POLYTECH MARSEILLE
Aix*Marseille Université

27/05/2019

Rapport de projet

Fondement et application des bases de données graphes

Présenté par :

Alix COUPOUCHETTY
Manjary RASOLOFONJANAHARY

Tuteur :

A. HABI
M. QUAFAFOU

Remerciements

Nous tenons à remercier notre tuteur de projet A. HABI pour ses conseils tout au long du projet et ses suggestions de recherches de technologies.

Glossaire

- Neo4J : un système de gestion de base de données NOSQL au code source libre basée sur les graphes, développé en Java, par la société suédo-américaine Neo technology. Le produit existe depuis 2000, la 1ère version est sortie en février 2010.

- OSM : OpenStreetMap une technologie de cartographie qui a pour but de constituer une base de données géographiques libre du monde (permettant par exemple de créer des cartes sous licence libre), en utilisant le système GPS et d'autres données libres.

- Java : un langage de programmation orienté objet créé par James Gosling et Patrick Naughton, employés de Sun Microsystems, avec le soutien de Bill Joy (cofondateur de Sun Microsystems en 1982), présenté officiellement le 23 mai 1995 au SunWorld.
La particularité et l'objectif central de Java est que les logiciels écrits dans ce langage doivent être très facilement portables sur plusieurs systèmes d'exploitation.

- Maven : un outil de gestion et d'automatisation de production des projets logiciels Java en général et Java EE en particulier. Il est utilisé pour automatiser l'intégration continue lors d'un développement de logiciel.

- ReactJS : une bibliothèque JavaScript libre développée par Facebook depuis 2013. Le but principal de cette bibliothèque est de faciliter la création d'application web monopage, via la création de composants dépendant d'un état et générant une page (ou portion) HTML à chaque changement d'état.

- MapBox GL : une bibliothèque JavaScript qui utilise WebGL pour rendre des cartes interactives à partir de tuiles vectorielles et de styles Mapbox. Il fait partie de l'écosystème Mapbox GL, qui inclut Mapbox Mobile, un moteur de rendu compatible écrit en C+++ avec des fixations pour les plateformes desktop et mobiles.

Table des matières

Introduction

a) Base de données NoSQL

Depuis les années 70, la base de données relationnelle était l'incontournable référence pour gérer les données d'un système d'information. Toutefois, face aux 3V (Volume, Velocity, Variety), le relationnel peut difficilement lutter contre cette vague de données. Le NoSQL s'est naturellement imposé dans ce contexte en proposant une nouvelle façon de gérer les données, sans reposer sur le paradigme relationnel, d'où le "Not Only SQL".

Cette approche propose de relâcher certaines contraintes lourdes du relationnel pour favoriser la distribution (structure des données, langage d'interrogation ou la cohérence).

Dans un contexte bases de données, il est préférable d'avoir un langage de haut niveau pour interroger les données plutôt que tout exprimer en Map/Reduce. Toutefois, avoir un langage de trop haut niveau comme SQL ne facilite pas la manipulation. Et c'est en ce sens que l'on peut parler de "Not Only SQL", d'autres solutions peuvent être proposées pour résoudre le problème de

distribution. Ainsi, le NoSQL est à la fois une autre manière d'interroger les données, mais aussi de les stocker.

Les besoins de stockage et de manipulation dans le cadre d'une base de données sont variables et dépendent principalement de l'application que vous souhaitez intégrer. Pour cela, différentes familles de bases NoSQL existent : Clé/Valeur, colonnes, documents, graphes. Chacune de ces familles répond à des besoins très spécifiques.

b) Base de données orientées graphes : Quels sont ses intérêts ?

Les bases de données orientées graphe sont des bases de données orientées objet utilisant la théorie des graphes. On représente donc les données sous forme de nœuds et d'arc.
Cela nous permet donc d'utiliser certains algorithmes issus de la théorie des graphes (Comme par exemple la recherche du plus court chemin avec l'algorithme de Dijkstra, A*).

Ces bases de données sont bien plus rapides et performantes que les SGBD Relationnelles.

En effet, l'ajout de nouvelle donnée dans la base n'affecte pas la performance des bases de données graphes en générales comparé aux SGBDR qui commencent à se dégrader au fur et à mesure où la taille des données augmente.

