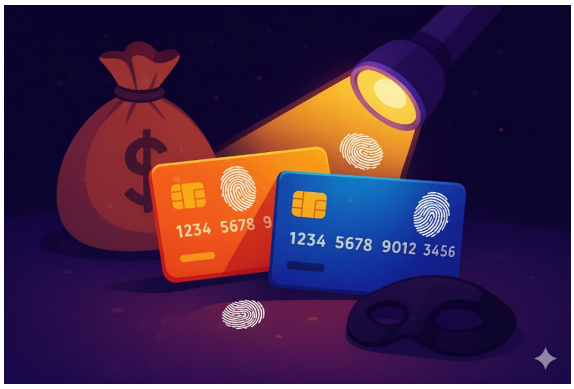
UNIVERSIDAD DE GRANADA

ETSIIT Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

DECSAI

Metodología de la Programación

Curso 2025/2026



Guion de prácticas

Fraud1

Implementación de un algoritmo de clustering

Marzo de 2026

Silvia Acid, Andrés Cano, Luis Castillo
Dpto. Ciencias de la Computación e Inteligencia Artificial
Universidad de Granada

Índice

1. Introducción

Como ya se indicó con anterioridad, las prácticas tienen por objeto realizar distintas aplicaciones tomando como entrada un conjunto de datos sobre compras realizadas por ciertos clientes en distintos establecimientos del campus de la Universidad de Princeton en EEUU.

En esta práctica vamos a desarrollar una aplicación que lea los datos de un conjunto de localizaciones desde la entrada estándar y los agrupe en K grupos (clústeres) usando el algoritmo de las *K-medias*. Este algoritmo toma como entrada un conjunto de localizaciones, cada una determinada por una posición `x` e `y`, y el número K de clústeres deseado y obtiene como salida un conjunto de K clústeres disjuntos, en el que cada uno contiene localizaciones más o menos próximas. Por ejemplo, la figura 1 muestra un posible clustering para el conjunto de las 21 localizaciones de la universidad de Princeton usando $K = 5$. Cada clúster se distingue por colores diferentes. Como puede ver, cada clúster obtenido puede tener un número diferente de localizaciones.

Figura 1: Ejemplo de clustering obtenido para las 21 localizaciones del Campus de la Universidad de Princeton. Imagen descargada en Enero de 2026 de la web `https://www.cs.princeton.edu/courses/archive/spring25/cos226/assignments/fraud/specification.php`

2. Arquitectura de las prácticas

Como ya se indicó en la práctica anterior, las prácticas `Fraud` se han diseñado por etapas; las primeras contienen estructuras más sencillas,

sobre las cuales se asientan otras estructuras más complejas y se van completando con nuevas funcionalidades.

En `Fraud1` conservamos las clases `Location` y `VectorLocation` desarrolladas en la práctica anterior y añadiremos las clases `VectorInt` y `Clustering`. En la clase `VectorLocation` solo será necesario añadir tres nuevos métodos. El módulo `ArrayLocation` desaparece en esta práctica, pues ya no será necesario, aunque una de las funciones que contenía, la función `ReadArrayLocation`, nos puede ser de ayuda para implementar el método `load()` de la clase `VectorLocation`, ya que ambos módulos son muy similares. Vea la figura 2.

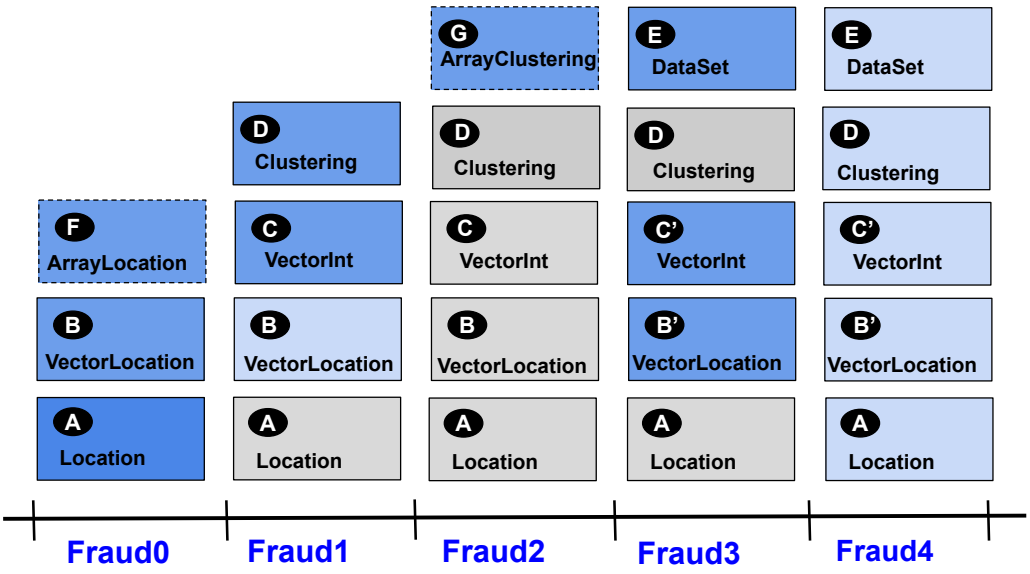


Figura 2: Arquitectura de las prácticas de MP 2026. Como podemos observar, los bloques correspondientes a cada módulo, aparecen en distintos colores. El color azul indica software que requiere ser desarrollado por el estudiante. Los cambios considerables (como los cambios en la estructura interna de una clase) se muestran en color intenso, mientras que pequeños cambios, como la incorporación de nuevas funcionalidades, se muestran en su correspondiente color suave. Finalmente, en gris se muestran módulos que no sufren cambios respecto a la versión anterior de las prácticas.

Describamos brevemente los módulos que usaremos en esta práctica:

A Location
: La clase Location se implementó en la práctica anterior y no necesita de ningún cambio en esta práctica.

B VectorLocation
: La clase VectorLocation solo necesita añadir tres nuevos métodos (ver sección 5.4), los métodos (`nearest()`, `assign()` y `load()`).

C VectorInt
: Nueva clase que permite guardar un vector de números enteros (tipo `int`) (ver sección 5.4).

D Clustering
: Nueva clase que va a contener los datos para almacenar un agrupamiento *clustering*) de un conjunto de localizaciones, junto con la implementación del algoritmo para obtenerlo (ver las secciones 4 y 5.4).

main.cpp Contiene la función main() cuya finalidad será la de obtener un agrupamiento para un conjunto de localizaciones.

3. Objetivos

El desarrollo de la práctica `Fraud1` persigue los siguientes objetivos:

- Comprender la importancia de la interfaz de una clase (ocultamiento de información y encapsulamiento).
- Seguir practicando con los **calificadores** public, private, const y static en métodos y datos miembro.
- Seguir practicando con la devolución por referencia y referencia constante por parte de un método o función.
- Continuar practicando con la lectura de datos por redireccionamiento desde la entrada estándar.
- Reforzar la comprensión de los conceptos de compilación separada.
- Aprender a adecuar los pasos de parámetros (por valor, por referencia, por referencia constante) a métodos y funciones con las especificaciones proporcionadas.
- Aprender a distinguir cuándo un método debe ser declarado como constante.
- Conocer cómo pasar un flujo como parámetro de una función.

4. El algoritmo de las K-medias. Clase Clustering

Como se ha indicado en la introducción, en esta práctica debemos implementar el algoritmo de las K-medias. Este algoritmo será implementado dentro de la clase `Clustering`, concretamente en el método `run()`. Este algoritmo toma como entrada el número K de clústeres deseado y un conjunto de localizaciones, cada una definida por unas coordenadas `x` e `y`, y obtiene como salida un conjunto de K clústeres disjuntos, en el que cada uno contiene un conjunto de localizaciones más o menos próximas entre sí. Las localizaciones de cada clúster serán subconjuntos disjuntos de las localizaciones originales. Cada clúster será representado por una localización especial llamada *centroide*, que se calcula como el vector de

medias de las localizaciones incluídas en su clúster (ver más adelante la fórmula 1).

Por ejemplo, en la figura 3 se muestra en el gráfico de la derecha el resultado de aplicar el algoritmo K-medias al conjunto de localizaciones del gráfico de la izquierda, tomando K = 2. Se marcan con círculos rellenos, las localizaciones de uno de los clústeres, y con cuadrados no rellenos las localizaciones del otro. También se pueden identificar mediante una cruz, cada uno de los centroides calculados a partir de los puntos de cada clúster.

