Capítulo 5

Técnicas Incompletas

5.1. Búsqueda Alternativa

Heurística: Son criterios, principios o métodos que permite determinar entre un conjunto de posibilidades, aquella que promete ser la más eficaz para cumplir un objetivo. Las heurísticas representan el compromiso entre dos exigencias: la necesidad de tener criterios simples y al mismo tiempo la necesidad de distinguir entre una buena y una mala elección. Un método heurístico puede ser un método empírico utilizado para guiar acciones. Son algoritmos que buscan encontrar soluciones aceptables. Se usan porque ellas son, en general, eficientes computacionalmente y/o fáciles de implementar. Ellas nos son muy precisas ni predecibles. El mejor resultado se obtiene al mezclar heurísticas con técnicas o algoritmos de optimización.

Reparar una solución: Buscar una nueva solución a partir de una solución existente cambiando el valor que tiene asignado una sola variable. Es decir, se está buscando una nueva solución en el ”vecindario”de la solución actual.

Función Objetivo: Es la función clave dentro de la búsqueda. Debe reflejar el Objetivo que se está buscando y debe ser capaz de medir que tan buena (o mala) es una solución en términos del mismo. Una función objetivo es capaz de representar uno o varios objetivos, esto se logra a través de una ponderación de los ”subobjetivos”, la que es directamente proporcional al grado de importancia de ellos en la búsqueda.

Característica principal: Problemas NP-Completos, Explosión Combinatoria.

5.1.1. TSP (Travel Salesman Problem- Problema del Vendedor Viajero)

Problema:

Una persona debe visitar un conjunto de n ciudades comenzando en una ciudad específica, debiendo regresar a esa misma ciudad, luego de haber visitado todas las ciudades y sin haberse devuelto a ninguna.

Objetivo:

Encontrar la secuencia óptima de visita, medida a través de un criterio como: minimizando costo, minimizando tiempo o maximizando velocidad....

5.1.2. SPP (Space Planning problem - Problema de distribución espacial)

Problema:

Se tiene una pieza (un plano, una planta de un edificio, una placa) en donde se desean posicionar objetos sin que se superpongan.

El problema puede tener objetivos de al menos dos clases:

Optimización: Buscando introducir la mayor cantidad de objetos posibles en la pieza o buscar la menor dimensión de la pieza que permita que todos los objetos sean ubicados dentro de la pieza.

Satisfacción: Buscando la ubicación de los objetos que se sabe caben en la pieza.

5.1.3. TT (TimeTabling problem - Problema de distribución horaria)

Problema:

Una tabla horaria es la localización de un conjunto de reuniones en el tiempo. Una reunión es una combinación de recursos (salas, personas, equipos..) algunos de los cuales están especificados en el problema y otros se asignan como parte de la resolución del problema.

Puede tener básicamente 2 objetivos distintos:

Optimización: Minimizar el tiempo ocioso.

Satisfacción: Alcanzar a realizar todas las actividades en el tiempo dado.

5.1.4. k-Colouring Problem (Colorear grafos con k colores)

Problema:

Colorear un grafo con k colores de tal forma que los nodos adyacentes no tengan el mismo color.

Puede también tener 2 prismas en cuanto a objetivos:

Optimización: Encontrar el valor mínimo de k que permita cumplir con las restricciones.

Satisfacción: Dado un cierto valor k, por ejemplo $k = 3$, colorear el grafo con esos tres colores sin que los nodos adyacentes tengan el mismo color.

Podemos ver que estamos trabajando, en general, con problemas denominados ”Problemas de Satisfacción de Restricciones”(CSP) y ”Problemas de Optimización con Satisfacción de Restricciones”(CSOP) que pertenecen a la clase NP-completos debido a su complejidad.

5.2. Técnicas que usan heurísticas

5.2.1. Hill-Climbing (Escalando)

Corresponde a un término genérico para describir un algoritmo que busca ir mejorando el valor de la función objetivo de una casi-solución, al realizar reparaciones de la misma.

