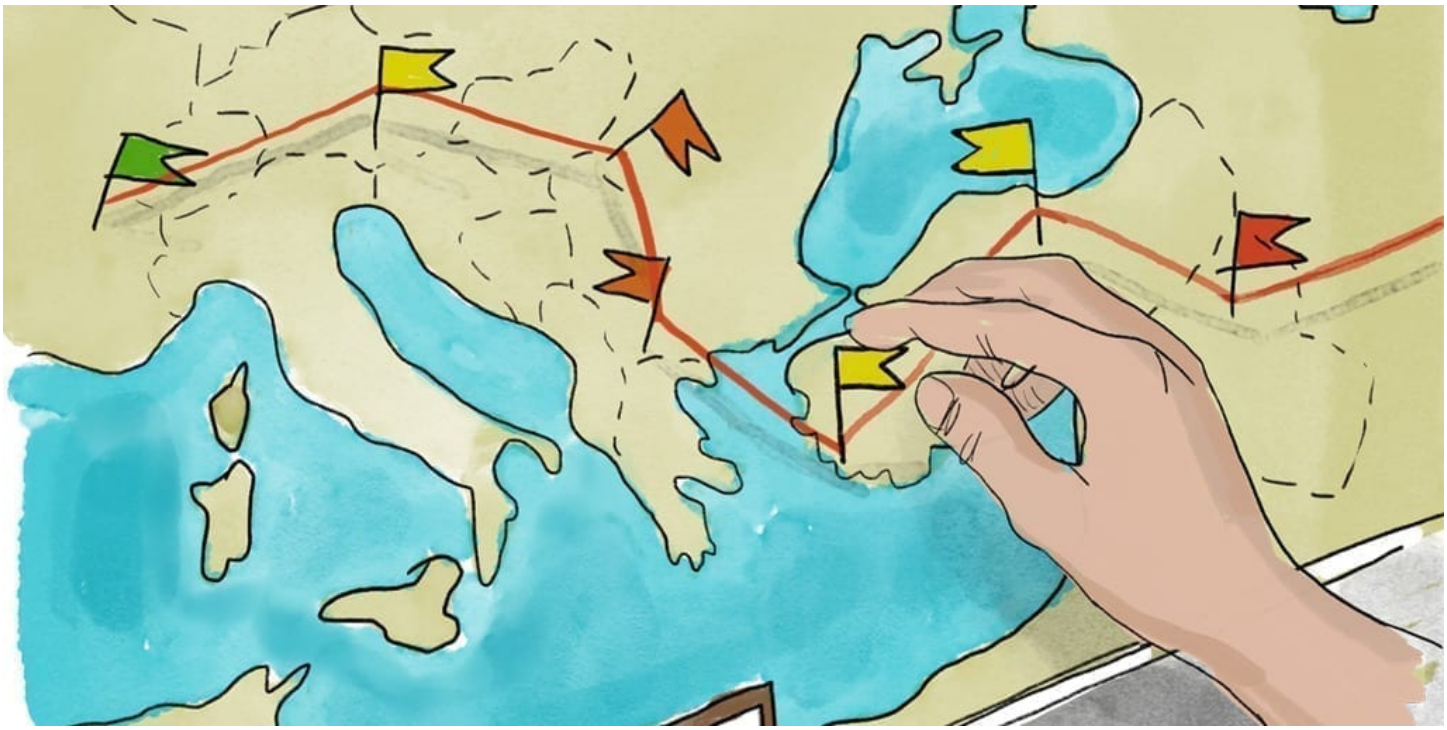
Travel Loader Rapport

DATE:	29/02/2024
REALISE PAR	**Christian Ndoradoumngue, Amine Amarzouk, Samy Hadj-ali et Kevin Hamza Djeradi**

OBJECTIF DU PROJET

L'objectif est de créer une IA qui détermine l'itinéraire de train le plus rapide lorsqu'il est dicté vocalement. Le message est retranscrit de l'oral à l'écrit puis analysé via des outils de traitement du langage naturel pour déterminer un itinéraire.



Organisation

Pour garantir une avancée efficace du projet, nous avons réparti les responsabilités de la manière suivante :

Développement de l'interface vocale : Un membre de l'équipe est chargé du développement de l'interface vocale. Son rôle consiste à concevoir et à mettre en œuvre les fonctionnalités permettant de transcrire de manière précise les commandes vocales en texte.

