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

Biased Random Key Genetic Algorithm for the 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

ALGORITMO GENÉTICO DE CLAVE ALEATORIA SESGADA PARA EL PROBLEMA DE ORIENTACIÓN DE EQUIPO

En el problema de orientación (OP), se da un conjunto de vértices, cada uno con un puntaje determinado. El objetivo es determinar un camino, limitado en su longitud, que visita a algunos clientes a modo de maximizar la suma de los puntajes obtenidos. La orientación se puede formular de la siguiente manera. Dado n nodos en el plano euclidiano, cada uno con un puntaje, donde $puntaje(vertice_i) \geq 0$ y $puntaje(vertice_1) = puntaje(vertice_n) = 0$, encuentre una ruta de puntaje máximo a través de estos nodos, inciando en $vertice_1$ y termina en $vertice_n$ de longitud no mayor que $tMax$.

El problema de orientación de equipo (TOP) es la generalización al caso de múltiples recorridos del problema de orientación, también conocido como Problema de vendedor ambulante selectivo (TSP). TOP implica encontrar un conjunto de rutas desde el punto de inicio hasta el punto final, de modo que la recompensa total obtenida al visitar un subconjunto de ubicaciones se maximice y la longitud de cada ruta esté restringida por un límite preestablecido. TOP es conocido como un problema NP-completo. La solución a este problema requiere no solo determinar un orden para cada recorrido, sino también seleccionar qué subconjunto de vertices en el gráfico visitar.

En esta tesis, se propone un enfoque de algoritmo genético de clave aleatoria sesgada (BRKGA) para el problema de orientación de equipo. Los algoritmos genéticos de clave aleatoria inicializaron su población con un conjunto de vectores de clave aleatoria y un decodificador. El decodificador convierte un vector de clave aleatoria en una solución válida del problema. Los algoritmos genéticos de clave aleatoria sesgada son una variante de los algoritmos genéticos de clave aleatoria, donde uno de los padres utilizado para el apareamiento está predispuesto a tener una mejor forma física que el otro progenitor.

Además, en cada nueva generación, la mejor solución no mejorada se mejora con una secuencia de heurística de búsqueda local. Las heurísticas de búsqueda local como Insertar, Cambiar, Reemplazar y 2-opt se utilizan para encontrar mejores soluciones locales para 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 (CGW), Tang y Miller-Hooks (TMH) y Archetti, Hertz, Speranza (AHS). Aunque mis resultados fueron muy buenos y competitivos en la mayoría de las intancias, en algunos no fueron tan buenos como mencioné trabajos anteriores.

Keywords: Problema de Orientación de Equipo, Algoritmo Genético de Clave Aleatoria Sesgada, Problema de Enrutamiento, Heurística de Búsqueda Local, Construcción de Soluciones Golosas.

BIASED RANDOM KEY GENETIC ALGORITHM FOR THE TEAM ORIENTEERING PROBLEM

In the orienteering problem (OP), a set of vertices is given, each with a certain score. The objective is to determine a path, limited in length, that visits some clients in order that maximises the sum of the collected scores. Orienteering can be formulated in the following way. Given n nodes in the Euclidean plane each with a score, where $score(vertex_i) \geq 0$ and $score(vertex_1) = score(vertex_n) = 0$, find a route of maximum score through these nodes beginning at $vertex_1$ and ending at $vertex_n$ of length no greater than $tMax$.

The team orienteering problem (TOP) is the generalization to the case of multiple tours of the Orienteering Problem, know also as Selective Traveling Salesman Problem (TSP). TOP involves finding a set of paths from the starting point to the ending point such that the total collected reward received from visiting a subset of locations is maximized and the length of each path is restricted by a pre-specified limit. TOP is known as an NP-complete problem. The solution to this problem 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 biased random key genetic algorithm (BRKGA) approach is proposed for the team orienteering problem. Random-key genetic algorithms initialized their population with a set of random-key vectors and a decoder. The decoder converts a random-key vector in a valid solution of the problem. Biased random-key genetic algorithms are a variant of random-key genetic algorithms, where one of the parents used for mating is biased to be of higher fitness than the other parent.

Also, in every new generation, the best unenhanced solution is enhanced with a sequence of local search heuristics. Local search heuristics such as Insert, Swap, Replace and 2-opt are used in order to find better local solutions for a given solution.

