Universidad del Istmo de Guatemala
Facultad de Ingenieria
Ing. en Sistemas
Informatica 2
Prof. Ernesto Rodriguez - erodriguez@unis.edu.gt

Repaso Parcial #2

Los *algoritmos geneticos* son algoritmos de busqueda que permiten encontrar soluciones a problemas que son dificiles de resolver desde un punto de vista practico. Estos algoritmos se inspiran en el mecanismo que utilizan los seres vivos para adaptarse a ambientes diferentes.

Interfaz IProblem

Los problemas de busqueda utilizando programación genetica seran representados por la interfaz generica `IProblem`$\langle$`T`$\rangle$. **Como primer ejercicio, debe implementar la interfaz `IProblem` la cual tendra los siguientes metodos:**

El primer metodo es el *operador de intercambio*. Dicho operador lo representaremos con metodo llamado `Crossover` de tipo "`Crossover : T[ ]`$\otimes$`T[ ]` $\rightarrow$ `T[ ]`". Este operador recive dos arreglos, los cuales son aproximaciones a una solución representadas como arreglos, y las intenta combinar para obtener una mejor solución.

El segundo elemento de un algoritmo genetico es el *operador de mutación*. Este operador lo representaremos con un metodo llamado `Mutate` de tipo "`Mutate : T[ ]` $\rightarrow$ `T[ ]`". El cual se le proporciona una aproximación de solución, y la modifica de forma aleatorea para obtener una nueva possible solución.

El tercer elemento de un algoritmo genetico es el *criterio de rendimiento*. Este criterio evalua que tan buena o mala es una solución en particular. Este criterio se representara con un metodo llamado `Score` de tipo `Score : T[ ]` $\rightarrow$ `double`.

El ultimo elemento del algoritmo es un metodo que permite generar soluciones aleatoreas. Este metodo llamado `Guess` tiene tipo `Guess : void` $\rightarrow$ `T[ ]`. Este metodo no recive ningun parametro y retorna una solución generada al azar del problema en mano.

Problema del vendedor ambulante

El problema del vendedor ambulante es uno de los problemas más "molestamente complejo" que hay en la computación. Consiste en encontrar el camino más corto que recorra un grupo de ciudades y que luego regresa a la ciudad de partida. Este problema es un canditato perfecto para aproximar con algoritmos geneticos. Supongamos que se desea recurrer una cantidad de n ciudades. Cada ciudad se representara con un numero $0 \leq i < n$. Con esto, podemos representar un reccorido r como un arreglo de enteros $[i_0 \dots i_n]$ en donde cada indice del arreglo, representa la ciudad a la que se viajara luego de haber visitado el indicie anterior. Supongamos que tenemos las ciudades:

1. Guatemala
2. Managua
3. San Salvador

Entonces, el arreglo $[2, 1, 3]$ representa un recorrido que empieza en Managua, luego pasa por Guatemala, de ahi pasa por San Salvador para luego regresar a Managua. Las distancias entre ciudades, se pueden representar con un arreglo de dos dimensiones llamado d donde una pareja de indices representa la distancia entre dos puntos. Por ejemplo en el ejemplo anterior, las distancias se representarian con el arreglo de dos dimensions (matriz):

0	1020	454
1020	0	511
454	511	0

La cual muestra la distancia entre las ciudades en kilometros por tierra. Para evaluar que tan buena es una solución, simplemente se suman las distancias del recorrido. Para el recorrido $[2, 1, 3]$, su puntaje seria $1020 + 454 + 511$ o 1985.

Con esta información, se puede implementar la clase abstracta `TSP` (Traveling salesman problem), con los siguientes criterios:

- La clase `TSP` debe implementar la interfaz `IProblem`⟨`int`⟩.
- El constructor de `TSP` debe recibir un arreglo de dos dimensiones (matriz) de `double`, el cuar representara las distancias entre un conjunto de ciudades, como se explico anteriormente.
- Los metodos `Crossover` y `Mutate` deben ser abstractos
- El metodo `Score` utiliza la matriz pasada en el construcotr para caluclar la distancia de un recorrido en particular y la retorna.
- El metodo `Guess` retorna un recorrido (array de enteros) al azar. Tomar en cuenta que cada numero **solo debe aparecer una vez** en el arreglo.

Implementar la clase abstracta TSP

Instancia concreta de TSP

Los metodos `Crossover` y `Mutate` para `TSP` tienen varias posibles implementaciónes. El rendimiento del algoritmo genetico depende de dichas implementaciones. En esta sección consideraremos las siguientes:

Para el metodo `Mutate`, dado un arreglo, este metodo seleccionara dos indices aleatoreamente e intercambiara los valores. Por ejemplo, para el arreglo $[2, 1, 3$, si se seleccionan los indices 0 y 3, el resultado seria $[3, 1, 2]$;

El metodo `Crossover`, construye un nuevo arreglo, primero combinando los dos arreglos en un arreglo con el doble de longitud que los arreglos originales y luego seleccionando los numeros del nuevo arreglo en orden hasta tener todos los numeros. En un arreglo de longitud original. Por ejemplo, dados los arreglos $[3, 1, 2]$ y $[2, 1, 3]$, primero creamos el arreglo $[3, 2, 1, 1, 2, 3]$ y luego seleccionamos la primera ocurrencia de cada numero en orden para obtener el arreglo $[3, 2, 1]$.

Crear una clase llamada TSPEjemplo, la cual hereda de la clase TSP e implementa los metodos Mutate y Crossover como esta descrito en esta seccion.

La clase Solver

Crear una clase llamada `Solver`, la cual tiene un metodo estatico generico llamado `Solve` de tipo $\texttt{Solve}\langle\texttt{T}\rangle : \texttt{IProblem}\langle\texttt{T}\rangle \otimes \texttt{double} \rightarrow \texttt{T}[\,]$. Este metodo busca una solución al problema que se le dio, la cual debe tener un punteo (o score) menor al segundo parametro. Para ello sigue los siguientes pasos:

1. Utilizar el metodo `Guess` para generar 5 soluciones aleatoreas
2. Generar otras 5 soluciones aplicando el metodo `Mutate` a cada solución
3. De las 10 soluciones, seleccionar 5 parejas aleatoreamente, utilizar estas 5 parejas para generar otras 5 soluciones más utilizando el metodo `Crossover`
4. De las 15 soluciones, seleccionar las 5 con puntaje o score más bajo, descartar el resto.
5. Si alguna de estas 5 soluciones tiene un puntaje menor al segundo parametro de este metodo, retornar dicha solución, de lo contrario, ir al paso 2 y repetir.

Implementar el algoritmo Solve.

Optimizaciones

Siempre se pueden considerar mejoras, por ejemplo:

- Cambiar el metodo corssover o mutate
- Utilizar un pool mas grande de soluciones candidatos

Tienen la libertad de probar :)