PAPE ADAMA MBOUP

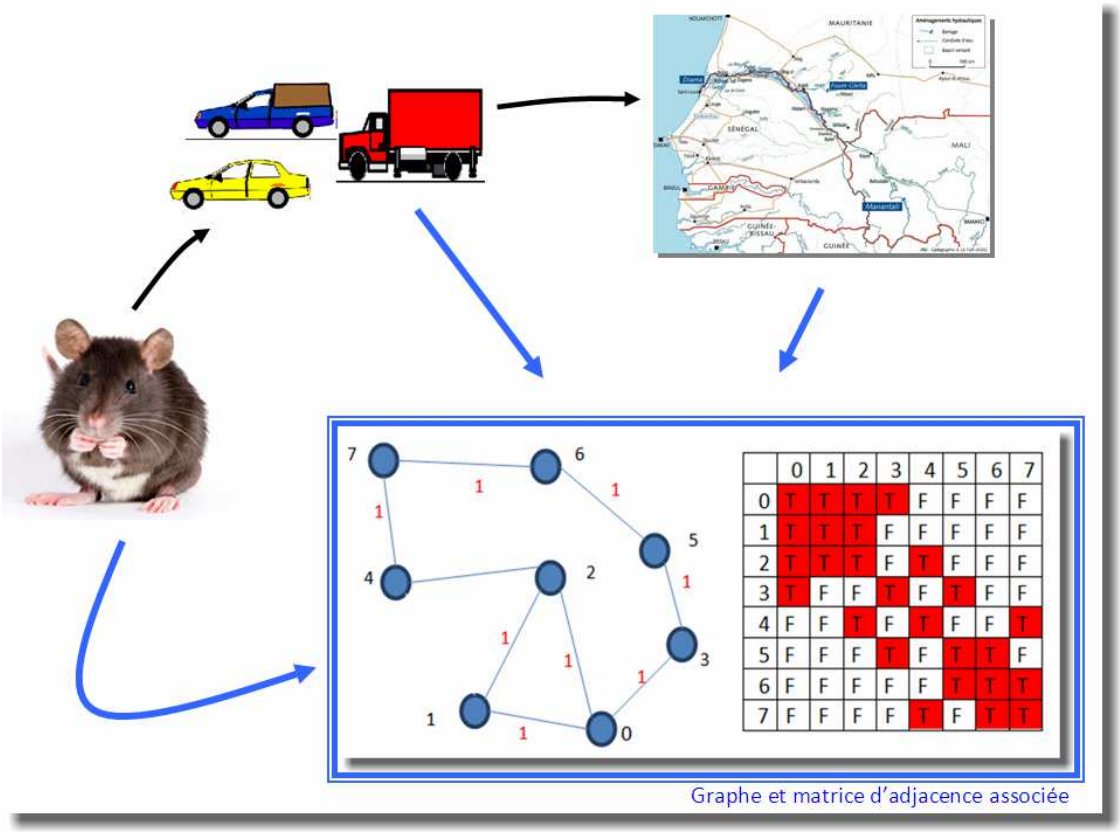
Année 2011-2012

UNIVERSITÉ CHEICK ANTA DIOP

Master 2 : BIO-INFORMATIQUE

RAPPORT MASTER DEUXIÈME ANNÉE

Construction d'un environnement de simulation multi-agents pour l'étude de la diffusion du rat noir au Sénégal au cours du siècle écoulé



	0	1	2	3	4	5	6	7
0	T	T	T	T	F	F	F	F
1	T	T	T	F	F	F	F	F
2	T	T	T	F	T	F	F	F
3	T	F	F	T	F	T	F	F
4	F	F	T	F	T	F	F	T
5	F	F	F	T	F	T	T	F
6	F	F	F	F	F	T	T	T
7	F	F	F	F	T	F	T	T

Enseignant tuteur : Karim Konaté
Maître de stage : Jean Le Fur

Institut de recherche pour le développement

Centre de Biologie pour la Gestion des Populations

Construction d'un environnement de simulation multi-agents pour le déplacement de véhicules sur des routes.

Remerciements

Après avoir remercié Allah Le créateur de l'Univers et prié sur son Prophète Mouhammad, je remercie mon père Tafsir Balla Mboup et ma mère Oumy Thioune,

je tiens à remercier :

Monsieur Jean Le Fur pour sa disponibilité, sa patience, les longues explications nécessaires à ma rapide compréhension de la plate-forme. Merci de m'avoir formé à la modélisation multi-agent et à la programmation orientée objet,

Monsieur Karim Konaté, mon co-encadreur, pour m'avoir soutenu, encouragé et m'avoir soutenu pour le bon déroulement de ce travail,

Monsieur Mouhamadou Diallo, responsable de la formation BioInformatique-BioMathématiques ainsi que tous les professeurs qui m'ont prodigué la formation indispensable à la réussite de ce travail. Merci particulièrement à Monsieur Josef Ndong pour ses explications et son brillant cours de Java,

Monsieur Laurent Granjon, responsable du CBGP-IRD de Bel-air pour m'avoir donné toutes les facilités pour la réalisation pratique de ce stage.

Toute l'équipe du CBGP de Dakar Bel-Air pour leur accueil chaleureux et leur hospitalité.

Toute l'équipe du CBGP de Montpellier, dont particulièrement Monsieur Sylvain Piry, pour avoir gentiment répondu à toutes les questions que j'ai eu à poser.

Tous mes frères, sœurs et amis qui ont contribué de près ou de loin à la réussite de ce travail.

Je remercie enfin les membres du jury pour avoir bien voulu évaluer ce travail.

Je dédie ce travail à :

Mon père Tafsir Balla Mboup et ma mère Oumy Thioune qui n'ont ménagé aucun effort pour notre éducation.

A mes frères et sœurs qui m'aiment tant et que j'aime tant moi aussi.

A ma grand-mère Adji Ngoma Dieng et sa famille qui m'ont hébergé avec hospitalité depuis que j'ai eu mon bac.

A tous mes parents et amis.

Introduction

Ce stage rentre dans le cadre du projet « Chancira[1] » dont l'objet est d'illustrer et de comprendre les processus qui régissent la diffusion du rat noir (*Rattus rattus*) et le passage à l'homme des pathogènes à travers la dynamique de la population de ce rongeur et des maladies dont il est porteur dans l'espace sénégalo-malien. Ainsi ce projet vise à décrire les processus de diffusion du rat noir au Sénégal au cours du siècle passé et jusqu'à nos jours.

Pour comprendre ces dynamiques complexes, les auteurs du projet se sont proposés dans une première phase de simuler la dynamique de colonisation de l'espèce au Sénégal depuis un siècle et ceci en relation avec l'évolution des dynamiques humaines.

Dans cette problématique, notre travail consiste à construire un simulateur sur la base d'une plate-forme[2] déjà existante de modélisation de populations de rongeurs. Ce simulateur doit permettre la représentation dans le temps des rats qui montent ou non et descendent ou non dans divers types de véhicules circulant sur les routes et dans les villes et marchés du Sénégal. Il doit aussi prendre en compte l'évolution sur un siècle des villes, des marchés, des routes et des flux de véhicules.

Nous aborderons dans un premier temps l'état de la connaissance sur la modélisation du déplacement d'agent dans un espace. Dans un second temps nous présenterons l'environnement de simulation en parlant des logiques que suivent Repast Simphony et la plate-forme SimMasto, ce qui est nécessaire à la compréhension de l'élaboration de notre module. Enfin dans un troisième temps nous présenterons la conception de notre modèle multi-agent de déplacement d'agents dans un espace représentant les routes du Sénégal.

1. Revue des approches pour la modélisation du déplacement d'agents dans un espace

Différents types d'agents sur différents types d'espaces par différentes méthodes ont été utilisés pour la modélisation du déplacement d'agents dans un espace.

Les applications développées dans ce domaine sont de nature très diverses. Wang *et al.* (2008) par exemple, ont développé un projet d'étude de système cybernétique de transport dans lequel des véhicules automatisés sans conducteur doivent évoluer dans un trafic routier. Ces auteurs ont utilisé un système multi-agents et l'algorithme de Dijkstra pour calculer le plus court chemin dans le réseau routier. Ces auteurs ont représenté les véhicules en tant qu'agents mais pas les conducteurs de ces véhicules. Doniec *et al.* (2008) se sont aussi intéressés à la modélisation du trafic routier. Ces auteurs ont aussi utilisé un système multi-agents pour représenter les conducteurs de voiture cette fois. Ils ne se sont cependant intéressés qu'au comportement des agents dans les intersections sans avoir à calculer des trajets. Ce type d'étude a aussi été réalisé par Ruskin et Wang (2002), mais en utilisant des automates cellulaires. Zamith *et al.* (2012) ont proposé un modèle de trafic fondé sur une approche probabiliste du déplacement des conducteurs. Ce travail portait sur l'intensité du trafic sur les autoroutes mais s'intéressait surtout au comportement des conducteurs sans prendre en compte de parcours dans un réseau. Nguyen *et al.* (2012) ont présenté une plate-forme complète de simulation de déplacement fondée elle aussi sur un système multi-agents et prenant en compte diverses couches d'activité (marche à pied, vélo, bus, voiture, parking, circulation). Ils ont développé une connexion avec des SIG pour représenter les réseaux dans

[1] **CHANCIRA** : **CHAN**gement environnementaux, **CI**rculation de biens et de personnes : de l'invasion de réservoirs à l'apparition d'anthropozoonoses. Le cas du **RA**t noir dans l'espace Sénégalo-malien (projet de recherche ANR-11CEPL-010, janvier 2011, 65p).

[2] http://simmasto.org

lesquels chaque segment de route était représenté comme un agent. Cependant, ici non plus, le parcours d'un réseau de routes n'a pas été abordé et l'étude concernait surtout les déplacements dans une ville. Saha *et al.* (2012) ont utilisé un SIG pour évaluer en détail le coût d'utilisation d'une route ou une autre et l'ont pris en compte. Dans leurs travaux, ces derniers auteurs ont largement utilisé l'algorithme de Dijkstra pour trouver le plus court chemin entre un point de départ et une destination.

Très généralement, les auteurs utilisent les systèmes multi-agents pour aborder le trafic routier. Une autre voie aussi bien explorée est l'utilisation d'automates cellulaires (Nguyen *et al.*, 2012). Cette approche est plus utilisée pour traiter d'un grand nombre de véhicules dotés de comportements très simples. Presque tous les documents consultés ont utilisé l'algorithme de Dijkstra pour trouver les plus courts chemins. Enfin, certaines approches originales sont aussi rencontrées comme celle de Negenborn *et al*, (2007) qui ont utilisé l'analogie avec le courant électrique et comparé les propriétés entre les circuits en série et les circuits en parallèle dans des réseaux électriques pour étudier le meilleur moyen de contrôler la circulation.

2. Présentation de l'approche retenue : les systèmes multi-agents (SMA)

La diffusion des rongeurs sur un territoire pendant un siècle est un problème assez complexe qui peut être utilement abordé par la modélisation. Etant donné que la plate-forme que nous avons utilisé et sur laquelle nous avons ajouté un module était déjà basée sur une modélisation de type système multi-agent (SMA), nous avons continué sur cette voie.

Sans entrer dans le détail (voir à ce sujet le cours de Ndiaye, 2012, Ferber, 1995), nous décrivons ci-dessous les principes de la modélisation multi-agents. En ce qui concerne son implémentation dans la plate-forme SimMasto, on peut se référer aux travaux précédents de Baduel, 2010, Longueville, 2011. Nous nous limitons ici à donner une définition des SMA et d'un agent.

Définition des Systèmes multi-agents (SMA) : la modélisation du comportement intelligent d'une seule entité informatique ou agent est l'intelligence Artificielle (IA) classique. Pour traiter plusieurs agents, l'intelligence artificielle distribuée (IAD) cherche à représenter le comportement collectif via le comportement individuel pour la résolution de problèmes complexes. Elle s'intéresse alors aux comportements intelligents provenant de l'activité coopérative de plusieurs agents. Ce qui introduit le concept de système multi-agents.

Les systèmes multi-agents (SMA) sont composés d'entités informatiques distribuées qui interagissent entre elles. Il s'intéresse aux comportements collectifs produits par des interactions de plusieurs entités autonomes et flexibles. Ces entités peuvent opérer de façon collective et décentralisée (notion de distribution) pour accomplir les tâches pour lesquelles elles ont été créées (Ndiaye, 2012). Les caractéristiques du modèle de l'agent sont : la coopération, la coordination et la communication.

