Reporte 10:
Algoritmo Genético

Jorge Torres

1 de mayo de 2022

1. Objetivo

El problema de la mochila (inglés: knapsack) es un problema clásico de optimización, particularmente de programación entera, donde la tarea consiste en seleccionar un subconjunto de objetos de tal forma que (i) no se exceda la capacidad de la mochila en términos de la suma de los pesos de los objetos incluidos, y que (ii) el valor total de los objetos incluidos sea lo máximo posible. Este problema es pseudo-polinomial, ya que existe un algoritmo de tabulación que determina la combinación óptima.

Un algoritmo genético representa posibles soluciones a un problema en términos de un genoma, que en este caso va a ser un vector de verdades y falsos, indicando cuáles objetos se van a incluir en la mochila (`TRUE` o 1 significa que llevamos el objeto, `FALSE` o 0 que lo descartamos de la selección).

El objetivo de esta actividad consiste en generar tres instancias con tres distintas reglas:

- Instancia 1: el peso y el valor de cada objeto se generan independientemente con una distribución uniforme.
- Instancia 2: el valor de cada objeto se genera independientemente con una distribución exponencial y su peso es inversamente correlacionado con el valor.
- Instancia 3: el peso de cada objeto se genera independientemente con una distribución normal y su valor es (positivamente) correlacionado con el cuadrado del peso, con un ruido normalmente distribuido de baja magnitud.

2. Desarrollo

El desarrollo de la actividad está basado en el código implementado por E. Schaeffer, en el que primero calcula el óptimo teórico del problema de la mochila y después lo compara con los resultados del algoritmo genético [1]. La función `knapsack()` del código 1 calcula el óptimo teórico del problema pseudo-polinomial. El desarrollo completo puede encontrarse en el repositorio en GitHub de J. Torres [2].

Código 1: Solución Óptima

```
def knapsack(peso_permitido, pesos, valores):
    assert len(pesos) == len(valores)
    peso_total = sum(pesos)
    valor_total = sum(valores)
    if peso_total < peso_permitido:
        return valor_total
    else:
        V = dict()
        for w in range(peso_permitido + 1):
            V[(w, 0)] = 0
        for i in range(len(pesos)):
            peso = pesos[i]
            valor = valores[i]
            for w in range(peso_permitido + 1):
                cand = V.get((w - peso, i), -float('inf')) + valor
                V[(w, i + 1)] = max(V[(w, i)], cand)
        return max(V.values())
```

Las funciones `factible()` y `objetivo()` del código 2 determinan si el peso de los objetos seleccionados es menor al límite y el valor total de dichos objetos, respectivamente.

Código 2: Peso y Valor Totales de la Selección

```
def factible(seleccion, pesos, capacidad):
    return np.inner(seleccion, pesos) <= capacidad

def objetivo(seleccion, valores):
    return np.inner(seleccion, valores)
```

En el código 3 se tienen cuatro funciones. La función `normalizar()` arregla una distribución entre 0 y 1 de un conjunto de datos, que se utiliza para obtener valores aleatorios con esta distribución en subsecuentes funciones. La función `poblacion_inicial()` crea un arreglo bidimensional de n columnas (objetos) y tam filas (individuos) con n valores 0 ó 1 aleatoria e uniformemente distribuídos para cada individuo. La función `mutacion()` selecciona objetos al azar de los individuos y, si el valor actual es 0, lo cambia a 1, creando un individuo diferente. La función `reproduccion()` toma dos individuos, selecciona una posición al azar en sus vectores de objetos, y la utiliza como posición de corte. Después realiza un cruzamiento entre los vectores cortados, creando un nuevo individuo que tiene información de los individuos originales.

Código 3: Funciones Normalizar, Población Inicial, Mutación y Reproducción

```
def normalizar(data):
    menor = min(data)
    mayor = max(data)
    rango  = mayor - menor
    data = data - menor # > 0
    return data / rango # entre 0 y 1

def poblacion_inicial(n, tam):
    pobl = np.zeros((tam, n))
    for i in range(tam):
        pobl[i] = (np.round(np.random.uniform(size = n))).astype(int)
    return pobl

def mutacion(sol, n):
    pos = randint(0, n - 1)
    mut = np.copy(sol)
    mut[pos] = 1 if sol[pos] == 0 else 0
    return mut

def reproduccion(x, y, n):
    pos = randint(2, n - 2)
    xy = np.concatenate([x[:pos], y[pos:]])
    yx = np.concatenate([y[:pos], x[pos:]])
    return (xy, yx)
```

Para realizar observaciones basadas en las instancias mencionadas en la sección 1, se incluyen un total de 6 generadores de pesos y valores aleatorios, cuyos códigos se discuten en las siguientes secciones.