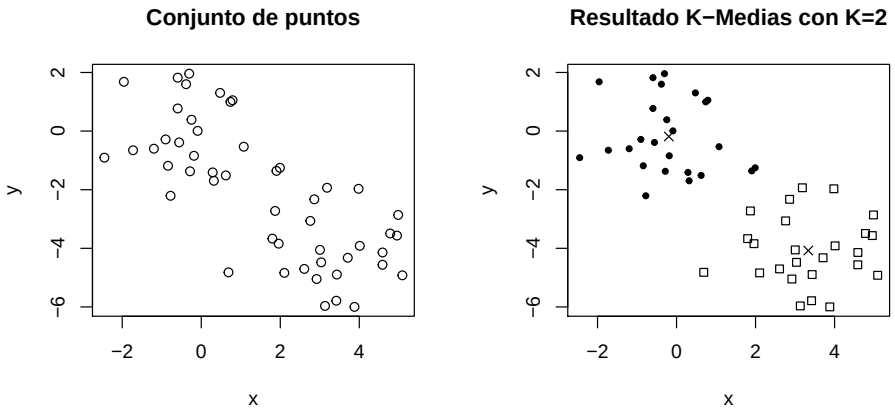


Figura 3: Aplicación del algoritmo K-medias con K=2

4.1. La clase Clustering

La clase `Clustering` ofrece la funcionalidad necesaria para realizar el clustering del conjunto de localizaciones mediante el algoritmo de las K-medias. A continuación detallamos los datos miembro que tiene esta clase:

- `VectorLocation _locations`: Contiene la lista de localizaciones objeto del algoritmo de clustering.
- `int _K`: Número de clústeres a obtener.
- `VectorInt _clusters`: Vector de enteros que, una vez ejecutado el algoritmo de clustering, permite obtener el clúster asignado a cada localización. Tiene una longitud igual a la del número de localizaciones que estemos usando. Así, el elemento 0 de este vector (`_clusters.at(0)`) contendrá el número de clúster asignado a la localización número 0 de `_locations`; el elemento 1, contendrá el número de clúster asignado a la localización número 1 de `_locations` y así sucesivamente. Los números incluidos en el vector `_clusters` son números enteros entre 0 y `_K-1` (ambos inclusive).

  Por ejemplo, supongamos que el vector `_clusters` tiene el siguiente contenido:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	2	2	2	2	2	1	1	1	1	1	4	4	4	4	4	4	4	4	4	3

El anterior vector `_clusters` indicaría que la localización 0 estaría asignada al clúster 0, la localización 1 al clúster 2, . . ., la localización 20 al clúster 3. De esta forma, los clústeres obtenidos por el algoritmo K-medias estarían formados por los siguientes subconjuntos de localizaciones:

- Clúster 0: $\{0\}$
- Clúster 1: $\{6, 7, 8, 9, 10\}$
- Clúster 2: $\{1, 2, 3, 4, 5\}$
- Clúster 3: $\{20\}$
- Clúster 4: $\{11, 12, 13, 14, 15, 16, 17, 18, 19\}$

Los números enteros incluídos en cada uno de los subconjuntos anteriores, corresponden a posiciones en el vector de localizaciones (`_locations`). Por ejemplo el clúster 3 contiene la localización número 20. Esta localización está almacenada en la posición 20 del vector `_locations`. El objeto `Location` con toda la información de esa localización podría obtenerse mediante `_locations.at(20)`.

- `VectorLocation _centroids`: Contiene el vector de los K centroides. Cada centroide será recalculado en cada iteración del algoritmo K-medias como el vector de medias de las localizaciones de su clúster. Así, si llamamos C_j al clúster j, p_i a las coordenadas x e y de la localización número i del clúster C_j, y c_j al centroide del clúster C_j, entonces el centroide c_j será recalculado en cada iteración del algoritmo de las K-medias de la siguiente forma:

$$c_j = \frac{1}{|C_j|} \sum_{p_i \in C_j} p_i \tag{1}$$

En la anterior expresión, $|C_j|$ denota el número de localizaciones incluidas en el clúster C_j. Tenga en cuenta que la suma de dos puntos (localizaciones) p_1 y p_2 es otro punto p_r en el que la coordenada x se obtiene como suma de las coordenadas x de p_1 y p_2, y de forma similar con la coordenada y. O sea:

$$p_r(x, y) = (p_1(x) + p_2(x), p_1(y) + p_2(y)) \tag{2}$$

- `unsigned int _seed`: Semilla que debe usarse para inicializar el generador de números aleatorios. La inicialización de la semilla debe hacerse solo desde el método `initialClusterAssignment()`, mediante una llamada a la función `srand(_seed)`.

- `bool _isDone`: Indica si el método `run()` ha sido ya ejecutado. El método `run()` debe poner a `true` este campo una vez termine su cometido.

- `double _sumWCV`: Suma de las *within-cluster variances* de los clústeres obtenidos. La *within-cluster variance* de un clúster C_j (variabilidad interna del clúster C_j) se calcula de la siguiente forma:

$$WCV(C_j) = \sum_{p_i \in C_j} squaredDistance(p_i, c_i) \tag{3}$$

  En la anterior expresión, $squaredDistance(p_i, c_i)$ representa el valor al cuadrado de la distancia entre p_i y c_i, siendo p_i un punto del clúster C_j y c_i el centroide de C_j. O sea:

$$squaredDistance(p_i, c_i) = (p_i(x) - c_i(x))^2 + (p_i(y) - c_i(y))^2 \tag{4}$$

  Recuerda que la clase `Location` dispone de un método que permite calcular el valor `squaredDistance()` entre dos localizaciones.

  El valor `_sumWCV` se obtendría entonces como:

$$sumWCV = \sum_{C_j} WCV(C_j) \tag{5}$$

  El valor de este campo lo debe calcular el método `run()` una vez termine su cometido, haciendo uso del método `calculateSumWCV()`.

  Dese cuenta de que también puede calcular el valor de `sumWCV` (fórmula 5) al implementar el método `calculateSumWCV()`, usando un solo bucle que recorra las localizaciones p_i que tenemos en `_locations` y haciendo la sumatoria del valor al cuadrado de la distancia (método `Location::squaredDistance()`) entre p_i y el centroide c_i del clúster en el que está incluído p_i:

$$sumWCV = \sum_{i} squaredDistance(p_i, c_i) \tag{6}$$

  El centroide c_i en el que está incluido el punto p_i puede obtenerse fácilmente con `_centroids.at(_clusters.at(i))`.

- `int _numIterations`: Número de iteraciones usadas por el algoritmo de las K-medias hasta que dejan de producirse cambios en la asignación de localizaciones a un clúster. El valor de este campo lo debe establecer el método `run()` una vez termine su cometido.

4.2. Asignación inicial de un clúster a cada localización

En el algoritmo de las K-medias (ver sección 4.3) es necesario generar números aleatorios entre 0 y $K-1$ para hacer una asignación inicial de cada localización a uno de los K clústeres. Cuando un programa genera números aleatorios, es necesario inicializar la semilla que usa el generador de números aleatorios. En C++ podemos inicializar la semilla de números aleatorios llamando a la función `srand(semilla)`,

donde `semilla` es un número entero sin signo (`unsigned int`). La semilla determina la secuencia de números aleatorios que obtendremos a continuación. Si la semilla es diferente cada vez que ejecutamos el programa, entonces la secuencia de números aleatorios que se obtiene será diferente cada vez. Si usamos la misma semilla en dos ejecuciones distintas, entonces las dos ejecuciones obtendrán las mismas secuencias de números aleatorios. En nuestro programa, la inicialización de la semilla con la función `srand(semilla)` se debe hacer en el método `initialClusterAssignment()`. La semilla empleada será el valor que tenga el dato miembro `_seed`.

En el método `initialClusterAssignment()` haremos además la asignación inicial aleatoria de cada localización a uno de los K clústeres. Para generar un número aleatorio entre 0 y $K-1$ usaremos la función `rand()` de la siguiente forma: `rand() % _K`. De esta forma, el método `initialClusterAssignment()` generará los números de clúster para cada localización y los guardará en el vector `_clusters`.

