UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE CIENCIAS

PROBLEMA DE LA MANTA RECTANGULAR: IMPLEMENTACIÓN DE HEURÍSTICAS Y METAHEURÍSTICAS

T E S I S

QUE PARA OBTENER EL TÍTULO DE:

LICENCIADO EN MATEMÁTICAS APLICADAS

P R E S E N T A :

JULIO CÉSAR ROJAS VIGUERAS

TUTORA:

DRA. CLAUDIA ORQUÍDEA LÓPEZ SOTO

CIUDAD UNIVERSITARIA, CIUDAD DE MÉXICO, 2022

1. Datos del alumno
Rojas
Vigueras
Julio César
5536410637
Universidad Nacional Autónoma de México
Facultad de Ciencias
Matemáticas Aplicadas
314126704

2. Datos de la tutora
Dra.
Claudia Orquídea
López
Soto

3. Datos del sinodal 1
Dr.
Javier
Ramírez
Rodríguez

4. Datos del sinodal 2
Dra.
Zaida Estefanía
Alarcón
Bernal

5. Datos del sinodal 3
M. en C.
David Chaffrey
Moreno
Fernández

6. Datos del sinodal 4
Mat.
Ana Lilia
Anaya
Muñoz

7. Datos del trabajo escrito
Problema de la manta rectangular: Implementación
de heurísticas y metaheurísticas
168 p.
2022

A mi abuelo Rogelio ...

Agradecimientos

Agradezco profundamente a mi mamá y a mi papá la ayuda y apoyo que me han brindado en cada paso que he dado en mi vida. Les agradezco porque siempre han estado para mí en todo momento, por escucharme, por sus consejos y palabras de aliento, por su cariño, por su confianza, por creer en mí. Son mis guías, mis referentes y mis héroes. Gracias por todo.

Agradezco a mi hermano porque sé que siempre puedo contar con él. Agradezco su confianza y su cuidado.

Agradezco a mi familia, pues sé que siempre puedo contar con cada uno de ellos, y agradezco a la vida por haberme permitido formar parte de esta familia.

Agradezco a mi tutora Claudia por toda su ayuda y apoyo. Le agradezco todas sus clases y también las sesiones que tuvimos en línea. Agradezco su comprensión, su paciencia conmigo, aprecio todas las revisiones que realizó del presente trabajo y por el tiempo que me dedicó para poder culminarlo. Le agradezco mucho.

Les agradezco a Javier Ramírez, Zaida Estefanía, David Chaffrey y Ana Lilia por haberme apoyado en la revisión y en las correcciones de la tesis.

Agradezco mucho a Chaffrey, que me hizo notar un gran error que tenía en el trabajo escrito y así pudiera corregirlo a tiempo.

Agradezco mucho a mis amigos Leo, Hugo, Héctor, Luis, Rey, Zuno, que en todo momento he contado con ellos. Gracias por su amistad, por sus consejos, por su apoyo y por su tiempo. Les aprecio mucho.

Agradezco a cada profesor que he tenido, pues cada uno ha dejado una enseñanza en mí.

Agradezco mucho a la orientadora vocacional con quien acudí y a Cony, quienes me ayudaron a tomar la mejor elección para mí.

Y agradezco a la UNAM la oportunidad de haber estudiado en Prepa 5 y en la Facultad de Ciencias.

Índice general

Índice de figuras

Capítulo 1

Introducción

En los procesos industriales es importante ahorrar la mayor cantidad de recursos, tiempo y material desperdiciado. Realizar pequeños cambios, como modificar el proceso de producción o implementar uno diferente, puede suponer ahorrar grandes cantidades de dinero y tiempo.

A continuación desarrollaremos tres ejemplos con el fin de dar una idea de lo que vamos a entender por el Problema de la Manta Rectangular. Así, hago una invitación al lector a imaginar una empresa de textiles que intenta abrirse paso en el mundo de la moda. Imagine que dicha empresa se dedica solamente a fabricar pantalones de tres tamaños diferentes, por lo que tienen para ellos tres plantillas definidas. Lo malo es que aún no cuentan con el presupuesto para abrir una tienda por sí mismos, por lo cual han pedido a diversas tiendas que ofrezcan sus pantalones. Una de éstas acepta, pero ha puesto algunas condiciones. Estas son aceptadas por la empresa, y entre esas exigencias se encuentra que la tienda se quedará con un porcentaje de la venta y que deberán enviar en fechas definidas cierta cantidad acordada de cada uno de los tres tamaños de pantalones. Para ello disponen de mantas de tela rectangulares con dimensiones dadas. Es en ellas donde cortarán las plantillas de los pantalones.

Entonces, el problema que enfrentan es poder cubrir la demanda para cada tamaño de pantalones, los cuales serán obtenidos a través del corte de mantas de tela rectangulares, minimizando la cantidad de tela desperdiciada (esta sería aquella que ya no puede usarse para obtener más pantalones). A este tipo de problemas se les conoce como problema de corte de material en dos dimensiones (*two-dimensional cutting stock problem*), y pertenece a la categoría de problemas que se denomina problema de corte y empaquetamiento (*cutting and stock problem*) en los cuales lo que se requiere es dados ciertos objetos y determinada cantidad de material con un tamaño específico, obtener los objetos requeridos a través del corte del material, de tal manera que se iguale o se acerque lo más posible a la cantidad de objetos pedidos a su vez minimizando el material desperdiciado.

Otro posible escenario es considerar objetos pequeños y uno o varios objetos de mayor tamaño, por ejemplo libros y cajas respectivamente, donde el objetivo es buscar la manera de guardar la mayor cantidad de objetos pequeños dentro del grande intentando minimizar el espacio que quede vacío dentro del objeto grande o bien guardar todos los objetos pequeños que tengamos en la menor cantidad de objetos grandes, por lo que se busca minimizar la cantidad necesaria de objetos grandes para una cantidad fija de objetos pequeños.

Retomando la historia que estábamos contando, tal empresa decidió contratar a una persona capacitada para solucionarles su problema, y al cabo de cinco años han logrado crecer y tomar lugar en el mundo de la moda. Millones de personas los siguen, por lo que sus pantalones son

muy solicitados en el país y en consecuencia han logrado elevar enormemente sus ingresos.

Al cabo de un tiempo deciden abrir una tienda muy amplia en uno de los centros comerciales más concurridos del país, pero poco después se dan cuenta que el piso del local es demasiado resbaladizo. La solución que propone uno de los empleados es poner alfombras antideslizantes que eviten que el personal y los clientes resbalen. Esta propuesta es bien recibida, por lo que deciden trazar un plano que indique el área del piso que necesita alfombra y aquella que no, por ejemplo, en donde hubiese una mesa que mostrara los pantalones, una estantería o el lugar ocupado por las cajas registradoras no tendrían que tener alfombra. Así, han realizado un plano a escala con gran detalle y exactitud en el cual queda indicado el área que necesita alfombra y el que no. Se dan cuenta que comprando determinada cantidad de alfombras de metro cuadrado cada una puede funcionar para cubrir todo lo necesario, aunque lo malo en esta idea radica en que es más costoso mandar a hacer tal monto de alfombras con esas dimensiones que menos alfombras pero de mayor tamaño.

El dueño de la empresa decide marcar de nuevo a aquél que contrató alguna vez para resolver este problema. Este último accede y rápidamente se da cuenta del problema que enfrenta, el cual se denomina problema de partición rectangular mínima para polígonos rectilíneos simples (*minimum rectangular partition problem for simple rectilinear polygons*), el cual consiste en dado un polígono rectilíneo simple, particionar su interior con la cantidad mínima de rectángulos posibles. Si estos no se sobrelapan entonces es una partición, de lo contrario es una cubierta ([18]). De esta manera el dueño ha conseguido solucionar el problema que se le presentó y ahora tanto empleados como clientes están felices de ya no resbalarse y en consecuencia los accidentes en la tienda disminuyeron drásticamente.

Ahora, imaginemos el caso de otra empresa. Esta se dedica a enviar tela a otras empresas. La manera de envío es por cajas y mandan un máximo de tres a cada empresa. El proceso de guardado es el siguiente: la tela, cuyas formas son iguales para mantener el contorno de la altura constante, es apilada una sobre otra hasta llegar a los tres metros, pero al verla desde arriba se contempla una forma irregular, es decir, la forma de cada tela es la misma, se acomodan una sobre otra manteniendo la silueta constante, pero vista de arriba hacia abajo se puede distinguir una figura irregular, y esta es cortada en máximo tres partes usando dos grandes cuchillas, las cuales realizan el corte de arriba hacia abajo, una perpendicular a la otra, y finalmente es guardada y enviada a las empresas con quienes tienen contrato. Las cajas no son de un tamaño definido aún, sino que ellos pueden decidir sus dimensiones siempre y cuando la forma de éstas sea rectangular y mantengan la altura establecida de tres metros. Así, el problema consiste en, una vez apilada la tela, encontrar la dimensión que deben tener las cajas que necesitan (máximo tres) de tal manera que se decida la forma de cortar la tela para minimizar el espacio no usado por cada una, pero también minimizando el desperdicio de tela, ya que puede quedar tela sin guardar siempre y cuando suponga disminuir la pérdida de dinero, pues las cajas en que son enviadas no son baratas y cuanto mayor tamaño tengan, mayor será su costo, por lo que puede ser preferible desperdiciar pedazos de tela a encargar cajas de mayor tamaño.

En el anterior ejemplo, podemos observar que el problema tiene tintes de pertenecer al tipo empaquetamiento, pues se pretende maximizar el área usada por las cajas, y al tipo de partición, ya que se pide elegir las dimensiones de las cajas, a excepción de la altura, condicionadas a ser rectangulares. Este problema, aunque tenga características de los dos mencionados, no pertenece a ninguno de ellos, pues el objetivo que debe cumplirse es mandar a hacer un máximo de tres cajas y decidir sus dimensiones (sólo la altura es predeterminada) para guardar en cada una una parte de tela cortada, que deben ser la misma cantidad que las cajas a usar, con el objetivo de

minimizar el desperdicio de la tela, a su vez minimizando el espacio no ocupado de las cajas. A los problemas que sean de este tipo les vamos a llamar Problema de la Manta Rectangular (*Rectangle Blanket Problem (RBP)*), definido así en [9].

1.1. Objetivo

Implementar heurísticas y metaheurísticas que encuentren soluciones al Problema de la Manta Rectangular (*Rectangle Blanket Problem (RBP)*) y determinar cuál de ellas resulta mejor para el problema.

1.2. Estructura del trabajo

A continuación damos una breve explicación del contenido de cada capítulo de la tesis:

- **Capítulo 1.** Corresponde a la introducción.
- **Capítulo 2.** Hablamos de los antecedentes.
- **Capítulo 3.** Explicamos en qué consiste el Problema de la Manta Rectangular, presentamos la literatura relevante y planteamos un modelo matemático para el problema. Comentamos la forma en la cual lo trabajamos, pues las imágenes dos-dimensionales y las mantas que usamos son codificadas en forma matricial por medio de un método raster, y finalizamos comentando tres observaciones importantes respecto a cómo abordamos el problema.
- **Capítulo 4.** Explicamos qué es un algoritmo, las motivaciones de las heurísticas y los pasos para construir un modelo matemático de un problema de optimización combinatoria. Luego, decimos en qué consiste la complejidad temporal y las clases de problemas P, NP, NP-completos y NP-duros. Finalmente, exponemos qué son las heurísticas y metaheurísticas y explicamos qué ingredientes necesitan para poder implementarse adecuadamente.
- **Capítulo 5.** Describimos dos maneras de construir una solución inicial para el Problema de la Manta Rectangular: de manera aleatoria y no aleatoria, y explicamos cómo definimos las vecindades para el problema.
- **Capítulo 6.** Desarrollamos las heurísticas y metaheurísticas que implementamos: Búsqueda Local, Búsqueda Local Iterada, Recocido Simulado, Búsqueda Tabú y Algoritmos Genéticos. De cada una damos un ejemplo de aplicación, consideraciones en cuanto a cómo las implementamos y los valores que dimos a cada parámetro y en qué consiste cada uno de ellos.
- **Capítulo 7.** Comparamos por pares las heurísticas implementadas, aplicando la prueba estadística U de Mann-Whitney a los resultados obtenidos y, con base en lo obtenido, concluimos con cuál de ellas obtuvimos mejores resultados.

- **Capítulo 8.** Se presenta una última implementación que consiste en hacer uso de la heurística que mejores resultados obtuvo y, posteriormente, aplicar un posprocedimiento, el cual tiene por objetivo afinar la búsqueda de una mejor solución en un periodo breve de tiempo. Finalmente presentamos los resultados obtenidos.
- **Capítulo 9.** Corresponde a las conclusiones.
- **Capítulo 10.** Proponemos una aplicación del Problema de la Manta Rectangular.

Capítulo 2

Antecedentes

El área de la Investigación de Operaciones es un área relativamente nueva, pues se considera que nació a inicios de la Segunda Guerra Mundial, cuando ejércitos estadounidenses e ingleses empezaron a apoyarse en el material tecnológico y científico de entonces para tomar decisiones eficaces respecto a la distribución de recursos, estrategias apropiadas que les favorecieran en la guerra, planeación de bombardeos, entre otras acciones [20].

De acuerdo con [16], el primer uso del término *Operations Research* data de 1938 en Gran Bretaña, cuando la cadena de radares (radares antiaéreos) fue lo suficientemente desarrollada como para facilitar los ejercicios de intercepción aérea o *"operational researchers"* a una escala considerable. Además nos comentan que este desarrollo se debió a la implementación de un programa de investigación que buscó mejorar la capacidad de intercepción de la fuerza de defensa aérea Británica. Así, la Investigación de Operaciones nació como una herramienta para apoyar y mejorar las estrategias militares.

Eiselt y Sandblom expresan en [12] que las bases de la Investigación de Operaciones fueron puestas mucho tiempo antes de la Segunda Guerra Mundial, pues nos comentan lo siguiente: F. W. Taylor, a quien se considera padre de la Administración Científica, empezó sus estudios de tiempo en la planta de Midvale. Nos dicen en [22] que, esencialmente, Taylor sugirió que la eficiencia de producción en una tienda o fábrica podría ser mejorada eliminando el movimiento y el tiempo desperdiciado de sus operaciones. Esto formó la base de sus teorías subsecuentes de la administración científica. La pregunta que se planteó y trabajó fue *"¿Cuál es la mejor forma de realizar un trabajo?"*.

Luego, nos hablan en [12] que Henry L. introdujo gráficas de barras para planificar los problemas. Luego, en 1909, Agner Krarup abordó los problemas del tráfico telefónico en su trabajo *"La teoría de las probabilidades y las conversaciones telefónicas"*. Finalmente, en 1913, F. W. Harris desarrolló la cantidad económica de pedido (*economic order quantity*) para el control de inventarios.

Para la siguiente definición nos apoyamos en [28]. La **Investigación de Operaciones** es un área de la matemática aplicada que hace uso del método científico para investigar y estudiar las actividades y operaciones de una organización. Tiene por objetivo la búsqueda y determinación de la mejor solución posible de un problema que consista en tomar una decisión respecto a un proceso operativo, sujeto a retricciones relativas a la limitación o escasez de recursos.

Esta área tiene numerosas aplicaciones en muy diversos campos, entre ellas en la industria al decidir el material a usar dependiendo del costo del mismo, en la forma de organizar una empresa al asignar distintas tareas a los empleados basándose en sus habilidades, en la búsqueda

de una manera eficaz de distribuir una vacuna contra alguna enfermedad peligrosa a diversos hospitales, al buscar un camino que nos lleve de determinado punto a otro de tal manera que la ruta minimice el tiempo de traslado, en la forma de acomodar un inventario de cosas dentro de camiones de tal manera que se renten los menos posibles, en la construcción al cortar material de tal manera que pueda aprovecharse de la mejor forma posible, entre otras aplicaciones.

Uno de estos problemas consiste en la elección de no más de una cantidad definida de rectángulos que aproximen de mejor manera a una imagen dos dimensional [9]. Tales rectángulos no pueden encimarse entre ellos y el objetivo es minimizar la suma del área que quedó sin cubrir de la imagen dos dimensional más el desperdicio o resto de los rectángulos escogidos. A este problema se le conoce como Problema de la Manta Rectangular y es el que trabajaremos en la presente tesis.

El Problema de la Manta Rectangular (*Rectangular Blanket Problem (RBP)*) lo definieron, trabajaron y desarrollaron, en el año 2019, Evrim Demiröz, Kuban Altınel y Lale Akarun en [9]. En él explican este problema y su parecido con el problema de corte y empaquetamiento, con el problema de cubrimiento y el problema de partición, pero muestran que no es igual a ninguno de ellos, pues aunque los objetivos puedan ser parecidos, difieren.

El Problema de la Manta Rectangular, como se establece en [9], consiste en cubrir una imagen dos dimensional, que llamaremos objeto maestro, con una cantidad no mayor a cierto valor predefinido de rectángulos alineados a los ejes tales que no se sobrelapen entre sí, y a este conjunto de rectángulos le llamaremos manta rectangular. El objetivo es minimizar la suma del área no cubierta del objeto maestro y el área de la manta que no cubre nada del mismo, por lo que este problema no es ni de cubrimiento, pues puede haber partes de la imagen dos dimensional que no sean cubiertas, ni de partición, pues la manta no está restringida a estar contenida en el objeto maestro.

En el mismo artículo nos explican que resolvieron el problema usando un algoritmo de ramificación y precio (*branch-and-price*), el cual es un híbrido entre ramificación y acotamiento (*branch-and-bound*) y generación de columnas (*column generation*).

Nos comentan que para instancias más grandes el algoritmo que aplicaron resulta ser muy demandante computacionalmente, por lo que propusieron e implementaron tres heurísticas para encontrar buenas soluciones: Divide y Ajusta (*Split and Fit*), *Fast Adaptive Silhouette Area based Template Matching (FAST)* y Recocido Simulado Restringido (*Constrained Simulated Annealing (CSA)*).

Evrim Demiröz en [8] implementó una novedosa estrategia de ramificación y lo probó con el Problema de la Manta Rectangular. Al comparar resultados con el algoritmo *branch-and-price* del artículo [9] resulta dar buenos resultados, aunque no siempre superando a este algoritmo, y Evrim comenta que probablemente esto se deba a que la estrategia se usa en un esquema de ramificación y precio con generación de columnas y hay múltiples soluciones óptimas.

Capítulo 3

Problema de la Manta Rectangular: modelación del problema

3.1. Problema de la Manta Rectangular (RBP)

El Problema de la Manta Rectangular entra en la categoría de optimización combinatoria. En [9] nos comentan que consiste en definir dos conjuntos de elementos: Uno a quien llamaremos objeto grande u objeto maestro (imagen a cubrir) y otro conjunto de objetos pequeños (rectángulos no idénticos). Luego, el problema radica en elegir de manera adecuada una cantidad no mayor a cierto número de elementos del conjunto de objetos pequeños y agruparlos en un nuevo subconjunto (manta rectangular). Una vez hecho, lo asociamos al objeto maestro y vemos si se cumplen una serie de condiciones geométricas, tal como que ningún rectángulo quede encimado, estén alineados a los ejes y que no necesariamente cubran toda la imagen (objeto grande).

Evrim et al., en [9], definen al *Problema de la Manta Rectangular* como determinar un conjunto de rectángulos que conste de no más de k rectángulos, los cuales deben ser alineados a los ejes y no deben sobrelaparse, tales que al encimar o sobreponer en una imagen predefinida de dos dimensiones (imagen la cual queremos cubrir) minimice el área de la imagen que no está siendo cubierta por el conjunto de rectángulos y al mismo tiempo minimice el área de los rectángulos que no cubren nada de área de la imagen. De esta manera, el conjunto de máximo k rectángulos tiene como intención representar de la manera más aproximada, o si se puede igual, la forma de una imagen de dos dimensiones. A este conjunto de rectángulos que se obtenga le llamaremos **manta o cobija rectangular**, y su uso es una estrategia para realizar con mayor rapidez operaciones computacionales, pues como veremos podrá ser respresentada por una matriz cuyas entradas son unos y ceros.

El objetivo es optimizar una función objetivo que consiste en maximizar el área cubierta de la imagen por la manta rectangular y minimizar el área de los rectángulos pequeños que no cubren la imagen ([9]). Así, la cobija se vuelve más precisa a medida que el área no encimada por la cobija sobre la imagen decrece y al mismo tiempo, el área encimada por la cobija sobre la imagen crece y con esto la cobija pretende reemplazar un objeto irregular (imagen a cubrir) aproximándolo con un conjunto de rectángulos alineados con los ejes.

Un rectángulo está alineado a los ejes si sus aristas adyacentes (que deben de ser ortogonales) son paralelas a los ejes x y y.

Dos rectángulos se sobreponen o enciman cuando comparten una porción de área en común.

Una observación importante es que el valor elegido para k debe ser un número natural ya que es la cantidad máxima de rectángulos que se pueden usar para construir la manta rectangular.

La cantidad de rectángulos afecta directamente la calidad de la aproximación pues a mayor cantidad de rectángulos posibles a elegir la aproximación se volverá más fina. Puede pensarse lo anterior como la cantidad de pixeles que puede tener una fotografía, pues si ésta consta de pocos pixeles la imagen no será representada adecuadamente, pues realizará una aproximación no muy buena, mientras que una que disponga de una enorme cantidad de rectángulos podrá representar de mejor manera la imagen. Por tanto cuando el valor de k tiende a infinito la aproximación a la imagen se vuelve perfecta debido a que la manta dispondrá de rectángulos suficientes para igualar la imagen.

Antes de continuar me gustaría plantear de otra manera el objetivo del Problema de la Manta Rectangular. El objetivo es determinar un conjunto de rectángulos, denominado manta rectangular, que contenga no más de k rectángulos alineados a los ejes y que no estén encimados, tal que la manta generada maximice el área cubierta de la imagen que se quiere aproximar y al mismo tiempo se minimice el área de la manta que no logra cubrir ninguna porción de la imagen (excedente de la manta).

En el problema que planteamos el objeto maestro consiste en la imagen que queremos cubrir con una manta formada por rectángulos, los cuales son los objetos que podemos manipular.

La manta construida puede pasar las dimensiones del objeto a cubrir (es decir, la manta no está restringida a permanecer en el interior de la imagen) con la condición de que ningún rectángulo encime parte de otro rectángulo.

Evrim et al., en [9], proponen una forma de encontrar la solución exacta, pero, como comentan, es muy costosa computacionalmente, por lo que después implementaron heurísticas (más adelante, en otra sección, explicaremos en qué consisten) para reducir los costos computacionales. Las heurísticas que implementaron fueron *Split and Fit (SF)*, *Fast Adaptative Silhouette Area Based Template Matching (FAST)* y *Constrained Simulated Annealing (CSA)*.

De igual manera, implementaremos algunas heurísticas para poder encontrar buenas soluciones al problema. Una vez obtenidas las compararemos para ver qué heurística encuentra en general mejores soluciones con la esperanza de acercarnos o incluso ser la solución óptima.

3.2. Modelo del problema

3.2.1. Elementos para construirlo

Primero se debe determinar la imagen en dos dimensiones que se desea trabajar y la cantidad máxima k de rectángulos a usar. Una vez hecho esto, procedemos a definir los materiales que usaremos para la construcción del modelo.

Una *Manta* (notación: M) es la unión (o un conjunto) de una cantidad máxima de k rectángulos con las restricciones de que no se traslapen y cada uno esté alineado a los ejes.

Para poder operar tanto con la imagen como con la Manta de una manera más flexible utilizaremos el método raster que explicaremos a continuación.

3.2.1.1. Método raster

El método raster que utilizaremos fue propuesto en [24]. Consiste en dividir la superficie de una imagen en regiones discretas ordenadas para representarla como una cuadrícula/malla codificada en la forma de una matriz.

Los pasos a seguir del método son:

1. Elige la figura con la cual quieras trabajar. Debe ser de dos dimensiones y debe tenerse claro el área que queremos cubrir con una manta M.
2. Realizamos una cuadrícula sobre la imagen. Esta cuadrícula debe cumplir que cada cuadrado sea del mismo tamaño que los demás.
3. A los cuadrados que tengan una parte del área de la figura les asignamos un 1; a los cuadrados vacíos (aquellos que no contengan área de la figura) les asignamos un 0.
4. Le asociamos una matriz a la cuadrícula construida con 1's y 0's. Si la cuadrícula es de $n \times m$ cuadrados, la dimensión de la matriz será $n \times m$, donde cada entrada tendrá el valor correspondiente en la cuadrícula. Esta matriz se construye de manera ordenada, pues representará a la imagen a cubrir.

Así, aplicamos este método en la imagen que queremos cubrir para flexibilizar su uso. La matriz que se obtiene como resultado de este procedimiento de codificación y representación la denominaremos *Imagen Objetivo* y será denotada como IO. Con esto obtenemos una representación más flexible de la imagen, pues llevamos el problema al terreno de las matrices, con las cuales ya sabemos operar y será más sencillo trabajar de esta manera. En consecuencia, ahora el problema es encontrar codificaciones matriciales para un máximo de k rectángulos, y la suma de estas matrices, cada una representante de un rectángulo, será la representación matricial de la manta rectangular. Por tanto, en lugar de trabajar con una cantidad infinita de rectángulos y conjuntos infinitos de ellos, el nuevo espacio de soluciones será un espacio finito y la unión de conjuntos es representada como una suma matricial.

En la Figura 3.1 se muestra el proceso de transformación con el método raster de la imagen azul en la Imagen Objetivo.

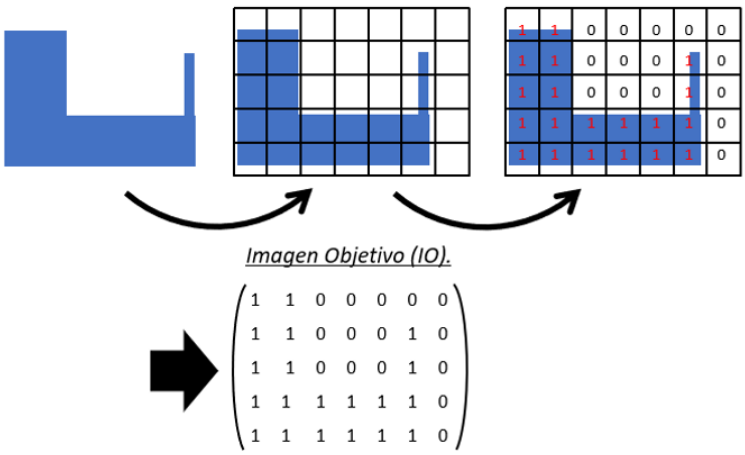


Figura 3.1: Ejemplo matriz IO asociada a una figura

Aunque este método es bastante útil, tiene una desventaja pues es incapaz de representar adecuadamente formas irregulares. Una manera de aminorar este problema es realizando cuadriculados más "finos", lo cual es un mallado de la imagen tal que obtengamos una mayor cantidad de cuadrados de menor área, con lo cual obtendremos matrices de mayor dimensión.

3.2.2. Modelo matemático del problema

Definición: Un rectángulo R lo definimos como una matriz $M_{(r_1,r_2),(r_3,r_4)}$ de dimensión $n \times m$ tal que cumple lo siguiente:
$r_1, r_2, r_3, r_4 \in \mathbb{N}$;
$1 \leq r_1 \leq r_3 \leq n$;
$1 \leq r_2 \leq r_4 \leq m$;
donde

$$(M_{(r_1,r_2),(r_3,r_4)})_{i,j} = \begin{cases} 1 & si \;\; r_1 \leq i \leq r_3 \wedge \; r_2 \leq j \leq r_4 \quad i,j \in \mathbb{N} \\ \\ 0 & \qquad\qquad\qquad en\ otro\ caso \end{cases}$$

En esta definición, (r_1, r_2) es la coordenada de la esquina superior izquierda del rectángulo y (r_3, r_4) es la coordenada de su esquina inferior derecha.

Para visualizar la anterior definición, en la Figura 3.2 se muestra la matriz del rectángulo R.

$$R_{n\times m} = M_{(r_1,r_2),(r_3,r_4)} = \begin{array}{c} \\ \\ \\ \\ r_1 \\ \\ \\ \\ r_3 \\ \\ \\ \end{array} \begin{array}{c} \begin{array}{cccccccccccccc} & & & & & r_2 & & & & & r_4 & & & \\ \end{array} \\ \begin{pmatrix} 0 & 0 & \dots & 0 & 0 & 0 & \dots & 0 & 0 & 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & \dots & 0 & 0 & 0 & \dots & 0 & 0 & 0 & 0 & \dots & 0 & 0 \\ \vdots & \vdots & & \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots & \vdots & & \vdots & \vdots \\ 0 & 0 & \dots & 0 & 0 & 0 & \dots & 0 & 0 & 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & \dots & 0 & 1 & 1 & \dots & 1 & 1 & 1 & 0 & \dots & 0 & 0 \\ 0 & 0 & \dots & 0 & 1 & 1 & \dots & 1 & 1 & 1 & 0 & \dots & 0 & 0 \\ \vdots & \vdots & & \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots & \vdots & & \vdots & \vdots \\ 0 & 0 & \dots & 0 & 1 & 1 & \dots & 1 & 1 & 1 & 0 & \dots & 0 & 0 \\ 0 & 0 & \dots & 0 & 1 & 1 & \dots & 1 & 1 & 1 & 0 & \dots & 0 & 0 \\ 0 & 0 & \dots & 0 & 0 & 0 & \dots & 0 & 0 & 0 & 0 & \dots & 0 & 0 \\ \vdots & \vdots & & \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots & \vdots & & \vdots & \vdots \\ 0 & 0 & \dots & 0 & 0 & 0 & \dots & 0 & 0 & 0 & 0 & \dots & 0 & 0 \end{pmatrix} \end{array}$$

Figura 3.2: Visualización de la matriz de un rectángulo $R_{n\times m}$

En la siguiente imagen se ejemplifica la construcción de un rectángulo. El rectángulo a construir es $M_{(r_1=2,r_2=3),(r_3=4,r_4=6)}$ de dimensión 5×7.

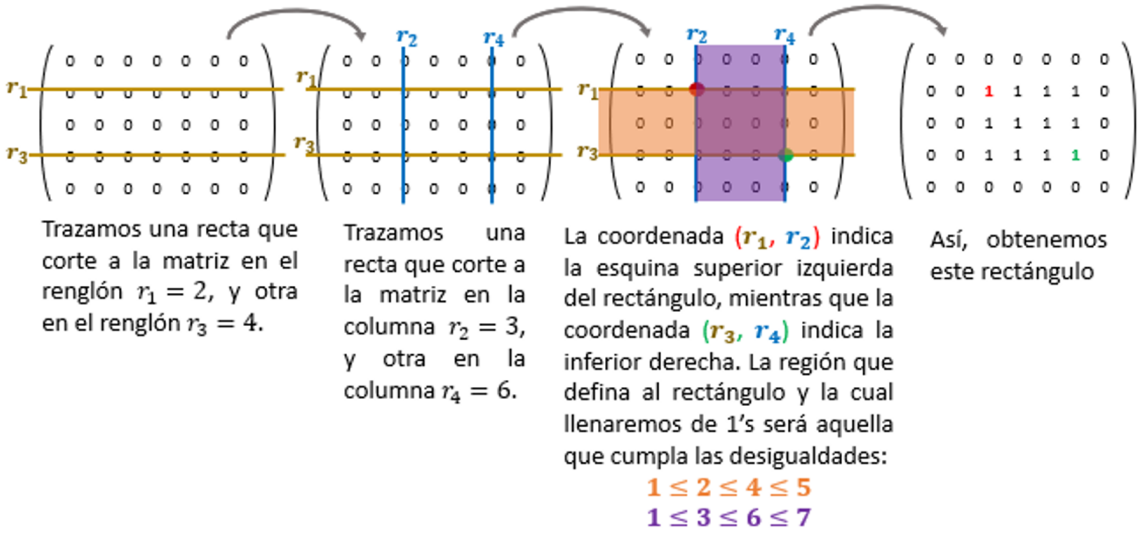


Figura 3.3: Visualización de la matriz de un rectángulo $R_{n\times m}$

El modelo matemático usado en este trabajo considera la imagen objetivo como:

$IO \in M_{n\times m}(\{0,1\})$ fija.
$k \in \mathbb{N}$ fija.
$R_i \in M_{n\times m}(\{0,1\})\ \forall i = 1,2,...,p,\ \ p \leq k,\ \ i \in \mathbb{N}.$
$M := \sum_{i=1}^{k} R_i$
donde: IO es la representación matricial de la imagen a cubrir al aplicar el método raster, R_i un rectángulo para toda i definida y M la manta rectangular.

$$min\ z = \sum_{i=1}^{n}\sum_{j=1}^{m}(|M - IO|)_{i,j}$$

$$s.a.$$

$$\sum_{i=1}^{n}\sum_{j=1}^{m}(|R_r + R_t|)_{i,j} = \sum_{i=1}^{n}\sum_{j=1}^{m}(|R_r - R_t|)_{i,j}\ \ \forall\ r,t = 1,2,...,p\ \ ,r \neq t$$

La función objetivo z consiste en la suma de todas las entradas del valor absoluto de la resta de matrices $M - IO$. Esto es, suma las unidades de la imagen que no están siendo cubiertas por la manta y también las unidades de la manta que no cubren nada de la imagen. Este valor es el que se desea minimizar, pues cuanto menor sea mejor será la aproximación de la manta a la imagen.

El conjunto de restricciones verifican que ningún rectángulo encime a algún otro. Se aplica a todos los posibles pares de rectángulos distintos usados. El lado izquierdo de la igualdad lo que hace es sumar la representación matricial de los rectángulos r y t, aplicar valor absoluto

a cada entrada de la matriz resultante y luego sumar todas las entradas. El lado derecho de la igualdad consiste en restar las matrices de los rectángulos r y t, aplicar valor absoluto a todas las entradas de la matriz resultante y posteriormente sumar todas las entradas. Si dos rectángulos se encimaran, la suma de ambas matrices provocará que haya entradas iguales a dos, mientras que la resta provocará que en esas mismas entradas el valor sea cero. Así, la igualdad no se cumplirá, pues obtendremos en el lado izquierdo un valor mayor que el obtenido en el lado derecho. En el caso de que los rectángulos no se sobrelapen, al sumar las matrices conseguiremos una matriz con entradas iguales a cero o uno, mientras que en la resta tendremos entradas iguales a cero, uno o menos uno, pero al aplicarles valor absoluto, en ambas matrices tendremos la misma cantidad de ceros y de unos, por lo que al sumar todas las entradas se cumplirá la igualdad.

Observación: En total tendremos $\frac{p(p-1)}{2}$, $p \leq k$ restricciones, donde p es la cantidad de rectángulos usados y k la cantidad máxima de rectángulos que pueden construirse. Para obtener este valor, podemos realizar un grafo simple (no dirigido[1] y sin lazos[2]) con k vértices. Asociamos a cada rectángulo empleado un vértice, por lo que tendremos p vértices asociados. Como por cada par de rectángulos distintos existe una restricción, en el grafo lo representaremos uniendo los vértices para los cuales deba verificarse la restriccion. En consecuencia, obtendremos un subgrafo[3] completo[4] de p vértices.

En Teoría de Gráficas tenemos el siguiente resultado:

Teorema. Sea K_n una gráfica completa con n vértices. En total, K_n tiene $\frac{n(n-1)}{2}$ aristas.

Así, usando este resultado en nuestro subgrafo completo obtenemos que tiene $\frac{p(p-1)}{2}$ aristas, y como existe una restricción por cada arista, se traduce finalmente en que habrá que verificar $\frac{p(p-1)}{2}$ restricciones.

Algo importante que mencionar en el cálculo anterior es que el grafo tiene k vértices y $p \leq k$. Si $p = k$ implica que en total se construyeron k rectángulos, mientras que $p < k$ implica que $k - p > 0$ rectángulos no se construyeron aunque hubiera la posibilidad de hacerlo. En el grafo mencionado, éstos $k - p$ vértices serán de grado cero[5].

La matriz resultante de la operación $M - IO$, podrá tener como entradas posibles los siguientes valores:

1) -1, indica una unidad de IO que no está siendo cubierta por la manta M.

2) 0, indica que no hay ninguna unidad de M ni de IO o bien, los espacios en los que M logra cubrir a IO.

3) 1, indica una unidad de M que no cubre ninguna de IO.

---

[1] Un grafo es **no dirigido** si sus aristas no tienen dirección, es decir, si para todo vértice a y b del grafo para los que exista una trayectoria $T_1 = a, x_1, ..., x_{n-1}, b$, entonces también existe la trayectoria $T_2 = b, x_{n-1}, ..., x_1, a$.

[2] Un **lazo** es una arista que une un vértice consigo mismo.

[3] Sea G un grafo tal que A es su conjunto de aristas y V su conjunto de vértices. H es **subgrafo** de G si $E \subseteq A$ y $B \subseteq V$, donde E es el conjunto de aristas de H y B el conjunto de vértices de H.

[4] Una gráfica simple es **completa** si existe una arista entre cada par de vértices.

[5] El **grado** de un vértice v (denotado $\delta(v)$) indica la cantidad de aristas adyacentes a él.
Un vértice v es de **grado cero** si no tiene aristas adyacentes, es decir, si $\delta(v) = 0$.

Ejemplo:

Supongamos tenemos la matriz IO y se propone la manta M:

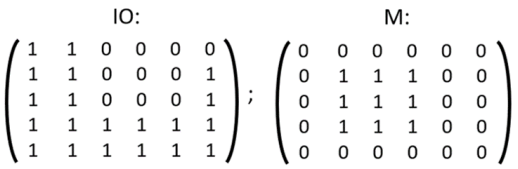


Figura 3.4: Matrices IO y M dadas para el ejemplo

Al realizar la resta $M - IO$ obtenemos lo siguiente:

Figura 3.5: Ejemplo $M - IO$ con explicación de matriz resultante

3.2.3. Tres observaciones muy importantes de las soluciones que encontraremos para el problema en este trabajo

- Debemos observar que las restricciones permiten la construcción de mantas inconexas, pues lo único que prohíben es el traslape entre rectángulos. Esto es importante, pues habrá Imágenes Objetivo para las cuales sea más apropiada la utilización de una manta inconexa en lugar de una conexa.

- Sea $IO_{n\times m}$ la representación matricial de la imagen en dos dimensiones que se desea trabajar. Los posibles valores que k puede tomar, o de otra forma, la cantidad de rectángulos que pueden usarse en la construcción de una manta para trabajar el problema es:

$$1 \leq k \leq n * m, \ donde \ IO_{n\times m} \ ^{6}$$

- El valor de k lo fijamos en el inicio, es decir, la cantidad de rectángulos que toda manta factible debe tener es igual a k. Una vez obtenida la manta solución quitamos todos los rectángulos que empeoren el valor de la misma, pues algunos pueden o no estar cubriendo ninguna unidad de IO o el residuo del rectángulo resulta mayor que las unidades cubiertas.

[6]La cantidad máxima de rectángulos que podemos usar es $n * m$, lo cual sería un rectángulo por cada entrada de la *Imagen Objetivo*.

Capítulo 4

Simplificando la realidad

4.1. Algoritmos

Al momento de enfrentarnos con un problema, podemos recurrir a la computadora como una herramienta que nos puede ayudar a encontrar una posible solución a un determinado problema.

Las computadoras son capaces de realizar una enorme cantidad de cálculos en muy poco tiempo, por lo que son de suma importancia en la actualidad. Claramente no son entes que piensan y toman decisiones por sí solas, sino que somos nostros quienes debemos darles instrucciones para que logren un objetivo. Las intrucciones deben ser en un lenguaje que ellas puedan interpretar y deben ser lo más claras posibles, diciendo qué queremos que hagan ante todos los escenarios que puedan suceder. Así, lo que tenemos que realizar es una especie de receta, donde se indique paso por paso lo que debe hacerse ante cada situación, y en computación, a estas recetas les damos el nombre de algoritmo.

Los algoritmos son procedimientos que siguen una serie de instrucciones de forma ordenada con el fin de obtener un resultado adecuado para un problema determinado. Su importancia radica en que podemos programarlos para que las máquinas puedan leerlos y realizar las instrucciones de manera eficaz y rápida.

Hay problemas para los cuales ya se han encontrado algoritmos que encuentren la solución óptima y hay otros para los que aún no se ha diseñado uno. Para estos últimos, tenemos dos posibilidades: Intentar realizar un algoritmo que nos de la solución exacta o realizar uno que nos de una buena solución aunque quizá no óptima.

En general lo que queremos es obtener la mejor solución, pero hay veces en las que el problema a resolver o resulta muy complicado o resulta muy caro computacionalmente, en el sentido que la máquina tardará mucho en encontrar la solución óptima puesto que necesita realizar una enorme cantidad de cálculos, por lo cual optamos por acercarnos a la mejor solución, pero ¿cómo sabemos cuándo la mejor solución es muy difícil de encontrar o implica un proceso computacional demasiado costoso?. Esta pregunta será respondida en secciones posteriores.

4.2. ¿Cuándo dejar de buscar el óptimo?

Iniciemos con un ejemplo: El Problema del Agente Viajero (TSP). [1]

Imaginemos a un turista en la Ciudad de México. Dispone de poco tiempo para recorrer la ciudad y tiene en mente visitar solamente cinco destinos turísticos: la Plaza de la Constitución, Bellas Artes, Ciudad Universitaria, el centro de Tlalpan y el centro de Coyoacán. Por tanto, el objetivo del turista es visitar los cinco destinos turísticos mencionados en el menor tiempo posible para que así, si le queda tiempo, pueda visitar algún parque de la ciudad, pero además quiere empezar y terminar en un mismo destino turístico, pues es su deseo comer en el mismo lugar donde empezará su travesía.

Entonces, a partir de los datos dados construimos la siguiente gráfica donde el tiempo (o peso) que hace entre cada par de puntos turísticos está dado en minutos:

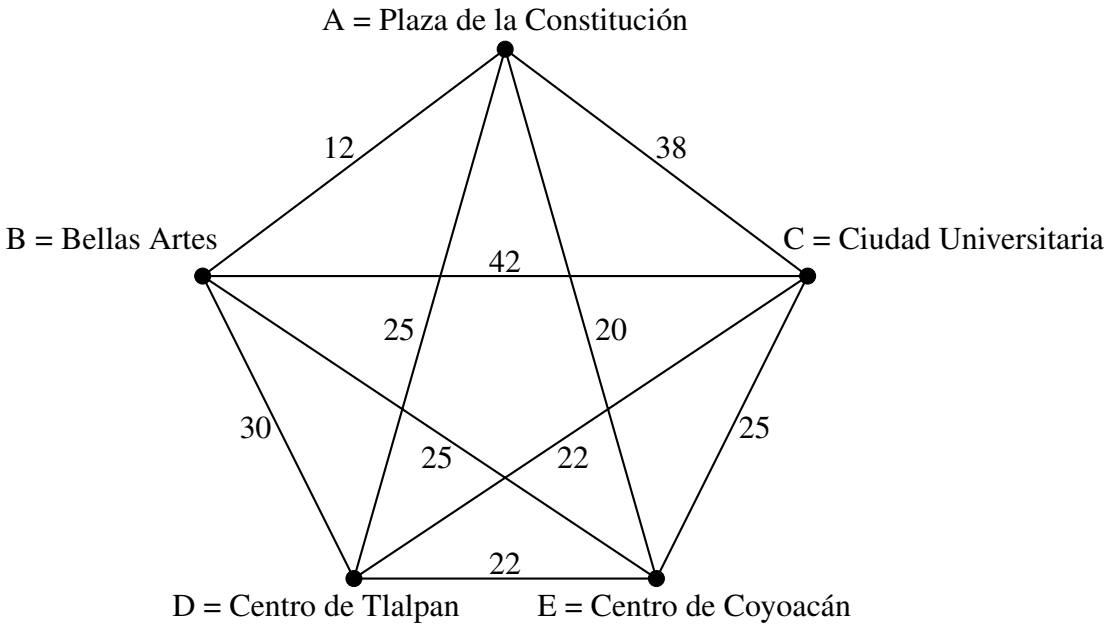


Figura 4.1: Grafo del ejemplo del Problema del Agente Viajero

Una forma de encontrar la mejor solución es, digamos, haciendo uso de la fuerza bruta, lo cual sería hacer una lista de todos los posibles ciclos que puede realizar, calcular el tiempo que le toma recorrer cada uno, compararlos y elegir el de menor tiempo posible.

Para este ejemplo tendríamos que hacer una lista de 24 caminos (aunque si nos damos cuenta a tiempo se reduce a 12, pues, por ejemplo, el tiempo que toma recorrer el camino $A-B-C-A$ es igual al que toma $A-C-B-A$), lo cual podría ser una buena opción, pero conforme vamos añadiendo destinos turísticos a visitar, esta lista va creciendo de forma alarmante, al punto que si tuviéramos 12 destinos para visitar ¡tendríamos que hacer una lista de más de diecinueve millones de posibles caminos! (en total 19958400 caminos posibles, donde ya quitamos las posibilidades del tipo, por ejemplo, $A-B-C-A$ igual a $A-C-B-A$). Así,

[1] El problema consiste en encontrar el ciclo hamiltoniano de peso mínimo. Un ciclo hamiltoniano es un ciclo que pasa por todos los vértices de un grafo sin repetir a excepción del primero, es decir, inicia y termina en un mismo vértice pasando por todos los demás vértices del grafo una única vez.

podemos ver que el uso de la fuerza bruta, método conocido como **Búsqueda exhaustiva**, puede ser útil cuando tenemos pocos destinos, pero de nada servirá intentarlo para muchos. Es por esto que recurrimos a métodos para que nos den el mejor camino posible, o al menos un buen camino, sin que tengamos que hacer esta enorme lista.

4.3. La necesidad de conocer una respuesta

Cuando nos enfrentamos a un problema de optimización es posible que se trate de uno de los siguientes casos: no ha sido posible diseñar un algoritmo que nos dé la mejor solución, no hemos podido encontrar un algoritmo que nos dé el óptimo o si encontramos o diseñamos uno que lo haga, tarda tanto tiempo que cuando sepamos la respuesta ya no será útil. Estas razones nos motivan ya no a saber la mejor respuesta, sino a conocer al menos una respuesta.

Muy probablemente sea fácil dar soluciones a distintos problemas de optimización. Consideremos el ejemplo anterior, cuyo objetivo es encontrar el ciclo de menor tiempo. Una manera de hacerlo sería comenzar en la ciudad A y recorrer los vértices en sentido de las manecillas de un reloj. Este ciclo toma un total de 127 minutos. Muy probablemente no sea la mejor solución, y quizá no sea buena, pero es una solución de tantas que hay y es mejor tener una a no tener ninguna. Como puede notar logramos encontrar una solución en muy poco tiempo.

Otra forma de generar un ciclo es: Iniciamos en un punto aleatorio. Vemos el peso de las aristas que salen de donde estamos y nos movemos por la arista de menor peso. A donde lleguemos, nos movemos por la arista de menor peso que salga de donde estemos pero descartando lugar y arista por donde pasamos anteriormente. Repetimos este proceso de manera iterativa hasta llegar al punto inicial. Aplicando este procedimiento al ejemplo anterior y tomando como vértice aleatorio la ciudad B, obtenemos un ciclo de 118 minutos.

Quizá tardamos más tiempo en dar el camino de 118 minutos que el de 127, pero resultó ser mejor camino, pues tardaremos menos tiempo en recorrerlo. Aún así, es peor que el mejor ciclo cuyo tiempo es de 109 minutos pero tardamos más en encontrar este último. Así, quizá sea mejor sacrificar la mejor solución por mejorar el tiempo de búsqueda de una buena solución.

Estas son las motivaciones y objetivos de las heurísticas, encontrar una buena solución en un periodo de tiempo accesible.

Ahora supongamos que sabemos que se realizó una manifestación en el camino que va de Ciudad Universitaria al Centro de Coyoacán, por lo cual este camino resulta inaccesible. Además, supongamos que el camino de Bellas Artes a Plaza de la Constitución se encuentra cerrado debido a que será remodelado, por lo cual será imposible pasar por él. Así, todos los ciclos que no incluyan a las aristas $\overline{CE}$ y $\overline{AB}$ serán **soluciones factibles**, pues harán uso de caminos accesibles, mientras que aquellos que no cumplan lo mencionado no serán soluciones del problema. Finalmente, a estas prohibiciones se les conoce como **restricciones**. Por tanto, el problema se resume en encontrar el ciclo que pase por todos los lugares sin repetir en el menor tiempo posible, restringido a que no puede utilizar los caminos $\overline{CE}$ y $\overline{AB}$.

La intersección de las restricciones de todo modelo de optimización genera al espacio de soluciones factibles, que como comentamos son todas aquellas soluciones que satisfacen todas las restricciones. Sin embargo, estamos en busca de la mejor. Para ello es necesario proponer una solución inicial e ir moviéndonos por el espacio de soluciones hacia mejores soluciones. Dicho movimiento es entendido como realizar un cambio a una solución para obtener una solución diferente. Esto nos lleva a un nuevo concepto: vecindad.

4.4. Vecindades

Una vez se tenga bien definido el espacio de soluciones, veremos que hay, normalmente, una infinidad de soluciones posibles. Para llegar de una a otra solución puede tomar poco o mucho tiempo, dependiendo de qué tan alejadas estén y qué rutas puedas elegir para llegar de una a otra, pero ¿qué es lo que influye en esto?.

Supongamos elegimos aleatoriamente una de las soluciones. Partiendo de ella, podemos ver que tiene muchas soluciones a su alrededor, unas que se encuentran cerca y otras que se encuentran lejos, y es aquí donde surge naturalmente una pregunta: ¿cómo sabemos qué soluciones se encuentran cerca de la actual y cuáles más lejos?.

Decimos que una solución es cercana a la actual cuando ambas tienen muchos elementos en común, y por lo tanto una solución es lejana cuando tienen pocos elementos en común.

Ahora bien, los elementos que tienen en común varía de problema en problema, pero además interviene un elemento muy importante: las vecindades. La vecindad de una solución contiene a todas las soluciones que se encuentran cerca, tanto que puedes llegar a ellas en "un paso". Así, tenemos que definir qué elemento o elementos podemos cambiar a nuestra solución y cuánto podemos cambiarlos para poder definir bien una vecindad. Con esto podremos formular una definición:

Sea x una solución del problema en el que se está trabajando. Una **vecindad** de una solución x son las soluciones cercanas a x a las cuales se puede llegar realizando un movimiento. De otra forma, son las soluciones a las cuales puedes llegar realizando un cambio o una modificación pequeña a x. A la vecindad de x se le denota con $N(x)$.

Estas son de suma importancia, pues son las que nos permiten movernos en el espacio de soluciones, y los algoritmos que queremos son aquellos que se mueven en el espacio de soluciones de manera inteligente, siguiendo un procedimiento definido que nos permita acercarnos en cada paso a la solución óptima.

4.5. Factibilidad

El espacio de todas las soluciones de un problema de optimización está delimitado por las restricciones. Al conjunto de puntos que cumplen las restricciones le denominamos **región factible** y cada punto dentro de esta región es una **solución factible** del problema considerado. Los puntos que no cumplan al menos una restricción se encontrarán fuera de la región factible y por ende no son solución del problema. Veamos un ejemplo:

$$\begin{aligned} \text{función objetivo:} \quad & Max \;\; x+2y \\ \text{sujeto a:} \quad & \\ & x+3y \leq 8 \\ & 0 \leq x \leq 5 \\ & y \geq 0 \\ & x,y \in \mathbb{N} \cup \{0\} \end{aligned}$$

Supongamos tenemos los siguentes dos puntos:

1. $x = 5$ y $y = 2$
2. $x = 2$ y $y = 2$

Al evaluarlos en la función objetivo, con el primer punto obtenemos el valor 9 y con el segundo 6. A primera vista, como el objetivo es maximizar la función objetivo, podríamos decir que el primer punto es mejor que el segundo, pero aún no podemos asegurar esta aseveración, pues falta comprobar si se encuentran dentro de la región factible.

Claramente ambas cumplen la segunda y la tercera restricción, pero la primera sólo se satisface por la segunda opción, pues sustituyendo obtenemos:

1. con la primera opción: $(5) + 3(2) = 11 \nleq 8$
2. con la segunda opción: $(2) + 3(2) = 8 \leq 8$

De esta manera comprobamos que la primera opción no es una solución factible, mientras que la segunda sí lo es.

Ahora, supongamos que definimos las vecindades de la siguiente manera:

Sea $w = (x, y)$ cualquier punto en el espacio de búsqueda [2]. La vecindad de w es:
$$N(w) = \{(x+1, y), (x-1, y), (x, y+1), (x, y-1)\}.$$

Ahora, obtengamos las vecindades de los siguientes dos puntos:
$N(5,2) = \{(6,2), (4,2), (5,3), (5,1)\}$ y $N(2,2) = \{(3,2), (1,2), (2,3), (2,1)\}$

Si queremos quedarnos únicamente con las soluciones factibles, descartamos para la primera vecindad las opciones (6,2), (4,2) y (5,3), y para la segunda vecindad descartamos (3,2) y (2,3), pues no cumplen con alguna de las restricciones del problema.

Evaluando en la función objetivo aquellas que cumplen todas las restricciones, obtenemos:

1. Valor de las soluciones factibles de $N(5,2)$:
   - $(5) + 2(1) = 7$
2. Valor de las soluciones factibles de $N(2,2)$:
   - $(1) + 2(2) = 5$
   - $(2) + 2(1) = 4$

Podemos ver que la mejor solución entre estas soluciones factibles es $(x, y) = (5, 1)$, que da un valor de 7, y de hecho puede demostrarse que ésta es la solución óptima para éste problema.

Estos ejemplos nos sirven para remarcar dos cosas muy importantes: la primera es que **las vecindades están definidas tanto para soluciones factibles como para puntos que no son solución factible**.

Segundo, es algo que puede suceder en las heurísticas (que hablaremos de ellas a continuación). Observemos que llegamos a la solución óptima por medio de una vecindad generada por un punto que no se encuentra en la región factible mientras que con la vecindad de la solución factible no llegamos al óptimo. Esto es importante de mencionar, pues nos muestra que en algunas ocasiones si queremos acercarnos a la solución óptima **no es necesario movernos siempre dentro de la región factible del problema, pues a veces podremos llegar más rápido al óptimo pasando por un punto que se encuentre fuera de esta región**. Esto en algunos procedimientos funciona y en otros no, todo depende del problema que se esté resolviendo, de cómo se hayan definido las vecindades y del método a emplear.

[2] El espacio de búsqueda es el espacio donde se encuentra tanto la región factible como todos los puntos que no son solución.

4.6. La dificultad en los problemas

4.6.1. Complejidad temporal

De manera informal, la complejidad temporal de un algoritmo es la cantidad de pasos que le toma a una máquina ejecutarlo. Este nos sirve de apoyo, junto con otras medidas como la complejidad espacial[3], para comparar la eficiencia entre algoritmos y es un concepto dentro del análisis de algoritmos.

Definimos un **paso** como la ejecución de una sentencia y definimos el **conteo de pasos** como el número de veces que se ejecuta cada sentencia o declaración del algoritmo, el cual está en función del tamaño de la instancia de entrada. El tamaño de la entrada n varía de problema en problema, pues para algunos podría ser que n sea el número de elementos a ordenar y para otros el tamaño de una matriz cuadrada.

En [6], página 25, se asume que cada ejecución de la i-ésima sentencia de un algoritmo tarda una cantidad de tiempo constante c_i, al cual llamaremos costo de la i-ésima sentencia.
Para observar cómo es que se realiza este conteo, veamos el algoritmo que se muestra en [6], página 26:

Algorithm 1 Ordenamiento por inserción(A)

		cost	*times*
1	**for** $j = 2$ *to* $A.length$ **do**	c_1	n
2	$key = A[j]$	c_2	$n-1$
3	Insert $A[j]$ into the sorted sequence $A[1..j-1]$	0	$n-1$
4	$i = j-1$	c_4	$n-1$
5	**while** $i > 0$ and $A[i] > key$ **do**	c_5	$\sum_{j=2}^{n} t_j$
6	$A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7	$i = i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8	$A[i+1] = key$	c_8	$n-1$

En el pseudocódigo anterior, t_j denota el número de veces que el ciclo while necesita correrse para el valor j, donde $j = 2, 3, ..., n$.

El costo temporal del algoritmo en función del tamaño n de la entrada, denotado por $T(n)$, será la suma de los costos multiplicados por los tiempos, esto es:

$$T(n) = c_1 n + c_2(n-1) + 0(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

Ahora, consideremos que A es un arreglo de número enteros ordenados de manera descendente.

El procedimiento es el siguiente: cada elemento de este arreglo A será comparado con todos los demás elementos del subarreglo ordenado, por lo que esto sucederá $j-1$ veces para cada valor $j = 2, ..., n$, lo que resulta en que $t_j = j$ para todo $j = 2, ..., n$. Calculando, se obtiene:

[3] "***Complejidad espacial**: mide el número de unidades de memoria que necesita un algoritmo (o una simple sentencia) para resolver un problema con una entrada de tamaño N, y se denota por S(N).*" (Esta definición la tomamos de [23], página 8).

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$$

$$\sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$

Con esto podemos hacer un cálculo del costo temporal del algoritmo para la peor instancia de tamaño n:

$$T(n) = c_1 n + c_2(n-1) + 0(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

$$= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \left(\frac{n(n+1)}{2} - 1 \right) + c_6 \left(\frac{n(n+1)}{2} \right) + c_7 \left(\frac{n(n-1)}{2} \right) + c_8(n-1)$$

$$= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2} \right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8 \right) n - (c_2 + c_4 + c_5 + c_8)$$

y como toda c_i es constante, igualamos:

$$a = \frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}$$
$$b = c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8$$
$$c = c_2 + c_4 + c_5 + c_8$$

y sustituyendo en lo obtenido se tiene por tanto que en el peor caso, el costo temporal necesario que le toma al algoritmo resolver tal instancia, y por tanto el máximo tiempo que le toma al algoritmo resolver cualquier instancia, es:

$$T(n) = an^2 + bn + c \quad , \textit{donde } a, b, c \textit{ son constantes que dependen de los costos } c_i.$$

Así, obtuvimos una función cuadrática que nos sirve para calcular el máximo costo temporal del algoritmo para cualquier valor del tamaño de la instancia A, es decir, de n.

Ahora, una vez visto cómo calcular la complejidad temporal, veremos las definiciones formales que nos llevan a ello.

Basándonos en [1], podemos definir para un algoritmo una función $T(n) : \mathbb{N} \to \mathbb{N}$ tal que $T(n)$ es igual al máximo costo de tiempo computacional (cantidad máxima de sentencias ejecutables que implican un costo o tiempo) que le lleva a un algoritmo correr para instancias de tamaño n.

Ahora necesitamos definir algún concepto que nos ayude a medir la eficiencia temporal de los algortimos al resolver problemas. Notemos que cuando el tamaño de la entrada crece, los algoritmos demoran más. En consecuencia, es de interés conocer el comportamiento asintótico de $T(n)$ de los algoritmos para n suficientemente grande.

Para las siguientes definiciones nos basamos en [1], y son las que nos ayudarán a medir la eficiencia temporal de los algoritmos:

Definición. (Notación Big-Oh). Sean f,g dos funciones tal que $f(n), g(n) : \mathbb{N} \to \mathbb{N}$. Decimos que

1. $f(n) \in O(g(n))$ si existe una constante $c > 0$ tal que $0 \leq f(n) \leq c \cdot g(n)$ para toda n suficientemente grande. Esto es, $f(n)$ crece a una razón menor o igual que $g(n)$. De otro modo, $c \cdot g(n)$ es una cota superior asintótica de $f(n)$ para alguna constante $c > 0$.

2. $f(n) \in \Omega(g(n))$ si $g(n) \in O(f(n))$. Esto es, la función $f(n)$ crece a una razón mayor o igual que $g(n)$. De otro modo, $g(n)$ es una cota inferior asintótica de $c \cdot f(n)$ para alguna constante $c > 0$.

