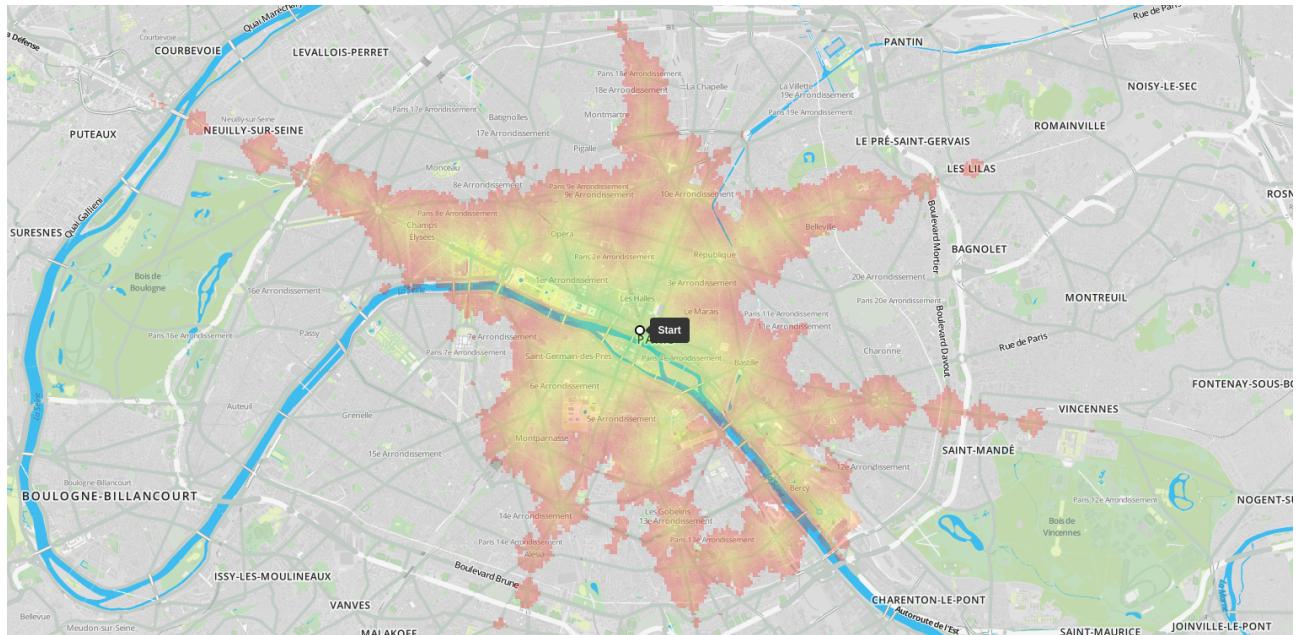


RAPPORT DE STAGE INGÉNIEUR

4 AVRIL 2016 — 26 AOÛT 2016

Création d'une API isochrone



Zoé BRUNET
Option : urbaniSTIC

Tuteur entreprise : Guillaume PINOT
tutrice école : Myriam SERVIÈRES

Table des matières

Remerciements	2
Introduction	3
I Contexte du stage	4
I.1 Présentation de Kisio Digital	4
I.1.1 Présentation général	4
I.1.2 La méthode agile	5
I.1.3 L'information voyageur	6
I.2 Navitia une API open source	7
I.2.1 Les API	7
I.2.2 L'open source, l'open data et l'open service	7
I.2.3 Le logiciel Navitia	8
I.3 Présentation du stage	9
II Déroulement du stage	10
II.1 Définition des nouvelles API	10
II.1.1 Les isochrones	10
II.1.2 Les problématiques d'interface	11
II.1.3 La réponse de l'API	13
II.2 Algorithmie	18
II.2.1 Pseudo code	19
II.2.2 La géométrie sphérique	19
II.2.3 Résolution mathématiques	20
II.3 Les performances	22
II.3.1 Méthodologie	22
II.3.2 Influence du trie	23
II.3.3 Echantillonnage	24
II.3.4 Distance et projection	26
II.4 La visualisation	26
II.4.1 Navitia Explorer	27
II.4.2 Navitia Playground	28
Conclusion	31
Table des figures	32
Bibliographie	33

Remerciements

Je tiens à remercier mon maître de stage Guillaume Pinot pour sa disponibilité, son sens de la pédagogie et ses démonstrations de yoyo en pause et en grooming. Je souhaite remercier également Stephan Simart ou "chef" pour les intimes, ses remarques abondantes en démo et son point de vue marketing ont été bénéfique pour ce stage.

Je remercie tout particulièrement toute l'équipe de RO pour son accueil bienveillant. Grâce à leur curiosité et à leur conseils je ne me suis jamais sentie trop bloquée. J'ai aimé la bonne ambiance au sein de l'équipe qui m'a permis de me sentir à l'aise tout au long de mon stage. Merci spécialement à Antoine et Pierre-Etienne pour m'avoir laissé gagner chaque fois que nous faisions la course à la piscine. Merci également à XL pour ces quizz C++, ses cours sur Pokemon Go et pour ne pas m'avoir laissé toute seule dans les bureaux en août.

Merci à Nicolas Sirletti pour ses rappels sur le cour de traitement d'images et à Antoine Frediani pour son expertise Qgis.

Enfin je remercie l'ensemble des professeurs de l'option urbaniSTIC pour cette année d'enseignement. J'ai découvert des problématiques inter disciplinaires qui m'ont réellement intéressé et qui ont remis en question la vision que j'avais de la ville.

Introduction

L'option urbaniSTIC a pour but d'enseigner à ses élèves comment utiliser de façon optimale les outils informatiques utiles dans les villes contemporaines. Au cours des sept mois passés au sein de cette formation j'ai pu m'interesser à diverses thématiques : politiques et problématiques urbaines, modélisation et représentation de la ville, acquisition et gestion de données. Lors de nombreux cours il a été souligné l'importance des transports en commun intra et inter urbain pour assurer la cohésion d'une ville. C'est pourquoi j'ai jugé intéressant d'effectuer un stage dans une entreprise s'intéressant aux systèmes d'information voyageurs (SIV).

Kisio Digital est un éditeur SaaS de solutions d'information voyageur multimodales. En plus de son domaine d'activité, c'est l'implication de cette entreprise dans l'open source, l'open service et l'open data qui m'a donné envie d'y faire mon stage. J'ai donc rejoint l'équipe de recherche opérationnelle en avril 2016 pour une durée de vingt-et-une semaines.

Ce stage m'a permis d'approfondir les cours de programmation enseignés à l'Ecole Centrale de Nantes. J'ai pu y utiliser plusieurs langages comme Python, C++ et javascript. Je me suis également familiarisée avec la méthode agile utilisée dans de nombreuses entreprises d'informatique.

Ce rapport s'attachera à présenter le contexte de mon stage ingénieur, chez Kisio Digital. Il décrira la mise en place de deux nouvelles API calculant des isochrones dans le logiciel Navitia. Il expliquera le cheminement qui a été suivi afin de créer ces API et de les mettre en production.

Partie I

Contexte du stage

I.1 Présentation de Kisio Digital

I.1.1 Présentation général

Pour mieux accompagner les autorités organisatrices de transport public et de mobilité, le Groupe Keolis (leader international du transport public de voyageurs) a regroupé ses expertises de Solutions et Services sous une marque unique, Kisio : Analysis, Consulting, Services, Solutions, et Digital. Kisio (650 collaborateurs et 60 millions de chiffre d'affaire) propose un ensemble de solutions et services aux acteurs de la mobilité.

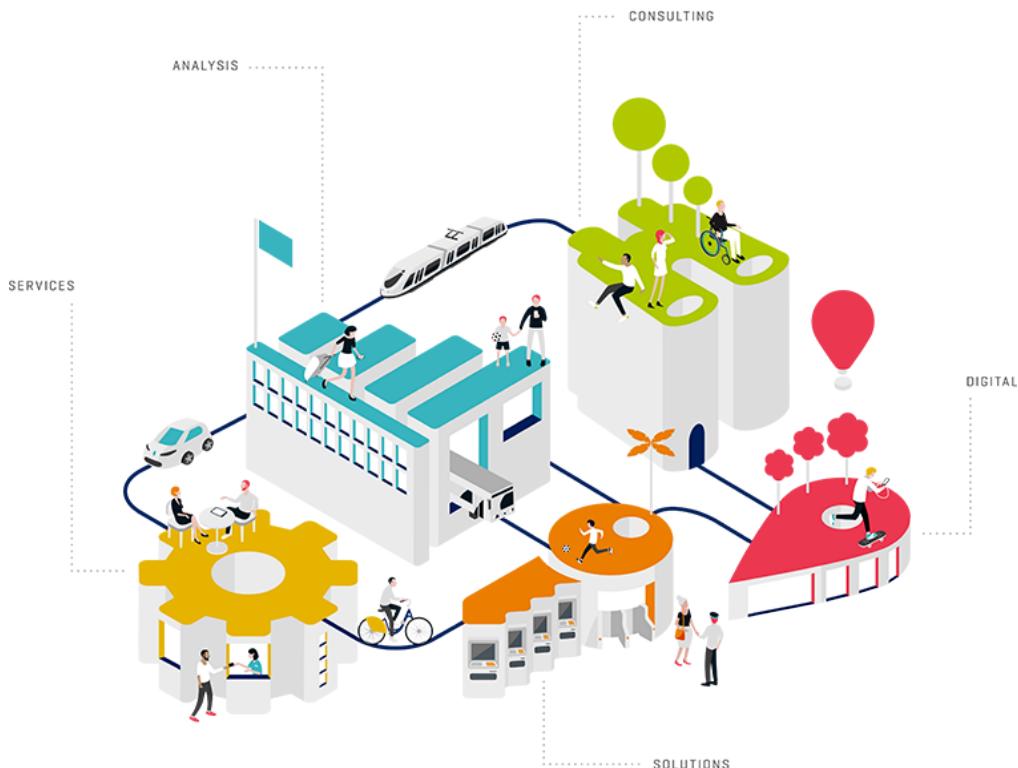


FIGURE I.1: Organisation au sein de Kisio

Leader de l'information voyageurs en France, Kisio Digital est une entreprise de services numériques. Elle édite des solutions de mobilité intelligentes au service des voyageurs et des acteurs du transport depuis 15 ans. Plus concrètement, elle réalise des sites web, des applis mobiles et des systèmes d'information voyageurs intégrant la recherche d'itinéraire multimodal et le temps réel, l'achat de titres de transport et bientôt le mobile-ticketing. Son domaine d'activité s'étend des réseaux de transports urbains et régionaux, aux réseaux ferroviaires nationaux (SNCF), en passant par les systèmes d'information-voyageur des Régions.

Kisio Digital encourage aussi l'innovation ouverte auprès des communautés de réutilisateurs de ses logiciels open source tels que Navitia et de son API navitia.io qui calcule 7 milliards d'itinéraires tous les ans.

Par ailleurs Kisio Digital se définit comme le pionnier de la *Responsive Locomotion* dont l'entreprise propose 4 définitions différentes et complémentaires[1] :

1. La responsive locomotion consiste à repenser les réseaux de transports à partir du contexte, des besoins et du point de vue de chaque voyageur. Pas l'inverse.
2. La responsive locomotion est un ensemble d'outils qui visent à insuffler de l'agilité et de la souplesse dans les déplacements de chacun en autorisant les détours impromptus, en dessinant des parcours personnalisés, en organisant des déviations de dernière minute, etc.
3. La responsive locomotion est un projet qui englobe tous les moyens de transport : la responsive locomotion part du principe que le parcours et les besoins de chaque voyageur importent plus que le type de véhicule emprunté.
4. La responsive locomotion est une révolution d'interfaces et de services, dont l'ambition est de refonder la relation des hommes aux véhicules qui les meuvent.

I.1.2 La méthode agile

Depuis 2014, Kisio Digital a opéré des changements au sein de son entreprise pour adopter un mode de développement plus agile :

"La méthode Agile est une méthode de gestion de projets Web introduite "officiellement" en 2001 avec la parution du "Manifesto for Agile Software Development" co-écrit par 17 grands acteurs du domaine de l'informatique et du développement de logiciel. Leurs volontés étaient de proposer un nouveau mode de conception des programmes informatiques.

Cette méthode fonctionne sur la base de l'itératif et de l'incrémental, les tâches vont s'effectuer petit à petit, par ordre de priorité, avec des phases de contrôle et d'échange avec le client." [2]

La méthode agile permet un dialogue plus riche entre le client et l'entreprise. A tout moment le client peut donner son opinion car il a régulièrement accès aux différentes versions fonctionnelles du logiciel. Les logicielles produits sont adaptés en fonction des retours clients et répondent donc précisément aux besoins exprimés. Elle repose en outre sur l'auto-organisation et donne plus de liberté aux développeurs et de visibilité sur le projet aux managers.

Organisation au sein d'une équipe Le temps de travail chez Kisio Digital est donc rythmé par des *sprints* d'une durée de 15 jours. Au début de chaque sprint les équipes se réunissent pour faire un *sprint planning* où elles estimeront la durée et la difficulté des tâches qu'elles ont à réaliser. Ses tâches sont priorisées dans un *backlog* en fonction des *parties prenantes* où chaque équipe est représentée et remonte des besoins au moyen d'*user stories*. Au sein de chaque équipe on trouve un *Product Owner* qui est garant de la vision fonctionnelle de l'équipe. Lors du sprint l'équipe se réunit quotidiennement pour faire des *stand up*, c'est à dire des réunions où tous les membres de l'équipe se coordonnent et donnent des informations sur leur avancement. Les membres de l'équipe font également des *grooming* où ils se réunissent pour réévaluer le temps nécessaire à chaque user story, affiner les besoins qui y sont liés et les découper en sous-tâches. A l'issue de chaque sprint il faut présenter les travaux effectués dans une *démo* où les clients sont conviés. Après chaque démo l'équipe doit faire une *rétro* pour discuter des points positifs et négatifs du sprint. C'est lors de ces rétros que les équipes décident d'*actions* à mettre en place afin de rendre le sprint suivant encore plus performant.

