

	$Tiempo(GA)$	$Tiempo(A_3^1)$	$Tiempo(A_4^2)$	$Tiempo(A_7^2)$
burma14	0,11	0,22	0,13	0,09
ulysses16	0,13	0,05	0,02	0,03
gr17	0,15	0,29	0,11	0,09
gr21	0,22	0,36	0,30	0,23
ulysses22	0,23	0,13	0,07	0,11
gr24	0,29	0,05	0,03	0,04
fri26	0,34	0,07	0,04	0,06
bayg29	0,43	0,17	0,11	0,10
bays29	0,43	0,21	0,19	0,19
att30	0,45	0,21	0,13	0,13
dantzig30	0,44	0,20	0,09	0,10
eil30	0,44	0,40	0,33	0,33
gr30	0,47	0,35	0,18	0,17
hk30	0,44	0,15	0,04	0,04
swiss30	0,47	0,14	0,05	0,05
eil35	0,59	0,18	0,10	0,09
gr35	0,70	0,18	0,09	0,09
swiss35	0,72	0,16	0,13	0,10
eil40	0,80	0,26	0,13	0,13
dantzig42	1,03	0,37	0,20	0,17
swiss42	1,04	0,39	0,23	0,20
eil45	1,28	0,11	0,07	0,07
Avg	0,51	0,21	0,13	0,12
SD	0,31	0,11	0,08	0,07

Fuente: Elaboración Propia, (2022)

4.2.6 Pruebas estadísticas

Debido a la naturaleza estocástica de los experimentos, se hace necesario realizar pruebas estadísticas para verificar si existe una diferencia significativa entre los algoritmos generados y el GA existente en la literatura. Se tiene como hipótesis nula que los algoritmos generados no tienen un $fitness$ significativamente más bajo que el GA, y se plantea como hipótesis alternativa que los algoritmos generados si tienen un $fitness$ significativamente menor.

En primer lugar, se necesita comprobar la normalidad de los datos con un test de normalidad (Luque & Alba, 2011). En este caso, se usa el test de Kolmogorov-Smirnov, para lo que se evalúan los algoritmos generados sobre todas los ejemplos con tiempos de servicio pequeños y se promedian los $fitness$. Se plantea como hipótesis nula que los datos siguen una distribución normal y como hipótesis alternativa que no la siguen. Al ejecutar el test, se obtenie un p-value de 0.0000378, lo que rechaza la hipótesis nula e indica que los datos no siguen una distribución normal.

Debido a que los datos no siguen una distribución normal, se debe usar un test no paramétrico para verificar si los algoritmos propuestos son significativamente mejores que el GA. Para esto, se aplica el test no paramétrico de MannWhitney, el que entrega un p-value de 0.0419, lo que rechaza la hipótesis nula e indica que la familia de algoritmos propuestos son significativamente distintos. De lo anterior, los algoritmos generados son significativamente mejores que el propuesto en la literatura para los tiempos de servicio pequeños.

CAPÍTULO 5. CONCLUSIONES

En esta tesis se han generado nuevos algoritmos metaheurísticos de manera automática para resolver el TSP-TS utilizando técnicas de PG. Utilizando una máquina preparada para realizar PG, se combinaron componentes fundamentales de metaheurísticas clásicas para generar algoritmos que fueran capaces de resolver distintas instancias del TSP-TS. Además, se revisó y analizó el proceso evolutivo desde el que emergen estos nuevos algoritmos, para posteriormente evaluar la calidad de la solución y el tiempo de ejecución frente a diversos ejemplos de problemas del TSP-TS y compararlos con los mejores algoritmos presentes en la literatura.