Algoritmos tipo hill-climbing, en general, comienzan con una casi-solución generada en forma aleatoria (asignándoles en forma aleatoria valores a las variables desde su dominio). Luego, realiza reparaciones de esa casi-solución buscando que la función objetivo de la nueva casi-solución sea mejor que la casi-solución actual.

Para ello es necesario contar con:

- Una función objetivo que mida la casi-solución.
- Un criterio para seleccionar la variable que se va a modificar.
- Un criterio para elegir un valor para esa variable seleccionada.

En particular para los problemas de satisfacción de restricciones (CSP), Minton et al. propusieron el siguiente algoritmo iterativo estocástico denominado *min-conflicts: Comienza con una casi-solución generada en forma completamente aleatoria y utiliza como*

- Función objetivo: Número de restricciones satisfechas
- Criterio para seleccionar la variable a modificar: Aleatorio
- Criterio para elegir un valor para la variable: Cambia el valor de la variable ssi existe otro valor dentro del dominio de la misma, que haga satisfacer más restricciones que la casi-solución actual.

Dificultad de *min-conflicts*: Se puede quedar en un óptimo local porque no hay mejores soluciones en el vecindario de la casi-solución actual.

Ejemplo:Heurísticas K-opt para búsqueda local TSP

Heurística K-Opt de Lin, Bell Systems Technical J.44 (1965) 2245-2269
Lin & Kernighan, Operations Research 21 (1973) 498-512

Definición K-exchange: Es un procedimiento que reemplaza K arcos de un tour de TSP por K nuevos arcos, tal que el tour resultante es un tour factible del TSP.

Ejemplo:

1. En el tour: A - B - C - D - E - A
   Reemplace (A, B) & (C, D) por (A , C) & (B, D) tour con “2-exchange”: A - C - B - D - A

2. En el tour para un TSP simétrico: A - B - C - D - E - A
   Reemplace (A, B), (C, D), & (D, E) por (B, D), (A, D), & (C, E) tour con “3-exchange”: A - D - B - C - E - A

Heurística K-Opt:
Sea N(S) el vecindario de una solución S formado por el conjunto de todas las soluciones que se pueden obtener a partir de ejecutar el procedimiento K-exchange sobre S.

Paso 1: Encontrar un tour inicial para el TSP usando cualquier heurística.
Esta es la primera solución S.

Paso 2: Buscar en el vecindario de S, N(S) hasta encontrar una solución S’ con F(S’) $<$ F(S), donde F(.) es la función objetivo a minimizar.
Reemplace S por S’.

Paso 3: Repetir el Paso 2 hasta que no se pueda encontrar una mejor solución en el vecindario N(S).

5.2.2. Tabu Search

Tabu search enfrenta el problema de ciclos impidiendo temporalmente movimientos que podrían hacer volver a una solución que ha sido recientemente visitada.

El efecto es prevenir ciclos a “corta” duración, sin embargo se pueden revisitar soluciones sobre grandes intervalos de tiempo.

Algoritmo Tabu Search:

Paso 0: Inicialización

X:=solución inicial factible

tmax:= máximo número de iteraciones

Mejor solución:=X

número de soluciones = t :=0

lista tabu:= vacía

Paso 1: Parada

Si cualquier movimiento posible de la solución actual es tabu o si t=tmax entonces parar. Entregar Mejor solución.

Paso 2: Mover

Elegir algún movimiento no-tabu factible $\triangle x(t+1)$

Paso 3: Iteración

Modificar $X(t+1) := X(t) + \triangle x(t+1)$

Paso 4: Reemplazar el mejor

Seleccionar aquel $X(t+1)$ dentro del vecindario con el mejor valor de la función objetivo Mejor solución:= X(t+1)

Paso 5: Actualizar Lista Tabu

Eliminar desde la lista tabu cualquier movimiento que ha permanecido un suficiente número de iteraciones en la lista.

Agregar un conjunto de movimientos que involucran un retorno inmediato desde X(t+1) a X(t)

Paso 6: Incrementar

t:=t+1, volver a Paso 1.

