Documentation complète
du
SimTaxi

Simulation pour l'optimisation des politiques d'utilisation de l'information à l'intérieur d'une compagnie de taxis.

Julien BURDY
Grégory BURRI
Lucien CHABOUDEZ
Alexandre D'AMICO
Vincent DECORGES
Patrice FERROT
Lionel GUÉLAT
Joël JAQUEMET

15 juillet 2003

Résumé

SimTaxi est un logiciel de simulation visant à optimiser des politiques d'utilisation de l'information à l'intérieur d'une compagnie de taxis. Le but de la compagnie de taxis est de minimiser le nombre total de kilomètres parcourus. Pour réaliser ce projet un réseau modélisant Lausanne (en ordre de grandeur) a été réalisé.

Ce projet est réalisé dans le cadre d'un laboratoire de "Projet de groupe en informatique"[1] compris dans la formation d'ingénieur en logiciel de l'école d'ingénieurs du canton de Vaud.

[1] Laboratoire n'ayant pas de cours associé.

Table des matières

Chapitre 1

Introduction

1.1 Organisation

Ce document est fractionné de la façon suivante :

Introduction présentant le problème lui-même (l'énoncé), l'équipe que nous sommes, le choix des outils utilisés ainsi que le langage de programmation choisi.

Documentation de développement présentant la conception et le raffinement général du problème en modules, puis décrit le rôle de chaque module ainsi que les méthodes mises en oeuvre pour y arriver. Cette partie de documentation fait un maximum abstraction de l'implémentation. L'implémentation précise est décrite par la documentation technique au format html (cf. chapitre 4).

Politiques présente les politiques implémentées et comment on peut les comparer.

Annexes Dans les annexes nous trouverons nos documents internes, la documentation utilisateur, le journal de travail, etc...

De façon à pouvoir uniformiser le travail, nous avons créé nos propres documents internes. Soit :

Documentation qualité pour uniformiser l'écriture du code source (cf. annexe 6)

Documentation de rédaction pour uniformiser l'écriture de la documentation (cf. annexe 7)

1.2 Énoncé

Il s'agit de développer un programme de simulation visant à optimiser des politiques d'utilisation de l'information à l'intérieur d'une compagnie de taxis.

1.2.1 Description du système

- Un certain nombre de taxis circulent dans la ville, leur activité étant partiellement coordonnée à partir d'un central. Un certain nombre de stations, dans lesquelles les taxis attendent leur prochaine course, sont réparties dans la ville.

- Lorsqu'un taxi a fini sa course, il se dirige vers une station (choisie en fonction d'une certaine politique). Il transmet cette information au central.

- Au moment où le central reçoit un appel d'un client demandant un taxi, il transmet l'ordre de course à un taxi (choisi selon les critères de la politique en cours). Il peut s'agir soit d'un taxi en station ou d'un taxi qui est en train de retourner à une station.

- Le but de la compagnie de taxis est de minimiser le nombre de kilomètres parcourus par les taxis à vide (sans client).

- Les carrefours sont modélisés par des sommets et les morceaux de rues (entre 2 croisements) sont modélisés par les arêtes d'un graphe orienté et connexe. On supposera que les stations de taxis et les points de départ/fin d'une course se trouvent au milieu des arêtes du graphe. Pour tous les trajets effectués, les taxis emprunteront les chemins les plus courts.

1.2.2 Modélisation

La modélisation réaliste de ce problème est une tâche dépassant largement le cadre du projet. C'est pourquoi nous n'avons pas simulé les fluctuations de la demande et l'encombrement du trafic en fonction de l'heure (heures de pointe, heures creuses). Pour simplifier, nous avons donc supposé que nous nous trouvions dans une situation de charge moyenne. Nous avons néanmoins veillé à respecter certains ordres de grandeur au niveau :

- du réseau : le graphe doit comporter 800 sommets (carrefours) et 1400 arêtes (morceaux de rues). Le diamètre du graphe doit être de 12 km et le temps maximum pour aller de 2 points se situant à chaque extrémité doit être d'environ 20 min.

- de la demande : globalement, on compte environ 2700 courses à effectuer chaque jour. En moyenne, une course représente environ 8 km (distance de transport d'un client).

- des stations : le graphe comprend 30 stations, une de 20 places, deux de 10 places et le reste avec 5 places.

- des taxis : 180 taxis circulent en permanence et effectuent en moyenne 160 km par jour.

1.3 Équipe

L'équipe de SimTaxi est constituée de huit étudiants de la classe EIA-6 de l'eivd[1]. La répartition du travail à été faite de la manière suivante :

Julien Burdy	Chef de groupe
Grégory Burri	Interface utilisateur
Lucien Chaboudez	Gestionnaires taxis et stations
Alexandre D'Amico	Échéancier, central et statistiques
Vincent Decorges	Central, politiques et graphiques
Patrice Ferrot	Initialisations et génération des courses clients
Lionel Guélat	Algorithmique dans le réseau et optimisations
Joël Jaquemet	Génération et structure du réseau (graphe) et comparaison des politiques.

Nous avons eu environ 60h/personne à disposition.

1.4 Choix du langage

Pour réaliser ce programme, nous nous sommes tournés vers le langage Python (interprété, interactif, orienté-objet, portable), permettant une rapidité de développement imbattable à notre connaissance mais au profit d'une perte de performance de la simulation même.

[1] http ://ina.eivd.ch

1.5 Outils

Toute la documentation a été créée avec LaTeX[2]. La gestion des différentes versions de chacun des fichiers du programme a été réalisée grâce au CVS[3] de Sourceforge[4]. Pour les schémas UML, nous avons eu recours à Umbrello[5]. L'extraction de documentation technique (depuis les sources) à été faite avec HappyDoc[6].

1.6 Environnement et librairies

L'interface utilisateur est implémentée à l'aide de wxPython[7], l'affichage du réseau avec PyOpenGL[8] et le tout est sur un environnement Python[9] récent.

[2] http ://www.latex-project.org
[3] http ://www.cvshome.org
[4] http ://www.sf.net
[5] http ://uml.sf.net
[6] http ://happydoc.sf.net/
[7] http ://www.wxpython.org
[8] http ://pyopengl.sf.net
[9] http ://www.python.org

Chapitre 2

Documentation de développement

2.1 Raffinement général

La décomposition du problème s'est faite de façon assez rapide. Nous avons fractionné au maximum pour faciliter le travail en groupe. La figure 5.2 montre en un coup d'oeil cette décomposition.

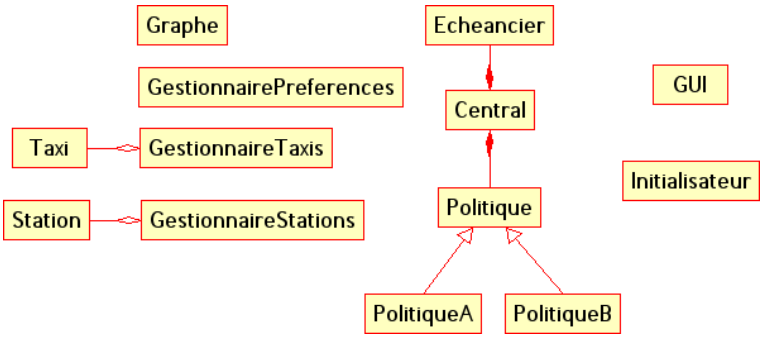


FIG. 2.1 – *Raffinement général des modules*

Nous avons rapidement identifié un graphe, contenant les coordonnées cartésiennes des différents carrefours, facilitant ainsi l'affichage et la génération du réseau. C'est aussi à lui qu'on demande les chemins les plus courts d'un point à un autre. Il est unique et accessible depuis n'importe quel autre module de la simulation.

Différents gestionnaires, qui permettent de regrouper et de gérer différents éléments, comme les taxis, les stations, les paramètres, les statistiques, etc... Ils évitent (dans les cas des gestionnaires de taxis et de stations) aux autres modules de devoir itérer toutes les instances pour faire leur sélection («quel est le taxi le plus proche de ?»).

Ensuite, comme toute simulation classique, nous avons un échéancier contenant les événements. Événements qui sont traités depuis le module central (ce qui est assez proche de la vie réelle).

Les modules politiques définissent le comportement des taxis. Avant chaque décision le central consulte la politique courante pour savoir quel taxi choisir ou où va se diriger un taxi. La figure 2.2 image un peu la sélection d'un taxi.

Il est assez facile d'écrire une nouvelle politique, en choisissant parmi les méthodes fournies par les gestionnaires pour sélectionner un taxi ou un lieu (station).

Une interface utilisateur permet de visionner plusieurs états :

- L'état du graphe, l'emplacement des stations, l'apparition et la disparition de clients ainsi que les mouvements des taxis.
- Certaines valeurs numériques de la simulation (nombre d'événements déjà traités, évolution de la simulation, etc...).
- Des courbes indiquant certaines charges (nombre de taxis disponibles, etc...).

Il est indépendant de la simulation, il ne fait que consulter l'état et l'emplacement des éléments pour ensuite les afficher.

C'est aussi l'outil de travail permettant d'observer, de comparer les politiques pour pouvoir en créer de nouvelles ou les modifier. Une fois les politiques matures, il n'est pas conseillé d'exécuter les simulations avec l'interface utilisateur car ce dernier ralentit le temps de traitement.

Puis un outils permettant de comparer les politiques entre elles après avoir recueilli les statistiques de chacune (cf. section 3.2).

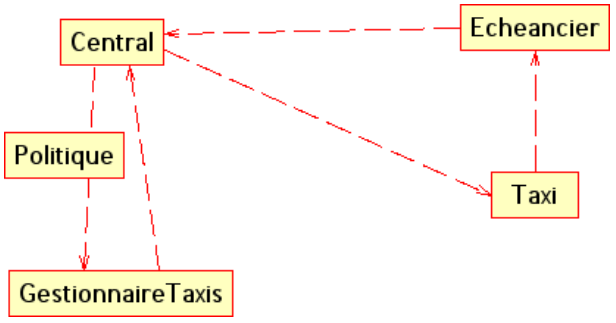


FIG. 2.2 – *Communication au coeur de la simulation*

2.2 Le central

Le central est le coordinateur global, c'est lui qui gère tout. Il comprend :

- **Un échéancier** : contient les événements(Client, Charger un client, Poser un Client et Arriver en station).
- **Un gestionnaire de taxis** : gère les taxis, fournit un taxi libre suivant la politique utilisée.
- **Un gestionnaire de stations** : gère les stations, indique une station suivant la politique utilisée.
- **Une politique** : détermine le choix d'un taxi libre plutôt qu'un autre pour un client donné. C'est elle aussi qui détermine le choix de la station à laquelle le taxi se rendra après sa course.

2.2.1 Les événements

Les événements sont au coeur de SimTaxi, ce sont eux qui déclenchent les actions à effectuer lors de la simulation.

Ils sont de 4 types :

- **Client** : correspond à une demande d'un client.
- **Charger un client** : un taxi va charger un client.
- **Poser un client** : un taxi dépose le client à sa destination.
- **Arriver station** : un taxi arrive à une station.

Chaque événement dispose d'un champ *temps* qui indique à quel moment ils doivent survenir. Tous les événements sont stockés dans un échéancier dans leur ordre chronologique.

Conception

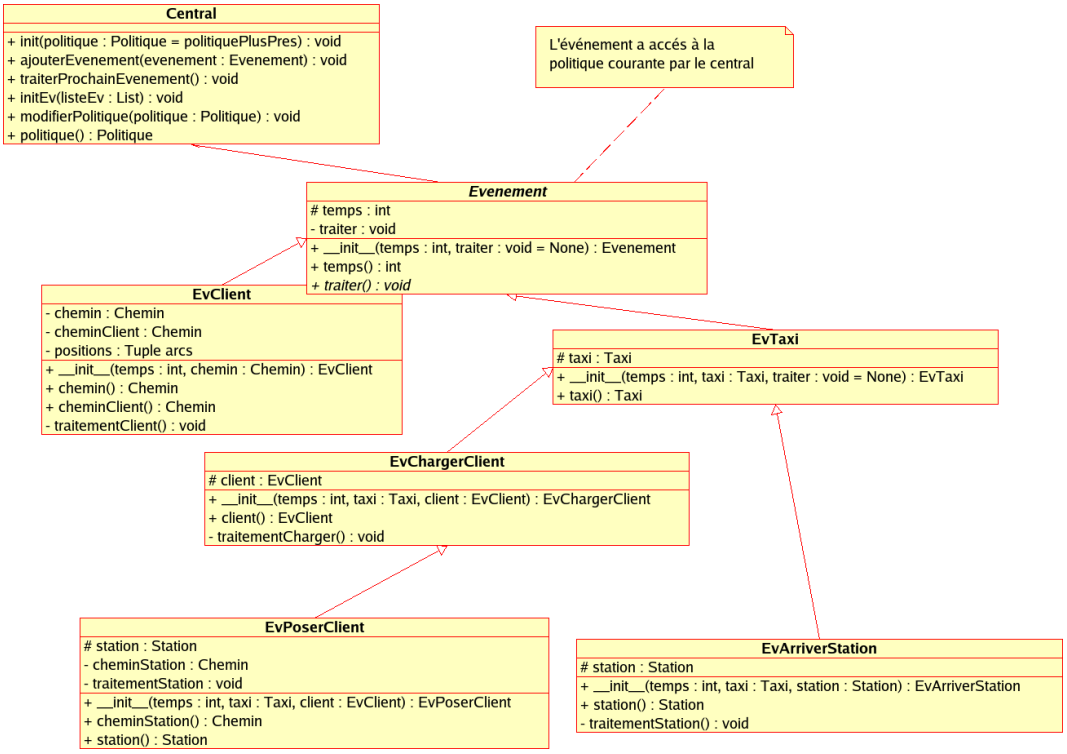


FIG. 2.3 – *Diagramme de classe des événements*

Le diagramme de classe (figure 2.3) montre l'implémentation des différents événements. Tous les événements dérivent de la classe abstraite *Evenement* qui définit leurs structures de base. Le champ *temps* permet de fixer à quel moment l'événement va avoir lieu et le champ *traiter* permet d'associer une action à un événement.

Les actions peuvent être définies à l'extérieur de l'événement (sous-programme externe) ou à l'intérieur de celui-ci. Pour SimTaxi tous les événements définissent eux-mêmes leurs actions.

Voici la liste des actions pour chaque événement.

EvClient :

1. Calcule le chemin de la position du client à la destination.
2. Demande à la politique courante (voir 2.10) un taxi.
3. Si aucun taxi n'est disponible, met le client en attente dans le central.
4. Autrement s'envoie au taxi.

EvChargerClient :

1. S'envoie au taxi.

EvPoserClient :

1. Demande à la politique courante une station.
2. S'envoie au taxi.

EvArriverStation :

1. S'envoie au taxi.

Séquence des événements

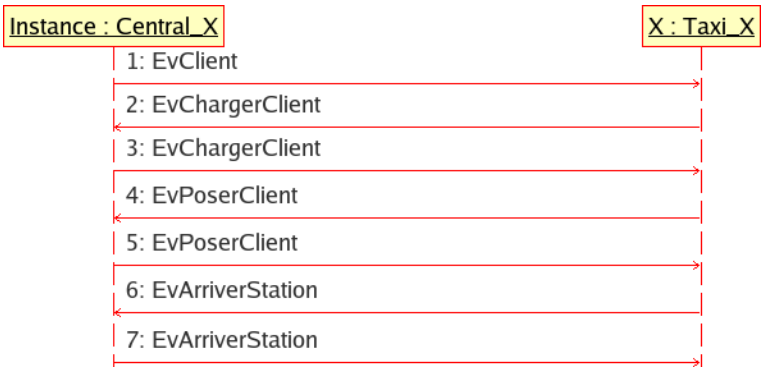


FIG. 2.4 – *Séquences des événements*

La figure 2.4 montre l'enchaînement des événements pour une course.

On peut la résumer ainsi :

1. Un événement client (*EvClient*) est sorti de l'échéancier du central. Il est envoyé à un taxi par le biais de sa méthode *traiter*.
2. Le taxi reçoit l'*EvClient*. Il calcule à quelle date il va charger le client et peut ainsi créer un événement charger client (*EvChargerClient*) qu'il transmet au central.
3. Le central transmet l'*EvChargerClient* (méthode *traiter*) au taxi lorsque la date est atteinte pour lui dire qu'il a chargé un client.
4. Le taxi calcule la date à laquelle il va déposer le client et envoie l'événement *EvPoserClient* au central.
5. Le central envoie *EvPoserClient* au taxi lorsque la date est atteinte.
6. Le taxi calcule la date d'arrivée à la station est transmet *EvArriverStation* au central.
7. Le central transmet *EvArriverStation* au taxi pour lui dire qu'il est arrivé à la station.

2.2.2 L'échéancier

Description

L'échéancier a pour but le stockage des événements à venir. Ils sont ordonnés dans un ordre croissant, en fonction de la date à laquelle ils auront lieu.

Conception

L'échéancier n'est autre qu'une queue de priorité, avec une insertion dichotomique.

2.3 Statistiques

Le gestionnaire de statistiques stocke toutes les informations liées à chaque événement. Ce qui nous permet de créer toutes sortes de statistiques concernant les données.

Chaque événement du central est transmis au gestionnaire de statistique, le gestionnaire les traite différemment suivant leur type. Car chaque type d'événement contient des informations spécifiques.

2.3.1 Stockage des informations

Un dictionnaire a été utilisé pour stocker les informations relatives à chaque événement. Les clés du dictionnaire sont les heures auxquelles les événements ont eu lieu.

La structure pour les informations est la suivante, [nb de courses effectuées, infos taxis, infos clients, état stations].

La structure pour les informations taxis, [noTaxi, ta, [tsChClient, tsAvClient, tsReStation], na, [nsChClient, nsAvClient, nsReStation], [lChClient, lAvClient, lReStation]].

La structure pour les informations clients, [taClient, tsClient, naClient, nsClient].

La structure pour l'état des stations, no station :[occupation, demande, capacité], ...

