Práctica 6. Mallas regulares

Sesiones de prácticas: 2

Objetivos

Utilizar mallas regulares para minimizar el tiempo de búsqueda de los coches más cercanos a un punto de recarga.

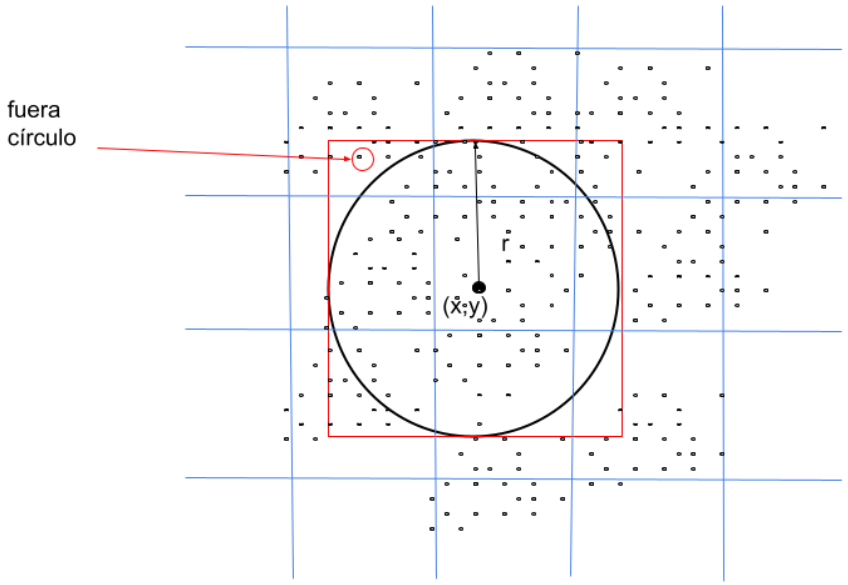
Descripción de la EEDD

La aplicación desarrollada hasta ahora ha usado las estructuras de datos más adecuadas en cada proceso, sin embargo, no se ha tenido en cuenta el uso de las coordenadas UTM. En esta práctica vamos a utilizar una malla regular para almacenar los coches. Se puede seguir la implementación de la Lección 17, pero haciendo que la función de búsqueda devuelva objetos de tipo T (en vez de T*).

```
template <Typename T>
class MallaRegular {
            …
public:
      MallaRegular(float aXMin, float aYMin, float aXMax, float aYMax, int
nDiv);
….
      vector<T> buscarRadio(float xcentro, float ycentro, float radio);
      T buscar (float x, float y, const T &dato);
      unsigned maxElementosPorCelda();
      float promedioElementosPorCelda();
}
```

Esta clase tiene la funcionalidad de la Malla Regular dada en la Lección 17, pero añadiendo las funciones definidas arriba. Las funciones *maxElementosPorCelda()* y *promedioElementosPorCelda()* sirven de métrica para conocer la carga y distribución de puntos en la estructura y decidir así el tamaño de ésta. La función buscarRadio() devuelve todos los coches dentro de un *radio* centrado en un punto con coordenadas *(xcentro,ycentro)*. Podemos asumir que todo tipo T con el que se instancie la clase tiene los métodos getX() y getY() implementados, que en nuestro caso son respectivamente la longitud y la latitud en coordenadas UTM del coche. Para realizar esta operación, se considera el cuadrado de búsqueda que engloba al círculo y luego se calcula la distancia al centro. Se deben considerar sólo los puntos dentro del círculo, como se indica en el dibujo. Las coordenadas de las cuatro

esquinas del cuadrado que contiene al círculo son: (x+r,y+r), (x+r,y-r), (x-r,y-r) y (x-r,y+r). Para conocer la equivalencia entre kms y coordenadas terrestres basta con saber que un grado de longitud o latitud equivale a 111.1 kms.



