Rapport final de SAÉ - NaviGator

Réalisé par

Loris CAZAUX, Thomas NALIX, Maxence TOURNIAYRE

Projet encadré par

Malo GASQUET, Romain LEBRETON

Pour l'obtention du BUT Informatique

Année universitaire 2022 - 2023

Rapport SAE NaviGator

Cazaux - Nalix - Tourniayre

Remerciements

Nous tenons à remercier chaleureusement Romain LEBRETON et Malo GASQUET pour leur aide précieuse durant ce projet. Grâce à leurs conseils judicieux, leurs précieuses explications et leur soutien renouvelé, nous avons pu aboutir à un résultat qui nous satisfait.

Nous ne pouvons que nous réjouir des réponses qu'ils nous ont proposées (malgré le grand nombre de questions), car elles nous ont permis d'améliorer considérablement notre travail et de le rendre plus pertinent.

Nous leur sommes très reconnaissants pour le temps et l'énergie qu'ils ont consacrés à accompagner notre projet et nous permettre d'atteindre nos objectifs.

M. LEBRETON et M. GASQUET nous ont été d'une grande aide et encore une fois, nous les remercions.

Rapport SAE NaviGator

Cazaux - Nalix - Tourniayre

Résumé

NaviGator est une application web respectant les bonnes pratiques de performances et de qualité logicielle, avec une interface ergonomique offrant des fonctionnalités utiles pour les utilisateurs pour planifier facilement leurs voyages en France. De plus, la sécurité de l'application est prise en compte avec une vérification des failles de sécurité. Enfin, le bilan écologique de l'application web est considéré comme une réduction de l'impact environnemental pour les conducteurs. NaviGator est donc un outil pratique et responsable pour tous les voyageurs en France.

Mots clés
application web, performances, qualité logicielle, interface ergonomique, fonctionnalités, sécurité, bilan écologique, impact environnemental.

NaviGator is a web application that adheres to good performance and software quality practices, with an ergonomic interface offering useful features for users to easily plan their trips in France. In addition, the application's security is taken into account with a check for security vulnerabilities. Finally, the ecological impact of the web application is considered as a reduction of the environmental impact for drivers. NaviGator is thus a practical and responsible tool for all travellers in France.

Keywords
web application, performance, software quality, ergonomic interface, features, security, ecological impact, environmental impact.

Rapport SAE NaviGator

Cazaux - Nalix - Tourniayre

Table des matières

Glossaire

Dijkstra : Un algorithme de recherche de chemin qui calcule le chemin le plus court entre un nœud de départ et un nœud d'arrivée dans un graphe pondéré. L'algorithme Dijkstra utilise une approche de recherche en largeur d'abord (BFS). Il garantit de trouver le chemin le plus court si les coûts des arêtes sont tous positifs.

A* : algorithme de recherche de chemin utilisé pour résoudre des problèmes d'optimisation de trajet en explorant les noeuds d'un graphe en choisissant le prochain noeud en fonction du coût total jusqu'à présent, combinant le coût de déplacement de chaque étape et une estimation heuristique du coût restant. L'algorithme A* utilise une approche de recherche en profondeur d'abord (DFS). Il garantit de trouver le chemin le plus court si la fonction heuristique est admissible et cohérente.

OpenSet : il s'agit de la liste ouverte présente dans l'algorithme A* qui contient l'ensemble des nœuds qui ont été découverts mais pas encore visités par l'algorithme. C'est une liste de nœuds triée par leur coût total (g + h) estimé pour atteindre l'objectif.

Pathfinding : le pathfinding pour un graphe consiste à trouver le chemin le plus court ou le plus optimal entre deux nœuds dans un graphe. Cela implique souvent de naviguer dans un réseau complexe de nœuds interconnectés en utilisant des algorithmes de recherche de chemin tels que Dijkstra ou A* par exemple.

Harvesine : La formule de Haversine est une formule mathématique utilisée pour calculer la distance entre deux points sur une sphère. Elle est souvent utilisée dans le contexte d'une heuristique pour l'algorithme A* lorsqu'on cherche à trouver un chemin optimal entre deux points sur la surface de la Terre. La formule prend en compte la latitude et la longitude de deux points pour calculer leur distance en utilisant la courbure de la Terre comme référence.

Versioning : les systèmes de contrôle de version (versioning en anglais ou versionnage en français) sont une catégorie d'outils logiciels qui permettent de gérer les changements apportés à un code source au fil du temps.

Impact environnemental : mesure des effets environnementaux d'un logiciel sur l'air, l'eau, le sol et les ressources naturelles.

Index (base de données) : Une structure de données qui permet d'accélérer les recherches et les requêtes dans une table en permettant un accès rapide aux

données en fonction de la valeur d'une ou plusieurs colonnes. Il s'agit d'une copie partielle des données dans une table, organisée de manière à faciliter la recherche et la récupération des données. Les index peuvent grandement améliorer les performances d'une base de données, mais peuvent ralentir les opérations d'insertion, de mise à jour et de suppression. Il est donc important de considérer soigneusement la création d'index en fonction des besoins de l'application et de la fréquence d'utilisation de la table.

xDebug : Un outil open-source de débogage pour PHP. Il peut être utilisé pour analyser le code PHP en cours d'exécution, identifier les erreurs de code et optimiser les performances en surveillant le temps d'exécution et la consommation de mémoire.

API (Application Programming Interface) : Une API est une interface de programmation d'application qui permet à différents logiciels ou services web de communiquer entre eux et d'échanger des données. Les API externes sur le web sont généralement utilisées pour permettre à des applications tierces d'accéder à des fonctionnalités ou des données spécifiques d'une plateforme web ou d'un service en ligne. Les développeurs peuvent utiliser ces API pour créer des applications qui tirent parti de ces fonctionnalités et données sans avoir à recréer toute l'infrastructure de l'application elle-même.

Introduction

Le développement de logiciels est un processus complexe et en constante évolution qui exige une attention particulière pour les performances, la qualité, l'ergonomie, la sécurité et l'impact environnemental*. Dans ce contexte, notre projet consiste à évaluer et à optimiser une application Web existante, nommée NaviGator, qui calcule l'itinéraire le plus court entre deux communes en France.

Notre projet répond à plusieurs enjeux et besoins, tels que l'amélioration de l'expérience utilisateur, l'optimisation des performances, la garantie de la qualité logicielle, la sécurité et la prise en compte de l'impact environnemental. Il propose également des fonctionnalités supplémentaires telles que l'estimation du temps et des coûts impliqués, une trace visuelle de l'itinéraire, la possibilité d'entrer des itinéraires avec plusieurs points intermédiaires et de sauvegarder des itinéraires favoris.

Pour atteindre ces objectifs, notre rapport est divisé en quatre parties distinctes.

Tout d'abord, nous présenterons une analyse de l'application NaviGator afin d'identifier les points à améliorer.
Ensuite, dans le cahier d'optimisation, nous détaillerons les solutions pour améliorer les performances, la qualité logicielle, l'ergonomie, la sécurité et l'impact environnemental.
Par la suite, la troisième partie exposera notre méthodologie et notre organisation du projet, y compris la gestion d'équipe, la méthode de développement et les outils utilisés.
Finalement, nous examinerons les impacts environnementaux de l'application et proposerons des recommandations pour les réduire.

1. Analyse

La partie d'analyse a été une part très importante. En effet, c'est à partir d'elle que tout notre travail a découlé. Nous verrons donc dans cette partie comment nous avons identifié les différentes étapes par lesquelles nous sommes passés afin d'optimiser l'application. Nous verrons point par point comment nous avons identifié ces problèmes.

Afin d'identifier les points à optimiser, nous avons dans un premier temps regardé le temps que prend chaque partie en temps. Pour ce faire, nous n'avons pas utilisé xDebug* dans un premier temps, mais une classe utilitaire écrite par nous. De cette manière, il nous a été aisé de voir les parties qui prennent le plus de temps.

Au niveau de la base de données, cela a été le même principe. Nous avons utilisé EXPLAIN, une commande SQL permettant de connaître en détail l'exécution de la requête. De ce fait, nous avons pu déterminer si oui ou non les index* étaient utilisés.

1. Point d'amélioration numéro 1

Le tout premier point et qui est sans doute le plus simple à mettre en place réside dans le fait de mettre des index sur certaines colonnes de la base de données. Cela va donc permettre d'améliorer grandement les requêtes en base de données.

2. Point d'amélioration numéro 2

Un autre élément tout aussi important est de tout simplement changer d'algorithme. En effet, Dijkstra* est un très bon algorithme, mais A* * est encore meilleur lorsqu'il s'agit de graphe planaire et d'autant plus s'il s'agit d'une représentation d'un réseau routier. Cela permet ainsi d'améliorer grandement le temps d'exécution puisque le nombre d'itérations est bien moins important qu'avec Dijkstra. Évidemment, pour que A* soit efficace, il faut qu'il soit accompagné d'une heuristique adaptée. On peut par exemple utiliser

3. Point d'amélioration numéro 3

Un autre point un peu lié au premier est de ne pas mettre les colonnes qu'on n'utilise pas dans la requête "select". Cela permet encore une fois de faire gagner du temps mais aussi de la mémoire.

2. Cahier d’optimisation

Nous avons basés nos tests de rapidité par un set très simple composé de 6 itinéraires. Nous les avons choisies par leurs fiabilités, allant d’un nombre de kilomètres de 0, à la plus grande distance de France : 1 301.83 km.
De plus, pour faciliter la relecture ainsi que l’analyse, nos tests sont réalisés sur la vue initiale, qui ne prend pas en compte les sens interdits.
Nous avons basé uniquement nos tests sur la vitesse, cependant, il y a un aspect mémoire que nous aborderons plus en détail dans la suite de cette partie.

2.1. Algorithme Dijkstra

2.1.0. Code initial

1. Statistiques

	Itérations	**Distance**	**Temps**
Menton-Menton	1	0km	15.09s
Montpellier-Nîmes	X	X	> 120s
Montpellier-Orange	X	X	> 120s
Montpellier-Lyon	X	X	> 120s
Montpellier-Paris	X	X	> 120s
Menton-Porspoder	X	X	> 120s

Ci-dessus les statistiques sur quelques trajets de la version initiale.

Le code initial comporte de nombreux problèmes au niveau de l'algorithme, mais aussi en base de données.
Notre façon de procéder sera assez simple, nous allons tout d'abord revoir l'architecture et les requêtes au sein de la base de données. Et quand nous estimons avoir fait le maximum, nous commencerons à nous occuper de la partie php.
C'est pour nous une bonne manière de procéder, car d'après nos analyses réalisées avec XDebug*, le lien entre postgres et le site est celui qui prend le plus de temps.

2.1.1. Optimisation de la requête BD

1. Explication

Afin d'optimiser et de réduire le temps de la requête, nous avons eu l'idée d'ajouter un index sur les tables, notamment sur les colonnes geom permettant ainsi d'augmenter radicalement la vitesse de calcul. Aussi, nous avons eu l'idée de changer radicalement la requête étant donné qu'elle possédait un *UNION* et un *produit cartésien* ce qui n'est pas recommandé pour de l'optimisation.
Étant donné le nombre d'itération, nous avons créé des index permettant d'augmenter la vitesse de traitement. Et plus spécifiquement, pour les geom, ce qui est très intéressant pour notamment faire des calculs (http://postgis.net/workshops/postgis-intro/indexing.html).

