Rapport de Projet

Générateur d'Itinéraires Touristiques Optimisés

Alexandre Da Silva, Roshan Jeyakumar, Wassim Badraoui

10 avril 2025

Table des matières

1 Introduction **3**
- 1.1 Contexte . . . 3
- 1.2 Objectifs du Projet . . . 3
- 1.3 Périmètre . . . 3

2 Architecture et Méthodologie **4**
- 2.1 Collecte de Données (`city_generator.py`) . . . 4
- 2.2 Représentation des Données (`city_graph.py`) . . . 4
- 2.3 Calcul des Temps de Trajet (`distance_api.py`) . . . 4
- 2.4 Optimisation de l'Itinéraire (`solver.py`) . . . 5
- 2.5 Interface Utilisateur (Frontend Web) . . . 5

3 Détails d'Implémentation **6**
- 3.1 Technologies Utilisées . . . 6
- 3.2 Structure du Code . . . 6
- 3.3 Points Clés de l'Implémentation . . . 6

4 Résultats et Démonstration **7**
- 4.1 Fonctionnement Typique . . . 7
- 4.2 Exemple de Sortie Formatée . . . 7
- 4.3 Génération de Faits Intéressants . . . 8

5 Discussion et Limitations **9**
- 5.1 Précision des Données LLM . . . 9
- 5.2 Estimations des Temps de Trajet . . . 9
- 5.3 Performance du Solveur . . . 9
- 5.4 Flexibilité des Contraintes . . . 9
- 5.5 Interface Utilisateur . . . 9

6 Conclusion et Perspectives **10**
- 6.1 Conclusion . . . 10
- 6.2 Perspectives d'Amélioration . . . 10

A Code Source **11**

B Références **11**

Résumé

Ce projet présente un système automatisé pour la génération d'itinéraires touristiques optimisés, désormais accessible via une interface web conviviale. En s'appuyant sur des modèles de langage avancés (LLM) via l'API OpenAI pour la collecte de données et sur des techniques de programmation par contraintes (OR-Tools) pour l'optimisation, le système propose des parcours cohérents et personnalisés. L'interface web permet aux utilisateurs de spécifier leurs préférences (destination, horaires, nombre de lieux, restaurants, visites obligatoires, méthode de calcul de distance) et de recevoir l'itinéraire généré directement dans leur navigateur. Le système optimise l'itinéraire en maximisant l'intérêt touristique tout en minimisant le temps de trajet, en se basant sur une représentation graphique de la ville (NetworkX) et un calcul réaliste des temps de trajet. Des faits intéressants sur la destination sont également présentés pendant la phase de planification.

1 Introduction

1.1 Contexte

La planification d'un voyage touristique, même pour une seule journée dans une ville, peut s'avérer complexe. Il faut jongler entre les lieux à visiter, leurs horaires d'ouverture, les distances à parcourir, les modes de transport disponibles, les pauses repas et les contraintes de temps globales. Effectuer cette planification manuellement est souvent chronophage et ne garantit pas une solution optimale en termes d'expérience ou d'efficacité.

1.2 Objectifs du Projet

L'objectif principal de ce projet est de développer une solution web capable de :

— Collecter automatiquement des données à jour sur les points d'intérêt (attractions touristiques, restaurants) d'une ville spécifiée, incluant leurs coordonnées, horaires, type, intérêt, durée de visite estimée et coût.
— Calculer ou estimer les temps de trajet entre ces points d'intérêt en utilisant différents modes de transport (marche, transports en commun, voiture).
— Générer un itinéraire optimisé pour une journée, respectant un créneau horaire défini par l'utilisateur.
— Intégrer des contraintes spécifiques comme des visites obligatoires ou la nécessité d'inclure un ou plusieurs repas au restaurant.
— Optimiser l'itinéraire selon des critères multiples : maximisation de l'intérêt total des lieux visités et minimisation du temps de trajet.
— Afficher l'itinéraire résultant de manière claire dans le navigateur.
— Fournir également des informations contextuelles (faits intéressants) sur la ville.

1.3 Périmètre

Le système se concentre sur la génération d'itinéraires pour une seule journée dans une ville donnée. Il utilise des données générées par LLM, qui peuvent nécessiter une vérification pour une précision absolue. L'optimisation est réalisée à l'aide d'un solveur de contraintes, visant une solution optimale ou quasi-optimale dans un temps de calcul raisonnable.

2 Architecture et Méthodologie

Le système est structuré autour de plusieurs modules Python interdépendants, chacun responsable d'une partie spécifique du processus.