5.2.3. Aplicando Tabu Search al problema de la mochila

Problema de la mochila:

$$\text{máx}\, 18X_1 + 25X_2 + 11X_3 + 14X_4$$

s.a.

$$\begin{aligned} 2X_1 + 2X_2 + X_3 + X_4 &\leq 3 \\ X_1, \ldots, X_4 &= 0 \text{ ó } 1 \end{aligned}$$

Supuestos: Movimiento: Cambios de un componente 0 a 1 ó un componente 1 a 0. Solución inicial: (1,0,0,0)

1. Hill-Climbing

Vecinos permitidos:(0,0,0,0),(1,0,1,0) y (1,0,0,1) cuyas funciones de evaluación serían: 0, 29 y 32.

El que más mejora la función objetivo es (1,0,0,1).

Vecinos permitidos: (0,0,0,1) y (1,0,0,0) donde ninguno mejora la función objetivo entonces el óptimo encontrado sería 32.

2. Hill-Climbing con Restart

Nuevas soluciones iniciales: (0,1,0,0) y (0,0,1,0) óptimos serían 36 y 39.

3. Tabu con Sol. inicial: (1,0,0,0) con tmax:= 5 iteraciones

t	X(t)	Valor	Mejor Solución	Movimiento	$\triangle$ Objetivo
0	(1,0,0,0)	18	18	j=4	14
1	(1,0,0,**1**)	32	32	j=1	-18
2	(**0**,0,0,1)	14	32	j=2	25
3	(**0**,**1**,1,1)	39	39	j=4	-14
4	(0,**1**,0,**0**)	25	39	j=3	11
5	(0,1,**1**,**0**)	36	39	parar	

5.2.4. Simulated Annealing

Simulated Annealing controla los ciclos aceptando movimientos que empeoran de acuerdo a una probabilidad comparada con el valor de un número generado aleatoriamente.

El movimiento será aceptado siempre que el movimiento mejore la función objetivo o en el caso que:

Probabilidad de aceptación = $\exp(\triangle obj/q)$

Algoritmo Simulated Annealing

Paso 0: Inicialización

X:=solución inicial factible

tmax:= máximo número de iteraciones

q:= Temperatura alta inicial

Mejor solución:= X

número de soluciones =t :=0

Paso 1: Parada

Si no hay un movimiento posible de la solución actual o si t=tmax entonces parar. Entregar Mejor solución.

Paso 2: Posible Movimiento

Elegir aleatoriamente algún movimiento factible $\triangle x(t+1)$

Calcular el incremento (o disminución) $\triangle$objetivo

Paso 3: Aceptación

Si X(t+1) mejora el objetivo o si $P(\exp(\triangle objetivo/q)) \Rightarrow random(0,1)$.

Modificar $X(t+1) := X(t) + \triangle x(t+1)$ sino volver al Paso 2.

Paso 4: Reemplazar el mejor

Si el valor de la función objetivo de X(t+1) es superior a Mejor solución entonces a Mejor solución:= X(t+1)

Paso 5: reducción de la Temperatura

Si ha pasado un número suficiente de iteraciones desde el último cambio de la temperatura, reduzca q.

Paso 6: Incrementar

t:=t+1, volver a Paso 1.

5.2.5. Aplicando el problema de la Mochila a Simulated Annealing

Solución inicial:= (1,0,0,0)
Temperatura inicial = q :=10
tmax:= 3
Números aleatorios generados: 0.72;0.83;0.33;0.41;0.09;0.94.

t	X(t)	Valor	Mejor Solución	q	Movimiento	△ Objetivo	Decisión
0	(1,0,0,0)	18	18	10	j=4	14	aceptado
1	(1,0,0,1)	32	32	10	j=4	-14	rechazado
					j=1	-18	aceptado
2	(0,0,0,1)	14	32	10	j=3	11	aceptado
3	(0,0,1,1)	25	32	10	parar		

