MANUAL DE REFERENCIA DEL SISTEMA PARA LA DISTRITACIÓN ELECTORAL 2015

INSTITUTO NACIONAL ELECTORAL

MANUAL DE REFERENCIA DEL SISTEMA PARA LA DISTRITACIÓN ELECTORAL 2015

INSTITUTO NACIONAL ELECTORAL

15 de abril del 2015

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. La mejor solución encontrada por el algoritmo se devuelve al usuario.

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.

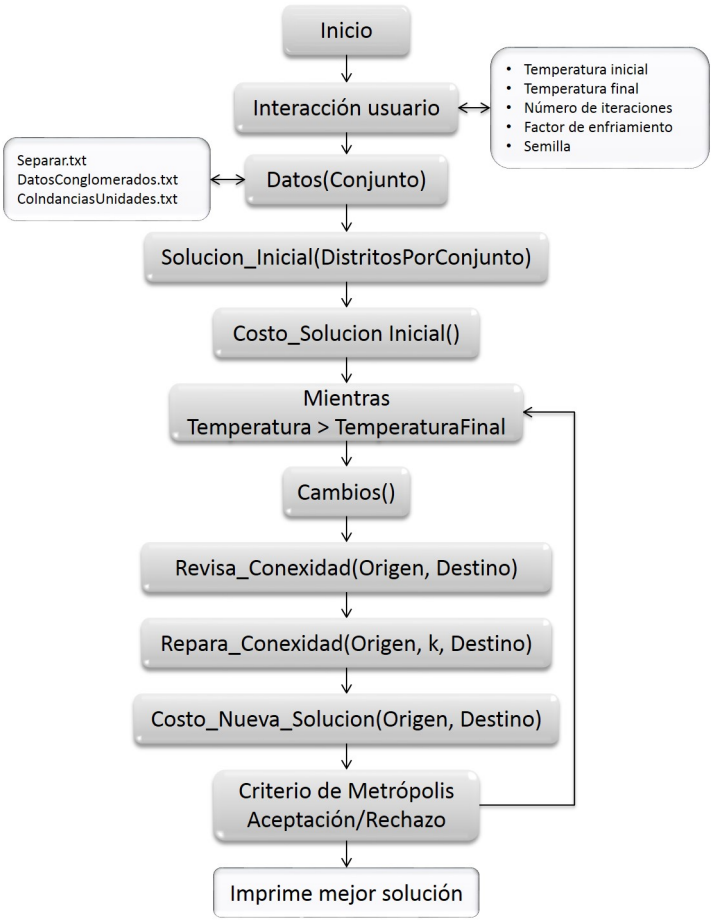

Figura 1.1 Diagrama de bloques del algoritmo basado en Recocido Simulado.

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 2015. 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 Mejores_Distritos	**int[]**	Son arreglos con la solución actual y la mejor solución visitada por el algoritmo hasta el momento.
Compacidad_Nueva DesviacionPoblacional_Nueva	**double**	Indican el costo de compacidad y desviación poblacional de la nueva solución, construida como vecina de la solución actual.
PoblacionDistrito_Origen PoblacionDistrito_Destino 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 DesviacionPoblacional_Destino AreaDistrito_Origen AreaDistrito_ Destino PerimetroDistrito_Origen PerimetroDistrito_Destino CompacidadDistrito_Origen CompacidadDistrito_Destino	**double**	Indican los costos de los distritos que se modifican al construir una solución nueva.
MedidaArea 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 Compacidad_Actual	**double**	Indican el costo del escenario actual, es la suma de los costos de los distritos.

DesviacionPoblacionalZonas_Actuales 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 Distrito_Origen	**int**	Indican los distritos que son modificados para generar una solución nueva.
ConjuntoActual UnidadesPorConjunto 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 Temperatura 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 100 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 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 MenorCompacidad	**double**	Guardan el costo de la mejor solución visitada por el algoritmo durante una corrida.
Iteraciones_Caliente Iteraciones _Templado Iteraciones_Frio	**int**	Indican el número de iteraciones que se realizará a diferentes niveles de aceptación.
FactorEnfriamiento_Caliente FactorEnfriamiento_Templado FactorEnfriamiento_Frio	**double**	Indican el factor de enfriamiento a diferentes niveles de aceptación del algoritmo.
DesviacionPoblacional_EscenarioFinal 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 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 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 100 semillas. En caso de que el número de semillas en el archivo Semillero.txt sea mayor, sólo se considerarán las primeras 100 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. 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 1.

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.

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

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.

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.

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.

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 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.

2.8. 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 5.

Algoritmo 5: Pseudocódigo de la función Revisa_Conexidad

```
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, digamos
   |  ComponenteConexa_{k_j}
14 |  Para k_i ≠ k_j hacer
15 |  |  Repara_Conexidad(Origen, k_i, Destino).
16 |  fin
17 fin
```

2.9. 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 6.

Algoritmo 6: 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
```

2.10. 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 7.

Algoritmo 7: 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
```

2.11. 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 $Costo$.

2.12. 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 $Costo$.

2.13. 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 $semilla$.

2.14. 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. La mejor solución encontrada por el algoritmo se devuelve al usuario.

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.

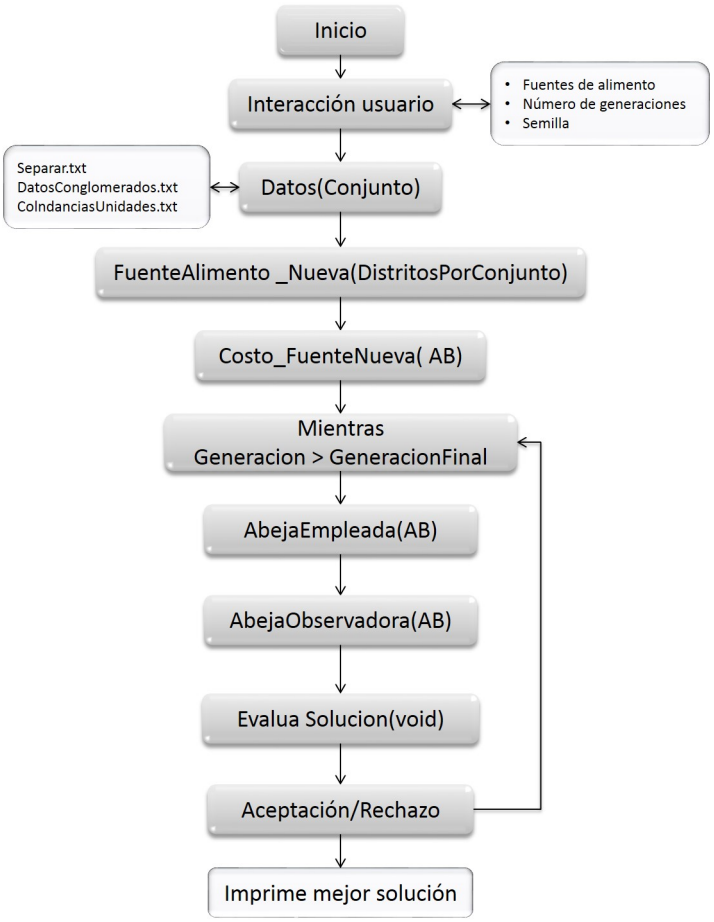

Figura 3.1 Diagrama del algoritmo basado en Colonia de Abejas Artificiales.

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 2015. 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 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 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 Compacidad_FuenteAlimento 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 DesviacionPoblacional_Nueva 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 UnidadesPorConjunto 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 MejorCompacidad	**double[]**	Almacenan el costo de cada distrito del escenario construido por el algoritmo después de una corrida.
Menor_DesviacionPoblacional 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 Compacidad_MejorEscenario	**double**	Guardan los costos del mejor escenario construido por el algoritmo,
DesviacionPoblacional_Hibrida 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 100 semillas. En caso de que el número de semillas en el archivo Semillero.txt sea mayor, sólo se considerarán las primeras 100 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. 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 8.

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.

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.

Algoritmo 8: 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 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 9.

Algoritmo 9: 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 10.

Algoritmo 10: 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 RevisaConexidad_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 11.

Algoritmo 11: 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 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.

4.8. 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 12.

Algoritmo 12: 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
```

4.9. 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 13.

Algoritmo 13: 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
```

4.10. 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 14.

Algoritmo 14: 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.

4.11. 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 15.

Algoritmo 15: 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
```

4.12. 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 16.

Algoritmo 16: 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
```

4.13. 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 17.

Algoritmo 17: 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
```

4.14. 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 18.

Algoritmo 18: 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
```

4.15. 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 $Costo$.

4.16. 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 $Costo$.

4.17. 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 $semilla$.

4.18. 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 ge-

nerar, 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 RECOCIDO SIMULADO

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[100], 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];
  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 PARMETROS

