# Introduction aux Systèmes Multi-Agents
## Projet : SimTransport

FRANÇOIS HERNANDEZ - LÉO PONS
*CentraleSupélec*
March 21, 2017

# Table des matières

# Introduction

Un Système Multi-Agents est un système composé d'un ensemble d'agents, situés dans un certain environnement et interagissant selon certaines relations. Un agent est une entité caractérisée par le fait qu'elle est, au moins partiellement, autonome. Le fonctionnement du système est défini par le fonctionnement des agents, le système central ne fait qu'office d'environnement d'évolution et de communication des différents agents. On parle d'intelligence artificielle distribuée.

Ce projet a pour objectif de construire une simulation des transports sur le plateau de Saclay. Pour cela, on utilisera la plateforme Jade. La plateforme Jade est une plateforme répartie d'agents s'exécutant de manière asynchrone (chaque agent est une thread). Chaque agent définit ses actions à l'aide de comportements qu'il exécute les uns après les autres. La programmation de ces comportements se fait de manière à les "entrelacer' dans l'exécution de l'agent. Dans ces comportements, les agents peuvent envoyer et recevoir des messages, mais ils peuvent aussi déclencher d'autre comportements ou même créer d'autres agents.

Le but du projet est de simuler et d'étudier des politiques d'aménagement des transports à l'aide d'un simulateur multi-agent. Nous adoptons une modélisation simplifiée pour étudier quelques hypothèses et mesurer l'impact des paramètres sur la simulation.

Ce rapport fait le tour de notre travail mais ne va pas forcément au coeur des implémentations, celles-ci étant commentées à même le code.

# I Présentation générale

Nous avons donc construit un prototype de plateforme pour simuler les déplacements des utilisateurs sur le plateau de Saclay. Cette simulation n'est pas une fin en soi, l'idée est de pouvoir simuler et donc tenter prédire le comportement de ces utilisateurs (par exemple leur propension à choisir les transports en commun) en fonction de différents paramètres (par exemple la fréquence des transports en commun, ou la fermeture d'une route). L'application doit donc pouvoir être utilisée pour estimer l'impact de paramètres réels sur le traffic.

## I.1 Fonctionnement général

Le prototype comprend un simulateur des déplacements d'agents sur le plateau sur une journée. Les agents qui évoluent au cours de cette journée sont déterminés à l'avance, et ont un emploi du temps à suivre et un choix de mode de transport fixé : soit la voiture, soit les transports en commun et la marche. Au cours de la journée, des embouteillages peuvent se créer pour les voyageurs en voiture s'ils sont trop nombreux, ce qui les ralentit considérablement. Les performances des deux modes de transport (temps passé à voyager) sont relevées pendant toute la journée.

À partir de ce simulateur sont possibles deux types de déductions :

- Des estimations de performances du réseau : avec un pool d'utilisateurs donné, simplement mesurer les performances du réseau, et évaluer la criticité du blocage de certaines routes ou de la suppression de certaines lignes de bus.
- Des estimations du comportement des utilisateurs : la simulation peut être effectuée sur plusieurs jours. Les agents sont les mêmes de jour en jour, mais choisissent chaque matin d'utiliser la voiture ou les transports en commun en fonction des performances des jours précédents. On peut ainsi étudier l'impact d'un changement dans le réseau sur plusieurs jours et donc l'évolution des choix des utilisateurs.

## I.2 Structure globale du code

Le programme comporte les packages suivants :

- `Environment` : contient les différents éléments de l'environnement dans lequel se déroulera la simulation (horloge et environnement) ;
- `Environment.Points` : contient les différents types de points ;
- `Environment.Paths` : contient les différents types de chemins ;
- `Graphics` : contient les différentes classes associées à la représentation graphique de la simulation ;
- `Individuals` : contient les différentes classes et comportements liés aux individus.
- `Util` : contient la classe permettant d'importer les points et chemins stockés au format `csv` ;
- `mainPackage` : contient la classe agent `Starter` et son comportement `Days` qui permettent de créer l'environnement et les agents pour lancer la simulation.