Capítulo 6

Algoritmos Genéticos

6.1. Principales áreas de los A.E.

- Programación Evolucionista $rightarrow$ L. Fogel 1962 (San Diego, CA)
- Algorítmos Genéticos $rightarrow$ J. Holland 1962 (Ann Arbor, MI)
- Estrategias Evolucionistas $rightarrow$ I. Rechenberg & H.P. Schwefel 1965 (Berlin, Germany)
- Programación Genética $rightarrow$ J. Koza 1989 (Palo Alto, cA)

6.2. Introducción

La teoría de la evolución muestra que los seres vivos evolucionan bajo el efecto del medio ambiente: los que están mejor adaptados tienen más posibilidades de sobrevivir y de reproducirse. En cada generación las características de los individuos mejor adaptados tienen más posibilidades de estar presente en la población. La genética cuyo objetivo es estudiar los mecanismos de herencia propone un modelo que permite explicar la transmisin de estas características de una generación a otra.
Existen entidades responsables de la producción de caracteres hereditarios, que se llaman *genes* y el conjunto de genes de un individuo define su *genotipo*. Por otro lado, el *fenotipo* de un individuo corresponde a su apariencia física, más exactamente a un conjunto de caracter
isticas que uno puede observar, medir o calificar en él; el fenotipo es posible que cambie en el tiempo por efecto del medio en el cual evoluciona, pero sus variaciones no pueden transmitirse. Un gen es un segmento de *cromosoma*, parte del ADN que es el material genético de todas las células. Dos mecanismos permiten fabricar nuevas células. El primero es por división celular: una célula duplica su material genético antes de cortarse en dos copias de si misma. O casi copias fieles: un error de reproducción puede producirse, afectando un gen, ocurre entonces una *mutación* de ese gen. Este

mecanismo lleva a cabo la reproducción *asexuada*. El segundo mecanismo hace intrevenir dos padres para fabricar un hijo, es lo que se conoce como reproducción . En este mecanismo las células sexuales transmitidas por los padres aportan cada una la mitad de los cromosomas del hijo.
Los algoritmos genéticos están inspirados en estos mecanismos.

6.3. Estructura de un Programa Evolucionista

Inicio t:=0

inicializar P(t)

evaluar P(t)

Mientras (no sea condición de término) **haga**

t:=t+1

Seleccionar desde P(t-1)

Transformar usando operadores

Evaluar P(t)

Fin

Problema

$$\text{máx}\, f(X_1, X_2) = 21{,}5 + X_1 \sin\{4\Pi X_1\} + X_2 \sin\{20\Pi X_2\}$$

donde

$$\begin{aligned} -3{,}0 \leq \quad X_1 \quad &\leq 12{,}1 \\ 4{,}1 \leq \quad X_2 \quad &\leq 5{,}8 \end{aligned}$$

$\hookrightarrow$ Standard GA

6.4. Representación en un GA

- $X_1 \in [-3{,}0, 12{,}1]$. Este rango se divide en $15{,}1 \times 10000$ rangos de igual tamaño. En consecuencia se requiere 18 bits para representar la variable X_1.
- $X_2 \in [4{,}1, 5{,}8]$. Se divide en $1{,}7 \times 10000$ rangos de igual tamaño. Luego se requiere 15 bits para X_2.
- En total se requiere un string con 33 bits:
(010001001011010000111110010101000 10)

6.5. Conversión y Evaluación

- Si $X \in [a, b]$
- String de largo k que representa X: $(i_1, \ldots, i_k)$.
- El valor de X corresponde a: $a + decimal(i_1, \ldots, i_k) \times (b - a)/(2^k - 1)$.
- (01000100101101000011111001010100010).
- $decimal(i_1, \ldots, i_k) = \sum i_{k-j} \times 2^j, j = 0 \ldots k - 1$
- $X_1 = -3{,}0 + 70352 \times (15{,}1)/262143 = 1{,}0524$
- $X_2 = 5{,}7553$
- Función Objetivo (X_1, X_2)=20.2526

6.5.1. Ejemplo de Operadores

El problema es maximizar $f(x) = x^2$

Número	String	Aptitud	% del Total
1	01101	169	14.4
2	11000	576	49.2
3	01000	64	5.5
4	10011	361	30.9
Total		1170	100.0

