Universidad Nacional Autónoma de México

Facultad de Ciencias

Implementación de heurísticas para el problema de empaquetamiento de n esferas idénticas en un contenedor esférico

T E S I S

que para obtener el título de:

Licenciada en Matemáticas aplicadas

presenta:

Ana Karen Ochoa Chávez

Tutora: Claudia Orquídea López Soto

Ciudad de México 2021

1. Datos del alumno
   Ochoa
   Chávez
   Ana Karen
   5540383249
   Universidad Nacional Autónoma de México
   Facultad de Ciencias
   Matemáticas Aplicadas
   313508800

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

3. Datos del sinodal 1
   Dr.
   Javier
   Ramírez
   Rodríguez

4. Datos del sinodal 2
   M. en C.
   David Chaffrey
   Moreno
   Fernández

5. Datos del sinodal 3
   M. en I.
   Adrián
   Girard
   Islas

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

7. Datos del trabajo escrito
   Implementación de heurísticas para el problema de empaquetamiento de n esferas idénticas en un contenedor esférico
   108 p.
   2021

A mi papá...

Agradecimientos

A mi papá por apoyarme toda la vida; a mis hermanas Fer y Samy por no dejar que me rindiera; a Mane por estar ahí cada segundo y confiar en mí; a Moka por ser mi compañera incondicional, día y noche; a David y Claudia por tenerme tanta paciencia; a Yahir, José y Vic por hacerme reír durante todo este proceso; a Jurgen, Pam, Ana Paula, Alex, Andrea y Donovan por creer en mí; al Dr. Javier Ramírez Rodríguez, al M. en I. Adrián Girard Islas y a la Dra. Zaida Estefanía Alarcón Bernal por sus valiosos comentarios que ayudaron a enriquecer el contenido de esta tesis, a mis primas por echarme porras y a mi mamá, por cuidarme desde donde sea que esté.

Introducción

El empaquetamiento de objetos en contenedores de distintas formas es un tema que se discute frecuentemente en investigaciones recientes debido a la alta aplicabilidad de dichos modelos en áreas como la medicina [41], la industria [16] y [19], en la química [39], entre otras áreas del conocimiento. En el presente documento busco aproximar una solución a un problema de empaquetamiento, denominado *Empaquetamiento de n esferas idénticas en un contenedor esférico*, expuesto por Hifi Mhan y Yousef Labib [38], en el que se busca introducir n esferas del mismo radio en un contenedor, sin que éstas se traslapen entre sí, de manera que todas se encuentren dentro del envase y el tamaño del contenedor sea mínimo. El problema a tratar es un problema *NP-duro* [26], por lo que se hará uso de las metaheurísctcas *Búsqueda local*, *GRASP*, *Recocido Simulado*, *Búsqueda Tabú* y *Algoritmos genéticos* para intentar solucionarlo, teniendo en cuenta que, a pesar de que se lleguen a buenos resultados, no se asegura que se ha llegado al óptimo global.

Se mencionan además algunas formas en que científicos de todo el mundo han atacado este problema y se implementan los métodos ya mencionados en *RStudio* con el fin de comparar los resultados obtenidos mediante estos métodos contra los mejores conocidos actualmente y poder analizar el desempeño de las metaheurísticas no solo en la teoría, sino también en la práctica.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Planteamiento del problema

1.1. Introducción

En este capítulo se plantea de manera formal el problema con el que se trabaja a lo largo del documento, después de haber definido un importante concepto: el problema de optimización.

Problema de optimización

La optimización en el ámbito científico se refiere a encontrar la mejor solución posible para un problema determinado. El objetivo en un problema de optimización es encontrar el valor de las variables de decisión para las que una función objetivo alcanza su valor máximo o mínimo. El valor de las variables está generalmente sujeto a ciertas restricciones. Formalmente, este tipo de problemas se formulan de la siguiente manera: [9]

$$\min_{\vec{x}\epsilon R^n} f_i(\vec{x}), \qquad (i = 1, 2, ..., M) \tag{1.1}$$

$$s.a. \qquad h_j(\vec{x}) = 0, \qquad (j = 1, 2, ..., J) \tag{1.2}$$

$$g_k(\vec{x}) \leq 0, \qquad (k = 1, 2, ..., K),$$

donde $f_i(\vec{x})$, $h_j(\vec{x})$ y $g_k(\vec{x})$ son funciones del vector $\vec{x} = (x_1, x_2, ..., x_n)$.

La expresión (1.1) representa a la función objetivo (que se busca maximizar o minimizar) y la ecuación (1.2) refiere a las resticciones que determinan la factibilidad de la solución. Cuando las variables son discretas, decimos que trabajamos sobre un problema de *optimización combinatoria*.

Problema de empaquetamiento de n esferas idénticas

Un problema de empaquetamiento es un tipo de problema de optimización combinatoria que busca introducir un conjunto de n objetos de tamaño $s_1, s_2, ..., s_n$ en la menor cantidad de contenedores posible, de forma que todos los objetos estén dentro y no se sobrepase la capacidad de los contenedores. El problema con el que se trabaja a lo largo de este documento es precisamente uno de empaquetamiento, que se enuncia a continuación: [38]

Se tiene un conjunto N con n esferas del mismo radio (r), donde cada esfera i=1,....,n es representada por r_i=1. El objetivo es minimizar el radio R_0 de un contenedor esférico S, de manera que todo elemento $r_i \, \epsilon \, N$, esté dentro de S y no existan traslapes entre esferas. Formalmente, el problema se formula como:

$$min \; R_0 \tag{1.3}$$

sujeto a

$$d(i,j) \geq (r_i + r_j), \forall (i,j) \epsilon N^2, i < j \tag{1.4}$$

$$d(0,i) \leq (R_0 - r_i), \forall i \epsilon N \tag{1.5}$$

Aquí, $d(i,j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2}, (i,j) \; \epsilon \; N \times N, \; i \neq j$
$y d(0,i) = \sqrt{x_i^2 + y_i^2 + z_i^2}, \; i \; \epsilon \; N.$

En las ecuaciones anteriores, (1.3) corresponde a la función objetivo, la cual minimiza el tamaño del contenedor, (1.4) asegura que para cualesquiera dos esferas i,j con centros en $(x_i,\ y_i,\ z_i)$, $(x_j,\ y_j,\ z_j)$ respectivamente, éstas no se superponen, pues la distancia entre estas debe ser mayor o igual que la suma de sus radios. Finalmente, la expresión (1.5) asegura que la esfera i está dentro del contenedor, que denotamos con una S, pues la distancia entre el origen y la esfera i debe ser menor o igual que la diferencia entre el radio de S y r_i.

Capítulo 2

Antecedentes

Si bien no se sabe el origen exacto de los problemas de empaquetamiento, la primera referencia que se realiza a estos es la *Conjetura de Kepler* [13], que surgió en el siglo XVII cuando un marino inglés llamado Sir Walter Raleigh (1554-1618) preguntó a su asistente Thomas Harriot(1560-1621), en medio de un proyecto de colonización de América del Norte financiado por la reina Isabel I, cuántas balas de cañón podían apilarse en la cubierta de un barco utilizando el menor espacio posible. El matemático, intentando encontrar una respuesta, le escribió al astrónomo y matemático alemán Johannes Kepler (1571-1630), quien respondió que el sistema más adecuado para hacerlo, era el del apilamiento en forma de pirámide, "al igual que los fruteros colocan sus frutas", haciendo referencia particular a las naranjas. [13]

A principios del siglo XIX, el matemático Carl Friedrich Gauss (1777-1855) dio el primer resultado significativo, considerando empaquetamientos reticulares[1], es decir, aquellos en que las esferas forman patrones muy simétricos y ordenados, demostrando que estos son los de mayor densidad en el espacio. Esto último es crucial ya que la razón entre el volumen que ocupan las esferas y el volumen del contenedor es la densidad, por lo que el problema se reduce a encontrar la manera de llenar el contenedor de manera que la densidad sea máxima. La manera de acomodar las esferas es la siguiente: se empieza por colocar una capa de esferas (A) distribuidas en un retículo, con centros c. La siguiente capa de esferas (B) debe colocarse en los huecos que forman las esferas de la primera capa, sin embargo,

[1]El empaquetamiento reticular tiene la propiedad de que si sobre cualquier línea recta hay dos esferas separadas una distancia a, entonces hay esferas en todos los puntos a la misma distancia, a lo largo de la recta prolongada en ambos sentidos.

hay dos posibilidades para hacer esto: podemos colocar las nuevas esferas con centros en b o con centros en k (véase Figura 2.1). Para la capa C, se puede elegir nuevamente colocar las esferas de forma que tengan centro c, o bien, en el hueco que se dejó vacío en la capa anterior (b o k). Al posicionar las esferas de esta tercera capa sobre los centros c, tendríamos un empaquetamiento del estilo ABA, que recibe por nombre "Empaquetamiento hexagonal compacto". En caso de que decidiéramos poner las esferas centrándolas en los huecos que no han sido utilizados (sean b o k), formaríamos un empaquetamiento del estilo ABC, al que se le conoce como "Empaquetamiento cúbico compacto". El problema de resolver esta conjetura de esta manera es que la mayor parte de los empaquetamientos no son reticulares.

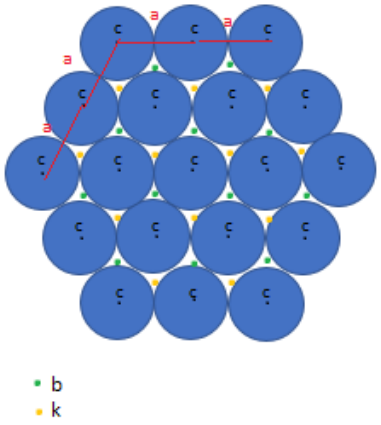


Figura 2.1: Ordenamiento reticular

En 1953, László Fejes Tóth (1915-2005) redujo el problema a una serie finita de cálculos para un determinado volumen de casos específicos. Planteó la división del espacio utilizando los diagramas de Voronoi:

Definición: Dado un conjunto de puntos $p_1, ..., p_n$ en el espacio, denominaremos diagrama de Voronoi a la subdivisión del plano en subregiones tales que la región i es el conjunto de puntos más cercanos a p_i que a cualquiera de los p_j, $j \neq i$, es decir, se divide el plano en tantas regiones como puntos p tengamos de tal forma que a cada punto le asignemos la

región formada por todo lo que está más cerca de él que de ningún otro. [6]

Sin embargo, László Tóth se dio cuenta que aunque había reducido la conjetura a un número finito de variables, no podía demostrarla, siendo el primero en sugerir que las computadoras podrían ayudar a su resolución.

Así, en 1998, el científico estadounidense Thomas Hales, utilizando la idea de separar el espacio en celdas de Voronoi, demostró lo que Kepler había afirmado en un prinicipio: que la mejor forma de acomodar las esferas usando el menor espacio posible, era apilándolas en forma de pirámide. Para esto, demostró que la máxima densidad que puede alcanzar un empaquetamiento de esferas en el espacio es

$$\frac{\pi}{\sqrt{18}} \simeq 0.74048 \tag{2.1}$$

que se da cuando una esfera está rodeada por otras 12, que es el máximo número de esferas que la pueden rodear, y redujo el problema al estudio de grafos planos, formulando una ecuación de 150 variables describiendo cerca de cinco mil agrupamientos de esferas distintos, y utilizando herramientas de programación lineal con entre mil y dos mil restricciones, además de un código de tres gigabites. Sin embargo, tras cinco años de revisión de la demostración (en 2003), hecha por 12 científicos de prestigio, los especialistas indicaron que aunque el 99 por ciento de la investigación era correcta, el porcentaje restante era imposible de verificar, cosa que sigue sin poderse hacer en la actualidad[43].

Los problemas de empaquetamiento se han ido expandiendo desde que Johannes Kepler comenzó a tratarlos y, en la actualidad, estos han sido estudiados por distintos científicos de maneras diferentes. Lochmann, Oger, y Stoyan (2006) publicaron un análisis estadístico para poder empaquetar n esferas con radios aleatorios, distintos entre sí, en un contenedor cilíndrico, utilizando un método de búsqueda llamado *Search method for crystalline sub-structures*, que se basa, como su nombre lo indica, en el acomodo de esferas como si éstas fueran átomos en una estructura cristalina.

Sutou y Dai. (2002) proponen una optimización global para el empaquetamiento de esferas de distintos radios en polítopos, y Stoyan, Yaskow, y Scheithauer (2003) describen

un modelo matemático para empaquetar esferas en un contenedor abierto, donde tanto la altura como el ancho de éste están fijos y la longitud es variable. Los autores proponen una búsqueda basada en crear vecindades sobre puntos extremos.

M'Hallah, Alkandari, y Mladenovic (2013) proponen una heurística basada en la Búsqueda de Vecindad Variable, que explora vecindarios distantes de la solución actual, y pasa de allí a una nueva vecindad si y solo si se realiza una mejora.

Soontrapa y Chen (2013), utilizan un método de búsqueda por Monte Carlo, que se basa mayormente en la aleatoriedad, para intentar resolver el problema de empaquetamiento de esferas idénticas en un contenedor esférico.

Birgin y Sobral (2008) propusieron dos modelos de programación no lineal, uno para contenedores esféricos y otro para circulares, utilizando un software llamado ALGENCAN, que no utiliza matrices y es capaz de resolver problemas de programación no lineal muy extensos, en un tiempo moderado. (AL es por las siglas en inglés Augmented Lagrangian y GENCAN es un código utilizado para minimizar funciones suaves).

Stoyan G., Scheithauer, G. y Yaskov, N. (2016) plantean una modificación del algoritmo *Búsqueda de salto* o *Jump Algorithm* para el problema de empaquetamiento de n esferas de distintos radios, en contenedores esférico, cúbico y cilíndrico circular. La idea básica del método utilizado es implementar una transición continua de un mínimo local a otro con un mejor valor en la función objetivo, saltándose algunos elementos de la solución.

En Liu J., Yao Y., Zheng Y., Geng H., Zhou G. (2009) se plantea el problema de empaquetamiento de objetos circulares/esféricos distintos entre sí, en contenedores circulares/esféricos cuyo radio se busca minimizar. El problema se plantea tanto en 2D como en 3D. Se utiliza un método conocido como "The energy landscape paving", que es un algoritmo de optimización global que puede dirigir la búsqueda de soluciones lejos de las regiones ya visitadas mediante funciones de penalización.

Finalmente, Hifi Mhan y Yousef Labib (2018) buscan dar una solución al problema de empaquetar n esferas del mismo radio en un contenedor esférico, minimizando el radio de este último. Para hacerlo, proponen un algoritmo que combina la "optimización enjambre de partículas" con un procedimiento de optimización local continua. La optimización enjambre de partículas se basa en una población donde se pueden explorar eficientemente diversos espacios de soluciones candidatas durante el proceso de búsqueda. Cada solución recibe el nombre de *partícula* y cada una tiene un valor de aptitud que representa el valor de la función objetivo en la función a optimizar. Además, cada partícula tiene una velocidad con la que se mueve en el espacio de búsqueda, siguiendo a las partículas "mejor posicionadas" de manera similar al vuelo de las aves, donde éstas se trasladan en un conjunto liderado por solo unas cuantas.

La optimización continua tiene como objetivo encontrar un conjunto de valores $x_1, x_2, \ldots, x_n$ que minimicen (o maximicen) una función $f(x_1, x_2, \ldots, x_n)$ posiblemente sujeto a una serie de restricciones sobre las variables.

El rendimiento del algoritmo propuesto se evalúa en un conjunto de puntos de referencia estándar, es decir, resultados ya conocidos, y los resultados obtenidos se comparan con estos. Los autores llegan a la conclusión de que sus resultados son bastante cercanos a los mejores conocidos, clasificando su método como competitivo.

Capítulo 3

Caracterización de problemas y representación de soluciones

Para cada problema de optimización combinatoria, suponemos que su conjunto de soluciones factibles F y su función de costo *c(S)* están dadas por dos algoritmos A_F y A_c respectivamente. Dado un objeto $S \epsilon 2^E$ y un conjunto P_F de parámetros, el algoritmo de reconocimiento A_F determina si el objeto S pertenece a F, que es el conjunto de soluciones factibles caracterizado por los parámetros en P_F. Análogamente, dada una solución factible $S \epsilon F$ y un conjunto de parámetros P_c, el algoritmo de la función de costo A_c calcula el valor de la función de costo *c(S)*. Así, podemos decir que cada problema de optimización combinatoria está caracterizado por el algoritmo de reconocimiento A_F y el algoritmo de la función de costo A_c, mientras que cada una de sus instancias está asociada con un par de conjuntos de parámetros P_F y P_c.

Para el problema de empaquetamiento de n esferas idénticas en un contenedor esférico, que fue enunciado en el capítulo *Planteamiento del problema*, identificamos lo siguiente:

- P_F: Los conjuntos de parámetros que establecen la factibilidad son $d(i, j)$ y $d(0, i)$.
- Un objeto que es candidato a ser una solución factible se caracteriza por un movimiento m de las esferas en S_0.
- P_c: El conjunto de parámetros en la función de costo es R_0.
- A_F: El algoritmo de reconocimiento A_F verifica que las esferas no se traslapan y que

están todas dentro del contenedor.

- A_c: Una vez que una solución factible ha sido establecida, el algoritmo de la función de costo A_c arroja un valor que calcula la función de costo.

Un problema tiene tres versiones:

- Problema de optimización: Dadas las representaciones de los conjuntos de parámetros P_F y P_c para algoritmos A_F y A_c respectivamente, encuentra una solución óptima factible. [45]

- Problema de evaluación: Dadas las representaciones para el conjunto de parámetros P_F y P_c para los algoritmos A_F y A_c encuentra el costo de una solución óptima factible. En este punto, suponiendo que A_c es un algoritmo polinomial (el costo de cualquier solución se calcula eficientemente), el problema de evaluación no puede ser más difícil que su versión de optimización, ya que una vez que se ha resuelto el problema de optimización y su solución óptima es conocida, su costo puede ser calculado en tiempo polinomial usando el algoritmo de la función de costos A_c. [45]

- Problema de decisión: Dadas las representaciones para los conjuntos de parámetros P_F y P_c para los algoritmos A_F y A_c, y un número entero B considerado como cota, ¿Existe una solución factible $S \epsilon F$ tal que $c(S) \leq B$? En este punto, el problema de decisión no puede ser más difícil que su respectivo problema de evaluación, pues una vez obtenido el costo de una solución óptima, se puede comparar con B para dar la respuesta al problema (sí o no). [45]

Para el problema que se trata en este documento, las versiones se definen de la siguiente manera:

- *Problema de optimización*: Dado un conjunto de esferas $S_0 = \{ 1,...,n \}$ con distancias no negativas *d(i,j)* entre cada par de esferas $i,j \epsilon S_0$, encuentra un acomodo adecuado de los elementos en S_0 tal que $R_0 =$ mín R: $d(i,j) \geq (r_i + r_j)$, $d(0,i) \leq (R_0 - r_i)$.

- *Problema de evaluación:* Dado un conjunto de esferas $S_0 = \{ 1,...,n \}$ con distancias no negativas *d(i,j)* entre cada par de esferas $i,j \epsilon S_0$, calula el radio de menor longitud del contenedor *S*.

- *Problema de decisión:* Dado un conjunto de esferas $S_0 = \{ 1,...,n \}$ con distancias no negativas *d(i,j)* entre cada par de esferas $i,j \epsilon S_0$, ¿Existirá una forma de acomodar n esferas idénticas dentro de un contenedor esférico de forma que el radio (R_0) de este cumpla que $R_0 \leq B$?, B un entero positivo.

3.1. Complejidad Computacional: Problemas P y NP

En éste capítulo vemos la definición de complejidad computacional, sus clasificaciones y una breve explicación de uno de los problemas más grandes de la historia de las matemáticas: ¿P=NP?

La *Complejidad Computacional* es una rama de la computación que busca determinar los recursos necesarios para resolver un problema dado, en un ordenador. Por un lado, si un cálculo requiere más tiempo que otro decimos que es más complejo temporalmente. Por otro lado, si requiere más espacio de almacenamiento que otro, decimos que es más complejo espacialmente.

En 1936, el matemático inglés Alan Turing introdujo en [53] la idea de un modelo formal de computador: la *máquina de Turing*, que es un dispositivo computacional con una cinta dividida en celdas que funciona como una memoria, un cabezal que escanea las celdas de la cinta, y un programa. En cada iteración, la máquina realiza un movimiento, indicado por el programa y por el símbolo escaneado por el cabezal. La máquina cambia su estado, escribe un símbolo en la celda escaneada y mueve el cabezal a la celda izquierda o derecha. [47]

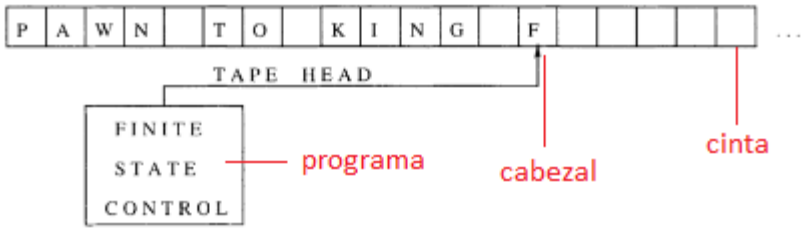


Figura 3.1: Máquina de Turing. [47]

Mediante este modelo y el análisis de la complejidad de los algoritmos, fue posible la categorización de problemas computacionales en *P*, *NP*, *NP-completo* y *NP-duro* de acuerdo a su comportamiento:

- **P** : Un problema pertenece a esta clase si tiene un algoritmo determinista[1] con complejidad polinomial que lo resuelve, es decir, si la relación entre el tamaño del problema y su tiempo de ejecución es polinómica.[2] [11]

- **NP**: Un problema de decisión es NP si existe un algoritmo no determinista[3] con complejidad polinomial que verifica si una solución al problema es válida. [11]

Otra clase de problemas es aquella cuyos elementos pueden ser resueltos de manera eficiente si suponemos que tenemos un algoritmo eficiente para resolver un segundo problema que nos ayude a resolver el primero. Ésta clase lleva el nombre de *NP-completos*.

Definición: Sean Q_1 y Q_2 dos problemas de decisión. Se dice que Q_1 se reduce en tiempo polinomial a Q_2 si y sólo si cumple alguna de las siguientes condiciones:

- Si existe un algoritmo A_1 de tiempo polinomial para Q_1 que use varios pasos o subrutinas de un algoritmo A_2, que es utilizado para resolver Q_2.

- Si se encuentra un algoritmo A_1 que transforme Q_1 al problema Q_2.

[1]El azar no está involucrado en el desarrollo de los futuros estados del problema, es decir, producirá siempre la misma salida a partir de las mismas condiciones iniciales

[2]En computación, cuando el tiempo de ejecución de un algoritmo (mediante el cual se obtiene la solución de un problema) es menor que cierto valor calculado a partir del número de variables implicadas (variables de entrada) usando una fórmula polinómica, se dice que dicho problema se puede resolver en un tiempo polinómico[12]

[3]Algoritmo que con la misma entrada ofrece muchos posibles resultados y no una solución única

Si Q_1 se reduce polinomialmente a Q_2, se denota $Q_1 \ \alpha \ Q_2$.

- Un problema Q es **NP-completo** si:
  - Q es NP.
  - Todos los problemas que son NP-completos se reducen polinomialmente a Q.
- Los problemas **NP-duros** son aquellos problemas de optimización tales que su problema de decisión es NP-completo.

En 1972, Richard Karp publicó [26], donde demostró que 21 problemas (conocidos como *Los 21 problemas de Karp*) son *NP-Completos*. El problema de decisión del problema de empaquetamiento se encuentra en la posición cuatro de esta lista. De esta manera, y con la definición anterior, el problema de empaquetamiento es **NP-duro**.

Existe un gran conflicto que concierne tanto a los problemas P como a los NP. En 1971, el matemático Húngaro-Americano John Von Neumann introdujo por primera vez uno de los problemas matemáticos más importantes hasta ahora en [10]: ¿Es todo problema NP también un problema P, en cuyo caso, P = NP? En decir, ¿será posible siempre encontrar un algoritmo eficiente que resuelva cualquier problema, o hay problemas para los que no existe algoritmo alguno que los resuelva? Desde ese entonces, cientos de científicos se han dedicado a intentar demostrarlo, de manera que incluso se volvió uno de los célebres "Problemas del Milenio". Si bien aún no existe una respuesta afirmativa a la pregunta en cuestión, tampoco se ha podido demostrar que no existan algoritmos polinomiales para los problemas NP, ni que éstos sean irresolubles. Es importante notar que, si para un problema de la clase *NP* se encontrara un algoritmo polinomial, entonces todos ellos se resolverían en tiempo polinomial y además NP colapsaría a P (P=NP), por lo que podríamos resolver en tiempos muy cortos, cientos de problemas útiles para los que hoy tenemos algoritmos muy costosos. [34]

Capítulo 4

Heurísticas

Después de haber manifestado la dificultad del problema de empaquetamiento en cuestión, en este capítulo se presenta un acercamiento sobre cómo se va a atacar este conflicto: con heurísticas.

La palabra *heurística* viene del griego *heuriskein*, cuyo significado es "el arte de descubrir nuevas estrategias o reglas para solucionar problemas" [52]. En la Investigación de Operaciones, definimos a las heurísticas como métodos que se limitan a proporcionar buenas soluciones a problemas complejos, sin dar necesariamente la solución óptima, pero en tiempos razonables. Se utilizan cuando el problema no tiene un método exacto para su solución o cuando el método exacto es computacionalmente muy costoso, por lo que son excelentes para tratar los ya mencionados problemas *NP-duros*. Son muy útiles también para eliminar rutas de búsqueda no prometedoras y explorar ampliamente espacios de búsqueda de soluciones.

Es muy común escuchar el término de *metaheurísticas* cuando se habla de técnicas de optimización. En [36] se califica de heurístico a "un procedimiento que encuentra soluciones de alta calidad con un coste computacional razonable, aunque no se garantice su optimalidad o su factibilidad", dando como ejemplo a la *búsqueda local*. Además, define a las metaheurísticas como "estrategias para diseñar y/o mejorar los procedimientos heurísticos orientados a obtener un alto rendimiento.", donde menciona al *Recocido Simulado*. Sin embargo, en [55] definen a las heurísticas como " métodos que exploran el espacio de soluciones construyendo caminos con el objetivo de encontrar la mejor solución al problema", poniendo como ejemplo al *Recocido Simulado* y a la *Búsqueda Tabú*. Es por esto por lo que, en este trabajo, se utilizarán los términos de *heurísticas* y *metaheurísticas* como sinónimos.

En el capítulo *Antecedentes* vimos que existen muchas formas de atacar el problema de empaquetamiento de esferas, así como algunas de sus variaciones. Pese a esto, lo que los ejemplos expuestos anteriormente tienen en común, es que utilizaron heurísticas para tratar de resolverlo, razón por la cual a lo largo de este documento se aborda el problema de empaquetamiento a través de métodos heurísticos.

A continuación, se presentan cuatro conceptos cruciales para el entendimiento de las heurísticas.

- *Definición:* Sea un conjunto finito E=$\{1,\ldots,n\}$. Recordemos que cualquier solución S para un problema de optimización combinatoria está definida por un subconjunto de elementos del conjunto E. Una *solución factible* es aquella que satisface todas las restricciones del problema.

Denotamos por F al conjunto de soluciones factibles y por $\bar{F}$ al conjunto formado por todos los subconjuntos de elementos del conjunto E (soluciones factibles y no factibles).

- *Definición:* La *vecindad* de una solución $S \epsilon F$ se puede definir como cualquier subconjunto de F. Formalmente, es un mapeo que asocia a cada solución factible $S \epsilon F$ con un subconjunto $N(S)$=$\{S_1, S_2,\ldots,S_p\}$ de posibles soluciones en F. Cada solución $S' \epsilon N(S)$ puede ser alcanzada desde S a través de un operador llamado movimiento (m), que perturba la solución S[52].

- *Definición:* En términos de una vecindad, una solución $s \epsilon S$ es un *óptimo local* si tiene mejor calidad que el resto de sus vecinos, es decir, $f(s) \leq f(s') \; \forall \; s \epsilon N(s)$.

- *Definición:* Una *instancia* de un problema de optimización combinatoria es un caso particular de este. Está caracterizada por el conjunto finito E, un conjunto de soluciones factibles F y una función de costo $c : F \rightarrow R$ que asocia a un valor real $c(S)$ a cada solución factible $S \epsilon F$.

4.1. Búsqueda local

A partir de este momento se presentan los métodos con los que se obtuvieron las soluciones aproximadas al problema de empaquetamiento de esferas. La búsqueda local es la base de muchos de los métodos usados en problemas de optimización.

Antecedentes

Su primera utilización se remonta a finales de 1950 e inicios de 1960, cuando se introdujo por primera vez el "Problema del agente viajero", que es un problema de optimización cuyo objetivo es encontrar un recorrido completo que conecte todos los nodos de una red, visitándolos tan solo una vez y volviendo al punto de partida, y que además minimice la distancia total de la ruta, o el tiempo total del recorrido. [2]
En los años siguientes, el uso de la búsqueda local se incrementó debido a que el planteamiento de problemas de optimización creció en gran medida, destacando los de rutas vehiculares y los problemas de corte. Pese a esto, su éxito duró poco pues se empezó a considerar que el método no era muy eficaz ya que únicamente se veía la utilidad práctica de este, sin nunca avanzar conceptualmente. [2]
A inicios de los años 2000, este método se retomó y empezó a recibir diversas modificaciones, sirviendo como base de algunas de las metaheurísticas más importantes tales como GRASP y Algoritmos Genéticos, de las cuales hablaré más adelante.

Algoritmo

La búsqueda local empieza con una solución inicial factible x y busca en una vecindad de x (en el conjunto de vecindades$N(x)$) una solución x' tal que $f(x') < f(x)$. En caso de que x' cumpla esta condición, ésta es renombrada como x y comenzamos la búsqueda de una mejor solución alrededor (en la vecindad) de la nueva solución x.
El algoritmo se detiene cuando todas las soluciones candidatas son peores que la solución actual, significando que el óptimo local ha sido alcanzado[52].

