Guía para el desarrollo del proyecto

1 – Introducción

En este documento se detalla la funcionalidad de cada clase y de todos sus métodos y atributos para que cualquier desarrollador pueda seguir con el desarrollo de la aplicación. El documento está estructurado por clases sobre las cuales se habla de todo lo necesario para una mejor comprensión del código realizado. Finalmente, se exponen varias mejoras realizables al proyecto y extensiones de funcionalidades. Antes de leer este documento se recomienda la lectura del *paper* (informe final) del proyecto para conocer de que trata y como funciona el programa.

2 – Detallado de las clases

2.1 – Edge.h/cpp

Clase destinada a la representación de las aristas del grafo. Cada instanciación de esta clase corresponde a una arista del grafo creado. Esta clase se compone por los siguientes atributos y métodos:

Atributos

Vertex* Origen, Destino: apuntadores a los nodos origen y destino de la arista.

Double dist, desnivel: variables que almacenan la distancia y el desnivel del tramo.

Bool stop: variable utilizada por los algoritmos de búsqueda para saber si se ha pasado por esta arista o no.

Edge* complteGraphEdge: apuntador que utilizan las aristas del grafo simplificado para referenciar a la arista que sustituye en el inicio del tramo entre nodos bifurcación. Se utiliza para la reconstrucción del camino encontrado.

Edge* reverseEdge: apuntador a la arista reversa a esta.

Métodos

Únicamente se compone de dos **constructores**, uno utilizado para la creación del grafo completo y el segundo para la creación del grafo simplificado.

2.2 – Vertex.h/cpp

Clase destinada a la representación de los nodos del grafo. Cada instanciación de esta clase corresponde a un nodo del grafo creado. Esta clase se compone por los siguientes atributos y métodos:

Atributos

Int id: número que identifica únicamente a cada nodo.

List<Edge*> Edges: lista de aristas que tienen este nodo como nodo origen.

Double lat, lon, x, y, elevación: variables que almacenan la georreferenciación del nodo y su elevación.

Double DijkstraDistance: variable utilizada por el algoritmo de Dijkstra para encontrar el mejor camino posible.

Edge* antecesor: apuntador utilizado por Dijkstra para reconstruir el camino en el algoritmo Greedy, que apunta a la arista por la cual se ha llegado a este nodo.

Bool salesVisit: variable para marcar si este nodo tiene que ser visitado obligatoriamente por el algoritmo de búsqueda.

Bool saveGraphVisit: variable que se utiliza para guardar el grafo en un fichero, que indica si el nodo actual ya ha sido guardado.

Double sortFactor: variable que indica el factor de ordenación de las aristas del nodo.

Métodos

Constructor del nodo con todo lo necesario para inicializar todos sus atributos.

Ordena(): función encargada de ordenar las aristas del nodo dependiendo del caso seleccionado por el usuario y el factor calculado por el algoritmo de busqueda.

2.3 – Graph.h/cpp

Clase destinada a la representación del grafo. Esta se encarga de almacenar todos los nodos y aristas del grafo y contiene todas las funciones necesarias para su creación y la interactuación con este.

Atributos

Vector<list<Vertex*>> Vértices: Vector de listas de vértices (nodos) que almacena todos los nodos del grafo en el cuadrante que le corresponda a cada uno. Cada lista de nodos corresponde a un cuadrante determinado.

List<Edge> Edges: lista que contiene todas las aristas del grafo.

Float R: constante que contiene la distancia del radio de la tierra en km para calcular distancias entre puntos.

Float tram_Dif: distancia mínima en km entre un nodo guardado y el siguiente para que se de como valido y se analice.

Float point_dif: distancia en metros con la que un nodo se considera duplicado si se encuentra a menos de esta distancia de otro nodo anteriormente guardado.

Int divCuadrante: contante que indica la cantidad de filas y columnas en las que se divide el mapa para crear los diferentes cuadrantes.

Int pathLength: longitud máxima en cuanto a nodos se refiere para poder encontrar un camino alternativo entre dos vértices del grafo.

Métodos

Constructor y **destructor** del grafo para una correcta inicialización y finalización de este.

Clear(): función encargada de resetear todos los valores del grafo.

NewVertex(): función que crea un nuevo nodo en el cuadrante correcto y lo devuelve.

SearchById(): función que busca un nodo en el grafo según su id y lo devuelve en caso de que lo encuentre. Sino devuelve *nullptr*.

DeleteSection(): función que elimina del grafo una sección de nodos que ha sido detectada como duplicada.