2.1. Generadores - Instancia 1

En el código 4, con las funciónes `generador_pesos()` y `generador_valores()` se generan independientemente una cantidad n de pesos y valores de manera aleatoria y uniformemente distribuídos.

Código 4: Generadores de Pesos y Valores de Instancia 1

```
def generador_pesos(cuantos, low, high):
    return np.round(normalizar(np.random.uniform(low=low, high=high, size = cuantos)) * (high -
    low) + low)

def generador_valores(pesos, low, high):
    return np.round(normalizar(np.random.uniform(low = low, high = high, size = pesos)) * (high -
    low) + low)
```

2.2. Generadores - Instancia 2

La función `generador_valores2()` del código 5 crea aleatoriamente una cantidad n de valores con una distribución exponencial, mientras que la función `generador_pesos2()` obtiene la misma cantidad de pesos cuyo valor numérico está correlacionado con el inverso de los valores.

Código 5: Generadores de Pesos y Valores de Instancia 2

```
def generador_pesos2(valores, low, high):
    cant = 1 / valores
    return np.round(((normalizar(cant))) * (high - low) + low)

def generador_valores2(pesos, low, high):
    cant = np.arange(0, pesos)
    return np.round(normalizar(expon.pdf(cant))  * (high - low) + low)
```

2.3. Generadores - Instancia 3

En la tercera instancia, se necesitan obtener pesos independientemente con una distribución normal, mietras que los valores están positivamente correlacionados con el cuadrado de los pesos, lo cual se implementa con las funciones `generador_pesos3()` y `generador_valores3()` del código 6.

Código 6: Generadores de Pesos y Valores de Instancia 3

```
def generador_pesos3(cuantos, low, high):
    return np.round(normalizar(np.random.normal(size = cuantos)) * (high - low) + low)

def generador_valores3(pesos, low, high):
    return np.round((pesos**2) * (high - low) + low)
```

Para las tres instancias se crean $n = 100$ pesos y valores, el algoritmo se itera en 150 pasos y se realizan 20 repeticiones de cada instancia. Sin embargo, se hacen dos combinaciones distintas variando la probabilidad de mutación, la cantidad de reproducciones o cruzamientos, y la población inicial de individuos. La primera combinación consiste en una probabilidad de mutación $pm = 0{,}05$, una cantidad de cruzamientos $rep = 50$ y una población inicial $init = 100$. La segunda combinación consiste en $pm = 0{,}1$, $rep = 100$ e $init = 50$. Estos parámetros se pueden observar en el código 7.

Código 7: Parámetros Iniciales

```
n = 100
tmax = 150
iteraciones = 20

#####Combinacion 1#####

pm, rep, init = 0.05, 50, 100

#####Combinacion 2#####

pm, rep, init = 0.1, 100, 50
```

Al iniciar las iteraciones, se generan los pesos y valores, se define la capacidad máxima de la mochila, se calcula el valor óptimo con la función `knapsack()` y se crea la población inicial de individuos, como se observa en el código 8.

Código 8: Inicio de Iteraciones

```
for runs in range(iteraciones):
    pesos = generador_pesos(n, 15, 80)
    valores = generador_valores(n, 10, 500)
    capacidad = int(round(sum(pesos) * 0.65))
    optimo = knapsack(capacidad, pesos, valores)
    p = poblacion_inicial(n, init)
    tam = p.shape[0]
    assert tam == init
    mejor = None
    mejores = []
```

En el código 9, se inician las mutaciones y reproducciones de los individuos, creando nuevos individuos que se agregan a la lista total. De esta lista se toman los valores de factibilidad y objetivo obtenidos con las funciones del código 2 y se ordenan los individuos en orden descendente de factibilidad.

