Tarea 12

Programa. Univariate Marginal Distribution Algorithm

0.1 Metodología

Esta técnica pertenece a una clase de algoritmos evoultivos denominados EDA (Estimation of Distribution Algorithms) en los cuales se guía la búsqueda de una solución óptima al problema construyendo modelos probabilísticos y sacando muestras de soluciones candidatas prometedoras. Esto se logra midiendo la frecuencia de cada gen en la población y usando las probabilidades para influenciar la selección probabilística de genes en la construcción de nuevos individuos.

La estrategia de este algoritmo es usar la frecuencia de los genes del cromosoma de cada individuo o solución candidata en una población para la construcción de nuevos individuos.

Para este algoritmo en específico los genes de cada individuo están representados por dos valores, 0 o 1, por lo que el cromosoma C_i del individuo i-ésimo estará representado por una cadena de n bits. En la primera población que se genera, al no tener una distribución calculada, se usa una distribución uniforme para generar cada gen o bit g_j del cromosoma de cada individuo. Es decir, $P(g_j = 1) = 0.5$. A partir de la segunda población, cada $P(g_j = 1)$ con $j = 1, 2, ..., n$ se irá actualizando, por lo que ahora se tendrá una distribución multinomial.

La forma en que se seleccionarán a los mejores individuos es mediante Ranking, es decir, se orden de acuerdo al valor de fitness que tenga cada individuo, de mayor a menor, o de menor a mayor dependiento si se quiere encontrar un mínimo o un máximo óptimo. En este caso se busca encontrar el mínimo de una función $f(x_1, x_2)$, por lo que el valor de fitness de cada individuo será $f(x_1, x_2)$ mismo. Donde los valores de x_1 y x_2 son el fenotipo del individuo codificado por su representación en cadena binaria $s1$, $s2$ concatenados, formando así el genotipo o cromosoma $C_i = s1s2$.

A continuación se muestra el presudocódigo del algoritmo implementado.

Figure 1: Pseudocódigo de algoritmo UMD

```
Input: Bits_num, Population_size, Selection_size
Output: S_best
Population ← InitializePopulation(Bits_num, Population_size)
EvaluatePopulation(Population)
S_best ← GetBestSolution(Population)
While (¬StopCondition())
   Selected ← SelectFitSolutions(Population, Selection_size)
   V ← CalculateFrequencyOfComponents(Selected)
   Offspring ← ∅
   For (i To Population_size)
      Offspring ← ProbabilisticallyConstructSolution(V)
   End
   EvaluatePopulation(Offspring)
   S_best ← GetBestSolution(Offspring)
   Population ← Offspring
End
Return (S_best)
```

0.2 Implementación

Para la implementación de este algoritmo se consideró la posibilidad de extender o modificar su funcionalidad a otros algoritmos genéticos, por lo que se trató de separar y aislar en clases distintas de manera que no se tengan que hacer muchas modificaciones posteriores. A continuación se describen las clases implementadas:

1. Clase Organism: Representa a cada individuo o solución candidata.
2. Clase Population: Representa a la población de individuos
3. Clase GeneticOps: Contiene las operaciones a nivel individuo, es decir, selección, cruza y si se llegara a requerir, la operación de mutación.
4. Clase Selection: Contiene los distintos algoritmos de selección. Por el momento sólo se implementa Ranking.
5. Clase Environment: Contiene la forma en que se evaluará el fitness de cada individuo, de acuerdo al significado que tenga su cromosoma. En este caso también se definen las funciones en las que se quiere encontrar el mínimo.
6. Clase GA: Sirve para contener todos los parámetros del algoritmo y ejecutar todas las operaciones de acuerdo al pseudocódigo mostrado en la metodología. También se encarga de guardar la solución que se encuentra al terminar el algoritmo.

La clase GA nos permite abstraer los detalles de cómo está implementado un individuo o de qué algoritmo de selección o cruza se usó. Mientras que la clase Environment permite definir

cómo se evaluará el fitness de cada individuo sin la necesidad de modificar alguna otra clase. Las clases Selection y GeneticOps permiten extender el algoritmo a otros métodos de manera más sencilla.