La demande représente le nombre de clients qui ont fait des demandes à proximité de la station mais qui n'ont pas forcément été desservis par un taxi de cette station. ce qui donne [nbCourses, [noTaxi, ta, [tsChClient, tsAvClient, tsReStation], na, [nsChClient, nsAvClient, nsReStation], [lChClient, lAvClient, lReStation]], [taClient, tsClient, naClient, nsClient], no station : [nb taxis, nb demandes, capacité], ...].

nbCourses	[0]
noTaxi	[1][0]
ta (taxi)	[1][1]
tsChClient	[1][2][0]
tsAvClient	[1][2][1]
tsReStation	[1][2][2]
na (taxi)	[1][3]
nsChClient	[1][4][0]
nsAvClient	[1][4][1]
nsReStation	[1][4][2]
lChClient	[1][5][0]
lAvClient	[1][5][1]
lReStation	[1][5][2]
taClient	[2][0]
tsClient	[2][1]
naClient	[2][2]
nsClient	[2][3]
l'état des stations	[3]

2.3.2 Traitement des événements

Pour tous les événements, lors du traitement les données suivantes sont stockées :

- l'heure de l'événement.
- le nombre de courses.

- le numéro du taxi affecté au client.
- le nombre de taxis en attente.
- le nombre de taxis en service allant chercher un client.
- le nombre de taxis en service avec un client.
- le nombre de taxis en service retournant en station.
- le nombre de clients en attente.
- le nombre de clients en service.
- l'occupation et la charge des stations.

Événement client

Un événement client est traité seulement s'il contient un taxi, s'il n'en contient pas cela veut dire qu'il n'y a plus de taxi disponible et l'événement client est mis en attente dans le central(un autre événement client sera généré lorsqu'il y aura un taxi de libre).

Lorsqu'un événement client est traité, les valeurs suivantes sont stockées, modifiées :

- le nombre de courses, est incrémenté de un.
- le nombre de taxis en service allant chercher un client, un taxi supplémentaire. Étant donné qu'il y a un taxi en plus pour aller chercher un client, il y a donc un taxi de moins dans un autre état. Un taxi recevant un événement client peut être dans plusieurs états, en attente ou retournant en station. On doit donc mettre à jour la valeur correspondante à son état. Il faut donc décrémenter respectivement le nombre de taxis en attente et le nombre de taxis retournant en stations.
- le temps d'attente ou le temps de service retour station suivant l'état du taxi. La valeur est le temps inter événement.
- la distance retour station dans le cas ou le taxi a été intercepté lors ce qu'il retournait en station.
- le nombre de clients en attente, un client supplémentaire en attente d'un taxi.

Il reste un problème avec les événements client mis en attente dans le central. Comme on ne les traite pas directement mais seulement lors du nouvel événement client qui sera généré lorsqu'il y aura à nouveau un taxi de libre, ils ne sont pas comptabilisés dans le nombre de clients en attente. Le nombre de clients en attente n'est donc pas exact à tout moment.

Événement charger client

Lors du traitement d'un événement charger client, les valeurs suivantes sont stockées, modifiées :

- le nombre de taxis en service allant chercher un client, est décrémenté de un.
- le nombre de taxis en service avec un client est incrémenté de un.
- le nombre de clients en attente est décrémenté de un.
- le nombre de clients en service est incrémenté de un.
- le temps de service du taxi pour aller chercher le client.
- la distance parcourue par le taxi pour aller chercher le client.
- le temps d'attente du client.

Événement poser client

Lors du traitement d'un événement poser client, les valeurs suivantes sont stockées, modifiées :

- le nombre de taxis en service avec un client est décrémenté de un.
- le nombre de taxis en service retournant en station est incrémenté de un.
- le nombre de clients en service est décrémenté de un.
- le temps de service du taxi avec le client.
- la distance parcourue par le taxi avec le client.

- le temps de service du client.

Événement arriver station

Un événement arriver station peut être traité de différentes façons suivant l'état de la station concernée. Lors de son arrivée en station : si elle n'est pas pleine on peut mettre à jour le nombre de taxis en attente et celui des taxis en service retournant en station. Si elle est pleine, le taxi ne va pas pouvoir rester dans cette station, il va donc aller vers une autre. Il sera toujours dans l'état retour station. Un autre événement retour station sera donc généré. La distance parcourue est de toute façon stockée, car on la calcule grâce à l'événement précédent, idem pour le temps.

Lors du traitement d'un événement arriver station, les valeurs suivantes peuvent être stockées, modifiées suivant les cas :

- le nombre de taxis en service retournant en station est décrémenté de un.
- le nombre de taxis en attente est incrémenté de un.
- le temps de service du taxi retournant en station.
- la distance parcourue par le taxi pour rentrer en station.

2.3.3 Extraction des informations

Pour l'extraction des informations voulues il suffit de parcourir le dictionnaire et de récupérer les données souhaitées tout en faisant bien attention avec les valeurs stockées pour les événements client et arriver station qui peuvent varier suivant l'état des taxis respectivement des stations.

2.3.4 Stockage des statistiques

Le dictionnaire des statistiques ainsi que la liste des taxis peuvent être enregistrés dans un fichier ayant comme nom le nom de la classe de la politique utilisée avec l'extension *.sta dans le dossier de la ville. Ce qui permet l'importation des données dans n'importe quelle application. Il est donc possible d'utiliser des logiciels permettant de créer des graphiques indépendamment de l'application SimTaxi, en créant une application parcourant le dictionnaire et sauvegardant les données voulues dans un format spécifique au programme voulu.

2.4 Le réseau (graphe)

Le rôle du réseau est de modéliser la carte routière sur laquelle nous voulons faire une simulation.

Nous avons donc utilisé un graphe pour représenter le réseau. Ce graphe est orienté pour permettre de définir des sens uniques. Un dictionnaire est utilisé pour implémenter le graphe avec comme clés, les sommets. A chaque clé est associée une liste contenant en première position les attributs du sommet et ensuite les sommets qui sont reliés depuis le sommet décrit par la clé. On a donc pour chaque entrée du dictionnaire : un sommet, ses attributs et ses arcs sortant. Nous avons implémenté toutes les méthodes "usuelles" permettant de manipuler un graphe et deux méthodes permettant d'enregistrer et de récupérer un graphe dans, et respectivement à partir d'un fichier.

2.4.1 Génération d'un réseau

Il nous a fallu créer une méthode permettant de générer un réseau avec les contraintes suivantes : nombre de carrefours, nombre de portions de rues (orientées) et distance maximale entre 2 carrefours). Nous ne voulions pas avoir un réseau "américain" (rues perpendiculaires), nous avons donc imaginé un algorithme qui donne un résultat relativement proche des plans de villes réels.

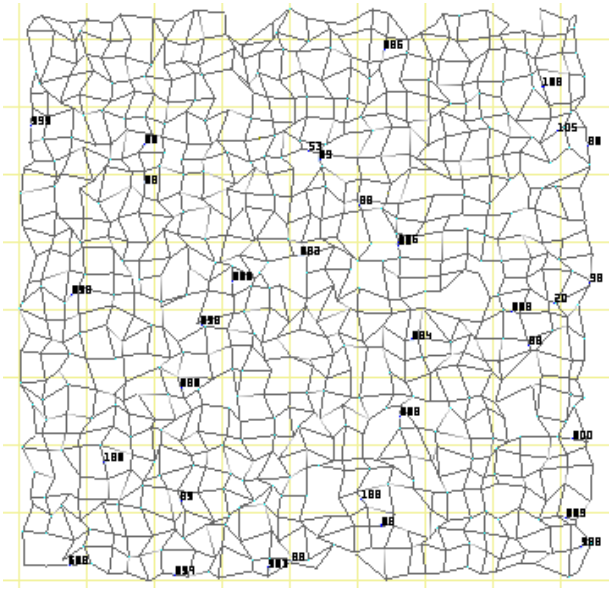


FIG. 2.5 – *Un graphe résultant de la génération*

Pour ce faire, nous formons dans un premier temps un réseau "américain" (n*n) contenant le nombre suivant de carrefours :

$$n = \lceil \sqrt{nbCarrefoursDemands} \rceil^2$$

Chaque carrefour est cependant placé aléatoirement dans un périmètre qui lui est réservé (les périmètres ne se chevauchant pas) pour supprimer la forme "américaine" du réseau ! Ensuite nous créons aléatoirement un chemin formé de rues bi-directionnelles qui passe par tous les carrefours pour assurer la connexité du réseau. Il ne nous reste plus qu'a supprimer aléatoirement des carrefours en garantissant la connexité (on fusionne les rues qui passent par les carrefours supprimés (pour chaque carrefour indépendemment) jusqu'à en avoir le nombre demandé et qu'a ajouter (toujours aléatoirement) le nombre de rues manquantes (uniquement entre voisins).

Pour gérer le problème de la distance maximale entre deux carrefours, nous utilisons la formule suivante qui nous donne la longueur d'un côté du carré représentant la ville de départ (ville "américaine") :

$$longueurCote = \frac{distanceMax}{\sqrt{2}} 0.8$$

Nous multiplions par 0.8 pour garantir que la distance maximale soit respectée car dans notre cas, on regarde la distance à vol d'oiseau alors qu'il faut regarder le chemin le plus court. En effet, nous avons mesuré qu'en moyenne, la distance à vol d'oiseau est d'environ 20% plus courte, d'où le 0.8 !

2.4.2 Recherche du plus court chemin (PCC)

Pour l'acheminement des clients et pour les décisions du système, il est nécessaire de connaître la distance et le chemin le plus court d'un endroit à un autre du graphe. D'après la donnée du problème, nous avons des stations et des clients situés au milieu des arcs. Il s'agit donc de trouver le chemin le plus court d'un arc à un autre. Le chemin renvoyé contient tous les sommets/arcs du chemin ainsi que la distance totale.

PCC entre deux arcs

Pour trouver un PCC entre deux arcs, on aurait pu imaginer de compléter le graphe avec des sommets au centre de chaque arc. Mais pour minimiser la taille du graphe, nous avons

choisi plutôt de rechercher et de comparer les chemins depuis les sommets qui composent les extrémités de chacun des deux arcs. Nous avons donc au maximum quatre chemins à calculer pour un PCC entre deux arcs.

Dijkstra

Pour trouver les PCC entre deux sommets nous utilisons l'algorithme de Dijkstra qui va en fait nous donner l'arbre des PCC d'un sommet de départ. Pour diminuer le temps de calcul nous avons d'abord imaginé de stocker au fur à mesure ces arbres, pour ne pas le reconstruire à chaque fois. Finalement, nous avons décidé de créer une fois pour toutes cet arbre pour chaque sommet et l'enregistrer dans un ficher. L'algorithme de Dijkstra parcourt tout le graphe depuis un sommet de départ et nous donne son arbre des PCC. Pour stocker et pour pouvoir construire le chemin qui nous intéresse, on va enregistrer pour chaque sommet de l'arbre, le sommet qui permet de l'atteindre et la distance qui le sépare du sommet de départ. Ainsi nous pouvons retrouver le chemin en partant du sommet à atteindre et en remontant les parents jusqu'au départ.

Recherche de la station la plus proche

Un cas particulier dans la recherche de chemin pour cette simulation est le chemin le plus court pour retourner à une station. Ce cas est très fréquent car chaque fois qu'un taxi a fini une course, il doit retourner dans la station libre la plus proche. Pour accélérer cette recherche, nous avons décidé de stoquer cette information pour chaque position (=arc). Après avoir générés les arbres des PCC, on va calculer pour chaque arc les distances à chaque station. Ainsi, pendant la simulation, il suffit de consulter la liste des stations triées par leur distance, et de prendre la première de libre.

Performances

Tous ces pré-calculs permettent d'améliorer la vitesse d'exécution, mais augmentent le volume de données à stocker. Pour limiter cette taille et donc le temps de chargement, les sommets, les arcs et les stations sont indexés par des entiers. Les données sont ainsi stockées sous forme de listes plutôt que sous forme de dictionnaires.

2.5 Les gestionnaires

Le rôle d'un gestionnaire est de regrouper des instances de différents objets. Il dérive d'un dictionnaire et 2 méthodes ont été définies pour permettre de le gérer un minimum. Tout comme le graphe, les gestionnaires sont des singletons. Cela signifie qu'ils sont uniques.

2.5.1 Gestionnaire des stations

Le rôle du gestionnaire de stations est de regrouper les stations qui sont employées pour la simulation. Il doit aussi permettre d'accéder à une station en donnant son numéro et de trouver la station la plus proche d'une position donnée.

Quand on demande le taxi suivant à une station, celle-ci renvoie celui qui est en tête de la liste de taxis. La politique de gestion de cette liste de taxi est FIFO.

Recherche d'une station

Pour simplifier l'écriture de politiques, une méthode privée de la classe est chargée de trouver la station la plus proche. Cette méthode prend plusieurs paramètres permettant de modifier son comportement. Des méthodes plus simples et publiques ont été créées et servent à appeler cette méthode privée de choix de station. Ces "petites" méthodes prennent un nombre de paramètres

restreint et permettent ainsi de choisir une station plus simplement qu'en appelant une seule et même méthode avec une énorme liste de paramètres.

Cette méthode privée demande au graphe une liste des stations ordonnées suivant la distance entre la station et la position courante. Il suffit simplement de parcourir cette liste et de renvoyer la première station qui satisfait toutes les contraintes (taux d'occupation, etc). Si aucune station n'existe, la méthode renvoie $None$.

Il est possible de faire en sorte qu'une partie des stations soient tabous. C'est-à-dire que la station qu'on renvoie ne peut pas faire partie des stations tabous. Cette option est utilisée quand un taxi fait "le tour" des stations jusqu'à ce qu'il en trouve une avec une place de libre. Il va, à chaque fois, à la station la plus proche et qui non tabou. Si il n'y avait pas cette liste de tabous, le taxi ferait des aller-retours entre 2 stations jusqu'à ce qu'une place se libère dans l'une des 2.

La complexité de la recherche d'une station est de $O(s/2)$, s étant le nombre de stations. En effet, vu que les stations sont dans une liste, il faudra parcourir en moyenne la moitié de la liste avant de trouver une station correspondant aux différents critères de recherche.

2.5.2 Taxis

Un taxi est principalement caractérisé par son état. Celui-ci peut être parmis les suivants :

- A l'arrêt, donc en station.
- En route pour aller chercher un client.
- En train de conduire un client.
- Sur le chemin de retour à une station.

Ces états changent à chaque fois que le taxi reçoit un évènement. A chaque changement d'état, le chemin que le taxi doit suivre change aussi. Un sémaphore a du être mis pour limiter les accès à la variable dans laquelle chemin le chemin est enregistré. En effet, il pouvait arriver que pendant que le taxi change d'état, l'affichage lui demande sa position, et vu que la recherche de la position a besoin du chemin, on se retrouvait dans un problème de concurrence et une erreur pouvait apparaître. Cette concurrence provient du fait que l'affichage n'est pas dans le même thread que le programme principal. L'utilisation d'un sémaphore permet donc d'éviter qu'une erreur se produise.

2.5.3 Gestionnaire des taxis

A chaque fois qu'on ajoute un taxi dans le gestionnaire de taxis, une instance de taxi est créée et elle est ajoutée au gestionnaire. A ce moment, on affecte également le taxi directement à une station dont le no a été passé lors de la demande d'ajout d'un taxi.

Recherche d'un taxi

Pour simplifier l'écriture de politiques, une méthode privée de la classe est chargée de trouver un taxi. Cette méthode prend plusieurs paramètres permettant de modifier le comportement de celle-ci. Des méthodes plus simples et publiques ont été créées et servent à appeler cette méthode privée. Ces "petites" méthodes prennent un nombre de paramètres restreint et permettent ainsi de choisir un taxi plus simplement qu'en appelant une seule et même méthode avec une énorme liste de paramètres.

Si aucune station ne contient de taxi ou que le gestionnaire de taxi ne contient aucun taxi, la méthode retourne $None$.

La complexité de cette méthode est en $O(s+\frac{t}{4})$, s étant le nombre de stations et le t le nombre de taxis.

2.5.4 Gestionnaire de préférences

Le gestionnaire de préférences est consultable et modifiable depuis n'importe où et tout au long de la simulation.

Au démarrage il charge les options contenues dans le fichier texte $simul.cfg$ qu'il fusionne avec le $params.cfg$ contenu dans le dossier de la ville. Les options peuvent être modifiées lors de l'exécution. Aucune modification n'est faite au fichier, seule l'édition manuelle le modifie.

2.5.5 Gestionnaire de politiques

Le gestionnaire de politiques permet le chargement dynamique des politiques de SimTaxi. A son instanciation il va essayer de charger toute les politiques se trouvant dans un répertoire donné. Le répertoire par défaut et le répertoire *politiques* de l'arborescence de SimTaxi. Pour que le chargement s'effectue sans erreurs, il faut que les fichiers **.py* se trouvant dans le répertoire soient tous des politiques. Si ce n'est pas le cas l'exception *ErreurPolitiques* est levée.

La méthode *politiques* permet d'obtenir la liste des politiques chargées. Ces politiques sont déjà instanciées, elles sont donc directement utilisables.

2.6 Mise en place des stations

Cette opération nécessite de connaître un certain nombre d'informations :

- Le graphe associé à la simulation.
- Le nombre de stations désirées ainsi que leur nombre respectif de places (représenté par une liste d'entiers). Si le nombre de stations est supérieur à la taille de la liste donnée, le nombre de places les stations restantes correspondra à la dernière valeur de la liste.

Tout d'abord, notons que les stations sont situées au centre d'un arc et que deux stations ne peuvent pas se trouver sur le même arc. De plus afin d'améliorer quelque peu leur répartition, deux stations ne pourront se trouver sur deux arcs consécutifs. Les stations dont le nombre de places est indiqué dans la liste sont placées au centre de la ville : la 1ère au centre, la 2e légèrement à l'est (du centre), la 3e légèrement à l'ouest, la 4e légèrement au sud, la 5e légèrement au nord et ainsi de suite. Ensuite, les stations dont le nombre de places n'est pas explicitement indiqué dans la liste (le nombre de places est implicitement égal à la dernière valeur de la liste) sont placées aléatoirement sur le graphe. Finalement, nous allons indiquer au gestionnaire de stations qu'il peut ajouter chaque station comportant le nombre de places choisi à l'endroit voulu.