2. Mise en oeuvre

Nous avons changé la requête issue de la méthode *getVoisins* comme suit :

```
SELECT nr2.gid as noeud_routier_gid,
    tr.gid as troncon_gid,
    tr.longueur FROM noeud_routier nr
JOIN troncon_route tr ON ST_DWithin(nr.geom, tr.geom, 0.0001)
JOIN noeud_routier nr2 ON ST_DWithin(tr.geom, nr2.geom, 0.0001)
WHERE nr.gid = :gidTag AND nr2.gid != :gidTag;
```

Ainsi que l'ajout d'index comme dit précédemment.

```
CREATE INDEX troncon_route_geom_idx ON troncon_route USING GIST (geom);
CREATE INDEX noeud_routier_geom_idx ON noeud_routier USING GIST (geom);
create index noeud_commune_nom_comm_index on nalixt.noeud_commune (nom_comm);
create index noeud_routier_id_rte500_index on nalixt.noeud_routier (id_rte500);
```

3. Le résultat

	Itérations	**Distance**	**Temps**
Menton-Menton	1	0km	0.04s
Montpellier-Nîmes	9387	48.84km	23.88s
Montpellier-Orange	27160	102.05km	67.35s
Montpellier-Lyon	X	X	> 120s
Montpellier-Paris	X	X	> 120s
Menton-Porspoder	X	X	> 120s

Nous pouvons d'ores et déjà constater une nette évolution par rapport à la solution initiale. En effet, bien que le calcul soit long, il est maintenant envisageable de rechercher un chemin entre 2 villes séparées par 100 kilomètres.
Cependant, nous remarquons qu'une seule itération prend certe, peu de temps (0.0025s), mais si l'on multiplie cette valeur par un nombre d'itération tel que 100 000, le résultat s'approche de 250s, sans compter d'autres facteurs de ralentissement de l'algorithme. Ainsi, notre objectif principal est de diminuer drastiquement aussi bien le temps, que le nombre d'itération réalisé par l'algorithme pour trouver le plus court chemin.

Node type	**Count**	**Time spent**	**% of query**
Index Scan	3	3.075ms	88.24%
Nested Loop Inner Join	2	0.412ms	11.8%

Ci-dessus, nous pouvons voir la trace de la requête, comme nous pouvons le constater, une seule requête prend 4.3ms à s'exécuter, ce qui est considérable. De plus, 88% du temps est passé dans le scan des index. Si nous allons plus loin dans l'analyse, *Index Scan using troncon_route_geom_idx*, c'est cet index qui montre des difficultés.

4. Piste à explorer

Nous avons remarqué que les calculs des nœuds des voisins sont effectués à chaque requête. Nous pensons donc qu'il faudrait privilégier une vue matérialisée qui fasse préalablement l'ensemble de ces calculs pour augmenter les performances.

2.1.2. Ajout d'une vue matérialisée

1. Explication

Dans un second temps, nous avons essayé de trouver un moyen d'optimiser de lourds calculs coûteux que sont les calculs des voisins.
En effet, à chaque itération, la requête refait un calcul à l'aide de coordonnées géographiques des nœuds ce qui est l'équivalent d'une boucle faisant un lourd calcul à chaque fois qu'il cherche à calculer les voisins d'un nœud géographique.
C'est pourquoi nous envisageons de créer une vue matérialisée permettant de stocker tous les voisins en faisant un seul gros précalcul au préalable et ainsi de ne plus avoir à le faire à chaque fois.
Pour accélérer d'autant plus le traitement, nous avons ajouté des index à cette vue matérialisée afin de l'optimiser elle aussi dans les appels SQL du code PHP.

2. Mise en oeuvre

Nous avons donc changé de nouveau la requête issue de la méthode *getVoisins* comme suit :

```
SELECT noeud_routier_gid_2 as noeud_routier_gid, troncon_gid, longueur FROM
nalixt.calcul_noeud_troncon
WHERE noeud_routier_gid = :gidTag;
```

Création d'une vue matérialisée permettant de stocker tous les voisins en faisant un précalcul avec ses index.

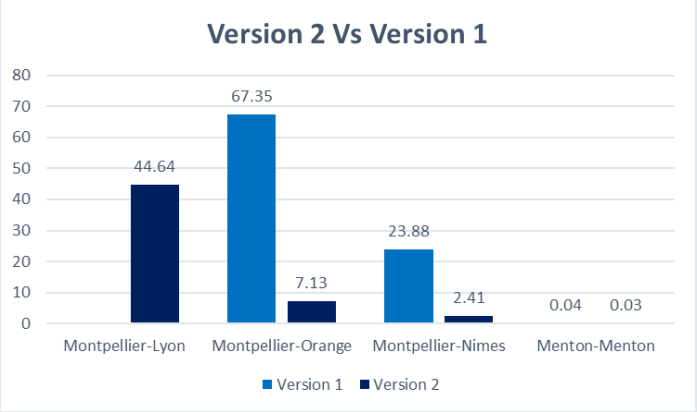
```
create index calcul_noeud_troncon_idx_nr_gid
  on nalixt.calcul_noeud_troncon (noeud_routier_gid);
```

```
CREATE MATERIALIZED VIEW nalixt.calcul_noeud_troncon AS
SELECT nr.gid as noeud_routier_gid,
nr2.gid as noeud_routier_gid_2,
tr.gid as troncon_gid,
tr.longueur,
tr.geom as troncon_coord
FROM nalixt.noeud_routier nr
JOIN nalixt.troncon_route tr ON tr.geom && ST_Expand(nr.geom, 0.0001) AND ST_DWithin(nr.geom,
tr.geom, 0.0001)
JOIN nalixt.noeud_routier nr2 ON nr2.geom && ST_Expand(tr.geom, 0.0001) AND
ST_DWithin(tr.geom, nr2.geom, 0.0001)
WHERE nr2.gid != nr.gid;
```

Notre vue est relativement simple à analyser. Pour chaque nœud routier, nous recherchons tous ses voisins en indiquant le tronçon qui permet l'accès. Cette vue compte environ 2 600 000 lignes.

3. Le résultat

	Itérations	Distance	Temps
Menton-Menton	1	0km	0.03s
Montpellier-Nîmes	9387	48.84km	2.41s
Montpellier-Orange	27160	102.05km	7.13s
Montpellier-Lyon	167777	286.67km	44.64s
Montpellier-Paris	X	X	> 120s
Menton-Porspoder	X	X	> 120s



Graphique comparant la version 2 à la version 1

Le fait d'avoir fait une vue matérialisée a permis d'augmenter radicalement la vitesse, permettant de faire un calcul sur une distance presque 3 fois plus élevée pour un temps plus court (67 secondes pour 102.05kilomètres => 44.64 secondes pour 286.67 kilomètres).
De plus, toujours dans l'idée d'optimiser le temps de traitement d'une seule itération, nous avons atteint une exécution bien plus rapide, atteignant 0.00026s.

Sans index			
Node type	**Count**	**Time spent**	**% of query**
Gather	1	24.289 ms	16.91%
Seq Scan	1	119.418 ms	83.1%

Avec index			
Node type	**Count**	**Time spent**	**% of query**
Bitmap Heap Scan	1	0.019ms	32.76%
Bitmap Index Scan	1	0.04ms	68.97%

Nous allons préciser les résultats obtenus en analysant la trace de la requête SQL sur la vue matérialisée.
L'ajout de l'index améliore de manière significative le requête, passant de 143ms à 0.023ms. Ce résultat est en effet prévisible, étant donné que la vue possède 2 600 000 lignes, il est primordial d'indexer les gid des nœuds routiers pour favoriser la vitesse de traitement.

5. Piste à explorer

Nous pensons avoir atteint notre premier objectif qui était d'optimiser le côté base de données. Nous allons dans les prochaines étapes nous concentrer sur la partie php.

2.1.3. Regroupement de 2 requêtes séparés

1. Explication

> A notre stade, une requête SQL ne peut mettre moins d'un certain temps. Que l'on récupère 1 ligne, ou 2000, l'optimisation réalisée avec les index fait que cela prend un temps minimum dans tous les cas.

Nous avons noté que dans le code existant, il y a 2 accès en base de donné par itération :
- *recupererParClePrimaire()*
- *getVoisins()*

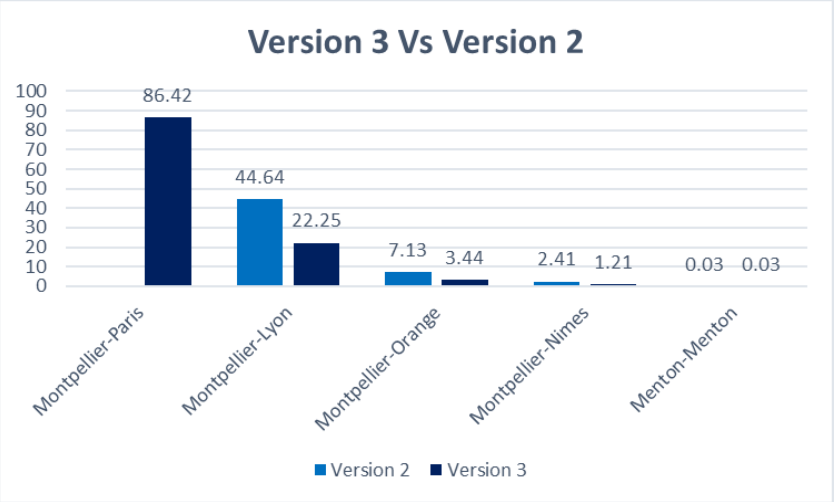
Étant limité par la base de données, si nous faisons qu'un seul accès en mémoire au lieu de 2, le temps de calculs serait potentiellement divisé par 2.
La fonction *recupererParClePrimaire()* crée automatiquement un objet NœudRoutier, qui implique lors de la création de récupérer ses voisins avec la méthode *getVoisins()*.

2. Mise en oeuvre

Dans cette mise à jour, nous avons supprimé l'attribut id_rte_500 car il n'est pas utile pour le moment lors des traitements que nous effectuons.
Nous avons également changé la caractéristique d'un NoeudRoutier en modifiant les attributs qui sont maintenant : gid, voisins[]
Pour finir, nous avons ajouté une méthode *getNoeudRoutier()* qui remplace les 2 anciennes (*getVoisins* et *recupererParClePrimaire*) et qui pour un gid donné, renvoi le nœud routier et ses voisins.

3. Le résultat

	Itérations	**Distance**	**Temps**
Menton-Menton	1	0km	0.03s
Montpellier-Nîmes	9387	48.84km	1.21s
Montpellier-Orange	27160	102.05km	3.44s
Montpellier-Lyon	167777	286.67km	22.25s
Montpellier-Paris	628014	687.69km	86.42s
Menton-Porspoder	X	X	> 120s



Graphique comparant la version 3 à la version 2

Comme prévu, nous avons divisé le temps d'exécution par 2. Nous sommes maintenant en capacité de calculer une distance de 687.69 kilomètres.
Cependant, ce qui est intéressant à constater, c'est le temps que nous mettons maintenant à calculer une distance telle que Montpellier Orange, qui ne prend plus que quelques secondes. Alors qu'initialement, nous mettions plus de 1 minute.

4. Piste à explorer

La prochaine étape serait sûrement de trouver un moyen de diminuer le nombre d'accès à la base de données, en effet, pour de longues distances telles que Paris → Limoges (366.51 km), le nombre d'accès en mémoire est de 409189. Il serait ainsi préférable de trouver une solution qui charge en cache une certaine quantité de données.

