

Índice

Índice	2
Introducción	3
Aeropuerto	3
Solución	3
Vecino	3
Utils	4
Algoritmo Greedy básico	5
Algoritmo de búsqueda local del primer mejor	6
Algoritmo de búsqueda tabú	7
Tablas	10
Semillas	10
Greedy	10
Búsqueda Local	11
Búsqueda Tabú	11
Resultados Globales	12
Análisis	14
Tiempo	14
Costes	14
Algoritmo genético generacional	15
Algoritmo principal	15
Análisis	18
1) Elite 2 cruce OX2 Vs. Elite 2 cruce MOC	18
2) Elite 3 cruce OX2 Vs. Elite 3 cruce MOC	19
El mejor de 1) Vs. el mejor de 2)	19
Comparativa Búsqueda Local, Búsqueda Tabú y Genético	19
Archivos log	21

Introducción

Antes de comentar los algoritmos, existen unas clases sobre las que estos se apoyan y no resulta baladí su presentación.

Aeropuerto

Esta clase almacena la información, leídos de los archivos de datos y que posteriormente usarán los algoritmos. Para desempeñar dicha funcionalidad hace uso de cuatro atributos:

- Matriz de flujos
- Matriz de distancias
- Número de puertas del aeropuerto
- Un valor lógico que indica si el aeropuerto es simétrico o no

Entre sus métodos, destacar su constructor; recibirá el archivo de datos y a partir de él construye los atributos.

Solución

Clase que representa una solución. Será usada por todos los algoritmos, para almacenar de forma más conceptual sus soluciones. Veámos sus atributos:

- Vector de la solución
- Coste de la solución

Vecino

Esta clase representa el intercambio de dos posiciones, realizadas sobre una solución. La existencia de esta clase se justifica por la forma tan sencilla con la que se trabaja con un intercambio, sobre todo a través de sus métodos. Almacenaremos:

- Vector de posiciones

Todos los métodos implementados son para facilitar la lectura en los algoritmos. Como los métodos `setPrimeraPosicion`, `getSegundaPosicion` o `getSegundaPosicion`. Fáciles de leer y definen claramente lo que se esperan de ellas.

Utils

En lugar de clase, se podría definir como una agrupación de funciones comunes entre todos los algoritmos; ya que no tiene ningún atributo. Sus métodos:

- `void escribirFichero(String nombreFichero, String mensaje)`
`void escribirSolucionInicial(Solucion solucionInicial, int iteracion)`: Concatena en una cadena la solución inicial, posteriormente dicha cadena se pasará a la función anterior para escribir el *archivo de logs*.
- `void escribirMovimientoEnFichero(int entorno, Vecino vecino, int coste, int iteraciones)`: Concatena en una cadena el movimiento realizado, así como el coste del mismo, las iteraciones y el entorno en las que se haya. También usado para después escribir en *logs*.
- `void realizarMovimiento(int[] v, Vecino)`: Dato un vector solución le aplica el vecino, es decir, se mueve en el entorno.
- `int calcularCoste(int[] solucion)`: Calcula el coste de una solución, teniendo en cuenta si el aeropuerto es simétrico o no.
- `int calcularCosteParametrizado(Solucion solucion, Vecino vecino)`: Calcula el coste de una solución; teniendo solo el `vecino` que se ha intercambiado. Después se restan al coste total para obtener el coste final asociado.

```
int calcularCosteParametrizado(Solucion solucion, Vecino vecino) {
    costeAntes = 0, costeDespues = 0;
    costeAntes = calcularCosteMovimiento();
    realizarMovimiento();
    costeDespues = calcularCosteMovimiento();
    // Deshacemos el intercambio
    realizarMovimiento();

    return solucion.coste + costeDespues - costeAntes;
}
```

- `int calcularCosteMovimiento(int[] solucion, Vecino vecino)`: Calcula el coste que implica un movimiento, sobre una solución. La forma en la que procede es obteniendo el coste asociado de una puerta con el resto de puertas (tenemos dos puertas intercambiadas, `vecino.getPrimeraPosicion()` y `vecino.getSegundaPosicion()`).
- `Solucion generarSolucionInicial(int tamSolucion)`: Genera una solución inicial factible de manera aleatoria.

```
Solucion generarSolucionInicial(int tamSolucion) {
    solucionInicial <- inicializarVector(tamSolucion);
    posiciones <- creacionVectorPosiciones(tamSolucion);
    tamLogico = tamSolucion;

    while (tamLogico != 0) {
        posAleatoria <- random entre 0 y tamLogico;
        solucionInicial[tamSolucion - tamLogico] <- posiciones[posAleatoria];
        posiciones = posiciones - posAleatoria;
    }
}
```

```

    tamLogico--;
}
solucionInicial.coste = calcularCoste();

return solucionInicial;
}

```

Ésta última función será usada por la Búsqueda Local y la Búsqueda Tabú.

