Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Biased Random Key Genetic Algorithm con Búsqueda Local para el Team Orienteering Problem

Tesis presentada para optar al título de
Licenciado en Ciencias de la Computación

Alejandro Federico Lix Klett

Directora: Loiseau, Irene

Buenos Aires, 2018

BIASED RANDOM KEY GENETIC ALGORITHM CON BÚSQUEDA LOCAL PARA EL TEAM ORIENTEERING PROBLEM

En el *Orienteering Problem* (OP) [25], se da un conjunto de nodos, cada uno con un beneficio determinado. El objetivo es determinar una ruta, limitada en su longitud, que visite a algunos nodos maximizando la suma de los beneficios obtenidos. El OP se puede formular de la siguiente manera: dado n nodos en el plano euclidiano cada uno con un beneficio, donde $beneficio(nodo_i) > 0$ si $0 < i < n$ y $beneficio(nodo_1) = beneficio(nodo_n) = 0$, se debe encontrar una ruta de beneficio máximo a través de estos nodos, iniciando en el $nodo_1$ y finalizando en el $nodo_n$, de longitud no mayor que d_{max}. Tsiligirides [25] fue el primero en presentar este problema y lo llamó *Score Orienteering Event* (SOE).

El *Team Orienteering Problem* (TOP) [9] es la generalización al caso de múltiples rutas del *Orienteering Problem*. Resolver el TOP implica encontrar un conjunto de rutas desde el nodo de inicio hasta el nodo final de forma tal que se maximice la sumatoria de los beneficios recolectados, la distancia de todas las rutas no supere a d_{max} y ningún nodo sea visitado más de una vez. El OP pertenece a la clase problemas NP-Completo ya que contiene al problema *Traveling Salesman Problem* como caso especial (ver Garey y Johnson [13]). De la misma manera, el TOP pertenece a la clase de problemas NP-Completo porque contiene al OP como un caso especial donde sólo hay una ruta. Resolver el TOP requiere determinar que subconjunto de nodos se visitarán, a que ruta serán asignados y el orden en que serán visitados.

En este trabajo, propongo una combinación del *Biased Random Key Genetic Algorithm* (BRKGA) [4] y de búsquedas locales para resolver el TOP. El BRKGA es una clase de algoritmos genéticos cuya población inicial es generada utilizando un decodificador que convierte un conjunto de vectores de números enteros aleatorios, en un conjunto de soluciones válidas del problema. El BRKGA es una variante del *Random Key Genetic Algorithm* (RKGA). Estos algoritmos se diferencian en el proceso de apareamiento (*crossover*), mientras que en el RKGA los padres son elegidos al azar entre todos los individuos de la población, en la mayoría de las implementaciones del BRKGA uno de los padres siempre pertenece al subconjunto de los mejores individuos de la población y este padre tiene mayor probabilidad de trasmitir sus genes al individuo resultante del proceso de apareamiento.

En mi algoritmo, en cada nueva generación, la mejor solución se mejora con algunas búsquedas locales. Dada una solución s, un algoritmo de búsqueda local básicamente busca mejores soluciones en la vecindad de s. La solución s' en la vecindad de s, es mejor que s si el beneficio total recolectado por s' es mayor al de s o si sus beneficios recolectados son iguales y la distancia recorrida por las rutas de s' es menor a la de s. En este trabajo implementé los algoritmos de búsqueda local: *Insert*, *Swap*, *2-Opt*, *Simple Replace* y *Mutiple Replace*.

Los experimentos computacionales los realicé en instancias estándar de la literatura. Las instancias se dividen en siete conjuntos. Los primeros tres conjuntos de instancias son los de Tsiligirides [25] y los siguientes cuatro conjuntos son los de Chao et al. [9]. Todas las instancias pueden encontrarse en [18]. Mis resultados fueron comparados con los

resultados obtenidos por los siguientes autores: Chao, Golden y Wasil 1996 [9] (CGW), Tang y Miller-Hooks 2005 [24] (TMH), Archetti, Hertz, Speranza 2007 [1] (AHS), Ke, Archetti y Feng 2008 [19] (KAF) y Bouly, Dang y Moukrim 2010 [5] (BDM).

Los resultados de mi algoritmo son muy buenos dado que en el 70 % de las instancias del benchmark mi implementación obtuvo la mejor solución conocida y para el 30 % restante obtuvo valores competitivos con los trabajos previamente mencionados.

Palabras clave: Team Orienteering Problem, Biased Random Key Genetic Algorithm, Routing Problem, Búsqueda Local, Decodificador.

BIASED RANDOM KEY GENETIC ALGORITHM WITH LOCAL SEARCH FOR THE TEAM ORIENTEERING PROBLEM

In the *Orienteering Problem* (OP) [25], a set of nodes is given, each with a certain benefit. The objective is to determine a path, limited in length, that visits some nodes in order that maximizes the sum of the collected benefits. The OP can be formulated in the following way: given n nodes in the euclidean plane each with a benefit, where $benefit(node_i) > 0$ if $0 < i < n$ and $benefit(node_1) = benefit(node_n) = 0$, find a route of maximum benefit through these nodes beginning at $node_1$ and ending at $node_n$ of length no greater than d_{max}. Tsiligirides [25] was the first one to present this problem and called it *Score Orienteering Event* (SEO).

The *Team Orienteering Problem* (TOP) [9] is the generalization to the case of multiple tours of the *Orienteering Problem.* Solving TOP involves finding a set of paths from the starting node to the ending node such that the total collected benefit received from visiting a subset of nodes is maximized, the length of each path is restricted by d_{max} and no node is visited more than once. The OP belongs to the class of NP-Hard problems, as it contains the well known *Travelling Salesman Problem* as a special case (see Garey y Johnson [13]). In the same way, TOP belongs to the NP-Hard problems as it contains the OP as a special case when there is only one path. Solving TOP requires not only determining a calling order on each tour, but also selecting which subset of nodes in the graph to visit.

In this dissertation, I propose a combination of the *Biased Random Key Genetic Algorithm* (BRKGA) [4] and local searchs to solve the TOP. The BRKGA is a class of genetic algorithms that initializes its population using a decoder that converts a set of random integer vectors into a set of valid solutions of the problem. The BRKGA is a variant of the *Random Key Genetic Algorithm* (RKGA). These algorithms differ in the mating process (*crossover*), while in the RKGA the parents are chosen randomly between all individuals of the population, in most of the BRKGA implementations one of the parents always belongs to the subset of best individuals of the population and this parent has better chances of transmitting his gens to the individual resulting from the mating process.

In my algorithm, in every new generation, the best solution is enhanced with some local searches. Given a solution s, a local search algorithm searches for better solutions in the neighborhood of s. The solution s' in the neighborhood of s, is better than s if the total collected benefit from s' is greater than the one from s or their total collected benefit are equal and the distance traveled by the routes of s' is less than the distance traveled by the routes of s. In this work, I implemented the following local search algorithms: *Insert*, *Swap*, *2-Opt*, *Simple Replace* y *Mutiple Replace*.

I performed the computational experiments in standard instances of the literature. The instances are divided in seven sets. The first three sets of instances are those of Tsiligirides [25] and the other four sets are those of Chao et al. [9]. All instances can be found in [18]. My results were compared with the results obtained by the following authors: Chao, Golden and Wasil 1996 [9] (CGW), Tang and Miller-Hooks 2005 [24] (TMH), Archetti, Hertz and

Speranza 2007 [1] (AHS), Ke, Archetti and Feng 2008 [19] (KAF) and Bouly, Dang and Moukrim 2010 [5] (BDM).

The results of my algorithm are very good given that in 70 % of the instances of the benchmark my implementation obtained the best known solution and for the remaining 30 % it obtained competitive values with the previously mentioned works.

Keywords: Team Orienteering Problem, Biased Random Key Genetic Algorithm, Routing Problem, Local Search Heuristic, Routing Problems, Decoder.

Índice general

1. INTRODUCCIÓN

1.1. Historia

La orientación [9] es un deporte originario de Escandinavia jugado al aire libre normalmente en bosques o zonas montañosas. Con ayuda de un mapa y una brújula, un competidor comienza en un punto de control específico e intenta visitar tantos otros puntos de control como le sea posible dentro de un límite de tiempo prescrito y regresa a un punto de control especificado. Cada punto de control tiene una puntuación asociada, de modo que el objetivo de este deporte es maximizar la puntuación total. Un competidor que llegue al punto final después de que el tiempo haya expirado es descalificado. El competidor elegible con la puntuación más alta es declarado ganador. Dado que el tiempo es limitado, un competidor puede no ser capaz de visitar todos los puntos de control. Por lo tanto cada competidor debe seleccionar un subconjunto de puntos de control para visitar que maximizarán la puntuación total. Este problema se conoce como el *Orienteering Problem* y se denota por OP.

El equipo de orientación extiende la versión de un solo competidor del deporte a un equipo formado por varios competidores (digamos 2, 3 o 4 miembros). Todos los competidores comienzan en el mismo punto y cada miembro del equipo intenta visitar tantos puntos de control como le sea posible dentro de un límite de tiempo prescrito, terminando en el punto final. Una vez que un miembro del equipo visita un punto y se le otorga la puntuación asociada, ningún otro miembro del equipo puede obtener una puntuación por visitar el mismo punto. Por lo tanto, cada miembro de un equipo tiene que seleccionar un subconjunto de puntos de control para visitar, de modo que dos miembros del mismo equipo no visiten el mismo punto de control, el límite de tiempo no sea violado y la puntuación total del equipo sea maximizada. Este problema se conoce como el *Team Orienteering Problem* y lo denotan por TOP.

El OP es NP-Completo como demostraron Golden, Levy, y Vohra [16], por lo que el TOP es al menos tan difícil ya que lo contiene. Es por este motivo que la mayoría de las propuestas para estos problemas se han centrado en proporcionar enfoques heurísticos.

1.2. Aplicaciones Practicas

Tanto OP como TOP han sido reconocidos como modelo de muchas aplicaciones reales diferentes como por ejemplo:

i El deporte de orientación de equipo explicado anteriormente (ver Chao et al [9]).

ii Algunas aplicaciones de servicios de recogida o entrega que implican el uso de transportistas comunes y flotas privadas (ver Ballou y Chowdhury [3]).

iii Tsiligirides [25] hace mención del caso del *Travelling Sales Person* (TSP) sin tiempo suficiente para visitar a todos los clientes y que conoce de antemano el valor aproximado de las ventas que realizará en cada ciudad.

iv La planificación de viajes turísticos donde existen varios puntos de interés que el turista quiere visitar. Cada uno de estos puntos de interés tienen un valor dado por el turista y un tiempo mínimo para poder visitarlo.

v El problema de entrega de combustible con múltiples vehículos de Golden, Levy y Vohra [16]. Una flota de camiones debe entregar combustible a una gran cantidad de clientes diariamente. Una característica clave de este problema es que el suministro de combustible del cliente debe mantenerse en un nivel adecuado en todo momento. Es decir, cada cliente tiene una capacidad de tanque conocida y se espera que su nivel de combustible permanezca por encima de un valor crítico preespecificado que puede denominarse punto de reabastecimiento. Las entregas siguen un sistema de empuje en el sentido de que están programados por la empresa en base a un pronóstico de los niveles de los tanques de los clientes. Los desabastecimientos son costosos y deben evitarse cuando sea posible.

vi El reclutamiento de jugadores de fútbol americano universitario de Butt y Cavalier [7]. Un método exitoso de reclutamiento utilizado en muchas pequeñas divisiones del *National Collegiate Athletic Association* es visitar los campus de las escuelas secundarias y reunirse con los miembros superiores de los equipos de fútbol americano. A modo de maximizar su potencial para reclutar futuros jugadores, deben visitar tantas escuelas secundarias como sea posible dentro de un radio de 100 km del campus, sabiendo por experiencia previa que visitar todas las escuelas en esta área no es posible. Por lo tanto, deben visitar el mejor subconjunto de escuelas en el área.

1.3. Instancias del Benchmark

Para la generación y comparación de resultados se utilizaron instancias de test de Tsiligirides y de Chao [18]. Las instancias de problemas de ambos autores comparten el mismo formato.

Una instancia de TOP contiene:

- N vehículos de carga. Cada vehículo tiene una distancia máxima, llamada d_{max}, que puede recorrer. En esta implementación cada vehículo puede tener una distancia máxima diferente. De todos modos en las instancias de test utilizadas, los vehículos tienen el mismo d_{max}.
- M clientes. Un cliente es un nodo con beneficio mayor a cero. Cada cliente tiene un set de coordenadas X e Y que representan su ubicación en el plano.
- Un nodo de inicio y fin de ruta. Todos los vehículos inician y finalizan el recorrido en estos nodos. Ambos nodos tienen un beneficio de cero y tienen un set de coordenadas X e Y.

También es importante mencionar que:

- Todos los clientes en las instancias del benchmark tienen dos coordenadas y se utiliza la distancia euclidiana para medir distancias.
- Una solución es válida si:
  - Para todo vehículo, la distancia de su ruta es menor o igual al d_{max} del vehículo que realiza tal ruta.
  - Ningún cliente pertenece a dos rutas distintas.
  - Toda ruta parte del nodo de inicio y finaliza en el nodo de fin.
- La función objetivo retorna la sumatoria de los beneficios de los clientes visitados.

1.4. Resumen

En el capítulo 2 se encuentra la revisión bibliográfica, donde se sintetizaron las propuestas de los trabajos previos que resolvieron TOP. En el capítulo 3 se presentará el modelo matemático de TOP que presentaron Tang y Miller-Hooks [24] . En el capítulo 4 describiré los algoritmos genéticos en general, luego se explica en particular el RKGA y el BRKGA. Por último se comenta sobre el algoritmo decodificador que utiliza el BRKGA. En el capítulo 5 se detalla en profundidad la implementación de toda la solución, comenzando por los decodificadores utilizados, su eficiencia, desventajas y comportamiento. Luego se explica mi implementación del algoritmo BRKGA, detallando las configuraciones testeadas, resultados parciales y problemas encontrados. Por último, en ese mismo capítulo se describen las búsquedas locales implementadas, el objetivo de cada una, los distintos órdenes en que se aplicaron las búsquedas locales y los resultados parciales sobre un subconjunto diverso del benchmark de instancias. En el capítulo 6 muestro los resultados finales obtenidos sobre las instancias del benchmark de problemas. Por último, en el capítulo 7 comento sobre las conclusiones y trabajos futuros.

2. REVISIÓN BIBLIOGRÁFICA

Existe una gran cantidad de aplicaciones que pueden ser modelas por TOP. Es por esto que la clase de problemas de enrutamiento de vehículos con ganancias es amplia. En el 2013, C. Archetti, M.G. Speranza, D. Vigo [2] publicaron una revisión sobre esta clase de problemas. Luego en 2016, Gunawan et al. [17] publicaron una revisión complementando la revisión bibliográfica que contenía la revisión de Archetti et al. [2]. En este capítulo se sintetizan varias publicaciones y trabajos previos que encontré relacionados con el TOP.

La primera heurística propuesta para el TOP es un algoritmo de construcción simple introducido por Butt y Cavalier [7] en 1994 y probado en pequeñas instancias de tamaño con hasta 15 nodos. En su heurística *MaxImp*, se asignan pesos a cada par de nodos de modo que cuanto mayor es el peso, más beneficioso es no solo visitar esos dos nodos, sino visitarlos en el mismo recorrido. Su peso depende de cuán beneficioso sean los nodos y las sumas de las distancias de ir y volver por esos nodos. Butt y Cavalier intrudujeron este problema con el nombre *Multiple Tour Maximum Collection Problem*, que posteriormente fue nombrado TOP.

Los primeros que utilizaron el nombre TOP para referenciar el problema fueron Chao, Golden y Wasil (CGW) [9] en 1996, para resaltar la conexión con el más ampliamente estudiado caso de un solo vehículo (OP). En su trabajo utilizaron una heurística de construcción más sofisticada donde la solución inicial se refina a través de movimientos de los clientes, los intercambios y varias estrategias de reinicio. En este trabajo mencionan que TOP puede ser modelado como un problema de optimización multinivel. En el primer nivel, se debe seleccionar un subconjunto de puntos para que el equipo visite. En el segundo nivel, se asignan puntos a cada miembro del equipo. En el tercer nivel, se construye un camino a través de los puntos asignados a cada miembro del equipo. El algoritmo resultante se prueba en un conjunto de 353 instancias de prueba con hasta 102 clientes y hasta 4 vehículos.

El primer algoritmo exacto para TOP fue propuesto por Butt y Ryan [8] en 1999. Comienzan a partir de una formulación de partición configurada y su algoritmo hace un uso eficiente tanto de la generación de columnas como de la bifurcación de restricciones. Gracias a este nuevo algoritmo pudieron resolver instancias con hasta 100 clientes potenciales cuando las rutas incluyen solo unos pocos clientes cada uno. Más recientemente, Boussier et al. [6] presentaron un algoritmo de *Branch and Price*. Gracias a diversos procedimientos de aceleración en el paso de generación de columnas, puede resolver instancias con hasta 100 clientes potenciales del gran conjunto de instancias de referencia propuestas en Chao et al. [9].

El algoritmo de *Búsqueda Tabú* (BT) demostró poder resolver TOP en el trabajo de Tang y Miller-Hooks (TMH) [24] en 2005. Su BT está incorporado en un *Adaptive Memory Procedure* (AMP) que alterna entre vecindarios pequeños y grandes durante la búsqueda. La heurística de búsqueda tabú propuesta por TMH para el TOP se puede caracterizar en términos generales en tres pasos: inicialización, mejora de la solución y evaluación. Como señala Golden et al. [15] 1997, el AMP funciona de forma similar a los algoritmos genéticos, con la excepción de que la descendencia (en AMP, las nuevas soluciones iniciales) se puede generar a partir de más de dos padres. La BT de Tang y Miller-Hooks produjo consistentemente soluciones de alta calidad sobre el conjunto de problemas de Chao et al., superando las propuestas publicadas hasta el momento.

Archetti et al. [1] 2007, proponen dos variantes de un algoritmo de BT generalizada y de un algoritmo llamado *Variable Neighborhood Search* (VNS). El VNS parte de una solución titular s, desde donde se realiza un salto a una solución s'. Se llama salto porque se hace dentro de un vecindario más grande que el vecindario utilizado para la búsqueda tabú. Luego aplican una búsqueda tabú en s' para tratar de mejorarla. La solución resultante s'' se compara luego con s. Si se sigue una estrategia VNS, entonces s'' se convierte en el nuevo titular solo si s'' es mejor que s. En la estrategia de búsqueda tabú generalizada, se establece $s = s''$ incluso si s'' es peor que s. Este proceso se repite hasta que se cumplan algunos criterios de detención.

Ke et al. [19] 2008, proponen un *Algoritmo de la Colonia de Hormigas* (ACH) que utiliza cuatro métodos diferentes para construir soluciones candidatas. Su algoritmo pertenece a la clase de metaheurísticas basadas en una población de soluciones. Utiliza una colonia de hormigas, que están guiadas por rastros de feromonas e información heurística, para construir soluciones de forma iterativa para un problema. El procedimiento principal se puede describir de la siguiente manera: una vez que se inicializan todos los rastros y parámetros de feromonas, las hormigas construyen soluciones iterativamente hasta que se alcanza un criterio de detención. El procedimiento iterativo principal consta de dos pasos. En el primer paso, cada hormiga construye una solución de acuerdo con la regla de transición. Entonces se puede adoptar un procedimiento de búsqueda local para mejorar una o más soluciones. En el segundo paso, los valores de las feromonas se actualizan de acuerdo con una regla de actualización de feromonas. Un punto clave del ACH es construir soluciones candidatas, Ke et al. [19] proponen cuatro métodos: secuencial, determinista-concurrente, aleatorio-concurrente y simultáneo.

Los autores Vansteenwegen et al. [26] 2009, crearon un algoritmo compuesto donde primero construyen una solución y luego la mejoran con una combinación de búsquedas locales. Las búsquedas locales utilizadas son: *Swap*, *Replace*, *Move*, *Insert* y *2-Opt*. Una vez que la solución es mejorada, si es la mejor encontrada hasta el momento la guardan. Luego tienen un método para encontrar nuevas soluciones partiendo de una solución, quitándole destinos a las rutas y así poder explorar distintas opciones.

Souffriau et al. [22] en 2010 combinan un *Greedy Randomised Adative Search Procedure* (GRASP) con un *Path Relinking* (PR) para resolver el TOP. Su algoritmo a grandes rasgos consta de una iteración de cuatro partes que se detiene una vez que no encuentra una mejor solución luego de una determinada cantidad de iteraciones. El primer paso es el de la construcción de una solución utilizando GRASP. Luego se realiza una búsqueda local donde aplican *2-Opt*, *Swap*, *Insert* y *Replace*. En el tercer paso se hace el PR entre la solución construida y las soluciones del conjunto de elite. En el último paso se actualiza el conjunto de soluciones de elite. Si el conjunto de elite no esta completo aún, se inserta la mejor solución obtenida en la iteración. En caso de que el conjunto de elite esté completo, si la peor solución de elite es superada por la mejor solución de la iteración actual, se reemplazan.

Bouly et al. [5] 2010, idearon un *Algoritmo Memético* (AM) para resolver el TOP. Los AM son una combinación de un algoritmo genético y técnicas de búsqueda local. En su trabajo usan una codificación indirecta simple que denotan como un recorrido gigante, y un procedimiento de división óptima como el proceso de decodificación. Se dice que una codificación es indirecta si se necesita un procedimiento de decodificación para extraer soluciones de los cromosomas. El procedimiento de división que propusieron es específico del TOP. Sus resultados fueron muy buenos y en cinco instancias del benchmark de problemas superaron al mejor resultado obtenido en la literatura al momento de su publicación.

Dang et al. [11] 2011, proponen un *Algoritmo Memético basado en Optimización por Enjambre de Partículas* (AMOEP) para resolver el TOP. Su algoritmo AMOEP provee de soluciones de alta calidad para el TOP. El algoritmo está relativamente cerca del AM propuesto en Bouly et al. [5] y presenta los mismos componentes básicos, como la técnica de división de rutas, el inicializador de población y la vecindad de la búsqueda local. Sin embargo, el esquema global se ha modificado por una optimización de enjambre de partículas. La *Optimización por Enjambre de Partículas* (OEP) es una de las técnicas de inteligencia de enjambre con la idea básica de simular la inteligencia colectiva y el comportamiento social de los animales salvajes.

