Contents

1 Résumé

Ce document présente la solution apportée par notre groupe au projet que nous à proposé CIL4SYS : utiliser l'apprentissage statistique dans le but de fluidifier le trafic en zone urbaine. Nous proposons à ce stade un modèle DQN ayant pour but de contrôler les feux tricolores sur un morceau de quartier d'Issy les Moulineaux.

2 Introduction

Alors que l'écologie devient un enjeu international d'importance, de plus en plus de recherches sont faites sur des transports verts, ou sur des moyens d'économiser de l'énergie. Dans le même temps, les avancées technologiques laissent présager une arrivée de véhicules autonomes capables de communiquer avec leur environnement, ou du moins de l'intelligence artificielle comme assistance de conduite. Dans ce contexte, plusieurs études ont été metnées dans le but de réduire les émissions de polluants des véhicules par l'intermédiaire de la fluidification du trafic routier. Parmi ce domaine de recherche, on distingue notamment les travaux se concentrant sur les zones extra-urbaines (autoroutes et grands axes routiers) [citer papiers sans commenter], des zones urbaines qui sont au c{œ}ur du sujet traité dans ce rapport. La start-up CIL4SYS nous propose, dans le cadre d'un projet fil-rouge, de nous pencher sur la possibilité de réduire les émissions de polluants dans l'hypothèse de feux tricolores intelligents capables d'envoyer des consignes de vitesse aux véhicules traversants un quartier en zone urbaine dans le but de limiter les accélérations et décélérations des véhicules sources de fortes consommations en carburant.

Le projet s'articule donc autour de CIL4SYS créée en juin 2015 par Philippe GICQUEL qui a travaillé pendant 25 ans en R\D automobile chez EDAG et PSA Peugeot-Citroën. Le c{œ}ur de métier de CIL4SYS étant de proposer une solution permettant de générer des cahiers des charges et des spécifications techniques à partir de diagrammes UML décrivant le fonctionnement désiré d'un système. Leur cible principale étant l'industrie automobile, un environnement de simulation a été spécifiquement développé dans le but de répondre à cette industrie. Mais aussi de SOFTEAM, grand groupe fournissant un panel de service divers, allant du consulting opérationnel et métier, à la modélisation et l'automatisation, en passant par l'innovation. Leur rôle, durant ce projet, sera de nous assister lors de la phase de machine-learning en nous apportant une expertise avancée en data science. Enfin TélécomParisTech constitue le dernier acteur de ce projet par l'intermédiaire de notre groupe d'étudiants chargé d'effectuer les premiers développements et essais ainsi que de proposer les solutions et méthodes à explorer.\

L'objectif principal de ce projet est donc de démontrer à l'échelle d'un quartier que par un contrôle à la fois des feux tricolores et des vitesses des véhicules (dans un contexte où les véhicules sont communicants et peuvent recevoir des instructions de vitesse), il est possible de trouver un optimum de diminution des rejets de polluants et de CO2. Les méthodes mises en place au cours du projet devront faire appel à nos connaissances en apprentissage automatique afin de proposer une première approche de résolution.

Il nous est demandé de d'utiliser nos connaissances pour proposer une solution de machine learning à ce problème.

3 Formulation d'optimisation

Plus classiquement, il est possible de donner une formulation d'un point de vue optimisation du problème. En effet, l'objectif principal demandé par CIL4SYS est de trouver des lois de commande des feux et des véhicules de manière à réduire au maximum les émissions de CO2. Au cours de ce projets les émissions de CO2 seront modélisées uniquement par l'intermédiaire des accélérations. Une première approche peut donc être de minimiser

$$\min_{\theta(t)} \mathbb{E}\left[\sum_{j=1}^{m} \int_{t=0}^{t_j^{sortie}} |a_j(t, \theta(t), \mathcal{C}_j(\xi))| dt\right], \tag{1}$$

où $a_j(t)$ représente les accélérations subies par le véhicule j tout au long de sa trajectoire $\mathcal{C}_j$ affectée aléatoirement par la variable aléatoire ξ et $\theta(t)$ représente la loi de commande imposée aux véhicules.

On constate d'ors et déjà que la solution à un tel problème d'optimisation est évidente puisqu'il suffit d'imposer une loi de commande de telle sorte que les véhicules restent à l'arrêts pour minimiser les emissions. Le problème initial d'optimisation se doit donc d'être complété par l'intermédiaire de

contraintes afin répondre à un certain réalisme. Un première contrainte à envisager est le temps de trajet pour la réalisation du parcours $\mathcal{C}$ du véhicule. En utilisant les données TOMTOM, il sera donc possible de déterminer au sein du quartier qui sera choisi par la suite quel est le temps de trajet moyen réalisé par un véhicule pour une trajectoire imposée $\bar{t}(\mathcal{C})$. Une contrainte seuil sur le temps de trajet pourra alors être imposé avec pour référence le temps de trajet calculé. Le problème de minimisation se reformule alors de la manière suivante

$$\begin{cases} \min\limits_{\theta(t)} \mathbb{E}\left[\sum\limits_{j=1}^{m} \int_{t=0}^{t_j^{sortie}} |a_j(t, \theta(t), \mathcal{C}(\xi))| dt\right] \\ \text{s.t. } t_j^{sortie} \leq \alpha \cdot \bar{t}(\mathcal{C}) \end{cases} \tag{2}$$

La formalisation du problème d'optimisation reste à compléter et on sera amené à ajouter d'autres contraintes au cours du projet au fur et à mesure de sa prise en main. La fonction de coût à minimiser sera elle aussi amenée à être changée si par exemple le nombre de voitures qui traversent le quartier devient aléatoire lui aussi etc...