En plus d'une potentielle mise en cache, nous pouvons envisager une meilleure structure de données tel que le tas de Fibonacci :

- https://kbaile03.github.io/projects/fibo_dijk/fibo_dijk.html
- https://www.youtube.com/watch?v=fRpsjKCfQjE
- https://www.youtube.com/watch?v=6JxvKfSV9Ns
- https://www.youtube.com/watch?v=C-kr17luY2g

Nous pouvons également envisager la solution du multithreading qui est cependant très complexe.

2.1.4. Mise en cache des départements

1. Explication

En analysant la base de données, nous avons remarqué que chaque nœud routier pouvait être relié à un département. C'est pourquoi nous pensons qu'une mise en cache peut être envisagée permettant de diminuer le nombre d'accès à la base de données.
En pratique, nous pouvons constater qu'il y a 96 départements, allant d'un minimum de quelques millier de nœuds pour le département du 90, à des dizaines de milliers pour d'autres.
Ainsi, si l'on fait une recherche de plus court chemin, le système se chargera de charger l'ensemble des nœuds d'un département, faisant économiser plusieurs dizaines de milliers de requêtes à la base de données.

2. Mise en oeuvre

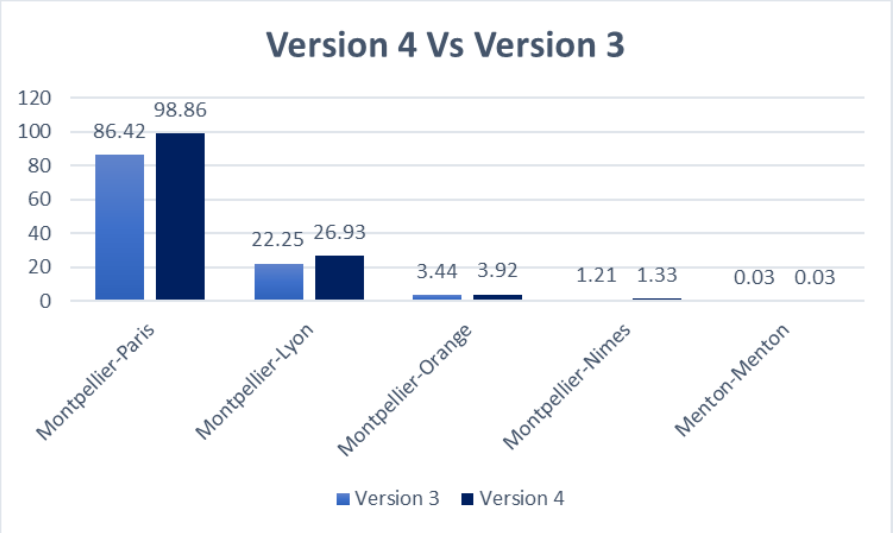
Pour réaliser cela, à chaque itération, l'algorithme vérifie le numéro de département du nœud Courant. Si ce département n'est pas chargé dans cache, une requête à la base de données est effectuée afin de charger l'intégralité des nœuds ainsi que de leur voisins dans un cache.

Voici la façon dont les départements sont stockés. Pour chaque département, un gid est stocké, et pour chacun des gid, ses voisins ainsi que le *troncon_gid*, la *distance*, ... sont indiqués

```
[
  numDepartement => [
     gid => [
           ...
      ],
      gid2 => [
         ...
      ]
   ],
   numDepartement2 => [ ... ]
]
```

3. Les résultats

	Itérations	Distance	Temps
Menton-Menton	1	0km	0.03s
Montpellier-Nîmes	9387	48.84km	1.33s
Montpellier-Orange	27160	102.05km	3.92s
Montpellier-Lyon	167777	286.67km	26.93s
Montpellier-Paris	628014	687.69km	98.86s
Menton-Porspoder	X	X	> 120s



Graphique comparant la version 4 à la version 3

Comme nous pouvons le constater, la mise en cache a empiré les temps d'exécution. Nous l'expliquons simplement par le fonctionnement de dijkstra. En effet, cet algorithme va rechercher de manière circulaire autour du point, et ainsi charger de nombreux nœuds qui ne sont pas utiles. Ainsi, pour Montpellier Paris, presque tous les départements de la France sont chargés, causant un surplus de consommation de mémoire et une exécution plus lente.
Il est maintenant intéressant de s'intéresser à l'impact mémoire de notre solution. Charger tous les départements de la France peut causer un débordement de mémoire, nous sommes en effet contraints de désactiver la limite que xampp nous impose. Dans un cas tel que Montpellier Paris, nous utilisons plus de 500mo de mémoire, ce qui est évidemment énorme.
Cependant, il y a un rapport à évaluer, est-il préférable de faire 10 000 accès à la base de données, ou bien qu'un seul ?

La question est intéressante et nous avons décidé de privilégier cette solution pour différente raisons :
Tout d'abord, nous pensons qu'un impact mémoire plus important ne pose pas de problème car nous n'avons pas pour but de le rendre publique. Cela pourrait poser un éventuel problème à grande échelle.
Cependant, l'analyse d'un tel problème est très complexe et dépend de plusieurs facteurs tels que la latence du réseau, les performances du serveur web et du serveur de la base de données, le nombre d'utilisateurs de l'application...
En effet, charger 500 Mo de données à partir du serveur web peut entraîner une charge importante sur les ressources du serveur, notamment la bande passante et la mémoire. Mais, si notre base de données est bien optimisée et configurée pour gérer un grand nombre d'accès, alors 10 000 accès peuvent être gérés de manière efficace sans causer de problèmes de performance.
Ainsi, il n'existe pas de réponse claire pour notre problème.

Il existe aussi des points positifs non négligeables, nous pouvons par exemple se servir de cette solution sur une extension qui est l'ajout d'étapes. En effet, les nœuds des départements étant chargés, une destination telle que Montpellier→Orange→Montpellier ne prendrait théoriquement que le temps de calculs coté php. Tandis qu'avec une solution plus classique, le traitement prendra 2 fois plus de temps.
Ainsi, la mise en cache est un avantage considérable et ne pose plus aucun problème dans notre cas, juste par la simple présence de cette extension.

4. Piste à explorer

Nous pensons que l'algorithme de Dijkstra nous pose beaucoup de problèmes comme expliqué précédemment et c'est pourquoi nous aimerions nous tourner vers A*, qui est un algorithme plus optimisé pour notre problématique.

2.2. Algorithme A*

2.2.1. Implémentation de A*

1. Explication

Nous avons décidé de changer d'algorithme de chemin le plus rapide en utilisant le célèbre A* (https://en.wikipedia.org/wiki/A*_search_algorithm). En effet, il est plus adapté car il ne cherche visite pas une zone "circulaire" comme dijkstra qui peut à terme devenir très lourd pour la mémoire.
Pour commencer cette première implémentation de A*, nous n'allons pas tout de suite utiliser notre système de mise en cache.

Il nous a fallu changer la vue et ajouter 2 nouveaux attributs : *noeud_coord* et *noeud_voisin_coord* qui sont des traductions des coordonnées geom. Ils vont servir à faire le calcul heuristique qui est très important pour A*.

Concernant l'heuristique, nous avons dû en choisir un qui était admissible avec notre algorithme. Dans le cas des algorithmes de pathfinding*, il est nécessaire afin qu'un heuristique soit admissible que la fonction heuristique ne surestime jamais le coût pour atteindre le but. Dit autrement, le coût que la fonction estime pour atteindre le but n'est pas plus élevé que le coût le plus bas possible à partir du point actuel du chemin.

Nous allons préciser les différents types d'heuristiques possibles ainsi qu'expliquer brièvement comment elles fonctionnent :

1. **Haversine** : L'heuristique de Haversine est basée sur la distance à vol d'oiseau entre deux points sur la surface terrestre, en utilisant la latitude et la longitude de ces points. Elle calcule la distance entre les deux points en suivant un arc de cercle sur la surface de la Terre, en tenant compte de sa courbure.

2. **Loxodromique** : L'heuristique loxodromique est basée sur la distance le long d'une ligne droite sur la surface d'un cylindre enroulé autour de la Terre, en utilisant la latitude et la longitude de deux points. Elle est également connue sous le nom de "distance en ligne droite à angle constant" ou "route orthodromique".

3. **Euclidienne** : L'heuristique euclidienne est basée sur la distance en ligne droite entre deux points dans un espace euclidien à deux dimensions. Elle est calculée en utilisant la racine carrée de la somme des carrés des différences entre les coordonnées x et y des deux points.

4. **Manhattan** : L'heuristique de Manhattan est basée sur la distance entre deux points en utilisant uniquement les mouvements horizontaux et verticaux, sans prendre en compte les diagonales. Elle est calculée en sommant les différences absolues des coordonnées x et y entre les deux points. Elle est également connue sous le nom de "distance de la grille" ou "distance de taxi".

En outre, nous aurions pu sélectionner l'heuristique loxodromique qui peut être intéressante, cependant, nous avons jugé que la plus polyvalente serait celle de Harvesine*.
Harvesine satisfait la propriété énoncée plus haut et est ainsi admissible. La calcul se fait entre les coordonnées géographiques du nœud actuel et du nœud d'arrivée. Cette heuristique nous a permis d'économiser énormément de temps en minimisant le nombre d'itérations.

2. Mise en oeuvre

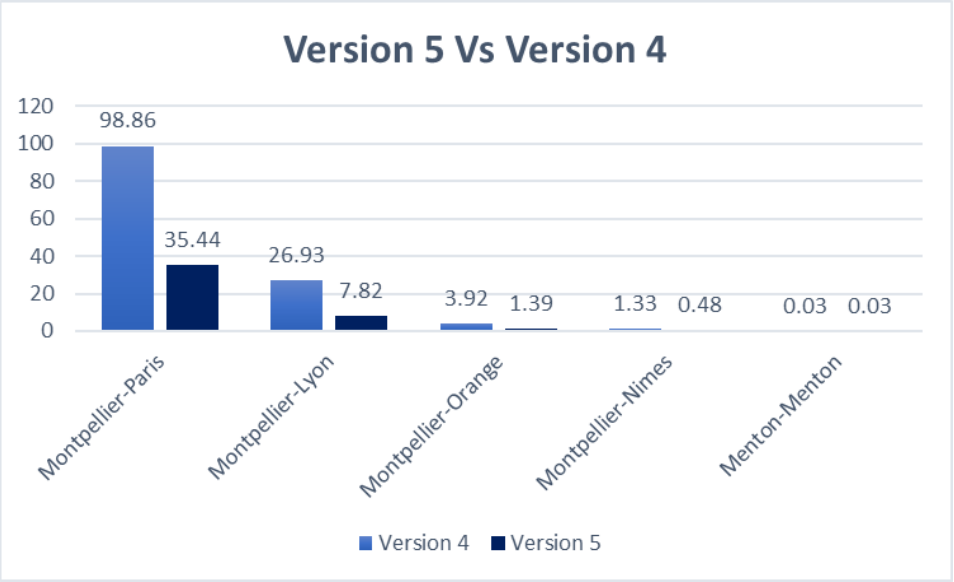
Comme expliqué brièvement précédemment, nous avons ajouté plusieurs attributs à la vue, notamment un attribut département à la car pour la mise en cache des départements nous avions besoin d'avoir quelque part le numéro de département du nœud actuel.
Nous avons aussi ajouté les coordonnées du nœud courant ainsi que les coordonnées du nœud voisin afin de ne pas à avoir à faire des requêtes externes pour accéder aux coordonnées des nœuds.
Nous avons également pensé à enlever les tronçons qui sont en **SENS INVERSE**. Même si ce n'est qu'environ 3000 - 4000 lignes.

```
CREATE MATERIALIZED VIEW nalixt.calcul_noeud_troncon AS
SELECT nr.gid                                                    AS noeud_courant_gid,
            concat(st_x(st_astext(nr.geom)::geometry),  ';',  st_y(st_astext(nr.geom)::geometry))      AS
noeud_courant_coord,
    nr2.gid                                                  AS noeud_voisin_gid,
             concat(st_x(st_astext(nr2.geom)::geometry),  ';',  st_y(st_astext(nr2.geom)::geometry))  AS
noeud_voisin_coord,
    tr.gid                                                AS troncon_gid,
    tr.longueur                                             AS longueur_troncon,
    tr.geom                                               AS troncon_coord,
    "left"(nc.insee_comm::text, 2)                                  AS num_departement
FROM nalixt.noeud_routier nr
      JOIN nalixt.noeud_commune nc ON nr.insee_comm::text = nc.insee_comm::text
      JOIN nalixt.troncon_route tr ON tr.geom && st_expand(nr.geom, 0.0001::double precision) AND
                        st_dwithin(nr.geom, tr.geom, 0.0001::double precision)
        JOIN nalixt.noeud_routier nr2 ON nr2.geom && st_expand(tr.geom, 0.0001::double precision)
AND
                         st_dwithin(tr.geom, nr2.geom, 0.0001::double precision)
WHERE nr2.gid <> nr.gid AND tr.sens <> 'Sens inverse';
```

