UANL
UNIVERSIDAD AUTÓNOMA DE NUEVO LEÓN

FIME
FACULTAD DE INGENIERÍA MECÁNICA Y ELÉCTRICA

Universidad Autónoma de Nuevo León

Facultad de Ingeniería Mecánica y Eléctrica

Producto Integrador de Aprendizaje.

Manual Técnico de Algoritmos Genéticos

Nombre: Ranfery Josua Peregrina Morales

Matrícula: 1924910

Semestre: Décimo

Materia: Temas Selectos de IA

Docente: Raymundo Said Zamora Pequeño

Grupo: 001

Fecha: 25/05/2024

Periodo: *Enero – Junio 2024*

Índice

Introducción

El propósito de este manual técnico es proporcionar una guía detallada sobre la implementación y el funcionamiento interno del sistema de algoritmos genéticos desarrollado como parte de este proyecto. Este manual está dirigido a desarrolladores, ingenieros de software y cualquier persona interesada en comprender la estructura, arquitectura y funcionamiento técnico del software.

El sistema de algoritmos genéticos es una aplicación desarrollada en el lenguaje de programación R, diseñada para encontrar soluciones óptimas mediante la simulación de procesos evolutivos inspirados en la teoría de la evolución de Darwin. El programa utiliza un conjunto de reglas definidas para evaluar y seleccionar individuos dentro de una población, que luego se cruzan y mutan para generar nuevas soluciones. Este enfoque basado en la evolución natural permite explorar y optimizar espacios de soluciones complejos en una amplia variedad de campos, desde la optimización de problemas matemáticos hasta la ingeniería de sistemas.

En este manual, se proporcionará una visión detallada de la arquitectura del sistema, los requisitos del entorno de ejecución, la estructura del código fuente, la implementación de las funcionalidades clave, así como las estrategias de prueba y depuración utilizadas durante el desarrollo. Además, se incluirán instrucciones paso a paso para la instalación y configuración inicial del sistema, así como recomendaciones para el despliegue y mantenimiento a largo plazo.

Arquitectura del Sistema

Descripción de la arquitectura general del sistema:

El sistema de algoritmos genéticos está implementado en el lenguaje de programación R y consta de varios componentes principales que interactúan para llevar a cabo la optimización de soluciones. La arquitectura general se compone de los siguientes elementos:

- **Generación de la población inicial:** Este módulo se encarga de crear una población inicial de individuos con características aleatorias, siguiendo las restricciones especificadas por el usuario en cuanto al tamaño de la población y la longitud de cada individuo.
- **Evaluación de la población:** Una vez generada la población, se evalúa cada individuo para determinar su aptitud o calificación en función de la función objetivo definida. Esta evaluación se realiza siguiendo un conjunto de reglas predefinidas que determinan la calidad de cada solución.
- **Selección de individuos:** Con base en las calificaciones obtenidas, se seleccionan los individuos más aptos para formar parte de la próxima generación. Este proceso se realiza mediante técnicas de selección que priorizan a los individuos con mejores calificaciones, favoreciendo así la convergencia hacia soluciones óptimas.
- **Cruzamiento:** Los individuos seleccionados se cruzan entre sí para generar descendencia, combinando características de los padres para explorar nuevas soluciones en el espacio de búsqueda. Este proceso de cruzamiento se realiza siguiendo diferentes estrategias, como el cruce de un solo punto o el cruce uniforme, entre otros.
- **Mutación:** Posterior al cruzamiento, se aplica un proceso de mutación a la descendencia resultante para introducir variabilidad genética en la población. Este proceso consiste en

cambiar aleatoriamente algunos genes de los individuos, lo que permite explorar nuevas regiones del espacio de búsqueda y evitar la convergencia prematura hacia soluciones subóptimas.

- **Iteración:** Todo el proceso de generación, evaluación, selección, cruzamiento y mutación se repite durante un número predeterminado de iteraciones o generaciones, hasta alcanzar un criterio de parada definido por el usuario o hasta que se cumplan ciertas condiciones de convergencia.

Esta arquitectura modular permite una fácil extensibilidad y mantenimiento del sistema, así como la posibilidad de adaptarse a diferentes problemas y contextos de aplicación.

Requisitos del Sistema

Especificaciones de hardware y software:

- **Hardware:** Se recomienda un ordenador con al menos 2 GB de memoria RAM y un procesador de doble núcleo a 2.0 GHz o superior para un rendimiento óptimo. Sin embargo, el programa es lo suficientemente liviano como para ejecutarse en sistemas con especificaciones inferiores.
- **Software:** El sistema operativo debe ser compatible con R, el lenguaje de programación en el que se ha desarrollado el programa de algoritmos genéticos. Además, se requiere una instalación funcional de R para ejecutar el programa.

Requisitos de sistema operativo:

El programa es compatible con los siguientes sistemas operativos:

- Windows: Windows 7 o superior.
- macOS: macOS 10.12 (Sierra) o superior.
- Linux: Ubuntu 16.04 LTS o una distribución compatible con R.

Dependencias de software externas:

El programa utiliza dos paquetes o librerías externas que deben estar instaladas en el entorno de R:

- **stringr:** Esta librería proporciona funciones para manipular y trabajar con cadenas de caracteres de forma eficiente.
- **ggplot2:** Este paquete se utiliza para generar gráficos estadísticos de alta calidad y es utilizado en el programa para visualizar la evolución de la puntuación en cada iteración del algoritmo genético.

Ambas librerías están disponibles en el repositorio de paquetes CRAN (Comprehensive R Archive Network) y pueden instalarse utilizando el gestor de paquetes de R, **install.packages("nombre_paquete")**.

Pruebas y Depuración

Estrategias de prueba utilizadas para garantizar la calidad del software:

El programa ha sido sometido a una serie de pruebas para garantizar su calidad y funcionalidad. Estas pruebas incluyen:

- **Pruebas de validez de entrada:** Se han realizado pruebas con una variedad de datos de entrada, incluidas poblaciones de diferentes tamaños y longitudes de individuos irregulares. Esto garantiza que el programa pueda manejar diferentes escenarios y condiciones de entrada de manera adecuada.
- **Pruebas de límites:** Se han probado los límites del programa, incluidas poblaciones muy pequeñas y longitudes de individuos inusuales, para identificar posibles problemas o errores en el código y asegurar que el programa funcione correctamente en todos los casos.

