Diseño de Algoritmos - Algoritmos II

Nazareno Aguirre
Departamento de Computación
Facultad de Ciencias Exactas, Físico-Químicas y Naturales
Universidad Nacional de Río Cuarto

Clase 12: Algoritmos Genéticos

Estrategias de Diseño de Algoritmos

Ya hemos visto una variedad de estrategias de diseño de algoritmos, que nos brindan herramientas para atacar el problema de construir soluciones algorítmicas para problemas. Hemos visto además que, en algunos casos, debemos conformarnos con soluciones aproximadas para ciertos problemas, debido a la complejidad de los mismos.

Estudiamos también una familia de técnicas de diseño de algoritmos, que denominamos Técnicas de Búsqueda. Veremos ahora una técnica particular, denominada Algoritmos Genéticos, que corresponde a un tipo particular de técnica de búsqueda.

Algoritmos Genéticos

Los algoritmos genéticos, desarrollados por John Holland (University of Michigan, 1970's), utilizan un mecanismo de búsqueda informada, basada en mecanismos que simulan la evolución biológica.

Fueron definidos para entender los procesos adaptativos de los sistemas naturales, y basados en éstos, tratar de diseñar sistemas de software que los imiten (y sean tan robustos como los sistemas naturales).

Los algoritmos genéticos proveen técnicas efectivas para aplicaciones de optimización, de aprendizaje, y muchos otros propósitos (y son ampliamente utilizados en la práctica).

Algoritmos Genéticos y Búsqueda

Los algoritmos genéticos son algoritmos de búsqueda informada, y como tales se aplican a problemas que pueden expresarse como una búsqueda de configuraciones exitosas a partir de ciertas configuraciones iniciales, mediante la aplicación de reglas predefinidas de reconfiguración o avance.

Se debe contar entonces con:

- una descripción precisa de qué constituye el estado o configuración (en algoritmos genéticos, denominada población)
- una descripción del estado de partida (población inicial, generalmente construida aleatoriamente)
- una función heurística, que permita evaluar la calidad de una configuración (denominada función de fitness)
- una descripción de cómo se puede avanzar, en un movimiento, desde una población hacia otra(s) alternativas (evolución).

La Población

En el contexto de algoritmos genéticos, la configuraciones o estados están dadas por conjuntos de individuos denominados poblaciones.

Los individuos, también denominados cromosomas, codifican soluciones candidatas al problema. Cada individuo o cromosoma está en general compuesto por genes, porciones independientes de la caracterización del problema.

Cada gen tiene una posición determinada en el cromosoma/individuo, y representa una propiedad particular del individuo.

Los valores posibles de cada gen son sus alelos. Cada gen puede poseer diferentes alelos.

La Población (cont.)

Consideremos el siguiente ejemplo. Supongamos que queremos resolver el problema de dar cambio con una cantidad mínima de monedas, con denominaciones 25, 10, 5 y 1.

monedas de 10c

monedas de 1c

Cromosoma: Compuesto por 4 genes, cada uno de los cuales representa una cantidad de monedas de una denominación.

v	d	c	u

Gen: Valor entero no negativo.

Gen en pos. 0: cant. de monedas de 25

monedas de 25c

Gen en pos. 1: cant. de monedas de 10

monedas de 5c

...

Alelos de cada gen: [0..] (su dominio)

La Población Inicial

En el contexto de algoritmos genéticos, la configuración inicial está dada por una población, que se suele construir aleatoriamente.

Para el ejemplo de dar cambio, suponiendo que tenemos que dar cambio por 78 centavos, los siguientes podrían ser individuos que constituyan la población inicial

1	1	1	1

0	2	0	0

0	0	0	0

10	0	0	1

1	0	0	1

Poblaciones Óptimas

Una solución al problema de búsqueda subyacente a un algoritmo genético consiste en alcanzar una población optima (o "cercana" al óptimo).

Una población es óptima si contiene una "solución candidata" que resuelve el problema de manera óptima. Es decir, una población es óptima si contiene a un individuo óptimo.

Para el ejemplo de dar cambio, suponiendo que tenemos que dar cambio por 78 centavos, en individuo óptimo es:

3	0	0	3

Cualquier población que contenga a este individuo es una población óptima.

