

## 2023- ML1a - Find Your Way

Arrous Thomas - Decker Benjamin - Guetteville Nathan - Le Boulc'h Erin

Double Licence Mathématiques Informatique - Semestre 4

## Sommaire

<b>1</b>	<b><u>Introduction</u></b>	<b>3</b>
1.1	Contexte . . . . .	3
1.2	Rappel du sujet . . . . .	3
1.3	Objectif du projet . . . . .	3
<b>2</b>	<b><u>Cahier des charges</u></b>	<b>3</b>
2.1	Fonction principale du programme . . . . .	3
2.2	Fonctionnalités exigées . . . . .	3
2.3	Fonctionnalités supplémentaires . . . . .	4
2.4	Axe de travail et priorités . . . . .	4
<b>3</b>	<b><u>Spécification fonctionnelle et détaillée</u></b>	<b>4</b>
3.1	Modélisation d'un graphe de bâtiment . . . . .	4
3.1.1	Classes du modèle . . . . .	5
3.1.2	Parseur et CSV . . . . .	10
3.1.3	Réutilisabilité . . . . .	15
3.2	Algorithme du chemin le plus court . . . . .	15
3.2.1	Déroulement de l'algorithme . . . . .	15
3.2.2	Optimisation . . . . .	17
3.3	Affichage graphique . . . . .	18
3.3.1	Classes de la vue . . . . .	18
3.3.2	Affichage des noeuds et chemins . . . . .	20
3.3.3	Entrées utilisateurs . . . . .	21
3.3.4	Indications . . . . .	22
3.3.5	Itinéraires favoris . . . . .	24
3.3.6	Sélection du bâtiment . . . . .	24
<b>4</b>	<b><u>Fonctionnalités non implémentées</u></b>	<b>25</b>
4.1	Autres bâtiments . . . . .	25
4.2	Faciliter le formatage des données . . . . .	26
4.3	Zoom du plan . . . . .	26
4.4	Sélection sur le plan . . . . .	26
4.5	Autocomplétion . . . . .	27
4.6	Historique . . . . .	27
4.7	Inversion de l'itinéraire . . . . .	27
<b>5</b>	<b><u>Annexes</u></b>	<b>27</b>
5.1	Diagramme de structure de dossiers et fichiers . . . . .	27
5.2	Diagramme de classes . . . . .	28
5.3	Guide d'utilisation . . . . .	28
5.4	Bibliographie . . . . .	29
<b>6</b>	<b><u>Conclusion</u></b>	<b>30</b>

# **1 Introduction**

## **1.1 Contexte**

Ce document est le rapport du projet **ML1a-Find Your Way** réalisé dans le cadre de l’UE Projet de Programmation du 4e semestre de licence d’Informatique à l’université Paris Cité. Le groupe ML1a est constitué de Thomas Arrous, Benjamin Decker, Nathan Guetteville et Erin Le Boulc’h, étudiants en Double Licence Mathématiques-Informatique. L’enseignant encadrant de ce groupe est Maximilien Lesellier.

## **1.2 Rappel du sujet**

Il est difficile pour les nouveaux venus sur le campus de trouver son chemin pour rejoindre les salles de cours et de TD. En particulier, on se demande comment trouver le plus court chemin permettant d’atteindre une salle donnée en partant d’une autre salle du campus. Il est demandé aux étudiants de concevoir un programme permettant de trouver le plus court chemin entre deux salles, sur un plan du campus comprenant les différents bâtiments où les cours peuvent se tenir, incluant la halle aux farines.

Le travail qu’il vous est demandé de réaliser est une modélisation des différents bâtiments dans lesquels vous avez cours, les plans vous seront fournis pour les mesures, et de programmer un algorithme calculant le plus court chemin entre deux points. Une interface doit afficher le ou les trajet(s) et les instructions pour le déplacement de l’utilisateur.

## **1.3 Objectif du projet**

Ce projet a pour objectif d’apporter une solution concrète à un problème donné sous la forme d’un programme informatique. Il a pour but de nous faire appliquer les notions et savoir-faire appris en Java jusqu’ici dans le cadre de la Licence d’Informatique. L’aspect de travail de groupe est également mis en avant car le succès de ce projet repose sur la capacité des membres à communiquer et à se répartir les tâches efficacement dans un délai donné.

# **2 Cahier des charges**

## **2.1 Fonction principale du programme**

Nous rappelons que le programme de ce projet doit pouvoir indiquer à un utilisateur le chemin le plus court entre deux salles d’un bâtiment qu’il a sélectionnés.

## **2.2 Fonctionnalités exigées**

Les fonctionnalités suivantes constituent le minimum requis dans le cadre de ce projet :

- Modéliser des trajets dans un bâtiment
- Programmer un algorithme de calcul de chemin le plus court
- Créer une interface utilisateur qui permet de sélectionner deux salles, visualiser le trajet le plus court et les instructions correspondantes

## 2.3 Fonctionnalités supplémentaires

Voici les fonctionnalités additionnelles que nous avons implémentées :

- Se rendre à des points d'intérêts (machines à café, crous, toilettes ...)
- Renseigner des contraintes réelles (accès aux ascenseurs permis ou non)
- Situer une salle
- Pouvoir enregistrer des lieux ou des itinéraires favoris
- Visualiser le graphe entier (mode debug)

## 2.4 Axe de travail et priorités

Nous avons dès les premières réunions concernant la conception et la modélisation de notre programme établi trois axes de travail : la modélisation d'un bâtiment, l'algorithme du chemin le plus court et l'interface utilisateur. Il s'agit ici de l'ordre chronologique dans lequel nous les avons traités et également l'ordre dans lequel ils sont abordés par la suite dans la spécification fonctionnelle.

# 3 Spécification fonctionnelle et détaillée

Dans cette partie, nous allons détailler l'implémentation des fonctionnalités de notre projet. Nous verrons dans un premier temps la modélisation d'un graphe de bâtiment. Dans un deuxième temps, nous présenterons l'algorithme du chemin le plus court. Dans un troisième et dernier temps, nous détaillerons l'affichage graphique et l'interface utilisateur.

## 3.1 Modélisation d'un graphe de bâtiment

Une des premières tâches auxquelles nous avons réfléchi est la représentation des bâtiments. Nous avons rapidement établi un lien entre navigation spatiale et graphes.

En mathématiques, un graphe est un ensemble de sommets, représentant des objets, qui sont mis en relation par des arêtes. Cette structure de donnée nous a semblé particulièrement adaptée à la modélisation des différents chemins que l'on peut emprunter dans un bâtiment. En effet, on peut facilement se représenter les sommets d'un graphe comme étant des lieux d'un bâtiment dans lequel on souhaite se déplacer en passant par des chemins, en l'occurrence, des arêtes du graphe. Un graphe peut être orienté ou non. Si un graphe est orienté, même s'il y a un lien entre deux sommets, selon sur lequel on se trouve, on ne peut pas forcément l'emprunter s'il n'est dirigé que dans l'autre sens. Dans notre choix de modélisation, nous avons décidé d'utiliser un graphe non orienté puisqu'il n'y a pas de sens de circulation imposé dans les bâtiments de la faculté.

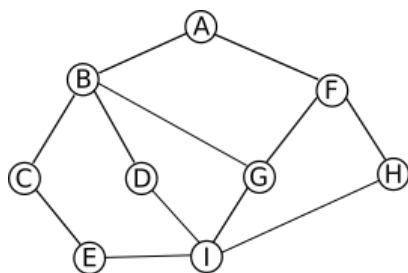


Figure 1: Graphe non orienté à 10 sommets

Toutefois plusieurs questions se sont posées. Premièrement, nous avons différentes possibilités pour coder un graphe. Utiliser une matrice d'adjacence avec des liens pondérés semblait être l'usage le plus répandu en programmation. Toutefois, nous n'en utilisons pas dans notre projet. Pour résumer, comme nous avons besoin de distinguer les points par plusieurs attributs, une matrice n'indiquant que les liens, voire les distances, entre les nœuds, ne semblait pas la solution idéale. De plus, une matrice aussi large que le nombre de nœuds d'un bâtiment semblait difficile à manipuler. Nous allons expliquer en détail la modélisation choisie pour les nœuds et leurs voisins dans cette partie, puis dans la suivante nous expliquerons comment leur initialisation est réalisée et la manière dont nous stockons leurs informations.

Par ailleurs, notamment à cause de l'affichage de plusieurs étages, nous nous demandions s'il était mieux de n'utiliser qu'un seul grand graphe pour tout le bâtiment ou plutôt un graphe par étage. La séparation initiale entre les classes *Batiment* et *Etage* vient de cette réflexion. Nous en sommes venus à nous dire que, les étages étant reliés entre eux par des escaliers ou des ascenseurs, et un seul chemin pouvant parcourir plusieurs étages, il était nécessaire de modéliser un seul grand graphe. Cependant, nous avons conservé la distinction entre ces classes pour la vue, non seulement pour faciliter l'affichage des nœuds par étage, mais aussi pour pouvoir détecter les changements d'étage à signaler dans les indications à l'utilisateur.

Afin de modéliser un graphe représentant un bâtiment de l'université, nous avons tout d'abord eu besoin de modéliser les différents types de sommets de ce graphe. Pour cela, nous avons évoqué l'idée d'utiliser une énumération comme carrefour, salle, porte, escalier et ascenseur. Finalement, nous nous sommes rendu compte que ces sommets auraient des attributs spécifiques comme un nom pour les salles ou un contrôle d'accès pour les ascenseurs. Nous avons donc choisi de séparer les sommets en cinq classes, remodelées par la suite.

### 3.1.1 Classes du modèle

#### Noeud

La classe *Noeud* est une classe abstraite dont tous les sommets des graphes héritent.

Cette classe contient tout d'abord un attribut entier *id* qui permet d'identifier chaque nœud individuellement. Elle contient également un dictionnaire *HashMap* qui prend en clé un nœud et en valeur un double représentant une distance. Ce dictionnaire indique les voisins du nœud ainsi que la distance qui les sépare. Au lieu d'utiliser un *HashMap*, nous avons aussi songé à nous servir d'une liste de chemins partant de ce nœud. Ces chemins auraient été caractérisés par deux nœuds, une distance, ainsi qu'un booléen d'accès autorisé. Finalement, nous avons bien conservé une classe *Chemin* mais elle ne représente plus les liens partants d'un nœud. Enfin, la classe *Noeud* possède un attribut static entier *nbNoeud* qui compte le nombre de nœuds instanciés et

qui permet de leur attribuer le bon identifiant. Plus tard dans le projet, nous avons également choisi de donner un attribut *Batiment* aux nœuds afin de connaître à quel graphe de bâtiment ils appartiennent.