El pseudocódigo para este proceso es el siguiente:

Require: x^0, $N(.)$, $f(.)$;
  $x \leftarrow x^0$;
  while x no es localmente óptimo respecto a su vecindad *N(x)* **do**
    Sea $y \in$ *N(x)* tal que *f(y)* <*f(x)*
    $x \leftarrow y$
  end while
    return x

Figura 4.1: Pseudocódigo Búsqueda Local

En la imagen anterior, $f(.)$ representa la función objetivo, *N(x)* es una vecindad de x y x^0 es una solución inicial.

Este proceso tiene la ventaja de encontrar soluciones muy rápidamente, pero queda atrapado fácilmente en mínimos locales y su solución final depende fuertemente de la solución inicial. Además, es un método determinístico y sin memoria: dada una misma entrada, regresa la misma salida.

Ejemplo

A continuación se presenta un ejemplo obtenido de [52] para el mejor entendimiento de la Búsqueda local.

Se busca maximizar la función *f(x)* = x^3 - $60x^2$ + 900 x. Las soluciones se representan con 5 elementos en forma binaria y las vecindades se crean cambiando un bit. Supongamos que $x = 10001$, es decir, $x = 17$ y *f(x)* = 2774.

Calculando *N(x)*:

Solución vecina	*f(x)*
00001	742
11001	525
10101	1602
10011	2200
10000	3027

Tabla 4.1: Soluciones vecinas a 10001

Hacemos entonces $x \leftarrow x' = 10000$, $x' \epsilon N(x)$, de forma que $x = 16$ y $f(x) = 3027$. Repetimos el proceso.

Solución vecina	*f(x)*
00000	0
11000	765
10100	1901
10010	2493
10001	2774

Tabla 4.2: Soluciones vecinas a 10000

Como no hay mejora a la solución actual, el proceso termina, siendo $x = 10000$ un óptimo local.
El óptimo global de esta función es $f(01010) = f(10) = 4000$. No podemos llegar a éste a partir de la búsqueda local con la solución inicial dada, pues este procedimiento no permite tomar peores soluciones a la actual, causando una reducción del espacio de búsqueda.

Implementación del problema de las n esferas

Para el problema que se trata en este trabajo, teniendo una solución inicial con pequeñas esferas que cumplen que están dentro del contenedor y que no se traslapan, se utiliza una función que crea tres distintos tipos de vecinos: busca en la solución inicial cuál es la esfera que está más alejada con respecto al eje x, se generan tres números aleatorios menores a 0.5 y se sustituyen las coordenadas anteriores por estas nuevas, siempre y cuando no haya traslapes entre esferas. Análogamente, lo hago para la esfera cuyo centro esté más alejado del origen con respecto al eje y y aquella cuya diferencia con respecto al centro sea mayor, en el eje z. Una vez creadas las vecindades, se realiza todo el proceso de búsqueda local.

La implementación se hizo en RStudio para *n=10, 20, 30, 40, 50, 60, 70, 80, 90 y 100*, utilizando 5000 iteraciones, pues, para n=30, al utilizar 500 iteraciones, los resultados obtenidos fueron de 0.5912013 unidades en 10.05 segundos. Aumentando a 1000 iteraciones se obtuvo un radio de 0.5729185 unidades en 24.47 segundos y, al aumentar a 5000 iteraciones, se obtuvo un radio de 0.5595677 unidades en 66.62 segundos. Para n = 50 ocurrió algo similar, pues al utilizar 500 repeticiones se obtuvo un radio de 0.8659507

en 12.77 segundos; con 1000 iteraciones fue de 0.7282326 en 33.71 segundos y con 5000 iteraciones fue de 0.6584125 en 85.36 segundos.
Además de esto, fijé una semilla en 11, con el fin de que, a pesar de utilizar aleatorizaciones, los resultados puedan ser reproducidos y comparados con mayor precisión utilizando heurísticas diferentes. Si bien pude haber utilizado cualquier semilla, hice uso de esta pues fue la que mejores resultados me daba en casi todas las heurísticas. En Búsqueda local, para n=10, obtuve lo siguiente:

- Semilla 11: R(s)= 0.3598139
- Semilla 7: R(s)=0.3861871
- Semilla 15: R(s)= 0.4725155
- Semilla 46: R(s)=0.3986968
- Semilla 21 : R(s)= 0.3598799

Notamos que la semilla 11 fue la que produjo el mejor resultado.

Una función que se repite en todos los métodos, es la creación del radio del contenedor R, pues lo que hice, para asegurar que se cumpla la condición del problema de que todas las esferas deben estar dentro del contenedor, fue calcular las distancias del centro de la esfera (que se encuentra en el origen) a los centros de cada una de las otras esferas (que se eligieron aleatoriamente), tomar la distancia máxima y sumarle el radio de su esfera correspondiente, de forma que el radio del contenedor es mínimo y contiene a todas las esferas.

Para asegurar que la condición $d(0,i) \leq (R_0 \;\; r_i)$, $\forall \; i \; \epsilon \; N$ se cumple, hice uso de una matriz, la cual vemos a continuación:

Recordemos primero que una matriz :

$$A_{m,n} = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{pmatrix}$$

es cuadrada si tiene la misma cantidad de renglones y de columnas (nxn). Además, llamamos *matriz diagonal* a aquella matriz cuadrada cuyos elementos a_{ij}, con $i \neq j$, son iguales a cero[29], es decir:

$$\mathrm{A}_{m,n} = \begin{bmatrix} a_{11} & 0 & 0 & ... & 0 \\ 0 & a_{22} & 0 & ... & 0 \\ 0 & 0 & a_{33} & ... & 0 \\ 0 & 0 & 0 & ... & 0 \\ 0 & 0 & 0 & ... & a_{nn} \end{bmatrix}$$

Dicho lo anterior, para comprobar que la nueva solución encontrada mediante la búsqueda local cumple con las condiciones de no traslapamiento, creé una función que calcula las distancias entre dos esferas, calculando las distancias de centro a centro, y guardando el valor de traslape en una matriz, donde las entradas $[i, i]$ tienen valor de 0.2 ($2r$) pues es el "traslape" de una esfera consigo misma, pero las $[i, j]$, tales que $i \neq j$, deben tener un valor igual a cero si no hay traslape; así, la matriz debe ser diagonal para asegurar que no existen superposiciones.
Para valores de n mayores, sería muy poco práctico imprimir esta matriz. Por esto, utilicé una función ya creada de *RStudio* que devuelve el valor TRUE si la matriz es diagonal y FALSE si no lo es.

A continuación presento los resultados para n= 10 y n=100, a reserva de que el resto se muestran en un apartado posterior del documento, donde se hace un análisis con las demás heurísticas implementadas.

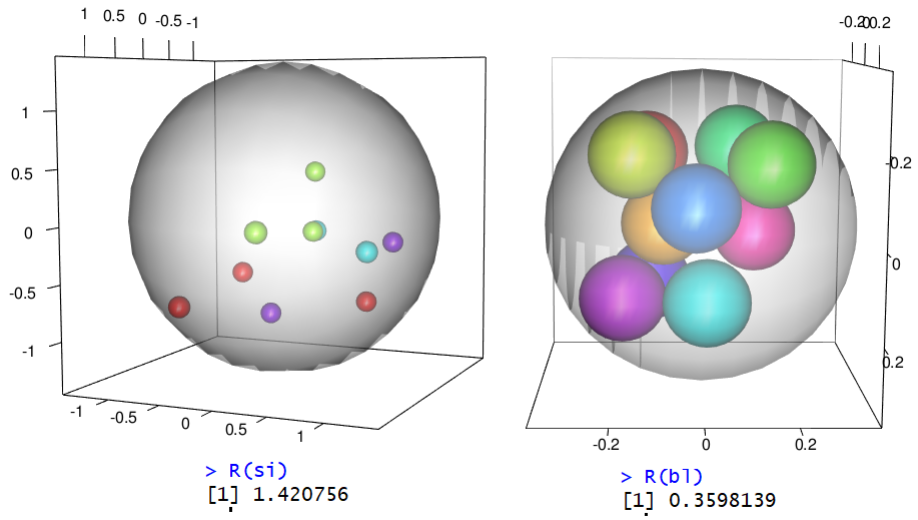


Figura 4.2: Búsqueda local: *n=10*

En la Figura 4.2 tenemos a la solución inicial en el lado izquierdo, con un radio de R=1.420756 unidades y a la solución con Búsqueda Local en el lado derecho, con R=0.3598139. Hubo una mejora con respecto a la solución inicial, disminuyendo 1.0609421 unidades.

Para n=100:

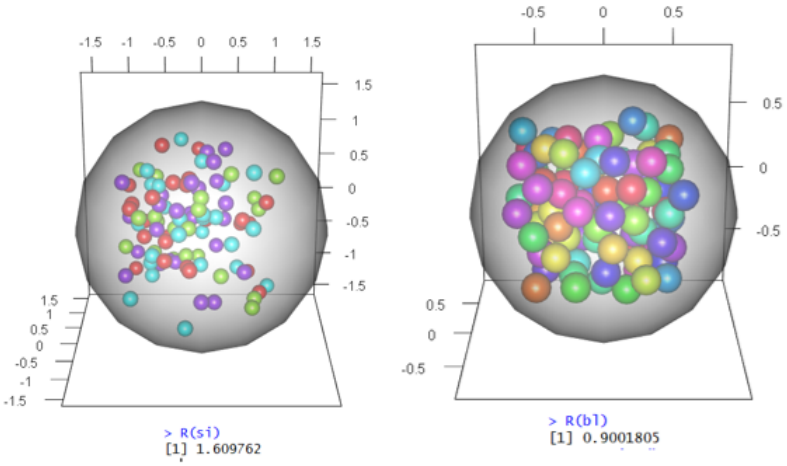


Figura 4.3: Búsqueda local: *n=100*

En la Figura 4.3, la solución inicial (gráfica de la izquierda) produjo un radio de

R=1.609762, mientras que el radio final del contenedor (imagen derecha) $R = 0.9001805$. En este caso hubo una disminución de 0.7095815 unidades.

4.2. GRASP

Se trabajó también con la heurística GRASP (Greedy randomized adaptive search procedure), la cual se explica a profundidad en esta sección. Posteriormente se muestran algunos resultados de pruebas que se obtuvieron en RStudio.

Antecedentes

El algoritmo GRASP (Procedimiento de Búsqueda Miope Aleatorizado y Adaptativo en español) fue mencionado por primera vez de manera formal en 1989 [15]. Esta heurística se compone de dos fases: la construcción y la búsqueda local.

Su nombre corresponde a las siglas en inglés Greedy Randomized Adaptive Search Procedure, que quiere decir Procedimiento de búsqueda adaptativa miope aleatorizado; decimos que es miope ya que en una iteración escoge la mejor opción que tiene disponible, sin ver que esto puede obligarle a tomar malas decisiones posteriormente; es aleatorizado ya que elige a los candidatos para solución inicial al azar y es adaptativo porque va trabajando con lo que va surgiendo.

Algoritmo GRASP

Este procedimiento está compuesto básicamente por dos métodos: contrucción y búsqueda local. A continuación se explica a detalle cada una de estas etapas.

Construcción:

En esta fase, como su nombre lo indica, se busca crear una solución cuya vecindad es investigada hasta llegar a un mínimo local a partir del cual el algoritmo puede realizar la búsqueda local. Esta solución es muy importante puesto que tiene que ser factible.
Existen diversos mecanismos para llegar a la solución mencionada. Estos procedimientos

mezclan miopía con aleatorización de distintas maneras, pero, en general, se van agregando elementos a la solución de uno en uno [44]. Para elegir estos elementos se hace uso de una función miope o de sensibilidad, que mide la contribución local de cada elemento a una solución parcial. Luego entra la aleatoriedad, en donde elegimos a los candidatos con mejor valor en la función miope, creando una nueva lista llamada lista restringida de candidatos (RCL), los cuales iremos tomando de manera aleatoria para pasar a ser parte de la solución inicial deseada. Dicha lista puede construirse de diversas formas, a continuación expongo algunas de estas:

- Construcción por cardinalidad:

Calcula los valores de función miope para cada elemento, elige los k con el menor valor para formar la lista restringida y luego toma un elemento aleatorio de ésta. La actualiza eliminando el elemento tomado y repite hasta que no haya más candidatos. [44]

Require: k, E, c (.) ;
  $x \leftarrow \emptyset$;
  $C \leftarrow E$;
  Calcular costo miope $c(.)\ \forall\ e \in C$;
  while $C \neq \emptyset$ **do**
    $RCL \leftarrow \{k$ elementos $e \in C$ con el menor $c(e)$ $\}$;
    Seleccionar un elemento s de RCL al azar
    $x \leftarrow x\ U\{s\ \}$;
  Actualizar el conjunto candidato C;
  Calcular el costo miope $c(e)\ \forall\ e \in C$
end while
  return x

Figura 4.4: Construcción por cardinalidad [44]

- Construcción por valor

Aquí, denotamos por c_* y c^* al menor y mayor valor de la función miope, respectivamente, y definimos un α tal que $0 \leq \alpha \leq 1$. En este tipo de listas, la lista restringida de candidatos consiste en todos los elementos *(e)* cuyo valor de función miope *(c(e))* es tal que $c(e) \leq c_* + \alpha(c* - c_*)$. Aquí, si $\alpha = 0$, este esquema de selección es un algoritmo miope, mientras que si $\alpha = 1$, es totalmente aleatorio[44].

Require: $\alpha, E, c(.)$;
$x \leftarrow \emptyset$;
$C \leftarrow E$;
Calcular costo miope $c(.)\ \forall\ e\ \epsilon\ C$;
while $C \neq \emptyset$ **do**
    $c_* \leftarrow min\{\ c(e) | \text{e } \epsilon \text{ C}\}$
    $c^* \leftarrow max\{\ c(e) | \text{e } \epsilon \text{ C}\}$
    RCL $\leftarrow \{\ e\ \epsilon$ C$| c(e) \leq \text{c}_* + \alpha(c^* - \text{c}_*)\ \}$;
    Seleccionar un elemento s de RCL al azar
    $x \leftarrow x\ U\{s\ \}$;
    Actualizar el conjunto candidato C;
    Calcular el costo miope $c(e)\ \forall\ e\ \epsilon\ C$
end while
    return x

Figura 4.5: Construcción por valor [44]

- Construcción aleatoria

Mientras la lista de candidatos sea no vacía, elegimos secuencialmente un conjunto de elementos candidatos al azar, calculamos sus costos miopes y después completamos la solución usando un algoritmo miope, el cual toma el elemento más pequeño de la lista de candidatos actual y lo agrega a la solución. [44]

Require: k, E, c (.) ;
  $x \leftarrow \emptyset$;
  $C \leftarrow E$;
  Calcular costo miope $c(.)\ \forall\ e \in C$;
  for i = 1, 2, ... k **do**
    if C $\neq \emptyset$ **then**
      Elegir el elemento e al azar de C;
      $x \leftarrow x \cup \{ e\}$;
      Actualizar el conjunto de candidatos C;
      Calcular el costo miope $c(e)\ \forall\ e \in C$;
    end if
end for
while $C \neq \emptyset$ **do**
    $e_* \leftarrow argmin\{c(e)|e \in C\}$;
    $x \leftarrow\ x \cup \{ e_*\}$;
    Actualizar el conjunto candidato C;
    Calcular el costo miope $c(e)\ \forall\ e \in C$
end while
    return x

Figura 4.6: Construcción aleatoria [44]

- Construcción con perturbaciones

Se aplican perturbaciones o cambios aleatorios a los datos del problema, se calcula el costo miope de los datos perturbados y se aplica un algoritmo miope similar al de la construcción aleatoria. [44]

Require: E, c (.) ;
  $x \leftarrow \emptyset$;
  $C \leftarrow E$;
  Perturbar aleatoriamente los datos del problema;
  Calcular el costo miope perturbado $c'(e)\ \forall\ e \in C$;
  while $C \neq \emptyset$ **do**
    $e_* \leftarrow argmin\{c'(e)|e \epsilon C\}$;
    $x \leftarrow\ x \cup \{ e_*\}$;
    Actualizar el conjunto candidato C;
    Calcular el costo miope $c(e)\ \forall\ e \in C$
end while
    return x

Figura 4.7: Construcción con perturbaciones [44]

- Construcción sesgos

En este procedimiento, en lugar de seleccionar el elemento de la RCL al azar con las mismas probabilidades para cada elemento, se asignan diferentes probabilidades, favoreciendo a elementos bien evaluados. Los elementos de la RCL se ordenan de acuerdo a los valores de la función miope[44].

La probabilidad $(r(e))$ de seleccionar el elemento e es:

$$\pi(r(e)) = \frac{sesgo(r(e))}{\sum sesgo(r(e'))} \tag{4.1}$$

Donde *r(e)* es la posición del elemento e en la RCL.

Además, hay varias alternativas para asignar sesgos a los elementos:

- Sesgo aleatorio: sesgo(r)=1
- Sesgo lineal: sesgo(r)=$1/r$
- Sesgo exponencial: sesgo(r)=e^{-r}

Require: $\alpha, E, c(.)$;
$x \leftarrow \emptyset$;
$C \leftarrow E$;
Calcular costo miope $c()\ \forall\ e\ \epsilon\ C$;
while $C \neq \emptyset$ **do**
$c_* \leftarrow min\{c(e)|e\epsilon C\}$
$c^* \leftarrow max\{c(e)|e\epsilon C\}$
RCL $\leftarrow \{e\ \epsilon$ C$|c(e) \leq$c$_* + \alpha(c^* -$c$_*)\ \}$;
Asignar un rango $e(e)$, $\forall\ e\ \epsilon RCL$;
Asignar una probabilidad $\pi(r(e))$ de elegir el elemento e en RCL que favorezca a los candidatos con mejores rangos;
Elegir el elemento s de RCL al azar con probabilidad $\pi(r(e))$;
$x \leftarrow x\ \cup \{s\ \}$;
Actualizar el conjunto candidato C;
Calcular el costo miope $c(e)\ \forall\ e\ \epsilon\ C$
end while
return x

Figura 4.8: Construcción sesgos [44]

El elemento que se incorpora en la construcción de la solución, se selecciona aleatoriamente dentro de la RCL. Una vez incorporado se actualiza la lista de candidados y se re-evaluan los costos incrementales.

Búsqueda Local

Una vez terminada la fase de construcción, se realiza una búsqueda local como la ya explicada anteriormente, que se puede ver como un proceso iterativo que empieza en una solución factible, y visita otra solución, ya sea factible o no factible, hasta que la solución factible ya no pueda ser mejorada.

Así, el pseudocódigo completo de GRASP queda de la siguiente manera:

```
Require: i_max
  f* ←∞ ;
  for i ≤ i_max do
      x ← Construccion();
    x ← BusquedaLocal(x);
    if f(x) < f* then
        f* ← f(x);
        x* ← x
    end if
end for
     return x
```

Figura 4.9: Pseudocódigo GRASP. Obtenido de [44]

Este algoritmo tiene la ventaja de comenzar con una solución inicial construida para ser buena, y no aleatoria. Sin embargo, una vez construida la solución, solo aplica búsqueda local, por lo que carece de memoria.

Ejemplo

A continuación se presenta un pequeño ejemplo de aplicación del algoritmo.
El problema a tratar es una instancia del muy famoso *Problema del viajero*, en el que, al tener una lista de n ciudades y las distancias entre cada par de ellas, se busca encontrar la ruta más corta posible que visita cada ciudad exactamente una vez y al finalizar regresa a la ciudad de origen. [3]
Supongamos que tenemos 6 ciudades, cuyas distancias se presentan en la Tabla 4.3:

	1	2	3	4	5	6
1	-	23	54	17	132	41
2	23	-	12	100	59	48
3	54	12	-	45	28	31
4	17	100	45	-	22	142
5	132	59	28	22	-	39
6	41	48	31	142	39	-

Tabla 4.3: Distancias entre las 6 ciudades

Consideramos la ciudad inicial como punto de inicio 1 y la Lista Restringida de Candidatos se tomó de tamaño 3. Veamos cómo se construye la solución:

Las distancia de la ciudad 1 a las demás, denotadas como $d(c_1, c_i)$, $i = 2, 3, ..., 6$ están dadas por el siguiente vector: (0, 23, 54, 17, 132, 41). La RCL de 3 elementos se forma eligiendo a aquellos cuyo costo miope sea menor, i.e. cuya distancia con la ciudad inicial sea más pequeña. De esta forma, RCL=$\{4,2,6\}$. De manera aleatoria se extrae un elemento de esa lista para formar parte de la solución. Este procedimiento se repite eliminando a los elementos ya utilizados.

	Distancias					
	1	2	3	4	5	6
1	-	23	54	17	132	41
6	-	48	31	142	39	-
3	-	12	-	45	28	-
5	-	59	-	22	-	-
2	-	-	-	100	-	-

RCL	Aleatorio	Solución
{4,2,6}	6	(1,6,-,-,-,-)
{3,5,2}	3	(1,6,3,-,-,-)
{2,5,4}	5	(1,6,3,5,-,-)
{4,2}	2	(1,6,3,5,2,-)
{4}	4	(1,6,3,5,2,4)

Tabla 4.4: Ejemplo GRASP: Construcción por cardinalidad

Una vez construida la solución inicial, viene el proceso de mejora de la solución, en el que se aplica búsqueda local como ya se explicó anteriormente.

Implementación del problema de las n esferas

Implementé este código utilizando las mismas vecindades que en búsqueda local, es decir, encontrando las esferas más alejadas en los ejes x, y y z, respectivamente, y construyendo la solución inicial mediante cardinalidad de la siguiente manera: se crea una solución inicial

aleatoria con n círculos. Se calculan los costos miopes de esta solución: para cada esfera se calcula la distancia del origen al centro de ésta y se suma su radio (0.1), de manera que la esfera más cercana al centro tiene el menor costo miope. Posteriormente se ordenan estos costos miopes de menor a mayor y se genera un número aleatorio k. El valor del costo miope del k-ésimo elemento será utilizado como cota para formar la Lista Restringida de Candidatos, es decir, todos los elementos cuyo costo miope sea menor o igual a la cota ingresan a ésta. A continuación, elegimos un elemento de esta lista aleatoriamente para que forme parte de la solución construida. Repetimos este proceso p veces, generando una solución aleatoria distinta cada vez, hasta tener una nueva solución que será utilizada posteriormente con búsqueda local.

Los parámetros utilizados fueron k=4 y un total de 1000 iteraciones en la Búsqueda Local, debido a que, para n=10, se inicia con 20 iteraciones y la misma k obteniendo un radio de 0.4237438 unidades en 4.32 segundos; al aumentar a 100 iteraciones, el resultado fue de 0.3887875 unidades en 6.61 segundos, con 500 iteraciones R(s)=0.3601815 en 8.16 segundos y finalmente, al aplicar 1000 iteraciones, se obtuvo un radio de 0.3329987 en 11.73 segundos. Además, se fijó una semilla en 11, pues buscaba comparar los resultados con Búsqueda Local cuya semilla ya estaba fija. De igual forma, probé distintas semillas para ver si obtenía mejores soluciones con alguna otra y los resultados para *n=10* fueron los siguientes:

- Semilla 11: R(s)= 0.3329987
- Semilla 21: R(s)= 0.3816265
- Semilla 7: R(s)= 0.4149418
- Semilla 15: R(s)= 0.4418693
- Semilla 46: R(s)= 0.4015101

Apliqué este método para *n= 10, 20, 30, 40, 50, 60, 70, 80, 90 y 100*. A continuación presento los resultados para n=50 y n=80, pues el resto será expuesto en el apartado de resultados.

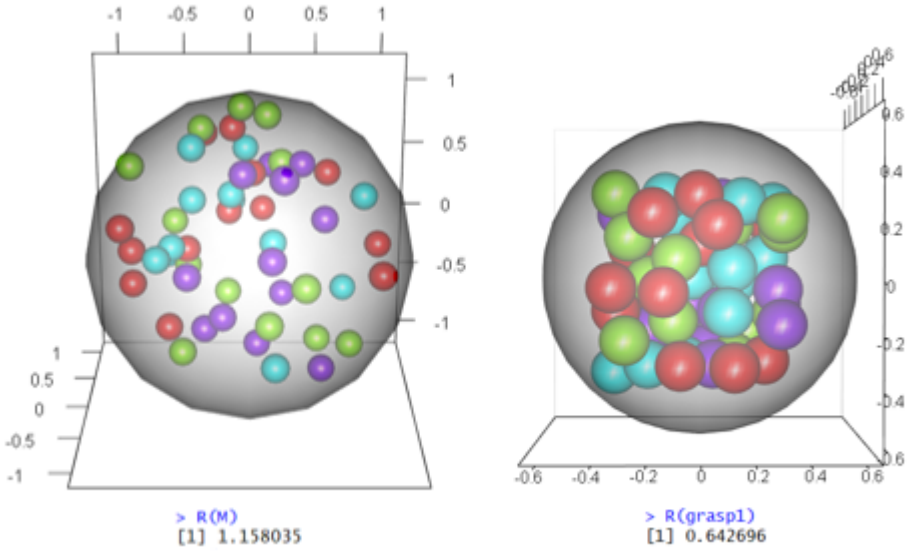


Figura 4.10: GRASP, n=50

En la Figura 4.10 podemos observar la solución inicial en la gráfica de la izquierda y la solución final usando el método GRASP en la gráfica de la derecha. Observemos que el radio inicial producido fue R= 1.158035, mientras que el radio final del contenedor fue de R = 0.642696. Con esto vemos que hubo un cambio de 0.515339 unidades, lo cual representa una mejora.

Ahora veremos el caso en que n=80.

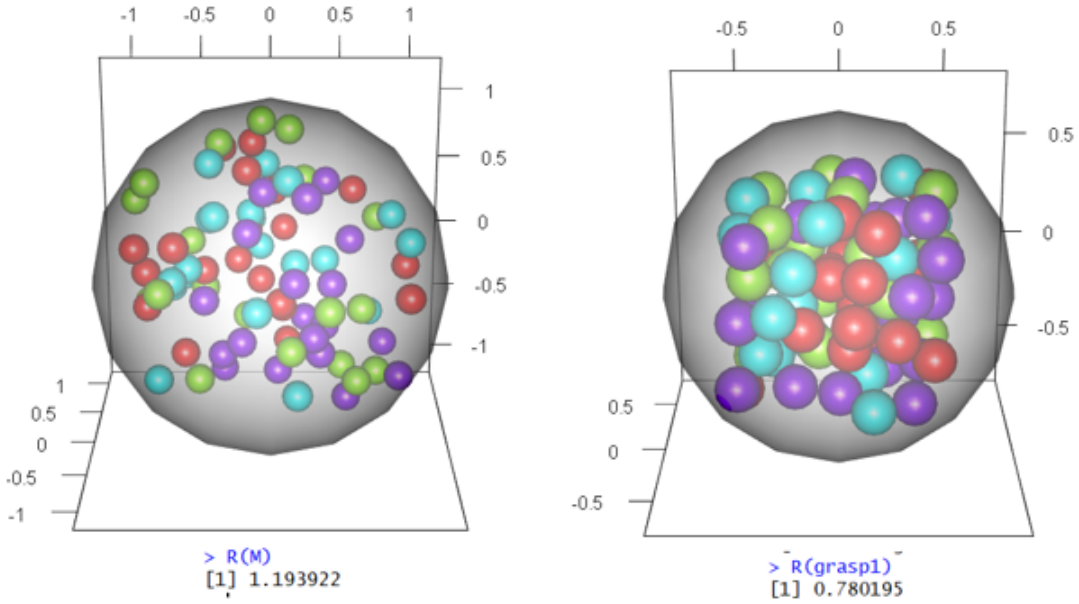


Figura 4.11: GRASP, n=80

En la Figura 4.11 vemos que la solución inicial (gráfica de la izquierda) produjo un radio de R= 1.193922, mientras que el radio final del contenedor (imagen derecha) fue de R = 0.780195. En este caso, el radio disminuyó 0.413727 unidades con respecto a su solución inicial, mejorando la solución.

4.3. Recocido simulado

En esta parte del documento se trabaja con la metaheurística *Recocido Simulado*. Al final se presentan los resultados obtenidos de la implementación en RStudio.

Antecedentes

En 1983 se da la primera descripción de este método y la podemos encontrar en [27]. Está inspirado en el proceso físico de recocido de acero, que consiste en calentar y luego enfriar muy lentamente el material hasta obtener una estructura cristalina sólida, convergiendo a un estado de equilibrio[52]; es por esto por lo que existen ciertas analogías

entre el sistema físico y un problema de optimización. (Tabla 4.4)

Sistema físico	Problema de optimización
Estado del sistema	Solución
Posiciones moleculares	Variables de decisión
Energía	Función objetivo
Estado fundamental	Solución óptima global
Estado metaestable	Óptimo local
Enfriamiento rápido	Búsqueda local
Temperatura	Parámetro de control T
Recocido de acero	Recocido simulado

Tabla 4.5: Analogías entre el Recocido de acero y el Recocido simulado. Traducida de [52]

Algoritmo

El Recocido Simulado (Simulated Annealing en inglés) a diferencia de la búsqueda local, permite de alguna manera tomar "peores soluciones" a la actual para no quedarse atrapado en un óptimo local y explorar mejor el espacio de soluciones.

En la imagen que se encuentra a continuación, hay tres puntos: solución inicial, mínimo local y nueva solución. Este método acepta la nueva solución a pesar de que su valor en la función objetivo sea mayor, debido a que esta nos permite acercarnos más al mínimo deseado.