4.3. Algoritmo de las K-medias

El algoritmo de las K-medias se implementará en el método `run()` de la clase `Clustering`. Este método `run()` realizará los siguientes pasos:

1. Inserta aleatoriamente cada localización en uno de los clústeres (**asignación inicial**). Lo haremos mediante una llamada al método `initialClusterAssignment()`.

2. Repite mientras que alguna localización cambie de clúster (basta con un cambio de alguna de ellas):

   a) Para cada uno de los K clústeres, se recalcula el centroide del clúster (ver fórmula 1) a partir de las localizaciones que actualmente tenga asignadas. O sea, aquí debemos actualizar cada uno de los componentes del vector `_centroids`.

   b) Asigna cada localización a un clúster actualizando los componentes del vector `_clusters`. Cada localización se asigna al clúster que está a menor distancia de su centroide. Para ello, puedes ayudarte del método `nearest()` de la clase `VectorLocation`.

5. Práctica a entregar

5.1. Finalidad del programa

Como se ha indicando anteriormente, el programa `Fraud1` va a trabajar con un conjunto de datos sobre distintos establecimientos (*localizaciones*), los cuales ofrecen algún tipo de servicio o producto. El propósito del

programa será leer de la entrada estándar el valor K (número de clústeres deseado) y los datos del conjunto de localizaciones y obtener un agrupamiento (clustering) de las localizaciones en K subconjuntos disjuntos, mostrando el resultado en la salida estándar.

El programa llevará a cabo los siguientes pasos:

1. El programa comenzará leyendo todos los datos necesarios desde la entrada estándar (valor de K y conjunto de localizaciones). El conjunto de localizaciones debe almacenarse en un objeto `VectorLocation` declarado en `main()` como variable local (`locations`).
2. Ejecutar el algoritmo de clustering (K-medias) llamando al método `run()` de la clase `Clustering`.
3. Mostrar en la salida estándar la información sobre el clustering obtenido.

Ejemplo 1 *Un ejemplo de conjunto de datos que podría proporcionarse al programa por la entrada estándar es el siguiente (cuyo contenido también puede verse en el fichero* `CodeProjects/Datasets/dataP1/princeton.p1in`*):*

```
5
21
3.3 9.1 Graduate College
11.8 6.9 Wawa Public Restrooms
12.6 7.3 Princeton Station
10.1 9.7 Forbes College
13.3 12.2 Dinky Bar & Kitchen
15.2 11.2 Whitman College
12.6 17.5 Admission Information Center
12.5 20 Rockefeller College
13.7 22.8 Art Museum Store
17 23.3 Princeton University Store
18.6 22.8 Labyrinth Books
20.9 15.4 Frist Campus Center
23.7 14.8 Cannon Dial Elm
24.8 14.9 Quadrangle
25.6 14.9 Ivy
26.4 14.9 Cottage
27.3 14.5 Cap & Gown
28.2 15 Cloister
29.1 15.2 Charter
28.8 18.7 Engineering Quadrangle
18.5 6.7 Yeh College
```

La primera línea contiene el número K de clústeres que queremos obtener, 5 en este caso. En la segunda línea, el número entero 21 indica que a continuación deben leerse 21 objetos `Location`*. Al igual que en la práctica* `Fraud0`*, los campos de cada localización están separados por un espacio en blanco, aunque podría ser también cualquier otro separador (tabulador por ejemplo). También, puede ver que el nombre de la localización puede estar compuesto de varias palabras.*

5.2. Sintaxis y ejemplos de ejecución

Al igual que en `Fraud0`, la entrada al programa se realizará completamente desde la entrada estándar y puesto que el número de datos que hay que proporcionar al programa es muy grande, lo ejecutaremos con

redireccionamiento de la entrada estándar desde un fichero, en lugar de usar el teclado, para que sea más cómodo de ejecutar.

La **sintaxis de ejecución** del programa desde un terminal es:

```
Fraud1> build/Fraud1 < <inputFile.p1in>
```

Veamos algunos ejemplos de ejecución del programa desde la línea de comandos:

Ejemplo 2 *Redireccionamiento desde el fichero* `princeton.p1in`

```
Fraud1> build/Fraud1 < ../Datasets/dataP1/princeton.p1in
```

El contenido del fichero `princeton.p1in` *es el que aparece en el ejemplo 1.*

La salida que debería obtenerse al ejecutar el programa sería la siguiente:

```
K=5
Sum of within-cluster variances: 161.136889
Number of iterations: 5
Cluster number for each location:
21
0 2 2 2 2 2 1 1 1 1 1 4 4 4 4 4 4 4 4 4 3
Centroids:
5
3.300000 9.100000
14.880000 21.280000
12.600000 9.460000
18.500000 6.700000
26.088889 15.366667
Cluster 0 information:
0 3.300000 9.100000 Graduate College
Cluster 1 information:
6 12.600000 17.500000 Admission Information Center
7 12.500000 20.000000 Rockefeller College
8 13.700000 22.800000 Art Museum Store
9 17.000000 23.300000 Princeton University Store
10 18.600000 22.800000 Labyrinth Books
Cluster 2 information:
1 11.800000 6.900000 Wawa Public Restrooms
2 12.600000 7.300000 Princeton Station
3 10.100000 9.700000 Forbes College
4 13.300000 12.200000 Dinky Bar & Kitchen
5 15.200000 11.200000 Whitman College
Cluster 3 information:
20 18.500000 6.700000 Yeh College
Cluster 4 information:
11 20.900000 15.400000 Frist Campus Center
12 23.700000 14.800000 Cannon Dial Elm
13 24.800000 14.900000 Quadrangle
14 25.600000 14.900000 Ivy
15 26.400000 14.900000 Cottage
16 27.300000 14.500000 Cap & Gown
17 28.200000 15.000000 Cloister
18 29.100000 15.200000 Charter
19 28.800000 18.700000 Engineering Quadrangle
```

La salida del programa mostrada más arriba puede verse también en el fichero `CodeProjects/Datasets/dataP1/princeton.p1out`.

Los clústeres anteriores corresponden con los que se muestran gráficamente en la figura 1.

En la carpeta `CodeProjects/Datasets/dataP1`, puede encontrar varios ficheros de texto con extensión `.p1in` con los que podrá hacer diferentes pruebas. Esta carpeta también contiene ficheros con extensión

`.p1out`. Cada uno de estos ficheros `.p1out` corresponde con la salida que debería obtenerse ejecutando el programa con el correspondiente fichero de entrada. Así por ejemplo, usando `princeton.p1in` como entrada, debería obtenerse el contenido del fichero `princeton.p1out`.

Además, se proporcionan una serie de ficheros de tests con extensión `.test` que contienen tests de integración, y que pueden usarse con el script `runTest.sh` para ejecutar automáticamente tales tests de integración según se explicó en el guion de `Fraud0`.

5.3. Configurar el proyecto en VSCode para esta práctica

Para la elaboración de la práctica `Fraud1`, tiene disponible en el repositorio de `Github`, una plantilla de proyecto VSCode. Entre los ficheros proporcionados, puede encontrar:

- En la carpeta `include`: `Location.h`, `VectorLocation.h`, `VectorInt.h` y `Clustering.h`.
- En la carpeta `src`: `Location.cpp`, `VectorLocation.cpp`, `VectorInt.cpp`, `Clustering.cpp` y `main.cpp`.
- En la carpeta `doc`: `documentation.doxy`.
- En la carpeta `tests`: una serie de ficheros con extensión `.test`.
- En la carpeta `scripts`: una serie de scripts (ficheros con extensión `.sh`).