Intégration des outils de traitement du langage naturel : Un autre membre se concentre sur l'intégration des outils de traitement du langage naturel. Son objectif est de sélectionner les meilleurs outils disponibles et de les configurer pour analyser efficacement les requêtes transmises sous forme de texte.

Implémentation des algorithmes de pathfinding : Un troisième membre est en charge de l'implémentation des algorithmes de pathfinding. Son travail consiste à intégrer les algorithmes A* et Dijkstra dans le système, en veillant à leur efficacité et à leur adaptabilité aux besoins spécifiques du projet.

Tests et évaluation : Enfin, le dernier membre de l'équipe est responsable des tests et de l'évaluation du système. Il s'assure que l'IA fournit des résultats précis et rapides, et identifie les éventuels points d'amélioration à mesure que le développement progresse.

Communication et Collaboration :

Pour maintenir une communication fluide et une collaboration efficace, nous avons mis en place les mesures suivantes :

- Des réunions régulières sont organisées pour discuter de l'avancement du projet, partager les résultats obtenus et résoudre les éventuels problèmes rencontrés.
- Nous utilisons des outils de gestion de projet pour suivre les tâches assignées à chaque membre de l'équipe et pour assurer une traçabilité des actions réalisées.
- Un canal de communication instantanée est ouvert afin de faciliter les échanges rapides et spontanés entre les membres de l'équipe en cas de besoin.

Datasets

Dans un projet de Traitement du Langage Naturel (NLP), le dataset joue un rôle fondamental en tant que pilier central de la construction et de l'évaluation des modèles. Le dataset, ou ensemble de données, fournit les exemples sur lesquels les modèles de NLP sont formés, testés et évalués.

1- Dataset contenant les trajets et les gares

Ce dataset prend le fichier CSV d'horaires de train présent dans le projet en entrée, le formate pour rendre les données plus lisibles, puis écrit les informations formatées dans un nouveau fichier CSV appelé "timetables.formatted.csv". Elle s'assure également que les noms des gares de départ et d'arrivée commencent par "Gare de" pour assurer la cohérence des données. En résumé, cette fonction est essentielle pour préparer les données nécessaires à notre projet de manière efficace et organisée.

```
import csv

def parseTimeTable():
    with open('timetables.csv', 'r') as infile, open('timetables.formatted.csv', 'w', newline='') as outfile:
        csvreader = csv.reader(infile, delimiter='\t')
        csvwriter = csv.writer(outfile, delimiter=',')

        headers = next(csvreader)
        csvwriter.writerow([headers[0], "departure", "destination", headers[2]])

        for row in csvreader:
            trajet = row[1].split(" - ")

            if not trajet[0].startswith("Gare de"):
                trajet[0] = "Gare de " + trajet[0]
            if not trajet[1].startswith("Gare de"):
                trajet[1] = "Gare de " + trajet[1]

            departure = trajet[0].split("Gare de ")[1]
            destination = trajet[1].split("Gare de ")[1]

            formatted_row = [row[0], departure, destination, row[2]]
            csvwriter.writerow(formatted_row)

parseTimeTable()
```