Como resultado de este trabajo se puede indicar que los algoritmos generados pueden resolver las instancias para las cuales estaba dirigido originalmente el proyecto. Dentro de los algoritmos generados se identifican algunos que pueden resolver todas las instancias que se usaron tanto para evolucionar como evaluar ambos grupos de algoritmos. Específicamente, algoritmos que presentan un rendimiento destacable como A_3^1, A_4^2 y A_7^2, son capaces de resolver el grupo 1 y grupo 2 de instancias con tiempos de servicio pequeños con valores de $hits$ del 100%. Más aún, los algoritmos mencionados también son capaces de resolver ejemplos distintos, considerando las instancias originales, pero modificando los tiempos de servicio de los nodos con polinomio de tiempo medios y grandes. Además, los valores de error obtenidos por lo algoritmos propuestos obtienen porcentajes de error más pequeños que el algoritmo propuesto en Cacchiani et al. (2020) en una gran cantidad de instancias tanto para las funciones de tiempo pequeño como tiempo medio. Sin embargo, a la hora de evaluar las instancias con tiempo grande se obtienen errores promedio mayores. Dado lo anterior, los algoritmos generados tienen un rendimiento mejor en las instancias con tiempos pequeños y medios, sin dejar de lado que la generación de los algoritmos solo contemplaba la resolución de instancias con tiempo pequeño. En consecuencia, es posible asegurar que se pueden generar algoritmos automáticamente para el TSP-TS, que pueden resolver distintos ejemplos del problema encontrando valores óptimos y en tiempos acotados.

No todos los terminales ni funciones aparecen en los mejores algoritmos elegidos, lo que indica que algunos de éstos tienen un mejor desempeño que otros. Se presentan en una gran cantidad terminales como $NR\_Movimiento2Opt$, $ShifCityAndBoltzman$ o $NR1\_ShiftCity$ y terminales de ejecución probabilística. Sin embargo, el modificar los valores de los parámetros podría dar pie a que se generen algoritmos con una mayor variabilidad de terminales y funciones o que su composición sea completamente distinta pudiendo incluso mejorar los resultados.

En los nuevos algoritmos automáticamente generados se detecta la presencia de módulos que contienen las metaheurísticas clásicas. Así como se ha demostrado en la literatura, la combinación de metaheurísticas producen mejores resultados que las metaheurísticas por si solas. Los algoritmos A_3^1 y A_7^2 presentan en su estructura componentes fundamentales de las

metaheurísticas seleccionadas y presentan valores de $fitness$ menores que GA en instancias con tiempo pequeño y medio.

El proceso evolutivo converge rápidamente sistemáticamente para todos los procesos evolutivos. Para cada uno de los procesos, se generan 50000 algoritmos (500 algoritmos por 100 generaciones) y se obtiene una estabilización casi constante de la mejora de los $fitness$ de los algoritmos generados dentro de las primeras 30 generaciones, lo que indica que se podrían realizar evoluciones con un costo computacional mejor obteniendo la misma calidad en los algoritmos generados.

Desde el trabajo realizado, emanan brechas que pueden ser cubiertas por trabajos realizables a posterior. Por ejemplo, se pueden modificar los parámetros elegidos para realizar el proceso evolutivo, con la finalidad de potenciar la diversificación en el espacio de búsqueda. Esto se podría lograr reduciendo específicamente el valor de cruzamiento. Además, se puede probar el rendimiento de los algoritmos generados frente a instancias que no han sido contempladas en el trabajo. Existe una gran cantidad de instancias clásicas del TSP que podrían ser transformadas en instancias del TSP-TS, cuya resolución no se aborda en este trabajo ni en trabajos de actuales de la literatura, por lo que queda un amplio camino por recorrer a la hora de evaluar los algoritmos propuesto. Por otro lado, se puede también probar con instancias con diferentes funciones de tiempo (medio, grande, cuadrático u otros) para realizar el proceso evolutivo y evaluativo, generando una gran cantidad de posibles combinaciones entre grupos de evolución y evaluación.

REFERENCIAS BIBLIOGRÁFICAS

Abeledo, H., Fukasawa, R., Pessoa, A., & Uchoa, E. (2013). The time dependent traveling salesman problem: Polyhedra and algorithm. *Mathematical Programming Computation*.

Acevedo, N., Rey, C., Contreras-Bolton, C., & Parada, V. (2020). Automatic design of specialized algorithms for the binary knapsack problem. *Expert Systems with Applications*, *141*, 112908.