# II Objets et structure du programme

Avant même de considérer la partie simulation multi-agents, il faut créer la structure du programme permettant de simuler le problème. La structure choisie est détaillée ci-après.

## II.1 Points

Afin de représenter les différents points utiles à notre simulation, nous créons les classes suivantes, regroupées dans le package `Environment.Points` :

- `Point` : point "classique" ne représentant pas de lieu particulier, comporte des coordonnées (`x`, `y`, `z`), un `id` entier et un nom `name` ainsi que les différents getters et setters utiles ;
- `EntryPoint` : point "source" à partir duquel les individus entreront dans la simulation, hérite de `Point` ;
- `InterestPoint` : point "d'intérêt" auquel les individus désireront se rendre, hérite de `Point` ;
- `PreEntryPoint` : point se trouvant avant les points d'entrée, afin d'éviter toute situation où un agent d'un certain type serait bloqué à un `EntryPoint` ne disposant pas de chemin compatible avec son type.

## II.2 Chemins

Nous représentons aussi différents types de chemins. Pour cela, nous avons fait le choix d'un typage fort des chemins. Nous aurons un objet par type de chemin et par sens (de A vers B). Ainsi, par exemple, pour un chemin qui peut être emprunté par des piétons et des vélos, dans les deux sens, il y aura quatre objets. Les classes sont donc définies de la manière suivante dans le package `Environment.Paths` :

- `Path` : classe mère dont hériteront tous les chemins 'typés', comporte un point `A` source et un point `B` destination, les getters et setters associés ainsi que différentes méthodes permettant de récupérer la distance (dans l'espace et projetée sur le plan), le dénivelé et la pente ;
- `FootPath` : chemin piéton héritant de `Path` et comportant une méthode `weight()` retournant le temps de parcours nécessaire (pour une vitesse moyenne de 5km/h, et prenant en compte la difficulté accrue pour un dénivelé positif) ;
- `RoadPath` : chemin routier (agglomération) héritant de `Path` et comportant une méthode `weight()` retournant le temps de parcours nécessaire pour une vitesse moyenne de base de 35 km/h, la durée étant exponentiellement allongée lorsque la densité d'utilisateurs sur la voie dépasse un seuil critique de 1 utilisateur pour 10m (embouteillages) ;
- `HighwayPath` : chemin routier (nationale) héritant de `Path` et comportant une méthode `weight()` retournant le temps de parcours nécessaire pour une vitesse moyenne de base de 90 km/h, la durée étant exponentiellement allongée lorsque

la densité d'utilisateurs sur la voie dépasse un seuil critique de 2 utilisateurs pour 10m (embouteillages) ;

- `BusPath` : chemin représentant un autobus (de vitesse moyenne 50 km/h), héritant de `Path` et comportant un temps d'attente moyen et une méthode `weight()` retournant le temps de parcours nécessaire (calculé à partir de la vitesse et du temps perdu aux arrêts) ;
- `RerPath` : chemin représentant un RER (de vitesse moyenne 80 km/h), héritant de `Path` et comportant un temps d'attente moyen et une méthode `weight()` retournant le temps de parcours nécessaire (calculé à partir de la vitesse et du temps perdu aux arrêts) ;
- `EntryPath` : chemin entre les `PreEntryPoint` et `EntryPoint`, de poids nul (un agent débutera, sans délai, au point d'entrée le plus proche correspondant à son type).

## II.3 Temps

Nous avons aussi créé une classe `Time` afin de représenter le temps et de pouvoir implémenter une horloge pour notre problème. Cette classe comporte trois attributs byte `hours`, `minutes` et `seconds`, ainsi que leurs getters. Elle comporte aussi les méthodes suivantes :

- `incMinute(byte m)` : incrémente l'heure de `m` minutes ;
- `incSeconds(byte s)` : incrémente l'heure de `s` secondes ;
- `equals(Time toCompare)` : retourne un booléen vrai si `this` et `toCompare` sont égaux ;
- `randomBegin()` : retourne une heure aléatoire de départ du point d'entrée, distribuée selon une gaussienne autour de 10h du matin, entre 2h et 20h, et les minutes sont déterminées aléatoirement ;
- `randomEnd(Time begin)` : retourne une heure aléatoire de départ du travail, distribuée selon une gaussienne autour de 18h, comprise entre l'heure d'arrivée plus deux heures et minuit, et les minutes sont déterminées aléatoirement.

