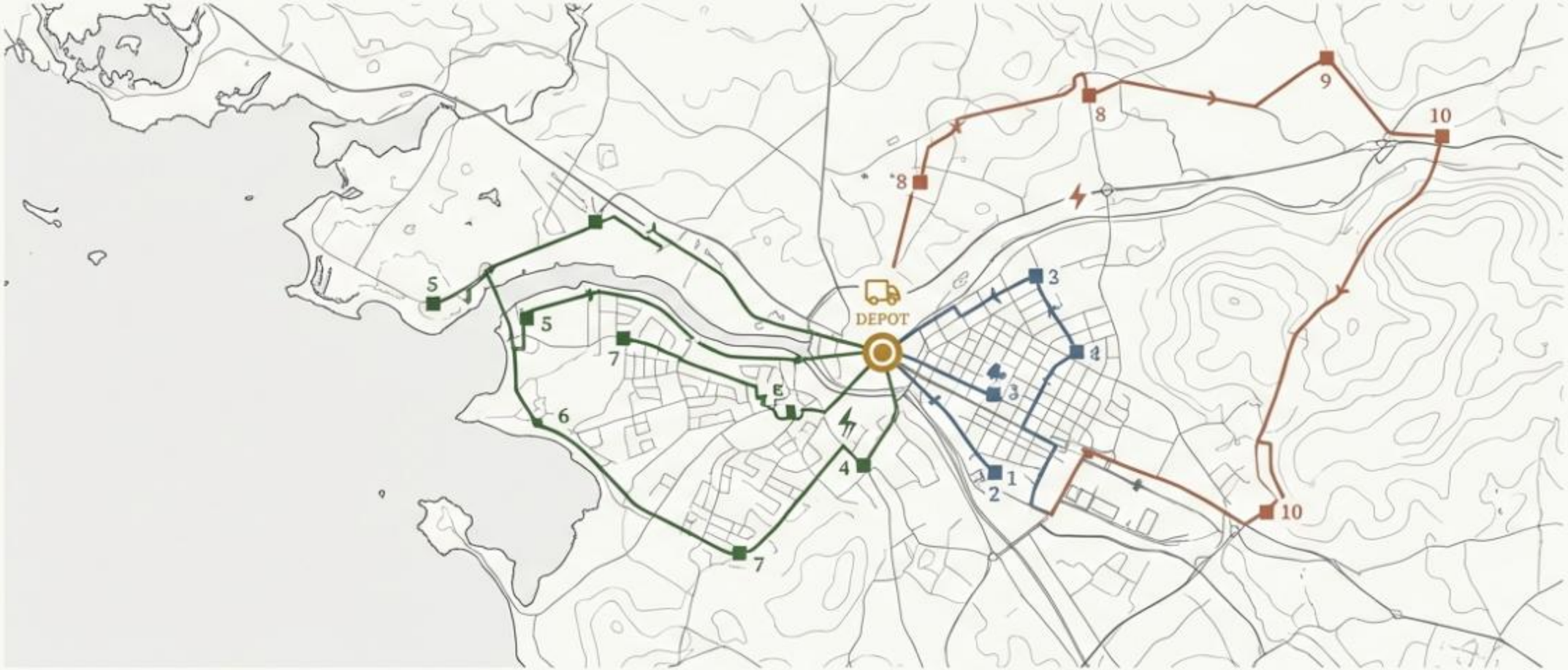
Optimisation de Tournées de Véhicules : Un Solveur VRP Interactif

Implémentation d'un solveur pour VRP Classique (CVRP) et Vert (E-VRP) avec une interface web dynamique.