Feature team Lorsque je suis arrivée chez Kisio Digital les équipes étaient organisées par produit dont elles étaient responsables et les membres de l'équipe avaient souvent des compétences similaires. J'étais par exemple dans l'équipe de Recherche Opérationnelle (RO) qui s'occupait du logiciel Navitia, tous les membres de cette équipe avaient des compétences en Python et en C++. En Juillet, Kisio Digital a changé son organisation pour faire des *feature teams*. Les équipes ont été organisées autour de besoin, ce qui permettait, pour chaque user story, de modifier tous les produits impactés sans changer d'équipe. En outre ce type d'organisation permet une plus grande variété de profil au sein d'une même équipe. Cependant, pour garder la cohérence de chaque produit des *guildes* se sont formées. Ainsi à partir de juillet 2016 les membres de l'ancienne équipe Navitia ont été répartis entre l'équipe itinéraire et l'équipe horaire mais ont aussi fondé la guilde Navitia pour pouvoir garder une interface cohérente au sein du logiciel.

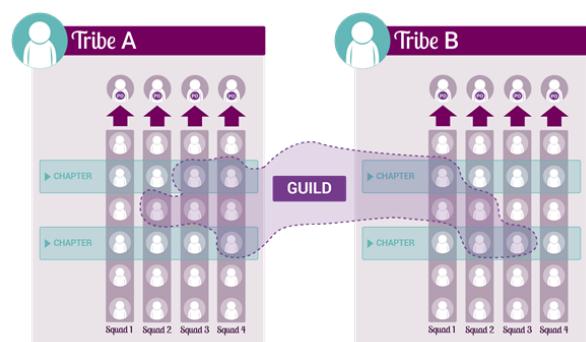


FIGURE I.2: Organisation des feature team

I.1.3 L'information voyageur

Face à l'augmentation du nombre et de la complexité des déplacements, les usagers souhaitent disposer d'une information fiable sur l'ensemble des modes de transport qui sont mis à leur disposition. Cette information multimodale est difficile à mettre en œuvre en raison du nombre important d'acteurs intervenant dans l'organisation des transports et de l'information et donc de la multiplicité des sources d'informations. En outre, la mise à jour des informations et la prise en compte des perturbations constituent un enjeu important d'un point de vue opérationnel, car les usagers ont besoin d'informations fiables avant et pendant le voyage.

Kisio Digital propose à ses clients un panel de solutions leur permettant d'installer un service d'information voyageur performant et adapté à tous les voyageurs. Dans cette partie nous définiront brièvement les fonctions d'un tel système.

Un système d'information voyageur répond à trois besoins importants :

Simplifier la vie du voyageur : Le voyageur peut préparer son itinéraire en toute sérénité. Il dispose de plusieurs informations lui permettant d'estimer la durée de son voyage mais également son coût, son impact sur l'environnement et les différents modes de transport qu'il va pouvoir emprunter.

Diminuer le stress : L'opérateur peut communiquer avec le voyageur pour le rassurer, le prévenir en cas de perturbation et lui proposer des itinéraires alternatifs.

Connaître les besoins de déplacement : Les autorités organisatrices peuvent mieux adapter leur offre et répondre de façon optimale à la demande grâce aux informations remontées par les SIV.

Historiquement les acteur de la mobilité étaient les transporteurs publics et les véhicules personnels. L'information voyageur n'était disponible que pour les transports publics et se présentait sous forme de fiches papier. Elle ne prenait pas en compte le temps réel et ne proposait aucune personnalisation. Avec la multimodalité due aux nouvelles mobilités et à l'apparition du numérique la demande d'information voyageur a changé. Les SIV doivent maintenant être interactifs et proposer plusieurs services :

Calculer des itinéraires : Il faut pouvoir proposer un "bon" itinéraire, mais cette notion est compliquée à définir. Il y a un compromis à faire entre l'itinéraire arrivant au plus tôt à destination et l'itinéraire avec le moins de correspondances.

Accompagner le voyageur au cours du trajet : Il faut pouvoir proposer en temps réel des alternatives en cas de perturbation sur l'itinéraire initialement prévu.

Informier sur l'accessibilité d'un lieu : Le voyageur a besoin de connaître l'offre de transport à proximité de son point de départ.

Les solutions proposées par Kisio Digital sont des solutions informatiques alimentées par le logiciel Navitia que nous décrirons plus en détail dans la partie suivante.

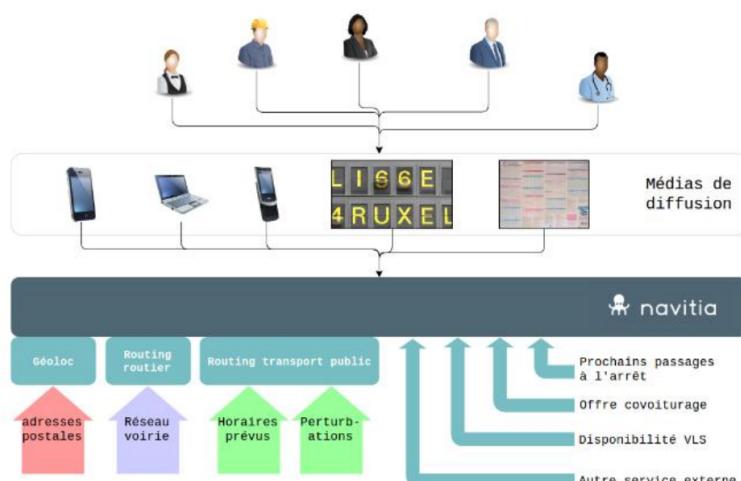


FIGURE I.3: Architecture du SIV proposé par Kisio Digital

I.2 Navitia une API open source

La technologie Navitia est un agrégateur de données, qui facilite la création et l'exploitation de services d'information voyageur. Au-delà des fonctions de recherches d'itinéraires, cette solution offre toutes les fonctions permettant de prendre en compte les horaires adaptés (intégrant les différentes sources de perturbations), et l'ensemble des informations liées à l'accessibilité des transports, aux cheminements piétons ainsi que les tarifs. Navitia est d'ailleurs largement utilisé en France et couvre notamment les transports en commun de l'île de France.



I.2.1 Les API

Navitia est un logiciel qui renvoie des flux fait pour être exploités par d'autres logiciels. On appelle cela une *API* (Application Programming Interface). L'API utilisant le logiciel Navitia s'appelle navitia.io. Ce sont ensuite les utilisateurs de navitia.io qui mettent en forme les flux retournés par l'API pour les présenter aux utilisateurs de leur application ou site web.

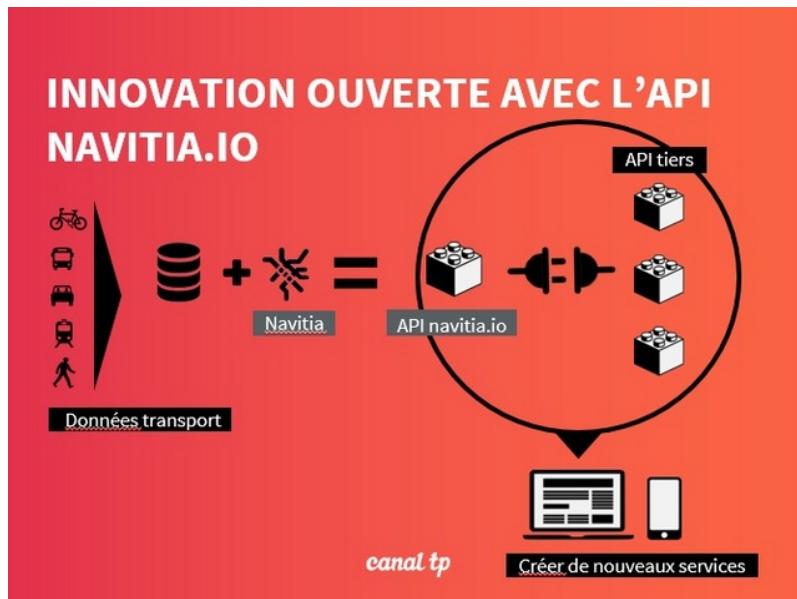


FIGURE I.4: Innovation ouverte avec l'AI navitia.io

En août 2016 navitia.io comptait 1400 réutilisateurs. Parmis eux : destineo.fr, le site de calculateur d'itinéraires pour la région Pays de la Loire et Flat Turtle : une petite startup belge qui installe des écrans d'informations dans les halls d'entreprises.

I.2.2 L'open source, l'open data et l'open service

En 2012 Kisio Digital change de philosophie et se tourne vers l'open source. Un logiciel open source est un programme informatique dont le code source est distribué sous une licence permettant à quiconque de lire, modifier ou redistribuer ce logiciel. Le passage en open source a permis au logiciel de favoriser l'innovation du produit grâce à l'implication d'une communauté large de développeurs et d'assurer une pérennité de la solution pour tous ses clients.

En restant dans cet esprit, Kisio Digital est maintenant un fournisseur d'open data. L'open data consiste à rendre accessible à tous des données, ici géographiques, rassemblées par une organisation. Cependant Kisio Digital ne va pas sur le terrain pour collecter les données qu'il met en open data. Grâce à son logiciel Fusio, l'entreprise agrège différentes sources de données dans différents formats pour produire un fichier au format gtfs et un autre au format ntfs. Ces deux fichiers ainsi que toutes les données open data dont fusio s'est servi sont disponibles gratuitement et librement sur le site navitia.io.

Enfin navitia.io est un open service. le site met à disposition de tous les services fournis par le logiciel Navitia. L'utilisateur peut ensuite utiliser Navitia qui est alimenté par les données open data fournies par Kisio Digital.

I.2.3 Le logiciel Navitia

L'architecture du logiciel Navitia a beaucoup évolué ces dix dernières années.

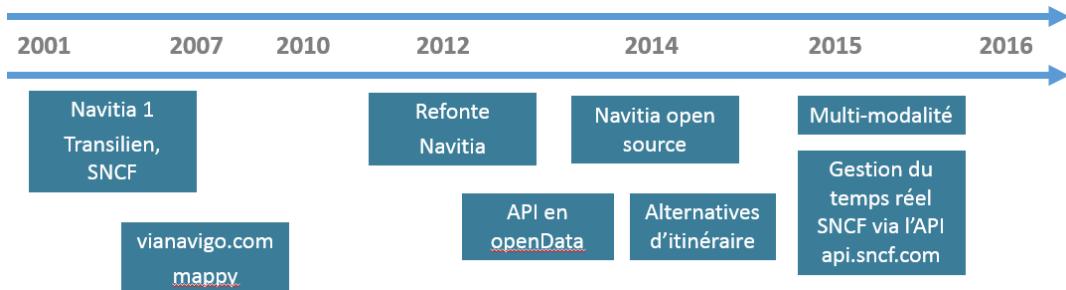


FIGURE I.5: Historique de Canal tp

Maintenant Navitia est un SOA sous forme d'ESB intégrant des ETLs spécifiques traitant le MDM.

SOA : concept d'une architecture orientée service. Organiser de manière homogène les traitements métiers déportés dans des services spécifiques.

ESB : Concept centrale d'une architecture SOA. Se connecte aux briques métiers pour fournir les services spécifiques.

MDM : standardisation des données. Agrégation : traitement des données dupliquées / dédoublées / enrichissement

ETL : connecteur aux données métier. Transforme les données d'un format vers un autre.

Navitia repose sur un calculateur codé en C++ nommé Kraken et une interface : Jörmungandr codée en python. Avant tout calcul, tartare un autre logiciel crée une base de données à partir de fichier ntfs. Le format ntfs a été créé pour Navitia et est assez proche du format gtfs. Les données liées à la topologie des territoires et au filaire de voirie sont extraite d'open street map qui est une base de données géographiques collaborative.

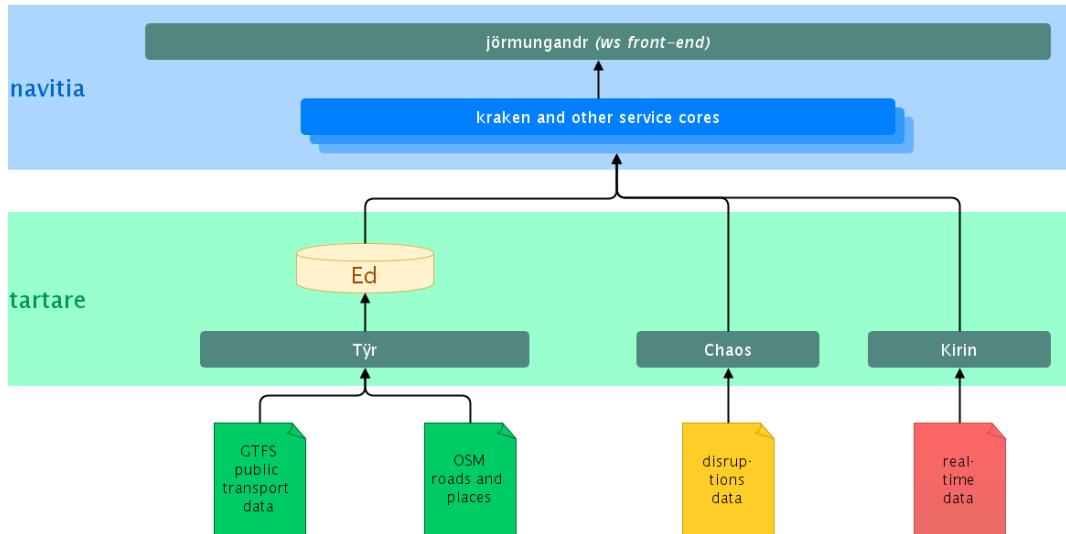


FIGURE I.6: Architecture de Navitia

Le calcul d'itinéraire qui est fait dans Kraken est basé sur l'algorithme RAPTOR. Kraken lit et interprète ensuite les itinéraires pour que Jörmungandr ne renvoie que les meilleurs itinéraires en fonction du profil du voyageur. Les informations sont échangées entre Jörmungandr et Kraken via des fichiers binarisés générés grâce à des fichiers en protocol buffer.

