



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

Director: Loiseau, Irene
Buenos Aires, 2018

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

En el *Orienteering Problem* (OP), 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 $beneficio(nodo_i) \geq 0$ 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} . En el artículo *Heuristic Methods Applied to Orienteering* de Tsiligridis [22] se llama a este problema como *Generalized Traveling Salesman Problem* (GTSP).

El *Team Orienteering Problem* (TOP) 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 solo hay una ruta. Resolver el TOP requiere determinar el orden en que se visitan los nodos y además, seleccionar qué subconjunto de nodos a visitar, ya que no necesariamente se visitan todos los nodos.

En esta tesis propongo una combinación del *Biased Random Key Genetic Algorithm* (BRKGA) y de búsquedas locales para resolver el TOP. El BRKGA es un algoritmo genético 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 el BRKGA uno de los padres siempre pertenece al subconjunto de los mejores individuos de la población y este padre tiene mayor probabilidad de transmitir 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 siete conjuntos. Los primeros tres conjuntos de instancias son los de Tsiligridis [22] y los siguientes cuatro conjuntos son los de Chao et al. [9]. Todas las instancias pueden encontrarse en [17]. Mis resultados fueron comparados con los resultados

obtenidos por los siguientes autores: Chao, Golden y Wasil [9] (CGW), Tang y Miller-Hooks [21] (TMH), Archetti, Hertz, Speranza [1] (AHS), Ke, Archetti y Feng [18] (KAF) y Bouly, Dang y Moukrim [5] (BDM).

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

Palabras clave: Problema de Orientación de Equipo, Algoritmos Genéticos de Clave Aleatoria Sesgados, Problema de Enrutamiento, Búsqueda Local, Decodificador.

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

In the *Orienteering Problem* (OP), 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 $benefit(node_i) \geq 0$ y $benefit(node_1) = benefit(node_n) = 0$, 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} . In the article *Heuristic Methods Applied to Orienteering* by Tsiligirides [22] this problem is refers as the *Generalized Traveling Salesman Problem* (GTSP).

The *Team Orienteering Problem* (TOP) 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 *Traveling 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 thesis I propose a combination of the *Biased Random Key Genetic Algorithm* (BRKGA) and local searches to solve the TOP. The BRKGA is a genetic algorithm that initialize 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 the BRKGA 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 [22] and the other four sets are those of Chao et al. [9]. All instances can be found in [17]. My results were compared with the results obtained by the following authors: Chao, Golden and Wasil [9] (CGW), Tang and Miller-Hooks [21] (TMH), Archetti, Hertz and Speranza [1] (AHS), Ke, Archetti and Feng [18] (KAF) and Bouly, Dang and Moukrim [5] (BDM).

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

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

Índice general

1..	Introducción	1
2..	Revisión Bibliográfica	4
3..	Modelo Matemático	7
4..	Biased Random Key Genetic Algorithm	9
4.1.	Algoritmos Genéticos	9
4.2.	Random Key Genetic Algorithm	9
4.3.	Biased Random Key Genetic Algorithm	10
4.4.	Decodificador del BRKGA	11
5..	Desarrollo del algoritmo BRKGA con búsqueda local para el TOP	13
5.1.	Decodificador	15
5.1.1.	Orden en que los clientes se intentan agregar a las rutas.	15
5.1.2.	Decodificador Simple	16
5.1.3.	Características y debilidades del decodificador simple	18
5.1.4.	Decodificador Goloso	19
5.2.	Biased Random Key Genetic Algorithms	21
5.2.1.	Configuración	22
5.2.2.	Descripción y codificación de las propiedades del objeto configuración	22
5.2.3.	Inicialización de la Población	24
5.2.4.	Condición de parada	24
5.2.5.	Evolución de la población	25
5.2.6.	Resultados de la primer versión	28
5.3.	Búsquedas Locales	30
5.3.1.	Centro de Gravedad	30
5.3.2.	Swap	31
5.3.3.	Insert	33
5.3.4.	2-Opt	34
5.3.5.	Replace Simple	36
5.3.6.	Replace Multiple	38
5.3.7.	Encoder	40
5.3.8.	Orden de ejecución de las búsquedas locales	43
6..	Resultados	46
7..	Conclusiones	52
7.1.	Trabajos Futuros	52

1. INTRODUCCIÓN

La orientación es un deporte al aire libre normalmente jugado 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 comienza 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 haya una superposición mínima en los puntos visitados por cada miembro del equipo, 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.

El TOP ha sido reconocido como un 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 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.
- IV 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.

- v 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.
- VI El enrutamiento de técnicos para atender a los clientes en ubicaciones geográficamente distribuidas. En este contexto, cada vehículo en el modelo TOP representa un solo técnico y hay a menudo una limitación en el número de horas que cada técnico puede programar para trabajar en un día dado. Por lo tanto, puede no ser posible incluir a todos los clientes que requieren servicio en los horarios de los técnicos para un día determinado. En su lugar, se seleccionará un subconjunto de los clientes. Las decisiones sobre qué clientes elegir para su inclusión en cada uno de los horarios de los técnicos de servicio pueden tener en cuenta la importancia del cliente o la urgencia de la tarea. Este requisito de selección de clientes también surge en muchas aplicaciones de enrutamiento en tiempo real.

Para la generación y comparación de resultados se utilizaron instancias de test de Tsiligrídes y de Chao [17]. 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 tienen un set de coordenadas X e Y que representan su ubicación en un plano cartesiano.
- Un nodo de inicio y fin de ruta. Todos los vehículos comparten 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:

- Todas las instancias del benchmark utilizan un plano cartesiano y se utiliza la distancia euclidiana para medir distancias.
- Una solución es valida 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.

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, es el mismo que presentaron Tang y Miller-Hooks [21] . En el capítulo 4 se explica sobre 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 el algoritmo BRKGA, detallando las configuraciones testeadas, resultados parciales y problemas encontrados. Por ultimo en este mismo capitulo se describen las búsquedas locales implementadas, el objetivo de cada una, los distintos ordenes 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 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 modeladas 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. 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] 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 introdujeron 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] 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]. 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 *Tabu Search* (TS) demostró poder resolver TOP como es el caso del trabajo de Tang y Miller-Hooks (TMH) [21]. Su TS 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. Paso A, iniciación desde el AMP: dada la solución actual s determinada en el AMP, establece los parámetros tabú a una pequeña etapa del vecindario en la que solo

se explorará una pequeña cantidad de soluciones de vecindario. Paso B, mejora: genera mediante procedimientos aleatorios y golosos una cantidad de soluciones de vecindario (válidas e inválidas) a la solución actual s en función de los parámetros tabú actuales. En iteraciones selectas, la secuencia de cada una de estas soluciones de vecindario se mejora mediante procedimientos heurísticos. Paso C, evaluación: Se selecciona la mejor solución que no sea tabú entre los candidatos generados en el paso B. El estado tabú puede anularse si la mejor solución tabú es mejor que la mejor solución factible actual. Dependiendo del tamaño actual del vecindario y la calidad de la solución, se establece el parámetro del tamaño del vecindario en etapas grandes o pequeñas y regresa al paso A o al B. Como señala Golden et al. [15], 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. Sus resultados de experimentos computacionales realizados sobre el mismo conjunto de problemas de Chao et al. muestran que la técnica propuesta produce consistentemente soluciones de alta calidad superando las propuestas publicadas hasta el momento.