Ferreira et al. [12] 2014, implementan un *Algoritmo Genético* (AG) para resolver TOP. Su algoritmo consiste básicamente de tres componentes. El más elemental, llamado cromosoma, representa un conjunto de vehículos y sus rutas. El segundo componente es su proceso de evolución, responsable de hacer el cruzamiento y mutaciones dentro de una población. Su último componente es el algoritmo responsable de controlar el proceso evolutivo, asegurándose que los cromosomas sean válidos respecto de las restricciones de la instancia del TOP. En su proceso de cruzamiento se toman dos cromosomas y generan dos nuevos cromosomas utilizando aleatoriamente rutas de los cromosomas originales. Sus resultados fueron buenos, pero son superados por los resultados de Dang et al. [11] 2011 y Bouly et al. [5] 2010.

Lin y Yu [21] 2015, implementan un *Multi-start Simulated Annealing* (MSA) para resolver TOP. Un *Simulated Annealing* (SA) generalmente comienza con una solución generada al azar. En cada iteración el algoritmo selecciona una nueva solución de la vecindad de la solución actual. Si el valor de la función objetivo de la nueva solución supera al de la solución actual, la nueva solución se convierte en la actual y la búsqueda continua. Existe una baja probabilidad de que la nueva solución se convierta en la nueva solución actual a pesar de que su valor sea peor que el valor de la actual. El MSA que Lin et at. proponen, combina el SA con una estrategia *Multi-start Hill Climbing*.

Ke et al. [20] 2016, proponen un *Pareto Mimic Algorithim* (PMA). Usan un nuevo operador que llaman *mimic* para generar una nueva solución al imitar otra solución. Actualizan un conjunto de soluciones basándose en la eficiencia de *Pareto*, un concepto utilizado en economía e ingeniería. Tienen varios indicadores para comparar la eficacia de las soluciones, una solución puede no superar a otra en todos sus indicadores, es ahí cuando utilizan la eficiencia de *Pareto*.

Esos fueron los trabajos encontrados en mi investigación sobre trabajos previos. Hay algunos que implementan algoritmos genéticos pero ninguno que implemente un *Biased*

Random Key Generation Algorithm (BRKGA). De entre todos los trabajos mencionados decidí comparar mi resultados con los obtenidos por el AM de Bouly et al. [5], el $\mathrm{VNS}_{\mathrm{slow}}$ de Archetti et al. [1] y el $\mathrm{ACH}_{\mathrm{seq}}$ de Ke et al. [19] por que estos trabajos obtuvieron muy buenos resultados y plantean algoritmos diversos entre si. Además publicaron sus resultados con todas las instancias del benchmark de Tsiligirides y de Chao [18], permitiendo una comparación completa y objetiva con los resultados de este trabajo.

3. MODELO MATEMÁTICO

La formulación para el TOP donde el punto de inicio y fin son el mismo, ha sido presentada por Tang H. y Miller-Hooks E. [24]. Tal formulación puede ser extendida al caso donde el punto de inicio y fin pueden ser diferentes. A continuación presentaré la formulación matemática extendida del TOP propuesta por Ke L., Archetti C. y Feng Z. [19].

Sea un grafo completo $G = (V, E)$ donde $V = \{1, ..., n\}$ es el conjunto de vértices y $E = \{(i, j)|i, j \in V\}$ es el conjunto de ejes. Cada vértice i en V tiene un beneficio r_i. El punto de inicio es el vértice 1, el punto de fin es el vértice n y $r_1 = r_n = 0$. Todo eje (i, j) en E, tiene un costo no negativo c_{ij} asociado, donde c_{ij} es la distancia entre i y j. El TOP consiste en encontrar m caminos que comiencen en el vértice 1 y terminen en el vértice n de forma tal que el beneficio total de los vértices visitados sea maximizado. Cada vértice debe ser visitado a lo sumo una sola vez. Para cada vehículo, el tiempo total para visitar los vértices no puede superar un limite pre-especificado d_{max}. En el presente modelo matemático se asume que hay una proporcionalidad directa entre la distancia recorrida de un vehículo y el tiempo consumido por el vehículo. Por lo tanto no hay diferencia en considerar d_{max} como una distancia o un tiempo. Para evitar conflictos el valor es considerado como el valor de distancia máxima.

Sea $y_{ik} = 1 (i = 1, ..., m)$ si $\exists j \in V$ tal que el eje (i, j) es visitado por el vehículo k, sino $y_{ik} = 0$. Sea $x_{ijk} = 1$ $(1 \leq i < j \leq n; 1 \leq k \leq m)$ si el eje (i, j) es visitado por el vehículo k, sino $x_{ijk} = 0$. Sea U un subconjunto de V. Luego TOP puede ser descrito de la siguiente manera:

$$max \sum_{i=2}^{n-1} \sum_{k=1}^{m} r_i y_{ik} \tag{3.1}$$

sujeto a

$$\sum_{j=2}^{n} \sum_{k=1}^{m} x_{1jk} = \sum_{i=1}^{n-1} \sum_{k=1}^{m} x_{ink} = m \tag{3.2}$$

$$\sum_{i<j} x_{ijk} + \sum_{i>j} x_{jik} = 2y_{jk} \quad (i = 2, ..., n-1) \quad (k = 1, ..., m) \tag{3.3}$$

$$\sum_{k=1}^{m} y_{ik} \leq 1 \quad (i = 2, ..., n-1) \tag{3.4}$$

$$\sum_{i=1}^{n-1} \sum_{j>i} c_{ij} x_{ijk} \leq d_{max} \quad (k = 1, ..., m) \tag{3.5}$$

$$\sum_{i,j \in U \; i<j} x_{ijk} \leq |U| - 1 \quad (U \subset V \setminus \{1, n\}; 3 \leq |U| \leq n-2; k = 1, ..., m) \tag{3.6}$$

$$x_{ijk} \in \{0,1\} \quad (1 \leq i < j \leq n; k = 1, ..., m) \tag{3.7}$$

$$y_{1k} = y_{nk} = 1, \quad y_{ik} \in \{0,1\} \quad (i = 2, ..., n-1; k = 1, ..., m) \tag{3.8}$$

La restricción 3.2 asegura que todo vehículo comienza en el vértice 1 y termina en el vértice n. La restricción 3.3 asegura la conectividad de cada camino. La restricción 3.4 asegura que cada vértice (excepto el 1 y el n) debe ser visitado a los sumo una vez. La restricción 3.5 describe la limitación de distancia. La restricción 3.6 asegura que no hay ciclos. La restricción 3.7 y 3.8 establecen que todas las variables son enteras.

4. BIASED RANDOM KEY GENETIC ALGORITHM

4.1. Algoritmos Genéticos

Los *Genetic Algorithms* (GA) [14] están motivados en el concepto de supervivencia del más apto para encontrar soluciones óptimas o casi óptimas a los problemas de optimización combinatoria. Los GA hacen una analogía entre una solución y un individuo que pertenece a una población, donde cada individuo es un cromosoma que codifica una solución. Un cromosoma consiste en una cadena de genes. Cada gen, llamado alelo, toma un valor de algún alfabeto. Cada cromosoma tienen asociado un nivel de condición física que está correlacionado con el correspondiente valor de la función objetivo de la solución que codifica.

Los algoritmos genéticos manejan un conjunto de individuos que forman una población a lo largo de varias generaciones. En cada generación se crea una nueva población con individuos provenientes de tres fuentes distintas. La primer fuente de individuos es el conjunto de soluciones elite, es decir los individuos de mejor condición física. La segunda fuente de individuos son los individuos resultantes del *crossover*. El *crossover* es el método por el cual se obtiene un nuevo individuo a partir de otros dos individuos. Por último se completa la nueva generación con individuos mutantes. Los mutantes son individuos generados al azar con el fin de escapar de atrapamientos en mínimos locales y diversificar la población.

El concepto de supervivencia del más apto puede aparecer en los algoritmos genéticos de varias formas, dependiendo de la implementación en particular. Generalmente las soluciones de mayor aptitud física pasan directamente a la nueva generación. Además en el método del *crossover*, cuando los individuos son seleccionados para aparearse y producir descendencia, aquellos con mejor aptitud física tienen mayor probabilidad de ser elegidos para generar descendientes y mayor probabilidad de transmitir sus genes a sus hijos.

4.2. Random Key Genetic Algorithm

Los *Random Key Genetic Algorithm* (RKGA) fueron introducido por Bean [4]. En los RKGA, los cromosomas o individuos son representados por un vector de números reales generados al azar en el intervalo [0, 1]. El decodificador es el responsable de convertir un cromosoma en una solución del problema de optimización combinatoria, para el cual se calcula su valor objetivo o aptitud física. Los RKGA evolucionan una población de vectores de números reales aleatorios sobre una serie de iteraciones llamadas generaciones. La población inicial se compone de p_t vectores de claves aleatorias. Todos los vectores contienen la misma cantidad de claves aleatorias llamadas alelos. Cada alelo se genera independientemente al azar en el intervalo real [0, 1]. Después de obtener las soluciones utilizando el decodificador, se calcula la aptitud de cada individuo de la población, luego la población se divide en dos grupos de individuos. Se obtiene por un lado un pequeño grupo de individuos de élite. Estos son los individuos con los mejores valores de aptitud física. Denotamos el tamaño del conjunto de elite como p_e. Por el otro lado se conforma el grupo de todos los individuos restantes llamado el grupo de no-elite y su tamaño es

$p_t - p_e$. Todo individuo del conjunto de elite tiene mayor aptitud física que cualquier individuo del conjunto de no-elite y el tamaño del conjunto de elite es menor al tamaño del conjunto de no-elite, es decir $p_e < p_t - p_e$. Con el fin de evolucionar a la población, un RKGA utiliza una estrategia elitista ya que todos los individuos de élite de la generación k se copian sin cambios a la generación $k + 1$. Esta estrategia mantiene un seguimiento de las buenas soluciones encontradas durante las iteraciones del algoritmo que resulta en una heurística de mejora monotónica. En el RKGA los individuos mutantes se generan a partir de vectores de números reales aleatorios, de la misma manera que los individuos de la población inicial. Con la población p_e (elites) y la población p_m (mutantes), un conjunto adicional de tamaño $p_t - p_e - p_m$ es requerido para completar la generación $k + 1$. Los individuos que completan la nueva generación se obtienen mediante el proceso de *crossover*, donde los padres son elegidos al azar sobre toda la población y cada padre tiene la misma probabilidad de transmitir sus genes al individuo resultante.

4.3. Biased Random Key Genetic Algorithm

El *Biased Random Key Genetic Algorithm* (BRKGA), difiere del RKGA en la forma en que los padres son seleccionados para el *crossover*. En el BRKGA, cada elemento se genera combinando un elemento seleccionado al azar del conjunto de elite y el otro de la partición no-elite. En algunos casos el segundo padre se selecciona de toda la población mientras sean dos padres diferentes. Se permite la repetición en la selección de un padre, entonces un individuo puede producir más de un hijo. En la figura 4.2 se puede observar como funciona la evolución. Como el tamaño del conjunto de elite es menor al tamaño del conjunto de no-elite ($p_e < p_t - p_e$), la probabilidad de que un individuo de elite sea seleccionado para el apareamiento es mayor que la de un individuo no-elite. Por lo tanto un individuo de elite tiene una mayor probabilidad de transmitir sus genes a las generaciones futuras. Otro factor que contribuye a este fin es el *parameterized uniform crossover* (Spears y DeJong [23]), el mecanismo utilizado para implementar el apareamiento en BRKGA. Sea $\rho_e > 0,5$ la probabilidad de que un descendiente herede el alelo de su padre de elite, sea n el número de alelos de un individuo, para $i = 1, ..., n$ el i-ésimo alelo c_i del descendiente c, este alelo c_i toma el valor del i-ésimo alelo e_i del padre de elite e con una probabilidad ρ_e y el valor del e'_i del padre no-elite con probabilidad $1 - \rho_e$. Como $\rho_e > 0,5$, entones $\rho_e > 1 - \rho_e$, por lo tanto es más probable que el individuo resultante herede características del padre de élite que las del padre de no-élite. Si el decodificador se implementa de forma tal que cualquier vector de números reales se convierta en una solución válida, entonces el cromosoma resultante del *crossover* siempre decodifica en una solución válida del problema de optimización combinatoria.

En la figura 4.1 se puede observar como funciona el *parameterized uniform crossover*. El primer vector de alelos es de un individuo de elite, el segundo vector de alelos es de un individuo no-elite. Se decide que alelos tomará el individuo resultante utilizando un vector de números reales aleatorios del mismo tamaño que los vectores de alelos. Los números reales aleatorios toman un valor real en el intervalo [0,1]. En este ejemplo se utiliza un $\rho_e = 0{,}70$ incrementando la probabilidad de que el cromosoma resultante obtenga los alelos del cromosoma elite.

Fig. 4.1: Bias Crossover

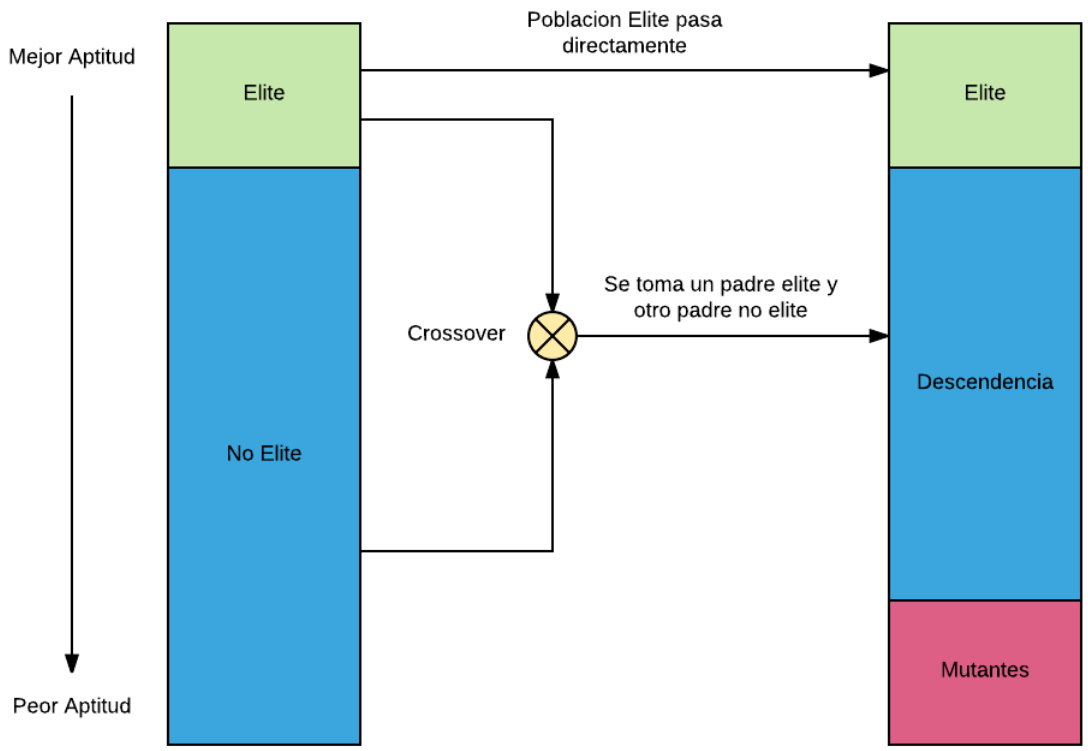
En la figura 4.1 podemos observar como el individuo resultante hereda el primer alelo del individuo de no-elite por que el número real aleatorio en la primera posición resulto mayor a 0,70

4.4. Decodificador del BRKGA

Una característica importante para mencionar del BRKGA es que el decodificador es el único modulo del algoritmo que requiere conocimiento del dominio del problema. El decodificador transforma un vector de números reales aleatorios en una solución válida del problema. El decodificador funciona como un adaptador, por lo tanto si hacemos un decodificador para otro problema podríamos reutilizar el modulo del BRKGA.

En el caso de mi implementación del BRKGA, entre cada generación se ejecuta una búsqueda local sobre las mejores soluciones de la población. Como estas búsquedas locales trabajan con una solución decodificada, luego de optimizar una solución se debe actualizar su vector de números reales. Es por eso que implementé un algoritmo capaz de convertir una solución s del problema en un vector v de números reales aleatorios tal que al decodificar el vector v se obtenga la solución s inicial. De este modo, una vez que se mejora una solución, se actualiza su vector de números reales aleatorios que lo representa y luego el BRKGA sigue su curso normal.

Fig. 4.2: Evolución de una Población



En la figura 4.2 los individuos son ordenados por su aptitud y marcados como elite y no-elite. Vemos como los individuos de elite pasan directamente a la siguiente generación. Un porcentaje pequeño de la nueva generación es conformado por individuos mutantes, generados al azar como la población inicial. Y complementa la nueva población los individuos generados por el proceso de *crossover* entre un individuo elite con un individuo no-elite.

5. DESARROLLO DEL ALGORITMO BRKGA CON BÚSQUEDA LOCAL PARA EL TOP

En el presente capítulo describiré en detalle la solución que implementé para el *Team Orienteering Problem*. La implementación se la puede dividir en 3 módulos importantes. Primero el decodificador, que como mencioné en el capítulo anterior, tiene la tarea de convertir un vector de números reales aleatorios en una solución válida del problema. El siguiente módulo sería el algoritmo BRKGA, cuya implementación podría hacerse con total independencia del problema a resolver. Por último las búsquedas locales aplicadas en cada nueva generación a los mejores individuos de la población.

En este capítulo mostraré los resultados parciales obtenidos a lo largo del desarrollo de mi implementación. A modo de analizar el rendimiento de una solución de forma simple y rápida creé un índice que llamo *índice de efectividad* (i_e). El i_e muestra que tan buena es la solución encontrada. Esto se hace comparando el beneficio de mi solución encontrada con el beneficio de la mejor solución previamente publicada para la misma instancia del problema. Se utilizaron los resultados de los trabajos previos mencionados en la revisión bibliográfica para crear el i_e. Es importante destacar que el i_e no es la función objetivo. La función objetivo es maximizar el beneficio a recolectar.

Se seleccionó un subconjunto del benchmark de instancias de problemas de Chao y Tsiligirides [18] para medir el progreso de mi desarrollo. Las seis instancias seleccionadas varían en cantidad de clientes y vehículos, esto es importante para que el análisis del rendimiento sea lo más objetivo posible. Luego de obtener una solución para cada una de estas instancias, se calcula el i_e que definí de la siguiente manera:

$$i_e(brkga_{v.x}, ins_n) = funcionObjetivo(brkga_{v.x}(ins_n))/bestPublishedValue(ins_n) \quad (5.1)$$

En la función (5.1) $brkga_{v.x}$ representa la versión x de mi $brkga$, ins_n representa la n-ésima instancia del benchmark de problemas. La función $brkga_{v.x}(ins_n)$ retorna la mejor solución que mi $brkga_{v.x}$ encontró para la instancia ins_n. La función $funcionObjetivo$ retorna la sumatoria de todos los beneficios recolectados en la solución que recibe como parámetro. La función $bestPublishedValue(ins_n)$ retorna el valor de la mejor solución de entre todos los trabajos previos publicados para la instancia que obtiene como parámetro. Analizar el rendimiento de mi implementación utilizando el i_e es muy sencillo. Si $i_e = 1$ para ins_n, entonces la solución encontrada es tan buena como la mejor solución encontrada hasta el momento, lo que significa que mi implementación es óptima para la instancia ins_n. Cuanto más cercano a 1 es i_e, más competitiva es mi implentación frente a los trabajos previos. Lamentablemente no existe instancia en el benchmark de problemas tal que mi implementación haya obtenido un resultado mayor al mejor beneficio obtenido para la misma instancia en algún trabajo previo, por lo tanto $i_e \leq 1$.

Se explicará en detalle el funcionamiento de los módulos que componen la implementación, incluyendo su pseudocódigo y los resultados parciales obtenidos luego de implementar tal módulo. El pseudocódigo utilizado sigue la sintaxis de c#, el lenguaje en el

Tab. 5.1: Instancias seleccionadas para el monitoreo del progreso del desarrollo.

Autor	Instancia	Nodos	Vehículos	d_{max}
Tsiligirides	p2.2.k	21	2	22.50
Tsiligirides	p2.3.g	21	3	10.70
Tsiligirides	p3.4.p	33	4	22.50
Chao	p5.3.x	66	3	40.00
Chao	p7.2.e	102	2	50.00
Chao	p7.4.t	102	4	100.00

cual implementé el desarrollo. El objetivo del pseudocódigo es facilitar el entendimiento del funcionamiento de los algoritmos que describiré.

Para la valuación de i_e elegí seis instancias del benchmark bien diversas entre sí. Tomé dos pequeñas, dos medianas y dos grandes, cuyas descripciones pueden observarse en la tabla 5.1. Como mencioné en el abstract, el benchmark de instancias se compone de siete conjuntos. Dentro de cada set, todas las instancias contienen los mismo clientes. Eso significa que tiene las mismas coordenadas en el eje cartesiano y el mismo beneficio. Dentro de un set las instancias se diferencian entre si por la cantidad de vehículos que poseen y el d_{max} de sus rutas. Eso es importante mencionarlo ya que ayuda a analizar los resultados parciales que obtuve sobre las seis instancias seleccionadas. Los nombres de las instancias nos dan información sobre el conjunto del benchmarck al cual pertenecen y la cantidad de vehículos que tienen. Por ejemplo la instancia *p2.3.g* pertenece al conjunto número 2 del benchmark y tiene 3 vehículos. La instancia *p2.2.k* tiene 2 vehículos y también pertenece al conjunto número 2 del benchmark, lo que significa que tiene los mismos clientes que *p2.3.g*.

A lo largo del desarrollo muestro algunos resultados parciales en distintas tablas. Estos resultados parciales serán los valores de las n ejecuciones que se realizaron sobre cada una de las instancias de monitoreo. El valor de n varía y será especificado para cada tabla. Esas tablas contendrán todas o algunas de las siguientes columnas:

Instancia	N/V/D	Config	T_{avg}	B_{max}	B_{min}	B_{avg}	i_{eMax}	i_{eAvg}	Best

Descripciones:

- *Instancia*: Nombre de la instancia utilizada.
- *N/V/D*: Cantidad de **N**odos / Cantidad de **V**ehículos / **D**istancia máxima de la ruta del vehículo.
- *Config*: La configuración utilizada de mi BRKGA al ejecutar la prueba. Es un código que sintetiza la configuración global del algoritmo, explicado en detalle más adelante (ver sección 5.2.1).
- T_{avg}: El **T**iempo promedio en milisegundos de la ejecución del algoritmo para la instancia mencionada sobre las n ejecuciones.