Roulette Wheel Selection

$$P = f_1 / \sum_{j-1}^{d} f_j$$

Figura 6.1: Ruleta

One-Point Crossover (SPX)

$$P_c \in [0{,}6 \ldots 1{,}0]$$

Padres:

- 01 | 101 (169)
- 11 | 000 (576)

Descendientes:

- 01000 (64)
- 11101 (841)

Mutación

$$P_m \in [0{,}001 \ldots 0{,}1]$$

Figura 6.2: Mutación

6.6. Algoritmo Genético Standard

Este es una implementación de un algoritmo genético simple, donde la función de evaluación toma valores positivos solamente y la aptitud de un individuo es el mismo valor de la función objetivo.

```
#include <stdio.h> #include <stdlib.h> #include <math.h>

/* Change any of these parameters to match your needs */

#define POPSIZE 20              /* population size */ #define
MAXGENS 1000            /* max. number of generations */ #define
NVARS 2                 /* no. of problem variables */ #define
PXOVER 0.25             /* probability of crossover */ #define
PMUTATION 0.01          /* probability of mutation */ #define
TRUE 1 #define FALSE 0 #define PI 3.14159

int generation;                 /* current generation no. */ int
cur_best;                  /* best individual */ FILE *galog;
/* an output file */

struct genotype /* genotype (GT), a member of the population */ {
  double gene[NVARS];        /* a string of variables */
  double fitness;            /* GT's fitness */
  double upper[NVARS];       /* GT's variables upper bound */
  double lower[NVARS];       /* GT's variables lower bound */
  double rfitness;           /* relative fitness */
  double cfitness;           /* cumulative fitness */
};
```

```
struct genotype population[POPSIZE+1];    /* population */ struct
genotype newpopulation[POPSIZE+1]; /* new population; */
                                          /* replaces the */
                                          /* old generation */

/* Declaration of procedures used by this genetic algorithm */

void initialize(void); double randval(double, double); void
evaluate(void); void keep_the_best(void); void elitist(void); void
select(void); void crossover(void); void Xover(int,int); void
swap(double *, double *); void mutate(void); void report(void);
```

6.6.1. Función de inicialización

Inicializa los valores de los genes dentro de los límites de las variables. También inicializa (en cero) todos los valores de aptitud (*fitness*) para cada miembro de la población. La función lee los límites superiores e inferiores de cada variable desde un archivo de entrada *'gadata.txt'*, y genera aleatoriamente valores entre estos límites para cada gen de cada genotipo en la población. El formato del archivo de entrada 'gadata.txt' es:

```
var1_lower_bound var1_upper bound\\
var2_lower_bound var2_upper bound\\


void initialize(void) { FILE *infile; int i, j; double lbound,
ubound;

if ((infile = fopen("gadata.txt","r"))==NULL)
      {
      fprintf(galog,"\nCannot open input file!\n");
      exit(1);
      }

/* initialize variables within the bounds */

for (i = 0; i < NVARS; i++)
      {
      fscanf(infile, "%lf",&lbound);
      fscanf(infile, "%lf",&ubound);

      for (j = 0; j < POPSIZE; j++)
           {
           population[j].fitness = 0;
           population[j].rfitness = 0;
           population[j].cfitness = 0;
```

```
            population[j].lower[i] = lbound;
            population[j].upper[i]= ubound;
            population[j].gene[i] = randval(population[j].lower[i],
                                    population[j].upper[i]);
            }
      }

fclose(infile); }

/*************************************************************/ /*
Random value generator: Generates a value within bounds */
/*************************************************************/

double randval(double low, double high) { double val;
val = ((double)(rand()%1000)/1000.0)*(high - low) + low;
return(val); }
```

6.6.2. Función de evaluación

Esta función toma una función definida por el usuario. Cada vez que es cambiada, el código debe ser recompilado.

