



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, 2017

## 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, limitado en su longitud, que visita 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  $nodo_1$  y finalizando en  $nodo_n$ , de longitud no mayor que  $tMax$ . Tsiligirides [22] se refirió a esto como *Generalized Traveling Salesman Problem* (GTSP).

El *Team Orienteering Problem* (TOP) es la generalización al caso de múltiples rutas del *Orienteering Problem*. Resolver TOP implica encontrar un conjunto de rutas desde el punto de inicio hasta el punto final de forma tal que se maximice la sumatoria de beneficios recolectados, la distancia de todas las rutas no supere a  $tMax$  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, TOP pertenece a la clase de problemas NP-Completo porque contiene a OP como un caso especial donde solo hay una ruta. Resolver TOP requiere determinar un orden para cada ruta y además, seleccionar qué subconjunto de nodos a visitar, ya que no necesariamente se visitan todos los nodos.

En esta tesis, se propone un algoritmo genético llamado *Biased Random Key Genetic Algorithm* (BRKGA) para resolver TOP. El BRKGA inicializa su población utilizando un decodificador que convierte un conjunto de vectores de clave aleatoria, en un conjunto de soluciones válidas del problema. El BRKGA es una variante del *Random Key Genetic Algorithm* (RKGA). El BRKGA se diferencia del RKGA en el proceso de apareamiento, en BRKGA uno de los padres utilizado siempre pertenece al subconjunto de las mejores soluciones de la población y este padre tiene mayor probabilidad de transmitir sus genes a la solución hija.

En mi algoritmo, en cada nueva generación, la mejor solución se mejora con una búsqueda local. Los algoritmos de búsqueda local *Insert*, *Swap*, *Replace* y *2-Opt* se utilizan para encontrar mejores soluciones vecinas de una solución dada.

Los experimentos computacionales se realizan en instancias estándar. Luego, estos resultados se compararon con los resultados obtenidos por Chao, Golden y Wasil [9] (CGW), Tang y Miller-Hooks [21] (TMH) y Archetti, Hertz, Speranza [1] (AHS). Mis resultados son muy buenos y competitivos en la mayoría de las instancias.

**Keywords:** Team Orienteering Problem, Biased Random Key Genetic Algorithm, Problema de Enrutamiento, Búsqueda Local, Construcción de Soluciones Golosas.

## 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 score through these nodes beginning at  $node_1$  and ending at  $node_n$  of length no greater than  $tMax$ . Tsiligirides [22] refers to this 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 point to the ending point such that the total collected benefit received from visiting a subset of nodes is maximized, the length of each path is restricted by  $tMax$  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 service.

In this thesis, a genetic algorithm called *Biased Random Key Genetic Algorithm* (BRKGA) is proposed to solve TOP. The BRKGA algorithm initialize its population using a decoder that converts a set of random-key vectors into a set of valid solutions of the problem. The BRKGA is a variant of the *Random Key Genetic Algorithm* (RKGA). The BRKGA differs from RKGA in the mating process, in BRKGA one of the parents always belongs to the set of best solutions of the population and this parent has better chances of transmitting his gens to the child solution.

In my algorithm, in every new generation, the best solution is enhanced with a local search. The local search algorithms *Insert*, *Swap*, *Replace* and *2-Opt* are used in order to find better neighbor solutions of a given solution.

Computational experiments are made on standard instances. Then, this results, were compared to the results obtained by Chao, Golden, and Wasil [9] (CGW), Tang and Miller-Hooks [21] (TMH) and Archetti, Hertz, Speranza [1] (AHS). My results are are very good and competitive in most instances.

**Keywords:** Team orienteering problem, Biased Random Key Genetic Algorithm, Routing Problem, Local Search Heuristic, Greedy Solution Construction.

## Índice general

1..	Introducción . . . . .	1
2..	Revisión Bibliográfica . . . . .	4
3..	Modelo Matematico . . . . .	7
4..	Biased Random Key Genetic Algorithm . . . . .	8
4.1.	Algoritmos Genéticos . . . . .	8
4.2.	Random Key Genetic Algorithms . . . . .	8
4.3.	Biased Random Key Genetic Algorithms . . . . .	9
4.4.	Decodificador del BRKGA . . . . .	10
5..	Algoritmo BRKGA con búsqueda local para TOP . . . . .	12
5.1.	Decodificador . . . . .	14
5.1.1.	Orden de los clientes a considerar . . . . .	14
5.1.2.	Decodificador simple . . . . .	14
5.1.3.	Características y debilidades del decodificador simple . . . . .	16
5.1.4.	Decodificador Goloso . . . . .	16
5.2.	BRKGA . . . . .	18
5.2.1.	Configuración . . . . .	18
5.2.2.	Descripción y codificación de las propiedades del objeto configuración . . . . .	19
5.2.3.	Inicialización de la Población . . . . .	20
5.2.4.	Condición de parada . . . . .	21
5.2.5.	Evolución de la población . . . . .	21
5.2.6.	Resultados de la primer versión . . . . .	24
5.3.	Heurísticas de busqueda local . . . . .	25
5.3.1.	Center of Gravity . . . . .	26
5.3.2.	Swap . . . . .	27
5.3.3.	Insert . . . . .	28
5.3.4.	2-opt . . . . .	29
5.3.5.	Replace . . . . .	30
5.3.6.	Encoder . . . . .	31
5.3.7.	Secuencia de Aplicación de las Heurísticas . . . . .	34
6..	Resultados . . . . .	38
7..	Conclusiones . . . . .	44
7.1.	Trabajos Futuros . . . . .	44

## 1. INTRODUCCIÓN

*Orienteering* es un deporte al aire libre usualmente jugado en una zona montañosa o fuertemente boscosa. 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 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 la orientación 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. Un competidor tiene que seleccionar un subconjunto de puntos de control para visitar que maximizarán la puntuación total sujeto a la restricción de tiempo. Esto se conoce como *Orienteering Problem* y se denota por OP.

El equipo de orientación extiende la versión de un solo competidor del deporte. Un equipo formado por varios competidores (digamos 2, 3 o 4 miembros) comienza en el mismo punto. Cada miembro del equipo intenta visitar tantos puntos de control como sea posible dentro de un límite de tiempo prescrito, y luego termina 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. Se llama *Team Orienteering Problem* y lo denotan por TOP.

La versión de un solo competidor (OP) de este problema es NP-Completa como demostraron Golden, Levy, y Vohra [16], por lo que el TOP es al menos tan difícil. Por lo tanto, 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 juego deportivo 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.
- IV El problema de entrega de combustible con vehículos múltiples 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 (nivel de inventario) 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 maximizar su potencial para reclutar futuros jugadores, deben visitar tantas escuelas secundarias como sea posible dentro de un radio de 100 km del campus. Pero, sabían por experiencia previa que visitar todas las escuelas en esta área no era posible. Por lo tanto, querían visitar el mejor subconjunto de escuelas en esta 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. Notar que 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 Tsiligirides y de Chao [17]. Ambos autores comparten el mismo formato.

Una instancia de TOP contiene:

- $N$  vehículos de carga. Cada vehículo tiene una distancia máxima, llamada  $tMax$ , 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  $tMax$ .
- $M$  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 a la distancia máxima ( $tMax$ ) 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 decoder 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 199 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 voy a referenciar varias de las publicaciones que referenciaron, y otros trabajos que encontré relacionados con TOP.

La primera heurística propuesta para el TOP es un algoritmo de construcción simple introducido en 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. Fue introducido con el nombre *Multiple Tour Maximum Collection Problem*.

Una heurística de construcción más sofisticada se da en Chao, Golden y Wasil (CGW) [9]. Este es la primera publicación donde aparece el nombre TOP. Este nombre fue acuñado por Chao et al. para resaltar la conexión con el más ampliamente estudiado caso de un solo vehículo (OP). 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 es 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. El algoritmo es capaz de 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].

Luego, se aplicaron varias metaheurísticas al TOP, partiendo del algoritmo de *Tabu Search* introducido en Tang y Miller-Hooks (TMH) [21], que 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 tabu 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



(validas e invalidas) a la solución actual en función de los parámetros tabú actuales. Note que 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 tabu puede anularse si la mejor solución tabu 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 regrese al Paso A o al B. Nótese que 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 y supera a otras heurísticas publicadas hasta tal momento para el TOP.

Archetti et al. [1] proponen dos variantes de un algoritmo de un *Tabu Search* generalizado y 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. ACO es una clase de metaheurísticas basadas en población. 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 en ACO es construir soluciones candidatas. Proponen cuatro métodos, los 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 heurísticas de búsqueda local. Tales heurísticas como *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 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*, *Replace* y *Insert*. 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 elite. Se inserta la mejor solución obtenida si el conjunto de elite no esta completo aun. En caso de que el conjunto de elite este completo, si la peor solución del conjunto es superada por la solución de la iteración actual, se reemplazan.

