UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

Facultad de Ciencias

EL PROBLEMA DE EMPAQUETAMIENTO A TRAVÉS DE RECOCIDO SIMULADO

T E S I S

QUE PARA OBTENER EL TÍTULO DE:

MATEMÁTICA

PRESENTA:

ALINE GAMBOA HERNÁNDEZ

DIRECTOR DE TESIS:

DRA. CLAUDIA ORQUÍDEA LÓPEZ SOTO

CIUDAD UNIVERSITARIA, CD. MX

2018

Agradecimientos

A mis padres, por su amor y apoyo incondicional.

A mis guías, la Dra. Claudia Orquídea López Soto y al M. en C. David Chaffrey Moreno Fernández, por sus enseñanzas y su tiempo.

A mis sinodales, La M. en I.O. María del Carmen Hernández Ayuso, la Dra. Bibiana Obregón Quintana, la Dra. Zaida Estefanía Alarcón Bernal, y la Mat. Ana Lilia Anaya Muñoz, por sus aportaciones, observaciones y tiempo para este proyecto.

Especiales agradecimientos al proyecto PAPIIT, con número de expediente $IA106916$, por el apoyo económico durante la realización de este proyecto.

Índice general

Introducción

La disciplina de Investigación de Operaciones es un área de las matemáticas que lidia con problemas de optimización cuyo fin es obtener la "mejor" configuración de un conjunto de variables para lograr ciertos propósitos. Normalmente se busca determinar el máximo (de beneficio, desempeño, ganancias, etc.) o el mínimo (de pérdidas, riesgo, tiempo, etc.) de algún objetivo del mundo real. Dentro de los problemas de optimización se ubican aquellos cuyas soluciones son variables en $\mathbb{R}$ y aquellos cuyas soluciones son codificadas mediante variables discretas. La clase de problemas de Optimización Combinatoria (OC) se localiza entre estos últimos. Problemas como el del agente viajero, el de la mochila, el de asignación cuadrática y el problema de empaquetamiento son algunos que se encuentran en OC. Como existe una gran variedad de problemas de optimización que modelan asuntos de interés tanto en la industria como en la investigación, se abrió camino al estudio y a la clasificación de éstos; dicha clasificación se divide principalmente en dos categorías: los problemas que son *fáciles* de resolver y aquellos que son catalogados como *difíciles*. El problema de empaquetamiento, el cual es de nuestro interés para este trabajo, se encuentra entre estos últimos.

Dada la naturaleza del problema con el que se trabajará en este proyecto, se requiere de herramientas y técnicas especiales que van más allá de un método exacto para poder abordarlo; aquí es donde entran los metaheurísticos. El estudio de éstos empezó desde hace tres décadas buscando formas de resolver problemas de optimización que no se podían resolver eficientemente con los algoritmos existentes. Uno de los metaheurísticos más conocido es Recocido Simulado, el cual estudiaremos a profundidad en este trabajo pues es el mecanismo que usaremos para poder dar una buena solución al problema de empaquetamiento. Recocido Simulado extiende la estrategia básica de búsqueda local permitiendo, con cierto control, movimientos que aparentemente no convienen pero son los que permiten escapar de óptimos locales y obtener una solución más cercana al óptimo global. Recocido Simulado fue uno de los primeros procesos que dio una estrategia explícita para escapar de óptimos locales. Además, es una herramienta que ha ganado fama en distintas áreas por la facilidad de aplicarlo a casi cualquier problema y por su eficiencia en la práctica.

Este proyecto tiene como objetivo principal resolver el problema de empaquetamiento a través de Recocido Simulado programándolo en **R**; para llevar a cabo esto, el trabajo está estructurado de la siguiente manera:

Capítulo 1. Para entender la dificultad del problema de empaquetamiento, en esta parte nos adentraremos a la complejidad computacional y a la clasificación de los problemas de optimización. Posteriormente, se va a presentar el problema de empaquetamiento y sus distintos enfoques, se especificará que se va a trabajar con un problema de empaquetamiento circular y se presentarán los principales planteamientos de éste. Al final se presentarán algunas aplicaciones del problema de empaquetamiento en general.

Capítulo 2. En este capítulo se profundizará acerca de qué es un heurístico y un metaheurístico, poniendo especial atención a la estrategia básica de búsqueda local y siempre enfocándonos en Recocido Simulado.

Capítulo 3. En esta parte del trabajo nos dedicaremos totalmente a Recocido Simulado; se mostrará la historia detrás de este metaheurístico, se verá detalladamente cómo funciona este proceso y lo que se debe tomar en cuenta antes de aplicarlo.

Capítulo 4. Para finalizar, se va a presentar el planteamiento que se ocupará para resolver el problema de empaquetamiento usando Recocido Simulado, es decir, cómo se adaptó el problema para aplicarle este metaheurístico. También se expondrán las estrategias que se utilizarán para resolverlo junto con los resultados obtenidos.

Y finalmente se presentarán las conclusiones del proyecto en el ***Capítulo 5***. Aquí se verá si el objetivo del trabajo se logró satisfactoriamente, un breve resumen de lo que consistió la tesis, y se mostrarán algunos consejos para mejorar los resultados obtenidos.

Capítulo 1

El problema de empaquetamiento

El problema de empaquetamiento es un problema de Optimización Combinatoria (OC) científicamente desafiante que ha despertado el interés en una amplia gama de investigadores, esto se debe al reto que conlleva resolverlo y a la gran variedad de aplicaciones que tiene en la industria. Por ejemplo, en el cargamento de contenedores, la dispersión de instalaciones y en redes de comunicación [3].
Sin embargo, antes de adentrarnos al problema de empaquetamiento y sus aplicaciones, es necesario entender el tipo de problema con el que se va a trabajar. Para esto, hay una gran variedad de áreas que se dedican a estudiar los problemas de oprimización a través de distintos enfoques; en particular, la teoría de la complejidad computacional nos ha dado herramientas para poder clasificar estos problemas en dos principales subclases: los problemas que son *fáciles* de resolver y los que son catalogados como *difíciles*. Para los primeros, existen algoritmos eficientes (en cuanto al tiempo) que los resuelven hasta la optimalidad. Para los segundos, se cree que no existen algoritmos que puedan obtener la solución óptima en un tiempo razonable. A consecuencia de esto, se recurren a distintos procedimientos para poder abordar estos problemas, por ejemplo: los heurísticos y metaheurísticos, los cuales se estudiarán a fondo en este trabajo (Capítulo 2).
El problema de empaquetamiento, el cual se abordará en este proyecto, se encuentra entre los problemas que son catalogados como *difíciles* de resolver y se presentará en este capítulo de la siguiente manera: primero se presentará lo que es la complejidad computacional, luego, se abordará la NP - Completez para poder entender qué hace que un problema sea difícil o fácil de resolver y saber en dónde queda clasificado el problema con el que nos enfrentaremos. Posteriormente se presentará el problema de empaquetamiento con el que se trabajará y las principales formas de plantearlo, para finalmente, hacer algunas observaciones de éste y ver algunas de sus aplicaciones.

1.1. Un poco de complejidad computacional

Al estudiar un problema de optimización es inevitable estudiar los algoritmos que lo resuelven. De hecho, dichos algoritmos son los que nos permiten saber el tipo de problema con el que se trabaja.

Existen varios criterios para juzgar la eficiencia de los algoritmos; por ejemplo, la complejidad del tiempo y espacio de éstos, y la calidad de la solución que generan [6]. Mientras unos juzgan a un algoritmo por lo complejo que es analizarlo, otros prefieren evaluarlo por cómo se desempeña en la práctica con problemas del mundo real. Nosotros clasificaremos a un algoritmo mediante la complejidad del tiempo y espacio de éstos.
Antes de explicar estos criterios, es necesario entender lo que es una instancia de un problema de optimización; la cual se puede entender como un caso particular de un modelo. Formalmente:

Definición 1.1. Una instancia de un problema de optimización es una pareja (F, c), donde F es el conjunto de soluciones factibles del problema y $c : F \longrightarrow \mathbb{R}$ es la función de costo. Donde se busca encontrar $x \in F$ tal, que $c(x) \leq c(y)$ para todo $y \in F$ (para un problema de minimización).

1.1.1. Complejidad del tiempo y espacio

La complejidad del tiempo y del espacio de un algoritmo es la forma analítica de evaluarlo; esto se refiere al tiempo que un algoritmo se tarda para resolver un problema de tamaño n, y el espacio que se requiere para poder ejecutarlo. Es importante notar que es imposible medir con exactitud la complejidad de estos factores en un algoritmo, pues el tiempo y espacio requeridos para ejecutarlo va a depender del hardware y software instalados en el sistema con el que se esté trabajando [6]. Así, el enfoque que se ha dado para medir estos criterios de una manera objetiva es contar el número total de *operaciones* hechas por el algoritmo en términos del tamaño de los argumentos de entrada. En este contexto, el tamaño de los argumentos de entrada se refiere a la cantidad de memoria requerida para almacenar la información de cualquier instancia de un problema (también se le conoce como el tamaño del problema). Por otro lado, cuando hablamos de *operaciones* nos referimos al conjunto de instrucciones de un algoritmo, cuya cantidad es independiente al tamaño del problema.
Aún así, puede ser complicado contar con exactitud el número total de operaciones de un algoritmo, por lo que se recurre a una función que acote esta cantidad. Es decir, se recurre a una función que nos dice, a través del tamaño de un problema, cómo va creciendo el tiempo computacional requerido para resolverlo. Para esto, se introduce la notación O.

Definición 1.2. Una función positiva $f(n)$ se dice que es $O(g(n))$ si existen dos constantes $c \geq 1$, $n_0 \geq 1$ tales, que $f(n) \leq c \cdot g(n)$ para todo $n \geq n_0$. A la función $g(n)$ se le conoce como el término de orden mayor.

Notamos que, por definición, para una función $f(n)$ pueden existir funciones $g'(n)$, $g''(n)$ distintas tales que $f(n)$ es $O(g'(n))$ y también es $O(g''(n))$. Sin embargo, se prefiere usar la función $g(n)$ con el menor crecimiento posible [6].
De esta forma, la complejidad del tiempo y espacio de un algoritmo se expresa con la notación O, la cual describe el índice de crecimiento de dicho algoritmo en términos del tamaño del problema. El índice de crecimiento de un algoritmo es la principal limitante para el uso práctico del algoritmo en cuestión; pues aquellos que requieren de mayor

tiempo para ser ejecutados, rápidamente se vuelven inútiles cuando se quieren aplicar a problemas reales [16].
La notación O también es de gran ayuda al querer comparar dos algoritmos. Por ejemplo, para problemas de tamaño n, con n muy grande, el algoritmo con menor índice de crecimiento es el mejor; y cuando dos algoritmos tienen constantes c y n_0 muy cercanas, el algoritmo con la función $g(n)$ de menor crecimiento va a requerir menos tiempo para ser ejecutado [6].
Es importante hacer notar que el análisis de complejidad nos indica que, en el peor de los casos, con el crecimiento del tamaño de un problema el algoritmo requerirá más y más operaciones (y tiempo) para dar una solución definida [2]. Es decir, lo que nos da la complejidad computacional es tiempo y espacio requeridos para resolver una instancia pensando en el peor escenario posible.
Teniendo una idea más clara de lo que es la complejidad del tiempo y espacio de un algoritmo, cómo se calcula y la información que podemos sacar de ésta, podemos andentrarnos a lo que es la NP - Completez.

1.1.2. NP - Completez

Desde antes de los 70's, investigadores ya estaban conscientes de la existencia de problemas que computacionalmente los resuelven algoritmos con un tiempo de complejidad tipo polinomial $O(n)$, $O(n^2)$, $O(n^3)$, etcétera; para estos problemas, el tiempo computacional requerido para resolverlos es bajo. Mientras que por otro lado, hay problemas que computacionalmente son resueltos por algoritmos con tiempo de complejidad de tipo exponencial $O(2^n)$ y $O(n!)$ [6]. Para estos últimos, la velocidad con la que crece su respectiva función $g(n)$ es muy alta, por lo que aún para valores pequeños de n, el tiempo que requiere el algoritmo para ser ejecutado puede ser demasiado; como consecuencia de esto, la mayoría de las veces el algoritmo se vuelve inútil en la práctica. De esta manera, surgieron los heurísticos y aumentó la importancia del estudio de los problemas de optimización y la clasificación de sus respectivos problemas de decisión.

Al hablar de la clasificación de los problemas de optimización combinatoria, mediante la complejidad del tiempo y espacio de los algoritmos que los resuelven, se habla de la NP - Completez. La NP - Completez clasifica los problemas de decisión como difíciles o fáciles de resolver. Un problema de decisión es aquel cuyas únicas respuestas posibles son "*si*" o "*no*"; cada problema de optimización tiene asociado un problema de decisión. A continuación se mencionan algunas propiedades de la relación entre problemas de optimización y sus respectivos problemas de decisión [6]:

• La solución de un problema de optimización se usa directamente para responder el problema de decisión correspondiente.
• Como consecuencia de la observación anterior, un problema de optimización es al menos tan difícil de resolver como su respectivo problema de decisión.
• Si se prueba que un problema de decisión es computacionalmente intratable, entonces

el problema de optimización correspondiente también será un problema computacionalmente intratable.

Formalmente, las dos principales clases en las que se dividen los problemas de decisión son:

Definición 1.3 (clase P)**.** Un problema de decisión Q pertenece a la clase P si existe un algoritmo que resuelve cualquiera de sus instancias en tiempo polinomial.

Definición 1.4 (clase NP)**.** Un problema de decisión Q pertenece a la clase NP si y sólo si cualquiera de sus instancias cuya respuesta sea "*si*", se puede resolver en tiempo polinomial por un algoritmo no determinístico.

Coloquialmente, los problemas *fáciles* pertenecen a P y los *difíciles* pertenecen a NP. Se sabe que $P \subseteq NP$, y aunque $NP \subseteq P$ siga siendo un problema abierto en la teoría de complejidad computacional, la conjetura que se usa es $P \neq NP$. Posteriormente, surgió la clase de los problemas de decisión llamados $NP - Completos$.

Definición 1.5 (problemas NP - Completos)**.** Se dice que un problema de decisión Q es $NP - Completo$ si Q $\in NP$ y si para cualquier otro problema en NP, existe una transformación de éste a Q de tiempo polinomial.

El estudio de problemas $NP-Completos$ es muy importante pues éstos tienen la siguiente propiedad: si existe un algoritmo de tiempo polinomial que resuelva algún problema $NP-Completo$, entonces existirán algoritmos de tiempo polinomial que resuelvan todos los problemas en NP [16]. Para demostrar que un problema Q es $NP-Completo$ se requiere demostrar dos cosas; que dicho problema está en NP y dar una transformación de tiempo polinomial de un problema en NP a Q. Para tener una idea más clara de la organización de estos problemas, La Figura 1.1 ilustra la clasificación de los problemas de decisión.

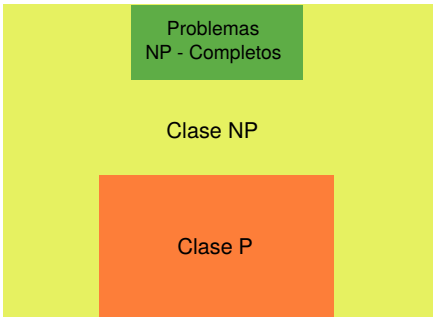


Figura 1.1. Clasificación de los problemas de decisión.

Definición 1.6 (problemas NP - Duros)**.** Un problema de optimización se dice que es $NP - Duro$ si su respectivo problema de decisión es $NP - Completo$

Para los problemas $NP-Duros$ se cree que no hay ningún algoritmo de tiempo tipo polinomial que resuelva todas sus instancias hasta encontrar el óptimo. Entre los problemas $NP-Duros$ más destacados están el problema del agente viajero y el problema de empaquetamiento; en particular, del problema de empaquetamiento sólo se ha encontrado el óptimo de muy pocas instancias, y son de aquellas cuyo tamaño es pequeño.
Por la complejidad de un problema de empaquetamiento, para dar una solución *buena* a éste, se requiere el uso de procedimientos metaheurísticos que combinan heurísticos (como el de búsqueda local) y estrategias adicionales para expandir la búsqueda en el espacio de soluciones [7]; lo cuál se verá con mayor detalle en el Capítulo 2.

1.2. El problema de empaquetamiento

Un problema de empaquetamiento general en dos dimensiones consiste en acomodar N objetos geométricos con tamaño y forma determinada sin traslaparse en una región Ω dada. Normalmente la calidad de este arreglo depende del tamaño de la región Ω y de la distancia entre los N objetos geométricos [3].

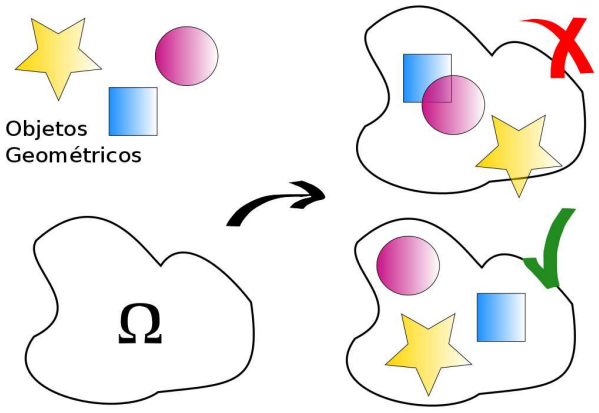


Figura 1.2. Ejemplo de un problema general de empaquetamiento

Sin embargo, el problema de empaquetamiento que más se ha estudiado, debido a que da paso a problemas de optimización y por sus aplicaciones en la industria, es el Problema de Empaquetamiento de Círculos (PEC). En este problema los objetos geométricos a acomodar son círculos, los cuales pueden ser idénticos o no, y la región Ω puede ser un círculo, cuadrado, rectángulo, etc.
En particular, para fines de este proyecto, se enfocará en el PEC donde Ω es también un círculo. A continuación se mostrarán los dos principales planteamientos del **PEC**, los cuales son equivalentes.

1.2.1. Problema de Empaquetamiento de Círculos 1 (PEC1)

El problema **PEC1** se enfoca en encontrar el máximo radio r de N círculos idénticos tales, que estos círculos no se traslapen y que estén contenidos en un círculo fijo C en $\mathbb{R}^2$; sin pérdida de generalidad, tomaremos a C como el círculo unitario. Aunque este planteamiento no tenga gran utilidad en la industria, resolverlo no deja de ser un gran desafío. Algunos autores proponen cambiar entre el sistema cartesiano y el polar en la resolución de este problema; por ende, el planteamiento suele estar expresado usando ambos sistemas de coordenadas [7]. Sin embargo, con la finalidad de ilustrar este problema, el modelo se formulará únicamente usando el sistema cartesiano. Para esto, sean N el número de círculos con los que se trabajará, r el radio de los N círculos idénticos, (x_i, y_i) las coordenadas del centro del círculo i y $d_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$ la distancia entre los centros de los círculos i y j. La formulación queda expresada como [10]:

$$
\begin{aligned}
max \quad & r & & \\
sujeto\ a & & & \\
\sqrt{(x_i)^2 + (y_i)^2} &\leq 1 - r,\ i = 1, ..., N & & (1.1)\\
2r &\leq d_{ij},\ i, j = 1, ..., N;\ i < j & & (1.2)\\
-1 &\leq x_i \leq 1,\ i = 1, ..., N & & (1.3)\\
-1 &\leq y_i \leq 1,\ i = 1, ..., N & & (1.4)\\
0 &\leq r & & (1.5)
\end{aligned}
$$

Donde la expresión 1.1 asegura que los círculos idénticos estén contenidos en el círculo unitario, la expresión 1.2 nos asegura que los círculos no se traslapen entre ellos, las expresiones 1.3 y 1.4 aseguran que los centros de los círculos estén dentro del cuadrado con vértices en $(\pm 1, \pm 1)$ y la expresión 1.5 nos asegura que los radios de los círculos no sea negativo.

1.2.2. Problema de Empaquetamiento de Círculos 2 (PEC2)

Este planteamiento busca encontrar el mínimo valor del radio R de un contenedor circular C tal, que se puedan acomodar N círculos con radios fijos y arbitrarios sin traslaparse entre ellos y que queden dentro del contenedor circular. Para plantear este problema, sean R y (x, y) el radio y las coordenadas del centro de C respectivamente, r_i y (x_i, y_j) el radio y las coordenadas del centro del círculo i respectivamente y $d_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$ la distancia entre los centros de los círculos i y j. Así, la formulación queda expresada como [7]:

$$
\begin{aligned}
min \quad & R \\
sujeto\ a & \\
\sqrt{(x_i - x)^2 + (y_i - y)^2} \quad &\leq \quad R - r_i,\ i = 1, ..., N \qquad (1.6)\\
r_i + r_j \quad &\leq \quad d_{ij},\ i, j = 1, ..., N;\ i < j \qquad (1.7)\\
\sqrt{\sum_{i=1}^{N} (r_i)^2} \quad &\leq \quad R \qquad (1.8)
\end{aligned}
$$

Donde la expresión 1.6 nos asegura que los círculos estén totalmente contenidos en el contenedor C, la expresión 1.7 nos asegura que los círculos no se traslapan entre ellos y la expresión 1.8 da una cota inferior del radio del contenedor. Esta última sustituye la restricción de no negatividad cuya omisión haría de este problema uno no acotado y por lo tanto, no tendría solución [7].

Proposición 1.1. PEC1 $\Leftrightarrow$ PEC2

PEC1 y **PEC2** son equivalentes pues hay una relación uno a uno entre ellos la cual está basada en un factor de escala [10]. Para ilustrar esta equivalencia, basta dar una idea de la demostración de $PEC1 \Rightarrow PEC2$ pues la demostración de $PEC2 \Rightarrow PEC1$ es análoga; para esto, se basará en la demostración dada en [10].

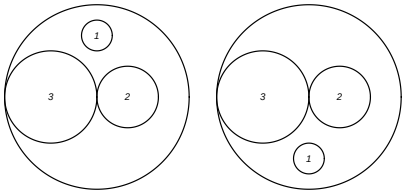
Demostración. Con respecto a **PEC1** supongamos que, sin pérdida de generalidad, el contenedor circular es el círculo unitario; y con respecto a **PEC2** supondremos que, sin pérdida de generalidad, los círculos a acomodar son de radio 1.
Ahora, sea $N \in \mathbb{N}$, s^* una solución óptima de **PEC1** de N círculos y r^* el radio máximo de los N círculos idénticos asociado a s^*. Notamos que, como el área del contenedor es directamente proporcional al cuadrado de su radio, minimizar el área del contenedor circular se logra minimizando el radio de éste. Si multiplicamos el radio del círculo unitario (contenedor) y el radio de los N círculos idénticos asociados a s^* por el factor $\frac{1}{r^*}$, tendremos un círculo de radio $\frac{1}{r^*}$ y N círculos unitarios. Este contenedor circular resultante de radio $\frac{1}{r^*}$ va a ser tal, que podrá contener a los N círculos unitarios sin traslaparse; y además, el arreglo resultante será una solución óptima del **PEC2** pues se empezó con una solución óptima de $PEC1$ (de lo contrario se llegaría a una contradicción).
Para demostrar que $PEC2 \Rightarrow PEC1$ es similar salvo que el factor de escala en este caso sería r^*. □

Antes de cerrar esta sección, cabe destacar las siguientes observaciones:

•Hay una infinidad no numerable de soluciones óptimas de un PEC [7]
Teniendo una solución óptima de un PEC, ésta se puede rotar y/o reflejar obteniendo una solución distinta (pues se modificó la posición de los círculos) pero conservando la optimalidad; ya que no se modificaron los radios de los círculos. Otra forma de generar

más soluciones óptimas distintas es encontrando en el arreglo un círculo al cual se pueda modificar la posición de su centro sin perturbar el radio de los círculos que determinan la optimalidad (véase Figura 1.3).

•El PEC de nuestro interés
El PEC con el que se estará trabajando en este proyecto corresponde al de un $PEC2$. Sin embargo, para fines prácticos se utiliza una formulación distinta de éste para resolverlo con Recocido Simulado. Es importante notar que se expuso una de tantas formas de modelar un $PEC2$.
El planteamiento del $PEC2$ que se utilizará y los detalles de éste, se verán con profundidad en el Capítulo 4.



