1. Processus de conceptions de l'outils de calcul de l'itinéraire le plus court

Je suis partie sur l'implémentation d'une class <Graph> qui contiendra toutes les méthodes associées à la détermination du chemin le plus court à partir d'une liste de waypoints.
Afin d'accomplir mon objectif, le besoin s'est imposé d'utiliser un algorithme de détermination du chemin le plus court (Dijkstra, A*, Bellman Ford …). Pour des raisons de simplicité, j'ai décidé avec Dijkstra.
La nécessité d'implémenter les méthodes comme:

```
std::vector<int> getShortestPath(const Waypoint& start, const Waypoint& end);
```

Méthode qui permet où Dijkstra sera proprement implémenté.

```
  void visualizePath(const std::vector<int>& path);
```

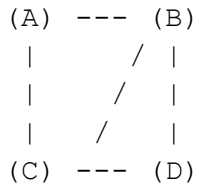
Méthode qui permet de visualiser le chemin le plus court.

```
      double getDistance();
```

Méthode où la distance totale que représente le chemin le plus court est calculée.

```
void adjacencyListToEdgeListFile(const std::unordered_map<std::string,
std::vector<std::pair<std::string, double>>>& adjacencyList, const std::string&
filename);
```

Méthode qui permet de représenter l'adjacency list en edge list et qui sauvegarde le résultat dans un fichier .txt. Adjacency list et l'edgeList sont des manières de représenter les graphes. La représentation d'un graphe sous forme d'adjacency list est optimale pour les algorithmes de calcul du chemin le plus court. Tandis que la représentation en edge list est optimale pour les algorithmes de visualisations de graphes.



GRAPH

```
adjacency_list = {
   'A': ['B', 'C'],
   'B': ['A', 'C', 'D'],
   'C': ['A', 'B', 'D'],
      'D': ['B', 'C']
    }
```

ADJACENCY LIST

```
edge_list = [
   ('A', 'B'),
   ('A', 'C'),
```

```
('B', 'C'),
('B', 'D'),
('C', 'D')
 ]
```