2- **Dataset d'apprentissage du NER**

Le Dataset d'apprentissage du NER est un ensemble de données utilisé pour entraîner des modèles de reconnaissance d'entités nommées. Il comprend des textes annotés manuellement avec des balises indiquant les positions et les types des entités nommées. Ce dataset est crucial pour entraîner des modèles précis et robustes capables d'identifier différentes entités dans divers contextes, ce qui est essentiel pour de nombreuses applications de traitement automatique du langage naturel.

```python
# Liste de villes et modèles de phrases (simplifiés pour l'exemple)
villes = [
    "Paris", "Marseille", "Lyon", "Toulouse", "Nice", "Nantes", "Strasbourg", "Montpellier",
    "Bordeaux", "Lille", "Rennes", "Reims", "Le Havre", "Saint-Étienne", "Toulon", "Grenoble",
    "Dijon", "Angers", "Nîmes", "Villeurbanne", "Clermont-Ferrand", "Le Mans", "Aix-en-Provence",
    "Brest", "Tours", "Amiens", "Limoges", "Annecy", "Perpignan", "Boulogne-Billancourt",
    "Metz", "Besançon", "Orléans", "Saint-Denis", "Argenteuil", "Rouen", "Mulhouse", "Montreuil",
    "Caen", "Saint-Paul", "Nancy", "Nouméa", "Roubaix", "Tourcoing", "Nanterre", "Avignon",
    "Vitry-sur-Seine", "Créteil", "Dunkerque", "Poitiers"
]

phrases_modeles = [
    ("Je veux aller de {} à {}.", "START", "STOP"),
    ("Je souhaite me rendre de {} à {}.", "START", "STOP"),
    ("Je prévois de voyager de {} à {}.", "START", "STOP"),
    ("Mon itinéraire est de {} à {}.", "START", "STOP"),
    ("Je planifie un trajet de {} à {}.", "START", "STOP"),
    ("Mon voyage débute à {} et se termine à {}.", "START", "STOP"),
    ("Je dois me déplacer de {} à {}.", "START", "STOP"),
    ("J'envisage un déplacement de {} à {}.", "START", "STOP"),
    ("Je pars de {} pour arriver à {}.", "START", "STOP"),
    ("Je commence mon périple à {} pour finir à {}.", "START", "STOP"),
    ("Je m'apprête à partir de {} à {}.", "START", "STOP"),
    ("Mon départ est prévu de {} à destination de {}.", "START", "STOP"),
    ("Je compte quitter {} pour {}.", "START", "STOP"),
    ("Je projette de quitter {} pour {}.", "START", "STOP"),
    ("Mon intention est de voyager de {} à {}.", "START", "STOP"),
    ("Je dois effectuer le trajet de {} à {}.", "START", "STOP"),
    ("Je suis en route de {} vers {}.", "START", "STOP"),
    ("Je planifie de me déplacer de {} vers {}.", "START", "STOP"),
    ("Il est prévu que je me rende de {} à {}.", "START", "STOP"),

]
```

Cette liste de villes et de modèles de phrases est utilisée par les deux fonctions suivantes afin de créer le dataset utilisé pour entraîner ou tester des modèles de traitement du langage naturel

Le dataset est constitué de plus de 500 000 lignes afin d'optimiser l'apprentissage de notre modèle.

```python
def generer_dataset_index_name(n):
    """
    Generate a dataset containing sentences with named entities (cities), and the start and stop indexes of these entities within each sentence.

    Parameters:
    n (int): Number of sentences to generate in the dataset.

    Returns:
    dict: A dictionary containing three keys: 'sentence', 'start_index', and 'stop_index'.
    """
    data = {'sentence': [], 'start_index': [], 'stop_index': []}
    for _ in range(n):
        modele = random.choice(phrases_modeles)
        ville_start, ville_stop = random.sample(villes, 2)

        # Générer la phrase
        sentence = modele[0].format(ville_start, ville_stop)

        # Trouver les indices des villes de départ et d'arrivée
        start_index = sentence.find(ville_start)
        stop_index = sentence.find(ville_stop)

        # Ajouter les informations dans le dictionnaire
        data['sentence'].append(sentence)
        data['start_index'].append((start_index, start_index + len(ville_start)))
        data['stop_index'].append((stop_index, stop_index + len(ville_stop)))

    return pd.DataFrame(data)
```

```python
def generer_dataset_city_name(n):
    """
    Generates a dataset containing sentences with placeholders filled with randomly selected city names.
    It records the first and last city names used in each sentence.
    Parameters:
    n (int): The number of sentences to generate in the dataset.

    Returns:
    DataFrame: A pandas DataFrame with columns 'sentence' for the generated sentences,
               'START' for the first city name used in each sentence, and 'STOP' for the last city name used in each sentence.
    """
    data = {'sentence': [], 'START': [], 'STOP': []}
    for i in range(n):
        modele_tuple = random.choice(phrases_modeles)
        modele_str = modele_tuple[0]  # Extrait la chaîne de caractères du tuple
        villes_selectionnees = random.sample(villes, modele_str.count("{}"))

        sentence = modele_str.format(*villes_selectionnees)

        # Supposons que ville_start correspond au premier élément et ville_stop au dernier
        ville_start = villes_selectionnees[0]
        ville_stop = villes_selectionnees[-1]

        data['sentence'].append(sentence)
        data['START'].append(ville_start)
        data['STOP'].append(ville_stop)
        print(f"ligne {i}")
    return pd.DataFrame(data)

# Générer le DataFrame
df = generer_dataset_index_name(500000)

# Sauvegarder le DataFrame dans un fichier CSV
df.to_csv('dataset_travel_500K.csv', index=False)
```

Speech Recognition

Dans le cadre de notre projet de Reconnaissance d'Entités Nommées (NER), nous avons exploré diverses solutions de reconnaissance vocale pour convertir la parole en texte. Notre

objectif était de trouver la technologie la plus adaptée pour transcrire avec précision et efficacité les commandes vocales de l'utilisateur en texte, qui serait ensuite analysé par notre modèle NER. Ce document compare trois principales solutions : Google Speech-to-Text, PocketSphinx, et Microsoft Azure Speech Service.