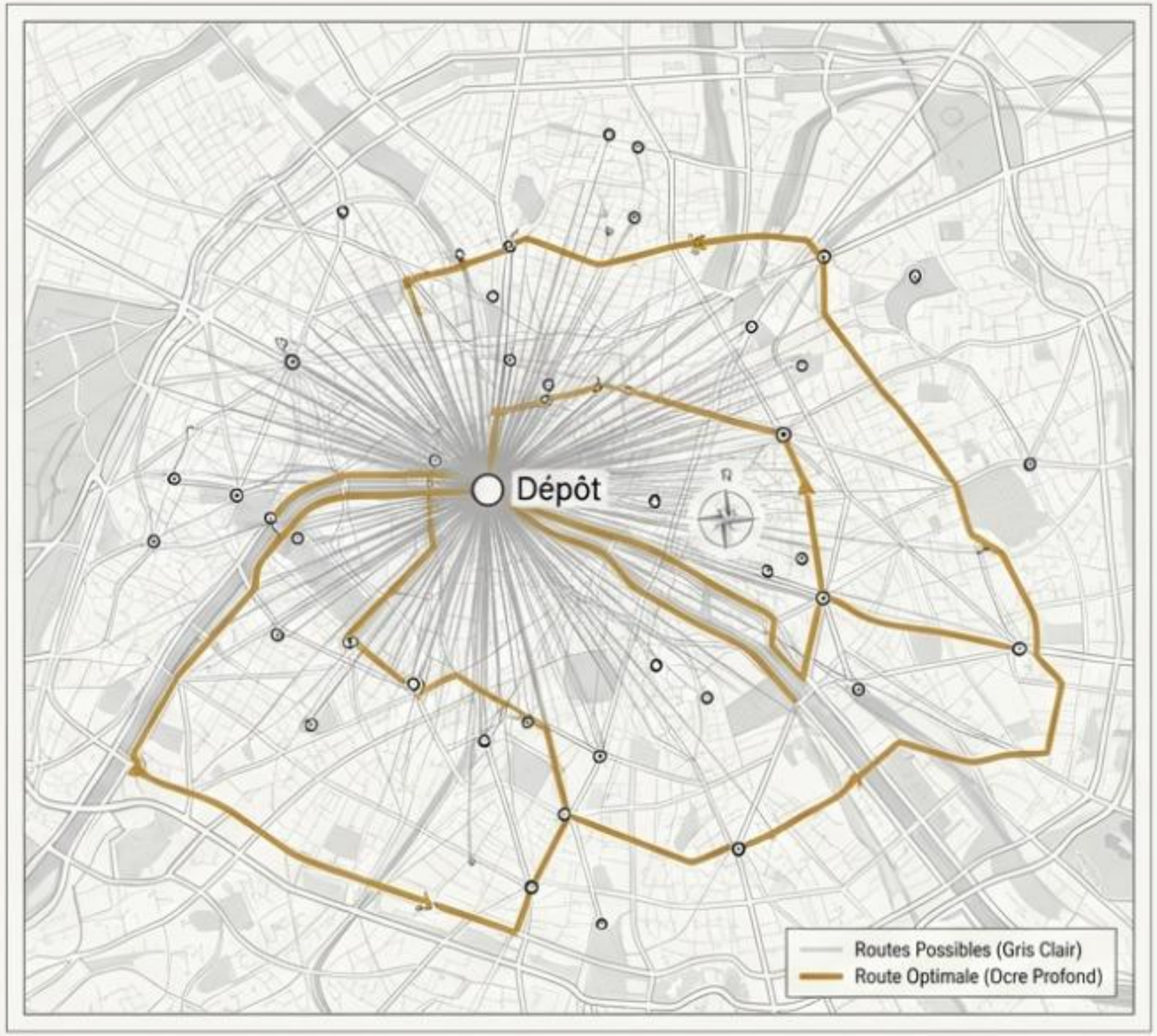
Alexis DHERMY, Clément CARON, Grégoire BRUN

Projet réalisé dans le cadre du cours d'Intelligence Artificielle II, EPF.

Le défi logistique : des milliards de routes possibles, une seule optimale.

Le Problème de Tournées de Véhicules (VRP) est un défi d'optimisation combinatoire au cœur de la logistique mondiale. Déterminer les tournées optimales pour une flotte de véhicules est crucial pour réduire les coûts, les délais et l'impact environnemental.

