# MANUAL DE REFERENCIA DEL SISTEMA PARA LA DISTRITACIÓN ELECTORAL 2016

**INSTITUTO NACIONAL ELECTORAL**

# MANUAL DE REFERENCIA DEL SISTEMA PARA LA DISTRITACIÓN ELECTORAL 2016

**INSTITUTO NACIONAL ELECTORAL**

**12 de abril del 2016**

# Índice general

## CAPÍTULO 1

# DIAGRAMA DEL ALGORITMO BASADO EN RECOCIDO SIMULADO

El objetivo del algoritmo basado en Recocido Simulado es generar $r$ distritos conexos, con las unidades geográficas que forman cada entidad federativa, de tal forma que se respeten los criterios de equilibrio poblacional, compacidad geométrica.

Una solución es representada mediante un vector: $Distritos\_Actuales[x_1, x_2, ..., x_n]$, donde la *i-ésima* entrada representa a la unidad geográfica $i$. La variable $x_i$ toma valores entre 1 y $r$, que corresponden al distrito al cual es asignada la unidad geográfica $i$.

En la Figura 1.1 se presenta un diagrama de bloques del funcionamiento del algoritmo basado en Recocido Simulado. Al inicio se obtiene información, dada por el usuario, de los parámetros que deberán emplearse y datos sobre las unidades geográficas, disponibles en archivos de texto. Posteriormente se construye de forma aleatoria una solución inicial, y se evalúa su costo. A partir de este momento se inicia un ciclo que dura hasta que se alcanza el valor de la variable $TemperaturaFinal$. Durante esta etapa se realizan modificaciones en los distritos de la solución generada por el algoritmo, se evalúa el costo de estas modifica-

ciones, y se emplea el criterio de Metrópolis para guiar el proceso de búsqueda, mediante aceptaciones o rechazos probabilísticos.

Al alcanzar la temperatura final, se realiza un búsqueda local en un vecindario de la mejor solución encontrada. Si hay mejoras, la solución es actualizada. Al terminar esta exploración, la solución se devuelve al usuario como el mejor escenario visitado.

Es importante destacar que cada una de las unidades geográficas ha sido previamente asignada a un conjunto territorial mediante la tipología de cada entidad federativa. De esta forma, cada conjunto está formado por un conjunto de unidades geográficas, y en él deben construirse un número preestablecido de distritos. Por lo anterior, el algoritmo fue diseñado para realizar en cada conjunto territorial una distritación electoral independiente del resto del estado. Al terminar con todos los conjuntos se obtiene la distritación electoral del estado.

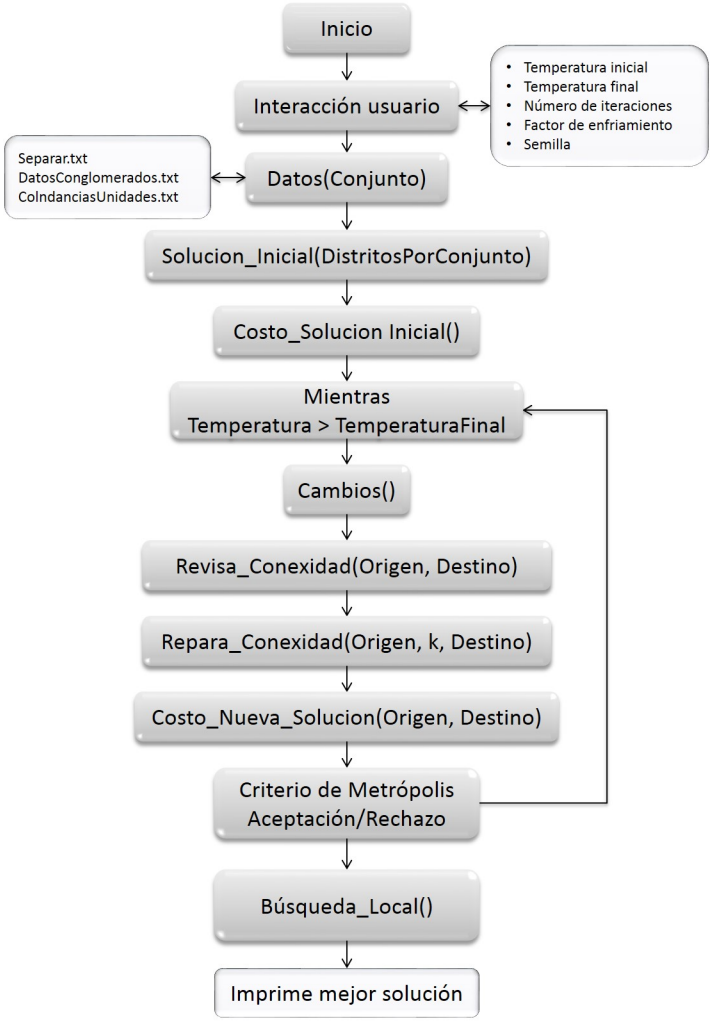


**Figura 1.1** Diagrama de bloques del algoritmo basado en Recocido Simulado.

# CAPÍTULO 2

# DESCRIPCIÓN TÉCNICA DEL ALGORITMO BASADO EN RECOCIDO SIMULADO

En este capítulo se describe la forma en que operan cada una de las funciones del algoritmo basado en recocido simulado empleado en el sistema de distritación electoral 2016. Primero se presentan las variables globales empleadas en el algoritmo, junto con una descripción breve del uso que se hace de ellas, en las secciones restantes se describen las funciones usadas por el algoritmo.

## 2.1. Variables globales

En esta sección, en la Tabla 1.1, se presentan las variables globales más importantes empleadas por este algoritmo. En la primera columna se coloca el nombre de la variable, la segunda columna indica el tipo de variable empleado, y en la tercera columna se hace una pequeña descripción del uso que se le da durante la ejecución del algoritmo.

| Variable | Tipo | Descripción |
|---|---|---|
| Distritos_Actuales<br>Mejores_Distritos | **int[]** | Son arreglos con la solución actual y la mejor solución visitada por el algoritmo hasta el momento. |
| Compacidad_Nueva<br>DesviacionPoblacional_Nueva | **double** | Indican el costo de compacidad y desviación poblacional de la nueva solución, construida como vecina de la solución actual. |
| PoblacionDistrito_Origen<br>PoblacionDistrito_Destino<br>PoblacionDistritos_Actuales | **int** | Indican el número de habitantes en los distritos que se modifican al construir una solución nueva. |
| PoblacionDistritos_Actuales | **int[]** | Indica el número de habitantes en los distritos de la solución actual. |
| DesviacionPoblacional_Origen<br>DesviacionPoblacional_Destino<br>AreaDistrito_Origen<br>AreaDistrito_ Destino<br>PerimetroDistrito_Origen<br>PerimetroDistrito_Destino<br>CompacidadDistrito_Origen<br>CompacidadDistrito_Destino | **double** | Indican los costos de los distritos que se modifican al construir una solución nueva. |
| MedidaArea<br>MedidaPerimetro | **double[]** | Indican el área y perímetro de los distritos en la solución actual. |
| PerimetroFrontera | **double[][]** | Indica el perímetro compartido por unidades geográficas vecinas. |
| DesviacionPoblacional_Actual<br>Compacidad_Actual | **double** | Indican el costo del escenario actual, es la suma de los costos de los distritos. |

| | | |
|---|---|---|
| DesviacionPoblacionalZonas_Actuales<br>CompacidadDistritos_Actuales | **double[]** | Indican el costo de los distritos en la solución actual. |
| PoblacionUnidadGeografica | **int[]** | Guarda la cantidad de habitantes en cada unidad geográfica. |
| AreaUnidadGeografica | **double[]** | Guarda el área de cada unidad geográfica. |
| Vecinos | **int[][]** | Indica las unidades geográficas colindantes. |
| Semilla | **long** | Guarda el valor de la semilla propuesta por el usuario. |
| Unidades_Cambiadas | **int[]** | Guarda el indicador de las unidades geográficas que se han cambiado para generar una solución nueva. |
| Distrito_Destino<br>Distrito_Origen | **int** | Indican los distritos que son modificados para generar una solución nueva. |
| ConjuntoActual<br>UnidadesPorConjunto<br>NDistritos | **int** | Indican el conjunto territorial que se está optimizando, el número de unidades geográficas que lo forman y el número de distritos que deben generarse en él. |
| Conversion | **int[]** | Asigna un identificador a cada unidad geográfica, que se usará durante el proceso de optimización. |
| DistritosFinales | **int[]** | Guarda la solución final generada por el algoritmo. |
| MediaEstatal | **double[]** | Guarda la media poblacional. |
| ConjuntosTotales | **int[]** | Guarda el número de distritos que deben construirse. |

**Tabla 2.1** Variables globales.

En las siguientes secciones se presentan y describen las funciones más importantes empleadas por el algoritmo basado en recocido simulado.

### 2.2. Función main()

La función main() inicia con la asignación de valores para algunas de las variables, tanto globales como locales, empleadas durante la ejecución del algoritmo. Las variables locales más importantes de esta función se presentan en la Tabla 2.2.

Nuevamente, en la primera columna se coloca el nombre de la variable, la segunda columna indica el tipo de variable empleado, y en la tercera columna se hace una pequeña descripción del uso que se le da durante la ejecución del algoritmo.

| Variable | Tipo | Descripción |
|---|---|---|
| Temperatura_Usuario<br>Temperatura<br>TemperaturaFinal | **double** | Indican la temperatura inicial dada por el usuario, la temperatura actual del sistema y la temperatura en la que termina la ejecución del algoritmo, respectivamente. |
| EquilibrioFinal | **int** | Determina el número de iteraciones que debe hacer el algoritmo en cada temperatura. |
| Numero_de_Semillas | **int** | Cuenta el número de semillas en el semillero para indicar el número de corridas que deben realizarse. Una corrida por cada semilla. Cuando el usuario da una semilla esta variable toma el valor de 1. |
| Semillas | **int[]** | Guarda hasta 1000 semillas obtenidas del semillero o bien la semilla dada por el usuario. |
| Distritos_Por_Conjunto | **int[]** | Indica el número de distritos que deben construirse en cada conjunto territorial. |

| | | |
|---|---|---|
| Entrada<br>Aceptada | **double** | Cuentan el número de veces que el algoritmo visita una solución de peor calidad y el número de veces que la acepta, respectivamente. El cociente $Aceptada/Entrada$ es conocido como el nivel de aceptación del algoritmo. |
| Numero_de_Conjuntos | **int** | Indica el número de conjuntos territoriales en el estado. |
| MenorCostoPoblacional<br>MenorCompacidad | **double** | Guardan el costo de la mejor solución visitada por el algoritmo durante una corrida. |
| Iteraciones_Caliente<br>Iteraciones _Templado<br>Iteraciones_Frio | **int** | Indican el número de iteraciones que se realizará a diferentes niveles de aceptación. |
| FactorEnfriamiento_Caliente<br>FactorEnfriamiento_Templado<br>FactorEnfriamiento_Frio | **double** | Indican el factor de enfriamiento a diferentes niveles de aceptación del algoritmo. |
| DesviacionPoblacional_EscenarioFinal<br>Compacidad_EscenarioFinal | **double[]** | Guardan el costo de cada uno de los distritos construidos por el algoritmo. |
| Poblacion_EscenarioFinal | **int[]** | Guarda el número de habitantes. |
| Mejor_Escenario | **int[]** | Guarda la mejor solución encontrada por el algoritmo. Al terminar cada corrida se actualiza en caso haber encontrado una mejor solución. |
| CostoPoblacional_MejorEscenario<br>Compacidad_MejorEscenario | **double** | Guardan los costos del mejor escenario construido por el algoritmo, |
| Solucion_Hibrida | **int[]** | Guarda la mejor solución para cada conjunto territorial encontrada por el algoritmo. |
| DesviacionPoblacional_Hibrida<br>Compacidad_Hibrida | **double[]** | Guardan los costos de la solución híbrida construida por el algoritmo. |

**Tabla 2.2** Variables locales de la función main.

La función inicia asignando el valor de los parámetros que el algoritmo empleará durante su ejecución:

- Temperatura_Usuario
- TemperaturaFinal
- FactorEnfriamiento_Caliente
- FactorEnfriamiento_Templado
- FactorEnfriamiento_Frio
- Iteraciones_Caliente
- Iteraciones_Templado
- Iteraciones_Frio

El uso de los factores Caliente, Templado y Frío dependerá del nivel de aceptación, el cual se calcula como el cociente del número de soluciones de mala calidad aceptadas entre el número de soluciones de mala calidad visitadas, $Aceptada/Entrada$.

Los factores Calientes se usan cuando el nivel de aceptación es mayor que 0.60, los factores Templados se usan cuando el nivel de aceptación está entre 0.40 y 0.60, y los factores Fríos se emplean cuando el nivel de aceptación es menor que 0.40.

Posteriormente se revisa el valor asignado a la variable Semilla. Si el usuario coloca un valor para esta variable, se realizará una corrida empleando esta semilla para iniciar el generador de números aleatorios. Si el usuario elige la opción semillero se lee el archivo de texto Semillero.txt, y se realiza una corrida por cada una de las semillas encontradas en el archivo. Es importante destacar que el algoritmo está diseñado para leer hasta 1000 semillas. En caso de que el número de semillas en el archivo Semillero.txt sea mayor, sólo se considerarán las primeras 1000 semillas.

El siguiente paso consiste en realizar la lectura del archivo de texto ConjuntosDistritos.txt para determinar cuántos distritos se deberán crear en cada conjunto territorial, y se llama a la función Datos(int Conjunto) para obtener la información de cada unidad geográfica y poder emplearla durante el resto de la ejecución.

Cuando se ha obtenido la información de cada unidad geográfica, se inicia la construcción y optimización de distritos para cada conjunto territorial por separado.

Para cada conjunto territorial se repiten los siguientes pasos.

**RS1** Se crea, de forma aleatoria, una solución inicial y se evalúa su costo mediante el uso de las funciones Solucion_Inicial(int DistritosPorConjunto) y Costo_Inicial() respectivamente.

**RS2** Se inicia el proceso de mejora mediante un ciclo que durará hasta que la temperatura llegue a la temperatura final. Durante este ciclo se realizan los siguientes pasos:

- **RS2.1** La solución actual es modificada para crear una solución nueva, mediante la función Cambios.
- **RS2.2** La solución nueva es evaluada, mediante la función Costo_Nueva_Solucion(int Origen, int Destino).
- **RS2.3** Se determina si la solución actual es reemplazada por la solución nueva mediante el criterio de Metrópolis.

**RS3** La mejor solución encontrada es guardada en memoria.

Cuando cada uno de los conjuntos territoriales han sido procesados mediante los pasos **RS1**-**RS3** se da por concluida una corrida del algoritmo. Cuando el algoritmo alcanza la temperatura final indicada por el usuario, se realiza una exploración en el vecindario de la mejor solución encontrada, mediante el uso de la función Busqueda_Local. La unión de las soluciones obtenidas para cada conjunto territorial se convierte en el escenario final. Es importante insistir en que se realizará una corrida por cada semilla dada al algoritmo.

Si el usuario propuso el valor para la variable Semilla, sólo se genera un escenario que es devuelto después de la primera corrida. Si el usuario eligió la opción Semillero, entonces se generarán tantos escenarios como semillas se tengan. Además, al concluir todas las corridas se devolverá la mejor distritación encontrada al combinar los mejores distritos para cada conjunto territorial, de cada uno de los escenarios finales obtenidos.

El pseudocódigo de la función main se presenta en el Algoritmo 21.

**Algoritmo 1:** Pseudocódigo de la función main

```
1  Solicitar valores de parámetros al usuario.
2  Leer el archivo ConjuntosDistritos.txt, y llamar a la función Datos(Conjunto).
3  Para cada conjunto territorial hacer
4      Crear solución inicial con la función Solucion_Inicial(DistritosPorConjunto).
5      Evaluar la calidad de la solución inicial con la función Costo_Inicial().
6      Mientras Temperatura > TemperaturaFinal hacer
7          Modificar Distritos_Actuales para obtener una solución nueva mediante la función Cambios().
8          Evaluar la calidad de la solución nueva mediante la función Costo_Nueva_Solucion(Origen, Destino).
9          Usar el criterio de Metrópolis para determinar si la solución nueva reemplaza a Distritos_Actuales.
10     fin
11     Guardar en memoria la mejor solución encontrada.
12 fin
13 Juntar las soluciones encontradas para cada conjunto y devolverlas como mejor distritación encontrada para el
   estado.
```

### 2.3. Función Datos(int Conjunto)

La función Datos($Conjunto$) recibe como parámetro el conjunto territorial para el cual se va a iniciar el proceso de optimización. Esta función se emplea para leer tres archivos de texto en los cuales se encuentra la información de las secciones necesaria para la construcción y optimización de distritos del conjunto territorial indicado: Separar.txt, DatosConglomerados.txt, ColindanciasUnidades.txt.

El archivo de texto Separar.txt está formado por dos columnas, con los identificadores de los municipios que deben considerarse como no vecinos por tiempos de traslado. Esta información se guarda en la variable local $Separar[][]$ de tipo entero.

El archivo de texto DatosConglomerados.txt contiene la siguiente información de cada sección: Municipio, número de sección, área, población, conglomerado al que pertenece y conjunto territorial al que pertenece. Esta información es guardada en variables que representan la cantidad de habitantes, y el área de cada unidad geográfica, como se muestra en la Tabla 2.3.

El archivo de texto ColindanciaUnidades.txt contiene para cada sección el número de la sección con la cual colinda y el perímetro que comparten en dicha colindancia, en la Tabla 2.4 se muestra como ejemplo las colindancias de una sección hipotética vecina de cuatro secciones.

Esta información es utilizada para determinar las unidades geográficas que son vecinas entre sí, y el perímetro de colindancia que comparten. Esta información es guardada en las

| Variable | Tipo | Dato almacenado |
|---|---|---|
| PoblacionUnidadGeografica | **int[]** | Cantidad de habitantes en cada conglomerado |
| AreaUnidadGeografica | **double[]** | Área de cada conglomerado |

**Tabla 2.3** Variables empleadas para los conglomerados

| Sección A | Sección B | Perímetro de colindancia |
|---|---|---|
| 1 | 2 | 1703 |
| 1 | 3 | 1498.1 |
| 1 | 15 | 468.9 |
| 1 | 16 | 1018.1 |

**Tabla 2.4** Colindancias de una sección.

variables $Vecinos[][]$ y $PerimetroFrontera[][]$.

Es importante mencionar que en este punto se usan los valores almacenados en la variable $Separar[]$, para determinar cuándo dos unidades geográficamente colindantes no deben considerarse como vecinas debido a tiempos de traslado.

## 2.4. Función Solucion_Inicial(int DistritosPorConjunto)

La función Solucion_Inicial($DistritosPorConjunto$) recibe como parámetro de entrada el número de distritos que debe generar, $DistritosPorConjunto$. Para generar $r$ distritos primero se eligen de forma aleatoria $r$ unidades geográficas, y cada unidad es asignada a un distrito diferente.

Después, se repiten los siguientes pasos hasta que cada unidad geográfica ha sido asignada a exactamente un distrito:

**S1** Elegir de manera aleatoria un distrito, $Distrito_i$.

**S2** Hacer una lista con las unidades geográficas que colindan con $Distrito_i$, y que aún no han sido asignadas a un distrito.

**S3** Elegir de forma aleatoria una de estas unidades geográficas y agregarla a $Distrito_i$.

**S4** Marcar la unidad geográfica seleccionada como ya asignada.

Los pasos **S1-S4** se repiten hasta que toda unidad geográfica ha sido asignada en algún distrito. De esta forma, por construcción, toda solución inicial está formada por $r$ distritos conexos.

El pseudocódigo de la función Solucion_Inicial se presenta en el Algoritmo 2.

**Algoritmo 2:** Pseudocódigo de la función Solucion_Inicial

```
Se eligen de forma aleatoria r unidades geográficas y cada una se asigna a un distrito distinto.
Mientras queden unidades geográficas sin asignar hacer
    Para cada distrito hacer
        Crear una lista con las unidades geográficas colindantes que aún no han sido asignadas.
        Elegir una unidad geográfica de la lista formada.
        Asignar la unidad geográfica al distrito.
        Marcar la unidad geográfica como ya asignada.
    fin
fin
```

### 2.5. Función Costo_Solucion_Inicial()

La función Costo_Solucion_Inicial() calcula el número de habitantes, área y perímetro de los distritos generados por la función Solucion_Inicial, tomando en cuenta la información de las unidades geográficas que forman a cada distrito. Estos datos son almacenados en las variables $PoblacionDistritos\_Actuales[]$, $MedidaArea[]$ y $MedidaPerimetro[]$.

Con estas variables se puede cuantificar la desviación poblacional y la compacidad geométrica mediante las funciones Desviacion_Poblacional y Compacidad respectivamente. Los costos obtenidos para cada distrito son guardados en las variables $DesviacionPoblacionalDistritos\_Actuales[]$ y $CompacidadDistritos\_Actuales[]$.

El pseudocódigo de la función Costo_Inicial se presenta en el Algoritmo 3.

### 2.6. Función Cambios()

La función Cambios() modifica la solución actual al cambiar de distrito a una unidad geográfica, para lo cual se realizan los siguientes pasos.

**C1** Se elige de forma aleatoria un distrito que contenga al menos dos unidades geográficas.

**Algoritmo 3:** Pseudocódigo de la función Costo_Inicial

```
Para cada Distrito_i, 1 ≤ i ≤ r hacer
    Calcular población de Distrito_i.
    Calcular área de Distrito_i.
    Calcular perímetro de Distrito_i.
fin
Para cada Distrito_i, 1 ≤ i ≤ r hacer
    Desviacion_Poblacional(PoblacionDistritos_Actuales[]).
    Compacidad(MedidaArea[], MedidaPerimetro[]).
fin
```

**C2** Se hace una lista, $L$, con todas las unidades geográficas, del distrito seleccionado, que colinden con otro distrito dentro del mismo conjunto territorial.

**C3** Se elige de forma aleatoria una de las unidades geográficas, $UG$, dentro de la lista *L*.

**C4** La unidad geográfica $UG$ es cambiada a un distrito con el cual colinde. En caso de haber más de una opción se elige una de forma aleatoria.

**C5** Se revisa si se ha perdido la conexidad del distrito mediante la función Revisa_Conexidad. En caso de ser así, deberá repararse con la función Repara_Conexidad.

La solución obtenida después de hacer estas modificaciones es evaluada mediante la función Costo_Nueva_Solucion. Los costos de la nueva solución son almacenados en las variables $DesviacionPoblacional\_Nueva$ y $Compacidad\_Nueva$.

El pseudocódigo de la función Cambios se presenta en el Algoritmo 4.

**Algoritmo 4:** Pseudocódigo de la función Cambios()

```
Elegir de forma aleatoria un distrito con al menos dos unidades geográficas, Distrito_i.
Hacer una lista L con las unidades geográficas de Distrito_i que colinden con otro distrito del mismo conjunto.
Elegir aleatoriamente una unidad geográfica, UG, de L.
Enviar la unidad UG a un distrito vecino elegido de forma aleatoria.
Revisar la conexidad de Distrito_i, en caso de ser necesario, deberá repararse. Evaluar la solución obtenida con el
cambio de UG.
```

## 2.7. Función Busqueda_Local()

La función Busqueda_Local() realiza una búsqueda local en la mejor solución encontrada por el algoritmo para converger a un óptimo local.

**BL1** Se visitan todas las unidades geográficas.

**BL2** Si una unidad geográfica colinda con otro distrito, la unidad geográfica es cambiada al distrito vecino y se evalúa el costo de la nueva solución.

**BL3** Si la nueva solución mejora el costo se acepta el cambio, en caso contrario se deshace el cambio.

La solución obtenida al final de este proceso es un óptimo local.
El pseudocódigo de la función Busqueda_Local se presenta en el Algoritmo 35.

**Algoritmo 5:** Pseudocódigo de la función Busqueda_Local()

```
1 Visitar todas las unidades geográficas.
2 Determinar si la unidad geográfica puede moverse a un distrito vecino.
3 Enviar la unidad UG a cada uno de los distritos con los cuales colinde.
4 Evaluar la solución obtenida con el cambio de UG a a cada uno de los distritos.
5 Aceptar el cambio si mejora el costo de la mejor solución conocida.
```

### 2.8. Función Cardinalidad_Distrito(int Distrito)

La función Cardinalidad_Distrito($Distrito$) recibe el identificador de un distrito. Su trabajo consiste en contar y devolver el número de unidades geográficas que lo forman.

### 2.9. Función Revisa_Conexidad(int Origen, int Destino)

La función Revisa_Conexidad($Origen, Destino$) recibe como parámetros los identificadores de dos distritos, $Origen$ y $Destino$, y cuenta el número de componentes conexas, $N$, que tiene el distrito $Origen$.

Si $N = 1$, significa que el distrito $Origen$ es conexo y la función termina.

Si $N \geq 1$, significa que el distrito $Origen$ es disconexo y debe repararse.

En este caso busca a la componente conexa que tenga el mayor número de unidades geográficas, y es conservada como el distrito $Origen$. Las unidades geográficas en otras componentes conexas son enviadas al distrito $Destino$ mediante la función Repara_Conexidad.

El pseudocódigo de la función Revisa_Conexidad se presenta en el Algoritmo 6.

**Algoritmo 6:** Pseudocódigo de la función Revisa_Conexidad

```
Sea NU el número de unidades geográficas en el distrito Origen.
Seleccionar de forma aleatoria una unidad geográfica, k_1, en distrito Origen.
Construir la ComponenteConexa_{k_1} de distrito Origen que contiene a k_1.
si |ComponenteConexa_{k_1}| = NU entonces
    El distrito Origen es conexo.
fin
si |ComponenteConexa_{k_1}| < NU entonces
    El distrito Origen tiene más de una componente conexa.
    Mientras alguna unidad no se encuentre en una componente hacer
        Seleccionar de forma aleatoria una unidad geográfica, k_i, que aún no esté en ninguna componente.
        Construir la ComponenteConexa_{k_i} que contiene a k_i.
    fin
    El nuevo distrito Origen es la componente con mayor número de unidades, digamos
    ComponenteConexa_{k_j}
    Para k_i ≠ k_j hacer
        Repara_Conexidad(Origen, k_i, Destino).
    fin
fin
```

### 2.10. Función Repara_Conexidad(int Origen, int k, int Destino)

La función Repara_Conexidad($Origen, k, Destino$) es llamada cuando se ha comprobado disconexión en un distrito. La función Repara_Conexidad recibe tres parámetros, el identificador del distrito que perdió la conexidad, $Origen$, el identificador de una unidad geográfica, $k$, que actualmente se encuentra en el distrito $Origen$, y el identificador de un distrito vecino, $Destino$.

La función visita a todas las unidades geográficas vecinas de $k$, y construye de forma creciente una componente conexa del distrito $Origen$ que contiene a $k$. Después, todas las unidades geográficas en esta componente conexa son enviadas al distrito $Destino$.

De esta forma, el distrito $Origen$ tiene una componente conexa menos.

El pseudocódigo de la función Repara_Conexidad se presenta en el Algoritmo 7.

**Algoritmo 7:** Pseudocódigo de la función Repara_Conexidad

```
1 Sean ComponenteConexa y Lista dos arreglos.
2 Agregar a k en ComponenteConexa y en Lista.
3 Para cada unidad geográfica i en Lista hacer
4 |   Visitar a los vecinos de i.
5 |   si una unidad vecina está en el distrito Origen entonces
6 |   |   Agregarla a ComponenteConexa y a Lista.
7 |   fin
8 fin
9 Cambiar a todas las unidades geográficas en ComponenteConexa al distrito Destino
```

### 2.11. Función Costo_Nueva_Solucion(int Origen, int Destino)

La función Costo_Nueva_Solucion($Origen, Destino$) recibe como parámetros los identificadores de los distritos que han sido modificados por el intercambio de unidades geográficas, producido después de haber utilizado a la función Cambios, y se encarga de calcular el costo de la nueva solución. Es importante destacar que la función Cambios sólo modifica a dos distritos, uno del cual se sacan unidades geográficas, $Distrito_{Origen}$, y otro en el cual se insertan, $Distrito_{Destino}$. Por lo tanto la función Costo_Nueva_Solucion únicamente requiere calcular el costo de estos distritos.

Para determinar el número de habitantes y el área de cada distrito, basta con sumar o restar la población y el área de las unidades geográficas que han sido agregadas o quitadas, según sea el caso.

Para obtener el perímetro de cada distrito se deben considerar las modificaciones sufridas en sus fronteras, para lo cual se revisan las colindancias de las unidades geográficas que han sido cambiadas de distrito.

Finalmente se evalua la desviación poblacional y la compacidad geométrica mediante las funciones Desviacion_Poblacional y Compacidad respectivamente. El costo de los distritos que han sido modificados por la función Cambios se guarda en las variables $DesviacionPoblacional\_Origen$, $DesviacionPoblacional\_Destino$, $CompacidadDistrito\_Origen$ y $CompacidadDistrito\_Destino$. Mientras que el costo total de la nueva solución es guardado en las variables $DesviacionPoblacional\_Nueva$ y $Compacidad\_Nueva$.

El pseudocódigo de la función Costo_Nueva_Solucion se presenta en el Algoritmo 8.

**Algoritmo 8:** Pseudocódigo de la función Costo_Nueva_Solucion

```
Para i = {Distrito_Origen, Distrito_Destino} hacer
    Calcular población de Distrito_i.
    Calcular área de Distrito_i.
    Calcular perímetro de Distrito_i.
fin
Para i = {Distrito_Origen, Distrito_Destino} hacer
    Desviacion_Poblacional(Distrito_i).
    Compacidad(Distrito_i).
fin
```

### 2.12. Función Desviacion_Poblacional(int Poblacion)

La función Desviacion_Poblacional($Poblacion$) recibe como parámetro la cantidad de habitantes en un distrito, $Poblacion$, y devuelve el costo poblacional correspondiente, aplicando la siguiente ecuación:

$$Costo = \left( \frac{1 - \left( \frac{Poblacion}{MediaEstatal} \right)}{0.15} \right)^2 \tag{2.1}$$

Si el valor de la variable $Costo$ es mayor que 1, se considera que se está violando la restricción de no exceder un $\pm 15\,\%$ de desviación poblacional con respecto a la media estatal, y se agrega una penalización dada por la siguiente ecuación:

$$Costo := Costo + 10 * (Costo - 1) \tag{2.2}$$

Finalmente devuelve el valor de $Costo$.

### 2.13. Función Compacidad(double Area,double Perimetro)

La función Compacidad($Area, Perimetro$) recibe como parámetros el área, $Area$, y perímetro, $Perimetro$, de un distrito, y devuelve el costo de compacidad aplicando la siguiente ecuación:

$$Costo = \left( \left( \frac{Perimetro}{\sqrt{Area}} * 0.25 \right) - 1.00 \right) * 0.5 \tag{2.3}$$

Finalmente devuelve el valor de $Costo$.

### 2.14. Función SiguienteAleatorioReal0y1(long * semilla)

La función SiguienteAleatorioReal0y1(* $semilla$) recibe un apuntador a la variable $semilla$. Emplea el valor de esta variable para generar, con una distribución uniforme, un número aleatorio en el intervalo $[0, 1]$. Antes de devolver el número generado modifica el valor de la variable $semilla$.

### 2.15. Función SiguienteAleatorioEnteroModN(long * semilla, int n)

La función SiguienteAleatorioEnteroModN(* $semilla$, $n$) recibe un apuntador a la variable $semilla$ y un número entero $n$. Emplea el valor de a variable $semilla$ para generar, con una distribución uniforme, un número entero aleatorio que se encuentra en el intervalo $[0, n-1]$. Antes de devolver el número generado modifica el valor de la variable $semilla$.

## CAPÍTULO 3

# DIAGRAMA DEL ALGORITMO BASADO EN COLONIA DE ABEJAS ARTIFICIALES

El objetivo del algoritmo basado en Colonia de Abejas Artificiales es generar $r$ distritos conexos, con las unidades geográficas que forman a cada entidad federativa, de tal forma que se respeten los criterios de equilibrio poblacional y compacidad geométrica.

Una solución es representada mediante un vector: $FuenteAlimento[j][x_1, x_2, ..., x_i, ..., x_n]$, donde la *i-ésima* entrada representa a la unidad geográfica $i$. La variable $x_i$ toma valores entre 1 y $r$, que corresponden al distrito en el cual es asignada la unidad geográfica $i$, y la variable $j$ toma valores de 3 a 500, que representan el número de fuentes de alimento con las que debe trabajar el algoritmo.

En la Figura 3.1 se presenta un diagrama que representa el funcionamiento del algoritmo basado en Colonia de Abejas Artificiales. Al inicio se obtiene información, dada por el usuario, de los parámetros que deberán emplearse y datos sobre las unidades geográficas, disponibles en archivos de texto. Posteriormente se construye de forma aleatoria el número de soluciones iniciales, o fuentes de alimento, indicados por el usuario, y se evalúa el costo

de cada una. A partir de este momento se inicia un ciclo que se repite tantas veces como lo indique la variable $GeneracionFinal$. Durante esta etapa se realizan modificaciones en todas las fuentes de alimento, se evalúa el costo de estas modificaciones, y se emplea un criterio glotón para guiar el proceso de búsqueda, es decir, sólo se aceptan las modificaciones que lleva a soluciones de menor costo.

Al completar el número de generaciones pedidas por el usuario, se realiza un búsqueda local en un vecindario de la mejor solución encontrada. Si hay mejoras, la solución es actualizada. Al terminar esta exploración, la solución se devuelve al usuario como el mejor escenario visitado.

Es importante destacar que cada una de las unidades geográficas ha sido previamente asignada a un conjunto territorial mediante la tipología de cada entidad federativa. De esta forma, cada conjunto está formado por un conjunto de unidades geográficas, y en él deben construirse un número preestablecido de distritos. Por lo anterior, el algoritmo fue diseñado para realizar en cada conjunto territorial una distritación independiente del resto del estado. Al terminar con todos los conjuntos se obtiene la distritación electoral del estado.

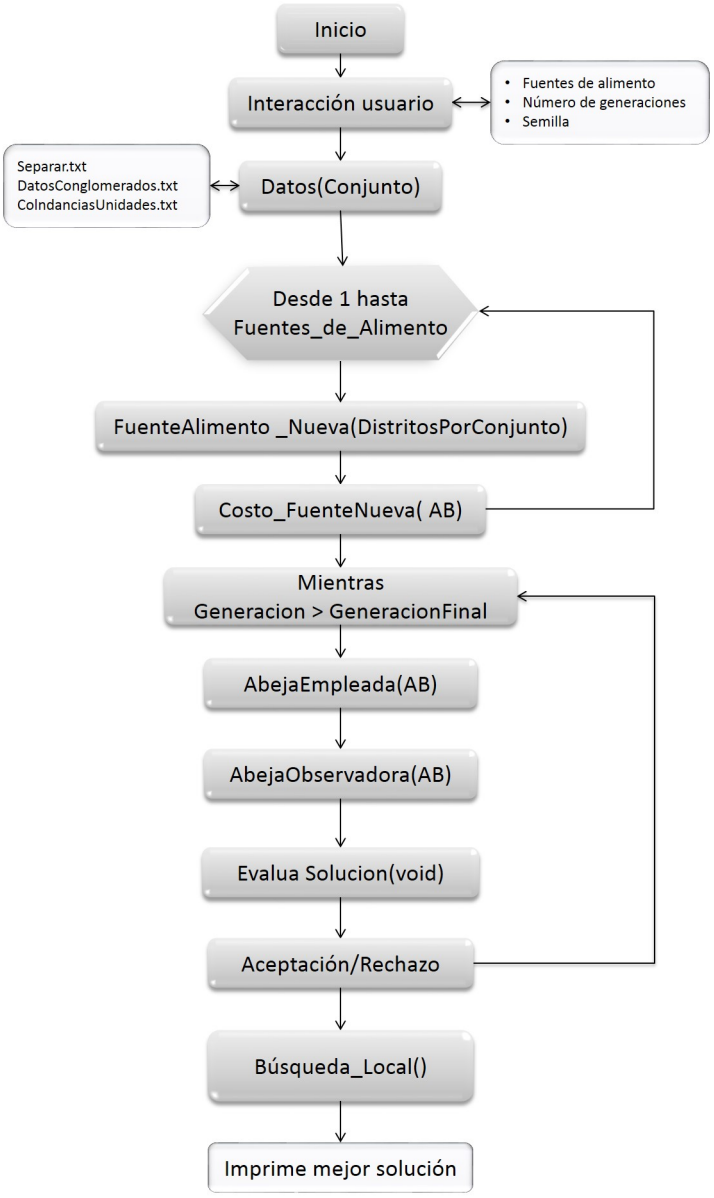


**Figura 3.1** Diagrama del algoritmo basado en Colonia de Abejas Artificiales.

# CAPÍTULO 4

# DESCRIPCIÓN TÉCNICA DEL ALGORITMO BASADO EN COLONIA DE ABEJAS ARTIFICIALES

En este capítulo se describe la forma en que operan cada una de las funciones del algoritmo basado en colonia de abejas artificiales empleado en el sistema de distritación electoral 2016. Primero se presentan las variables globales utilizadas en el algoritmo, junto con una descripción breve del uso que se hace de ellas, en las secciones restantes se describen dichas funciones.

## 4.1. Variables globales

En esta sección, en la Tabla 4.1, se presentan las variables globales más importantes empleadas por este algoritmo. En la primera columna se coloca el nombre de la variable, la segunda columna indica el tipo de variable empleado, y en la tercera columna se hace una pequeña descripción del uso que se le da durante la ejecución del algoritmo.

| Variable | Tipo | Descripción |
|---|---|---|
| Distrito | **int[]** | Arreglo con la asignación actual de cada unidad geográfica a un distrito. |
| AreaDistrito<br>PerimetroDistrito | **double[]** | Indican el área y perímetro por distrito de la solución guardada en la variable $Distrito[]$. |
| PoblacionDistrito | **int[]** | Indica la cantidad de habitantes por distrito de la solución en la variable $Distrito[]$. |
| DesviacionPoblacionalDistrito<br>CompacidadDistrito | **double[]** | Indican la desviación poblacional y compacidad por distrito de la solución guardada en la variable $Distrito[]$. |
| PerimetroFrontera | **double[][]** | Indica el perímetro compartido por dos unidades geográficas colindantes. |
| PoblacionUnidadGeografica | **int[]** | Guardan la cantidad de habitantes en cada unidad geográfica. |
| AreaUnidadGeografica | **double[]** | Guarda el área de cada unidad geográfica. |
| Vecinos | **int[][]** | Indica las unidades geográficas colindantes. |
| Semilla | **long** | Guarda el valor de la semilla propuesta por el usuario. |
| FuenteAlimento | **int[][]** | Permite almacenar hasta 500 soluciones o fuentes de alimento. |

| | | |
|---|---|---|
| Costo_FuenteAlimento<br>Compacidad_FuenteAlimento<br>DesviacionPoblacional_Fuen<br>teAlimento | **double[]** | Almacenan el costo total, el costo por compacidad y el costo por desviación poblacional de las fuentes de alimento. |
| Costo_Nueva<br>DesviacionPoblacional_Nueva<br>Compacidad_Nueva | **double** | Almacenan el costo total, el costo por compacidad y el costo por desviación poblacional de la solución guardada en la variable $Distrito[]$. |
| Fuentes_de_Alimento | **int** | Indica el número de fuentes de alimento que se emplearán durante el algoritmo. |
| ConjuntoActual<br>UnidadesPorConjunto<br>NDistritos | **int** | Indican el conjunto territorial que se está optimizando, el número de unidades geográficas que lo forman y el número de distritos que deben generarse en él. |
| Conversion | **int[]** | Asigna un identificador a cada unidad geográfica, que se usará durante el proceso de optimización. |
| DistritosFinales | **int[]** | Guarda la solución final generada por el algoritmo. |
| MediaEstatal | **int[]** | Guarda la media poblacional. |
| ConjuntosTotales | **int[]** | Guarda el número de distritos que deben construirse. |

**Tabla 4.1** Variables globales.

## 4.2. Función main()

La función main() inicia con la asignación de valores para algunas de las variables, tanto globales como locales, empleadas durante la ejecución del algoritmo. Las variables locales más importantes de esta función se presentan en la Tabla 4.2.