(a) (b)

Figura 1.3. Ejemplo de cómo se pueden generar soluciones distintas con el mismo valor en la función objetivo (ambos contenedores circulares tienen centro en el origen y radio $r = 6$).

1.3. Algunas aplicaciones

En años recientes, la investigación en el problema de empaquetamiento ha aumentado y no debe sorprender, pues las aplicaciones de este problema son de gran ayuda para las empresas y se han tomado en cuenta para hacer diferentes aplicaciones en la insdustria. Algunas de éstas incluyen el cargamento de contenedores, la dispersión de instalaciones y redes de comunicación.

Un ejemplo en el área de cargamento de contenedores se puede apreciar en la industria del papel y pulpa; en esta industria los productos suelen transportarse en contenedores en forma de hojas planas o bien en carretes. Cuando se transportan en carretes éstos son de distintos diámetros y se colocan parados, por lo que este problema equivale a un problema de empaquetamiento de círculos, en este caso no uniformes, en dos dimensiones [3]. En cuanto a la dispersión de instalaciones y redes de comunicación, un ejemplo puede verse cuando se plantea poner cierto número de torres de internet inalámbrico en una región; se busca localizar estas torres de tal forma, que estén lo suficientemente lejos para cubrir la mayor región posible con internet, pero no tan lejos para dejar sin este servicio algunas partes de la región. Este problema se puede plantear como un $PEC1$, así las torres estarán tan dispersas como sea posible. De la misma forma se puede plantear un problema

de facilidad de dispersión, donde en vez de buscar ubicar torres de internet inalámbrico, se busca situar locales; como de comida rápida [3].

Estas son solo algunas de las aplicaciones que puede llegar a tener el problema de empaquetamiento en la industria. Sin duda han sido de gran ayuda y dan cabida a seguir adentrándose en este problema para encontrar más aplicaciones y/o propiedades.

1.4. En Resumen

La gran aplicabilidad a problemas de la industria y el desafío que puede representar resolverlos, son algunas de las razones por las que los problemas de optimización combinatoria han sido objeto de estudio en diversas ramas. En particular, la teoría de complejidad computacional da herramientas para poder clasificar estos problemas en cuanto a lo complicado que puede llegar a ser resolverlos. En general, estos problemas se dividen en dos clases: los que son *sencillos* y para los cuales hay algoritmos que son capaces de dar una solución óptima de manera rápida, y los que son notoriamente *difíciles* de resolver y para los cuales se requieren de distintas herramientas para poder abordarlo; pues no existen algoritmos que obtengan la solución óptima de manera rápida.
El problema de nuestro interés, el problema de empaquetamiento, es catalogado como $NP - Duro$; esto nos dice que se requiere una mayor cantidad de recursos y/o distintas maneras de abordarlo para poder resolverlo. Sin embargo, las aplicaciones que tiene en problemas reales y lo complicado que puede llegar a obtener una buena solución, el problema de empaquetamiento no deja de ser un tema de investigación y de interés para muchos.

Capítulo 2

Heurísticos y Metaheurísticos

Dentro de la comunidad de Investigación de Operaciones, es ampliamente aceptable hacer uso de heurísticos y metaheurísticos para resolver problemas del mundo real. Esto se debe a que la gran mayoría de los problemas de decisión asociados a problemas complejos del mundo real, al ser modelados como un problema de optimización, pertenecen a la clase de problemas $NP - Duros$. Así, el uso de métodos exactos para estos problemas no será factible para instancias de gran escala [2].

En 1966, Ronald L. Graham introdujo formalmente los métodos de aproximación también conocidos como heurísticos. Éstos emergieron al enfrentarse con problemas, conocidos como NP-Duros, para los cuales un método exacto no era eficiente. Etimológicamente, la palabra heurístico viene del griego *heuriskein* que significa encontrar o descubrir. Aunque hay distintas definiciones de un heurístico, para nosotros los heurísticos son criterios, métodos, o principios para elegir cuál serie de acciones, de entre varias, promete ser la más efectiva para poder llegar a un objetivo. Un heurístico es aproximado en el sentido de que talvez proporciona una buena solución por relativamente poco esfuerzo, pero no garantiza la optimalidad [2].

En las últimas décadas han emergido un nuevo tipo de métodos de aproximación que trata de combinar heurísticos con nuevas estrategias buscando explorar el espacio de soluciones de una manera eficiente y efectiva. En un principio, a estos métodos les llamaban "heurísticos modernos" y ahora se conocen como metaheurísticos [1]. Aún no hay una definición común aceptada de qué es un metaheurístico, sin embargo, se puede pensar a un metaheurístico como un procedimiento de alto nivel que coordina heurísticos clásicos y reglas para encontrar soluciones de alta calidad de problemas en los que los heurísticos clásicos no son efectivos. Un metaheurístico va más allá de la solución que podría proporcionar un heurístico clásico [16].

A lo largo de este capítulo se verá con más detalle los métodos de aproximación y se hará la distinción de éstos con los métodos exactos, pues aunque el objetivo de los dos es encontrar la solución óptima de un problema, lo que cada método nos garantiza obtener es esencialmente distinto. Posteriormente, se estudiará con más profundidad a los metaheurísticos, su definición y su clasificación. Para fines de este proyecto, se prestará especial atención al heurístico de búsqueda local básica, pues es parte fundamental del

metaheurístico Recocido Simulado, el cual es de nuestro interés.

2.1. Métodos exactos vs. Métodos de aproximación

Un método exacto, al aplicarlo a cualquier instancia finita de un problema de optimización, nos garantiza encontrar en un tiempo acotado el óptimo global. Además, hay manera de rectificar que la solución final es óptima y, en caso de que el óptimo no exista, se garantiza con una demostración sobre la inexistencia de éste. Un ejemplo de un método exacto es el famoso algoritmo Simplex, el cual es la herramienta básica para resolver problemas de programación lineal. Sin embargo, aunque los problemas que se pueden resolver mediante estos métodos han ido aumentando por los avances tecnológicos y teóricos, aún hay problemas, o instancias de problemas, que no es posible resolverlos mediante un método exacto pues requerirían de tiempos computacionales tan altos, que se vuelven inútiles en la práctica [16].
En el Capítulo 1 se vio que hay problemas clasificados como NP- Duros para los cuales aún no se conoce, y es poco probable de que exista, un método exacto que obtenga una solución óptima en un tiempo razonable. Para poder manejar estos problemas en la práctica, se debe de tener un acercamiento distinto al de un método exacto. Aquí es donde entran los heurísticos, los cuales son criterios, métodos, o principios para elegir cuál serie de acciones, de entre varias, promete ser la más efectiva para poder llegar a un objetivo. En otras palabras, los heurísticos son estrategias para resolver un problema de optimización bien definido mediante una aproximación intuitiva, en la que se utiliza la estructura del problema de forma inteligente para obtener una buena solución. También se puede pensar a un heurístico como una técnica, que consiste en una o varias reglas, que busca (y ojalá encuentre) buenas soluciones a un costo computacional razonable. En un método de aproximación, a comparación de uno exacto, se sacrifica la garantía de encontrar la solución óptima por la esperanza de encontrar una buena solución en un tiempo considerablemente menor [1]. No obstante, en la práctica corre más rápido un método de aproximación que uno exacto y a comparación de estos últimos, un heurístico es más eficiente al resolver grandes instancias de un problema [12]. Esto se debe en gran parte al tiempo computacional requerido, pues para querer resolver instancias de gran tamaño con un método exacto, el costo computacional llega a ser tan grande que el método termina siendo inaplicable. Por ello, en la práctica se espera que un heurístico sea [12];

- **Eficiente.** Que el tiempo computacional requerido sea realista.
- **Bueno.** La solución heurística debe estar, en promedio, cerca del óptimo.
- **Robusto.** La probabilidad de que la solución heurística sea mala, es baja.

En el estudio de los métodos de aproximación destacan dos principales tipos de heurísticos; los *constructivos* y los de *búsqueda local*. Los primeros son aquellos que construyen la solución inicial desde cero y están basados en algoritmos voraces. Los segundos son aquellos que empiezan con una solución factible y se va mejorando iterativamente mediante

cambios locales de ésta hasta que ya no se pueda mejorar la solución actual.
A continuación se verá a fondo el heurístico básico de búsqueda local ya que Recocido Simulado, el método de nuestro interés en este proyecto, es una extensión de esta técnica.

2.1.1. Búsqueda Local Básica

Cualquier técnica de búsqueda local tendrá subyacente una función de vecindad, la cual servirá como guía para encontrar una buena solución al problema con el que se esté trabajando.

Definición 2.1. Una función de vecindad es un mapeo $N : S \longrightarrow 2^S$, con S el conjunto de soluciones factibles del problema en cuestión, donde a cada solución $i \in S$ se le asocia un conjunto de soluciones $N(i) \subseteq S$ las cuales, de alguna forma, son *cercanas* a i. A $N(i)$ se le denomina *vecindad* de i y a cada $j \in N(i)$ se le denomina *vecino* de i. [5]

Las vecindades y la estructura de éstas son fundamentales y juegan un papel importante en un heurístico de búsqueda local; pues la función de vecindad define el recorrido en el espacio de soluciones factibles que se usará para explorarlo. Además, de las vecindades depende la calidad de la solución heurística obtenida.
Una instancia de un problema de optimización combinatorio consiste en:

- Un conjunto S de soluciones factibles
- Una función objetivo de costo f no negativa

Donde la tarea es encontrar una solución óptima $i^* \in S$ con costo de función objetivo f^*.
La técnica tradicional de búsqueda local también se conoce como *mejoramiento iterativo* o *búsqueda monótona*, la cual va explorando el conjunto S mediante la función de vecindad N iterativamente buscando mejorar el costo de la función objetivo. Es decir, en cada iteración se va moviendo la solución actual $i \in S$ a una solución vecina $i' \in N(i)$ siempre y cuando esta solución vecina mejore el costo de la función objetivo. Hay varias formas de elegir un vecino: se puede escoger como vecino al primer elemento encontrado que mejore la función objetivo, o se puede hacer una búsqueda exhaustiva hasta encontrar al mejor de entre todos los vecinos, o alguna opción intermedia.
Por definición, utilizando esta técnica se llega a un óptimo local el cual no siempre coincide con el óptimo global, además, la solución final depende totalmente de la solución inicial. Para tratar con este problema, se han recomendado hacer algunas variaciones a este heurístico, por ejemplo, empezar con distintas soluciones iniciales o hacer la estructura de las vecindades más compleja para ampliar el alcance de búsqueda [5]. Sin embargo, aún con estas variaciones, los resultados no han sido del todo satisfactorios. Cabe destacar que el uso exitoso de esta técnica se debe, normalmente, a la función de vecindad que se define y a la estrategia de búsqueda que se decida usar.

La Figura 2.1 [15] muestra un ejemplo para un problema de minimización. Con ésta se exhibe de una forma clara el problema que se tiene al trabajar con la técnica tradicional de búsqueda local.

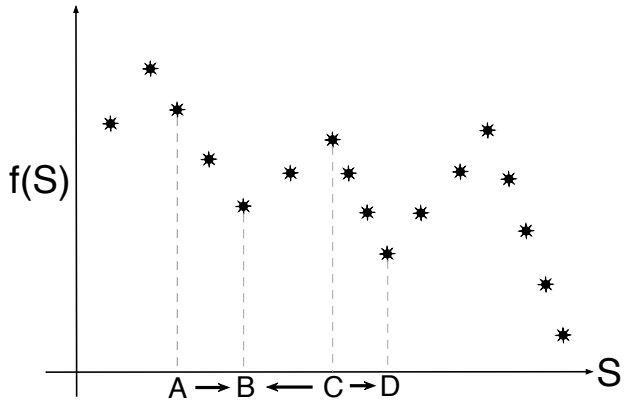


Figura 2.1. Representación de óptimos locales y global.

En este ejemplo, se tiene una gráfica donde en el eje de las abcisas se toman los elementos del conjunto de soluciones factibles y en el eje de las ordenadas están los valores que toman estas soluciones al aplicarles la función de costo f; este ejemplo es una función en donde cada solución tiene dos vecinos, representados por los puntos que están exactamente a la izquierda y/o derecha de ésta.
Ahora, si suponemos que empezamos con la solución inicial A, la única solución a la que se llegará será B; pues usando búsqueda local no se aceptan deterioraciones en la función de costo, a comparación de otras técnicas que se verán más adelante (como Recocido Simulado). Sin embargo, si se toma a C como solución inicial, bien se puede terminar en la solución B o en la solución D, esto depende de la estrategia de búsqueda que se use. De esta forma, se ilustra que la solución final depende totalmente de la solución con la que se empezó, y además, los óptimos locales que se pudieron llegar a obtener, B o D, no coinciden con el óptimo global.
Así, la técnica tradicional de búsqueda local sólo te asegura que la solución final es, entre sus vecinos, la mejor.

2.1.2. Ejemplo

Para ejemplificar el heurístico tradicional de búsqueda local y sus desventajas, trabajaremos con una instancia del problema de la mochila binario. El cual se sabe que entra en la categoria de los problemas NP-Duros.
El problema de la mochila binario consiste en maximizar el beneficio dado por el peso de n objetos distintos, respetando la capacidad máxima de la mochila. El planteamiento general de este problema es de la siguiente forma:

$$x_i = \begin{cases} 1 \text{ si se incluye el objeto i en la mochila} \\ 0 \text{ en otro caso} \end{cases}$$

$$\text{máx} \sum_{i=1}^{n} w_i x_i$$

$$s.a. \sum_{i=1}^{n} p_i x_i \leq C$$

$$x_i \in \{0,1\}, i = 1, ..., n$$

Donde x_i son las variables de decisión, p_i es el peso del objeto i, w_i es la utilidad o beneficio del objeto i, C es la capacidad máxima de la mochila y n es el número de objetos.
Para esta instancia se tendrán 10 objetos, $n = 10$, la capacidad de la mochila será $C = 700$, la utilidad o beneficio, y el peso de cada objeto se muestra en la siguiente tabla;

Objeto	1	2	3	4	5	6	7	8	9	10
p_i	100	155	50	112	70	80	60	118	110	55
w_i	150	190	100	155	110	130	120	158	151	105

De esta forma, el planteamiento de la instancia a resolver será de la siguiente forma:

$$Max f(x) = 150x_1 + 190x_2 + 100x_3 + 155x_4 + 110x_5 + 130x_6 + 120x_7 + 158x_8 + 151x_9 + 105x_{10}$$

$$s.a. 100x_1 + 155x_2 + 50x_3 + 112x_4 + 70x_5 + 80x_6 + 60x_7 + 118x_8 + 110x_9 + 55x_{10} \leq 700$$

$$x_i \in \{0,1\}, i = 1, ..., 10$$

Ahora, para usar búsqueda local debemos definir tres elementos:

- La estructura del espacio de soluciones (S) y las vecindades ($N(S)$).
- La forma de elegir vecinos.
- La solución inicial.

Para empezar, definiremos la estructura del espacio de soluciones y las vecindades. Para esto, consideremos $c = (100, 155, 50, 112, 70, 80, 60, 118, 110, 55)$. Así, el espacio de soluciones factibles S para esta instancia se define como:

$$S = \{s \in \{0,1\}^{10} \,|\, sc^T \leq 700\}$$

Donde dada $s \in S$, la entrada i de s está asociada con la variable de decisión x_i. Ahora, se definirá el subconjunto $N(s) \subseteq S$ para cada $s = (s_1, s_2, ..., s_{10}) \in S$;

$$N(s) = \{t = (t_1, t_2, ..., t_{10}) \in S \,|\exists i \in \{1, ..., 10\}; s_i \neq t_i \wedge s_j = t_j \forall j \in \{1, ..., 10\} \setminus \{i\}\}$$

Así, dos soluciones s, $t \in S$ serán vecinas si y sólo si difieren en exactamente una entrada. En otras palabras, dos soluciones serán vecinas si una de ellas lleva en la mochila los mismos elementos que la otra con un elemento extra. De esta forma, cualquier solución en S tiene exactamente diez vecinos (uno por cada entrada de la solución). La forma de escoger a un vecino será aleatoria; se escogerá al azar un número del uno al diez y se cambiará el valor a la entrada correspondiente. Si la entrada que se escogió al azar tiene un uno, se le asigna el valor de cero y viceversa.

Como último punto para la aplicación del método de búsqueda local, se construyó una solución inicial con la cual se muestra la clara desventaja de usar esta técnica: que es quedar atrapado en óptimos locales pobres. Por ende, la solución inicial está dada por $s_0 = (1, 1, 0, 1, 0, 1, 0, 1, 1, 0)$ con valor en la función objetivo de $f(s_0) = 934$. Para construir esta solución inicial se buscó que ésta fuera un óptimo local, mas no global (el cual está dado por $s^* = (1, 0, 1, 1, 1, 1, 1, 1, 1, 0)$ con $f(s^*) = 1074$). Para lograr esto, se aseguró que s_0 fuera una solución (distinta a la óptima) en la que no sea posible incluir más elementos a la mochila, pues de hacerlo sobrepasarían la capacidad de ésta y la solución dejaría de ser factible; y de esta forma, no se podrá llegar al ótimo global usando búsqueda local por la manera en la que esta técnica opera. A continuación se explicará el argumento anterior.

Recordamos que el heurístico de búsqueda monótona empieza con una solución inicial e iterativamente se va modificando la solución actual con una pequeña perturbación de ésta (cambiar el valor de una entrada) hasta que ya no se pueda encontrar soluciones mejores a la actual.

Notamos que $w_i > 0 \ \forall i \in \{1, ..., 10\}$, y como el problema es a maximizar, nos conviene que $x_i = 1 \ \forall i \in \{1, ..., 10\}$ (lo cual no se puede por la capacidad máxima de la mochila). Con esta observación podemos deducir que, usando búsqueda local, no serán aceptados los vecinos de s_0 que cambian alguna de las entradas que tienen el valor de uno; pues el valor en la función objetivo de estos vecinos será menor al de s_0. Además, la capacidad que se ocupa de la mochila con s_0 es de 675. Usando lo anterior, como la capacidad máxima de la mochila es de 700, para que un vecino de s_0 se acepte como solución actual debe de agregar un elemento nuevo que tenga un costo menor o igual a 25. En otras palabras, la única forma en la que búsqueda local se mueva de s_0 es mediante un vecino de ésta que cambie una entrada en donde se tiene el valor cero y que el costo respectivo a esta entrada sea menor o igual a 25. Lo cual no pasa pues el peso p_i es mayor a 25 $\forall i \in \{1, ..., 10\}$.

Todo lo anterior prueba que s_0 es un óptimo local, pues ninguno de sus vecinos mejora el valor en la función objetivo de s_0. Es decir, s_0 es de entre sus vecinos la mejor solución, y como consecuencia, la solución final será la inicial. Con este ejemplo se muestra que la solución final del heurístico clásico de búsqueda monótona depende de la solución inicial y que esta técnica tiende a quedarse atrapada en óptimos locales.

Las desventajas claras que se presentan al utilizar un heurístico de búsqueda local abrieron camino a procedimientos para evitar estas desventajas, como estrategias para saber qué vecino escoger en cada iteración, modificar la estructura de las vecindades para que la

forma de moverse en el espacio de soluciones sea inteligente, etc. Justo estas estrategias entran en la categoría de los metaheurísticos.

2.2. Metaheurísticos

Los metaheurísticos surgen cuando no se puede resolver satisfactoriamente un problema mediante heurísticos o métodos exactos. Pues estos dos últimos generalmente no resultan ser eficientes para una variedad de problemas, como los problemas NP-Duros, que en la práctica son bastantes. El objetivo de un metaheurístico, aunque no garantiza el óptimo o una solución cercana a éste (y en muchos casos ni una simple solución puede ser garantizada), es proveer de soluciones cercanas al óptimo la "mayoría" de las veces [2]. Aunque todavía no hay una definición común aceptada de un metaheurístico, se recopiló información de [1] y [2] para poder dar las principales características de un metaheurístico y la definición más completa y apropiada para nosotros.

Lo que distingue a un metaheurístico de otras estrategias son las siguientes propiedades:

• Los metaheurísticos son procesos que "guían" el proceso de búsqueda.
• El objetivo de un metaheurístico es explorar eficientemente el espacio de búsqueda en busca de soluciones cercanas al óptimo.
• Las técnicas de estos algoritmos van desde búsqueda local hasta procesos complejos de aprendizaje.
• Un metaheurístico es un método de aproximación y usualmente no determinista.
• Pueden usar mecanismos para no quedar atrapados en áreas confinadas del espacio de búsqueda.
• Los metaheurísticos no son estrategias específicas para un sólo problema.
• Los metaheurísticos más avanzados en la actualidad usan la búsqueda basada en experiencia para moverse en el espacio de soluciones factibles.

De esta forma obtenemos la siguiente definición para un metaheurístico:

Definición 2.2 (Metaheurístico)**.** Un metaheurístico es un proceso iterativo de alto nivel que guía y modifica las operaciones de heurísticos adyacentes para así, producir de manera eficiente soluciones de alta calidad. Éste puede manipular una sola solución o una colección de soluciones en cada iteración. Los heurísticos subyacentes pueden ser procedimientos de alto (o bajo) nivel, o una búsqueda local básica, o un método constructivo.

Es natural, a partir de esta definición, proseguir con la clasificación de los metaheurísticos.

2.3. Clasificación

Existen varias maneras de clasificar a un metaheurístico dependiendo de sus características. Por ejemplo, se puede distinguir a un metaheurístico por sus orígenes, es decir, si fue

inspirado en la naturaleza o no [1]. Entre los metaheurísticos que fueron creados basados en la naturaleza se encuentran los algoritmos hormiga y algoritmos genéticos, y entre los que no están basados en la naturaleza se encuentra búsqueda tabú. Otra forma de clasificar un metaheurístico se da en cómo usa la función objetivo, en otras palabras, si deja la función objetivo "como es" durante todo el proceso, o si la va modificando. Entre estos últimos se encuentra el metaheurístico de búsqueda local guiada [1]. También se puede distinguir a un metaheurístico por el número de soluciones con las que se trabaja al mismo tiempo; es decir, metaheurísticos basados en la población y aquellos basados en la búsqueda de un sólo punto.
De entre todas las maneras de clasificar a un metaheurístico se escogió esta última pues es la que permite una clasificación más clara de los metaheurísticos [1].

Metaheurísticos basados en la población
Estos algoritmos trabajan con varias soluciones al mismo tiempo y realizan procesos de búsqueda que describen la evolución de un conjunto de puntos en el espacio de soluciones factibles. Un ejemplo de éstos es el algoritmo de la colonia de hormigas, que en resumen, busca los mejores caminos o rutas en grafos.

Metaheurísticos basados en la búsqueda en un sólo punto
Estos metaheurísticos, también conocidos como métodos de trayectoria, tienen la propiedad de definir una trayectoria en el espacio de soluciones factibles durante el proceso de búsqueda. Éstos engloban a los metaheurísticos basados en búsqueda local. Dentro de éstos se encuentra Recocido Simulado (pues está basado en búsqueda local) y búsqueda tabú.

2.4. En Resumen

Cuando se empezó a trabajar con problemas de optimización para los cuales un método exacto no era suficiente para resolverlos, surgieron los heurísticos y los metaheurísticos. Un heurístico es una estrategia inteligente para resolver un problema eficientemente. En otras palabras, se estudia la información que se tiene del problema a resolver (como su estructura) y con base en ésta, se modifica la manera en la que se resuelve el problema de tal forma que se obtienen soluciones de alta calidad con recursos (usualmente tiempo) razonables. Aunque en la práctica son evidentes las ventajas de trabajar con un heurístico a comparación de un método exacto, aún así siguen existiendo problemas, o instancias de éstos, para los cuales los heurísticos suelen dar un desempeño insatisfactorio; aquí es cuando surgieron los metaheurísticos. Los metaheurísticos son estrategias de alto nivel que guían heurísticos subordinados para explorar de forma eficiente el espacio de soluciones factibles. De esta forma, un metaheurístico va "más allá" de la solución obtenida por un heurístico clásico. Por esto, los metaheurísticos han sido, y seguirán siendo, una herramienta de cajón para resolver problemas que llegan a ser complicados y para los cuales los métodos exactos y de aproximación no son eficientes.