EDGE LIST

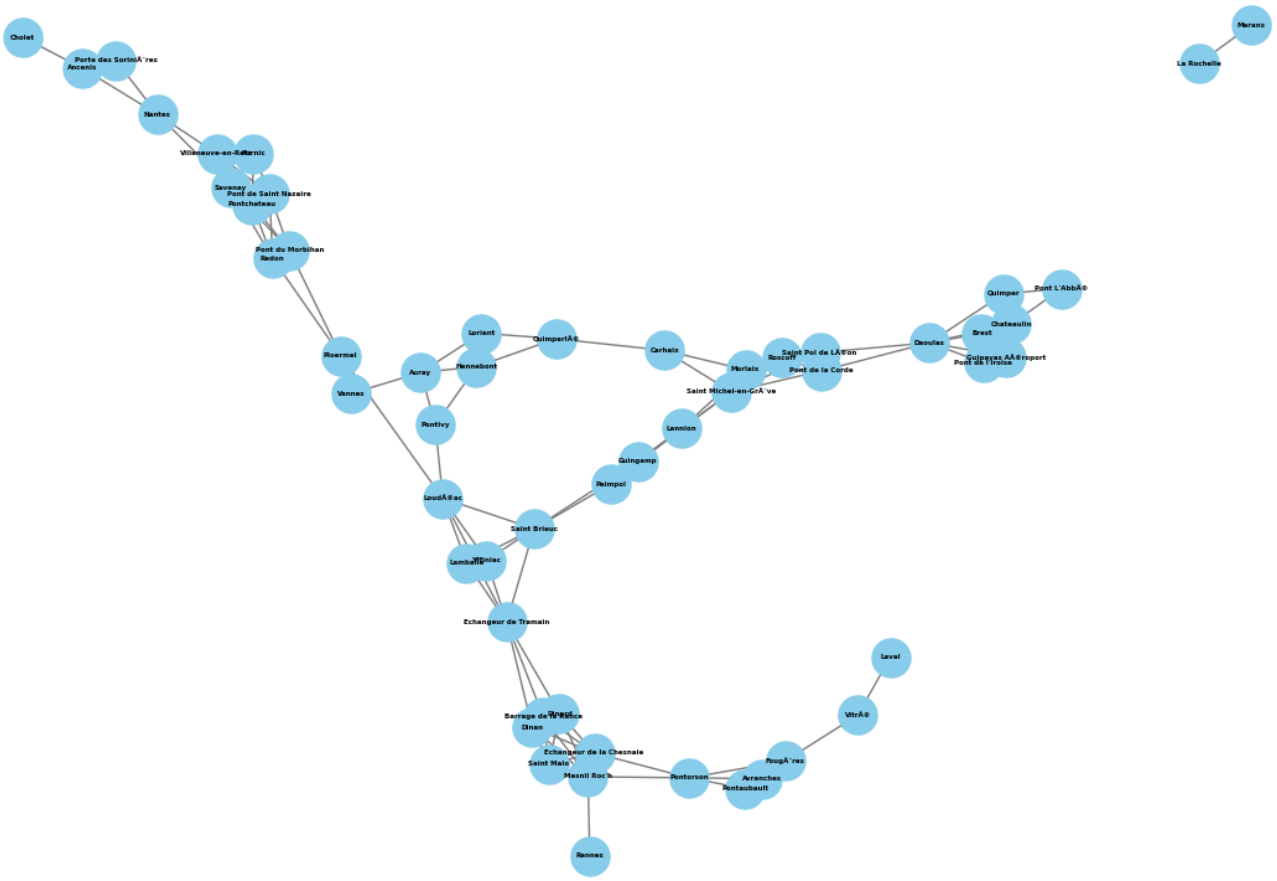
```
void createAdjacencyList();
```

Méthode qui permet de créer l'adjacency list.

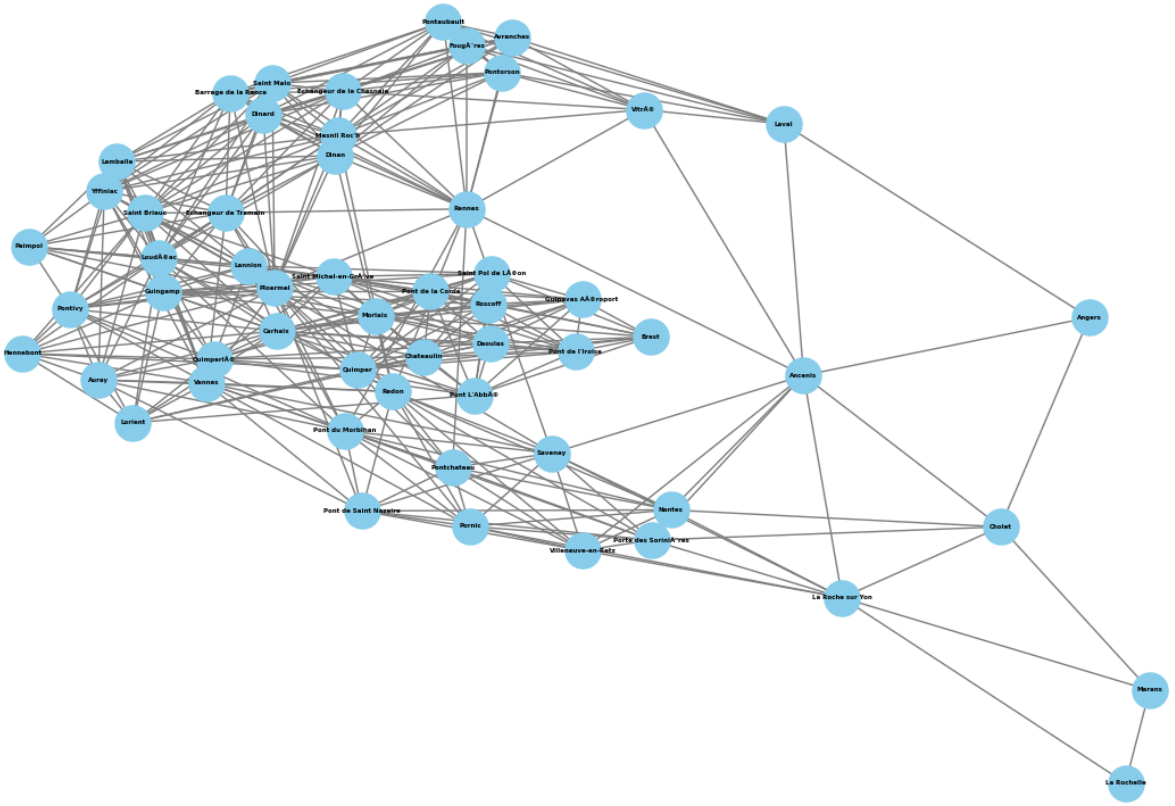
2. **Comment fonctionne l'algorithme Dijkstra ?**
   Graph Data Structure 4. Dijkstra's Shortest Path Algorithm