3. $f(n) \in \Theta(g(n))$ si $f(n) \in O(g(n))$ y $g(n) \in O(f(n))$. De otro modo, $f(n) \in \Theta(g(n))$ si existen dos constantes c_1 y c_2 mayores que cero tal que $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ para n suficientemente grande. En consecuencia, $f(n)$ y $g(n)$ crecen a la misma razón, y $f(n)$ está acotada, para algunas constantes c_1 y c_2 mayores que cero, de manera asintótica tanto inferior como superiormente por las funciones $c_1 \cdot g(n)$ y $c_2 \cdot g(n)$, respectivamente.

La definición 1. sirve para dar una cota superior al costo temporal de un problema para determinado algoritmo (se encuentra calculando el costo de la peor instancia de tamaño n); la definición 2. para una cota inferior (se conoce al calcular la mejor instancia de tamaño n); y la definición 3. sirve para mostrar igualdad en el costo temporal (cuando el costo de la peor y la mejor instancia de tamaño n coinciden). De ellas nos vamos a centrar en la notación $O(\cdot)$ ("Big-Oh")[4], pues es la que nos ayudará a entender en qué consisten los problemas P y NP.

Definimos el **orden de crecimiento** como el conjunto de funciones cuyo comportamiento asintótico creciente es considerado equivalente ([11]). Por ejemplo, el orden de crecimiento $O(n^3)$ contiene todas las funciones cuyo orden de crecimiento es cúbico, por ejemplo $f(n) = 5n^3$; $g(n) = 7n^3 + 8n^2 + 4$; $r(n) = n^3 + 1$, son tal que $f, g, r \in O(n^3)$.

La siguiente tabla (la cual se encuentra en [11], página 21) muestra algunos órdenes de crecimiento, los cuales, en el análisis de algoritmos, van de los ideales a los menos deseables:

Orden de crecimiento	Nombre
$O(1)$	Constante
$O(log_b n)$	Logarítmico (para cualquier b)
$O(n)$	Lineal
$O(nlog_b n)$	"en log en"
$O(n^2)$	Cuadrático
$O(n^3)$	Cúbico
$O(c^n)$	Exponencial (para cualquier c)

Finalmente daremos unas últimas definiciones que nos servirán para entender la categoría de problemas P y NP:

Definición. Decimos que un algoritmo A es de **tiempo polinómico** si el costo temporal $T(n)$ de tal algoritmo cumple que $T(n) \in O(n^k)$ para algún $k \in \mathbb{N}$ y para todo valor n, donde n es el tamaño de la entrada de A.

Definición.[5] Decimos que un problema M es **tratable** si existe al menos un algoritmo de tiempo polinómico que resuelva todas las instancias de M. Caso contrario, diremos que M es **intratable**.

[4] A $O(\cdot)$ se le conoce como **notación Big-Oh**; $\Omega(\cdot)$ como **notación Omega**, y $\Theta(\cdot)$ como **notación Theta**.

[5] Para esta definición nos apoyamos en [27], página 42.

4.6.2. P vs. NP

Al empezar con un problema normalmente lo primero que vemos son los casos más sencillos, cuando tenemos pocas soluciones posibles y podemos compararlas todas, pero cuando analizamos casos más generales nos encontramos con diferentes escenarios a considerar.

Podemos ir subiendo poco a poco la complejidad de un problema, especialmente tomando valores cada vez más altos del tamaño de la instancia, para ir anotando el incremento que va teniendo la cantidad de operaciones que necesita un algoritmo determinado para resolver un problema. De esta manera podremos relacionar el tiempo computacional que tarda un algoritmo en encontrar una solución a un problema en concreto, con la complejidad cuando esta última va incrementando. Con esto nos es posible comparar la eficacia de los algoritmos.

El tiempo computacional que tarda la computadora en encontrar con un algoritmo la solución a un problema al ir aumentando la dificultad puede ir incrementando de forma logarítmica, lineal, polinomial o no polinomial.

Para los dos problemas presentados al inicio, tanto el de corte como el de empaquetamiento, aún no se han encontrado algoritmos que encuentren la mejor solución en un tiempo corto o razonable, o lo que es lo mismo, no se ha encontrado ningún algoritmo A que resuelva esos problemas tal que $T_A(n) \in O(n^k)$ para algún $k \in \mathbb{N}$. En consecuencia, el problema de corte y el de empaquetamiento son intratables.

Llamamos clase P (*polynomial (time)*) al conjunto de problemas de decisión (más adelante explicamos qué es un problema de decisión) para los cuales podemos encontrar la solución en un tiempo razonable, lo que sería a lo más en un tiempo polinomial. De otro modo, la clase P contiene todos los problemas de decisión tratables.

Ahora, para los dos problemas mencionados, si bien aún no se conoce ningún algoritmo que los resuelva eficientemente, sí se puede comprobar si una solución dada es correcta de manera "sencilla". Por ejemplo, para el problema de corte basta ver que no se traslapen los objetos y que salgan completos, es decir, que no se obtengan objetos incompletos, y para el problema de empaquetamiento basta ver que no se traslapen los objetos y que se encuentren dentro del límite marcado por el objeto grande. Así, este tipo de problemas pertenecen a la clase de problemas NP (*nondeterministic polynomial (time)*), la cual se define como el conjunto de problemas de decisión para los cuales podemos verificar la validez de todas sus soluciones en un tiempo razonable, o lo que es lo mismo, a lo más polinomial, por lo que para un problema que esté en la clase NP puede ser muy difícil y tardado encontrar la mejor solución (por ejemplo, que el tiempo que tarde un algoritmo en encontrar el óptimo aumente exponencialmente conforme la complejidad del problema crece) pero comprobar si una respuesta dada es válida para el problema resulta relativamente sencillo. Por ejemplo, resolver un cubo rubik de $1000 \times 1000 \times 1000$ puede resultar sumamente difícil, pero comprobar si es correcta la solución es fácil ya que basta con verificar que cada cara conste únicamente de un color.

Todo problema P es NP, pero aún no se sabe si todo problema NP es P. Comprobar que $P = NP$ o $P \neq NP$ es uno de los 7 problemas matemáticos abiertos del milenio.

El problema que desarrollaremos en el presente trabajo está en el conjunto de problemas NP[6], por lo que intentaremos encontrar alguna vía para encontrar una buena aproximación a la solución óptima.

[6] [9] dan una explicación del por qué este problema pertenece a la clase NP. Más aún, muestran es NP-duro e incluso NP-completo, conceptos que veremos más adelante, al igual que una breve explicación de cómo lo concluyeron.

4.6.3. Las tres versiones de un problema

De acuerdo con Singh, A. et al., en [29], un problema tiene tres versiones, las cuales son:

1) **Problema de optimización:** Dada una instancia del problema, construir una solución óptima.
2) **Problema de evaluación:** Dada una instancia del problema, encuentra el costo de una solución óptima.
3) **Problema de decisión:** Dada una instancia del problema y un número entero E, determina si existe una solución factible S cuyo costo $c(S)$ sea $c(S) \leq E$, en caso de ser un problema de minimización, o $c(S) \geq E$, si es un problema de maximización. Este problema devuelve dos posibles resultados: 1, en caso de cumplir la desigualdad, y 0 en caso contrario.

Las tres versiones del Problema de la Manta Rectangular son:

Sean $S \in F_S$, donde S es una solución del problema y F_S el conjunto de todas las soluciones factibles, IO una Imagen Objetivo y $c(S) = \sum_{i=1}^{n} \sum_{j=1}^{m} (|S - IO|)_{i,j}$ el costo de la solución S. Entonces:
1) **RBP como problema de optimización:** Dada una IO, construir una manta $S^* \in F_S$ tal que $c(S^*) = \text{mín}_{S \in F_S}\{c(S)\}$.
2) **RBP como problema de evaluación:** Dada una IO, encontrar el valor $c(S^*)$ que cumpla $c(S^*) = \text{mín}_{S \in F_S}\{c(S)\}$.
3) **RBP como problema de decisión:** Dada una IO y un entero E, ¿existe una manta $S^* \in F_S$ tal que $c(S^*) \leq E$?

Con lo visto podemos ver dos definiciones más:
Def. Decimos que un problema de decisión P_D está en el conjunto de problemas P si P_D es tratable.
Def. Decimos que un problema de decisión P_D está en el conjunto de problemas NP si para cada una de sus soluciones S que den como resultado "1" puede verificarse con un algoritmo de tiempo polinómico A que S es efectivamente solución factible.

4.6.4. Problemas NP-completos y NP-duros

Ahondaremos un poco más en la dificultad de los problemas.
Antes de definir los problemas NP-completos, vamos a dar la definición de la reducción de un problema de decisión L_1 a otro problema de decisión L_2. Lo siguiente fue tomado del artículo [21] escrito por Zoltán Ádám Mann:

"Una reducción de un problema L_1 a un problema L_2 es una función f con las siguientes propiedades:
(i) f puede ser computada en un tiempo polinomial.
(ii) f mapea cada entrada de L_1 a una entrada de L_2.
(iii) para cada entrada x de L_1: la respuesta a L_1, cuando la entrada es x, es 'sí' si y sólo si la respuesta a L_2 con entrada igual a $f(x)$ es 'sí'."

De otro modo, en la teoría de la complejidad computacional:

Def. Sean P_1 y P_2 dos problemas de decisión. El problema P_1 es **Karp-reducible** a un problema P_2 si existe una función o algoritmo f computable en tiempo polinomial tal que para toda instancia x de P_1:
(i) $f(x)$ es instancia de P_2.
(ii) el resultado de x en P_1 es *'sí'* si y sólo si el resultado de $f(x)$ en P_2 es *'sí'*.

La notación para decir que P_1 es reducible a P_2 es $P_1 \leq P_2$. La importancia de este término es que si ocurre que $P_1 \leq P_2$ y conocemos un algoritmo que encuentra soluciones para P_2, entonces ese mismo algoritmo nos puede servir como subrutina para encontrar soluciones del problema P_1, o de otro modo, tal algoritmo induce de alguna forma soluciones para P_1. Además, si un deteminado algoritmo resuelve el problema P_2 de manera eficiente, entonces tal algoritmo puede emplearse como subrutina para resolver P_1 eficientemente. Finalmente, que $P_1 \leq P_2$ indica que P_1 no puede ser más difícil de resolver que P_2, pero también que P_2 es al menos tan difícil de resolver como P_1.

Con lo anterior en mente, nos apoyamos en [21] para dar la siguiente definición: un problema P está en la categoría de problemas NP-duros (*NP-hard*) si todos los problemas que se encuentran en la categoría de problemas NP pueden ser reducidos a la versión *Problema de decisión* del problema P.

Por último, en [21] se define que un problema P es NP-completo (*NP-complete*) si está en la categoría de problemas NP y además está en la categoría de problemas NP-duros[7]. Aquí es importante aclarar que no todo problema NP-duro es NP-completo, o de otro modo, no todo problema NP-duro está en la categoría de problemas NP, y un ejemplo de ello es El Problema de la detención (*The Halting Problem*).

En el caso del Problema de la Manta Rectangular, en [9] dan una explicación con la cual muestran que este problema es NP-completo, con lo cual tenemos una razón más para emplear heurísticas y metaheurísticas (en la siguiente sección definimos estos conceptos), pues si intentáramos y lográramos construir un algoritmo que encuentre la solución óptima para este problema, muy probablemente tardaría una eternidad y por tanto la solución dejaría de ser útil. Por otro lado, si la encontrara en un tiempo eficiente, demostraríamos que $P = NP$, problema que no estamos buscando resolver por el momento.

Ahora bien, a continuación una breve explicación extraída casi tal cual de [9] del por qué el Problema de la Manta Rectangular es NP-completo:

Karp mostró en [15], páginas 85-103, que el Problema de Empaquetamiento (*SP*) es NP-completo. Con esto en mente, primero consideremos la versión de decisión del problema *SP* pero con objetos ponderados. Tal problema es el Problema de Empaquetamiento de Peso Máximo (*MWSP*). Es explicado en [9]:
INSTANCIA: Una familia finita F de objetos, pesos enteros positivos denotados por $c(f), f \in F$ y un entero positivo L.
PREGUNTA: ¿Acaso F contiene un subconjunto $B \subseteq F$ de objetos mutuamente ajenos cuyos pesos cumplen $\sum_{f \in B} c(f) \geq L$?

[7]Notemos que en la definición de problemas NP-completos está implícito que todos los problemas en esta categoría son de decisión, ya que todos los problemas en NP son de decisión. Por otro lado, es importante mencionar que todo problema de optimización tiene su versión de decisión. En consecuencia, no todo problema NP-duro es de decisión, pues en esta categoría también podemos encontrar problemas de optimización.

Formulado así, *MWSP* es NP-completo ya que al considerar únicamente objetos con pesos unitarios ($c(f) = 1, f \in F$) se convierte en solucionar *SP*, problema que como mencionamos es NP-completo y en consecuencia *MWSP* también lo es.

Ahora, una versión de decisión del Problema de la Manta Rectangular (*RBP*) es:
INSTANCIA: Una imagen dos-dimensional es representada con una matriz de entradas binarias $I_{W \times H}$, una familia F de rectángulos representados por matrices binarias, pesos positivos enteros $c(R)$ para cada rectángulo $R \in F$ y dos enteros positivos denotados por K y L.
PREGUNTA: ¿Acaso F contiene un subconjunto $B \subseteq F$ de rectángulos mutuamente ajenos (una manta) con cardinalidad $|B| \leq K$ y el peso de la manta cumpla que $\sum_{R \in B} c(r) \geq L$?

Formulado así, *RBP* es NP-completo ya que coincide con el problema *MWSP* (cuando se considera a la familia de objetos únicamente como rectángulos) al configurar $K = |I| \leq W \times H$, lo cual hace reduntante la restricción de cardinalidad (como $|I| =$ cantidad de unos en la matriz, entonces lo anterior hace que pueda usarse un rectángulo por cada uno en la matriz I) y calculando los pesos de los rectángulos con la fórmula:

$$c(R) = \sum_{i=1}^{W} \sum_{j=1}^{H} (R)_{i,j} - 2 \sum_{i=1}^{W} \sum_{j=1}^{H} (I)_{i,j} \times (R)_{i,j}$$

Esta expresión cuenta la cantidad de unos que tiene R y restar dos veces la cantidad de ocasiones en las cuales I y R tienen un uno en las mismas coordenadas de sus respectivas matrices.

Lo que vimos, recapitulando, fue que partiendo de que *SP* es NP-completo, el problema *MWSP*, al considerar únicamente pesos unitarios, se resume en solucionar el problema *SP* y consecuentemente tiene que ser NP-completo, y con ello la versión de decisión de *RBP* al configurarle una restricción de cierta manera, se convierte en un problema *MWSP*, con lo que obtenemos que *RBP* tiene que ser NP-completo.

4.7. ¿Qué son las heurísticas y las metaheurísticas?

4.7.1. Heurísticas

Cuando nos enfrentamos a un problema de optimización para el cual aún no ha sido diseñado un algoritmo capaz de encontrar la mejor solución o, si existe alguno, tarda tanto tiempo que al obtenerla ya no resulta útil, optamos por obtener al menos una solución rápidamente. Es entonces cuando ponemos en una balanza la calidad de la solución contra el tiempo que tardamos en obtenerla.

A lo largo del tiempo se han diseñado distintos algoritmos para obtener soluciones de alta calidad en un tiempo computacionalmente breve. Estos algoritmos incluyen las denominadas técnicas heurísticas y metaheurísticas ([7]).

Las heurísticas son métodos o procedimientos usados para encontrar buenas soluciones a problemas difíciles en un periodo breve de tiempo. Un ejemplo es Búsqueda Local.

4.7.2. Metaheurísticas

Aún con las heurísticas, existen problemas de optimización combinatoria para los cuales estos algoritmos no son suficientemente eficientes, es decir, encuentran soluciones pero no de alta calidad o les toma mucho tiempo computacional encontrar una buena solución. Esto motivó a la búsqueda y desarrollo de nuevas técnicas para poder encontrar buenas soluciones de una manera más eficiente a este tipo de problemas, lo cual dio inicio a las metaheurísticas. El prefijo meta ($\mu\varepsilon\tau\acute{\alpha}$) significa después de o más allá, y se refiere a que estas técnicas van más allá que las heurísticas. Presentaremos tres definiciones:

Son procedimientos de un nivel más alto, en comparación con las heurísticas ([31]), las cuales nos van a dar muy buenas soluciones, llegando en ocasiones a dar la óptima.

Las metaheurísticas son métodos para encontrar soluciones que resultan de la interacción entre procedimientos de mejora local y estrategias de alto nivel para crear un proceso capaz de escapar de óptimos locales y realizar una búsqueda robusta en un espacio de soluciones ([14]).

También, Ibrahim H. Osman y James P. Kelly en [25], página 3, dieron la siguiente definición: *"Las metaheurísticas son una clase de métodos de aproximación diseñados para resolver problemas difíciles de optimización combinatoria en los que los heurísticos clásicos no son efectivos ni eficientes. Las metaheurísticas proporcionan un marco general para crear nuevos algoritmos híbridos combinando conceptos diferentes derivados de heurísticas clásicas, inteligencia artificial, evolución biológica, sistemas neuronales y mecánica estadística."*

Como ejemplos de metaheurísticas tenemos Búsqueda Local Iterada, Búsqueda Tabú, Recocido Simulado y Algoritmos Genéticos.

4.7.3. Construcción de heurísticas y metaheurísticas

Como hemos mencionado, las heurísticas y metaheurísticas son procedimientos para encontrar buenas soluciones, las cuales son aproximaciones al óptimo global o incluso llegan a él. Estas técnicas, para poder implementarlas adecuadamente, requieren de los siguientes ingredientes: una solución inicial, espacio de soluciones, definir vecindades, función que verifique la factibilidad, función objetivo y restricciones del problema.

La solución inicial es fundamental, pues estos procedimientos lo que hacen es tomar una solución y mejorarla. Es a partir de ella donde iniciará el procedimiento de búsqueda de una mejor solución. No es importante que esta solución sea excelente, pues es por medio del procedimiento seleccionado que se buscará mejorar sustancialmente la solución.

El espacio de soluciones y las restricciones están íntimamente relacionadas, pues las restricciones delimitan la región del espacio de soluciones. Para poder saber cuándo un punto cumple las restricciones es necesario evaluarlo en una función que verifique la factibilidad. En esta parte, además, se puede indicar el grado de factibilidad que puede aceptarse, es decir, quizá no me mueva solamente por soluciones factibles, sino que se pueden permitir movimientos hacia puntos fuera de la región factible (esto es opcional).

La función que indicará qué tanto es posible alejarse de la región factible es una función de comparación, la cual asigna una puntuación a la solución y decide si puedes o no moverte hacia esa solución. De esta manera, por ejemplo, un punto que no cumpla todas las restricciones tendrá un puntaje muy bajo y por tanto no se permitirá tan fácilmente (quizá con poca probabilidad o penalizando su valor haciéndolo menos atractivo) moverse a él. En cambio, un punto que no cumpla una restricción pero que esté muy cerca del espacio de soluciones factibles tendrá un

puntaje no tan bajo y en consecuencia existirá una no tan baja probabilidad de movernos a él.

El objetivo de los movimientos a puntos fuera de la región factible es diversificar la búsqueda en el espacio de soluciones, pues sirve especialmente cuando la región de soluciones factibles es no conexa, por lo que al permitir cierta libertad de movimiento a dichos puntos podríamos trazar un camino que nos lleve de una región factible a otra (ambas ajenas[8]) el cual contenga puntos que no son solución, lo cual no podría lograrse si no se permitiese esta flexibilidad.

Las vecindades son esenciales para las heurísticas, pues van a indicar todos los posibles movimientos que tiene permitido hacer cada solución. Con ellas se le da cierto grado de libertad a los movimientos. Si no las definiéramos no sería posible movernos por el espacio de soluciones.

Las vecindades pueden ser tan grandes o pequeñas como se quiera, pero lo importante es que permitan moverse de manera inteligente y rápida por las soluciones. Si la vecindad fuera demasiado grande podríamos realizar una búsqueda extensa en el espacio de soluciones, sin embargo, tardaríamos demasiado tiempo en calcular cada vecindad de cada solución por las que pasemos, lo cual sería contradictorio para los objetivos de estos algoritmos que es obtener buenas soluciones en un tiempo razonable. Por el contrario, si la vecindad fuera muy pequeña, podríamos obtener soluciones rápidamente, pero muy probablemente no serían de alta calidad debido a la limitada libertad de movimiento que las vecindades permitirían y salir de óptimos locales sería complicado. De esta manera lo ideal es definir para el problema que se esté trabajando vecindades que permitan movimientos inteligentes, esto es, que se logre que en la menor cantidad de movimientos posibles nos acerquemos lo más que podamos al óptimo global.

Con lo anterior es claro que para un problema se pueden definir de muchas formas las vecindades, por lo que la manera en la cual se explora el espacio de soluciones depende de cómo estén definidas.

La función objetivo es la que queremos maximizar o minimizar. Esta le va a dar un valor a cada solución que nos ayudará a comparar las soluciones en estos procedimientos y decidir cuál es, para nosotros, más valiosa.

[8]Nos referimos a que si R es la región de soluciones factibles de un problema y es una región no conexa, entonces existen $R_1 \neq \emptyset$ y $R_2 \neq \emptyset$ tales que $R_1 \cup R_2 \subseteq R$ y $R_1 \cap R_2 = \emptyset$

Capítulo 5

La esencia de las Heurísticas: navegando sobre la manta rectangular

Ahora que ya tenemos el modelo matemático continuaremos con los pilares de toda heurística.

5.1. Solución inicial para el problema de la manta rectangular

En este trabajo construiremos las soluciones iniciales de dos formas: **no aleatoria** y **aleatoria**.

5.1.1. Construcción no aleatoria

La primera solución que proponemos para iniciar las heurísticas en este problema es la no aleatoria, y se define como:

Sea $IO \in M_{n\times m}$ la Imagen Objetivo que desamos cubrir. La solución inicial SO que proponemos para cualquier problema con estas características y que darán inicio a las heurísticas es:

$$(SO)_{i,j} = 1 \ \ \forall\, i = 1,2,..,n \ \ y \ \ \forall\, j = 1,2,...,m.$$

donde una vez obtenida SO esta se particiona en k rectángulos (en un momento expondremos la construcción de esta partición).

Esta solución inicial construida con k rectángulos asegura que IO esté cubierta totalmente pero no es la mejor manta, pues es casi seguro que tendrá muchos excedentes. Además será factible por el procedimiento que se aplica al particionar la manta, pues ningún rectángulo se traslapará con otro.

5.1.1.1. Particionamiento en k rectángulos para la solución inicial no aleatoria

La construcción de una manta solución inicial $SO_{n\times m}$ con k (número de rectángulos que debe tener toda solución factible) fijo dada la Imagen Objetivo $IO_{n\times m}$ es como sigue:

Para k = 1. El único rectángulo $R1$ se define:
$R1_{n\times m} = (M_{(1,1),(n,m)})_{i,j}$.
Así, $SO = R1$ será una solución inicial factible con 1 rectángulo.

Esto es, la solución inicial es una manta con un solo rectángulo, cuya matriz asociada es de dimensión $n \times m$ y todas sus entradas son iguales a uno. Claramente es factible.

Para k = 2. Los rectángulos $R1$ y $R2$ los definiremos como:
$R1_{n\times m} = (M_{(1,1),(n,\lfloor \frac{m}{2} \rfloor)})_{i,j}$
$R2_{n\times m} = (M_{(1,\lfloor \frac{m}{2} \rfloor+1),(n,m)})_{i,j}$
De esta manera, $SO = R1 + R2$ será solución inicial factible del problema en cuestión compuesta de 2 rectángulos.

Lo que realizamos es partir por la mitad de manera vertical una matriz $U_{n\times m}$ de la cual todas sus entradas son iguales a uno. El primer rectángulo se construye dejando como están las entradas de la parte izquierda de U y las demás hacerlas 0; análogamente construimos el segundo rectángulo basándonos en la parte derecha.

Para k $\geq$ 3. Los rectángulos $R1, R2, ..., Rk$ los definimos de la siguiente forma:
Sean $k_1 = \lfloor \frac{k}{2} \rfloor$ y $k_2 = k - \lfloor \frac{k}{2} \rfloor$.

Con esto en mente, definimos los siguientes vectores, los cuales servirán para construir adecuadamente k rectángulos no sobrelapados:
$V_1 = (1, \lceil \frac{n}{k_1+1} \rceil, \lceil \frac{2n}{k_1+1} \rceil, ..., \lceil \frac{(k_1-1)n}{k_1+1} \rceil, n+1)$ y $V_2 = (1, \lceil \frac{n}{k_2+1} \rceil, \lceil \frac{2n}{k_2+1} \rceil, ..., \lceil \frac{(k_2-1)n}{k_2+1} \rceil, n+1)$, los cuales tienen longitud $k_1 + 1$ y $k_2 + 1$ respectivamente.

Finalmente definimos los k rectángulos como sigue:

$$R_\alpha \mu_{n\times m} = (M_{(V_1(\mu),1),(V_1(\mu+1)-1,\lfloor \frac{m}{2} \rfloor)})_{i,j} \ \ \forall\ \mu = 1,2,...,k_1$$
$$R_\beta \varphi_{n\times m} = (M_{(V_2(\varphi),\lfloor \frac{m}{2} \rfloor+1),(V_2(\varphi+1)-1,m)})_{i,j} \ \ \forall\ \varphi = 1,2,...,k_2$$

De esta manera, $SO = (\sum_{\mu=1}^{k_1} R_\alpha \mu) + (\sum_{\varphi=1}^{k_2} R_\beta \varphi)$.

Si redefinimos estos rectángulos obtenemos:
$Ri = R_\alpha i_{n\times m} \ \ \forall\ i = 1,...,k_1$ y $R(k1 + j) = R_\beta j_{n\times m} \ \ \forall\ j = 1,...,k_2$.

Con esto la suma anterior quedaría como

$$SO = (\sum_{\mu=1}^{k_1} R_\alpha \mu) + (\sum_{\varphi=1}^{k_2} R_\beta \varphi) = (\sum_{i=1}^{k_1} Ri) + (\sum_{j=1}^{k_2} R(k_1 + j))$$
$$= (\sum_{i=1}^{\lfloor \frac{k}{2} \rfloor} Ri) + (\sum_{j=1}^{k-\lfloor \frac{k}{2} \rfloor} R(\lfloor \frac{k}{2} \rfloor + j)) = \sum_{i=1}^{k} Ri$$

$\therefore SO = \sum_{i=1}^{k} Ri$, esto es k rectángulos construyen una solución inicial factible SO.

Explicaremos ahora esta construcción. Empezamos considerando una matriz $U_{n\times m}$ cuyas entradas son iguales a uno y la dividimos verticalmente en dos mitades. Luego, la parte izquierda la cortamos horizontalmente en $\lfloor \frac{k}{2} \rfloor$ cachitos, donde de cada uno se obtiene un rectángulo dejando ese cachito con entradas iguales a 1 y las demás las igualamos a 0, y de aquí obtenemos $\lfloor \frac{k}{2} \rfloor$ rectángulos. Análogamente, para la parte derecha de U la cortamos horizontalmente con el mismo procedimiento de tal manera que se construyan los rectángulos restantes $(k - \lfloor \frac{k}{2} \rfloor)$.

A continuación, en la siguiente imagen ejemplificamos este procedimiento para una matriz de dimensión $M_{8\times 5}$ con entradas iguales a 1 y $k = 7$. En ella, siguiendo las flechas moradas, empezamos dividiendo verticalmente por la mitad a una matriz de dimensión 8×5 cuyas entradas son todas iguales a uno, aunque como en este caso cuenta con 5 columnas (número impar), dividimos en el punto intermedio $\lfloor \frac{5}{2} \rfloor = 2$, con lo cual conseguimos dos mitades. Luego, calculamos $\lfloor \frac{7}{2} \rfloor = 3$, valor que indica que debemos dividir el lado izquierdo de la matriz en 3 cachitos y posteriormente el lado derecho en $7 - 3 = 4$ cachitos. Después, el lado izquierdo es cortado horizontalmente en 2 puntos y el lado derecho en 3, con el fin de tener en ellos 3 y 4 pedacitos respectivamente. Así, habremos dividido con este procedimiento a la matriz en 7 cachitos.

Finalmente, de cada uno construimos un rectángulo al realizar para cada uno el siguiente procedimiento: las entradas dentro del cachito las dejamos tal cual (iguales a uno) y todas las demás entradas las igualamos a cero. Con esto hemos construido una manta con 7 rectángulos (en la imagen son todos aquellos dentro de la línea punteada color azul).

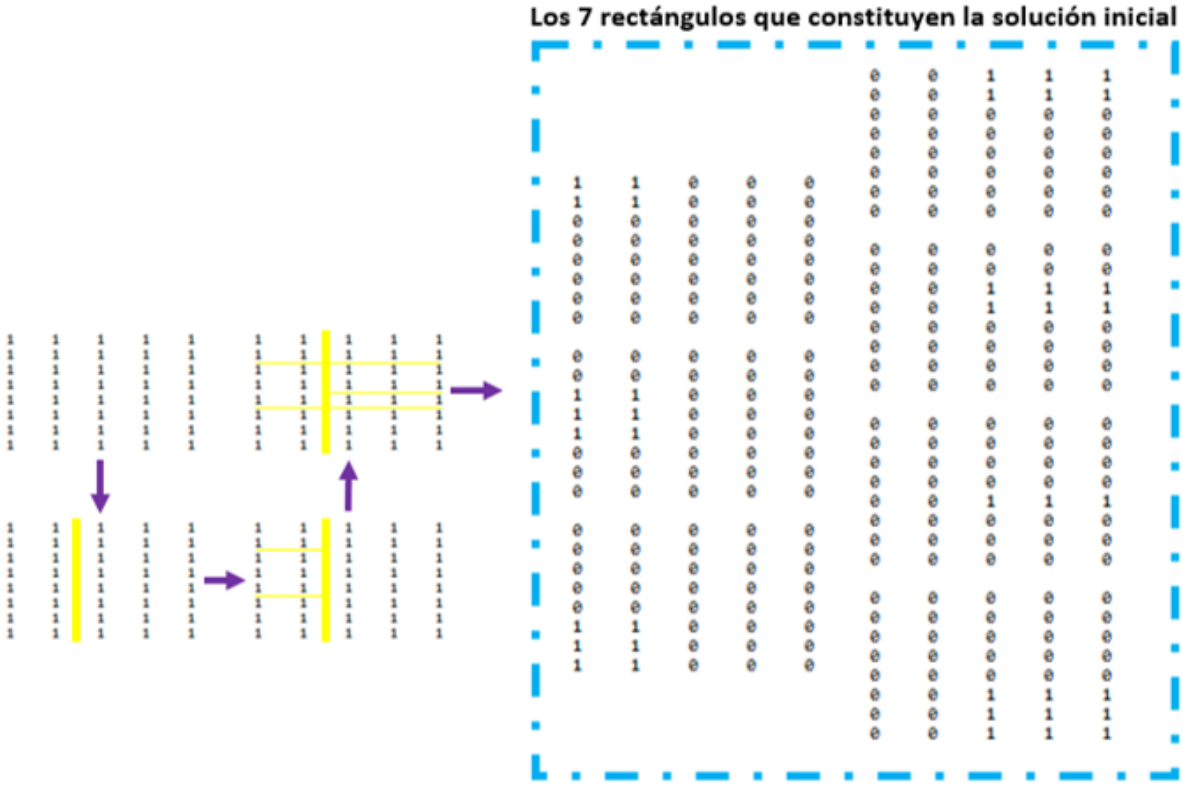


Figura 5.1: Ejemplo construcción rectángulos de solución inicial con k dado

5.1.2. Construcción aleatoria

A continuación proponemos una manera de construir soluciones aleatorias:

Dada la Imagen Objetivo cuya matriz definiremos como IO de dimensión $n \times m$ y un parámetro k fijo (cantidad máxima de rectángulos que debe tener la manta), empezaremos con una matriz M de dimensión $n \times m$ con todas sus entradas iguales a cero.

Realizaremos una partición de M en k matrices como sigue:

Empezaremos definiendo $Aleatorio(\{0,1\})$ una función que regresa aleatoriamente un 0 o un 1, la cual decidirá el tipo de corte a realizar: si $Aleatorio = 0$ quiere decir que se realizará un corte horizontal a la matriz para obtener dos matrices, pero si $Aleatorio = 1$ entonces el corte, para la obtención de dos matrices, se realizará de manera vertical. Además, definiremos los puntos de corte como aquellos números que indican entre qué par de columnas (o filas) se realiza la división de una matriz. Por ejemplo, sea $v = (v_{11}, v_{12}, v_{13}, v_{14})$ y sea $Aleatorio = 1$. Entonces, v tiene 3 puntos de corte verticales: el punto 1 lo divide en (v_{11}) y (v_{12}, v_{13}, v_{14}), el punto 2 en (v_{11}, v_{12}) y (v_{13}, v_{14}) y el punto 3 en (v_{11}, v_{12}, v_{13}) y (v_{14}).

Es importante notar que una matriz de dimensión $n \times m$ tiene $n-1$ puntos de corte horizontales, los cuales dividen en dos a la matriz de $n-1$ maneras diferentes, y $m-1$ puntos de corte verticales, que dividen en dos a la matriz de $m-1$ formas diferentes.

Con lo anterior en mente, lo primero que haremos será comparar el valor de k:

Si $k = 1$ ya acabamos la partición, por lo que la matriz $M1_{n \times m}$ es igual a la misma matriz M, pues M es partición de sí misma. Caso contrario, elige con $Aleatorio$ el tipo de corte que se realizará.

Si el corte es horizontal o 0, se elige de manera aleatoria el punto de corte en las filas. De otro modo, si el tipo de corte es vertical o 1, procedemos a escoger de manera aleatoria el punto de corte en las columnas.

Sin pérdida de generalidad supongamos $Aleatorio = 0$, lo que implica un corte horizontal, y supongamos el corte se realiza en el punto i, con $1 \leq i \leq n-1$. Así, obtendremos una partición, que constará de dos matrices, a partir de la matriz original, las cuales serán $M1_{i \times m}$ y $M2_{n-i \times m}$.

Después comparamos de nuevo el valor de k:

Si $k = 2$ ya acabamos, pues construimos una partición de M en dos matrices. Caso contrario, lo que haremos ahora será elegir de manera aleatoria la matriz $M1$ o la matriz $M2$.

Una vez seleccionada alguna de ellas, procedemos a aplicarle el mismo procedimiento que mencionamos, esto es, obtener el tipo de corte que le realizaremos y luego el punto de corte para así construir dos nuevas matrices, las cuales sustituirán a la matriz de la cual fueron obtenidas.

Una vez construidas, tendremos ahora una partición de M con tres matrices: $M1$, $M2$ y $M3$.

Posteriormente, compararemos de nuevo el valor de k:

Si $k = 3$ hemos terminado el proceso de partición, pero si no es así procedemos a escoger de manera aleatoria una matriz, ya sea $M1$, $M2$ o $M3$, y una vez se haya elegido se vuelve a decidir aleatoriamente tanto el tipo como el punto de corte que se le aplicará, de tal manera que dos nuevas matrices, partición de ella, le sustituirán, con lo cual llegaremos a una nueva partición de M que tendrá cuatro matrices: $M1$, $M2$, $M3$ y $M4$.

Este procedimiento se repite de manera iterativa hasta llegar a la igualdad con el valor de k, esto es, hasta haber realizado $k-1$ cortes, con lo cual, finalmente, habremos logrado la construcción de una partición de M en k matrices.

Luego de obtener la partición $\{M1, M2, ..., Mk\}$ de M, lo que haremos será utilizar cada una de estas matrices para construir k rectángulos que conformen una manta factible.

Sea Mj con j fijo tal que $1 \leq j \leq k$. Iniciaremos identificando la posición de Mj dentro de M. Como Mj forma parte de la partición de $M_{n\times m}$, podremos obtener sus entradas correspondientes a las esquinas superior izquierda y la inferior derecha ubicadas en el interior de la matriz M. Tales entradas, respectivamente, son (a,b) y (c,d), con $1 \leq a \leq c \leq n$ y $1 \leq b \leq d \leq m$. De esta manera, conoceremos la posición de tal matriz dentro de M.

El siguiente paso consiste en seleccionar de manera aleatoria cuatro números enteros, que nos referiremos a ellos como α, β, γ y δ, tales que cumplen $a \leq \alpha \leq \gamma \leq c$ y $b \leq \beta \leq \delta \leq d$. Estos números van a definir la ubicación de un rectángulo, el cual tendrá la característica de encontrarse dentro de las coordenadas de Mj. Tal rectángulo será $Rj_{n\times m} = M_{(\alpha,\beta),(\gamma,\delta)}$.

Este procedmiento descrito se realiza para cada Mi, con $i = 1, ..., k$, por lo que construimos k rectángulos, los cuales son $R1, R2, ..., Rk$.

Como fueron construidos cada uno dentro de matrices que de manera conjunta formaban una partición de M, ninguno de ellos encimará a ningún otro rectángulo dentro de este conjunto de rectángulos, por lo que la solución $S = \sum_{i=1}^{k} Rk$ cumple que ninguno de sus Ri rectángulos se enciman y además éstos están alineados a los ejes, todo esto por construcción.

5.1.2.1. Ejemplo de la construcción de soluciones aleatorias

Supongamos que queremos construir una solución aleatoria para una Imagen Objetivo cuya dimensión es de 7×8 y fijando $k = 5$. En la Figura 5.2 se muestra cómo se va particionando una matriz de ceros hasta obtener k matrices, con el fin de construir un rectángulo dentro de cada uno de los pedazos de la partición. El proceso en cada paso compara el valor de k con la cantidad de matrices que tiene la partición actual, y se detiene hasta que la partición cuente con k matrices. Si la partición aún no tiene k matrices iteramos como sigue: escogemos de manera aleatoria cualquier pedazo de la partición y seleccionamos aleatoriamente tanto el tipo de corte (ya sea horizontal o vertical) como un punto de corte (indica dónde cortar la matriz elegida) para obtener dos submatrices que le sustituyan en la partición.

Finalmente repetimos el proceso, comparando la cantidad de matrices que tiene la partición actualizada con el valor de k.

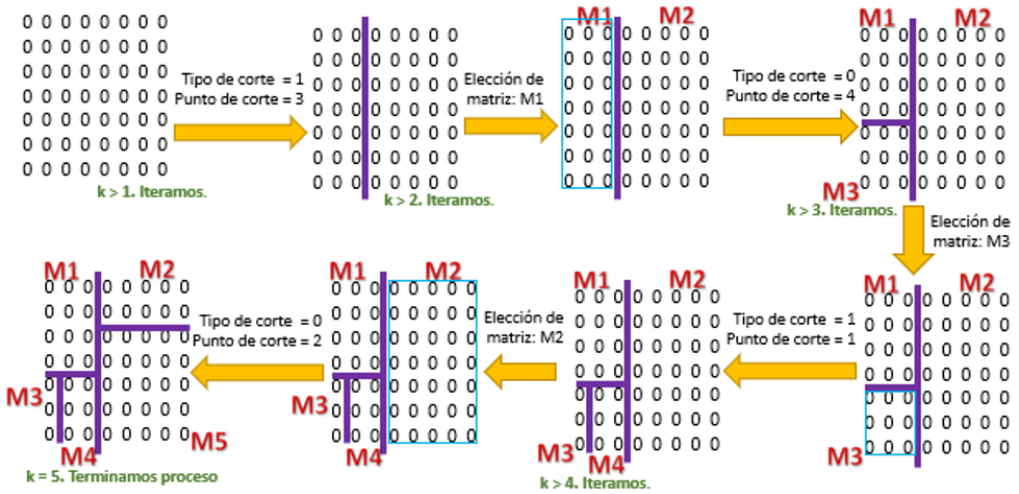


Figura 5.2: Ejemplo construcción de soluciones aleatorias (partición)

Una vez construida la partición, se trabaja por separado con cada una de las matrices que contenga para construir k rectángulos.

En la siguiente imagen vemos reflejado este procedimiento, el cual inicia identificando la posición que ocupa dentro de la matriz. Luego, seleccionando aleatoriamente dos pares de valores que se encuentren dentro de las coordenadas del pedazo con el cual se está trabajando, tales que cumplan las desigualdades vistas, y finalmente construyendo un rectángulo a partir de ellas.

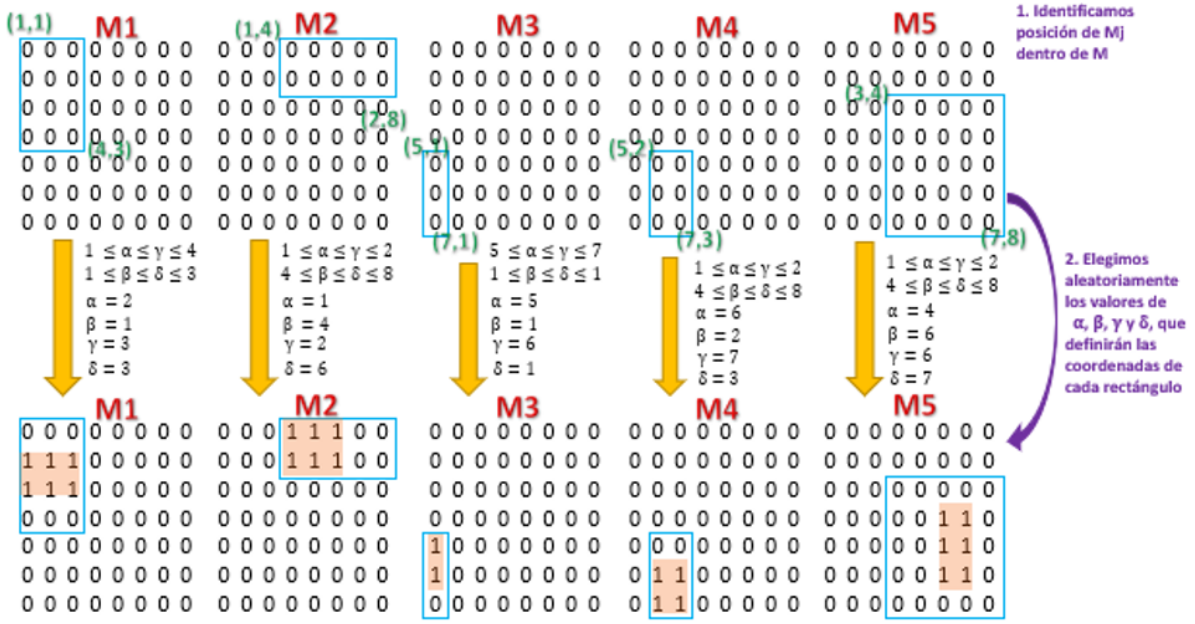


Figura 5.3: Ejemplo construcción de soluciones aleatorias (construcción rectángulos)

Con esto hemos obtenido los k rectángulos que constituirán a la solución factible S.

Como en secciones anteriores hemos mencionado, de manera matricial la manta se refleja únicamente sumando las matrices de los rectángulos, por lo que, finalmente, la solución es: $S = R1 + R2 + R3 + R4 + R5$, y para ilustrar cómo es que resulta lo anterior, a continuación mostramos la representación matricial de la solución junto con los rectángulos que codifica:

$$\begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Figura 5.4: Ejemplo construcción de soluciones aleatorias (Manta obtenida)

5.2. Construcción de vecindades para el problema de la manta rectangular

Antes de definir las vecindades que usaremos, he de recordar dos aspectos a tener en cuenta de cómo trabajaremos con k.

Dada una matriz IO, la cual representa la imagen a cubrir, los posibles valores que podrá tomar k cuando usamos la construcción no aleatoria son:

$$1 \leq k \leq (2 * |Renglones\ de\ la\ IO|) - 2$$

En el caso de usar la construcción aleatoria, la cual decidimos emplear en todas las heurísticas, k podrá tomar los siguientes valores:

$$1 \leq k \leq |Columnas\ de\ la\ IO| * |Renglones\ de\ la\ IO|$$

Además, empezamos trabajando con un caso particular del modelo, pues en lugar de que el valor de k que decida fijarse se defina como el número máximo de rectángulos a usar, será **el número de rectángulos que la manta debe de tener**, pero una vez finalizada la heurística aplicamos una función a la manta obtenida que elimina aquellos rectángulos tales que al quitarlos resulte en una mejor solución.

5.2.1. Vecindad que usaremos

Primero necesitamos definir los movimientos que permitiremos.

Sea $(R_{(t_1,t_2),(t_3,t_4)})_{n\times m}$ un rectángulo de dimensión $n \times m$ con $1 \leq t_1, t_3 \leq n$ y $1 \leq t_2, t_4 \leq m$, donde (t_1, t_2) indica la entrada donde aparece el primer uno del rectángulo y (t_3, t_4) la última entrada donde se encuentra el último uno del rectángulo, contando de izquierda a derecha, empezando en la primera fila y siguiendo hacia abajo. Definimos las **modificaciones** (o **configuraciones**) de tal rectángulo como sigue:

- **Agrandar hacia arriba**: Es el nuevo rectángulo $(R_{(t_1-1,t_2),(t_3,t_4)})_{n\times m}$.
- **Agrandar hacia abajo**: Es el nuevo rectángulo $(R_{(t_1,t_2),(t_3+1,t_4)})_{n\times m}$.
- **Agrandar hacia la derecha**: Es el nuevo rectángulo $(R_{(t_1,t_2),(t_3,t_4+1)})_{n\times m}$.
- **Agrandar hacia la izquierda**: Es el nuevo rectángulo $(R_{(t_1,t_2-1),(t_3,t_4)})_{n\times m}$.
- **Achicar desde arriba**: Es el nuevo rectángulo $(R_{(t_1+1,t_2),(t_3,t_4)})_{n\times m}$.
- **Achicar desde abajo**: Es el nuevo rectángulo $(R_{(t_1,t_2),(t_3-1,t_4)})_{n\times m}$.
- **Achicar desde la derecha**: Es el nuevo rectángulo $(R_{(t_1,t_2),(t_3,t_4-1)})_{n\times m}$.
- **Achicar desde la izquierda**: Es el nuevo rectángulo $(R_{(t_1,t_2+1),(t_3,t_4)})_{n\times m}$.

Finalmente, la vecindad que usaremos para cualquier solución de este problema la vamos a definir como sigue:

Sea M una manta solución con k rectángulos. La vecindad de la manta M, denotada por $N(M)$, se define como un conjunto que contiene todas las posibles soluciones factibles que pueden obtenerse al aplicarse cada modificación a cada uno de los k rectángulos de M.

En la siguiente imagen se muestran las ocho modificaciones que se pueden aplicar a un rectángulo. De color amarillo las modificaciones de agrandar el rectángulo y con color verde las modificaciones correspondientes a achicar el rectángulo:

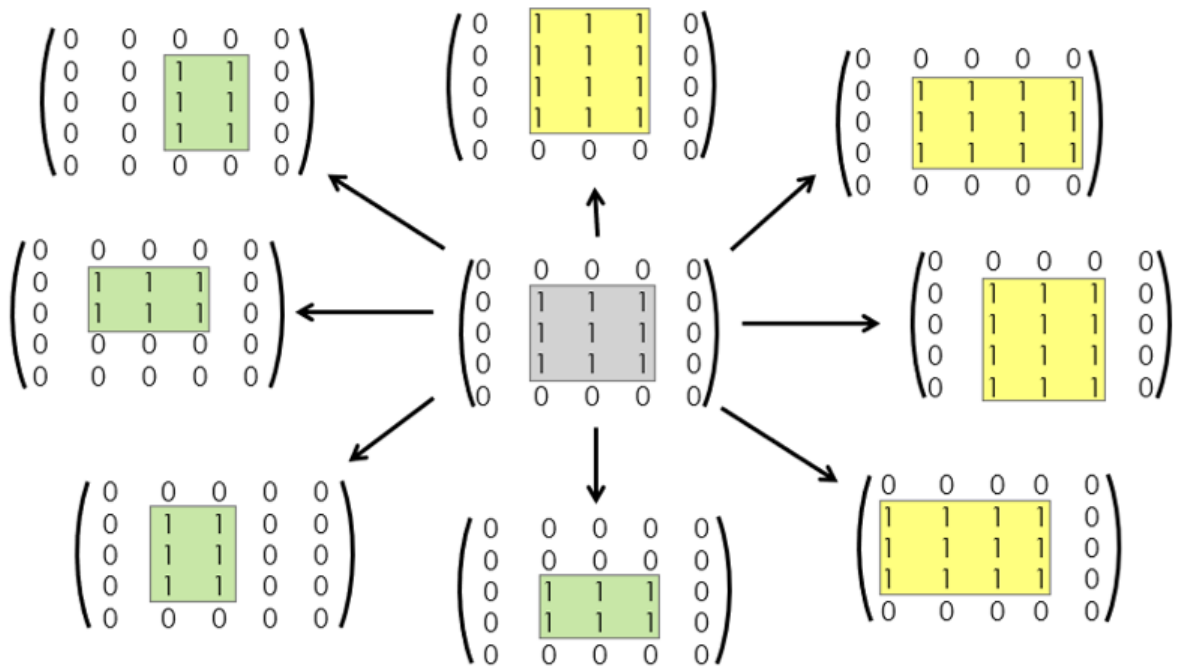


Figura 5.5: Modificaciones aplicadas a un rectángulo

Como mencionamos, las vecindades constan únicamente de soluciones factibles, por lo que si tenemos rectángulos adyacentes y les aplicamos las modificaciones para conseguir soluciones vecinas, aquellas donde queden rectángulos encimados debido a la aplicación de una modificación no estarán en la vecindad.

A continuación se presenta un ejemplo de la vecindad de una manta (color gris) aplicándole las modificaciones posibles cuando $k = 2$ (soluciones con modificación de agrandar se muestran de color amarillo, mientras que de achicamiento color verde):

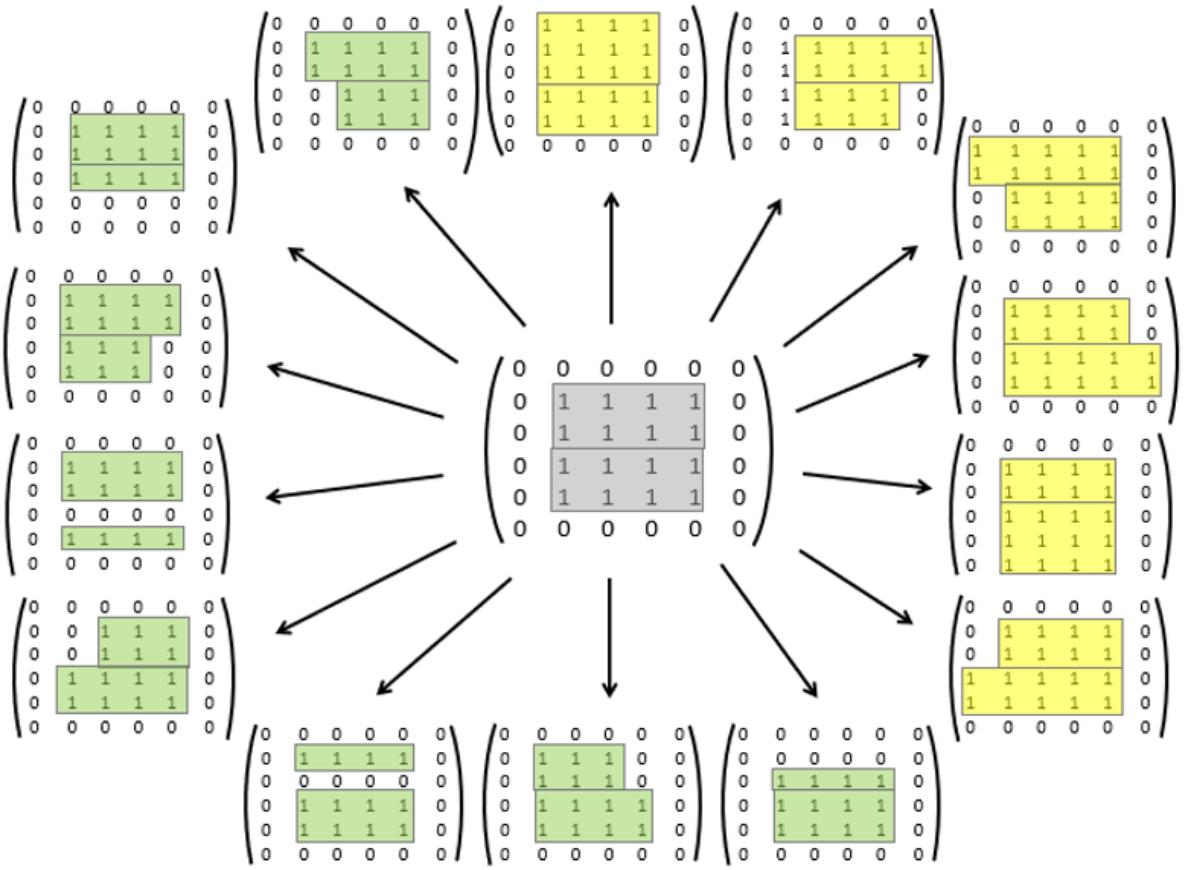


Figura 5.6: Ejemplo modificaciones cuando $k = 2$

Capítulo 6

Implementación de heurísticas y metaheurísticas para el Problema de la Manta Rectangular

A continuación explicaremos en qué consisten las heurísticas y metaheurísticas que implementaremos para buscar soluciones de alta calidad en el Problema de la Manta Rectangular y comentaremos los resultados obtenidos de sus usos.

El objetivo es comparar las soluciones obtenidas con cada una, así como el tiempo que tardan en encontrarlas, con el fin de decidir cuál de las heurísticas o metaheurísticas funciona de mejor manera, en general, para este problema.

Además, con cada heurísticas mostraremos un ejemplo para el siguiente problema:

Problema del Cuadrado Naranja: Acomodar un cuadrado color naranja, denotémosle R, de tamaño 5×5 en un mallado definido, de tal manera que maximice la cantidad de círculos azul claro con posición fija (denotados por o_i con $i = 1, \ldots, t$ donde t es la cantidad total de círculos y además $O = \bigcup_{i=1}^{t}$ es el conjunto que contiene todos los círculos) que contenga restringido a que el cuadrado esté completamente dentro del mallado, sus aristas se encuentren alineadas a los ejes y que el interior del cuadrado cubra por completo o no cubra nada de los cuadraditos formados por el mallado.

La siguiente imagen presenta la **solución inicial** con la que trabajaremos.

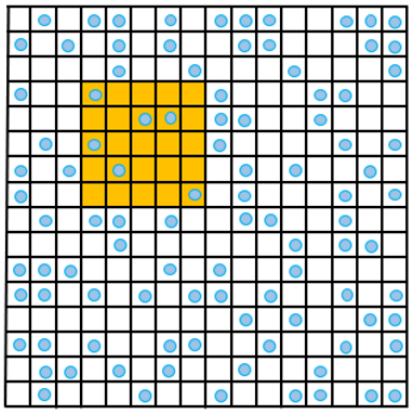


Figura 6.1: Solución inicial que usaremos en el Problema del Cuadrado Naranja

Las vecindades de un cuadrado R_s constan de máximo 4 cuadrados. Estos se obtienen moviendo ya sea una unidad hacia arriba ($R_{s,ar}$), o una hacia abajo ($R_{s,ab}$), o una a la izquierda ($R_{s,iz}$) o una a la derecha ($R_{s,de}$) a R_s. Contiene máximo 4 cuadrados, pues cuando R_s se encuentra pegado en alguna pared no podrá realizar algún o algunos movimientos, pues de hacerlo una parte de él quedaría fuera del mallado, lo cual no está permitido en el problema.

De esta manera, si R_{si} es un cuadrado que no se encuentra pegada en ningún borde límite del mallado, su vecindad contiene 4 cuadrados:

$$N(R_{si}) = \{R_{si,ar}, R_{si,ab}, R_{si,iz}, R_{si,de}\}$$

A continuación presentamos una imagen que ilustra lo anterior:

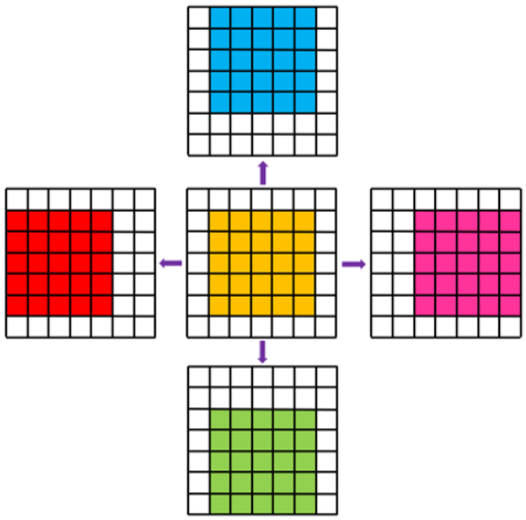


Figura 6.2: Ejemplo vecindad en el Problema del Cuadrado Naranja

Se aprecia que la vecindad del cuadrado naranja es:

$$N(\blacksquare) = \{\blacksquare, \blacksquare, \blacksquare, \blacksquare\}$$

6.1. Búsqueda Local

Búsqueda local es un procedimiento iterativo que en cada paso selecciona una solución que se encuentre en la vecindad de la solución actual y de acuerdo con algún criterio decide si moverse o no a esa solución, de tal manera que vaya construyéndose un recorrido en el espacio de soluciones hasta cumplir un criterio de parada o no se pueda seleccionar ninguna otra solución dentro de la vecindad que cumpla el criterio definido.

Con búsqueda local podremos, si lo aplicamos iterativamente y con el criterio de que en cada paso únicamente nos movemos por soluciones que mejoren la solución actual hasta no poder encontrar soluciones en la vecindad que cumplan este criterio, llegar a óptimos locales.
A continuación un posible pseudocódigo para búsqueda local:

Algorithm 2 Búsqueda Local

$s = GeneraUnaSoluciónInicial()$; # Se encuentra una solución s para trabajar a partir de ella
while $N(s) \neq \emptyset$ **do**
  $s_v = SoluciónAleatoria(N(s))$; # $N(s)$ es la vecindad de s
  if s_v es mejor solución que s **then**
    $s = s_v$
    Actualizamos vecindad $N(s)$
  else
    $N(s) = N(s) - s_v$
  end if
end while

6.1.1. Ejemplo de aplicación de Búsqueda Local

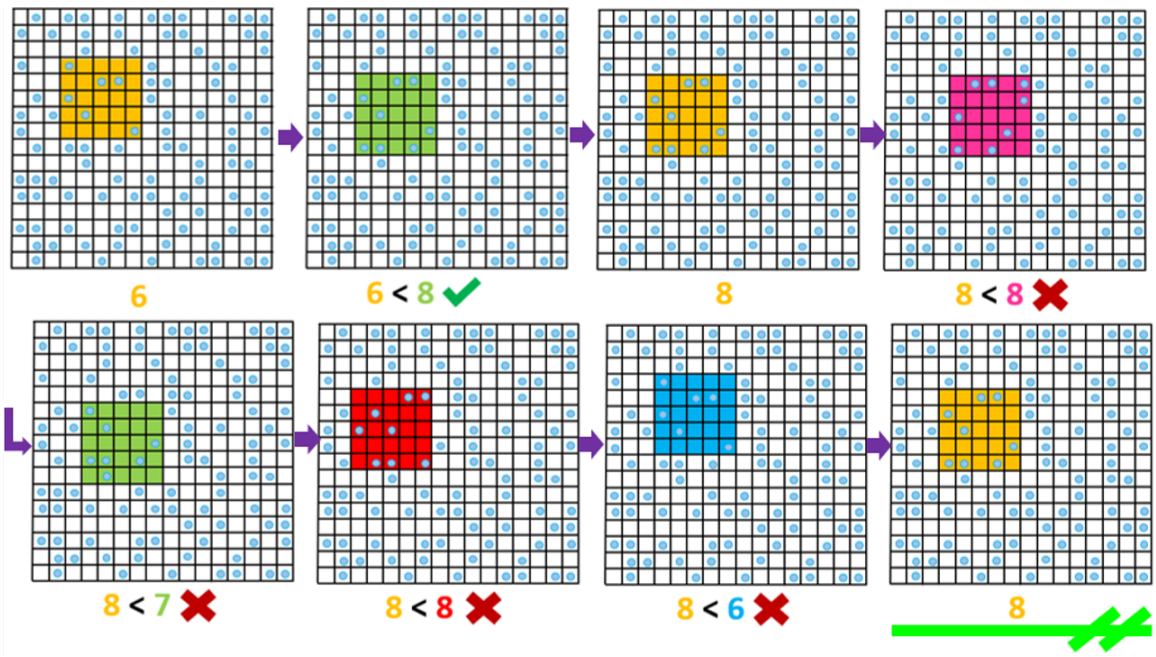


Figura 6.3: Ejemplo Búsqueda Local

En la imagen se inicia con un cuadro naranja como solución inicial. Luego elige aleatoriamente una solución en la vecindad, en este caso elige moverse hacia abajo (cuadrado verde), compara este cuadrado con el naranja y como el movimiento mejora el número de círculos empaquetados, realizamos el movimiento. Luego actualiza la solución y el cuadro verde se vuelve naranja. Posteriormente elige una solución aleatoria de la vecindad, en este caso elige moverse hacia la derecha (color rosa) y compara círculos encerrados, pero como el movimiento no mejora el número de círculos empaquetados que se tenían, no realiza el movimiento. Después elige al azar un cuadrado en la vecindad distinto al rosa, en este caso elige moverse hacia abajo (color verde), pero como al comparar círculos resulta no mejorar la cantidad, no se mueve a él. Posterior a ello, de nuevo, elige al azar un cuadrado distinto a los mencionados, en este caso

elige moverse hacia la izquierda (color rojo), pero al comparar círculos empaquetados la cantidad no mejora, por lo cual decide seguir intentando con otras soluciones de la vecindad actual. Finalmente elige el último cuadrado con el cual aún no ha comparado cantidad de círculos empaquetados, esto es moverse hacia la derecha (color azul), pero como, de nuevo, la solución no mejora decide no moverse a ella. Al buscar otro cuadrado en la vecindad actual se da cuenta que ya probó todas las opciones, por lo tanto se queda donde está y devuelve esa solución como resultado.