Bouly et al. [5] idearon un *Memetic Algorithm* (MA) para resolver 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 al momento de su publicación.

Dang et al. [11], este siendo el mejor actual en su clase, proponen un *Particle Swarm Optimization based Memetic Algorithm* (PSOMA) para resolver TOP. Su algoritmo PSO-MA provee de soluciones de alta calidad para TOP. El algoritmo está relativamente cerca de 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 los barrios de búsqueda local. Sin embargo, el esquema global se ha modificado por una optimización de enjambre de partículas. La optimización de enjambre de partículas (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 (crossover) y mutaciones dentro de una población. Su último componente vendría a ser el algoritmo responsable de controlar el proceso evolutivo, asegurándose que los cromosomas (soluciones) sean validas respecto de las restricciones de la instancia de 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 razonablemente buenos, pero no superiores a los de Dang et al. [11] y Bouly et al. [5], por este motivo no los compare con los de este trabajo.

Esos fueron los trabajos encontrados en mi investigación sobre trabajos previos, hay algunos que implementen algoritmos genéticos pero ninguno que implemente un *Biased Random Key Generation Algorithm* (BRKGA).

### **3. MODELO MATEMATICO**

## 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. Se hace una analogía entre una solución y un individuo en una población. Cada individuo es un cromosoma que codifica una solución. Un cromosoma consiste en una cadena de genes. Cada gen puede tomar un valor, llamado alelo, de algún alfabeto. Los cromosomas tienen asociado a ellos 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 las de mejor condición física. La segunda fuente de individuos provienen del *crossover* (apareamiento), el *crossover* es el método a través por el cual se obtiene un nuevo individuo a partir de dos individuos de la generación anterior. 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 el conjunto de soluciones de elite pasan directamente a la nueva generación. Además en el método de apareamiento, cuando los individuos son seleccionados para aparearse y producir descendencia, aunque los individuos sean seleccionados al azar, 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 Algorithms

El *Random Key Genetic Algorithms* (RKGA) fue introducido por Bean en *Genetic algorithms and random keys for sequencing and optimization*. [4]. En RKGA, los cromosomas 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 puede calcular su valor objetivo o aptitud física. Los algoritmos RKGAs evolucionan una población de vectores de claves aleatorias sobre una serie de iteraciones llamadas generaciones. La población inicial se compone de  $p_t$  vectores de claves aleatorias. Cada alelo se genera independientemente al azar en el intervalo real  $[0, 1]$ . Después de calcular la aptitud de cada individuo por el decodificador, la población se divide en dos grupos de individuos. Un pequeño grupo de individuos de élite  $p_e$ , es decir, aquellos con mejores valores de aptitud individual. El segundo el conjunto remanente de  $p_t - p_e$  no elite individuos donde  $p_e < p_t - p_e$ , es decir el conjunto de elite es menor en tamaño al conjunto de no elite. 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 los RKGAs, se generan individuos mutantes a partir de un vector de claves aleatorias, 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 individuos  $p_t - p_e - p_m$  es requerido para completar la generación  $k + 1$ . Esto se hace generando una descendencia mediante el proceso de *crossover*, en el RKGA 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 Algorithms

El *Biased Random Key Genetic Algorithms* (BRKGA), difiere del RKGA en la forma en que los padres son seleccionados para el *crossover*. En un 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 y, por lo tanto, un individuo puede producir más de un hijo. Como el tamaño del conjunto de elite es mayor al conjunto de no elite ( $p_e < p_t - p_e$ ), la probabilidad de que un individuo de élite sea seleccionado para el apareamiento es mayor que la de un individuo no élite. Por lo tanto un individuo de élite tiene un mayor probabilidad de transmitir sus genes a 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 BRKGAs. Sea  $\rho_e > 0,5$  la probabilidad de que un descendiente herede el alelo de su padre de elite. Sea  $n$  el número de cromosomas 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  $\rho_e$  y el valor del  $e'_i$  del padre no elite con probabilidad  $1 - \rho_e$ . Este proceso se puede ver en la figura ???. De esta manera, es más probable que la descendencia herede características del padre de élite que las del padre no de élite. Dado que asumimos que cualquier vector de clave aleatoria puede ser decodificado en una solución, entonces el descendiente del *crossover* siempre decodifica en una solución válida del problema de optimización combinatoria.

En la figura 4.2 se puede observar como funciona la evolución. 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 *crossover* entre un individuo elite con un individuo no elite.

En la figura ??? se puede observar como funciona el *parameterized uniform crossover*. El primer vector de cromosomas es de un individuo de elite, el segundo vector de cromosomas es de un individuo no elite. Se decide que cromosomas tomara el individuo hijo utilizando un vector de reales aleatorio del mismo tamaño que los vectores de cromosomas. En este ejemplo se utiliza un  $\rho_e = 0,70$ , de este modo la descendencia se asemejara mas al padre de elite.

Fig. 4.1: bias crossover

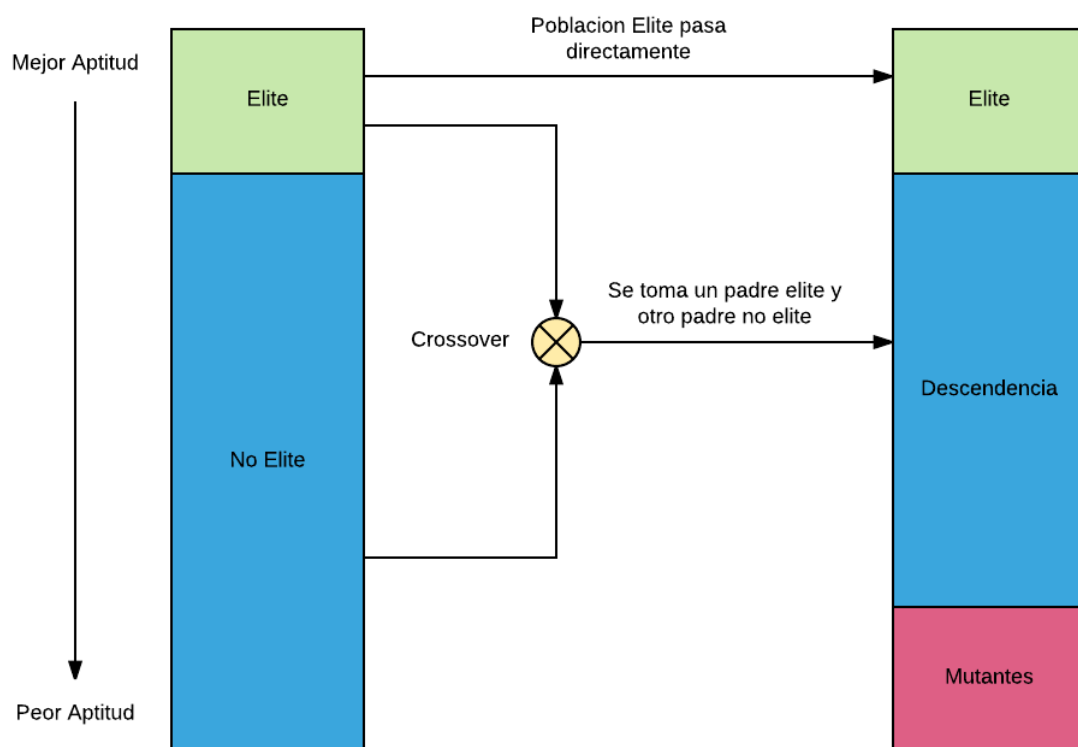


#### 4.4. Decodificador del BRKGA

Una característica importante para mencionar del BRKGA es que el decodificador es el único módulo del algoritmo que requiere conocimiento del dominio del problema. El decodificador transforma un vector aleatorio de enteros en una instancia de una solución del problema. Es un adaptador, por lo tanto si hacemos un decodificador para otro problema podríamos reutilizar el módulo de BRKGA.

En el caso de mi implementación del BRKGA entre cada generación se ejecutan unas heurísticas de búsqueda local sobre las mejores soluciones de la población. Como estas heurísticas trabajan sobre una instancia de la solución, requiere que exista un objeto codificador. Es decir, es decir un algoritmo capaz de convertir una solución del problema en un vector de enteros aleatorios. De este modo, una vez que se mejora una solución, podemos actualizar su vector aleatorio que lo representa y seguir el curso del BRKGA.

Fig. 4.2: Evolución de una población



## 5. ALGORITMO BRKGA CON BÚSQUEDA LOCAL PARA TOP

En el presente capítulo ahondaremos en detalle la solución implementada para el *Team Orienteering Problem*. La implementación se la puede dividir en 3 módulos importantes. Primero el decodificador, que como mencionamos en el capítulo BRKGA tiene la tarea de convertir un arreglo de enteros aleatorios en una solución válida del problema. Luego el algoritmo BRKGA, cuya implementación puede hacerse con total independencia del problema a resolver. Por último las heurísticas locales aplicadas en cada nueva generación a el mejor individuo de la población, al cual no se le hayan aplicado previamente.