  FactorEnfriamiento_Templado = 0.98;
```

```
26  FactorEnfriamiento_Frio = 0.99;
27
28  //SE LEEN LOS PARAMETROS SOLICITADOS POR EL USUARIO
29  char *direccion = "Recocido_Simulado\\Parametros_RS.txt";
30  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 < 100)
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 2015\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 2015\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  direccion = "Sistema_de_visualizacion\\Insumos\\ConjuntosDistritos.txt";
97  fp = fopen(direccion, "r");
98  while((c=fgetc(fp))!=EOF)
99  {
100   fscanf(fp,"%d %d",&k,&i);
101   Distritos_Por_Conjunto[i] = k;
102   Numero_de_Conjuntos = i;
103   ConjuntosTotales += k;
104 }
105 fclose(fp);
106
107 for(i=0; i<6500; i++)
108   {
109       Solucion_Hibrida[i] = 6500;
110   }
111
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\\RS_Costo_Por_Conjunto.csv");
121             fp = fopen(dummy, "w");
122             fprintf(fp,"Semilla %d,Costo Total, Equilibrio Poblacional,Compacidad\n", Semillas[Corrida]);
123             fclose(fp);
124       }
125       if(Corrida > 0)
126     {
127       sprintf(dummy, "Sistema_de_visualizacion\\Resumen_Costos\\RS_Costo_Por_Conjunto.csv");
128             fp = fopen(dummy, "a");
129             fprintf(fp,"Semilla %d,Costo Total, Equilibrio Poblacional,Compacidad\n", Semillas[Corrida]);
130             fclose(fp);
131       }
132
133     for(i=0; i<6500; i++)
134         {
135             DistritosFinales[i] = 6500;
136         }
137
138     if(Numero_de_Semillas > 1)
139             printf("\n\n\t Inicia la corrida = %d con la Semilla = %d\n",Corrida+1,Semillas[Corrida]);
140
141     Semilla = Semillas[Corrida];
142     //SE INICIA EL TIEMPO DE EJECUCION
143     clock_t start, end;
144     start = clock();
145
146     DistritosAcumulados = 0;
147
148     //SE INICIA EL PROCESO DE CONSTRUCCION DE DISTRITOS PARA CADA CONJUNTO
149     for(ConjuntoActual = 1; ConjuntoActual <= Numero_de_Conjuntos; ConjuntoActual++)
150     {
151       printf("\nConjunto territorial %d. ",ConjuntoActual);
152
153       for(i = 0; i < 6500; i++)
154             {
155                 Conversion[i] = 6500;
156             }
157
158       //LA FUNCION Datos() LEE LA INFORMACION NECESARIA PARA CONSTRUIR LOS DISTRITOS
159       //POR EJEMPLO, COLINDANCIAS, AREA, POBLACION, ETC.
160       Datos(ConjuntoActual);
161       NDistritos = Distritos_Por_Conjunto[ConjuntoActual];
162
163       //SE CONSTRUYE LA SOLUCION INICIAL Y SE EVALUA SU COSTO (PARA EL CONJUNTO EN CURSO)
164       Solucion_Inicial(NDistritos);
165
166       //Distritos_Actuales[i] ES LA SOLUCION ACTUAL DURANTE LA EJECUCION DEL ALGORITMO
167       if(Distritos_Por_Conjunto[ConjuntoActual] == 1)
168               goto Final;
169       //SE CALCULA EL COSTO DE LA SOLUCION CONSTRUIDA
170       Costo_Solucion_Inicial();
171
172       //LA SOLUCION ACTUAL SE GUARDA COMO LA MEJOR SOLUCION CONOCIDA HASTA EL MOMENTO
173       for(j = 0; j < UnidadesPorConjunto; j++)
174             {
175                 Mejores_Distritos[j] = Distritos_Actuales[j];
176             }
177
178
179       MenorCostoPoblacional = DesviacionPoblacional_Actual;
180       MenorCompacidad = Compacidad_Actual;
181       DesviacionPoblacional_Nueva = DesviacionPoblacional_Actual;
182       Compacidad_Nueva = Compacidad_Actual;
183       Entrada = Aceptada = 0;
184       DesviacionPoblacional_Nueva = DesviacionPoblacional_Actual;
185       Compacidad_Nueva = Compacidad_Actual;
186
187       //INICIA PROCESO DE MEJORA
188       Equilibrio = -1;
189       Temperatura = Temperatura_Usuario;
190       Precalentado = 1;
191       EquilibrioFinal = Iteraciones_Caliente;
192       alfa = FactorEnfriamiento_Caliente;
193
194       while(Temperatura >= TemperaturaFinal)
195       {
196         Equilibrio++;
197         if(Equilibrio >= EquilibrioFinal)
198         {
199           Entrada++;
200           //EN ESTA SECCION SE MODIFICAN LOS PARAMETROS DEL ALGORITMO DEPENDIENDO
201           //DEL NIVEL DE ACEPTACION (Aceptada/Entrada)
202           //El ESTADO PUEDE SER: CALIENTE, TEMPLADO O FRIO
```

```
203
204           if(0.60 <= Aceptada/Entrada && Precalentado == 0)
205           {
206             EquilibrioFinal = Iteraciones_Caliente;
207             alfa = FactorEnfriamiento_Caliente;
208           }
209
210           if(0.40 <= Aceptada/Entrada && Aceptada/Entrada < 0.60 && Precalentado == 0)
211           {
212             EquilibrioFinal = Iteraciones_Templado;
213             alfa = FactorEnfriamiento_Templado;
214           }
215
216           if(Aceptada/Entrada < 0.40 && Precalentado == 0)
217           {
218             EquilibrioFinal = Iteraciones_Frio;
219             alfa = FactorEnfriamiento_Frio;
220           }
221
222           //CUANDO EL NIVEL DE ACEPTACION ES MUY BAJO SE CONCLUYE EL PROCESO DE MEJORA
223           if(Aceptada/Entrada < 0.01)
224                         Temperatura = TemperaturaFinal;
225
226           //CUANDO EL NIVEL DE ACEPTACION INICIA POR ABAJO DE 0.80 SE
227           //LLEVA A CABO UN PERIODO DE PRECALENTADO (SE AUMENTA LA TEMPERATURA),
228           //HASTA OBTENER UNA ACEPTACION DE AL MENOS 0.80
229           if(Aceptada/Entrada < 0.8 && Precalentado == 1)
230                         Temperatura = Temperatura * 1.1;
231           else
232           {
233             Temperatura = Temperatura * alfa;
234             Precalentado = 0;
235           }
236
237           Equilibrio=-1;
238           Entrada = Aceptada = 0;
239         }
240
241         //SE REALIZA UN CAMBIO Y DENTRO DE LA FUCION Cambios() SE EVALUA SU COSTO TOTAL
242         Cambios();
243
244         //LA SOLUCION ACTUAL SE GUARDA CUANDO MEJORA A LA MEJOR SOLUCION CONOCIDA
245         u5 = DesviacionPoblacional_Nueva + Compacidad_Nueva;
246         b1 = MenorCostoPoblacional + MenorCompacidad;
247         if(u5 <= b1)
248         {
249           for(i=0; i<UnidadesPorConjunto; i++)
250                     {
251                         Mejores_Distritos[i] = Distritos_Actuales[i];
252                     }
253
254
255           for(i=0; i<Unidades_Cambiadas[6499]; i++)
256           {
257             m = Unidades_Cambiadas[i];
258             Mejores_Distritos[m] = Distrito_Destino;
259           }
260
261           MenorCostoPoblacional = DesviacionPoblacional_Nueva;
262           MenorCompacidad = Compacidad_Nueva;
263         }
264
265         //SE DETERMINA SI LOS CAMBIOS DE UNIDADES GEOGAFICAS SERAN ACEPTADOS
266         //SE ACEPTARAN CON PROBABILIDAD 1 SI EL CAMBIO MEJORA EL COSTO DE LA SOLUCION ACTUAL
267         //EN OTRO CASO SE USARA EL CRITERIO DE METROPOLIS
268
269         u5 = (DesviacionPoblacional_Actual - DesviacionPoblacional_Nueva);
270         u6 = (Compacidad_Actual - Compacidad_Nueva);
271         u5 = exp( (u6 + u5) / Temperatura);
272         if(u5 < 1)
273                     Entrada++; //SE CUENTA EL NUMERO DE SOLUCIONES DE MENOR CALIDAD VISITADAS
274         b1 = SiguienteAleatorioReal0y1(& Semilla);
275         if(u5 > b1)
276         {
277           if(u5 < 1)
278                         Aceptada++;//SE CUENTA EL NUMERO DE VECES QUE SE ACEPTA UNA SOLUCION DE MENOR CALIDAD
279
280           for(i=0; i<Unidades_Cambiadas[6499]; i++)
281           {
282             m = Unidades_Cambiadas[i];
283             Distritos_Actuales[m] = Distrito_Destino;
284           }
285           //SE REALIZAN LOS CAMBIOS SUGERIDOS EN LA FUNCION Cambios()
286           //Y SE ACTUALIZAN LOS COSTOS
287           DesviacionPoblacional_Actual = DesviacionPoblacional_Nueva;
288           Compacidad_Actual = Compacidad_Nueva;
289           PoblacionDistritos_Actuales[Distrito_Destino] = PoblacionDistrito_Destino;
290           MedidaArea[Distrito_Destino] = AreaDistrito_Destino;
291           MedidaPerimetro[Distrito_Destino] = PerimetroDistrito_Destino;
```

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

```
      }

      DistritosAcumulados += NDistritos;
      //TERMINA CADA CONJUNTO
    }
    //TERMINA EL PROCESO DE MEJORA

    end = clock();
    seed1 = end - start;
    z = seed1 / CLOCKS_PER_SEC;
    seed1 = seed1 / (60 * CLOCKS_PER_SEC);
    i = (int) seed1;
    seed1 = z - (60 * i);
    z = i / 60;
    printf("\n\nEL TIEMPO DE EJECUCION FUE DE: %d MINUTOS %f SEGUNDOS\n\n",i,seed1);
    for(i=0; i< 6500; i++)
        {
            Mejores_Distritos[i] = DistritosFinales[i];
        }


    //SE OBTIENE EL COSTO TOTAL DE LA SOLUCION CONSTRUIDA
    CostoTotal = DesviacionPoblacional_Nueva = Compacidad_Nueva = 0;
    for(i=0; i < ConjuntosTotales; i++)
    {
      CostoTotal += DesviacionPoblacional_EscenarioFinal[i] + Compacidad_EscenarioFinal[i];
      DesviacionPoblacional_Nueva += DesviacionPoblacional_EscenarioFinal[i];
      Compacidad_Nueva += Compacidad_EscenarioFinal[i];
    }

    //SE IMPRIME EL COSTO TOTAL DEL ESCENARIO ACTUAL
        sprintf(dummy, "Sistema_de_visualizacion\\Resumen_Costos\\RS_Costo_Por_Conjunto.csv");
        fp = fopen(dummy, "a");
        fprintf(fp,"Costo Total,%f,%f,%f\n",CostoTotal,DesviacionPoblacional_Nueva, Compacidad_Nueva);
        fclose(fp);

        if(Corrida >= 1)
        {
            sprintf(dummy, "Sistema_de_visualizacion\\Resumen_Costos\\RS_Costo_Por_Conjunto.csv");
            fp = fopen(dummy, "a");
            fprintf(fp,"Costo Total,%f,%f,%f\n",CostoTotal,DesviacionPoblacional_Nueva, Compacidad_Nueva);
            fclose(fp);
        }