Archetti et al. [1] proponen dos variantes de un algoritmo de un TS generalizado y de un algoritmo llamado *Variable Neighborhood Search* (VNS). El VNS parte de una solución titular s , desde donde dan 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. [18] proponen un *Ant colony Optimization* (ACO) que utiliza cuatro métodos diferentes para construir soluciones candidatas. El ACO 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 ACO es construir soluciones candidatas, Ke et al. proponen cuatro métodos: secuencial, determinista-concurrente, aleatorio-concurrente y simultáneo.

Los autores Vansteenwegen et al. [23], 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. [19] combinan un *Greedy Randomised Adaptive 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 está 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] idearon un *Memetic Algorithm* (MA) para resolver el TOP. Los MA 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] proponen un *Particle Swarm Optimization based Memetic Algorithm* (PSOMA) para resolver el TOP. Su algoritmo PSOMA provee de soluciones de alta calidad para el TOP. El algoritmo está relativamente cerca del MA 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. El *Particle Swarm Optimization* (PSO) 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] implementan un *Genetic Algorithm* (GA) 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] y Bouly et al. [5].

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). Entre todos los trabajos mencionados decidí comparar mi resultados con los obtenidos por el MA de Bouly et al. [5], el VNS_{slow} de Archetti et al. [1] y el ACO_{seq} de Ke et al. [18] por que estos trabajos obtuvieron muy buenos resultados y plantean algoritmos diversos entre sí.

3. MODELO MATEMÁTICO

La formulación para el TOP donde el punto de inicio y fin son el mismo ha sido presentado por Tang H. y Miller-Hooks E. [21]. Tal formulación puede ser extendida al caso donde el punto de inicio y fin pueden ser diferentes. A continuación, la formulación matemática extendida del TOP presentada por Ke L., Archetti C. y Feng Z. [18].

Dado un grafo completo $G = (V, E)$ donde $V = \{1, \dots, n\}$ 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 que toma en 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 $y_{ik} = 1$ ($i = 1, \dots, m$) si el eje (i, j) es visitado por el vehículo k , sino $y_{ik} = 0$. Sea $x_{ijk} = 1$ ($i, j = 1, \dots, n; k = 1, \dots, m$) si el eje (i, j) es visitado por el vehículo k , sino $x_{ijk} = 0$. Como $c_{ij} = c_{ji}$ solo x_{ijk} ($i < j$) se define. Sea U un subconjunto de V . Luego TOP puede ser descrito de la siguiente manera:

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

$$\text{sujeeto a } \sum_{j=2}^n \sum_{k=1}^m x_{1jk} = \sum_{i=1}^{n-1} \sum_{k=1}^m x_{ink} = m \quad (3.2)$$

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

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

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

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

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

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

Donde 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 lo sumo una vez. La restricción 3.5 describe la limitación de distancia. La restricción 3.6 asegura que los sub-caminos están prohibidos. La restricción 3.7 y 3.8 establecen el requerimiento integral en cada variable.

4. BIASED RANDOM KEY GENETIC ALGORITHM

4.1. Algoritmos Genéticos

Los *Genetic Algorithms* (GA) [14] aplican 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

El *Random Key Genetic Algorithm* (RKGA) fue introducido por Bean en *Genetic algorithms and random keys for sequencing and optimization*. [4]. En 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 [20]), 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 $i = 1, \dots, n$ 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 $\rho_e > 0,5$, entonces $\rho_e > 1 - \rho_e$, por lo tanto es más probable que el individuo resultante herede características del padre de élite que las del padre de no-élite. Dado que asumimos que cualquier vector de números reales aleatorios puede ser decodificado en una solución, 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, requiere que exista un objeto codificador para actualizar el vector de números reales de la solución mejorada. Es decir, 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 BRKGA 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 mostré 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 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 [17] 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:

$$i_e(brkga_{v,x}, ins_n) = benefit(brkga_{v,x}(ins_n)) / bestBenefitFor(ins_n) \quad (5.1)$$

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 y $bestBenefitFor(ins_n)$ representa el mejor beneficio obtenido para la misma instancia en los trabajos previamente publicados. 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 implementación 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 cual implementé el desarrollo. El objetivo del pseudocódigo es facilitar el entendimiento del funcionamiento de los algoritmos que describiré.

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

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. Esto 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. Esto es importante mencionarlo ya que ayuda a entender los resultados parciales que obtuve sobre las seis instancias seleccionadas. Los nombres de las instancias nos dan información sobre el conjunto del benchmark al cual pertenecen y la cantidad de vehículos que tienen. Por ejemplo la instancia *p2.3.g* tiene pertenece al conjunto número 2 del benchmark y tiene 3 vehículos. La instancia *p2.2.k* también pertenece al conjunto número 2 del benchmark, lo que significa que tiene los mismos clientes que *p2.3.g*.

Las tablas donde mostraré el i_e y el beneficio de los resultados parciales tendrá un subconjunto de los siguientes encabezados:

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.
- B_{max} : El **B**eneficio máximo que obtuve para la instancia mencionada.
- B_{min} : El **B**eneficio mínimo que obtuve para la instancia mencionada.
- B_{avg} : El **B**eneficio promedio que obtuve para la instancia mencionada.
- i_{eMax} : Índice de efectividad máximo. Utiliza mi beneficio máximo obtenido para la instancia mencionada.

- i_{eAvg} : Índice de efectividad promedio. Utiliza mi beneficio promedio obtenido para la instancia mencionada.
- *Best*: Máximo beneficio obtenido por alguna 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. Este es el orden por el cual se tomarán 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* recién 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 vehículos: 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. 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, que retorna *true* cuando c_i se puede agregar a la ruta y *false* en caso contrario

Formula matemática que modela el método *CanVisit*.

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

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 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 solución 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	40	102	175	0.37	0.64	275
p2.3.g	21/3/10.70	45	83	140	0.57	0.97	145
p3.4.p	33/4/22.50	90	170	270	0.30	0.48	560
p5.3.x	66/3/40.00	195	295	405	0.19	0.26	1555
p7.2.e	102/2/50.00	8	39	98	0.13	0.34	290
p7.4.t	102/4/100.00	40	116	221	0.11	0.21	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 numero 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 $O(\#clientes + \#vehiculos)$. En la práctica nunca se llega a visitar a todos los clientes ya que cambia de vehículo en cuanto encontró un cliente que no logro insertar en su ruta. Por lo tanto en la practica 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 $\#Poblacion$ 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 $i_0 = 0, i_1, i_2, \dots, i_m$ 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 tiempo, 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

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

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.

