Analisis y Diseño de Algoritmos

Proyecto Final
Optimizacion

José de Jesús Mendoza Becerril 2173034496
Brayan Santiago Sánchez Reyna 2163031947
November 25, 2019

0.1 Estrategia de Implementación

En esta sección se mostrarán tanto las funciones como estructuras utilizadas para la resolución del problema y por cada una de ellas se realizara una breve descripción de lo que se hizo.

- Struct usada para representar un individuo

```
1              typedef struct{
2              int *x; // cadena binaria (genotipo)
3              double f; // fenotipo
4              }INDIVIDUO;
5
```

Para representar la abstracción de un individuo, se utilizó la struct anterior, en la que cuenta con una cadena binaria que tendrá información sobre sus alelos o cromosomas. Y por otra parte cuenta con su fenotipo que es la evaluación de su genotipo con las funciones alpha1, alpha2 solicitadas en la practica, con la operación lógica XOR.

- Función f1

```
1      double f1(int *x, int n){ //XOR con la norma
2      //printf("Este es n%i\n",n);
3      int *alpha1 = calloc(n, sizeof(int));
4      double norm = 0;
5      for (int i = 0; i < n; i++){ // 1 0 1 0 1 0 1 0 1 0 1 0
6          if(i%2 == 0) alpha1[i] = 1;
7          else alpha1[i] = 0;
8      }
9
10     for (int i = 0; i < n; i++){alpha1[i] = x[i] ^ alpha1[i
       ];} //XOR
11     for (int i = 0; i < n; i++) norm+= alpha1[i]; //Calculo
       norma
12
13     norm = sqrt(norm);
14     return norm;
15 }
16
```

Con esta función se crea alpha1 que es el arreglo con el que se le aplicará XOR, después se aplica xor a la misma y después de la aplicación de la función se obtiene su norma la cual se guarda en f del individuo, esta alpha fue la que constaba de la siguiente secuencia [0,1,0,1,0,1,0,1]

- Función f2

```
   double f2(int *x, int n){ //XOR con la norma
    //printf("Este es n%i\n",n);
    int *alpha2 = calloc(n, sizeof(int));
    double norm = 0;
    int con = 0;
  int v = 1;
  for (int i = 0; i < n; i++){ // 0 0 0 1 1 1 0 0 0 1 1 1
    if(con < 3){ alpha2[i] = v; con++; }
    else if(con == 3 && v == 1){ v = 0; con = 0; i--;}
    else if(con == 3 && v == 0){ v = 1; con = 0; i--;}
  }
    for (int i = 0; i < n; i++){alpha2[i] = x[i] ^ alpha2[i
   ];} //XOR
    for (int i = 0; i < n; i++) norm+= alpha2[i]; //Calculo
   norma
    norm = sqrt(norm);
    return norm;
}

```

Con esta función se crea alpha2 que es el otro arreglo con el que se aplica XOR, y después se calcula la norma, para poder asignar la aptitud al individuo, esta alpha fue la que constaba de la siguiente secuencia [0,0,0,1,1,1,0,0,0,1,1,1,0,0,0,1,1]

- Función Objetivo

```
        void funcion_objetivo(INDIVIDUO *ind, int n){
            ind->f = f1(ind->x,n); //cambie x f2
        }

```

Mediante el uso de esta función se seleccionaba la función objetivo es decir a la cual se quería encontrar otra función con la cual su función de aptitud aplicada a la xor se obtenía la aptitud y se asigna a cada individuo en su campo f.

- Alojar Individuos

```
        void alloc_inds(INDIVIDUO *P, int N, int n){
    for(int i=0; i<N; i++){
        P[i].x = calloc(n, sizeof(int));
    }
}
```

```

```

Esta funcion sirve para reservar memoria para los alelos de cada uno de los N individuos de la Poblacion P.

- Liberar Individuos

```
      void free_inds(INDIVIDUO *P, int N){
            for(int i=0; i<N; i++){
                free(P[i].x);
            }
    }

```

Con esta función se libera memoria de cada uno de los alelos de la cadena binaria de los individuos.

- Inicializar Poblacion

```
    void inicializa_pop(INDIVIDUO *P, int N,int n){
     for(int i=0; i<N; i++) // recorre los individuos
     {
        for(int j=0; j<n; j++) // recorre los bits
        {
            P[i].x[j] = (randomperc()<0.5)? 1: 0;
        }
     }
    }

```

Función mediante la cual se recorre a cada uno de los individuos de la poblacion, y se llena aleatoriamente cada uno de los alelos del genotipo ya sea con 0 o 1, mediante una funcion probabilistica.