Nuevamente, en la primera columna se coloca el nombre de la variable, la segunda columna indica el tipo de variable empleado, y en la tercera columna se hace una pequeña descripción del uso que se le da durante la ejecución del algoritmo.

| Variable | Tipo | Descripción |
|---|---|---|
| MejorDesviacionPoblacional<br>MejorCompacidad | **double[]** | Almacenan el costo de cada distrito del escenario construido por el algoritmo después de una corrida. |
| Menor_DesviacionPoblacional<br>Menor_Compacidad | **double** | Almacenan el costo de la mejor fuente de alimento visitada por el algoritmo. |
| GeneracionFinal | **int** | Indica la cantidad de generaciones que ejecutará el algoritmo en cada conjunto territorial. |
| Calidad_FuenteAlimento | **double[]** | Indica la calidad de cada una de las fuentes de alimento |
| GeneracionesSinMejora | **int[]** | Indica el número de generaciones consecutivas sin mejora, para cada fuente de alimento. |
| Mejor_Escenario | **int[]** | Guarda la mejor solución encontrada por el algoritmo. Al terminar cada corrida se actualiza en caso haber encontrado una mejor solución. |
| Solucion_Hibrida | **int[]** | Guarda la mejor solución para cada conjunto territorial encontrada por el algoritmo. |

| | | |
|---|---|---|
| CostoPoblacional_MejorEscenario<br>Compacidad_MejorEscenario | **double** | Guardan los costos del mejor escenario construido por el algoritmo, |
| DesviacionPoblacional_Hibrida<br>Compacidad_Hibrida | **double[]** | Guardan los costos de la solución híbrida construida por el algoritmo. |

**Tabla 4.2** Variables locales de la función main()

La función inicia asignando el valor de los parámetros que el algoritmo empleará durante su ejecución:

- Fuentes_de_Alimento.
- GeneracionFinal.

Posteriormente se revisa el valor asignado a la variable Semilla. Si el usuario coloca un valor para esta variable, sólo se realizará una corrida empleando esta semilla para iniciar el generador de números aleatorios. Si el usuario elige la opción semillero se lee el archivo de texto Semillero.txt, y se realiza una corrida por cada una de las semillas encontradas en el archivo. Es importante destacar que el algoritmo está diseñado para leer hasta 1000 semillas. En caso de que el número de semillas en el archivo Semillero.txt sea mayor, sólo se considerarán las primeras 1000 semillas.

El siguiente paso consiste en realizar la lectura del archivo de texto ConjuntosDistritos.txt para determinar cuántos distritos se deberán crear en cada conjunto territorial, y se llama a la función Datos(int Conjunto) para obtener la información de cada unidad geográfica y poder emplearla durante el resto de la ejecución.

Cuando se ha obtenido la información de cada unidad geográfica, se inicia la construcción y optimización de distritos para cada conjunto territorial por separado.

La definición original del algoritmo de colonia de abejas artificiales establece que se debe calcular la calidad de las fuentes de alimento, mediante la siguiente ecuación:

$$Calidad\_FuenteAlimento[i] = \frac{1}{1 + Costo\_FuenteAlimento[i]} \tag{4.1}$$

Con esta información se calcula la $CalidadTotal$ de las fuentes de alimento actuales con la siguiente ecuación:

$$CalidadTotal = \sum_{i} Calidad\_FuenteAlimento[i] \tag{4.2}$$

Para cada conjunto territorial se repiten los siguientes pasos.

**ABC1** Se utiliza la función FuenteAlimento_Nueva, para crear de forma aleatoria, tantas soluciones como lo indique la variable $Fuente\_de\_Alimento$ .

**ABC2** Se evalúa el costo de cada una de las soluciones creadas mediante el uso de la función Costo_FuenteNueva.

**ABC3** Se inicia el proceso de mejora mediante un ciclo que durará hasta alcanzar el número de generaciones indicado por la variable $GeneracionFinal$. Durante este ciclo se realizan los siguientes pasos:

- **ABC3.1** La función AbejaEmpleada es aplicada a cada una de las fuentes de alimento. Esta función cambia una unidad geográfica a un distrito vecino. El cambio se acepta si mejora el costo, en caso contrario se rechaza.
- **ABC3.2** Si la fuente de alimento $i$ mejoró después de aplicarle la función AbejaEmpleada, entonces se reinicia el contador $GeneracionesSinMejora[i] = 0$. En caso de no obtener una mejora, el contador debe aumentarse en una unidad, $GeneracionesSinMejora[i] := GeneracionesSinMejora[i] + 1$.
- **ABC3.3** Si el contador de la fuente de alimento $i$ alcanza el valor de 100, entonces se crea una fuente de alimento aleatoria, mediante la función FuenteAlimento_Nueva, para sustituirla. Se evalúa el costo de la nueva fuente de alimento y se reinicia el contador $GeneracionesSinMejora[i] = 0$.
- **ABC3.4** Cuando la función AbejaEmpleada ha sido aplicada a todas las fuentes de alimento se procede a calcular la calidad de cada una de las soluciones y la calidad total, mediante las ecuaciones 4.1 y 4.2 respectivamente.

**ABC3.5** La función AbejaObservadora es llamada tantas veces como fuentes de alimento se estén usando en el programa. Esta función elige una fuente de alimento al azar, pero le da más probabilidad a las fuentes de alimento de mejor calidad. La AbejaObservadora modifica un distrito de la fuente de alimento seleccionada, al quitarle una unidad geográfica y añadirle otra.

**ABC3.6** Si la fuente de alimento $i$ mejoró después de aplicarle la función AbejaObservadora, entonces se reinicia el contador $GeneracionesSinMejora[i] = 0$. En caso contrario, los cambios de unidades geográficas son rechazados.

**ABC4** La mejor solución encontrada se guarda en memoria.

Cuando cada uno de los conjuntos territoriales han sido procesados mediante los pasos **ABC1**-**ABC4** se da por concluida una generación del algoritmo. Al concluir todas las generaciones indicadas por el usuario, se realiza una exploración en el vecindario de la mejor solución encontrada, mediante el uso de la función Busqueda_Local. La unión de las soluciones obtenidas para cada conjunto territorial se convierte en el escenario final. Es importante insistir en que se realizará una corrida por cada semilla dada al algoritmo.

Si el usuario propuso el valor para la variable Semilla, sólo se genera un escenario que es devuelto después de la primera corrida. Si el usuario eligió la opción Semillero, entonces se generarán tantos escenarios como semillas se tengan. Además, al concluir todas las corridas se devolverá la mejor distritación encontrada al combinar los mejores distritos para cada conjunto territorial, de cada uno de los escenarios finales obtenidos.

El pseudocódigo de la función main se presenta en el Algoritmo 9.

### 4.3. Función Datos(int Conjunto)

La función Datos($Conjunto$) recibe como parámetro el conjunto territorial para el cual se va a iniciar el proceso de optimización. Esta función se emplea para leer tres archivos de texto en los cuales se encuentra la información de las secciones necesaria para la construcción y optimización de distritos del conjunto territorial indicado: Separar.txt, DatosConglomerados.txt, ColindanciasUnidades.txt.

**Algoritmo 9:** Pseudocódigo de la función main

```
1  Solicitar valores de parámetros al usuario.
2  Lectura del archivo Conjuntos.txt, y llamado a la función Datos(Conjunto).
3  Para cada conjunto territorial hacer
4      Crear las fuentes de alimento con la función FuenteAlimento_Nueva(NDistritos), y evaluar el costo de cada
       fuente de alimento con la función Costo_FuenteNueva(AB).
5      Mientras Generación < GeneracionFinal hacer
6          Para i = 1 hasta Fuentes_de_Alimento hacer
7              Modificar la fuente de alimento i mediante la función AbejaEmpleada(i), y evaluar el costo del
               cambio realizado.
8              Si el costo de la fuente de alimento disminuye, se acepta el cambio y se actualiza el contador
               GeneracionesSinMejora[i] = 0.
9              Si el costo de la fuente de alimento aumenta, se rechaza el cambio, y se aumenta el contador
               GeneracionesSinMejora[i] := GeneracionesSinMejora[i] + 1.
10             Si el contador GeneracionesSinMejora[i] alcanza el valor 100, se debe sustituir la fuente de
               alimento i, con una solución nueva.
11             Calcular la calidad de la fuente de alimento i mediante la ecuación 4.1
12         fin
13         Calcular CalidadTotal mediante la ecuación 4.2.
14         Para i = 1 hasta Fuentes_de_Alimento hacer
15             Elegir una fuente de alimento j, dando más probabilidad a las de mejor calidad.
16             Modificar la fuente de alimento i mediante la función AbejaObservadora(j), y evaluar el costo de
               los cambios realizados.
17             Si el costo de la fuente de alimento disminuye, se acepta el cambio y se actualiza el contador
               GeneracionesSinMejora[j] = 0.
18             Si el costo de la fuente de alimento aumenta, se rechazan los cambios.
19         fin
20         Guardar en memoria la mejor solución encontrada.
21     fin
22 fin
23 Juntar las soluciones encontradas para cada conjunto y devolverlas como mejor distritación encontrada para el
   estado.
```

El archivo de texto Separar.txt está formado por dos columnas, con los identificadores de los municipios que deben considerarse como no vecinos por tiempos de traslado.

El archivo de texto DatosConglomerados.txt contiene la siguiente información de cada sección: Municipio, número de sección, área, población, conglomerado al que pertenece y conjunto territorial al que pertenece. Esta información es guardada en variables que representan la cantidad de habitantes y el área de cada unidad geográfica, como se muestra en la Tabla 4.4.

| Variable | Tipo | Dato almacenado |
|---|---|---|
| PoblacionUnidadGeografica | **int[]** | Cantidad de habitantes en cada conglomerado |
| AreaUnidadGeografica | **double[]** | Área de cada conglomerado |

**Tabla 4.4** Variables empleadas para los conglomerados

El archivo de texto ColindanciaUnidades.txt contiene para cada sección el número de la sección con la cual colinda y el perímetro que comparten en dicha colindancia, en la Tabla 4.5 se muestra como ejemplo las colindancias de una sección hipotética vecina de cuatro secciones.

| Sección A | Sección B | Perímetro de colindancia |
|---|---|---|
| 1 | 2 | 1703 |
| 1 | 3 | 1498.1 |
| 1 | 15 | 468.9 |
| 1 | 16 | 1018.1 |

**Tabla 4.5** Colindancias de una sección.

La información es utilizada para determinar las unidades geográficas que son vecinas entre sí, y el perímetro de colindancia que comparten. Esta información es guardada en las variables $Vecinos[][]$ y $PerimetroFrontera[][]$.

Es importante mencionar que en este punto se usan los valores almacenados en la variable $Separar[]$, para determinar cuándo dos unidades geográficamente colindantes no deben considerarse como vecinas debido a tiempos de traslado.

### 4.4. Función FuenteAlimento_Nueva(int DistritosPorConjunto)

La función FuenteAlimento_Nueva($DistritosPorConjunto$) recibe como parámetro de entrada el número de distritos que debe generar, $DistritosPorConjunto$. Para generar $r$ distritos primero se eligen de forma aleatoria $r$ unidades geográficas, y cada unidad es asignada a un distrito diferente.

Después, se repiten los siguientes pasos hasta que cada unidad geográfica ha sido asignada a exactamente un distrito:

**FA1** Elegir de manera aleatoria un distrito, $Distrito_i$.

**FA2** Hacer una lista con las unidades geográficas que colindan con $Distrito_i$, y que aún no han sido asignadas a un distrito.

**FA3** Elegir de forma aleatoria una de estas unidades geográficas y se agrega a $Distrito_i$.

**FA4** Marcar la unidad geográfica seleccionada como ya asignada.

Los pasos **FA1-FA4** se repiten hasta que toda unidad geográfica ha sido asignada en algún distrito. De esta forma, por construcción, toda solución inicial esta formada por $r$ distritos conexos.

El pseudocódigo de la función FuenteAlimento_Nueva se presenta en el Algoritmo 10.

**Algoritmo 10:** Pseudocódigo de la función FuenteAlimento_Nueva

```
1 Se eligen de forma aleatoria r unidades geográficas y cada una se asigna a un distrito diferente.
2 Mientras queden unidades geográficas sin asignar hacer
3     Para cada distrito hacer
4         Crear una lista con las unidades geográficas colindantes que aún no han sido asignadas.
5         Elegir una unidad geográfica de la lista formada.
6         Asignar la unidad geográfica al distrito.
7         Marcar la unidad geográfica como ya asignada.
8     fin
9 fin
```

### 4.5. Función Costo_FuenteNueva(int AB)

La función Costo_FuenteNueva($AB$) recibe como parámetro el identificador de una fuente de alimento, $AB$. Calcula el número de habitantes, área y perímetro de los distritos,

tomando en cuenta la información de las unidades geográficas que forman a cada distrito en la solución $AB$. Estos datos son almacenados en las variables $PoblacionDistrito[]$, $AreaDistrito[]$ y $PerimetroDistrito[]$.

Con estas variables se puede cuantificar la desviación poblacional y la compacidad geométrica mediante las funciones Desviacion_Poblacional y Compacidad respectivamente. Los costos totales para cada fuente de alimento son guardados en las variables $Desviacion Poblacional\_FuenteAlimento[AB]$ y $Compacidad\_FuenteAlimento[AB]$.

El pseudocódigo de la función Costo_FuenteNueva se presenta en el Algoritmo 23.

**Algoritmo 11:** Pseudocódigo de la función Costo_FuenteNueva

```
1 Para cada Distrito_i, 1 ≤ i ≤ r hacer
2 |   Calcular población de Distrito_i.
3 |   Calcular área de Distrito_i.
4 |   Calcular perímetro Distrito_i.
5 fin
6 Para cada Distrito_i, 1 ≤ i ≤ r hacer
7 |   Desviacion_Poblacional(PoblacionDistrito[]).
8 |   Compacidad(AreaDistrito[], PerimetroDistrito[]).
9 fin
```

## 4.6. Función AbejaEmpleada(int AB)

La función AbejaEmpleada($AB$) recibe como parámetro el identificador de una fuente de alimento, $AB$. El trabajo de esta función consiste en modificar la solución $AB$ al cambiar de distrito a una unidad geográfica, para lo cual se realizan los siguientes pasos.

**ABE1** Se elige un distrito de forma aleatoria que contenga al menos dos unidades geográficas.

**ABE2** Se hace una lista, $L$, con todas las unidades geográficas, del distrito seleccionado, que colinden con otro distrito dentro del mismo conjunto territorial.

**ABE3** Se elige de forma aleatoria una de las unidades geográficas, $UG$, dentro de la lista $L$.

**ABE4** La unidad geográfica $UG$ es cambiada a un distrito con el cual colinde. En caso de haber más de una opción se elige uno de forma aleatoria.

**ABE5** Se revisa si se ha perdido la conexidad del distrito mediante la función Revisa-Conexidad_Empleada. En caso de ser así, deberá repararse con la función ReparaConexidad_Empleada.

La solución obtenida después de hacer estas modificaciones es evaluada mediante la función Costo_FuenteNueva.

El pseudocódigo de la función AbejaEmpleada se presenta en el Algoritmo 12.

**Algoritmo 12:** Pseudocódigo de la función AbejaEmpleada(int AB)

1 Elegir de forma aleatoria un distrito con al menos dos unidades geográficas, $Distrito_i$.
2 Hacer una lista $L$ con las unidades geográficas de $Distrito_i$ que colinden con otro distrito del mismo conjunto.
3 Elegir aleatoriamente una unidad geográfica, $UG$, de $L$.
4 Enviar la unidad $UG$ a un distrito vecino elegido de forma aleatoria.
5 Revisar la conexidad de $Distrito_i$, en caso de ser necesario, deberá repararse. Evaluar la solución obtenida con el cambio de $UG$.

## 4.7. Función Busqueda_Local()

La función Busqueda_Local() realiza una búsqueda local en la mejor solución encontrada por el algoritmo para converger a un óptimo local.

**BL1** Se visitan todas las unidades geográficas.

**BL2** Si una unidad geográfica colinda con otro distrito, la unidad geográfica es cambiada al distrito vecino y se evalúa el costo de la nueva solución.

**BL3** Si la nueva solución mejora el costo se acepta el cambio, en caso contrario se deshace el cambio.

La solución obtenida al final de este proceso es un óptimo local.
El pseudocódigo de la función Busqueda_Local se presenta en el Algoritmo 35.

**Algoritmo 13:** Pseudocódigo de la función Busqueda_Local()

1 Visitar todas las unidades geográficas.
2 Determinar si la unidad geográfica puede moverse a un distrito vecino.
3 Enviar la unidad $UG$ a cada uno de los distritos con los cuales colinde.
4 Evaluar la solución obtenida con el cambio de $UG$ a a cada uno de los distritos.
5 Aceptar el cambio si mejora el costo de la mejor solución conocida.

### 4.8. Función Cardinalidad_Distrito(int Fuente, int Z)

La función Cardinalidad_Distrito($Fuente, Z$) recibe el identificador de un distrito. Su trabajo consiste en contar y devolver el número de unidades geográficas que lo forman.

### 4.9. Función RevisaConexidad_Empleada(int Origen, int Destino)

La función RevisaConexidad_Empleada($Origen, Destino$) recibe como parámetros los identificadores de dos distritos, $Origen$ y $Destino$, y cuenta el número de componentes conexas, $N$, que tiene el distrito $Origen$.

Si $N = 1$, significa que el distrito $Origen$ es conexo y la función termina.

Si $N \geq 1$, significa que el distrito $Origen$ es disconexo y debe repararse.

En este caso busca a la componente conexa que tenga el mayor número de unidades geográficas, y es conservada como el distrito $Origen$. Las unidades geográficas en otras componentes conexas son enviadas al distrito $Destino$ mediante la función ReparaConexidad_Empleada.

El pseudocódigo de la función RevisaConexidad_Empleada se presenta en el Algoritmo 14.

### 4.10. Función ReparaConexidad_Empleada(int Origen, int Unidad, int Destino)

La función ReparaConexidad_Empleada($Origen, Unidad, Destino$) es llamada cuando se ha comprobado disconexión en un distrito. La función ReparaConexidad_Empleada recibe tres parámetros, el identificador del distrito que perdió la conexidad, $Origen$, el identificador de una unidad geográfica, $k$, que actualmente se encuentra en el distrito $Origen$, y el identificador de un distrito vecino, $Destino$.

La función visita a todas las unidades geográficas vecinas de $k$, y construye de forma creciente una componente conexa del distrito $Origen$ que contiene a $k$. Después, todas las unidades geográficas en esta componente conexa son enviadas al distrito $Destino$.

**Algoritmo 14:** Pseudocódigo de la función RevisaConexidad_Empleada

```
1  Sea NU el número de unidades geográficas en el distrito Origen.
2  Seleccionar de forma aleatoria una unidad geográfica, k_1, en distrito Origen.
3  Construir la ComponenteConexa_{k_1} de distrito Origen que contiene a k_1.
4  si |ComponenteConexa_{k_1}| = NU entonces
5  |   El distrito Origen es conexo.
6  fin
7  si |ComponenteConexa_{k_1}| < NU entonces
8  |   El distrito Origen tiene más de una componente conexa.
9  |   Mientras alguna unidad no se encuentre en una componente hacer
10 |   |   Seleccionar de forma aleatoria una unidad geográfica, k_i, que aún no esté en ninguna componente.
11 |   |   Construir la ComponenteConexa_{k_i} que contiene a k_i.
12 |   fin
13 |   El nuevo distrito Origen es la componente con mayor número de unidades, denominada
   |   ComponenteConexa_{k_j}
14 |   Para k_i ≠ k_j hacer
15 |   |   ReparaConexidad_Empleada(Origen, k_i, Destino).
16 |   fin
17 fin
```

De esta forma, el distrito $Origen$ tiene una componente conexa menos.

El pseudocódigo de la función ReparaConexidad_Empleada se presenta en el Algoritmo 15.

**Algoritmo 15:** Pseudocódigo de la función ReparaConexidad_Empleada

```
1  Sean ComponenteConexa y Lista dos arreglos.
2  Agregar a k en ComponenteConexa y en Lista.
3  Para cada unidad geográfica i en Lista hacer
4  |   Visitar a los vecinos de i.
5  |   si una unidad vecina está en el distrito Origen entonces
6  |   |   Agregarla a ComponenteConexa y a Lista.
7  |   fin
8  fin
9  Cambiar a todas las unidades geográficas en ComponenteConexa al distrito Destino
```

### 4.11. Función AbejaObservadora(int AB)

La función AbejaObservadora($AB$) recibe como parámetro el identificador de una fuente de alimento, $AB$, y modifica uno de sus distritos al quitarle y agregarle dos unidades

geográficas mediante los siguientes pasos.

**ABO1** Selecciona de forma aleatoria un distrito, $OrigenAB$, de la fuente de alimento $AB$.

**ABO2** Selecciona de forma aleatoria una unidad geográfica, $k$, en el distrito $Origen$.

**ABO3** Selecciona de forma aleatoria una fuente de alimento, $AB1$, diferente a $AB$.

**ABO4** Se identifica al distrito al que pertenece la unidad $k$ en la fuente de alimento $AB1$, sea $OrigenAB1$.

**ABO5** Se elige de forma aleatoria una unidad geográfica que esté en la frontera del distrito $OrigenAB$, pero que no esté en $OrigenAB1$, y se cambia a un distrito vecino.

**ABO6** Se revisa si el distrito $OrigenAB$ perdió la conexidad mediante la función RevisaConexidad_Observadora1, de ser así se repara con la función ReparaConexidad_Observadora.

**ABO7** Se elige de forma aleatoria una unidad geográfica que colinde con el distrito $OrigenAB$, y que pertenezca al distrito $OrigenAB1$, y se cambia al distrito $OrigenAB$.

**ABO8** Se revisa si los distritos modificados perdieron la conexidad mediante la función RevisaConexidad_Observadora2, de ser así se reparan con la función ReparaConexidad_Observadora.

**ABO9** Si el costo de la nueva solución es mejor que el costo de $AB$, los cambios se aceptan, en otro caso los cambios se rechazan.

El pseudocódigo de la función AbejaObservadora se presenta en el Algoritmo 16.

### 4.12. Función RevisaConexidad_Observadora1(int DistritoAnalizado)

La función RevisaConexidad_Observadora1($DistritoAnalizado$) recibe como parámetro el identificador de un distrito, $DistritoAnalizado$, y cuenta el número de componentes conexas, $N$, que tiene este distrito.

Si $N = 1$, significa que $DistritoAnalizado$ es conexo y la función termina.

Si $N \geq 1$, significa que $DistritoAnalizado$ es disconexo y debe repararse.

**Algoritmo 16:** Pseudocódigo de la función AbejaObservadora

1 Sea $AB$ una fuente de alimento.
2 Seleccionar de forma aleatoria un distrito, $OrigenAB$.
3 Seleccionar de forma aleatoria una unidad geográfica, $k$, del distrito $Origen$.
4 Seleccionar de forma aleatoria una fuente de alimento, $AB1$, diferente a $AB$.
5 Sea $OrigenAB1$ el distrito al que pertenece la unidad $k$ en la fuente de alimento $AB1$.
6 Elegir de forma aleatoria una unidad geográfica en la frontera de $OrigenAB$, que no esté en $OrigenAB1$, y cambiarla a un distrito vecino.
7 Revisar si el distrito $OrigenAB$ perdió la conexidad, de ser así se repara.
8 Elegir de forma aleatoria una unidad geográfica en $OrigenAB1$ que colinde con $OrigenAB$, y cambiarla al distrito $OrigenAB$.
9 Revisar si los distritos modificados perdieron la conexidad, de ser así se reparan.
10 Si el costo de la nueva solución es mejor que el costo de $AB$, los cambios se aceptan, en otro caso los cambios se rechazan.

En este caso busca a la componente conexa que tenga el mayor número de unidades geográficas, y es conservada como $DistritoAnalizado$. Las unidades geográficas en otras componentes conexas son enviadas a distritos vecinos mediante la función ReparaConexidad_Observadora.

El pseudocódigo de la función RevisaConexidad_Observadora1 se presenta en el Algoritmo 17.

**Algoritmo 17:** Pseudocódigo de la función RevisaConexidad_Observadora1

1 Sea $NU$ el número de unidades geográficas en $DistritoAnalizado$.
2 Seleccionar de forma aleatoria una unidad geográfica, $k_1$, en $DistritoAnalizado$.
3 Construir la $ComponenteConexa_{k_1}$ de $DistritoAnalizado$ que contiene a $k_1$.
4 **si** $|ComponenteConexa_{k_1}| = NU$ **entonces**
5 $DistritoAnalizado$ es conexo.
6 **fin**
7 **si** $|ComponenteConexa_{k_1}| < NU$ **entonces**
8 $DistritoAnalizado$ tiene más de una componente conexa.
9 **Mientras** *alguna unidad no se encuentre en una componente* **hacer**
10 Seleccionar de forma aleatoria una unidad geográfica, $k_i$, que aún no esté en ninguna componente.
11 Construir la $ComponenteConexa_{k_i}$ que contiene a $k_i$.
12 **fin**
13 El nuevo $DistritoAnalizado$ es la componente con mayor número de unidades, digamos $ComponenteConexa_{k_j}$
14 **Para** $k_i \neq k_j$ **hacer**
15 ReparaConexidad_Observadora($DistritoAnalizado, k_i$).
16 **fin**
17 **fin**

### 4.13. Función RevisaConexidad_Observadora2(int DistritoOrigen, int UnidadOrigen)

La función RevisaConexidad_Observadora2($DistritoOrigen, UnidadOrigen$) recibe como parámetros el identificador de un distrito, $DistritoOrigen$, y el identificador de una unidad geográfica, $UnidadOrigen$, y cuenta el número de componentes conexas, $N$, que tiene $DistritoOrigen$.

Si $N = 1$, significa que $DistritoOrigen$ es conexo y la función termina.

Si $N \geq 1$, significa que $DistritoOrigen$ es disconexo y debe repararse.

En este caso busca a la componente conexa que contenga a la unidad geográfica $Unidad-Origen$, y es conservada como $DistritoOrigen$. Las unidades geográficas en otras componentes conexas son enviadas a distritos vecinos mediante la función ReparaConexidad_Observadora.

El pseudocódigo de la función RevisaConexidad_Observadora2 se presenta en el Algoritmo 18.

**Algoritmo 18:** Pseudocódigo de la función RevisaConexidad_Observadora2

```
Sea NU el número de unidades geográficas en DistritoOrigen.
Seleccionar de forma aleatoria una unidad geográfica, k_1, en DistritoOrigen.
Construir la ComponenteConexa_{k_1} de DistritoOrigen que contiene a k_1.
si |ComponenteConexa_{k_1}| = NU entonces
    DistritoOrigen es conexo.
fin
si |ComponenteConexa_{k_1}| < NU entonces
    DistritoOrigen tiene más de una componente conexa.
    Mientras alguna unidad no se encuentre en una componente hacer
        Seleccionar de forma aleatoria una unidad geográfica, k_i, que aún no esté en ninguna componente.
        Construir la ComponenteConexa_{k_i} que contiene a k_i.
    fin
    El nuevo DistritoOrigen es la componente que contiene a UnidadOrigen, digamos
    ComponenteConexa_{k_j}
    Para k_i ≠ k_j hacer
        ReparaConexidad_Observadora(DistritoAnalizado, k_i).
    fin
fin
```

### 4.14. Función ReparaConexidad_Observadora(int DistritoOrigen, int UnidadOrigen)

La función ReparaConexidad_Observadora($DistritoOrigen, UnidadOrigen$) es llamada cuando se ha comprobado disconexión en un distrito. La función ReparaConexidad_Observadora recibe dos parámetros, el identificador del distrito que perdió la conexidad, $DistritoOrigen$, y el identificador de una unidad geográfica, $UnidadOrigen$.

La función visita a todas las unidades geográficas vecinas de $UnidadOrigen$, y construye de forma creciente una componente conexa de $DistritoOrigen$ que contiene a $UnidadOrigen$. Después, todas las unidades geográficas en esta componente conexa son enviadas a un distrito vecino elegido de forma aleatoria.

De esta forma, $DistritoOrigen$ tiene una componente conexa menos.

El pseudocódigo de la función ReparaConexidad_Observadora se presenta en el Algoritmo 19.

**Algoritmo 19:** Pseudocódigo de la función ReparaConexidad_Observadora

```
1  Sean ComponenteConexa, Lista y Destinos tres arreglos.
2  Agregar a UnidadOrigen en ComponenteConexa y en Lista.
3  Para cada unidad geográfica i en Lista hacer
4  |   Visitar a los vecinos de i.
5  |   si la unidad vecina está en DistritoOrigen entonces
6  |   |   Agregarla a ComponenteConexa y a Lista.
7  |   fin
8  |   si la unidad vecina no está en DistritoOrigen entonces
9  |   |   Agregar el distrito de la unidad vecina a Destinos.
10 |   fin
11 fin
12 Elegir de forma aleatoria un distrito de la lista Destinos.
13 Cambiar a todas las unidades geográficas en ComponenteConexa al distrito elegido
```

### 4.15. Función Evalua_Solucion(void)

La función Evalua_Solucion() calcula el número de habitantes, área y perímetro de los distritos almacenados en la variable $Distrito[]$, tomando en cuenta la información de las unidades geográficas que forman a cada distrito en la solución $AB$. Estos datos son almacenados en las variables $PoblacionDistrito[]$, $AreaDistrito[]$ y $PerimetroDistrito[]$.

Con estas variables se puede cuantificar la desviación poblacional y la compacidad geométrica mediante las funciones Desviacion_Poblacional y Compacidad respectivamente. Los costos totales para esta solución son guardados en las variables $DesviacionPoblacional\_Nueva$ y $Compacidad\_Nueva$.

El pseudocódigo de la función Evalua_Solucion se presenta en el Algoritmo 20.

**Algoritmo 20:** Pseudocódigo de la función Evalua_Solucion

```
1 Para cada Distrito_i, 1 ≤ i ≤ r hacer
2 |   Calcular población de Distrito_i.
3 |   Calcular área de Distrito_i.
4 |   Calcular perímetro de Distrito_i.
5 fin
6 Para cada Distrito_i, 1 ≤ i ≤ r hacer
7 |   Desviacion_Poblacional(PoblacionDistrito[]).
8 |   Compacidad(AreaDistrito[], PerimetroDistrito[]).
9 fin
```

### 4.16. Función Desviacion_Poblacional(int Poblacion)

La función Desviacion_Poblacional($Poblacion$) recibe como parámetro la cantidad de habitantes en un distrito, $Poblacion$, y devuelve el costo poblacional correspondiente, aplicando la siguiente ecuación:

$$Costo = \left( \frac{1 - \left( \frac{Poblacion}{MediaEstatal} \right)}{0.15} \right)^2 \tag{4.3}$$

Si el valor de la variable $Costo$ es mayor que 1, se considera que se está violando la restricción de no exceder un $\pm 15\,\%$ de desviación poblacional con respecto a la media, y se agrega una penalización dada por la siguiente ecuación:

$$Costo := Costo + 10 * (Costo - 1) \tag{4.4}$$

Finalmente devuelve el valor de $Costo$.

### 4.17. Función Compacidad(double Area,double Perimetro)

La función Compacidad($Area, Perimetro$) recibe como parámetros el área, $Area$, y perímetro, $Perimetro$, de un distrito, y devuelve el costo de compacidad aplicando la siguiente ecuación:

$$Costo = \left( \left( \frac{Perimetro}{\sqrt{Area}} * 0.25 \right) - 1.00 \right) * 0.5 \tag{4.5}$$

Finalmente devuelve el valor de $Costo$.

### 4.18. Función SiguienteAleatorioReal0y1(long * semilla)

La función SiguienteAleatorioReal0y1(* $semilla$) recibe un apuntador a la variable $semilla$. Emplea el valor de esta variable para generar, con una distribución uniforme, un número aleatorio en el intervalo $[0, 1]$. Antes de devolver el número generado modifica el valor de la variable $semilla$.

### 4.19. Función SiguienteAleatorioEnteroModN(long * semilla, int n)

La función SiguienteAleatorioEnteroModN(* $semilla$, $n$) recibe un apuntador a la variable $semilla$ y un un número entero $n$. Emplea el valor de a variable $semilla$ para generar, con una distribución uniforme, un número entero aleatorio que se encuentra en el intervalo $[0, n-1]$. Antes de devolver el número generado modifica el valor de la variable $semilla$.

## CAPÍTULO 5

# DIAGRAMA DEL ALGORITMO ABC-RS

El objetivo del algoritmo ABC-RS es generar $r$ distritos conexos, con las unidades geográficas que forman a cada entidad federativa, de tal forma que se respeten los criterios de equilibrio poblacional y compacidad geométrica.

Una solución es representada mediante un vector: $FuenteAlimento[j][x_1, x_2, ..., x_i, ..., x_n]$, donde la *i-ésima* entrada representa a la unidad geográfica $i$. La variable $x_i$ toma valores entre 1 y $r$, que corresponden al distrito en el cual es asignada la unidad geográfica $i$, y la variable $j$ toma valores de 3 a 500, que representan el número de fuentes de alimento con las que debe trabajar el algoritmo.

En la Figura 5.1 se presenta un diagrama que representa el funcionamiento del algoritmo ABC-RS. Al inicio se obtiene información, dada por el usuario, de los parámetros que deberán emplearse y datos sobre las unidades geográficas, disponibles en archivos de texto. Posteriormente se construye de forma aleatoria el número de soluciones iniciales, o fuentes de alimento, indicados por el usuario, y se evalúa el costo de cada una. A partir

de este momento se inicia un ciclo que dura hasta que se alcanza el valor de la variable $TemperaturaFinal$. Durante esta etapa se realizan modificaciones en los distritos de las soluciones generadas por el algoritmo, para lo cual se pueden emplear dos funciones: CambiosRS() o AbejaObservadora(AB). Se evalúa el costo de estas modificaciones, y se emplea el criterio de Metrópolis para guiar el proceso de búsqueda, mediante aceptaciones o rechazos probabilísticos. La mejor solución encontrada en este ciclo es almacenada y usada en futuras iteraciones.

Al alcanzar la temperatura final, se realiza un búsqueda local en un vecindario de la mejor solución encontrada. Si hay mejoras, la solución es actualizada. Al terminar esta exploración, la solución se devuelve al usuario como el mejor escenario visitado.

Es importante destacar que cada una de las unidades geográficas ha sido previamente asignada a un conjunto territorial mediante la tipología de cada entidad federativa. De esta forma, cada conjunto está formado por un conjunto de unidades geográficas, y en él deben construirse un número preestablecido de distritos. Por lo anterior, el algoritmo fue diseñado para realizar en cada conjunto territorial una distritación electoral independiente del resto del estado. Al terminar con todos los conjuntos se obtiene la distritación electoral del estado.

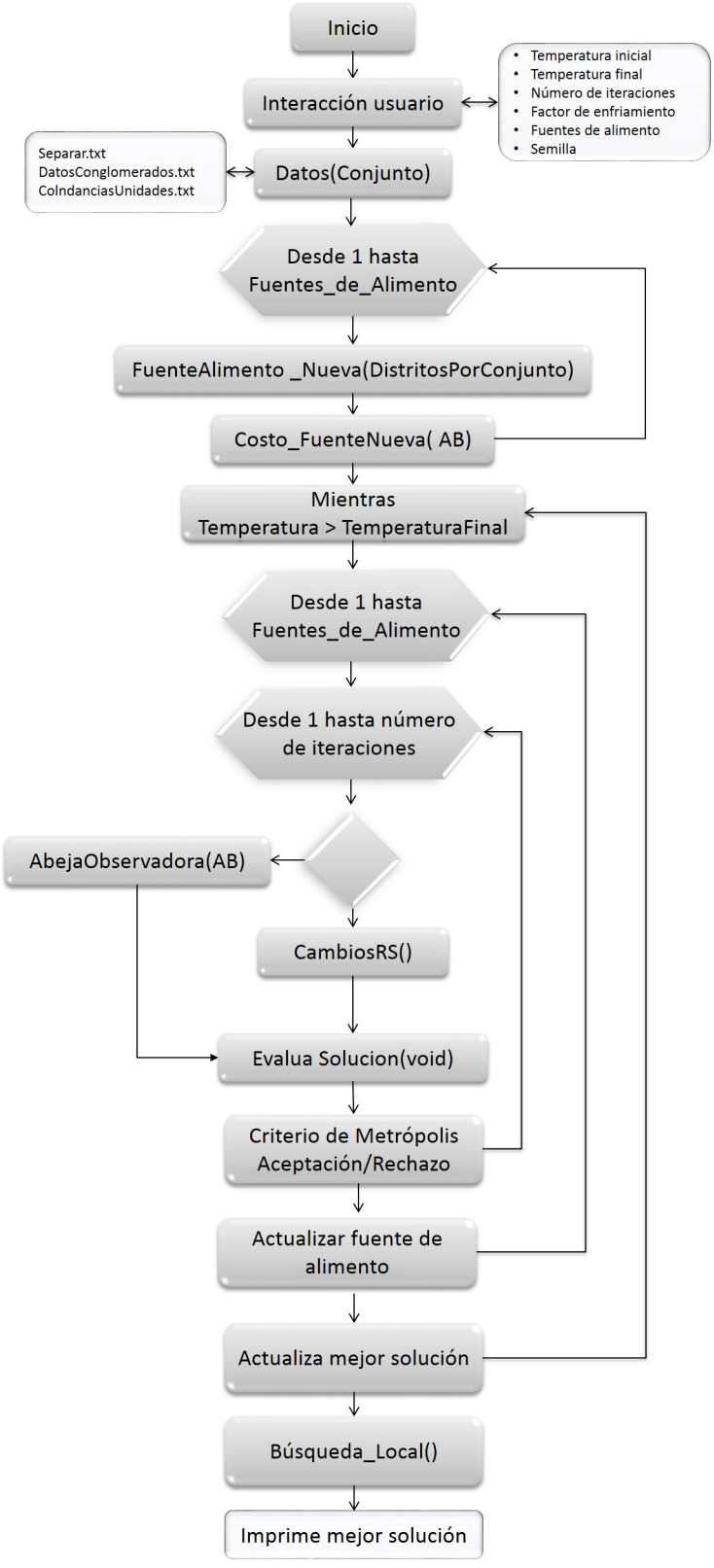


**Figura 5.1** Diagrama del algoritmo ABC-RS.

# CAPÍTULO 6

# DESCRIPCIÓN TÉCNICA DEL ALGORITMO ABC-RS

En este capítulo se describe la forma en que operan cada una de las funciones del algoritmo ABC-RS empleado en el sistema de distritación electoral 2016. Primero se presentan las variables globales empleadas en el algoritmo, junto con una descripción breve del uso que se hace de ellas, en las secciones restantes se describen las funciones usadas por el algoritmo.