Computational experiments are made on standard instances. Then, this results, were compared to the results obtained by Chao, Golden, and Wasil (CGW), Tang and Miller-Hooks (TMH) and Archetti, Hertz, Speranza (AHS). Though my results where very good and competitive in most intances, in some they were not as good as mentioned previous works.

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

Índice general

1. INTRODUCCIÓN

1.1. Historia

Orientación (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 cotrol 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 problema de orientación de un solo competidor (Single-Competitor 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. Lo llamamos el Problema de Orientación de Equipo (Team Orienteering Problem) y lo denotan por TOP.

Notar que la versión de un solo competidor (OP) de este problema ha demostrado ser NP-dura por Golden, Levy, y Vohra [9], por lo que el TOP es al menos tan difícil. Por lo tanto, la mayoría de la investigación sobre estos problemas se han centrado en proporcionar enfoques heurísticos.

1.2. Aplicaciones de TOP

El TOP ha sido reconocido como un modelo de muchas aplicaciones reales diferentes. El juego deportivo de orientación de equipo por Chao et al [6]. Algunas aplicaciones de servicios de recogida o entrega que implican el uso de transportistas comunes y flotas privadas por Ballou y Chowdhury [17]. También hay plicaciones en varios campos, como la planificación de viajes turísticos, enrutamiento técnico y reclutamiento de atletas.

El problema de entrega de combustible con vehículos múltiples de Golden, Levy y Vohra [9]. 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.

El reclutamiento de jugadores de fútbol americano universitario de Butt y Cavalier [4]. 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.

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.

1.3. Como se modelo TOP en nuestra solucion

Para la generación y comparación de resultados se utilizaron instancias de test de Tsiligirides y de Chao. Las intancias de Tsiligirides y de Chao comparten el mismo formato.

Una instancea 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 intancias de test utilizadas todos los vehículos tienen el mismo valor de distancia máxima.
- M clientes. Cada cliente tiene un beneficio mayor a cero. Además tienen un set de coordenadas X e Y que representan su ubicación en un plano cartesiano.
- Un punto de inicio y fin de ruta para cada vehículo. Ambos puntos tienen un beneficio de cero y tienen un set de coordenadas X e Y.

También es importante mencionar que:

- 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.
  - Ningun cliente pertenece a dos rutas distintas.
  - Toda ruta parte del vértice de inicio y finaliza en el punto de vértice.
- La función objetivo retorna la sumatoria de los beneficios de los clientes visitados.

2. REVISIÓN BIBLIOGRÁFICA

Existen una gran cantidad de aplicaciones que pueden ser modelas por TOP. Es por esto que la clase de problemas de enrutamiento de vehículos con ganancias es amplia. En el 2013, C. Archetti, M.G. Speranza, D. Vigo [1] publicaron una encuesta sobre esta clase de problemas. En este capitulo voy a referenciar varias de la publicaciones que referenciaron, y otras trabajos que encontre relacionados con TOP.

La primera heurística propuesta para el TOP es un algoritmo de construcción simple introducido en Butt y Cavalier [4] y probado en pequeñas instancias de tamaño con hasta 15 vértices. En su heuristica *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 cuan beneficio son los nodos y las sumas de las distancias de ir y volver por esos vertices. Fue introducido con el nombre *Multiple Tour Maximum Collection Problem*.

Una heurística de construcción más sofisticada se da en Chao, Golden y Wasil (CGW) [6]. Este es la primer publicacion donde aparece e nombre TOP. Este nombre fue acuñado por Chao et al. para resaltar la conexión con el más ampliamente estudió 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 procedimiento de solución óptima para TOP es propuesto por Butt y Ryan [5]. 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. [18] presentaron un algoritmo de ramificación y precio. 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. [6].

Luego, se aplicaron varias metaheurísticas al TOP, partiendo del algoritmo de búsqueda tabú introducido en Tang y Miller-Hooks (TMH) [12], que está incorporado en un procedimiento de memoria adaptativa (AMP) que alterna entre vecindarios pequeños y grandes durante la búsqueda. La heurística de búsqueda tabú propuesta 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 tabu 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. Notese que como señala Golden et al. [7], 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. [2] proponen dos variantes de un algoritmo de búsqueda tabú generalizada y un algoritmo de búsqueda de vecindario variable (VNS). Dada una solucion titular s, 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 tabu 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 tabu generalizada, 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. [10] utilizan un enfoque de optimización de colonia de hormigas (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. Para resolver un problema de optimización combinatoria estática con ACO, 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.