Définition d'un agent : Suivant les auteurs, il y a plusieurs définitions de la notion d'agent dans un SMA. Nous retenons celle de Ferber (1995): un agent est une entité autonome, réelle ou abstraite, qui est capable d'agir sur elle-même et sur son environnement (perception), qui, dans un univers multi-agents, peut communiquer avec d'autres agents, et dont le comportement (action) est la conséquence de ses observations, de ses connaissances (mémorisation) et des interactions avec les autres agents (Ferber, 1995).

3. Présentation de l'environnement de simulation

Pour bien aborder le rapport technique et pour une facile compréhension du lecteur, nous jugeons nécessaire de faire une petite introduction sur l'outil **Repast Simphony** et la logique qu'il suit puis sur ce qui existait déjà et que nous avons utilisé de la **plate-forme SimMasto**.

A. Repast Simphony

Publié le 5 mars 2012, Repast Simphony 2.0 est un système multi plate-forme très interactif de modélisation basé sur **Java** et incorporé dans l'environnement de développement **Eclipse**[3]. Il fonctionne sous Microsoft Windows, Apple Mac OS X et Linux. Il prend en charge le développement extrêmement flexible de modèles d'interaction d'agents pour une utilisation sur des ordinateurs personnels et sur des machines en clusters de petite taille. Les modèles de Repast Simphony peuvent être développés sous différentes formes, y compris ReLogo, point-and-click flowcharts[4], Groovy, ou Java, et tous ceux-ci peuvent être entrelacés de manière fluide. Les modèles développés dans NetLogo peuvent également être importés. Repast Simphony a été utilisé dans beaucoup de domaines y compris les sciences sociales, les produits consommables, les chaînes de ravitaillement, la construction et le trafic pédestre… (http://www.repast.sourceforge.net/).

Nous n'allons pas étudier tout le fonctionnement du logiciel, dans le cadre de la plate-forme SimMasto, on peut voir à ce sujet les travaux précédents de Baduel et Le Fur (2010), Realini (2011), Longueville (2011), Comte (2012)[5] et les références de Repast Simphony[6]. Nous nous limitons ici à expliquer ce qui est nécessaire à la compréhension de l'étude : le contexte, les agents, le modèle de paramètres, les projections et le pas de temps ou step ou tick.

Le contexte : Le contexte (dans Repast Simphony "context") est l'élément le plus abstrait et le plus basique de la simulation. Il est immatériel mais englobe l'ensemble de la simulation. C'est l'infrastructure minimale permettant de contenir une population d'agents sans prendre en compte la notion d'espace ou de relation.

Les agents : Nous avons expliqué ce qu'est un agent dans la section 2, page 4. Ici nous allons dire tout simplement que les agents dans notre modèle sont les humains avec leur véhicule (HumanCarrier) et les rats (Rodent).

Le modèle de paramètres : C'est un modèle permettant de définir les paramètres utilisés par la simulation et que l'utilisateur peut régler par l'intermédiaire de l'onglet "Parameters" de l'interface graphique[7] (figure 1).

[3] http://www.eclipse.or

[4] Cette méthode de développement utilise une interface graphique permettant de programmer les interactions et le comportement des agents à l'aide de schémas créés et manipulés à la souris.

[5] http://lefur.jean.free.fr/1jean/publications/2009_ChainesTraitements.pdf et http://lefur.jean.free.fr/1jean/publications/2011-RapportStage_AR.pdf

[6] "RepastJavaGettingStarted" dans le dossier "docs" de Repast Simphony téléchargeable sur le site http://www.repast.sourceforge.net

[7] Repast Simphony gère l'interface graphique de la simulation

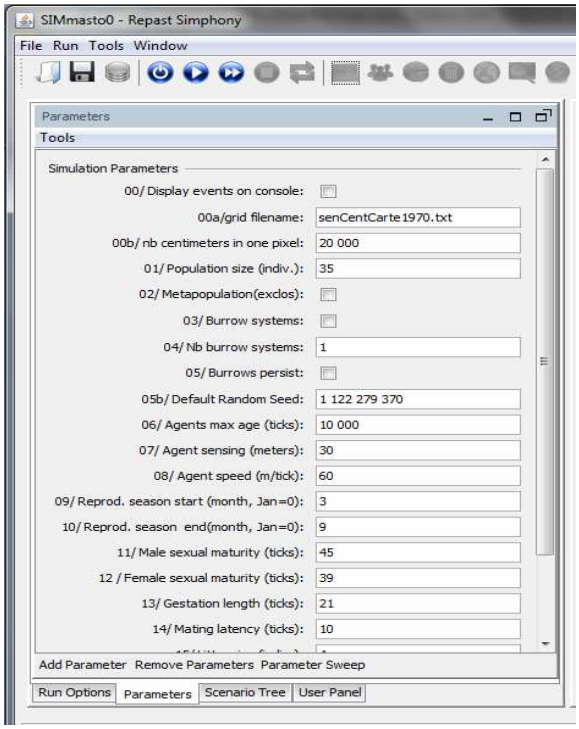


Figure 1 : L'onglet Parameters de Repast Simphony

Les projections : le contexte est immatériel et ne prend pas en compte la notion d'espace et de relation Il est donc nécessaire d'avoir une structure permettant de le faire. Les projections sont les supports réels qui permettent la localisation et le déplacement des agents. Elles permettent aussi l'interaction entre eux et entre eux et l'environnement. Les projections sont créées pour des contextes spécifiques et vont contenir automatiquement tous les agents contenus dans le contexte. Quatre types de projection de l'espace de Repast Simphony peuvent être utilisés (SIG, réseau, grille et espace continu). Mais seules deux de ces projections nous intéressent dans ce rapport :

- **La grille** : il s'agit d'une matrice, un plan découpé en cellules où chaque cellule a une valeur (un nombre décimal)[8]. Sa position est donnée par des coordonnées discrètes (sa ligne et sa colonne). Le nombre de lignes et de colonnes de la grille et la taille de ses cellules sont à déterminer et peuvent être données en paramètre. Donc si une cellule fait 1km² et que l'on utilise les coordonnées suivant la grille, alors tous les agents sur une cellule (de 1km²) auront les mêmes coordonnées. Mais ce problème est réglé par l'espace continu (ci dessous). Avec les grilles il est plus facile d'identifier les différentes zones de l'espace (ville, routes, marchés…) et de gérer la notion de voisinage[9]. (Figure 2).

[8] Dans l'affichage la valeur de chaque cellule peut être représentée par une couleur.
[9] Utile pour construire le graphe (page 13).

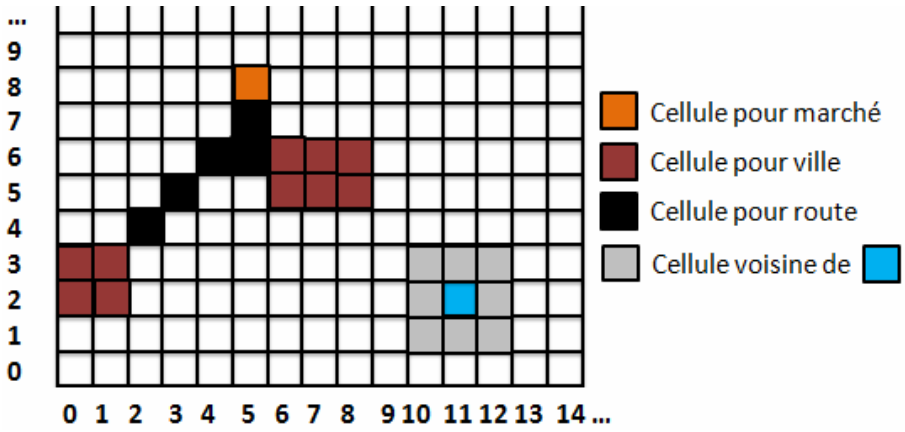


Figure 2 : Exemple de grille et de son utilisation

Comme dans la plate-forme, la valeur d'une cellule sera appelée « affinité » dans la suite du document. La construction du graphe se fera en n'utilisant que la grille comme projection.

- **L'espace continu :** Il s'agit d'un plan muni d'un repère cartésien c'est-à-dire où la position de chaque point est représentée par des coordonnées réelles (décimales). Elles sont utiles aux déplacements et à la localisation exacte des agents. En effet si un agent est dans une cellule de la grille, l'espace continu lui permet de se déplacer dedans et de connaître à chaque instant sa position exacte. Il peut de plus identifier son voisinage dans son rayon de perception (paramétrable) (Figure 3).

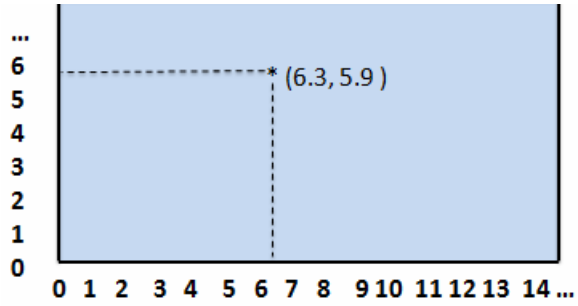


Figure 3 : Un point et ses coordonnées continues dans l'Espace continu

L'espace continu et la grille sont superposés et peuvent être utilisés simultanément (figure 4).

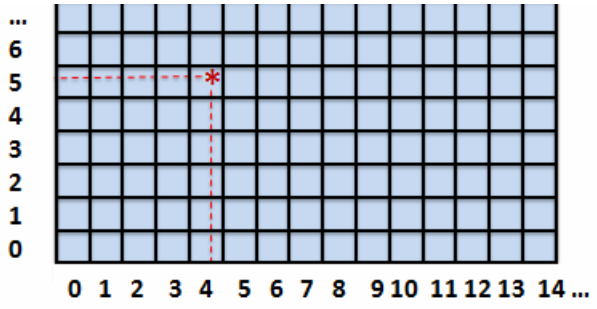


Figure 4 : Image illustrant l'espace continu et la grille superposés

Note : Les projections SIG et réseau n'ont pas été traitées ici car elles n'ont pas été utilisées.

- **Le pas de temps ou step ou tick:** une simulation se déroule suivant une boucle. On appelle **pas de temps** ou **step** ou **tick** chaque itération de la simulation. Toutes les méthodes de la simulation ne s'exécutent pas à chaque pas de temps, mais seules celles ayant juste en dessus d'elles l'annotation

@ScheduledMethod(start=1, interval=1) le font. Ces méthodes sont aussi par convention nommées step() dans les codes sources.

B. La plate-forme SimMasto

Accueilli par le laboratoire CBGP[10] et développé sous Repast Simphony, SimMasto est une plate-forme de simulation sur les rongeurs, conçu avec robustesse et flexibilité. Avec les travaux précédents de Jean Le Fur depuis 2009, Baduel (2010), Realini (2011), Longueville (2011), Comte (2012), cette plate-forme englobe plusieurs simulateurs (hybridation, ravage des cultures, simulation d'expérimentation en cages et en enclos, et ici, circulation centennale) avec une interface de sortie des résultats sous forme de courbes, de diagrammes et de graphes. Comme nous l'avons fait avec Repast Simphony nous allons présenter uniquement les éléments de la plate-forme qui intéressent directement l'élaboration de notre module.

SimMasto est fondée sur une architecture trois tiers. C'est-à-dire que le développement s'est porté sur les trois étages "Data-Business-Presentation" présentés sur la figure 5.

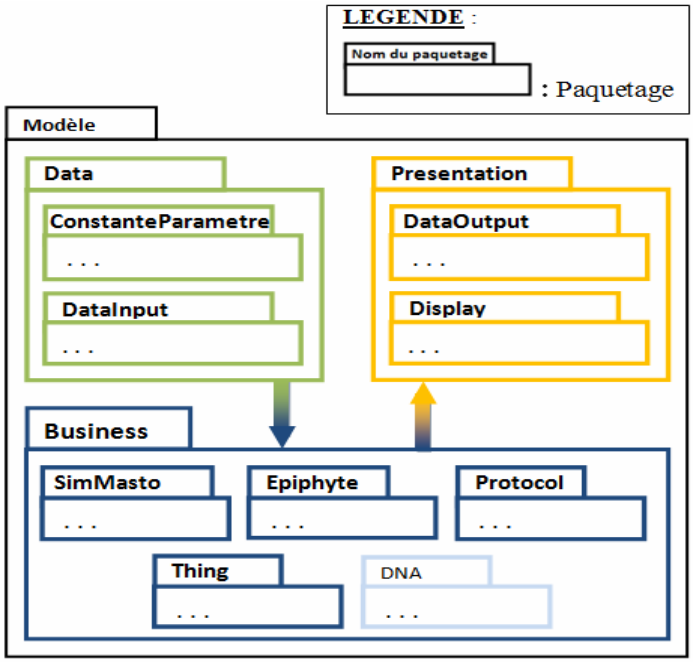


Figure 5 : Architecture Data-Business-Presentation du modèle SimMasto

Dans cette architecture :

- **Le paquetage Data** est le paquetage de gestion des données d'entrée de la simulation. Il contient :
  - Une interface contenant les constantes, et une classe permettant de lire les données en paramètre.
  - Le paquetage **DataInput** qui contient des classes permettant de lire des données dans une base de données.
- **Le packetage Presentation** gère les données de sortie de la simulation. Il a deux sous paquetage que sont :
  - **Display** gère les affichages sur l'écran (GUI).