2.7 Mise en place des taxis

Signalons pour commencer que cette opération n'est réalisable que si le graphe associé comporte déjà des stations, faute de quoi la méthode se terminera sans avoir placé un seul taxi. Mis à part le graphe, il n'est nécessaire de connaître que le nombre de taxis à placer. Avant de commencer à placer les taxis, on récupère la liste des stations associées au graphe. Ensuite, pour placer un taxi, on choisit une station aléatoirement dans la liste et on lui affecte un taxi. Après cette affectation, si la station ne comporte plus de place, on la supprime de la liste des stations. On répète l'opération jusqu'à ce que le nombre désiré de taxis ait été affecté ou alors jusqu'à ce qu'il n'y ait plus de station dans la liste.

2.8 Adaptation de la moyenne des courses

Comme nous avions pu le constater à la fin du premier semestre, lors de la présentation de notre projet après quelques mois de travail, la génération de la demande nécessitait passablement

de modifications. A cette époque, les demandes étaient générées avant le début de la simulation, ce qui bien sûr n'est pas très réaliste. Cette façon de faire avait cependant l'avantage de fortement décharger le processeur durant la simulation à proprement parler, ainsi que de faciliter l'adaptation de la moyenne des distance des courses. Le principal défi de cette deuxième partie de projet a donc été de trouver une technique permettant d'adapter la moyenne des courses durant la simulation.

Pour réaliser ceci, la méthode suivante nous a été proposée : Il s'agit de déterminer un paramètre associé à une fonction simple, comme par exemple L dans une fonction triangulaire, avec le maximum de cette fonction à 1. A chaque distance serait associé un nombre se trouvant dans l'intervalle $[0, 1]$. Plus précisément, la distance 0 ainsi que la distance maximale d'un parcours du graphe (une valeur à peine supérieure en fait, de manière à ne pas rendre impossible la réalisation de ce parcours de longueur maximale) prendraient la valeur 0, alors que la distance L prendrait la valeur 1.

Ensuite, lors de la génération d'une course (cf. section suivante pour plus de détails sur ce point), on calcule le résultat de la distance de cette course par notre fonction, puis on génère une valeur aléatoire dans l'intervalle $[0, 1]$. Si la valeur de la variable générée aléatoirement est plus petite que le résultat de la distance de notre course par la fonction, on planifie la course, sinon, on en génère une autre.

En pseudo-code, cela nous donne :

- Générer une course
- Calculer $D =$ le résultat de la distance de la course par la fonction f_L
- Générer $U =$ variable aléatoire uniforme dans $[0, 1]$
- Si $U \leq D$ alors : Planifier la course en question
- Sinon : Recommencer l'algorithme

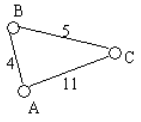
Comme on peut le remarquer, plus la distance d'une course se trouvera proche du paramètre L, plus cette course aura de chances d'être planifiée.

Maintenant, il convient de savoir comment déterminer ce paramètre L de manière à atteindre une distance moyenne des courses égale à M.

Cette opération sera réalisée par une quête dichotomique. La valeur de L est initialisée à $\frac{DistanceMaxGraphe}{2}$. Ensuite, on calcule la moyenne des distances de toutes les courses en pondérant chaque distance par son résultat par la fonction simple. Si la valeur ainsi trouvée est inférieure à M, alors il faudra recommencer le calcul de la moyenne en modifiant la valeur de L à $\frac{DistanceMaxGraphe+L}{2}$. Si au contraire la valeur trouvée s'est avérée être supérieure à M, il convient de modifier L à $\frac{L}{2}$. Ceci correspond à la deuxième étape de la dichotomie. Lorsque que la moyenne calculée est égale à la moyenne cherchée (ceci peut aussi bien arriver à la première qu'à la dixième étape de la dichotomie), la valeur de L est celle qui permettra d'adapter la moyenne comme on le souhaite.

L'exemple ci-dessous permettra certainement au lecteur un peu désorienté de mieux comprendre le calcul du L.

Supposons le graphe suivant :

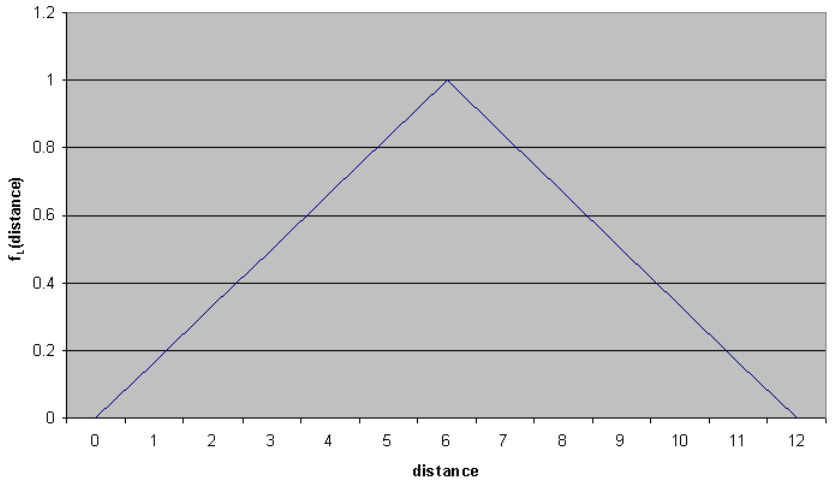


Si on considère que la fonction f_L est une fonction triangulaire simple, valant 1 à L, et 0 pour des distances de 0 ou 12, et que la distance moyenne désirée est $M = 8$, la recherche du L se déroule comme suit :

Recherche dichotomique, première étape :

$$L = \frac{DistanceMaxGraphe}{2} = 6$$

Le graphe de la fonction est donc le suivant :



Pour AB $f_L(4) = 4/6 = 2/3$
Pour AC $f_L(11) = 1/6$
Pour BC $f_L(5) = 5/6$

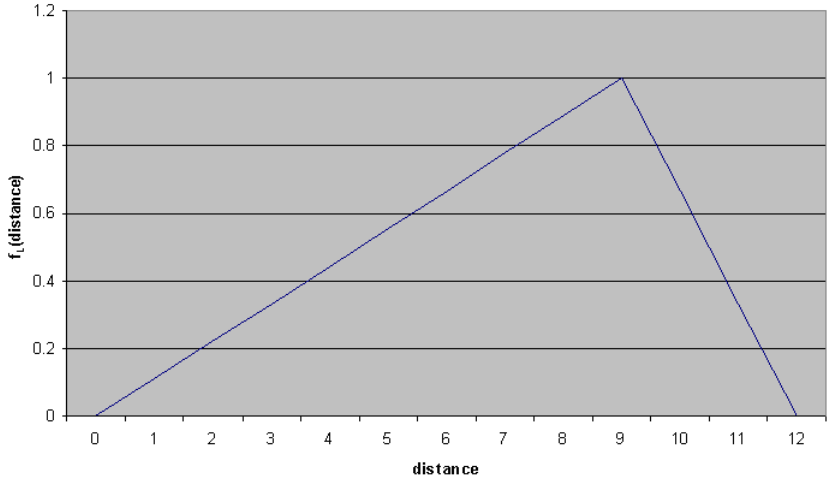
$$\Longrightarrow Moyenne = \frac{\frac{2}{3}4 + \frac{1}{6}11 + \frac{5}{6}5}{\frac{2}{3} + \frac{1}{6} + \frac{5}{6}} = \frac{26}{5} = 5.2 < 8 = M$$

Il faut augmenter L :

$$L \leftarrow \frac{L + DistanceMaxGraphe}{2} = \frac{6 + 12}{2} = 9$$

Recherche dichotomique, deuxième étape : $L = 9$

Le graphe de la nouvelle fonction f_L a donc l'allure suivante :



Pour AB $f_L(4) = 4/9$
Pour AC $f_L(12) = 3/9$
Pour BC $f_L(5) = 5/9$

$$\Longrightarrow Moyenne = \frac{\frac{4}{9}4 + \frac{3}{9}12 + \frac{5}{9}5}{\frac{4}{9} + \frac{3}{9} + \frac{5}{9}} = \frac{77}{12} = 6.4 < 8 = M$$

Il faut encore augmenter L :

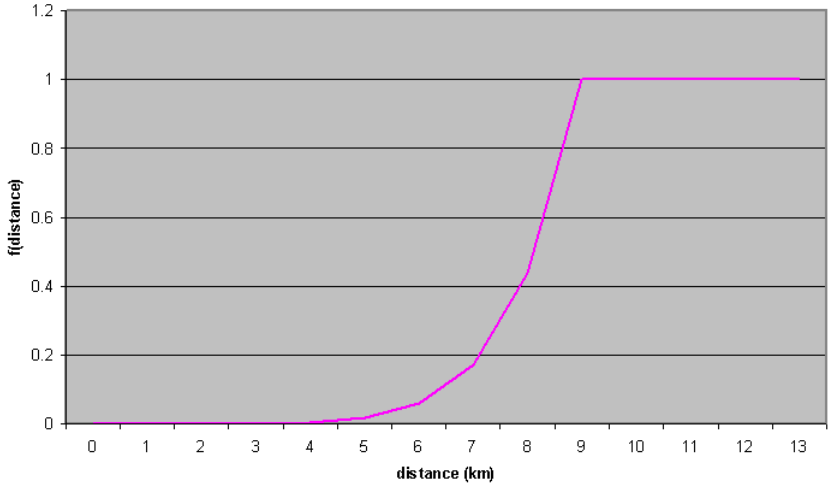
$$L \leftarrow \frac{L + DistanceMaxGraphe}{2} = \frac{9 + 12}{2} = 10.5$$

Etc... jusqu'à avoir trouvé une valeur de L fournissant la moyenne cherchée.

Dans le cadre de notre simulation, il a été nécessaire d'adapter la fonction f_L afin d'obtenir des résultats valides. En effet, nous désirons atteindre une distance moyenne de 8 km, alors que la distance de la course la plus longue et d'à peine plus de 12 km. Étant donné la nature de notre graphe, il est évident que les courses de petites distances sont bien plus nombreuses que celle de longueur supérieure à la moyenne cherchée. De ce fait, une simple fonction triangulaire n'aurait jamais permis d'atteindre la moyenne désirée. Pour remédier à ce problème, nous avons remplacé la fonction triangulaire par une fonction favorisant largement les longs trajets :

$$f_L(distance) = \frac{distance^7}{L^7} si\ distance < L$$
$$f_L(distance) = 1\ sinon$$

ce qui nous donne une fonction de la forme suivante :



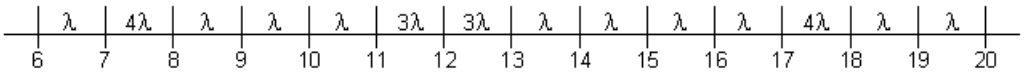
Grâce à cette fonction, nous sommes parvenus à adapter la moyenne des distances des courses durant la simulation.

Notons tout de même qu'en raison de la taille du graphe, la recherche du L a duré plusieurs heures. Heureusement, cette opération n'a besoin d'être effectuée qu'une seule fois pour un réseau et une moyenne désirée donnés.

2.9 Génération de la demande

Comme nous l'avons déjà mentionné dans la section précédente, la génération de la demande a maintenant lieu durant la simulation, et non plus avant le départ de celle-ci. Afin de fixer les dates des événements, nous avons donc naturellement eu recours à un processus de Poisson. Notre problème suppose qu'il y aura $2'700$ clients en 24 heures ($86'400$ secondes). On peut donc calculer la paramètre λ de notre loi : $2'700$ courses en $86'400$ secondes, donc $2'700/86'400 = 1/32$ course

par seconde. En d'autre termes, cela signifie qu'il y aura en moyenne une course toutes les 32 secondes. Cependant, nous n'avons pas réparti ces courses de manière identique sur toute la journée : elles seront réparties de la manière suivante entre $6h00$ et $20h00$:



Cette répartition respecte bien la demande de $2'700$ courses, étant donné que la somme des λ vaut 24 (24 heures dans la journée). On pourra cependant constater certains phénomènes d'heures de pointe : il y aura par exemple quatre fois plus de demandes entre $17h00$ et $18h00$ qu'entre $18h00$ et $19h00$. Ceci sera très intéressant à analyser dans la section traitant les statistiques.

Au niveau de notre implantation de cette façon de procéder, nous disposons dans notre classe GenerateurDemande d'un attribut privé dateActuelle nous permettant de connaître la date du dernier client généré. La méthode genererProchainClient retournera un événement client dont la date a été calculée en fonction de dateActuelle : on ajoute à cette date la valeur retournée par un processus de Poisson de paramètre λ, 3λ ou 4λ (suivant l'heure, comme présenté dans le tableau ci-dessus). L'attribut dateActuelle sera finalement mis à jour avec la valeur de cette nouvelle date.

En ce qui concerne le trajet du client, on génère aléatoirement un point de départ et un point d'arrivée puis on applique la méthode du L (cf. section précédente pour plus de détails) au chemin le plus court reliant ces deux points.

Finalement, lorsque nous disposons des deux éléments constituant un événement client, une date et un parcours, il suffit de retourner un nouvel objet EvClient formé de ces deux éléments.

2.10 Les politiques

2.10.1 Description

SimTaxi permet de définir différentes politiques de gestion des courses. Les politiques permettent de répondre aux deux questions suivantes : *Quel taxi va prendre en charge un client ?* et *A quel station un taxi va se rendre une fois sa course terminée ?*.
SimTaxi fournit les primitives pour permettre de répondre à ces deux questions. Une politique consiste à assembler ces différentes briques.

2.10.2 Conception

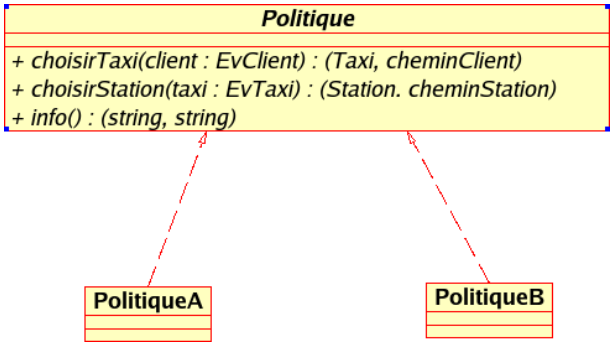


FIG. 2.6 – *Diagramme de classe des politiques*

La classe abstraite *Politique* (voir figure 2.6) fournit l'interface permettant de réaliser une politique. Une politique doit redéfinir les méthodes abstraites *choisirTaxi* et *choisirStation* qui permettent réciproquement de choisir le taxi à affecter à un client et de choisir une station pour un

taxi. La méthode abstraite *info* doit aussi être redéfinie. Elle doit retourner un tuple contenant le nom de la politique et une description de celle-ci.

Ce sont les gestionnaires de stations, de taxis et de statistiques (voir section 2.5) qui fournissent les briques de base à la réalisation de la politique en mettant à disposition des sous-programmes permettant de retourner le taxi le plus proche d'un client, ou un taxi d'une station particulière etc... Il en est de même pour les stations.

Il est possible de changer de politique grâce à la méthode *modifierPolitique* du central.

2.11 Interface graphique

Le but de l'interface graphique est de fournir un support visuel au déroulement de la simulation, elle permet, en temps réel, de voir l'évolution des taxis, l'état des stations ainsi que d'autres données moins rapidement cernables au travers de chiffres.

Ceci est important afin de rapidement se rendre compte du bon fonctionnement des différents algorithmes, le comportement des taxis peut être analysé en un clin d'oeil afin d'en vérifier la cohérence.

Il est primordial d'avoir une vision graphique du problème, ceci afin de faciliter sa compréhension et de pouvoir plus rapidement imaginer de nouveaux algorithmes.

2.11.1 Interface utilisateur

L'interface de SimTaxi est réalisée avec wxWindows pour Python (wxPython), le choix s'est naturellement orienté vers cet outil car il possède beaucoup de points forts comme sa facilité d'utilisation, son grand choix de composants (menu, bouton, etc..) et bien sûr sa gratuité.

Diagramme des classes

Voici le diagramme statique de l'interface graphique de SimTaxi.

'FenetreControl 'est la fenêtre principale, elle possède toutes les autres fenêtres ainsi qu'un certain nombre de panel de contrôle comme 'AffichagePan ', 'ControlPan 'et 'PolitiquePan '. L'ajout de panel est décrit dans la prochaine section.

Ajout de nouveaux panels

La fenêtre de contrôle contient un ensemble de sous cadres (panels) permettant le contrôle de la simulation ou de l'affichage, actuellement il y a un panel pour gérer l'affichage de certain élément graphique, un panel pour le contrôle du temps de la simulation et un panel pour choisir la politique.
Un panel est représenté par un fichier dans le répertoire '/gui/interfaces ', Pour ajouter un nouveau panel il faut donc créer un nouveau fichier en se calquant sur la structure de ceux déjà existants comme 'ControlPan.py '. Puis éditer le fichier '/gui/FenetreControl.py 'et suivre les instructions de la ligne 53.

2.11.2 Interface OpenGL

Au regard du nombre de taxis, routes et stations à afficher en temps réel il est nécessaire d'utiliser une bibliothèque graphique de bas niveau contrairement aux outils de dessin fournis par wxWindows. Il faut donc un affichage rapide et pouvant permettre l'affichage d'animation (double tampon d'affichage), le choix de OpenGL s'est alors naturellement fait.