- Evaluar Poblacion

```
   void evaluar_pop(INDIVIDUO *P, int N, int n){
    for(int i=0; i<N; i++) // recorre los individuos
    {
        funcion_objetivo(&P[i],n);
    }
}

```

Con esta funcion se recorre a cada uno de los individuos de la Población P y dependiendo la función objetivo se hace la evaluación para obtener

su fenotipo, esto haciendo uso de la función y operación xor para sacar su norma.

- Torneo Binario

```
INDIVIDUO *torneo(INDIVIDUO *ind1, INDIVIDUO *ind2){
    if(ind1->f <= ind2->f ) return ind1;
    else return ind2;

}

```

Para la selección de los padres que serán candidatos a ser cruzados y mutados para crear la nueva generación se utilizó un torneo binario que tan solo consiste en tomar al mejor Individuo entre un par de ellos, es decir el cual tenga su fenotipo menor.

- CrossOver Individual

```
INDIVIDUO crossover_Ind(INDIVIDUO *ind1, INDIVIDUO *ind2,
    int n){
  int c = n/2; //cuantos de cada papa
  INDIVIDUO nuevo;
  nuevo.x = calloc(n, sizeof(int));
    nuevo.f = 0;

  for (int i = 0; i < n; i++){
    if(i < c){ //Papa1
      nuevo.x[i] = ind1->x[i];
    }
    else{ //Papa2
      nuevo.x[i] = ind2->x[i];
    }
  }
return nuevo;
}

```

La cruza de un par de inviduos se hacia de la siguiente manera: -Se tomaron los primeros c alelos del Padre 1 , y los restantes del Padre 2, de tal manera que el nuevo individuo tendría cierta información genética de cada uno de los dos. Para ello tan solo se recorrieron a los padres y se extrajeron los alelos correspondientes.

- CrossOver Poblacional

```
void crossover_pop(INDIVIDUO *P, INDIVIDUO *Q, int N, int n)
    {
        int barajeo1[N];
        int barajeo2[N];
        int ganadores1[N/2];
        int ganadores2[N/2];
        for (int i = 0; i < N; i++){barajeo1[i] = i;}
        for (int i = 0; i < N; i++){barajeo2[i] = i;}
        for (int i = 0; i < N; i++){ganadores1[i] = 0;
    ganadores2[i]=0;}

        myshuffle(barajeo1,N);//Barajeo 1
        myshuffle(barajeo2,N); //Barajeo 2

        int j=0;

        for (int i = 0; i < N; i+=2){
            if(P[barajeo1[i]].f <= P[barajeo1[i+1]].f ){
    ganadores1[j] = barajeo1[i];}
            else { ganadores1[j] = barajeo1[i+1];}
            j++;
        }
        j=0;
        for (int i = 0; i < N; i+=2){
            if(P[barajeo2[i]].f <= P[barajeo2[i+1]].f ){
    ganadores2[j] = barajeo2[i];}
            else { ganadores2[j] = barajeo2[i+1];}
            j++;
        }

        //Cruza
        j=0; //meto a Q iniciando desde 0
        for (int i = 0; i < N/2; i+=2){ //Cruzo ganadores1
          INDIVIDUO h1 = crossover_Ind(&P[ganadores1[i]],&P[
    ganadores1[i+1]],n);
          INDIVIDUO h2 = crossover_Ind(&P[ganadores1[i+1]],&
    P[ganadores1[i]],n);
          Q[j] = h1;
          Q[j+1] = h2;
          j+=2; //Muevo en Q
        }
        j= N/2; //Meto a Q iniciando desde mitad para
    adelante
        for (int i = 0; i < N/2; i+=2){ //Cruzo ganadores2
           INDIVIDUO h1 = crossover_Ind(&P[ganadores2[i]],&P
```

```
    [ganadores2[i+1]],n);
39          INDIVIDUO h2 = crossover_Ind(&P[ganadores2[i
    +1]],&P[ganadores2[i]],n);
40          Q[j] = h1; //Meto en Q
41          Q[j+1] = h2;
42          j+=2; //Muevo en Q
43        }
44
45      printf("\n");
46  }
47
```

La función de cruza Poblaciónal fue una de las mas importantes para el algoritmo y se realizó de la siguiente manera: Primero se barajearon las posiciones de toda la población un par de veces. Teniendo los dos barajeos por separado, se tomaron entonces los individuos i y i+1, de esta manera se aseguraran que no se pudieran poner a competir los mismos en los torneos binarios.

