**Universidad de Costa Rica**

**Facultad de Ingeniería  
Escuela de Ciencias de la Computación e Informática  
CI-0113 Programación I**

**Profesor: Edgar Casasola  
  
Estudiantes:  
Katherine Angulo Mendoza B70442  
Josué Zeledón Ruiz B78567  
Nicole Castillo Arguedas B71755  
  
Grupo: 03  
  
Tercer Tarea Programada  
  
Semestre I - 2018**

**2. Descripción del problema**(el enunciado)

El objetivo de esta Tarea es producir una solución escalable mediante el uso de Polimorfismo heredando de un conjunto de clases abstractas pre-diseñadas. El problema es escalar una aplicación que permite visualizar grupos de elementos que se forman a partir de la aplicación de diferentes algoritmos de agrupamiento utilizando una función de distancia mutua. En esta tarea se deben implementar clases que hereden de las clases abstractas definidas en clase.

Se deberán implementar Fábricas que permitan producir cada uno de los nueve tipos de Producto específico. Los productos serán de tres tipos: elementos, agrupadores y visualizadores. En la aplicación que se desarrolló, cualquier instancia de cada uno de estos tres grupos debe tener la capacidad de interactuar con instancias de los otros dos grupos restantes. En otras palabras, los tres elementos pueden ser agrupados por cualquier agrupador y las listas de elementos (grupos) generados podrán ser visualizados por cualquiera de los tres visualizadores que generen. Es más, si se compilan juntas implementaciones de otros grupos de estudiantes deberían poder interactuar entre sí, ya que solo pueden interactuar mediante los métodos especificados en los .h respectivos.

Cada tipo de elemento debe contar con una función para calcular la distancia entre objetos del mismo tipo. La Lista es un Elemento por lo que también se puede calcular la distancia entre una lista y otra.

Se debe crear al menos los siguientes elementos:

1. Palabras, cuya distancia se calcula utilizando bigramas y distancia obtenida con la fórmula: D(p1,p2) = 1 - 2 |C| / ( |A| + |B|) Donde A y B corresponden a los conjuntos de bigramas de letra que se obtienen de las palabras p1 y p2 respectivamente. Y C el conjunto de bigramas en común entre ambos conjuntos.

2. Vectores de números reales con distancia cartesiana.

3. Elemento opcional: Puede ser el elemento Hilera de palabras mencionado en clase utilizando distancia de coseno entre vectores con las frecuencias de las palabras en cada hilera. Aunque pueden implementar otro elemento de su elección.

Se debe crear al menos tres tipos de algoritmos de agrupamiento:

1. El jerárquico aglomerativo (HAC)

2. El de K-means

3. Y uno definido por el grupo de trabajo

Para visualizar los grupos y sus elementos, se deben implementar tres tipos de visualizadores:

1. XML standalone

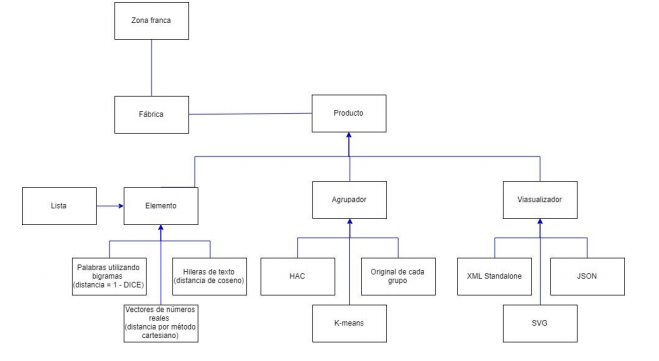
2. JSON

3. SVG

Los archivos para cargar los elementos cuentan con el identificador o nombre del elemento en la primera línea, la cantidad n de elementos en la segunda línea, y los datos para crearlos en las siguientes n líneas, una línea por elemento.

Ya se implementó en clase el Main.cpp, la ZonaFranca.cpp.

En la siguiente figura se presenta un diagrama representando la relación entre las clases ZonaFranca, la Fábrica y los Productos. Cabe resaltar que deben crearse 9 fábricas, para los 9 tipos de productos especificados.



Nota: Debe crear clases que heredan de las especificadas en los .h indicados e implementar sus respectivos .cpp. No puede hacer cambios a los .h o .cpp suministrados, excepto los que dentro del código así lo indiquen. En caso de que se requiera agregarles algo más, esto se debe discutir en clase.