- B_{max}: El **B**eneficio máximo que obtuve para la instancia mencionada sobre las n ejecuciones.
- B_{min}: El **B**eneficio mínimo que obtuve para la instancia mencionada sobre las n ejecuciones.
- B_{avg}: El **B**eneficio promedio que obtuve para la instancia mencionada sobre las n ejecuciones.
- i_{eMax}: Indice de efectividad máximo. Utiliza mi beneficio máximo obtenido para la instancia mencionada.
- i_{eAvg}: Indice de efectividad promedio. Utiliza mi beneficio promedio obtenido para la instancia mencionada.
- *Best*: Máximo beneficio obtenido por algún trabajo previo sobre la misma instancia mencionada.

5.1. Decodificador

El decodificador debe generar una solución válida del problema dado un vector de claves aleatorios y conociendo la instancia del problema (vehículos disponibles, clientes, d_{max}, etc). Con tal objetivo construye una solución asignando clientes a las rutas de los vehículos disponibles respetando d_{max}. De acá en adelante llamaré clientes a todos los nodos que tienen un beneficio mayor a cero. Es decir, todos los nodos excepto el nodo de inicio y fin de recorrido. El orden en que toma los clientes a asignar es clave y determina la solución resultante. Tal orden es determinado por el vector de claves aleatorias. Por lo tanto el vector tendrá una longitud equivalente a la cantidad de clientes del problema.

Propuse dos decodificadores, uno al cual llamé *Decodificador Simple* y al otro lo llamé *Decodificador Goloso*. Ambos decodificadores tienen sus ventajas y desventajas.

5.1.1. Orden en que los clientes se intentan agregar a las rutas.

Como dije, dado una instancia de un problema con n clientes y un vector de claves aleatorias del mismo tamaño n, un decodificador genera una solución válida de un problema. En mi implementación, modelé una clave aleatoria con el objeto *RandomKey*. Un *RandomKey* tiene dos propiedades, un entero aleatorio llamado *Key* y otro entero llamado *ClientId* que siempre toma el valor de un identificador de uno de los clientes de la instancia. Podemos ver el objeto *RandomKey* en el pseudocódigo 5.1.

Pseudocódigo 5.1: Objeto *RandomKey*.

```
public class RandomKey
{
  public int Key { get; private set; }
  public int ClientId { get; private set; }
}
```

El propósito de *ClientId* es asociar un *RandomKey* con un cliente. Existe un vector de clientes en el Mapa del problema, a cada cliente se le asigna un identificador que es un número entero en el intervalo $[1, \#clientes]$. Luego para un vector de *RandomKeys* de tamaño *#clientes* no existen dos *RandomKeys* con mismo valor de *ClientId* y todos los *ClientId* se encuentran en el intervalo $[1, \#clientes]$. De esta forma cada *RandomKey* siempre se asocia con un solo cliente. Luego de asociar cada cliente con su correspondiente *RandomKey*, se los ordena de forma ascendente por el *Key* del *RandomKey* con el cual se asoció. Es decir, ordena los clientes al azar ya que el valor de los enteros en el vector *RandomKeys* es aleatorio. Este es el orden por el cual se tomaran los clientes para ser asignados a los vehículos. El pseudocódigo 5.2 muestra como se obtienen los clientes ordenados dado un vector de *RandomKeys*. En el figura 5.1 se puede observar un vector de *RandomKeys* en su estado inicial y luego ordenado por su campo *Key* mostrando el orden en que se tomarán los clientes.

Pseudocódigo 5.2: Dada una lista de *RandomKeys*, se obtienen los clientes ordenados.

```
public List<Client> GetOrderedClients(List<RandomKey> randomKeys)
{
  var orderedKeys = randomKeys.OrderBy(r => r.Key)
  return orderedKeys.Select(r => Map.Clients[r.ClientId]);
}
```

Fig. 5.1: Ejemplo de como el vector de *RandomKeys* determina el orden de los clientes.

Vector de RandomKeys recien inicializado:

Key	27	13	79	45	21	7	98	54
ClientId	1	2	3	4	5	6	7	8

Vector de RandomKeys ordenado por propiedad Key:

Key	7	13	21	27	45	54	79	98
ClientId	6	2	5	1	4	8	3	7

Orden en que se consideraran los clientes a los vehiculos: 6, 2, 5, 1, 4, 8, 3 y 7

5.1.2. Decodificador Simple

El decodificador simple recibe como parámetro el vector de *RandomKeys* y lo primero que hace es obtener los clientes ordenados como describimos anteriormente. Luego por cada vehículo, si el siguiente cliente se puede incluir en la ruta se incluye sino considera que la ruta esta completa y pasa al siguiente vehículo. Es decir, ordena los clientes al azar utilizando el vector de *RandomKeys* y los asigna en orden a los vehículos mientras no estén saturados. Estos se puede ver en detalle en el pseudocódigo 5.3.

Pseudocódigo 5.3: Función *Decode* del decodificador simple.

```
public Solution Decode(List<RandomKey> randomKeys, ProblemInfo pi)
{
  var clients = GetOrderedClients(randomKeys);
  var vehicles = pi.GetVehicles();
  var iv = 0;
  var ic = 0;
  do
  {
    if(vehicles[iv].CanVisit(clients[ic]))
    {
      vehicles[iv].AddClient(clients[ic]);
      ic++;
    }
    else
    {
      iv++;
    }

  } while(iv < vehicles.Length && ic < clients.Length)
  var solution = pi.InstanceSolution(vehicles);
  return problem;
}
```

Un cliente c_i se puede agregar a la ruta si al agregarlo, la ruta no supera su distancia máxima permitida. Sean v vehículo, d_{max} la distancia máxima de v, d_{act} la distancia actual de la ruta de v, n_f el nodo final de la ruta, c_u el último cliente agregado a la ruta de v y c_i cliente que se intenta agregar a la ruta de v. Si aún no se insertaron clientes en la ruta, $c_u = n_i$ donde n_i representa el nodo inicial de la ruta. Como podemos observar en el pseudocódigo 5.3, creé un método llamado *CanVisit*, que modela la fórmula 5.2. *CanVisit* retorna *true* cuando c_i se puede agregar a la ruta y *false* en caso contrario.

El método *CanVisit* retorna *true* cuando la siguiente fórmula es válida:

$$d_{act} + distancia(c_u, c_i) + distancia(c_i, n_f) - distancia(c_u, n_f) \leq d_{max} \tag{5.2}$$

Una vez que terminé de implementar el decodificador simple, analicé el rendimiento de las soluciones generadas a partir de este decodificador. Hice este análisis para saber que tan buena sería la población inicial de mi algoritmo BRKGA cuando se utiliza el decodificador simple. Para realizar este análisis creé aleatoriamente 200 vectores de *RandomKeys* que el decodificador simple convirtió en 200 soluciones válidas del problema. Sobre estas 200 soluciones calculé el beneficio máximo, promedio y mínimo, y sus índices de efectividad promedio y máximo. Esto se realizó para cada una de las seis instancias del benchmark seleccionadas anteriormente 5.1. Podemos observar los resultados en la tabla 5.2.

En los resultados de la tabla 5.2 vemos que para instancias pequeñas los resultados son mejores. En la instancia *p2.3.g* una de las 200 soluciones quedo muy cercana a la mejor solucion conocida, obteniendo un $i_{eMax} = 0{,}97$. Esta instancia tiene los mismos clientes que

Tab. 5.2: Resultados de las 200 soluciones generadas por el decodificar simple.

Instancia	N/V/D	B_{min}	B_{avg}	B_{max}	i_{eAvg}	i_{eMax}	$Best$
p2.2.k	21/2/22.50	50	101	170	0.37	0.62	275
p2.3.g	21/3/10.70	45	84	140	0.58	0.97	145
p3.4.p	33/4/22.50	100	168	270	0.30	0.48	560
p5.3.x	66/3/40.00	195	301	425	0.19	0.27	1555
p7.2.e	102/2/50.00	8	39	113	0.13	0.39	290
p7.4.t	102/4/100.00	38	114	238	0.11	0.22	1077

la instancia *p2.2.k* pero como el d_{max} de *p2.3.g* es prácticamente la mitad que el de *p2.2.k* luego la cantidad de combinaciones de rutas diferentes disminuye considerablemente. Es por eso que obtuve mejores resultados de *p2.3.g*. Por otro lado podemos observar i_{eAvg} disminuye a medida que la instancia tiene mayor número de clientes. Los peores resultados los obtuvo la instancia *p7.4.t*, la instancia de mayor cantidad de clientes y mayor d_{max}.

5.1.3. Características y debilidades del decodificador simple

Este decodificador es simple y rápido, su orden de complejidad es de $O(\#clientes + \#vehículos)$. En la práctica nunca se llega a visitar a todos los clientes ya que cambia de vehículo en cuanto encontró un cliente que no logró insertar en su ruta. Por lo tanto en la práctica nunca llega al orden de complejidad mencionado. Esto es una gran ventaja ya que en cada iteración del BRKGA se va a decodificar una cantidad $\#Población$ de veces. Luego una decodificación rápida nos permitirá mayor cantidad de generaciones.

Una característica menos relevante es el orden en que quedan los clientes asignados en los vehículos al ver el vector de *RandomKeys*. Sea v el vector de clientes ordenados por un vector de *RandomKeys*. Existen $m + 1$ índices $i_0 = 0, i_1, i_2, .., i_m$ donde m es la cantidad de vehículos y $0 \leq i_j \leq \#clientes$ tales que el vehículo j incluye en su recorrido a todos los clientes del subvector $v[i_{j-1}, i_j - 1]$. Luego todos los clientes en el subvector $v[i_m, v.Length-1]$ son clientes no alcanzados por la solución. En otras palabras, los clientes quedan agrupados por vehículo cuando los vemos en el vector ordenado. Esto puede verse en la figura 5.2.

Fig. 5.2: Posible distribución de clientes utilizando el decodificador simple para el vector de *RandomKeys* de ejemplo. La primer ruta visita primero al cliente 6 y luego al 2. Como no pudo incluir al cliente 5, se cerró la ruta del primer vehículo y siguió con el próximo vehículo disponible. La segunda ruta visita al cliente 5 y luego al cliente 1. Como no pudo visitar al cliente 4 por la limitación de distancia, no intento agregar a los siguientes clientes.

Key	7	13	21	27	45	54	79	89
ClientId	6	2	5	1	4	8	3	7

Vehículo 1: 6 -> 2
Vehículo 2: 5 -> 1

Un problema que tiene este decodificar es en la existencia de un cliente inalcanzable. Un cliente inalcanzable es aquel que no puede insertarse en una ruta aún cuando la ruta no contiene ningún otro cliente. En la función 5.3 podemos distinguir cuando un cliente no se lo puede agregar en ninguna ruta.

Función para evaluar si un cliente es inalcanzable.

$$distancia(n_i, c) + distancia(c, n_f) > d_{max} \tag{5.3}$$

Si utilizamos el decodificador simple y tenemos un cliente inalcanzable en el mapa, existe el escenario en el cual hay soluciones de la población donde todos los vehículos tienen sus rutas vacías. Supongamos que existe un cliente inalcanzable y que es el primer cliente que se intenta agregar en la ruta del primer vehículo. Como el cliente es inalcanzable no entra en la ruta, entonces se considera que el vehículo tiene la ruta completa y se pasa al siguiente vehículo dejando su ruta vacía. Esto se repite con todos los vehículos ya que tienen el mismo valor de d_{max}. La solución óptima a este problema es filtrar todos los clientes inalcanzables previo a la ejecución del BRKGA utilizando un método que modele la función 5.3. Haciendo esto no solo evitamos el escenario de rutas vacías, también reducimos el tamaño del problema antes de comenzar a resolverlo.

Como el decodificador simple cambia de vehículo al primer intento fallido de expandir su ruta, las soluciones que genera tienen rutas muy pequeñas. Seguramente existen algunos clientes que podrían insertarse a la ruta del vehículo actual. Es por este motivo implementé el *Decodificador Goloso*.

5.1.4. Decodificador Goloso

El decodificador goloso en principio funciona igual que el decodificador simple hasta que llega a un cliente que no puede agregar a la ruta de un vehículo determinado. En este caso, en vez de pasar a trabajar con el siguiente vehículo disponible, intenta agregar al siguiente cliente y así sucesivamente hasta que no hay más clientes con los cuales intentar agregar al vehículo actual. Después, al pasar al siguiente vehículo intenta con los clientes no asignados a los vehículos anteriores y siempre respetando el orden de los clientes asignado por el vector de *RandomKeys*. Podemos ver en detalle como implementé el método *Decode* del decodificador goloso en el pseudocódigo 5.4.

Pseudocódigo 5.4: Función *Decode* del decodificador goloso.

```
public Solution Decode(List<RandomKey> randomKeys, ProblemInfo pi)
{
  var clients = GetOrderedClients(randomKeys);
  var cIterator = new Iterator(clients);
  var vehicles = pi.GetVehicles();
  var iv = 0;
  while(iv < vehicles.Length)
  {
    var currentClient = cIterator.Next;
    while(currentClient != null)
    {
      if(vehicles[iv].CanVisit(currentClient))
      {
        vehicles[iv].AddClient(currentClient);
        cIterator.Remove(currentClient);
      }
      currentClient = cIterator.Next;
    }
    cIterator.ToStartingPosition;
  }
  var solution = pi.InstanceSolution(vehicles);
  return problem;
}
```

Como podemos observar en el pseudocódigo 5.4, su complejidad es $0(\#clientes * \#vehículos)$. El método *Decode* es usado tantas veces a lo largo del BRKGA que este pequeño aumento en su complejidad algorítmica tiene un impacto visible en el tiempo de ejecución total. Por otro lado, utilizar el decodificador goloso mejora el beneficio de las soluciones generadas respecto de las soluciones generadas por el decodificador simple. Esa es la compensación que tenemos entre el decodificador simple y el goloso.

Otra característica que podemos mencionar sobre el decodificador goloso es que al observar el vector ordenado de *RandomKeys*, ya no tenemos a los clientes de forma continua según su vehículo asignado como sucedía con el decodificador simple. Esto puede verse en la figura 5.3.

Le hice un análisis de rendimiento al decodificador goloso del mismo modo que lo hice con el decodificador simple. Utilicé los mismos 200 vectores de *RandomKeys* para cada una de las mismas seis instancias de problemas y el decodificador goloso creó 200 soluciones válidas.

Como podemos ver en la tabla 5.3, todos los resultados promedio, mínimo y máximo mejoran considerablemente respecto de los resultados obtenidos con el decodificador simple (tabla 5.2). Tal es así, que tanto el i_{eAvg} como el i_{eMax} en algunos casos es mayor al doble de lo obtenido con en el decodificador simple. También se vuelve a observar como disminuye i_{eAvg} a medida que crece el tamaño de la instancia del problema.

Fig. 5.3: Posible distribución de clientes utilizando el decodificador goloso para el vector de RandomKeys de ejemplo. La primer ruta visita a los clientes 6, 2 y por último al 8. El decodificador goloso no cerró la ruta del primer vehículo al no poder incluir al cliente 5, en cambio intenta con el resto de los clientes aún no visitados manteniendo el orden y logra insertar al cliente 8. De un modo similar, sucede con la segunda ruta al no poder incluir al cliente 4. Con el cliente 8 no lo intenta por que esta asignado a la primer ruta. Intenta con éxito insertar al cliente 3 y por último falla con el cliente 7.

Key	7	13	21	27	45	54	79	89
ClientId	6	2	5	1	4	8	3	7

Vehículo 1: 6 -> 2 -> 8
Vehículo 2: 5 -> 1 -> 3

Tab. 5.3: Resultados de las 200 soluciones generadas por el decodificar goloso.

Instancia	N/V/D	B_{min}	B_{avg}	B_{max}	i_{eAvg}	i_{eMax}	$Best$
p2.2.k	21/2/22.50	105	166	230	0.60	0.84	275
p2.3.g	21/3/10.70	90	122	145	0.84	1.00	145
p3.4.p	33/4/22.50	190	285	400	0.51	0.71	560
p5.3.x	66/3/40.00	305	412	560	0.26	0.36	1555
p7.2.e	102/2/50.00	37	95	171	0.33	0.59	290
p7.4.t	102/4/100.00	165	273	439	0.25	0.41	1077

5.2. Biased Random Key Genetic Algorithms

En una primera instancia se implementa un BRKGA estándar. Dado una instancia de un problema, primero se genera la población inicial. Luego mientras no se cumpla la condición de parada, evolucionamos la población. Es decir se crea una nueva generación de soluciones a partir de la generación anterior como se explica en el capítulo 4.3. En el pseudocódigo 5.5 podemos ver un punto de vista macro del algoritmo BRKGA implementado.

Pseudocódigo 5.5: Función *RunBrkga*, vista macro de mi implementación.

```
public Solution RunBrkga(ProblemManager problemManager)
{
  var population = InitializePopulation();

  while (!StoppingRuleFulfilled(population))
    EvolvePopulation(population);

  return GetMostProfitableSolution(population);
}
```

5.2.1. Configuración

A modo de poder probar distintas configuraciones del BRKGA, creé el objeto *Configuration* que instancia todas las variables que pueden impactan en el resultado final del BRKGA. Este objeto es esencial para ajustar mi implementación de una forma rápida y ordenada. Al centralizar todas las variables que podrían impactar en resultado final, gané mucho tiempo al probar variaciones de mi implementación. Además, al estar centralizada toda la información variable, se obtiene una lectura veloz del BRKGA que se esta probando. En otras palabras incrementé mi capacidad de monitoreo y control del desarrollo. El pseudocódigo 5.6 muestra todas las propiedades configurables de mi desarrollo.

Pseudocódigo 5.6: Objeto *Configuration* donde se encuentran todas las variables importantes del BRKGA.

```
public class Configuration
{
  public string Description { get; }
  public int MinIterations { get; set; }
  public int MinNoChanges { get; set; }
  public int PopulationSize { get; set; }
  public decimal ElitePercentage { get; set; }
  public decimal MutantPercentage { get; set; }
  public decimal EliteGenChance { get; set; }
  public List<ILocalSearch> LocalSearches { get; set; }
  public int ApplyLocalSearchesToTop { get; set; }
  public DecoderEnum DecoderType { get; set; }
  private void SetDescription();
}
```

5.2.2. Descripción y codificación de las propiedades configurables

Como mencioné, uno de los beneficios de tener todas las variables configurables en un solo objeto es poder leer rápidamente que BRKGA estoy probando. A continuación explico las propiedades del objeto Configuración:

- **Description**: Describe la instancia del objeto *Configuration* utilizando la clave y valor del resto de las propiedades.
- **MinIterations**: Clave **MI**. Valor entero utilizado en la función de corte. Cantidad mínima de generaciones que deben completar para cortar el BRKGA.
- **MinNoChanges**: Clave **MNC**. Valor entero utilizado en la función de corte. Cantidad mínima de generaciones sin que aparezca una nueva mejor solución requerido para cortar el BRKGA.
- **PopulationSize**: Clave **PS**. Valor entero que denota el tamaño de la población.
- **ElitePercentage**: Clave **EP**. Valor decimal en el intervalo $(0, 1)$ que determina el tamaño de la población elite. Es un porcentaje de la población total.
- **MutantPercentage**: Clave **MP**. Valor decimal en el intervalo $(0, 1)$ que determina el tamaño mínimo de la población mutante. Es un porcentaje de la población total.

- **EliteGenChance**: Clave **EGC**. Valor decimal en el intervalo $(0, 1)$ que determina la probabilidad que tiene el alelo del padre de elite para transmitirse al individuo resultante del crossover.
- **LocalSearches**: Clave **LS**. Secuencia de algoritmos de búsquedas locales que se le aplicaran a la mejor solución de cada generación:
  - **Swap**: Valor **S**.
  - **Insert**: Valor **I**.
  - **2-Opt**: Valor **O**.
  - **Replace Simple**: Valor **Rs**.
  - **Replace Multiple**: Valor **Rm**.
- **ApplyLocalSearchesToTop**: Clave **TOP**. Valor entero que denota la cantidad de soluciones a las cuales se les aplicaran las búsquedas locales.
- **DecoderType**: Clave **D**. Es una enumeración que determina el decodificador que se va a utilizar.
  - **Simple**: Valor **S**.
  - **Goloso**: Valor **G**.

El método *SetDescription()* toma las tuplas de clave y valor del restos de las propiedades del objeto *Configuración* y los concatena intercalados por un separador creando un *string*. Podemos observar como lo hace en el pseudocódigo 5.7.

Pseudocódigo 5.7: Método que instancia la propiedad *Description*.

```
public void SetDescription()
{
  var prop = Properties.Where(x => x.Name != "Description");
  var claveValores = prop.Select(p => p.Clave + "." + p.Valor);
  Description = string.Join(";", claveValores);
}
```

Entonces leyendo la propiedad *Description* podemos ver como esta configurado el BRKGA. Por ejemplo si dice:

"MI.200;MNC.10;PS.100;EP.0,3;MP.0,1;EGC.0,7;LS.ISRsORm;TOP.2;D.G"

- MinIterations: 200
- MinNoChanges: 10
- PopulationSize: 100
- ElitePercentage: 0,3
- MutantPercentage: 0,1
- EliteGenChance: 0,7
- LocalSearches: ISRsORm. Es la Secuencia: Insert, Swap, Replace Simple, 2-Opt y Replace Multiple.
- ApplyLocalSearchesToTop: 2
- DecoderType: Decodificador Goloso

5.2.3. Inicialización de la Población

La población inicial se crea llamando al método *AddMutants* que se explica en detalle más adelante. La inicialización de la población utiliza el mismo método que para crear soluciones mutantes ya que en ambos casos se requieren soluciones aleatorias. En el pseudocódigo 5.8 vemos como el método *InitializePopulation* devuelve las soluciones generadas por *AddMutants*. El método *AddMutants* toma como único parámetro el *PopulationSize* que determina cuantos individuos deben ser creados.