4 Apprentissage par renforcement

Comme dit précédemment, une des méthodes utilisée dans la littérature actuelle pour effectuer de la fluidification de trafic est l'apprentissage par renforcement. C'est la méthode d'apprentissage sur laquelle nous avons décidée de nous concentrer car novatrice dans le domaine de l'optimisation de trafic routier. Dans cette section, nous présentons les grandes ligne de cette méthode d'apprentissage.

L'apprentissage par renforcement est une méthode statistique de prise de décision dans un environnement donné. L'environnement est modélisé par un état, et l'acteur peut réaliser une action qui affectera l'état. L'algorithme est guidé lors de la phase d'entraînement par une fonction de récompense. Après avoir pris une action a à un état e, l'observation du nouvel état permet le calcul d'une récompense, $r \in \mathbb{R}$.

Notons $\mathcal{E}$ l'ensemble des états possibles de notre environnement, $\mathcal{A}$ l'ensemble des actions possibles, et r_t la récompense obtenue au pas de temps t. En modélisant la récompense cumulative comme un processus de Markov, elle ne dépend que de l'action prise et de l'état du système aux temps $t \geq t_0$ et est définie de la manière suivante:

$$R_{t_0} = \sum_{t=t_0}^{\infty} \gamma^{t-t_0} r_{t+1}$$

Où le coefficient $\gamma \in [0, 1]$ permet de donner plus de poids aux récompenses proches de t_0 dans le temps et à la somme de converger.

Le principe de l'apprentissage par renforcement est alors de chercher une fonction $Q^* : \mathcal{E} \times \mathcal{A} \longrightarrow \mathbb{R}$ qui estime le retour cumulatif R_{t_0} pour une action $a \in \mathcal{A}$ réalisée à un état $e \in \mathcal{E}$.

La fonction $\pi(e, a)$ retourne la probabilité de réaliser une action a à l'état e. Nous avons donc $\sum_{a \in \mathcal{A}} \pi(e, a) = 1$. Définissons $Q^\pi(e, a)$, la fonction qui prédit l'espérance de la récompense cumulative sous π sachant e et a:

$$Q^\pi(e, a) = \mathbb{E}_\pi[R_t | e_t = e, a_t = a] \tag{3}$$

Définissions $\mathcal{P}^a_{ee'}$, la probabilité de passer d'un état e à un état e' sachant un état e et une action a donnés :

$$\mathcal{P}^a_{ee'} = \mathbb{P}(e_{t+1} = e' | e_t = e, a_t = a)$$

Définissions aussi $\mathcal{R}^a_{ee'}$, l'espérance de la récompense r_{t+1} sachant un état e, et une action a à l'instant t ainsi qu'un état e' à l'instant $t+1$:

$$\mathcal{R}^a_{ee'} = \mathbb{E}(r_{t+1}|e_t = e, e_{t+1} = e', a_t = a)$$

Il est alors possible de d'exprimer Q^π de la manière suivante:

$$\begin{aligned}
Q^\pi(e,a) &= \mathbb{E}_\pi[R_t|e_t = e, a_t = a] && (4)\\
&= \mathbb{E}_\pi[\sum_{t=t_0}^{\infty} \gamma^{t-t_0} r_t|e_{t_0} = e, a_{t_0} = a] && (5)\\
&= \mathbb{E}_\pi[r_{t+1} + \gamma \sum_{t=t_0}^{\infty} \gamma^{t-t_0} r_{t_0+1}|e_{t_0} = e, a_{t_0} = a] && (6)\\
&= \sum_{e'} \mathcal{P}^a_{ee'} \left[\mathcal{R}^a_{ee'} + \gamma \mathbb{E}_\pi \left(\sum_{t=t_0}^{\infty} \gamma^{t-t_0} r_{t_0+1}|e_{t+1} = e'\right)\right] && (7)\\
&= \sum_{e'} \mathcal{P}^a_{ee'} \left[\mathcal{R}^a_{ee'} + \gamma \sum_{a'} \mathbb{E}_\pi \left(\sum_{t=t_0}^{\infty} \gamma^{t-t_0} r_{t_0+1}|e_{t+1} = e', a_{t+1} = a'\right)\right] && (8)\\
&= \sum_{e'} \mathcal{P}^a_{ee'} \left[\mathcal{R}^a_{ee'} + \gamma \sum_{a'} \pi(e', a') Q^\pi(e', a')\right] && (9)
\end{aligned}$$