TODO: Hablar un poco de cada uno de los siguientes:

1. A guided local search metaheuristic for the team orienteering problem. Vansteenwegen et al. [13].

2. A path relinking approach for the team orienteering problem. Souffriau et al. [19].

3. A memetic algorithm for the team orienteering problem Bouly et al. [3].

4. A PSO-based memetic algorithm for the team orienteering problem. Dang et al. [8], este último siendo el mejor actual en la clase.

5. Solving the team orienteering problem Souffriau et al. [19].

6. A greedy randomized adaptive search procedure for the team orienteering problem Ferreira et al. [20].

7. A novel Randomized Heuristic for the Team Orienteering Problem Zettam y Elbenani [21].

En la investigación sobre los trabajos realizados, no se encontraron trabajos que implementen algoritmos geneticos. En este trabajo se propone resolver TOP utilizando biased random key generation algorithim (BRKGA)

3. BIASED RANDOM KEY GENETIC ALGORITHM

3.1. Algoritmos Genéticos

Los algoritmos genéticos, o GAs (Genetic Algorithims), [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 tiene un cromosoma correspondiente que codifica la 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 resuelven un conjunto de individuos que forman una población a lo largo de varias generaciones. En cada generación, una nueva población se crea combinando elementos de la población actual para producir hijos que conforman la próxima generación. La mutación aleatoria también tiene lugar en algoritmos genéticos como un medio para escapar de atrapamiento en mínimos locales. El concepto de supervivencia del más apto juega en los algoritmos genéticos cuando los individuos son seleccionados para aparearse y producir descendencia. Los individuos son seleccionados al azar, pero aquellos con mejor estado físico son preferidos sobre aquellos que son menos aptos.

3.2. RKGA

Introducido por Bean [15], en los algoritmos genéticos con claves aleatorias, ó Random Key Genetic Algorithms (RKGA) los cromosomas son representados por un vector de números reales generados aleatoriamente en el intervalo [0, 1]. Un algoritmo determinístico, llamado decodificador, toma como entrada un cromosoma y asocia con ella una solución del problema de optimización combinatoria para el cual se puede calcular un valor objetivo o aptitud física. Los algoritmos RKGAs evolucionan una población de vectores de claves aleatorios sobre una serie de iteraciones llamadas generaciones. La población inicial se compone de p 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 $p - p_e$ no elite individuos donde $p_e < p - p_e$. Con el fin de evolucionar a la población, un RKGA utiliza una estrategia elitista ya que todos los individuos de élite de la generación k se copian sin cambios a la generación $k + 1$. Esta estrategia mantiene un seguimiento de las buenas soluciones encontradas durante las iteraciones del algoritmo que resulta en una heurística de mejora monotónica. La mutación se utiliza para permitir que los GAs escapen de la trampa en mínimos locales. En RKGA un mutante es un individuo generado a partir de un vector de claves aleatorias, de la misma manera que un elemento de la población inicial. Con la poblacion p_e elite y la p_m mutantes, un conjunto adicional de individuos $p - p_e - p_m$ es requerido para completar la generacion $k + 1$. Esto se hace generando una descendencia mediante el proceso de apareamiento.

