30/12/2016

Rapport de TP

Intelligence collective : Simulation de trafic

Maxime DEGRES
Jean-Baptiste DURIEZ
Jordane QUINCY

Table des matières

I. Création, initialisation - fonction setup

Le but ici est de simuler un ou plusieurs carrefour(s) de la vie réelle afin d'y étudier le trafic, les vitesses moyennes… en appliquant certains comportements aux conducteurs et à la signalisation (feux uniquement dans ce cas-ci).

1. Création des carrefours

Dans un premier temps il faut savoir que certains patches utilisés pour la création de l'interface appartiennent aussi aux patches intersections et ou routes ; chaque patch route possédant une direction de circulation.

Les boutons seront en gras

grid_x et **grid_y** permettent de modifier le nombre de carrefours dans l'interface avec un maximum de 25 carrefours et un minimum de 0, ce qui donne une simple route.

Setup_globals permet de définir grid_x_inc, grid_y_inc (calculés en fonction de la valeur des boutons ci-dessus) et la couleur des patches non routes/intersections, avec les deux premières variables qui vont être utilisées pour le positionnement des intersections dans setup-patches.

Setup-patches permet de créer les intersections et d'y placer les drapeaux qui permettront par la suite de créer la signalisation tricolore ; on y crée aussi les routes dans setup-road.

> Setup-road permet d'ajouter les directions aux patches (N, S, E, O) et de modifier la couleur des patches de routes (patches dont la position aura été calculée précédemment grâce en partie à la taille de la route **road_size** 1 à 3 voies) et des intersections.
>
> Setup-cars est l'initialisation des agents véhicules ;

On leurs donne donc une couleur aléatoire, une position aléatoire à condition que celle-ci soit sur la route et au minimum ou maximum des latitudes et longitudes possible afin de ne pas retrouver une voiture au milieu de la route ou sur des intersections au départ de la simulation !

Leur forme/shape est initialisée grâce à setHeadingAndShapeAccordingCarDirection procédure qui permet en fonction de la direction du patch de déterminer la forme à affecter.

Les formes ont été créés préalablement (« car », « cartonorth », « cartosouth », « cartowest ») dans « tools » => « turtle shape editor ».

Finalement, une vitesse maximale et une patience leurs sont attribués, avec la patience utile afin de changer d'itinéraire en cours de route si la circulation est trop ralentie dans le carrefour.

2. Création des feux

La position des patches feux/lights est calculée en fonction de la position des bannières et de la taille de la route **road_size.**

3. Création des voitures

On crée autant de voitures qu'il y en a de spécifié dans la variable « num-cars » que l'utilisateur peut modifier avant le set-up. C'est dans la méthode « setup-cars » qu'on va pouvoir retrouver le code où les voitures sont générées. Chaque voiture se voit affecter certaines variables permettant certains comportements par la suite. Une voiture va notamment se voir affecter une vitesse max qui est un random de la « speed-limit », une patience qui est un random entre la patience-min et la patience-max, une couleur, une direction etc. Chaque voiture sera ensuite mise sur le bord du terrain, sur une route correspondant à leur direction prête à partir lorsque l'utilisateur cliquera sur « go ».

II. Fréquence des feux

1. Automatisme

C'est « ticks » qui est utilisé afin de déclencher l'événement changement de couleur des feux. Ticks est une fonction qui permet de récupérer la valeur du compteur de tick.

A. Les variables

time : nombre de ticks indiquant quand les feux ont été modifiés.

frequenceRedGreen : nombre de ticks avant passage au vert ou orange.

frequenceOrange : additionnée à frequenceRedGreen donne le nombre de ticks avant le passage au rouge.

B. Explications

Afin d'obtenir le nombre de patches feux pour un carrefour il suffit de multiplier le nombre de directions (N, E, S, O) par le nombre de voies de circulation * deux (pour les deux sens)

Pour une route à 2 voies en sens unique cela donnerait :

4 * 2 * 2 = 16 patches

Le but est donc d'agir sur chaque patch afin de lui donner la couleur correspondant au nombre de ticks atteints.

