Documentation du Projet Java-SAE

Architecture globale

Ce projet implémente différents algorithmes d'optimisation d'itinéraires pour planifier des trajets entre différentes villes. L'objectif est de trouver des parcours optimaux pour effectuer des ventes, en commençant et terminant à Vélizy.

Structure du projet

Le projet est organisé selon le modèle MVC (Modèle-Vue-Contrôleur) :

- **Modèle** : Contient les classes de données et les algorithmes
- **Vue** : Gère l'interface utilisateur
- **Contrôleur** : Fait le lien entre le modèle et la vue

Classes principales du modèle

Interface `IAlgorithme`

```
/**
 * Interface commune pour tous les algorithmes de parcours.
 * Permet d'uniformiser l'utilisation des différents algorithmes.
 */
public interface IAlgorithme {
    /**
     * Génère un itinéraire à partir d'un scénario donné.
     * @param scenario Le scénario contenant les ventes à effectuer
     * @return Liste des villes à visiter dans l'ordre
     */
    List<Ville> genererItineraire(Scenario scenario);

    /**
     * Calcule la distance totale d'un itinéraire donné.
     * @param itineraire Liste des villes à visiter dans l'ordre
     * @return La distance totale en kilomètres
     */
    int calculerDistanceTotale(List<Ville> itineraire);
}
```

Classe `CarteGraph`

```
/**
 * Représente la carte sous forme de graphe avec les distances entre les villes.
 * Cette classe est utilisée par les algorithmes pour calculer les distances des
itinéraires.
 */
```

```
public class CarteGraph {
    /** Map contenant les distances entre les paires de villes */
    private Map<Pair<Ville, Ville>, Integer> distances;

    /**
     * Constructeur de la carte.
     * @param distances Map contenant les distances entre les paires de villes
     */
    public CarteGraph(Map<Pair<Ville, Ville>, Integer> distances);

    /**
     * Retourne la distance entre deux villes.
     * @param a Première ville
     * @param b Deuxième ville
     * @return La distance entre les deux villes, ou Integer.MAX_VALUE si non
trouvée
     */
    public int getDistance(Ville a, Ville b);

    /**
     * Retourne l'ensemble de toutes les villes de la carte.
     * @return Un ensemble contenant toutes les villes
     */
    public Set<Ville> getToutesLesVilles();
}
```

Classe KMeilleuresSolutions

```
/**
 * Classe implémentant l'algorithme des k meilleures solutions (Algo 3)
 * qui utilise un backtracking pour explorer différentes possibilités
 * et retourner les k meilleurs itinéraires selon la distance totale.
 * L'itinéraire commence et se termine à Velizy.
 */
public class KMeilleuresSolutions implements IAlgorithme {
    /** La carte des distances entre les villes */
    private CarteGraph carte;

    /** Nombre de solutions à conserver */
    private int k;

    /**
     * Constructeur avec paramètres.
     * @param carte La carte des distances
     * @param k Le nombre de meilleures solutions à conserver
     */
    public KMeilleuresSolutions(CarteGraph carte, int k);

    /**
     * Constructeur par défaut qui conserve les 3 meilleures solutions.
     * @param carte La carte des distances
```

```
     */
    public KMeilleuresSolutions(CarteGraph carte);

    /**
     * Génère un itinéraire en retournant la meilleure solution parmi les k
     * calculées. L'itinéraire commence et se termine à Velizy.
     * @param scenario Le scénario contenant les ventes à effectuer
     * @return Liste des villes à visiter dans l'ordre de la meilleure solution
     */
    @Override
    public List<Ville> genererItineraire(Scenario scenario);

    /**
     * Génère plusieurs itinéraires (jusqu'à k) pour un scénario donné.
     * @param scenario Le scénario contenant les ventes à effectuer
     * @return Map associant un numéro de solution à son itinéraire
     */
    public Map<Integer, List<Ville>> genererPlusieursItineraires(Scenario
scenario);

    /**
     * Génère les k meilleures solutions pour un scénario donné.
     * Chaque solution commence et se termine à Velizy.
     * @param scenario Le scénario contenant les ventes à effectuer
     * @return Liste des k meilleures solutions triées par distance croissante
     */
    public List<Solution> genererKMeilleuresSolutions(Scenario scenario);

    /**
     * Classe interne pour représenter une solution (itinéraire + distance)
     */
    private static class Solution implements Comparable<Solution> {
        /** Liste des villes de l'itinéraire */
        List<Ville> itineraire;

        /** Distance totale de l'itinéraire */
        int distance;

        /**
         * Constructeur d'une solution.
         * @param itineraire Liste des villes à visiter
         * @param distance Distance totale de l'itinéraire
         */
        Solution(List<Ville> itineraire, int distance);

        /**
         * Compare deux solutions par leur distance.
         * @param autre L'autre solution à comparer
         * @return un entier négatif, zéro ou positif selon que cette solution
         *         a une distance inférieure, égale ou supérieure à l'autre
         */
        @Override
        public int compareTo(Solution autre);
```

```
    }
}
```