Applegate, D. L., Bixby, R. E., Chvátal, V., & Cook, W. J. (2011). *The traveling salesman problem: A computational study*.

Arigliano, A., Calogiuri, T., Ghiani, G., & Guerriero, E. (2018). A branch-and-bound algorithm for the time-dependent travelling salesman problem. *Networks*.

Arigliano, A., Ghiani, G., Grieco, A., Guerriero, E., & Plana, I. (2019). Time-dependent asymmetric traveling salesman problem with time windows: Properties and an exact algorithm. *Discrete Applied Mathematics*.

Bigras, L. P., Gamache, M., & Savard, G. (2008). The time-dependent traveling salesman problem and single machine scheduling problems with sequence dependent setup times. *Discrete Optimization*.

Boland, N., Hewitt, M., Marshall, L., & Savelsbergh, M. (2017). The continuous-time service network design problem. *Operations Research*.

Burke, E. K., Gendreau, M., Hyde, M., Kendall, G., Ochoa, G., Özcan, E., & Qu, R. (2013). Hyper-heuristics: A survey of the state of the art. *Journal of the Operational Research Society*, *64*(12), 1695–1724.

Cacchiani, V., Contreras-Bolton, C., & Toth, P. (2020). Models and algorithms for the Traveling Salesman Problem with Time-dependent Service times. *European Journal of Operational Research*, *283*(3), 825–843.
URL `https://doi.org/10.1016/j.ejor.2019.11.046`

Contreras Bolton, C., Gatica, G., & Parada, V. (2013). Automatically Generated Algorithms for the Vertex Coloring Problem. *PLoS ONE*, *8*(3), e58551.
URL `https://dx.plos.org/10.1371/journal.pone.0058551`

Cordeau, J. F., Ghiani, G., & Guerriero, E. (2014). Analysis and branch-and-cut algorithm for the time-dependent travelling salesman problem. *Transportation Science*.

Fortnow, L. (2009). The status of the P versus NP problem. *Communications of the ACM*, *52*(9), 78–86.

Garey, M. R., & Johnson, D. S. (1979). Computers and Intractability: A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences). *Computers and Intractability*.

Gendreau, M., & Potvin, J.-Y. (2019). *Handbook of Metaheuristics*. Springer International Publishin.

Ichoua, S., Gendreau, M., & Potvin, J. Y. (2003). Vehicle dispatching with time-dependent travel times. *European Journal of Operational Research*.

Jun, S., Lee, S., & Yih, Y. (2021). Pickup and delivery problem with recharging for material handling systems utilising autonomous mobile robots. *European Journal of Operational Research*, *289*(3), 1153–1168.

Kayé, B. K. B., Diaby, M., Koivogui, M., & Oumtanaga, S. (2021). A memetic algorithm for an external depot production routing problem. *Algorithms*, *14*(1), 1–24.

Kinable, J., Cire, A. A., & van Hoeve, W. J. (2017). Hybrid optimization methods for time-dependent sequencing problems. *European Journal of Operational Research*.

Koza, J. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*.

Koza, J. (1994). *Genetic Programming II: Automatic Discovery of Reusable Programs*.

Koza, J. (1999). *Genetic Programming III: Darwinian Invention and Problem Solving*.

Koza, J. (2003). *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*.

Loyola, C., Sepúlveda, M., Solar, M., Lopez, P., & Parada, V. (2016). Automatic design of algorithms for the traveling salesman problem. *Cogent Engineering*, *3*(1), 1–15.
URL http://dx.doi.org/10.1080/23311916.2016.1255165

Luke, S. (2010). The ECJ Owner ' s Manual. *Statistics*, *177*, 1–16.

Luque, G., & Alba, E. (2011). Parallel genetic algorithms: Theory and realworld applications. *Studies in Computational Intelligence*.

