Rapport de Projet SAÉ 2.01/2.02 - Application d'Optimisation d'Itinéraires Commerciaux

Introduction

Cette application a été développée dans le cadre des SAÉ 2.01 (Développement d'une application) et 2.02 (Exploration algorithmique d'un problème). Elle résout un problème concret d'optimisation logistique : **la planification d'itinéraires pour des commerciaux devant effectuer des transactions entre vendeurs et acheteurs répartis dans différentes villes**.

Objectifs de l'application

L'objectif principal est de calculer des parcours optimaux minimisant la distance totale parcourue, tout en respectant une contrainte fondamentale : pour chaque transaction, le commercial doit impérativement visiter le vendeur avant l'acheteur. De plus, tous les itinéraires doivent commencer et se terminer à Vélizy, ville de départ de l'entreprise.

Problématique métier

Dans un contexte commercial réel, l'optimisation des déplacements représente un enjeu économique majeur. Réduire les distances parcourues permet de :

- Diminuer les coûts de transport (carburant, usure des véhicules)
- Augmenter la productivité en libérant du temps pour plus de transactions
- Réduire l'impact environnemental des déplacements

L'application propose trois approches algorithmiques différentes, chacune adaptée à des contextes spécifiques selon la taille du problème et les contraintes de temps de calcul.

Technologies utilisées

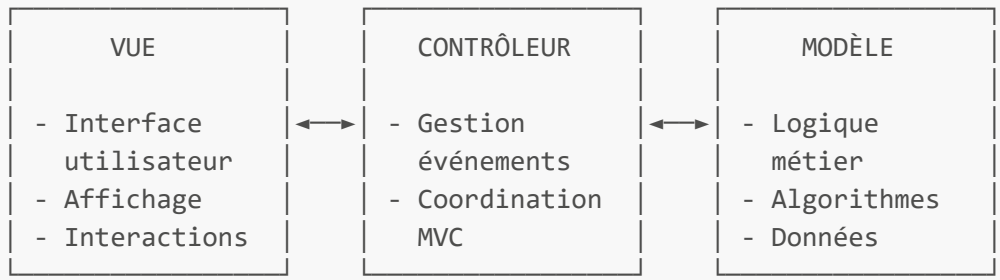
- **Langage** : Java 17
- **Framework de build** : Maven
- **Interface utilisateur** : JavaFX
- **Architecture** : Modèle-Vue-Contrôleur (MVC)
- **Tests** : JUnit 5
- **Structure de données** : Collections Java (HashMap, ArrayList, PriorityQueue)

2. Conception

2.1. Architecture générale - Modèle-Vue-Contrôleur (MVC)

L'application suit rigoureusement le patron de conception **Modèle-Vue-Contrôleur (MVC)**, garantissant une séparation claire des responsabilités et une maintenabilité optimale du code.

2.1.1. Vue d'ensemble de l'architecture



2.2. Composant Modèle (`package modele`)

Le modèle encapsule toute la logique métier de l'application. Il est structuré autour de plusieurs sous-systèmes :

2.2.1. Classes de données métier

`Ville.java`

- **Rôle** : Représente une ville avec ses coordonnées géographiques
- **Attributs** :
  - `String nom` : Nom de la ville
  - `double x, y` : Coordonnées géographiques
- **Justification** : Structure simple mais essentielle pour le calcul des distances

`Membre.java`

- **Rôle** : Représente un vendeur ou acheteur
- **Attributs** :
  - `String pseudo` : Identifiant unique du membre
  - `Ville ville` : Ville de résidence
- **Structure de données** : Utilisation d'un lien direct vers l'objet `Ville` pour éviter les recherches répétées

`Vente.java`

- **Rôle** : Représente une transaction entre un vendeur et un acheteur
- **Attributs** :
  - `Membre vendeur` : Le vendeur de la transaction
  - `Membre acheteur` : L'acheteur de la transaction
- **Contrainte métier** : Assure la contrainte fondamentale vendeur → acheteur