## II.4 Environnement

Afin de représenter l'environnement global de notre simulation, nous avons créé une classe `Environment` qui sera instanciée une unique fois et contiendra les différents éléments du problème.

### II.4.1 Attributs

Cette classe contient les différents éléments de la simulation :

- `points` : une `ArrayList` d'objets `Point` (classe mère et sous-classes) comprenant l'ensemble des points considérés ;
- `carPaths` : une `ArrayList` d'objets `Path` n'autorisant que les usagers en voiture (`RoadPath` et `HighwayPath`) ;

- `publicTransportPaths` : une `ArrayList` d'objets `Path` n'autorisant que les usagers non motorisés (`FootPath`, `BikePath`, `BusPath` et `RerPath`).

### II.4.2 Méthodes

En plus des getters et setters usuels, cette classe contient également les méthodes permettant d'implémenter un algorithme de Djikstra afin de retourner le plus court chemin d'un point A à un point B sur le graphe des points de l'environnement.

- `initializeWeights` : à partir d'une `ArrayList<Point>` et d'une `ArrayList<Path>`, retourne la matrice des poids minimal entre les points, c'est à dire la matrice de taille `nxn` avec `n` le nombre de points, contenant l'éventuel poids minimal (s'il existe un chemin) entre deux points ;
- `initializePaths` : à partir d'une `ArrayList<Point>`, d'une `ArrayList<Path>` et d'une matrice de poids (`double[] []`, retourne la matrice des éventuels chemins de poids minimal entre les points ;
- `shortestPathPointList` : implémente, à partir d'une `ArrayList<Point>`, d'une `ArrayList<Path>`, d'un point `source` et d'un point `target`, l'algorithme de Djikstra et retourne une `ArrayList<Point>` contenant les points formant le plus court chemin entre deux points ;
- `shortestPath` : récupère la liste des chemins formant le plus court chemin, à partir des méthodes `findShortestPath` et `initializePaths` ;
- `shortestCarPath` : retourne les chemins formant le plus court chemin pour un usager automobiliste ;
- `shortestPublicTransportPath` : retourne les chemins formant le plus court chemin pour un usager non motorisé.

## II.5 Import

La classe `Import` du package `Util` permet d'importer les points et chemins depuis des fichiers `csv`, afin de faciliter l'ajout de nouveaux éléments et la création des différents objets dans le programme.

Cette classe constitue un objet qui sera instancié en début de simulation, et qui contient en attributs les différentes listes d'objets (points et chemins) qui constitueront l'environnement. Cette classe contient une méthode pour chaque type d'objet, qui parcourt et parse chaque fichier `csv` (un par type d'objet), appelle le constructeur associé sur chaque instance, et l'ajoute à l'`ArrayList` attribut correspondant. Les fichiers `csv` sont stockés dans le dossier `objects`.

Au début de la simulation, il suffit d'instancier un objet de la classe `Import` à l'aide de son constructeur, qui appelle dans l'ordre défini les différentes méthodes d'importation des objets, et ensuite de récupérer les différentes listes dont on aura l'utilité à l'aide des getters associés.

# III Agents et comportements

## III.1 Horloge

Dans le package `Environment` on retrouve deux classes : `Clock` et `ClockTick`. Notre premier agent, le plus important, est en effet l'horloge. Un seul agent d'horloge est créé pour la journée, et sont rôle sera de mesurer l'heure et d'en informer les autres agents.