Martí, R., Martínez-Gavara, A., & Sánchez-Oro, J. (2021). The capacitated dispersion problem: an optimization model and a memetic algorithm. *Memetic Computing*, *13*(1), 131–146.
URL https://doi.org/10.1007/s12293-020-00318-1

Minh Vu, D., Hewitt, M., Boland, N., & Savelsbergh, M. (2018). Solving time dependent traveling salesman problems with time windows. *Optimization Online*.

Miranda-Bront, J. J., Méndez-Díaz, I., & Zabala, P. (2014). Facets and valid inequalities for the time-dependent travelling salesman problem. *European Journal of Operational Research*.

Montero, A., Méndez-Díaz, I., & Miranda-Bront, J. J. (2017). An integer programming approach for the time-dependent traveling salesman problem with time windows. *Computers and Operations Research*.

Parada, L., Herrera, C., Sepúlveda, M., & Parada, V. (2016). Evolution of new algorithms for the binary knapsack problem. *Natural Computing*, *15*(1), 181–193.
URL https://link.springer.com/article/10.1007/s11047-015-9483-8

Picard, J. C., & Queyranne, M. (1978). TIME-DEPENDENT TRAVELING SALESMAN PROBLEM AND ITS APPLICATION TO THE TARDINESS PROBLEM IN ONE-MACHINE SCHEDULING. *Operations Research*.

Ryser-Welch, P., Miller, J. F., & Asta, S. (2015). Generating human-readable algorithms for the Travelling Salesman Problem using hyper-heuristics. *GECCO 2015 - Companion Publication of the 2015 Genetic and Evolutionary Computation Conference*, (July), 1067–1074.

Smith-Miles, K., & Lopes, L. (2012). Measuring instance difficulty for combinatorial optimization problems. *Computers and Operations Research*, *39*(5), 875–889.
URL http://dx.doi.org/10.1016/j.cor.2011.07.006

Sun, P., Veelenturf, L. P., Dabia, S., & Van Woensel, T. (2018). The time-dependent capacitated profitable tour problem with time windows and precedence constraints. *European Journal of Operational Research*.

Talbi, E.-G. (2009). *Metaheuristics: From Design to Implementation*.

Taş, D., Gendreau, M., Jabali, O., & Laporte, G. (2016). The traveling salesman problem with time-dependent service times. *European Journal of Operational Research*, *248*(2), 372–383.

Vander Wiel, R. J., & Sahinidis, N. V. (1996). An exact solution approach for the time-dependent traveling-salesman problem. *Naval Research Logistics*.

Vela, C. R., Afsar, S., Palacios, J. J., González-Rodríguez, I., & Puente, J. (2020). Evolutionary tabu search for flexible due-date satisfaction in fuzzy job shop scheduling. *Computers and Operations Research*, *119*, 104931.