Por otro lado, para calcular en kms la distancia entre dos puntos con coordenadas terrestres, se debe considerar la esfericidad de la tierra. Para ello se puede utilizar el algoritmo de *Haversine* (mirar Harversine.pdf adjunto a la práctica) que devuelve los kilómetros entre dos coordenadas terrestres. El algoritmo de conversión se describe a continuación:

```
Input: (lat1, lon1) (lat2, lon2)  (dos puntos en coord.
terrestres)
Output: d  (distancia entre los dos puntos en kms)
begin
      R = radio medio de la Tierra (6.378kms)
      IncrLat = (lat2 - lat1)*(PI/180)
      IncrLon = (lon2 - lon1)*(PI/180)
      a = sin² (IncrLat/2) + cos (lat1*(PI/180))*
      cos(lat2*(PI/180))* sin²(IncrLon/2)
      c = 2 * atan2 (sqrt (a), sqrt (1-a))
      d = R * c
end
```

Para más información visitar:

https://www.genbeta.com/desarrollo/como-calcular-la-distancia-entre-dos-puntos-geograficos-en-c-formula-de-haversine

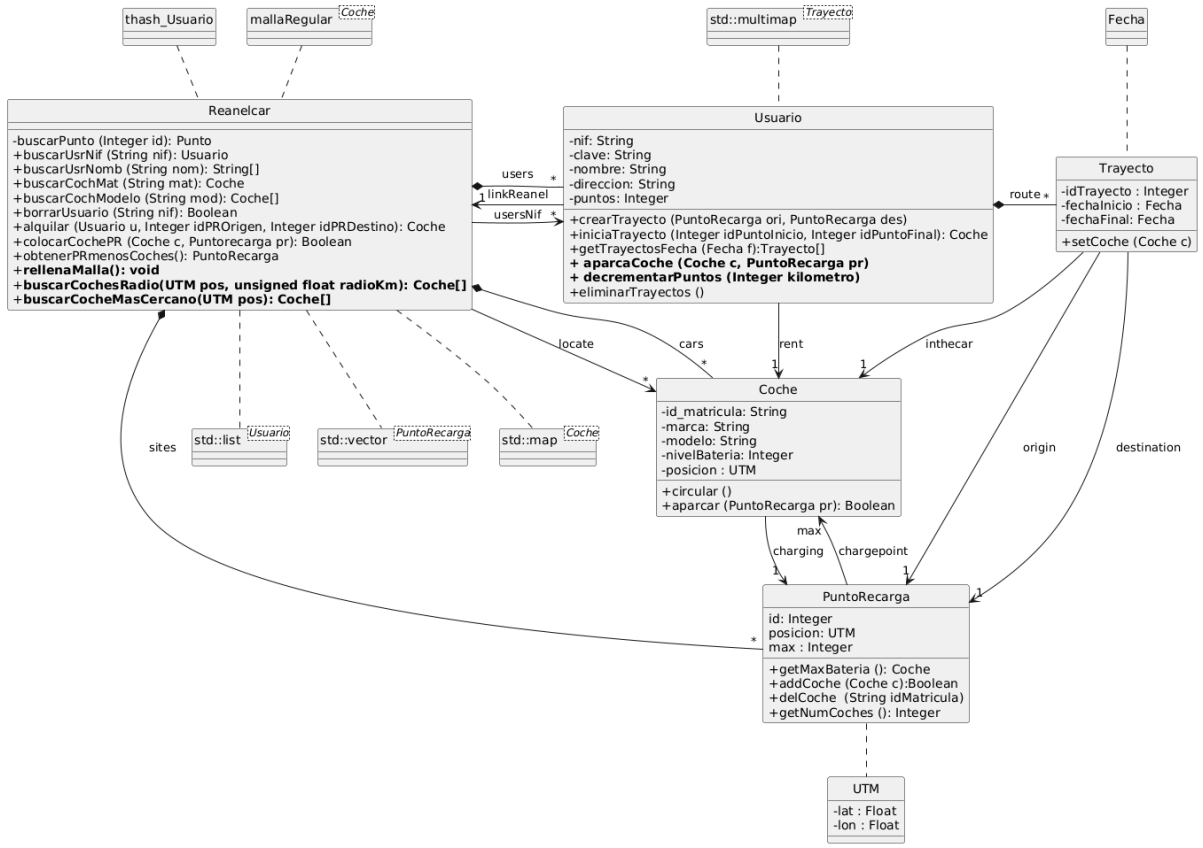
Descripción de la práctica:

En esta práctica se va a añadir una relación doble entre *Reanelcar* y *Coche*. En concreto, mantendremos la relación de composición *Reanelcar::cars* y añadiremos una nueva relación de asociación, *Reanelcar::locate*, que será implementada mediante una malla regular que posiciona los distintos coches a partir de su posición (*UTM*).
La malla se crea a partir de las coordenadas antes citadas instanciadas a objetos tipo *UTM*. Se consideran *(aXmin, aYmin)* las coordenadas X e Y más pequeñas localizadas al leer el fichero *"destinos.csv"* y *(aXmax, aYmax)* las coordenadas máximas en X e Y respectivamente, también obtenidas de ese fichero. El número de casillas en la malla vendrá determinado por el promedio de coches por casilla una vez insertados los coches en la malla, debiendo estar entre 10-15. Por lo tanto, se debe probar con diferentes números de casillas de la malla hasta obtener un promedio de coches por casilla entre 10-15. Además, se debe calcular cuántos coches tiene la casilla más poblada.

En el siguiente esquema UML se representa la nueva funcionalidad. El constructor de la clase *Reanelcar* se encargará de generar todo el sistema. Para construir la malla regular es necesario cargar previamente todos los coches en la estructura *cars* y, tras esto, seguir las indicaciones del programa de prueba, ya que la malla regular no se rellenará hasta que se comiencen a aparcar los coches. Estos coches se irán introduciendo en la malla conforme se vayan aparcando. El proceso de aparcamiento del coche vendrá especificado en el programa de prueba. La malla debe instanciarse como *MallaRegular<Coche*>* para albergar las direcciones de memoria de los *Coches* aparcados.

Reanelcar:

- *Reanelcar::rellenaMalla()*, Método privado para insertar los coches una vez aparcados en la malla.
- *Reanelcar::buscarCochesRadio(UTM pos, unsigned float radioKm) : Coche[]*. Dada una posición devuelve un vector de coches que se encuentran en un determinado radio.
- *Reanelcar::buscarCocheMasCercano(UTM pos): Coche[]*. Dada una posición, devuelve el coche más cercano a dicha posición.



Los contenedores a usar para las siete relaciones “uno a muchos” son:

- ReanelCar::cars → std::map<string, Coche> con clave la matrícula
- ReanelCar::users → std::list<Usuario>
- ReanelCar::sites → std::vector<PuntoRecarga>
- Reanelcar::usersNif → ThashUsuario
- **PuntoRecarga::chargepoint → std::multimap<int, Coche*>** con clave el nivel de batería.
- Usuario::route → std::multimap<Fecha, Trayecto> con clave la fecha de inicio
- **Reanelcar::locate → MallaRegular<Coche*>**

En la práctica anterior, se añadió a la clase *Usuario* el atributo *puntos,* el cual estará inicializado a 100, y para esta práctica se irá decrementando 2 puntos por cada kilómetro de distancia entre la posición actual del coche y el punto de recarga destino. Para ello el método *Usuario::aparcaCoche()* recibe el punto de recarga destino y si es diferente llama a *Usuario::decrementarPuntos()* para decrementar los puntos correspondientes.

Programa de prueba:

Crear un programa de prueba con las siguientes indicaciones:

1. Instanciar Reanelcar con los datos de los ficheros usuarios, coches y puntos de recarga. Adaptar los métodos de Reanelcar existentes para trabajar con las nuevas estructuras.
2. Implementar la malla regular como un template según la Lección 17 añadiendo a Reanelcar los métodos *buscarCochesRadio() y buscarCocheMasCercano()* descritos anteriormente.
3. Leer el fichero *"destino.csv"* y cada posición leída almacenarla en un objeto de tipo UTM e introducirla en un vector doble manteniendo la relación entre el usuario y el objeto UTM. A continuación, determinar la latitud y longitud mínima y máxima para realizar la creación de la malla regular, teniendo en cuenta que se van a introducir en ella aproximadamente un total de 1489 coches.
4. Todos los coches se distribuirán entre los puntos de recarga siguiendo la misma lógica que en las prácticas anteriores. En esta ocasión, todos los coches tendrán un punto de recarga asociado y no será necesario multiplicar por dos el máximo de coches de cada punto de recarga, ya que el fichero ha sido modificado de tal manera que todos los coches se coloquen en los puntos de recarga correspondientes.
5. Buscar todos los usuarios que comienzan por "A" y hacer que alquilen un coche de la misma forma que en las prácticas anteriores. Para la creación del trayecto, se inicializará el *idTrayecto* de forma incremental y la fecha de inicio será el día de 25/11/2024. La fecha de fin se calculará sumando 1 ó 2 días de forma aleatoria a la fecha actual. Mostrar los 10 primeros usuarios con todos sus datos, incluido el trayecto y coche alquilado.
6. Buscar todos los usuarios que comienzan por "B" y alquilar un coche realizando el mismo procedimiento que en el apartado anterior.
7. De los usuarios que han alquilado un coche, los que empiezan por "B" van a aparcar el coche en el punto de recarga de destino creado al crear el trayecto en el apartado anterior. Una vez aparcado, el coche se introducirá en la malla regular. Mostrar los trayectos de los 10 primeros usuarios con todos sus datos (origen, destino, coche alquilado y fecha), así como los puntos acumulados.