Algoritmo Greedy básico

El algoritmo Greedy está implementado en una clase homónima. Dicha clase tiene las siguientes propiedades:

- Vector sumatorio de flujos
- Vector sumatorio de distancias
- El tamaño de la solución
- La solución

Con sumatorio nos referimos a la suma por filas de las matrices de flujos y distancias creadas en el aeropuerto (leídas del archivo de datos). El resultado suma de cada fila se almacena en sumatorio flujos/distancias dependiendo de la matriz.

En el constructor es el lugar donde se efectúa la suma de filas:

```

Greedy() {
    tamSolucion <- numPuertasDelAeropuerto;
    sumatorioFlujos, sumatorioDistancias <- inicializarVector(tamSolucion);

    for (i desde 0 hasta tamSolucion)
        for (j desde 0 hasta tamSolucion) {
            sumatorioFlujos[i] += aeropuerto.flujos[i][j];
            sumatorioDistancias[i] += aeropuerto.distancias[i][j];
        }
    solucion <- inicializarVector(tamSolucion);
}

```

El enfoque escogido para la creación de una solución viene definida por el siguiente esquema, que a la vez viene implementado en la clase `Greedy`.

```

void algoritmoGreedy() {
    tamLogicoSolucion <- 0;
    do {
        seleccionarMejoresPosiciones();
        eliminarMejores();
        solucion <- mejoresPosiciones
        tamLogicoSolucion++;
    }
}

```

```

}while (tamLogicoSolucion < tamSolucion);

solucion.coste = calcularCoste();
}

```

La parte que resulta más importante comentar es la función `seleccionarMejoresPosiciones()`. Su funcionamiento es, seleccionar el mayor flujo (desde el vector `sumatorioFlujos`) con la mejor distancia (desde el vector `sumatorioDistancia`).

En la función `eliminarMejores()`, marcamos esas dos posiciones con una `MARCA`, -1, para ignorarlas en la función `seleccionarMejoresPosiciones()`.

Algoritmo de búsqueda local del primer mejor

La implementación del algoritmo podemos encontrarlo, al igual que con el algoritmo Greedy, en su clase homónima. Entre sus propiedades:

- La solución actual
- El mejor vecino
- Coste del mejor vecino
- Matriz de los vecinos generados
- El tamaño de la solución

Mencionar que la `solucionActual` será nuestra solución final al problemas, tras terminar la ejecución. La cuestión podría ser la justificación de usar una matriz de `vecinosGenerados`. Pues bien, para evitar la generación de 10 vecinos y que algunos de ellos ya se hayan generado (sólo en esa generación de vecinos), se marca ese vecino con `VECINO_SELECCIONADO`.

Los demás atributos son usados como variables globales de la clase, con motivo de una simplificación de los parámetros a funciones.

El algoritmo principal de la clase sería:

```

void algoritmoBusquedaLocal() {
    iteraciones, intentos, entorno <- 0;
    solucionActual = generarSolucionInicial();

    do {
        generarMejorVecino();
        if (costeMejorVecino < solucionActual.coste) {
            realizarMovimiento();
            solucionActual.coste = costeMejorVecino;
            iteraciones++;
            intentos = 0;
        } else intentos++;
        entorno++;
        vecinosGenerados <- 0;
    } while (entorno < MAX_ENTORNO);
}

```

```

    } while (iteraciones < MAX_ITERACIONES && intentos != MAX_INTENTOS);
}

```

Nota: La generación de los logs se han obviado en el pseudocódigo.

La función más destacada es la generación del mejor vecino. En dicha función se genera 10 vecinos distintos y se almacena el mejor (en los atributos `mejorVecino` y `costeMejorVecino`):

```

void generarMejorVecino(){
    vecinoActual <- Vecino;
    costeMejorVecino = MAX_VALUE;
    costeVecinoActual;

    for (int i = 0; i < NUM_VECINOS; i++) {
        vecinoActual = generarVecino();
        costeVecinoActual = calcularCosteParametrizado();

        if (costeVecinoActual < costeMejorVecino) {
            mejorVecino <- vecinoActual;
            costeMejorVecino = costeVecinoActual;
        }
    }
}

```

```

Vecino generarVecino() {
    nuevoVecino <- Vecino;
    do {
        nuevoVecino.setPrimeraPosicion(Random entre 0 y tamSolucion);
        nuevoVecino.setSegundaPosicion(Random entre 0 y tamSolucion);
        if (nuevoVecino.PrimeraPosicion > nuevoVecino.SegundaPosicion) {
            nuevoVecino.intercambiarPosiciones();
        }
    } while (esVecinoSeleccionado(nuevoVecino) || nuevoVecino.tienePosicionesIguales());
    anadirVecino(nuevoVecino);

    return nuevoVecino;
}

```

El vecino tiene sus posiciones ordenadas para una mejor comparación. Por otro lado, una vez que generamos el vecino, y es factible, añadimos dicho vecino a la matriz de `vecinosGenerados`.