Métodos de depuración para identificar y corregir errores en el código:

Para identificar y corregir errores en el código, se han utilizado varios métodos de depuración, que incluyen:

- **Impresión de mensajes de depuración:** Se han insertado mensajes de depuración en el código para imprimir información útil durante la ejecución del programa, como mensajes de advertencia sobre tamaños de cadena incorrectos o advertencias sobre poblaciones muy pequeñas.
- **Pruebas paso a paso:** Se ha realizado una depuración paso a paso del código utilizando herramientas disponibles en el entorno de desarrollo, como RStudio. Esto ha permitido identificar y corregir errores de manera sistemática y eficiente.

Despliegue y Mantenimiento

Procedimientos para desplegar el sistema en un entorno de producción:

El despliegue del sistema se puede realizar siguiendo estos procedimientos:

1. **Preparación del entorno:** Antes de desplegar el sistema en un entorno de producción, asegúrese de que el entorno cumpla con los requisitos de hardware y software necesarios para ejecutar el programa. Esto puede incluir la instalación de R y las bibliotecas necesarias, como **stringr** y **ggplot2**.
2. **Instalación del programa:** Copie los archivos del programa en el entorno de producción y configure las dependencias externas según sea necesario. Asegúrese de que los permisos de archivo sean adecuados para ejecutar el programa.
3. **Configuración inicial:** Si es necesario, realice cualquier configuración inicial requerida por el programa, como la configuración de parámetros o la especificación de la población inicial.
4. **Pruebas de aceptación:** Antes de poner el sistema en producción, realice pruebas exhaustivas para asegurarse de que funcione correctamente en el entorno de producción.

Esto puede incluir pruebas de integración, pruebas de carga y pruebas de aceptación del usuario.

5. **Despliegue:** Una vez que se hayan completado las pruebas satisfactoriamente, despliegue el sistema en el entorno de producción y asegúrese de que esté disponible para su uso.

Consideraciones de mantenimiento a largo plazo y actualizaciones futuras:

Para garantizar un mantenimiento efectivo a largo plazo y gestionar las actualizaciones futuras del sistema, se recomienda seguir estas consideraciones:

1. **Gestión de versiones:** Utilice un sistema de control de versiones, como Git, para mantener un historial de cambios en el código y gestionar las actualizaciones de manera efectiva. Este proyecto en este momento NO se encuentra en GitHub, pero es libre de distribuirlo de forma libreen esa o cualquier otra plataforma.

2. **Documentación:** Mantenga una documentación actualizada del sistema, incluyendo instrucciones de instalación, guías de usuario y manual técnico. Esto facilitará el mantenimiento y la resolución de problemas futuros.

3. **Monitoreo y supervisión:** Implemente herramientas de monitoreo y supervisión para supervisar el rendimiento del sistema y detectar posibles problemas o cuellos de botella. Esto le permitirá tomar medidas proactivas para abordar cualquier problema que surja.

4. **Actualizaciones regulares:** Mantenga el sistema actualizado con las últimas versiones de software y bibliotecas, así como parches de seguridad y correcciones de errores. Realice pruebas exhaustivas antes de implementar cualquier actualización en un entorno de producción.

5. **Soporte técnico:** Proporcione un mecanismo de soporte técnico para que los usuarios puedan informar de problemas y recibir asistencia en caso de que surjan problemas técnicos. Esto puede incluir un sistema de tickets de ayuda, un correo electrónico de soporte o un foro de discusión en línea. Pero por ahora no hay ningún sistema de retroalimentación. Puede implementarlo en caso de ser necesario.

Siguiendo estas consideraciones, podrá mantener el sistema de manera efectiva a lo largo del tiempo y asegurar su funcionalidad y rendimiento continuos.

Instalación

Instrucciones paso a paso para instalar el programa:

1. **Instalar R:** Si aún no tienes R instalado en tu sistema, necesitarás descargar e instalar R desde el sitio web oficial de R: https://www.r-project.org/. Sigue las instrucciones de instalación proporcionadas para tu sistema operativo específico.

2. **Instalar RStudio (opcional):** Aunque no es obligatorio, se recomienda utilizar RStudio como entorno de desarrollo integrado (IDE) para ejecutar el programa. Puedes descargar RStudio desde su sitio web oficial: https://www.rstudio.com/products/rstudio/download/. Sigue las instrucciones de instalación proporcionadas para tu sistema operativo específico.

3. **Descargar el código fuente:** Descarga el archivo de código fuente del programa desde el repositorio o la ubicación donde se encuentre disponible.

4. **Abre RStudio:** Inicia RStudio en tu sistema.

5. **Abre el archivo de código fuente:** En RStudio, abre el archivo de código fuente del programa que acabas de descargar.

6. **Ejecuta el código:** Una vez abierto el archivo de código fuente en RStudio, puedes ejecutar el programa haciendo clic en el botón "Run" (Ejecutar) o presionando la combinación de teclas **Ctrl + Enter**.

Configuración inicial del sistema:

No se requiere una configuración inicial específica del sistema para ejecutar el programa. Sin embargo, asegúrate de tener instaladas todas las dependencias necesarias, como se mencionó en la sección de "Requisitos del Sistema".

Configuración de dependencias externas:

Las dependencias externas del programa, como las bibliotecas o paquetes de R, se pueden instalar utilizando el gestor de paquetes de R, **install.packages("nombre_paquete")**. Asegúrate de tener instaladas las siguientes dependencias antes de ejecutar el programa:

- **stringr: install.packages("stringr")**
- **ggplot2: install.packages("ggplot2")**

Documentación del Código

En esta sección, proporcionaremos una explicación detallada del código utilizado en el programa. Analizaremos cada función, su propósito y cómo contribuye al funcionamiento general del sistema. Esta documentación ayudará a comprender mejor la implementación del software y su lógica subyacente.

```
library(stringr)
library(ggplot2)

#Estos datos no se cambian.
#Pero si se cambian, el programa está preparado.
```

```
Tamaño_Poblacion <- 8
Tamaño_Individuo <- 8
```

library(stringr): Esta línea carga la biblioteca "stringr", que proporciona funciones para manipular cadenas de texto de manera más conveniente en R. Esto es útil para operaciones como la manipulación de cadenas, búsqueda y reemplazo de patrones, entre otros.

library(ggplot2): Esta línea carga la biblioteca "ggplot2", que es una de las bibliotecas más utilizadas para crear gráficos en R. Proporciona una interfaz fácil de usar para crear una amplia variedad de gráficos, incluyendo gráficos de dispersión, líneas, barras, y muchos más.