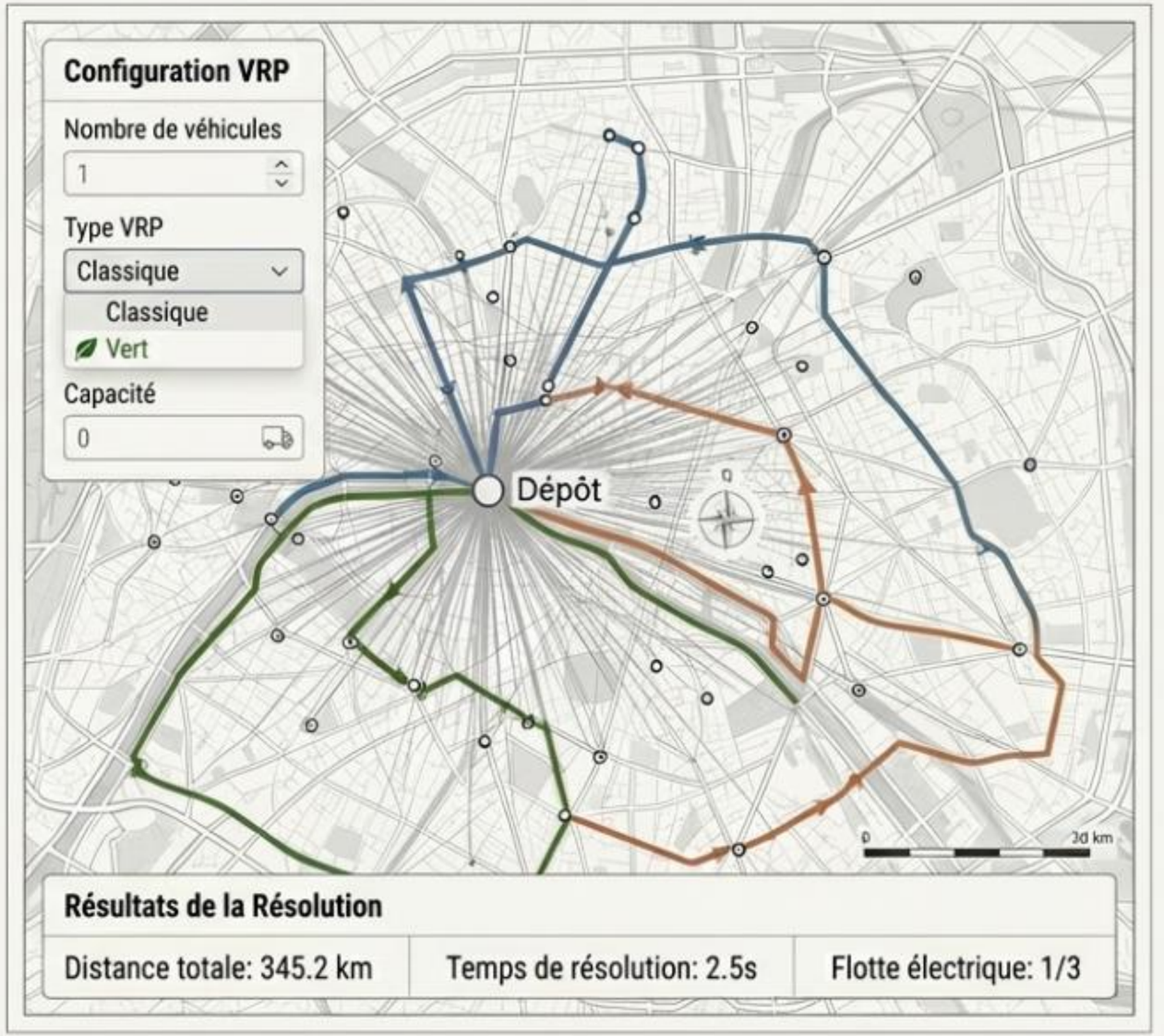
- **Flottes de véhicules :** Gérer de multiples véhicules avec des contraintes uniques.
- **Demandes clients :** Servir un grand nombre de clients avec des besoins spécifiques.
- **Fenêtres temporelles :** Respecter des créneaux de livraison stricts.
- **Impact environnemental :** Minimiser la distance et la consommation, surtout pour les flottes électriques.



Notre solution : une application web complète pour visualiser et résoudre le VRP.

Nous avons développé une plateforme interactive qui modélise, résout et visualise des problèmes de tournées complexes en temps réel, supportant à la fois les flottes traditionnelles et électriques.

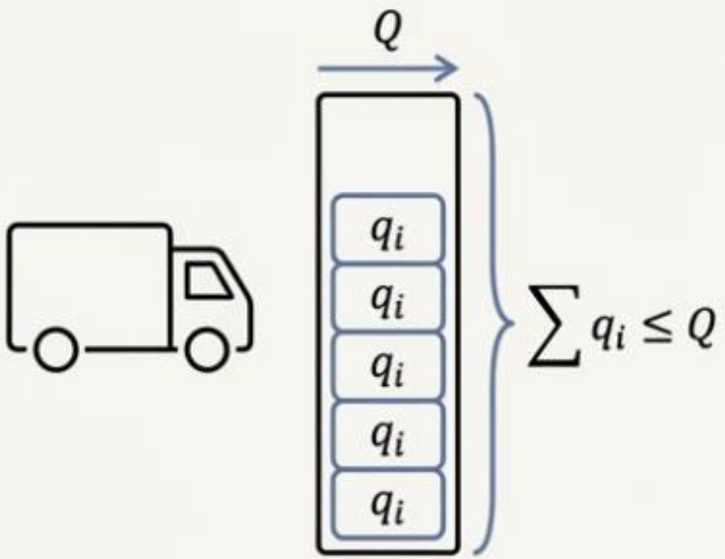
- **Interface web interactive** avec carte dynamique (Leaflet.js)
- **Double compétence :** supporte le VRP Classique (capacité, temps) et le VRP Vert (autonomie, recharge)
- **Résolution en temps réel** avec suivi de la progression
- **Configuration flexible** : véhicules multiples, capacités, fenêtres temporelles...
- **Précision géographique** avec calcul de distances via la formule de Haversine