Classe ParcoursSimple

```
/**
 * Classe implémentant l'algorithme de parcours simple (Algo 1)
 * qui suit l'ordre vendeur → acheteur pour générer un itinéraire.
 */
public class ParcoursSimple implements IAlgorithme {
    /** La carte des distances entre les villes */
    private CarteGraph carte;

    /**
     * Constructeur.
     * @param carte La carte des distances
     */
    public ParcoursSimple(CarteGraph carte);

    /**
     * Génère un itinéraire simple en suivant l'ordre des ventes (vendeur → acheteur)
     * et en commençant et terminant à Velizy.
     * @param scenario Le scénario contenant les ventes à effectuer
     * @return Liste des villes à visiter dans l'ordre
     */
    @Override
    public List<Ville> genererItineraire(Scenario scenario);
}
```

Classe ParcoursHeuristique

```
/**
 * Classe implémentant l'algorithme de parcours heuristique (Algo 2)
 * qui utilise une approche gloutonne pour trouver un itinéraire proche de l'optimal.
 */
public class ParcoursHeuristique implements IAlgorithme {
    /** La carte des distances entre les villes */
    private CarteGraph carte;

    /**
     * Constructeur.
     * @param carte La carte des distances
     */
    public ParcoursHeuristique(CarteGraph carte);

    /**
     * Génère un itinéraire heuristique en choisissant à chaque étape
```

```
     * la ville la plus proche qui répond aux contraintes.
     * L'itinéraire commence et se termine à Velizy.
     * @param scenario Le scénario contenant les ventes à effectuer
     * @return Liste des villes à visiter dans l'ordre
     */
    @Override
    public List<Ville> genererItineraire(Scenario scenario);
}
```

Classes de données

Classe `Ville`

```
/**
 * Représente une ville avec son nom et ses coordonnées.
 */
public class Ville {
    /** Nom de la ville */
    private String nom;

    /**
     * Constructeur.
     * @param nom Nom de la ville
     */
    public Ville(String nom);

    /**
     * Retourne le nom de la ville.
     * @return Le nom de la ville
     */
    public String getNom();

    /**
     * Vérifie l'égalité avec un autre objet.
     * Deux villes sont égales si elles ont le même nom.
     * @param obj L'objet à comparer
     * @return true si les villes sont égales, false sinon
     */
    @Override
    public boolean equals(Object obj);

    /**
     * Retourne le code de hachage de la ville.
     * @return Le code de hachage basé sur le nom de la ville
     */
    @Override
    public int hashCode();
}
```