En el capítulo *Resultados* se mostrara en detalle los resultados obtenidos de la versión final de la implementación sobre el benchmark de problemas de Chao y Tsiligirides [17]. Se compararon los resultados con los obtenidos de los trabajos previos de Chao et al. [9], de Archetti et al. [1] y los de Tang and E. Miller-Hooks [21]. En este capítulo se utilizaran los resultados de tales trabajos previos para crear un índice de efectividad a modo de poder evaluar la efectividad de la solución generada por el modulo analizado.

Además contaré las dificultades encontradas y desiciones tomadas para llegar a soluciones más competitivas con los trabajos previos seleccionados. A modo de monitorear el progreso, primero se seleccionó un subconjunto diverso del benchmark de problemas de Chao y Tsiligirides [17]. Luego, para cada versión de la implementación que fui generando, se les calculó su índice de efectividad. Tal índice lo definí de la siguiente manera:

$$bestProfit(i) = Max(Profit(CGW(i)), Profit(AHS(i)), Profit(TMh(i)))$$

$$efectividad(imp_{v.xyz}) = Profit(imp_{v.xyz}(i)) / bestProfit(i)$$

Donde  $Profit(Implementacion_{v.xyz}(i))$  representa la ganancia de la mejor solución obtenida luego de ejecutar una cierta cantidad de veces la version  $xyz$  para la instancia del problema  $i$ . Luego  $Profit(CGW(i))$ ,  $Profit(AHS(i))$ ,  $Profit(TMh(i))$  representan la ganancia de las implementaciones de Chao et al., Archetti et al. y Tang Miller-Hooks respectivamente para la misma instancia  $i$ . El índice es simplemente dividir la ganancia obtenida por la mejor ganancia encontrada por los trabajos previos seleccionados. Si el  $efectividad(imp_{v.xyz}) = 1$ , luego nuestra implementación habra generado una solución tan buena como los trabajos previos. Si  $0,95 \leq efectividad(imp_{v.xyz}) < 1$ , mi solución no sera tan buena pero considero que esta en un rango donde es competitiva. A lo largo de todo el desarrollo, el objetivo fue maximizar el índice de efectividad sin tener certeza de llegar a buenas soluciones utlizando la heurística BRKGA.

Para cada uno de los siguientes modulos y submodulos, se mostraran su pseudocodigo y su índice de efectividad. El pseudocódigo utilizado sigue la sintaxis de c#, el lenguaje en el cual implemente el desarrollo. El objetivo del pseudocodigo es meramente descriptivo, el método que representa puede ser implementado con ciertas variaciones que no viene al caso describir. Tales variaciones para no mostrar: variables de medición de tiempo, variables



para monitorear estados, testing, manejo de excepciones, persistencia de resultados, y demas, con el objetivo de ser mas ilustrativo de la funcionalidad del algoritmo.

Para la valuación del índice de efectividad se eligieron seis instancias del benchmark de modo que fueran variadas entre si. Tome dos pequeñas, dos medianas y dos grandes, cuyas descripción pueden observarse en la siguiente tabla:

Autor	Instancea	Vertices	Vehículos	TMax
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

Las tablas donde mostraré el índice de efectividad de los resultados parciales tendrá un subconjunto de los siguientes encabezados:

$I$	$\#N$	$\#V$	$tMax$	$C$	$\#S$	$T_{avg}$	$B_{max}$	$B_{min}$	$B_{avg}$	$i_f$	$BTP_{max}$
-----	-------	-------	--------	-----	-------	-----------	-----------	-----------	-----------	-------	-------------

Donde:  $Profit(CGW(i))$ ,  $Profit(AHS(i))$ ,  $Profit(TMH(i))$

- $I$ : Nombre de la instancia. Ej: p2.3.g
- $\#N$ : Cantidad de nodos de la instancia. Ej: 21
- $\#V$ : Cantidad de vehículos de la instancia. Ej: 3
- $tMax$ : Distancia máxima de la ruta de los vehículos de la instancia. Ej: 10.70
- $C$ : Configuración general del BRKGA. Es un código que sintetisa la configuración basica global, explicado en detalla más adelante (ver sección ).
- $\#S$ : Cantidad de soluciones de las cuales se sacaron los datos. Ej: 200
- $T_{avg}$ : Tiempo promedio en milisegundos de la ejecución total de la implementación. Ej: 21578
- $B_{max}$ : Beneficio máximo de las  $\#S$  soluciones generadas. Ej: 140
- $B_{min}$ : Beneficio mínimo de las  $\#S$  soluciones generadas. Ej: 45
- $B_{avg}$ : Beneficio promedio de las  $\#S$  soluciones generadas. Ej: 45
- $i_f$ : Índice de efectividad. Se utiliza el  $B_{avg}$  para tener un valor que mejor refleje la efectividad real. Si usara  $B_{max}$ , el indice aparentaría muy bueno si entre las  $\#S$  soluciones se genero una muy buena aleatoriamente. Ej:  $B_{avg}/BTP_{max} = 0,57$
- $BTP_{max}$ : Beneficio de Trabajos Previos máximo para la misma instancia. Los trabajos previos utilizados son CGW, AHS y TMH. Ej: 145

## 5.1. Decodificador

El decodificador debe generar una solución valida del problema dado un vector aleatorio de enteros y conociendo la instancia del problema (vehículos disponibles, clientes, tmax, etc). A modo El decodificador construye una solución valida asignando clientes a las rutas de los vehículos disponibles respetando tmax, su distancia maáxima de recorrido. El orden en que toma los clientes a asignar es clave y determina la solución resultante. Es ahí donde utilizaremos el vector aleatorio de enteros, el vector aleatorio de enteros marcara el orden en el cual se tomaran los clientes para asignarlos a una ruta. Por lo tánto el vector aleatorio de enteros tendrá una longitud equivalente a la cantidad de clientes del problema (vertices con beneficio mayor a cero).

### 5.1.1. Orden de los clientes a considerar

Dado una instacia de un problema con  $n$  clientes y un vector de enteros aleatorio de tamaño  $n$ , un decoder genera una solución valida de un problema. El vector, en mi implementacion, es un vector de RandomKeys. Un *RandomKey* tiene dos propiedades, el entero aleatorio llamado *Key* y otro entero llamado *PositionIndex*.

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

El proposito de *PositionIndex* es para mapear un Key con un Cliente. Existe un arreglo de clientes en el Mapa del problema. Los clientes siempre tienen la misma posición y esa posición es un entero en el intervalo  $[0, \#Clientes - 1]$ . Luego para un vector de RandomKeys de tamaño  $\#Clientes$  no existen dos RandomKeys con mismo valor de PositionIndex y todos los PositionIndex se encuentran en el intervalo  $[0, \#Clientes - 1]$ . De forma que cada RandomKey matchea siempre con solo RandomKey. Luego se matchean los clientes con los randomKeys por su position index y se los ordena de forma ascendente por el Key aleatorio. Esto se puede ver en la figura 5.1.

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

### 5.1.2. Decodificador simple

El decodificador simple recibe como parametro el vector de enteros aleatorios y con eso genera los clientes ordenados. Luego por cada vehiculo, si el siguiente cliente se puede incluir en la ruta se incluye sino considera que la ruta esta completa y pasa al siguiente vehiculo. Un cliente  $c_i$  se puede agregar a la ruta si al agregarlo no supera la distancia maxima permitida para la ruta. Sean  $v$  vehiculo,  $dMax$  la distancia maxima de  $v$ ,  $dAct$  la distancia actual de  $v$ ,  $f$  el destino final de la ruta,  $c_u$  el ultimo cliente agregado a la

Fig. 5.1: RandomKeys ordenando clientes

Vector de RandomKeys recién inicializado:

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

Vector de RandomKeys ordenado por propiedad Key:

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

ruta de  $v$  y  $c_i$  cliente a evaluar agregar a la ruta de  $v$ . Luego cree la funcion can visit que implementa la siguiente formula:

$$dAct + distancia(c_u, c_i) + distancia(c_i, f) - distancia(c_u, f) \leq dMax$$

Pseudocodigo del método 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;
}
```

A partir de 200 *RandomKeys* generé 200 soluciones con el decoder simple para las subconjunto de instancias diversas elegidas. Se utilizaron distintos sets de 200 *RandomKeys* para cada instancia. La siguiente tabla tiene los resultados obtenidos (ver 5 para significado de columnas):

$I$	$\#N$	$\#V$	$tMax$	$\#S$	$B_{max}$	$B_{min}$	$B_{avg}$	$i_f$	$BTP_{max}$
p2.2.k	21	2	22.50	200	175	40	102	0.37	275
p2.3.g	21	3	10.70	200	140	45	83	0.57	145
p3.4.p	33	4	22.50	200	270	90	170	0.30	560
p5.3.x	66	3	40.00	200	405	195	295	0.19	1555
p7.2.e	102	2	50.00	200	98	8	39	0.13	290
p7.4.t	102	4	100.00	200	221	40	116	0.11	1077

En estos resultados ya se puede observar una correlación que sera recurrente, para intancias pequeñas los resultados son mejores.

### 5.1.3. Características y debilidades del decodificador simple

Tiene un orden de complejidad de  $O(\#clientes)$ .

Sea  $v$  el vector de clientes ordenados por un vector aleatorio de enteros. Existen  $m + 1$  indices  $i_0 = 0, i_1, i_2, \dots, i_m$  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. 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
PositionIndex	6	2	5	1	4	8	3	7

Un problema que tiene este decodificar es en la existencia de un cliente inalcanzable, es decir si existe  $c$  cliente tal que:

$$distancia(i, c) + distancia(c, f) > dMax$$

Esto puede generar soluciones de la poblacion donde existan vehiculos con rutas vacias. La solución fácil a este problema es filtrando todos los clientes inalcanzables previo a la ejecución del BRKGA.

Otro problema que tiene este decodificador es que cambia de vehículo al primer intento fallido de expandir su ruta. Luego puede generar soluciones con rutas muy pequeñas, por este motivo implemente un 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 pudo agregar a la ruta de un vehículo. 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 mas clientes que intentar. Luego al pasar al siguiente vehiculo intenta solamente con los clientes no asignadas a los vehículos anteriores y siempre respetando el orden de los clientes asignado por el vector de enteros aleatorios.

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

Con esta modificación su orden complejidad aumenta a  $O(\#clientes * \#vehiculos)$ . Aunque la cantidad de vehículos de las instancias del benchmark esta acotado por cuatro, esta diferencia de complejidad tiene un impacto ya que este algoritmo de decodificación corre *PopulationSize* por generación. Luego para mismo tamaño de población y cantidad de generaciones, usar el decoder simple contra el goloso tiene un efecto visible en el tiempo de ejecución. Por otro lado, nn promedio aumenta el profit de las soluciones generadas por el decodificador. Ese es trade-off que tenemos entre el decodificador simple y el goloso.

Un detalle de menor importancia, el decoder goloso no tiene los clientes de forma continua por vehiculo asignado al ordenar los clientes por el RandomKeys. Esto puede verse en la figura 5.3.

Nuevamente, utilizando distintos sets de 200 *RandomKeys* para cada instancia se generaron los siguientes resultados:

<i>I</i>	<i>#N</i>	<i>#V</i>	<i>tMax</i>	<i>#S</i>	<i>B<sub>max</sub></i>	<i>B<sub>min</sub></i>	<i>B<sub>avg</sub></i>	<i>i<sub>f</sub></i>	<i>BTP<sub>max</sub></i>
p2.2.k	21	2	22.50	200	260	95	164	0.60	275
p2.3.g	21	3	10.70	200	140	95	122	0.84	145
p3.4.p	33	4	22.50	200	410	180	288	0.51	560
p5.3.x	66	3	40.00	200	525	305	412	0.26	1555
p7.2.e	102	2	50.00	200	163	31	96	0.33	290
p7.4.t	102	4	100.00	200	438	160	280	0.26	1077

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 el cliente 5, intenta en orden con el resto de los clientes aún no visitados y logra insertar el cliente 8. De un modo similar, sucede con la segunda ruta al no poder incluir al cliente 4.

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

Todos los resultados promedio, mínimo y máximos mejoran considerablemente. Tal es así, que el  $i_f$  en algunos casos es mayor al doble de lo obtenido en el decodificador simple. Por otro lado, se vuelve a observar la correlación entre tamaño de la instancia y el  $i_f$ .

## 5.2. BRKGA

En una primera instancia se implementa un BRKGA estandar. 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 generamos una nueva generación de soluciones a partir de la generación anterior como se explico en la sección BRKGA (ver sección 4.3).

Pseudocódigo de una vista macro general del algoritmo BRKGA implementado.

```
public Solution RunBrkga(ProblemManager problemManager)
{
    ProblemManager.InitializePopulation();

    while (!ProblemManager.StoppingRuleFulfilled())
        ProblemManager.EvolvePopulation();

    return ProblemManager.Population.GetMostProfitableSolution();
}
```

El objeto ProblemManager es el orquestador del mi BRKGA, se setea con un objeto Configuración y el objeto PopulationGenerator. Tiene acceso de forma indirecta, através de PopulationGenerator, de toda la información del problema (vehículos, clientes, tmax, etc.).

### 5.2.1. Configuración

A modo de poder testear distintas configuraciones del BRKGA, se creo un objeto *Configuration* que setea todas las configuraciones que impactan en el resultado final del BRKGA. Este objeto es esencial para tunear el implementación de una forma rápida y ordenada. Al centralizar todas las variables que podrían impactar en resultado final, ganaba mucho tiempo en al testear variaciones de mi implementación. Además, al estar

centralizada toda la información variable se obtiene una lectura veloz del BRKGA que se esta usando. En otras palabras incrementamos nuestra capacidad de monitoreo y control de mi implementación.

```
public class BrkgaConfiguration
{
    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 int EliteGenChance { get; set; }
    public List<ILocalSearchHeuristic> Heuristics { get; set; }
    public int ApplyHeuristicsToTop { get; set; }
    public DecoderEnum DecoderType { get; set; }

    private void SetDescription();
}
```

### 5.2.2. Descripción y codificación de las propiedades del objeto configuración

Descripción de las propiedades del objeto Configuración:

- **Description:** Es una especie de hash descriptivo de la instancia del objeto. Su funcionalidad es poder identificar rápidamente la configuración global del BRKGA. Solo puede ser seteado con el metodo SetDescription() que utiliza la clave y el valor de el resto de las propiedades.
- **MinIterations:** Clave **MI**. Valor entero utilizado en la función de corte. Cantidad mínima de generaciones que debe genererar el BRKGA para finalizar.
- **MinNoChanges:** Clave **MNC**. Valor entero utilizado en la función de corte. Cantidad de generaciones sin modificaciones del mejor beneficio necesario para cortar el algoritmo BRKGA.
- **PopulationSize:** Clave **PS**. Valor entero denota el tamaño de la población.
- **ElitePercentage:** Clave **EP**. Valor decimal en el itervalo (0,1) que determina el tamaño de la poblacion elite.
- **MutantPercentage:** Clave **MP**. Valor decimal en el itervalo (0,1) que determina el tamaño mínimo de la poblacion mutante.
- **EliteGenChance:** Clave **EGC**. Valor entero en el intervalo (0,100) que determina la probabilidad que tiene un key del padre elite, en transmitirse a su descendiente.
- **Heuristics:** Clave **HEU**. Secuencia de heurísticas de busqueda local que se le aplicaran a la mejor solución de cada generación, a la cual no se le hayan aplicado las heurísticas locales aún. Se implementaron cuatro heurísticas locales:

- **Swap:** Valor **S**.
- **Insert:** Valor **I**.
- **2-Opt:** Valor **O**.
- **Replace:** Valor **R**.
- **ApplyHeuristicsToTop:** Clave **TOP**. Valor entero que denota la cantidad de soluciones a las cuales se les aplicaran las heurísticas de búsqueda local.
- **DecoderType:** Clave **D**. Es una enumeración que determina el decoder que se va a utilizar.
  - **Simple:** Valor **S**.
  - **Goloso:** Valor **G**.

El método `SetDescription()` toma las tuplas de clave y valor de todas las propiedades del objeto `configuration` excepto `Description` y genera un string intercalando con un separador.

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

Luego dado un `description` podemos ver como esta configurado el BRKGA.

Ej: "MI.200;MNC.10;PZ.100;EP.0,3;MP.0,1;EGC.70;HEU.ISIRT;TOP.2;D.G"

- `MinIterations:` 200
- `MinNoChanges:` 10
- `PopulationSize:` 100
- `ElitePercentage:` 0,3
- `MutantPercentage:` 0,1
- `EliteGenChance:` 70
- `Heuristics:` ISIRT. Que es la Secuencia Insert, Swap, Insert, Replace, 2-Opt.
- `ApplyHeuristicsToTop:` 2
- `DecoderType:` Decodificador Goloso

### 5.2.3. Inicialización de la Población

Utilizando `PopulationSize` del objeto `Configuration` seteo el tamaño de la población. Luego para generar la población inicial se generaran `PopulationSize` vectores de enteros aleatorios de tamaño `#clientes` de la instancia del problema. Este conjunto de vectores se lo pasa como argumento al decodificador, quien genere un individuo por cada vector de enteros aleatorios.



#### 5.2.4. Condición de parada

En un principio la condición de parada era simple, el bucle terminaba cuando iteraba una  $x$  cantidad de veces seteado en el objeto *Configurations* por *MinIterations*. Es decir que el bucle principal cortaba luego de evolucionar la población *MinIterations* veces. Luego de analizar las últimas generaciones de la solución y ver que era frecuente que la mejor solución se había generado recientemente, agregue una condición de corte adicional. Ahora, para cortar además de la condición anterior, el beneficio de la mejor solución no debería haberse modificado durante las últimas  $n$  generaciones. Es decir durante las últimas  $n$  generaciones no debe haber aparecido una nueva mejor solución. Este valor es ajustable desde la propiedad *MinNoChanges* del objeto *Configurations*.

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

#### 5.2.5. Evolución de la población

Se toma la población y se los ordena descendientemente según su beneficio calculado por la función objetivo. Se setea la población de elite y la non-elite. La población de elite son los mejores  $x$  individuos de la población. Siendo  $x$  un porcentaje de la población total seteado con la propiedad *ElitePercentage* del objeto *Configuration*. La población non-elite son todos aquellos individuos no incluidos en la población de elite. Luego se generan  $y$  individuos mutantes,  $y$  es un porcentaje de la población total seteado por la propiedad *MutantPercentage*. Pasan a la nueva generación todos los individuos de la población de elite y se agregan los de la población mutante. Finalmente se completa la nueva generación emparentando individuos de la población de elite con individuos de la población non-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.