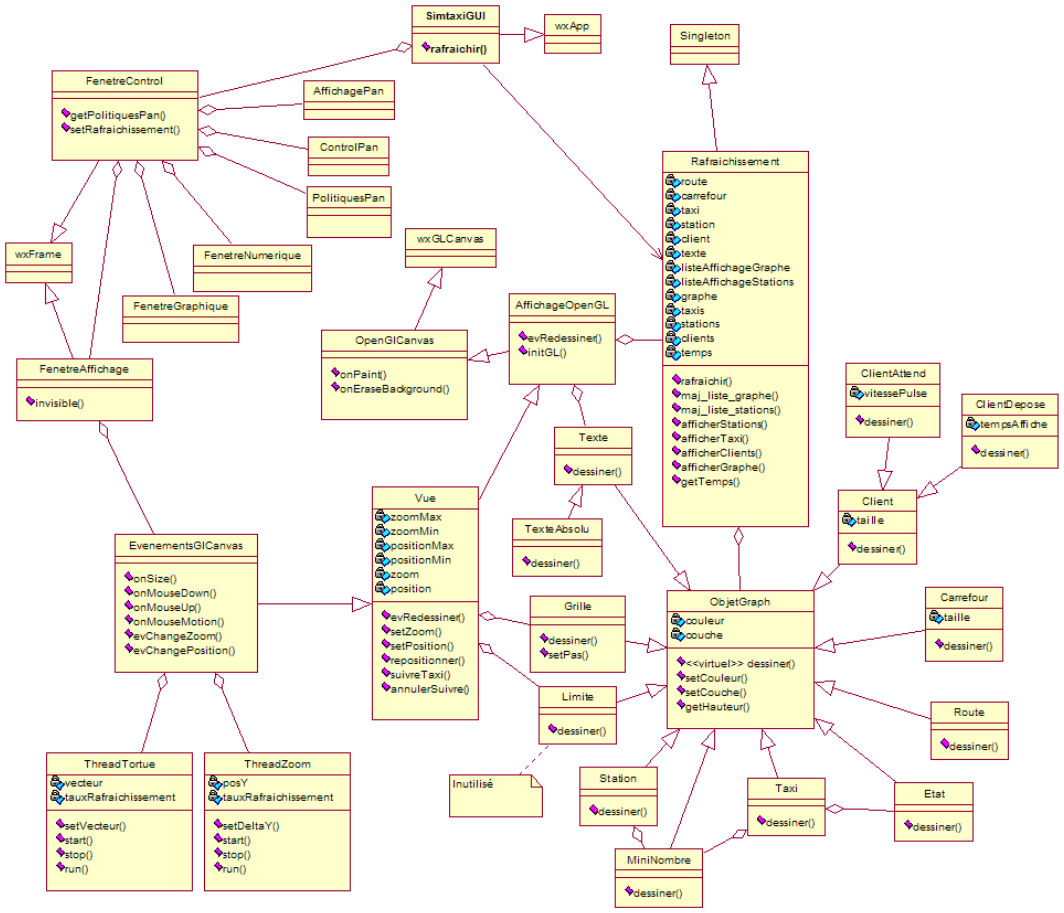


FIG. 2.7 – *Diagramme de classe de l'interface graphique*

OpenGL est à la base destiné à afficher des scènes en trois dimensions, dans notre cas il n'est pas nécessaire d'avoir une telle possibilité. La caméra est donc placée sur l'axe des Z et une projection orthogonale est utilisée pour avoir une vue en deux dimensions.

Les objets de la scène (taxis, stations etc.) sont placés sur différentes couches qui sont définies par une valeur en Z. On peut donc facilement définir quel objet est au dessus d'un autre. Par exemple les taxis sont au dessus des routes.

Chaque objet est composé à l'aide de polygones, il est important de réduire au maximum le nombre de polygone des objets, par exemple un taxi n'est formé que d'un seul polygone et une route de deux polygones, afin de diminuer le temps de calcul de l'affichage.

Chapitre 3

Les politiques

3.1 Description des politiques implantées

3.1.1 Politique du plus près

Cette politique est implémentée dans le fichier *PolitiquePlusPres.py*. Elle a pour but de minimiser les trajets fait par le taxi à vide.

Affectation d'un taxi à un client

L'affectation ce fait comme suit : on choisit le taxi se trouvant dans la station la plus proche du client pour aller chercher celui-ci.

Affectation de la station de retour au taxi

Une fois que le taxi a déposé son client, celui-ci se rendra dans la station la plus proche de lui. Si à l'arrivée de la station, celle-ci est pleine, il se déroutera sur la station la plus proche. On itère le processus jusqu'à ce que le taxi trouve une station avec une place de libre.

3.1.2 Politique du plus près version 2

Cette politique est implémentée dans le fichier *PolitiquePlusPres2.py*. Elle a deux objectifs : minimiser les trajets du taxi fait à vide et assurer un taux d'occupation minimum des stations.

Affectation d'un taxi à un client

Même algorithme que pour *Politique du plus près*.

Affectation de la station de retour au taxi

Même algorithme que pour *Politique du plus près*, sauf que si on se trouve dans le cas où une station se trouve dans un rayon de 3km du taxi et que celle-ci est pleine à moins de 20%, alors le taxi se rendra dans cette station.

3.1.3 Politique du plus près avec réservation

Cette politique est implémentée dans le fichier *PolitiquePlusPresReser.py*. Elle a comme objectif de minimiser les trajets du taxi fait à vide mais de manière plus performante que la Politique du plus près.

Affectation d'un taxi à un client

Le taxi le plus proche du client va le chercher. Celui-ci peut être dans une station ou en train de rentrer à une station (détournement).

Affectation de la station de retour au taxi

Une fois le client déposé, le taxi se rend dans la station la plus proche dans laquelle il y a encore des places libres. Pour être sûr que la station ne soit pas pleine lorsque le taxi y arrive, celui-ci fait une réservation.

3.1.4 Politique assurant un taux minimum d'occupation des stations

Cette politique est implémentée dans le fichier *RemplirStation.py*. Elle a comme objectif d'assurer un taux minimum des stations de 15 %.

Affectation d'un taxi à un client

Le taxi qui va cherche un client est prix dans la station la plus proche qui est pleine à plus de 60%. Si aucune station ne répond à ce critère, on prend le taxi le plus proche.

Affectation de la station de retour au taxi

Le taxi est placé dans une station qui est remplie à moins de 15%. Si aucune station ne remplit ce critère on renvoie le taxi dans la station la plus proche (sans faire de réservation).

3.1.5 Politique de la demande

Cette politique est implémentée dans le fichier *PolitiqueDemande.py*. Elle a comme objectif d'amener les taxis dans les stations où la demande est la plus forte, tout en essayant de préserver une certaine répartition des taxis dans les stations. Pour cela, elle utilise un nombre aléatoire compris entre 0 et 1, représentant un pourcentage d'occupation comme critère de décision. Le germe de la valeur aléatoire ce trouve dans le fichier *params.cfg* de la ville et s'appelle *germePolitiqueDemande*.

Affectation d'un taxi à un client

Le taxi le plus proche du client va le chercher. Celui-ci peut être dans une station ou en train de rentrer à une station (détournement).

Affectation de la station de retour au taxi

Nous tirons une demande d aléatoire comprise entre 0 et 1. Nous parcourons la liste des stations et prenons la première qui a un taux de demande plus grand ou égale à d. Dans le cas où aucune station ne répond à ce critère, la station la plus proche du taxi est choisie. Nous effectuons ensuite une réservation dans la station sélectionnée. Si la demande de réservation échoue nous essayons d'effectuer une réservation dans la prochaine station la plus proche du taxi jusqu'à ce que la demande de réservation réussisse.

3.2 Comparaison des politiques

Notre simulation a pour but principal de trouver la meilleure politique à adopter pour minimiser les trajets des taxis à vide. Donc, pour pouvoir déterminer quelle est la meilleure des politiques simulées, il faut les comparer les unes aux autres. Pour ce faire, nous avons donc mis en place un outil permettant d'afficher graphiquement les résultats des politiques après simulation.

3.2.1 Introduction

Étant donné que notre programme de simulation ne permet pas de faire tourner plusieurs politiques simultanément, il nous est donc impossible de comparer les politiques en cours de simulation. De plus, les statistiques ne seraient pas comparables étant donné qu'à un instant quelconque, le nombre de clients pris en charge diffère d'une politique à l'autre.

Nous avons donc créé un outil de comparaison des résultats après simulation. Le principe est simple : lorsqu'une simulation se termine, on enregistre les statistique dans un fichier nommé $NomDeLaPolitique.sta$ et qui se trouve dans le dossier de la ville. Ensuite, notre programme récupérera tous les fichiers de statistiques se trouvant dans le dossier de la ville choisie et superposera les courbes des politiques trouvées.

Voici la liste des graphiques affichés :

- Le nombre de taxis en fonction des distances parcourues à vide.
- Le nombre de taxis en fonction des durées des temps de service.
- Le nombre de taxis en fonction des durées des temps d'attente.
- Le nombre de taxis en fonction des distances parcourues avec clients.

Pour générer ces courbes, nous avons discrétisé les statistiques afin d'obtenir un certain nombre de taxis pour une certaine tranche donnée. Nous appelons la largeur d'une tranche "le pas". Le pas détermine le nombre de mètres pour les distances et le nombre de secondes pour les durées. Les courbes sont aussi représentées par le biais d'histogrammes afin de mettre en évidence les différentes tranches. Cependant, il est difficile d'interpréter les histogrammes que nous affichons, c'est pourquoi nous avons gardé les courbes.

Le graphique représentant le nombre de taxis en fonction des distances parcourues avec clients peut sembler inutile étant donné que la demande doit être la même pour toutes les politiques. Cependant, ce graphique permet de vérifier la véracité des statistiques que l'on compare. En effet, si les courbes des différentes politiques ne se superposent pas (ou presque pas), c'est qu'il y a un problème. Ainsi, on peut par exemple s'apercevoir très vite que les simulations n'ont pas toutes été effectuées pendant une période identique, ce qui fausserait complètement les comparaisons.

3.2.2 Utilisation du logiciel

Lancement du programme

Le lancement du programme se fait à l'aide de $comparerPolitiques$: Il faut donner un paramètre au lancement du programme en l'état du dossier où se trouve la ville[1] et aussi le pas (par défaut 1000).

```
./comparerPolitiques dossierDeLaVille/ [lePas]
```

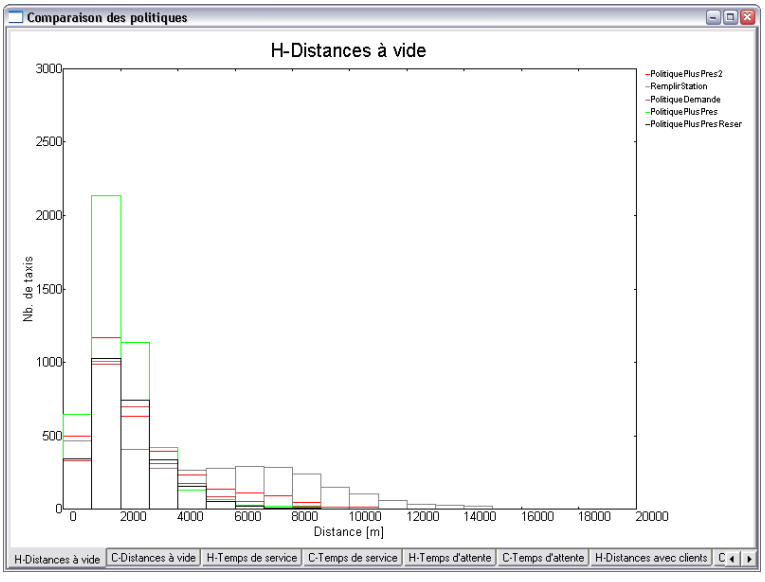
[1] Ne pas oublier le "slash" à la fin.

Un démarrage correct doit afficher les lignes suivantes[2] :

```
La politique "PolitiqueDemande" a été trouvée.
La politique "PolitiquePlusPres" a été trouvée.
La politique "PolitiquePlusPres2" a été trouvée.
La politique "PolitiquePlusPresReser" a été trouvée.
La politique "RemplirStation" a été trouvée.
Importation des données...

Distances totales pour "PolitiquePlusPres2":
Avec clients: 21350 [km]
Sans clients: 8887 [km]
Distances totales pour "RemplirStation":
Avec clients: 21702 [km]
Sans clients: 17025 [km]
Distances totales pour "PolitiqueDemande":
Avec clients: 21350 [km]
Sans clients: 6597 [km]
Distances totales pour "PolitiquePlusPres":
Avec clients: 21921 [km]
Sans clients: 9296 [km]
Distances totales pour "PolitiquePlusPresReser":
Avec clients: 21350 [km]
Sans clients: 6017 [km]

Calcul des courbes...
Calculs terminés: affichage des graphiques...
```



Notre programme recherche d'abord les politiques disponibles (fichiers $*.sta$). Ensuite, il importe les données contenues dans les fichiers et affiche les distances totales de chaque politique. Il calcule les différents points des courbes et des histogrammes et pour terminer, il crée la fenêtre et dessine tous les graphiques (voir la figure ci-dessus).

Affichage d'un graphique

Pour afficher le graphique que l'on désire, il suffit de choisir l'onglet correspondant. La première lettre du titre d'un graphique indique son type : un "H" indique que la représentation se fait par le

[2] Si toutes les politiques ont été simulées pendant 24 heures.

biais d'histogrammes alors qu'un "C" indique que ce sont des courbes.

Terminaison du programme

Pour terminer le programme, il suffit de fermer la fenêtre comportant les graphiques.

3.2.3 Résultats et analyse des résultats

Toutes les simulations ont été effectuées sur une période de 24 heures avec la même demande. Ensuite, nous avons choisi un pas de 1'000 pour l'utilisation de notre programme de comparaison des politiques.

Les distances totales

Dans cette section, nous exhibons les distances totales obtenues pour chaque politique implémentée. Ces résultats sont basés sur la distance totale parcourue avec clients et celle parcourue sans clients. Notre but étant de minimiser la distance parcourue sans clients, il nous suffit donc de classifier les politiques en fonction de cette distance.

Cependant, les distances parcourues avec clients ne sont pas toutes identiques, bien qu'elles devraient l'être. Nous avons donc deux politiques qui nous posent problème : $RemplirStation$ et $PolitiquePlusPres$. D'après une première analyse du problème, il semblerait que le nombre d'événements clients générés soit différent avec ces politiques. Ceci ne devrait pourtant pas être le cas. Cependant, nous sommes certain que la demande est identique entre les simulations. Il apparaît donc que le bug provient du fait qu'il n'y pas tout le temps un taxi disponible pour prendre un client est que nous devons dans ce cas différer l'événement. Malgré que ceci soit la raison la plus probable de ce bug, nous n'en sommes pas sûr à 100%.

Pour que notre bug ne fausse pas nos résultats, nous présentons également la proportion de la distance parcourue sans clients par rapport à la distance totale parcourue (avec et sans clients). Il nous suffira alors de classifier les politiques selon ce critère. La politique la meilleure sera donc celle qui aura le plus petit rapport.

Voici donc nos résultats :

Politique du plus près

Distance totale avec clients : 21921 [km]

Distance totale sans clients : 9296 [km]

Rapport : 29.8%

Politique du plus près version 2

Distance totale avec clients : 21350 [km]

Distance totale sans clients : 8887 [km]

Rapport : 29.4%

Politique du plus près avec réservation

Distance totale avec clients : 21350 [km]

Distance totale sans clients : 6017 [km]

Rapport : 22.0%

Politique évitant que les stations ne se vident

Distance totale avec clients : 21702 [km]

Distance totale sans clients : 17025 [km]

Rapport : 44.0%

Politique de la demande

Distance totale avec clients : 21350 [km]

Distance totale sans clients : 6597 [km]

Rapport : 23.6%

Nous pouvons donc classer les politiques de la meilleure à la moins bonne comme ceci :

1. Politique du plus près avec réservation
2. Politique de la demande
3. Politique du plus près version 2
4. Politique du plus près
5. Politique évitant que les stations ne se vident

Analyse des graphiques

Nous n'allons pas faire une analyse de tous les graphiques car il sera difficile de distinguer les différentes courbes si le document est imprimé en noir blanc. Nous allons donc plutôt montrer comment interpréter les graphiques de façon à ce que l'utilisateur puisse le faire lui-même en exécutant le programme[3].

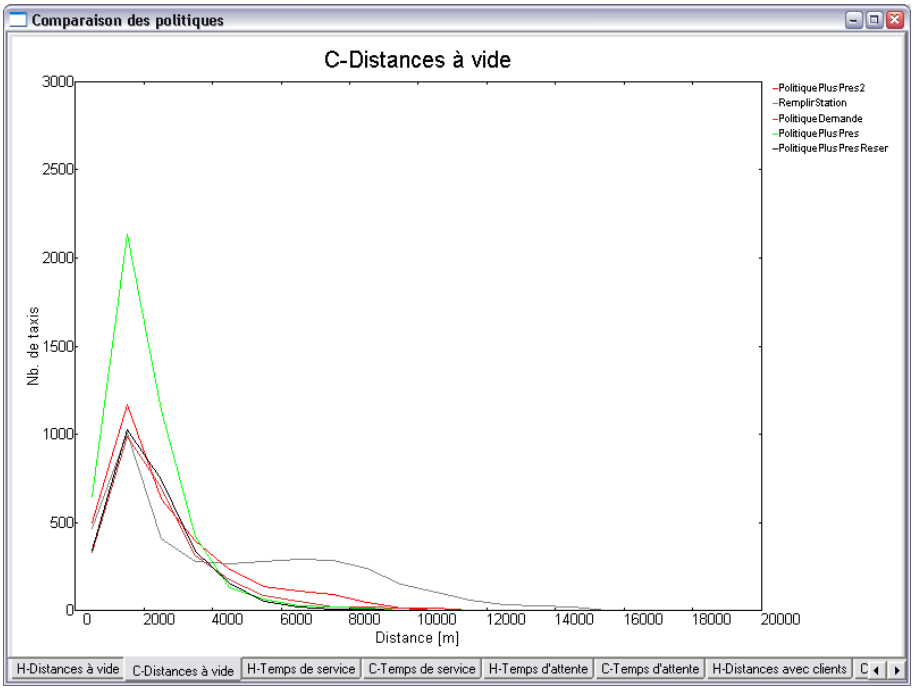


FIG. 3.1 – *Courbes du nombre de taxis en fonction des distances parcourues sans clients.*

[3] En couleur cette fois-ci !

