Inteligencia Computacional Evolutiva

Fundamentos de Inteligencia Computacional

Prof. D.Sc. BARSEKH-ONJI Aboud

Facultad de Ingeniería
Universidad Anáhuac México

8 de octubre de 2025

Agenda

1. Introducción a la Computación Evolutiva
   - 1.1 Antecedentes Históricos y Fundamentos Biológicos
   - 1.2 Pioneros y Paradigmas Computacionales
2. Programación Evolutiva (EP)
   - 2.1 Algoritmo y Representación
3. Estrategias Evolutivas (ES)
   - 3.1 Evolución de las Estrategias
   - 3.2 Ejemplo de Aplicación: Optimización de Función
4. Algoritmos Genéticos (AG)
   - 4.1 El Algoritmo Genético Canónico
   - 4.2 Operadores de Cruce (Crossover)
     - Cruce para Representaciones Binarias
     - Cruce para Representaciones de Punto Flotante
     - Operadores de Cruce Multiparentales
   - 4.3 Operadores de Mutación
     - Mutación para Representaciones Binarias
     - Mutación para Representaciones de Punto Flotante
     - Operador de Macromutación: Gallina sin Cabeza
   - 4.4 Variantes de Algoritmos Genéticos

¿Qué es la Computación Evolutiva (CE)?

Inspiración: La Evolución Biológica

La Computación Evolutiva (CE) es un subcampo de la Inteligencia Artificial que se inspira directamente en los principios de la evolución biológica para desarrollar algoritmos de búsqueda y optimización.

Mecanismo General

Los algoritmos evolutivos operan sobre una **población** de soluciones candidatas, aplicando procesos de **selección**, **reproducción** y **variación** para mejorar iterativamente la calidad de las soluciones a lo largo de **generaciones**.

Antes de empezar..

Evolutionary Computing for Problem Solving

https://youtu.be/D3zUmfDd79s?si=rXT-3SIH1oIoU_4Y

Evolutionary Computation vs. Deep Learning

https://youtu.be/bRuCZyu6kQ0?si=EtU57-h5wUptorRb

Evolutionary Computation: TEDx

https://youtu.be/D3zUmfDd79s?si=BA16nkCQ1D17mARY

Antecedentes Biológicos: Primeras Ideas

Lamarckismo (Principios del s. XIX)

- Propuso el primer mecanismo coherente de evolución.
- Se basaba en la **'herencia de características adquiridas'**: los cambios en el entorno modificaban a los organismos, y estas modificaciones se transmitían a la descendencia.
- Aunque incorrecto, fue una idea revolucionaria sobre la adaptación gradual.

Selección Natural (Darwin y Wallace, 1859)

- Dentro de una población existe **variación natural**.
- Los individuos con variaciones ventajosas tienen más probabilidades de sobrevivir y reproducirse (**'supervivencia del más apto'**).
- Con el tiempo, este proceso conduce a la adaptación de las poblaciones.

La Pieza Faltante y la Síntesis Moderna

El Problema de la Herencia

La principal debilidad de la teoría de Darwin era la falta de un mecanismo de herencia. La teoría de la 'herencia por mezcla' (que sugería que las características se promediaban) contradecía la selección natural.

La Genética Mendeliana

Sin que Darwin lo supiera, Gregor Mendel demostró que la herencia es **particulada**, no por mezcla. Los rasgos se heredan a través de 'factores' discretos (hoy llamados **genes**) que no se diluyen.

La Pieza Faltante y la Síntesis Moderna

La Síntesis Moderna (Neo-Darwinismo)

Unificó la selección natural de Darwin con la genética mendeliana. La evolución es el resultado de la interacción de:

- **Variación Genética:** (Mutación y Recombinación).
- **Herencia:** Transmisión de genes.
- **Selección Natural:** Supervivencia diferencial.

Este es el marco conceptual que inspira directamente a la Computación Evolutiva.

Pioneros y Paradigmas Fundacionales

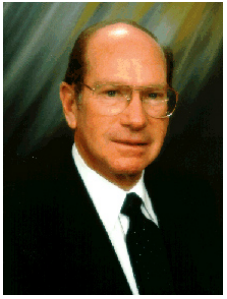
Primeros Vislumbres Computacionales

La idea de simular la evolución en una computadora es casi tan antigua como la computación misma. Visionarios como **Alan Turing** ya lo sugerían en 1950. En las décadas de 1950 y 1960, pioneros como Barricelli, Fraser y Box realizaron las primeras simulaciones de procesos evolutivos.