## Carrefour

*Carrefour* est une classe qui hérite de *Noeud*. Elle nous sert à représenter des intersections du bâtiment modélisé, dans les couloirs ou les escaliers par exemple.

Dans un premier temps, cette classe ne possédait pas d'attribut supplémentaire à la classe *Noeud* car elle n'englobait pas encore les escaliers et les ascenseurs et de ce fait, tous les carrefours étaient accessibles. Nous avons ensuite décidé de regrouper la classe *Carrefour* avec la classe *Escalier* car en réalité, les escaliers et les ascenseurs sont des carrefours entre les étages du bâtiment. Nous avons donc ajouté un attribut booléen *isAscenseur* permettant de savoir s'il est nécessaire d'avoir une autorisation pour passer par ce *Carrefour*. Enfin, lors de l'implémentation de l'affichage, nous avons donné aux carrefours trois nouveaux attributs entiers : *etage* qui est l'étage auquel se situe le noeud dans le bâtiment et qui nous indique s'il est nécessaire d'afficher le noeud en fonction de l'étage sur lequel la vue se trouve, ainsi que *x* et *y* qui sont les coordonnées en abscisse et en ordonnée du point qui sera affiché dans l'application pour représenter ce carrefour.

## Salle

La troisième classe représentant des sommets, et dernière à avoir été conservée, est la classe *Salle*, qui hérite également de *Noeud*.

Cette classe permet de représenter les différentes salles du bâtiment entre lesquelles il est possible de se déplacer. À l'origine, cette classe possédait un attribut String *nom* ainsi qu'une ArrayList de *Porte* contenant les différentes portes menant à la salle. Finalement, la classe ne conservera que son attribut *nom*, la classe *Porte* n'ayant pas été gardée. Contrairement aux carrefours, les salles n'ont pas de coordonnées car elles ne sont pas affichées dans la vue. Cette décision a été prise car certaines salles comme les toilettes sont présentes physiquement à plusieurs endroits dans le bâtiment mais ne sont représentées dans le graphe que par un unique nœud, qui lui est relié à chacun des carrefours aux entrées de la salle.

## Escalier [supprimée]

La classe *Escalier* représentait les escaliers et les ascenseurs des bâtiments. Elle héritait de la classe *Noeud* et possédait un attribut booléen *isAscenseur* de contrôle d'accès indiquant s'il fallait une autorisation pour passer ce nœud. Comme dit précédemment, cette classe n'a pas été conservée et, du fait de leur proximité conceptuelle, a été fusionnée avec la classe *Carrefour*.

## Porte [supprimée]

La classe *Porte* héritait également de *Noeud* et devait faire le lien entre les carrefours et les salles. Elle avait comme seul attribut une *Salle* *salle* indiquant la salle vers laquelle la porte mène. Cette classe a elle aussi été supplantée par la classe *Carrefour* car elle n'apportait pas grand-chose du point de vue de la modélisation. De plus, l'algorithme de chemin le plus court était censé être utilisé entre deux portes. Cela impliquait pour les salles ayant plusieurs portes que l'algorithme tourne une fois pour chaque porte de la salle de départ et de la salle d'arrivée. Sa

suppression a donc permis d'alléger la structure du modèle ainsi que d'améliorer les performances de l'application.

## Batiment

A sa construction, un *Batiment* construit la liste de ses étages selon ses étages *min* et *max* (6 étages pour la Halle aux Farines, du rez-de-chaussée au 5e étage). C'est aussi à sa construction que tous les nœuds du graphe et leurs liens sont initialisés grâce au parseur, appelé avec le path du dossier contenant les données du bâtiment.

Un bâtiment a également une échelle, qui est actuellement prédéfinie selon celle des plans de la Halle aux Farines, pour des besoins d'affichage qui seront détaillés plus loin. Pour l'établir, nous avons mesuré une distance réelle entre deux points du graphe dans le bâtiment afin de l'utiliser comme étalon et de le comparer avec sa distance selon la base de données.

## Etage

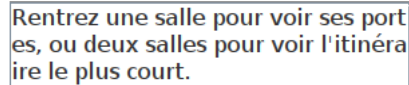
Initialement, les étages possédaient une liste de *Carrefour* et une liste de *Salle*. Les nœuds étaient ajoutés par le parseur en même temps qu'ils étaient ajoutés à la liste générale des nœuds du bâtiment. Quant aux salles, elles étaient construites à l'aide d'une fonction spécifique qui n'existe plus : *initSalles(int numeroEtage)*. Par un switch selon le numéro de l'étage à créer, elle devait construire les salles propres à celui-ci en appelant une fonction spécifique du type *initRdc()* ou *initTroisiemeEtage()*. Bien entendu, nous avons corrigé cette façon inefficace de construire le graphe qui ne pouvait s'appliquer qu'à ce bâtiment et n'était pas du tout réutilisable pour d'autres.

Désormais, la classe *Etage* regroupe simplement les carrefours d'un bâtiment qui sont sur cet étage. Ses instances sont créées en même temps que le bâtiment lui-même, et remplies avec leurs carrefours par le parseur au moment de leur création. Le but principal de cette classe est de pouvoir afficher une portion de graphe correspondant à un plan d'étage, notamment pour le debugging.

## Chemin

Pour de multiples raisons, nous avons eu besoin d'un objet représentant une suite de nœuds modélisant un itinéraire. Cette classe *Chemin* a été construite en premier lieu pendant l'implémentation de l'algorithme de Dijkstra afin de stocker le chemin le plus court entre deux nœuds, à la place d'une simple liste chaînée de nœuds. Les chemins sont utiles aussi à l'affichage pour le visuel du chemin le plus court, ainsi que pour les indications de direction. C'est pour cela que l'on retrouve dans ses attributs une liste de nœuds ainsi qu'une chaîne de caractères *indications*. L'attribut *distance* est quant à lui utilisé par l'algorithme du chemin le plus court pour comparer les longueurs des chemins possibles entre deux points du graphe et choisir la plus petite.

**Indications** On souhaite avoir une zone à l'écran qui contient un texte correspondant au chemin à suivre entre les deux salles choisies par l'utilisateur. S'il n'y a pas de salles sélectionnées ou seulement une, un message s'affiche dans cette zone pour enjoindre l'utilisateur à calculer un itinéraire. Bien sûr, le texte doit être mis à jour à chaque fois qu'un chemin est calculé.



Rentrez une salle pour voir ses portes, ou deux salles pour voir l'itinéraire le plus court.

Figure 2: Indication par défaut

Comme les indications de direction sont propres à chaque chemin, elles sont stockées dans l'attribut String *indications* de cette classe et mises à jour à chaque ajout de nœud par la fonction *updateIndications()* qui parse le chemin en *Segment*. Il s'agit d'une classe interne qui représente une portion de chemin par deux nœuds, le nombre d'étage entre les deux, s'il s'agit d'un segment horizontal ou vertical sur le plan et la distance les séparant.

Les indications que l'on souhaite donner entre le départ et l'arrivée sont : avancer et sur quelle distance, quand tourner et dans quelle direction, et quand prendre un ascenseur ou un escalier. Pour cela, on regarde successivement deux segments du chemin, disons s1 et s2 pour faciliter l'explication, que l'on parcourt du début à la fin dans une boucle. On distingue plusieurs cas :

- si on change d'étage dans s1 et dans s2, on incrémente le compteur d'étages;
- si on a changé d'étage dans s1 et non dans s2, alors on sort d'un ascenseur ou un escalier, donc on affiche l'instruction de changement d'étage, en indiquant le moyen. On remet le compteur d'étages à 0.

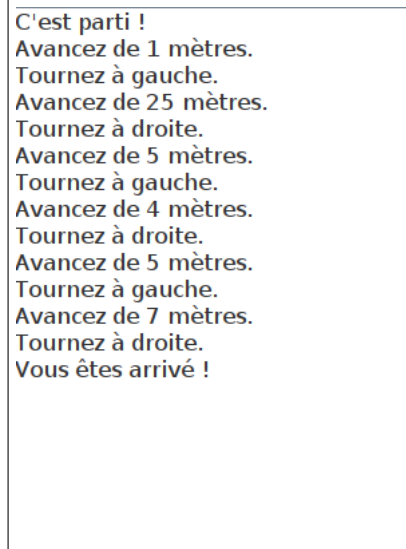
Si on ne change pas d'étages, alors c'est qu'on avance ou on tourne.

- si s1 et s2 ont la même direction, on ajoute la distance de s1 au compteur de distance car on est en ligne droite;
- si s1 est horizontal et s2 est vertical, ou inversement, cela veut dire qu'on tourne entre les deux segments. On affiche donc la distance accumulée de la ligne droite à effectuer puis on remet le compteur à zéro. On regarde selon les coordonnées s'il s'agit d'un virage à gauche ou à droite et on l'indique également à l'utilisateur.

On veut pouvoir indiquer à l'utilisateur des distances fiables. Il faut donc une échelle réaliste entre les coordonnées des carrefours de la base de données et les distances réelles. Un segment sur le plan - c'est-à-dire, une distance entre deux carrefours - doit correspondre à une distance réaliste. Ainsi, dans le cas d'une ligne droite dans la fonction qui parse le chemin, on accumule la distance entre chaque nœud de la ligne droite, puis une fois celle-ci terminée (par un virage, un changement d'étage ou l'arrivée) on affiche en mètres cette distance, convertie par un produit en croix avec l'échelle du bâtiment.

**Réutilisabilité** Dans la logique de réutilisabilité du code pour d'autres bâtiments, certains paramètres sont modifiables comme la longueur minimum d'affichage : on ne veut pas forcément afficher les indications telles que "avancer de 0.7 mètres" si une portion du chemin est représentée par un petit segment dans la base de données. Il y a donc une variable correspondante dans *updateIndications()*, *minDist*, qui peut être modifiée pour n'afficher que les lignes droites supérieures à *minDist*. Elle est fixée selon l'échelle du bâtiment des nœuds du chemin de sorte à être égale à environ 1 mètre.