## 6.1. Variables globales

En esta sección, en la Tabla 6.1, se presentan las variables globales más importantes empleadas por este algoritmo. En la primera columna se coloca el nombre de la variable, la segunda columna indica el tipo de variable empleado, y en la tercera columna se hace una pequeña descripción del uso que se le da durante la ejecución del algoritmo.

| Variable | Tipo | Descripción |
|---|---|---|
| Distrito | **int[]** | Arreglo con la asignación actual de cada unidad geográfica a un distrito. |
| AreaDistrito<br>PerimetroDistrito | **double[]** | Indican el área y perímetro por distrito de la solución guardada en la variable $Distrito[]$. |
| PoblacionDistrito | **int[]** | Indica la cantidad de habitantes por distrito de la solución en la variable $Distrito[]$. |
| DesviacionPoblacionalDistrito<br>CompacidadDistrito | **double[]** | Indican la desviación poblacional y compacidad por distrito de la solución guardada en la variable $Distrito[]$. |
| PerimetroFrontera | **double[][]** | Indica el perímetro compartido por dos unidades geográficas colindantes. |
| PoblacionUnidadGeografica | **int[]** | Guardan la cantidad de habitantes en cada unidad geográfica. |
| AreaUnidadGeografica | **double[]** | Guarda el área de cada unidad geográfica. |
| Vecinos | **int[][]** | Indica las unidades geográficas colindantes. |
| Semilla | **long** | Guarda el valor de la semilla propuesta por el usuario. Un número entero entre 0 y $2^{32} - 1$ |
| Fuentes_de_Alimento | **int** | Permite almacenar hasta 200 soluciones o fuentes de alimento. |
| Costo_FuenteAlimento<br>Compacidad_FuenteAlimento<br>DesviacionPoblacional_Fuen teAlimento | **double[]** | Almacenan el costo total, el costo por compacidad y el costo por desviación poblacional de las fuentes de alimento. |

| | | |
|---|---|---|
| Costo_Nueva<br>DesviacionPoblacional_Nueva<br>Compacidad_Nueva | **double** | Almacenan el costo total, el costo por compacidad y el costo por desviación poblacional de la solución guardada en la variable $Distrito[]$. |
| NConjuntos<br>ConjuntoActual<br>UnidadesPorConjunto<br>NDistritos | **int** | Indican el número de conjuntos territoriales en el estado, conjunto territorial que se está optimizando, el número de unidades geográficas que lo forman, el número de distritos que deben generarse en él. |
| Conversion | **int[]** | Asigna un identificador a cada unidad geográfica, que se usará durante el proceso de optimización. |
| DistritosFinales | **int[]** | Guarda la solución final generada por el algoritmo. |
| Unidades_Cambiadas | **int[]** | Guarda el indicador de las unidades geográficas que se han cambiado para generar una solución nueva. |
| Distrito_Destino<br>Distrito_Origen | **int** | Indican los distritos que son modificados para generar una solución nueva. |
| PoblacionDistrito_Origen<br>PoblacionDistrito_Destino | **int** | Indican el número de habitantes en los distritos que se modifican al construir una solución nueva. |
| DesviacionPoblacional_Origen<br>DesviacionPoblacional_Destino<br>AreaDistrito_Origen<br>AreaDistrito_ Destino<br>PerimetroDistrito_Origen<br>PerimetroDistrito_Destino<br>CompacidadDistrito_Origen<br>CompacidadDistrito_Destino | **double** | Indican los costos de los distritos que se modifican al construir una solución nueva. |

| | | |
|---|---|---|
| RS_Soluciones<br>Soluciones_Iniciales | **int[][]** | Permiten almacenar hasta 200 soluciones generadas mediante RS, y 200 soluciones iniciales. |
| RS_Costos<br>RS_Compacidad<br>RS_DesviacionPoblacional | **double** | Indican el costo total, el costo de compacidad y el cotos desviación poblacional de las soluciones generadas mediante RS. |
| Costo_Iniciales | **int[]** | Indica el costo de las soluciones iniciales. |
| MediaEstatal | **int[]** | Guarda la media poblacional del estado. |
| ConjuntosTotales | **int[]** | Guarda el número de distritos que deben construirse. |
| VecindarioRS | **int** | Indica el tipo de vecindario que debe usar RS. |
| Pronostico_Vecindario | **int[]** | Indica que tan eficiente ha sido cada vecindario. |

**Tabla 6.1** Variables globales de ABC-RS.

En las siguientes secciones se presentan y describen las funciones más importantes empleadas por el algoritmo basado en recocido simulado.

## 6.2. Función main()

La función main() inicia con la asignación de valores para algunas de las variables, tanto globales como locales, empleadas durante la ejecución del algoritmo. Las variables locales más importantes de esta función se presentan en la Tabla 6.2.

Nuevamente, en la primera columna se coloca el nombre de la variable, la segunda columna indica el tipo de variable empleado, y en la tercera columna se hace una pequeña descripción del uso que se le da durante la ejecución del algoritmo.

| Variable | Tipo | Descripción |
|---|---|---|
| Numero_de_Semillas | **int** | Cuenta el número de semillas en el semillero para indicar el número de corridas que deben realizarse. Una corrida por cada semilla. Cuando el usuario da una semilla esta variable toma el valor de 1. |
| Semillas | **int[]** | Guarda hasta 1000 semillas obtenidas del semillero o bien la semilla dada por el usuario. |
| MejorDesviacionPoblacional<br>MejorCompacidad | **double[]** | Almacenan el costo de cada distrito del escenario construido por el algoritmo después de una corrida. |
| Menor_DesviacionPoblacional<br>Menor_Compacidad | **double** | Almacenan el costo de la mejor fuente de alimento visitada por el algoritmo. |
| Calidad_FuenteAlimento | **double[]** | Indica la calidad de cada una de las fuentes de alimento |
| MejoresDistritos | **int[]** | Guarda la mejor solución encontrada por el algoritmo durante cada iteración. |
| GeneracionesSinMejora | **int[]** | Indica el número de generaciones consecutivas sin mejora, para cada fuente de alimento. |
| Solucion_Hibrida | **int[]** | Guarda la mejor solución para cada conjunto territorial encontrada por el algoritmo. |
| Mejora_Hibrida | **int[]** | Indica si la solución híbrida ha sido actualizada. |
| DesviacionPoblacional_Hibrida<br>Compacidad_Hibrida | **double[]** | Guardan los costos de la solución híbrida construida por el algoritmo. |
| Precalentado | **int[]** | Indica si debe elevarse o disminuirse la temperatura del algoritmo. |
| Aceptacion_Promedio | **int[]** | Indica el porcentaje promedio de soluciones de mala calidad aceptadas. |

| | | |
|---|---|---|
| Temperatura<br>TemperaturaFinal<br>TemperaturaInicial | **double** | Indican la temperatura actual del sistema, la temperatura en la que termina la ejecución del algoritmo, y la temperatura proporcionada por el usuario, respectivamente. |
| EquilibrioFinal | **int** | Determina el número de iteraciones que debe hacer el algoritmo en cada temperatura. |
| FactorEnfriamiento_Caliente<br>FactorEnfriamiento_Templado<br>FactorEnfriamiento_Frio | **double** | Indican el factor de enfriamiento a diferentes niveles de aceptación del algoritmo. |
| Iteraciones_Caliente<br>Iteraciones _Templado<br>Iteraciones_Frio | **int** | Indican el número de iteraciones que se realizará a diferentes niveles de aceptación. |

**Tabla 6.2** Variables locales de la función main.

La función inicia asignando el valor de los parámetros que el algoritmo empleará durante su ejecución:

- TemperaturaInicial
- TemperaturaFinal
- FactorEnfriamiento_Caliente
- FactorEnfriamiento_Templado
- FactorEnfriamiento_Frio
- Iteraciones_Caliente
- Iteraciones_Templado
- Iteraciones_Frio
- Fuentes_de_Alimento

El uso de los factores Caliente, Templado y Frío dependerá del nivel de aceptación promedio, el cual se calcula como el cociente del número de soluciones de mala calidad aceptadas entre el número de soluciones de mala calidad visitadas.

Los factores Calientes se usan cuando el nivel de aceptación es mayor que 0.40, los factores Templados se usan cuando el nivel de aceptación está entre 0.20 y 0.40, y los factores Fríos se emplean cuando el nivel de aceptación es menor que 0.20.

Posteriormente se revisa el valor asignado a la variable Semilla. Si el usuario coloca un valor para esta variable, se realizará una corrida empleando esta semilla para iniciar el generador de números aleatorios. Si el usuario elige la opción semillero se lee el archivo de texto Semillero.txt, y se realiza una corrida por cada una de las semillas encontradas en el archivo. Es importante destacar que el algoritmo está diseñado para leer hasta 1000 semillas. En caso de que el número de semillas en el archivo Semillero.txt sea mayor, sólo se considerarán las primeras 1000 semillas.

El siguiente paso consiste en realizar la lectura del archivo de texto ConjuntosDistritos.txt para determinar cuántos distritos se deberán crear en cada conjunto territorial, y se llama a la función Datos(int Conjunto) para obtener la información de cada unidad geográfica y poder emplearla durante el resto de la ejecución.

Cuando se ha obtenido la información de cada unidad geográfica, se inicia la construcción y optimización de distritos para cada conjunto territorial por separado.

Para cada conjunto territorial se repiten los siguientes pasos.

**ABC-RS1** Se utiliza la función FuenteAlimento_Nueva, para crear de forma aleatoria, tantas soluciones como lo indique la variable $Fuente\_de\_Alimento$. Las soluciones generadas son almacenadas en las variables $Costo\_Iniciales$, $RS\_Costos$ y $Costo\_FuenteAlimento$

**ABC-RS2** Se evalúa el costo de cada una de las soluciones creadas mediante el uso de la función Costo_FuenteNueva.

**ABC-RS3** Se inicia el proceso de mejora mediante un ciclo que durará hasta que la temperatura llegue a la temperatura final. Durante este ciclo se realizan los siguientes pasos:

- **3.1** La función Recocido_Simulado es aplicada a cada una de las fuentes de alimento. Esta función modifica la solución guardada en $RS\_Soluciones$. Cuando se mejora el menor costo conocido se actualiza la variable $MejoresDistritos$.
- **3.2** Si la aceptación promedio es mayor a 0.2 se aplica la función Recocido_Simulado2 a la variable $Soluciones\_Iniciales$. Cuando se encuentra una solución que mejora el menor costo conocido se actualiza la variable *MejoresDistritos*.

**ABC-RS4** Se actualizan el número de iteraciones y el factor de enfriamiento dependiendo del la variable $Aceptacion\_Promedio$

**ABC-RS5** Se actualiza la variable *Temperatura*

Cuando cada uno de los conjuntos territoriales han sido procesados mediante los pasos **ABC-RS1**-**ABC-RS5** se da por concluida una corrida del algoritmo. Cuando el algoritmo alcanza la temperatura final indicada por el usuario, se realiza una exploración en el vecindario de la mejor solución encontrada, mediante el uso de la función Busqueda_Local. La unión de las soluciones obtenidas para cada conjunto territorial se convierte en el escenario final. Es importante insistir en que se realizará una corrida por cada semilla dada al algoritmo.

Si el usuario propuso el valor para la variable Semilla, sólo se genera un escenario que es devuelto después de la primera corrida. Si el usuario eligió la opción Semillero, entonces se generarán tantos escenarios como semillas se tengan. Además, al concluir todas las corridas se devolverá la mejor distritación encontrada al combinar los mejores distritos para cada conjunto territorial, de cada uno de los escenarios finales obtenidos.

El pseudocódigo de la función main se presenta en el Algoritmo 21.

**Algoritmo 21:** Pseudocódigo de la función main

```
1  Solicitar valores de parámetros al usuario.
2  Leer el archivo ConjuntosDistritos.txt, y llamar a la función Datos(Conjunto).
3  Para cada conjunto territorial hacer
4      Crear las fuentes de alimento con la función FuenteAlimento_Nueva(NDistritos), y evaluar el costo de cada
       fuente de alimento con la función Costo_FuenteNueva(AB).
5      Mientras Temperatura > TemperaturaFinal hacer
6          Para i = 1 hasta Fuentes_de_Alimento hacer
7              Modificar RS_Soluciones[i] para obtener una solución nueva mediante la función
               Recosido_Simulado(Temperatura, EquilibrioFinal, AB).
8              Evaluar la calidad de la solución nueva mediante la función Costo_Nueva_Solucion(Origen,
               Destino).
9              Reemplazar la variable MejoresDistritos si su costo es mejorado
10             Si la aceptación promedio es mayor a 0.2 aplicar la función Recocido_Simulado2 a la variable
               Soluciones_Iniciales[i].
11             Reemplazar la variable MejoresDistritos si se encuentra una solución de menor costo.
12         fin
13         Actualizar las variables EquilibrioFinal, alfa, y Temperatura
14     fin
15     Guardar en memoria la mejor solución encontrada.
16 fin
17 Juntar las soluciones encontradas para cada conjunto y devolverlas como mejor distritación encontrada para el
   estado.
```

## 6.3. Función Datos(int Conjunto)

La función Datos($Conjunto$) recibe como parámetro el conjunto territorial para el cual se va a iniciar el proceso de optimización. Esta función se emplea para leer tres archivos de texto en los cuales se encuentra la información de las secciones necesaria para la construcción y optimización de distritos del conjunto territorial indicado: Separar.txt, DatosConglomerados.txt, ColindanciasUnidades.txt.

El archivo de texto Separar.txt está formado por dos columnas, con los identificadores de los municipios que deben considerarse como no vecinos por tiempos de traslado. Esta información se guarda en la variable local $Separar[][]$ de tipo entero.

El archivo de texto DatosConglomerados.txt contiene la siguiente información de cada sección: Municipio, número de sección, área, población, conglomerado al que pertenece y conjunto territorial al que pertenece. Esta información es guardada en variables que repre-

sentan la cantidad de habitantes, y el área de cada unidad geográfica, como se muestra en la Tabla 6.3.

| Variable | Tipo | Dato almacenado |
|---|---|---|
| PoblacionUnidadGeografica | **int[]** | Cantidad de habitantes en cada conglomerado |
| AreaUnidadGeografica | **double[]** | Área de cada conglomerado |

**Tabla 6.3** Variables empleadas para los conglomerados

El archivo de texto ColindanciaUnidades.txt contiene para cada sección el número de la sección con la cual colinda y el perímetro que comparten en dicha colindancia, en la Tabla 6.4 se muestra como ejemplo las colindancias de una sección hipotética vecina de cuatro secciones.

| Sección A | Sección B | Perímetro de colindancia |
|---|---|---|
| 1 | 2 | 1703 |
| 1 | 3 | 1498.1 |
| 1 | 15 | 468.9 |
| 1 | 16 | 1018.1 |

**Tabla 6.4** Colindancias de una sección.

Esta información es utilizada para determinar las unidades geográficas que son vecinas entre sí, y el perímetro de colindancia que comparten. Esta información es guardada en las variables $Vecinos[][]$ y $PerimetroFrontera[][]$.

Es importante mencionar que en este punto se usan los valores almacenados en la variable $Separar[]$, para determinar cuándo dos unidades geográficamente colindantes no deben considerarse como vecinas debido a tiempos de traslado.

## 6.4. Función FuenteAlimento_Nueva(int DistritosPorConjunto)

La función FuenteAlimento_Nueva($DistritosPorConjunto$) recibe como parámetro de entrada el número de distritos que debe generar, $DistritosPorConjunto$. Para generar $r$ distritos primero se eligen de forma aleatoria $r$ unidades geográficas, y cada unidad es

asignada a un distrito diferente.

Después, se repiten los siguientes pasos hasta que cada unidad geográfica ha sido asignada a exactamente un distrito:

**FA1** Elegir de manera aleatoria un distrito, $Distrito_i$.

**FA2** Hacer una lista con las unidades geográficas que colindan con $Distrito_i$, y que aún no han sido asignadas a un distrito.

**FA3** Elegir de forma aleatoria una de estas unidades geográficas y se agrega a $Distrito_i$.

**FA4** Marcar la unidad geográfica seleccionada como ya asignada.

Los pasos **FA1-FA4** se repiten hasta que toda unidad geográfica ha sido asignada en algún distrito. De esta forma, por construcción, toda solución inicial esta formada por $r$ distritos conexos.

El pseudocódigo de la función FuenteAlimento_Nueva se presenta en el Algoritmo 22.

**Algoritmo 22:** Pseudocódigo de la función FuenteAlimento_Nueva

```
1 Se eligen de forma aleatoria r unidades geográficas y cada una se asigna a un distrito diferente.
2 Mientras queden unidades geográficas sin asignar hacer
3     Para cada distrito hacer
4         Crear una lista con las unidades geográficas colindantes que aún no han sido asignadas.
5         Elegir una unidad geográfica de la lista formada.
6         Asignar la unidad geográfica al distrito.
7         Marcar la unidad geográfica como ya asignada.
8     fin
9 fin
```

## 6.5. Función Costo_FuenteNueva(int AB)

La función Costo_FuenteNueva($AB$) recibe como parámetro el identificador de una fuente de alimento, $AB$. Calcula el número de habitantes, área y perímetro de los distritos, tomando en cuenta la información de las unidades geográficas que forman a cada distrito en la solución $AB$. Estos datos son almacenados en las variables $PoblacionDistrito[]$, $AreaDistrito[]$ y $PerimetroDistrito[]$.

Con estas variables se puede cuantificar la desviación poblacional y la compacidad geométrica mediante las funciones Desviacion_Poblacional y Compacidad respectivamente. Los costos totales para cada fuente de alimento son guardados en las variables $Desviacion Poblacional\_FuenteAlimento[AB]$ y $Compacidad\_FuenteAlimento[AB]$.

El pseudocódigo de la función Costo_FuenteNueva se presenta en el Algoritmo 23.

**Algoritmo 23:** Pseudocódigo de la función Costo_FuenteNueva

```
Para cada Distrito_i, 1 ≤ i ≤ r hacer
    Calcular población de Distrito_i.
    Calcular área de Distrito_i.
    Calcular perímetro Distrito_i.
fin
Para cada Distrito_i, 1 ≤ i ≤ r hacer
    Desviacion_Poblacional(PoblacionDistrito[]).
    Compacidad(AreaDistrito[], PerimetroDistrito[]).
fin
```

### 6.6. Función RevisaConexidad_Empleada(int Origen, int Destino)

La función RevisaConexidad_Empleada($Origen, Destino$) recibe como parámetros los identificadores de dos distritos, $Origen$ y $Destino$, y cuenta el número de componentes conexas, $N$, que tiene el distrito $Origen$.

Si $N = 1$, significa que el distrito $Origen$ es conexo y la función termina.

Si $N \geq 1$, significa que el distrito $Origen$ es disconexo y debe repararse.

En este caso busca a la componente conexa que tenga el mayor número de unidades geográficas, y es conservada como el distrito $Origen$. Las unidades geográficas en otras componentes conexas son enviadas al distrito $Destino$ mediante la función ReparaConexidad_Empleada.

El pseudocódigo de la función RevisaConexidad_Empleada se presenta en el Algoritmo 24.

**Algoritmo 24:** Pseudocódigo de la función RevisaConexidad_Empleada

```
1  Sea NU el número de unidades geográficas en el distrito Origen.
2  Seleccionar de forma aleatoria una unidad geográfica, k_1, en distrito Origen.
3  Construir la ComponenteConexa_{k_1} de distrito Origen que contiene a k_1.
4  si |ComponenteConexa_{k_1}| = NU entonces
5  |   El distrito Origen es conexo.
6  fin
7  si |ComponenteConexa_{k_1}| < NU entonces
8  |   El distrito Origen tiene más de una componente conexa.
9  |   Mientras alguna unidad no se encuentre en una componente hacer
10 |   |   Seleccionar de forma aleatoria una unidad geográfica, k_i, que aún no esté en ninguna componente.
11 |   |   Construir la ComponenteConexa_{k_i} que contiene a k_i.
12 |   fin
13 |   El nuevo distrito Origen es la componente con mayor número de unidades, denominada
   |   ComponenteConexa_{k_j}
14 |   Para k_i ≠ k_j hacer
15 |   |   ReparaConexidad_Empleada(Origen, k_i, Destino).
16 |   fin
17 fin
```

## 6.7. Función ReparaConexidad_Empleada(int Origen, int Unidad, int Destino)

La función ReparaConexidad_Empleada($Origen, Unidad, Destino$) es llamada cuando se ha comprobado disconexión en un distrito. La función ReparaConexidad_Empleada recibe tres parámetros, el identificador del distrito que perdió la conexidad, $Origen$, el identificador de una unidad geográfica, $k$, que actualmente se encuentra en el distrito $Origen$, y el identificador de un distrito vecino, $Destino$.

La función visita a todas las unidades geográficas vecinas de $k$, y construye de forma creciente una componente conexa del distrito $Origen$ que contiene a $k$. Después, todas las unidades geográficas en esta componente conexa son enviadas al distrito $Destino$.

De esta forma, el distrito $Origen$ tiene una componente conexa menos.

El pseudocódigo de la función ReparaConexidad_Empleada se presenta en el Algoritmo 25.

**Algoritmo 25:** Pseudocódigo de la función ReparaConexidad_Empleada

```
Sean ComponenteConexa y Lista dos arreglos.
Agregar a k en ComponenteConexa y en Lista.
Para cada unidad geográfica i en Lista hacer
    Visitar a los vecinos de i.
    si una unidad vecina está en el distrito Origen entonces
        Agregarla a ComponenteConexa y a Lista.
    fin
fin
Cambiar a todas las unidades geográficas en ComponenteConexa al distrito Destino
```

### 6.8. Función AbejaObservadora(int AB)

La función AbejaObservadora($AB$) recibe como parámetro el identificador de una fuente de alimento, $AB$, y modifica la solución $Distrito[]$ al quitar y agregar unidades geográficas de un distrito mediante los siguientes pasos.

**ABO1** Selecciona de forma aleatoria un distrito, $OrigenAB$, de la solución $Distrito[]$.

**ABO2** Selecciona de forma aleatoria una unidad geográfica, $k$, en el distrito $Origen$.

**ABO3** Genera de forma aleatoria un número $AB1$ entre 0 y $Fuentes\_de\_Alimento$ - 1. Con probabilidad 0.5 elige entre las soluciones $Soluciones\_Iniciales[AB1][]$ y $Fuente-Alimento[Fuentes\_de\_Alimento][]$. La solución seleccionada será combinada con $Distrito[]$.

**ABO4** Se identifica al distrito al que pertenece la unidad $k$ en la solución $Distrito[]$, sea $OrigenAB1$.

**ABO5** Se elige de forma aleatoria una unidad geográfica que esté en la frontera del distrito $OrigenAB$, pero que no esté en $OrigenAB1$, y se cambia a un distrito vecino.

**ABO6** Se revisa si el distrito $OrigenAB$ perdió la conexidad mediante la función RevisaConexidad_Observadora1, de ser así se repara con la función ReparaConexidad_Observadora.

**ABO7** Se elige de forma aleatoria una unidad geográfica que colinde con el distrito $OrigenAB$, y que pertenezca al distrito $OrigenAB1$, y se cambia al distrito $OrigenAB$.

**ABO8** Se revisa si los distritos modificados perdieron la conexidad mediante la función RevisaConexidad_Observadora2, de ser así se reparan con la función ReparaConexidad_Observadora.

**ABO9** Si el costo de la nueva solución es mejor que el costo de $Distrito[]$, los cambios se aceptan, en otro caso los cambios se rechazan.

El pseudocódigo de la función AbejaObservadora se presenta en el Algoritmo 26.

**Algoritmo 26:** Pseudocódigo de la función AbejaObservadora

1 Sea $Distrito[]$ una solución.
2 Seleccionar de forma aleatoria un distrito, $OrigenAB$.
3 Seleccionar de forma aleatoria una unidad geográfica, $k$, del distrito $Origen$.
4 Seleccionar de forma aleatoria un número $AB1$ entre 1 y Fuentes_de_Alimento - 1.
5 Seleccionar de forma aleatoria entre las soluciones $Soluciones\_Iniciales[AB1][]$ y $FuenteAlimento[Fuentes\_de\_Alimento][]$. La solución seleccionada se combinará con $Distrito[]$.
6 Sea $OrigenAB1$ el distrito al que pertenece la unidad $k$ en la solución $AB1$.
7 Elegir de forma aleatoria una unidad geográfica en la frontera de $OrigenAB$, que no esté en $OrigenAB1$, y cambiarla a un distrito vecino.
8 Revisar si el distrito $OrigenAB$ perdió la conexidad, de ser así se repara.
9 Elegir de forma aleatoria una unidad geográfica en $OrigenAB1$ que colinde con $OrigenAB$, y cambiarla al distrito $OrigenAB$.
10 Revisar si los distritos modificados perdieron la conexidad, de ser así se reparan.
11 Si el costo de la nueva solución es mejor que el costo de $AB$, los cambios se aceptan, en otro caso los cambios se rechazan.

### 6.9. Función RevisaConexidad_Observadora1(int DistritoAnalizado)

La función RevisaConexidad_Observadora1($DistritoAnalizado$) recibe como parámetro el identificador de un distrito, $DistritoAnalizado$, y cuenta el número de componentes conexas, $N$, que tiene este distrito.

Si $N = 1$, significa que $DistritoAnalizado$ es conexo y la función termina.

Si $N \geq 1$, significa que $DistritoAnalizado$ es disconexo y debe repararse.

En este caso busca a la componente conexa que tenga el mayor número de unidades geográficas, y es conservada como $DistritoAnalizado$. Las unidades geográficas en otras componentes conexas son enviadas a distritos vecinos mediante la función ReparaConexidad_Observadora.

El pseudocódigo de la función RevisaConexidad_Observadora1 se presenta en el Algoritmo 27.

**Algoritmo 27:** Pseudocódigo de la función RevisaConexidad_Observadora1

```
1  Sea NU el número de unidades geográficas en DistritoAnalizado.
2  Seleccionar de forma aleatoria una unidad geográfica, k_1, en DistritoAnalizado.
3  Construir la ComponenteConexa_{k_1} de DistritoAnalizado que contiene a k_1.
4  si |ComponenteConexa_{k_1}| = NU entonces
5  |   DistritoAnalizado es conexo.
6  fin
7  si |ComponenteConexa_{k_1}| < NU entonces
8  |   DistritoAnalizado tiene más de una componente conexa.
9  |   Mientras alguna unidad no se encuentre en una componente hacer
10 |   |   Seleccionar de forma aleatoria una unidad geográfica, k_i, que aún no esté en ninguna componente.
11 |   |   Construir la ComponenteConexa_{k_i} que contiene a k_i.
12 |   fin
13 |   El nuevo DistritoAnalizado es la componente con mayor número de unidades, digamos
   |   ComponenteConexa_{k_j}
14 |   Para k_i ≠ k_j hacer
15 |   |   ReparaConexidad_Observadora(DistritoAnalizado, k_i).
16 |   fin
17 fin
```

## 6.10. Función RevisaConexidad_Observadora2(int DistritoOrigen, int UnidadOrigen)

La función RevisaConexidad_Observadora2($DistritoOrigen, UnidadOrigen$) recibe como parámetros el identificador de un distrito, $DistritoOrigen$, y el identificador de una unidad geográfica, $UnidadOrigen$, y cuenta el número de componentes conexas, $N$, que tiene $DistritoOrigen$.

Si $N = 1$, significa que $DistritoOrigen$ es conexo y la función termina.

Si $N \geq 1$, significa que $DistritoOrigen$ es disconexo y debe repararse.

En este caso busca a la componente conexa que contenga a la unidad geográfica $Unidad-Origen$, y es conservada como $DistritoOrigen$. Las unidades geográficas en otras componentes conexas son enviadas a distritos vecinos mediante la función ReparaConexidad_Observadora.

El pseudocódigo de la función RevisaConexidad_Observadora2 se presenta en el Algoritmo 28.

**Algoritmo 28:** Pseudocódigo de la función RevisaConexidad_Observadora2

```
1  Sea NU el número de unidades geográficas en DistritoOrigen.
2  Seleccionar de forma aleatoria una unidad geográfica, k_1, en DistritoOrigen.
3  Construir la ComponenteConexa_{k_1} de DistritoOrigen que contiene a k_1.
4  si |ComponenteConexa_{k_1}| = NU entonces
5  |   DistritoOrigen es conexo.
6  fin
7  si |ComponenteConexa_{k_1}| < NU entonces
8  |   DistritoOrigen tiene más de una componente conexa.
9  |   Mientras alguna unidad no se encuentre en una componente hacer
10 |   |   Seleccionar de forma aleatoria una unidad geográfica, k_i, que aún no esté en ninguna componente.
11 |   |   Construir la ComponenteConexa_{k_i} que contiene a k_i.
12 |   fin
13 |   El nuevo DistritoOrigen es la componente que contiene a UnidadOrigen, digamos
   |   ComponenteConexa_{k_j}
14 |   Para k_i ≠ k_j hacer
15 |   |   ReparaConexidad_Observadora(DistritoAnalizado, k_i).
16 |   fin
17 fin
```

### 6.11. Función ReparaConexidad_Observadora(int DistritoOrigen, int UnidadOrigen)

La función ReparaConexidad_Observadora($DistritoOrigen, UnidadOrigen$) es llamada cuando se ha comprobado disconexión en un distrito. La función ReparaConexidad_Observadora recibe dos parámetros, el identificador del distrito que perdió la conexidad, $DistritoOrigen$, y el identificador de una unidad geográfica, $UnidadOrigen$.

La función visita a todas las unidades geográficas vecinas de $UnidadOrigen$, y construye de forma creciente una componente conexa de $DistritoOrigen$ que contiene a $UnidadOrigen$. Después, todas las unidades geográficas en esta componente conexa son enviadas a un distrito vecino elegido de forma aleatoria.

De esta forma, $DistritoOrigen$ tiene una componente conexa menos.

El pseudocódigo de la función ReparaConexidad_Observadora se presenta en el Algoritmo 29.

**Algoritmo 29:** Pseudocódigo de la función ReparaConexidad_Observadora

```
1  Sean ComponenteConexa, Lista y Destinos tres arreglos.
2  Agregar a UnidadOrigen en ComponenteConexa y en Lista.
3  Para cada unidad geográfica i en Lista hacer
4  |   Visitar a los vecinos de i.
5  |   si la unidad vecina está en DistritoOrigen entonces
6  |   |   Agregarla a ComponenteConexa y a Lista.
7  |   fin
8  |   si la unidad vecina no está en DistritoOrigen entonces
9  |   |   Agregar el distrito de la unidad vecina a Destinos.
10 |   fin
11 fin
12 Elegir de forma aleatoria un distrito de la lista Destinos.
13 Cambiar a todas las unidades geográficas en ComponenteConexa al distrito elegido
```

### 6.12. Función Evalua_Solucion(void)

La función Evalua_Solucion() calcula el número de habitantes, área y perímetro de los distritos almacenados en la variable $Distrito[]$, tomando en cuenta la información de las unidades geográficas que forman a cada distrito en esta solución. Estos datos son almacenados en las variables $PoblacionDistrito[]$, $AreaDistrito[]$ y $PerimetroDistrito[]$.

Con estas variables se puede cuantificar la desviación poblacional y la compacidad geométrica mediante las funciones Desviacion_Poblacional y Compacidad respectivamente. Los costos totales para esta solución son guardados en las variables $DesviacionPoblacional\_Nueva$ y $Compacidad\_Nueva$.

El pseudocódigo de la función Evalua_Solucion se presenta en el Algoritmo 30.

**Algoritmo 30:** Pseudocódigo de la función Evalua_Solucion

```
1  Para cada Distrito_i, 1 ≤ i ≤ r hacer
2  |   Calcular población de Distrito_i.
3  |   Calcular área de Distrito_i.
4  |   Calcular perímetro de Distrito_i.
5  fin
6  DesviacionPoblacional_Nueva := 0. Compacidad_Nueva := 0. Para cada Distrito_i, 1 ≤ i ≤ r hacer
7  |   DesviacionPoblacional_Nueva += Desviacion_Poblacional(PoblacionDistrito[]).
8  |   Compacidad_Nueva += Compacidad(AreaDistrito[], PerimetroDistrito[]).
9  fin
```

### 6.13. Función Desviacion_Poblacional(int Poblacion)

La función Desviacion_Poblacional($Poblacion$) recibe como parámetro la cantidad de habitantes en un distrito, $Poblacion$, y devuelve el costo poblacional correspondiente, aplicando la siguiente ecuación:

$$Costo = \left( \frac{1 - \left( \frac{Poblacion}{MediaEstatal} \right)}{0.15} \right)^2 \tag{6.1}$$

Si el valor de la variable $Costo$ es mayor que 1, se considera que se está violando la restricción de no exceder un $\pm 15\,\%$ de desviación poblacional con respecto a la media, y se agrega una penalización dada por la siguiente ecuación:

$$Costo := Costo + 10 * (Costo - 1) \tag{6.2}$$

Finalmente devuelve el valor de $Costo$.

### 6.14. Función Compacidad(double Area,double Perimetro)

La función Compacidad($Area, Perimetro$) recibe como parámetros el área, $Area$, y perímetro, $Perimetro$, de un distrito, y devuelve el costo de compacidad aplicando la siguiente ecuación:

$$Costo = \left( \left( \frac{Perimetro}{\sqrt{Area}} * 0.25 \right) - 1.00 \right) * 0.5 \tag{6.3}$$

Finalmente devuelve el valor de $Costo$.

### 6.15. Función Recocido_Simulado(float Temperatura, int EquilibrioFinal, int AB)

La función Recocido_Simulado recibe como parámetros la temperatura, $Temperatura$, el número de iteraciones que debe realizar, $EquilibrioFinal$, y la fuente de alimento en la cual está trabajando el algoritmo, $AB$. La fuente de alimento $AB$ es copiada en $Distrito[]$, y a partir de ella se construyen soluciones vecinas mediante el uso de las funciones AbejaObservadora(*AB*) y CambioRS(). La solución vecina sustituye a la fuente de alimento $AB$ cuando su costo es menor, de esta forma la fuente de alimento conserva las mejores soluciones vecinas visitadas desde $Distrito[]$. Por otra parte, se determina si la variable $Distrio[]$ es reemplazada por la solución vecina mediante el criterio de

Metrópolis. La calidad de las soluciones vecinas obtenidas, cuando se modifica la variable $Distrito$ mediante la función CambiosRS(), se emplea para actualizar el valor de la variable $Pronostico\_Vecindario[]$. Cuando una solución vecina es de mejor calidad se le asigna un valor más alto al vecindario empleado, por otro lado, si se obtiene una solución vecina de mala calidad le asigna un valor menor. La variable $Pronostico\_Vecindario[]$ se emplea para elegir, mediante una estrategia tipo ruleta, el vecindario que empleará la función CambiosRS() la próxima vez que sea llamada. Finalmente, la función Recocido_Simulado devuelve el nivel de aceptación de soluciones de mala calidad.

El pseudocódigo de la función Recocido_Simulado se presenta en el Algoritmo 31.

**Algoritmo 31:** Pseudocódigo de la función Recocido_Simulado

```
1  Distritos[] ⟵ FuenteAlimento[AB][]
2  Mientras Equilibrio ≤ EquilibrioFinal hacer
3      si Equilibrio %5==0 entonces
4          AbejaObservadora(AB).
5      fin
6      else
7          CambioRS().
8      fin
9      si solucion vecina es mejor que FuenteAlimento[AB][] entonces
10         FuenteAlimento[AB][] ⟵ solucionvecina.
11     fin
12     Usar el criterio de Metrópolis para determinar si la solución vecina reemplaza a Distritos[].
13     si solucion vecina fue creada con CambioRS() entonces
14         Actualizar Pronostico_Vecindario[].
15         si solucion vecina fue rechazada entonces
16             Seleccionar un nuevo vecindario en base al valor dado por Pronostico_Vecindario[].
17         fin
18     fin
19     Equilibrio++.
20 fin
21 Devolver nivel de aceptación de soluciones de mala calidad.
```

### 6.16. Función Recocido_Simulado2(float Temperatura, int AB, int Iteraciones)

La función Recocido_Simulado2 recibe como parámetros la temperatura, $Temperatura$, la fuente de alimento en la cual está trabajando el algoritmo, $AB$, y el número de iteraciones que debe realizar, $Iteraciones$. Al inicio, la variable $Distrito[]$ es igual a la variable $Soluciones_Iniciales[AB][]$. Después se generan soluciones vecinas de la variable $Distrito[]$ mediante llamadas a la función CambioRS(). Se decide, mediante el criterio de Metrópolis, si la solución vecina generada reemplaza a la variable $Soluciones_Iniciales[AB][]$.

Este proceso se repite durante el número de iteraciones indicado. El pseudocódigo de la función Recocido_Simulado se presenta en el Algoritmo 32.

**Algoritmo 32:** Pseudocódigo de la función Recocido_Simulado2

```
1 Distritos[] ⟵ Soluciones_Iniciales[AB][]
2 Mientras Equilibrio ≤ Iteraciones hacer
3 |   CambioRS().
4 |   Usar el criterio de Metrópolis para determinar si la solución vecina reemplaza a Soluciones_Iniciales[AB][].
5 |   Equilibrio++.
6 fin
```

### 6.17. Función CambioRS(void)

La función CambiosRS() modifica la solución actual al cambiar de distrito algunas unidades geográficas, para lo cual se realizan los siguientes pasos.

**C1** Se elige de forma aleatoria un distrito que contenga al menos dos unidades geográficas, $DistritoOrigen$.

**C2** Se hace una lista, $L$, con todas las unidades geográficas, de $DistritoOrigen$, que colinden con otro distrito dentro del mismo conjunto territorial.

**C3** Se elige de forma aleatoria una de las unidades geográficas, $UG_1$, dentro de la lista *L*.

**C4** La unidad geográfica $UG_1$ es cambiada a un distrito con el cual colinde, $DistritoDestino$. En caso de haber más de una opción se elige una de forma aleatoria.

**C5** Si se está empleando el vecindario 2, y $DistritoOrigen$ tiene suficientes unidades geográficas, se elije de forma aleatoria una unidad geográfica en este distrito, $UG_2$, que colindante con $UG_1$, y se cambia al distrito $DistritoDestino$.

**C5** Si se está empleando el vecindario 3, y $DistritoOrigen$ tiene suficientes unidades geográficas, se elije de forma aleatoria una unidad geográfica en este distrito, que colindante con $UG_1$ o $UG_2$, y se cambia al distrito $DistritoDestino$.

**C5** Se revisa si se ha perdido la conexidad del distrito mediante la función Revisa_Conexidad. En caso de ser así, deberá repararse con la función Repara_Conexidad.

La solución obtenida después de hacer estas modificaciones es evaluada mediante la función Costo_Nueva_Solucion. Los costos de la nueva solución son almacenados en las

variables $DesviacionPoblacional\_Nueva$ y $Compacidad\_Nueva$.

El pseudocódigo de la función Cambios se presenta en el Algoritmo 33.

**Algoritmo 33:** Pseudocódigo de la función CambiosRS()

1 Elegir de forma aleatoria un distrito con al menos dos unidades geográficas, $Distrito_i$.
2 Hacer una lista $L$ con las unidades geográficas de $Distrito_i$ que colinden con otro distrito del mismo conjunto.
3 Elegir aleatoriamente una unidad geográfica, $UG$, de $L$.
4 Enviar la unidad $UG$ a un distrito vecino elegido de forma aleatoria, $Distrito_j$.
5 En caso de ser posible, y dependiendo del vecindario empleado, enviar 1 o 2 UG adicionales del distrito $Distrito_i$ al distrito $Distrito_j$.
6 Revisar la conexidad de $Distrito_i$, en caso de ser necesario, deberá repararse.
7 Evaluar la solución obtenida con el cambio de $UG$.

## 6.18. Función Cardinalidad_DistritoRS(int Distritos)

La función Cardinalidad_DistritoRS recibe como parámetro el identificador del distrito analizado, $Distritos$, y devuelve el número de unidades geográficas que lo forman de acuerdo a la solución $Distritos[]$.

El pseudocódigo de la función Cardinalidad_DistritoRS se presenta en el Algoritmo 34.

**Algoritmo 34:** Pseudocódigo de la función Cardinalidad_DistritoRS

```
1 Cardinalidad := 0.
2 Para i = 1 hasta número de unidades geográficas hacer
3 |   si Distrito[i] == Distritos entonces
4 |   |   Cardinalidad++.
5 |   fin
6 fin
7 Devolver Cardinalidad.
```

## 6.19. Función Busqueda_Local()

La función Busqueda_Local() realiza una búsqueda local en la mejor solución encontrada por el algoritmo para converger a un óptimo local.

**BL1** Se visitan todas las unidades geográficas.

**BL2** Si una unidad geográfica colinda con otro distrito, la unidad geográfica es cambiada al distrito vecino y se evalúa el costo de la nueva solución.

**BL3** Si la nueva solución mejora el costo se acepta el cambio, en caso contrario se deshace el cambio.

La solución obtenida al final de este proceso es un óptimo local.
El pseudocódigo de la función Busqueda_Local se presenta en el Algoritmo 35.

**Algoritmo 35:** Pseudocódigo de la función Busqueda_Local()

```
1 Visitar todas las unidades geográficas.
2 Determinar si la unidad geográfica puede moverse a un distrito vecino.
3 Enviar la unidad UG a cada uno de los distritos con los cuales colinde.
4 Evaluar la solución obtenida con el cambio de UG a a cada uno de los distritos.
5 Aceptar el cambio si mejora el costo de la mejor solución conocida.
```

## 6.20. Función SiguienteAleatorioReal0y1(long * semilla)

La función SiguienteAleatorioReal0y1(* $semilla$) recibe un apuntador a la variable $semilla$. Emplea el valor de esta variable para generar, con una distribución uniforme, un número aleatorio en el intervalo $[0, 1]$. Antes de devolver el número generado modifica el valor de la variable $semilla$.

