



INSTITUTO POLITÉCNICO NACIONAL
ESCUELA SUPERIOR DE CÓMPUTO
EVOLUTIONARY COMPUTING



PRÁCTICA 5: ALGORITMOS GENÉTICOS PARA PROBLEMAS COMBINATORIOS

ALUMNO: ORTEGA VICTORIANO IVAN

PROF.: JORGE LUIS ROSAS TRIGUEROS

FECHA DE REALIZACIÓN DE LA PRÁCTICA: 06/03/2018

FECHA DE ENTREGA DEL REPORTE: 14/03/2018

MARCO TEÓRICO.

Los algoritmos genéticos (denominados originalmente “planes reproductivos genéticos”) fueron desarrollados por John H. Holland a principios de los 1960s, motivado por resolver problemas de aprendizaje automático. [1]

El algoritmo genético enfatiza la importancia de la cruza sexual (operador principal) sobre el de la mutación (operador secundario), y usa selección probabilística. El algoritmo básico es el siguiente: [1]

1. Generar (aleatoriamente) una población inicial.
2. Calcular aptitud de cada individuo.
3. Seleccionar (probabilísticamente) con base a su aptitud.
4. Aplicar operadores genéticos (cruza y mutación) para generar la siguiente población.
5. Ciclar hasta que cierta condición se satisfaga.

La representación tradicional es la binaria, tal y como se ejemplifica en la figura 1:

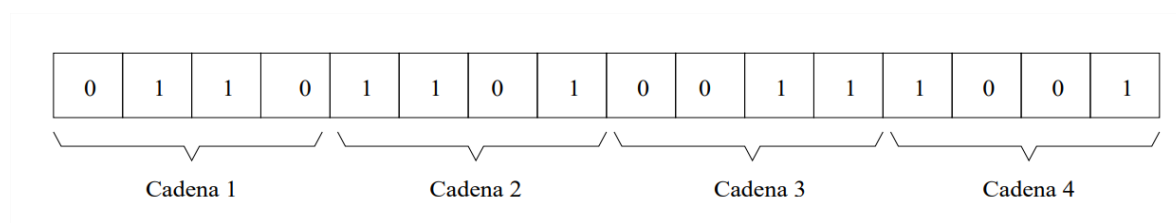


Figura 1. Ejemplo de la codificación (mediante cadenas binarias) usada tradicionalmente en los algoritmos genéticos. [1]

A la cadena binaria se le llama “cromosoma”. A cada posición de la cadena se le denomina “gen” y al valor dentro de esta posición se le llama “alelo”. Para

poder aplicar el algoritmo genético se requiere de los 5 componentes básicos siguientes: [1]

1. Una representación de las soluciones potenciales del problema.
2. Una forma de crear una población inicial de posibles soluciones (normalmente un proceso aleatorio).
3. Una función de evaluación que juegue el papel del ambiente, clasificando las soluciones en términos de su “aptitud”.
4. Operadores genéticos que alteren la composición de los hijos que se producirán para las siguientes generaciones.
5. Valores para los diferentes parámetros que utiliza el algoritmo genético (tamaño de la población, probabilidad de cruce, probabilidad de mutación, número máximo de generaciones, etc.)

Algoritmos genéticos vs otras técnicas evolutivas.

El AG usa selección probabilística al igual que la Programación Evolutiva, y en contraposición a la selección determinística de las Estrategias Evolutivas. El AG usa representación binaria para codificar las soluciones a un problema, por lo cual se evoluciona el genotipo y no el fenotipo como en la Programación Evolutiva o las Estrategias Evolutivas. El operador principal en el AG es la cruce, y la mutación es un operador secundario. En la Programación Evolutiva, no hay cruce y en las Estrategias Evolutivas es un operador secundario. Ha sido demostrado que el AG requiere de elitismo (o sea, retener intacto al mejor individuo de cada generación) para poder converger al óptimo. [1]

MATERIAL Y EQUIPO.