C'est parti !  
Avancez de 1 mètres.  
Tournez à gauche.  
Avancez de 25 mètres.  
Tournez à droite.  
Avancez de 5 mètres.  
Tournez à gauche.  
Avancez de 4 mètres.  
Tournez à droite.  
Avancez de 5 mètres.  
Tournez à gauche.  
Avancez de 7 mètres.  
Tournez à droite.  
Vous êtes arrivé !

Figure 3: Indications de direction pour le chemin 237C → 264E

### Points d'intérêt

En plus des salles de TD, de TP, des amphithéâtres et des toilettes, nous avons choisi d'implémenter d'autres points d'intérêt de la Halle aux Farines vers lesquels l'utilisateur pourrait souhaiter se déplacer. Nous avons donc ajouté deux nœuds *Salle* dans notre graphe pour représenter les machines à café et le restaurant universitaire du CROUS. Puisqu'il y a plusieurs machines à café dans le bâtiment, cette salle a été implémentée de la même manière que les toilettes. C'est-à-dire que le nœud *Salle* café est relié à plusieurs carrefours dans tout le deuxième étage, ce qui permet de toujours se déplacer vers le distributeur le plus proche. Quant au CROUS, il est relié à la sortie de la Halle au Farines la plus proche de l'entrée du restaurant universitaire.

Il aurait pu être souhaitable d'ajouter également des carrefours au graphe afin d'avoir par exemple un chemin qui sort de la Halle pour aller jusqu'à l'entrée du CROUS. Cependant, cela aurait nécessité de décaler la quasi totalité des fichiers *csv* pour ajouter ces carrefours au bon étage afin de les afficher correctement dans la vue, mais puisque les salles ne sont pas affichées, ce problème ne se pose pas si nous nous contentons d'ajouter des nœuds *Salle* et de les relier à des carrefours déjà existants.

### 3.1.2 Parseur et CSV

#### Geogebra

Afin d'établir les données du graphe de la Halle aux Farines, il a fallu construire celui-ci en plaçant les différentes salles, intersections, escaliers et ascenseurs du bâtiment. Geogebra nous a permis de tracer ce graphe par dessus les plans de chaque étage à l'aide de points et de segments. Les points représentaient les nœuds et les segments les arêtes du graphes. Avec cette représentation, nous avons pu déterminer les coordonnées des carrefours que nous allons utiliser pour la vue ainsi que la distance entre chaque nœuds voisins. Nous avons également établi un code couleur pour les nœuds afin de différencier leurs types ou leurs attributs : les nœuds bleus représentaient des carrefours, les verts des salles et les rouges des ascenseurs. Une fois tous les nœuds placés, reliés et de la bonne couleur, nous avons rapporté toutes ces informations dans les fichiers *csv* permettant d'initialiser les nœuds et leurs voisins.

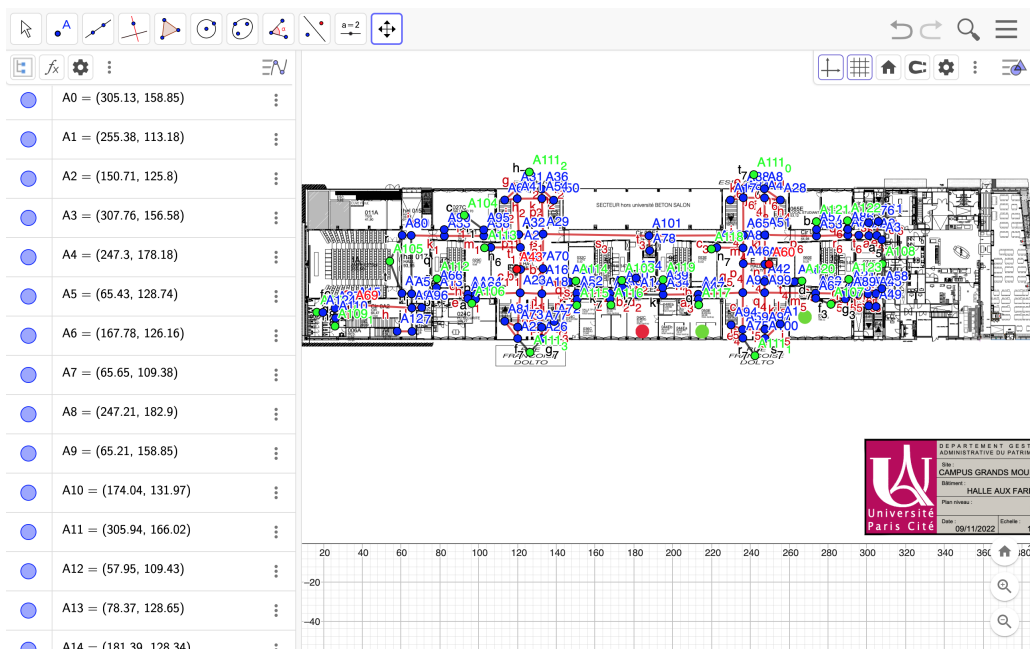


Figure 4: Plan et graphe du rez-de-Chaussée de la Halle aux Farines sur Geogebra

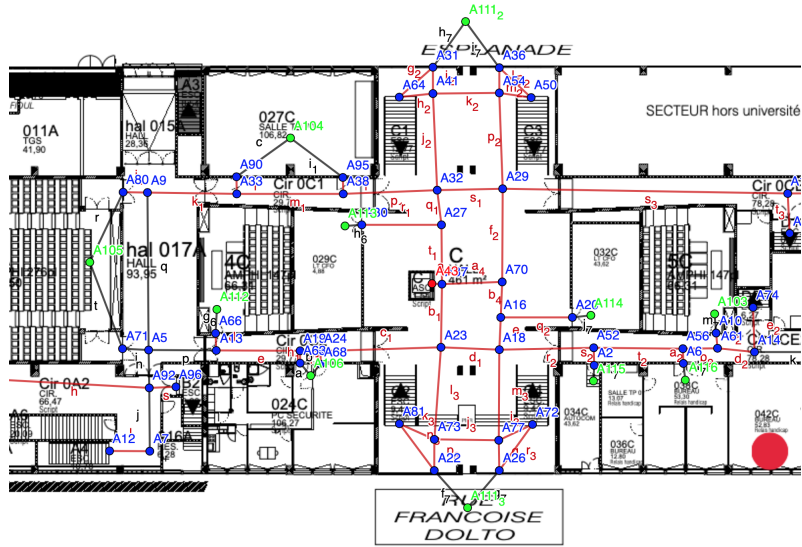


Figure 5: Détail du graphe du rez-de-chaussée de la Halle aux Farines.

Il a également fallu rajouter les liens entre les étages manuellement. En effet, les tracés sur Geogebra correspondent aux portions de graphes visibles par étage. Les liens entre les escaliers et les ascenseurs avec les étages supérieur et inférieur n'y étant pas, nous les avons insérés, en choisissant une distance de 1 pour tous les changements d'étage par ascenseurs et 2 pour tous les escaliers en prenant en compte l'algorithme de Dijkstra.

Pour répertorier les coordonnées des carrefours, nous avons récupéré celles indiquées par les graphes créés sur Geogebra. Il a fallu prêter attention aux systèmes de coordonnées qui diffèrent entre Geogebra et l'affichage graphique en Java. En particulier, l'axe des ordonnées n'est pas dans le même sens pour les deux.

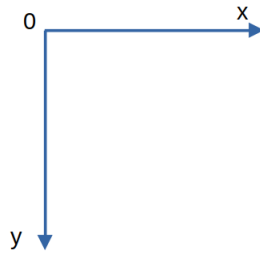


Figure 6: Système de coordonnées de l'affichage graphique

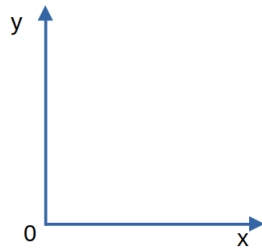


Figure 7: Système de coordonnées de Geogebra

Nous avons donc copié les coordonnées en abscisse de Geogebra. Pour les coordonnées en ordonnée, un programme simple nous a permis de faire la conversion des valeurs entre les deux systèmes. Les valeurs ont été décrémentées de 279,66, 279,66 étant la hauteur de l'image des plans sur géogebra. Nous avons ainsi pu obtenir la translation de tous les points. Cependant, les nombres réels ne peuvent pas être représentés avec une précision infinie en Java car ils sont stockés sous forme de valeurs binaires, avec une taille limitée, donc des erreurs d'arrondi peuvent survenir lorsque l'on fait des opérations arithmétiques sur des *Double*. C'est ce qui s'est passé pour la plupart des valeurs : les valeurs des ordonnées sur nos *csv* ressemblent pour la plupart à ce genre de valeur «117.0000000000003 », ce qui ne pose pas de réel problème.

## CSV

Le projet nécessitant de nombreuses données concernant l'existence des nœuds, leur emplacement et autres propriétés, nous avons eu besoin de trouver un moyen efficace de renseigner ces informations afin que le programme puisse les exploiter. Nous avons d'abord exploré l'idée d'utiliser des fichiers *.json*, couramment utilisés pour stocker des données, mais qui nécessitait d'installer des dépendances dans notre projet et comprenait une syntaxe un peu lourde. D'un autre côté, le format de fichier *.csv* présentait l'avantage d'être simple et d'utiliser moins de lignes. Nous avons donc opté pour ce dernier en utilisant une syntaxe bien particulière.

Il suffit de regrouper les informations d'un bâtiment dans ces fichiers selon ce format spécifique puis de les interpréter avec le parseur pour pouvoir construire le graphe. On distingue deux fichiers : en premier lieu celui contenant les informations de chaque nœud telles que sa nature (*Salle* ou *Carrefour*) et les attributs propres à celle-ci : nom de salle ou permission ascenseur et coordonnées. En second lieu, celui contenant tous les liens entre nœuds et leurs distances. Chaque bâtiment doit donc avoir un dossier dédié avec ces fichiers dans le dossier *ressources/csv* et donner l'adresse de ce dossier au parseur à sa construction.