Después se realizaron los torneos binarios con los cuales se obtuvieron N ganadores y ya teniendo las posiciones de los ganadores tan solo se sacaban de P para ser insertados en Q. Para la realización del barajeo se utilizó la función llamada suffle que se describirá después.

Al final Q ya tendría los N nuevos hijos, que tendrán su fenotipo mejorado con cierta probabilidad ya que los mejores padres fueron los que tenían mas probabilidad de aparecer ya que iban ganando los torneos binarios y los posibles a cruzar.

- Mutacion

```
1   void mutacion_pop(INDIVIDUO *Q, int N, int n){  //cambie
    N por n
2    double Pm = (double) 1 / n;
3    double a=0;
4    //printf("Pm %lf\n",Pm);
5    for (int i = 0; i < N; i++){
6        for (int j = 0; j < n; j++){
7             a = randomperc();
8            //printf("%lf\n",a);
9            if(a < Pm){
10               if(Q[i].x[j] == 0) Q[i].x[j] = 1;
11               else Q[i].x[j] = 0;
12           }
```

```
13          }
14
15      }
16  }
17
```

La mutación de los individuos tan solo consistió en la posibilidad de cambiar alguno de los valores de su cadena binaria. Para ello lo primero que se hizo fue calcular la probabilidad con la que se tomaría la decisión si se cambiaba el valor o no, después se recorrieron cada uno de los individuos de la población y para cada uno de los individuos se recorrieron sus alelos, en cada una de estas iteraciones se calculo un valor aleatorio entre 0 y 1 y si este valor era menor que Pm entonces el alelo se cambiaba a su valor contrario en caso contrario se quedaría igual.

- Copiar Poblacion

```
1   void copy_pop(INDIVIDUO *P, INDIVIDUO *Q, int N){
2    // P=Q
3    for (int i = 0; i < N; i++){
4        P[i] = Q[i];
5    }
6   }
7
```

Simplemente se copiaron todos los elementos de Q a P

- Unir Población

```
1   void union_pop(INDIVIDUO *M, INDIVIDUO *P, INDIVIDUO *Q,
     int N){
2   for (int i = 0; i < N; i++){
3     M[i] = P[i];
4   }
5   for (int i = 0; i < N; i++){
6     M[i+(N)] = Q[i];
7   }
8   }
9
```

Con esta función se extrajeron los N individuos de P y los N individuos de Q y fueron depositados en M

- Seleccionar Población

```
1  void seleccionar(INDIVIDUO *P, INDIVIDUO *M, int N){ // P
     = {a los N mejores individuos de M}
2  for (int i = 0; i < N; i++){ P[i] = M[i];}
3 }
4
```

Con esta función se extrajeron los N individuos de M y fueron colocados en P, dado que antes de utilizar esta función, todos los elementos de M fueron ordenados con QuickSort entonces se aseguraba que los Individuos que se iban a extraer serian los mejores de toda la población, es decir, los que tenían mejor aptitud.

- Imprimir Poblacion

```
1  void imprimirIndividuo(INDIVIDUO *ind1 , int n){
2   printf("[");
3   for (int i = 0; i < n; i++){
4     printf("%d",ind1->x[i]); //Imprime Alelos o cadena de
      bits
5   }
6   printf("] ");
7     printf("Aptitud %lf\n",ind1->f);
8 }
9  void imprimirPoblacion(INDIVIDUO *P, int N, int n){ //N
     numero de Agentes Poblacion  n numero de Bits Cadena o
     alelos
10  for (int i = 0; i < N; i++){
11    imprimirIndividuo(&P[i],n);
12  }
13 }
14
```

Esta fue una función auxiliar con la cual solo imprimía cada uno de los individuos con su cadena de bits y su aptitud, realizaba esto para toda la población en General.