3.3. BRKGA

En RKGA, Bean [15] selecciona dos padres al azar de toda la población. Un BRKGA (Biased Random Key Genetic Algorithms), difiere de RKGA en la forma en que los padres son seleccionados para el apareamiento. En un BRKGA, cada elemento se genera combinando un elemento seleccionado al azar de la partición de elite en la población actual y uno de la partición no elitista. En algunos casos, el segundo padre se selecciona de toda la población. Se permite la repetición en la selección de un padre y, por lo tanto, un individuo puede producir más de un hijo. Como $p_e < p - p_e$, la probabilidad de que un individuo de élite sea seleccionado para el apareamiento es mayor que la de un individuo no élite y 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 crossover uniforme parametrizado (Spears y DeJong [16]), 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 $i = 1, ..., n$ 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 p_e y el valor del e'_i del padre no elite con probabilidad $1 - p_e$. 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 de apareamiento es siempre válido, es decir, puede ser decodificado en una solución del problema de optimización combinatoria. En la $figuraX$ se puede observar como la evolución funciona. Los individuos son ordenados por su aptitud y marcados como elite y no-elite. Los vectores aleatorios de elite y pasan directamente a la siguiente generación. Un porcentage pequeño de la nueva generacion sera conformado por individuos mutantes, generados aleatoriamente como la poblacion inicial. Por último, el último conjunto de la nueva población se crean del cruce entre individuos de la población de elite con individuos de la población no-elite. En la figura el primer vector de cromosomas es de un individuo de elite, el segundo vector de cromosomas es de un edividuo no-elite. Se decide que cromosomas tomara el individuo hijo utilizando un vector aleatorio del mismo tamaño que los vectores de cromosmos. En este ejemplo se utiliza un $\rho_e = 0{,}70$, de este modo la descendencia se asemejara mas al padre de elite. Con estos cruces se completa la nueva generación de soluciones.

3.4. Decodificador del BRKGA

Una caracterisitca importante para mencionar del BRKGA es que que el decodificador es el unico modulo del algoritmo que requiere conocimiento del dominio del problema. El decodificador transforma un vector aleatorio de enteros en una instancia de una solucion del problema. Es un adaptador, por lo tanto si hacemos un decodificador para otro problema podriamos reutilizar el modulo de BRKGA. En el caso de esta impementación de BRKGA entre cada generación se ejecutan unas heuristicas de busqueda local sobre las mejores soluciones de la población. Como estas heuristicas trabajan sobre una instancia de la solución, requiere que la converción del decodificador sea reversible. Es decir, debe poder convertir una solución del problema en un vector de enteros que al convertirlo nuevamente en solución sea identica a la solución original.

4. 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 hacerce 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 [22]. Se compararon los resultados con los obtenidos de los trabajos previos de Chao et al. [6], de Archetti et al. [2] y los de Tang and E. Miller-Hooks [12]. 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 [22]. 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 intancia 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 simplemenete devidir 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 ilutrativo de la funcionalidad del algoritmo.

Para la valuación del índice de efectividad se eligieron seis intancias 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: Indice 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

4.1. 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).

4.1.1. 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 propositio 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 posicion y esa posicion 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 itervalo $[0, \#Clientes - 1]$. 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.

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

4.1.2. 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

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 metodo 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 4 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.

4.1.3. Caracteristicas 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 $i_0 = 0, i_1, i_2, .., i_m$ 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 solucion.

Un problema que tiene este decodificar es en la existencia de un cliente inalcansable, 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 solucion facil a este problema es filtrando todos los clientes inalcansables previo a la ejecucion del BRKGA.

Otro problema que tiene este decodificador es que cambia de vehiculo al primer intento fallido de expandir su ruta. Luego puede generar soluciones con rutas muy pequeñas, por este motivo implemente un Decodificador Goloso.

TODO Imagen de un vector de cuadrados pintados con colores segun a que vehiculo pertenece el cliente.

4.1.4. 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 siguiene 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 $0(\#clientes * \#vehiculos)$. De todos modos la cantidad de vehículos en todas las intancias del becnhmark utilizado siempre son menores o iguales a 4, luego no tiene un impacto en su performance. En promedio aumenta el profit de las soluciones generadas por el decodificador.

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

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

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.

4.2. BRKGA

En una primera intancia 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 poblacion. Es decir generamos una nueva generacion de soluciones a partir de la generación anterior como se explico en la sección BRKGA (ver sección 3.3).

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

4.2.1. 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 implementacion 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();
}
```

4.2.2. 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 utilza 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 heuristicas de busqueda local que se le aplicaran a la mejor solución de cada generación, a la cual no se le hayan aplicado las heuristicas locales aún. Se implementaron cuatro heuristicas 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 heuristicas de busqueda local.
- **DecoderType**: Clave **D**. Es una enumeracion 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

4.2.3. Inicialización de la Población

Utilizando $PopulationSize$ del objeto $Configuration$ seteo el tamaña de la población. Luego para generar la población inicial se genereran $PopulationSize$ 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.

4.2.4. 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 $MinIerations$. Es decir que el bucle principal cortaba luego de evolucionar la poblacion $MinIerations$ veces. Luego de analizar las últimas generaciones de la solución y ver que era frecuente que la mejor solución se habia generado recientemente, agregue una condicion de corte adicional. Ahora, para cortar ademas 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 genereciones 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);
}
```