NewEdge(): existen dos variantes de esta. Una principal que crea una arista del grafo completo, y una secundaria que crea una arista del grafo simplificado a la cual se le pasan todos los datos de la nueva arista.

AnotherPath(): función que busca en el grafo si existe un camino alternativo entre dos nodos al que se está analizando para posteriormente comprobar si este es duplicado.

DuplicatedPath(): función que coge los caminos encontrados por la función anotherPath(), y comprueba si el camino analizado es una duplicidad de uno de estos. Esto lo hace calculando la distancia de cada camino encontrado y en caso de que la diferencia de distancias con el analizado sea menor a 20 metros, el analizado será dado como duplicado.

Read(): función principal de la lectura de las rutas que utiliza la librería QtXml para moverse por el archivo leído. Esta función se encarga de llamar a todas las funciones necesarias y realizar las comprobaciones adecuadas para crear el grafo correctamente, eliminando todas las duplicidades posibles. Esta función es recursiva para una lectura correcta de los archivos y poder leer ficheros con mas de una ruta en caso de que lo contenga.

Load(): función que inicia la lectura de cada fichero seleccionado por el usuario llamando a la función read() para cada uno de ellos.

DuplicatedEdge(): función que comprueba si una arista ya ha sido creada y existe en el grafo.

Dist(): función que calcula la distancia entre dos puntos en coordenadas geográficas decimales.

Duplicado(): función principal que determina si un nodo esta duplicado. Esta función llama a las funciones necesarias para ello y comprueba si existe un nodo en el grafo que se encuentre a menos de 11 metros del nodo nuevo que quiere crearse. En caso de que así sea, se considera duplicado y se devuelve el nodo por el que se ha considerado duplicado.

CuadrantesCercanos(): función que comprueba si un nodo que se esta analizando para sabe si es duplicado, se encuentra a menos de 11 metros de otro cuadrante para analizar los nodos de este también.

Cuadrante(): función que devuelve el cuadrante al que pertenece un nodo.

Buscar(): función que busca en un cuadrante concreto si el nodo se considera duplicado.

IsInPath(): función que determina si el pixel clicado por el usuario sobre el mapa es un pixel perteneciente al grafo, con un magnetiso de 10 pixeles entre el nodo mas cercano i el pixel clicado.

2.4 – Graph_Object.h/cpp

Esta clase es la encargada de realizar la conexión entre la interfaz y la clase grafo para poder acceder a todos sus datos e interactuar con ella. Contiene las funcionalidades de iniciar la lectura de las rutas, determinar si un píxel clicado pertenece al grafo, guardar la ruta, guardar el grafo, buscar la ruta en el grafo y mostrar el grafo y la ruta buscada. Esta clase utiliza threads para algunas de sus acciones.

Atributos

Graph graph, simplifiedGraph: intanciaciones de la clase grafo las cuales representan al grafo completo y al grafo simplificado respectivamente.

List<Vertex*> visitar, simplifiedVisits: listas de apuntadores que contienen los nodos seleccionados por el usuario como origen, fin y paradas intermedias.

List<QString> paths: lista de rutas a los archivos que el usuario selecciona los cuales contienen los tracks o rutas a leer.

Track t: ruta encontrada por el algoritmo de búsqueda.

Int act: variable que determina la acción que se va a ejecutar en la inicialización de un nuevo thread.

Double distanciaMax, denivelMax: variables que contienen la distancia máxima y el desnivel acumulado máximo del grafo creado después de leer las rutas seleccionadas.

Float searchDist, searchElev: distancia y desnivel seleccionados por el usuario para realizar la búsqueda de la ruta.

Métodos

Signals: conjunto de funciones que emiten señales a la interfaz para enviar datos en un momento exacto de la ejecución del programa y poder realizar así la conexión con esta.

Run(): función heredada de la librería QThread, que contiene el código que se ejecuta en el lanzamiento de un nuevo thread.

Load(): trata los paths seleccionados por el usuario e inicia un thread el cual ejecuta la lectura de los ficheros en la clase graph para crear el grafo.

IsInPath(): comprueba gracias a los métodos de la clase graph si el pixel seleccionado pertenece al grafo y si es así, almacena el nodo en la lista de visitas.

Conjunto de **getters** necesarios para que la interfaz pueda acceder a los datos que necesita y mostrarlos por pantalla de forma correcta.

SearchTrack(): función que instancia un buscador de rutas con los parámetros seleccionados por el usuario e inicia un thread el cual ejecuta la búsqueda de la ruta en el grafo.

Show(): función que transforma los nodos de cada arista del grafo y del camino encontrado, de coordenadas geográficas a coordenadas pixel de la imagen, y emite una señal para cada arista, con la que se envían los datos calculados de cada nodo del grafo para poder mostrarlos sobre el mapa.