I.3 Présentation du stage

L'intitulé de mon stage était le suivant :

"Simplification dans l'utilisation du service Isochrone de l'API de web-service <http://api.navitia.io>. Le stage concerne la simplification de l'utilisation du service "isochrone" qui permet de remonter les temps d'accès depuis un point donné vers toute destination. Il consiste à transformer la sortie actuelle de Navitia (qui est un tableau de point avec pour chacun une heure d'arrivée) en éléments graphiques vectoriels permettant de dessiner une carte, plus simple à intégrer. Il nécessite de manipuler principalement du code c++, ainsi que du code Python, des librairies de cartographie et des bases de données."

Ce stage implique de coder dans Kraken afin de récupérer les isochrones déjà calculés par l'algorithme RAP-TOR pour les traiter et les organiser de manière à rendre ces informations plus visuelles. Il faut ensuite coder dans Jörmungandr afin d'implémenter la nouvelle API.

Partie II

Déroulement du stage

II.1 Définition des nouvelles API

II.1.1 Les isochrones

Les isochrones sont des outils permettant de mesurer l'accessibilité d'un lieu en terme de temps. Ils partent du principe que la proximité géographique de deux lieux n'implique pas nécessairement qu'il sera rapide d'aller de l'un à l'autre. Les isochrones permettent de raisonner en terme de temps d'accès plutôt qu'en terme de distance.

La définition communément admise en cartographie pour un isochrone est : ensemble des points atteignables en un temps donné en partant d'un point donné. Cette définition n'est pourtant pas satisfaisante si on l'applique aux transports en commun.

En effet contrairement au mode de transport tel que le vélo, la marche à pied ou la voiture, on ne peut pas se contenter de réfléchir en terme de durée lorsqu'on prend les transports en commun. Selon l'heure de départ ou d'arrivée souhaitée par l'usager l'offre de transport ne sera pas la même. Par conséquent le temps de trajet dépend de l'horaire où ce trajet est effectué. Dès lors il faut redéfinir notre notion d'isochrone : un isochrone est l'ensemble des points atteignables en un temps donné en partant d'un point donné à une date et un horaire donné.

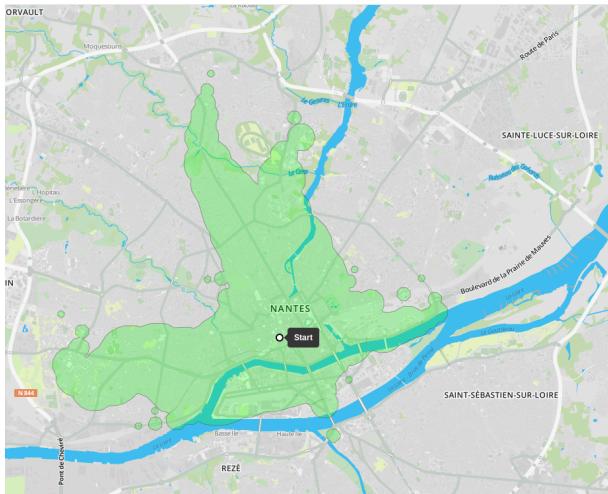


FIGURE II.1: Isochrone à 8h00

Isochrones de 10 minutes partant de Commerce

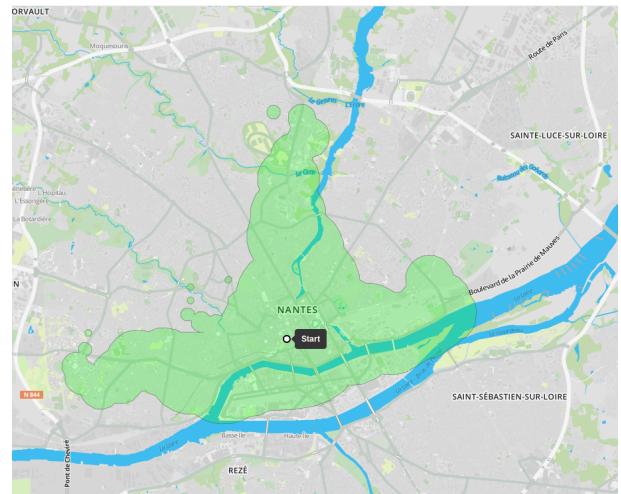


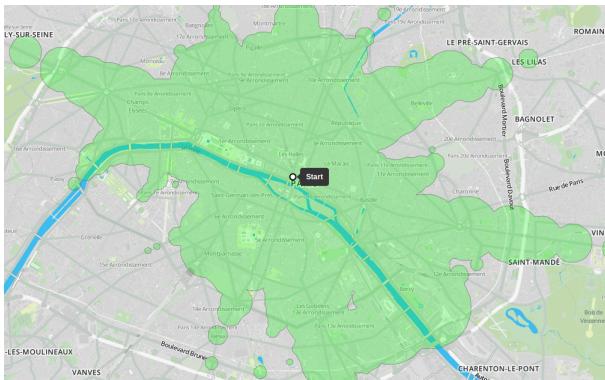
FIGURE II.2: Isochrone à 00h30

Selon les usages qui en seront fait la définition des isochrones peut être étendue. Dans ce stage nous avons défini deux usages et donc deux types d'isochrones bien distincts :

Les isochrones vectoriels Du début de mon stage à juin 2016 j'ai considéré qu'un isochrone permettait de repérer des services à proximité d'un point de départ. Ces isochrones sont des multipoligones utilisés pour calculer des intersections avec d'autres données vectorielles. A chaque isochrone correspond une et une seule

durée qui est la durée maximale souhaitée par l'utilisateur. On peut par exemple souhaiter savoir où se trouve les boulangeries à moins de 10 minutes de son lieu de travail. Pour cela on trace un isochrone de 10 minutes autour de ce lieux et on sélectionne les boulangerie à l'intérieur du multipolygone ainsi obtenu.

Les heat maps De juin à la fin de mon stage je me suis concentrée sur des isochrones permettant des tracés cartographiques plus précis. Cette fois il n'y a plus une seule durée. Le but est en un seul appel à l'API d'obtenir des données permettant de tracer une carte de chaleur selon le temps d'accès de chaque points. Contrairement à l'isochrone précédent la donnée en sortie de l'API n'est plus vectorielle.



Isochrones de 20 minutes partant de Châtelet à 8h00



II.1.2 Les problématiques d'interface

Comme nous l'avons vu dans la présentation de Navitia, l'architecture du logiciel est assez complexe. La première étape de mon stage consistait donc à spécifier les paramètres d'entrée et de sortie de l'API. Ces paramètres ont été affinés tout au long du stage et ceux présentés dans cette partie diffèrent légèrement des paramètres définis au début de mon stage.

Navitia est un unique logiciel qui propose différents services : calcul d'itinéraires, autocomplétion, affichage des horaires d'une ligne de bus, ... Pour que le logiciel reste facilement utilisable il est nécessaire que toutes les APIs soient cohérentes entre elles. Cela signifie : que les APIs doivent être nommées de la même manière, que le nom des paramètres ne doit pas changer d'une API à l'autre, que le cheminement pour accéder à une API doit toujours être le même. Il faut aussi que les nom propre à chaque API soit claire et sans équivoque.

Quelques règles à respecter Pour le nommage des paramètres d'une API dans Navitia il y a quelques règles à respecter : les paramètres qui peuvent être appellés plusieurs fois sont suivis de crochets *exemple : first_section_mode[]*. Les paramètres avancés et ne servant qu'aux développeurs (par exemple pour debuguer) sont précédés d'un underscore *exemple : _current_datetime*. Le nom des APIs et de leurs paramètres sont en snake case et au pluriel *exemple : heat_maps*.

Outres les paramètres indispensables à la construction d'un isochrone que nous avons vu dans la partie précédente, il a fallu adapter les paramètres utilisés dans le calcul d'itinéraire afin de maintenir la cohérence de l'interface. Voici un tableau des différents paramètres implémentés :

Paramètres communs aux API isochrones et heat_maps				
Requis	Nom	Type	Description	valeur par défaut
non	from	id	Identifiant du point de départ de l'isochrone.	
non	to	id	Identifiant du point d'arrivé de l'isochrone, ce paramètre est utilisé quand on calcule un isochrone inverse.	
oui	datetime	iso-date-time	Date et horaire d'arrivée ou de départ de l'isochrone.	
non	forbidden_uris[]	id	Identifiant permettant d'éviter une ligne, un mode, un réseau, etc.	
non	first_section_mode[]	tableau de string	<p>Force le premier mode de rabattement qui n'est pas du transport en commun. Ce paramètre peut prendre les valeurs suivantes : <i>walking, car, bike, bss</i>. <i>Bss</i> correspond aux vélos en libre service. C'est un tableau, il peut donc y avoir plusieurs modes sélectionnés.</p> <p><i>Note</i> : Choisir <i>bss</i> implique que la marche est également autorisée puisqu'il faut marcher pour aller jusqu'à la station des vélos en libre service.</p> <p><i>Note 2</i> : Le paramètre est inclusif et non exclusif, donc pour exclure un mode il faut ajouter tous les autres modes</p>	walking
non	last_section_mode[]	tableau de string	Comme <i>first_section_mode[]</i> mais pour le rabattement en fin d'itinéraire.	walking
non	max_transferts	int	Nombre de correspondances maximum autorisées.	
non	max_duration_to_pt	int	<p>Durée maximum de rabattement pour atteindre un transport public afin de limiter la marche à pied ou le vélo. Son unité est la seconde</p> <p><i>Note</i> : On peut régler indépendamment la durée maximale de rabattement pour chaque mode en utilisant les paramètres : <i>max_walking_duration_to_pt, max_bike_duration_to_pt, max_bss_duration_to_pt, max_car_duration_to_pt</i></p>	15*60s
non	walking_speed	float	Vitesse de marche à pied en m/s	1.12 m/s (4km/h)
non	bike_speed	float	Vitesse en vélo en m/s	4.1 m/s (14.7 km/h)
non	bss_speed	float	Vitesse en vélo en libre service en m/s	4.1 m/s (14.7 km/h)
non	car_speed	float	Vitesse en voiture en m/s	16.8 m/s (60 km/h)
non	data_freshness	enum	Permet d'utiliser les données temps réel ou les données théoriques. <i>data_freshness</i> peut prendre deux valeurs : <i>realtime</i> ou <i>base_schedule</i>	base_schedule
non	traveler_type	enum	C'est un profil prédéfini où les paramètres <i>first_section_mode[], last_section_mode[], walking_speed, bike_speed</i> et <i>bss_speed</i> sont modifiés en fonction de la valeur choisie. Les valeurs possibles sont : <i>standard, slow_walker, fast_walker</i> et <i>luggage</i>	standard

API isochrones				
Requis	Nom	Type	Description	valeur par défaut
non	max_duration	int	Durée maximale de l'isochrone en secondes	
non	min_duration	int	Durée minimale de l'isochrone en seconde	0 secondes
non	Boundary_duration[]	int	Pour faire plusieurs isochrones en une requête (limité à 10 boundary_duration[] par requête)	

Aucun de ces paramètres n'est obligatoire individuellement mais on ne peut pas faire un isochrone sans un paramètre *max_duration* ou *boundary_duration[]*. De plus tous les *boundary_duration[]* ayant une valeur supérieur au paramètre *max_duration* (s'il est renseigné) seront ignorés.

API heat_maps				
Requis	Nom	Type	Description	valeur par défaut
oui	max_duration	int	Durée maximale de l'isochrone en secondes	
non	resolution	int	Nombre de pixel sur une ligne ou une colonne de la grille qui sert à tracer l'isochrone	500

II.1.3 La réponse de l'API

API isochrones

Navitia renvoi un flux json quel que soit l'API requétée. Il fallait donc trouver le moyen de renvoyer des informations géographiques dans un tel flux. Le format geojson convenait parfaitement à ces critères.

GeoJSON (de l'anglais Geographic JSON, signifiant littéralement JSON géographique) est un format ouvert d'encodage d'ensemble de données géospatiales simples utilisant la norme JSON (JavaScript Object Notation). [3].

Pendant les trois premiers mois de mon stage j'ai mis en place une première API *isochrones* qui traçait des cercles tout autour des points d'arrêts atteignables en transport en commun. En faisant varier le rayon de ces cercles en fonction du temps d'accès à ce point d'arrêt on obtient une assez bonne estimation des différents lieux accessibles en un temps donné.