4.2.5. Evolución de la población

Se toma la población y se los ordena descendientemente segun 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 incluídos 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 genereción 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 el la sección BRKGA (ver sección 3.3). Durante el apareamiento, no es tan extraño que se genere una solución identica 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 mutatants = Generate(MutatansSize);
  var evolvedPopulation = new pop(elites, mutatants);
  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 generedos a partir de un vector de RandomKeys, del mismo modo que la población inicial. Con el fín de mostrar lo simple que es la generación de un nuevo vector de RandomKeys, presento el pseudocodigo 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 deterministico. 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 unico 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, itercalados con un separador. Es decir, es el orden de la cuidades 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 genereron la solución ya existente, porque consideré que la probabilidad de volver a generar nuevamente una solución existente es alta. Ademas, aún si en un segundo intento no generera 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 desición para optimizar el apareamiento y disminuir soluciones dentro de un mismo vecindario por genereración.

```
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 de generar una solucion 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 ídenticos. Si además la solución repetida se ecuentra 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 llego al caso donde toda una población constituía de una unica 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 multiples soluciones repetidas.

4.2.6. 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 indices de efectividad que mostrare a continuación corresponden con una primera versión que no admitía repetidos y el objeto configuracion. Para estos primeros resultados genere resultados por intancia y utilice una configuración estandar sin heurisitcas locales aún:
'MI.250;MNC.10;PS.100;EP.0,3;MP.0,1;EGC.70;HEU.;TOP.0;MI.G' (ver sección 4.2.2).

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 heuristicas 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 observació 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 vés de 100 reduciendo la combinatoria de soluciones posibles.

Como estos resultados no eran satisfactorios, tomé las dos intancias de este subconjunto con menor i_f y las utilice para testear distintas configuracions. Probé multipes 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 sintesis de estos resultados digo que la configuracion basica 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 en 'p7.4.t' el intervalo es [0.45,0.47]. Para ambas intancias hay una configuración que es claramente peor y es la configuracion **C = 4** donde se utiliza el decodificar **simple** en vez del **goloso**. Luego claramente en esta version 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 poblacion, poblacion 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 heuristicas de busqueda local para aplicar a las mejores individuos de cada generación. Tome el camino de la segunda opción.

4.3. Heuristicas de busqueda local

En pos de optimizar los resultados mencionados anteriormente se implementaron heuristicas de busqueda local. La idea fue aplicar estas heurísticas a la mejores x soluciónes 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 heuristicas 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 indiviuos del conjunto de elite pasan a la siguiente generación. La idea de implementar heuristicas de busqueda local la obtuve

de la publicación *A guided local search metaheuristic for the team orienteering problem.* de Vansteenwegen et al. [13]. Todas las heuristicas de busqueda local al modificar una solucion, la misma sigue siendo una solucion valida. Es decir se siguen respetando las restricciones de distancia y la cantidad de vehiculos.

4.3.1. Center of Gravity

Para las heuriticas de busqueda local Insert y Replace se deben tomar una lista de clientes a considerar con algun 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 tendra el cliente. Implementé el calculo de COG de la forma que lo describen Vansteenwegen et al. [13]. La coordena COG de una ruta se calcula con las siguietes formulas:

$$x_{cog} = (\sum_{\forall i \in ruta} x_i * B_i) / \sum_{\forall i \in ruta} B_i \tag{4.1}$$

$$y_{cog} = (\sum_{\forall i \in ruta} y_i * B_i) / \sum_{\forall i \in ruta} B_i \tag{4.2}$$

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

Sea r una ruta:

$$r.x_{cog} = \frac{\sum_{\forall i \in ruta} x_i * B_i}{\sum_{\forall i \in ruta} B_i} = \frac{r.x_{cog.num}}{r.x_{cog.den}} \tag{4.3}$$

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

$$r'.x_{cog} = \frac{\sum_{\forall i \in ruta \wedge i \neq j} x_i * B_i}{\sum_{\forall i \in ruta \wedge i \neq j} B_i} = \frac{r.x_{cog.num} - x_j * B_j}{r.x_{cog.den} - B_j} \tag{4.4}$$

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

$$r'.x_{cog} = \frac{(\sum_{\forall i \in ruta} x_i * B_i) + x_k * B_k}{(\sum_{\forall i \in ruta} B_i) + B_k} = \frac{r.x_{cog.num} + x_k * B_k}{r.x_{cog.den} + B_k} \tag{4.5}$$

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).

4.3.2. Swap

El objetivo de esta heuristica 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 vehiculos 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 heuristica 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 metaheuristica 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:

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

4.3.3. Insert

El objetivo de esta heuristica es encontrar una posición en alguna ruta para un cliente no visitado. Básicamente para cada vehiculo 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 vehiculo, se inserta el cliente en tal posicion. El orden en que se toman los clientes no visitados es segun 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 = AnalizeInsert(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 metodo *AnalizeInsert* 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:

$$0(vehiculos * clientesNoVisitados * mediaClientesEnRuta)$$

4.3.4. 2-opt

El algoritmo 2-opt es un simple algoritmo de búsqueda local propuesto por Croes [23]. 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 metodo 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 $0(vehiculos * mediaClientesEnRuta * (mediaClientesEnRuta - 1)/2)$ $0(vehiculos * mediaClientesEnRuta^2/2)$.

4.3.5. 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, empeznndo por los mas cercanos.