Puede montar su propio proyecto VSCode para la práctica `Fraud1` de forma similar a como se hizo con el proyecto `Fraud0` en la práctica anterior. Tiene varias opciones para ello:

1. Hacer una copia de su proyecto VSCode de la práctica anterior (`MiFraud0`), asignándole el nombre `MiFraud1`, proyecto que modificaremos de forma adecuada. Para ello realice los siguientes pasos:
   a) Crear el directorio para el proyecto `MiFraud1`. Por ejemplo, esto puede hacerse en VSCode mediante Menú File --> Add Folder to Workspace --> New Folder.
   b) Copiar el contenido de los directorios `include` y `src` del proyecto `MiFraud0` en el proyecto `MiFraud1`.
   c) Borrar los ficheros `ArrayLocation.h` y `ArrayLocation.cpp` del proyecto `MiFraud1`, pues no los necesitamos en esta práctica. También debemos borrar el fichero `main.cpp`, pues tendremos que sustituirlo por el nuevo de esta práctica.
   d) Incluir en el fichero `VectorLocation.h` el prototipo de los tres nuevos métodos (`nearest()`, `assign()` y `load()`) que aparecen en este módulo. Copiar su prototipo del fichero `VectorLocation.h` del proyecto `Fraud1` descargado del repositorio Github.

e) Copiar en los directorios `include` o `src` (según el caso) de tu proyecto `MiFraud1`, los nuevos ficheros `VectorInt.h`, `Clustering.h`, `VectorInt.cpp`, `Clustering.cpp` y `main.cpp` del proyecto `Fraud1` descargado del repositorio Github.

f) Copiar en `MiFraud1` las carpetas `doc`, `tests` y `scripts` del proyecto `Fraud1` descargado del repositorio Github.

g) Copiar en `MiFraud1` el fichero `CMakeList.txt` del proyecto `Fraud1` descargado del repositorio Github.

2. Hacer una copia del proyecto `Fraud1` del repositorio, asignándole el nombre `MiFraud1`, proyecto que modificaremos de forma adecuada. Para ello realice los siguientes pasos:

   a) Crear el directorio para el proyecto `MiFraud1`. Por ejemplo, esto puede hacerse en VSCode mediante Menú File --> Add Folder to Workspace --> New Folder.

   b) Copiar el contenido del directorio `Fraud1` del repositorio en el directorio de su proyecto `MiFraud1`.

   c) En su nuevo proyecto `MiFraud1` copiar el código desarrollado en la práctica anterior (clases `Location`, `VectorLocation`). O sea, la implementación de los métodos y funciones externas que hizo en `Location.cpp` y `VectorLocation.cpp`.

5.4. Módulos del proyecto

A la hora de implementar los nuevos métodos y funciones de esta práctica, fíjese en las cabeceras de tales métodos y funciones en los ficheros `*.h` correspondientes y en los comentarios que les acompañan, pues en ellos están detalladas sus especificaciones. Concretamente, debe revisar los nuevos métodos que aparecen en la clase `VectorLocation` (métodos `nearest()`, `assign()` y `load()`) y todos los métodos de las clases `VectorInt` y `Clustering`.

Nota importante: Se han retirado a propósito todos los `const` y `&` para los parámetros de los nuevos métodos y funciones externas que aparecen en los módulos comentados más arriba. También se ha retirado el calificador `const` que debe aplicarse a algunos métodos (los métodos que no modifican el objeto implícito). Debe revisar por tanto la forma en que se hace el paso de argumento para cada parámetro (por valor, por referencia o por referencia constante) y si los nuevos métodos que aparezcan en alguna clase deben calificarse o no con `const`. También se ha retirado el `const` y `&` en métodos que devuelven por referencia o referencia constante. Por ejemplo, revise los métodos `at()` de la clase `VectorInt` y el método `getCentroids()` de la clase `Clustering`.

Por tanto, revise todas las cabeceras de los nuevos métodos y funciones externas. El número de argumentos y los tipos han sido establecidos y **no se han de cambiar**.

Se le invita a definir todos los métodos y funciones externas adicionales que estime oportuno para una adecuada modularización del código.

En esta práctica debe tener en cuenta las especificaciones indicadas en los siguientes módulos:

- Clase `Location`. En esta clase no se necesita ningún cambio respecto a la práctica anterior.
- Clase `VectorLocation`. En esta clase debe implementar los nuevos métodos `nearest()`, `assign()` y `load()`.
- Clase `VectorInt`. El fichero **VectorInt.h** (ver detalles en sección 6.3) contiene la declaración de la clase `VectorInt`. Los objetos de esta clase permiten guardar un vector de números enteros (tipo `int`). En esta práctica la clase usa un array alojado en memoria automática (en la pila) de capacidad fija. En la práctica `Fraud3`, modificaremos esta clase para que pase a usar un array alojado en memoria dinámica. Como puede ver, el dato miembro `_values` es el array de enteros, que tiene una capacidad máxima de 100 elementos. El dato miembro `_size` permite conocer en todo momento el número de elementos usados en el array.

```
class VectorInt {
...
private:
    /**
     * Constant with the capacity of the array _values
     */
    static const int DIM_VECTOR_VALUES = 100;

    /**
     * Array of integers with the integer elements in this object
     */
    int _values[DIM_VECTOR_VALUES];

    /**
     * Number of elements contained in the array _values
     */
    int _size;
}; // end class VectorInt
```

- Clase `Clustering`. Consulte la sección 4, en la que se ha explicado esta clase.

```
class Clustering {
...
private:
    /**
     * Default random seed used to initialize the random number
     * generator used in the initialClusterAssignment() method.
     */
    static const unsigned long int DEFAULT_RANDOM_SEED=1761560597L;

    /**
     * Vector of Location objects. This will be the input for the
     * clustering algorithm
     */
    VectorLocation _locations;

    /**
     * Number of clusters
     */
    int _K;

    /**
     * Vector of n integers (number of elements in _locations). This will
     * be the output of the clustering algorithm.
     * The integers in this vector define the cluster number (values in the
     * interval [0 - _K-1]) for each one of the Location object in _locations
     */
    VectorInt _clusters;

    /**
     * A vector of @p _K Location objects with the centroids of each cluster.
     */
    VectorLocation _centroids;

    /**
     * It indicates whether the run() method has already been executed to
     * obtain a clustering of the set of Location objects. The run() method
     * should set this field to true when it ends its execution.
```

```
     */
    bool _isDone;

    /**
     * Seed used to initialize the random number generator (with srand(seed)) in
     * the initialClusterAssignment() method.
     */
    unsigned int _seed;

    /**
     * This is the sum of the within-cluster variances of the clusters in
     * this Clustering object.
     * The value for this field should be set at the end of run() method
     * with the method calculateSumWCV(). Before running the
     * run() method, this field will likely contain a garbage value.
     * @note The value of the sum of the within-cluster variances can also be
     * retrieved using the method calculateSumWCV().
     * This field is introduced in this class to calculate its value once,
     * and thus avoiding having to recalculate it every time it
     * is needed
     */
    double _sumWCV;

    /**
     * This field represents the number of iterations used in the KMeans
     * algorithm until no cluster change occurred. Before running the
     * run() method, this field will likely contain a garbage value.
     * The value for this field should be set at the end of run() method.
     */
    int _numIterations;

    ....

}; // end class Clustering
```

- Módulo `main.cpp`. Este programa tiene por finalidad lo descrito en la sección 5.1. Teniendo en cuenta los detalles comentados en tal sección y los pasos esbozados en el código proporcionado para esta función (sección 6.5), complete el código de este módulo.

Recuerde además corregir los fallos detectados en la práctica anterior, pues se procede con una nueva evaluación de cada uno de los métodos implementados, por lo que los errores no corregidos volverían a penalizar en esta práctica.

5.5. Entrega de la práctica

La práctica deberá ser entregada en Prado, durante el periodo habilitado en la propia actividad de entrega, y consistirá en un fichero ZIP del proyecto. El nombre del fichero puede ser cualquiera, pero tendrá la extensión zip. Por ejemplo, podría llamarse `PRACTICAMP.zip`. Para obtener este fichero se sugiere utilizar el script `runZipProject.sh` (ver el guion de `Fraud0`). Compruebe que lo entregado es compilable y operativo y, no olvide poner el nombre de los componentes del equipo en la cabecera del fichero `main.cpp`.