1. Modification manuelle

On doit pouvoir modifier manuellement la fréquence des feux

Il est donc possible d'utiliser la fonction [set-frequence num_label nouvelle_frequence] afin d'affecter une nouvelle fréquence à un carrefour, toutefois il est nécessaire de réaliser cette opération juste après un changement de couleur des feux (passage du vert au rouge ou inversement).

A. Les variables

La plupart des variables ont été présentées précédemment (cf. II. 1. A.)

Label : cette variable indique le numéro de la bannière placée dans l'intersection.

B. Explications

Il suffit de modifier la variable frequenceRedGreen en lui affectant la nouvelle_frequence qui elle-même permet de modifier la couleur des feux (cf. II. 1.)

III. Comportement des voitures

1. Comportement général

De manière générale, sur la route la voiture va aller tout droit (sauf lorsqu'elle passe dans les intersections, voir point suivant). Comme préciser au-dessus, la voiture a une vitesse maximale qu'elle peut atteindre et elle accélèrera toujours afin d'atteindre cette vitesse sauf dans certains cas. S'il n'y a personne devant elle, la voiture va donc augmenter sa vitesse en fonction de la valeur de la variable d'accélération que l'utilisateur peut paramétrer avant de lancer le setup. Ceci est le cas le plus simple de l'application, lorsqu'il n'y a personne ! La voiture est aussi capable de détecter si des voitures sont présentes devant elle ou non, ainsi que de savoir à quelle distance et à quelle vitesse. En fonction de la variable ahead-vision (qui est à 3 par défaut) la voiture va pouvoir voir un certain nombre de patchs devant elle. Lorsqu'une voiture détecte une autre voiture devant elle, si et uniquement si la voiture devant elle va moins vite, alors elle va ralentir pour ne pas entrer en collision. Afin d'avoir un comportement fluide, nous avons décidé que plus la voiture détectée est lointaine, plus la voiture de derrière va ralentir. La voiture va donc freiner fort au début puis moins fort. Comme la voiture ne ralentie que si la voiture détectée va moins vite qu'elle, en faisant freiner fortement la voiture dès le début, il est possible qu'au « tour » d'après la voiture ait la même vitesse que celle qui est devant et les 2 voitures pourront alors se suivre quasiment à la même vitesse ce qui donnera un peu plus de fluidité.

Nous avons mis en place un autre comportement pour la voiture, c'est sa patience. En effet, comme spécifié plus haut, chaque voiture se voit attribuer une patience. La voiture part au départ avec 0 en « wait-time » et dès qu'elle est à l'arrêt, la variable wait-time monte. Lorsque la voiture redémarre, si elle ne s'arrête pas dans les 10 prochains « tours », alors le wait-time revient à 0. Ainsi, si la voiture est bloquée dans un carrefour, son wait-time monte et s'il dépasse la valeur de sa patience, alors la voiture va tourner à droite (si elle le peut) afin de pouvoir continuer à rouler. En effet nous avons constaté au début que lorsqu'il y a avait des blocages de carrefour, il y avait toujours des voitures qui potentiellement pouvait tourner à droite. Nous nous sommes donc dits que ce comportement sembler normal, puisque dans la vraie vie, cela arrive de changer de direction parce qu'on attend depuis trop longtemps dans un carrefour bloqué. Grâce à cela, même si un carrefour se retrouve bloqué, il se débloquera tout seul avec un peu de temps grâce à ce comportement.

2. Tourner

Tout se joue dans la fonction « move », c'est ici que sont appelées toutes les fonctions nécessaires aux changements de directions.

La solution choisie est de définir à chaque entrée d'intersection la prochaine direction de l'agent. Dans un premier temps on vérifie que l'agent est sur un patch appartenant à l'intersection en vérifiant la variable booléenne des patches ; si c'est le cas on passe dans la procédure « moveInIntersection ».

On vérifie que l'on n'a pas déjà effectué le mouvement pour tourner et que l'on soit passé ici une seule fois pour cette intersection; si cette condition n'existait pas le mouvement serait réalisé en

passant sur chaque patch de l'intersection ce qui ferait tourner le véhicule jusqu'à ce qu'il sorte de l'intersection !

La fonction « getNumIntersection permet de récupérer le label (cf. II. 2. A.) de la bannière ce qui la rend unique.