Si la politique d'action à choisir dans un état donné consiste à maximiser la récompense cumulative, alors:

$$\pi^*(e) = \underset{a}{\text{argmax}} Q^*(e,a)$$

Cependant, nous ne connaissons pas la fonction Q^*, nous utilisons donc un modèle statistique pour l'approcher. En prenant alors:

$$\pi(e) = \underset{a}{\text{argmax}} Q^\pi(e,a)$$

L'équation de $Q^\pi(e,a)$ sous cette politique se simplifie alors:

$$Q^\pi(e,a) = r + \gamma Q^\pi(e', \pi(e'))$$

4.0.1 Exploration vs exploitation

Au début de l'apprentissage, la fonction approchée par notre modèle ne sera pas de bonne qualité, et donc notre politique de prendre l'action avec la plus grande valeur de Q^π_θ peut potentiellement ne pas converger. Pour encourager l'algorithme à explorer son espace d'action, on introduit un paramètre $\epsilon \in [0,1]$. On modifie alors notre politique d'action de manière à faire une action aléatoire dans $\epsilon\%$ des cas:

$$\pi(e) = \begin{cases} \text{random}(\mathcal{A}), & \text{si random}([0,1]) < \epsilon. \\ \underset{a}{\text{argmax}} Q^\pi(e,a), & \text{sinon.} \end{cases}$$

Il est aussi possible de faire varier ϵ dans le temps, avec par exemple des valeurs plus grosses en début d'entraînement pour ensuite diminuer. Le Listing 1 propose une implémentation possible de stratégie ϵ – greedy en Python.

```
def act(self, state):
    if np.random.rand() < self.epsilon:
        return random.randrange(self.action_size)
    else:
        probas = self.model.predict(state)
        return np.argmax(probas)
```

Listing 1: Stratégie ϵ – greedy en Python.

4.0.2 Deep Q-Network

Q^π étant inconnue, l'idée consiste à l'approcher avec un réseau de neurones dense multi-couches, ce qui revient alors a choisir un paramètre θ de manière a minimiser δ, l'erreur de différence temporelle de notre modèle approché $Q^\pi_\theta(e, a)$:

$$\theta \in \underset{\theta}{\operatorname{argmin}} \mathcal{L}(\delta)$$

$$\delta = Q^\pi_\theta(e, a) - (r + \gamma \underset{a'}{\operatorname{argmax}} Q^\pi_\theta(e', a')) \tag{10}$$

Ou $\mathcal{L}$ est une fonction de perte. En pratique, l'optimisation est réalisée par "batchs" de transitions B a l'aide d'une descente de gradient stochastique~[?].

En effet, après chaque action prise, le calcul de la récompense obtenue est réalisé et ces transitions sont stockées dans une mémoire $\mathcal{M}$ composée de quadruplets $(a, e, e', r) \in \mathcal{A} \times \mathcal{E}^2 \times \mathbb{R}$. A chaque action, nous mettons en mémoire le quaduplet obtenu et réalisons une descente de gradient sur un batch $B \in \mathcal{B}$.

Les paramètres de notre modélisation sont donc nombreux:

- θ, les paramètres de notre modèle statistique
- γ, le poids données aux récompenses plus tôt dans le temps
- $\mathcal{L}$, la fonction de perte
- Le fonction de calcul de récompense, $\psi : \mathcal{E} \longrightarrow \mathbb{R}$

Le réseau de neurones d'un DQN prend donc en entrée les états accumulés dans la mémoire $\mathcal{M}$ au cours de l'exploration. C'est une fonction à $|\mathcal{A}|$ neurones en sortie où $\mathcal{A}$ est l'espace des actions. L'activation de cette dernière couche est **linéaire**, en effet, l'espérance de la récompense pour une action et un état donnés prend une valeur sur $\mathbb{R}$. Enfin, la fonction à minimiser est celle de la différence temporelle (équation 10). Une implémentation utilisant la librairie *Keras* est proposée dans le Listing 2.

Le réseau est entraîné à chaque pas de temps sur un batch *aléatoire*, pioché dans la mémoire $\mathcal{M}$. En effet, cette mémoire contient toutes les paires d'états à une action d'intervalle explorées, il est donc possible d'estimer l'équation 3 tout les pas de temps sur ces paires avec le modèle. Le Listing 3 propose une implémentation en Python d'un algorithme d'entraînement du DQN.

```python
def build_DQN(self):
    model = Sequential([
        Dense(24, input_shape=(self.state_size,)),
        Activation('relu'),
        Dense(24),
        Activation('relu'),
        Dense(self.action_size),
        Activation('linear'),
    ])
    model.compile(loss=self.temporal_difference, optimizer=Adam(lr=self.learning_rate))

    return model
```

Listing 2: Implémentation d'un réseau DQN dense bi-couche avec *Keras*.

```python
def replay(self, batch_size):
    minibatch = random.sample(self.memory, batch_size)

    for state, action, reward, next_state, done in minibatch:
        q_values = self.model.predict(state)
        if done:
            q_values[0][action] = reward
        else:
            q_values[0][action] = (reward + self.gamma *
                                   np.amax(self.q_values_model.predict(next_state)[0]))
        self.model.fit(state, q_values, epochs=1, verbose=0)

    if self.epsilon > self.epsilon_min:
        self.epsilon *= self.epsilon_decay
```

Listing 3: Entraînement du DQN en Python.