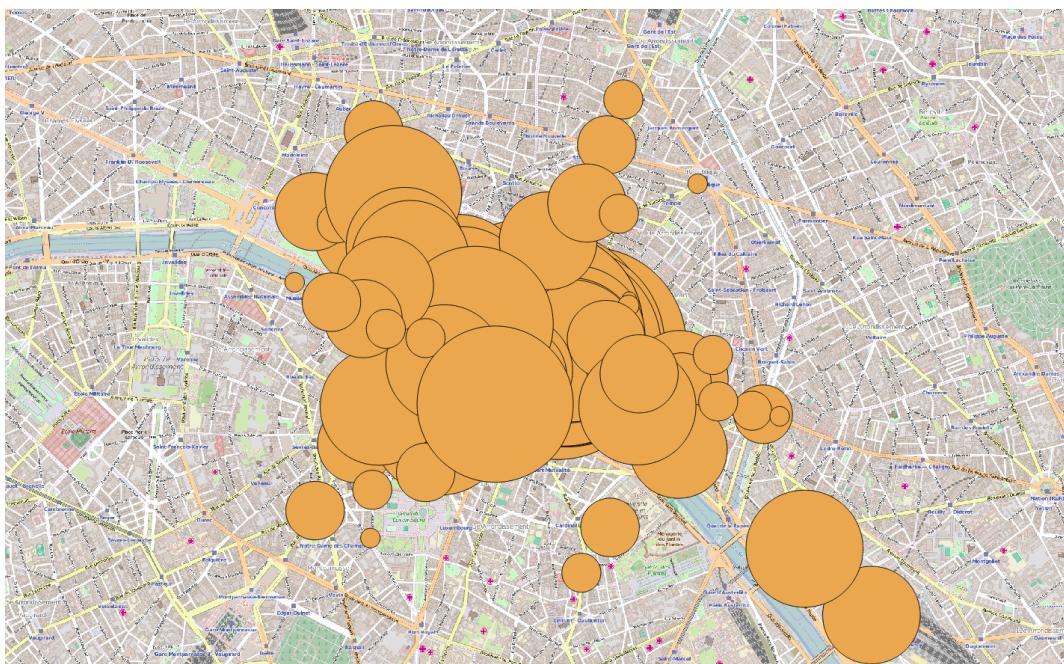


FIGURE II.5: Cercles autour des points d'arrêts

Pour avoir un rendu plus agréable et pour n'avoir qu'un seul objet à manier on fusionne ces cercles en un multi polygone.

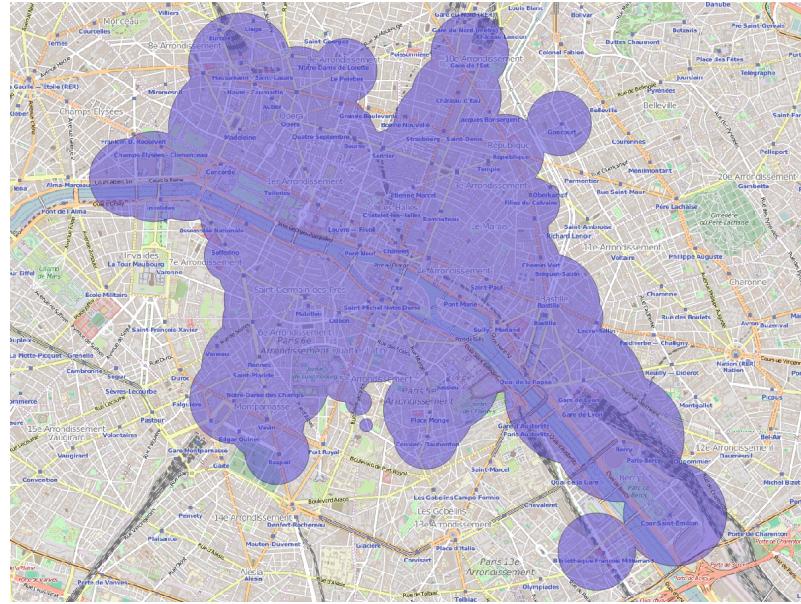


FIGURE II.6: Multipolygone obtenu après fusion des cercles

Généralement, il est intéressant de tracer plusieurs isochrones à différentes durées maximales. Mais pour afficher ensemble et correctement plusieurs isochrones il ne faut pas qu'il y ait de chevauchement entre eux puisque leur transparence s'ajouteraient et rendraient la carte difficilement lisible.

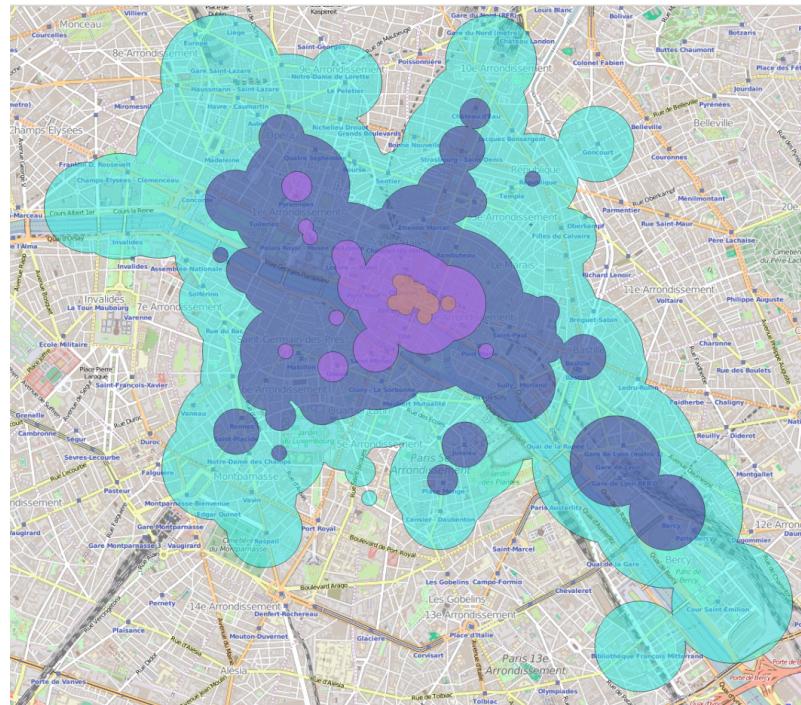


FIGURE II.7: Multiisochrone

Pour résoudre ce problème une borne min a été implémenté, elle permet de tracer un isochrone entre deux durées.

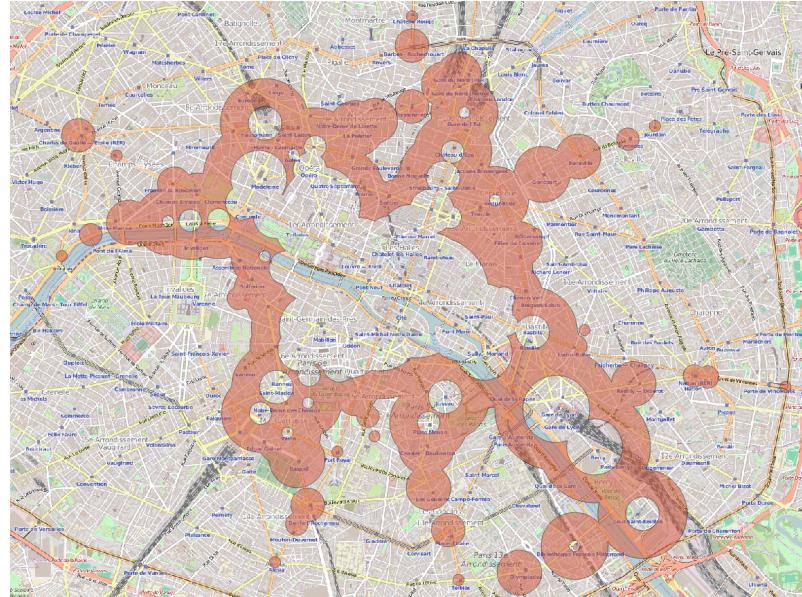


FIGURE II.8: Isochrone entre deux durées

La réponse de l'API *isochrones* est un flux json comportant entre autre un flux geojson qui décrit les multi-polygones renvoyés. Le flux prend la forme suivante :

```

1  {
2      isochrones :
3      [
4          0 :
5              {
6                  max_duration : 120,
7                  from : {...},
8                  geojson : {...},
9                  min_duration : 0,
10                 requested_date_time: "20160718T164045"
11             },
12             1 : {...},
13             2 : {...}
14         ],
15         feed_publishers :
16         [],
17         links :
18         [
19             0 :
20                 {
21                     href : "http://localhost:5000/v1/coverage/default/stop_areas
22                         /{stop_area.id}",
23                     type : "stop_area",
24                     rel : "stop_areas",
25                     templated : true
26                 }
27         ],
28         warnings :
29         [
30             0 :
31                 {
32                     message : "This service is under construction. You can help
33                         through github.com/CanalTP/navitia",
34                     id : "beta_api"
35                 }]
36     }

```

C'est dans le champs geojson qu'est stoqué l'information géographique. Si on le regarde plus en détail on a un champ de cette forme :

```

1
2   {
3     "geojson": {
4       "type": "MultiPolygon",
5       "coordinates": [
6         [
7           [
8             [
9               [
10                [
11                  [
12                    [
13                      [
14                        [
15                          [
16                            [
17                              [
18                                [
19                                  [
20                                    [
21                                      [
22                                        [
23                                          [
24                                            [
25                                              [
26                                                [
27

```

Cependant ces dessins d'isochrones ne prennent pas en compte une certain nombre de contraintes physiques inhérentes au territoire où ils sont tracés comme le filaire de voirie ou les cours d'eau. Ainsi un isochrone peut indiquer que l'autre côté de la berge se trouve à moins de 5 min du point de départ demandé alors que le pont pour y accéder se trouve à plus de 10 minutes.

API heat_maps

Nous venons de voir que l'API isochrones manquait parfois de précision, c'est pourquoi à la fin de mon stage j'ai fait des recherches sur une autre forme d'isochrones plus précis qui tiendrait compte du filaire de voirie. Pour celà j'ai dû discréteriser la zone géographique sur laquelle je calculais l'isochrone. J'ai ainsi obtenu l'API `heat_maps`.

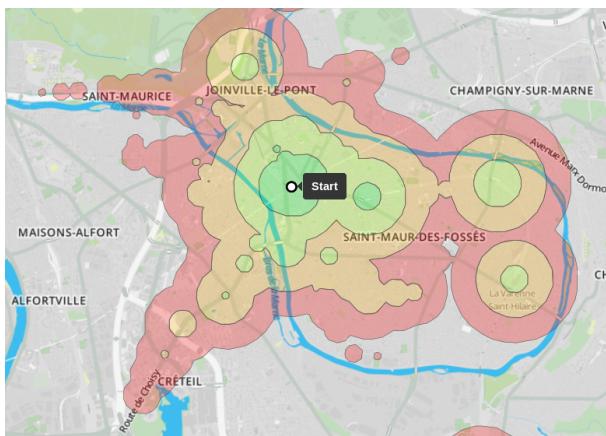


FIGURE II.9: Isochrone traversant une rivière

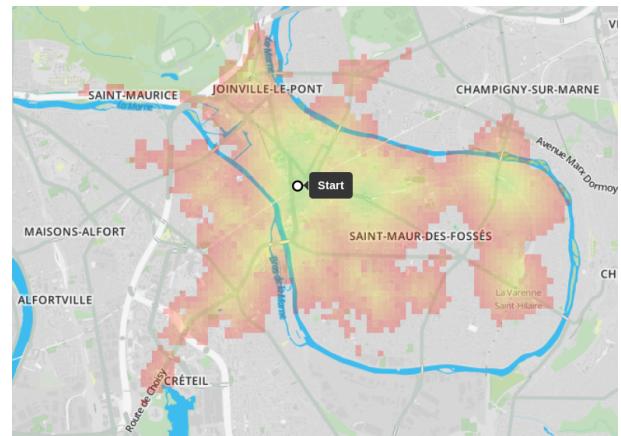


FIGURE II.10: Heat map respectant la voirie

Cette API est faite pour être plus précise que l'API isochrones, il est donc important de pouvoir définir le degré de discréétisation que l'on souhaite obtenir. Voici ici un exemple où on fait varier le degré de discréétisation.

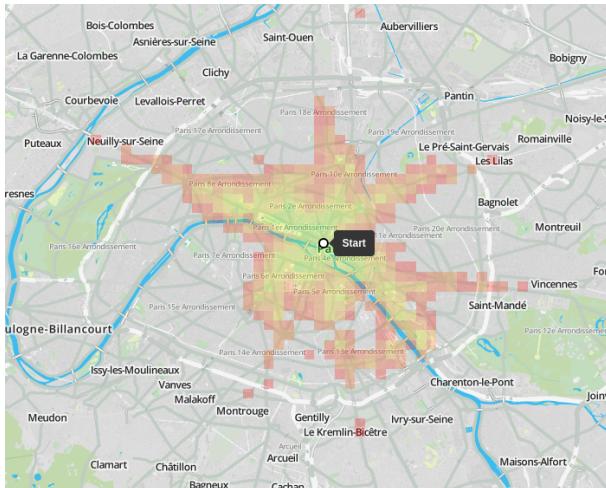


FIGURE II.11: Heat map résolution resolution = 50

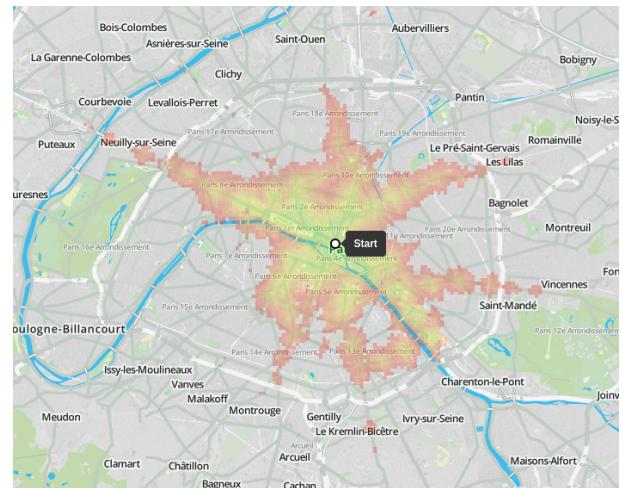


FIGURE II.12: Heat map resolution = 150

Cette fois l'API ne renvoie pas uniquement des informations géographiques. L'API doit renseigner les heures d'arrivées au plus tôt (ou les heure de départ au plus tard si on calcule un isochrone inverse) de tous les éléments discréétisés, elle doit également contenir les informations géographiques permettant de géolocaliser chacun de ces éléments. De plus, le flux json renvoyé doit être le plus petit possible, il ne faut donc pas avoir d'information redondante. Le flux renvoyé est le suivant :

```

1   {
2     heat_maps :
3     [
4       0 :
5         {
6           requested_date_time : "20160718T164045",
7           from : {...},
8           heat_matrix : {...}
9         }
10      ],
11      links :
12      [
13        0 :
14          {
15            href : "http://localhost:5000/v1/coverage/default/stop_areas
16              /{stop_area.id}",
17            type : "stop_area",
18            rel : "stop_areas",
19            templated : true
20          }
21        ],
22      warnings :
23      [
24        0 :
25          {
26            message : "This service is under construction. You can help
27              through github.com/CanalTP/navitia",
28            id : "beta_api"
29          }
30        ]
31   }