```
// Dentro de la clase ReplaceHeuristicas
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.GetCurrentUnvistedDestination;
  var changed = false;
  vehicle.Route.ActivateCog();
  uClients = uClients.OrderBy(x => vehicle.DistanceToCog(x));

  foreach (var client in uClients)
  {
    var res = AnalizeInsert(solution, vehicle, client);
    vehicle.AddDestinationAt(destination, res.BestInsertPosition);
    if (!res.CanBeInserted)
    {
      var 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 *AnalizeInsert*. 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 invalida. 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 efectuo un replace y aumento el beneficio total de la ruta. Replace tiene una complejidad de $O(vehiculos * clientesNoVisitados * mediaClientesEnRuta * 2)$.

4.3.6. Encoder

Agregar heuristicas de busqueda 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 solucion no optimizada por las búsquedas locales. Del mismo modo que una persona no hereda las cirujias esteticas de sus padres.

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

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 el sección del Decodificador (ver sección 4.1.1), los clientes se ordenan de forma ascendente por la propiedad *Key* del objecto *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 menciones en Decoder es que el primer vehiculo por el que empieza es por el de menor *Id* ya que los ordena por su *Id* de forma ascendente. Los vehiculos son indistinguibles al tener el mismo *tMax* en el benchmark de intances de problemas utilizado. De todos modos ahora debo respetar la desición 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 intentara adjudicarselo al primer vehículo que sera el de menor *Id*. Luego tomaré el segundo cliente del mismo vehículo y le asignare 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* ascendentemente. Finalmente, no tendré mas 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 mantenian antes de la optimización. Es decir, dentro de los clientes no visitados, el cliente que previamente tenia el *RandomKey* con menor *Key*, le asigne el *RandomKey* de menor *Key* que habia disponible. Con esto me aseguro que todos los clientes que estaban en el primer vehículo, esten en el primer vehiculo al decodificar el random keys.

TODO Dibujo didactico?

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 primer ruta, como es un decoder goloso, intentará agregar a la primer ruta al resto de los clientes y supongamos que efectivamente encuentra uno. Cuando vi la posibilidad de este escenario, decidi agregar unos delimitadores de modo que si el decoder se encuentra con uno, siempre cambie de vehículo. Estos delimitadores son modelas por la propiedad, *ForceVehicleChangeAfterThis*, con la que extendia 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:

$$ApplyHeuristic(s) = Decoder.Decode(s.RandomKeys, ProblemInfo) \tag{4.6}$$

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. La primera es que el cliente perteneciera al subconjunto de clientes que no estaba visitado. En este caso aumentaria el beneficio total. Pero para que esto suceda, la heuristica de insert no debería haber funcionado correctamente ó no fue la última en aplicarse. En caso de que el cliente perteneciece a otro vehículo, el beneficio total no se modifica. El peor escenario es que este segundo vehiculo se quede con el cliente de un tercer vehiculo, sucediendo un especie de cambios en cadenados de clientes entre una secuencia de vehículos sin decrementar el beneficio total. Luego por mas que tomé la desicion de agregar los delimitadores, considero que ambas opciones eran igual de buenas considerando solamente la función objetivo. Como agregar los delimitadores asegura la valides de la ecuación 4.6.

```
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);
  }
  // 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;
}
```