3. **Explication de ma méthodologie de raisonnement, visualisation, error, correction**

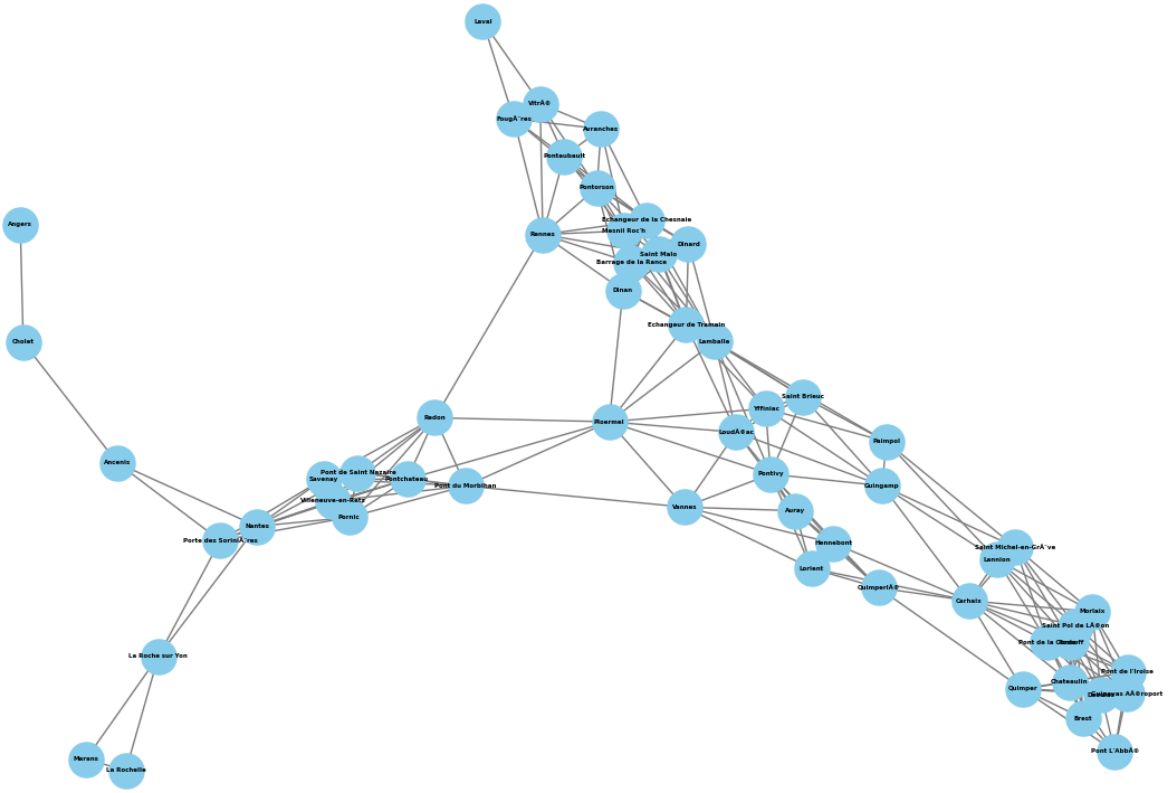
Le principale challenge rencontré lors de l'implémentation des méthodes de la classe <Graph> la création d'un adjacency list. j'ai eu du mal à déterminer le critère(distance, temps, coût...) qui permet d'identifier le voisin d'un point donné. Au regard des données (waypoints) que j'avais, je me suis dit qu'un waypoint B est voisin d'un waypoint A si la distance qui les sépare est inférieure à un certain seuil.
Après plusieurs tests sur diverses valeurs de seuil, j'ai un premier temps trouvé satisfaction avec une valeur de seuil égale à 70.