---

1. Google Speech-to-Text

Bibliothèque Utilisée : SpeechRecognition en Python.

Avantages :

- Précision élevée : Excellente reconnaissance vocale, même dans des environnements bruyants.
- Support Multilingue : Large support des langues, y compris le français.
- Facilité d'intégration : Utilisation simple avec la bibliothèque SpeechRecognition.

Inconvénients :

- Connexion Internet : Nécessite une connexion internet stable.
- Limitations d'Usage Gratuit : Quotas et limitations sur l'utilisation gratuite.

Test :

```
import speech_recognition as sr;

r = sr.Recognizer()
with sr.Microphone() as source:
    audio = r.listen(source)
    text = r.recognize_google(audio, language='fr-FR')
```

2. PocketSphinx

Bibliothèque Utilisée : SpeechRecognition en Python.

- Avantages :

Fonctionne Hors Ligne : Aucune connexion internet requise.

Libre d'Usage : Pas de coûts d'utilisation ou de limitations d'API.

Inconvénients :

- Précision inférieure : Moins précis, particulièrement pour le français.
- Configuration requise : Peut nécessiter une configuration et un entraînement supplémentaires.

Code:

```
1
2    r.recognize_sphinx(audio)
```

3. Microsoft Azure Speech Service

Bibliothèque Utilisée : azure-cognitiveservices-speech.

Avantages :

- Haute Précision : Comparable à Google pour la qualité de reconnaissance.
- Personnalisation : Possibilité de personnaliser les modèles pour des cas d'usage spécifiques.

Inconvénients :

- Coût : Payant au-delà d'un certain quota d'utilisation.
- Dépendance Internet : Comme Google, nécessite une connexion internet.

Code:

```
import azure.cognitiveservices.speech as speechsdk
speech_config = speechsdk.SpeechConfig(subscription=speech_key, region=service_region)
speech_recognizer = speechsdk.SpeechRecognizer(speech_config=speech_config)
result = speech_recognizer.recognize_once_async().get()
```

Choix de la Solution pour Notre Projet

Après avoir évalué ces solutions, nous avons opté pour Google Speech-to-Text comme solution de reconnaissance vocale pour notre projet NER, et ce, pour plusieurs raisons :

Précision et Fiabilité : Google Speech-to-Text s'est montré significativement plus précis dans la reconnaissance du français, ce qui est crucial pour notre application visant à analyser des commandes vocales complexes.

Support Multilingue : Notre application ciblant potentiellement un public international, la capacité de Google à reconnaître une variété de langues est un avantage considérable.

Traitement du langage naturel(NLP)

Dans le cadre de notre projet de Reconnaissance d'Entités Nommées (NER), nous avons exploré diverses solutions de traitement du langage naturel pour analyser le texte transcrit à partir de commandes vocales. Notre objectif était de trouver la technologie la plus adaptée pour extraire avec précision les entités nommées et les informations pertinentes du texte vocal converti, afin de les utiliser dans notre modèle NER. Ce document compare trois principales solutions : SpaCy, NLTK (Natural Language Toolkit), et Stanford NER.

1- SpaCy:

Langues : Spacy prends en charge plusieurs langues, dont l'anglais, l'allemand, le français, l'espagnol, le portugais, l'italien, le néerlandais, etc.

Précision: SpaCy fournit des modèles pré-entraînés qui offrent une grande précision pour les tâches NER. Il utilise des architectures d'apprentissage profond comme CNN et LSTM.

Personnalisation : Permet d'affiner et de personnaliser les modèles pré-entraînés sur des données spécifiques à un domaine à l'aide de l'apprentissage par transfert.

Vitesse : SpaCy est connu pour sa vitesse et son efficacité, ce qui le rend adapté au traitement de grands volumes de données textuelles.

Intégration : Offre une intégration transparente avec d'autres composants SpaCy pour la tokenisation, l'étiquetage de la partie du discours, l'analyse des dépendances, etc.

