Documentation Détaillée du Projet CVRPTW (IA50)

Ce document fournit une explication approfondie de l'architecture, des algorithmes et du code source du projet de résolution du problème de tournée de véhicules avec capacités et fenêtres de temps (CVRPTW). Il est conçu pour servir de référence technique pour la soutenance.

1. Vue d'ensemble du Projet

Le projet implémente un solveur hybride pour le **CVRPTW (Capacitated Vehicle Routing Problem with Time Windows)**. L'objectif est de trouver un ensemble d'itinéraires optimaux pour une flotte de véhicules afin de desservir un ensemble de clients, en respectant les contraintes suivantes : * **Capacité** : Chaque véhicule a une capacité de chargement maximale. * **Fenêtres de Temps** : Chaque client doit être servi dans un intervalle de temps spécifique `[ready_time, due_date]`. * **Flotte** : Le nombre de véhicules n'est pas fixé a priori mais doit être minimisé implicitement en optimisant la distance totale.

L'approche choisie est une **Métaheuristique Hybride** en trois étapes : 1. **ACO (Ant Colony Optimization)** : Construction de solutions initiales de bonne qualité. 2. **GA (Genetic Algorithm)** : Diversification et exploration globale pour améliorer les solutions. 3. **Tabu Search** : Intensification et recherche locale pour raffiner la meilleure solution trouvée.

2. Structure du Code

Le projet est structuré de manière modulaire dans le dossier `src/` :

```
src/
├── core/           # Composants fondamentaux (Modèles, Solution)
│   ├── models.py   # Définition des Noeuds, Routes et Instances
│   ├── solution.py # Classe wrapper pour une Solution complète
│   └── interfaces.py # Interface abstraite pour les solveurs
├── solvers/        # Implémentation des algorithmes
│   ├── aco.py      # Algorithme de Colonies de Fourmis
```

```
│   ├── ga.py       # Algorithme Génétique
│   ├── tabu.py     # Recherche Tabou
│   └── hybrid.py   # Orchestrateur de la stratégie hybride
├── utils/          # Utilitaires
│   ├── logger.py   # Gestion des logs
│   └── plotting.py # Visualisation (Graphes, Gantt)
├── config.py       # Classes de configuration (Dataclasses)
└── cli.py          # Interface en ligne de commande
app.py              # Application Web (Streamlit)
```

3. Analyse Détaillée des Composants

3.1. Core (`src/core`)

`models.py`

Ce fichier définit les structures de données de base. * `Node` : Représente un point sur la carte (Client ou Dépôt). * Attributs : `id`, `x`, `y`, `demand` (demande), `ready_time`, `due_date`, `service_time`. * Méthode `distance_to(other)` : Calcule la distance euclidienne. * `Route` : Représente le trajet d'un seul véhicule. * Attributs : `nodes` (liste de `Node`), `total_distance`, `total_load`, `schedule` (horaires). * Méthode `is_feasible(...)` : Vérifie si la route respecte la capacité et les fenêtres de temps. * Méthode `calculate_metrics(...)` : Recalcule la distance et le planning (arrivée, attente, début service, départ). * `CVRPTWInstance` : Générateur d'instances aléatoires. * Génère un dépôt au centre et des clients avec des coordonnées, demandes et fenêtres de temps aléatoires mais faisables.

`solution.py`

- `Solution` : Encapsule une solution complète (ensemble de routes).
  - Attributs : `routes`, `total_distance`, `total_wait`, `is_feasible`.
  - Méthode `fitness()` : Retourne la distance totale (ou infini si infaisable), utilisée pour comparer les solutions.

`interfaces.py`

- `SolverStrategy` : Classe abstraite (ABC) définissant la méthode `solve()`. Tous les algorithmes héritent de cette classe, assurant une interchangeabilité.

3.2. Solvers (`src/solvers`)

`aco.py` (Ant Colony Optimization)

- **Principe** : Simule des fourmis qui construisent des routes en se basant sur des phéromones (mémoire collective) et une heuristique (distance inverse).
- **Fonctionnement** :
  - Initialise une matrice de phéromones.
  - À chaque itération, plusieurs fourmis construisent des solutions complètes nœud par nœud.
  - **Choix probabiliste** : La probabilité de passer de `i` à `j` dépend de la phéromone `tau[i][j]` et de la visibilité `eta = 1/distance`.
  - **Mise à jour** : Les phéromones s'évaporent (pour éviter la convergence prématurée) et sont renforcées sur les arcs des meilleures solutions trouvées.
- **Rôle dans l'hybride** : Fournir une population initiale de solutions prometteuses pour l'Algorithme Génétique.