[10] CBGP : **C**entre de recherche multi-terrains pour la **B**iologie et la **G**estion des **P**opulations

- **DataOutput** gère l'écriture des données de sortie dans une base de données ou dans un fichier.

- **Business** est l'unité centrale de la simulation. Si Data et Presentation gèrent respectivement les données d'entrées et de sorties de la simulation, **Business** lui s'occupe de la simulation en tant que telle. Il a cinq sous paquetages, mais nous n'avons utilisé que les quatre suivants :

  - **Simmasto** : Si **Business** est l'unité centrale de la simulation, alors **Simmasto** est l'unité centrale de **Business**. Il contient les classes et les interfaces permettant de construire la base du modèle. On peut en citer principalement deux :

    - ContextCreator : Cette classe permet de créer le contexte tout au début de la simulation (figure 6),

    - RasterManager : Instanciée dans ContextCreator, cette classe est chargée de la construction et de la gestion de tout ce qui est espace. Elle construit le sol avec sa grille et son espace continu et met en relation les agents et l'espace au cours de la simulation (figure 6).

  - **Epiphyte** : il contient les inspecteurs qui se chargent de récupérer et de stocker les données et de les restituer au besoin. Par exemple ils contiennent les listes des rongeurs males, femelles… Il y a un inspecteur général commun à tous les simulateurs et un inspecteur propre à chaque simulateur.

  - **Protocol** : Pour chaque simulateur, la création des agents et leur positionnement sur le sol au début de la simulation est géré par la fonction d'initialisation d'un protocole contenu dans ce paquetage. A la création du contexte, à partir des paramètres, ContextCreator choisit puis crée un de ces protocoles, et appelle sa fonction d'initialisation (figure 6). Le protocole pour notre simulateur est une classe qui s'appelle ProtocolCentenal

  - **Thing** : Ce paquetage contient les interfaces, les abstracts et les classes permettant de créer les objets agents de la simulation sous l'ordre du protocole. Par exemple la classe Rodents pour instancier les agents rats… Ce paquetage contient aussi un sous paquetage nommé **ground** (sol) contenant les classes de création et de gestion du sol. Parmi ces classes on peut citer :

    - SoilCell : Comme dans la plate-forme, nous appelons la **grille soilCellMatrix**. Cette dernière est une matrice d'objets de type SoilCell qui est une structure de données permettant de connaître au besoin de nombreuses informations sur chaque cellule de la grille :
      - son numéro ou ID sur la grille, très utile pour la construction et l'utilisation du graphe,
      - son affinité pour savoir le type de la cellule (ville, marché, routes…) et dans quelle zone (LandPlot) elle se trouve (Dakar, Thiès…),
      - ses coordonnées sur la grille et sur l'espace continu,
      - quels sont à chaque instant les agents qu'elle contient,
      - …

      Une instance de cette classe sera appelé soilCell et sera beaucoup utilisée dans la construction de graphe.

- LandPlot : c'est une structure de données contenant un tableau pouvant contenir des SoilCell[11]. Elle sert à stocker les blocs de cellules ayant des valeurs d'affinité identiques (champ, routes, villes, etc.)[12].

Les étapes successives du déroulement d'une simulation de la plate-forme existante sont synthétisées dans le diagramme des séquences (UML) présenté sur la figure 6 :

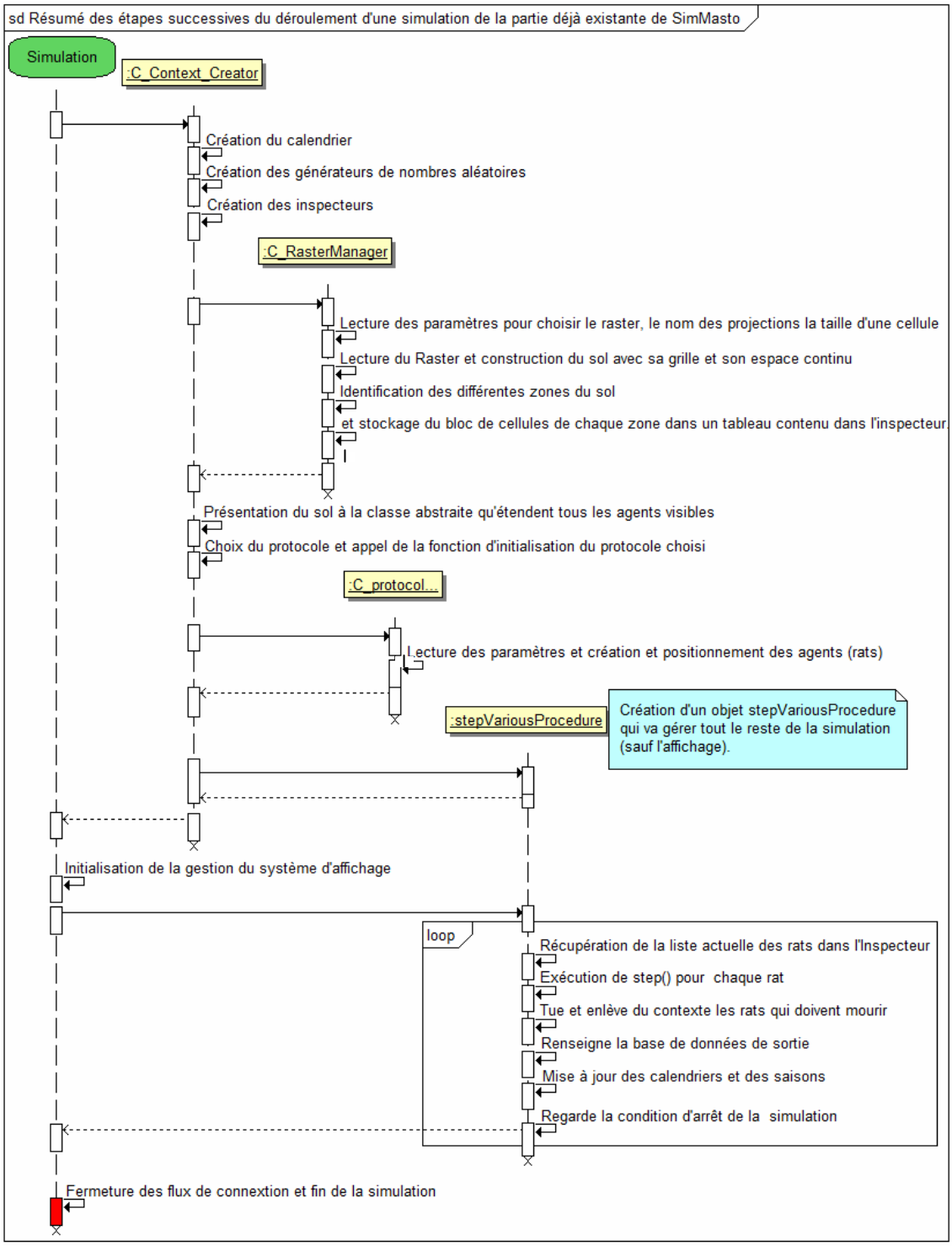


Figure 6 Résumé des étapes successives du déroulement d'une simulation (partie déjà existante) de SimMasto

[11] En UML, on peut dire que C_LandPlot est composé de C_SoilCell.
[12] Voir : 2.2.2 Identification des villes, marchés et routes

4. Conception d'un modèle multi-agents de déplacement d'agents dans un espace

A. Présentation

Au cours de notre stage nous avons conçu un simulateur permettant le déplacement sur des cartes du Sénégal, des agents transporteurs avec leur véhicule, d'une ville à une autre en passant par les routes et par le plus court chemin. Dans ce schéma nous avons aussi représenté des agents rats qui ont la possibilité de monter et de descendre de ces véhicules, ce qui leur permet d'être transportés d'une ville à une autre. Tout en réutilisant autant que possible et en restant compatible avec la partie déjà existante de SimMasto présentée précédemment, nous avons utilisé la théorie des graphes pour déplacer les transporteurs sur les routes et l'algorithme de Dijkstra pour trouver les plus courts chemins entre les villes.

Nous allons commencer par la modélisation préliminaire de l'espace, ensuite nous montrerons comment nous avons construit le graphe des routes, nous continuerons par la modélisation des agents, ensuite celle du déplacement des véhicules et enfin du déplacement des rats via les véhicules.

B. Modélisation préliminaire de l'espace

La modélisation préliminaire de l'espace suit la séquence : (*i*) numérisation des cartes en raster, (*ii*) construction du sol à partir de la carte raster, (*iii*) identification des différentes zones (landPlot) du sol et (*iv*) identification des villes et marchés à partir des zones identifiées.

a) Numérisation des cartes en Raster

La simulation doit prendre en compte l'évolution de l'infrastructure routière, le développement de nouvelles agglomérations et les flux de transport dans le pays. Ce problème est géré par la lecture successive de cartes du Sénégal présentant l'évolution de ces aspects au fil du temps. Ainsi nous avons travaillé avec les deux cartes[13] suivantes que nous ont fournies les géographes du projet (Figure 7):

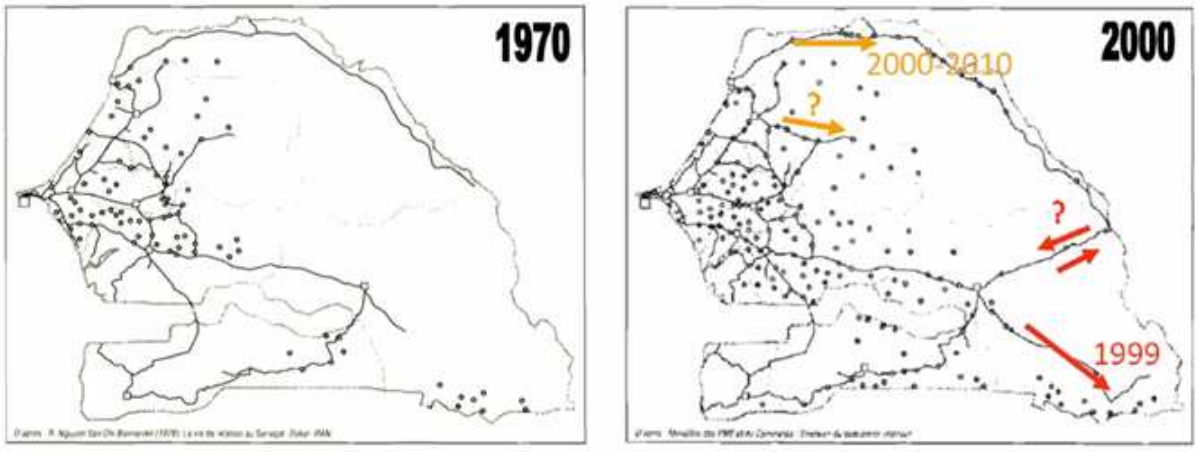


Figure 7 : Cartes des villes, routes et marchés ruraux et hebdomadaire de 1970 et 2000

Nous n'avons pas utilisé directement ces cartes en image de format JPEG. Car leurs dimensions sont grandes (respectivement 864 x 688 et 687 x 544) et les valeurs des pixels représentants les routes et les contours des marchés et villes ne sont pas significativement

[13] Dans le cadre de ce mémoire nous n'avons travaillé qu'avec deux cartes. Mais dans le cadre du projet Chancira, il y en aura d'autres.

différentes pour pouvoir être différentiées[14]. Ainsi nous avons créé des images Raster en numérisant manuellement ces deux cartes en utilisant des papiers calques, Microsoft PowerPoint et Excel et enfin Notepad++ :

- Dans Microsoft PowerPoint nous avons découpé chaque image sur les médianes en quatre parties égales. Ensuite nous les avons zoomé quatre fois et imprimées chacune sur un papier calque (figure 8).