## 6.21. Función SiguienteAleatorioEnteroModN(long * semilla, int n)

La función SiguienteAleatorioEnteroModN(* $semilla$, $n$) recibe un apuntador a la variable $semilla$ y un número entero $n$. Emplea el valor de a variable $semilla$ para generar, con una distribución uniforme, un número entero aleatorio que se encuentra en el intervalo $[0, n-1]$. Antes de devolver el número generado modifica el valor de la variable $semilla$.

# ANEXO A
# CÓDIGO DEL ALGORITMO BASADO EN RS

### Anexo : Función main()

```
int main()
{
  double z,seed1;
  double CostoTotal;
  double Temperatura, Temperatura_Usuario, TemperaturaFinal,EquilibrioFinal, alfa,b1,u5, u6, Equilibrio;
  int i,j,k,m;
  int Numero_de_Semillas;
  double Entrada,Aceptada;
  int Semillas[1000], Corrida;
  long c;
  double DesviacionPoblacional_EscenarioFinal[45], Compacidad_EscenarioFinal[45];
  int Precalentado;
  int Iteraciones_Caliente, Iteraciones_Templado, Iteraciones_Frio;
  double FactorEnfriamiento_Caliente, FactorEnfriamiento_Templado, FactorEnfriamiento_Frio;
  char dummy[1000];
  int Solucion_Hibrida[6500], Mejora_Hibrida;
  double DesviacionPoblacional_Hibrida[45], Compacidad_Hibrida[45];
  double MenorCostoPoblacional, MenorCompacidad;
  int Distritos_Por_Conjunto[45], Numero_de_Conjuntos;

  FILE *fp;

  //SE ASIGNAN VALORES A ALGUNOS PARAMETROS

  FactorEnfriamiento_Templado = 0.98;
  FactorEnfriamiento_Frio = 0.99;

  //SE LEEN LOS PARAMETROS SOLICITADOS POR EL USUARIO
  char *direccion = "Recocido_Simulado\\Parametros_RS.txt";
  fp = fopen(direccion, "r");
```

```
31  while((c=fgetc(fp))!=EOF)
32  {
33    fscanf(fp,"%f %f %d %f %d",&Temperatura_Usuario, &TemperaturaFinal, &Iteraciones_Caliente, &FactorEnfriamiento_Caliente, &
          Semilla);
34  }
35  fclose(fp);
36
37  Iteraciones_Templado = 2 * Temperatura_Usuario;
38  Iteraciones_Frio = 3 * Temperatura_Usuario;
39
40  //CUANDO LOS REQUERIMIENTOS DEL USUARIO SEAN MAYORES QUE LOS RECOMENDADOS POR
41  //EL SISTEMA SE ACTUALIZARAN LOS PARAMETROS CORRESPONDIENTES
42  if(Iteraciones_Caliente > Iteraciones_Templado)
43        Iteraciones_Templado = Iteraciones_Caliente;
44  if(Iteraciones_Caliente > Iteraciones_Frio)
45        Iteraciones_Frio = Iteraciones_Caliente;
46  if(FactorEnfriamiento_Caliente > FactorEnfriamiento_Templado)
47        FactorEnfriamiento_Templado = FactorEnfriamiento_Caliente;
48  if(FactorEnfriamiento_Caliente > FactorEnfriamiento_Frio)
49        FactorEnfriamiento_Frio = FactorEnfriamiento_Caliente;
50
51  //SI EL USUARIO SELECCIONO EL SEMILLERO SE LEEN LAS SEMILLAS QUE SE UTILIZARAN
52  if(Semilla == -1)
53  {
54    Numero_de_Semillas = 0;
55    direccion = "Sistema_de_visualizacion\\Insumos\\Semillero.txt";
56    fp = fopen(direccion, "r");
57    while((c=fgetc(fp))!=EOF && Numero_de_Semillas < 1000)
58    {
59      fscanf(fp,"%d",&i);
60      Semillas[Numero_de_Semillas] = i;
61      Numero_de_Semillas++;
62    }
63    fclose(fp);
64  }
65  //EN CASO CONTRARIO SOLO SE REALIZARA UNA CORRIDA
66  else
67  {
68    Numero_de_Semillas = 1;
69    Semillas[0] = Semilla;
70  }
71
72  if(Numero_de_Semillas == 1)
73  {
74    printf("\n\n\n\t    Sistema para generar Zonas Electorales 2016\n\n\n");
75    printf("  Se realiza una corrida empleando los siguientes parametros:\n\n");
76    printf("  Temperatura inicial = %f\n\n",Temperatura_Usuario);
77    printf("  Temperatura final = %f\n\n",TemperaturaFinal);
78    printf("  Numero de iteraciones = %d\n\n",Iteraciones_Caliente);
79    printf("  Factor de enfriamiento = %f\n\n",FactorEnfriamiento_Caliente);
80    printf("  Semilla = %d\n\n",Semilla);
81  }
82
83  else
84  {
85    printf("\n\n\n\t    Sistema para generar Zonas Electorales 2016\n\n\n");
86    printf("  Se realizan %d corridas, empleando los siguientes parametros:\n\n", Numero_de_Semillas);
87    printf("  Temperatura inicial = %f\n\n",Temperatura_Usuario);
88    printf("  Temperatura final = %f\n\n",TemperaturaFinal);
89    printf("  Numero de iteraciones = %d\n\n",Iteraciones_Caliente);
90    printf("  Factor de enfriamiento = %f\n\n",FactorEnfriamiento_Caliente);
91    printf("  Semilla = Se usan los valores incluidos en el semillero\n");
92  }
93
94  // SE LEE EL NUMERO DE DISTRITOS ASIGNADOS A CADA CONJUNTO
95  ConjuntosTotales = 0;
96  Numero_de_Conjuntos = 0;
97  direccion = "Sistema_de_visualizacion\\Insumos\\ConjuntosDistritos.txt";
98  fp = fopen(direccion, "r");
99  while((c=fgetc(fp))!=EOF)
100 {
101   fscanf(fp,"%d %d",&k,&i);
102   Distritos_Por_Conjunto[i] = k;
103   Numero_de_Conjuntos++;
104   ConjuntosTotales += k;
105 }
106 fclose(fp);
107
108 for(i=0; i<6500; i++)
109   {
110       Solucion_Hibrida[i] = 6500;
111   }
112
113
114 //SE REALIZAN TANTAS CORRIDAS COMO NUMERO DE SEMILLAS SE ENCUENTREN EN EL SEMILLERO
115 //O SOLO UNA SI EL USUARIO DA LA SEMILLA
116 for(Corrida = 0; Corrida < Numero_de_Semillas; Corrida ++)
117 {
118     //SE CREA UN ARCHIVO PARA GUARDAR LOS COSTOS DE CADA CONJUNTO TERRITORIAL
```

```
119         if(Corrida == 0)
120     {
121         sprintf(dummy, "Sistema_de_visualizacion\\Resumen_Costos\\RS_Costo_Por_Conjunto.csv");
122             fp = fopen(dummy, "w");
123             fprintf(fp,"Semilla %d,Costo Total, Equilibrio Poblacional,Compacidad\n", Semillas[Corrida]);
124             fclose(fp);
125         }
126         if(Corrida > 0)
127     {
128         sprintf(dummy, "Sistema_de_visualizacion\\Resumen_Costos\\RS_Costo_Por_Conjunto.csv");
129             fp = fopen(dummy, "a");
130             fprintf(fp,"Semilla %d,Costo Total, Equilibrio Poblacional,Compacidad\n", Semillas[Corrida]);
131             fclose(fp);
132         }
133
134     for(i=0; i<6500; i++)
135         {
136             DistritosFinales[i] = 6500;
137         }
138
139         Mejora_Hibrida = 0;
140
141     if(Numero_de_Semillas > 1)
142             printf("\n\n\t Inicia la corrida = %d con la Semilla = %d\n",Corrida+1,Semillas[Corrida]);
143
144     Semilla = Semillas[Corrida];
145     //SE INICIA EL TIEMPO DE EJECUCION
146     clock_t start, end;
147     start = clock();
148
149     DistritosAcumulados = 0;
150
151     //SE INICIA EL PROCESO DE CONSTRUCCION DE DISTRITOS PARA CADA CONJUNTO
152     for(ConjuntoActual = 1; ConjuntoActual <= Numero_de_Conjuntos; ConjuntoActual++)
153     {
154       printf("\nConjunto territorial %d. ",ConjuntoActual);
155
156       for(i = 0; i < 6500; i++)
157             {
158                 Conversion[i] = 6500;
159             }
160
161       //LA FUNCION Datos() LEE LA INFORMACION NECESARIA PARA CONSTRUIR LOS DISTRITOS
162       //POR EJEMPLO, COLINDANCIAS, AREA, POBLACION, ETC.
163       Datos(ConjuntoActual);
164       NDistritos = Distritos_Por_Conjunto[ConjuntoActual];
165
166       //SE CONSTRUYE LA SOLUCION INICIAL Y SE EVALUA SU COSTO (PARA EL CONJUNTO EN CURSO)
167       Solucion_Inicial(NDistritos);
168
169       //Distritos_Actuales[i] ES LA SOLUCION ACTUAL DURANTE LA EJECUCION DEL ALGORITMO
170       if(Distritos_Por_Conjunto[ConjuntoActual] == 1)
171                 goto Final;
172       //SE CALCULA EL COSTO DE LA SOLUCION CONSTRUIDA
173       Costo_Solucion_Inicial();
174
175       //LA SOLUCION ACTUAL SE GUARDA COMO LA MEJOR SOLUCION CONOCIDA HASTA EL MOMENTO
176       for(j = 0; j < UnidadesPorConjunto; j++)
177             {
178                 Mejores_Distritos[j] = Distritos_Actuales[j];
179             }
180
181
182       MenorCostoPoblacional = DesviacionPoblacional_Actual;
183       MenorCompacidad = Compacidad_Actual;
184       DesviacionPoblacional_Nueva = DesviacionPoblacional_Actual;
185       Compacidad_Nueva = Compacidad_Actual;
186       Entrada = Aceptada = 0;
187       DesviacionPoblacional_Nueva = DesviacionPoblacional_Actual;
188       Compacidad_Nueva = Compacidad_Actual;
189
190       //INICIA PROCESO DE MEJORA
191       Equilibrio = -1;
192       Temperatura = Temperatura_Usuario;
193       Precalentado = 1;
194       EquilibrioFinal = Iteraciones_Caliente;
195       alfa = FactorEnfriamiento_Caliente;
196
197       while(Temperatura >= TemperaturaFinal)
198       {
199         Equilibrio++;
200         if(Equilibrio >= EquilibrioFinal)
201         {
202           Entrada++;
203           //EN ESTA SECCION SE MODIFICAN LOS PARAMETROS DEL ALGORITMO DEPENDIENDO
204           //DEL NIVEL DE ACEPTACION (Aceptada/Entrada)
205           //El ESTADO PUEDE SER: CALIENTE, TEMPLADO O FRIO
206
207           if(0.60 <= Aceptada/Entrada && Precalentado == 0)
```

```
208             {
209               EquilibrioFinal = Iteraciones_Caliente;
210               alfa = FactorEnfriamiento_Caliente;
211             }
212
213             if(0.40 <= Aceptada/Entrada && Aceptada/Entrada < 0.60 && Precalentado == 0)
214             {
215               EquilibrioFinal = Iteraciones_Templado;
216               alfa = FactorEnfriamiento_Templado;
217             }
218
219             if(Aceptada/Entrada < 0.40 && Precalentado == 0)
220             {
221               EquilibrioFinal = Iteraciones_Frio;
222               alfa = FactorEnfriamiento_Frio;
223             }
224
225             //CUANDO EL NIVEL DE ACEPTACION ES MUY BAJO SE CONCLUYE EL PROCESO DE MEJORA
226             if(Aceptada/Entrada < 0.01)
227                         Temperatura = TemperaturaFinal;
228
229             //CUANDO EL NIVEL DE ACEPTACION INICIA POR ABAJO DE 0.80 SE
230             //LLEVA A CABO UN PERIODO DE PRECALENTADO (SE AUMENTA LA TEMPERATURA),
231             //HASTA OBTENER UNA ACEPTACION DE AL MENOS 0.80
232             if(Aceptada/Entrada < 0.8 && Precalentado == 1 && Temperatura > TemperaturaFinal)
233                         Temperatura = Temperatura * 1.1;
234             else
235             {
236               Temperatura = Temperatura * alfa;
237               Precalentado = 0;
238             }
239
240             Equilibrio=-1;
241             Entrada = Aceptada = 0;
242           }
243
244           //SE REALIZA UN CAMBIO Y DENTRO DE LA FUCION Cambios() SE EVALUA SU COSTO TOTAL
245           Cambios();
246
247           //LA SOLUCION ACTUAL SE GUARDA CUANDO MEJORA A LA MEJOR SOLUCION CONOCIDA
248           u5 = DesviacionPoblacional_Nueva + Compacidad_Nueva;
249           b1 = MenorCostoPoblacional + MenorCompacidad;
250           if(u5 <= b1)
251           {
252             for(i=0; i<UnidadesPorConjunto; i++)
253                       {
254                           Mejores_Distritos[i] = Distritos_Actuales[i];
255                       }
256
257
258             for(i=0; i<Unidades_Cambiadas[6499]; i++)
259             {
260               m = Unidades_Cambiadas[i];
261               Mejores_Distritos[m] = Distrito_Destino;
262             }
263
264             MenorCostoPoblacional = DesviacionPoblacional_Nueva;
265             MenorCompacidad = Compacidad_Nueva;
266           }
267
268           //SE DETERMINA SI LOS CAMBIOS DE UNIDADES GEOGAFICAS SERAN ACEPTADOS
269           //SE ACEPTARAN CON PROBABILIDAD 1 SI EL CAMBIO MEJORA EL COSTO DE LA SOLUCION ACTUAL
270           //EN OTRO CASO SE USARA EL CRITERIO DE METROPOLIS
271
272           u5 = (DesviacionPoblacional_Actual - DesviacionPoblacional_Nueva);
273           u6 = (Compacidad_Actual - Compacidad_Nueva);
274           u5 = exp( (u6 + u5) / Temperatura);
275           if(u5 < 1)
276                       Entrada++; //SE CUENTA EL NUMERO DE SOLUCIONES DE MENOR CALIDAD VISITADAS
277           b1 = SiguienteAleatorioReal0y1(& Semilla);
278           if(u5 > b1)
279           {
280             if(u5 < 1)
281                         Aceptada++;//SE CUENTA EL NUMERO DE VECES QUE SE ACEPTA UNA SOLUCION DE MENOR CALIDAD
282
283             for(i=0; i<Unidades_Cambiadas[6499]; i++)
284             {
285               m = Unidades_Cambiadas[i];
286               Distritos_Actuales[m] = Distrito_Destino;
287             }
288             //SE REALIZAN LOS CAMBIOS SUGERIDOS EN LA FUNCION Cambios()
289             //Y SE ACTUALIZAN LOS COSTOS
290             DesviacionPoblacional_Actual = DesviacionPoblacional_Nueva;
291             Compacidad_Actual = Compacidad_Nueva;
292             PoblacionDistritos_Actuales[Distrito_Destino] = PoblacionDistrito_Destino;
293             MedidaArea[Distrito_Destino] = AreaDistrito_Destino;
294             MedidaPerimetro[Distrito_Destino] = PerimetroDistrito_Destino;
295             CompacidadDistritos_Actuales[Distrito_Destino] = CompacidadDistrito_Destino;
296             DesviacionPoblacionalDistritos_Actuales[Distrito_Destino] = DesviacionPoblacional_Destino;
```

```
297             PoblacionDistritos_Actuales[Distrito_Origen] = PoblacionDistrito_Origen;
298             MedidaArea[Distrito_Origen] = AreaDistrito_Origen;
299             MedidaPerimetro[Distrito_Origen] = PerimetroDistrito_Origen;
300             CompacidadDistritos_Actuales[Distrito_Origen] = CompacidadDistrito_Origen;
301             DesviacionPoblacionalDistritos_Actuales[Distrito_Origen] = DesviacionPoblacional_Origen;
302           }
303           else
304           {
305             DesviacionPoblacional_Nueva = DesviacionPoblacional_Actual;
306             Compacidad_Nueva = Compacidad_Actual;
307           }
308         }
309         //TERMINA EL PROCESO DE MEJORA DE UN CONJUNTO TERRITORIAL
310
311         //SE REALIZA UNA BUSQUEDA LOCAL
312         j = Busqueda_Local();
313         //EN CASO DE MEJORA SE ACTUALIZA LA INFORMACION
314         if(j == 1)
315               {
316                   for(i=0; i<UnidadesPorConjunto; i++)
317                   {
318                       Mejores_Distritos[i] = Distritos_Actuales[i];
319                   }
320
321                   MenorCostoPoblacional = DesviacionPoblacional_Actual;
322                   MenorCompacidad = Compacidad_Actual;
323               }
324
325         //SE GUARDAN LOS MEJORES DISTRITOS CONSTRUIDOS EN LA VARIABLE DistritosFinales
326         //AL TEMINAR CADA CORRIDA EL ARREGLO DistritosFinales TENDRA EL ESCENARIO COMPLETO
327 Final:
328
329         for(i=0; i<UnidadesPorConjunto; i++)
330         {
331           Distritos_Actuales[i] = Mejores_Distritos[i];
332           for(j=0; j<6500; j++)
333           {
334             if(Conversion[j] == i)
335                         DistritosFinales[j] = Mejores_Distritos[i] + DistritosAcumulados;
336           }
337
338         }
339         Costo_Solucion_Inicial();
340         for(i = 0; i < NDistritos; i++)
341         {
342           DesviacionPoblacional_EscenarioFinal[i + DistritosAcumulados] = DesviacionPoblacionalDistritos_Actuales[i];
343           Compacidad_EscenarioFinal[i + DistritosAcumulados] = CompacidadDistritos_Actuales[i];
344         }
345
346         printf(" Costo final %f = %f  +  0.5 * %f\n",DesviacionPoblacional_Actual + Compacidad_Actual, DesviacionPoblacional_Actual, 2 *
                 Compacidad_Actual);
347
348               //SE IMPRIME EL COSTO DE CADA CONJUNTO TERRITORIAL
349                   sprintf(dummy, "Sistema_de_visualizacion\\Resumen_Costos\\RS_Costo_Por_Conjunto.csv");
350                   fp = fopen(dummy, "a");
351                   fprintf(fp,"Conjunto %d,%f,%f,%f\n", ConjuntoActual,DesviacionPoblacional_Actual + Compacidad_Actual,
                             DesviacionPoblacional_Actual, Compacidad_Actual);
352                   fclose(fp);
353
354               //SE CREA O SE ACTUALIZA EL Escenario_Hibrido
355         if(Corrida == 0 && Numero_de_Semillas > 1)
356         {
357           Mejora_Hibrida = 1;
358           for(i=0; i<UnidadesPorConjunto; i++)
359           {
360             for(j=0; j<6500; j++)
361             {
362               if(Conversion[j] == i)
363                           Solucion_Hibrida[j] = Mejores_Distritos[i] + DistritosAcumulados;
364             }
365           }
366           for(i = 0; i < NDistritos; i++)
367           {
368             DesviacionPoblacional_Hibrida[i + DistritosAcumulados] = DesviacionPoblacional_EscenarioFinal[i + DistritosAcumulados];
369             Compacidad_Hibrida[i + DistritosAcumulados] = Compacidad_EscenarioFinal[i + DistritosAcumulados];
370           }
371         }
372         if(Corrida >= 1)
373         {
374           Entrada = Aceptada = 0;
375           for(i = 0; i < NDistritos; i++)
376           {
377             Entrada += DesviacionPoblacional_Hibrida[i + DistritosAcumulados] + Compacidad_Hibrida[i + DistritosAcumulados];
378             Aceptada += DesviacionPoblacional_EscenarioFinal[i + DistritosAcumulados] + Compacidad_EscenarioFinal[i +
                     DistritosAcumulados];
379           }
380           if(Entrada > Aceptada)
381           {
382             Mejora_Hibrida = 1;
```

```
383             for(i=0; i<UnidadesPorConjunto; i++)
384             {
385               for(j=0; j<6500; j++)
386               {
387                 if(Conversion[j] == i)
388                                 Solucion_Hibrida[j] = Mejores_Distritos[i] + DistritosAcumulados;
389               }
390             }
391             for(i = 0; i < NDistritos; i++)
392             {
393               DesviacionPoblacional_Hibrida[i + DistritosAcumulados] = DesviacionPoblacional_EscenarioFinal[i + DistritosAcumulados];
394               Compacidad_Hibrida[i + DistritosAcumulados] = Compacidad_EscenarioFinal[i + DistritosAcumulados];
395             }
396           }
397         }
398
399         DistritosAcumulados += NDistritos;
400         //TERMINA CADA CONJUNTO
401       }
402       //TERMINA EL PROCESO DE MEJORA
403
404       end = clock();
405       seed1 = end - start;
406       z = seed1 / CLOCKS_PER_SEC;
407       seed1 = seed1 / (60 * CLOCKS_PER_SEC);
408       i = (int) seed1;
409       seed1 = z - (60 * i);
410       z = i / 60;
411       printf("\n\nEL TIEMPO DE EJECUCION FUE DE: %d MINUTOS %f SEGUNDOS\n\n",i,seed1);
412       for(i=0; i< 6500; i++)
413           {
414               Mejores_Distritos[i] = DistritosFinales[i];
415           }
416
417
418       //SE OBTIENE EL COSTO TOTAL DE LA SOLUCION CONSTRUIDA
419       CostoTotal = DesviacionPoblacional_Nueva = Compacidad_Nueva = 0;
420       for(i=0; i < ConjuntosTotales; i++)
421       {
422         CostoTotal += DesviacionPoblacional_EscenarioFinal[i] + Compacidad_EscenarioFinal[i];
423         DesviacionPoblacional_Nueva += DesviacionPoblacional_EscenarioFinal[i];
424         Compacidad_Nueva += Compacidad_EscenarioFinal[i];
425       }
426
427       //SE IMPRIME EL COSTO TOTAL DEL ESCENARIO ACTUAL
428           sprintf(dummy, "Sistema_de_visualizacion\\Resumen_Costos\\RS_Costo_Por_Conjunto.csv");
429           fp = fopen(dummy, "a");
430           fprintf(fp,"Costo Total,%f,%f,%f\n",CostoTotal,DesviacionPoblacional_Nueva, Compacidad_Nueva);
431           fclose(fp);
432
433           if(Corrida >= 1)
434           {
435               sprintf(dummy, "Sistema_de_visualizacion\\Resumen_Costos\\RS_Costo_Por_Conjunto.csv");
436               fp = fopen(dummy, "a");
437               fprintf(fp,"Costo Total,%f,%f,%f\n",CostoTotal,DesviacionPoblacional_Nueva, Compacidad_Nueva);
438               fclose(fp);
439           }
440
441       //SE IMPRIME EN ARCHIVO DE csv LA SOLUCION FINAL
442       sprintf(dummy, "Sistema_de_visualizacion\\RS_Escenario_%f_%d.csv",DesviacionPoblacional_Nueva + Compacidad_Nueva , Semillas[
              Corrida]);
443       fp = fopen(dummy, "w");
444       fprintf(fp,"Seccion,DISTRITO\n");
445       for(i = 0; i < 6500; i++)
446           {
447               if(Mejores_Distritos[i] < 6500)
448                   fprintf(fp,"%d,%d\n",i,Mejores_Distritos[i] + 1);
449           }
450       fclose(fp);
451
452       //SE IMPRIME LA SOLUCION HIBRIDA CONSTRUIDA CON LOS ESCENARIOS OBTENIDOS
453           //SI SE OBTUVO ALGUNA MEJORA Y SE EMPLEO EL SEMILLERO
454           if(Numero_de_Semillas > 1 && Mejora_Hibrida == 1)
455           {
456               Mejora_Hibrida = 0;
457               direccion = "Sistema_de_visualizacion\\Escenario_Hibrido_RS.csv";
458               fp = fopen(direccion, "w");
459               fprintf(fp,"Seccion,DISTRITO\n");
460               for(i = 0; i < 6500; i++)
461               {
462                   if(Solucion_Hibrida[i] < 6500)
463                       fprintf(fp,"%d,%d\n",i,Solucion_Hibrida[i] + 1);
464               }
465
466               fclose(fp);
467           }
468   }
469   //TERMINA LA CORRIDA ACTUAL
470
```

```
471    printf("\t Se han completado las corridas solicitadas. \n\n \t Presione la tecla Enter para conluir el proceso.");
472    getchar();
473
474    return(1);
475  }
```

## Anexo : Función Datos(int Conjunto)

```
1
2  void Datos(int Conjunto)
3  //LEE ALGUNOS ARCHIVOS DE TEXTO COMO DatosConglomerados, ColindanciasUnidades, etc. PARA OBTENER INFORMACION SOBRE
4  //LAS UNIDADES GEOGRAFICAS QUE EMPLEARA EN CADA CONJUNTO TERRITORIAL
5  {
6    int i,k,l,m,c,mun,o,j;
7    double p;
8    int U, f, Conglomerado[10500];
9    int ConjuntoTerritorial[6500];
10   int Separar[8][2], SeccionMun[6500], Aux;
11   FILE *fp;
12   char *direccion;
13   UnidadesPorConjunto = 0;
14   MediaEstatal = 0;
15 /*
16   direccion = "Sistema_de_visualizacion\\Insumos\\Separar.txt";
17   U = 0;
18   fp = fopen(direccion, "r");
19   while((c=fgetc(fp))!=EOF)
20   {
21     fscanf(fp,"%d %d",&i,&k);
22     Separar[U][0] = i;
23     Separar[U][1] = k;
24     U++;
25   }
26   fclose(fp);
27 */
28   for(i = 0; i < 10500; i++)
29   {
30     Conglomerado[i] = -1;
31   }
32   m = 0;
33   direccion = "Sistema_de_visualizacion\\Insumos\\DatosConglomerados.txt";
34   fp = fopen(direccion, "r");
35   while((c=fgetc(fp))!=EOF)
36   {
37     fscanf(fp,"%d %d %f %d %d %d",&mun,&i,&p,&j,&o,&k);
38     MediaEstatal += j;
39     ConjuntoTerritorial[i] = k;
40     f = 0;
41     SeccionMun[i] = mun;
42     if(k == Conjunto)
43     {
44         if(Conglomerado[o] != -1)
45         {
46           Conversion[i] = Conglomerado[o];
47           AreaUnidadGeografica[Conglomerado[o]] += p;
48           PoblacionUnidadGeografica[Conglomerado[o]] += j;
49         }
50
51         if(Conglomerado[o] == -1)
52         {
53           Conversion[i] = m;
54           AreaUnidadGeografica[m] = p;
55           PoblacionUnidadGeografica[m] = j;
56           UnidadesPorConjunto++;
57           Conglomerado[o] = m;
58           m++;
59         }
60     }
61   }
62   fclose(fp);
63   MediaEstatal = MediaEstatal / ConjuntosTotales;
64
65   for(i=0; i<6500; i++)
66   {
67     for(j=0; j<6500; j++)
68     {
69       PerimetroFrontera[i][j] = 0;
70     }
71   }
72
73
```

```
74    for(i=0; i<6500; i++)
75    {
76      for(j=0; j<60; j++)
77           {
78                Vecinos[i][j] = 6500;
79           }
80      Vecinos[i][60] = 0;
81    }
82      Separar[k][0] = Separar[k][1] = -1;
83    direccion = "Sistema_de_visualizacion\\Insumos\\ColindanciasUnidades.txt";
84    fp = fopen(direccion, "r");
85    while((c=fgetc(fp))!=EOF)
86    {
87      fscanf(fp,"%d %d %lf",&i,&j,&p);
88      if(ConjuntoTerritorial[i] == Conjunto)
89      {
90        Aux = 0;
91        if(j != 0)
92              {
93                  for(k=0; k<8;k++)
94                  {
95                       if(SeccionMun[i] == Separar[k][0] && SeccionMun[j] == Separar[k][1])
96                       {
97                            m = j;
98                            Aux = 1;
99                            j = 0;
100                           break;
101                      }
102                 }
103             }
104       if(j != 0)
105       {
106         if(ConjuntoTerritorial[j] == Conjunto)
107         {
108           l = Conversion[i];
109           m = Conversion[j];
110           if(l != m)
111           {
112             PerimetroFrontera[l][m] += p;
113
114             f = 0;
115             for(U = 0; U < Vecinos[l][60]; U++)
116             {
117               if(Vecinos[l][U] == m)
118               {
119                 f = 1;
120                 break;
121               }
122             }
123             if(f == 0)
124             {
125               Vecinos[l][Vecinos[l][60]] = m;
126               Vecinos[l][60]++;
127             }
128           }
129         }
130
131         else
132         {
133           l = Conversion[i];
134           PerimetroFrontera[l][l] += p;
135         }
136         if(Aux == 1)
137                     j = m;
138       }
139       if(j == 0)
140       {
141         l = Conversion[i];
142         PerimetroFrontera[l][l] += p;
143       }
144     }
145   }
146   fclose(fp);
147 }
```

## Anexo : Función Solucion_Inicial(int DistritosPorConjunto)

```
void Solucion_Inicial(int DistritosPorConjunto)
//CONSTRUYE UNA SOLUCION INICIAL CON EL NUMERO DE DISTRITOS INDICADOS PARA EL CONJUNTO TERRITORIAL ACTUAL
//TODOS LOS DISTRITOS SON CONEXOS Y TODAS LAS UNIDADES GEOGRAFICAS PERTENCEN EXACTAMENTE A UN DISTRITO
{
  int i, j, k,l,pp,u[45],contador[6500],seed, z;
  int Unidades[6500],Distrito_Auxiliar[6500];

  for(i = 0; i < UnidadesPorConjunto; i++)
  {
    Distritos_Actuales[i] = -1;
    Distrito_Auxiliar[i] = -1;
  }

  //SE GENERAN DistritosPorConjunto SEMILLAS PARA EL CONJUNTO ACTUAL
  for(i = 0; i < UnidadesPorConjunto; i++)
    {
        Unidades[i] = i;
        contador[i]=0;
    }

  j = UnidadesPorConjunto;

  for(k = 0; k < DistritosPorConjunto; k++)
  {
    i = SiguienteAleatorioEnteroModN(& Semilla, j);
    u[k] = Unidades[i];
    contador[u[k]] = 1;

    //SE INICIALIZAN LOS DISTRITOS CON LAS UNIDADES ELEGIDAS
    Distritos_Actuales[Unidades[i]] = k;
    j--;
    for(l = i; l < j; l++)
    {
      Unidades[l] = Unidades[l + 1];
    }
  }

  //SE EMPIEZA LA CONSTRUCCION DE LA SOLUCION INICIAL
  j = 0;

  while(j != UnidadesPorConjunto)
  {
        k = SiguienteAleatorioEnteroModN(& Semilla,DistritosPorConjunto);

    //SE ENCUENTRAN LAS COLINDANCIAS DEL DISTRITO k
    for(i=0; i<UnidadesPorConjunto; i++)
    {
      if(Distritos_Actuales[i] == k)
      {
        for(j=0; j<Vecinos[i][60]; j++)
        {
          if(contador[Vecinos[i][j]] == 0)
                        Distrito_Auxiliar[Vecinos[i][j]] = k;
        }
      }
    }

    //SE CUENTAN A TODOS LOS VECINOS QUE SE PUEDEN AGREGAR AL DISTRITO  k
    pp = 0;
    for(j=0; j<UnidadesPorConjunto; j++)
    {
      if(Distrito_Auxiliar[j] == k)
                pp++;
    }
    if(pp == 1)
              seed = 0;
    if(pp > 1)
              seed = SiguienteAleatorioEnteroModN(& Semilla,pp);
    if(pp > 0)
    {
      z = 0;
      //SE SELECCIONA A UN VECINO DEL DISTRITO k Y LO AGREGA
      for(j=0; j<UnidadesPorConjunto; j++)
      {
        if(Distrito_Auxiliar[j] == k)
        {
          if( z == seed)
          {
            contador[j] = 1;
            Distritos_Actuales[j] = k;
            j = UnidadesPorConjunto;
            break;
          }
          z++;
```

```
85          }
86        }
87          }
88      j = 0;
89      for(i=0; i<UnidadesPorConjunto; i++)
90          {
91              j += contador[i];
92              Distrito_Auxiliar[i] = -1;
93          }
94    }
95    for(i=0; i<UnidadesPorConjunto; i++)
96    {
97      Mejores_Distritos[i] = Distritos_Actuales[i];
98      Distrito_Auxiliar[i] = -1;
99    }
100 }
```

## Anexo : Función Costo_Solucion_Inicial()

```
1  void Costo_Solucion_Inicial()
2  //CALCULA EL COSTO DE LA SOLUCION INICIAL, EN ESTE CASO SE DEBEN CALCULAR LOS COSTOS DE TODOS LOS DISTRITOS
3  {
4    int i, j, k, o, p;
5    for(i=0; i<NDistritos; i++)
6    {
7      MedidaPerimetro[i] = 0;
8      PoblacionDistritos_Actuales[i] = 0;
9      MedidaArea[i] = 0;
10     for(j=0; j<UnidadesPorConjunto; j++)
11     {
12       if(Distritos_Actuales[j] == i)
13       {
14         PoblacionDistritos_Actuales[i] += PoblacionUnidadGeografica[j];
15         MedidaArea[i] += AreaUnidadGeografica[j];
16       }
17     }
18   }
19   for(k = 0; k < UnidadesPorConjunto; k++)
20   {
21     i = Distritos_Actuales[k];
22     MedidaPerimetro[i] += PerimetroFrontera[k][k];
23     o = Vecinos[k][60];
24     for(j = 0; j < o; j++)
25     {
26       p = Vecinos[k][j];
27       if(Distritos_Actuales[p] != i )
28               MedidaPerimetro[i] += PerimetroFrontera[k][p];
29     }
30   }
31   DesviacionPoblacional_Actual = Compacidad_Actual = 0;
32   for(i=0; i<NDistritos; i++)
33   {
34     DesviacionPoblacionalDistritos_Actuales[i] = Desviacion_Poblacional(PoblacionDistritos_Actuales[i]);
35     CompacidadDistritos_Actuales[i] = Compacidad(MedidaArea[i],MedidaPerimetro[i]);
36     DesviacionPoblacional_Actual += DesviacionPoblacionalDistritos_Actuales[i];
37     Compacidad_Actual += CompacidadDistritos_Actuales[i];
38   }
39 }
```

## Anexo : Función Cambios()

```
int Cambios(void)
//SE GENERA UNA SOLUCION VECINA DE Distritos_Actuales
{
  int b, i, j, n, k, n3;
  int DistritoOrigen, DistritoDestino, Unidad_Elegida;
  int destinosE[30], Candidatos[6500];

  j = 1;
  //SE ELIGE UN DISTRITO AL AZAR
  while(j <= 1)
  {
    DistritoOrigen = SiguienteAleatorioEnteroModN(& Semilla, NDistritos);

    //SE EVALUA SI EL DistritoOrigen TIENE MAS DE UNA UNIDAD GEOGRAFICA
    j = Cardinalidad_Distrito(DistritoOrigen);
  }

  //SE HACE UNA LISTA CON LAS UNIDADES GEOGRAFICAS QUE SE PUEDEN CAMBIAR DEL DistritoOrigen
  //EN ESTE CASO SE CONSIDERAN A TODAS LAS UNIDADES DEL DistritoOrigen QUE COLINDAN CON OTRO DISTRITO
  for(i = 0; i < UnidadesPorConjunto; i++)
  {
    Candidatos[i] = 6500;
    Unidades_Cambiadas[i] = 6500;
  }
  n = 0;
  for(i = 0; i < UnidadesPorConjunto; i++)
  {
    if(Distritos_Actuales[i] == DistritoOrigen)
    {
      k = j = 0;
      while(k < 6500)
      {
        k = Vecinos[i][j];
        if(Vecinos[i][j] < 6500)
        {
          Unidad_Elegida = Vecinos[i][j];
          if(Distritos_Actuales[Unidad_Elegida] != DistritoOrigen)
          {
            Candidatos[n] = i;
            n++;
            k = 6500;
          }
        }
        j++;
      }
    }
  }

  //SE SELECCIONA UNA UNIDAD GEOGRAFICA PARA SER ENVIADA A OTRO DISTRITO
    //SE ELIGE CON UNA PROBABILIDAD 1/n
  b = SiguienteAleatorioEnteroModN(& Semilla, n);
  Unidad_Elegida = Candidatos[b];

  //SE DETERMINAN LOS DISTRITOS VECINOS DE Unidad_Elegida
  k = 0;
  for(i=0; i<Vecinos[Unidad_Elegida][60]; i++)
  {
    if(Distritos_Actuales[Vecinos[Unidad_Elegida][i]] != DistritoOrigen)
    {
      if(k > 0)
      {
        j = 0;
        for(n3 = 0; n3 <k; n3++)
        {
          if(destinosE[n3] != Distritos_Actuales[Vecinos[Unidad_Elegida][i]])
                    j++;
        }
        if(j == k)
        {
          destinosE[k] = Distritos_Actuales[Vecinos[Unidad_Elegida][i]];
          k++;
        }
      }
      if(k == 0)
      {
        destinosE[k] = Distritos_Actuales[Vecinos[Unidad_Elegida][i]];
        k++;
      }
    }
  }

  //SE ELIGE UN DISTRITO DESTINO PARA Unidad_Elegida
  //SE ELIGE CON UNA PROBABILIDAD 1/n
  if(k == 1)
```

```
        n3 = 0;
    else
        n3 = SiguienteAleatorioEnteroModN(& Semilla, k);

  DistritoDestino = destinosE[n3];
  Distritos_Actuales[Unidad_Elegida] = DistritoDestino;
  Distrito_Origen = DistritoOrigen;
  Distrito_Destino = DistritoDestino;
  Unidades_Cambiadas[0] = Unidad_Elegida;
  Unidades_Cambiadas[6499] = 1;

  //SE REVISA SI SE PROVOCO ALGUNA DISCONEXION
  j = Revisa_Conexidad(DistritoOrigen,DistritoDestino);

  //SE EVALUA EL COSTO DEL NUEVO ESCENARIO
  j = Costo_Nueva_Solucion(DistritoOrigen, DistritoDestino);

  //SE REGRESAN TODAS LAS UNIDADES CAMBIADAS AL DISTRITO Origen
  for(i=0; i<Unidades_Cambiadas[6499]; i++)
  {
    j = Unidades_Cambiadas[i];
    Distritos_Actuales[j] = DistritoOrigen;
  }

  return(1);
}
```