**Syntaxe des noeuds** Dans un fichier comme *ressources/csv/haf/noeuds.csv*, les nœuds sont regroupés par étage. Pour les salles ayant des portes à plus d'un étage, la convention adoptée est de les mettre dans le plus bas. Nous rappelons à toutes fins utiles que le numéro d'étage n'est un attribut que pour les instances de *Carrefour* et que cet usage sert simplement à éviter de créer les mêmes salles plusieurs fois à différents étages. Chaque étage est signalé par un retour à la ligne et d'un point virgule suivi du numéro de l'étage.

```
carrefour;;174.54;151.31000000000003
;3
carrefour;;309.57;106.52000000000004
carrefour;;79.76;129.78000000000003
carrefour;;138.04;150.62000000000003
carrefour;;263.7;150.89000000000001
```

Figure 8: Les noeuds situés sous la ligne indiquant l'étage 3 sont tous situés à l'étage 3

Sur les autres lignes qui n'indiquent pas un changement d'étage au parseur se situent les informations des noeuds à créer. Une ligne contient les propriétés d'un noeud. Les informations sont séparées par des points-virgules et toujours dans le même ordre. La syntaxe est la suivante : le premier mot désigne la nature du noeud (*Salle* ou *Carrefour*).

S'il s'agit d'un *Carrefour*, la deuxième information concerne le statut d'ascenseur : si le carrefour est un ascenseur, on écrit "ascenseur", sinon, si le noeud est un carrefour normal dans un couloir ou un escalier, on ne met rien. Enfin les coordonnées en abscisse, puis en ordonnée, sont indiquées. L'étage est retenu par le parseur selon sa position dans le fichier, c'est pour cela qu'il est important de respecter le regroupement des noeuds sous la ligne indiquant leur étage.

```
carrefour;;92.13;164.55
```

Figure 9: Ligne représentant un Carrefour à la position (92.13 ; 164.55) du plan

```
carrefour;ascenseur;325.12;152.8
```

Figure 10: Ligne représentant un ascenseur à la position (325.12 ; 152.8) du plan

S'il s'agit d'une *Salle*, la deuxième et dernière information est le nom qu'elle porte.

```
salle;314B
```

Figure 11: Ligne représentant la salle 314B

**Syntaxe des voisins** Dans un fichier comme *ressources/csv/haf/voisins.csv*, les blocs ne concernent plus les noeuds d'un étage mais les voisins d'un seul noeud. La même syntaxe utilisée pour signaler au parseur un changement d'étage est utilisée pour signaler un changement de

nœud. On indique sur une nouvelle ligne l'*id* du nœud dont on va lister les voisins, disons *n* ici. Les lignes qui suivent respectent la syntaxe suivante : chaque ligne indique l'*id* d'un voisin de *n*, séparé de la distance entre les deux nœuds. Quand tous les voisins ont été listés, on passe au nœud suivant.

```

;622
639;13.83
666;2.04
786;15.55
;623
696;2.31

```

Figure 12: Ici, les voisins du nœud 622 sont listés : 622 est relié à 639, 666, 786. La distance entre les nœuds 622 et 639 est de 13.83 selon le système de coordonnées de Geogebra. On voit également le début de la liste des voisins du nœud 623.

```

;226
225;2.07

```

Figure 13: Le nœud 226 de la Halle aux Farines n'est lié qu'au nœud 225, avec une distance 2.07.

```

;225
125;4.86
207;5.61
226;2.07

```

Figure 14: On retrouve le nœud 226 dans les voisins de 225, avec la même distance.

## Parseur

Une instance de la classe *Parseur* peut construire le graphe du bâtiment qui lui est attribué grâce au dossier de *csv* transmis par le path.

Elle contient deux méthodes aux rôles bien définis qui sont appelés à la construction du bâtiment. La première est *createNoeuds()* qui s'occupe, ligne par ligne, de créer les instances de tous les nœuds du graphe et de les ajouter au bâtiment. Pour cela, un scanner analyse chaque ligne du fichier *noeuds.csv* et les éléments qui s'y trouvent, séparés par des points virgules. Il peut ainsi détecter si la ligne signale un passage au prochain étage à construire ou bien la description d'un nœud à créer. Ensuite, il crée le nœud correspondant aux informations données par la ligne.

La deuxième fonction est *initVoisins()*, qui permet de construire les liens entre les nœuds qui ont été créés. Selon le nœud en cours de traitement, elle enrichit la map des voisins avec les nœuds listés et leurs distances, selon la syntaxe vue dans la section précédente.

### 3.1.3 Réutilisabilité

Dans le cadre de ce projet, nous avons travaillé sur le bâtiment de la Halle aux Farines dont nous avons les plans. Bien entendu, il est souhaitable que cette application soit utilisable pour d'autres bâtiments. Dans cette perspective, malgré nos idées et implémentations initiales trop spécifiques au seul bâtiment de la Halle aux Farines, nous avons progressivement rendu le code réutilisable.

La structure de classes permet de modéliser n'importe quel graphe de bâtiments grâce à sa modularité. Il suffit d'avoir les données particulières du graphe dans des fichiers *csv* formatés selon le modèle présenté plus haut. Le parseur est réalisé de sorte qu'il n'ait besoin que du bâtiment et du dossier contenant ces informations pour construire le graphe.

Par ailleurs, nous avons inclus dans la vue la possibilité de choisir un bâtiment, actuellement restreinte à la Halle aux Farines. Cet aspect sera détaillé dans le paragraphe **Sélection du Bâtiment**.

## 3.2 Algorithme du chemin le plus court

### 3.2.1 Déroulement de l'algorithme

Maintenant que l'on sait comment est modélisée l'université, nous pouvons considérer le bâtiment comme un graphe pondéré. Nous allons nous concentrer sur les chemins qui relient deux salles, ce qui nous intéresse étant bien évidemment de trouver le plus court. Pour cela, nous utiliserons un algorithme bien connu en mathématiques qui se nomme «algorithme de Dijkstra».

Nous avons choisi de modéliser cet algorithme dans une classe *Dijkstra*. Elle a la particularité d'être utilitaire et n'a donc pas de constructeur. C'est la méthode *trouverCheminPlusCourt* qui va dérouler l'algorithme entre le sommet de départ et le sommet d'arrivée.

Premièrement, le but est de trouver quelle distance sépare chaque sommet du graphe du sommet de départ. Pour cela, on commence par initialiser une *Map<Nœud, Double>* que l'on va appeler **map des distances**. A la fin, cette map des distances va représenter une liste de couples. Ces couples seront composés d'un sommet du graphe et de la distance la plus courte qui sépare ce sommet du sommet de départ. Tous les sommets accessibles depuis le sommet de départ seront représentés dans cette Map. En parallèle, on initialise une *List<Nœud>* que l'on appelle **liste de priorité**. Elle aura comme premier élément le sommet le plus proche du sommet de départ trouvé pour le moment. Ainsi, les nœuds de cette liste sont classés par ordre croissant de leurs distances avec le sommet de départ. Enfin, on crée une *Map<Nœud, Nœud>* que l'on appelle **map des prédécesseurs**. Cette map des prédécesseurs contiendra un sommet, ainsi que le voisin de ce sommet le plus proche du sommet de départ.

Dans l'algorithme de Dijkstra, toutes les distances sont initialisées à l'infini. En java, ce n'est pas possible donc nous allons considérer la valeur *Double.MAX\_VALUE* à la place, sachant que cette valeur est à peu près 1,79E+308, ce qui ne pose aucun problème, car aucune distance de chemin dans le contexte de ce projet ne devrait dépasser cette valeur. Il s'agit du rôle de la méthode *initNoeudList*. Bien sûr, la distance avec le sommet de départ est initialement à zéro et il est donc au début de notre liste de priorité.

Jusqu'à ce que la liste de priorité soit vide, on effectue les étapes suivantes. A chaque tour de boucle, on examine tous les voisins du premier nœud de la liste de priorité. On met à jour leur distance. Pour calculer leur nouvelle distance, on regarde quelle distance sépare le sommet actuel du sommet de départ, puis on ajoute à cela la distance qui sépare le sommet actuel du sommet voisin. Si on obtient une distance plus courte pour aller sur un voisin en passant par ce sommet, alors la distance qui sépare ce voisin du sommet de départ est mise à jour, et sa position dans la liste de priorité est modifiée. On cherche cette nouvelle position par dichotomie, grâce à la méthode *actualiserNoeudOrdreCroissant*. Ensuite, on ajoute à la map des prédécesseurs le fait qu'en passant par ce sommet, on arrive plus rapidement au sommet voisin en question. Enfin, on supprime le sommet qui vient d'être traité de la liste de priorité. En effet, il vient d'être traité, et ne sera par conséquent pas retraité une deuxième fois, à moins qu'il y ait une autre manière d'accéder à ce sommet qui se révèle plus courte.

Il y a quelques cas particuliers. En effet, certains sommets nécessitent une autorisation pour y accéder, comme les ascenseurs. Avant de mettre à jour la distance qui sépare un sommet du sommet de départ, on vérifie si on a la permission nécessaire pour se rendre sur ce sommet. Si ce n'est pas le cas, ce noeud restera bloqué à une distance infinie du noeud de départ et ne sera pas ajouté à la liste de priorité, et donc ne sera pas traité. De même, si un sommet est une salle, on considère qu'un chemin ne doit pas passer par elle, dans le cas où elle serait fermée ou occupée. On fait en sorte, tout comme pour les ascenseurs, que les salles restent à une distance infinie de la salle de départ. Bien entendu, ces exceptions ne concernent pas les sommets de départ et d'arrivée. Une fois la boucle terminée, nous avons donc traité tous les sommets accessibles depuis le sommet de départ.

A la fin de l'algorithme, on reconstruit le chemin grâce à la méthode *reconstruireChemin*. Pour cela on parcourt le chemin à l'envers : on part du sommet d'arrivée, puis on regarde lequel de ses voisins est le plus proche du sommet de départ. On ajoute ce sommet au chemin, puis on recommence avec ce sommet, et ce, jusqu'à arriver au départ. Pour faire cela, il n'y a pas besoin de vérifier chaque voisin un par un puisque la map des prédécesseurs connaît déjà quel sommet voisin est le plus proche du sommet d'arrivée. Il est possible que celui-ci n'ait aucun prédécesseur. Cela signifie alors qu'il n'existe aucun chemin qui part de ce sommet de départ et se termine à ce sommet d'arrivée et le programme retourne *null*. Le chemin le plus court entre les deux sommets est ainsi établi.

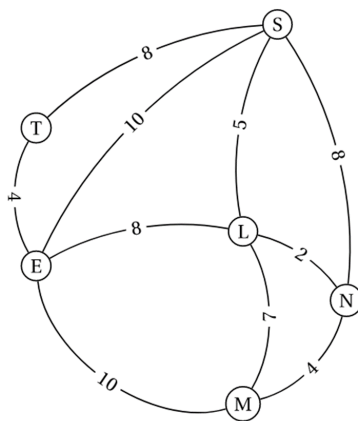


Figure 15: Graphe pour l'exemple de l'algorithme.