**3. Descripción de la solución del problema (clases y métodos)**

**Solución del Problema**

**Elementos:**

* **Vector:**

En la clase Vector se desarrollaron específicamente los métodos que son heredados virtualmente por parte de la clase Elemento. Se calcula la distancia entre un vector y otro por medio de la fórmula de distancia Similitud de Cosenos. Además, se implementaron las debidas medidas para el correcto funcionamiento y el efectivo cálculo de la distancia entre 2 vectores, validando si los 2 vectores a comparar se encontraban o poseían la misma dimensión, entre otras validaciones.

* **Palabra:**

Para programar la clase palabra se desarrollaron los métodos que debían ser implementados ya que palabra utilizar el polimorfismo, el método el cual tenía que dar la distancia entre una palabra y otra se hace con el coeficiente de DICE pero como debe de dar un número entre 0 y 1, donde más se acerque a cero esta distancia el elemento será agrupable la fórmula pasaría a ser el coeficiente de Jaccard, esta fórmula es:

1-((|A∩B|) / (|AUB|)).

* **Punto:**

En la clase punto (el elemento opcional que decidimos programar) se implementaron los métodos que este heredaba de la clase padre elemento. Para calcular la distancia entre puntos se utilizó la distancia euclidiana y, seguidamente,normalizamos el resultado para conseguir un dígito entre 0 y 1.

**Visualizadores:**

* **XML:**

En el método visualizar se crea un archivo con extensión xml donde se escribe en este formato.

* **Json:**

En el método visualizar se crea un archivo con extensión json donde se escribe en este formato.

* **SVG:**

En el método visualizar se crea un archivo con extensión .svg que se puede visualizar por medio del navegador Internet Explorer. Corresponde a una imagen vectorial de la lista ya agrupada.

**Agrupadores:**

* **Kmeans:**

El algoritmo Kmeans se hace con dos centroides (los cuales fueron elegidos por el grupo de trabajo), en la primera iteración estos centroides son elegidos al azar,elegimos el primer elemento de la lista y el segundo, para elegir los siguientes centroides en las demás iteraciones desarrollamos un método el cual toma una lista de los grupos que se formaron con cada centroide, y revisa las distancias de todos los elementos con todos los elementos, al haber obtenido estas distancias se realiza la suma que provocó cada elemento y se compara con las demás, la que sea menor es la del elemento que sería el mejor centroide, esto se hace por separado para cada lista(son dos, por los dos centroides).

Al investigar notamos que Kmeans termina cuando converja, para esto dan 3 soluciones, la primera es cuando se vuelvan a elegir los mismos centroides, cuando las listas no tengan ningún cambio, o dar un máximo de iteraciones, nosotros decidimos dar un máximo de iteraciones para determinar la condición de parada del algoritmo.

* **Hac:**

Siguiendo el algoritmo de HAC, construimos primeramente una matriz que guarda las distancias de todos los elementos con todo el resto. Con dicha matriz somos capaces de encontrar cuales son los dos primeros elementos que agrupamos ya que son los que tendrán una distancia menor (están más cerca). Estos elementos se guardarán en una lista y se prosigue a calcular las distancias de todos los elementos con la lista recién creada para encontrar el más cercano y ahora se almacenará el elemento escogido con la lista pasada.El procedimiento se repite hasta que ya todos los elementos estén agrupados.

* **DBSCAN**

El algoritmo DBSCAN se realiza por medio de núcleos, los cuales se eligen entre la lista de los elementos. Estos núcleos, para ser considerados así deben tener a su alrededor (por medio de la distancia entre este posible núcleo y los demás elementos) un *número mínimo* de elementos asociados, determinados bajo una distancia máxima *epsilum*. Conforme se van agrupando los elementos alrededor del núcleo, estos elementos pueden ser núcleos a su vez, por lo que deben también evaluarse, esto con el objetivo de agrupar la mayor cantidad de elementos posibles dentro de un grupo o “clúster”.

Se realizó primero una matriz que contenía la distancia entre todos los elementos contra todos los otros, luego se va evaluando si un elemento es un posible núcleo, y de ser así, busca por medio de las posiciones en la matriz a todos los elementos que sean cercanos a él, además haciendo que el proceso se repita para cada uno de estos nuevos elementos, como otros posibles núcleos. Se controla o se sabe si un elemento ya fue agrupado por medio de un vector de enteros que almacena un número relacionado al estado de dicho elemento (agrupado, no agrupado o no evaluado). Finalmente los elementos que no hayan podido ser agrupados pasan a formar parte de la lista como elementos independientes, lo que se conoce como “Ruido” del Clúster.