6. Código para la práctica

6.1. Location.h

```
/*
 * Metodología de la Programación
 * Curso 2025/2026
 */

/**
 * @file Location.h
 * @author Silvia Acid Carrillo <acid@decsai.ugr.es>
 * @author Andrés Cano Utrera <acu@decsai.ugr.es>
 * @author Luis Castillo Vidal <L.Castillo@decsai.ugr.es>
 *
 * Created on 6 de octubre de 2025, 12:27
 */

#ifndef LOCATION_H
#define LOCATION_H

#include <iostream>
#include <string>

/**
 * @class Location
 * @brief It stores data about a location where people spend money on some
 * service or product. A location contains data about the x and y position
 * (double values) in the 2D space and the name of the location
 * (a string that can contain several words).
 */
class Location {
public:
    /**
     * @brief Constructor that builds a Location object with a value of 0
     * for the x and y coordinates, and an empty string for the name.
     */
    Location();

    /**
     * @brief Returns the x-coordinate of this location
     * Query method
     * @return The x-coordinate of this location
     */
    double getX() const;

    /**
     * @brief Returns the y-coordinate of this location
     * Query method
     * @return The y-coordinate of this location
     */
    double getY() const;

    /**
     * @brief Returns the name of this location
     * Query method
     * @return The name of this location
     */
    std::string getName() const;

    /**
     * @brief Checks if the position of this object is inside the area
     * determined by the two given locations: Location A (@p bottomLeft) and
     * B (@p topRight).
     *
     *      +------------+(B)
     *      |            |
     *      |            |  +(O3)
     *      |            |
     *      |     +(O2)  |
     *      |            |
     *      |            |
     *   (A)+------------+
     *         +(O1)
     *
     * Examples of returning false:
     * - Object O3 is outside the given area (O3.x > B.x)
     * - Object O1 is outside the given area (O1.y < A.y)
     * Example of returning true:
     * - Object O2 is inside the given area
     * - Object A is inside the given area
     * - Object B is inside the given area
     *
     * Query method.
     * @param bottomLeft The Location of the bottom left point. Input parameter
     * @param topRight The Location of the top right point. Input parameter
     * @return true if the Location of this object is inside the area
     * determined by the two given Locations
     */
    bool isInsideArea(const Location &bottomLeft, const Location &topRight) const;

    /**
     * @brief Calculates the square of the Euclidean distante from this location
     * to the provided location. That is, it loc1 and loc2 are two
     * Location objects, then this method returns:
     * \f$ (loc1.x - loc2.x)^2 + (loc1.y - loc2.y)^2 \f$
```

```
     * Query method
     * @param location A Location object. Input parameter
     * @return Returns the square of the Euclidean distante from this
     * location to the provided location.
     */
    double squaredDistance(const Location& location) const;

    /**
     * @brief Calculates the Euclidean distante from this location to the provided
     * location. That is, it loc1 and loc2 are two Location objects, then
     * this method returns:
     * \f$ \sqrt{ (loc1.x - loc2.x)^2 + (loc1.y - loc2.y)^2 } \f$
     * Note that this method can be implemented using the squaredDistance()
     * method.
     * Query method
     * @param location A Location object. Input parameter
     * @return The Euclidean distante from this location to the provided
     * location.
     */
    double distance(const Location& location) const;

    /**
     * @brief Obtains a string with the x and y coordinates
     * and the name (whitespace separated). If the name is an empty string,
     * then only the x and y coordinates are included in the resulting string.
     * To convert the x and y coordinates to a string you must use the
     * std::to_string(int) C++ function.
     * Query method
     * @return string with information about this Location object
     */
    std::string toString() const;

    /**
     * @brief Sets the x-coordinate in this object
     * Modifier method
     * @param x The new value for the x-coordinate. Input parameter
     */
    void setX(double x);

    /**
     * @brief Sets the y-coordinate in this object
     * Modifier method
     * @param y The new value for the y-coordinate. Input parameter
     */
    void setY(double y);

    /**
     * @brief Sets the name in this object
     * Modifier method
     * @param name A string with the new value for the name. Input parameter
     */
    void setName(const std::string& name);

    /**
     * @brief Sets the x and y coordinates, and the the name in this object
     * Modifier method
     * @param x The new value for the x-coordinate. Input parameter
     * @param y The new value for the y-coordinate. Input parameter
     * @param name A string with the new value for the name. Input parameter
     */
    void set(double x, double y, const std::string& name);

    /**
     * @brief Reads from the provided input stream the information
     * to fill this Location object.
     * @note This method reads two double values (space separated) from the
     * input stream that are used to set the x and y coordinates of this object.
     * Then, it reads the string that can be formed from the current position of
     * the input stream to the end of the line. This string is then trimmed
     * (see the Trim(string) function) to remove its spaces at the beginning
     * and at the end. The resulting string is used to set the name of this
     * Location object. Note that the name can have several words separated by
     * whitespaces.
     * @param is Input stream. Input/output parameter
     */
    void load(std::istream& is);

private:
    /**
     * The x-coordinate of this location
     */
    double _x;

    /**
     * The y-coordinate of this location
     */
    double _y;

    /**
     * The name of this location. It can contain several words.
     */
    std::string _name;
}; // end of class Location

/**
 * Removes spaces and \t characters at the beginning and at the end of the
 * provided string @p myString. If the provided string @p myString is empty
 * or contains only spaces or \t characters then @p myString will contain an
 * empty string after calling to this function.
 * @note This function can be easily implemented using the methods
```

```
 * find_first_not_of(string) and find_last_not_of(string) of class string.
 * @param myString a string. Input/Output parameter
 */
void Trim(std::string & myString);

#endif /* LOCATION_H */
```