`Clock` possède un attribut `currentTime` de type `Time` décrit plus haut. Cet attribut représente l'heure de notre système. Elle est régulièrement incrémentée à l'aide du comportement `ClockTick` héritant de `TickerBehaviour` de Jade. Ce type de comportement est appelé à intervalles réguliers, l'intervalle étant défini ici à l'aide de la durée réelle `simulationTime` d'exécution souhaitée pour une journée simulée ; et de l'intervalle de temps minimal simulé `stepLength`, typiquement 15 secondes (un intervalle plus grand, de une minute par exemple, a l'inconvénient d'arrondir à une minute toutes les durées inférieures, ce qui fausse vite la simulation sur les segments courts).

À chaque tick d'horloge, `ClockTick` envoie aussi un message contenant l'heure du système à tous les autres agents. Le comportement s'arrête quand minuit est atteint. Un message est alors envoyé à l'agent `Starter` pour prévenir de la fin de journée.

## III.2 Individus

### III.2.1 Principaux attributs

Chaque personne est représentée par un agent `Person`. Ils possèdent chacun des attributs comme :

- Leur point de pré-entrée, déterminé à la construction
- Leur emploi du temps `schedule`, déterminé à la construction et qui sera "consommé" au fur et à mesure de la journée
- Leur localisation (point et éventuellement chemin)
- Leur choix de mode de transport, déterminé à la construction
- L'ID de la ligne de transport en commun qu'ils empruntent actuellement (0 sinon). Cela servira pour détecter les changements de ligne.

L'emploi du temps `schedule` consiste en une liste de rendez-vous `Appointement`, qui sont eux-mêmes de simples couples (`Point`, `Time`) correspondants à la destination et à l'heure de départ (et non d'arrivée).

Ces agents fonctionnent comme des machines à état, et un type énuméré `PersonState` est donc créé pour décrire ces états. Trois états sont possibles :

- `in_place` : La personne est là où elle doit se trouver, mais il reste des déplacements prévus dans son emploi du temps.
- `moving` : La personne est en déplacement.

- `gone` : La personne est rentrée chez elle, son emploi du temps a été entièrement exécuté, elle n'interviendra plus de la journée.

Cet état est stocké dans l'attribut `personState`. Les comportements `InPlace` et `Moving` associés sont décrits dans les parties suivantes (L'état `gone` n'a pas de comportement associé : cet état correspond à l'arrêt des différents comportements).

#### III.2.2 Comportement InPlace

Quand il est `in_place`, l'agent attend un signal de départ à l'aide de son comportement `InPlace`. C'est un comportement déclenché à chaque réception de tick d'horloge. L'heure est comparée avec l'heure du départ du prochain `Appointement`. Si il est l'heure de partir, on enclenche le trajet en créant un nouveau comportement `Moving` associé au point de destination. Le rendez-vous est supprimé du `schedule`, `PersonState` est modifié, et le comportement `InPlace` est ensuite arrêté.

#### III.2.3 Comportement Moving

Quand il est `moving`, l'agent se déplace vers son point de destination à l'aide de son comportement `Moving`. Le comportement est créé avec un point de destination, conservé en attribut. Lors de son déplacement, l'agent peut se trouver dans deux états différents correspondant à des phases différentes du mouvement. On créera pour celà un type énuméré `MovingState` :

- `none` : La personne n'est pas dans un état de mouvement `moving`
- `point` : La personne est à un `Point`, en attente du choix du prochain `Path`
- `path` : La personne est en train de parcourir un `Path`

Quand il est sur un point, l'agent récupère le prochain segment à emprunter à l'aide de Djikstra. On récupère le temps nécessaire pour parcourir le segment (prenant en compte comme expliqué la côte à pied, les embouteillages en voiture, et le temps d'arrêt en transports en commun). On y ajoute l'éventuel temps d'attente moyen lors d'un changement de ligne. On stocke ce temps dans un attribut `timeNeeded` et on initialise un compteur de temps passé sur le segment `currentPathProgress` à 0. On incrémente les compteurs globaux de temps passé dans les transports. Le moving state est finalement passé à `path`.