    //SE IMPRIME EN ARCHIVO DE csv LA SOLUCION FINAL
    sprintf(dummy, "Sistema_de_visualizacion\\RS_Escenario_%f_%d.csv",DesviacionPoblacional_Nueva + Compacidad_Nueva , Semillas[
        Corrida]);
    fp = fopen(dummy, "w");
    fprintf(fp,"Seccion,DISTRITO\n");
    for(i = 0; i < 6500; i++)
        {
            if(Mejores_Distritos[i] < 6500)
                fprintf(fp,"%d,%d\n",i,Mejores_Distritos[i] + 1);
        }
    fclose(fp);
  }
  //TERMINA LA CORRIDA ACTUAL

  //SE IMPRIME LA SOLUCION HIBRIDA CONSTRUIDA CON LOS ESCENARIOS OBTENIDOS
  //SOLO SI SE EMPLEO EL SEMILLERO
  if(Numero_de_Semillas > 1)
  {
    direccion = "Sistema_de_visualizacion\\Escenario_Hibrido_RS.csv";
    fp = fopen(direccion, "w");
    fprintf(fp,"Seccion,DISTRITO\n");
    for(i = 0; i < 6500; i++)
        {
            if(Solucion_Hibrida[i] < 6500)
                fprintf(fp,"%d,%d\n",i,Solucion_Hibrida[i] + 1);
        }

    fclose(fp);
  }

  printf("\t Se han completado las corridas solicitadas. \n\n \t Presione una tecla para conluir el proceso.");
  getchar();

  return(1);
}
```

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 %lf %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)
```

```
85          n3 = 0;
86        else
87          n3 = SiguienteAleatorioEnteroModN(& Semilla, k);
88
89      DistritoDestino = destinosE[n3];
90      Distritos_Actuales[Unidad_Elegida] = DistritoDestino;
91      Distrito_Origen = DistritoOrigen;
92      Distrito_Destino = DistritoDestino;
93      Unidades_Cambiadas[0] = Unidad_Elegida;
94      Unidades_Cambiadas[6499] = 1;
95
96      //SE REVISA SI SE PROVOCO ALGUNA DISCONEXION
97      j = Revisa_Conexidad(DistritoOrigen, DistritoDestino);
98
99      //SE EVALUA EL COSTO DEL NUEVO ESCENARIO
100     j = Costo_Nueva_Solucion(DistritoOrigen, DistritoDestino);
101
102     //SE REGRESAN TODAS LAS UNIDADES CAMBIADAS AL DISTRITO Origen
103     for(i=0; i<Unidades_Cambiadas[6499]; i++)
104     {
105       j = Unidades_Cambiadas[i];
106       Distritos_Actuales[j] = DistritoOrigen;
107     }
108
109     return(1);
110 }
```

Anexo : Función Cardinalidad_Distrito(int Distrito)

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

Anexo : Función Revisa_Conexidad(int Origen, int Destino)

```
1  int Revisa_Conexidad(int Origen, int Destino)
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[6500],N = 0,n;
6    int Lista1[6500], Lista2[6500];
7    int Contactado[6500];
8
9    suma = 0;
10
11   //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     Componente[m] = 0;
17     if(Distritos_Actuales[m] == Origen)
18     {
19       Lista1[suma] = m;
20       suma++;
21     }
22   }
23
24   //SI EL NUMERO DE UNIDADES GEOGRAFICAS ES MENOR O IGUAL A 1 NO SE HA PERDIDO LA CONEXIDAD
25   if( suma == 0)
26   {
27     printf("\n ERROR Distrito vacia \n");
28     getchar();
29     return(0);
30   }
```

```
31
32    if( suma == 1)
33          return(0);
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        n++;
47        for(j=0; j < n; j++)
48        {
49          for(m=0; m < Vecinos[Lista2[j]][60]; m++)
50          {
51            if(Distritos_Actuales[Vecinos[Lista2[j]][m]] == Origen && Contactado[Vecinos[Lista2[j]][m]] == 0)
52            {
53              Lista2[n] = Vecinos[Lista2[j]][m];
54              Contactado[Vecinos[Lista2[j]][m]] = 1;
55              n++;
56            }
57          }
58        }
59        Componente[N] = n;
60        N++;
61      }
62    }
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
3   //DEL DISTRITO Origen Y DEL DISTRITO Destino
4   {
5     int j, k;
6     double MedidaPerimetroE[30], MedidaAreaE[30] , DesviacionPoblacionalE;
7     int PoblacionDistritoE[30];
8     double DesviacionPoblacionalDistritoE[30], MedidaCompacidadE[30], CompacidadTotalE;
9     int o, p;
10    Distrito_Origen = Origen;
11    Distrito_Destino = Destino;
12    DesviacionPoblacionalE = DesviacionPoblacional_Nueva - DesviacionPoblacionalDistritos_Actuales[Origen] -
            DesviacionPoblacionalDistritos_Actuales[Destino];
13    CompacidadTotalE = Compacidad_Nueva - CompacidadDistritos_Actuales[Origen] - CompacidadDistritos_Actuales[Destino];
14
15    PoblacionDistritoE[Origen] = PoblacionDistritos_Actuales[Origen];
16    PoblacionDistritoE[Destino] = PoblacionDistritos_Actuales[Destino];
17    MedidaAreaE[Origen] = MedidaArea[Origen];
18    MedidaAreaE[Destino] = MedidaArea[Destino];
19
20    for(k = 0; k < Unidades_Cambiadas[6499]; k++)
21    {
22      PoblacionDistritoE[Origen] -= PoblacionUnidadGeografica[Unidades_Cambiadas[k]];
23      PoblacionDistritoE[Destino] += PoblacionUnidadGeografica[Unidades_Cambiadas[k]];
24      MedidaAreaE[Origen] -= AreaUnidadGeografica[Unidades_Cambiadas[k]];
25      MedidaAreaE[Destino] += AreaUnidadGeografica[Unidades_Cambiadas[k]];
26    }
27
28    MedidaPerimetroE[Origen] = MedidaPerimetroE[Destino] = 0;
29    for(k = 0; k < UnidadesPorConjunto; k++)
30    {
31      if(Distritos_Actuales[k] == Origen)
32      {
33        MedidaPerimetroE[Origen] += PerimetroFrontera[k][k];
34        o = Vecinos[k][60];
35        for(j = 0; j < o; j++)
36        {
37          p = Vecinos[k][j];
38          if(Distritos_Actuales[p] != Origen )
39                    MedidaPerimetroE[Origen] += PerimetroFrontera[k][p];
40        }
41
42      }
43      if(Distritos_Actuales[k] == Destino)
44      {
45        MedidaPerimetroE[Destino] += PerimetroFrontera[k][k];
46        o = Vecinos[k][60];
47        for(j = 0; j < o; j++)
48        {
49          p = Vecinos[k][j];
50          if(Distritos_Actuales[p] != Destino )
51                    MedidaPerimetroE[Destino] += PerimetroFrontera[k][p];
52        }
53      }
54
55    }
56
57    DesviacionPoblacionalDistritoE[Origen] = Desviacion_Poblacional(PoblacionDistritoE[Origen]);
```

```
58    DesviacionPoblacionalDistritoE[Destino] = Desviacion_Poblacional(PoblacionDistritoE[Destino]);
59    MedidaCompacidadE[Origen] = Compacidad(MedidaAreaE[Origen],MedidaPerimetroE[Origen]);
60    MedidaCompacidadE[Destino] = Compacidad(MedidaAreaE[Destino],MedidaPerimetroE[Destino]);
61    DesviacionPoblacional_Nueva = DesviacionPoblacionalE + DesviacionPoblacionalDistritoE[Origen] + DesviacionPoblacionalDistritoE[
          Destino];
62    Compacidad_Nueva = CompacidadTotalE + MedidaCompacidadE[Origen] + MedidaCompacidadE[Destino];
63    DesviacionPoblacional_Destino = DesviacionPoblacionalDistritoE[Destino];
64    CompacidadDistrito_Destino = MedidaCompacidadE[Destino];
65    PoblacionDistrito_Destino = PoblacionDistritoE[Destino];
66    AreaDistrito_Destino = MedidaAreaE[Destino];
67    PerimetroDistrito_Destino = MedidaPerimetroE[Destino];
68    DesviacionPoblacional_Origen = DesviacionPoblacionalDistritoE[Origen];
69    CompacidadDistrito_Origen = MedidaCompacidadE[Origen];
70    PoblacionDistrito_Origen = PoblacionDistritoE[Origen];
71    AreaDistrito_Origen = MedidaAreaE[Origen];
72    PerimetroDistrito_Origen = MedidaPerimetroE[Origen];
73
74    return(1);
75  }
```

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)

```
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 COLONIA DE ABEJAS ARTIFICIALES

Anexo : Función main()

```
int main()
{
  double z,seed1;
  double b1,u5,u6;
  int i,j, m6,k;
  int Semillas[100], 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];
  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 %d",&Fuentes_de_Alimento, &GeneracionFinal, &Semilla);
```