5 Présentation des outils utilisés

L'apprentissage par renforcement reposant sur l'interaction avec un environnement, l'entraînement s'appuie souvent sur un simulateur. C'est le cas de notre modèle, ce qui implique donc l'interaction de plusieurs unités logiques du code entre elles. Afin de simplifier au maximum ces interactions tout en rendant le passage à l'échelle de l'entraînement possible, notre modèle repose sur un certain nombre d'outils qu'il est nécessaire d'introduire à ce stade du document, c'est ce que nous faisons à présent.

5.1 Sumo & Traci

Sumo est un simulateur de trafic microscopique multi-plateforme, libre, développé et maintenu par le ministère des transports allemand et possédant une API Python nommée *Traci* de haute qualité. Ce simulateur permet entre autre de requêter et modifier l'état d'une simulation à chaque pas de temps via Traci. C'est ce qui permet de prendre des actions et de calculer les récompenses lors de l'entraînement de notre agent.

Le module Traci de Sumo permet entre autre de requêter les émissions de chaque voiture, l'état de chaque feu, les collisions éventuelles, ainsi que la redirection, suppression et instanciation de véhicules à chaque pas de temps.

Dans Sumo, les routes sont représentées par des *arêtes* (*edges* en anglais), chaque arête pouvant avoir une ou plusieurs *voies* (*lanes* en anglais). Chaque arête et chaque voie est identifiée de manière unique par une chaîne de caractères, ce qui permet d'interagir facilement avec les véhicules de la simulation.

Les *feux* sont placés à certaines intersections sur la carte. Mais dans Sumo, le nombre de feux peut être supérieur au nombre d'intersections. L'état des feux à une intersection est défini par une chaîne de caractères où chaque caractère représente l'état d'un des feux de l'intersection. Par exemple, l'état des feux d'une intersection gérée par trois feux pourrait être `rGG` ce qui voudrait dire que le premier feu est *rouge* (red) et les deux suivants *verts* (green). Le premier caractère de la chaîne représente l'état du feu le plus au nord de l'intersection gérée, le suivant le feu directement après en allant dans le sens anti-horaire, et ainsi de suite jusqu'à la fin de la chaîne. Enfin, au même titre que les arêtes, Sumo identifie chaque jeux de feux gérant une intersection par une chaîne de caractère unique permettant de requêter le feu via Traci.

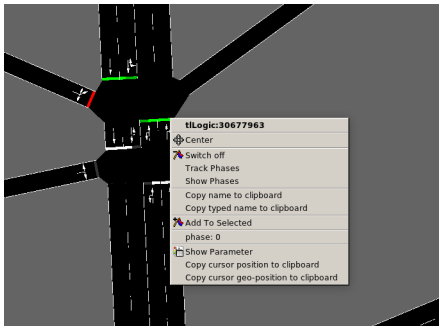
5.2 RLlib

RLlib est une suite d'outils pour Python facilitant la mise en place d'une infrastructure d'apprentissage par renforcement. D'un part il fournit les algorithmes classiques du domaine, dont le DQN, *à la Scikit-learn*, et d'autre part il permet l'entraînement distribué d'agent. En effet, le passage à l'échelle de notre modèle implique un entraînement distribué, et donc la capacité à instancier plusieurs processus Sumo et interagir avec, ce que RLlib permet de réaliser.

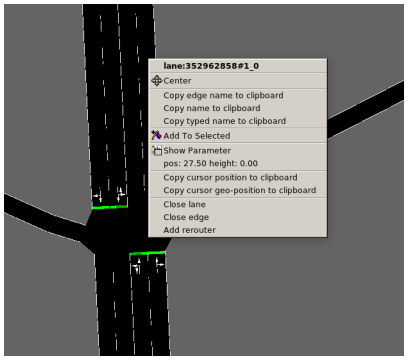
5.3 Flow

L'intégration de Sumo est RLlib étant complexe à mettre en œuvre, un laboratoire de UC Berkeley travaille sur une abstraction open source de RLlib et Sumo de manière à faciliter l'application de l'apprentissage par renforcement au problème de fluidification de trafic routier. C'est donc à l'aide cette librairie Python appelée Flow que nous avons implémenté notre solution. Dans l'optique de permettre au lecteur de bien comprendre les prochaines sections du présent document, nous introduisons les concepts fondamentaux de Flow ainsi que son jargon.