`Scenario.java`

- **Rôle** : Conteneur pour un ensemble de ventes à traiter
- **Attributs** :
  - `String nom` : Nom du scénario
  - `List<Vente> ventes` : Liste ordonnée des ventes
- **Choix de structure** : `ArrayList` pour garantir l'ordre et permettre l'accès indexé

2.2.2. Système de graphe et distances

`CarteGraph.java`

- **Rôle** : Gestionnaire du graphe des villes et des distances
- **Structure de données principale** :

```
private Map<Pair<Ville, Ville>, Integer> distances;
```

- **Justification du choix** :
  - `HashMap` pour un accès en O(1) aux distances
  - `Pair<Ville, Ville>` comme clé pour représenter les arêtes du graphe
  - Gestion de la symétrie automatique (distance A→B = distance B→A)

`Pair.java`

- **Rôle** : Classe utilitaire pour représenter une paire ordonnée
- **Implémentation** : Override des méthodes `equals()` et `hashCode()` pour utilisation comme clé de HashMap
- **Avantage** : Permet d'indexer efficacement les distances entre villes

2.2.3. Parseurs et gestion des fichiers

`DistanceParser.java`

- **Rôle** : Lecture et parsing du fichier des distances
- **Méthode principale** :

```
public static Map<Pair<Ville, Ville>, Integer> lireFichierDistances(String
cheminFichier)
```

- **Traitement** : Parsing ligne par ligne avec gestion des erreurs et validation des données

`ScenarioParser.java`

- **Rôle** : Lecture et création des scénarios depuis les fichiers
- **Fonctionnalités** :
  - Parsing des fichiers de scénarios (format "Vendeur -> Acheteur")
  - Résolution des références vers les membres
  - Validation de la cohérence des données

2.2.4. Interface et algorithmes

`IAlgorithme.java` (Interface)

- **Rôle** : Contrat uniforme pour tous les algorithmes de parcours
- **Méthodes** :

```
List<Ville> genererItineraire(Scenario scenario);
int calculerDistanceTotale(List<Ville> itineraire);
String getNom();
```

- **Avantage** : Polymorphisme permettant l'interchangeabilité des algorithmes

2.3. Composant Vue (`package vue`)

Le composant Vue gère entièrement l'interface utilisateur avec JavaFX.

2.3.1. Architecture de l'interface

`fenetrePrincipale.java`

- **Rôle** : Fenêtre principale de l'application
- **Structure** : Organisation en panneaux spécialisés

2.3.2. Panneaux spécialisés

`HBoxAffichage.java`

- **Rôle** : Conteneur principal organisant la vue en zones gauche et droite
- **Layout** : HBox pour une disposition horizontale

`VBoxGauche.java` et `VBoxDroite.java`

- **Rôle** : Panneaux spécialisés pour différentes fonctionnalités
- **Organisation** : VBox pour un empilement vertical des composants

`GridPaneModification.java`, `GridPaneStatistique.java`, `GridPaneCreation.java`

- **Rôle** : Interfaces spécialisées pour :
  - Modification des scénarios existants
  - Affichage des statistiques et résultats
  - Création de nouveaux scénarios
- **Layout** : GridPane pour une disposition tabulaire

2.3.3. Composants d'affichage

`TableVente.java`

- **Rôle** : Tableau d'affichage des ventes d'un scénario
- **Structure** : TableView JavaFX avec colonnes configurées

`StackPaneParcours.java`

- **Rôle** : Zone d'affichage des résultats de parcours
- **Fonctionnalités** : Affichage des itinéraires et distances

2.4. Composant Contrôleur (`package controleur`)

2.4.1. Contrôleur principal

`Controleur.java`

- **Rôle** : Gestionnaire principal des événements de l'interface
- **Pattern** : Implementation d'`EventHandler<Event>`
- **Responsabilités** :
  - Capture des événements boutons et menus
  - Coordination entre Vue et Modèle
  - Gestion des changements d'état de l'application