Graph seuil 50

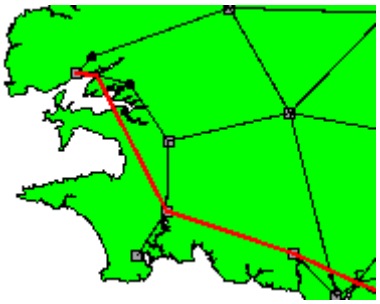


Graph seuil 100



Graph seuil 70

Vient maintenant le moment fatidique de tester notre algorithme d'identification du chemin le plus court.



On pourra remarquer que le chemin représentant le chemin le plus court ne passe pas par les routes prédéfinies. Le doute commence à prendre place sur la validité de la construction du graphe (adjacency list), des questions sont soulevées. Pourquoi la voie ne passe pas par les deux points où elle est supposée passer ? Est-ce parce que ces points ne sont pas considérés comme des voisins ? Si tel est le cas, celà est dû à quoi ? Est-ce la valeur du seuil qui cause problème ? Si oui, pourquoi ?

Afin d'éclaircir ma compréhension, j'ai prié une valeur de seuil inférieur à 70 (50). En essayant de visualiser le résultat, j'ai remarqué que le point de destination n'était pas atteint, celà était dû au fait que la distance des points se trouvant sur le chemin était bien plus grande que celui du seuil et par conséquent, ces points n'étaient comptés comme voisins. J'ai naturellement testé la construction de l'adjacent list avec seuil supérieur à 70 (100). Cette

fois-ci, le résultat donnait un chemin direct, c'est-à-dire sans passer par des points intermédiaires.

Mais comment faire en sorte que mon algorithme soit correct et efficace sur n'importe quel calcul de trajet le plus court ? Je me suis lancé dans la recherche de la valeur seuil optimale en faisant appelle à des fonctions linéaire et logarithmique qui augmente/diminue la valeur du seuil en fonction de la distance.

```
double Graph::calculateLinearPrecision(double distance) {
    // Paramètres à ajuster
    double minPrecision = 10.0; // Précision minimale (pour les distances courtes)
    double maxPrecision = 100.0; // Précision maximale (pour les distances longues)
    double maxDistance = 300.0; // Distance maximale pour le scaling linéaire

    // Calcul linéaire de la précision
    return minPrecision + (maxPrecision - minPrecision) * (distance / maxDistance);
}
```

Fonction linéaire

```
double Graph::calculateLogarithmicPrecision(double distance) {
    // Paramètres à ajuster
    double minPrecision = 10.0;
    double maxPrecision = 100.0;
    double logBase = 1.1; // Base du logarithme (ajuster pour contrôler la courbe)

    // Calcul logarithmique de la précision
    return minPrecision + (maxPrecision - minPrecision) * log(distance) / log(logBase);
}
```