Un **scénario** est un objet qui définit la simulation en elle même, c'est à dire le définition des arêtes et des voies, le nombre de voitures et les rails qu'elles suivent, ainsi que la position des feux. Flow fournit la classe abstraite `BaseScenario` à sous-classer lors de l'implémentation d'un scénario de calcul. Cette classe



(a) Feux 30677963 à l'état GrG en partant du haut et en allant dans le sens anti-horaire.



(b) File 1 de l'arrête 3529628581 .

Figure 1: Nomenclature utilisée par Sumo.

propose notamment des méthodes pour importer des maillages routiers de différents formats, dont celui de *Open Street Map*.

Un **environnement** est un objet qui sous-classe `BaseEnv` et qui définit l'espace des états, $\mathcal{E}$, celui des actions $\mathcal{A}$, les transitions d'états par une action, ainsi que la récompense via les méthodes `observation_space`, `action_space`, `_apply_rl_action` et `compute_reward`. De cette manière, il est facile de définir plusieurs modélisations du système pour un scénario donné.

Une **expérience** définit les paramètres de l'algorithme utilisé pour maximiser l'espérance de la récompense (DQN, PPO, etc. . .) ainsi que d'autres paramètres de simulation tels que le nombre de processus Sumo à instancier, et le nombre d'itérations à réaliser durant l'entraînement.

Enfin, un **kernel** est un objet qui fait abstraction de l'API du simulateur utilisé pour l'entraînement, cela permet d'utiliser l'abstraction de Flow pour interagir avec Sumo plutôt que *Traci* et donc de découpler notre code du simulateur. Il serait donc tout à fait possible d'implémenter un kernel Flow pour interagir avec le simulateur de CIL4SYS, *Sim4sys*, directement et donc de supprimer la dépendance à Sumo.

5.4 Sim4sys

5.5 Architectures logicielles

Pour satisfaire les besoins métier tout en gardant l'état de l'art en matière d'infrastructure d'entraînement, nous avons choisi d'entraîner l'agent sur *Sumo* via *Flow* d'une part, et de l'interfacer avec le simulateur de production de CIL4SYS, *Sim4sys* d'autre part. L'analyse des données TomTom sur Issy les Moulineaux a permis de déterminer une zone d'étude. L'import du terrain d'analyse dans Sumo s'effectue grâce à l'outil d'export d'*Open Street Map* et la fonctionnalité d'import de carte fournie par *Flow*.

Une fois l'agent entraîné, il sera « servi » par un serveur *Flask* communiquant en Websocket via le module Python `asyncio`. Les repères utilisés pour la génération des états étant différents entre *Sumo* et *Sym4sys*, une transformation (qui reste à finaliser à ce jour) doit donc être réalisée lors de l'interaction entre l'agent et le simulateur de production.

Les figures 2 et 3 schématisent ces architectures.

6 Quartier d'étude

Le choix du quartier d'étude doit répondre à plusieurs contraintes : la présence de feux tricolores, un trafic qui ne soit pas trop dense comme l'est celui de Paris intra-muros, mais qui soit suffisamment congestionné pour que notre solution puisse apporter une amélioration. En considération de la haute dimension de la