6.2. VectorLocation.h

```
/*
 * Metodología de la Programación
 * Curso 2025/2026
 */

/**
 * @file VectorLocation.h
 * @author Silvia Acid Carrillo <acid@decsai.ugr.es>
 * @author Andrés Cano Utrera <acu@decsai.ugr.es>
 * @author Luis Castillo Vidal <L.Castillo@decsai.ugr.es>
 *
 * Created on 11 de diciembre de 2025, 11:27
 */

#ifndef VECTORLOCATION_H
#define VECTORLOCATION_H

#include "Location.h"

/**
 * @class VectorLocation
 * @brief A VectorLocation object contains a vector of Location objects. It has
 * a capacity (maximun number of Locations that can be stored in the vector)
 * and a size (number of Locations the vector currently contains).
 * The public methods of this class do not allow a VectorLocation to contain two
 * Location objects with identical names.
 * This class uses an array of Location objects with a fixed capacity
 * to store the set of locations.
 */
class VectorLocation {
public:
    /**
     * @brief It builds a VectorLocation object (vector of Location objects)
     * with a size equal to the provided value (@p size).
     * Each element in the vector is initialized with the default Location
     * constructor.
     * @throw std::out_of_range Throws a std::out_of_range exception if
     * @p size < 0 or size>DIM_VECTOR_LOCATIONS
     * @param size The size for the vector of Location in this object. Input
     * parameter
     */
    VectorLocation(int size = 0);

    /**
     * @brief Gets the number of elements in the vector of this object
     * Query method
     * @return The number of elements
     */
    int getSize() const;

    /**
     * @brief Gets the capacity of the vector in this object
     * Query method
     * @return The capacity of the vector in this object
     */
    int getCapacity() const;

    /**
     * @brief Obtains a string with information about this VectorLocation object,
     * in the following format:
     * - First line, the number of Location objects in this vector converted to
     * a string (using the to_string(int) C++ function).
     * - For each Location, a line with its x and y coordinates, converted to
     * a string (using the to_string(int) C++ function) and separated by a
     * whitespace.
     * Query method
     * @return string with information about this VectorLocation object
     */
    std::string toString() const;

    /**
     * @brief Searches the provided Location in the array of locations in this
     * object. If found, it returns the position where it was found. If not,
     * it returns -1. We consider that position 0 is the first location in the
     * list of locations and this->getSize()-1 the last location.
     * In order to find a location consider only equality in the name field.
     * Query method
     * @param location A Location. Input parameter
     * @return If found, it returns the position where the location
     * was found. Otherwise it returns -1
     */
    int findLocation(const Location &location) const;

    /**
     * @brief Returns a VectorLocation object with those locations whose
     * positions are inside the area determined by the two given Locations.
     * Query method
     * @param bottomLeft The Location of the bottom left point. Input parameter
     * @param topRight The Location of the top right point. Input parameter
     * @return A VectorLocation with the selected Locations.
     */
    VectorLocation select(const Location &bottomLeft,
                          const Location &topRight) const;

    /**
     * @brief Removes all the elements in this object, leaving the container
     * with a size equal to 0. It only needs to set the number of elements
```

```
     * (_size field) to zero.
     * Modifier method
     */
    void clear();

    /**
     * @brief Gets a const reference to the Location element at the given
     * position
     * Query method
     * @throw std::out_of_range Throws an std::out_of_range exception if the
     * given position is not valid.
     * @param pos position in the VectorLocation object. Input parameter
     * @return A const reference to the Location element at the given position
     */
    const Location &at(int pos) const;

    /**
     * @brief Gets a reference to the Location element at the given position.
     * Modifier method
     * @throw std::out_of_range Throws an std::out_of_range exception if the
     * given position is not valid
     * @param pos position in the VectorLocation object. Input parameter
     * @return A reference to the Location element at the given position.
     */
    Location &at(int pos);

    /**
     * @brief Appends a copy of the given Location object at the first free
     * position in the array of Location in this object. The location is
     * not appended to this object if it was already found in this object.
     * @throw std::out_of_range Throws a std::out_of_range exception if the
     * provided location is going to be appended but the array of Location
     * was full (its capacity was full). If the provided location is not going
     * to be appended because it was already found in this object or its name
     * is an empty string, then no exception is thrown.
     * Modifier method
     * @param value the new Location object to be appended. Input parameter
     * @return true if the given Location could be inserted in this
     * VectorLocation object; false otherwise (the location was already found
     * in this object)
     */
    bool append(const Location &location);

    /**
     * @brief Appends to this VectorLocation object, the list of
     * Location objects contained in the provided VectorLocation object
     * that are not found (using VectorLocation::findLocation(Location)) in
     * this object.
     * This method could be implemented with the help of the method
     * VectorLocation::append(const Location & location), to append to this
     * object, the Locations of the provided VectorLocation object.
     * Modifier method
     * @param crimeSet A VectorLocation object. Input parameter
     */
    void join(const VectorLocation &locations);

    /**
     * Sorts the array of locations in this object by increasing alphabetical
     * order of the name of its location (a string).
     * Modifier method
     */
    void sort();

    /**
     * @brief Gets the position in this vector of the Location object nearest to
     * the provided location
     * Query method
     * @param location A Location object. Input parameter
     * @return the position of the Location object nearest to the provided
     * location.
     * If returns -1 if this vector is empty
     */
    int nearest(Location location);

    /**
     * Assigns the provided value to all the elements in this vector
     * Modifier method
     * @param location A Location object. Input parameter
     */
    void assign(Location location);

    /**
     * @brief Reads from the provided input stream the information
     * to fill this VectorLocation object.
     * @note This method should remove any Location previously contained in the
     * provided VectorLocation object. See files *.loc in the folder DataSets
     * as examples of this kind of file.
     * @note This operator throws an exception in some error cases (see below).
     * Before throwing the corresponding exception, this method clears
     * the object (it calls to clear() method) to leave the object in a
     * consistent state.
     * Modifier method
     * @throw std::out_of_range Throws a std::out_of_range exception if the
     * number of Location read from the input stream is negative.
     * @throw std::out_of_range Throws a std::out_of_range exception if the
     * number of locations read from the input stream is greater than the capacity
     * of this VectorLocation object.
     * @param is Input stream. Input/output parameter
     */
    void load(std::istream is);
```

```
private:
    /**
     * Constant with the capacity of the array _locations
     */
    static const int DIM_VECTOR_LOCATIONS = 100;

    /**
     * Pointer to a dynamic array of Locations
     */
    Location _locations[DIM_VECTOR_LOCATIONS];

    /**
     * Number of Location objects contained in the dynamic array _locations
     */
    int _size;
}; // end of class VectorLocation

#endif /* VECTORLOCATION_H */
```

6.3. VectorInt.h

```
/*
 * Metodología de la Programación
 * Curso 2025/2026
 */

/**
 * @file VectorInt.h
 * @author Silvia Acid Carrillo <acid@decsai.ugr.es>
 * @author Andrés Cano Utrera <acu@decsai.ugr.es>
 * @author Luis Castillo Vidal <L.Castillo@decsai.ugr.es>
 *
 * Created on 30 de julio de 2025, 9:54
 */

#ifndef VECTORINT_H
#define VECTORINT_H

#include <iostream>
#include <string>

/**
 * @class VectorInt
 * @brief A VectorInt object contains a vector of integer elements. It has
 * a capacity (maximun number of elements that can be stored in the vector)
 * and a size (number of elements the vector currently contains).
 */
class VectorInt {
public:

    /**
     * @brief It builds a VectorInt object (vector of integers) with a
     * size and capacity equal to the provided value (@p size). Each element
     * will be filled with a value equal to 0.
     * @throw std::out_of_range Throws a std::out_of_range exception if
     * @p size < 0 or size>DIM_VECTOR_VALUES
     * @param size The size for the vector of integers in this object. Input
     * parameter
     */
    VectorInt(int size=0);

    /**
     * @brief Gets the number of elements in the vector of this object
     * Query method
     * @return The number of elements
     */
    int getSize();

    /**
     * @brief Gets the capacity of the vector in this object
     * Query method
     * @return The capacity of the vector in this object
     */
    int getCapacity();

    /**
     * @brief Compares the integer vectors of this object and the provided
     * object, and returns the number of identical elements in both of them. The
     * comparison is performed in order, element by element, in both vectors.
     * For example, given the following two vectors:
     * 2 1 2 3 5
     * 1 1 2 4 5
     * this method will return 3 (there is a match in positions 1, 2 and 4)
     * @throw std::invalid_argument Throws an std::invalid_argument exception
     * if the sizes of this and the provided object are different
     * Query method
     * @param other A VectorInt object. Input parameter
     * @return The number of identical elements in the vectors of this
     * and the provided object.
     */
    int countIdenticalElements(VectorInt other);

    /**
     * @brief Obtains a string with information about this VectorInt object,
     * in the following format:
     * - First line, the number of elements in this vector.
     * - Second line, the elements in this vector, separated by a whitespace.
     * Take into account that a '\n' should appear after the last element
     * instead of a whitespace.
     * Query method
     * @return string with information about this VectorInt object
     */
    std::string toString();

    /**
     * @brief Gets the Euclidean distance between this and the provided object.
     * The Euclidean distance between two points \f$ P=(p_1, p_2, ... , p_n) \f$
     * and \f$ Q=(q_1, q_2, ... , q_n) \f$ in an n-dimensional space \f$ R^n \f$
     * is the length of the line segment connecting them, calculated as the
     * square root of the sum of the squared differences of their components:
     * \f$ d(P,Q)=\sqrt{ \sum_{i=1}^{n} (p_i-q_i)^{2} } \f$
     * @throw std::invalid_argument Throws an std::invalid_argument exception
     * if the size of this object and the provided one are not equal
     * @throw std::invalid_argument Throws an std::invalid_argument exception
     * if the size of the provided object is zero
     * Query method
     * @param other A VectorInt. Input parameter
     * @return The Euclidean distance between this and the provided objects
```

```
     */
    double distance(VectorInt other);

    /**
     * @brief Assigns the provided value to all the elements in this vector
     * Modifier method
     * @param value An integer value. Input parameter
     */
    void assign(int value=0);

    /**
     * @brief Appends the given integer value at the end (first free position)
     * of the array of integers in this object.
     * @throw std::out_of_range Throws a std::out_of_range exception if the
     * array of Location was full (its capacity was full).
     * Modifier method
     * @param value the new integer value to be appended. Input parameter
     */
    void append(int value);

    /**
     * @brief Removes all the elements in this object, leaving the container
     * with a size equal to 0. It only need to set the number of elements
     * (_size field) to zero.
     * Modifier method
     */
    void clear();

    /**
     * @brief Gets a const reference to the integer element at the given
     * position
     * Query method
     * @param pos position in the VectorInt object. Input parameter
     * @throw std::out_of_range Throws an std::out_of_range exception if the
     * given position is not valid.
     * @return A const reference to the integer element at the given position
     */
    int at(int pos);

    /**
     * @brief Gets a reference to the integer element at the given position.
     * Modifier method
     * @param pos position in the VectorInt object. Input parameter
     * @throw std::out_of_range Throws an std::out_of_range exception if the
     * given position is not valid
     * @return A reference to the integer element at the given position.
     */
    int at(int pos);

private:
    /**
     * Constant with the capacity of the array _values
     */
    static const int DIM_VECTOR_VALUES = 100;

    /**
     * Array of integers with the integer elements in this object
     */
    int _values[DIM_VECTOR_VALUES];

    /**
     * Number of elements contained in the array _values
     */
    int _size;

}; // end of class VectorInt

#endif /* VECTORINT_H */
```