8. De los usuarios que han alquilado un coche, los que empiezan por "A" van a aparcar el coche en la posición leída del fichero *"destino.csv"*. Para ello, de forma secuencial, cada usuario que tenga que aparcar su coche, accede al vector de posiciones creado en el apartado 3 y lo aparca en la posición que corresponde a cada usuario (primera posición del vector). Una vez aparcado, el coche se introducirá en la malla regular. Tras esto, eliminará dicha posición del vector. Mostrar los trayectos de los 10 primeros usuarios con todos sus datos (origen, destino, coche alquilado y fecha), así como los puntos acumulados.
9. Buscar los coches aparcados en un radio de 10 kms de Jaén capital (Longitud: -3.7902800, Latitud: 37.7692200). ¿Cuál es el coche más cercano a Jaén capital (Longitud: -3.7902800, Latitud: 37.7692200)?
10. Mostrar los datos del coche más cercano al punto de recarga con ID 43.
11. ¿Qué punto de recarga tiene más coches en un radio de 15km de Jaén capital? Ahora para el punto de recarga con más coches, comprueba los coches que están en el rango de 25 km. A cada coche, se tiene que quitar 2 puntos por cada kilometro de distancia y eliminar del sistema a los usuarios que aparquen a una distancia mayor a 20 km del punto de recarga con más coches.

Para los que trabajan por parejas:

Comprobar el número de coches que hay en la posición (37.79143, -3.77716) y si el número de coches es mayor al número de elementos promedio por celda, crear un nuevo punto de recarga en esa posición y añadirlo al vector.

VOLUNTARIO: Visualización 2D con la clase Img

Para poder visualizar el resultado de forma gráfica, se adjunta a esta práctica la clase **Img**. Esta clase es en realidad una matriz de píxeles, creada tomando como parámetro el tamaño de un recuadro en número de píxeles en (tamaFilas, tamaColumnas). Esta clase es capaz de dibujarse como imagen de modo que cada consulta generará un fichero imagen resultado con *Img::guardar()*. Ejecutar la función *main()* que aparece junto al código para comprobar el funcionamiento y visualizar la imagen resultante. En la imagen adjunta a la práctica se puede ver la disposición espacial de todos los aeropuertos.

Para que esta clase sea útil en nuestro caso, hacemos coincidir las esquinas de la imagen con las del cuadro de trabajo donde se incluyen todas las coordenadas de las imágenes. Consideramos, pues, que la esquina inferior izquierda de nuestros datos es la latitud y longitud mínima de las posiciones de los coches aparcados y la esquina superior derecha es la

latitud y longitud máxima de los coches que están aparcados. En el ejemplo de prueba se pinta un recuadro y se pinta también un pixel azul.

Utilizar esta clase para dibujar con distinto color cada uno de los puntos de recarga. De igual forma, dibujar todos los coches que están aparcados en otro color. Todo esto debe mostrarse en una imagen de 600*600.

Estilo y requerimientos del código:

1. El código debe ser claro, tener un estilo definido y estar perfectamente indentado, para ello se pueden seguir algunos de los estilos preestablecidos para el lenguaje C++ (http://geosoft.no/development/cppstyle.html).

2. Deben comprobarse todas los posibles errores y situaciones de riesgo que puedan ocurrir (desbordamientos de memoria, parámetros con valores no válidos, etc.) y lanzar las excepciones correspondientes, siempre que tenga sentido. Leer el tutorial de excepciones disponible en el repositorio de la asignatura en docencia virtual.

3. Se valorará positivamente la calidad general del código: claridad, estilo, ausencia de redundancias, etc.