0.2 Algoritmo AG-(miu,lambda)

```
1
2               evaluar_pop(P, N,n);
3               imprimirPoblacion(P,N,n);
4               obtenerMejor(P,N,0);
5
6               for(int gen=1; gen<=max_gen; gen++){ //Miu , lambda
    LISTO
7                   printf("\nGeneration:%d",gen);
8                   crossover_pop(P, Q, N,n);
9                   mutacion_pop(Q, N,n);
10                  evaluar_pop(Q, N,n);
11                  // (mu, lambda)-GA
12                  imprimirPoblacion(Q,N,n);
13                  theBest = obtenerMejor(Q,N,gen);
14                  if(theBest==0)break;
15                  copy_pop(P, Q,N);
16              }
17
```

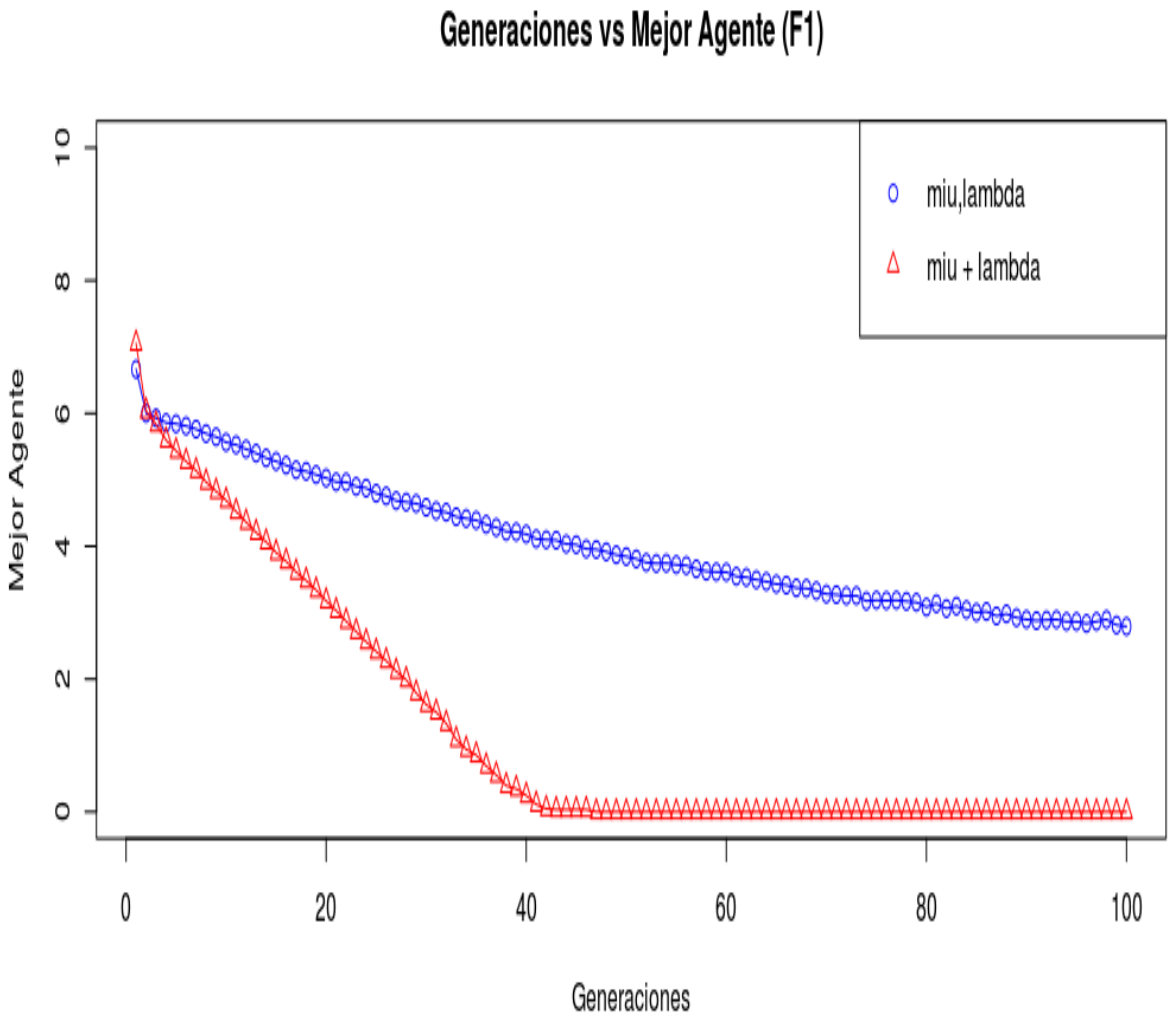
Para el primer Algoritmo primero se evaluó la población P, y cada que se evaluaba después se obtenía al mejor Agente. Después maxgen numero de generaciones se realizo la Cruza de los elementos de P para generar los nuevos elementos en Q. Lo siguiente fue mutar esos N elementos que se encontraban en Q. Como ya son nuevos individuos entonces también se debió evaluar a cada uno de ellos para obtener su fenotipo correspondiente. Se volvió a imprimir pero ahora la nueva población y se obtuvo el mejor de cada población este procesos o se realizaba las generaciones solicitadas por el usuario o hasta que se llegara al valor óptimo que en este caso seria 0. Al final se copiaban los elementos de Q a P para así volver a trabajar con la nueva población en P.