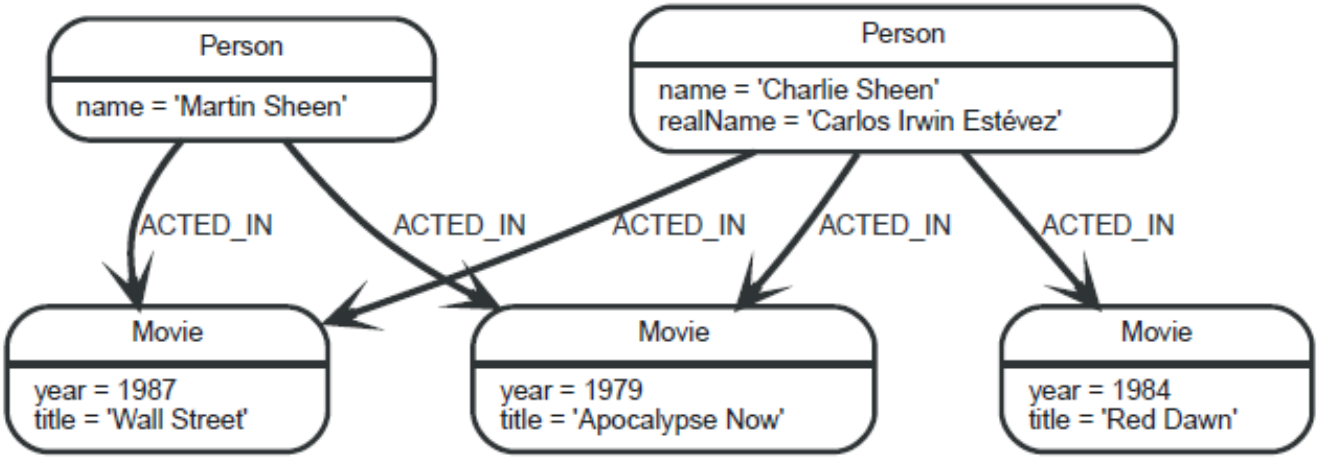
De plus, les nouvelles relations ou les nœuds peuvent être ajoutés sans interférence avec les requêtes et les applications existantes.

Les bases de données Graphe sont utilisé dans plusieurs domaine.
Elles sont par exemples utilisé dans les moteurs de recherches/recommandations, dans les données géospatiales, les services de routage, la détection des fraudes, les réseaux sociaux.

c) Neo4J(1)

Neo4J est une base de données orienté graphe développé en JAVA et qui est d'autant plus Open Source. Chaque objet est donc représenté par des nœuds ou bien des arcs. Les nœuds étant reliés entre eux par les arcs.

Voici un exemple de visualisation des données sur Neo4J :



Neo4J est basé sur le langage Cypher, qui est un langage assez simple et efficace, pour faire les différentes requêtes de parcours à travers le graphe. Les requêtes sont de la forme :

```
MATCH (wallstreet:Movie { title: 'Wall Street' })<-[:ACTED_IN]-(actor)
RETURN actor.name
```

Cette dernière requête nous permet par exemple de chercher le nom des acteurs qui jouent dans le film 'Wall Street'.

d) Réseaux routiers

Comme Neo4J est une SGBD orientée graphe, elle est donc bien adaptée pour gérer les réseaux routiers. Elle est assez simple d'utilisation avec son langage "Cypher" qui nous permet d'avoir une vue assez visuel des données.
De plus, grâce à sa structure en graphe, on peut facilement utiliser les algorithmes de la théorie des graphes notemment la recherche du plus court chemin. Notemment l'algorithme de Dijkstra ou l'algorithme de A*.

Neo4J est donc bien adapter à notre application qui fait les recherches du plus court chemin entre différents points d'intérêts dans une ville.

Présentation de l'application et objectifs

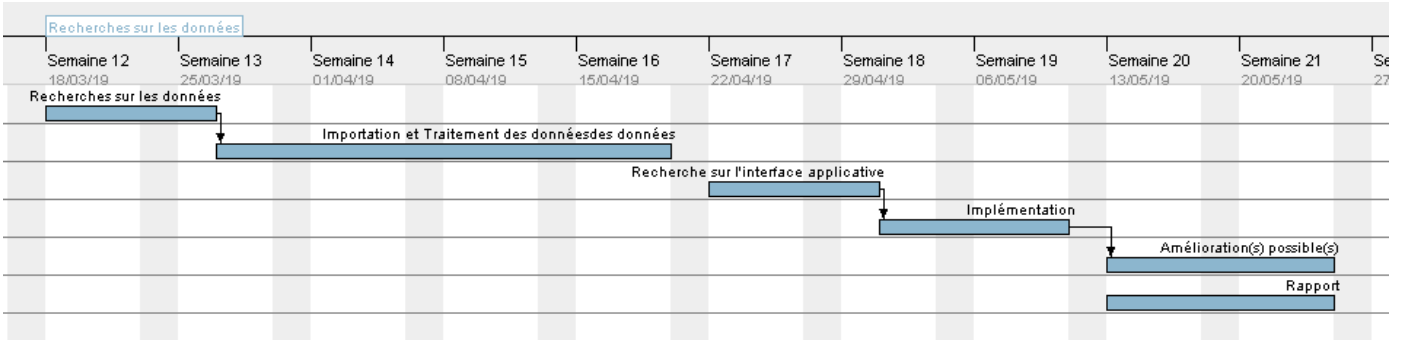
Dans le cadre de notre deuxième année en école d'ingénieurs nous avons eu pour projet la réalisation d'une application dédiée à la recherche de plus court chemin dans un réseau routier.