- Equipo de Cómputo
- Python3

DESARROLLO DE LA PRÁCTICA.

Se pidió implementar la solución para el “0/1 *Knapsack Problem*” y el “*Minimum Change Problem*” utilizando algoritmos genéticos.

0/1 KNAPSACK PROBLEM

Empezando por el “0/1 *Knapsack Problem*”, se propuso representar a los individuos como cadenas binarias de tamaño igual al número de ítems o elementos que podían estar o no en la mochila. Esto tiene mucho sentido, ya que un 0 significa que el elemento en la *i*-ésima posición del arreglo (lista), no entra en la mochila, 1 en caso contrario.

Teniendo la representación anterior es posible generar la población inicial como cadenas binarias aleatorias.

En cuanto a la función de aptitud, probé con tres distintas formas que me dieron resultados favorables.

La primera fue la propuesta en clase que se calculaba de acuerdo con la siguiente fórmula: $aptitud = V - \gamma(W - C)$ si y solo si $W > C$; o $aptitud = V$ si y solo si $W \leq C$, donde:

- V es el valor total acumulado con los ítems propuestos
- W es el peso total de los ítems propuestos
- γ es un factor de penalización
- C es la capacidad de la mochila.

Para que esta funcionara adecuadamente en los casos en que $W > C$, tenía que poner un factor de penalización muy alto (que, en principio, debería ser bajo), debido a que independientemente de los valores de W y C , si la penalización era muy baja, su valor de aptitud no sería el indicado para el individuo en ciertos casos. Supongamos que tenemos una mochila de capacidad $C=10$, los pesos de los ítems $w=[2,3,4,4,5]$, y sus valores $v=[3,3,3,5,6]$, además supongamos que se generó el individuo $[1,1,1,1,1]$, si en la función de aptitud propuesta dejamos un factor de penalización bajo $\gamma = 0.1$, la aptitud de dicho individuo estaría dada por:

$$aptitud = 20 - 0.1(18 - 10) = 20 - 0.1(8) = 20 - 0.8 = 19.2$$

Lo cual, comparado a la solución óptima de este problema que es el individuo $[1,1,0,0,1]$ cuya aptitud es de valor 12, no tiene sentido, pues el individuo planteado al inicio no se acerca siquiera a la solución. Por ello para este caso, recurrí a utilizar un valor de penalización un poco alto, lo cual implicaba en ciertos casos, tener valores negativos de aptitud.

La otra posible solución menos estricta y que me ofreció mejores resultados a la hora de ejecutar la implementación del algoritmo, la cuál fue la operación módulo. De igual forma, esta alternativa solo es para el caso en el que $W > C$, pues aplicando la operación módulo aquellos individuos que al evaluar su peso este fuese mayor a C , independientemente de que tan grande fuera, su aptitud siempre sería menor a la de la solución óptima, si bien no es lo más adecuado quizás, me dio mejores resultados que la función propuesta en clase.

Sin embargo, la que me dio mejores resultados fue basarme únicamente en la aptitud como el valor V obtenido, y en el caso de que el peso sumado de los ítems W fuese mayor que la capacidad de la mochila, iteraba sobre el cromosoma cambiando alguno de los unos por 0 hasta que el peso fuera menor o igual a la capacidad de la mochila, de esta forma eliminaba todos esos casos que no fueran muy aptos, y en caso de W fuera menor que C , la aptitud era igual a V , de tal manera que la función de aptitud se define como sigue:

Si $W > C$ entonces:

Mientras $W > C$ hacer:

Para i desde 0 hasta longitud_cromosoma hacer:

Si cromosoma[i] == 1 hacer

cromosoma[i] ^= 1

Romper ciclo

$W = \text{obtener_peso}(\text{cromosoma})$ // Calculamos su peso

$V = \text{obtener_valor}(\text{cromosoma})$ // Obtenemos el valor

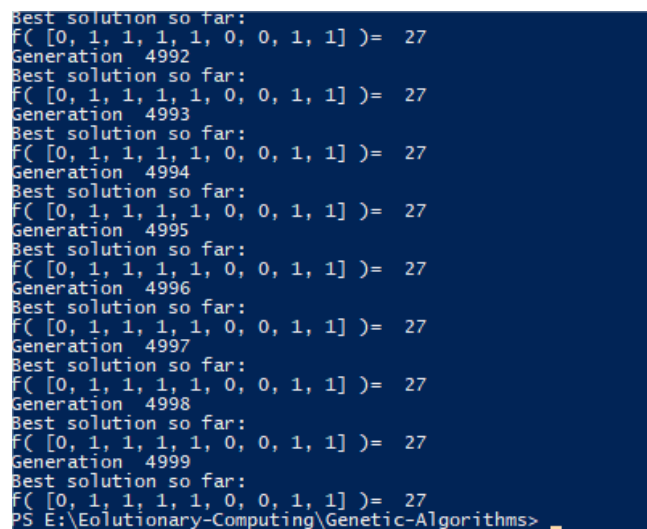
Aptitud = V

En otro caso:

Aptitud = V

El resto del proceso se realiza de igual forma que en el primer ejemplo que vimos, únicamente cambiamos los valores retornados por la función de comparación de cromosomas, para cambiar el orden de los elementos de la población, pues el ejemplo al ser de minimizar siempre nos dejaría al individuo con menor aptitud como el mejor individuo. Otro aspecto que no se utilizó fue la decodificación del cromosoma. Y en cuanto a la mutación dado que son cadenas binarias, se optó por la inversión de bits utilizando la operación xor.

Ya implementado nuestro programa se procedió a ejecutar el correspondiente script de Python obteniendo los siguientes resultados que se muestran de la figura 2 a la 5, utilizando como valores $C=20$, $w=[2,3,4,4,5,10,12,1,3]$, y $v=[3,3,3,5,6,7,7,5,5]$ y una condición de finalización de 2000 iteraciones del algoritmo, donde existen tres soluciones óptimas $[0,1,1,1,1,0,0,1,1]$, $[1,1,0,1,1,0,0,1,1]$ y $[1,0,1,1,1,0,0,1,1]$ con un valor $V=27$:



```
Best solution so far:
F( [0, 1, 1, 1, 1, 0, 0, 1, 1] )= 27
Generation 4992
Best solution so far:
F( [0, 1, 1, 1, 1, 0, 0, 1, 1] )= 27
Generation 4993
Best solution so far:
F( [0, 1, 1, 1, 1, 0, 0, 1, 1] )= 27
Generation 4994
Best solution so far:
F( [0, 1, 1, 1, 1, 0, 0, 1, 1] )= 27
Generation 4995
Best solution so far:
F( [0, 1, 1, 1, 1, 0, 0, 1, 1] )= 27
Generation 4996
Best solution so far:
F( [0, 1, 1, 1, 1, 0, 0, 1, 1] )= 27
Generation 4997
Best solution so far:
F( [0, 1, 1, 1, 1, 0, 0, 1, 1] )= 27
Generation 4998
Best solution so far:
F( [0, 1, 1, 1, 1, 0, 0, 1, 1] )= 27
Generation 4999
Best solution so far:
F( [0, 1, 1, 1, 1, 0, 0, 1, 1] )= 27
PS E:\Evolutionary-Computing\Genetic-Algorithms>
```