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

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 con el 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 $O(\#clientes * \#vehiculos)$. 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. Generé otros 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 ultimo falla con el cliente 7.

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

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

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	95	164	260	0.60	0.95	275
p2.3.g	21/3/10.70	95	122	140	0.84	0.97	145
p3.4.p	33/4/22.50	180	288	410	0.51	0.73	560
p5.3.x	66/3/40.00	305	412	525	0.26	0.34	1555
p7.2.e	102/2/50.00	31	96	163	0.33	0.56	290
p7.4.t	102/4/100.00	160	280	438	0.26	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 la sección BRKGA (ver sección 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 impactar 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 estaba probando. En otras palabras incrementé mi capacidad de monitoreo y control del desarrollo. El pseudocódigo 5.6 muestra todas las propiedades configurables del mi desarrollo.

Pseudocódigo 5.6: Objeto *Configuración* 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 del objeto configuración

Como mencioné, uno de los beneficios de tener todas las variables configurables en un solo objeto es poder leer rápidamente que BRKGA estoy probando. Es que le agregue la propiedad *Description* que se instancia automáticamente al instanciar el objeto *Configuration*. Su valor es del tipo *string* y codifica el valor del resto de las propiedades del objeto *Configuration*. A continuación explico las propiedades del objeto Configuración:

- **Description:** Es una especie de hash descriptivo de la instancia del objeto *Configuration*. Codifica los valores del resto de las propiedades utilizando su clave y valor.
- **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: Generación 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. 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 la sección BRKGA (ver 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 generaron 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*. En el pseudocódigo 5.11 observamos el proceso de apareamiento como lo describí en la sección 4.3. 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 decimal aleatorio en el intervalo $[0, 1]$ y su comparación con la propiedad *EliteGenChance*.

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);
    }
}
```

Como mencioné en el proceso de evolución, mi implementación verifica que no se inserten soluciones duplicadas en la población. Verificar si dos soluciones son iguales tiene un costo temporal muy bajo en BRKGA. Como mencioné anteriormente 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 esto no requiere comparar las rutas de dos soluciones para saber si son idénticas, es suficiente con comparar el orden que de los clientes que determinan sus vectores de *RandomKeys*. En el pseudocódigo 5.13 podemos ver como se crea el hash de una solución 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 concatena con un símbolo separador. El hash se calcula una sola vez en el momento que se instancia la solución y su complejidad es de $O(\text{clientes} * \log(\text{clientes}))$. Comparar dos hashes tiene una complejidad $O(1)$ ya que simplemente es comparar dos *strings*. Por lo tanto verificar que la solución que se va a insertar no existe en la población tiene un complejidad de $O(\text{población})$. En la figura 5.4 podemos ver el ejemplo del hash de un vector de *RandomKeys*.

Decidí no reintentar el apareamiento entre los dos padres que generaron la solución repetida, porque consideré que la probabilidad de volver a generar nuevamente una solución existente es alta cuando ya sucedió una vez. Tomé la decisión de reemplazar por un mutante la solución repetida para optimizar el tiempo de ejecución del apareamiento y disminuir la cantidad de soluciones dentro de un mismo vecindario por generación.

Fig. 5.4: Vector de *RandomKeys* ordenado y su hash.

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

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

Pseudocódigo 5.13: Generación del hash de una solución.

```

private string pseudoHash;
public string GetHash()
{
    if (!string.IsNullOrEmpty(pseudoHash))
        return pseudoHash;

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

```

En una primera instancia se insertaban los individuos sin verificar la existencia de un 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 bajo 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 ampliamente la cantidad de soluciones diferentes exploradas, por ende reduce la frecuencia con la que una nueva mejor solución es generada. Además el algoritmo se vuelve mas lento ya que repite cálculos en donde obtiene los mismos resultados. Lo que significa que el costo temporal total de validar unicidad en la inserción, que conlleva un orden de complejidad $O(población * población.NoElite)$ por evolución, resulta muy bajo comparado con el costo de trabajar con múltiples soluciones duplicadas.

5.2.6. 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

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

De estos primeros resultados podemos ver que el BRKGA puro funciona muy bien para instancias de testeo pequeñas. 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 soluciones 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 experimentos 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. Búsquedas Locales

Con el objetivo de optimizar los resultados obtenidos hasta el momento, implementé algunas búsquedas locales. La idea fue aplicar estas 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. Implementé el cálculo del COG de la forma que lo describen Vansteenwegen et al. [23]. La coordenada del COG de una ruta se calcula como muestran las formulas 5.4 y 5.5.

Coordenada X del COG de una ruta.

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

Coordenada Y del COG de una ruta.

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

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

Sea r una ruta:

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

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

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

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

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

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 actualización 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 formula 5.6. La formula 5.7 muestra como se actualiza el COG en $O(1)$ luego de remover un cliente utilizando los valores de $x_{cog}.num$ y $x_{cog}.den$. Ídem formula 5.8 para el caso en que se inserta un cliente.

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 formulas 5.9, 5.10 y 5.11.

$$r_a.Dist + r_b.Dist < r'_a.Dist + r'_b.Dist \quad (5.9)$$

$$r'_a.Dist \leq v'_a.d_{max} \quad (5.10)$$

$$r'_b.Dist \leq v'_b.d_{max} \quad (5.11)$$

En el pseudocódigo 5.14 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 5.15, 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 5.14: 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 5.15: El método *SwapDestinationsBetween*, prueba intercambiar todos los clientes entre dos rutas.

```
public bool SwapDestinationsBetween(Vehicle v1, Vehicle v2)
{
    var changed = false;
    var v1Bans = new Dictionary<int, bool>();
    var v2Bans = new Dictionary<int, bool>();
    for (var i = 0; i < v1.Route.RouteLenght(); i++)
    {
        if (v1Bans.ContainsKey(i))
            continue;
        for (var j = 0; j < v2.Route.RouteLenght(); j++)
        {
            if (v2Bans.ContainsKey(j))
                continue;
            if (!Swaps(i, j, ref leftRoute, ref rightRoute))
                continue;
            changed = true;
            v1Bans.Add(i, true);
            v2Bans.Add(j, true);
            break; // Para que cambie i
        }
    }
    return changed;
}
```

El orden de complejidad del método *ApplyLocalSearch* de la clase *SwapHeuristic* es:

$$O((n * (n - 1)/2) * \text{clientes}/n * \text{clientes}/n) \approx O(\text{clientes}^2/2)$$

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 límite 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 5.16: 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 = AnalyzeInsert(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 5.16 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 *AnalyzeInsert* 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 método *ApplyLocalSearch* del *Insert* es:

$$O(\text{vehiculos} * \text{clientesNoVisitados} * \text{mediaClientesEnRuta})$$

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 5.16 se aplica el método *Do2OptSwap* a cada vehículo de la solución.

Pseudocódigo 5.17: 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 todas las posibles combinaciones 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 5.18 muestra como implementé el *2-Opt*.

Pseudocódigo 5.18: 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].Position1;
        var pos2 = combinations[index].Position2;
        var swaped = vehicle.Route.SwapIfImproves(pos1, pos2);
        if (swaped)
        {
            changed = true;
            index = 0;
        }
        else
            index++;
    }
    return changed;
}

```

Este algoritmo tiene un orden de complejidad:

$$\begin{aligned} O(2Opt) &= O(\text{vehículos} * \text{mediaClientesEnRuta} * (\text{mediaClientesEnRuta} - 1)/2) \\ &= O(\text{vehículos} * \text{mediaClientesEnRuta}^2/2) \end{aligned}$$

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 5.19 vemos que se aplica el *Replace Simple* a cada vehículo de la solución.