Cas traité :	Informations	S	T	M	L	N	E
Initialisation	Distance de S	0	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$
	Liste de priorité	[S]					
Nœud de départ (S)							
Voisin 1 (T)	Distance de S	0	8	$\infty$	$\infty$	$\infty$	$\infty$
	Prédécesseur le plus proche de S		S				
	Liste de priorité	[T]					
Voisin 2 (E)	Distance de S	0	8	$\infty$	$\infty$	$\infty$	10
	Prédécesseur le plus proche de S		S				S
	Liste de priorité	[T,E]					
Voisin 3 (L)	Distance de S	0	8	$\infty$	5	$\infty$	10
	Prédécesseur le plus proche de S		S		S		S
	Liste de priorité	[L,T,E]					
Voisin 4 (N)	Distance de S	0	8	$\infty$	5	8	10
	Prédécesseur le plus proche de S		S		S	S	S
	Liste de priorité	[L,N,T,E]					
Nœud (L)							
Voisin 1(M)	Distance de S	0	8	12	5	8	10
	Prédécesseur le plus proche de S		S	L	S	S	S
	Liste de priorité	[N,T,E,M]					
Voisin 2 (E)	Distance de S	0	8	12	5	8	10
	Prédécesseur le plus proche de S		S	L	S	S	S
	Liste de priorité	[N,T,E,M]					
	Commentaire	Pour ce voisin il ne se passe rien, car la nouvelle distance calculée est 12 pour arriver à E. Donc rien n'est mis à jour.					
...							
Résultat final :	Distance de S	0	8	10	5	7	10
	Prédécesseur le plus proche de S		S	N	S	L	S
	Liste de priorité	[]					
	Chemins les plus courts :	De M vers S : M->N->L->S On peut inverser ce chemin obtenue : De S vers M : S->L->N->M					

Figure 16: Détail des étapes de l'algorithme de Dijkstra.

### 3.2.2 Optimisation

La première étape de simplification de cet algorithme a été mentionné dans la description des classes du modèle. En effet, initialement, il était prévu que l'algorithme s'applique entre deux portes de *Salle*. Cependant, si la salle de départ avait trois portes, et celle d'arrivée aussi, l'algorithme aurait tourné 9 fois pour trouver le chemin le plus court pour chaque combinaison de portes, puis il aurait fallu sélectionner le chemin le plus court parmi ces 9 itinéraires. La simplification des salles permet de ne faire tourner l'algorithme qu'une seule fois entre deux salles.

Nous avons essayé d'optimiser la complexité de l'algorithme en gardant la liste de priorités toujours triée et en rangeant les éléments, lorsqu'ils doivent l'être, de manière dichotomique. En

faisant ainsi, on évite de chercher de manière linéaire le sommet avec la distance la plus courte depuis le sommet de départ. Ce qui aurait donné une complexité dans le pire cas de l'ordre de  $\theta(n^2)$ , avec  $n$  le nombre de sommets de l'arbre. Au lieu de cela, lorsque l'on range le sommet dans la liste de priorité, on obtient une complexité  $\theta(\log(n))$ . De plus, lorsque l'on cherche le prochain sommet à traiter, on prend le premier de la file de priorité, ce qui donne une complexité de  $\theta(1)$ .

### 3.3 Affichage graphique

Une fois les différentes classes du modèle implémentées, nous nous sommes attelés à réaliser l'affichage graphique de notre logiciel.

Notre application est composée de deux écrans, l'écran d'accueil et l'écran principal, entre lesquels l'utilisateur navigue avec des boutons. Ils comportent tous deux un panneau de contrôle permettant de lancer une recherche d'itinéraire ou de salle. C'est sur l'écran principal que l'utilisateur a accès aux plans du bâtiment.

#### 3.3.1 Classes de la vue

**Vue** La classe centrale de la vue du projet est la classe *Vue*, héritant de *JFrame*, qui s'occupe de la transition entre les deux écrans qui composent notre programme, *Home* et *MainApp*, nommés respectivement *accueil* et *app*. Elle possède également un attribut *control* correspondant au contrôleur du programme. Des entiers finaux *WIDTH* et *HEIGHT* définissent les dimensions de la fenêtre, tandis que *listImages*, une *ArrayList* de *BufferedImage* contient les plans du bâtiment actuel. Enfin, la classe *Vue* contient deux méthodes importantes : *resetApp* et *majApp*. La première s'occupe de passer de l'écran principal à l'écran d'accueil, tandis que la deuxième sert à passer de l'accueil à la page principale ou à construire une nouvelle app à partir des entrées de l'utilisateur.

**Fenetre** La classe *Fenetre* est une classe abstraite héritant de *JPanel* représentant un écran de l'application. Elle contient un *JPanel* *controlPanel* auquel on ajoute les attributs suivants :

- deux *TextFieldBox*, *start* et *finish*, permettant à l'utilisateur de saisir sa recherche;
- une *JCheckBox* *ascenseur*, permettant à l'utilisateur de spécifier s'il veut utiliser les ascenseurs du bâtiment;
- un *GoButton* permettant de lancer la recherche.

*Fenetre* possède aussi un attribut *view* et un attribut *control*, correspondant respectivement à la vue et au contrôleur du programme.

**TextFieldBox** La classe *TextFieldBox* hérite de *JTextField*. Son texte par défaut s'efface quand le focus est gagné et réapparaît quand ce dernier est perdu. Lorsque la touche ENTER est saisie en ayant le focus, elle a le même rôle qu'un *GoButton*. De plus elle possède l'attribut *view*, la *Vue* du programme, afin de vérifier la validité de la salle saisie via la méthode *estSalle*.

**GoButton** La classe *GoButton* étend *JButton* et utilise la méthode *majApp* de la *Vue*, utilisée comme paramètre dans le constructeur, afin de créer une nouvelle *MainApp* lorsqu'on clique dessus, ou qu'on appuie sur ENTER quand le focus est sur lui.

**Menu d'accueil** La classe *Home*, héritant de *Fenetre* représente un menu d'accueil sobre avec son panel de contrôle *controlPanel* au milieu et le logo de l'université au-dessus, stocké dans le *JLabel logoLabel*.

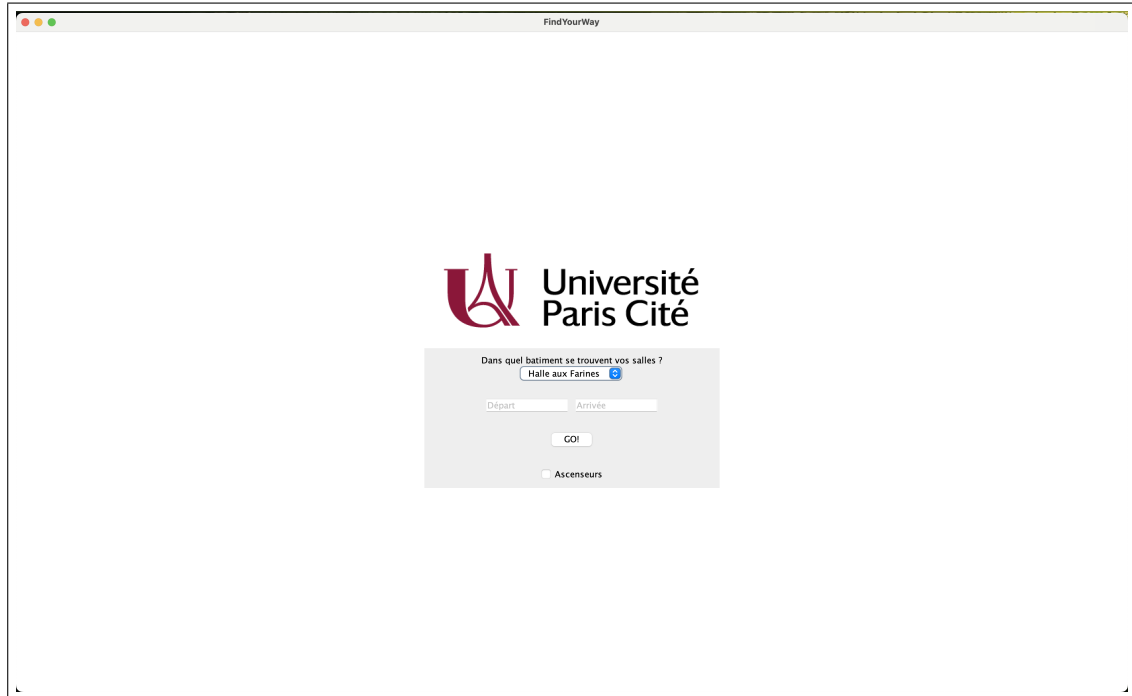


Figure 17: Page d'accueil de l'application

**Page principale** La classe *MainApp* correspond à l'écran principal de l'application et hérite de la classe *Fenetre*. Son *controlPanel* est placé au haut de l'écran et contient aussi le logo de l'université. Sur la droite, on trouve le panel permettant de changer d'étage et les indications. Sur le reste de l'écran s'affichent les plans. La classe contient *listImages* de type *ArrayList<BufferedImage>* (la même que celle de la classe *vue*) contenant les plans du bâtiment actuel. Les attributs *cheminActuel* et *salle*, quand ils ne sont pas *null*, correspondent au résultat de la recherche de l'utilisateur, indiquant alors quoi dessiner sur les plans.

Le *JPanel etagesPanel*, permettant de changer d'étage, est muni de deux *JButton*, *upButton* et *downButton*, permettant de changer d'étage, ainsi qu'un *JLabel etageLabel* indiquant l'étage actuel. Ensuite, la classe *MainApp* possède un attribut *PlanPanel planPanel* chargé d'afficher les plans.

Nous avons ajouté à la classe *MainApp* une *AbstractAction* attendant la combinaison de touches **CTRL+D** pour changer la valeur de l'attribut booléen *debug* et ainsi afficher le graphe entier du bâtiment.

A propos des méthodes, *afficherChemin* met à jour la vue en dessinant le chemin calculé selon les entrées saisies par l'utilisateur. S'il n'y a qu'une entrée saisie ou valide, *afficherPortes* met à jour la vue en dessinant les portes de cette salle.