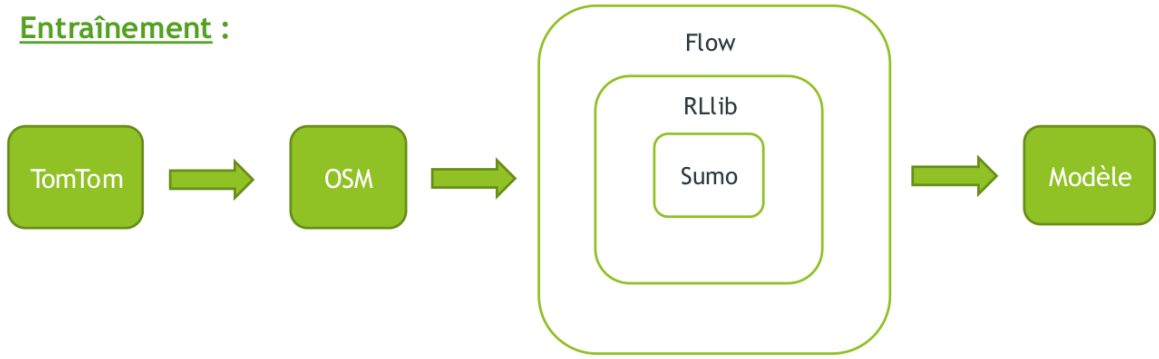


Figure 2: Architecture d'entraînement

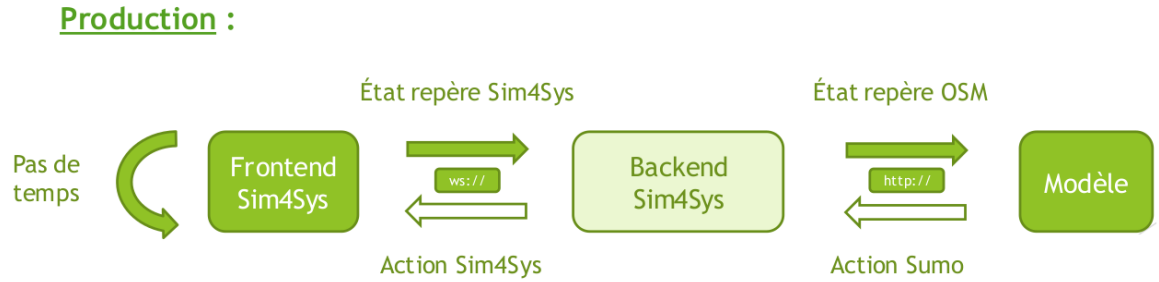


Figure 3: Architecture de production

formulation mathématique du problème, nous avons choisi d'étudier un ensemble de 4 intersections à Issy les Moulineaux, contrôlées par trois feux tricolores.

En effet, nous avons observé des congestions dans cette zone en période de pointe à l'aide de l'API TomTom et voudrions étudier si une solution par renforcement peut réduire ces bouchons.

7 TODO Ajouter image

La figure 4 bla bla bla.

8 TODO Ajouter justification

9 Modélisations

Le DQN nous permettant de résoudre la formulation mathématique de notre problème en fonction d'un ensemble d'états, d'actions possibles sur ces états et d'une récompense entre ces transitions, il faut maintenant les définir.

9.1 États

Dans une logique d'apprentissage, la question devient de trouver une représentation du système à un temps t qui encode toutes les informations nécessaires pour permettre de prendre *la meilleure* action possible selon une certaine mesure qui reste à définir. Sans nécessairement formuler formellement à ce stade cette mesure, nous savons que le but de l'action prise est de fluidifier le trafic. Ainsi, nous aimerions donc que l'état encode des informations relatives à chaque véhicule simulé ainsi qu'aux feux tricolores.

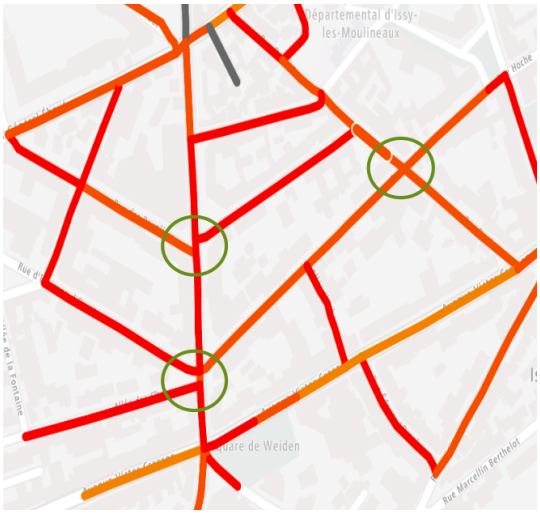


Figure 4: Les données de trafic TomTom sur le quartier séléctioné. Les intersections contrôlées par des feux sont encerclées en vert.

Notre modèle d'état est donc une matrice composée de deux parties, l'une décrivant l'état des voitures, e^v, et l'autre des feux, e^f, à un pas de temps t donné:

$$e = [e^v, e^f], \forall e \in \mathcal{E} \tag{11}$$

9.1.1 Représentation des voitures observées, e^v

En pratique l'algorithme de gestion des feux n'aura pas forcément la capacité de connaître l'état de tout les véhicules. Nous proposons donc un modèle où l'état d'apprentissage inclut seulement un sous ensemble de véhicules - envoyés par la mairie pour sonder le trafic par exemple, à l'aube des voitures électriques autonomes, cela semble tout à fait possible. Par ailleurs, la dimensionalité du problème d'étude s'en retrouve réduite.

Formellement, pour β voitures observées à chaque pas de temps t, l'état e_i^v du i ème véhicule observé est décrit de la manière suivante :

$$e_i^v = (x_i, y_i, \theta_i, v_i, \kappa_i, t_i^{v_0}) \subset \mathbb{R}^6 \text{ pour } i = 1, \ldots, \beta \tag{12}$$

Où $x_i, y_i, \theta_i, v_i, \kappa_i, t_i^{v_0}$ représentent l'ordonnée, l'abscisse, l'orientation, la vitesse absolue, le taux d'émission de CO_2, et le temps passé à l'arrêt pour le i ème véhicule observé. En concaténant les i états, nous obtenons :

$$e^v = [e_1^v, \ldots, e_\beta^v] \subset \mathbb{R}^{6\times\beta} \tag{13}$$

9.1.2 Représentation des feux, e^f

Contrairement à l'état des voitures, nous considérons que l'algorithme de gestion des feux a connaissance de l'état de l'ensemble des feux à synchroniser à chaque pas de temps. Bien que le quartier simulé n'ait que trois intersections contrôlées par des feux tricolores, il peut y avoir plus d'un feu par croisement. En

effet, chaque file peut avoir un ou plusieurs feux en considérant les feux de type *tourner à droite* ou *U-turn*. En l'occurrence, le quartier d'analyse comporte 27 feux.

