Lyon Poussette GPS : Schraen Matthias / Dufour Tim / Coron Valentin / Donadey Remi

Découpage Fonctionnel

Fichier principal (Projet_Poussette.py)

Fenêtre principale

classe Mainwindow() :

méthodes de classes :

- initWidget(self) → initialise l'ensemble des widgets sur la fenêtre principale et les place

- loop(self) → lance la boucle principale d'attente des événements de la fenêtre

- bouton_param_av(self, event) → fonction callback pour le bouton des paramètres avancés. Lance l'ouverture de la fenêtre des paramètres avancés

- bouton_change_iti(self, event) → fonction callback pour le bouton qui permet de changer d'itinéraire. Ferme la fenêtre trajet si elle est ouverte et affiche le menu d'accueil de la fenêtre principale.

- start_research(self, event) → fonction callback du bouton « calcul itinéraire ». Lance le calcul de l'itinéraire, change les widgets de la fenêtre principale.
Appelle : setup_adjacence_param, compute_iti, draw_iti_gene

- load_all_datas(self) → charge les données (une seule fois en début de programme, lors de l'ouverture de la fenêtre car assez couteux de charger l'ensemble des fichiers).
Appelle : charger_donnees

- open_window_trajet(self, event) → fonction callback du bouton « commencer itinéraire ». Ouvre la fenêtre trajet lorsque celui-ci est calculé.

- bouton_stop_iti(self, event) → fonction callback du bouton « arrêter itinéraire ». Ferme la fenêtre trajet si elle est ouverte et affiche le menu d'accueil de la fenêtre principale.

- effacer_prop_start/end(self, event) → fonction callback qui permet d'effacer les propositions par défaut (« départ » et « arrivée ») lorsque l'utilisateur veut écrire dans les champs d'entrée de destination.

Fenêtre Paramètres avancés

classe TopLevelParams(racine) :

Méthodes de classes :

- initWidget(self) → initialise l'ensemble des widgets et les place sur la fenêtre des paramètres avancés.

- bouton_valider(self, event) → fonction callback du bouton valider. Enregistre les paramètres sélectionnés et ferme la fenêtre paramètres

- bouton_reinitialiser(self, event) → fonction callback du bouton réinitialiser. Remet les paramètres par défauts mais ne ferme pas la fenêtre.

Fenêtre Trajet

classe TopLevelTrajet(racine):

Méthodes :

- initWidget(self) → initialise l'ensemble des widgets sur la fenêtre et les place.
Appelle : show_iti, compute_cross

- show_iti(self, routes) → dessine le carrefour en question, les routes qui sont à ce carrefour et mise en évidence de la route empruntée par l'itinéraire.

- precedent(self, event) → fonction callback du bouton « précédent ». Mise à jour de l'affichage sur la fenêtre pour visualiser le carrefour précédent.
Appelle : show_iti, compute_cross

- suivant(self, event) → fonction callback du bouton « suivant ». Mise à jour de l'affichage sur la fenêtre pour visualiser le carrefour suivant.
Appelle : show_iti, compute_cross

Fichier secondaire

Load_Files.py → fichier avec les méthodes pour charger les fichiers contenant les données relatives aux routes et chemins ainsi que les différents algorithmes de calcul pour trouver le chemin le plus court ou afficher la carte et le trajet fidèlement à la réalité.

Méthodes :

- charger_donnees() → charge dans le programme les fichiers nécessaires pour créer le dictionnaires des tronçons (identifiés par un tuple (codefuv, codetroncon) unique à chacun) avec leurs caractéristiques et calculer le dictionnaire d'adjacence de ces tronçons (indispensable à notre algorithme pour calculer le chemin le plus court).
Retourne : le dictionnaire des tronçons et le dictionnaire d'adjacence

- compute_iti(adjacence_rues, depart_fuv_troncon, arrivee_fuv_troncon) → algorithme qui va calculer le chemin le plus court entre un départ et une arrivée dans un dictionnaire d'adjacence préalablement modifié pour contenir seulement les tronçons qui respectent les conditions de l'utilisateur.
Appelle : algo_a_star, dist_between_points, setup_adjacence_param
Retourne : l'itinéraire trouvé

- compute_cross(adjacence_rues, fuv_troncon_precedent, fuv_troncon_suivant) → cherche les tronçons adjacents au nœud en cours et leurs coordonnées GPS.
Retourne : les tronçons adjacents au nœud

- algo_a_star(start_fuv_troncon, end_fuv_troncon, adj_rue) → on va utiliser l'algorithme A* pour trouver le chemin le plus court entre le point de départ et le point d'arrivé de l'utilisateur. En effet, il semble plus optimisé dans notre cas de graphe « spatial ». En effet on connaît facilement une direction optimale à suivre, la distance en ligne droite « à vol d'oiseau » entre le départ et l'arrivée.
Retourne : la liste des codes fuv+tronçon emprunté par l'itinéraire calculé