**Clases:**

* **Elemento**

friend istream& operator>>(istream & entrada, Elemento \* elemento) método sobrecargado para cargar

friend ostream& operator<<(ostream & salida, Elemento \* elemento) método sobrecargado para imprimir

virtual ostream & imprimir(ostream &)=0 método que para imprimir, devuelve el mismo flujo que recibe de parámetro y es virtual puro por lo que todos los hijos de elemento deberán implementarlo

virtual istream & cargar(istream &)=0 método que para cargar, devuelve el mismo flujo que recibe de parámetro y es virtual puro por lo que todos los hijos de elemento deberán implementarlo

virtual ~Elemento() destructor de elemento, método virtual con una implementación vacía (todos los hijos lo implementaron)

virtual Elemento \* clonar()=0 método en el que un elemento se clona a el mismo y devuelve un puntero de su copia

virtual double distancia(Elemento \*)=0 método para calcular la distancia entre dos elementos,es virtual puro por lo que todos los hijos de elemento deberán implementarlo

* **Fábrica**

void \_initHilera(char \*\* atributoPtr, const char \* nombre)metodo para inicializar un atributo

void setTipoProducto(const char \* nombre) metodo que le asigna un tipo al producto

void setNombreProducto(const char \* nombre)metodo que le asigna un nombre al producto

Fabrica(const char \* tipoProducto, const char \* nombreProducto) constructor con parametros

virtual ~Fabrica()destuctor de fabrica

virtual const char \* getNombreProducto() metodo que retorna el nombre del producto

virtual const char \* getTipoProducto()metodo que retorna el tipo del producto

virtual Producto \* producir()=0 metodo que crea una instancia (produce) de un producto (devuelve un puntero a un producto)

virtual int produce(const char \* nombreProducto)metodo que compara el nombre de dos productos (el mismo y el que recibe de parametro)

virtual int esTipo(const char \* tipoProducto)método que compara el tipo de dos productos (el mismo y el que recibe de parámetro)

* **Visualizador**

virtual void visualizar( Lista \* )=0 metodo para visualizar los elemento de la lista que recibe de parámetro en un formato específico implementado en cada clase que herede de agrupador

* **Agrupador**

virtual Lista \* agrupar(Lista \*)=0 método para agrupar los elementos dentro de una lista que recibe de parámetro,es virtual puro por lo que todos los hijos de elemento deberán implementarlo

* **ZonaFranca**

int cantidadDeFabricas(), método que retorna la cantidad de fábricas presentes en la Zona Franca

agregar(Fabrica \*), procedimiento que logra agregar las fábricas de todos los productos a necesitar

ZonaFranca(), constructor de la instancia de ZonaFranca

~ZonaFranca(), destructor de la Zona Franca

Fabrica \* getFabrica(const char\*, const char \*), método que crea una fábrica y la retorna para ser implementada en la ZonaFranca

* **Lista**

void destruir(), método que se encarga de eliminar los elementos de la lista

void aplanar(Lista &), método que “aplana” la lista para poder hacer el cálculo de la distancia de sus elementos

ostream & imprimir(ostream &) método para imprimir una Lista

istream & cargar(istream &) metodo para cargar una Lista

Iterator begin(), método que retorna un iterator apuntando al inicio de la lista

Iterator end(), método que retorna un iterator apuntando al final de la lista

Lista(), constructor de la clase Lista

Lista( Elemento \*, istream &, int n), método que carga una lista con un flujo de datos

~Lista(), método destructor de la clase Lista

Lista \* clonar(), método que clona una Lista y devuelve dicha clonación

double distancia(Elemento \*), método que retorna la distancia de la Lista hacia un Elemento

Lista & operator=(Elemento &), método que asigna a la lista un Elemento, ya sea una lista o un elemento sólo

Lista & operator+=(Elemento \*), método que agrega un elemento al final de la lista

Lista & insertar(Iterator&, Elemento \*), método que inserta una copia del elemento a la lista

Lista & borrar(Iterator&), método que borra el elemento apuntado por el Iterator

Lista & push\_front(Elemento \*), método que agrega un elemento al inicio de la lista

Elemento \* pop\_front(), método que devuelve el primer elemento de la lista

Elemento \* pop\_back(), método que devuelve el último elemento de la lista

* **Celda**

Celda(Elemento \*), que es el método constructor de la celda

~Celda(), método que destruye la celda actual

* **Iterator**

Iterator(), que es el constructor por omisión

Iterator( Celda \* ), que es el constructor que apunta a la celda recibida como parámetro