En este ejemplo la cantidad de círculos empaquetados al aplicar Búsqueda Local mejoró en 2 unidades.

6.1.2. RBP: Implementación Búsqueda Local

6.1.2.1. Consideraciones para Búsqueda Local

Se implementó tal cual, es decir, calcula la vecindad de la solución actual, elige aleatoriamente una de esas soluciones y si esa solución es mejor que la actual, entonces nos movemos a esa solución, actualizamos la solución definiendo a la que nos movimos como solución actual y repetimos la comparación calculando la vecindad de la solución actualizada, pero si resulta no ser mejor que la solución actual entonces elegimos aleatoriamente otra solución diferente dentro de esta vecindad y realizamos este mismo procedimiento de comparación hasta agotar todas las posibles soluciones dentro de la vecindad en la que nos encontremos, lo que indicará, cuando ya no haya soluciones que comparar, que la solución actual es la mejor dentro de su entorno y por lo tanto hemos llegado a un óptimo local.

6.2. Búsqueda Local Iterada

Es una metaheurística que puede verse como una extensión de la heurística Búsqueda Local. Construye un camino aleatorio sesgado en el espacio de soluciones al realizar iterativamente un procedimiento que consiste, primero, en una búsqueda local para llegar a un óptimo local. Después, a tal solución aplicarle algunas modificaciones o perturbaciones que permitan un movimiento inteligentemente por el espacio de soluciones, y finalmente un criterio de aceptación para decidir si continuar o no la búsqueda en la solución encontrada [4]. De esta forma, en lugar de aplicar a muchas soluciones aleatorias una búsqueda local, se realizan múltiples búsquedas locales a diversas soluciones que en cada paso van mejorando, es decir, en cada paso se desea encontrar un óptimo local que sea mejor que el anterior.

La idea esencial de Búsqueda Local Iterada es enfocar la búsqueda en un subespacio del espacio de soluciones definido por óptimos locales. [19].

Esta metaheurística consta de cuatro fases:

1. **Generar una Solución Inicial:** Esta solución será el punto de partida de la heurística.

2. **Búsqueda Local:** Se aplica un procedimiento de búsqueda local con el objetivo de mejorar la solución inicial y llegar a un óptimo local.

3. **Perturbación:** En esta fase se genera un nuevo punto de partida al aplicar perturbaciones o modificaciones a la solución que se obtuvo con la búsqueda local [32]. Los objetivos de esta fase son modificar la solución actual para generar una nueva y prometedora solución

de partida para la siguiente aplicación de búsqueda local [30], y escapar del óptimo local actual [32]. Además, como nos explican en [30], si la perturbación a la solución es muy débil o pequeña se corre el riesgo de regresar al óptimo local del cual se partió, mientras que si se realizan demasiadas modificaciones a la solución se podrá perder gran parte de la calidad y estructura del óptimo local que se halló.

4. **Criterio de aceptación:** Se usa un criterio de aceptación para decidir el siguiente punto de partida. Si la solución obtenida con la anterior fase pasa este criterio, entonces ese punto se convertirá en la solución actualizada, mientras que si no lo pasa, entonces el punto de partida será alguna solución obtenida previamente. Como nos comentan en [32], la elección de este criterio es importante ya que, junto a la fase de Perturbación, controla el balance entre la intensificación y la diversificación de la búsqueda de soluciones.

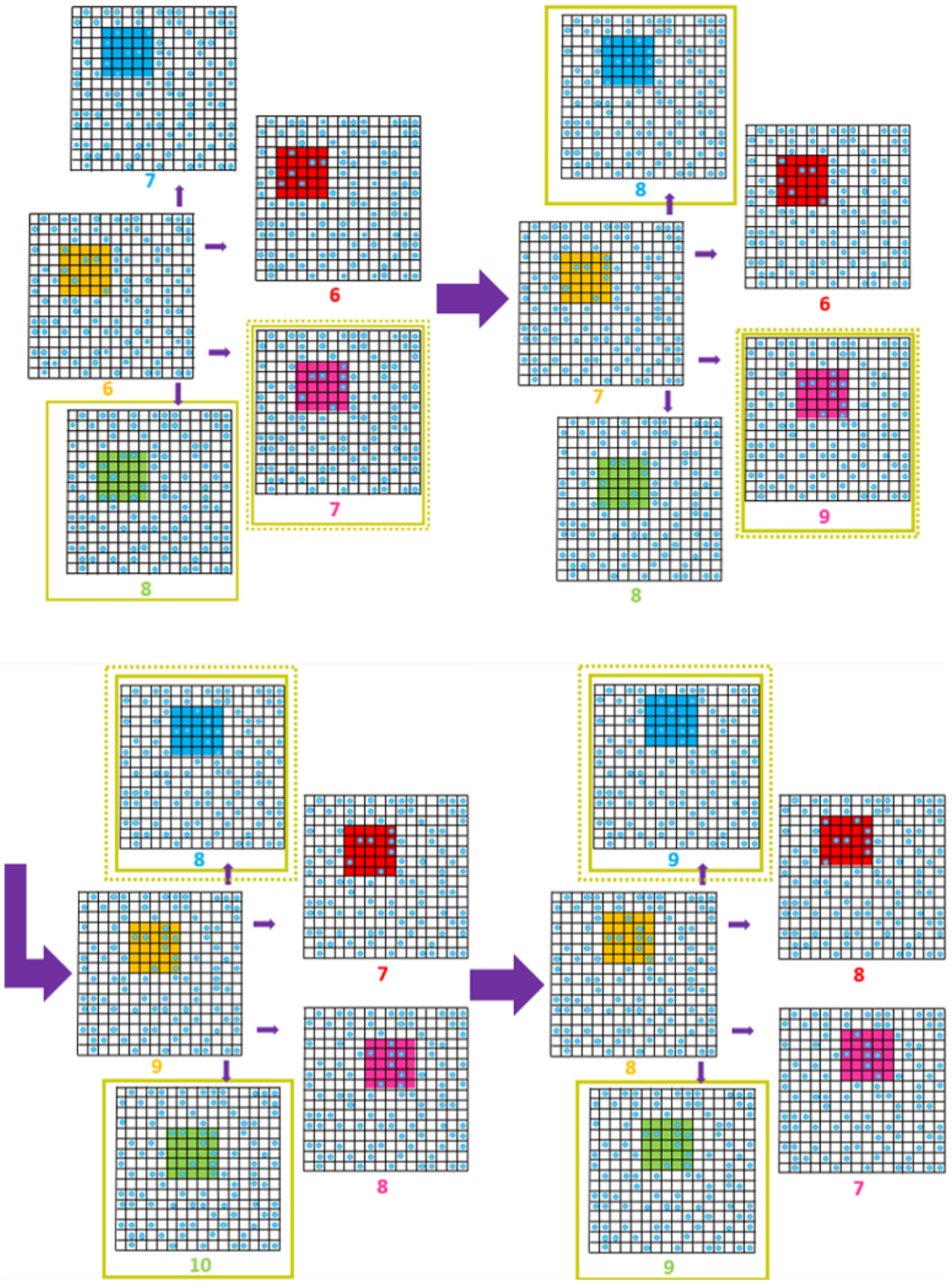
El siguiente pseudocódigo lo tomamos de [32], página 205:

Algorithm 3 Búsqueda Local Iterada

$s = GeneraUnaSoluciónInicial()$; # Fase de Generar una Solución Inicial s
$s = BúsquedaLocal(s)$; # Fase de Búsqueda Local
while criterio de paro no se cumpla **do**
  $s* = Perturbación(s, historia)$; # Fase de Perturbación
  $s* = BúsquedaLocal(s*)$; # Fase de Búsqueda Local
  $s = CriterioDeAceptación(s, s*, historia)$; # Fase de Criterio de aceptación
end while

6.2.1. Ejemplo de aplicación de Búsqueda Local Iterada

Para el siguiente ejemplo, a la solución inicial se le aplicó directamente la fase de Perturbación, la cual consiste en realizar durante cuatro iteraciones lo siguiente: seleccionar aleatoriamente una de las dos mejores soluciones de la vecindad y actualizar la solución. El criterio de aceptación es: si el cuadrado no empeora en más de dos su valor, entonces actualiza la solución. Luego de realizar una vez esta fase, a la solución obtenida se le aplicó Búsqueda Local.



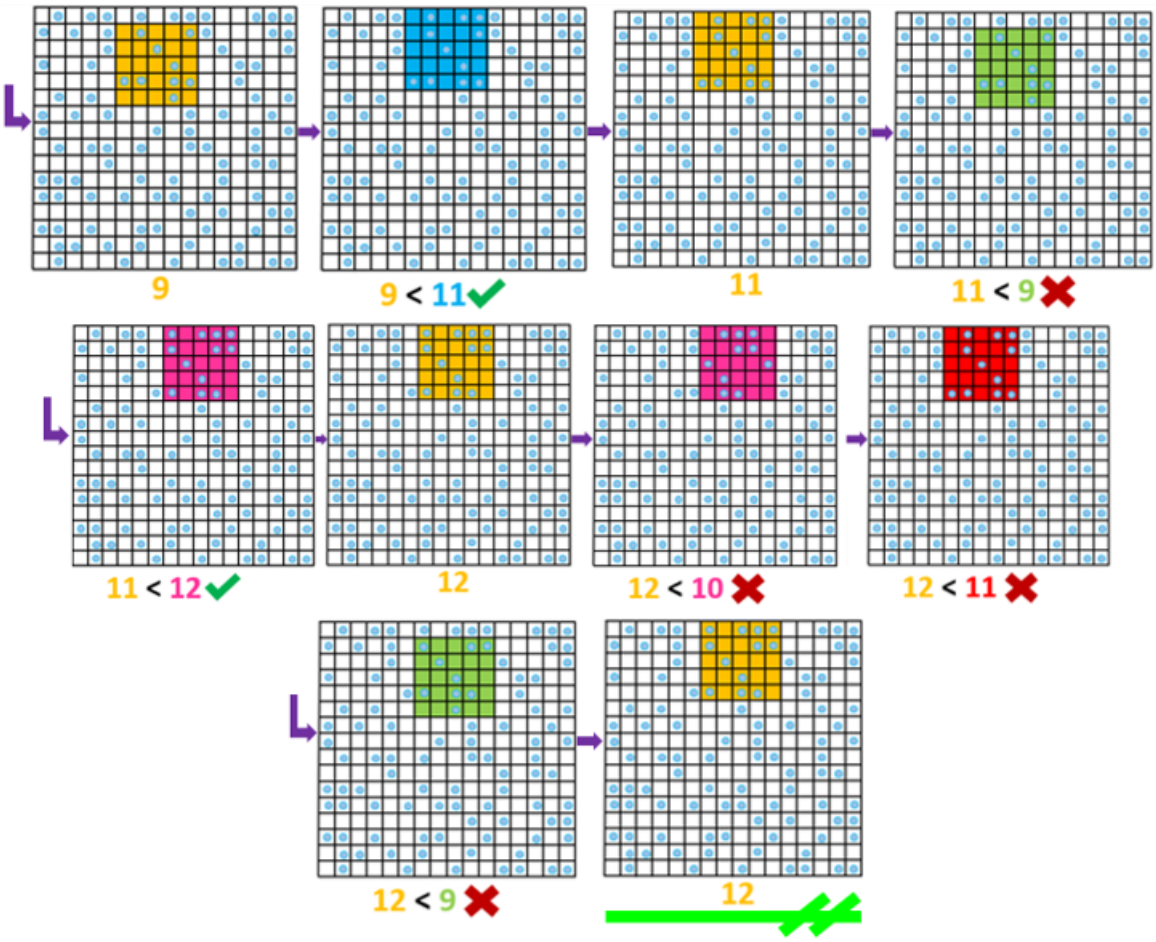


Figura 6.4: Ejemplo Búsqueda Local Iterada

Así, logramos obtener un cuadrado el cual empaqueta 12 círculos y que además es la mejor solución por la cual se pasó.

6.2.2. RBP: Implementación Búsqueda Local Iterada

6.2.2.1. Valores dados a los parámetros y consideraciones para Búsqueda Local Iterada

En total el código repite 30 veces las fases de Perturbación y de Búsqueda Local.

La fase de Perturbación consiste en crear, durante r iteraciones, una lista con los c mejores candidatos de la vecindad actual y moverse a uno de ellos de manera aleatoria, donde $r = max\{t,s \mid IO_{t\times s}\ la\ Imagen\ Objetivo\}$, mientras que $c = [\frac{|Vecinos(A)|}{2}]$, con A la solución actual en la heurística y $[.]$ el entero más cercano.

No usamos ningún criterio de aceptación, pues se observó que se diversificó la búsqueda de manera eficaz.

6.3. Recocido Simulado

Comenzaremos esta sección con la definición de Recocido Simulado presentada en [10]: *"Recocido simulado es un método de optimización inspirado en el proceso de templado de metales. El algoritmo de recocido simulado (SA) es un método iterativo que inicia con cierto estado s. Mediante un proceso particular genera un estado vecino s' al estado actual. Si la energía, o evaluación, del estado s' es mejor que la del estado s se cambia el estado s por s'. Si la evaluación de s' es peor que la de s, entonces se puede "empeorar" eligiendo s' en lugar de s con una cierta probabilidad que depende de las diferencias de las evaluaciones $|\Delta f| = |f(s) - f(s')|$ y de la temperatura actual del sistema T. La posibilidad de elegir un estado peor al actual es lo que le permite a SA salir de óptimos locales para poder llegar a los óptimos globales. "*

La fórmula que acostumbra usarse para decidir si cambiamos o no a un estado peor que el actual es la siguiente: $P(|\Delta f|, T) = e^{-\frac{|\Delta f|}{T}}$.

La temperatura decae gradualmente conforme avanza la heurística. Una manera de actualizarla cada vez que itera el algoritmo es: $T = T * \alpha$, donde $\alpha \in \mathbb{R}$, $0 < \alpha < 1$. Si α se fija cerca de cero la temperatura descenderá rápidamente, mientras que si se fija cerca del uno tardará en enfriar.

Como criterio de paro normalmente se usa un parámetro conocido como *threshold* o umbral, el cual es un número muy pequeño que acota superiormente a T. Esto es, si el valor de T, después de actualizarlo, es más pequeño que un número muy pequeño (fijo), termina el ciclo.

El siguiente pseudocódigo representa una idea general del algoritmo de recocido simulado:

Algorithm 4 Recocido Simulado

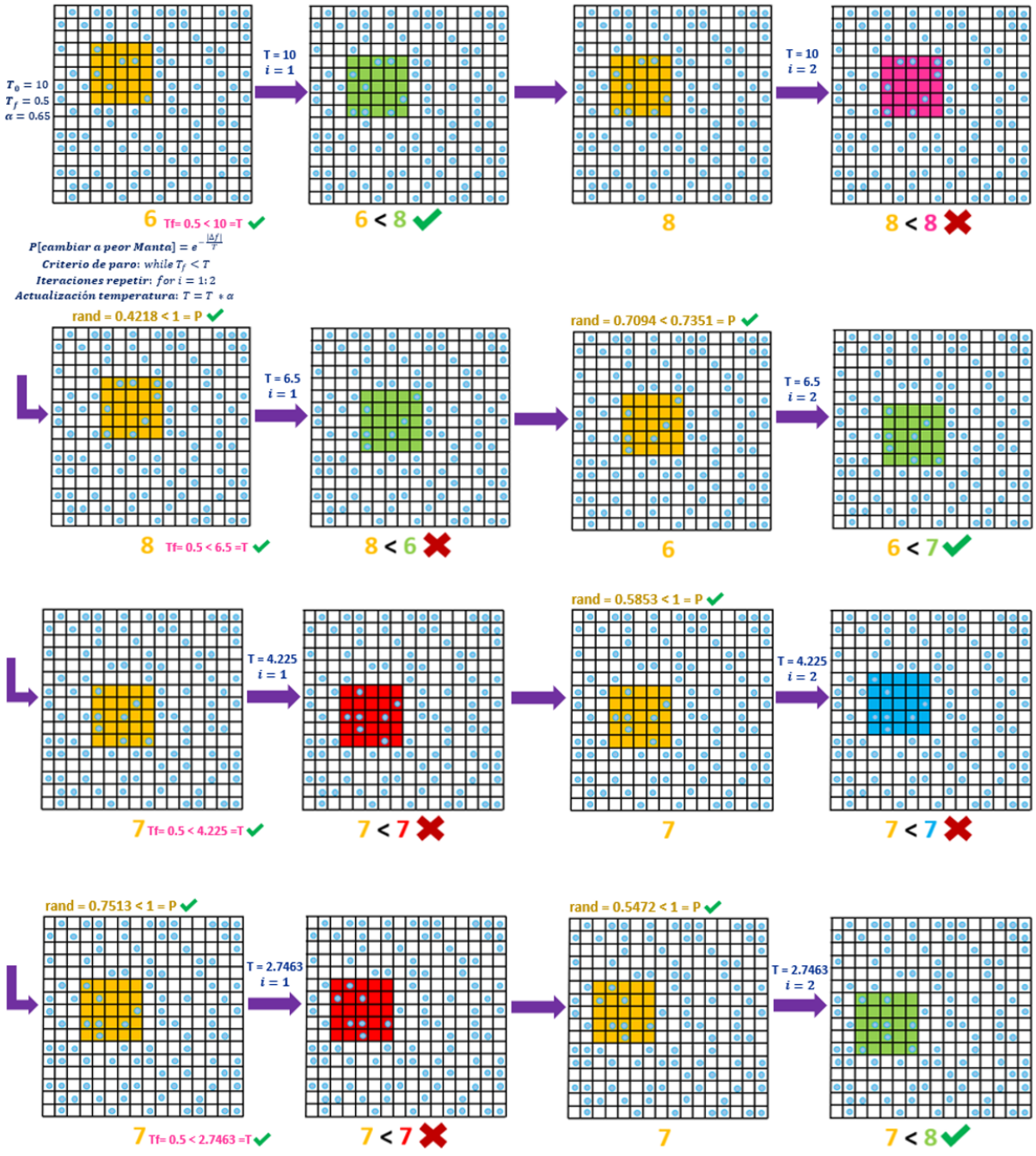
```
s = GeneraUnaSolucionInicial(); # Se encuentra una solución inicial s
T = T_0; # Se define una temperatura inicial T con la que se trabajará
while criterio de paro no se cumpla do
    # Con el criterio de paro irá disminuyendo el valor de T hasta que alcance cierto valor
    for número de iteraciones a repetir do
        # Ciclo for para iterar un número fijo de veces antes de actualizar T lo que sigue
        s' = TomaUnVecinoAleatorioDe(s); # Escogemos de manera aleatoria a un vecino de
        # s y lo guardamos como s'
        if s' es mejor solución que s then
            s = s'; #Guardamos a s' como nueva solución s para trabajar con ella
        else
            # Si la solución s' no es mejor que la solución s, iniciamos el siguiente ciclo:
            if random(0,1) < CriterioDeProbabilidad(f(s), f(s'), T) then
                s = s'; # Nos cambiamos a la solución s' y la guardamos como nuestra nueva s
            else
                s = s; # No nos cambiamos de solución
            end if
        end if
    end for
    actualiza temperatura T; # Se actualiza valor de T, el cual disminuirá
end while
```

6.3.1. Ejemplo de aplicación de Recocido Simulado

En este ejemplo iniciamos con una temperatura inicial $T = 10$, la cual va actualizándose (cada dos iteraciones contadas por la condición $for\ i = 1:2$) con la fórmula $T = T \cdot \alpha$ donde $\alpha = 0.65$ establece la proporción mediante la cual la temperatura disminuye.

Cuando se compara la Manta actual con una Manta vecina y esta última resulta ser peor, si la siguiente desigualdad se cumple entonces permitimos el cambio a la solución vecina y actualizamos la solución: $rand < e^{-\frac{|\triangle f|}{T}}$ donde *rand* es un número aleatorio dentro del intervalo $[0,1]$ y $|\triangle f|$ es la diferencia entre el valor de la Manta actual y la Manta vecina.

El algoritmo termina cuando la siguiente desigualdad, el cual es el criterio de paro, deja de cumplirse: $T_f < T$.



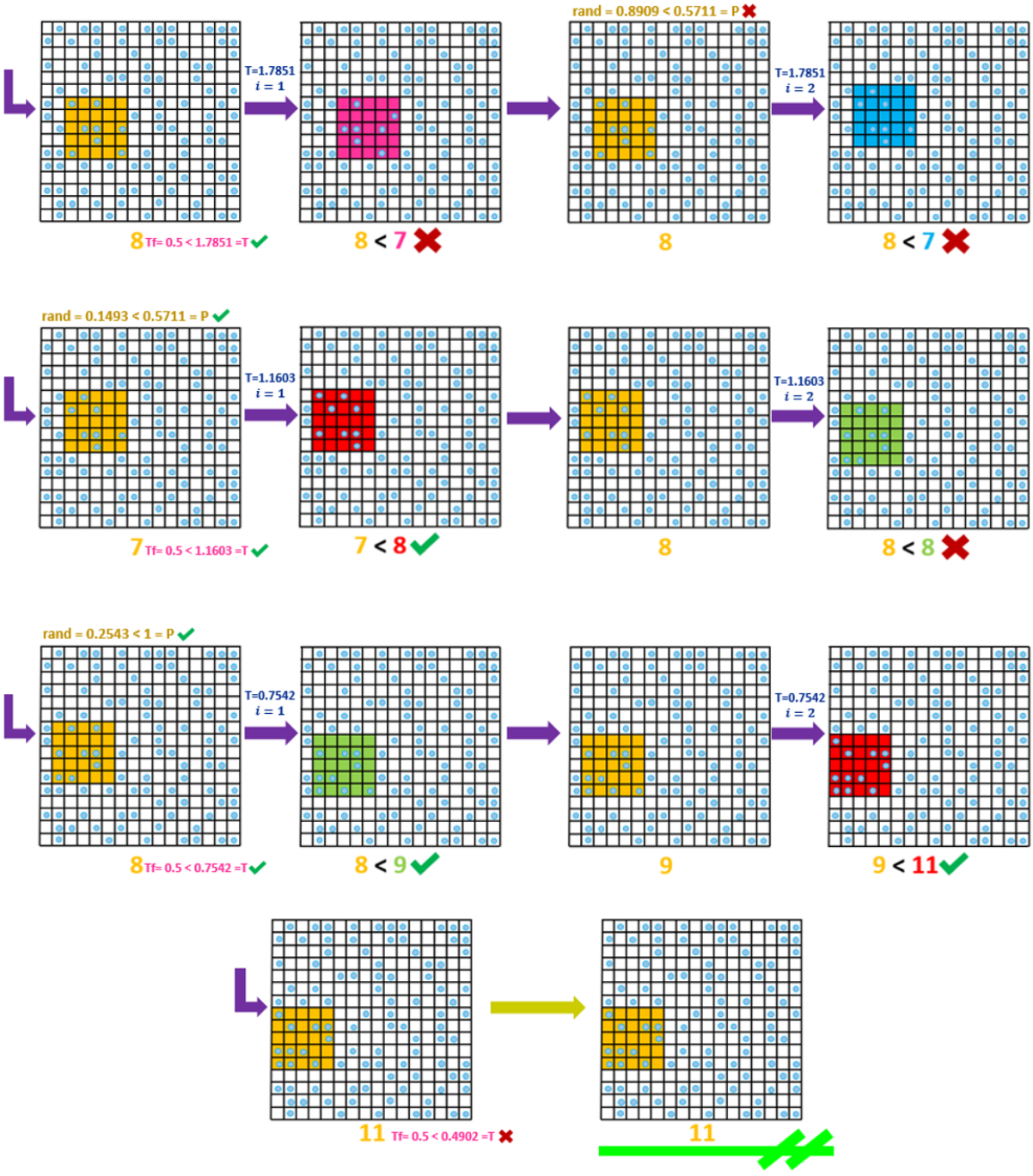


Figura 6.5: Ejemplo Recocido Simulado

En la secuencia anterior los cuadrados color naranja indican la solución actual hasta ese momento. Las soluciones de otros colores son soluciones vecinas, las cuales se compara el valor para ver si cambiamos y actualizamos la solución o no.

La palomita verde a la derecha de una desigualdad que se encuentre debajo del mallado indica que la solución vecina es mejor y por tanto nos cambiamos a tal solución, pero en caso de que sea un tache este indica que la solución vecina no es mejor y por tanto con cierta probabilidad decidiremos si nos movemos a ella o no. Esto último lo marca la desigualdad color dorado la cual compara el valor de un número aleatorio entre el intervalo 0 y 1 con

el valor resultante de evaluar $e^{-\frac{|\triangle f|}{T}}$, y si el valor aleatorio es menor que el valor calculado mediante la fórmula permite el cambio a la solución vecina (indicado por flecha verde a la derecha de esta desigualdad), caso contrario no permite el movimiento (marcado con tache rojo) y permanecemos en la misma solución.

La desigualdad color rosa compara el valor de la temperatura actual, el cual se muestra actualizado mediante la fórmula $T = T \cdot \alpha$, con el valor de T_f, y si la desigualdad se cumple (una palomita verde lo indica) seguimos iterando, pero en caso de que no se cumpla (marcado por un tache rojo) el algoritmo termina y devuelve la mejor solución encontrada en todo el algoritmo.

En el ejemplo llegamos a una solución cuyo valor es 11.

6.3.2. RBP: Implementación Recocido Simulado

6.3.2.1. Consideraciones para Recocido Simulado

Para esta heurística el criterio de probabilidad implementado con el cual decidimos si nos movemos a una solución peor que la actual es:

Si $r < e^{-|\frac{valor(B)-valor(A)}{T}|}$, entonces nos movemos a la solución B; caso contrario nos quedamos en A, donde:
$r \in [0,1]$ es un número aleatorio, $valor(A)$ es el valor de la manta actual, $valor(B)$ es el valor de la manta que está en la vecindad de A, con $valor(B) < valor(A)$ y T es el valor de la temperatura actual.

6.3.2.2. Valores dados a los parámetros en Recocido Simulado

- $T = 100$: T es el valor de la temperatura con el cual iniciamos e irá actualizándose en pasos posteriores.
- $T_{final} = 0.005$: Este parámetro es la temperatura a partir de la cual, para valores menores o iguales a ella, termina la heurística.
- $\alpha = 0.995$: Con este parámetro, cuya única restricción es que sea menor que uno pero mayor que cero, actualizamos el valor de la temperatura con la siguiente fórmula: $T = T * \alpha$.

El valor de la temperatura es actualizado cada 20 iteraciones.

6.4. Búsqueda Tabú

En [26] se menciona que la metaheurística Búsqueda Tabú surge en un intento de dotar de "inteligencia" a los métodos de búsqueda local, pues tiene por principio básico auxiliar a estos métodos para poder explorar el espacio de soluciones más allá de óptimos locales, cuyo objetivo es dirigir la búsqueda teniendo en cuenta la historia de ésta.

Esta metaheurística nos permite escapar de óptimos locales y evitar ciclados.

La característica que lo distingue es el uso de una memoria de corto plazo y una memoria de largo plazo, donde se auxilia con la llamada lista Tabú. La memoria a corto plazo se encarga de recordar los últimos pasos que se han hecho para evitar repetir soluciones y así evitar ciclados. La memoria a largo plazo tiene como objetivos intensificar la búsqueda, regresando a regiones conocidas para explorarlas con mayor profundidad, y diversificar la búsqueda, de manera que se recorran regiones que aún no han sido exploradas con el fin de encontrar mejores soluciones ([26]).

Durante el procedimiento se tiene que guardar la mejor solución encontrada hasta ese paso para dar al final la mejor solución por la que se pasó durante todo el procedimiento.

La memoria a corto plazo puede guardar soluciones en las que se estuvo poco tiempo atrás o algunos de los atributos o características que presenten ciertas soluciones (especialmente las encontradas recientemente). Estos se añaden a la lista denominada Lista Tabú, en la cual permanecerán durante un tiempo hasta que se cumpla un plazo definido o algún criterio.

La memoria a largo plazo puede guardar la cantidad de veces que se ha pasado por una solución, la cantidad de veces que se ha realizado un determinado movimiento o modificación (es decir, por ejemplo, 5 veces a la izquierda y 16 hacia abajo, o 4 permutaciones del primer elemento con el quinto y 20 permutaciones entre el segundo con el tercero) o alguna otra característica que permita buscar en regiones aún inexploradas.

Se cuenta con un Criterio de Aspiración el cual va a permitir movimientos hacia alguna o algunas soluciones que se encuentren en la Lista Tabú (o características o atributos que estén en esa lista) si consideran que el movimiento tendrá como consecuencia mejores resultados. De otra forma, en ocasiones la Lista Tabú contiene elementos con características prometedoras que pueden guiarnos a una mejor solución, por lo que se permite el movimiento, solución o atributo en dado caso. Uno de los criterios más comunes y sencillos es permitir que el o los elementos de la Lista Tabú salgan de ella antes de cumplir su plazo allí siempre que den un mejor valor en la función objetivo que la mejor solución encontrada hasta ese momento (si el elemento de la Lista Tabú, al realizar la evaluación correspondiente en la función objetivo, tiene como consecuencia un mejor valor que el mejor valor que se había encontrado hasta ese momento).

Para la memoria a corto plazo normalmente se tiene que el elemento añadido a la Lista Tabú saldrá cuando hayan pasado n iteraciones o cumplan el criterio de aspiración, y para la memoria a largo plazo muchas veces se guarda la frecuencia o la cantidad de veces que se ha repetido una solución o que se ha hecho un determinado movimiento. También puede guardar soluciones por las que ha pasado antes para que en algún punto regrese a ellas con el fin de explorar otras regiones. Estas pueden irse guardando aleatoriamente durante el procedimiento, almacenar las que presenten ciertas características prometedoras o soluciones que aparezcan en determinadas iteraciones.

El pseudocódigo de Búsqueda Tabú es el siguiente: [1]

[1] En este pseudocódigo, la memoria a largo plazo guarda soluciones en las que se estuvo para intentar recorrer zonas inexploradas.

Algorithm 5 Búsqueda Tabú

```
s = GeneraUnaSolucionInicial();
q = Cantidad máxima de iteraciones en T;
while criterio de paro no se cumpla do
    N(s) = Vecindad(s);
    identificar T Lista Tabú;
    identificar A;
    N*(s) = (N(s) − T) + A;
    escoger la mejor solución r de N*(s);
    s = r;
    actualizar T;
    Guardar mejor solución hasta este paso;
end while
C = Conjunto de soluciones pasadas;
NuevasRutas;  #Indicar el camino por el cual las soluciones en C buscarán mejores
# soluciones en zonas inexploradas
BúsquedaTabú*(C);  # Aplicar Búsqueda Tabú a cada solución en C. No es la
# misma Búsqueda Tabú, pues BúsquedaTabú* no busca obtener soluciones para
# hallar nuevas rutas inexploradas, sino guardar |C| soluciones buenas
return MejorSolucion;
```

En el pseudocódigo se inicia con una solución inicial s y se da un valor a q, el cual indica la cantidad máxima de pasos que puede permanecer un atributo en la Lista Tabú T. Después, inicia un ciclo **while** el cual se detendrá hasta que se cumpla una cantidad predefinida de iteraciones. El ciclo empieza encontrando la vecindad de s, denotada como $N(s)$. Luego, identificamos tanto las soluciones en $N(s)$ que tengan atributos que se encuentren en T como la cantidad de iteraciones que llevan los atributos en esta lista. Posteriormente, identificamos las soluciones de $N(s)$ que tengan características que estén en T pero que cumplan el criterio de aspiración definido, y los guardamos en la lista A. Después, definimos un nuevo conjunto $N^*(s)$, al cual se le conoce como vecindad reducida, que se construye, primero, agregando todas las soluciones de $N(s)$, y segundo, sacando las que presenten atributos que estén en T exceptuando aquellas que estén en A. Después, se escoge la mejor solución de $N^*(s)$ y la denotamos como r. Luego, actualizamos: la solución (realizando $s = r$), la cantidad de iteraciones que han permanecido en T sus elementos, y los elementos de T, añadiéndole el atributo de la solución por la que se pasó y sacando los elementos que ya hayan permanecido en ella durante q pasos consecutivos. El último paso dentro del ciclo es guardar la mejor solución conseguida hasta ese momento. Además en el ciclo while deben guardarse aleatoriamente en un conjunto C, que denominaremos como conjunto de soluciones pasadas, algunas soluciones por las que se haya pasado.

Luego, para cada una de las soluciones en C se indica el camino por el cual buscarán mejores soluciones en zonas inexploradas. Una vez terminen su ruta, a cada una de las soluciones a las cuales hayan llegado se les aplica Búsqueda Tabú, aunque esta vez sin tener en cuenta la memoria a largo plazo[2]. La heurística termina dando como resultado la mejor solución encontrada durante todo el algoritmo.

[2]En este caso, la memoria a largo plazo se encargó de guardar en el conjunto C algunas de las soluciones por las cuales se pasó, con el objetivo de intensificar y diversificar la búsqueda de soluciones.

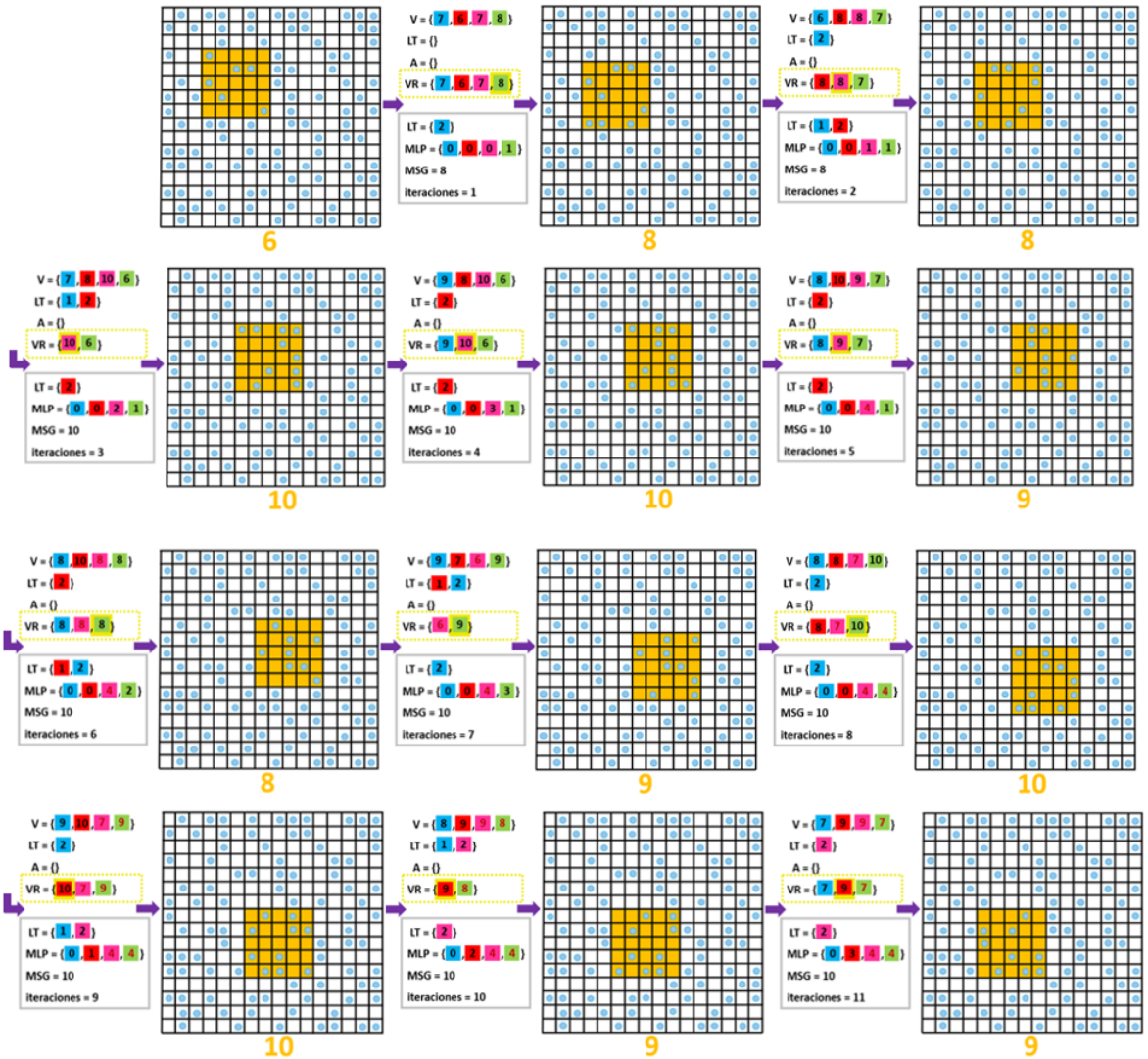
6.4.1. Ejemplo de aplicación de Búsqueda Tabú

En el siguiente ejemplo de Búsqueda Tabú la memoria a corto plazo guarda el movimiento contrario al realizado durante dos iteraciones mandándolo a una Lista Tabú (LT) para que de esta forma se evite regresar a soluciones pasadas y diversifique la búsqueda. La memoria a largo plazo (MLP) guarda la cantidad de veces que se ha realizado cada movimiento y su función será evitar hacer muchas veces un mismo movimiento. La manera en la cual lo logramos consiste en penalizar restando una unidad a los valores de las soluciones vecinas a las cuales se llegue con algún movimiento que ya se haya realizado cuatro o más veces a lo largo del algoritmo.

Tanto la mejor solución encontrada hasta el momento como su valor asociado son guardados en MSG.

El criterio de aspiración lo cumplen aquellos movimientos de la Lista Tabú que logren una mejoría respecto a la solución guardada en MSG, y aquellos que cumplan esto los guardamos en A. Además, denotamos con V a la vecindad de la solución actual, y con VR a la vecindad reducida.

En el siguiente ejemplo la metaheurística corre durante 14 iteraciones, y si en alguna de ellas el valor más alto de una solución en VR es compartido por más de una solución vecina, cambiaremos a alguna de ellas de manera aleatoria.



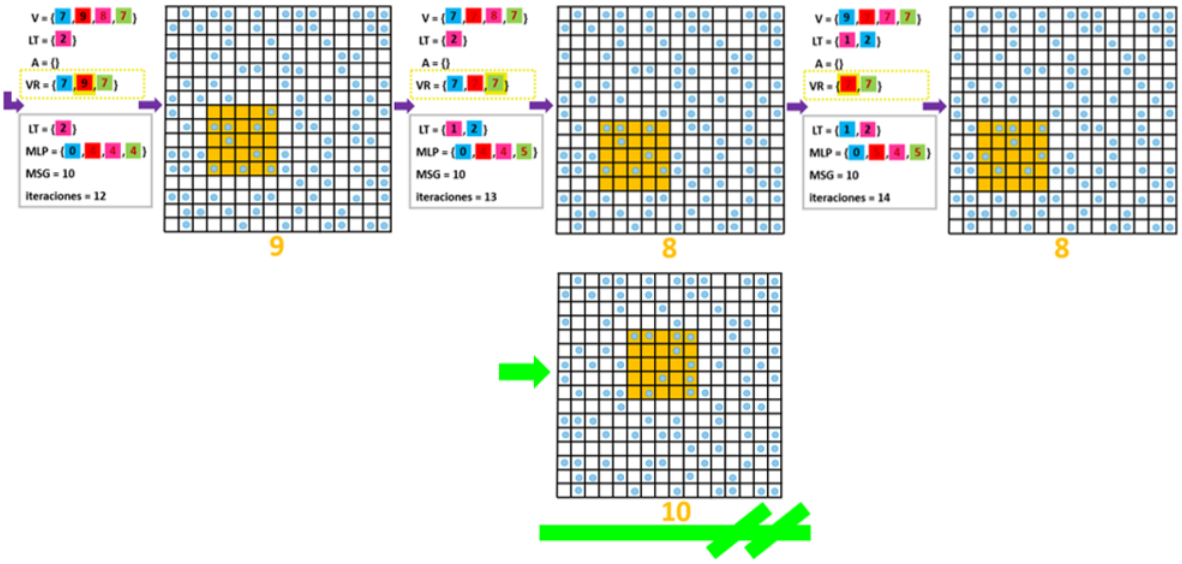


Figura 6.6: Ejemplo Búsqueda Tabú

En el ejemplo presentado empezamos en una solución aleatoria, calculamos su vecindad y guardamos la cantidad de círculos que contiene cada una de estas soluciones vecinas (en la imagen, cada iteración mostramos los círculos que contiene el realizar cada movimiento). Luego, vemos los elementos que estén en la Lista Tabú y consideramos aquellos que cumplan el criterio de aspiración para, posteriormente, construir la vecindad reducida, de la cual, una vez obtenida, elegiremos aquella solución que contenga más círculos, nos movemos a ella y actualizamos la solución. Posteriormente actualizamos la Lista Tabú, la memoria a largo plazo, la mejor solución encontrada hasta el momento y la cantidad de iteraciones que llevamos hasta tal punto (en el ejemplo esto se muestra en los cuadros contorno color gris). Todo este procedimiento descrito es repetido hasta haber realizado 14 iteraciones.

El cuadro que devuelve la metaheurística es el mejor encontrado de manera global. En el ejemplo es el que se obtuvo en la tercera iteración, cuyo valor es 10.

6.4.2. RBP: Implementación Búsqueda Tabú

6.4.2.1. Consideraciones para Búsqueda Tabú

Implementamos Búsqueda Tabú de dos formas, las cuales varían en cómo trabajan con la memoria.

La memoria a corto plazo trabaja de la siguiente forma: la Lista Tabú guarda la modificación contraria a la que se aplicó para así intentar evitar regresar a la solución anterior, pero además guarda la modificación realizada, aunque durante menos iteraciones. Por ejemplo, si el movimiento fue achicar el tercer rectángulo de arriba hacia abajo, los movimientos que van a la Lista Tabú son el de agrandar de abajo hacia arriba el tercer rectángulo, y, aunque por menos iteraciones, el de achicar el tercer rectángulo de arriba hacia abajo (al inicio de la heurística se le da una etiqueta a cada rectángulo que conforma la manta inicial para poder identificarlos individualmente). Estas modificaciones se vuelven Tabú durante un número definido de iteraciones, pero pueden usarse antes si al aplicarlas cumplen el criterio de aspiración.

El criterio de aspiración consiste en aplicar a la solución actual la modificación que se encuentra en la Lista Tabú consiguiendo obtener un mejor resultado que la mejor solución que se había encontrado hasta ese momento, entonces la modificación sale de la Lista Tabú y puede usarse si es que no hay otra modificación que logre dar mejores resultados.

La memoria a largo plazo guarda durante un pequeño periodo de tiempo modificaciones contrarias a las realizadas, también guarda la cantidad de veces que se ha realizado una modificación, pero esto no va a tener un efecto sobre esta modificación, sino sobre la contraria. Así, por ejemplo, si se ha realizado al tercer rectángulo de la manta la modificación achicar de arriba hacia abajo un total de quince veces y posteriormente se realiza esta modificación una vez más, entonces la memoria a largo plazo le asignará el número quince más uno a la modificación agrandar el tercer rectángulo de la manta de abajo hacia arriba. Con esto conseguimos conocer la cantidad de veces que se ha realizado a cada rectángulo cada modificación y que este valor afecte a la modificación contraria al mismo rectángulo, y si este número resulta ser mayor que el propuesto a una cota superior (el cual se define al inicio de la heurística), entonces se va a penalizar este movimiento en iteraciones sucesivas, es decir, a partir de haber superado la cota, y mientras mayor sea su número asignado, mayor será la penalización.

Ahora consideremos la memoria a corto plazo y el criterio de aspiración antes mencionados y modifiquemos la memoria a largo plazo. Esta última lo que hace es guardar a lo largo de la heurística algunas de las mantas que se obtuvieron con el fin de regresar a ellas una vez finalizada la Búsqueda Tabú, pues tiene por propósito diversificar la búsqueda en el espacio de soluciones. Para lograrlo, realizamos lo que sigue a cada una de las mantas que fueron guardadas: primero, ordenamos de mejor a peor las soluciones de su entorno y escogemos la i-ésima mejor solución, con i >1 (por ejemplo, la décima mejor solución de un entorno que cuente con 30 soluciones). Luego, al movernos, volvemos a repetir este procedimiento un número fijo de veces. Finalmente, a la manta que hayamos obtenido después de estas iteraciones, le aplicamos Búsqueda Tabú pero ya sin usar la memoria a largo plazo. El objetivo que tiene lo anterior es tratar de cambiar la ruta usada en el espacio de soluciones para intentar llegar a una mejor solución que la encontrada anteriormente.

La manta que arroja como resultado esta heurística será la mejor encontrada en todo el proceso.

Nota: Para ambos casos, al momento de construir la Vecindad Reducida ((entorno de la solución - Lista Tabú) + soluciones que cumplen el criterio de aspiración) obtenemos un conjunto de soluciones con los cuales podemos trabajar. En Búsqueda Tabú elegimos movernos a la mejor manta que se encuentre en esta vecindad reducida, pero hay ocasiones en las que existe más de una solución en este entorno que al ser evaluadas en la función objetivo consiguen obtener el mismo valor. Cuando esto pasa, como las soluciones de la vecindad están guardadas en una lista y vamos moviéndonos en ella de manera ordenada, al momento de llegar a una de estas mejores soluciones la guardamos temporalmente. Después, seguimos comparando ordenadamente con las demás soluciones de la lista hasta llegar a una que tenga el mismo valor que la guardada, y vamos a cambiar de solución a esta otra si cumple un criterio de probabilidad (por ejemplo, si $r < 0.5$, con $r \in [0,1]$ un número aleatorio). Una vez aplicado este criterio, seguimos moviéndonos ordenadamente en esta lista. Cuando hayamos finalizado de comparar las soluciones de la lista, la manta que haya quedado en la memoria temporal será la solución a la que nos moveremos.

6.4.2.2. Valores dados a los parámetros en Búsqueda Tabú

Parámetros cuando la memoria a largo plazo guarda la cantidad total de veces que se ha realizado cada modificación:

- $iteraciones = 1800$: indica el número de iteraciones que se realiza Búsqueda Tabú.
- q: número de iteraciones que permanece en la Lista Tabú la modificación contraria a la aplicada a determinado rectángulo. El valor que toma es:
  $q = \begin{cases} 12 & \text{si } k \geq 3 \\ 2 & \text{si } k < 3 \end{cases}$
- $q_{alt} = \lceil \frac{q}{6} \rceil$: número de iteraciones que permanece en la Lista Tabú la modificación aplicada a determinado rectángulo.
- $iterPenalizo = \lceil \frac{iteraciones}{30} \rceil$: es la cantidad a partir de la cual empieza la penalización de un movimiento. De otra forma, indica el número de iteraciones que puede realizarse una modificación antes de tener penalizaciones, pues una vez rebase esta cantidad, el valor de la manta que se encuentra en la vecindad al aplicar esta modificación será penalizado.
- $probabCambio = 0.38$: Al tener la Vecindad Reducida, las soluciones dentro de ella las tenemos en forma de lista. Al ir comparando de solución en solución vamos guardando temporalmente la solución de la vecindad con el mejor valor encontrado hasta ese momento. Así, puede ocurrir que las soluciones que se estén comparando tengan el mismo valor. Entonces, este parámetro indica la probabilidad de cambiar de la solución guardada temporalmente a la que se esté comparando cuando la situación mencionada ocurra. Esto ayuda a diversificar la búsqueda y evitar ciclos.
- $penalizacion = \lceil \frac{\#aplicacionModificacion}{4} \rceil$: aquí $\#aplicacionModificacion$ es el total de veces que se ha aplicado una modificación. Este parámetro sirve para evitar emplear modificaciones que ya se hayan usado mucho para intentar diversificar la búsqueda. Las soluciones de una vecindad que hagan uso de una modificación que haya sido utilizada más de $iterPenalizo$ veces serán penalizadas. Así, los valores de tales soluciones dependerán tanto del valor obtenido al evaluar la solución misma en la función objetivo como de la cantidad de veces que se haya usado la modificación, pues su nuevo valor (digamos penalizando una solución B) se calculará de la siguiente forma: $valor(B) = valor(B) + penalizacion$.

Parámetros cuando la memoria a largo plazo guarda algunas mantas por las que pasó:

- $iteraciones = 1000$
- $q = \begin{cases} 12 & \text{si } k \geq 3 \\ 2 & \text{si } k < 3 \end{cases}$
- $q_{alt} = \lceil \frac{q}{6} \rceil$
- $probabCambio = 0.38$
- $mantaIteracion = [150\ 250\ 400\ 600\ 800]$: vector cuyas entradas indican que la manta obtenida en tales iteraciones serán guardadas para ser usadas después.

- $numVecino = \lceil \frac{k*8}{5} \rceil$: ($k$ es el número de rectángulos que tiene una Manta). Al terminar Búsqueda Tabú tenemos una lista donde hemos guardado algunas soluciones (las encontradas en las iteraciones indicadas en el vector *mantaIteracion*) y se trabajará individualmente con cada una. Primero vemos la vecindad, luego la ordenamos de la mejor a la peor solución y posteriormente elegimos una solución dentro de este entorno a la cual nos moveremos. Así, este parámetro indica la solución a la que nos cambiaremos, esto es, nos moveremos a la *numVecino-ésima* mejor solución de la vecindad.

- $TMOiter = 200$: Es la cantidad de iteraciones que elegiremos la *numVecino-ésima* mejor solución de la vecindad.

- $iteracionesRep = 500$: Una vez realizadas las $TMOiter$ iteraciones a cada solución guardada (indicadas por *mantaIteracion*) obtenemos nuevas soluciones. A estas les aplicaremos Búsqueda Tabú (a cada una por separado) la cantidad de iteraciones indicada por este parámetro (la manta final será la mejor encontrada globalmente en toda la heurística).

6.5. Algoritmos Genéticos

Algoritmos Genéticos es una metaheurística que nace en alusión a conocimientos de la biología. Se basa en un proceso de selección y adaptación, tomando como referencia la teoría evolutiva, la selección natural y la genética. Fueron desarrollados por John Holland en 1975.

De manera introductoria, para poder dar contexto biológico en lo que se refiere, dejo a continuación un fragmento escrito por David Beasley, David R. Bull y Ralph R. Martin en [2] donde explican con suma claridad la idea biológica de estos algoritmos:

"En la naturaleza los individuos de una población compiten entre ellos en la búsqueda de recursos tales como comida, agua y refugio. Además, los miembros de una misma especie compiten a menudo en la búsqueda de un compañero. Aquellos individuos que tienen más éxito en sobrevivir y en atraer compañeros tendrán muy probablemente un mayor número de descendientes. Individuos poco dotados producirán pocos o incluso ningún descendiente. Esto significa que los genes de los individuos mejor adaptados se propagarán en cada generación sucesiva hacia un número creciente de individuos. La combinación de buenas características provenientes de diferentes ancestros puede a veces producir descendientes "superadaptados", cuya adaptación es mayor que la de cualquiera de sus ancestros. De esta manera, las especies evolucionan para mejorar cada vez más su nivel de adaptación en el entorno donde viven."

Antes de empezar a explicar en qué consisten, necesitamos establecer la terminología usada en estos algoritmos. Tal como se indica en [17], los términos que se toman de la biología para su uso en algoritmos genéticos se utilizan solamente como analogía con lo que ocurre en la verdadera biología. Así, debe quedar claro que no se toman las definiciones, que a continuación presentaremos, de manera literal de la biología, sino solamente se busca que den contexto y ejemplifiquen el código, pues intentan parecerse, en lo posible, a las definiciones correspondientes en el área de biología pero adaptándolos para estos algoritmos.

Una **población** la vamos a definir como el conjunto de todas las soluciones en una determinada generación. Un **individuo** es una solución de una población determinada. Definimos como **cromosoma** al conjunto de características que componen y definen a un individuo. **Gen**

es cada una de estas características. Finalmente, **alelo** es el valor que toma un gen de un cromosoma particular, es decir, el valor que toma un alelo muestra si esa característica se presenta o no en el individuo. Estas definiciones quedan ilustradas en la Figura 6.7.

Además, tenemos las siguientes dos definiciones en términos de los Algoritmos Genéticos: **genotipo** es la población en el espacio computacional. En este espacio, las soluciones se representan de tal manera que sea sencillo entender y manipular la información usando un sistema de cómputo; **fenotipo** es la expresión del genotipo, es decir, la solución en el mundo real del problema.

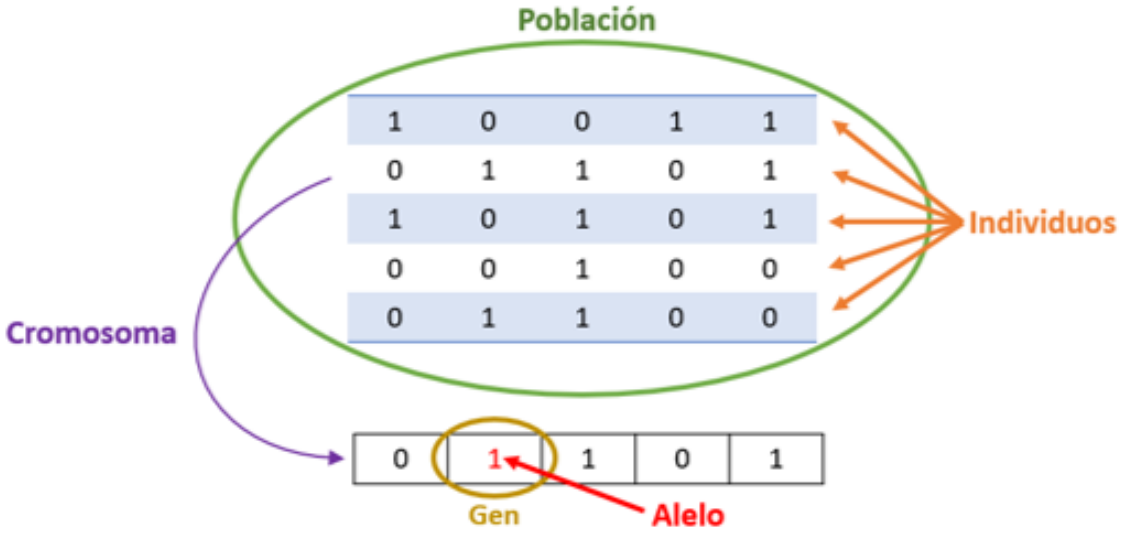


Figura 6.7: Algoritmos Genéticos: visualización conceptos

Para iniciar el algoritmo se elige un conjunto de soluciones al que llamaremos población inicial, la cual consiste en una alta cantidad de soluciones encontradas de manera aleatoria. La aleatoridedad es esencial en este inicio, pues permitirá obtener diversidad en las soluciones y en consecuencia explorar de manera más amplia el espacio de soluciones. De esta población se realiza un proceso de selección que consiste en definir las parejas que van a cruzarse.

Posteriormente, estos pasan por un proceso de apareamiento con lo cual dan lugar a soluciones hijas, donde estos descendientes conservan las mejores características de sus padres, pero algunos de ellos pueden sufrir cambios o mutaciones en su material genético, lo que permitirá al algoritmo realizar una búsqueda más robusta que explore más allá de óptimos locales. Déspués, las soluciones obtenidas definen una nueva población con la que se repetirá este procedimiento hasta que se cumpla cierta condición, como detenerse hasta llegar a cierto número de generaciones o hasta que las soluciones cumplan algún requisito. De esta manera, en cada paso, si bien es probable que la solución más sobresaliente de una generación sea superior que la mejor de la siguiente generación, de manera global las generaciones van mejorando. Esto se observa calculando para cada generación la suma del valor de todas las soluciones de la generación dividida entre la cantidad de soluciones. De otra forma, con el paso del tiempo las generaciones van perfeccionándose, pues van guardando y presentando las mejores características de sus antepasados.

Es importante mencionar que para poder comparar a los individuos de una población se usa una función de aptitud. Esta puede consistir en la misma función objetivo o también darle mayor peso a la presencia de ciertas características en las soluciones.

En esta metaheurística se hace uso de operadores genéticos, los cuales tienen como objetivo definir, modificar y afectar la composición genética de los hijos. Estos operadores son **selección**, **cruzamiento** y **mutación**.

El objetivo de la selección es elegir a los padres que dejarán descendencia. Hay varios tipos de selección, por lo que nombraremos tres de ellos:

- **Método de la ruleta.** Esta técnica fue propuesta por DeJong (1975). La idea es asignar mayor probabilidad de ser elegido a los individuos que presenten mejores características. Esta se encuentra dividiendo el valor de aptitud del individuo (por ejemplo la evaluación de la solución en la función objetivo) entre la suma de los valores de aptitud de todos los individuos[3]. Este método permite elegir a un individuo más de una vez o no elegirlo, dependiendo de su probabilidad de ser escogido. Así, podremos construir una población del mismo tamaño que la anterior. Además, permite mayor probabilidad que en la siguiente generación prevalezcan las características más sobresalientes de la anterior generación.

- **Selección por torneo**. Consiste en formar parejas de manera aleatoria y de cada pareja seleccionar al individuo que presente mejores características o cuya evaluación en la función objetivo sea mejor. Así, de cada pareja se escoge como padre a uno de ellos y al otro no se le permite dejar descendencia. Con esta forma de seleccionar padres pueden prevalecer los mejores rasgos de cada generación, pero perdemos diversidad de soluciones, pues limita el espacio de búsqueda de soluciones.

- **Selección proporcional**. Lo que hace es asignar la misma probabilidad de selección a todos los individuos, es decir, si hay n individuos en la población, la probabilidad de cada uno de ser escogido es $\frac{1}{n}$. Esta forma de seleccionar padres puede no ser muy efectiva, pues en cada generación pueden desaparecer las mejores características y pasar a la siguiente las peores, aunque lo que se gana con este método es diversificar la búsqueda.

De manera general, estas formas de selección buscan ponderar a los individuos, de tal manera que se asigne cierta probabilidad de selección a cada uno de ellos.

El cruzamiento es el operador que define el material genético que aportará cada padre a cada uno de los hijos, pues dos padres pueden dar lugar a más de un hijo, aunque normalmente se decide que se obtengan dos hijos por cada dos padres para que la cantidad de individuos en la población permanezca constante en cada generación. Este cruzamiento puede ser sexual o asexual.

[3]Nota: esta forma de asignar probabilidad funciona cuando se busca maximizar la función objetivo, pero no cuando se quiere minimizar. Para este último caso, puede realizarse la asignación de probabilidad a cada solución como sigue: a la mejor solución (aquella con la que se obtenga el valor más pequeño) se le asigna el resultado de dividir el peor valor (valor más grande) entre la suma de todos los valores. A la segunda mejor solución se le asigna el resultado de dividir el segundo peor valor entre la suma de todos los valores y así sucesivamente hasta asignar a la peor solución el resultado de dividir el mejor valor (valor más pequeño) entre la suma de todos los valores.