6.4. Clustering.h

```
/*
 * Metodología de la Programación
 * Curso 2025/2026
 */

/**
 * @file Clustering.h
 * @author Silvia Acid Carrillo <acid@decsai.ugr.es>
 * @author Andrés Cano Utrera <acu@decsai.ugr.es>
 * @author Luis Castillo Vidal <L.Castillo@decsai.ugr.es>
 *
 * Created on 22 de octubre de 2025, 11:57
 */

#ifndef CLUSTERING_H
#define CLUSTERING_H

#include "VectorInt.h"
#include "VectorLocation.h"

/**
 * @class Clustering
 * @brief This class is used to obtain a clustering of a given set of locations
 * (Location objects). A clustering algorithm groups the input locations
 * into K disjoint groups (clusters). The clustering algorithm used in this
 * class is the K-Means algorithm. Each group will consist of a disjunct subset
 * of the original location set and a representative location (centroid), which
 * is calculated as the vector of the means of the locations in the group. The
 * run() method in this class executes the algorithm. An example of use is the
 * following:
VectorLocation locations;
// Suppose here code to fill the vector of locations
Clustering clustering;
clustering.set(locations, 5);
clustering.run();
 */
class Clustering {
public:
    /**
     * @brief Constructor that builds a Clustering object initializing
     * the field (_K) for the number of clusters with 0; the field (_isDone)
     * that tells whether the run() method has already been executed
     * with false; the fields _sumWCV and _numIterations with 0 and the field
     * _seed with DEFAULT_RANDOM_SEED.
     * The fields (_locations, _clusters, _centroids) are initialized with
     * the default constructor of their classes.
     */
    Clustering();

    /**
     * @brief Gets the number of clusters
     * Query method
     * @return The number of clusters
     */
    int getK();

    /**
     * @brief Gets the current vector of centroids
     * Query method
     * @return A const reference to the vector of centroids (a VectorLocation
     * object).
     */
    VectorLocation getCentroids();

    /**
     * @brief Indicates whether the clustering algorithm (run() method) has
     * already been executed for this Clustering object.
     * Query method
     * @return true if the run() method has already been executed for this
     * Clustering object; false otherwise
     */
    bool isDone();

    /**
     * @brief Gets the number of locations in the set of locations of this
     * Clustering object.
     * Query method
     * @return The number of locations in the set of locations of this
     * Clustering
     */
    int getNumLocations();

    /**
     * @brief Gets the cluster number for the Location determined by the
     * provided integer (@p locationIndex)
     * Query method
     * @param locationIndex An integer that determines the position of a
     * location in the vector of Location objects. Input parameter
     * @return The cluster number for the Location determined by
     * the provided integer (@p locationIndex). If the
     * clustering algorithm has not been run (_isDone is false), then it returns
     * -1.
     */
    int clusterOf(int locationIndex);

    /**
     * @brief Gets the value of the sum of the within-cluster variances of this
```

```
     * Clustering. This value is obtained from the field _sumWCV
     * @note The returned value will likely be a garbage value if the run()
     * method has not yet been executed (_isDone is false)
     * Query method
     * @return A double with the sum of within-cluster variances of this
     * Clustering
     */
    double getSumWCV();

    /**
     * @brief  Gets the number of iterations used in the KMeans
     * algorithm until no cluster assignment change occurs.
     * It obtains the number of iterations (field _numberIterations) used
     * in the KMeans algorithm until no cluster change occurred.
     * @note The returned value will likely be a garbage value if the run()
     * method has not yet been executed (_isDone is false)
     * Query method
     * @return The number of iterations used in the KMeans algorithm
     */
    int getNumIterations();

    /**
     * @brief  Gets a string with information about the provided cluster (index
     * of a cluster). That string contains a line for each location in the given
     * cluster. The information of each location consists of an integer with
     * its location index (whithin the _location field), a whitespace, and
     * the information returned by the Location::toString() method for that
     * location. An example is the following (a cluster with three locations):
0 3.300000 9.100000 Graduate College
6 12.600000 17.500000 Admission Information Center
7 12.500000 20.000000 Rockefeller College
     * Query method
     * @param cluster An integer, that defines the index of a cluster. Input
     * parameter
     * @return Gets a string with information about the provided cluster. If the
     * clustering algorithm has not been run (_isDone is false) or an
     * invalid value of cluster is provide, then it returns an empty string.
     */
    std::string clusterInfo(int cluster);

    /**
     * @brief Obtains a string with the some statistics data about this
     * Clustering object.
     * The first line contains the value of K; the second line contains the
     * value of the sum of within-cluster variances; the third line contains
     * the number of iterations used in the KMeans algorithm.
     * Query method
     * @return A string with the content described above
     */
    std::string getStatistics();

    /**
     * @brief Indicates whether this Clustering object is equivalent to the
     * provided Clustering object (@p other). Two Clustering objects are
     * considered equivalent if they have the same K value,
     * the same sum of within-cluster variances and the same number of
     * iterations in the KMeans algorithm.
     * If either this Clustering object or the provided Clustering object
     * (@p other) has not yet run the clustering algorithm (the run() method),
     * then they are not considered equivalent.
     * Query method
     * @param other A Clustering object. Input parameter
     * @return true if this Clustering object is equivalent to the provided
     * Clustering object (@p other); false otherwise
     */
    bool isEquivalentTo(Clustering other);

    /**
     * @brief Obtains a string with the fields of this Clustering object.
     * The first line contains the value of K; the second line contains the
     * value of the sum of within-cluster variances; the third line contains
     * the number of iterations used in the KMeans algorithm.
     * In the next three lines, information about the cluster number for each
     * location.
     * If follows information about each centroid
     * Next, for each cluster, information about its locations
     * Here, an example is shown:
K=5
Sum of within-cluster variances: 161.136889
Number of iterations: 5
Cluster number for each location:
21
0 2 2 2 2 2 1 1 1 1 1 4 4 4 4 4 4 4 4 4 3
Centroids:
5
3.300000 9.100000
14.880000 21.280000
12.600000 9.460000
18.500000 6.700000
26.088889 15.366667
Cluster 0 information:
0 3.300000 9.100000 Graduate College
Cluster 1 information:
6 12.600000 17.500000 Admission Information Center
7 12.500000 20.000000 Rockefeller College
8 13.700000 22.800000 Art Museum Store
9 17.000000 23.300000 Princeton University Store
10 18.600000 22.800000 Labyrinth Books
Cluster 2 information:
1 11.800000 6.900000 Wawa Public Restrooms
```

```
2 12.600000 7.300000 Princeton Station
3 10.100000 9.700000 Forbes College
4 13.300000 12.200000 Dinky Bar & Kitchen
5 15.200000 11.200000 Whitman College
Cluster 3 information:
20 18.500000 6.700000 Yeh College
Cluster 4 information:
11 20.900000 15.400000 Frist Campus Center
12 23.700000 14.800000 Cannon Dial Elm
13 24.800000 14.900000 Quadrangle
14 25.600000 14.900000 Ivy
15 26.400000 14.900000 Cottage
16 27.300000 14.500000 Cap & Gown
17 28.200000 15.000000 Cloister
18 29.100000 15.200000 Charter
19 28.800000 18.700000 Engineering Quadrangle
     * Query method
     * @return A string with the content described above
     */
    std::string toString();

    /**
     * @brief Sets the vector of locations (_locations), the value of K (_K)
     * and the seed (_seed) with the provided values.
     * The fields _isDone, _clusters and _centroids_ are also initialized
     * appropriately: _isDone is set to false; _clusters is initialized with
     * a VectorInt object with size equal to the number of locations;
     * _centroids is initialized with a VectorLocation object with size K;
     * the fields _sumWCV and _numIterations are set to 0.
     * Modifier method
     * @throw std::invalid_argument Throws an std::invalid_argument exception
     * if K<=0
     * @param locations The vector of locations to used in the clustering
     * algorithm. Input parameter
     * @param K The parameter K for the K-Means algorithm. Input parameter
     * @param seed The seed used to initialize the random number generator
     * (with srand(seed)) in the initialClusterAssignment() method.
     * Input parameter
     */
    void set(VectorLocation locations, int K,
        unsigned int seed=DEFAULT_RANDOM_SEED);

    /**
     * @brief Run the clustering algorithm. The clustering algorithm implemented
     * in this method is the KMeans algorithm.
     * After running this method the following postcondition should be verified:
     * @post The _centroids field should contain the vector of the means of
     * locations in each group
     * @post _clusters (a VectorInt object) should contain the cluster number
     * for each location in the vector of locations (_locations)
     * @post The _isDone field should be set to true
     * @post The _sumWCV field should be set to the sum of the within-cluster
     * variances using the claculateSumWithinClusterVariances() method.
     * @post The _numberIterations field should be set to the number of
     * iterations used by the KMeans algorithm.
     *
     * A brief description of the KMeans algorithm is the following:
     *
     * -# Perform an initial assignment of a cluster to each of the Location
     * objects. This must be done by initializing the random number
     * generator using srand(seed);
     * Then, for each location i, assign cluster number rand() % _K
     * to that location. Note that this step is done by modifying the _clusters
     * field.
     * -# Repeat while any location changes its cluster number:
     *     - For each cluster, recalculate its centroid (update _centroids
     *       field).
     *     - Assign each location to a cluster (update _clusters field).
     *       Each location is assigned to the cluster that provides
     *       the shortest distance to its centroid.
     *
     * Query method
     */
    void run();

private:
    /**
     * Default random seed used to initialize the random number
     * generator used in the initialClusterAssignment() method.
     */
    static const unsigned int DEFAULT_RANDOM_SEED=1761560597U;

    /**
     * Vector of Location objects. This will be the input for the
     * clustering algorithm
     */
    VectorLocation _locations;

    /**
     * Number of clusters
     */
    int _K;

    /**
     * Vector of n integers (number of elements in _locations). This will
     * be the output of the clustering algorithm.
     * The integers in this vector define the cluster number (values in the
     * interval [0 - _K-1]) for each one of the Location object in  _locations
     */
    VectorInt _clusters;
```

```
    /**
     * A vector of @p _K Location objects with the centroids of each cluster.
     */
    VectorLocation _centroids;

    /**
     * It indicates whether the run() method has already been executed to
     * obtain a clustering of the set of Location objects. The run() method
     * should set this field to true when it ends its execution.
     */
    bool _isDone;

    /**
     * Seed used to initialize the random number generator (with srand(seed)) in
     * the initialClusterAssignment() method.
     */
    unsigned int _seed;

    /**
     * This is the sum of the whithin-cluster variances of the clusters in
     * this Clustering object.
     * The value for this field should be set at the end of run() method
     * with the method calculateSumWCV(). Before running the
     * run() method, this field will likely contain a garbage value.
     * @note The value of the sum of the within-cluster variances can also be
     * retrieved using the method calculateSumWCV().
     * This field is introduced in this class to calculate its value once,
     * and thus avoiding having to recalculate it every time it
     * is needed
     */
    double _sumWCV;

    /**
     * This field represents the number of iterations used in the KMeans
     * algorithm until no cluster change occurred. Before running the
     * run() method, this field will likely contain a garbage value.
     * The value for this field should be set at the end of run() method.
     */
    int _numIterations;

    /**
     * @brief Performs an initial assignment of a cluster to each one of the
     * Location objects. This method begins by initializing the random number
     * generator using:
     * srand(seed);
     * Then, for each location i, assign cluster number rand() % _K to that
     * location.
     * Modifier method
     */
    void initialClusterAssignment();

    /**
     * @brief Calculates the sum of the within-cluster variances of this
     * Clustering object.
     * The within-cluster variance of a cluster C_j is calcultated with:
     * $\sum_{p_i \in C_j} squaredDistance(p_i, centroid_{C_j})$
     * @return A double with the sum of within-cluster variances of this
     * Clustering
     * Modifier method
     */
    double calculateSumWCV();
}; // end of class Clustering

#endif /* CLUSTERING_H */
```