Capítulo 3

Recocido Simulado

Desde que se introdujo Recocido Simulado en 1983 como técnica para resolver problemas de optimización, este metaheurístico tuvo un gran auge en las siguientes décadas pues es muy simple de adaptar, fácil de entender, y ampliamente aplicable. Desde la aparición de esta técnica, una variedad de investigadores de distintas ramas experimentaron con ésta para la resolución de sus propios problemas mostrando resultados satisfactorios, haciendo de Recocido Simulado el principal exponente de los metaheurísticos de búsqueda global. Por estas razones Recocido Simulado ha sido tema de estudio y una gran herramienta para resolver problemas. Este método es una variante del heurístico conocido como búsqueda local o mejoramiento iterativo, con el cual puede haber desventajas al aplicarlo pues es muy fácil quedar atrapado en óptimos locales. Sin embargo, Recocido Simulado muestra una alternativa para evitar este problema: aceptando, de manera controlada, soluciones que de alguna forma son peores que la anterior. Esto permite que Recocido Simulado evite quedar atrapado en óptimos locales y es justo lo que lo hace un metaheurístico de búsqueda global.
No obstante, como este método es muy general, para hacerlo que funcione adecuadamente hay una serie de decisiones que se deben tomar para poder obtener una solución cercana a la óptima. Estas decisiones dependen de la experiencia, la estructura del problema, y de la prueba y error [15]; para esto, la teoría y resultados empíricos ofrecen consejos para poder tomar estas desiciones de manera eficiente.
Antes de presentar Recocido Simulado, se hablará un poco de dónde viene esta técnica para comprender de una mejor manera los cimientos de este metaheurístico. Después, se introducirá Recocido Simulado junto con las desiciones que se deben tomar antes de aplicarlo a cualquier problema.

3.1. La historia detrás de Recocido Simulado

Recocido Simulado, como método para resolver problemas de optimización, fue introducido por Kirkpatrick *et al.* en 1983. Sin embargo, desde 1953 se presentaron las ideas esenciales de este metaheurístico con un algoritmo que simula el proceso de un sólido en un baño de calor para llegar al equilibrio térmico creado por Metropolis *et al.*. A este

proceso se le conoce como recocido.
La idea de recocido se puede entender esencialmente en dos pasos [6]:

• Calentar el material más allá del punto en el que se empieza a derretir.

• Enfriar *cuidadosamente* hasta llegar a un estado estable del sólido.

Este proceso se utiliza para cambiar las propiedades de un material, y si la temperatura inicial no es suficientemente alta y/o la etapa de enfriamento no se hace de manera adecuada (decreciendo la temperatura de forma rápida), el material resultante mostrará imperfecciones [15]. El ejemplo más común es el recocido del vidrio, el cual se aplica para mejorar las propiedades de éste, como su resistencia. Si el proceso no se hace adecuadamente, el material se puede romper fácilmente o pueden quedar imperfecciones, como burbujas en el vidrio.
Es importante señalar que en la etapa en la que se incrementa la temperatura del material, las partículas de éste tienen prácticamente plena libertad de movimiento, mientras que en la etapa de enfriamento, las partículas se van acomodando de una forma estructurada y van perdiendo la libertad para moverse quedando en un estado *congelado*.
A este algoritmo que creó Metropolis *et al.* en 1953 se le conoce como algoritmo de Metropolis, el cual simula este proceso de Recocido de materiales para llegar al equilibrio térmico [6].

El algoritmo de Metropolis empieza con un estado i del sólido con energía E_i, luego se genera un nuevo estado j, con energía E_j, el cual se crea a partir de hacerle una pequeña perturbación al estado i, como por ejemplo, el desplazamiento de una partícula. Ahora, si $E_j - E_i \leq 0$, se acepta a j como el nuevo estado. Si por el contrario, $E_j - E_i > 0$, j se acepta como el nuevo estado con una probabilidad dada por

$$exp\left(\frac{E_i - E_j}{k_B T}\right) \tag{3.1}$$

Donde T denota la temperatura del sistema y $k_B > 0$ es una constante física llamada constante de Boltzmann. A la decisión tomada con probabilidad dada por la expresión 3.1 se le conoce como el criterio de Metropolis.

De este algoritmo podemos notar que, cuando $E_j - E_i > 0$, tenemos que $E_i - E_j < 0$ y así, $\frac{E_i - E_j}{k_B T} < 0$. Además;

• Si T es muy grande, $\frac{E_i - E_j}{k_B T} \longrightarrow 0$ y así $exp\left(\frac{E_i - E_j}{k_B T}\right) \longrightarrow 1$

• Si T es muy chica, $\frac{E_i - E_j}{k_B T} \longrightarrow -\infty$ y así $exp\left(\frac{E_i - E_j}{k_B T}\right) \longrightarrow 0$

Esta observación nos muestra que la probabilidad con la que se aceptarán deterioraciones

será muy alta cuando la temperatura sea grande y muy baja cuando la temperatura sea pequeña.
Así, después de treinta años de que se publicara este algoritmo, Kirkpatrick *et al.* mostraron, independientemente, que el algoritmo de Metropolis se puede aplicar a la resolución de problemas de optimización haciendo un mapeo de los elementos del proceso de recocido, a los elementos de un problema de optimización. Este mapeo se puede ver en la Tabla 3.1 [4].

Simulación Termodinámica	Optimización Combinatoria
Estados del sistema	Soluciones factibles
Energía	Valor de la función de costo
Cambio de estado	Solución vecina
Temperatura	Parámetro de control
Estado congelado	Solución heurística

Tabla 3.1. Equivalencia entre elementos del algoritmo de Metropolis y los elementos de un problema de optimización.

Cabe mencionar que con este mapeo el algoritmo de Metropolis está pensado para un problema a minimizar; acepta inmediatamente vecinos que hacen decrecer el valor en la función objetivo y acepta, con cierto control, vecinos que incrementan la función objetivo. Con esta analogía, el metaheurístico Recocido Simulado se puede ver como una secuencia de algoritmos de Metropolis. Además, cualquier implementación de búsqueda local se puede convertir en una implementación de Recocido Simulado; eligiendo aleatoriamente las soluciones vecinas, siempre aceptando mejoras en la función objetivo y permitiendo deterioraciones en ésta de forma limitada mediante el criterio de Metropolis [15].
Notamos que la probabilidad que usa el criterio de Metropolis (expresión 3.1) va cambiando conforme el algoritmo va avanzando, de esta forma, mientras la temperatura va cambiando, la aceptación de soluciones que deterioran el costo de la función objetivo también va cambiando; al principio del algoritmo, teniendo temperaturas altas, será muy probable que se acepten soluciones que deterioran la función de costo, lo cual permite escapar de óptimos locales. Mientras que cuando el algoritmo está por terminar, teniendo temperaturas bajas, ya es poco probable que se acepten deterioraciones en la función de costo y a esta altura, sólo se aceptarán soluciones que mejoren esta función, es decir, en este punto Recocido Simulado es, prácticamente, un heurístico de búsqueda local.
Es importante hacer notar que no hay una limitación en cuanto al tamaño de las deterioraciones en la función objetivo y la aceptación de éstas [6]. En Recocido Simulado, deterioraciones arbitrariamente grandes son aceptadas con una probabilidad pequeña, pero positivas.

3.2. Recocido Simulado Básico

Como ya se explicó anteriormente, la principal desventaja de un heurístico de búsqueda local es la tendencia de quedarse atrapado en óptimos locales. Para evitar esto, Recocido Simulado ofrece una manera de solucionarlo; aceptando de forma limitada soluciones que deterioran el costo de la función objetivo sin perder la idea básica de una búsqueda monótona. A continuación se mostrará, de una forma general, el metaheurístico Recocido Simulado a seguir para resolver un problema cuya función objetivo es a minimizar, con S el conjunto de soluciones factibles, f su función de costo y N su función de vecindad [4].

Generar una solución inicial s_0;
Seleccionar la temperatura inicial $T_0 > 0$;
Seleccionar una función de reducción de la temperatura α;
Repetir
    Repetir
        Seleccionar aleatoriamente $s \in N(s_0)$
        $\delta = f(s) - f(s_0)$
        Si $\delta < 0$
        entonces $s_0 = s$
        en otro caso
        seleccionar aleatoriamente un valor u con distribución uniforme en el intervalo (0,1);
            Si $u < exp(\frac{-\delta}{T_0})$ entonces $s_0 = s$;
    Hasta que *cuenta iteraciones = nrep*
    Hacer $T_0 = \alpha(T_0)$;
Hasta que *criterio de parada = verdadera*.
s_0 es la aproximación a la solución óptima.

Tabla 3.2. Procedimiento general de Recocido Simulado para un problema cuya función objetivo es a minimizar.

Para que Recocido Simulado quede expuesto de una forma más clara y completa, lo presentamos como un pseudocódigo (tomando en cuenta que se tiene un problema cuya función objetivo es a minimizar, S es el conjunto de soluciones factibles, f su función de costo y N su función de vecindad):

Paso 0. Determinar la solución inicial ($s_0 \in S$), la manera en la que irá disminuyendo la temperatura (α), el criterio de parada (T_{final}), la temperatura inicial (T_0) y el número de iteraciones a realizar antes de bajar la temperatura ($nrep$). Se toma $t := T_0$ y $s := s_0$, e ir al *paso 1*.

Paso 1. Verificar si ya se cumplió el criterio de parada. Si $t \leq T_{final}$, **FIN**. De lo con-

trario, tomar $i = 0$ e ir al *paso 2*.

Paso 2. Verificar si ya se cumplieron las iteraciones a realizar antes de bajar la temperatura. Si $i = nrep$, ir al *paso 5*. De lo contrario, ir al *paso 3*.

Paso 3. Tomar aleatoriamente un vecino s' de s ($s' \in N(s)$) y verificar si s' mejora la función objetivo. Si $f(s') < f(s)$, tomar a s' como la nueva solución actual (hacer $s := s'$), hacer $i := i + 1$ e ir al *paso 2*. De lo contrario, ir al *paso 4*.

Paso 4. Tomar aleatoriamente un valor u con distribución uniforme en el intervalo $(0, 1)$ y verificar si este valor es menor al criterio de metropolis (expresión 3.1). Si $u < exp\left(\frac{f(s)-f(s')}{T_0}\right)$, tomar a s' como la nueva solución actual (hacer $s := s'$), hacer $i := i + 1$ e ir al *paso 2*. De lo contrario, quedarse con s como la solución actual, hacer $i := i + 1$ e ir al *paso 2*.

Paso 5. Disminuir la temperatura con base en α, es decir, tomar a $\alpha(t)$ como la temperatura actual (hacer $t := \alpha(t)$) e ir al *paso 1*.

También se puede explicar Recocido Simulado mediante un diagrama de flujo como se muestra en la Figura 3.2, la cual se encuentra al final de este capítulo.
Notamos que la temperatura no tiene ninguna analogía física pues sólo nos sirve como un parámetro de control, además, como la constante de Boltzmann no tiene ningún significado en los problemas de optimización, se toma $k_B = 1$ [15] y a los factores que determinan la forma en la que se reduce la temperatura, constituyen lo que se conoce como *programa de enfriamento*.
Claramente antes de aplicar Recocido Simulado a un problema en particular, se tienen que tomar varias decisiones y se debe tomar en cuenta la estructura del problema en cuestión. Estas decisiones se dividen en dos categorías; las decisiones genéricas y las decisiones específicas [15]. Las decisiones genéricas tienen que ver con el programa de enfriamento; se manejan factores como la temperatura inicial, la manera y la velocidad en la que ésta se reduce, el criterio de parada y el número de iteraciones a hacer en cada temperatura antes de bajar ésta ($nrep$). Las decisiones específicas tienen que ver con la definición del espacio de soluciones y de vecindades así como la estructura de estas últimas, la función objetivo y la manera en la que se obtiene la solución inicial.
Estas decisiones son importantes a la hora de aplicar Recocido Simulado pues de éstas depende la calidad de la solución metaheurística que se obtiene [4]. Como Recocido Simulado es un método general ampliamente aplicable a problemas de varias ramas, no existen reglas que dicten qué decisiones tomar para que la solución obtenida sea buena, por lo que estas decisiones dependen de la experiencia y de la estructura del problema. Sin embargo, hay consejos y resultados teóricos que sirven de guía para tomar estas decisiones. A continuación se verán más a detalle los dos tipos de decisiones y algunos resultados que

pueden ser útiles a la hora de llevar Recocido Simulado a la práctica.

3.2.1. Decisiones Genéricas

Las decisiones genéricas tienen que ver con los parámetros que definen el programa de enfriamiento del cual depende la velocidad a la que converge Recocido Simulado [6]; la temperatura inicial, la manera y la velocidad en la que se reduce la temperatura, la temperatura final (las condiciones que indican que el sistema ya está *frio*), y el número de iteraciones a realizar en cada temperatura antes que ésta disminuya. Como ya se mencionó, no existen reglas a seguir para tomar estas decisiones, pero lo que sí existen son resultados teóricos y algunos consejos que pueden ayudar a tomar estas decisiones.

• *Manera de reducir la temperatura*
Resultados teóricos basados en cadenas de Markov demostraron que bajando la temperatura de una forma infinitamente lenta, la probabilidad de que se llegue a la solución óptima utilizando Recocido Simulado es de 1. Claramente en la práctica no se podrá reducir la temperatura de una manera infinitamente lenta, ya que en la práctica se utilizan programas de enfriamiento finito. Sin embargo, para que la solución metaheurística esté a una distancia arbitrariamente próxima del óptimo, son necesarios tiempos computacionales de tipo exponencial [4]. Hajek (1988) demostró que para asegurar una solución asintótica al óptimo, se debe reducir la temperatura de acuerdo a la siguiente expresión: $T_k = \frac{\Gamma}{ln(1+k)}$, donde k es el número de iteraciones y Γ es la máxima profundidad necesaria para escapar de cualquier óptimo local (véase la Figura 3.1) [4]. Por lo general Γ no se conoce, pero sirve para dar una idea de cómo debe de estar relacionada la manera en la que se enfría la temperatura y el espacio de soluciones del problema.

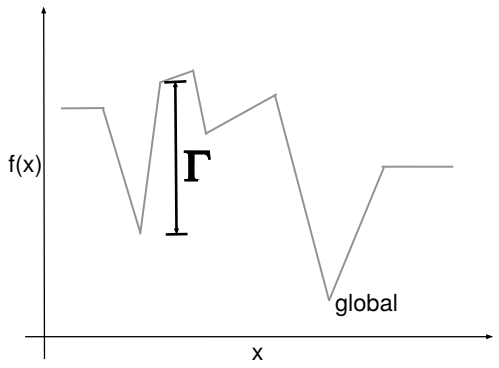


Figura 3.1. Explicación del valor Γ

• *Temperatura inicial*
Como se vio anteriormente, uno de los problemas de los heurísticos de búsqueda local es que la solución final depende totalmente de la solución inicial. Como Recocido Simulado es un metaheurístico de búsqueda global, se espera que la solución metaheurística sea independiente de la solución inicial y para asegurar esto, la temperatura inicial debe ser independiente de la solución inicial y lo suficientemente alta como para permitir casi libremente soluciones vecinas [4]. Aún cuando éstas empeoren el costo de la función objetivo, una temperatura alta es justo lo que dará la posibilidad de escapar de óptimos locales.

• *Velocidad de reducción de la temperatura*
Este factor depende del número de iteraciones a realizar en cada temperatura, lo cual denotaremos como *nrep*, y de la función de reducción de la temperatura $\alpha(T)$. La teoría nos recomienda que antes de bajar la temperatura, el sistema debe estar cerca de un estado estacionario [6]. De esta forma *nrep* debe ser tal, que antes de bajar la temperatura, el sistema esté tendiendo a un óptimo local. La teoría también sugiere que se vaya cambiando *nrep* conforme se va disminuyendo la temperatura; se recomienda pasar mucho tiempo en temperaturas bajas antes de decrecerlas para así asegurar que el óptimo local se alcance, es decir, aumentar *nrep* conforme la temperatura va descendiendo [4].
En los primeros años de Recocido Simulado, se utilizaron diversos programas de enfriamento y al momento de elegir la función de reducción de la temperatura α, se vio una clara inclinación por parte de los investigadores que resolvían problemas reales; para elegir una función de reducción de tipo geométrico $\alpha(t) = at$ con $a \in [0.8, 0.99]$. Evidencias empíricas mostraron que con esta función α, se dieron los mejores resultados [4].

• *Temperatura final*
La temperatura final también es importante pues este factor será el que determine cuando el sistema ya esté *frío*. Teóricamente, la temperatura debe descender hasta 0, sin embargo, en la práctica normalmente el óptimo local final se alcanza bastante antes de que $T = 0$ [4]. Aún así, hay que tener cuidado al elegir la temperatura final, pues si este valor se elige muy cercano a cero, Recocido Simulado se mantendrá bastante tiempo en temperaturas bajas, tiempo que podría aprovecharse mejor en temperaturas altas. Si por el contrario, la temperatura final toma un valor grande, Recocido Simulado pueda que no alcance ningún óptimo local.

Cabe mencionar que técnicamente existen dos tipos de programas de enfriamento; estático y dinámico. En un programa de enfriamento estático los valores para cada factor son fijos, a diferencia de un programa de enfriamento dinámico, en el cual los valores de los factores van cambiando conforme Recocido Simulado va avanzando.

3.2.2. Decisiones específicas

Las decisiones específicas toman en cuenta la función objetivo, cómo se genera la solución inicial, el espacio de soluciones y la estructura de las vecindades. Resultados de Hajek

(1988) apoyan la idea de que la forma en la que se toman estas decisiones influye en la solución final que se obtiene [4].

• *Solución inicial y función objetivo*
Generalmente, la solución inicial se genera de manera aleatoria. Pero como se mencionó antes, no hay reglas estrictas a seguir para aplicar Recocido Simulado y estas decisiones, así como las genéricas dependen de la persona que aplica este método, de la experiencia y la estructura del problema.
Para la función objetivo se hace la observación que, para ahorrar tiempo computacional, es recomendable no calcular el cambio de valor en la función de costo que se pide en cada iteración de Recocido Simulado, en cambio se recomienda calcular la parte relativa que cambió de la solución anterior [4]. Así no se calcula el valor de la función de costo en la solución completa y sólo se hace un cálculo más pequeño.

• *Espacio de soluciones y estructura de vecindades*
En los primeros años de Recocido Simulado, en la teoría se usaba vecindades uniformes y simétricas, es decir, teniendo todas las vecindades del espacio de soluciones del mismo tamaño y que se cumpliera que si $i \in N(j) \Rightarrow j \in N(i)$. Sin embargo, ahora se sabe que basta pedir una condición más suave; se exige que cualquier solución pueda alcanzar a cualquier otra, mediante entornos, a través de una cadena válida de movimientos. Esta propiedad se le conoce como *alcanzabilidad* [4]. Esto se debe tomar en cuenta al momento de definir la estructura del espacio de soluciones y la función de vecindad N. Que se exiga la alcanzabilidad en la estructura de las vecindades tiene sentido pues si las vecindades no cumplen con esta propiedad, como la manera de búsqueda de Recocido Simulado es mediante vecindades, habrá elementos en el espacio de soluciones que no se podrán tomar lo cual muestra una desventaja ya que entre las soluciones que no se podrán tomar, puede que se encuentre el óptimo global o una solución muy cercana a ésta.
Como Recocido Simulado es una variante de la técnica de búsqueda monótona, y como en esta técnica la calidad de la solución resultante se le atribuye a la forma en la que se explora el espacio de soluciones, puede que la calidad de la solución resultante de Recocido Simulado no dependa de la estructura de las vecindades tanto como en una búsqueda monótona; sin embargo, cómo se definan las vecindades es de gran importancia y juegan un papel importante para poder llegar a una buena solución.

3.3. Ejemplo

Para ejemplificar Recocido Simulado y las ventajas que éste tiene a comparación del heurístico de búsqueda local, se resolverá la misma instancia del problema de la mochila binario de 2.1.2, el cual se resolvió usando búsqueda local en 2.1.2.

Para aplicar Recocido Simulado como se ha planteado anteriormente, formularemos el problema cuya función objetivo es a minimizar. Para esta instancia se tienen 10 objetos,

la capacidad de la mochila es de $C = 700$, el peso p_i y el beneficio o utilidad w_i de cada objeto se muestra en la siguiente tabla;

Objeto	1	2	3	4	5	6	7	8	9	10
p_i	100	155	50	112	70	80	60	118	110	55
w_i	-150	-190	-100	-155	-110	-130	-120	-158	-151	-105

Tabla 3.3. Datos del problema de la mochila binaria.

De esta forma, el planteamiento de la instancia a resolver será de la siguiente forma:

$$\text{mín} -150x_1 - 190x_2 - 100x_3 - 155x_4 - 110x_5 - 130x_6 - 120x_7 - 158x_8 - 151x_9 - 105x_{10}$$

$$s.a. 100x_1 + 155x_2 + 50x_3 + 112x_4 + 70x_5 + 80x_6 + 60x_7 + 118x_8 + 110x_9 + 55x_{10} \leq 700$$

$$x_i \in \{0, 1\}, i = 1, ..., 10$$

Antes de aplicar Recocido Simulado, se requiere definir las siguientes cosas;

- La función objetivo
- Las vecindades, la estructura de éstas y del espacio de soluciones factibles
- La solución inicial
- La temperatura inicial, la temperatura final, la velocidad a la que decrecerá la temperatura, y el número de iteraciones a realizar antes de bajar la temperatura

La función objetivo $f(x)$ se tomará del planteamiento, haciendo $f(x) = -150x_1 - 190x_2 - 100x_3 - 155x_4 - 110x_5 - 130x_6 - 120x_7 - 158x_8 - 151x_9 - 105x_{10}$. Ahora definiremos al espacio de soluciones, las vecindades y la estructura de ambas; para esto, consideremos $c = (100, 155, 50, 112, 70, 80, 60, 118, 110, 55)$. Así, el espacio de soluciones factibles S para esta instancia se define como:

$$S = \{s \in \{0, 1\}^{10} \,|\, sc^T \leq 700\}$$

Donde dada $s \in S$, la entrada i de s está asociada con la variable de decisión x_i. Para definir una vecindad $N \subseteq S$ del problema, se explicará cómo se genera un vecino a partir de una solución en S. Sea $s = (s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9, s_{10}) \in S$, para generar un vecino de s se escoge de manera aleatoria una entrada de s y se cambia el valor de esta entrada. Es decir, si en la entrada que se escogió aleatoriamente se tenía un 1, se le asigna un 0 a esa entrada y viceversa. Si al hacer este cambio a s se conserva la factibilidad, el nuevo vector generado s' será aceptado como vecino de s. Si por el contrario, al hacer este cambio se pierde la factibilidad, se tomarán en cuenta los objetos asociados que sí están en la mochila, en otras palabras, nos fijaremos en las entradas de este vector perturbado s' que tienen un 1. Después, se compararán los cocientes $\frac{p_i}{w_i}$ asociados a estas entradas, los

cuales nos dicen lo que aporta cada variable a la función objetivo y tomaremos el máximo de estos cocientes. Finalmente, a la entrada asociada con este máximo se le asignará un 0 y así sucesivamente hasta recuperar la factibilidad y el vector resultante será vecino de s. Lo que se hace, si al perturbar una solución en S se pierde la factibilidad, es fijarse en los elementos que están en la mochila y quitar uno a uno el elemento que menos beneficio aporta a la función objetivo hasta que se recupere la factibilidad. Escogemos el máximo de estos cocientes pues como nuestro problema es a minimizar, la variable que más hace crecer la función objetivo es a la que menos nos conviene asignar el valor de uno. De esta forma, vamos quitanto los elementos de la mochila que nos conviene acercándonos al óptimo de forma eficiente.

Tomaremos $s_0 = (1, 1, 0, 1, 0, 1, 0, 1, 1, 0)$ con $f(s_0) = -934$ como la solución inicial, la misma que se usó para ejemplificar búsqueda local. Finalmente, para los parámetros del programa de enfriamento, se tomarán los siguientes valores; temperatura inicial $T_0 = 1000$, temperatura final $T_{final} = 10^{-}4$, velocidad en la que se disminuirá la temperatura $\alpha(t) = (0.99)t$ y el número de iteraciones a hacer antes de bajar la temperatura $nrep = 100$. Estos valores se basaron en [14] y se fijaron después de jugar con ellos y hacer varios experimentos.