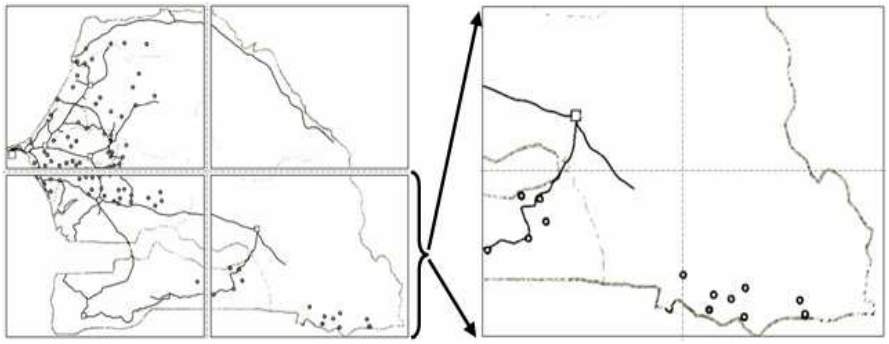


Figure 8 : Découpage et zoom d'une carte dans Power Point

- Sous Microsoft Excel, la taille des cellules a été diminuée sur 206 colonnes et 156 lignes jusqu'à ce que 103 colonnes et 78 lignes puissent se tenir sur l'écran et correspondent au quart zoomé de l'image découpée[15]. Puis nous avons plaqué à tour de rôle les papiers calques transparents sur un écran plat pour redessiner la carte sur Microsoft Excel en donnant des valeurs (affinités) aux cellules correspondant à chaque point de la carte (figure 9 A).

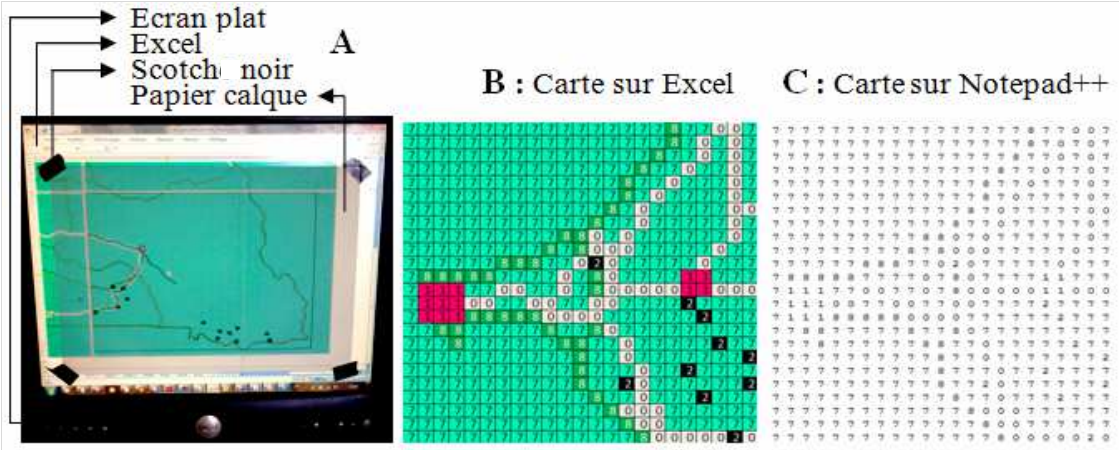


Figure 9 : Les trois étapes de la numérisation des cartes en Raster

Les affinités ont été :

- 0 pour les cellules représentant les routes (en blanc sur les cartes).
- 1 pour les cellules représentant les villes. Chaque ville est un carré de 4 cellules sauf Dakar qui est de 9 (en rouge sur les cartes).
- 2 pour les cellules représentant les marchés (en noir sur les cartes).

[14] La valeur les pixels dans ces différentes zones sont quasiment les mêmes (unicolore). Il est donc presque impossible de transformer ces images en Raster avec les logiciels de traitement d'images sans faire au préalable des retouches importantes.

[15] 206 colonnes et 156 lignes seront les dimensions du raster. Cette diminution des dimensions sera compensée par une échelle

- 7 pour les cellules représentant le fond de la carte (en vert moins foncé sur les cartes).
- 8 pour les cellules représentant les contours de la carte (en vert plus foncé sur la carte). (figure 9 B).

Ainsi les valeurs des différents types de cellules de la carte sont bien différenciées.

- Les cartes de Microsoft Excel sont finalement copiées dans Notepad++ pour remplacer les tabulations par de simples espaces et les enregistrer sous format texte (figure 9 C).

b) Identification des éléments du paysage

Après avoir lu le raster et construit le sol (constitué de la grille et de l'espace continu), SimMasto relie la grille soilCellMatrix et identifie toutes les parties en blocs de cellules de même affinité (elles correspondent par exemple à des champs, des villes, des routes) et les stocke dans un tableau trié de l'inspecteur général. Nous appelons ces blocs des **landPlot**s. Chaque landPlot a pour affinité celle des cellules qu'il contient.

c) Identification des villes et des marchés à partir des éléments de paysage.

Pour positionner les véhicules dans les villes (et ou marchés) ou les déplacer d'une ville (ou marché) à une autre, il est nécessaire au préalable de les identifier sur la grille. Pour ce faire, nous avons créé une procédure dans notre protocole (ProtocolCentenal) qui lit le tableau trié des landPlots et qui à partir de leur affinité identifie les différentes zones qui nous intéressent. C'est-à-dire les villes avec l'affinité un (1) et les marchés avec l'affinité deux (2). Ainsi nous avons identifié les villes et les marchés et nous les avons stockés respectivement dans deux tableaux triés dans notre inspecteur (InspectorCentenal). Puisque nous avons utilisé des tableaux triés, la première ville dans le tableau des villes est Dakar, la suivante est Thiès… L'ordre sera l'ordre de parcours de la grille qui est de colonne par colonne en partant du coin d'en bas à gauche (figure 10).

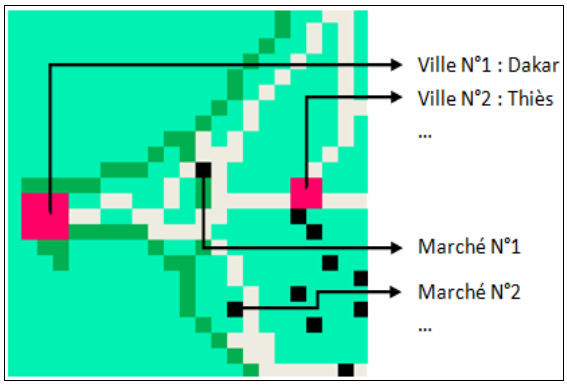


Figure 10 : Numéros et noms des différentes zones suivant leur ordre

C. Construction du graphe des routes à partir de la grille

La théorie des graphes nous permet de résoudre de nombreux problèmes dont celui de trouver et en un temps record le plus court chemin entre deux points du graphe, ce qui correspond à ce que nous voulons faire c'est-à-dire déplacer les véhicules d'une ville à une autre en passant par les routes et par le plus court chemin. Ainsi, à partir de la carte (ou de la grille) nous avons construit un graphe reliant par les routes l'ensemble des villes et des marchés.

Pour construire ce système nous nous sommes tout d'abord intéressés à la théorie des graphes.

Rappel sur la théorie des graphes

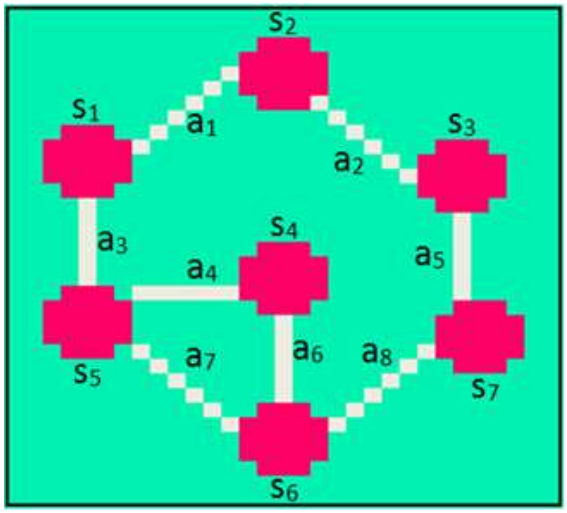
Un *graphe* permet de décrire un ensemble d'objets et leurs relations, c'est à dire les liens entre les objets.

- Les objets sont appelés les *nœuds* ou les *sommets* du graphe.
- Un lien entre deux objets est appelé une *arête*

Un **graphe *G*** est donc un couple ***(S, A)*** où

- ***S*** est un ensemble (fini) d'objets. Les éléments de ***S*** sont appelés les **sommets** ou **nœuds** du graphe.
- ***A*** est sous-ensemble de ***S*x*S***. Les éléments de ***A*** sont appelés les **arêtes** du graphe.

Une arête **a** du graphe est une paire **a**= (**s_1**, **s_2**) de sommets. Les sommets **s_1** et **s_2** sont les extrémités de l'arête **a** (figure 11 A).



	1	2	3	4	5	6	7
1	T	T	F	F	T	F	F
2	T	T	T	F	F	F	F
3	F	T	T	F	F	F	T
4	F	F	F	T	T	T	F
5	T	F	F	T	T	T	F
6	F	F	F	T	T	T	T
7	F	F	T	F	F	T	T

Figure 11 : Exemple de graphe à 7 sommets comportant 8 arêtes avec sa matrice d'adjacence

Un **chemin** est une liste de sommets telle qu'il existe dans le graphe une arête entre chaque paire de sommets successifs.
La **longueur** d'un chemin ou **distance** est le nombre d'arêtes sur le chemin, c'est aussi le nombre de sommets du chemin moins un[16].
Deux sommets **adjacents** sont deux sommets reliés par une arête.
Une **matrice d'adjacence** M est une matrice carrée, symétrique, booléenne d'ordre N telle que :

$$M(i,j) = \begin{cases} T \text{ si } s_i \text{ et } s_j \text{ sont adjacents} \\ F \quad \text{sinon} \end{cases} \quad i \text{ et } j \text{ allant de } 1 \text{ à } N$$

N est le nombre de sommets du graphe, *T* pour dire True (vrai) et *F* pour dire False (faux)

Cette matrice nous permet de savoir si deux sommets sont adjacents ou pas. Par exemple il n'y a pas d'arête entre les sommets **s_3** et **s_4** donc dans la matrice d'adjacence l'intersection entre la ligne 3 et la colonne 4 vaut F (faux) et celle de la ligne 4 et la colonne 3 vaut aussi F. Et puisqu'il a une arête entre **s_1** et **s_5** donc dans la matrice d'adjacence l'intersection entre la ligne 1 et la colonne 5 vaut T (vraie) et symétriquement. (Figure 11 B).

Notre définition d'un graphe correspond au cas des graphes *simples*, pour lesquels il existe au plus une arête liant deux sommets. Dans le cas contraire le graphe est dit multiple. Nous ne

[16] Dans un circuit ouvert nombre d'intervalles égal nombre de piquets moins un.

nous intéresserons ici qu'aux graphes simples non étiquetés, c'est-à-dire dont toutes les arêtes ont le même poids (même longueur, même durée de parcours…) et ce poids est l'unité (1).

Si nous considérons le graphe suivant (figure 12) : ville 1 et ville 2 sont les seuls sommets et il n'y a que deux (2) routes, A et B. Utiliser les graphes non étiquetés ne nous permet pas de savoir le plus court chemin entre route A et route B car chacune vaut 1. C'est donc exactement la même chose de passer sur la route A que sur la route B.

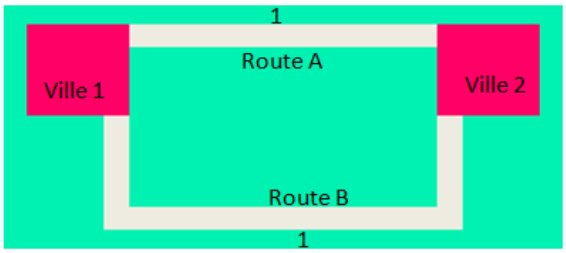


Figure 12 : Graphe à 2 sommets

Mais si nous considérons le graphe si dessous (figure 13), en considérant que chaque cellule des villes et des routes est un sommet et que la distance entre deux nœuds successifs (arête) est égale à l'unité (1), alors nous pouvons bien faire la différence entre la route A et la route B. En effet, de la cellule **α** à la cellule **β**, la longueur de la route A est 17 (16 + 1) et celle de la route B est 27 (26 + 1). Ce graphe a 66 sommets (toutes les cellules rouges et gris claires).

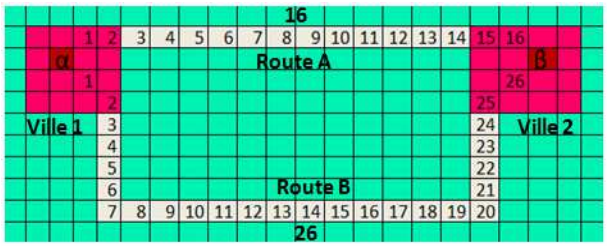


Figure 13 : Graphe à 66 sommets

Suivant cette logique, nous avons utilisé la grille pour construire notre graphe non étiqueté tout en prenant en compte les distances (figure 14).