```

public Population Evolve(Population population)
{
    var ordPopulation = population.GetOrderByMostProfitable();
    var elites = ordPopulation.Take(EliteSize);
    var nonElites = ordPopulation.Skip(EliteSize).Take(NonEliteSize);
    var mutataants = Generate(MutatansSize);
    var evolvedPopulation = new pop(elites, mutataants);
    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.Add(GenerateSolution());
        else
            evolvedPopulation.Add(childSolution);
    }
    return evolvedPopulation;
}

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.Next(100) >= 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);
}

```

Como mencioné anteriormente, los individuos mutantes son individuos generados a partir de un vector de RandomKeys, del mismo modo que la población inicial. Con el fin de mostrar lo simple que es la generación de un nuevo vector de RandomKeys, presento el pseudocódigo del GenerateSolution que al final termina llamando al método decode del decoder descripto anteriormente.

```

private Solution GenerateSolution() ()
{
    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)
    }
    return Decoder.Decode(randomKeys, ProblemInfo);
}

```

Verificar que dos soluciones son iguales tiene un costo muy bajo en BRKGA. Esto se debe a que el decodificador es un algoritmo determinístico. Por lo tanto para dos vectores de RandomKeys cuyo orden de clientes que genere sea el mismo, el decodificador generará la misma solución. Luego lo único que hay que comparar es el hash de ambas soluciones y esto se puede hacer en  $O(1)$ . El hash de una solución, se calcula una sola vez cuando se construye la solución a partir de su vector de RandomKey. El hash es una concatenación de la propiedad PositionIndex de cada RandomKey en el vector ordenados por la propiedad Key, intercalados con un separador. Es decir, es el orden de la ciudades que toma el decoder intercalados con un separador. Si ya existe un individuo con el mismo hash, se genera una solución mutante y continua el apareamiento de otros dos individuos. Decidí no reintentar el apareamiento entre los dos padres que generaron la solución ya existente, porque consideré que la probabilidad de volver a generar nuevamente una solución existente es alta. Además, aún si en un segundo intento no generara una solución idéntica a alguna de la nueva población, lo más seguro que la solución originada se muy cercana a una existente. Luego tome esta decisión para optimizar el apareamiento y disminuir soluciones dentro de un mismo vecindario por generación.