Documentation et communauté : SpaCy dispose d'une documentation complète et d'une communauté active pour le support et les contributions.

Code:

```
import spacy

# Load the SpaCy English model
nlp = spacy.load("en_core_web_sm")

# Perform NER on a sample text
text = "Apple is looking at buying U.K. startup for $1 billion"
doc = nlp(text)

# Extract named entities from the text
for ent in doc.ents:
    print(ent.text, ent.label_)
```

2- NLTK (Natural Language Toolkit):

Langues : NLTK prend en charge plusieurs langues, mais peut nécessiter des ressources supplémentaires et des configurations manuelles pour les langues autres que l'anglais.

Précision: NLTK fournit des modèles probabilistes et basés sur des règles pour le NER, qui peuvent ne pas atteindre le même niveau de précision que les modèles basés sur l'apprentissage profond comme SpaCy.

Personnalisation : NLTK permet aux utilisateurs de créer des modèles NER personnalisés en utilisant différents algorithmes et ensembles de fonctionnalités, ce qui offre une certaine flexibilité mais nécessite plus d'efforts manuels.

Rapidité : NLTK peut ne pas être aussi efficace que SpaCy pour les tâches NER à grande échelle en raison de sa dépendance aux techniques traditionnelles d'apprentissage automatique.

Intégration : NLTK offre une intégration avec d'autres tâches et outils de TAL, ce qui le rend adapté à la recherche et à l'enseignement.

Documentation et communauté : NLTK dispose d'une documentation complète et d'une grande communauté d'utilisateurs, bien qu'il ne soit pas aussi activement maintenu que SpaCy.

Code:

```
from nltk.tag import StanfordNERTagger
from nltk.tokenize import word_tokenize

# Set the path to the Stanford NER model and JAR file
stanford_ner_path = 'path_to/stanford-ner-4.2.0'
model_path = stanford_ner_path + '/classifiers/english.all.3class.distsim.crf.ser.gz'
jar_path = stanford_ner_path + '/stanford-ner.jar'

# Initialize the Stanford NER tagger
st = StanfordNERTagger(model_path, jar_path)

# Tokenize the text
text = "Apple is looking at buying U.K. startup for $1 billion"
tokens = word_tokenize(text)

# Perform NER
tags = st.tag(tokens)

# Extract named entities from the tagged text
for tag in tags:
    if tag[1] != 'O':
        print(tag)
```

3- Stanford NER:

Langues : Stanford NER prend principalement en charge l'anglais, mais peut être adapté à d'autres langues avec des données de formation supplémentaires.

Précision: Stanford NER utilise des champs aléatoires conditionnels (CRF) et atteint une précision élevée pour les tâches NER en anglais, en particulier avec ses modèles pré-entraînés.

Personnalisation : Offre des options pour la formation de modèles NER personnalisés utilisant des CRF avec des caractéristiques définies par l'utilisateur, ce qui permet d'effectuer des tâches NER spécifiques à un domaine.

Vitesse : Stanford NER peut être plus lent que SpaCy en raison de son recours à la CRF, qui implique une ingénierie complexe des caractéristiques.

Intégration : Stanford NER peut être intégré dans des applications basées sur Java et offre également des liens pour Python via NLTK.

Documentation et communauté : Stanford NER dispose d'une bonne documentation et est largement utilisé dans la recherche et l'industrie, bénéficiant de la réputation du Stanford NLP Group.

```python
import nltk
from nltk.tag import StanfordNERTagger
import os

def stanford_ner(text):
    # Chemin vers le fichier .jar Stanford NER
    ner_jar_path = "/chemin/vers/stanford-ner.jar"

    # Chemin vers le modèle NER pré-entraîné (ex. anglais)
    ner_model_path = "/chemin/vers/english.all.3class.distsim.crf.ser.gz"

    # Initialisation du tagger NER Stanford
    st = StanfordNERTagger(ner_model_path, ner_jar_path)

    # Tokenisation du texte
    words = nltk.word_tokenize(text)

    # Appel de Stanford NER
    entities = st.tag(words)

    return entities

# Exemple d'utilisation :
texte = "Barack Obama was born in Hawaii."
resultat_ner = stanford_ner(texte)
print(resultat_ner)
```

Choix de la Solution pour Notre Projet

En conclusion, SpaCy excelle en termes de vitesse, de précision et de facilité d'utilisation, ce qui en fait notre choix privilégié pour notre tâches de NER.