Los Tres Pilares de la Computación Evolutiva

En la década de 1960, surgieron de forma independiente tres líneas de investigación que se convertirían en los pilares del campo, cada una con un enfoque y una motivación distintos.

Los Tres Paradigmas Fundacionales



Programación Evolutiva (EP)

Lawrence J. Fogel (EE.UU.)

- **Motivación:** IA, crear sistemas que se adaptan y predicen.
- **Enfoque:** Evolucionar el **comportamiento** (fenotipo) de las soluciones.
- **Operador Clave: Mutación**.

Los Tres Paradigmas Fundacionales

Estrategias Evolutivas (ES)

Rechenberg y Schwefel (Alemania)

- **Motivación:** Optimización de problemas de **ingeniería** con parámetros de valor real.
- **Concepto Clave: Auto-adaptación** (el algoritmo aprende y ajusta sus propios parámetros).

Los Tres Paradigmas Fundacionales

Algoritmos Genéticos (GA)

John H. Holland (EE.UU.)

- **Motivación:** Entender teóricamente los principios de la **adaptación**.
- **Enfoque:** Énfasis en la **codificación** de la solución (genotipo).
- **Operador Clave: Cruce (Crossover)**.

Comparación de los Paradigmas Fundacionales

Característica	Programación Evolutiva	Estrategias Evolutivas	Algoritmo Genético
Representación	Fenotípica (Real, Autómatas)	Fenotípica (Valores Reales)	Genotípica (Binaria tradic.)
Operador Principal	Mutación	Mutación	Cruce (Crossover)
Auto-Adaptación	Posible	Característica central	No es estándar
Recombinación	Ausente originalmente	Presente (discreta, intermedia)	Operador principal
Selección	Probabilística (Torneo)	Determinística	Probabilística (Ruleta)

Agenda

1. Introducción a la Computación Evolutiva
   1.1 Antecedentes Históricos y Fundamentos Biológicos
   1.2 Pioneros y Paradigmas Computacionales
2. Programación Evolutiva (EP)
   2.1 Algoritmo y Representación
3. Estrategias Evolutivas (ES)
   3.1 Evolución de las Estrategias
   3.2 Ejemplo de Aplicación: Optimización de Función
4. Algoritmos Genéticos (AG)
   4.1 El Algoritmo Genético Canónico
   4.2 Operadores de Cruce (Crossover)
      Cruce para Representaciones Binarias
      Cruce para Representaciones de Punto Flotante
      Operadores de Cruce Multiparentales
   4.3 Operadores de Mutación
      Mutación para Representaciones Binarias
      Mutación para Representaciones de Punto Flotante
      Operador de Macromutación: Gallina sin Cabeza
   4.4 Variantes de Algoritmos Genéticos

Programación Evolutiva (EP)

Origen y Premisa

Propuesta por Lawrence J. Fogel, la EP se basa en la idea de que la **inteligencia es un comportamiento adaptativo**.

Enfoque Principal

La EP se centra en la evolución a nivel **fenotípico**, es decir, en el **comportamiento observable** de las soluciones, más que en su codificación genética subyacente.

El Algoritmo

- Cada solución (padre) produce una descendencia a través de la **mutación**.
- La mutación es el **único** operador de variación; no se utiliza el cruce.
- Los mejores individuos de la combinación de padres e hijos son seleccionados para formar la siguiente generación.

Aplicaciones y Evolución del Paradigma

Evolución del Paradigma

Con el tiempo, la Programación Evolutiva se extendió más allá de los autómatas para abordar la **optimización de funciones continuas**.

- En este contexto, los individuos son vectores de números reales.
- Una característica clave de la EP moderna es la **auto-adaptación**: cada individuo evoluciona sus propios parámetros de mutación, permitiendo que el algoritmo ajuste dinámicamente su comportamiento de búsqueda.

Aplicaciones y Evolución del Paradigma

Aplicaciones

Las aplicaciones de la Programación Evolutiva incluyen:

- Predicción y control automático.
- Diseño y entrenamiento de redes neuronales.
- Reconocimiento de patrones.
- Optimización de problemas complejos como el del viajante de comercio (TSP).