Tamaño_Poblacion <- 8: Esta línea define la variable "Tamaño_Poblacion" y le asigna el valor 8. Esta variable representa el tamaño de la población de individuos en el algoritmo genético. En este caso, se establece en 8, pero se puede cambiar según sea necesario.

Tamaño_Individuo <- 8: Similar a la anterior, esta línea define la variable "Tamaño_Individuo" y le asigna el valor 8. Esta variable representa la longitud de cada individuo en la población. También se establece en 8 en este caso.

```
Generar_IndividuosAleatorios <- function(length) {
  individuo <- paste(sample(c(0, 1), length, replace = TRUE), collapse = "")
  if (nchar(individuo) > 8) {
    individuo <- substr(individuo, 1, 8)
    print("Se indicó un tamaño de cadena de más de 8 caracteres. Pero eso no
es posible")
    print("Se escribirán individuos sólo con los primeros 8 caracteres.")
  }

  return(individuo)
}
```

1. **Generar_IndividuosAleatorios <- function(length)**: Esta línea define una nueva función llamada **Generar_IndividuosAleatorios**, que toma un argumento **length**. Esta función se utiliza para generar individuos aleatorios para la población del algoritmo genético.
2. **individuo <- paste(sample(c(0, 1), length, replace = TRUE), collapse = "")**: En esta línea, se utiliza la función **sample** para generar una muestra aleatoria de longitud **length** a partir de los elementos 0 y 1. Con **replace = TRUE**, se permite que los elementos se seleccionen con reemplazo, lo que significa que un mismo elemento puede aparecer más de una vez en la muestra. Luego, la función **paste** se utiliza para concatenar los elementos de la muestra en una cadena de caracteres. La opción **collapse = ""** indica que no se debe insertar ningún separador entre los elementos de la muestra.
3. **if (nchar(individuo) > 8) {...}**: Esta estructura condicional verifica si la longitud del individuo generado es mayor que 8 caracteres. Si es así, significa que se generaron más de 8 caracteres aleatorios, lo cual no es posible según las restricciones del problema. En este

caso, se trunca la cadena utilizando la función **substr** para conservar solo los primeros 8 caracteres. Además, se imprime un mensaje de advertencia para informar al usuario sobre esta situación.

4. **return(individuo)**: Esta línea devuelve el individuo generado. La función retorna la cadena de caracteres que representa al individuo aleatorio.

```
Generar_Poblacion <- function(Tamaño_Poblacion, Tamaño_Individuo,
Poblacion_Predefinida = NULL) {
  if (!is.null(Poblacion_Predefinida)) {
    Poblacion <- Poblacion_Predefinida
  } else {
    CuentaRepeticion <- 0
    Poblacion <- character(Tamaño_Poblacion)
    Individuos_Generados <- character()
    max_intentos <- 10

    if (Tamaño_Individuo < 8) {
      print("Se indicó un tamaño de cadena de menos de 8 caracteres. Pero
eso no es posible")
      print("Se escribirán individuos de 8 caracteres.")
      Tamaño_Individuo <- 8
    }

    if(Tamaño_Poblacion <= 7){
      print(" - - - - - - - -¡¡ADVERTENCIA!! - - - - - - -")
      print("La población introducida es tan pequeña que no habrá
suficientes individuos para combinar.")
      print("El programa está hecho para funcionar con 8")
      cat("El programa no  se detendrá, pero así no debería operar.",
"\n\n")
    }

    for (i in 1:Tamaño_Poblacion) {
      intentos <- 0
      repeat {
        nuevo_individuo <- Generar_IndividuosAleatorios(Tamaño_Individuo)
        if (!(nuevo_individuo %in% Individuos_Generados)) {
          intentos <- intentos - 1
          Poblacion[i] <- nuevo_individuo
          Individuos_Generados <- c(Individuos_Generados, nuevo_individuo)
          break
        } else {
          intentos <- intentos + 1
          if (intentos >= max_intentos) {
```

```
            cat("No hay más combinaciones posibles sin repetir los números.
Permitida la repetición.\n")
            Poblacion[i] <- nuevo_individuo
            break
          }
        }
      }
    }
  }
  return(Poblacion)
}
```

1. **Generar_Poblacion <- function(Tamaño_Poblacion, Tamaño_Individuo, Poblacion_Predefinida = NULL)**: Esta línea define una función llamada **Generar_Poblacion** que toma tres argumentos: **Tamaño_Poblacion**, que representa el número de individuos en la población; **Tamaño_Individuo**, que representa la longitud de cada individuo; y **Poblacion_Predefinida**, que permite especificar una población inicial si es necesario.

2. **if (!is.null(Poblacion_Predefinida)) {...} else {...}**: Esta estructura condicional comprueba si se proporcionó una población predefinida. Si es así, se asigna esa población a la variable **Poblacion**. De lo contrario, se generará una nueva población aleatoria.

3. **CuentaRepeticion <- 0**: Esta línea inicializa una variable llamada **CuentaRepeticion** que no se utiliza en el resto de la función. Podría ser un remanente de una versión anterior del código.

4. **Poblacion <- character(Tamaño_Poblacion)**: Aquí se crea un vector llamado **Poblacion** con longitud igual a **Tamaño_Poblacion**. Este vector almacenará los individuos de la población.

5. **Individuos_Generados <- character()**: Se inicializa un vector vacío llamado **Individuos_Generados**. Este vector se utilizará para realizar un seguimiento de los individuos generados para evitar duplicados en la población.

6. **max_intentos <- 10**: Se define la variable **max_intentos** con un valor de 10. Esta variable se utiliza para limitar el número máximo de intentos para generar un individuo único antes de permitir la repetición.