Algoritmo de búsqueda tabú

La búsqueda tabú al implementar una búsqueda local mantiene muchas características comunes. Los nuevos atributos añadidos con respecto a la búsqueda local:

- Una lista para los movimientos tabúes
- Una matriz para la memoria a largo plazo

- Una solución para registrar la mejor solución

La estructura de datos elegida, para el almacenamiento de los vecinos tabúes, es una lista. El esquema del algoritmo principal:

```
void algoritmoTabu() {
    iteraciones, intentos, entorno <- 0;
    solucionActual = generarSolucionInicial();
    mejorSolucion <- solucionActual;
    do {
        generarMejorVecino();
        solucionActual <- realizarMovimiento();
        memoriaLargoPlazo <- mejorVecino;
        listaTabu <- mejorVecino;

        iteraciones++;

        if (solucionActual.coste < mejorSolucion.coste) {
            mejorSolucion <- solucionActual;
            intentos = 0;
        } else intentos++;

        if (intentos == MAX_INTENTOS) {
            calcularEstrategia();
            solucionActual <- generarEntorno();
            intentos = 0;
            entorno++;
        }
    } while (iteraciones < MAX_ITERACIONES);
}
```

Los métodos que requieren atención son `calcularEstrategia()` y `generarEntorno()` que es donde se elige si diversificar o intensificar y dónde se genera una solución a partir de la memoria a largo plazo, respectivamente.

```
void calcularEstrategia() {
    double probabilidad = Random(entre 0 y 1);
    DIVERSIFICAR = probabilidad < 0.5;
}
```

La variable booleana `DIVERSIFICAR`, es una variable global, y su uso es el de decidir entre:

```
DIVERSIFICAR = true then diversificamos
DIVERSIFICAR = false then intensificamos
```

Para finalizar, comentar los puntos clave de la función `generarEntorno()`, la función que tendrá en cuenta que tipo de oscilación se está produciendo y generará una solución factible:

```

private void generarEntorno() {
    nuevoEntorno <- Solucion
    mejorValor, posicionMejor, esMejor;

    for (i desde 0 hasta dimensionMemoriaLargoPlazo) {
        mejorValor = reiniciarMejorValor();
        posicionMejor = 0;
        for (j desde 0 hasta dimensionMemoriaLargoPlazo) {
            if (i != j) {
                esMejor = calcularSiEsMejor(i, j, mejorValor);

                if (esMejor && si j no está en nuevoEntorno)) {
                    mejorValor = memoriaLargoPlazo[i][j];
                    posicionMejor = j;
                }
            }
        }
        nuevoEntorno.solucion[i] <- posicionMejor;
    }
    nuevoEntorno.coste = calcularCoste();
}

```

Para saber si el valor contenido en la posición actual de la matriz `memoriaLargoPlazo`, es el mejor, tenemos:

```

boolean calcularSiEsMejor(int fila, int columna, int mejorValor) {
    if (DIVERSIFICAR) return memoriaLargoPlazo[fila][columna] < mejorValor;
    else return memoriaLargoPlazo[fila][columna] > mejorValor;
}

```

Vemos que dicha función tiene en cuenta si estamos diversificando o intensificando. De esta manera podemos comprobar si estamos buscando un valor menor o mayor que el `mejorValor` dentro de la `memoriaLargoPlazo`.

Tablas

Para mejorar la visualización de los resultados se ha decidido mostrar los resultados por Aeropuertos, para Málaga y Madrid.

Semillas

Se muestra las semillas utilizadas en cada ejecución.