Figura 2. El algoritmo converge a la primera solución óptima.

```

f( [1, 0, 1, 1, 1, 0, 0, 1, 1] )= 27
Generation 4992
Best solution so far:
f( [1, 0, 1, 1, 1, 0, 0, 1, 1] )= 27
Generation 4993
Best solution so far:
f( [1, 0, 1, 1, 1, 0, 0, 1, 1] )= 27
Generation 4994
Best solution so far:
f( [1, 0, 1, 1, 1, 0, 0, 1, 1] )= 27
Generation 4995
Best solution so far:
f( [1, 0, 1, 1, 1, 0, 0, 1, 1] )= 27
Generation 4996
Best solution so far:
f( [1, 0, 1, 1, 1, 0, 0, 1, 1] )= 27
Generation 4997
Best solution so far:
f( [1, 0, 1, 1, 1, 0, 0, 1, 1] )= 27
Generation 4998
Best solution so far:
f( [1, 0, 1, 1, 1, 0, 0, 1, 1] )= 27
Generation 4999
Best solution so far:
f( [1, 0, 1, 1, 1, 0, 0, 1, 1] )= 27

```

Figura 3. El algoritmo converge a la segunda solución óptima.

```

f( [1, 1, 1, 1, 1, 0, 0, 1, 0] )= 25
Generation 75
Best solution so far:
f( [1, 1, 1, 1, 1, 0, 0, 1, 0] )= 25
Generation 76
Best solution so far:
f( [0, 1, 1, 1, 1, 0, 0, 1, 1] )= 27
Generation 77
Best solution so far:
f( [0, 1, 1, 1, 1, 0, 0, 1, 1] )= 27
Generation 78
Best solution so far:
f( [0, 1, 1, 1, 1, 0, 0, 1, 1] )= 27
Generation 79
Best solution so far:
f( [0, 1, 1, 1, 1, 0, 0, 1, 1] )= 27
Generation 80

```

Figura 4. El algoritmo converge a la solución óptima desde la generación número 76.

```

Generation 129
Best solution so far:
f( [0, 0, 0, 0, 1, 1, 0, 1, 1] )= 23
Generation 130
Best solution so far:
f( [0, 0, 0, 0, 1, 1, 0, 1, 1] )= 23
Generation 131
Best solution so far:
f( [1, 1, 0, 1, 1, 0, 0, 1, 1] )= 27
Generation 132
Best solution so far:
f( [1, 1, 0, 1, 1, 0, 0, 1, 1] )= 27
Generation 133
Best solution so far:
f( [1, 1, 0, 1, 1, 0, 0, 1, 1] )= 27
Generation 134
Best solution so far:

```

Figura 5. El algoritmo converge a la tercera solución óptima desde la generación número 131.

Los resultados fueron favorables, y en problemas pequeños el algoritmo llegaba a converger desde muy pocas generaciones.

MINIMUM CHANGE PROBLEM.

Para resolver este problema fue un poco más difícil al anterior, pero no muy diferente, en el 0/1 KP se consideraban dos cosas: el valor total y el peso total. En este caso es semejante, pues hay que considerar el número de monedas utilizadas y el valor total de la suma de estas. Sin embargo, lo que consideré cambiar fue que ahora los individuos en vez de ser representados mediante cadenas binarias fuesen representados por números naturales que corresponden al número de monedas utilizadas. Si tuviéramos una lista de denominaciones $d = [1, 2, 3]$, el individuo $[0, 2, 3]$ indicaría que se están usando 2 monedas de denominación 2, 3 monedas de denominación 3 y ninguna de denominación 1. De tal forma, que opté por generar a los cromosomas de forma aleatoria con números entre $(0, N)$, donde N es el valor en cambio requerido.

Considerando estos cambios, primero expliquemos la función de aptitud, consideremos las siguientes variables:

- N = valor requerido de cambio
- C = número de monedas utilizadas
- V = El valor total de la suma de las monedas utilizadas
- γ = factor de penalización

Ahora bien, si V es igual N , entonces significa que logramos encontrar una solución en la que daremos el cambio adecuado, sin embargo, dado que pudiera ser el caso de que existan distintas posibilidades para ello, le daremos un factor de penalización de acuerdo con el número de monedas utilizadas, para este caso se propone que γ sea un valor muy bajo (0.01, por ejemplo). Por otro lado, si V es mayor entonces se aplicará la misma medida que en KP, eliminando elementos, en este caso monedas, de tal forma que el valor sumado sea menor o igual al valor N requerido, y así finalmente aplicar la misma regla que cuando se daba el caso que V fuera igual a N . Por otro lado, si $V < N$, aplica lo mismo que si V fuera igual a N .