Para este problema, sabemos que la solución óptima está dada por $s^* = (1, 0, 1, 1, 1, 1, 1, 1, 1, 0)$ con valor de función objetivo $f^* = -1074$ y que el óptimo local al que se llegó usando mejoramiento iterativo fue $s = (1, 1, 0, 1, 0, 1, 0, 1, 1, 0)$ con $f(s) = -934$.
Como sabemos, la ventaja que tiene Recocido Simulado sobre la búsqueda monótona es que este metaheurístico nos permite escapar de óptimos locales, permitiendo así, la oportunidad de encontrar soluciones más cercanas (o iguales) al óptimo.
La solución a la que se llegó usando Recocido Simulado fue $s' = (1, 0, 1, 1, 1, 1, 1, 0, 1, 1)$ con $f(s') = -1021$. Se hicieron varios experimentos durante los cuales se iban modificando los valores de la temperatura inicial y de $nrep$ con el objetivo de encontrar una solución mejor a s', pero no se llegó a mejor solución que ésta. Sin embargo, este ejemplo sirve para ver cómo funciona Recocido Simulado pues este método escapó del óptimo local s y encontró una solución más cercana al óptimo.

3.4. En Resumen

Recocido Simulado ha ganado fama en las últimas décadas pues es un metaheurístico que es a la vez simple y ampliamente aplicable. La idea básica de este método es la de búsqueda local, con la diferencia que Recocido Simulado acepta, con cierto control, soluciones que deterioran la función objetivo y esto es justo lo que hace de Recocido Simulado, un metaheurístico de búsqueda global.
La misma aplicabilidad de Recocido Simulado en varias ramas hace que este método sea muy general y que haya una serie de decisiones a tomar antes de aplicarlo. Estas decisiones son de gran importancia pues de éstas depende la calidad de la solución y la velocidad a

la que se llega a ésta. Estas desiciones están basadas en el tipo de problema con el que se está trabajando, en la experiencia que se tiene trabajando con este metaheurístico y en la prueba y error.
No obstante, Recocido Simulado ha servido como una gran herramienta para resolver diversos problemas complejos.

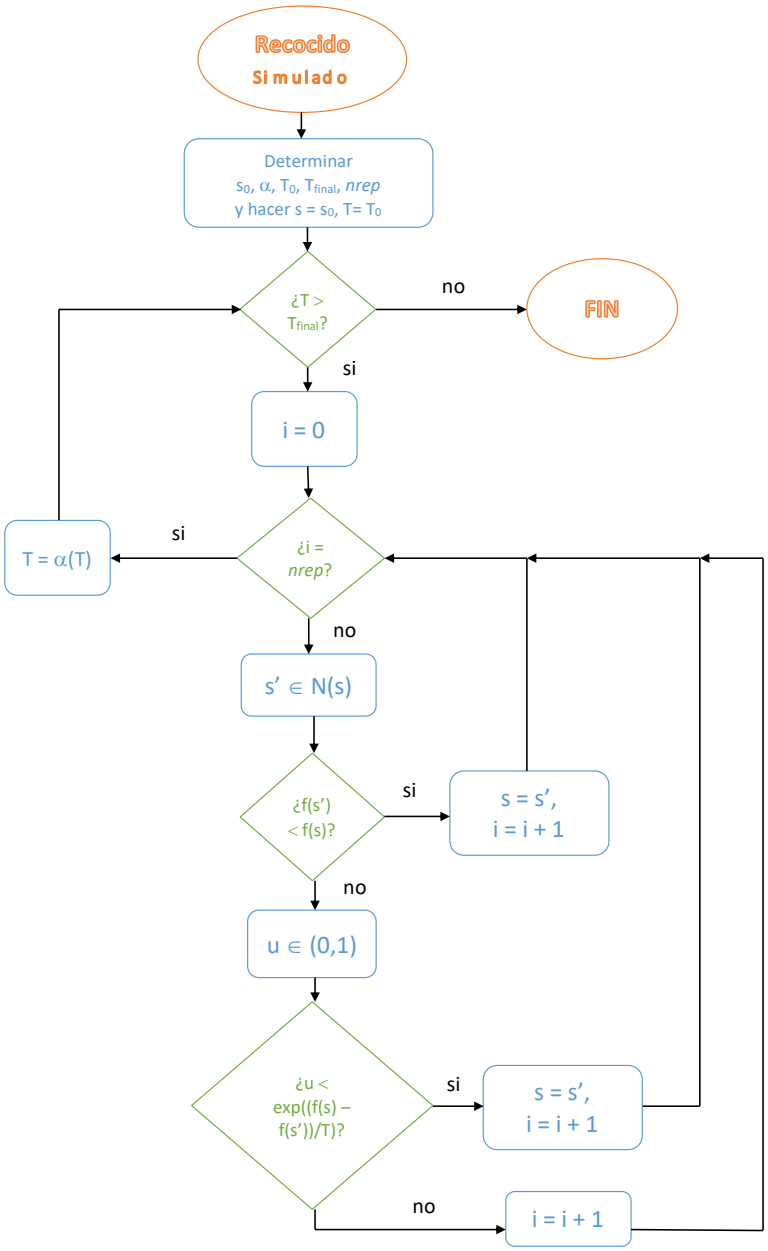


Figura 3.2. Diagrama de flujo creado a partir de [4] que explica el proceso de Recocido Simulado tomando en cuenta que s_0 es la solución inicial, T_0 la temperatura inicial, T_{final} el criterio de parada, α la forma en la que se decrece la temperatura y $nrep \in \mathbb{N}$ el número de iteraciones a realizar antes de bajar la temperatura.

Capítulo 4

El Problema de Empaquetamiento usando Recocido Simulado

El problema de empaquetamiento es uno de los problemas más famosos, esto se debe al desafío científico que conlleva resolverlo y a la amplia variedad de aplicaciones que tiene en la industria. En particular para este proyecto, se trabajará con un $PEC2$; en el cual se busca minimizar el radio de un círculo que contenga N círculos de radio fijo sin que éstos se traslapen.
Debido a que este problema pertenece a los problemas $NP - Duros$, sólo de pocas instancias de éste se conoce la solución óptima y por ende, se requiere más que un algoritmo exacto para poder abordarlo. Por esto, se utilizará el metaheurístico Recocido Simulado para resolverlo ya que éste es uno de los grandes exponentes de búsqueda global, es fácil de entender y de aplicar.
Se tomó como base [14] al trabajar con este problema usando Recocido Simulado, sin embargo, a lo largo del proyecto se fueron modificando parámetros y estrategias con base en los resultados que se iban obteniendo buscando mejorarlos. En este capítulo se verá a mayor detalle el problema con el que se trabajó, cómo se aplicó Recocido Simulado para resolverlo, se expondrán los resultados obtenidos y se muestra la comparación con los mejores resultados registrados en [17].

4.1. Planteamiento

En este proyecto se trabajó con N círculos de radio $r_i = i$, para cada $i = 1, ..., N$, los cuales se buscan acomodar de tal forma que, sin traslaparse, el radio R del contenedor circular sea mínimo.
Se denota al centro del círculo i como (x_i, y_i) y tomando, sin pérdida de generalidad, al origen como el centro del contenedor circular, el radio R del contenedor se define a partir de la posición de los N círculos de la siguiente manera [14]:

$$R := max_i \left\{ \sqrt{x_i^2 + y_i^2} + r_i \right\}$$

Lo que se hace para obtener el radio del contenedor circular es calcular la suma de la distancia de los N centros con sus respectivos radios, y tomar el máximo de éstos. En otras palabras, se toma el círculo más lejano al origen y a partir de éste se hace el contenedor circular. Por definición se tiene que los N círculos están totalmente adentro del contenedor circular, lo único que falta para satisfacer las restricciones del problema de empaquetamiento y poder trabajar con soluciones factibles, es cuidar el traslape entre los N círculos. Para trabajar con esto, se recurre a la función penalizadora P definida en la ecuación 4.1 [14]. Donde $d_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$ es la distancia euclidiana entre los centros de los círculos i y j.

$$P(i,j) = \begin{cases} r_i + r_j - d_{ij} & \textit{si los círculos } i \textit{ y } j \textit{ se traslapan} \\ \\ 0 & \textit{en otro caso} \end{cases} \tag{4.1}$$

Notamos que $P(i,j) \geq 0 \; \forall \; i,j \in \{1, ..., N\}$; pues si hay dos círculos i y j que se traslapen, $r_i + r_j > d_{ij}$, por lo que $P(i,j) = r_i + r_j - d_{ij} > 0$. Y si $r_i + r_j \leq d_{ij}$, los círculos i y j no se traslapan y $P(i,j) = 0$.
La función P nos da la cantidad de traslape que hay entre dos círculos. La Figura 4.1 ilustra de una forma clara la información que nos da la función P.

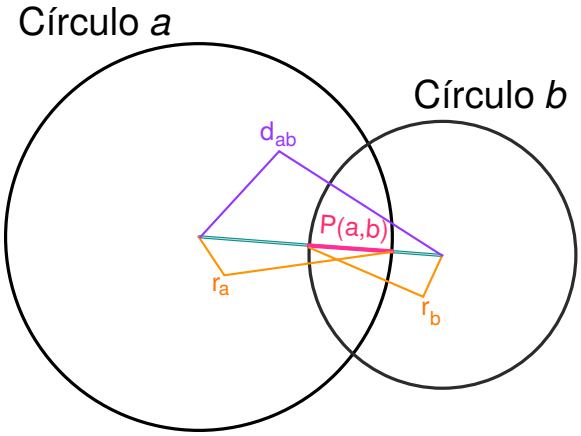


Figura 4.1. Ejemplo de cómo funciona la función P con dos círculos a y b con radios r_a y r_b respectivamente.

Notamos que la función P sólo mide cuánto se traslapan dos círculos; para medir qué tanto hay de traslape en un arreglo de N círculos se recurre a la siguiente expresión [14]:

$$\sum_{i=1}^{N-1} \sum_{j=i+1}^{N} P(i,j) \tag{4.2}$$

La expresión 4.2 obtiene el traslape que hay entre N círculos de la siguiente forma: calcula el traslape que hay en círculos de dos en dos sin repetición, usando la función P, y los suma todos. En total, esta expresión tiene $\frac{N(N-1)}{2}$ sumandos.

Para que una solución sea factible se pide que la expresión 4.2 sea igual a cero; es decir, que no haya traslape alguno entre los círculos que se buscan acomodar de una forma óptima. Así, el planteamiento del problema con el que se trabajará en este proyecto queda de la siguiente manera:

$$\begin{aligned} \text{mín } R &= max_i \left\{ \sqrt{x_i^2 + y_i^2} + r_i \right\} \\ sujeto \quad a & \quad \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} P(i,j) = 0 \end{aligned}$$

De esta forma, la condición que pide que los N círculos estén comprendidos en el contenedor circular está incluida en la función objetivo, y la restricción nos asegura que en el arreglo no hay traslapes entre los N círculos.
Sabemos que antes de resolver este problema usando Recocido Simulado, se requiere determinar un conjunto de factores: el espacio de soluciones factibles, el programa de enfriamento, cómo se genera la solución inicial y la forma de generar vecinos; los cuales se definirán a continuación.

4.2. Preparación para aplicar Recocido Simulado

4.2.1. Espacio de Soluciones Factibles

Como el contenedor depende de la posición de los N círculos, una solución queda determinada por las coordenadas de los centros de los respectivos círculos. Así, una solución s se puede representar como una matriz de la siguiente manera [14]:

$$s = \begin{bmatrix} x_1 & x_2 & \dots & x_i & \dots & x_N \\ y_1 & y_2 & \dots & y_i & \dots & y_N \\ r_1 & r_2 & \dots & r_i & \dots & r_N \end{bmatrix}$$

Donde la columna j nos da la información necesaria del círculo j para que éste quede definido: su radio dado por $s_{3j} = r_j = j$ y las coordenadas de su centro dadas por (s_{1j}, s_{2j}). Por lo tanto, el espacio de soluciones factibles S para este problema queda expresado como:

$$S = \left\{ s \in M_{3xN}(\mathbb{R}) | \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} P(s^i, s^j) = 0 \right\}$$

Donde s^i y s^j denotan las columnas i y j respectivamente de la matriz s. Es decir, s^i y s^j representan a los círculos i y j respectivamente definidos por s.

4.2.2. Programa de Enfriamento

Como ya sabemos, el programa de enfriamento está constituido por los siguientes factores: la temperatura inicial T_0, la manera en la que se reduce la temperatura α, la temperatura final T_{final} y el número de iteraciones a realizar antes de reducir la temperatura *nrep*.
Con base en [14], se tomó $\alpha : \mathbb{R} \longrightarrow \mathbb{R}$ como $\alpha(t) = (0.9)t$ y $T_{final} = 10^{-4}$; sin embargo, los factores T_0 y *nrep* fueron distintos para cada N. Se definieron estos últimos después de correr varios experimentos en cada instancia y mediante la prueba y error; los valores de éstos, para cada N, se muestran en la sección de resultados de este capítulo.

4.2.3. Solución Inicial y Vecindades

La solución inicial se genera de forma aleatoria, pero factible; se generan aleatoriamente N puntos en el plano, los cuales servirán como los N centros de los círculos con los que se trabajarán, y se verifica que el arreglo generado a partir de éstos sea factible. Este proceso se hace iterativamente hasta encontrar N puntos en $\mathbb{R}^2$ que definan una solución inicial factible.
Antes de definir las vecindades, es importante destacar que cada vez que se aplica una función de vecindad a una solución factible s, se verifica con la ayuda de la expresión 4.2 que ésta sea factible; es decir, que no haya traslapes en la solución generada a partir de s. Si la solución resultante es factible, se acepta a ésta como vecino de s, de lo contrario, nos quedamos con la última solución factible generada como el vecino de s (la cual es la misma s, pues en principio s es una solución factible).
Una vez teniendo una solución inicial factible, nos empezamos a mover por el espacio de soluciones S de las siguientes dos maneras:

• Forma de generar vecinos - 1
En la primer manera de generar un vecino de una solución $s \in S$, se busca reducir la distancia que hay del origen a los centros (x_i, y_i), para todo círculo $i \in \{1, ..., N\}$. Es decir, acerca a los N círculos al origen al mismo tiempo. Para lograr esto, nos apoyamos de la siguiente función: $V1 : \mathbb{R}^2 \longrightarrow \mathbb{R}^2$; lo que hace esta función es tomar un punto (x, y) en $\mathbb{R}^2$ y lo pasa a sus coordenadas polares (r, θ), en las cuales la primer entrada corresponde a la distancia del punto (x, y) al origen. Para finalizar, se considera el punto $(\left(\frac{5}{6}\right) \cdot r, \theta)$ y se toma como $V1(x, y)$ al punto en coordenadas cartesianas de este último. En otras palabras, $V1$ disminuye la distancia de un punto al origen por $\left(\frac{5}{6}\right) \approx 0.8$. Sabemos que para reducir esta distancia se debe de multiplicar por un valor menor a uno; no se usó un número muy cercano a cero pues esto haría que los círculos se acerquen de una forma brusca al origen, ocasionando traslapes desde la primer iteración en la que se aplique $V1$. Por ende se escogió un número cercano (pero menor) a uno. Matemáticamente la función $V1$ se define como sigue:

$$V1\,(x, y) = \left(\left(\left(\frac{5}{6}\right) \cdot \sqrt{x^2 + y^2}\right) \cdot \cos\left(arctan\left(\frac{y}{x}\right)\right), \left(\left(\frac{5}{6}\right) \cdot \sqrt{x^2 + y^2}\right) \cdot \text{sen}\left(arctan\left(\frac{y}{x}\right)\right)\right)$$

Y la Figura 4.2 ilustra cómo opera la función $V1$.

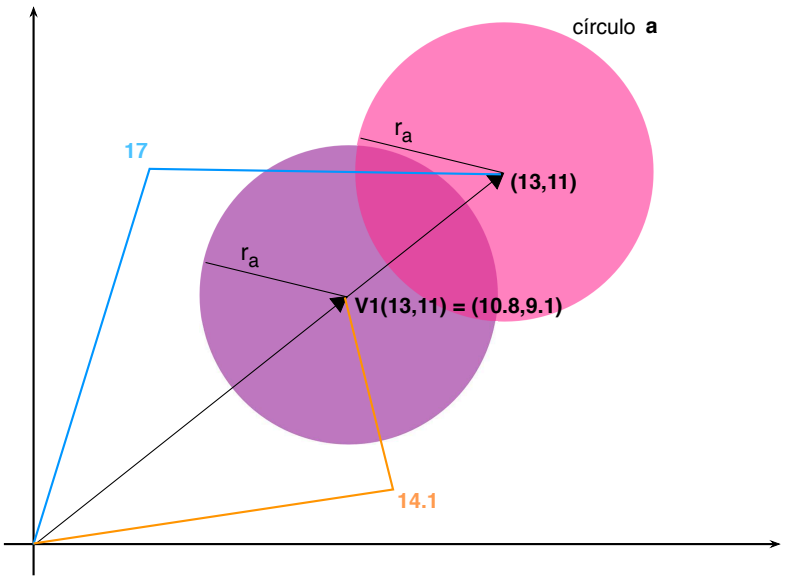


Figura 4.2. Ejemplo de la función $V1$ aplicada al centro del círculo a con radio r_a.

Esta forma de generar vecinos mueve a todos los N círculos de un arreglo al mismo tiempo usando $V1$ (siempre tomando en cuenta la factibilidad de estos arreglos) y se utiliza al principio de Recocido Simulado, es decir, mientras la temperatura actual t cumpla que $T_0 \geq t \geq T_0 - 5$ (donde T_0 es la temperatura inicial).

• Forma de generar vecinos - 2

A diferencia de la forma anterior de generar vecinos, la cual mueve a todos los círculos del arreglo, en esta estrategia sólo se mueve uno de los N círculos. Dado un arreglo de N círculos, se esoge aleatoriamente uno y se modifica la posición éste de la siguiente forma: sea $s \in S$,

$$s = \begin{bmatrix} x_1 & x_2 & \dots & x_i & \dots & x_N \\ y_1 & y_2 & \dots & y_i & \dots & y_N \\ r_1 & r_2 & \dots & r_i & \dots & r_N \end{bmatrix}$$

Se escoge aleatoriamente un círculo $i \in \{1, ..., N\}$, y sean ξ_1, $\xi_2 \in [-r_i \cdot \frac{9}{10}, -r_i \cdot \frac{1}{2}] \bigcup [r_i \cdot \frac{1}{2}, r_i \cdot \frac{9}{10}]$. De esta forma, el vecino de s (s'), cambia únicamente la posición del círculo i de la siguiente manera:

$$s' = \begin{bmatrix} x_1 & x_2 & \dots & x_{i-1} & x_i + \xi_1 & x_{i+1} & \dots & x_N \\ y_1 & y_2 & \dots & y_{i-1} & y_i + \xi_2 & y_{i+1} & \dots & y_N \\ r_1 & r_2 & \dots & r_{i-1} & r_i & r_{i+1} & \dots & r_N \end{bmatrix}$$

Notamos que el conjunto del que se toman los valores ξ_1 y ξ_2 depende del círculo i que se escogió mover; entre más grande es el radio de este círculo, más lejos se podrá mover el

centro de éste. Por ejemplo, si se mueve el círculo 10 (con radio $r_{10} = 10$), los valores ξ_1 y ξ_2 estarán en $[-9, -5] \bigcup [5, 9]$ y si se modifica el círculo 1 (con radio $r_1 = 1$), los valores ξ_1 y ξ_2 estarán en $[-0.9, -0.5] \bigcup [0.5, 0.9]$.
Cabe notar que los valores de ξ_1 y ξ_2 se hicieron depender del radio para que los círculos más grandes se movieran de una forma significativa. Además, los valores $\frac{9}{10}$ y $\frac{1}{2}$ se fijaron después de varios experimentos. La Figura 4.3 nos ayuda a ver de una forma clara cómo se pueden mover los centros de los círculos con esta estrategia para generar vecinos.

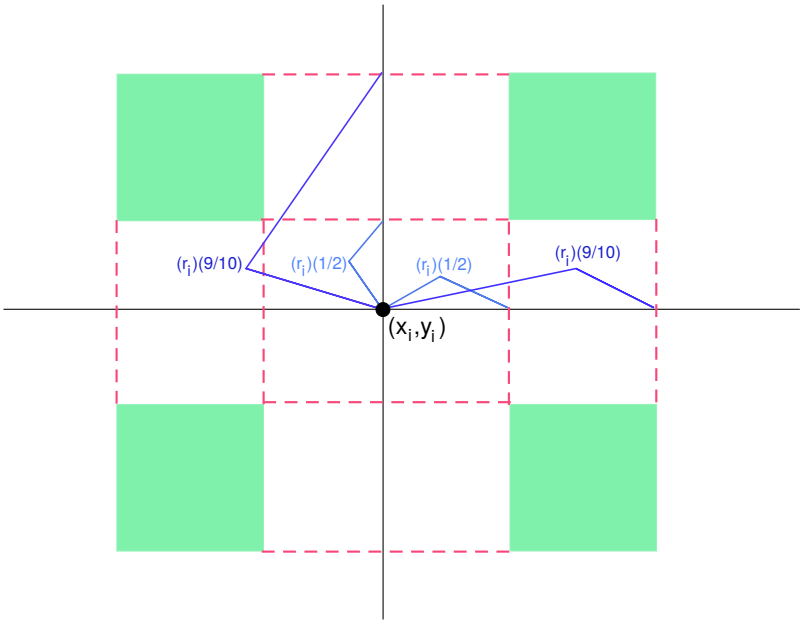


Figura 4.3. El conjunto de color verde son todos los puntos a los que se puede mover el centro del círculo i.

Esta forma de generar vecinos se utiliza en la mayor parte de Recocido Simulado, pues se aplica mientras la temperatura actual t cumpla que
$T_0 - 5 > t \geq \frac{T_0}{4}$.

La estrategia para movernos en el espacio de soluciones S durante la etapa final de este metaheurístico es prácticamente igual a la forma de generar vecinos 2, salvo que en ésta cambia en conjunto del cual se toman los valores ξ_1 y ξ_2. Dicho conjunto para esta estrategia es el siguiente:

$$\left[-r_i \cdot \frac{4}{10}, -r_i \cdot \frac{1}{10}\right] \bigcup \left[r_i \cdot \frac{1}{10}, r_i \cdot \frac{4}{10}\right]$$

Este conjunto es parecido al señalado en la Figura 4.3, pero en este caso, el conjunto modificado está más cerca del punto (x_i, y_i). Es decir, los círculos se moverán de una manera más restringida. Por ejemplo, si se mueve el círculo 10 (con radio $r_{10} = 10$), los

valores ξ_1 y ξ_2 estarán en $[-4,-1] \bigcup [1,4]$ y si se modifica el círculo 1 (con radio $r_1 = 1$), los valores ξ_1 y ξ_2 estarán en $[-0.4,-0.1] \bigcup [0.1,0.4]$. Esta forma de generar vecinos se utiliza mientras la temperatura actual t cumpla que $\frac{T_0}{4} > t \geq T_{final}$ (donde T_{final} es la temperatura final).