Agenda

1. Introducción a la Computación Evolutiva
   1.1 Antecedentes Históricos y Fundamentos Biológicos
   1.2 Pioneros y Paradigmas Computacionales
2. Programación Evolutiva (EP)
   2.1 Algoritmo y Representación
3. Estrategias Evolutivas (ES)
   3.1 Evolución de las Estrategias
   3.2 Ejemplo de Aplicación: Optimización de Función
4. Algoritmos Genéticos (AG)
   4.1 El Algoritmo Genético Canónico
   4.2 Operadores de Cruce (Crossover)
      Cruce para Representaciones Binarias
      Cruce para Representaciones de Punto Flotante
      Operadores de Cruce Multiparentales
   4.3 Operadores de Mutación
      Mutación para Representaciones Binarias
      Mutación para Representaciones de Punto Flotante
      Operador de Macromutación: Gallina sin Cabeza
   4.4 Variantes de Algoritmos Genéticos

Estrategias Evolutivas (ES)

Origen y Motivación

Desarrolladas en Alemania, las Estrategias Evolutivas (ES) surgieron de la necesidad de resolver problemas de **optimización en ingeniería** que eran difíciles de abordar con métodos matemáticos tradicionales.

Enfoque Principal

Su principal aplicación siempre ha sido la optimización de **parámetros de valor real**.

Evolución de las Estrategias

La Estrategia (1+1)-ES

Es la versión más simple:

- Un solo padre genera un solo hijo mediante mutación.
- Si el hijo es mejor que el padre, lo reemplaza; de lo contrario, se descarta.
- Es una selección determinista y elitista.

Estrategias Basadas en Poblaciones

Se introdujeron poblaciones más grandes, con dos esquemas de selección principales:

- **$(\mu + \lambda)$-ES:** Los μ mejores de la **unión de padres e hijos** forman la siguiente generación. Es **elitista**.
- **(μ, λ)-ES:** Los μ mejores se seleccionan **únicamente de entre los hijos**. No es elitista, lo que le permite escapar de óptimos locales.

Ejemplo: Paso a Paso de una Iteración (1+1)-ES

Problema

Minimizar la función de Rosenbrock: $f(x_1, x_2) = 100(x_1^2 - x_2)^2 + (1 - x_1)^2$

1. **Inicialización (Padre):** Partimos de un punto inicial $\mathbf{x}^t = (-1{,}0; 1{,}0)$, con una aptitud (costo) de $f(\mathbf{x}^t) = 4{,}0$.

Ejemplo: Paso a Paso de una Iteración (1+1)-ES

Problema

Minimizar la función de Rosenbrock: $f(x_1, x_2) = 100(x_1^2 - x_2)^2 + (1 - x_1)^2$

1. **Inicialización (Padre):** Partimos de un punto inicial $\mathbf{x}^t = (-1{,}0; 1{,}0)$, con una aptitud (costo) de $f(\mathbf{x}^t) = 4{,}0$.
2. **Mutación:** Se genera un nuevo individuo (hijo) añadiendo ruido Gaussiano. Supongamos que el resultado es $\mathbf{x}^{t+1} = (-0{,}39; 1{,}57)$.

Ejemplo: Paso a Paso de una Iteración (1+1)-ES

Problema

Minimizar la función de Rosenbrock: $f(x_1, x_2) = 100(x_1^2 - x_2)^2 + (1 - x_1)^2$

1. **Inicialización (Padre):** Partimos de un punto inicial $\mathbf{x}^t = (-1{,}0; 1{,}0)$, con una aptitud (costo) de $f(\mathbf{x}^t) = 4{,}0$.
2. **Mutación:** Se genera un nuevo individuo (hijo) añadiendo ruido Gaussiano. Supongamos que el resultado es $\mathbf{x}^{t+1} = (-0{,}39; 1{,}57)$.
3. **Evaluación:** Se calcula la aptitud del nuevo individuo. $f(\mathbf{x}^{t+1}) = f(-0{,}39; 1{,}57) = 201{,}416$.