Quand il est sur un segment, l'agent écoute les tick d'horloge et incrémente son compteur de temps passé sur le segment `currentPathProgress`. Quand `timeNeeded` est atteint, on change finalement l'attribut `localisation` de l'agent pour le placer sur le prochain point. Si le point atteint est la destination, on arrête le mouvement et on passe en `in_place` si il reste des rendez-vous et en `gone` sinon. Si le point atteint n'est pas la destination, le mouvement continue et on recommence ce cycle en passant le moving state à `point`.

### III.2.4 Construction

Les personnes présentent un constructeur qui permet de les construire à la main, mais nous utiliserons plutôt un constructeur `rand_AllerRetour` qui construit une personne avec un emploi du temps simple : un aller retour depuis un point de pré-entrée vers un point d'intérêt. Les heures des trajets sont définies à l'aide de `randomBegin()` et `randomEnd()` décrite plus haut dans la section Time.

On se rendra vite compte que la création de trop nombreux agents peut créer des problèmes de mémoire et de CPU suivant la machine et les paramètres. Pour pouvoir constater des embouteillages plus facilement, on introduira dans le starter un paramètre `realUsersPerPerson` qui représente le nombre d'usagers réels représentés par un agent `Person`. Cette valeur interviendra dans les calculs de densité d'utilisateurs des routes.

# IV Interface graphique

Afin d'avoir une représentation graphique claire des différents points, chemins, et des évènements de la simulation, nous avons choisi de définir une interface graphique à l'aide des outils `Swing`. Cela consiste en deux classes `Window` et `Panel`. `Window` hérite de la classe `JFrame` et définit les caractéristiques de la fenêtre qui contiendra le `Panel`. `Panel` hérite de la classe `JPanel` et contient les méthodes associées au dessin des représentations. La méthode `paintComponent` dessine les différents éléments dans un `bufferGraphics`, attribut d'une image. L'horloge est également affichée.

Afin d'avoir une simulation plus réaliste et de pouvoir mieux se représenter le problème, nous avons choisi de nous baser sur une carte réelle (Google Maps) pour définir nos points et chemins.