Con todo esto, ya se definió lo necesario para poder usar Recocido Simulado y a continuación se muestran los resultados obtenidos para las instancias donde:

$$N \in \{5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 60, 70, 80, 90, 100\}$$

4.3. Resultados

En esta sección se presentan los resultados obtenidos con base a lo visto en este capítulo; a continuación se exponen una tabla e imágenes con los datos de las soluciones obtenidas para cada N. Además, se calcula la desviación porcentual con respecto a las mejores soluciones conocidas hasta ahora [17], la cual nos dice qué tan lejos estamos de éstas y se calcula de la siguiente manera (tomando como $s^* \in S$ la solución obtenida en este proyecto y como $S^* \in S$ la solución encontrada en [17]):

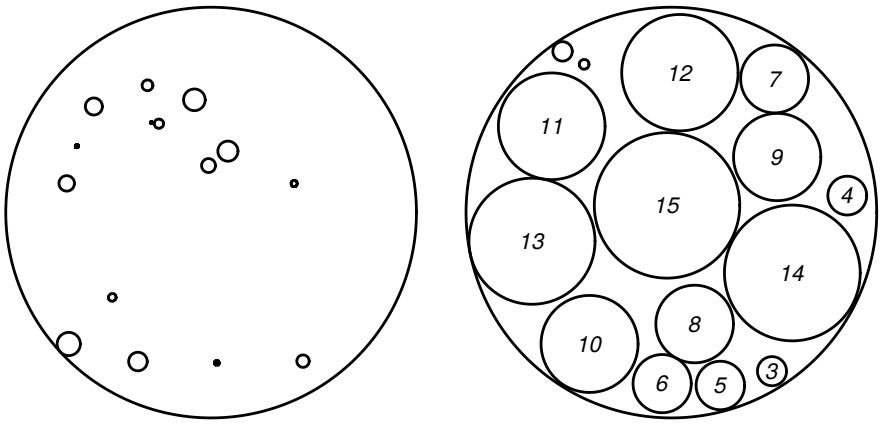
$$\left(\frac{R(s^*) - R(S^*)}{R(S^*)} \right) \cdot (100)$$

Entre más cercana esté la desviación porcentual al 0, se está más cercana a S^*; y en dado caso de que la desviación porcentual sea negativa, quiere decir que se mejoró la solución proporcionada por [17].
Antes de presentar la tabla, definiremos la notación ocupada: N es el número de círculos con los que se trabajó, s_0 es la solución inicial, s^* es la solución heurística final, S^* es la mejor solución conocida proporcionada por [17], $Desv\ \%$ es la desviación porcentual entre las soluciones generadas en este proyecto y las proporcionadas por [17], $nrep$ es el número de iteraciones a realizar antes de bajar la temperatura, y el *Tiempo Total* es el tiempo computacional requerido en segundos para generar la solución obtenida.

N	$R(s_0)$	$R(s^*)$	$R(S^*)$	Desv %	Temp Inicial T_0	$nrep$	Tiempo Total (s)
5	27.8745	9.46368	9.00139	05.1357	1 000	100	30.64
10	116.5787	25.52523	22.00019	16.0227	3 000	500	89.52
15	265.6280	42.34987	38.83799	09.0423	5 000	1000	288.05
20	476.0628	63.66483	58.40056	09.0140	10 000	2 000	1002.53
25	787.2825	87.84015	80.28586	09.4092	10 000	3 000	2197.83
30	1082.9610	116.89880	104.54036	11.8216	25 000	5 000	5497.01
35	1376.3710	147.34420	130.84907	12.6062	40 000	7 000	10344.53
40	1858.8320	175.79730	159.02157	10.5493	55 000	9 000	17697.28
45	2405.9530	208.89300	188.96496	10.5458	80 000	9 000	25060.38
50	2963.7450	244.96820	220.56540	11.0637	100 000	10 000	37603.93
60	4383.0880	325.52480	289.34226	12.5051	300 000	20 000	108324.03
70	6392.6310	401.38700	363.53783	10.4113	600 000	40 000	285120.13
80	7793.3740	488.35310	442.71915	10.3076	800 000	50 000	457200.96
90	9855.2540	583.15810	526.90301	10.6765	1 100 000	70 000	813600.08
100	12159.7100	675.31750	615.86763	09.6530	1 550 000	100 000	1452723.01

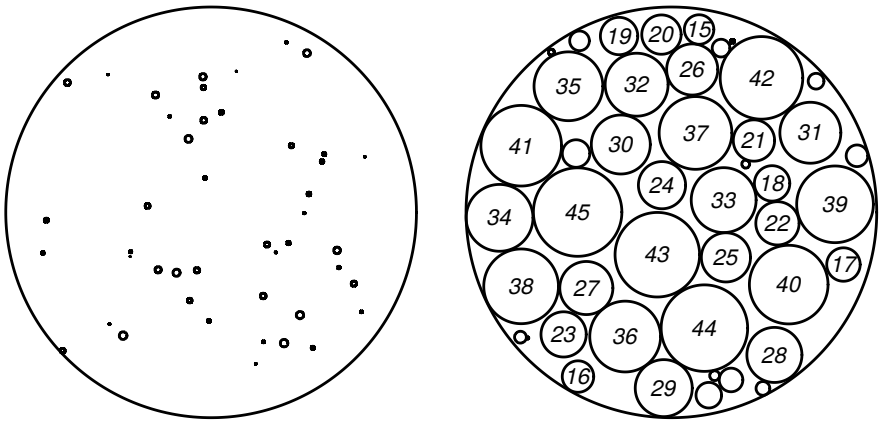
Tabla 4.1. Resultados computacionales.

A continuación, se muestran gráficamente algunas de las soluciones iniciales s_0 (izquierda) y las respectivas soluciones obtenidas s^* (derecha) para cada N respectivamente.



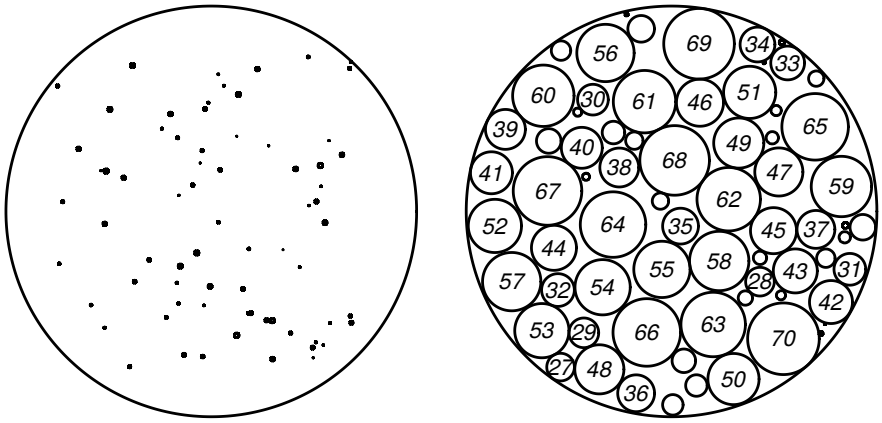
(a) s_0, con $R(s_0) = 265.6280$ (b) s^*, con $R(s^*) = 42.34987$

Figura 4.4. N=15



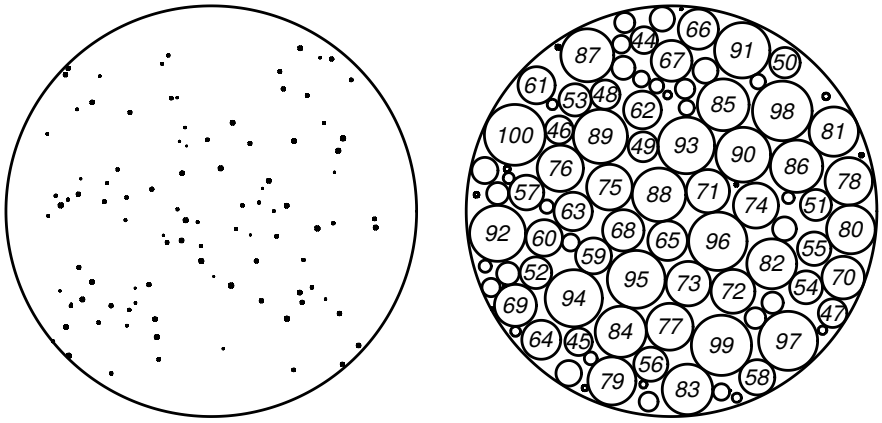
(a) s_0, con $R(s_0) = 2405.9530$ (b) s^*, con $R(s^*) = 208.89300$

Figura 4.5. N=45



(a) s_0, con $R(s_0) = 6392.6310$ (b) s^*, con $R(s^*) = 401.38700$

Figura 4.6. N=70



(a) s_0, con $R(s_0) = 12159.7100$ (b) s^*, con $R(s^*) = 675.31750$

Figura 4.7. N=100

4.4. En Resumen

Debido a la complejidad del problema de empaquetamiento, solo de muy pocas instancias se conoce la solución óptima y por ende se debe de tomar estrategias distintas a la de un algoritmo exacto para poder abordarlo. Como consecuencia de esto, este problema se ha vuelto un desafío muy interesante durante las últimas décadas y ha ganado popularidad tanto como en la industria como en la investigación.
En este proyecto se abordó este problema mediante el famoso metaheurístico Recocido Simulado, el cual ha demostrado ser una gran herramienta para resolver problemas debido a su facilidad de ser aplicado a una gran cantidad de problemas en varias ramas; además de que los recursos necesarios (como el tiempo computacional) para aplicarlo son viables y convenientes a comparación de otros procesos.
Durante el proyecto se fueron cambiando estrategias para poder mejorar las soluciones obtenidas hasta que se logró obtener los resultados expuestos en este capítulo, los cuales se pueden seguir mejorando. Sin duda Recocido Simulado es una gran herramienta para trabajar con problemas desafiantes como lo es el prolema de empaquetamiento.

Conclusiones

El objetivo principal de este proyecto sí se consiguió; se logró resolver satisfactoriamente el problema de empaquetamiento a través de Recocido Simulado programándolo en **R**. Además, se alcanzó a dar una explicación detallada del proceso de Recocido Simulado, de los heurísticos y metaheurísticos, y del programa con el que se trabajó: **RStudio**. También se presentó el código que se usó para resolver el problema y se expuso cada función que se programó. Por lo que este ejemplar también sirve como libro de consulta para entender a fondo Recocido Simulado, qué es un heurístico y un metaheurístico, para tener una idea general de qué es la complejidad computacional, y para saber cómo se puede programar Recocido Simulado en **R**.

En este proyecto se presentó y planteó el problema de empaquetamiento, no sin antes empaparnos un poco de qué es la complejidad computacional; esta última es justo el ŕea que se encarga del estudio de los problemas de optimización y su clasificación. También se adentró a lo que es un heurístico y un metaheurístico, sin perder enfoque en la técnica de nuestro interés: Recocido Simulado. Se mostró un poco de la historia de este metaheurístico, la motivación de éste, y las decisiones que se deben tomar para poder aplicarlo. Además, se hizo la comparación de resultados entre búsqueda local y Recocido Simulado, aplicados a una misma instancia del problema de la mochila (el cual es $NP - Duro$); mostrando la clara desventaja que se tiene al trabajar con búsqueda local, y que para resolver eficientemente problemas de este tipo ($NP - Duros$), se requiere abordarlos de otra forma.

Se mostró cómo se adaptó el metaheurístico Recocido Simulado para poderlo aplicar al problema de empaquetamiento, así como las estrategias que se usaron para poder llegar a las soluciones que se expusieron. Finalmente, se presentó el programa que se utilizó para programar, el manual del usuario para usar éste, y el código que se utilizó para llevar a cabo este proyecto.
Las estrategias que se usaron en este trabajo se fueron modificando con base en los resultados obtenidos, corriendo varios experimentos, y con ideas que iban surgiendo para mejorar los resultados. Un ejemplo de estas estrategias, fue la manera de irse moviendo a través del espacio de soluciones, pues se combinaron tres maneras distintas de escoger vecinos.

Aún así, quedan estrategias para seguir mejorando los resultados expuestos; algunas de

las ideas para darles seguimiento en un futuro y así poder mejorar el rendimiento de Recocido Simulado aplicándolo al problema de empaquetamiento son las siguientes:

• **Mejorar la solución inicial.** En este proyecto, la solución inicial se generó de manera factible pero aleatoria. De esta forma, las soluciones iniciales en algunas instancias no eran nada cercanas a una buena solución (el radio del contenedor de éstas era muy grande); esto implicaba invertir mayor tiempo computacional para reducir notoriamente el radio del contenedor en cuestión. Si se escogía de una manera inteligente la solución inicial (una cuyo radio del contenedor sea razonable), se requeriría de menor tiempo computacional para poder llegar a una buena solución.
• **Cambiar la manera de generar vecinos.** Como sabemos, la manera de moverse en el espacio de soluciones es un factor importante en la calidad de la heurística obtenida. De esta forma, se puede seguir innovando la función vecindad, moverse de una forma más inteligente por el espacio de soluciones factibles, y así generar mejores soluciones.

No cabe duda que resolver el problema de empaquetamiento resultó ser un reto interesante a conquistar, y que Recocido Simulado es una gran herramienta para trabajar.

Apéndices

Programa y Manual del Usuario

El código que se escribió para llevar a cabo este proyecto se programó en **R** usando **RStudio**. A lo largo de este capítulo se busca introducir al usuario al programa **RStudio** y explicar cada función que se programó para:

i) Resolver la instancia del problema de la mochila expuesta en el capítulo 3 usando Recocido Simulado
ii) Resolver el problema de empaquetamiento usando Recocido Simulado
iii) Graficar las soluciones expuestas en el capítulo 4.

.1. Introducción a RStudio

RStudio es un entorno de desarrollo integrado para el lenguaje de programación **R**. Este programa facilita el uso y la escritura de este lenguaje, y está compuesto por cuatro secciones distintas expuestas en la Figura 8. La primer sección corresponde al editor de sintaxis, en esta ventana se escribe lo que es el código; éste se puede escribir directamente en la segunda sección, pero al momento de querer corregir o editar el código se dificulta el manejo de éste. Por ello, se recomienda escribir el código en el editor de sintaxis pues facilita su uso. El segundo sector corresponde a la consola, esta parte es la que se encarga de correr el código y hacer todos los cálculos. La tercer ventana corresponde al espacio de trabajo, esta parte guarda todos los datos que se han generado: como valores, resultados, funciones, entre otros; así como códigos escritos días pasados. La cuarta y última sección corresponde a las herramientas adicionales; éste campo es muy útil pues en él se puede graficar, se muestran los *paquetes* que se han descargado y se puede solicitar ayuda al momento de escribir el código.
Ya que se conoció un poco del programa que se usó al hacer este proyecto, se puede introducir las funciones que se usaron para llevarlo a cabo. A continuación se explicará cada función utilizada y se mostrará, a modo de imágenes, cómo se debe escribir el código en **RStudio** para que las funciones se lleven a cabo. Para fines estéticos, se estará trabajando con la sección correspondiente a la consola y en algunos casos con la sección de herramientas adicionales (para las gráficas).

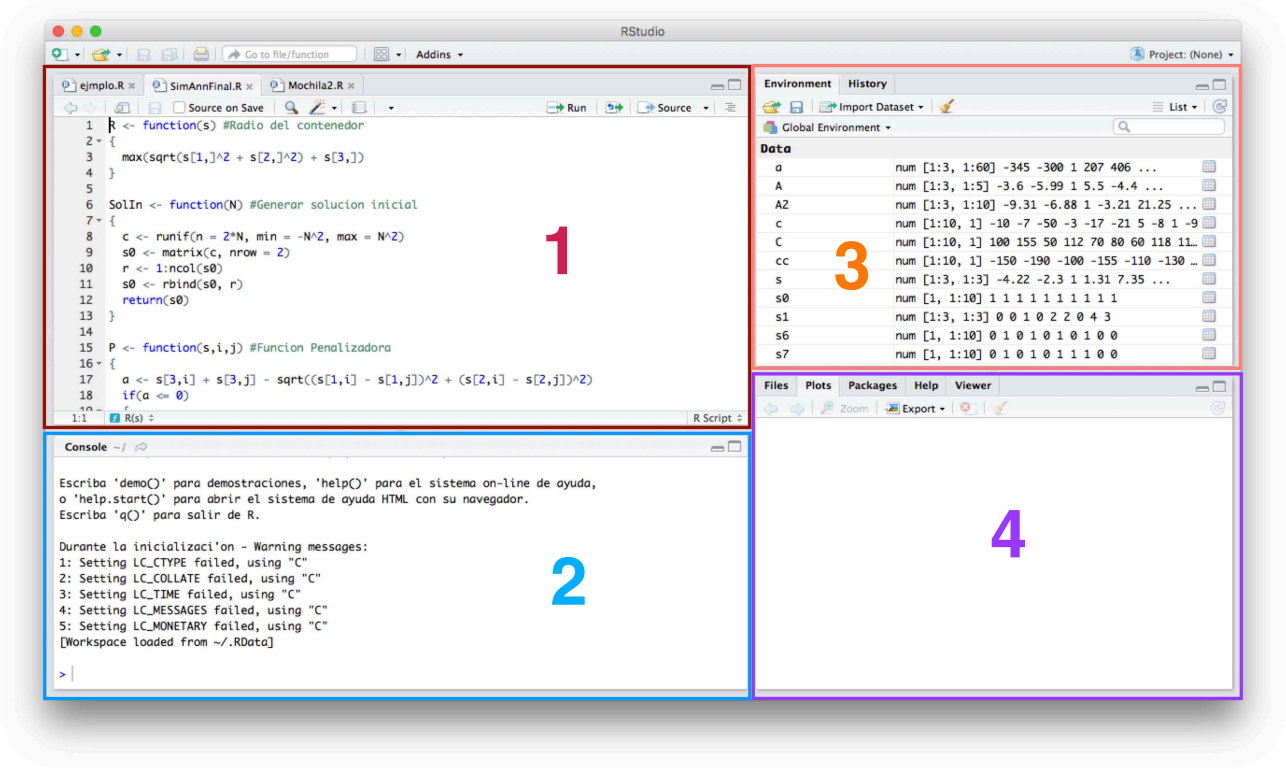


Figura 8. RStudio y sus respectivas secciones; sección 1: editor de sintaxis, sección 2: consola, sección 3: espacio de trabajo, sección 4: herramientas adicionales.

.2. Problema de la Mochila

Sabemos que el problema de la mochila consiste en maximizar el beneficio que pueden aportar a lo más n objetos tomando en cuenta el peso de cada objeto, respetando la capacidad máxima de la mochila. La instancia del problema de la mochila con la que se trabajó y la que se resolvió usando Recocido Simulado es la siguiente:

$$x_i = \begin{cases} 1 \text{ si se incluye el objeto i en la mochila} \\ 0 \text{ en otro caso} \end{cases}$$

$$\text{mín} -150x_1 - 190x_2 - 100x_3 - 155x_4 - 110x_5 - 130x_6 - 120x_7 - 158x_8 - 151x_9 - 105x_{10}$$

$$s.a. 100x_1 + 155x_2 + 50x_3 + 112x_4 + 70x_5 + 80x_6 + 60x_7 + 118x_8 + 110x_9 + 55x_{10} \leq 700$$

$$x_i \in \{0, 1\}, i = 1, ..., 10$$

Para programar la resolución de este problema usando Recocido Simulado se programaron cinco funciones en **R**: f, $rest$, cof, V, $SimAnn$. Las cuales se explicarán con detalle a continuación y se presentará gráficamente lo que se espera ver aplicándolas en **RStudio**.

.2.1. Función f

La función $f : \{0,1\}^{10} \longrightarrow \mathbb{R}$ es la que calcula el valor de la función objetivo correspondiente para cada $s \in \{0,1\}^{10}$. la Figura 9 nos muestra cómo programar esta función y cómo usarla tomando como ejemplo $s = (0,1,0,1,1,1,0,1,0,1)$.

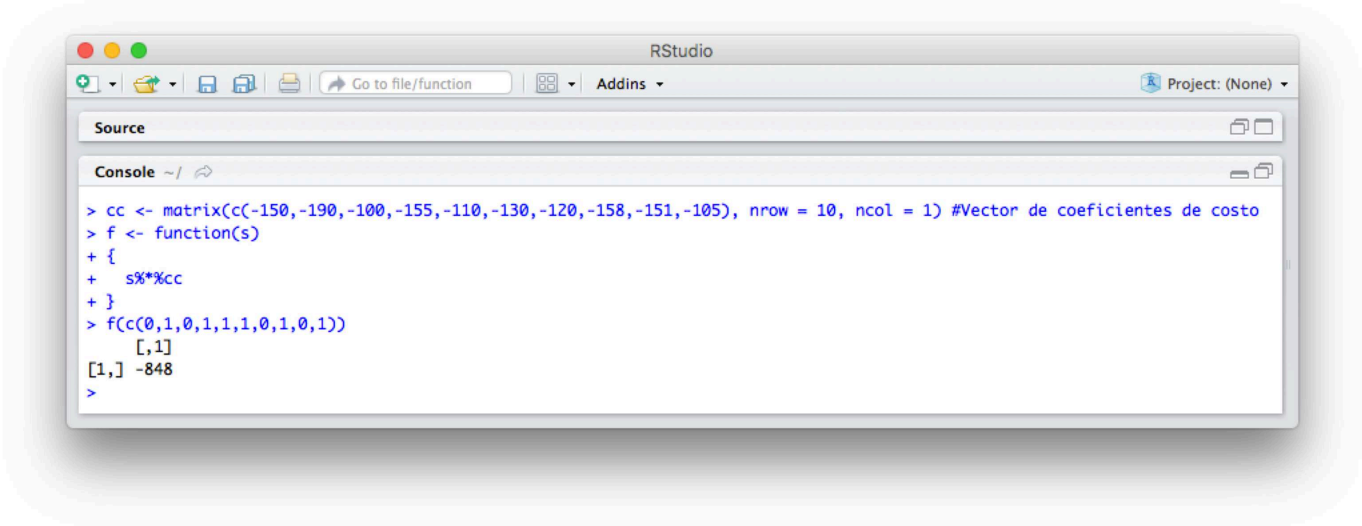


Figura 9. Ejemplo de cómo funciona f en **RStudio**

.2.2. Función *rest*

La función $rest : \{0,1\}^{10} \longrightarrow \mathbb{R}$ es la que asocia a cada $s \in \{0,1\}^{10}$, el peso correspondiente que se lleva en la mochila. La Figura 10 nos muestra cómo programar esta función y cómo usarla tomando como ejemplo la misma s usada en la función f.

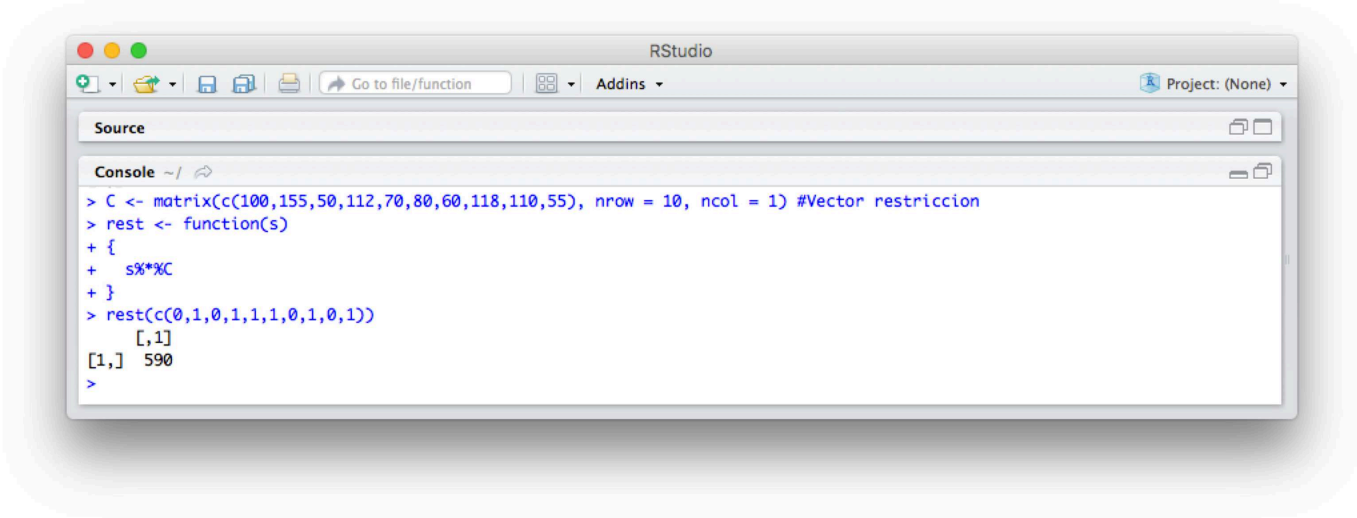


Figura 10. Ejemplo de cómo funciona *rest* en **RStudio**

.2.3. Función *cof*

La función $cof : \{0,1\}^{10} \longrightarrow \{1,...,10\}$ es la que nos ayuda al momento de escoger un vecino. Como se explicó en el Capítulo 3, en el proceso de generar un vecino se puede perder la factibilidad (el peso que se lleva en la mochila sobrepasa su capacidad) y en este caso, dentro de los objetos que están en la mochila, se saca aquél que menos nos conviene llevar (esto se hace sucesivamente hasta recuperar la factibilidad); y justo la función *cof* nos dice qué objeto se saca de la mochila, es decir, esta función nos dice qué entrada nos conviene convertir en 0. La Figura 11 nos muestra cómo programar esta función y cómo usarla tomando como ejemplo la misma s usada en la función f.