ANEXO A. ESTRUCTURA DE ALGORITMOS

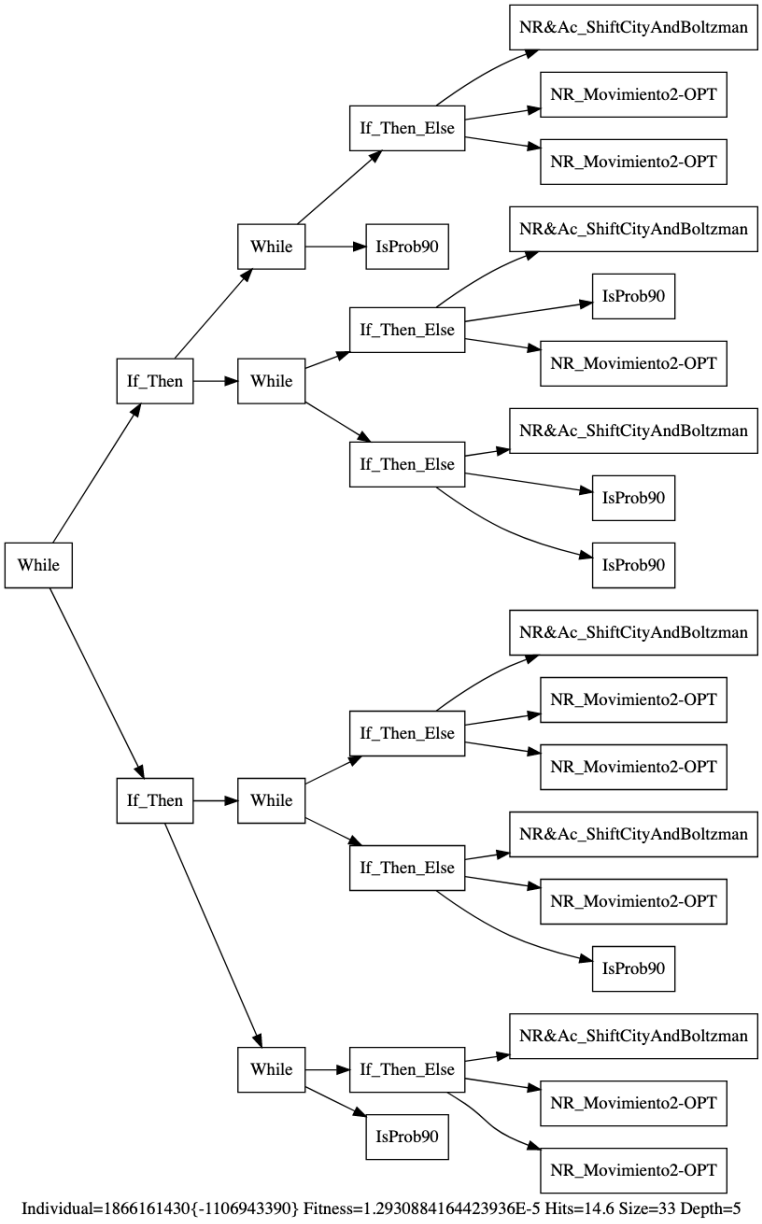

Figura A.1: Estructura A_3^1. Fuente: Elaboración propia (2022)