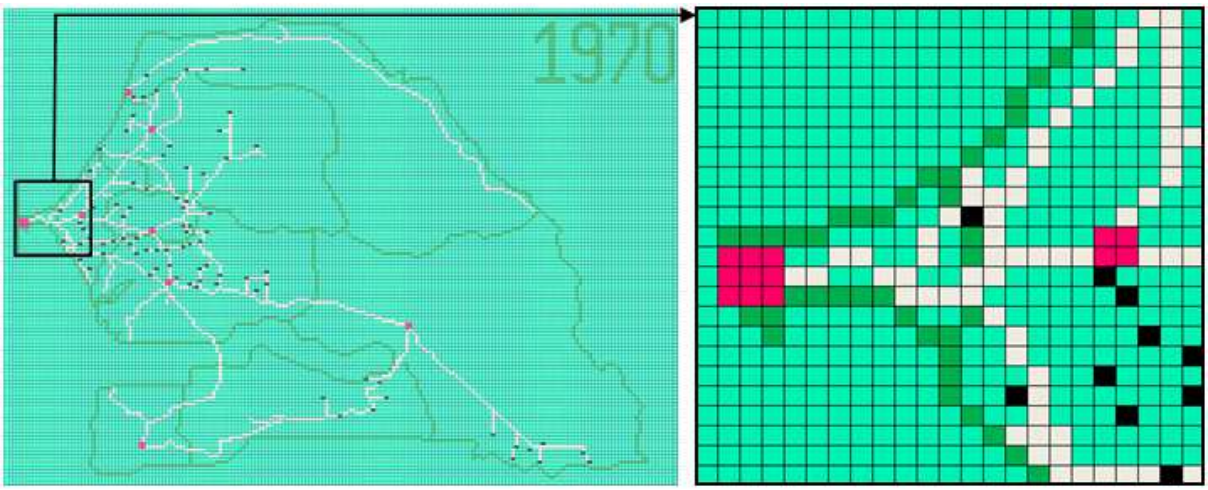


Figure 14 : Représentation de la carte numérisée du Sénégal en 1970 et détail de la grille

Construction du graphe

Un graphe non étiqueté peut être entièrement déterminé par un tableau contenant l'ensemble de ses nœuds et sa matrice d'adjacence. Si nous considérons que les nœuds du graphe sont l'ensemble des cellules des villes, des marchés et des routes alors construire le graphe revient à créer un tableau contenant l'ensemble des cellules de la grille d'affinité 0, 1 et 2 et une matrice d'adjacence de tous ces nœuds. Ainsi nous avons commencé par construire une structure permettant de contenir des données se présentant sous forme de graphe non étiqueté en créant une classe nommée **UnweightedGraph**. Cette classe est non seulement une structure de données pour la création du graphe mais aussi contient les méthodes nécessaires pour son utilisation. Voici ci-dessous le diagramme UML pour cette classe (figure 15).

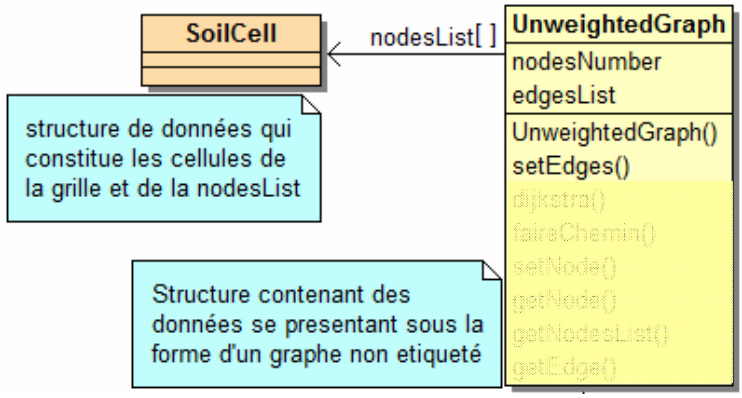


Figure 15 : Diagramme des classes UML liées à la construction du graphe (UnweightedGraph est composé de SoilCell)

Cette classe contient trois champs :

- **nodesNumber** : une variable entière permettant de contenir le nombre de nœuds du graphe,
- **nodesList** : un tableau simple de taille **nodesNumber** et de type SoilCell qui va contenir par ordre tous les nœuds (c'est-à-dire les références ou les adresses des nœuds) du graphe,
- **edgesMatrix** : est la matrice d'adjacence des nœuds. Les indices de cette matrice correspondent exactement aux indices de **nodesList**.

Seules deux méthodes de cette classe sont utilisées pour la création et la construction du graphe :

- **UnweightedGraph** (**nombreDeNoeud** : entier) : est le constructeur du graphe. Il prend en paramètre le nombre total de nœuds du graphe et s'en sert pour initialiser **nodesNumber**, instancier **nodesList** et **edgesMatrix** et initialiser à faux (F) toute la matrice **edgesMatrix**. Donc nous ne pouvons pas créer le graphe avant de connaitre son nombre de nœuds (**nombreDeNoeud**),
- **setEdges**(**i** : entier, **j** : entier, **b** : booléen) : est une méthode permettant de renseigner la matrice d'adjacence **edgesMatrix,** elle prend trois paramètres dont **i** et **j** représentent dans **nodesList** et dans **edgesMatrix** les numéros de deux nœuds dont on veut savoir s'ils sont adjacents ou pas et **b** un booléen qui vaut **True** s'il y a adjacence et **False** sinon.

Nous avons ainsi une classe prête pour la création du graphe.

Comme déjà précisé, le graphe est construit à partir de la grille. Etant donné que **RasterManager** gère le sol et contient alors la **grille** et les méthodes qui lui sont associées, nous avons réutilisé ses codes en créant une sous classe nommée **RasterGraphManager** qui hérite d'elle. En plus de ce que peut faire **RasterManager** et couplé à **UnweightedGraph**, **RasterGraphManager** est la classe qui gère la création et la gestion du graphe (figure 16).

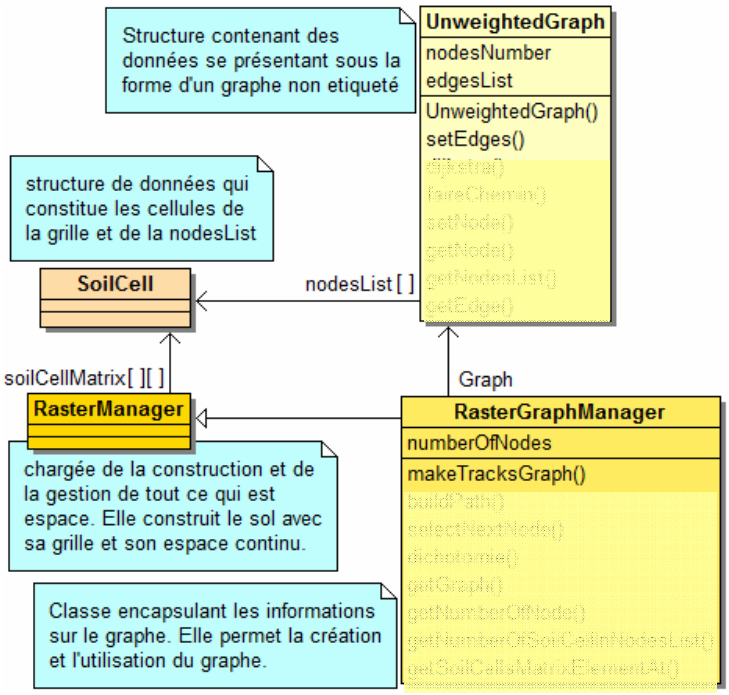


Figure 16 : diagramme des classes UML pour le graphe

Pour pouvoir créer le graphe, nous avons mis dans **RasterGraphManager** respectivement deux champs et une méthode :

- **numberOfNodes** : comme **nodesNumber** de **UnweightedGraph** ce champ sert à contenir le nombre de nœuds du graphe (avant la création de ce dernier),
- **Graph** : un champ de type **UnweightedGraph** permettant de contenir tout le graphe,
- **makeTracksGraph( )** : crée et construit tout le graphe.

Expliquer comment nous avons construit le graphe revient à expliquer comment fonctionne la méthode **makeTracksGraph( )**. Le fonctionnement de cette méthode peut être divisé en trois étapes :

Comme nous l'avons déjà dit pour créer un graphe avec notre structure de données **UnweightedGraph**, il est nécessaire de connaître son nombre de nœud.

La première étape crée temporairement un TreeSet (tableau trié) nommé **nodesListTmp**, de type générique SoilCell pour contenir temporairement et par ordre tous les nœuds du graphe. Ensuite on parcoure stoute la grille et à chaque fois qu'on rencontre une cellule dont l'affinité est celle d'une ville, marché ou route, on l'ajoute (ajoute sa référence, son adresse) dans **nodesListTmp** et on incrémente **numberOfNodes**. A la fin du parcours de la grille, toutes les cellules nœuds de notre futur graphe sont dans **nodesListTmp** et le nombre total de ses nœuds dans **numberOfNodes.**

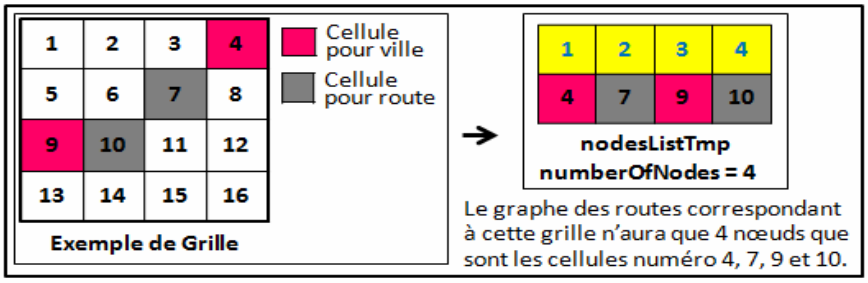


Figure 17 : Un exemple de grille

Ainsi nous pouvons aisément créer le graphe avec son nombre de nœuds et construire son tableau des nœuds. Ce qui correspond à l'étape suivante.

La deuxième étape de cette méthode consiste à créer le graphe et à copier dans son **nodesList** tous les nœuds contenus dans **nodesListTmp**. Le graphe (c'est-à-dire le nombre, le tableau et la matrice d'adjacence des sommets de la grille) de l'exemple précédent est présenté sur la figure 18.

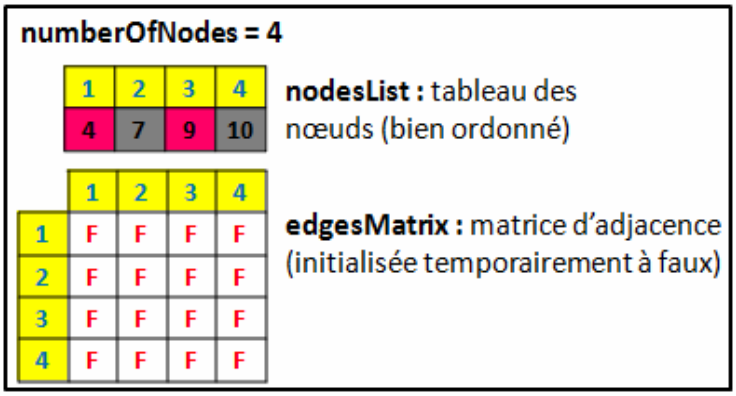


Figure 18 : les deux tableaux initiaux nécessaires à l'élaboration du chemin souhaité dans l'exemple précédent.

Comme indiqué sur la figure 18 ci-dessus, la matrice d'adjacence est partout initialisée à Faux. La troisième étape consiste donc à rectifier et à bien renseigner cette matrice. Pour mieux expliquer comment **makeTracksGraph()** le fait, nous allons au préalable revenir sur la notion de voisinage sur la grille. Deux cellules sont voisines si elles sont côte à côte. Chaque cellule de la grille a 3, 5 ou 8 cellules voisines. Si nous reprenons la figure 17, la cellule numéro 1 a trois voisines (2, 5 et 6), la cellule 2 en a cinq (1, 3, 5, 6 et 7) et la cellule 6 en a huit (1, 2, 3, 5, 7, 9, 10 et 11)... La cellule 7 est une route et parmi ses huit voisines, seules deux sont des nœuds du graphe (4 et 10). La matrice d'adjacence est renseignée sur le principe suivant : si sur la grille deux cellules voisines sont des nœuds, alors ce sont deux nœuds adjacents[17].

Etant donné que toute la matrice d'adjacence est initialisée à Faux, la rectifier revient à prendre chaque nœud de **nodesList**, identifier sur la grille ses voisins qui sont des nœuds, revenir sur le **nodesList** pour récupérer les indices des ces nœuds et mettre True (T) aux cellules correspondantes dans la matrice d'adjacence. C'est ce que fait la méthode **makeTracksGraph()** dont la troisième étape de fonctionnement est décrite à présent en continuant avec l'exemple précédent :

[17] Dire arbitrairement que chaque nœud est adjacent à lui même ou le contraire n'influe pas sur l'utilisation de la matrice d'adjacence.