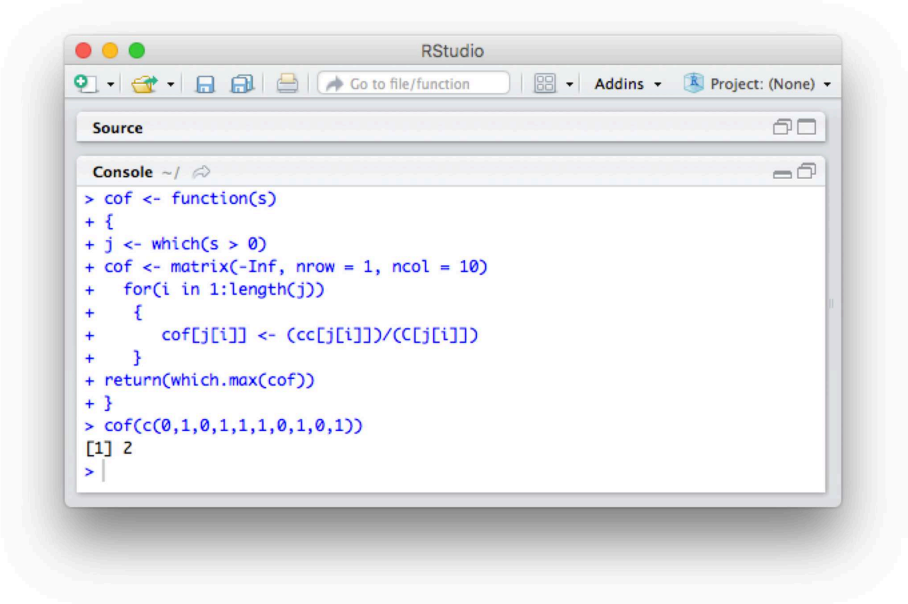


Figura 11. Ejemplo de cómo funciona *cof* en **RStudio**

.2.4. Función V

Tomando $c = (100, 155, 50, 112, 70, 80, 60, 118, 110, 55)$; sea $S = \left\{ s \in \{0,1\}^{10} \,\middle|\, sc^T \leq 700 \right\}$ el espacio de soluciones factibles de este problema, la función $V : S \longrightarrow S$ es la que nos genera un vecino factible a partir de cualquier solución factible. Cabe reiterar que V no sólo genera vecinos factibles, esta función además genera el vecino factible que más nos conviene puesto que va quitando, cuando se requiere, los objetos de la mochila que menor beneficio proporcionan; la función *cof* antes vista es justo la que nos dice qué objeto nos conviene quitar de la mochila, y se hace uso de ésta en la función V. La Figura nos muestra cómo programar esta función y cómo usarla tomando como ejemplo la misma s usada en la función f.

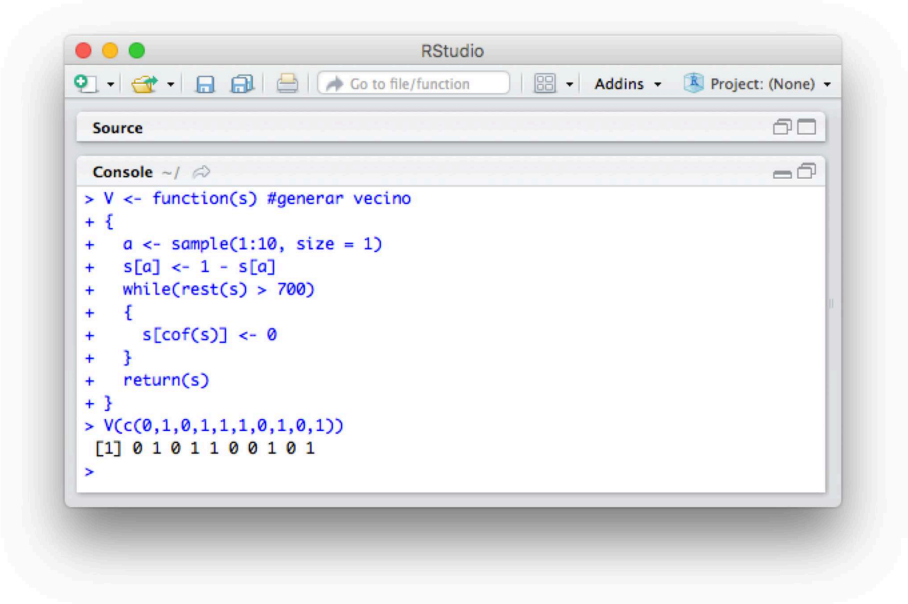


Figura 12. Ejemplo de cómo funciona V en **RStudio**

.2.5. Función *SimAnn*

La función $SimAnn : S \times \mathbb{R} \times \mathbb{R} \times \mathbb{R} \times \mathbb{N} \longrightarrow S \times \mathbb{R}$ es justo el proceso de Recocido Simulado aplicado a esta instancia del problema de la mochila. Para empezar a usar esta función se requiere empezar con un vector con cinco entradas; en la primer entrada, se pone la solución inicial, en la segunda entrada, se pone el factor a que corresponde a la función $\alpha(t) = at$ la cual controla cómo decrece la temperatura, en la tercera entrada, la temperatura inicial, en la cuarta entrada, la temperatura final, y en la quinta entrada, el número de iteraciones por realizar antes de bajar la temperatura ($nrep$).
Al meter estos datos en la función, se hace el proceso de este metaheurístico y como resultado nos da la solución obtenida con el valor correspondiente en la función objetivo. En **R**, este resultado se muestra en forma de lista. La Figura 13 nos muestra cómo programar esta función y cómo usarla tomando como solución inicial la misma s usada en la función f.

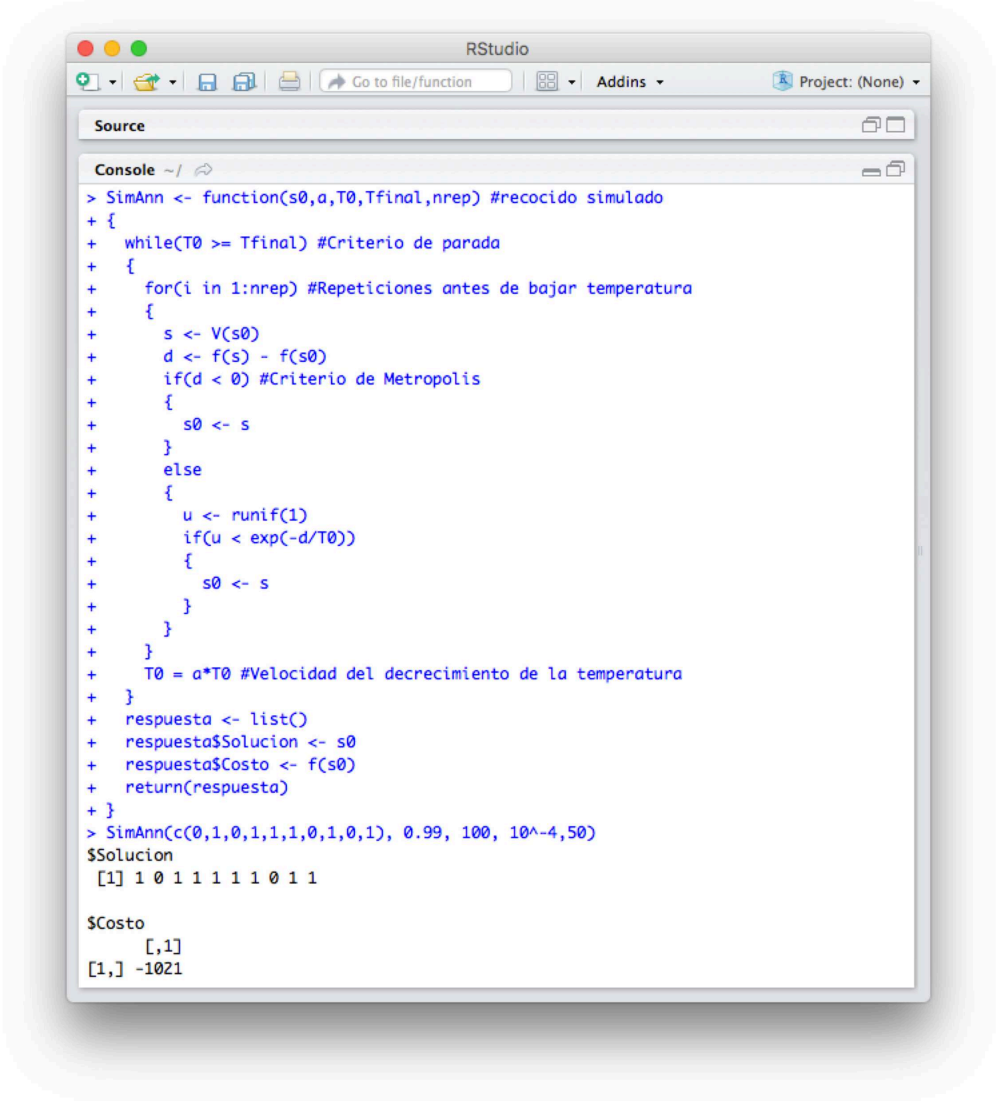


Figura 13. Ejemplo de cómo funciona *SimAnn* en **RStudio**

.3. El problema de empaquetamiento

Recordemos que en el problema de empaquetamiento con el que se trabajó se busca minimizar el radio de un círculo que contenga N círculos de radio fijo sin que éstos se traslapen. El planteamiento para este problema, como se vio en el capítulo 4, es el siguiente:

$$\begin{aligned} \text{mín } R &= max_i \left\{ \sqrt{x_i^2 + y_i^2} + r_i \right\} \\ sujeto \quad a \quad & \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} P(i,j) = 0 \end{aligned}$$

Para poder resolver cualquier instancia de este problema en **R** se programaron diez funciones: R, $SolIn$, P, PP, pol, car, $V1$, $V2$, $V3$, $RecSim$; las cuales se explicarán con detalle a continuación y se presentará gráficamente lo que se espera ver aplicándolas en **RStudio**.

.3.1. Función R

La función $R : M_{3\times N}(\mathbb{R}) \longrightarrow \mathbb{R}$ es la que calcula el radio del contenedor circular que contiene N círculos. En la subsección 4.2.1 se explica cómo queda definido un arreglo de N círculos en una matriz de dimensión $3 \times N$. La Figura 14 nos muestra cómo programar esta función y cómo usarla tomando como ejemplo el arreglo s de tres círculos descrito de la siguiente manera:

$$s = \begin{bmatrix} 3 & 1 & -5 \\ 3 & -2 & 3 \\ 1 & 2 & 3 \end{bmatrix}$$

Este arreglo queda definido por tres círculos; el círculo 1 tiene radio $r_1 = 1$ y centro en $(3, 3)$, el círculo 2 tiene radio $r_2 = 2$ y centro en $(1, -2)$, y el círculo 3 tiene radio $r_3 = 3$ y centro en $(-5, 3)$.

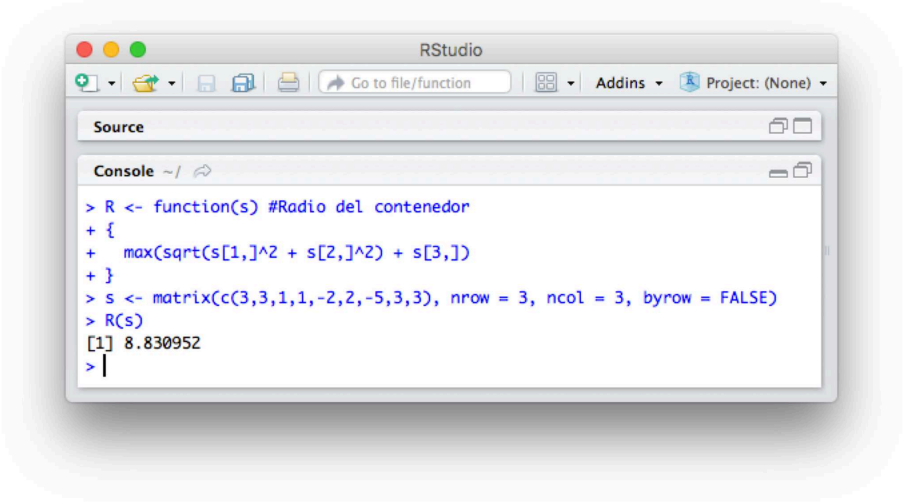


Figura 14. Ejemplo de cómo funciona R en **RStudio**

.3.2. Función $SolIn$

La función $SolIn : \mathbb{N} \longrightarrow M_{3\times N}(\mathbb{R})$ es la que genera la solución inicial; como ya sabemos, la solución inicial se genera de manera aleatoria y si $N \in \mathbb{N}$ es el número de círculos con los que se trabajará, $SolIn(N)$ da una solución de N círculos. Cabe destacar que la solución que nos da $SolIn$ puede no llegar a ser factible (para trabajar con la factibilidad se requiere de la función PP la cual se ve más adelante). La Figura 15 nos muestra cómo programar esta función y cómo usarla tomando como ejemplo $N = 3$.

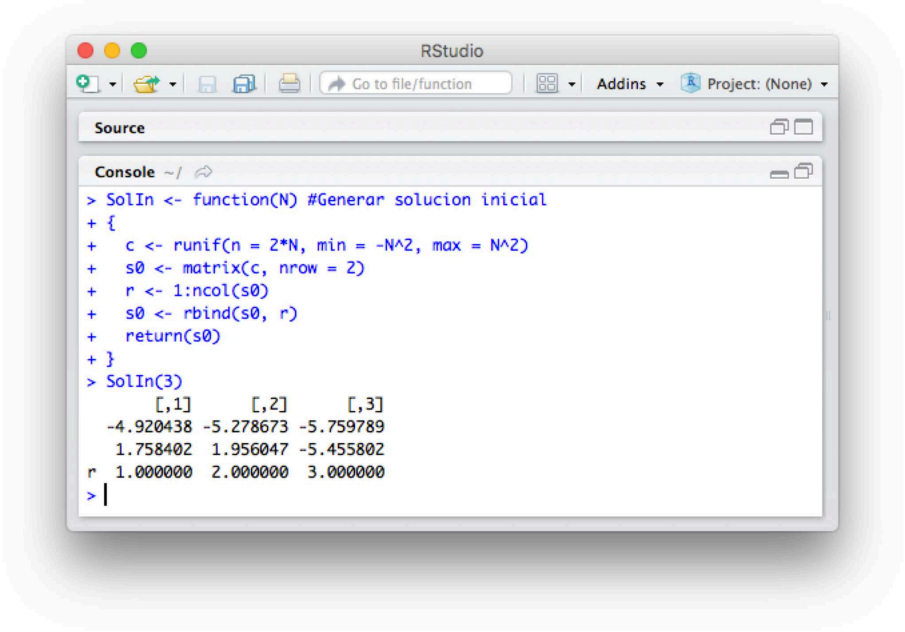


Figura 15. Ejemplo de cómo funciona $SolIn$ en **RStudio**

.3.3. Función P

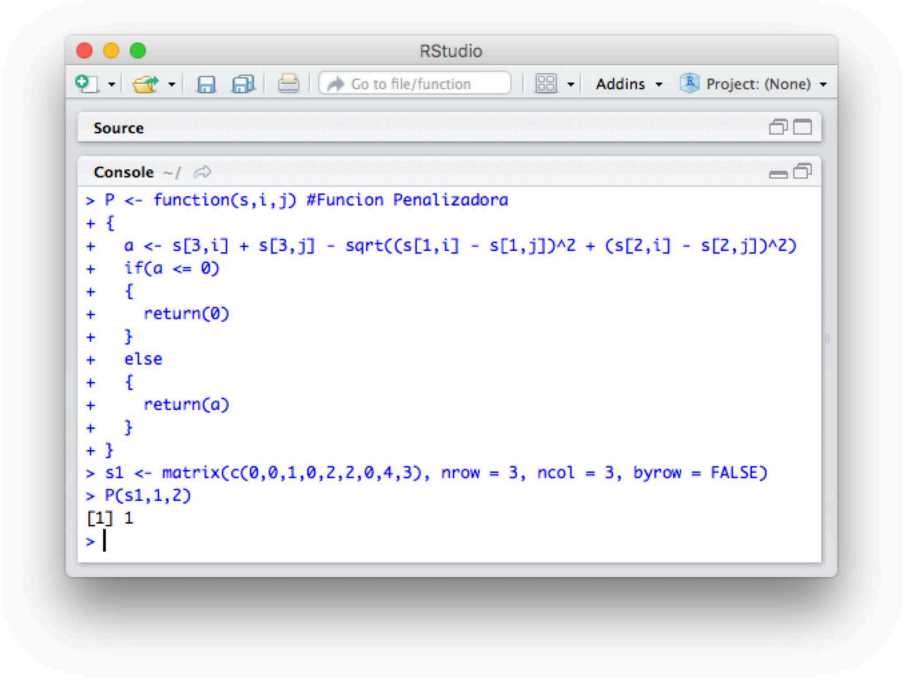
Sea N el número de círculos con los que se van a trabajar y d_{ij} la distancia euclidiana entre los centros de los círculos i y j, la función $P : M_{3\times N}(\mathbb{R}) \times \{1, ..., N\} \times \{1, ..., N\} \longrightarrow \mathbb{R}$, definida como:

$$P(s,i,j) = \begin{cases} r_i + r_j - d_{ij} & \text{si los círculos } i \text{ y } j \text{ de la solución } s \text{ se traslapan} \\ \\ 0 & \text{en otro caso} \end{cases}$$

Es la función que mide el traslape de dos círculos i y j dentro de una solución s; si hay traslape entre estos círculos, la función P nos da la cantidad de este traslape y si no hay traslape entre éstos, la función P dará cero. Esta función sólo sirve para medir el traslape entre dos círculos de una solución, pero más adelante nos ayudará para calcular el traslape entre N círculos arbitrarios. Para ejemplificar esta función se toma el arreglo $s1$ que se muestra adelante, el cual describe tres círculos: la primer columna define al círculo 1 con

radio $r_1 = 1$ y centro en $(0, 0)$, la segunda columna al círculo 2 con radio $r_2 = 2$ y centro en $(0, 2)$. y la tercer columna al círculo 3 con radio $r_3 = 3$ y centro en $(0, 4)$. La Figura 16 nos muestra cómo programar esta función y cómo usarla tomando como ejemplo a $s1$ y comparando los círculos 1 y 2 de éste. Notamos que, como $r_1 + r_2 = 1 + 2 = 3 > d_{12} = \sqrt{(0-0)^2 + (0-2)^2} = 2$, sí hay traslape entre los círculos 1 y 2; y la cantidad de traslape es de $r_1 + r_2 - d_{12} = 1$.

$$s1 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 2 & 4 \\ 1 & 2 & 3 \end{bmatrix}$$



(a)

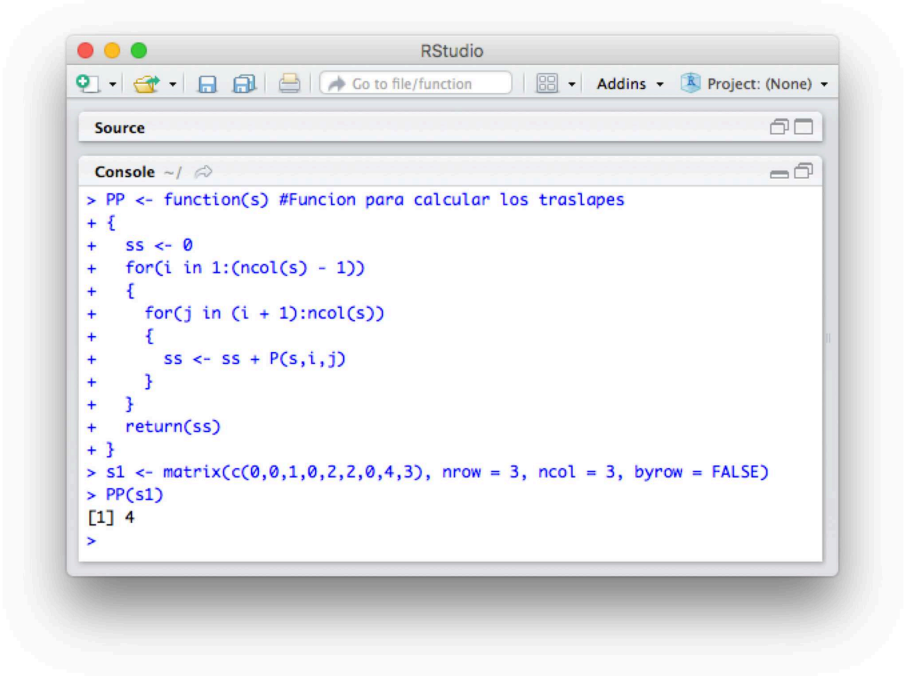
(b)

Figura 16. (a) Ejemplo de cómo funciona P en **RStudio**, (b) Cómo se ve gráficamente $s1$

.3.4. Función PP

A comparación de P, la función $PP : M_{3\times N}(\mathbb{R}) \longrightarrow \mathbb{R}$ es la que nos dice cuánto se traslapan todos los N círculos de una solución. Esta función tiene en total $\frac{N(N-1)}{2}$ sumandos y calcula el traslape de los círculos de dos en dos (sin repetición) con la función P y los suma todos. Esta función está definida como lo muestra la expresión 3, y la Figura 17 nos muestra cómo programar esta función y cómo usarla tomando como ejemplo el mismo arreglo $s1$ utilizado en la función P.

$$\sum_{i=1}^{N-1} \sum_{j=i+1}^{N} P(i,j) \tag{3}$$

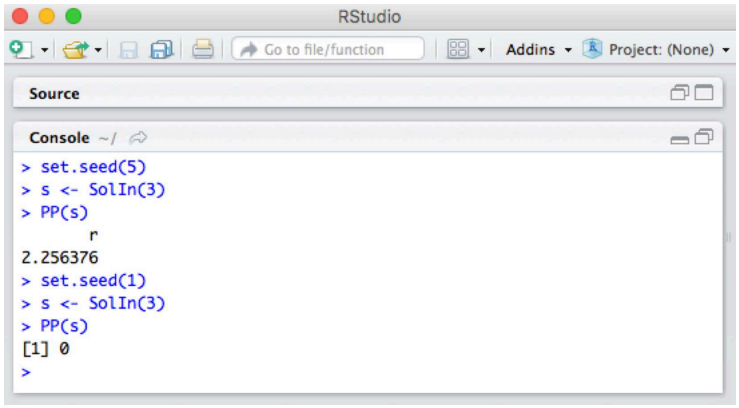


(a)

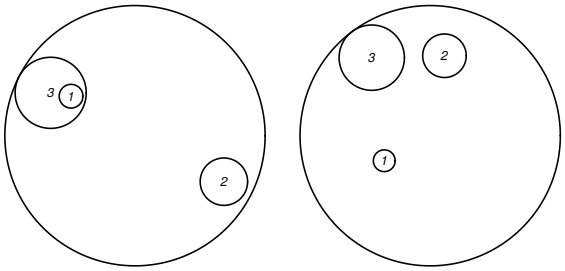
(b)

Figura 17. (a) Ejemplo de cómo funciona PP en **RStudio**, (b) Cómo se ve gráficamente $s1$

•NOTA
Esta función es justo la expresión 4.2 y nos dice si una solución s es o no factible. Con esta función se verifica si la solución inicial generada por $SolIn$ es factible, si ésta no es factible se cambia de semilla hasta llegar con una solución inicial factible (véase Figura 18).