Si se elige un cruzamiento sexual, hay varios métodos:

- **Soluciones Discretas.** Las soluciones discretas consisten en asignar a las características un uno si se presentan y un cero si no se presentan, es decir en forma de bit. Mientras más bits tenga un cromosoma, mayor diversidad de búsqueda, aunque esto varía en cada problema. [4]

    - **Cruce de un punto**. Se toman las cadenas de bits de dos padres. Luego, se divide cada uno en dos partes iguales, eligiendo para ambas un mismo punto de corte, y lo único que se realiza es combinar ambos obteniendo así dos hijos. Ejemplo:
      Solucion1 = A:B:C:D:E
      Solucion2 = 1:2:3:4:5
      Punto de corte aleatorio:
      Solucion1 = A:B|C:D:E
      Solucion2 = 1:2|3:4:5
      Soluciones hijas:
      Hijo1 = A:B:3:4:5
      Hijo2 = 1:2:C:D:E.
    - **Cruce de dos puntos**. En lugar de cortar en un punto, se corta en dos y se combina el material genético. Ejemplo:
      Solucion1 = A:B:C:D:E
      Solucion2 = 1:2:3:4:5
      Puntos de corte aleatorios:
      Solucion1 = A:B|C:D|E
      Solucion2 = 1:2|3:4|5
      Soluciones hijas:
      Hijo1 = A:B:3:4:E
      Hijo2 = 1:2:C:D:5.
    - **Cruce uniforme**. En este cruce se pueden construir muchos hijos. Para construir uno, se realiza el siguiente procedimiento: Se toman las cadenas de dos padres, digamos de n bits cada uno. Luego, se le asigna la probabilidad p de que el primer bit tenga el alelo del primer padre y probabilidad $n - p$ que presente el alelo del segundo padre, y esto se realiza con cada bit.

- **Soluciones Continuas.** En este caso no podemos realizar los mismos métodos que se usan con soluciones discretas. Cuando esto ocurre, recurrimos a otros métodos, aunque solo explicaremos uno de ellos.

    - **Media.** Con esta forma de cruce sólo se obtiene un hijo. El procedimiento consiste en que el gen de la descendencia toma el valor medio de los genes de los padres ([13]).

[4] Para los ejemplos y las ideas de estos métodos de cruce nos basamos en [13].

En cambio, para un cruce asexual tenemos los siguientes métodos:

- **Copia**. Consiste en realizar la copia de uno o varios padres, es decir, seleccionar varios padres y que estos pasen directamente a la siguiente generación.
- **Elitismo**. Consiste en elegir a los mejores padres (o si se ve por pares, al mejor de ambos) y que pasen directamente a la siguiente generación. Este es un caso particular del operador copia ([13]).

El operador de mutación realiza cambios al material genético de algunos hijos. Esto se realiza para que algunas soluciones puedan estar definidas por nuevas características que no se hayan presentado antes y como consecuencia realice una búsqueda más profunda en el espacio de soluciones.

A cada hijo obtenido se le asigna una probabilidad de mutar (normalmente muy pequeña). Los individuos que muten pueden cambiar uno o varios rasgos de sus características. En soluciones discretas solamente tenemos que cambiar en los bits elegidos el uno por cero y viceversa. En cambio, en soluciones continuas se tiene que definir otra forma de mutar, lo cual varía en cada problema. Una forma sencilla de realizar esta mutación que puede aplicarse en una cantidad considerable de problemas con soluciones continuas es solamente aumentar el valor de un alelo o disminuirlo.

En la Figura 6.8 se ilustra el procedimiento de algoritmos genéticos:

Figura 6.8: Procedimiento Algoritmos Genéticos

A continuación presentamos el pseudocódigo de Algoritmos Genéticos:

Algorithm 6 Algoritmos Genéticos

$n = CantidadPoblacionInicial$; # Número de soluciones que tendrá la población inicial
$p = ProbabilidadMutar$; # Probabilidad que tiene una solución de mutar
$Poblacion = FuncionSolucionAleatoria(n)$; # Esta primer población es la
# población inicial. Se encuentran de manera aleatoria n soluciones del problema
while criterio o condición de paro no se cumpla **do**
  $Padres = FuncionSeleccionDePadres(Poblacion)$; # Escoger por medio de un método
  # del operador de selección las soluciones de *Poblacion* que dejarán descendencia

  $Hijos = FuncionCruzamiento(Padres)$; # A través del operador de cruzamiento aplicado
  # al conjunto *Padres* se obtienen a los descendientes, los cuales serán guardados en un
  #nuevo conjunto de soluciones al que llamaremos $Hijos$

  $NuevaGeneracion = FuncionMutacion(Hijos, p)$; # Mediante el operador de mutación
  # se realiza una mutación a algunas soluciones del conjunto $Hijos$. Cada solución tiene
  # probabilidad p de mutar

  $Poblacion = NuevaGeneracion$; # Se redefine al conjunto *Poblacion* como el conjunto de
  # nuevas soluciones, o soluciones hijas, que componen la nueva generación en cuestión
end while

6.5.1. Ejemplo de aplicación de Algoritmos Genéticos

La dimensión del mallado en el cual puede ser colocado un cuadro es de 16×16 y la dimensión de un cuadro solución es de 5×5.

Un cuadro será codificado con un vector de dos entradas (a,b) el cual indicará la posición de su esquina superior izquierda dentro del mallado, donde el valor de a indica el renglón y b la columna donde tal esquina es posicionada. Así, el vector (a,b) define el cuadrado alineado a los ejes de 5×5 cuya esquina superior izquierda se encuentra en el renglón a, columna b; su esquina superior derecha en el renglón a, columna $b+4$; su esquina inferior izquierda en el renglón $a+4$, columna b y su esquina inferior derecha en el renglón $a+4$, columna $b+4$.

Por ejemplo, el vector $(3,4)$ codifica el cuadro cuya esquina superior izquierda se encuentra en el renglón 3, columna 4, y sólo esta información basta para construir totalmente el cuadrado asociado, pues como toda solución tiene dimensión 5×5, su esquina superior derecha se encontrará en el renglón 3, columna 8, su esquina inferior izquierda en el renglón 7, columna 4 y su esquina inferior derecha en el renglón 7, columna 8. En la Figura 6.9 se muestra el cuadrado en cuestión:

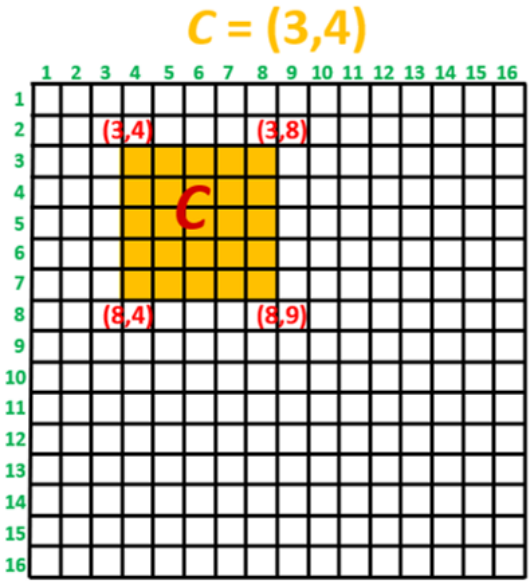


Figura 6.9: Representación de la solución $C = (3,4)$

Para que un cuadro cumpla la restricción de tener dimensión 5×5 y estar dentro del mallado (la condición de estar alineado a los ejes se cumple con la construcción descrita a partir de un vector de dos entradas dado) debemos notar que debemos restringir los valores que toman las entradas del vector (a,b) y esto lo logramos con las siguientes restricciones:

$$1 \leq a \leq 12,\ 1 \leq b \leq 12,\ a,b \in \mathbb{N}$$

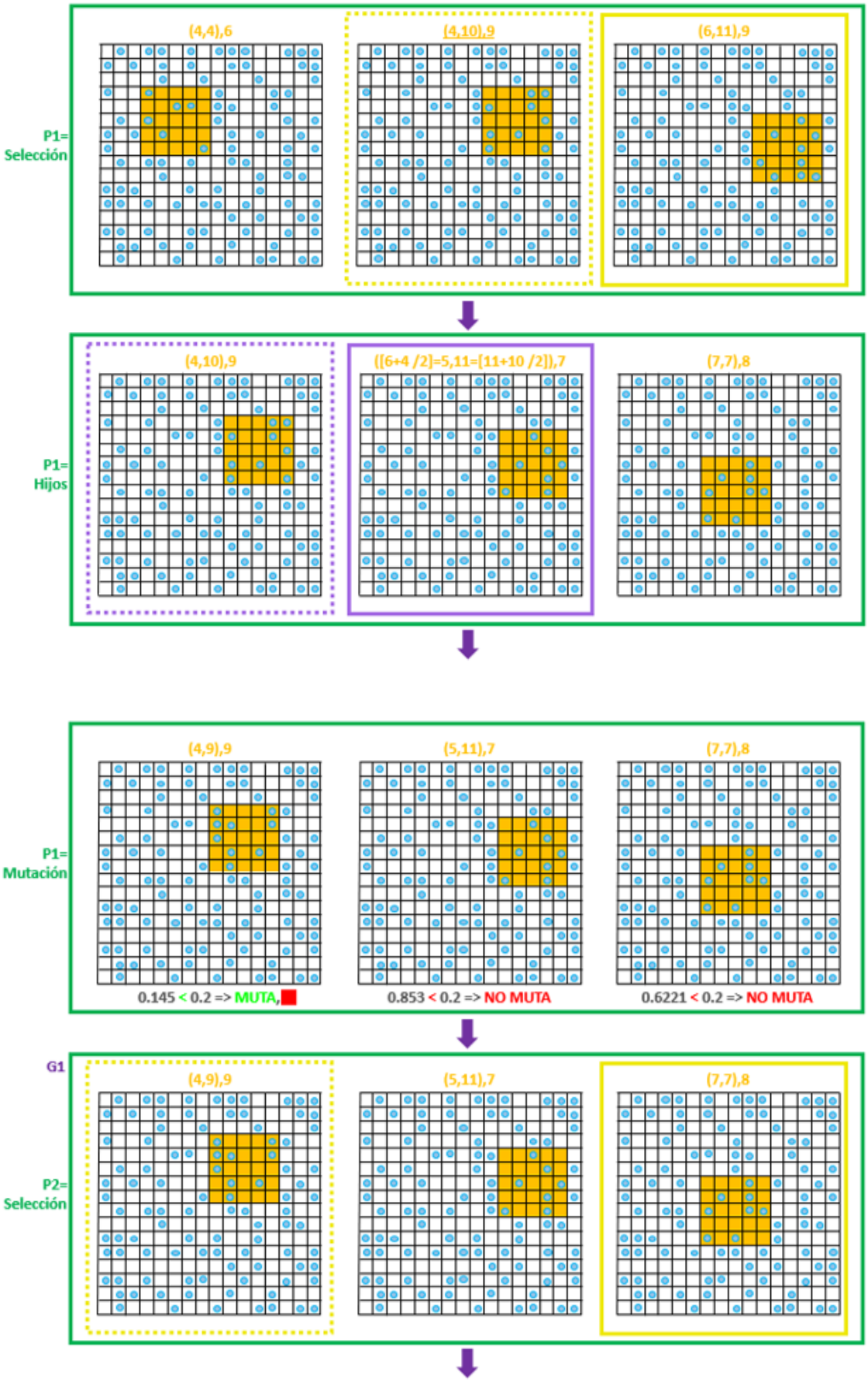
Con lo anterior en mente, para el ejemplo la población en cada generación constará de 3 individuos. En la primera generación se encontrarán de manera aleatoria, y para las siguientes:

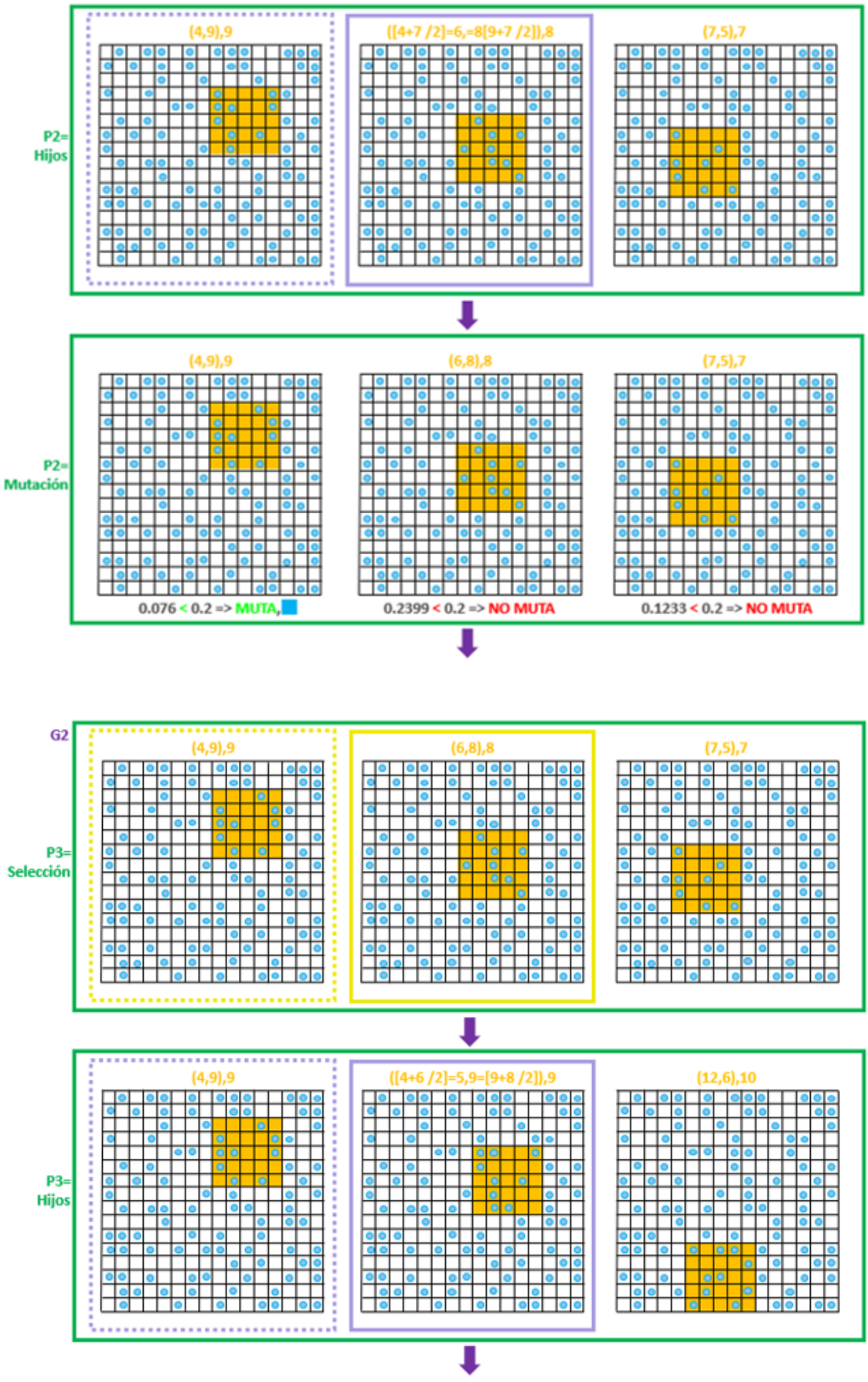
Primero se escoge de la población las dos mejores soluciones (selección por torneo por tercias), digamos (a,b) y (c,d), de las cuales se obtendrá una primer solución hija con vector asociado $([\frac{a+c}{2}],[\frac{b+d}{2}])$ donde $[\cdot]$ redondea al entero más cercano, con tipo de cruce: solución continua, *Media*. Después una segunda solución será el mejor padre entre (a,b) y (c,d), con el tipo de cruce: solución continua, *Elitismo*, y una tercera solución se generará de manera aleatoria. Así, conseguiremos una nueva población de tres individuos para la siguiente generación, y este proceso se repetirá sucesivamente hasta llegar, para este ejemplo, a la 5^{ta} generación.

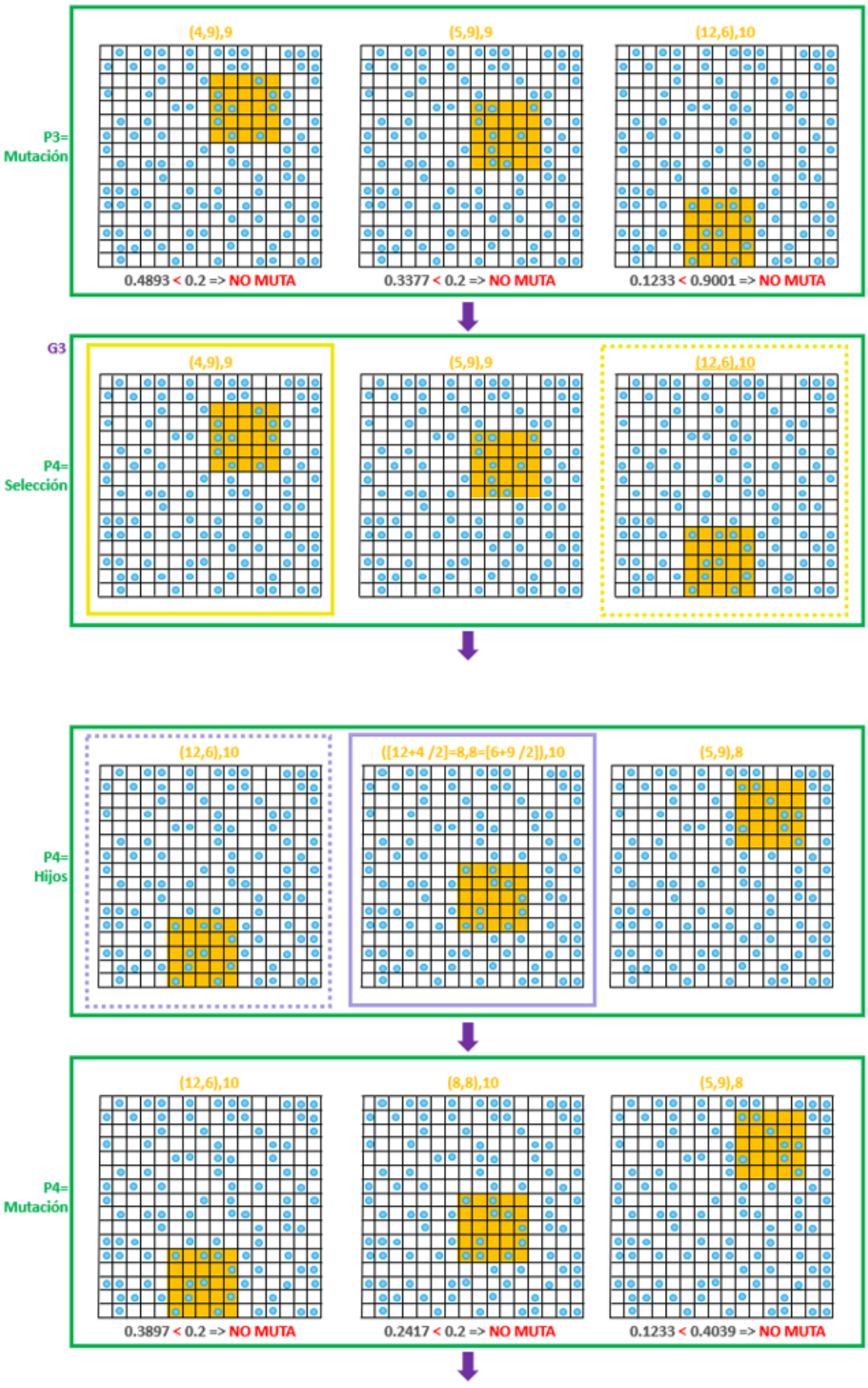
Ahora, si tenemos la población $P = \{(a,b),(c,d),(e,f)\}$, los tres individuos que la conforman son (a,b), (c,d) y (e,f). De esta población el cromosoma del individuo (e,f) es $\{e,f\}$, el cual tiene dos genes: **primer entrada del vector** y **segunda entrada del vector**, y finalmente el alelo es el valor que toma un gen del individuo, que para este mismo individuo, por ejemplo, el alelo de **primera entrada del vector** es e y el alelo de **segunda entrada del vector** es f.

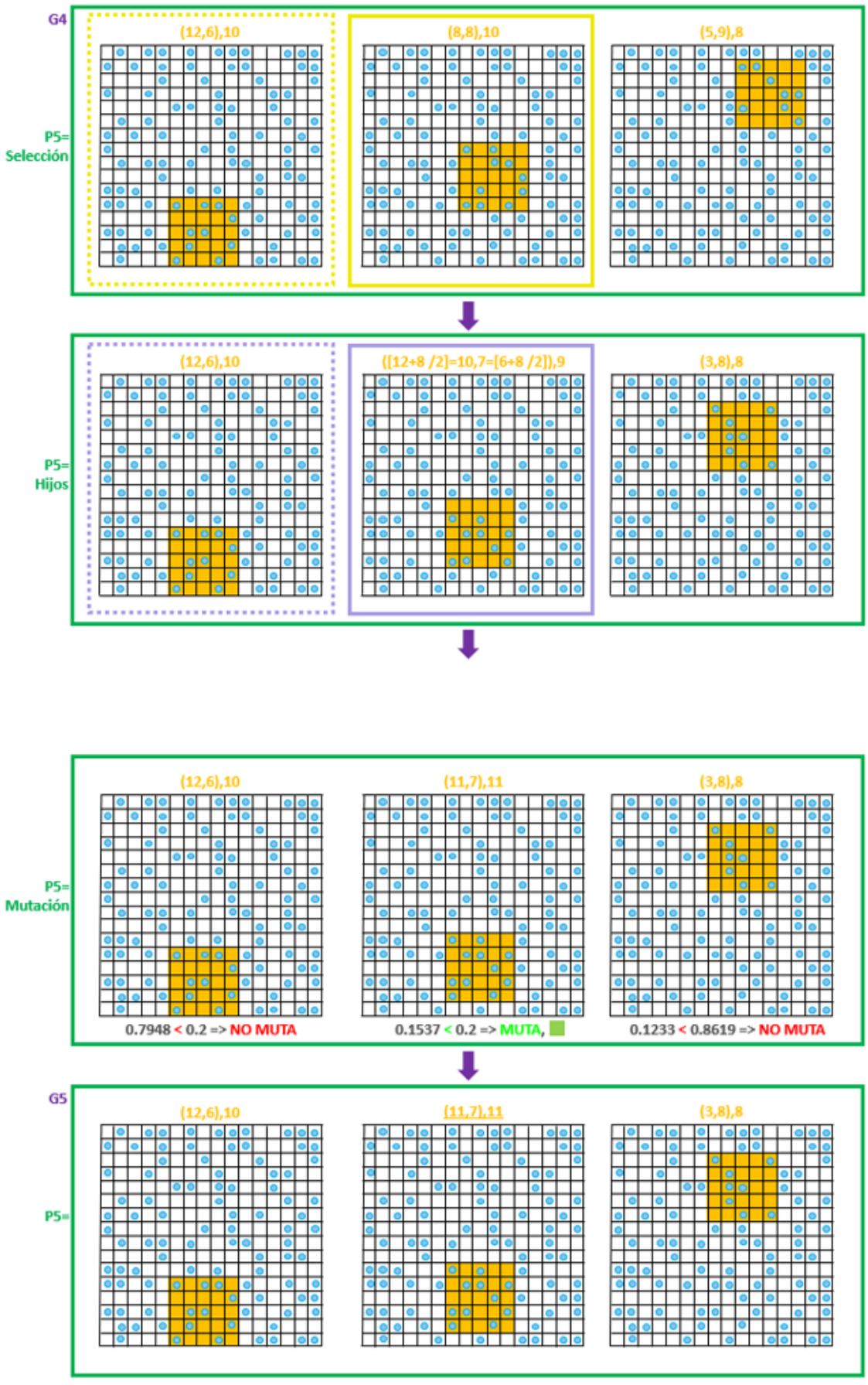
El genotipo de una solución cuya esquina superior izquierda tiene coordenadas (e,f), su esquina superior derecha coordenadas $(e,f+4)$, su esquina inferior izquierda $(e+4,f)$ y su esquina inferior derecha $(e+4,f+4)$ es simplemente el vector (e,f) y el fenotipo de su solución asociada es la representación visual del cuadrado descrito.

Cada nuevo individuo tendrá probabilidad 0.2 de mutar, y e3ta consistirá en realizar únicamente un movimiento de manera aleatoria, ya sea moverse hacia arriba, hacia abajo, a la izquierda o a la derecha.


P1=
Selección
(4,4),6
(4,10),9
(6,11),9
P1=
Hijos
(4,10),9
([6+4 /2]=5,11=[11+10 /2]),7
(7,7),8
P1=
Mutación
(4,9),9
(5,11),7
(7,7),8
0.145 < 0.2 => MUTA,
0.853 < 0.2 => NO MUTA
0.6221 < 0.2 => NO MUTA
G1
P2=
Selección
(4,9),9
(5,11),7
(7,7),8


P2=
Hijos
(4,9),9
([4+7 /2]=6,=8[9+7 /2]),8
(7,5),7
P2=
Mutación
(4,9),9
(6,8),8
(7,5),7
0.076 < 0.2 => MUTA,
0.2399 < 0.2 => NO MUTA
0.1233 < 0.2 => NO MUTA
G2
P3=
Selección
(4,9),9
(6,8),8
(7,5),7
P3=
Hijos
(4,9),9
([4+6 /2]=5,9=[9+8 /2]),9
(12,6),10


P3=
Mutación
(4,9),9
(5,9),9
(12,6),10
0.4893 < 0.2 => NO MUTA
0.3377 < 0.2 => NO MUTA
0.1233 < 0.9001 => NO MUTA
G3
P4=
Selección
(4,9),9
(5,9),9
(12,6),10
P4=
Hijos
(12,6),10
([12+4 /2]=8,8=[6+9 /2]),10
(5,9),8
P4=
Mutación
(12,6),10
(8,8),10
(5,9),8
0.3897 < 0.2 => NO MUTA
0.2417 < 0.2 => NO MUTA
0.1233 < 0.4039 => NO MUTA


G4
P5=
Selección
(12,6),10
(8,8),10
(5,9),8
P5=
Hijos
(12,6),10
([12+8 /2]=10,7=[6+8 /2]),9
(3,8),8
P5=
Mutación
(12,6),10
(11,7),11
(3,8),8
0.7948 < 0.2 => NO MUTA
0.1537 < 0.2 => MUTA,
0.1233 < 0.8619 => NO MUTA
G5
P5=
(12,6),10
(11,7),11
(3,8),8

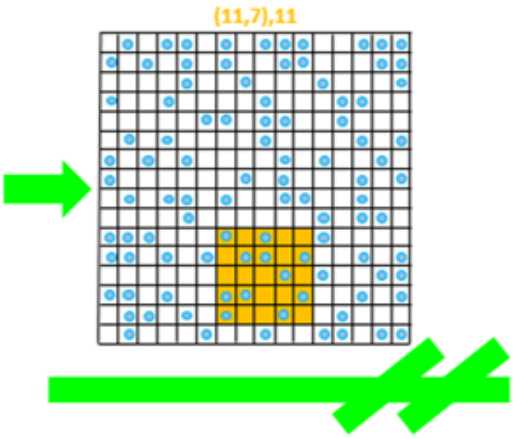


Figura 6.10: Ejemplo Algoritmos Genéticos

En la sucesión de imágenes empezamos con una población $P1$ con 3 individuos encontrados de manera aleatoria. De ellos, se escogen los dos mejores padres (encerrados con cuadro dorado) de los cuales se construye un hijo (encerrado cuadro morado sólido) y entre ambos padres se hace una copia del mejor (el mejor es aquél que se encuentra encerrado en cuadro dorado con contorno punteado) y pasa como solución hija (cuadro morado con contorno punteado), y por último una última solución hija se encuentra de manera aleatoria. Posetriormente cada solución tiene propbabilidad del $2\,\%$ de mutar, y aquellos que mutan se les aplica de manera aletoria un movimiento (cuadro azul implica movimiento hacia arriba, rojo a la izquierda, rosa a la derecha y verde hacia abajo). Una vez terminado el procedimiento de mutación, las soluciones obtenidas definen la población de una nueva generación (Gi) y este procedimiento se repite hasta llegar a la 5^{ta} generación $G5$ (igual a la 6^{ta} población $P6$), y la solución que devuelve el algoritmo una vez finalizado es la mejor encontrada en todas las poblaciones Pi, que en este ejemplo es un cuadro cuyo genotipo es $(11,7)$ el cual contiene 11 círculos.

6.5.2. RBP: Implementación Algoritmos Genéticos

6.5.2.1. Consideraciones para Algoritmos Genéticos

El **operador de selección** que se implementó fue el método de la ruleta.
Para el **operador de cruzamiento** tenemos dos casos:

- $k = 1$: En este caso lo que se hace para obtener dos soluciones es: como tenemos dos padres cada uno con un rectángulo, construir a partir de ellos una solución hija que se encuentre a la mitad de distancia de ambos, y la base y altura que le daremos será el promedio de las bases y alturas de los padres. La segunda se obtiene al comparar a los padres, de los cuales pasará a la siguiente generación aquél que sea mejor solución.

- $k > 1$: Ahora tenemos dos padres cada uno con k rectángulos. Primero guardamos los rectángulos de ambos en una misma lista, por lo que esta constará de $2k$ rectángulos. Luego, evaluamos cada uno de ellos como si fueran una manta de un solo rectángulo en la función objetivo. Esto lo hacemos para ver qué tan buenos rectángulos son individualmente. Después, ordenamos la lista del mejor al peor rectángulo. Posteriormente, eliminamos algunos de los peores rectángulos y acortamos la lista con la restricción de

que esta cuente con al menos k buenos rectángulos (por ejemplo, si $k = 14$ la lista inicial constará de 28 rectángulos, y tras ordenarlos decidimos quitarle los 8 peores para así quedarnos con una lista ordenada de tamaño $20 > k$). Una vez realizado lo anterior, intentamos construir una nueva solución de la siguiente manera: iniciamos eligiendo el primer rectángulo de la lista como solución de partida en la construcción. Seguido de ello, tomamos el segundo elemento de la lista, y si al agregar este rectángulo a la solución de partida resulta en una manta factible, lo añadimos a la misma, la cual ahora sería una manta con dos rectángulos, pero en el caso de que esto no suceda no lo agregamos. Posteriormente pasamos con el siguiente rectángulo de la lista para realizar este mismo procedimiento descrito, y seguimos así hasta lograr la construcción de una manta factible que tenga k rectángulos. El objetivo es conseguir una buena solución factible.

Ahora, puede pasar que con el proceso explicado no logremos obtener una nueva solución debido a que nos agotemos los rectángulos de la lista al no conseguir armar una solución factible. Cuando esto ocurra, lo que haremos será una selección por torneo entre los dos padres, es decir, pasa a la siguiente generación el mejor padre.

A la manta que obtengamos de lo anterior, ya sea por el cruce descrito o por el torneo, le llamaremos *Hijo 1*. Una segunda manta será encontrada de manera aleatoria y le llamaremos *Hijo 2*. Así, por cada par de padres tendremos dos hijos, y todos estos nuevos individuos formarán una nueva población.

Finalmente, cada uno de ellos podrá mutar con cierta probabilidad. Es aquí donde entra el **operador de mutación**, el cual consiste en aplicar un número fijo de modificaciones a la manta. Las modificaciones que se apliquen se escogerán de manera aleatoria, y un mismo rectángulo podrá sufrir una misma modificación más de una vez. La única restricción es que con cada mutación sigamos teniendo una manta factible. En caso de que esto no suceda, no aplicaremos tal modificación y seguiremos mutándole si es que aún faltan modificaciones que aplicar.

6.5.2.2. Valores dados a los parámetros en Algoritmos Genéticos

En esta heurística contamos con los paramétros que siguen:

- $CantidadPoblacion = 140$: es el número de individuos con los que contará la población y esta cantidad permanece constante a lo largo de las generaciones.
- $Generaciones = 3000$: es la cantidad de generaciones que deseamos obtener.
- $ProbabilidadMutacion = 0.15$: es la probabilidad de que un individuo mute.
- $MaxRectDetener = k + \lceil \frac{k}{4} \rceil$: ($k$ el número de rectángulos de una manta). Recordemos al construir una de la soluciones hijas lo primero que hacemos es ordenar los rectángulos de ambos padres en una lista del mejor al peor (tomando cada uno como una manta de un sólo rectángulo). Este parámetro lo que hace es quedarse de esa lista con los $MaxRectDetener$ mejores elementos y no tomar en cuenta los demás. Así, nos quedaremos para intentar la construcción de una solución hija con aproximadamente los k mejores rectángulos más la sexta parte de k mejores rectángulos que siguen.
- $g = \lceil \frac{k}{4} \rceil$: Recordemos cada hijo tiene cierta probabilidad de mutar. La mutación consiste en elegir y aplicar g modificaciones (pueden repetirse) de manera aleatoria a la solución.

Capítulo 7

Análisis de resultados

A continuación compararemos las heurísticas entre sí para poder determinar, entre ellas, cuál da mejores resultados de manera consistente, sustentando lo recabado por medio de pruebas estadísticas.

Para ello, con antelación, mediante una extenuante experimentación computacional, definimos los valores de los parámetros de cada heurística[1].

Todas las heurísticas fueron probadas para 12 instancias distintas y para cada una trabajamos con valores distintos de k, donde k indica la cantidad máxima de rectángulos que puede tener una Manta, los cuales fueron: k = 10, 20, 40, 60 y 80 rectángulos.

Para cada uno de los valores de k se emplearon 8 distintas soluciones iniciales, y con el fin de comparar los resultados de la manera más fiable, para obtener cada una se empleó una semilla definida, con lo cual se asegura que todas las heurísticas coincidan en las soluciones iniciales, exceptuando Algoritmos Genéticos, la cual únicamente coincidirá en las semillas empleadas, pues por la naturaleza de su implementación esta empieza con 140 soluciones iniciales construidas de forma aleatoria.

De esta manera, compararemos cada par de heurísticas con el fin de concluir, en la medida de lo posible y con sustento estadístico, cuál obtiene mejores resultados consistentemente para distintos valores de k, pues podría ser posible que una de ellas funcione mejor cuando el valor de k es más pequeño, pero no sea la mejor opción cuando k tome valores altos. Así, en cada comparación y para tres valores de k (10, 40 y 80) mostraremos de tres instancias distintas la comparación de sus diagramas de caja para poder conjeturar cuál podría ser mejor heurística.

Posteriormente, veremos si la conjetura realizada es estadísticamente significativa para cada valor de k (10, 20, 40, 60 y 80) y para cada instancia por medio de la prueba U de Mann-Whitney (también conocida como prueba de suma de rango de Wilcoxon). Decidimos emplear esta prueba no paramétrica puesto que desconocemos la distribución de los datos, además que no contamos con una gran cantidad de valores. Esta prueba nos servirá para comparar dos grupos de resultados y determinar si hay diferencia significativa entre ambos o si no hay evidencia suficiente para concluir lo anterior. Los datos que obtuvimos cumplen con los supuestos para poder emplear la prueba, pues son independientes, no relacionados y son variables cuantitativas.

[1]Los valores de los parámetros propios de cada heurística fueron dados, y los podrá encontrar, en los capítulos de su respectiva heurística.

A continuación una breve explicación extraida de [5], página 70, respecto a cómo se usa y qué determina la prueba U de Mann-Whitney:

"El test U de Mann-Whitney es usado para comparar dos muestras no relacionadas o independientes. Las dos muestras son combinadas y ordenadas en rango en conjunto. La estrategia es determinar si los valores de las dos muestras están mezclados aleatoriamente en el ordenamiento de rango o si están agrupados en extremos opuestos al ser combinados. Un orden de rango aleatorio significará que las dos muestras no difieren, mientras que agrupaciones que contengan valores de una misma muestra indicarán una diferencia entre ellas."

En general, para todos los casos que veremos definiremos las hipótesis como [2]

H_0 : No hay tendencia en el rango de los valores obtenidos por una heurística que sea significativamente mayor (o menor) que el obtenido por la otra heurística.

H_1 : El rango de los valores obtenidos por una heurística es sistemáticamente mayor (o menor) que el obtenido por la otra heurística.

En lo anterior, al comparar los rangos no nos estamos refiriendo a la distancia entre el valor más pequeño y el valor más grande de una muestra, sino a que si alguna de las heurísticas tiende a obtener valores mayores (o menores) en comparación de la otra de manera consistente. Así pues, pudiera ocurrir que con una heurística hayamos obtenido la mejor solución de todas, pero si esto no lo hace de manera consistente quizá no sea la mejor opción. En cambio, si la comparáramos con una heurística que no haya obtenido la mejor solución de las dos pero que de manera consistente lograra obtener excelentes soluciones, podría ser una mejor opción esta última heurística.

Es importante obtener los mejores valores (en nuestro caso, las mejores mantas), pero si lo consiguiéramos muy rara vez no sería eficiente de manera consistente. De esta manera premiamos la consistencia de obtener muy buenas soluciones, aunque también no debemos dejar de lado el tiempo requerido para la obtención de los resultados.

Para las pruebas de hipótesis mantendremos un nivel de significancia $\alpha = 0.05$.

El estadístico que emplearemos para la prueba U de Mann-Whitney lo calcularemos como sigue

$$min\{U_1, U_2\} \; con \; U_i = n_1 n_2 + \frac{n_i(n_i+1)}{2} - \sum R_i$$

donde U_i es el test estadístico para la i-ésima muestra[3], n_i la cantidad de observaciones de la muestra i y $\sum R_i$ es la suma total de los rangos asignados a los valores de la i-ésima muestra.

Para el valor crítico nos fijamos en una tabla de valores críticos para la prueba U de Mann-Whitney, y como todas las muestras que usaremos consisten cada una de ocho valores obtenidos, el **valor crítico** cuando consideramos $n_1 = 8$, $n_2 = 8$ y $\alpha = 0.05$ es **U = 15**.

[2]Para definir así las hipótesis nos basamos en cómo se presentaron en [5], página 72.

[3]El parámetro i puede tomar el valor 1 o 2, los cuales indican la muestra de procedencia y la heurística empleada.

De esta manera, si en alguna prueba se cumple la desigualdad $min\{U_1, U_2\} \leq 15$ implicará que hay evidencia significativa para rechazar H_0 en favor de H_1, que en nuestras pruebas implicarán que una de las heurísticas tiende a obtener mejores resultados que la otra. Por el contrario, si la desigualdad $min\{U_1, U_2\} > U$ se cumple, no habrá evidencia suficiente para rechazar H_0, que en nuestras pruebas indicarán que no hay evidencia suficiente para afirmar que una de las heurísticas tiende a encontrar mejores mantas (o peores) que la otra. Más aún, en el caso de rechazar la hipótesis nula y si, por ejemplo, ocurriera que $\sum R_1 < \sum R_2$, podremos afirmar que la heurística usada para conseguir los valores correspondientes a la muestra 1 tiende a obtener valores más pequeños, y en consecuencia mejores mantas, que la heurística respectiva a la muestra 2.

Ahora, es importante notar lo siguiente:
Supongamos $\sum R_1 < \sum R_2$ y $n_1 = n_2$. Así

$$-\sum R_1 > -\sum R_2 \Rightarrow n_1 n_2 + \frac{n_1(n_1+1)}{2} - \sum R_1 > n_1 n_2 + \frac{n_2(n_2+1)}{2} - \sum R_2 \Rightarrow U_1 > U_2$$

Por tanto, bajo el supuesto de $n_1 = n_2$ concluimos que

$$\sum R_1 < R_2 \Rightarrow U_1 > U_2$$

Esto es importante, pues las pruebas que haremos cumplen $n_1 = n_2 = 8$, y en consecuencia siempre que rechacemos H_0 y suponiendo ocurre que $\sum R_1 < \sum R_2$ entonces la heurística que tiende a construir mejores mantas es aquella asociada al estadístico U_1.

Para facilitar la lectura de las tablas que mostraremos en cada comparación de heurísticas, primero deberemos tener en cuenta que el valor crítico a considerar en todas ellas es **U = 15**.

Los estadísticos que calculemos para las muestras de valores los representaremos con U_{BL}, U_{BLI}, U_{RS}, U_{BT1}, U_{BT2} y U_{AG}, los cuales correspnderán a Búsqueda Local, Búsqueda Local Iterada, Recocido Simulado, Búsqueda Tabú (caso 1), Búsqueda Tabú (caso 2) y Algoritmos Genéticos respectivamente. Aquellos que sean menores que el valor crítico, y en consecuencia se rechace H_0, los mostraremos color rojo, y la contraparte color verde. En consecuencia la heurística asociada al valor que sea coloreado de verde tenderá a construir mejores mantas de manera consistente en comparación con la otra heurística, para el valor de k en cuestión. Además, en los casos donde ninguno de los dos estadísticos sean menores que el valor crítico, colorearemos ambos de azul.

También graficaremos el tiempo promedio que tardan en correr las heurísticas para distintos valores de k.

Finalmente, con la información recabada procederemos a concluir la comparación comentando qué heurística consideramos mejor opción y por qué.

Las 12 Imágenes Objetivo que empleamos son las que se muestran a continuación.

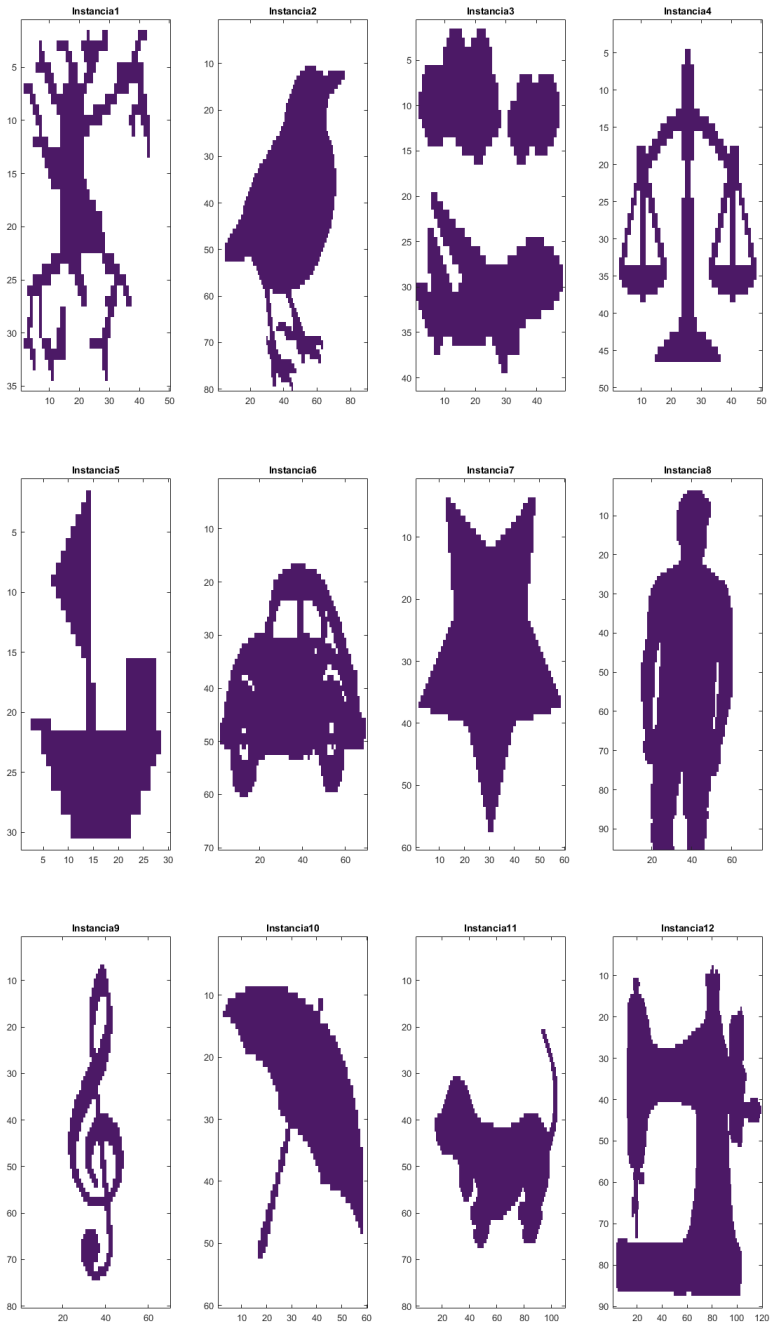


Figura 7.1: Imágenes Objetivo a emplear

La experimentación fue realizada en MATLAB, haciendo uso de una computadora cuyas características son: Intel Core i7-8565U, 1.8GHz y 4 núcleos de procesador.

Hemos de hacer especial mención que para el cálculo de los estadísticos para la prueba U de Mann-Whitney empleamos el script *mwwtest* de Cardillo ([3]).

7.1. Búsqueda Local vs. Búsqueda Local Iterada

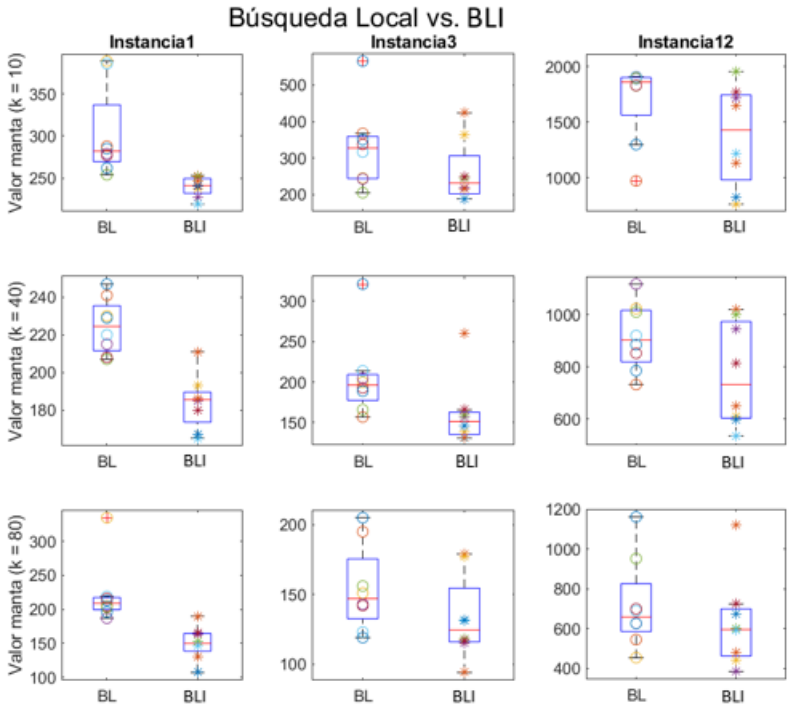


Figura 7.2: Diagramas de caja para Búsqueda Local y Búsqueda Local Iterada.

Por medio de los diagramas de caja para tres instancias distintas, podemos conjeturar que, en general, obtendremos mejores resultados con Búsqueda Local Iterada que con Búsqueda Local para cualquier valor que tome k. Esta suposición nace de forma natural debido a la naturaleza de Búsqueda Local Iterada, ya que, como mencionamos en el capítulo que le dedicamos, podemos ver a esta metaheurística como una extensión de Búsqueda Local.

A continuación presentamos los resultados de la prueba U de Mann-Whitney, y apoyándonos en ellos veremos si nuestra suposición resulta válida.

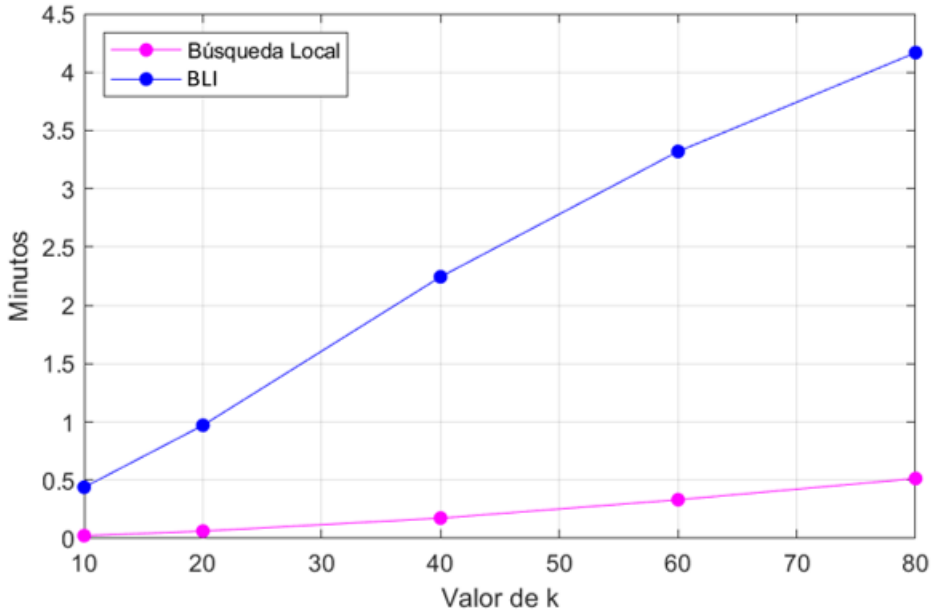


Figura 7.3: Tiempo total promedio para diferentes valores de k: Búsqueda Local vs. BLI.

Instancia	$k = 10$		$k = 20$		$k = 40$		$k = 60$		$k = 80$	
	U_{BL}	U_{BLI}	U_{BL}	U_{BLI}	U_{BL}	U_{BLI}	U_{BL}	U_{BLI}	U_{BL}	U_{BLI}
1	0	64	2	62	2	62	0	64	1	63
2	23	41	27	37	15	49	26	38	22	42
3	20	44	15.5	48.5	10	54	12.5	51.5	16	48
4	0	64	12	52	14.5	49.5	14	50	12	52
5	7	57	15.5	48.5	13.5	50.5	19	45	19	45
6	6	58	11.5	52.5	12	52	13	51	19	45
7	10.5	53.5	14.5	49.5	12	52	14.5	49.5	23.5	40.5
8	19	45	17	47	30	34	21.5	42.5	36	28
9	8	56	8.5	55.5	19	45	17	47	15	49
10	21.5	42.5	19.5	44.5	18	46	21.5	42.5	22.5	41.5
11	20.5	43.5	25	39	23.5	40.5	22	42	22	42
12	16	48	10	54	18	46	20	44	22	42

Podemos observar que cuando el parámetro k toma valores pequeños, la heurística que mejor funciona es BLI, y conforme aumentamos su valor la diferencia entre usar una u otra heurística va disminuyendo, aunque por lo visto, ante cualquier duda, convendrá usar BLI, pues en general es consistente en lograr obtener mantas iguales o mejores que Búsqueda Local.

A pesar de que BLI tarda más tiempo en correr, este no es excesivamente superior.

Así, podemos concluir que, en este caso, la metaheurística Búsqueda Local Iterada es mejor opción en comparación con Búsqueda Local.

7.2. Búsqueda Local vs. Recocido Simulado

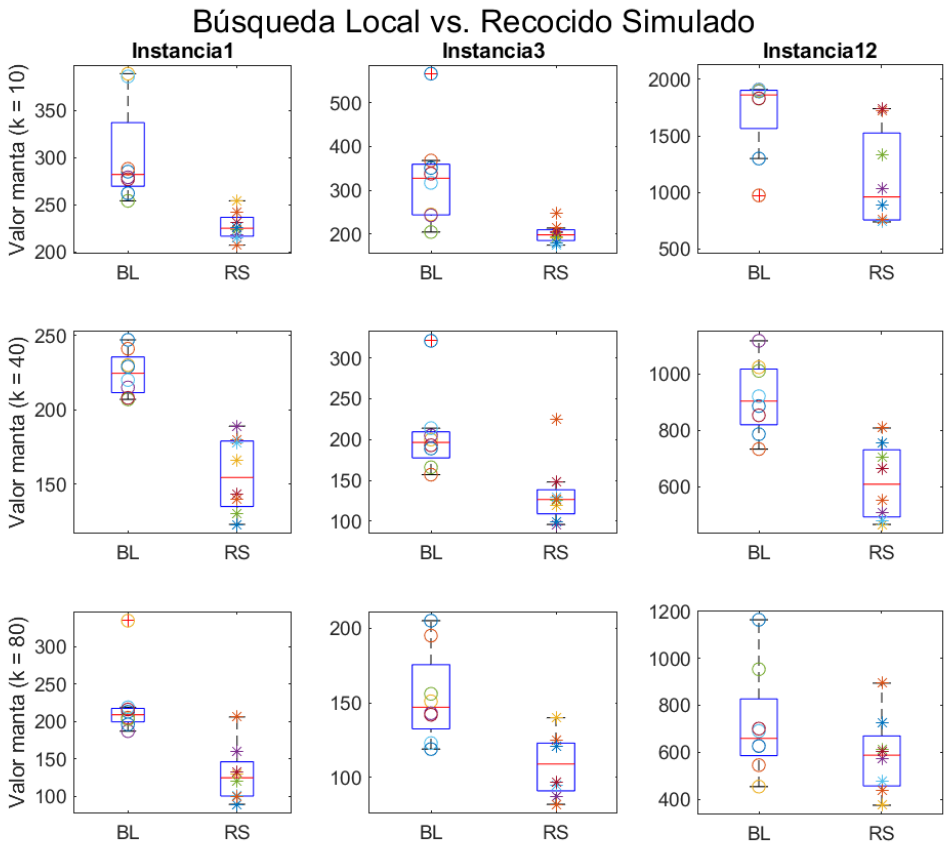


Figura 7.4: Diagramas de caja para Búsqueda Local y Recocido Simulado.

La Figura 7.4 nos permite conjeturar que obtendremos mejores resultados con Recocido Simulado, pues teniendo en cuenta todas las comparaciones mostradas es claro que Recocido Simulado tiende a encontrar mejores soluciones, supuesto que será validado o desmentido por la prueba de Mann-Whitney.

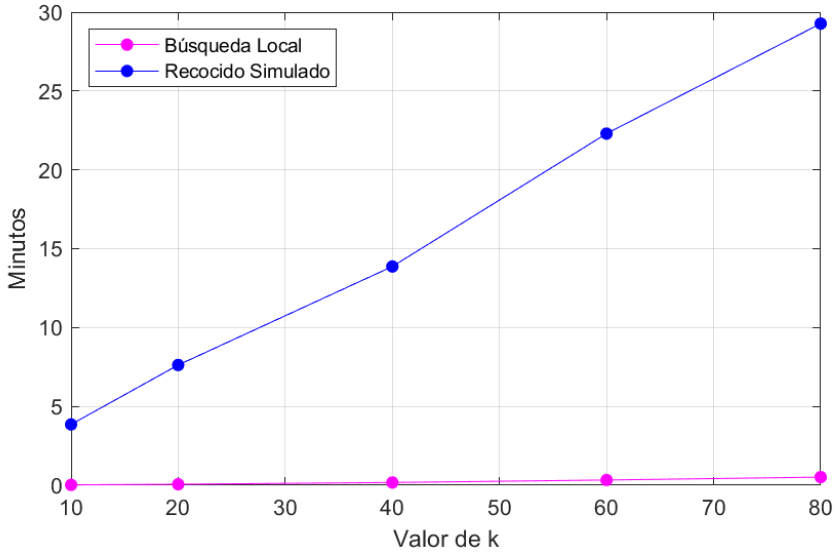


Figura 7.5: Tiempo total promedio para diferenes valores de k: Búsqueda Local vs. Recocido Simulado.

Instancia	$k = 10$		$k = 20$		$k = 40$		$k = 60$		$k = 80$	
	U_{BL}	U_{RS}	U_{BL}	U_{RS}	U_{BL}	U_{RS}	U_{BL}	U_{RS}	U_{BL}	U_{RS}
1	0.5	63.5	0	64	0	64	0	64	4	60
2	14	50	10	54	15	49	11.5	52.5	18.5	45.5
3	5	59	0	64	7	57	8	56	6	58
4	0	64	2.5	61.5	2	62	8	56	3	61
5	0	64	0	64	2	62	0	64	1	63
6	5	59	0	64	4	60	5	59	3	61
7	5	59	1	63	6	58	5	59	8	56
8	7	57	9	55	6	58	26	38	29	35
9	1	63	0	64	8	56	6	58	7	57
10	1	63	3	61	4	60	5.5	58.5	11	53
11	14.5	49.5	18	46	17.5	46.5	18	46	20	44
12	7	57	5	59	3	61	17	47	19	45

Independientemente de que Recocido Simulado sea más tardado que Búsqueda Local (ver la Figura 7.5), es más que claro que es la mejor opción en esta ocasión, pues es consistente en lograr obtener mejores mantas que Búsqueda Local.

7.3. Búsqueda Local vs. Búsqueda Tabú (caso 1)

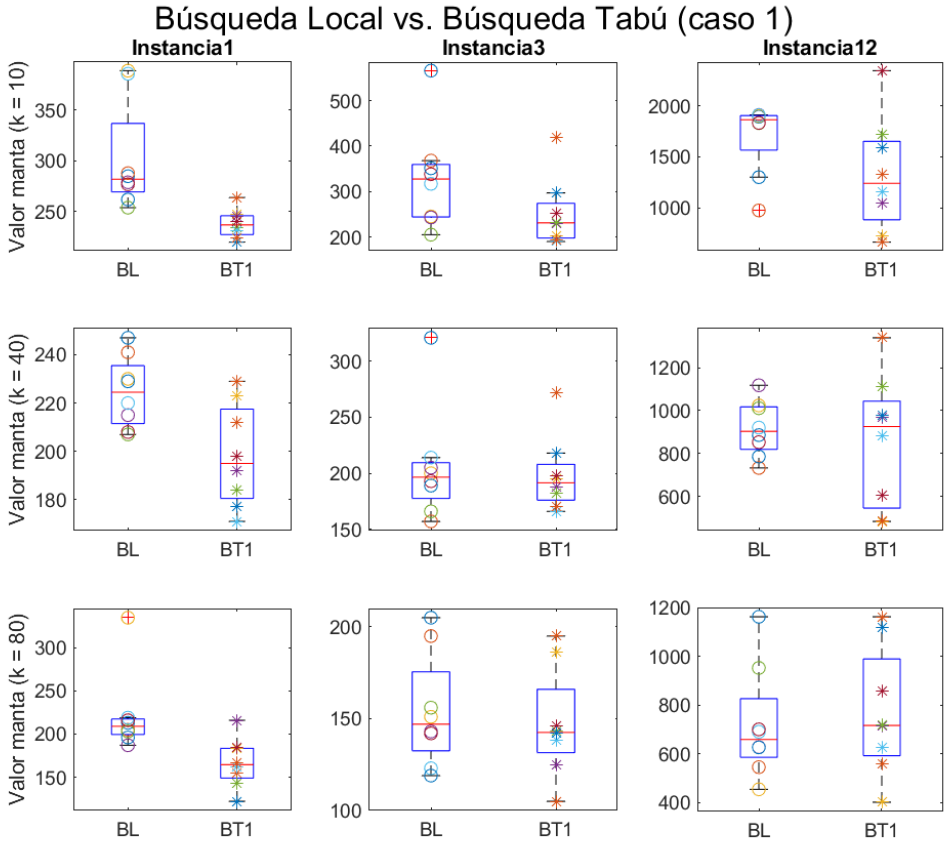


Figura 7.6: Diagramas de caja para Búsqueda Local y Búsqueda Tabú (caso 1).

Podemos conjeturar a partir de los diagramas de la Figura 7.6 que, en general, no hay mucha diferencia entre usar una u otra heurística, aunque habrá algunos casos en los cuales Búsqueda Tabú (caso 1) arroje mejores resultados, pero ello no implica que sea mejor usarla, puesto que aún falta considerar el tiempo que tarda cada una.

Antes, sustentemos mediante la prueba U de Mann-Whitney si hay diferencia en los resultados al usar una u otra heurística o si no los hay.