Pseudocódigo 5.8: Generección de la población inicial.

```
public Population InitializePopulation()
{
  var population = new Population();
  population.AddMutants(PopulationSize);
  return population;
}
```

5.2.4. Condición de parada

En una primera instancia de mi implementación la condición de parada era simple, el bucle terminaba cuando iteraba *MinIterations* veces. Es decir que el bucle principal cortaba luego de evolucionar la población *MinIterations* veces. Después de analizar varias ejecuciones, noté que a veces la mejor solución de la población se había encontrado en las últimas evoluciones, por lo tanto agregue una condición de corte adicional: la mejor solución no debe haberse encontrado durante las últimas *MinNoChanges* generaciones. Por lo tanto el algoritmo para cuando se satisfacen ambas condiciones a la vez. Podemos ver ambas condiciones de parada en el pseudocódigo 5.9.

Pseudocódigo 5.9: Condición de parada del BRKGA.

```
public bool StoppingRuleFulfilled()
{
  return GenerationNum >= MinIterations && NoChanges();
}
private bool NoChanges()
{
  var currentProfit = CurrentBestSolution.GetProfit();
  return LastProfits.All(p => p == currentProfit);
}
```

El *LastProfits* del pseudocódigo 5.9, es una cola de tamaño *MinNoChanges* que contiene los beneficios de las mejores soluciones de las últimas generaciones. Si todos son iguales al beneficio de la mejor solución actual significa que no hubo cambios en las últimas *MinNoChanges* generaciones.

5.2.5. Evolución de la población

Se toma la población y se ordenan sus individuos de forma descendente según su beneficio calculado con la función objetivo. Los mejores individuos pasan a ser parte de la población de elite y el resto de la población no-elite. El tamaño de la población de elite depende de la propiedad *ElitePercentage* del objeto *Configuration*. Luego se generan individuos mutantes, su cantidad es un porcentaje de la población total seteado por la propiedad *MutantPercentage*. Finalmente se completa la nueva generación emparentando individuos de la población de elite con individuos de la población de no-elite. Los padres son elegidos al azar y el proceso de apareamiento se realiza como se describe en el la sección 4.3. Durante el apareamiento, no es tan extraño que se genere una solución idéntica a otra ya existente en la población. De modo de no repetir soluciones, antes de insertar el individuo resultante se verifica que no exista otra solución idéntica en la nueva generación. De ser idéntica a otra solución, se genera una solución aleatoria del mismo modo que se generan los mutantes. En el pseudocódigo 5.10 podemos ver como evoluciona una población.

Pseudocódigo 5.10: Evolución de la población.

```
public Population Evolve(Population population)
{
  var ordPopulation = population.GetOrderByMostProfitable();
  var elites = ordPopulation.Take(EliteSize);
  var nonElites = ordPopulation.Skip(EliteSize).Take(NonEliteSize);
  var evolvedPopulation = new Population(elites);
  evolvedPopulation.AddMutants(MutatansSize);
  while (evolvedPopulation.Size() < PopulationSize)
  {
    var anElite = GetRandomItem(elites);
    var aNoneElite = GetRandomItem(nonElites);
    var childSolution = Mate(anElite, aNoneElite);
    if (evolvedPopulation.Any(x => x.Equals(childSolution)))
      evolvedPopulation.AddMutants(1);
    else
      evolvedPopulation.Add(childSolution);
  }
  return evolvedPopulation;
}
```

Un proceso clave de la evolución es el *crossover*. Se crea un vector de *RandomKeys* a partir de los vectores de *RandomKeys* de un individuo elite y otro no-elite. Se determina que alelos recibirá el individuo resultante a partir de un vector de decimales aleatorios en el intervalo [0, 1] y su comparación con la propiedad *EliteGenChance*. En el pseudocódigo 5.11 observamos el proceso de apareamiento como lo describí en la sección 4.3 y se puede observar en la figura 4.1. Este proceso es independiente del problema que se esta resolviendo.

Pseudocódigo 5.11: Crossover de dos individuos.

```
private Solution Mate(Solution eliteP, Solution nonEliteP)
{
  var childRandomKeys = new List<RandomKey>();
  for (var index = 0; index < eliteP.RandomKeys.Count; index ++)
  {
    int key = 0;
    if(Random.NextDecimal(0, 1) >= EliteGenChance)
      key = eliteP.RandomKeys[index].Key;
    else
      key = nonEliteP.RandomKeys[index].Key;
    var randomKey = new RandomKey(key, index);
    childRandomKeys.Add(randomKey);
  }
  return Decoder.Decode(childRandomKeys, ProblemInfo);
}
```

Otro proceso muy importante de la evolución es la generación de los individuos mutantes. Cada uno es generado al azar a partir de un vector de *RandomKeys*, del mismo modo que son generados los individuos de la población inicial como mencioné en 5.2.3.

El pseudocódigo 5.12 muestra como se crean los individuos mutante. El método *AddMutants* toma como parámetro la cantidad de individuos que debe generar y por cada uno que debe generar crea un vector de *RandomKeys*. Si la solución resultante del vector de *RandomKeys* decodificado es igual a alguna de las soluciones existentes, deshace el vector de *RandomKeys* y genera uno nuevo. En caso contrario agrega la solución a la población de soluciones.

Pseudocódigo 5.12: Crossover de dos individuos.

```
private void AddMutants(int amount)
{
  var solutions = new List<Solution>();
  for(var j = 0; j < amount; j++)
  {
    var randomKeys = new List<RandomKey>();
    for(i = 0; i < ProblemInfo.Clients.Length; i++)
    {
      var key = Random.Next(1000);
      var randomKey = new RandomKey(key, index);
      randomKeys.Add(randomKey)
    }
    var solution = Decoder.Decode(randomKeys, ProblemInfo);
    if (Solutions.Any(x => x.Equals(solution)))
      j--;
    else
      Solutions.Add(solution);
  }
}
```

5.2.6. Hash de un individuo

En una primera instancia se insertaban los individuos sin verificar la existencia de otro individuo idéntico en la población. Dada una población de soluciones no repetidas, la probabilidad de generar una solución existente al evolucionar la población es baja. Aún así, una vez que se genera una solución ya existente en la población, la probabilidad de que se genere otra copia más aumenta considerablemente. Esto se debe a que bajó la diversidad dentro de la población. A partir de la existencia de duplicados existe la posibilidad de utilizar padres idénticos. Si además la solución repetida se encuentra dentro del subconjunto de elite, la probabilidad aumenta aún más. Esto genera un efecto avalancha, donde la cantidad de individuos duplicados aumenta en cada evolución de la población. He llegado a obtener una población constituida de una única solución excepto por las soluciones mutantes. La existencia de duplicados reduce la cantidad de soluciones diferentes exploradas, por lo tanto reduce la probabilidad de encontrar una nueva mejor solución por generación. Además, si admitimos duplicados, el algoritmo repite cómputos en donde obtiene los mismos resultados.

Como mencioné en el proceso de evolución, mi implementación verifica que no se inserten individuos duplicadas en la población. Para ver si dos individuos son iguales se compara el hash de cada individuo. Implementé dos formas de obtener un hash a partir de un individuo.

El primer método que implementé requiere conocer solamente el vector de *RandomKeys* del individuo. El vector de *RandomKeys* determina el orden de los clientes y para un orden de clientes determinado los decodificadores siempre generan la misma solución. Por lo tanto, para dos vectores de *RandomKeys* que generen el mismo orden de clientes se obtienen la misma solución. Gracias a eso no es necesario conocer la solución que el vector de *RandomKeys* codifica. En el pseudocódigo 5.13 podemos ver como se crea el hash de un individuo a partir de su vector de *RandomKeys*. Para calcular el hash se ordena el vector de *RandomKeys* por su campo *Key*, después se toman los *ClientId* y se los concatenan con un símbolo separador. Este método tiene un orden de complejidad de $O(clientes * log(clientes))$. En la figura 5.4 podemos ver de un vector de *RandomKeys* y el hash resultante.

Pseudocódigo 5.13: Generación del hash de un individuo.

```
private string hash;
public string GetHashFromRandomKeys()
{
  if (!string.IsNullOrEmpty(hash))
    return hash;

  var ork = RandomKeys.OrderBy(r => r.Key)
  hash = string.Join("@", ork.Select(k => k.ClientId));
  return hash;
}
```

Fig. 5.4: Vector de *RandomKeys* ordenado y el hash resultante.

Key	7	13	21	27	45	54	79	89
PositionIndex	6	2	5	1	4	8	3	7

GetHashFromRandomKeys: 6@2@5@1@4@8@3@7

El segundo método es más preciso pero requiere conocer la solución en la que el vector de *RandomKeys* decodifica. En el pseudocódigo 5.14 podemos ver como se crea el hash de un individuo a partir de la solución a la cual decodifica su vector de *RandomKeys*. Para calcular el hash, se ordenan los vehículos de la solución por el id del primer cliente que visitan. Luego se concatenan los hashs de cada vehículo separados por un símbolo separador. El hash de cada vehículo es la concatenación de los clientes que visita en orden y separados por otro símbolo separador (ver pseudocódigo 5.15). Este método tiene un orden de complejidad $O(vehiculos * log(vehiculos) * clientes)$. En la figura 5.5 podemos ver un vector de *RandomKeys* que codifica en una solución con dos vehículos cuyos clientes que visitan están coloreados y también podemos ver el hash resultante.

Pseudocódigo 5.14: Generación del hash de un individuo.

```
private string hash;
public string GetHashFromSolution()
{
  if (!string.IsNullOrEmpty(hash))
    return hash;

  var ov = Solution.Vehicles.OrderBy(v => v.FirstClient)
  hash = string.Join("#", ov.Select(v => v.GetVehicleHash()));
  return hash;
}
```

Pseudocódigo 5.15: Generación del hash de una ruta.

```
public string GetVehicleHash()
{
  return string.Join("@", Route.Select(x => x.ClientId));
}
```

Fig. 5.5: Solución en la que decodifica un vector de *RandomKeys* y el hash resultante.

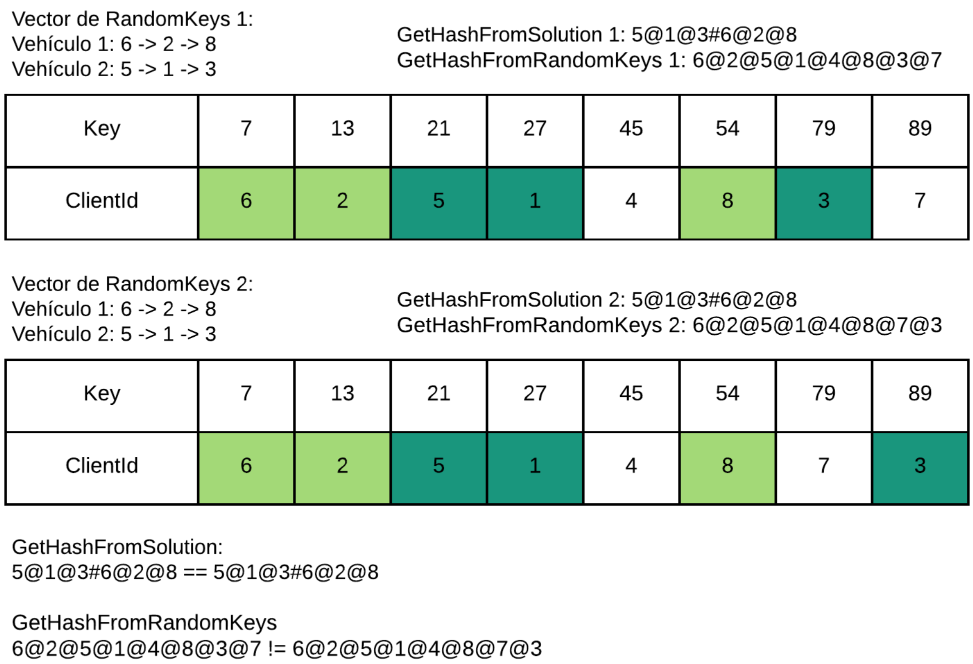
Key	7	13	21	27	45	54	79	89
ClientId	6	2	5	1	4	8	3	7

GetHashFromSolution: 5@1@3#6@2@8

El primer método tiene la ventaja de no tener que conocer la solución en la que decodifica el vector de *RandomKeys*, por lo tanto es independiente del problema que se esta resolviendo. La desventaja que tiene es que existen casos donde no detecta que dos individuos son iguales, cuando lo son. Por ejemplo si intercambiamos los ids de los clientes de las últimas dos posiciones del vector de *RandomKeys* de la figura 5.5 obtenemos dos hashs distintos pero ambas vectores de *RandomKeys* decodifican en la misma solución. En la figura 5.6, podemos ver un ejemplo donde dos *RandomKeys* diferentes decodifican en la misma solución, luego el método *GetHashFromRandomKeys* falla en detectar que ambos individuos son el iguales. La desventaja del segundo método es que requiere conocer la solución, luego conoce el dominio del problema que se resuelve. En mi implementación decidí utilizar el segundo método ya que es mas preciso y no generaba ningún problema el hecho de que requiera conocer la solución para obtener el hash.

En ambos casos, el hash se calcula una sola vez en el momento que se requiere por primera vez. Comparar dos hashs tiene una complejidad $O(clientes)$ ya que simplemente es comparar dos *strings* que tienen un largo que depende de la cantidad de clientes. Por lo tanto verificar que la solución que se va a insertar no existe en la población tiene un complejidad de $O(población * clientes)$.

Fig. 5.6: Ejemplo en donde el método *GetHashFromSolution* detecta individuos idénticos mientras que el método *GetHashFromRandomKeys* falla.



5.2.7. Resultados de la primer versión

Una vez que implementé el BRKGA puro (sin búsquedas locales), ejecuté la implementación diez veces para cada una de las seis instancias del benchmark previamente seleccionadas. Se pueden observar los resultados obtenidos en la tabla 5.4. Configuré el BRKGA de la siguiente manera:

MI.250;MNC.10;PS.100;EP.0,3;MP.0,1;EGC.70;LS.;TOP.0;D.G (ver sección 5.2.2).

Tab. 5.4: Resultados de 10 ejecuciones del BRKGA puro sobre las seis instancias seleccionadas.

Instancia	N/V/D	T_{avg}	B_{min}	B_{avg}	B_{max}	i_{eAvg}	i_{eMax}	$Best$
p2.2.k	21/2/22.50	2977	240	249	260	0.91	0.95	275
p2.3.g	21/3/10.70	1990	145	145	145	1.00	1.00	145
p3.4.p	33/4/22.50	6482	430	438	450	0.78	0.80	560
p5.3.x	66/3/40.00	17908	610	635	660	0.41	0.42	1555
p7.2.e	102/2/50.00	8753	204	217	246	0.75	0.85	290
p7.4.t	102/4/100.00	31532	458	481	513	0.45	0.48	1077

De estos primeros resultados podemos ver que el BRKGA puro funciona muy bien para instancias de testeo pequeñas. Esto es razonable por que al ser una instancia pequeña, las cantidad de combinaciones posibles es mucho menor. Luego el BRKGA explora un mayor

Tab. 5.5: Resultados de 10 ejecuciones del BRKGA puro sobre las seis instancias seleccionadas para cinco configuraciones generales diferentes.

Instancia	N/V/D	Config	T_{avg}	B_{min}	B_{avg}	B_{max}	i_{eAvg}	i_{eMax}	$Best$
p5.3.x	66/3/40.00	1	60782	635	656	700	0.42	0.45	1555
p5.3.x	66/3/40.00	2	23363	620	636	660	0.41	0.42	1555
p5.3.x	66/3/40.00	3	22357	615	643	685	0.41	0.44	1555
p5.3.x	66/3/40.00	4	7311	475	498	555	0.32	0.36	1555
p5.3.x	66/3/40.00	5	54239	630	668	750	0.43	0.48	1555
p7.4.t	102/4/100.00	1	143760	472	506	542	0.47	0.50	1077
p7.4.t	102/4/100.00	2	42255	471	485	504	0.45	0.47	1077
p7.4.t	102/4/100.00	3	45952	463	488	542	0.45	0.50	1077
p7.4.t	102/4/100.00	4	11587	268	284	322	0.26	0.30	1077
p7.4.t	102/4/100.00	5	96642	478	491	509	0.46	0.47	1077

porcentaje del conjunto de soluciones válidas del problema. Esto se refleja en la instancia *p2.3.g* que siempre se llegó a la mejor solución posible y en *p2.2.k* donde el i_eAvg supera el 0.90. Luego a medida que incrementa el tamaño de la instancia, disminuye el i_eAvg. Es interesante ver como en la instancia *p7.2.e* se obtuvieron resultados mucho mejores que aquellos obtenidos en la instancia *p7.2.t* considerando que ambas pertenecen al *set* siete. Es decir, tienen el mismo mapa de clientes y difieren en la cantidad de vehículos y el d_{max} de los vehículos. Claramente al aumentar la cantidad de vehículos y el d_{max} de los vehículos, aumenta considerablemente la cantidad de solucione posibles.

Con el objetivo de encontrar la mejor configuración general del BRKGA, tomé las dos instancias con menor i_{eAvg} de la tabla 5.4 y las utilice para probar distintas configuraciones. Los resultados de estos experimento se pueden observar en la tabla 5.5.

Configuraciones:

- **Config = 1**: MI.100;MNC.100;PS.500;EP.0,30;MP.0,05;EGC.70;LS.;TOP.0;D.G
- **Config = 2**: MI.150;MNC.30;PS.200;EP.0,25;MP.0,05;EGC.60;LS.;TOP.0;D.G
- **Config = 3**: MI.150;MNC.70;PS.200;EP.0,30;MP.0,10;EGC.70;LS.;TOP.0;D.G
- **Config = 4**: MI.150;MNC.70;PS.200;EP.0,30;MP.0,10;EGC.70;LS.;TOP.0;D.S
- **Config = 5**: MI.250;MNC.50;PS.250;EP.0,15;MP.0,05;EGC.50;LS.;TOP.0;D.G

Como síntesis de estos resultados observo que la configuración básica no tiene un fuerte impacto sobre el beneficio final de la ejecución. En el caso de la instancia *p5.3.x*, el i_{eAvg} siempre se encuentra en el intervalo [0.41,0.43] y en en *p7.4.t* el intervalo es [0.45,0.47] siempre que se utiliza el decodificador simple. Para ambas instancias hay una configuración que es claramente peor y es la configuración **4** donde se utiliza el decodificar simple en vez del goloso. Por lo tanto en esta versión del BRKGA puro el decodificador tiene gran impacto en el resultado final. Lamentablemente, el resto de las configuraciones impacta muy poco en el beneficio total cuando la instancia del problema es grande (Mínima cantidad de iteraciones, mínima cantidad de iteraciones sin cambios, tamaño de la población,

población elite, etc). Si observamos los tiempos de ejecución, al comparar el T_{avg} de la configuración **3** y **4** podemos ver que la configuración **3** es aproximadamente tres veces más lenta que la configuración **4** y solo difieren en el tipo de decodificador. Es muy claro que aunque el mejor beneficio obtenido con el decodificador simple es mucho menor, su tiempo de ejecución es mucho menor.

Como podemos ver en los resultados de la tabla 5.4, se pueden mejorar bastante los beneficios obtenidos y los tiempos de ejecución son bastante rápidos. Por lo tanto, hay lugar para agregar mejoras al algoritmo sacrificando tiempo de ejecución en búsqueda de mejores resultados.

5.3. Búsqueda Local

Con el objetivo de optimizar los resultados obtenidos hasta el momento, implementé algunas búsquedas locales. La idea fue aplicar las búsquedas a algunas de las mejores soluciones de cada generación. La cantidad de individuos a mejorar por generación es regida por el atributo *ApplyLocalSearchesToTop* del objeto *Configuration*. En caso de que a la solución ya se le hubiese aplicado las búsquedas en una generación anterior, se aplican a la siguiente mejor solución. Esto puede suceder ya que las mejores soluciones pertenecen al conjunto de elite y todos los individuos del conjunto de elite pasan directamente a la siguiente generación. Todas las búsquedas locales pueden modificar una solución ya sea para reducir su tiempo de recorrido o beneficio recolectado. La solución resultante de aplicar las búsquedas siempre es válida. Es decir, la solución resultante respeta la distancia máxima de la ruta de los vehículos y ningún cliente es visitado más de una vez.

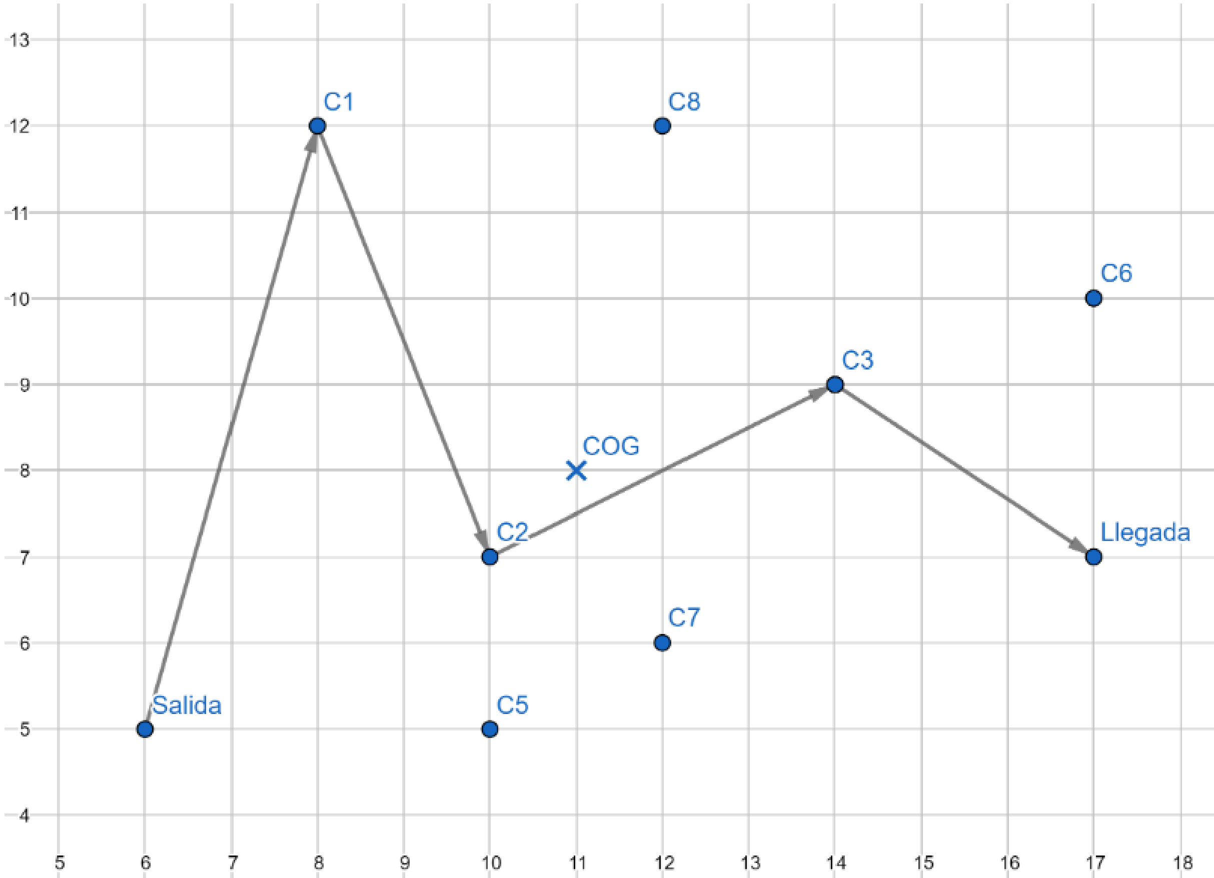
5.3.1. Centro de Gravedad

Para las búsquedas locales *Insert* y *Replace* se deben tomar una lista de clientes con algún orden. Este orden es importante ya que queremos empezar por las mejores opciones. El orden de clientes que se utiliza esta dado por su distancia al centro de gravedad (COG) de la ruta a la cual se le aplica el *Insert* o el *Replace*. Cuanto más cercano sea el cliente al COG de la ruta a optimizar, mayor prioridad tendrá el cliente.