7. **if (Tamaño_Individuo < 8) {...}**: Esta estructura condicional comprueba si el tamaño de cada individuo es menor que 8. Si es así, se imprime un mensaje de advertencia informando al usuario que los individuos deben tener al menos 8 caracteres. Luego se ajusta el tamaño del individuo a 8.

8. **if(Tamaño_Poblacion <= 7) {...}**: Aquí se verifica si el tamaño de la población es menor o igual a 7. Si es así, se imprime una advertencia indicando que la población es demasiado pequeña para el funcionamiento adecuado del programa. Se sugiere que el programa funciona mejor con una población de al menos 8 individuos.

9. **for (i in 1**

ño_Poblacion) {...}: Este bucle **for** se utiliza para generar cada individuo de la población. Itera sobre el rango de 1 a **Tamaño_Poblacion**.

10. **repeat {...}**: Se utiliza un bucle **repeat** para intentar generar un nuevo individuo único hasta que se logre. Este bucle se ejecuta hasta que se rompa explícitamente.
11. **nuevo_individuo <- Generar_IndividuosAleatorios(Tamaño_Individuo)**: Se genera un nuevo individuo aleatorio utilizando la función **Generar_IndividuosAleatorios** con el tamaño especificado por **Tamaño_Individuo**.
12. **if (!(nuevo_individuo %in% Individuos_Generados)) {...} else {...}**: Se verifica si el nuevo individuo generado ya está presente en la lista de individuos generados previamente. Si el individuo es único, se agrega a la población y a la lista de individuos generados. De lo contrario, se aumenta el contador de intentos y se vuelve a intentar generar otro individuo único.
13. **return(Poblacion)**: La función devuelve la población completa una vez que se han generado todos los individuos.

```
Evaluar_Mitad <- function(Mitad) {
    Cuenta <- 0
    Cuenta_1s <- 0
    Cuenta_0s <- 0

  #Si se detecta 1, se suma 1 🚫
    for (i in 1:length(Mitad)) {
      bit <- Mitad[i]

      if (bit == "1") {
        Cuenta <- Cuenta + 1
        Cuenta_1s <- Cuenta_1s + 1
      }
  #Si se detecta 0 se resta 1 🚫
    if (bit == "0") {
      Cuenta <- Cuenta - 1
      Cuenta_0s <- Cuenta_0s + 1
    }

    # Si no estamos en el primer caracter
    # Ya podemos comparar con el anterior
    if (i > 1 && Mitad[i - 1] == bit) {
  # Si hay doble "1", se suma 1🚫
      if (bit == "1") {
        Cuenta <- Cuenta + 1
      }
  #Si hay doble "0", se resta 1🚫
      if (bit == "0") {
```

```
        Cuenta <- Cuenta - 1
      }
    }
    }


  #Y solo hasta que acabemos de contar a la mitad
  #Y no en cada nuevo bit contado...

  #Si hay más 1s que 0s, se suma 1🚫
    if (Cuenta_1s > Cuenta_0s) {
        Cuenta <- Cuenta + 1
  #Si hay más 0s que 1s, se resta 1🚫
    } else if (Cuenta_0s > Cuenta_1s) {
    Cuenta <- Cuenta - 1
    }

  return(Cuenta)
}
```

1. **Evaluar_Mitad <- function(Mitad)**: Esta línea define una función llamada **Evaluar_Mitad** que toma un argumento llamado **Mitad**, que representa una mitad de un individuo en la población.
2. **Cuenta <- 0**: Se inicializa una variable llamada **Cuenta** con un valor de 0. Esta variable se utiliza para llevar la cuenta del puntaje de la mitad del individuo.
3. **Cuenta_1s <- 0** y **Cuenta_0s <- 0**: Se inicializan dos variables, **Cuenta_1s** y **Cuenta_0s**, con un valor de 0. Estas variables se utilizan para contar la cantidad de unos y ceros en la mitad del individuo, respectivamente.
4. **for (i in 1**

(Mitad)) {...}: Este bucle **for** itera sobre cada bit en la mitad del individuo.

5. **bit <- Mitad[i]**: Se asigna el valor del bit actual a la variable **bit**.
6. **if (bit == "1") {...}**: Si el bit actual es igual a "1", se incrementa la variable **Cuenta** en 1 y se incrementa la variable **Cuenta_1s** en 1.
7. **if (bit == "0") {...}**: Si el bit actual es igual a "0", se decrementa la variable **Cuenta** en 1 y se incrementa la variable **Cuenta_0s** en 1.
8. **if (i > 1 && Mitad[i - 1] == bit) {...}**: Esta condición comprueba si el bit actual es igual al bit anterior y si no estamos en el primer bit. Si se cumple, significa que hay dos bits iguales consecutivos.
9. **if (bit == "1") {...}** y **if (bit == "0") {...}**: Dentro de esta condición, se incrementa o decrementa la variable **Cuenta** según el valor del bit.

10. **if (Cuenta_1s > Cuenta_0s) {...} else if (Cuenta_0s > Cuenta_1s) {...}**: Esta condición compara la cantidad de unos y ceros en la mitad del individuo. Si hay más unos que ceros, se incrementa la variable **Cuenta** en 1. Si hay más ceros que unos, se decrementa la variable **Cuenta** en 1.

11. **return(Cuenta)**: La función devuelve el valor final de **Cuenta**, que representa el puntaje de la mitad del individuo.

```
Evaluar_Individuo <- function(individual) {
  Vector_Binario <- as.integer(strsplit(individual, "")[[1]])

  PrimerMitad <- Vector_Binario[1:(length(Vector_Binario) / 2)]
  SegundaMitad <- Vector_Binario[(length(Vector_Binario) / 2 +
1):length(Vector_Binario)]

  Calificacion_PrimerMitad <- Evaluar_Mitad(PrimerMitad)
    print(paste("La primer mitad está calificada como:  ",
Calificacion_PrimerMitad))
  Calificacion_SegundaMitad <- Evaluar_Mitad(SegundaMitad)
    print(paste("La segunda mitad está calificada como:  ",
Calificacion_SegundaMitad))
  print("   -   -   -   -   -   -   -   -   -   -   -")

  PuntuacionTotal <- Calificacion_PrimerMitad + Calificacion_SegundaMitad
#Sumando ambas mitades
  return(PuntuacionTotal)
}
```

1. **Evaluar_Individuo <- function(individual)**: Esta línea define la función **Evaluar_Individuo** que toma un argumento llamado **individual**, que representa un individuo en la población.