Ejemplo: Paso a Paso de una Iteración (1+1)-ES

Problema

Minimizar la función de Rosenbrock: $f(x_1, x_2) = 100(x_1^2 - x_2)^2 + (1 - x_1)^2$

1. **Inicialización (Padre):** Partimos de un punto inicial $\mathbf{x}^t = (-1{,}0; 1{,}0)$, con una aptitud (costo) de $f(\mathbf{x}^t) = 4{,}0$.
2. **Mutación:** Se genera un nuevo individuo (hijo) añadiendo ruido Gaussiano. Supongamos que el resultado es $\mathbf{x}^{t+1} = (-0{,}39; 1{,}57)$.
3. **Evaluación:** Se calcula la aptitud del nuevo individuo. $f(\mathbf{x}^{t+1}) = f(-0{,}39; 1{,}57) = 201{,}416$.
4. **Selección:**

Como $f(\text{hijo}) > f(\text{padre})$, la nueva solución es peor. En una (1+1)-ES, el hijo se descarta y el padre sobrevive para la siguiente generación. El proceso se repite.

Ejemplo de Código: (1+1)-ES para la Función de Rosenbrock

Ejemplo de la aplicació de (1+1)-ES en Python

https://github.com/AboudOnji/Ex_Fund_IC/blob/main/Cap7/listing7.1.py

Ejemplo de la aplicació de (1+1)-ES en MATLAB

https://github.com/AboudOnji/Ex_Fund_IC/blob/main/Cap7/listing7_2.m

Agenda

1. Introducción a la Computación Evolutiva
   1.1 Antecedentes Históricos y Fundamentos Biológicos
   1.2 Pioneros y Paradigmas Computacionales
2. Programación Evolutiva (EP)
   2.1 Algoritmo y Representación
3. Estrategias Evolutivas (ES)
   3.1 Evolución de las Estrategias
   3.2 Ejemplo de Aplicación: Optimización de Función
4. Algoritmos Genéticos (AG)
   4.1 El Algoritmo Genético Canónico
   4.2 Operadores de Cruce (Crossover)
      Cruce para Representaciones Binarias
      Cruce para Representaciones de Punto Flotante
      Operadores de Cruce Multiparentales
   4.3 Operadores de Mutación
      Mutación para Representaciones Binarias
      Mutación para Representaciones de Punto Flotante
      Operador de Macromutación: Gallina sin Cabeza
   4.4 Variantes de Algoritmos Genéticos

Algoritmos Genéticos (AG)

Concepto

Popularizados por John H. Holland, los Algoritmos Genéticos (AG) son modelos que simulan la **evolución genética**. A diferencia de EP y ES, el énfasis está en el **genotipo** (la codificación de la solución).

Operadores Principales

El proceso evolutivo en un AG es impulsado principalmente por:

- **Selección:** Modela la supervivencia del más apto.
- **Cruce (Crossover):** Modela la recombinación y la reproducción sexual. Es el operador de búsqueda principal.

El Algoritmo Genético Canónico

La Propuesta Original de Holland

El AG 'canónico' o clásico se define por las siguientes características específicas:

- **Representación:** Se utiliza una **cadena de bits** (*bitstring*) de longitud fija para codificar cada solución.

El Algoritmo Genético Canónico

La Propuesta Original de Holland

El AG 'canónico' o clásico se define por las siguientes características específicas:

- **Representación:** Se utiliza una **cadena de bits** (*bitstring*) de longitud fija para codificar cada solución.
- **Selección:** Se emplea la **selección proporcional a la aptitud** (a menudo implementada como una 'rueda de ruleta') para elegir a los padres.

El Algoritmo Genético Canónico

La Propuesta Original de Holland

El AG 'canónico' o clásico se define por las siguientes características específicas:

- **Representación:** Se utiliza una **cadena de bits** (*bitstring*) de longitud fija para codificar cada solución.
- **Selección:** Se emplea la **selección proporcional a la aptitud** (a menudo implementada como una 'rueda de ruleta') para elegir a los padres.
- **Cruce (Crossover):** El **cruce de un punto** se utiliza como el método principal para producir descendencia, combinando 'bloques de construcción' de los padres.

El Algoritmo Genético Canónico

La Propuesta Original de Holland

El AG 'canónico' o clásico se define por las siguientes características específicas:

- **Representación:** Se utiliza una **cadena de bits** (*bitstring*) de longitud fija para codificar cada solución.
- **Selección:** Se emplea la **selección proporcional a la aptitud** (a menudo implementada como una 'rueda de ruleta') para elegir a los padres.
- **Cruce (Crossover):** El **cruce de un punto** se utiliza como el método principal para producir descendencia, combinando 'bloques de construcción' de los padres.
- **Mutación:** La mutación (invertir un bit al azar) se considera un operador de fondo, de menor importancia, con el único fin de mantener la diversidad genética.