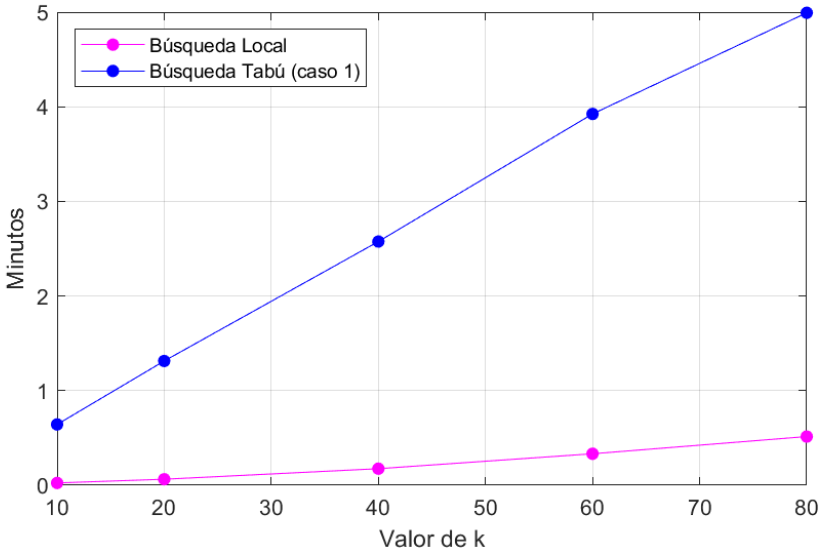


Figura 7.7: Tiempo total promedio para diferenes valores de k: Búsqueda Local vs. Búsqueda Tabú (caso 1).

Instancia	$k = 10$		$k = 20$		$k = 40$		$k = 60$		$k = 80$	
	U_{BL}	U_{BT1}	U_{BL}	U_{BT1}	U_{BL}	U_{BT1}	U_{BL}	U_{BT1}	U_{BL}	U_{BT1}
1	2	62	2	62	10.5	53.5	4.5	59.5	5.5	58.5
2	22	42	30	34	25	39	30.5	33.5	41	23
3	15	49	20.5	43.5	29.5	34.5	28	36	26.5	37.5
4	0	64	10	54	10	54	23	41	19.5	44.5
5	1	63	9	55	21.5	42.5	20	44	16	48
6	11	53	6	58	26	38	21	43	20	44
7	9	55	21.5	42.5	24	40	13	51	26.5	37.5
8	4	60	16	48	21	43	21	43	37.5	26.5
9	5	59	2.5	61.5	20	44	25	39	14	50
10	13	51	30	34	23.5	40.5	27	37	30.5	33.5
11	17	47	19	45	19	45	23	41	25	39
12	16	48	20	44	28	36	25	39	38	26

Cuando el valor del parámetro k es pequeño, es recomendado usar Búsqueda Tabú, y conforme el valor de k va creciendo la diferencia entre usar una u otra va disminuyendo. De cualquier manera y aún teniendo en cuenta el tiempo que tarda cada heurística, concluimos que la heurística Búsqueda Tabú es mejor opción para este problema.

7.4. Búsqueda Local vs. Búsqueda Tabú (caso 2)

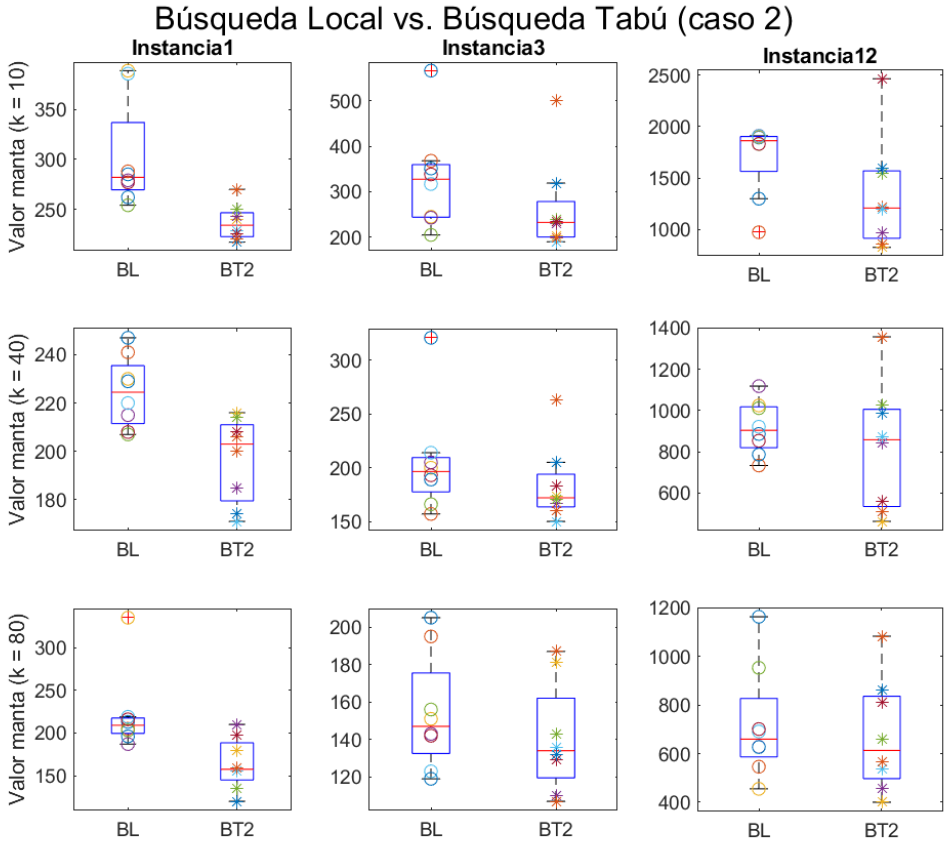


Figura 7.8: Diagramas de caja para Búsqueda Local y Búsqueda Tabú (caso 2).

Para esta comparación tenemos los mismos comentarios que los realizados en la anterior comparación, aunque hemos de añadir que, al parecer, se ven más erráticos los resultados que obtiene Búsqueda Tabú (caso 2), aunque no por eso peores, pues la media de los valores obtenidos por medio de Búsqueda Tabú (caso 2) resulta estar por debajo de la media obtenida por Búsqueda Local en todos los diagramas.

Sustentaremos a continuación si hay diferencia entre el uso de ambas heurísticas.

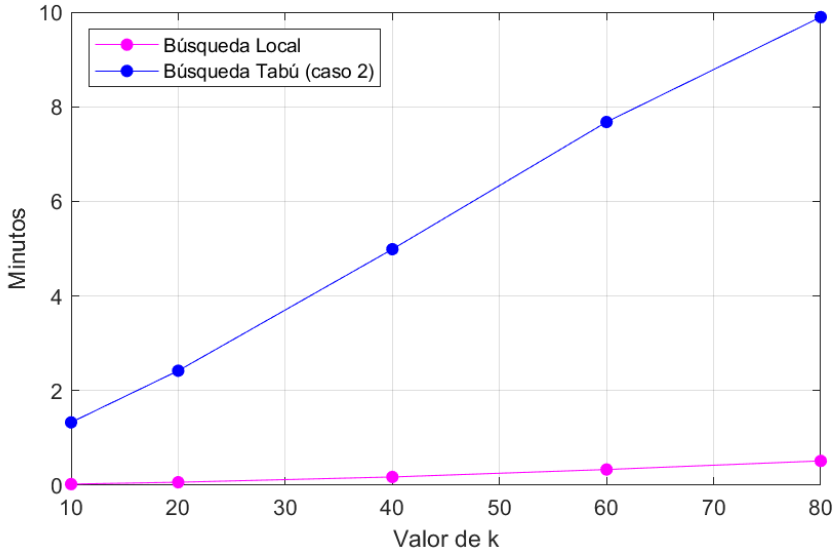


Figura 7.9: Tiempo total promedio para diferenes valores de k: Búsqueda Local vs. Búsqueda Tabú (caso 2).

Instancia	$k = 10$		$k = 20$		$k = 40$		$k = 60$		$k = 80$	
	U_{BL}	U_{BT2}	U_{BL}	U_{BT2}	U_{BL}	U_{BT2}	U_{BL}	U_{BT2}	U_{BL}	U_{BT2}
1	2	62	0	64	6.5	57.5	5	59	6	58
2	21	43	30	34	20	44	25	39	34	30
3	14	50	15	49	21.5	42.5	17	47	21.5	42.5
4	0	64	7	57	6.5	57.5	13	51	12	52
5	4	60	9	55	17	47	7	57	13.5	50.5
6	12.5	51.5	6	58	16	48	12	52	9.5	54.5
7	8.5	55.5	16.5	47.5	18	46	9	55	18	46
8	18	46	16	48	13	51	14	50	36	28
9	2	62	2	62	13.5	50.5	18	46	10	54
10	19	45	20	44	19.5	44.5	24	40	22.5	41.5
11	15	49	22	42	20	44	23	41	19	45
12	14	50	12	52	25	39	22	42	27	37

Como comentamos antes, el ser más erráticos no implica obtener peores valores. A diferencia del caso 1 de Búsqueda Tabú, podemos concluir con más contundencia que es mejor opción usar el caso 2 de Búsqueda Tabú, en lugar de Búsqueda Local, para cualquier valor de k.

7.5. Búsqueda Local vs. Algoritmos Genéticos

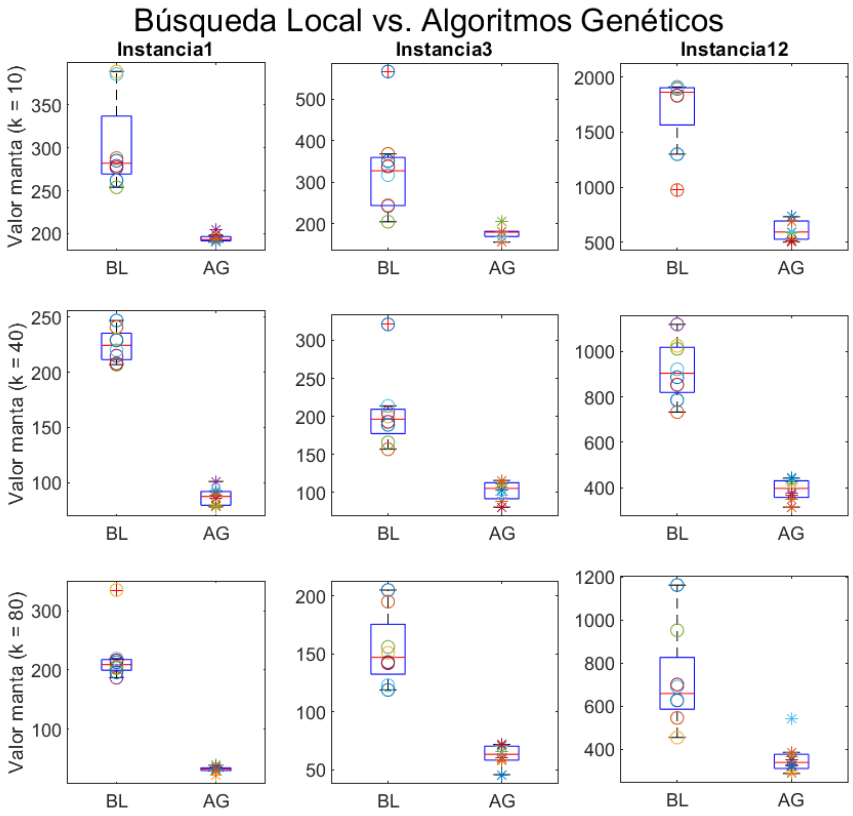


Figura 7.10: Diagramas de caja para Búsqueda Local y Algoritmos Genéticos.

Claramente vemos que Algoritmos Genéticos resulta ser mejor heurística que Búsqueda Local, aunque aún así esto será sustentado estadísticamente a continuación.

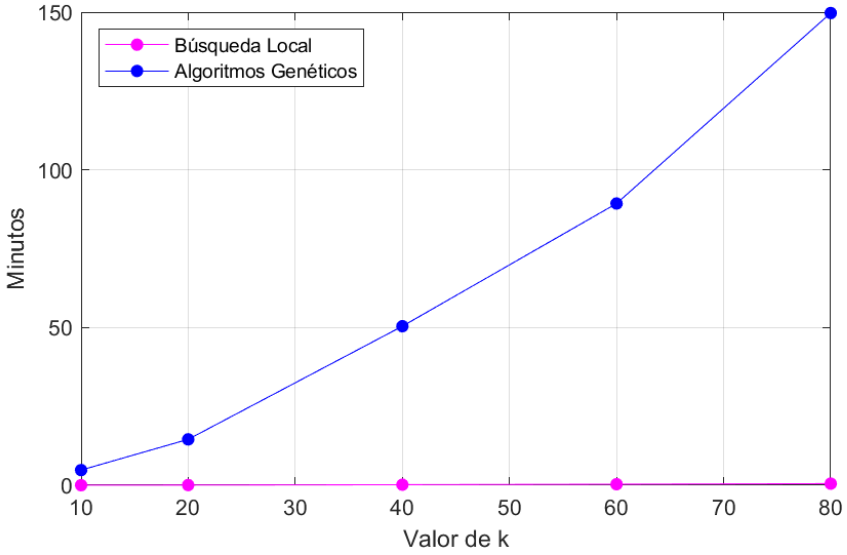


Figura 7.11: Tiempo total promedio para diferenes valores de k: Búsqueda Local vs. Algoritmos Genéticos.

Instancia	$k = 10$		$k = 20$		$k = 40$		$k = 60$		$k = 80$	
	U_{BL}	U_{AG}	U_{BL}	U_{AG}	U_{BL}	U_{AG}	U_{BL}	U_{AG}	U_{BL}	U_{AG}
1	0	64	0	64	0	64	0	64	0	64
2	0	64	0	64	0	64	0	64	0	64
3	0.5	63.5	0	64	0	64	0	64	0	64
4	0	64	0	64	0	64	0	64	0	64
5	0	64	0	64	0	64	0	64	0	64
6	0	64	0	64	0	64	0	64	0	64
7	0	64	0	64	0	64	0	64	0	64
8	0	64	0	64	0	64	1	63	3	61
9	0	64	0	64	0	64	0	64	0	64
10	0	64	0	64	0	64	0	64	0	64
11	0	64	0	64	0	64	0	64	0	64
12	0	64	0	64	0	64	0	64	1	63

Ahora sí, con un fuerte sustento estadístico, concluimos que Algoritmos Genéticos es mejor opción a pesar del tiempo que tardó en llegar al resultado.

7.6. Búsqueda Local Iterada vs. Recocido Simulado

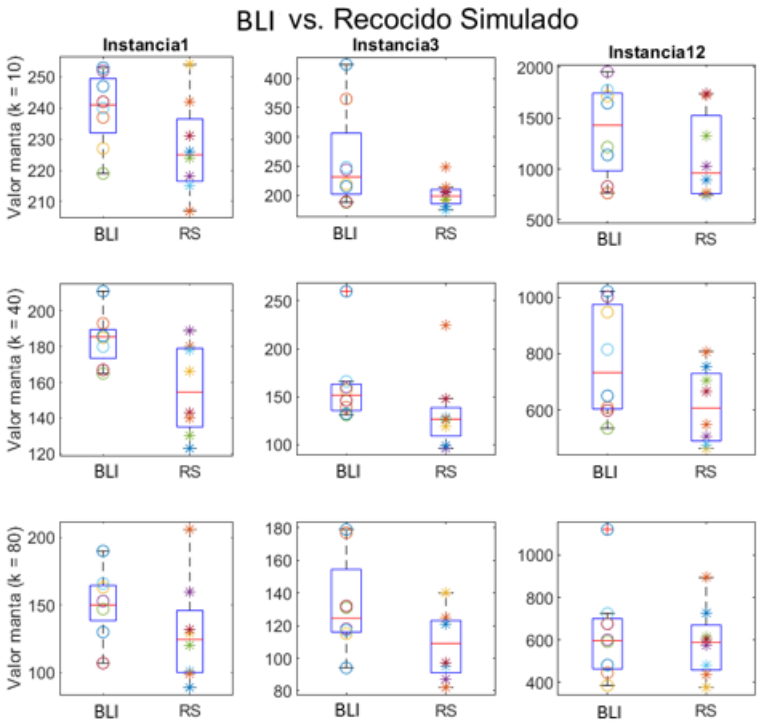


Figura 7.12: Diagramas de caja para Búsqueda Local Iterada y Recocido Simulado.

Al compararlas podemos observar que Recocido Simulado da mejores resultados, pero debemos probar si tal mejoría es estadísticamente significativa o no.

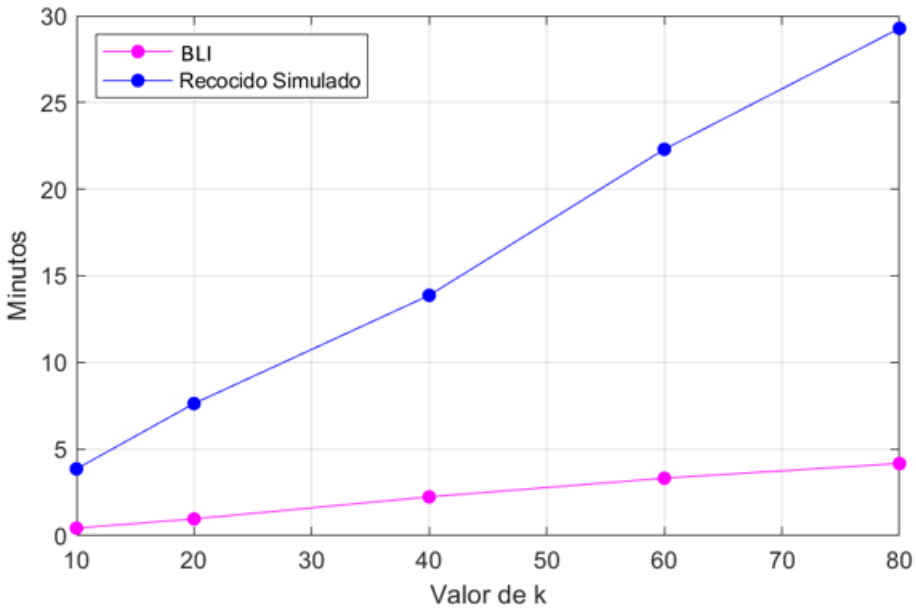


Figura 7.13: Tiempo total promedio para diferenes valores de k: BLI vs. Recocido Simulado.

Instancia	$k = 10$		$k = 20$		$k = 40$		$k = 60$		$k = 80$	
	U_{BLI}	U_{RS}	U_{BLI}	U_{RS}	U_{BLI}	U_{RS}	U_{BLI}	U_{RS}	U_{BLI}	U_{RS}
1	16.5	47.5	13	51	11.5	52.5	10	54	17	47
2	16	48	16	48	29	35	18	46	29	35
3	15.5	48.5	12.5	51.5	11	53	17	47	20	44
4	0	64	13	51	9	55	17	47	14	50
5	4.5	59.5	11.5	52.5	17	47	5.5	58.5	7.5	56.5
6	23.5	40.5	6	58	20.5	43.5	24	40	9	55
7	27.5	36.5	8	56	18	46	14	50	14.5	49.5
8	18	46	19	45	12	52	36	28	26	38
9	11	53	10	54	19	45	23	41	17.5	46.5
10	6	58	8	56	11	53	18	46	13	51
11	22.5	41.5	24	40	30	34	27	37	27	37
12	21	43	16	48	17	47	28	36	29.5	34.5

Podemos observar en la tabla que Recocido Simulado suele conseguir resulados iguales o mejores que Búsqueda Local Iterada, aunque conforme incrementamos el valor de k la disparidad disminuye. No obstante, nuestra postura y conclusión da prioridad a la heurística Recocido Simulado por encima de Búsqueda Local Iterada.

7.7. Búsqueda Local Iterada vs. Búsqueda Tabú (caso 1)

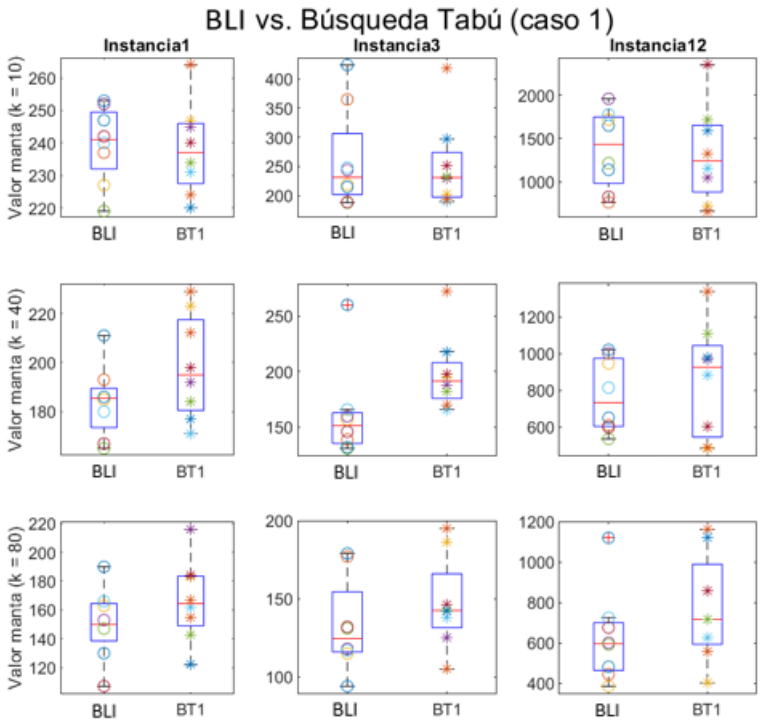


Figura 7.14: Diagramas de caja para Búsqueda Local Iterada y Búsqueda Tabú (caso 1).

Nuestra suposición es que para valores grandes de k, Búsqueda Local Iterada es mejor que Búsqueda Tabú (caso 1), aunque para valores chicos pudiera ser que no haya diferencia significativa entre usar una u otra, e incluso, quizá, de mejores resultados Búsqueda Tabú (caso 1). A continuación procedemos a usar la prueba U de Mann-Whitney.

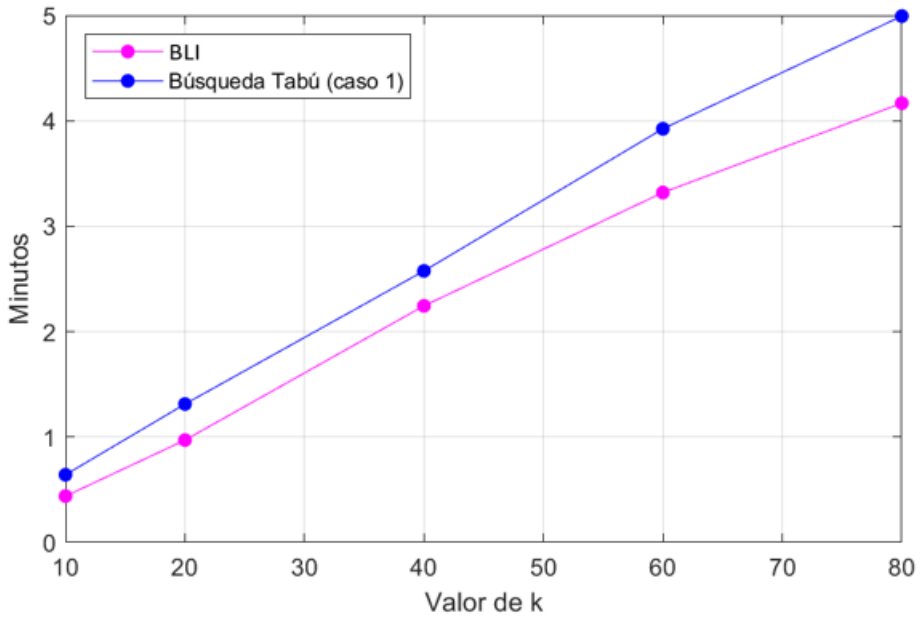


Figura 7.15: Tiempo total promedio para diferenes valores de k: Búsqueda Local Iterada vs. Búsqueda Tabú (caso 1).

Instancia	$k = 10$		$k = 20$		$k = 40$		$k = 60$		$k = 80$	
	U_{BLI}	U_{BT1}	U_{BLI}	U_{BT1}	U_{BLI}	U_{BT1}	U_{BLI}	U_{BT1}	U_{BLI}	U_{BT1}
1	28	36	36.5	27.5	44	20	54	10	42	22
2	30	34	37	27	39	25	40	24	50.5	13.5
3	33	31	38	26	56.5	7.5	47	17	45	19
4	22	42	33.5	30.5	26	38	41	23	45	19
5	14	50	29	35	44.5	19.5	39	25	34	30
6	36	28	23	41	47	17	42.5	21.5	35.5	28.5
7	35	29	39	25	48.5	15.5	32.5	31.5	34.5	29.5
8	17	47	29	35	24	40	34	30	35	29
9	20	44	28.5	35.5	38	26	42	22	28	36
10	19.5	44.5	39	25	42.5	21.5	39	25	44	20
11	25	39	30	34	31	33	32	32	35.5	28.5
12	26.5	37.5	35	29	35	29	39	25	43	21

Con los datos obtenidos podemos observar que, en general, no hay diferencia estadísticamente significativa entre usar una u otra heurística. En este caso, a diferencia de los que hemos estudiado hasta el momento, podemos darle mayor peso al tiempo, con lo cual nuestra recomendación sería usar Búsqueda Local Iterada en lugar de este caso de Búsqueda Tabú, aunque, al igual que con los valores que se obtienen con ambas, tampoco vemos mucha diferencia en el tiempo de las dos.

7.8. Búsqueda Local Iterada vs. Búsqueda Tabú (caso 2)

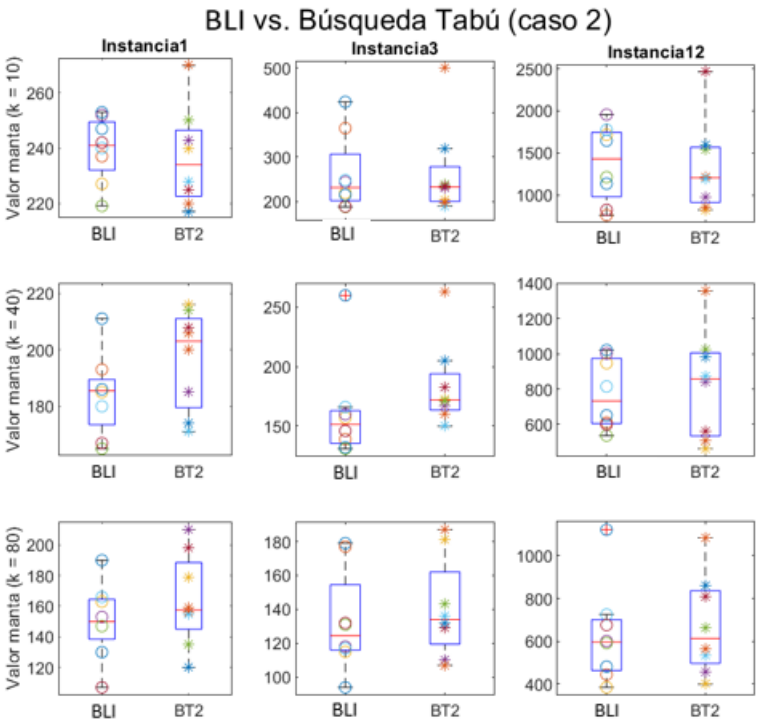


Figura 7.16: Diagramas de caja para Búsqueda Local Iterada y Búsqueda Tabú (caso 2).

En este caso, a diferencia de Búsqueda Tabú (caso 1), para valores pequeños de k conjeturamos que no habrá diferencia entre ambas heurísticas, y cuando k tome valores grandes quizá tampoco haya diferencia estadísticamente significativa en favor de Búsqueda Local Iterada.

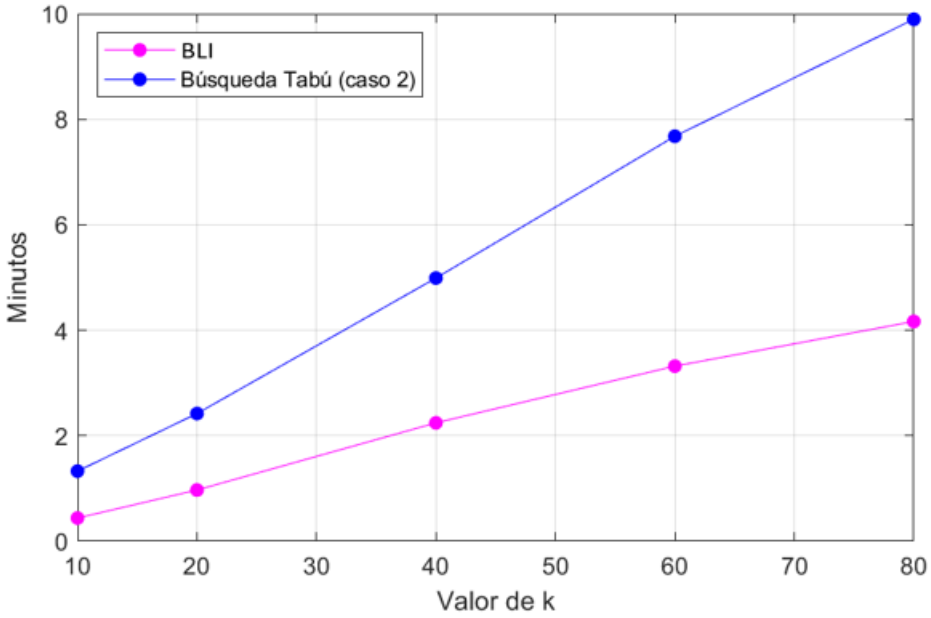


Figura 7.17: Tiempo total promedio para diferenes valores de k: Búsqueda Local Iterada vs. Búsqueda Tabú (caso 2).

Instancia	$k = 10$		$k = 20$		$k = 40$		$k = 60$		$k = 80$	
	U_{BLI}	U_{BT2}	U_{BLI}	U_{BT2}	U_{BLI}	U_{BT2}	U_{BLI}	U_{BT2}	U_{BLI}	U_{BT2}
1	26.5	37.5	40.5	23.5	44.5	19.5	46	18	41	23
2	27.5	36.5	37	27	36	28	32	32	41	23
3	32	32	34.5	29.5	52.5	11.5	38	26	39.5	24.5
4	23	41	30	34	23	41	32	32	42	22
5	20.5	43.5	25	39	39.5	24.5	25	39	31	33
6	41	23	22	42	37	27	35	29	23	41
7	36	28	34	30	45	19	26	38	24	40
8	32	32	27	37	17	47	28	36	33	31
9	19	45	24.5	39.5	24	40	35	29	28	36
10	27	37	35	29	37	27	34	30	41	23
11	19	45	32	32	30	34	31	33	30.5	33.5
12	28	36	18	46	33	31	38	26	35	29

Nuestra opinión en esta comparación es la misma que en la anterior, pues no hay diferencia significativa de los datos obtenidos con una u otra heurística para cualquier valor de k. Nuestra recomendación sería usar Búsqueda Local Iterada en lugar del caso 2 de Búsqueda Tabú, pues es más rápida para encontrar buenas mantas.

7.9. Búsqueda Local Iterada vs. Algoritmos Genéticos

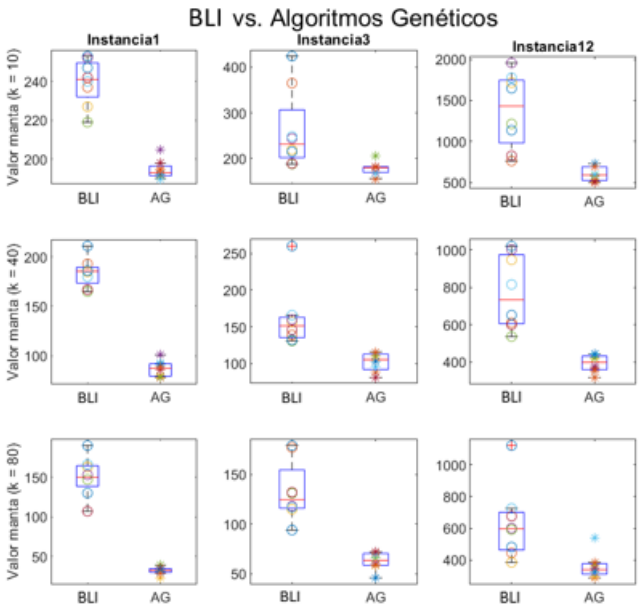


Figura 7.18: Diagramas de caja para Búsqueda Local Iterada y Algoritmos Genéticos.

De nuevo, nuestra conjetura gira en torno a una sustancial diferencia en favor de la heurística Algoritmos Genéticos.

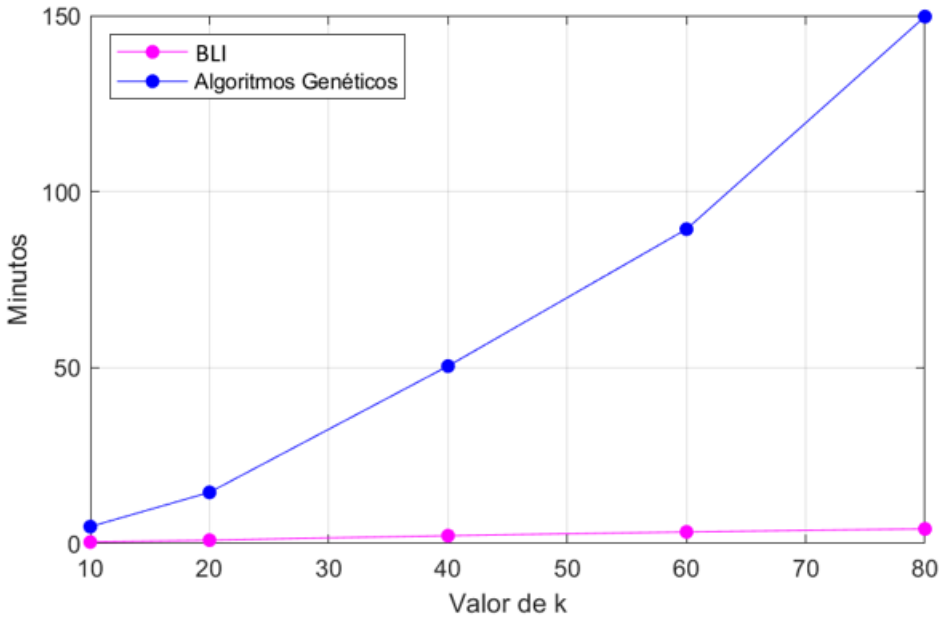


Figura 7.19: Tiempo total promedio para diferenes valores de k: Búsqueda Local Iterada vs. Algoritmos Genéticos.

Instancia	$k = 10$		$k = 20$		$k = 40$		$k = 60$		$k = 80$	
	U_{BLI}	U_{AG}	U_{BLI}	U_{AG}	U_{BLI}	U_{AG}	U_{BLI}	U_{AG}	U_{BLI}	U_{AG}
1	0	64	0	64	0	64	0	64	0	64
2	0	64	0	64	0	64	0	64	0	64
3	2	62	6	58	0	64	0	64	0	64
4	0	64	0	64	0	64	0	64	0	64
5	1.5	62.5	0	64	0	64	0	64	0	64
6	0	64	0	64	0	64	0	64	0	64
7	1	63	0	64	0	64	0	64	0	64
8	0	64	0	64	0	64	5.5	58.5	2	62
9	0	64	0	64	0	64	0	64	0	64
10	0	64	0	64	0	64	0	64	0	64
11	0	64	0	64	0	64	0	64	0	64
12	0	64	0	64	0	64	2	62	4	60

No hay duda que Algoritmos Genéticos es mejor opción que Búsqueda Local Iterada, pues como podemos ver en la tabla la diferencia de los valores entre ambas es estadísticamente significativa para todos los casos.

Rechazamos H_0 en todos los casos y $U_{BLI} < U_{AG}$, con lo cual concluimos que Algoritmos Genéticos construye en todos los casos mejores mantas que Búsqueda Local Iterada, de manera consistente.

7.10. Recocido Simulado vs. Búsqueda Tabú (caso 1)

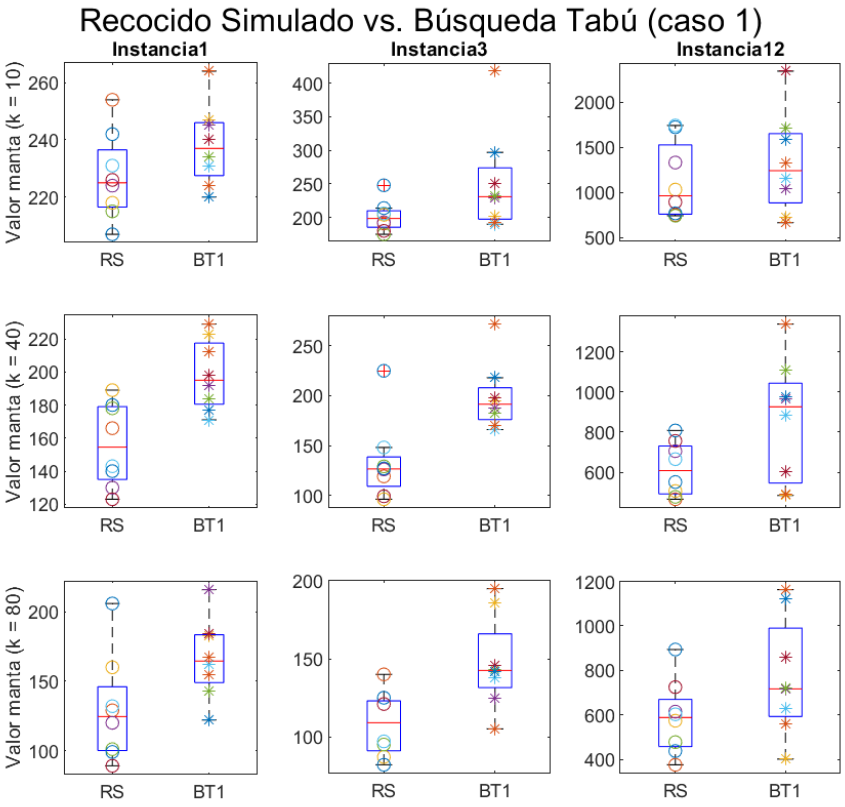


Figura 7.20: Diagramas de caja para Recocido Simulado y Búsqueda Tabú (caso 1).

Podemos ver que, en general, se obtuvieron mejores resultados con Recocido Simulado que con Búsqueda Tabú (caso 1). A continuación veremos si nuestra conjetura se cumple.

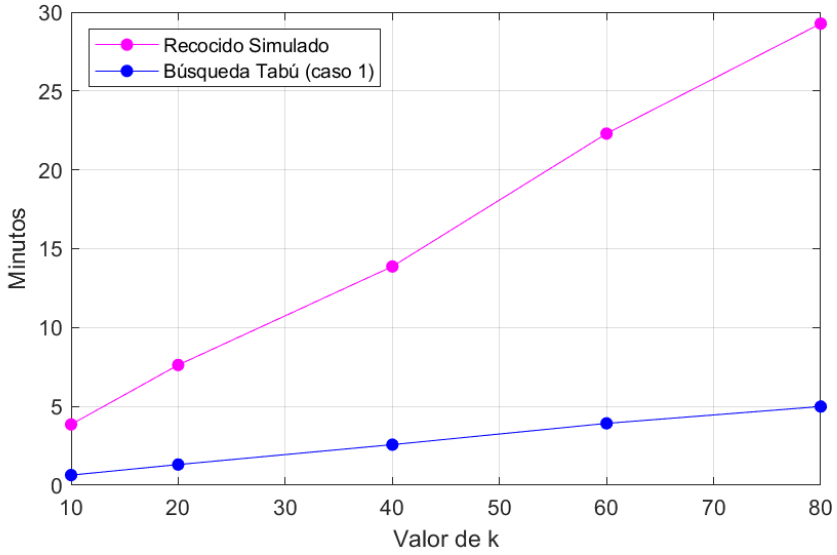


Figura 7.21: Tiempo total promedio para diferenes valores de k: Recocido Simulado vs. Búsqueda Tabú (caso 1).

Instancia	$k = 10$		$k = 20$		$k = 40$		$k = 60$		$k = 80$	
	U_{RS}	U_{BT1}	U_{RS}	U_{BT1}	U_{RS}	U_{BT1}	U_{RS}	U_{BT1}	U_{RS}	U_{BT1}
1	46	18	54.5	9.5	57	7	64	0	52	12
2	44	20	51	13	40	24	52	12	52	12
3	47.5	16.5	59	5	57	7	55	9	57.5	6.5
4	55	9	48	16	52.5	11.5	53	11	59	5
5	49.5	14.5	57	7	58	6	64	0	60	4
6	44	20	49	15	58	6	52.5	11.5	59	5
7	40	24	58.5	5.5	55	9	49	15	52.5	11.5
8	31	33	39.5	24.5	49	15	30.5	33.5	40	24
9	43	21	51.5	12.5	47	17	48	16	41.5	22.5
10	48	16	58	6	57	7	56	8	52	12
11	38	26	42	22	34.5	29.5	38	26	41	23
12	35	29	46	18	48	16	37.5	26.5	45	19

Podemos ver que para valores muy pequeños no hay diferencia significativa entre usar una u otra heurística, pero cuando el parámetro k toma valores cada vez más grandes, la diferencia de los valores de las mantas que construyen ambas heurísticas incrementa y queda sustentada estadísticamente.

Cuando k toma valores pequeños, la diferencia de tiempos no es mucha, pero a medida que incrementamos su valor, tal diferencia de tiempos también crece. Es importante notar que aunque Recocido Simulado demore en construir mantas, cuando termina suelen ser mejores que las obtenidas por medio de Búsqueda Tabú.

Podemos concluir con esta comparación que Recocido Simulado es mejor opción para encontrar buenas mantas en este problema.

7.11. Recocido Simulado vs. Búsqueda Tabú (caso 2)

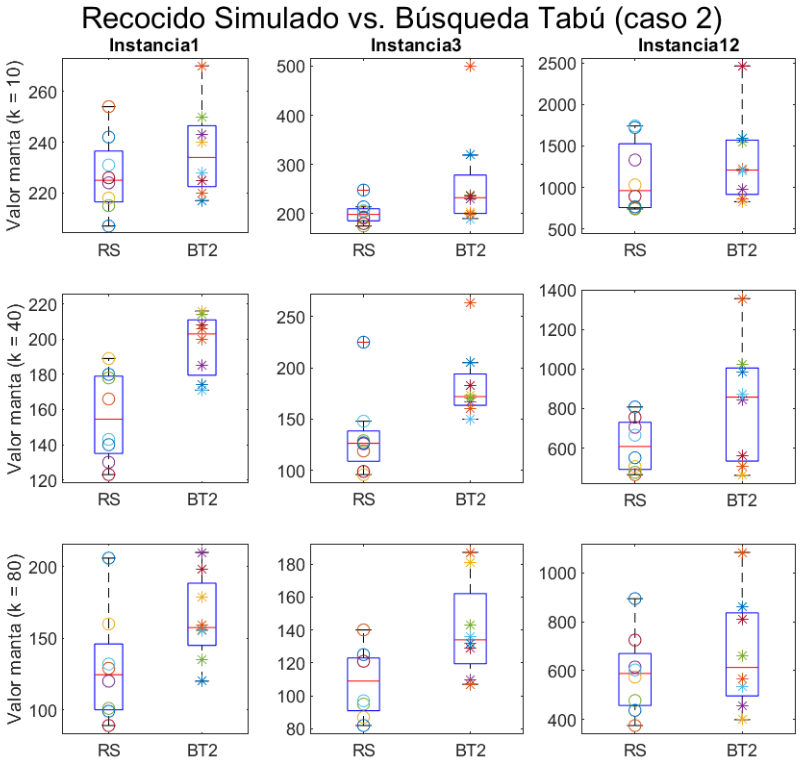


Figura 7.22: Diagramas de caja para Recocido Simulado y Búsqueda Tabú (caso 2).

Vemos que, en ocasiones, obtenemos mejores resultados con Recocido Simulado, aunque pudiera ser que no sean estadísticamente significativos. Veamos a continuación si hay diferencia significativa entre los resultados.

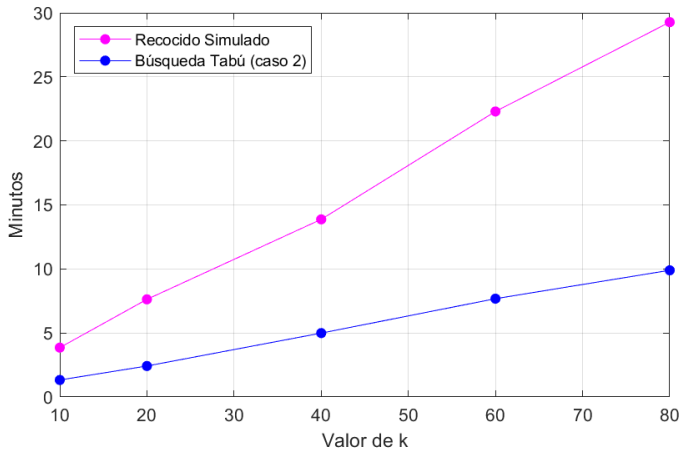


Figura 7.23: Tiempo total promedio para diferentes valores de k: Recocido Simulado vs. Búsqueda Tabú (caso 2).

Instancia	$k = 10$		$k = 20$		$k = 40$		$k = 60$		$k = 80$	
	U_{RS}	U_{BT2}	U_{RS}	U_{BT2}	U_{RS}	U_{BT2}	U_{RS}	U_{BT2}	U_{RS}	U_{BT2}
1	42	22	54	10	57	7	64	0	49.5	14.5
2	44	20	51	13	38	26	49	15	46	18
3	47	17	57	7	57	7	54	10	53	11
4	59	5	49	15	55	9	50	14	58	6
5	57	7	55.5	8.5	56	8	56	8	58.5	5.5
6	47.5	16.5	50	14	49	15	46	18	57	7
7	42	22	57	7	57	7	43	21	48.5	15.5
8	47	17	38	26	43	21	23.5	40.5	39	25
9	45.5	18.5	50.5	13.5	40	24	43	21	44.5	19.5
10	58	6	58	6	56	8	53	11	52	12
11	34	30	42.5	21.5	32.5	31.5	36	28	36	28
12	40	24	38	26	47	17	34.5	29.5	37	27

Viendo la tabla de resultados de la prueba de Mann-Whitney podemos observar que en algunas ocasiones es estadísticamente significativa la diferencia de valores en favor a Recocido Simulado, aunque en la mayoría de los casos no es estadísticamente significativa esta diferencia. De cualquier modo, nuestra recomendación en este caso es emplear Recocido Simulado, pues si bien por la Figura 7.23 podemos notar que Recocido Simulado tarda más en encontrar una solución, es probable que esta sea igual o mejor solución que alguna encontrada por el caso 2 de Búsqueda Tabú y muy poco probable que sea peor solución.

7.12. Recocido Simulado vs. Algoritmos Genéticos

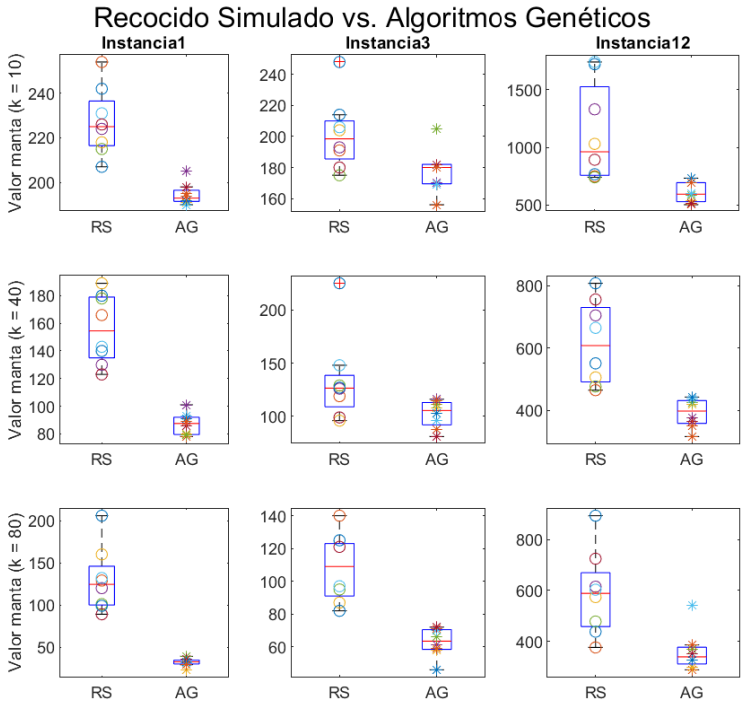


Figura 7.24: Diagramas de caja para Recocido Simulado y Algoritmos Genéticos.

Al igual que en casos anteriores, podemos observar con claridad que Algoritmos Genéticos obtiene, en general, mejores mantas que Recocido Simulado, conjetura que veremos si es válida o no con la prueba U de Mann- Whitney.

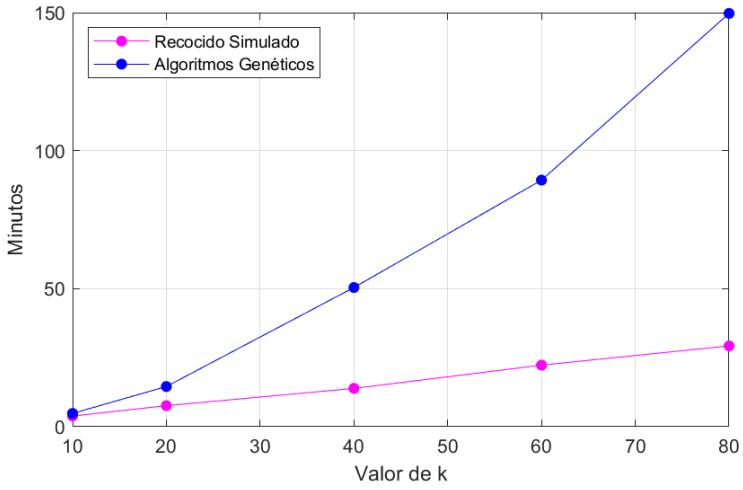


Figura 7.25: Tiempo total promedio para diferenes valores de k: Recocido Simulado vs. Algoritmos Genéticos.

Instancia	$k = 10$		$k = 20$		$k = 40$		$k = 60$		$k = 80$	
	U_{RS}	U_{AG}	U_{RS}	U_{AG}	U_{RS}	U_{AG}	U_{RS}	U_{AG}	U_{RS}	U_{AG}
1	0	64	0	64	0	64	0	64	0	64
2	0	64	0	64	0	64	0	64	2	62
3	12	52	8	56	10.5	53.5	3	61	0	64
4	0	64	0	64	0	64	0	64	0	64
5	20.5	43.5	3.5	60.5	4	60	0	64	0	64
6	0	64	3	61	0	64	0	64	1	63
7	4.5	59.5	0	64	0	64	0	64	0	64
8	2	62	0	64	2	62	4	60	5	59
9	0	64	0	64	0	64	0	64	0	64
10	4	60	0	64	0	64	0	64	1	63
11	0	64	0	64	0	64	0	64	2	62
12	0	64	0	64	0	64	4	60	4	60

Con lo recabado en la tabla queda sustentada nuestra conjetura, pues la diferencia de entre los valores que se obtienen es estadísticamente significativa . Así, en conclusión, Algoritmos Genéticos es mejor opción.

7.13. Búsqueda Tabú (caso 1) vs. Búsqueda Tabú (caso 2)

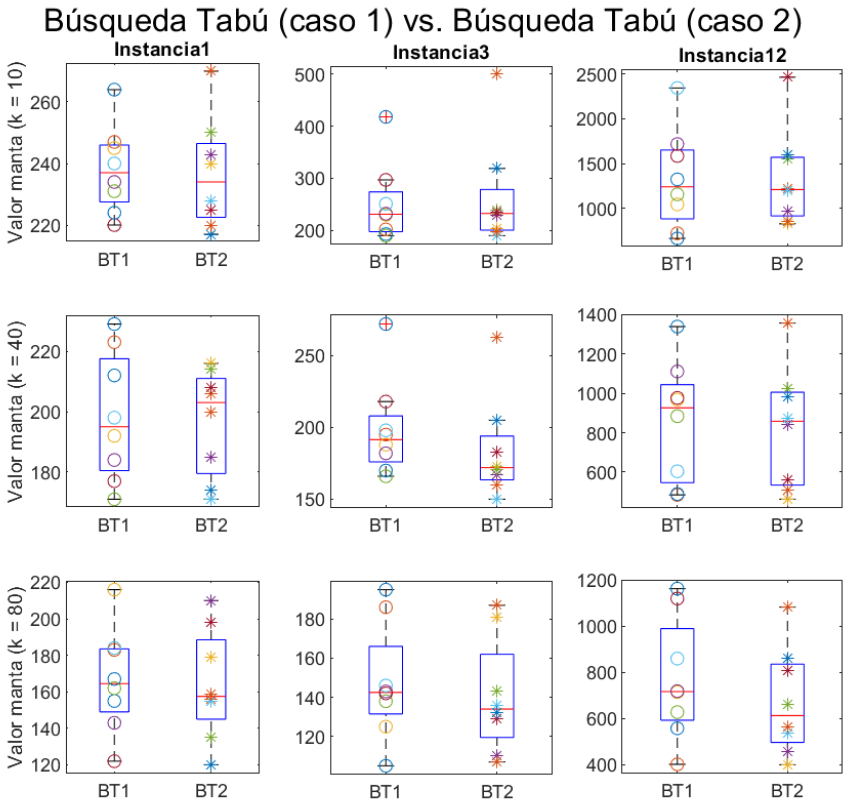


Figura 7.26: Diagramas de caja para Búsqueda Tabú (caso 1) y Búsqueda Tabú (caso 2).

Nuestra conjetura para este caso es que no habrá diferencia significativa entre usar una u otra heurística, aunque cuando el valor de k es alto pudiera resultar mejor el segundo caso de Búsqueda Tabú, pero no nos atrevemos a afirmar que así será. Procedamos a usar la prueba de Mann-Whitney.

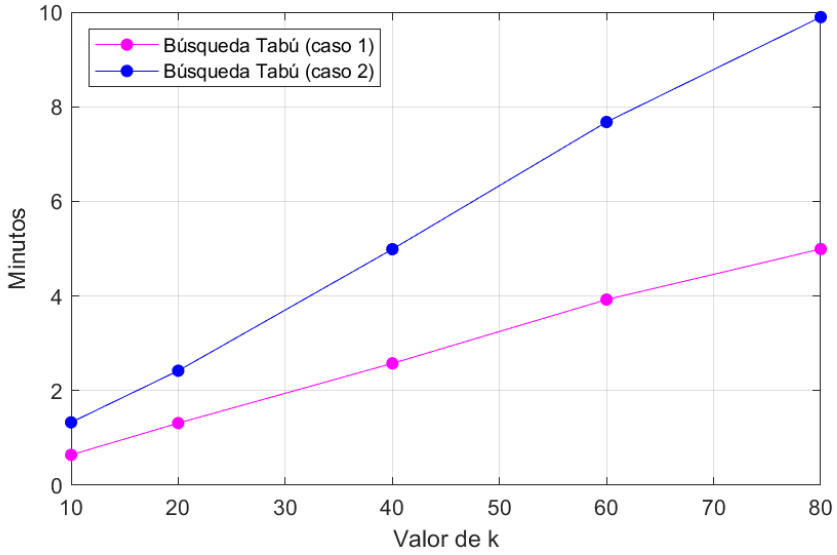


Figura 7.27: Tiempo total promedio para diferenes valores de k: Búsqueda Tabú (caso 1) vs. Búsqueda Tabú (caso 2).

Instancia	$k = 10$		$k = 20$		$k = 40$		$k = 60$		$k = 80$	
	U_{BT1}	U_{BT2}	U_{BT1}	U_{BT2}	U_{BT1}	U_{BT2}	U_{BT1}	U_{BT2}	U_{BT1}	U_{BT2}
1	29	35	31	33	31.5	32.5	16.5	47.5	28.5	35.5
2	27.5	36.5	30.5	33.5	26.5	37.5	25.5	38.5	25.5	38.5
3	34	30	29	35	21	43	22	42	25.5	38.5
4	39.5	24.5	30	34	29	35	22	42	24	40
5	43	21	29	35	21.5	42.5	13	51	21.5	42.5
6	37	27	28.5	35.5	21	43	21	43	20.5	43.5
7	35	29	22	42	22.5	41.5	25	39	19.5	44.5
8	48.5	15.5	31.5	32.5	21.5	42.5	18	46	30	34
9	32.5	31.5	28.5	35.5	25	39	23	41	30	34
10	39	25	24.5	39.5	25.5	38.5	27	37	21	43
11	28	36	32.5	31.5	30.5	33.5	29	35	27	37
12	33	31	24	40	30	34	27.5	36.5	24	40

Es claro que si nos basamos únicamente en los valores que se obtienen, no hay diferencia entre usar uno u otro caso de Búsqueda Tabú. Sin embargo, al tomar en cuenta el tiempo promedio que tardaron en obtener las mantas, recomendamos como opción al caso 1 de Búsqueda Tabú por encima del caso 2.

7.14. Búsqueda Tabú (caso 1) vs. Algoritmos Genéticos

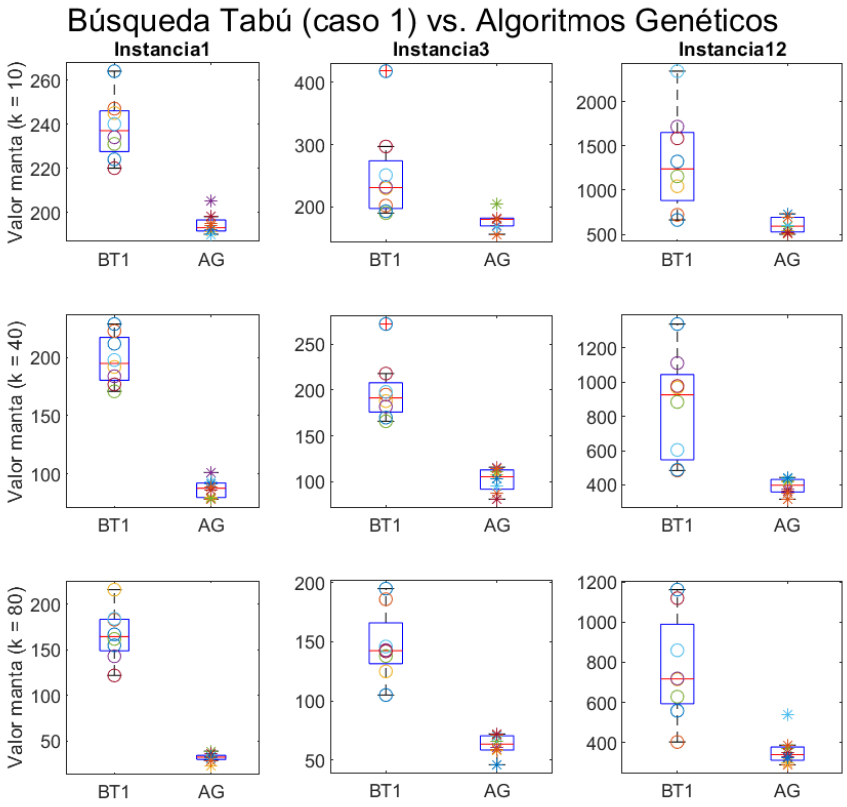


Figura 7.28: Diagramas de caja para Búsqueda Tabú (caso 1) y Algoritmos Genéticos.

Una vez más vemos la misma tendencia que hemos visto hasta el momento con Algoritmos Genéticos, por lo cual nuestra conjetura sigue estando a favor de Algoritmos Genéticos.

