S 2.02

DEV

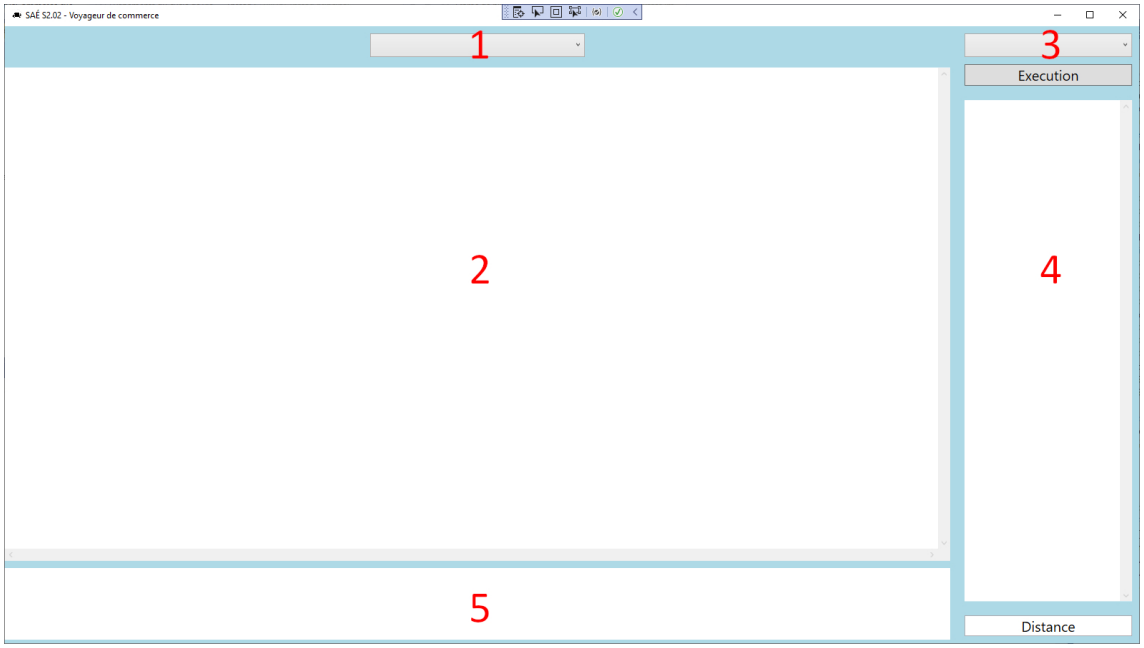
# Framework

## 1 Présentation générale

L'objectif de ce TP est de vous faire découvrir le framework que vous allez utiliser pour cette SAÉ. Ce framework prend la forme d'un projet C# que vous devrez compléter où toute la partie "graphique" est déjà implémentée. Il vous permettra donc de tester vos heuristiques sur des exemples et de visualiser leurs résultats.

## 2 Interface principale

Vous trouverez le projet dans le répertoire **VoyageurDeCommerce** sur le commun. Récupérez le et lancez le avec Visual Studio et exécutez-le. La fenêtre ci-dessous s'ouvre alors.




Cette fenêtre contient 5 éléments importants (certains n'étant pas encore fonctionnels à ce stade du projet) :

1. Une liste déroulante permettant de choisir un fichier ".gph" qui représente un graphe.
2. Une zone centrale dans laquelle se dessinera le graphe une fois celui-ci chargé.
3. Une deuxième liste déroulante permettant de choisir l'heuristique que l'on souhaite exécuter et, juste en dessous, un bouton permettant de l'exécuter.
4. Une zone dans laquelle s'afficheront les différentes étapes intermédiaires que vous aurez choisies de sauver.
5. Une zone dans laquelle la tournée sélectionnée sera affichée.

