INSTITUTO POLITÉCNICO NACIONAL

ESCUELA SUPERIOR DE CÓMPUTO

ALGORITMO GENÉTICO DE SELECCIÓN POR RULETA

Práctica 3

Dominguez de la Rosa Bryan

GRUPO 3CM5

Profesor: Morales Güitron Sandra Luz

10 de octubre de 2018

Introducción

Contenido

Para la implementación del algoritmo de ruleta, implementé 4 arreglos de bits para controlar las distintas etapas que se realizan en el algoritmo:

- Población inicial.
- Población de individuos seleccionados por ruleta.
- Población después de cruza.
- Población después de mutación.

En la primer etapa, se llena aleatoriamente el arreglo de población inicial con series de 5 bits. Una vez que se tiene la primer población, se ejecuta el algoritmo de ruleta, de la siguiente forma:

1. Se genera un número aleatorio entre 0 y el valor de la aptitud total de la población inicial.
2. Se genera un ciclo en el que en se acumulará la aptitud total de cada individuo de la población.
3. El ciclo se detiene cuando la suma acumulada supere el valor aleatorio generado.
4. El padre seleccionado será el individuo que genere que se sobrepase el valor aleatorio generado.

```
bitset<BITS_PER_INDIVIDUAL> rouletteSelection(bitset<BITS_PER_INDIVIDUAL> set[], int totalAptitude) {

    int r = rand() % (totalAptitude + 1);
    int add = 0;
    int i;
    for(i = 0; i < TOTAL_INDIVIDUALS && add < r; i++) {
        add += getIndividualAptitude(set[i]);
    }
    return set[i];
}
```

Figura 1: Algoritmo genético de selección de ruleta

Una vez teniendo la población de selección de padres, se realiza una cruza de individuos de la siguiente manera:

1. Se utilizan 2 individuos de la población de padres.
2. Se define un punto de cruza estático para todas las generaciones.
3. Se cruzan los individuos.
4. Se retorna el individuo resultante.

```
bitset<BITS_PER_INDIVIDUAL> crossAlgorithm(bitset<BITS_PER_INDIVIDUAL> &p1, bitset<BITS_PER_INDIVIDUAL> &p2, int cross_point) {

    bitset<BITS_PER_INDIVIDUAL> aux = p1;

    for (int i = 0; i <= cross_point; i++)
    {
        aux.set(cross_point - i, p2[cross_point - i]);
    }

    return aux;
}
```

Figura 2: Algoritmo de cruza de individuos

Al obtener la población de individuos después de la cruza se necesita realizar una mutación. En este caso se generó una mutación del 10 % de la población. Nuestra población total es de 32 elementos, entonces la cantidad de individuos a redondear es 3.2, redondeado como 3.

La mutación se realiza de la siguiente forma:

1. El algoritmo se realizará 3 veces.
2. La mutación buscará mejorar al individuo, por lo tanto, se buscará cambiar un bit 0 por un bit 1.
3. Debido a que se requiere buscar un 0 en el individuo a mutar, y es posible que el individuo no tenga bits 0, se define un número máximo de iteraciones para evitar que el programa se cicle.
4. Cuando se encuentre un bit 0, se cambia por un bit 1.

```
bitset<BITS_PER_INDIVIDUAL> mutationAlgorithm(bitset<BITS_PER_INDIVIDUAL> individual){
    bitset<BITS_PER_INDIVIDUAL> result = individual;

    int cont = 0;

    while(cont <= MAX_SEARCH_VALUE)
    {

        int mutation_point = rand() % BITS_PER_INDIVIDUAL;
        if(result[mutation_point] == 0){
            result.set(mutation_point, 1);
            break;
        }
        cont++;
    }

    return result;
}
```

Figura 3: Algoritmo de mutación de individuos

Una vez que se tenga la población mutada, se establece ésta como población inicial, para realizar el algoritmo de ruleta en la siguiente generación.

Al obtener la población final de una generación, se obtiene la aptitud del individuo de menor valor, la aptitud del individuo de mayor valor y el promedio de aptitud de cada generación.

Ejemplo con 5 generaciones:

```
bitset<BITS_PER_INDIVIDUAL> mutationAlgorithm(bitset<BITS_PER_INDIVIDUAL> individual){
    bitset<BITS_PER_INDIVIDUAL> result = individual;

    int cont = 0;

    while(cont <= MAX_SEARCH_VALUE)
    {

        int mutation_point = rand() % BITS_PER_INDIVIDUAL;
        if(result[mutation_point] == 0){
            result.set(mutation_point, 1);
            break;
        }
        cont++;
    }

    return result;
}
```

Figura 4: Resultado de 5 generaciones

Ejemplo con 10 generaciones:

```
bitset<BITS_PER_INDIVIDUAL> mutationAlgorithm(bitset<BITS_PER_INDIVIDUAL> individual){
    bitset<BITS_PER_INDIVIDUAL> result = individual;

    int cont = 0;

    while(cont <= MAX_SEARCH_VALUE)
    {

        int mutation_point = rand() % BITS_PER_INDIVIDUAL;
        if(result[mutation_point] == 0){
            result.set(mutation_point, 1);
            break;
        }
        cont++;
    }

    return result;
}
```

Figura 5: Resultado de 10 generaciones

Ejemplo con 15 generaciones:

```
bitset<BITS_PER_INDIVIDUAL> mutationAlgorithm(bitset<BITS_PER_INDIVIDUAL> individual){
    bitset<BITS_PER_INDIVIDUAL> result = individual;

    int cont = 0;

    while(cont <= MAX_SEARCH_VALUE)
    {

        int mutation_point = rand() % BITS_PER_INDIVIDUAL;
        if(result[mutation_point] == 0){
            result.set(mutation_point, 1);
            break;
        }
        cont++;
    }

    return result;
}
```

Figura 6: Resultado de 10 generaciones

Conclusión