Informe practica algorismes genetics

Hugo Díaz Rubio

Dario Alexander Plesa Plesa

Tabla de contenido

1. Introducción 3
1.1 Dificultad del enfoque 3
1.2 ¿Por qué un algoritmo genético? 3
2. Decisiones de diseño 4
2.1 void init_poblacion(): 4
2.2 void seleccionar_padres(): 4
2.3 void mutar(): 4
2.4 void imprimir_estado(): 5
2.5 int *ejecutar_GA(): 5
2.6 void insercioParam(): 5
2.7 void evaluaFormula() 5
2.8 void onePointCrossover() 6
2.9 void libMem() 6
3. Pruebas realizadas y resultados 7
3.1 Test `init_poblacion()` y `evaluaFormula()` 7
3.2 Test `seleccionar_padres()` 7
3.3 Test `ejecutar_GA()` 7
3.4 Test `mutar()` 7
3.5 Test `insercioParam()` 8
4. Cronología Trabajo 9
4.1. Semana 1: 9
4.2. Semana 2: 9

1. Introducción

En este proyecto abordamos el problema de “abrir” una caja fuerte cuyo candado está protegido por una contraseña binaria de 30 bits. Cada contraseña posible puede entenderse como un vector v=(v0,v1,…,v29) donde cada vi∈{0,1}. Sabemos además que la contraseña correcta satisface la siguiente ecuación:

$$\sum_{i=1}^{L} v_i \; x\, i^2 = 1977$$

En otras palabras, el problema consiste en encontrar un vector de 30 bits cuya combinación ponderada de índices al cuadrado sume exactamente 1977 (es decir 1977-1977 = 0).

1.1 Dificultad del enfoque

- Existen 2^30 ≈ 1,07×10^9 posibles combinaciones.
- Probar una a una todas las contraseñas llevaría un tiempo prohibitivo en una máquina corriente (días u semanas).
- Por ello, recurrimos a un **algoritmo evolutivo**, favoreciendo las mejores aproximaciones y descartando progresivamente las peores.

1.2 ¿Por qué un algoritmo genético?

Un **Algoritmo Genético (GA)** es un método inspirado en la selección natural y la genética biológica. Sus características principales aplicadas a nuestro problema son:

1. **Población de soluciones**: en vez de probar un candidato aislado, mantenemos un conjunto (población) de posibles contraseñas.
2. **Selección**: escoge los ejemplares con menor “error” (diferencia respecto a 1977) para reproducirse.
3. **Cruce (crossover)**: combina partes de dos “padres” para crear nuevos “hijos”, transmitiendo rasgos de ambos.
4. **Mutación**: introduce pequeñas variaciones aleatorias (negación de bits), esenciales para escapar de óptimos locales.
5. **Relevo generacional**: sustituye la población completa o parte de ella por la siguiente generación de hijos.

Este procedimiento, repetido durante varias generaciones, tiende a converger hacia soluciones de error cada vez menor, hasta llegar idealmente a 0.

2. Decisiones de diseño

Primeramente, una explicación de cómo hemos decidido diseñar los procedimientos más importantes para el funcionamiento del programa principal:

2.1 `void init_poblacion():`

Proposito: Se encarga de generar aleatoriamente la población inicial de cromosomas.

- **Parámetros**
  - poblacion: matriz de tamaño num_cromosomes × NUM_GENS.
  - num_cromosomes: número de individuos en la población (N).
- **Variables internas**
  - i recorre cada cromosoma.
  - j recorre los `NUM_GENS` genes (30 bits).
- **Justificación**
  - Usamos `rand() % 2` para garantizar sólo 0 o 1.
  - Se inicializa toda la población antes de cualquier evaluación o selección.

2.2 `void seleccionar_padres():`

Proposito: Selecciona N padres por **torneo de tamaño K** usando el array de errores (`fitness[]`) que se desea minimizar.

- **Parámetros**
  - `fitness[i]`: error absoluto de cada cromosoma i.
  - `seleccionados`: matriz de padres *temporal*.
  - `k`: tamaño del torneo (K).
- **Flujo**
  - Para cada padre p, sortea k candidatos.
  - Elige el de **menor** fitness.
  - Copia entero el vector de 30 genes a `seleccionados[p]`.
- **Ventaja**
  - Permite explorar diversidad: torneos pequeños ⇒ más exploración; torneos grandes ⇒ mayor presión selectiva.

2.3 `void mutar():`

Proposito: Función auxiliar que, dado un bit gen y una probabilidad de mutación, decide si lo invierte (0<-->1).