```

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.PositionIndex));
    return pseudoHash;
}

```

En una primera instancia se insertaban los individuos sin verificar que la solución no existiese en la población. Dado una población de soluciones no repetidas, la probabilidad de generar una solución existente al evolucionar la población, es baja. Aún así, una vez que sucede, la probabilidad de generar otra más aumenta considerablemente ya que ahora hay mas probabilidades de utilizar padres idénticos. Si además la solución repetida se encuentra dentro del subconjunto de elite, la probabilidad aumenta aún más. Esto generaba un efecto bola de nieve donde cada nueva generación tenía cada vez más individuos repetidos. Incluso se llegó al caso donde toda una población constituía de una única solución excepto por las soluciones mutantes. Esto reducía ampliamente la cantidad de soluciones diferentes

exploradas, luego reducía fuertemente la frecuencia con la que una nueva mejor solución aparecía. Además si uno tiene varios individuos iguales en una solución, el algoritmo se vuelve menos eficiente ya que repite trabajo en donde obtiene los mismos resultados. Entonces el costo total de validar unicidad en la inserción, que conlleva un orden de complejidad  $O(PopulationSize * NonElitePopulationSize)$ , resulta muy bajo comparado con el costo de trabajar con múltiples soluciones repetidas.

### 5.2.6. Resultados de la primer versión

La primer versión del BRKGA para TOP no incluía el objeto de configuración y permitía insertar soluciones repetidas en una misma generación. Los índices de efectividad que mostrare a continuación corresponden con una primera versión que no admitía repetidos y el objeto configuración. Para estos primeros resultados genere resultados por instancia y utilice una configuración estandar sin heurísticas locales aún:

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

$I$	$\#N$	$\#V$	$tMax$	$T_{avg}$	$B_{max}$	$B_{min}$	$B_{avg}$	$i_f$	$BTP_{max}$
p2.2.k	21	2	22.50	2977	260	240	249	0.91	275
p2.3.g	21	3	10.70	1990	145	145	145	1.00	145
p3.4.p	33	4	22.50	6482	450	430	438	0.78	560
p5.3.x	66	3	40.00	17908	660	610	635	0.41	1555
p7.2.e	102	2	50.00	8753	246	204	217	0.75	290
p7.4.t	102	4	100.00	31532	513	458	481	0.45	1077

De estos primeros resultados podemos ver que el BRKGA puro sin otras heurísticas funciona muy bien para instancias de testeo pequeñas esta versión funcionaba tan bien como Chao, Golden y Wasil (CGW), Tang y Miller-Hooks (TMH) y Archetti, Hertz, Speranza (AHS). Esto se refleja en la instancia  $p2.3.k$  que siempre se llegó a la mejor solución posible y en  $p2.2.k$  donde el  $i_f$  supera el 0.90. Luego a medida que incrementa el tamaño de la instancia, disminuye el  $i_f$ . Una observación que quiero destacar de estos resultados es sobre el  $i_f$  de la instancia 'p7.2.e'. Notar que tiene tantos nodos como 'p7.4.t' y sin embargo su  $i_f$  ampliamente mayor. Esto se puede estar dando por que solo son 2 vehículos, en vez de 4, adicionalmente la distancia máxima,  $tMax$ , es 50 en vez de 100 reduciendo la combinatoria de soluciones posibles.

Como estos resultados no eran satisfactorios, tomé las dos instancias de este subconjunto con menor  $i_f$  y las utilice para testear distintas configuraciones. Probé múltiples variaciones del objeto configuración. A continuación el resultado de algunas de tales variaciones:

$I$	$\#N$	$\#V$	$tMax$	$C$	$\#S$	$T_{avg}$	$B_{max}$	$B_{min}$	$B_{avg}$	$i_f$	$BTP_{max}$
p5.3.x	66	3	40.00	1	10	60782	700	635	656	0.42	1555
p5.3.x	66	3	40.00	2	10	23363	660	620	636	0.41	1555
p5.3.x	66	3	40.00	3	10	22357	685	615	643	0.41	1555
p5.3.x	66	3	40.00	4	10	7311	555	475	498	0.32	1555
p5.3.x	66	3	40.00	5	10	54239	750	630	668	0.43	1555
p7.4.t	102	4	100.00	1	10	143760	542	472	506	0.47	1077
p7.4.t	102	4	100.00	2	10	42255	504	471	485	0.45	1077
p7.4.t	102	4	100.00	3	10	45952	542	463	488	0.45	1077
p7.4.t	102	4	100.00	4	10	11587	322	268	284	0.26	1077
p7.4.t	102	4	100.00	5	10	96642	509	478	491	0.46	1077

Configuraciones:

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

Como síntesis de estos resultados digo que la configuración básica poco influye en el beneficio final de la solución. En el caso de la instancia 'p5.3.x', el  $i_f$  siempre se encuentra en el intervalo  $[0.41, 0.43]$  y en 'p7.4.t' el intervalo es  $[0.45, 0.47]$ . Para ambas instancias hay una configuración que es claramente peor y es la configuración  $C = 4$  donde se utiliza el decodificar **simple** en vez del **goloso**. Luego claramente en esta versión del BRKGA el decodificador tiene gran impacto en resultado final. Lamentablemente el resto de las configuraciones impacta muy poco en el beneficio total cuando la instancia del problema es grande (Minima cantidad de iteraciones, minima cantidad de iteraciones sin cambios, tamaño de la población, población elite, etc). Si impactan en el tiempo en que finaliza el algoritmo. Cuando llegue a este punto en mi trabajo, considere que tenía dos opciones por donde continuar. La primera, crear nuevos decodificadores. La segunda, agregar heurísticas de búsqueda local para aplicar a las mejores individuos de cada generación. Tome el camino de la segunda opción.

### 5.3. Heurísticas de búsqueda local

En pos de optimizar los resultados mencionados anteriormente se implementaron heurísticas de búsqueda local. La idea fue aplicar estas heurísticas a la mejores  $x$  soluciones de cada nueva generación. Donde  $x$  toma el valor de la propiedad *ApplyHeuristicsToTop*. En caso de que a la solución ya se le hubiese aplicado las heurísticas 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 a la siguiente generación. La idea de implementar heurísticas de búsqueda local la obtuve

de la publicación *A guided local search metaheuristic for the team orienteering problem*. de Vansteenwegen et al. [23]. Todas las heurísticas de búsqueda local al modificar una solución, la misma sigue siendo una solución válida. Es decir se siguen respetando las restricciones de distancia y la cantidad de vehículos.

### 5.3.1. Center of Gravity

Para las heurísticas de búsqueda local Insert y Replace se deben tomar una lista de clientes a considerar con algún orden. Este orden es importante ya que queremos empezar por las mejores opciones. El orden de clientes que se utiliza es por su distancia al centro de gravedad (COG) de una ruta. A menor distancia, mayor prioridad tendrá el cliente. Implementé el cálculo de COG de la forma que lo describen Vansteenwegen et al. [23]. La coordenada COG de una ruta se calcula con las siguientes fórmulas:

$$x_{cog} = \left( \sum_{\forall i \in ruta} x_i * B_i \right) / \sum_{\forall i \in ruta} B_i \quad (5.1)$$

$$y_{cog} = \left( \sum_{\forall i \in ruta} y_i * B_i \right) / \sum_{\forall i \in ruta} B_i \quad (5.2)$$

Donde  $x_i$  e  $y_i$  son las coordenadas de un cliente de la ruta y  $B_i$  es su beneficio. El cálculo de COG tiene una complejidad de  $O(ruta.Length)$ . No es tan costoso, de todos modos como no quiero realizar cálculos innecesarios, el COG de una ruta solo se calcula cuando se necesita, es decir cuando se es. Además se calcula una vez y se mantiene en memoria. Cuando se modifica la ruta, actualizo el COG.

Sea  $r$  una ruta:

$$r.x_{cog} = \frac{\sum_{\forall i \in ruta} x_i * B_i}{\sum_{\forall i \in ruta} B_i} = \frac{r.x_{cog.num}}{r.x_{cog.den}} \quad (5.3)$$

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

$$r'.x_{cog} = \frac{\sum_{\forall i \in ruta \wedge i \neq j} x_i * B_i}{\sum_{\forall i \in ruta \wedge i \neq j} B_i} = \frac{r.x_{cog.num} - x_j * B_j}{r.x_{cog.den} - B_j} \quad (5.4)$$

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

$$r''.x_{cog} = \frac{(\sum_{\forall i \in ruta} x_i * B_i) + x_k * B_k}{(\sum_{\forall i \in ruta} B_i) + B_k} = \frac{r.x_{cog.num} + x_k * B_k}{r.x_{cog.den} + B_k} \quad (5.5)$$

Luego actualizar el COG de una ruta al agregar ó remover un cliente tiene una complejidad de  $O(1)$  si guardamos en memoria y actualizamos los valores de  $x_{cog.num}$  y  $x_{cog.den}$  para cada ruta (Lo mismo aplica para la coordenada  $y$ ).

### 5.3.2. Swap

El objetivo de esta heurística es encontrar e intercambiar clientes entre dos rutas distintas con el fin de disminuir la suma de las distancias recorridas de ambas rutas mientras sigan respetando la restricción de distancia máxima por vehículo. Es decir dados  $v_a$ ,  $v_b$  vehículos y sus respectivas rutas  $r_a$ ,  $r_b$ , 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  $r_a$ , valen:

$$r_a.Dist + r_b.Dist < r'_a.Dist + r'_b.Dist$$

$$r'_a.Dist \leq v'_a.dMax$$

$$r'_b.Dist \leq v'_b.dMax$$

Al aplicar esta heurística a una solución, para todos par de rutas se ejecuta el método *SwapDestinationsBetween*. Si hay  $n$  rutas, la cantidad de pares de rutas es  $n * (n - 1) / 2$ . *SwapDestinationsBetween* prueba cada cliente de la ruta  $a$  con cada cliente de la ruta  $b$ , si efectivamente conviene hacer un swap, lo realiza y continua probando pares de clientes. De modo de no estar cambiando multiples veces un mismo cliente entre dos rutas en una misma ejecución de *SwapDestinationsBetween*, cuando se cambia de ruta a un cliente, se lo banea de cambios hasta que no termine la actual ejecución de *SwapDestinationsBetween*. La metaheurística Swap no mejora el beneficio total de una solución. Lo que hace es disminuir la distancia recorrida de alguna ruta, lo que aumente la probabilidad de encontrar algun cliente no visitado que se pueda insertar en alguna de las rutas modificadas.

```
// Dentro de clase SwapHeuristic
public void ApplyHeuristic(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);
    }
    if(changed)
        solution = Encoder.UpdateRandomKeys(solution);
}
```

```

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 ApplyHeuristic 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 heurística es encontrar una posición en alguna ruta para un cliente no visitado. Básicamente para cada vehículo y cada cliente no visitado se busca en que posición se puede insertar minimizando el incremento de distancia. Si la distancia resultante es menor a la distancia máxima del vehículo, se inserta el cliente en tal posición. El orden en que se toman los clientes no visitados es según su distancia al COG de la ruta a optimizar, de forma ascendente.



```

// Dentro de clase InsertHeuristic
public void ApplyHeuristic(Solution solution)
{
    // Lista de los clientes no visitados
    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;
            }
        }
    }
    if(changed)
        solution = Encoder.UpdateRandomKeys(solution);
}

```

Como menciones en sobre la implementacion de COG, solo se utiliza cuando es necesario. Luego para cada vehiculo lo primero que hace es activar el COG. Segundo, por cada cliente no visitado hasta el momento se analiza el insert. El método *AnalyzeInsert* retorna un objeto que tiene seteado dos campos importantes. Un campo bool que denota que el cliente puede ser insertado o no en el vehiculo consultado. Y otro campo que tiene la posicion donde se debe insertar, en caso de poder insertarlo. En caso de poder insertar el cliente, este se lo agrega al vehiculo que actualiza su COG, y se remueve el cliente de la lista de no visitados. El orden de complejidad del método *ApplyHeuristic* de la clase *InsertHeuristic* es:

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

#### 5.3.4. 2-opt

El algoritmo 2-opt es un simple algoritmo de búsqueda local propuesto por Croes [10]. El fin es buscar un orden alternativo de los clientes visitados en una dentro de una misma ruta, de modo que disminuya la distancia recorrida de la misma. Es decir, un swap de posiciones de dos clientes dentro de una misma ruta.

```

// Dentro de clase 2-opt
public void ApplyHeuristic(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++;
    }
    if(changed)
        solution = Encoder.UpdateRandomKeys(solution);
}

private bool Do2OptSwap(Vehicle vehicle)
{
    var changed = false;
    var combinations =
        GetCombinationsFor(vehicle.Route.GetDistance());
    var index = 0;
    while (index < combinations.Count)
    {
        var position1 = combinations[index].Item1 - 1;
        var position2 = combinations[index].Item2 - 1;
        var swaped = vehicle.Route.SwapIfImprovesDistance(position1,
            position2);
        if (!swaped)
            index++;
        changed = true;
        index = 0;
    }
    return changed;
}

```

Basicamente a cada vehículo le aplica el 2-opt. El método Do2OptSwap primero obtiene una lista de las permutaciones posibles. Luego por cada combinacion intenta hacer un swap dentro de la ruta. Si el swap sucede, luego vuelve a empezar desde el principio ya que ese cambio puede generar nuevos cambios. Como el swap solo sucede cuando la nueva distancia es estrictamente menor, el loop siempre termina. Una ruta no puede estar mejorando infinitamente. Este algoritmo tiene un order de complejidad  $O(\text{vehiculos} * \text{mediaClientesEnRuta} * (\text{mediaClientesEnRuta} - 1)/2)$   $O(\text{vehiculos} * \text{mediaClientesEnRuta}^2/2)$ .