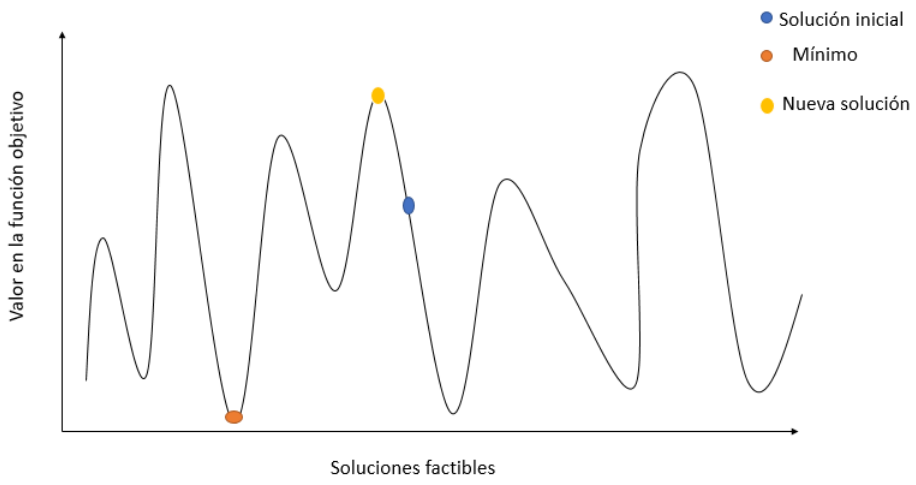


Figura 4.12: Recocido Simulado

Como en los algoritmos de búsqueda local, partimos de una solución inicial x_0 generalmente aleatoria. Además, empezamos con una temperatura inicial t_0, que se recomienda que sea alta [14], y definimos un criterio de paro, que puede ser la temperatura deseada, número de iteraciones, etcétera.
Sea x_0 una solución inicial. Tomamos una solución x que esté en la vecindad de x_0, denotada por *N(x_0)* y evaluamos ambas en la función objetivo. Si *f(x)* $>f(x_0)$, sustituimos el valor de x_0 por x, pero si *f(x)* $<f(x_0)$, definimos una *delta* tal que :

$$\delta = f(x_0) - f(x)$$

Posteriormente hay una parte aleatoria pues tomamos un número u uniforme entre cero y uno tal que, si u es menor a la exponencial de *delta* sobre t, x_0 toma el valor de x. En otro caso, buscamos otra solución vecina. [14]

En este punto hay que notar que, cuando tenemos temperaturas muy altas, es decir, valores de t muy grandes, $e^{-\delta/t}$ tiende a 1, por lo que es casi seguro que las soluciones vecinas sean muy aceptadas al principio, pero mientras más avance el algoritmo, la temperatura disminuye y se vuelve cada vez menos probable aceptar nuevas soluciones.
Repetimos todo lo mencionado anteriormente hasta que se cumple el criterio de paro.

De acuerdo a lo descrito anteriorente, en la figura 4.13 presentamos el pseudocódigo de esta metaheurística, para un problema de maximización.

```
Require: x_0, N(.), nrep, t_0
  t ←t_0 ;
  while t <t_final do
      for n ≤ nrep  do
          Crear una solución x ϵN(x_0)
          if f(x_0) < f(x) then
       x_0 ←x
       else
       δ = f(x_0) − f(x)
       u ϵ U(0,1)
          if u < e^{−δ/t} then
       x_0 ← x
          end if

      end for
       t = α(t)
  end while
       return x
```

Figura 4.13: Pseudocódigo Recocido Simulado para problema de maximización

La mayor ventaja de este método es que impide que quedemos atrapados en mínimos locales, al aceptar soluciones que empeoran la actual; esto mismo permite que el espacio de soluciones se explore en mayor medida. La mayor desventaja que presenta es que puede llegar a ser muy lento, pues, tal como en el método físico en el que se basa, las disminuciones del parámetro T son muy lentas.

Ejemplo

A continuación se presenta un pequeño ejemplo tomado de [52], para la implementación del Recocido Simulado.

Se busca maximizar *f(x)* = x^3 - $60x^2$ + 900 x + 100, con $x \epsilon R$. En este ejemplo se define a una vecindad como el cambio de un bit, por lo que es necesario cambiar el espacio de soluciones, de forma que, en lugar de trabajar con $x \epsilon R$, trabajaremos con un vector binario.

La solución inicial está representada por 10011, donde $x = 19$ y *f(x)* = 2399. Además, T_0= 500.

Aleatoriamente se elige cambiar el bit que se encuentra en la posición 1, obteniendo la

solución 00011, donde x=3 y $f(x)$=2287. Se calcula δ = 2399 - 2287 = 112, que nos permite sustituir la solución inicial. Se calcula la disminución de temperatura mediante la función $T_k = \frac{\Gamma}{log_k}$, donde Γ es una constante, y se repite el procedimiento con la nueva solución. A continuación se presenta una tabla con el resto de las iteraciones:

T	Bit a cambiar	Solución	$f(x)$	δ	¿$x \leftarrow x_0$?	Nueva solución vecina
500	1	00011	2287	112	Sí	00011
450	3	00111	3803	<0	Sí	00111
405	5	00110	3556	247	Sí	00110
364.5	2	01110	3684	<0	Sí	01110
328	4	01100	3998	<0	Sí	01100
295.2	3	01000	3972	16	Sí	01000
265.7	4	01010	4100	<0	Sí	01010
239.1	5	01011	4071	29	Sí	01011
215.2	1	11011	343	3728	No	01011

La mejor solución fue 01010, donde x = 10 y $f(x)$ = 4100.

Implementación del problema de las n esferas

Como en todas las heurísticas, la definición y experimentación de parámetros juegan un papel muy importante en el resultado obtenido. Para el Recocido Simulado, utilicé una temperatura inicial de 1000 y una final de 10^{-4} pues estos parámetros deben estar muy alejados entre sí para dar buenos resultados.
Además, para el parámetro α se han hecho diversos experimentos, encontrándose que el mejor valor recomendado es entre 0.95 y 0.99 [1].

Para la implementación de este algoritmo en R, la vecindad utilizada consiste en encontrar las esferas más alejadas respecto al centro del contenedor en los ejes x, y y z, respectivamente, y sustituyendo sus coordenadas por números aleatorios menores a 0.5. Para verificar que no existen traslapes entre esferas, se utiliza una matriz cuyas entradas representan el traslape entre una esfera y otra, de manera que, para que la solución sea factible, ésta matriz debe tener ceros en todas las entradas, salvo en la diagonal (se considera 0.2 para expresar el traslape de una esfera consigo misma). Además, los parámetros que mostraron un mejor comportamiento para el problema de empaquetamiento

fueron $\alpha = 0.99$ y 40 repeticiones. Para verificar esto, se definió un contador para ver en cuántas ocasiones el algoritmo entra en el *ciclo* de la temperatura, econtrando que se hacía 1604 veces dando un número de repeticiones = 20, para $n = 10$, obteniendo un radio de 0.3522297 en 264.9 segundos. Al aumentar al doble las repeticiones, es decir, a 40, el radio del contenedor disminuyó a 0.3321368 tomando un tiempo de 389 segundos. De esta manera, al haber 40 repeticiones en cada una de las 1604 ya mencionadas, el algoritmo realiza 64,160 repeticiones en total.

En adición, es necesario mencionar que se fijó una semilla en 11, al igual que en los algoritmos anteriores, para poder comparar los resultados de mejor manera. De cualquier forma, experimenté con otras semillas, cuyos resultados, para n=10, se presentan a continuación:

- Semilla 11: R(s)= 0.3321368
- Semilla 21: R(s)= 0.3455382
- Semilla 7: R(s)= 0.3496352
- Semilla 15: R(s)= 0.3454823
- Semilla 46: R(s)=0.3329084

Presentaré los resultados para n=10 y n=50. El resto se expondrán en la sección de Resultados.

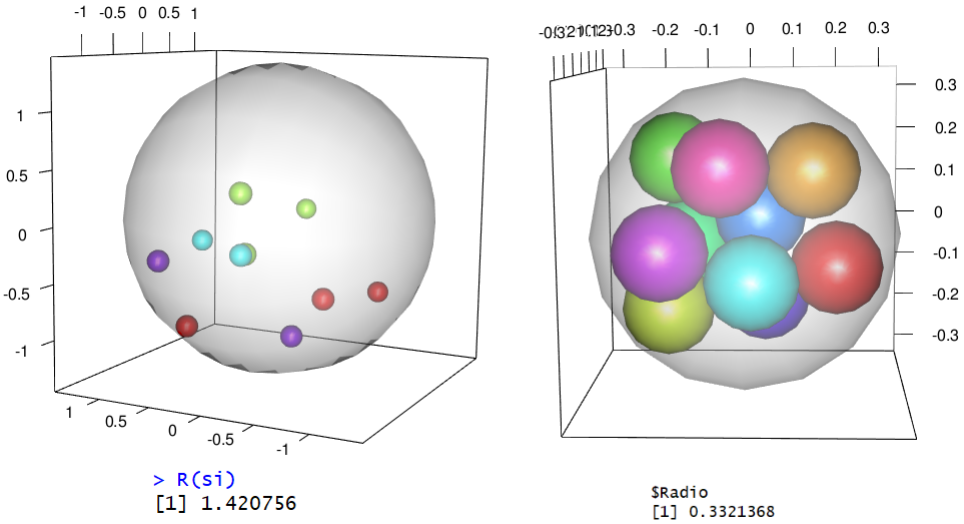


Figura 4.14: Recocido simulado, n=10

El radio pasó de 1.420756 unidades a 0.3321368, habiendo una disminución de 1.0886192.

Para n=50

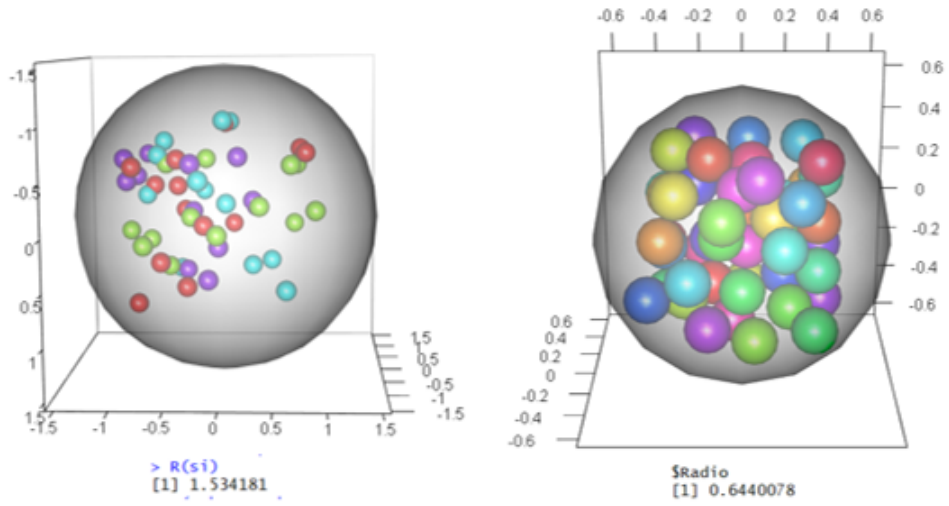


Figura 4.15: Recocido simulado, n=50

Hubo un cambio de 0.8901732 unidades respecto a la solución inicial.

4.4. Búsqueda Tabú

En este apartado se presenta un método que permite hacer movimientos de no mejora, aumentando las posibilidades de llegar a una mejor solución que la actual, alejándose del valor óptimo actual pero explorando en gran medida el espacio de búsqueda.

Antecedentes

La *búsqueda tabú* tiene como base a la *búsqueda local*; sin embargo, una de las características más importantes de esta metaheurística es el uso de la memoria, que se destaca sobre la de otros algoritmos que utilizan memorias basadas en herencia.
Los principios básicos de la *búsqueda tabú* fueron elaborados en una serie de artículos a finales de los años 80 y 90, unificados en [20], escrito por quien se conoce como el fundador de este método: Fred W. Glover.

Algoritmo

Si bien este algoritmo se basa en la idea de buscar mejores soluciones en vecindades, presenta características que lo hacen particularmente mejor que la búsqueda local:

- *Movimientos tabú* son restricciones que se usan para prevenir el ciclado cuando nos alejamos de óptimos locales hacia movimientos de no mejora. Estos movimientos se declaran "tabú" pues se prohíbe que se realicen durante cierto número de iteraciones. [21]

- *Lista tabú* contiene a todos los elementos actualmente prohibidos. Ésta funciona como una *memoria a corto plazo*, pues contiene las soluciones que fueron visitadas en el pasado reciente.

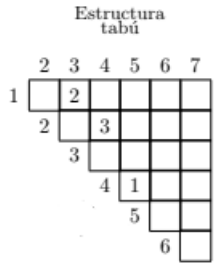


Figura 4.16: Glover, F. y Melián, B. (2003), p. 4

En la Figura 4.16 vemos un ejemplo de lista tabú en el que, para ese problema, el movimiento *1,3* está prohibido por dos iteraciones, el *2,4* por tres iteraciones y el *4,5* por una.

- *Periodo de tenacidad:* es el número de iteraciones por las que la *lista tabú* almacena cada elemento. [35]

- *Memoria a largo plazo:* almacena la mejor solución obtenida hasta el momento (x^*), además de almacenar la cantidad de veces que el movimiento ha sido utilizado, penalizando su uso y obligando al algoritmo a explorar nuevas soluciones.

- *Criterio de aspiración:* éste hace uso de la *memoria a largo plazo*, pues compara cada nueva solución con x^*, permitiendo que, en caso de que un movimiento m esté dentro de la *lista tabú* pero al aplicarlo se obtenga mejor solución que x^*, se aplique m. [21]

En general, podemos decir que la *búsqueda tabú* se desarrolla de la siguiente manera: Se crea una solución inicial x_0 y se calculan sus vecinos. Se toma al mejor de estos vecinos (x') y se verifica si este pertenece a la lista tabú. En caso afirmativo, se elimina a este elemento de la lista de vecinos candidatos y se toma al mejor de esta nueva lista. En el caso negativo, se cambia el valor de *(x_0)* por *x'* y se actualiza la lista tabú. El procedimiento se repite hasta cumplirse el criterio de paro. En la Figura 4.17 se presenta un diagrama de flujo de la Búsqueda Tabú.

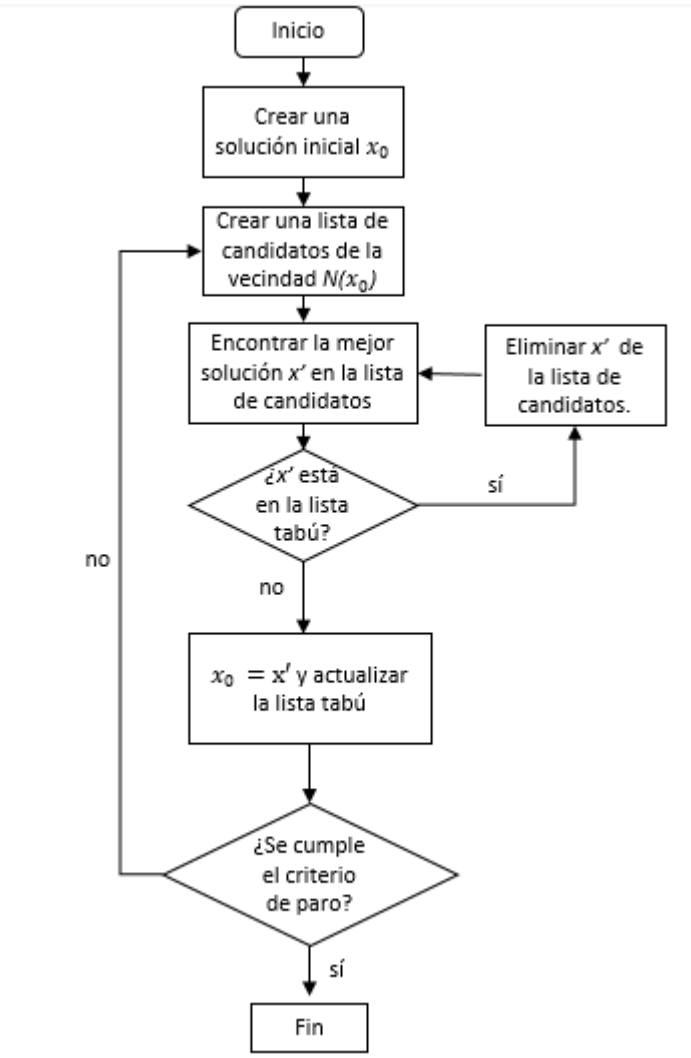


Figura 4.17: Algoritmo búsqueda tabú. Traducido de [23]

El criterio de paro se elige de acuerdo al problema a tratar. Puede ser un número de iteraciones fijo, el tiempo que se tarda la computadora en resolver el problema, un número de iteraciones en que ya no se han visto mejoras, entre otras cosas.

Este método tiene la ventaja de utilizar la memoria a largo plazo para evitar que un movimiento se repita muchas veces, evitando caer en mínimos locales y la memoria a corto plazo, que permite tomar distintas soluciones en cada iteración, ampliando el espacio de búsqueda. Este método llega a ser computacionalmente muy costoso pues debe calcular las soluciones vecinas, así como su aportación a la función objetivo y actualizar la lista

tabú en cada iteración.

Ejemplo

A continuación se presenta un ejemplo obtenido de [33] en el que se hace uso de este método.

Se busca optimizar la disposición de ciertos materiales de manera que se maximice el poder aislante de estos. La *función objetivo* está dada por una función cuyo parámetro de entrada es una permutación de los materiales y cuyo parámetro de salida es un valor de aislamiento. Suponemos además que contamos con siete materiales, por lo que las soluciones están dadas por los ordenamientos de éstos.

El sistema de vecinos que se utilizará es el *swap*, que consiste en intercambiar la posición de dos materiales y la *lista tabú* va a restringir por 3 iteraciones el par de módulos que han sido intercambiados.

Empezamos con la solución inicial que se encuentra a continuación:

2	5	7	3	4	6	1

Valor en la función objetivo = 10

La lista tabú inicia vacía y calculamos todas las soluciones vecinas a la solución actual:

Lista tabú

	2	3	4	5	6	7
1						
2						
3						
4						
5						
6						

Soluciones vecinas

Swap	Valor
5,4	6
7,4	4
3,6	2
2,3	0
4,1	-1

Para cada solución vecina tenemos la variación en la función objetivo, es decir, en cuánto aumentará o decrecerá el valor actual de la función al realizar cada *swap*. Notemos

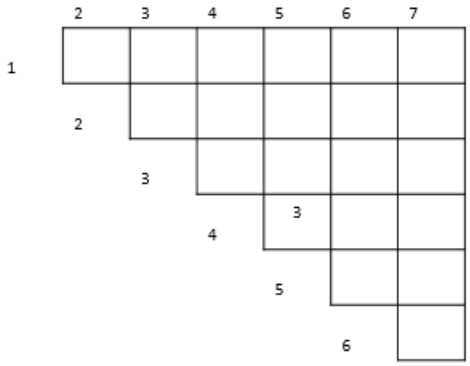
que el mejor cambio que podemos realizar es intercambiar el 4 y el 5 para así aumentar la función en 6 unidades. De esta manera, nuestra nueva solución queda de la siguiente forma:

2	4	7	3	5	6	1

Valor en la función objetivo = 16

Debemos actualizar la lista tabú, restringiendo el movimiento que acabamos de realizar durante 3 iteraciones. Además, calculamos nuevamente los valores vecinos a la nueva solución:

Lista tabú



Soluciones vecinas

Swap	Valor
3,1	2
2,3	1
3,6	-1
7,1	-2
6,1	-4

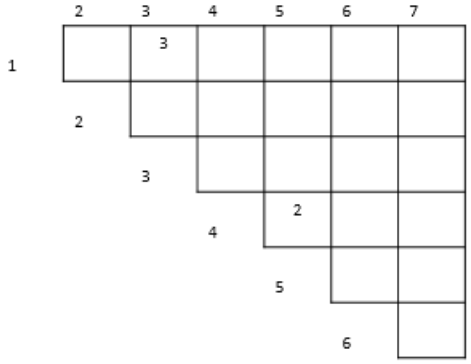
En esta ocasión debemos cambiar el 3 por el 1, para aumentar nuestra función objetivo en 2 unidades.

2	4	7	1	5	6	3

Valor en la función objetivo = 18

Prohibimos este movimiento y actualizamos la lista tabú:

Lista tabú



Soluciones vecinas

Swap	Valor	
1,3	-2	T
2,4	-4	
7,6	-6	
4,5	-7	T
5,3	-9	

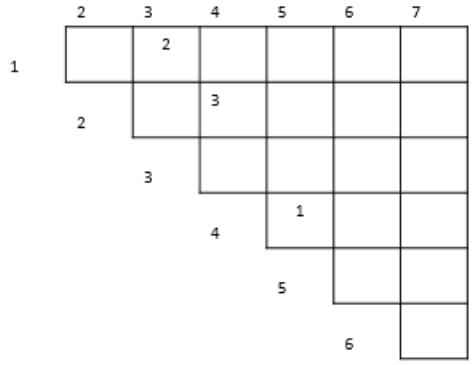
En esta ocasión vemos que el mejor movimiento es *1,3*, sin embargo, este está dentro de la lista tabú, por lo que lo ignoramos, tomando en su lugar la segunda mejor iteración, que sería intercambiar 2 y 4.

4	2	7	1	5	6	3

Valor en la función objetivo = 14

Actualizamos las iteraciones de la lista tabú y calculamos la solución vecina a la actual:

Lista tabú

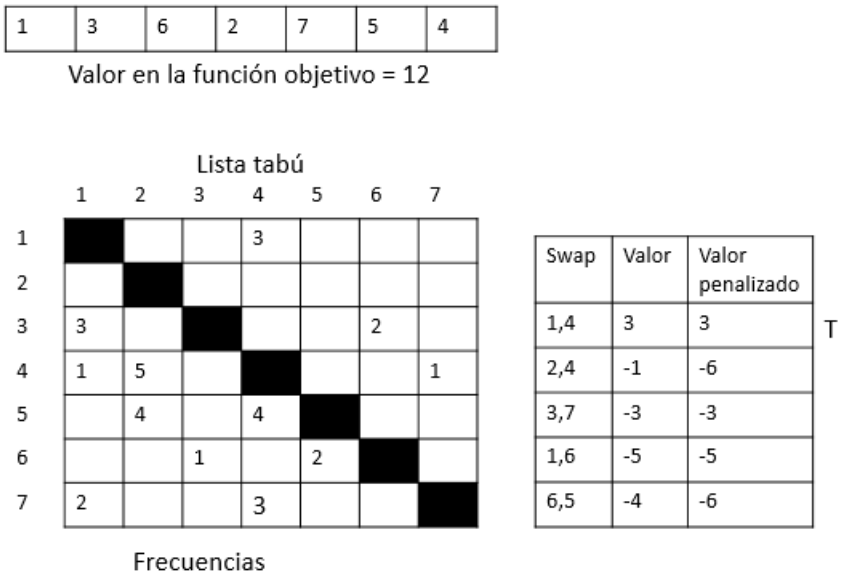


Soluciones vecinas

Swap	Valor	
4,5	6	T
5,3	2	
7,1	0	
1,3	-3	T
2,6	-6	

Vemos aquí que el mejor movimiento es *4,5*, que se encuentra en la lista tabú. Sin embargo, si lo tomáramos, nuestra función objetivo alcanzaría el mejor valor encontrado hasta el momento. Utilizamos el criterio de aspiración y realizamos el intercambio.

Después de realizadas 26 iteraciones, se tienen los siguientes resultados:



Debajo de la tabla de la lista tabú se encuentra una nueva tabla denominada " tabla de frecuencias ", lo que nos indica, después de todas las iteraciones, cuántas veces se ha realizado cada movimiento. Aquí es donde entra la memoria a largo plazo, gracias a la cual podemos penalizar las soluciones que más se han utilizado, con el fin de explorar mejor el espacio de soluciones.

El algoritmo termina cuando se cumple el criterio de paro, devolviendo la mejor solución encontrada a lo largo del proceso.

Implementación del problema de las n esferas

Implementé Búsqueda Tabú en R, poniendo un máximo de iteraciones = 1500 como criterio de paro, pues, para n=10, al utilizar 500 iteraciones, el radio fue de 1.08581 unidades, en un tiempo de 7.14 segundos. Aumenté a 1000 iteraciones, obteniendo un radio de 0.3560491 en 7.16 segundos y finalmente, con 1500 iteraciones, obtuve un radio de 0.3551351 en 7.71 segundos. Al aumentar el número de iteraciones a 20,000, el radio resultó igual.

Además, el criterio para que un elemento ingrese a la lista tabú es el siguiente: Si en el movimiento k moví la esfera s_i, se prohíbe que esta misma se mueva por los siguientes tres

movimientos, es decir, en el $k+1$, $k+2$ y $k+3$. Esto permite que otras esferas tengan la oportunidad de cambiar de lugar. Además se implementó la *memoria a largo plazo* para siempre comparar la mejor solución encontrada, con las que iban surgiendo.

Finalmente, tal como en el resto de las heurísticas, fijé la semilla en 11. A continuación los resultados del uso de las otras semillas, para n=10:

- Semilla 11: R(s)= 0.3551351
- Semilla 21: R(s)= 0.4236207
- Semilla 7: R(s)= 0.3825118
- Semilla 15: R(s)=0.4296165
- Semilla 46: R(s)=0.3660727

Si bien los resultados se expondrán en otro apartado del documento, a continuación presento aquellos para n=60 y n=100:

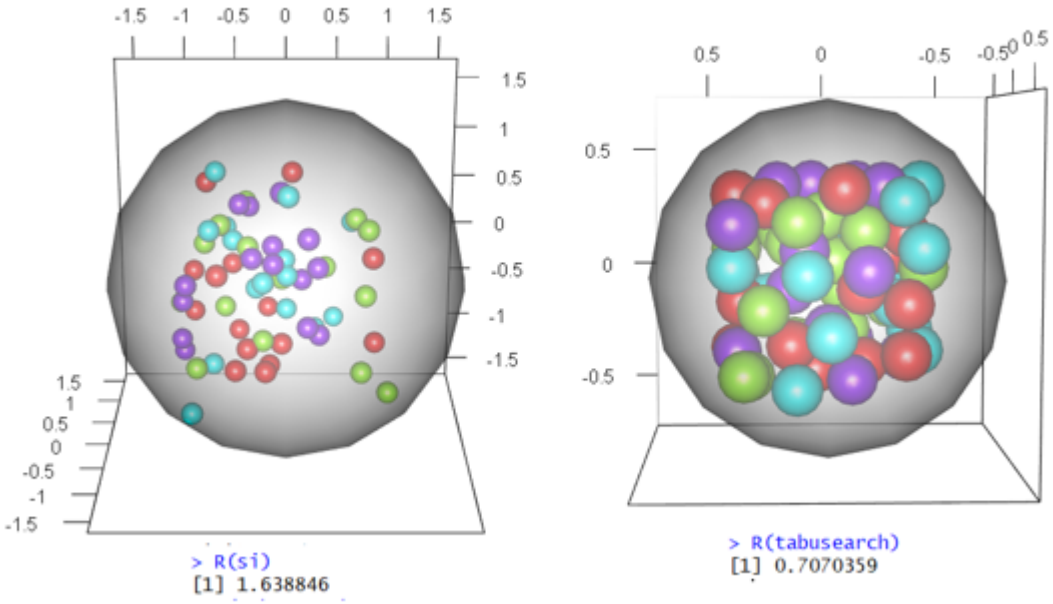


Figura 4.18: Búsqueda Tabú, n=60

Hubo un cambio de 0.8271451 unidades entre ambos radios.

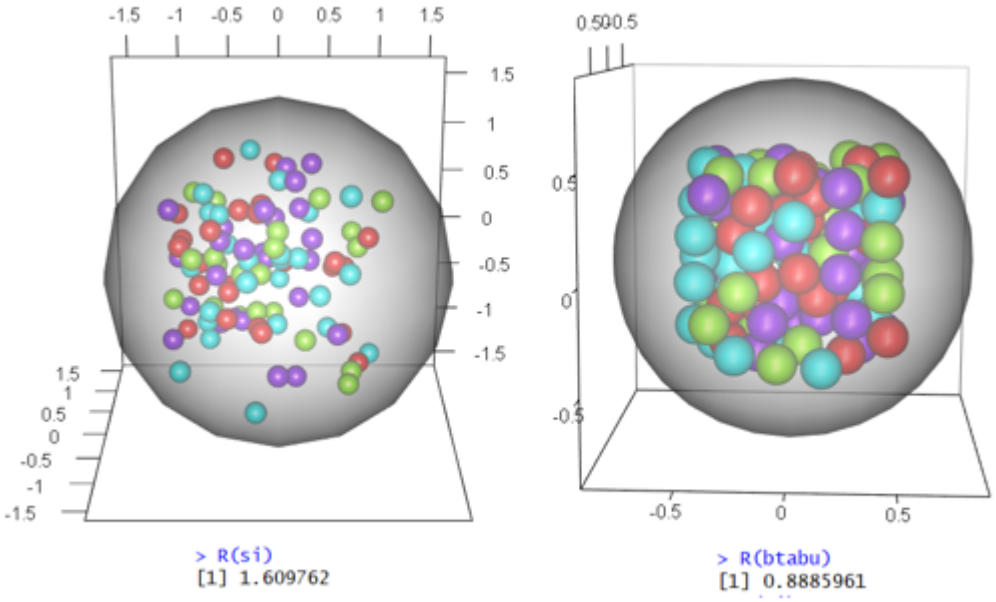


Figura 4.19: Búsqueda Tabú, n=100

El radio del contenedor inició en 1.609762 unidades, terminando en 0.8885961 después de aplicado el algoritmo.

4.5. Algoritmos genéticos

Finalmente, tenemos los *Algoritmos genéticos*, que son una técnica de optimización basada tanto en búsqueda local, como en principios de genética y selección natural.