Les règles du jeu : anatomie du Problème de Tournées de Véhicules

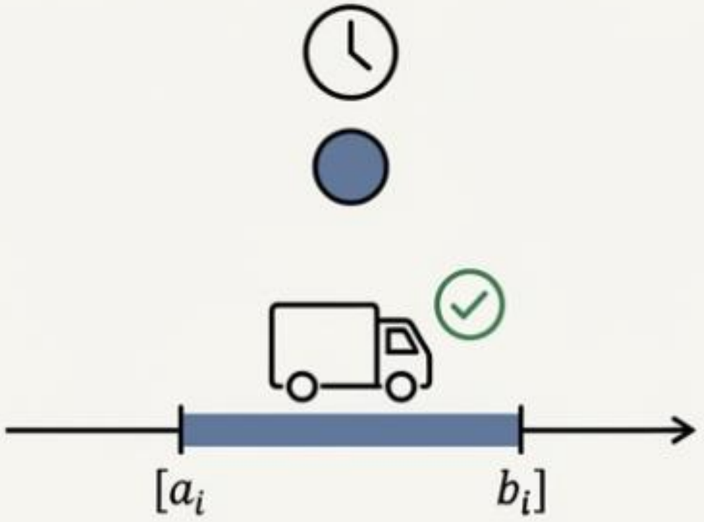
Objectif principal : Minimiser la distance totale parcourue par une flotte de véhicules.

1. Capacité (CVRP)



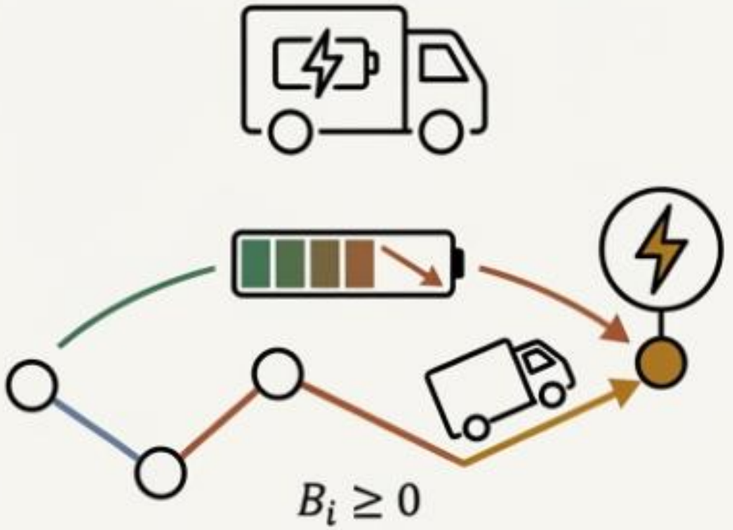
La somme des demandes des clients sur une tournée ne doit pas excéder la capacité du véhicule.

2. Fenêtres Temporelles (VRPTW)



Chaque client doit être visité dans son créneau horaire défini $[a_i, b_i]$.

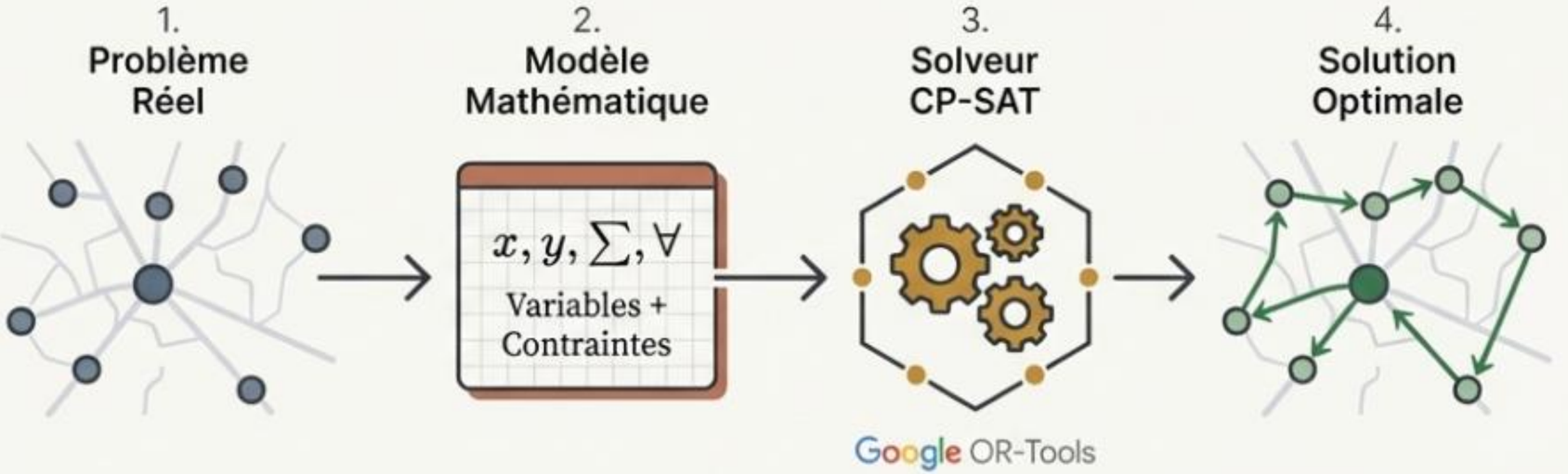
3. Autonomie (E-VRP)