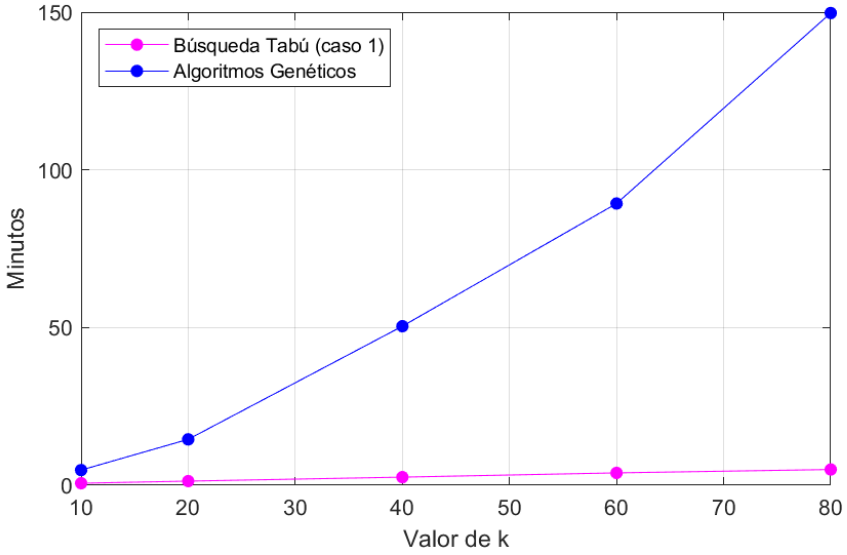


Figura 7.29: Tiempo total promedio para diferenes valores de k: Búsqueda Tabú (caso 1) vs. Algoritmos Genéticos.

Instancia	$k = 10$		$k = 20$		$k = 40$		$k = 60$		$k = 80$	
	U_{BT1}	U_{AG}	U_{BT1}	U_{AG}	U_{BT1}	U_{AG}	U_{BT1}	U_{AG}	U_{BT1}	U_{AG}
1	0	64	0	64	0	64	0	64	0	64
2	0	64	0	64	0	64	0	64	0	64
3	3	61	0	64	0	64	0	64	0	64
4	0	64	0	64	0	64	0	64	0	64
5	5.5	58.5	0	64	0	64	0	64	0	64
6	0	64	1	63	0	64	0	64	0	64
7	3	61	0	64	0	64	0	64	0	64
8	0	64	0	64	0	64	1	63	2	62
9	0	64	0	64	0	64	0	64	0	64
10	1	63	0	64	0	64	0	64	0	64
11	0	64	0	64	0	64	0	64	0	64
12	4	60	0	64	0	64	4	60	1	63

Una vez más vemos con claridad que Algoritmos Genéticos resulta mejor opción que Búsqueda Tabú.

7.15. Búsqueda Tabú (caso 2) vs. Algoritmos Genéticos

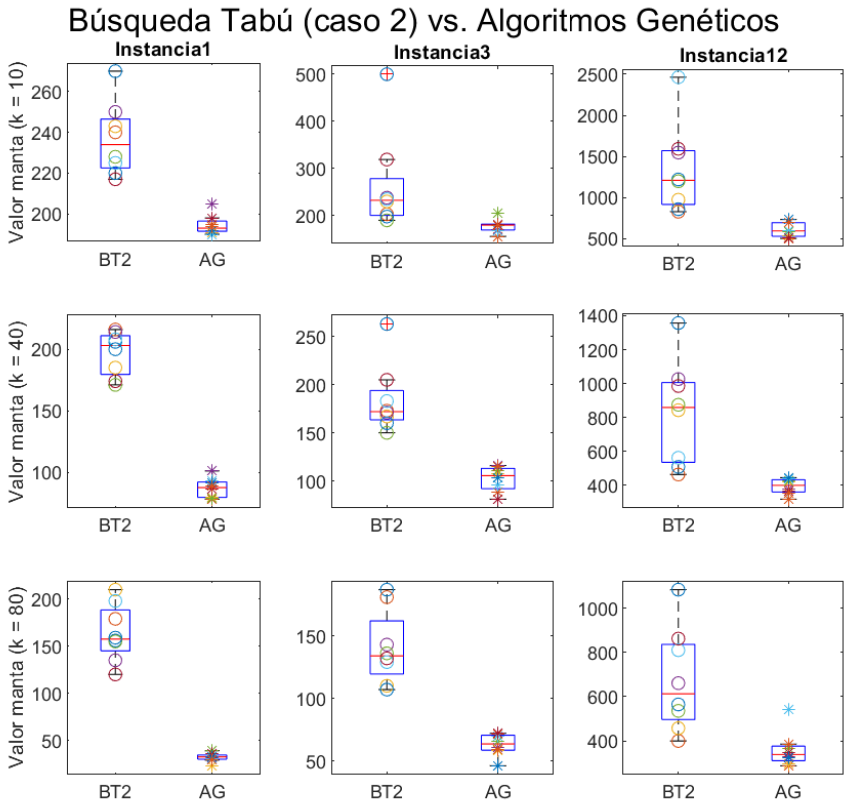


Figura 7.30: Diagramas de caja para Búsqueda Tabú (caso 2) y Algoritmos Genéticos.

Para esta última comparación no debe sorprender nuestra conjetura, la cual queda respaldada por los diagramas, la cual, como en todos los casos anteriores, es que Algoritmos Genéticos resultará ser mejor hurística.

Procedamos a ver si nuestra conjetura queda respaldada y sustentada estadísticamente.

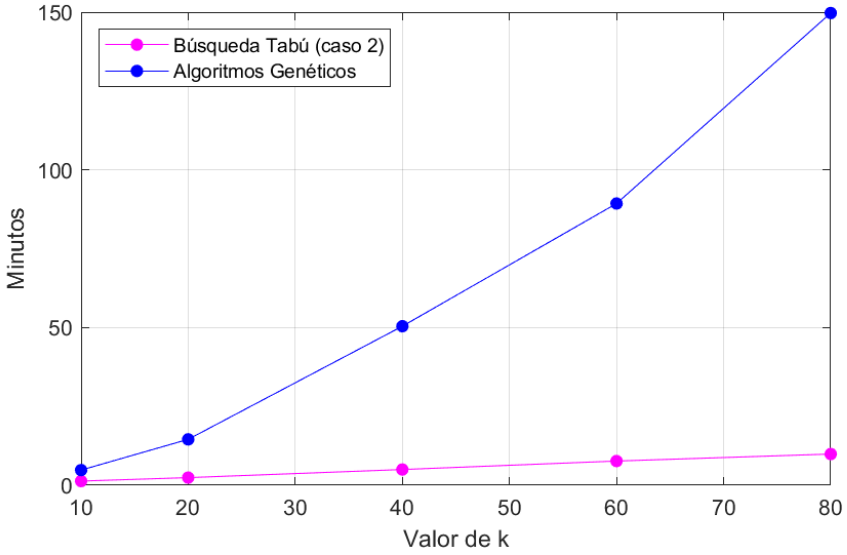


Figura 7.31: Tiempo total promedio para diferenes valores de k: Búsqueda Tabú (caso 2) vs. Algoritmos Genéticos.

Instancia	$k = 10$		$k = 20$		$k = 40$		$k = 60$		$k = 80$	
	U_{BT2}	U_{AG}	U_{BT2}	U_{AG}	U_{BT2}	U_{AG}	U_{BT2}	U_{AG}	U_{BT2}	U_{AG}
1	0	64	0	64	0	64	0	64	0	64
2	0	64	0	64	0	64	0	64	0	64
3	3	61	0	64	0	64	0	64	0	64
4	0	64	0	64	0	64	0	64	0	64
5	0	64	0	64	0	64	0	64	0	64
6	0	64	0	64	0	64	0	64	0	64
7	0	64	0	64	0	64	0	64	0	64
8	0	64	0	64	0	64	5	59	1	63
9	0	64	0	64	0	64	0	64	0	64
10	0	64	0	64	0	64	0	64	0	64
11	0	64	0	64	0	64	0	64	0	64
12	0	64	0	64	0	64	4	60	3	61

Finalmente, en esta última comparación, vemos que, de nuevo, Algoritmos Genéticos es mejor opción, lo cual queda sustendo estadísticamente por medio de lo recabado en la tabla presentada.

Capítulo 8

Una última implementación: Algoritmos Genéticos con Búsqueda Local Iterada

En el anterior capítulo comparamos por pares todas las heurísticas. Algoritmos Genéticos resultó mejor opción en todas las comparaciones, las cuales fueron estadísticamente significativas con un valor $\alpha = 0.05$.

Algoritmos Genéticos logra preservar las mejores características de los individuos de la población en cada generación, y busca tanto mejorar aún más estos rasgos como producir descendientes "*superadaptados*", los cuales conjunten las mejores características de sus ancestros.

Esta heurística asegurará construir una población de individuos "*superadaptados*", mas no que alguno de ellos logre ser óptimo local. Fue por ello que procedimos a analizar las mejores soluciones globales obtenidas por Algoritmos Genéticos, y notamos que en contadas ocasiones logramos obtener como mejor solución global un óptimo local. Esto último es la motivación del presente capítulo.

El objetivo ahora es conseguir llegar a un óptimo local de manera eficiente por medio de un posprocedimiento de búsqueda aplicado a las mejores soluciones globales obtenidas por medio de Algoritmos Genéticos. Para conseguirlo, primero se propusieron tres opciones: Búsqueda Local, Búsqueda Local Iterada y Recocido Simulado. Las primeras dos heurísticas nos aseguran llegar a un óptimo local, mientras que la última nos permite acercarnos lo más posible a alguno, llegando en ocasiones a ser una gran aproximación al óptimo global.

Es de suma importancia tener en mente que trabajaremos con las mejores soluciones globales obtenidas con Algoritmos Genéticos. Estas soluciones serán las soluciones iniciales de las heurísticas que usaremos como posprocedimiento. Por esta razón, no será necesaria una búsqueda exhaustiva en el espacio de soluciones. Por el contrario, como nos encontramos en una solución que muy probablemente se encuentre cerca o sea un óptimo local, requerimos movernos de manera rápida al mejor óptimo local cercano.

Búsqueda Local resulta ser la heurística más rápida de las tres, siguiéndole Búsqueda Local Iterada y finalmente Recocido Simulado. Esta última realiza una búsqueda exhaustiva con alta diversificación. Aún así, no nos asegura que lleguemos a un óptimo local. Estas cuestiones dan pie a su descarte.

Por otro lado, cada iteración de Búsqueda Local Iterada consta de dos fases, donde una busca diversificar la búsqueda, y la otra intensificarla. Así, tanto Búsqueda Local Iterada como Búsqueda Local aparecen como dos candidatos potenciales a cumplir los objetivos propuestos.

Por un lado, Búsqueda Local es una heurística que encuentra un óptimo local rápidamente;

por el otro lado, Búsqueda Local Iterada es una heurística que es capaz de encontrar múltiples óptimos locales, pues en cada iteración diversifica su búsqueda, y la solución que da como resultado es la mejor manta encontrada globalmente.

A continuación comparamos los resultados obtenidos de implementar estas dos heurísticas como posprocedimiento, no sin antes mencionar que en el caso de Búsqueda Local Iterada redujimos la cantidad de iteraciones que realizan ambas fases tras una intensa experimentación computacional. Anteriormente las repetía 30 veces, pero la nueva cantidad de repeticiones que le dimos fue 10.

8.1. ¿Algoritmos Genéticos con o sin posprocedimiento?

A continuación mostraremos en cuántas ocasiones encontró una mejor manta Algoritmos Genéticos con posprocedimiento.

Como comentamos, el posprocedimiento consiste en aplicar una heurística a la mejor solución global encontrada por Algoritmos Genéticos. Las dos heurísticas que probamos son Búsqueda Local y Búsqueda Local Iterada, y con cada una para distintos valores de k se corrieron 96 códigos, pues empleamos 12 instancias distintas, cada una con 8 semillas.

Las siguientes gráficas presentan los resultados obtenidos de cada una de las heurísticas implementadas como posprocedimiento. Se muestra de color azul la cantidad de veces que el posprocedimiento logró encontrar una mejor manta, mientras que de color rojo se nos dan a conocer las veces que no mejoró la solución.

Como observación, si Algoritmos Genéticos consigue llegar a un óptimo local entonces no tendrá caso usar Búsqueda Local, pues esta heurística únicamente cambia de solución si existe una manta en su vecindad que sea mejor que la actual. Por otro lado, Búsqueda Local Iterada es incapaz de mejorar la solución en una de estas dos situaciones: cuando no tiene éxito en la fase de diversificación de encontrar caminos que lleven a un óptimo local distinto o cuando no es capaz de encontrar un óptimo local que sea mejor que el actual.

De cualquier modo, gracias a las gráficas podremos notar que la probabilidad de que la mejor manta global encontrada por Algoritmos Genéticos sea un óptimo local disminuye cuando el valor de k crece, pues de lo contrario habría una mayor cantidad de mantas sin mejora para valores de k grandes.

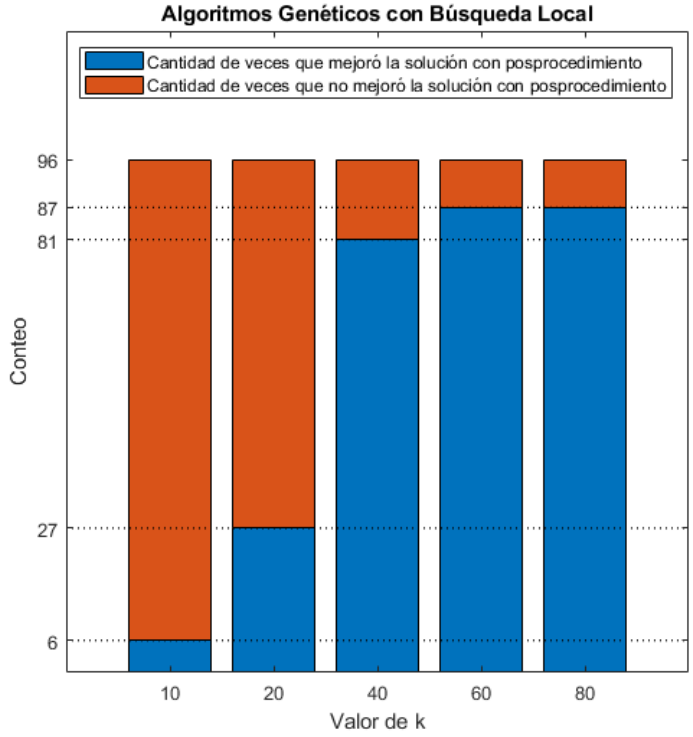


Figura 8.1: Algoritmos Genéticos con Búqueda Local como posprocedimiento

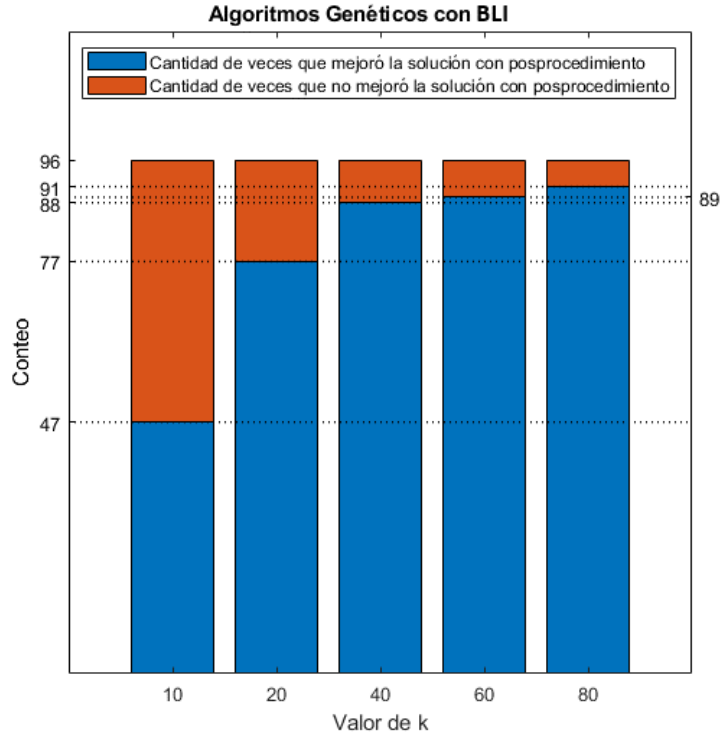


Figura 8.2: Algoritmos Genéticos con Búsqueda Local Iterada como posprocedimiento

Es claro que usando una u otra heurística como posprocedimiento es excelente opción incluso para valores pequeños de k.

Podríamos poner nuestra anterior afirmación en tela de juicio si el usar una heurística después de Algoritmos Genéticos conllevara a un aumento sustancial en el tiempo de espera, pero la siguiente gráfica responde a esta cuestión, pues aporta que no hay un considerable aumento de tiempo, pues usando Búsqueda Local, en promedio, aumenta en 1.9046 segundos cuando k toma valores grandes, mientras que con Búsqueda Local Iterada, en promedio, aumenta 68.1 segundos.

La gráfica muestra, para distintos valores de k, el tiempo promedio que tardó en terminar el posprocedimiento para diversas instancias empleando múltiples semillas.

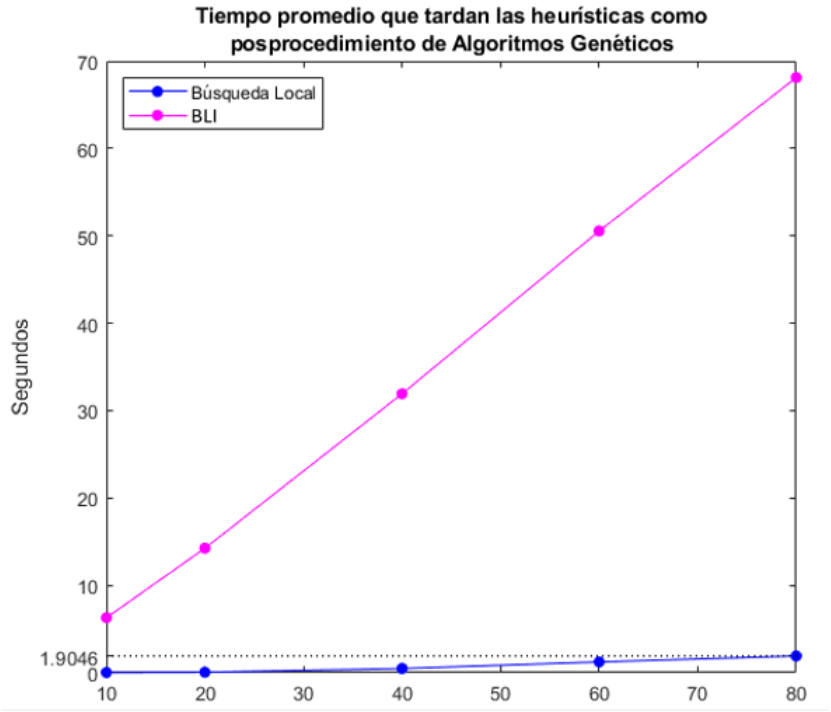


Figura 8.3: Tiempo promedio del posprocedimiento para Algoritmos Genéticos, ocupando Búsqueda Local y Búsqueda Local Iterada.

Ahora, nos parece mejor opción hacer uso de Búsqueda Local Iterada en lugar de Búsqueda Local por dos razones: primero, mejoró la solución en más ocasiones empleando Búsqueda Local Iterada que con Búsqueda Local, lo cual es más notorio al darle valores pequeños a k; segundo, en general las mantas obtenidas con Búsqueda Local Iterada como posprocedimiento resultaron iguales o mejores que las obtenidas con Búsqueda Local. Esto último se debe a la naturaleza de Búsqueda Local Iterada, pues itera múltiples búsquedas locales con el objetivo de mejorar en cada paso el último óptimo local visitado.

8.2. Resultados obtenidos con Algoritmos Genéticos y Búsqueda Local Iterada como posprocedimiento

En la anterior sección presentamos una gráfica de barras que mostró, para distintos valores de k, la cantidad de veces en las que Búsqueda Local Iterada encontró una mejor solución y en las que no fue capaz, al ser usada como posprocedimiento de Algoritmos Genéticos, en 96 corridas. Vimos que en un gran número de ocasiones encontró una mejor manta para el problema en cuestión, además de resultar muy rápido su cálculo.

Así, podemos darnos cuenta que Algoritmos Genéticos en conjunto con Búsqueda Local Iterada obtiene las mejores mantas para el problema de la manta rectangular.

La siguiente gráfica contiene cuántas unidades disminuyó el valor de la manta, al usar posprocedimiento para distintos valores de k, únicamente para los casos en los que hubo mejora. Recordemos el problema de la manta rectangular tiene por ojetivo encontrar la manta con valor mínimo.

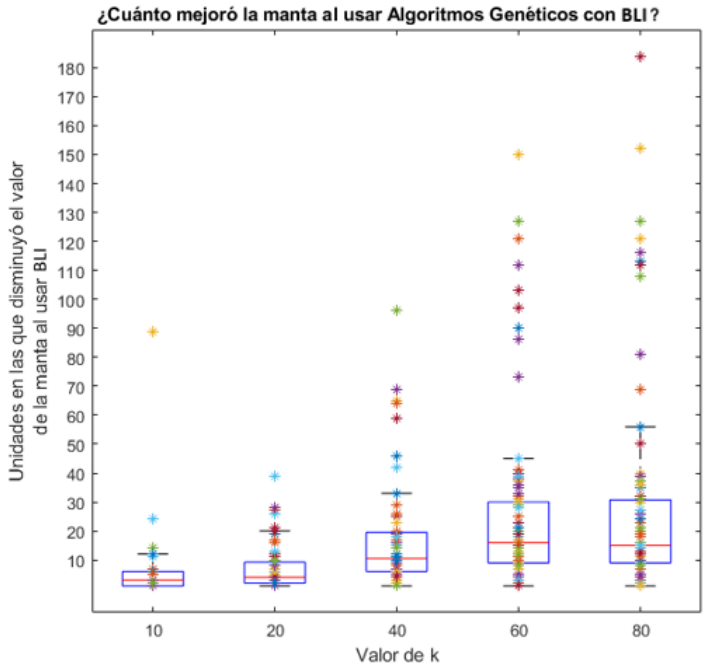
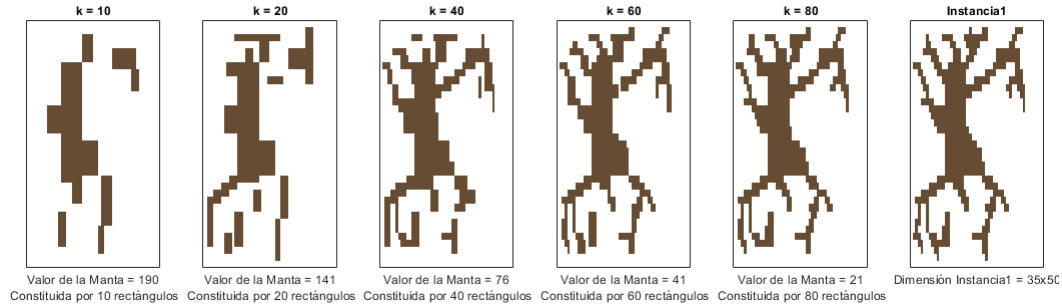


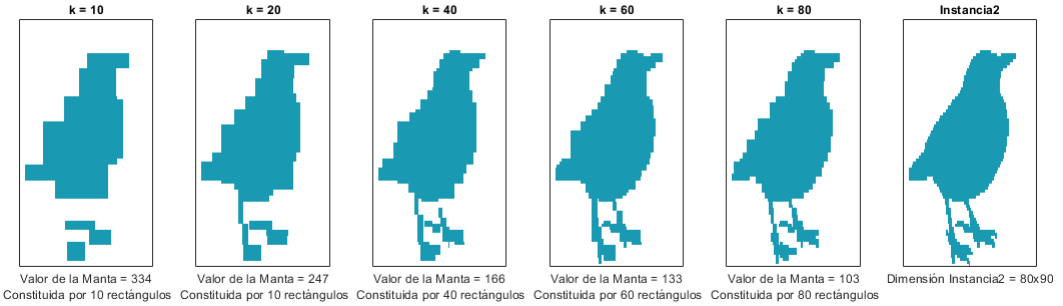
Figura 8.4: Unidades de mejora al usar Búsqueda Local Iterada como posprocedimiento de Algoritmos Genéticos.

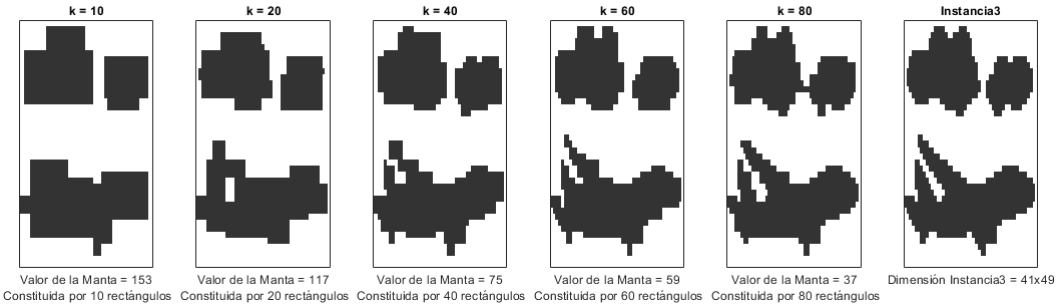
Podemos ver que en muchas ocasiones la disminución del valor no fue mucha, pero conforme el valor de k es mayor, también hay mayor margen de mejora, el cual busca cubrir el posprocedimiento.

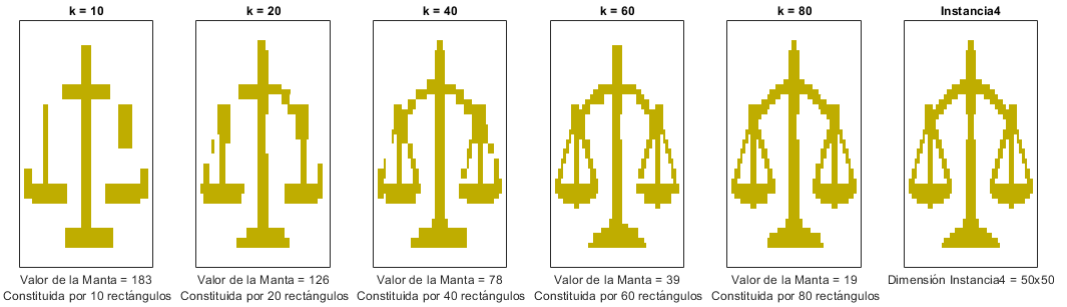
A continuación mostramos para las 12 instancias trabajadas la mejor manta que construyó[1] Algoritmos Genéticos con Búsqueda Local Iterada como posprocedimiento, para distintos valores de k. Podrá encontrar en ellas el valor de la manta y los rectángulos que la constituyen.

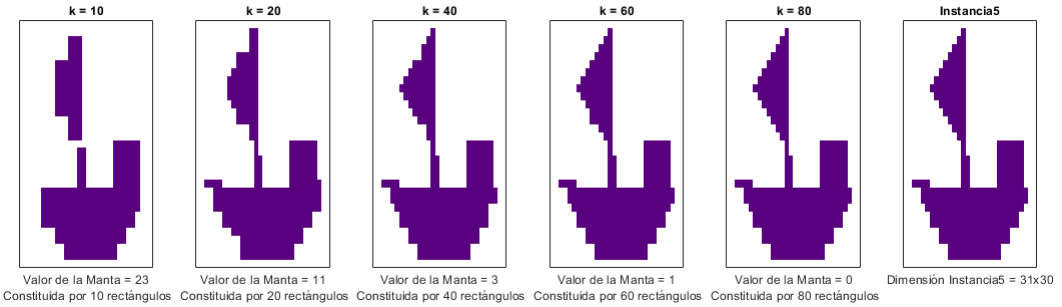
[1]Recordemos para cada manta empleamos 8 semillas distintas.

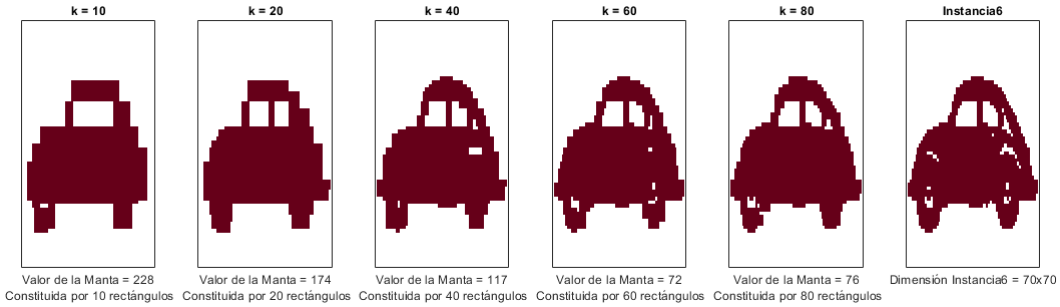

k = 10
k = 20
k = 40
k = 60
k = 80
Instancia1
Valor de la Manta = 190
Constituida por 10 rectángulos
Valor de la Manta = 141
Constituida por 20 rectángulos
Valor de la Manta = 76
Constituida por 40 rectángulos
Valor de la Manta = 41
Constituida por 60 rectángulos
Valor de la Manta = 21
Constituida por 80 rectángulos
Dimensión Instancia1 = 35x50


k = 10
k = 20
k = 40
k = 60
k = 80
Instancia2
Valor de la Manta = 334
Constituida por 10 rectángulos
Valor de la Manta = 247
Constituida por 10 rectángulos
Valor de la Manta = 166
Constituida por 40 rectángulos
Valor de la Manta = 133
Constituida por 60 rectángulos
Valor de la Manta = 103
Constituida por 80 rectángulos
Dimensión Instancia2 = 80x90

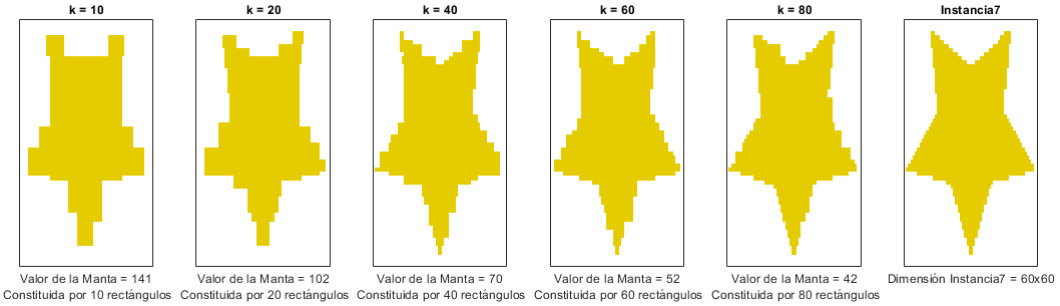

k = 10
k = 20
k = 40
k = 60
k = 80
Instancia3
Valor de la Manta = 153
Constituida por 10 rectángulos
Valor de la Manta = 117
Constituida por 20 rectángulos
Valor de la Manta = 75
Constituida por 40 rectángulos
Valor de la Manta = 59
Constituida por 60 rectángulos
Valor de la Manta = 37
Constituida por 80 rectángulos
Dimensión Instancia3 = 41x49


k = 10
k = 20
k = 40
k = 60
k = 80
Instancia4
Valor de la Manta = 183
Constituida por 10 rectángulos
Valor de la Manta = 126
Constituida por 20 rectángulos
Valor de la Manta = 78
Constituida por 40 rectángulos
Valor de la Manta = 39
Constituida por 60 rectángulos
Valor de la Manta = 19
Constituida por 80 rectángulos
Dimensión Instancia4 = 50x50

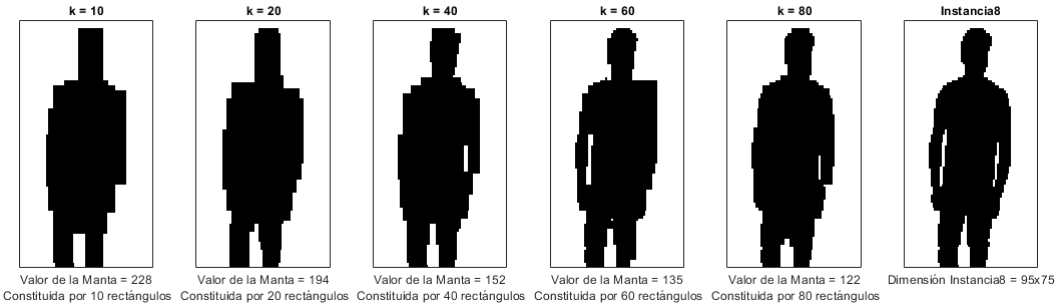

k = 10
k = 20
k = 40
k = 60
k = 80
Instancia5
Valor de la Manta = 23
Constituida por 10 rectángulos
Valor de la Manta = 11
Constituida por 20 rectángulos
Valor de la Manta = 3
Constituida por 40 rectángulos
Valor de la Manta = 1
Constituida por 60 rectángulos
Valor de la Manta = 0
Constituida por 80 rectángulos
Dimensión Instancia5 = 31x30



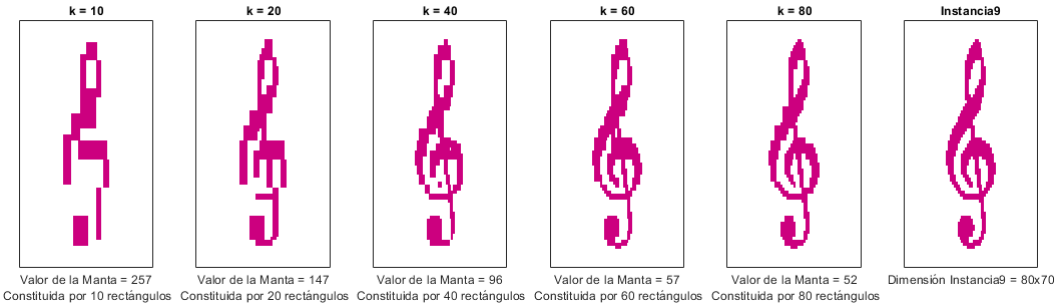
Valor de la Manta = 228 Constituida por 10 rectángulos | Valor de la Manta = 174 Constituida por 20 rectángulos | Valor de la Manta = 117 Constituida por 40 rectángulos | Valor de la Manta = 72 Constituida por 60 rectángulos | Valor de la Manta = 76 Constituida por 80 rectángulos | Dimensión Instancia6 = 70x70



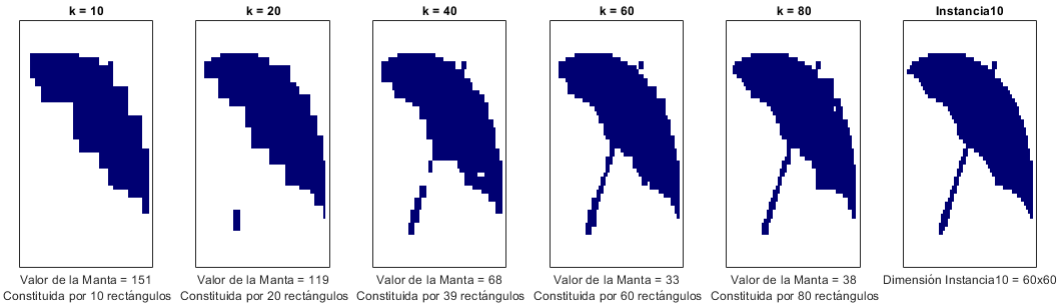
Valor de la Manta = 141 Constituida por 10 rectángulos | Valor de la Manta = 102 Constituida por 20 rectángulos | Valor de la Manta = 70 Constituida por 40 rectángulos | Valor de la Manta = 52 Constituida por 60 rectángulos | Valor de la Manta = 42 Constituida por 80 rectángulos | Dimensión Instancia7 = 60x60



Valor de la Manta = 228 Constituida por 10 rectángulos | Valor de la Manta = 194 Constituida por 20 rectángulos | Valor de la Manta = 152 Constituida por 40 rectángulos | Valor de la Manta = 135 Constituida por 60 rectángulos | Valor de la Manta = 122 Constituida por 80 rectángulos | Dimensión Instancia8 = 95x75



Valor de la Manta = 257 Constituida por 10 rectángulos | Valor de la Manta = 147 Constituida por 20 rectángulos | Valor de la Manta = 96 Constituida por 40 rectángulos | Valor de la Manta = 57 Constituida por 60 rectángulos | Valor de la Manta = 52 Constituida por 80 rectángulos | Dimensión Instancia9 = 80x70



Valor de la Manta = 151 Constituida por 10 rectángulos | Valor de la Manta = 119 Constituida por 20 rectángulos | Valor de la Manta = 68 Constituida por 39 rectángulos | Valor de la Manta = 33 Constituida por 60 rectángulos | Valor de la Manta = 38 Constituida por 80 rectángulos | Dimensión Instancia10 = 60x60

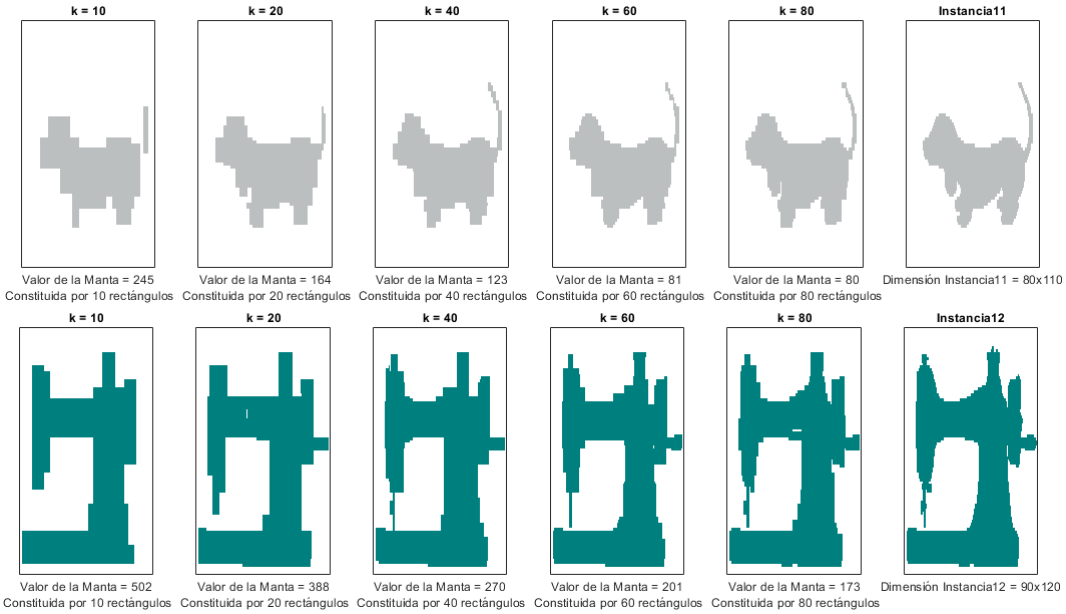


Figura 8.5: Mejores Mantas obtenidas para distintos valores de *k* usando Algoritmos Genéticos con Búsqueda Local Iterada como posprocedimiento.

Podemos notar que la implementación realizada obtuvo muy buenas aproximaciones a las imágenes a cubrir, por lo cual la mejor opción, dentro de las implementadas, fue Algoritmos Genéticos con apoyo de Búsqueda Local Iterada como posprocedimiento.

Capítulo 9

Conclusiones

Las heurísticas y las metaheurísticas son algoritmos capaces de encontrar buenas soluciones a cualquier problema de optimización combinatoria. Estos son usados principalmente cuando el problema a trabajar se encuentra en la categoría de problemas *NP*.

La calidad de las soluciones que encuentran las heurísticas y las metaheurísticas dependen de muy diversos factores, tales como la manera en la cual se implemente el algoritmo, cómo se especifiquen las vecindades, cómo se definan los movimientos de una solución a otra o el mismo problema a tratar, por mencionar algunos. Tal es el caso de nuestro tratamiento al Problema de la Manta Rectangular, pues discretizamos las imágenes por medio de un método raster, y consecutivamente definimos todo lo necesario para poder programar algunas heurísticas.

La metaheurística que construyó de manera consistente las mejores mantas fue Algoritmos Genéticos. A pesar de que esta metaheurística tarda considerablemente en comparación con cualquiera de las otras, logra cumplir el objetivo principal de encontrar las mejores mantas para cualquier valor de k, razón por la cual es la mejor heurística para el problema tratado.

Recocido Simulado es la segunda mejor opción para cualquier valor de k, pues a pesar de ser muy tardada logra encontrar mejores mantas que cualquier heurística a excepción de Algoritmos Genéticos.

Posteriormente le sigue Búsqueda Local Iterada (BLI), la cual logra encontrar muy buenas mantas en un tiempo considerable. Esta metaheurística, en general, encuentra mantas con valores parecidos a ambos casos de Búsqueda Tabú, pero a diferencia de estos BLI encuentra las soluciones de manera más rápida.

Para valores muy grandes de k BLI encontrará mantas con valores similares a Búsqueda Local. Por esta razón, cuando tenemos un valor de k muy alto (k>80) convendrá emplear Búsqueda Local en lugar de BLI debido a que es considerablemente más rápida.

Luego de BLI le siguen ambos casos de Búsqueda Tabú, para los cuales no hay mucha diferencia en las mantas que encuentran. Para valores de k que no sean altos lo mejor será usar el caso 1 de Búsqueda Tabú, pues encuentra mantas más rápido. Por otro lado, para valores altos de k podemos ver que aunque no haya diferencia estadísticamente significativa en los valores de las mantas, todos los estadísticos de prueba asociados al caso 2 son mayores que los asociados al caso 1, por lo cual podría ser probable que para valores mayores de k la diferencia de los valores de las mantas resulte significativa en favor del caso 2 de Búsqueda Tabú.

Finalmente tenemos a Búsqueda Local. Esta heurística, en general, no encuentra buenas soluciones (en comparación con las demás heurísticas) aunque su ventaja radica en que es la más rápida de todas.

En resumen, **Algoritmos Genéticos es la mejor opción**. Le sigue Recocido Simulado. Después BLI para valores no tan altos de k ($k <= 80$) y para valores muy grandes de este parámetro ($k > 80$) Búsqueda Local. Posteriormente, el caso 1 de Búsqueda Tabú para valores de k menores que 80 y probablemente para valores más grandes convendrá el caso 2 de Búsqueda Tabú. La peor es Búsqueda Local a pesar de su rapidez.

Posteriormente, procedimos a aplicar a las mejores mantas encontradas por Algoritmos Genéticos una heurística como posprocedimiento. Se tuvieron en cuenta Recocido Simulado, BLI y Búsqueda Local, debido a que la primera encuentra muy buenas mantas y las últimas por ser rápidas y porque aseguran llegar a un óptimo local. De ellas se descartó Recocido Simulado porque no necesitamos una búsqueda exhaustiva y porque no garantiza llegar a un óptimo local. Entre las restantes BLI resultó ser la mejor opción. Esta heurística tuvo la capacidad de encontrar diversos óptimos locales, elegir el mejor de ellos y realizarlo en un breve periodo de tiempo. De esta manera logró mejorar en gran parte de los casos y de manera sustancial la manta encontrada por Algoritmos Genéticos.

Así, la metaheurística Algoritmos Genéticos con la metaheurística Búsqueda Local Iterada como posprocedimiento es capaz de encontrar de manera eficaz y eficiente soluciones al Problema de la Manta Rectangular, en la manera en la cual fue implementado.

Las limitaciones de lo expuesto subyacen justamente en la manera en la cual tratamos el problema, ya que la calidad de las imágenes disminuye cuando son discretizadas con el método raster. Para contrarrestar este efecto debe realizarse una discretización más fina, es decir, una discretización que resulte en una mayor precisión de la representación de la imagen dos dimensional. El problema que ello acarrea es el incremento en el tiempo que tardará en terminar la heurística. No obstante, para cualquiera que sea el caso, **aquella que resultará mejor opción** por la calidad y consistencia de las mantas que encuentra y la eficiencia y eficacia en la cual lo hace, en comparación con las demás heurísticas implementadas, **será Algoritmos Genéticos con Búsqueda Local Iterada como posprocedimiento**.

Capítulo 10

Propuesta de aplicación: Detección de personas por medio de cámaras en un edificio.

En este último capítulo proponemos una aplicación del Problema de la Manta Rectangular.

Imaginemos que tenemos un edificio al cual únicamente permitimos el ingreso a personal autorizado. Para poder mantenerlo vigilado contamos con una gran cantidad de cámaras de alta resolución que tienen la capacidad de detectar adecuadamente la silueta de las personas. El problema que nos enfrentamos radica en que no contamos con suficiente espacio para almacenar todos los videos durante más de un día, por lo cual requerimos de alternativas que permitan almacenarlos durante más tiempo.

Una propuesta es guardarlos con menor resolución. El problema que ello conlleva es que, en dado caso que sea necesario revisar las grabaciones para poder dar una descripción precisa de algún sospechoso a las autoridades, la baja calidad de la imagen podría representar un enorme impedimento. Así, podemos optar por sacrificar fotogramas por segundo para poder mantener la resolución. Supongamos que decidimos que las cámaras graben a 4 fotogramas por segundo. Aún con lo anterior, el almacenamiento podría seguir siendo un problema, razón por la cual será necesario implementar otra estrategia.

Proponemos lo siguiente: Empezamos considerando un valor ligeramente mayor al máximo número de personas que se pueden encontrar en una habitación. A tal valor lo denotamos p.

Como las cámaras pueden detectar las siluetas de las personas y diferenciarlas de su entorno, se propone aplicar lo siguiente a cada uno de los fotogramas obtenidos:

1. Decidimos la cantidad máxima de rectángulos que nos gustaría emplear para aproximar la silueta de una persona. A tal cantidad la denotamos como m.
2. Se discretiza la imagen por medio del método raster. Esto se logra aprovechando que las cámaras son capaces de detectar y diferenciar las siluetas de las personas. Así, asignamos un 1 a los cuadros en los que se encuentren porciones de siluetas de personas y 0 en los que no. A la matriz obtenida la denominamos Imagen Objetivo (IO).
3. Convertimos lo anterior en un **Problema de la Manta Rectangular**, donde la IO será la matriz construida y $k = p * m$ será la cantidad máxima de rectángulos que puede tener la manta.

4. Implementamos una metaheurística para poder encontrar una buena manta.

5. Una vez encontrada, encimamos la matriz que representa a la manta construida en el fotograma original.

6. Finalmente, almacenamos únicamente las porciones de imagen que encierren los rectángulos.

Lo anterior permitirá guardar porciones de cada fotograma sin sacrificar resolución. Así, lograremos almacenar durante mayor tiempo porciones importantes de los videos.

Lo que ganamos al no extraer una representación perfecta de las siluetas es identificar el entorno y entender el contexto de donde se encuentran las personas, pues la manta no está restringida a cubrir totalmente las siluetas.

Por otro lado, si agregáramos la restricción de conseguir coberturas completas de las siluetas, podría aumentar el tamaño de cada fotograma y en consecuencia impedir que alcanzemos el objetivo planteado.

Un problema que enfrentamos con esta propuesta es que puede ocurrir que la manta no cubra partes importantes de algunos fotogramas, como por ejemplo alguna cara (ver la figura 10.1). Lo anterior podemos combatirlo variando la heurística (por ejemplo, alternando entre las heurísticas presentadas en este trabajo) y el valor de k en cada fotograma.

A continuación presentamos dos ejemplos, con dos fotos, que ilustran el procedimiento descrito.

Para la figura 10.1, supongamos deseamos emplear a lo más 5 rectángulos por persona, y el número máximo de personas permitidas en tal lugar es 6. Con esto en mente, fijamos el valor de $k = 6 * 5 = 30$. Luego, detectamos las siluetas a trabajar y las discretizamos con el método raster. Después, identificamos la IO y el valor de k y aplicamos la heurística (en este ejemplo se usó Algoritmos Genéticos con Búsqueda Local como posprocedimiento). Posteriormente, encimamos la manta encontrada en la fotografía original. Terminamos obteniendo únicamente la porción de la fotografía que cubre la manta.

En la figura 10.2, primero, decidimos la cantidad máxima de personas que se pueden encontrar en tal lugar. Supongamos decidimos que a lo más 5 personas pueden estar allí, y además deseamos emplear máximo 4 rectángulos por cada una. Así, $k = 5 * 4 = 20$. Ahora, se identifican las siluetas de las personas. Después, discretizamos la fotografía. Posteriormente, calculamos una manta rectangular (en este ejemplo también se empleó Algorimos Genéticos con Búsqueda Local como posprocedimiento). Una vez construida, ponemos la manta encima de la foto y extraemos aquellas porciones que se encuentren cubiertas por la manta. Finalmente, en este caso, podemos juntar los rectángulos encontrados y agrandar la imagen obtenida.

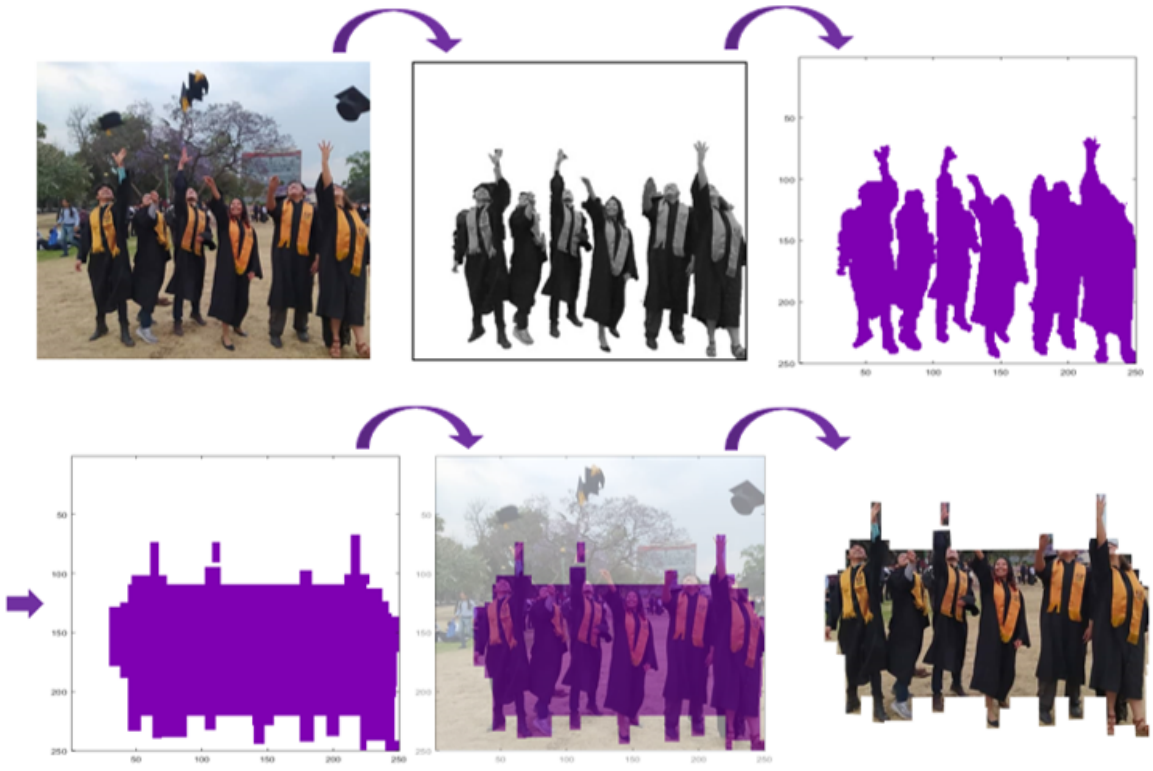


Figura 10.1: Primer ejemplo de aplicación

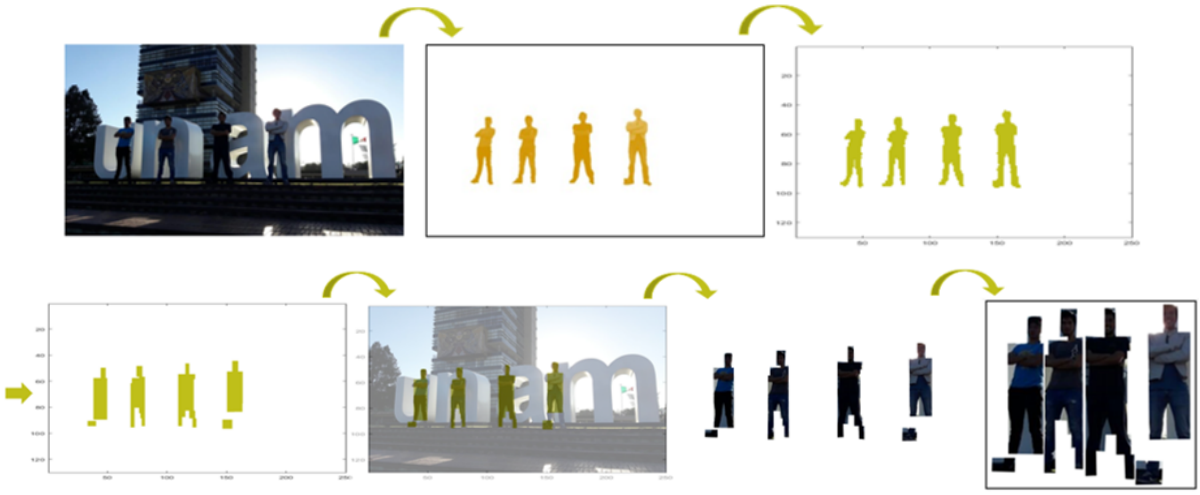


Figura 10.2: Segundo ejemplo de aplicación

Bibliografía

[1] Arora, S. y Barak, B. (2009). *Computational complexity: a modern approach*. Cambridge University Press.

[2] Beasley, D., Bull, D. R., y Martin, R. R. (1993). An overview of genetic algorithms: Part 1, fundamentals. *University computing*, 15(2):56–69.

[3] Cardillo, G. (2009). Mwwtest: Mann-whitney-wilcoxon non parametric test for two unpaired samples. http://www.mathworks.com/matlabcentral/fileexchange/25830.

[4] Chiarandini, M. y Stützle, T. (2002). An application of iterated local search to the graph coloring problem. In *Proceedings of the Computational Symposium on Graph Coloring and its Generalizations*, pages 112–125. Ithaca New York (USA).

[5] Corder, G. W., y Foreman, D. I. (2014). *Nonparametric statistics: A step-by-step approach*. John Wiley & Sons.

[6] Cormen, T. H., Leiserson, C. E., Rivest, R. L. y Stein, C. (2009). *Introduction to algorithms*. MIT press.

[7] de Antonio Suárez, O. (2011). Una aproximación a la heuristica y metaheuristicas. *INGE@ UAN-Tendencias en la Ingeniería*, 1(2).

[8] Demiröz, B. E. (2020). A balanced branching strategy for set packing problems.

[9] Demiröz, B. E., Altınel, İ. K. y Akarun, L. (2019). Rectangle blanket problem: Binary integer linear programming formulation and solution algorithms. *European Journal of Operational Research*, 277(1):62–83.

[10] Departamento de Matemáticas, CSI/ITESM. (2008). Recocido simulado.

[11] Downey, A. B. (2012). Think complexity, version 1.2.3.

[12] Eiselt, H. A. y Sandblom, C. -L. (2012). *Operations research: A model-based approach*. Springer Science & Business Media.

[13] Gestal, M., Rivero, D., Rabuñal, J. R., Dorado, J. y Pazos, A. (2010). *Introducción a los Algoritmos Genéticos y la Programación Genética*. Universidade da Coruña.

[14] Glover, F. W. y Kochenberger, Gary A. (2006). *Handbook of metaheuristics*, volume 57. Springer Science & Business Media.

[15] Karp, R. M. (1972). Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer.

[16] Kirby, M. W. (2008). Operations research in world war two: Its role in raf fighter command. *Military Operations Research*, pages 65–72.

[17] Lima Romero, M. J. (2000). Algoritmos genéticos aplicados al diseño de redes de telecomunicaciones.

[18] Liou, W. T., Tan, J. J.-M. y Lee, R. C. T. (1990). Minimum rectangular partition problem for simple rectilinear polygons. *IEEE transactions on computer-aided design of integrated circuits and systems*, 9(7):720–733.

[19] Lourenço, H. R., Martin, O. C. y Stützle, T. (2000). Iterated local search.

[20] Luss, H., y Rosenwein, M. B. (1997). Operations research applications: Opportunities and accomplishments. *European Journal of Operational Research*, 97(2):220–244.

[21] Mann, Z. Á. (2017). The top eight misconceptions about np-hardness. *Computer*, 50(5):72–79.

[22] Mee, J. F. (2021). Frederick W. Taylor. https://www.britannica.com/biography/Frederick-W-Taylor.

[23] Minguillón i Alfonso, J. (2010). *Complejidad algorítmica: Eficiencia de algoritmos y tipos abstractos de datos*. Editorial UOC.

[24] Oliveira, J. F. C. y Ferreira, J. A. S. (1993). Algorithms for nesting problems. In *Applied simulated annealing*, pages 255–273. Springer.

[25] Osman, I. H., y Kelly, J. P. (1996). Meta-heuristics: an overview. *Meta-heuristics*, pages 1–21.

[26] Riojas Cañari, A. C. (2005). Búsqueda tabú: conceptos, algoritmo y aplicación al problema de las n-reinas. *Universidad Nacional Mayor de San Marcos, Facultad de Ciencias Matemáticas. EAP. de Investigación Operativa*.

[27] Sandoya, F. (2013). Algoritmos factibles, problemas tratables y la complejidad computacional de una variante del problema de la diversidad máxima. *Matemática*, 11(2):41–46.

[28] Silva, C. y Malcata, E. (Sin fecha). Investigação operacional. documento de apoio das aulas teórico-práticas. https://moodle1819.up.pt/pluginfile.php/127168/mod_resource/content/3/Aulas%20Pr%C3%A1ticas%20IO_2018_2019.pdf.

[29] Singh, A., Gries, D., y Schneider, F. B. (2009). *Elements of computation theory*. Springer.

[30] Stützle, T. y Ruiz, R (2018). Iterated local search: A concise review.

[31] Vargas Paredes, J., y Penit Granado, V. (2016). Estudio y aplicación de metaheurísticas y comparación con métodos exhaustivos.

[32] Vaz Penna, P. H., Subramanian, A. y Satoru Ochi, L. (2013). An iterated local search heuristic for the heterogeneous fleet vehicle routing problem. *Journal of Heuristics*, 19(2):201–232.

Apéndice A

Código empleado

En la siguiente lista mostramos los códigos que usa cada heurística y metaheurística implementada. Por ejemplo, Búsqueda Local corresponde al código *BusquedaLocal_v2_VG*. Este hace uso de 3 funciones, los cuales a su vez hacen uso de otras funciones, tal como FuncionEliminaRectangulosNoAportan, la cual hace uso del código FuncionValorManta.

El símbolo "[...]" es con motivo de evitar escribir una y otra vez todas las funciones que emplea determinado código. Por ejemplo, en BLI podemos notarlo en "FuncionVecindadesGrandes [...]". Aquí, tal función hace uso de 9 funciones las cuales puede encontrar en lo respectivo a Búsqueda Local.

Posterior a la lista se encuentran todos los códigos. Estos se programaron en MATLAB.