Classe `Vente`

```java
/**
 * Représente une vente entre un vendeur et un acheteur.
 */
public class Vente {
    /** Ville du vendeur */
    private Ville villeVendeur;

    /** Ville de l'acheteur */
    private Ville villeAcheteur;

    /** Membre vendeur */
    private Membre vendeur;

    /** Membre acheteur */
    private Membre acheteur;

    /**
     * Constructeur.
     * @param villeVendeur Ville du vendeur
     * @param villeAcheteur Ville de l'acheteur
     * @param vendeur Membre vendeur
     * @param acheteur Membre acheteur
     */
    public Vente(Ville villeVendeur, Ville villeAcheteur, Membre vendeur, Membre
acheteur);

    /**
     * Retourne la ville du vendeur.
     * @return La ville du vendeur
     */
    public Ville getVilleVendeur();

    /**
     * Retourne la ville de l'acheteur.
     * @return La ville de l'acheteur
     */
    public Ville getVilleAcheteur();

    /**
     * Retourne le membre vendeur.
     * @return Le membre vendeur
     */
    public Membre getVendeur();

    /**
     * Retourne le membre acheteur.
     * @return Le membre acheteur
     */
    public Membre getAcheteur();
}
```

Classe Membre

```java
/**
 * Représente un membre de l'application pouvant être vendeur ou acheteur.
 */
public class Membre {
    /** Nom du membre */
    private String nom;

    /** Prénom du membre */
    private String prenom;

    /**
     * Constructeur.
     * @param nom Nom du membre
     * @param prenom Prénom du membre
     */
    public Membre(String nom, String prenom);

    /**
     * Retourne le nom du membre.
     * @return Le nom du membre
     */
    public String getNom();

    /**
     * Retourne le prénom du membre.
     * @return Le prénom du membre
     */
    public String getPrenom();
}
```

Classe Scenario

```java
/**
 * Représente un scénario contenant une liste de ventes à effectuer.
 */
public class Scenario {
    /** Liste des ventes du scénario */
    private List<Vente> ventes;

    /** Identifiant du scénario */
    private int id;

    /**
     * Constructeur.
     * @param ventes Liste des ventes du scénario
     * @param id Identifiant du scénario
     */
    public Scenario(List<Vente> ventes, int id);
```

```
    /**
     * Retourne la liste des ventes du scénario.
     * @return La liste des ventes
     */
    public List<Vente> getVentes();

    /**
     * Retourne l'identifiant du scénario.
     * @return L'identifiant du scénario
     */
    public int getId();
}
```

Utilitaires

Classe DistanceParser

```
/**
 * Classe utilitaire pour parser les fichiers de distances.
 */
public class DistanceParser {
    /**
     * Parse un fichier de distances et retourne une carte avec les distances
entre les villes.
     * @param cheminFichier Chemin vers le fichier de distances
     * @return Une carte contenant les distances entre les villes
     * @throws IOException En cas d'erreur de lecture du fichier
     */
    public static CarteGraph parseDistances(String cheminFichier) throws
IOException;
}
```

Classe ScenarioParser

```
/**
 * Classe utilitaire pour parser les fichiers de scénario.
 */
public class ScenarioParser {
    /**
     * Parse un fichier de scénario et retourne un objet Scenario.
     * @param cheminFichier Chemin vers le fichier de scénario
     * @param id Identifiant du scénario
     * @return Un objet Scenario contenant les ventes du fichier
     * @throws IOException En cas d'erreur de lecture du fichier
     */
    public static Scenario parseScenario(String cheminFichier, int id) throws
IOException;
}
```