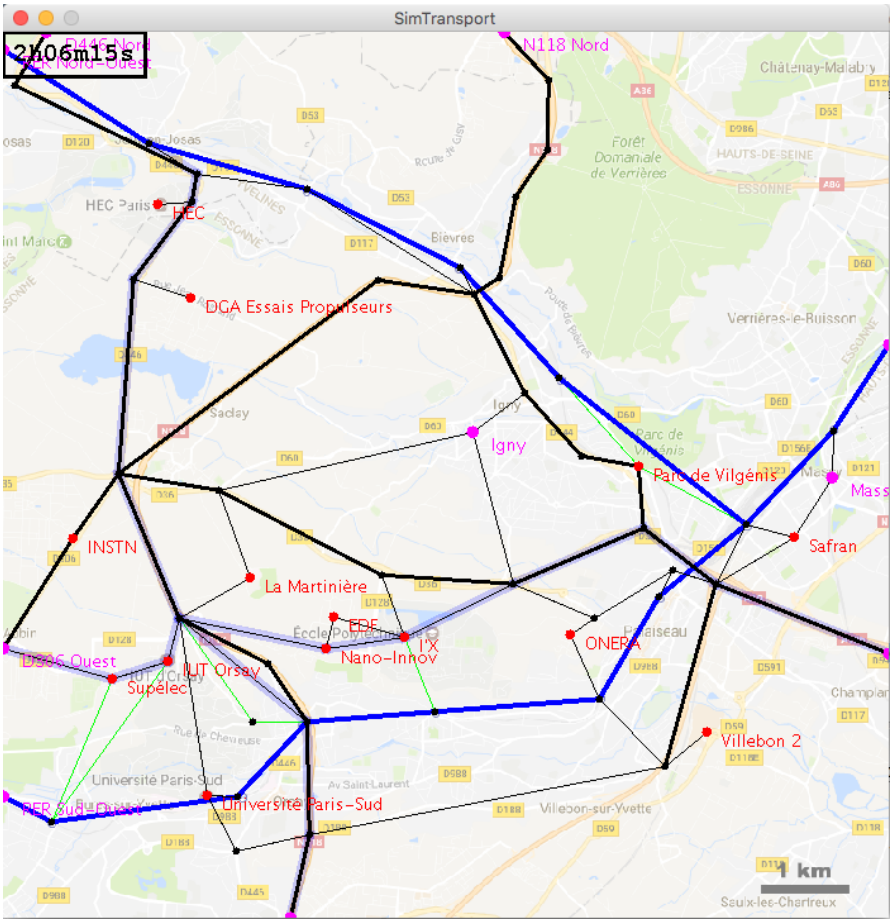


Figure 1: Représentation graphique de la simulation

Les différents éléments sont ajoutés au `bufferGraphics` dans la méthode `paintComponent` via les méthodes de dessin classique de `Swing`. Les points d'entrée sont représentés par des disques (`fillCirce`) roses, et les points d'intérêt sont représentés par des disques rouges et leur nom est affiché (`drawString`).

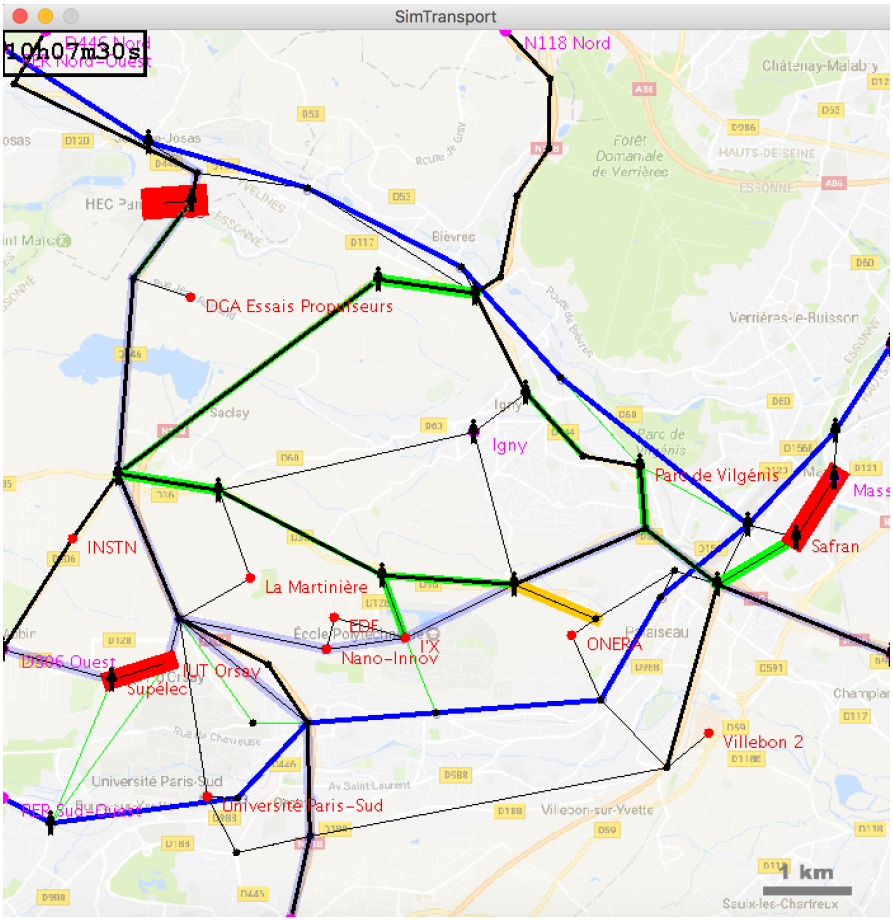


Figure 2: Simulation provoquant la saturation sur certaines routes

Les chemins de type route sont représentés en noir, et avec un trait plus épais pour les chemins de type nationale ou autoroute. Les chemins de type RER sont représentés en bleu légèrement plus épais. Les chemins piétons et cyclistes sont représentés en vert. Les chemins d'autobus sont représentés en bleu transparent légèrement plus large, afin d'apparaître sur les chemins routier. Enfin, pour les chemin de type `roadPath` et `highwayPath`, qui peuvent être saturés, nous avons fait le choix de représenter la saturation par une couleur (vert, orange ou rouge selon le niveau), ainsi qu'avec une épaisseur variable (en fonction du rapport entre le nombre d'agents et la capacité effective).