Para inicializar el algoritmo se requiere de los siguientes parámetros:

- pop_size: Tamaño de la población.
- num_bits: Número de bits del cromosoma (mitad para x1 y mitad para x2).
- func: Índice de función a optimizar (valor 1 o 2).
- left: Límite inferior de los valores que pueden tomar x1 y x2.
- right: Límite superior de los valores que pueden tomar x1 y x2.
- iters: Número máximo de iteraciones.
- percentage: Porcentaje de individuos a seleccionar en cada iteración durante el Ranking.
- epsilon: Condición de paro.

Para la condición de paro del algoritmo, se propuso obtener una medida de 'varianza' del valor de fitness de los individuos. Donde el valor de fitness del indiviuo i-ésimo es $f_i(x_1, x_2)$, es decir, la evaluación de la función con el fenotipo del individuo. Posteriormente se calcula la media de los $f_i(x_1, x_2)$, esto es, $\mu = \frac{\sum_{i=1}^{m} f_i(x_1,x_2)}{m}$, para una población de m individuos. Finalmente se obtienen la varianza $\sigma = \frac{\sum_{i=1}^{m} (\mu - f_i(x_1,x_2))^2}{m-1}$. Esto bajo la intuición de que si todos los individuos tienen un fitness bastante similar será ya muy dificil que las nuevas generaciones sigan explorando el espacio en nuevas regiones.

0.3 Resultados

Para validar el algoritmo se usaron dos funciones. $f1(x_1, x_2) = {x_1}^2 + {x_2}^2 + 4$ la cual tiene un sólo mínimo en $x_1 = 0, x_2 = 0$ con valor $f(0, 0) = 4$. Y una función $f2(x_1, x_2) = cos(x_1) \times cos(x_2)$ la cual puede tener múltiples mínimos y máximos dependiendo de la región donde se defina, en este caso se búsca un óptimo en el rango $-4 \leq x_1 \leq 4$, $-4 \leq x_2 \leq 4$, donde existen 4 mínimos en $(x_1 = 0x_2 = -\pi)$, $(x_1 = 0, x_2 = \pi)$, $(x_1 = -\pi, x_2 = 0)$ y $(x_1 = \pi, x_2 = 0)$. A continuación se muestran las gráficas de dichas funciones:

Tarea 12

Figure 2: $f1(x_1, x_2) = {x_1}^2 + {x_2}^2 + 4$

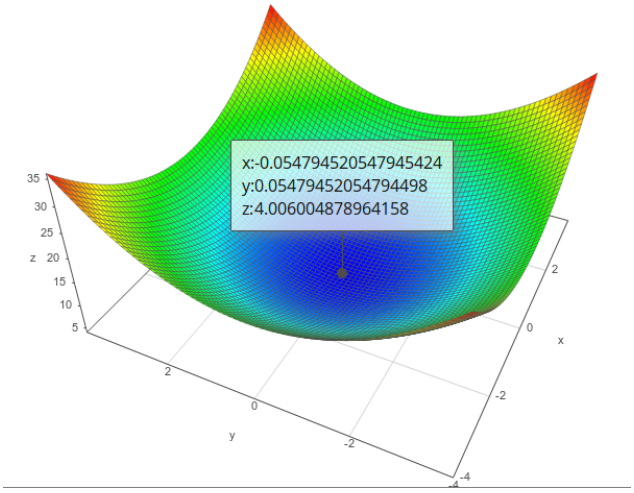
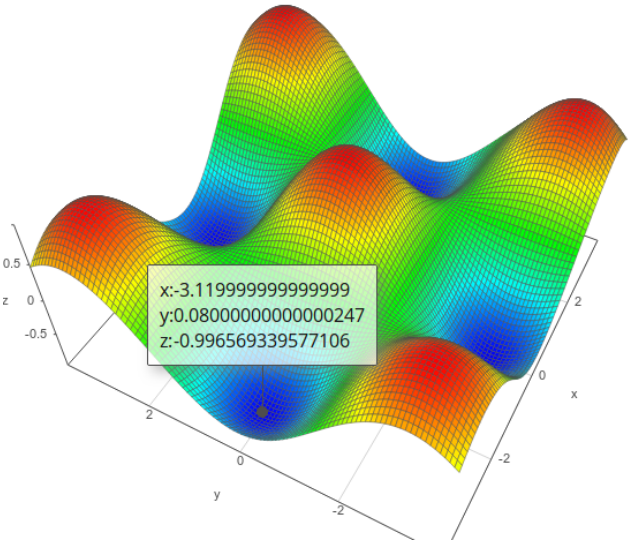