## Anexo : Función Busqueda_Local()

```
int Busqueda_Local()
{
    int i, j, m, k, n3, Mejora = 0;
    int Cardinalidad, Unidad_Elegida;
    int DistritoOrigen, DistritoDestino;
  int destinosE[30], Destino;
  double Costo_DestinoE;

    for(i=0; i<UnidadesPorConjunto; i++)
        Distritos_Actuales[i] = Mejores_Distritos[i];

    Costo_Solucion_Inicial();
    Costo_DestinoE = DesviacionPoblacional_Actual + Compacidad_Actual;
    Destino = -1;
    m = 0;

  while(m == 0)
    {
        m = 1;
        for(Unidad_Elegida = 0; Unidad_Elegida < UnidadesPorConjunto; Unidad_Elegida++)
        {
            //SE REVISA QUE EL DISTRITO CONTENGA MAS DE 2 UNIDADES GEOGRAFICAS
            Cardinalidad = Cardinalidad_Distrito(Distritos_Actuales[Unidad_Elegida]);
            if(Cardinalidad >= 2)
            {
                DistritoOrigen = Distritos_Actuales[Unidad_Elegida];
                //SE DETERMINAN LOS DISTRITOS VECINOS DE Unidad_Elegida
                k = 0;
                for(i=0; i<Vecinos[Unidad_Elegida][60]; i++)
                {
                    if(Distritos_Actuales[Vecinos[Unidad_Elegida][i]] != DistritoOrigen)
                    {
                        if(k > 0)
                        {
                            j = 0;
                            for(n3 = 0; n3 <k; n3++)
                            {
                                if(destinosE[n3] != Distritos_Actuales[Vecinos[Unidad_Elegida][i]])
                                    j++;
                            }
                            if(j == k)
                            {
                                destinosE[k] = Distritos_Actuales[Vecinos[Unidad_Elegida][i]];
                                k++;
                            }
                        }
                        if(k == 0)
                        {
                            destinosE[k] = Distritos_Actuales[Vecinos[Unidad_Elegida][i]];
                            k++;
                        }
                    }
```

```
53                  }
54                  //SE REVISAN LOS POSIBLES CAMBIOS DE Unidad_Elegida A DISTRITOS VECINOS
55                  for(i = 0; i < k; i++)
56                   {
57                      //SE ELIGE UN DISTRITO DESTINO PARA Unidad_Elegida
58                      DistritoDestino = destinosE[i];
59                      Distritos_Actuales[Unidad_Elegida] = DistritoDestino;
60                      Distrito_Origen = DistritoOrigen;
61                      Distrito_Destino = DistritoDestino;
62                      Unidades_Cambiadas[0] = Unidad_Elegida;
63                      Unidades_Cambiadas[6499] = 1;
64
65                      //SE REVISA SI SE PROVOCO ALGUNA DISCONEXION
66                      j = Revisa_Conexidad(DistritoOrigen,DistritoDestino);
67
68                      //SE EVALUA EL COSTO DEL NUEVO ESCENARIO
69                      j = Costo_Nueva_Solucion(DistritoOrigen, DistritoDestino);
70                      //SE GUARDA LA INFORMACION EN CASO DE MEJORA
71                      if(Costo_DestinoE > DesviacionPoblacional_Nueva + Compacidad_Nueva)
72                      {
73                          Costo_DestinoE = DesviacionPoblacional_Nueva + Compacidad_Nueva;
74                          Destino = i;
75                      }
76
77                      //SE REGRESAN TODAS LAS UNIDADES CAMBIADAS AL DISTRITO Origen
78                      for(m=0; m<Unidades_Cambiadas[6499]; m++)
79                      {
80                          j = Unidades_Cambiadas[m];
81                          Distritos_Actuales[j] = DistritoOrigen;
82                      }
83
84                      //SE REGRESAN LOS COSTOS A SU NIVEL ORIGINAL
85              DesviacionPoblacional_Nueva = DesviacionPoblacional_Actual;
86              Compacidad_Nueva = Compacidad_Actual;
87                  }
88
89                  if(Destino != -1)
90                  {
91                      i = Destino;
92                      Destino = -1;
93                      DistritoDestino = destinosE[i];
94                      Distritos_Actuales[Unidad_Elegida] = DistritoDestino;
95                      Distrito_Origen = DistritoOrigen;
96                      Distrito_Destino = DistritoDestino;
97                      Unidades_Cambiadas[0] = Unidad_Elegida;
98                      Unidades_Cambiadas[6499] = 1;
99
100                     //SE REVISA SI SE PROVOCO ALGUNA DISCONEXION
101                     j = Revisa_Conexidad(DistritoOrigen,DistritoDestino);
102
103                     //SE EVALUA EL COSTO DEL NUEVO ESCENARIO
104                     j = Costo_Nueva_Solucion(DistritoOrigen, DistritoDestino);
105
106                     //SE ACTUALIZA EL COSTO Y LA SOLUCION
107                     for(i=0; i<Unidades_Cambiadas[6499]; i++)
108             {
109               m = Unidades_Cambiadas[i];
110                Distritos_Actuales[m] = Distrito_Destino;
111             }
112             //SE REALIZAN LOS CAMBIOS SUGERIDOS
113             //Y SE ACTUALIZAN LOS COSTOS
114             DesviacionPoblacional_Actual = DesviacionPoblacional_Nueva;
115             Compacidad_Actual = Compacidad_Nueva;
116             PoblacionDistritos_Actuales[Distrito_Destino] = PoblacionDistrito_Destino;
117             MedidaArea[Distrito_Destino] = AreaDistrito_Destino;
118             MedidaPerimetro[Distrito_Destino] = PerimetroDistrito_Destino;
119             CompacidadDistritos_Actuales[Distrito_Destino] = CompacidadDistrito_Destino;
120             DesviacionPoblacionalDistritos_Actuales[Distrito_Destino] = DesviacionPoblacional_Destino;
121             PoblacionDistritos_Actuales[Distrito_Origen] = PoblacionDistrito_Origen;
122             MedidaArea[Distrito_Origen] = AreaDistrito_Origen;
123             MedidaPerimetro[Distrito_Origen] = PerimetroDistrito_Origen;
124             CompacidadDistritos_Actuales[Distrito_Origen] = CompacidadDistrito_Origen;
125             DesviacionPoblacionalDistritos_Actuales[Distrito_Origen] = DesviacionPoblacional_Origen;
126
127                     m = 0;
128                     Mejora = 1;
129                 }
130             }
131         }
132     }
133     return(Mejora);
134 }
```

## Anexo : Función Cardinalidad_Distrito(int Distrito)

```
int Cardinalidad_Distrito(int Distrito)
//CALCULA EL NUMERO DE UNIDADES GEOGRAFICAS EN Distrito
{
  int m,suma;
  suma = 0;
  for(m = 0; m< UnidadesPorConjunto; m++)
  {
    if(Distritos_Actuales[m] == Distrito)
          suma++;
  }
  return(suma);
}
```

## Anexo : Función Revisa_Conexidad(int Origen, int Destino)

```
int Revisa_Conexidad(int Origen, int Destino)
//CON ESTA FUNCION SE REVISA SI EL DISTRITO Origen PERDIO LA CONEXIDAD
{
  int j,i,m,suma;
  int Representante[6500], Componente[6500],N = 0,n;
  int Lista1[6500], Lista2[6500];
  int Contactado[6500];

  suma = 0;

  //SE REVISA EL NUMERO DE UNIDADES GEOGRAFICAS EN EL DISTRITO
  for(m=0; m<UnidadesPorConjunto; m++)
  {
    Contactado[m] = 0;
    Representante[m] = -1;
    Componente[m] = 0;
    if(Distritos_Actuales[m] == Origen)
    {
      Lista1[suma] = m;
      suma++;
    }
  }

  //SI EL NUMERO DE UNIDADES GEOGRAFICAS ES MENOR O IGUAL A 1 NO SE HA PERDIDO LA CONEXIDAD
  if( suma == 0)
  {
    printf("\n ERROR Distrito vacia \n");
    getchar();
    return(0);
  }

  if( suma == 1)
        return(0);

  //SE REVISA EL NUMERO DE COMPONENTES CONEXAS EN EL DISTRITO
  //SI EL NUMERO DE COMPONENTES CONEXAS ES IGUAL A 1 NO SE HA PERDIDO LA CONEXIDAD
  //SI EL NUMERO DE COMPONENTES CONEXAS ES MAYOR O IGUAL A 2 SE DEBE REPARAR LA CONEXIDAD
  for(i=0; i< suma; i++)
  {
    n = 0;
    if(Contactado[Lista1[i]] == 0)
    {
      Representante[N] = Lista1[i];
      Lista2[n] = Lista1[i];
      Contactado[Lista2[n]] = 1;
      n++;
      for(j=0; j < n; j++)
      {
        for(m=0; m < Vecinos[Lista2[j]][60]; m++)
        {
          if(Distritos_Actuales[Vecinos[Lista2[j]][m]] == Origen && Contactado[Vecinos[Lista2[j]][m]] == 0)
          {
            Lista2[n] = Vecinos[Lista2[j]][m];
            Contactado[Vecinos[Lista2[j]][m]] = 1;
            n++;
          }
        }
      }
      Componente[N] = n;
      N++;
    }
  }
```

```
63
64    if( N == 1)
65          return(0);
66
67    n = 0;
68    j = Componente[0];
69
70    //SE DETERMINA CUAL ES LA COMPONENTE CON MAYOR NUMERO DE UNIDADES GEOGRAFICAS
71    //PARA DEJARLA COMO EL DISTRITO Origen
72    for(i = 1; i < N; i++)
73    {
74      if(Componente[i] > j)
75      {
76        j = Componente[i];
77        n = i;
78      }
79    }
80
81    //EL RESTO DE LAS COMPONETES SON ENVIADAS AL DISTRITO Destino
82    for(i = 0; i < N; i++)
83    {
84      if(i != n)
85              Repara_Conexidad(Origen,Representante[i], Destino);
86    }
87    return(0);
88  }
```

## Anexo : Función Repara_Conexidad(int Origen, int k, int Destino)

```
1   int Repara_Conexidad(int Origen, int k, int Destino)
2   //ESTA FUNCION ES LLAMADA CUANDO SE HA COMPROBADO FALTA DE CONEXIDAD EN EL DISTRITO Origen
3   //LA UNIDAD GEOGRAFICA k ES UN REPRESENTANTE DE UNA COMPONENTE CONEXA DEL DISTRITO Origen
4   {
5     int j,i,m,n,o;
6     int Lista[6500];
7     int ComponenteConexa[6500];
8
9     for(m=0; m<UnidadesPorConjunto; m++)
10      ComponenteConexa[m] = 0;
11
12    Lista[0] = k;
13    ComponenteConexa[Lista[0]] = 1;
14    n = 1;
15    //SE ENCUENTRA A TODAS LAS UNIDADES GEOGRAFICAS EN LA MISMA COMPONENTE CONEXA QUE k
16    for(j=0; j < n; j++)
17    {
18      o = Vecinos[Lista[j]][60];
19      for(m=0; m< o; m++)
20      {
21        if(Distritos_Actuales[Vecinos[Lista[j]][m]] == Origen && ComponenteConexa[Vecinos[Lista[j]][m]] == 0)
22        {
23          Lista[n] = Vecinos[Lista[j]][m];
24          ComponenteConexa[Vecinos[Lista[j]][m]] = 1;
25          n++;
26        }
27      }
28    }
29    //TODAS LAS UNIDADES GEOGRAFICAS EN LA COMPONENTE CONEXA DE k SON ENVIADAS AL DISTRITO Destino
30    for(i = 0; i < UnidadesPorConjunto; i++)
31    {
32      if(ComponenteConexa[i] == 1)
33      {
34        Distritos_Actuales[i] = Destino;
35        Unidades_Cambiadas[Unidades_Cambiadas[6499]] = i;
36        Unidades_Cambiadas[6499]++;
37      }
38    }
39    return(0);
40  }
```

## Anexo : Función Costo_Nueva_Solucion(int Origen, int Destino)

```
1   int Costo_Nueva_Solucion(int Origen, int Destino)
2   //CALCULA EL COSTO DE LA NUEVA SOLUCION, PARA ESTO BASTA CON CALCULAR EL COSTO
```

```
//DEL DISTRITO Origen Y DEL DISTRITO Destino
{
  int j, k;
  double MedidaPerimetroE[30], MedidaAreaE[30] , DesviacionPoblacionalE;
  int PoblacionDistritoE[30];
  double DesviacionPoblacionalDistritoE[30], MedidaCompacidadE[30], CompacidadTotalE;
  int o, p;
  Distrito_Origen = Origen;
  Distrito_Destino = Destino;
  DesviacionPoblacionalE = DesviacionPoblacional_Nueva - DesviacionPoblacionalDistritos_Actuales[Origen] -
        DesviacionPoblacionalDistritos_Actuales[Destino];
  CompacidadTotalE = Compacidad_Nueva - CompacidadDistritos_Actuales[Origen] - CompacidadDistritos_Actuales[Destino];

  PoblacionDistritoE[Origen] = PoblacionDistritos_Actuales[Origen];
  PoblacionDistritoE[Destino] = PoblacionDistritos_Actuales[Destino];
  MedidaAreaE[Origen] = MedidaArea[Origen];
  MedidaAreaE[Destino] = MedidaArea[Destino];

  for(k = 0; k < Unidades_Cambiadas[6499]; k++)
  {
    PoblacionDistritoE[Origen] -= PoblacionUnidadGeografica[Unidades_Cambiadas[k]];
    PoblacionDistritoE[Destino] += PoblacionUnidadGeografica[Unidades_Cambiadas[k]];
    MedidaAreaE[Origen] -= AreaUnidadGeografica[Unidades_Cambiadas[k]];
    MedidaAreaE[Destino] += AreaUnidadGeografica[Unidades_Cambiadas[k]];
  }

  MedidaPerimetroE[Origen] = MedidaPerimetroE[Destino] = 0;
  for(k = 0; k < UnidadesPorConjunto; k++)
  {
    if(Distritos_Actuales[k] == Origen)
    {
      MedidaPerimetroE[Origen] += PerimetroFrontera[k][k];
      o = Vecinos[k][60];
      for(j = 0; j < o; j++)
      {
        p = Vecinos[k][j];
        if(Distritos_Actuales[p] != Origen )
                MedidaPerimetroE[Origen] += PerimetroFrontera[k][p];
      }

    }
    if(Distritos_Actuales[k] == Destino)
    {
      MedidaPerimetroE[Destino] += PerimetroFrontera[k][k];
      o = Vecinos[k][60];
      for(j = 0; j < o; j++)
      {
        p = Vecinos[k][j];
        if(Distritos_Actuales[p] != Destino )
                MedidaPerimetroE[Destino] += PerimetroFrontera[k][p];
      }
    }

  }

  DesviacionPoblacionalDistritoE[Origen] = Desviacion_Poblacional(PoblacionDistritoE[Origen]);
  DesviacionPoblacionalDistritoE[Destino] = Desviacion_Poblacional(PoblacionDistritoE[Destino]);
  MedidaCompacidadE[Origen] = Compacidad(MedidaAreaE[Origen],MedidaPerimetroE[Origen]);
  MedidaCompacidadE[Destino] = Compacidad(MedidaAreaE[Destino],MedidaPerimetroE[Destino]);
  DesviacionPoblacional_Nueva = DesviacionPoblacionalE + DesviacionPoblacionalDistritoE[Origen] + DesviacionPoblacionalDistritoE[
        Destino];
  Compacidad_Nueva = CompacidadTotalE + MedidaCompacidadE[Origen] + MedidaCompacidadE[Destino];
  DesviacionPoblacional_Destino = DesviacionPoblacionalDistritoE[Destino];
  CompacidadDistrito_Destino = MedidaCompacidadE[Destino];
  PoblacionDistrito_Destino = PoblacionDistritoE[Destino];
  AreaDistrito_Destino = MedidaAreaE[Destino];
  PerimetroDistrito_Destino = MedidaPerimetroE[Destino];
  DesviacionPoblacional_Origen = DesviacionPoblacionalDistritoE[Origen];
  CompacidadDistrito_Origen = MedidaCompacidadE[Origen];
  PoblacionDistrito_Origen = PoblacionDistritoE[Origen];
  AreaDistrito_Origen = MedidaAreaE[Origen];
  PerimetroDistrito_Origen = MedidaPerimetroE[Origen];

  return(1);
}
```

## Anexo : Función Desviacion_Poblacional(int Poblacion)

```
double Desviacion_Poblacional(int Poblacion)
//CALCULA EL EQUILIBRIO POBLACIONAL
{
```

```
4    double Costo;
5    Costo = 1.00 -(Poblacion / MediaEstatal);
6    Costo = Costo / 0.15;
7    Costo = pow(Costo, 2);
8    if(Costo > 1)
9          Costo += 10 * (Costo - 1);
10   return(Costo);
11 }
```

## Anexo : Función Compacidad(double Area, double Perimetro)

```
double Compacidad(double Area,double Perimetro)
//CALCULA EL COSTO DE LA COMPACIDAD
{
  double Costo;
  Costo = ((Perimetro / sqrt(Area)) * 0.25 - 1.0) * 0.5;
  return(Costo);
}
```

## Anexo : Función SiguienteAleatorioReal0y1(long *semilla)

```
double SiguienteAleatorioReal0y1(long * semilla)
//DEVUELVE UN ALEATORIO ENTRE 0 Y 1
{
  long double zi, mhi31 = 2147483648u, ahi31 = 314159269u, chi31 = 453806245u;
  long int dhi31;
  zi = *semilla;
  zi = (ahi31 * zi) + chi31;
  if(zi > mhi31)
  {
    dhi31 = (long int) (zi / mhi31);
    zi = zi - (dhi31 * mhi31);
  }
  *semilla = (int) zi;
  zi = zi / mhi31;
  return (zi);
}
```

## Anexo : Función SiguienteAleatorioEnteroModN(long *semilla, int n)

```
int SiguienteAleatorioEnteroModN(long * semilla, int n)
// DEVUELVE UN ENTERO ENTRE 0 y n-1
{
  double a;
  int v;
  long double zi, mhi31 = 2147483648u, ahi31 = 314159269u, chi31 = 453806245u;
  long int dhi31;
  zi = *semilla;
  zi = (ahi31 * zi) + chi31;
  if(zi > mhi31)
  {
    dhi31 = (long int) (zi / mhi31);
    zi = zi - (dhi31 * mhi31);
  }
  *semilla = (long int)zi;
  zi = zi / mhi31;
  a = zi;
  v = (int)(a * n);
  if (v == n)
    return(v-1);
  return(v);
}
```

# ANEXO B
# CÓDIGO DEL ALGORITMO BASADO EN ABC

### Anexo : Función main()

```
int main()
{
  double z,seed1;
  double b1,u5,u6;
  int i,j, m6,k;
  int Semillas[1000], Numero_de_Semillas;
  long c;
  double MejorDesviacionPoblacional[45],MejorCompacidad[45];
  double Menor_DesviacionPoblacional, Menor_Compacidad;
  int Generacion, GeneracionFinal;
  double Calidad_FuenteAlimento[500], Calidad_Total;
  int Corrida;
  int MejoresDistritos[6500];
  int GeneracionesSinMejora[500];
  int Solucion_Hibrida[6500], Mejora_Hibrida;
  double DesviacionPoblacional_Hibrida[45], Compacidad_Hibrida[45];
  char dummy[1000];
  FILE *fp;

  //SE LEEN LOS PARAMETROS SOLICITADOS POR EL USUARIO
  char *direccion = "Colonia_De_Abejas_Artificiales\\Parametros_ABC.txt";
  fp = fopen(direccion, "r");
  while((c=fgetc(fp))!=EOF)
  {
    fscanf(fp,"%d %d %ld",&Fuentes_de_Alimento, &GeneracionFinal, &Semilla);
  }
  fclose(fp);

  //SI EL USUARIO SELECCINO EL SEMILLERO SE LEEN LAS SEMILLAS QUE SE UTILIZARAN
  if(Semilla == -1)
```

```
{
  Numero_de_Semillas = 0;
  direccion = "Sistema_de_visualizacion\\Insumos\\Semillero.txt";
  fp = fopen(direccion, "r");
  while((c=fgetc(fp))!=EOF && Numero_de_Semillas < 1000)
  {
    fscanf(fp,"%d",&i);
    Semillas[Numero_de_Semillas] = i;
    Numero_de_Semillas++;
  }
  fclose(fp);
}
//EN CASO CONTRARIO SOLO SE REALIZARA UNA CORRIDA
else
{
  Numero_de_Semillas = 1;
  Semillas[0] = Semilla;
}

if(Numero_de_Semillas == 1)
{
  printf("\n\n\n\t    Sistema para generar Zonas Electorales 2016\n\n\n");
  printf("    Se realiza una corrida empleando los siguientes parametros:\n\n");
  printf("    Numero de fuentes de alimento = %d\n\n",Fuentes_de_Alimento);
  printf("    Numero de generaciones = %d\n\n",GeneracionFinal);
  printf("    Semilla = %d\n\n",Semilla);
}

else
{
  printf("\n\n\n\t    Sistema para generar Zonas Electorales 2016\n\n\n");
  printf("    Se realizan %d corridas, empleando los siguientes parametros:\n\n", Numero_de_Semillas);
  printf("    Numero de fuentes de alimento = %d\n\n",Fuentes_de_Alimento);
  printf("    Numero de generaciones = %d\n\n",GeneracionFinal);
  printf("    Semilla = Se usan los valores incluidos en el semillero\n\n");
}


// SE LEE EL NUMERO DE DISTRITOS ASIGNADOS A CADA CONJUNTO
ConjuntosTotales = 0;
NConjuntos = 0;
direccion = "Sistema_de_visualizacion\\Insumos\\ConjuntosDistritos.txt";
fp = fopen(direccion, "r");
while((c=fgetc(fp))!=EOF)
{
  fscanf(fp,"%d %d",&k,&i);
  DistritosPorConjunto[i] = k;
  NConjuntos++;
  ConjuntosTotales += k;
}
fclose(fp);


for(i=0; i<6500; i++)
Solucion_Hibrida[i] = 6500;

//SE REALIZAN TANTAS CORRIDAS COMO NUMERO DE SEMILLAS SE ENCUENTREN EN EL SEMILLERO
//O SOLO UNA SI EL USUARIO DA LA SEMILLA
for(Corrida = 0 ; Corrida < Numero_de_Semillas; Corrida++)
{
  //SE CREA UN ARCHIVO PARA GUARDAR LOS COSTOS DE CADA CONJUNTO TERRITORIAL
    if(Corrida == 0)
  {
    sprintf(dummy, "Sistema_de_visualizacion\\Resumen_Costos\\ABC_Costo_Por_Conjunto.csv");
          fp = fopen(dummy, "w");
          fprintf(fp,"Semilla %d,Costo Total, Equilibrio Poblacional,Compacidad\n", Semillas[Corrida]);
          fclose(fp);
      }
      if(Corrida > 0)
  {
    sprintf(dummy, "Sistema_de_visualizacion\\Resumen_Costos\\ABC_Costo_Por_Conjunto.csv");
          fp = fopen(dummy, "a");
          fprintf(fp,"Semilla %d,Costo Total, Equilibrio Poblacional,Compacidad\n", Semillas[Corrida]);
          fclose(fp);
      }

      Mejora_Hibrida = 0;

      if(Numero_de_Semillas > 1)
          printf("\n\n\t Inicia la corrida = %d con la Semilla = %d\n\n",Corrida+1,Semillas[Corrida]);

  Semilla = Semillas[Corrida];

  for(i=0; i< 6500; i++)
  DistritosFinales[i] = 6500;

  //SE INICIA EL TIEMPO DE EJECUCION
  clock_t start, end;
  start = clock();
```

```
120
121     DistritosAcumulados = 0;
122
123     //SE INICIA EL PROCESO DE CONSTRUCCION DE DISTRITOS PARA CADA CONJUNTO
124     for(ConjuntoActual = 1; ConjuntoActual <= NConjuntos; ConjuntoActual++)
125     {
126       printf("\nConjunto territorial %d. ",ConjuntoActual);
127       for(i = 0; i < 6500; i++)
128               Conversion[i] = 6500;
129
130       //LA FUNCION Datos() LEE LA INFORMACION NECESARIA PARA CONSTRUIR LOS DISTRITOS
131       //POR EJEMPLO, COLINDANCIAS, AREA, POBLACION, ETC.
132       Datos(ConjuntoActual);
133       NDistritos = DistritosPorConjunto[ConjuntoActual];
134
135
136       //SE CONSTRUYEN NUEVAS FUENTES DE ALIMENTO Y SE EVALUA SU COSTO (PARA EL CONJUNTO EN CURSO)
137
138       if(DistritosPorConjunto[ConjuntoActual] == 1)
139       {
140         FuenteAlimento_Nueva(NDistritos);
141         for(i = 0; i < UnidadesPorConjunto; i++)
142                 MejoresDistritos[i] = Distrito[i];
143         goto Final;
144       }
145
146       for(j = 0; j < Fuentes_de_Alimento; j++)
147       {
148         FuenteAlimento_Nueva(NDistritos);
149         for(i = 0; i < UnidadesPorConjunto; i++)
150                 FuenteAlimento[j][i] = Distrito[i];
151         Costo_FuenteNueva(j);
152       }
153
154       for(i = 0; i < UnidadesPorConjunto; i++)
155               MejoresDistritos[i] = FuenteAlimento[0][i];
156       Menor_DesviacionPoblacional = DesviacionPoblacional_FuenteAlimento[0];
157       Menor_Compacidad = Compacidad_FuenteAlimento[0];
158
159       for(i = 0; i < Fuentes_de_Alimento; i++)
160               GeneracionesSinMejora[i] = 0;
161
162       //INICIA PROCESO DE MEJORA
163       Generacion = 0;
164       while(Generacion < GeneracionFinal)
165       {
166         //PARA CADA FUENTE DE ALIMENTO SE LLAMA UNA VEZ A LA ABEJA EMPLEADA
167         for(i = 0; i < Fuentes_de_Alimento; i++)
168         {
169
170           //SI LA NUEVA FUENTE DE ALIMENTO ES MEJOR SE ACEPTA Y SE REGRESA UN VALOR DE 0
171           j = AbejaEmpleada(i);
172
173           //SI HUBO MEJORA SE ACTUALIZA LA CALIDAD DE LA FUENTE DE ALIMENTO
174           if(j == 0)
175             GeneracionesSinMejora[i] = 0;
176
177           else
178                   GeneracionesSinMejora[i]++;
179
180           b1 = Menor_DesviacionPoblacional + Menor_Compacidad;
181           u5 = DesviacionPoblacional_FuenteAlimento[i] + Compacidad_FuenteAlimento[i];
182
183           //SI LA NUEVA FUENTE DE ALIMENTO ES LA MEJOR CONOCIDA SE GUARDA EN MEMORIA
184           if(u5 < b1)
185           {
186             Menor_DesviacionPoblacional = DesviacionPoblacional_FuenteAlimento[i];
187             Menor_Compacidad = Compacidad_FuenteAlimento[i];
188             for(k = 0; k < UnidadesPorConjunto; k++)
189             {
190               MejoresDistritos[k] = FuenteAlimento[i][k];
191             }
192           }
193
194           //SI LA FUENTE DE ALIMENTO NO HA SIDO MEJORADA EN 100 GENERARIONES CONSECUTIVAS SE REEMPLAZA
195           if(GeneracionesSinMejora[i] >= 100)
196           {
197             GeneracionesSinMejora[i] = 0;
198             FuenteAlimento_Nueva(NDistritos);
199             for(j = 0; j < UnidadesPorConjunto; j++)
200                     FuenteAlimento[i][j] = Distrito[j];
201             Costo_FuenteNueva(i);
202           }
203         }
204
205         //SE CALCULA LA CALIDAD DE CADA FUENTE DE ALIMENTO
206         //LA SUMA DE SUS CALIDADES ES LA CALIDAD TOTAL
207         Calidad_Total = 0;
208         for(i = 0; i < Fuentes_de_Alimento; i++)
```

```
209                 {
210                     Calidad_FuenteAlimento[i] = 1 /(1 + Costo_FuenteAlimento[i]);
211                     Calidad_Total += Calidad_FuenteAlimento[i];
212                 }
213
214         //LA ABEJA OBSERVADORA ES LLAMADA TANTAS VECES COMO FUENTES DE ALIMENTO SE TIENE
215         //PERO VISITA CON MAS PROBABILIDAD A LAS FUENTES DE ALIMENTO CON MEJOR CALIDAD
216         for(j = 0; j < Fuentes_de_Alimento; j++)
217         {
218           b1 = SiguienteAleatorioReal0y1(& Semilla);
219           u5 = Calidad_FuenteAlimento[0] / Calidad_Total;
220           i = 0;
221           if(b1 < u5)
222                       i = 0;
223           else
224           {
225             while(b1 > u5)
226             {
227               i++;
228               u5 += Calidad_FuenteAlimento[i] / Calidad_Total;
229             }
230           }
231           //SI LA FUENTE DE ALIMENTO i ES MEJORADA SU CONTADOR SE REINICIA CON UN VALOR DE 0
232           m6 = AbejaObservadora(i);
233           if(m6 == 0)
234                       GeneracionesSinMejora[i] = 0;
235           b1 = Menor_DesviacionPoblacional + Menor_Compacidad;
236           u5 = DesviacionPoblacional_FuenteAlimento[i] + Compacidad_FuenteAlimento[i];
237
238           //SI LA NUEVA FUENTE DE ALIMENTO ES LA MEJOR CONOCIDA SE GUARDA EN MEMORIA
239           if(u5 < b1)
240           {
241             Menor_DesviacionPoblacional = DesviacionPoblacional_FuenteAlimento[i];
242             Menor_Compacidad = Compacidad_FuenteAlimento[i];
243             for(k = 0; k < UnidadesPorConjunto; k++)
244               MejoresDistritos[k] = FuenteAlimento[i][k];
245           }
246         }
247         Generacion++;
248
249       }
250       //TERMINA EL PROCESO DE MEJORA DE UN CONJUNTO TERRITORIAL
251
252       //SE GUARDA LA INFORMACION DE LA MEJOR SOLUCION EN LA FUENTE DE ALIMENTO 0
253       for(k = 0; k < UnidadesPorConjunto; k++)
254                   FuenteAlimento[0][k] = MejoresDistritos[k];
255             DesviacionPoblacional_FuenteAlimento[0] = Menor_DesviacionPoblacional;
256             Compacidad_FuenteAlimento[0] = Menor_Compacidad;
257       //SE REALIZA UNA BUSQUEDA LOCAL
258       j = Busqueda_Local();
259
260       //SI SE ENCUENTRAN MEJORAS SE ACTUALIZA LA SOLUCION
261       Menor_DesviacionPoblacional = DesviacionPoblacional_FuenteAlimento[0];
262             Menor_Compacidad = Compacidad_FuenteAlimento[0];
263       for(k = 0; k < UnidadesPorConjunto; k++)
264                   MejoresDistritos[k] = FuenteAlimento[0][k];
265
266
267
268       //SE GUARDAN LOS MEJORES DISTRITOS CONSTRUIDOS EN LA VARIABLE DistritosFinales
269       //AL TEMINAR CADA CORRIDA EL ARREGLO DistritosFinales TENDRA EL ESCENARIO COMPLETO
270 Final:
271
272       for(i=0; i<UnidadesPorConjunto; i++)
273       {
274         for(j=0; j<6500; j++)
275         {
276           if(Conversion[j] == i)
277                       DistritosFinales[j] = MejoresDistritos[i] + DistritosAcumulados;
278         }
279
280       }
281
282       for(i=0; i< UnidadesPorConjunto; i++)
283       Distrito[i] = MejoresDistritos[i];
284       Evalua_Solucion();
285
286       for(i = 0; i < NDistritos; i++)
287       {
288         MejorDesviacionPoblacional[i + DistritosAcumulados] = DesviacionPoblacionalDistrito[i];
289         MejorCompacidad[i + DistritosAcumulados] = CompacidadDistrito[i];
290       }
291
292       printf(" Costo final %f = %f  +  0.5 * %f\n",DesviacionPoblacional_Nueva + Compacidad_Nueva,DesviacionPoblacional_Nueva, 2 *
                 Compacidad_Nueva);
293
294       //SE IMPRIME EL COSTO DE CADA CONJUNTO TERRITORIAL
295             sprintf(dummy, "Sistema_de_visualizacion\\Resumen_Costos\\ABC_Costo_Por_Conjunto.csv");
296             fp = fopen(dummy, "a");
```

```
297             fprintf(fp,"Conjunto %d,%f,%f,%f\n", ConjuntoActual, DesviacionPoblacional_Nueva + Compacidad_Nueva,
                        DesviacionPoblacional_Nueva, Compacidad_Nueva);
298             fclose(fp);
299
300
301       //SE CREA O SE ACTUALIZA EL Escenario_Hibrido
302       if(Corrida == 0 && Numero_de_Semillas > 1)
303       {
304         Mejora_Hibrida = 1;
305         for(i=0; i<UnidadesPorConjunto; i++)
306         {
307           for(j=0; j<6500; j++)
308           {
309             if(Conversion[j] == i)
310                           Solucion_Hibrida[j] = MejoresDistritos[i] + DistritosAcumulados;
311           }
312         }
313         for(i = 0; i < NDistritos; i++)
314         {
315           DesviacionPoblacional_Hibrida[i + DistritosAcumulados] = MejorDesviacionPoblacional[i + DistritosAcumulados];
316           Compacidad_Hibrida[i + DistritosAcumulados] = MejorCompacidad[i + DistritosAcumulados];
317         }
318       }
319       if(Corrida >= 1)
320       {
321         u5 = u6 = 0;
322               for(i = 0; i < NDistritos; i++)
323         {
324           u5 += DesviacionPoblacional_Hibrida[i + DistritosAcumulados] + Compacidad_Hibrida[i + DistritosAcumulados];
325           u6 += MejorDesviacionPoblacional[i + DistritosAcumulados] + MejorCompacidad[i + DistritosAcumulados];
326         }
327         if(u5 > u6)
328         {
329           Mejora_Hibrida = 1;
330           for(i=0; i<UnidadesPorConjunto; i++)
331           {
332             for(j=0; j<6500; j++)
333             {
334               if(Conversion[j] == i)
335                             Solucion_Hibrida[j] = MejoresDistritos[i] + DistritosAcumulados;
336             }
337           }
338           for(i = 0; i < NDistritos; i++)
339           {
340             DesviacionPoblacional_Hibrida[i + DistritosAcumulados] = MejorDesviacionPoblacional[i + DistritosAcumulados];
341             Compacidad_Hibrida[i + DistritosAcumulados] = MejorCompacidad[i + DistritosAcumulados];
342           }
343         }
344       }
345
346         DistritosAcumulados += NDistritos;
347     }
348     //TERMINA EL PROCESO DE MEJORA DEL ESCENARIO COMPLETO
349
350     end = clock();
351     seed1 = end - start;
352     z = seed1 / CLOCKS_PER_SEC;
353     seed1 = seed1 / (60 * CLOCKS_PER_SEC);
354     i = (int) seed1;
355     seed1 = z - (60 * i);
356     z = i / 60;
357     printf("\n\nEL TIEMPO DE EJECUCION FUE DE: %d MINUTOS %f SEGUNDOS\n\n",i,seed1);
358
359
360     for(i=0; i< 6500; i++)
361             MejoresDistritos[i] = DistritosFinales[i];
362     //SE OBTIENE EL COSTO TOTAL DE LA SOLUCION CONSTRUIDA
363     Costo_Nueva = DesviacionPoblacional_Nueva = Compacidad_Nueva = 0;
364     for(i=0; i<ConjuntosTotales; i++)
365     {
366       Costo_Nueva += MejorDesviacionPoblacional[i] + MejorCompacidad[i];
367       DesviacionPoblacional_Nueva += MejorDesviacionPoblacional[i];
368       Compacidad_Nueva += MejorCompacidad[i];
369     }
370
371     //SE IMPRIME EL COSTO TOTAL DEL ESCENARIO ACTUAL
372         sprintf(dummy, "Sistema_de_visualizacion\\Resumen_Costos\\ABC_Costo_Por_Conjunto.csv");
373         fp = fopen(dummy, "a");
374         fprintf(fp,"Costo Total,%f,%f,%f\n",Costo_Nueva, DesviacionPoblacional_Nueva, Compacidad_Nueva);
375         fclose(fp);
376
377     //SE IMPRIME EN ARCHIVO DE TEXTO LA SOLUCION FINAL
378     sprintf(dummy, "Sistema_de_visualizacion\\ABC_Escenario_%f_%d.csv", DesviacionPoblacional_Nueva + Compacidad_Nueva, Semillas[
            Corrida]);
379     fp = fopen(dummy, "w");
380     fprintf(fp,"Seccion,DISTRITO\n");
381     for(i = 0; i < 6500; i++)
382             if(MejoresDistritos[i] < 6500)
383                 fprintf(fp,"%d,%d\n",i,MejoresDistritos[i] + 1);
```

```
384     fclose(fp);
385
386     //SE IMPRIME LA SOLUCION HIBRIDA CONSTRUIDA CON LOS ESCENARIOS OBTENIDOS
387         //SI SE OBTUVO ALGUNA MEJORA Y SE EMPLEO EL SEMILLERO
388         if(Numero_de_Semillas > 1 && Mejora_Hibrida == 1)
389         {
390             Mejora_Hibrida = 0;
391             direccion = "Sistema_de_visualizacion\\Escenario_Hibrido_ABC.csv";
392             fp = fopen(direccion, "w");
393             fprintf(fp,"Seccion,DISTRITO\n");
394             for(i = 0; i < 6500; i++)
395                 if(Solucion_Hibrida[i] < 6500)
396                     fprintf(fp,"%d,%d\n",i,Solucion_Hibrida[i] + 1);
397             fclose(fp);
398         }
399   }
400   //TERMINA LA CORRIDA ACTUAL
401
402   printf("\t Se han completado las corridas solicitadas. \n\n \t Presione la tecla Enter para conluir el proceso.");
403   getchar();
404
405   return(1);
406
407 }
```

## Anexo : Función Datos(int Conjunto)

```
1  void Datos(int Conjunto)
2  //LEE LOS ARCHIVOS DE TEXTO Separar.txt, DatosConglomerados.txt y ColindanciasUnidades.txt PARA OBTENER INFORMACION SOBRE
3  //LAS UNIDADES GEOGRAFICAS QUE EMPLEARA EN CADA CONJUNTO TERRITORIAL
4  {
5    int i,k,l,m,c,mun,o,j;
6    double p;
7    int U, f, Conglomerado[10500];
8    int ConjuntoTerritorial[6500];
9    int Separar[100][2], Aux, Separadas;
10   int SeccionMun[6500];
11
12   FILE *fp;
13   char *direccion;
14   UnidadesPorConjunto = 0;
15   MediaEstatal = 0;
16
17   direccion = "Sistema_de_visualizacion\\Insumos\\Separar.txt";
18   Separadas = 0;
19   fp = fopen(direccion, "r");
20   while((c=fgetc(fp))!=EOF)
21   {
22     fscanf(fp,"%d %d",&i,&k);
23     Separar[Separadas][0] = i;
24     Separar[Separadas][1] = k;
25     Separadas++;
26     Separar[Separadas][0] = k;
27     Separar[Separadas][1] = i;
28     Separadas++;
29   }
30   fclose(fp);
31
32   for(i = 0; i < 10500; i++)
33   {
34     Conglomerado[i] = -1;
35   }
36   m = 0;
37   direccion = "Sistema_de_visualizacion\\Insumos\\DatosConglomerados.txt";
38   fp = fopen(direccion, "r");
39   while((c=fgetc(fp))!=EOF)
40   {
41     fscanf(fp,"%d %d %lf %d %d %d",&mun,&i,&p,&j,&o,&k);
42         MediaEstatal += j;
43     ConjuntoTerritorial[i] = k;
44     f = 0;
45     SeccionMun[i] = mun;
46     if(k == Conjunto)
47     {
48         if(Conglomerado[o] != -1)
49         {
50           Conversion[i] = Conglomerado[o];
51           AreaUnidadGeografica[Conglomerado[o]] += p;
52           PoblacionUnidadGeografica[Conglomerado[o]] += j;
53         }
54
```

```
55          if(Conglomerado[o] == -1)
56          {
57            Conversion[i] = m;
58            AreaUnidadGeografica[m] = p;
59            PoblacionUnidadGeografica[m] = j;
60            UnidadesPorConjunto++;
61            Conglomerado[o] = m;
62           m++;
63          }
64      }
65    }
66    fclose(fp);
67    MediaEstatal = MediaEstatal / ConjuntosTotales;
68
69    for(i=0; i<6500; i++)
70    {
71      for(j=0; j<6500; j++)
72      {
73        PerimetroFrontera[i][j] = 0;
74      }
75    }
76
77
78    for(i=0; i<6500; i++)
79    {
80      for(j=0; j<60; j++)
81      Vecinos[i][j] = 6500;
82      Vecinos[i][60] = 0;
83    }
84  //      Separar[k][0] = Separar[k][1] = -1;//ARREGLAR CUANDO SE TENGAN TIEMPOS DE TRASLADO
85    direccion = "Sistema_de_visualizacion\\Insumos\\ColindanciasUnidades.txt";
86    fp = fopen(direccion, "r");
87    while((c=fgetc(fp))!=EOF)
88    {
89      fscanf(fp,"%d %d %lf",&i,&j,&p);
90      if(ConjuntoTerritorial[i] == Conjunto)
91      {
92        Aux = 0;
93        if(j != 0)
94              {
95                  for(k=0; k<Separadas;k++)
96                  {
97                      if(SeccionMun[i] == Separar[k][0] && SeccionMun[j] == Separar[k][1])
98                      {
99                          m = j;
100                         Aux = 1;
101                         j = 0;
102                         break;
103                     }
104                 }
105             }
106       if(j != 0)
107       {
108         if(ConjuntoTerritorial[j] == Conjunto)
109         {
110           l = Conversion[i];
111           m = Conversion[j];
112           if(l != m)
113           {
114             PerimetroFrontera[l][m] += p;
115
116             f = 0;
117             for(U = 0; U < Vecinos[l][60]; U++)
118             {
119               if(Vecinos[l][U] == m)
120               {
121                 f = 1;
122                 break;
123               }
124             }
125             if(f == 0)
126             {
127               Vecinos[l][Vecinos[l][60]] = m;
128               Vecinos[l][60]++;
129             }
130           }
131         }
132
133         else
134         {
135           l = Conversion[i];
136           PerimetroFrontera[l][l] += p;
137         }
138         if(Aux == 1)
139                     j = m;
140       }
141       if(j == 0)
142       {
143         l = Conversion[i];
```

```
144         PerimetroFrontera[l][l] += p;
145       }
146     }
147   }
148   fclose(fp);
149 }
```