`ga.py` (Genetic Algorithm)

- **Principe** : Évolution d'une population de solutions inspirée de la sélection naturelle.
- **Opérateurs** :
  - **Sélection** : Tournoi (on prend les meilleurs parmi un sous-groupe aléatoire).
  - **Crossover (Croisement)** : `Ordered Crossover`. Combine les routes de deux parents pour créer un enfant, en préservant l'ordre relatif des clients pour minimiser les violations.
  - **Mutation** : Échange aléatoire de deux clients (Swap) pour introduire de la diversité.
  - **Élitisme** : La meilleure solution est toujours conservée d'une génération à l'autre.
- **Rôle dans l'hybride** : Explorer l'espace de recherche globalement et combiner les traits des bonnes solutions ACO.

`tabu.py` (Tabu Search)

- **Principe** : Recherche locale qui explore le voisinage de la solution courante en acceptant parfois des dégradations pour échapper aux optimums locaux, tout en interdisant (Tabou) de revenir sur des mouvements récents.

- **Voisinage** :
  - **Relocate** : Déplacer un client d'une route à une autre.
  - **Swap** : Échanger deux clients entre deux routes ou au sein de la même route.
- **Liste Tabou** : Mémorise les mouvements récents pour les interdire temporairement.
- **Rôle dans l'hybride** : Raffiner la meilleure solution trouvée par le GA (étape de polissage final).

`hybrid.py` (Orchestrateur)

- Coordonne l'exécution séquentielle : `ACO -> GA -> Tabu`.
- Passe les résultats d'une étape comme entrée de la suivante.
- Collecte l'historique de convergence pour la visualisation.

3.3. Configuration (`src/config.py`)

Utilise des `dataclasses` pour typer et structurer les hyperparamètres : * `ACOConfig` : Nombre de fourmis, alpha (poids phéromone), beta (poids heuristique), évaporation. * `GAConfig` : Taille population, nombre générations, taux mutation. * `TabuConfig` : Nombre d'itérations, taille voisinage, durée tabou.

3.4. Visualisation (`src/utils/plotting.py`)

- `plot_solution` : Affiche la carte avec le dépôt, les clients et les itinéraires colorés.
- `plot_convergence` : Trace l'évolution du coût (distance) à travers les 3 étapes (ACO, GA, Tabu).
- `plot_gantt` : Affiche un diagramme de Gantt montrant l'emploi du temps de chaque véhicule (trajet, attente, service).

4. Exécution

Interface Web (Streamlit)

Le fichier `app.py` lance une interface utilisateur interactive. * Permet de configurer les paramètres de l'instance (nombre clients, capacité...) et des algorithmes. * Visualise les résultats en temps réel avec des graphiques interactifs. * Commande : `streamlit run app.py`

Interface Ligne de Commande (CLI)

Le fichier `src/cli.py` permet une exécution rapide sans interface graphique. * Commande : `python -m src.cli --customers 50 --ants 20 ...`

5. Points Clés pour la Soutenance

1. **Justification de l'Hybridation** :
    - ACO est excellent pour l'exploration initiale mais lent à converger finement.
    - GA est robuste pour la recherche globale mais peut stagner.
    - Tabu est très efficace pour l'optimisation locale mais sensible à l'initialisation.
    - La combinaison tire parti des forces de chacun.
2. **Gestion des Contraintes** :
    - Les contraintes (Capacité, Temps) sont vérifiées à chaque insertion ou modification de route (`is_feasible`).
    - Les solutions infaisables sont fortement pénalisées ou rejetées.
3. **Complexité** :
    - Le problème est NP-Difficile. L'approche métaheuristique ne garantit pas l'optimal absolu mais vise une très bonne solution en un temps raisonnable.

Ce code démontre une maîtrise de la programmation orientée objet en Python, des algorithmes d'optimisation avancés et de l'intégration logicielle complète (Backend + Frontend).