En la figura 5.7 podemos observar diez nodos, de los cuales hay ocho clientes, un punto de salida y uno de llegada. También podemos observar una ruta que incluye a los clientes $C_1, C_2\ y\ C_3$. El punto COG se calcula a partir de las coordenadas de los nodos de salida, llegada y los clientes de la ruta. El cliente no visitado mas cercano al COG es C_7, seguido de $C_5, C_8\ y\ C_6$ respectivamente. Las distancias entre el COG y los nodos no visitados son calculadas de la misma forma que se calculan las distancias entre dos nodos cualesquiera, utilizando la distancia euclidiana.

Implementé el cálculo del COG de la forma que lo describen Vansteenwegen et al. [26]. La coordenada del COG de una ruta se calcula como muestran las fórmulas 5.4 y 5.5. Las variables x_i e y_i son las coordenadas de un cliente de la ruta y B_i es su beneficio. El cálculo del COG tiene una complejidad de $O(ruta.Length)$. Para no realizar cálculos innecesarios, el COG de una ruta solo se calcula cuando se necesita. Es decir, cuando la

Fig. 5.7: Ejemplo del COG para una ruta.



solución es seleccionada para ser mejorada. Se calcula una sola vez y cuando se modifica la ruta, se actualiza su COG.

Cuando una ruta es modificada, el COG debe modificarse acorde. Si se elimina un cliente de una ruta, hay que remover del COG el impacto que tenía el cliente eliminado sobre el COG (Ídem cuando se inserta un cliente en la ruta). La actualizacion del COG puede implementarse con un complejidad de $O(1)$ si mantenemos los valores de $x_{cog.num}$ y $x_{cog.den}$ cuando calculamos el COG por primera vez como se observa en la fórmula 5.6. La fórmula 5.7 muestra como se actualiza el COG en $O(1)$ luego de remover un cliente utilizando los valores de $x_{cog.num}$ y $x_{cog.den}$. Idem fórmula 5.8 para el caso en que se inserta un cliente.

Coordenada X del COG de una ruta.

$$x_{cog} = (\sum_{\forall i \in ruta} x_i * B_i) / \sum_{\forall i \in ruta} B_i \tag{5.4}$$

Coordenada Y del COG de una ruta.

$$y_{cog} = (\sum_{\forall i \in ruta} y_i * B_i) / \sum_{\forall i \in ruta} B_i \tag{5.5}$$

Sea r una ruta:

$$r.x_{cog} = \frac{\sum_{\forall i \in r.ruta} x_i * B_i}{\sum_{\forall i \in r.ruta} B_i} = \frac{r.x_{cog.num}}{r.x_{cog.den}} \tag{5.6}$$

Sea $r' = r.Remove(c_j)$ con c_j cliente y $c_j \in r$:

$$r'.x_{cog} = \frac{\sum_{\forall i \in r.ruta \wedge i \neq j} x_i * B_i}{\sum_{\forall i \in r.ruta \wedge i \neq j} B_i} = \frac{r.x_{cog.num} - x_j * B_j}{r.x_{cog.den} - B_j} \tag{5.7}$$

Sea $r'' = r.Add(c_k)$ con c_k cliente y $c_k \notin r$:

$$r'.x_{cog} = \frac{(\sum_{\forall i \in r.ruta} x_i * B_i) + x_k * B_k}{(\sum_{\forall i \in r.ruta} B_i) + B_k} = \frac{r.x_{cog.num} + x_k * B_k}{r.x_{cog.den} + B_k} \tag{5.8}$$

5.3.2. Swap

El objetivo de esta búsqueda es encontrar e intercambiar clientes entre dos rutas distintas con el fin de disminuir la suma de las distancias recorridas de ambas rutas. Es decir, dados v_a y v_b vehículos y sus respectivas rutas r_a y r_b, se puede realizar un *Swap* entre sus rutas si existe un cliente c_{a_i} en la ruta de r_a y otro cliente c_{b_j} en r_b tal que agregando c_{a_i} en alguna posición de r_b y agregando c_{b_j} en alguna posición de r_a son válidas las fórmulas 5.9, 5.10 y 5.11.

Sean r_a, r_b, r'_a y r'_b rutas, c_{a_i} cliente en r_a y c_{b_j} cliente en r_b tales que:

$$r'_a = r_a.Remove(c_{a_i}).Add(c_{b_j})$$

$$r'_b = r_b.Remove(c_{b_j}).Add(c_{a_i})$$

Luego se realiza el *Swap* si las siguientes formulas son válidas:

$$r_a.Dist + r_b.Dist > r'_a.Dist + r'_b.Dist \tag{5.9}$$

$$r'_a.Dist \leq v'_a.d_{max} \tag{5.10}$$

$$r'_b.Dist \leq v'_b.d_{max} \tag{5.11}$$

En el pseudocódigo 5.16 podemos ver que al aplicar esta búsqueda a una solución se ejecuta el método *SwapDestinationsBetween* para todo par de rutas de la solución. Por lo tanto, este método será llamado $vehículos * (vehículos - 1)/2$ veces (la cantidad de combinaciones posibles de tomar dos elementos de un conjunto). El método *SwapDestinationsBetween*, que podemos observar en el pseudocódigo 5.17, prueba intercambiar cada cliente de la ruta a con cada cliente de la ruta b y si efectivamente conviene hacer un *Swap*, lo realiza. De modo de no estar cambiando múltiples veces a un mismo cliente entre dos rutas en una misma ejecución, cuando se cambia de ruta a un cliente se lo agrega en una lista de clientes prohibidos para intercambiar hasta que termine la ejecución actual del *SwapDestinationsBetween*. Esta búsqueda local no mejora el beneficio total de una solución, lo que hace es disminuir la distancia recorrida de alguna ruta aumentando la probabilidad de insertar clientes no visitados.

Pseudocódigo 5.16: Método *ApplyLocalSearch* del *Swap*.

```
public bool ApplyLocalSearch(Solution solution)
{
  var changed = false;
  var combinations = GetCombinationsFor(solution.Vehicles.Count);
  foreach (var combination in combinations)
  {
    var v1 = solution.Vehicles[combination.Left];
    var v2 = solution.Vehicles[combination.Right];
    changed = changed || SwapDestinationsBetween(v1, v2);
  }
  return changed;
}
```

Pseudocódigo 5.17: El método *SwapDestinationsBetween*, prueba intercambiar todos los clientes entre dos rutas.

```
public bool SwapDestinationsBetween(Vehicle v1, Vehicle v2)
{
  var changed = false;
  var reset = false;
  var bans = new Dictionary<int, bool>();
  for (var i = 0; i < v1.Route.RouteLenght(); i++)
  {
    var client1 = v1.Route[i];
    if(bans.ContainsKey(client1.Id))
      continue;
    for (var j = 0; j < v2.Route.RouteLenght(); j++)
    {
      var client2 = v2.Route[j];
      if (v2Bans.ContainsKey(client2.Id))
        continue;
      if (!Swaps(i, j, ref leftRoute, ref rightRoute))
        continue;
      changed = true;
      reset = true;
      bans.Add(client1.Id, true);
      bans.Add(client2.Id, true);
      break; // Para resetear j
    }
    if(reset)
    {
      i = 0;
      reset = false;
    }
  }
  return changed;
}
```

Como podemos ver en el pseudocódigo de *SwapDestinationsBetween* 5.17, el método itera sobre todos los clientes de la ruta del primer vehículo y por cada uno de esos clientes, itera sobre todos los clientes de la ruta del segundo vehículo. Ademas, si se realiza un *swap*, se resetean los indices usados para iterar sobre los clientes visitados. Un cliente es intercambiado entre rutas a lo sumo una sola vez, entonces los indices se resetean a lo sumo una cantidad de veces equivalente a la cantidad de clientes visitados en la ruta más corta de ambos vehículos. El *Swap* llama al método *SwapDestinationsBetween* por cada combinación de vehículos en la solución. Por lo tanto el orden de complejidad del *Swap* es:

$$Swap \in O(vehículos^2 * clientes^3)$$

5.3.3. Insert

El objetivo de esta búsqueda local es encontrar una posición en alguna ruta para un cliente no visitado sin sobrepasar el limite de distancia máxima de la ruta. Básicamente para cada vehículo y cada cliente no visitado se busca en que posición se debe insertar el cliente de forma tal que minimice el incremento de distancia recorrida. Si la distancia resultante es menor a la distancia máxima del vehículo, se inserta al cliente en tal posición. En caso contrario, no se inserta y se prueba con el siguiente cliente no visitado. El orden en que se toman los clientes no visitados es según su distancia al COG de la ruta a optimizar, de forma ascendente.

Pseudocódigo 5.18: Método *ApplyLocalSearch* del *Insert*.

```
public bool ApplyLocalSearch(Solution solution)
{
  var changed = false;
  var uClients = solution.GetUnvistedClients;
  var vehicles = solution.Vehicles;
  foreach (var vehicle in vehicles)
  {
    vehicle.Route.ActivateCog();
    uClients = uClients.OrderBy(x => vehicle.DistanceToCog(x));
    for (var index = 0; index < uClients.Count; index++)
    {
      var res = AnalizeInsert(solution, vehicle, uClients[index]);
      if (res.CanBeInserted)
      {
        vehicle.AddDestinationAt(uClients[index],
            res.BestPosition);
        uClients.Remove(uClients[index]);
        changed = true;
      }
    }
  }
  return changed;
}
```

Como vemos en el pseudocódigo 5.18 al empezar la ejecución del *Insert*, lo primero que hace es activar el COG. Hasta el momento no había sido calculado para evitar cálculos innecesarios. Después de activar el COG, por cada cliente no visitado hasta el momento se analiza el *insert*. El método *AnalizeInsert* devuelve un objeto con dos propiedades que contienen la información necesaria para saber si se puede hacer el *Insert* y donde. El objeto tiene una propiedad de tipo *bool* que dice si el cliente puede ser insertado en el vehículo consultado. La otra propiedad dice cual es la mejor posición para insertar el cliente. Si el cliente es insertado en la ruta, se actualiza el COG de la ruta y se remueve el cliente de la lista de no visitados. El orden de complejidad del *Insert* es:

$$Insert \in 0(vehículos * clientes^2)$$

5.3.4. 2-Opt

El algoritmo *2-Opt* es un algoritmo simple de búsqueda local propuesto por Croes [10]. El objetivo es buscar un orden alternativo de los clientes visitados dentro de una misma ruta, de modo que disminuya la distancia recorrida por el vehículo. Es decir, un intercambio de posiciones entre dos clientes dentro de una misma ruta. Como podemos observar en el pseudocódigo 5.18 se aplica el método *Do2OptSwap* a cada vehículo de la solución.

Pseudocódigo 5.19: Método *ApplyLocalSearch* del *Insert*.

```
public bool ApplyLocalSearch(Solution solution)
{
  var index = 0;
  var changed = false;
  var vehicles = solution.Vehicles;
  while (index < vehicles.Count)
  {
    var currentDistance = vehicles[index].Route.GetDistance();
    changed = changed || Do2OptSwap(vehicles[index]);
    index++;
  }
  return changed;
}
```

El método *Do2OptSwap* primero obtiene una lista de las todas las posibles cambios que se pueden hacer dentro de una ruta. Después por cada combinación intenta hacer un intercambio de posiciones dentro de la ruta. El intercambio se realiza solo si al hacerlo la distancia recorrida disminuye. Si en efecto se realiza el reemplazo, el bucle vuelve a empezar desde el principio debido a que el cambio puede generar nuevos reemplazos. Como el reemplazo solo sucede cuando la distancia resultante es estrictamente menor a la distancia original, el bucle siempre termina ya que la distancia del recorrido de la ruta no se puede reducir indefinidamente. El pseudocódigo 5.20 muestra como implementé el *2-Opt*.

Pseudocódigo 5.20: Método *ApplyLocalSearch* del *Insert*.

```
private bool Do2OptSwap(Vehicle vehicle)
{
  var changed = false;
  var combinations = GetCombinationsFor(vehicle.Route);
  var index = 0;
  while (index < combinations.Count)
  {
    var pos1 = combinations[index].Postion1;
    var pos2 = combinations[index].Postion2;
    var swaped = vehicle.Route.SwapIfImproves(pos1, pos2);
    if (swaped)
    {
      changed = true;
      index = 0;
    }
    else
      index++;
  }
  return changed;
}
```

El *2-Opt* llama al *Do2OptSwap* una vez por cada vehículo. En *Do2OptSwap* se analiza intercambiar toda las posibles combinaciones de los clientes visitados por el vehículo. Por lo tanto el *2-Opt* tiene un orden de complejidad:

$$2\text{-}Opt \in 0(vehículos * clientes^2)$$

5.3.5. Replace Simple

Esta búsqueda tiene como objetivo intercambiar un cliente no visitado por un cliente visitado de una ruta de forma tal que aumente el beneficio de la ruta. Del mismo modo que la búsqueda *Insert*, los clientes no visitados se toman en orden según su distancia al COG de la ruta, empezando por los más cercanos. En el pseudocódigo 5.21 vemos que se aplica el *Replace Simple* a cada vehículo de la solución.

Pseudocódigo 5.21: Método *ApplyLocalSearch* del *Replace*.

```
public bool ApplyLocalSearch(Solution solution)
{
  var vehicles = solution.Vehicles;
  var changed = false;
  foreach (var vehicle in vehicles)
    changed = changed || Replace(solution, vehicle);
  return changed;
}
```

El *Replace Simple* es similar al *Insert*. El *Replace Simple* comienza analizando cual es la mejor posición para insertar el cliente no visitado y lo inserta sin verificar que el d_{max} del vehículo haya sido superado. En caso de que la solución siga siendo válida continua con el siguiente cliente del mismo modo que lo hace el *Insert*. El *Replace Simple* se diferencia del *Insert* en el caso en que la solución pase a ser inválida. En tal caso, se debe remover algún cliente de forma tal que se vuelva a respetar la restricción de la distancia máxima, d_{max}. Por lo tanto el *Replace Simple* busca un cliente de menor beneficio que el insertado, tal que al removerlo la ruta vuelva a tener una distancia recorrida menor a d_{max}. En caso de no encontrar ninguno se remueve el cliente que se había insertado en un principio. Si existen múltiples clientes candidatos a removerse, se elige el que minimice el recorrido de la ruta. Elegí priorizar distancia sobre beneficio en el caso de múltiples candidatos a remover. Tomé tal decisión por que también implementé otro reemplazo enfocado puramente en mejorar el beneficio de la ruta. De este modo los vecindarios explorados por ambas búsquedas de reemplazo contienen menor cantidad de soluciones en común. Llamé a esta búsqueda *Replace Simple* por que tiene una complejidad algorítmica menor que el otro reemplazo y cuando se efectuá un reemplazo siempre se remueve a lo sumo un cliente visitado. Podemos ver el pseudocódigo del *Replace Simple* en la figura 5.22.

Pseudocódigo 5.22: Método *Replace.*

```
private bool Replace(Solution solution, Vehicle vehicle)
{
  var unvisited = solution.GetCurrentUnvistedDestination;
  var changed = false;
  vehicle.Route.ActivateCog();
  uClients = uClients.OrderBy(x => vehicle.DistanceToCog(x));

  foreach (var client in uClients)
  {
    var res = AnalizeInsert(solution, vehicle, client);
    vehicle.AddDestinationAt(destination, res.BestInsertPosition);
    if (!res.CanBeInserted)
    {
      var justChanged = false;
      if(IsReplaceSimple())
        justChanged = RemoveWorst(vehicle, client);
      else
        justChanged = RemoveWorstGroup(vehicle, client);
      changed = changed || justChanged;
    }
    else
      changed = true;
  }
  return changed;
}
```

El método *RemoveWorst* recibe como parámetros el vehículo que excedió d_{max} y al cliente que se le insertó. El cliente que se insertó es el candidato que se removerá por defecto. Como podemos ver en el pseudocódigo 5.23, para cada cliente con menor beneficio que el beneficio del cliente insertado, se calcula la distancia resultante si se lo remueve. Finalmente, se remueve aquel cliente que minimice la distancia de la ruta que en el peor de los casos es el mismo cliente que se insertó.

Pseudocódigo 5.23: Selección del cliente a remover en el *Replace Simple*.

```
public bool RemoveWorst(Vehicle vehicle, Client inserted)
{
  var toRemove = inserted;
  var minDistance = vehicle.dMax;
  var route = vehicle.Route;
  foreach (var client in route)
  {
    if (client.Profit > inserted.Profit)
      continue;
    var distance = GetDistanceWithout(route, client);
    if (distance <= minDistance)
    {
      minDistance = distance;
      toRemove = client;
    }
  }
  vehicle.Route.RemoveClient(toRemove);
  return toRemove.Id != inserted.Id;
}
```

Por cada vehículo de la solución y para todo cliente no visitado el *Replace Simple* analiza cual es la mejor posición en donde insertar tal cliente y si la solución resultante es inválida analiza cual es el cliente visitado que requiere remover para que la solución resultante sea válida. Por lo tanto el *Replace Simple* tiene un orden de complejidad de:

$$ReplaceSimple \in 0(vehículos * clientes^2)$$

5.3.6. Replace Multiple

El *Replace Multiple* se diferencia del *Replace Simple* de dos formas. La primer diferencia es que explora todos los reemplazos posibles incluso remover múltiples clientes visitados por un cliente no visitado. La segunda diferencia es que dentro de todas sus opciones para reemplazar, selecciona aquella que maximice el beneficio. En cambio el *Replace Simple* cuando encuentra múltiples candidatos a remover elige aquel que minimice la distancia que recorre el vehículo. El *Replace Multiple* tiene una complejidad algorítmica mayor al *Replace Simple* ya que dentro de las opciones que explora incluye aquellas exploradas por el *Replace Simple*. La segunda diferencia la agregue con el objetivo de diferenciar, aún más, las soluciones resultantes de aplicar uno u otra búsqueda.

En mi desarrollo el *Replace Multiple* se diferencia del *Replace Simple* durante la ejecución del método *Replace* 5.22 descrito anteriormente. Si se está ejecutando un *Replace Simple* se llama al método *RemoveWorst* y si se está ejecutando el *Replace Multiple* se llama al método *RemoveWorstGroup*. Cuando comienza a ejecutarse el *RemoveWorstGroup* 5.24 lo primero que hace es armar una lista de los potenciales grupos de clientes a remover. Un grupo de clientes tiene el potencial de ser removido si la sumatoria de sus beneficios es menor al beneficio del cliente insertado. En el pseudocódigo 5.24 un *CandidateGroup* representa un potencial grupo de clientes a remover. Obtenida la lista de los grupos de clientes candidatos se filtran dejando solo aquellos grupos que al removerlos, la ruta resultante recorra una distancia menor a d_{max}. Si después del filtrado queda un solo grupo, se eliminan los clientes de tal grupo. Si hay varios grupos de clientes candidatos, se remueven de la ruta los clientes del grupo cuya sumatorio de beneficios sea menor. Si no queda ningún grupo, se elimina el cliente que se insertó.

Pseudocódigo 5.24: Selección de los clientes a remover en el *Replace Multiple*.

```
public bool RemoveWorstGroup(Vehicle vehicle, Client inserted)
{
  var changed = false;
  var candidatesToRemove = new CandidateGroup(inserted);
  var route = vehicle.Route;
  var candidateGroups = new List<CandidateGroup>();
  candidateGroups = GetCandidateGroups(route, client);

  var bestOption = false;
  var valid = false;

  foreach(var candidateGroup in candidateGroups)
  {
    var distance = GetDistanceWithout(route, candidateGroup);
    valid = distance <= vehicle.dMax;
    bestOption = candidateGroup.Profit < candidatesToRemove.Profit;
    if(valid && bestOption)
    {
      candidatesToRemove = candidateGroup;
      changed = true;
    }
  }

  foreach (var client in candidatesToRemove.Clients)
    vehicle.Route.RemoveClient(client);

  return changed;
}
```

Pseudocódigo 5.25: Método *GetCandidateGroups*.

```
public List<CandidateGroup> GetCandidateGroups(Vehicle v, Client c)
{
  var candidateGroups = new List<CandidateGroup>();
  var route = v.Route;
  for (var client in route)
  {
    if (client.Profit >= c.Profit)
      continue;
    var newGroup = new CandidateGroup(client);
    foreach (var candidateGroup in candidateGroups)
      if (candidateGroup.Profit + client.Profit < inserted.Profit)
        candidateGroup.Add(client);
    candidateGroups.Add(newGroup);
  }
  return candidateGroups;
}
```

El pseudocódigo 5.25 muestra como se obtienen los potenciales grupos de clientes a remover a partir del vehículo y el cliente insertado. Este método crea una lista con un tamaño máximo de $2^n - 1$, siendo n la cantidad de clientes visitados por el vehículo. El peor caso sucede cuando el cliente insertado tiene un beneficio mayor a la sumatoria de los beneficios de los clientes visitados por el vehículo. Por lo tanto el orden de complejidad de *GetCandidateGroups* es de $O(2^n)$. Agregar una metaheuristica con orden exponencial debe ser justificado. A modo de analizar el uso de tal metaheuristica desarrollé dos pruebas con el objetivo de responder las siguientes preguntas:

- Cuánto mejora la aptitud física de la solución final usando *Replace Multiple*?
- Es necesario acotar la cantidad de combinaciones de *GetCandidateGroups* de modo que el *Replace Multiple* tenga un orden de complejidad polinomial?