2.1 Collecte de Données (`city_generator.py`)

Ce module est responsable de l'interaction avec l'API OpenAI pour générer les données nécessaires :

- **Génération des POI :** La fonction `generate_city_data` utilise un LLM (GPT-4o) pour créer une liste de points d'intérêt (attractions et restaurants) pour une ville donnée. Le prompt spécifie le format JSON attendu, incluant ID, nom, horaires, type, intérêt (score 1-10), durée de visite, coût, et coordonnées GPS précises. Une étape de "sanitization" assure que les types de données sont corrects (entiers, flottants).
- **Génération de Faits Intéressants :** La fonction `generate_city_fun_facts` utilise un LLM plus léger (GPT-4o-mini) pour générer une liste de faits amusants et pertinents sur la ville, également au format JSON.
- **Création du Graphe Initial :** Après la génération des POI, les données sont utilisées pour créer un graphe initial de la ville via la fonction `create_graph` du module `city_graph`.

L'utilisation de prompts système et utilisateur spécifiques vise à guider le LLM pour obtenir des données structurées et pertinentes.

2.2 Représentation des Données (`city_graph.py`)

La ville et ses points d'intérêt sont modélisés sous forme de graphe non orienté à l'aide de la bibliothèque `NetworkX`.

- **Nœuds :** Chaque nœud représente un POI (attraction ou restaurant) et stocke ses attributs (nom, type, horaires, intérêt, durée, coût, coordonnées, etc.). Les identifiants des nœuds sont assurés d'être des entiers.
- **Arêtes :** Initialement, des arêtes sont créées entre toutes les paires de nœuds. Ces arêtes stockeront ultérieurement les informations de temps de trajet calculées.
- **Persistance :** Le graphe généré pour une ville est sauvegardé dans un fichier binaire (`.pkl`) à l'aide de `pickle` pour une réutilisation rapide (`save_graph`, `load_graph`). Cela évite de devoir régénérer les données à chaque exécution.

2.3 Calcul des Temps de Trajet (`distance_api.py`)

Le module `DistanceCalculator` gère le calcul des temps de trajet entre deux POI pour différents modes de transport.

- **Modes de Transport :** Marche, transports en commun, voiture.
- **API OpenAI :** Par défaut (`use_api=True`), le module interroge l'API OpenAI (GPT-4o-mini) pour obtenir des estimations de temps de trajet réalistes basées sur les coordonnées et le mode de transport.
- **Batching :** Pour optimiser les appels API, les requêtes de temps de trajet peuvent être regroupées par lots (`batch_size`).
- **Caching :** Les résultats des calculs de temps de trajet sont mis en cache en mémoire pour éviter les requêtes répétées pour les mêmes paires origine-destination-mode.
- **Fallback :** Si l'API n'est pas utilisée (`use_api=False`) ou en cas d'erreur API, une méthode de secours calcule une estimation basée sur la distance Haversine et des vitesses moyennes prédéfinies pour chaque mode de transport.

2.4 Optimisation de l'Itinéraire (`solver.py`)

Le cœur du système réside dans la classe `TouristItinerarySolver`, qui utilise le solveur CP-SAT de Google OR-Tools.