## Anexo : Función FuenteAlimento_Nueva(int DistritosPorConjunto)

```
1  void FuenteAlimento_Nueva(int DistritosPorConjunto)
2  //CONSTRUYE UNA SOLUCION NUEVA CON EL NUMERO DE DISTRITOS INDICADOS PARA EL CONJUNTO TERRITORIAL ACTUAL
3  //TODOS LOS DISTRITOS SON CONEXOS Y TODAS LAS UNIDADES GEOGRAFICAS PERTENCEN EXACTAMENTE A UN DISTRITO
4  {
5    int i, j, k,l,pp,u[45],contador[6500],seed, z, i2, j2;
6    int Unidades[6500],Distrito_Auxiliar[6500];
7    for(i = 0; i < UnidadesPorConjunto; i++)
8    {
9      Distrito[i] = -1;
10     Distrito_Auxiliar[i] = -1;
11   }
12
13   //SE OBTIENEN UnidadesPorConjunto UNIDADES GEOGRAFICAS PARA EL CONJUNTO TERRITORIAL ACTUAL
14   for(i = 0; i < UnidadesPorConjunto; i++)
15   {
16     Unidades[i] = i;
17         contador[i]=0;
18   }
19   j = UnidadesPorConjunto;
20   for(k = 0; k < DistritosPorConjunto; k++)
21   {
22     i = SiguienteAleatorioEnteroModN(& Semilla, j);
23     u[k] = Unidades[i];
24     contador[u[k]] = 1;
25
26     //SE INICIALIZAN LOS DISTRITOS CON LAS UNIDADES SELECCIONADAS
27     Distrito[Unidades[i]] = k;
28     j--;
29     for(l = i; l < j; l++)
30     {
31       Unidades[l] = Unidades[l + 1];
32     }
33   }
34
35   //EMPIEZA CONSTRUCCION DE SOLUCION INICIAL
36   j = 0;
37   while(j != UnidadesPorConjunto)
38   {
39       k = SiguienteAleatorioEnteroModN(& Semilla,DistritosPorConjunto);
40     //ENCUENTRA LAS COLINDANCIAS DEL DISTRITO k
41     for(i=0; i<UnidadesPorConjunto; i++)
42     {
43       if(Distrito[i] == k)
44       {
45         for(j=0; j<Vecinos[i][60]; j++)
46         {
47           if(contador[Vecinos[i][j]] == 0)
48                         Distrito_Auxiliar[Vecinos[i][j]] = k;
49         }
50       }
51     }
52     pp = 0;
53     for(j=0; j<UnidadesPorConjunto; j++)
54     {
55       //CUENTA A TODOS LOS VECINOS QUE SE PUEDEN AGREGAR AL DISTRITO k
56       if(Distrito_Auxiliar[j] == k)
57                   pp++;
58     }
59     if(pp == 1)
60                 seed = 0;
61     if(pp > 1)
62                 seed = SiguienteAleatorioEnteroModN(& Semilla,pp);
63     if(pp > 0)
64     {
65       z = 0;
66       //SELECCIONA A UN VECINO DEL DISTRITO k Y LO AGREGA
67       for(j=0; j<UnidadesPorConjunto; j++)
68       {
69         if(Distrito_Auxiliar[j] == k)
70         {
71           if( z == seed)
72           {
```

```
            contador[j] = 1;
            Distrito[j] = k;
            break;
          }
          z++;
        }
      }
    }
    j = 0;
    for(i=0; i<UnidadesPorConjunto; i++)
    {
      Distrito_Auxiliar[i] = -1;
      j += contador[i];
      }
  }
}
```

## Anexo : Función Costo_FuenteNueva(int AB)

```
void Costo_FuenteNueva(int AB)
{

  int i, j, k,p;

  for(i=0;i<NDistritos;i++)
  {
    PerimetroDistrito[i] = 0;
    PoblacionDistrito[i] = 0;
    AreaDistrito[i] = 0;
  }


  for(j=0;j<UnidadesPorConjunto;j++)
  {
    PoblacionDistrito[Distrito[j]] += PoblacionUnidadGeografica[j];
    AreaDistrito[Distrito[j]] += AreaUnidadGeografica[j];
    PerimetroDistrito[Distrito[j]] += PerimetroFrontera[j][j];
    for(k = 0;k < Vecinos[j][60];k++)
    {
      p = Vecinos[j][k];
      if(Distrito[p] != Distrito[j] )
              PerimetroDistrito[Distrito[j]] += PerimetroFrontera[j][p];
    }

  }

  DesviacionPoblacional_FuenteAlimento[AB] = Compacidad_FuenteAlimento[AB] = 0;
  for(i=0;i<NDistritos;i++)
  {
    DesviacionPoblacionalDistrito[i] = Desviacion_Poblacional(PoblacionDistrito[i]);
    CompacidadDistrito[i] = Compacidad(AreaDistrito[i],PerimetroDistrito[i]);
    DesviacionPoblacional_FuenteAlimento[AB] += DesviacionPoblacionalDistrito[i];
    Compacidad_FuenteAlimento[AB] += CompacidadDistrito[i];
  }

  Costo_FuenteAlimento[AB] = DesviacionPoblacional_FuenteAlimento[AB] + Compacidad_FuenteAlimento[AB];

}
```

## Anexo : Función AbejaEmpleada(int AB)

```
int AbejaEmpleada(int AB)
{

  int i, j, n, n1;
  int Origen, Destino, Unidad_Elegida;
  int destinosE[30], Candidatos[6500];
  double b1, u5;

  //SE ELIGE UN DISTRITO AL AZAR CON MAS DE UNA UNIDAD GEOGRAFICA
  j = 1;
  while(j <= 1)
  {
    Origen = SiguienteAleatorioEnteroModN(& Semilla, NDistritos);
```

```
    j = Cardinalidad_Distrito(AB,Origen);
  }
  //SE COPIAN LOS DATOS DE LA FUENTE DE ALIMENTO FuenteAlimento[AB][] EN LA VARIABLE Distrito[]
  for(i = 0; i < UnidadesPorConjunto; i++)
  {
    Distrito[i] = FuenteAlimento[AB][i];
    Candidatos[i] = 6500;
  }

  //SE HACE UNA LISTA CON LAS UG QUE SE PUEDEN CAMBIAR DEL DISTRITO Origen
  n = 0;
  for(i = 0; i < UnidadesPorConjunto; i++)
  {
    if(Distrito[i] == Origen)
    {
      n1 = j = 0;
      while(n1 < 6500)
      {
        n1 = Vecinos[i][j];
        if(Vecinos[i][j] < 6500)
        {
          if(Distrito[Vecinos[i][j]] != Origen)
          {
            Candidatos[n] = i;
            n++;
            n1 = 6500;
          }
        }
        j++;
      }
    }
  }

  //SE ELIGE UNA UNIDAD PARA SER CAMBIADA DE DISTRITO
    //SE ELIGE CON UNA PROBABILIDAD 1/n
  i = SiguienteAleatorioEnteroModN(& Semilla, n);
    Unidad_Elegida = Candidatos[i];


  //SE DETERMINAN LOS DISTRITOS VECINOS DE Unidad_Elegida
  n1 = 0;
  for(i=0; i<Vecinos[Unidad_Elegida][60]; i++)
  {
    if(Distrito[Vecinos[Unidad_Elegida][i]] != Origen)
    {
      if(n1 > 0)
      {
        j = 0;
        for(n = 0; n <n1; n++)
        {
          if(destinosE[n] != Distrito[Vecinos[Unidad_Elegida][i]])
                        j++;
        }
        if(j == n1)
        {
          destinosE[n1] = Distrito[Vecinos[Unidad_Elegida][i]];
          n1++;
        }
      }
      if(n1 == 0)
      {
        destinosE[n1] = Distrito[Vecinos[Unidad_Elegida][i]];
        n1++;
      }
    }
  }

  //SE ELIGE UN DISTRITO DESTINO PARA Unidad_Elegida
  //SE ELIGE CON UNA PROBABILIDAD 1/n
  if(n1 == 1)
        Destino = 0;
    else
        Destino = SiguienteAleatorioEnteroModN(& Semilla, n1);

  Destino = destinosE[Destino];
  Distrito[Unidad_Elegida] = Destino;

  //SE REVISA SI SE PROVOCO ALGUNA DISCONEXION
  j = RevisaConexidad_Empleada(Origen, Destino);

  //SE EVALUA EL COSTO DEL NUEVO ESCENARIO
  Evalua_Solucion();

  b1 = DesviacionPoblacional_FuenteAlimento[AB] + Compacidad_FuenteAlimento[AB];
  u5 = DesviacionPoblacional_Nueva + Compacidad_Nueva;

  //LA NUEVA SOLUCION ES ACEPTADA SI MEJORA EL COSTO DE FuenteAlimento[AB][]
  if(u5 < b1)
  {
```

```
    DesviacionPoblacional_FuenteAlimento[AB] = DesviacionPoblacional_Nueva;
    Compacidad_FuenteAlimento[AB] = Compacidad_Nueva;
    Costo_FuenteAlimento[AB] = Costo_Nueva;
    for(i = 0; i < UnidadesPorConjunto; i++)
    {
      FuenteAlimento[AB][i] = Distrito[i];
    }
    Evalua_Solucion();
    return(0);
  }

  return(1);

}
```

## Anexo : Función Busqueda_Local()

```


int Busqueda_Local()
{
    int i, j, m, n, n1;
  int Origen, Destino, Mejor_Destino, Unidad_Elegida;
  int destinosE[30], Mejora, Cardinalidad;
  double b1, u5;


  //SE COPIAN LOS DATOS DE LA FUENTE DE ALIMENTO FuenteAlimento[0][] EN LA VARIABLE Distrito[]
  for(i = 0; i < UnidadesPorConjunto; i++)
    Distrito[i] = FuenteAlimento[0][i];

    m = 0;
    Mejor_Destino = -1;
    Mejora = 0;
    b1 = DesviacionPoblacional_FuenteAlimento[0] + Compacidad_FuenteAlimento[0];

  while(m == 0)
    {
        m = 1;
        for(Unidad_Elegida = 0; Unidad_Elegida < UnidadesPorConjunto; Unidad_Elegida++)
        {
            Origen = Distrito[Unidad_Elegida];

            //SE REVISA QUE EL DISTRITO CONTENGA MAS DE 2 UNIDADES GEOGRAFICAS
            Cardinalidad = Cardinalidad_Distrito(0,Origen);

            if(Cardinalidad >= 2)
            {

                //SE DETERMINAN LOS DISTRITOS VECINOS DE Unidad_Elegida
                n1 = 0;
                for(i=0; i<Vecinos[Unidad_Elegida][60]; i++)
                {
                    if(Distrito[Vecinos[Unidad_Elegida][i]] != Origen)
                    {
                        if(n1 > 0)
                        {
                            j = 0;
                            for(n = 0; n <n1; n++)
                            {
                                if(destinosE[n] != Distrito[Vecinos[Unidad_Elegida][i]])
                                    j++;
                            }
                            if(j == n1)
                            {
                                destinosE[n1] = Distrito[Vecinos[Unidad_Elegida][i]];
                                n1++;
                            }
                        }
                        if(n1 == 0)
                        {
                            destinosE[n1] = Distrito[Vecinos[Unidad_Elegida][i]];
                            n1++;
                        }
                    }
                }

                //SE REVISAN LOS POSIBLES CAMBIOS DE Unidad_Elegida A DISTRITOS VECINOS
                for(i = 0; i < n1; i++)
                {
                    Destino = destinosE[i];
```

```
65                      Distrito[Unidad_Elegida] = Destino;
66
67                      //SE REVISA SI SE PROVOCO ALGUNA DISCONEXION
68                      j = RevisaConexidad_Empleada(Origen, Destino);
69
70                      //SE EVALUA EL COSTO DEL NUEVO ESCENARIO
71                      Evalua_Solucion();
72
73                      u5 = DesviacionPoblacional_Nueva + Compacidad_Nueva;
74
75                      //SI MEJORA EL COSTO DE FuenteAlimento[0][] SE GUARDA LA INFORMACION
76                      if(u5 < b1)
77                      {
78                          Mejor_Destino = Destino;
79                          b1 = u5;
80                      }
81
82                      //SE REGRESA LA INFORMACION A SU ESTADO INICIAL PARA EL SIGUIENTE CAMBIO
83                      for(j = 0; j < UnidadesPorConjunto; j++)
84                      {
85                          Distrito[j] = FuenteAlimento[0][j];
86                      }
87                  }
88
89                  //SI SE ENCONTRO UNA MEJORA SE REALIZA EL CAMBIO
90                  if(Mejor_Destino != -1)
91                  {
92                      Distrito[Unidad_Elegida] = Destino = Mejor_Destino;
93
94                      //SE REVISA SI SE PROVOCO ALGUNA DISCONEXION
95                      j = RevisaConexidad_Empleada(Origen, Destino);
96
97                      //SE EVALUA EL COSTO DEL NUEVO ESCENARIO
98                      Evalua_Solucion();
99
100                     //LA NUEVA SOLUCION ES ACEPTADA PORQUE MEJORA EL COSTO DE FuenteAlimento[AB][]
101                     DesviacionPoblacional_FuenteAlimento[0] = DesviacionPoblacional_Nueva;
102                     Compacidad_FuenteAlimento[0] = Compacidad_Nueva;
103                     Costo_FuenteAlimento[0] = Costo_Nueva;
104                     for(i = 0; i < UnidadesPorConjunto; i++)
105                     {
106                         FuenteAlimento[0][i] = Distrito[i];
107                     }
108
109                     Mejor_Destino = -1;
110                     Mejora = 1;
111                     m = 0;
112                 }
113             }
114         }
115     }
116     return(Mejora);
117 }
```

## Anexo : Función Cardinalidad_Distrito(int Fuente, int Z)

```
int Cardinalidad_Distrito(int Fuente, int Z)
{
  int i, suma = 0;

  for(i = 0; i < UnidadesPorConjunto; i++)
  {
    if(FuenteAlimento[Fuente][i] == Z)
            suma++;
    if(suma > 1)
            break;
  }
  return(suma);
}
```

## Anexo : Función RevisaConexidad_Empleada(int Origen, int Destino)

```
int RevisaConexidad_Empleada(int Origen, int Destino)
//CON ESTA FUNCION SE REVISA SI EL DISTRITO Origen PERDIO LA CONEXIDAD
{
  int j,i,m,suma;
  int Representante[6500], Componente[6500],N = 0,n;
  int Lista1[6500], Lista2[6500];
  int Contactado[6500];

  suma = 0;

    //SE REVISA EL NUMERO DE COMPONENTES CONEXAS EN EL DISTRITO
  for(m=0; m<UnidadesPorConjunto; m++)
  {
    Contactado[m] = 0;
    Representante[m] = -1;
    Componente[m] = 0;
    if(Distrito[m] == Origen)
    {
      Lista1[suma] = m;
      suma++;
    }
  }

    //SI EL NUMERO DE COMPONENTES ES MENOR O IGUAL A 1 NO SE HA PERDIDO LA CONEXIDAD
  if( suma == 0)
  {
    printf("\n ERROR Distrito vacia\n");
    getchar();
    return(0);
  }

  if( suma == 1)
        return(0);

    //SI EL NUMERO DE COMPONENTES CONEXAS ES MAYOR O IGUAL A 2 SE DEBE REPARAR LA CONEXIDAD
  for(i=0; i< suma; i++)
  {
    n = 0;
    if(Contactado[Lista1[i]] == 0)
    {
      Representante[N] = Lista1[i];
      Lista2[n] = Lista1[i];
      Contactado[Lista2[n]] = 1;
      n++;
      for(j=0; j < n; j++)
      {
        for(m=0; m < Vecinos[Lista2[j]][60]; m++)
        {
          if(Distrito[Vecinos[Lista2[j]][m]] == Origen && Contactado[Vecinos[Lista2[j]][m]] == 0)
          {
            Lista2[n] = Vecinos[Lista2[j]][m];
            Contactado[Vecinos[Lista2[j]][m]] = 1;
            n++;
          }
        }
      }
      Componente[N] = n;
      N++;
    }
  }

```

```
62    if( N == 1)
63          return(0);
64
65    n = j = 0;
66    j = Componente[0];
67
68      //SE DETERMINA CUAL ES LA COMPONENTE CON MAYOR NUMERO DE UNIDADES GEOGRAFICAS
69    //PARA DEJARLA COMO EL DISTRITO Origen
70    for(i = 1; i < N; i++)
71    {
72      if(Componente[i] > j)
73      {
74        j = Componente[i];
75        n = i;
76      }
77    }
78
79      //EL RESTO DE LAS COMPONENTES SON ENVIADAS AL DISTRITO Destino
80    for(i = 0; i < N; i++)
81    {
82      if(i != n)
83            ReparaConexidad_Empleada(Origen,Representante[i], Destino);
84    }
85    return(0);
86
87  }
```

## Anexo : Función ReparaConexidad_Empleada(int Origen, int Unidad, int Destino)

```
int ReparaConexidad_Empleada(int Origen, int Unidad, int Destino)
//ESTA FUNCION ENVIA A LAS UNIDADES QUE FORMAN UNA COMPONENTE CONEXA JUNTO CON Unidad DE
//DISTRITO Origen A DISTRITO Destino
{
  int j,i,m,n;
  int Lista[6500];
  int ComponenteConexa[6500];

  for(m=0; m<UnidadesPorConjunto; m++)
        ComponenteConexa[m] = 0;

  Lista[0] = Unidad;
  ComponenteConexa[Lista[0]] = 1;
  n = 1;
    //CON EL SIGUIENTE CICLO SE ENCUENTRA A TODAS LAS UNIDADES GEOGRAFICAS EN LA MISMA COMPONENTE CONEXA QUE Unidad
  for(j=0; j < n; j++)
  {
    for(m=0; m < Vecinos[Lista[j]][60]; m++)
    {
      if(Distrito[Vecinos[Lista[j]][m]] == Origen && ComponenteConexa[Vecinos[Lista[j]][m]] == 0)
      {
        Lista[n] = Vecinos[Lista[j]][m];
        ComponenteConexa[Vecinos[Lista[j]][m]] = 1;
        n++;
      }
    }
  }
    //TODAS LAS UNIDADES GEOGRAFICAS EN LA COMPONENTE CONEXA DE Unidad SON ENVIADAS AL DISTRITO Destino
  for(i = 0; i < UnidadesPorConjunto; i++)
  {
    if(ComponenteConexa[i] == 1)
      Distrito[i] = Destino;
  }


  return(0);

}
```

## Anexo : Función AbejaObservadora(int AB)

```
int AbejaObservadora(int AB)
{
  int b, i, j, k, n, n1, n2, n3, m2, o, o2, o3,AB1;
  int Candidatos[6500], Candidatos1[6500], Candidatos2[6500], Candidatos3[6500];
  double b1, u5;
  int OrigenAB, OrigenAB1, Unidad_Elegida;
  int ii,kk;

  //SE ELIGE UN DISTRITO AL AZAR DE LA FUENTE DE ALIMENTO AB
  OrigenAB = SiguienteAleatorioEnteroModN(& Semilla, NDistritos);

  //SE HACE UNA LISTA CON LAS UG DE DISTRITO Origen
  n = 0;

  for(i = 0; i < UnidadesPorConjunto; i++)
  {
    Distrito[i] = FuenteAlimento[AB][i];
    Candidatos[i] = 6500;
    if(Distrito[i] == OrigenAB)
    {
      Candidatos[n] = i;
      n++;
    }
  }
  if( n == 1)
        k = Candidatos[0];

  //SE ELIGEN UNA UNIDAD DE DISTRITO Origen CON PROBABILIDAD 1/n
  else
  {
    b = SiguienteAleatorioEnteroModN(& Semilla, n);
    k = Candidatos[b];
  }
    //SE ELIGE UNA FUENTE DE ALIMENTO, AB1, DIFERENTE DE AB
    AB1 = AB;
    while(AB1 == AB)
```

```
37      {
38          AB1 = SiguienteAleatorioEnteroModN(& Semilla, Fuentes_de_Alimento);
39      }
40    OrigenAB1 = FuenteAlimento[AB1][k];
41
42      if(n >= 2)
43      {
44          //SE LE QUITA UNA UNIDAD A LA FUENTE DE ALIMENTO OrigenAB
45          //Candidatos1[] GUARDA LAS UNIDADES QUE ESTAN EN OrigenAB PERO NO ESTAN EN OrigenAB1
46
47          n1 = 0;
48          for(j = 0; j < UnidadesPorConjunto; j++)
49          {
50              if(Distrito[j] == OrigenAB && FuenteAlimento[AB1][j] != OrigenAB1)
51              {
52                  Candidatos1[n1] = j;
53                  n1++;
54              }
55          }
56
57          //Candidatos2[] GUARDA LAS UNIDADES DE Candidatos1[] QUE ESTAN EN LA FRONTERA DE OrigenAB
58          n2 = 0;
59          for(j = 0; j < n1; j++)
60          {
61              o = Candidatos1[j];
62              for(o2 = 0; o2 < Vecinos[o][60]; o2++)
63              {
64                  o3 = Vecinos[o][o2];
65                  if(Distrito[o3] != OrigenAB )
66                  {
67                      Candidatos2[n2] = o;
68                      n2++;
69                      break;
70                  }
71              }
72          }
73
74          if(n2 == 0)
75              j = -1;
76          if(n2 == 1)
77              j = 0;
78          if(n2 > 1)
79          {
80              //SE ELIGE CON UNA OPCION CON PROBABILIDAD 1/n
81              j = SiguienteAleatorioEnteroModN(& Semilla, n2);
82          }
83
84          if(j > -1)
85          {
86              kk = 0;
87              Unidad_Elegida = Candidatos2[j];
88              for(o2 = 0; o2 < Vecinos[Unidad_Elegida][60]; o2++)
89              {
90                  o3 = Vecinos[Unidad_Elegida][o2];
91                  if(Distrito[o3] != Distrito[Unidad_Elegida])
92                  {
93                      Candidatos1[kk] = Distrito[o3];
94                      kk++;
95                  }
96              }
97              //SE SELECCIONA UN DISTRITO PARA ENVIAR A LA UNIDAD GEOGRAFICA SELECCIONADA
98              if(kk == 1)
99                  Distrito[Unidad_Elegida] = Candidatos1[0];
100             if(kk > 1)
101             {
102                 //SE ELIGE CON UNA PROBABILIDAD 1/n
103                 kk = SiguienteAleatorioEnteroModN(& Semilla, kk);
104                 kk = Candidatos1[kk];
105                 Distrito[Unidad_Elegida] = kk;
106
107             }
108         }
109
110         //SE REVISA LA CONEXIDAD DE DISTRITO OrigenAB
111         //EN CASO DE PERDERSE, EL NUEVO DISTRITO SERA LA COMPONENTE CONEXA QUE CONTENGA A LA UNIDAD k
112         RevisaConexidad_Observadora2(OrigenAB,k);
113     }
114
115   //SE LE AGREGA UNA UNIDAD A LA FUENTE DE ALIMENTO OrigenAB
116   //PRIMERO SE BUSCAN LAS UNIDADES QUE NO ESTAN EN OrigenAB PERO SI ESTAN EN OrigenAB1
117   n1 = 0;
118   for(j = 0; j < UnidadesPorConjunto; j++)
119   {
120     if(Distrito[j] != OrigenAB && FuenteAlimento[AB1][j] == OrigenAB1)
121     {
122       Candidatos1[n1] = j;
123       n1++;
124     }
125   }
```

```
126
127     //EN Candidatos2[] SE INCLUYEN LAS UNIDADES DE Candidatos1[] QUE SON VECINOS DE OrigenAB
128   n2 = 0;
129   for(j = 0; j < n1; j++)
130   {
131     o = Candidatos1[j];
132
133     for(o2 = 0; o2 < Vecinos[o][60]; o2++)
134     {
135       o3 = Vecinos[o][o2];
136       if(Distrito[o3] == OrigenAB)
137       {
138         Candidatos2[n2] = o;
139         n2++;
140         break;
141       }
142     }
143   }
144
145   n3 = 0;
146   for(j = 0; j < n2; j++)
147   {
148     o = Candidatos2[j];
149     m2 = 0;
150     for(ii = 0; ii < UnidadesPorConjunto; ii++)
151     //SE REVISA CUALES UNIDADES DE Candidatos2[] PERTENECEN A UN DISTRITO CON MAS DE DOS UNIDADES GEOGRAFICAS
152     {
153       if(Distrito[ii] == Distrito[o])
154                 m2++;
155       if(m2 > 2)
156                 break;
157     }
158     //Candidatos3[] CONTIENE UNIDADES DE Candidatos2[] QUE PERTENECEN A UN DISTRITO CON MAS DE DOS UNIDADES GEOGRAFICAS
159     if(m2 > 2)
160     {
161       Candidatos3[n3] = o;
162       n3++;
163     }
164   }
165
166   if(n3 == 0)//NO HAY CAMBIOS POSIBLES
167         j = -1;
168   if(n3 == 1)//SOLO HAY UN POSIBLE CAMBIO
169         j = 0;
170   if(n3 > 1)//HAY DOS O MAS CAMBIOS POSIBLES
171   {
172       //SE ELIGE UNA UNIDAD GEGRAFICA CON PROBABILIDAD 1/n
173       j = SiguienteAleatorioEnteroModN(& Semilla, n3);
174   }
175
176   if(j > -1)
177   {
178     Unidad_Elegida = Candidatos3[j];
179     m2 = Distrito[Unidad_Elegida];
180     Distrito[Unidad_Elegida] = OrigenAB;
181
182     //EL CANDIDATO Unidad_Elegida ES CAMBIADO AL DISTRITO OrigenAB
183     kk = -1;
184     for(j = 0; j < UnidadesPorConjunto; j++)
185     //SE CUENTA EL NUMERO DE UNIDADES EN EL DISTRITO QUE ACABA DE ABANDONAR Unidad_Elegida
186     //QUE NO ESTAN EN EL MISMO DISTRITO QUE LA UNIDAD k EN LA FUENTE DE ALIMENTO AB1
187     {
188       if(Distrito[j] == m2 && FuenteAlimento[AB1][j] != FuenteAlimento[AB1][k])
189       {
190         kk++;
191         Candidatos1[kk] = j;
192       }
193     }
194
195     if(kk < 0)
196             RevisaConexidad_Observadora1(m2);//REVISA CONEXIDAD Y EN CASO DE HABERSE PERDIDO
197                                              //LA COMPONENTE CONEXA MAS GRANDE SERA EL NUEVO DISTRITO m2
198     else
199     {
200       if(kk == 0)
201                 kk = Candidatos1[0];
202       else
203       {
204         kk++;
205         kk = SiguienteAleatorioEnteroModN(& Semilla, kk);
206         kk = Candidatos1[kk];
207       }
208       RevisaConexidad_Observadora2(m2,kk); //REVISA CONEXIDAD Y EN CASO DE HABERSE PERDIDO
209                                            //LA COMPONENTE CONEXA CON LA UNIDAD kk SERA EL NUEVO DISTRITO m2
210     }
211   }
212
213   Evalua_Solucion();
214   b1 = DesviacionPoblacional_FuenteAlimento[AB] + Compacidad_FuenteAlimento[AB];
```

```
  u5 = DesviacionPoblacional_Nueva + Compacidad_Nueva;

  //SI LA NUEVA SOLUCION TIENE MEJOR CALIDAD SE ACEPTA Y REEMPLAZA A LA FUENTE DE ALIMENTO AB
  if(u5 < b1 && u5 != (DesviacionPoblacional_FuenteAlimento[AB1] + Compacidad_FuenteAlimento[AB1]))
  {
    DesviacionPoblacional_FuenteAlimento[AB] = DesviacionPoblacional_Nueva;
    Compacidad_FuenteAlimento[AB] = Compacidad_Nueva;
    Costo_FuenteAlimento[AB] = Costo_Nueva;
    for(i = 0; i < UnidadesPorConjunto; i++)
    {
      FuenteAlimento[AB][i] = Distrito[i];
    }
    return(0);
  }
  return(1);
}
```

## Anexo : Función RevisaConexidad_Observadora1(int DistritoAnalizado)

```
int RevisaConexidad_Observadora1(int DistritoAnalizado)
//CON ESTA FUNCION SE REVISA SI EL DistritoAnalizado PERDIO LA CONEXIDAD
{
  int j,i,m,suma;
  int Representante[6500], Componente[6500],N = 0,n;
  int Lista1[6500], Lista2[6500];
  int Contactado[6500];

  suma = 0;

  //SI EL NUMERO DE UNIDADES GEOGRAFICAS ES MENOR O IGUAL A 1 NO SE HA PERDIDO LA CONEXIDAD
  for(m=0;m<UnidadesPorConjunto;m++)
  {
    Contactado[m] = 0;
    Componente[m] = 0;
    if(Distrito[m] == DistritoAnalizado)
    {
      Lista1[suma] = m;
      suma++;
    }
  }

  if ( suma <= 1)
    return(0);

  //SE REVISA EL NUMERO DE COMPONENTES CONEXAS EN EL DISTRITO
  //SI EL NUMERO DE COMPONENTES CONEXAS ES IGUAL A 1 NO SE HA PERDIDO LA CONEXIDAD
  //SI EL NUMERO DE COMPONENTES CONEXAS ES MAYOR O IGUAL A 2 SE DEBE REPARAR LA CONEXIDAD
  for(i=0; i< suma; i++)
  {
    n = 0;
    if(Contactado[Lista1[i]] == 0)
    {
      Representante[N] = Lista1[i];
      Lista2[n] = Lista1[i];
      Contactado[Lista2[n]] = 1;
      Componente[N]++;
      n++;
      for(j=0; j < n; j++)
      {
        for(m=0; m < Vecinos[Lista2[j]][60]; m++)
        {
          if(Distrito[Vecinos[Lista2[j]][m]] == DistritoAnalizado && Contactado[Vecinos[Lista2[j]][m]] == 0)
          {
            Lista2[n] = Vecinos[Lista2[j]][m];
            Contactado[Vecinos[Lista2[j]][m]] = 1;
            Componente[N]++;
            n++;
          }
        }
      }
      N++;
    }
  }

  if( N == 1)
        return(0);

  //SE DETERMINA CUAL ES LA COMPONENTE CON MAYOR NUMERO DE UNIDADES GEOGRAFICAS
  //PARA DEJARLA COMO EL DISTRITO Origen
  n = 0;
  j = Componente[n];
```

```
63    for(i = 0; i < N; i++)
64    {
65      if(Componente[i] > j)
66      {
67        j = Componente[i];
68        n = i;
69      }
70    }
71
72    for(i = 0; i < N; i++)
73    {
74      if(i != n)
75        ReparaConexidad_Observadora(DistritoAnalizado,Representante[i]);
76    }
77
78    return(0);
79  }
```

## Anexo : Función RevisaConexidad_Observadora2(int DistritoOrigen, int UnidadOrigen)

```
1   int RevisaConexidad_Observadora2(int DistritoOrigen, int UnidadOrigen)
2   //CON ESTA FUNCION SE REVISA SI EL DISTRITO Origen PERDIO LA CONEXIDAD
3   {
4     int j,i,m,suma;
5     int Representante[6500], Componente,N = 0,n;
6     int Lista1[6500], Lista2[6500];
7     int Contactado[6500];
8     suma = 0;
9
10    //SE REVISA EL NUMERO DE UNIDADES GEOGRAFICAS EN EL DISTRITO
11    for(m=0;m<UnidadesPorConjunto;m++)
12    {
13      Contactado[m] = 0;
14      Representante[m] = -1;
15      if(Distrito[m] == DistritoOrigen)
16      {
17        Lista1[suma] = m;
18        suma++;
19      }
20    }
21
22    //SI EL NUMERO DE UNIDADES GEOGRAFICAS ES MENOR O IGUAL A 1 NO SE HA PERDIDO LA CONEXIDAD
23    if ( suma == 0)
24    {
25      printf("\n ERROR Distrito vacia\n");
26      getchar();
27      return(0);
28    }
29
30    if ( suma == 1)
31    {
32      return(0);
33    }
34
35    //SE REVISA EL NUMERO DE COMPONENTES CONEXAS EN EL DISTRITO
36    //SI EL NUMERO DE COMPONENTES CONEXAS ES IGUAL A 1 NO SE HA PERDIDO LA CONEXIDAD
37    //SI EL NUMERO DE COMPONENTES CONEXAS ES MAYOR O IGUAL A 2 SE DEBE REPARAR LA CONEXIDAD
38    for(i=0; i< suma; i++)
39    {
40      n = 0;
41      if(Contactado[Lista1[i]] == 0)
42      {
43        Representante[N] = Lista1[i];
44        Lista2[n] = Lista1[i];
45        Contactado[Lista2[n]] = 1;
46        if(Lista2[n] == UnidadOrigen)
47                  Componente = N;
48
49        n++;
50        for(j=0; j < n;j++)
51        {
52          for(m=0; m < Vecinos[Lista2[j]][60]; m++)
53          {
54            if(Distrito[Vecinos[Lista2[j]][m]] == DistritoOrigen && Contactado[Vecinos[Lista2[j]][m]] == 0)
55            {
56              Lista2[n] = Vecinos[Lista2[j]][m];
57              Contactado[Vecinos[Lista2[j]][m]] = 1;
58              if(Lista2[n] == UnidadOrigen)
59                          Componente = N;
```

```
            n++;
          }
        }
      }
      N++;
    }
  }

  if( N == 1)
        return(0);

    //LAS UNIDADES SON ENVIADAS A DISTRITOS VECINOS, EXCEPTO LAS UBICADAS EN Componente
  for(i = 0; i < N; i++)
  {
    if(i != Componente)
          ReparaConexidad_Observadora(DistritoOrigen,Representante[i]);
  }
  return(0);
}
```

## Anexo : Función ReparaConexidad_Observadora(int DistritoOrigen, int UnidadOrigen)

```
int ReparaConexidad_Observadora(int DistritoOrigen, int UnidadOrigen)
//ESTA FUNCION ES LLAMADA CUANDO SE HA COMPROBADO FALTA DE CONEXIDAD EN DistritoOrigen
//LA UNIDAD GEOGRAFICA UnidadOrigen ES UN REPRESENTANTE DE UNA COMPONENTE CONEXA DE DistritoOrigen
{
  int j,i,m,n0;
  int Lista2[6500];
  int Destinos[45],Destinos2[45],mm;
  int Contactado[6500];

  for(m = 0;m< NDistritos;m++)
  {
    Destinos[m] = 0;
    Destinos2[m] = -1;
  }
  for(m=0;m<UnidadesPorConjunto;m++)
        Contactado[m] = 0;

  n0 = 0;
  Lista2[n0] = UnidadOrigen;
  Contactado[Lista2[n0]] = 1;
  n0++;

  //SE ENCUENTRA A TODAS LAS UNIDADES GEOGRAFICAS EN LA MISMA COMPONENTE CONEXA QUE UnidadOrigen
  //SE HACE UNA LISTA CON TODOS LOS DISTRITOS A LOS QUE SE PUEDE ENVIAR A LA COMPONENTE
  mm = 0;
  for(j=0; j < n0;j++)
  {
    for(m=0; m < Vecinos[Lista2[j]][60]; m++)
    {
      if(Distrito[Vecinos[Lista2[j]][m]] == DistritoOrigen && Contactado[Vecinos[Lista2[j]][m]] == 0)
      {
        Lista2[n0] = Vecinos[Lista2[j]][m];
        Contactado[Vecinos[Lista2[j]][m]] = 1;
        n0++;
      }

      if(Distrito[Vecinos[Lista2[j]][m]] != DistritoOrigen && Destinos[Distrito[Vecinos[Lista2[j]][m]]] == 0)
      {
        Destinos[Distrito[Vecinos[Lista2[j]][m]]] = 1;
        Destinos2[mm] = Distrito[Vecinos[Lista2[j]][m]];
        mm++;
      }
    }
  }

    //SE ELIGE UN DESTINO PARA LAS UNIDADES GEOGRAFICAS EN LA COMPONENTE CONEXA
  if(mm > 1)
        mm = SiguienteAleatorioEnteroModN(&Semilla,mm);
  else
        mm = 0;
  m = Destinos2[mm];

  //LAS UNIDADES SON ENVIADAS A LA COMPONENTE ELEGIDA
  for(i = 0; i < UnidadesPorConjunto; i++)
  {
    if(Contactado[i] == 1)
          Distrito[i] = m;
```

```
58    }
59
60    return(0);
61  }
```

## Anexo : Función Evalua_Solucion(void)

```
void Evalua_Solucion(void)
{
  int i, j, k, p;

  for(i=0;i<NDistritos;i++)
  {
    PerimetroDistrito[i] = 0;
    PoblacionDistrito[i] = 0;
    AreaDistrito[i] = 0;
  }

  for(j=0;j<UnidadesPorConjunto;j++)
  {
    PoblacionDistrito[Distrito[j]] += PoblacionUnidadGeografica[j];
    AreaDistrito[Distrito[j]] += AreaUnidadGeografica[j];
    PerimetroDistrito[Distrito[j]] += PerimetroFrontera[j][j];
    for(k = 0;k < Vecinos[j][60];k++)
    {
      p = Vecinos[j][k];
      if(Distrito[p] != Distrito[j] )
              PerimetroDistrito[Distrito[j]] += PerimetroFrontera[j][p];
    }
  }

  DesviacionPoblacional_Nueva = Compacidad_Nueva = 0;
  for(i=0;i<NDistritos;i++)
  {
    DesviacionPoblacionalDistrito[i] = Desviacion_Poblacional(PoblacionDistrito[i]);
    CompacidadDistrito[i] = Compacidad(AreaDistrito[i],PerimetroDistrito[i]);
    DesviacionPoblacional_Nueva += DesviacionPoblacionalDistrito[i];
    Compacidad_Nueva += CompacidadDistrito[i];
  }
  Costo_Nueva = DesviacionPoblacional_Nueva + Compacidad_Nueva;
}
```

## Anexo : Función Desviacion_Poblacional(int Poblacion)

```
double Desviacion_Poblacional(int Poblacion)
//CALCULA EL EQUILIBRIO POBLACIONAL
{
  double PoblacionEstatal;
  PoblacionEstatal = 1.00 - (Poblacion / MediaEstatal);
  PoblacionEstatal = PoblacionEstatal / 0.15;
  PoblacionEstatal = pow(PoblacionEstatal, 2);
  if(PoblacionEstatal > 1)
       PoblacionEstatal += 10 * (PoblacionEstatal - 1);
  return(PoblacionEstatal);
}
```

## Anexo : Función Compacidad(double Area,double Perimetro)

```
double Compacidad(double Area,double Perimetro)
//CALCULA LA COMPACIDAD
{
  double Costo;
  Costo = ((Perimetro / sqrt(Area)) * 0.25 - 1.0) * 0.5;
  return(Costo);
}
```

## Anexo : Función SiguienteAleatorioReal0y1(long *semilla)

```
double SiguienteAleatorioReal0y1(long * semilla)
//DEVUELVE UN ALEATORIO ENTRE 0 Y 1
{
  long double zi, mhi31 = 2147483648u, ahi31 = 314159269u, chi31 = 453806245u;
  long int dhi31;
  zi = *semilla;
  zi = (ahi31 * zi) + chi31;
  if(zi > mhi31)
  {
    dhi31 = (long int) (zi / mhi31);
    zi = zi - (dhi31 * mhi31);
  }
  *semilla = (int) zi;
  zi = zi / mhi31;
  return (zi);
}
```