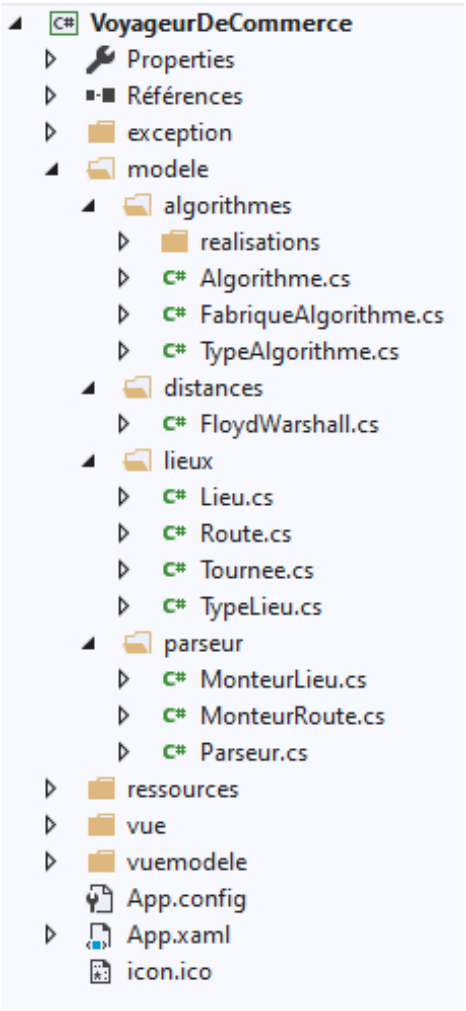
Votre version du framework n'est pas (complètement) fonctionnelle. En effet, il vous reste à en coder une partie (lecture du fichier, heuristiques...), il ne vous est donc pas vraiment possible de tester vous-même son fonctionnement pour le moment.

**Question 1 :** Observez le fonctionnement d'une version fonctionnelle du framework avec votre enseignant.

# 3 Structure du code

## 3.1 Structure générale

Observons maintenant la partie code du framework. Une grande partie du code qui vous est fournie est en charge de l'IHM, vous n'aurez pas à le modifier. Voici la structure générale du projet :




Comme vous pouvez le voir, le code est structuré en cinq packages principaux :

- **exception** : contient toute la partie relative à la gestion d'exceptions (vous n'aurez pas besoin de le toucher).
- **modele** : contient la couche métier du programme, c'est là que la majorité de votre code se trouvera.
- **ressources** : répertoire contenant les 2 graphes par défaut.
- **vue** : contient toute la partie graphique du projet (vous n'aurez pas besoin de le toucher).
- **vuemodele** : contient ce qui permet de faire le lien entre la couche IHM et la couche métier (vous n'aurez pas besoin de le toucher).

Le package **modele** est lui même structuré en quatre packages :

- **algorithmes** : qui contiendra vos classes où vous implémenterez vos heuristiques.
- **distances** : qui contient une version fonctionnelle de l'algorithme de Floyd Warshall.
- **lieux** : qui contient toutes les classes représentant les éléments de base : lieux, routes...
- **parseur** : qui contient toutes les classes en charge de la lecture des fichiers de graphes.

**Question 2 :** Listez toutes les classes de la couche métier du projet et donnez le rôle de chacune d'elles.

## 3.2 Création d'un graphe à la main

Nous allons maintenant voir comment les éléments de base fonctionnent et implémenter quelques fonctionnalités pour celles-ci. Pour cela, nous allons créer "à la main" un graphe (plus tard nous verrons comment importer un graphe depuis un fichier).

**Question 3 :** Dans la méthode "Parser" de la classe "Parseur", créez les cinq lieux suivants et ajoutez-les au dictionnaire ListeLieux.

| Nom du lieu | Type du lieu | X | Y |
|---|---|---|---|
| 1 | USINE | 0 | 0 |
| 2 | MAGASIN | 2 | 0 |
| 3 | MAGASIN | -2 | 2 |
| 4 | MAGASIN | 4 | 2 |
| 5 | MAGASIN | 1 | 4 |

**Question 4 :** Lancez l'application et sélectionnez dans la liste déroulante n'importe lequel des deux graphes. Vérifiez que vos cinq lieux s'affichent bien.

**Question 5 :** Toujours dans la méthode "Parser", créez les routes suivantes et ajoutez les à la liste ListeRoutes.

| **Nom du lieu de départ** | **Nom du lieu d'arrivé** | Distance |
|---|---|---|
| 1 | 2 | 2 |
| 1 | 3 | 3 |
| 1 | 5 | 6 |
| 2 | 4 | 1 |
| 2 | 5 | 3 |
| 3 | 5 | 4 |
| 4 | 5 | 1 |

**Question 6 :** Lancez l'application et sélectionnez dans la liste déroulante n'importe lequel des deux graphes. Vérifiez que votre graphe s'affiche bien.