```

Les informations sont regroupées dans le champs heat_matrix qui est de la forme suivante :

```

1     heat_matrix :
2     {
3         line_headers :
4         [
5             0 :
6             {
7                 cell_lat :
8                 {
9                     min_lat : 48.7971,
10                    max_lat : 48.7996,
11                    center_lat : 48.7984
12                }
13            },
14            1 : {...},
15            2 : {...},
16            3 : {...},
17        ],
18        lines :
19        [
20            0 :
21            {
22                duration :
23                [
24                    0 : 1159,
25                    1 : null,
26                    2 : 1047,
27                    3 : 1035
28                ],
29                cell_lon :
30                {
31                    min_lon : 2.36887,
32                    center_lon : 2.37088,
33                    max_lon : 2.3729
34                }
35            },
36            1 : {...},
37            2 : {...},
38            3 : {...},
39        ]
40    }

```

Pour dessiner les heat maps il faut allouer une couleur à chaque durée puis parcourir les objets correspondants au champ *lines*. grâce au champ *cell_lon* on peut avoir la longitude de la cellule à colorer. Pour obtenir sa latitude il faut faire correspondre chaque objet du tableau dans le champ *duration* à l'objet de même indice dans *line_headers*. Dans l'exemple précédent par exemple la cellule dont la longitude est comprise entre 2.36887 et 2.3729 et dont la latitude est entre 48.7971 et 48.7984 correspond à un temps d'accès de 1159 secondes.

II.2 Algorithmie

Les deux APIs présentées dans ce rapport reposent sur des algorithmes très différents. Le but de cette partie est de présenté sommairement le raisonnement sur lequel elles reposent et d'expliquer les difficultés rencontrées. Le code détaillé est disponible sur <https://github.com/CanalTP/navitia>.

L'algorithme RAPTOR de Navitia possède une fonction isochrone. Celle-ci renvoie un tableau de tous les points d'arrêts avec leur temps d'accès atteignables à partir du point de départ dans l'horizon de temps donné. Je suis partie de ce tableau pour créer les deux APIs *isochrones* et *heat_maps*.

II.2.1 Pseudo code

isochrones : Cette API consiste à tracer des cercles autour des points d'arrêts atteignables.

```

Data: isochrones de RAPTOR, point de départ, durée de l'isochrone
Result: multipolygone sous forme de geojson
initialiser l'isochrone avec un cercle autour du point de départ en fonction de la durée de l'isochrone;
for tous les points d'arrêts dans l'isochrone de raptor do
    if temps d'accès < durée maximale then
        calculer temps restant;
        tracer un cercle dont le rayon dépend du temps restant;
        fusionner le cercle avec le multipolygone;
    end
end
```

Algorithm 1: Algorithme de l'API isochrones

heat_maps : Cette API est basée sur l'algorithme de Dijkstra. Cette algorithme permet de résoudre les problèmes de plus court chemin dans un graphe. En appliquant l'algorithme de Dijkstra au filaire de voirie on peut donc déterminer le temps d'accès minimum pour accéder à chaque noeud de la voirie en partant d'un point donné. Pour l'API heat_maps nous avons besoin d'utiliser un algorithme de Dijkstra un peu particulier car nous ne donnons pas un seul point de départ. En effet l'algorithme de Dijkstra parcourt le graphe toujours à la même "vitesse". Donc pour prendre en compte les différents modes de transport, il faut donner plusieurs points de départ, avec des dates de départs différentes, à l'algorithme de Dijkstra. Par la suite nous appellerons cette algorithme Dijkstra_multi_start.

```

Data: isochrones de RAPTOR, point de départ, durée de l'isochrone, resolution de la grille, filaire de voirie
Result: tableau de durée et de coordonnées géographiques
for tous les points d'arrêts dans l'isochrone de raptor do
    initialiser Dijkstra_multi_start ;
end
lancer Dijkstra_multi_start ;
tracer la grille for chaque cellule de la grille do
    projeter le centre de la cellule sur le filaire de voirie ;
    récupérer le temps d'accès calculé par Dijkstra_multi_start pour le noeud du graphe le plus proche;
    remplir le tableau pour la cellule;
end
```

Algorithm 2: Algorithme de l'API heat_maps

II.2.2 La géométrie sphérique

Pour les deux APIs, j'ai eu besoin d'outils géométriques afin de tracer les isochrones. Navitia doit pouvoir être utiliser quelque soit l'endroit où on se trouve sur la Terre. Il est donc impossible de faire des approximations locales permettant de raisonner en géométrie euclidienne. J'ai donc dû utiliser de la géométrie sphérique.

La géométrie sphérique est la géométrie utilisée sur une surface sphérique. Elle diffère de la géométrie euclidienne pour laquelle la surface de base est un plan. En géométrie sphérique deux droites parallèles peuvent se couper sans être confondu et la somme des angles d'un triangle est supérieure à l'angle plat. Beaucoup de définitions géométriques doivent donc être redéfinies. Pour ce type de géométrie on n'utilise plus les coordonées cartésiennes mais les coordonnées sphériques. Dans le cas de la terre on appelle ces coordonnées latitude et longitude. Dans la suite de ce rapport la longitude sera notée λ et la latitude θ , les longueurs seront exprimées en mètres et les angles en radians.

Faire une grille sur une sphère

Pour créer l'API heat_maps j'ai eu besoin de diviser la zone que j'étudiais en une grille. Pour cela j'ai utilisé des parallèles et des méridiens de la sphère terrestre. ces lignes délimitent de petites zones qui en géométrie

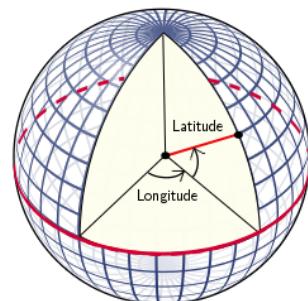


FIGURE II.13: Coordonnées à la surface de la Terre

euclidienne seraient équivalentes à des rectangles. Cette méthode est très simple à intégrer mais ne garantit pas la même précision selon l'endroit du globe où est calculé l'isochrone. Ainsi plus on se rapprochera d'un pôle, plus l'air d'une cellule sera faible et donc plus la précision augmentera.

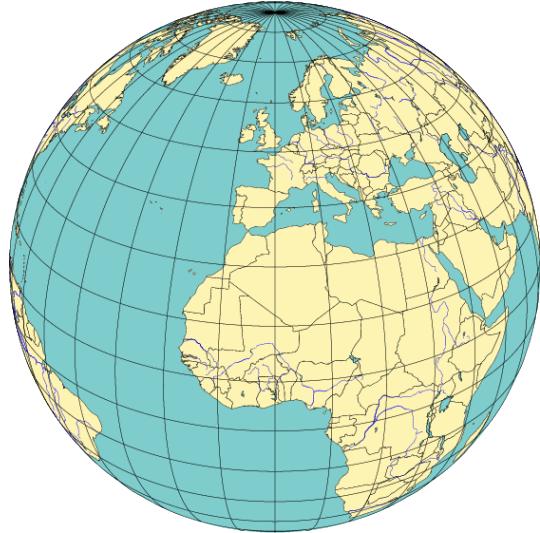


FIGURE II.14: Grille à la surface d'une sphère

Tracer un cercle à la surface d'une sphère

Comme nous l'avons vu précédemment l'algorithme de l'API isochrones consiste essentiellement à tracer des cercles autour de points précis. Si on revient à la définition d'un cercle on peut dire que c'est l'ensemble de points à une distance constante que nous appellerons *rayon* d'un point donné que nous appellerons *centre*.

Le geojson est un format qui permet de caractériser des polygones comportant un nombre fini de points, il est donc impossible de tracer un cercle parfait qui est un ensemble infini de points. Il faut discréteriser le cercle. Pour que les points soient uniformément répartis il faut échantillonner le cercle en fonction de l'angle au centre. Cela permet également de facilement choisir le nombre de points et donc la précision qui sera nécessaire à la cartographie sans trop alourdir le flux. La première chose à savoir faire est donc être capable de déterminer les coordonnées sphériques d'un point à une distance donnée d'un centre et avec un angle au centre donné. L'angle au centre est arbitrairement choisi comme étant l'angle entre le segment du méridien passant par le centre et le segment reliant le point projeté et le centre, sur le schéma ci-contre il est noté θ .

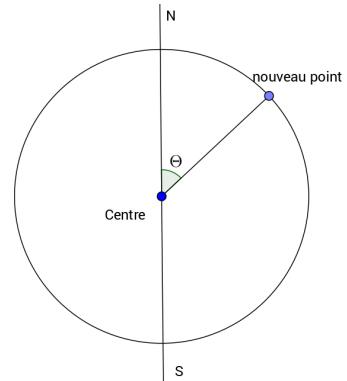
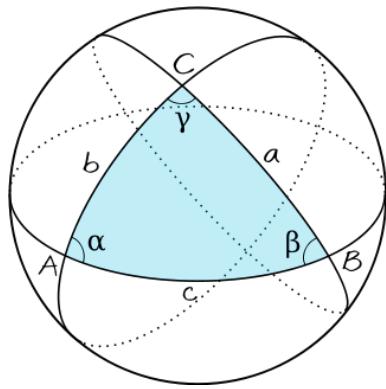


FIGURE II.15: point projeté

II.2.3 Résolution mathématiques

Pour implémenter la fonction que nous venons de décrire, il est nécessaire d'avoir quelques notions de géométrie sphérique. Nous avons donc utilisé la trigonométrie sphérique, qui permet d'établir des liens entre les distances et les angles, pour résoudre ce problème. Nous présentons ici les relations qui nous seront utiles dans un triangle rectangle sphérique.

En géométrie sphérique un triangle est une surface délimitée par des arcs de grands cercles. Un triangle rectangle sphérique est un rectangle dont l'un des angles mesure $\frac{\pi}{2}$ rad. Il est toutefois important de noter que la somme des angles d'un tel triangle est supérieure à π et que les relations de trigonométrie euclidienne ne sont pas valides ici. A la place on utilise les relations suivantes :



$$\sin(a) = \sin(c) \sin(\alpha) \quad (\text{II.1})$$

$$\cos(c) = \cos(b) \cos(a) \quad (\text{II.2})$$

FIGURE II.16: Triangle rectangle spérique

Grâce à ses propriétés nous pouvons maintenant procéder à la résolution mathématiques. Supposons que le point P de coordonnées (θ_0, λ_0) soit le centre du cercle que nous souhaitons tracer. On nommera respectivement λ et θ la longitude et la latitude du point projeté formant un angle ϕ avec le méridien passant par P. Le rayon du cercle est d et α correspond à l'ouverture d'angle entre P et le nouveau point. Enfin R est le rayon de la Terre.

On modélise notre problème grâce aux figures suivantes :

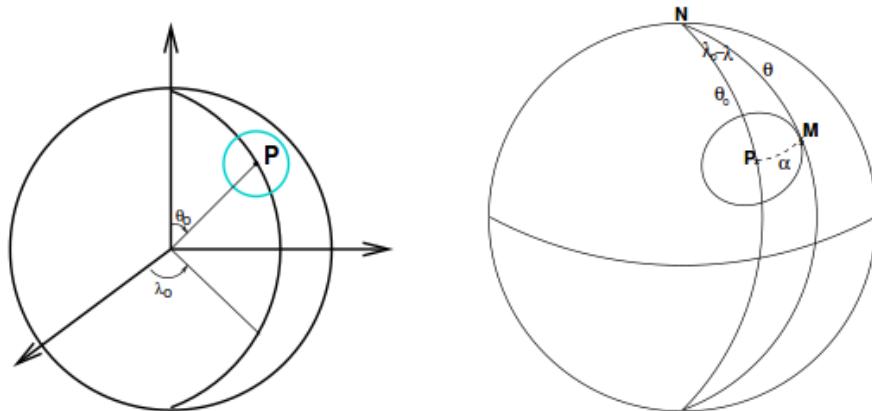
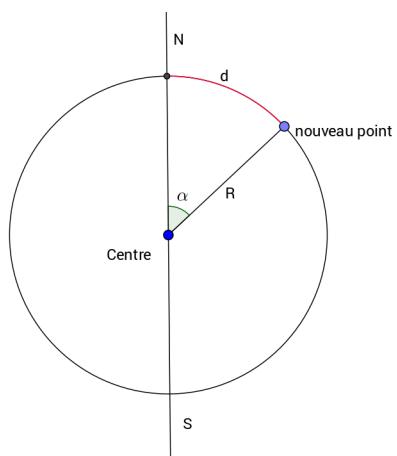


FIGURE II.17: Cercle à la surface d'une sphère

On peut donc facilement en déduire une relation entre d , α et R :



$$\alpha = \frac{d}{R} \quad (\text{II.3})$$

FIGURE II.18: Relation rayon ouverture d'angle

Par ailleurs on sait que l'équation d'un cercle sur une sphère est l'ensemble des points de coordonnées θ et λ tel que [4] :

$$\cos \alpha = \cos \theta \cos \theta_0 + \sin \theta \sin \theta_0 \cos(\lambda - \lambda_0) \quad (\text{II.4})$$

En mettant en commun toutes ces équations on obtient donc le système d'équation suivant :

$$p = \arcsin(\sin(\alpha) \sin(\phi)) \quad (\text{II.5})$$

$$\Delta\theta = \begin{cases} \arccos\left(\frac{\cos(\alpha)}{\cos(p)}\right) & \text{si } \phi \geq 0 \\ -\arccos\left(\frac{\cos(\alpha)}{\cos(p)}\right) & \text{si } \phi < 0 \end{cases} \quad (\text{II.6})$$

$$\theta = \theta_0 + \Delta\theta \quad (\text{II.7})$$

$$\Delta\lambda = \begin{cases} \arccos\left(\frac{\cos(\alpha) - \sin(\theta) \sin(\theta_0)}{\cos(\theta) \cos(\theta_0)}\right) & \text{si } \phi \in [-\frac{\pi}{2}, \frac{\pi}{2}] \\ -\arccos\left(\frac{\cos(\alpha) - \sin(\theta) \sin(\theta_0)}{\cos(\theta) \cos(\theta_0)}\right) & \text{si } \phi \in [\frac{\pi}{2}, \frac{3\pi}{2}] \end{cases} \quad (\text{II.8})$$

$$\lambda = \lambda_0 + \Delta\lambda \quad (\text{II.9})$$

Il suffit ensuite de faire varier ϕ entre $[0, 2\pi]$ avec un pas constant pour obtenir un ensemble de point qui correspondra au cercle discréétisé que nous souhaitons tracer.