Figure 3: $f2(x_1, x_2) = cos(x_1) \times cos(x_2)$

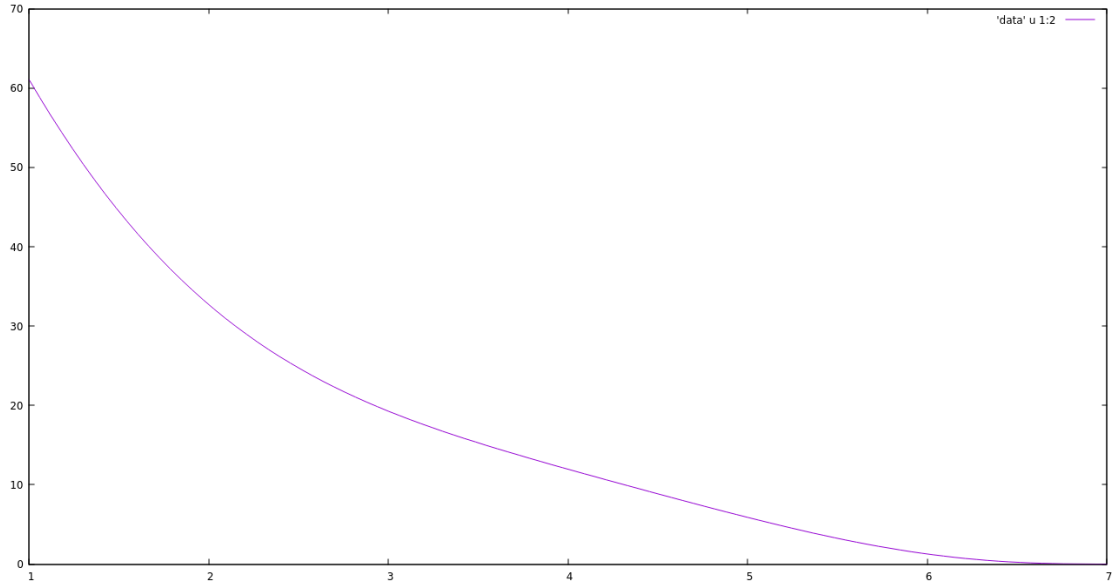


A continuación se muestra el resultado de la ejecución del algoritmo para la función $f1$ y la respectiva gráfica de convergencia, donde se muestra la varianza del valor fitness de cada generación hasta que se alcanza el valor deseado:

Figure 4: Punto x_1, x_2 mínimo encontrado por UMDA para función $f2$

```
isaac@irb ~/Documents/CIMAT/Programación y Algoritmos/Programaci-n-y-algoritmos/tarea 12 ? master ● g++ -std=c++11 Organism.cpp Popu
ronment.cpp GA.cpp test.cpp -o test && ./test 12 20 1 -4 4 50 0.3 0.000001

UMD Algorithm excecuted with following parameters:

* Population size: 12
* Chromosome size: 20
* Function option to minimize: 1
* Interval of search: [-4, 4]
* Selection method: Ranking
* Ranking percentage: 30%
* Max number of iterations: 50
* Stopping variance value: 1e-06

Best solution found: f(0.296875, -0.0234375) = 4.08868

Found after 5 iterations
```

Figure 5: Convergencia para $f1$, medida por la varianza del valor fitness de los individuos de cada generación