Primero realicé una prueba para mostrar en cuanto mejora la solución final utilizando el *Replace Multiple*. Para esta prueba seleccioné las dos instancias grandes para las cuales mi implementación tenía los peores resultados, las instancias *p5.3.x* y *p7.4.t*. Luego ejecuté 25 veces una implementación que utilizaba el reemplazo múltiple y otras 25 veces una implementación que no la utilizaba. Además, solo para estas pruebas, creé un *Replace Multiple* acotado. Este reemplazo múltiple acotado recibe como parámetro la cantidad máxima de combinaciones que tiene permitido probar al buscar reemplazos. Por lo tanto si su cota es la cantidad de clientes, el orden de complejidad de *GetCandidateGroups* pasaría a ser $O(clientes)$. De modo de poder observar la progresión en que mejora la solución final y el tiempo que se demora la implementación en encontrarla, ejecuté la implementación variando el valor de la cota. Primero con un valor de 2, luego 4, 8, 16 y así sucesivamente hasta 128. Se realizaron 25 ejecuciones para cada una de las configuraciones mencionas y se modificó la condición de parada del BRKGA de forma tal que siempre termine luego de *MinIterations*, así todas las pruebas duran la misma cantidad de iteraciones globales. Los resultados pueden observarse en la tabla 5.6.

La tabla 5.6 tiene las filas ordenadas según su configuración. Primero la configuración con todas las búsquedas locales mencionadas anteriormente excepto *Replace Multiple*, luego

Tab. 5.6: Prueba de eficiencia del *Replace Multiple*.

Instancia	N/V/D	Config	T_{avg}	B_{min}	B_{avg}	B_{max}	I_{eAvg}	Best
p5.3.x	66/3/40.00	OSIRs	2894	1410	1443	1485	0.93	1555
p5.3.x	66/3/40.00	OSIRsRma002	3565	1460	1485	1510	0.95	1555
p5.3.x	66/3/40.00	OSIRsRma004	3650	1455	1484	1510	0.95	1555
p5.3.x	66/3/40.00	OSIRsRma008	3869	1470	1494	1525	0.96	1555
p5.3.x	66/3/40.00	OSIRsRma016	4035	1480	1495	1515	0.96	1555
p5.3.x	66/3/40.00	OSIRsRma032	4103	1480	1497	1525	0.96	1555
p5.3.x	66/3/40.00	OSIRsRma064	4047	1470	1493	1525	0.96	1555
p5.3.x	66/3/40.00	OSIRsRma128	4059	1470	1495	1525	0.96	1555
p5.3.x	66/3/40.00	OSIRsRm	4054	1465	1496	1525	0.96	1555
p7.4.t	102/4/100.00	OSIRs	5080	945	972	1022	0.90	1077
p7.4.t	102/4/100.00	OSIRsRma002	7123	969	995	1039	0.92	1077
p7.4.t	102/4/100.00	OSIRsRma004	7788	977	1004	1024	0.93	1077
p7.4.t	102/4/100.00	OSIRsRma008	8561	1003	1019	1045	0.95	1077
p7.4.t	102/4/100.00	OSIRsRma016	10277	998	1022	1042	0.95	1077
p7.4.t	102/4/100.00	OSIRsRma032	10387	1006	1023	1045	0.95	1077
p7.4.t	102/4/100.00	OSIRsRma064	10320	1008	1024	1045	0.95	1077
p7.4.t	102/4/100.00	OSIRsRma128	10352	1010	1024	1044	0.95	1077
p7.4.t	102/4/100.00	OSIRsRm	10386	1005	1027	1046	0.95	1077

le sigue una configuración que agrega el *Replace Multiple Acotado* por 2, luego por 4, por 8, etc. Por último se utiliza directamente el *Replace Multiple* sin cota. Podemos ver que el beneficio promedio aumenta considerablemente cuando se le agrega el *Replace Multiple* acotado por 2. Luego sigue mejorando a medida que se incrementa el valor de la cota hasta que llega un punto donde prácticamente deja de mejorar cuando la cota vale 8 o 16. El último caso corresponde al *Replace Multiple* sin cota y es donde encontramos los mejores resultados en la instancia *p7.4.t*. En la instancia *p5.3.x* no se ven diferencias de beneficio entre el *Replace Multiple* y los *Replace Multiple* acotados con valores mayores a 8. Por otro lado si observamos los tiempos de ejecución sucede algo similar. Los tiempos promedios incrementan a medida que se incrementa la cota y a partir de que la cota vale 16 o más, se estabiliza y no sigue incrementando. Estas instancias tienen 66 y 102 clientes, superando la cota de 16 ampliamente. Por lo tanto para estas instancias el *Replace Multiple* y su versión acotada por la cantidad de clientes, tienen el mismo rendimiento por más que uno tiene un orden de complejidad exponencial y el otro lineal.

La segunda prueba la realicé sobre una instancia del benchmark y una copia idéntica a la cual le alteré el beneficio de sus clientes. La idea en esta prueba es maximizar la probabilidad de que suceda el peor escenario en el método *GetCandidateGroups*. Una instancia que maximice la probabilidad del peor escenario sería tal donde el beneficio de cada cliente es mayor a la sumatoria de todos los clientes ya definidos. Si el beneficio del primer cliente es 1, el siguiente será 2, luego 4, 8 y así sucesivamente. Por lo tanto el beneficio del último cliente será de $2^{clientes-1}$. Como este valor crece rápidamente, seleccioné la instancia *p2.2.k* que tiene 19 clientes además de los nodos de salida y llegada. A la instancia inventada la llamé *p2.2.k.DeltaProfit*. Cada instancia se ejecutó 25 veces por cada una de las variaciones

Tab. 5.7: Prueba de eficiencia temporal del *Replace Multiple* en el peor escenario.

Instancia	N/V/D	Config	T_{avg}
p2.2.k	21/2/22.50	OSIRs	647
p2.2.k	21/2/22.50	OSIRsRma002	678
p2.2.k	21/2/22.50	OSIRsRma004	673
p2.2.k	21/2/22.50	OSIRsRma008	687
p2.2.k	21/2/22.50	OSIRsRma016	718
p2.2.k	21/2/22.50	OSIRsRma032	719
p2.2.k	21/2/22.50	OSIRsRma064	720
p2.2.k	21/2/22.50	OSIRsRma128	701
p2.2.k	21/2/22.50	OSIRsRm	715
p2.2.k.DeltaProfit	21/2/22.50	OSIRs	602
p2.2.k.DeltaProfit	21/2/22.50	OSIRsRma002	639
p2.2.k.DeltaProfit	21/2/22.50	OSIRsRma004	661
p2.2.k.DeltaProfit	21/2/22.50	OSIRsRma008	860
p2.2.k.DeltaProfit	21/2/22.50	OSIRsRma016	997
p2.2.k.DeltaProfit	21/2/22.50	OSIRsRma032	994
p2.2.k.DeltaProfit	21/2/22.50	OSIRsRma064	992
p2.2.k.DeltaProfit	21/2/22.50	OSIRsRma128	1018
p2.2.k.DeltaProfit	21/2/22.50	OSIRsRm	1032

del *Replace Multiple* de la prueba anterior. El objetivo de este experimento es solo analizar la diferencia de tiempos de una instancia con el peor escenario del método *GetCandidateGroups*, luego no muestro los resultados de los beneficios obtenidos. Los resultados se encuentran en la tabla 5.7.

En la tabla 5.7 podemos observar que en ambas instancias los tiempo de ejecución van incrementando a medida que se incrementa la cota, hasta que se estabiliza cuando la cota del *Replace Multiple* vale 16. En la instancia *p2.2.k.DeltaProfit* los tiempors son mayores, lo que significa que efectivamente se incrementó la probabilidad de ocurrencia del peor escenario. Luego de ambas pruebas podemos justificar el uso del *Replace Multiple* sin cota ya que en la practica tiene un tiempo de ejecución similar al del *Replace Multiple* acotado por la cantidad de clientes y el orden de complejidad de la versión acotada es polinomial.

Entre ambas pruebas realizadas se concluye que no es necesario acotar el método *GetCandidateGroups* ya que en la práctica los tiempos de ejecución son similares en ambas versiones. En cuanto al incremento de la aptitud física de la solución final obtenida, cuando comparamos los resultados en la tabla 5.6 de la configuraciones OSIRs y OSIRsRm, vemos que agregar el *Replace Multiple* incrementa el i_{eAvg} en 0,04 en promedio. Lo cual es bastante considerando que el i_{eAvg} de OSIRs es de 0.93 para la instancia *p5.3.x* y 0.90 para la instancia *p7.4.t*.

El *Replace Multiple* mejoró considerablemente el i_{eAvg} en instancias grandes ya que explora más opciones que el *Replace Simple* y prioriza el beneficio sobre la distancia. Esto lo hace sacrificando tiempo ya que su complejidad algorítmica es mayor a la del *Replace Simple*. El *Replace Multiple* tiene un orden de complejidad:

$$ReplaceMultiple \in O(vehículos * clientes * 2^{clientes})$$

En la práctica tiene un tiempo de ejecución similar al *Replace Multiple* acotado por cantidad de clientes que tiene un orden de complejidad de:

$$\begin{aligned} ReplaceMultipleAcotado &\in O(vehículos * clientes * clientes^2) \\ &= O(vehículos * clientes^3) \end{aligned}$$

5.3.7. Encoder

Agregar búsquedas locales entre generación de poblaciones conlleva un problema que debe resolverse. Al mejorar la solución se modifican sus rutas. Ahora bien, si no se actualizan los genes de una solución acorde a los cambios realizados por las búsquedas locales sus descendientes heredarán los genes de la solución no optimizada. Una vez que se optimiza una solución el vector de *RandomKeys* debe ser actualizado de forma que sean válidas la ecuaciones 5.12 y 5.13.

Ecuaciones que deben ser válidas para que el *crossover* sea correcto.

$$ApplyLocalSearch(s) = s' \tag{5.12}$$

$$s' = Decoder.Decode(s'.RandomKeys, ProblemInfo) \tag{5.13}$$

Como mencioné en la sección del Decodificador (ver sección 5.1.1), los clientes se ordenan de forma ascendente por la propiedad *Key* del objeto *RandomKey* asociado según la propiedad *ClientId*. Por lo tanto el primer cliente con el que trabaja el decodificador, es el cliente con menor valor de *Key*. Algo que no mencioné sobre la implementación de los decodificadores es que el primer vehículo por el que empieza es el de menor *Id* ya que los ordena por su *Id* de forma ascendente. Los vehículos son indistinguibles al tener el mismo d_{max} en el benchmark de instancias de problemas. De todos modos ahora debo respetar la decisión que tomé en el desarrollo de los decodificadores. Por lo tanto al primer cliente de la ruta del vehículo con menor *Id* se le debe asociar el *RandomKey* que tenga el menor *Key*. Así, cuando el decodificador inicie, lo primero que hará es tomar este cliente e intentará adjudicárselo al primer vehículo, que justamente será el de menor *Id*. Continuando con esta lógica el segundo cliente del mismo vehículo debe tener asignado el segundo *RandomKey* de menor *Key*. Y así sucesivamente, hasta tener mapeados todos los clientes del primer vehículo con su nuevo *RandomKey*. Este proceso debe repetirse con los clientes del siguiente vehículo ordenados por *Id* ascendentemente. Finalmente, quedarán sin *RandomKey* asignado todos los clientes no visitados. En principio a estos clientes se les podría asignar cualquier *RandomKey*. Aún así, en pos de disminuir los cambios genéticos sobre el individuo, se les asigna un *RandomKey* tal que entre ellos mantengan el mismo orden que tenían antes de la mejora. Es decir, dentro de los clientes no visitados, el cliente que previamente tenía el *RandomKey* con menor *Key*, se le asigna el *RandomKey* de menor *Key* que queda disponible.

En la figura 5.8 podemos observar un vector de *RandomKeys* ordenado por su campo *Key*. Los *ClientId* resaltados con color verde claro pertenecen al primer vehículo y los resaltados con verde oscura pertenecen al segundo vehículo. Debajo del vector de *RandomKeys* podemos ver el hash de la solución y por último las rutas que fueron generadas por el decodificador goloso (los mismo clientes que estaban coloreados en el vector de *RandomKeys*).

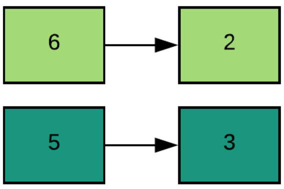
Fig. 5.8: Dado un *RandomKeys*, se obtiene un hash y una solución generada por el decodificador goloso.

Dado el siguiente vector ordenado de RandomKeys:

Key	7	13	21	27	45	54	79	89
ClientId	6	2	5	1	4	8	3	7

Hash de la solución generada: 6@2#5@3

El decodificar goloso genera una solución que contiene las rutas:

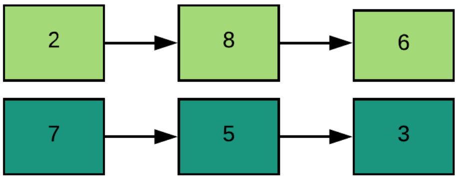


A la solución de la figura 5.8 se le aplican algunas búsquedas locales y podemos ver los cambios en la figura 5.9. Ambas rutas se extienden con un nuevo cliente, la primer ruta incluso cambia el orden en que se visitan el cliente 2 y el 6. Debajo de las rutas vemos el vector de *RandomKeys* ordenado por su campo *Key* con los campos *ClientId* actualizados de forma tal que al decodificar el vector de *RandomKeys* genere la solución optimizada.

Existe un escenario en donde al decodificar un *RandomKeys* actualizado por el codificador no genere exactamente la misma solución que fue optimizada. Supongamos que tenemos el *RandomKeys* de la figura 5.9 que fue actualizado por el codificador luego de que la solución a la cual corresponde fuera optimizada por las búsquedas locales. Cuando el decodificador goloso le asigna el ClientId 6 al primer vehículo antes de cambiar de vehículo intentará asignar el resto de los clientes al primer vehículo. Lo que normalmente sucederá es que no logre insertar ninguno de los otros clientes, pero podría ser que alguno de los clientes pueda insertarse en el primer vehículo. De ser así existen dos opciones. La primera es que el cliente extra que se agregara a la primer ruta pertenezca a otro vehículo.

Fig. 5.9: Se aplican búsquedas locales a la solución de la figura 5.8, después se actualiza el *RandomKeys* y el hash de la solución.

Las búsquedas locales mejoran las rutas agregando clientes y modificando el orden del recorrido:



El encoder actualiza el mapeo entre Key y ClientId del vector de RandomKeys:

Key	7	13	21	27	45	54	79	89
ClientId	2	8	6	7	5	3	1	4

Hash de la nueva solución: 2@8@6#7@5@3

En el ejemplo de la figura 5.9 podría ser el caso los ClientId: 7, 5 o 3. Si sucede esto, el segundo vehículo tendría más tiempo disponible y menor beneficio. Si al segundo vehículo no se le inserta ningún cliente extra, el beneficio total de la solución se mantiene. En caso contrario algún cliente no visitado ahora tiene lugar en el segundo vehículo y el beneficio de la solución incremento. La segunda opción es que el cliente extra insertado en el primer vehículo era un cliente no visitado. En el ejemplo de la figura 5.9 podría ser el caso los ClientId: 1 o 4. Luego el segundo vehículo no se vería afectado. Lo que significa que el beneficio total de la solución aumento. Por lo tanto, existen escenarios donde la ecuación 5.12 no es válida pero el beneficio total de la solución resultante es igual o mayor a la que debería ser. Eso no es un problema si consideramos la función objetivo, además es un escenario poco factible ya que significaría que las búsquedas *Insert*, *Replace Simple* y *Replace Multiple* no encontraron esta mejor solución que es vecina de la solución encontrada. De todos modos, opte por implementar unos delimitadores que agrega el codificador al actualizar el vector de *RandomKeys* de forma tal que cuando los decodificadores pasan por uno de estos delimitadores son forzados a cambiar de vehículo. Esto asegura la validez de la ecuación 5.12. Podemos observar el pseudocódigo 5.26 que actualiza el vector de *RandomKeys* luego de que la solución es optimizada por las búsquedas locales.

Pseudocódigo 5.26: Método que actualiza el vector de *RandomKeys*.

```
public static Solution UpdateRandomKeys(Solution s);
{
  var randomKeys = s.GetOrderedRandomKeys();
  // Todas las keys ordenadas ascendente
  var keys = randomKeys.Select(k => k.Key).ToList();
  var ClientIds = new List<int>();
  // Get ClientId from Visited Clients
  foreach (var r in s.Routes)
  {
    var d = r.GetDestinations();
    var rci = d.Select(d => d.ClientId);
    ClientIds.AddRange(rci);
  }
  // Get ClientId from Unvisited Clients
  var uClientIds = GetUnvisitedClientIds(randomKeys, newRoutes);
  ClientIds.AddRange(uClientIds);

  // Hay un break por cada cantidad de clientes en ruta
  var breaks = new Queue(newRoutes.Select(r => r.ClientsCount));

  var newRandomKeys = new List<RandomKey>();
  var endRoute = false;
  var acumBreak = 0;
  for (var index = 0; index < keys.Count; index++)
  {
    if (breaks.Count > 0)
    {
      endRoute = index + 1 == (int)breaks.Peek() + acumBreak;
      if (endRoute)
        acumBreak += (int)breaks.Dequeue();
    }
    var randomKey = new RandomKey()
    {
      Key = keys[index],
      ClientId = ClientIdes[index],
      ForceVehicleChangeAfterThis = endRoute
    };
    newRandomKeys.Add(randomKey);
  }
  s.SetRandomKeys(newRandomKeys);
  return s;
}
```

Como se puede observar en los pseudocódigo de las búsquedas locales (5.16, 5.18, 5.19 y 5.21), todas implementan el método *ApplyLocalSearch* que toma una solución y retorna un booleano que vale *true* si la solución fue modificada y false en caso contrario. Esto lo implementé así para poder ejecutar todas las búsquedas locales en una secuencia sin saber cuál es el orden de la secuencia antes de la instanciación del BRKGA. De esta forma puedo setear la secuencia con el objeto *Configuration* en el momento que se instancia el BRKGA. Gracias a esto pude probar las búsquedas locales en múltiples órdenes distintos. En el pseudocódigo 5.27 podemos ver cómo se aplican las búsquedas según el orden en que se encuentran en la lista que toma de parámetro. Después de aplicar las búsquedas, si la solución fue modificada se actualiza el vector de *RandomKeys*.

Pseudocódigo 5.27: Aplicación de las búsquedas locales a una solución.

```
public Solution ApplyLS(List<ILocalSearch> list, Solution solution)
{
  var changed = false;
  foreach (var localSearch in list)
    changed = changed || localSearch.ApplyLocalSearch(solution);

  if(changed)
    solution = Encoder.UpdateRandomKeys (solution);

  return solution
}
```

5.3.8. Orden de ejecución de las búsquedas locales

Como mencioné previamente, la lista de búsquedas locales se setea en el momento en que se instancia el BRKGA y las búsquedas se aplican en el orden en que se encuentran en la lista. Por ejemplo, si *lista = (Swap, Insert, Replace Multiple)*, entonces primero se aplicará el *Swap*, seguido del *Insert* y finalmente el *Replace Multiple*. La lista puede contener búsquedas repetidas, por ejemplo *lista = (Swap, Insert, 2-Opt, Insert)*. El orden en que se ejecutan las búsquedas locales tiene un impacto fuerte sobre la solución final generada.

Con el objetivo de encontrar la mejor secuencia de búsquedas locales a aplicar, generé 7 configuraciones distintas para mi BRKGA que sólo difieren en las listas de búsquedas locales. Para cada una de las 7 configuraciones ejecuté el BRKGA 25 veces sobre las dos instancias de problemas para las cuales había obtenido los peores resultados en el BRKGA puro. La configuración básica que comparten todas las configuraciones es la siguiente:

MI.400;MNC.100;PS.150;EP.0,3;MP.0,1;EGC.0,70;TOP.2;DT.S.

Recordando los códigos de las búsquedas locales:

- **I**: Insert (Cliente no visitado)
- **Rs**: Replace Simple (Cliente no visitado por uno visitado)
- **Rm**: Replace Mutiple (Cliente no visitado por uno o varios visitado/s)
- **0**: 2-Opt (Swap dentro de una misma ruta)
- **S**: Swap (Swap entre dos rutas distintas)

En la tabla 5.8, se pueden observar los resultados de variar el orden en que se aplican las búsquedas locales. Claramente, el peor orden de búsquedas locales es IRmRsOS (*Insert*, *Replace Multiple*, *Replace Simple*, *2-Opt*, *Swap*). Las búsquedas *Insert* y *Replace* intentan agregar más clientes o intercambiar por clientes más rentables mejorando el beneficio de

Tab. 5.8: Resultados de aplicar distintas listas de búsquedas locales.