A. Tourner à droite

Dans ce cas, on va juste effectuer le mouvement, trouver une nouvelle direction pour la prochaine intersection, récupérer le numéro de l'intersection dans une variable globale et finalement passer la variable booléenne de validation de mouvement à vrai.

B. Tourner à gauche

Dans ce cas-ci, il nous faut calculer « d » qui est la distance après laquelle la voiture peut tourner à gauche. La raison pour laquelle il faut signaler que le mouvement est réalisé, c'est pour continuer à avancer dans le cas où l'agent souhaite tourner à gauche.

d = (**road_size** – lane_ + 1) + speed

lane_ : variable initialisée par la fonction « getLane » qui permet d'obtenir le numéro de la voie sur laquelle se situe l'agent car.

Une fois d obtenu on récupère le patch à cette position pour que l'agent avance jusqu'à celui-ci.

IV. Résultats et statistiques

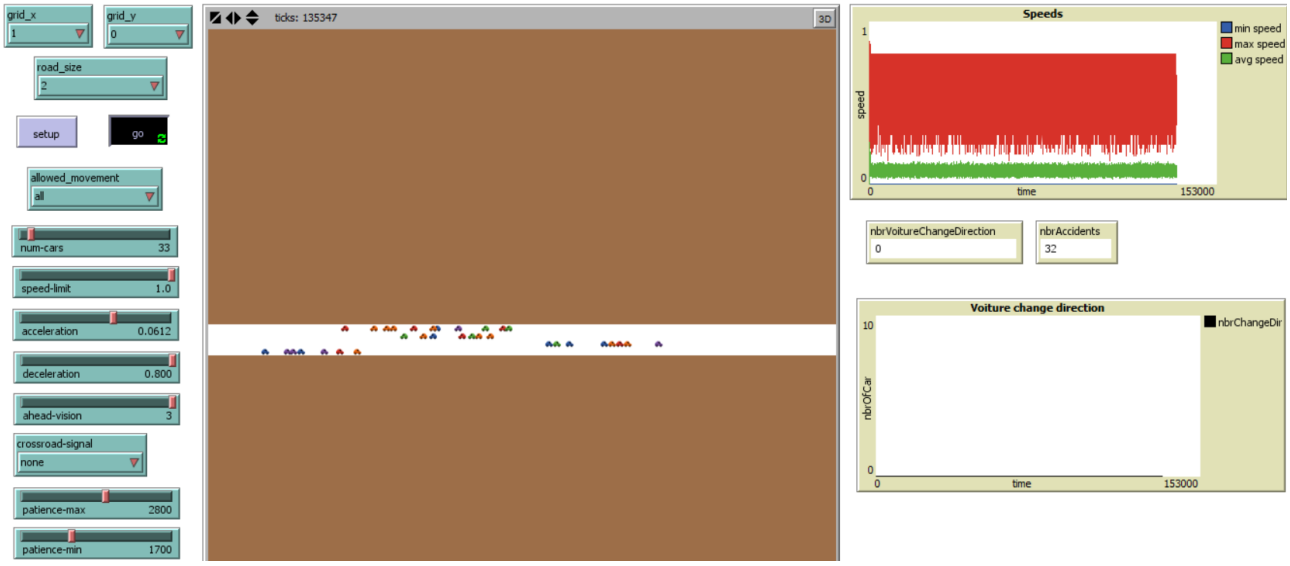
Nous avons réalisé diverses simulations afin de montrer les performances de notre système, c'est donc dans cette partie que nous allons vous présenter nos résultats.

Afin « d'évaluer » notre solution, nous nous sommes basés sur un critère fondamental qui est le nombre d'accidents. Et nous avons joué sur l'accélération maximale, la décélération maximale, le nombre de carrefours, le nombre de voiture afin de voir le comportement de notre système dans différentes situations.

Pour information, durant toutes les simulations il y a toujours un nombre d'accident à la base liée au nombre de voitures. Les voitures étant générées les unes sur les autres, comme on détecte un accident dès qu'au moins 2 voitures se trouvent sur le même patch alors à l'initialisation le nombre d'accident passe directement de 0 à environ « num-cars ». Ces accidents ne sont donc pas à prendre en compte, ceux qui viennent après sont en revanche de réel accident.