Analysons donc la figure 3.1. Tout d'abord, notons que si nous avions une courbe qui serait plus basse que les autres en tous points, on pourrait en déduire que la politique correspondante serait la meilleure, mais ce cas est peu probable. En effet, certaines politiques minimisent plutôt les longues distances alors que d'autres minimisent plutôt les courtes. Cependant, il vaut mieux avoir une politique qui minimise le nombre de longues distances. Par exemple, sur la figure 3.1, on voit très bien que la politique $RemplirStation$ est mauvaise car sa courbe se situe bien au-dessus des autres dans les longues distances, alors que dans les petites distances, elle ne se distingue pas des autres. On peut aussi voir que la politique $PolitiquePlusPres$ se distingue nettement dans le nombre de petites distances. Cela s'explique par le fait que lorsqu'un taxi arrive dans une station déjà pleine, il va aller à la station suivante et que le trajet entre les deux stations est un autre trajet qui sera forcément assez court.

Les deux politiques décrites ci-dessus ne sont donc pas à considérer comme des bonnes politiques. Nous allons donc les laisser de côté pour la suite de l'analyse graphique.

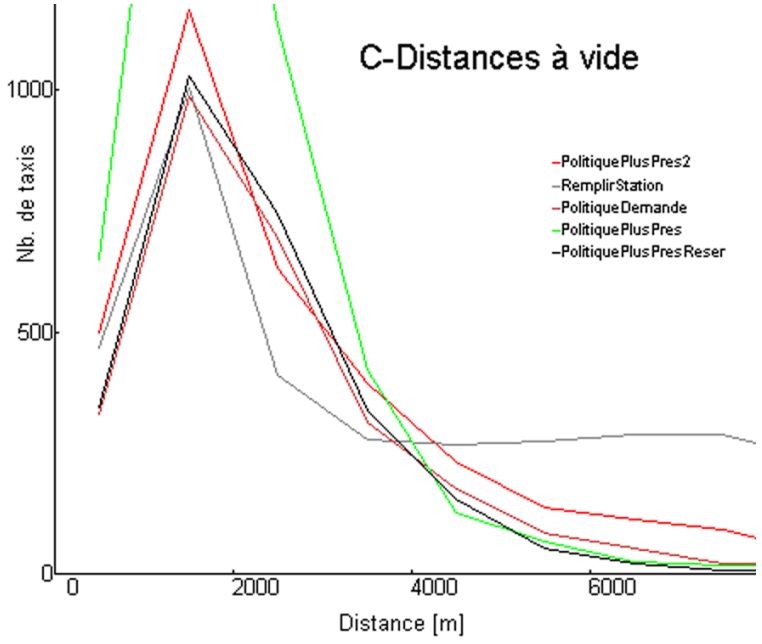


FIG. 3.2 – *Zoom de la figure 3.1.*

Si l'on regarde de plus près les courbes où elles se croisent (voir la figure 3.2), on peut constater que la politique $PolitiquePlusPresReser$ est la meilleure sur les longues distances alors que la politique $PolitiqueDemande$ est la meilleure sur les petites distances. Ces deux politiques sont très proches, mais notre préférence se porte sur la politique $PolitiquePlusPresReser$ étant donné qu'elle minimise plutôt les longues distances.

Nous pouvons donc dire qu'en règle générale, pour classifier les politiques, il suffit de regarder les courbes dans les longues distances pour autant que les courbes soient passablement groupées dans les courtes distances. Dans notre cas, on voit que cette règle fonctionne parfaitement si l'on tient compte du fait que la politique $PolitiquePlusPres$ se distingue nettement des autres dans les petites distances. On peut donc arriver graphiquement à dresser le même classement des politiques qu'en se basant sur les distances totales. De plus, les graphiques nous permettent de visualiser les points forts et les points faibles de chaque politique.

3.2.4 Conclusion

Pour les trois politiques "du plus près", il nous paraît évidant que celle avec réservation soit la meilleure d'entre elles étant donné qu'elle minimise la distance que les taxis doivent parcourir pour rentrer en station. Pour la politique qui évite que les stations ne se vident, elle est nettement la moins bonne car elle s'efforce à remplir, entre autre, des stations où la demande est faible.

Par contre, la différence entre les deux meilleures politiques[4] est faible (1.6%). Regardons ces politiques de plus près : si l'on se place du côté des taxis, la politique du plus près avec réservation est en effet meilleure, mais si l'on se place du côté des clients, on peut supposer que la politique de la demande permet de servir les clients plus rapidement, étant donné que les stations à forte demande sont mieux remplies. Par contre, l'algorithme de la politique de la demande est plus complexe et plus coûteux que celui de la politique du plus près avec réservation.

Nous préconisons donc d'utiliser une des deux meilleures politiques mais pas forcément la meilleure, car la décision doit être prise en fonction des paramètres cités dans le paragraphe précédant.

Notons aussi que d'autres politiques, que nous n'avons pas imaginées peuvent s'avérer meilleures. Ce qui est sûr, c'est qu'il faut faire des réservations lorsque les taxis rentrent en station et qu'il faut aussi permettre les détournements de taxis si l'on veut optimiser une politique.

[4] *Politique du plus près avec réservation* et *Politique de la demande*.

Chapitre 4

Documentation technique

La documentation technique a été directement extraite de nos codes source à l'aide d'Happy-Doc. Le format étant le HTML et ce genre de documentation n'étant pas vraiment agréable à lire d'une seule traite, mais plutôt par partie, elle n'a pas été imprimée. Mais elle est consultable dans le dossier $web/index.html$ de SimTaxi ou directement sur le site $http://SimTaxi.sf.net$. A noter que HappyDoc nous génère aussi notre site web.

Chapitre 5

Conclusion

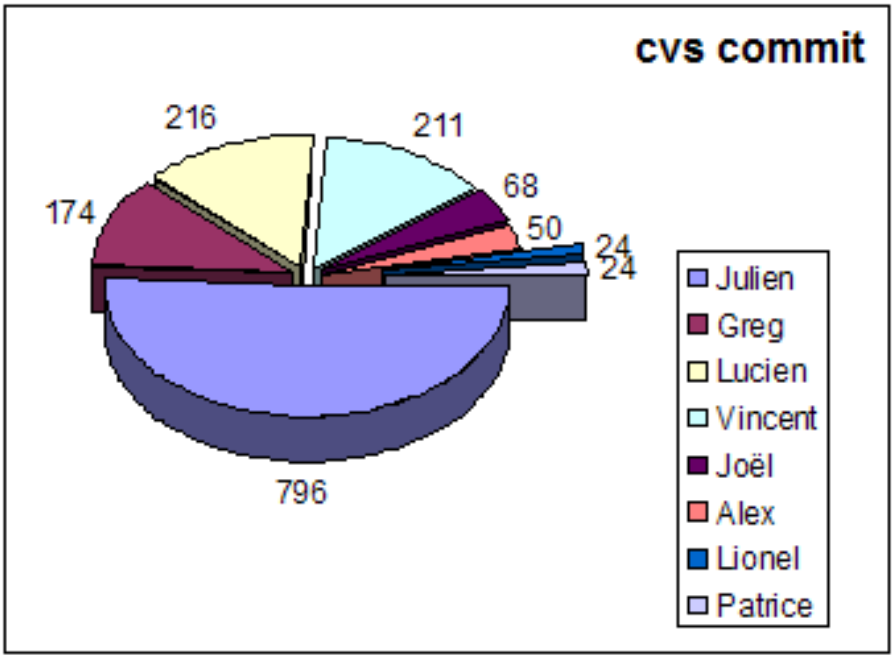


FIG. 5.1 – *1563 commit selon les statistiques du CVS*

Dans l'ensemble, le projet s'est bien déroulé, les problèmes de communication sont resté minimes. Même si le chef de projet a eu beaucoup de difficultés à faire travailler les membres du groupe, la motivation étant difficile à maintenir pour cette fin de formation.

Nos documents internes nous ont permis de travailler de manière uniforme et efficace. C'est grâce à eux que nous avons pu, dès le début, définir comment allaient être créées les différentes documentations. Cela nous a donc évité de partir chacun de son côté et de perdre du temps par la suite pour tout uniformiser.

Il reste certaines erreurs numériques dans la simulation. Il aurait fallu stopper le développement et mettre en place des structures de tests pour corriger le tout et le rendre plus robuste avant de continuer. Mais ce genre de techniques ne sont pas enseignées à l'eivd (tout comme la gestion de projet d'ailleurs). Nous sommes toujours handicapés par nos problèmes de performance, la simulation est lourde.

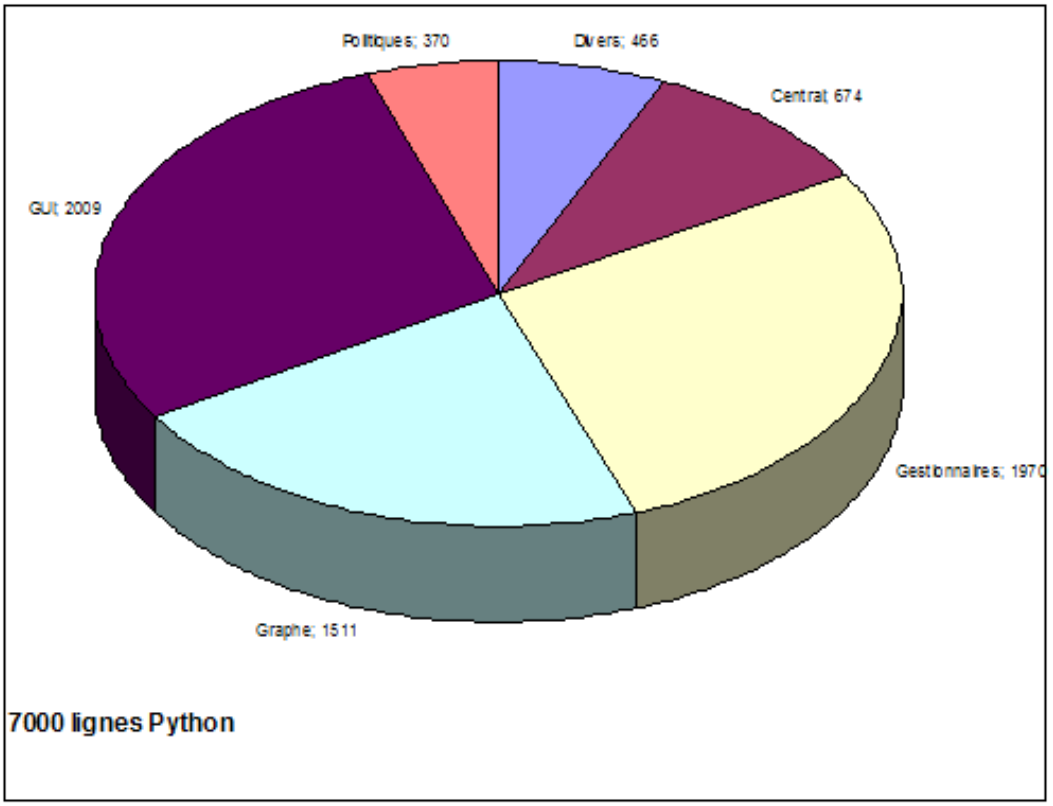


FIG. 5.2 – *Un total de 7000 lignes de codes exactement ! !*

Chapitre 6

Documentation Qualité

6.1 Remarque préliminaire

Ce document a été écrit dans le but d'uniformiser l'écriture du code source de SimTaxi. Ceci est nécessaire pour les raisons suivantes :

- Chaque développeur a ses propres règles pour écrire du code. Pour cette raison il est parfois difficile de relire le code de quelqu'un d'autre. Utiliser les mêmes conventions assurera une compréhension plus facile du code.
- Les standards de qualité permettent d'obtenir du code plus robuste qui facilite la détection des bugs.
- Il est plus facile de reprendre un projet où des standards de qualité ont été définis.

6.2 Règles de qualité

Cette section décrit les différents standards de qualité à appliquer au projet SimTaxi.

6.2.1 Noms des fichiers

Les noms de fichiers Python (module) devront respecter le nom de la classe principale qu'ils contiennent afin d'éviter les problèmes de portabilité entre les différents systèmes d'exploitation. Le nom des autres fichiers sont en minuscule ainsi que les extensions.

Rappelons qu'en Python un fichier est un module (contenant des classes, des fonctions, ...) et qu'un dossier est un paquetage.

6.2.2 Tabulations

Les tabulations sont de longueur 4 (utiliser des espaces). La longueur d'une ligne ne doit pas excéder 78 caractères.

6.2.3 Expressions

Chaque opérateur mathématique et l'opérateur d'affectation doivent être précédés et suivis d'un espace. Une virgule (',') est toujours suivie d'un espace.

6.2.4 Noms

Les noms ne doivent pas comporter de souligné ou de caractères spéciaux. La séparation des mots dans un identificateur est indiqué grâce à une majuscule.

Un nom de classe commence toujours par une majuscule. Tous les autres identificateurs commencent par une minuscule.

Exemples :

```
1  # Définition d'une classe
2  class SimTaxiApp :
3
4      # Déclarations de fonctions et de variables
5      def maFonction(unEntier, ...):
6          maVariableEntiere = unEntier
```

6.2.5 En-têtes

Chaque fichier doit comporter l'en-tête suivante :

```
1  #!/usr/bin/env python
2  """
3  Description succincte du module (1 ligne maximum) terminée par un point.
4
5  Description plus complète de ce module :
6  -A quoi sert-il?
7  -Les classes qu'il contient
8  -Principe de fonctionnement
9  -Comment l'utiliser
10
11 $Id: st-qualite.tex,v 1.5 2003/03/16 12:13:04 vega01 Exp $
12
13 """
14
15 __version__ = "$Revision: 1.5 $"
16 __author__ = "EI5a, eivd, SimTaxi (Groupe Burdy)"
17 __date__ = "2002-10-30"
```

Les identifiants mis entre $ seront automatiquement mis à jour par le CVS lors des commit. Il ne faut plus les toucher après la création du fichier.

Les classes devront comporter l'en-tête suivante :

```
"""
Description succincte de la classe (1 ligne maximum) terminée par un point.

Description plus complète de la classe :
-A quoi sert-elle?
-Principe de fonctionnement
-Comment l'utiliser
"""
```

Cette en-tête se place sous la ligne de déclaration de la classe.
Exemple :

```
class MaClasse
    """
    Une classe d'exemple.

    Cette classe ne sert que pour exemple. Elle n'apporte
    aucune fonctionnalité et n'est pas utilisable.
    """
```

Enfin les en-têtes de méthodes se présentent sous cette forme :

```
def methode(self, param):
    """
    Courte description sur une ligne finie par un point.

    Description plus complète si nécessaire.

    nom (type) -- Description du paramètre

    retourne (type) -- Description de l'objet retourné par la fonction
                       (si elle en retourne un)

    - depuis - num version : Depuis quelle version cette méthode existe

    - autheur - Nom
    """
```

Exemple :

```
def methodeExemple(self, param1, param2):
    """
    Cette méthode n'a pas d'action.

    param1 (tuple(int, list)) -- Tuple (nombre de bagages, clients)

    param2 (Taxi) -- le taxi pour ...

    retourne (bool) -- si le taxi peut le faire...

    - depuis - 1.0

    - depuis - Vincent Decorges
    """
```

6.2.6 Description des types

Dans le cas où un type n'est pas connu, il n'est pas utile de le préciser.

Les types les plus courants en Python sont : `tuple, list, int, float, dict, str, ...` pour connaître le type d'un objet il existe toujours une fonction `type`.
Exemple :

```
>>> type("blabla")
<type 'str'>
```

Le type booléen n'existe pas. Il est représenté par des entiers. Mais nous le considèreront quand même comme un type. Rappelons que 0 = faux, tout le reste = vrai.

Tout comme l'exemple de la rubrique en-têtes il est commode de préciser les types contenus dans un tuple.

6.2.7 Gestion des exceptions

Chaque exceptions propre à SimTaxi doivent être dérivées de la classe *ExceptionSimTaxi*. Ceci permet d'effectuer un traitement centralisé des exceptions.
Exemple :

```

    class TestException(ExceptionSimTaxi): pass

    raise TestException(__name__, __version__, "Problème dans le test")
```

6.2.8 Commentaires

Les commentaires peuvent contenir des caractères accentués.

Les commentaires entre """ décrivant nos en-têtes, sont utilisés pour générer la documentation technique. La phase de génération comporte un formatage de texte. Ce qui explique pourquoi il faut mettre une ligne vide entre deux phrases pour passer à la ligne (ce qui peut paraître lourd dans le code). Les détails de ce formatage ne sont pas décris dans ce document, mais vous pouvez déjà visionner (dans la doc technique) le résultat d'un - - en milieu de ligne ou d'un - en début de ligne.

Chapitre 7

Documentation de Rédaction

7.1 Remarque préliminaire

Ce document a été écrit dans le but d'informer sur les techniques d'écriture de la documentation de SimTaxi et de permettre à ceux ne connaissant pas LaTeX d'écrire leur partie de documentation.

La documentation SimTaxi comportera les parties suivantes :

La documentation utilisateur Description des paramètres et manipulation pour l'utilisation de SimTaxi. Utilisation de l'interface utilisateur. Interprétation des résultats de simulation. Définition d'une politique.

La documentation développement Description des diverses parties, des choix de conceptions (ainsi que les schémas), des algorithmes, etc... Cette partie est la plus lourde et sera fractionnée en plusieurs fichiers (voir section 7.3).

La documentation technique La documentation technique sera entièrement générée depuis les commentaires du code source.

Pour faciliter l'écriture de cette documentation nous allons utiliser LaTeX . En deux mots, une documentation LaTeX s'écrit dans un simple fichier texte (extension .tex), qui se «compile» pour générer un document au format ps, pdf, html, etc... Lors de l'écriture d'un document LaTeX on ne se préoccupe ni de la disposition du texte, ni des marges, ni de la césure des mots, ni du format de sortie. On ne fait qu'écrire le texte ainsi que quelques instructions pour le formatage final.

Le fait que les fichiers sources soient au format texte nous permet d'utiliser le CVS et donc de travailler à plusieurs sur la documentation. Comment travailler à 8 sur un «.doc» ?