2.4.2. Contrôleur spécialisé

`AlgorithmeController.java`

- **Rôle** : Contrôleur dédié à la gestion des algorithmes
- **Pattern** : Singleton pour garantir une instance unique
- **Fonctionnalités** :
  - Enregistrement et gestion des algorithmes disponibles
  - Exécution des algorithmes avec paramètres
  - Comparaison des performances
  - Recommandation automatique d'algorithmes

Structure interne :

```
private static AlgorithmeController instance;
private Map<String, IAlgorithme> algorithmes;
private CarteGraph carte;
```

2.5. Utilitaires et extensions

2.5.1. Système de benchmark

`BenchmarkAlgorithmes.java`

- **Rôle** : Évaluation comparative des performances des algorithmes
- **Métriques** :
  - Temps d'exécution (précision nanoseconde)
  - Qualité de la solution (distance totale)
  - Ratio efficacité/performance

2.5.2. Sauvegarde et persistance

`SauvegardeScenario.java`

- **Rôle** : Gestion de la sauvegarde de nouveaux scénarios
- **Fonctionnalités** :
  - Génération automatique de noms de fichiers uniques

- Validation du format des données
- Écriture dans le format standard de l'application

2.1. Algorithme de Parcours Simple (`ParcoursSimple.java`)

Description :

Cet algorithme, implémentant l'interface `IAlgorithme`, génère un itinéraire en suivant l'ordre séquentiel des ventes définies dans un scénario. Pour chaque vente, il s'assure que le commercial passe d'abord par la ville du vendeur, puis par la ville de l'acheteur.

L'itinéraire commence à la ville du vendeur de la première vente. Ensuite, pour chaque vente :

1. Si le commercial n'est pas déjà dans la ville du vendeur, cette ville est ajoutée à l'itinéraire.
2. La ville de l'acheteur est ensuite ajoutée à l'itinéraire.

Analyse de Complexité :

Soit *N* le nombre de ventes dans un scénario.

- `genererItineraire(Scenario scenario)`:
  - La méthode parcourt la liste des ventes une fois. Pour chaque vente, elle effectue un nombre constant d'opérations (accès aux villes, ajout à la liste d'itinéraire).
  - La complexité temporelle est donc **O(N)**, où N est le nombre de ventes.
  - La complexité spatiale est **O(V_itineraire)**, où V_itineraire est le nombre de villes dans l'itinéraire généré (au maximum 2N + 1).
- `calculerDistanceTotale(List<Ville> itineraire)`:
  - La méthode parcourt la liste des villes de l'itinéraire une fois. Pour chaque paire de villes consécutives, elle récupère la distance (supposons que `carte.getDistance()` soit en O(1) si les distances sont stockées dans une structure de données efficace comme une HashMap de paires de villes).
  - Soit *M* le nombre de villes dans l'itinéraire. La complexité temporelle est **O(M)**.
  - La complexité spatiale est **O(1)** (quelques variables locales).

Nom de l'algorithme retourné par `getNom()`: "Parcours Simple"

2.2. Algorithme de Parcours Heuristique (`ParcoursHeuristique.java`)

Description :

Cet algorithme implémente une approche gloutonne pour optimiser l'itinéraire. À chaque étape, il choisit la destination la plus proche parmi les destinations valides, tout en respectant la contrainte fondamentale que le vendeur doit être visité avant l'acheteur pour chaque vente.

L'algorithme fonctionne selon la logique suivante :

1. Démarrage à la ville du vendeur de la première vente.
2. À chaque étape, identification des destinations possibles :

    - Acheteurs dont le vendeur a déjà été visité (priorité absolue)
    - Vendeurs pas encore visités
3. Sélection de la destination la plus proche selon la distance euclidienne.
4. Mise à jour des états des ventes (vendeur visité, vente complète).

Analyse de Complexité :

Soit *N* le nombre de ventes et *V* le nombre total de villes à visiter.