Un véhicule électrique ne peut pas parcourir une distance supérieure à l'autonomie de sa batterie sans recharger.

Le choix de l'expert : la puissance de OR-Tools CP-SAT

Pour résoudre ce problème NP-difficile, nous avons utilisé CP-SAT, le solveur de programmation par contraintes de Google OR-Tools.



Fonctionnement

1. **Modélisation:** Le problème est décrit avec des variables (entières, booléennes) et des contraintes.
2. **Résolution:** Le solveur explore l'espace des solutions pour trouver une affectation optimale ou réalisable qui satisfait toutes les contraintes.

Pourquoi CP-SAT ?

Avantages

- **Flexibilité:** Permet d'ajouter facilement des contraintes complexes (temps, batterie, etc.).
- **Performance:** Fournit des solutions exactes et optimales pour des problèmes de taille moyenne.
- **Efficacité:** Intègre des techniques de recherche et de propagation de contraintes très performantes.

Limites reconnues

- Le temps de résolution peut être exponentiel dans le pire des cas, nécessitant des limites de temps pour les grandes instances.

De la théorie au code : la modélisation du problème avec CP-SAT.

Le cœur de notre solution réside dans la traduction des contraintes logistiques en un modèle mathématique rigoureux.

Variables de décision clés :

- **x[i, j, k] (booléenne)** : Le véhicule *k* se déplace-t-il de *i* vers *j* ?
- **temps_arrivee[i, k] (entière)** : Heure d'arrivée du véhicule *k* au nœud *i*.
- **charge[i, k] (entière)** : Charge du véhicule *k* en arrivant au nœud *i*.
- **batterie[i, k] (entière)** : Niveau de batterie du véhicule *k* au nœud *i* (pour le VRP Vert).

```
# Variables de décision
x = {}
for i, j, k in arcs:
  x[i, j, k] = model.NewBoolVar(f'x_{i}_{j}_{k}')
```

```
# Variables temporelles
temps_arrivee = {}
for i in nodes:
   temps_arrivee[i] = model.NewIntVar(a_i, b_i, f't_{i}')
```

Exemples de contraintes implémentées :

1. **Visite unique** : Chaque client est visité exactement une fois.
2. **Conservation du flux** : Si un véhicule entre dans un nœud, il doit en sortir.
3. **Fenêtres temporelles** : temps_arrivee[i, k] doit être dans la plage **[a_i, b_i]** (avec gestion de l'attente si arrivée précoce).
4. **Consommation batterie (VRP Vert)** : B_j = B_i - c * d_ij.

```
# Contrainte de visite unique
for client in clients:
    model.Add(sum(x[i, client, k] for i, _ in arcs if _ == client) == 1)
```

```
# Contrainte de flux
for node in nodes:
    sum_in = sum(x[i, node, k] for i, _ in arcs if _ == node)
    sum_out = sum(x[node, j, k] for _, j in arcs if _ == node)
    model.Add(sum_in == sum_out)
```

```
# Contrainte de batterie
model.Add(batterie[j] == batterie[i] - consumption[i, j]).OnlyEnforceIf(x[i, j, k])
```

La précision avant tout : des distances réelles pour des tournées réalistes

Approximation Euclidienne

Une carte est une projection 2D d'une surface sphérique. Calculer des distances euclidiennes ("à vol d'oiseau" sur un plan) est imprécis pour des coordonnées GPS.

Calcul par Haversine

Précis (distance orthodromique)

Notre approche : Nous utilisons la **formule de Haversine** pour calculer la distance orthodromique (le chemin le plus court sur la surface d'un globe) entre deux points géographiques.