## Anexo : Función SiguienteAleatorioEnteroModN(long * semilla, int n)

```
int SiguienteAleatorioEnteroModN(long * semilla, int n)
// DEVUELVE UN ENTERO ENTRE 0 Y n-1
{
  double a;
  int v;
  long double zi, mhi31 = 2147483648u, ahi31 = 314159269u, chi31 = 453806245u;
  long int dhi31;
  zi = *semilla;
  zi = (ahi31 * zi) + chi31;
  if(zi > mhi31)
  {
    dhi31 = (long int) (zi / mhi31);
    zi = zi - (dhi31 * mhi31);
  }
  *semilla = (long int)zi;
  zi = zi / mhi31;
  a = zi;
  v = (int)(a * n);
  if (v == n)
    return(v-1);
  return(v);
}
```

# ANEXO C
# CÓDIGO DEL ALGORITMO ABC-RS

### Anexo : Función main()

```
int main()
{
  double z,seed1;
  double b1,u5,u6;
  int i,j, m6,k;
  int Semillas[1000], Numero_de_Semillas;
  long c;
  double MejorDesviacionPoblacional[45],MejorCompacidad[45];
  double Menor_DesviacionPoblacional, Menor_Compacidad;
  double Calidad_FuenteAlimento[200];
  int Corrida;
  int MejoresDistritos[6500];
  int GeneracionesSinMejora[200];
  int Solucion_Hibrida[6500], Mejora_Hibrida;
  double DesviacionPoblacional_Hibrida[45], Compacidad_Hibrida[45];
  char dummy[1000];
  int Precalentado;
  double Aceptacion_Promedio;
  FILE *fp;

    double Temperatura, TemperaturaFinal, TemperaturaInicial;
    int EquilibrioFinal;
    double alfa;
    double FactorEnfriamiento_Caliente, FactorEnfriamiento_Templado, FactorEnfriamiento_Frio;
    int Iteraciones_Caliente, Iteraciones_Templado, Iteraciones_Frio;

  //SE LEEN LOS PARAMETROS SOLICITADOS POR EL USUARIO
  char *direccion = "ABC_RS\\Parametros_ABC_RS.txt";
  fp = fopen(direccion, "r");
  while((c=fgetc(fp))!=EOF)
```

```
31  {
32    fscanf(fp,"%f %f %d %f %d %d",&TemperaturaInicial, &TemperaturaFinal, &Iteraciones_Caliente, &FactorEnfriamiento_Caliente, &
          Semilla, &Fuentes_de_Alimento);
33  }
34  fclose(fp);
35
36  FactorEnfriamiento_Templado = 0.90;
37  FactorEnfriamiento_Frio = 0.95;
38
39    Iteraciones_Templado = 2 * Iteraciones_Caliente;
40  Iteraciones_Frio = 3 * Iteraciones_Caliente;
41
42  if(FactorEnfriamiento_Templado < FactorEnfriamiento_Caliente)
43        FactorEnfriamiento_Templado = FactorEnfriamiento_Caliente;
44
45  if(FactorEnfriamiento_Frio < FactorEnfriamiento_Caliente)
46        FactorEnfriamiento_Frio = FactorEnfriamiento_Caliente;
47
48  //SI EL USUARIO SELECCINO EL SEMILLERO SE LEEN LAS SEMILLAS QUE SE UTILIZARAN
49  if(Semilla == -1)
50  {
51    Numero_de_Semillas = 0;
52    direccion = "Sistema_de_visualizacion\\Insumos\\Semillero.txt";
53    fp = fopen(direccion, "r");
54    while((c=fgetc(fp))!=EOF && Numero_de_Semillas < 1000)
55    {
56      fscanf(fp,"%d",&i);
57      Semillas[Numero_de_Semillas] = i;
58      Numero_de_Semillas++;
59    }
60    fclose(fp);
61  }
62
63  //EN CASO CONTRARIO SOLO SE REALIZARA UNA CORRIDA
64  else
65  {
66    Numero_de_Semillas = 1;
67    Semillas[0] = Semilla;
68  }
69
70  if(Numero_de_Semillas == 1)
71  {
72    printf("\n\n\n\t    Sistema para generar Zonas Electorales 2016\n\n\n");
73    printf("  Se realiza una corrida empleando los siguientes parametros:\n\n");
74    printf("  Temperatura inicial = %f\n\n",TemperaturaInicial);
75    printf("  Temperatura final = %f\n\n",TemperaturaFinal);
76    printf("  Numero de iteraciones = %d\n\n",Iteraciones_Caliente);
77    printf("  Factor de enfriamiento = %f\n\n",FactorEnfriamiento_Caliente);
78    printf("  Numero de individuos = %d\n\n",Fuentes_de_Alimento);
79    printf("  Semilla = %d\n\n",Semilla);
80  }
81
82  else
83  {
84    printf("\n\n\n\t    Sistema para generar Zonas Electorales 2016\n\n\n");
85    printf("  Se realizan %d corridas, empleando los siguientes parametros:\n\n", Numero_de_Semillas);
86    printf("  Temperatura inicial = %f\n\n",TemperaturaInicial);
87    printf("  Temperatura final = %f\n\n",TemperaturaFinal);
88    printf("  Numero de iteraciones = %d\n\n",Iteraciones_Caliente);
89    printf("  Factor de enfriamiento = %f\n\n",FactorEnfriamiento_Caliente);
90    printf("  Numero de individuos = %d\n\n",Fuentes_de_Alimento);
91    printf("  Semilla = Se usan los valores incluidos en el semillero\n\n");
92  }
93
94
95  // SE LEE EL NUMERO DE DISTRITOS ASIGNADOS A CADA CONJUNTO
96  ConjuntosTotales = 0;
97  NConjuntos = 0;
98  direccion = "Sistema_de_visualizacion\\Insumos\\ConjuntosDistritos.txt";
99  fp = fopen(direccion, "r");
100 while((c=fgetc(fp))!=EOF)
101 {
102   fscanf(fp,"%d %d",&k,&i);
103   DistritosPorConjunto[i] = k;
104   NConjuntos++;
105   ConjuntosTotales += k;
106 }
107 fclose(fp);
108
109
110 for(i=0; i<6500; i++)
111       Solucion_Hibrida[i] = 6500;
112
113 //SE REALIZAN TANTAS CORRIDAS COMO NUMERO DE SEMILLAS SE ENCUENTREN EN EL SEMILLERO
114 //O SOLO UNA SI EL USUARIO DA LA SEMILLA
115 for(Corrida = 0 ; Corrida < Numero_de_Semillas; Corrida++)
116 {
117   //SE CREA UN ARCHIVO PARA GUARDAR LOS COSTOS DE CADA CONJUNTO TERRITORIAL
118     if(Corrida == 0)
```

```
119     {
120       sprintf(dummy, "Sistema_de_visualizacion\\Resumen_Costos\\ABC_RS_Costo_Por_Conjunto.csv");
121             fp = fopen(dummy, "w");
122             fprintf(fp," , Costo Total, Equilibrio Poblacional,Compacidad, Tiempo(min)\n");
123             fclose(fp);
124       }
125
126       Mejora_Hibrida = 0;
127
128       if(Numero_de_Semillas > 1)
129             printf("\n\n\t Inicia la corrida = %d con la Semilla = %d\n\n",Corrida+1,Semillas[Corrida]);
130
131     Semilla = Semillas[Corrida];
132
133     for(i=0; i< 6500; i++)
134     DistritosFinales[i] = 6500;
135
136     //SE INICIA EL TIEMPO DE EJECUCION
137     clock_t start, end;
138     start = clock();
139
140     DistritosAcumulados = 0;
141
142     //SE INICIA EL PROCESO DE CONSTRUCCION DE DISTRITOS PARA CADA CONJUNTO
143     for(ConjuntoActual = 1; ConjuntoActual <= NConjuntos; ConjuntoActual++)
144     {
145       printf("\nConjunto territorial %d. ",ConjuntoActual);
146       for(i = 0; i < 6500; i++)
147       Conversion[i] = 6500;
148
149       //LA FUNCION Datos() LEE LA INFORMACION NECESARIA PARA CONSTRUIR LOS DISTRITOS
150       //POR EJEMPLO, COLINDANCIAS, AREA, POBLACION, ETC.
151       Datos(ConjuntoActual);
152       NDistritos = DistritosPorConjunto[ConjuntoActual];
153
154       //SE CONSTRUYEN NUEVAS FUENTES DE ALIMENTO Y SE EVALUA SU COSTO (PARA EL CONJUNTO EN CURSO)
155       //SE GUARDA UNA COPIA DE LAS FUENTES DE ALIMENTO NUEVAS EN Soluciones_Iniciales
156
157       if(DistritosPorConjunto[ConjuntoActual] == 1)
158       {
159         FuenteAlimento_Nueva(NDistritos);
160         for(i = 0; i < UnidadesPorConjunto; i++)
161                   MejoresDistritos[i] = Distrito[i];
162         goto Final;
163       }
164
165       for(j = 0; j < Fuentes_de_Alimento; j++)
166       {
167         FuenteAlimento_Nueva(NDistritos);
168         for(i = 0; i < UnidadesPorConjunto; i++)
169                   Soluciones_Iniciales[j][i] = RS_Soluciones[j][i] = FuenteAlimento[j][i] = Distrito[i];
170         Costo_FuenteNueva(j);
171                 Costo_Iniciales[j] = RS_Costos[j] = Costo_FuenteAlimento[j];
172                 RS_DesviacionPoblacional[j] = DesviacionPoblacional_FuenteAlimento[j];
173                 RS_Compacidad[j] = Compacidad_FuenteAlimento[j];
174       }
175
176       Menor_DesviacionPoblacional = DesviacionPoblacional_FuenteAlimento[0];
177       Menor_Compacidad = Compacidad_FuenteAlimento[0];
178
179       //INICIA PROCESO DE MEJORA
180
181       Temperatura = TemperaturaInicial;
182       Precalentado = 1;
183       EquilibrioFinal = Iteraciones_Caliente;
184       alfa = FactorEnfriamiento_Caliente;
185       Pronostico_Vecindario[1] = Pronostico_Vecindario[2] = 1.00;
186       Pronostico_Vecindario[3] = 1.00;
187       VecindarioRS = 1;
188
189
190       while(Temperatura > TemperaturaFinal)
191       {
192         //PARA CADA FUENTE DE ALIMENTO SE LLAMA UNA VEZ A LA FUNCION Recocido_Simulado
193
194         u5 = 1;
195         Aceptacion_Promedio = 0.00;
196         for(j = 0; j < Fuentes_de_Alimento; j++)
197                 {
198                     for(i = 0; i < UnidadesPorConjunto; i++)
199                     {
200                         Distrito[i] = RS_Soluciones[j][i];
201                     }
202
203                     u6 = Recocido_Simulado(Temperatura, EquilibrioFinal,j);
204                     Aceptacion_Promedio += u6/Fuentes_de_Alimento;
205
206                     for(i = 0; i < UnidadesPorConjunto; i++)
207                     {
```

```
                    RS_Soluciones[j][i] = Distrito[i];
                }


                b1 = Menor_DesviacionPoblacional + Menor_Compacidad;
    u5 = Costo_FuenteAlimento[j];

    //SI LA NUEVA FUENTE DE ALIMENTO ES LA MEJOR CONOCIDA SE GUARDA EN MEMORIA

    if(u5 < b1)
    {
      Menor_DesviacionPoblacional = DesviacionPoblacional_FuenteAlimento[j];
      Menor_Compacidad = Compacidad_FuenteAlimento[j];
      for(k = 0; k < UnidadesPorConjunto; k++)
        MejoresDistritos[k] = FuenteAlimento[j][k];
    }

                //SI EL NIVEL DE ACEPTACION DE LA FUNCION Recocido_Simulado ES MAYOR A 0.20 SE
                // REALIZAN EMJORAS EN Soluciones_Iniciales

                if(0.20 <= u6 )
                {
                    for(i = 0; i < UnidadesPorConjunto; i++)
                    {
                        Distrito[i] = Soluciones_Iniciales[j][i];
                    }

                    Recocido_Simulado2(Temperatura, j, 50);
                }

                if(u5 < b1)
    {
      Menor_DesviacionPoblacional = DesviacionPoblacional_FuenteAlimento[j];
      Menor_Compacidad = Compacidad_FuenteAlimento[j];
      for(k = 0; k < UnidadesPorConjunto; k++)
        MejoresDistritos[k] = FuenteAlimento[j][k];
    }
  }

            //DEPENDIENDO DEL NIVEL DE ACEPTACION DE LA FUNCION Recocido_Simulado
            //SE EMPLEAN DIFERENTES ESQUEMAS DE ENFRIAMIENTO

            if(0.40 <= Aceptacion_Promedio && Precalentado == 0)
            {
                EquilibrioFinal = Iteraciones_Caliente;
                alfa = FactorEnfriamiento_Caliente;
            }

            if(0.20 <= Aceptacion_Promedio && Aceptacion_Promedio < 0.40 && Precalentado == 0)
            {
                EquilibrioFinal = Iteraciones_Templado;
                alfa = FactorEnfriamiento_Templado;
            }

            if(Aceptacion_Promedio < 0.20 && Precalentado == 0)
            {
                EquilibrioFinal = Iteraciones_Frio;
                alfa = FactorEnfriamiento_Frio;
            }

            if(Aceptacion_Promedio < 0.5 && Precalentado == 1)
                Temperatura = Temperatura * 1.1;

            if(Aceptacion_Promedio >= 0.5)
                Precalentado = 0;

            if(Aceptacion_Promedio < 0.001)
                Temperatura = TemperaturaFinal;

            if(Precalentado == 0)
                Temperatura = Temperatura * alfa;
}

//TERMINA EL PROCESO DE MEJORA DE UN CONJUNTO TERRITORIAL

      //SE REALIZA UNA BUSQUEDA LOCAL EN CADA FUENTE DE ALIMENTO

      for(j = 0; j < Fuentes_de_Alimento; j++)
      {
          for(i = 0; i < UnidadesPorConjunto; i++)
          {
              Distrito[i] = FuenteAlimento[j][i];
          }

          if(Busqueda_Local() == 1)
          {
              Evalua_Solucion();
              b1 = Menor_DesviacionPoblacional + Menor_Compacidad;
              u5 = DesviacionPoblacional_Nueva + Compacidad_Nueva;
```

```
297                     if(u5 < b1)
298                     {
299                         Menor_DesviacionPoblacional = DesviacionPoblacional_Nueva;
300                         Menor_Compacidad = Compacidad_Nueva;
301                         for(k = 0; k < UnidadesPorConjunto; k++)
302                             MejoresDistritos[k] = Distrito[k];
303                     }
304                 }
305             }
306
307             //SE REALIZA UNA BUSQUEDA LOCAL EN LA MEJOR SOLUCION ENCONTRADA
308
309         j = -1;
310             for(i = 0; i < UnidadesPorConjunto; i++)
311                 Distrito[i] = MejoresDistritos[i];
312
313         j = Busqueda_Local();
314
315         //EN CASO DE MEJORA SE ACTUALIZA LA INFORMACION
316         if(j == 1)
317             {
318                 for(i=0; i<UnidadesPorConjunto; i++)
319                 {
320                     MejoresDistritos[i] = Distrito[i];
321                 }
322             }
323
324         //SE GUARDAN LOS MEJORES DISTRITOS CONSTRUIDOS EN LA VARIABLE DistritosFinales
325         //AL TEMINAR CADA CORRIDA EL ARREGLO DistritosFinales TENDRA EL ESCENARIO COMPLETO
326 Final:
327
328         for(i=0; i<UnidadesPorConjunto; i++)
329         {
330           for(j=0; j<6500; j++)
331           {
332             if(Conversion[j] == i)
333                         DistritosFinales[j] = MejoresDistritos[i] + DistritosAcumulados;
334           }
335         }
336
337         for(i=0; i< UnidadesPorConjunto; i++)
338                 Distrito[i] = MejoresDistritos[i];
339         Evalua_Solucion();
340
341         for(i = 0; i < NDistritos; i++)
342         {
343           MejorDesviacionPoblacional[i + DistritosAcumulados] = DesviacionPoblacionalDistrito[i];
344           MejorCompacidad[i + DistritosAcumulados] = CompacidadDistrito[i];
345         }
346
347         printf(" Costo final %f = %f  +  0.5 * %f\n",DesviacionPoblacional_Nueva + Compacidad_Nueva,DesviacionPoblacional_Nueva, 2 *
                 Compacidad_Nueva);
348
349         //SE IMPRIME EL COSTO DE CADA CONJUNTO TERRITORIAL
350
351             sprintf(dummy, "Sistema_de_visualizacion\\Resumen_Costos\\ABC_RS_Costo_Por_Conjunto.csv");
352             fp = fopen(dummy, "a");
353             fprintf(fp,"Conjunto %d,%f,%f,%f\n", ConjuntoActual, DesviacionPoblacional_Nueva + Compacidad_Nueva,
                        DesviacionPoblacional_Nueva, Compacidad_Nueva);
354             fclose(fp);
355
356         //SE CREA O SE ACTUALIZA EL Escenario_Hibrido
357         if(Corrida == 0 && Numero_de_Semillas > 1)
358         {
359             Mejora_Hibrida = 1;
360           for(i=0; i<UnidadesPorConjunto; i++)
361           {
362             for(j=0; j<6500; j++)
363             {
364               if(Conversion[j] == i)
365                         Solucion_Hibrida[j] = MejoresDistritos[i] + DistritosAcumulados;
366             }
367           }
368           for(i = 0; i < NDistritos; i++)
369           {
370             DesviacionPoblacional_Hibrida[i + DistritosAcumulados] = MejorDesviacionPoblacional[i + DistritosAcumulados];
371             Compacidad_Hibrida[i + DistritosAcumulados] = MejorCompacidad[i + DistritosAcumulados];
372           }
373         }
374
375         if(Corrida >= 1)
376         {
377           u5 = u6 = 0;
378                 for(i = 0; i < NDistritos; i++)
379           {
380             u5 += DesviacionPoblacional_Hibrida[i + DistritosAcumulados] + Compacidad_Hibrida[i + DistritosAcumulados];
381             u6 += MejorDesviacionPoblacional[i + DistritosAcumulados] + MejorCompacidad[i + DistritosAcumulados];
382           }
383           if(u5 > u6)
```

```
384         {
385           Mejora_Hibrida = 1;
386           for(i=0; i<UnidadesPorConjunto; i++)
387           {
388             for(j=0; j<6500; j++)
389             {
390               if(Conversion[j] == i)
391                                 Solucion_Hibrida[j] = MejoresDistritos[i] + DistritosAcumulados;
392             }
393           }
394           for(i = 0; i < NDistritos; i++)
395           {
396             DesviacionPoblacional_Hibrida[i + DistritosAcumulados] = MejorDesviacionPoblacional[i + DistritosAcumulados];
397             Compacidad_Hibrida[i + DistritosAcumulados] = MejorCompacidad[i + DistritosAcumulados];
398           }
399         }
400       }
401
402       DistritosAcumulados += NDistritos;
403     }
404     //TERMINA EL PROCESO DE MEJORA DEL ESCENARIO COMPLETO
405
406     end = clock();
407     seed1 = end - start;
408     z = seed1 / CLOCKS_PER_SEC;
409     seed1 = seed1 / (60 * CLOCKS_PER_SEC);
410     i = (int) seed1;
411     seed1 = z - (60 * i);
412     z = i / 60;
413     printf("\n\nEL TIEMPO DE EJECUCION FUE DE: %d MINUTOS %f SEGUNDOS\n\n",i,seed1);
414         seed1 = i + (seed1 / 60);
415
416     for(i=0; i< 6500; i++)
417             MejoresDistritos[i] = DistritosFinales[i];
418
419     //SE OBTIENE EL COSTO TOTAL DE LA SOLUCION CONSTRUIDA
420     Costo_Nueva = DesviacionPoblacional_Nueva = Compacidad_Nueva = 0;
421     for(i=0; i<ConjuntosTotales; i++)
422     {
423       Costo_Nueva += MejorDesviacionPoblacional[i] + MejorCompacidad[i];
424       DesviacionPoblacional_Nueva += MejorDesviacionPoblacional[i];
425       Compacidad_Nueva += MejorCompacidad[i];
426     }
427
428     //SE IMPRIME EL COSTO TOTAL DEL ESCENARIO ACTUAL
429         sprintf(dummy, "Sistema_de_visualizacion\\Resumen_Costos\\ABC_RS_Costo_Por_Conjunto.csv");
430         fp = fopen(dummy, "a");
431         fprintf(fp,"Costo Total,%f,%f,%f,%f\n\n",Costo_Nueva, DesviacionPoblacional_Nueva, Compacidad_Nueva,seed1);
432         fclose(fp);
433
434     //SE IMPRIME EN ARCHIVO DE TEXTO LA SOLUCION FINAL
435     sprintf(dummy, "Sistema_de_visualizacion\\ABC_RS_Escenario_%f_%d.csv", DesviacionPoblacional_Nueva + Compacidad_Nueva, Semillas[
            Corrida]);
436     fp = fopen(dummy, "w");
437     fprintf(fp,"Seccion,DISTRITO\n");
438     for(i = 0; i < 6500; i++)
439         {
440             if(MejoresDistritos[i] < 6500)
441                 fprintf(fp,"%d,%d\n",i,MejoresDistritos[i] + 1);
442         }
443     fclose(fp);
444
445         //SE IMPRIME LA SOLUCION HIBRIDA CONSTRUIDA CON LOS ESCENARIOS OBTENIDOS
446         //SI SE OBTUVO ALGUNA MEJORA Y SE EMPLEO EL SEMILLERO
447
448         if(Numero_de_Semillas > 1 && Mejora_Hibrida == 1)
449         {
450             Mejora_Hibrida = 0;
451             direccion = "Sistema_de_visualizacion\\Escenario_Hibrido_ABC_RS.csv";
452             fp = fopen(direccion, "w");
453             fprintf(fp,"Seccion,DISTRITO\n");
454             for(i = 0; i < 6500; i++)
455                 if(Solucion_Hibrida[i] < 6500)
456                     fprintf(fp,"%d,%d\n",i,Solucion_Hibrida[i] + 1);
457             fclose(fp);
458         }
459   }
460   //TERMINA LA CORRIDA ACTUAL
461
462   //SE IMPRIME LA SOLUCION HIBRIDA CONSTRUIDA CON LOS ESCENARIOS OBTENIDOS
463   //SOLO SI SE EMPLEO EL SEMILLERO
464
465   if(Numero_de_Semillas > 1)
466   {
467     direccion = "Sistema_de_visualizacion\\Escenario_Hibrido_ABC_RS.csv";
468     fp = fopen(direccion, "w");
469     fprintf(fp,"Seccion,DISTRITO\n");
470     for(i = 0; i < 6500; i++)
471     if(Solucion_Hibrida[i] < 6500)
```

```
472             fprintf(fp,"%d,%d\n",i,Solucion_Hibrida[i] + 1);
473       fclose(fp);
474     }
475
476     printf("\t Se han completado las corridas solicitadas. \n\n \t Presione la tecla Enter para conluir el proceso.");
477     getchar();
478
479     return(1);
480   }
```

## Anexo : Función Datos(int Conjunto)

```
1   void Datos(int Conjunto)
2   //LEE LOS ARCHIVOS DE TEXTO Separar.txt, DatosConglomerados.txt y Colindancias.txt PARA OBTENER INFORMACION SOBRE
3   //LAS UNIDADES GEOGRAFICAS QUE EMPLEARA EN CADA CONJUNTO TERRITORIAL
4   {
5     int i,k,l,m,c,mun,o,j;
6     double p;
7     int U, f, Conglomerado[10500];
8     int ConjuntoTerritorial[6500];
9     int Separar[100][2], SeccionMun[6500], Aux, Separadas;
10
11    FILE *fp;
12    char *direccion;
13    UnidadesPorConjunto = 0;
14    MediaEstatal = 0;
15
16    direccion = "Sistema_de_visualizacion\\Insumos\\Separar.txt";
17    Separadas = 0;
18    fp = fopen(direccion, "r");
19    while((c=fgetc(fp))!=EOF)
20    {
21      fscanf(fp,"%d %d",&i,&k);
22      Separar[Separadas][0] = i;
23      Separar[Separadas][1] = k;
24      Separadas++;
25      Separar[Separadas][0] = k;
26      Separar[Separadas][1] = i;
27      Separadas++;
28    }
29    fclose(fp);
30
31    for(i = 0; i < 10500; i++)
32      Conglomerado[i] = -1;
33
34    m = 0;
35    direccion = "Sistema_de_visualizacion\\Insumos\\DatosConglomerados.txt";
36    fp = fopen(direccion, "r");
37    while((c=fgetc(fp))!=EOF)
38    {
39      fscanf(fp,"%d %d %lf %d %d %d",&mun,&i,&p,&j,&o,&k);
40      MediaEstatal += j;
41      ConjuntoTerritorial[i] = k;
42      f = 0;
43      SeccionMun[i] = mun;
44      if(k == Conjunto)
45      {
46        if(Conglomerado[o] != -1)
47              {
48                  Conversion[i] = Conglomerado[o];
49                  AreaUnidadGeografica[Conglomerado[o]] += p;
50                  PoblacionUnidadGeografica[Conglomerado[o]] += j;
51              }
52
53        if(Conglomerado[o] == -1)
54              {
55                  Conversion[i] = m;
56                  AreaUnidadGeografica[m] = p;
57                  PoblacionUnidadGeografica[m] = j;
58                  UnidadesPorConjunto++;
59                  Conglomerado[o] = m;
60                  m++;
61              }
62      }
63    }
64    fclose(fp);
65
66    MediaEstatal = MediaEstatal / ConjuntosTotales;
67
68    for(i=0; i<6500; i++)
69    {
```

```
70      for(j=0; j<6500; j++)
71      {
72        PerimetroFrontera[i][j] = 0;
73      }
74    }
75
76
77    for(i=0; i<6500; i++)
78    {
79      for(j=0; j<60; j++)
80      {
81              Vecinos[i][j] = 6500;
82          }
83      Vecinos[i][60] = 0;
84    }
85
86    direccion = "Sistema_de_visualizacion\\Insumos\\ColindanciasUnidades.txt";
87    fp = fopen(direccion, "r");
88    while((c=fgetc(fp))!=EOF)
89    {
90      fscanf(fp,"%d %d %lf",&i,&j,&p);
91      if(ConjuntoTerritorial[i] == Conjunto)
92      {
93        Aux = 0;
94        if(j != 0)
95              {
96                  for(k=0; k<Separadas;k++)
97                  {
98                      if(SeccionMun[i] == Separar[k][0] && SeccionMun[j] == Separar[k][1])
99                      {
100                         m = j;
101                         Aux = 1;
102                         j = 0;
103                         break;
104                     }
105                 }
106             }
107       if(j != 0)
108       {
109         if(ConjuntoTerritorial[j] == Conjunto)
110         {
111           l = Conversion[i];
112           m = Conversion[j];
113           if(l != m)
114           {
115             PerimetroFrontera[l][m] += p;
116
117             f = 0;
118             for(U = 0; U < Vecinos[l][60]; U++)
119             {
120               if(Vecinos[l][U] == m)
121               {
122                 f = 1;
123                 break;
124               }
125             }
126             if(f == 0)
127             {
128               Vecinos[l][Vecinos[l][60]] = m;
129               Vecinos[l][60]++;
130             }
131           }
132         }
133
134         else
135         {
136           l = Conversion[i];
137           PerimetroFrontera[l][l] += p;
138         }
139         if(Aux == 1)
140                     j = m;
141       }
142       if(j == 0)
143       {
144         l = Conversion[i];
145         PerimetroFrontera[l][l] += p;
146       }
147     }
148   }
149   fclose(fp);
150 }
```

## Anexo : Función FuenteAlimento_Nueva(int DistritosPorConjunto)

```
void FuenteAlimento_Nueva(int DistritosPorConjunto)
//CONSTRUYE UNA SOLUCION NUEVA CON EL NUMERO DE DISTRITOS INDICADOS PARA EL CONJUNTO TERRITORIAL ACTUAL
//TODOS LOS DISTRITOS SON CONEXOS Y TODAS LAS UNIDADES GEOGRAFICAS PERTENCEN EXACTAMENTE A UN DISTRITO
{
  int i, j, k,l,pp,u[45],contador[6500],seed, z;
  int Unidades[6500],Distrito_Auxiliar[6500];
  for(i = 0; i < UnidadesPorConjunto; i++)
  {
    Distrito[i] = -1;
    Distrito_Auxiliar[i] = -1;
  }

  //SE OBTIENEN UnidadesPorConjunto UNIDADES GEOGRAFICAS PARA EL CONJUNTO TERRITORIAL ACTUAL
  for(i = 0; i < UnidadesPorConjunto; i++)
  {
    Unidades[i] = i;
        contador[i]=0;
  }
  j = UnidadesPorConjunto;
  for(k = 0; k < DistritosPorConjunto; k++)
  {
    i = SiguienteAleatorioEnteroModN(& Semilla, j);
    u[k] = Unidades[i];
    contador[u[k]] = 1;

    //SE INICIALIZAN LOS DISTRITOS CON LAS UNIDADES SELECCIONADAS
    Distrito[Unidades[i]] = k;
    j--;
    for(l = i; l < j; l++)
    {
      Unidades[l] = Unidades[l + 1];
    }
  }

  //EMPIEZA CONSTRUCCION DE SOLUCION INICIAL
  j = 0;
  while(j != UnidadesPorConjunto)
  {
      k = SiguienteAleatorioEnteroModN(& Semilla,DistritosPorConjunto);

    //ENCUENTRA LAS COLINDANCIAS DEL DISTRITO k
    for(i=0; i<UnidadesPorConjunto; i++)
    {
      if(Distrito[i] == k)
      {
        for(j=0; j<Vecinos[i][60]; j++)
        {
          if(contador[Vecinos[i][j]] == 0)
                        Distrito_Auxiliar[Vecinos[i][j]] = k;
        }
      }
    }

    pp = 0;
    for(j=0; j<UnidadesPorConjunto; j++)
    {
      //CUENTA A TODOS LOS VECINOS QUE SE PUEDEN AGREGAR AL DISTRITO k
      if(Distrito_Auxiliar[j] == k)
                pp++;
    }
    if(pp == 1)
              seed = 0;
    if(pp > 1)
              seed = SiguienteAleatorioEnteroModN(& Semilla,pp);
    if(pp > 0)
    {
      z = 0;
      //SELECCIONA A UN VECINO DEL DISTRITO k Y LO AGREGA
      for(j=0; j<UnidadesPorConjunto; j++)
      {
        if(Distrito_Auxiliar[j] == k)
        {
          if( z == seed)
          {
            contador[j] = 1;
            Distrito[j] = k;
            break;
          }
          z++;
        }
      }
    }
    j = 0;
    for(i=0; i<UnidadesPorConjunto; i++)
```

```
85      {
86        Distrito_Auxiliar[i] = -1;
87        j += contador[i];
88          }
89    }
90  }
```

## Anexo : Función Costo_FuenteNueva(int AB)

```
1   void Costo_FuenteNueva(int AB)
2   //CALCULA EL COSTO DE UNA FUENTE DE ALIMENTO CREADA DE FORMA ALEATORIA
3   {
4
5     int i, j, k,p;
6
7     for(i=0;i<NDistritos;i++)
8     {
9       PerimetroDistrito[i] = 0;
10      PoblacionDistrito[i] = 0;
11      AreaDistrito[i] = 0;
12    }
13
14
15    for(j=0;j<UnidadesPorConjunto;j++)
16    {
17      PoblacionDistrito[Distrito[j]] += PoblacionUnidadGeografica[j];
18      AreaDistrito[Distrito[j]] += AreaUnidadGeografica[j];
19      PerimetroDistrito[Distrito[j]] += PerimetroFrontera[j][j];
20      for(k = 0;k < Vecinos[j][60];k++)
21      {
22        p = Vecinos[j][k];
23        if(Distrito[p] != Distrito[j] )
24                  PerimetroDistrito[Distrito[j]] += PerimetroFrontera[j][p];
25      }
26
27    }
28
29    DesviacionPoblacional_FuenteAlimento[AB] = Compacidad_FuenteAlimento[AB] = 0;
30    for(i=0;i<NDistritos;i++)
31    {
32      DesviacionPoblacionalDistrito[i] = Desviacion_Poblacional(PoblacionDistrito[i]);
33      CompacidadDistrito[i] = Compacidad(AreaDistrito[i],PerimetroDistrito[i]);
34      DesviacionPoblacional_FuenteAlimento[AB] += DesviacionPoblacionalDistrito[i];
35      Compacidad_FuenteAlimento[AB] += CompacidadDistrito[i];
36    }
37
38    Costo_FuenteAlimento[AB] = DesviacionPoblacional_FuenteAlimento[AB] + Compacidad_FuenteAlimento[AB];
39
40  }
```

## Anexo : Función RevisaConexidad_Empleada(int Origen, int Destino)

```
1   int RevisaConexidad_Empleada(int Origen, int Destino)
2   //CON ESTA FUNCION SE REVISA SI EL DISTRITO Origen PERDIO LA CONEXIDAD
3   //DE SER ASI SE EMPLEA LA ESTRATEGIA DE REPARACION ReparaConexidad_Empleada
4   {
5     int j,i,m,suma;
6     int Representante[6500], Componente[6500],N = 0,n;
7     int Lista1[6500], Lista2[6500];
8     int Contactado[6500];
9
10    suma = 0;
11
12      //SE REVISA EL NUMERO DE COMPONENTES CONEXAS EN EL DISTRITO
13
14    for(m=0; m<UnidadesPorConjunto; m++)
15    {
16      Contactado[m] = 0;
17      Representante[m] = -1;
18      Componente[m] = 0;
19      if(Distrito[m] == Origen)
20      {
21        Lista1[suma] = m;
22        suma++;
```

```
    }
  }

    //SI EL NUMERO DE COMPONENTES ES MENOR O IGUAL A 1 NO SE HA PERDIDO LA CONEXIDAD
  if( suma == 0)
  {
    printf("\n ERROR Distrito vacio\n");
    getchar();
    return(0);
  }

  if( suma == 1)
        return(0);

    //SI EL NUMERO DE COMPONENTES CONEXAS ES MAYOR O IGUAL A 2 SE DEBE REPARAR LA CONEXIDAD

  for(i=0; i< suma; i++)
  {
    n = 0;
    if(Contactado[Lista1[i]] == 0)
    {
      Representante[N] = Lista1[i];
      Lista2[n] = Lista1[i];
      Contactado[Lista2[n]] = 1;
      n++;
      for(j=0; j < n; j++)
      {
        for(m=0; m < Vecinos[Lista2[j]][60]; m++)
        {
          if(Distrito[Vecinos[Lista2[j]][m]] == Origen && Contactado[Vecinos[Lista2[j]][m]] == 0)
          {
            Lista2[n] = Vecinos[Lista2[j]][m];
            Contactado[Vecinos[Lista2[j]][m]] = 1;
            n++;
          }
        }
      }
      Componente[N] = n;
      N++;
    }
  }

  if( N == 1)
        return(0);

  n = j = 0;
  j = Componente[0];

    //SE DETERMINA CUAL ES LA COMPONENTE CON MAYOR NUMERO DE UNIDADES GEOGRAFICAS
  //PARA DEJARLA COMO EL DISTRITO Origen
  for(i = 1; i < N; i++)
  {
    if(Componente[i] > j)
    {
      j = Componente[i];
      n = i;
    }
  }

    //EL RESTO DE LAS COMPONENTES SON ENVIADAS AL DISTRITO Destino
  for(i = 0; i < N; i++)
  {
    if(i != n)
          ReparaConexidad_Empleada(Origen,Representante[i], Destino);
  }

  return(0);
}
```

## Anexo : Función ReparaConexidad_Empleada(int Origen, int Unidad, int Destino)

```
int ReparaConexidad_Empleada(int Origen, int Unidad, int Destino)
//ESTA FUNCION ENVIA A LAS UNIDADES QUE FORMAN UNA COMPONENTE CONEXA JUNTO CON Unidad DE
//DISTRITO Origen A DISTRITO Destino
{
  int j,i,m,n;
  int Lista[6500];
  int ComponenteConexa[6500];
    int Destinos[6000], Destinos2[6500], nm;

  for(m=0; m<UnidadesPorConjunto; m++)
```

```
11          ComponenteConexa[m] = 0;
12
13    Lista[0] = Unidad;
14    ComponenteConexa[Lista[0]] = 1;
15    n = 1;
16
17      //CON EL SIGUIENTE CICLO SE ENCUENTRA A TODAS LAS UNIDADES GEOGRAFICAS EN LA MISMA COMPONENTE CONEXA QUE Unidad
18
19    for(j=0; j < n; j++)
20    {
21      for(m=0; m < Vecinos[Lista[j]][60]; m++)
22      {
23        if(Distrito[Vecinos[Lista[j]][m]] == Origen && ComponenteConexa[Vecinos[Lista[j]][m]] == 0)
24        {
25          Lista[n] = Vecinos[Lista[j]][m];
26          ComponenteConexa[Vecinos[Lista[j]][m]] = 1;
27          n++;
28        }
29      }
30    }
31
32      //CADA COMPONENTE ES ENVIADA A UN DISTRITO VECINO
33
34    while(n > 0)
35      {
36
37          for(j=0; j < n;j++)
38          {
39              for(m = 0;m < NDistritos;m++)
40              {
41                  Destinos[m] = 0;
42                  Destinos2[m] = -5;
43              }
44
45              mm = 0;
46              for(m=0; m < Vecinos[Lista[j]][60]; m++)
47              {
48                  if(Distrito[Vecinos[Lista[j]][m]] != Origen && Destinos[Distrito[Vecinos[Lista[j]][m]]] == 0)
49                  {
50                      Destinos[Distrito[Vecinos[Lista[j]][m]]] = 1;
51                      Destinos2[mm] = Distrito[Vecinos[Lista[j]][m]];
52                      mm++;
53                  }
54              }
55
56              //SE ELIGE UN DESTINO PARA LAS UNIDADES GEOGRAFICAS EN LA COMPONENTE CONEXA
57
58              if(mm > 1)
59              {
60                  mm = SiguienteAleatorioEnteroModN(& Semilla, mm);
61                  Distrito[Lista[j]] = Destinos2[mm];
62
63                  for(m = j; m < n-1; m++)
64                      Lista[m] = Lista[m+1];
65                  n--;
66                  j--;
67                  mm = 0;
68              }
69
70              if( mm == 1)
71              {
72                  Distrito[Lista[j]] = Destinos2[0];
73                  for(m = j; m < n-1; m++)
74                      Lista[m] = Lista[m+1];
75                  n--;
76                  j--;
77              }
78          }
79      }
80
81      //TODAS LAS UNIDADES GEOGRAFICAS EN LA COMPONENTE CONEXA DE Unidad SON ENVIADAS AL DISTRITO Destino
82
83    for(i = 0; i < UnidadesPorConjunto; i++)
84    {
85      if(ComponenteConexa[i] == 1)
86        Distrito[i] = Destino;
87    }
88
89    return(0);
90  }
```