De plus, chaque agent (pouvant représenter plusieurs personnes), est représenté par un pictogramme de la forme d'un bonhomme. Ces pictogrammes se déplacent sur la carte (aux différents points de l'environnement) en suivant le déplacement de l'agent associé. Pour cela, la liste des agents `Persons` est ajoutée au constructeur du `Panel`, afin de pouvoir accéder aux informations des objets.

Un exemple de simulation avec des chemins saturés est présenté figure 2.

# V Utilisation et inflluence des paramètres

## V.1 Lancement d'une simulation

Le lancement d'une simulation se fait via le lancement d'un agent `Starter`. Pour cela le plus simple est de créer une nouvelle Run Configuration avec comme Main class `jade.Boot` et comme Program arguments :

```
-local-host 127.0.0.1 -agents "starter:mainPackage.Starter"
```

Les différents paramètres disponibles sont décrits dans la partie suivante.

## V.2 Paramètres d'affichage

### V.2.1 Console

Un paramètre `verbose` est présent dans le starter. Il décrit la quantité d'information affichée en console lors de la simulation. Différents niveaux de précision sont disponibles :

- `verbose <= -1` : Rien n'est affiché
- `verbose = 0` : Seulement le bilan des différentes journées est affiché
- `verbose = 1` : Sont affichés le bilan des journées, les lancements des personnes, les départs et arrivées des différentes personnes, le témoin des imports
- `verbose >= 2` : Sont en plus affichés le détail des trajets des personnes

### V.2.2 Interface graphique

Un paramètre booléen `showSimulation` est disponible pour l'affichage de la simulation. Un paramètre boolean `showTraffic` est aussi disponible pour camoufler les bordures affichées représentant la densité d'utilisateurs. La taille de la fenêtre est aussi modifiable.

## V.3 Paramètres généraux de simulation

Des paramètres généraux de simulation dont certains ont déjà été évoqués plus haut sont modifiables dans `Starter` :

- `simulationTime` : durée réelle de la simulation d'une journée de 24 heures
- `stepLength` : intervalle minimal de temps simulé (15 secondes recommandé)
- `startHour` : heure de début de la simulation
- `nbPersons` : nombre de personnes simulées
- `realUsersPerPerson` : nombre d'utilisateurs réels représentés par un agent (pour l'encombrement des routes)

Les paramètres des lois gaussiennes des heures générées aléatoirement lors de la création des emplois du temps : `centerBeginTime, sigmaBeginTime, centerEndTime, sigmaEndTime`.

### V.4 Paramètres des Path

Les paramètres des classes `Path` comme la vitesse moyenne, le seuil critique de densité pour les routes automobiles, ou le temps perdu à chaque arrêt pour les transports en commun, sont accessibles directement dans les classes concernées. Ce sont des paramètres qui concernent le système réel lui même, nous n'avons donc pas désiré les faire apparaître directement dans le starter avec les autres paramètres.

On peut aussi imaginer des `Path` qui auraient une certaine densité d'utilisateurs "de base", indépendante de notre système, comme sur la N118 par exemple (qui peut être embouteillée indépendamment des comportements des usagers du plateau de Saclay).