### 5.3.5. Replace

Esta heuristica busca intercambiar un cliente no visitado por uno visitado de una ruta de modo que aumente el beneficio de la ruta. Del mismo modo que la heuristica insert, los clientes no visitados se toman en orden segun su distancia al COG de la ruta, empezndo por los mas cercanos.

```

// Dentro de la clase ReplaceHeurísticas
public void ApplyHeuristic(Solution solution)
{
    var vehicles = solution.Vehicles;
    var changed = false;
    foreach (var vehicle in vehicles)
        changed = changed || Replace(solution, vehicle);
    if(changed)
        solution = Encoder.UpdateRandomKeys(solution);
}

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 removedClient = RemoveWorstOrDefault(vehicle, client);
            changed = changed || removedClient.Id != client.Id;
        }
        else
            changed = true;
    }
    return changed;
}

```

La heurística de replace es muy parecida al insert y esto se refleja en el pseudocódigo. Ante un cliente no visitado, lo primero que hace es analizar si se puede insertar y cual es la mejor posición donde se puede insertar, llamando al método *AnalyzeInsert*. Luego sin siquiera verificar si realmente se puede insertar, lo inserta en la posición que menos incrementa la distancia de la ruta. Este es el único momento de toda la implementación donde puede existir una solución inválida. Por lo tanto ahora se fija si realmente se podía insertar. En caso afirmativo, continua con el siguiente cliente. En caso contrario remueve de la ruta el peor destino, que en el peor de los casos es el mismo cliente que se acaba de insertar. Cualquier otro cliente que se remueva de la ruta significará que se efectuó un replace y aumento el beneficio total de la ruta. Replace tiene una complejidad de  $O(\text{vehículos} * \text{clientesNoVisitados} * \text{mediaClientesEnRuta} * 2)$ .

### 5.3.6. Encoder

Agregar heurísticas de búsqueda local entre generación de poblaciones conlleva un problema que debe resolverse. Al mejorar la solución, se modifican sus rutas. Ahora bien, si no se modifica su genética, es decir su *RandomKeys*, sus descendientes heredarán los

genes que generan una solución no optimizada por las búsquedas locales. Del mismo modo que una persona no hereda las cirujías estéticas de sus padres.

$$\text{ApplyHeuristic}(s) \neq \text{Decoder.Decode}(s.\text{RandomKeys}, \text{ProblemInfo})$$

Para solucionar esto se debe modificar el vector aleatorio de enteros, *RandomKeys*, de la solución mejorada de modo que al decodificar tal vector aleatorio de enteros, genere la solución modificada. 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 *PositionIndex*. Luego el primer cliente con el que trabaja el decoder, es el cliente con menor *Key* de su *RandomKey* asociado. Algo que no mencioné en Decoder 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 *tMax* en el benchmark de instancias de problemas utilizado. De todos modos ahora debo respetar la decisión que tome en el decoder. Por lo tanto tomo el primer cliente de la ruta del vehículo con menor *Id* y a ese cliente le voy a asociar el *RandomKey* que tenga el menor *key*. Así, cuando el decoder inicie lo primero que hará es tomar este cliente e intentará adjudicárselo al primer vehículo que será el de menor *Id*. Luego tomaré el segundo cliente del mismo vehículo y le asignaré el segundo *RandomKey* de menor *Key*. Y así sucesivamente, hasta tener mapeados todos los clientes del primer vehículo con su nuevo *RandomKey*. Luego repito el procedimiento con los clientes del siguiente vehículo ordenados por *Id* ascendente. Finalmente, no tendré más vehículos y quedará un resto de clientes no visitados a los cuales le tengo que asociar algún *RandomKey*. A estos clientes podría asignarle cualquier *RandomKey*. Aún así, en pos de disminuir los cambios genéticos sobre el individuo, les asigne un *RandomKey* tal que entre ellos mantuvieran el mismo orden que mantenían antes de la optimización. Es decir, dentro de los clientes no visitados, el cliente que previamente tenía el *RandomKey* con menor *Key*, le asigne el *RandomKey* de menor *Key* que había disponible. Con esto me aseguro que todos los clientes que estaban en el primer vehículo, estén en el primer vehículo al decodificar el *random keys*. Este proceso puede verse en la figura 5.4 5.5.

Ahora bien, digamos que existe un escenario donde el decoder se encuentra trabajando con un *RandomKeys* que fue generado por el encoder y sucede lo siguiente. Al agregar al último cliente de la primera ruta, como es un decoder goloso, intentará agregar a la primera ruta al resto de los clientes y supongamos que efectivamente encuentra uno. Cuando vi la posibilidad de este escenario, decidí agregar unos delimitadores de modo que si el decoder se encuentra con uno, siempre cambie de vehículo. Estos delimitadores son modelados por la propiedad, *ForceVehicleChangeAfterThis*, con la que extendía al objeto *RandomKey*. De este modo el Decoder, luego de utilizar el *RandomKey*, se fija si *ForceVehicleChangeAfterThis* es *true* y si lo es, deja de intentar agregar clientes a la ruta del vehículo actual, pasando al siguiente. Los delimitadores no son heredados a los descendientes. Estos delimitadores me aseguran que valga:

$$\text{ApplyHeuristic}(s) = \text{Decoder.Decode}(s.\text{RandomKeys}, \text{ProblemInfo}) \quad (5.6)$$

En caso de no tener los delimitadores, como dije antes, una ruta podría tener un cliente extra, que para fines del algoritmo, no es malo. Al agregar un cliente, hay dos opciones.

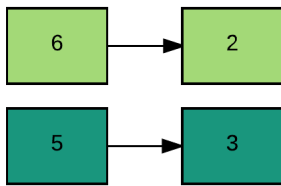
Fig. 5.4: Como se mejora una solución

Dado el siguiente vector ordenado de RandomKeys:

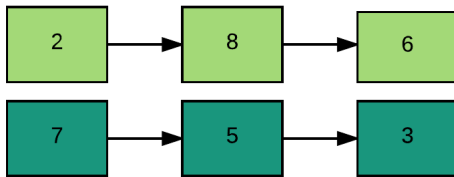
Key	7	13	21	27	45	54	79	89
PositionIndex	6	2	5	1	4	8	3	7

Hash de la solución generada: 62514837

El decodificar goloso genera una solución que contiene las rutas:



Luego las búsquedas locales mejoran tales rutas agregando algunos clientes y modificando el orden del recorrido:



La primera es que el cliente perteneciera al subconjunto de clientes que no estaba visitado. En este caso aumentaría el beneficio total. Pero para que esto suceda, la heurística de insert no debería haber funcionado correctamente ó no fue la última en aplicarse. En caso de que el cliente perteneciera a otro vehículo, el beneficio total no se modifica. El peor escenario es que este segundo vehículo se quede con el cliente de un tercer vehículo, sucediendo un especie de cambios en cadenas de clientes entre una secuencia de vehículos sin decrementar el beneficio total. Luego por mas que tomé la decisión de agregar los delimitadores, considero que ambas opciones eran igual de buenas considerando solamente la función objetivo. Como agregar los delimitadores asegura la validez de la ecuación 5.6.

```

public static Solution UpdateRandomKeys(Solution s);
{
    var randomKeys = s.GetOrderedRandomKeys();
    // Todas las keys ordenadas ascendente
    var keys = randomKeys.Select(k => k.Key).ToList();
    var positionIndexes = new List<int>();
    // Get Position Indexes from Visited Clients
    foreach (var r in s.Routes)
    {
        var d = r.GetDestinations();
        var rpi = d.Select(d => d.PositionIndex);
        positionIndexes.AddRange(rpi);
    }
}
  
```

Fig. 5.5: Como se actualiza el *RandomKeys* y hash de una solución luego de mejorar la solución 5.5

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

Hash de la nueva solución: 28675314

```
// Get Position Indexes from Unvisited Clients
var uPositionIndexes = GetUnvisitedPositionIndexes(randomKeys,
    newRoutes);
positionIndexes.AddRange(unvistedPositionIndexes);

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

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],
        PositionIndex = positionIndexes[index],
        ForceVehicleChangeAfterThis = endRoute
    };
    newRandomKeys.Add(randomKey);
}
s.SetRandomKeys(newRandomKeys);
return s;
}
```

### 5.3.7. Secuencia de Aplicación de las Heurísticas

Todas las heurísticas mencionadas, implementan el interfaz:

```
public interface ILocalSearchHeuristic
{
    void ApplyHeuristic(Solution solution);
}
```

Esto lo hice así de modo de poder inicializar el vector de heurísticas a aplicar en la creación del algoritmo BRKGA. Luego al momento de aplicar las búsquedas locales, llama en orden de inserción de tal vector y ejecuta su método ApplyHeuristic. Así se pueden testear múltiples ordenes distintos de búsquedas locales con un simple modificación del parámetro de entrada del constructor del BRKGA. El orden en que las heurísticas se aplican es muy relevante. Por ejemplo, si la última heurística que aplicamos es Swap o 2-Opt, luego la distancias disminuidas de las rutas no sería aprovechado por ninguna cliente.