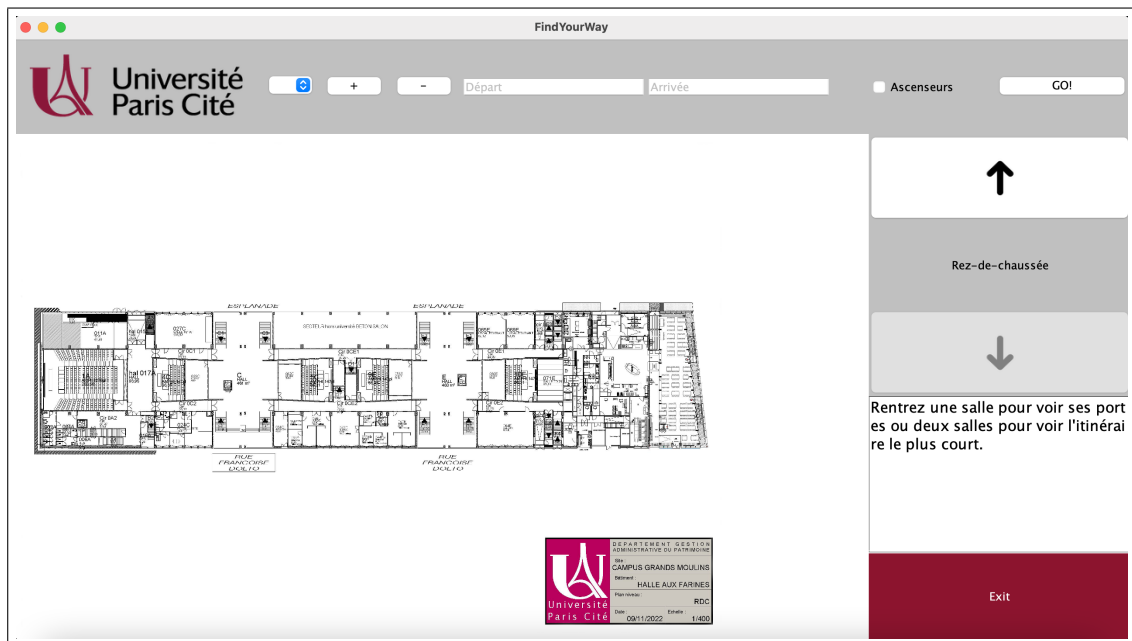


Figure 18: Page principale de l'application

### 3.3.2 Affichage des noeuds et chemins

La classe *PlanPanel* hérite de *JPanel* et permet l'affichage des plans et le dessin des chemins, portes et graphes dessus.

L'image affichée est stockée dans *image* de type *BufferedImage* et fait partie d'une des listes d'images disponibles. On en compte deux : *blankPlans* qui contient les plans vierges de toute modification et *debugPlans*, qui contient les plans sur lesquels sont dessinés les graphes. Cette dernière est utilisée pour le mode debug par *drawDebug*. À l'inverse, la méthode *eraseDebug* se sert de *blankPlans* pour afficher l'image sans son graphe.

Pour dessiner sur l'image et afficher un chemin, on utilise *drawPath* : on parcourt la liste de nœuds du chemin actuel et on dessine une ligne entre chaque nœud se trouvant à l'étage actuel. Pour les portes d'une salle, on utilise *drawRoom* : on parcourt les voisins du nœud représentant la salle, qui sont donc ses portes, et on dessine un point à leur emplacement s'ils appartiennent à l'étage actuel. La classe redéfinit *paintComponent* afin de redessiner l'image à chaque modification.

Les chemins et points qui apparaissent sur les plans clignotent en rouge. *PlanPanel* implémente *Runnable* dans ce but, et comporte un booléen *increasing* indiquant si les dessins apparaissent ou disparaissent. Elle redéfinit aussi *run* afin de faire varier l'attribut *alpha* en fonction de *increasing*, qui correspond au degré de transparence des tracés sur les images.

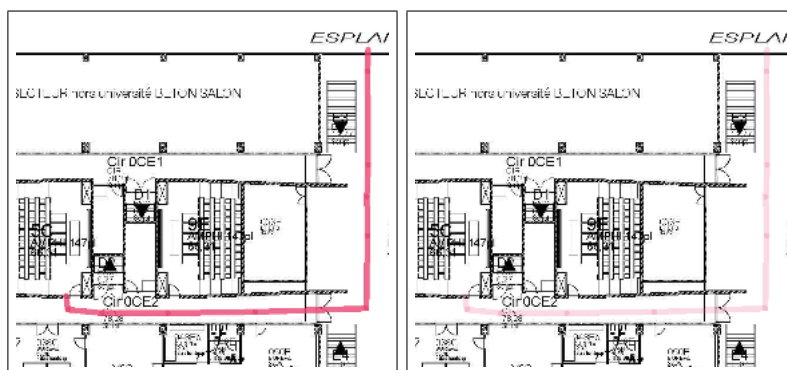


Figure 19: Un chemin apparaît et disparaît.

On peut aussi noter les attributs entiers *viewX*, *viewY*, ainsi que les double *scale* et *maxScale*, utilisés lors des calculs pour redessiner les images et s'assurer que l'image occupe tout le panel. Leurs noms témoignent de notre volonté initiale d'implémenter le zoom sur les plans.

### 3.3.3 Entrées utilisateurs

Ce sont donc les entrées utilisateur qui permettent de déterminer si l'on doit et de quelle manière il faut mettre à jour la vue.

Pour vérifier qu'il est possible de mettre à jour la vue, on utilise la méthode *verifGoButton* de la classe *Controller*. Celle-ci vérifie que les textes contenus dans les *TextFieldBox* sont soit des noms de salles existantes dans le bâtiment, soit "Départ" pour la première et "Arrivée" pour la seconde ou bien des textes blancs. On vérifie qu'un nom est bien celui d'une salle du bâtiment avec la méthode *estSalle* et s'il est blanc avec la méthode *isBlank*.

Si la méthode *verifGoButton* renvoie faux, alors le fait d'appuyer sur le bouton Go de la fenêtre change la couleur du ou des textes invalides dans les *TextFieldBox*. Dans le cas contraire, la vue se met à jour en prenant en compte les éventuelles salles de départ et d'arrivée ainsi que la permission d'utiliser les ascenseurs déterminée par la *JCheckBox*. Quand la vue se met à jour, les textes présents dans les *TextFieldBox* sont conservés dans une couleur plus claire pour se rappeler du chemin demandé ou bien remplacés par "Départ" ou "Arrivée" si ces textes étaient blancs. La *JCheckBox* garde aussi le même état qu'avant la mise à jour de la vue pour la même raison que la conservation des noms de salles.

Il faut également noter que pour les salles comme les toilettes ou les machines à café qui sont physiquement à plusieurs endroits, il n'est pas possible de débiter un chemin à partir de ces salles. En effet, puisqu'on ne peut pas sélectionner le point de départ exact, on se retrouverait à toujours partir de la porte la plus proche de l'arrivée ce qui n'est pas toujours ce que l'utilisateur souhaite.

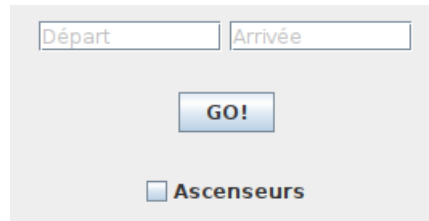


Figure 20: Panel de contrôle pour les saisies utilisateur sur la page d'accueil

**Contrôleur** Le contrôleur faisant le lien entre le modèle et la vue, la classe *Controller* possède donc deux attributs : une Vue *vue* et un Batiment *batActuel*. Le *Controller* a pour fonction principale de mettre à jour la vue selon les entrées utilisateur dans les *TextFieldBox* et la *JCheckBox* de la vue. C'est la méthode *majApp* qui se charge d'effectuer cette action. On différencie plusieurs cas de figure :

- Les deux *TextFieldBox* sont vides, c'est à dire que la première contient "Départ" et la seconde "Arrivée" ou bien qu'elles contiennent des textes blancs (espaces, tabulations...). Dans ce cas, seuls les plans du bâtiment sont affichés dans le *PlanPanel* de la vue.
- Une seule des deux *TextFieldBox* contient le nom d'une salle du bâtiment et l'autre est vide, ou bien les deux *TextFieldBox* contiennent le même nom de salle : on affiche alors la ou les portes de la salle en question avec des points sur les plans aux endroits nécessaires.
- Les deux *TextFieldBox* contiennent le nom de salles différentes : lorsque l'on met à jour la vue, un chemin indiquant le trajet le plus court entre les deux salles est tracé sur les plans.
- Au moins une des deux *TextFieldBox* contient le nom d'une salle qui n'existe pas. La vue ne se met donc pas à jour.

### 3.3.4 Indications

Pour une question de lisibilité, on souhaite avoir accès à tout le texte, quelle que soit sa longueur. Initialement, nous avons implémenté cette fonctionnalité avec un simple *JPanel* et un *JTextArea* à l'intérieur, dans le panneau de contrôle à droite de l'écran. Puis, en se rendant compte par des essais que pour des chemins assez longs, le texte n'était pas entièrement visible, nous avons choisi de changer l'implémentation pour un *JScrollPane* permettant à l'utilisateur de faire glisser le texte de haut en bas s'il sort du cadre alloué. De plus, pour une question d'ergonomie, on ne veut pas avoir à scroller de gauche à droite pour voir la fin de chaque ligne, ce qui est le cas si nous laissons l'implémentation telle quelle. Nous avons donc ajouté une ligne de code pour préciser au texte qu'il doit être justifié avec *setLineWrap(true)*.

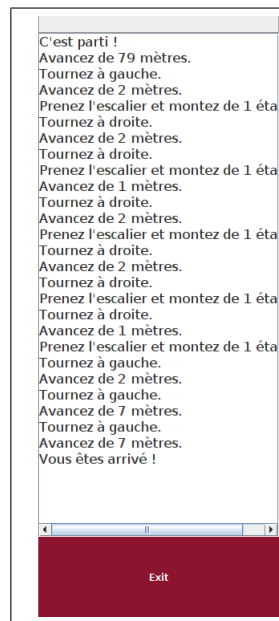


Figure 21: Un texte d'indications est visible entièrement grâce au JScrollPane (sans `setLineWrap`)

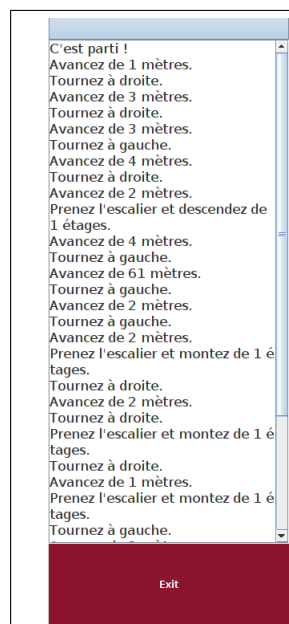


Figure 22: Un texte d'indications est visible entièrement grâce au JScrollPane, avec le `setLineWrap`.