Ejecución	Semilla
1	70931043
2	23456781
3	34567812
4	45678123
5	56781234

Greedy

Greedy	Madrid01		Madrid02		Madrid03		Madrid04	
	S	T(s)	S	T(s)	S	T(s)	S	T(s)
Ejec	3148	0	3130	0	4584	0	4438	0
Media	22.49%	0	28.38%	0	27.47%	0	18.54%	0

Greedy	Malaga01		Malaga02		Malaga03		Malaga04	
	S	T(s)	S	T(s)	S	T(s)	S	T(s)
Ejec	3941	0	33954	0	13806	0	191745	0
Media	7.01%		25.40%		4.77%		26.63%	

Búsqueda Local

Búsqueda Local	Madrid01		Madrid02		Madrid03		Madrid04	
	S	T(s)	S	T(s)	S	T(s)	S	T(s)
Ejec1	2676	0	2552	0	3774	0	3888	0
Ejec2	2656	0	2570	0	3750	0	3806	0
Ejec3	2674	0	2638	0	3820	0	3900	0
Ejec4	2680	0	2524	0	3690	0	3870	0
Ejec5	2652	0	2576	0	3782	0	3882	0
Media	3.80%	0	5.50%	0	4.65%	0	3.34%	0
Desv. típica	0.49%	0	1.73%	0	1.34%	0	0.99%	0

Búsqueda Local	Malaga01		Malaga02		Malaga03		Malaga04	
	S	T(s)	S	T(s)	S	T(s)	S	T(s)
Ejec1	3784	0	31196	0	13418	0	177452	0
Ejec2	3876	0	30849	0	13433	0	176760	0
Ejec3	3809	0	31019	0	13441	0	176081	0
Ejec4	3813	0	31385	0	13412	0	179591	0
Ejec5	3792	0	31432	0	13421	0	178026	0
Media	3.58%		15.14%		1.87%		17.27%	
Desv. típica	0,98%		0.91%		0.09%		0.88%	

Búsqueda Tabú

Búsqueda Tabú	Madrid01		Madrid02		Madrid03		Madrid04	
	S	(s)	S	T(s)	S	T(s)	S	T(s)
Ejec1	2570	4	2438	2	3602	2	3744	3

Ejec2	2570	4	2442	2	3604	2	3744	3
Ejec3	2574	2	2442	2	3596	2	3744	5
Ejec4	2570	2	2438	2	3596	2	3744	3
Ejec5	2570	3	2448	2	3602	2	3744	3
Media	0,03%	3	0,15%	2	0,11%	2	0%	3,40
Desv. típica	0.07%	1	0,17%	0	0,10%	0	0%	0,89

Búsqueda Tabú	Malaga01		Malaga02		Malaga03		Malaga04	
	S	T(s)	S	T(s)	S	T(s)	S	T(s)
Ejec1	3722	5	27076	6	13342	5	151426	8
Ejec2	3683	5	27076	7	13348	4	151426	9
Ejec3	3683	6	27076	7	13352	4	151426	8
Ejec4	3683	5	27076	7	13354	4	151426	8
Ejec5	3683	7	27076	7	13373	4	151426	9
Media	0,21%	5,60	0%	6,80	1.33%	4,20	0%	8,40
Desv. típica	0,47%	0,89	0%	0,45	0.09%	0,45	0%	0,55

Resultados Globales

	Madrid01		Madrid02		Madrid03		Madrid04	
	S	T(s)	S	T(s)	S	T(s)	S	T(s)
Greedy	22,49 (0)	0 (0)	28,38 (0)	0 (0)	27,47 (0)	0 (0)	18,54 (0)	0 (0)
Búsqueda Local	3,80 (0,49)	0 (0)	5,50 (1,73)	0 (0)	4,65 (1,34)	0 (0)	3,34 (0,99)	0 (0)
Búsqueda Tabú	0,03(0,07)	3(1)	0,15(0,17)	2(0)	0,11(0,10)	2(0)	0(0)	3,40(0,89)
Coste Mejor Solución	Tabú 2570		Tabú 2438		Tabú 3596		Tabú 3744	

	Malaga01		Malaga02		Malaga03		Malaga04	
	S	T(s)	S	T(s)	S	T(s)	S	T(s)
Greedy	7,01 (0)	0 (0)	25,40 (0)	0 (0)	4,77 (0)	0 (0)	26,63 (0)	0 (0)
Búsqueda Local	3,58 (0,98)	0 (0)	15,14 (0,91)	0 (0)	1,87 (0,09)	0 (0)	17,27 (0,88)	0 (0)
Búsqueda Tabú	0,21(0,47)	5,60(0,89)	0(0)	6,80(0,45)	1,33(0,09)	1,33(0,09)	0(0)	8,40(0,55)
Coste Mejor Solución	Tabú 3683		Tabú 27076		Tabú 13342		Tabú 151426	

	Media	
	S	T(s)
Greedy	20.08 (0)	0 (0)
Búsqueda Local	6.89 (0,926)	0 (0)
Búsqueda Tabú	3.59 (0,626)	121.5 (13,59)
Coste Mejor Solución	Tabú 3719	

Observando la tabla de medias, la desviación típica menor es el algoritmo de la búsqueda tabú. Podemos matizar que es más robusto que el de la búsqueda local porque al ser menor el valor obtiene soluciones más cercanas al óptimo global.