Operadores de Cruce (Crossover)

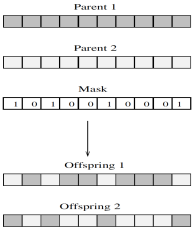
Concepto

Los operadores de cruce (o recombinación) se utilizan para crear nueva descendencia combinando el material genético de los padres. Se aplican de forma probabilística, controlada por una **tasa de cruce (p_c)**, que generalmente es alta.

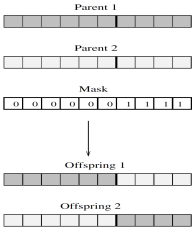
Clasificación por Aridad (Número de Padres)

- **Asexual:** Un descendiente se genera a partir de un solo padre.
- **Sexual:** Se utilizan dos padres para producir uno o dos descendientes (el más común).
- **Multirecombinación:** Se utilizan más de dos padres.

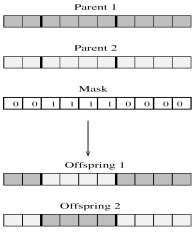
Cruce para Representaciones Binarias



(a) Uniform Crossover



(b) One-point Crossover



(c) Two-point Crossover

Cruce para Representaciones Binarias

Cruce de un Punto

- Se elige un punto de cruce al azar.
- Las cadenas de bits después de ese punto se intercambian entre los dos padres.

Cruce de Dos Puntos

- Se eligen dos puntos de cruce al azar.
- La cadena de bits **entre** estos dos puntos se intercambia.

Cruce Uniforme

- Para cada posición de bit, se decide al azar (con una probabilidad) si se intercambian o no los bits de los padres.

Ejemplo de implementación

Para ilustrar cómo funcionan estos operadores en la práctica, a continuación se presentan ejemplos de código en Python y MATLAB. En ambos casos, se definen funciones para el cruce de un punto, dos puntos y uniforme, y luego se aplican a un par de cromosomas padres de ejemplo para mostrar la descendencia resultante.

Ejemplo de cruce en Python

https://github.com/AboudOnji/Ex_Fund_IC/blob/main/Cap7/listing7_3.py

Ejemplo de cruce en MATLAB

https://github.com/AboudOnji/Ex_Fund_IC/blob/main/Cap7/listing7_4.m

Cruce para Representaciones de Punto Flotante

Recombinación Intermedia

A diferencia de los operadores discretos (que solo intercambian valores), los operadores de recombinación intermedia **mezclan** los componentes de los padres para crear nuevos valores que pueden no haber existido en la población.

Operadores Comunes

- **Cruce Lineal:** Genera tres descendientes candidatos (dos extrapolados y uno interpolado) y selecciona los dos mejores.

Cruce para Representaciones de Punto Flotante

Recombinación Intermedia

A diferencia de los operadores discretos (que solo intercambian valores), los operadores de recombinación intermedia **mezclan** los componentes de los padres para crear nuevos valores que pueden no haber existido en la población.

Operadores Comunes

- **Cruce Lineal:** Genera tres descendientes candidatos (dos extrapolados y uno interpolado) y selecciona los dos mejores.
- **Cruce Aritmético:** Crea un descendiente tomando un **promedio ponderado** de dos o más padres.

Cruce para Representaciones de Punto Flotante

Recombinación Intermedia

A diferencia de los operadores discretos (que solo intercambian valores), los operadores de recombinación intermedia **mezclan** los componentes de los padres para crear nuevos valores que pueden no haber existido en la población.

Operadores Comunes

- **Cruce Lineal:** Genera tres descendientes candidatos (dos extrapolados y uno interpolado) y selecciona los dos mejores.
- **Cruce Aritmético:** Crea un descendiente tomando un **promedio ponderado** de dos o más padres.
- **Cruce de Mezcla (BLX-α):** Elige aleatoriamente, para cada variable, un valor en un rango expandido alrededor de los padres.

Cruce para Representaciones de Punto Flotante

Recombinación Intermedia

A diferencia de los operadores discretos (que solo intercambian valores), los operadores de recombinación intermedia **mezclan** los componentes de los padres para crear nuevos valores que pueden no haber existido en la población.