1. Accélération faible par rapport à la décélération sans intersection

Le cas le plus simple, nous avons paramétré une faible accélération par rapport à la décélération et nous n'avons pas mis de carrefour. Aucun accident après les « faux-accidents » dû à l'initialisation et ceux même après plus de 130 000 ticks.

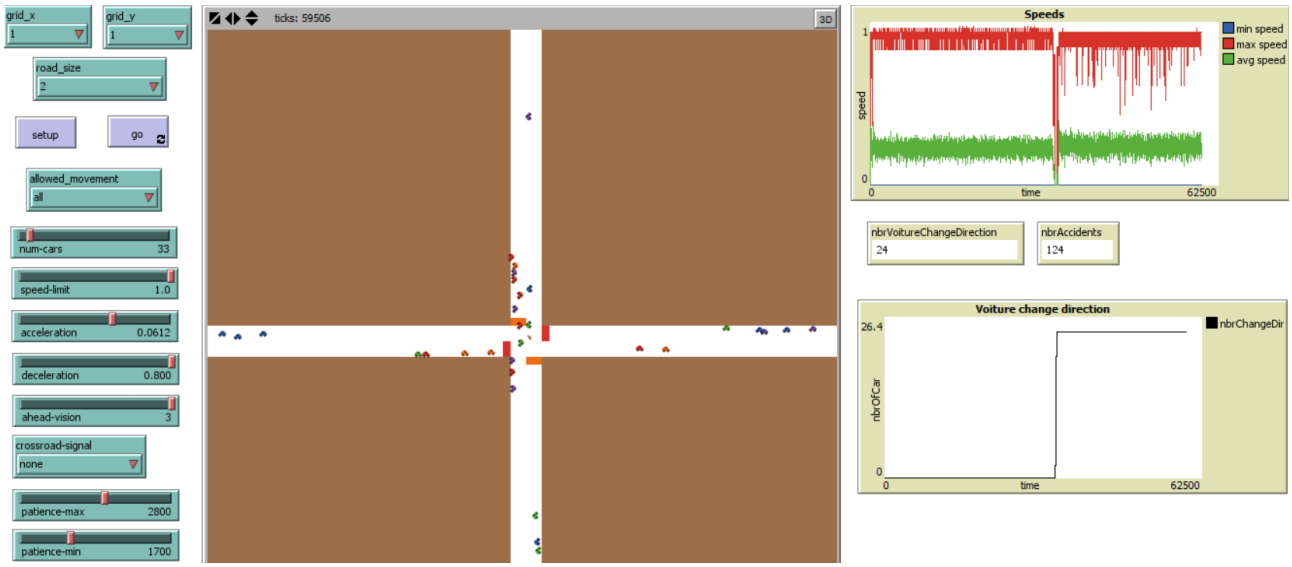


Bien entendu, heureusement qu'il n'y ait pas d'accident pour ce cas très simpliste !

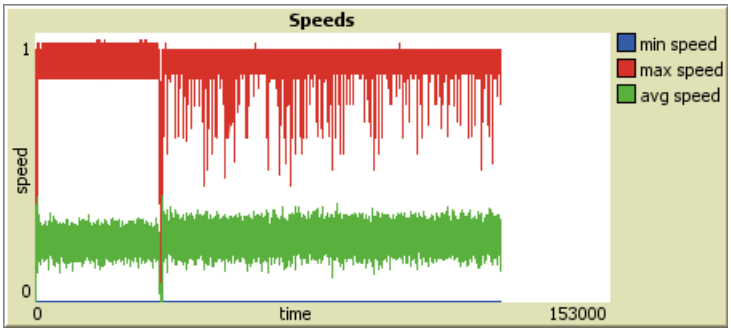
2. Même paramétrage que pour le 1 mais avec une intersection

Cette fois-ci il y a quelques accidents au début mais assez faible puis à un moment de cette simulation, le nombre de changement de direction des voitures a fortement augmentées ce qui a entraîné une hausse des accidents et qui fait donc baisser la vitesse max et moyenne des véhicules.