Instancia	Search	T_{avg}	B_{min}	B_{avg}	B_{max}	i_{eAvg}	i_{eMax}	$Best$
p5.3.x	IRmRsOS	49397	1460	1485	1540	0.95	0.99	1555
p5.3.x	ORsSIRm	39576	1485	1509	1525	0.97	0.98	1555
p5.3.x	SIORsSORm	43556	1495	1512	1535	0.97	0.99	1555
p5.3.x	SOIORsRmSORm	49449	1505	1522	1545	0.98	0.99	1555
p5.3.x	SOIRsRm	36595	1500	1512	1525	0.97	0.98	1555
p5.3.x	SOSIRsSORm	40375	1505	1521	1535	0.98	0.99	1555
p5.3.x	SRsOIRm	43423	1480	1510	1535	0.97	0.99	1555
p7.4.t	IRmRsOS	81876	1004	1038	1064	0.96	0.99	1077
p7.4.t	ORsSIRm	86537	1024	1038	1063	0.96	0.99	1077
p7.4.t	SIORsSORm	91839	1033	1049	1077	0.97	1.00	1077
p7.4.t	SOIORsRmSORm	126705	1042	1055	1069	0.98	0.99	1077
p7.4.t	SOIRsRm	82889	1032	1047	1071	0.97	0.99	1077
p7.4.t	SOSIRsSORm	94731	1038	1055	1071	0.98	0.99	1077
p7.4.t	SRsOIRm	90306	1024	1042	1067	0.97	0.99	1077

Tab. 5.9: Resultados de aplicar distintos decodificadores al BRKGA con búsquedas locales.

Instancia	Deco	T_{avg}	B_{min}	B_{avg}	B_{max}	i_{eAvg}	i_{eMax}	$Best$
p5.3.x	G	93131	1515	1525	1545	0.98	0.99	1555
p5.3.x	S	50313	1515	1524	1535	0.98	0.99	1555
p7.4.t	G	198831	1043	1047	1051	0.97	0.98	1077
p7.4.t	S	119328	1041	1051	1059	0.98	0.98	1077

la ruta. En cambio, las búsquedas *2-Opt* y *Swap* modifican la secuencia en que se visitan los clientes seleccionados con el objetivo de minimizar la distancia recorrida de una ruta. Al ejecutar primero las búsquedas que incrementan el beneficio y luego las que reducen la distancia recorrida, no se hace uso de la distancia disminuida por *2-Opt* y *Swap*.

Entre las otras 6 combinaciones aquellas con mejor resultados en sus 25 ejecuciones fueron SOSIRsSORm y SOIORsRmSORm. Ambas son las únicas que obtuvieron un i_{eAvg} de 0.98 para las dos instancias. La configuración SIORsSORm llegó a encontrar la mejor solución conocida para la instancia *p7.4.t* pero su i_{eAvg} es de 0.97. Decidí priorizar el valor del i_{eAvg} sobre el valor del i_{eMax} porque el i_{eMax} no es estable, por lo tanto no garantiza buenos resultados. Entre las dos configuraciones con mejor i_{eAvg} seleccioné SOSIRsSORm por que como se puede observar en la tabla 5.8 tuvo un tiempo de ejecución promedio al menos un 20 % menor al de la configuración SOIORsRmSORm.

Por último, antes de calcular los resultados finales sobre todo el bechmark de problemas, hice una prueba de eficiencia entre los dos decodificadores ahora que el BRKGA tiene búsquedas locales. Para las mismas dos instancias de la tabla 5.8, ejecuté 10 veces el BRKGA variando solamente los decodificadores utilizados. Para esta prueba utilicé la lista de búsquedas locales que mejores resultados obtuvo en el análisis previo: SOSIRsSORm.

En la tabla 5.9 podemos observar los resultados obtenidos sobre las instancias *p5.3.x* y *p7.4.t* para el BRKGA con búsquedas locales modificando solamente los decodificadores. Los beneficios obtenidos son prácticamente iguales utilizando cualquiera de los decodificadores. Donde podemos ver diferencias en los resultados es la columna T_{avg}, el tiempo promedio de ejecución. El decodificador goloso demoró aproximadamente un 80 % más en ejecutarse que el decodificador simple. Es por este motivo que elegí el decodificador simple para mis resultados finales.

Con estas últimas pruebas tengo definido la configuración final para el BRKGA. Una última observación que quiero hacer es lo mucho que mejoraron los resultados en las instancias grandes luego de implementar las búsquedas locales. En la tabla 5.4 la instancia *p7.4.t* obtuvo un i_{eAvg} de 0.45 mientras que en la tabla 5.9 obtuvo un i_{eAvg} de 0.98. Lo que significa que para las instancias grandes, mis resultados obtuvieron un valor un 2 % menor que el mejor resultado publicado para tales instancias, es por eso que consideré que los resultados se encontraban en un nivel muy satisfactorio y decidí pasar a calcular los resultados finales sobre todo el benchmark de problemas.

6. RESULTADOS

Los resultados fueron generados corriendo la implementación en una laptop HP con las siguientes especificaciones:

- Procesador: Intel Core i7 5500u
- Memoria: DDR3 12 GBytes
- Graphics: Intel HD Graphics 5500
- Sistema Operativo: Windows 10 64-bit Home
- Ide: Visual Studio Enterprice 2015
- Lenguaje: C# .Net Framework 4.5

Una última mejora que implementé en mi algoritmo es la aplicación de una secuencia larga de búsquedas locales a la mejor solución cada 10 generaciones. Esta secuencia contiene el doble de búsquedas que la secuencia que describí anteriormente. El objetivo fue disminuir la probabilidad de no encontrar una mejor solución vecina a la mejor solución encontrada. La configuración final utilizada para el BRKGA fue:

- **MinIterations**: 250
- **MinNoChanges**: 100
- **PopulationSize**: 100
- **ElitePercentage**: 0.3
- **MutantPercentage**: 0.1
- **EliteGenChance**: 0.70
- **LocalSearches**: Swap, 2-Opt, Swap, Insert, Replace Simple, Swap, 2-Opt, Replace Multiple
- **ApplyLocalSearchesToTop**: 2
- **DecoderType**: Simple

Dentro de todos los trabajos previos de la literatura que encontré que incluyeran resultados del benchmark de instancias de Chao y los de Tsiligirides [18], comparé mis resultados con los obtenidos del *Memetic Algorithm* (MA) de Bouly et al. [5], del *Ant Colony Optimization* (ACO_{seq}) de Ke et al. [19] y del *Variable Neighborhood Search* (VNS_{slow}) de Archetti et al. [1]. Elegí estos trabajos previos por que contienen enfoques diversos y obtuvieron los mejores resultados de la literatura. Los resultados se pueden observar en las tablas: 6.2, 6.3, 6.4, 6.5, 6.6, 6.7, 6.8 y 6.9. Cada instancia se ejecutó 3 veces para poder comparar los resultados del mismo modo que lo hicieron los trabajos previos mencionados. Las columnas B_{min}, B_{avg} y B_{max} muestran el valor mínimo, promedio y máximo respectivamente obtenido por mi implementación para la instancia correspondiente. Agregue también la columna i_{eMax} para obtener una rápida lectura de la fila, si $i_{eMax} = 1$ entonces en al menos una de las 3 ejecuciones se obtuvo el mejor beneficio publicado para tal instancia. Como muestran las tablas, los resultados obtenidos fueron muy buenos. Se alcanzó el mejor resultado conocido para un 70 % de las instancias del benchmark.

La tabla 6.1 muestra una síntesis de los resultados. En esta tabla se agregaron los resultados del *Tabu Search* de Tang et al. [24] y del trabajo de Chao et al. [9]. También incluyo las otras propuestas de Archetti et al. [1]: el $\text{TS}_{\text{penalty}}$, $\text{TS}_{\text{feasible}}$ y el VNS_{fast}. El resto de los trabajos son los mismos con los que comparé mis resultados finales. Según el resumen de resultados, la eficiencia de mi algoritmo se encuentra por debajo del trabajo de Tang et al. [24] y por encima de la propuesta de Chao et al. [9]. Las ecuaciones 6.1, 6.2 y 6.3 muestran como se obtiene los valores de las columnas de la tabla 6.1 que resume los resultados. El valor δZ_{min} del algoritmo X es la sumatoria de las diferencias entre el mejor valor obtenido por cualquier algoritmo para cierta instancia y el menor valor obtenido para la misma instancia por el algoritmo X (idem δZ_{max}). El valor de δZ es la diferencia entre δZ_{min} y δZ_{max}.

$$\delta Z_{min} = \sum_{i \in instance} Best_i - B_{min} \tag{6.1}$$

$$\delta Z_{max} = \sum_{i \in instance} Best_i - B_{max} \tag{6.2}$$

$$\delta Z = \delta Z_{min} - \delta Z_{max} \tag{6.3}$$

Tab. 6.1: Síntesis de los resultados.

Algoritmo	δZ_{min}	δZ_{max}	δZ
CGW	4340	-	-
BRKGA	3054	1636	1418
TMH	2404	-	-
$\text{TS}_{\text{penalty}}$	2376	981	1395
$\text{TS}_{\text{feasible}}$	1184	399	785
VNS_{fast}	1436	352	1084
VNS_{slow}	427	84	343
ACO_{seq}	-	204	-
MA	434	80	354

Tab. 6.2: Resultados finales (Parte 1).

Instancia	B_{min}	B_{avg}	B_{max}	ACO_{seq}	VNS_{slow}	MA	i_{eMax}	$Best$
p2.2.a	90	90	90	90	90	90	1.00	90
p2.2.b	120	120	120	120	120	120	1.00	120
p2.2.c	140	140	140	140	140	140	1.00	140
p2.2.d	160	160	160	160	160	160	1.00	160
p2.2.e	190	190	190	190	190	190	1.00	190
p2.2.f	200	200	200	200	200	200	1.00	200
p2.2.g	200	200	200	200	200	200	1.00	200
p2.2.h	230	230	230	230	230	230	1.00	230
p2.2.i	230	230	230	230	230	230	1.00	230
p2.2.j	260	260	260	260	260	260	1.00	260
p2.2.k	275	275	275	275	275	275	1.00	275
p2.3.a	70	70	70	70	70	70	1.00	70
p2.3.b	70	70	70	70	70	70	1.00	70
p2.3.c	105	105	105	105	105	105	1.00	105
p2.3.d	105	105	105	105	105	105	1.00	105
p2.3.e	120	120	120	120	120	120	1.00	120
p2.3.f	120	120	120	120	120	120	1.00	120
p2.3.g	145	145	145	145	145	145	1.00	145
p2.3.i	200	200	200	200	200	200	1.00	200
p2.3.j	200	200	200	200	200	200	1.00	200
p2.3.k	200	200	200	200	200	200	1.00	200
p2.4.a	10	10	10	10	10	10	1.00	10
p2.4.b	70	70	70	70	70	70	1.00	70
p2.4.c	70	70	70	70	70	70	1.00	70
p2.4.d	70	70	70	70	70	70	1.00	70
p2.4.e	70	70	70	70	70	70	1.00	70
p2.4.f	105	105	105	105	105	105	1.00	105
p2.4.g	105	105	105	105	105	105	1.00	105
p2.4.h	120	120	120	120	120	120	1.00	120
p2.4.i	120	120	120	120	120	120	1.00	120
p2.4.j	120	120	120	120	120	120	1.00	120
p2.4.k	180	180	180	180	180	180	1.00	180
p1.2.b	15	15	15	15	15	15	1.00	15
p1.2.c	20	20	20	20	20	20	1.00	20
p1.2.d	30	30	30	30	30	30	1.00	30
p1.2.e	45	45	45	45	45	45	1.00	45
p1.2.f	80	80	80	80	80	80	1.00	80
p1.2.g	90	90	90	90	90	90	1.00	90
p1.2.h	110	110	110	110	110	110	1.00	110
p1.2.i	135	135	135	135	135	135	1.00	135
p1.2.j	155	155	155	155	155	155	1.00	155
p1.2.k	175	175	175	175	175	175	1.00	175
p1.2.l	195	195	195	195	195	195	1.00	195
p1.2.m	215	215	215	215	215	215	1.00	215
p1.2.n	235	235	235	235	235	235	1.00	235

Tab. 6.3: Resultados finales (Parte 2).

Instancia	B_{min}	B_{avg}	B_{max}	ACO_{seq}	VNS_{slow}	MA	i_{eMax}	$Best$
p1.2.o	240	240	240	240	240	240	1.00	240
p1.2.p	250	250	250	250	250	250	1.00	250
p1.2.q	265	265	265	265	265	265	1.00	265
p1.2.r	280	280	280	280	280	280	1.00	280
p1.3.c	15	15	15	15	15	15	1.00	15
p1.3.d	15	15	15	15	15	15	1.00	15
p1.3.e	30	30	30	30	30	30	1.00	30
p1.3.f	40	40	40	40	40	40	1.00	40
p1.3.g	50	50	50	50	50	50	1.00	50
p1.3.i	105	105	105	105	105	105	1.00	105
p1.3.j	115	115	115	115	115	115	1.00	115
p1.3.k	135	135	135	135	135	135	1.00	135
p1.3.l	155	155	155	155	155	155	1.00	155
p1.3.m	175	175	175	175	175	175	1.00	175
p1.3.n	190	190	190	190	190	190	1.00	190
p1.3.p	220	220	220	220	220	220	1.00	220
p1.3.q	230	230	230	230	230	230	1.00	230
p1.4.d	15	15	15	15	15	15	1.00	15
p1.4.e	15	15	15	15	15	15	1.00	15
p1.4.f	25	25	25	25	25	25	1.00	25
p1.4.g	35	35	35	35	35	35	1.00	35
p1.4.h	45	45	45	45	45	45	1.00	45
p1.4.i	60	60	60	60	60	60	1.00	60
p1.4.j	75	75	75	75	75	75	1.00	75
p1.4.k	100	100	100	100	100	100	1.00	100
p1.4.l	120	120	120	120	120	120	1.00	120
p1.4.m	130	130	130	130	130	130	1.00	130
p1.4.n	155	155	155	155	155	155	1.00	155
p1.4.o	165	165	165	165	165	165	1.00	165
p1.4.p	175	175	175	175	175	175	1.00	175
p1.4.q	190	190	190	190	190	190	1.00	190
p1.4.r	210	210	210	210	210	210	1.00	210
p3.2.a	90	90	90	90	90	90	1.00	90
p3.2.b	150	150	150	150	150	150	1.00	150
p3.2.c	180	180	180	180	180	180	1.00	180
p3.2.d	220	220	220	220	220	220	1.00	220
p3.2.e	260	260	260	260	260	260	1.00	260
p3.2.f	300	300	300	300	300	300	1.00	300
p3.2.g	360	360	360	360	360	360	1.00	360
p3.2.h	410	410	410	410	410	410	1.00	410
p3.2.i	460	460	460	460	460	460	1.00	460
p3.2.j	510	510	510	510	510	510	1.00	510
p3.2.k	550	550	550	550	550	550	1.00	550
p3.2.l	590	590	590	590	590	590	1.00	590
p3.2.m	620	620	620	620	620	620	1.00	620

Tab. 6.4: Resultados finales (Parte 3).

Instancia	B_{min}	B_{avg}	B_{max}	ACO_{seq}	VNS_{slow}	MA	i_{eMax}	$Best$
p3.2.n	660	660	660	660	660	660	1.00	660
p3.2.o	690	690	690	690	690	690	1.00	690
p3.2.p	720	720	720	720	720	720	1.00	720
p3.2.q	760	760	760	760	760	760	1.00	760
p3.2.r	790	790	790	790	790	790	1.00	790
p3.2.s	800	800	800	800	800	800	1.00	800
p3.2.t	800	800	800	800	800	800	1.00	800
p3.3.a	30	30	30	30	30	30	1.00	30
p3.3.b	90	90	90	90	90	90	1.00	90
p3.3.c	120	120	120	120	120	120	1.00	120
p3.3.d	170	170	170	170	170	170	1.00	170
p3.3.e	200	200	200	200	200	200	1.00	200
p3.3.f	230	230	230	230	230	230	1.00	230
p3.3.g	270	270	270	270	270	270	1.00	270
p3.3.h	300	300	300	300	300	300	1.00	300
p3.3.i	330	330	330	330	330	330	1.00	330
p3.3.j	380	380	380	380	380	380	1.00	380
p3.3.k	440	440	440	440	440	440	1.00	440
p3.3.l	480	480	480	480	480	480	1.00	480
p3.3.m	520	520	520	520	520	520	1.00	520
p3.3.n	570	570	570	570	570	570	1.00	570
p3.3.o	590	590	590	590	590	590	1.00	590
p3.3.p	640	640	640	640	640	640	1.00	640
p3.3.q	680	680	680	680	680	680	1.00	680
p3.3.r	710	710	710	710	710	710	1.00	710
p3.3.s	720	720	720	720	720	720	1.00	720
p3.3.t	750	756	760	760	760	760	1.00	760
p3.4.a	20	20	20	20	20	20	1.00	20
p3.4.b	30	30	30	30	30	30	1.00	30
p3.4.c	90	90	90	90	90	90	1.00	90
p3.4.d	100	100	100	100	100	100	1.00	100
p3.4.e	140	140	140	140	140	140	1.00	140
p3.4.f	190	190	190	190	190	190	1.00	190
p3.4.g	220	220	220	220	220	220	1.00	220
p3.4.h	240	240	240	240	240	240	1.00	240
p3.4.i	270	270	270	270	270	270	1.00	270
p3.4.j	310	310	310	310	310	310	1.00	310
p3.4.l	380	380	380	380	380	380	1.00	380
p3.4.m	390	390	390	390	390	390	1.00	390
p3.4.n	440	440	440	440	440	440	1.00	440
p3.4.o	500	500	500	500	500	500	1.00	500
p3.4.p	560	560	560	560	560	560	1.00	560
p3.4.q	560	560	560	560	560	560	1.00	560
p3.4.r	600	600	600	600	600	600	1.00	600
p3.4.s	670	670	670	670	670	670	1.00	670

Tab. 6.5: Resultados finales (Parte 4).

Instancia	B_{min}	B_{avg}	B_{max}	ACO_{seq}	VNS_{slow}	MA	i_{eMax}	$Best$
p3.4.t	670	670	670	670	670	670	1.00	670
p6.2.d	192	192	192	192	192	192	1.00	192
p6.2.e	360	360	360	360	360	360	1.00	360
p6.2.f	588	588	588	588	588	588	1.00	588
p6.2.g	660	660	660	660	660	660	1.00	660
p6.2.h	774	778	780	780	780	780	1.00	780
p6.2.i	882	884	888	888	888	888	1.00	888
p6.2.j	942	944	948	948	948	948	1.00	948
p6.2.k	1032	1032	1032	1032	1032	1032	1.00	1032
p6.2.l	1104	1104	1104	1116	1116	1116	0.99	1116
p6.2.m	1158	1166	1176	1188	1188	1188	0.99	1188
p6.2.n	1224	1236	1242	1260	1260	1260	0.99	1260
p6.3.g	282	282	282	282	282	282	1.00	282
p6.3.h	444	444	444	444	444	444	1.00	444
p6.3.i	636	638	642	642	642	642	1.00	642
p6.3.j	828	828	828	828	828	828	1.00	828
p6.3.k	894	894	894	894	894	894	1.00	894
p6.3.l	984	994	1002	1002	1002	1002	1.00	1002
p6.3.m	1074	1076	1080	1080	1080	1080	1.00	1080
p6.3.n	1158	1160	1164	1170	1170	1170	0.99	1170
p6.4.l	696	696	696	696	696	696	1.00	696
p6.4.m	906	908	912	912	912	912	1.00	912
p6.4.n	1068	1068	1068	1068	1068	1068	1.00	1068
p5.2.b	20	20	20	20	20	20	1.00	20
p5.2.c	50	50	50	50	50	50	1.00	50
p5.2.d	80	80	80	80	80	80	1.00	80
p5.2.e	180	180	180	180	180	180	1.00	180
p5.2.f	240	240	240	240	240	240	1.00	240
p5.2.g	320	320	320	320	320	320	1.00	320
p5.2.h	410	410	410	410	410	410	1.00	410
p5.2.i	480	480	480	480	480	480	1.00	480
p5.2.j	580	580	580	580	580	580	1.00	580
p5.2.k	670	670	670	670	670	670	1.00	670
p5.2.l	770	783	800	800	800	800	1.00	800
p5.2.m	850	853	860	860	860	860	1.00	860
p5.2.n	920	921	925	925	925	925	1.00	925
p5.2.o	990	1000	1010	1020	1020	1020	0.99	1020
p5.2.p	1095	1108	1120	1150	1150	1150	0.97	1150
p5.2.q	1165	1178	1190	1195	1195	1195	1.00	1195
p5.2.r	1250	1255	1260	1260	1260	1260	1.00	1260
p5.2.s	1315	1325	1330	1340	1340	1330	0.99	1340
p5.2.t	1375	1378	1380	1400	1400	1400	0.99	1400
p5.2.u	1450	1453	1460	1460	1460	1460	1.00	1460
p5.2.v	1500	1501	1505	1505	1505	1505	1.00	1505
p5.2.w	1555	1556	1560	1560	1565	1560	1.00	1565

Tab. 6.6: Resultados finales (Parte 5).

Instancia	B_{min}	B_{avg}	B_{max}	ACO_{seq}	VNS_{slow}	MA	i_{eMax}	$Best$
p5.2.x	1580	1586	1595	1610	1610	1610	0.99	1610
p5.2.y	1615	1623	1630	1645	1635	1645	0.99	1645
p5.2.z	1645	1655	1665	1680	1670	1680	0.99	1680
p5.3.b	15	15	15	15	15	15	1.00	15
p5.3.c	20	20	20	20	20	20	1.00	20
p5.3.d	60	60	60	60	60	60	1.00	60
p5.3.f	110	110	110	110	110	110	1.00	110
p5.3.g	185	185	185	185	185	185	1.00	185
p5.3.h	260	260	260	260	260	260	1.00	260
p5.3.i	335	335	335	335	335	335	1.00	335
p5.3.j	465	468	470	470	470	470	1.00	470
p5.3.k	495	495	495	495	495	495	1.00	495
p5.3.l	585	588	595	595	595	595	1.00	595
p5.3.m	650	650	650	650	650	650	1.00	650
p5.3.n	755	755	755	755	755	755	1.00	755
p5.3.o	870	870	870	870	870	870	1.00	870
p5.3.p	990	990	990	990	990	990	1.00	990
p5.3.q	1065	1066	1070	1070	1070	1070	1.00	1070
p5.3.r	1110	1113	1120	1125	1125	1125	1.00	1125
p5.3.s	1185	1186	1190	1190	1190	1190	1.00	1190
p5.3.t	1250	1253	1255	1260	1260	1260	1.00	1260
p5.3.u	1330	1333	1335	1345	1345	1345	0.99	1345
p5.3.v	1415	1418	1425	1425	1425	1425	1.00	1425
p5.3.w	1460	1460	1460	1485	1485	1485	0.98	1485
p5.3.x	1515	1525	1535	1540	1555	1555	0.99	1555
p5.3.y	1565	1575	1585	1590	1595	1590	0.99	1595
p5.3.z	1615	1620	1625	1635	1635	1635	0.99	1635
p5.4.c	20	20	20	20	20	20	1.00	20
p5.4.d	20	20	20	20	20	20	1.00	20
p5.4.e	20	20	20	20	20	20	1.00	20
p5.4.f	80	80	80	80	80	80	1.00	80
p5.4.g	140	140	140	140	140	140	1.00	140
p5.4.h	140	140	140	140	140	140	1.00	140
p5.4.i	240	240	240	240	240	240	1.00	240
p5.4.j	340	340	340	340	340	340	1.00	340
p5.4.k	340	340	340	340	340	340	1.00	340
p5.4.l	430	430	430	430	430	430	1.00	430
p5.4.m	550	550	550	555	555	555	0.99	555
p5.4.n	620	620	620	620	620	620	1.00	620
p5.4.o	685	686	690	690	690	690	1.00	690
p5.4.p	760	760	760	765	765	760	0.99	765
p5.4.q	840	850	860	860	860	860	1.00	860
p5.4.r	940	953	960	960	960	960	1.00	960
p5.4.s	1010	1015	1025	1030	1030	1030	1.00	1030
p5.4.t	1140	1153	1160	1160	1160	1160	1.00	1160

Tab. 6.7: Resultados finales (Parte 6).