Pseudocódigo 5.19: Método *ApplyLocalSearch* del *Replace Simple*.

```
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 5.20.

Pseudocódigo 5.20: Método *Replace*.

```

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

    foreach (var client in uClients)
    {
        var res = AnalyzeInsert(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 5.21, 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 5.21: 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;
}
```

El *Replace Simple* tiene un orden de complejidad:

$$O(\text{ReplaceSimple}) = O(\text{vehículos} * \text{clientesNoVisitados} * \text{mediaClientesVisitados} * 2)$$

5.3.6. Replace Multiple

El *Replace Multiple* se diferencia del *Replace Simple* de dos formas. La primera 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* 5.20 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* 5.22 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 5.22 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 5.22: 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;
}
```

El pseudocódigo 5.23 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 en la ruta del vehículo. El peor caso sucede cuando el cliente insertado tiene un beneficio mayor a la sumatoria de beneficios de los clientes visitados. Por lo tanto el orden de complejidad de *CandidateGroup* es de $O(2^{\text{clientesVisitados}})$ que es muy alto. De todos modos el peor escenario tiene una probabilidad muy baja de suceder, entonces el la practica no es tan lento.

Pseudocódigo 5.23: 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 *Replace Multiple* mejoro considerablemente el i_{eAvg} en instancias grandes ya que explora muchas mas 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:

$$O(\text{ReplaceMultiple}) = O(\text{vehículos} * \text{clientesNoVisitados} * 2^{\text{clientesVisitados}})$$

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 sea valida la ecuación 5.12.

Ecuación que debe ser válida para que el *crossover* sea correcto.

$$\text{ApplyLocalSearch}(s) = \text{Decoder.Decode}(s.\text{RandomKeys}, \text{ProblemInfo}) \quad (5.12)$$

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 por 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 5.5 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 ultimo las rutas que fueron generadas por el decodificador goloso (los mismo clientes que estaban coloreados en el vector de *RandomKeys*).

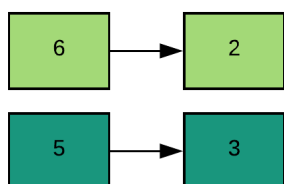
Fig. 5.5: Dado un *RandomKeys*, se obtiene un hash y una solución generada por el decodificador goloso.

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@1@4@8@3@7

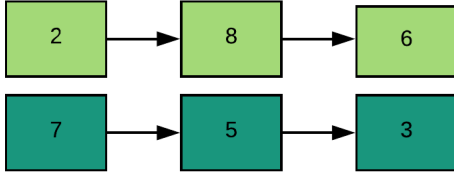
El decodificar goloso genera una solución que contiene las rutas:



A la solución de la figura 5.5 se le aplican algunas búsquedas locales y podemos ver los cambios en la figura 5.6. Ambas rutas se extienden con un nuevo cliente, la primera 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 que al decodificar el vector de *RandomKeys* genere la solución optimizada.

Fig. 5.6: Se aplican búsquedas locales a la solución de la figura 5.5, 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@1@4

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 5.6 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 primera ruta pertenece a otro vehículo. En el ejemplo de la figura 5.6 podría ser el caso los ClientId: 7, 5 o 3. Si sucede esto, el segundo vehículo tendría mas 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 5.6 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 valida pero el beneficio total de la solución resultante es igual o mayor a la que debería ser. Entonces no es un problema grave, 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 asegurando la valides de la ecuación 5.12. Podemos observar el pseudocódigo 5.24 que actualiza el vector de *RandomKeys* luego de que la solución es optimizada por las búsquedas locales.