3. Résultats

	Itérations	Distance	Temps
Menton-Menton	1	0km	0.03s
Montpellier-Nîmes	2012	48.84km	0.48s
Montpellier-Orange	6518	102.05km	1.39s
Montpellier-Lyon	31631	286.67km	7.82s
Montpellier-Paris	152400	687.69km	35.44s
Menton-Porspoder	X	X	> 120s

Nous pouvons tout de suite constater une énorme évolution en terme de visite des nœuds en comparaison avec dijkstra.
Nous pouvons par exemple comparer le nombre d'itération, et constater une très grande évolution, passant de 628 014 à 152 400, soit 4 fois moins.



Graphique comparant la version 5 à la version 4

Nous pouvons également constater d'après ce graphique une amélioration significative sur les très grande distance, divisant le temps de calcul par 3. Cependant, cette première implémentation ne permet pas encore le calcul de la plus grande distance de France : Menton-Porspoder.

4. Piste à explorer

La prochaine piste à explorer est la mise en cache des données pour permettre d'augmenter radicalement la vitesse de traitement.

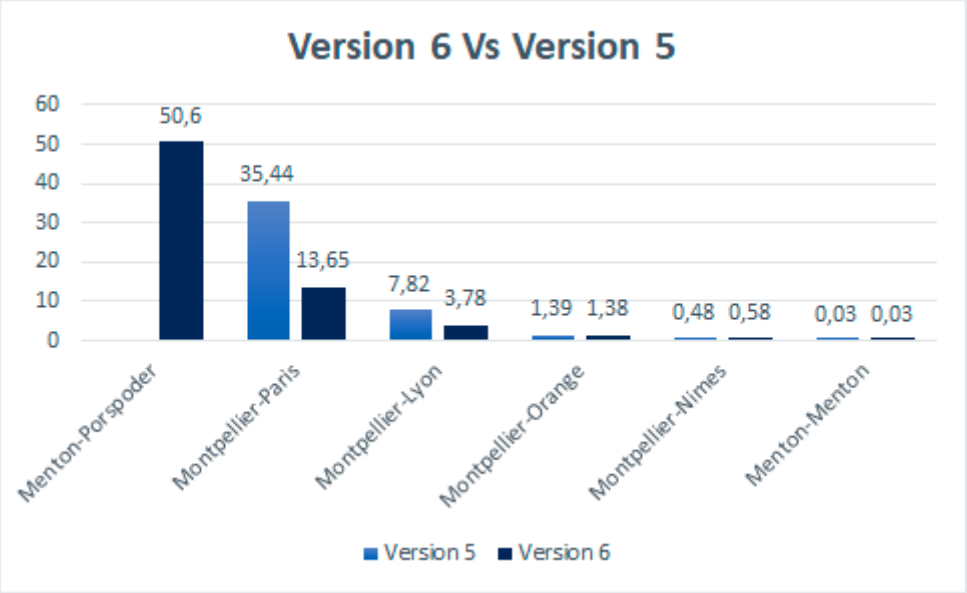
2.2.2. Implémentation du cache par département

1. Explication

De la même manière qu'avec Dijkstra, nous avons implémenté le même algorithme de mise en cache.
Nous avons décidé de réimplémenter la mise en cache, car contrairement a Dijkstra, la zone de recherche ne se fait pas de manière circulaire, mais s'oriente vers le nœud d'arrivée. Ainsi, A* garantie de charger beaucoup moins de départements, rendant notre système très puissant.

2. Résultats

	Itérations	**Distance**	**Temps**
Menton-Menton	1	0km	0.03s
Montpellier-Nîmes	2012	48.84km	0.58s
Montpellier-Orange	6518	102.05km	1.38s
Montpellier-Lyon	31631	286.67km	3.78s
Montpellier-Paris	152406	687.69km	13.65s
Menton-Porspoder	626762	1301.83km	50.60s



Graphique comparant la version 6 à la version 7

Nous pouvons constater qu'il reste des d'améliorations à faire, en effet, sur les très petites distances, cela est moins optimisé que sans le cache, bien que minime. Cependant, cela nous a permis de faire le calcul de la plus longue distance de France : Menton-Porspoder avec comme premier temps 50.60s.

2.2.3. Changement vers une structure BST et optimisation en base de données

1. Explication

Après avoir observé les temps que prenait chaque fonction, nous avons constaté que ce qui posait problème était le mécanisme permettant de récupérer le nœud voisin ayant la plus petite distance avec le nœud courant. Nous avons donc cherché un moyen d'accélérer cela. Nous sommes donc partis sur la structure de données Binary Search Tree (arbre binaire de recherche). Celle-ci semblait prometteuse puisqu'elle affichait une complexité de l'ordre de O(h) avec h la hauteur de l'arbre. Cela permettait donc d'obtenir le minimum en un temps très rapide et que le tri se fasse automatiquement lors de l'ajout et de la suppression.

En comparaison, auparavant, nous devions trier la structure entièrement (avec une complexité de O(n*log n)) puis itérer sur la liste des nœuds à visiter afin de trouver celui ayant la distance correspondante avec le plus petit élément de la liste précédemment triée. Cela était donc extrêmement coûteux en temps.

Du côté de la base de données, nous avons tout d'abord changé la nomenclature des attributs de la vue et nous avons ajouté 2 colonnes num_departement_depart et num_departement_arrivee. Nous avons aussi enlevé le **SENS INVERSE** le temps d'être sûr que la partie code est fonctionnelle et totalement optimisée.

Enfin nous avons créé une nouvelle vue matérialisée stockant un gid et son département permettant de remplacer notre requête qui faisait un LEFT sur le département, ici il ne sera appelé que lors de la création de la vue et plus après.
Cette vue permet de récupérer le département du nœud dans lequel nous nous trouvons, ce département sera ensuite donné en argument à une requête faisant appel à la vue principale qui donne tous les tronçons d'un département donné.
Comme dit précédemment nous pouvions garder notre requête qui faisait exactement ce que fait la vue mais c'est plus optimisé de créer une vue et ainsi faire une requête très simple (select from where).

2. Mise en oeuvre

La structure de données BST a donc été codée en suivant l'implémentation en langage naturel disponible sur internet. Celle-ci nous a permis de gagner un temps considérable, faisant passer notre temps de calcul de 50 secondes pour Menton - Porspoder à 19 secondes.

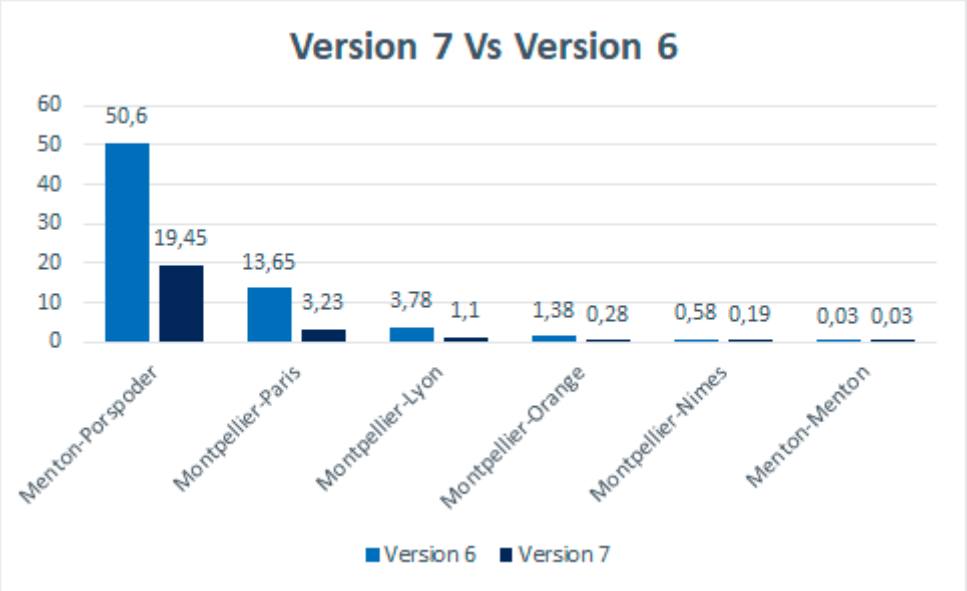
```
CREATE MATERIALIZED VIEW nalixt.noeuds_from_troncon AS
SELECT
nr.gid AS noeud_depart_gid,
concat(st_x(st_astext(nr.geom)::geometry), ';', st_y(st_astext(nr.geom)::geometry))   AS
noeud_depart_coord,
nr2.gid AS noeud_arrivee_gid,
concat(st_x(st_astext(nr2.geom)::geometry), ';', st_y(st_astext(nr2.geom)::geometry)) AS
noeud_arrivee_coord,
tr.gid AS troncon_gid,
tr.longueur AS longueur_troncon,
tr.geom AS troncon_coord,
"left"(nc.insee_comm::text, 2) AS num_departement_depart,
"left"(nc2.insee_comm::text, 2) AS num_departement_arrivee
FROM nalixt.troncon_route tr

JOIN nalixt.noeud_routier nr ON nr.geom && st_expand(tr.geom, 0.0001::double precision) AND
st_dwithin(tr.geom, nr.geom, 0.0001::double precision)
JOIN nalixt.noeud_routier nr2 ON nr2.geom && st_expand(tr.geom, 0.0001::double precision) AND
st_dwithin(tr.geom, nr2.geom, 0.0001::double precision)
JOIN nalixt.noeud_commune nc ON nr.insee_comm::text = nc.insee_comm::text
JOIN nalixt.noeud_commune nc2 ON nr2.insee_comm::text = nc2.insee_comm::text

WHERE nr2.gid <> nr.gid AND nr.gid < nr2.gid;
```

```
CREATE materialized view nalixt.noeud_gid_dep AS
SELECT nc.gid, "left"(nc.insee_comm::text, 2) as num_departement
FROM nalixt.noeud_routier nc
```

```
– nalixt.noeuds_from_troncon
create index noeuds_from_troncon_idx_departement_depart
   on nalixt.noeuds_from_troncon (num_departement_depart);

create index noeuds_from_troncon_idx_departement_arrivee
   on nalixt.noeuds_from_troncon (num_departement_arrivee);

– nalixt.noeud_gid_dep
create index noeuds_from_troncon_idx_departement
   on nalixt.noeud_gid_dep (num_departement);

create index noeuds_from_troncon_idx_gid
   on nalixt.noeud_gid_dep (gid);
```

3. Résultats

	Itérations	Distance	Temps
Menton-Menton	1	0km	0.03s
Montpellier-Nîmes	1928	48.84km	0.19s
Montpellier-Orange	6426	102.05km	0.28s
Montpellier-Lyon	31332	286.67km	1.10s
Montpellier-Paris	151929	687.69km	3.23s
Menton-Porspoder	827365	1301.83km	19.45s



Graphique comparant la version 7 à la version 6

Nous pouvons constater que la structure BST nous a permis de faire d'énormes progrès sur le code, en effet nous avons divisé par plus de deux le temps de calcul sur les grosses distances. Néanmoins 19,45 secondes pour Menton - Porspoder reste trop grand et il faut encore réduire cette donnée.