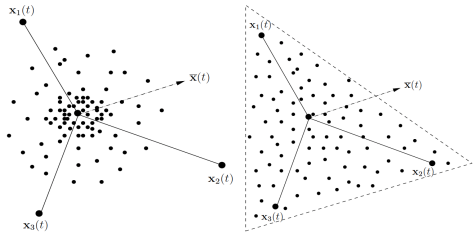
Operadores Comunes

- **Cruce Lineal:** Genera tres descendientes candidatos (dos extrapolados y uno interpolado) y selecciona los dos mejores.
- **Cruce Aritmético:** Crea un descendiente tomando un **promedio ponderado** de dos o más padres.
- **Cruce de Mezcla (BLX-α):** Elige aleatoriamente, para cada variable, un valor en un rango expandido alrededor de los padres.
- **Cruce Binario Simulado (SBX):** Simula el comportamiento del cruce de un punto binario, pero para variables de valor real.

Operadores de Cruce Multiparentales

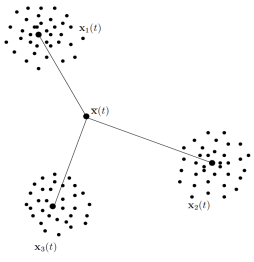
Objetivo Principal

El objetivo de los operadores multiparentales es **intensificar la exploración** del espacio de búsqueda al agregar información de más de dos padres, logrando una mayor diversidad en la descendencia.



(a) UNDX Operator

(b) SPX Operator

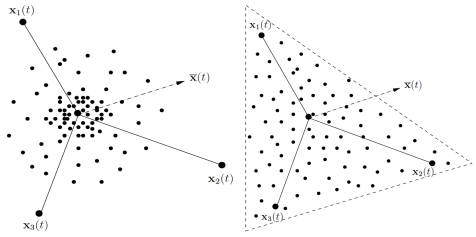


(c) PCX Operator

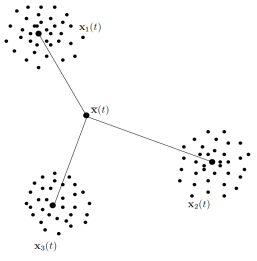
Operadores de Cruce Multiparentales

Enfoques Comunes

- **Basados en Centro de Masa (UNDX, SPX):** Generan descendencia en una región definida por el centro de masa de los padres seleccionados.
- **Centrados en los Padres (PCX):** Generan descendencia alrededor de los padres seleccionados, en lugar del centro de masa.



(a) UNDX Operator (b) SPX Operator



(c) PCX Operator

Ejemplo de implementación

Para ilustrar los operadores de cruce más avanzados que utilizan múltiples padres, a continuación se presenta un ejemplo de código para el Cruce por Escaneo de Genes en Python.

Ejemplo de cruce en Python

https://github.com/AboudOnji/Ex_Fund_IC/blob/main/Cap7/listing7_6.py

Cruce diagonal

El operador de cruce diagonal desarrollado por Eiben et al. 1994 es una generalización del cruce de n-puntos para más de dos padres. Se seleccionan $n \geq 1$ puntos de cruce y se aplican a todos los $n_\mu = n + 1$ padres. Se pueden generar uno o $n + 1$ descendientes seleccionando segmentos de los padres a lo largo de las diagonales, como se ilustra en la Figura.

Cruce diagonal

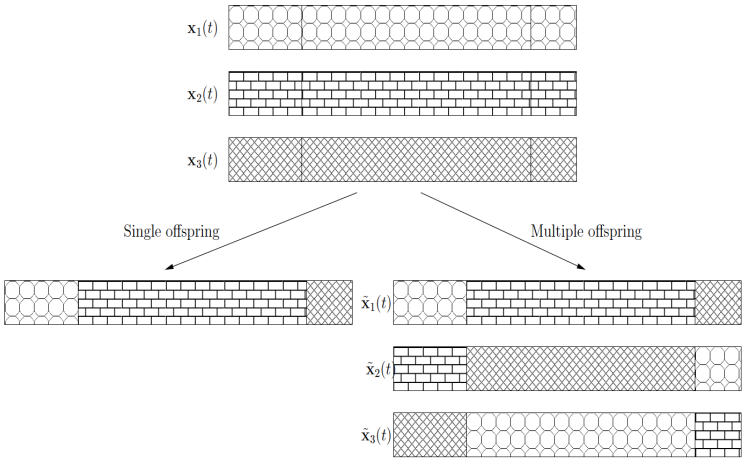


Figura 4: Ilustración del operador de cruce diagonal para $n = 2$ puntos de cruce y $n_\mu = 3$ padres. Se pueden generar uno o múltiples descendientes.