Ce nombre d'accidents reste quand même raisonnable puisqu'il ne dépasse pas 4 fois le nombre de voiture après environ 34 000 ticks.



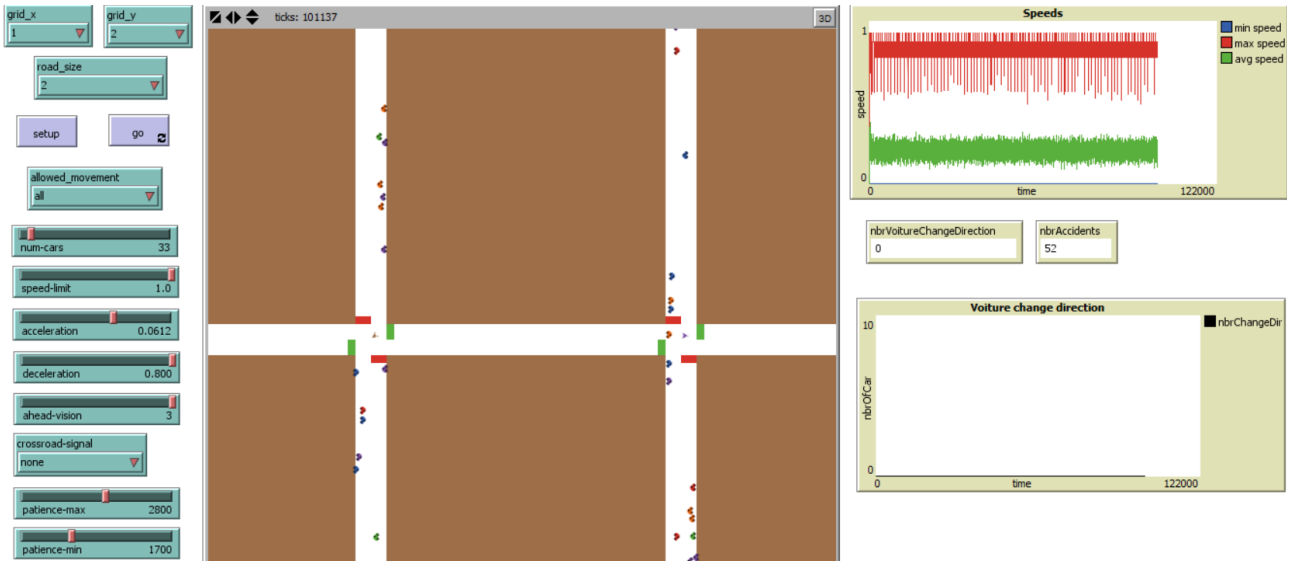
Après la montée du nombre de changement de direction, tout redevient normal et le nombre d'accidents n'augmente que de très peu : 124 à 156 en doublant le temps d'exécution (118 000 ticks). Ainsi la vitesse max et moyenne du trafic redeviennent normales.



On peut en conclure que les changements de direction causent quand même plus d'accident.