Función de Fitness

Los algoritmos genéticos corresponden a problemas de búsqueda informada. Como tales, deben contar con una función heurística (o función de valoración), que permita evaluar la calidad de una población.

La calidad de una población se evalúa midiendo la calidad de sus individuos. La función de fitness es una función que evalúa la calidad de un individuo.

Para el ejemplo de dar cambio, y suponiendo que se da cambio por un valor menor a 100, la función de fitness podría estar dada por:

fitness (c) = (100 - (c[3]*25+c[2]*10+c[1]*5+c[0]))

Esta función permite determinar que, entre las siguientes dos soluciones candidatas

2	0	0	3

0	0	0	3

la primera es más apta (mejor valor de fitness).

Reglas de Avance (Evolución)

Las reglas de avance en algoritmos genéticos son bastante particulares. Están basadas en la idea de evolución, y consisten en:

combinación de candidatos, denominada crossover, y basada en la idea de reproducción

mutación de candidatos, basada en la idea de cambios imprevistos en la evolución.

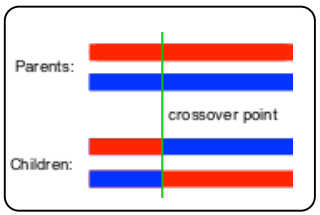
Para hacer evolucionar una población, se eligen individuos para realizar la combinación y/o la mutación, y así aumentar el tamaño de la población.

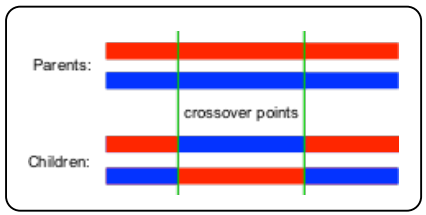
Crossover

La combinación de candidatos (crossover) se puede realizar de muchas maneras diferentes. Si bien uno puede definir operaciones de crossover propias del problema en cuestión, existen algunas operaciones estándares de crossover:

one-point crossover: se elige una posición particular en dos individuos (la misma en ambos), y se construyen hijos combinando la parte previa de uno de los padres con la posterior a la posición en el otro.

two-point crossover: se eligen dos posiciones en dos individuos (las mismas) y se construyen hijos intercambiando las partes centrales de los padres.





Mutación

La mutación es uno de los ingredientes más interesantes de los algoritmos genéticos. Básicamente, una mutación produce un cambio imprevisto en algún individuo de la población.

La mutación ayuda a mantener la diversidad en la población, y en muchos casos a recuperar información perdida en la evolución de la población.

Una de las formas más comunes de mutación es el cambio en el valor de un gen de un cromosoma, por otro de sus alelos.

Antes: (1 0 1 1 0 1 1 0)

Después: (0 1 1 0 0 1 1 0)

Antes: (1.38 -69.4 326.44 0.1)

Después: (1.38 -67.5 326.44 0.1)

Eliminación de Individuos

Evaluar la aptitud de una población depende, evidentemente, del tamaño de la misma. Es por esto esencial mantener acotado el tamaño de las poblaciones.

Por esta razón, durante la evolución se deben eliminar individuos de la población. Los individuos elegidos para eliminar son aquellos "menos aptos" (de acuerdo a la función de fitness), basados claramente en la idea de que los individuos más aptos tienen más chances de sobrevivir (selección natural)

Estructura de un Algoritmo Genético

```
{
    inicializar población;
    evaluar población;
    while (criterio de terminación no satisfecho)
    {
        seleccionar individuos para reproducción;
        realizar crossover y mutación;
        evaluar población;
        descartar algunos individuos;
    }
}
```

Un Ejemplo Simple

El problema del agente viajero

Encontrar un tour tal que:

- cada ciudad se visita sólo una vez
- la distancia total del tour es la mínima posible

Representación

La representación es una lista ordenada de ciudades

1) London	3) Dunedin	5) Beijing	7) Tokyo
2) Venice	4) Singapore	6) Phoenix	8) Victoria

CityList1 (3 5 7 2 1 6 4 8)

CityList2 (2 5 7 6 8 1 3 4)

Crossover

Crossover para este problema

Parent1 (3 5 7 2 1 6 4 8)

Parent2 (2 5 7 6 8 1 3 4)

Child (5 8 7 2 1 6 3 4)