Búsqueda Local

■ *BusquedaLocal_v2_VG*
↪ FuncionVecindadesGrandes
↪ FuncionAgrandarRectanguloArriba
↪ FuncionAgrandarRectanguloAbajo
↪ FuncionAgrandarRectanguloIzquierda
↪ FuncionAgrandarRectanguloDerecha
↪ FuncionAchicarRectanguloArriba
↪ FuncionAchicarRectanguloAbajo
↪ FuncionAchicarRectanguloIzquierda
↪ FuncionAchicarRectanguloDerecha
↪ FuncionVerificaFactibilidadManta
↪ FuncionValorManta
↪ FuncionEliminaRectangulosNoAportan
↪ FuncionValorManta

Búsqueda Local Iterada

■ *BLI_v2_VG*
↪ FuncionValorManta
↪ FuncionPerturbacion_VG

↪ FuncionVecindadesGrandes [...]
↪ FuncionValorManta
↪ FuncionBusquedaLocal_VG
↪ FuncionVecindadesGrandes [...]
↪ FuncionValorManta
↪ FuncionEliminaRectangulosNoAportan
↪ FuncionValorManta

Recocido Simulado

■ *RecocidoSimulado_v2_VG*
↪ FuncionValorManta
↪ FuncionVecindadesGrandes [...]
↪ FuncionEliminaRectangulosNoAportan
↪ FuncionValorManta

Búsqueda Tabú (caso 1)

■ *BusquedaTabuMCyLP_P_v2_VG_NEW2*
↪ FuncionValorManta
↪ FuncionVecindadesGrandesBT
↪ FuncionAgrandarRectanguloArriba
↪ FuncionAgrandarRectanguloAbajo
↪ FuncionAgrandarRectanguloIzquierda
↪ FuncionAgrandarRectanguloDerecha
↪ FuncionAchicarRectanguloArriba
↪ FuncionAchicarRectanguloAbajo
↪ FuncionAchicarRectanguloIzquierda
↪ FuncionAchicarRectanguloDerecha
↪ FuncionVerificaFactibilidadManta
↪ FuncionMejorVecino
↪ FuncionVecindadesGrandes [...]
↪ FuncionValorManta
↪ FuncionEncontrarMovimientoInversoDelQueSeHizo_Alterno
↪ FuncionVecindadesGrandesBT [...]
↪ FuncionCriterioDeAspiracion
↪ FuncionValorManta
↪ FuncionVecindadReducidaBT
↪ FuncionMejorEnVecindadReducidaMemoriaLargoPlazo_paraNEW2
↪ FuncionValorManta
↪ FuncionEliminaRectangulosNoAportan
↪ FuncionValorManta

Búsqueda Tabú (caso 2)

■ *BusquedaTabuMCyLP_GuardSolAleat_v2_VG_forNUEVO*
↪ FuncionValorManta
↪ FuncionVecindadesGrandes [...]
↪ FuncionMejorVecino
    ↪ FuncionVecindadesGrandes [...]
    ↪ FuncionValorManta
↪ FuncionEncontrarMovimientoInversoDelQueSeHizo_Alterno [...]
↪ FuncionVecindadesGrandesBT [...]
↪ FuncionCriterioDeAspiracion [...]
↪ FuncionVecindadReducidaBT
↪ FuncionMejorEnVecindadReducida_forNUEVO
    ↪ FuncionValorManta
↪ FuncionRoundMejorVecino
    ↪ FuncionVecindadesGrandes [...]
    ↪ FuncionValorManta
↪ FuncionBuscarTabuMCyLP_GuardSolAleat_forNUEVO
    ↪ FuncionValorManta
    ↪ FuncionVecindadesGrandes [...]
    ↪ FuncionMejorVecino [...]
    ↪ FuncionEncontrarMovimientoInversoDelQueSeHizo_Alterno [...]
    ↪ FuncionVecindadesGrandesBT [...]
    ↪ FuncionCriterioDeAspiracion [...]
    ↪ FuncionVecindadReducidaBT
    ↪ FuncionMejorEnVecindadReducida_forNUEVO [...]
    ↪ FuncionEncontrarMovimientoInversoDelQueSeHizo
        ↪ FuncionVecindadesGrandes [...]
↪ FuncionEliminaRectangulosNoAportan
    ↪ FuncionValorManta

Algoritmos Genéticos

■ *AlgoritmosGeneticos_v3_VG*
↪ FuncionSolucionAleatoria
    ↪ FuncionPosicionRectangulo
↪ FuncionValorManta
↪ FuncionProbabilidadesRuletaInvertirCol
    ↪ FuncionValorManta
↪ FuncionSeleccionIndividuosRuleta
↪ FuncionNuevoCruce
    ↪ FuncionValorManta
    ↪ FuncionVerificaFactibilidadSumaRectangulos

↪ FuncionSolucionAleatoria
↪ FuncionPosicionRectangulo
↪ FuncionPoblacionMutacion_v3
↪ FuncionVerificaFactibilidadManta
↪ FuncionEliminaRectangulosNoAportan
↪ FuncionValorManta

BusquedaLocal_v2_VG

```
function [tiempo,LaMejorSolucion,valorLaMejorSolucion, ...
    IteracionesComparaSolucion,IteracionesCambiaSolucion, ...
    IteracionesComparaSolucionGlobal,IteracionesCambiaSolucionGlobal, ...
    IteracionCambioSolucionGlobal,valorIteracionCambioSolucionGlobal, ...
    UnContadorDeIteraciones] = ...
    BusquedaLocal_v2_VG(M,SolucionInicial,seeed)
rng('default');
rng(seeed); %inicializamos semilla

    [A] = SolucionInicial; %Solucion Inicial
    k = length(A{2}); %numero rectangulos de A
    % -_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_

    tic;

[VecindadA] = FuncionVecindadesGrandes(A,k);
[valorA] = FuncionValorManta(M,A{1});

IteracionesCambiaSolucion = 0; %cuenta cantidad de veces que cambia ...
    de solucion
IteracionesComparaSolucion = 0; %cuenta cantidad de veces que ...
    compara soluciones
IteracionesCambiaSolucionGlobal = 0; %cuenta cantidad de veces que ...
    cambia a mejor solucion global
IteracionesComparaSolucionGlobal = 0; %cuenta cantidad de veces que ...
    compara la mejor solucion global
IteracionCambioSolucionGlobal = []; %guarda la iteracion en la que ...
    cambio mejor solucion global
valorIteracionCambioSolucionGlobal = []; %guarda valor mejor ...
    solucion global actualizada en iteracion donde mejoro

while 0 < length(VecindadA) %mientras la VecindadA sea no vacia
    B = randsample(VecindadA,1); %B es una solucion de VecindadA ...
        escogida aleatoriamente
    [valorB] = FuncionValorManta(M,B{1}{1});
    IteracionesComparaSolucion = IteracionesComparaSolucion + 1;
    if valorB < valorA %Si B mejor que A, actualizamos solucion, ...
        valor solucion y vecindad
        A = B{1};
        valorA = valorB;
        [VecindadA] = FuncionVecindadesGrandes(A,k);
        IteracionesCambiaSolucion = IteracionesCambiaSolucion + 1;
```

```
30          IteracionCambioSolucionGlobal = ...
               [IteracionCambioSolucionGlobal,IteracionesComparaSolucion];
31          valorIteracionCambioSolucionGlobal = ...
               [valorIteracionCambioSolucionGlobal,valorA];
32      else %if valorB ≥valorA %realizar lo siguiente: VecindadA = ...
           VecindadA - B
33              ActualLength = length(VecindadA);
34              i = 1;
35              while (ActualLength - 1) < length(VecindadA) %ciclo ...
                   encuentra y elimina B de VecindadA cuando B peor o ...
                   igual que A
36                  if VecindadA{i}{1} == B{1}{1}
37                      VecindadA(i) = [];
38                  else
39                      i = i+1;
40                  end
41              end
42      end
43  end
44
45  IteracionesCambiaSolucionGlobal = IteracionesCambiaSolucion;
46  IteracionesComparaSolucionGlobal = IteracionesComparaSolucion;
47  UnContadorDeIteraciones = IteracionesComparaSolucion;
48
49  LaMejorSolucion = FuncionEliminaRectangulosNoAportan(A,M,k);
50  valorLaMejorSolucion = FuncionValorManta(M,LaMejorSolucion{1});
51  tiempo = toc;
52
53  disp("--- Busqueda Local --- ")
54  disp("Tiempo que tarda en encontrar la solucion: " + tiempo)
55  disp("Valor de la mejor manta encontrada: " + valorLaMejorSolucion)
56
57  end
```

FuncionVecindadesGrandes

```
1   function [Vecindad] = FuncionVecindadesGrandes(Manta,valork)
2   % Funcion que devuelve las soluciones que estan en la vecindad de ...
       Manta junto con los rectangulos que componen a cada solucion:
3   % Vecindad es un Cell Array  con cada entrada compuesta de dos ...
       entrdas: 1ra solucion; 2da rectangulos
4
5   indiceVecindad = 1;
6   for i = 1:valork %del 1 hasta la cantidad de rectangulos
7       Movimientos = {"FuncionAgrandarRectanguloArriba", ...
            "FuncionAgrandarRectanguloAbajo", ...
            "FuncionAgrandarRectanguloIzquierda", ...
            "FuncionAgrandarRectanguloDerecha", ...
            "FuncionAchicarRectanguloArriba", ...
            "FuncionAchicarRectanguloAbajo", ...
            "FuncionAchicarRectanguloIzquierda", ...
            "FuncionAchicarRectanguloDerecha"};
```

```
8
9       for j = 1:8 %del 1 hasta el 8, donde 8 = #cantidad movimientos ...
            posibles de un rectangulo
10          RealizarMovimiento = str2func(Movimientos{j});
11          [RectanguloConMovimiento,SiNo] = ...
                RealizarMovimiento(Manta{2}{i});
12          if isequal(SiNo,1) %movimiento permitido
13              MatrizConstruida = Manta{1} - Manta{2}{i} + ...
                    RectanguloConMovimiento;
14              [factibleSiNo] = ...
                    FuncionVerificaFactibilidadManta(MatrizConstruida);
15              if isequal(factibleSiNo,1) %si Si es factible agrego la ...
                    manta encontrada a la vecindad de A
16                  Vecindad{indiceVecindad}{1} = MatrizConstruida;
17                  Vecindad{indiceVecindad}{2} = Manta{2};
18                  Vecindad{indiceVecindad}{2}{i} = ...
                        RectanguloConMovimiento;
19                  indiceVecindad = indiceVecindad + 1;
20              end
21          end
22      end
23  end
24
25  end
```

FuncionAgrandarRectanguloArriba

```
1   function [RectanguloAgrandadoArriba,SiNo] = ...
        FuncionAgrandarRectanguloArriba(Rectangulo)
2   % Funcion que agranda Rectangulo hacia arriba.
3   % Devuelve SiNo = 1 si fue posible agrandarlo o SiNo = 0 si no fue ...
        posible agrandarlo
4
5   [r1,r2] = size(Rectangulo); %dimension matriz Rectangulo
6   [a1,a2] = find(Rectangulo,1,'first'); %indices primer uno en Rectangulo
7   [b1,b2] = find(Rectangulo,1,'last'); %indices ultimo uno en Rectangulo
8
9   if isequal(a1,1) %si el rectangulo esta hasta arriba NO puede ...
        agrandarse. Se queda igual y devuelve SiNo=0
10      RectanguloAgrandadoArriba = Rectangulo;
11      SiNo = 0;
12
13  else %si el rectangulo no esta hasta arriba Si puede agrandarse. Se ...
        agranda y devuelve SiNo=1
14      CerosArr = zeros((a1-1)-1,r2);
15      CerosIzq = zeros(b1-(a1-1)+1,a2-1);
16      CerosDer = zeros(b1-(a1-1)+1,r2-b2);
17      CerosAba = zeros(r1-b1,r2);
18      UnosMed = ones(b1-(a1-1)+1,b2-a2+1);
19
20      M_Med = [CerosIzq,UnosMed,CerosDer];
21      RectanguloAgrandadoArriba = [CerosArr;M_Med;CerosAba];
```

```
22
23      SiNo = 1;
24  end
25
26  end
```

FuncionAgrandarRectanguloAbajo

```
1  function [RectanguloAgrandadoAbajo,SiNo] = ...
       FuncionAgrandarRectanguloAbajo(Rectangulo)
2  % Funcion que agranda Rectangulo hacia abajo.
3  % Devuelve SiNo = 1 si fue posible agrandarlo o SiNo = 0 si no fue ...
       posible agrandarlo
4
5  [r1,r2] = size(Rectangulo); %dimension matriz Rectangulo
6  [a1,a2] = find(Rectangulo,1,'first'); %indices primer uno en Rectangulo
7  [b1,b2] = find(Rectangulo,1,'last'); %indices ultimo uno en Rectangulo
8
9  if isequal(b1,r1) %si el rectangulo esta hasta abajo NO puede ...
       agrandarse. Se queda igual y devuelve SiNo=0
10     RectanguloAgrandadoAbajo = Rectangulo;
11     SiNo = 0;
12
13 else %si el rectangulo no esta hasta abajo Si puede agrandarse. Se ...
       agranda y devuelve SiNo=1
14     CerosArr = zeros(a1-1,r2);
15     CerosIzq = zeros((b1+1)-a1+1,a2-1);
16     CerosDer = zeros((b1+1)-a1+1,r2-b2);
17     CerosAba = zeros(r1-(b1+1),r2);
18     UnosMed = ones((b1+1)-a1+1,b2-a2+1);
19
20     M_Med = [CerosIzq,UnosMed,CerosDer];
21     RectanguloAgrandadoAbajo = [CerosArr;M_Med;CerosAba];
22
23     SiNo = 1;
24 end
25
26 end
```

FuncionAgrandarRectanguloDerecha

```
1  function [RectanguloAgrandadoDerecha,SiNo] = ...
       FuncionAgrandarRectanguloDerecha(Rectangulo)
2  % Funcion que agranda Rectangulo hacia la derecha.
3  % Devuelve SiNo = 1 si fue posible agrandarlo o SiNo = 0 si no fue ...
       posible agrandarlo
4
5  [r1,r2] = size(Rectangulo); %dimension matriz Rectangulo
6  [a1,a2] = find(Rectangulo,1,'first'); %indices primer uno en Rectangulo
```

```
7  [b1,b2] = find(Rectangulo,1,'last'); %indices ultimo uno en Rectangulo
8
9  if isequal(b2,r2) %si el rectangulo esta hasta la derecha NO puede ...
       agrandarse. Se queda igual y devuelve SiNo=0
10     RectanguloAgrandadoDerecha = Rectangulo;
11     SiNo = 0;
12
13 else %si el rectangulo no esta hasta la derecha Si puede agrandarse. ...
       Se agranda y devuelve SiNo=1
14     CerosArr = zeros(a1-1,r2);
15     CerosIzq = zeros(b1-a1+1,a2-1);
16     CerosDer = zeros(b1-a1+1,r2-(b2+1));
17     CerosAba = zeros(r1-b1,r2);
18     UnosMed = ones(b1-a1+1,(b2+1)-a2+1);
19
20     M_Med = [CerosIzq,UnosMed,CerosDer];
21     RectanguloAgrandadoDerecha = [CerosArr;M_Med;CerosAba];
22
23     SiNo = 1;
24 end
25
26 end
```

FuncionAgrandarRectanguloIzquierda

```
1  function [RectanguloAgrandadoIzquierda,SiNo] = ...
       FuncionAgrandarRectanguloIzquierda(Rectangulo)
2  % Funcion que agranda Rectangulo hacia la izquierda.
3  % Devuelve SiNo = 1 si fue posible agrandarlo o SiNo = 0 si no fue ...
       posible agrandarlo
4
5  [r1,r2] = size(Rectangulo); %dimension matriz Rectangulo
6  [a1,a2] = find(Rectangulo,1,'first'); %indices primer uno en Rectangulo
7  [b1,b2] = find(Rectangulo,1,'last'); %indices ultimo uno en Rectangulo
8
9  if isequal(a2,1) %si el rectangulo esta hasta la izquierda NO puede ...
       agrandarse. Se queda igual y devuelve SiNo=0
10     RectanguloAgrandadoIzquierda = Rectangulo;
11     SiNo = 0;
12
13 else %si el rectangulo no esta hasta la izquierda Si puede ...
       agrandarse. Se agranda y devuelve SiNo=1
14     CerosArr = zeros(a1-1,r2);
15     CerosIzq = zeros(b1-a1+1,(a2-1)-1);
16     CerosDer = zeros(b1-a1+1,r2-b2);
17     CerosAba = zeros(r1-b1,r2);
18     UnosMed = ones(b1-a1+1,b2-(a2-1)+1);
19
20     M_Med = [CerosIzq,UnosMed,CerosDer];
21     RectanguloAgrandadoIzquierda = [CerosArr;M_Med;CerosAba];
22
23     SiNo = 1;
```

```
24 end
25
26 end
```

FuncionAchicarRectanguloArriba

```
1  function [RectanguloAchicadoArriba,SiNo] = ...
       FuncionAchicarRectanguloArriba(Rectangulo)
2  % Funcion que achica Rectangulo desde arriba.
3  % Devuelve SiNo = 1 si fue posible achicarlo o SiNo = 0 si no fue ...
       posible achicarlo
4
5  [r1,r2] = size(Rectangulo); %dimension matriz Rectangulo
6  [a1,a2] = find(Rectangulo,1,'first'); %indices primer uno en Rectangulo
7  [b1,b2] = find(Rectangulo,1,'last'); %indices ultimo uno en Rectangulo
8
9  if isequal(a1,b1) %si el rectangulo es de dimension 1xm NO puede ...
       achicarse. Se queda igual y devuelve SiNo=0
10     RectanguloAchicadoArriba = Rectangulo;
11     SiNo = 0;
12
13 else %si el rectangulo no es de dimension 1xm Si puede achicarse. Se ...
       achica y devuelve SiNo=1
14     CerosArr = zeros((a1+1)-1,r2);
15     CerosIzq = zeros(b1-(a1+1)+1,a2-1);
16     CerosDer = zeros(b1-(a1+1)+1,r2-b2);
17     CerosAba = zeros(r1-b1,r2);
18     UnosMed = ones(b1-(a1+1)+1,b2-a2+1);
19
20     M_Med = [CerosIzq,UnosMed,CerosDer];
21     RectanguloAchicadoArriba = [CerosArr;M_Med;CerosAba];
22
23     SiNo = 1;
24 end
25
26 end
```

FuncionAchicarRectanguloAbajo

```
1  function [RectanguloAchicadoAbajo,SiNo] = ...
       FuncionAchicarRectanguloAbajo(Rectangulo)
2  % Funci n que achica Rect ngulo desde abajo.
3  % Devuelve SiNo = 1 si fue posible achicarlo o SiNo = 0 si no fue ...
       posible achicarlo
4
5  [r1,r2] = size(Rectangulo); %dimensi n matriz Rect ngulo
6  [a1,a2] = find(Rectangulo,1,'first'); % ndices  primer uno en ...
       Rect ngulo
7  [b1,b2] = find(Rectangulo,1,'last'); % ndices   ltimo  uno en ...
```

```
    Rect ngulo
8
9  if isequal(a1,b1) %si el rect ngulo es de dimensi n 1xm NO puede ...
       achicarse. Se queda igual y devuelve SiNo=0
10     RectanguloAchicadoAbajo = Rectangulo;
11     SiNo = 0;
12
13 else %si el rect ngulo no es de dimensi n 1xm  S   puede achicarse. ...
       Se achica y devuelve SiNo=1
14     CerosArr = zeros(a1-1,r2);
15     CerosIzq = zeros((b1-1)-a1+1,a2-1);
16     CerosDer = zeros((b1-1)-a1+1,r2-b2);
17     CerosAba = zeros(r1-(b1-1),r2);
18     UnosMed = ones((b1-1)-a1+1,b2-a2+1);
19
20     M_Med = [CerosIzq,UnosMed,CerosDer];
21     RectanguloAchicadoAbajo = [CerosArr;M_Med;CerosAba];
22
23     SiNo = 1;
24 end
25
26 end
```

FuncionAchicarRectanguloIzquierda

```
1  function [RectanguloAchicadoIzquierda,SiNo] = ...
       FuncionAchicarRectanguloIzquierda(Rectangulo)
2  % Funcion que achica Rectangulo desde la izquierda.
3  % Devuelve SiNo = 1 si fue posible achicarlo o SiNo = 0 si no fue ...
       posible achicarlo
4
5  [r1,r2] = size(Rectangulo); %dimension matriz Rectangulo
6  [a1,a2] = find(Rectangulo,1,'first'); %indices primer uno en Rectangulo
7  [b1,b2] = find(Rectangulo,1,'last'); %indices ultimo uno en Rectangulo
8
9  if isequal(a2,b2) %si el rectangulo es de dimension nx1 NO puede ...
       achicarse. Se queda igual y devuelve SiNo=0
10     RectanguloAchicadoIzquierda = Rectangulo;
11     SiNo = 0;
12
13 else %si el rectangulo no es de dimension nx1 Si puede achicarse. Se ...
       achica y devuelve SiNo=1
14     CerosArr = zeros(a1-1,r2);
15     CerosIzq = zeros(b1-a1+1,(a2+1)-1);
16     CerosDer = zeros(b1-a1+1,r2-b2);
17     CerosAba = zeros(r1-b1,r2);
18     UnosMed = ones(b1-a1+1,b2-(a2+1)+1);
19
20     M_Med = [CerosIzq,UnosMed,CerosDer];
21     RectanguloAchicadoIzquierda = [CerosArr;M_Med;CerosAba];
22
23     SiNo = 1;
24 end
25
26 end
```

FuncionAchicarRectanguloDerecha

```
function [RectanguloAchicadoDerecha,SiNo] = ...
    FuncionAchicarRectanguloDerecha(Rectangulo)
% Funcion que achica Rectangulo desde la derecha.
% Devuelve SiNo = 1 si fue posible achicarlo o SiNo = 0 si no fue ...
    posible achicarlo

[r1,r2] = size(Rectangulo); %dimension matriz Rectangulo
[a1,a2] = find(Rectangulo,1,'first'); %indices primer uno en Rectangulo
[b1,b2] = find(Rectangulo,1,'last'); %indices ultimo uno en Rectangulo

if isequal(a2,b2) %si el rectangulo es de dimension nx1 NO puede ...
    achicarse. Se queda igual y devuelve SiNo=0
     RectanguloAchicadoDerecha = Rectangulo;
     SiNo = 0;

else %si el rectangulo no es de dimension nx1 Si puede achicarse. Se ...
    achica y devuelve SiNo=1
     CerosArr = zeros(a1-1,r2);
     CerosIzq = zeros(b1-a1+1,a2-1);
     CerosDer = zeros(b1-a1+1,r2-(b2-1));
     CerosAba = zeros(r1-b1,r2);
     UnosMed = ones(b1-a1+1,(b2-1)-a2+1);

     M_Med = [CerosIzq,UnosMed,CerosDer];
     RectanguloAchicadoDerecha = [CerosArr;M_Med;CerosAba];

     SiNo = 1;
end

end
```

FuncionVerificaFactibilidadManta

```
function [factibleSiNo] = FuncionVerificaFactibilidadManta(Manta)
% Funcion que verifica la factibilidad de Manta ya construida. La ...
    manera de hacerlo es buscar si tiene alguna
% entrada mayor o igual a 2. Si la tiene, entonces NO es factible, ...
    si todas son 0 o 1 Si es factible.

factibilidad = find(Manta≥2,1);
if (factibilidad > 0)
     factibleSiNo = 0;
else
     factibleSiNo = 1;
end
```

```
11
12 end
```

FuncionValorManta

```
1  function [value] = FuncionValorManta(ImagenObjetivo,Manta)
2  % Funcion asigna valor a Manta respecto a la ImagenObjetivo.
3
4  value = sum(sum(abs(Manta-ImagenObjetivo)));
5
6  end
```

FuncionEliminaRectangulosNoAportan

```
1  function [MantaNueva] = FuncionEliminaRectangulosNoAportan(Manta,IO,k)
2  % Esta funcion elimina de la Manta los rectangulos que perjudican su ...
       valor
3  % (trabaja con CellArray (1er entrada Manta, 2da entrada ...
       rectangulos) y no con la matriz Manta
4
5  MantaNueva = Manta; %iniciamos con la misma Manta
6  i = k; %contador que va evaluando los rectangulos
7  while i>0
8      [valorMantaNueva] = FuncionValorManta(IO,MantaNueva{1}); %valor ...
           de MantaNueva
9      MantaPrueba = MantaNueva; %igualamos MantaPrueba con MantaNueva
10     MantaPrueba{2}(i) = []; %eliminamos el i-esimo rectangulo a ...
           MantaPrueba
11     MantaPrueba{1} = zeros(size(IO)); %matriz de zeros dimension = ...
           dim(IO)
12     for j = 1:length(MantaPrueba{2}) %sumamos los rectangulos de ...
           MantaPrueba
13         MantaPrueba{1} = MantaPrueba{1} + MantaPrueba{2}{j};
14     end
15     [valorMantaPrueba] = FuncionValorManta(IO,MantaPrueba{1});
16     if valorMantaPrueba < valorMantaNueva %si el valor sin el ...
           i-esimo rectangulo es mejor que con el
17         MantaNueva = MantaPrueba;
18     end
19     i = i-1;
20 end
21
22 end
```

BLI_v2_VG

```matlab
function [tiempo,LaMejorSolucion,valorLaMejorSolucion, ...
    IteracionesComparaSolucion,IteracionesCambiaSolucion, ...
    IteracionesComparaSolucionGlobal,IteracionesCambiaSolucionGlobal, ...
    IteracionCambioSolucionGlobal,valorIteracionCambioSolucionGlobal, ...
    i] = BLI_v2_VG(M,SolucionInicial,seeed)
rng('default');
rng(seeed); %inicializamos semilla

     [A] = SolucionInicial; %Solucion Inicial
     k = length(A{2}); %numero rectangulos de A
     % -_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_

     tic;

iteracionesAmbasFases = 30; %ParametroQuePodemosModificar
iteracionesFasePerturbacion = max(size(M)); ...
    %ParametroQuePodemosModificar

[valorA] = FuncionValorManta(M,A{1});
C = A;
valorC = valorA;

IteracionesCambiaSolucion = 0; %cuenta cantidad de veces que cambia ...
    de solucion
IteracionesComparaSolucion = 0; %cuenta cantidad de veces que ...
    compara soluciones
IteracionesCambiaSolucionGlobal = 0; %cuenta cantidad de veces que ...
    cambia a mejor solucion global
IteracionesComparaSolucionGlobal = 0; %cuenta cantidad de veces que ...
    compara la mejor solucion global
IteracionCambioSolucionGlobal = []; %guarda la iteracion en la que ...
    cambio mejor solucion global
valorIteracionCambioSolucionGlobal = []; %guarda valor mejor ...
    solucion global actualizada en iteracion donde mejoro

for i = 1:iteracionesAmbasFases

     %%% FASE DE PERTURBACION
     for j = 1:iteracionesFasePerturbacion
          [A,valorA] = FuncionPerturbacion_VG(M,A,k); ...
              %ParametroQuePodemosModificar
          IteracionesCambiaSolucion = IteracionesCambiaSolucion + 1;
          IteracionesComparaSolucion = IteracionesComparaSolucion + 1;
     end

     %%% FASE DE MEJORA
     [A,valorA,IteracionesCambiaSolucion,IteracionesComparaSolucion]= ...
         FuncionBusquedaLocal_VG(M,A,k,IteracionesCambiaSolucion, ...
         IteracionesComparaSolucion);

     IteracionesComparaSolucionGlobal = ...
         IteracionesComparaSolucionGlobal + 1;
     if (valorA < valorC)
          C = A;
          valorC = valorA;
```

```
            IteracionesCambiaSolucionGlobal = ...
                IteracionesCambiaSolucionGlobal + 1;
            IteracionCambioSolucionGlobal = ...
                [IteracionCambioSolucionGlobal,i];
            valorIteracionCambioSolucionGlobal = ...
                [valorIteracionCambioSolucionGlobal,valorC];
        end

    end

    LaMejorSolucion = FuncionEliminaRectangulosNoAportan(C,M,k);
    valorLaMejorSolucion = FuncionValorManta(M,LaMejorSolucion{1});
    tiempo = toc;

    disp("--- BLI --- ")
    disp("Tiempo que tarda en encontrar la solucion: " + tiempo)
    disp("Valor de la mejor manta encontrada: " + valorLaMejorSolucion)

    end
```

FuncionPerturbacion_VG

```
function [B,valorB] = FuncionPerturbacion_VG(M,A,k)

[VecindadA] = FuncionVecindadesGrandes(A,k);
[aa,bb] = size(VecindadA);
MatrizDeValores = [];
MatrizDeVecinos = [];

for i=1:bb
     [valorVecinoAi] = FuncionValorManta(M,VecindadA{i}{1});
     MatrizDeVecinos = [MatrizDeVecinos;i];
     MatrizDeValores = [MatrizDeValores;valorVecinoAi];
end
MatrizConstruida = [MatrizDeVecinos,MatrizDeValores];

% % % % % En esta parte ordenamos respecto a la segunda columna % % %
[gamma,kappa] = sort(MatrizConstruida(:,2));
MatrizOrdenadaValorAscendente = [MatrizConstruida(kappa),gamma];
% % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % %

ListaRestringida = {};
CantidadVecinosListaRestringida = round(bb/2); ...
    %ParametroQuePodemosModificar
for i=1:CantidadVecinosListaRestringida
     ListaRestringida{i} = VecindadA{MatrizOrdenadaValorAscendente(i,1)};
end

B1 = randsample(ListaRestringida,1); %definimos a B como una ...
    solucion vecina de A en ListaRestringida elegida aleatoriamente
B = B1{1};
[valorB] = FuncionValorManta(M,B{1});
```

```
29
30 end
```

FuncionBusquedaLocal_VG

```
1  function ...
       [A,valorA,IteracionesCambiaSolucion,IteracionesComparaSolucion] = ...
       FuncionBusquedaLocal_VG(M,A1,k,IteracionesCambiaSolucion, ...
       IteracionesComparaSolucion)
2
3  A = A1;
4  [VecindadA] = FuncionVecindadesGrandes(A,k);
5  [valorA] = FuncionValorManta(M,A{1});
6
7  while 0 < length(VecindadA)
8      B = randsample(VecindadA,1);
9      [valorB] = FuncionValorManta(M,B{1}{1});
10     IteracionesComparaSolucion = IteracionesComparaSolucion + 1;
11     if valorB < valorA
12         A = B{1};
13         valorA = valorB;
14         [VecindadA] = FuncionVecindadesGrandes(A,k);
15         IteracionesCambiaSolucion = IteracionesCambiaSolucion + 1;
16     else %if valorB ≥valorA realizar lo siguiente: VecindadA = ...
           VecindadA - B
17             ActualLength = length(VecindadA);
18             i = 1;
19             while (ActualLength - 1) < length(VecindadA)
20                 if VecindadA{i}{1} == B{1}{1}
21                     VecindadA(i) = [];
22                 else
23                     i = i+1;
24                 end
25             end
26     end
27 end
28
29 end
```

RecocidoSimulado_v2_VG

```
1  function [tiempo,LaMejorSolucion,valorLaMejorSolucion, ...
       IteracionesComparaSolucion,IteracionesCambiaSolucion, ...
       IteracionesComparaSolucionGlobal,IteracionesCambiaSolucionGlobal, ...
       IteracionCambioSolucionGlobal,valorIteracionCambioSolucionGlobal, ...
       UnContadorDeIteraciones] = ...
       RecocidoSimulado_v2_VG(M,SolucionInicial,seeed)
2  rng('default');
3  rng(seeed); %inicializamos semilla
```

```

    [A] = SolucionInicial; %Solucion Inicial
    k = length(A{2}); %numero rectangulos de A
    % -_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_

    tic;

T = 100; %ParametroQuePodemosModificar
Tfinal = 0.005; %ParametroQuePodemosModificar
alpha = 0.995; %ParametroQuePodemosModificar

[valorA] = FuncionValorManta(M,A{1});
C = A;
valorC = valorA;

IteracionesCambiaSolucion = 0; %cuenta cantidad de veces que cambia ...
    de solucion
IteracionesComparaSolucion = 0; %cuenta cantidadd de veces que ...
    compara soluciones
IteracionesCambiaSolucionGlobal = 0; %cuenta cantidad de veces que ...
    cambia a mejor solucion global
IteracionesComparaSolucionGlobal = 0; %cuenta cantidad de veces que ...
    compara la mejor solucion global
IteracionCambioSolucionGlobal = []; %guarda la iteracion en la que ...
    cambio mejor solucion global
valorIteracionCambioSolucionGlobal = []; %guarda valor mejor ...
    solucion global actualizada en iteracion donde mejoro
UnContadorDeIteraciones = 0; %nos ayuda a ver la iteracion en la que ...
    nos encontramos (sirve para calcular IteracionCambioSolucionGlobal)

while (T > Tfinal)
    UnContadorDeIteraciones = UnContadorDeIteraciones + 1;
    for i=1:20 %ParametroQuePodemosModificar
        [VecindadA] = FuncionVecindadesGrandes(A,k);
        B = randsample(VecindadA,1);
        [valorB] = FuncionValorManta(M,B{1}{1}); %valor de la manta B
        IteracionesComparaSolucion = IteracionesComparaSolucion + 1;

        if valorB < valorA
            A = B{1};
            valorA = valorB;
            IteracionesCambiaSolucion = IteracionesCambiaSolucion + 1;
        else %if valorB ≥ valorA
            if rand < exp( -(abs(valorB-valorA))/(T) )
                A = B{1};
                valorA = valorB;
                IteracionesCambiaSolucion = ...
                    IteracionesCambiaSolucion + 1;
            end
        end
        % - Guardamos mejor solucion encontrada hasta el momento -
        IteracionesComparaSolucionGlobal = ...
            IteracionesComparaSolucionGlobal + 1;
        if valorA < valorC
            C = A;
```

```
                valorC = valorA;
                IteracionesCambiaSolucionGlobal = ...
                    IteracionesCambiaSolucionGlobal + 1;
                IteracionCambioSolucionGlobal = ...
                    [IteracionCambioSolucionGlobal,UnContadorDeIteraciones];
                valorIteracionCambioSolucionGlobal = ...
                    [valorIteracionCambioSolucionGlobal,valorC];
            end
            % - - - - - - - - - - - - - - - - - - - - - - - - - - -
        end
        T = alpha * T;
    end



    LaMejorSolucion = FuncionEliminaRectangulosNoAportan(C,M,k);
    valorLaMejorSolucion = FuncionValorManta(M,LaMejorSolucion{1});
    tiempo = toc;

    disp("--- Recocido Simulado --- ")
    disp("Tiempo que tarda en encontrar la solucion: " + tiempo)
    disp("Valor de la mejor manta encontrada: " + valorLaMejorSolucion)

    end
```

BusquedaTabuMCyLP_P_v2_VG_NEW2

```
function [tiempo,LaMejorSolucion,valorLaMejorSolucion, ...
    IteracionesComparaSolucion,IteracionesCambiaSolucion, ...
    IteracionesComparaSolucionGlobal,IteracionesCambiaSolucionGlobal, ...
    IteracionCambioSolucionGlobal,valorIteracionCambioSolucionGlobal, ...
    iteraciones] = ...
    BusquedaTabuMCyLP_P_v2_VG_NEW2(M,SolucionInicial,seeed)
rng('default');
rng(seeed); %inicializamos semilla

    [A] = SolucionInicial; %Solucion Inicial
    k = length(A{2}); %numero rectangulos de A
    % -_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_

    tic;

iteraciones = 1800; %Numero de iteraciones que se realiza Busqueda ...
    Tabu %ParametroQuePodemosModificar
iterPenalizo = ceil(iteraciones/30); %cantidad a partir de la cual ...
    empieza penalizacion de un movimiento (se ve con la memoria a ...
    largo plazo) %ParametroQuePodemosModificar
if k ≥ 3
    q = 12; %Numero de iteraciones que se permanece en Lista Tabu ...
              %ParametroQuePodemosModificar
else %if k == 1 o k == 2
    q = 2; ...
```

```
                                                                    ...
        %ParametroQuePodemosModificar
end
r = 1; %servira para el ciclo while
T = zeros(2,k*8); %crea matriz de ceros de dimension ...
    2x#VecinosQuePuedenTenerse
[indiceT1,indiceT2] = size(T);
%Aqui se creo matriz T de dim 2x#VecinosQuePuedenTenerse. El vector de
%arriba indicara la cantidad de veces que ha permanecido la modificacion
%para ver en cuantas iteraciones sale. El vector de abajo indicara la
%cantidad de movimientos que se han hecho con esa modificacion.
contador = 1; %contador para ver cuantas veces se permanece en una ...
    solucion durante cierto periodo de iteraciones

[valorA] = FuncionValorManta(M,A{1});
IteracionesCambiaSolucion = 0; %cuenta cantidad de veces que cambia ...
    de solucion
IteracionesComparaSolucion = 0; %cuenta cantidadd de veces que ...
    compara soluciones
IteracionesCambiaSolucionGlobal = 0; %cuenta cantidad de veces que ...
    cambia a mejor solucion global
IteracionesComparaSolucionGlobal = 0; %cuenta cantidad de veces que ...
    compara la mejor solucion global
IteracionCambioSolucionGlobal = []; %guarda la iteracion en la que ...
    cambio mejor solucion global
valorIteracionCambioSolucionGlobal = []; %guarda valor mejor ...
    solucion global actualizada en iteracion donde mejoro


% % % r = 1 %primer iteracion
IteracionesCambiaSolucion = IteracionesCambiaSolucion + 1;
IteracionesComparaSolucion = IteracionesComparaSolucion + 1;
[VecindadA] = FuncionVecindadesGrandesBT(A,k); %vecindad de la ...
    matriz A
[B,valorB] = FuncionMejorVecino(M,A,k); %mejor vecino de A

[ElMovimientoFue1,IndiceDelMovimientoInverso1] = ...
    FuncionEncontrarMovimientoInversoDelQueSeHizo_Alterno(A,B,k);
[ElMovimientoFue,IndiceDelMovimientoInverso] = ...
    FuncionEncontrarMovimientoInversoDelQueSeHizo_Alterno(B,A,k);
     %Actualizacion 1 Lista Tabu
T(1,IndiceDelMovimientoInverso1) = ceil(q/6); %memoria corto plazo ...
    %ParametroQuePodemosModificar
T(2,IndiceDelMovimientoInverso1)=T(2,IndiceDelMovimientoInverso1)+1; ...
    %memoria largo plazo
T(1,IndiceDelMovimientoInverso) = q; %memoria corto plazo ...
    %ParametroQuePodemosModificar
T(2,IndiceDelMovimientoInverso)=T(2,IndiceDelMovimientoInverso)+1; ...
    %memoria largo plazo
A = B;
valorA = valorB;

MejorManta = A;
MejorValor = valorA; %MejorValor es el mejor valor hasta el momento
IteracionCambioSolucionGlobal = [IteracionCambioSolucionGlobal,r];
```

```
55 valorIteracionCambioSolucionGlobal = ...
       [valorIteracionCambioSolucionGlobal,MejorValor];
56 r = r+1;
57
58 while r≤iteraciones % #iteraciones - 1
59     [VecindadA] = FuncionVecindadesGrandesBT(A,k); %vecindad de la ...
           matriz A
60
61     [VecinosCumplenCriterioDeAspiracion, ...
           IndiceVecinosCriterioAspiracionEnVecindadA] = ...
           FuncionCriterioDeAspiracion(VecindadA,T,MejorValor,M); ...
           %Criterio de Aspiracion
62
63     [VecindadReducida] = FuncionVecindadReducidaBT(VecindadA,T, ...
           IndiceVecinosCriterioAspiracionEnVecindadA,M);
64
65     [MejorVecinoEnVecindadReducida, ...
           IndiceMejorVecinoEnVecindadReducida, ...
           valorMejorVecinoEnVecindadReducida] = ...
           FuncionMejorEnVecindadReducidaMemoriaLargoPlazo_paraNEW2( ...
           VecindadReducida,M,T,iterPenalizo); %ParametroQuePodemosModificar
66     % .:.:.:.:.:.:.:. (en la linea de codigo anterior se puede ...
           cambiar el valor de penalizacion)
67
68     % % % Actualizo Lista Tabu
69     [ElMovimientoFue1,IndiceDelMovimientoInverso1] = ...
           FuncionEncontrarMovimientoInversoDelQueSeHizo_Alterno(A, ...
           MejorVecinoEnVecindadReducida,k);
70     [ElMovimientoFue,IndiceDelMovimientoInverso] = ...
           FuncionEncontrarMovimientoInversoDelQueSeHizo_Alterno( ...
           MejorVecinoEnVecindadReducida,A,k);
71     T(1,IndiceDelMovimientoInverso1) = ceil(q/6)+1; %memoria corto ...
           plazo %ParametroQuePodemosModificar
72     T(2,IndiceDelMovimientoInverso1) = ...
           T(2,IndiceDelMovimientoInverso1) + 1; %memoria largo plazo
73     T(1,IndiceDelMovimientoInverso) = q+1; %memoria corto plazo ...
           %ParametroQuePodemosModificar
74     T(2,IndiceDelMovimientoInverso) = ...
           T(2,IndiceDelMovimientoInverso) + 1; %memoria largo plazo
75     for i = 1:indiceT2
76         if T(1,i) ≠ 0 %esto es, si VecindadA{i} estaba en Lista Tabu
77             T(1,i) = T(1,i) - 1; %Anteriormente se puso q+1 en la ...
                   nueva entrada. Aqui se corrige esa y se actualizan ...
                   las demas
78         end
79     end
80
81     % % % Actualizo respuesta
82     A = MejorVecinoEnVecindadReducida;
83     valorA = valorMejorVecinoEnVecindadReducida;
84     IteracionesCambiaSolucion = IteracionesCambiaSolucion + 1;
85     IteracionesComparaSolucion = IteracionesComparaSolucion + 1;
86
87     % % % mejor respuesta encontrada hasta el momento
88     IteracionesComparaSolucionGlobal = ...
```

```
            IteracionesComparaSolucionGlobal + 1;
89      if valorA < MejorValor
90          MejorManta = A;
91          MejorValor = valorA;
92          IteracionesCambiaSolucionGlobal = ...
                IteracionesCambiaSolucionGlobal + 1;
93          IteracionCambioSolucionGlobal = ...
                [IteracionCambioSolucionGlobal,r];
94          valorIteracionCambioSolucionGlobal = ...
                [valorIteracionCambioSolucionGlobal,MejorValor];
95      end
96
97      r = r+1;
98
99  end
100
101
102 LaMejorSolucion = FuncionEliminaRectangulosNoAportan(MejorManta,M,k);
103 valorLaMejorSolucion = FuncionValorManta(M,LaMejorSolucion{1});
104 tiempo = toc;
105
106 disp("--- Busqueda Tabu (Caso 1) --- ")
107 disp("Tiempo que tarda en encontrar la solucion: " + tiempo)
108 disp("Valor de la mejor manta encontrada: " + valorLaMejorSolucion)
109
110 end
```

FuncionVecindadesGrandesBT

```
1  function [Vecindad] = FuncionVecindadesGrandesBT(Manta,valork)
2  % Funcion que devuelve las soluciones que estan en la vecindad de Manta
3  % junto con los rectangulos que componen a cada solucion:
4  % Vecindad es un Cell Array  con cada entrada compuesta de de dos ...
       entrdas: 1ra, solucion; 2da, rectangulos
5
6  MantaConValorMuyFeo = ones(size(Manta{1}))*2; %cuando no es factible ...
       o se sale de dimension, agremamos esta matriz con un valor muy ...
       feo para que no pueda ser seleccionada
7  indiceVecindad = 1;
8  for i = 1:valork %del 1 hasta la cantidad de rectangulos
9      Movimientos = {"FuncionAgrandarRectanguloArriba", ...
           "FuncionAgrandarRectanguloAbajo", ...
           "FuncionAgrandarRectanguloIzquierda", ...
           "FuncionAgrandarRectanguloDerecha", ...
           "FuncionAchicarRectanguloArriba", ...
           "FuncionAchicarRectanguloAbajo", ...
           "FuncionAchicarRectanguloIzquierda", ...
           "FuncionAchicarRectanguloDerecha"};
10
11     for j = 1:8 %del 1 hasta el 8, donde 8 = #cantidad movimientos ...
           posibles de un rectangulo
12         RealizarMovimiento = str2func(Movimientos{j});
```

```
13          [RectanguloConMovimiento,SiNo] = ...
               RealizarMovimiento(Manta{2}{i});
14          if isequal(SiNo,1) %movimiento permitido
15              MatrizConstruida = Manta{1} - Manta{2}{i} + ...
                   RectanguloConMovimiento;
16              [factibleSiNo] = ...
                   FuncionVerificaFactibilidadManta(MatrizConstruida);
17              if isequal(factibleSiNo,1) %si Si es factible
18                  Vecindad{indiceVecindad}{1} = MatrizConstruida;
19                  Vecindad{indiceVecindad}{2} = Manta{2};
20                  Vecindad{indiceVecindad}{2}{i} = ...
                       RectanguloConMovimiento;
21                  indiceVecindad = indiceVecindad + 1;
22              else %si NO es factible %lo siguiente se hace para que ...
                   sea imposible elegir esas mantas
23                  Vecindad{indiceVecindad}{1} = MantaConValorMuyFeo;
24                  Vecindad{indiceVecindad}{2} = {};
25                  indiceVecindad = indiceVecindad + 1;
26              end
27          else %si movimiento no permitido %lo siguiente se hace para ...
               que sea imposible elegir esas mantas
28              Vecindad{indiceVecindad}{1} = MantaConValorMuyFeo;
29              Vecindad{indiceVecindad}{2} = {};
30              indiceVecindad = indiceVecindad + 1;
31          end
32      end
33  end
34
35  end
```

FuncionMejorVecino

```
1   function [B,valorB] = FuncionMejorVecino(M,A,k)
2
3   [VecindadA] = FuncionVecindadesGrandes(A,k);
4   [aa,bb] = size(VecindadA);
5   MatrizDeValores = [];
6   MatrizDeVecinos = [];
7
8   for i=1:bb
9       [valorVecinoAi] = FuncionValorManta(M,VecindadA{i}{1});
10      MatrizDeVecinos = [MatrizDeVecinos;i];
11      MatrizDeValores = [MatrizDeValores;valorVecinoAi];
12  end
13  MatrizConstruida = [MatrizDeVecinos,MatrizDeValores];
14
15  % % % En esta parte ordenamos respecto a la segunda columna % % %
16  [gamma,kappa] = sort(MatrizConstruida(:,2));
17  MatrizOrdenadaValorAscendente = [MatrizConstruida(kappa),gamma];
18  % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % %
19
20  B = VecindadA{MatrizOrdenadaValorAscendente(1,1)}; %B es el mejor vecino
```

```
21 [valorB] = FuncionValorManta(M,B{1});
22
23 end
```

FuncionEncontrarMovimientoInversoDelQueSeHizo_Alterno

```
1  function [ElMovimientoFue,IndiceDelMovimientoInverso] = ...
       FuncionEncontrarMovimientoInversoDelQueSeHizo_Alterno( ...
       MantaALaQueSeLlego,MantaDeLaQueSalio,k)
2
3  [VecindadMantaALaQueSeLlego] = ...
       FuncionVecindadesGrandesBT(MantaALaQueSeLlego,k); %vecindad de la ...
       matriz MantaALaQueSeLlego
4  i = 1; %servira para ubicar la solucion igual a MantaDeLaQueSalio en ...
       VecindadMantaALaQueSeLlego para hacer Tabu la modificacion
5  %que nos permita regresar a esa solucion
6  Encontrado = 0; %es falso que ya hayamos encontrado el movimiento ...
       inverso
7
8  while Encontrado == 0 %mientras no hayamos encontrado el movimiento ...
       inverso
9
10     if isequal(MantaDeLaQueSalio{1},VecindadMantaALaQueSeLlego{i}{1})
11     %encontramos que el movimiento inverso (para
12         %regresar a la solucion y guardarla en Tabu) es ...
               VecindadMantaALaQueSeLlego{i}
13         ElMovimientoFue = VecindadMantaALaQueSeLlego{i};
14         IndiceDelMovimientoInverso = i;
15         %Asi, prohibimos cierto numero de iteraciones elegir el ...
               vecino i, el cual guarda la informacion del
16         %procedimiento inverso que se le aplico, ya sea achicar o ...
               agrandar y el rectangulo al que se le efectuo
17         %la modificacion para evitar regresar a la solucion cierto ...
               numero de iteraciones
18         Encontrado = 1; %pues es cierto que ya encontramos el ...
               movimiento inverso
19     else
20         i = i+1;
21     end
22
23 end
24
25 end
```

FuncionCriterioDeAspiracion

```
1  function [VecinosCumplenCriterioDeAspiracion, ...
       IndiceVecinosCriterioAspiracionEnVecindadA] = ...
       FuncionCriterioDeAspiracion(VecindadA,T,MejorValor,M)
```

```
2
3  VecinosCumplenCriterioDeAspiracion = {};
4  IndiceVecinosCriterioAspiracionEnVecindadA = [];
5  [indiceT1,indiceT2] =size(T);
6  cont = 1;
7
8  for i = 1:indiceT2
9      if T(1,i)≠0 %(si T(1,i) distinto de 0) indica indice del ...
           movimiento que esta en Lista Tabu
10         [valorVi] = FuncionValorManta(M,VecindadA{i}{1});
11         if valorVi < MejorValor %mejora el mejor valor encontrado ...
               hasta el momento y cumple criterio de aspiracion
12             [VecinosCumplenCriterioDeAspiracion{cont}] = VecindadA{i};
13             cont = cont + 1;
14             IndiceVecinosCriterioAspiracionEnVecindadA = ...
                   [IndiceVecinosCriterioAspiracionEnVecindadA,i];
15         end
16     end
17 end
18
19 end
```

FuncionVecindadReducidaBT

```
1  function [VecindadReducida] = FuncionVecindadReducidaBT(VecindadA,T, ...
       IndiceVecinosCriterioAspiracionEnVecindadA,M)
2
3  VecindadReducida = VecindadA;
4
5  MatrizUnos = ones(size(M))*2; % Solucion con valor feo para no ser ...
       elegido
6  [indiceT1,indiceT2] = size(T);
7  [IVCR1,IVCA2] = size(IndiceVecinosCriterioAspiracionEnVecindadA);
8  [indiceVecindadA1,indiceVecindadA2] = size(VecindadA);
9
10 % % % VECINDAD CASI REDUCIDA = Vecindad - ListaTabu
11 for i = 1:indiceT2
12     if T(1,i) ≠ 0 %(si  T(1,i) distinto de 0) VecindadA{i} esta en ...
           Lista Tabu
13         [VecindadReducida{i}{1}] = MatrizUnos;
14     end
15 end
16
17 % % % VECINDAD REDUCIDA = (Vecindad - ListaTabu) + CriterioAspiracion
18 if IVCA2 ≠ 0 %si ocurre esto, el criterio de aspiracion es no vacio
19     for i = 1:IVCA2
20         indiceVecinoCritAsp = ...
               IndiceVecinosCriterioAspiracionEnVecindadA(i);
21         [VecindadReducida{indiceVecinoCritAsp}] = ...
               VecindadA{indiceVecinoCritAsp};
22     end
23 end
```

```
24
25 end
```

FuncionMejorEnVecindadReducidaMemoriaLargoPlazo_paraNEW2

```
1  function [MejorVecinoEnVecindadReducida, ...
       IndiceMejorVecinoEnVecindadReducida, ...
       valorMejorVecinoEnVecindadReducida] = ...
       FuncionMejorEnVecindadReducidaMemoriaLargoPlazo_paraNEW2( ...
       VecindadReducida,M,T,iterPenalizo)
2
3  %penalizacion1 y penalizacion2 deben ser iguales ...
       %ParametroQuePodemosModificar
4  probabCambio = 0.38; %ParametroQuePodemosModificar
5  MejorVecinoEnVecindadReducida = VecindadReducida{1};
6  IndiceMejorVecinoEnVecindadReducida = 1;
7
8  if T(2,1) < iterPenalizo %si ese movimiento no se ha hecho mucho
9      valorMejorVecinoEnVecindadReducida = ...
           FuncionValorManta(VecindadReducida{1}{1},M);
10 else %if T(2,1) ≥ iterPenalizo
11     penalizacion1 = ( ceil(T(2,1)/4) ); %ParametroQuePodemosModificar
12     valorMejorVecinoEnVecindadReducida = ...
           FuncionValorManta(VecindadReducida{1}{1},M) + penalizacion1 ; ...
           % .:.:.:.:.:.:.:.
13 end
14
15 [IndiceVecindadReducida1,IndiceVecindadReducida2] = ...
       size(VecindadReducida);
16
17 for i = 2:IndiceVecindadReducida2
18
19     if T(2,i) < iterPenalizo %si ese movimiento no se ha hecho mucho
20         valorVecRed = FuncionValorManta(VecindadReducida{i}{1},M);
21     else %if T(2,i) ≥ iterPenalizo
22         penalizacion2 = ( ceil(T(2,i)/4) ); ...
               %ParametroQuePodemosModificar
23         valorVecRed = FuncionValorManta(VecindadReducida{i}{1},M) + ...
               penalizacion2 ; % .:.:.:.:.:.:.:.
24     end
25
26     if valorVecRed < valorMejorVecinoEnVecindadReducida
27         MejorVecinoEnVecindadReducida = VecindadReducida{i};
28         IndiceMejorVecinoEnVecindadReducida = i;
29         valorMejorVecinoEnVecindadReducida = ...
               FuncionValorManta(VecindadReducida{i}{1},M);
30
31     elseif isequal(valorVecRed,valorMejorVecinoEnVecindadReducida)
32             if rand < probabCambio
33                 MejorVecinoEnVecindadReducida = VecindadReducida{i};
34                 IndiceMejorVecinoEnVecindadReducida = i;
35                 valorMejorVecinoEnVecindadReducida = ...
```

```
                    FuncionValorManta(VecindadReducida{i}{1},M);
            end
        end
end

end
```

BusquedaTabuMCyLP_GuardSolAleat_v2_VG_forNUEVO

```
function [tiempo,LaMejorSolucion,valorLaMejorSolucion, ...
    IteracionesComparaSolucion,IteracionesCambiaSolucion, ...
    IteracionesComparaSolucionGlobal,IteracionesCambiaSolucionGlobal, ...
    IteracionCambioSolucionGlobal,valorIteracionCambioSolucionGlobal, ...
    UnContadorDeIteraciones] = ...
    BusquedaTabuMCyLP_GuardSolAleat_v2_VG_forNUEVO(M, ...
    SolucionInicial,seeed)
rng('default');
rng(seeed); %inicializamos semilla

    [A] = SolucionInicial; %Solucion Inicial
    k = length(A{2}); %numero rectangulos de A
    % -_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_

    tic;

iteraciones = 1000; %Numero de iteraciones que se realiza Busqueda ...
    Tabu  %ParametroQuePodemosModificar
iteracionesRep = 500; %Numero de iteraciones que se realiza Busqueda ...
    Tabu en las soluciones guardadas %ParametroQuePodemosModificar
TMOiter = 200; %Numero de iteraciones que elige vecino #numVecino en ...
    las soluciones guardadas %ParametroQuePodemosModificar
if k ≥ 3
    q = 12; %Numero de iteraciones que se permanece en Lista Tabu ...
        %ParametroQuePodemosModificar
else %if k == 1 o k == 2
    q = 2; %ParametroQuePodemosModificar
end
numVecino = ceil((k*8)/(5)); %vecino que se elige durante #TMOiter ...
    seguidas para diversificar busqueda %ParametroQuePodemosModificar
manta_iteracion = [150, 250, 400, 600, 800]; %iteraciones en las ...
    cuales guardaremos las mantas encontradas ...
    %ParametroQuePodemosModificar

r = 1; %servira para el ciclo while
T = zeros(2,k*8); %crea matriz de ceros de dimension ...
    2x#VecinosQuePuedenTenerse
[indiceT1,indiceT2] = size(T);
%Aqui se creo matriz T de dim 2x#VecinosQuePuedenTenerse. El vector de
%arriba indicara la cantidad de veces que ha permanecido la modificacion
%para ver en cuantas iteraciones sale. El vector de abajo indicara la
%cantidad de movimientos que se han hecho con esa modificacion.
contador = 1; %contador para ver cuantas veces se permanece en una ...
```

```
    solucion durante cierto periodo de iteraciones
30
31 [valorA] = FuncionValorManta(M,A{1});
32 SolucionesAGuardar ...
    =[manta_iteracion(1),manta_iteracion(2),manta_iteracion(3),
33 manta_iteracion(4),manta_iteracion(5)];
34 IteracionesCambiaSolucion = 0; %cuenta cantidad de veces que cambia ...
    de solucion
35 IteracionesComparaSolucion = 0; %cuenta cantidadd de veces que ...
    compara soluciones
36 IteracionesCambiaSolucionGlobal = 0; %cuenta cantidad de veces que ...
    cambia a mejor solucion global
37 IteracionesComparaSolucionGlobal = 0; %cuenta cantidad de veces que ...
    compara la mejor solucion global
38 IteracionCambioSolucionGlobal = []; %guarda la iteracion en la que ...
    cambio mejor solucion global
39 valorIteracionCambioSolucionGlobal = []; %guarda valor mejor ...
    solucion global actualizada en iteracion donde mejoro
40
41 % % % r = 1 %primer iteracion
42 IteracionesCambiaSolucion = IteracionesCambiaSolucion + 1;
43 IteracionesComparaSolucion = IteracionesComparaSolucion + 1;
44 [VecindadA] = FuncionVecindadesGrandes(A,k); %vecindad de la matriz A
45 [B,valorB] = FuncionMejorVecino(M,A,k); %mejor vecino de A
46
47 [ElMovimientoFue1,IndiceDelMovimientoInverso1] = ...
    FuncionEncontrarMovimientoInversoDelQueSeHizo_Alterno(A,B,k);
48 [ElMovimientoFue,IndiceDelMovimientoInverso] = ...
    FuncionEncontrarMovimientoInversoDelQueSeHizo_Alterno(B,A,k);
49     %Actualizacion 1 Lista Tabu
50 T(1,IndiceDelMovimientoInverso1) = ceil(q/6); %memoria corto plazo ...
    %ParametroQuePodemosModificar
51 T(2,IndiceDelMovimientoInverso1)=T(2,IndiceDelMovimientoInverso1)+1; ...
    %memoria largo plazo
52 T(1,IndiceDelMovimientoInverso) = q; %memoria corto plazo ...
    %ParametroQuePodemosModificar
53 T(2,IndiceDelMovimientoInverso)=T(2,IndiceDelMovimientoInverso)+1; ...
    %memoria largo plazo
54 A = B;
55 valorA = valorB;
56
57 MejorManta = A;
58 MejorValor = valorA; %MejorValor es el mejor valor hasta el momento
59 IteracionCambioSolucionGlobal = [IteracionCambioSolucionGlobal,1];
60 valorIteracionCambioSolucionGlobal = ...
    [valorIteracionCambioSolucionGlobal,MejorValor];
61 r = r+1;
62
63 while r≤iteraciones % #iteraciones - 1
64     [VecindadA] = FuncionVecindadesGrandesBT(A,k); %vecindad de la ...
        matriz A
65
66     [VecinosCumplenCriterioDeAspiracion, ...
        IndiceVecinosCriterioAspiracionEnVecindadA] = ...
        FuncionCriterioDeAspiracion(VecindadA,T,MejorValor,M); ...
```

```
        %Criterio de Aspiracion
67
68      [VecindadReducida] = FuncionVecindadReducidaBT(VecindadA,T, ...
            IndiceVecinosCriterioAspiracionEnVecindadA,M);
69
70      [MejorVecinoEnVecindadReducida, ...
            IndiceMejorVecinoEnVecindadReducida, ...
            valorMejorVecinoEnVecindadReducida] = ...
            FuncionMejorEnVecindadReducida_forNUEVO(VecindadReducida,M); ...
            %ParametroQuePodemosModificar
71
72      % % % Actualizo Lista Tabu
73      [ElMovimientoFue1,IndiceDelMovimientoInverso1] = ...
            FuncionEncontrarMovimientoInversoDelQueSeHizo_Alterno(A, ...
            MejorVecinoEnVecindadReducida,k);
74      [ElMovimientoFue,IndiceDelMovimientoInverso] = ...
            FuncionEncontrarMovimientoInversoDelQueSeHizo_Alterno( ...
            MejorVecinoEnVecindadReducida,A,k);
75      T(1,IndiceDelMovimientoInverso1) = ceil(q/6)+1; %memoria corto ...
            plazo %ParametroQuePodemosModificar
76      T(2,IndiceDelMovimientoInverso1) = ...
            T(2,IndiceDelMovimientoInverso1) + 1; %memoria largo plazo
77      T(1,IndiceDelMovimientoInverso) = q+1; %memoria corto plazo ...
            %ParametroQuePodemosModificar
78      T(2,IndiceDelMovimientoInverso) = ...
            T(2,IndiceDelMovimientoInverso) + 1; %memoria largo plazo
79      for i = 1:indiceT2
80          if T(1,i) ≠ 0 %esto es, si VecindadA{i} estaba en Lista Tabu
81              T(1,i) = T(1,i) - 1; %Anteriormente se puso 4 en la ...
                    nueva entrada. Aqui se corrige esa y se actualizan ...
                    las demas
82          end
83      end
84
85      % % % Actualizo respuesta
86      A = MejorVecinoEnVecindadReducida;
87      valorA = valorMejorVecinoEnVecindadReducida;
88      IteracionesCambiaSolucion = IteracionesCambiaSolucion + 1;
89      IteracionesComparaSolucion = IteracionesComparaSolucion + 1;
90
91      % % % mejor respuesta encontrada hasta el momento
92      IteracionesComparaSolucionGlobal = ...
            IteracionesComparaSolucionGlobal + 1;
93      if valorA < MejorValor
94          MejorManta = A;
95          MejorValor = valorA;
96          IteracionesCambiaSolucionGlobal = ...
                IteracionesCambiaSolucionGlobal + 1;
97          IteracionCambioSolucionGlobal = ...
                [IteracionCambioSolucionGlobal,r];
98          valorIteracionCambioSolucionGlobal = ...
                [valorIteracionCambioSolucionGlobal,MejorValor];
99      end
100
101
```

```
102     if any(r == SolucionesAGuardar)
103         if isequal(r,manta_iteracion(1))
104             Manta1 = A;
105         elseif isequal(r,manta_iteracion(2))
106                 Manta2 = A;
107         elseif isequal(r,manta_iteracion(3))
108                     Manta3 = A;
109         elseif isequal(r,manta_iteracion(4))
110                     Manta4 = A;
111         else %if isequal(r,manta_iteracion(5))
112             Manta5 = A;
113         end
114     end
115
116     r = r+1;
117
118 end
119
120
121
122 %durante TMOiter iteraciones elijo numVecino mejor opcion de la vecindad
123 for i = 1:TMOiter
124 [G1,valorG1,Vecc1,Veccindad1] = ...
       FuncionRoundMejorVecino(M,Manta1,k,numVecino);
125 Manta1 = G1;
126 valorManta1 = valorG1;
127 IteracionesComparaSolucion = IteracionesComparaSolucion + 1;
128
129 [G2,valorG2,Vecc2,Veccindad2] = ...
       FuncionRoundMejorVecino(M,Manta2,k,numVecino);
130 Manta2 = G2;
131 valorManta2 = valorG2;
132 IteracionesComparaSolucion = IteracionesComparaSolucion + 1;
133
134 [G3,valorG3,Vecc3,Veccindad3] = ...
       FuncionRoundMejorVecino(M,Manta3,k,numVecino);
135 Manta3 = G3;
136 valorManta3 = valorG3;
137 IteracionesComparaSolucion = IteracionesComparaSolucion + 1;
138
139 [G4,valorG4,Vecc4,Veccindad4] = ...
       FuncionRoundMejorVecino(M,Manta4,k,numVecino);
140 Manta4 = G4;
141 valorManta4 = valorG4;
142 IteracionesComparaSolucion = IteracionesComparaSolucion + 1;
143
144 [G5,valorG5,Vecc5,Veccindad5] = ...
       FuncionRoundMejorVecino(M,Manta5,k,numVecino);
145 Manta5 = G5;
146 valorManta5 = valorG5;
147 IteracionesComparaSolucion = IteracionesComparaSolucion + 1;
148 end
149
150 [MejorManta1,MejorValor1,IteracionesComparaSolucion] = ...
       FuncionBuscarTabuMCyLP_GuardSolAleat_forNUEVO(M,Manta1,k,q, ...
```

```
    iteracionesRep,IteracionesComparaSolucion);
[MejorManta2,MejorValor2,IteracionesComparaSolucion] = ...
    FuncionBuscarTabuMCyLP_GuardSolAleat_forNUEVO(M,Manta2,k,q, ...
    iteracionesRep,IteracionesComparaSolucion);
[MejorManta3,MejorValor3,IteracionesComparaSolucion] = ...
    FuncionBuscarTabuMCyLP_GuardSolAleat_forNUEVO(M,Manta3,k,q, ...
    iteracionesRep,IteracionesComparaSolucion);
[MejorManta4,MejorValor4,IteracionesComparaSolucion] = ...
    FuncionBuscarTabuMCyLP_GuardSolAleat_forNUEVO(M,Manta4,k,q, ...
    iteracionesRep,IteracionesComparaSolucion);
[MejorManta5,MejorValor5,IteracionesComparaSolucion] = ...
    FuncionBuscarTabuMCyLP_GuardSolAleat_forNUEVO(M,Manta5,k,q, ...
    iteracionesRep,IteracionesComparaSolucion);

ListaMejoresMantas = ...
    {MejorManta1,MejorManta2,MejorManta3,MejorManta4,MejorManta5};
UnContadorDeIteraciones = iteraciones;
for i = 1:5
    UnContadorDeIteraciones = UnContadorDeIteraciones + 1;
    IteracionesComparaSolucion = IteracionesComparaSolucion + 1;
    valori = FuncionValorManta(ListaMejoresMantas{i}{1},M);
    IteracionesComparaSolucionGlobal = ...
        IteracionesComparaSolucionGlobal + 1;
    if valori < MejorValor
        MejorManta = ListaMejoresMantas{i};
        MejorValor = valori;
        IteracionesCambiaSolucion = IteracionesCambiaSolucion + 1;
        IteracionesCambiaSolucionGlobal = ...
            IteracionesCambiaSolucionGlobal + 1;
        IteracionCambioSolucionGlobal = ...
            [IteracionCambioSolucionGlobal,UnContadorDeIteraciones];
        valorIteracionCambioSolucionGlobal = ...
            [valorIteracionCambioSolucionGlobal,MejorValor];
    end
end


LaMejorSolucion = FuncionEliminaRectangulosNoAportan(MejorManta,M,k);
valorLaMejorSolucion = FuncionValorManta(M,LaMejorSolucion{1});
tiempo = toc;

disp("--- Busqueda Tabu (Caso 2) --- ")
disp("Tiempo que tarda en encontrar la solucion: " + tiempo)
disp("Valor de la mejor manta encontrada: " + valorLaMejorSolucion)

end
```