4. Piste à explorer

Tas de fibonacci et optimisation de la base de donnée.

2.2.4. Changement de l'encodage des coordonnnées

1. Explication

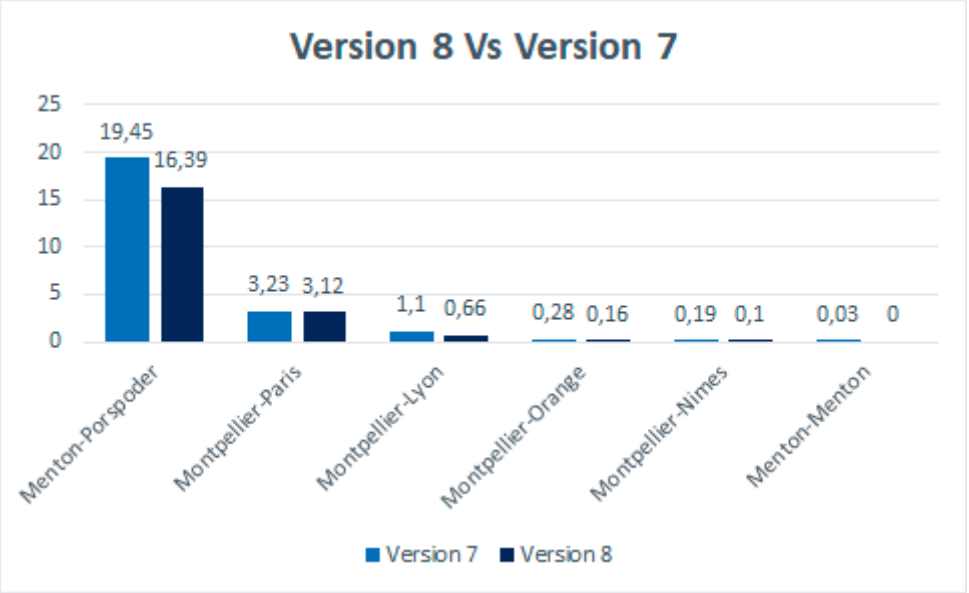
Nous avons scindé l'attribut des coordonnées du départ et de l'arrivée en deux nouveaux attributs qui sont leur longitude et leur latitude afin de ne pas à avoir à faire de calculs côté PHP pour récupérer la longitude et latitude.
En effet, il y avait un explode des coordonnés, encodés initialement de la manière suivante : "-5.3789478934; 43.3457262746". Cet explode est normalement de l'ordre de l'instantané. Cependant, lorsqu'il est appelé quelques millions de fois, des limites se font ressentir. Nous faisant perdre 1s sur la plus grande distance.

2. Modification

```
CREATE MATERIALIZED VIEW nalixt.noeuds_from_troncon AS
SELECT
nr.gid AS noeud_depart_gid,
st_x(st_astext(nr.geom)::geometry) AS noeud_depart_long,
st_y(st_astext(nr.geom)::geometry) AS noeud_depart_lat,
nr2.gid AS noeud_arrivee_gid,
st_x(st_astext(nr2.geom)::geometry) AS noeud_arrivee_long,
st_y(st_astext(nr2.geom)::geometry) AS noeud_arrivee_lat,
tr.gid AS troncon_gid,
tr.longueur AS longueur_troncon,
tr.geom AS troncon_coord,
"left"(nc.insee_comm::text,2) AS num_departement_depart,
"left"(nc2.insee_comm::text,2) AS num_departement_arrivee

FROM nalixt.troncon_route tr

JOIN nalixt.noeud_routier nr ON nr.geom && st_expand(tr.geom, 0.0001::double precision) AND st_dwithin(tr.geom, nr.geom, 0.0001::double precision)
JOIN nalixt.noeud_routier nr2 ON nr2.geom && st_expand(tr.geom, 0.0001::double precision) AND st_dwithin(tr.geom, nr2.geom, 0.0001::double precision)
JOIN nalixt.noeud_commune nc ON nr.insee_comm::text = nc.insee_comm::text
JOIN nalixt.noeud_commune nc2 ON nr2.insee_comm::text = nc2.insee_comm::text

WHERE nr2.gid <> nr.gid
AND nr.gid < nr2.gid;
```

3. Résultats

	Itérations	Distance	Temps
Menton-Menton	1	0km	0s
Montpellier-Nîmes	1928	48.84km	0.10s
Montpellier-Orange	6438	102.05km	0.16s
Montpellier-Lyon	31356	286.67km	0.66s
Montpellier-Paris	152063	687.69km	3.12s
Menton-Porspoder	630714	1301.83km	16.39s



Graphique comparant la version 8 à la version 7

Nous constatons que d'avoir scindé l'attribut des coordonnées de départ et d'arrivée en deux à permis d'alléger la charge en général pour tous les trajets permettant une légère réduction du temps qui est tout de même à noter.
En effet, l'appel à l'heuristique se fait extrêmement souvent, pour chaque itération, l'algorithme fait plusieurs requêtes à l'heuristique pour chacun des voisins du nœud courant, amenant généralement à deux voire trois fois plus de d'appel.
Ce changement, bien que minime, permet d'économiser sur les grands trajets, quelques secondes.

2.2.5. Structure PriorityQueue et optimisations diverses

1. Explication

Afin de pouvoir avoir les vitesses des tronçons pour le calcul de la distance, du temps et de la vitesse, nous stockons dans la vue un nouvel attribut nommé vitesse qui est une traduction de l'attribut vocation de la table tronçon en entier indiquant la vitesse maximum sur le tronçon (un réseau routier finalement.).
Nous avons également décidé de changer notre structure de donnée et de prendre une structure native a PHP, une priorityQueue. Cela nous a permis de gagner beaucoup de temps et nous faisons maintenant face à une limite sur les extract qui prennent 80% du temps total.

Nous avons décidé de changer radicalement notre vue matérialisée. Initialement, nous stockions un nœud, ainsi que tous ses nœuds voisins. Cependant, nous allons dès à présent les encoder de manière différente.
Il y aura donc une ligne par tronçon, avec pour chacun des tronçons, ses nœuds voisins. Nous allons donc pouvoir économiser de nombreuses lignes, passant de 2 600 000 à 1 300 000 lignes. Mais également beaucoup d'espace mémoire.

Nous avons optimisé le côté mémoire et vitesse de traitement en constatant une donnée très lourde et pas utilisée lors de la recherche du chemin le plus court. Il s'agit de la donnée *troncon_geom*. Théoriquement, ce changement va pouvoir nous faire économiser beaucoup d'espace mémoire.
L'affichage de l'itinéraire sur la carte fera donc appel à une toute nouvelle méthode dédiée qui fera des appels en base de données pour récupérer les geom à partir des *troncon_gid*.

Nous nous sommes également rendu compte que la fonction 'getDepartement' prenait beaucoup de temps. Nous avons donc optimisé l'appel de la fonction, ce qui nous a permis de gagner 2 secondes sur le temps total. Pour ce faire, nous avons vérifié avant même d'itérer sur l'ensemble des départements chargés si le département du nœud n'était pas celui du département courant.

2. Mise en oeuvre

Pour l'implémentation, nous nous sommes basés sur la classe existante dans PHP : SplPriorityQueue. Nous avons simplement écrit une classe Wrapper (une surcouche) permettant d'avoir plus de contrôle sur celle-ci. De cette manière, nous avons pu très simplement inverser l'ordre de la priorité de la méthode extract qui par défaut extrait le plus grand.

```
class PriorityQueue extends SplPriorityQueue {

   public function compare(mixed $priority1, mixed $priority2) : int {
       if ($priority1 === $priority2)
           return 0;
       return $priority1 < $priority2 ? 1 : -1;
   }

   public function insert(mixed $value, mixed $priority){
       parent::insert($value, $priority);
   }

   public function extract() : mixed {
       return parent::extract();
   }
}
```

Nous avons également tenté afin de rendre la structure de donné encore plus rapide d'ajouter un cache internet, donc une map en utilisant les arrays de php au sein même de la classe :

```
private $cache = [];

$this->cache[$value] = $priority;
isset($this->cache[$value])
```

Cela permettait donc d'avoir une complexité de recherche en O(1) au détriment de l'utilisation de la ram. Malheureusement, cela a causé de légères erreurs de calculs de l'ordre du mètre sur un trajet de plusieurs centaines de kilomètres. Nous avons donc abandonné cette idée en partie puisqu'en déplaçant ce principe dans la classe PlusCourtChemin, les imprécisions ont disparu.

```
CREATE MATERIALIZED VIEW nalixt.vitesses_route AS
SELECT
    nr.gid AS noeud_depart_gid,
    st_x(st_astext(nr.geom)::geometry) AS noeud_depart_long,
    st_y(st_astext(nr.geom)::geometry) AS noeud_depart_lat,
    nr2.gid AS noeud_arrivee_gid,
    st_x(st_astext(nr2.geom)::geometry) AS noeud_arrivee_long,
    st_y(st_astext(nr2.geom)::geometry) AS noeud_arrivee_lat,
    tr.gid AS troncon_gid,
    tr.longueur AS longueur_troncon, tr.geom AS troncon_coord,
    "left"(nc.insee_comm::text, 2) AS num_departement_depart,
    "left"(nc2.insee_comm::text, 2) AS num_departement_arrivee,
    CASE
        WHEN vocation = 'Bretelle' THEN 60
        WHEN vocation = 'Liaison locale' THEN 40
        WHEN vocation = 'Liaison principale' THEN 80
        WHEN vocation = 'Liaison régionale' THEN 70
        WHEN vocation = 'Type autoroutier' THEN 130
    END AS vitesse
FROM nalixt.troncon_route tr

JOIN nalixt.noeud_routier nr ON nr.geom && st_expand(tr.geom, 0.0001::double precision) AND
st_dwithin(tr.geom, nr.geom, 0.0001::double precision)
JOIN nalixt.noeud_routier nr2 ON nr2.geom && st_expand(tr.geom, 0.0001::double precision) AND
st_dwithin(tr.geom, nr2.geom, 0.0001::double precision)
JOIN nalixt.noeud_commune nc ON nr.insee_comm::text = nc.insee_comm::text
JOIN nalixt.noeud_commune nc2 ON nr2.insee_comm::text = nc2.insee_comm::text

WHERE nr2.gid <> nr.gid AND nr.gid < nr2.gid AND tr.sens <> 'Sens inverse';
```

3. Tentatives

Afin de tenter de rendre notre algorithme encore plus rapide, nous avons tenté diverses solutions, qui ont, pour certaines, fonctionnés, et d'autres que nous n'avons pas eu le temps.