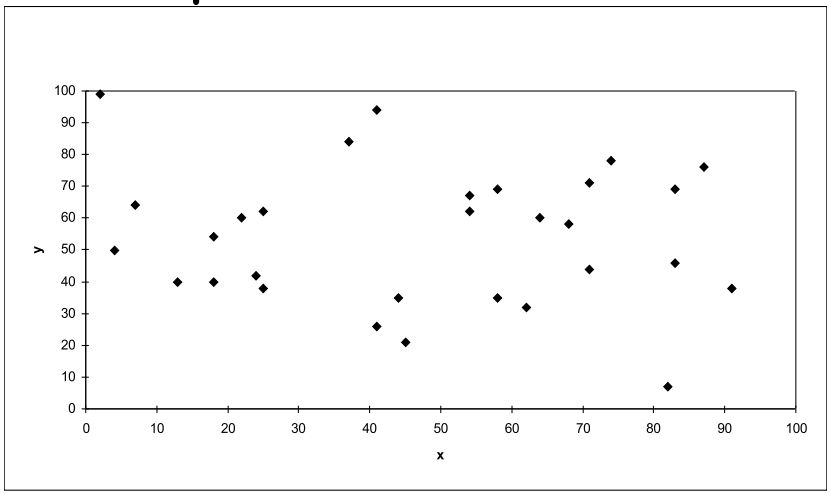
Mutación

Mutación en este caso reordena la lista

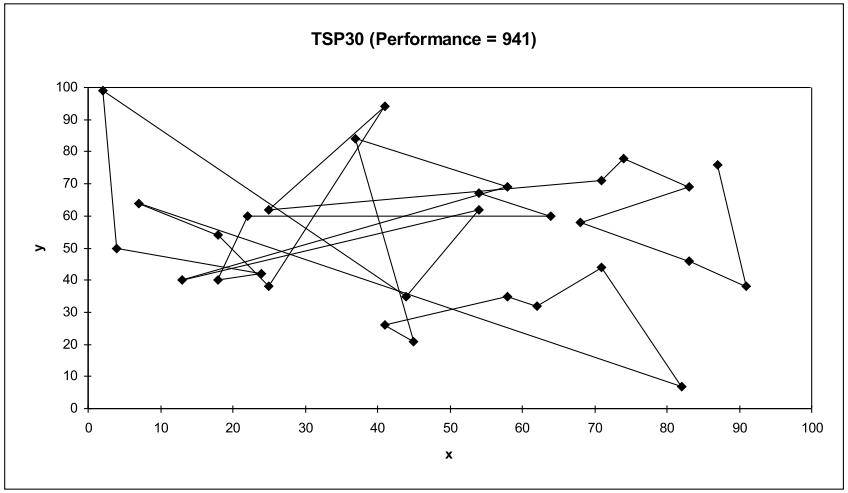
Antes: (5 8 7 2 1 6 3 4)

Después: (5 8 6 2 1 7 3 4)