Pseudocódigo 5.24: 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 = ClientIds[index],
            ForceVehicleChangeAfterThis = endRoute
        };
        newRandomKeys.Add(randomKey);
    }
    s.SetRandomKeys(newRandomKeys);
    return s;
}

```

5.3.8. Orden de ejecución de las búsquedas locales

Cuando se crea una instancia de mi algoritmo BRKGA, se setea una lista de búsquedas locales. En el momento que se realicen las búsquedas locales a una determinada solución, se ejecutaran todas las búsquedas locales de tal lista en el mismo orden que se encuentran en la lista. Por ejemplo, si $lista = (Swap, Insert, 2 - Opt, Insert)$, entonces primero se aplicara *Swap*, seguido de *Insert* y finalmente *Replace Multiple*. Esto me permite probar

múltiples combinaciones de búsquedas locales modificando solamente una lista. 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. Esta lista la agregue al objeto configuración del BRKGA.

Con el objetivo de encontrar la mejor secuencia de búsquedas locales a aplicar, generé múltiples instancias de mi implementación con distintas listas de búsquedas locales. Cada una de estas instancias la ejecute diez veces y persistí los resultados. Todas las instancias comparten los mismos valores en el resto de su configuración, de forma que los resultados solo varíen por las búsquedas locales aplicadas. La configuración básica que comparten es: MI.200;MNC.50;PS.150;EP.0,3;MP.0,1;EGC.70;TOP.2.

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 5.6, se pueden observar los resultados de variar el orden en que se ejecutan las búsquedas locales. Claramente el peor orden de búsquedas locales es IRsRmOS (*Insert, Replace Simple, Replace Multiple, 2-Opt, Swap*). Esto sucede porque *Insert* y *Replace* intentan agregar mas clientes o intercambiar por clientes mas rentables mejorando el beneficio de la ruta. Mientras que *2-Opt* y *Swap* hacen un reordenamiento de la secuencia en que se visitan los clientes seleccionados, luego minimizan la distancia recorrida de una ruta. Al ejecutar primero las búsquedas que incrementan el beneficio y luego las que decrementan la distancia recorrida, no hacemos uso de la distancia disminuida por *2-Opt* y *Swap*.

Luego el mejor orden según estos resultados, fue SIORSOR. Notar que repito búsquedas en esta secuencia, en una primera impresión parece redundante pero no lo es. Por ejemplo, la primera vez que se aplica *2-Opt* se hace luego de un *Insert* que utiliza toda la distancia disponible y la segunda vez que se aplica el *2-Opt* es posterior de un *Swap*.

Sobre T_{avg} , notar entre la configuración 2 y 3 el tiempo puede llegar a ser más del doble y la única diferencia es que se uso el decodificador simple versus el goloso. Esto lo logra con un i_eAvg prácticamente igual en todas las instancias. Lo mismo podemos observar entre la configuración 4 y 5. Por lo tanto, habiendo agregado las búsquedas locales, el decodificador goloso no mejora el beneficio final y el decodificador simple reduce el tiempo de ejecución. Por este motivo para los resultados finales utilicé el decodificar simple.

Estos resultados me parecieron muy buenos ya que en dos de las seis instancias se llega a la mejor solución conocida y en los resultados de las otras cuatro instancias son competitivos con los trabajos previos de la literatura. Por eso decidí correr mi algoritmo para el resto de las instancias del benchmark.

Tab. 5.6: Resultados de distintos ordenes de búsquedas locales.

I	$\#N/V/tMax$	Orden	D	t_{avg}	B_{min}	B_{avg}	B_{max}	i_{eMed}	$Best$
p2.2.k	21/2/22.50	IRsRmOS	G	4002	270	271	275	0.99	275
p2.2.k	21/2/22.50	SIORsSORm	G	4118	270	272	275	0.99	275
p2.2.k	21/2/22.50	SIORsSORm	S	2613	270	272	275	0.99	275
p2.2.k	21/2/22.50	SOIRsRm	G	3904	270	273	275	0.99	275
p2.2.k	21/2/22.50	SOIRsRm	S	2294	270	274	275	1.00	275
p2.2.k	21/2/22.50	SOISORsRm	G	4089	270	273	275	0.99	275
p2.3.g	21/3/10.70	IRsRmOS	G	2181	145	145	145	1.00	145
p2.3.g	21/3/10.70	SIORsSORm	G	2293	145	145	145	1.00	145
p2.3.g	21/3/10.70	SIORsSORm	S	1880	145	145	145	1.00	145
p2.3.g	21/3/10.70	SOIRsRm	G	2649	145	145	145	1.00	145
p2.3.g	21/3/10.70	SOIRsRm	S	1777	145	145	145	1.00	145
p2.3.g	21/3/10.70	SOISORsRm	G	2765	145	145	145	1.00	145
p3.4.p	33/4/22.50	IRsRmOS	G	10816	530	542	550	0.97	560
p3.4.p	33/4/22.50	SIORsSORm	G	11438	540	553	560	0.99	560
p3.4.p	33/4/22.50	SIORsSORm	S	4760	550	558	560	1.00	560
p3.4.p	33/4/22.50	SOIRsRm	G	10723	540	550	560	0.98	560
p3.4.p	33/4/22.50	SOIRsRm	S	4618	550	555	560	0.99	560
p3.4.p	33/4/22.50	SOISORsRm	G	11620	550	558	560	1.00	560
p5.3.x	66/3/40.00	IRsRmOS	G	35470	1455	1471	1490	0.95	1555
p5.3.x	66/3/40.00	SIORsSORm	G	37699	1485	1495	1525	0.96	1555
p5.3.x	66/3/40.00	SIORsSORm	S	16790	1485	1497	1515	0.96	1555
p5.3.x	66/3/40.00	SOIRsRm	G	29260	1485	1506	1525	0.97	1555
p5.3.x	66/3/40.00	SOIRsRm	S	12601	1490	1508	1535	0.97	1555
p5.3.x	66/3/40.00	SOISORsRm	G	34053	1485	1502	1515	0.97	1555
p7.2.e	102/2/50.00	IRsRmOS	G	13134	290	290	290	1.00	290
p7.2.e	102/2/50.00	SIORsSORm	G	13259	290	290	290	1.00	290
p7.2.e	102/2/50.00	SIORsSORm	S	7500	290	290	290	1.00	290
p7.2.e	102/2/50.00	SOIRsRm	G	12695	290	290	290	1.00	290
p7.2.e	102/2/50.00	SOIRsRm	S	6828	290	290	290	1.00	290
p7.2.e	102/2/50.00	SOISORsRm	G	13658	290	290	290	1.00	290
p7.4.t	102/4/100.00	IRsRmOS	G	60299	984	1013	1034	0.94	1077
p7.4.t	102/4/100.00	SIORsSORm	G	68191	1006	1029	1047	0.96	1077
p7.4.t	102/4/100.00	SIORsSORm	S	37733	1019	1037	1063	0.96	1077
p7.4.t	102/4/100.00	SOIRsRm	G	63010	1023	1033	1051	0.96	1077
p7.4.t	102/4/100.00	SOIRsRm	S	31241	1020	1035	1050	0.96	1077
p7.4.t	102/4/100.00	SOISORsRm	G	67730	1012	1030	1052	0.96	1077

6. RESULTADOS