2. **Vector_Binario <- as.integer(strsplit(individual, "")[[1]])**: Esta línea divide el individuo en una lista de caracteres y luego convierte esos caracteres en un vector de números enteros. Cada elemento del vector representa un bit del individuo.

3. **PrimerMitad <- Vector_Binario[1:(length(Vector_Binario) / 2)]**: Se extrae la primera mitad del individuo dividiendo el vector de bits a la mitad.

4. **SegundaMitad <- Vector_Binario[(length(Vector_Binario) / 2 + 1)**

(Vector_Binario)]: Se extrae la segunda mitad del individuo, comenzando desde la mitad más uno hasta el final del vector de bits.

5. **Calificacion_PrimerMitad <- Evaluar_Mitad(PrimerMitad)**: Se llama a la función **Evaluar_Mitad** para evaluar la primera mitad del individuo y se guarda el resultado en la variable **Calificacion_PrimerMitad**.

6. **print(paste("La primer mitad está calificada como: ", Calificacion_PrimerMitad))**: Se imprime el puntaje de la primera mitad del individuo.

7. **Calificacion_SegundaMitad <- Evaluar_Mitad(SegundaMitad)**: Se llama a la función **Evaluar_Mitad** para evaluar la segunda mitad del individuo y se guarda el resultado en la variable **Calificacion_SegundaMitad**.

8. **print(paste("La segunda mitad está calificada como: ", Calificacion_SegundaMitad))**: Se imprime el puntaje de la segunda mitad del individuo.

9. **print(" - - - - - - - - - - ")**: Se imprime una línea separadora para mejorar la legibilidad de la salida.

10. **PuntuacionTotal <- Calificacion_PrimerMitad + Calificacion_SegundaMitad**: Se calcula la puntuación total del individuo sumando los puntajes de ambas mitades.

11. **return(PuntuacionTotal)**: La función devuelve la puntuación total del individuo.

```r
Evaluar_Poblacion <- function(Poblacion) {
  Putuaciones <- numeric(length(Poblacion))
  for (i in 1:length(Poblacion)) {
    Putuaciones[i] <- Evaluar_Individuo(Poblacion[i])
  }
  print(Putuaciones)
  return(Putuaciones)
}
```

La función **Evaluar_Poblacion** toma la población completa y devuelve las puntuaciones de cada individuo en forma de un vector. Aquí está el desglose de la función:

1. **Evaluar_Poblacion <- function(Poblacion)**: Esta línea define la función **Evaluar_Poblacion** que toma un argumento llamado **Poblacion**, que representa la población completa de individuos.

2. **Putuaciones <- numeric(length(Poblacion))**: Se crea un vector numérico llamado **Putuaciones** con la misma longitud que la población. Este vector se utilizará para almacenar las puntuaciones de cada individuo.

3. **for (i in 1(Poblacion)) { ... }**: Se inicia un bucle **for** que recorre cada individuo en la población.

4. **Putuaciones[i] <- Evaluar_Individuo(Poblacion[i])**: Para cada individuo en la población, se llama a la función **Evaluar_Individuo** para obtener su puntuación, y se guarda esa puntuación en la posición correspondiente del vector **Putuaciones**.

5. **print(Putuaciones)**: Se imprime el vector de puntuaciones en la consola para que el usuario pueda ver las puntuaciones de todos los individuos en la población.

6. **return(Putuaciones)**: La función devuelve el vector **Putuaciones**, que contiene las puntuaciones de todos los individuos en la población.

```r
Calcular_PuntuacionTotal <- function(Putuaciones) {
  PuntuacionTotal <- sum(Putuaciones)
  return(PuntuacionTotal)
}
```

La función **Calcular_PuntuacionTotal** calcula la puntuación total de una población sumando las puntuaciones individuales de cada individuo. Aquí está el detalle de la función:

1. **Calcular_PuntuacionTotal <- function(Putuaciones)**: Esta línea define la función **Calcular_PuntuacionTotal**, que toma un argumento llamado **Putuaciones**, que es un vector que contiene las puntuaciones individuales de los individuos en la población.
2. **PuntuacionTotal <- sum(Putuaciones)**: Utilizando la función **sum**, se suman todas las puntuaciones individuales en el vector **Putuaciones**, lo que da como resultado la puntuación total de la población.
3. **return(PuntuacionTotal)**: La función devuelve la puntuación total de la población.

```r
Elegir_MejoresIndividuos <- function(Poblacion, Putuaciones,
MejorIndividuo_Indice) {
  Individuos_Ordenados <- order(Putuaciones, decreasing = TRUE)
  MejoresIndividuos <-
Poblacion[Individuos_Ordenados[1:MejorIndividuo_Indice]]

  # Imprimir los mejores individuos seleccionados
  print("Mejores individuos seleccionados:")
  print(MejoresIndividuos)

  return(MejoresIndividuos)
}
```

La función **Elegir_MejoresIndividuos** selecciona los mejores individuos de una población basándose en sus puntuaciones. Aquí está la explicación detallada de la función:

1. **Elegir_MejoresIndividuos <- function(Poblacion, Putuaciones, MejorIndividuo_Indice)**: Esta línea define la función **Elegir_MejoresIndividuos**, que toma tres argumentos:
    - **Poblacion**: Vector que contiene los individuos de la población.
    - **Putuaciones**: Vector que contiene las puntuaciones individuales de los individuos en la población.
    - **MejorIndividuo_Indice**: Número entero que indica cuántos de los mejores individuos deben seleccionarse.
2. **Individuos_Ordenados <- order(Putuaciones, decreasing = TRUE)**: Utilizando la función **order**, los índices de las puntuaciones en **Putuaciones** se ordenan de mayor a menor y se almacenan en el vector **Individuos_Ordenados**.