Fonction Logarithmique

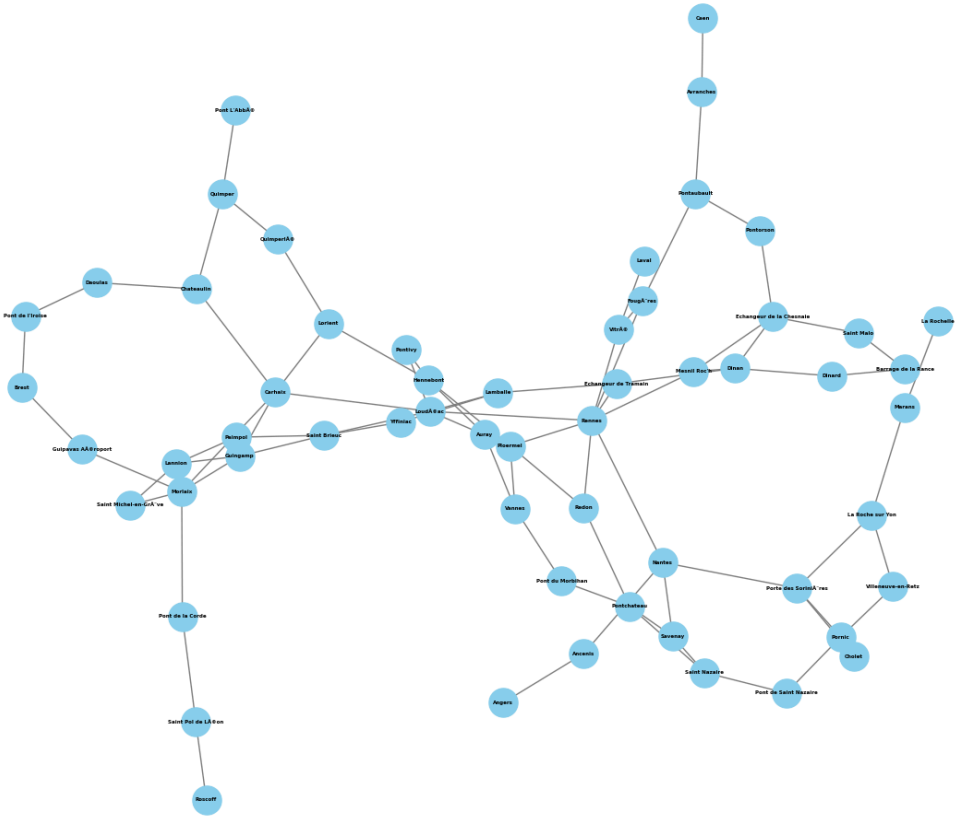
```
double calculateAngle(const Waypoint& a, const Waypoint& b, const Waypoint& c) {
    double ab_x = b.getLongitude() - a.getLongitude();
    double ab_y = b.getLatitude() - a.getLatitude();
    double bc_x = c.getLongitude() - b.getLongitude();
    double bc_y = c.getLatitude() - b.getLatitude();

    double dot = ab_x * bc_x + ab_y * bc_y;
    double cross = ab_x * bc_y - ab_y * bc_x;
    return atan2(cross, dot);
```