La función dada en el ejemplo es:

$$21{,}5 + x[1] \times \sin\{4 \times PI \times x[1]\} + x[2] \times \sin\{20 \times PI \times x[2]\}$$

```
void evaluate(void) { int mem; int i; double x[NVARS+1];

for (mem = 0; mem < POPSIZE; mem++)
      {
      for (i = 0; i < NVARS; i++)
            x[i+1] = population[mem].gene[i];

      population[mem].fitness =
      21.5 + x[1]*sin(4*PI*x[1]) + x[2]*sin(20*PI*x[2]);
      }
}

/*************************************************************/
/* Keep_the_best function: This function keeps track of the    */
/* best member of the population. Note that the last entry in  */
/* the array Population holds a copy of the best individual    */
/*************************************************************/

void keep_the_best() { int mem; int i; cur_best = 0; /* stores the
index of the best individual */
```

```
for (mem = 0; mem < POPSIZE; mem++)
      {
      if (population[mem].fitness > population[POPSIZE].fitness)
            {
            cur_best = mem;
            population[POPSIZE].fitness = population[mem].fitness;
            }
      }
/* once the best member in the population is found, copy the genes
*/ for (i = 0; i < NVARS; i++)
      population[POPSIZE].gene[i] = population[cur_best].gene[i];
}
```

6.6.3. Función elitista

El mejor miembro de la generación previa es almacenada como el último en el arreglo. Si el mejor miembro de la generación actual es peor que el mejor miembro de la generación anterior, este último reemplazaría al peor miembro de la población actual.

```
void elitist() { int i; double best, worst;             /* best
and worst fitness values */ int best_mem, worst_mem; /* indexes of
the best and worst member */

best = population[0].fitness; worst = population[0].fitness; for
(i = 0; i < POPSIZE - 1; ++i)
      {
      if(population[i].fitness > population[i+1].fitness)
            {
            if (population[i].fitness >= best)
                  {
                  best = population[i].fitness;
                  best_mem = i;
                  }
            if (population[i+1].fitness <= worst)
                  {
                  worst = population[i+1].fitness;
                  worst_mem = i + 1;
                  }
            }
      else
            {
            if (population[i].fitness <= worst)
                  {
                  worst = population[i].fitness;
```

```
                worst_mem = i;
                }
            if (population[i+1].fitness >= best)
                {
                best = population[i+1].fitness;
                best_mem = i + 1;
                }
            }
        }
/* if best individual from the new population is better than */ /*
the best individual from the previous population, then    */ /*
copy the best from the new population; else replace the   */ /*
worst individual from the current population with the     */ /*
best one from the previous generation                     */

if (best >= population[POPSIZE].fitness)
    {
    for (i = 0; i < NVARS; i++)
       population[POPSIZE].gene[i] = population[best_mem].gene[i];
    population[POPSIZE].fitness = population[best_mem].fitness;
    }
else
    {
    for (i = 0; i < NVARS; i++)
       population[worst_mem].gene[i] = population[POPSIZE].gene[i];
    population[worst_mem].fitness = population[POPSIZE].fitness;
    }
}
```

6.6.4. Función de selección

La selección proporcional estándar para los problemas de maximización incorporan el modelo elitista -asegurarse que el mejor miembro sobreviva.

```
void select(void) { int mem, i, j, k; double sum = 0; double p;

/* find total fitness of the population */ for (mem = 0; mem <
POPSIZE; mem++)
      {
      sum += population[mem].fitness;
      }

/* calculate relative fitness */ for (mem = 0; mem < POPSIZE;
mem++)
      {
      population[mem].rfitness =  population[mem].fitness/sum;
```

```
      }
population[0].cfitness = population[0].rfitness;

/* calculate cumulative fitness */ for (mem = 1; mem < POPSIZE;
mem++)
      {
      population[mem].cfitness =  population[mem-1].cfitness +
                          population[mem].rfitness;
      }

/* finally select survivors using cumulative fitness. */

for (i = 0; i < POPSIZE; i++)
      {
      p = rand()%1000/1000.0;
      if (p < population[0].cfitness)
            newpopulation[i] = population[0];
      else
            {
            for (j = 0; j < POPSIZE;j++)
                  if (p >= population[j].cfitness &&
                              p<population[j+1].cfitness)
                        newpopulation[i] = population[j+1];
            }
      }
/* once a new population is created, copy it back */

for (i = 0; i < POPSIZE; i++)
      population[i] = newpopulation[i];
}
```