Ejemplo de implementación

Para ilustrar los operadores de cruce más avanzados que utilizan múltiples padres, a continuación se presenta un ejemplo de código para el Cruce diagonal en MATLAB.

Ejemplo de cruce diagonal en MATLAB

https://github.com/AboudOnji/Ex_Fund_IC/blob/main/Cap7/listing7_8.m

Operadores de Mutación

Objetivo Principal

El objetivo de la mutación es **introducir nuevo material genético** en un individuo, es decir, añadir **diversidad** a las características genéticas de la población. Actúa como un complemento del cruce para asegurar que todo el rango de posibles soluciones sea accesible.

Tasa de Mutación (p_m)

La mutación se aplica con una cierta probabilidad a cada gen del individuo. Esta probabilidad, conocida como tasa de mutación, suele ser un **valor pequeño** para asegurar que las buenas soluciones encontradas no se distorsionen demasiado.

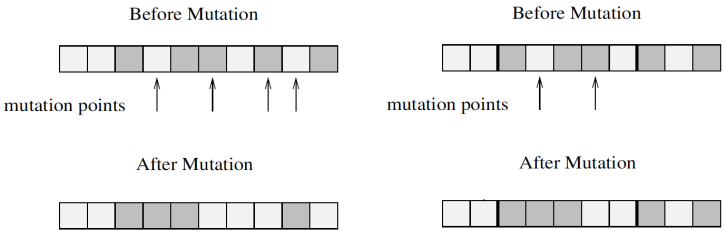
Mutación para Representaciones Binarias

Mutación Uniforme (o Aleatoria)

Cada bit del cromosoma tiene una probabilidad independiente (p_m) de ser invertido (cambiar de 0 a 1, o de 1 a 0).

Mutación en Orden (Inorder)

Se seleccionan dos puntos de mutación al azar y solo los bits **entre** estos dos puntos están sujetos a la mutación uniforme.



(a) Random Mutate (b) Inorder Mutate

Mutación Uniforme

Algorithm 1 Mutación Uniforme/Aleatoria

1: **for** $j = 1, \ldots, n_x$ **do**
2: **if** $U(0,1) \leq p_m$ **then**
3: $x'_{ij}(t) = \neg \tilde{x}_{ij}(t)$ ▷ donde $\neg$ denota el operador booleano NOT
4: **end if**
5: **end for**

Mutación en Orden

Algorithm 2 Mutación en Orden

1: Seleccionar puntos de mutación, $\xi_1, \xi_2 \sim U(1, \ldots, n_x)$.
2: **for** $j = \xi_1, \ldots, \xi_2$ **do**
3: **if** $U(0,1) \leq p_m$ **then**
4: $x'_{ij}(t) = \neg \tilde{x}_{ij}(t)$
5: **end if**
6: **end for**

Ejemplo de Código: Mutación Binaria

Para ilustrar estos operadores de mutación, a continuación se presentan ejemplos de código. El primer ejemplo, en Python, implementa la **mutación uniforme**, donde cada bit tiene una probabilidad independiente de ser invertido. El segundo ejemplo, en MATLAB, implementa la **mutación en orden**, que aplica la mutación solo a un segmento contiguo y aleatorio del cromosoma.

Ejemplo de mutación uniforme en Python

https://github.com/AboudOnji/Ex_Fund_IC/blob/main/Cap7/listing7_9.py

Ejemplo de mutación en orden en MATLAB

https://github.com/AboudOnji/Ex_Fund_IC/blob/main/Cap7/listing7_10.m

Mutación para Representaciones de Punto Flotante

Enfoque Principal

Cuando las variables de decisión son de valor real, es más eficiente aplicar operadores de mutación directamente sobre esa representación, en lugar de convertirla a binario.