De tal manera que la función de aptitud se define como sigue:

```
Gamma = 0.01
```

```
Si  $V > N$  entonces:
```

```
    Mientras  $V > N$  hacer:
```

```
        Para  $i$  desde 0 hasta longitud_cromosoma hacer:
```

```
            Si cromosoma[i]  $\geq 1$  hacer
```

```
                cromosoma[i] -= 1
```

```
            Romper ciclo
```

```
     $V = \text{obtener\_valor}(\text{cromosoma})$  // Calculamos su valor
```

```
Número_de_monedas = obtener_monedas(cromosoma)
```

```
Aptitud = V - Gamma * Número_de_monedas
```

En otro caso:

```
Aptitud = V - Gamma * Número_de_monedas
```

En cuanto a la mutación, fue la parte que consideré más difícil de elegir. Sin embargo, opté por lo sencillo realizando sumas o restas, lo interesante fue que dicha mutación es que cuando se realiza, añadí un nuevo factor de probabilidad, que es la probabilidad de que un valor del *offspring* se le sume 1, si un número aleatorio es menor o igual a la probabilidad, de lo contrario se le resta 1. Para fines prácticos lo dejé en un 50 a 50, es decir, si un número real aleatorio entre 0 y 1 es mayor a 0.5 se realiza una resta entre el valor actual del *offspring* en algún índice aleatorio y 1, es decir: $o[i] = o[i] - 1$, en caso contrario se realiza una suma con uno, es decir: $o[i] = o[i] + 1$.

Puede sonar un poco extraño, sin embargo, los resultados que obtuve fueron satisfactorios.

Definido todo, en las figuras 6 a la 9 se muestran algunas capturas de los resultados obtenidos por la implementación del algoritmo en un script de Python con denominaciones $d = [1,3,5,6,8,7]$ y $N = 129$ donde la solución óptima es utilizar 17 monedas (según la solución por Programación Dinámica) siendo una de las soluciones el individuo $[0,0,1,2,14,0]$, con un número de iteraciones máximas $itmax=5000$.

```
Best solution so far:
f([0, 1, 0, 0, 14, 2]) = 129
Generation 4996
Best solution so far:
f([0, 1, 0, 0, 14, 2]) = 129
Generation 4997
Best solution so far:
f([0, 1, 0, 0, 14, 2]) = 129
Generation 4998
Best solution so far:
f([0, 1, 0, 0, 14, 2]) = 129
Generation 4999
Best solution so far:
f([0, 1, 0, 0, 14, 2]) = 129
PS E:\Evolutionary-Computing\Genetic-Algorithms>
```

Figura 6. El algoritmo converge a otra solución óptima donde de igual manera se utilizan 17 monedas.

```
Generation 4995
Best solution so far:
f([0, 0, 0, 0, 10, 7]) = 129
Generation 4996
Best solution so far:
f([0, 0, 0, 0, 10, 7]) = 129
Generation 4997
Best solution so far:
f([0, 0, 0, 0, 10, 7]) = 129
Generation 4998
Best solution so far:
f([0, 0, 0, 0, 10, 7]) = 129
Generation 4999
Best solution so far:
f([0, 0, 0, 0, 10, 7]) = 129
PS E:\Evolutionary-Computing\Genetic-Algorithms>
```

Figura 7. El algoritmo converge a otra solución óptima donde de igual manera se utilizan 17 monedas.

```

Best solution so far:
f( [0, 0, 0, 1, 11, 5] )= 129
Generation 4997
Best solution so far:
f( [0, 0, 0, 1, 11, 5] )= 129
Generation 4998
Best solution so far:
f( [0, 0, 0, 1, 11, 5] )= 129
Generation 4999
Best solution so far:
f( [0, 0, 0, 1, 11, 5] )= 129
PS E:\Evolutionary-Computing\Genetic-Algorithms>

```

Figura 8. El algoritmo converge a otra solución óptima donde de igual manera se utilizan 17 monedas.