7.1.1 Environnement requis

Pas besoin d'installer la structure LaTeX sur vos systèmes, elle fait partie des distributions Linux, mais est assez lourde à installer sur du Win32.

Des documents intermédiaires au format pdf seront disponibles régulièrement pour que l'on puisse se rendre compte de l'état du travail.

7.2 Syntaxe minimum

L'écriture d'un document LaTeX est donc très simple, il suffit d'écrire le texte accompagné de quelques instructions/commandes qui permettront au «compilateur» de gérer la structure du document.

La documentation de SimTaxi sera fractionnée en plusieurs fichiers sources .tex, ce qui va alléger un maximum les instructions des fichiers «feuilles» (la structure étant une sorte d'arbre) et minimisera le nombre de personne travaillant sur un même fichier (idéalement nous aurons chacun nos fichiers).

7.2.1 Caractères spéciaux

Un commentaire en LaTeX est précédé du caractère %.

7.2.2 Paragraphes

Voici un (mauvais) exemple de source .tex

```
Les noms de fichiers Python (module) devront respecter le nom de la classe principa
qu'ils        contiennent afin           d'éviter les problèmes
de % ceci est un commentaire
portabilité entre les différents systèmes
 d'exploitation.

Le nom des autres fichiers sont en minuscules
ainsi que les extensions    .
\paragraph{}
Rappelons qu'en Python un fichier est un module (contenant des classes,
des fonctions, ...) et qu'un dossier est un paquetage.
```

Ce qui donnera :

Les noms de fichiers Python (module) devront respecter le nom de la classe principale qu'ils contiennent afin d'éviter les problèmes de portabilité entre les différents systèmes d'exploitation.
Le nom des autres fichiers sont en minuscules ainsi que les extensions.

Rappelons qu'en Python un fichier est un module (contenant des classes, des fonctions, ...) et qu'un dossier est un paquetage.

On peut voir qu'un retour à la ligne se fait avec une ligne vide séparant les deux lignes[1] et qu'un nouveau paragraphe s'introduit grâce à `\paragraph{}`.

7.2.3 Sections

Les sections sont très importantes, ce sont elles qui permettent de structurer le document, de générer la table des matières, etc... Il y a trois niveaux d'imbrication : `\section{sonNom}`, `\subsection{sonNom}`, `\subsubsection{sonNom}`.

Exemple des sections de ce document :

```
\section{Syntaxe minimum}
L'écriture d'un document \LaTeX\ (.tex) est....

   \subsection{Listes}
   Voici un ...
```

Pas besoin de s'amuser avec l'indentation.

[1] comme pour HappyDoc

7.2.4 Listes

Il y a plusieurs façons de faire des listes, voici des exemples :

```
\begin{description}
\item[La documentation utilisateur] Description des paramètres...
\item[La documentation développement] Description des...
\item[La documentation technique] La document...
\end{description}
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
\begin{itemize}
\item blabla
\item Les standards...
\item Il est plus...
\end{itemize}
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
\begin{enumerate}
\item blabla
\item Les standards...
\item Il est plus...
\end{enumerate}
```

La documentation utilisateur Description des paramètres...

La documentation développement Description des...

La documentation technique La document...

- blabla
- Les standards...
- Il est plus...

1. blabla
2. Les standards...
3. Il est plus...

7.2.5 Styles

```
Voici \textit{italique}, \textbf{gras} et \underline{souligné}.
Ou alors le mode $math$ pour écrire des expressions $a_z+bx^2=0$.
```

Voici *italique*, **gras** et souligné. Ou alors le mode $math$ pour écrire des expressions $a_z + bx^2 = 0$.

7.2.6 Notes de bas de page

Les notes de bas de page sont assez agréables à condition de ne pas en abuser.

```
Après ce mot\footnote{ici la note de bas de page} une note de bas de page.
```

Après ce mot[2] une note de bas de page.

[2] ici la note de bas de page

FIG. 7.1 – Description de l'image XY

7.2.7 Stopper le formatage

L'environnement «verbatim» permet de stopper le formatage de texte, c'est lui qui a été utilisé pour présenter les exemples de code dans ce document.

```
begin{verbatim}
   Il manque le \ avant le begin et le end...
end{verbatim}
```

7.2.8 Les étiquettes

Les étiquettes sont très pratiques dans les grands documents. Elles permettent de faire référence à une autre partie du même document sans connaître à l'avance la numérotation qu'elle aura, ni la page où elle se trouvera. Il suffit de placer un «label» à l'emplacement auquel nous voulons faire référence.

```
\label{xy} Comme nous pouvons le voir au point \ref{xy} de la page \pageref{xy}...
```

Comme nous pouvons le voir au point 7.2.8 de la page 43...

7.2.9 Schémas, figures, images

Tous nos graphiques seront dans un bloc «figure» ce qui permet la numérotation du graphique. Les figures ne sont pas spécialement insérées à l'endroit de leur description (LaTeX s'occupe de gérer la répartition pour ne pas alourdir le document), il faut donc y faire référence lorsqu'on en parle.

```
\begin{figure}
% ici viendra le schéma UML de....
\caption{\label{shemaXY}Description de l'image XY}
\end{figure}
... comme nous pouvons le voir sur la figure \ref{shemaXY} à la page \pageref{shema
```

... comme nous pouvons le voir sur la figure 7.1 à la page 43

7.2.10 Math

LaTeX est pratiquement fait pour écrire des maths, mais il y a trop à dire...

```
$$ \sum_{i=0}^{\infty} x_i = 1$$
```

$$\sum_{i=0}^{\infty} x_i = 1$$

7.3 Sectionnement de la documentation

Voici une 1ère proposition de fractionnement. Dans un 1er temps elle ne concerne que les modules. Il y aura aussi toute la partie documentation utilisateur, introduction, etc...

st-gui.tex description de l'affichage ainsi que de l'interface utilisateur.

st-graphe.tex description générale du graphe, de la structure de données.

st-graphe-generation.tex méthode pour la génération du graphe.

st-graphe-PCC.tex méthode du plus court chemin.

st-initialisateur.tex description des techniques d'initialisation des diverses parties.

st-central.tex description du rôle du central, avec qui il communique.

st-central-evenements.tex description de la hiérarchie des événements.

st-central-echeancier.tex description de l'échéancier.

st-politiques.tex description des politiques, sous quelle formes elles sont représentées. Celle que nous avons implémentée.

st-gestionnaires.tex description du rôles des gestionnaires.

st-gest-stations.tex description du gestionnaire des stations mais aussi de ce qu'est une station.

st-gest-taxis.tex description du gestionnaire des taxis ainsi que du comportement d'un taxi, les étapes d'une course.

st-gest-preferences.tex description de la structure des préférences.

7.4 Conclusion

Il y a peu de chance qu'après lecture de ce document vous ayez les connaissances suffisantes pour écrire ce que vous voulez. N'hésitez pas à me demander[3] ce qu'il vous manque.

Sur le CVS vous pouvez trouver le source .tex de notre documentation qualité, qui est un bon exemple (ce document ne l'est pas vraiment).

[3] par email de préférence -> *jb at urbanet.ch*

Chapitre 8

Documentation Utilisateur

8.1 Introduction

Le but de ce guide est de vous permettre de mieux comprendre le programme et vous apprendre à le configurer correctement.

Toutes les étapes d'installation et d'exécution du programme sont expliquées.

Vous y trouverez notamment une description complète de tous les fichiers de configurations qui sont utilisés ainsi que de toutes les fenêtres qui composent l'affichage.

8.2 Pré-requis

L'exécution de SimTaxi requiert les éléments suivant :

- Python $\geq$ 2.2.1
- wxPython $\geq$ 2.3.3
- PyOpenGL $\geq$ 2

Nous n'avons pas fait de tests sur des versions antérieures des librairies.

8.3 Installation

SimTaxi ne requiert pas d'installation particulière.

Il suffit de posséder le dossier des sources que l'ont peut obtenir via CVS à l'aide de la commande suivante :

```
cvs -z3 -d:pserver:anonymous@cvs.sf.net:/cvsroot/simtaxi co dev
```

ou depuis le site http://SimTaxi.sf.net

8.4 Configuration

SimTaxi se configure à l'aide d'un fichier texte ($simul.cfg$).

Ce fichier contient 2 parties, une pour la configuration des statistiques et l'autre pour la configuration de l'interface utilisateur.

Les options contenues dans chacune de ces parties sont décrites ci-dessous.

8.4.1 Statistiques

statActive	($True$ ou $False$) Permet de dire si l'on désire enregistrer les statistiques lors de la simulation.
statGraphique	($True$ ou $False$) Permet de dire si il faut créer les courbes pour les graphiques.
statTest	($True$ ou $False$) Permet d'activer ou non le mode de validation des statistiques.
finDeSimulation	Durée de la simulation en secondes. Cette durée peut être exprimée sous forme d'expression mathématique. Exemple : $60 * 60 * 12$ signifie que la simulation durera 12 heures.

8.4.2 Interface Utilisateur

gui	($True$ ou $False$) Pour utiliser ou non l'affichage graphique. Si la valeur est $False$, seule la console sera affichée.
politique	Permet de spécifier la politique à utiliser dans le cas ou l'affichage graphique n'est pas employé ($gui = False$).
pseudoContinu	($True$ ou $False$) Permet ou non de créer un intervalle de temps entre les évènements. Utilisé seulement si $gui = True$. Cela signifie que l'affichage ne sera rafraîchit seulement quand un évènement sera traité. Le temps entre le traitement des évènements sera constant.
dureeSec	Est utilisé uniquement si $pseudoContinu = True$. Permet de spécifier le temps entre le traitement des évènements.
tailleClient	La taille d'un client (le carré bleu) cf : 8.6.2
texteRemplissageStation	($True$ ou $False$) Permet de dire si l'on veut que le taux de remplissage des stations soit affiché dans le fenêtre OpenGL.
noTaxi	($True$ ou $False$) Permet de dire si l'on veut que les numéros des taxis soient affichés à côté de ceux-ci.
symboleTaxi	($True$ ou $False$) Permet de dire si il faut afficher ou non l'état des taxis. cf : 8.6.3

8.5 Utilisation

8.5.1 Génération des fichiers

Avant de pouvoir lancer le programme avec une ville, il faut générer les fichiers correspondants.

Pour ce faire, il faut se placer dans le dossier dev et lancer la commande suivante :

Attention, il faut au préalable avoir crée le dossier de la ville et y avoir mis un fichier appelé $params.cfg$.

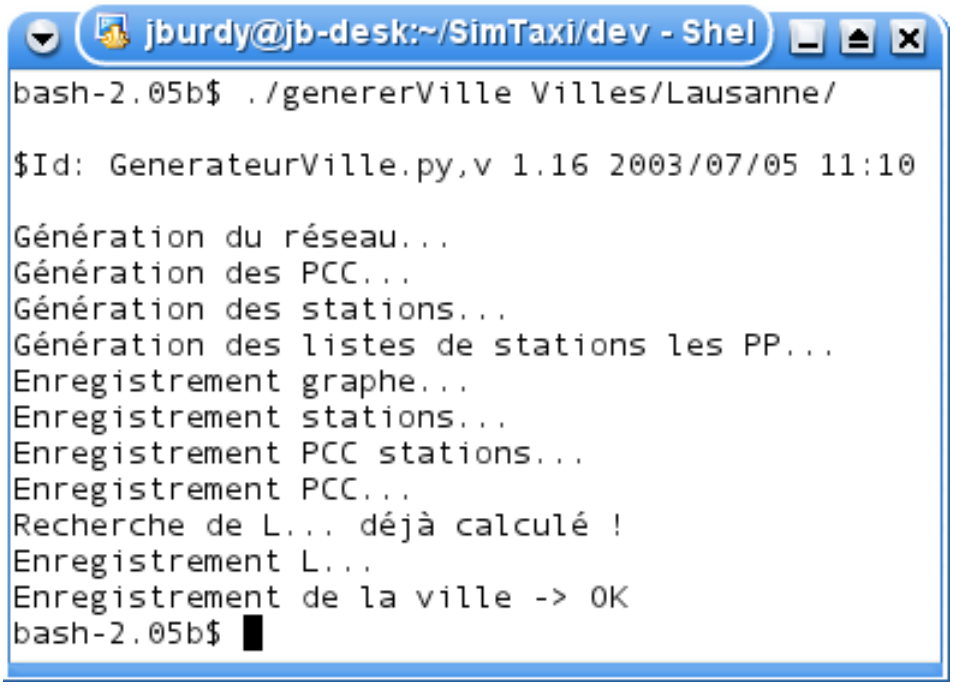
Ce fichier contient des informations pour générer la ville, son contenu est décrit plus loin (cf 8.5.2).

Il faut aussi y mettre un fichier nommé L.

Voici donc la commande qu'il faut exécuter.

```
./genererVille dossierDeLaVille
```

Durant la génération des fichiers, les lignes suivantes devraient s'afficher.



Une fois que tous les fichiers de la ville ont été générés, le programme peut être lancé avec la ville.

8.5.2 Fichiers de configuration pour la génération d'une ville

Les fichiers de configuration, bien qu'ils aient l'extension cfg, ne sont rien d'autre que des fichiers textes.

Pour les éditer, on peut utiliser n'importe quel éditeur de texte.

Fichier $params.cfg$

Le fichier de configuration contient les informations suivante :

Toutes ces informations sont modifiable par l'utilisateur.

nbStation	Le nombre de station qui devront être réparties dans la ville.
tailleStation	Une liste permettant de spécifier la taille des stations. Exemple : si la liste vaut [20,10,10,5], cela signifie qu'il y aura une station de 20 places, 2 de 10 places et le reste des stations aura 5 places.
germeStation	Germe pour le placement des stations. (Pas utilisé actuellement).
nbCarrefour	Le nombre de carrefours que devra comprendre la ville.
nbRue	Le nombre de rues de la ville. Attention, ce nombre de rues est doublé. Une rue à sens unique a un poids de 1 tandis qu'une rue bi-directionnelle a un poids de 2.
diametre	Le diamètre que devra avoir la ville en mètres.
germeReseau	Germe pour la génération de la ville. (Pas utilisé pour le moment).
germePolitiqueDemande	Germe utilisé pour tirer au hasard un taux d'occupation de station. Ce taux d'occupation est utilisé pour amener les taxis dans les stations où il y a la plus forte demande. Ceci n'est utilisé que dans la politique $PolitiqueDemande$.
moyenneCourseKm	Le nombre de kilomètres moyen que devront avoir les courses.
distanceMaxKm	La distance maximum que pourra avoir une course.
nbCoursesJour	Le nombre de courses qui devront être effectuées en 24h.
germeCourse	Germe pour le calcul du temps entre chaque course.
nbTaxis	Le nombre de taxis qui circuleront dans la ville.
germeTaxis	Germe utilisé pour le placement des taxis dans les stations au début de la simulation.
vitesseTaxiKMH	La vitesse à laquelle les taxis rouleront durant la simulation.

Fichier L

La valeur contenue dans ce fichier est utilisée pour l'adaptation de la distance des courses des clients.

La valeur du fichier L doit être recalculée pour chaque nouveau graphe/ville qui est utilisé.
Pour générer cette valeur, il faut qu'un fichier nommé L existe et qu'il contienne la valeur 0.
Pour information, il a fallu 4 heures sur une machine 2 GHz pour calculer cette valeur.
Si une autre valeur que 0 est contenue dans le fichier, c'est cette valeur qui sera utilisée.

8.5.3 Fichiers crées durant la génération

Plusieurs fichiers sont crées durant la génération d'une ville.
Voici leur nom, ce qu'ils contiennent et la taille approximative qu'ils ont.

$graphe$	Contient le graphe représentant la ville	100 ko
PCC	Contient les arbres des plus courts chemin. Pour chaque sommet du graphe, on enregistre les plus courts chemins jusqu'à tous les autres sommets.	10 Mo
$stations$	Contient les informations relatives au positionnement des stations dans le graphe	< 10 ko
$stationsPCC$	Pour chaque station, on mémorise quels sont les chemins les plus courts pour atteindre toutes les autres stations	1.2 Mo

8.5.4 Lancement du programme

Le lancement de SimTaxi se fait à l'aide de $startSimTaxi.py$:

Il faut donner un paramètre au lancement du programme en l'état du dossier où se trouve la ville à simuler.

```
./startSimTaxi.py dossierDeLaVille
```

Un démarrage correct doit afficher les lignes suivantes :

```
Chargement graphe...

Chargement L...

Chargement stations...

Chargement PPC stations...

Chargement PCC...

Chargement de la ville -> OK
```

Par défaut, la simulation ne démarre pas automatiquement.

Pour la démarrer, il faut cliquer sur le bouton $Play$.

8.5.5 Modes d'utilisation

Il existe deux possibilités d'exécution du programme, une avec affichage et l'autre sans affichage.

L'avantage du mode sans affichage est que la simulation se déroule plus rapidement.

Pour passer d'un mode d'exécution à l'autre, il faut faire une modification dans le fichier $simul.cfg$.

La modification consiste à mettre la valeur $False$ au paramètre gui.

Ce paramètre permet en effet de dire si l'on désire utiliser l'affichage ou pas.

Inversement, si vous désirez utiliser le mode d'exécution avec affichage, il faut mettre la valeur $True$ pour le paramètre gui.

8.5.6 Terminaison du programme

La simulation durera un peu plus longtemps que le temps spécifié.

En effet, le moment de fin de la simulation correspond à la dernière demande d'un client pour se faire conduire.

La simulation se terminera donc quand le dernier client sera arrivé à destination.

Une fois que la simulation est terminée, toutes les statistiques qui ont été enregistrée durant celle-ci sont écrites dans un fichier portant le nom de la politique utilisée pour la simulation et ayant l'extension $.sta$.

Exemple : $PolitiquePlusPres.sta$

8.5.7 Durée d'exécution du programme

Pour donner une idée, voici quelques temps d'exécutions qui ont été mesurés sans affichage.

Ces temps ont été mesurés sur une machine 800MHz.

Nombre de jours simulés	Temps d'exécution
0.5	3 min
1	16 min
2	50 min
7	9 h 20 min

8.6 Interprétation de l'affichage

8.6.1 Fenêtre "Simtaxi"

Cette fenêtre est la fenêtre principale du programme.