Análisis

Tiempo

¿Por qué el tiempo de cómputo es diferente entre búsqueda local y la búsqueda tabú?

El algoritmo de búsqueda local una vez llega al punto de estancamiento, es decir, una vez que ha generado a 100 mejores vecino, y ninguno mejora a la situación actual, el algoritmo se detiene. Éste algoritmo nunca llega a las 50000 iteraciones, por tanto su límite está en el número de intentos.

Esta es la principal diferencia con el algoritmo de búsqueda tabú, en el que una vez que se queda estancado sigue explorando soluciones desde otro entorno, bien intensifica o diversifica, finalizando la ejecución cuando ha completado las 50000 iteraciones, y no antes.

Costes

¿Cuándo se aplica la oscilación estratégica con más intensidad?

Al principio de la ejecución del algoritmo, el entorno se incrementa muy lentamente porque la solución mejor de mejores se actualiza con mucha facilidad, es decir, está encontrando mejores soluciones en un entorno donde aún no se sabe cuál es el óptimo local.

Cuanto más nos movemos más difícil es mejorar al mejor de mejores y se va produciendo estancamiento con frecuencia; por ende la generación de entornos es más frecuente y es más difícil encontrar una mejor solución que mejore a la mejor de mejores.

Observando los resultados obtenidos en las tablas, se puede observar que el algoritmo que encuentra el mejor coste, el más cercano a la solución global, es el algoritmo de Búsqueda Tabú, aunque el tiempo de cómputo es mayor.

¿Por qué funciona mejor el algoritmo tabú que el local del mejor?

La clave son los entornos. En el algoritmo de búsqueda local generamos una solución inicial y a partir de ella nos movemos en el espacio de soluciones. Cabe puntualizar que todos estos movimientos en el entorno, es eso: movimiento en un único entorno.

El algoritmo de búsqueda local plantea la problemática de quedarse en óptimo locales.

En contraposición, el algoritmo de búsqueda local escapa de esos óptimos locales permitiendo movimientos de empeoramiento. Además, gracias al salto a un nuevo entorno permite escapar de un entorno demasiado explotado, y en el que no se está encontrando mejores soluciones que la mejor de mejores.

Este simple mecanismo nos devolverá mejores soluciones, para la gran mayoría de los casos, que la búsqueda local.

Algoritmo genético generacional

La implementación del algoritmo se encuentra en la clase llamada `Genetico`. Atributos:

- Poblacion `poblacion`
- Poblacion `poblacionDescendiente`
- Entero `TAM_POBLACION`

La clase población a continuación se presentará, pero para explicar de manera escueta su lógica, decir que encapsula toda la información de una población.

Para dicho algoritmo se han implementado a su vez dos clases de apoyo:

- **Poblacion**: Representa las características de una población. Propiedades:
 - Solucion[] `individuos`
 - Vector `posElites`
 - Vector `costeElites`
 - Vector `posPeores`
 - Vector `costePeores`
- **Reproducción**: Implementa los dos tipos de cruces que se realizan sobre dos individuos. Atributos:
 - Solucion[] `progenitores`
 - Entero `TAM_PROGENITORES`
 - Entero `POS_MADRE`, `POS_PADRE`, `NUM_PROGENITORES`
 - Matriz de enteros `posMarcadasOX2`
 - Matriz de booleanos `posMarcadasMOC`

La clase **Población** se encarga de gestionar toda la información relativa a una población. Por medio de sus métodos, entre los que podemos encontrar `calcularElites()`, `evaluar()`, etc: manejaremos de una forma lógica toda una población.

La clase Reproducción será la responsable de realizar el cruce *OX2* y *MOC*. Más adelante veremos cómo funcionan.

Algoritmo principal

Pseudocódigo:

```
void algoritmoGenetico() {
    iteraciones <- 0
    generacion <- 0
    NUM_EVALUACIONES <- 5000;
    inicializarPoblacion()
    poblacion <- evaluar()
    while (iteraciones < NUM_EVALUACIONES) {
        crearPoblacionDescendiente()
        seleccionar()
```

```

    recombinar()
    mutar()
    poblacion <- calcularElites()
    poblacionDescendiente <- evaluar()
    reemplazar()
    generacion++;
}
}

```

NOTA: La escritura en logs se obvia en la representación de pseudocódigo

Antes de comenzar con las iteraciones, debemos de inicializar la población, y eso es precisamente lo que se hace al `inicializarPoblacion()`. Dicha población se ha creado con todos los individuos de manera aleatoria. Después hay que evaluar dicha población, ya que tan sólo la hemos creado.