- **Initialisation :** Charge le graphe de la ville, définit les paramètres du tour (heures de début/-fin, visites obligatoires, nombre de restaurants souhaités, nombre max de POI), et initialise le calculateur de distance.
- **Pré-calculs :**
  - **Voisins les Plus Proches :** Calcule les `max_neighbors` plus proches voisins pour chaque POI en se basant sur la distance Haversine, afin de limiter le nombre d'arêtes à considérer activement dans le modèle d'optimisation.
  - **Temps de Trajet Préférés :** Détermine et met en cache le mode de transport "préféré" et le temps de trajet associé entre les voisins proches et les POI obligatoires (`_precompute_travel_times`, `_select_preferred_transport_mode`). La sélection du mode prend en compte la distance et applique des heuristiques (privilégier la marche pour courtes distances, les transports en commun pour moyennes distances, avec une pénalité pour la voiture).
- **Modélisation CP-SAT :**
  - **Variables de Décision :** Variables booléennes pour indiquer si un POI est visité (`visit[i]`), et s'il est à une certaine position dans l'itinéraire (`pos[i][p]`). Variables entières pour les heures d'arrivée (`arrival_time[i]`) et de départ (`departure_time[i]`) de chaque POI, exprimées en minutes relatives au début de la journée.
  - **Contraintes :**
    - Structurelles : Unicité des POI, unicité des positions, pas de "trous" dans l'itinéraire.
    - Temporelles : Respect de la durée de visite, respect des horaires d'ouverture (gestion des intervalles multiples), contrainte de temps total disponible.
    - Séquencement : Le temps d'arrivée à un POI doit être supérieur ou égal au temps de départ du POI précédent plus le temps de trajet (`get_travel_time` utilise les temps pré-calculés).
    - Restaurants : Contraintes spécifiques pour inclure le nombre requis de restaurants (lunch et/ou dîner) pendant les créneaux horaires appropriés (12h-13h, 19h-20h). Contraintes pour éviter les visites touristiques pendant les heures de repas si aucun restaurant n'est prévu à ce moment-là.
    - Spécifiques : Visites obligatoires, nombre maximum de POI touristiques.
  - **Objectif :** Maximiser une fonction combinée : '(Score d'Intérêt Total * 10) - Temps de Trajet Total'. Le score d'intérêt est pondéré plus fortement pour le favoriser.
- **Résolution :** Le solveur CP-SAT cherche une solution optimale ou faisable respectant toutes les contraintes dans un temps limite (e.g., 60 secondes).
- **Formatage :** La solution trouvée est formatée en un itinéraire lisible (`format_itinerary`) et peut être convertie en dictionnaire (`_convert_itinerary_to_dict`).

2.5 Interface Utilisateur (Frontend Web)

Développée en HTML, CSS et JavaScript, l'interface utilisateur est le point d'entrée pour l'utilisateur. Elle permet de :

- Saisir les paramètres de l'itinéraire via un formulaire structuré et stylisé (destination, horaires, nombre max de POI, nombre de restaurants, visites obligatoires textuelles, méthode de calcul de distance).
- Soumettre la demande au backend via des requêtes asynchrones (Fetch API) vers des endpoints dédiés (e.g., `/api/plan`, `/api/fun-facts`).
- Afficher un état de chargement interactif pendant le traitement backend, agrémenté de faits intéressants cycliques sur la ville demandée.
- Afficher l'itinéraire final ou les messages d'erreur dans une section dédiée de la page.

3 Détails d'Implémentation

3.1 Technologies Utilisées

- **Langage :** Python 3.x
- **Bibliothèques Principales :**
  - `openai` : Interaction avec l'API OpenAI pour la génération de données et les temps de trajet.
  - `ortools` : Modélisation et résolution du problème d'optimisation par contraintes (CP-SAT).
  - `networkx` : Création, manipulation et stockage de la structure de graphe de la ville.
  - `json` : Manipulation des données JSON retournées par l'API.
  - `pickle` : Sérialisation/désérialisation du graphe NetworkX.
  - `os`, `sys` : Gestion des chemins de fichiers et du path système.
  - `math` : Calculs mathématiques (Haversine).
  - `datetime` : Manipulation des heures.
  - `dotenv` (implicitement via `load_dotenv` dans `solver.py`) : Gestion des clés API via un fichier `.env`.
  - `matplotlib` (dans `city_graph.py`) : Visualisation basique du graphe (fonction `display_graph_wi`
- **Frontend :** HTML5, CSS3, JavaScript (ES6+).
- **Librairies/Outils Frontend :** Bootstrap Icons (pour les icônes). Utilisation native de la Fetch API pour les requêtes asynchrones.

3.2 Structure du Code

Le code est organisé en modules logiques :

- `src/` : Contient le code source principal.
  - `city_generator.py` : Génération des données initiales.
  - `distance_api.py` : Calcul des distances/temps de trajet.
  - `solver.py` : Logique d'optimisation de l'itinéraire.
- `data/` : Contient les données persistantes et la logique associée.
  - `city_graph.py` : Fonctions pour créer, sauvegarder, charger le graphe.
  - `city_graphs/` (créé par `city_graph.py`) : Sous-dossier où les fichiers `.pkl` des graphes sont sauvegardés.

Une configuration via un fichier `.env` est recommandée pour stocker la clé API OpenAI.