1. Json

La première d'entre elles a été, en reprenant notre idée de département, de pré calculer en json le résultat exact retourne par la base de données. Il suffisait ensuite de lire le contenu du fichier et de le parse en json et d'ajouter ce résultat dans notre cache. Cela s'est avéré plus rapide que la base de données de l'IUT mais plus lent en utilisant la base de données en local.

Voici le code en python permettant de générer les différents fichiers json :

```
sqlQuery = '''
           SELECT *
           FROM nalixt.vitesses_route
           WHERE num_departement_depart = %s
           OR
           num_departement_arrivee = %s;
          '''

conn = psycopg2.connect(host=hostname, database=bdd_name, user=login, password=password)

departments = ['01', '02', '03', '04', '05', '06', '07', '08', '09', '10', '11', '12', '13', '14',
'15', '16', '17','18', '19', '2A', '2B', '21', '22', '23', '24', '25', '26', '27', '28', '29', '30',
'31', '32', '33','34', '35', '36', '37', '38', '39', '40', '41', '42', '43', '44', '45', '46', '47',
'48', '49', '50', '51', '52', '53', '54', '55', '56', '57', '58', '59', '60', '61', '62', '63',
'64', '65', '66', '67','68', '69', '70', '71', '72', '73', '74', '75', '76', '77', '78', '79', '80',
'81', '82', '83', '84','85', '86', '87', '88', '89', '90', '91', '92', '93', '94', '95']


def process_department(department):
   cur = conn.cursor()
   cur.execute(sqlQuery, (department, department))
   with open(f'{output_dir}/{department}.json', 'w') as f:
       f.truncate(0)
       content = cur.fetchall()
       nodes = {}
       for node in content:
           noeud_depart_gid = node[0]
           noeud_depart_long = node[1]
           noeud_depart_lat = node[2]
           noeud_arrivee_gid = node[3]
           noeud_arrivee_long = node[4]
           noeud_arrivee_lat = node[5]
           troncon_gid = node[6]
           longueur_troncon = node[7]
           num_departement_depart = node[8]
           num_departement_arrivee = node[9]
           vitesse = node[10]

           if num_departement_depart == department:
               nodes.setdefault(department, {}).setdefault(noeud_depart_gid, []).append({
                   "noeud_gid": noeud_arrivee_gid,
                   "noeud_courant_lat": noeud_depart_lat,
                   "noeud_courant_long": noeud_depart_long,
                   "noeud_coord_lat": noeud_arrivee_lat,
                   "noeud_coord_long": noeud_arrivee_long,
                   "troncon_gid": troncon_gid,
                   "longueur_troncon": longueur_troncon,
```

```
                    "vitesse": vitesse,
                })
            if num_departement_arrivee == department:
                nodes.setdefault(department, {}).setdefault(noeud_arrivee_gid, []).append({
                    "noeud_gid": noeud_depart_gid,
                    "noeud_courant_lat": noeud_arrivee_lat,
                    "noeud_courant_long": noeud_arrivee_long,
                    "noeud_coord_lat": noeud_depart_lat,
                    "noeud_coord_long": noeud_depart_long,
                    "troncon_gid": troncon_gid,
                    "longueur_troncon": longueur_troncon,
                    "vitesse": vitesse,
                })

        f.write(json.dumps(nodes))

    cur.close()
    return department

def generate_departments():
    with concurrent.futures.ThreadPoolExecutor() as executor:
        futures = []
        for dep in departments:
            futures.append(executor.submit(process_department, dep))
        for future in concurrent.futures.as_completed(futures):
            result = future.result()
            print(f'Department loaded: {result}.json')


if __name__ == '__main__':
    generate_departments()
```

2. MultiPriorityQueue

Une autre tentative a été de réaliser une liste de priorités. Cela peut sembler être une idée un peu saugrenue, mais elle repose sur le fait que moins il y a d'éléments dans la PriorityQueue, plus rapide est la méthode extract. Or, la méthode extract est utilisée plusieurs milliers de fois, voire plus d'un million de fois pour la distance la plus éloignée. Nous avons donc tenté de réduire ce temps en utilisant quatre PriorityQueue, puis deux PriorityQueue.

Voici l'implémentation de la MultiPriorityQueue :

```
class MultiPQ extends PriorityQueue {

    private array $priorityQueues = [];
    private int $nbPriorityQueues;
    private int $currentPriorityQueue = 0;

    public function __construct(int $nbPriorityQueues) {
        $this->nbPriorityQueues = $nbPriorityQueues;
        for ($i = 0; $i < $nbPriorityQueues; $i++)
            $this->priorityQueues[$i] = new PriorityQueue();
    }

    public function extract(): mixed {
```

```
        $minPriority = PHP_INT_MAX;
        $minPriorityQueue = null;
        foreach ($this->priorityQueues as $priorityQueue) {
            if ($priorityQueue->valid()) {
                $priority = $priorityQueue->top();
                if ($priority < $minPriority) {
                    $minPriority = $priority;
                    $minPriorityQueue = $priorityQueue;
                }
            }
        }
        if ($minPriorityQueue)return $minPriorityQueue->extract() return null;
    }

    public function insert(mixed $value, mixed $priority) : void {
        $this->priorityQueues[$this->currentPriorityQueue]->insert($value, $priority);
        $this->currentPriorityQueue = ($this->currentPriorityQueue + 1) % $this->nbPriorityQueues;
    }

    public function contains($value): bool {
        foreach ($this->priorityQueues as $priorityQueue)
            if ($priorityQueue->contains($value)) return true;
        return false;
    }

    public function top() : mixed {
        $minPriority = PHP_INT_MAX;
        $minPriorityQueue = null;
        foreach ($this->priorityQueues as $priorityQueue) {
            if ($priorityQueue->valid()) {
                $priority = $priorityQueue->top();
                if ($priority < $minPriority) {
                    $minPriority = $priority;
                    $minPriorityQueue = $priorityQueue;
                }
            }
        }
        if ($minPriorityQueue) return $minPriorityQueue->top();
        return null;
     }

    public function valid() {
        foreach ($this->priorityQueues as $priorityQueue) {
            if ($priorityQueue->valid())return true;
        return false;
    }
}
```

Malheureusement encore, celle-ci s'est avérée tout aussi rapide voir très légèrement plus lente.

3. Tas de fibonacci

L'une des dernières tentatives que nous avons essayées a été de faire un tas de Fibonacci. Ce fut un énorme défi, car c'est une structure de données très complexe, avec une implémentation également très complexe. En contrepartie, elle affiche des complexités hors du commun. En effet, la plupart des opérations sont en O(1) amorti. Après de nombreuses longues heures passées à comprendre et tenter d'implémenter la structure, nous avons enfin pu tester et voir si oui ou non nos efforts avaient porté leurs fruits. À notre grand désarroi, ce fut un échec. Le calcul s'est avéré plus lent qu'avant. Plus lent que notre arbre binaire de recherche, lui-même plus lent que la PriorityQueue. Encore aujourd'hui, nous ne savons toujours pas si c'était dû au fait qu'il soit peu adapté (bien qu'il semble très efficace sur Dijkstra, donc pour A* également), ou bien tout simplement à une mauvaise implémentation faisant perdre la propriété de l'amortissement à l'algorithme. C'est à ce jour la possibilité la plus probable étant donné que le tas de Fibonacci sans le temps amorti a une complexité sur ses opérations du même ordre qu'un arbre binaire de recherche.

4. IDA*

Enfin, nous avons voulu nous essayer à ce que nous pensions être la clé de notre succès (et du bonheur par la même occasion) : l'algorithme IDA* (Iterative Deepening A*). Cet algorithme, contrairement à A* qui effectue une recherche en largeur, effectue une recherche en profondeur. Dans notre cas, IDA* est le meilleur choix puisque nous avons affaire ici à un graphe dense qui se trouve être la carte routière de France. Dans notre cas, il aurait donc été extrêmement efficace, permettant ainsi de visiter bien moins de nœuds et donc potentiellement de se passer du cache. Cela aurait donc accéléré grandement la résolution. Malheureusement, après de multiples tentatives, nous ne sommes pas parvenus à réaliser cet algorithme dans le temps imparti.

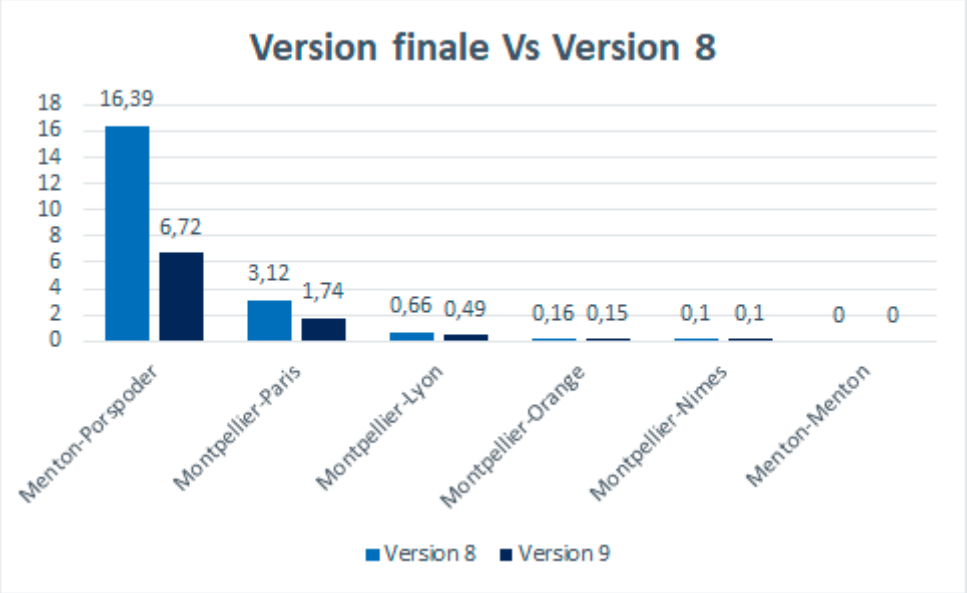
5. A* bidirectionel

Pour finir, la dernière idée à laquelle nous avons pensé est un peu étrange. Elle a été proposée par un des membres de l'équipe et est complètement sortie de notre imagination. Elle consiste à faire deux A* en partant du point d'arrivée et du point de départ. Il suffit ensuite de vérifier si le dernier nœud ajouté dans l’openset* est le même que celui de l’autre openset. Il suffit enfin de relier ces deux openset pour y déterminer le chemin. Malheureusement, nous n’avons pas eu le temps de tenter d’implémenter cette méthode et de voir si elle fonctionnait ou non.

4. Résultats finaux

	Itérations	Distance	Temps
Menton-Menton	1	0km	0s
Montpellier-Nîmes	1939	48.84km	0.10s
Montpellier-Orange	6447	102.05km	0.15s
Montpellier-Lyon	31449	286.67km	0.49s
Montpellier-Paris	152304	687.69km	1.74s
Menton-Porspoder	631595	1301.83km	6.72s
Gravelines-Plouarzel-Urepel-Passa-Lucéram-Lauterbourg-Ombrée d'Anjou-Saint-Victor-Morsang-sur-Seine	1101681	5521.77km	10.51s

Voir annexe numéro 1



Graphique comparant la version finale à la version 8

Avant de conclure sur notre dernière version, nous allons de nouveau détailler l'aspect mémoire de notre site.
Tout d'abord, le changement d'architecture de la vue nous a fait descendre à 320mo, nous faisant descendre de 200mo.
De plus, La suppression du geom de la vue matérialisée nous a fait passer de plus

de 320mo à 153mo. Les coûts mémoire sont donc très faibles et permettent de charger 3 départements en mémoire cache pour le même poids qu'un seul avant cette amélioration.