(a)



(b) (c)

Figura 18. (a) Ejemplo de cómo se genera una solución factible con la ayuda de PP en **RStudio**, (b) Cómo se ve gráficamente $SolIn(3)$ con semilla 5, (c) Cómo se ve gráficamente $SolIn(3)$ con semilla 1

.3.5. Función *pol*

La función $pol : M_{3\times N}(\mathbb{R}) \longrightarrow M_{3\times N}(\mathbb{R})$ pasa los centros de los N círculos del sistema cartesiano al polar. Es decir, dada $A = (a_{ij}) \in M_{3\times N}(\mathbb{R})$, la función pol convierte de coordenadas cartesianas a polares a los puntos (a_{1j}, a_{2j}) para todo $j = 1, ..., N$ y así, la función pol queda definida para cada círculo $j \in \{1, ..., N\}$ como:

$$(pol(A))_{ij} = \begin{cases} \sqrt{a_{ij}^2 + a_{2j}^2} & \text{si i=1} \\ arctan(\frac{a_{ij}}{a_{1j}}) & \text{si i=2} \\ a_{ij} & \text{si i=3} \end{cases}$$

Esta función nos servirá más adelante al momento de generar vecinos. La Figura 19 nos muestra cómo programar esta función y cómo usarla tomando como ejemplo el mismo arreglo s que se usó en la función R.

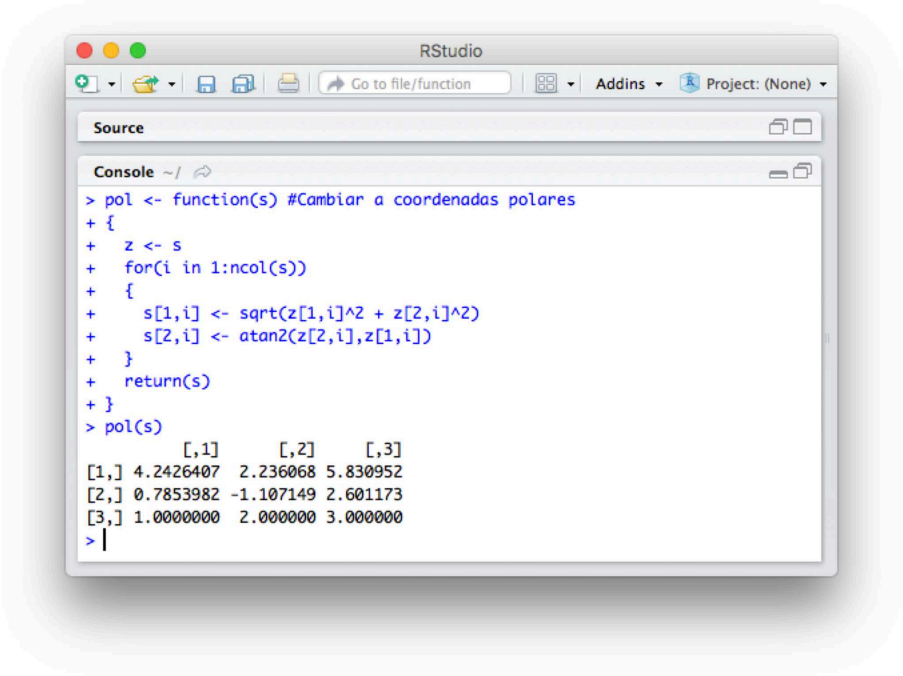


Figura 19. Ejemplo de cómo funciona *pol* en **RStudio**

.3.6. Función *car*

La función $car : M_{3\times N}(\mathbb{R}) \longrightarrow M_{3\times N}(\mathbb{R})$ convierte de coordenadas polares a cartesianas a los centros de los N círculos. Es decir, dada $A = (a_{ij}) \in M_{3\times N}(\mathbb{R})$, la función *car* convierte de coordenadas polares a cartesianas a los puntos (a_{1j}, a_{2j}) para todo $j = 1, ..., N$ y así, la función *car* queda definida para cada círculo $j \in \{1, ..., N\}$ como:

$$(car(A))_{ij} = \begin{cases} (a_{ij}) \cos a_{2j} & \text{si i=1} \\ (a_{1j}) \sin a_{ij} & \text{si i=2} \\ a_{ij} & \text{si i=3} \end{cases}$$

Esta función, al igual que la función *pol*, nos servirá más adelante al momento de generar vecinos. La Figura 20 nos muestra cómo programar esta función y cómo usarla tomando como ejemplo el arreglo resultante de aplicarle la función *pol* al arreglo s que se usó en la función R.

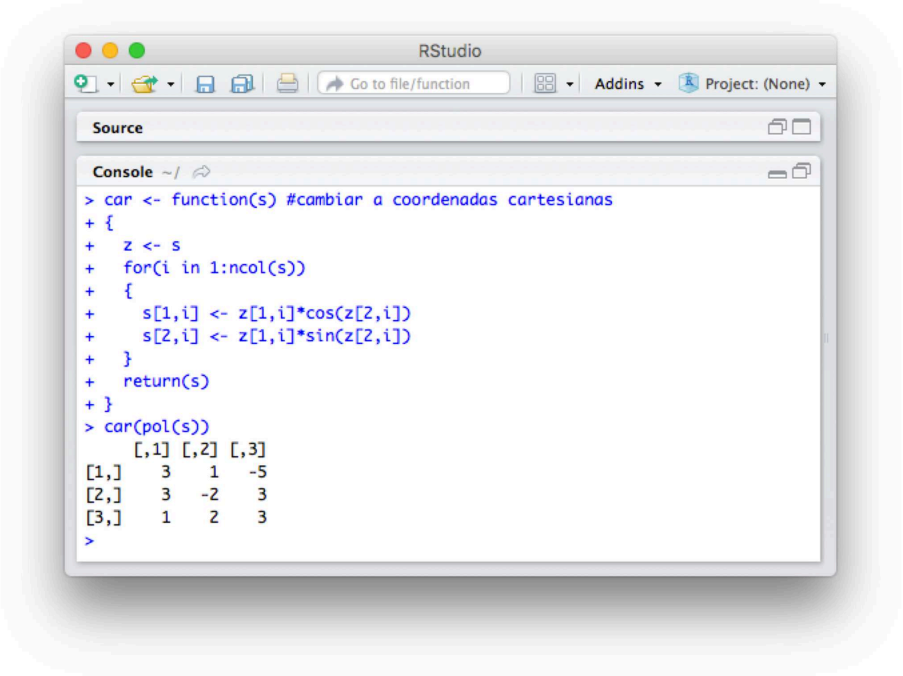
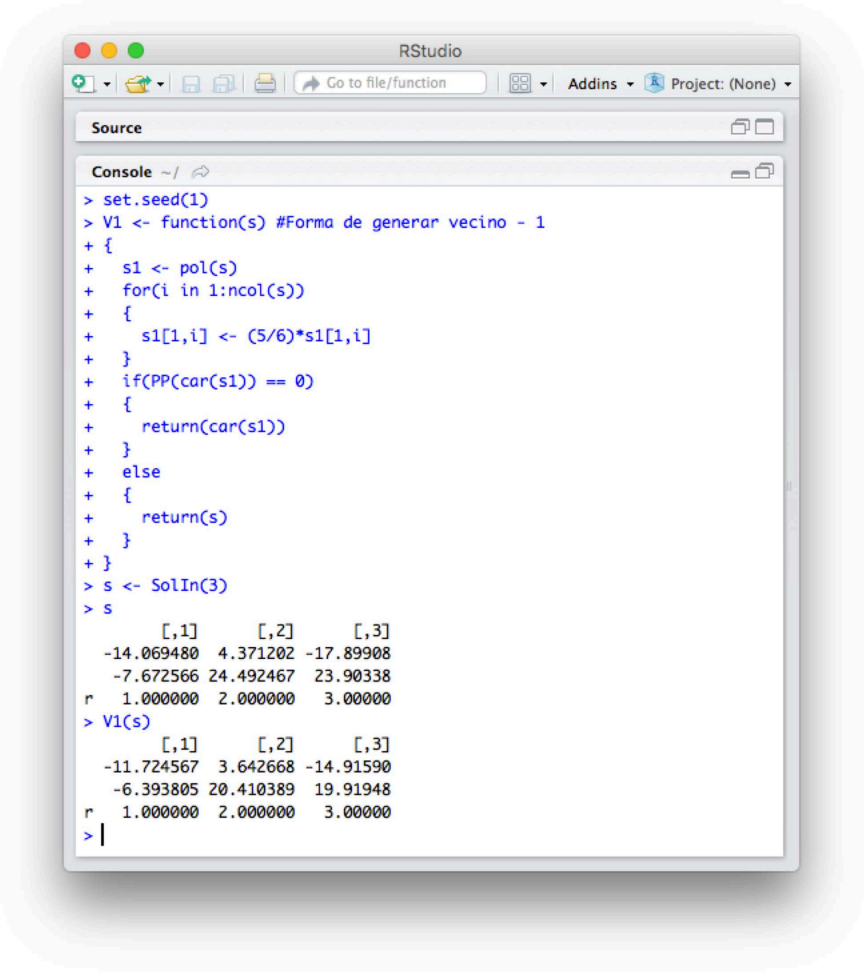


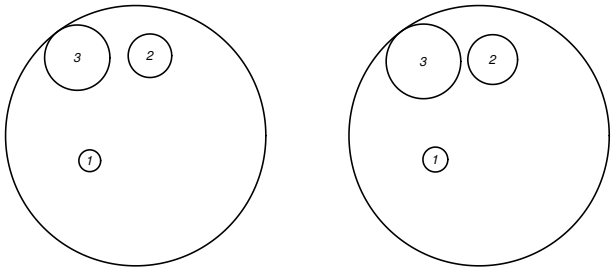
Figura 20. Ejemplo de cómo funciona *car* en **RStudio**

.3.7. Función $V1$

Sea $S = \{s \in M_{3\times N}(\mathbb{R}) | PP(s) < 10^{-5}\}$ el espacio de soluciones factibles, la función $V1 : S \longrightarrow S$ es la que representa la *forma de generar vecinos - 1*. Es decir, dada $s \in S$, $V1$ reduce la distancia que hay de los centros de los N círculos de s al origen, por un factor de $\frac{5}{6}$. Recordamos que cada vez que se aplica la función $V1$, se verifica la factibilidad de estas soluciones obtenidas con la ayuda de la función PP; tomando en cuenta que si la nueva solución es factible, se acepta a ésta como la solución vecina de s, de lo contrario, nos quedamos con la última solución factible generada como el vecino de s (la cual es la misma s, pues en principio s es una solución factible). Dentro de esta función, se utilizan las funciones *pol* y *car*, ya que para reducir la distancia de un punto de $\mathbb{R}^2$ al origen se recurre a su representación polar (como se explica en el Capítulo 4). La Figura 21 nos muestra cómo programar esta función y cómo usarla tomando como ejemplo el arreglo resultante de $SolIn(3)$ con semilla 1.



(a)

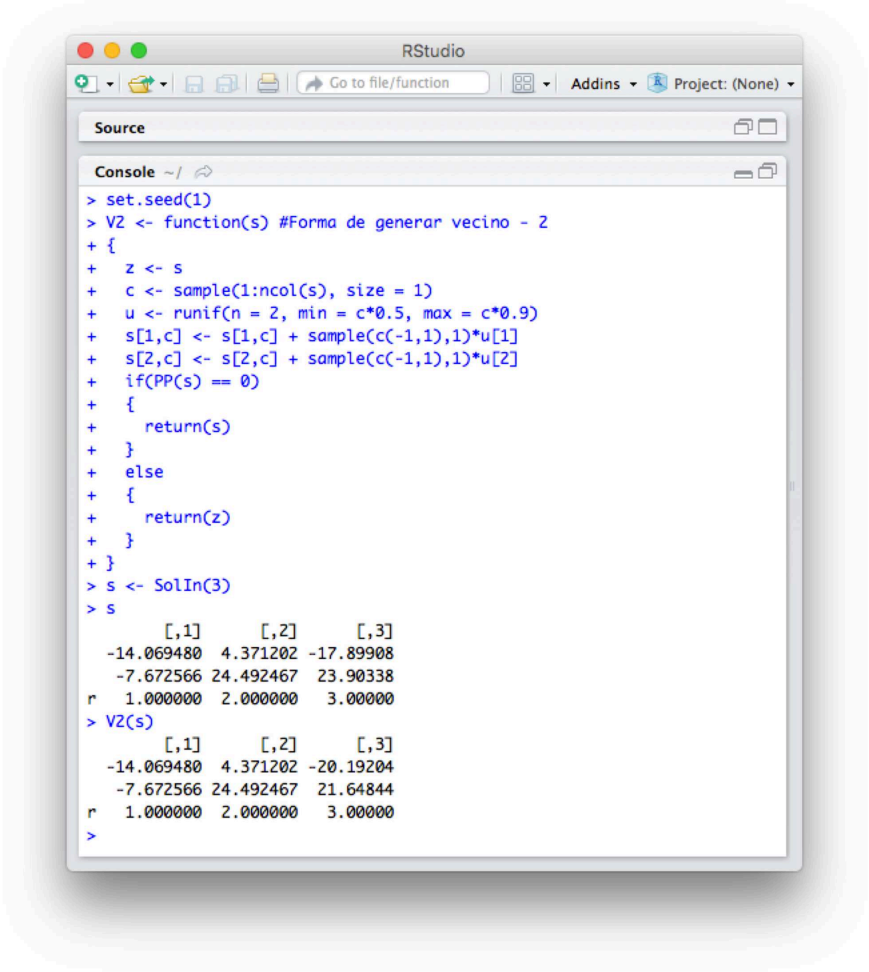


(b) R(s)=11.95865 (c) R(V1(s))=10.46554

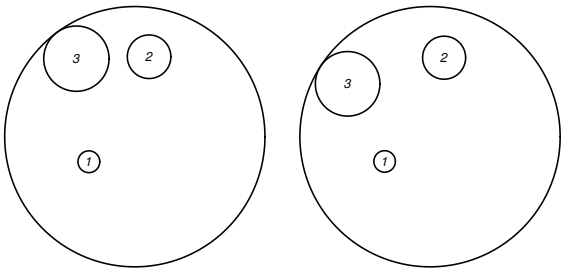
Figura 21. (a) Ejemplo de cómo funciona $V1$ en **RStudio** con la solución $s = SolIn(3)$, (b) Cómo se ve gráficamente s, (c) Cómo se ve gráficamente $V1(s)$

.3.8. Función $V2$

La función $V2 : S \longrightarrow S$ es la que representa la *forma de generar vecinos - 2*. Con esta función, dada $s \in S$, lo que hace $V2$ es escoger aleatoriamente un círculo de s y cambiar la posición de su respectivo centro, después, se verifica la factibilidad de esta solución perturbada apoyándonos de la función PP; recordando que si la nueva solución es factible, se acepta a ésta como la solución vecina de s, de lo contrario, nos quedamos con la última solución factible generada como el vecino de s (la cual es la misma s, pues en principio s es una solución factible). En el Capítulo 4 se ve a detalle cómo se puede ir moviendo el centro (x_i, y_i) para cada $i \in \{1, ..., N\}$. La Figura 22 nos muestra cómo programar esta función y cómo usarla tomando como ejemplo el arreglo resultante de $SolIn(3)$ con semilla 1.



(a)

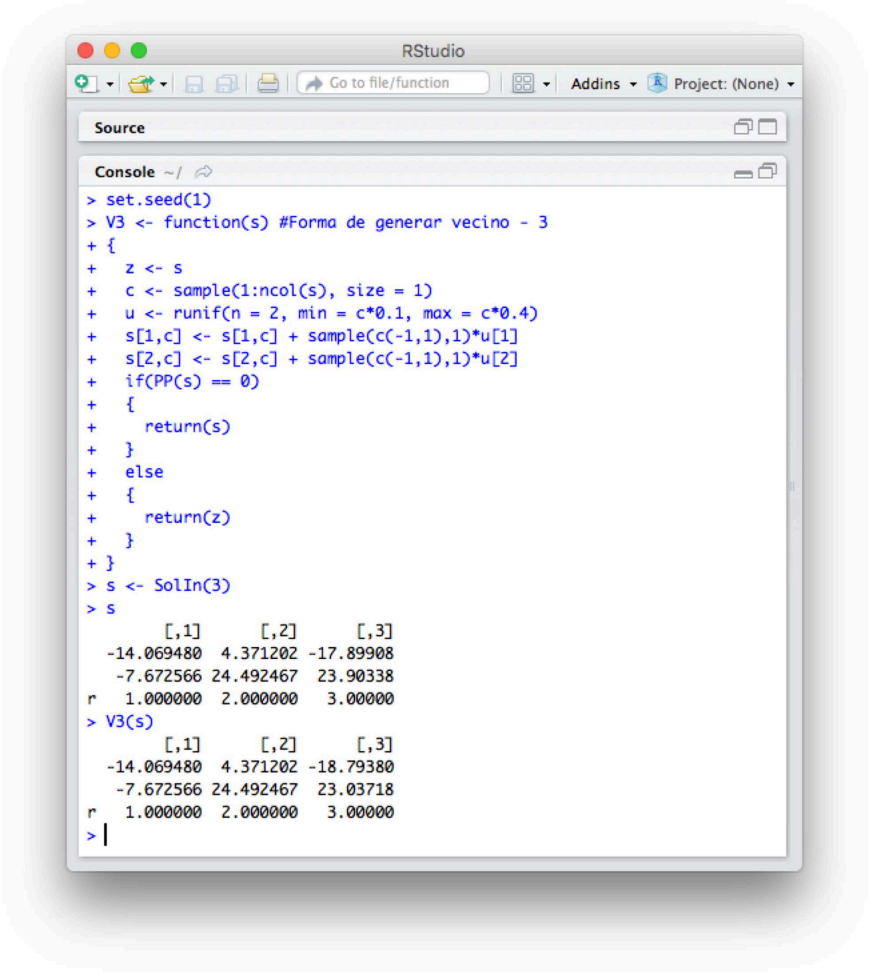


(b) R(s)=11.95865 (c) R(V2(s))=12.10409

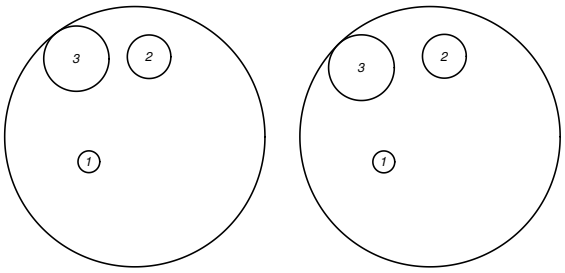
Figura 22. (a) Ejemplo de cómo funciona $V2$ en **RStudio** con la solución $s = SolIn(3)$, (b) Cómo se ve gráficamente s, (c) Cómo se ve gráficamente $V2(s)$

.3.9. Función $V3$

La función $V3 : S \longrightarrow S$ es la que representa la *forma de generar vecinos - 3*. Esta función funciona igual que $V2$; lo único que cambia es que $V3$ mueve de una forma más restringida el centro del círculo que se escogió aleatoriamente. En el Capítulo 4 se ve con detalle cómo cambia la forma de mover los centros (x_i, y_i) para cada $i \in \{1, ..., N\}$. La Figura 23 nos muestra cómo programar esta función y cómo usarla tomando como ejemplo el arreglo resultante de $SolIn(3)$ con semilla 1.



(a)



(b) R(s)=11.95865 (c) R(V3(s))=11.88785

Figura 23. (a) Ejemplo de cómo funciona $V3$ en **RStudio** con una solución $s = SolIn(3)$, (b) Cómo se ve gráficamente s, (c) Cómo se ve gráficamente $V3(s)$

.3.10. Función *RecSim*

La función $RecSim : \mathbb{N} \times \mathbb{R} \times \mathbb{R} \times \mathbb{R} \times \mathbb{N} \longrightarrow S \times \mathbb{R}$ es la función que hace todo el proceso de Recocido Simulado aplicado al problema de empaquetamiento. Los datos que requiere esta función para poder aplicarla son $(N, a, T_0, T_{final}, nrep)$; $N \in \mathbb{N}$ define con cuántos círculos se va a trabajar, $a \in \mathbb{R}$ es el factor de la función $\alpha(t) = at$ para definir cómo drecece la temperatura, $T_0 \in \mathbb{R}$ la temperatura inicial, $T_{final} \in \mathbb{R}$ la temperatura final, y $nrep \in \mathbb{N}$ el número de iteraciones a realizar antes de bajar la temperatura. Definidos estos datos, la función *RecSim* nos dará como resultado la solución heurística obtenida s^* junto con el valor $R(s^*)$ y se verá en **RStudio** en forma de lista. Las Figuras 24, 25, 26, y 27 nos muestran cómo programar esta función y cómo usarla tomando como semilla el número 1.