Clear(): resetea todas las variables de esta clase y de la clase graph.

ClearParadas(): elimina las paradas intermedias seleccionadas por el usuario de la lista de visitas.

ResetTrack(): resetea todos los datos de la ruta encontrada.

SaveTrack(): función que crea un nuevo archivo GPX en el directorio seleccionado por el usuario con la ruta encontrada por el algoritmo de búsqueda.

SaveGraph(): función que crea un nuevo archivo GPX en el directorio seleccionado por el usuario, el cual contendrá el grafo creado e inicia la escritura del grafo en el archivo creado.

SaveGraphRecursive(): función recursiva que escribe cada tramo entre nodos bifurcaciones del grafo en el archivo. Cada vez que se llega a un nodo bifurcación se lanza una llamada recursiva a esta función para cada arista saliente de este en caso de que el destino no haya sido ya visitado anteriormente, haciendo así posible que el grafo se guarde como pequeñas rutas independientes.

CreateSimplifiedGraph(): función que inicia la creación del grafo simplificado llamando a su función recursiva con la cual se crea un grafo con únicamente los nodos que son bifurcaciones y aquellos que se quieran visitar.

CreateSimplifiedGraphRecursive(): función recursiva la cual crea una arista entre cada par de nodos que son bifurcaciones o si uno de ellos es un nodo a visitar. Esta función lanza una llamada recursiva a esta para cada arista saliente al llegar a uno de estos nodos, siempre y cuando, ese camino no haya sido ya creado anteriormente.

CreateSimplifiedVisits(**):** función que transforma las visitas seleccionadas por el usuario las cuales corresponden al grafo completo, a visitas del grafo simplificado para poder realizar la búsqueda sobre este.

CalculateGraphParams(): calcula la distancia y desnivel máximos del grafo creado.

2.5 – Track.h/cpp

Clase que almacena la ruta encontrada por el algoritmo de búsqueda.

Atributos

Double dist, desnivel: variables que contienen la distancia y el desnivel acumulado de la ruta completa.

List<Edge*> Edges: lista de aristas que forman la ruta.

Métodos

Constructor que inicializa la ruta.

AddFirst(): función que añade una nueva arista a la lista por delante.

AddLast(): función que añade una nueva arista a la lista por detrás.

Clear(): función que resetea los datos de la ruta.

Calcular(): calcula la distancia y el desnivel de la ruta completa.

2.6 – TrackSearcher.h/cpp

Clase dedicada a la busqueda de la ruta deseada en el grafo. Esta clase contiene los algoritmos de búsqueda y todos los parámetros necesarios para realizar la búsqueda de la ruta lo más precisa y rápidamente posible.

Atributos

Double distance, elevation: distancia y desnivel acumulado seleccionado por el usuario para la busqueda de la ruta.

Double bestDistAct, bestElevationAct: variables que almacenan la mejor distancia y desnivel acumulado encontrados en la búsqueda de la ruta con respecto a los parámetros seleccionados por el usuario.

Stack<Edge*> finalTrack: pila de aristas que acaban formando la mejor ruta encontrada por el algoritmo.

Vertex* last: nodo destino del camino buscado.

Int numSections: numero de secciones que debe tener el camino final encontrado si se cuenta que una sección corresponde a un tramo entre dos nodos visita.

Bool found: variable que determina si se ha encontrado una ruta valida para poder detener la ejecución del algoritmo.

Clock_t timer: variable que contiene la hora del momento de la iniciación de la búsqueda.

Métodos

Constructor que inicializa los valores de forma que elige un algoritmo de búsqueda para ser utilizado.

SearchTrack(): función que llama al algoritmo de búsqueda seleccionado por el constructo iniciando así la búsqueda de la ruta.

Marcar(): marca los nodos de la lista de visitas como nodos a visitar.

DijkstraQueueMod(): algoritmo de Dijkstra que utiliza una cola de prioridad para poder encontrar el mejor camino con paradas intermedias.

TrackGreedy(): algoritmo de búsqueda Greedy el cual se utiliza en caso de que el usuario no haya seleccionado ningún parámetro de distancia ni desnivel. Este algoritmo utiliza Dijkstra para encontrar la solución y únicamente devuelve una ruta final que pasa por todos los nodos seleccionados, con un rendimiento muy elevado.