Node type	Count	Time spent	% of query
Bitmap Heap Scan	1	1.738 ms	76.64%
Bitmap Index Scan	2	0.529 ms	23.33%
Bitmap OR	1	0.002 ms	0.09%

Voici ci-dessus la trace des requêtes pour get tous les nœuds d'un département, nous pouvons constater que le temps d'exécution pour récupérer les nœuds est extrêmement faible, étant donné le très grand nombre de nœuds.

Nous avions précédemment décrit les différents points positifs et problèmes de notre mise en cache. Un des arguments principaux était la comparaison entre un nombre de requêtes très élevé par rapport à un espace mémoire conséquent. Cependant, nous pouvons maintenant nous conforter dans l'idée que notre solution de cache était bel et bien viable, ne coûtant presque plus de ressources mémoire par rapport à notre première solution.
Ainsi, nous pouvons conclure que dans notre cas, une mémoire cache est une bien meilleure solution que 600 000 itérations, voire plusieurs millions si l'on rajoute des étapes préalable.

Finalement, avec ce dernier graphique, nous constatons que la version finale est de loin la meilleure de toutes avec encore une fois une division pas plus de deux du plus gros trajet de France notamment dû aux diverses optimisations citées ci-dessus.
Les choix que nous avons fait pour parvenir jusqu'ici en termes d'optimisation de code et de base de données ont permis de faire un temps jusque-là record qui est sûrement améliorable et que nous aurions adoré réduire avec du temps supplémentaire.

3. Méthodologie et organisation du projet

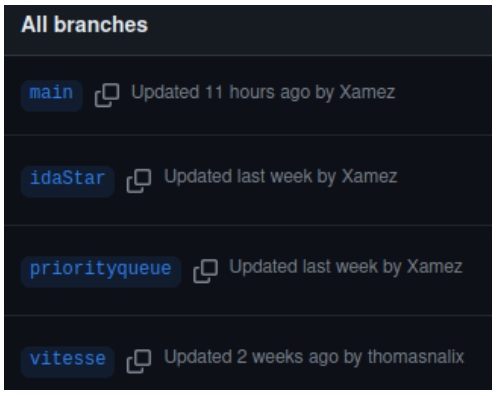
La méthodologie ainsi que l'organisation d'un projet **sont clés** dans la réalisation d'un projet. En effet, c'est notamment grâce à ces deux piliers qu'un projet peut être mené à bien. Nous allons donc voir les dessous de l'organisation derrière Navigator. Pour ce faire, nous nous attarderons tout d'abord sur la gestion de l'équipe, puis nous verrons les méthodes ainsi que les outils exploités. Enfin, nous prendrons du recul sur notre organisation en réalisant un bilan critique.

3.1. Gestion d'équipe

Ayant déjà travaillé ensemble par le passé, l'organisation a été plutôt naturelle. Chacun a eu des responsabilités précises pour mener à bien les différentes tâches du projet. Nous avons donc été majoritairement autonomes, même si nous travaillions souvent ensemble en vocal, par exemple. Cela nous a permis de nous motiver en travaillant en groupe. Durant ces séances de travail, nous pouvions ainsi nous entraider et rester à jour sur l'avancée du travail des autres.

En cas de difficultés rencontrées, ce qui est arrivé assez souvent, chacun a pu proposer ses idées afin d'essayer de surmonter le problème. Il est malheureusement arrivé que personne n'y parvienne. Fort heureusement, la plupart du temps, des recherches poussées ou bien une aide d'un professeur durant les séances de SAE dédiée à cela ont suffit.

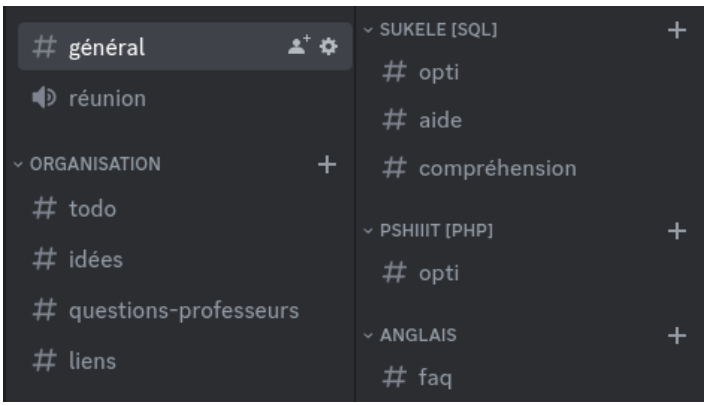
Afin de bien nous organiser au niveau du code, nous avons décidé d'utiliser github, un outil de versioning*. Cet outil nous a permis de développer plusieurs fonctionnalités en parallèle sans nous marcher dessus. Ceci a pu être réalisé grâce au principe de branche.



Liste des branches sur github

3.2. Méthode de développement et outils

Étant donné que nous sommes une équipe de trois, nous avons fait le choix de ne pas utiliser les méthodes agiles. Ce choix a été pris ensemble. Nous avons estimé que réaliser un planning poker, faire un product backlog, maintenir un trello à jour, etc. était bien trop coûteux en temps pour le peu de bénéfice que cette méthode allait nous apporter.
Afin de tout de même s'organiser sans trop d'efforts en temps, notre choix s'est porté vers Discord. Nous avons créé un serveur discord avec chacun des salons dédiés à une tâche en particulier. De cette manière, il était aisé pour nous de nous y retrouver.



Liste des salons sur discord

Nous avons tout de même réalisé un semblant de sprint. En effet, nous avons amélioré de manière itérative notre application en ajoutant et optimisant certaines parties au fur et à mesure. Par exemple, nous avons commencé par optimiser la base de données pendant une semaine, puis la semaine suivante, nous avons effectué des optimisations uniquement côté PHP, et ainsi de suite.

Cette organisation a été pour nous le meilleur choix possible, et nous n'avons pas eu le sentiment d'être oppressés et contraints par les formalités imposées par les méthodes agiles. Cela nous a donc permis d'être bien plus efficaces et de réaliser une quantité de travail réelle sur l'application plus importante.

3.3. Bilan critique

Nous sommes très fiers du travail accompli ainsi que de l'organisation que nous avons mise en place. Évidemment, rien n'est parfait. Si nous avions disposé de plus de temps, nous aurions pu améliorer davantage l'algorithme en explorant encore plus de pistes auxquelles nous avions pensé.

Dans l'ensemble, nous avons achevé tout ce qui était demandé, et même eu le temps d'ajouter un bon nombre de fonctionnalités / extensions :

- Une carte du monde zoomable.
- Une auto complétion au niveau des villes.
- L'affichage des villes sélectionné sur la carte.
- La possibilité de cliquer n'importe ou sur la carte et de réaliser le chemin entre ces deux (ou plusieurs) points.
- La visualisation du chemin le plus court entre deux points.
- La possibilité d'ajouter des étapes sur un trajet (jusqu'à 10).
- Un zoom automatique (un focus) sur le chemin le plus court.
- Un historique des trajets réalisés.
- Pouvoir s'enregistrer et se connecter.
- Une estimation en temps et de la consommation en carburant réel en fonction du trajet et de notre voiture si sélectionné (utilisation d'une api externe).

L'organisation du projet a été rondement menée. En effet, à l'exception de l'algorithme et des fonctionnalités de favoris, nous avons réalisé tout ce que nous avions prévu. Nous n'avons connu aucun désaccord ou mésentente concernant le projet. Nous avons traversé ensemble les moments de joie comme de déception et sommes fiers du produit final.

Notre équipe a néanmoins un défaut majeur : nous voulons accomplir trop de choses dans un temps limité, et les contraintes extérieures ont tendance à rendre ce temps encore plus réduit. Malgré ce défaut, nous avons su composer avec les épreuves et les difficultés rencontrées afin de produire le résultat souhaité.

4. Impacts Environnementaux

La question du climat fait consensus dans notre société. Nous, développeurs, en sommes tout à fait conscients. Nous voulons, et à juste titre, réduire le plus possible les impacts environnementaux liés à l'informatique.

4.1 Analyse du cycle de vie

Afin de prendre conscience du coût d'une telle application, nous allons réaliser une analyse sur le cycle de vie de celle-ci, en prenant en compte différents aspects, et en particulier :

1. Serveur

   Notre application effectue des calculs au niveau du serveur, lesquels étant lourds et coûteux, elle utilise une somme d'énergie conséquente. Il est donc important de prendre en compte les ressources requises par le serveur. En effet, pour notre application, il est nécessaire d'avoir un serveur puissant avec une grande quantité de RAM. En réalité, plutôt que de créer un serveur de zéro, il serait préférable de mettre notre application dans le cloud et d'utiliser des équipements déjà existants, ce qui permettrait de réduire son impact environnemental.

2. Réseaux

   De nos jours, l'utilisation du réseau est omniprésente, et avec l'avènement de l'Internet des objets (IoT)*, la consommation de données a explosé. Il est donc raisonnable de supposer que nos utilisateurs auront une connexion internet suffisante pour l'usage de notre application. Il convient de noter que nos réseaux sont bien développés et que notre application ne requiert pas beaucoup de ressources réseau. En effet, nous avons opté pour une solution qui nécessite peu de requêtes dans la base de données, même si cela entraîne une légère augmentation de la consommation de mémoire. Par conséquent, nous avons décidé de ne pas tenir compte de l'impact des ressources réseau sur notre application.

3. Terminaux utilisateur

   Dans notre analyse, nous ne prêtons pas une attention particulière aux terminaux utilisateurs. Nous considérons que presque toute la population française est équipée de divers matériels informatiques tels qu'un téléphone portable, un ordinateur ou une tablette. De plus, notre solution est déployée sur un site internet et ne dépend pas des obsolescences prématurées des technologies. Il est également important de prendre en compte le fait qu'un utilisateur n'achètera pas ou ne créera pas le besoin

d'achat d'un terminal pour utiliser notre solution. Ainsi, nous estimons que notre application NaviGator a un impact extrêmement faible sur les consommations découlant de la fabrication d'un terminal, l'équivalent d'une goutte d'eau dans un océan.

4.2 Réflexion sur les usages

Le marché des applications de géolocalisation est assez vaste. En effet, de nos jours avec Waze, Google Maps et Apple Maps, il est facile de choisir l'application qui nous convient le mieux. Sachant cela, nous partons du principe que nous avons un avantage concurrentiel car NaviGator est intégré dans une application de covoiturage, à savoir Cocovoit. Cette fonctionnalité, ajoutée au service de covoiturage, permet aux utilisateurs de planifier des trajets en donnant au conducteur le chemin le plus court possible pour un trajet avec plusieurs arrêts (autant que de passagers). Ainsi, nous croyons que cette intégration de NaviGator dans Cocovoit apporte une valeur ajoutée significative à notre solution de géolocalisation.