Código 9: Mutaciones, Reproducciones y Ordenamiento de Factibilidad

```
    for t in range(tmax):
        for i in range(tam): # mutarse con probabilidad pm
            if random() < pm:
                p = np.vstack([p, mutacion(p[i], n)])
        for i in range(rep):  # reproducciones
            padres = sample(range(tam), 2)
            hijos = reproduccion(p[padres[0]], p[padres[1]], n)
            p = np.vstack([p, hijos[0], hijos[1]])
        tam = p.shape[0]
        d = []
        for i in range(tam):
            d.append({'idx': i, 'obj': objetivo(p[i], valores),
                      'fact': factible(p[i], pesos, capacidad)})
        d = pd.DataFrame(d).sort_values(by = ['fact', 'obj'], ascending = False)
```

Por último, se toma una cantidad igual a la población inicial de los individuos más factibles y se elimina el resto. Para cada iteración, se agrega a una lista el mejor valor obtenido al final de la iteración y se calcula una proporción comparada con el valor óptimo de acuerdo a la ecuación 1 y se implementa en el código 10,

$$P = \frac{(O - M)}{O} \tag{1}$$

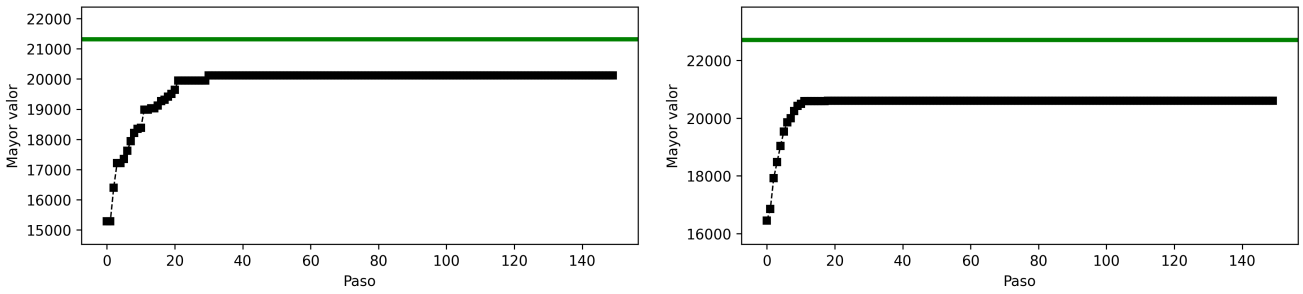
donde P es la proporción, O es el valor óptimo obtenido de la función `knapsack()` y M es el mejor valor obtenido al final de la iteración.

Código 10: Eliminación de Individuos Menos Factibles y Cálculo de Propirción

```
        mantener = np.array(d.idx[:init])
        p = p[mantener, :]
        tam = p.shape[0]
        assert tam == init
        factibles = d.loc[d.fact == True,]
        mejor = max(factibles.obj)
        mejores.append(mejor)
    resultados1.append((optimo - mejor) / optimo)
```

3. Resultados

En la figura 1 se puede ver una de las iteraciones del algoritmo genético para la primera instancia. La figura 1a muestra el desarrollo del algoritmo para la combinación 1, mientras que la figura 1b lo muestra para la combinación 2. Se puede observar cómo para la combinación 1 hay una mejora gradual de los valores hasta que alcanza un máximo y se estanca. Para la combinación 2, la mejora es un poco más rápida, pero la separación entre el valor óptimo y el valor donde se estanca es mayor.

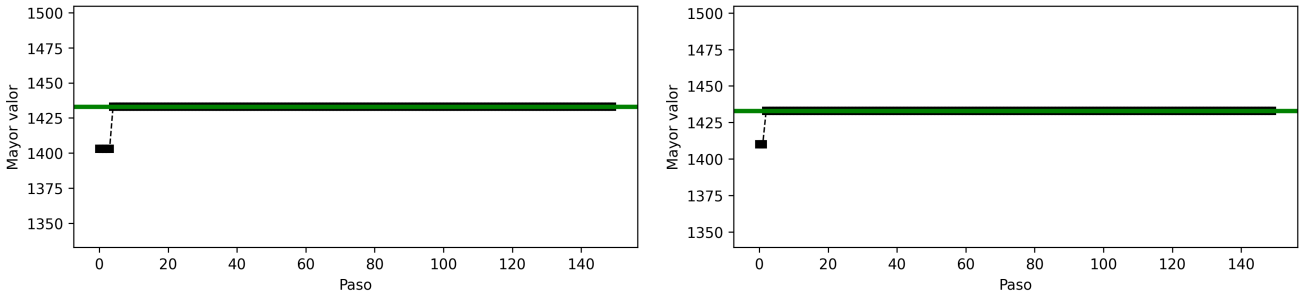


(a) Desarrollo del algoritmo para la combinación 1.