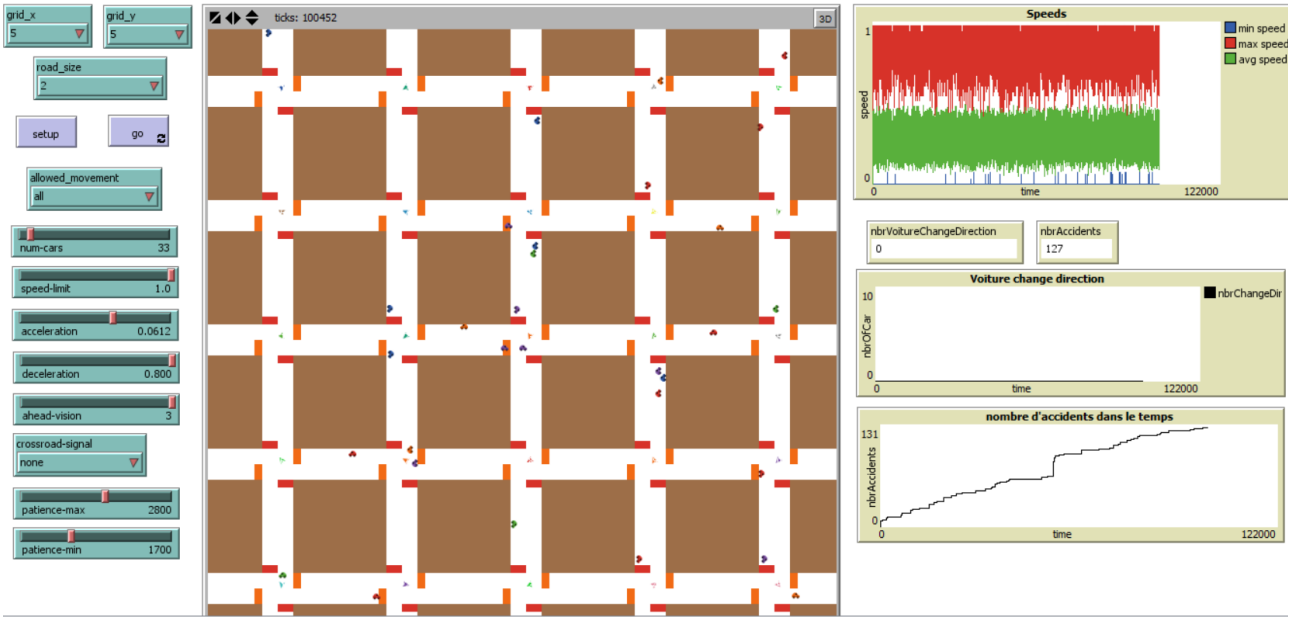
3. 2 carrefours avec changement de direction

Avec 2 carrefours les voitures sont mieux réparties, de ce fait il y a moins de « faux accidents » à l'initialisation. Et le nombre d'accident ne change quasi pas même après 175 000 ticks (52 accidents au final).

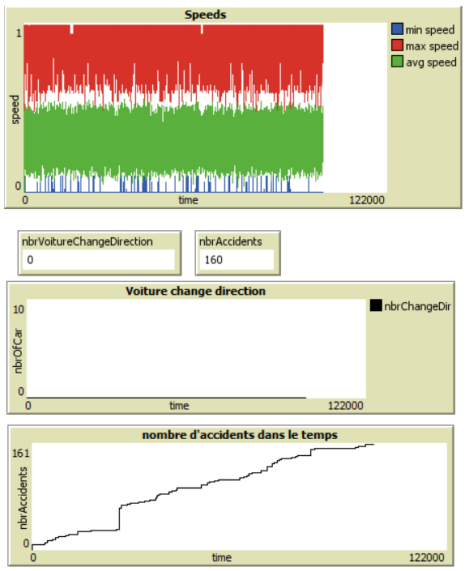


4. 5*5 carrefours avec changement de direction

Comme pour le cas 3, il y a très peu d'accidents, le nombre d'accidents fait une droite plutôt linéaire avec une très faible inclinaison. Notre solution semble donc très bien fonctionner !