Le **premier** nœud de nodesList est la cellule numéro 4 sur la grille. Et parmi ses voisines sur cette dernière, seule la cellule numéro 7 est un nœud. Donc les nœuds 4 et 7 sont adjacents. En recherchant sur nodesList[18], nous voyons que le nœud 7 correspond au **deuxième** élément. Alors nous pouvons aller sur la matrice d'adjacence et mettre T à l'intersection entre la **première** ligne et la **deuxième** colonne et symétriquement figure 20).

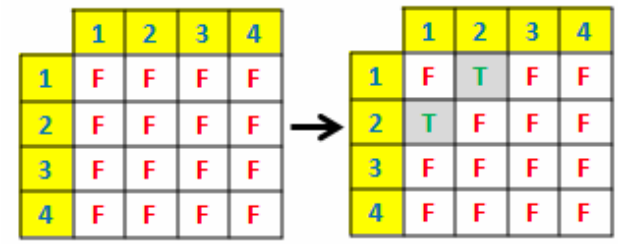


Figure 19 : edgesMatrix après la première itération de nodesList

Le **deuxième** nœud de nodesList est la cellule numéro 7 sur la grille. Parmi ses voisines, seules les cellules 4 et 10 sont des nœuds. Donc le nœud 7 est adjacent aux nœuds 4 et 10. En retournant avec la dichotomie (voir note 18) sur nodesList, nous voyons que le nœud 4 correspond au **premier** élément et le nœud 10 au **quatrième**. Alors nous pouvons aller sur la matrice d'adjacence et mettre T à l'intersection entre la **deuxième** ligne et la **première** colonne et symétriquement puis à l'intersection entre la **deuxième** ligne et la **quatrième** colonne et symétriquement (figure 21).

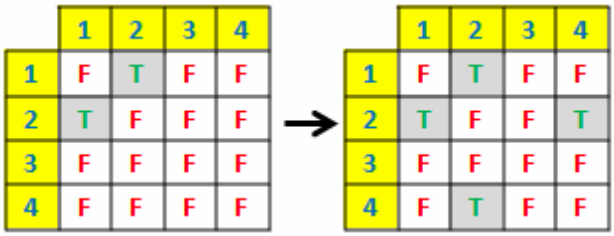


Figure 20 : edgesMatrix après la deuxième itération de nodesList

A la **quatrième** itération de nodesList, la matrice d'adjacence est renseignée (figure 22):

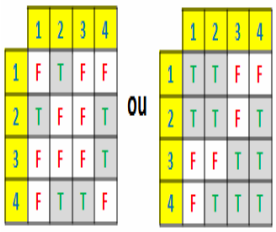


Figure 21: edgesMatrix finale

[18] Pour effectuer ces recherches fréquentes nous avons développé et utilisé un algorithme de parcours d'un tableau par dichotomie : Si nous cherchons un élément dans une matrice ou un tableau simple, nous n'avons pas besoin de parcourir toute la matrice ou tout le tableau jusqu'à voir l'élément, il suffit juste de connaître l'indice pour le tableau ou les indices (ligne et colonne) pour matrice pour y accéder directement. Par contre si nous avons un élément du tableau et que nous voulons savoir à quel indice il correspond, alors il faut parcourir le tableau jusqu'à l'élément pour ensuite récupérer l'indice. Une des manières les plus optimales pour réaliser ce parcours est la dichotomie avec une complexité de O(log2 n) où n est la taille du tableau. Mais pour appliquer la dichotomie sur un tableau, il faut que ce dernier soit ordonné ce qui est le cas avec notre nodesList. Nous avons ainsi créé dans RasterGraphManager une méthode nommée **dichotomie()** que nous avons utilisé a chaque fois que l'on disposait du numéro (ID) d'un nœud sur la grille et que nous souhaitions connaître son numéro(ou indice) dans nodesList.

Comme nous l'avons dit, nous avons travaillé avec deux cartes du Sénégal que nous avons lu successivement au cours de la simulation. Après la lecture de chacune d'elle, nous avions une nouvelle grille et donc nous devions construire un nouveau graphe.

D. Modélisation des agents

D'après la définition d'un agent dans un SMA (page 4), deux types d'agent sont représentés dans notre simulateur : les conducteurs humains avec leur véhicule (**HumanCarrier**) et les rats (**Rodent**). Le concept des SMA est très rapproché du concept de la programmation orientée objets (POO) : les agents sont des entités (classes) qui possèdent des caractéristiques (attributs) et qui sont capables de réaliser des actions (méthodes). La logique de programmation pour la simulation ne nécessite donc pas beaucoup d'apprentissage si on connaît déjà la programmation orientée objets. Ainsi modéliser les agents revient à concevoir un modèle en diagramme de classe UML renfermant les propriétés et les relations des agents.
Comme déjà présenté (page 8), SimMasto est une plate-forme de simulation de rongeurs où nous avons ajouté un module. La classe Rodent y était déjà implémentée comme agent de base de la plate-forme et de tous les simulateurs qu'elle héberge. Une super classe abstraite de la classe **Rodent** nommée **Animal** est aussi représentée qui modélise les propriétés et les comportements d'un animal en général.

Pour représenter les agents transporteurs associés à un véhicule pouvant contenir des rats, nous avons créé une classe (**HumanCarrier**) qui hérite de **Animal,** associée à une classe (**Vehicle**) permettant de décrire plusieurs type de véhicules. Chaque véhicule peut contenir un certain nombre de rats avec une probabilité de monter et de descendre. Les agents transporteurs associés aux véhicules peuvent choisir une ville comme destination mais se déplacent comme se déplace un animal c'est-à-dire ne suivent aucune route, ils vont rectilignement de leur position initiale à leur destination (figure 24).

a) Véhicules

La classe **Vehicle** a six attributs :

- **type** : champ de type objet qui fait référence au type du véhicule,
- **speed_UmeterByTick** : une variable décimale pour contenir la vitesse moyenne du transporteur, c'est-à-dire le nombre de mètres par tick (pas de temps) que parcourt en moyenne ce véhicule,
- **rodentList** : un tableau trié de type générique **Rodent** permettant de contenir des rats,
- **rodentMaxLoad** : une variable entière permettant de contenir le nombre maximal de rats que peut contenir ce véhicule,
- **loadingProba** : une variable décimale pour contenir la probabilité pour qu'un rat monte dans ce véhicule,
- **unLoadingProba** : une variable décimale pour contenir la probabilité pour qu'un rat descende de ce véhicule.

Dans la classe ProtocolCentenal nous avons mis un tableau static de type HashMap (clé valeur) nommé **VEHICLE_SPEC** contenant par ordre et suivant les différents types de véhicules, les spécifications suivantes : la vitesse du véhicule, le nombre maximal de rats qu'il peut contenir, la probabilité pour qu'un rat puisse y monter et la probabilité pour qu'un rat puisse y descendre (tableau 1).

	0	1	2	3
CAMION	30	30	0,6	0,62
MINIBUS	50	3	0,3	0,31
CAMIONNETTE	300	5	0,5	0,45
VOITURE	100	1	0,1	0,12

0 : vitesse moyenne
1 : rats charge max
2 : proba montée
3 : proba descente

Tableau 1 : Tableau des spécifications des différents types de véhicules

Ce tableau peut être paramétré par les utilisateurs du simulateur.

En plus des getters de ses attributs, la classe **Vehicle** a un constructeur qui prend en paramètre le type du véhicule que l'on veut créer, et qui à partir de **VEHICLE_SPEC** renseigne tous ses attributs sauf le tableau **rodentLoad** qui sera renseigné au cours de la simulation selon que les rats montent ou descendent de ce véhicule (figure 23, C_Vehicle). Dans cette version du simulateur, les rats peuvent percevoir jusqu'à trente (30) mètres autour d'eux.

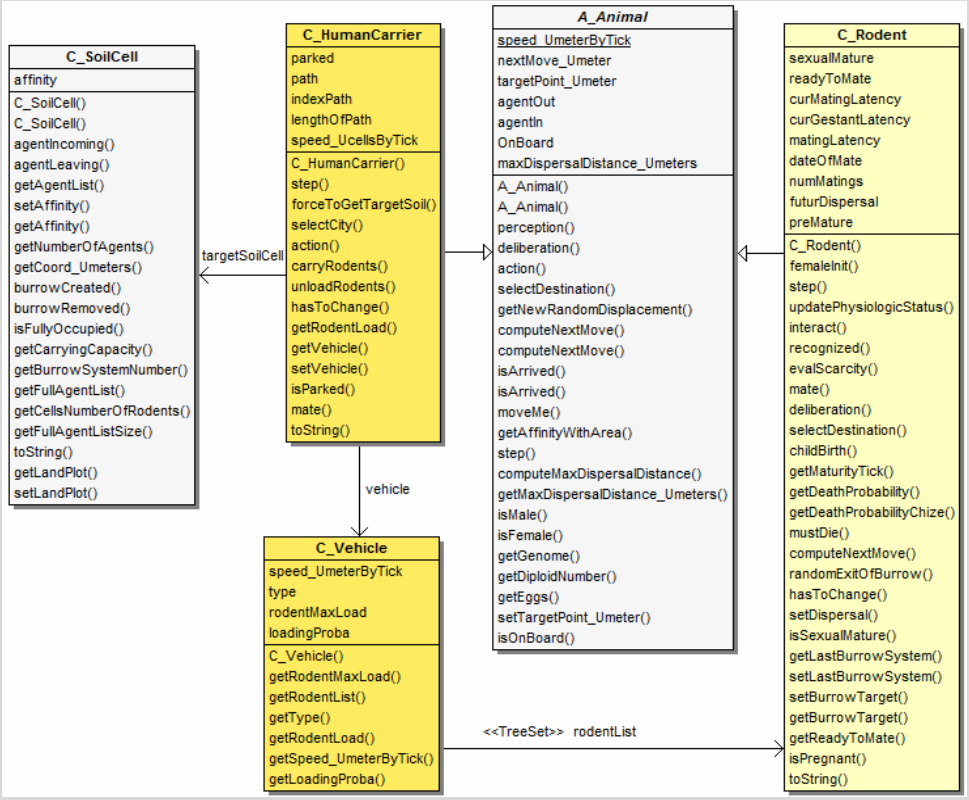


Figure 22 : diagramme des principales classes (hors espace) du package business

b) Classe générique Animal

La classe abstraite **Animal** hérite elle aussi d'une autre classe abstraite, combinées ensemble elles ont beaucoup d'attributs et de méthodes (figure 23, A_Animal) parmi lesquels nous allons citer ceux qui ont intéressé directement notre module.

Les attributs :

- **currentSoilCell** : une variable de type **SoilCell** permettant de connaitre le soilCell où se trouve l'animal,
- **speed_UmeterByTick** : une variable décimale pour contenir la vitesse moyenne de l'animal, c'est-à-dire le nombre de mètres par tick (pas de temps) que parcourt cet animal,

- **targetPoint_Umeter** : une variable de type "Coordinate" (paire de valeurs x,y) qui contient les coordonnées dans l'espace continu du point de destination de l'animal,
- **nextMove_Umeter** : une variable de type "Coordinate" qui contient la distance en mètres à parcourir suivant l'axe des ***x*** et l'axe des ***y*** pour le prochain mouvement que doit faire l'animal au prochain pas de temps.

Les méthodes :

- **step()** : s'exécute à chaque pas de temps, permettant à l'animal d'effectuer ses activité, physiologie, perception des camions, action, etc.,
- **getCurrentSoilCell()** : permet de connaître le soilCell courant,
- **getCoord_Umeters()** : permet de connaître les coordonnées d'un animal dans l'espace continu,
- **selectDestination()** : prend en paramètre une liste de soilCell et en sélectionne un comme destination. Elle retourne un **SoilCell** à partir duquel la variable **targetPoint_Umeter** est renseignée (on dispose ainsi des coordonnées suivant la grille (soilCell) et suivant l'espace continu (**targetPoint_Umeter**)),
- **computeNextMove()** : A partir de sa position actuelle, de sa vitesse (**speed_UmeterByTick**) et de sa destination (**targetPoint_Umeter**), cette méthode permet à l'animal de renseigner son **nextMove_Umeter**,
- **action()** : permet à l'animal de se déplacer après avoir calculé son **nextMove_Umeter**,
- **isArrived()** : Permet de marquer l'animal s'il est arrivé à destination ou pas.

Sur l'espace continu, un animal se déplace d'un point à un autre en passant par la droite reliant ces deux points (figure 24).