On peut ainsi vérifier le comportement du prototype et de ces paramètres en effectuant des simulations. Par exemple, on effectuera deux simulations, en changeant le temps d'arrêt et le temps moyen d'attente des transports communs. On vérifie en console les durées annoncées sur chaque trajet, et on trouve bien des différences de performances, reportées en figure 3.

| **RER** stopTime | **RER** waitingTime | **Bus** stopTime | **Bus** waitingTime | **Aller-Retour moyen en transports en commun** |
|---|---|---|---|---|
| 15 | 3 | 20 | 5 | 3199 secondes |
| 30 | 6 | 40 | 10 | 3711 secondes |

Figure 3 : Différences de performances selon les paramètres des transports

### V.5 Paramètres du modèle de décision

Comme précisé en première partie, au delà de vouloir mesurer les performances du système selon les conditions, les routes éventuellement fermées, les lignes de bus créées etc, il est aussi possible de modéliser le comportement des utilisateurs et leur choix de transport sur plusieurs jours.

Les paramètres concernant cet aspect du prototype sont disponibles dans `Starter`, juste en dessous des paramètres généraux. On trouve :

- `totalDays` : le nombre de jours
- `carFactorInit` : la probabilité initiale qu'une personne choisisse la voiture face aux transports en commun
- `carFactor()` : dans cette fonction réside le modèle de décision des agents. Elle peut se servir des scores de la journée précédente (nombre d'utilisateurs pour chaque mode de transport et performance de chaque mode) et doit renvoyer une valeur de `carFactor` qui correspondra à la nouvelle probabilité pour une personne de choisir la voiture.

Le choix de `carFactor()` est délicat et représente donc la façon de penser des utilisateurs. Cette fonction devra être perfectionnée par l'utilisateur du prototype selon ses métadonnées et sa connaissance du comportement des usagers du plateau de Saclay.

Nous avons nous même implémenté un simple ratio (le moyen de transport le plus rapide est le plus choisi). On pourrait aussi imaginer une notion de coût du trajet ou de praticité qui dissuade les utilisateurs dans un sens ou dans un autre.

On effectue par exemple une simulation avec seulement 100 personnes par journée, sur 8 jours, sans traffic supplémentaire que ces 100 personnes. Selon notre fonction de décision, on intuite que l'absence d'embouteillages devrait rendre la voiture plus efficace et donc pousser les utilisateurs à l'adopter. On récupère les résultats en console et on vérifie ainsi bien le comportement en figure 4.

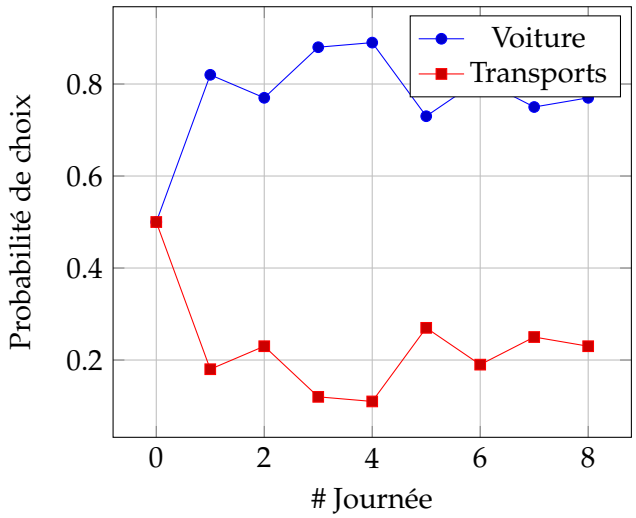


Figure 4 : Evolution des comportements sur 8 jours sans embouteillages

On remarque d'ailleurs qu'il y a un certain bruit dans le comportement, mais qui pourrait être diminué avec un choix de modèle plus peaufiné, pondéré sur plusieurs jours par exemple.

## Conclusion

On aura donc développé un prototype d'application permettant l'étude des déplacements des usagers du Plateau de Saclay. Les simulations sont très modulables ce qui permet donc à l'utilisateur de personnaliser ses modélisation selon ses besoins et ses métadonnées.

Les simulations sont observables sous différents angles et à l'aide de différentes métriques : interface graphique, log des déplacements, compte rendu des performances sur une journée, fonction de décision des utilisateurs... Qui nous on permis de vérifier le fonctionnement du prototype et qui permettront aux utilisateurs de tirer des conclusions de leurs simulations.