- dist_between_points(start_fuv_troncon, end_fuv_troncon, adj_rue) → permet de retourner la distance entre ces 2 points « à vol d'oiseau ».
Retourne : la distance « à vol d'oiseau »

- calcul_min_max_xy(dico_sommets) → détermine les coordonnées min et max en x et y des points dans le plan du canevas
Appelle : xy_from_lat_long
Retourne : les x et y min et max

- xy_from_lat_long(latitude, longitude) → effectue la conversion de la latitude et longitude en coordonnées x, y dans le plan du canevas sans normalisation
Retourne : les coordonnées x et y calculées en fonction de la latitude et longitude mis en paramètre

- xy_repere_cartesien (latitude, longitude, offset1, offset2) → effectue le changement de repère complet de la longitude d'une ville à sa position normalisée dans le repère (x,y) du canevas veillant à respecter les bordures
Retourne : les valeurs des coordonnées normalisées dans le plan du canevas
Appelle : xy_from_lat_long

-setup_adjacence_param(adjacence_ini, params_user*) → permet de retourner le dictionnaire d'adjacence des codes FUV qui respectent les conditions choisies par l'utilisateur sur l'application.
Retourne : le nouveau dictionnaire d'adjacence avec seulement les routes qui respectent les critères

- give_troncon_nearest_address([str] address) → permet de trouver le code tronçon + FUV le plus proche de l'adresse écrite par l'utilisateur.
Retourne : le code troncon+FUV (dictionnaire d'adjacence) associé à l'adresse de l'utilisateur

- give_troncon_nearest_gps([list] co_gps) → permet de trouver le code tronçon+FUV le plus proche des coordonnées GPS données par l'utilisateur.
Retourne : le code troncon+FUV (dictionnaire d'adjacence) associé aux coordonnées GPS

- draw_iti_gene([list] coord_gpd_iti) → dessine sur un fond de carte du Grand Lyon l'itinéraire général avec le module Geopandas. Intègre cette carte dans un widget compatible avec tkinter
Retourne : widget graphique matplotlib compatible avec tkinter.

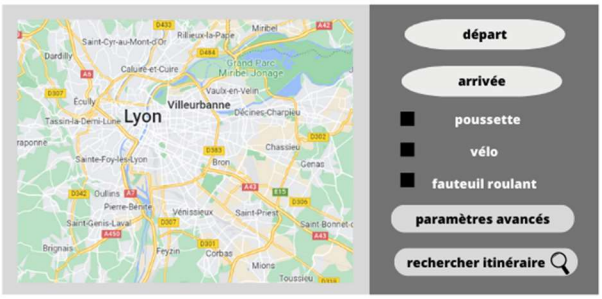
- coords_carrefour_iti(code troncon+FUV) → Permet de récupérer les coordonnées GPS du carrefour en cours ainsi que des routes qui sont reliées à ce carrefour pour pouvoir tracer pour tracer les différentes parties du carrefour.
Retourne : l'ensemble des coordonnées pour tracer les différentes parties du carrefour dans un canvas

- turn_rigth_direction_co([list] coords_canvas) → Permet d'appliquer une rotation aux coordonnées du carrefour et surtout des rues adjacentes pour que celui-ci soit toujours affiché dans le sens montant pour faciliter la compréhension (tronçon sur lequel est actuellement l'utilisateur vertical et venant du bas de la fenêtre).
Retourne : l'ensemble des coordonnées pour tracer les différentes parties du carrefour dans un canvas, dans la bonne direction

Annexes

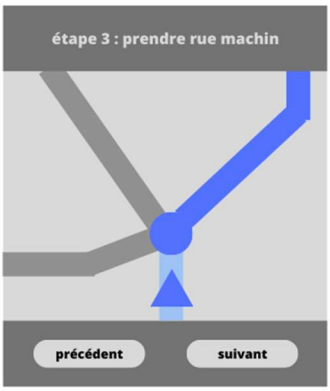
Aperçu du *graphique user interface (GUI)* :

fenêtre principale : acceuil

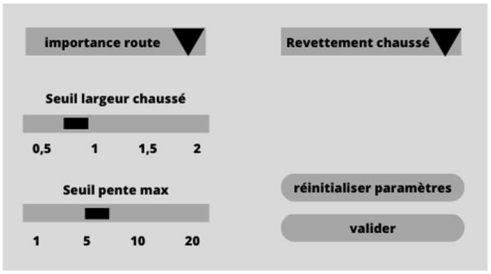


fenêtre principale : proposition itinéraire

fenêtre trajet : étape 3



fenêtre paramètres :



Description des intéractions/actions de l’utilisateur :