Les besoins relatifs au traitement des données ont évolué à un rythme soutenu ces dernières années. Il faut en effet être en mesure de répondre aux contraintes toujours plus importantes de sites web aux millions d'utilisateurs (réseaux sociaux, plates-formes de navigation et de diffusion de contenus, etc.). Ces enjeux en termes de clustering et de scalabilité ont fait naître la plus dynamique des familles de moteurs de données : le NoSQL.

Parmi ces moteurs de données, ceux orientés « graphes », e.g. Neo4j, connaissent un succès fulgurant du fait de leur souplesse et de leur évolutivité extrême. Ne reposant sur aucun schéma, un graphe peut accepter de nouveaux flux/jeux de données sans imposer d'interventions lourdes.

Déroulement du projet

Afin d'avoir un appui temporel pour notre projet, nous nous sommes appuyés sur ce planning prévisionnel afin de déterminer l'avancement de notre projet au cours du temps durant ces deux mois.



1. Planification

Le planning se découpait en 4 phases :

- Recherches sur les types données qu'on pouvait utiliser, savoir lesquels étaient le plus approprié et le plus facile à manipuler,
- L'importation et traitements des données avec Neo4J : rechercher un moyen de convertir nos données en données lisibles par Neo4J, les importer et traiter ces données afin de pouvoir les utiliser sur une map,
- Recherches sur l'interface web et technologies utiles pour notre application,
- L'implémentation de ce dernier,
- Détermination des améliorations possibles,
- Rédaction du rapport.

Cependant, étant donné que nous nous sommes trop attardés sur l'importation des données. Nous avions pris du retard sur l'ensemble du projet.

Technologies utilisées

Afin de répondre aux objectifs demandés, nous avons dû utiliser des technologies liées à notre formation :

- Une base données du type graphe était le plus approprié, donc nous avons pris Neo4J, étant donné que nous avons eu une formation sur celle-ci.

- Les données sont de type OSM.

- La conversion des données OSM en données lisibles par Neo4J et l'importation ont été faites grâce à une application Java et l'outil « maven ».

- L'application web a été faite avec Mapbox GL pour avoir une carte intéractive et le langage ReactJS.

Conversion et Importation des données

Il existe un modèle de données OSM et un importateur ave le plugin « Neo4J Spatial » qui supporte Neo4j 1.x, 2.x et 3.x. Cependant cet outil a quelques problèmes :

Il n'a pas d'échelle. L'utilisation d'un indice de lucène pour le mappage OSM-id à Neo4j node-id pour créer des relations met un plafond sur la taille effective des données pouvant être chargée. Généralement, les utilisateurs ne chargent que des villes ou tout au plus de très petits pays. Les grands pays sont très difficiles à charger et la planète entière est complètement hors de portée.

Il n'a aucun rapport avec le nouvel indice spatial intégré dans Neo4j 3.4. Il a été conçu pour fonctionner exclusivement avec l'interface RTree index et GeometryEncoder GIS du projet Neo4j Spatial.

Il a été conçu pour refléter le modèle graphique réel de l'OSM qui supporte un seul graphique éditable de toutes les données. Cela en fait un modèle très complexe qui ne convient pas à certains cas d'utilisation comme le routage.

Il y a un intérêt à obtenir un bon modèle de données OSM pour de nombreux cas d'utilisation (bac à sable, utilisateurs OSM existants de Neo4j Spatial, etc.) et cela nous amène à vouloir créer un nouveau modèle et importateur avec les caractéristiques suivantes :

- Rapide et évolutif (en utilisant l'importateur parallèle de lots introduit dans Neo4j 2.x)
- Peut être utilisé sans 'Neo4j Spatial',
- Peut remplacer l'ancien OSMImporter dans Neo4j Spatial,
- Prise en charge d'un plus large éventail de cas d'utilisation, y compris le routage.

Par conséquent, nous avons télécharger une application Java[(1)] qui justement convertit et importe nos données OSM[(2)] nos données dans Neo4J.

Comme c'est un projet Maven, il faut compiler l'application Java grâce à la commande :

mvn clean install

La compilation produit quatre fichiers *jar*. Le fichier *procedures.jar* est à déplacer dans le dossier *plugins* de Neo4J afin d'effectuer le traitement des données qu'on verra plus tard.