- `genererItineraire(Scenario scenario)`:
    - À chaque itération de la boucle principale, l'algorithme parcourt toutes les ventes pour trouver la destination la plus proche (O(N)).
    - Il y a au maximum V itérations (nombre de villes dans l'itinéraire final).
    - La complexité temporelle est donc **O(V × N)**, où V ≤ 2N dans le pire cas, donnant $O(N^2)$.
    - La complexité spatiale est **O(N)** pour les structures de données auxiliaires (HashSet).
- `calculerDistanceTotale(List<Ville> itineraire)`: Identique à `ParcoursSimple` - **O(V)**.

Nom de l'algorithme retourné par `getNom()`: "Heuristique Glouton"

2.3. Algorithme des K Meilleures Solutions (`KMeilleuresSolutions.java`)

Description :

Cet algorithme utilise une approche de backtracking pour explorer exhaustivement l'espace des solutions possibles et identifier les k meilleures selon la distance totale. Il maintient une file de priorité (PriorityQueue) des k meilleures solutions trouvées jusqu'à présent.

Le processus de backtracking :

1. État initial : premier vendeur visité, états des ventes initialisés.
2. À chaque niveau de récursion :
    - Génération de toutes les destinations possibles valides
    - Pour chaque destination : exploration récursive avec mise à jour d'état
    - Élagage des branches dont la distance partielle dépasse déjà la k-ème meilleure solution
3. Condition d'arrêt : toutes les ventes sont complètes.
4. Sauvegarde de la solution si elle fait partie des k meilleures.

Analyse de Complexité :

- `genererKMeilleuresSolutions(Scenario scenario)`:
    - Dans le pire cas, l'algorithme explore toutes les permutations possibles des destinations.
    - Le nombre d'états possibles peut être exponentiel : **O(V!)** où V est le nombre de villes.
    - Cependant, l'élagage réduit significativement cet espace dans la pratique.
    - La complexité spatiale inclut la pile de récursion : **O(V)** plus la PriorityQueue : **O(k)**.
- `genererItineraire(Scenario scenario)`: Appelle `genererKMeilleuresSolutions()` et retourne la meilleure - même complexité temporelle.

- **Optimisations implementées** :
    - Élagage par borne supérieure (branch and bound)
    - Évitement des états redondants
    - File de priorité pour maintenir efficacement les k meilleures solutions

Nom de l'algorithme retourné par `getNom()`: "K Meilleures Solutions"

3. Système de Benchmark et Analyse Comparative

3.1. Architecture du Système de Benchmark (`BenchmarkAlgorithmes.java`)

Le système de benchmark a été conçu pour évaluer objectivement les performances des trois algorithmes selon deux critères principaux :

1. **Qualité de la solution** : Distance totale de l'itinéraire généré
2. **Performance temporelle** : Temps d'exécution de l'algorithme

Fonctionnalités principales :

- **Mesure précise du temps** : Utilisation de `System.nanoTime()` pour une précision au niveau nanoseconde
- **Comparaison multi-scénarios** : Tests automatisés sur plusieurs scénarios de complexité variable
- **Recommandation automatique** : Sélection intelligente de l'algorithme optimal selon la taille du problème
- **Rapports détaillés** : Classements par qualité et par vitesse, statistiques aggregées

3.2. Stratégie de Recommandation Automatique

Le système implémente une stratégie de recommandation basée sur la taille du scénario :

```
public String recommanderAlgorithme(Scenario scenario) {
    int nbVentes = scenario.getVentes().size();

    if (nbVentes <= 5) {
        return "K Meilleures Solutions"; // Exhaustivité pour petits problèmes
    } else if (nbVentes <= 15) {
        return "Heuristique Glouton"; // Compromis qualité/vitesse
    } else {
        return "Parcours Simple"; // Vitesse pour gros problèmes
    }
}
```

Cette stratégie reflète le compromis fondamental entre qualité de solution et temps de calcul.