Elemento \* operator\*(), método que retorna el elemento apuntado por Iterator

Iterator &operator++(), método que mueve el Iterator adelante

Iterator &operator--(), método que mueve el Iterator hacia atrás

Iterator operator++(int noSeUsa), método que mueve el Iterator hacia adelante(pos incremento)

Iterator operator--(int noSeUsa), método que mueve el Iterator hacia atrás( pos incremento)

int operator==(const Iterator &), método que compara la posición de un iterator contra otro y retorna un entero con base a ello

int operator!=(const Iterator & otro), método que compara si un Iterator es distinto a otro y retorna un entero con base a ello

Iterator & operator=(Elemento \*), método que asigna un elemento nuevo a apuntar por sí mismo

* **Producto**

virtual ~Producto() método destructor que será heredado por todos los hijos de producto

* **Punto**

ostream & imprimir(ostream &) método para imprimir un Punto

istream & cargar(istream &) metodo para cargar un Punto

Punto();// constructor por omisión

Punto(int , int ) constructor con params

Punto( const Punto &) constructor por copia

~Punto() destructor de Punto

Punto \* clonar() método encargado de clonar un Punto

double distancia(Elemento \*) método que calcula la distancia entre dos Punto mediante la fórmula de la distancia Euclidiana

* **Vector**

int cuenta\_Numeros(const char \*, const char \*), método que retorna la cantidad de números del vector

ostream & imprimir(ostream &) método para imprimir un vector

istream & cargar(istream &) metodo para cargar un vector

Vector(), método constructor de vector

~Vector(), método destructor

Vector \* clonar(), Clonar que devuelve un puntero a Vector (una copia)

void set\_Dimension(int), método que devuelve asigna la dimensión de un vector

double calculo\_Norma(), método que se encarga de calcular la norma de un vector

double distancia(Elemento \*), método que retorna la distancia de cosenos

double calculo\_Distancia(Vector \*), método que realiza el calculo de la distancia entre 2 vectores

int get\_Dimension(), método que retorna la dimension del Vector

* **Palabra**

ostream & imprimir(ostream &) método para imprimir una palabra

istream & cargar(istream &) metodo para cargar una palabra

Palabra() constructor por omisión de la clase

Palabra(string palabra) constructor con parámetro de la clase

Palabra( const Palabra &) constructor por copia de la clase

~Palabra() destructor de la clase palabra

Palabra \* clonar() método que clona una palabra

double distancia(Elemento \*) método que calcula la distancia entre una palabra y otra, utilizando la fórmula del coeficiente de Jaccard.

double intersección(string, string) método para calcular la intersección que existe entre una palabra y otra posteriormente calcula la distancia se utiliza en la fórmula de distancia

* **SVG**

void visualizar(Lista \*) método que se encarga de crear un archivo .svg para la visualización del agrupamiento

* **XML**

void visualizar(Lista \*) método que se encarga de crear un archivo .xml para la visualización del agrupamiento

* **Json**

void visualizar(Lista \*) método que se encarga de crear un archivo .json para la visualización del agrupamiento

* **Kmeans**

Kmeans() constructor por defecto

double \*\* matrizDistancias(Lista \*,Lista::Iterator, Lista::Iterator) calcula la distancia que existe entre los centroides y los otros elementos

int distanciaMenor(double, double) busca en la matriz la menor distancia

Lista::Iterator \* centroide(Lista \*, int) busca posición centroides después de la primera búsqueda

Lista \* agrupar(Lista \*) método final que devuelve la lista de los grupos realizados

int tamanoLista(Lista \*) calcula el tamaño de la lista

void reiniciarLista(Lista &) vacía una lista que recibe por referencia

int buscarMenor(double \*, int) método que busca el menor en un vector

int matrizMenor(Lista \*) para encontrar el centroide

* **HAC**

HAC(); //constructor por defecto

void matrizDistancias(Lista \*) calcula la distancia que existe entre todos los elementos de una lista y da la posición de los más cercanos

void vectorDistancias(Lista \*, Lista \*) calcula la distancia entre los elementos de una lista y otra lista y da la posición del más cercano

Lista::Iterator buscarIterador(Lista \*, int) busca el iterador de la lista basado en la posición que se le indica