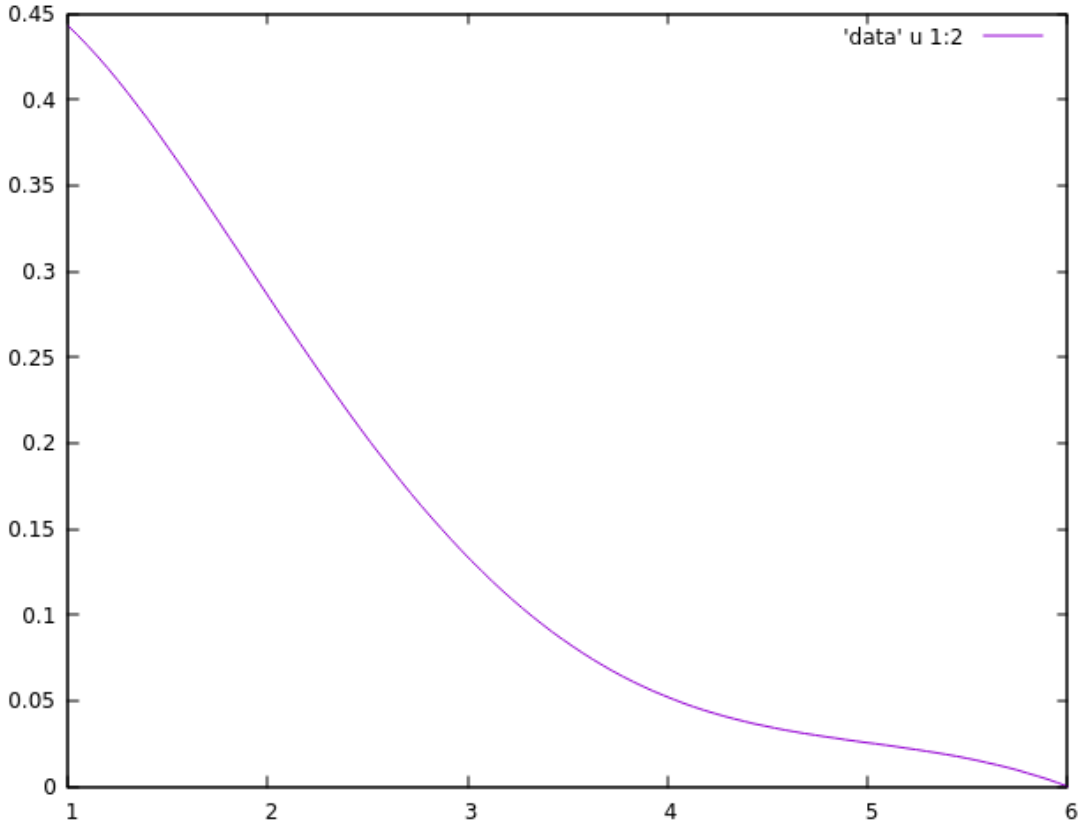


Finalmente, se muestra el resultado de la ejecución del algoritmo para la función $f2$ y la respectiva gráfica de convergencia, donde se muestra la varianza del valor fitness de cada generación hasta que se alcanza el valor deseado:

Figure 6: Punto x_1, x_2 mínimo encontrado por UMDA para función $f2$

```
isaac@irb ~/Documents/CIMAT/Programación y Algoritmos/Programaci-n-y-algoritmos/tarea 12 ? master ● g++ -std=c++11 Organi
ronment.cpp GA.cpp test.cpp -o test && ./test 12 20 2 -4 4 50 0.3 0.000001

UMD Algorithm excecuted with following parameters:

* Population size: 12
* Chromosome size: 20
* Function option to minimize: 2
* Interval of search: [-4, 4]
* Selection method: Ranking
* Ranking percentage: 30%
* Max number of iterations: 50
* Stopping variance value: 1e-06

Best solution found: f(3.08594, 0.125) = -0.990661

Found after 6 iterations
```

Figure 7: Convergencia para $f2$, medida por la varianza del valor fitness de los individuos de cada generación



0.4 Conclusión

La que la manera en la que funcionan los algoritmos genéticos es explorarando un espacio de posibles soluciones mediante operaciones de cruza y mutación, pero también explotando ciertas zonas del espacio mediante la selección de individuos que dan una mejor aproximación al óptimo, todo esto con muchos individuos al mismo tiempo.

Y apesar de que en este algoritmo no se implementó una técnica de mutación, para las funciones mostradas se pudo encontrar una solución o punto mínimo local. A pesar de esto, sería importante implementarlo ya que sin la mutación se corre el riesgo de que se llegue a una convergencia a un mínimo subóptimo de manera muy rápida sin llegar a explorar una mayor región del espacio antes.

Otra posible mejora es implementar más algoritmos de selección como lo son Ruleta o Torneo.

Finalmente, en cuando a la Programación Orientada a Objetos, se observó que separando las

diferentes funcionalidades del algoritmo en clases permite una mejor abstracción del problema, ya que, por ejemplo, la clase GA no requiere saber cómo se maneja una población ni como están implementados los individuos. Esto permite que el algoritmo sea más escalable y se puedan hacer extensiones sin tener que reescribir el código. Una mejora en este aspecto que se podría implementar es aplicar el concepto de herencia y polimorfismo, lo cual también ayudaría a reutilizar código que ya tiene una clase y sólo agregar las nuevas funcionalidades de la clase hija que la diferencian de la clase padre.