II.3 Les performances

Pour que l'API soit efficace il était nécessaire que ses performances permettent de travailler avec des grands jeux de données. A titre d'exemple : l'île de France comporte plus de 40 000 points d'arrêts. Optimiser le temps de réponse de Navitia était donc un enjeu majeur de mon stage. Dans cette partie nous verrons les différents moyens utilisés pour optimiser le temps de réponse de l'API isochrones et heat_maps.

Le logiciel Navitia est paramétré de façon à ce que Jormungandr renvoie un message d'erreur si le temps de calcul dans Kraken dépasse 10 secondes. Il a donc fallu prendre en compte cette contrainte pour rendre les deux APIs opérationnelles. Dans la suite de cette partie, toutes les courbes représentées feront référence à l'API isochrones qui a bénéficié d'une plus longue étude de ses performances.

II.3.1 Méthodologie

Pour étudier les performances dans Kraken il fallait savoir combien de temps le programme passait dans chaque fonction. L'outil *google CPU profiler* permet d'identifier les morceaux de codes où le programme passe le plus de temps.

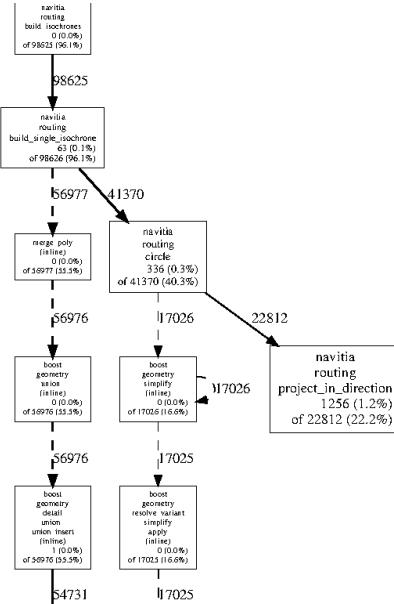


FIGURE II.19: Interface de google CPU profiler

Google CPU profiler permet d'estimer le nombre d'appels fait à chaque fonction ainsi que le pourcentage de temps passé dans chaque fonction. Pour avoir une estimation fiable de ces données on a utilisé un script shell permettant de faire une centaine d'appel de façon automatique aux APIs isochrones et heat_maps. On peut ainsi identifier les fonctions les plus coûteuse en temps. Cette méthode permet d'identifier les fonctions à optimiser en priorité.

En une requête on peut demander au plus 10 isochrones. Il faut donc que le temps de réponse pour un isochrone soit de l'ordre de 0.5 seconde et ceux quelque soit la durée et l'emplacement de l'isochrone. Nous avons estimé que si cette condition était satisfaite à 8h00 du matin à la station châtelet alors l'API pouvait passer en production. Avant toute optimisation on avait les temps de réponse suivant :

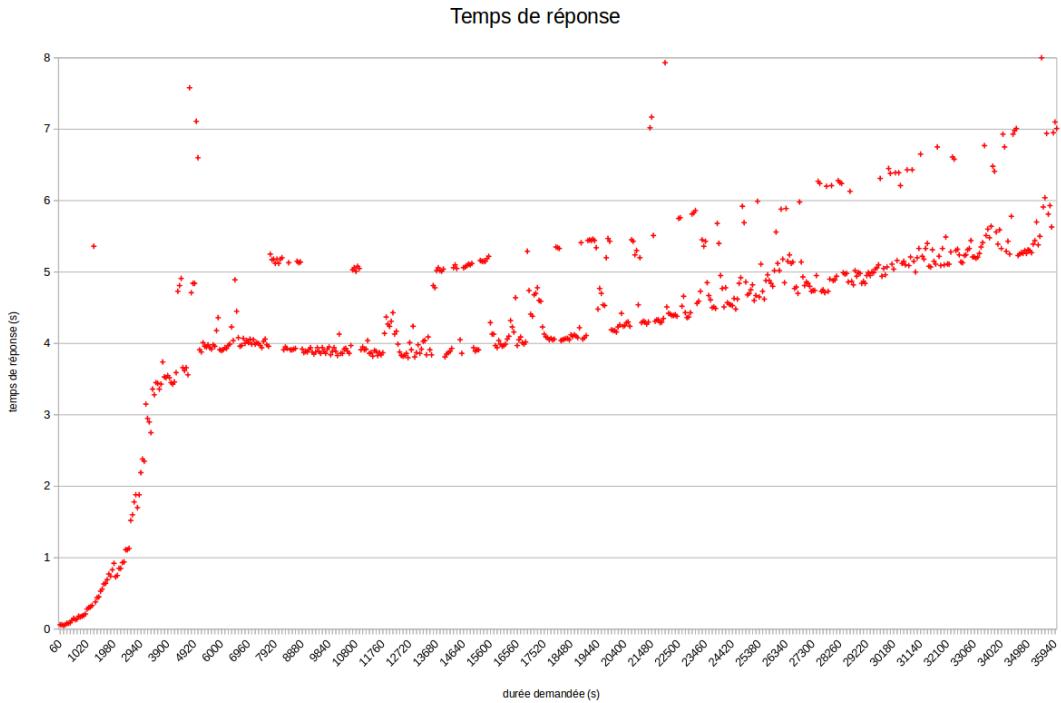


FIGURE II.20: Profile de temps de réponse avant optimisation pour l'API isochrones

II.3.2 Influence du tri

Comme nous l'avons dit précédemment le réseau de transport en commun de l'île de France comporte plus de 40 000 points d'arrêts. Pour de grandes durées le nombre de points d'arrêts contenus dans un isochrone est donc très important. Or dans l'algorithme utilisé le temps de réponse dépend du nombre de polygones dans le multipolygone qui représente l'isochrone. Il faut donc avoir le moins de polygones possible dans le multipolygone. C'est pourquoi, à chaque fois qu'on ajoute un cercle on essaye de le faire fusionner avec les polygones déjà existant.

Lorsque les points d'arrêts sont pris dans un ordre aléatoire on à la courbe suivante pour un isochrone partant de châtelet à 8h00 et ayant une durée d'une heure. Chaque itération correspond à la prise en compte d'un nouveau point d'arrêt.

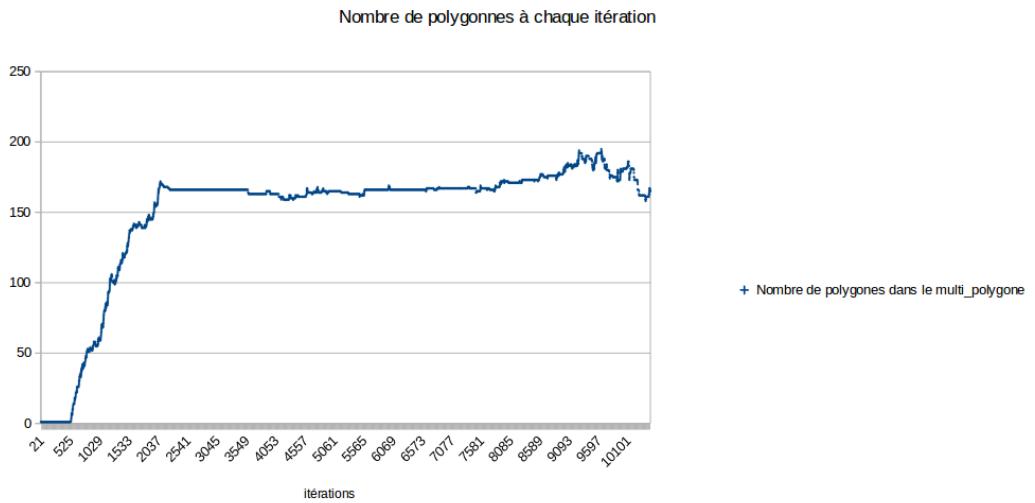


FIGURE II.21: Nombre de polygones pour un ordre aléatoire

On observe que le nombre de polygones augmente très rapidement. Maintenant si on pense à trier les cercles en fonction de la distance qui les sépare du centre de l'isochrone, en prenant d'abord en compte les cercles les plus proches du centre, on peut supposer que les polygones seront plus souvent adjacents et donc qu'on pourra aisément les faire fusionner entre eux, ce qui minimisera leur nombre.

Lorsqu'on trie les points d'arrêts en fonction de leur proximité avec le point de départ de l'isochrone on obtient la courbe suivante :

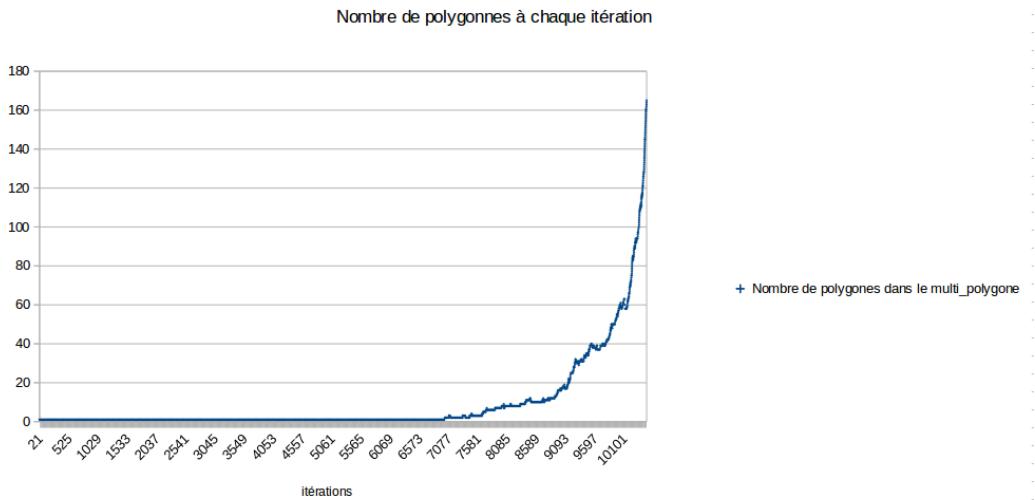


FIGURE II.22: Nombre de polygones lorsqu'on ordonne les points d'arrêts

Le nombre de polygones n'augmente qu'à partir des dernières itérations, ce qui minimise considérablement les calculs fait pendant les itérations précédentes et contribue à améliorer les temps de calcul dans Kraken.

II.3.3 Echantillonnage

Nous l'avons vu, le calcul des coordonnées d'un cercle sur une sphère est fastidieux et fait appel à des fonctions trigonométriques. Or ces fonctions sont coûteuses en temps, il faut donc les appeler un minimum de fois. Pour cela il est judicieux d'exploiter les nombreuses symétries du cercles. En effet on peut calculer toutes les coordonnées d'un quart de cercle puis utiliser ses symétries par rapport à l'axe des abscisses et des ordonnées d'un repère polaire afin de reconstruire le cercle en entier. Cette opération est d'autant plus facile que ces symétries sont

les mêmes qu'en géométrie euclidienne. Cette méthode permet de ne plus appeler les fonctions construisant un point d'un cercle qu'une fois là où elles étaient auparavant appelées quatre fois.

D'autre part, nous avons déjà vu que les cercles que nous tracions étaient en réalité des cercles échantillonés. Initialement ces polygones possédaient 360 points. En divisant ce nombre par deux, on réduit par deux les nombre d'appel au fonctions trigonométriques utilisées ce qui permet là encore d'améliorer le temps de réponse de l'API.