## 3.3 Création du premier algorithme

Nous allons implémenter notre première heuristique : Tournée croissante. Dans notre code, chaque heuristique aura sa propre classe (héritant de la classe abstraite Algorithme). Cette classe disposera d'une property Nom (à surcharger) qui doit renvoyer le nom de l'heuristique (pour l'affichage) et une méthode "Executer" dont le but est de construire progressivement une tournée de lieu. Attention, la tournée que vous générez doit impérativement être celle accessible par la property "Tournee" de la classe mère.

**Question 7 :** Dans le package "algorithmes>realisations" créez une classe "AlgorithmeCroissant" héritant de la classe abstraite "Algorithme".

**Question 8 :** Surchargez la property "Nom" de cette classe pour qu'elle renvoie "Tournée croissante".

**Question 9 :** Implémentez la méthode "Executer" de cette classe pour qu'elle ajoute les lieux un à un dans la tournée "this.Tournee".
Si vous lancez l'application maintenant vous verrez que votre algorithme n'apparait pas dans la liste des algorithmes disponibles.

**Question 10 :** Ajoutez un élément CROISSANT dans l'énumération.

**Question 11 :** Modifiez la fabrique des algorithmes pour ajouter ce nouveau cas en vous inspirant du code déjà présent dans la classe.

**Question 12 :** Lancez le programme, sélectionnez un graphe puis votre algorithme et cliquez sur le bouton Exécuter.

Rien ne se passe. C'est normal, pour que le programme affiche votre tournée, il faut lui demander d'en prendre une "photo" à un instant donné, pour cela, il suffit de rajouter une commande dans votre code.

**Question 13 :** Ajoutez à la fin de la méthode "Executer" de votre algorithme la ligne :

this.NotifyPropertyChanged("Tournee") ;

**Question 14 :** Lancez le programme, sélectionnez un graphe puis votre algorithme et cliquez sur le bouton Exécuter.

Une tournée apparait dans la zone de droite ! Mais elle indique une distance de -1 et si vous cliquez dessus, votre programme plante... C'est normal... enfin non, mais c'est juste que nous n'avons pas lancé notre algorithme de calcul de plus courts chemins et donc le calcul des distances ne se fait pas correctement.

Pour des questions d'efficacité (surtout les autres heuristiques), l'algorithme retenu ici est celui de Floyd-Warshall qui calcule d'un seul coup les distances et les plus courts chemins entre toutes les paires de

sommets. L'algorithme est déjà implémenté mais il nous faut lui faire lancer les calculs au début de notre algorithme.

**Question 15 :** Ajoutez au début de la méthode "Executer" de votre algorithme la ligne :

FloydWarshall.calculerDistances(listeLieux, listeRoute) ;

**Question 16 :** Implémentez la property "Distance" de la classe "Tournée". On rappelle que la distance d'une tournée (1,2,3) est égale à

d(1,2) + d(2,3) + d(3,1).

**Question 17 :** Testez votre algorithme. Vous devez obtenir une tournée de longueur 17.

Quand vous cliquez sur une tournée (dans la zone de droite), celle-ci devrait s'afficher dans la zone en bas. Actuellement, ce n'est pas le cas.

**Question 18 :** Implémentez la méthode "ToString" de la classe "Tournee" pour qu'elle renvoie "1 => 2 => 3 => 1" pour une tournée (1,2,3).

**Question 19 :** Vérifiez que l'affichage de la tournée se fait bien.
Dans la question 13, nous avons vu comment prendre une photo d'une tournée. Il est possible de prendre plusieurs photos de la tournée à différents moments de l'algorithme afin de pouvoir voir son évolution au cours de l'algorithme.

**Question 20 :** Modifiez votre code pour qu'il prenne une photo de la tournée après chaque ajout de lieu (et non pas uniquement après le dernier ajout).

# 4 Chargement des fichiers de graphes et mesure d'efficacité

## 4.1 Chargement des fichiers de graphes

Maintenant que nous disposons d'un algorithme fonctionnel, il est temps de ne plus créer notre graphe à la main directement dans le code mais de le créer à partir d'un fichier.

**Question 21 :** Supprimez le contenu de la méthode "Parser" de la classe "Parseur".
La classe Parseur a pour rôle de lire un fichier de graphe et d'en extraire une liste de lieux et une liste de routes. Son constructeur demande en paramètre le nom du fichier à lire. Il est appelé automatiquement quand l'utilisateur sélectionne un fichier dans la liste déroulante en haut de la fenêtre. Ce nom va nous permettre de déterminer l'adresse du fichier à ouvrir.