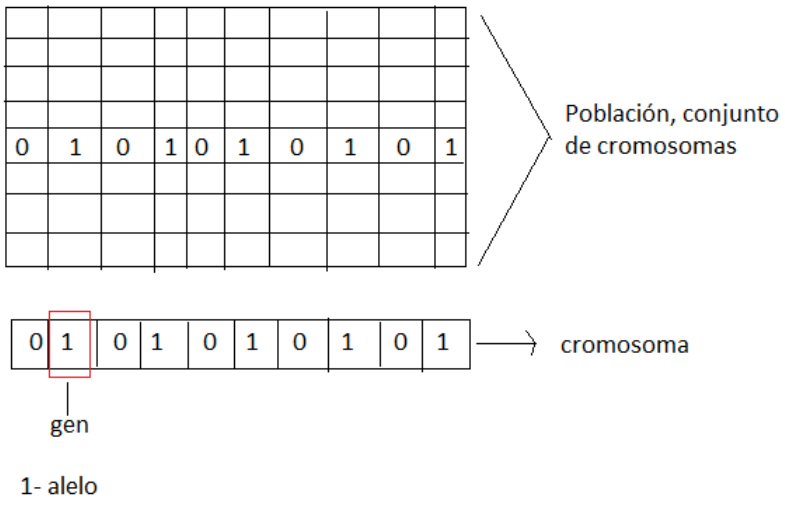
Antecedentes

Este tipo de algoritmos forman parte de una rama de la computación conocida como "Computación evolutiva"; se basan en la teoría de la evolución de Darwin, y sus bases fueron creadas por John Holland en 1962, aunque no fue hasta 1984 cuando formalizó y exploró diversos conceptos de este método, en su publicación *Genetic Algorithms and Adaptation*, que se encuentra en [25].

Algoritmo

Algunos conceptos importantes para este algoritmo son:

- Población: conjunto de todas las posibles soluciones.
- Individuos: solución candidata.
- Cromosomas: solución dada. Puede expresarse mediante una cadena de números binarios (1 o 0), pero también se puede realizar la codificación mediante números enteros o incluso cadenas de palabras, dependiendo del problema a tratar.
- Gen: elemento de la solución.
- Alelo: valor que toma un gen de un cromosoma en particular.



Además, tenemos el genotipo, que es la población en el espacio computacional y el fenotipo, que son las soluciones en el mundo real.

Estructura básica del algoritmo

Como ya se mencionó anteriormente, este método está basado en el proceso de selección natural. Es por esto por lo que algunas de las etapas del algoritmo llevan nombres como *mutación* o *cruzamiento*.

Si bien existen muchas variantes del método en cuestión, en general, la estructura de los algoritmos genéticos se representa como sigue :

1. Como en el resto de los algoritmos de optimización, es necesario generar una población inicial aleatoria, para así asegurar la diversidad de soluciones.

2. Evaluamos la puntuación de cada individuo de la población mediante la función de aptitud, que indica la contribución de cada elemento a la solución general, es decir, si el objetivo es maximizar, cuanto mayor sea el valor de la función para el individuo, mayor su fitness. En el caso de minimización, ocurre lo contrario.

3. Viene la etapa del cruzamiento o *crossover*, en la que se utilizan características de dos individuos para generar nuevos.

4. Con cierta probabilidad de mutación, se realiza un cambio en alguna característica del nuevo individuo.

5. Se organiza la nueva población y se repite el paso dos hasta que se cumpla el criterio de paro definido.

A continuación veremos con mayor detalle las etapas mencionadas hace unas líneas.

Selección

De acuerdo a la teoría de evolución de Darwin, los individuos más aptos son los que sobreviven. Es por esto por lo que existen diversas formas de seleccionar a los padres que generarán la nueva generación.

- Selección por Ruleta: Se crea para esta selección una "ruleta" con los cromosomas presentes en una generación, de forma que cada cromosoma tiene una parte de esta ruleta, mayor o menor en función a su valor de aptitud; así, mientras mejor sea su contribución a la función objetivo, más probabilidad tendrá de ser elegido. Se hace girar la ruleta y se selecciona el cromosoma en el que esta se detenga. [42] El problema con este método es que, si el valor de aptitud de unos pocos individuos es muy

superior al resto, estos serán seleccionados de forma repetida habrá poca diversidad genética. En la Figura 4.20 se muestra un ejemplo de éste tipo de selección.

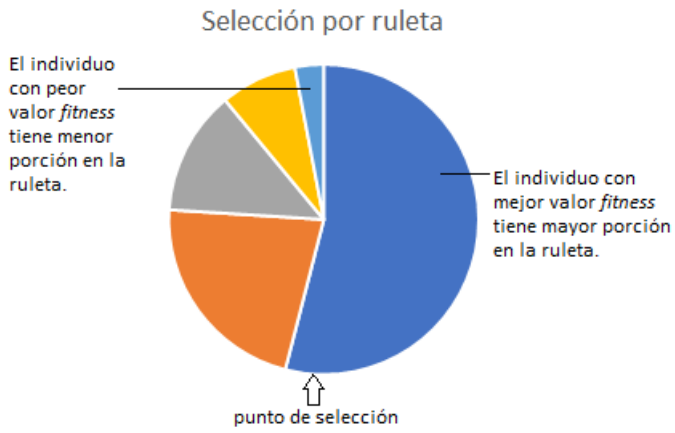


Figura 4.20: Selección por ruleta

- Selección por torneo: Se seleccionan aleatoriamente dos parejas de individuos de la población, todos con la misma probabilidad. De cada pareja se selecciona al que tenga mayor *fitness*, y así sucesivamente, hasta obtener al final a los dos padres que se cruzarán [42]. Por ejemplo, en la Figura 4.21 podemos ver que hay seis individuos en la población inicial, caracterizados por las letras A, B, C, D, E y F, con los cuáles se forman dos parejas (C,F y A,B); se comparan los valores de la función objetivo de cada individuo de la pareja y se elige aquel con el mejor valor para obtener finalmente a los progenitores C y B.

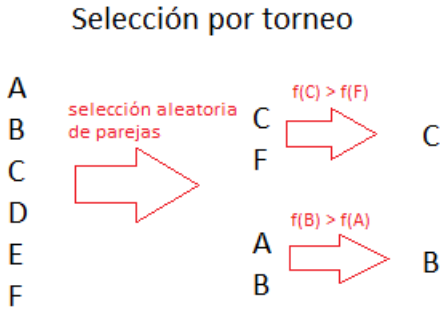


Figura 4.21: Selección por torneo

- Selección truncada: se realizan selecciones aleatorias de individuos, habiendo descartado primero a los n individuos con peor valor de aptitud de la población [42]. En la Figura 4.22 tenemos una población inicial cuyos individuos son identificados con las letras A, B, C, D, E, F, G y H. Se ordenan estos individuos de mejor a peor de acuerdo a su valor de aptitud y luego se eliminan los cuatro peores (E, D, B y G) para elegir posteriormente, de manera aleatoria, a quienes serán los progenitores (H y C).

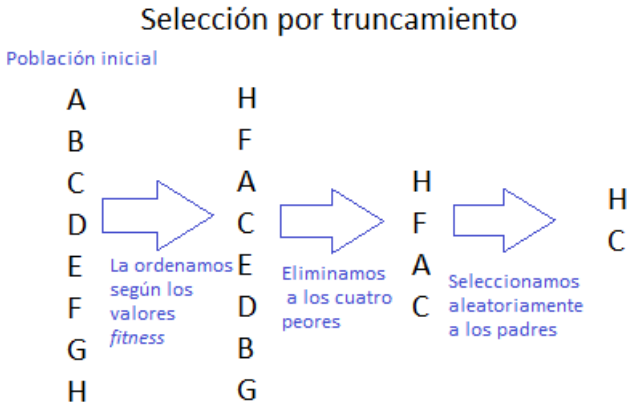


Figura 4.22: Selección truncada

- Selección elitista: Se copia el o los mejores cromosomas de la generación anterior para formar parte automáticamente de la nueva solución. El resto de la población se elige mediante alguno de los métodos ya mencionados. [42]

Cruzamiento

Una vez seleccionados los padres, es necesario combinarlos de manera que se generen nuevas soluciones que formarán parte de la nueva generación.

- Cruzamiento a partir de un solo punto: se define un punto aleatorio de corte en ambos cromosomas padres. Se copia la información genética de uno de ellos desde el inicio hasta el punto de cruce, y el resto se copia del otro progenitor. Como resultado de este proceso, por cada cruce, se generan dos nuevos individuos. En la Figura 4.23 se intercambian características de ambos padres a partir de un punto de corte, dando lugar a dos individuos distintos. [42]

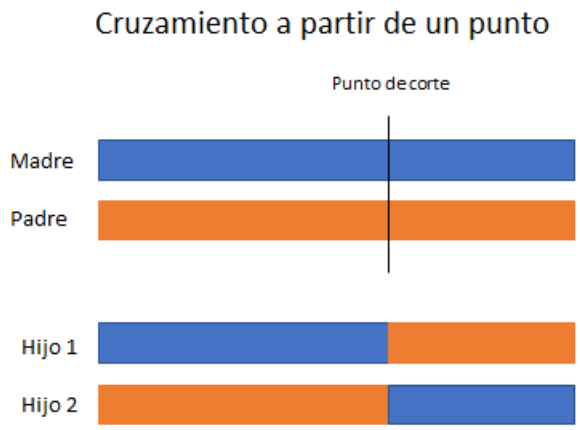


Figura 4.23: Cruzamiento a partir de un punto

- Cruzamiento a partir de dos puntos: es muy similar al método anterior, salvo que, en esta ocasión, se seleccionan aleatoriamente dos puntos de corte. Cada individuo se divide por los puntos de corte y se intercambian las partes[42]. En la Figura 4.24 se intercambian características de los progenitores a partir de dos puntos de corte, dando como resultado dos hijos, uno de los cuales está más cargado con información genética de la madre y otro contiene mayormente información genética del padre.

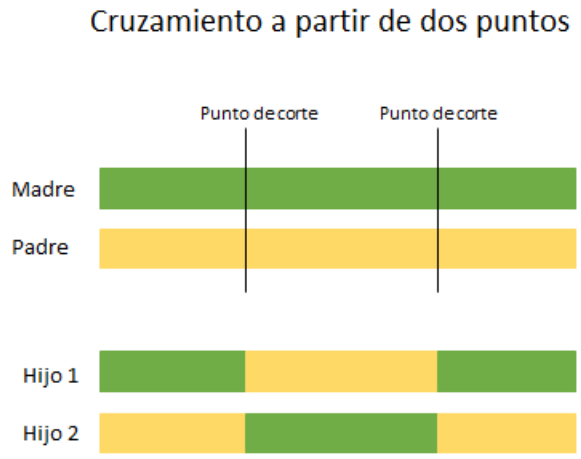


Figura 4.24: Cruzamiento a partir de dos puntos

- Cruzamiento uniforme: el valor del gen de cada individuo se obtiene copiando el

correspondiente gen de uno de los dos padres, escogido de acuerdo a una máscara de cruce generada aleatorimante, donde, al haber un 1, se copia el gen del primer progenitor, y al haber un 0, se copia el del segundo[42]. En la Figura 4.25 tenemos una máscara que copia cinco genes del padre y tres de la madre, dando como resultado un único hijo.

Figura 4.25: Cruzamiento uniforme

- Cruzamiento aritmético: Los padres se recombinan según algún operador aritmético para generar a los miembros de la nueva generación. [42]

Mutación

Una vez generados los nuevos individuos, cada uno de ellos se somete a un proceso de cambio, donde alguna de sus características puede verse modificada con una probabilidad p. Este paso es importante para añadir diversidad al proceso y evitar que el algoritmo caiga en mínimos locales debido al parecido entre generaciones. Existen diversas formas de mutar a los cromosomas, que dependen generalmente de cómo está representada la solución. Por ejemplo, podríamos intercambiar las posiciones de características, es decir, si un individuo se compone por $[i,j,k]$, una mutación de este podría verse como $[i,k,j]$. Otra manera de mutar la posición j, sería sumándole a ésta un número u tal que $u \epsilon$ U(0,1). De otra forma, la mutación de la posición j podría conseguirse reemplazando el valor de j por un nuevo valor aleatorio dentro del rango permitido para esa variable.

Podemos decir que, de manera general, los algoritmos genéticos son mecanismos que simulan la evolución de las especies inspirándose en la idea de que el individuo más apto es quien sobrevive. De esta manera, calcula los valores de aptitud a los individuos de una población para que haya cruzamientos y mutaciones entre ellos que generen nuevos individuos, a partir de los mejores progenitores de una población. En la Figura 4.26 se presenta un diagrama de flujo con los pasos que siguen los algoritmos genéticos.

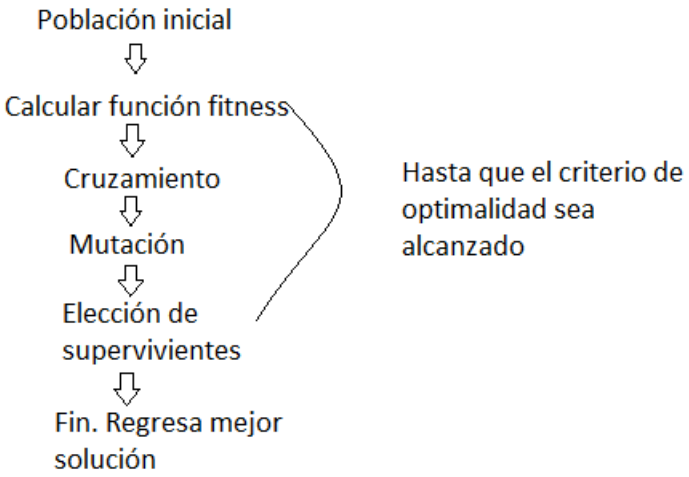


Figura 4.26: Diagrama de flujo de algoritmos genéticos

Este método de optimización garantiza que con cada iteración la solución va mejorando, pues se basa en el principio de sobreviviencia del más fuerte. Además, el que se implementen mutaciones permite que las soluciones se diversifiquen. Las mayores desventajas aquí consisten en que los cruzamientos y mutaciones pueden definirse de formas tan distintas, que es difícil encontrar el mejor para el problema que se busca resolver.

Ejemplo

A continuación veremos un ejemplo obtenido de [8] en que se utiliza la metaheurística de *Algoritmos genéticos*:

Se desea minimizar $f(x,y) = xsen(4x) + 1.1y\ sen(2y)$, para $0 \leq x \leq 10$, $0 \leq y \leq 10$. Sea N_{par} = dimensión de los cromosomas y $N_{poblacin}$ = número de individuos en la población.

Como los cromosomas son de la forma *x,y*, $N_{par} = 2$. Además, $N_{poblacin} = 12$ cromosomas generados aleatoriamente. Se define una tasa de mutación $\mu = 0.2$ y un total de 100 iteraciones. Finalmente, en lugar de reemplazar a toda la población de padres con la nueva generación, mantendremos la mitad más adaptada. En la Tabla 4.6 se presenta la población inicial.

x	*y*	*f(x,y)*
6.7874	6.9483	13.5468
7.5774	3.1710	-6.5696
7.4313	9.5022	-5.7656
3.9223	0.3445	0.3149
6.5548	4.3874	8.7209
1.7119	3.8156	5.0089
7.0605	7.6552	3.4901
0.3183	7.9520	1.3994
2.7692	1.81687	-3.9137
0.4617	4.8976	-1.5065
0.9713	4.4559	1.7482
8.2346	6.4631	10.7287

Tabla 4.6: Población inicial

Se ordena a los individuos de acuerdo a su valor en *f(x,y)*, que es mejor mientras más pequeño sea, pues el objetivo es minimizar $f(x,y) = xsen(4x) + 1.1y\ sen(2y)$. De esta manera, el individuo (7.5774, 3.1710) es considerado el mejor y el individuo (6.7874, 6.9483) es el peor. Una vez ordenados, se eliminan los seis con peores valores. En la Tabla 4.7 se muestra la población sobreviviente a este método de selección.

Rango	*x*	*y*	*f(x,y)*
1	7.5774	3.1710	-6.5696
2	7.4313	9.5022	-5.7656
3	2.7692	1.81687	-3.9137
4	0.4617	4.8976	-1.5065
5	0.3183	7.9520	-1.3994
6	3.9223	0.3445	0.3149

Tabla 4.7: Población sobreviviente después de la tasa de selección

La probabilidad de que el cromosoma que se encuentra en el lugar n sea un padre, se

calcula de la siguiente manera:

$$P_n = \frac{N_1 - n + 1}{\sum_{i=1}^{N_1} i}$$

Donde N_1 es el número de individuos en la primera generación y n es el lugar que ocupa el cromosoma.

Así, la probabilidad de que el cromosoma 1 de la Tabla 4.7 sea un padre es:

$$P(C_1) = \frac{6 - 1 + 1}{21} = \frac{6}{21} = 0.285714 = 28.5714\,\%$$

y, de la misma manera, la probabilidad de que el cromosoma 6 sea progenitor es:

$$P(C_1) = \frac{6 - 6 + 1}{21} = \frac{1}{21} = 0.0476 = 4.76\,\%$$

Cada pareja de padres genera 2 hijos, por lo que necesitamos 3 pares de cromosomas para crear la nueva generación. Una vez que tenemos los pares de cromosomas $m = [x_m, y_m]$ y $p = [x_p, y_p]$, elegimos aleatoriamente un punto de cruce k. Para generar a los hijos, primero generamos una combinación lineal convexa entre los padres. Para ello, necesitamos un parámetro $\beta\ \epsilon$ [0,1]. Una vez determinado el valor de β, la coordenada x asociada a cada par de individuos denotados cokmo x_{nuevo1} y x_{nuevo2} se generan del siguiente modo:

x_{nuevo1} = (1-β) x_m + β x_p

x_{nuevo2} = (1-β) x_p + β x_m

Así los hijos tienen la siguiente forma:

$hijo_1 = [x_{nuevo1},\ y_m]$ e $hijo_2 = [x_{nuevo2},\ y_p]$

En ete ejemplo se cruzaron los individuos 1 con 3, con un valor $\beta = 0.3463$, 4 con 2, con un valor para $\beta = 0.7898$ y finalmente cruzamos a los individuos 5 con 6, con un valor para $\beta = 0.5468$.

Los cruzamientos nos dieron como resultado 6 hijos, de forma que la nueva población se presenta en la Tabla 4.8.

Rango	x	y	$f(x,y)$
1	7.5774	3.1710	-6.5696
2	7.4313	9.5022	-5.7656
3	2.7692	1.81687	-3.9137
4	0.4617	4.8976	-1.5065
5	0.3183	7.9520	-1.3994
6	3.9223	0.3445	0.3149
7	5.9123	3.1710	-5.6847
8	4.4343	1.8687	-5.12996
9	5.9662	4.8976	-7.6449
10	1.9267	9.5022	3.51790
11	2.2889	7.9520	-1.0951
12	1.9516	0.3445	2.19031

Tabla 4.8: Nueva población

Recordemos que la tasa de mutación es el 20 %, por lo que el número de mutaciones a realizar es el siguiente:

$$\#mutaciones = \mu(N_{poblacin} - 1)N_{par} = 0.2(12 - 1)(2) = 4.4$$

Por lo tanto, se realizan 4 cambios aleatorios a los individuos de la nueva población. Repetimos el proceso anterior hasta que se cumpla el criterio de paro (100 iteraciones).

Implementación del problema de las n esferas

A continuación presento los detalles de implementación para el problema de empaquetamiento de esferas dentro de un contenedor esférico, siguiendo el procedimiento presentado en Algoritmos Genéticos.

1. Genero una población inicial de tamaño n, con n par.

2. Defino número de iteraciones y la función de aptitud, que da un valor para R_0 suponiendo que solo existe la esfera i dentro de S.

3. Calculo la función de aptitud para cada elemento de la polación y ordeno estos elementos de mejor a peor con respecto al valor obtenido.

4. Creo a los padres con el método de la *ruleta*.

5. Genero los cruzamientos utilizando un cruzamiento aritmético: Cada madre se empareja con un padre, y cada pareja de progenitores genera dos hijos; suponiendo que la madre tiene coordenadas (x_m, y_m, z_m) y el padre (x_p, y_p, z_p), los hijos estarán definidos como:

   - $h_1 = (x_{nuevom}, y_{nuevom}, z_{nuevom})$
   - $h_2 = (x_{nuevop}, y_{nuevop}, z_{nuevop})$,
     donde $x_{nuevom} = (1- \beta)x_m + \beta\ x_p$, $x_{nuevop} = (1- \beta)x_p + \beta\ x_m$, $y_{nuevom} = (1-\beta)y_m + \beta\ y_p$, $y_{nuevop} = (1-\beta)y_p + \beta\ y_m$, $z_{nuevom} = (1-\beta)z_m + \beta\ z_p$ y $z_{nuevop} = (1-\beta)\ z_p + \beta\ z_m$

6. No se implementa elitismo: los hijos entran a la población tomando el lugar de los padres, pues de esta forma, en cada iteración tenemos una población completamente distinta.

7. Con una probabilidad de $\mu = 0.2$ se muta a los individuos, sumándole un número $u \in (-0.3, 0.3)$ a la coordenada más alejada, pues de esta forma podemos acercar esta coordenada al centro o alejarla no más de 0.3 unidades.

8. Se reordenan los nuevos individuos y se repite el procedimiento k veces.

Fueron necesarias 1500 iteraciones, ya que con 1000 iteraciones, para n=10, obtenía un radio final de 0.3736917 unidades en 6.80 segundos, y con 1500 iteraciones un radio de 0.3581568 en 10.17 segundos. En el caso en que n=50 ocurrieron cosas similares pues al utilizar 1000 iteraciones, la solución obtenida R(s)=0.6824516 fue encontrada en 18.2 segundos, mientras que al elevar el número de iteraciones a 1500, el nuevo radio fue de 0.63288 en 57.11 segundos. Intenté aumentar el número de iteraciones a 15,000, sin embargo, no hubo cambios en la solución.

De igual forma, utilicé una semilla de 11 pues, al utilizar otras, obtenía los siguientes resultados (para n=10):

- Semilla 11: R(s)= 0.3581568
- Semilla 21: R(s)= 0.3629289

- Semilla 7: R(s)= 0.4791553
- Semilla 15: R(s)=0.4091834
- Semilla 46: R(s)=0.3758962

Ahora veremos los resultados para n=30 y n=90.

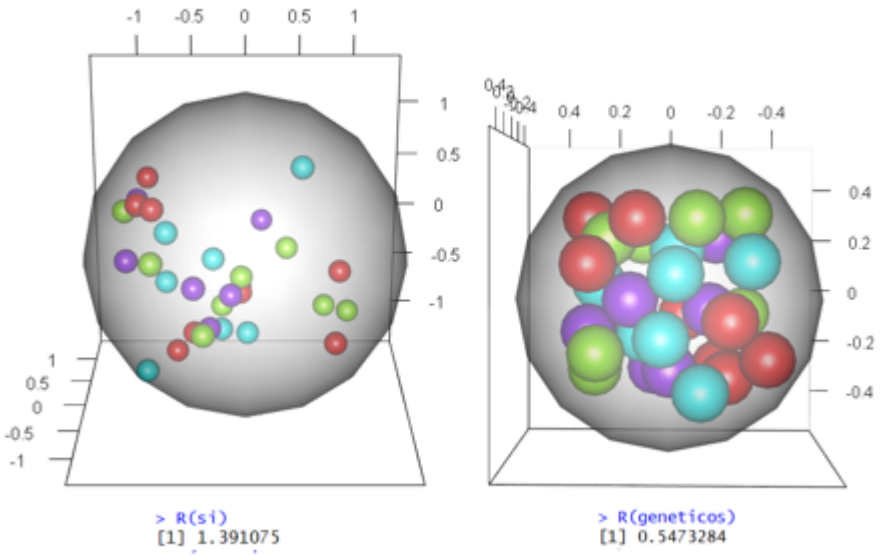


Figura 4.27: Algoritmos genéticos, n=30

Hubo un cambio de 0.8439466 unidades con respecto al radio de la solución inicial.

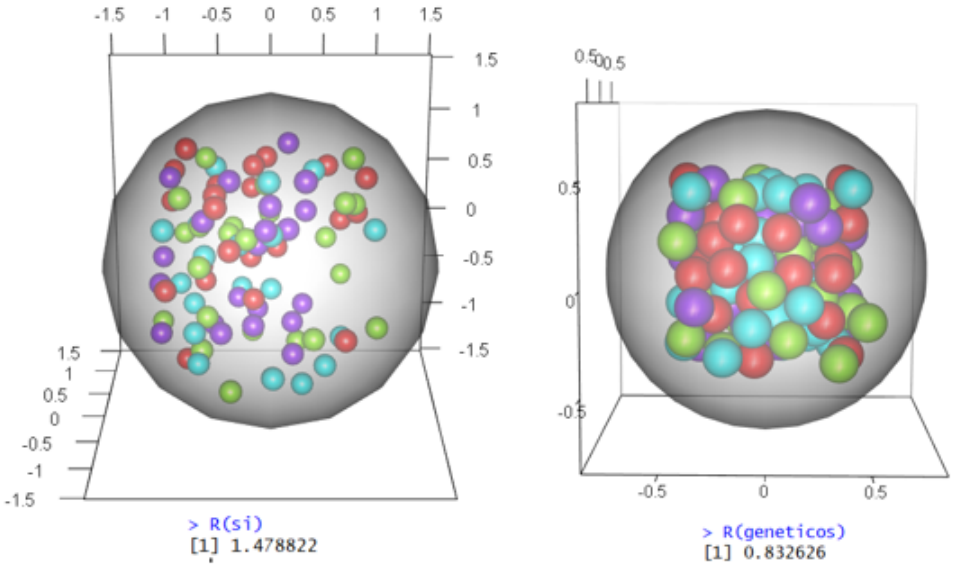


Figura 4.28: Algoritmos genéticos, n=90

El radio final fue menor que el inicial por 0.646196 unidades.

Capítulo 5

Resultados

A continuación veremos los resultados para cada instancia, con cada heurística, lo que nos permitirá comparar cómo fueron los radios obtenidos mediante cada método para el contenedor S. Es importante notar que en todas las heurísticas comencé con la misma solución inicial, de forma que las comparaciones de tamaño de los radios del contenedor fueran lo más acertadas posible. Esto cambia únicamente en GRASP, pues recordemos que este método inicia con una solución creada en la primera fase del método.
En cada una de las figuras (5.1 - 5.10) se muestran las imágenes de las soluciones iniciales a la izquierda, sobre sus respectivos radios, y las de las soluciones finales a la derecha.

Si bien las imágenes pueden darnos una idea de cuánto mejoraron las soluciones con respecto a su solución inicial, así como cuál de las heurísticas generó el mejor cambio, siempre es necesario revisar los datos numéricos. Es por esto por lo que en la Tabla 5.1 se muestran los valores obtenidos para R_0 en cada heurística.