Mutación Gaussiana

Es uno de los métodos más comunes. Consiste en perturbar cada variable (gen) del cromosoma añadiéndole un valor aleatorio extraído de una **distribución normal (Gaussiana)**. Este enfoque permite realizar pequeños ajustes finos a la solución, facilitando la explotación local.

Demostración de Mutación Gaussiana en MATLAB

https://github.com/AboudOnji/Ex_Fund_IC/blob/main/Cap7/listing7_11.m

Operador de Macromutación: Gallina sin Cabeza

Concepto

Este operador, propuesto por Jones, introduce una gran cantidad de diversidad en la población. Consiste en crear un descendiente recombinando un individuo padre con un **individuo completamente nuevo y generado al azar**.

Propósito

Es una forma de **mutación drástica** que puede ayudar al algoritmo a **escapar de óptimos locales** al introducir material genético totalmente nuevo en la población.

Ejemplo de aplicación: Gallina sin cabeza

El siguiente código en Python ilustra el funcionamiento del operador de **macromutación 'gallina sin cabeza'**. La función toma un cromosoma padre y lo cruza con un cromosoma completamente nuevo y generado al azar. El resultado es una forma de mutación drástica que puede introducir una gran diversidad en la población, ayudando a escapar de óptimos locales. En este ejemplo, se utiliza un cruce de un punto para la recombinación.

Ejemplo en Python

https://github.com/AboudOnji/Ex_Fund_IC/blob/main/Cap7/listing7_12.py

Variantes de Algoritmos Genéticos: GGA vs. SSGA

Brecha Generacional

Se refiere a la cantidad de superposición entre la población actual y la nueva población. La estrategia de reemplazo (cómo los hijos sustituyen a los padres) define dos clases principales de AG.

AG Generacional (GGA)

- **Estrategia: Todos** los padres de la población actual son reemplazados por la nueva generación de hijos al mismo tiempo.
- **Resultado:** No hay superposición entre generaciones (brecha generacional cero), a menos que se use elitismo.

AG de Estado Estacionario (SSGA)

- **Estrategia:** Los hijos reemplazan a los miembros de la población actual **uno por uno**, tan pronto como son creados.
- **Resultado:** Hay una gran superposición entre generaciones, ya que la población evoluciona de forma incremental.

Algoritmos Genéticos 'Messy' (mGA)

El Problema con los AG Estándar

Los AG estándar operan sobre cromosomas de longitud fija, lo que puede dificultar la preservación y combinación de 'bloques de construcción' (grupos de genes) óptimos. El cruce puede romper combinaciones buenas.

La Solución 'Messy'

El AG 'messy' (desordenado) encuentra soluciones evolucionando explícitamente estos **bloques de construcción** y luego combinándolos.

- Los individuos tienen **longitud variable** y se especifican como una lista de pares (posición, valor).
- Ejemplo: `((1,0), (3,1))` representa un individuo parcial donde el gen 1 tiene valor 0 y el gen 3 tiene valor 1.

Fases del Algoritmo 'Messy'

Fases del mGA

Un mGA se implementa en dos fases principales que se repiten:

1. **Fase Primordial:** Se aplica únicamente selección para enriquecer la población con buenos 'bloques de construcción' cortos, sin usar cruce ni mutación.

Fases del Algoritmo 'Messy'

Fases del mGA

Un mGA se implementa en dos fases principales que se repiten:

1. **Fase Primordial:** Se aplica únicamente selección para enriquecer la población con buenos 'bloques de construcción' cortos, sin usar cruce ni mutación.
2. **Fase Yuxtaposicional:** Se aplican operadores especiales de **corte** (para dividir) y **empalme** (para unir) para combinar los bloques de construcción y formar soluciones más completas.

Cuando el Humano es la Función de Aptitud

El Problema: La Aptitud Subjetiva

En los AG estándar, la selección se basa en una función de aptitud matemática. Pero, ¿cómo se define matemáticamente la 'calidad' de una obra de arte, una pieza musical o una animación?

La Solución: Evolución Interactiva

La Evolución Interactiva (EI) involucra a un **usuario humano** directamente en el proceso evolutivo.

- En cada generación, el sistema presenta al usuario un conjunto de soluciones candidatas (e.g., imágenes).
- El usuario **selecciona interactivamente** aquellas que considera 'mejores' o más estéticas.
- Las soluciones seleccionadas se convierten en los padres de la siguiente generación. El humano actúa como la función de aptitud.