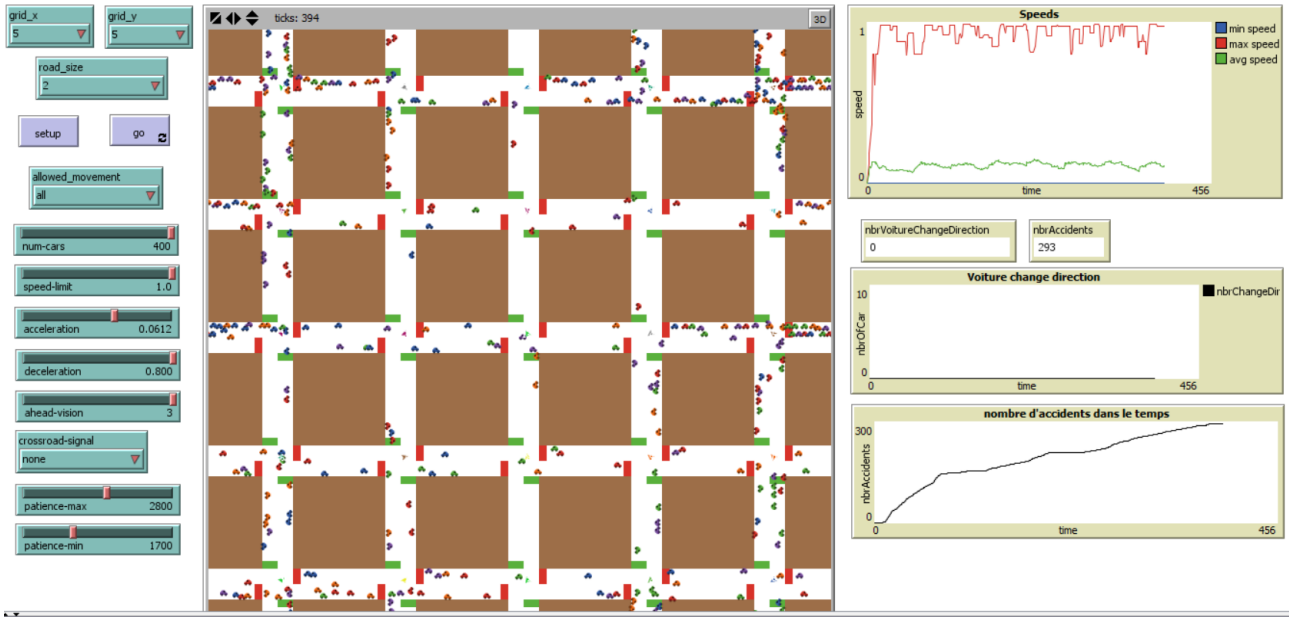


Parmi toutes nos simulations, c'est la seule pour laquelle nous avons eu une vitesse minimale supérieure à 0, c'est-à-dire c'est la seule simulation où à un moment aucune voiture n'était à l'arrêt.



5. 5*5 carrefours avec changement de direction + grand nombre de véhicule

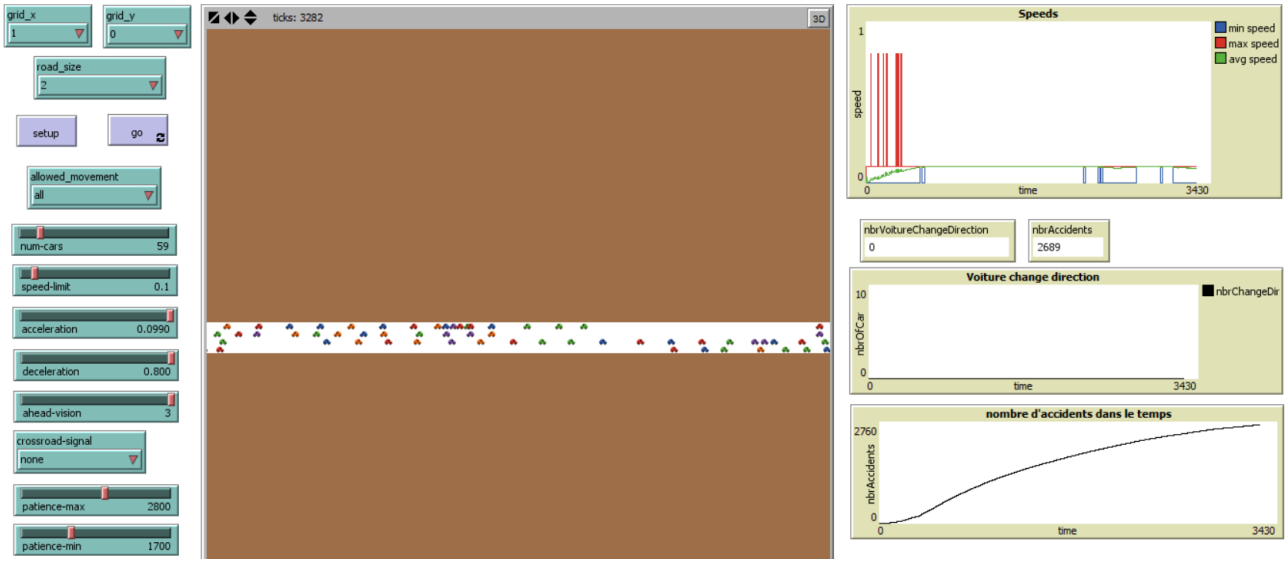
Même paramétrage que pour le cas 4, mais cette fois-ci, on a multiplié le nombre de véhicule par plus de 10. Il y a donc un peu plus d'accident mais le nombre reste assez faible, et il n'y a pas de blocage total du système.