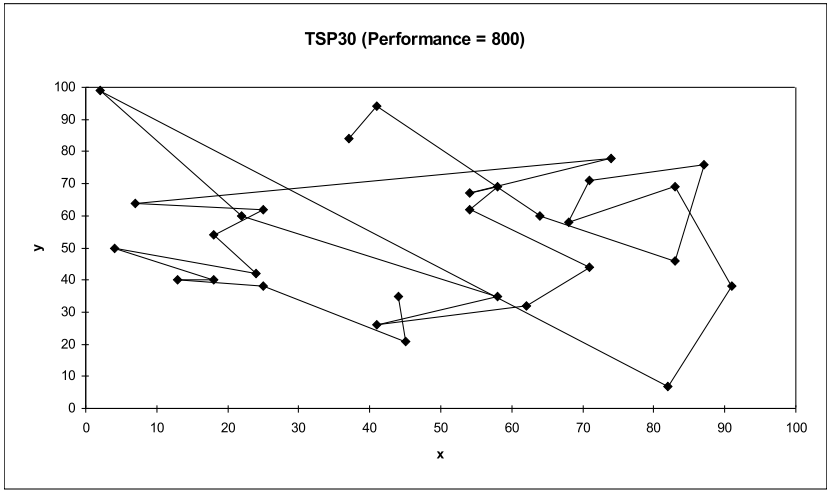
Ejemplo TSP: 30 Ciudades



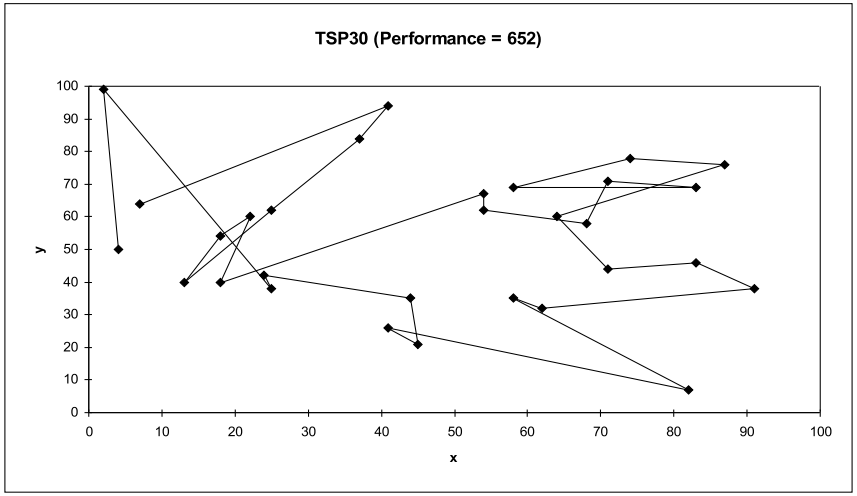
Solución (Distancia = 941)



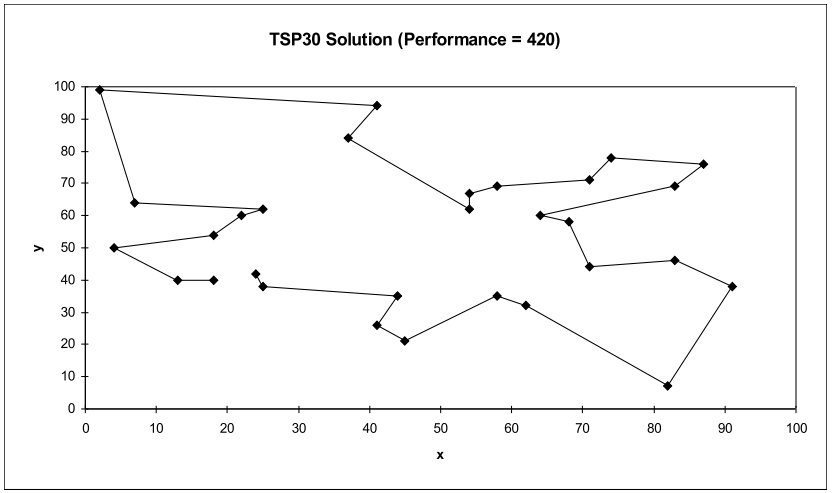
Solución (Distancia = 800)



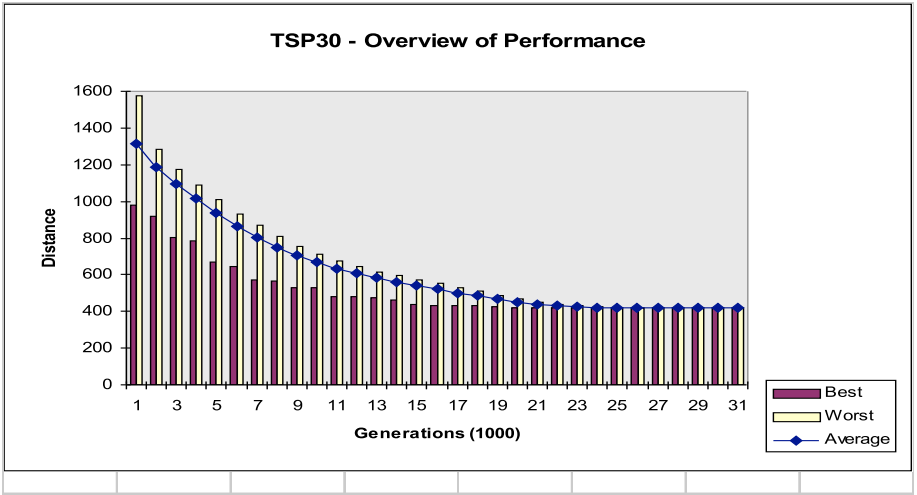
Solución (Distancia = 652)