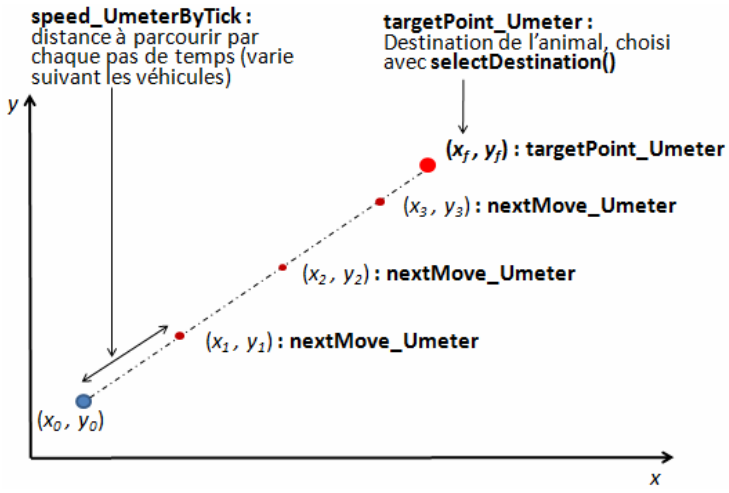


Figure 23 : Déplacement d'un animal d'un point à un autre dans l'espace continu

c) Transporteurs routiers

En plus de ce que peut faire la classe abstraite **Animal,** la classe **HumanCarrier** contient des attributs et des méthodes propres à un agent transporteur :

Les attributs :

- **vehicle** : une variable de type **Vehicle**,
- **parked** : une variable booléenne qui vaut **vraie** (True) si le véhicule est parqué et qui vaut **faux** (false) dans le cas contraire. Utile pour dire aux rats que le véhicule roule ou s'est arrêté.

- **targetSoilCell** : une variable de type **SoilCell** permettant de connaitre le soilCell visé par l'animal,
- **path** : un tableau de type entier pour contenir successivement les numéros dans **nodesList** des cellules de la grille composant le chemin que doit suivre le véhicule du transporteur pour aller d'un point à un autre.
- **indexPath** : une variable entière permettant de parcourir le tableau des chemins **path**.
- **lengthOfPath** : pour contenir la longueur du chemin.
- **speed_UcellsByTick** : nous expliquerons cet attribut dans la partie modélisation du déplacement des véhicules.

Les méthodes :

- **HumanCarrier()** : le constructeur de la classe, il prend en paramètre le type du véhicule de ce transporteur,
- **step()** : une redéfinition de la méthode step() de **Animal**,
- **selectCity()** : permet de choisir une destination. La méthode renvoie un landPlot (un groupe de soilCell). La classe **LandPlot** a une méthode **getCells()** permettant de récupérer sur un tableau de **SoilCell** les soilCells que contient un landPlot. Donc **selectDestination(selectCity().getCells())** permet de choisir une destination dans une ville.
- **carryRodents()** : effectue le déplacement des rats dans l'espace conformément à la position du véhicule,
- **unloadRodents()** : décharge les rats du véhicule du transporteur,
- **hasToChange()** : signale que l'agent a changé de statut (ex : porte des rats ou ne porte plus de rats). La méthode est utilisée pour gérer l'affichage sur le GUI,
- **isParked()** : marque si le véhicule est arrêté ou pas.

E. Modélisation du déplacement des véhicules sur les routes

Pour aller d'une ville à une autre nous avons choisi de faire passer les véhicules par les plus courts chemins. Nous avons utilisé la théorie des graphes qui est un moyen efficace et couramment utilisé pour traiter ce genre de problèmes. Le moyen le plus simple pour trouver le plus court chemin entre deux sommets d'un graphe consiste à calculer la longueur de tous les chemins possibles entre ces deux sommets et de prendre celui qui offre la plus courte distance. Mais cette méthode prend beaucoup trop de temps. Dans le cas d'un graphe complet d'ordre ***n*** c'est-à-dire un graphe de ***n*** sommets où chaque sommet est relié à tous les autres, la complexité pour parcourir tous les chemins entre deux sommets est supérieure à O((***n*** - 2)!). Si nous retenons cette estimation, sur un graphe complet de seulement 22 sommets, avec un ordinateur pouvant calculer 1 milliard de chemins avec leur longueur par second, il nous faudra plus de 77 ans pour obtenir le plus court trajet. Et plus de 490 millions d'années si le graphe a 27 sommets. Edsger **Dijsktra** nous propose un algorithme nommé Dijkstra nous permettant de résoudre le même problème avec une complexité de O($\boldsymbol{n}^2 + \boldsymbol{n}\ (\boldsymbol{n}- 1)/2$) (Le Bot, 2006)., c'est-à-dire, ce que l'ordinateur précédent fait en 77ans, avec Dijkstra il le fait 1 400 000 fois en une seconde. Ainsi nous avons utilisé cet algorithme dans notre module.

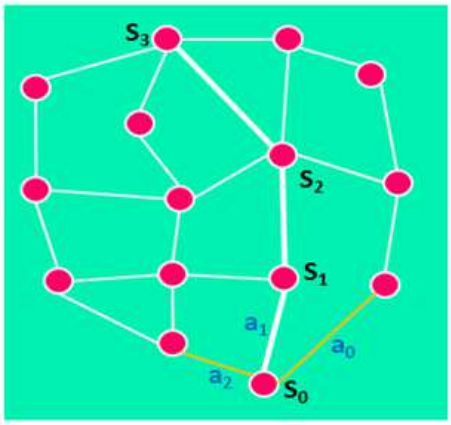
Algorithme de Dijkstra :

L'algorithme de Dijkstra permet de calculer les plus courts chemins d'un sommet **s** à tous les autres sommets d'un graphe. Il est conçu suivant les deux propriétés suivantes :

- un plus court chemin entre deux sommets est élémentaire.
- un plus court chemin vérifie le principe de **sous-optimalité.**

Le principe de sous-optimalité :

Tous les sous chemin d'un plus court chemin sont des plus courts chemins (figure 24). C'est-à-dire on n'a pas besoin de calculer la longueur de toutes les combinaisons possibles de chemin. Si par exemple passer par l'arête $\mathbf{a_1}$ constitue le plus court chemin entre les sommets $\mathbf{S_0}$ et $\mathbf{S_1}$ alors on ne passera pas par $\mathbf{a_0}$ ou $\mathbf{a_2}$ pour aller de $\mathbf{S_0}$ à $\mathbf{S_2}$.



Le plus court chemin entre S_0 et S_3 est
(S_0, S_1, S_2, S_3) alors
(S_0, S_1, S_2) est le plus court chemin entre S_0 et S_2
(S_0, S_1) est le plus court chemin entre S_0 et S_1
(S_1, S_2, S_3) est le plus court chemin entre S_1 et S_3
…

Figure 24 Le principe de sous-optimalité

Si ***D(y)*** est la longueur du plus court chemin d'un sommet ***y*** à ***s***, le principe de sous-optimalité se traduit localement par les égalités (Le Bot, 2006):

$$D(y) = \begin{cases} 0 \text{ si } y = s \\ min\{D(x) + d(x,y) \mid x \text{ voisin de } y\} \text{ sinon} \end{cases}$$

$d(x,y)$ étant la longueur de l'arête (x,y)

$d(x,y)$ étant la longueur de l'arete (x,y)

Principe de l'algorithme (Montcouquiol, 2007):

- On construit petit à petit, à partir de l'ensemble $\{S_0\}$, un ensemble M de sommets marqués. Pour tout sommet marqué S_i, l'estimation $D(S_i)$ est égale à la distance du plus court chemin entre S_0 et S_i.
- A chaque étape, on sélectionne le (un) sommet non marqué S_i dont la distance estimée $D(S_i)$ est la plus petite parmi tous les sommets non marqués.
- On marque alors S_i (on rajoute S_i à M), puis on met à jour à partir de S_i les distances estimées des successeurs non marqués de S_i.
- On recommence, jusqu'à épuisement des sommets non marqués.

L'algorithme détaillé de Dijkstra avec exemple est disponible en annexe.

Pour simplifier la lecture, dans la suite du document nous confondons véhicule et transporteur associé à un véhicule.

Par défaut un objet de type **Animal** se déplace de façon rectiligne du soilCell où il se trouve au soilCell où il veut aller. Nous allons utiliser cette propriété pour déplacer les véhicules suivant les routes.

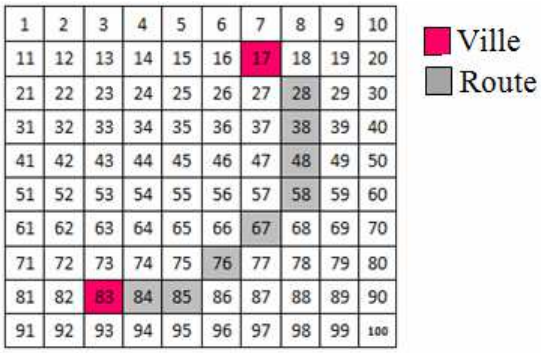


Figure 25 : Un exemple de grille avec une route reliant deux villes

Considérons la grille de la figure 25 ci-dessus. Supposons qu'un agent transporteur veuille se déplacer du soilCell numéro 83 au soilCell numéro 17 en passant par la route. Comme le véhicule se déplace en ligne droite du soilCell où il se trouve au soilCell où il veut aller, l'agent se donne comme destination la cellule 84. S'il arrive, il se donne la cellule 85, puis la cellule 76 et ainsi de suite jusqu'à ce qu'il arrive à la cellule 28 où il se donne comme destination la cellule 17. De cette façon, le véhicule se déplace de façon rectiligne entre deux cellules successives mais se déplace par la route, de la cellule 83 à la cellule 17 (comme déjà précisé, cela permet de mesure la longueur du chemin à parcourir tout en n'utilisant pas de graphe étiqueté).

Nous connaissons le soilCell où se trouve un véhicule (currentSoilCell) et le soilCell où il veut aller (targetSoilCell). Mais nous ne connaissons pas a priori l'ensemble des soilCells routes reliant currentSoilCell et targetSoilCell. Nous avons ainsi créé dans **RasterGraphManager** une méthode nommée **buildpath()** qui prend en paramètre currentSoilCell, targetSoilCell et le tableau **path** de type entier. Ce tableau a la taille du nombre de nœuds **nodeNumber** qui composent le graphe et est partout initialisé à -1. Il permet de contenir successivement les numéros des cellules composant le plus court chemin reliant currentSoilCell et targetSoilCell.

Pour avoir ce plus court chemin nous avons mis une méthode nommée **dijkstra()** (en référence à l'algorithme de parcours de graphe du même nom) qui prend en paramètre respectivement le numéro dans **nodesList** de currentSoilCell et de targetSoilCell et le tableau **path**. Cette méthode applique l'algorithme du plus court chemin de Dijkstra pour donner en un temps minimal le (un) plus court chemin entre currentSoilCell et targetSoilCell. Une fois calculé, il reste à lire successivement les numéros du tableau **path**, et de récupérer les coordonnées sur la grille et sur l'espace continu des soilCell correspondant à ces numéros. Pour cela une méthode nommée **selectNextNode()** a été placée dans **RasterGraphManager**. Les appels synchronisés de ces méthodes sont réalisés dans la méthode **step()** de **HumanCarrier**. Nous allons décrire à présent ces méthodes plus en détail (figure 26).