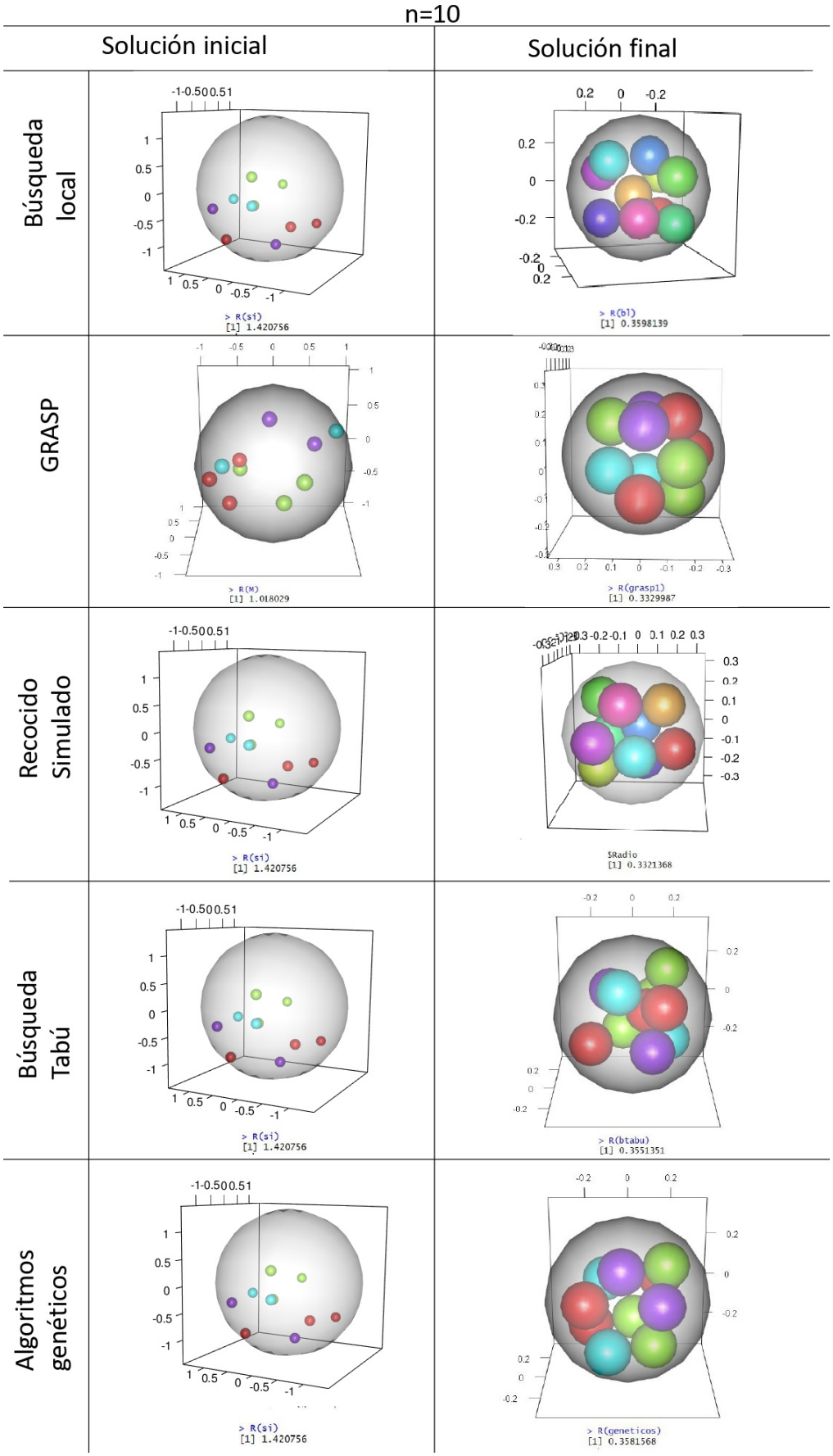


Figura 5.1: Resultados n=10

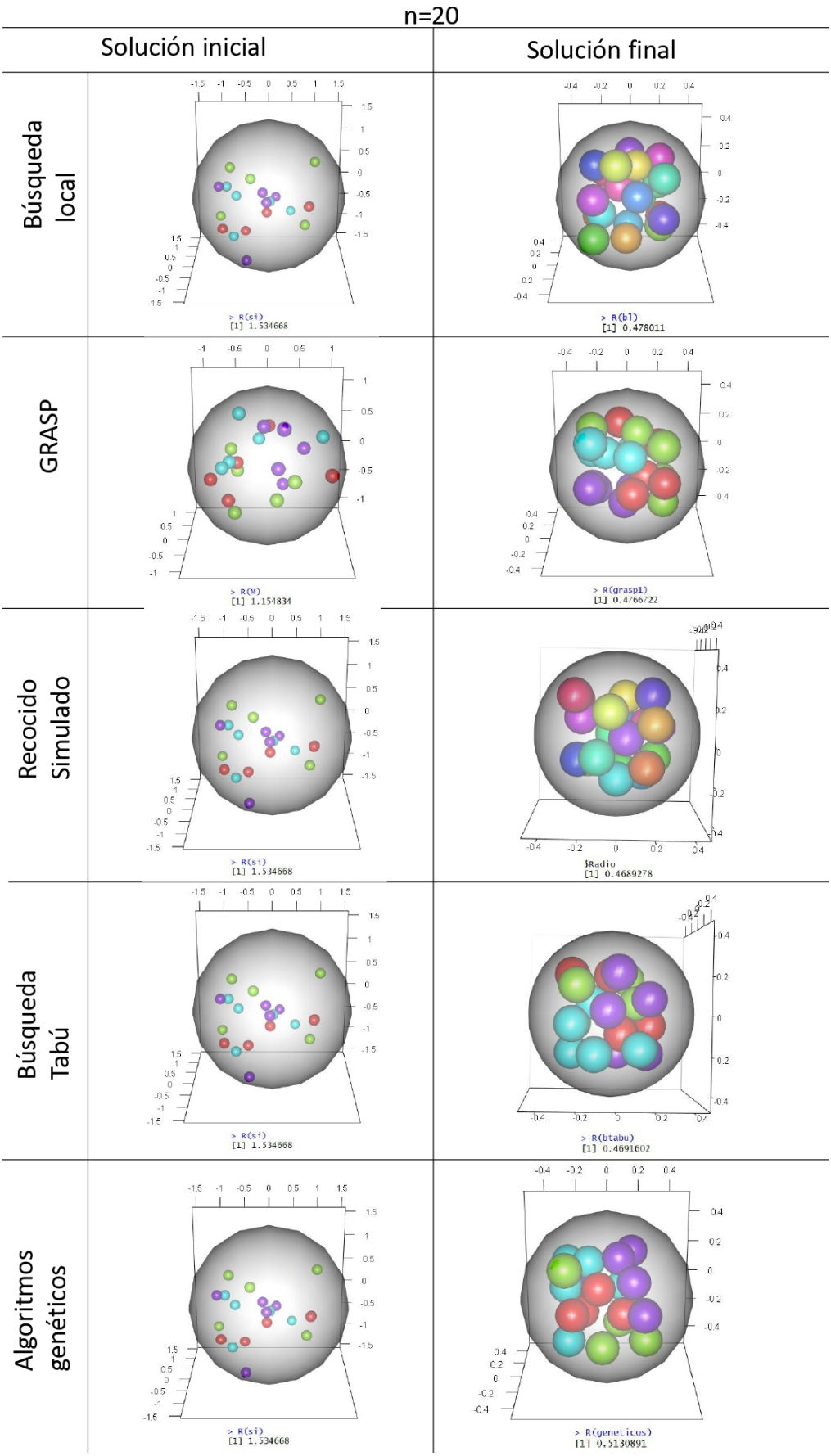


Figura 5.2: Resultados n=20

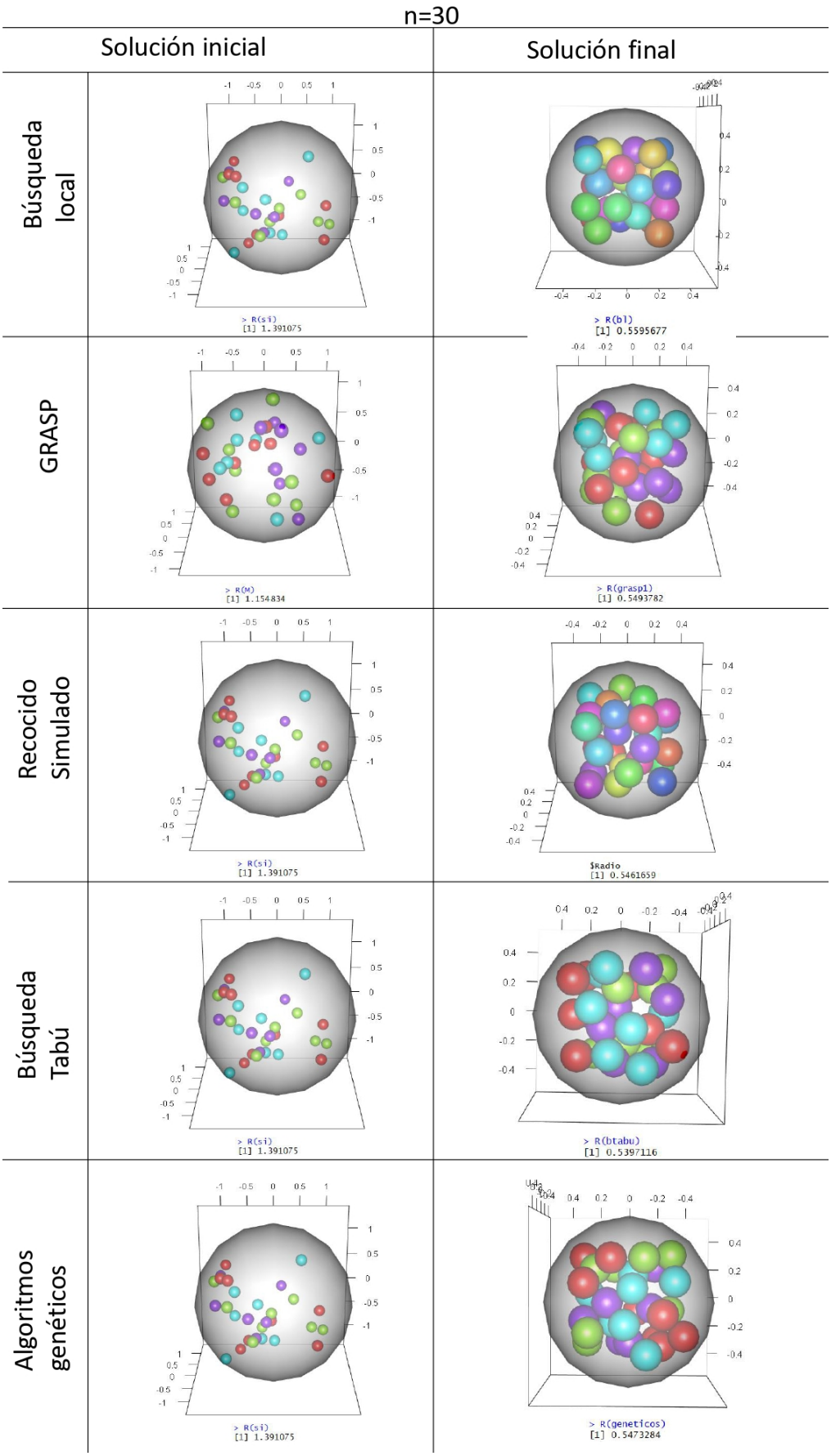


Figura 5.3: Resultados n=30

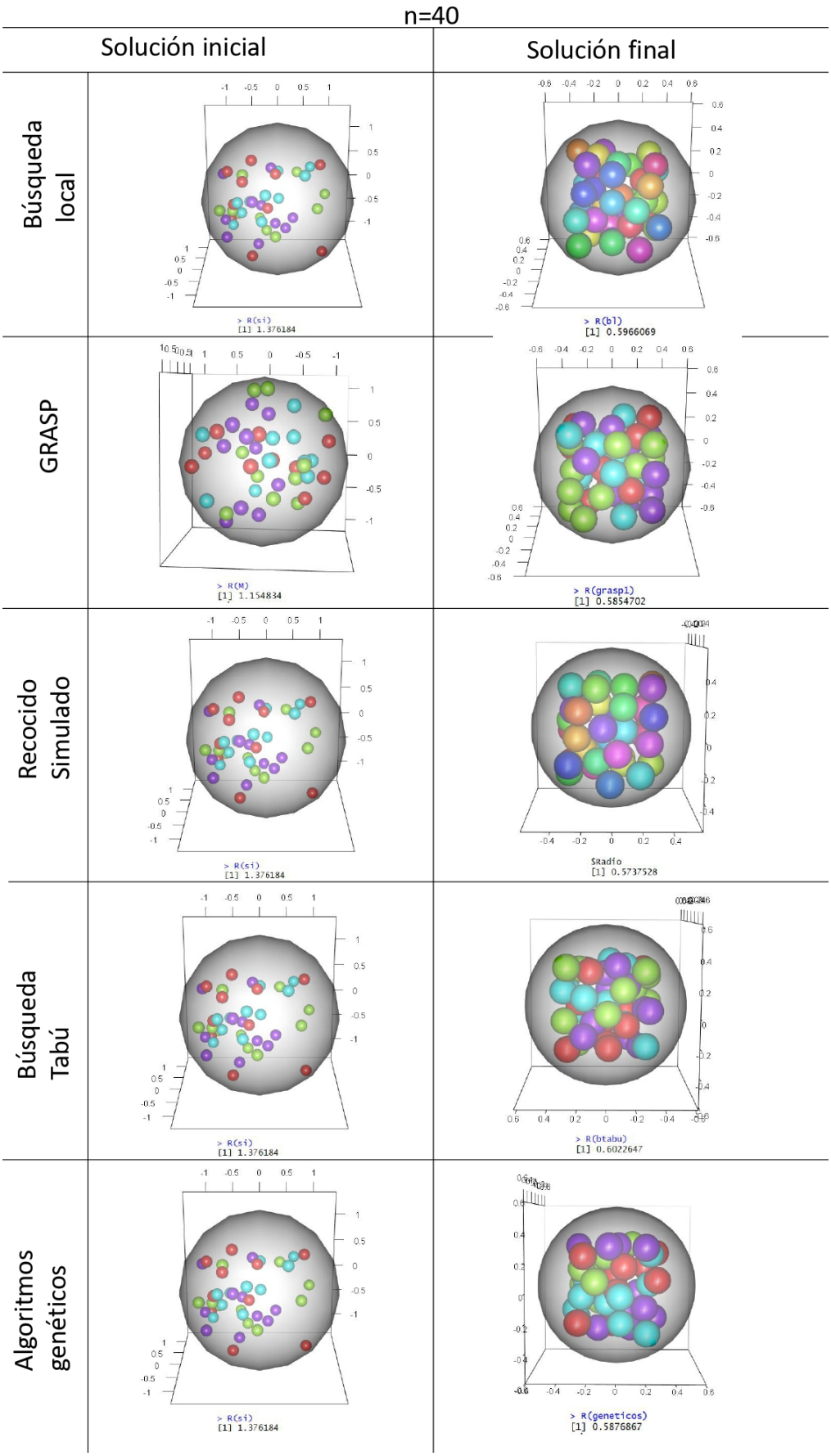


Figura 5.4: Resultados n=40

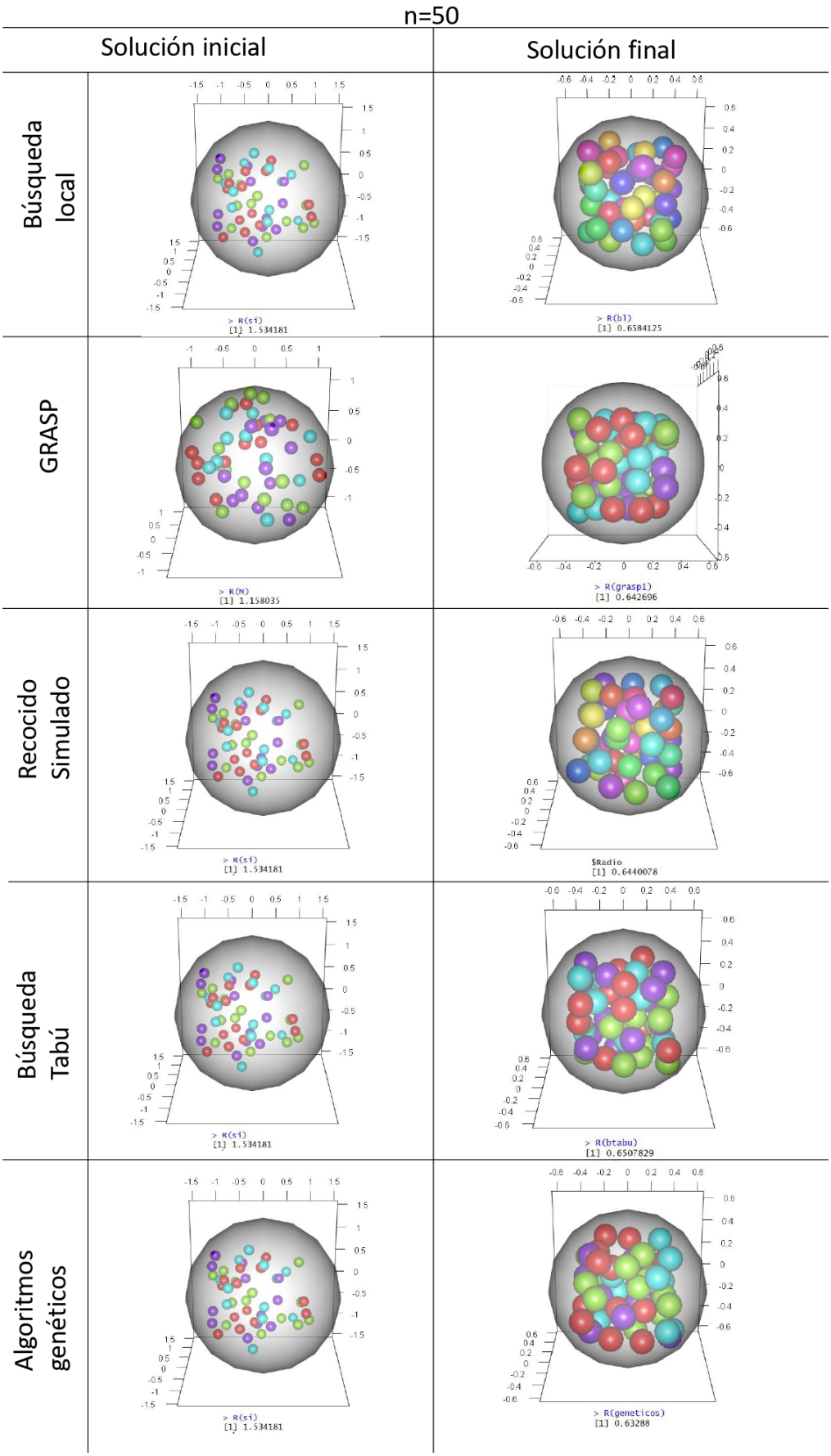


Figura 5.5: Resultados n=50

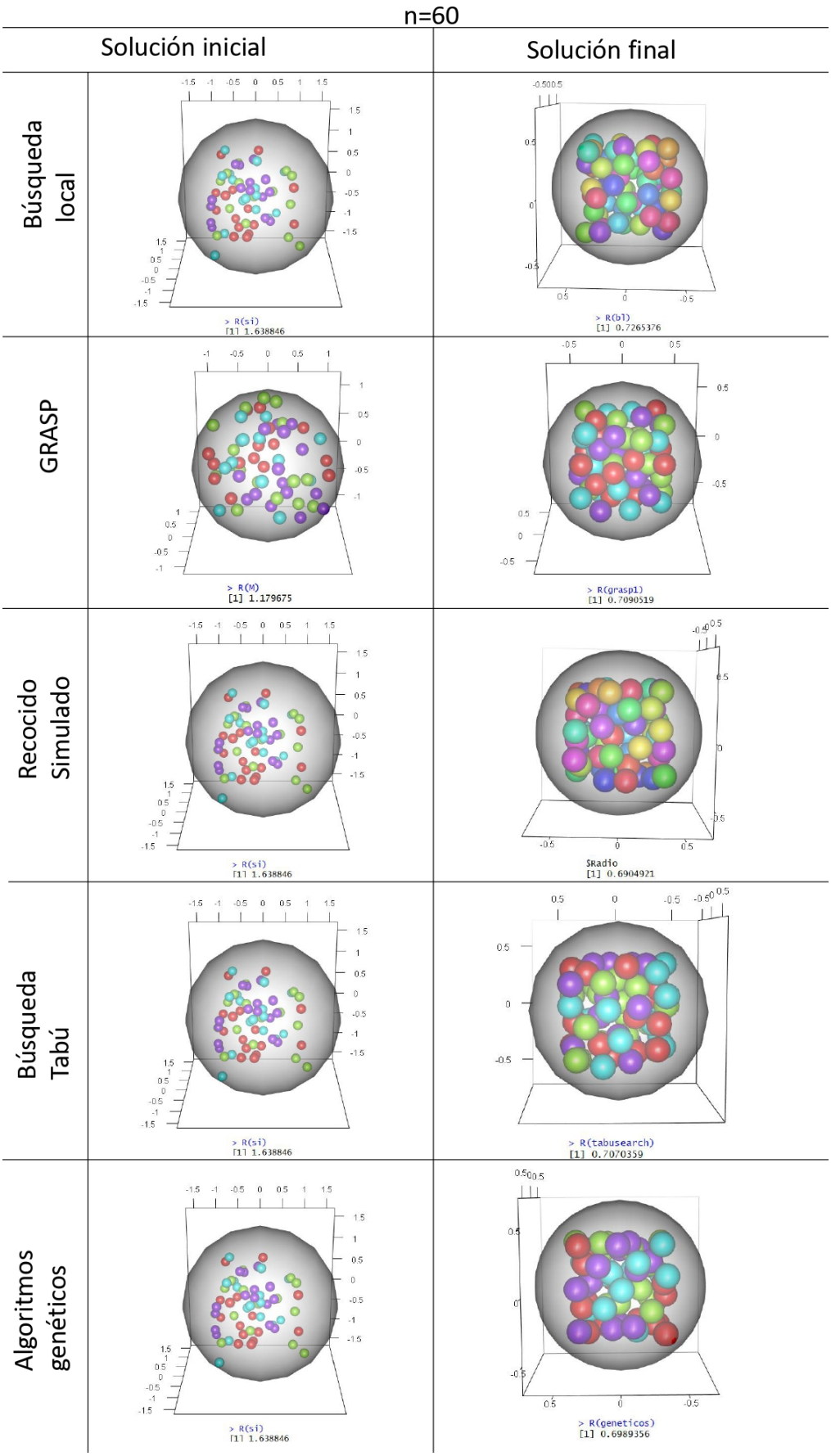


Figura 5.6: Resultados n=60

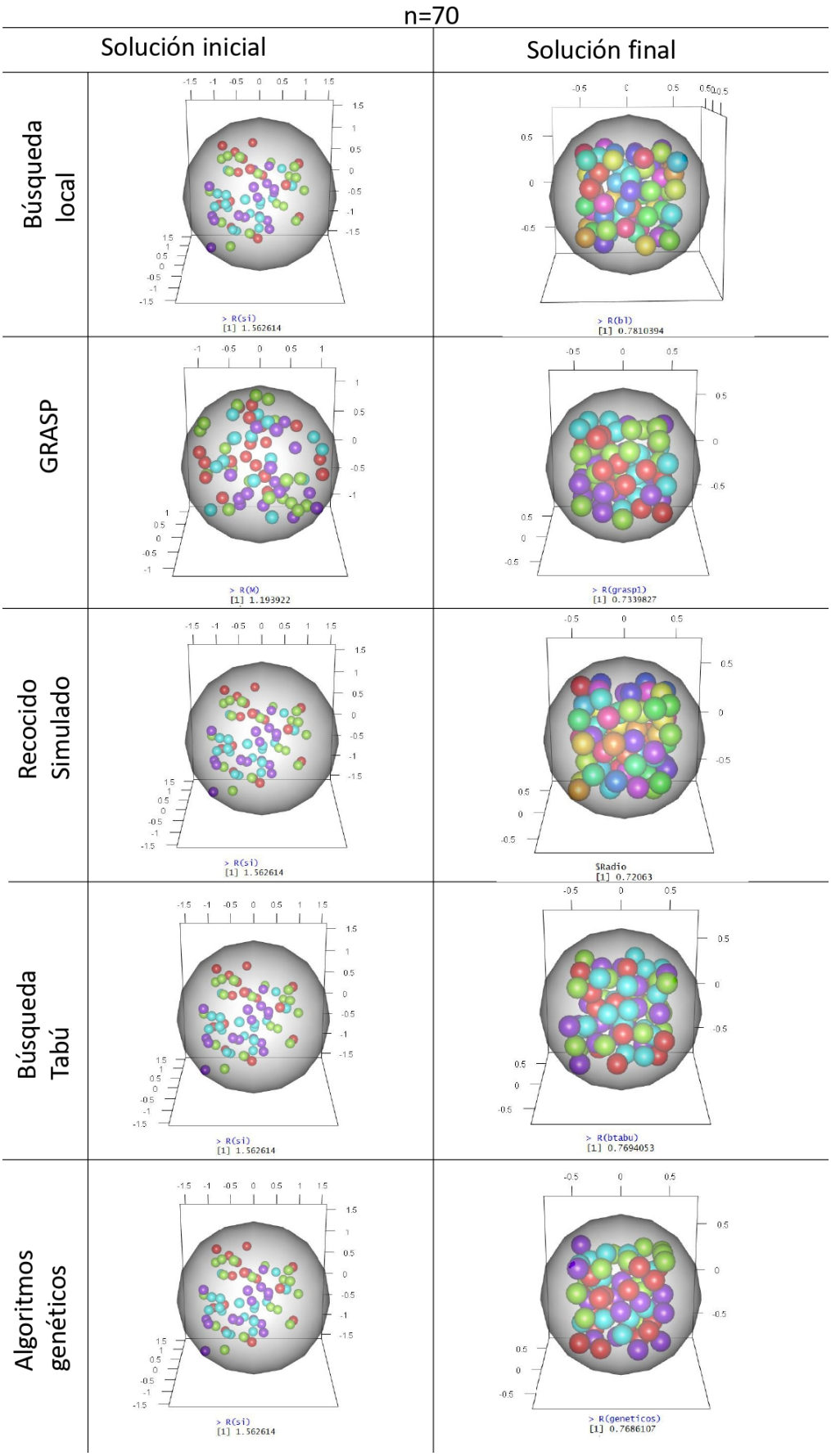


Figura 5.7: Resultados n=70

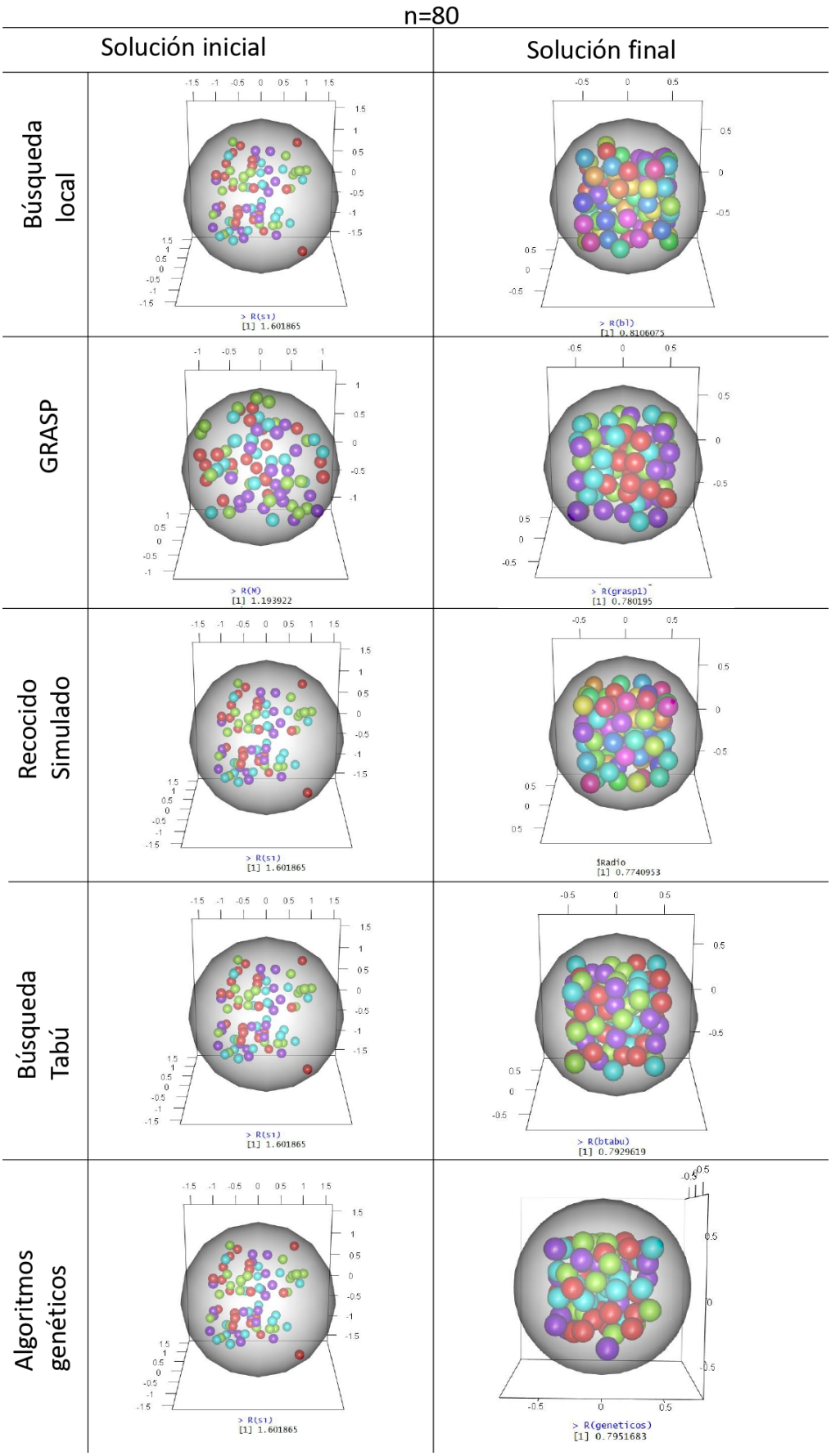


Figura 5.8: Resultados n=80

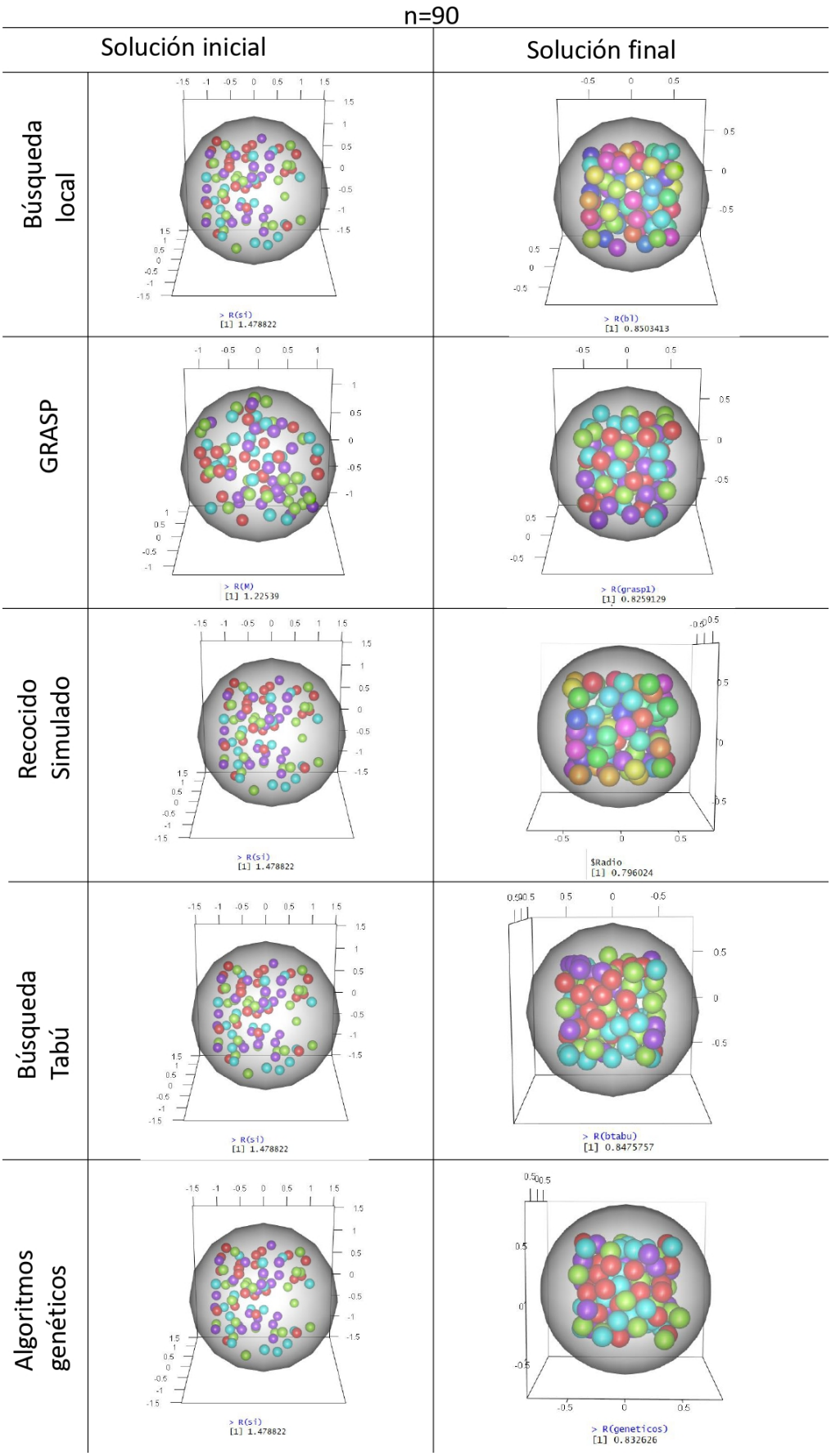


Figura 5.9: Resultados n=90

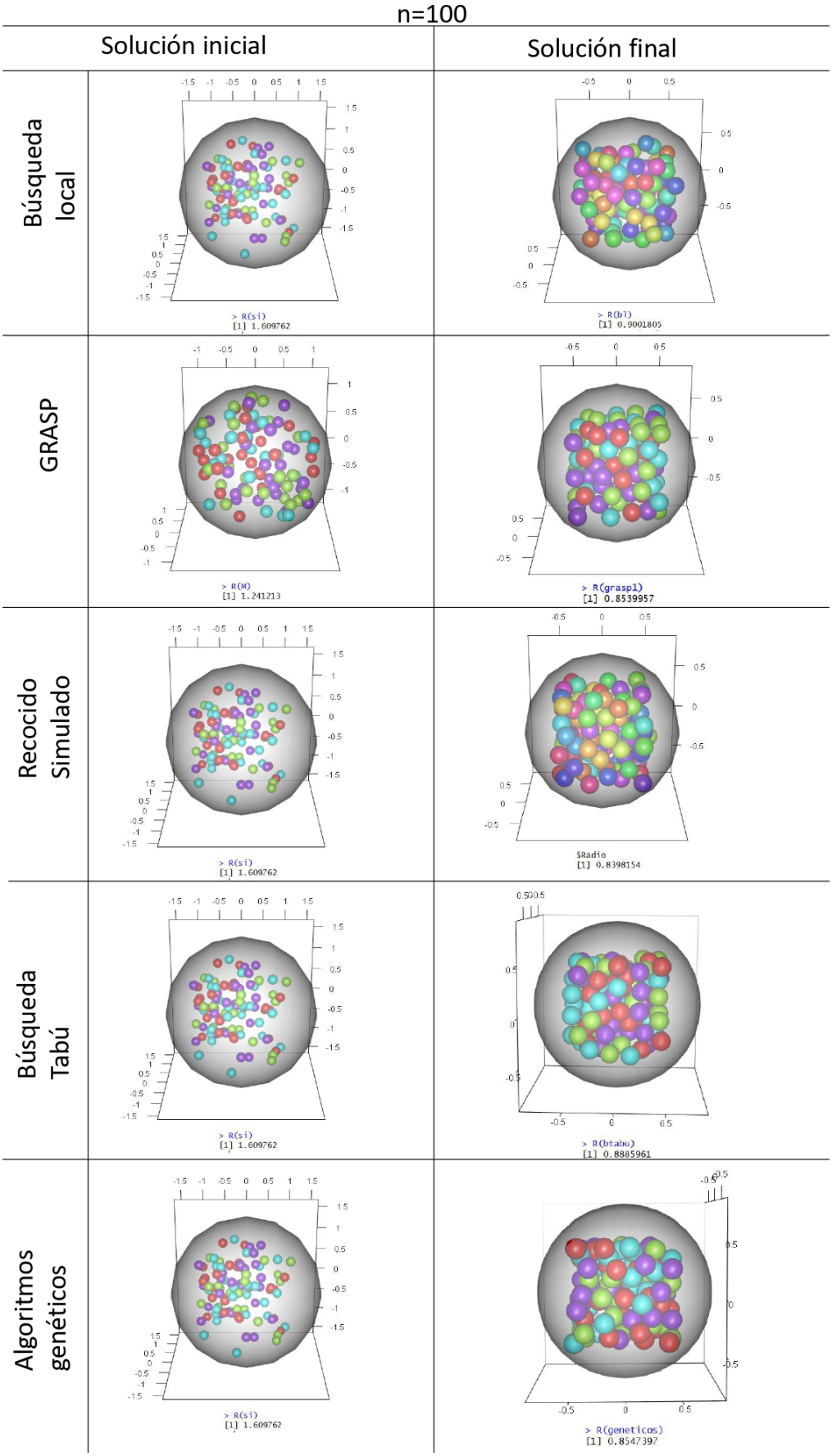


Figura 5.10: Resultados n=100

Ahora presentaré una tabla donde se pueden ver de manera ordenada los radios obtenidos mediante la aplicación de cada método:

n		Búsqueda Local	GRASP	Recocido Simulado	Búsqueda Tabú	Algoritmos genéticos
10	s.i.	1.420756	1.018029	1.420756	1.420756	1.420756
	final	0.3598039	0.3329987	0.3321368	0.3551351	0.3581568
20	s.i.	1.534668	1.154834	1.534668	1.534668	1.534668
	final	0.478011	0.4766722	0.4689278	0.4691602	0.5130891
30	s.i.	1.391075	1.134884	1.391075	1.391075	1.391075
	final	0.5595677	0.5493782	0.5461659	0.5397116	0.5473284
40	s.i.	1.37614	1.156737	1.37614	1.37614	1.37614
	final	0.5966069	0.5854702	0.5737528	0.6022647	0.5876867
50	s.i.	1.534181	1.158035	1.534181	1.534181	1.534181
	final	0.6584125	0.642696	0.6440078	0.6507829	0.63288
60	s.i.	1.638846	1.179675	1.638846	1.638846	1.638846
	final	0.7265376	0.7090519	0.6904921	0.7070359	0.6989356
70	s.i.	1.562614	1.198743	1.562614	1.562614	1.562614
	final	0.7810394	0.7339827	0.72063	0.7694053	0.7686107
80	s.i.	1.601865	1.193922	1.601865	1.601865	1.601865
	final	0.8106075	0.780195	0.7740953	0.7929619	0.7951683
90	s.i.	1.478822	1.22539	1.478822	1.478822	1.478822
	final	0.8503413	0.8259129	0.796024	0.8475757	0.832626
100	s.i.	1.609762	1.241213	1.609762	1.609762	1.609762
	final	0.9001805	0.8539957	0.8398154	0.8885961	0.8547397
mean		0.67211083	0.64903535	0.63860479	0.66226294	0.65892213

Tabla 5.1: Soluciones iniciales y finales

Recordemos además, que una de las partes más importantes de la programación es el costo temporal de ejecución del código. A continución presento cuánto tardó en compilar cada código programado para este documento:

Número de esferas	Búsqueda Local	GRASP	Recocido Simulado	Búsqueda Tabú	Algoritmos genéticos
10	9.99	11.73	389.07	7.71	10.17
20	26.01	32.36	467.14	12.10	13.36
30	66.62	70.53	1534.64	17.77	15.91
40	95.21	101.19	3142.25	45.72	27.13
50	112.61	131.48	3886.11	129.07	39.2
60	143.97	166.12	5386.51	158.01	57.11
70	188.12	210.73	7572.84	197.94	69.36
80	203.1	224.89	8612.27	218.15	80.18
90	221.88	250.6	12867.06	247.77	97.14
100	245.43	288.47	14647.21	263.52	118.11
mean	131.294	148.81	5850.51	129.776	52.767
Los tiempos de ejecución están en segundos					

Tabla 5.2: Tiempos de ejecución

Comparando ambas tablas, podemos notar que, en promedio, el *Recocido Simulado* fue el que nos dio valores más pequeños para R_0, sin embargo, fue el que tardó más tiempo en ejecutarse, tomándose 5797.743 segundos más que *Algoritmos genéticos*, que fue el que se llevó a cabo más rápido.

GRASP fue el segundo algoritmo con los mejores valores para el radio, siendo el segundo más tardado, solo superado por *Recocido Simulado*, pero alejándose solo por 96.043 segundos de la metaheurística más rápida.

Programé nuevamente GRASP para probar otro tipo de construcción de la solución inicial, con el fin de ver si había alguna mejora con respecto a la otra construcción, comenzando en esta ocasión con una solución inicial construida por valor. Recordemos, del apartado de esta heurística, que esta construcción calcula los costos miopes de la solución, toma el mínimo (c_*) y el máximo (c^*), y crea la Lista Restringida de Candidatos (RCL) a partir de aquellos costos miopes que son menores o iguales a $c_* + \alpha(c^* - c_*)$, tal que $0 < \alpha < 1$. Como ya se mencionó también, en caso de que $\alpha = 0$, el algoritmo es miope, y cuando $\alpha = 1$, el algoritmo es aleatorio. Es por esto último por lo que utilicé $\alpha = 0.5$, así como una semilla de 11 para comparar los resultados obtenidos con el resto de las heurísticas. Al igual que en GRASP, utilicé 10,000 iteraciones. En la Tabla 5.3 se muestran las soluciones con ambas construcciones hechas con GRASP, así como sus tiempos de ejecución:

n	Solución	GRASP usando cardinalidad	Tiempo	GRASP usando Valor	Tiempo
10	s.i. final	1.018029 0.3329987	11.73	1.243879 0.3591528	6.9
20	s.i. final	1.154834 0.4766722	32.36	1.427766 0.4787477	51.23
30	s.i. final	1.134884 0.5493782	70.53	1.345751 0.5418756	82.68
40	s.i. final	1.156737 0.5854702	101.19	1.28266 0.5980907	97.71
50	s.i. final	1.158035 0.642696	131.48	1.391086 0.6591961	118.82
60	s.i. final	1.179675 0.7090519	166.12	1.541101 0.7165073	173.54
70	s.i. final	1.198743 0.7339827	210.73	1.391854 0.7206392	189.65
80	s.i. final	1.193922 0.780195	224.89	1.430581 0.7891698	202.47
90	s.i. final	1.22539 0.8259129	250.6	1.461618 0.8488458	250.88
100	s.i. final	1.241213 0.8539957	288.47	1.473208 0.8517988	303.85
promedio		0.64903535	148.81	0.65639042	207.773

Tabla 5.3: Construcciones GRASP

Podemos notar que, en las instancias n= 30, 70 y 100, la nueva construcción proporcionó mejores resultados, siendo más notorio en n=70. En adición, si comparamos la Tabla 5.3 con la Tabla 5.1, GRASP con la construcción por valor nos dio mejor resultado, incluso que Recocido Simulado, en n=30, por 0.0042903 unidades y en tiempos mucho menores. Sin embargo, para n=40 y 50, nos dio incluso peor que Búsqueda Local.

Además, en la Tabla 5.3 vemos que, en promedio, la construcción por cardinalidad proporcionó mejores resultados y en un menor tiempo.

En adición a esto, es de mencionar que la idea de GRASP es construir una solución y no generar una aleatoria, con el propósito de que se parta de una mejor solución. De esta manera, y comparando las soluciones iniciales de ambas construcciones, podemos concluir que fue mejor la de cardinalidad. Además, esta última fue la segunda metaheurística que dio mejores resultados, solo después del Recocido Simulado. De esta manera, buscando que el tiempo de ejecución disminuyera y que se obtuviera una mejor solución, utilicé la

solución obtenida en GRASP (construcción por cardinalidad) como solución inicial del Recocido Simulado. Los resultados obtenidos se presentan a continuación, en la Tabla 5.4.

n	Búsqueda local	GRASP	Recocido Simulado	Búsqueda tabú	Algoritmos genéticos	Recocido + GRASP
10	0.3592121	0.3329987	0.3321368	0.3551351	0.3581568	0.3218657
20	0.478011	0.4766722	0.4689278	0.4691602	0.5130891	0.4367498
30	0.5595677	0.5493782	0.5461659	0.5397116	0.5473284	0.523253
40	0.5966069	0.5854702	0.5737528	0.6022647	0.5876867	0.5590568
50	0.6584125	0.642696	0.6440078	0.6507829	0.63288	0.6321369
60	0.7265376	0.7090519	0.6904921	0.7070359	0.6989356	0.6881568
70	0.7810394	0.7339827	0.72063	0.7694053	0.7686107	0.7187456
80	0.8106075	0.780195	0.7740953	0.7929619	0.7951683	0.7737279
90	0.8503413	0.8259129	0.796024	0.8475757	0.832626	0.7930826
100	0.9001805	0.8539957	0.8398154	0.8885961	0.8547397	0.8204372
promedio	0.67205165	0.64903535	0.63860479	0.66226294	0.65892213	0.62672123

Tabla 5.4: Heurísticas incluyendo Recocido + GRASP

Los radios obtenidos mediante esta combinación de heurísticas fueron los más pequeños globalmente. No hubo cambios muy grandes con respecto al Recocido Simulado en los casos en que n=80 y n=90, pero en n=20 y n= 40 los radios disminuyeron en buena medida. Tomé de igual forma los tiempos de ejecución, pues también buscaba disminuirlos:

Número de esferas	Búsqueda Local	GRASP	Recocido Simulado	Búsqueda Tabú	Algoritmos genéticos	Recocido + GRASP
10	9.99	11.73	389.07	7.71	10.17	169.93
20	26.01	32.36	467.14	12.10	13.36	475.42
30	66.62	70.53	1534.64	17.77	15.91	796.83
40	95.21	101.19	3142.25	45.72	27.13	1565.11
50	112.61	131.48	3886.11	129.07	39.2	1895.48
60	143.97	166.12	5386.51	158.01	57.11	2926.33
70	188.12	210.73	7572.84	197.94	69.36	3500.61
80	203.1	224.89	8612.27	218.15	80.18	6237.37
90	221.88	250.6	12867.06	247.77	97.14	7691.2
100	245.43	288.47	14647.21	263.52	118.11	10401.49
mean	131.294	148.81	5850.51	129.776	52.767	3565.977
Los tiempos de ejecución están en segundos						

Tabla 5.5: Tiempos de ejecución incluyendo Recocido + GRASP

Si bien la ejecución de Recocido Simulado + GRASP fue menor que la del Recocido Simulado por 2,284.533 segundos, esperaba que fuera mucho más pequeña, pues la heurística empezaba con una solución inicial buena, a diferencia del *Recocido simulado*, que iniciaba con una aleatoria mucho más lejana a la solución final obtenida.

Ahora veremos cómo fueron los resultados obtenidos en relación con los mejores conocidos hasta el momento.
Mhan, H. y Labib, Y. (2018), usan los siguientes parámetros para sus implementaciones computacionales:

- $r_i = 1$
- Tiempo máximo = 7200 segundos.
- n = 10, 20, 30, 40 y 50.

A su vez, [38] compara sus resultados con [5], quienes utilizan los parámetros que se muestran a continuación:

- $r_i = 1$
- Tiempo máximo = 5400 segundos
- n= 10, 20, 30, 40, 50, 60, 70, 80, 90, 100.

En la Tabla 5.6, presento los radios R entre los resultados obtenidos en [38] y [5] y los ya mencionados en este documento de Recocido Simulado + GRASP, pues fue el método que funcionó mejor. Considerando que se tomó $r_i = 0.1$, se multiplicó R por 10 para compararlos.

n	Mhan, H. y Labib, Y.	Birgin, E. y Sobral, F.	Recocido + GRASP
10	2.83246456105391	2.832416745	3.218657
20	3.47353896224525	3.473363343	4.367498
30	3.91649168869767	3.916364072	5.23253
40	4.25533314299430	4.255187329	5.590568
50	4.55095440529171	4.55041577	6.321369
60	-	4.774065618	6.881568
70	-	5.032882129	7.187456
80	-	5.275231771	7.737279
90	-	5.477182604	7.930826
100	-	5.666101817	8.204372

Tabla 5.6: Comparaciones con resultados base

Notemos que los resultados expuestos en [38] y [5] son muy cercanos entre sí. Para determinar qué tanta diferencia hay entre los resultados base y los míos, utilicé la variación porcentual (Tabla 5.6), la cual se calculó como $\frac{V_{rsgrasp}-V_{base}}{V_{base}} * 100$.

Mis resultados estuvieron bastante alejados de los mejores conocidos hasta el momento, particularmente para n=80. Si bien ninguna supera el 50 %, todas rebasan el 10 %.

Recocido + GRASP contra		
n	Mhan, H. y Labib, Y.	Birgin, E. y Sobral, F.
10	13.63450206 %	13.63642041 %
20	25.73626055 %	25.742618 %
30	33.60247936 %	33.60683285 %
40	31.37791595 %	31.38241794 %
50	38.9020508 %	38.91849271 %
60	-	44.14481389 %
70	-	42.80994103 %
80	-	46.67183048 %
90	-	44.797546 %
100	-	44.79746861 %

Tabla 5.7: Variación porcentual entre Recocido + GRASP y resultados base

En cuanto a los tiempos de ejecución, [5] no presenta datos, sin embargo, sabemos que definió como tiempo máximo 5400 segundos, por lo que, de acuerdo a la Tabla 5.5, *Recocido + GRASP* hubiera superado ese límite en n = 80, 90 y 100. Respecto a [38], sabemos que se definió un límite de 7200 segundos, que es rebasado también por las instancias 90 y 100 del método que yo utilicé. Sin embargo, Mhan, H. y Labib, Y. (2018) exponen los tiempos de ejecución de su programa (Tabla 5.7), los cuales fueron bastante mayores que los que se obtuvieron en este trabajo.