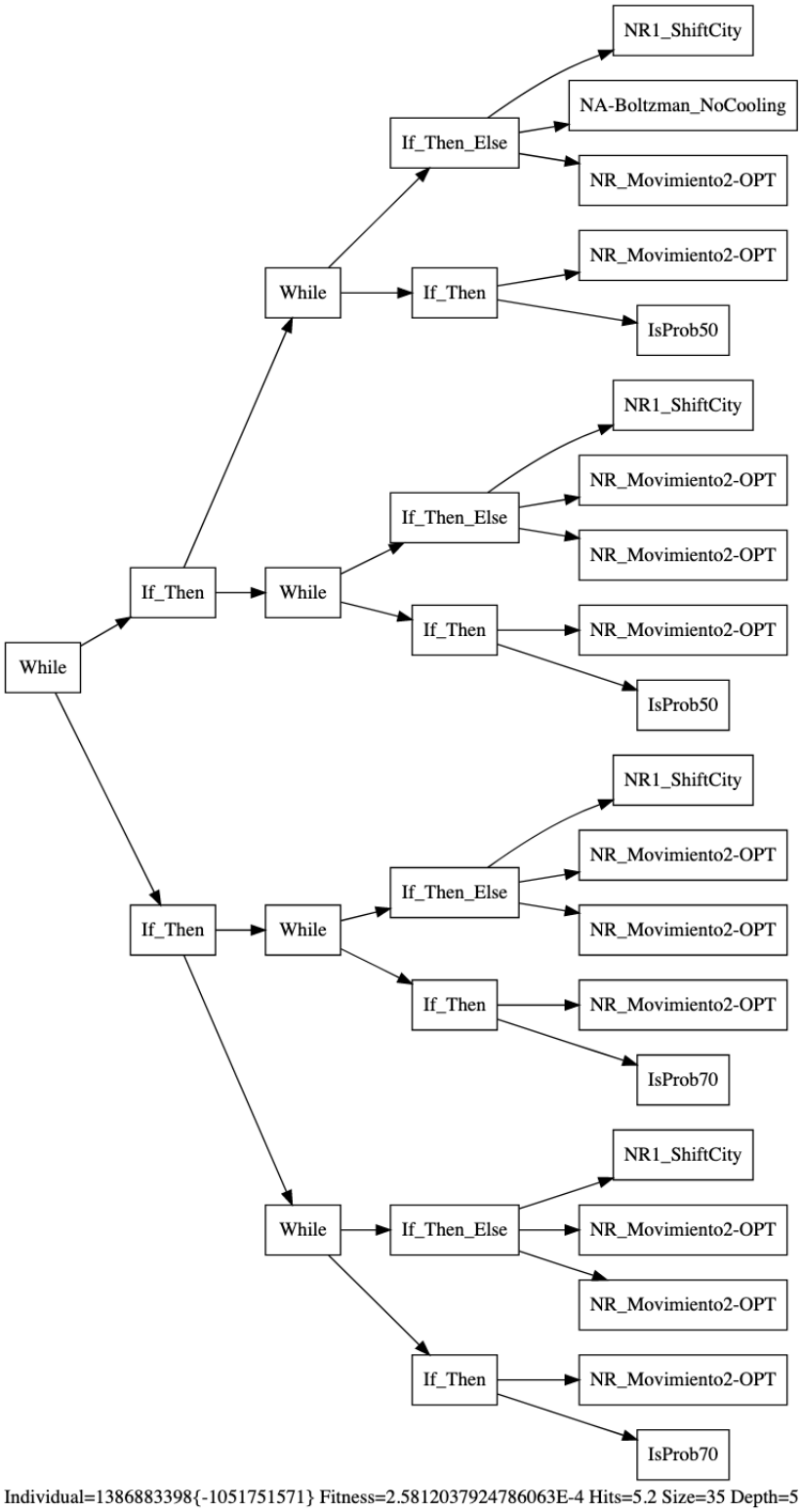

Figura A.2: Estructura A_4^2. Fuente: Elaboración propia (2022)

Figura A.3: Estructura A_7^2. Fuente: Elaboración propia (2022)

ANEXO B. ALGORITMO 2-OPT

Algoritmo B.1: Pseudocódigo de 2OptSwap (ruta, i, k)

1: DEFINIR nueva_ruta COMO arreglo
2: primera_ruta = camino de $ruta_0$ A $ruta_{i-1}$
3: AGREGAR primera_ruta A nueva_ruta
4: segunda_ruta = camino de $ruta_k$ A $ruta_i$
5: AGREGAR segunda_ruta A nueva_ruta
6: final_ruta = camino de $ruta_{k+1}$ A $ruta_{largo(ruta)}$
7: AGREGAR final_ruta A nueva_ruta
8: DEVOLVER nueva_ruta

Algoritmo B.2: Pseudocódigo de 2OptSwap (solución_propuesta)

```
1:  mejor_costo = calcular_costo(solución_propuesta)
2:  for i = 0; i <= tamaño_solución - 1; i++ do
3:     for  k = i + 1; k <= tamaño_solución; k++ do
4:        nueva_ruta = 2optSwap(solución_propuesta, i, k)
5:        nuevo_costo = calcular_costo(nueva_ruta)
6:        if  nuevo_costo < mejor_costo then
7:           solución_propuesta = nueva_ruta
8:           mejor_costo = nuevo_costo
9:        end if
10:    end for
11: end for
```