Mejor Solución (Distancia = 420)



Performance para TSP30



Ejemplo: Algoritmos Genéticos y Optimización de Movimiento

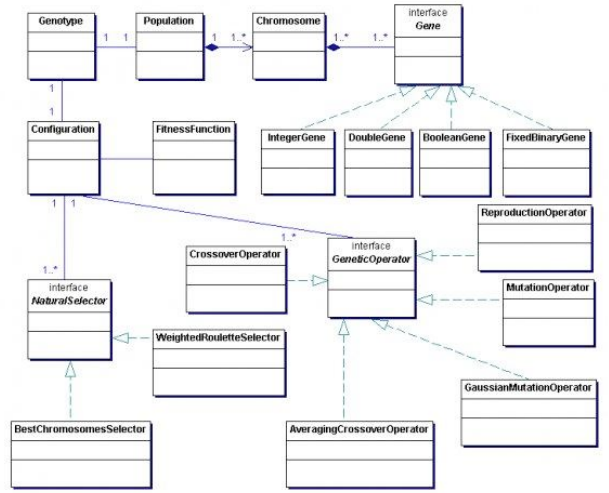
Flexible Muscle-Based Locomotion for Bipedal Creatures

SIGGRAPH ASIA 2013

Thomas Geijtenbeek
Michiel van de Panne
Frank van der Stappen

JGAP: Una Librería para Algoritmos Genéticos

JGAP es una librería para algoritmos genéticos escrita en Java. Básicamente, permite construir algoritmos genéticos instanciando clases e interfaces provistas, y brindando como parte de la librería los algoritmos estándar para hacer evolucionar una población.



Paso 1: Definición de cromosomas

Chromosome with four genes

```
Gene[] sampleGenes = new Gene[ 4 ];

sampleGenes[0] = new IntegerGene(conf, 0, 3 );  // Quarters
sampleGenes[1] = new IntegerGene(conf, 0, 2 );  // Dimes
sampleGenes[2] = new IntegerGene(conf, 0, 1 );  // Nickels
sampleGenes[3] = new IntegerGene(conf, 0, 4 );
```

Paso 2: Implementar la función de fitness

```
public class MinimizingMakeChangeFitnessFunction extends FitnessFunction
{
    private final int m_targetAmount;

    public MinimizingMakeChangeFitnessFunction( int a_targetAmount )
    {
        if( a_targetAmount < 1 || a_targetAmount > 99 )
        {
            throw new IllegalArgumentException(
                "Change amount must be between 1 and 99 cents." );
        }

        m_targetAmount = a_targetAmount;
    }


    public double evaluate( IChromosome a_subject )
    {
        int changeAmount = amountOfChange( a_subject );
        int totalCoins = getTotalNumberOfCoins( a_subject );
        int changeDifference = Math.abs( m_targetAmount - changeAmount );

        double fitness = ( 99 - changeDifference );

        if( changeAmount == m_targetAmount )
        {
            fitness += 100 - ( 10 * totalCoins );
        }

        return fitness;
    }
```

Paso 3: Setear un objeto de configuración

```
Configuration conf = new DefaultConfiguration();

 FitnessFunction myFunc =
   new MinimizingMakeChangeFitnessFunction( targetAmount );

 conf.setFitnessFunction( myFunc );

 Gene[] sampleGenes = new Gene[ 4 ];

 sampleGenes[0] = new IntegerGene(conf, 0, 3 );  // Quarters
 sampleGenes[1] = new IntegerGene(conf, 0, 2 );  // Dimes
 sampleGenes[2] = new IntegerGene(conf, 0, 1 );  // Nickels
 sampleGenes[3] = new IntegerGene(conf, 0, 4 );  // Pennies

 Chromosome sampleChromosome = new Chromosome(conf, sampleGenes );

 conf.setSampleChromosome( sampleChromosome );

 conf.setPopulationSize( 500 );
```

Paso 4: Setear población inicial y hacerla evolucionar

```
Genotype population = Genotype.randomInitialGenotype( conf );

for( int i = 0; i < MAX_ALLOWED_EVOLUTIONS; i++ )
{
    population.evolve();
}

IChromosome bestSolutionSoFar = population.getFittestChromosome();
```