Instancia	B_{min}	B_{avg}	B_{max}	ACO_{seq}	VNS_{slow}	MA	i_{eMax}	$Best$
p5.4.u	1230	1256	1270	1300	1300	1300	0.98	1300
p5.4.v	1300	1313	1320	1320	1320	1320	1.00	1320
p5.4.w	1375	1376	1380	1390	1390	1380	0.99	1390
p5.4.x	1430	1431	1435	1450	1450	1450	0.99	1450
p5.4.y	1500	1506	1510	1520	1520	1520	0.99	1520
p5.4.z	1570	1575	1580	1620	1620	1620	0.98	1620
p4.2.a	206	206	206	206	206	206	1.00	206
p4.2.b	341	341	341	341	341	341	1.00	341
p4.2.c	452	452	452	452	452	452	1.00	452
p4.2.d	528	528	529	531	531	531	1.00	531
p4.2.e	599	602	607	618	618	618	0.98	618
p4.2.f	668	671	676	687	687	687	0.98	687
p4.2.g	731	739	744	757	757	757	0.98	757
p4.2.h	817	820	823	827	835	835	0.99	835
p4.2.i	896	897	899	918	918	918	0.98	918
p4.2.j	932	935	937	965	962	964	0.97	965
p4.2.k	988	997	1007	1022	1022	1022	0.99	1022
p4.2.l	1047	1057	1068	1071	1074	1071	0.99	1074
p4.2.m	1087	1098	1105	1130	1132	1132	0.98	1132
p4.2.n	1147	1151	1157	1168	1174	1174	0.99	1174
p4.2.o	1179	1184	1195	1215	1218	1217	0.98	1218
p4.2.p	1210	1220	1227	1242	1241	1242	0.99	1242
p4.2.q	1242	1244	1248	1263	1263	1267	0.99	1267
p4.2.r	1260	1262	1264	1288	1285	1292	0.98	1292
p4.2.s	1276	1279	1286	1304	1301	1304	0.99	1304
p4.2.t	1292	1296	1299	1306	1306	1306	0.99	1306
p4.3.b	38	38	38	38	38	38	1.00	38
p4.3.c	193	193	193	193	193	193	1.00	193
p4.3.d	335	335	335	335	335	335	1.00	335
p4.3.e	460	465	468	468	468	468	1.00	468
p4.3.f	567	575	579	579	579	579	1.00	579
p4.3.g	637	645	652	653	653	653	1.00	653
p4.3.h	708	709	711	720	729	725	0.98	729
p4.3.i	782	784	786	796	809	809	0.97	809
p4.3.j	825	830	833	861	861	861	0.97	861
p4.3.k	891	902	908	918	919	919	0.99	919
p4.3.l	955	956	958	979	979	974	0.98	979
p4.3.m	998	1019	1032	1053	1062	1063	0.97	1063
p4.3.n	1075	1081	1090	1121	1121	1121	0.97	1121
p4.3.o	1118	1127	1132	1170	1172	1172	0.97	1172
p4.3.p	1149	1156	1169	1221	1222	1222	0.96	1222
p4.3.q	1206	1211	1215	1252	1245	1252	0.97	1252
p4.3.r	1244	1247	1252	1267	1273	1273	0.98	1273
p4.3.s	1262	1264	1267	1293	1295	1295	0.98	1295
p4.3.t	1281	1286	1292	1305	1304	1304	0.99	1305

Tab. 6.8: Resultados finales (Parte 7).

Instancia	B_{min}	B_{avg}	B_{max}	ACO_{seq}	VNS_{slow}	MA	i_{eMax}	$Best$
p4.4.d	38	38	38	38	38	38	1.00	38
p4.4.e	183	183	183	183	183	183	1.00	183
p4.4.f	324	324	324	324	324	324	1.00	324
p4.4.g	461	461	461	461	461	461	1.00	461
p4.4.h	555	560	571	571	571	571	1.00	571
p4.4.i	643	650	657	657	657	657	1.00	657
p4.4.j	725	727	730	732	732	732	1.00	732
p4.4.k	801	805	815	821	821	821	0.99	821
p4.4.l	842	847	851	880	880	879	0.97	880
p4.4.m	895	906	912	918	918	918	0.99	919
p4.4.n	933	936	942	961	976	969	0.96	977
p4.4.o	994	1004	1017	1036	1061	1061	0.96	1061
p4.4.p	1057	1065	1073	1111	1120	1124	0.95	1124
p4.4.q	1101	1109	1114	1145	1161	1161	0.96	1161
p4.4.r	1170	1173	1179	1200	1207	1216	0.97	1216
p4.4.s	1200	1215	1239	1249	1260	1259	0.98	1260
p4.4.t	1247	1257	1268	1281	1285	1284	0.99	1285
p7.2.a	30	30	30	30	30	30	1.00	30
p7.2.b	64	64	64	64	64	64	1.00	64
p7.2.c	101	101	101	101	101	101	1.00	101
p7.2.d	190	190	190	190	190	190	1.00	190
p7.2.e	290	290	290	290	290	290	1.00	290
p7.2.f	384	385	387	387	387	387	1.00	387
p7.2.g	454	455	457	459	459	459	1.00	459
p7.2.h	521	521	521	521	521	521	1.00	521
p7.2.i	575	577	579	580	579	579	1.00	580
p7.2.j	637	639	641	646	644	646	0.99	646
p7.2.k	698	699	701	705	705	704	0.99	705
p7.2.l	753	757	759	767	767	767	0.99	767
p7.2.m	818	824	827	827	827	827	1.00	827
p7.2.n	864	869	876	888	888	888	0.99	888
p7.2.o	921	926	935	945	945	945	0.99	945
p7.2.p	981	986	995	1002	1002	1002	0.99	1002
p7.2.q	1022	1024	1027	1043	1043	1044	0.98	1044
p7.2.r	1065	1073	1082	1094	1094	1094	0.99	1094
p7.2.s	1104	1114	1125	1136	1135	1136	0.99	1136
p7.2.t	1145	1150	1155	1179	1179	1179	0.98	1179
p7.3.b	46	46	46	46	46	46	1.00	46
p7.3.c	79	79	79	79	79	79	1.00	79
p7.3.d	117	117	117	117	117	117	1.00	117
p7.3.e	175	175	175	175	175	175	1.00	175
p7.3.f	247	247	247	247	247	247	1.00	247
p7.3.g	344	344	344	344	344	344	1.00	344
p7.3.h	419	423	425	425	425	425	1.00	425
p7.3.i	485	485	486	487	487	487	1.00	487

Tab. 6.9: Resultados finales (Parte 8).

Instancia	B_{min}	B_{avg}	B_{max}	ACO_{seq}	VNS_{slow}	MA	i_{eMax}	$Best$
p7.3.j	557	559	562	564	564	563	1.00	564
p7.3.k	620	624	630	633	633	633	1.00	633
p7.3.l	674	677	680	684	683	683	0.99	684
p7.3.m	739	742	748	762	762	762	0.98	762
p7.3.n	801	805	813	820	813	820	0.99	820
p7.3.o	858	860	863	874	874	874	0.99	874
p7.3.p	912	915	918	929	927	927	0.99	929
p7.3.q	961	965	970	987	987	987	0.98	987
p7.3.r	1000	1003	1006	1026	1026	1024	0.98	1026
p7.3.s	1057	1057	1057	1081	1081	1081	0.98	1081
p7.3.t	1101	1102	1103	1118	1117	1120	0.98	1120
p7.4.b	30	30	30	30	30	30	1.00	30
p7.4.c	46	46	46	46	46	46	1.00	46
p7.4.d	79	79	79	79	79	79	1.00	79
p7.4.e	123	123	123	123	123	123	1.00	123
p7.4.f	164	164	164	164	164	164	1.00	164
p7.4.g	217	217	217	217	217	217	1.00	217
p7.4.h	285	285	285	285	285	285	1.00	285
p7.4.i	364	364	366	366	366	366	1.00	366
p7.4.j	462	462	462	462	462	462	1.00	462
p7.4.k	518	518	518	520	520	518	1.00	520
p7.4.l	578	581	583	590	590	590	0.99	590
p7.4.m	643	644	646	646	646	646	1.00	646
p7.4.n	711	716	719	730	726	726	0.98	730
p7.4.o	774	775	777	781	781	779	0.99	781
p7.4.p	834	835	836	846	846	846	0.99	846
p7.4.q	891	894	896	909	909	907	0.99	909
p7.4.r	949	949	950	970	970	970	0.98	970
p7.4.s	997	1005	1012	1022	1022	1022	0.99	1022
p7.4.t	1050	1058	1063	1077	1077	1077	0.99	1077

7. CONCLUSIONES

El *Team Orienteering Problem* combina la decisión de qué clientes seleccionar con la decisión de cómo planificar la ruta. Al ser TOP un problema reconocido como modelo de muchas aplicaciones reales, se han generado varios trabajos que lo encaran. Incluso algunos pocos con algoritmos genéticos pero no encontré ninguno que implemente un BRKGA. Mi contribución al problema TOP consiste en generar una implementación que utilice como base de su construcción de soluciones al algoritmo BRKGA, mejorando las soluciones con búsquedas locales y analizar que tan efectivo es tal combinación de metaheurísticas.

Los resultados finales obtenidos son muy buenos, con al menos un 70 % de los resultados llegaron a la mejor solución conocida de la instancia testeada. El resto obtuvo un i_{eMax} en el intervalo [0.97, 0.99], salvo algunos pocos que quedaron en el intervalo [0.94, 0.96]. Los resultados del BRKGA puro no fueron lo suficientemente buenos para instancias grandes del problema, llegando a tener un i_{eMax} aproximado de 0,50. Quizá esto se deba a como funciona el *crossover* en TOP, las soluciones hijas terminan siendo muy diferentes de sus padres. Si ese fuera el caso, el BRKGA solo puede llegar a buenas soluciones con la ayuda de otras metaheurísticas como es el caso de mi desarrollo.

Uno de los problemas del BRKGA para TOP es que su secuencia de alelos no es utilizada de forma completa cuando se lo decodifica ya que parte del problema es que no todos los clientes pueden ser visitados. Por lo tanto, asignar todos los clientes a algún vehículo siempre generaría una solución no factible o la instancia del problema no sería del TOP. Este problema de matching entre alelos y clientes visitados quizá puede ser resuelto modificando lo que representa un gen.

Al iniciar el desarrollo, se esperaba obtener buenos resultados del BRKGA puro en base a las características de los algoritmos genéticos tales como la supervivencia del mas apto, la herencia de las características del padre de elite, una población de mejora monotónica, etc. Sin embargo los resultados durante el desarrollo dejaron en claro que el BRKGA puro no obtuvo buenos resultados y que los resultados finales dependieron demasiado de las búsquedas locales. De todos modos este trabajo deja algunos aportes e ideas de como resolver ciertos problemas cuando se implementa un BRKGA. Tales como:

- Se implementaron dos decodificadores, uno que genera una solución lo mas rápido posible y el otro que intenta generar mejores soluciones con una estrategia golosa. Se generaron diversas pruebas sobre estos decodificadores, se compararon y se analizaron sus resultados.
- Se ideo e implementó un codificador de soluciones de modo que al combinar el BRKGA con búsquedas locales, el método de *crossover* siga siendo independiente del problema que resuelve y logrando que los descendientes hereden genes que se corresponden a la aptitud de la solución de la cual es hija.

- Se desarrollaron dos método para poder decidir equivalencia entre soluciones. Uno de los métodos aunque no asegura unicidad de soluciones dentro de una población, reduce considerablemente los repetidos y no requiere conocimiento del problema. Se realizaron pruebas de eficiencia entre ambos métodos.

- Se implementaron las búsquedas locales clásicas, pero ademas se hicieron variantes de algunas de forma que varíen la vecindad que exploran y se plantearon alternativas cuando el orden de complejidad era muy alto.

- A lo largo de toda la implementación el objetivo de llegar a la mejor solución posible se mantuvo sin dejar de lado la optimización temporal algorítmica. Como fue el caso de calcular el centro de gravedad de una ruta de modo de seleccionar primero los clientes mas cercanos a la ruta al intentar insertarlos pero solo calcularlo en el momento que es necesario y mantenerlo de un modo eficiente. También en el caso de la búsqueda local de reemplazo de múltiples clientes a la vez. Esta búsqueda tiene un orden de complejidad alto pero en la practica es muy bueno, por lo tanto se hicieron pruebas para verificar que en la practica no era tan costoso y se realizaron pruebas para medir cuanto aportaba a la solución final.

7.1. Trabajos Futuros

Sería útil tener una herramienta para visualizar las soluciones en un plano cartesiano, pudiendo ver rápidamente qué clientes se quedaron sin ser visitados y así poder idear alternativas para que los clientes cercanos sean incluidos. También para poder ver la similitud entre un individuo y los individuos descendientes, a modo de tener una idea clara de qué tan parecidos son. De todos modos, para un análisis más preciso de la correlación entre padres e hijos, sería más eficiente idear una función que analizando las rutas de ambas soluciones genere un índice de parentesco.

El BRKGA puro para TOP necesita mejoras, los resultados obtenidos utilizando solo el BRKGA están lejos de ser competitivos. Si continuara el desarrollo del BRKGA sin búsquedas locales exploraría cambios en el decodificador y en el método de *crossover*.

En el decodificador buscaría alguna manera de que su secuencia de alelos se use de forma completa, es decir que todo alelo impacte en la formación de la solución. Para entender esto tomar como ejemplo el decodificador simple, una instancia con 10 clientes y digamos que la solución generada a partir de su secuencia de alelos visita a 6 de los 10 clientes. Por lo tanto, los últimos 4 alelos de la secuencia no impactan en el resultado final. Es decir, estos 4 alelos podrían cambiar de posición entre si y la solución resultante sería la misma. Quizá se podría implementar de tal forma, que los clientes se distribuyan uniformemente entre todos los vehículos y luego con un proceso de limpieza se convierta la solución en una factible. Otra alternativa sería implementar sectores preestablecidos asignados a un determinado vehículo, basados en cercanía ó el centro de gravedad del sector. Con ese índice de parentesco podríamos determinar cuan familiares son los descendientes y si hay diferencia entre realizar el *crossover* o generar soluciones mutantes.

El segundo punto por el que intentaría mejorar los resultados del BRKGA es modificando el algoritmo de apareamiento. Quizá, el individuo resultante deba heredar rutas enteras. Entonces el individuo descendiente herede dos rutas de un padre y la tercer ruta del otro. Finalmente, con algún proceso de limpieza se muevan los clientes que se visitan de forma repetida y se incluyen otros. En este contexto, la cantidad de alelos que tendría una solución estaría dictaminado por la cantidad de vehículos. Esto podría representar un problema ya que existen muchos menos vehículos que clientes, generando baja diversidad de soluciones, es decir explorando muy poco el dominio de soluciones posibles.

Como vimos en los resultados parciales durante el desarrollo de la implementación, el algoritmo BRKGA puro no tuvo éxito resolviendo TOP. En instancias grandes el i_eAvg apenas llegó a obtener valores de 0.50 aproximadamente. Es bastante probable que esto se deba a una baja correlación entre padres e hijos. Es decir, que las soluciones generadas a partir del *crossover* sean soluciones muy distintas a sus padres. Si este fuera el caso, no habría mucha diferencia entre generar una solución al azar y el método de *crossover* que se implementó. Por lo tanto, como potencial trabajo futuro se podría hacer una investigación exhaustiva para determinar si existe combinación de método de *crossover* y decodificador tal que el BRKGA puro llegue a valores de i_eAvg de 0.80 o más. Para esta investigación habría que plantear diversos combinaciones de métodos de *crossover* y de decodificadores soluciones. Luego formular distintos índices de similitud entre soluciones para medir parentesco. Finalmente realizar un test estadístico para determinar si existe correlación entre las implementaciones con un alto índice de parentesco y buenos resultados finales. En otras palabras tales trabajos futuros deberían poder responder las siguientes preguntas:

- ¿Existe combinación de método de *crossover* y de decodificador de soluciones tal que el BRKGA puro obtenga un $i_eAvg > 0{,}80$?
- ¿Existe correlación entre buenos resultados del BRKGA y una alta similitud entre padres e hijos?

Sobre trabajos futuros relacionados con las búsquedas locales, se podría implementar la búsqueda *Move*, para mover un cliente visitado de una ruta hacia otra, acumulando mayor distancia libre en una sola ruta. También se podría implementar alguna heurística local tabú de modo de salir de mínimos locales. De todos modos, no continuaría por las búsquedas locales ya que es un tema que esta muy desarrollado.

Los resultados finales fueron muy buenos, en el caso de continuar trabajando en mi desarrollo haría foco en las ideas sobre cambio del método de *crossover* y en el decodificador del BRKGA.

REFERENCES

[1] Archetti C., Hertz A. y M.G. Speranza. *Metaheuristics for the team orienteering problem.* Journal of Heuristics, 13:49–76, (2007).

[2] Archetti C., Speranza M.G. y Vigo D. *Vehicle Routing Problems with Profits.* Department of Economics and Management, University of Brescia, Italy (2013).

[3] Ballou R. y Chowdhury M. *MSVS: An extended computer model for transport mode selection.* The Logistics and Transportation. (1980).

[4] Bean J.C. *Genetic algorithms and random keys for sequencing and optimization.* ORSA Journal on Computing 6:154–160 (1994).

[5] Bouly H., Dang D.-C. y Moukrim A. *A memetic algorithm for the team orienteering problem.* A Quarterly Journal of Operations Research, 8:49–70, (2010).

[6] Boussier S., Feillet D. y Gendreau M. *An exact algorithm for the team orienteering problem.* A Quarterly Journal of Operations Research, 5:211–230, (2007).

[7] Butt S.E. y Cavalier T.M. *A heuristic for the multiple tour maximum collection problem.* Computers and Operations Research, 21:101–111, (1994).

[8] Butt S.E. y Ryan D.M. *An optimal solution procedure for the multiple tour maximum collection problem using column generation.* Computers and Operations Research, 26:427–441, (1999).

[9] Chao I-M., Golden B.L. y Wasil E.A. *The team orienteering problem.* European Journal of Operational Research, 88:464–474, (1996).

[10] Croes G.A. *A Method for Solving Travelling-Salesman Problems.* Operations Research, 6:791-812, (1958).

[11] Dang D.C., Guibadj R.N. y Moukrim A. *A PSO-based memetic algorithm for the team orienteering problem.* In: Di Chio C. et al. (eds) Applications of Evolutionary Computation. EvoApplications (2011).

[12] Ferreira J., Quintas A., Oliveira J. A., Pereira G. A. B. y Dias L. *Solving the team orienteering problem* Universidade do Minho, Braga, Portugal, (2014).

[13] Garey, M., y Johnson, D. *Computers and Intractability.* Freeman, San Francisco, (1979).

[14] Goldberg D. *Genetic algorithms in search, optimization and machine learning.* 1st Ed., Addison-Wesley, Massachusetts, (1989).

[15] Golden B., Laporte G. y Taillard E. *An adaptive memory heuristic for a class of vehicle routing problems with minmax.* Computers and Operations Research, 24:445–52, (1997).

[16] Golden B.L., Levy L. y Vohra R. *The orienteering problem.* Naval Research Logistics, 34:307–318, (1987).

[17] Gunawana A., Laua H.C., Vansteenwegen P. *Orienteering Problem: A survey of recent variants, solution approaches and applications.* European Journal of Operational Research, 255:315-332, (2016).

[18] Benchmark de instancias de problemas del TOP por Chao I-M. y Tsiligirides T. `https://www.mech.kuleuven.be/en/cib/op#section-3`

[19] Ke L., Archetti C. y Feng Z. *Ants can solve the team orienteering problem.* Computers and Industrial Engineering, 54:648–665, (2008).

[20] Ke, L., Zhai, L., Li, J., Chan, F. T. S. *Pareto mimic algorithm: an approach to the team orienteering problem.* Omega, 61:155-166.

[21] Lin S-W., Yu V.F. *Solving the team orienteering problem using effective multi-start simulated annealing.* Applied Soft Computing, 37:632-642. (2015)

[22] Souffriau W., Vansteenwegen P., Vanden Berghe G. y Van Oudheusden D. *A path relinking approach for the team orienteering problem.* Computers and Operations Research, 37:1853–1859, (2010).

[23] Spears W. M. y De Jong K. A. *On the virtues of parameterized uniform crossover.* (1991).

[24] Tang H. y Miller-Hooks E. *A tabu search heuristic for the team orienteering problem.* Computers and Operations Research, 32:1379–1407, (2005).

[25] Tsiligirides, T. *Heuristic Methods Applied to Orienteering.* Journal of the Operational Research Society, 35(9), 797-809, (1984).

[26] Vansteenwegen P., Souffriau W., Vanden Berghe G. y Van Oudheusden D. *A guided local search metaheuristic for the team orienteering problem.* European Journal of Operational Research, 196:118–127, (2009).