Classe `BenchmarkAlgorithmes`

```
/**
 * Classe pour comparer les performances des différents algorithmes.
 */
public class BenchmarkAlgorithmes {
    /**
     * Classe interne pour représenter un résultat de benchmark.
     */
    public static class ResultatBenchmark {
        /** Nom de l'algorithme */
        private String nomAlgorithme;

        /** Distance totale de l'itinéraire */
        private int distance;

        /** Temps d'exécution en millisecondes */
        private long tempsExecution;

        /**
         * Constructeur.
         * @param nomAlgorithme Nom de l'algorithme
         * @param distance Distance totale de l'itinéraire
         * @param tempsExecution Temps d'exécution en millisecondes
         */
        public ResultatBenchmark(String nomAlgorithme, int distance, long
tempsExecution);

        // Getters pour les propriétés
    }

    /**
     * Exécute un benchmark sur un algorithme donné avec un scénario.
     * @param algorithme L'algorithme à tester
     * @param nomAlgorithme Le nom de l'algorithme
     * @param scenario Le scénario à utiliser pour le test
     * @return Le résultat du benchmark
     */
    public static ResultatBenchmark executerBenchmark(IAlgorithme algorithme,
String nomAlgorithme, Scenario scenario);

    /**
     * Compare les performances de plusieurs algorithmes sur un même scénario.
     * @param algorithmes Map associant le nom d'un algorithme à son instance
     * @param scenario Le scénario à utiliser pour les tests
     * @return Liste des résultats de benchmark triés par distance croissante
     */
    public static List<ResultatBenchmark> comparerAlgorithmes(Map<String,
IAlgorithme> algorithmes, Scenario scenario);
}
```

Contrôleur

Classe `Controleur`

```java
/**
 * Classe principale de contrôle de l'application.
 * Fait le lien entre le modèle et la vue.
 */
public class Controleur {
    /** Instance de la carte */
    private CarteGraph carte;

    /** Liste des algorithmes disponibles */
    private Map<String, IAlgorithme> algorithmes;

    /** Scénario courant */
    private Scenario scenarioCourant;

    /**
     * Initialise le contrôleur avec les données de la carte.
     * @param cheminFichierDistances Chemin vers le fichier de distances
     * @throws IOException En cas d'erreur de lecture du fichier
     */
    public void initialiserCarte(String cheminFichierDistances) throws
IOException;

    /**
     * Charge un scénario à partir d'un fichier.
     * @param cheminFichierScenario Chemin vers le fichier de scénario
     * @param id Identifiant du scénario
     * @throws IOException En cas d'erreur de lecture du fichier
     */
    public void chargerScenario(String cheminFichierScenario, int id) throws
IOException;

    /**
     * Exécute un algorithme sur le scénario courant.
     * @param nomAlgorithme Nom de l'algorithme à exécuter
     * @return L'itinéraire généré par l'algorithme
     */
    public List<Ville> executerAlgorithme(String nomAlgorithme);

    /**
     * Compare les performances des algorithmes sur le scénario courant.
     * @return Liste des résultats de benchmark
     */
    public List<BenchmarkAlgorithmes.ResultatBenchmark> comparerAlgorithmes();
}
```

Vue

Classe `VBoxRoot`

```
/**
 * Classe principale de la vue, contenant l'ensemble des éléments de l'interface.
 */
public class VBoxRoot extends VBox {
    /** Contrôleur de l'application */
    private Controleur controleur;

    /**
     * Constructeur.
     * @param controleur Le contrôleur de l'application
     */
    public VBoxRoot(Controleur controleur);

    /**
     * Initialise les composants de l'interface.
     */
    private void initialiserComposants();

    /**
     * Met à jour l'affichage de l'itinéraire.
     * @param itineraire L'itinéraire à afficher
     */
    public void afficherItineraire(List<Ville> itineraire);
}
```

Conclusion

Cette documentation fournit une vue d'ensemble des principales classes et interfaces du projet Java-SAE. Le projet implémente différents algorithmes d'optimisation d'itinéraires (simple, heuristique et k-meilleures solutions) pour trouver des parcours optimaux entre des villes, en commençant et terminant à Vélizy.

L'architecture MVC permet une séparation claire entre les données (modèle), l'interface utilisateur (vue) et la logique de contrôle (contrôleur), facilitant ainsi la maintenance et l'évolution du projet.