NLTK offre une flexibilité et une valeur éducative, tandis que Stanford NER offre une grande précision pour la NER anglaise, mais peut nécessiter plus d'efforts pour la personnalisation et l'intégration par rapport à SpaCy.

Path Finding

Dans le cadre de notre projet impliquant la détermination de l'itinéraire de train le plus rapide, nous avons exploré divers algorithmes de pathfinding pour trouver la solution la plus

adaptée. Notre objectif était de sélectionner l'algorithme le plus efficace et précis pour calculer rapidement l'itinéraire optimal entre deux points sur un réseau de voies ferrées. Ce document compare trois principales solutions : A*, Dijkstra et l'algorithme de recherche en profondeur d'abord (Depth-First Search - DFS).

1- A* :

- **Avantages** :

Efficacité avec une bonne heuristique : A* est très efficace pour trouver le chemin le plus court entre deux points lorsque une bonne heuristique est disponible.

Garantie de l'optimalité avec une heuristique admissible : Si l'heuristique est admissible (ne surestime pas le coût restant), A* garantit de trouver l'itinéraire le plus court.

Utilisation efficace de la mémoire : A* utilise généralement moins de mémoire que Dijkstra car il n'a pas besoin de stocker tous les nœuds visités dans une file de priorité.

- **Inconvénients** :

Dépendance à une bonne heuristique : L'efficacité d'A* dépend fortement de la qualité de l'heuristique utilisée. Une mauvaise heuristique peut entraîner des performances médiocres voire une inefficacité totale.

Coût de calcul potentiellement élevé : Dans le pire des cas, A* peut réduire à Dijkstra, ce qui signifie qu'il peut avoir des coûts de calcul élevés dans certaines situations.

Code:

```python
def a_star(graph, start, goal, heuristic):
    queue = [(0, start)]
    visited = set()

    distances = {node: float('inf') for node in graph.nodes}
    distances[start] = 0

    while queue:
        current_distance, current_node = heapq.heappop(queue)

        if current_node == goal:
            return distances

        if current_node in visited:
            continue

        visited.add(current_node)

        for neighbor, cost in graph.edges[current_node]:
            distance = current_distance + cost
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(queue, (distance + heuristic(neighbor, goal), neighbor))

    return distances
```

2- Dijkstra :

- **Avantages :**

Optimalité : Dijkstra garantit de trouver l'itinéraire le plus court entre deux points, quelle que soit la configuration du graphe, tant que les coûts sont positifs.

Simplicité : Dijkstra est relativement simple à implémenter et à comprendre par rapport à A*, car il ne nécessite pas d'heuristique.

Adaptabilité : Peut être utilisé dans des graphes avec des coûts variables ou non uniformes.

- **Inconvénients :**

Complexité temporelle : La complexité temporelle de Dijkstra peut être élevée, en particulier dans les graphes de grande taille, car il explore tous les nœuds dans l'ordre de leur distance par rapport au point de départ.

Utilisation de la mémoire : Dijkstra peut utiliser plus de mémoire qu'A* car il stocke tous les nœuds visités dans une file de priorité.

Code:

```
def dijkstra(graph, start):
    queue = [(0, start)]
    visited = set()

    distances = {node: float('inf') for node in graph.nodes}
    distances[start] = 0

    while queue:
        current_distance, current_node = heapq.heappop(queue)

        if current_node in visited:
            continue

        visited.add(current_node)

        for neighbor, cost in graph.edges[current_node]:
            distance = current_distance + cost
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(queue, (distance, neighbor))

    return distances
```

3- Algorithme de Recherche en Profondeur d'Abord (DFS) :

- Avantages :

Simplicité : DFS est très simple à implémenter et à comprendre.

Utilisation efficace de la mémoire : DFS utilise généralement moins de mémoire que A* et Dijkstra car il ne stocke pas tous les nœuds visités dans une file de priorité.

Exploration exhaustive : Peut être utilisé pour explorer tout le graphe, pas seulement pour trouver un chemin spécifique.

- Inconvénients :

Pas de garantie d'optimalité : DFS ne garantit pas de trouver l'itinéraire le plus court entre deux points.

Performance médiocre dans certains cas : Peut être inefficace dans les graphes avec de nombreux chemins possibles, en particulier lorsque le chemin le plus court est profondément niché dans le graphe.