ANEXO C. PSEUDOCÓDIGO DE MEJORES ALGORITMOS

Algoritmo C.2: Pseudocódigo de A_4^2

```
1:  for n = 0; n < 30; AUMENTAR n EN 1 do
2:      DEFINIR IsProb70WasExecuted_1 COMO false;
3:      while IsProb70() do
4:          DEFINIR IsProb70WasExecuted_1 COMO true, NR_Movimiento2-OPT();
5:          if NR_Movimiento2-OPT() then
6:              NR_Movimiento2-OPT();
7:          else
8:              NR1_ShiftCity();
9:          end if
10:     end while
11:     if IsProb70WasExecuted_1 then
12:         DEFINIR IsProb70WasExecuted_2 COMO false;
13:         while IsProb70WasExecuted_2 do
14:             NR_Movimiento2-OPT();
15:             if NR_Movimiento2-OPT() then
16:                 NR_Movimiento2-OPT();
17:             else
18:                 NR1_ShiftCity();
19:             end if
20:         end while
21:         DEFINIR IsProb50WasExecuted COMO false;
22:         while IsProb50() do
23:             DEFINIR IsProb50WasExecuted COMO true, NR_Movimiento2-OPT();
24:             if NR_Movimiento2-OPT() then
25:                 NR_Movimiento2-OPT();
26:             else
27:                 NR1_ShiftCity();
28:             end if
29:         end while
30:         while IsProb50WasExecuted AND IsProb50() do
31:             NR_Movimiento2-OPT();
32:             if NR_Movimiento2-OPT() then
33:                 NA-Boltzman_NoCooling();
34:             else
35:                 NR1_ShiftCity();
36:             end if
37:         end while
38:     end if
39: end for
```

Algoritmo C.1: Pseudocódigo de A_7^2

```
1:  for n = 0; n < 30; AUMENTAR n EN 1 do
2:      DEFINIR Prob30WasExecuted COMO false;
3:      while Prob30() do
4:          DEFINIR Prob30WasExecuted COMO true;
5:          if NR_Movimiento2-OPT() then
6:              LS_SwapCitiesRule();
7:          else
8:              NRn_BlockReverse();
9:          end if
10:     end while
11:     if Prob30WasEjecuted then
12:         while CSch_LinearCooling() do
13:             if NR_Movimiento2-OPT() then
14:                 NR_Movimiento2-OPT();
15:             else
16:                 NRn_BlockReverse();
17:             end if
18:         end while
19:     end if
20:     if Prob30WasExecuted then
21:         while Prob30() do
22:             if NR_Movimiento2-OPT() then
23:                 NR_Movimiento2-OPT();
24:             else
25:                 NR1_ShifCity();
26:             end if
27:         end while
28:         while CSch_LinearCooling() do
29:             if NR_Movimiento2-OPT() then
30:                 NR_Movimiento2-OPT();
31:             else
32:                 NRn_BlockReverse();
33:             end if
34:         end while
35:     end if
36: end for
```

Algoritmo C.3: Pseudocódigo de A_3^1

```
1:  for n = 0; n < 30; AUMENTAR n EN 1 do
2:      DEFINIR isProb90WasExecuted_1 COMO false, isBetter COMO true, isAccepted COMO true, isAccepted_2
        COMO true;
3:      while IsProb90() do
4:          DEFINIR isProb90WasExecuted_1 COMO true;
5:          if NR_Movimiento2-OPT() then
6:              NR_Movimiento2-OPT();
7:          else
8:              NR&Ac_ShiftCityAndBoltzman();
9:          end if
10:     end while
11:     if isProb90WasExecute_1 then
12:         while isBetter AND isAccepted do
13:             if IsProb90() then
14:                 isBetter COMO NR_Movimiento2-OPT();
15:             else
16:                 isAccepted = ShiftCityAndBoltzman();
17:             end if
18:             if  NR_Movimiento2-OPT() then
19:                 NR_Movimiento2-OPT();
20:             else
21:                 NR&Ac_ShiftCityAndBoltzman();
22:             end if
23:         end while
24:     end if
25:     if isProb90WasExecuted_1 then
26:         while isAccepted_2 do
27:             if NO IsProb90() then
28:                 isAccepted_2 = NR&Ac_ShiftCityAndBoltzman();
29:             end if
30:             if  NO NR_Movimiento2-OPT()  then
31:                 NR&Ac_ShiftCityAndBoltzman();
32:             end if
33:         end while
34:         while IsProb90() do
35:             if NR_Movimiento2-OPT() then
36:                 NR_Movimiento2-OPT();
37:             else
38:                 NR&Ac_ShiftCityAndBoltzman();
39:             end if
40:         end while
41:     end if
42: end for
```