Configuraciones:

- **Comun a todas:** MI.200;MNC.50;PS.150;EP.0,3;MP.0,1;EGC.70;TOP.2
- **C = 1:** HEU.IROS;D.G
- **C = 2:** HEU.SIORSOR;D.G
- **C = 3:** HEU.SIORSOR;D.S
- **C = 4:** HEU.SOIR;D.G
- **C = 5:** HEU.SOIR;D.S
- **C = 6:** HEU.SOISOIR;D.G

Recordando los códigos de HUE:

- **I:** Insert (Cliente no visitado)
- **R:** Replace (Cliente no visitado por uno visitado)
- **O:** 2-Opt (Swap dentro de una misma ruta)
- **S:** Swap (Swap entre dos rutas distintas)

Genere soluciones para las mismas seis instancias diversas, varían en la configuración solamente el orden de las búsquedas locales y el decodificador obteniendo los siguientes resultados:

$I$	$\#N$	$\#V$	$tMax$	$C$	$\#S$	$t_{avg}$	$B_{max}$	$B_{min}$	$B_{avg}$	$i_f$	$BTP_{max}$
p2.2.k	21	2	22.50	1	10	4724	275	260	267	0.97	275
p2.2.k	21	2	22.50	2	10	5136	275	260	268	0.97	275
p2.2.k	21	2	22.50	3	10	3333	275	260	268	0.97	275
p2.2.k	21	2	22.50	4	10	4410	270	260	269	0.98	275
p2.2.k	21	2	22.50	5	10	2610	270	260	267	0.97	275
p2.2.k	21	2	22.50	6	10	4884	270	265	269	0.98	275
p2.3.g	21	3	10.70	1	10	2710	145	145	145	1.00	145
p2.3.g	21	3	10.70	2	10	3025	145	145	145	1.00	145
p2.3.g	21	3	10.70	3	10	2047	145	145	145	1.00	145
p2.3.g	21	3	10.70	4	10	2880	145	145	145	1.00	145
p2.3.g	21	3	10.70	5	10	1954	145	145	145	1.00	145
p2.3.g	21	3	10.70	6	10	2883	145	145	145	1.00	145
p3.4.p	33	4	22.50	1	10	10613	540	510	524	0.94	560
p3.4.p	33	4	22.50	2	10	11692	540	540	540	0.96	560
p3.4.p	33	4	22.50	3	10	5361	560	540	550	0.98	560
p3.4.p	33	4	22.50	4	10	10341	540	540	540	0.96	560
p3.4.p	33	4	22.50	5	10	4101	560	540	550	0.98	560
p3.4.p	33	4	22.50	6	10	11247	560	540	543	0.97	560
p5.3.x	66	3	40.00	1	10	32077	1435	1340	1382	0.89	1555
p5.3.x	66	3	40.00	2	10	39871	1505	1475	1489	0.96	1555
p5.3.x	66	3	40.00	3	10	19296	1505	1470	1487	0.96	1555
p5.3.x	66	3	40.00	4	10	32672	1485	1450	1464	0.94	1555
p5.3.x	66	3	40.00	5	10	10854	1490	1425	1454	0.94	1555
p5.3.x	66	3	40.00	6	10	35290	1490	1455	1469	0.94	1555
p7.2.e	102	2	50.00	1	10	14956	290	280	286	0.99	290
p7.2.e	102	2	50.00	2	10	15837	290	285	289	1.00	290
p7.2.e	102	2	50.00	3	10	8859	290	289	289	1.00	290
p7.2.e	102	2	50.00	4	10	13623	290	290	290	1.00	290
p7.2.e	102	2	50.00	5	10	6605	290	285	289	1.00	290
p7.2.e	102	2	50.00	6	10	14637	290	281	288	0.99	290
p7.4.t	102	4	100.00	1	10	57849	988	925	949	0.88	1077
p7.4.t	102	4	100.00	2	10	66773	1008	978	995	0.92	1077
p7.4.t	102	4	100.00	3	10	33420	1020	979	997	0.93	1077
p7.4.t	102	4	100.00	4	10	52095	1003	957	983	0.91	1077
p7.4.t	102	4	100.00	5	10	18879	1015	963	989	0.92	1077
p7.4.t	102	4	100.00	6	10	57895	1012	955	983	0.91	1077

Observaciones de estos resultados:

Claramente el peor orden de heurísticas locales es IROS (Insert, Replace, 2-Opt, Swap). Esto es por lo que explicamos anteriormente, Insert y Replace intentan agregar mas clientes o intercambiar 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 primero los de beneficio y luego los de distancia, no se utiliza la distancia ahorrada.



Luego el mejor orden segun estos resultados, fue SIORSOR. Notar que repito heurísticas en este orden, esto 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 luego de un Swap que modifíco rutas de forma tal que quizá disminuyo la distancia recorrida.

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 vs el goloso. Esto lo logra con un  $i_f$  prácticamente igual en todas las intancias. Lo mismo podemos observar entre la configuración 4 y 5. Por lo tanto, habiendo agregado las heurísticas de búsqueda local, 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.

## 6. RESULTADOS

En este capítulo mostraré los resultados finales obtenidos. Dentro de todos los papers de trabajos previos que encontré que incluyeran resultados que utilizaran el benchmark de instancias de Chao y los de Tsiligrides, seleccioné el de Archetti et al. [1]. En su paper presentan varias soluciones, algunas utilizando un tabu search y otras utilizando una búsqueda de vecindario variable. Luego las comparan con los resultados de Chao et al. Chao [9] y 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.

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 heurísticas 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 solucion, la solucion media y la mejor solucion 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<sub>min</sub></i>	<i>B<sub>avg</sub></i>	<i>B<sub>max</sub></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>Bmin</i>	<i>Bavg</i>	<i>Bmax</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 problema de orientación de equipo es el problema donde se puede visitar un grupo de clientes, con un beneficio garantizado para cada visita. Un equipo de vehículos está disponible y cada miembro del equipo puede visitar cualquier conjunto de clientes dentro de un límite de tiempo determinado. El beneficio de cada cliente puede ser recaudado por un solo vehículo como máximo. El problema combina la decisión de qué clientes seleccionar con la decisión de cómo planificar la ruta.

Al ser un problema reconocido como un modelo de muchas aplicaciones reales diferentes, 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. Y el resto obtuvo un  $i_f$  en el intervalo  $[0.95, 0.99]$ , salvo por algunas excepciones. Ahora, si considero el resultado obtenido por el BRKGA puro, los resultados son muy malos para instancias grandes del problema, llegando a tener un  $i_f$  aproximado de '0.5'. Esto no me deja satisfecho con el BRKGA. De hecho creo que los individuos resultantes del método de apareamiento son muy distintos a sus individuos progenitores. Lo que significaría que estoy explorando el dominio de soluciones de una manera errática ó aleatoria. Luego el algoritmo de BRKGA solo puede llegar a buenas soluciones con la ayuda de otras metaheurísticas como es en este caso.

Uno de los grandes problemas del BRKGA para TOP es que su secuencia genética no es utilizada completamente ya que parte del problema es que no todos los clientes son visitables dadas las restricciones del problema. Asignar todos los clientes a algun vehiculo siempre generaría una solución no factible. De serlo, no sería TOP. Este problema de completitud entre genes y clientes visitados quiza puede ser resuelto modificando lo que representa un gen. Es decir, con un decoder totalmente diferente.

### 7.1. Trabajos Futuros

Para empezar me hubiese gustado 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 cercanos son. De todos modos, para un analisis 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 necesita mejoras, no estoy contento con los resultados obtenidos utilizando solo el BRKGA. Si tuviera tiempo hubiese encarado por los dos ángulos que creo que pueden impactar fuertemente en su beneficio final.

Primero por el decodificador. Buscar alguna manera de que su secuencia genética sea 'completa', es decir que cada gen impacte en la formación de la solución. Para entender esto tomar como ejemplo el decodificador simple. Sea una instancia con 10 clientes y una solución generada a partir de su secuencia de genes que visita a 6 de los 10 clientes, luego existen 4 alelos/genes que no impactan en el resultado final. Quizá no sea parte del problema que tiene BRKGA, pero no me imagino una especie con millones de años de evolución que llegue a tener un 40

El segundo ángulo 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 algun proceso de limpieza se muevan los clientes que se visitan de forma repetida y se incluyan otros. En este contexto, la cantidad de genes 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 a las heurísticas de búsqueda local, podría haber implementado la heurística 'move', para mover un cliente visitado de una ruta hacia otra, acumulando mayor distancia libre en una sola ruta. También podría optimizar la heurística '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 podría haber implementado alguna heurística local tabu de modo de salir de mínimos locales. Estas optimizaciones no las realice por que ya había llegado a muy buenos resultados al agregar las búsquedas locales y no tuve más tiempo.

## 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] Instancias de 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).