TrackBacktracking(): función que inicializa todo lo necesario para la ejecución de Backtracking. Esta función se utiliza siempre que el usuario seleccione una distancia, un desnivel o ambas cosas que debe tener la ruta encontrada. La función primeramente ejecuta el algoritmo Greedy para inicial el Backtracking con una ruta base y inicializar con los valores de esta ruta las variables de bestDistAct y bestElevationAct. Ordena las aristas de los nodos según la distancia y desnivel seleccionados e inicia la busqueda. Al finalizar reconstruye el camino y lo devuelve.

Backtracking(): algoritmo Backtracking el cual es un algoritmo recursivo que ejecuta una búsqueda en árbol empezando por el nodo inicial de la lista de visitas. Este comprueba que no se ha excedido el tiempo de búsqueda, si es así, detiene la ejecución y devuelve el mejor

camino encontrado hasta el momento. Por el contrario, comprueba si el camino actual puede podarse ya que su distancia y desnivel respecto a los seleccionados son peores que los mejores encontrados hasta el momento. Si el camino no es podado, se comprueba si se ha llegado al nodo destino y el camino ha pasado por todos los nodos seleccionados. En caso afirmativo, se comprueba si su distancia y desnivel son mejores que la mejor distancia y desnivel encontrados hasta el momento. Si es así, se actualizan las variables y en caso de la distancia y desnivel encontrados tengan una diferencia menor a uno con las seleccionadas por el usuario, la ejecución se detiene y se devuelve este camino encontrado, sino se sigue buscando. En caso de no haber llegado al nodo final o no haber visitado todos los nodos necesarios, se realiza una llamada recursiva a Backtracking() para todas aquellas aristas salientes del nodo actual no visitadas hasta el momento, y el proceso se repite hasta finalizar la ejecución por tiempo o por encontrar una ruta valida.

Ordena(): función que calcula el factor por el cual se ordenan las aristas de cada nodo y inicia su ordenación. Este factor es una aproximación de que distancia y desnivel puede tener una arista en la ruta final a raíz del camino encontrado con Greedy y los parámetros seleccionados por el usuario.

CamiConstruct(): función que transforma la pila de aristas que representa la ruta encontrada a un Track con todas las aristas de la pila.

ReconstructComplete(): función que reconstruye la ruta de forma que pasa de ser una ruta del grafo simplificado a una ruta del grafo completo.

2.7 – UTM.h/cpp

Conjunto de funciones que transforman las coordenadas geográficas decimales a UTM y viceversa.

LLtoUTM(): convierte las coordenadas geográficas decimales a UTM.

UTMtoLL(): convierte las coordenadas UTM a geográficas decimales.

2.8 – Map.h/cpp

Clase que contiene el mapa cargado por el usuario y todos los datos necesarios para la georreferenciación de sus pixeles. Esta clase utiliza el patrón de diseño *singleton* para así poder instanciar la clase desde varios lugares y poder acceder a la misma información asegurando que no se crea otro mapa.

Atributos

Map* mapa: variable que contiene la instanciación de Map asegurando así que solamente existe una.

Cv::Mat mapImage: matriz de OpenCv que contiene la imagen cargada.

Cv::Mat printedImage: matriz de OpenCv que contiene la imagen mostrada por pantalla.

Cv::Mat scaledMapImage: matriz de OpenCv que contiene la imagen escalada de la cargada originalmente.

Double x_size, y_size: tamaño de un píxel de la imagen con respecto a las coordenadas UTM.

Double x_coordinate, y_coordinate: coordenadas UTM de la esquina superior izquierda de la imagen.

Double xFin_coordinate, yFin_coordinate: coordenadas UTM de la esquina inferior derecha de la imagen.

Double scaled_ratio: escalado que se le aplica a la imagen para ser mostrado por la pantalla.

Double latI, lonI: coordenadas geográficas decimales de la esquina superior izquierda de la imagen.

Double latF, lonF: coordenadas geográficas decimales de la esquina inferior derecha de la imagen.

Double difLat, difLon, difX, difY: tamaño de un cuadrante en coordenadas geográficas decimales y UTM.

Int screenWidth, screenHeight: ancho y alto de la pantalla en la que se ejecuta la aplicación.

Int croppedX, croppedY: valor de la esquina superior derecha de la imagen mostrada por pantalla.

Métodos

Constructor del mapa inicializando todas las variables de forma correcta.

GetInstance(): método que devuelve la instancia de Map en caso de haber sido creada, y que la crea en caso de que no.

SetMap(): función que lee el mapa seleccionado por el usuario y el archivo TFW e inicializa todas las variables de georreferenciación con respecto a este.

SetScaleRatio(): función que calcula el factor de escala de la imagen.

Getters para acceder a las matrices que contienen los mapas.