Comenzamos nuestras iteraciones hasta que alcancemos las 50000 evaluaciones de individuos. Empezamos por crear una población de descendientes vacía (`poblacionDescendiente`), para después ir rellenandola con selecciones de la población anterior (`poblacion`). La selección se realiza por torneo binario de dos individuos:

```

private void seleccionar() {
    posIndividuoGanador
    for (i hasta tamaPoblacion) {
        posIndividuoGanador = torneoBinario();
        poblacionDescendiente.incluirIndividuo(poblacion.individuos[posIndividuoGanador]);
    }
}

```

```

private int torneoBinario() {
    posPrimerIndividuo <- Random(tamPoblacion);
    posSegundoIndividuo <- generarPosDistintaPrimerIndividuo()
    return mejorIndividuo(posPrimerIndividuo, posSegundoIndividuo)
}

```

Una vez que tenemos la población de descendientes completa procedemos a cruzar individuos de acuerdo con los parámetros suministrados por el usuario, bien cruce *OX2* o *MOC*.

Los dos tipos de cruce implementados en pseudocódigo quedarían tal que así:

```

void cruceMOC() {
    puntoDeCorte <- Random(entre 1 y TAMINDIVIDUO - 1)
    posMarcadasMOC <- new boolean[NUM_PROGENITORES][TAM_PROGENITORES];
    eliminacionCruzada()
    rellenarValoresCruzados()
}

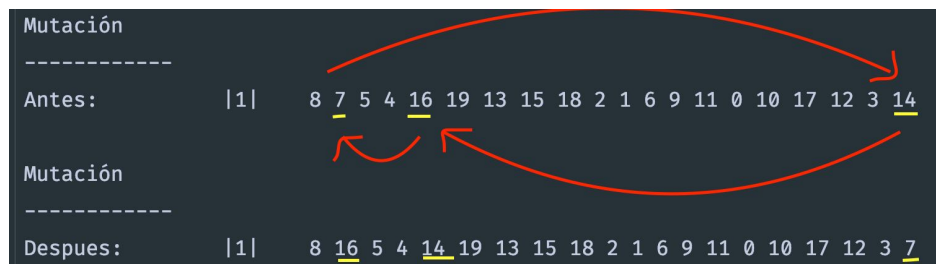
```

```

void cruceOX2() {
    progenitorOrigen <- 0
    crearCopiaProgenitores()
    while (progenitorOrigen < NUM_PROGENITORES) {
        seleccionAleatoria()
        eliminacionProgenitorDestino()
        rellenarValores()
        progenitorOrigen++
    }
}

```

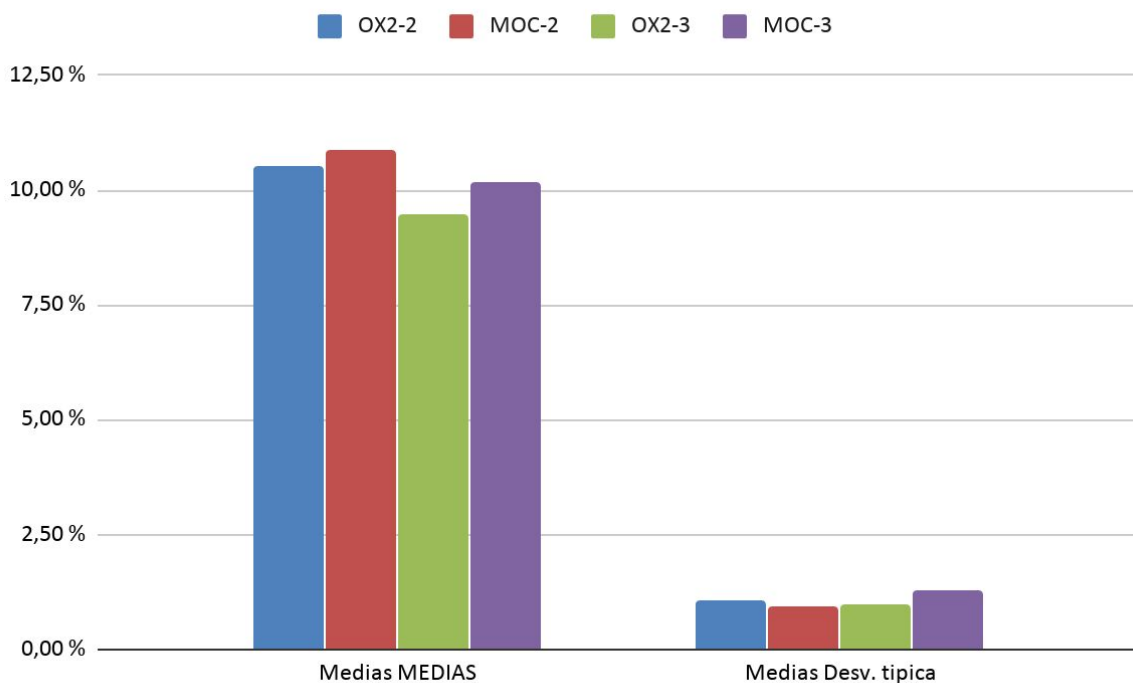
Después de cruzar los individuos (algunos de ellos de acuerdo a la probabilidad 70%). Procedemos a mutar cada posición de un individuo. En caso de que una posición haya que mutarla, se realiza una selección de otras dos posiciones (completamente aleatoria y distintas) y se intercambia de la siguiente forma:



Estamos listos para sustituir la población con la población de descendientes, pero antes debemos comprobar si los elites están en la población de descendientes. Calculamos los elites de la población anterior y vemos cuántos de ellos están en la población de descendiente. Los elites que no están, se sustituyen por los peores de la población de descendientes, es decir, si tenemos un elitismo de tres individuos y tan sólo hay un elite ubicado en la población de descendientes, los dos elites que no se encuentran en la población de descendientes se insertan por los dos peores.

Análisis

1) Elite 2 cruce OX2 Vs. Elite 2 cruce MOC



Según estos resultados podemos concluir que el operador de cruce OX2 obtiene mejor resultado que el cruce MOC, es decir, obtiene resultados más factibles y cercanos al mejor conocido, aunque no es una diferencia muy significativa.

Analizando podemos decir que cuando se genera un punto de corte en el cruce MOC puede ocurrir casos en los que el punto se encuentra muy cerca del tamaño del individuo. Y por tanto los cambios producidos son mínimos (sólo se modifica un número muy pequeño de posiciones para el descendiente). Esto quiere decir que se hace una menor exploración del entorno, que con el operador OX2.

El operador OX2 presenta un mejor desempeño ya que realiza una exploración más amplia del entorno. Esto viene reflejado en el propio funcionamiento del cruce, al realizar una modificación aleatoria de cada posición. En media, se realizarán un gran número de intercambios aleatorios y llegaremos a posiciones no exploradas. El individuo irá moviéndose generación tras generación a posiciones más prometedoras, arrastrando elitismo.

2) Elite 3 cruce OX2 Vs. Elite 3 cruce MOC

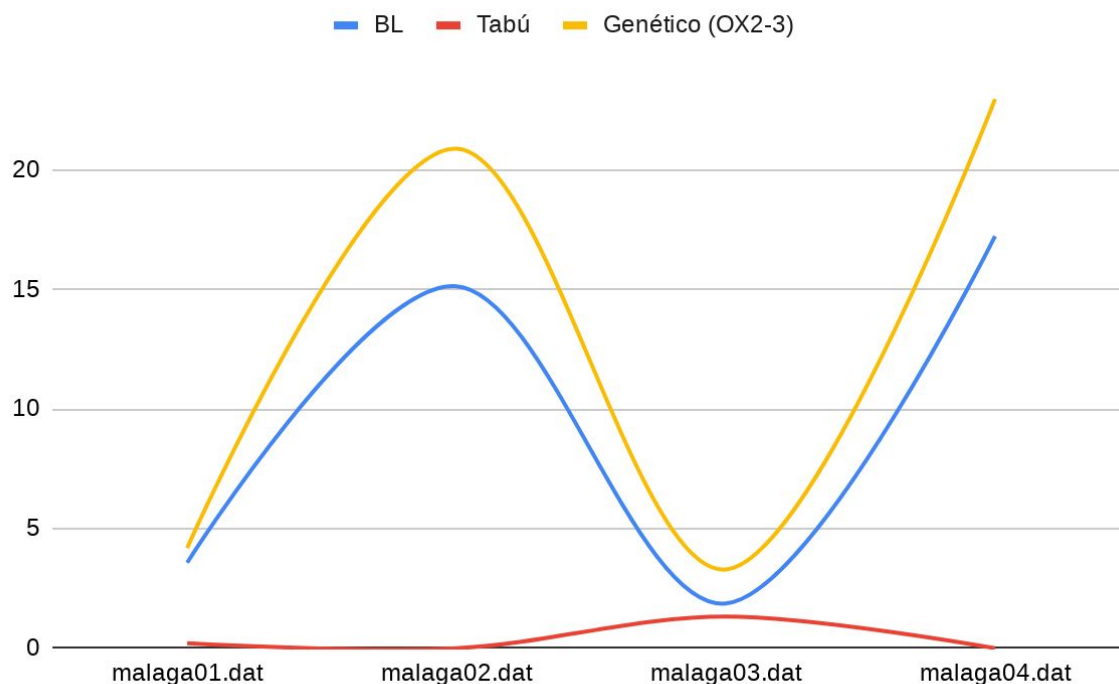
Se produce el mismo comportamiento que en el análisis 1, porque el cruce OX2 produce mejores resultados.