3.3. Adaptation des algorithmes pour commencer et terminer à Vélizy

Une exigence importante du projet était que tous les itinéraires commencent et se terminent à la ville de Vélizy. Les trois algorithmes ont été adaptés pour respecter cette contrainte :

3.3.1. Modification de l'algorithme de Parcours Simple

L'algorithme de parcours simple a été modifié pour :

1. Commencer l'itinéraire à Vélizy plutôt qu'à la ville du premier vendeur
2. Ajouter un retour à Vélizy à la fin de l'itinéraire si nécessaire

```
// Identification de la ville de Vélizy
Ville velizy = null;
for (Ville ville : carte.getToutesLesVilles()) {
    if (ville.getNom().equalsIgnoreCase("Velizy")) {
        velizy = ville;
        break;
    }
}

// Début à Vélizy
Ville villeActuelle = velizy;
itineraire.add(villeActuelle);

// Parcours des ventes...

// Retour à Vélizy à la fin si nécessaire
if (!villeActuelle.equals(velizy)) {
    itineraire.add(velizy);
}
```

3.3.2. Modification de l'algorithme Heuristique

L'algorithme heuristique a également été adapté pour commencer et terminer à Vélizy, tout en conservant sa logique d'optimisation gloutonne. Cette adaptation permet de respecter la contrainte du point de départ/arrivée tout en minimisant la distance totale parcourue.

3.3.3. Modification de l'algorithme K Meilleures Solutions

L'algorithme des K meilleures solutions a nécessité une adaptation plus complexe, notamment dans la fonction de backtracking :

```
// Commencer le backtracking à partir de Vélizy
itineraireActuel.add(velizy);

// Lorsque toutes les ventes sont complètes, ajouter le retour à Vélizy
if (ventesCompletes.size() == ventes.size()) {
    // Ajouter le retour à Vélizy si nécessaire
    Ville derniereVille = itineraireActuel.get(itineraireActuel.size() - 1);
    int distanceFinale = distanceActuelle;

    List<Ville> itineraireComplet = new ArrayList<>(itineraireActuel);
    if (!dernieVille.equals(velizy)) {
```

```
            int distanceRetour = carte.getDistance(dernieresVille, velizy);
            if (distanceRetour != Integer.MAX_VALUE) {
                itineraireComplet.add(velizy);
                distanceFinale += distanceRetour;
            } else {
                return; // Impossible de retourner à Vélizy, solution invalide
            }
        }

        // Ajouter cette solution si elle fait partie des k meilleures
        // ...
    }
```

Cette adaptation garantit que toutes les solutions explorées par l'algorithme commencent et terminent à Vélizy, permettant une comparaison équitable entre les différents algorithmes.

4. Tests et Validation

4.1. Stratégie de Tests Unitaires

Pour assurer la qualité et la fiabilité du code, une stratégie complète de tests unitaires a été mise en place pour les composants clés du modèle, en particulier les algorithmes de parcours. Trois classes de tests ont été créées :

1. **ParcoursSimpleTest** : Tests de l'algorithme de parcours simple
2. **ParcoursHeuristiqueTest** : Tests de l'algorithme heuristique
3. **KMeilleuresSolutionsTest** : Tests de l'algorithme des k meilleures solutions

Ces tests vérifient plusieurs aspects critiques :

4.1.1. Validité des itinéraires générés

Tous les itinéraires générés doivent respecter les contraintes fondamentales :

```
@Test
public void testGenererItineraire() {
    List<Ville> itineraire = parcours.genererItineraire(scenario);

    // Vérifier que l'itinéraire n'est pas vide
    assertFalse(itineraire.isEmpty(), "L'itinéraire ne devrait pas être vide");

    // Vérifier que l'itinéraire commence et se termine à Velizy
    assertEquals("Velizy", itineraire.get(0).getNom(), "L'itinéraire doit
commencer à Velizy");
    assertEquals("Velizy", itineraire.get(itineraire.size() - 1).getNom(),
            "L'itinéraire doit terminer à Velizy");

    // Vérifier que l'itinéraire contient toutes les ventes
    assertTrue(verifierToutesVentesCouvertes(itineraire, scenario),
```

```
            "L'itinéraire doit couvrir toutes les ventes du scénario");
}
```