```
26    }
27    fclose(fp);
28
29    //SI EL USUARIO SELECCINO EL SEMILLERO SE LEEN LAS SEMILLAS QUE SE UTILIZARAN
30    if(Semilla == -1)
31    {
32      Numero_de_Semillas = 0;
33      direccion = "Sistema_de_visualizacion\\Insumos\\Semillero.txt";
34      fp = fopen(direccion, "r");
35      while((c=fgetc(fp))!=EOF && Numero_de_Semillas < 100)
36      {
37        fscanf(fp,"%d",&i);
38        Semillas[Numero_de_Semillas] = i;
39        Numero_de_Semillas++;
40      }
41      fclose(fp);
42    }
43    //EN CASO CONTRARIO SOLO SE REALIZARA UNA CORRIDA
44    else
45    {
46      Numero_de_Semillas = 1;
47      Semillas[0] = Semilla;
48    }
49
50    if(Numero_de_Semillas == 1)
51    {
52      printf("\n\n\n\t     Sistema para generar Zonas Electorales 2015\n\n\n");
53      printf("    Se realizara una corrida empleando los siguientes parametros:\n\n");
54      printf("    Numero de fuentes de alimento = %d\n\n",Fuentes_de_Alimento);
55      printf("    Numero de generaciones = %d\n\n",GeneracionFinal);
56      printf("    Semilla = %d\n\n",Semilla);
57    }
58
59    else
60    {
61      printf("\n\n\n\t     Sistema para generar Zonas Electorales 2015\n\n\n");
62      printf("    Se realizaran %d corridas, empleando los siguientes parametros:\n\n", Numero_de_Semillas);
63      printf("    Numero de fuentes de alimento = %d\n\n",Fuentes_de_Alimento);
64      printf("    Numero de generaciones = %d\n\n",GeneracionFinal);
65      printf("    Semilla = Se usaran los valores incluidos en el semillero\n\n");
66    }
67
68
69    // SE LEE EL NUMERO DE DISTRITOS ASIGNADOS A CADA CONJUNTO
70    ConjuntosTotales = 0;
71    direccion = "Sistema_de_visualizacion\\Insumos\\ConjuntosDistritos.txt";
72    fp = fopen(direccion, "r");
73    while((c=fgetc(fp))!=EOF)
74    {
75      fscanf(fp,"%d %d",&k,&i);
76      DistritosPorConjunto[i] = k;
77      NConjuntos = i;
78      ConjuntosTotales += k;
79    }
80    fclose(fp);
81
82
83    for(i=0; i<6500; i++)
84    Solucion_Hibrida[i] = 6500;
85
86    //SE REALIZAN TANTAS CORRIDAS COMO NUMERO DE SEMILLAS SE ENCUENTREN EN EL SEMILLERO
87    //O SOLO UNA SI EL USUARIO DA LA SEMILLA
88    for(Corrida = 0 ; Corrida < Numero_de_Semillas; Corrida++)
89    {
90      //SE CREA UN ARCHIVO PARA GUARDAR LOS COSTOS DE CADA CONJUNTO TERRITORIAL
91        if(Corrida == 0)
92      {
93        sprintf(dummy, "Sistema_de_visualizacion\\Resumen_Costos\\ABC_Costo_Por_Conjunto.csv");
94              fp = fopen(dummy, "w");
95              fprintf(fp,"Semilla %d,Costo Total, Equilibrio Poblacional,Compacidad\n", Semillas[Corrida]);
96              fclose(fp);
97        }
98        if(Corrida > 0)
99      {
100       sprintf(dummy, "Sistema_de_visualizacion\\Resumen_Costos\\ABC_Costo_Por_Conjunto.csv");
101             fp = fopen(dummy, "a");
102             fprintf(fp,"Semilla %d,Costo Total, Equilibrio Poblacional,Compacidad\n", Semillas[Corrida]);
103             fclose(fp);
104       }
105
106       if(Numero_de_Semillas > 1)
107             printf("\n\n\t Inicia la corrida = %d con la Semilla = %d\n\n",Corrida+1,Semillas[Corrida]);
108
109     Semilla = Semillas[Corrida];
110
111     for(i=0; i< 6500; i++)
112     DistritosFinales[i] = 6500;
113
114     //SE INICIA EL TIEMPO DE EJECUCION
```

```
115     clock_t start, end;
116     start = clock();
117
118     DistritosAcumulados = 0;
119
120     //SE INICIA EL PROCESO DE CONSTRUCCION DE DISTRITOS PARA CADA CONJUNTO
121     for(ConjuntoActual = 1; ConjuntoActual <= NConjuntos; ConjuntoActual++)
122     {
123       printf("\nConjunto territorial %d. ",ConjuntoActual);
124       for(i = 0; i < 6500; i++)
125               Conversion[i] = 6500;
126
127       //LA FUNCION Datos() LEE LA INFORMACION NECESARIA PARA CONSTRUIR LOS DISTRITOS
128       //POR EJEMPLO, COLINDANCIAS, AREA, POBLACION, ETC.
129       Datos(ConjuntoActual);
130       NDistritos = DistritosPorConjunto[ConjuntoActual];
131
132
133       //SE CONSTRUYEN NUEVAS FUENTES DE ALIMENTO Y SE EVALUA SU COSTO (PARA EL CONJUNTO EN CURSO)
134
135       if(DistritosPorConjunto[ConjuntoActual] == 1)
136       {
137         FuenteAlimento_Nueva(NDistritos);
138         for(i = 0; i < UnidadesPorConjunto; i++)
139                 MejoresDistritos[i] = Distrito[i];
140         goto Final;
141       }
142
143       for(j = 0; j < Fuentes_de_Alimento; j++)
144       {
145         FuenteAlimento_Nueva(NDistritos);
146         for(i = 0; i < UnidadesPorConjunto; i++)
147                 FuenteAlimento[j][i] = Distrito[i];
148         Costo_FuenteNueva(j);
149       }
150
151       for(i = 0; i < UnidadesPorConjunto; i++)
152               MejoresDistritos[i] = FuenteAlimento[0][i];
153       Menor_DesviacionPoblacional = DesviacionPoblacional_FuenteAlimento[0];
154       Menor_Compacidad = Compacidad_FuenteAlimento[0];
155
156       for(i = 0; i < Fuentes_de_Alimento; i++)
157               GeneracionesSinMejora[i] = 0;
158
159       //INICIA PROCESO DE MEJORA
160       Generacion = 0;
161       while(Generacion < GeneracionFinal)
162       {
163         //PARA CADA FUENTE DE ALIMENTO SE LLAMA UNA VEZ A LA ABEJA EMPLEADA
164         for(i = 0; i < Fuentes_de_Alimento; i++)
165         {
166
167           //SI LA NUEVA FUENTE DE ALIMENTO ES MEJOR SE ACEPTA Y SE REGRESA UN VALOR DE 0
168           j = AbejaEmpleada(i);
169
170           //SI HUBO MEJORA SE ACTUALIZA LA CALIDAD DE LA FUENTE DE ALIMENTO
171           if(j == 0)
172             GeneracionesSinMejora[i] = 0;
173
174           else
175                       GeneracionesSinMejora[i]++;
176
177           b1 = Menor_DesviacionPoblacional + Menor_Compacidad;
178           u5 = DesviacionPoblacional_FuenteAlimento[i] + Compacidad_FuenteAlimento[i];
179
180           //SI LA NUEVA FUENTE DE ALIMENTO ES LA MEJOR CONOCIDA SE GUARDA EN MEMORIA
181           if(u5 < b1)
182           {
183             Menor_DesviacionPoblacional = DesviacionPoblacional_FuenteAlimento[i];
184             Menor_Compacidad = Compacidad_FuenteAlimento[i];
185             for(k = 0; k < UnidadesPorConjunto; k++)
186             {
187               MejoresDistritos[k] = FuenteAlimento[i][k];
188             }
189           }
190
191           //SI LA FUENTE DE ALIMENTO NO HA SIDO MEJORADA EN 100 GENERARIONES CONSECUTIVAS SE REEMPLAZA
192           if(GeneracionesSinMejora[i] >= 100)
193           {
194             GeneracionesSinMejora[i] = 0;
195             FuenteAlimento_Nueva(NDistritos);
196             for(j = 0; j < UnidadesPorConjunto; j++)
197                             FuenteAlimento[i][j] = Distrito[j];
198             Costo_FuenteNueva(i);
199           }
200         }
201
202         //SE CALCULA LA CALIDAD DE CADA FUENTE DE ALIMENTO
203         //LA SUMA DE SUS CALIDADES ES LA CALIDAD TOTAL
```

```
204         Calidad_Total = 0;
205         for(i = 0; i < Fuentes_de_Alimento; i++)
206                 {
207                     Calidad_FuenteAlimento[i] = 1 /(1 + Costo_FuenteAlimento[i]);
208                     Calidad_Total += Calidad_FuenteAlimento[i];
209                 }
210
211         //LA ABEJA OBSERVADORA ES LLAMADA TANTAS VECES COMO FUENTES DE ALIMENTO SE TIENE
212         //PERO VISITA CON MAS PROBABILIDAD A LAS FUENTES DE ALIMENTO CON MEJOR CALIDAD
213         for(j = 0; j < Fuentes_de_Alimento; j++)
214         {
215           b1 = SiguienteAleatorioReal0y1(& Semilla);
216           u5 = Calidad_FuenteAlimento[0] / Calidad_Total;
217           i = 0;
218           if(b1 < u5)
219                       i = 0;
220           else
221           {
222             while(b1 > u5)
223             {
224               i++;
225               u5 += Calidad_FuenteAlimento[i] / Calidad_Total;
226             }
227           }
228           //SI LA FUENTE DE ALIMENTO i ES MEJORADA SU CONTADOR SE REINICIA CON UN VALOR DE 0
229           m6 = AbejaObservadora(i);
230           if(m6 == 0)
231                       GeneracionesSinMejora[i] = 0;
232           b1 = Menor_DesviacionPoblacional + Menor_Compacidad;
233           u5 = DesviacionPoblacional_FuenteAlimento[i] + Compacidad_FuenteAlimento[i];
234
235           //SI LA NUEVA FUENTE DE ALIMENTO ES LA MEJOR CONOCIDA SE GUARDA EN MEMORIA
236           if(u5 < b1)
237           {
238             Menor_DesviacionPoblacional = DesviacionPoblacional_FuenteAlimento[i];
239             Menor_Compacidad = Compacidad_FuenteAlimento[i];
240             for(k = 0; k < UnidadesPorConjunto; k++)
241               MejoresDistritos[k] = FuenteAlimento[i][k];
242           }
243         }
244         Generacion++;
245
246       }
247       //TERMINA EL PROCESO DE MEJORA DE UN CONJUNTO TERRITORIAL
248
249       //SE GUARDAN LOS MEJORES DISTRITOS CONSTRUIDOS EN LA VARIABLE DistritosFinales
250       //AL TEMINAR CADA CORRIDA EL ARREGLO DistritosFinales TENDRA EL ESCENARIO COMPLETO
251 Final:
252
253       for(i=0; i<UnidadesPorConjunto; i++)
254       {
255         for(j=0; j<6500; j++)
256         {
257           if(Conversion[j] == i)
258                       DistritosFinales[j] = MejoresDistritos[i] + DistritosAcumulados;
259         }
260
261       }
262
263       for(i=0; i< UnidadesPorConjunto; i++)
264       Distrito[i] = MejoresDistritos[i];
265       Evalua_Solucion();
266
267       for(i = 0; i < NDistritos; i++)
268       {
269         MejorDesviacionPoblacional[i + DistritosAcumulados] = DesviacionPoblacionalDistrito[i];
270         MejorCompacidad[i + DistritosAcumulados] = CompacidadDistrito[i];
271       }
272
273       printf(" Costo final %f = %f  +  0.5 * %f\n",DesviacionPoblacional_Nueva + Compacidad_Nueva,DesviacionPoblacional_Nueva, 2 *
              Compacidad_Nueva);
274
275       //SE IMPRIME EL COSTO DE CADA CONJUNTO TERRITORIAL
276             sprintf(dummy, "Sistema_de_visualizacion\\Resumen_Costos\\ABC_Costo_Por_Conjunto.csv");
277             fp = fopen(dummy, "a");
278             fprintf(fp,"Conjunto %d,%f,%f,%f\n", ConjuntoActual, DesviacionPoblacional_Nueva + Compacidad_Nueva,
                    DesviacionPoblacional_Nueva, Compacidad_Nueva);
279             fclose(fp);
280
281
282       //SE CREA O SE ACTUALIZA EL Escenario_Hibrido
283       if(Corrida == 0 && Numero_de_Semillas > 1)
284       {
285         for(i=0; i<UnidadesPorConjunto; i++)
286         {
287           for(j=0; j<6500; j++)
288           {
289             if(Conversion[j] == i)
290                         Solucion_Hibrida[j] = MejoresDistritos[i] + DistritosAcumulados;
```