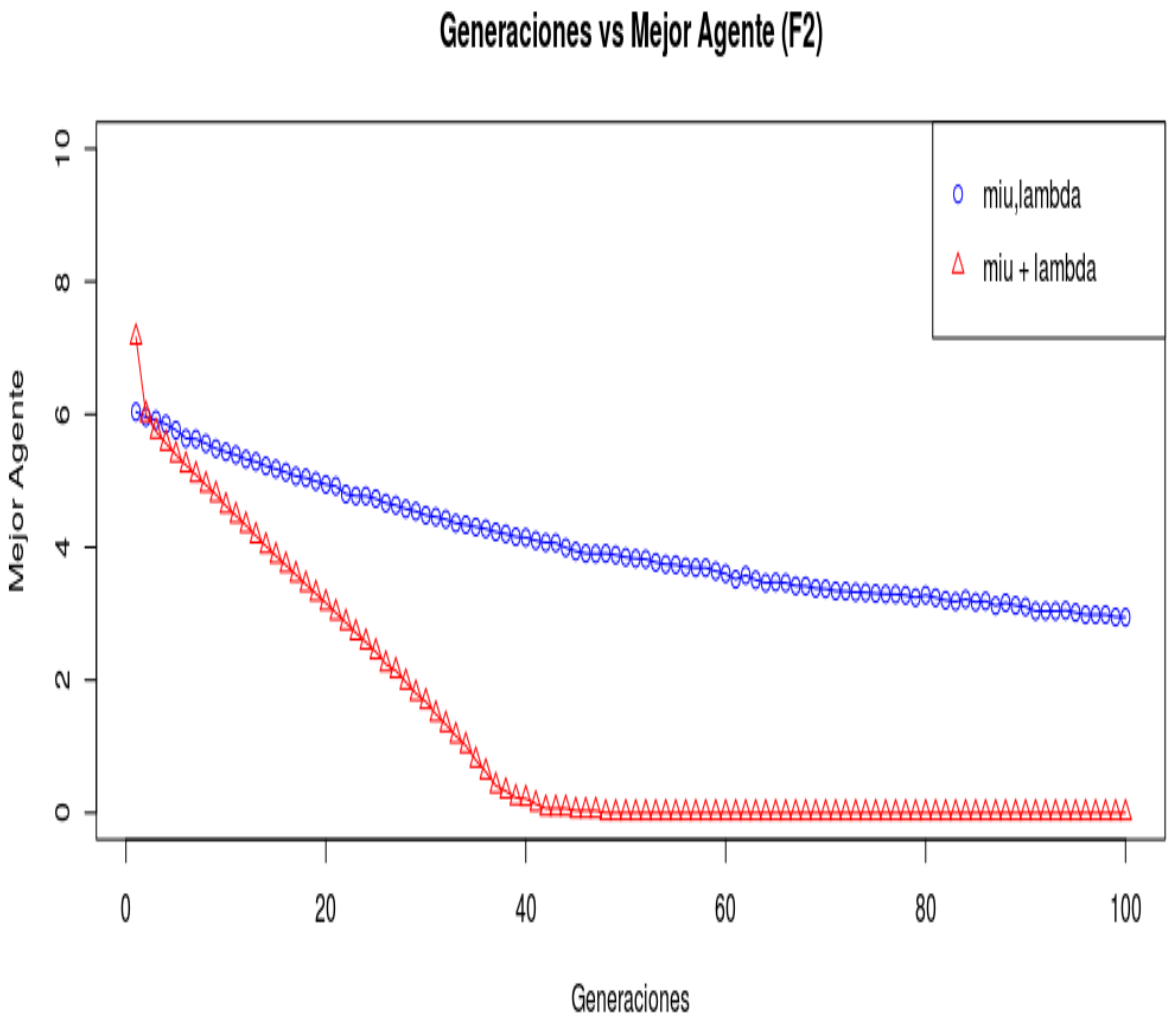
0.3 Algoritmo AG-(miu + lambda)

```
1
2             evaluar_pop(P, N,n);
3             //imprimirPoblacion(P,N,n);
4             obtenerMejor(P,N,0);
5
6             for (int gen=1; gen < max_gen; gen++){ //Miu +
    lambda LISTO
7               //printf("\nGeneration:%d",gen);
8               crossover_pop(P, Q, N,n);
9                 mutacion_pop(Q, N,n);
10                evaluar_pop(Q, N,n);
11            union_pop(M, P, Q, N);
12            qsort(M, 2*N, sizeof(INDIVIDUO), &compareByFitness);
13            seleccionar(P,M,N);
14            //imprimirPoblacion(P,N,n);
15            theBest = obtenerMejor(P,N,gen);
16            if(theBest==0)break;
17          }
18
```