## Anexo : Función AbejaObservadora(int AB)

```
int AbejaObservadora(int AB)
{
  int b, i, j, k, n, n1, n2, n3, m2, o, o2, o3,AB1;
  int Candidatos[6500], Candidatos1[6500], Candidatos2[6500], Candidatos3[6500];
  double b1, u5;
  int OrigenAB, OrigenAB1, Unidad_Elegida;
  int ii,kk;

  //SE ELIGE UN DISTRITO AL AZAR DE LA FUENTE DE ALIMENTO AB
  OrigenAB = SiguienteAleatorioEnteroModN(& Semilla, NDistritos);

  //SE HACE UNA LISTA CON LAS UG DE DISTRITO Origen
  n = 0;

  for(i = 0; i < UnidadesPorConjunto; i++)
  {
    Solucion[i] = Distrito[i];
    Candidatos[i] = 6500;
    if(Distrito[i] == OrigenAB)
    {
      Candidatos[n] = i;
      n++;
    }
  }
  if( n == 1)
        k = Candidatos[0];

  //SE ELIGEN UNA UNIDAD DE DISTRITO Origen CON PROBABILIDAD 1/n
  else
  {
    b = SiguienteAleatorioEnteroModN(& Semilla, n);
    k = Candidatos[b];
  }

    //SE ELIGE UNA FUENTE DE ALIMENTO, AB1, DIFERENTE DE AB
    AB1 = AB;
    while(AB1 == AB)
    {
        AB1 = SiguienteAleatorioEnteroModN(& Semilla, Fuentes_de_Alimento);
    }

    b1 = SiguienteAleatorioReal0y1(&Semilla);

    if( b1 < 0.5)
    {
        for(i=0; i<UnidadesPorConjunto;i++)
            FuenteAlimento[Fuentes_de_Alimento][i] = Soluciones_Iniciales[AB1][i];
        AB1 = Fuentes_de_Alimento;
    }

  OrigenAB1 = FuenteAlimento[AB1][k];

    if(n >= 2)
    {
        //SE LE QUITA UNA UNIDAD A LA FUENTE DE ALIMENTO OrigenAB
        //Candidatos1[] GUARDA LAS UNIDADES QUE ESTAN EN OrigenAB PERO NO ESTAN EN OrigenAB1

        n1 = 0;
        for(j = 0; j < UnidadesPorConjunto; j++)
        {
            if(Distrito[j] == OrigenAB && FuenteAlimento[AB1][j] != OrigenAB1)
            {
                Candidatos1[n1] = j;
                n1++;
            }
        }

        //Candidatos2[] GUARDA LAS UNIDADES DE Candidatos1[] QUE ESTAN EN LA FRONTERA DE OrigenAB
        n2 = 0;
        for(j = 0; j < n1; j++)
        {
            o = Candidatos1[j];
            for(o2 = 0; o2 < Vecinos[o][60]; o2++)
            {
                o3 = Vecinos[o][o2];
                if(Distrito[o3] != OrigenAB )
                {
                    Candidatos2[n2] = o;
                    n2++;
                    break;
                }
            }
        }

        if(n2 == 0)
            j = -1;
        if(n2 == 1)
            j = 0;
```

```
89          if(n2 > 1)
90          {
91              //SI NO HAY MUNICIPIOS CON UN ALTO PORCENTAJE DE POBLACION INDIGENA
92              //SE ELIGE CON UNA OPCION CON PROBABILIDAD 1/n
93              j = SiguienteAleatorioEnteroModN(& Semilla, n2);
94          }
95
96          if(j > -1)
97          {
98              kk = 0;
99              Unidad_Elegida = Candidatos2[j];
100             for(o2 = 0; o2 < Vecinos[Unidad_Elegida][60]; o2++)
101             {
102                 o3 = Vecinos[Unidad_Elegida][o2];
103                 if(Distrito[o3] != Distrito[Unidad_Elegida])
104                 {
105                     Candidatos1[kk] = Distrito[o3];
106                     kk++;
107                 }
108             }
109
110             //SE SELECCIONA UN DISTRITO PARA ENVIAR A LA UNIDAD GEOGRAFICA SELECCIONADA
111
112             if(kk == 1)
113                 Distrito[Unidad_Elegida] = Candidatos1[0];
114
115             if(kk > 1)
116             {
117                 //SI NO HAY MUNICIPIOS CON UN ALTO PORCENTAJE DE POBLACION INDIGENA
118                 //SE ELIGE CON UNA PROBABILIDAD 1/n
119                     kk = SiguienteAleatorioEnteroModN(& Semilla, kk);
120                     kk = Candidatos1[kk];
121                     Distrito[Unidad_Elegida] = kk;
122             }
123         }
124
125         //SE REVISA LA CONEXIDAD DE DISTRITO OrigenAB
126         //EN CASO DE PERDERSE, EL NUEVO DISTRITO SERA LA COMPONENTE CONEXA QUE CONTENGA A LA UNIDAD k
127         RevisaConexidad_Observadora2(OrigenAB,k);
128     }
129
130 //SE LE AGREGA UNA UNIDAD A LA FUENTE DE ALIMENTO OrigenAB
131 //PRIMERO SE BUSCAN LAS UNIDADES QUE NO ESTAN EN OrigenAB PERO SI ESTAN EN OrigenAB1
132
133 n1 = 0;
134
135 for(j = 0; j < UnidadesPorConjunto; j++)
136 {
137   if(Distrito[j] != OrigenAB && FuenteAlimento[AB1][j] == OrigenAB1)
138   {
139     Candidatos1[n1] = j;
140     n1++;
141   }
142 }
143
144   //EN Candidatos2[] SE INCLUYEN LAS UNIDADES DE Candidatos1[] QUE SON VECINOS DE OrigenAB
145
146 n2 = 0;
147
148 for(j = 0; j < n1; j++)
149 {
150   o = Candidatos1[j];
151
152   for(o2 = 0; o2 < Vecinos[o][60]; o2++)
153   {
154     o3 = Vecinos[o][o2];
155     if(Distrito[o3] == OrigenAB)
156     {
157       Candidatos2[n2] = o;
158       n2++;
159       break;
160     }
161   }
162 }
163
164 n3 = 0;
165
166 for(j = 0; j < n2; j++)
167 {
168   o = Candidatos2[j];
169   m2 = 0;
170
171   for(ii = 0; ii < UnidadesPorConjunto; ii++)
172   //SE REVISA CUALES UNIDADES DE Candidatos2[] PERTENECEN A UN DISTRITO CON MAS DE DOS UNIDADES GEOGRAFICAS
173   {
174     if(Distrito[ii] == Distrito[o])
175               m2++;
176     if(m2 > 2)
177               break;
```

```
178       }
179
180       //Candidatos3[] CONTIENE UNIDADES DE Candidatos2[] QUE PERTENECEN A UN DISTRITO CON MAS DE DOS UNIDADES GEOGRAFICAS
181
182       if(m2 > 2)
183       {
184         Candidatos3[n3] = o;
185         n3++;
186       }
187     }
188
189     if(n3 == 0)//NO HAY CAMBIOS POSIBLES
190           j = -1;
191     if(n3 == 1)//SOLO HAY UN POSIBLE CAMBIO
192           j = 0;
193     if(n3 > 1)//HAY DOS O MAS CAMBIOS POSIBLES
194     {
195         //SI NO HAY MUNICIPIOS CON UN ALTO PORCENTAJE DE POBLACION INDIGENA
196           //SE ELIGE UNA UNIDAD GEGRAFICA CON PROBABILIDAD 1/n
197           j = SiguienteAleatorioEnteroModN(& Semilla, n3);
198     }
199
200     if(j > -1)
201     {
202       Unidad_Elegida = Candidatos3[j];
203       m2 = Distrito[Unidad_Elegida];
204       Distrito[Unidad_Elegida] = OrigenAB;
205
206       //EL CANDIDATO Unidad_Elegida ES CAMBIADO AL DISTRITO OrigenAB
207       kk = -1;
208       for(j = 0; j < UnidadesPorConjunto; j++)
209       //SE CUENTA EL NUMERO DE UNIDADES EN EL DISTRITO QUE ACABA DE ABANDONAR Unidad_Elegida
210       //QUE NO ESTAN EN EL MISMO DISTRITO QUE LA UNIDAD k EN LA FUENTE DE ALIMENTO AB1
211       {
212         if(Distrito[j] == m2 && FuenteAlimento[AB1][j] != FuenteAlimento[AB1][k])
213         {
214           kk++;
215           Candidatos1[kk] = j;
216         }
217       }
218
219       if(kk < 0)
220             RevisaConexidad_Observadora1(m2);//REVISA CONEXIDAD Y EN CASO DE HABERSE PERDIDO
221                                              //LA COMPONENTE CONEXA MAS GRANDE SERA EL NUEVO DISTRITO m2
222       else
223       {
224         if(kk == 0)
225               kk = Candidatos1[0];
226         else
227         {
228           kk++;
229           kk = SiguienteAleatorioEnteroModN(& Semilla, kk);
230           kk = Candidatos1[kk];
231         }
232         RevisaConexidad_Observadora2(m2,kk); //REVISA CONEXIDAD Y EN CASO DE HABERSE PERDIDO
233                                              //LA COMPONENTE CONEXA CON LA UNIDAD kk SERA EL NUEVO DISTRITO m2
234       }
235     }
236
237     Evalua_Solucion();
238
239     return(1);
240   }
```

## Anexo : Función RevisaConexidad_Observadora1(int DistritoAnalizado)

```
1   int RevisaConexidad_Observadora1(int DistritoAnalizado)
2   //CON ESTA FUNCION SE REVISA SI EL DistritoAnalizado PERDIO LA CONEXIDAD
3   {
4     int j,i,m,suma;
5     int Representante[6500], Componente[6500],N = 0,n;
6     int Lista1[6500], Lista2[6500];
7     int Contactado[6500];
8
9     suma = 0;
10
11    //SI EL NUMERO DE UNIDADES GEOGRAFICAS ES MENOR O IGUAL A 1 NO SE HA PERDIDO LA CONEXIDAD
12    for(m=0;m<UnidadesPorConjunto;m++)
13    {
14      Contactado[m] = 0;
15      Componente[m] = 0;
```

```
    if(Distrito[m] == DistritoAnalizado)
    {
      Lista1[suma] = m;
      suma++;
    }
  }

  if ( suma <= 1)
    return(0);

  //SE REVISA EL NUMERO DE COMPONENTES CONEXAS EN EL DISTRITO
  //SI EL NUMERO DE COMPONENTES CONEXAS ES IGUAL A 1 NO SE HA PERDIDO LA CONEXIDAD
  //SI EL NUMERO DE COMPONENTES CONEXAS ES MAYOR O IGUAL A 2 SE DEBE REPARAR LA CONEXIDAD
  for(i=0; i< suma; i++)
  {
    n = 0;
    if(Contactado[Lista1[i]] == 0)
    {
      Representante[N] = Lista1[i];
      Lista2[n] = Lista1[i];
      Contactado[Lista2[n]] = 1;
      Componente[N]++;
      n++;
      for(j=0; j < n; j++)
      {
        for(m=0; m < Vecinos[Lista2[j]][60]; m++)
        {
          if(Distrito[Vecinos[Lista2[j]][m]] == DistritoAnalizado && Contactado[Vecinos[Lista2[j]][m]] == 0)
          {
            Lista2[n] = Vecinos[Lista2[j]][m];
            Contactado[Vecinos[Lista2[j]][m]] = 1;
            Componente[N]++;
            n++;
          }
        }
      }
      N++;
    }
  }

  if( N == 1)
        return(0);

  //SE DETERMINA CUAL ES LA COMPONENTE CON MAYOR NUMERO DE UNIDADES GEOGRAFICAS
  //PARA DEJARLA COMO EL DISTRITO Origen

  n = 0;
  j = Componente[n];
  for(i = 0; i < N; i++)
  {
    if(Componente[i] > j)
    {
      j = Componente[i];
      n = i;
    }
  }

  for(i = 0; i < N; i++)
  {
    if(i != n)
      ReparaConexidad_Observadora(DistritoAnalizado,Representante[i]);
  }

  return(0);
}
```

## Anexo : Función RevisaConexidad_Observadora2(int DistritoOrigen, int UnidadOrigen)

```
int RevisaConexidad_Observadora2(int DistritoOrigen, int UnidadOrigen)
//CON ESTA FUNCION SE REVISA SI EL DISTRITO Origen PERDIO LA CONEXIDAD
{
  int j,i,m,suma;
  int Representante[6500], Componente,N = 0,n;
  int Lista1[6500], Lista2[6500];
  int Contactado[6500];
  suma = 0;

  //SE REVISA EL NUMERO DE UNIDADES GEOGRAFICAS EN EL DISTRITO

```

```
12    for(m=0;m<UnidadesPorConjunto;m++)
13    {
14      Contactado[m] = 0;
15      Representante[m] = -1;
16      if(Distrito[m] == DistritoOrigen)
17      {
18        Lista1[suma] = m;
19        suma++;
20      }
21    }
22
23    //SI EL NUMERO DE UNIDADES GEOGRAFICAS ES MENOR O IGUAL A 1 NO SE HA PERDIDO LA CONEXIDAD
24
25    if ( suma == 0)
26    {
27      printf("\n ERROR Distrito vacio\n");
28      getchar();
29      return(0);
30    }
31
32    if ( suma == 1)
33    {
34      return(0);
35    }
36
37    //SE REVISA EL NUMERO DE COMPONENTES CONEXAS EN EL DISTRITO
38    //SI EL NUMERO DE COMPONENTES CONEXAS ES IGUAL A 1 NO SE HA PERDIDO LA CONEXIDAD
39    //SI EL NUMERO DE COMPONENTES CONEXAS ES MAYOR O IGUAL A 2 SE DEBE REPARAR LA CONEXIDAD
40
41    for(i=0; i< suma; i++)
42    {
43      n = 0;
44      if(Contactado[Lista1[i]] == 0)
45      {
46        Representante[N] = Lista1[i];
47        Lista2[n] = Lista1[i];
48        Contactado[Lista2[n]] = 1;
49        if(Lista2[n] == UnidadOrigen)
50                  Componente = N;
51
52        n++;
53        for(j=0; j < n;j++)
54        {
55          for(m=0; m< Vecinos[Lista2[j]][60]; m++)
56          {
57            if(Distrito[Vecinos[Lista2[j]][m]] == DistritoOrigen && Contactado[Vecinos[Lista2[j]][m]] == 0)
58            {
59              Lista2[n] = Vecinos[Lista2[j]][m];
60              Contactado[Vecinos[Lista2[j]][m]] = 1;
61              if(Lista2[n] == UnidadOrigen)
62                          Componente = N;
63              n++;
64            }
65          }
66        }
67        N++;
68      }
69    }
70
71    if( N == 1)
72          return(0);
73
74      //LAS UNIDADES SON ENVIADAS A DISTRITOS VECINOS, EXCEPTO LAS UBICADAS EN Componente
75
76    for(i = 0; i < N; i++)
77    {
78      if(i != Componente)
79              ReparaConexidad_Observadora(DistritoOrigen,Representante[i]);
80    }
81
82    return(0);
83  }
```

## Anexo : Función ReparaConexidad_Observadora(int DistritoOrigen, int UnidadOrigen)

```
1  int ReparaConexidad_Observadora(int DistritoOrigen, int UnidadOrigen)
2  //ESTA FUNCION ES LLAMADA CUANDO SE HA COMPROBADO FALTA DE CONEXIDAD EN DistritoOrigen
3  //LA UNIDAD GEOGRAFICA UnidadOrigen ES UN REPRESENTANTE DE UNA COMPONENTE CONEXA DE DistritoOrigen
4  {
```

```
5   int j,i,m,n;
6   int Lista[6500];
7   int Destinos[45],Destinos2[45],mm;
8   int Contactado[6500];
9
10  for(m = 0;m< NDistritos;m++)
11  {
12    Destinos[m] = 0;
13    Destinos2[m] = -1;
14  }
15  for(m=0;m<UnidadesPorConjunto;m++)
16        Contactado[m] = 0;
17
18  n = 0;
19  Lista[n] = UnidadOrigen;
20  Contactado[Lista[n]] = 1;
21  n++;
22
23  //SE ENCUENTRA A TODAS LAS UNIDADES GEOGRAFICAS EN LA MISMA COMPONENTE CONEXA QUE UnidadOrigen
24  //SE HACE UNA LISTA CON TODOS LOS DISTRITOS A LOS QUE SE PUEDE ENVIAR A LA COMPONENTE
25
26  mm = 0;
27  for(j=0; j < n;j++)
28  {
29    for(m=0; m< Vecinos[Lista[j]][60]; m++)
30    {
31      if(Distrito[Vecinos[Lista[j]][m]] == DistritoOrigen && Contactado[Vecinos[Lista[j]][m]] == 0)
32      {
33        Lista[n] = Vecinos[Lista[j]][m];
34        Contactado[Vecinos[Lista[j]][m]] = 1;
35        n++;
36      }
37
38      if(Distrito[Vecinos[Lista[j]][m]] != DistritoOrigen && Destinos[Distrito[Vecinos[Lista[j]][m]]] == 0)
39      {
40        Destinos[Distrito[Vecinos[Lista[j]][m]]] = 1;
41        Destinos2[mm] = Distrito[Vecinos[Lista[j]][m]];
42        mm++;
43      }
44    }
45  }
46
47    //CADA COMPONENTE ES ENVIADA A UN DISTRITO VECINO
48
49  while(n > 0)
50    {
51
52        for(j=0; j < n;j++)
53        {
54            for(m = 0;m< NDistritos;m++)
55            {
56                Destinos[m] = 0;
57                Destinos2[m] = -5;
58            }
59
60            mm = 0;
61            for(m=0; m< Vecinos[Lista[j]][60]; m++)
62            {
63                if(Distrito[Vecinos[Lista[j]][m]] != DistritoOrigen && Destinos[Distrito[Vecinos[Lista[j]][m]]] == 0)
64                {
65                    Destinos[Distrito[Vecinos[Lista[j]][m]]] = 1;
66                    Destinos2[mm] = Distrito[Vecinos[Lista[j]][m]];
67                    mm++;
68                }
69            }
70
71            //SE ELIGE UN DESTINO PARA LAS UNIDADES GEOGRAFICAS EN LA COMPONENTE CONEXA
72            if(mm > 1)
73            {
74                mm = SiguienteAleatorioEnteroModN(& Semilla, mm);
75                Distrito[Lista[j]] = Destinos2[mm];
76
77                for(m = j; m< n-1; m++)
78                    Lista[m] = Lista[m+1];
79                n--;
80                j--;
81                mm = 0;
82            }
83
84            if( mm == 1)
85            {
86                Distrito[Lista[j]] = Destinos2[0];
87                for(m = j; m< n-1; m++)
88                    Lista[m] = Lista[m+1];
89                n--;
90                j--;
91            }
92        }
93    }
```

```
94
95    return(0);
96  }
```

## Anexo : Función Evalua_Solucion(void)

```
1  void Evalua_Solucion(void)
2  //CALCULA EL COSTO DE UNA FUENTE DE ALIMENTO CREADA A PARTIR DE FUENTES DE ALIMENTO YA EXISTENTES
3  {
4    int i, j, k, p;
5
6    for(i=0;i<NDistritos;i++)
7    {
8      PerimetroDistrito[i] = 0;
9      PoblacionDistrito[i] = 0;
10     AreaDistrito[i] = 0;
11   }
12
13   for(j=0;j<UnidadesPorConjunto;j++)
14   {
15     PoblacionDistrito[Distrito[j]] += PoblacionUnidadGeografica[j];
16     AreaDistrito[Distrito[j]] += AreaUnidadGeografica[j];
17     PerimetroDistrito[Distrito[j]] += PerimetroFrontera[j][j];
18     for(k = 0;k < Vecinos[j][60];k++)
19     {
20       p = Vecinos[j][k];
21       if(Distrito[p] != Distrito[j] )
22               PerimetroDistrito[Distrito[j]] += PerimetroFrontera[j][p];
23     }
24   }
25
26   DesviacionPoblacional_Nueva = Compacidad_Nueva = 0;
27   for(i=0;i<NDistritos;i++)
28   {
29     DesviacionPoblacionalDistrito[i] = Desviacion_Poblacional(PoblacionDistrito[i]);
30     CompacidadDistrito[i] = Compacidad(AreaDistrito[i],PerimetroDistrito[i]);
31     DesviacionPoblacional_Nueva += DesviacionPoblacionalDistrito[i];
32     Compacidad_Nueva += CompacidadDistrito[i];
33   }
34   Costo_Nueva = DesviacionPoblacional_Nueva + Compacidad_Nueva;
35 }
```

## Anexo : Función Desviacion_Poblacional(int Poblacion)

```
1  double Desviacion_Poblacional(int Poblacion)
2  //CALCULA EL EQUILIBRIO POBLACIONAL
3  {
4    double Costo;
5    Costo = 1.00 -(Poblacion / MediaEstatal);
6    Costo = Costo / 0.15;
7    Costo = pow(Costo, 2);
8    if(Costo > 1)
9          Costo += 10 * (Costo - 1);
10   return(Costo);
11 }
```

## Anexo : Función Compacidad(double Area,double Perimetro)

```
1  double Compacidad(double Area,double Perimetro)
2  //CALCULA LA COMPACIDAD
3  {
4    double Costo;
5    Costo = ((Perimetro / sqrt(Area)) * 0.25 - 1.0) * 0.5;
6    return(Costo);
7  }
```

## Anexo : Función Recocido_Simulado(float Temperatura, int EquilibrioFinal, int AB)

```
float Recocido_Simulado(float Temperatura, int EquilibrioFinal, int AB)
{
    int Equilibrio = 0;
    double Aceptada = 1, Entrada = 1;
    double u5, u6, b1, u7;
    int i, j;
    double MenorCostoPoblacional, MenorCompacidad;
    double DesviacionPoblacional_Actual, Compacidad_Actual;
    int Algoritmo_Usado;

    MenorCostoPoblacional = DesviacionPoblacional_FuenteAlimento[AB];
    MenorCompacidad = Compacidad_FuenteAlimento[AB];
    DesviacionPoblacional_Actual = RS_DesviacionPoblacional[AB];
    Compacidad_Actual = RS_Compacidad[AB];

    while(Equilibrio <= EquilibrioFinal)
    {
        Equilibrio++;

        //SE EMPLEA UNA ESTRATEGIA DE BUSQUEDA BASADA EN COLONIA DE ABEJAS ARTIFICIALES O EN RECOCIDO SIMULADO
        if(Equilibrio %5 == 0)
        {
            AbejaObservadora(AB);
            Algoritmo_Usado = 1;
        }
        else
        {
            CambioRS();
            Algoritmo_Usado = 0;
        }

    //LA SOLUCION ACTUAL SE GUARDA CUANDO MEJORA A LA MEJOR SOLUCION CONOCIDA

        u5 = DesviacionPoblacional_Nueva + Compacidad_Nueva;
        b1 = Costo_FuenteAlimento[AB];

        if(u5 < b1)
        {
            for(i=0; i<UnidadesPorConjunto; i++)
            {
                FuenteAlimento[AB][i] = Distrito[i];
            }

            Costo_FuenteAlimento[AB] = u5;
            DesviacionPoblacional_FuenteAlimento[AB] = DesviacionPoblacional_Nueva;
            Compacidad_FuenteAlimento[AB] = Compacidad_Nueva;
        }

        //SE DETERMINA SI LOS CAMBIOS DE UNIDADES GEOGAFICAS SERAN ACEPTADOS
        //SE ACEPTARAN CON PROBABILIDAD 1 SI EL CAMBIO MEJORA EL COSTO DE LA SOLUCION ACTUAL
        //EN OTRO CASO SE USARA EL CRITERIO DE METROPOLIS

        u5 = (DesviacionPoblacional_Actual - DesviacionPoblacional_Nueva);
        u6 = (Compacidad_Actual - Compacidad_Nueva);

        u5 = exp( (u6 + u5) / Temperatura);
        b1 = SiguienteAleatorioReal0y1(& Semilla);

        //SI SE USA UNA ESTRATEGIA DE BUSQUEDA BASADA EN RECOCIDO SIMULADO SE HACE UN
        //PRONOSTICO BASADO EN SUAVIZAMIENTO EXPONENCIAL PARA PREDECIR EL VECINDARIO QUE
        //TENDRA UNA MAYOR PROBABILIDAD DE GENERAR MEJORES SOLUCIONES
        if(u5 < 1)
        {
            Entrada++; //SE CUENTA EL NUMERO DE SOLUCIONES DE MENOR CALIDAD VISITADAS
            if(b1 >= u5)
            {
                if(Algoritmo_Usado == 0)
                {
                    Pronostico_Vecindario[VecindarioRS] = Pronostico_Vecindario[VecindarioRS] + 0.2 * (0 - Pronostico_Vecindario[
                        VecindarioRS]);
                    u6 = 0.00;
                    for(i = 1; i < 4; i++)
                    {
                        if(Pronostico_Vecindario[i] < (7.00 - i) * 0.004761905)
                            Pronostico_Vecindario[i] = (7.00 - i) * 0.004761905;
                        u6 += Pronostico_Vecindario[i];
                    }

                    //SE ELIGE UN VECINDARIO DANDOLE MAYOR PROBABILIDAD AL DE MEJOR PRONOSTICO

                    i = 1;
                    u7 = Pronostico_Vecindario[i]/u6;
                    while(b1 >= u7 && i < 3)
                    {
```

```
                        i++;
                        u7 += Pronostico_Vecindario[i] / u6;
                    }
                    VecindarioRS = i;
                }
            }
        }

        if(u5 >= 1)
        {
            if(Algoritmo_Usado == 0)
                Pronostico_Vecindario[VecindarioRS] = Pronostico_Vecindario[VecindarioRS] + 0.2 * (1 - Pronostico_Vecindario[
                    VecindarioRS]);
        }

        if(u5 < 1 && b1 < u5)
        {
            if(Algoritmo_Usado == 0)
                Pronostico_Vecindario[VecindarioRS] = Pronostico_Vecindario[VecindarioRS] + 0.2 * (0.5 - Pronostico_Vecindario[
                    VecindarioRS]);
        }

        if(b1 < u5)
        {
            if(u5 < 1)
                Aceptada++;//SE CUENTA EL NUMERO DE VECES QUE SE ACEPTA UNA SOLUCION DE MENOR CALIDAD

            //SE REALIZAN LOS CAMBIOS SUGERIDOS EN LA FUNCION Cambios()
            //Y SE ACTUALIZAN LOS COSTOS
            DesviacionPoblacional_Actual = DesviacionPoblacional_Nueva;
            Compacidad_Actual = Compacidad_Nueva;

        }
        else
        {
            //SE RECHAZAN LOS CAMBIOS SUGERIDOS EN LA FUNCION Cambios()
            for(i=0; i<UnidadesPorConjunto; i++)
            {
                Distrito[i] = Solucion[i];
            }
            DesviacionPoblacional_Nueva = DesviacionPoblacional_Actual;
            Compacidad_Nueva = Compacidad_Actual ;
        }

    }
    RS_Compacidad[AB] = Compacidad_Actual;
    RS_DesviacionPoblacional[AB] = DesviacionPoblacional_Actual;

    RS_Costos[AB] = DesviacionPoblacional_Actual + Compacidad_Actual ;

    return(Aceptada / Entrada);
}
```

## Anexo : Función Recocido_Simulado2(float Temperatura, int AB, int Iteraciones)

```
void Recocido_Simulado2(float Temperatura, int AB, int Iteraciones)
//REALIZA MODIFICACIONES EN BUECA DE MEJORAS EN Soluciones_Iniciales
{
    int Equilibrio = 0;
    double Aceptada = 1, Entrada = 1;
    double u5, u6, b1, u7;
    int i, j;
    VecindarioRS = 1;
    while(Equilibrio <= Iteraciones)
    {
        Equilibrio++;
        //SE REALIZA UN CAMBIO Y DENTRO DE LA FUCION Cambios() SE EVALUA SU COSTO TOTAL
        CambioRS();

        //LA SOLUCION ACTUAL SE GUARDA CUANDO MEJORA A Soluciones_Iniciales

    u5 = DesviacionPoblacional_Nueva + Compacidad_Nueva;
        u7 = b1 = Costo_Iniciales[AB];

        u6 = exp( (b1 - u5) / Temperatura);
        b1 = SiguienteAleatorioReal0y1(& Semilla);

        if(b1 < u6)
        {
            for(i=0; i<UnidadesPorConjunto; i++)
            {
```

```
                Soluciones_Iniciales[AB][i] = Distrito[i];
            }
            Costo_Iniciales[AB] = u5;
        }
        else
        {
            for(i=0; i<UnidadesPorConjunto; i++)
            {
                Distrito[i] = Solucion[i];
            }
        }
    }
}
```

## Anexo : Función CambioRS(void)

```
void CambioRS(void)
//SE GENERA UNA SOLUCION VECINA DE Distritos_Actuales
{
  int b, i, j, n, k, n3, p0, p1;
  int DistritoOrigen, DistritoDestino, Unidad_Elegida;
  int destinosE[45], Candidatos[6500];
  double b1;

  b1 = SiguienteAleatorioReal0y1(& Semilla);

  eligem1:

  j = 1;
  //SE ELIGE UN DISTRITO AL AZAR

  while(j <= 1)
  {
    DistritoOrigen = SiguienteAleatorioEnteroModN(& Semilla, NDistritos);
    //SE EVALUA SI EL DistritoOrigen TIENE MAS DE UNA UNIDAD GEOGRAFICA
    j = Cardinalidad_DistritoRS(DistritoOrigen);
  }
  p0 = j;

  //SE HACE UNA LISTA CON LAS UNIDADES GEOGRAFICAS QUE SE PUEDEN CAMBIAR DEL DistritoOrigen
  //EN ESTE CASO SE CONSIDERAN A TODAS LAS UNIDADES DEL DistritoOrigen QUE COLINDAN CON OTRO DISTRITO

  C5: Unidades_Cambiadas = 6500;
  n = 0;
  for(i = 0; i < UnidadesPorConjunto; i++)
  {
    if(Distrito[i] == DistritoOrigen)
    {
      k = j = 0;
      while(k < 6500)
      {
        k = Vecinos[i][j];
        if(Vecinos[i][j] < 6500)
        {
          Unidad_Elegida = Vecinos[i][j];
          if(Distrito[Unidad_Elegida] != DistritoOrigen)
          {
            Candidatos[n] = i;
            n++;
            k = 6500;
          }
        }
        j++;
      }
    }
  }

  //SE SELECCIONA UNA UNIDAD GEOGRAFICA PARA SER ENVIADA A OTRO DISTRITO
    regresa:
  b = SiguienteAleatorioEnteroModN(& Semilla, n);
  Unidad_Elegida = Candidatos[b];
    p1 = b;

  //SE DETERMINAN LOS DISTRITOS VECINOS DE Unidad_Elegida
  k = 0;
  for(i=0; i<Vecinos[Unidad_Elegida][60]; i++)
  {
    if(Distrito[Vecinos[Unidad_Elegida][i]] != DistritoOrigen)
    {
      if(k > 0)
      {
```

```
66          j = 0;
67          for(n3 = 0; n3 <k; n3++)
68          {
69            if(destinosE[n3] != Distrito[Vecinos[Unidad_Elegida][i]])
70                        j++;
71          }
72          if(j == k)
73          {
74            destinosE[k] = Distrito[Vecinos[Unidad_Elegida][i]];
75            k++;
76          }
77        }
78        if(k == 0)
79        {
80          destinosE[k] = Distrito[Vecinos[Unidad_Elegida][i]];
81          k++;
82        }
83      }
84    }
85
86    //SE ELIGE UN DISTRITO DESTINO PARA Unidad_Elegida
87      if(k == 1)
88          n3 = 0;
89      else
90          n3 = SiguienteAleatorioEnteroModN(& Semilla, k);
91
92      for(i=0; i<UnidadesPorConjunto; i++)
93      {
94          Solucion[i] = Distrito[i];
95      }
96
97    DistritoDestino = destinosE[n3];
98    Distrito[Unidad_Elegida] = DistritoDestino;
99    Unidades_Cambiadas = Unidad_Elegida;
100   Distrito_Destino = DistritoDestino;
101   Distrito_Origen = DistritoOrigen;
102
103   //SE EVALUA EL COSTO DEL NUEVO ESCENARIO
104
105     if( 2 <= VecindarioRS && VecindarioRS <= 3 && 2 < p0)
106     {
107         n = 0;
108         for(i=0; i<Vecinos[Unidad_Elegida][60]; i++)
109         {
110             if(Distrito[Vecinos[Unidad_Elegida][i]] == DistritoOrigen)
111             {
112                 Candidatos[n] = Vecinos[Unidad_Elegida][i];
113                 n++;
114             }
115         }
116
117         if(n == 1)
118             Unidad_Elegida = Candidatos[0];
119
120         if(n > 1)
121         {
122             b = SiguienteAleatorioEnteroModN(&Semilla,n);
123             Unidad_Elegida = Candidatos[b];
124         }
125
126         if(n >= 1)
127         {
128             if(n > 1)
129             {
130                 for(i=b; i < n-1; i++)
131                     Candidatos[i] = Candidatos[i+1];
132                 n--;
133             }
134
135             Distrito[Unidad_Elegida] = DistritoDestino;
136         }
137
138         if(VecindarioRS >= 3 && 3 < p0)
139         {
140             for(i=0; i<Vecinos[Unidad_Elegida][60]; i++)
141             {
142                 if(Distrito[Vecinos[Unidad_Elegida][i]] == DistritoOrigen)
143                 {
144                     for(j = 0; j < n; j++)
145                         if(Candidatos[j] == Vecinos[Unidad_Elegida][i])
146                             break;
147                     if(j == n)
148                     {
149                         Candidatos[n] = Vecinos[Unidad_Elegida][i];
150                         n++;
151                     }
152                 }
153             }
154
```

```
155             if(n == 1)
156                 Unidad_Elegida = Candidatos[0];
157
158             if(n > 1)
159             {
160                 b = SiguienteAleatorioEnteroModN(&Semilla,n);
161                 Unidad_Elegida = Candidatos[b];
162             }
163
164             if(n >= 1)
165             {
166                 if(n > 1)
167                 {
168                     for(i=b; i < n-1; i++)
169                         Candidatos[i] = Candidatos[i+1];
170                     n--;
171                 }
172                 Distrito[Unidad_Elegida] = DistritoDestino;
173                 Unidades_Cambiadas = Unidad_Elegida;
174             }
175         }
176     }
177
178     RevisaConexidad_Empleada(DistritoOrigen, DistritoDestino);
179
180     //SE EVALUA EL COSTO DEL NUEVO ESCENARIO
181     Evalua_Solucion();
182 }
```

## Anexo : Función Cardinalidad_DistritoRS(int Distritos)

```
1  int Cardinalidad_DistritoRS(int Distritos)
2  //CALCULA EL NUMERO DE UNIDADES GEOGRAFICAS EN Distritos
3  {
4    int m,suma;
5    suma = 0;
6    for(m = 0; m< UnidadesPorConjunto; m++)
7    {
8      if(Distrito[m] == Distritos)
9              suma++;
10   }
11   return(suma);
12 }
```

## Anexo : Función Busqueda_Local()

```
1  int Busqueda_Local()
2  {
3      int i, j, m, k, n3, Mejora = 0;
4      int Cardinalidad, Unidad_Elegida;
5      int DistritoOrigen, DistritoDestino;
6    int destinosE[30], Destino, Mejor_Unidad;
7    double Costo_DestinoE;
8    int Distrito_Aux[6500];
9
10   j = 0;
11
12     for(i=0; i<UnidadesPorConjunto; i++)
13         Distrito_Aux[i] = Distrito[i];
14
15     Evalua_Solucion();
16     Costo_DestinoE = Costo_Nueva;
17     Destino = -1;
18     m = 0;
19
20   while(m == 0)
21     {
22         m = 1;
23         for(Unidad_Elegida = 0; Unidad_Elegida < UnidadesPorConjunto; Unidad_Elegida++)
24         {
25             //SE REVISA QUE EL DISTRITO CONTENGA MAS DE 2 UNIDADES GEOGRAFICAS
26             Cardinalidad = Cardinalidad_DistritoRS(Distrito[Unidad_Elegida]);
27             if(Cardinalidad >= 2)
28             {
```

```
                DistritoOrigen = Distrito[Unidad_Elegida];
                //SE DETERMINAN LOS DISTRITOS VECINOS DE Unidad_Elegida
                k = 0;
                for(i=0; i<Vecinos[Unidad_Elegida][60]; i++)
                {
                    if(Distrito[Vecinos[Unidad_Elegida][i]] != DistritoOrigen)
                    {
                        if(k > 0)
                        {
                            j = 0;
                            for(n3 = 0; n3 <k; n3++)
                            {
                                if(destinosE[n3] != Distrito[Vecinos[Unidad_Elegida][i]])
                                    j++;
                            }
                            if(j == k)
                            {
                                destinosE[k] = Distrito[Vecinos[Unidad_Elegida][i]];
                                k++;
                            }
                        }
                        if(k == 0)
                        {
                            destinosE[k] = Distrito[Vecinos[Unidad_Elegida][i]];
                            k++;
                        }
                    }
                }

                //SE REVISAN LOS POSIBLES CAMBIOS DE Unidad_Elegida A DISTRITOS VECINOS

                for(i = 0; i < k; i++)
                 {
                    //SE ELIGE UN DISTRITO DESTINO PARA Unidad_Elegida
                    DistritoDestino = destinosE[i];
                    Distrito[Unidad_Elegida] = DistritoDestino;
                    Distrito_Origen = DistritoOrigen;
                    Distrito_Destino = DistritoDestino;
                    Unidades_Cambiadas = Unidad_Elegida;

                    //SE REVISA SI SE PROVOCO ALGUNA DISCONEXION
                    j= RevisaConexidad_Empleada(DistritoOrigen, DistritoDestino);

                    //SE EVALUA EL COSTO DEL NUEVO ESCENARIO
                    Evalua_Solucion();

                    //SE GUARDA LA INFORMACION EN CASO DE MEJORA
                    if(Costo_DestinoE > Costo_Nueva)
                    {
                        Costo_DestinoE = Costo_Nueva;
                        Destino = i;
                    }

                    //SE REGRESAN TODAS LAS UNIDADES CAMBIADAS AL DISTRITO Origen
                        for(i=0; i<UnidadesPorConjunto; i++)
                            Distrito[i] = Distrito_Aux[i];

                    //SE REGRESAN LOS COSTOS A SU NIVEL ORIGINAL
                }

                if(Destino != -1)
                {
                    i = Destino;
                    Destino = -1;
                    DistritoDestino = destinosE[i];
                    Distrito[Unidad_Elegida] = DistritoDestino;
                    Distrito_Origen = DistritoOrigen;
                    Distrito_Destino = DistritoDestino;
                    Unidades_Cambiadas = Unidad_Elegida;

                    //SE REVISA SI SE PROVOCO ALGUNA DISCONEXION
                    j= RevisaConexidad_Empleada(DistritoOrigen, DistritoDestino);

                    //SE EVALUA EL COSTO DEL NUEVO ESCENARIO
                    Evalua_Solucion();

                    for(i=0; i<UnidadesPorConjunto; i++)
                        Distrito_Aux[i] = Distrito[i];

            //SE REALIZAN LOS CAMBIOS SUGERIDOS
            //Y SE ACTUALIZAN LOS COSTOS

                    m = 0;
                    Mejora = 1;
                }
            }
        }
    }
    return(Mejora);
```

```
118 }
```

## Anexo : Función SiguienteAleatorioEnteroModN(long * semilla, int n)

```
1  int SiguienteAleatorioEnteroModN(long * semilla, int n)
2  // DEVUELVE UN ENTERO ENTRE 0 Y n-1
3  {
4    double a;
5    int v;
6    long double zi, mhi31 = 2147483648u, ahi31 = 314159269u, chi31 = 453806245u;
7    long int dhi31;
8    zi     = *semilla;
9    zi     = (ahi31 * zi) + chi31;
10   if(zi > mhi31)
11   {
12     dhi31 = (long int) (zi / mhi31);
13     zi = zi - (dhi31 * mhi31);
14   }
15   *semilla = (long int)zi;
16   zi = zi / mhi31;
17   a = zi;
18   v = (int)(a * n);
19   if (v == n)
20     return(v-1);
21   return(v);
22 }
```

## Anexo : Función SiguienteAleatorioReal0y1(long * semilla)

```
1  double SiguienteAleatorioReal0y1(long * semilla)
2  {
3    long double zi, mhi31 = 2147483648u, ahi31 = 314159269u, chi31 = 453806245u;
4    long int dhi31;
5    zi     = *semilla;
6    zi     = (ahi31 * zi) + chi31;
7    if(zi > mhi31)
8    {
9      dhi31 = (long int) (zi / mhi31);
10     zi = zi - (dhi31 * mhi31);
11   }
12   *semilla = (int) zi;
13   zi = zi / mhi31;
14   return (zi);
15 }
```

# REFERENCIAS

[1] S. G. Cobos, J. G. Close, M. A. Gutiérrez, A. E. Martínez, “Problemas de optimización en Búsqueda y exploración estocástica”, Ed. México: UAM-I, 2010, pp. 33-98.

[2] S. Kirkpatrick, C. D. Gellat, M. P. Vecchi, “Optimization by simulated annealing”, Science, vol. 220, pp. 671-680.

[3] D Karaboga, An idea based on honey bee swarm for numerical optimization, Technical Report TR06, Computer Engineering Department, Erciyes University, Turkey 2005.

[4] D Karaboga, B Basturk, A powerful and efficient algorithm for numerical function optimization: artificial bee colony (ABC) algorithm, Journal of Global Optimization 39 (2007) 459-471.

[5] D Karaboga, B Basturk, On the performance of artificial bee colony (ABC) algorithm. Applied Soft Computing 8 (2008) 687-697.

[6] V. Cerny, “A thermodynamical approach to the travelling salesman problem: an efficient simulation algorithm”, Journal of Optimization Theory and Applications, vol. 45, pp. 41-55.