```
291             }
292           }
293           for(i = 0; i < NDistritos; i++)
294           {
295             DesviacionPoblacional_Hibrida[i + DistritosAcumulados] = MejorDesviacionPoblacional[i + DistritosAcumulados];
296             Compacidad_Hibrida[i + DistritosAcumulados] = MejorCompacidad[i + DistritosAcumulados];
297           }
298         }
299         if(Corrida >= 1)
300         {
301           u5 = u6 = 0;
302                 for(i = 0; i < NDistritos; i++)
303           {
304             u5 += DesviacionPoblacional_Hibrida[i + DistritosAcumulados] + Compacidad_Hibrida[i + DistritosAcumulados];
305             u6 += MejorDesviacionPoblacional[i + DistritosAcumulados] + MejorCompacidad[i + DistritosAcumulados];
306           }
307           if(u5 > u6)
308           {
309             for(i=0; i<UnidadesPorConjunto; i++)
310             {
311               for(j=0; j<6500; j++)
312               {
313                 if(Conversion[j] == i)
314                                 Solucion_Hibrida[j] = MejoresDistritos[i] + DistritosAcumulados;
315               }
316             }
317             for(i = 0; i < NDistritos; i++)
318             {
319               DesviacionPoblacional_Hibrida[i + DistritosAcumulados] = MejorDesviacionPoblacional[i + DistritosAcumulados];
320               Compacidad_Hibrida[i + DistritosAcumulados] = MejorCompacidad[i + DistritosAcumulados];
321             }
322           }
323         }
324
325         DistritosAcumulados += NDistritos;
326       }
327       //TERMINA EL PROCESO DE MEJORA DEL ESCENARIO COMPLETO
328
329       end = clock();
330       seed1 = end - start;
331       z = seed1 / CLOCKS_PER_SEC;
332       seed1 = seed1 / (60 * CLOCKS_PER_SEC);
333       i = (int) seed1;
334       seed1 = z - (60 * i);
335       z = i / 60;
336       printf("\n\nEL TIEMPO DE EJECUCION FUE DE: %d MINUTOS %f SEGUNDOS\n\n",i,seed1);
337
338
339       for(i=0; i< 6500; i++)
340             MejoresDistritos[i] = DistritosFinales[i];
341       //SE OBTIENE EL COSTO TOTAL DE LA SOLUCION CONSTRUIDA
342       Costo_Nueva = DesviacionPoblacional_Nueva = Compacidad_Nueva = 0;
343       for(i=0; i<ConjuntosTotales; i++)
344       {
345         Costo_Nueva += MejorDesviacionPoblacional[i] + MejorCompacidad[i];
346         DesviacionPoblacional_Nueva += MejorDesviacionPoblacional[i];
347         Compacidad_Nueva += MejorCompacidad[i];
348       }
349
350       //SE IMPRIME EL COSTO TOTAL DEL ESCENARIO ACTUAL
351           sprintf(dummy, "Sistema_de_visualizacion\\Resumen_Costos\\ABC_Costo_Por_Conjunto.csv");
352           fp = fopen(dummy, "a");
353           fprintf(fp,"Costo Total,%f,%f,%f\n",Costo_Nueva, DesviacionPoblacional_Nueva, Compacidad_Nueva);
354           fclose(fp);
355
356       //SE IMPRIME EN ARCHIVO DE TEXTO LA SOLUCION FINAL
357       sprintf(dummy, "Sistema_de_visualizacion\\ABC_Escenario_%f_%d.csv", DesviacionPoblacional_Nueva + Compacidad_Nueva, Semillas[
              Corrida]);
358       fp = fopen(dummy, "w");
359       fprintf(fp,"Seccion,DISTRITO\n");
360       for(i = 0; i < 6500; i++)
361       if(MejoresDistritos[i] < 6500)
362             fprintf(fp,"%d,%d\n",i,MejoresDistritos[i] + 1);
363       fclose(fp);
364     }
365     //TERMINA LA CORRIDA ACTUAL
366
367     //SE IMPRIME LA SOLUCION HIBRIDA CONSTRUIDA CON LOS ESCENARIOS OBTENIDOS
368     //SOLO SI SE EMPLEO EL SEMILLERO
369     if(Numero_de_Semillas > 1)
370     {
371       direccion = "Sistema_de_visualizacion\\Escenario_Hibrido_ABC.csv";
372       fp = fopen(direccion, "w");
373       fprintf(fp,"Seccion,DISTRITO\n");
374       for(i = 0; i < 6500; i++)
375       if(Solucion_Hibrida[i] < 6500)
376             fprintf(fp,"%d,%d\n",i,Solucion_Hibrida[i] + 1);
377       fclose(fp);
378     }
```

```
379
380   printf("\t Se han completado las corridas solicitadas. \n\n \t Presione una tecla para conluir el proceso.");
381   getchar();
382
383   return(1);
384
385 }
```

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 %f %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 %f",&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)

```
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, i2, j2;
  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++)
    {
      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)
```

```
36              {
37                Candidatos[n] = i;
38                n++;
39                n1 = 6500;
40              }
41            }
42            j++;
43          }
44        }
45      }
46
47      //SE ELIGE UNA UNIDAD PARA SER CAMBIADA DE DISTRITO
48        //SE ELIGE CON UNA PROBABILIDAD 1/n
49      i = SiguienteAleatorioEnteroModN(& Semilla, n);
50        Unidad_Elegida = Candidatos[i];
51
52
53      //SE DETERMINAN LOS DISTRITOS VECINOS DE Unidad_Elegida
54      n1 = 0;
55      for(i=0; i<Vecinos[Unidad_Elegida][60]; i++)
56      {
57        if(Distrito[Vecinos[Unidad_Elegida][i]] != Origen)
58        {
59          if(n1 > 0)
60          {
61            j = 0;
62            for(n = 0; n <n1; n++)
63            {
64              if(destinosE[n] != Distrito[Vecinos[Unidad_Elegida][i]])
65                          j++;
66            }
67            if(j == n1)
68            {
69              destinosE[n1] = Distrito[Vecinos[Unidad_Elegida][i]];
70              n1++;
71            }
72          }
73          if(n1 == 0)
74          {
75            destinosE[n1] = Distrito[Vecinos[Unidad_Elegida][i]];
76            n1++;
77          }
78        }
79      }
80
81      //SE ELIGE UN DISTRITO DESTINO PARA Unidad_Elegida
82      //SE ELIGE CON UNA PROBABILIDAD 1/n
83      if(n1 == 1)
84            Destino = 0;
85        else
86            Destino = SiguienteAleatorioEnteroModN(& Semilla, n1);
87
88      Destino = destinosE[Destino];
89      Distrito[Unidad_Elegida] = Destino;
90
91      //SE REVISA SI SE PROVOCO ALGUNA DISCONEXION
92      j = RevisaConexidad_Empleada(Origen, Destino);
93
94      //SE EVALUA EL COSTO DEL NUEVO ESCENARIO
95      Evalua_Solucion();
96
97      b1 = DesviacionPoblacional_FuenteAlimento[AB] + Compacidad_FuenteAlimento[AB];
98      u5 = DesviacionPoblacional_Nueva + Compacidad_Nueva;
99
100     //LA NUEVA SOLUCION ES ACEPTADA SI MEJORA EL COSTO DE FuenteAlimento[AB][]
101     if(u5 < b1)
102     {
103       DesviacionPoblacional_FuenteAlimento[AB] = DesviacionPoblacional_Nueva;
104       Compacidad_FuenteAlimento[AB] = Compacidad_Nueva;
105       Costo_FuenteAlimento[AB] = Costo_Nueva;
106       for(i = 0; i < UnidadesPorConjunto; i++)
107       {
108         FuenteAlimento[AB][i] = Distrito[i];
109       }
110       Evalua_Solucion();
111       return(0);
112     }
113
114     return(1);
115
116   }
```

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];
```

```
215   u5 = DesviacionPoblacional_Nueva + Compacidad_Nueva;
216
217   //SI LA NUEVA SOLUCION TIENE MEJOR CALIDAD SE ACEPTA Y REEMPLAZA A LA FUENTE DE ALIMENTO AB
218   if(u5 < b1 && u5 != (DesviacionPoblacional_FuenteAlimento[AB1] + Compacidad_FuenteAlimento[AB1]))
219   {
220     DesviacionPoblacional_FuenteAlimento[AB] = DesviacionPoblacional_Nueva;
221     Compacidad_FuenteAlimento[AB] = Compacidad_Nueva;
222     Costo_FuenteAlimento[AB] = Costo_Nueva;
223     for(i = 0; i < UnidadesPorConjunto; i++)
224     {
225       FuenteAlimento[AB][i] = Distrito[i];
226     }
227     return(0);
228   }
229   return(1);
230 }
```