n	Mhan, H. y Labib, Y.	Recocido + GRASP
10	1290.2	169.93
20	7000.69	475.42
30	6661.05	796.83
40	6588.66	1565.11
50	6346.64	1895.48

Tabla 5.8: Tiempos de ejecución [38] y Recocido + GRASP

Capítulo 6

Conclusiones

A lo largo de este documento se trabajó con un problema *NP-duro*, por lo que desde un inicio se sabía que, a pesar de utilizar metaheurísticas, nada aseguraba que se iba a llegar al mínimo global. Sin embargo, notamos que los métodos utilizados para la resolución del problema se comportaron de acuerdo a la teoría, pues la *Búsqueda Local* fue la heurística que se aproximó menos a los mejores resultados conocidos, mientras que el resto de éstas devolvieron valores para R similares entre sí.

Los valores obtenidos mediante los métodos que yo utilicé fueron alejados de aquellos que se encuentran en la literatura pues para $n = 10$ obtuve un radio de 3.218657 unidades, mientras que en las fuentes consultadas se reportaron radios de 2.83246456105391 y 2.832416745 unidades. Las diferencias pudieron deberse a que en los artículos contra los que comparé se utilizan modificaciones de las heurísticas mencionadas en el documento y probablemente se utilizan ordenadores más potentes que el que yo tengo.

Hay diversos factores a considerar cuando se trata de determinar cuál fue el mejor método para este problema; los dos más importantes son el tiempo y las soluciones obtenidas. En este caso, el *Recocido Simulado* fue el que proporcionó los valores para R_0 más cercanos a los ya conocidos en casi todas las instancias, sin embargo, fue el método que tomó más tiempo en arrojarlos, tardando más de tres horas para n =90 y n=100. La heurística de *Algoritmos Genéticos* fue la más rápida en general, encontrando los resultados en un tiempo promedio de 52.767 segundos, mientras que a la *Búsqueda Tabú* le tomó 129.776 segundos (en promedio). En cuanto a resultados, de los 10 radios calculados, 6 fueron mejores en *Agloritmos Genéticos* que en *Búsqueda Tabú*.

GRASP con la constucción de cardinalidad dio resultados cercanos a Recocido Simulado en casi todas las instancias, arrojando uno incluso mejor que éste en $n = 50$ y en un máximo de 2.4801 minutos. Por otro lado, *GRASP* con la construcción por valor proporcionó mejores resultados que el de cardinalidad en las instancias n= 30, 70 y 100, pero, para n=40 y 50, nos dio incluso peor que Búsqueda Local.

Si bien los radios más pequeños se obtuvieron cuando combiné las heurísticas de *GRASP* por cardinalidad con *Recocido Simulado*, los tiempos fueron similares a los del *Recocido Simulado*, siendo altos en comparación con las otras heurísticas.

De acuerdo a lo mencionado anteriormente y a los resultados expuestos en el documento, considero que la heurística que mejor funciona para este problema en particular es *GRASP* con construcción por cardinalidad, pues nos otorgó buenos resultados en un tiempo promedio 23.96328876 veces menor que la combinación de *Recocido Simulado* + *GRASP*, que fue la que dio los resultados más cercanos a los teóricos.

Bibliografía

[1] Aarts, E.H.L. y Korst, J. (1989). *Simulated annealing and Boltzmann machines: a stochastic approach to combinatorial optimization and neural computing*. Wiley, Nueva York.

[2] Aarts, E. y Lenstra, J. (2018). *Local Search in Combinatorial Optimization*. Princeton University. Estados Unidos.

[3] Arab Asadi, A., Naserasadi, A. y Arab Asadi, Z. (2011). A New Hybrid Algorithm for Traveler Salesman Problem based on Genetic Algorithms and Artificial Neural Networks. *International Journal of Computer Applications (0975 – 8887) Volume 24– No.5*. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.259.3099rep=rep1type=pdf

[4] Arranz de la Peña, J., Parra Truyol, A. (2007). Algoritmos genéticos. *Universidad Carlos III*.

[5] Birgin, E. y Sobral, F. (2008). Minimizing the object dimensions in circle and sphere packing problems. *Computers Operations Research*(35). 2357-2375. https://www.ime.usp.br/ egbirgin/publications/bs.pdf

[6] Bravo Trinidad, José (s.f.), *Diagrama de Voronoi*. http://matematicas.unex.es/ trinidad/mui/voronoi.pdf

[7] Caballero-Villalobos J., Alvarado-Valencia J. (2010). Greedy Randomized Adaptive Search Procedure (GRASP), una alternativa valiosa en la minimización de la tardanza total ponderada en una máquina. *Ingeniería y Universidad*. 14(2). https://biblat.unam.mx/hevila/Ingenieriayuniversidad/2010/vol14/no2/4.pdf

[8] Carr, Jenna. (2014). An Introduction to Genetic Algorithms. *Smart Educational Tecgnology Mathematics Education Laboratory.* http://www.javamath.com/snucode/lecture.pdf

[9] Christos H. Papadimitriou, Kenneth Steiglitz (1998). *Combinatorial Optimization: Algorithms and Complexity.* Courier Corporation.

[10] Cook, S. (1971). The complexity of theorem-proving procedures. *3rd Annual ACM Symposium on Theory of Computing.*ACM Press. 1971

[11] Cortéz, Augusto (2004). Teoría de la complejidad computacional y teoría de la computabilidad. *Revista investigación sist. Inform.* 102-105.https://revistasinvestigacion.unmsm.edu.pe/index.php/sistem/article/view/3216

[12] CRUZ, Marco; MORENO, Pedro y PERALTA, Jesús. (2014). Aplicación de la teoría de la complejidad en optimización combinatoria. *Inventio*, vol. 10, no. 20, p. 35-42.

[13] Domínguez Moreno, Lilian (2016). La conjetura de Kepler. *Revista de divulgación del CINESTAV*, Espacio Abierto. 1(3).

[14] Dowsland, Kathryn, Adenso Díaz, Belarmino (2003). *Heuristic design and fundamentals of the Simulated Annealing.* AEPIA.

[15] Feo, Thomas Resende, Mauricio. (1989). *A probabilistic heuristic for a computationally difficult set covering problem.* . Operations Research Letters.

[16] Fırat, H. y Alpaslan, N. (2020). An effective approach to the two-dimensional rectangular packing problem in the manufacturing industry. *Computers Industrial Engineering* Volume 148. https://doi.org/10.1016/j.cie.2020.106687.

[17] Flores Cabezas, Xavier (2014), Problemas del Milenio: P vs NP, *Revista de Divulgación Amarun*, 1, pp. 1-15. http://www.amarun.net/phocadownload/userupload/Diego_Chamorro/Rev_Div_Amarun_1_2014_PvsNP.pdf

[18] Fortnow, Lance (2013). *The Golden Ticket, P, NP, and the search for the impossible.* Princeton.

[19] Fraser, H. y George, J. (1994). Integrated container loading software for pulp and paper industry. *European Journal of Operations Research 77.*

[20] Glover, F. y Laguna, M. (1997). *Tabu Search.* Kluwer Academic Publishers.

[21] Glover, F. y Melián, B. (2006). *Introducción a la Búsqueda Tabú.* Vol.3. https://www.researchgate.net/profile/BelenMelian/publication/238691042Introduccion$_l$aBusquedaTab

[22] Glover, F. y Melián, B. (2003). Tabu Search. *Revista Iberoamericana de Inteligencia Artificial* (19). 29-48.https://www.researchgate.net/profile/Fred_Glover/publication/28076117 _Busqueda_Tabu/links/0fcfd50e607e93a116000000/Busqueda-Tabu.pdf

[23] Hao, Peng y Wang, Ziran y Wu, Guoyuan y Boriboonsomsin, Kanok y Barth, Matthew. (2017). *Intra-Platoon Vehicle Sequence Optimization for Eco-Cooperative Adaptive Cruise Control.* https://www.researchgate.net/publication/320508257Intra-PlatoonVehicleSequenceOptimizationforEco

[24] Hillier, F. y Lieberman, G. (2010) *Introducción a la Investigación de Operaciones.* McGraw Hill.

[25] Holland J.H. (1984) Genetic Algorithms and Adaptation. In: Selfridge O.G., Rissland E.L., Arbib M.A. (eds) Adaptive Control of Ill-Defined Systems. NATO Conference Series (II Systems Science), vol 16. Springer, Boston, MA. https://doi.org/10.1007/978-1-4684-8941-521

[26] Karp, Richard (1972). Reducibility Among Combinatorial Problems. *Complexity of Computer Computations.* Universidad de California.

[27] Kirkpatrick, S. y Gelatt, D. (1983). Optimization by Simmulated Annealing. *Science*, Volume 220, Number 4598.

[28] Korte, B. y Vygen, J. (2000) *Combinatorial Optimization. Theory and Algorithms.* Springer.

[29] LEHMANN, C. H. (1993). *Geometría Analítica* LIMUSA/NORIEGA EDITORES.

[30] Liu, J., Huang, W., Liu, W., Song, B., Sun, Y., Chen, M. (2014). Energy landscape paving with local search for global optimization of the BLN off-lattice model. *Korean Physical Society.* 64. https://www.researchgate.net/publication/263020602_Energy_landscape_paving_with_local_search_for_global_optimization_of_the_BLN_off-lattice_model

[31] Liu J., Yao Y., Zheng Y., Geng H., Zhou G. (2009). An Effective Hybrid Algorithm for the Circles and Spheres Packing Problems. *Combinatorial Optimization and Applications*, vol 5573, Springer, Berlin, Heidelberg. https://www.researchgate.net/publication/221335361_An_Effective_Hybrid_Algorithm_for_the_Circles_and_Spheres_Packing_Problems

[32] Lochman, k., Oger, L. y Stoyan, D. (2006). Statistical analisis of random sphere packings with variable radius distribution. *Solod State Sciences*, 8(12). 1397-1413. https://www.sciencedirect.com/science/article/abs/pii/S1293255806002238

[33] Lozano, José. [José A. Lozano]. (2015). *Busqueda Tabu.* Youtube. https://www.youtube.com/watch?v=NGo9P33aEV8

[34] Maldonado, Carlos (2013), Un problema fundamental en la investigación: Los problemas P vs. NP, *Revista Logos, Ciencia Tecnologóa*, vol. 4, núm. 2, pp. 10-20. https://www.redalyc.org/pdf/5177/517751544002.pdf

[35] Marmolejo-Correa, D., Juarez-Valdivia, R. y Rodriguez-Navarro, A. (2016). *Optimization of Preventive Maintenance Program for Imaging Equipment in Hospitals.* Computer Aided Chemical Engineering. https://reader.elsevier.com/reader/sd/pii/B9780444634283503106?token=C0E587670B82F1F75A92FB5D0ED78CCF4C9EB0753CB8B6F4809D31BCC0A96E2E0E6CACC926831F8A1FE1657ACD657A62

[36] Melián, Belén. Pérez, José A. (2003) Metaheurísticas: una visión global. *Inteligencia Artificial, Revista Iberoamericana de Inteligencia Artificial.* N.19. http://www.aepia.org/revista.

[37] M'Hallah, R., Alkandari, A., y Mladenovic´, N. (2013). Packing unit spheres into the smallest sphere using VNS and NLP. *Computers Operations Research*, 40(2), 603-615. https://www.sciencedirect.com/science/article/abs/pii/S030505481200192X

[38] Mhan, H. y Labib, Y. (2018). A hybrid algorithm for packing identical spheres into a container. *Expert Systems with Applications*. https:// www.sciencedirect.com/science/article/pii/S0957417417308084

[39] Mingos, D. M. P., Rohl, A. L. (1991). Size and shape characteristics of inorganic molecules and ions and their relevance to molecular packing problems. *Journal of the Chemical Society, Dalton Transactions*, (12), 3419-3425.

[40] Morales Manzanares, Eduardo (2004). Búsqueda Local (Local Search). INAOE. Obtenido de https://ccc.inaoep.mx/ emorales/Cursos/Busqueda/node58.html

[41] Pérez, Y., Pérez, I., Recarey, C. y Cerrolaza, M. (2010). Un nuevo método de empaquetamiento de partículas: aplicaciones en el modelado del tejido óseo. *Revista Internacional de Métodos Numéricos para Cálculo y Diseño en Ingeniería* 26.