6.5. main.cpp

```
/*
 * Metodología de la Programación
 * Curso 2025/2026
 */

/**
 * @file main.cpp
 * @author Silvia Acid Carrillo <acid@decsai.ugr.es>
 * @author Andrés Cano Utrera <acu@decsai.ugr.es>
 * @author Luis Castillo Vidal <L.Castillo@decsai.ugr.es>
 *
 * Created on 24 de octubre de 2025, 9:27
 */

#include <iostream>

#include "VectorLocation.h"
#include "Clustering.h"

using namespace std;

/**
 * The purpose of this program is to read a set of locations from
 * standard input, to cluster them into K clusters using the K-means
 * algorithm, and to show the resulting clustering on the standard output.
 *
 * This program first reads an integer K from standard input, which indicates
 * the number of clusters to be used in the clustering process. Then, it reads
 * a set of Location objects from standard input. After reading the data, it
 * performs the K-means clustering algorithm to group the locations into K
 * clusters. Finally, it displays the resulting clustering information,
 * including the cluster assignments for each location and the centroids of
 * each cluster.
 *
 * Be careful to show the output as in the below example.
 *
 * An example of data read from standard input is the following (see file
 * Datasets/dataP1/princeton.p1in):
5
21
3.3 9.1 Graduate College
11.8 6.9 Wawa Public Restrooms
12.6 7.3 Princeton Station
10.1 9.7 Forbes College
13.3 12.2 Dinky Bar & Kitchen
15.2 11.2 Whitman College
12.6 17.5 Admission Information Center
12.5 20 Rockefeller College
13.7 22.8 Art Museum Store
17 23.3 Princeton University Store
18.6 22.8 Labyrinth Books
20.9 15.4 Frist Campus Center
23.7 14.8 Cannon Dial Elm
24.8 14.9 Quadrangle
25.6 14.9 Ivy
26.4 14.9 Cottage
27.3 14.5 Cap & Gown
28.2 15 Cloister
29.1 15.2 Charter
28.8 18.7 Engineering Quadrangle
18.5 6.7 Yeh College
 *
 * Running sintax:
 * > build/Fraud1 <<inputFile.p1in>
 *
 * Running example:
 * > build/Fraud1 < ../Datasets/dataP1/princeton.p1in
K=5
Sum of within-cluster variances: 161.136889
Number of iterations: 5
Cluster number for each location:
21
0 2 2 2 2 2 1 1 1 1 1 4 4 4 4 4 4 4 4 4 3
Centroids:
5
3.300000 9.100000
14.880000 21.280000
12.600000 9.460000
18.500000 6.700000
26.088889 15.366667
Cluster 0 information:
0 3.300000 9.100000 Graduate College
Cluster 1 information:
6 12.600000 17.500000 Admission Information Center
7 12.500000 20.000000 Rockefeller College
8 13.700000 22.800000 Art Museum Store
9 17.000000 23.300000 Princeton University Store
10 18.600000 22.800000 Labyrinth Books
Cluster 2 information:
1 11.800000 6.900000 Wawa Public Restrooms
2 12.600000 7.300000 Princeton Station
3 10.100000 9.700000 Forbes College
4 13.300000 12.200000 Dinky Bar & Kitchen
5 15.200000 11.200000 Whitman College
Cluster 3 information:
20 18.500000 6.700000 Yeh College
Cluster 4 information:
```

```
11 20.900000 15.400000 Frist Campus Center
12 23.700000 14.800000 Cannon Dial Elm
13 24.800000 14.900000 Quadrangle
14 25.600000 14.900000 Ivy
15 26.400000 14.900000 Cottage
16 27.300000 14.500000 Cap & Gown
17 28.200000 15.000000 Cloister
18 29.100000 15.200000 Charter
19 28.800000 18.700000 Engineering Quadrangle
 */
int main(int argc, char* argv[]) {
    VectorLocation locations; // VectorLocation object to store the input locations
    Clustering clustering; // Clustering object
    int K; // number of clusters

    // Read de number of clusters

    // Read the locations directly into the VectorLocation object

    // Define the parameters of the clustering object

    // Execute the clustering algorithm

    // Show the resulting clustering in the standard output

    return 0;
}
```