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;
16      if(Distrito[m] == DistritoAnalizado)
17      {
18        Lista1[suma] = m;
19        suma++;
20      }
21    }
22
23    if ( suma <= 1)
24      return(0);
25
26    //SE REVISA EL NUMERO DE COMPONENTES CONEXAS EN EL DISTRITO
27    //SI EL NUMERO DE COMPONENTES CONEXAS ES IGUAL A 1 NO SE HA PERDIDO LA CONEXIDAD
28    //SI EL NUMERO DE COMPONENTES CONEXAS ES MAYOR O IGUAL A 2 SE DEBE REPARAR LA CONEXIDAD
29    for(i=0; i< suma; i++)
30    {
31      n = 0;
32      if(Contactado[Lista1[i]] == 0)
33      {
34        Representante[N] = Lista1[i];
35        Lista2[n] = Lista1[i];
36        Contactado[Lista2[n]] = 1;
37        Componente[N]++;
38        n++;
39        for(j=0; j < n; j++)
40        {
41          for(m=0; m < Vecinos[Lista2[j]][60]; m++)
42          {
43            if(Distrito[Vecinos[Lista2[j]][m]] == DistritoAnalizado && Contactado[Vecinos[Lista2[j]][m]] == 0)
44            {
45              Lista2[n] = Vecinos[Lista2[j]][m];
46              Contactado[Vecinos[Lista2[j]][m]] = 1;
47              Componente[N]++;
48              n++;
49            }
50          }
51        }
52        N++;
53      }
54    }
55
56    if( N == 1)
57          return(0);
58
59    //SE DETERMINA CUAL ES LA COMPONENTE CON MAYOR NUMERO DE UNIDADES GEOGRAFICAS
60    //PARA DEJARLA COMO EL DISTRITO Origen
61    n = 0;
62    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;
```

```
60              n++;
61            }
62          }
63        }
64        N++;
65      }
66    }
67
68    if( N == 1)
69          return(0);
70
71      //LAS UNIDADES SON ENVIADAS A DISTRITOS VECINOS, EXCEPTO LAS UBICADAS EN Componente
72    for(i = 0; i < N; i++)
73    {
74      if(i != Componente)
75              ReparaConexidad_Observadora(DistritoOrigen,Representante[i]);
76    }
77    return(0);
78  }
```

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,n0;
6     int Lista2[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    n0 = 0;
19    Lista2[n0] = UnidadOrigen;
20    Contactado[Lista2[n0]] = 1;
21    n0++;
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    mm = 0;
26    for(j=0; j < n0;j++)
27    {
28      for(m=0; m < Vecinos[Lista2[j]][60]; m++)
29      {
30        if(Distrito[Vecinos[Lista2[j]][m]] == DistritoOrigen && Contactado[Vecinos[Lista2[j]][m]] == 0)
31        {
32          Lista2[n0] = Vecinos[Lista2[j]][m];
33          Contactado[Vecinos[Lista2[j]][m]] = 1;
34          n0++;
35        }
36
37        if(Distrito[Vecinos[Lista2[j]][m]] != DistritoOrigen && Destinos[Distrito[Vecinos[Lista2[j]][m]]] == 0)
38        {
39          Destinos[Distrito[Vecinos[Lista2[j]][m]]] = 1;
40          Destinos2[mm] = Distrito[Vecinos[Lista2[j]][m]];
41          mm++;
42        }
43      }
44    }
45
46      //SE ELIGE UN DESTINO PARA LAS UNIDADES GEOGRAFICAS EN LA COMPONENTE CONEXA
47    if(mm > 1)
48          mm = SiguienteAleatorioEnteroModN(& Semilla, mm);
49    else
50          mm = 0;
51    m  = Destinos2[mm];
52
53    //LAS UNIDADES SON ENVIADAS A LA COMPONENTE ELEGIDA
54    for(i = 0; i < UnidadesPorConjunto; i++)
55    {
56      if(Contactado[i] == 1)
57              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);
}
```

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.

Instituto Nacional Electoral

Modelo Matemático y Algoritmos

Índice general

Índice de tablas

Capítulo 1

Modelo Matemático

Un Modelo de Optimización Multiobjetivo tiene dos componentes:

- Una Función Objetivo.
- Un Conjunto de Restricciones.

La tarea de la función objetivo, dentro del modelo de optimización, es calificar la calidad de las soluciones (o escenarios) que cumplen con todas las restricciones (soluciones factibles). Esta calificación se basa en dos criterios; el equilibrio poblacional en cada uno de los distritos y la forma de los distritos de acuerdo a su compacidad geométrica. La solución del modelo es cuando se obtiene aquel escenario cuyo valor objetivo es el menor posible. Este escenario puede ser único o puede que exista más de uno.

Otra tarea muy importante de la función objetivo es la de guiar a los algoritmos de búsqueda a ir mejorando las soluciones que se van obteniendo de tal manera que al final encuentren la mejor o una de las mejores soluciones al modelo.

1.1. Función Multiobjetivo

La función multiobjetivo debe conducir a los algoritmos a encontrar distritos con la menor desviación poblacional y que las formas geométricas de los distritos sean lo más cercano a polígonos regulares pero además los valores de estos dos objetivos deben ser comparables para que sean "competitivos" dentro de esta función.

1.1.1. Objetivo Poblacional

La desviación del promedio poblacional de cada distrito electoral, debe estar a $\pm 15\,\%$, esto significa que si P_D = Población del distrito y P_M = Población media estatal, entonces la población de cualquier distrito está dentro de este criterio si cumple con la siguiente desigualdad:

$$P_M - 0.15P_M \leq P_D \leq P_M + 0.15P_M$$

Haciendo algunas operaciones algebraicas se llega a lo siguiente:

$$-0.15P_M \leq P_D - P_M \leq 0.15P_M$$

$$-1 \leq \frac{1}{0.15}\left(\frac{P_D}{P_M} - 1\right) \leq 1$$

o equivalentemente

$$0 \leq \left|\frac{1}{0.15}\left(\frac{P_D}{P_M} - 1\right)\right| \leq 1$$

De esta forma, para penalizar de forma más estricta a distritos fuera de rango, se toma un valor de a mayor que 1, como se muestra en la siguiente expresión:

$$\left|\frac{1 - (\frac{P_D}{P_M})}{0.15}\right|^a \leq 1 \tag{1.1}$$

Se probaron algunos valores de $a \geq 1$ tales como: 1, 1.2, 1.5, 1.8 y 2 y después de varias experimentaciones se llegó a que el valor apropiado de a es 2, ya que para valores menores de dos, en algunos ejercicios de prueba, se obtuvieron distritos fuera de rango.

En virtud de lo anterior, la fórmula de equilibrio poblacional, para un distrito D, es igual a:

$$C_1(D) = \left(\frac{1 - \left(\frac{P_D}{P_M}\right)}{0.15}\right)^2 \tag{1.2}$$

Donde:
P_D = Población del distrito
P_M = Población media estatal

0.15 = Desviación máxima permitida de la media estatal.

1.1.2. Objetivo Compacidad Geométrica

La compacidad geométrica de cada distrito electoral, se puede entender como la situación en la que el perímetro de los distritos adquiera una forma geométrica lo más cercana a **un polígono regular**.
Este criterio se incluye en la función multiobjetivo. La función que evalúa la compacidad se definirá como un cociente del perímetro del distrito entre la raíz cuadrada del área del mismo, multiplicada por un coeficiente definido experimentalmente para normalizarla con respecto al objetivo del equilibrio poblacional.

En la literatura sobre estudios de diseño de zonas, y en particular sobre distritación electoral, se proponen muchas medidas de compacidad tales como envolventes conexas, cuadriculado, comparación de círculos inscritos, etc.; pero una fórmula para el cálculo de la compacidad geométrica que da muy buenos resultados y su cálculo numérico requiere de pocas operaciones es:

$$k * \left(\frac{Perímetro\ del\ distrito}{\sqrt{Área\ del\ distrito}} \right) \tag{1.3}$$

Donde k es una constante que hace la expresión 1.3 igual a uno dependiendo del polígono regular que se quiera. Por ejemplo, si deseamos que para un triángulo equilátero la expresión 1.3 sea igual a uno, $k = 0.21935$; para un cuadrado, $k = 0.25$, etc. En general para un polígono regular de m lados la expresión 1.3 vale uno, si el valor de k es igual a:

$$k = \frac{1}{2\sqrt{m \tan\left(\frac{\pi}{m}\right)}} \tag{1.4}$$

En la Tabla 1.1 se calculan algunos valores de k usando la expresión 1.4, tomando polígonos regulares con m lados, para $m = 3, 4, ..., 10, ...\infty$.

Por lo expresado anteriormente, para el cálculo de la compacidad geométrica de un distrito D se propone la expresión

m	k
3	0.21935
4	0.25000
5	0.26233
6	0.26864
7	0.27233
8	0.27467
9	0.27626
10	0.27738
—	—
∞	0.28209

TABLA 1.1: Valores de k calculados con la expresión 1.4 tomando polígonos regulares de m lados.

$$C_2(D) = \left(\left(\frac{Perímetro\ del\ distrito}{\sqrt{Área\ del\ distrito}} * 0.25 \right) - 1 \right) * 0.5 \tag{1.5}$$

Donde 0.25 corresponde al coeficiente que hace la expresión 1.3 igual a 1 si la forma del distrito corresponde a un cuadrado. Se eligió un cuadrado porque dentro de los polígonos regulares, los únicos capaces de formar una teselación [1] son: el triángulo equilátero, el cuadrado y el hexágono.

Por lo que se consideró restar a la expresión anterior la cantidad 1 para permitir que la mejor compacidad se encuentre cercana a cero y sea exactamente igual a cero cuando se forme un distrito de forma cuadrada.

1.1.3. Función Objetivo del Modelo Matemático

La función multiobjetivo que se propone para la distritación electoral será la suma ponderada del equilibrio poblacional y la compacidad geométrica dada por la siguiente expresión:

$$f(E) = \sum_{i=1}^{n} C_1(D_i) + 0.5 \sum_{i=1}^{n} C_2(D_i) \tag{1.6}$$

[1] Un teselado es una regularidad o patrón de figuras que cubre o pavimenta completamente una superficie plana que cumple con dos requisitos: 1) Que no queden huecos, 2) Que no se superpongan las figuras.

Donde, en la expresión 1.6, $E = (D_1, D_2, \ldots, D_n)$ es un escenario o plan distrital y el factor 0.5 refleja la importancia relativa de la compacidad geométrica con respecto al equilibrio poblacional. Se asigna al equilibrio poblacional un peso del doble con respecto a la compacidad geométrica, debido a la importancia relativa de ambos objetivos.