On retrouve une courbe du nombre d'accidents assez lisse, comme précédemment.

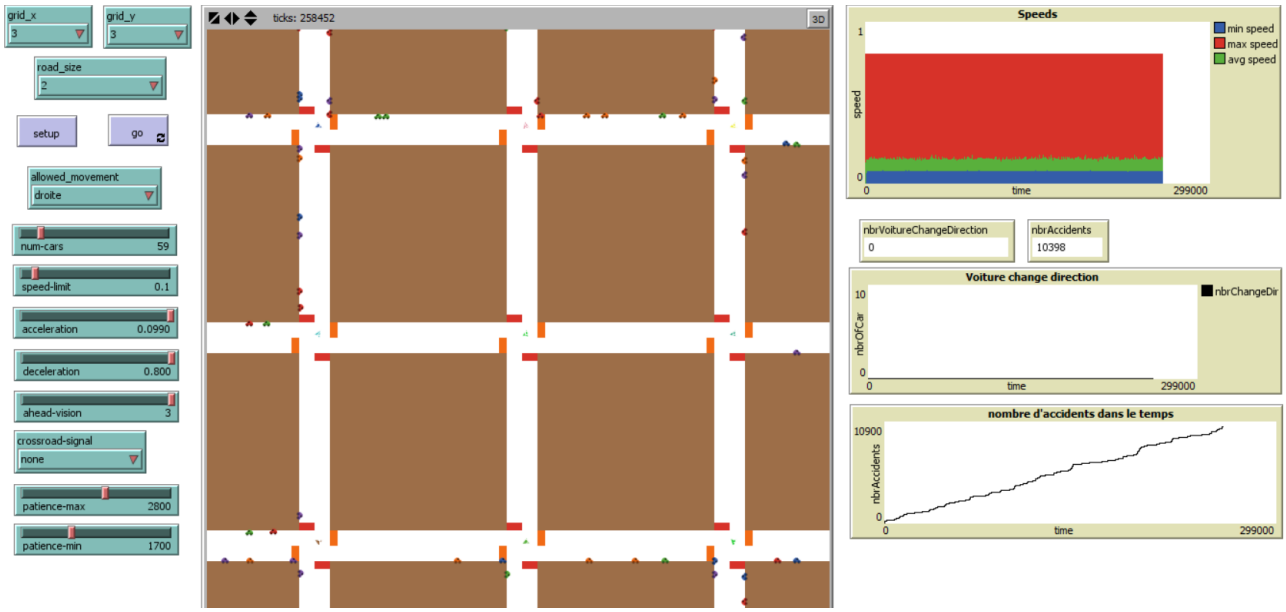
6. Aucun carrefours mais accélération très forte

Cette simulation nous a montré une faille de notre système, même s'il n'y a pas de carrefour, avec une très forte accélération il y a beaucoup d'accident… Pour régler ce problème il faudrait une vision supérieure à 3 patchs permettant ainsi de freiner plus tôt.



7. 3*3 carrefours, pas de changement de direction vers la gauche

Dans cette simulation il n'y a pas de virage à gauche, de ce fait les accidents ne viennent pas d'un changement de direction (virage à droite ne cause pas d'accident) mais juste des voitures qui sont l'une derrière l'autre. Il y a quand même des accidents, mais ils sont surtout liés au fait que nous sommes dans un espace à la fois continu et discret, afin de détecter un accident nous travaillons sur des patchs mais potentiellement même si 2 voitures sont sur le même patch elles peuvent ne pas se toucher (l'une serait sur une partie du patch et l'autre sur l'autre partie, mais l'environnement considère qu'il y a bien 2 voitures sur le même patch).



8. 3*3 carrefours, avec virage à gauche en plus

Même simulation qu'au 7, avec virage à gauche. Il n'y a pas plus d'accident, ce qui montre bien que notre système gère parfaitement le changement de direction et que l'environnement discret et continu dans lequel on travaille et une des grandes raisons du nombre d'accidents.