- **Code:**

```
def dfs(graph, start):
    visited = set()

    def dfs_util(node):
        if node in visited:
            return
        visited.add(node)
        for neighbor, _ in graph.edges[node]:
            dfs_util(neighbor)

    dfs_util(start)
    return visited
```

Choix de la Solution pour Notre Projet:

Nous avons choisi l'algorithme A* pour notre projet de Reconnaissance d'Entités Nommées (NER). Avec sa capacité à garantir l'optimalité tout en offrant une efficacité notable, A* répond le mieux à nos besoins. Cette décision nous permettra de maximiser la précision de notre système tout en minimisant le temps de calcul.

Cette approche renforcera la performance globale de notre projet, en assurant une reconnaissance précise et efficace des entités nommées à partir des commandes vocales transposées en texte.

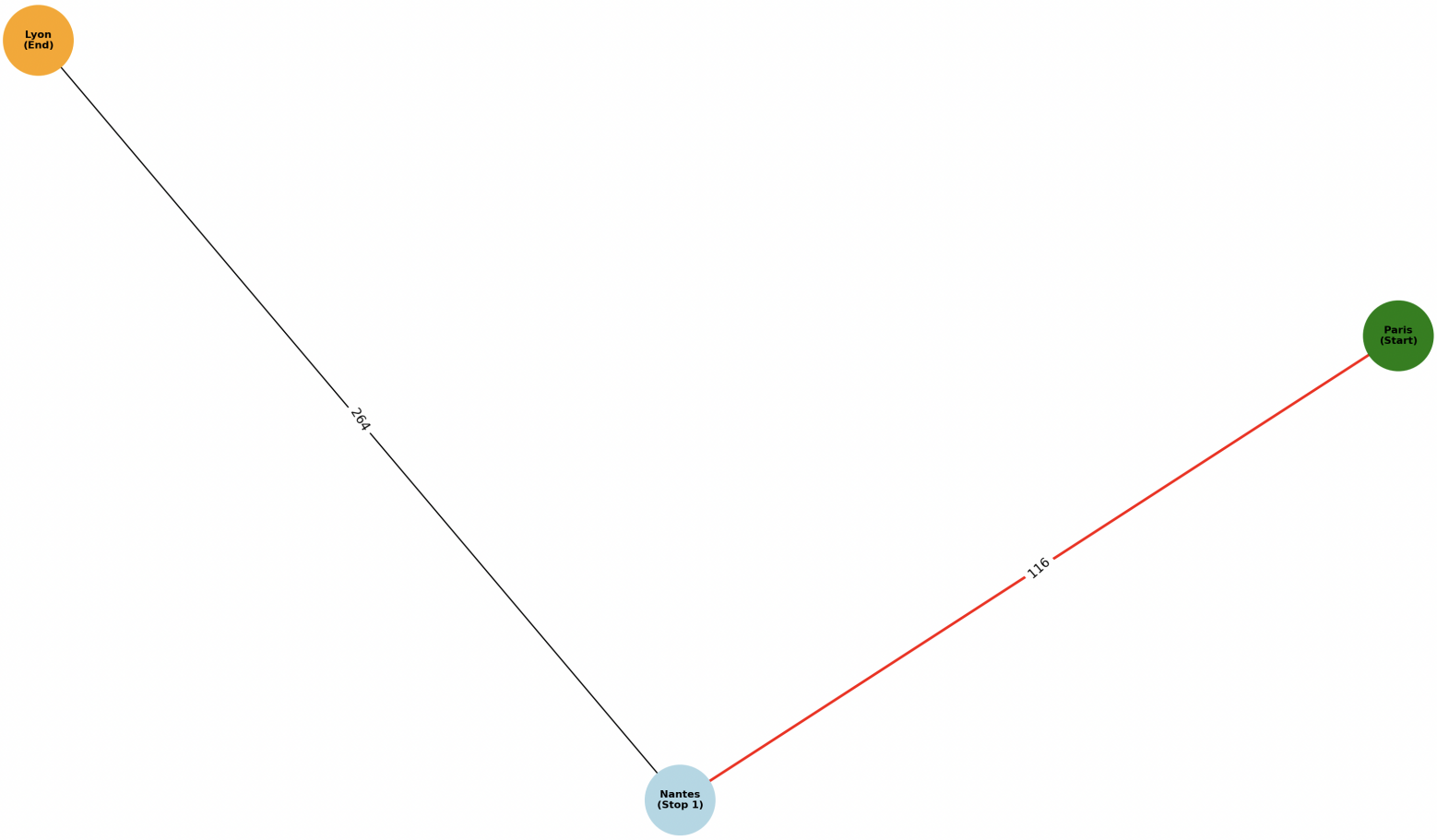
Résultats

1- Affichage des résultats

Les résultats de notre projets sont représentés dans un graphe représentant les relations entre les villes et leurs temps de trajet estimés, ce qui permettra par la suite d'effectuer des calculs et des analyses sur les itinéraires possibles.

```python
def create_graph(csv_file, start, end, stops):
    df = pd.read_csv(csv_file, sep=";")
    df["Relations"] = df["Relations"].str.title()

    G = nx.Graph()

    times = {}
    for _, row in df.iterrows():
        city1, city2 = row["Relations"].split(" - ")
        year = row["Année"]
        if (city1, city2) not in times or int(times[(city1, city2)][0]) < int(year):
            times[(city1, city2)] = (year, row["Temps estimé en minutes"])
        if (city2, city1) not in times or int(times[(city2, city1)][0]) < int(year):
            times[(city2, city1)] = (year, row["Temps estimé en minutes"])

    for city in [start] + stops + [end]:
        G.add_node(city)

    edge_length = 1.0  # Specify the desired length for all edges

    if stops:
        for i in range(len(stops) + 1):
            if i == 0:
                start_city = start
                end_city = stops[0]
            elif i == len(stops):
                start_city = stops[-1]
                end_city = end
            else:
                start_city = stops[i-1]
                end_city = stops[i]

            if (start_city, end_city) in times:
                weight = times[(start_city, end_city)][1]
                # Check for NaN before converting to int
                weight = int(weight) if not pd.isna(weight) else 0
                G.add_edge(start_city, end_city, weight=weight, length=edge_length)
            else:
                G.add_edge(start_city, end_city, weight=0, length=edge_length)
    else:
        if (start, end) in times:
            weight = times[(start, end)][1]