Elle regroupe toutes les commandes permettant de contrôler la simulation.

Si vous désirez afficher/masquer une des autres fenêtres du programme, il vous est possible de le faire depuis la fenêtre principale.

Pour ce faire, il faut aller dans le menu $Fentre$ et choisir la fenêtre à afficher/masquer.

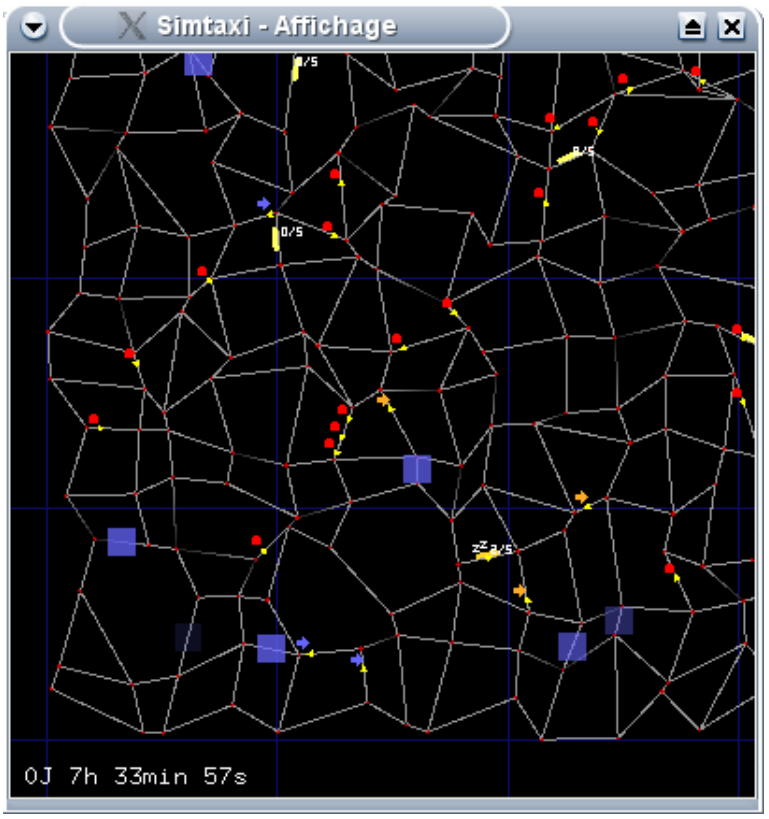
Description des parties de la fenêtre

Contrôle du temps	Permet de démarrer/mettre en pause la simulation. Affiche le temps qui s'est écoulé dans la simulation. La case à cocher $Temporis$ permet de spécifier la fréquence du rafraîchissement. Par défaut, cette case est décochée, cela signifie que le rafraîchissement sera effectué à chaque fois qu'un nouvel évènement est traité. Si vous désirez que l'affichage se fasse en "continu" (rafraîchissement périodique), il faut cocher la case. Dans le cas où l'option d'affichage continu est utilisée, il est possible de spécifier sa vitesse grâce à la barre qui est juste en dessus de la case. Le chiffre affiché en dessus de la barre précise le pourcentage de la vitesse normale. Exemple, si le pourcentage est de 200%, cela signifie que une secondes du temps réèl correspond à 2 secondes dans la simulation.
Affichage	Cette partie permet de choisir les éléments qui seront affichés dans la fenêtre OpenGL.
Politiques	Regroupe toutes les politiques disponibles dans une liste déroulante. A chaque fois que l'on change de politique dans la liste, une description de celle-ci s'affiche en dessous.

8.6.2 Fenêtre "Simtaxi - Affichage"

Permet d'afficher le graphique représentant la ville ainsi que les stations et les taxis qui circulent.

Carrés rouges	Ce sont les carrefours, reliés par des morceaux de route.
Carrés bleus clignotants	Ce sont des clients qui attendent un taxi. Quand ils viennent d'être déposé par un taxi, ils continuent à clignoter mais ils disparaissent progressivement.
Rectangles bruns	Représente une station. Son taux de remplissage est affiché juste à côté.
Triangles jaunes	Ce sont les taxis. Ils sont accompagnés par leur numéro ainsi que d'un symbole.



8.6.3 Signification des symboles près des taxis

Flèche bleue	Le taxi est en train d'aller chercher un client.
Rond rouge	Le taxi transporte un client.
Flèche orange	Le taxi retourne en station.

8.6.4 Fenêtre "Simtaxi - Graphiques"

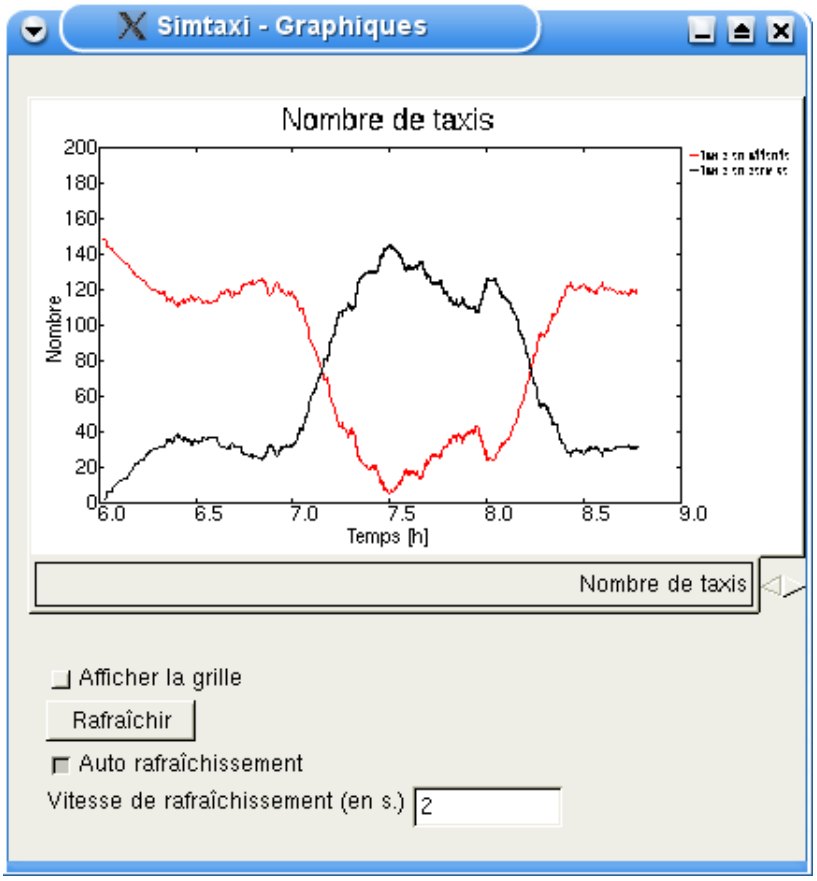
C'est dans cette fenêtre que sont regroupés tous les graphiques en rapport avec les différentes mesures qui sont effectuées.

Ces mesures sont réparties dans des onglets.

Par défaut le rafraîchissement des graphiques est faite toutes les 2 secondes.

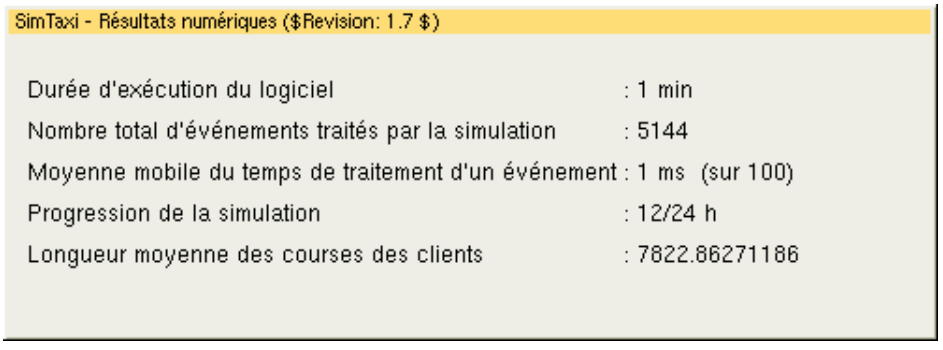
Il est possible de changer ce temps ou de désactiver le mode de rafraîchissement automatique.

Une grille peut également être affichée pour mieux visualiser les courbes.



8.6.5 Fenêtre "Simtaxi - Résultats Numériques"

Ici sont affichés toutes les valeurs numériques qui sont calculées durant la simulation. Ces valeurs sont mises à jour périodiquement.



8.7 Le comparateur de politique

Lancement du programme

Le lancement du programme se fait à l'aide de $comparerPolitiques$: Il faut donner un paramètre au lancement du programme en l'état du dossier où se trouve la ville[1] et aussi le pas (par défaut 1000).

[1] Ne pas oublier le "slash" à la fin.

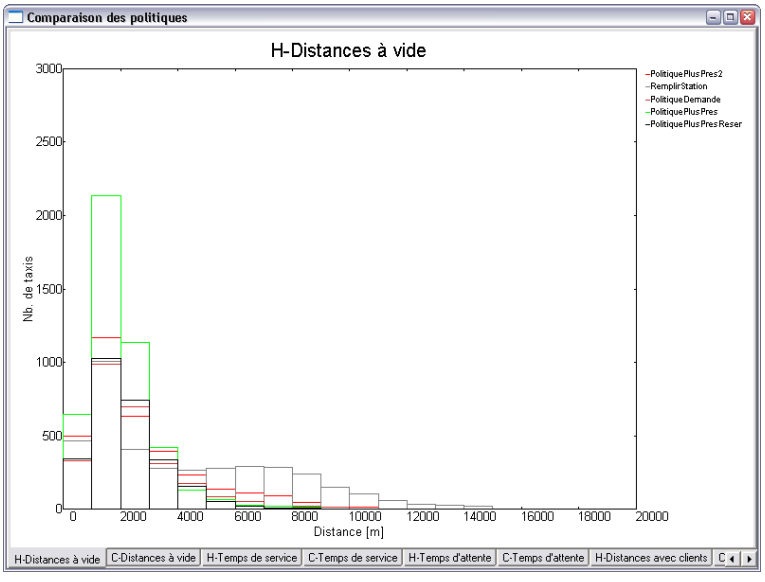
```
./comparerPolitiques dossierDeLaVille/ [lePas]
```

Un démarrage correct doit afficher les lignes suivantes[2] :

```
La politique "PolitiqueDemande" a été trouvée.
La politique "PolitiquePlusPres" a été trouvée.
La politique "PolitiquePlusPres2" a été trouvée.
La politique "PolitiquePlusPresReser" a été trouvée.
La politique "RemplirStation" a été trouvée.
Importation des données...

Distances totales pour "PolitiquePlusPres2":
Avec clients: 21350 [km]
Sans clients: 8887 [km]
Distances totales pour "RemplirStation":
Avec clients: 21702 [km]
Sans clients: 17025 [km]
Distances totales pour "PolitiqueDemande":
Avec clients: 21350 [km]
Sans clients: 6597 [km]
Distances totales pour "PolitiquePlusPres":
Avec clients: 21921 [km]
Sans clients: 9296 [km]
Distances totales pour "PolitiquePlusPresReser":
Avec clients: 21350 [km]
Sans clients: 6017 [km]

Calcul des courbes...
Calculs terminés: affichage des graphiques...
```



Notre programme recherche d'abord les politiques disponibles (fichiers $*.sta$). Ensuite, il importe les données contenues dans les fichiers et affiche les distances totales de chaque politique. Il calcule les différents points des courbes et des histogrammes et pour terminer, il crée la fenêtre et dessine tous les graphiques (voir la figure ci-dessus).

[2] Si toutes les politiques ont été simulées pendant 24 heures.

Affichage d'un graphique

Pour afficher le graphique que l'on désire, il suffit de choisir l'onglet correspondant. La première lettre du titre d'un graphique indique son type : un "H" indique que la représentation se fait par le biais d'histogrammes alors qu'un "C" indique que ce sont des courbes.

Terminaison du programme

Pour terminer le programme, il suffit de fermer la fenêtre comportant les graphiques.

Chapitre 9

Cahier des charges

Projet de groupe en informatique

Simulation de politiques pour les taxis lausannois

Chef du groupe : M. Julien Burdy, EI5a

But : il s'agit de développer un programme de simulation visant à optimiser des politiques d'utilisation de l'information à l'intérieur d'une compagnie de taxis.

Contrainte : l'outil développé doit fonctionner sous Linux.

Description du système :

- Un certain nombre de taxis circulent dans la ville de Lausanne, leur activité étant partiellement coordonnée à partir d'un central. Un certain nombre de stations, dans lesquelles les taxis attendent leur prochaine course, sont réparties dans la ville.
- Lorsqu'un taxi termine une course, il se dirige vers la station la plus proche et transmet cette information au central. Une fois arrivé à la station en question (disons s), le chauffeur peut soit y faire la queue et attendre sa prochaine course (ce qu'il fait si le nombre de taxis déjà présents ne dépasse pas un certain seuil n_s) ou alors (si le seuil est dépassé) se diriger vers la 2e station la plus proche (il en informe alors le central).
- Lorsque le central reçoit un appel d'un client demandant un taxi, il transmet l'ordre de course au taxi le plus proche du client. Il peut soit s'agir du premier taxi de la file d'attente d'une station, soit d'un taxi vide se dirigeant vers une station.
- Le but de la compagnie de taxis est de minimiser le nombre total de kilomètres parcourus.
- Les carrefours sont modélisés par des sommets et les morceaux de rues (entre 2 croisements) sont modélisés par des arêtes. On supposera que les stations de taxis et les points de départ/fin d'une course se trouvent au milieu des arêtes du graphe. Pour tous les trajets effectués, les taxis emprunteront des plus courts chemins.

Données : il est clair que la modélisation réaliste de ce problème est une tâche dépassant largement le cadre de ce projet. En particulier, les fluctuations de la demande et l'encombrement du traffic en fonction du temps (heures creuses, nuit, heures de pointe) seront négligés (pour simplifier, on supposera que l'on se trouve dans une situation de charge moyenne). On veillera néanmoins à ce que certains ordres de grandeurs soient respectés au niveau

- du réseau : la région lausannoise doit ressembler à un graphe comptant environ (au moins serait plus juste...) 800 sommets (carrefours) et 1400 arêtes (morceaux de rues). Le diamètre du graphe (éloignement maximum entre 2 points) est d'environ 12 km en distance et 20 minutes en temps.
- de la demande : globalement on compte environ 2700 courses à effectuer par jour. En moyenne une course représente 8 km (on ne parle ici que du trajet parcouru depuis la prise en charge du client, jusqu'à ce qu'il soit déposé).
- des stations : Lausanne comprend 30 stations et l'on peut supposer que leurs seuils n_s respectifs soient de 20 taxis pour la gare, de 10 pour Chauderon, de 10 pour St-François et de 5 pour toutes les autres.

- ▷ des taxis : 180 taxis circulent en permanence (à vitesse raisonnable...) et effectuent chacun en moyenne 160 km par jour (petit bilan : 2700 courses à 8 km = 21600 km et 180 taxis à 160 km = 28800 km. La différence (7200 km) devrait ressembler au trajet total effectué à vide chaque jour...)

Travail à effectuer :

- ▷ modéliser (compte tenu des limites énoncées ci-dessus) le réseau (sommets, arêtes, temps de trajet, distances), le placement des stations, la demande des clients (depuis où, vers où et quand ?)
- ▷ implémenter un gestionnaire d'événements et les diverses routines permettant de simuler le fonctionnement de la politique actuelle décrite plus haut.
- ▷ concevoir une interface graphique rudimentaire permettant d'observer le déroulement de la simulation.
- ▷ déceler et implémenter des mesures statistiques parlantes sur les performances du système (le nombre moyen de km à vide par exemple...)
- ▷ évaluer l'impact de la transmission de l'information « occupation actuelle de la station » du central vers le taxi, afin que celui-ci évite de faire des km supplémentaires en se dirigeant vers une station s contenant déjà n_s taxis (bien évidemment, les comparaisons entre 2 politiques sont à effectuer sur les mêmes ensembles de demandes aléatoires).
- ▷ proposer et tester l'impact de politiques de conseil plus pertinentes que le central pourrait envisager. En effet, ce dernier est le seul acteur de ce jeu à avoir une vision globale de la répartition des taxis sur le réseau... est-ce un avantage à exploiter ?

Remarque : le travail à effectuer diffère suivant si le groupe décide ou non de poursuivre ce projet durant le deuxième semestre (la description ci-dessus correspond à un projet sur toute l'année). Si le projet ne devait durer qu'un semestre, le groupe s'entendra rapidement avec le professeur responsable pour définir un sous-objectif raisonnable et cohérent.

Bonne chance !

Annexe A

Journal de projet

A.1 1er semestre

Soit deux périodes toutes les semaines.

2002-10-23	Formation du groupe. Choix du projet.
2002-10-30	Reçu énoncé du projet par ETR. Discussion générale.
2002-11-06	Reçu transparents de simulation par ETR. Proposition d'un design. Discussions puis acceptation de celui-ci. Annulation de ce design réorientation (ETR).
2002-11-13	Présentation d'un autre design orienté par ETR. Séparation des modules. Répartition des tâches.
2002-11-20	Discussions plus détaillées sur les interfaces. Conception du central/échéancier
2002-11-27	Présentation et démo du CVS. Présentation d'un prototype d'affichage du réseau. Discussion générale (Design Patterns).
2002-12-02	SimBouffe - Filet Wellington avec ses petits légumes (souper chez JB).
2002-12-04	Discussion avec ETR de la génération des clients et du graphe ; Discussion avec ETR de ce qu'est une politique ; Discussion générale.
2002-12-11	Décision de créer un gestionnaire de préférences.
2002-12-18	Réorganisation complète du système d'extraction de documentation. Utilisation d'HappyDoc.
2002-12-22	Fin des modifs des commentaires, génération de la doc technique. naissance de http ://SimTaxi.sf.net.
2002-12-28	Fin de la nouvelle documentation qualité ; Mise en place de la structure LaTeX qui permettra d'écrire la doc. Problème avec l'algo du chemin le plus court.. trop lent.
2003-01-08	Instruction pour l'écriture de la documentation.
2003-01-15	Intégration des modules, écriture du programme principal.
2003-01-22	La simulation tourne de A à Z.
2003-01-29	Démonstration de la simulation. Discussion des problèmes d'optimisation.
2003-02-05	Sérialisation de la demande. Amélioration de PCC (les arbres calculés sont stockés et sérialisés).
2003-02-12	Préparation de la présentation. Entrecôte de boeuf aux morilles (dîner cafète eivd).
2003-01-19	Présentation (Organisation interne, choix techniques, etc.).
2003-01-26	Relecture et mise au point de la documentation. Préparation de la démonstration.
2003-03-03	Rendu du projet avec sa documentation.
2003-03-05	Démonstration du logiciel.