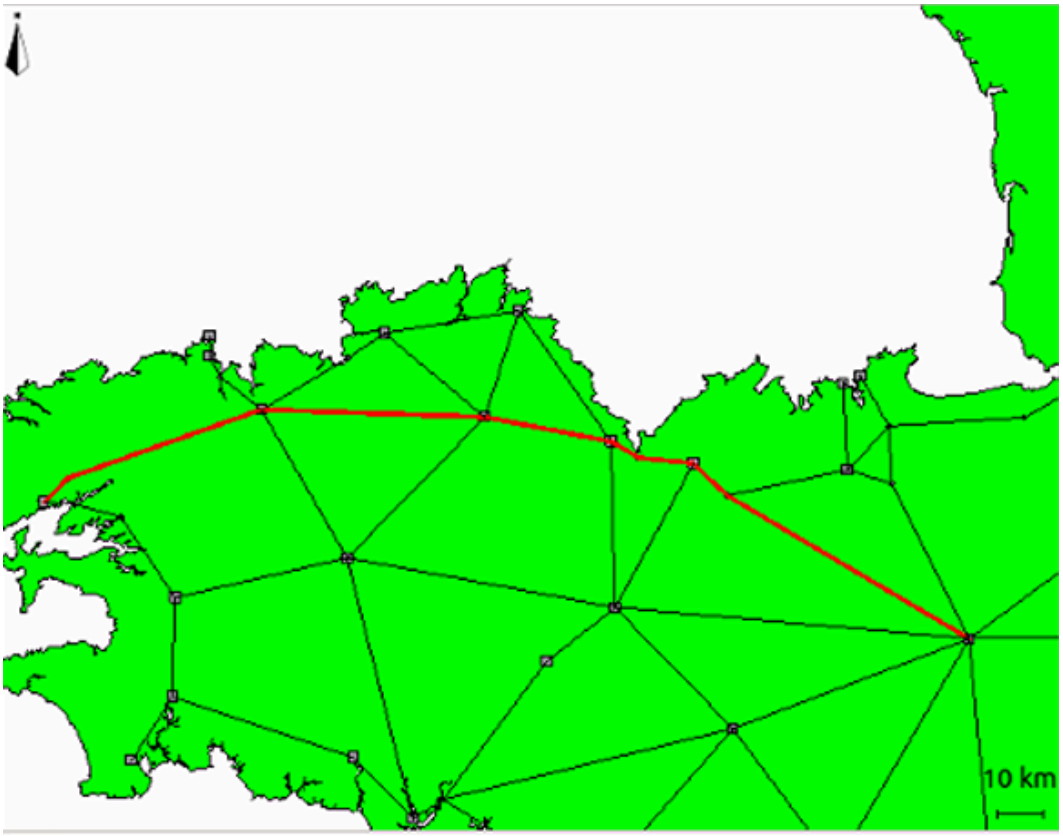
```
}
```

Fonction de calcul d’angle

A un moment donné, quand un raisonnement va aussi loin, il faut revenir sur terre et se dire qu’il ya plus simple. C’est donc ce que j’ai décidé de faire, revenir au fondamentaux, relire et analyser le sujet peut-être une information m’a échappé, comprendre proprement le problème. Je me suis remis à me poser des questions et à y réfléchir plus sérieusement. Des questions du genre: C’est quoi le souci ? Qu’est-ce que je voudrais voir comme résultat ? C’est la réponse à cette dernière qui m’a remis sur la voie vers la solution. La réponse était: Le chemin le plus court devrait passer sur les routes. J’ai commencé à prendre conscience du fait que mon approche était basée sur le calcul de distance cartésienne et donc c’est normal d’obtenir du direct. Cette réponse se traduit pour moi par le fait que je me suis trompé d’approche dans la construction de l’adjacency list à partir d’un seuil et de calcul de distance. Je suis donc retourné à la question fondamentale avec un angle de vision nouveau. Quelle est le critère qui permet d’identifier un voisin d’un waypoint donné ? Je me suis reformulé la question en me disant "Quand est-ce que deux points sont dits voisins ?” et la réponse qui naturellement est bien lorsqu’ils sont liés par une route. BOOOOM !!! Le critère que je cherchais depuis un bon moment, je l’ai trouvé. C’est l’existence d’une route en eux. Je m’empresse donc de confirmer mon hypothèse en allant jeter un coup d'œil à mes données et Bingo ! j’ai des données sur les routes associant deux waypoints. En plus j’avais déjà la distance qui les séparait.
Il ne me restait donc plus qu'à implémenter la méthode de création de l’adjacency list et à tester.



Correct waypoints's Graph



Chemin correct