El mejor de 1) Vs. el mejor de 2)

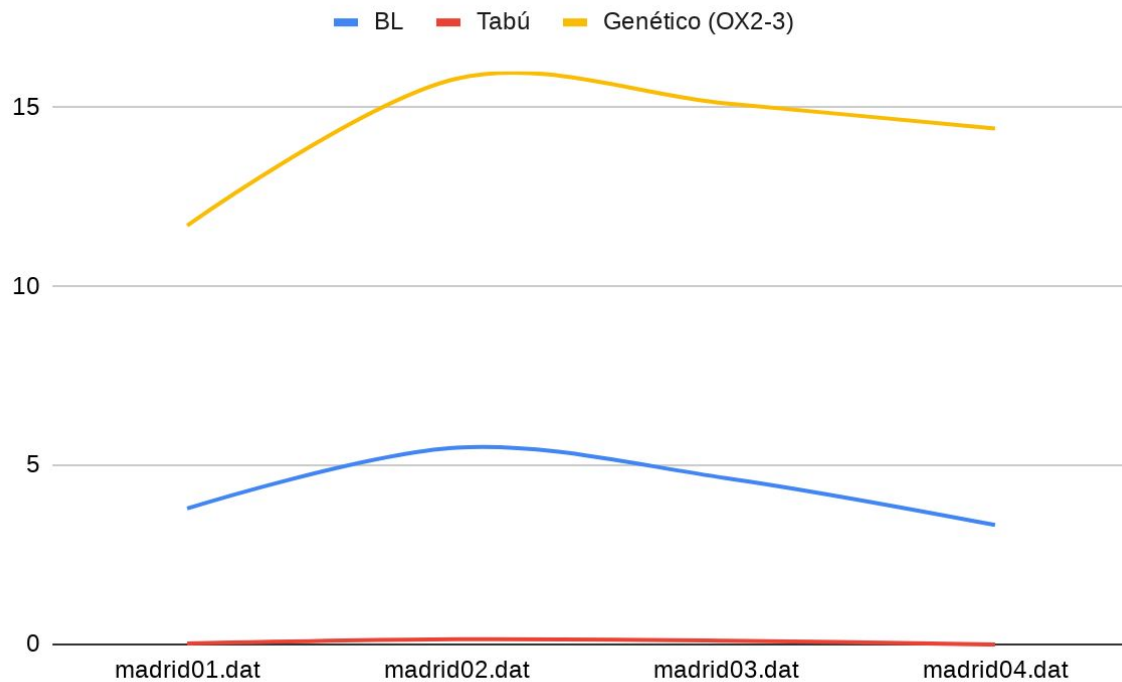
Según los resultado de la tabla podemos decir que al aumentar el número de élites en una unidad, se obtiene mejores resultados que con 2 elites. Al introducir un elite más se propagan los genotipos en las siguientes generaciones, es decir, en los cruces de la nueva generación se aprovechará la buena carga genética de los élites: produciendo a su vez mejores individuos.

Los élites juegan un papel muy importante en la mejora de resultados obtenidos. Comparando el OX2 con elitismo de 2 individuos y el OX2 con elitismo de 3, podemos concluir que el que mejor resultado obtiene es el OX2-3 elites, debido a que el mejor cruce es el OX2 y el aumento de élites hace que se compartan mayor genes entre los individuos.

Comparativa Búsqueda Local, Búsqueda Tabú y Genético



Mostramos una comparativa de los tres algoritmos implementados hasta ahora. Podemos afirmar que el algoritmo con mejor resultados hasta el momento es la búsqueda Tabú, porque los resultados son más cercanos al coste global. Por otra parte, el peor es el Genético porque está constantemente diversificando y la convergencia es demasiado lenta.



Con respecto a la búsqueda local, la convergencia de ésta hace que encuentre soluciones más cercanas a la óptima. Mientras que el genético gasta demasiado tiempo buscando en soluciones del entorno no prometedoras. Es un proceso muy aleatorio.