3.3 Points Clés de l'Implémentation

- **Gestion d'Erreurs :** Des blocs `try...except` sont utilisés, notamment lors des appels API et de la manipulation de données (parsing JSON, conversion de types), pour améliorer la robustesse. Des mécanismes de fallback (e.g., distance Haversine) sont présents.
- **Sanitization des Données :** Le script `city_generator.py` inclut une étape pour s'assurer que les données numériques (ID, intérêt, durée, coût, coordonnées) sont correctement formatées après leur récupération depuis le LLM.
- **Optimisation des Appels API :** L'utilisation du caching et du batching dans `distance_api.py` vise à réduire le nombre d'appels à l'API OpenAI et donc les coûts et temps de latence.
- **Pré-calculs dans le Solveur :** Le calcul préalable des voisins les plus proches et la sélection du mode de transport réduisent la complexité du modèle CP-SAT soumis au solveur.
- **Modélisation CP-SAT Détaillée :** Le solveur implémente des contraintes fines, notamment pour la gestion des horaires d'ouverture et l'intégration logique des repas, en utilisant des variables booléennes intermédiaires et des implications (`OnlyEnforceIf`, `AddImplication`).

4 Résultats et Démonstration

4.1 Fonctionnement Typique

Pour utiliser le système, un utilisateur (ou un script principal) instancierait `TouristItinerarySolver` en spécifiant la ville, les heures de début/fin, et éventuellement des contraintes supplémentaires (visites obligatoires, nombre de restaurants). Le solveur se chargerait de :

1. Charger ou générer le graphe de la ville (appelant `city_generator` si nécessaire).
2. Pré-calculer les temps de trajet pertinents (appelant `distance_api`).
3. Construire et résoudre le modèle CP-SAT.
4. Retourner l'itinéraire optimisé.

La fonction `format_itinerary` permet ensuite de présenter ce résultat de manière conviviale.

4.2 Exemple de Sortie Formatée

Ci-dessous un exemple *fictif* de ce que la sortie formatée pourrait ressembler (basé sur la structure de la fonction `format_itinerary`) :

```
Optimized Tourist Itinerary:
============================

1. Tour Eiffel (Touristique)
   Arrival: 09:00, Departure: 10:30
   Duration: 90 minutes
   Interest: 10/10
   Cost: €25.0
   Travel to next: 15 minutes by walking

2. Musée du Louvre (Touristique)
   Arrival: 10:45, Departure: 12:45
   Duration: 120 minutes
   Interest: 9/10
   Cost: €17.0
   Travel to next: 10 minutes by public transport

3. Le Bouillon Chartier (Restaurant)
   Arrival: 12:55, Departure: 13:55
   Duration: 60 minutes
   Interest: 7/10
   Cost: €20.0
   Travel to next: 20 minutes by public transport

4. Cathédrale Notre-Dame (Touristique)
   Arrival: 14:15, Departure: 15:15
   Duration: 60 minutes
   Interest: 9/10
   Cost: €0.0
   Travel to next: 25 minutes by walking

5. Sainte-Chapelle (Touristique)
   Arrival: 15:40, Departure: 16:25
```

```
   Duration: 45 minutes
   Interest: 8/10
   Cost: €11.5
   Travel to next: 30 minutes by public transport

6. Montmartre / Sacré-Cœur (Touristique)
   Arrival: 16:55, Departure: 18:10
   Duration: 75 minutes
   Interest: 8/10
   Cost: €0.0

Summary:
Total POIs visited: 6
Total interest score: 51
Total time (including travel): 550 minutes
Total cost: €73.5
Transport modes used: {'walking': 2, 'public transport': 3, 'car': 0}
```

4.3 Génération de Faits Intéressants

Le module `city_generator` peut également fournir une liste de faits intéressants sur la ville, comme :

```
["Paris est surnommée la 'Ville Lumière' depuis le règne de Louis XIV.",
 "La Tour Eiffel devait être démontée après l'Exposition Universelle de 1889.",
 "Le Louvre est le plus grand musée d'art du monde en termes de surface.",
 ...]
```

5 Discussion et Limitations

5.1 Précision des Données LLM

La qualité de l'itinéraire dépend fortement de la précision des données générées par le LLM (coordonnées GPS, horaires d'ouverture, durées de visite, intérêt). Bien que les modèles comme GPT-4o soient performants, des erreurs ou des imprécisions peuvent survenir. La phase de "sanitization" atténue certains problèmes de format, mais ne valide pas l'exactitude sémantique des données. Une validation croisée avec des sources de données plus fiables (API dédiées, bases de données ouvertes) pourrait améliorer la robustesse.

5.2 Estimations des Temps de Trajet

Les temps de trajet fournis par l'API OpenAI sont des estimations. Ils ne tiennent pas compte des conditions de trafic en temps réel, des retards imprévus dans les transports en commun, ou des difficultés de stationnement pour les trajets en voiture. Le fallback basé sur la distance Haversine est une approximation encore plus grossière. La sélection du mode de transport "préféré" est basée sur des heuristiques qui pourraient ne pas convenir à tous les utilisateurs ou à toutes les situations.