- **Parámetros**
  - `gen`: valor actual del bit (0 ó 1).
  - `prob_mut`: probabilidad de mutar cada gen (por ejemplo, 0.05).
- **Lógica**
  - Genera un float uniforme en [0,1).
  - Si es menor que prob_mut, devuelve 1-gen; sino, devuelve gen.
- **Ventajas de mutar bit a bit**:

- **Reutilización**: mutar() se puede usar en otros contextos sin duplicar bucles.
- **Simplicidad**: la lógica de flip de bit queda reducida a unas pocas líneas.

2.4 `void imprimir_estado():`

Proposito: Mostrar por pantalla el mejor cromosoma de cada generación, su error y su índice.

- **Parámetros**
  - gen(generacion): contador de generación.
  - mejor_error: valor mínimo de fitness[] en esta generación.
  - mejor_index: índice del cromosoma con ese error.
  - cromosoma: puntero al array de 30 bits del mejor individuo.
- **Lógica interna:** Se imprime "Generacion X | Mejor error: E | Índice: J | Genes: " y a continuación los 30 bits uno tras otro, sin espacios intermedios.

2.5 `int *ejecutar_GA():`

Lleva a cabo el ciclo evolutivo completo hasta `n_generaciones`, intentando minimizar el "error" de cada cromosoma (calculado por `evaluaFormula`). Devuelve un malloc con el mejor cromosoma encontrado (un array de `NUM_GENS` bits), o NULL si no hay memoria.

Propósito de la devolución: Queremos obtener al final el mejor cromosoma hallado (un array de `NUM_GENS` bits) y devolverlo al llamador para que lo use; por ejemplo, para imprimirlo o guardar la solución óptima.

2.6 `void insercioParam():`

Propósito:
Función que hace de wrapper para llamar a los getters y pedir los valores al usuario.

- **Parámetros**
  - nGene: puntero entero de la variable nGeneracions.
  - nCromo: puntero entero de la variable nCromosomes.
  - probMut: puntero *float* de la variable probMutacio.
  - kParam: puntero entero de la variable kParametre.
- **Lógica interna:**
  - La funcion pide para cada variable introducir un valor, usando un *do – while* hace un control de error sobre el parámetro que el usuario introduce y en caso positivo imprime por pantalla el valor introducide y pide el siguiente parámetro. Todo esto usando punteros que nos permite crear el *wrapper* de manera limpia agrupando asi a todos los *getters* y el control de error de cada variable.

2.7 `void evaluaFormula()`

Propósito:
Función que evalúa un cromosoma según lo que se acerca a la formula $\sum_{i=1}^{L} v_i \ x\ i^2 = 1977$

- **Parámetros**

- población: tabla de 2 dimensiones donde se guardan los cromosomas.
- fitness: tabla de 1 dimensión donde se guarda el valor *fitness* de cada cromosoma de la tabla población.
- num_cromosomas: entero con el valor que el usuario ha introducido.

- **Lógica interna**
  - La función usa dos *for* para recorrer la tabla población donde primero va comparando las columnas que vienen siendo los genes de un cromosoma y haciendo las operaciones de la formula para luego restar la suma total a 1977 para determinar el valor de un cromosoma, este proceso se repite "*num_cromosomes"* veces teniendo asi en la tabla de *fitness* los valores de todos los cromosomas de la población.

2.8 `void onePointCrossover()`

Propósito:
Intercambiar los valores de dos en dos cromosomas padre hasta un punto *random* en la tabla de población con la que trabajamos.

- **Parámetros**
  - taula: *array* de 2 dimensiones donde se guardan los padres
  - n_cromosomes: entero con valor introducido por el usuario
- **Lógica interna**
  - Primero el punto de crossover se escoge de manera *random,* a continuación en la tabla de población se van seleccionando de 2 en dos cromosomas padre los cuales intercambian sus valores hasta llegar al punto de crossover.

2.9 `void libMem()`

Propósito:
Wrapper que llama a las funciones libMemTaula1D y libMemTaula2D para cada tabla que usamos.

- **Parámetros**
  - población: tabla de poblacion
  - fitness: tabla de fitness
  - seleccionados: tabla de seleccionados
  - población_nueva: tabla de población nueva
  - mejor: tabla del mejor cromosoma
- **Lógica interna**
  - Llama a las funciones libMemTaula1D y libMemTaula2D según si la tabla introducida es de 1 o 2 dimensiones para desalocar su memoria.