Como vemos, debido al factor aleatorio, la solución por algoritmos genéticos nos puede ofrecer mayor cantidad de soluciones, debido a que el algoritmo que implementé en programación dinámica para reconstruir la solución es muy sistemático, y siempre va a llegar a una de las soluciones cuando existen una gran variedad en muchos casos.

CONCLUSIONES.

Esta práctica me fue muy interesante desde el aspecto de como la probabilidad y algunas condiciones aleatorias pueden ayudar a mejorar el rendimiento de algunos algoritmos, y más aún cuando el tamaño de problema es muy grande, principalmente en problemas donde sus soluciones son un conjunto de combinaciones. Hoy en día existen una gran variedad de heurísticas que ayudan a aproximar soluciones a problemas muy complejos como es el caso de los problemas NP.

El resolver este tipo de problemas mediante algoritmos basados en comportamientos, sistemas o individuos que podemos encontrar en la naturaleza, es algo, desde mi punto de vista, increíble. Por muy simple que nosotros pensemos que es nuestro comportamiento o el de algunos otros individuos, este es solo la parte visible de una serie de actividades demasiado complejas de las que entrar a detalle conlleva a desarrollar actividades muy extensas.

En cuanto a las implementaciones de los algoritmos, creo que el más sencillo de implementar fue el algoritmo genético para el 0/1 KP, ya que solo eran cadenas binarias, mientras que en el problema de las monedas eran cadenas de números enteros. En el código tuve que hacer una modificación, ya que por alguna razón todos los individuos generados eran iguales en varias ocasiones, aun teniendo en cuenta que los números eran pseudo-aleatorios. Por lo que al crear la ruleta en la parte del cálculo de la variable *acc* tuve que poner una condición de que si la variable *maxv* fuera igual a *fitness_values[p]* a la variable *acc* se le sumara un valor muy pequeño cercano a 0, en mi caso lo dejé como 0.0001, para evitar problemas de división por 0 a la hora de ejecución.

En el problema de las monedas fueron más veces las que lo tuve que ejecutar para converger a una solución óptima, en cambio en el KP había ocasiones en

que se convergía a la solución desde un número pequeño de generaciones. Cabe destacar que el número de cromosomas utilizados fue de 10, probando con más cromosomas se convergía más rápido a la solución, sin embargo, era ineficiente debido a que estaríamos casi llegando a realizar una búsqueda exhaustiva de la solución.

Finalmente, en cuanto al “Jugs Problem”, se me hizo un tanto compleja la implementación, sin embargo, la idea que tuve para solucionarlo se basaba en lo que vimos en la última clase en el tema “Genetic Programming” donde los individuos los modelábamos mediante árboles sintácticos. Mi posible solución para el problema de 3 jarrones era representar a cada jarrón como una variable diferente, digamos x , y , z , de tal forma que por cada movimiento que hiciéramos, por ejemplo, pasar contenido del jarrón x al jarrón y se modelara mediante $x-c$, donde c es la cantidad del jarrón x que pasó al jarrón y , por otro lado y lo veríamos como $y+c$, pues se le agregó cierta cantidad, y así sucesivamente. Tal vez la idea no esté completa, por lo que decidí no implementarla.

Referencias

- [1] C. A. Coello Coello, «Departamento de Computación CINVESTAV,» Mayo 2017. [En línea].
Available: <http://delta.cs.cinvestav.mx/~ccoello/compevol/apuntes.pdf>.