En este capítulo mostraré los resultados finales obtenidos. Dentro de todos los trabajos previos de la literatura que encontré y que incluyeran resultados que utilizaran el benchmark de instancias de Chao y los de Tsiligrides [17], seleccioné los resultados de Archetti et al. [1], los de Chao et al. Chao [9] y los de Tang et al. [21]. Dentro de las soluciones propuestas de Archetti et al., seleccione los resultados de su 'Slow VNS Feasible' ya que tenía los mejores resultados al correrlos con el benchmark de instancias. Seleccione estos trabajos porque plantean distintos enfoques y tienen los mejores resultados para el benchmark de problemas utilizado.

La configuración final utilizada para el BRKGA fue:

- **MinIterations:** 200
- **MinNoChanges:** 50
- **PopulationSize:** 150
- **ElitePercentage:** 0.3
- **MutantPercentage:** 0.1
- **EliteGenChance:** 70
- **Heuristics:** Swap, Insert, 2-Opt, Replace, Swap, 2-Opt, Replace
- **ApplyHeuristicsToTop:** 2
- **DecoderType:** Simple

Además agregué un último paso una vez que corta el algoritmo, antes de entregar la mejor solución encontrada se le aplica una secuencia larga de búsquedas locales distinta a la aplicada a los mejores individuos de cada generación.

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 Enterprise 2015
- Lenguaje: C# .Net Framework 4.5

A continuación los resultados de 199 instancias del bechmark para mi algoritmo, el de Archetti, Hertz y Speranza(AHS), Tang y Miller-Hooks(TMh) y Chao, Golden, y Wasi (CGW). En el caso de mi algoritmo, cada instancia se ejecuto diez veces y muestro B_{min} , B_{avg} y B_{max} denotando la peor solución, la solución media y la mejor solución encontrada respectivamente entre las diez ejecuciones.

I	$\#N$	$\#V$	$tMax$	t_{avg}	B_{min}	B_{avg}	B_{max}	AHS	CGW	TMH
p2.2.k	21	2	22.50	2577	260	268	270	275	270	270
p2.3.g	21	3	10.70	1608	145	145	145	145	140	140
p2.3.h	21	3	11.70	1774	165	165	165	165	165	165
p1.2.i	32	2	23.00	4075	130	134	135	135	130	135
p1.2.l	32	2	30.00	4929	190	190	195	195	190	190
p1.3.h	32	3	13.30	2260	70	70	70	70	75	70
p1.3.m	32	3	21.70	4343	170	173	175	175	175	170
p1.3.o	32	3	24.30	4573	195	200	205	205	215	205
p1.3.p	32	3	25.00	5067	210	215	220	220	215	220
p1.4.j	32	4	12.50	2240	75	75	75	75	70	75
p1.4.o	32	4	18.20	3765	165	165	165	165	160	165
p1.4.p	32	4	18.80	3821	175	175	175	175	160	175
p3.2.c	33	2	12.50	2017	180	180	180	180	170	180
p3.2.e	33	2	17.50	3293	260	260	260	260	260	250
p3.2.f	33	2	20.00	3339	300	300	300	300	300	290
p3.2.g	33	2	22.50	3726	350	354	360	360	350	350
p3.2.h	33	2	25.00	3931	390	390	390	410	390	410
p3.2.i	33	2	27.50	4056	440	447	460	460	440	460
p3.2.j	33	2	30.00	4358	500	500	500	510	470	490
p3.2.k	33	2	32.50	4627	540	549	550	550	540	540
p3.2.m	33	2	37.50	5183	610	619	620	620	620	620
p3.2.n	33	2	40.00	5660	630	645	660	660	660	660
p3.2.o	33	2	42.50	5636	670	672	680	690	680	690
p3.2.p	33	2	45.00	6185	700	709	720	720	710	710
p3.2.q	33	2	47.50	6097	730	734	750	760	750	760
p3.2.r	33	2	50.00	6659	750	764	780	790	780	780
p3.3.k	33	3	21.70	4060	430	433	440	440	430	430
p3.3.l	33	3	23.30	4295	470	474	480	480	470	470
p3.3.m	33	3	25.00	4426	500	504	510	520	520	510
p3.3.n	33	3	26.70	4114	550	564	570	570	550	550
p3.3.o	33	3	28.30	4241	580	583	590	590	580	590
p3.3.p	33	3	30.00	4405	620	628	640	640	620	640
p3.3.q	33	3	31.70	4484	650	662	670	680	630	680
p3.3.s	33	3	35.00	4850	710	712	720	720	710	710
p3.3.t	33	3	36.70	5009	720	723	750	760	720	750
p3.4.f	33	4	10.00	1808	190	190	190	190	180	190
p3.4.i	33	4	13.80	2483	270	270	270	270	260	260
p3.4.j	33	4	15.00	2534	300	305	310	310	300	310
p3.4.m	33	4	18.80	3442	390	390	390	390	380	380
p3.4.o	33	4	21.20	3797	480	489	500	500	490	490

<i>I</i>	<i>#N</i>	<i>#V</i>	<i>tMax</i>	<i>tavg</i>	<i>B_{min}</i>	<i>B_{avg}</i>	<i>B_{max}</i>	<i>AHS</i>	<i>CGW</i>	<i>TMH</i>
p3.4.p	33	4	22.50	4125	540	549	560	560	530	560
p6.2.j	64	2	30.00	12376	936	937	942	948	942	936
p6.2.l	64	2	35.00	15565	1086	1095	1104	1116	1104	1116
p6.2.m	64	2	37.50	16469	1152	1160	1170	1188	1176	1188
p6.2.n	64	2	40.00	15984	1206	1221	1230	1260	1242	1260
p6.3.i	64	3	18.30	9341	636	638	642	642	642	612
p6.3.k	64	3	21.70	11270	876	886	894	894	894	876
p6.3.l	64	3	23.30	11461	960	976	984	1002	972	990
p6.3.m	64	3	25.00	12060	1056	1066	1074	1080	1080	1080
p6.3.n	64	3	26.70	12674	1146	1153	1170	1170	1158	1152
p6.4.k	64	4	16.20	7011	522	526	528	528	546	522
p6.4.l	64	4	17.50	9316	690	691	696	696	690	696
p5.2.e	66	2	12.50	3683	180	180	180	180	175	180
p5.2.g	66	2	17.50	7357	315	319	320	320	315	320
p5.2.h	66	2	20.00	8407	390	393	400	410	395	410
p5.2.j	66	2	25.00	9663	560	577	580	580	580	560
p5.2.l	66	2	30.00	11332	740	754	770	800	790	770
p5.2.m	66	2	32.50	11001	825	838	860	860	855	860
p5.2.n	66	2	35.00	11990	890	904	925	925	920	920
p5.2.o	66	2	37.50	12420	960	973	985	1020	1010	975
p5.2.p	66	2	40.00	12641	1050	1067	1080	1150	1150	1090
p5.2.q	66	2	42.50	14157	1110	1128	1150	1195	1195	1185
p5.2.r	66	2	45.00	14695	1190	1211	1240	1260	1250	1260
p5.2.s	66	2	47.50	14831	1250	1280	1310	1340	1310	1310
p5.2.t	66	2	50.00	15616	1305	1341	1370	1400	1380	1380
p5.2.u	66	2	52.50	16996	1365	1395	1410	1460	1450	1445
p5.2.v	66	2	55.00	18688	1420	1441	1465	1505	1490	1500
p5.2.w	66	2	57.50	19290	1505	1507	1525	1560	1545	1560
p5.2.x	66	2	60.00	21139	1520	1543	1580	1610	1600	1610
p5.2.y	66	2	62.50	21692	1575	1594	1620	1635	1635	1630
p5.2.z	66	2	65.00	22642	1585	1613	1640	1670	1680	1665
p5.3.e	66	3	8.30	1654	95	95	95	95	110	95
p5.3.h	66	3	13.30	4279	260	260	260	260	255	260
p5.3.k	66	3	18.30	9420	485	491	495	495	480	495
p5.3.l	66	3	20.00	15294	570	577	585	595	595	575
p5.3.n	66	3	23.30	16880	745	753	755	755	755	755
p5.3.o	66	3	25.00	14868	830	850	870	870	870	835