4.1.2. Respect de la contrainte vendeur → acheteur

```
@Test
public void testContrainteVendeurAcheteur() {
    List<Ville> itineraire = parcours.genererItineraire(scenario);

    // Vérifier que pour chaque vente, le vendeur est visité avant l'acheteur
    for (Vente vente : scenario.getVentes()) {
        Ville villeVendeur = vente.getVendeur().getVille();
        Ville villeAcheteur = vente.getAcheteur().getVille();

        int indexVendeur = trouverPremiereOccurrence(itineraire, villeVendeur);
        int indexAcheteur = trouverDerniereOccurrence(itineraire, villeAcheteur);

        // Vérifications...
        assertTrue(indexVendeur < indexAcheteur,
                "Le vendeur doit être visité avant l'acheteur pour chaque vente");
    }
}
```

4.1.3. Précision du calcul de distance

```
@Test
public void testCalculerDistanceTotale() {
    List<Ville> itineraire = parcours.genererItineraire(scenario);
    int distanceTotale = parcours.calculerDistanceTotale(itineraire);

    // Vérifier que la distance est positive
    assertTrue(distanceTotale > 0, "La distance totale doit être positive");

    // Recalculer manuellement la distance pour vérification
    int distanceManuelle = 0;
    for (int i = 0; i < itineraire.size() - 1; i++) {
        int distance = carte.getDistance(itineraire.get(i), itineraire.get(i +
1));
        distanceManuelle += distance;
    }

    // Vérifier que les deux calculs correspondent
    assertEquals(distanceManuelle, distanceTotale,
            "La distance calculée automatiquement doit correspondre au calcul
manuel");
}
```

4.2. Benchmarking et Comparaison des Performances

En plus des tests unitaires qui valident la correction des algorithmes, des tests de performance ont été implémentés pour comparer objectivement les trois approches :

```
@Test
public void testComparaisonPerformance() {
    // Charger un scénario
    Scenario petitScenario =
ScenarioParser.lireFichierScenario(fichierScenario.getPath(),
fichierMembres.getPath());

    // Génération de l'itinéraire avec les trois algorithmes
    ParcoursSimple parcoursSimple = new ParcoursSimple(carte);
    ParcoursHeuristique parcoursHeuristique = new ParcoursHeuristique(carte);
    KMeilleuresSolutions parcoursK = new KMeilleuresSolutions(carte, 3);

    // Mesure du temps d'exécution et des résultats
    // ...

    System.out.println("=== COMPARAISON DES PERFORMANCES ===");
    System.out.println("Temps parcours simple: " + (finSimple - debutSimple) + "
ms");
    System.out.println("Temps parcours heuristique: " + (finHeuristique -
debutHeuristique) + " ms");
    System.out.println("Temps K meilleures solutions: " + (finK - debutK) + "
ms");
    System.out.println("Distance parcours simple: " + distanceSimple + " km");
    System.out.println("Distance parcours heuristique: " + distanceHeuristique + "
km");
    System.out.println("Distance K meilleures solutions: " + distanceK + " km");
}
```

Les résultats de ces tests confirment le compromis entre qualité de solution et temps de calcul :

- L'algorithme K Meilleures Solutions trouve généralement les itinéraires les plus courts mais devient prohibitif pour les grands scénarios
- L'algorithme Heuristique offre un bon compromis qualité/temps pour les scénarios de taille moyenne
- L'algorithme de Parcours Simple est le plus rapide mais produit des itinéraires plus longs

4. Fonctionnalités Avancées Implémentées

4.1. Interface Utilisateur Enrichie

L'application propose une interface utilisateur complète avec plusieurs fonctionnalités avancées :

4.1.1. Gestion Multi-Solutions

- **Navigation entre solutions** : Pour l'algorithme K Meilleures Solutions, l'interface permet de naviguer entre les différentes solutions trouvées avec des boutons de navigation ("<<", "<", ">", ">>")