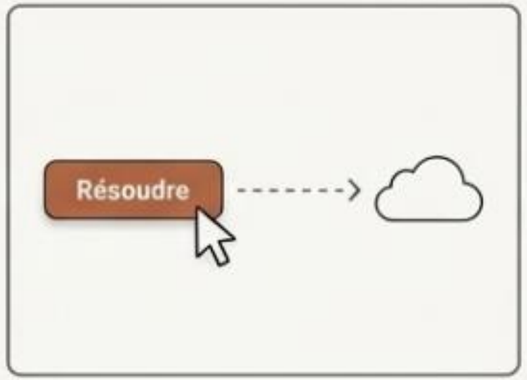
$$a = \sin^2(\frac{\Delta\varphi}{2}) + \cos\varphi_1 \cdot \cos\varphi_2 \cdot \sin^2(\frac{\Delta\lambda}{2}), \quad c = 2 \cdot \operatorname{atan2}\left(\sqrt{a}, \sqrt{1-a}\right), \quad d = R \cdot c$$

Cette rigueur garantit que les estimations de temps de trajet et de consommation d'énergie sont basées sur des distances géographiquement correctes.

Le parcours utilisateur : de la configuration à la solution en quelques clics

Configuration du Scénario

L'utilisateur définit le problème : placer le dépôt, ajouter les clients, et remplir les paramètres (véhicules, capacités...).

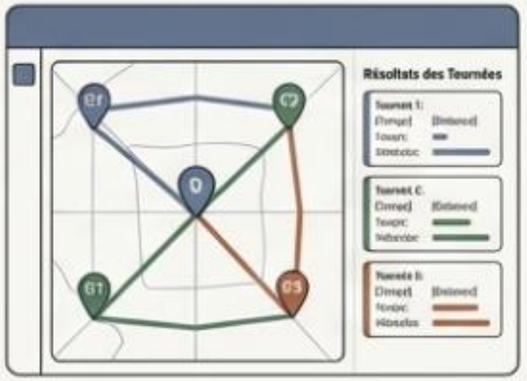


Lancement de l'Optimisation

Un clic sur "Résoudre" envoie la requête au backend pour traitement.

Suivi en Temps Réel

L'interface affiche l'état de la résolution et les solutions intermédiaires trouvées par le solveur.



Visualisation des Résultats

Les tournées optimales s'affichent instantanément sur la carte, avec un code couleur par véhicule et des informations détaillées.

Sous le capot : une architecture asynchrone pour une expérience fluide.

Le défi

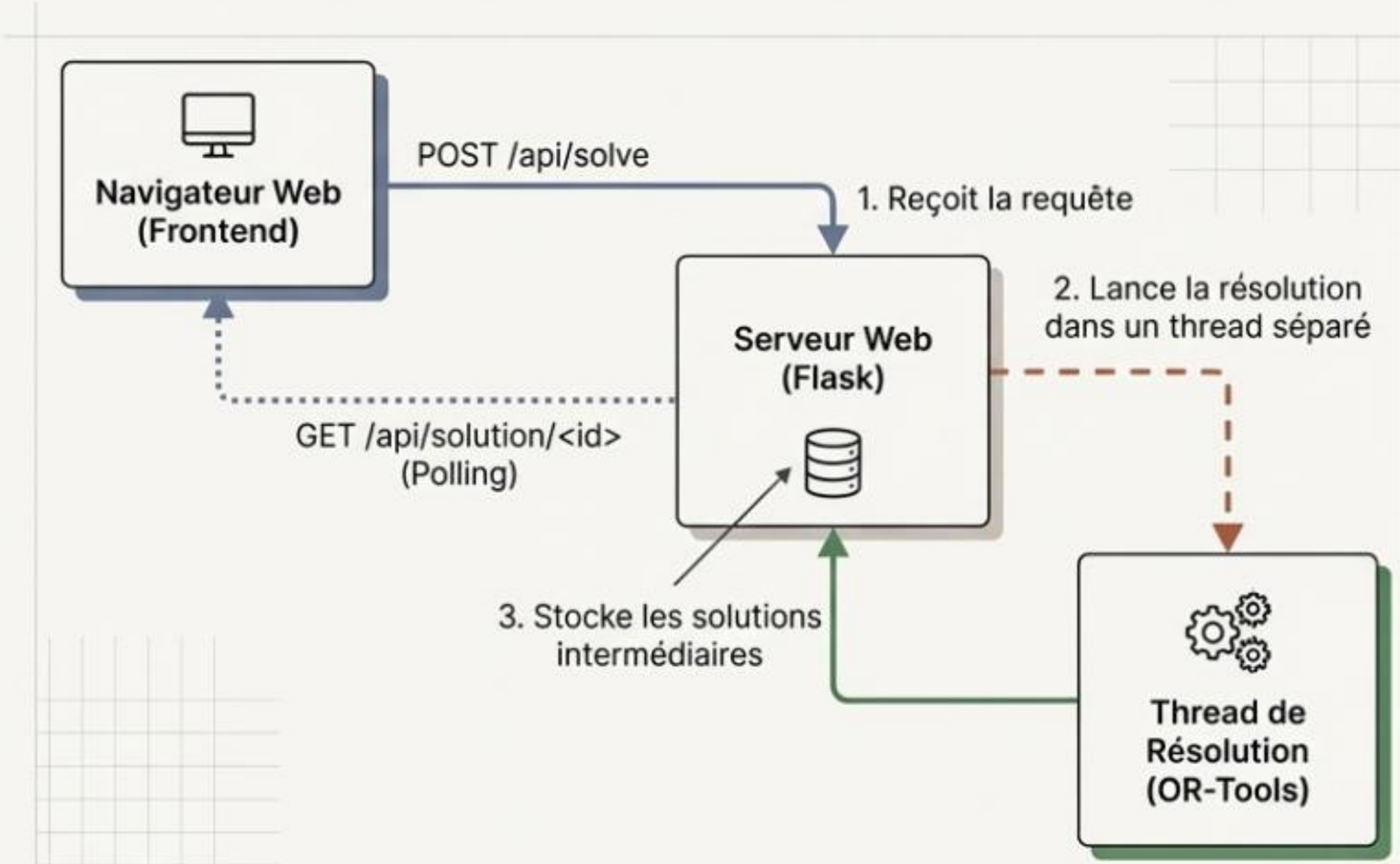
La résolution d'un VRP peut prendre du temps. Bloquer l'interface utilisateur pendant ce processus est inacceptable.