Ensuite nous devons obtenir toutes les dépendances nécessaires pour l'application :
mvn dependency:copy-dependencies

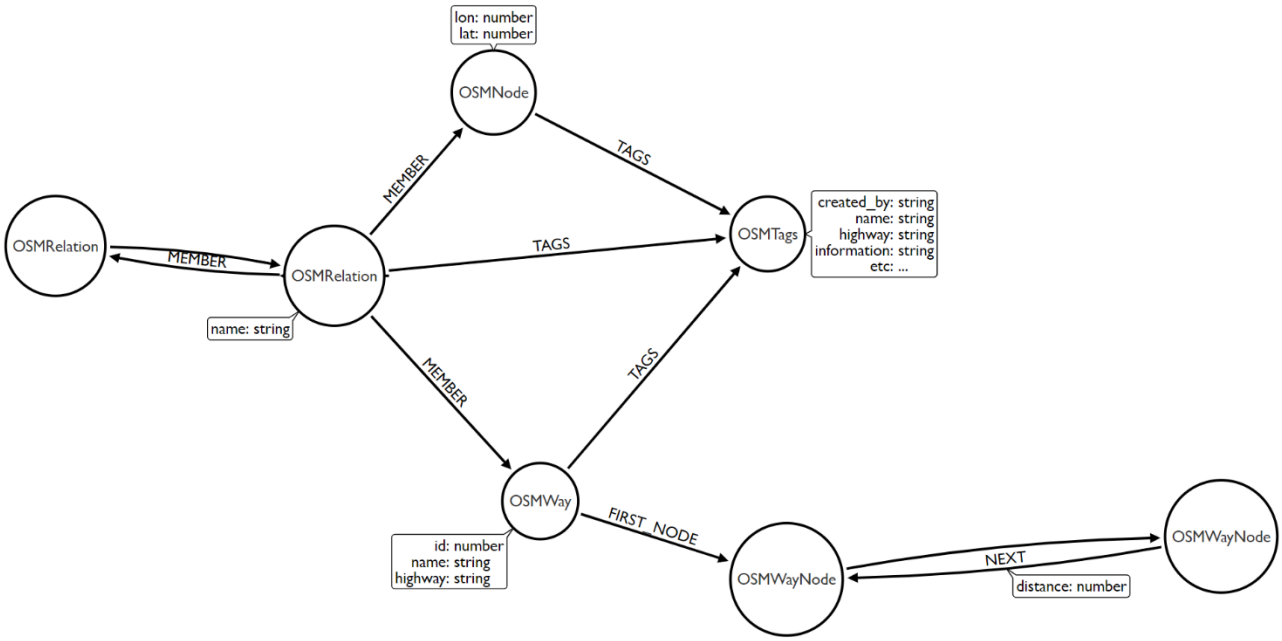
Nous pouvons ainsi lancer l'application qui se trouve dans le dossier « *target/* » pour convertir et importer les données :

java -Xms1280m -Xmx1280m -cp target\osm-0.2.2-neo4j-3.5.1.jar;target\dependency org.neo4j.gis.osm.OSMImportTool --skip-duplicate-nodes --delete --into target\databases\marseille samples\Marseille.osm*

La commande précédente est utilisée sur sous Windows. Pour effectuer cette étape avec un ordinateur sous Linux, la commande est la suivante :
java -Xms1280m -Xmx1280m -cp "target/osm-0.2.2-neo4j-3.5.1.jar:target/dependency/" org.neo4j.gis.osm.OSMImportTool --skip-duplicate-nodes --delete --into target/databases/marseille samples/Marseille.osm.*

Ceci convertit et importe le fichier « Marseille.osm » en base de données « target/databases/marseille » lisible par Neo4J.

Finalement, nous avons des données sous ce format qui définit les différents types nœuds que nous avons :



2. Schéma de l'organisation de nos données

Un OSMRelation possède plusieurs membres qui peuvent être des OSMNode, OSMWay ou OSMRelation.

Chaque OSMNode représente un point géographique défini par son altitude et longitude.

Un OSMWay représente un chemin. La relation FIRST_NODE indiquera donc le premier point de tout le chemin.

Un OSMWayNode est en fait un OSMNode mais qui appartient à un chemin. En effet, un chemin est juste un ensemble de OSMWayNode.

Les OSMTags représentent les différentes propriétés supplémentaires que peuvent avoir les OSMRelation, OSMNode ou OSMWay.

Traitement des données

Les données telles qu'elles sont n'était pas utilisables pour effectuer le plus court chemin. Il fallait traiter ces données pour créer des intersections, des routes et des points d'intérêts.