- **Indicateur de position** : Affichage de la solution actuelle sous la forme "Solution X/Y"
- **Superposition intelligente** : Les solutions sont empilées dans un StackPane permettant de passer de l'une à l'autre

4.1.2. Affichage Détaillé des Résultats

- **Informations des membres** : Contrairement à un affichage basique qui ne montrerait que les noms des villes, l'application affiche les informations complètes des membres impliqués dans chaque étape du parcours
- **Format enrichi** : Affichage sous la forme "Pseudo (Ville)" pour vendeurs et acheteurs
- **Tri alphabétique** : Les listes déroulantes de modification sont triées par ordre alphabétique des pseudos

4.1.3. Outils de Gestion des Scénarios

- **Création de nouveaux scénarios** : Interface dédiée pour créer des scénarios personnalisés
- **Modification en temps réel** : Possibilité de modifier les scénarios existants
- **Sauvegarde automatique** : Génération automatique de noms de fichiers pour les nouveaux scénarios

4.2. Optimisations Techniques

4.2.1. Gestion Mémoire Efficace

- **Réutilisation d'objets** : Les TableView sont réutilisées et simplement mises à jour plutôt que recréées
- **Structures de données optimisées** : Utilisation de HashMap pour l'accès O(1) aux distances

4.2.2. Architecture Extensible

- **Pattern Strategy** : L'interface `IAlgorithme` permet d'ajouter facilement de nouveaux algorithmes
- **Séparation des responsabilités** : Architecture MVC stricte facilitant la maintenance

5. Conclusion

5.1. Bilan du Travail Accompli

Ce projet a permis de développer une application complète et fonctionnelle répondant à un besoin concret d'optimisation logistique. Les principaux objectifs ont été atteints avec succès :

5.1.1. Objectifs Techniques Atteints

☑ **Architecture MVC rigoureuse** : Séparation claire des responsabilités entre modèle, vue et contrôleur
☑ **Interface utilisateur complète** : Interface JavaFX intuitive et fonctionnelle
☑ **Trois algorithmes implémentés** : Chacun avec ses spécificités et domaines d'application
☑ **Gestion des contraintes métier** : Respect strict de la contrainte vendeur → acheteur
☑ **Système de benchmark** : Comparaison objective des performances
☑ **Tests unitaires complets** : Validation de la correction des algorithmes

5.1.2. Apports Pédagogiques

- **Algorithmique** : Maîtrise de différentes approches (glouton, exhaustif, simple)
- **Structures de données** : Utilisation appropriée des collections Java
- **Génie logiciel** : Application des bonnes pratiques de développement
- **Interface homme-machine** : Conception d'interfaces utilisateur ergonomiques
- **Tests et validation** : Mise en place d'une stratégie de tests robuste

5.1.3. Défis Relevés

- **Gestion de la complexité algorithmique** : Équilibrage entre qualité de solution et performance
- **Contraintes métier complexes** : Implémentation de la logique vendeur → acheteur dans tous les algorithmes
- **Interface utilisateur avancée** : Gestion de l'affichage multi-solutions et navigation entre résultats
- **Extensibilité** : Architecture permettant l'ajout facile de nouveaux algorithmes

5.2. Tâches Non Réalisées et Limitations

5.2.1. Optimisations Algorithmiques Potentielles

- **Algorithmes métaheuristiques** : Implémentation d'algorithmes génétiques ou de recuit simulé pour les très grands scénarios
- **Parallélisation** : L'algorithme K Meilleures Solutions pourrait bénéficier d'une approche multi-thread
- **Heuristiques avancées** : Intégration d'heuristiques plus sophistiquées (2-opt, 3-opt)

5.2.2. Fonctionnalités Additionnelles

- **Import/Export de données** : Support de formats additionnels (CSV, JSON)
- **Visualisation graphique** : Affichage cartographique des itinéraires
- **Statistiques avancées** : Analyses plus poussées des performances et tendances

5.2.3. Interface Utilisateur

- **Thèmes visuels** : Système de thèmes personnalisables
- **Raccourcis clavier** : Navigation plus rapide dans l'interface
- **Aide contextuelle** : Tooltips et documentation intégrée