6.6.5. Selección crossover

Selecciona dos padres para realizar el crossover. Implementa un crossover de un solo punto.

```
void crossover(void) { int i, mem, one; int first  =  0; /* count
of the number of members chosen */ double x;

for (mem = 0; mem < POPSIZE; ++mem)
      {
      x = rand()%1000/1000.0;
      if (x < PXOVER)
            {
            ++first;
            if (first % 2 == 0)
```

```
                Xover(one, mem);
            else
                one = mem;
            }
      }
}
```

6.6.6. Crossover

Implementa un crossover de dos padres seleccionados.

```
void Xover(int one, int two) { int i; int point; /* crossover
point */

/* select crossover point */ if(NVARS > 1)
   {
   if(NVARS == 2)
         point = 1;
   else
         point = (rand() % (NVARS - 1)) + 1;

   for (i = 0; i < point; i++)
        swap(&population[one].gene[i], &population[two].gene[i]);

   }
}
```

6.6.7. Swap

Un procedimiento de swap que ayuda en el swapping de dos variables.

```
void swap(double *x, double *y) { double temp; temp = *x; *x = *y;
*y = temp; }
```

6.6.8. Mutación

Mutación aleatoria uniforme. Una variable seleccionada para mutación es reemplazada por un valor aleatorio entre los límites inferior y superior para esa variable.

```
void mutate(void) { int i, j; double lbound, hbound; double x;

for (i = 0; i < POPSIZE; i++)
      for (j = 0; j < NVARS; j++)
            {
            x = rand()%1000/1000.0;
            if (x < PMUTATION)
```

```
                {
                /* find the bounds on the variable to be mutated */
                lbound = population[i].lower[j];
                hbound = population[i].upper[j];
                population[i].gene[j] = randval(lbound, hbound);
                }
            }
}
```

6.6.9. Función de reporte

Reporta el progreso de la simulación. Datos descargados en el archivo de salida son separados por comas.

```
void report(void) { int i; double best_val;           /* best
population fitness */ double avg;                /* avg
population fitness */ double stddev;             /* std.
deviation of population fitness */ double sum_square;          /*
sum of square for std. calc */ double square_sum;          /*
square of sum for std. calc */ double sum;                 /*
total population fitness */

sum = 0.0; sum_square = 0.0;

for (i = 0; i < POPSIZE; i++)
      {
      sum += population[i].fitness;
      sum_square += population[i].fitness * population[i].fitness;
      }

avg = sum/(double)POPSIZE; square_sum = avg * avg * POPSIZE;
stddev = sqrt((sum_square - square_sum)/(POPSIZE - 1)); best_val =
population[POPSIZE].fitness;

fprintf(galog, "\n%5d,      %6.3f, %6.3f, %6.3f \n\n", generation,
                                    best_val, avg, stddev);
}
```

6.6.10. Función principal (main)

Cada generación implica seleccionar el mejor miembro, aplicar crossover y mutación, luego evaluar la población resultante, hasta que la condición de término es satisfecha.

```
void main(void) { int i;
```

```
if ((galog = fopen("galog.txt","w"))==NULL)
      {
      exit(1);
      }
generation = 0;

fprintf(galog, "\n generation  best  average  standard \n");
fprintf(galog, " number      value fitness  deviation \n");

initialize(); evaluate(); keep_the_best();
while(generation<MAXGENS)
      {
      generation++;
      select();
      crossover();
      mutate();
      report();
      evaluate();
      elitist();
      }
fprintf(galog,"\n\n Simulation completed\n"); fprintf(galog,"\n
Best member: \n");

for (i = 0; i < NVARS; i++)
   {
   fprintf (galog,"\n var(%d) = %3.3f",i,population[POPSIZE].gene[i]);
   }
fprintf(galog,"\n\n Best fitness = %3.3f",population[POPSIZE].fitness);
fclose(galog); printf("Success\n"); }
```