Pour cela, nous avons effectué quelques requêtes sur la base de données avec Neo4J.

Tout d'abord, nous avions besoin de la notion de distance sur nos données :

MATCH (awn:OSMWayNode)-[r:NEXT]-(bwn:OSMWayNode)
WHERE NOT exists(r.distance)
WITH awn, bwn, r LIMIT 10000

MATCH (awn)-[:NODE]->(a:OSMNode), (bwn)-[:NODE]->(b:OSMNode)
SET r.distance= distance(a.location,b.location)
RETURN COUNT(*)

Pour que cette notion soit optimale, cette requête doit être lancée plusieurs fois. Une approche un peu plus automatisé serait de lancer la requête suivante qui donne le même résultat que de lancer plusieurs fois la précédente :

CALL apoc.periodic.iterate('MATCH (awn:OSMWayNode)-[r:NEXT]-(bwn:OSMWayNode) WHERE NOT exists(r.distance) RETURN awn, bwn, r',

'MATCH (awn)-[:NODE]->(a:OSMNode), (bwn)-[:NODE]->(b:OSMNode)
SET r.distance= distance(a.location,b.location)', {batchSize:10000,parallel:false});

Cependant, pour appeler la fonction iapoc.periodic.iterat(), il fallait ajouter le plugin *apoc*[(3)] dans Neo4J.

Nous pouvions maintenant créer les intersections :

MATCH (n:OSMNode)
WHERE size((n)<-[:NODE]-(:OSMWayNode)-[:NEXT]-(:OSMWayNode)) > 2
AND NOT (n:Intersection)

*MATCH (n)<-[:NODE]-(wn:OSMWayNode), (wn)<-[:NEXT*0..100]-(wx), (wx)<-[:FIRST_NODE]-(w:OSMWay)-[:TAGS]->(wt:OSMTags) WHERE exists(wt.highway)*
SET n:Intersection
RETURN count(*);

Et ainsi la relation *ROUTE* entre les nœuds *Intersection* :

```
MATCH (x:Intersection)
CALL spatial.osm.routeIntersection(x,false,false,false)
YIELD fromNode, toNode, fromRel, toRel, distance, length, count
WITH fromNode, toNode, fromRel, toRel, distance, length, count
MERGE (fromNode)-[r:ROUTE {fromRel:id(fromRel),toRel:id(toRel)}]->(toNode)
ON CREATE SET r.distance = distance, r.length = length, r.count = count
RETURN count(*);
```

Enfin, pour avoir des points pour définir le plus court chemin, nous avons défini des points d'intérêt qui nous semblaient utiles :

```
UNWIND ["restaurant","fast_food","cafe","bar","pub","ice_cream","cinema"] AS amenity
MATCH (x:OSMNode)-[:TAGS]->(t:OSMTags)
  WHERE t.amenity = amenity AND NOT (x)-[:ROUTE]->()
WITH x, x.location as poi LIMIT 100

MATCH (n:OSMNode)
  WHERE distance(poi, n.location) < 100
WITH x, n

MATCH (n)<-[:NODE]-(wn:OSMWayNode), (wn)<-[:NEXT*0..10]-(wx),
    (wx)<-[:FIRST_NODE]-(w:OSMWay)-[:TAGS]->(wt:OSMTags)
WITH x, w, wt
  WHERE exists(wt.highway)
WITH x, collect(w) as ways
  CALL spatial.osm.routePointOfInterest(x,ways) YIELD node
  SET x:PointOfInterest

RETURN count(node);
```

Il nous resta plus que de lier ces points d'intérêt :

```
MATCH (x:Routable:OSMNode)
WHERE NOT (x)-[:ROUTE]->(:Intersection) WITH x LIMIT 100
CALL spatial.osm.routeIntersection(x,true,false,false)
YIELD fromNode, toNode, fromRel, toRel, distance, length, count
WITH fromNode, toNode, fromRel, toRel, distance, length, count
```