Par souci de simplicité, nous **restreignons** l'état de chaque feu à *rouge* ou *vert* et **ignorons** le *orange*, cela consiste donc en une représentation binaire : $e_i^f \in \{0,1\}$. Donc :

$$e^f \in \{0,1\}^\gamma \tag{14}$$

9.1.3 Représentation finale, $e \in \mathcal{E}$

Comme évoqué précédemment, l'état $e = e^v + e^f$ de l'environnement à chaque instant t pour les β voitures observées et γ feux contrôlés est un élément de $\mathcal{E} \subset \mathbb{R}^{6\times\beta} \times \{0,1\}^\gamma$.

9.2 Actions

Dans notre modèle, l'agent peut contrôler l'état de chacun des γ feux, chaque action $a \in \mathcal{A}$ est donc un vecteur de γ booléens puisque seul les états *rouge* et *vert* sont considérés dans notre analyse. Il y a donc 2^γ actions possibles sur l'état des feux à chaque pas de temps. Le quartier comportant 27 feux, cela signifie qu'il y a un peu plus de 130 millions d'actions possibles sur le système à chaque pas de temps.

Il est donc nécessaire de choisir un sous ensemble d'actions, $\mathcal{A}^* \subset \mathcal{A}$ de manière à réduire la dimensionalité de notre problème. Par ailleurs, certaines actions théoriquement possibles ne le sont pas en pratique, nous pensons notamment aux deux actions qui passent tous les feux au même état - ce qui au mieux immobiliserait tout le trafic et au pire serait catastrophique.

La cardinalité de $\mathcal{A}^*$ ainsi que ses éléments est arbitraire bien que le but soit de trouver un « petit » ensemble d'actions qui « aient du sens » d'un point de vue de la gestion du trafic. Pour nous aider à la tâche, nous avons essayer de borner $\mathcal{A}^*$ de la manière suivante: pour trois intersections contrôlées par des feux, il faudrait que chaque intersection ait au moins deux états. Alors, si n intersections sont contrôlées par $\gamma \geq n$ feux, le nombre minimal d'actions pour que le modèle ait un sens est de $2^n \leq \gamma^n$. Dans notre cas, pour $n = 3$ et $\gamma = 27$; nous avons donc $|\mathcal{A}^*|$ bornées entre 2^3 et $|\mathcal{A}| - 2 = 2^{27} - 2$ (le -2 provenant des actions qui changent tous les feux au même état).

Pour commencer par le cas le plus simple, nous avons donc recensé deux états de feux par intersection en se demandant lesquelles permettraient de faire passer toutes les voitures à un moment ou à un autre. Les états sélectionnés sont exposés dans la structure de données du Listing 4. Cette structure de données est un dictionnaire ayant pour clefs les chaîne de caractères identifiant les feux contrôlés et pour valeurs une liste de chaîne de caractères décrivant l'état des feux selon le format défini par Sumo. Son type est donc `OrderedDict< String, List<String> >`.

Le fait que le dictionnaire soit ordonné permet de calculer le produit cartésien de ses clefs, qui représente l'ensemble des actions possibles privé de l'action identité, dans le même ordre à chaque pas de temps et donc de pouvoir associer les actions de l'agent à un nouvel état du système aisément, voir l'algorithme présenté sur le Listing 5.

Nous retenons donc huit états de feux possibles et une action pour chacun d'entre eux. Cependant, nous avons choisi d'en inclure une neuvième, celle de l'action identité qui ne modifie pas l'état des feux. *En effet, nous émettons l'hypothèse que l'agent apprendra plus vite laquelle des neuf actions ne modifie pas l'état plutôt que laquelle des huit actions faut il prendre à un état donné pour ne pas le modifier.* Nous obtenons donc finalement $|\mathcal{A}^*| = 9$.

9.3 Récompenses

Les états et actions du modèle étant maintenant définis, nous pouvons concevoir les fonctions de récompenses, l'idée étant guider l'agent durant l'apprentissage. Cependant, contrairement à une approche par

```
action_spec = OrderedDict({
    # The main traffic light, in sumo traffic light state strings
    # dictate the state of each traffic light and are ordered counter
    # clockwise.
    "30677963": [
        "GGGGrrrGGGG",  # allow all traffic on the main way w/ U-turns
        "rrrrGGGrrrr",  # allow only traffic from edge 4794817
    ],
    "30763263": [
        "GGGGGGGGGG",  # allow all traffic on main axis
        "rrrrrrrrrr",  # block all traffic on main axis to unclog elsewhere
    ],
    "30677810": [  # the smallest of all controlled traffic lights
        "GGrr",
        "rrGG",
    ],
})
```