5.3. Perspectives d'Évolution et d'Amélioration

5.3.1. Évolutions Techniques

1. **Migration vers une architecture microservices** : Séparation des algorithmes en services indépendants
2. **Base de données** : Remplacement des fichiers texte par une base de données relationnelle
3. **API REST** : Exposition des fonctionnalités via une API pour intégration dans d'autres systèmes
4. **Interface web** : Développement d'une version web accessible depuis n'importe quel navigateur

5.3.2. Améliorations Algorithmiques

1. **Algorithmes hybrides** : Combinaison des approches existantes pour maximiser les avantages
2. **Apprentissage automatique** : Utilisation de l'historique pour prédire les meilleures stratégies
3. **Optimisation multi-objectifs** : Prise en compte de critères additionnels (temps, coût, préférences)

5.3.3. Fonctionnalités Métier

1. **Gestion des contraintes temporelles** : Intégration d'horaires d'ouverture, de créneaux de livraison
2. **Optimisation multi-véhicules** : Répartition des parcours sur plusieurs commerciaux
3. **Gestion des priorités** : Pondération des ventes selon leur importance
4. **Simulation dynamique** : Adaptation en temps réel aux changements (annulations, ajouts)

5.4. Retour d'Expérience

Ce projet a représenté un excellent exercice d'application des concepts théoriques en informatique. La combinaison entre développement d'application (SAÉ 2.01) et exploration algorithmique (SAÉ 2.02) a permis d'appréhender la complexité des systèmes informatiques réels.

Points forts du projet :

- Problématique concrète et applicable en entreprise
- Liberté dans le choix des algorithmes permettant la créativité
- Intégration de multiples compétences techniques

Difficultés rencontrées :

- Équilibrage entre fonctionnalités et délais de réalisation
- Optimisation des performances pour les scénarios complexes
- Gestion de la complexité croissante du code

Cette expérience confirme l'importance d'une architecture bien pensée dès le début du projet et de l'adoption de bonnes pratiques de développement pour maintenir la qualité du code à mesure que le projet évolue.

6. Annexes

Annexe A : Dossier de Tests Unitaires

Le projet inclut un ensemble complet de tests unitaires pour valider le comportement des algorithmes du modèle. Ces tests sont organisés dans le package `src/test/java/modele/` et comprennent :

A.1. Structure des Tests

```
src/test/java/modele/
├── ParcoursSimpleTest.java          # Tests de l'algorithme de parcours simple
├── ParcoursHeuristiqueTest.java     # Tests de l'algorithme heuristique
└── KMeilleuresSolutionsTest.java    # Tests de l'algorithme K meilleures
solutions
```

A.2. Méthodologie de Test

Chaque classe de test suit une méthodologie rigoureuse :

1. **Setup des données de test** : Création de cartes et scénarios de test standardisés
2. **Tests de validité** : Vérification que les itinéraires respectent toutes les contraintes

3. **Tests de performance** : Mesure et comparaison des temps d'exécution
4. **Tests de robustesse** : Gestion des cas limites et erreurs

A.3. Couverture des Tests

Les tests couvrent les aspects suivants :

- ☑ Validation de la contrainte vendeur → acheteur
- ☑ Vérification du départ et retour à Vélizy
- ☑ Calcul correct des distances totales
- ☑ Gestion des scénarios vides ou invalides
- ☑ Comparaison des performances entre algorithmes

A.4. Exécution des Tests

Les résultats des tests incluent :

- Temps d'exécution de chaque algorithme
- Distances calculées pour différents scénarios

Annexe B : Lien vers le Dépôt Git

Dépôt GitHub : https://github.com/shadowforce78/Java-SAE

Le dépôt contient :

- **Code source complet** avec commentaires détaillés
- **Tests unitaires** avec rapports de couverture
- **Documentation technique** (JavaDoc)
- **Scripts de build** (Maven)
- **Données de test** (scénarios et fichiers de distances)