Après toutes les optimisations que nous venons de décrire nous pouvons tracer la courbe bleue qui représente les nouveaux temps de réponse :

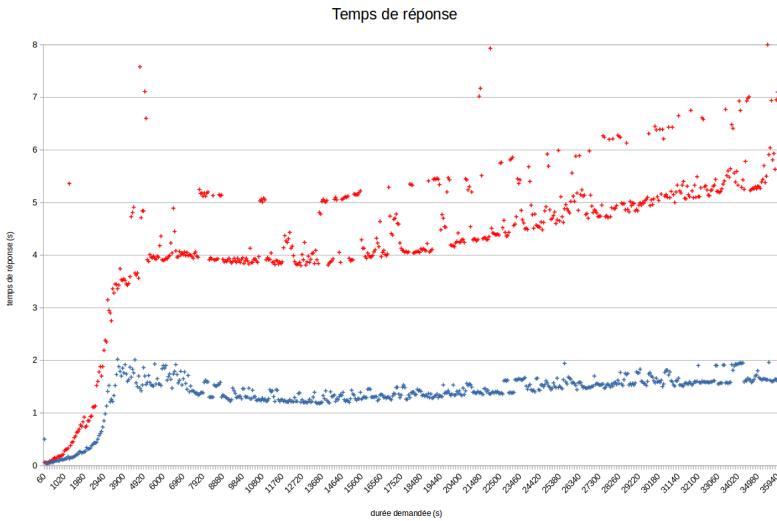


FIGURE II.23: En rouge : temps de réponse avant tri et rééchantillonage, en bleu : temps de réponse après

Enfin le rééchantillonage des polygones a également permis de réduire de moitié la taille des flux que l'API isochrones donnait en réponse.

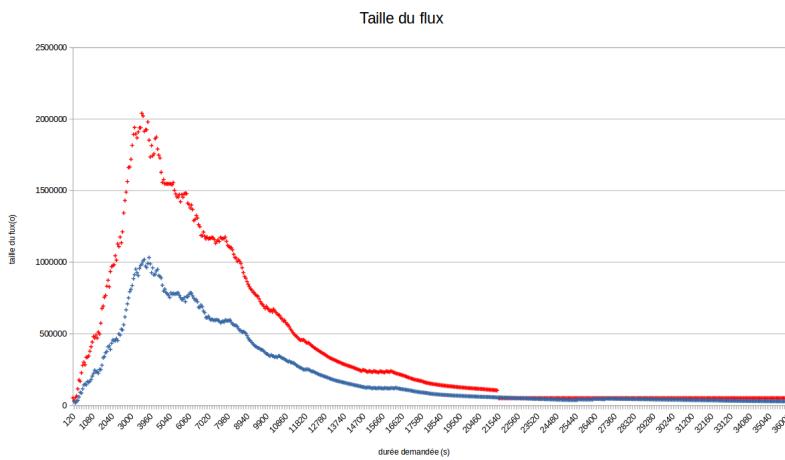


FIGURE II.24: En rouge : taille du flux avant tri et rééchantillonage, en bleu : taille du flux après

Dans l'API heat_maps la taille de la grille utilisée influe également sur le temps de calcul et la taille du flux de sortie. Pour que le temps de calcul n'excède pas 10 secondes le nombre de pixels sur une ligne de la grille qui divise la zone d'étude ne peut pas dépasser 1000.

II.3.4 Distance et projection

Les fonctions trigonométriques sont des fonctions qui demande un temps de calcul important, il est donc capitale de les utiliser le moins possible si on veut améliorer le temps de calcul. Cependant pour les APIs isochrones et heat_maps nous avons souvent besoin d'utiliser des fonctions permettant d'évaluer des distances entre deux points d'une sphère. Pour cela on utilise la formule d'Haversine :

$$d = 2r \arcsin \left(\sqrt{\sin^2 \left(\frac{\theta_2 - \theta_1}{2} \right) + \cos \theta_1 \cos \theta_2 \sin^2 \left(\frac{\lambda_2 - \lambda_1}{2} \right)} \right) \quad (\text{II.10})$$

Où d est la distance entre les deux points de coordonnées (θ_1, λ_1) et (θ_2, λ_2) .

Cette formule comporte ce nombreuses fonctions trigonométriques. Or dans la majorité de l'algorithme, ce n'est pas la distance en elle-même qui importe mais la possibilité de comparer deux distances entre elles. De plus sur des distances inférieures à une centaine de kilomètre le biais introduit par la courbure de la Terre peut être négligé. On utilise alors une autre formule pour calculer les distances.

$$d = \sqrt{r^2 (\theta_2 - \theta_1)^2 + (\lambda_2 - \lambda_1)^2} \quad (\text{II.11})$$

Cette nouvelle fonction permettant de calculer les distances est utilisée dans l'API isochrones afin d'ordonner les points d'arrêts et de calculer le rayon des cercles à tracer autour d'eux. Elle est aussi utilisée dans l'API heat_maps où elle permet de projeter les centre des pixels de la grille d'échantillonnage sur les arcs du graphe du filaire de voirie. Dans les deux cas cette formule abaisse significativement le temps de calcul dans Kraken.

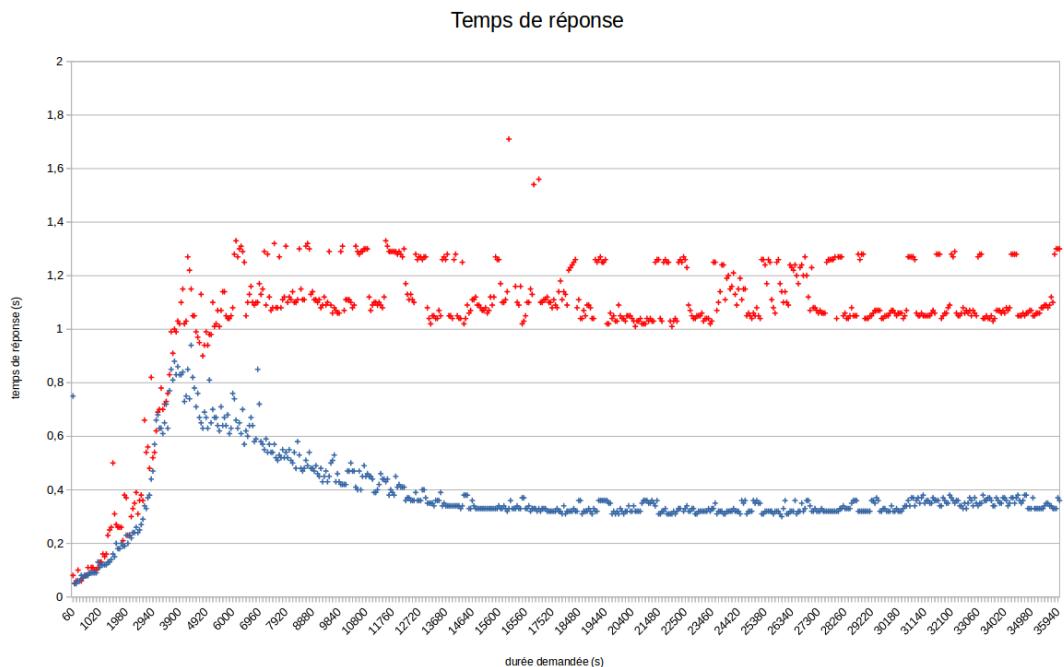


FIGURE II.25: En rouge : temps de réponse avant utilisation de la formule simplifiée, en bleu : temps de réponse après

Le temps de réponse moyen de l'API isochrones pour un isochrone partant de châtelet à 8h00 est de 0.39 secondes. Pour l'API heat_maps le temps moyen de calcul est de 3 secondes. On a donc estimer que les deux APIs pouvaient passer en production.

II.4 La visualisation

Même si mon stage consistait à retourner des flux json, il était essentiel de pouvoir visualiser ces flux afin de corriger les bugs. Dans un premier temps j'ai utilisé le logiciel qgis en redirigeant les flux json renvoyés par l'API

dans un fichier .txt. Je modifiais ensuite ce fichier à la main afin d'en extraire la partie affichable. Mais cette solution était peu satisfaisante, ne fonctionnait pas simplement avec l'API heat_maps et était trop coûteuse en temps. J'ai donc utilisé deux outils internes de Kisio Digital dans lesquels j'ai codé en javascript des appels aux API isochrones et heat_maps. Cette partie présentera le fonctionnement des deux interfaces homme machine (IHM) utilisées.

II.4.1 Navitia Explorer

Navitia Explorer est l'IHM qui a été utilisée jusqu'en juin 2016 par les développeurs de Kisio Digital. Développée par le pôle data elle permet de visualiser simplement les réponses de Navitia. Son code est disponible sur <https://github.com/CanalTP/navitia-explorer>. Cette IHM présente les réponses de Navitia sous forme de cartes, de liste et de tableau afin d'être facilement lisibles.

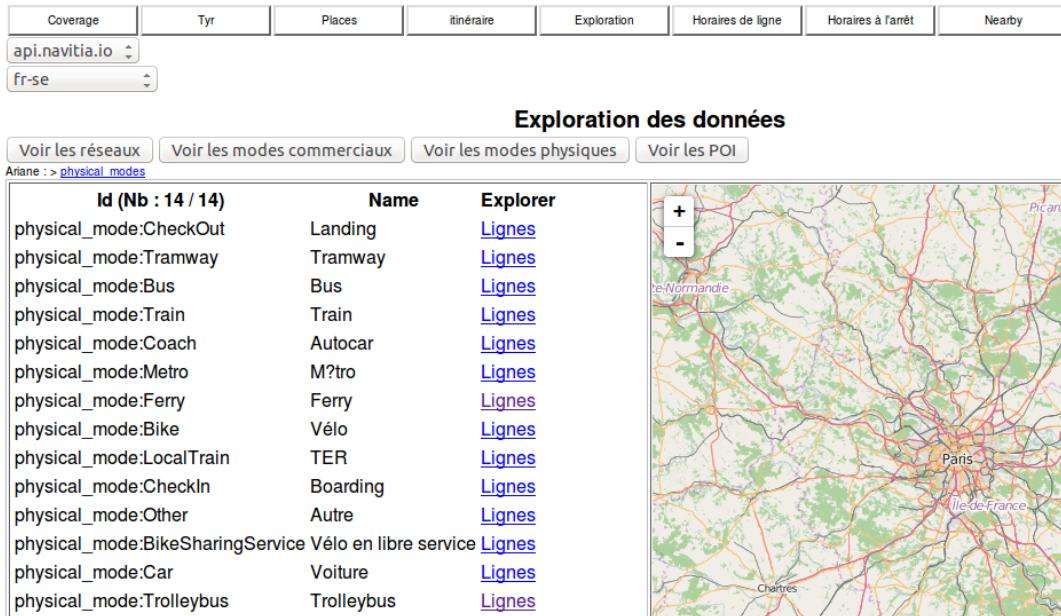


FIGURE II.26: Interface de Navitia explorer

Toutes les APIs de Navitia sont accessibles grâce aux onglets situés en haut de la page. Pour accéder à l'API isochrones j'ai donc créer un onglet supplémentaire.

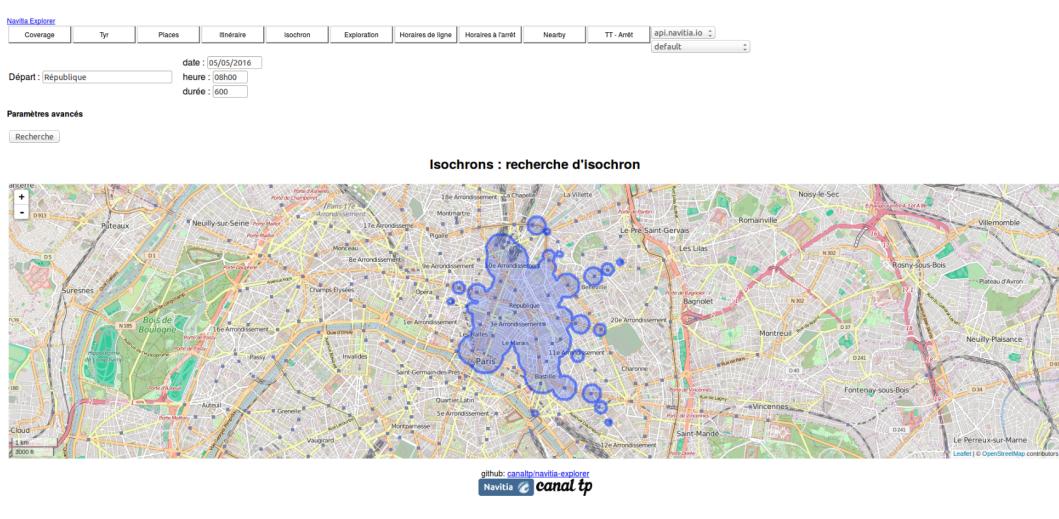


FIGURE II.27: Navitia explorer : onglet isochron

L'appel à l'API heat_maps n'a pas été codé dans navita explorer car cette API a été codée après juin 2016. En effet à cette période les développeurs ont commencé à utiliser une nouvelle IHM : Navitia playground.

II.4.2 Navitia Playground

En mai, Kisio Digital a monté une équipe afin d'implémenter une nouvelle IHM permettant de se familiariser avec le fonctionnement de Navitia. Cette équipe à coder en un mois *Navitia playground* dont le code est disponible sur <https://github.com/CanalTP/navitia-playground>. Contrairement à *Navitia explorer*, cette IHM n'est pas composée de plusieurs onglets.