Notre solution

Une architecture client-serveur asynchrone.

Bénéfice

L'interface reste parfaitement réactive et l'utilisateur bénéficie d'un retour visuel sur la progression de l'optimisation.



Deux facettes de l'optimisation : VRP Classique vs. VRP Vert.

VRP Classique (CVRPTW)

Objectif
Minimiser la distance en respectant les contraintes de livraison standard.

Contraintes clés
- Capacité des véhicules (ex: nombre de colis).
- Fenêtres temporelles des clients.
- Temps de service à chaque arrêt.

Modélisation
Utilise les variables de charge (`charge[i, k]`) et de temps (`temps_arrivee[i, k]`).

VRP Vert (E-VRPTW)

Objectif
Intégrer la gestion de l'autonomie et de la recharge dans l'optimisation.

Contraintes additionnelles
- Autonomie maximale de la batterie.
- Consommation d'énergie proportionnelle à la distance.
- Possibilité (et nécessité) de s'arrêter aux stations de recharge.

Modélisation
Ajoute les variables de batterie (`batterie[i, k]`) et une indexation spéciale pour distinguer clients et stations.

Au-delà du tracé : des résultats clairs et exploitables.

La solution n'est pas seulement visuelle. L'interface présente un résumé détaillé et structuré pour une analyse complète des tournées.

Détail des résultats affichés

Résumé par livreur:

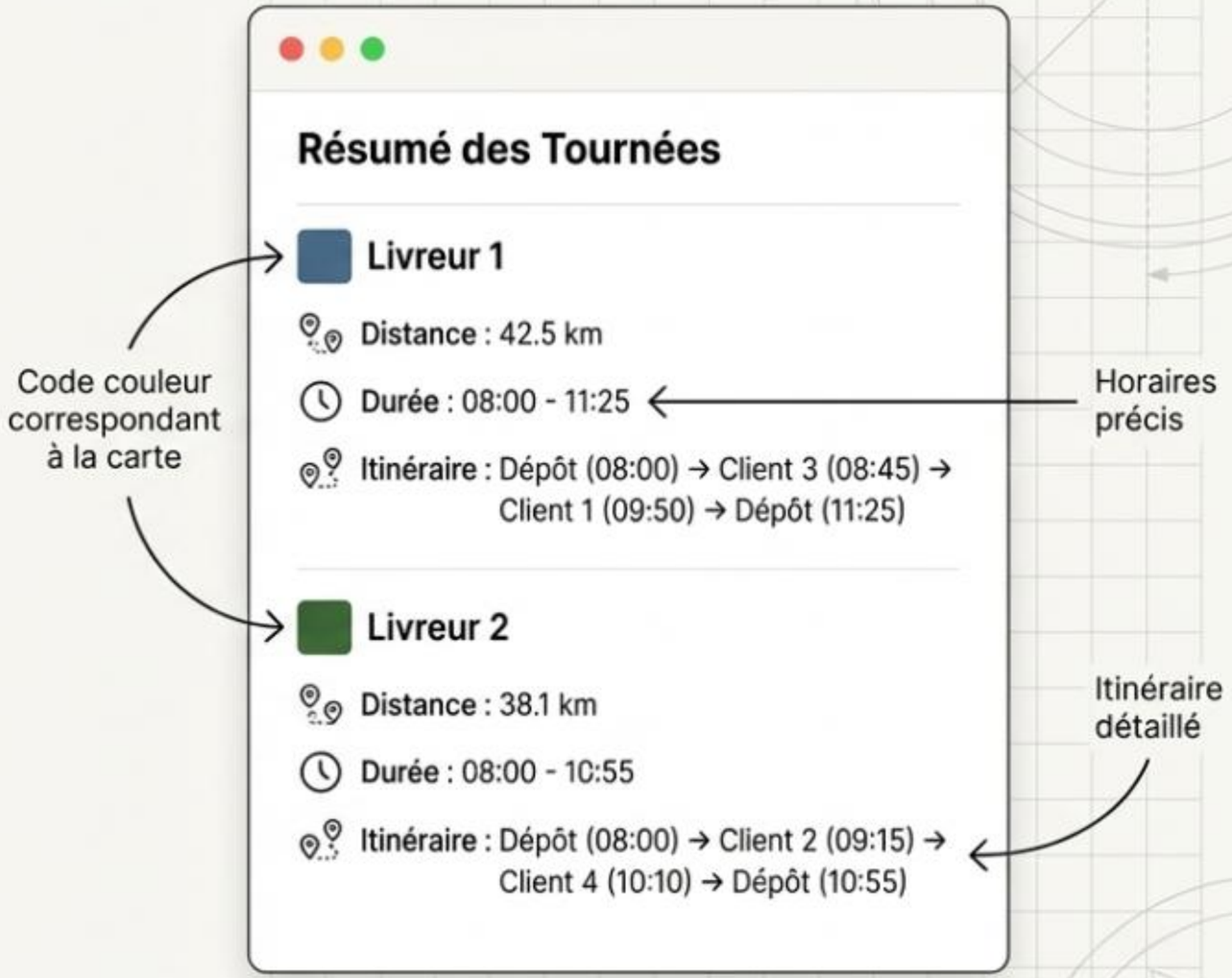
- Couleur de la tournée (correspondant au tracé sur la carte).
- Distance totale parcourue.
- Heure de départ (8h00) et heure de fin de tournée.
- Itinéraire complet avec horaires d'arrivée à chaque point.

Résumé par client:

- Heure de livraison prévue.
- Véhicule assigné à la livraison.

Le système de temps

La logique du solveur utilise des unités (1 unité = 1 minute).
Le frontend convertit ces unités en horaires HH:MM lisibles, en appliquant les règles de conversion : 1 km = 5 minutes de trajet, et 10 minutes de service par client.



La route à venir : pistes d'amélioration et perspectives d'évolution.

Notre solution basée sur un solveur exact est performante, mais pour des instances de très grande taille, des approches complémentaires peuvent être envisagées.

Heuristiques de Construction

- **Implémentation d'algorithmes** comme Clark-Wright Savings pour générer rapidement une bonne solution initiale.
- **Utilisation comme guide** pour le solveur exact afin d'accélérer la convergence.

Méta-heuristiques

- **Exploration d'approches** comme les Algorithmes Génétiques ou le Recuit Simulé.
- **Idéal pour les problèmes à très grande échelle** où une solution de haute qualité (non nécessairement optimale) est acceptable.

Optimisations Techniques

- **Parallélisation** pour explorer différentes stratégies de résolution en même temps.
- **Mise en cache** des matrices de distances pour éviter les recalculs.
- **Pré-traitement** des données pour éliminer les affectations impossibles avant la résolution.

Synthèse : un solveur VRP complet, de la théorie rigoureuse à l'implémentation robuste.

Ce projet démontre la maîtrise de trois compétences clés :

1. Rigueur Théorique et Mathématique

- Modélisation précise des contraintes du VRP avec CP-SAT.
- Utilisation de la formule de Haversine pour une justesse géographique.

2. Ingénierie Logicielle Robuste

- Architecture asynchrone client-serveur assurant une expérience utilisateur fluide.
- Code modulaire et bien structuré séparant la logique métier de la présentation.

3. Double Expertise Fonctionnelle

- Gestion des contraintes classiques (capacité, fenêtres temporelles).
- Extension élégante vers les contraintes modernes du VRP Vert (autonomie, recharge).