Para el caso del Plus cambiaba considerablemente el algoritmo ya que primero se evaluaba P y se obtenía el mejor, y después en cada iteración o nueva generación se realizaba: La cruza de P y con ello la generación de Q, después se mutaban los elementos de Q, y posteriormente se realizaba su evaluación, y el siguiente paso se diferencia del primer algoritmo ya que después tanto los antiguos Individuos encontrados en P, como los nuevos de Q se juntaban dentro de una nueva Población llamada M, que por ende seria de tamaño 2*N, como la nueva población solo debía ser de tamaño N entonces se seleccionaron a los mejores N elementos de toda la población, para ello se tuvieron que ordenar, después seleccionar a los mejores. Los cuales representarían a los nuevos individuos de la nueva población.

0.4 Gráficas De Convergencia

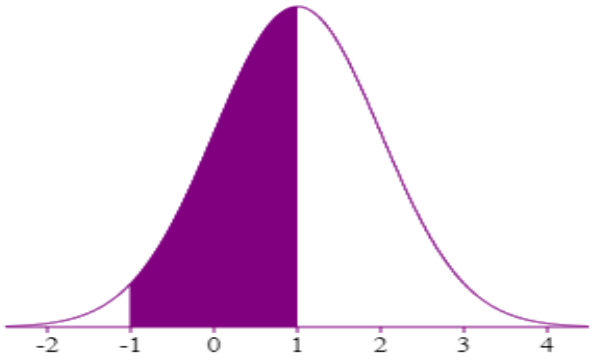


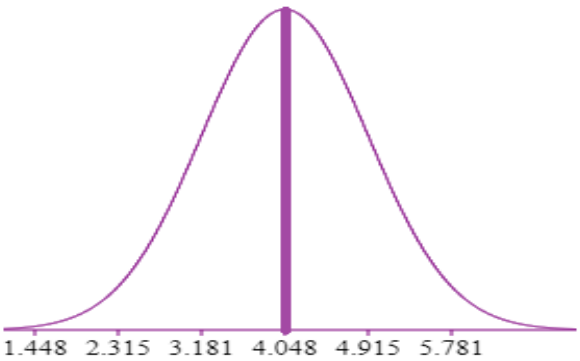

Generaciones vs Mejor Agente (F2)
Mejor Agente
0
2
4
6
8
10
0
20
40
60
80
100
Generaciones
miu,lambda
miu + lambda

0.5 Tabla Comparativa de Resultados

Algoritmo/Problema	M,L - F1	M+L - f1	M.L - F2	M+L - F2
Promedio	4,044776604	1,249177769	4,047984293	1,240108514
Desviación Estándar	0,9672391436	1,90627272	0,8666163613	1,896178939

0.6 Prueba no paramétrica (Test wilcoxon)





La muestra que usamos son los resultados de los mejores valores de aptitud de 30 ejecuciones independientes por algoritmo de GA-(M, L) y GA-(M + L)) en cada uno de los problemas de optimización a evaluar(F1 y F2). Utilizando el test de wilcoxon obtuvimos que un algoritmo fue significativamente mejor que otro con valor significativo menor a 0.05 para cada uno de los problemas de optimización en este caso parece ser mejor (M + L).

0.7 Bibliografía

Mitsuo Gen Runwei Cheng John Wiley. (1997). Genetic Algorithms and Engineering AlfaOmega.