ToPixel(): función que convierte las coordenadas UTM de un punto a coordenadas pixel de la imagen.

ToCoordinate(): función que convierte las coordenadas pixel de la imagen a coordenadas UTM.

CuadranteLL(): función que calcula el cuadrante al que pertenece un punto en coordenadas geográficas decimales.

CuadranteUTM(): función que calcula el cuadrante al que pertenece un punto en coordenadas UTM.

2.9 – MapProvider.h/cpp

Clase encargada de enviar la imagen necesaria en cada momento a la interfaz para que esta sea mostrada.

Atributos

Int width, height: ancho y alto actual de la imagen mostrada por pantalla.

Int zoom: zum que tiene aplicado la imagen.

Int screenWidth, screenHeight: ancho y alto de la pantalla en la que se ejecuta la aplicación.

Int croppedX, croppedY: valor de X e Y de la esquina superior izquierda de la imagen mostrada por pantalla sobre la imagen leída inicialmente.

Int widthInc: incremento y disminución del ancho en cada zum realizado.

Int* scaledCroppedX, scaledCroppedY: valor de X e Y de la esquina superior izquierda de la imagen mostrada por pantalla sobre la imagen escalada.

Double scale: escala aplicada a la imagen inicial.

Map* mapa: instancia de la clase Map.

Cv::Mat* mapImage: apuntador al mapa leído inicialmente.

Cv::Mat* printedImage: apuntador a la sección de mapa mostrado por pantalla.

Cv::Mat* scaledMapImage: apuntador al mapa inicial escalado según el zum.

Métodos

Constructor que inicializa todos los valores para un funcionamiento correcto de la clase.

RequestImage(): función que realiza la conexión con la interfaz y ejecuta la operación que el usuario desea realizar.

Initialization(): inicializa el mapa ejecutando la lectura e inicializando todos los valores necesarios para la georreferenciación.

Drag(): realiza la acción de arrastras haciendo posible que el usuario pueda moverse con libertad de movimiento. Se realiza de forma que se aumenta o disminuye el valor de la X y Y de la esquina superior izquierda, se recorta la imagen inicial y se escala a la medida de la pantalla.

ZoomIn(): realiza la acción de aumentar zum en la imagen. Para ello disminuye el ancho de el trozo de imagen mostrado por pantalla, se calcula el alto proporcional, se recorta la imagen y se escala a las medias de la pantalla.

ZoomOut():realiza la acción de disminuir zum en la imagen. Para ello aumenta el ancho de el trozo de imagen mostrado por pantalla, se calcula el alto proporcional, se recorta la imagen y se escala a las medias de la pantalla.

2.10 – Loader_Object.h/cpp

Clase creada para poder realizar un zum y un arrastrado de las imágenes mas eficientes a causa de las imágenes pesadas con las que se trabaja. Con esta clase las operaciones se pueden realizar con nuevos threads.

Atributos

Map *mapa: instancia de la clase Map.

Int caso, zoom, widthInc: acción que se desea realizar, zum actual y incremento del ancho que se debe aplicar en el zum.

Cv::Mat* mapImage: apuntador a la imagen leída originalmente

Cv::Mat* scaledMapImage: apuntador a la imagen escalada del mapa original.

QString path: ruta al archivo que contiene el mapa que se debe leer.

Métodos

ZoomInOut(): función que se encarga de escalar el mapa original para realizar el zum en la imagen.

Initialization(): función que inicia la inicialización del mapa.

Load(): función que ejecuta el thread creado, la cual redirecciona el flujo de la ejecución según la acción que se desea realizar.

LoadMap(): función que realiza la conexión con la interfaz para iniciar la inicialización.

MapAction(): función que realiza la conexión con la interfaz para iniciar la realización de un aumento o disminución del zoom.

2.11 – Main.cpp

Archivo principal del programa el cual inicial la aplicación de forma correcta.

2.12 – Main.qml

Archivo qml el cual contiene todos los elementos de la interfaz y todas las acciones necesarias para un funcionamiento correcto de esta.

3 – Mejoras

- Aumentar la precisión de la creación del grafo corrigiendo pequeños errores que este tiene.
- Aumentar la precisión y el rendimiento del algoritmo de búsqueda para poder generar mejores rutas.
- Permitir al usuario cargar varios trozos de mapa y que el programa sea capaz de ordenarlos de forma correcta y unirlos en uno solo.
- Permitir la lectura de ficheros de elevaciones para aumentar la precisión en el cálculo de desniveles.
- Permitir mostrar un mapa de calor sobre el mapa actual después de haber leído el fichero de elevaciones para poder visualizar los desniveles del terreno en el mapa.