3. Pruebas realizadas y resultados

Para comprobar el correcto funcionamiento de los procedimientos hemos desarrollado las siguientes pruebas:

3.1 Test `init_poblacion()` y `evaluaFormula()`

- **Qué hace:**
  - Reserva memoria para poblacion y fitness.
  - Llama a `init_poblacion(pobl, nCromo)`: rellena cada cromosoma con 30 bits aleatorios.
  - Llama a `evaluaFormula(pobl, fitness, nCromo)`: calcula el "error" de cada individuo según la formula descrita anteriormente.
- **Qué comprobamos:**
  - Que `fitness[i]` tenga un valor entero ≥ 0 para cada i.
  - Imprimimos todos los `fitness[i]` para verificar que no hay basura ni valores negativos.

3.2 Test `seleccionar_padres()`

- **Qué hace:**
  - Usando la población inicial y su `fitness[]`, genera con torneo de tamaño `kParam` un array seleccionados de padres.
- **Qué comprobamos:**
  - Que cada `seleccionados[i]` sea un cromosoma de 30 bits (solo 0 y 1).
  - Que el array `seleccionados` tenga exactamente `nCromosomes` filas.

3.3 Test `ejecutar_GA()`

- **Qué cubre:**
  - Invoca todo el flujo GA: evaluación, selección, crossover, mutación, swap, impresión con imprimir_estado().
- **Qué comprobamos:**
  - **Ejecución completa**: la función recorre todas las generaciones previstas (o se detiene al hallar error 0) sin interrumpirse por fallos de memoria.
  - **Resultado correcto**: devuelve un puntero mejor a un array de 30 enteros, todos ellos **0** o **1**.
  - **Salida en pantalla**: imprime una línea por cada generación con el formato

3.4 Test `mutar()`

- **Qué hace:**
  - Llama 5 veces a `mutar(original, probMutacio)` con original $\in \{0,1\}$ elegido al azar.
- **Qué comprobamos:**
  - Que mutar devuelva siempre 0 o 1.
  - Que aproximadamente en un x% (0-100%] de los casos (según `probMutacio`) el bit se invierta.

3.5 Test `insercioParam()`

- **Que hace:**
  - Declara punteros los cuales apuntan a las variables que el usuario tiene que introducir y luego llama la funcion insercioParam() el cual pide al usuario los valores de las variables.
- **Que comprobamos:**
  - Comprobamos que el control de errores funciona con parámetros erróneos que pueda introducir el usuario según la variable
  - Para los valores *int*: “El parametre ha de ser mes gran que 0 i mes petit que 9999”.
  - Para el valor *float:* “El parametre ha de ser mes gran que 0 i mes petit que 1”.

4. Cronología Trabajo

4.1. Semana 1:

Hugo:

- Entendimiento de la práctica y división de tareas equitativa según carga de trabajo.
- Declaración de funciones a trabajar sin una implementación aún hecha.
- Implementación de función `init_poblacion()` y correcciones según memoria dinámica.
- Implementación de función `mutar()`
- Implementación de función `seleccionar_padres()`
- Implementación temprana de función `ejecutar_GA()` con funciones faltantes por implementar.

Dario:

- Creación del proyecto i estructura de archivos.
- Creación de las funciones para liberar memoria i evaluar el fitness con la función `evaluaFormula()`.
- Creación de *getters* i la función `insercioParam ()`
- Control de errores implementado en `insercioParam()` y creación de las funciones: `esCorrecteInt()` y `esCorrecteFloat()`
- Creación del *wrapper* para agrupar las funciones `libMemTaula1D()` y `libMemTaula2D()` para cada tabla dinámica utilizada.
- Comentarios añadidos encima de cada función

4.2. Semana 2:

Hugo:

- Corrección tablas de 1 dimensión a 2 dimensiones con memoria dinámica.
- Corrección `ejecutar_GA()` para que devuelva mejor cromosoma.
- Implementación función `imprimir_estado()`.
- Creación de headers y división del código.
- Comentar toda función sus parámetros y funcionalidad.
- Creación de tests para comprobar funcionamiento de las funciones implementadas anteriormente y corrección de las funciones para eliminar errores del juego de pruebas esperado.

Dario:

- Creación de la función `onePointCrossover()`.
- Implementación de la función `faseSupervivencia()`.
- *testFuncions.c* creado con un `main()` alternativo donde probar las funciones por separado.
- Tests creados para las funciones `insercioParam()`, `evaluaFormula()` y `onePointCrossover()`.
- Correcciones en las funciones anteriormente implementadas para asegurar el funcionamiento correcto del programa.
- Comentarios añadidos en las funciones en los *headers.*
- Corrección del funcionamiento de la función `ejecutar_GA()` y finalización del programa.