Interface applicative

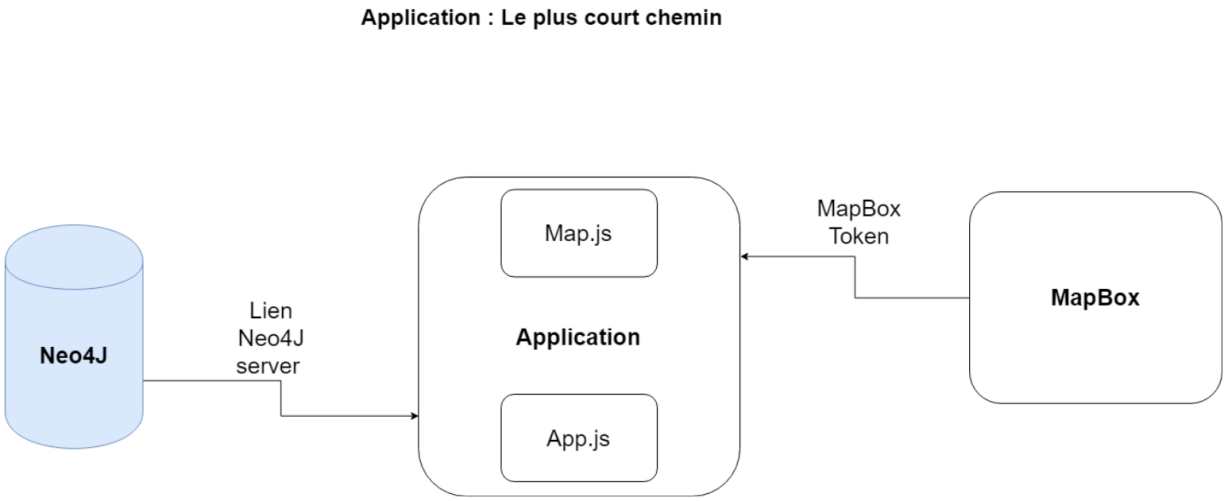
On a utilisé Create-React-App pour faciliter le développement de notre application web[4] et générer ainsi automatiquement un squelette de notre application. Afin de l'utiliser, on a eu besoin d'installer au préalable *Node et Npm*. Ainsi, pour vérifier si on les a déjà installés sur notre machine, on peut exécuter les commandes suivantes :

node --version

npm --version

Pour la partie affichage des données, on utilise la librairie Javascript "MapBox GL JS". Cette librairie nous permet entre autres d'afficher une carte du monde sur notre application web et de gérer tout ce qui concerne l'affichage de données depuis notre base de données graphe sur notre carte.

Enfin, on utilise le module "neo4j-driver" pour pouvoir exécuter des requêtes neo4j dans notre application web, notamment la recherche du plus court chemin entre deux points d'intérêts.



Pour que notre application communique avec Neo4J et utilise une map de MapBox GL, on utilise un fichier conf « *.env* » afin de déterminer la base de données, les identifiants si il y en faut pour se connecter à Neo4J et le token lié à un compte mapbox :

REACT_APP_NEO4J_URI=bolt://localhost:7687
REACT_APP_NEO4J_USER=neo4j
REACT_APP_NEO4J_PASSWORD=12Soleil
REACT_APP_MAPBOX_TOKEN=pk.eyJ1IjoiYS1jciIsImEiOiJjanZkajAxZ2QwNXN3NDZxdXU2N3Z5ZXFiIn0.IGrkegqO8BndW_z5NdQxDQ