(b) Desarrollo del algoritmo para la combinación 2.

Figura 1: Ejemplo de iteración para la Instancia 1.

La figura 2 muestra una iteración de cada combinación para la segunda instancia. Para ambos casos se puede observa cómo no solamente existe una mejora casi inmediata, sino que el valor máximo logra alcanzar el valor óptimo y quedarse sobre él. Sin embargo, se puede observar que esta instancia trabaja con valores más bajos.

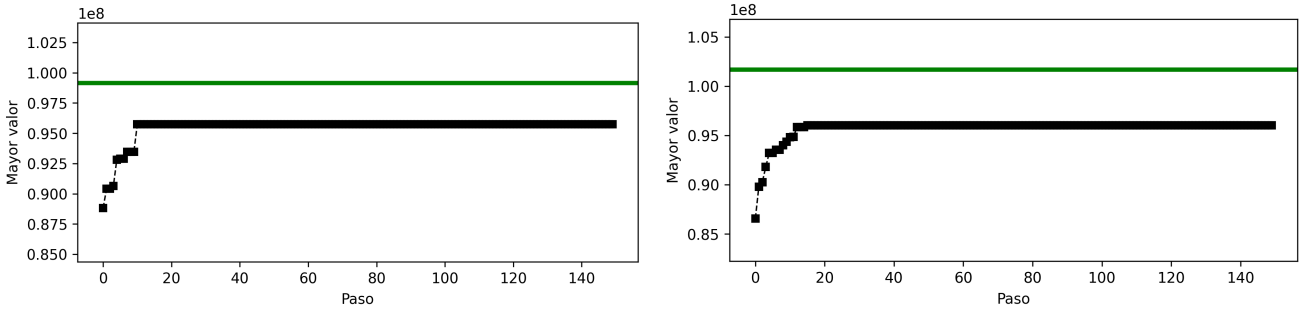


(a) Desarrollo del algoritmo para la combinación 1.

(b) Desarrollo del algoritmo para la combinación 2.

Figura 2: Ejemplo de iteración para la Instancia 2.

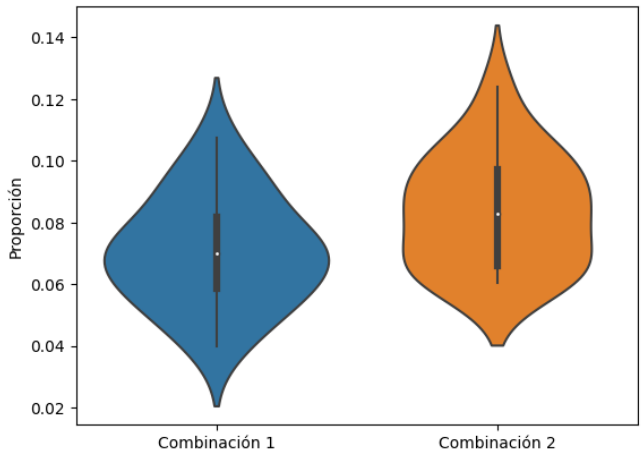
En la figura 3 se ven representados ejemplos de las dos combinaciones para la tercera instancia. Esta instancia puede trabajar con valores mucho mayores que las dos anteriores, en el rango de centenas de millones. Contrario al caso de la instancia 1, se puede ver cómo la primera combinación (figura 3a), presenta mejoras en ciertos intervalos y el valor máximo se aproxima más al óptimo, llegando también a estancarse casi al principio de la iteración. Por el otro lado, la combinación 2 (figura 3b), presenta un desarrollo más gradual y la diferencia entre el valor óptimo y el máximo es más grande.