A.2 2e semestre

Soit quatre périodes toutes les deux semaines.

2003-03-11	Réception du corrigé d'ETR. Mise à jour de la documentation selon certaines corrections. Discussion générale sur la suite du projet.
2003-03-25	Réorganisation du graphe dans un dossier ville. Suppression du module utilisateur. Création d'un générateur de demande client. Création d'une classe ville. Mise en place fixe des stations dans la ville.
2003-04-08	Modification de la mise en place des stations qui sont maintenant générées avec la ville. Amélioration du générateur de ville. Adaptation des comportements des taxis. Modification de l'interface utilisateur pour contrôler la simulation.
2003-04-14	SimBouffe à la brasserie du château à Lausanne.
	Vacances
2003-04-29	Implémentation du gestionnaire de statistiques. Premier graphique visible (à l'aide de R). Création d'un gestionnaire de politique pour le chargement dynamique des politiques.
2003-05-13	Démonstration à ETR des premiers graphiques de cours de simulation provenant des stats. Ajout du paramètre politique dans la configuration pour pouvoir choisir la politique dans les cas de simulation sans interface utilisateur. Création d'une politique basée sur la demande.
2003-05-27	Création d'une fenêtre numérique permettant de voir certaines données du déroulement de la simulation. Organisation pour la documentation. Création d'une politique du plus proche avec réservation.
	Vacances
2003-06-17	Création de l'outil de comparaison des politiques. Création d'une politique visant à ne pas laisser les stations vides. Rédaction de la documentation.
2003-07-01	Finalisation des derniers problèmes. Nettoyage. Rédaction de la documentation.
2003-07-15	Rendu du projet avec sa présentation et démonstration.

Annexe B

Listing des sources

Pour ne pas alourdir le document, uniquement une sélection des fichiers sources jugés important ont été imprimés. Cette sélection c’est faite sur le code des politiques.

B.1 Politique.py

La classe politique mère (classe abstraite).

```
#!/usr/bin/env python
"""
Module contenant la classe abstraite des politiques.

$Id: Politique.py,v 1.1 2003/03/29 11:30:47 vega01 Exp $
"""
__version__ = '$Revision: 1.1 $'
__author__ = 'EI5A, eivd, SimTaxi (Groupe Burdy)'
__date__ = '2002-12-22'



class ErreurAbstraite(Exception):
    """
    Exception pour les classes abstraites.
    """
    pass



class Politique:

    """
    Classe abstraite pour l'implémentation des politiques.
    """

    def info(self):
        """
        Retourne un tuple contenant le nom de la politique et une
        description de celle-ci.

        retourne (Tuple(nom, description))

        - depuis - 1.3

        - auteur - Vincent Decorges
        """
        raise ErreurAbstraite

    def choisirTaxi(self, client):
        """
        Retourne un taxi pour prendre en charge un client
        d'après la politique courante.

        client (EvClient) -- Le client qui veut faire la course

        retourne (Tuple(Taxi, Chemin)) -- Le taxi qui va prendre en charge la course

        - depuis - 1.0

        - auteur - Vincent Decorges
        """
        raise ErreurAbstraite


    def choisirStation(self, taxi):
        """
        Retourne une station d'après la politique courante.

        taxi (Taxi) -- Le taxi qui va à une station

        retourne (Station, Chemin) -- La station

        - depuis - 1.0

        - auteur - Vincent Decorges
        """
        raise ErreurAbstraite
```

B.1.1 PolitiqueDemande.py

```
"""
Implémente la politique de la demande. les taxi rentre dans les stations ou la demande et la plus
forte en faisant une réservation

$Id: PolitiqueDemande.py,v 1.2 2003/07/01 12:25:57 vega01 Exp $
"""
__version__ = '$Revision: 1.2 $'
__author__ = 'EI5a, eivd, SimTaxi (Groupe Burdy)'
__date__ = '2003-03-24'



from Politique import Politique
import GestionnaireTaxis
import GestionnaireStations
from GestionnaireStatistiques import GestionnaireStatistiques
import Ville
import Station
import Ville

class PolitiqueDemande (Politique):
    """
    Le taxi rentre dans la station avec la plus forte demande
    """

    def __init__(self):
        from GestionnairePreferences import GestionnairePreferences
        from random import Random
        self.r = Random(GestionnairePreferences().valeurDe('germePolitiqueDemande'))

    def info(self):
        """
        Retourne un tuple contenant le nom de la politique et une
        description de celle-ci.

        retourne (Tuple(nom, description))

        - depuis - 1.0

        - auteur - Vincent Decorges
        """
        self.__nom = "Demande"
        self.__description = "Choisi le taxi le plus proche de\n" + \
                             "la demande et le renvoie dans\n" + \
                             "la station où il y a la plus forte\n" + \
                             "demande. Le taxi réserve sa\n" + \
                             "place dans la station afin\n" + \
                             "d'éviter qu'elle soit pleine\n" + \
                             "quand il y arrive."

        return (self.__nom, self.__description)

    def choisirTaxi(self, client):
        """
        Retourne un taxi pour prendre en charge un client
        d'après la politique courante.

        client (EvClient) -- Le client qui veut faire la course

        retourne (Tuple(Taxi, Chemin)) -- Le taxi qui va prendre en charge la course

        - depuis - 1.0

        - auteur - Vincent Decorges
        """
        taxi, chemin = GestionnaireTaxis.GestionnaireTaxis().plusProcheDe(client)

        if taxi != None and taxi.estEnDeplacement():
            station = GestionnaireStations.GestionnaireStations().getStation(taxi.
                getNoStation())
```

```
70              station.annulerReservation()
71
72          return (taxi, chemin)
73
74
75      def choisirStation(self, taxi):
76          """
77          Retourne une station d'après la politique courante.
78
79          taxi (Taxi) -- Le taxi qui va à une station
80
81          retourne (Tuple(Station, Chemin)) -- La station et
82
83          - depuis - 1.0
84
85          - auteur - Vincent Decorges
86          """
87
88          stations = GestionnaireStatistiques().demandesStation()
89
90
91          sommeDemande = 0
92          for s in stations:
93              sommeDemande += stations[s]
94
95
96          for s in stations:
97              stations.update({s : (float) (stations[s]) / sommeDemande})
98
99
100
101         prob = self.r.random()
102
103         indice = -1
104         for s in stations:
105             if stations[s] >= prob:
106                 indice = s
107                 break
108         #si aucune station
109         if indice == -1:
110             station, chemin = GestionnaireStations.GestionnaireStations().plusProcheDe(
                    taxi)
111         #on cherche le chemin jusqu'à la sation
112         else:
113             station = GestionnaireStations.GestionnaireStations().getStation(indice)
114             chemin = Ville.Ville().cheminPlusCourt(taxi.arc()[0], station.arc())
115
116         stationTabou = []
117         #on cherche une autre station si elle est pleine autrement on fait une
                réservation
118         while station.reservationPossible() == 0:
119
120             stationTabou.append(station.getNo())
121             station, chemin = GestionnaireStations.GestionnaireStations().
                    plusProcheDeTabous(taxi.arc()[0], stationTabou)
122
123             #si il n'y a plus de station on recommence en vidant la liste des tabous
124             if station == None:
125                 stationTabou = []
126                 station, chemin = GestionnaireStations.GestionnaireStations().
                        plusProcheDeTabous(taxi.arc()[0], stationTabou)
127
128
129         station.reserverPlace()
130         return (station, chemin)
```

B.1.2 PolitiquePlusPres.py

```
#!/usr/bin/env python
"""
Module contenant la politique du plus près.

$Id: PolitiquePlusPres.py,v 1.4 2003/07/11 08:44:38 vega01 Exp $
"""
__version__ = '$Revision: 1.4 $'
__author__ = 'EI5a, eivd, SimTaxi (Groupe Burdy)'
__date__ = '2002-12-22'



from Politique import Politique
import GestionnaireTaxis
import GestionnaireStations

class PolitiquePlusPres (Politique):
    """
    Implémante la politique du taxi le plus près et de
    la station la plus proche.
    """

    def info(self):
        """
        Retourne un tuple contenant le nom de la politique et une
        description de celle-ci.

        retourne (Tuple(nom, description))

        - depuis - 1.5

        - auteur - Vincent Decorges
        """
        self.__nom = "Plus proche"
        self.__description = "Choisi le taxi le plus proche\n" + \
                             "de la demande et le renvoie\n" + \
                             "dans la station la plus proche de\n" + \
                             "la destination."


        return (self.__nom, self.__description)

    def choisirTaxi(self, client):
        """
        Retourne un taxi pour prendre en charge un client
        d'après la politique courante.

        client (EvClient) -- Le client qui veut faire la course

        retourne (Tuple(Taxi, Chemin)) -- Le taxi qui va prendre en charge la course

        - depuis - 1.0

        - auteur - Vincent Decorges
        """
        return GestionnaireTaxis.GestionnaireTaxis().plusProcheDeEnStation(client)


    def choisirStation(self, taxi):
        """
        Retourne une station d'après la politique courante.

        taxi (Taxi) -- Le taxi qui va à une station

        retourne (Tuple(Station, Chemin)) -- La station et

        - depuis - 1.0

        - auteur - Vincent Decorges
        """
        return GestionnaireStations.GestionnaireStations().plusProcheDe(taxi)
```

B.1.3 PolitiquePlusPres2.py

```
"""
Module contenant la politique du plus près modifié pour
que si une station se trouvant à moins de 3 km est remplie
à moins de 20% alors le taxi ira dans cette station plutôt
que dans celle la plus proche

$Id: PolitiquePlusPres2.py,v 1.5 2003/07/14 16:52:49 vega01 Exp $
"""
__version__ = '$Revision: 1.5 $'
__author__ = 'EI5a, eivd, SimTaxi (Groupe Burdy)'
__date__ = '2003-03-24'



from Politique import Politique
import GestionnaireTaxis
import GestionnaireStations
import Ville

class PolitiquePlusPres2 (Politique):
    """
    Implémante la politique du taxi le plus près et de
    la station la plus proche.
    """

    def info(self):
        """
        Retourne un tuple contenant le nom de la politique et une
        description de celle-ci.

        retourne (Tuple(nom, description))

        - depuis - 1.0

        - auteur - Vincent Decorges
        """
        self.__nom = "Plus proche 2"
        self.__description = "Choisi le taxi le plus proche de \n" + \
                             "la demande et le renvoie dans\n" + \
                             "la station la plus proche de la\n" + \
                             "destination, si une station avec\n" + \
                             "un taux de remplissage inférieur à\n" + \
                             "20% ne se trouve pas à moins de\n" + \
                             "3 km de la destination."


        return (self.__nom, self.__description)

    def choisirTaxi(self, client):
        """
        Retourne un taxi pour prendre en charge un client
        d'après la politique courante.

        client (EvClient) -- Le client qui veut faire la course

        retourne (Tuple(Taxi, Chemin)) -- Le taxi qui va prendre en charge la course

        - depuis - 1.0

        - auteur - Vincent Decorges
        """
        return GestionnaireTaxis.GestionnaireTaxis().plusProcheDe(client)


    def choisirStation(self, taxi):
        """
        Retourne une station d'après la politique courante.

        taxi (Taxi) -- Le taxi qui va à une station

        retourne (Tuple(Station, Chemin)) -- La station et
```

```
72
73          - depuis - 1.0
74
75          - auteur - Vincent Decorges
76          """
77
78          station, chemin = GestionnaireStations.GestionnaireStations().
                plusProcheDeOccupation(taxi, 20.0)
79
80          if station == None:
81              return GestionnaireStations.GestionnaireStations().plusProcheDe(taxi)
82          else:
83              distance = Ville.Ville().cheminPlusCourt(station.arc(), taxi.arc()[0])
84              if distance > 3000:
85                  return GestionnaireStations.GestionnaireStations().plusProcheDe(taxi)
86              else:
87                  return (station, chemin)
```

B.1.4 PolitiquePlusPresReser.py

```
"""
Implémente la politique du plus près. Mais avec réservation des
stations. De cette manière le taxi est sûr d'avoir une place quand il rentre
dans la station

$Id: PolitiquePlusPresReser.py,v 1.7 2003/05/13 16:55:27 vega01 Exp $
"""
__version__ = '$Revision: 1.7 $'
__author__ = 'El5a, eivd, SimTaxi (Groupe Burdy)'
__date__ = '2003-03-24'



from Politique import Politique
import GestionnaireTaxis
import GestionnaireStations
import Station
import Ville

class PolitiquePlusPresReser (Politique):
    """
    Implémante la politique du taxi le plus près et de
    la station la plus proche avec réservation.
    """

    def info(self):
        """
        Retourne un tuple contenant le nom de la politique et une
        description de celle-ci.

        retourne (Tuple(nom, description))

        - depuis - 1.0

        - auteur - Vincent Decorges
        """
        self.__nom = "Plus proche avec réservation"
        self.__description = "Choisi le taxi le plus proche de\n" + \
                             "la demande et le renvoie dans\n" + \
                             "la station la plus proche de la\n" + \
                             "destination. Le taxi réserve sa\n" + \
                             "place dans la station afin\n" + \
                             "d'éviter qu'elle soit pleine\n" + \
                             "quand il y arrive."

        return (self.__nom, self.__description)

    def choisirTaxi(self, client):
        """
        Retourne un taxi pour prendre en charge un client
        d'après la politique courante.

        client (EvClient) -- Le client qui veut faire la course

        retourne (Tuple(Taxi, Chemin)) -- Le taxi qui va prendre en charge la course

        - depuis - 1.0

        - auteur - Vincent Decorges
        """
        taxi, chemin = GestionnaireTaxis.GestionnaireTaxis().plusProcheDe(client)

        if taxi != None and taxi.estEnDeplacement():
            station = GestionnaireStations.GestionnaireStations().getStation(taxi.
                getNoStation())
            station.annulerReservation()

        return (taxi, chemin)


    def choisirStation(self, taxi):
```

```
71          """
72          Retourne une station d'après la politique courante.
73
74          taxi (Taxi) -- Le taxi qui va à une station
75
76          retourne (Tuple(Station, Chemin)) -- La station et
77
78          - depuis - 1.0
79
80          - auteur - Vincent Decorges
81          """
82
83          station, chemin = GestionnaireStations.GestionnaireStations().plusProcheDe(taxi)
84          stationTabou = []
85
86          #on cherche une autre station si elle est pleine autrement on fait une
                réservation
87          while station.reservationPossible() == 0:
88
89              stationTabou.append(station.getNo())
90              station, chemin = GestionnaireStations.GestionnaireStations().
                    plusProcheDeTabous(taxi.arc()[0], stationTabou)
91
92              #si il n'y a plus de station on recommence en vidant la liste des tabous
93              if station == None:
94                  stationTabou = []
95                  station, chemin = GestionnaireStations.GestionnaireStations().
                        plusProcheDeTabous(taxi.arc()[0], stationTabou)
96
97
98          station.reserverPlace()
99          return (station, chemin)
```

B.1.5 RemplirStation.py

```
#!/usr/bin/env python
"""
Evite que les stations soient remplie à moins de 15 pourcent.

$Id: RemplirStation.py,v 1.5 2003/07/11 08:44:38 vega01 Exp $
"""
__version__ = '$Revision: 1.5 $'
__author__ = 'EI5a, eivd, SimTaxi (Groupe Burdy)'
__date__ = '2003-03-25'



from Politique import Politique
import GestionnaireTaxis
import GestionnaireStations

class RemplirStation (Politique):
    """
    Evite que les stations soient remplie à moins de 15 pourcent et
    prends les taxis dans les stations remplie à plus de 60%. Si aucune
    stations répond à ces critère politique du plus proche
    """

    def info(self):
        """
        Retourne un tuple contenant le nom de la politique et une
        description de celle-ci.

        retourne (Tuple(nom, description))

        - depuis - 1.5

        - auteur - Vincent Decorges
        """
        self.__nom = "Remplir Station"
        self.__description = "Va chercher les taxis dans les\n" + \
                             "stations qui sont remplies à\n" + \
                             "plus de 60% et les places dans\n" + \
                             "des stations à moins de 15.%"


        return (self.__nom, self.__description)

    def choisirTaxi(self, client):
        """
        Retourne un taxi pour prendre en charge un client
        d'après la politique courante.

        client (EvClient) -- Le client qui veut faire la course

        retourne (Tuple(Taxi, Chemin)) -- Le taxi qui va prendre en charge la course

        - depuis - 1.0

        - auteur - Vincent Decorges
        """
        taxi, chemin = \
        GestionnaireTaxis.GestionnaireTaxis().getTaxiOccupation(client, 60.0, '>')

        if taxi == None:
            return GestionnaireTaxis.GestionnaireTaxis().plusProcheDe(client)
        else:
            return (taxi,chemin)

    def choisirStation(self, taxi):
        """
        Retourne une station d'après la politique courante.

        taxi (Taxi) -- Le taxi qui va à une station

        retourne (Tuple(Station, Chemin)) -- La station et
```

```
72
73          - depuis - 1.0
74
75          - auteur - Vincent Decorges
76          """
77
78          station, chemin =\
79          GestionnaireStations.GestionnaireStations().getStationOccupation(taxi, 15.0, '<')
80
81          if station == None:
82              return GestionnaireStations.GestionnaireStations().plusProcheDe(taxi)
83          else:
84              return (station, chemin)
```