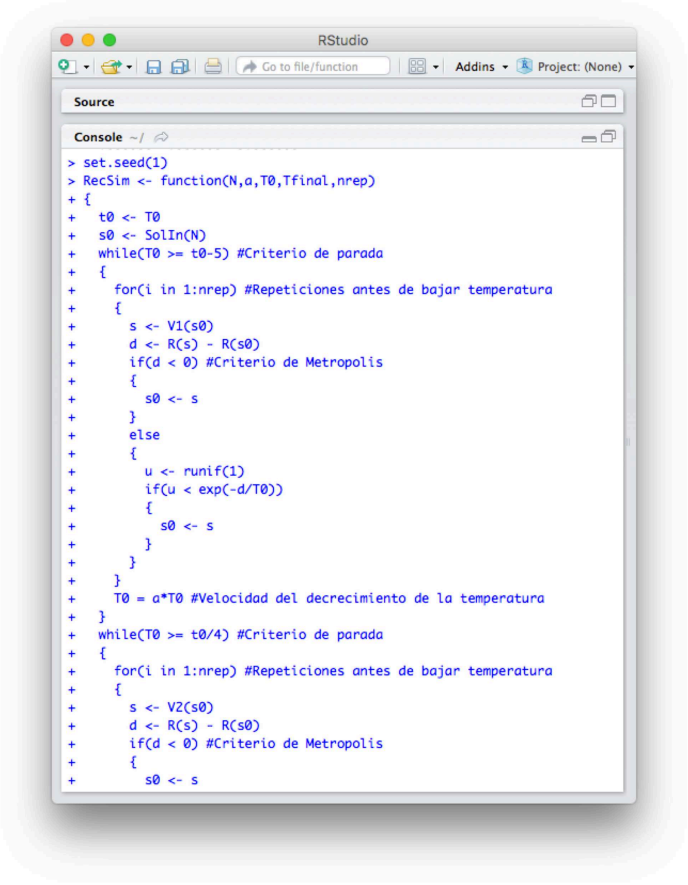


Figura 24. Ejemplo de cómo funciona *RecSim* en **RStudio** 1/3

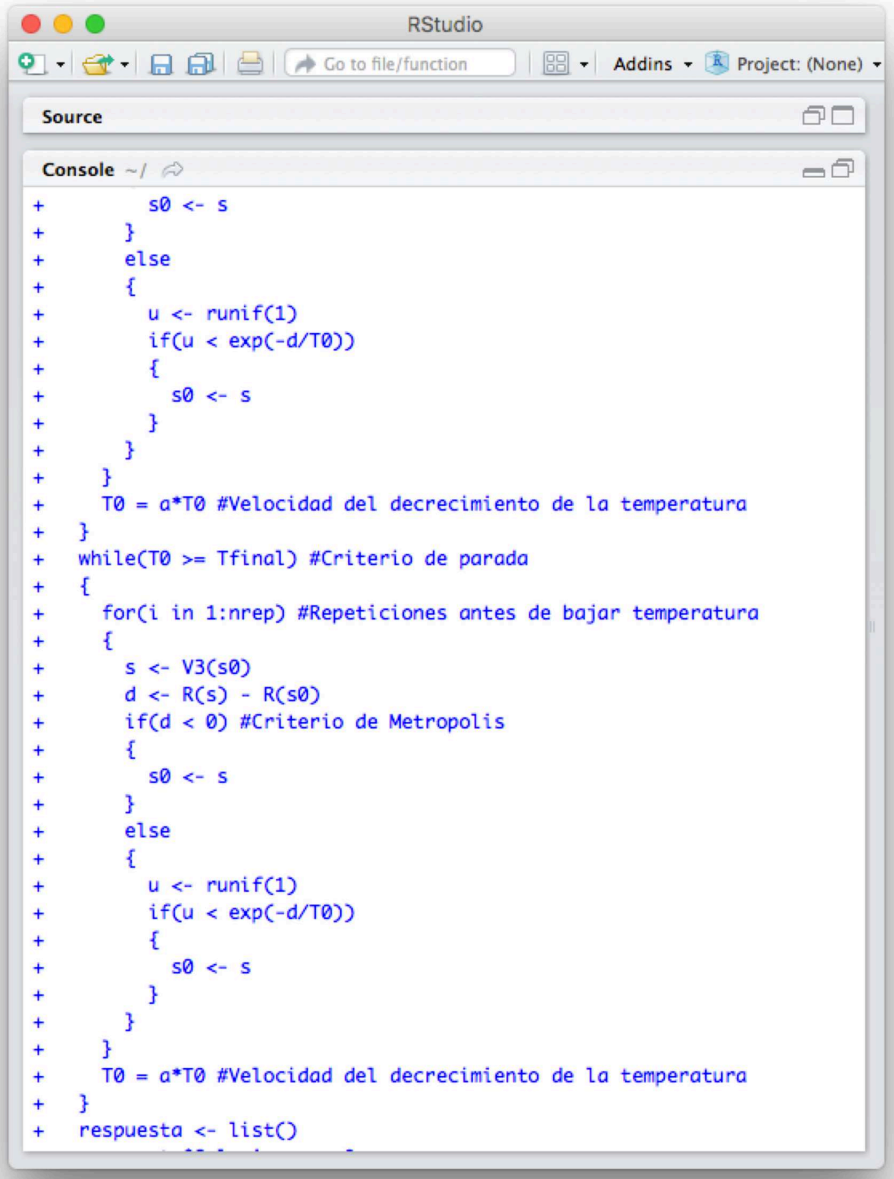


Figura 25. Ejemplo de cómo funciona *RecSim* en **RStudio** 2/3

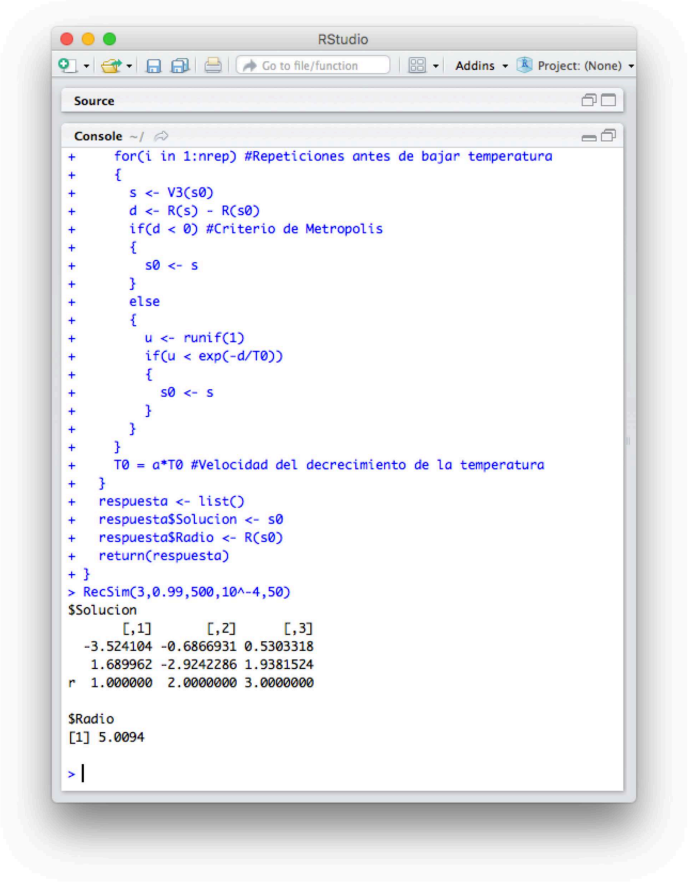
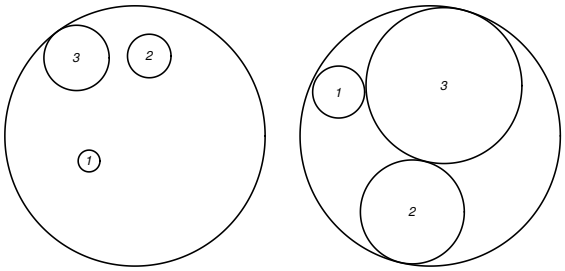


Figura 26. Ejemplo de cómo funciona *RecSim* en **RStudio** 3/3



(a) R(s_0)=11.95865 (b) R(s^*)=5.0094

Figura 27. (a) Cómo se ve gráficamente s_0, (b) Cómo se ve gráficamente s^*

.4. Gráficas en R

A lo largo de este proyecto se presentaron gráficamente los resultados que se iban obteniendo y como lo único que se graficó fueron círculos, a continuación se explicará la

función que se ocupó para esto.

.4.1. Función *circle*

La función $circle : \mathbb{R}^3 \longrightarrow C \subseteq \mathbb{R}^2$ asocia al punto (x, y, r) con el conjunto $C \subseteq \mathbb{R}^2$ de puntos que conforman al círculo con radio r y centro en (x, y). Esta función no regresa ningún resultado (pues el resultado es un conjunto de cardinalidad infinita) pero lo que hace es graficar el conjunto C de la siguiente manera: toma una secuencia regular de puntos en el conjunto $[0, 2 * \pi]$, éstos los acomoda de tal forma que pertenezcan al círculo que queremos graficar y al final los une, formando un polígono regular; entre más puntos se tomen, más se verá el polígono regular como un círculo. Durante todo el proyecto se tomaron 1000 puntos de forma regular del conjunto $[0, 2 * \pi]$.
Antes de graficar, se requiere llamar a la función *plot* incluida en **R**, en la cual determinamos el tamaño de la imagen y algunos datos técnicos (como el nombre de los ejes o si se quieren o no mostrar éstos). Se muestra la Figura 28 para saber cómo se programa esta función y cómo funciona.

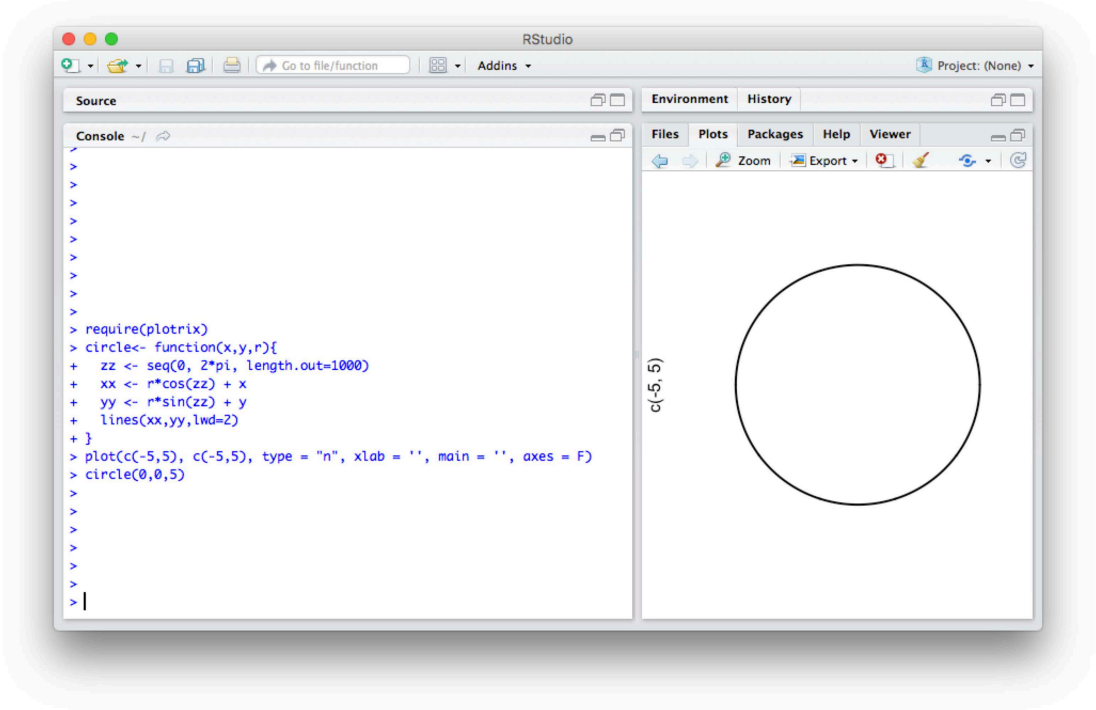


Figura 28. Ejemplo de cómo funciona *circle* en **RStudio**

Para graficar n círculos más en la misma imagen, se meten las funciones *circle* deseadas y los círculos se verán dentro de la misma imagen hasta que volvamos a meter la función *plot* la cual abre una nueva imagen. La Figura 29 nos muestra cómo programar esta función y cómo usarla tomando como ejemplo círculos concéntricos en el origen y con radio $r_i = i$ para $i = 1, .., 5$.

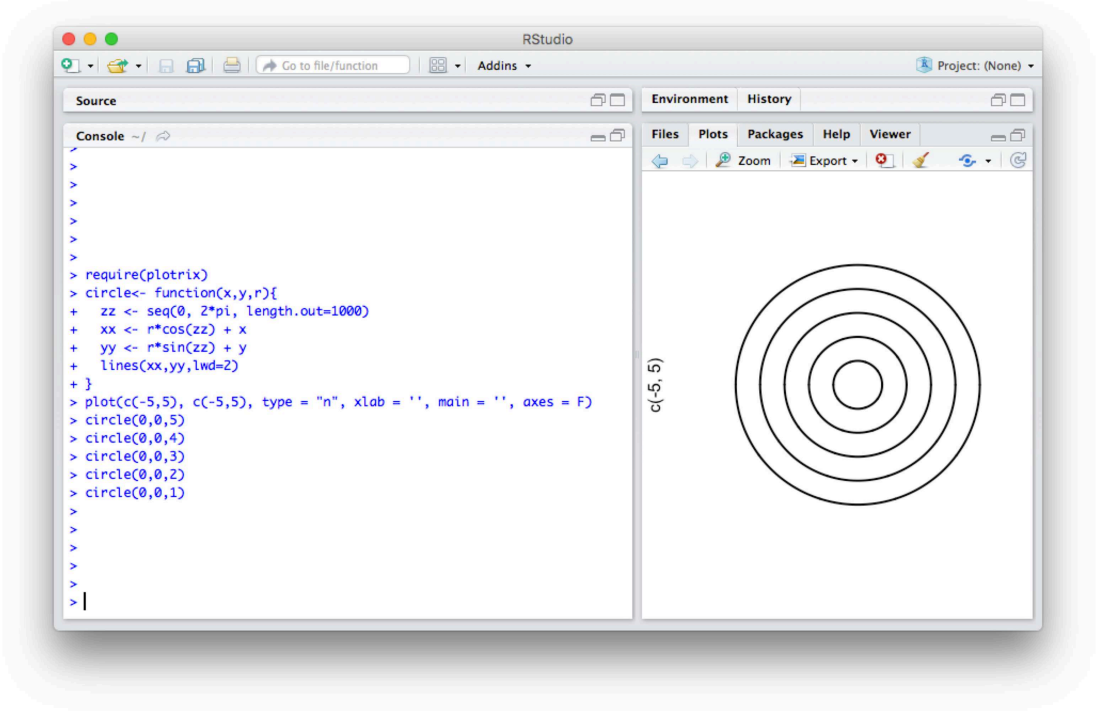


Figura 29. Ejemplo de cómo graficar varios círculos con *circle* en **RStudio**

Además, notamos que en las gráficas de las soluciones heurísticas obtenidas, cada círculo tiene en su respectivo centro un número (el cual representa el radio del círculo en cuestión). Para poder agregar texto en la imagen se manda llamar a la función *text*, la cual requiere de dos datos: las coordenadas en las que se desea el texto y el texto que se desea poner en la imagen. La Figura 30 nos muestra cómo programar esta función y cómo usarla tomando como ejemplo el círulo de radio 5 con centro en el origen.

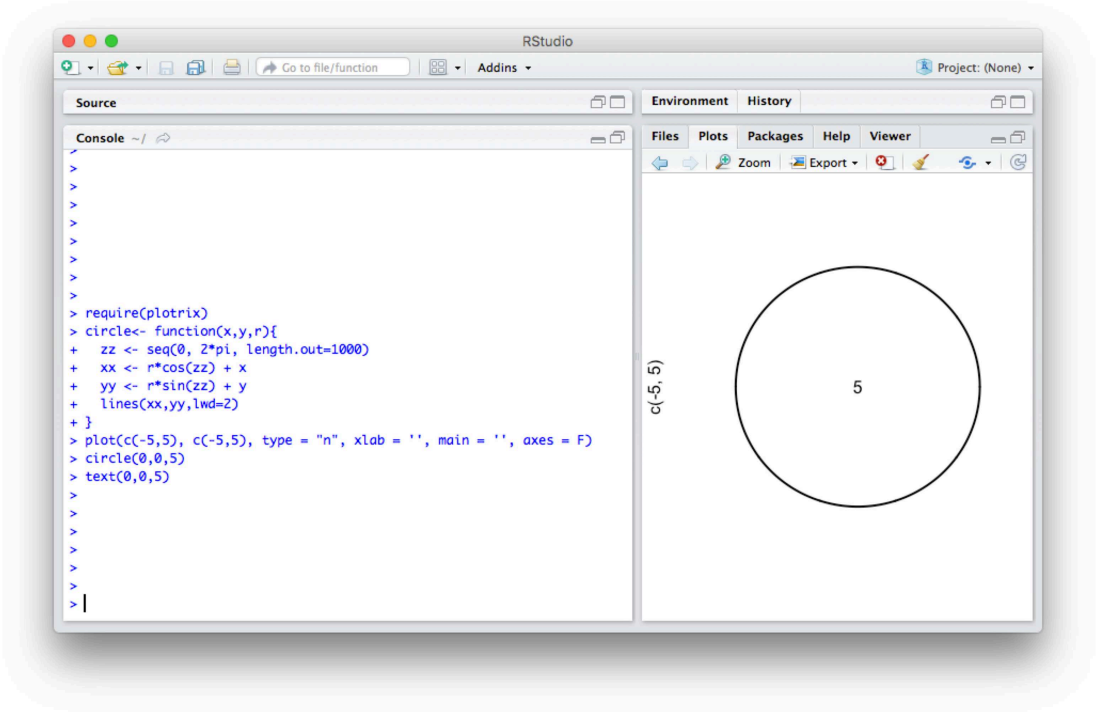


Figura 30. Ejemplo de cómo graficar con texto en **RStudio**

Código para Resolver el Problema de la Mochila Usando Recocido Simulado

```
C <- matrix(c(100,155,50,112,70,80,60,118,110,55), nrow = 10, ncol = 1)
#Vector restricci\'on
cc <- matrix(c(-150,-190,-100,-155,-110,-130,-120,-158,-151,-105), nrow = 10, ncol = 1)
#Vector de costo

f <- function(s)
{
  s%*%cc
}

rest <- function(s)
{
  s%*%C
}

cof <- function(s)
{
j <- which(s > 0)
cof <- matrix(-Inf, nrow = 1, ncol = 10)
  for(i in 1:length(j))
   {
      cof[j[i]] <- (cc[j[i]])/(C[j[i]])
   }
return(which.max(cof))
}

V <- function(s) #generar vecino
{
```

```
  a <- sample(1:10, size = 1)
  s[a] <- 1 - s[a]
  while(rest(s) > 700)
  {
    s[cof(s)] <- 0
  }
  return(s)
}

SimAnn <- function(s0,a,T0,Tfinal,nrep) #recocido simulado
{
  while(T0 >= Tfinal) #Criterio de parada
  {
    for(i in 1:nrep) #Repeticiones antes de bajar temperatura
    {
      s <- V(s0)
      d <- f(s) - f(s0)
      if(d < 0) #Criterio de Metropolis
      {
        s0 <- s
      }
      else
      {
        u <- runif(1)
        if(u < exp(-d/T0))
        {
          s0 <- s
        }
      }
    }
    T0 = a*T0 #Velocidad del decrecimiento de la temperatura
  }
  respuesta <- list()
  respuesta$Solucion <- s0
  respuesta$Costo <- f(s0)
  return(respuesta)
}
```

Código para Resolver el Problema de Empaquetamiento Usando Recocido Simulado

```
R <- function(s) #Radio del contenedor
{
  max(sqrt(s[1,]^2 + s[2,]^2) + s[3,])
}

SolIn <- function(N) #Generar soluci\'on inicial
{
  c <- runif(n = 2*N, min = -N^2, max = N^2)
  s0 <- matrix(c, nrow = 2)
  r <- 1:ncol(s0)
  s0 <- rbind(s0, r)
  return(s0)
}

P <- function(s,i,j) #Funci\'on Penalizadora
{
  a <- s[3,i] + s[3,j] - sqrt((s[1,i] - s[1,j])^2 + (s[2,i] - s[2,j])^2)
  if(a <= 0)
  {
    return(0)
  }
  else
  {
    return(a)
  }
}

PP <- function(s) #Funci\'on para calcular los traslapes
```

```
{
  ss <- 0
  for(i in 1:(ncol(s) - 1))
  {
    for(j in (i + 1):ncol(s))
    {
      ss <- ss + P(s,i,j)
    }
  }
  return(ss)
}

pol <- function(s) #Cambiar a coordenadas polares
{
  z <- s
  for(i in 1:ncol(s))
  {
    s[1,i] <- sqrt(z[1,i]^2 + z[2,i]^2)
    s[2,i] <- atan2(z[2,i],z[1,i])
  }
  return(s)
}

car <- function(s) #cambiar a coordenadas cartesianas
{
  z <- s
  for(i in 1:ncol(s))
  {
    s[1,i] <- z[1,i]*cos(z[2,i])
    s[2,i] <- z[1,i]*sin(z[2,i])
  }
  return(s)
}

V1 <- function(s) #Forma de generar vecino - 1
{
  s1 <- pol(s)
  for(i in 1:ncol(s))
  {
    s1[1,i] <- (5/6)*s1[1,i]
  }
  if(PP(car(s1)) == 0)
  {
```

```
    return(car(s1))
  }
  else
  {
    return(s)
  }
}

V2 <- function(s) #Forma de generar vecino - 2
{
  z <- s
  c <- sample(1:ncol(s), size = 1)
  u <- runif(n = 2, min = c*0.5, max = c*0.9)
  s[1,c] <- s[1,c] + sample(c(-1,1),1)*u[1]
  s[2,c] <- s[2,c] + sample(c(-1,1),1)*u[2]
  if(PP(s) == 0)
  {
    return(s)
  }
  else
  {
    return(z)
  }
}

V3 <- function(s) #Forma de generar vecino - 3
{
  z <- s
  c <- sample(1:ncol(s), size = 1)
  u <- runif(n = 2, min = c*0.1, max = c*0.4)
  s[1,c] <- s[1,c] + sample(c(-1,1),1)*u[1]
  s[2,c] <- s[2,c] + sample(c(-1,1),1)*u[2]
  if(PP(s) == 0)
  {
    return(s)
  }
  else
  {
    return(z)
  }
}

RecSim <- function(N,a,T0,Tfinal,nrep) #Recocido Simulado
```

```
{
  t0 <- T0
  s0 <- SolIn(N) #solucion inicial
  while(T0 >= t0-5) #Criterio de parada
  {
    for(i in 1:nrep) #Repeticiones antes de bajar temperatura
    {
      s <- V1(s0)
      d <- R(s) - R(s0)
      if(d < 0) #Criterio de Metropolis
      {
        s0 <- s
      }
      else
      {
        u <- runif(1)
        if(u < exp(-d/T0))
        {
          s0 <- s
        }
      }
    }
    T0 = a*T0 #Velocidad del decrecimiento de la temperatura
  }
  while(T0 >= t0/4) #Criterio de parada
  {
    for(i in 1:nrep) #Repeticiones antes de bajar temperatura
    {
      s <- V2(s0)
      d <- R(s) - R(s0)
      if(d < 0) #Criterio de Metropolis
      {
        s0 <- s
      }
      else
      {
        u <- runif(1)
        if(u < exp(-d/T0))
        {
          s0 <- s
        }
      }
    }
```

```
    T0 = a*T0 #Velocidad del decrecimiento de la temperatura
  }
  while(T0 >= Tfinal) #Criterio de parada
  {
    for(i in 1:nrep) #Repeticiones antes de bajar temperatura
    {
      s <- V3(s0)
      d <- R(s) - R(s0)
      if(d < 0) #Criterio de Metropolis
      {
        s0 <- s
      }
      else
      {
        u <- runif(1)
        if(u < exp(-d/T0))
        {
          s0 <- s
        }
      }
    }
    T0 = a*T0 #Velocidad del decrecimiento de la temperatura
  }
  respuesta <- list()
  respuesta$Solucion <- s0
  respuesta$Radio <- R(s0)
  return(respuesta)
}
```

Código para Graficar en R

```
require(plotrix)

circle<- function(x,y,r)
{
  zz <- seq(0, 2*pi, length.out=1000)
  xx <- r*cos(zz) + x
  yy <- r*sin(zz) + y
  lines(xx,yy,lwd=2)
}

par(mai=c(0.02, 0.02,  0.02, 0.02), font=3)
plot(c(-R(s), R(s)), c(-R(s), R(s)), type = "n", xlab='', ylab='', main='', axes=F)
#tama\~no de la imagen
circle(0,0,R(s)) #graficando el contenedor circular
for(j in 1:ncol(s)){circle(s[1,j], s[2,j], s[3,j]) #graficar los N c\'irculos
text(s[1,j], s[2,j], s[3,j])} #agregar el radio de cada c\'irculo en su centro
```

Bibliografía

[1] Blum, C., y Roli A. (2003). Metaheuristics in Combinatorial Optimization: Overview and Conceptual Comparison. *ACM Computing Surveys*.

[2] Caserta, M., y Voß, S. (2009). *Metaheuristics: Intelligent Problem Solving*. Springer.

[3] Castillo, I., Kampas, F. J., y Pintér, J. D. (2008). Solving circle packing problems by global optimization: Numerical results and industrial applications. *European Journal of Operational Research*.

[4] Dowsland, K. A., y Díaz, B. A. (2001). Diseño de Heurísticas y Fundamentos del Recocido Simulado. *Revista Iberoamericana de Inteligencia Artificial*.

[5] Edmund K. Burke, Graham Kendall (2005). *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*. Springer.

[6] Gonzales, T. F. (2007). *Handbook of Approximation Algorithms and Metaheuristics*. Chapman & Hall/CRC.

[7] Hifi, M., y M'Hallah, R. (2009). A Literature Review on Circle and Sphere Paching Problems: Models and Methodologies. *Advances in Operations Research*.

[8] Hifi, M., Paschos, V. T., y Zissimopoulos, V. (2004). A simulated annealing approach for the circular cutting problem. *European Journal of Operational Research*.

[9] Judea, P. (1984). *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison - Wesley Publishing Company.

[10] López, C. O., y Beasley, J.E. (2011). A heuristic for the circle packing problem with a variety of containers. *European Journal of Operational Research*.

[11] López, C. O., y Beasley, J.E. (2012). Packing unequal circles using formulation space search. *European Journal of Operational Research*.

[12] Martí, R. (2003). Procedimientos Metaheurísticos en Optimización Combinatoria.

[13] Melián, B., Moreno Pérez, J. A., y Moreno, J. M. (2003). Metaheurísticas: una visión global. *Revista Iberoamericana de Inteligencia Artificial*.

[14] Müller, A., Schneider, J. J., y Schömer, E. (2009). Packing a multidisperse system of hard disks in a circular environment. *Physical Review*.

[15] Reeves, C. B. (1993). *Modern Heursitic Techniques for Combinatorial Problems*. Halsted Press.

[16] Resende, G. C., y Ribeiro, C. C. (2016). *Optimization by GRASP*. Springer.

[17] Specht, E. (2015). http://www.packomania.com. última consulta: Diciembre 2017.

[18] Zhang, D. (2004). A Simulated Annealing Algorithm for the Circles Packing Problem. *Lecture Notes in Computer Science*.