### 3.3.5 Itinéraires favoris

Nous avons implémenté la possibilité pour l'utilisateur de sauvegarder des salles favorites de départ, et/ou d'arrivée. Ainsi, au lieu de rentrer à nouveau les salles, il n'a qu'à les sélectionner dans le menu déroulant des favoris.

Pour cela, on a défini la classe *Binôme* qui contient deux nœuds, un nœud de départ, et un nœud d'arrivée. Le modèle de la liste des favoris contient une *ArrayList<Binôme>*, qui indique les nœuds qui ont été enregistrés. La view qui étend *JPanel*, possède un *JComboBox<Binôme>*. Ainsi chaque fois que l'on ajoute un élément au modèle, on met à jour cette *JComboBox*. Les binômes s'affichent dans cette *JComboBox* grâce à la redéfinition de *toString* que l'on a fait pour la classe *Binôme*. Ainsi, si un nœud de départ et un nœud d'arrivée ont été enregistrés, alors ils apparaissent sous la forme : **départ ==> arrivée**. Si un seul nœud a été ajouté, le binôme sera composé du nœud et d'un nœud *null*. Il apparaît sous la forme : **nœud**.

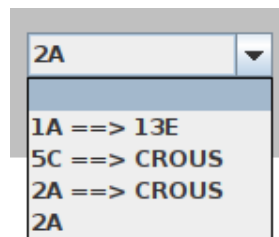


Figure 23: Menu déroulant des favoris. On y retrouve la salle 2A ainsi que trois itinéraires : de la 1A à la 13E, de la 5C au crous et de la 2A au crous.

On a ensuite ajouté deux *JButton* : le premier pour ajouter des favoris, le deuxième pour en supprimer. On a ajouté à ses boutons des *actionlistener* afin qu'à chaque fois qu'on appuie dessus, cela appelle le contrôleur et met à jour les informations en fonction de ce que l'on fait.

Enfin, lorsque l'on sélectionne un favori dans la *JComboBox*, les cases de saisie des salles se remplissent automatiquement : il ne reste plus qu'à sélectionner si on a ou non la possibilité d'utiliser un ascenseur et d'appuyer sur « GO ».

### 3.3.6 Sélection du bâtiment

Actuellement, seule la Halle aux Farines est disponible dans notre programme car nous n'avons que ses plans et donc ses données. Toutefois, il est possible de rajouter des bâtiments et d'y chercher des itinéraires avec le même fonctionnement. Pour cela, nous avons ajouté sur l'écran d'accueil la possibilité de choisir le bâtiment qui intéresse l'utilisateur. Initialement pensé comme une case à cocher, nous avons finalement choisi d'utiliser un *JComboBox* qui permet de ne sélectionner qu'un *Batiment* à la fois. Grâce à un *actionListener*, le bâtiment sélectionné et ses plans sont directement mis à jour dans le contrôleur avec la méthode *setBatiment* du *Controller* et *initListImages* de la *Vue*, de sorte qu'une fois qu'on lance l'écran principal avec le bouton *GO*, le bâtiment construit est le bon. Si l'utilisateur souhaite changer de bâtiment, il peut revenir à l'écran d'accueil grâce au bouton *Exit* situé en bas à droite de l'écran principal. Les entrées de salle et d'ascenseur qui étaient sélectionnées à ce moment-là sont gardées en mémoire dans les *TextFieldBox* afin que l'utilisateur relance la même recherche ou bien qu'il puisse en faire une nouvelle dans un autre *textitBatiment*.



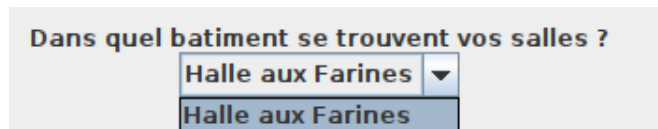


Figure 24: Le menu déroulant permettant de choisir le bâtiment

## 4 Fonctionnalités non implémentées

Ce qui suit regroupe les réflexions et pistes de développement possibles et envisagées au cours de notre projet. Certaines idées n'ont pas été retenues mais pour d'autres, nous avons implémenté notre code pour qu'il puisse les accueillir.

### 4.1 Autres bâtiments

Si le programmeur souhaite rajouter des bâtiments, il a quelques étapes à respecter. Il faut d'abord fournir un dossier de plans à afficher, avec autant d'images que d'étages et étant nommé selon l'étage qu'ils représentent de la manière suivante :

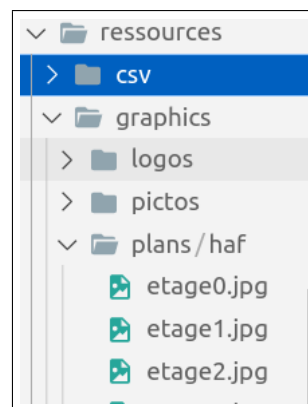


Figure 25: Les plans de la Halle aux Farines se situent dans le dossier */ressources/graphics/plans/haf*

Bien entendu, il doit également fournir les *csv* du bâtiment pour pouvoir construire les nœuds du graphe et leurs liens et les mettre dans le dossier *csv*.

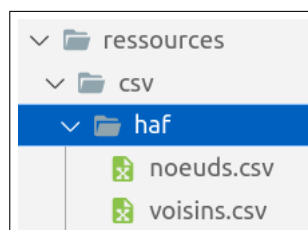


Figure 26: La Halle aux Farines a un dossier de *csv* dans */ressources* contenant les deux fichiers nécessaires à la construction de son graphe par le parseur.

Enfin, à la construction de la page d'accueil *Home*, il faut ajouter à la liste de ses bâtiments le nouveau bâtiment avec ses attributs, ainsi que les path des dossiers énoncés plus haut. Il sera ainsi ajouté dans la liste du JComboBox et disponible au choix de l'utilisateur. Ces manipulations se font dans la méthode *initBatiments()*.

Cette fonctionnalité assure donc la réutilisabilité du code. Avant ces modifications, il fallait construire le bâtiment de la Halle aux Farines dans *Main* puis le passer en attribut au constructeur du *Controller* qui le définissait comme *batActuel*. Désormais, *Main* ne fait que construire un controller sans argument et les étapes de construction et de sélection du bâtiment actuel se font comme indiqués, à l'aide de l'interface utilisateur.

## 4.2 Faciliter le formatage des données

Reporter à la main les informations de chaque nœud des fichiers *.ggb* aux fichiers *.csv* a pris beaucoup de temps. En plus d'être pénible et long, favorisant des erreurs de frappe, ce n'est pas une méthode optimale de procéder, surtout s'il faut répéter la tâche pour d'autres bâtiments, qui pourraient être encore plus gros. Ainsi, nous avons songé en cours de tâche à mieux exploiter les outils de Geogebra, notamment faire en sorte que les coordonnées et types de points soient rapportés dans un tableur, puis d'exporter celui-ci directement comme un fichier *.csv*. Cela n'a pas été effectivement réalisé dans le cadre de notre projet mais il s'agit d'une piste à exploiter pour un éventuel développement de ce code.

## 4.3 Zoom du plan

Nous avons envisagé d'ajouter la possibilité de zoomer sur les plans à l'aide de la souris, pour assurer une bonne visibilité. Pour ce faire, la classe *PlanPanel* implémentait *MouseListener* et *MouseMotionListener* et leurs méthodes respectives. L'idée était de permettre de zoomer avec la molette de la souris et de se déplacer sur les plans en maintenant le clic appuyé, tout en restant dans les limites de l'image.

Finalement, en plus de représenter une tâche ardue à cause du système de coordonnées, l'implémentation du zoom ne s'est pas révélée nécessaire à la compréhension des plans et itinéraires, et a donc été reléguée à la fin de notre liste de priorités.

## 4.4 Sélection sur le plan

Durant le développement, quelque chose a retenu notre attention : puisque chaque toilette dans le bâtiment correspond à un seul et même nœud dans notre modélisation, l'utilisateur ne peut pas démarrer un itinéraire dans des toilettes particulières. En effet, le programme sélectionne automatiquement la porte de toilettes la plus proche de la destination. Nous avons donc envisagé de rendre les nœuds du graphe cliquables sur les plans, afin de lancer un itinéraire entre deux nœuds quelconques du bâtiment.

La réalisation de cette tâche posait deux problèmes. D'abord, cela nous aurait forcé à afficher en permanence l'entièreté des Carrefour de l'étage actuel, surchargeant graphiquement le logiciel et rendant la lisibilité mauvaise. De plus, cela aurait nécessité de construire et placer autant d'objets cliquables qu'il n'y a de nœuds dans le graphe, ce qui est une tâche plutôt ardue. En guise de comparaison, l'affichage effectif des nœuds et liens via le mode debug n'est que le résultat du dessin sur les plans de ces derniers, sans interaction rendue possible avec eux.

Nous avons donc conclu qu'il n'était pas rédhibitoire d'empêcher l'utilisateur de saisir les toilettes comme point de départ.

## 4.5 Autocomplétion

Une piste d'amélioration de notre programme est d'implémenter l'autocomplétion dans les boites de saisie de texte. Pour cela, il faudrait les combiner avec une JComboBox contenant le nom de chaque salle, la liste s'affinant au fur et à mesure que le mot saisi par l'utilisateur se rallonge.

## 4.6 Historique

Une autre amélioration serait la possibilité de sauvegarder les précédentes recherches afin de les retrouver facilement, qui a été écartée au profit du système de favoris.

## 4.7 Inversion de l'itinéraire

Tardivement, l'idée de rajouter une fonctionnalité pour inverser le départ et l'arrivée saisis par l'utilisateur nous est venue. Non implémentée, il suffit cependant d'ajouter un bouton dans le panneau de contrôle qui, quand cliqué, permet de récupérer le texte dans chaque TextFieldBox et de l'attribuer à l'autre, puis de mettre à jour la vue.