3. **MejoresIndividuos <- Poblacion[Individuos_Ordenados[1]]**: Se seleccionan los primeros **MejorIndividuo_Indice** índices de **Individuos_Ordenados** y se usan para indexar la población original **Poblacion**, seleccionando así los mejores individuos.
4. **print("Mejores individuos seleccionados:")**: Se imprime un mensaje indicando que se van a mostrar los mejores individuos seleccionados.
5. **print(MejoresIndividuos)**: Se imprime la lista de los mejores individuos seleccionados.
6. **return(MejoresIndividuos)**: La función devuelve los mejores individuos seleccionados.

```
Cruzar_Poblacion <- function(ind1, ind2) {
  midpoint <- nchar(ind1) / 2
  child1 <- paste0(substr(ind1, 1, midpoint), substr(ind2, midpoint + 1,
nchar(ind2)))
  child2 <- paste0(substr(ind2, 1, midpoint), substr(ind1, midpoint + 1,
nchar(ind1)))

  # Imprimir los individuos y sus hijos después del cruce
  print(paste("Cruzando:", ind1, "y", ind2))
  print(paste("Resulta en hijos:", child1, "y", child2))

  return(c(child1, child2))
}
```

La función **Cruzar_Poblacion** cruza dos individuos para producir nuevos individuos descendientes. Aquí está la explicación detallada de la función:

1. **Cruzar_Poblacion <- function(ind1, ind2)**: Esta línea define la función **Cruzar_Poblacion**, que toma dos argumentos:
    - **ind1**: El primer individuo a cruzar.
    - **ind2**: El segundo individuo a cruzar.
2. **midpoint <- nchar(ind1) / 2**: Calcula el punto medio de la longitud del primer individuo **ind1**.
3. **child1 <- paste0(substr(ind1, 1, midpoint), substr(ind2, midpoint + 1, nchar(ind2)))**: Crea el primer hijo combinando la primera mitad del primer individuo **ind1** con la segunda mitad del segundo individuo **ind2**.
4. **child2 <- paste0(substr(ind2, 1, midpoint), substr(ind1, midpoint + 1, nchar(ind1)))**: Crea el segundo hijo combinando la primera mitad del segundo individuo **ind2** con la segunda mitad del primer individuo **ind1**.
5. **print(paste("Cruzando:", ind1, "y", ind2))**: Imprime un mensaje indicando los individuos que están siendo cruzados.

6. **print(paste("Resulta en hijos:", child1, "y", child2))**: Imprime un mensaje mostrando los hijos resultantes del cruce.

7. **return(c(child1, child2))**: La función devuelve un vector que contiene los dos hijos resultantes del cruce.

```
Generar_Nueva_Poblacion <- function(Poblacion, Putuaciones,
Tamaño_Poblacion) {
  MejoresIndividuos <- Elegir_MejoresIndividuos(Poblacion, Putuaciones,
Tamaño_Poblacion)
  Nueva_Poblacion <- character()

  for (i in seq(1, length(MejoresIndividuos))) {
    for (j in seq(i + 1, length(MejoresIndividuos))) {
      if (length(Nueva_Poblacion) < Tamaño_Poblacion) {
        children <- Cruzar_Poblacion(MejoresIndividuos[i],
MejoresIndividuos[j])
        Nueva_Poblacion <- c(Nueva_Poblacion, children[1])

        if (length(Nueva_Poblacion) < Tamaño_Poblacion) {
          Nueva_Poblacion <- c(Nueva_Poblacion, children[2])
        } else {
          break
        }
      }
    }
    if (length(Nueva_Poblacion) >= Tamaño_Poblacion) {
      break
    }
  }


  print("Nueva población generada:")
  print(Nueva_Poblacion)

  return(Nueva_Poblacion)
}
```

La función **Generar_Nueva_Poblacion** se encarga de generar una nueva población a partir de los mejores individuos seleccionados de la generación anterior. Aquí está la explicación detallada de la función:

1. **Generar_Nueva_Poblacion <- function(Poblacion, Putuaciones, Tamaño_Poblacion)**: Esta línea define la función **Generar_Nueva_Poblacion**, que toma tres argumentos:

    - **Poblacion**: La población actual.

- **Puntuaciones**: Las puntuaciones de cada individuo en la población actual.
- **Tamaño_Poblacion**: El tamaño deseado de la nueva población.

2. **MejoresIndividuos <- Elegir_MejoresIndividuos(Poblacion, Putuaciones, Tamaño_Poblacion)**: Llama a la función **Elegir_MejoresIndividuos** para seleccionar los mejores individuos de la población actual.
3. **Nueva_Poblacion <- character()**: Inicializa un vector vacío para almacenar la nueva población.
4. **for (i in seq(1, length(MejoresIndividuos))) {...}**: Itera sobre los mejores individuos seleccionados.
5. **for (j in seq(i + 1, length(MejoresIndividuos))) {...}**: Itera sobre los mejores individuos restantes para cruzarlos con el individuo actual.
6. **if (length(Nueva_Poblacion) < Tamaño_Poblacion) {...}**: Verifica si el tamaño de la nueva población es menor que el tamaño deseado.
7. **children <- Cruzar_Poblacion(MejoresIndividuos[i], MejoresIndividuos[j])**: Cruza los individuos actuales para generar hijos utilizando la función **Cruzar_Poblacion**.
8. **Nueva_Poblacion <- c(Nueva_Poblacion, children[1])**: Agrega el primer hijo generado a la nueva población.
9. **if (length(Nueva_Poblacion) < Tamaño_Poblacion) {...} else {...}**: Verifica si el tamaño de la nueva población sigue siendo menor que el tamaño deseado y, en caso afirmativo, agrega el segundo hijo generado a la nueva población.
10. **if (length(Nueva_Poblacion) >= Tamaño_Poblacion) {...}**: Verifica si el tamaño de la nueva población alcanzó el tamaño deseado y, en caso afirmativo, sale del bucle.
11. **print("Nueva población generada:")**: Imprime un mensaje indicando que se ha generado una nueva población.
12. **print(Nueva_Poblacion)**: Imprime la nueva población generada.
13. **return(Nueva_Poblacion)**: Devuelve la nueva población generada.

```r
Mutar_Bit <- function(individual, position) {
  bits <- unlist(strsplit(individual, ""))
  position <- ((position - 1) %% length(bits)) + 1
  if (bits[position] == "0") {
    bits[position] <- "1"
  } else {
    bits[position] <- "0"
  }
  Mutar_Individual <- paste(bits, collapse = "")
  return(Mutar_Individual)
}
```