Listing 4: Structure de données encodant les états de feux possibles du système.

```
def map_action_to_tl_states(self, rl_actions):
    """Maps an rl_action list to new traffic light states based on
    `action_spec` or keeps current traffic light states as they are.

    Parameters
    ---------
    rl_actions: [float] list of action probabilities of cardinality
        `self.get_num_actions()`
    """
    identity_action = [
        tuple(
            self.k.traffic_light.get_state(id)
            for id in self.action_spec.keys())
    ]
    all_actions = list(itertools.product(
        *list(self.action_spec.values()))) + identity_action
    return all_actions[rl_actions]
```

Listing 5: Fonction d'association d'une action prise à nouvel état des feux : $T : \mathcal{A}^* \longrightarrow \mathcal{E}^f$

optimisation sous contraintes, choisir une action de manière à maximiser l'espérance de la récompense ne garantit pas une solution contrainte. Par exemple, si la fonction de récompense pénalise les changements de feux trop fréquents mais récompense autre chose, alors l'agent évitera les changements trop fréquents, *mais ne les excluera pas de la solution.* L'agent pourra par exemple choisir de changer l'état des feux à faibles intervalles occasionnellement si cela maximise l'espérance de la récompense à cet état donné. Une solution apprise par renforcement se comporte donc différemment qu'une solution à un problème d'optimisation sous contrainte « équivalent ». Nous verrons comment pallier à ce problème ultérieurement et nous focalisons maintenant sur la conception de fonctions de récompenses pertinentes.

Nous voudrions concevoir une fonction de récompense qui encourage :

- le débit des flux
- les voitures en mouvement

Et pénalise :

- les accélérations
- les émissions de CO_2
- les changements de feux
- les voitures arrêtées pendant trop longtemps

Une fonction de récompense simple est par exemple :

$$\psi = \frac{\bar{v}}{\bar{\kappa}}$$

Où $\bar{v}$ et $\bar{\kappa}$ représentent la vitesse moyenne et le taux d'émission moyen des β véhicules observés par l'agent. Le but de cette fonction est de favoriser le débit tout en minimisant les émissions. Cependant, cette fonction est naïve et inadaptée à notre problème puisque la solution serait de laisser l'artère avec le débit de véhicules le plus important et de bloquer tous les autres. Il faudrait donc pouvoir intégrer d'autres variables à la fonction de récompense. L'image de cette fonction étant un sous ensemble de $\mathbb{R}$, ces différentes composantes ne peuvent être qu'additionnées entre elles. Cependant la fonction suivante n'a pas de sens :

$$\psi = C_1\bar{v} - C_2\bar{\kappa} - C_3\bar{t}^{v_0}$$

Où les C_i sont des constantes et $t^{\bar{v}_0}$ la moyenne d'une mesure de temps passé à l'arrêt par les voitures. En effet, le dimensionnement des constantes est très délicat puisque leurs unités sont incompatibles entre elles et ne peuvent être additionnées.

Une façon plus simple de composer la fonction de récompense et de définir un certain nombre de contraintes sur l'état des voitures et de compter les véhicules sous ou sur telle ou telle contrainte. Par exemple, une fonction qui récompense les voitures allant à une vitesse supérieure à v_{min} sera définie de la manière suivante :

$$\psi = \sum_{i=1}^{\beta} \mathbb{I}\{v_i \geq v_{min}\}$$

Cette manière de construire la récompense permet de composer ses termes. La composante due à l'état des voitures de la récompense de notre modèle, ψ^v est la suivante:

$$\psi^v = \sum_{i=1}^{\beta} \left[C_1 \cdot \mathbb{I}\{v_i \geq v_{min}\} - C_2 \cdot \mathbb{I}\{\kappa_i \geq \kappa_{min}\} - C_3 \cdot \mathbb{I}\{t_i^{v_0} \geq \tau\}\right] \tag{15}$$

Par ailleurs, pour inciter l'agent à ne changer l'état des feux que lorsqu'il « a beaucoup à gagner », nous incorporons une composante pénalisant les changements de feux :

$$\psi^f = \sum_{i=1}^{\gamma} \left[C_4 \cdot \mathbb{I}\{e_{i,t}^f \neq e_{i,t-1}^f\} \right] \tag{16}$$

Où $e_{i,t}^f$ est l'état du feu i à l'instant t. Nous obtenons alors la récompense de notre modèle aisément:

$$\psi = \psi^e + \psi^f \tag{17}$$

Cette fonction de récompense dépend donc de l'état du système au temps t mais aussi à quatre constantes, « hyperparamètres » en un sens, à accorder à la main pour le moment. En effet, valider une solution revient à la visualiser puis à la valider qualitativement à ce stade. Nous n'avons pas encore de méthode automatisable pour réaliser cela, les quatre constantes doivent donc être ajustées *manuellement*.

9.4 Environments de simulations

9.5 Notes d'implémentation

10 Infrastructure de calcul

10.1 Colab

10.2 Cluster de machines de TP

10.3 Serveur AWS

11 Méthode de validation des solutions

12 Résultats

13 Conclusion