1.2. Restricciones del Modelo

Las restricciones que el modelo considera son las siguientes:

1. El estado se dividirá en n distritos electorales.
2. Los distritos serán continuos y contiguos.
3. Cada distrito debe tener una desviación poblacional máxima dentro del 15 por ciento de la media estatal.
4. Integridad municipal, al considerar a los municipios que en forma integral se agruparán para formar un distrito.
5. Tiempos de traslado. Se calcula un tiempo de traslado de corte. Dos municipios se considerarán como no vecinos, si el tiempo de traslado entre ellos es mayor que el tiempo de corte.
6. Todo aquel municipio que pueda ser separado en un número entero de distritos dentro del 15 por ciento de desviación poblacional le serán asignados ese número de distritos y serán divididos hacia su interior.
7. Tipología de municipios.

Capítulo 2

Recocido Simulado.

Recocido simulado es una de las técnicas heurísticas más conocidas, que por su simplicidad y buenos resultados en numerosos problemas, se ha convertido en una herramienta muy popular, con aplicaciones en diferentes áreas de optimización. El concepto fue introducido en el campo de la optimización combinatoria a inicios de la década de los 80 por Kirkpatrick [2] y Cerny [6]. Esta heurística, se inspira en una analogía entre el proceso de recocido de sólidos y la forma en que se resuelven problemas de optimización combinatoria. Dicha analogía resulta importante para comprender la forma en que trabaja este algoritmo.

El recocido de sólidos es un proceso de tratamiento térmico que se aplica a varios materiales como el vidrio y ciertos metales y aleaciones para hacerlos menos quebradizos y más resistentes a la fractura. El objetivo de este proceso es minimizar la energía interna de la estructura atómica del material y eliminar posibles tensiones internas provocadas en las etapas anteriores de su procesado. Para lograrlo, los metales ferrosos y el vidrio se recuecen calentándolos a alta temperatura y enfriándolos lentamente. Cada vez que se baja la temperatura, las partículas se reacomodan en estados de más baja energía hasta que se obtiene un sólido con partículas acomodadas conforme a una estructura con energía mínima.

El algoritmo de recocido simulado está basado en técnicas de Monte Carlo y genera una sucesión de estados del sólido de la siguiente manera. Dado un estado actual i del sólido con energía S_i, entonces el siguiente estado j es generado al aplicar una pequeña perturbación al estado actual. La temperatura del nuevo estado es S_j. Si la diferencia de energía $S_i - S_j$, es menor o igual que cero, el estado j es aceptado

como estado actual. Si la diferencia es mayor que cero, el estado j es aceptado con una probabilidad dada por:

$$\exp\left(\frac{S_i - S_j}{k_B T}\right) \tag{2.1}$$

Donde T denota la temperatura a la cual se encuentra el sólido y k_B es una constante física llamada la constante de Boltzmann. Este criterio de aceptación es conocido como el criterio de Metrópolis. Si el decremento de la temperatura es suficientemente lento, el sólido alcanzará un equilibrio térmico en cada temperatura.

2.1. Algoritmo de Recocido Simulado

La técnica de recocido simulado, se inspiró en el hecho de que el algoritmo de Metrópolis puede ser utilizado para generar soluciones a problemas de optimización combinatoria si se hacen las siguientes consideraciones:

- Las soluciones del problema de optimización son equivalentes a los estados del sólido.
- El costo de una solución, denotado por $f(E_i)$, es equivalente a la energía de cada estado.
- La temperatura será sustituida por un parámetro de control que regulará la probabilidad de aceptación de nuevas soluciones.

El algoritmo de Recocido Simulado comienza con una solución inicial y una temperatura inicial T_0. Se recomienda que al inicio del algoritmo la temperatura sea suficientemente alta para permitir todo, o casi todo, movimiento, es decir, que la probabilidad de pasar de la solución actual E_i a la solución vecina E_j sea muy alta, sin importar la diferencia entre los costos de ambas soluciones, $f(E_i) - f(E_j)$.

En cada iteración se genera una solución vecina de manera aleatoria, si la nueva solución mejora el valor de la función objetivo con respecto a la solución actual, esta última es reemplazada. Cuando la nueva solución no mejora el valor de la función

objetivo, se puede aceptar el cambio de la solución actual con cierta probabilidad dada por:

$$\exp\left(\frac{f(E_i) - f(E_j)}{T}\right) \tag{2.2}$$

Donde, $f(E_i)$ es el costo de la solución actual, $f(E_j)$ es el costo de la solución vecina y T es la temperatura del proceso. Conforme el algoritmo avanza, el valor de la temperatura disminuye mediante un coeficiente de enfriamiento, $0 < \alpha < 1$, pero cada valor de T se mantiene constante durante L iteraciones, para permitir que el algoritmo explore distintas soluciones con la misma probabilidad de aceptación.

Debe observarse que al inicio, cuando la temperatura es alta, se tiene una mayor probabilidad de aceptar soluciones de menor calidad, lo cual permite la exploración del espacio de soluciones y evita la convergencia prematura a mínimos locales. Sin embargo, conforme el valor de la temperatura disminuye, el algoritmo se hace más selectivo y difícilmente acepta soluciones de menor calidad, iniciando una búsqueda que lo guía hacia un mínimo local. Finalmente, el algoritmo se detiene cuando la temperatura alcanza un valor límite, T_f, y devuelve la mejor solución encontrada.

Los parámetros de temperatura inicial T_0, coeficiente de enfriamiento α, temperatura final T_f y número de soluciones visitadas en cada temperatura L, son conocidos como *programa de enfriamiento* y juegan un papel importante en el desarrollo del algoritmo. Si se utilizan valores demasiado grandes el tiempo de ejecución podría ser excesivo, mientras que valores demasiado pequeños podrían provocar una convergencia prematura a un mínimo local. Para mayores detalles sobre esta técnica ver [1].

Sea T_k denote el valor de la temperatura y L_k el número de transiciones generadas en la k-ésima iteración del algoritmo de Metropolis. El algoritmo de Recocido Simulado puede describirse en pseudo-código como se muestra en el Algoritmo 1.

El algoritmo de Recocido Simulado comienza llamando a un procedimiento de inicialización donde se definen la solución inicial, parámetro de control inicial y el número inicial de generaciones necesarias para alcanzar el equilibrio térmico para la temperatura inicial. La parte medular del algoritmo consta de dos ciclos. El externo **Repite**...**hasta** y el interno **Para**...**finpara**. El ciclo interno mantiene fija la temperatura hasta que se generan L_k soluciones y se acepta o se rechaza la solución generada conforme el criterio de aceptación ya discutido. El ciclo externo disminuye el valor de la temperatura mediante el procedimiento CALCULA-CONTROL y

Algoritmo 1: Algoritmo de Recocido Simulado en pseudo-código

```
1  INICIALIZA (E_{i_inicial}, T_0, L_0)
2  k := 0
3  E_i := E_{i_inicial}
4  Repite
5  |  Para l := 1 a L_k hacer
6  |  |  GENERA (E_j vecino de E_i)
7  |  |  si f(E_j) ≤ f(E_i) entonces
8  |  |  |  E_i := E_j
9  |  |  fin
10 |  |  en otro caso
11 |  |  |  si exp((f(E_i)−f(E_j))/T_k) > número aleatorio en [0,1) entonces
12 |  |  |  |  E_i := E_j
13 |  |  |  fin
14 |  |  fin
15 |  fin
16 |  k := k + 1
17 |  CALCULA-LONGITUD (L_k)
18 |  CALCULA-CONTROL (T_k)
19 hasta Cumplir criterio de paro;
20 Termina algoritmo
```

calcula el número de soluciones a generar para alcanzar equilibrio térmico mediante el procedimiento CALCULA-LONGITUD. Este ciclo finaliza cuando la condición de paro se cumple.

Un rasgo característico del algoritmo de Recocido Simulado es que, además de aceptar mejoras en el costo, también acepta soluciones peores en costo. Inicialmente, para valores grandes de T, puede aceptar grandes soluciones deterioradas; cuando T decrece, únicamente pequeñas desviaciones serán aceptadas y finalmente, cuando el valor de T se aproxima a cero, no se aceptarán desviaciones. Este hecho significa que el algoritmo de Recocido Simulado tiene la capacidad de escapar de mínimos locales.

Note que la probabilidad de aceptar desviaciones está implementada al comparar el valor de $\exp(f(E_i) - f(E_j))/T$ con un número aleatorio generado de una distribución uniforme en el intervalo [0,1). Además, debe ser obvio que la velocidad de convergencia del algoritmo está determinada al escoger los parámetros L_k y T_k, k = 0,1,.... Si los valores T_k decrecen rápidamente o los valores de L_k no son grandes, se tendrá una convergencia más rápida que cuando los valores de T_k decrecen lentamente o los valores de L_k son grandes.

2.2. Aplicación a distritos electorales

La creación de distritos electorales, mediante recocido simulado, se realiza en dos etapas. Primero se crea una solución inicial formada por r distritos conexos, donde r es el número de distritos indicado para cada conjunto territorial. Posteriormente, se realiza un proceso de mejora destinado a explorar diferentes soluciones, a partir de la solución inicial, de tal forma que al final del proceso se obtenga una solución de buena calidad. Estas etapas se describen con más detalle en las siguientes secciones.

2.2.1. Solución inicial

El primer paso del algoritmo consiste en construir una solución inicial con distritos conexos, para lo cual selecciona de manera aleatoria r Unidades Geográficas (UG) que asigna a distritos diferentes, y las marca como UG no disponibles. Después se realizan las siguientes instrucciones hasta que todas las UG están marcadas como no disponibles:

- Elegir un distrito.
- Generar una lista con las UG disponibles que colindan con el distrito seleccionado.
- Seleccionar al azar una UG de la lista.
- Incluir en el distrito la UG elegida y marcarla como no disponible.

De esta forma se obtiene una solución con r distritos conexos ajenos que incluyen a todas las UG, cuya calidad no necesariamente es buena pero que podrá mejorarse en el proceso de búsqueda.

2.2.2. Solución vecina

El algoritmo de RS inicia con la construcción de una solución con distritos conexos, como ya se explicó en la sección 2.2.1, y durante el proceso de búsqueda y mejora, se garantizará que las nuevas soluciones conserven esta característica.

