Centro de Investigación en Cómputo
Instituto Politécnico Nacional
Metaheurísticas
Actividad No. 15
Estrategias de remplazo en Algoritmos Genéticos
Curso impartido por: Dra Yenny Villuendas Rey

Adrian González Pardo

7 de diciembre de 2020

1. Ventajas y Desventajas de GA

Ventajas	Desventajas
Permite realizar multiples busqueda de soluciones a los problemas	Puede que el metodo de mutación o cruza seleccionado puede que no ayude a encontrar una buena solución
Esta bioinspirado en la genética y en la selección natural (Darwinismo)	Puede que el que en la aplicación en alguna etapa del GA ya no avance
Es una heurística poblacional	Puede que genere bastante uso de recursos en memoria y procesamiento
Es posible el trabajar soluciones de formas paralelizables o distribuidas	Puede ser dificil de implementar

2. Modelo Generacional vs Estacionario

2.1. Semejanzas

- Ambos generan en cada iteración nuevas respuesta a analizar
- Ambos remplazan a la generación anterior (Con precaución de como son seleccionados y de como trabajan en la siguiente iteración).
- Ambos realizan conceptualmente las mismas operaciones (Solo que de diferente manera a la hora de selección y cruza).

2.2. Diferencias

- El modelo generacional crea una nueva población completa, mientra que el modelo estacionario escoje dos partes de la población de acuerdo al muestreo que realice y sobre ellos aplica los operadores genéticos.
- El modelo generacional remplaza completamente a la anterior generación, mientras que el modelo estacionario remplaza a los M cromosomas con los N descendientes de la población inicial $M \leq N$.
- El modelo generacional tiene un remplazo aleatorio, mientras que el modelo estacionario remplaza a los N peores.
- El modelo generacional teorícamente realiza una excesiva exploración lo cual no garantiza que tenga una convergencia en un optimo local (Explora espacialmente las regiones de solución), el modelo estacionario realiza una excesiva explotación lo cual converge en un optimo local (Busca mejorar al mejor individuo).

3. Operadores de Remplazo

3.1. Aleatorio

Como su nombre lo dice este tipo de operador esta basado en una selección aleatoria entre los padres e hijos, de tal forma en que se obtienen dos nuevos elementos seleccionados del problema de modo en que este tipo de selección puede no considerar el mejor fitness de los padres y los hijos.

3.2. Elitista

A diferencia del de arriba este aumente el costo computacional de modo en que busca encontrar los mejores cromosomas para pasar a la siguiente iteración del algoritmo, este operador genera una mayor presión entre la selectividad de los siguientes dos cromosomas.

3.3. Crowding determinístico

Para este tipo de operador se selecciona a un hijo que sea más parecido al padre de modo en que mantiene una diversidad de remplazo.

3.4. Torneo restringido entre semejantes

Este tipo de operador de acuerdo a un vector generado $w = [w_0, w_1, \cdots, w_N]$ de modo en que se buscan los cromosomas que tengan una mejor semejanza al vector w.

4. Estructuras de datos necesarias para la implementación

Para este tipo de operadores es necesario que pensemos en una lista de arreglos o en el uso de arreglos por separado de modo en que tanto padres como hijos son comparados deacuerdo al modelo u operador seleccionado de modo en que es más sencillo obtener los datos que deseamos, en muchos casos dependera del operador de si pasamos los valores de estos o las funciones fitness evaluadas al momento de la realización de la implementación.