NaviGator peut être utilisé pour plusieurs usages, tels que :

1. Réduire la consommation de carburant en parcourant moins de kilomètres, car une voiture pollue davantage si elle possède un moteur autre qu'électrique.
2. Planifier des itinéraires avec plusieurs arrêts, ce qui est utile pour les conducteurs qui doivent déposer plusieurs personnes à des endroits différents.
3. Estimer les coûts de trajet en prenant en compte le type de trajet, y compris les autoroutes si elles sont utilisées, ainsi que la consommation estimée.

4.3 Leviers d’action

Il est important d'évaluer les leviers d'action en fonction de leur faisabilité, en commençant par les plus simples à mettre en place. Voici quelques exemples de leviers d'action :

1. Ajouter une option pour éviter les péages et les autoroutes.
2. Utiliser une API* existante qui réalise une action souhaitée et qui est déjà optimisée et écologique, plutôt que de développer notre propre solution.
3. Regrouper les opérations coûteuses pour économiser de l'énergie.
4. Héberger notre site sur une machine virtuelle.

Conclusion

En conclusion, le projet NaviGator avait pour objectif d'optimiser une application web préexistante en mettant en place un algorithme de calcul du chemin le plus court entre deux villes de France. Nous avons débuté par une phase d'analyse minutieuse afin de mettre en place un plan d'action efficace pour atteindre nos objectifs.

A nos yeux, nous avons pleinement rempli les objectifs du projet NaviGator. Nous avons réussi à mettre en place un algorithme de A* efficace pour le calcul du chemin le plus court entre deux points sur le réseau routier Français, tout en améliorant l'expérience utilisateur grâce à l'utilisation de Twig pour la gestion des pages web ainsi que le JavaScript. Nous avons également intégré des API pour récupérer et traiter des données concernant notre potentielle voiture.

En ce qui concerne les perspectives d'évolution du projet, nous pourrions envisager d'étendre la couverture géographique de l'application pour inclure d'autres pays, ou encore d'ajouter de nouvelles fonctionnalités telles que la possibilité de trouver des points d'intérêt (restaurant, hôtel, etc.) sur le chemin ou d'avoir un réel réseau routier avec le trafic en temps réel.

Le projet NaviGator a apporté de nombreux bénéfices tant sur le plan technique que sur le plan humain. Nous avons pu renforcer et approfondir nos compétences en programmation réactive, en PHP et en JavaScript, tout en découvrant de nouvelles façons de programmer grâce à l'utilisation de Twig pour la gestion des pages web. Nous avons également appris à utiliser des API pour récupérer et traiter des données, ainsi qu'à mettre en œuvre des algorithmes et des structures de données complexes.

Parmi les principales difficultés techniques rencontrées, nous pouvons citer la mise en place de l'algorithme de A* pour le calcul du chemin le plus court, ainsi que la gestion des différents types de données (texte, coordonnées géographiques, etc.) nécessaires au bon fonctionnement de l'application.

En termes de travail d'équipe, nous avons amélioré notre communication et notre coordination en répartissant les tâches de manière équilibrée et en travaillant ensemble pour résoudre les problèmes rencontrés.

Dans l'ensemble, le projet NaviGator a été une expérience très enrichissante pour nous sur les plans technique et humain, nous permettant de développer de nouvelles compétences et de renforcer notre travail d'équipe.

Bibliographie

[1] “RapidAPI - The Next Generation API Platform.” [En ligne]. Disponible sur: https://rapidapi.com/.

[2] “Formule de Haversine — Wikipédia.” [En ligne]. Disponible sur: https://fr.wikipedia.org/wiki/Formule_de_haversine.

[3] “Fibonacci Heap Algorithm Operations in C++, Java & Python.” [En ligne]. Disponible sur: https://favtutor.com/blogs/fibonacci-heap-algorithm-operations-cpp-java-python#:~:text=A%20Fibonacci%20Heap%20is%20a,priority%20queue%20in%20Dijkstra’s%20algorithm.

[4] “Pairing heap - Wikipedia.” [En ligne]. Disponible sur: https://en.wikipedia.org/wiki/Pairing_heap.

[5] “Fibo Dijk.” [En ligne]. Disponible sur: https://kbaile03.github.io/projects/fibo_dijk/fibo_dijk.html.

[6] “Binary search tree - Wikipedia.” [En ligne]. Disponible sur: https://en.wikipedia.org/wiki/Binary_search_tree.

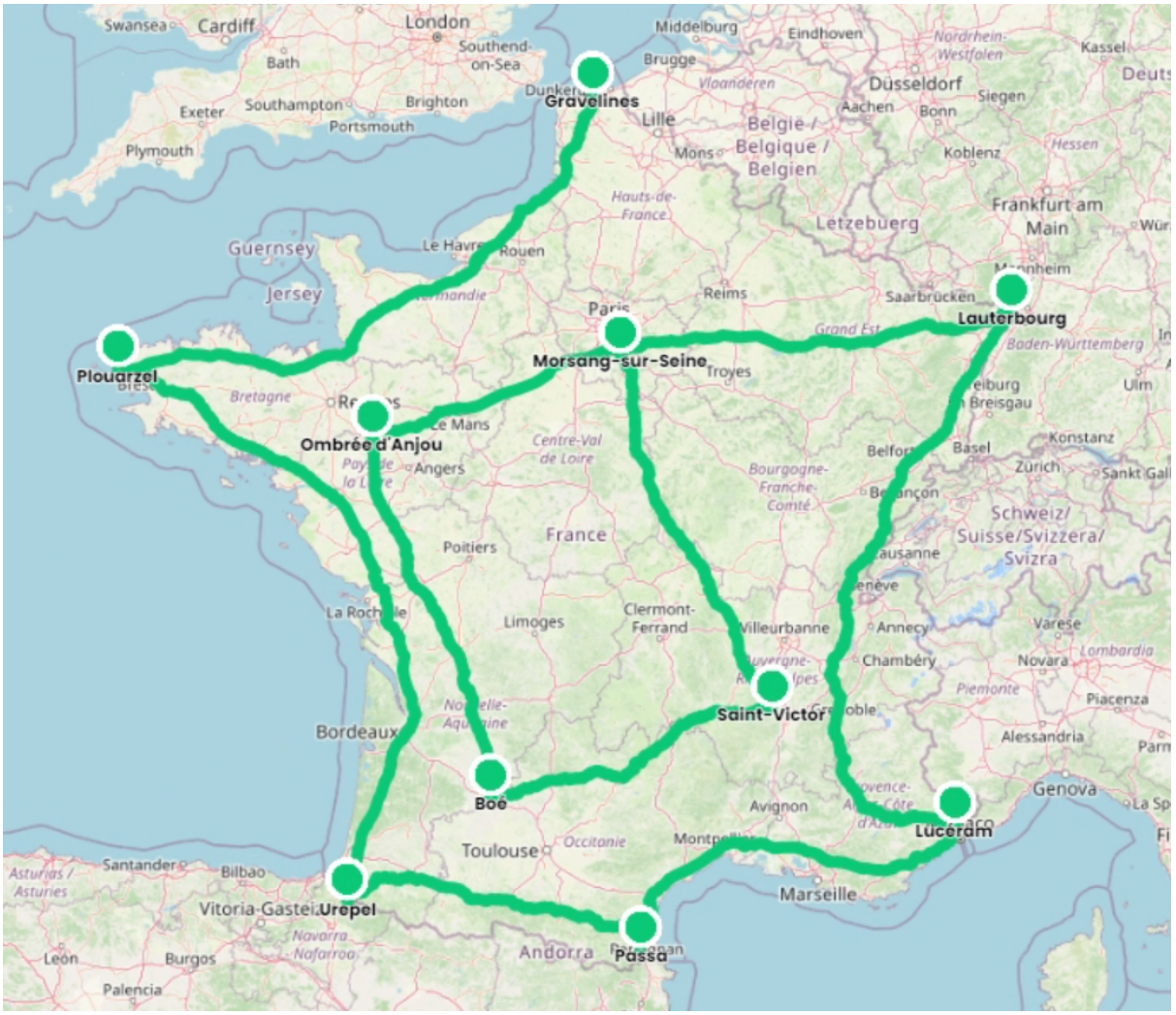
[7] “How to Implement Dijkstra’s Algorithm in Python - YouTube.” [En ligne]. Disponible sur: https://www.youtube.com/watch?v=fRpsjKCfQjE. Chaîne : ThinkX Academy.

[8] “Fibonacci Heap - YouTube.” [En ligne]. Disponible sur: https://www.youtube.com/watch?v=6JxvKfSV9Ns. Auteur: Abdul Bari.

[9] “Pairing Heaps - YouTube.” [En ligne]. Disponible sur: https://www.youtube.com/watch?v=C-kr17luY2g. Auteur: Abdul Bari.

Annexes techniques

1. Un itinéraire très long nécessitant de nombreuses étapes.



Rapport SAE NaviGator

Cazaux - Nalix - Tourniayre

README:

Maxence:
Pourcentage de répartition - **34%**
Durant ce projet, j'ai réalisé plusieurs tâches. La première a été de participer à la phase d'analyse. Une grande partie de mon temps a été consacrée à la recherche et à l'implémentation d'une structure de données adaptée à notre algorithme. J'ai pu en implémenter un bon nombre, dont le binary search tree, le tas de Fibonacci et enfin la PriorityQueue. À côté de cela, je me suis occupé de mettre en place le système de cache des départements. En parallèle, j'ai également essayé une autre approche pour le cache en essayant de me passer de la base de données pour accéder aux voisins en pré calculant l'ensemble des départements dans un fichier JSON à l'aide d'un script Python. Pour finir, j'ai implémenté l'API des voitures en calculant la consommation étant donné le trajet. J'ai également eu le temps de m'occuper de l'autocomplétion des villes et d'aider à droite à gauche les membres de l'équipe. Enfin, j'ai réalisé les tests sur le service du plus court chemin.

Loris:
Pourcentage de répartition - **30%**
Durant ce projet, j'ai été chargé de m'occuper de la mise en place et de la maintenance de la base de données afin de satisfaire des requêtes, de changer la structure de données, de réfléchir à des optimisations, de mettre en place des index, des procédures, des tables, des schémas E/A. J'ai aussi participé au PHP et au JS en m'occupant d'une partie de Twig, des services et de ses interfaces, du refactor du code et de l'entièreté de la programmation réactive. J'ai également participé au webdesign du site en mettant en place des éléments de mon portfolio dans la SAE (parallax, scroll). Enfin, j'ai aussi aidé à la création du discord ainsi qu'à son organisation architecturale.

Thomas:
Pourcentage de répartition - **36%**
Durant le projet, j'ai eu l'occasion de m'occuper de la réalisation de l'algorithme A* ainsi que de son heuristique et des différentes évolutions de celui-ci. J'ai également eu comme mission d'intégrer les premiers jeux de test (mock) et l'architecture vue en complément web telle que le routage, les dépendances, les services, Twig, ... Je me suis également occupé de l'UI et de l'UX du site en passant par le design, les scripts JS, les contrôleurs et la sécurité (cas d'erreurs) de l'utilisateur. Je me suis aussi investi dans l'optimisation côté SQL des requêtes ainsi que des vues. Pour finir, je me suis occupé de toute la partie historique de l'utilisateur. Pour finir, je me suis investi dans beaucoup des différentes étapes d'optimisation.

Cependant, notre pourcentage de répartition ainsi que nos différentes tâches sur le projet ne sont pas représentatifs de la réalité. En effet, dans la majorité des cas, nous avons travaillé ensemble à la réalisation des différents problèmes.
Nous avons pu nous organiser de cette manière car nous avons pris le projet au sérieux et nous avons commencé dès le lancement de celle-ci.