[42] Poon, P. W., Carter, J. N. (1995). Genetic algorithm crossover operators for ordering applications. *Computers Operations Research,* 22(1), 135-147.

[43] Quirós Garcián, A (2000), ¿2000? La conjetura de Kepler. *Números*, ISSN 0212-3096 (43-44). http://www.sinewton.org/numeros/numeros/43-44/Articulo100.pdf

[44] Resende, M y González, J. (2003), GRASP: Procedimientos de busqueda miopes aleatorizados y adaptativos. *Revista Iberoamericana de Inteligencia Artificial*, No.19. http://www.scielo.org.co/scielo.php?script=sci_arttextpid=S0123-21262010000200004

[45] Resende, M. y Ribeiro, C.(2016). *Optimization by GRASP: Greedy Randomized Adaptive Search Procedures*. Springer.

[46] Riojas Cañari, Alicia. (2005). Conceptos, algoritmo y aplicación al problema de las N – reinas. Universidad Nacional Mayor de San Marcos. https://sisbib.unmsm.edu.pe/bibvirtualdata/monografias/basic/riojasca/cap3.pdf

[47] Rudich, S. y Wigderson, A. (2004). *Computational Complexity Theory*. American Mathematical Society

[48] Soontrapa, K. y Chen, Y. (2013). Mono-sized sphere packing algorithm development using optimized Monte-Carlo technique. *Advanced Powder Technology*, 24(6), 955-961. https://www.sciencedirect.com/science/article/abs/pii/S0921883113000101

[49] Sutou, A. y Dai, Y. (2002). Global Optimization approach to unequal sphere packing problem in 3D. *Journal of Optimization Theory and Applications*, 114(3), 671-694. https://www.researchgate.net/publication/225118158_Global_Optimization_Approach_to_Unequal_Sphere_Packing_Problems_in_3D

[50] Stoyan, Y., Yaskow, G. y Scheithauer, G. (2003), Packing of various radii solid spheres into a parallelepiped. *Central European Journal of Operational Research*, 11, 389-407. https://www.researchgate.net/publication/266985052_Packing_of_Various_Radii_Solid_Spheres_Into_a_Parallelepiped

[51] Stoyan G., Scheithauer, G y Yaskov, N (2016). Packing Unequal Spheres into Various Containers. *Cybernetics and Systems Analysis*, 52, 419–426. https://www.researchgate.net/publication/303558087_Packing_Unequal_Spheres_into_Various_Containers

[52] Talbi, E. G. (2009). *Metaheuristics: from desgin to implmentation*. John Wiley Sons.

[53] Turing, A. M. (1936). On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society Series 2*.

[54] Wells, A. (1978). *Química inorgánica estructural*. Editorial Reverté. pp. 129

[55] Xhafa, Fatos y Abraham, Ajith. (2008). *Metaheuristics for Scheduling in Distributed Computing Environments*. Springer.

Apéndice

Código utilizado en este trabajo

Búsqueda Local

```
library("rgl")
t <- proc.time()

numcir = 10

set.seed(11)
solin <- function(n,r=0.1){
  #Generar los centros de manera uniforme
  ran <- runif(n = 3*n,min = -1,max = 1)
  #Construir matriz asociada con la soluci n
  s0 <- matrix(ran, ncol = 3, dimnames = list(1:n,c("x","y","z")))
  #Agregar los radios en la matriz
  s0 <- as.matrix(cbind(s0,r))
  #Regresar soluci n
  s0
}

s <- solin(numcir)
si <- s

R <- function(s){
  max(sapply(1:nrow(s),function(i) norm(s[i,1:3], type = "2") + s[i,4]))
}

R(si)

Nx <- function(s){
  #Ver si es factible la nueva soluci n
  fi <- 0
  #N mero de iteraciones al momento
  n <- 1
  #Vector de coordenadas x asociadas con las esferas
  x <- s[,1]
  #Esfera con coordenada x m s alejada del or ?gen
  xm <- which.max(abs(x))[1]
  #Esferas diferentes a xm
  otros <- (1:nrow(s))[-xm]
  #Mientras que no sea factible genera nueva soluci n vecina
  while(fi < 1 && n <= 100){
    #Hacer una copia de la soluci n
    ns <- s
    #Genero aleatorios
    u <- runif(n = 3,min = -.25,max = .25)
    #Cambio la esfera asociada con xm por aleatorios
    ns[xm,1:3] <- u
    #Verificar factibilidad de la nueva soluci n
    fi <- 1
    for(i in 1:length(otros)){
      #Distancia de xm al c ?rculo otros[i]
      distancia <- sqrt(sum((ns[xm,-4] - ns[otros[i],-4])^2))
```

```
      #Verificar si hay tralape
      fi <- fi*(distancia > ns[xm,4] + ns[otros[i],4])
      #Dejar de ver si es factible en el primer traslape
      if(fi == 0) break
    }
    n <- n + 1
  }
  #Regresar la nueva soluci n en caso que sea factible
  if(fi > 0){return(ns)} else {return(s)}
}


Ny <- function(s){
  #Ver si es factible la nueva soluci n
  fi <- 0
  #N mero de iteraciones al momento
  n <- 1
  #Vector de coordenadas y asociadas con las esferas
  y <- s[,2]
  #Esfera con coordenada y m s alejada del or ?gen
  ym <- which.max(abs(y))[1]
  #Esferas diferentes a xm
  otros <- (1:nrow(s))[-ym]
  #Mientras que no sea factible genera nueva soluci n vecina
  while(fi < 1 && n <= 100){
    #Hacer una copia de la soluci n
    ns <- s
    #Genero aleatorios
    u <- runif(n = 3,min = -.25,max = .25)
    #Traslado la esfera asociada con ym.
    ns[ym,1:3] <- u
    #Verificar factibilidad de la nueva soluci n
    fi <- 1
    for(i in 1:length(otros)){
      #Distancia de xm al c ?rculo otros[i]
      distancia <- sqrt(sum((ns[ym,-4] - ns[otros[i],-4])^2))
      #Verificar si hay tralape
      fi <- fi*(distancia > ns[ym,4] + ns[otros[i],4])
      #Dejar de ver si es factible en el primer traslape
      if(fi == 0) break
    }
    n <- n + 1
  }
  #Regresar la nueva soluci n en caso que sea factible
  if(fi > 0){return(ns)} else {return(s)}
}

Nz <- function(s){
  #Ver si es factible la nueva soluci n
  fi <- 0
  #N mero de iteraciones al momento
  n <- 1
  #Vector de coordenadas z asociadas con las esferas
  z <- s[,3]
  #Esfera con coordenada z m s alejada del or ?gen
  zm <- which.max(abs(z))[1]
  #Esferas diferentes a xm
  otros <- (1:nrow(s))[-zm]
  #Mientras que no sea factible genera nueva soluci n vecina
  while(fi < 1 && n <= 100){
    #Hacer una copia de la soluci n
    ns <- s
    #Genero aleatorios
    u <- runif(n = 3,min = -.25,max = .25)
    #Traslado la esfera asociada con xm.
    ns[zm,1:3] <- u
    #Verificar factibilidad de la nueva soluci n
    fi <- 1
```

```
    for(i in 1:length(otros)){
      #Distancia de xm al c ?rculo otros[i]
      distancia <- sqrt(sum((ns[zm,-4] - ns[otros[i],-4])^2))
      #Verificar si hay tralape
      fi <- fi*(distancia > ns[zm,4] + ns[otros[i],4])
      #Dejar de ver si es factible en el primer traslape
      if(fi == 0) break
    }
    n <- n + 1
  }
  #Regresar la nueva soluci n en caso que sea factible
  if(fi > 0){return(ns)} else {return(s)}
}

#Bsqueda local
bl <- s
for (i in 1:5000){
  fi <- 0
  #N mero de iteraciones al momento
  n <- 1
  #Vector de coordenadas x asociadas con las esferas
  x <- bl[,1]
  #Vector de coordenadas y asociadas con las esferas
  y <- bl[,2]
  #Vector de coordenadas z asociadas con las esferas
  z <- bl[,3]
  #Vector de esferas con coordenadas m s alejadas del or ?gen
  d <- c(max(abs(x)),max(abs(y)),max(abs(z)))
  #Generar nueva soluci n dependiendo de la coordenada m s lejana
  if (max(d) == d[1]){
    bl <<- Nx(bl)
  } else if (max(d) == d[2]) {
    bl <<- Ny(bl)
  } else {
    bl <<- Nz(bl)
  }
}
#Regresar nueva soluci n
open3d()
spheres3d(x = si[,1],
          y = si[,2],
          z = si[,3],
          radius = 0.1,
          color = rainbow(ncol(si)))
spheres3d(x = 0,y = 0,z = 0,radius = R(si), col="white", alpha=0.5)
axes3d()


open3d()
spheres3d(x = bl[,1],
          y = bl[,2],
          z = bl[,3],
          radius = 0.1,
          color = rainbow(nrow(bl)))
spheres3d(x = 0,y = 0,z = 0,radius = R(bl), col="white", alpha=0.5)
axes3d()

distancias <- function(si,sj){
  sqrt(sum((si-sj)^2))
}
#Matriz de distancias, nxn sin importar el nmero de esferas acomodadas por columnas
mat_distancia<-function(solucion){
  mat_dist<-c()
  for (vj1 in 1:(nrow(solucion))) { #renglones
    reng_dist<-c()
    for (vj2 in 1:(nrow(solucion))) { #columnas
      reng_dist[vj2]<-distancias(solucion[vj1,],solucion[vj2,])
    }
```

```
    mat_dist<-rbind(mat_dist,reng_dist)

  }
  return(mat_dist)
}
distancias_si<-mat_distancia(si)
distancias_bl<- mat_distancia(bl)

#Matriz de penalidades

mat_penalidades<-function(solucion,mdistancias){
  mat_penal<-c()
  for (vj1 in 1:(nrow(solucion))) { #renglones
    reng_penal<-c()
    for (vj2 in 1:(nrow(solucion))) { #columnas
      if(mdistancias[vj1,vj2]<2*solucion[1,4]){
        reng_penal[vj2]<-(2*solucion[1,4])-mdistancias[vj1,vj2]
      }
      else {
        reng_penal[vj2]<-0
      }

    }
    mat_penal<-rbind(mat_penal,reng_penal)

  }
  return(mat_penal)
}
penalidades_si<-mat_penalidades(si,distancias_si)
penalidades_bl <- mat_penalidades(bl,distancias_bl)
rownames(penalidades_bl) <- c(1:numcir)
R(bl)
proc.time() - t
```

GRASP cardinalidad

```
library("rgl")
t <- proc.time() #Para medir el tiempo
numcir <- 10
numcent = numcir
set.seed(11) #Fijamos semilla
#Creamos la funcin  que genera el radio grande
R <- function(s){
  max(sapply(1:nrow(s),function(i) norm(s[i,1:3], type = "2") + s[i,4])) #Toma la norma de la esfera que es
}

#Funcin  para calcular distancias entre las esferas
distancias <- function(si,sj){
  sqrt(sum((si-sj)^2))
}

penas <- function(s){
  #Nmero de esferas
  n <- nrow(s)
  #Nmero inicial de traslapes
  Penas <- 0
  #Son necesarios al menos dos  c rculos  para que haya un traslape
  if(n >= 2){
    #Comparar todos los pares distintos de esferas
    for (i in 1:(n-1)) {
      for (j in (i+1):n) {
        #Distancia entre las esferas
        distancia <- sqrt(sum((s[i,1:3]-s[j,1:3])^2))
        Penas <- Penas + (s[i,4] + s[j,4] - distancia)*(distancia < s[i,4] + s[j,4])
      }
    }
  }
  #Regresar medida de traslapamiento
  Penas
```

```
}

#Funcin  costo miope
cmiope <- function(s){
  R(s) + penas(s)
}

#Construccin
card <- function(numcir,numcent,r,k){
  M <- matrix(NA, nrow = numcir, ncol = 4) #Matriz vaca
  candidatos <- 1:numcent #Habra la misma cantidad de candidatos que de  c rculos
  ran <- runif(n = 3*numcent,min = -1,max = 1) #Creo centros aleatorios
  s0 <- matrix(ran, ncol = 3) #Solucin  inicial aleatoria
  colnames(s0) = c("x","y","z")
  s0 <- as.matrix(cbind(s0,r)) #Pegar los radios
  s0costosmiopes <- sapply(1:numcent,function(x) R(s0[x,,drop=F])) #Calculo los costos miopes de la solucin
  cota <- sort(s0costosmiopes, decreasing = F)[min(k,length(s0costosmiopes))] #Los ordeno de acuerdo a sus c
  #y saco el mnimo entre un nmero aleatorio y la cantidad de esferas de la s.i.
  #Como cota pongo el valor el costo miope del k- imo  elemento
  RCL <- candidatos[which(s0costosmiopes <= cota)] #Vemos qu  costos miopes de los candidatos son menores a
  elegido <- sample(RCL, 1) #Elegimos uno de esos
  posicion <- which(candidatos == elegido) #Con su posicin
  M[1,] <- s0[posicion,] #Encontramos sus coordenadas en la solucion inicial
  n <- 2 #Nmero de iteraciones
  while(n <= numcir){ #Mientras el nmero e iteraciones sea menor al numero de  c rculos
    ran <- runif(n = 3*numcent,min = -1,max = 1)
    s0 <- matrix(ran, ncol = 3) #Creo solucin  inicial
    colnames(s0) <- c("x","y","z")
    s0 <- as.matrix(cbind(s0,r)) #Pegar los radios
    costosmiopes <- sapply(1:numcent, function(x) cmiope(rbind(M[1:(n-1),],s0[x,,drop = F]))) #Calculo los c
    cota <- sort(costosmiopes, decreasing = F)[min(k,length(costosmiopes))] #La cota
    RCL <- candidatos[which(costosmiopes <= cota)] #Hago la lista de candidatos
    elegido <- sample(RCL, 1) #Obtengo a un elemento e la lista
    posicion <- which(candidatos == elegido)  #Encuentro su posicion
    M[n,] <- s0[posicion,] #Agrego sus coordenadas
    n <- n+1
  }
  return(M)

}
M <- card(10,10,0.1,4)

open3d()
spheres3d(x = M[,1],
          y = M[,2],
          z = M[,3],
          radius = 0.1,
          color = rainbow(ncol(M)))
spheres3d(x = 0,y = 0,z = 0,radius = R(M), col="white", alpha=0.5)
axes3d()


Nx <- function(s){
  #Ver si es factible la nueva soluci n
  fi <- 0
  #N mero de iteraciones al momento
  n <- 1
  #Vector de coordenadas x asociadas con las esferas
  x <- s[,1]
  #Esfera con coordenada x m s alejada del or ?gen
  xm <- which.max(abs(x))[1]
  #Esferas diferentes a xm
  otros <- (1:nrow(s))[-xm]
  #Mientras que no sea factible genera nueva soluci n vecina
  while(fi < 1 && n <= 100){
    #Hacer una copia de la soluci n
    ns <- s
    #Genero aleatorios
```

```
    u <- runif(n = 3,min = -.2,max = .2)
    #Traslado la esfera asociada con xm.
    ns[xm,1:3] <- u
    #Verificar factibilidad de la nueva soluci n
    fi <- 1
    for(i in 1:length(otros)){
      #Distancia de xm al c ?rculo otros[i]
      distancia <- sqrt(sum((ns[xm,-4] - ns[otros[i],-4])^2))
      #Verificar si hay tralape
      fi <- fi*(distancia > ns[xm,4] + ns[otros[i],4])
      #Dejar de ver si es factible en el primer traslape
      if(fi == 0) break
    }
    n <- n + 1
  }
  #Regresar la nueva soluci n en caso que sea factible
  if(fi > 0){return(ns)} else {return(s)}
}

Ny <- function(s){
  #Ver si es factible la nueva soluci n
  fi <- 0
  #N mero de iteraciones al momento
  n <- 1
  #Vector de coordenadas y asociadas con las esferas
  y <- s[,2]
  #Esfera con coordenada y m s alejada del or ?gen
  ym <- which.max(abs(y))[1]
  #Esferas diferentes a xm
  otros <- (1:nrow(s))[-ym]
  #Mientras que no sea factible genera nueva soluci n vecina
  while(fi < 1 && n <= 100){
    #Hacer una copia de la soluci n
    ns <- s
    #Genero aleatorios
    u <- runif(n = 3,min = -.2,max = .2)
    #Traslado la esfera asociada con xm.
    ns[ym,1:3] <- u
    #Verificar factibilidad de la nueva soluci n
    fi <- 1
    for(i in 1:length(otros)){
      #Distancia de xm al c ?rculo otros[i]
      distancia <- sqrt(sum((ns[ym,-4] - ns[otros[i],-4])^2))
      #Verificar si hay tralape
      fi <- fi*(distancia > ns[ym,4] + ns[otros[i],4])
      #Dejar de ver si es factible en el primer traslape
      if(fi == 0) break
    }
    n <- n + 1
  }
  #Regresar la nueva soluci n en caso que sea factible
  if(fi > 0){return(ns)} else {return(s)}
}

Nz <- function(s){
  #Ver si es factible la nueva soluci n
  fi <- 0
  #N mero de iteraciones al momento
  n <- 1
  #Vector de coordenadas z asociadas con las esferas
  z <- s[,3]
  #Esfera con coordenada z m s alejada del or ?gen
  zm <- which.max(abs(z))[1]
  #Esferas diferentes a xm
  otros <- (1:nrow(s))[-zm]
  #Mientras que no sea factible genera nueva soluci n vecina
  while(fi < 1 && n <= 100){
    #Hacer una copia de la soluci n
```

```
    ns <- s
    #Genero aleatorios
    u <- runif(n = 3,min = -.2,max = .2)
    #Traslado la esfera asociada con xm.
    ns[zm,1:3] <- u
    #Verificar factibilidad de la nueva soluci n
    fi <- 1
    for(i in 1:length(otros)){
      #Distancia de xm al c ?rculo otros[i]
      distancia <- sqrt(sum((ns[zm,-4] - ns[otros[i],-4])^2))
      #Verificar si hay tralape
      fi <- fi*(distancia > ns[zm,4] + ns[otros[i],4])
      #Dejar de ver si es factible en el primer traslape
      if(fi == 0) break
    }
    n <- n + 1
  }
  #Regresar la nueva soluci n en caso que sea factible
  if(fi > 0){return(ns)} else {return(s)}
}

#Bsqueda local
bl <- M
for (i in 1:1000){
  fi <- 0
  #N mero de iteraciones al momento
  n <- 1
  #Vector de coordenadas x asociadas con las esferas
  x <- bl[,1]
  #Vector de coordenadas y asociadas con las esferas
  y <- bl[,2]
  #Vector de coordenadas z asociadas con las esferas
  z <- bl[,3]
  #Vector de esferas con coordenadas m s alejadas del or ?gen
  d <- c(max(abs(x)),max(abs(y)),max(abs(z)))
  #Generar nueva soluci n dependiendo de la coordenada m s lejana
  if (max(d) == d[1]){
    bl <<- Nx(bl)
  } else if (max(d) == d[2]) {
    bl <<- Ny(bl)
  } else {
    bl <<- Nz(bl)
  }
}
#Regresar nueva soluci n
grasp1 <- bl

open3d()
spheres3d(x = grasp1[,1],
          y = grasp1[,2],
          z = grasp1[,3],
          radius = 0.1,
          color = rainbow(ncol(grasp1)))
spheres3d(x = 0,y = 0,z = 0,radius = R(grasp1), col="white", alpha=0.5)
axes3d()
grasp1

distancias <- function(si,sj){
  sqrt(sum((si-sj)^2))
}
#Matriz de distancias, nxn sin importar el nmero de esferas acomodadas por columnas
mat_distancia<-function(solucion){
  mat_dist<-c()
  for (vj1 in 1:(nrow(solucion))) { #renglones
    reng_dist<-c()
    for (vj2 in 1:(nrow(solucion))) { #columnas
      reng_dist[vj2]<-distancias(solucion[vj1,],solucion[vj2,])
    }
```

```
    mat_dist<-rbind(mat_dist,reng_dist)

  }
  return(mat_dist)
}
distancias_M<-mat_distancia(M)
distancias_grasp1<- mat_distancia(grasp1)

#Matriz de penalidades
mat_penalidades<-function(solucion,mdistancias){
  mat_penal<-c()
  for (vj1 in 1:(nrow(solucion))) { #renglones
    reng_penal<-c()
    row.names(1:n)
    for (vj2 in 1:(nrow(solucion))) { #columnas
      if(mdistancias[vj1,vj2]<2*solucion[1,4]){
        reng_penal[vj2]<-(2*solucion[1,4])-mdistancias[vj1,vj2]
      }
      else {
        reng_penal[vj2]<-0
      }

    }
    mat_penal<-rbind(mat_penal,reng_penal)

  }
  return(mat_penal)
}
penalidades_M<-mat_penalidades(M,distancias_M)
penalidades_grasp1 <- mat_penalidades(grasp1,distancias_grasp1)
rownames(penalidades_grasp1) <- c(1:numcir)
R(grasp1)
proc.time() - t
```

GRASP valor

```
library("rgl")
t <- proc.time() #Para medir el tiempo
numcir <- 10
numcent = numcir
#Creamos la funcin  que genera el radio grande
R <- function(s){
  max(sapply(1:nrow(s),function(i) norm(s[i,1:3], type = "2") + s[i,4])) #Toma la norma de la esfera que es
}

#Funcin  para calcular distancias entre las esferas
distancias <- function(si,sj){
  sqrt(sum((si-sj)^2))
}

penas <- function(s){
  #Nmero de esferas
  n <- nrow(s)
  #Nmero inicial de traslapes
  Penas <- 0
  #Son necesarios al menos dos  c rculos  para que haya un traslape
  if(n >= 2){
    #Comparar todos los pares distintos de esferas
    for (i in 1:(n-1)) {
      for (j in (i+1):n) {
        #Distancia entre las esferas
        distancia <- sqrt(sum((s[i,1:3]-s[j,1:3])^2))
        Penas <- Penas + (s[i,4] + s[j,4] - distancia)*(distancia < s[i,4] + s[j,4])
      }
    }
  }
  #Regresar medida de traslapamiento
  Penas
}
```

```
#Funcin  costo miope
cmiope <- function(s){
  R(s) + penas(s)
}
#Construccin
valor <- function(numcir,numcent,r,alpha){
  M<- matrix(NA, nrow = numcir, ncol = 4) #Matriz vaca
   candidatos <- 1:numcent #Habra la misma cantidad de candidatos que de  c rculos
   ran <- runif(n = 3*numcent,min = -1,max = 1) #Creo centros aleatorios
   s0 <- matrix(ran, ncol = 3) #Solucin  inicial aleatoria
   colnames(s0) = c("x","y","z")
   s0 <- as.matrix(cbind(s0,0.1)) #Pegar los radios
   s0costosmiopes <- sapply(1:numcent,function(x) R(s0[x,,drop=F])) #Calculo los costos miopes de la solucin
   cota <- alpha*max(s0costosmiopes) + (1-alpha)*min(s0costosmiopes) #Defino la cota
   #y saco el mnimo entre un nmero aleatorio y la cantidad de esferas de la s.i.
   #Como cota pongo el valor el costo miope del k- imo  elemento
   RCL <- candidatos[which(s0costosmiopes <= cota)] #Vemos qu  costos miopes de los candidatos son menores a
   elegido <- sample(RCL, 1) #Elegimos uno de esos
   posicion <- which(candidatos == elegido) #Con su posicin
   M[1,] <- s0[posicion,] #Encontramos sus coordenadas en la solucion inicial
   n <- 2 #Nmero de iteraciones
   while(n <= numcir){ #Mientras el nmero e iteraciones sea menor al numero de  c rculos
     ran <- runif(n = 3*numcent,min = -1,max = 1)
     s0 <- matrix(ran, ncol = 3) #Creo solucin  inicial
     colnames(s0) <- c("x","y","z")
     s0 <- as.matrix(cbind(s0,r)) #Pegar los radios
     costosmiopes <- sapply(1:numcent, function(x) cmiope(rbind(M[1:(n-1),],s0[x,,drop = F]))) #Calculo los c
     cota <- alpha*max(costosmiopes) + (1-alpha)*min(costosmiopes) #Defino la cota
     RCL <- candidatos[which(costosmiopes <= cota)] #Hago la lista de candidatos
     elegido <- sample(RCL, 1) #Obtengo a un elemento e la lista
     posicion <- which(candidatos == elegido)  #Encuentro su posicion
     M[n,] <- s0[posicion,] #Agrego sus coordenadas
     n <- n+1
   }
   return(M)

}
set.seed(11)
M<- valor(10,10,0.1,0.5)

open3d()
spheres3d(x = M[,1],
          y = M[,2],
          z = M[,3],
          radius = 0.1,
          color = rainbow(ncol(M)))
spheres3d(x = 0,y = 0,z = 0,radius = R(M), col="white", alpha=0.5)
axes3d()


Nx <- function(s){
  #Ver si es factible la nueva soluci n
   fi <- 0
  #N mero de iteraciones al momento
  n <- 1
  #Vector de coordenadas x asociadas con las esferas
  x <- s[,1]
  #Esfera con coordenada x m s alejada del or ?gen
  xm <- which.max(abs(x))[1]
  #Esferas diferentes a xm
  otros <- (1:nrow(s))[-xm]
  #Mientras que no sea factible genera nueva soluci n vecina
  while(fi < 1 && n <= 100){
    #Hacer una copia de la soluci n
    ns <- s
    #Genero aleatorios
    u <- runif(n = 3,min = -.2,max = .2)
```

```
    #Traslado la esfera asociada con xm.
    ns[xm,1:3] <- u
    #Verificar factibilidad de la nueva soluci n
    fi <- 1
    for(i in 1:length(otros)){
      #Distancia de xm al c ?rculo otros[i]
      distancia <- sqrt(sum((ns[xm,-4] - ns[otros[i],-4])^2))
      #Verificar si hay tralape
      fi <- fi*(distancia > ns[xm,4] + ns[otros[i],4])
      #Dejar de ver si es factible en el primer traslape
      if(fi == 0) break
    }
    n <- n + 1
  }
  #Regresar la nueva soluci n en caso que sea factible
  if(fi > 0){return(ns)} else {return(s)}
}

Ny <- function(s){
  #Ver si es factible la nueva soluci n
  fi <- 0
  #N mero de iteraciones al momento
  n <- 1
  #Vector de coordenadas y asociadas con las esferas
  y <- s[,2]
  #Esfera con coordenada y m s alejada del or ?gen
  ym <- which.max(abs(y))[1]
  #Esferas diferentes a xm
  otros <- (1:nrow(s))[-ym]
  #Mientras que no sea factible genera nueva soluci n vecina
  while(fi < 1 && n <= 100){
    #Hacer una copia de la soluci n
    ns <- s
    #Genero aleatorios
    u <- runif(n = 3,min = -.2,max = .2)
    #Traslado la esfera asociada con xm.
    ns[ym,1:3] <- u
    #Verificar factibilidad de la nueva soluci n
    fi <- 1
    for(i in 1:length(otros)){
      #Distancia de xm al c ?rculo otros[i]
      distancia <- sqrt(sum((ns[ym,-4] - ns[otros[i],-4])^2))
      #Verificar si hay tralape
      fi <- fi*(distancia > ns[ym,4] + ns[otros[i],4])
      #Dejar de ver si es factible en el primer traslape
      if(fi == 0) break
    }
    n <- n + 1
  }
  #Regresar la nueva soluci n en caso que sea factible
  if(fi > 0){return(ns)} else {return(s)}
}

Nz <- function(s){
  #Ver si es factible la nueva soluci n
  fi <- 0
  #N mero de iteraciones al momento
  n <- 1
  #Vector de coordenadas z asociadas con las esferas
  z <- s[,3]
  #Esfera con coordenada z m s alejada del or ?gen
  zm <- which.max(abs(z))[1]
  #Esferas diferentes a xm
  otros <- (1:nrow(s))[-zm]
  #Mientras que no sea factible genera nueva soluci n vecina
  while(fi < 1 && n <= 100){
    #Hacer una copia de la soluci n
    ns <- s
```

```
    #Genero aleatorios
    u <- runif(n = 3,min = -.2,max = .2)
    #Traslado la esfera asociada con xm.
    ns[zm,1:3] <- u
    #Verificar factibilidad de la nueva soluci n
    fi <- 1
    for(i in 1:length(otros)){
      #Distancia de xm al c ?rculo otros[i]
      distancia <- sqrt(sum((ns[zm,-4] - ns[otros[i],-4])^2))
      #Verificar si hay tralape
      fi <- fi*(distancia > ns[zm,4] + ns[otros[i],4])
      #Dejar de ver si es factible en el primer traslape
      if(fi == 0) break
    }
    n <- n + 1
  }
  #Regresar la nueva soluci n en caso que sea factible
  if(fi > 0){return(ns)} else {return(s)}
}
#Bsqueda local
set.seed(11)
bl <- M
for (i in 1:1000){
  fi <- 0
  #N mero de iteraciones al momento
  n <- 1
  #Vector de coordenadas x asociadas con las esferas
  x <- bl[,1]
  #Vector de coordenadas y asociadas con las esferas
  y <- bl[,2]
  #Vector de coordenadas z asociadas con las esferas
  z <- bl[,3]
  #Vector de esferas con coordenadas m s alejadas del or ?gen
  d <- c(max(abs(x)),max(abs(y)),max(abs(z)))
  #Generar nueva soluci n dependiendo de la coordenada m s lejana
  if (max(d) == d[1]){
    bl <<- Nx(bl)
  } else if (max(d) == d[2]) {
    bl <<- Ny(bl)
  } else {
    bl <<- Nz(bl)
  }
}
#Regresar nueva soluci n
grasp1 <- bl

open3d()
spheres3d(x = grasp1[,1],
          y = grasp1[,2],
          z = grasp1[,3],
          radius = 0.1,
          color = rainbow(ncol(grasp1)))
spheres3d(x = 0,y = 0,z = 0,radius = R(grasp1), col="white", alpha=0.5)
axes3d()
grasp1

distancias <- function(si,sj){
  sqrt(sum((si-sj)^2))
}
#Matriz de distancias, nxn sin importar el nmero de esferas acomodadas por columnas
mat_distancia<-function(solucion){
  mat_dist<-c()
  for (vj1 in 1:(nrow(solucion))) { #renglones
    reng_dist<-c()
    for (vj2 in 1:(nrow(solucion))) { #columnas
      reng_dist[vj2]<-distancias(solucion[vj1,],solucion[vj2,])
    }
    mat_dist<-rbind(mat_dist,reng_dist)
```

```
  }
  return(mat_dist)
}
distancias_M<-mat_distancia(M)
distancias_grasp1<- mat_distancia(grasp1)

#Matriz de penalidades

mat_penalidades<-function(solucion,mdistancias){
  mat_penal<-c()
  for (vj1 in 1:(nrow(solucion))) { #renglones
    reng_penal<-c()
    row.names(1:n)
    for (vj2 in 1:(nrow(solucion))) { #columnas
      if(mdistancias[vj1,vj2]<2*solucion[1,4]){
        reng_penal[vj2]<-(2*solucion[1,4])-mdistancias[vj1,vj2]
      }
      else {
        reng_penal[vj2]<-0
      }

    }
    mat_penal<-rbind(mat_penal,reng_penal)

  }
  return(mat_penal)
}
penalidades_M<-mat_penalidades(M,distancias_M)
penalidades_grasp1 <- mat_penalidades(grasp1,distancias_grasp1)
rownames(penalidades_grasp1) <- c(1:numcir)
R(grasp1)
proc.time() - t
```