FuncionMejorEnVecindadReducida_forNUEVO

```
function [MejorVecinoEnVecindadReducida, ...
    IndiceMejorVecinoEnVecindadReducida, ...
    valorMejorVecinoEnVecindadReducida] = ...
```

```
    FuncionMejorEnVecindadReducida_forNUEVO(VecindadReducida,M)

MejorVecinoEnVecindadReducida = VecindadReducida{1};
IndiceMejorVecinoEnVecindadReducida = 1;
[valorMejorVecinoEnVecindadReducida] = ...
    FuncionValorManta(M,VecindadReducida{1}{1});

[IndiceVecindadReducida1,IndiceVecindadReducida2] = ...
    size(VecindadReducida);

for i = 2:IndiceVecindadReducida2
     valorVecRed = FuncionValorManta(VecindadReducida{i}{1},M);

     if valorVecRed < valorMejorVecinoEnVecindadReducida
          MejorVecinoEnVecindadReducida = VecindadReducida{i};
          IndiceMejorVecinoEnVecindadReducida = i;
          valorMejorVecinoEnVecindadReducida = valorVecRed;

     elseif isequal(valorVecRed,valorMejorVecinoEnVecindadReducida)
          if rand < 0.38 %ParametroQuePodemosModificar
               MejorVecinoEnVecindadReducida = VecindadReducida{i};
               IndiceMejorVecinoEnVecindadReducida = i;
               valorMejorVecinoEnVecindadReducida = valorVecRed;
          end
     end

end

end
```

FuncionRoundMejorVecino

```
function [B,valorB,Vecc,Veccindad] = ...
    FuncionRoundMejorVecino(M,A,k,numVecino)

[VecindadA] = FuncionVecindadesGrandes(A,k);
[aa,bb] = size(VecindadA);
MatrizDeValores = [];
MatrizDeVecinos = [];

for i=1:bb
     valorVecinoAi = FuncionValorManta(M,VecindadA{i}{1});
     MatrizDeVecinos = [MatrizDeVecinos;i];
     MatrizDeValores = [MatrizDeValores;valorVecinoAi];
end
MatrizConstruida = [MatrizDeVecinos,MatrizDeValores];

% % % En esta parte ordenamos respecto a la segunda columna % % %
[gamma,kappa] = sort(MatrizConstruida(:,2));
MatrizOrdenadaValorAscendente = [MatrizConstruida(kappa),gamma];
% % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % %

```

```
20 B = VecindadA{MatrizOrdenadaValorAscendente(numVecino,1)}; ...
       %escogemos el vecino #numVecino
21 valorB = MatrizOrdenadaValorAscendente(numVecino,2);
22 Vecc = MatrizOrdenadaValorAscendente;
23 Veccindad = VecindadA;
24
25 end
```

BusquedaTabuMCyLP_GuardSolAleat_VG_forNUEVO

```
1  function [Todas_Cobijas_QuitarRect,Todos_valores_QuitarRect, ...
       promedio_tiempo] = ...
       BusquedaTabuMCyLP_GuardSolAleat_VG_forNUEVO(M,k,Repetite)
2  % % Repetite = 2; % ENTRADA
3
4  Todos_valores = [];
5  Todas_Cobijas = {};
6  Tiempo_empleado = [];
7
8  for Rep_BL = 1:Repetite
9
10     % _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
11     % | | | | | | | | | ELECCIoN IMAGEN OBJETIVO | | | | | | | | |
12     [A] = FuncionSolucionInicial(M,k); %Solucion Inicial
13     % _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
14
15     tic;
16
17
18
19 iteraciones = 700; %Numero de iteraciones que se realiza Busqueda ...
       Tabu %ParametroQuePodemosModificar
20 iteracionesRep = 500; %Numero de iteraciones que se realiza Busqueda ...
       Tabu en las soluciones guardadas %ParametroQuePodemosModificar
21 TMOiter = 150; %Numero de iteraciones que elige vecino #numVecino en ...
       las soluciones guardadas %ParametroQuePodemosModificar
22 q = 10; %Numero de iteraciones que se permanece en Lista Tabu ...
       %ParametroQuePodemosModificar
23 numVecino = ceil((k*8)/(3)); %vecino que se elige durante #TMOiter ...
       seguidas para diversificar busqueda %ParametroQuePodemosModificar
24 manta_iteracion = [150, 250, 400, 600]; %iteraciones en las cuales ...
       guardaremos las mantas encontradas %ParametroQuePodemosModificar
25
26 r = 1; %servira para el ciclo while
27 T = zeros(2,k*8); %crea matriz de ceros de dimension ...
       2x#VecinosQuePuedenTenerse
28 [indiceT1,indiceT2] = size(T);
29 %Aqui se creo matriz T de dim 2x#VecinosQuePuedenTenerse. El vector de
30 %arriba indicara la cantidad de veces que ha permanecido la modificacion
31 %para ver en cuantas iteraciones sale. El vector de abajo indicara la
32 %cantidad de movimientos que se han hecho con esa modificacion.
33 contador = 1; %contador para ver cuantas veces se permanece en una ...
```

```
    solucion durante cierto periodo de iteraciones
34
35 [valorA] = FuncionValorManta(M,A{1});
36 SolucionesAGuardar = ...
    [manta_iteracion(1),manta_iteracion(2),manta_iteracion(3), ...
    manta_iteracion(4)];
37
38 % % % r = 1 %primer iteracion
39 [VecindadA] = FuncionVecindadesGrandes(A,k); %vecindad de la matriz A
40 [B,valorB] = FuncionMejorVecino(M,A,k); %mejor vecino de A
41
42 [ElMovimientoFue,IndiceDelMovimientoInverso] = ...
    FuncionEncontrarMovimientoInversoDelQueSeHizo(B,A,k);
43     %Actualizacion 1 Lista Tabu
44 T(1,IndiceDelMovimientoInverso) = q; %memoria corto plazo
45 T(2,IndiceDelMovimientoInverso)=T(2,IndiceDelMovimientoInverso)+1; ...
    %memoria largo plazo
46 A = B;
47 valorA = valorB;
48
49 MejorManta = A;
50 MejorValor = valorA; %MejorValor es el mejor valor hasta el momento
51 r = r+1;
52
53 while r≤iteraciones % #iteraciones - 1
54     [VecindadA] = FuncionVecindadesGrandesBT(A,k); %vecindad de la ...
        matriz A
55
56     [VecinosCumplenCriterioDeAspiracion, ...
        IndiceVecinosCriterioAspiracionEnVecindadA] = ...
        FuncionCriterioDeAspiracion(VecindadA,T,MejorValor,M); ...
        %Criterio de Aspiracion
57
58     [VecindadReducida] = FuncionVecindadReducidaBT(VecindadA,T, ...
        IndiceVecinosCriterioAspiracionEnVecindadA,M);
59
60     [MejorVecinoEnVecindadReducida, ...
        IndiceMejorVecinoEnVecindadReducida, ...
        valorMejorVecinoEnVecindadReducida] = ...
        FuncionMejorEnVecindadReducida_forNUEVO(VecindadReducida,M); ...
        %ParametroQuePodemosModificar
61
62     % % % Actualizo Lista Tabu
63     [ElMovimientoFue,IndiceDelMovimientoInverso] = ...
        FuncionEncontrarMovimientoInversoDelQueSeHizo( ...
        MejorVecinoEnVecindadReducida,A,k);
64     T(1,IndiceDelMovimientoInverso) = q+1; %memoria corto plazo
65     T(2,IndiceDelMovimientoInverso) = ...
        T(2,IndiceDelMovimientoInverso) + 1; %memoria largo plazo
66     for i = 1:indiceT2
67         if T(1,i) ≠ 0 %esto es, si VecindadA{i} estaba en Lista Tabu
68             T(1,i) = T(1,i) - 1; %Anteriormente se puso 4 en la ...
                nueva entrada. Aqui se corrige esa y se actualizan ...
                las demas
69         end
```

```
70      end
71
72      % % % Actualizo respuesta
73      A = MejorVecinoEnVecindadReducida;
74      valorA = valorMejorVecinoEnVecindadReducida;
75
76      % % % mejor respuesta encontrada hasta el momento
77      if valorA < MejorValor
78          MejorManta = A;
79          MejorValor = valorA;
80      end
81
82
83      if any(r == SolucionesAGuardar)
84          if isequal(r,manta_iteracion(1))
85              Manta1 = A;
86          elseif isequal(r,manta_iteracion(2))
87                  Manta2 = A;
88          elseif isequal(r,manta_iteracion(3))
89                      Manta3 = A;
90          else% isequal(r,manta_iteracion(4))
91                      Manta4 = A;
92          end
93      end
94
95      r = r+1;
96
97  end
98
99
100
101 %durante TMOiter iteraciones elijo numVecino mejor opcion de la vecindad
102 for i = 1:TMOiter
103 [G1,valorG1,Vecc1,Veccindad1] = ...
        FuncionRoundMejorVecino(M,Manta1,k,numVecino);
104 Manta1 = G1;
105 valorManta1 = valorG1;
106
107 [G2,valorG2,Vecc2,Veccindad2] = ...
        FuncionRoundMejorVecino(M,Manta2,k,numVecino);
108 Manta2 = G2;
109 valorManta2 = valorG2;
110
111 [G3,valorG3,Vecc3,Veccindad3] = ...
        FuncionRoundMejorVecino(M,Manta3,k,numVecino);
112 Manta3 = G3;
113 valorManta3 = valorG3;
114
115 [G4,valorG4,Vecc4,Veccindad4] = ...
        FuncionRoundMejorVecino(M,Manta4,k,numVecino);
116 Manta4 = G4;
117 valorManta4 = valorG4;
118 end
119
120 [MejorManta1,MejorValor1] = ...
```

```
    FuncionBuscarTabuMCyLP_GuardSolAleat_forNUEVO(M,Manta1,k,q, ...
    iteracionesRep);
[MejorManta2,MejorValor2] = ...
    FuncionBuscarTabuMCyLP_GuardSolAleat_forNUEVO(M,Manta2,k,q, ...
    iteracionesRep);
[MejorManta3,MejorValor3] = ...
    FuncionBuscarTabuMCyLP_GuardSolAleat_forNUEVO(M,Manta3,k,q, ...
    iteracionesRep);
[MejorManta4,MejorValor4] = ...
    FuncionBuscarTabuMCyLP_GuardSolAleat_forNUEVO(M,Manta4,k,q, ...
    iteracionesRep);

ListaMejoresMantas = {MejorManta1,MejorManta2,MejorManta3,MejorManta4};
for i = 1:4
    valori = FuncionValorManta(ListaMejoresMantas{i}{1},M);
    if valori < MejorValor
        MejorManta = ListaMejoresMantas{i};
        MejorValor = valori;
    end
end



Tiempo_empleado(1,Rep_BL) = toc;
Todos_valores(1,Rep_BL) = MejorValor;
Todas_Cobijas{Rep_BL} = MejorManta;

end % Repetite

%   ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬
%   ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬
%   ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬
Todas_Cobijas_QuitarRect = {};
Todos_valores_QuitarRect = [];
for i = 1:Repetite
    Todas_Cobijas_QuitarRect{i} = ...
        FuncionEliminaRectangulosNoAportan(Todas_Cobijas{i},M,k);
    Todos_valores_QuitarRect(1,i) = ...
        FuncionValorManta(M,Todas_Cobijas_QuitarRect{i}{1});
end
%   ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬
%   ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬
%   ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬ ¬

disp("Las mantas encontradas estan en: ")
Todas_Cobijas_QuitarRect
disp("Los valores de las mantas encontradas son: ")
Todos_valores_QuitarRect
disp("--- Busqueda Tabu (Caso 2) --- ")
disp("El promedio de los valores es: " + mean(Todos_valores_QuitarRect))
disp("Valor de la peor manta encontrada: " + ...
    max(Todos_valores_QuitarRect))
disp("Valor de la mejor manta encontrada: " + ...
    min(Todos_valores_QuitarRect))
disp("Tiempo promedio que tarda en encontrar una solucion: " + ...
```

```
    mean(Tiempo_empleado))
164
165 promedio_tiempo = mean(Tiempo_empleado);
166 end
```

FuncionEncontrarMovimientoInversoDelQueSeHizo

```
1  function [ElMovimientoFue,IndiceDelMovimientoInverso] = ...
       FuncionEncontrarMovimientoInversoDelQueSeHizo(MantaALaQueSeLlego, ...
       MantaDeLaQueSalio,k)
2
3  [VecindadMantaALaQueSeLlego] = ...
       FuncionVecindadesGrandes(MantaALaQueSeLlego,k); %vecindad de la ...
       matriz MantaALaQueSeLlego
4  i = 1; %servira para ubicar la solucion igual a MantaDeLaQueSalio en ...
       VecindadMantaALaQueSeLlego para hacer Tabu la modificacion
5  %que nos permita regresar a esa solucion
6  Encontrado = 0; %es falso que ya hayamos encontrado el movimiento ...
       inverso
7
8  while Encontrado == 0 %mientras no hayamos encontrado el movimiento ...
       inverso
9
10     if isequal(MantaDeLaQueSalio{1},VecindadMantaALaQueSeLlego{i}{1})
11     %encontramos que el movimiento inverso (para
12         %regresar a la solucion y guardarla en Tabu) es ...
               VecindadMantaALaQueSeLlego{i}
13         ElMovimientoFue = VecindadMantaALaQueSeLlego{i};
14         IndiceDelMovimientoInverso = i;
15         %Asi, prohibimos cierto numero de iteraciones elegir el ...
               vecino i, el cual guarda la informacion del
16         %procedimiento inverso que se le aplico, ya sea achicar o ...
               agrandar y el rectangulo al que se le efectuo
17         %la modificacion para evitar regresar a la solucion cierto ...
               numero de iteraciones
18         Encontrado = 1; %pues es cierto que ya encontramos el ...
               movimiento inverso
19     else
20         i = i+1;
21     end
22
23 end
24
25 end
```

AlgoritmosGeneticos_v3_VG

```
1  function [tiempo_acumulado_por_generacion,LaMejorSolucion, ...
       valorLaMejorSolucion,valor_Mejor_Cobija_Hasta_El_Momento, ...
```

```
    promedio_mejores_70_valores_por_generacion_SinQuitarRectangulos, ...
    vector_Generaciones,IteracionesComparaSolucion, ...
    IteracionesCambiaSolucion,IteracionesComparaSolucionGlobal, ...
    IteracionesCambiaSolucionGlobal,IteracionCambioSolucionGlobal, ...
    valorIteracionCambioSolucionGlobal,i, ...
    cobijasUltimaGeneracionOrdenada,valoresUltimaGeneracionOrdenada] ...
    = AlgoritmosGeneticos_v3_VG(M,k,Generaciones,seeed)
rng('default');
rng(seeed); %inicializamos semilla
%     creamos     lo que queremos obtener
tiempo_por_generacion = []; %  y
Cobijas_por_generacion_SinQuitarRectangulos = {}; %  y
valores_por_generacion_SinQuitarRectangulos = {}; %  v
Mejor_Cobija_Hasta_El_Momento = {}; %  v
valor_Mejor_Cobija_Hasta_El_Momento = []; %  v
cada_gener = 100; %cada cuantas generaciones guardar datos
i_aux = 0; %indice auxiliar
% - - - - - - - -
%tiempo_acumulado_por_generacion %  v
%promedio valores por generacion quitando rectangulos %  v
%promedio valores por generacion dejando rectangulos %  v
%promedio mejores 70 valores por generacion quitando rectangulos %  v
%promedio mejores 70 valores por generacion dejando rectangulos %  v
%     fin     de lo que queremos obtener

MejoresSolucionesGlobales = {};
ValoresMejoresSolucionesGlobales =[];

ProbabilidadMutacion = 0.15; %es la probabilidad que tiene cada ...
    individuo de cada descendencia de mutar %ParametroQuePodemosModificar
CantidadPoblacion = 140; %ELEGIR UN NuMERO PAR (pues para numero ...
    impar no se definio como. En ese caso, puede copiarse uno ...
    automaticamente(el que no tuvo pareja o el mejor de todos)) ...
    %ParametroQuePodemosModificar

IteracionesCambiaSolucion = 0; %cuenta cantidad de veces que cambia ...
    de solucion
IteracionesComparaSolucion = 0; %cuenta cantidad de veces que ...
    compara soluciones
IteracionesCambiaSolucionGlobal = 0; %cuenta cantidad de veces que ...
    cambia a mejor solucion global
IteracionesComparaSolucionGlobal = 0; %cuenta cantidad de veces que ...
    compara la mejor solucion global
IteracionCambioSolucionGlobal = []; %guarda la iteracion en la que ...
    cambio mejor solucion global
valorIteracionCambioSolucionGlobal = []; %guarda valor mejor ...
    solucion global actualizada en iteracion donde mejoro


% % % Calculamos una poblacion inicial
for i = 1:CantidadPoblacion
    [PoblacionInicial{i}] = FuncionSolucionAleatoria(M,k);
end
PoblacionPrimerGeneracion = PoblacionInicial;

```

```
% >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
LaMejorSolucion = PoblacionInicial{1};
valorLaMejorSolucion = FuncionValorManta(LaMejorSolucion{1},M);
for iw = 1:CantidadPoblacion
    valorSol_iw = FuncionValorManta(M,PoblacionInicial{iw}{1});
    IteracionesComparaSolucion = IteracionesComparaSolucion + 1;
    IteracionesComparaSolucionGlobal = ...
       IteracionesComparaSolucionGlobal + 1;
    if ( valorSol_iw < valorLaMejorSolucion )
        LaMejorSolucion = PoblacionInicial{iw};
        valorLaMejorSolucion = valorSol_iw; %NOTA: QUITAR EL ";" ...
           PARA VER EVOLUCIoN DE LA MEJOR SOLUCIoN
        IteracionesCambiaSolucion = IteracionesCambiaSolucion + 1;
        IteracionesCambiaSolucionGlobal = ...
           IteracionesCambiaSolucionGlobal + 1;
    end
end
% >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

% ...

% INICIO ITERACIoN POR GENERACIONES
% ...


for i = 1:(Generaciones-10) % Inicio iter generaciones
    tic; %inicio conteo  %

% % % Calculamos Probabilidades por el metodo de la ruleta
[PoblacionIndiceValorProbabilidad] = ...
   FuncionProbabilidadesRuletaInvertirCol(M,PoblacionInicial);

% % % Seleccionamos individuos que cruzaremos
[IndicesPadres] = FuncionSeleccionIndividuosRuleta( ...
   PoblacionIndiceValorProbabilidad,CantidadPoblacion);

% % % Cruzamiento
[PoblacionHijos] = ...
   FuncionNuevoCruce(M,IndicesPadres,PoblacionInicial,k); ...
   %ParametroQuePodemosModificar

% % % Mutacion
% ¬¬¬¬¬¬¬¬¬¬ UNA SOLUCIoN PUEDE SUFRIR "g" MUTACIONES ¬¬¬¬¬¬¬¬¬¬
g = ceil(k/4); %    %ParametroQuePodemosModificar
[PoblacionNueva] = ...
   FuncionPoblacionMutacion_v3(PoblacionHijos,ProbabilidadMutacion,k,g);
% ¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬

% % % Iteracion de Datos
PoblacionInicial = PoblacionNueva;

% >>> CaLCULO MEJOR SOLUCION DE TODAS LAS GENERACIONES(menos primer ...
   generacion) >>>
if mod(i,cada_gener) == 0

```

```
84      i_aux = i_aux + 1;
85      Cobijas_por_generacion_SinQuitarRectangulos{i_aux} = ...
            PoblacionInicial; %
86
87  for iw = 1:CantidadPoblacion
88      valorSol_iw = FuncionValorManta(M,PoblacionInicial{iw}{1});
89      PoblacionValues(iw) = valorSol_iw;
90      IteracionesComparaSolucion = IteracionesComparaSolucion + 1;
91      IteracionesComparaSolucionGlobal = ...
            IteracionesComparaSolucionGlobal + 1;
92      if ( valorSol_iw < valorLaMejorSolucion )
93          LaMejorSolucion = PoblacionInicial{iw};
94          valorLaMejorSolucion = valorSol_iw; %NOTA: QUITAR EL ";" ...
                PARA VER EVOLUCIoN DE LA MEJOR SOLUCIoN
95          IteracionesCambiaSolucion = IteracionesCambiaSolucion + 1;
96          IteracionesCambiaSolucionGlobal = ...
                IteracionesCambiaSolucionGlobal + 1;
97          IteracionCambioSolucionGlobal = ...
                [IteracionCambioSolucionGlobal,i];
98          valorIteracionCambioSolucionGlobal = ...
                [valorIteracionCambioSolucionGlobal,valorLaMejorSolucion];
99      end
100 end
101 valores_por_generacion_SinQuitarRectangulos{i_aux} = ...
        PoblacionValues; %
102 valor_Mejor_Cobija_Hasta_El_Momento(i_aux) = valorLaMejorSolucion; % ...

103
104 else %if mod(i,cada_gener) ≠0
105
106 for iw = 1:CantidadPoblacion
107     valorSol_iw = FuncionValorManta(M,PoblacionInicial{iw}{1});
108     IteracionesComparaSolucion = IteracionesComparaSolucion + 1;
109     IteracionesComparaSolucionGlobal = ...
            IteracionesComparaSolucionGlobal + 1;
110     if ( valorSol_iw < valorLaMejorSolucion )
111         LaMejorSolucion = PoblacionInicial{iw};
112         valorLaMejorSolucion = valorSol_iw; %NOTA: QUITAR EL ";" ...
                PARA VER EVOLUCIoN DE LA MEJOR SOLUCIoN
113         IteracionesCambiaSolucion = IteracionesCambiaSolucion + 1;
114         IteracionesCambiaSolucionGlobal = ...
                IteracionesCambiaSolucionGlobal + 1;
115         IteracionCambioSolucionGlobal = ...
                [IteracionCambioSolucionGlobal,i];
116         valorIteracionCambioSolucionGlobal = ...
                [valorIteracionCambioSolucionGlobal,valorLaMejorSolucion];
117     end
118 end
119
120 end
121 % >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
122
123 tiempo_por_generacion(i) = toc; %
124
125 end % Fin iter generaciones-10
```

```
for i = (Generaciones-10+1):Generaciones % Inicio iter generaciones
    tic; %inicio conteo  %

% % % Calculamos Probabilidades por el metodo de la ruleta
[PoblacionIndiceValorProbabilidad] = ...
    FuncionProbabilidadesRuletaInvertirCol(M,PoblacionInicial);

% % % Seleccionamos individuos que cruzaremos
[IndicesPadres] = FuncionSeleccionIndividuosRuleta( ...
    PoblacionIndiceValorProbabilidad,CantidadPoblacion);

% % % Cruzamiento
[PoblacionHijos] = ...
    FuncionNuevoCruce(M,IndicesPadres,PoblacionInicial,k); ...
    %ParametroQuePodemosModificar

% % % Mutacion
% ¬¬¬¬¬¬¬¬¬¬ UNA SOLUCIoN PUEDE SUFRIR "g" MUTACIONES ¬¬¬¬¬¬¬¬¬¬¬
g = ceil(k/4); %    %ParametroQuePodemosModificar
[PoblacionNueva] = ...
    FuncionPoblacionMutacion_v3(PoblacionHijos,ProbabilidadMutacion,k,g);
% ¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬

% % % Iteracion de Datos
PoblacionInicial = PoblacionNueva;

% >>> CaLCULO MEJOR SOLUCION DE TODAS LAS GENERACIONES(menos primer ...
    generacion) >>>
if mod(i,cada_gener) == 0

    i_aux = i_aux + 1;

for iw = 1:CantidadPoblacion
    Cobija_QuitoRect = ...
        FuncionEliminaRectangulosNoAportan(PoblacionInicial{iw},M,k);
    valorSol_iw = FuncionValorManta(M,Cobija_QuitoRect{1});
    PoblacionValues(iw) = valorSol_iw;
    PoblacionCobijasQuitoRect{iw} = Cobija_QuitoRect;
    IteracionesComparaSolucion = IteracionesComparaSolucion + 1;
    IteracionesComparaSolucionGlobal = ...
        IteracionesComparaSolucionGlobal + 1;
    if ( valorSol_iw < valorLaMejorSolucion )
        LaMejorSolucion = Cobija_QuitoRect;
        valorLaMejorSolucion = valorSol_iw; %NOTA: QUITAR EL ";" ...
            PARA VER EVOLUCIoN DE LA MEJOR SOLUCIoN
        IteracionesCambiaSolucion = IteracionesCambiaSolucion + 1;
        IteracionesCambiaSolucionGlobal = ...
            IteracionesCambiaSolucionGlobal + 1;
        IteracionCambioSolucionGlobal = ...
            [IteracionCambioSolucionGlobal,i];
        valorIteracionCambioSolucionGlobal = ...
            [valorIteracionCambioSolucionGlobal,valorLaMejorSolucion];
```

```
        end
    end
    valores_por_generacion_SinQuitarRectangulos{i_aux} = ...
        PoblacionValues; %
    valor_Mejor_Cobija_Hasta_El_Momento(i_aux) = valorLaMejorSolucion; % ...

    Cobijas_por_generacion_SinQuitarRectangulos{i_aux} = ...
        PoblacionCobijasQuitoRect; %

else %if mod(i,cada_gener) ≠0

    for iw = 1:CantidadPoblacion
        Cobija_QuitoRect = ...
            FuncionEliminaRectangulosNoAportan(PoblacionInicial{iw},M,k);
        valorSol_iw = FuncionValorManta(M,Cobija_QuitoRect{1});
        IteracionesComparaSolucion = IteracionesComparaSolucion + 1;
        IteracionesComparaSolucionGlobal = ...
            IteracionesComparaSolucionGlobal + 1;
        if ( valorSol_iw < valorLaMejorSolucion )
            LaMejorSolucion = Cobija_QuitoRect;
            valorLaMejorSolucion = valorSol_iw; %NOTA: QUITAR EL ";" ...
                PARA VER EVOLUCIoN DE LA MEJOR SOLUCIoN
            IteracionesCambiaSolucion = IteracionesCambiaSolucion + 1;
            IteracionesCambiaSolucionGlobal = ...
                IteracionesCambiaSolucionGlobal + 1;
            IteracionCambioSolucionGlobal = ...
                [IteracionCambioSolucionGlobal,i];
            valorIteracionCambioSolucionGlobal = ...
                [valorIteracionCambioSolucionGlobal,valorLaMejorSolucion];
        end
    end

end
% >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

tiempo_por_generacion(i) = toc; %

end % Fin iter generaciones


% ...

% FIN ITERACIoN POR GENERACIONES
% ...


[IteracionCambioSolucionGlobal,index] = ...
    unique(IteracionCambioSolucionGlobal,'last');
valorIteracionCambioSolucionGlobal = ...
    valorIteracionCambioSolucionGlobal(index');

tiempo_acumulado_por_generacion(1) = tiempo_por_generacion(1); %
for time = 2:Generaciones %
    tiempo_acumulado_por_generacion(time) = ...
        tiempo_acumulado_por_generacion(time-1) + ...
```

```
        tiempo_por_generacion(time); %
end %

for time = 1:Generaciones
    if mod(Generaciones+1-time,cada_gener) ~=0
        tiempo_acumulado_por_generacion(Generaciones+1-time) = []; %
    end
end
Longitud = length(tiempo_acumulado_por_generacion); %


for kz = 1:Longitud %
    orden_valores_por_generacion_SinQuitarRectangulos{kz} = ...
        sort(valores_por_generacion_SinQuitarRectangulos{kz});
    mejores_70_valores_por_generacion_SinQuitarRectangulos{kz} = ...
        orden_valores_por_generacion_SinQuitarRectangulos{kz}([1:70]);
    promedio_mejores_70_valores_por_generacion_SinQuitarRectangulos( ...
        kz) = ...
        mean(mejores_70_valores_por_generacion_SinQuitarRectangulos{kz});
end %


% Mejores 20 Mantas ultima generacion
%Ordenamos de manera ascendente
IndicesPoblacion = [1:CantidadPoblacion]; %vector que indica indice ...
    de cada solucion
ValoresPoblacion = valores_por_generacion_SinQuitarRectangulos{i_aux};
PoblacionIndiceValor = [IndicesPoblacion',ValoresPoblacion']; ...
    %matriz de LongitudPoblacionx2: primer columna:indices; segunda ...
    columna: valores
[PoblacionObtenida,inddice] = sort(PoblacionIndiceValor(:,2)); ...
    %ordenamos matriz de forma ascendente respecto la segunda columna.
%PoblacionObtenida indica el valor una vez ordenado; indice indica ...
    que la entrada i del vector IndicesPoblacion se mueve a la ...
    entrada indice(i)
%,esto es indice(i) indica a donde manda (nueva posicion) la entrada ...
    i de IndicesPoblacion para que corresponda al ordenamiento requerido
PoblacionObtenida = ...
    [PoblacionIndiceValor(inddice),PoblacionObtenida]; %matriz de ...
    LongitudPoblacionx2. cuya primer columna indica el
%indice y la segunda columna el valor de la solucion respectiva una ...
    vez ordenado respecto a los valores
cobijasUltimaGeneracion = ...
    Cobijas_por_generacion_SinQuitarRectangulos{i_aux}
for inx = 1:(ceil(CantidadPoblacion/2))
cobijasUltimaGeneracionOrdenada{inx} = ...
    cobijasUltimaGeneracion{PoblacionObtenida(inx,1)};
valoresUltimaGeneracionOrdenada(inx) = PoblacionObtenida(inx,2);
end


vector_Generaciones = 1:Longitud;
vector_Generaciones = vector_Generaciones * cada_gener;

disp("--- Algoritmos Geneticos --- ")
```

```
248 disp("Valor de la mejor solucion global (de todas las iteraciones y ...
       todas las generaciones): " + valorLaMejorSolucion)
249
250 %En esta version se guarda el promedio de valores de cada ...
       generacion,  -_
251 %la mejor solucion global (sin comparar con soluciones iniciales),  >
252 %aleatoriamente se escoge el padre que intercambia rectangulos y
253 %una solucion puede sufrir "g" mutaciones
254
255 tiempo_acumulado_por_generacion = tiempo_acumulado_por_generacion';
256 valor_Mejor_Cobija_Hasta_El_Momento = ...
       valor_Mejor_Cobija_Hasta_El_Momento';
257 promedio_mejores_70_valores_por_generacion_SinQuitarRectangulos = ...
       promedio_mejores_70_valores_por_generacion_SinQuitarRectangulos';
258 vector_Generaciones = vector_Generaciones';
259
260 end
```

FuncionSolucionAleatoria

```
1  function [Solucion] = FuncionSolucionAleatoria(M,k)
2
3  [b1,b2] = size(M);
4  A = {[1,1,b1,b2]}; %cellArray que ira guardando vectores donde cada ...
       vector
5  %consta de 4 entradas, las cuales indican coordenadas de cuadrados que
6  %realizaran una particion del cuadro de ceros con dimension dim(M)
7  B = {}; %cellArray que guarda rectangulos de 1x1, que son los que no ...
       pueden hacerse mas peque os
8  r = 1; %sirve para guardar entradas en A
9  t = 1; %sirve para guardar entradas en B
10
11 if k≥2
12     for i = 2:k
13         [s1,s2] = size(A);
14         indiceVector = round( 1 + (s2-1)*rand ); %seleccion ...
               aleatoria del indice de un vector en el cellArray A
15         vector = A{indiceVector}; %vector que se escogio de manera ...
               aleatoria
16         v1 = vector(3)-vector(1)+1;
17         v2 = vector(4)-vector(2)+1;
18         %%% dim(vector) = [v1,v2] = [b1 - a1 + 1, b2 - a2 + 1]
19
20         if isequal(v1,1)
21             random = 1;
22         elseif isequal(v2,1)
23             random = 0;
24         else %if v1>1 & v2>1
25             random = round(rand);
26         end
27         %%% random es 0 o 1 ( si v1,v2>1 se escoge aleatoriamente).
28         %%% 0 -> corte horizontal
```

```
29          %%% 1 -> corte vertical
30
31          % --------------- Construccion vectores nuevos ---------------
32          if isequal(0,random) %corte horizontal
33              indiceCorte = round( vector(1) + ( (vector(3)-1) - ...
                    vector(1) )*rand );
34
35              vector1 = [vector(1),vector(2),indiceCorte,vector(4)];
36              vector2 = [indiceCorte+1,vector(2),vector(3),vector(4)];
37              sizev1a = vector1(3)-vector1(1)+1;
38              sizev1b = vector1(4)-vector1(2)+1;
39              sizev2a = vector2(3)-vector2(1)+1;
40              sizev2b = vector2(4)-vector2(2)+1;
41          %%% dim(vector) = [v1,v2] = [b1 - a1 + 1, b2 - a2 + 1]
42
43          if ( (sizev1a>1 || sizev1b>1) & (sizev2a>1 || sizev2b>1) ) ...
                %si las dos matrices construidas por vectores no son ...
                cuadrito de 1x1
44              A(:,indiceVector) = [];
45              A{r} = vector1;
46              A{r+1} = vector2;
47              r = r+1;
48          elseif (sizev1a==1 & sizev1b==1 & sizev2a==1 & sizev2b==1) ...
                %si los dos son cuadritos de 1x1
49              B{t} = vector1;
50              B{t+1} = vector2;
51              t = t+2;
52              A(:,indiceVector) = [];
53              r = r-1;
54          elseif (sizev1a==1 & sizev1b==1 & (sizev2a>1 || sizev2b>1)) ...
                %si uno es cuadrito 1x1 pero el otro es mas grande que ...
                cuadrito de 1x1
55              B{t} = vector1;
56              t = t+1;
57              A(:,indiceVector) = [];
58              A{r} = vector2;
59              % r = r
60          else %if ((sizev1a>1 || sizev1b>1) & sizev2a==1 & ...
                sizev2b==1) %si uno es mas grande que cuadrito de 1x1 ...
                pero el otro es cuadrito 1x1
61              A(:,indiceVector) = [];
62              A{r} = vector1;
63              % r = r
64              B{t} = vector2;
65              t = t+1;
66          end
67
68          else %if isequal(1,random) %corte vertical
69              indiceCorte = round( vector(2) + ( (vector(4)-1) - ...
                    vector(2) )*rand );
70
71              vector1 = [vector(1),vector(2),vector(3),indiceCorte];
72              vector2 = [vector(1),indiceCorte+1,vector(3),vector(4)];
73              sizev1a = vector1(3)-vector1(1)+1;
74              sizev1b = vector1(4)-vector1(2)+1;
```

```
75              sizev2a = vector2(3)-vector2(1)+1;
76              sizev2b = vector2(4)-vector2(2)+1;
77              %%% dim(vector) = [v1,v2] = [b1 - a1 + 1, b2 - a2 + 1]
78
79              if ( (sizev1a>1 || sizev1b>1) & (sizev2a>1 || sizev2b>1) ...
                    ) %si las dos matrices construidas por vectores no ...
                    son cuadrito de 1x1
80                  A(:,indiceVector) = [];
81                  A{r} = vector1;
82                  A{r+1} = vector2;
83                  r = r+1;
84              elseif (sizev1a==1 & sizev1b==1 & sizev2a==1 & ...
                    sizev2b==1) %si los dos son cuadritos de 1x1
85                  B{t} = vector1;
86                  B{t+1} = vector2;
87                  t = t+2;
88                  A(:,indiceVector) = [];
89                  r = r-1;
90              elseif (sizev1a==1 & sizev1b==1 & (sizev2a>1 || ...
                    sizev2b>1)) %si uno es cuadrito 1x1 pero el otro es ...
                    mas grande que cuadrito de 1x1
91                  B{t} = vector1;
92                  t = t+1;
93                  A(:,indiceVector) = [];
94                  A{r} = vector2;
95                  % r = r
96              else %if ((sizev1a>1 || sizev1b>1) & sizev2a==1 & ...
                    sizev2b==1) %si uno es mas grande que cuadrito de 1x1 ...
                    pero el otro es cuadrito 1x1
97                  A(:,indiceVector) = [];
98                  A{r} = vector1;
99                  % r = r
100                 B{t} = vector2;
101                 t = t+1;
102             end
103         end
104
105     end
106
107 end
108
109 %%% Hasta aqui obtenemos A y B, dos cellArrays con vectores donde A
110 %%% contiene vectores que codifican rectangulos que no son de ...
       dimension 1x1
111 %%% y B vectores que codifican rectangulos de dimension 1x1.
112
113 [a1,a2] = size(A);
114 [b1,b2] = size(B);
115 Particion = {};
116 for i = 1:a2
117     Particion{i} = A{i};
118 end
119
120 if b2>0
121     for i = 1:b2
```

```
122         Particion{a2+i} = B{i};
123     end
124 end
125
126 % Particion consiste en la particion de la matriz codificada con ...
      vectores
127
128 % - _ - _ - _ - Definir los cuadrados aleatoriamente - _ - _ - _ -
129 [p1,p2] = size(Particion);
130 Cuadrados = {};
131
132 for i = 1:p2
133     alpha1 = round( Particion{i}(1) + (Particion{i}(3) - ...
          Particion{i}(1))*rand );
134     alpha2 = round( alpha1 + (Particion{i}(3) - alpha1)*rand );
135     beta1 = round( Particion{i}(2) + (Particion{i}(4) - ...
          Particion{i}(2))*rand );
136     beta2 = round( beta1 + (Particion{i}(4) - beta1)*rand );
137     [Cuadrados{i}] = ...
          FuncionPosicionRectangulo(M,alpha1,beta1,alpha2,beta2);
138 end
139
140 % -- -- -- -- -- -- -- Construccion de la solucion -- -- -- -- -- -- --
141 MatrizSol = zeros(size(M));
142 for i = 1:p2
143     MatrizSol = MatrizSol + Cuadrados{i};
144 end
145
146 Solucion{1} = MatrizSol;
147 Solucion{2} = Cuadrados;
148
149 end
```

FuncionPosicionRectangulo

```
1  function [RectanguloUnosCeros] = ...
       FuncionPosicionRectangulo(Manta,a1,a2,b1,b2)
2  % Funcion devuelve Matriz de dimension igual a dim(Manta), donde
3  % el rectangulo con coordenadas de la esquina superior izquierda
4  % es (a1,a2) y de la esquina inferior derecha es (b1,b2) esta llena
5  % de unos, mientras que las demas entradas estan llenas de ceros.
6
7  [r1,r2] = size(Manta); %dimension de la matriz Manta
8  CerosArr = zeros(a1-1,r2);
9  CerosIzq = zeros(b1-a1+1,a2-1);
10 CerosDer = zeros(b1-a1+1,r2-b2);
11 CerosAba = zeros(r1-b1,r2);
12 UnosMed = ones(b1-a1+1,b2-a2+1);
13
14  M_Med = [CerosIzq,UnosMed,CerosDer];
15
16 RectanguloUnosCeros = [CerosArr;M_Med;CerosAba];
```

```
17
18 end
```

FuncionProbabilidadesRuletaInvertirCol

```
1  function [PoblacionProbabilidad] = ...
       FuncionProbabilidadesRuletaInvertirCol(IO,PoblacionInicial)
2  % % % Calculo de las probabilidades. Se usara en el metodo de ...
       seleccion: ruleta
3
4  % Ordenamos de manera ascendente
5  LongitudPoblacion = length(PoblacionInicial); %cantidad de ...
       soluciones que contiene PoblacionInicial
6  IndicesPoblacion = [1:LongitudPoblacion]; %vector que indica indice ...
       de cada solucion
7  for i = 1:LongitudPoblacion %inicio ciclo for
8      ValoresPoblacion(i) = ...
           FuncionValorManta(PoblacionInicial{i}{1},IO); %vector cuya ...
           entrada i contiene el valor de la solucion i
9  end %fin ciclo for
10 PoblacionIndiceValor = [IndicesPoblacion',ValoresPoblacion']; ...
       %matriz de LongitudPoblacionx2: primer columna:indices; segunda ...
       columna: valores
11 [PoblacionProbabilidad,indice] = sort(PoblacionIndiceValor(:,2)); ...
       %ordenamos matriz de forma ascendente respecto la segunda columna.
12 %PoblacionProbabilidad indica el valor una vez ordenado; indice ...
       indica que la entrada i del vector IndicesPoblacion se mueve a la ...
       entrada indice(i)
13 %,esto es indice(i) indica a donde manda (nueva posicion) la entrada ...
       i de IndicesPoblacion para que corresponda al ordenamiento requerido
14 PoblacionProbabilidad = ...
       [PoblacionIndiceValor(indice),PoblacionProbabilidad]; %matriz de ...
       LongitudPoblacionx2. cuya primer columna indica el
15 %indice y la segunda columna el valor de la solucion respectiva una ...
       vez ordenado respecto a los valores
16
17 % Calculamos probabilidades para usar en el metodo ruleta
18
19 % % % NOTA: Como en el Problema de la Manta Rectangular se busca
20 % minimizar la f.o., entonces lo que usaremos para asignar
21 % probabilidades es: ordenar de menor a mayor los valores. El
22 % resultado de dividir el menor valor entre la suma de los valores
23 % se lo asignaremos a la solucion cuyo valor asociado es el mas
24 % grande. Luego, dividimos el segundo menor valor entre la suma
25 % de todos los valores y la probabilidad obtenida se la asignamos
26 % al penultimo menor valor y asi sucesivamente hasta asignar el
27 % resultado de dividir el mayor valor entre la suma de todos los
28 % valores a la solucion que tiene asociado el menor valor.
29 % % %
30
31 TotalSumaValores = sum( PoblacionProbabilidad(:,2) ); %suma de todos ...
       los valores
```

```
for i = 1:LongitudPoblacion %inicio ciclo for
    valori = PoblacionProbabilidad(i,2); %calculamos valor de la ...
       solucion PoblacionInicial(PoblacionProbabilidad(i,2))
    Probabilidad(LongitudPoblacion + 1 - i) = ...
       valori/TotalSumaValores; %probabilidad asignada
end %fin ciclo for

PoblacionProbabilidad = [PoblacionProbabilidad,Probabilidad']; ...
   %matriz de LongitudPoblacionx3
%cuya primer columna indica indice de la solucion; segunda columna ...
   el valor respectivo
%y la tercer columna su probabilidad de seleccion asignada

end
```

FuncionSeleccionIndividuosRuleta

```
function [IndicesPosiblesPadres] = FuncionSeleccionIndividuosRuleta( ...
   PoblacionIndiceValorProbabilidad,CantidadPoblacion)
%Seleccion de individuos: Ruleta

LongitudPoblacion = length(PoblacionIndiceValorProbabilidad);
IntervaloProbabilistico = [0];
for i = 1:LongitudPoblacion
    IntervaloProbabilistico(i+1) = ...
       PoblacionIndiceValorProbabilidad(i,3) + ...
       IntervaloProbabilistico(i);
end

for t = 1:CantidadPoblacion

    random = rand;
    i = 1;

    while i < length(IntervaloProbabilistico)
        if (IntervaloProbabilistico(i) < random) && (random ≤ ...
           IntervaloProbabilistico(i+1))   %( ...
           IntervaloProbabilistico(i) < random ≤ ...
           IntervaloProbabilistico(i+1) )
            IndiceSolucion = PoblacionIndiceValorProbabilidad(i,1);
            i = length(IntervaloProbabilistico); %para finalizar ...
               while si se cumple la condici n del if
        else
            i = i + 1;
        end
    end

    IndicesPosiblesPadres(t) = IndiceSolucion;
end

end
```

FuncionNuevoCruce

```
function [PoblacionHijos] = ...
    FuncionNuevoCruce(IO,IndicesPadres,PoblacionConsiderada,k)

% % % En este cruce se juntan todos los rectangulos de Padre1 y
% Padre2 y se acomodan del mejor al peor. Se agrega el primero a
% Hijo1. Posteriormente vamos agregando uno por uno los que cumplan
% factibilidad hasta construir la solucion Hijo1 con k rectangulos.
% Hijo2 es una solucion aleatoria

if k ~= 1
    LongitudPoblacion = length(IndicesPadres);
    % % % criterio detenerse
    dosk = 2*k;
    MaxRectDetener = k + ceil(k/4); %ParametroQuePodemosModificar
    MaxRectDetenerMasUno = MaxRectDetener + 1;
    % % %

    for indice = 1:2:LongitudPoblacion %i de 1 a LongitudPoblacion ...
        con incrementos de 2 en 2, esto para que se cruce el 1 con el ...
        2, 3 con 4, 5 con 6 y asi sucesivamente
        Padre1 = PoblacionConsiderada{IndicesPadres(indice)}; ...
            %escogemos la solucion PoblacionConsiderada=z, con ...
            z=IndicesPadres(indice)
        Padre2 = PoblacionConsiderada{IndicesPadres(indice+1)}; ...
            %escogemos la solucion PoblacionConsiderada=w, con ...
            w=IndicesPadres(indice+1)

        TodosRectangulos = [Padre1{2},Padre2{2}];
        for i = 1:dosk
            Valores(i,1) = i;
            Valores(i,2) = FuncionValorManta(IO,TodosRectangulos{i});
        end
        %primer columna de Valores tiene indices; segunda columna de ...
            Valores tiene los indices
        [Values,Indexes] = sort(Valores(:,2)); %ordenamos matriz de ...
            forma ascendente respecto la segunda columna.

        Hijo1 = {};
        Hijo1{2}{1} = TodosRectangulos{Indexes(1,1)};
        Hijo1{1} = zeros(size(Hijo1{2}{1}));
        contadorRectangulos = 1;
        contador = 2;
        while contadorRectangulos < k & contador ~= MaxRectDetenerMasUno
            Hijo1{2}{contadorRectangulos+1} = ...
                TodosRectangulos{Indexes(contador,1)};
            [factibleSiNo] = ...
                FuncionVerificaFactibilidadSumaRectangulos(Hijo1, ...
                contadorRectangulos+1);
            if isequal(1,factibleSiNo) %si Si es factible
                contadorRectangulos = contadorRectangulos + 1;
```

```
39                  contador = contador + 1;
40              else %if isequal(0,factibleSiNo) %si NO es factible
41                  contador = contador + 1;
42              end
43          end
44
45          if contadorRectangulos < k & contador == ...
               MaxRectDetenerMasUno %si aun no tenemos k rectangulos y ...
               ademas ya no hay opciones de rectangulos para agregar
46              valorPadre1 = FuncionValorManta(IO,Padre1{1});
47              valorPadre2 = FuncionValorManta(IO,Padre2{1});
48              if valorPadre1 < valorPadre2
49                  Hijo1 = Padre1;
50              else %if valorPadre1 ≥ valorPadre2
51                  Hijo1 = Padre2;
52              end
53          end
54          %en este ciclo if se ve si no se pudo construir solucion ...
               Hijo1 en
55          %el anterior ciclo while, y si fue asi entonces ...
               seleccionamos por
56          %torneo, esto es, Hijo1 sera igual al mejor padre
57
58          if contadorRectangulos == k %si se construyo Hijo1 con ...
               while, entonces solo encontramos Hijo{1}, pero si no se pudo
59              %construir en el ciclo while entonces ya tenemos ...
                   Hijo1=Padre1, por lo que nos saltamos este ciclo if
60              MatrizFact = Hijo1{1}; %nos servira para sumar todos los ...
                   rectangulos y obtener el real Hijo{1}, pues antes no ...
                   lo actualizamos
61              for p = 1:k
62                  MatrizFact = MatrizFact + Hijo1{2}{p};
63              end
64              Hijo1{1} = MatrizFact;
65          end
66              [Hijo2] = FuncionSolucionAleatoria(IO,k);
67
68              PoblacionHijos{indice} = Hijo1;
69              PoblacionHijos{indice+1} = Hijo2;
70      end
71
72      %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
73
74  else %if k == 1
75
76      LongitudPoblacion = length(IndicesPadres);
77      ControlIndice = 0;
78
79      for indice = 1:2:LongitudPoblacion %i de 1 a LongitudPoblacion ...
           con incrementos de 2 en 2, esto para que se cruce el 1 con el ...
           2, 3 con 4, 5 con 6 y asi sucesivamente
80
81          Padre1 = PoblacionConsiderada{IndicesPadres(indice)}{1}; ...
               %escogemos la solucion PoblacionConsiderada=t, con ...
               t=IndicesPadres(indice)
```

```
82          Padre2 = PoblacionConsiderada{IndicesPadres(indice+1)}{1}; ...
               %escogemos la solucion PoblacionConsiderada=r, con ...
               r=IndicesPadres(indice+1)
83
84          [FirstRow1,FirstCol1] = find(Padre1,1); %devuelve los ...
               indices de la entrada donde aparece el primer "1" en Padre1
85          [LastRow1,LastCol1] = find(Padre1,1,'last'); %devuelve los ...
               indices de la entrada donde aparece el ultimo "1" en Padre1
86          [FirstRow2,FirstCol2] = find(Padre2,1); %devuelve los ...
               indices de la entrada donde aparece el primer "1" en Padre2
87          [LastRow2,LastCol2] = find(Padre2,1,'last'); %devuelve los ...
               indices de la entrada donde aparece el ultimo "1" en Padre2
88
89          Padre1DimAncho = LastCol1 - FirstCol1 +1;
90          Padre1DimAltura = LastRow1 - FirstRow1 +1;
91          Padre2DimAncho = LastCol2 - FirstCol2 +1;
92          Padre2DimAltura = LastRow2 - FirstRow2 +1;
93
94          randomz = rand;
95                NumCorreccionRedondeoAleatorio = -0.1; % % %
96          HijoDimAncho = round( (Padre1DimAncho + Padre2DimAncho)/2 + ...
               NumCorreccionRedondeoAleatorio );
97          HijoDimAltura = round( (Padre1DimAltura + Padre2DimAltura)/2 ...
               + NumCorreccionRedondeoAleatorio );
98
99          randomy = rand;
100         NumCorreccionPosicionAleatorio = -0.1; % % %
101         FirstRowHijo = round( (FirstRow1+FirstRow2)/2 + ...
               NumCorreccionPosicionAleatorio);
102         FirstColHijo = round( (FirstCol1+FirstCol2)/2 + ...
               NumCorreccionPosicionAleatorio);
103
104         Hijo = IO - IO; %matriz de ceros con dimension igual a size(M)
105
106         for IndiceAltura = FirstRowHijo:(FirstRowHijo+HijoDimAltura-1)
107             for IndiceAncho = FirstColHijo:(FirstColHijo+HijoDimAncho-1)
108                 Hijo(IndiceAltura,IndiceAncho) = 1;
109             end
110         end
111
112         IndiceReal = indice - ControlIndice;
113         Descendientes{IndiceReal} = Hijo;
114
115         valorPadre1 = sum(sum(abs(IO-Padre1)));
116         valorPadre2 = sum(sum(abs(IO-Padre2)));
117
118         if valorPadre1 < valorPadre2
119             MejoresPadres{IndiceReal} = Padre1;
120         else %if valorPadre1 ≥ valorPadre2
121             MejoresPadres{IndiceReal} = Padre2;
122         end
123
124         ControlIndice = ControlIndice + 1;
125     end
126
```

```
127      for i = 1:length(Descendientes)
128          Decendents{i}{1} = Descendientes{i};
129          Decendents{i}{2}{1} = Descendientes{i};
130          BestParents{i}{1} = MejoresPadres{i};
131          BestParents{i}{2}{1} = MejoresPadres{i};
132      end
133
134        PoblacionHijos = [Decendents,BestParents];
135  end
136
137  end
```

FuncionVerificaFactibilidadSumaRectangulos

```
1  function [factibleSiNo] = ...
       FuncionVerificaFactibilidadSumaRectangulos(Manta,k)
2  % Funcion que verifica la factibilidad de Manta sin construir.
3  % La manera de hacerlo es buscar si tiene alguna entrada mayor
4  % o igual a 2. Si la tiene, entonces NO es factible, si todas
5  % son 0 o 1 Si es factible.
6
7  MatrizFact = zeros(size(Manta{1})); %nos servira para sumar todos ...
       los rectangulos
8  for i = 1:k
9      MatrizFact = MatrizFact + Manta{2}{i};
10 end
11
12 factibilidad = find(MatrizFact≥2,1);
13 if (factibilidad > 0)
14     factibleSiNo = 0;
15 else
16     factibleSiNo = 1;
17 end
18
19 end
```

FuncionPoblacionMutacion_v3

```
1  function [PoblacionConMutacion] = ...
       FuncionPoblacionMutacion_v3(PoblacionHijos,ProbabilidadMutacion,k,g)
2  % g es el numero de mutaciones que puede sufrir una solucion
3
4  LongitudPoblacionHijos = length(PoblacionHijos);
5  for i = 1:LongitudPoblacionHijos %corre sobre cada individuo de la ...
       poblacion (i-esimo individuo en PoblacionHijos)
6
7      if rand ≤ ProbabilidadMutacion %Si cumple condicion de mutacion, ...
           entonces muta
8          for h = 1:g
```

```
9               ModificacionAleatoria = randsample( ...
                    {"FuncionAchicarRectanguloAbajo", ...
                    "FuncionAchicarRectanguloArriba", ...
                    "FuncionAchicarRectanguloDerecha", ...
                    "FuncionAchicarRectanguloIzquierda", ...
                    "FuncionAgrandarRectanguloAbajo", ...
                    "FuncionAgrandarRectanguloArriba", ...
                    "FuncionAgrandarRectanguloDerecha", ...
                    "FuncionAgrandarRectanguloIzquierda"},1 );
10              %randsample escoge n elementos al azar de list por (list,n)
11              RealizarModificacion = str2func(ModificacionAleatoria{1});
12              MutarRectangulo = ceil(k * rand); %mutamos el ...
                    ceil(k*rand)-esimo rectangulo de la solucion
13              [RectanguloMutado] = RealizarModificacion( ...
                    PoblacionHijos{i}{2}{MutarRectangulo});
14              Solucion = PoblacionHijos{i}{1} - ...
                    PoblacionHijos{i}{2}{MutarRectangulo} + RectanguloMutado;
15              %%% PoblacionHijos{i}{1} := Matriz solucion
16              %%% PoblacionHijos{i}{2}{MutarRectangulo} := Rectangulo ...
                    que va a mutar
17              %%% RectanguloMutado := Rectangulo que dijimos que iba a ...
                    mutar pero ya con mutacion
18              %%% Solucion := Matriz solucion pero con el rectangulo ...
                    ya mutado
19              [factibleSiNo] = FuncionVerificaFactibilidadManta(Solucion);
20              if isequal(factibleSiNo,1) %si Si es factible
21                  PoblacionHijos{i}{2}{MutarRectangulo} = ...
                        RectanguloMutado;
22                  PoblacionHijos{i}{1} = Solucion;
23              end
24          end
25      end
26
27  end
28
29  PoblacionConMutacion = PoblacionHijos;
30
31  end
```