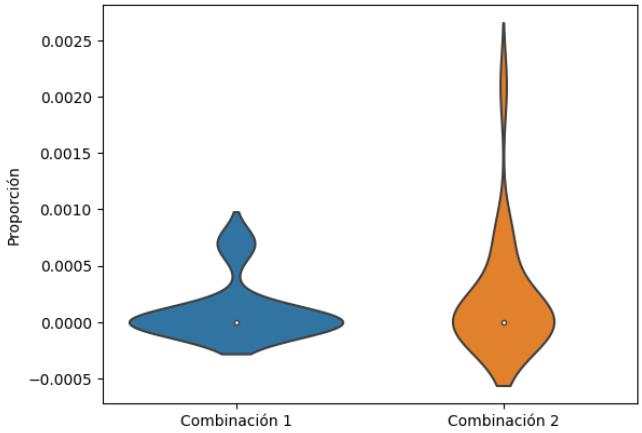
(a) Desarrollo del algoritmo para la combinación 1.

(b) Desarrollo del algoritmo para la combinación 2.

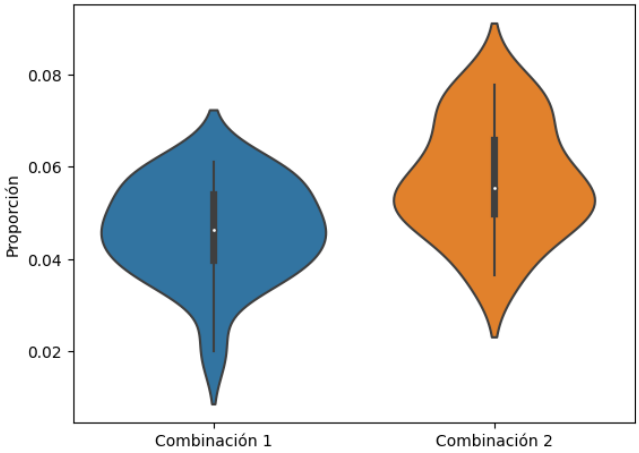
Figura 3: Ejemplo de iteración para la Instancia 3.



(a) Distribución de ambas combinaciones para la instancia 1.



(b) Distribución de ambas combinaciones para la instancia 2.



(c) Distribución de ambas combinaciones para la instancia 3.

Figura 4: Distribuciones de las mejores proporciones alcanzadas para cada instancia y cada combinación.

Para una mejor referencia visual, en la figura 4 se han representado las mejores proporciones logradas en las 20 iteraciones de cada instancia y cada combinación en diagramas tipo violín. De esta forma se puede observar la distribución que presentan, así como la media para cada combinación.

Se han realizado también análisis estadísticos de tipo ANOVA entre ambas combinaciones de cada instancia para dilucidar si existe una diferencia estadística al variar la probabilidad de mutación, la cantidad de cruzamientos y el tamaño de población inicial. Para la instancia 1, se ha concluido que sí existe una diferencia estadística, ya que el resultado del análisis arroja un valor p mucho menor a 0.05. El valor p encontrado para la instancia 2 es mayor a 0.05, por lo que se podría concluir que no existe diferencia estadísticamente significativa al variar los parámetros mencionados. En la instancia 3 también se encuentra que hay una diferencia estadística al obtener un valor p menor a 0.05.

4. Conclusiones

Este es un ejemplo bastante básico de la forma en que se puede implementar un algoritmo genético para hacer más eficiente la solución de problemas complejos.

De los análisis estadísticos se puede observar que, tanto para la instancia 1 como para la instancia 3, sí existe una diferencia en la calidad de las proporciones entre valor máximo y valor óptimo al variar los parámetros. Sin embargo, para la instancia 2 no existe tal diferencia en calidad. Esto podría deberse a la rapidez con la que el algoritmo llega al valor óptimo, lo cual a su vez podría deberse a la relación exponencial entre valores y pesos

cuando estos se generan.

Referencias

[1] E. Schaeffer. Geneticalgorithm, 2019. URL https://github.com/satuelisa/Simulation/blob/master/GeneticAlgorithm.

[2] J. Torres. P_10, 2022. URL https://github.com/FeroxDeitas/Simulacion-Nano/tree/main/Tareas/P_10.