6.7. Operadores No-standard

Figura 6.3: Operadores No-standard

6.8. Ideas: Operadores de Recombinación

1. Two-points crossover. Dos padres → dos hijos

2. Uniform crossover: Dos padres → dos hijos Selección aleatoria bit por bit.

3. (n,p,g): Un padre → un hijo

   - n: número de variables a modificar $\{\sharp \ldots\}$
   - p: criterio de selección de las posiciones a modificar {r,b}
   - g: criterio de selección de los nuevos valores {r,b}

4. Knowledge-augmented crossover. Dos padres → un hijo

   - Si las variables no están en conflicto en *ambos* padres elegir uno al hazar
   - Si hay un conflicto en un sólo padre, heredar del padre que no está en conflicto
   - Si los dos padres están en conflicto elegir el menos conflictivo de los dos.

5. UAX: Dos padres → un hijo

   Procedure UAX (Parent-1,Suppl-1, parent-2, suppl-2)
   Begin r¡pcross *parent-select*=selection(Parent-1,parent-2) parent-not-selected = (Parent-1, parent-2) - (*parent-select*) i:=1 to n $Suppl - parent - select$[i] = suppl-parent-not-selecte[i] interchange parents child[i]=$parent - select$[i] suppl-child[i]=$suppl - parent - select$[i] **End**

6.9. Representación Ordinal

- Lista de referencia (1 2 3 4 5 6 7 8 9)
- Un tour: 1 - 2 - 4 - 3 - 8 - 5 - 9 - 7
- Lista: (1 1 2 1 4 3 1 1)
- Cruzar dos padres:
  P1: (1 1 2 1 | 4 1 3 1 1)
  P2: (5 1 5 5 | 5 3 3 2 1)

  $Rightarrow$
  1 - 2 - 4 - 3 - 8 -5 - 9 - 6 - 7
  5 - 1 - 7 - 8 - 9 - 4 - 6 - 3 - 2

  1 - 2 - 4 - 3 - 9 - 7 - 8 - 6 - 5
  5 - 1 - 7 - 8 - 6 - 2 - 9 - 3 - 4

6.10. Problema del Vendedor Viajero

Aspectos Representación.

1. Tour: 5-1-7-8-9-4-6-2-3
   Representación:(5 1 7 8 9 4 6 2 3)
   Operador: PMX (Partially Mapped Crossover)
   P1:(1 2 3 | 4 5 6 7 | 8 9)
   P2:(4 5 2 | 1 8 7 6 | 9 3)

   a) (X X X | 1 8 7 6 | X X)
   (X X X | 4 5 6 7 | X X)
   b) Sin conflictos
   (X 2 3 | 1 8 7 6 | X 9)
   (X X 2 | 4 5 6 7 | 9 3)
   c) Cambios
   (4 2 3 | 1 8 7 6 | 5 9)
   (1 8 2 | 4 5 6 7 | 9 3)

2. Lista referencia (1 2 3 4 5 6 7 8 9)
   Tour: 1-2-4-3-8-5-9-6-7
   Representación: (1 1 2 1 4 1 3 1 1)
   P1:(1 1 2 1 | 4 1 3 1 1)
   P2:(5 1 5 5 | 5 3 3 2 1)

   a) Padres
   (1 2 4 3 8 5 9 6 7)
   (5 1 7 8 9 4 6 3 2)
   b) Hijos
   (1 2 4 3 9 7 8 6 5)
   (5 1 7 8 6 2 9 3 4)

6.11. Problema

Características:

- Posee una cantidad finita de objetos (no es un problema de optimización)
- Diferentes dimensiones de objetos (sólo rectángulos)

Restricciones:

- No se permiten rotaciones
- Las dimensiones de un objeto no pueden salir del espacio delimitado
- No se permiten superposiciones entre los objetos