Recocido Simulado

```
#LIBRERAS AUXILIARES
library("rgl")
tiempo <- proc.time()
numcir = 10
set.seed(11)
solin <- function(n,r=0.1){
  #Generar los centros de manera uniforme
  ran <- runif(n = 3*n,min = -1,max = 1)
  #Construir matriz asociada con la solucin
  s0 <- matrix(ran, ncol = 3, dimnames = list(1:n,c("x","y","z")))
  #Agregar los radios en la matriz
  s0 <- as.matrix(cbind(s0,r))
  #Regresar solucin
  s0
}

R <- function(s){
  max(sapply(1:nrow(s),function(i) norm(s[i,1:3], type = "2") + s[i,4]))
}

Nx <- function(s){
  #Ver si es factible la nueva solucin
  fi <- 0
  #Nmero de iteraciones al momento
  n <- 1
  #Vector de coordenadas x asociadas con las esferas
  x <- s[,1]
  #Esfera con coordenada x ms alejada del orgen
  xm <- which.max(abs(x))[1]
  #Esferas diferentes a xm
  otros <- (1:nrow(s))[-xm]
  #Mientras que no sea factible genera nueva solucin  vecina
  while(fi < 1 && n <= 100){
```

```
    #Hacer una copia de la solucin
    ns <- s
    #Genero aleatorios
    u <- runif(n = 3,min = -.25,max = .25)
    #Traslado la esfera asociada con xm.
    ns[xm,1:3] <- u
    #Verificar factibilidad de la nueva solucin
    fi <- 1
    for(i in 1:length(otros)){
      #Distancia de xm al crculo otros[i]
      distancia <- sqrt(sum((ns[xm,-4] - ns[otros[i],-4])^2))
      #Verificar si hay tralape
      fi <- fi*(distancia > ns[xm,4] + ns[otros[i],4])
      #Dejar de ver si es factible en el primer traslape
      if(fi == 0) break
    }
    n <- n + 1
  }
  #Regresar la nueva solucin en caso que sea factible
  if(fi > 0){return(ns)} else {return(s)}
}

Ny <- function(s){
  #Ver si es factible la nueva solucin
  fi <- 0
  #Nmero de iteraciones al momento
  n <- 1
  #Vector de coordenadas y asociadas con las esferas
  y <- s[,2]
  #Esfera con coordenada y ms alejada del orgen
  ym <- which.max(abs(y))[1]
  #Esferas diferentes a xm
  otros <- (1:nrow(s))[-ym]
  #Mientras que no sea factible genera nueva solucin vecina
  while(fi < 1 && n <= 100){
    #Hacer una copia de la solucin
    ns <- s
    #Genero aleatorios
    u <- runif(n = 3,min = -.25,max = .25)
    #Traslado la esfera asociada con xm.
    ns[ym,1:3] <- u
    #Verificar factibilidad de la nueva solucin
    fi <- 1
    for(i in 1:length(otros)){
      #Distancia de xm al crculo otros[i]
      distancia <- sqrt(sum((ns[ym,-4] - ns[otros[i],-4])^2))
      #Verificar si hay tralape
      fi <- fi*(distancia > ns[ym,4] + ns[otros[i],4])
      #Dejar de ver si es factible en el primer traslape
      if(fi == 0) break
    }
    n <- n + 1
  }
  #Regresar la nueva solucin en caso que sea factible
  if(fi > 0){return(ns)} else {return(s)}
}

Nz <- function(s){
  #Ver si es factible la nueva solucin
  fi <- 0
  #Nmero de iteraciones al momento
  n <- 1
  #Vector de coordenadas z asociadas con las esferas
  z <- s[,3]
  #Esfera con coordenada z ms alejada del orgen
  zm <- which.max(abs(z))[1]
  #Esferas diferentes a xm
  otros <- (1:nrow(s))[-zm]
```

```
  #Mientras que no sea factible genera nueva solucin  vecina
  while(fi < 1 && n <= 100){
    #Hacer una copia de la solucin
    ns <- s
    #Genero aleatorios
    u <- runif(n = 3,min = -.25,max = .25)
    #Traslado la esfera asociada con xm.
    ns[zm,1:3] <- u
    #Verificar factibilidad de la nueva solucin
    fi <- 1
    for(i in 1:length(otros)){
      #Distancia de xm al  c rculo  otros[i]
      distancia <- sqrt(sum((ns[zm,-4] - ns[otros[i],-4])^2))
      #Verificar si hay tralape
      fi <- fi*(distancia > ns[zm,4] + ns[otros[i],4])
      #Dejar de ver si es factible en el primer traslape
      if(fi == 0) break
    }
    n <- n + 1
  }
  #Regresar la nueva solucin  en caso que sea factible
  if(fi > 0){return(ns)} else {return(s)}
}

Ns <- function(s){
  #Ver si es factible la nueva solucin
  fi <- 0
  #Nmero de iteraciones al momento
  n <- 1
  #Vector de coordenadas x asociadas con las esferas
  x <- s[,1]
  #Vector de coordenadas y asociadas con las esferas
  y <- s[,2]
  #Vector de coordenadas z asociadas con las esferas
  z <- s[,3]
  #Vector de esferas con coordenadas ms alejadas del origen
  d <- c(max(abs(x)),max(abs(y)),max(abs(z)))
  #Generar nueva solucin  dependiendo de la coordenada ms lejana
  if (max(d) == d[1]){
    ns <- Nx(s)
  } else if (max(d) == d[2]) {
    ns <- Ny(s)
  } else {
    ns <- Nz(s)
  }
  #Regresar nueva solucin
  ns
}

penas <- function(s){
  #Nmero de esferas
  n <- nrow(s)
  #Nmero inicial de traslapes
  Penas <- 0
  #Son necesarios al menos dos  c rculos  para que haya un traslape
  if(n >= 2){
    #Comparar todos los pares distintos de esferas
    for (i in 1:(n-1)) {
      for (j in (i+1):n) {
        #Distancia entre las esferas
        distancia <- sqrt(sum((s[i,1:3]-s[j,1:3])^2))
        Penas <- Penas + (s[i,4] + s[j,4] - distancia)*(distancia < s[i,4] + s[j,4])
      }
    }
  }
  #Regresar medida de traslapamiento
  Penas
}
```

```
cmiope <- function(s){
  R(s) + penas(s)
}

k <- 0
si <- solin(numcir)

#Recocido
RecSimOpt <- function(numcir,alpha,T0,Tfinal,nrep){
  s0 <- solin(numcir)
  sopt <- s0
  zopt <- cmiope(sopt)
  Temp <- T0
  #Repetir hasta criterio
  while(Temp >= Tfinal)
  {
    for(i in 1:nrep)
    {
      s1 <- Ns(s0) #Soluci?n vecina
      d <- cmiope(s1) - cmiope(s0) #delta
      #Ver si se cambia soluci?n
      if(d < 0)
      {
        s0 <- s1
        if(cmiope(s0) < zopt){
          k <<- k + 1
          sopt <- s0
          zopt <- cmiope(sopt)

        }
      } else
      {
        u <- runif(1)
        if(u < exp(-d/Temp))
        {
          s0 <- s1

        }
      }
    }
    Temp <- alpha*Temp
  }

  respuesta <- list()
  respuesta$Solucion <- sopt
  respuesta$Radio <- zopt
  return(respuesta)
}

reD2 <- RecSimOpt(numcir = 10,alpha = 0.99,T0 = 1000,Tfinal = 10^-4, nrep = 50)

open3d()
spheres3d(x = reD2$Solucion[,1],
          y = reD2$Solucion[,2],
          z = reD2$Solucion[,3],
          radius = 0.1,
          color = rainbow(nrow(reD2$Solucion)))
spheres3d(x = 0,y = 0,z = 0,radius = reD2$Radio, col="white", alpha=0.5)
axes3d()

open3d()
spheres3d(x = si[,1],
          y = si[,2],
          z = si[,3],
          radius = 0.1,
          color = rainbow(ncol(si)))
```

```
spheres3d(x = 0,y = 0,z = 0,radius = R(si), col="white", alpha=0.5)
axes3d()

distancias <- function(si,sj){
  sqrt(sum((si-sj)^2))
}
#Matriz de distancias, nxn sin importar el nmero de esferas acomodadas por columnas
mat_distancia<-function(solucion){
  mat_dist<-c()
  for (vj1 in 1:(nrow(solucion))) { #renglones
    reng_dist<-c()
    for (vj2 in 1:(nrow(solucion))) { #columnas
      reng_dist[vj2]<-distancias(solucion[vj1,],solucion[vj2,])
    }
    mat_dist<-rbind(mat_dist,reng_dist)

  }
  return(mat_dist)
}
distancias_si<-mat_distancia(si)
distancias_reD2<- mat_distancia(reD2$Solucion)

#Matriz de penalidades

mat_penalidades<-function(solucion,mdistancias){
  mat_penal<-c()
  for (vj1 in 1:(nrow(solucion))) { #renglones
    reng_penal<-c()
    row.names(1:n)
    for (vj2 in 1:(nrow(solucion))) { #columnas
      if(mdistancias[vj1,vj2]<2*solucion[1,4]){
        reng_penal[vj2]<-(2*solucion[1,4])-mdistancias[vj1,vj2]
      }
      else {
        reng_penal[vj2]<-0
      }

    }
    mat_penal<-rbind(mat_penal,reng_penal)

  }
  return(mat_penal)
}
penalidades_si<-mat_penalidades(si,distancias_si)
penalidades_reD2 <- mat_penalidades(reD2$Solucion,distancias_reD2)
rownames(penalidades_reD2) <- c(1:10)

proc.time() - tiempo

reD2
```

Algoritmos genéticos

```
library("rgl") #Librera para esferas
t <- proc.time() #Tiempo de ejecucin
set.seed(11) #Semilla

#Funcin para crear solucin inicial (matriz con centro de las eferas)
solin <- function(n,r=0.1){
  #Generar los centros de manera uniforme
  ran <- runif(n = 3*n,min = -1,max = 1)
  #Construir matriz asociada con la solucin
  s0 <- matrix(ran, ncol = 3, dimnames = list(1:n,c("x","y","z")))
  #Agregar los radios en la matriz
  s0 <- as.matrix(cbind(s0,r))
  #Regresar solucin
  s0
}
#Funcin para el radio del contenedor:
```

```
#Calcula la norma entre el centro del contenedor y cadaesfera y toma la mayor + 0.1

R <- function(s){
  max(sapply(1:nrow(s),function(i) norm(s[i,1:3], type = "2") + s[i,4]))
}

penas <- function(s){
  #Nmero de esferas
  n <- nrow(s)
  #Nmero inicial de traslapes
  Penas <- 0
  #Son necesarios al menos dos  c rculos  para que haya un traslape
  if(n >= 2){
    #Comparar todos los pares distintos de esferas
    for (i in 1:(n-1)) {
      for (j in (i+1):n) {
        #Distancia entre las esferas
        distancia <- sqrt(sum((s[i,1:3]-s[j,1:3])^2))
        Penas <- Penas + (s[i,4] + s[j,4] - distancia)*(distancia < s[i,4] + s[j,4])
      }
    }
  }
  #Regresar medida de traslapamiento
  Penas
}
cmiope <- function(s){
  R(s) + penas(s)
}

numcir=10
si <- solin(numcir)
M <- si #Creo la solucion inicial
k <- 0
j <- 0
hijos <- matrix(NA, nrow = numcir, ncol = 4)

  Mmiopes <- sapply(1:nrow(M),function(x) R(M[x,,drop=F]))
  orden <- order(Mmiopes) #La ordeno segn los valores de la funcion fitness
  nueva <- M[orden,] #Mi nueva matriz ordenada
    for (k in 1:(numcir/2)-2){ #Repite numcir/2 veces el proceso para crear parejas de padres
      k <- k+1
      j <- j+1
      #pues en cada iteracin  se crean 2 y necesito numcir en total
      #Creamos a los padres mediante la ruleta:
      #Contador para la matriz de hijos
        orden1 <- 1/Mmiopes[orden] #Costos miopes ordenados de mejor a peor
        suma <- sum(orden1) #suma de los costos miopes totales
        cmiopesindividuales <- orden1/suma #Probabilidad de cada individuo
        acumulada <- cumsum(cmiopesindividuales) #sumas acumuladas de los costos miopes. Da 1.

        aleatorio1 <- runif(1, min=0, max=1) #Generamos un aleatorio entre 0 y 1
        madre <- nueva[aleatorio1<=acumulada, ] #Dependiendo del intervalo en que caiga, es  la esfera que t

        aleatorio2 <- runif(1, min=0, max=1) #Lo mismo con el padre
        padre <- nueva[aleatorio2<=acumulada,]

        if(madre[1] == padre[1]){ #En caso de que padre y madre sean iguales, genera otro aleatorio para el
          aleatorio2 <- runif(1, min=0, max=1)
          padre <- nueva[aleatorio2<=acumulada,]
          } else { #Y si no, deja al padre igual
            padre=padre
          }

      #Creamos a los hijos
        beta <- runif(1,min = 0, max = 1) #Creamos un aleatorio entre 0 y 1
        xnuevo_m <- ((1-beta) * madre[1]) + (beta * padre[1])
        xnuevo_p <- ((1-beta) * padre[1]) + (beta * madre[1])
        ynuevo_m <- ((1-beta) * madre[2]) + (beta * padre[2]) #Definimos las nuevas coordenadas como (1-beta
```

```
        ynuevo_p <- ((1-beta) * padre[2]) + (beta * madre[2])
        znuevo_m <- ((1-beta) * madre[3]) + (beta * padre[3])
        znuevo_p <- ((1-beta) * padre[3]) + (beta * madre[3])

        hijo_m <- c(xnuevo_m, ynuevo_m, znuevo_m, 0.1)
        hijo_p <- c(xnuevo_p, ynuevo_p, znuevo_p, 0.1)


        hijos[j,] <- hijo_m
        j <- j+1
        hijos[j,] <- hijo_p

        k <- k+1


       M <- hijos


        }
geneticos <- M

k <- 0
j <- 0
hijos <- matrix(NA, nrow = numcir, ncol = 4)

for (i in 1:1500){
Mmiopes <- sapply(1:nrow(geneticos),function(x) R(geneticos[x,,drop=F]))
orden <- order(Mmiopes) #La ordeno segn los valores de la funcion fitness
nueva <- geneticos[orden,] #Mi nueva matriz ordenada
for (k in 1:numcir/2){ #Repite numcir/2 veces el proceso para crear parejas de padres
  k <- k+1
  j <- j+1
  #pues en cada iteracin  se crean 2 y necesito numcir en total
  #Creamos a los padres mediante la ruleta:
  #Contador para la matriz de hijos
  orden1 <- 1/Mmiopes[orden] #Costos miopes ordenados de mejor a peor
  suma <- sum(orden1) #suma de los costos miopes totales
  cmiopesindividuales <- orden1/suma #Probabilidad de cada individuo
  acumulada <- cumsum(cmiopesindividuales) #sumas acumuladas de los costos miopes. Da 1.

  aleatorio1 <- runif(1, min=0, max=1) #Generamos un aleatorio entre 0 y 1
  madre <- nueva[aleatorio1<=acumulada, ] #Dependiendo del intervalo en que caiga, es  la esfera que toma

  aleatorio2 <- runif(1, min=0, max=1) #Lo mismo con el padre
  padre <- nueva[aleatorio2<=acumulada,]

  if(madre[1] == padre[1]){ #En caso de que padre y madre sean iguales, genera otro aleatorio para el padre
    aleatorio2 <- runif(1, min=0, max=1)
    padre <- nueva[aleatorio2<=acumulada,]
  } else { #Y si no, deja al padre igual
    padre=padre
  }
  #Creamos a los hijos
  beta <- beta <- runif(1,min = 0, max = 1) #Creamos un aleatorio entre 0 y 1
  xnuevo_m <- ((1-beta) * madre[1]) + (beta * padre[1])
  xnuevo_p <- ((1-beta) * padre[1]) + (beta * madre[1])
  ynuevo_m <- ((1-beta) * madre[2]) + (beta * padre[2]) #Definimos las nuevas coordenadas como (1-beta)Ym +
  ynuevo_p <- ((1-beta) * padre[2]) + (beta * madre[2])
  znuevo_m <- ((1-beta) * madre[3]) + (beta * padre[3])
  znuevo_p <- ((1-beta) * padre[3]) + (beta * madre[3])

  hijo_m <- c(xnuevo_m, ynuevo_m, znuevo_m, 0.1)
  hijo_p <- c(xnuevo_p, ynuevo_p, znuevo_p, 0.1)


  hijos[j,] <- hijo_m
  j <- j+1
  hijos[j,] <- hijo_p
```

```
  geneticos <- hijos
}
}

for (i in 1:1500){
  geneticos <- Ns(geneticos)
}

open3d()
spheres3d(x = si[,1],
          y = si[,2],
          z = si[,3],
          radius = 0.1,
          color = rainbow(ncol(si)))
spheres3d(x = 0,y = 0,z = 0,radius = R(si), col="white", alpha=0.5)
axes3d()

open3d()
spheres3d(x = geneticos[,1],
          y = geneticos[,2],
          z = geneticos[,3],
          radius = 0.1,
          color = rainbow(ncol(geneticos)))
spheres3d(x = 0,y = 0,z = 0,radius = R(geneticos), col="white", alpha=0.5)
axes3d()

distancias <- function(si,sj){
  sqrt(sum((si-sj)^2))
}

# #Matriz de distancias, nxn sin importar el nmero de esferas acomodadas por columnas
mat_distancia<-function(solucion){
  mat_dist<-c()
  for (vj1 in 1:(nrow(solucion))) { #renglones
    reng_dist<-c()
    for (vj2 in 1:(nrow(solucion))) { #columnas
      reng_dist[vj2]<-distancias(solucion[vj1,],solucion[vj2,])
    }
    mat_dist<-rbind(mat_dist,reng_dist)

  }
  return(mat_dist)
}
distancias_gen<-mat_distancia(geneticos)
#distancias_GA<- mat_distancia(nueva_generacion1)

#Matriz de penalidades

mat_penalidades<-function(solucion,mdistancias){
  mat_penal<-c()
  for (vj1 in 1:(nrow(solucion))) { #renglones
    reng_penal<-c()
    for (vj2 in 1:(nrow(solucion))) { #columnas
      if(mdistancias[vj1,vj2]<2*0.1){
        reng_penal[vj2]<-(2*0.1)-mdistancias[vj1,vj2]
      }
      else {
        reng_penal[vj2]<-0
      }

    }
    mat_penal<-rbind(mat_penal,reng_penal)

  }
  return(mat_penal)
}
penalidades_gen<-mat_penalidades(geneticos,distancias_gen)
```

```
rownames(penalidades_gen) <- c(1:numcir)

proc.time() - t
R(geneticos)
```

Recocido Simulado + GRASP

```
#LIBRERAS AUXILIARES
library("rgl")
tiempo <- proc.time()
numcir = 10
set.seed(11)
si <- grasp1

solin <- function(n,r=0.1){
  #Generar los centros de manera uniforme
  ran <- runif(n = 3*n,min = -1,max = 1)
  #Construir matriz asociada con la solucin
  s0 <- matrix(ran, ncol = 3, dimnames = list(1:n,c("x","y","z")))
  #Agregar los radios en la matriz
  s0 <- as.matrix(cbind(s0,r))
  #Regresar solucin
  s0
}

R <- function(s){
  max(sapply(1:nrow(s),function(i) norm(s[i,1:3], type = "2") + s[i,4]))
}

Nx <- function(s){
  #Ver si es factible la nueva solucin
  fi <- 0
  #Nmero de iteraciones al momento
  n <- 1
  #Vector de coordenadas x asociadas con las esferas
  x <- s[,1]
  #Esfera con coordenada x ms alejada del orgen
  xm <- which.max(abs(x))[1]
  #Esferas diferentes a xm
  otros <- (1:nrow(s))[-xm]
  #Mientras que no sea factible genera nueva solucin  vecina
  while(fi < 1 && n <= 100){
    #Hacer una copia de la solucin
    ns <- s
    #Genero aleatorios
    u <- runif(n = 3,min = -.25,max = .25)
    #Traslado la esfera asociada con xm.
    ns[xm,1:3] <- u
    #Verificar factibilidad de la nueva solucin
    fi <- 1
    for(i in 1:length(otros)){
      #Distancia de xm al  c rculo  otros[i]
      distancia <- sqrt(sum((ns[xm,-4] - ns[otros[i],-4])^2))
      #Verificar si hay tralape
      fi <- fi*(distancia > ns[xm,4] + ns[otros[i],4])
      #Dejar de ver si es factible en el primer traslape
      if(fi == 0) break
    }
    n <- n + 1
  }
  #Regresar la nueva solucin  en caso que sea factible
  if(fi > 0){return(ns)} else {return(s)}
}

Ny <- function(s){
  #Ver si es factible la nueva solucin
  fi <- 0
  #Nmero de iteraciones al momento
```

```
  n <- 1
  #Vector de coordenadas y asociadas con las esferas
  y <- s[,2]
  #Esfera con coordenada y ms alejada del orgen
  ym <- which.max(abs(y))[1]
  #Esferas diferentes a xm
  otros <- (1:nrow(s))[-ym]
  #Mientras que no sea factible genera nueva solucin  vecina
  while(fi < 1 && n <= 100){
    #Hacer una copia de la solucin
    ns <- s
    #Genero aleatorios
    u <- runif(n = 3,min = -.25,max = .25)
    #Traslado la esfera asociada con xm.
    ns[ym,1:3] <- u
    #Verificar factibilidad de la nueva solucin
    fi <- 1
    for(i in 1:length(otros)){
      #Distancia de xm al  c rculo  otros[i]
      distancia <- sqrt(sum((ns[ym,-4] - ns[otros[i],-4])^2))
      #Verificar si hay tralape
      fi <- fi*(distancia > ns[ym,4] + ns[otros[i],4])
      #Dejar de ver si es factible en el primer traslape
      if(fi == 0) break
    }
    n <- n + 1
  }
  #Regresar la nueva solucin  en caso que sea factible
  if(fi > 0){return(ns)} else {return(s)}
}
Nz <- function(s){
  #Ver si es factible la nueva solucin
  fi <- 0
  #Nmero de iteraciones al momento
  n <- 1
  #Vector de coordenadas z asociadas con las esferas
  z <- s[,3]
  #Esfera con coordenada z ms alejada del orgen
  zm <- which.max(abs(z))[1]
  #Esferas diferentes a xm
  otros <- (1:nrow(s))[-zm]
  #Mientras que no sea factible genera nueva solucin  vecina
  while(fi < 1 && n <= 100){
    #Hacer una copia de la solucin
    ns <- s
    #Genero aleatorios
    u <- runif(n = 3,min = -.25,max = .25)
    #Traslado la esfera asociada con xm.
    ns[zm,1:3] <- u
    #Verificar factibilidad de la nueva solucin
    fi <- 1
    for(i in 1:length(otros)){
      #Distancia de xm al  c rculo  otros[i]
      distancia <- sqrt(sum((ns[zm,-4] - ns[otros[i],-4])^2))
      #Verificar si hay tralape
      fi <- fi*(distancia > ns[zm,4] + ns[otros[i],4])
      #Dejar de ver si es factible en el primer traslape
      if(fi == 0) break
    }
    n <- n + 1
  }
  #Regresar la nueva solucin  en caso que sea factible
  if(fi > 0){return(ns)} else {return(s)}
}
Ns <- function(s){
  #Ver si es factible la nueva solucin
  fi <- 0
  #Nmero de iteraciones al momento
```

```
  n <- 1
  #Vector de coordenadas x asociadas con las esferas
  x <- s[,1]
  #Vector de coordenadas y asociadas con las esferas
  y <- s[,2]
  #Vector de coordenadas z asociadas con las esferas
  z <- s[,3]
  #Vector de esferas con coordenadas ms alejadas del orgen
  d <- c(max(abs(x)),max(abs(y)),max(abs(z)))
  #Generar nueva solucin  dependiendo de la coordenada ms lejana
  if (max(d) == d[1]){
    ns <- Nx(s)
  } else if (max(d) == d[2]) {
    ns <- Ny(s)
  } else {
    ns <- Nz(s)
  }
  #Regresar nueva solucin
  ns
}
penas <- function(s){
  #Nmero de esferas
  n <- nrow(s)
  #Nmero inicial de traslapes
  Penas <- 0
  #Son necesarios al menos dos  c rculos  para que haya un traslape
  if(n >= 2){
    #Comparar todos los pares distintos de esferas
    for (i in 1:(n-1)) {
      for (j in (i+1):n) {
        #Distancia entre las esferas
        distancia <- sqrt(sum((s[i,1:3]-s[j,1:3])^2))
        Penas <- Penas + (s[i,4] + s[j,4] - distancia)*(distancia < s[i,4] + s[j,4])
      }
    }
  }
  #Regresar medida de traslapamiento
  Penas
}

cmiope <- function(s){
  R(s) + penas(s)
}

RecSimOpt <- function(numcir,alpha,T0,Tfinal,nrep){
  s0 <- grasp1
  sopt <- s0
  zopt <- cmiope(sopt)
  Temp <- T0
  #Repetir hasta criterio
  while(Temp >= Tfinal)
  {
    for(i in 1:nrep)
    {
      s1 <- Ns(s0) #Soluci?n vecina
      d <- cmiope(s1) - cmiope(s0) #delta
      #Ver si se cambia soluci?n
      if(d < 0)
      {
        s0 <- s1
        if(cmiope(s0) < zopt){
          sopt <- s0
          zopt <- cmiope(sopt)
        }
      } else
      {
        u <- runif(1)
        if(u < exp(-d/Temp))
```

```
        {
          s0 <- s1
        }
      }
    }
    Temp <- alpha*Temp
  }

  respuesta <- list()
  respuesta$Solucion <- sopt
  respuesta$Radio <- zopt
  return(respuesta)
}

reD2 <- RecSimOpt(numcir = 10,alpha = 0.99,T0 = 1000,Tfinal = 10^-4, nrep = 40)
open3d()
spheres3d(x = reD2$Solucion[,1],
          y = reD2$Solucion[,2],
          z = reD2$Solucion[,3],
          radius = 0.1,
          color = rainbow(nrow(reD2$Solucion)))
spheres3d(x = 0,y = 0,z = 0,radius = reD2$Radio, col="white", alpha=0.5)
axes3d()

open3d()
spheres3d(x = si[,1],
          y = si[,2],
          z = si[,3],
          radius = 0.1,
          color = rainbow(ncol(si)))
spheres3d(x = 0,y = 0,z = 0,radius = R(si), col="white", alpha=0.5)
axes3d()

distancias <- function(si,sj){
  sqrt(sum((si-sj)^2))
}
#Matriz de distancias, nxn sin importar el nmero de esferas acomodadas por columnas
mat_distancia<-function(solucion){
  mat_dist<-c()
  for (vj1 in 1:(nrow(solucion))) { #renglones
    reng_dist<-c()
    for (vj2 in 1:(nrow(solucion))) { #columnas
      reng_dist[vj2]<-distancias(solucion[vj1,],solucion[vj2,])
    }
    mat_dist<-rbind(mat_dist,reng_dist)

  }
  return(mat_dist)
}
distancias_si<-mat_distancia(si)
distancias_reD2<- mat_distancia(reD2$Solucion)

#Matriz de penalidades

mat_penalidades<-function(solucion,mdistancias){
  mat_penal<-c()
  for (vj1 in 1:(nrow(solucion))) { #renglones
    reng_penal<-c()
    row.names(1:n)
    for (vj2 in 1:(nrow(solucion))) { #columnas
      if(mdistancias[vj1,vj2]<2*solucion[1,4]){
        reng_penal[vj2]<-(2*solucion[1,4])-mdistancias[vj1,vj2]
      }
      else {
        reng_penal[vj2]<-0
      }

    }
```

```
    mat_penal<-rbind(mat_penal,reng_penal)

  }
  return(mat_penal)
}
penalidades_si<-mat_penalidades(si,distancias_si)
penalidades_reD2 <- mat_penalidades(reD2$Solucion,distancias_reD2)
rownames(penalidades_reD2) <- c(1:10)
proc.time() - tiempo
```