Para generar una solución vecina se elige de manera aleatoria un distrito, D, y se genera una lista con las UG que pueden ser enviadas a un distrito contiguo. Por lo tanto, en la lista se incluyen las UG que se encuentran en colindancia con otros distritos. Se selecciona aleatoriamente una UG, i, de la lista, y se cambia al distrito con el cual colinda, $D^{'}$; en caso de que colinde con dos o más distritos se hace una elección aleatoria. En caso de que el distrito elegido inicialmente esté formado por una sola UG se evita el cambio, ya que esto implicaría una disminución en el número de distritos.

Cuando el cambio de la UG provoca una disconexión en D se recupera la conexidad de la siguiente forma:

1. Se identifican las diferentes componentes conexas en que se dividió D.
2. Se cuenta el número de UG que forman a cada componente conexa.
3. La componente conexa con el mayor número de UG es considerada como el distrito original, D.
4. El resto de las componentes conexas es enviado a $D^{'}$ junto con i.

Siguiendo este procedimiento, cada solución vecina es una solución con distritos conexos que se diferencia de la anterior por la ubicación de un conjunto de UG. Se debe mencionar que es importante elegir de manera aleatoria las UG que son cambiadas para evitar que el algoritmo favorezca algunas soluciones y para aumentar las posibilidades de visitar un mínimo global.

Capítulo 3

Colonia de Abejas Artificiales

Colonia de Abejas Artificiales (ABC por sus siglas en inglés) es una técnica metaheurística bio-inspirada, propuesta por Karaboga [3–5] que se basa en el comportamiento empleado por las abejas mielíferas para encontrar buenas fuentes de alimento y comunicar esta información al resto de la colmena. ABC utiliza parámetros como el tamaño de la colonia y el número máximo de generaciones. ABC es un algoritmo diseñado para resolver problemas de optimización combinatoria, que se basa en poblaciones donde las soluciones, llamadas fuentes de alimento, son modificadas por abejas artificiales. El objetivo de estas abejas es descubrir fuentes de alimento que tengan cada vez mayores cantidades de néctar, y finalmente devuelven la fuente de alimento más abundante que pudieron encontrar. De esta forma, se obtienen soluciones óptimas locales, e idealmente el óptimo global. En un sistema ABC, las abejas artificiales se mueven en un espacio de búsqueda multidimensional eligiendo fuentes de néctar dependiendo de su experiencia pasada y de sus compañeras de colmena. Cuando una abeja encuentran una mejor fuente de néctar, la memorizan y olvida la anterior. De este modo, ABC combina métodos de búsqueda local y búsqueda global, intentando equilibrar el balance entre exploración y explotación.

3.1. Algoritmo Colonia de Abejas Artificiales

El algoritmo ABC emplea varias soluciones del problema al mismo tiempo, cada solución es considerada como una fuente de alimento. La calidad de las soluciones es evaluada mediante la función objetivo, y puede ser vista como la cantidad de néctar

en la fuente de alimento. A las funciones destinadas a producir modificaciones en las soluciones, se les llamada *abejas artificiales* y por el tipo de función, se clasifican en tres grupos: *abejas empleadas*, *abejas observadoras* y *abejas exploradoras*. El trabajo coordinado de los tres tipos de abejas produce cambios en las soluciones, de tal forma que se pueden encontrar fuentes de alimento cada vez de mejor calidad.

Las abejas empleadas están asociadas con una fuente de alimento en particular, y son las que explotan el alimento, a la vez que llevan consigo la información de esta fuente particular de alimento a las observadoras. Las abejas observadoras son aquellas que están esperando en el área de danza de la colmena para que las abejas empleadas les compartan la información sobre las fuentes de alimento, y entonces tomen una decisión de elección de alguna fuente de alimento. Las abejas que salen del panal en busca de una fuente de alimento al azar son las llamadas exploradoras.

En el algoritmo de Colonia de Abejas Artificiales, el número de abejas empleadas y el número de abejas observadoras, es igual al número de fuentes de alimento que están alrededor del panal. Cuando una fuente de alimento se agota debe ser abandonada, y la abeja empleada que la explotaba se convierte en una abeja exploradora que busca de forma aleatoria una fuente nueva.

La posición de una fuente de alimento representa una solución factible del problema de optimización y el monto de néctar o de alimento de la fuente corresponde a la calidad de la solución asociada a dicha fuente. En el Algoritmo 2 se presenta el pseudocódigo de colonia de abejas artificiales

En la siguiente sección se da una descripción de la forma en que opera esta técnica.

3.2. Aplicación a Distritos Electorales

Para el diseño de distritos electorales, una fuente de alimento se define como una solución $E = \{D_1, D_2, ..., D_n\}$, donde D_s es un conjunto de UG para $s = 1, 2, \ldots, n$. Una población inicial de M fuentes de alimento se genera utilizando la estrategia descrita en la sección 2.2.1, de tal manera que cada solución está formada por n distritos conexos.

De acuerdo a la técnica propuesta por Karaboga, cada fuente de alimento, E_i, debe ser modificada por exactamente una abeja empleada. En este caso, cada solución

Algoritmo 2: Algoritmo de Colonia de Abejas Artificiales en pseudo-código

```
1  INICIALIZA. Generar de forma aleatoria M fuentes de alimento E_1, ..., E_M.
2  Repite
3      Para i := 1 a M hacer
4          Enviar una abeja empleada a la fuente de alimento E_i.
5          Modificar la fuente de alimento E_i y determinar la cantidad de néctar de
           la nueva solución, E'_i, mediante la función objetivo.
6          si la nueva solución es mejor entonces
7              E'_i ⟵ E_i
8          fin
9      fin
10     Para i := 1 a M hacer
11         Calcular la probabilidad de cada fuente de alimento para ser elegida por
           una abeja observadora.
12     fin
13     Para i := 1 a M hacer
14         Elegir en probabilidad una fuente de alimento E_j.
15         Enviar una abeja observadora a la fuente de alimento seleccionada.
16         Modificar la fuente de alimento E_j y determinar la cantidad de néctar de
           la nueva solución, E'_j, mediante la función objetivo.
17         si la nueva solución es mejor entonces
18             E'_j ⟵ E_j
19         fin
20     fin
21     Detener la exploración de las fuentes de alimento que han sido agotadas por
       las abejas.
22     Enviar a las abejas exploradoras para encontrar nuevas fuentes de alimento
       de forma aleatoria.
23     Memorizar la mejor fuente de alimento encontrada hasta el momento.
24 hasta Cumplir criterio de paro;
25 Termina el algoritmo
```

sufre un cambio para obtener la solución $E_i^{'}$ siguiendo la estrategia descrita en la sección 2.2.2. Si la nueva solución $E_i^{'}$ tiene una cantidad de néctar mejor que E_i, $E_i^{'}$ sustituye a E_i y se convierte en una nueva fuente de alimento explotada por la colmena. En otro caso, $E_i^{'}$ se rechaza y E_i se conserva.

En cuanto el proceso de las abejas empleadas se ha realizado en todas las M fuentes de alimento, empieza el turno de las abejas observadoras. Cada abeja observadora evalúa la calidad de las soluciones obtenidas hasta este momento, y elije una fuente de alimento con base en la probabilidad p_i, dada por:

$$p_i = \frac{\text{Calidad}_i}{\sum_{j=1}^{M} \text{Calidad}_j} \tag{3.1}$$

Donde:
Calidad$_i$ es la calidad de la fuente de alimento E_i, calculada mediante la función objetivo.
M es el número de fuentes de alimento.

Una vez que ha seleccionado una solución, E_{Origen}, la abeja observadora intentará mejorarla mediante un proceso que combina algunas características de esta fuente de alimento con otra solución, $E_{Destino}$, elegida de forma aleatoria. Este proceso de combinación se resume en los siguientes pasos.

Sea E_{Origen} la fuente de alimento que se va a modificar y $E_{Destino}$ la solución con la que deberá combinarse. Se elige una UG, k, de forma aleatoria. Entonces, existen un distrito $D_i \in E_{Origen}$ y un distrito $D_j \in E_{Destino}$ tales que $k \in D_i \cap D_j$. Ahora se deben considerar los siguientes conjuntos:

$$H_1 = \{l : x_{li} = 0, x_{lj} = 1\} \tag{3.2}$$

$$H_2 = \{l : x_{li} = 1, x_{lj} = 0\} \tag{3.3}$$

Donde:

$$x_{li} = \begin{cases} 1, & \text{si la UG } l \text{ pertenece al distrito } i \\ 0, & \text{en otro caso} \end{cases}$$

Entonces una UG en H_1 es insertada en D_i, y una UG en H_2 es extraída de D_i, e insertada en un distrito contiguo a D_i.

Es importante observar que estos movimientos pueden producir disconexión en el distrito D_i, por lo que debe realizarse un proceso que la repare. Para esto, se cuenta el número de componentes conexas en D_i. Si el número de componente conexas es 1, entonces el distrito no perdió su conexidad. En otro caso, la componente conexa que contiene a k (i.e., la UG usada en el proceso de combinación mencionado anteriormente) se define como el distrito D_i, mientras que el resto de las componentes son asignadas a otros distritos adyacentes.

Una vez que han concluido los procesos de combinación y reparación antes mencionados, se obtiene una nueva fuente da alimento E'_{Origen}. Si la nueva solución E'_{Origen} tiene una cantidad de néctar mejor que E_{Origen}, E'_{Origen} sustituye a E_{Origen} y se convierte en una nueva fuente de alimento explotada por la colmena. En otro caso, E'_{Origen} es rechazada y E_{Origen} es conservada.

Se considera una iteración del algoritmo después de que las M abejas empleadas y las M abejas observadoras hacen su labor. El algoritmo finaliza después de L iteraciones.

En cada iteración se lleva una estadística del número de veces que cada fuente de alimento sufre un cambio. Si una fuente de alimento no cambia después de un número N de iteraciones, entonces una abeja exploradora sustituye esta fuente de alimento usando la estrategia descrita en la sección 2.2.1.

Bibliografía

[1] S. G. de los Cobos, J. Goddard, M. A. Gutiérrez y A. E. Martínez, "Búsqueda y exploración estocástica", Ed. México: UAM-I, 2010.

[2] S. Kirkpatrick, C. D. Gellat y 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 y 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 y 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.