4.3.7. Secuencia de Aplicación de las Heuristicas

Todas las heuristicas mencionadas, implementan el interfaz:

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

Esto lo hice así de modo de poder inicializar el vector de heuristicas 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 metodo ApplyHeuristic. Así se pueden testear multiples ordenes distintos de búsquedas locales con un simple modificación del parametro de entrada del constructor del BRKGA. El orden en que las heuristicas se aplican es muy relevante. Por ejemplo, si la última heuristica que aplicamos es Swap o 2-Opt, luego la distancias disminuídas de las rutas no sería aprobechado por ninguna cliente.

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

TODO Resultados

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 codigos de HUE:

- **I**: Insert (Cliente no visitado)
- **R**: Replace (Cliente no visitado por uno visitado)
- **0**: 2-Opt (Swap dentro de una misma ruta)
- **S**: Swap (Swap entre dos rutas distintas)

Observaciones de estos resultados:

Claramente el peor orden de heuristicas 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 heuristicas 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 modifico rutas de forma tal que quiza disminuyo la distancia recorrida.

Sobre T_{avg}, notar entre la configuración 2 y 3 el tiempo puede llegar a ser mas del doble y la unica diferencia es que se uso el decodificador simple vs el goloso. Esto lo logra con un i_f practicamente igual. Lo mismo podemos observar entre la configuración 4 y 5. Por lo tanto, habiendo agregado las heuristicas 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 utilice el decodificar simple.

5. RESULTADOS

6. CONCLUCIONES

6.1. Trabajos Futúros

REFERENCES

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

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

[3] H. Bouly, D.-C. Dang, and A. Moukrim. *A memetic algorithm for the team orienteering problem.* 4OR, 8:49–70, 2010.

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

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

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

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

[8] D.-C. Dang, R.N. Guibadj, and A. Moukrim. *A PSO-based memetic algorithm for the team orienteering problem.* In C. Di Chio, A. Brabazon, G. Di Caro, R. Drechsler, M. Farooq, J. Grahl, G. Greenfield, C. Prins, J. Romero, G. Squillero, E. Tarantino, A. Tettamanzi, N. Urquhart, and A. Uyar, editors, Applications of Evolutionary Computation, Lecture Notes in Computer Science, pages 471–480. Springer, Berlin, 2011.

[9] B.L. Golden, L. Levy, and R. Vohra. *The orienteering problem.* Naval Research Logistics, 34:307–318, 1987.

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

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

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

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

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

[15] Bean, J.C. *Genetic algorithms and random keys for sequencing and optimization.* ORSA J. Comput. 6, 154–160 (1994)

[16] Villiam M. Spears , Kenneth A. De Jong *On the virtues of parameterized uniform crossover.* 1991

[17] Ballou, R., Chowdhury, M. *MSVS: An extended computer model for transport mode selection. The Logistics and Transportation.* 1980

[18] S. Boussier, D. Feillet, and M. Gendreau. *An exact algorithm for the team orienteering problem.* 5:211–230, 2007.

[19] W. Souffriau, P. Vansteenwegen, G. Vanden Berghe, and D. van Oudheusden *A greedy randomized adaptive search procedure for the team orienteering problem* Proceedings of the 9th EU/Meeting on Metaheuristics for Logistics and Vehicle Routing, Troyes, France, October 2008.

[20] Jo ao Ferreira, Artur Quintas, José A. Oliveira, Guilherme A.B. Pereira, and Luis Dias *Solving the team orienteering problem* Universidade do Minho, Braga, Portugal, 2012

[21] Manal Zettam, Bouazza Elbenani *A novel Randomized Heuristic for the Team Orienteering Problem* Mohammed V University Rabat, Morocco, 2016

[22] Fuente donde encontré las intancias de problemas de Chao y Tsiligirides `https://www.mech.kuleuven.be/en/cib/op#section-3`

[23] Croes GA *A Method for Solving Travelling-Salesman Problems.* Operations Research, Vol. 6, No. 6, pp. 791-812, (1958).