La función **Mutar_Bit** se encarga de mutar un bit en una posición específica de un individuo. Aquí está la explicación detallada de la función:

1. **Mutar_Bit <- function(individual, position)**: Esta línea define la función **Mutar_Bit**, que toma dos argumentos:
    - **individual**: El individuo al que se le realizará la mutación.
    - **position**: La posición del bit que se va a mutar.
2. **bits <- unlist(strsplit(individual, ""))**: Divide el individuo en una lista de bits utilizando la función **strsplit** y luego convierte esta lista en un vector unidimensional utilizando **unlist**.
3. **position <- ((position - 1) %% length(bits)) + 1**: Calcula la posición ajustada para la mutación. Si la posición es mayor que la longitud del individuo, se realiza el módulo para asegurarse de que la posición esté dentro de los límites del individuo.
4. **if (bits[position] == "0") {...} else {...}**: Verifica si el bit en la posición especificada es un "0" o un "1" y realiza la mutación correspondiente cambiando el bit a su opuesto.
5. **Mutar_Individual <- paste(bits, collapse = "")**: Une los bits mutados de vuelta en un solo individuo utilizando **paste** y especificando **collapse = ""** para que no haya ningún espacio entre los bits.
6. **return(Mutar_Individual)**: Devuelve el individuo mutado.

```
Mutar_Poblacion <- function(Poblacion) {
  for (i in 1:length(Poblacion)) {
    Poblacion[i] <- Mutar_Bit(Poblacion[i], i)

    print(paste("Individuo después de la mutación en posición", i, ":",
Poblacion[i]))
  }
  return(Poblacion)
}
```

La función **Mutar_Poblacion** se encarga de mutar cada individuo en una población dada. Aquí está la explicación detallada de la función:

1. **Mutar_Poblacion <- function(Poblacion)**: Esta línea define la función **Mutar_Poblacion**, que toma un argumento:
    - **Poblacion**: La población de individuos que se van a mutar.
2. **for (i in 1**

(Poblacion)) {...}: Itera sobre cada individuo en la población.

3. **Poblacion[i] <- Mutar_Bit(Poblacion[i], i)**: Llama a la función **Mutar_Bit** para mutar el bit en la posición **i** del individuo **Poblacion[i]**. El bit mutado se asigna de nuevo al individuo en la población.

4. **print(paste("Individuo después de la mutación en posición", i, ":", Poblacion[i]))**: Imprime el individuo después de la mutación, mostrando la posición en la que se realizó la mutación y el estado del individuo mutado.

5. **return(Poblacion)**: Devuelve la población mutada.

```
Hacer_Todo <- function(Tamaño_Poblacion, Tamaño_Individuo,
Limite_Iteraciones, Limite_Iteraciones_NoMejora) {
  Poblacion <- Generar_Poblacion(Tamaño_Poblacion, Tamaño_Individuo)
  Mejores_Calificaciones <- c()
  Puntuaciones_Iteraciones <- data.frame(Iteracion = integer(), Puntuacion =
numeric())
  Cuenta_NoMejora <- 0
  Iteracion <- 1

  repeat {
    if(Iteracion == 1){
      cat("\n\n")
      print("
========================================================================= ")
      print(" POBLACION INICIAL:                ")
      print(Poblacion)
      Puntuaciones <- Evaluar_Poblacion(Poblacion)
      PuntuacionTotal <- Calcular_PuntuacionTotal(Puntuaciones)
      cat("Puntuación inicial: ", PuntuacionTotal)
      Mejores_Calificaciones <- c(Mejores_Calificaciones, PuntuacionTotal)
      Puntuaciones_Iteraciones <- rbind(Puntuaciones_Iteraciones,
data.frame(Iteracion = Iteracion, Puntuacion = PuntuacionTotal))

      cat("\n")
      Nueva_Poblacion <- Generar_Nueva_Poblacion(Poblacion, Puntuaciones,
Tamaño_Poblacion)
      Poblacion <- Nueva_Poblacion
      cat("\n")
      Poblacion <- Mutar_Poblacion(Poblacion)
      Puntuaciones <- Evaluar_Poblacion(Poblacion)
      PuntuacionTotal <- Calcular_PuntuacionTotal(Puntuaciones)
      cat("Puntuación después de mutar: ", PuntuacionTotal, "\n")
    } else {
      cat("\n")
      print("=================================================================
===========")
      cat("Iteración: ", (Iteracion), "\n")
      print(Poblacion)
      Puntuaciones <- Evaluar_Poblacion(Poblacion)
      PuntuacionTotal <- Calcular_PuntuacionTotal(Puntuaciones)
```

```
      cat("Puntuación inicial: ", PuntuacionTotal)
      Mejores_Calificaciones <- c(Mejores_Calificaciones, PuntuacionTotal)
      Puntuaciones_Iteraciones <- rbind(Puntuaciones_Iteraciones,
data.frame(Iteracion = Iteracion, Puntuacion = PuntuacionTotal))

      cat("\n")
      Nueva_Poblacion <- Generar_Nueva_Poblacion(Poblacion, Puntuaciones,
Tamaño_Poblacion)
      Poblacion <- Nueva_Poblacion
      cat("\n")
      Poblacion <- Mutar_Poblacion(Poblacion)
      Puntuaciones <- Evaluar_Poblacion(Poblacion)
      PuntuacionTotal <- Calcular_PuntuacionTotal(Puntuaciones)
      cat("Puntuación después de mutar: ", PuntuacionTotal, "\n")

      if (length(Mejores_Calificaciones) > 1 &&
Mejores_Calificaciones[length(Mejores_Calificaciones)] ==
Mejores_Calificaciones[length(Mejores_Calificaciones) - 1]) {
        Cuenta_NoMejora <- Cuenta_NoMejora + 1
      } else {
        Cuenta_NoMejora <- 0
      }

      if (Cuenta_NoMejora >= Limite_Iteraciones_NoMejora || Iteracion >=
Limite_Iteraciones) {
        break
      }
    }
    Iteracion <- Iteracion + 1
  }

  max_calificacion <- max(Mejores_Calificaciones)
  iteracion_max <-
Puntuaciones_Iteraciones$Iteracion[which.max(Mejores_Calificaciones)]

  cat("\n\n")
  cat("La máxima calificación alcanzada fue: ", max_calificacion, " en la
iteración: ", iteracion_max, "\n")


  grafica <- ggplot(Puntuaciones_Iteraciones, aes(x = Iteracion, y =
Puntuacion)) +
    geom_line() +
    geom_point() +
    labs(title = "Evolución de la Puntuación en Cada Iteración",
```

```
        x = "Iteración",
        y = "Puntuación") +
    theme_minimal()

  # Imprimir la gráfica
  print(grafica)

  return(Poblacion)
}
```