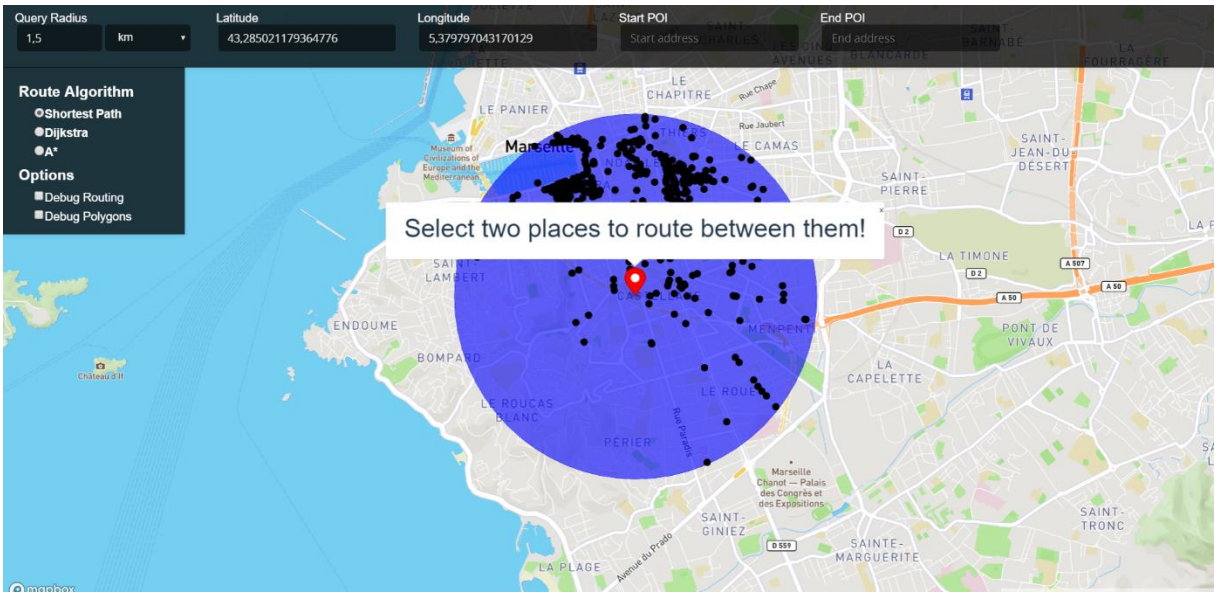
```
fetchBusinesses = () => {
  const { mapCenter } = this.state;
  const session = this.driver.session();

  let query;

  if (this.state.regionId) {
    query = `MATCH (r:OSMRelation) USING INDEX r:OSMRelation(relation_osm_id)
      WHERE r.relation_osm_id=$regionId AND exists(r.polygon)
    WITH r.polygon as polygon
    MATCH (p:PointOfInterest)
      WHERE distance(p.location, point({latitude: $lat, longitude:$lon})) < ($radius * 1000)
      AND amanzi.withinPolygon(p.location,polygon)
    RETURN p
    `;
  } else {
    query = `MATCH (p:PointOfInterest)
      WHERE distance(p.location, point({latitude: $lat, longitude:$lon})) < ($radius * 1000)
    RETURN p`;
  }
  session
    .run(query, {
      lat: mapCenter.latitude,
      lon: mapCenter.longitude,
      radius: mapCenter.radius,
      regionId: this.state.regionId
    })
    .then(result => {
      console.log(result);
      const pois = result.records.map(r => r.get("p"));
      this.setState({ pois });
      session.close();
    })
    .catch(e => {
      // TODO: handle errors.
      console.log(e);
      session.close();
    });
};

this.state = {
  focusedInput,
  startDate: moment("2014-01-01"),
  endDate: moment("2018-01-01"),
  businesses: [],
  starsData: [],
  reviews: [{ day: "2018-01-01", value: 10 }],
  categoryData: [],
  selectedBusiness: false,
  mapCenter: {
    latitude: 43.28503679952334,
    longitude: 5.380199374551836,
    radius: 1.5,
    zoom: 16
  },
```

Ici, nous initialisons notre application en définissant le centre de notre zone afin d'être directement sur la ville de Marseille.

Pour démarrer notre application, on utilise la commande suivante depuis un terminal en se plaçant sur le dossier racine de notre répertoire :

npm start

Une page internet s'ouvre automatiquement et affiche ceci :



Sur l'image précédente, les points d'intérêt sont représentés en points noirs.
Nous avons modifié cela, et défini des icons en fonctions des points d'intérêt.

On ajoute un attribut « *amenity* » pour les points d'intérêt :

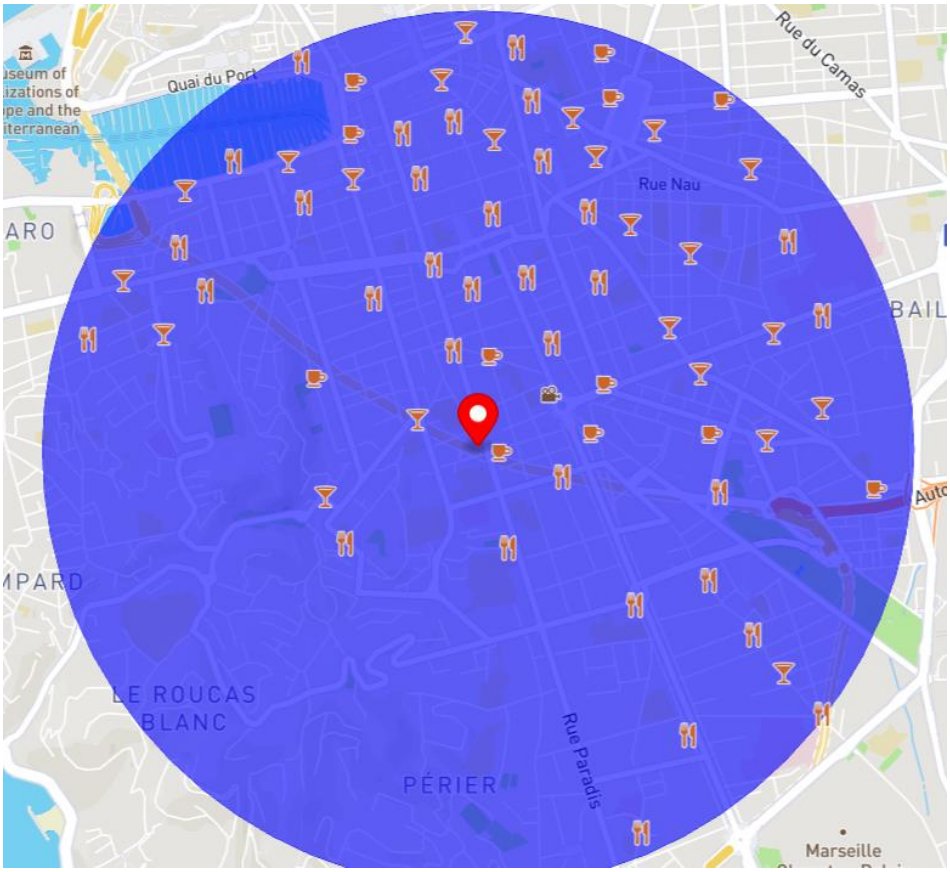
MATCH (n:PointOfInterest)-[:TAGS]->(o:OSMTags)
WHERE NOT EXISTS(n.amenity)
SET n.amenity=o.amenity
RETURN n;

```
geoJSONForPOIs = pois => {
  return pois.map(poi => {
    const p = poi.properties;
    return {
      type: "Feature",
      geometry: {
        type: "Point",
        coordinates: [p.location.x, p.location.y]
      },
      properties: {
        title: "",
        id: p.node_osm_id.toString(),
        name: p.name,
        amenity:p.amenity,
        icon: "monument",
        "marker-color": "#fc4353"
      }
    };
  });
```