5. Implementación

```ruby
#!/usr/bin/env ruby
class Replacement
  attr_accessor :parent1,:parent2,:child1,:child2
  def initialize(parent1=[],parent2=[],child1=[],child2=[])
    @parent1=parent1
    @parent2=parent2
    @child1=child1
    @child2=child2
  end

  def elitism(case_s=0,minmax=0)
    """
      En este en el mejor de los casos es bueno que los arreglos sean los
      valores evaluados de cada uno de los padres o que al menos se pueda
      obtener el valor de la evaluacion fitness de cada elemento de los arreglos
    """
    puts "Remplazo elitista mediante las funciones fitness"
    hash_min={}
    hash_min[:p1]=@parent1[0]
    hash_min[:p2]=@parent2[0]
    hash_min[:c1]=@child1[0]
    hash_min[:c2]=@child2[0]
    hash_min=hash_min.sort_by {|k, v| -v}
    arr_f1=[]
    arr_f2=[]
    if minmax==0
      puts "Minimizacion"
      arr_f1=hash_min[-1]
      arr_f2=hash_min[-2]
```

```
    else
      puts "Maximizacion"
      arr_f1=hash_min[0]
      arr_f2=hash_min[1]
    end
    puts "Seleccionados #{arr_f1.to_s}, #{arr_f2.to_s}"
    return [arr_f1[0],arr_f2[0]]
  end

  def crowding(semejanza=0)
    """
      En este operador se verificar cuales elementos tienen mejor semejanza con
      respecto a los otros elementos es decir padres con padres y con hijos.
      Este algoritmo analiza elemento a elemento
    """
    hash={:p1p2=>0,:p1c1=>0,:p1c2=>0,:p2c1=>0,:p2c2=>0}
    @parent1.length.times{|i|
      if @parent1[i]==@parent2[i]
        hash[:p1p2]+=1
      end
      if @parent1[i]==@child1[i]
        hash[:p1c1]+=1
      end
      if @parent1[i]==@child2[i]
        hash[:p1c2]+=1
      end
      if @parent2[i]==@child2[i]
        hash[:p2c2]+=1
      end
    }
    hash=hash.sort_by{|k,v| -v}
    puts "Valores ordenados #{hash.to_s}"
    return semejanza==0?[hash[-1][0],hash[-2][0]]:[hash[0][0],hash[1][0]]
  end

  def torneo()
    """
      Se genera aleatoriamente un arreglo de orden para comparar si es
      semejante via indice y asi seleccionar a los cromosomas mas acercados a el
    """
    array=[]
    @parent1.length.times{|i|
      array.push(i)
    }
    array.length.times{|i|
      c=rand(array.length)
      c1=rand(array.length)
      # Permuta el arreglo para saber cual seleccionar
      v=array[c]
      array[c]=array[c1]
      array[c1]=v
    }
    puts "Arreglo generado #{array.to_s}"
    hash={:arrp1=>0,:arrp2=>0,:arrc1=>0,:arrc2=>0}
    @parent1.length.times{|i|
      if @parent1[i]==array[i]
        hash[:arrp1]+=1
      end
      if @parent2[i]==array[i]
        hash[:arrp2]+=1
      end
      if @child1[i]==array[i]
        hash[:arrc1]+=1
      end
      if @child2[i]==array[i]
        hash[:arrc2]+=1
      end
    }
    hash=hash.sort_by{|k,v| -v}
    puts "Tabla generada #{hash.to_s}"
    return [hash[0][0],hash[1][0]]
```

```
  end

end
p1=[10]
p2=[20]
c1=[30]
c2=[20]
r=Replacement.new(p1,p2,c1,c2)
vs=r.elitism(0,1)
vs=vs.to_s
vs[":"]=""
vs[", :"]=","
vs["["]=""
vs["]"]=""
puts "Llaves seleccionadas #{vs.to_s}"
p1=[1,2,4,1,5,1]
p2=[5,2,6,7,5,1]
c1=[1,2,3,1,4,1]
c2=[5,6,7,5,5,1]
r.parent1=p1
r.parent2=p2
r.child1=c1
r.child2=c2
vs=r.crowding(1)
vs=vs.to_s
vs[":"]=""
vs[", :"]=","
vs["["]=""
vs["]"]=""
puts "Elementos semejantes #{vs.to_s}"

p1=[0,2,3,1]
p2=[1,0,3,2]
c1=[0,3,2,1]
c2=[0,1,2,3]
r.parent1=p1
r.parent2=p2
r.child1=c1
r.child2=c2
vs=r.torneo
vs=vs.to_s
vs[":"]=""
vs[", :"]=","
vs["["]=""
vs["]"]=""
puts "Elementos semenjantes al array #{vs.to_s}"
```