p5.3.q	66	3	28.30	15596	1015	1036	1065	1070	1060	1065
p5.3.r	66	3	30.00	16032	1080	1096	1110	1125	1105	1115
p5.3.s	66	3	31.70	17088	1150	1167	1175	1190	1175	1175

<i>I</i>	# <i>N</i>	# <i>V</i>	<i>tMax</i>	<i>tavg</i>	<i>Bmin</i>	<i>Bavg</i>	<i>Bmax</i>	<i>AHS</i>	<i>CGW</i>	<i>TMH</i>
p5.3.t	66	3	33.30	18116	1215	1227	1240	1260	1250	1240
p5.3.u	66	3	35.00	18238	1280	1299	1320	1345	1330	1330
p5.3.v	66	3	36.70	18760	1360	1376	1390	1425	1400	1410
p5.3.w	66	3	38.30	18311	1405	1423	1445	1485	1450	1465
p5.3.x	66	3	40.00	19632	1470	1493	1515	1555	1530	1530
p5.3.y	66	3	41.70	20212	1505	1541	1560	1595	1580	1580
p5.3.z	66	3	43.30	19735	1560	1593	1625	1635	1635	1635
p5.4.m	66	4	16.20	11400	520	541	555	555	495	555
p5.4.o	66	4	18.80	13407	675	684	690	690	675	680
p5.4.p	66	4	20.00	17679	740	753	760	765	750	760
p5.4.q	66	4	21.20	15822	830	843	860	860	860	860
p5.4.r	66	4	22.50	16051	895	917	940	960	950	960
p5.4.s	66	4	23.80	15676	990	997	1000	1030	1020	1000
p5.4.t	66	4	25.00	16781	1090	1118	1160	1160	1160	1100
p5.4.u	66	4	26.20	16274	1170	1201	1240	1300	1260	1275
p5.4.v	66	4	27.50	16367	1245	1272	1300	1320	1310	1310
p5.4.w	66	4	28.80	17748	1330	1347	1370	1390	1380	1380
p5.4.x	66	4	30.00	17136	1400	1410	1420	1450	1420	1410
p5.4.y	66	4	31.20	17561	1455	1478	1500	1520	1490	1520
p5.4.z	66	4	32.50	17839	1510	1547	1585	1620	1545	1575
p4.2.a	100	2	25.00	5823	199	201	202	206	194	202
p4.2.c	100	2	35.00	14615	434	442	450	452	440	438
p4.2.d	100	2	40.00	17963	501	513	521	531	531	517
p4.2.e	100	2	45.00	19502	572	582	593	618	580	593
p4.2.f	100	2	50.00	21063	636	653	668	687	669	666
p4.2.g	100	2	55.00	22950	694	705	715	753	737	749
p4.2.h	100	2	60.00	26489	745	770	795	835	807	827
p4.2.i	100	2	65.00	27996	810	826	852	918	858	915
p4.2.j	100	2	70.00	30658	855	885	959	962	899	914
p4.2.k	100	2	75.00	32992	899	930	959	1022	932	963
p4.2.l	100	2	80.00	35127	935	971	1018	1074	1003	1022
p4.2.m	100	2	85.00	40129	1000	1016	1043	1132	1039	1089
p4.2.n	100	2	90.00	42753	1036	1064	1098	1171	1112	1150
p4.2.o	100	2	95.00	48741	1081	1105	1130	1218	1147	1175
p4.2.p	100	2	100.00	51276	1130	1154	1192	1241	1199	1208
p4.2.q	100	2	105.00	56951	1167	1188	1213	1263	1242	1255
p4.2.r	100	2	110.00	62964	1204	1216	1229	1286	1199	1277
p4.2.s	100	2	115.00	68578	1235	1243	1254	1301	1286	1294
p4.2.t	100	2	120.00	73271	1241	1267	1292	1306	1299	1306
p4.3.c	100	3	23.30	2400	193	193	193	193	191	192

<i>I</i>	<i>#N</i>	<i>#V</i>	<i>tMax</i>	<i>tavg</i>	<i>B_{min}</i>	<i>B_{avg}</i>	<i>B_{max}</i>	<i>AHS</i>	<i>CGW</i>	<i>TMH</i>
p4.3.d	100	3	26.70	6868	323	329	332	335	333	333
p4.3.e	100	3	30.00	9120	445	453	461	468	432	465
p4.3.f	100	3	33.30	12305	538	545	555	579	552	579
p4.3.g	100	3	36.70	15315	605	611	619	653	623	646
p4.3.h	100	3	40.00	18026	670	687	697	729	717	709
p4.3.i	100	3	43.30	20932	737	744	756	807	798	785
p4.3.j	100	3	46.70	23307	797	806	822	861	829	860
p4.3.k	100	3	50.00	23912	835	854	880	919	889	906
p4.3.l	100	3	53.30	24719	889	903	931	978	946	951
p4.3.m	100	3	56.70	27541	937	964	985	1063	956	1005
p4.3.n	100	3	60.00	28898	991	1014	1047	1121	1018	1119
p4.3.o	100	3	63.30	29981	1045	1065	1102	1170	1078	1151
p4.3.p	100	3	66.70	31094	1083	1102	1129	1222	1115	1218
p4.3.q	100	3	70.00	33252	1107	1140	1170	1251	1222	1249
p4.3.r	100	3	73.30	35739	1145	1175	1191	1272	1225	1265
p4.3.s	100	3	76.70	36637	1185	1211	1249	1293	1239	1282
p4.3.t	100	3	80.00	38749	1209	1242	1273	1304	1285	1288
p4.4.e	100	4	22.50	1895	183	183	183	183	182	182
p4.4.f	100	4	25.00	4714	315	317	324	324	304	315
p4.4.g	100	4	27.50	7572	449	450	461	461	460	453
p4.4.h	100	4	30.00	9646	528	541	556	571	545	554
p4.4.i	100	4	32.50	12556	603	618	630	657	641	627
p4.4.j	100	4	35.00	15196	668	685	711	732	697	732
p4.4.k	100	4	37.50	17570	742	758	772	821	770	819
p4.4.l	100	4	40.00	19872	808	818	832	880	847	875
p4.4.m	100	4	42.50	21976	840	858	877	919	895	910
p4.4.n	100	4	45.00	24819	886	897	917	968	932	977
p4.4.o	100	4	47.50	26196	930	953	984	1061	995	1014
p4.4.p	100	4	50.00	26048	997	1011	1028	1120	996	1056
p4.4.q	100	4	52.50	28604	1016	1058	1086	1161	1084	1124
p4.4.r	100	4	55.00	27919	1091	1111	1137	1203	1155	1165
p4.4.s	100	4	57.50	28988	1116	1146	1190	1255	1230	1243
p4.4.t	100	4	60.00	28970	1159	1187	1234	1279	1253	1255
p7.2.e	102	2	50.00	6440	280	288	290	290	275	290
p7.2.f	102	2	60.00	10380	373	376	382	387	379	382
p7.2.g	102	2	70.00	14356	444	451	457	459	453	459
p7.2.h	102	2	80.00	15787	514	517	521	521	517	521
p7.2.i	102	2	90.00	17982	557	563	578	579	576	578
p7.2.j	102	2	100.00	19937	606	610	615	644	633	638
p7.2.k	102	2	110.00	21051	666	674	681	705	693	702

<i>I</i>	# <i>N</i>	# <i>V</i>	<i>tMax</i>	<i>tavg</i>	<i>Bmin</i>	<i>Bavg</i>	<i>Bmax</i>	<i>AHS</i>	<i>CGW</i>	<i>TMH</i>
p7.2.l	102	2	120.00	22722	713	721	730	767	758	767
p7.2.m	102	2	130.00	26512	759	775	792	827	811	817
p7.2.n	102	2	140.00	28018	805	819	834	888	864	864
p7.2.o	102	2	150.00	30010	860	876	896	945	934	914
p7.2.p	102	2	160.00	33751	902	920	950	1002	987	987
p7.2.q	102	2	170.00	35759	928	955	980	1044	1031	1017
p7.2.r	102	2	180.00	38337	985	1001	1035	1094	1082	1067
p7.2.s	102	2	190.00	37237	1022	1038	1071	1136	1127	1116
p7.2.t	102	2	200.00	36614	1058	1073	1084	1179	1173	1165
p7.3.e	102	3	33.30	2281	175	175	175	175	163	175
p7.3.f	102	3	40.00	3860	247	247	247	247	235	247
p7.3.g	102	3	46.70	6310	344	344	344	344	338	344
p7.3.h	102	3	53.30	9034	412	420	425	425	419	416
p7.3.i	102	3	60.00	12612	471	473	478	487	466	481
p7.3.j	102	3	66.70	15089	542	549	556	564	539	563
p7.3.k	102	3	73.30	17746	600	609	613	633	602	632
p7.3.l	102	3	80.00	19484	660	669	674	681	676	681
p7.3.m	102	3	86.70	20016	710	722	740	762	754	756
p7.3.n	102	3	93.30	21830	761	767	775	820	813	789
p7.3.o	102	3	100.00	22219	812	822	832	874	848	874