Dans *Map.js*, pour la source geoJSON des *pois*, on ajoute une attribut *amenity*.

```
this.map.addLayer({
  "id": "points",
  "type": "symbol",
  source: "geojson",
  "layout": {
      "icon-image": "{amenity}-15"
  }
});
```

Ainsi, la couche Layer de map liée au points d'intèrêt peut utiliser les icons dela librairie Mapbox GL.



Améliorations possibles

En dehors du cadre de notre projet, notre application pourrait être améliorée en ajoutant de nouvelles fonctionnalités :

- Le fait de sélectionner un point aléatoire sur la map,
- L'ajout d'un algorithme par l'utilisateur lui-même,
- Une map plus conséquente et sélection d'une ville choisie par l'utilisateur.
- Avoir une base de données « améliorée », par exemple lorsqu'un point d'intérêt est proche d'un tunnel, ne pas créer la route entre ce point et le tunnel.

Conclusion

À l'aide de ce projet nous avons pu comprendre et expérimenter les différentes étapes de la conception d'une application dédiée au plus court chemin.

Cela nous a permis également d'avoir une expérience un peu plus approfondie sur l'utilisation de Neo4J et des base de données de type graphes, ainsi que sur la librairie ReactJS.

Code source / Annexes et Sources

Le code source se trouve sur ce GitHub :

- https://github.com/Acr5-6/Le_plus_court_chemin

Nos recherches se sont basées sur ces sources :

(2) Les données utilisées :

- https://download.bbbike.org/osm/bbbike/Marseille/
- http://download.openstreetmap.fr/extracts/europe/france/provence_alpes_cote_d_azur/

(1) Conversion et Importation des données :

- https://www.youtube.com/watch?time_continue=304&v=ncKreSy1oYk&fbclid=IwAR3nDnFHIeKM_6ZtpApwnKfrfNK0RFEjovA3-zpxj6s6U-TThkg9jrsSXp8
- https://github.com/neo4j-contrib/osm?fbclid=IwAR0AQ2YP72VLPZeAsMDN7mlYr8__NwUdhFqL6zaJKiLcKUYfBUQDFj04b1Y

(3) Documentation et plugins de Neo4J utilisés :

- https://neo4j.com/docs/?_ga=2.145473012.1737707066.1558686366-707071412.1558438773
- https://github.com/neo4j-contrib/neo4j-graph-algorithms/releases/3.5.4.0?fbclid=IwAR2WoLxQH5rr85uR51BupZEf9_qVJAcf1H3gYfUlQJyZriEGg_wvcRZm8Og
- https://github.com/neo4j-contrib/neo4j-apoc-procedures/releases/3.5.0.3?fbclid=IwAR1732b7ib4dSFweoF92maiyjCMy3aAVy2yH2JM7OFsE-1Nb_055dRy2Ais

(4) Interface Web :

- https://docs.mapbox.com/mapbox-gl-js/api/?fbclid=IwAR26qD8Qblw6meYF2VKAkikpvWcGSpFWh-05eDPSpof22wGqH2r5ANFJUag
- https://facebook.github.io/create-react-app/docs/getting-started
- https://github.com/johnymontana/osm-routing-app?fbclid=IwAR3kkIoebYq-2_yJlQU1KKSi6TjT8oPIMjsbfUZKo-S_Hh2svzk22VWKm5s