**Question 22 :** Ajoutez un attribut "adresseFichier" de type string au parseur et ajoutez la commande suivante à la fin du constructeur :

this.adresseFichier = Path.GetDirectoryName(Assembly.GetEntryAssembly().Location) +
@"\ressources\" + nomDuFichier ;

**Question 23 :** Essayez de donner un sens à cette mystérieuse incantation.

Dans la méthode "Parser", nous allons ouvrir notre fichier puis le lire ligne à ligne. Pour cela nous allons utiliser une structure assez classique en C# que vous verrez (ou avez déjà vu) plus en détail dans un TP de la ressource R2.01.

**Question 24 :** Après l'avoir compris, recopiez le code suivant dans la méthode "Parser".

```
//Ouverture du fichier
using (StreamReader stream = new StreamReader(this.adresseFichier))
{
    string ligne;
    while ((ligne = stream.ReadLine()) != null)
    {
        //Lecture d'une ligne
    }
}
```

A l'intérieur du while, nous allons donc devoir expliquer comment traiter une ligne du fichier.

**Question 25 :** Ouvrez avec Visual Studio ou avec un éditeur de texte l'un des fichiers .gph fournis avec le projet.

Comme vous pouvez le voir, il y a deux types de lignes, celles pour les lieux qui commencent par USINE ou MAGASIN et celles pour les routes. Dans les deux cas, les informations dont nous avons besoin pour créer le lieu ou la route sont séparées par des espaces. La première chose à faire est donc de "couper" notre chaine de caractères en plusieurs morceaux.

**Question 26 :** Utilisez la méthode "split" des strings pour couper "ligne" en un tableau de string que nous appellerons "morceaux".

En regardant la valeur de "morceaux[0]" il nous est donc possible de savoir si nous devons fabriquer un lieu ou une route. Les autres cases du tableau contiennent les informations dont les constructeurs de Lieu ou de Route ont besoin mais sous forme de chaines de caractères qu'il nous faudra donc convertir.

La construction des routes et des lieux étant un peu technique (il faut récupérer des informations dans "morceaux", les convertir...), nous allons déporter cette partie du code dans deux classes "MonteurLieu" et "MonteurRoute" dont le rôle sera exclusivement de créer un lieu ou une route à partir du tableau "morceaux".

**Question 27 :** Complétez le code de la méthode "Parser" pour que :

- si morceaux[0]="ROUTE", on ajoute à listeRoutes le résultat de la méthode "Creer" de la classe "MonteurRoute".
- sinon, on ajoute au dictionnaire listeLieux, le résultat de la méthode "Creer" de la classe "MonteurLieu".

**Question 28 :** Implémentez la méthode "Creer" de la classe "MonteurLieu" pour qu'elle renvoie un lieu créé à partir du tableau "Morceaux". Pour cela, on pourra éventuellement regarder la valeur de "Morceaux[0]" pour savoir si l'on souhaite créer un Magasin ou une Usine puis convertir les autres morceaux du tableau avant de les donner au constructeur de la classe Lieu en utilisant la commande "Int32.Parse()".

**Question 29 :** Implémentez la méthode "Creer" de la classe "MonteurRoute".

**Question 30 :** Testez votre programme.

## 4.2 Mesure d'efficacité

Pour comparer des heuristiques entre elles, il peut être pertinent de comparer leurs vitesses d'exécution. En C#, nous disposons d'une classe spécialement conçue pour cela : Stopwatch. Nous utiliserons principalement 3 méthodes/properties de cette classe :

- **Start()** : qui démarre la stopwatch.
- **Stop()** : qui arrête la stopwatch.
- **ElapsedMilliseconds** : qui renvoie le nombre de millisecondes qui se sont écoulées.

**Attention**, comme nous l'avons vu précédemment, il est possible de faire prendre à notre algorithme des "photos" de la tournée de temps en temps afin de pouvoir suivre son évolution. Prendre ces photos rallonge arbitrairement le temps d'exécution de l'algorithme. Par conséquent, il faut penser à arrêter la stopwatch avant de prendre une photo et de la redémarrer après !

**Question 31 :** Modifiez le code de votre algorithme pour en mesurer le temps d'exécution. Pour cela :

- créez une stopwatch et démarrer la au début de la méthode Executer,
- interrompez là à chaque prise de photo,
- arrêtez la à la fin de la méthode,
- faire affichez le temps d'exécution avec

this.TempsExecution = sw.ElapsedMilliseconds ;

**Question 32 :** Mesurez le temps d'exécution de votre algorithme sur les deux graphes fournis.