1. **Hacer_Todo <- function(Tamaño_Poblacion, Tamaño_Individuo, Limite_Iteraciones, Limite_Iteraciones_NoMejora)**: Esta línea define la función **Hacer_Todo**, que toma cuatro argumentos:
    - **Tamaño_Poblacion**: El tamaño de la población inicial.
    - **Tamaño_Individuo**: El tamaño de cada individuo en la población.
    - **Limite_Iteraciones**: El límite máximo de iteraciones para ejecutar el algoritmo.
    - **Limite_Iteraciones_NoMejora**: El límite máximo de iteraciones consecutivas sin mejora en la puntuación.
2. **Poblacion <- Generar_Poblacion(Tamaño_Poblacion, Tamaño_Individuo)**: Genera la población inicial utilizando la función **Generar_Poblacion**.
3. **Mejores_Calificaciones <- c()**: Inicializa un vector para almacenar las mejores calificaciones de cada iteración.
4. **Puntuaciones_Iteraciones <- data.frame(Iteracion = integer(), Puntuacion = numeric())**: Inicializa un dataframe para almacenar las puntuaciones de cada iteración.
5. **Cuenta_NoMejora <- 0**: Inicializa un contador para el número de iteraciones consecutivas sin mejora.
6. **Iteracion <- 1**: Inicializa el número de iteración en 1.
7. **repeat {...}**: Inicia un bucle **repeat** que continuará ejecutando el algoritmo hasta que se alcance alguno de los criterios de parada.
8. **if(Iteracion == 1) {...} else {...}**: Se ejecuta un bloque de código diferente en la primera iteración y en las iteraciones posteriores.
9. **max_calificacion <- max(Mejores_Calificaciones)**: Calcula la máxima calificación alcanzada durante todas las iteraciones.
10. **iteracion_max <- Puntuaciones_Iteraciones$Iteracion[which.max(Mejores_Calificaciones)]**: Obtiene la iteración en la que se alcanzó la máxima calificación.
11. **grafica <- ggplot(Puntuaciones_Iteraciones, aes(x = Iteracion, y = Puntuacion)) + ...**: Genera una gráfica que muestra la evolución de la puntuación en cada iteración.

12. **print(grafica)**: Imprime la gráfica.

13. **return(Poblacion)**: Devuelve la población resultante.

```
Estos datos no se cambian.
Tamaño_Poblacion <- Tamaño_Poblacion
Tamaño_Individuo <- Tamaño_Individuo

#Estos datos sí se pueden cambiar para hacer pruebas.
Limite_Iteraciones <- 100
Limite_Iteraciones_NoMejora <- 10

Poblacion_Final <- Hacer_Todo(Tamaño_Poblacion, Tamaño_Individuo,
Limite_Iteraciones, Limite_Iteraciones_NoMejora)

# Imprimir la población final✨🎉
print(paste("Población final: ", Poblacion_Final))
```

1. **Estos datos no se cambian**: Esto indica que los valores de **Tamaño_Poblacion** y **Tamaño_Individuo** se mantendrán constantes y se utilizarán en el resto del código.

2. **Tamaño_Poblacion <- Tamaño_Poblacion**: Esto asigna el valor de **Tamaño_Poblacion** a sí mismo, lo que parece redundante.

3. **Tamaño_Individuo <- Tamaño_Individuo**: Similar a lo anterior, asigna el valor de **Tamaño_Individuo** a sí mismo.

4. **#Estos datos sí se pueden cambiar para hacer pruebas**: Este comentario indica que los valores de **Limite_Iteraciones** y **Limite_Iteraciones_NoMejora** pueden ser modificados para realizar pruebas.

5. **Limite_Iteraciones <- 100**: Asigna el valor **100** a la variable **Limite_Iteraciones**, que representa el límite máximo de iteraciones del algoritmo genético.

6. **Limite_Iteraciones_NoMejora <- 10**: Asigna el valor **10** a la variable **Limite_Iteraciones_NoMejora**, que representa el límite máximo de iteraciones consecutivas sin mejora en la puntuación.

7. **Poblacion_Final <- Hacer_Todo(Tamaño_Poblacion, Tamaño_Individuo, Limite_Iteraciones, Limite_Iteraciones_NoMejora)**: Llama a la función **Hacer_Todo** con los parámetros especificados y almacena el resultado en la variable **Poblacion_Final**.

8. **print(paste("Población final: ", Poblacion_Final))**: Imprime la población final después de ejecutar el algoritmo genético. El emoji que le puse no es indispensable. Pero se ve lindo llegar a un final funcional y que haya una confirmación visual de ese gusto. “🎉✨”

Bibliografía:

- *Algoritmos genéticos*. (s. f.). http://ceca.uaeh.edu.mx/algoritmos_geneticos/
- *R Para Ciencia de Datos - 14 Cadenas de caracteres*. (s. f.). https://es.r4ds.hadley.nz/14-strings.html
- Rojas, R. A. (2021, 9 mayo). *Algoritmos genéticos*. Laberintos & Infinitos. http://laberintos.itam.mx/algoritmos-geneticos/
- *RPubs - Función ggplot() de ggplot2*. (s. f.). https://rpubs.com/daniballari/ggplot