# 5 Annexes

## 5.1 Diagramme de structure de dossiers et fichiers

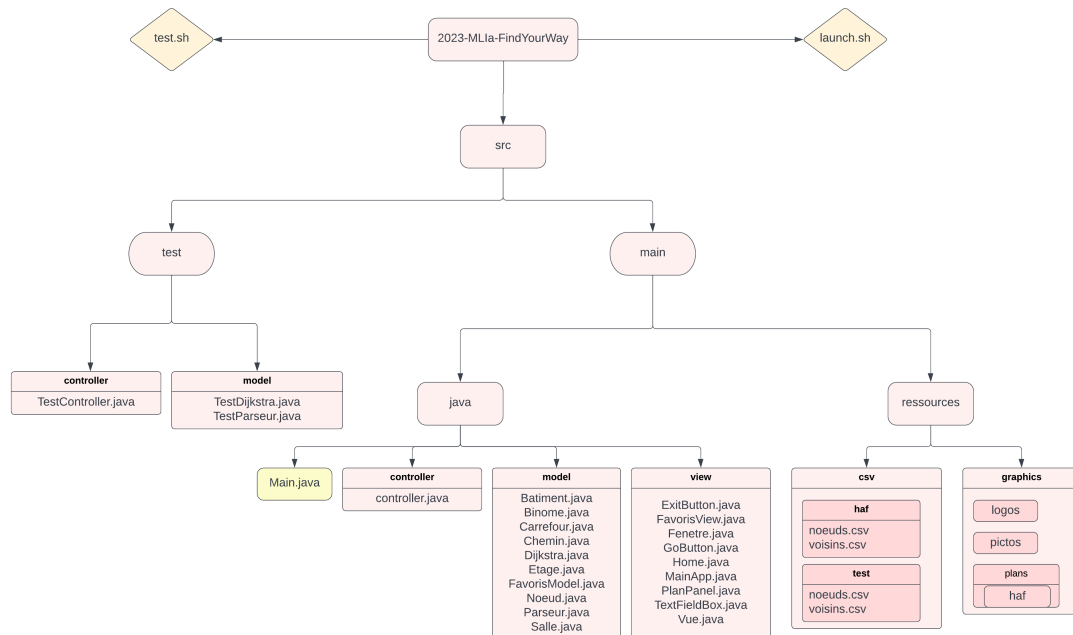


Figure 27: Structure du projet

## 5.2 Diagramme de classes

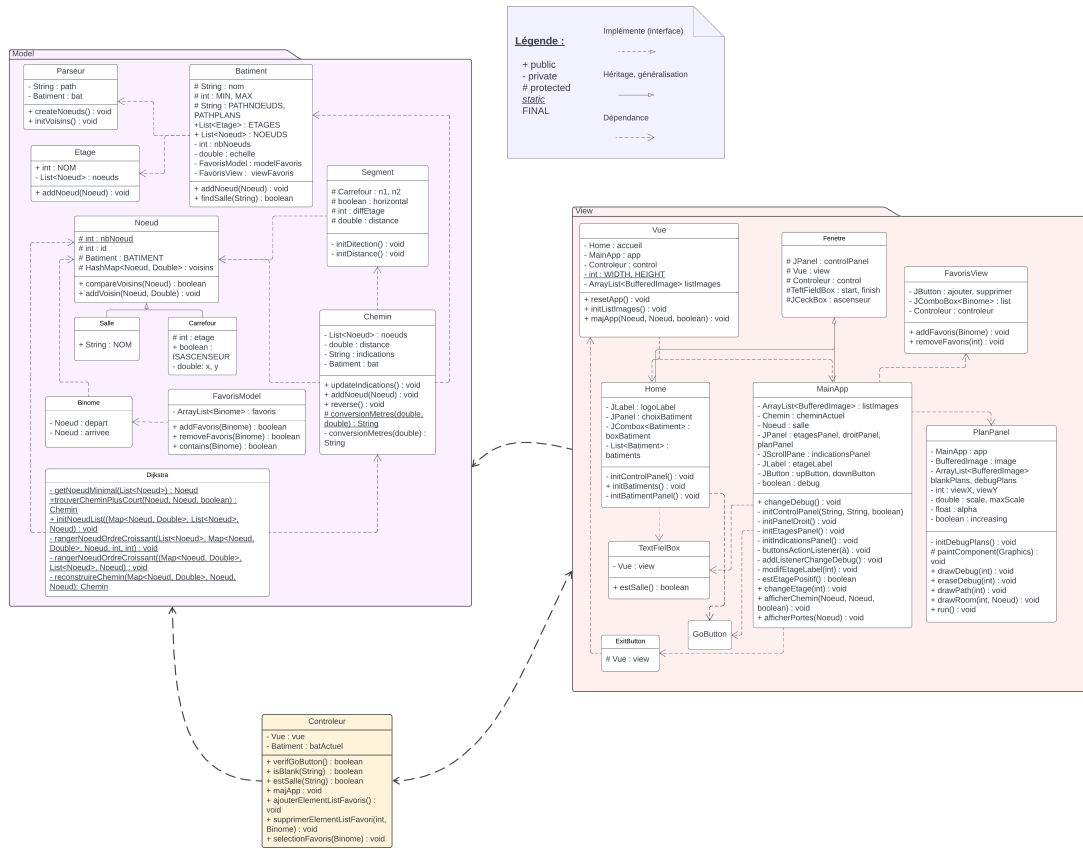


Figure 28: Diagramme UML des classes du projet

## 5.3 Guide d'utilisation

**Configuration requise** Avant tout, assurez vous d'avoir la version 11 de Java et d'avoir accès à la commande **bash**.

La commande **bash** vient généralement avec des OS UNIX. Si vous êtes sur Windows, utilisez Git Bash.

**Instructions** Pour démarrer le jeu, depuis le dossier *2023-ML1a-FindYourWay*, utilisez la commande **bash launch.sh** ou **./launch.sh**.

Après avoir exécuté la commande ci-dessus, une fenêtre apparaîtra sur votre écran.

**Démarrage** Au démarrage de l'application, vous trouverez un menu où vous pourrez renseigner plusieurs informations :

- le bâtiment dans lequel vous souhaitez obtenir un chemin, que vous pourrez sélectionner à l'aide d'un menu déroulant;

- la salle de départ et de fin du chemin, à saisir manuellement;
- un bouton 'GO', sur lequel vous pourrez appuyer une fois vos choix établis;
- une case à cocher pour indiquer si vous avez la permission d'accès aux ascenseurs.

Une fois le bouton 'GO' cliqué, la page va changer. Les plans vont apparaître et vous aurez accès à de nouvelles fonctionnalités.

**Après avoir appuyé sur 'GO'** Si vous aviez saisi une salle de départ et une salle d'arrivée, alors vous pourrez voir un chemin dessiné sur les plans ainsi que les directions à suivre à droite. Si vous n'avez saisi qu'une seule salle, vous verrez un ou plusieurs points apparaître au niveau des portes de la salle en question. Si vous n'avez rien saisi, vous aurez juste accès au plan.

**Autres fonctionnalités** A présent, en haut à droite, vous avez accès à deux boutons où des flèches sont dessinées, vous permettant de naviguer entre les étages. La flèche du haut permet de changer le plan par celui de l'étage supérieur, la flèche du bas, par celui de l'étage inférieur. En haut, vous avez toujours les mêmes éléments présentés précédemment, c'est à dire :

- deux emplacements pour saisir les salles de départ et de fin du chemin;
- un bouton 'GO', sur lequel vous pourrez appuyer une fois vos choix établis;
- une case à cocher pour indiquer si vous avez la permission d'accès aux ascenseurs.

Un nouvel élément fait son apparition : les favoris. En effet en haut à gauche, vous avez trois éléments qui sont apparus:

- un JComboBox qui permet de sélectionner une salle ou un chemin préalablement ajouté à la liste des favoris;
- un bouton '+' qui permet d'ajouter aux favoris les salles sélectionnées dans les emplacements de saisie;
- un bouton '-' qui permet de supprimer le favori actuellement sélectionné.

Lorsque vous sélectionnez un favori, cela remplace automatiquement les emplacements de saisie de salle de départ et d'arrivée. Vous devrez donc sélectionner si vous souhaitez ou non utiliser un ascenseur, et appuyer sur 'GO'.

## 5.4 Bibliographie

Voici les ressources que nous avons utilisées pour appréhender l'algorithme de Dijkstra :

- Algorithme de Dijkstra - Wikipedia
- Cours sur les graphes de visibilité - C. Nguyen
- Cours préparatoire sur l'algorithme de Dijkstra
- Programme NSI sur le plus court chemin dans un graphe
- Cours sur le plus court chemin

## 6 Conclusion

Nous avons atteint quasiment tous les objectifs fixés par le sujet de ce projet de programmation en groupe. Les fonctionnalités exigées sont toutes implémentées et fonctionnent, ainsi que des fonctionnalités supplémentaires. Toutefois, le sujet fait mention de tous les bâtiments du campus quand nous n'avons implémenté que celui de la Halle aux Farines. Cela est notamment dû à des causes extérieures, à savoir que seuls les plans de ce bâtiment ont pu nous être fournis. Toutefois, nous avons fait en sorte que notre programme offre la possibilité de rajouter de nouveaux bâtiments. Nous avons également présenté une liste non exhaustive des modifications qui peuvent être ajoutées afin d'offrir à l'utilisateur de nouvelles fonctionnalités.

En ce qui concerne le cadre plus général de ce projet, nous pouvons remarquer deux choses. Premièrement, ce projet constitue un nouveau travail d'équipe réussi en termes de communication et de répartition des tâches. Bien qu'il ne s'agisse pas d'une application directe des enseignements de ce semestre, il nous a permis d'être plus à l'aise dans ce genre de tâches qu'au semestre précédent où nous avions découvert des outils comme git ou l'intégration continue et des pratiques comme des réunions hebdomadaires. Nous avons remarqué une nouvelle aisance à les réappliquer avec de nouveaux membres sur un autre sujet. Secondement, en tant qu'étudiants de double licence Mathématiques - Informatique, nous avons pu avoir un exemple concret de l'association de ces deux domaines à travers l'implémentation d'un algorithme comme celui de Dijkstra et la modélisation de graphes, qui sont des concepts mathématiques importants. Ce projet a donc été une opportunité d'apercevoir des possibilités et des débouchés professionnels et universitaires qu'offre notre formation.