p7.3.p	102	3	106.70	23263	836	857	881	927	919	922
p7.3.q	102	3	113.30	23745	905	917	941	987	943	966
p7.3.r	102	3	120.00	26130	921	949	963	1022	1008	1011
p7.3.s	102	3	126.70	27066	980	1001	1034	1079	1064	1061
p7.3.t	102	3	133.30	29218	1014	1041	1060	1115	1095	1098
p7.4.f	102	4	30.00	2018	164	164	164	164	156	164
p7.4.g	102	4	35.00	3254	217	217	217	217	209	217
p7.4.h	102	4	40.00	4634	283	284	285	285	283	285
p7.4.i	102	4	45.00	7266	350	350	353	366	338	359
p7.4.k	102	4	55.00	12327	500	505	513	520	516	503
p7.4.l	102	4	60.00	14261	567	572	586	590	562	576
p7.4.m	102	4	65.00	17500	613	625	638	646	610	643
p7.4.n	102	4	70.00	19544	667	679	691	730	683	726
p7.4.o	102	4	75.00	19907	736	749	757	781	728	776
p7.4.p	102	4	80.00	20798	796	809	818	846	801	832
p7.4.q	102	4	85.00	22915	832	850	886	906	882	905
p7.4.r	102	4	90.00	23287	897	910	928	970	886	966
p7.4.s	102	4	95.00	24411	942	954	973	1022	990	1019
p7.4.t	102	4	100.00	24631	987	1005	1026	1077	1066	1067

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 encaran el mismo. 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 y analizar que tan efectivo puede ser el mismo. La implementación final utiliza el algoritmo BRKGA y además, unas búsquedas locales para mejorar algunos individuos selectos de cada generación. La efectividad de la implementación final terminó dependiendo fuertemente de las heurísticas de búsqueda local.

Los resultados obtenidos son muy buenos, con al menos un tercio de los resultados llegaron a la solución optima de la instancia testeada. El resto obtuvo un i_{eMax} en el intervalo $[0.95, 0.99]$, salvo por algunas excepciones. Ahora, si considero el resultado obtenido por el BRKGA puro, los resultados no son lo suficientemente buenos para instancias grandes del problema, llegando a tener un i_{eMax} aproximado de 0,5. Quizá por como funciona el *crossover* en TOP, las soluciones hijas terminan siendo muy diferentes de su padres. Si ese fuera el caso, el BRKGA solo puede llegar a buenas soluciones con la ayuda de otras metaheurísticas como es en este caso.

Considero que uno de los problemas del BRKGA para TOP es que su secuencia de alelos no es utilizada completamente ya que parte del problema es que no todos los clientes pueden ser visitados. Asignar todos los clientes a algún vehículo siempre generaría una solución no factible o la instancia del problema no sería de TOP. Este problema de matching entre alelos y clientes visitados quizá puede ser resuelto modificando lo que representa un gen. Es decir, haciendo un decodificador nuevo.

7.1. Trabajos Futuros

Sería útil tener una herramienta para visualizar las soluciones en un plano cartesiano, pudiendo ver rápidamente que 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 individuos y sus individuos progenitores, a modo de tener una idea clara de que tan parecidos son. De todos modos, para un análisis mas preciso de tal correlación, sería mas eficiente idear una función que analizando las rutas de ambas soluciones genere un índice de parentesco.

El BRKGA puro necesita mejoras, no estoy del todo conforme con los resultados obtenidos utilizando solo el BRKGA. Si continuara mi 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 goloso y sea una instancia con 10 clientes y una solución generada a partir de su secuencia de alelos que visita a 6 de los 10 clientes, luego los últimos 4 alelos de la secuencia no impactan en el resultado final. Es decir, estos 4 alelos los podría cambiar de posición entre si y la solución seria la misma. Quizá podría implementarse 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. Sino, que de alguna manera existan pre establecido sectores asignados a un solo vehículo, basados en cercanía ó el centro de gravedad del sector. Esto es por donde continuaría la investigación y el desarrollo.

El segundo punto por el que intentaría mejorar los resultados del BRKGA es modificando el algoritmo de apareamiento. Quizá cada alelo represente una ruta de un vehículo. Luego 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.

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 *Replace* que actualmente cambia a un cliente visitado por otro no visitado si incrementa el beneficio total. La optimización sería buscar un *Replace* que no necesariamente sea de uno por uno. Podría ser el caso que hayan dos cliente no visitados cuyo beneficio total supere el de un cliente visitado, pero el beneficio de cada uno por separado, sea menor que el visitado. En este escenario el *Replace* actual no efectúa el cambio. También se podría implementar alguna heurística local tabú de modo de salir de mínimos locales.

El objetivo de este trabajo era analizar el rendimiento del algoritmo BRKGA para TOP, es por esto que de hacer trabajos futuros sobre el tema 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, (2012).
- [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] Benchmark de instancias de problemas del TOP por Chao I-M. y Tsiligirides T.
<https://www.mech.kuleuven.be/en/cib/op#section-3>
 - [18] Ke L., Archetti C. y Feng Z. *Ants can solve the team orienteering problem*. Computers and Industrial Engineering, 54:648–665, (2008).
 - [19] 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).
 - [20] Spears W. M. y De Jong K. A. *On the virtues of parameterized uniform crossover*. (1991).
 - [21] Tang H. y Miller-Hooks E. *A tabu search heuristic for the team orienteering problem*. Computers and Operations Research, 32:1379–1407, (2005).
 - [22] Tsiligirides, T. *Heuristic Methods Applied to Orienteering*. Journal of the Operational Research Society, 35(9), 797-809, (1984).
 - [23] 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).