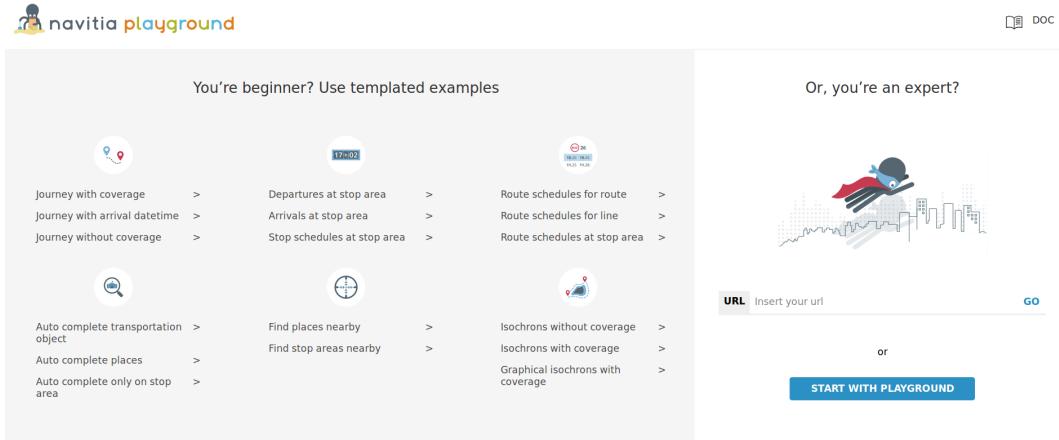


FIGURE II.28: page d'accueil de Navitia playground

Navitia playground permet de voir l'URL qui est utilisée pour interroger Navitia, de plus on a facilement accès au flux json renvoyé par l'API. Cela permet de comprendre plus facilement le fonctionnement du logiciel. En outre le fait d'avoir à la fois le flux json et les informations affichées sous forme de carte permet d'être plus efficace quant à la recherche de bug.

Quelque soit l'API requétée *Navitia playground* n'a qu'une seule interface qui génère une URL en fonction des paramètres rentrés via l'interface graphique.

This screenshot shows the 'Create a request' interface. It includes fields for 'API' (set to https://api.navitia.io/v1), 'Token' (set to 3b036afe-0110-4202-b9ed-9971847...), and 'Build your path' (with 'coverage' set to 'sandbox'). Under 'Add parameters', there are fields for 'from' (2.37715;48.846781), 'to' (poi:n3134285384), and 'traveler_type' (set to 'fast_walker', with other options like 'luggage', 'slow_walker', and 'standard' available in a dropdown). A URL field at the bottom shows the generated URL: https://api.navitia.io/v1/coverage/sandbox/journeys?from=2.37715%3B48.846781&to=poi%3An3134285384&traveler_type=fast_walker&datetime=20160804T133700&. A 'SUBMIT' button is at the bottom.

FIGURE II.29: Page de création de requête

Cette structure permet à *Navitia playground* d'avoir un code beaucoup plus factorisé que *Navitia explorer*, il est donc plus facile d'ajouter les réponses des APIs isochrones et heat_maps dans *Navitia playground* que dans navita explorer.

On a donc rapidement intégré les réponses des deux APIs dans *Navitia playground*. La majorité des illustrations de ce rapport viennent de cette IHM.

The screenshot shows the 'Explore the response' section of the Navitia playground. At the top, there is a 'SUBMIT' button. Below it, the status is shown as 'Status: OK (200), duration of the request: 64ms'. The response is described as 'response 3 isochrones'. There are three main sections for the isochrones:

- Isocrones[0]**: from GARE DE ST MAUR CRETEIL (Saint-Maur-des-Fossés), duration: [0s, 5min]. This section includes a 'MAP' button and a copy icon.
- Isocrones[1]**: from GARE DE ST MAUR CRETEIL (Saint-Maur-des-Fossés), duration: [5min, 10min]. This section also includes a 'MAP' button and a copy icon.
- Isocrones[2]**: from GARE DE ST MAUR CRETEIL (Saint-Maur-des-Fossés), duration: [10min, 15min]. This section includes a 'MAP' button and a copy icon.

Below these sections, there is a 'warnings[0]' message: 'This service is under construction. You can help through github.com/CanalTP/navitia'. A copy icon is next to this message. At the bottom, there is a 'beta.apl' message: 'Id: "beta.apl", "message": "This service is under construction. You can help through git ..."'.

FIGURE II.30: Navitia playground : requête d'isochrone

De plus Navitia playground permet d'isoler un objet quand plusieurs sont demandés en une seule requêtes. On peut ainsi choisir d'afficher les objets sur des cartes différentes où sur la même carte en fonction du besoin.

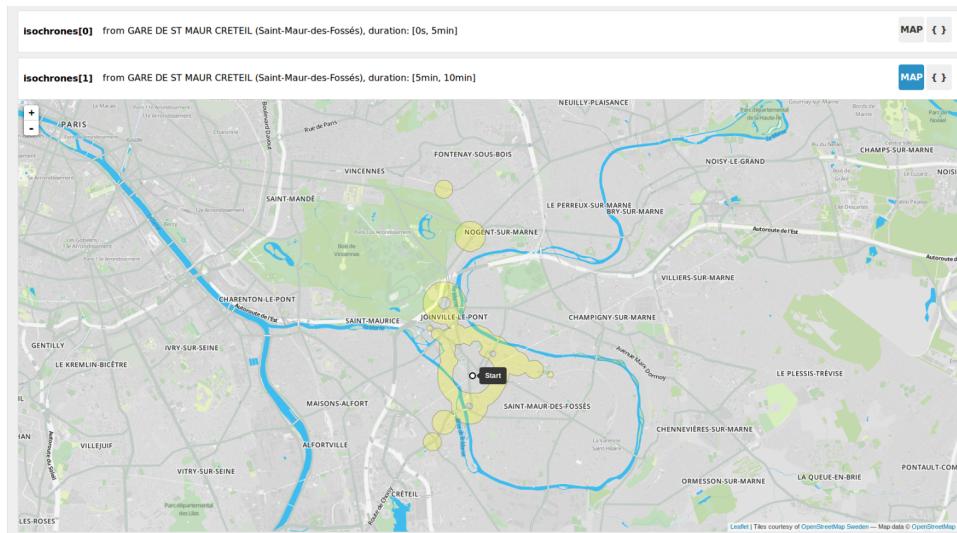


FIGURE II.31: Navitia playground : isochrone isolé

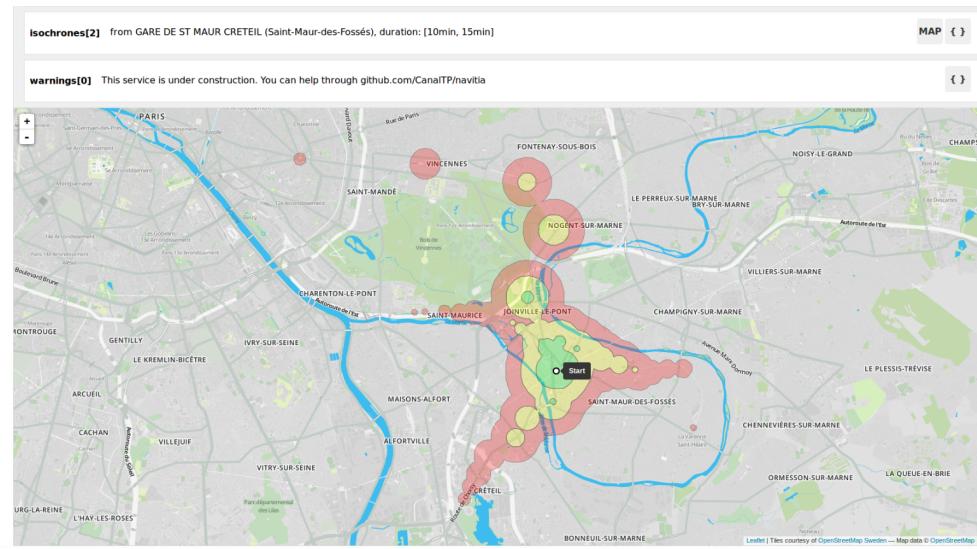


FIGURE II.32: Navitia playground : isochrones groupés

L'API heat_maps est aussi affichable dans Navitia playground, cependant si la résolution est trop importante, c'est à dire si la grille utilisée pour découper l'espace possède trop de pixels, alors Navitia playground ne parvient pas à afficher la heat_maps car il y a trop de géométries différentes à afficher.

Explore the response

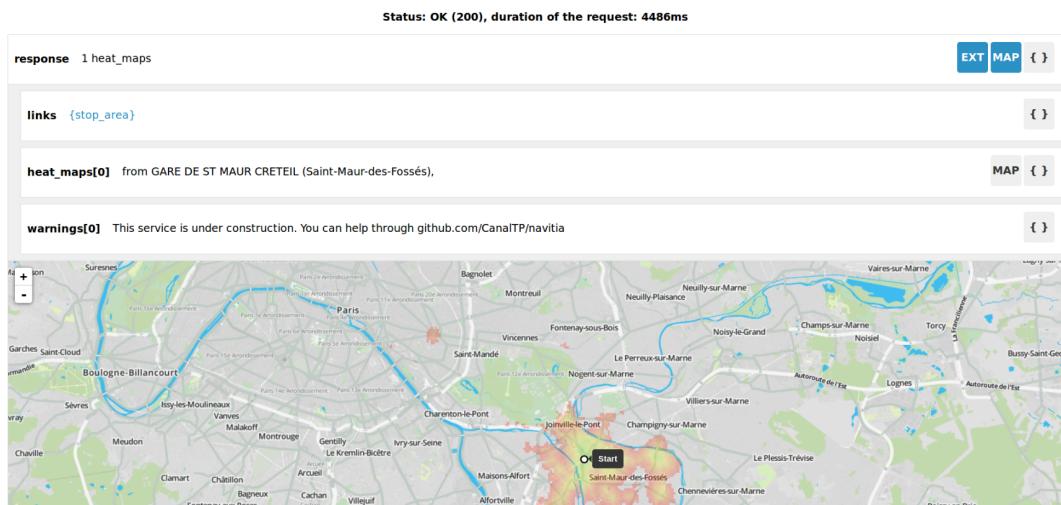


FIGURE II.33: Navitia playground : requête d'heat maps

Conclusion

Ce stage à permis d'implémenter deux APIs dans le logiciel Navitia. Ces deux APIs permettent de représenter des isochrones de façon différentes afin de s'adapter au différents usages qui peuvent en être fait. Ces APIs permettent d'avoir des outils cartographiques afin de mieux appréhender l'accessibilité au sein des villes en prenant en compte les transports en commun. Le tableau ci-dessous résume les différences entre ces deux APIs

Critère	Isochrone	Heat map
Mode de représentation	vectoriel	raster
Temps de réponse	< 1s	< 3 s
Taille du flux	< 1 Mo	4 Mo
Facilement utilisable pour faire des filtres	++	+
Précision des temps d'accès	+	++
Prend en compte la voirie à l'arrivée	-	++
facilement intégrable	++	+

Grâce à ce stage, j'ai pu découvrir le travail dans une entreprise d'informatique. Il était particulièrement intéressant d'assister au changement de méthode de management qui ont eu lieu au cours de mon stage avec l'arrivée des feature teams. Cela m'a permis de me familiariser avec la méthode agile et d'être plus à l'aise au sein d'une équipe.

J'ai également acquis des compétences techniques au cours de ce stage. J'ai appris comment développer une API, mais aussi comment la tester et comment optimiser ses performances. De plus j'ai grandement améliorer mes connaissances dans plusieurs langages informatiques comme le C++ et le python.

Enfin j'ai beaucoup apprécié de travailler sur les systèmes d'information voyageurs. Les problématiques qui y sont soulevées m'ont motivées car elles mêlent à la fois des challenges techniques mais aussi des aspects plus liés à l'utilisateur. Ce qui ne va pas sans me rappeler les objectifs de l'option urbaniSTIC.

Ce stage m'a également conforté dans l'idée que je voulais continuer à approfondir mes connaissances en informatique au cours de mes études. J'ai beaucoup aimé travailler dans une équipe de recherche opérationnelle et j'aimerais continuer sur cette voie en faisant des études me permettant de faire de la recherche.

Table des figures

I.1	Organisation au sein de Kisio	4
I.2	Organisation des feature team	5
I.3	Architecture du SIV proposé par Kisio Digital	6
I.4	Innovation ouverte avec l'AI navitia.io	7
I.5	Historique de Canal tp	8
I.6	Architecture de Navitia	8
II.1	Isochrone à 8h00	10
II.2	Isochrone à 00h30	10
II.3	Isochrone vectoriel	11
II.4	Heat map	11
II.5	Cercles autour des points d'arrêts	13
II.6	Multipolygone obtenu après fusion des cercles	14
II.7	Multiisochrone	14
II.8	Isochrone entre deux durées	15
II.9	Isochrone traversant une rivière	16
II.10	Heat map respectant la voirie	16
II.11	Heat map résolution resolution = 50	17
II.12	Heat map resolution = 150	17
II.13	Coordonnées à la surface de la Terre	19
II.14	Grille à la surface d'une sphère	20
II.15	point projeté	20
II.16	Triangle rectangle spérique	21
II.17	Cercle à la surface d'une sphere	21
II.18	Relation rayon ouverture d'angle	21
II.19	Interface de google CPU profiler	22
II.20	Profile de temps de réponse avant optimisation pour l'API isochrones	23
II.21	Nombre de polygones pour un ordre aléatoire	24
II.22	Nombre de polygones lorsqu'on ordonne les points d'arrêts	24
II.23	En rouge : temps de réponse avant tri et rééchantillonage, en bleu : temps de réponse après	25
II.24	En rouge : taille du flux avant tri et rééchantillonage, en bleu : taille du flux après	25
II.25	En rouge : temps de réponse avant utilisation de la formule simplifiée, en bleu : temps de réponse après	26
II.26	Interface de Navitia explorer	27
II.27	Navitia explorer : onglet isochron	27
II.28	page d'accueil de Navitia playground	28
II.29	Page de création de requête	28
II.30	Navitia playground : requête d'isochrone	29
II.31	Navitia playground : isochrone isolé	29
II.32	Navitia playground : isochrones groupés	30
II.33	Navitia playground : requête d'heat maps	30

Bibliographie

- [1] Canal tp, mai 2016. www.canaltp.fr.
- [2] senaweb, juillet 2016. <http://www.semaweb.fr>.
- [3] wikipedia, avril 2015. fr.wikipedia.org/wiki/GeoJSON.
- [4] Marianne Greff. *Elements de géodésie mathématiques*, 2007-2008.