Lista \* agrupar(Lista \*) método final que devuelve la lista de los grupos realizados

int tamanoLista(Lista \*) calcula el tamaño de la lista

void reiniciarLista(Lista &) vacía una lista que recibe por referencia

* **DBSCAN**

DBSCAN(), método constructor por omision

Lista \* agrupar(Lista \*), método que se encarga de agrupar la lista y devolverla por puntero

int cuenta\_Elementos(Lista \*), método que se encarga de contar la cantidad de elementos de la lista

void rellena\_Matriz\_Distancia(Lista \*, double\*\*), procedimiento que rellena una matriz de double con la distancia entre todos los elementos

Lista \* generar\_Lista(Lista \*, double \*\*), método que retorna la lista ejecutada con los algoritmos de agrupamiento

void unir\_Elementos(Lista \*, Lista \*, double \*\*, int\* , int),método que se encarga de unir los elementos en las listas y sublistas que se generarán

Elemento \* retorna\_Elemento (Lista\*, int, método que retorna el elemento apuntado por un interator de Lista

int verifica\_Min\_Points(double \*\*, int), método que devuelve el número de elementos que están asociados al núcleo seleccionado

* **FabricaPunto**

Producto \* producir() crea una instancia de la clase Punto

* **FabricaVector**

Producto \* producir() crea una instancia de la clase Vector

* **FabricaPalabra**

Producto \* producir() crea una instancia de la clase Palabra

* **FabricaJson**

Producto \* producir() crea una instancia de la clase Json

* **FabricaXML**

Producto \* producir() crea una instancia de la clase XML

* **FabricaSVG**

Producto \* producir() crea una instancia de la clase SVG

* **FabricaHAC**

Producto \* producir() crea una instancia de la clase HAC

* **FabricaKmeans**

Producto \* producir() crea una instancia de la clase Kmeans

* **FabricaDBSCAN**

Producto \* producir() crea una instancia de la clase DBSCAN

**4. Descripción de la compilación y ejecución del programa**

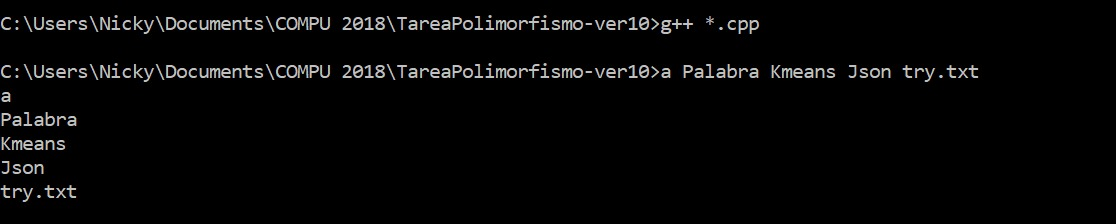
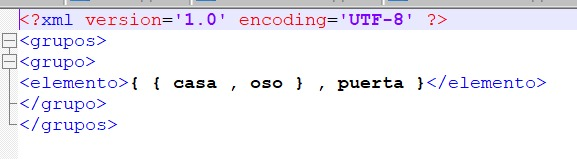
****

Figura 1. compilación del programa.

**** Figura 2. visualización de los grupos en XML con algoritmo HAC.

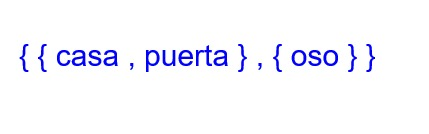
****

Figura 3. visualización con algoritmo HAC en SVG

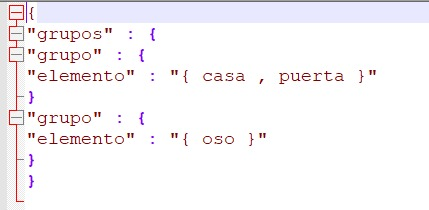
****

Figura 4. archivo .json creado con algoritmo Kmeans.

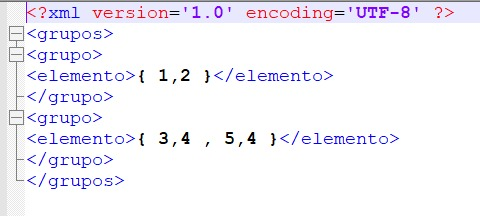


Figura 5. puntos agrupados con algoritmo Kmeans, visualización XML

**5. Diagrama de clases UML**

El diagrama UML se incluye externamente debido a que es muy grande y no se visualiza con claridad en una imagen.

El nombre del archivo es: UML programada 3.svg