            # Check for NaN before converting to int
            weight = int(weight) if not pd.isna(weight) else 0
            G.add_edge(start, end, weight=weight, length=edge_length)
        else:
            G.add_edge(start, end, weight=0, length=edge_length)

    # Add coordinates as node attributes
    city_coordinates = {city: get_city_coordinates(city) for city in G.nodes()}

    # Add coordinates to the graph nodes
    for city, coords in city_coordinates.items():
        G.nodes[city]['coords'] = coords
```

```
    return G
```



2- Gestion d'erreur:

Un système de gestion d'erreur a été mis en place. Celui ci intervient lorsque le système ne parvient pas à reconnaître les villes mentionnées par l'utilisateur. Dans de tels cas, le système invite simplement l'utilisateur à répéter sa demande pour obtenir les informations nécessaires. Cette approche vise à améliorer l'expérience utilisateur en réduisant les frictions et en permettant au système de recueillir des données supplémentaires pour améliorer sa précision globale.

Conclusion

En conclusion, ce projet a été une exploration passionnante dans le domaine de la planification d'itinéraires ferroviaires et de la reconnaissance vocale. Nous avons développé et intégré avec succès plusieurs fonctionnalités clés, notamment la création d'un graphe représentant les relations entre les villes et leurs temps de trajet estimés, ainsi que la mise en

œuvre d'un système de reconnaissance vocale pour interpréter les demandes des utilisateurs.

L'analyse et le traitement des données ont été au cœur de notre démarche, avec une attention particulière portée à la qualité et à la cohérence des données. Nous avons utilisé des techniques de nettoyage et de prétraitement pour garantir la fiabilité des résultats obtenus.

De plus, nous avons veillé à améliorer l'expérience utilisateur en mettant en place un système de gestion d'erreur efficace pour gérer les cas où les villes mentionnées par l'utilisateur ne sont pas correctement reconnues.

Enfin, ce projet ouvre la voie à de nombreuses possibilités d'extension et d'amélioration. Des fonctionnalités telles que l'optimisation des itinéraires en fonction de critères spécifiques, l'intégration de données en temps réel sur les horaires de train et l'adaptation du système de reconnaissance vocale à un vocabulaire plus large pourraient être explorées dans le cadre de développements futurs.

Dans l'ensemble, ce projet a été une expérience enrichissante qui a combiné des aspects techniques et pratiques pour créer un système robuste et fonctionnel. Il constitue une base solide pour de futures innovations dans le domaine de la planification d'itinéraires ferroviaires et de la reconnaissance vocale.