5.3 Performance du Solveur

Le solveur CP-SAT est puissant mais sa performance peut se dégrader avec l'augmentation du nombre de POI, du nombre de contraintes, ou de la durée de l'horizon temporel. Le pré-calcul des voisins proches aide à limiter la taille du problème, mais pour des villes très denses ou des demandes très complexes, le temps de résolution pourrait dépasser la limite fixée.

5.4 Flexibilité des Contraintes

Le modèle actuel intègre des contraintes courantes, mais pourrait être étendu pour inclure :
— Des budgets de coût total.
— Des préférences thématiques (e.g., musées d'art, parcs, histoire).
— La prise en compte de l'accessibilité pour les personnes à mobilité réduite.
— Des itinéraires sur plusieurs jours.

5.5 Interface Utilisateur

Le projet bénéficie d'une interface web dédiée. L'itinéraire est présenté textuellement. Une représentation sur une carte interactive (ex : avec Leaflet, OpenLayers, Mapbox GL JS) améliorerait grandement la compréhension spatiale.

6 Conclusion et Perspectives

6.1 Conclusion

Ce projet a démontré avec succès la faisabilité de la création d'un générateur d'itinéraires touristiques optimisés en combinant la puissance des LLM pour la collecte de données et la robustesse des solveurs de programmation par contraintes pour l'optimisation. Le système est capable de produire des itinéraires cohérents et personnalisés, respectant un ensemble de contraintes temporelles, logistiques et de préférences (visites obligatoires, repas). L'architecture modulaire et l'utilisation de bibliothèques standards comme NetworkX et OR-Tools rendent le système compréhensible et potentiellement extensible.

6.2 Perspectives d'Amélioration

Plusieurs pistes peuvent être explorées pour améliorer et étendre ce projet :

— **Amélioration de la Qualité des Données :** Intégrer des API spécifiques (e.g., Google Maps/-Places, OpenStreetMap) pour valider ou remplacer les données générées par LLM, notamment les horaires et les temps de trajet en temps réel.

— **Interface Utilisateur :** Développer une interface web ou mobile pour faciliter la saisie des paramètres et la visualisation de l'itinéraire (potentiellement sur une carte).

— **Personnalisation Avancée :** Permettre aux utilisateurs de spécifier des profils d'intérêt, des budgets, des contraintes de mobilité, ou de choisir explicitement les modes de transport préférés entre chaque étape.

— **Planification Multi-Jours :** Étendre le modèle pour générer des itinéraires sur plusieurs jours, en tenant compte de l'hébergement et de la continuité entre les journées.

— **Feedback et Apprentissage :** Intégrer une boucle de feedback où les utilisateurs pourraient noter les itinéraires générés, permettant potentiellement d'affiner les modèles ou les heuristiques utilisées.

— **Intégration de Réservations :** Connecter le système à des plateformes de réservation pour les billets d'entrée ou les restaurants.

Ces développements pourraient transformer cet outil en une solution de planification de voyage encore plus complète et performante.

Annexes

A Code Source

Le code source complet du projet est disponible sur GitHub : https://github.com/dslalex/TouristItineraryPlanner

B Références

Voici quelques références académiques et techniques pertinentes aux problèmes abordés dans ce projet :

Références

[1] Google AI. *OR-Tools : Google Optimization Tools*. https://developers.google.com/optimization

[2] Rossi, F., Van Beek, P., & Walsh, T. (Eds.). (2006). *Handbook of Constraint Programming*. Elsevier.

[3] Gavalas, D., Konstantopoulos, C., Mastakas, K., & Pantziou, G. (2014). A survey on algorithmic approaches for tourist trip design problems. *Journal of Heuristics*, 20(3), 291-343.

[4] Toth, P., & Vigo, D. (Eds.). (2014). *Vehicle Routing : Problems, Methods, and Applications (Second Edition)*. SIAM.

[5] OpenAI. *OpenAI API Documentation*. https://platform.openai.com/docs

[6] Sinnott, R. W. (1984). Virtues of the Haversine. *Sky and Telescope*, 68(2), 159.

[7] Zhao, W. X., Zhou, K., Li, J., Tang, T., Wang, X., Hou, Y., ... & Wen, J. R. (2023). A Survey of Large Language Models. *arXiv preprint arXiv :2303.18223*.