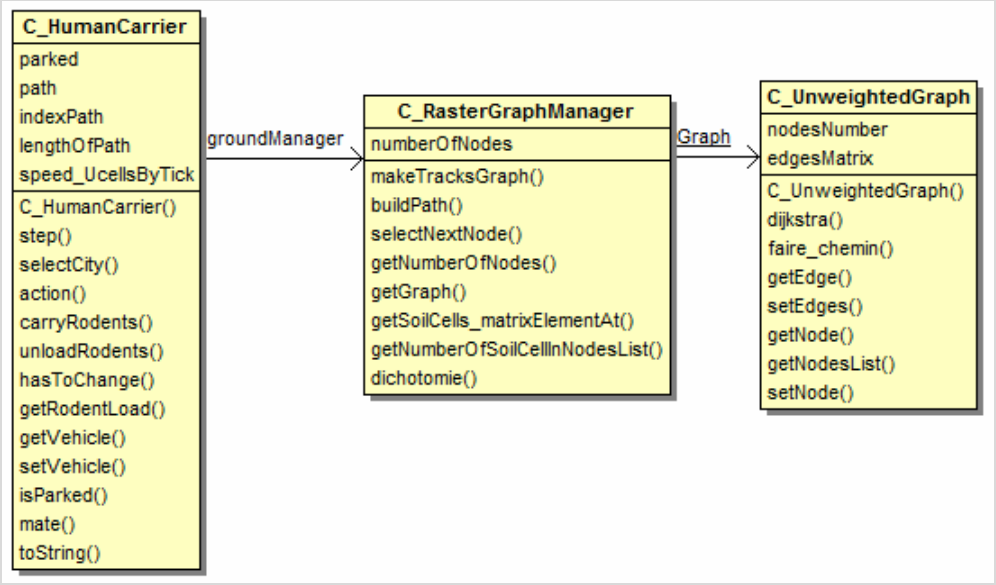


Figure 26 : Classe pour la construction et l'utilisation du graphe

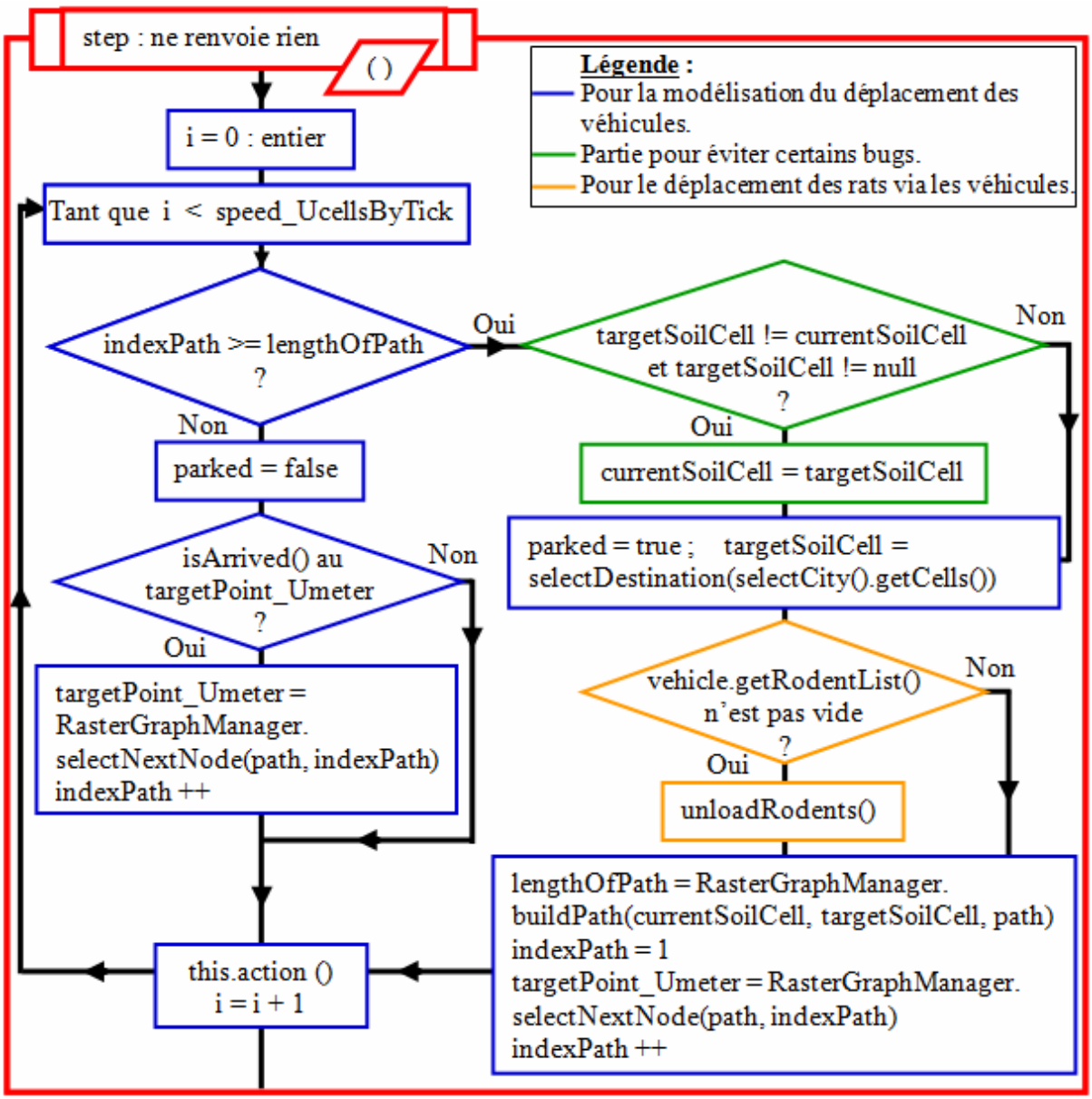
buildPath(currentSoilCell : **SoilCell**, targetSoilCell : **SoilCell**, path : tableau d'entier**)** :
A chaque fois que cette méthode est appelée, nous commençons par réinitialiser à -1 les éléments de **path** différents de -1. On récupère ensuite par rapport à la grille, les numéros (ID) de currentSoilCell et de targetSoilCell, ensuite on utilise la dichotomie pour récupérer de **nodesList** les numéros correspondants à ces IDs. Ces numéros sont stockés respectivement dans deux variables entières **pathStart** et **pathEnd**. Enfin nous appelons dans cette méthode **dijkstra(pathStart**, **pathEnd**, **path)**.

dijkstra(pathStart : entier, **pathEnd** : entier, path : adresse d'un tableau d'entier**)** : Cette méthode construit à partir de la matrice d'adjacence le plus court chemin entre le sommet d'entrée **pathStart** et le sommet de sortie **pathEnd**. Et met successivement dans le tableau **path** l'ensemble des numéros de **nodesList** des cellules composant ce chemin. Le parcours du tableau **path** est géré par la méthode **selectNextNode()**.

selectNextNode(path : tableau d'entier, **indexPath** : entier**) :** cette méthode prend en paramètre le tableau **path** et un entier **indexPath** pour le parcourir. Elle récupère ensuite du tableau nodesList du graphe, le soilCell dont le numéro est égal à l'élément d'indice **indexPath** du tableau **path**. Et enfin elle récupère les coordonnées dans l'espace continu de ce soilCell et l'envoie à la méthode **step()** de **HumanCarrier**.

step() :

Nous avons mis en place une base de données contenant pour chaque carte du Sénégal, le nombre initial de types de véhicules et le nombre initial de rats dans chaque ville. Ces nombres sont manipulables par les utilisateurs du simulateur. En lisant ces données, **ProtocolCentenal** crée les agents transporteurs ainsi que les rongeurs et les positionne respectivement dans les villes. Etant donné qu'un agent est autonome, chaque transporteur utilise sa méthode **step()** et se gère seul durant tout le reste de la simulation. Voici ci-dessous l'organigramme du déroulement de cette méthode.



Organigramme 1 : Organigramme de la méthode step()

Commentaire de l'organigramme :

speed_UcellsByTick : est la vitesse du véhicule (**speed_UmeterByTick**) divisée par la taille d'une cellule. C'est donc le nombre de cellules ou de fraction de cellule que le véhicule doit parcourir pour chaque pas de temps avec sa vitesse **speed_UmeterByTick**.

Comme la vitesse d'un véhicule peut être inférieure ou supérieure à la taille d'une cellule, o doit s'assurer que le transporteur n'ait comme destination intermédiaire qu'une cellule adjacente à celle où il se trouve. Sinon il risquerait de quitter les routes si sa vitesse speed_UmeterByTick dépasse la longueur d'une cellule. Ainsi pour un step :

Tant que le transporteur n'épuise pas le nombre de cellules qu'il doit parcourir pour chaque step (**i** < speed_UcellsByTick) :

Si indexPath est supérieur ou égal à lengthOfPath, alors soit il s'agit du début de la simulation[19] soit le véhicule a fini de parcourir son chemin ; alors :

- Le transporteur est parqué (donne alors aux rats la possibilité de monter) et se choisit une nouvelle destination,
- Il se construit ensuite un nouveau chemin à partir de sa position actuelle et de sa destination (**buildPath**(currentSoilCell, targetSoilCell, path)). Il continue par choisir pour son prochain déplacement, la première cellule de son chemin (**selectNextNode**(path, indexPath)) et incrémente le compteur indexPath,
- Et enfin il fait les calculs nécessaires pour faire le déplacement vers cette cellule (this.action) et incrémente le compteur **i**.

Sinon :

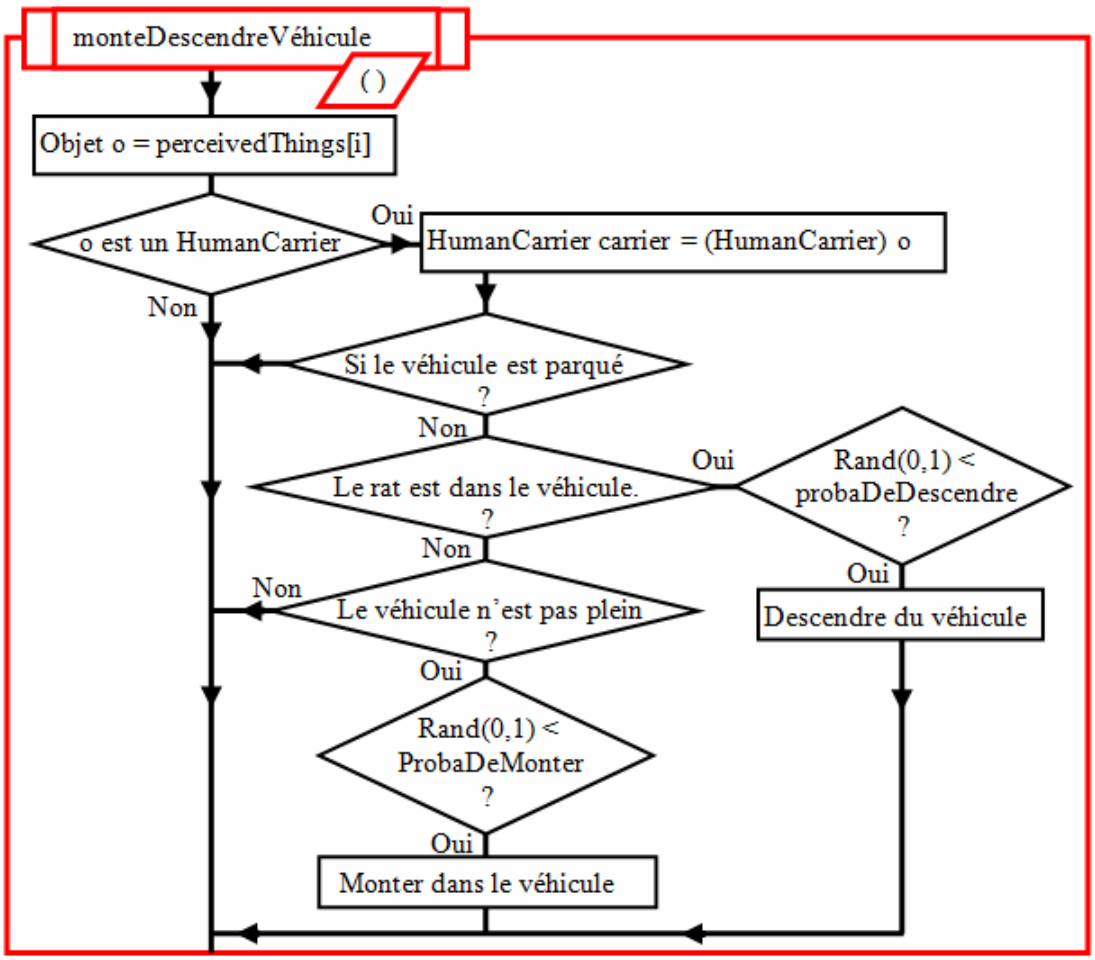
- Il n'est pas parqué (parked = false), aucun rat ne peut monter,
- Il regarde **s**'il est arrivé à la cellule qui suit celle où il était au pas de temps précédent :
    - **Si** oui, il sélectionne la cellule suivante (**selectNextNode**(path, indexPath)) et incrémente le compteur indexPath,
    - **Sinon** il continue à se déplacer pour y aller (this.action) et incrémente le compteur **i**.

F. Modélisation du déplacement des rats via les véhicules

La modélisation du déplacement des rats via les véhicules à commencé depuis la modélisation des agents, particulièrement à la modélisation de la classe véhicule (page 20).

La classe **SoilCell** et la classe **Animal** implémentent toutes une interface nommée **Situated_thing**. Comme déjà indiqué, un rat (**Rodent**) est un **Animal**, il peut donc se choisir une destination. Pour cela il peut percevoir l'environnement sur un rayon paramétrable (30 m ici). Il met les objets qu'il perçoit dans un tableau nommé **perceivedThings** de type **Situated_thing**. Si dans ce tableau il y a un objet de type **HumanCarrier** (associé à un véhicule) alors le rat a le choix de monter dans le véhicule ou d'y descendre (organigramme 2).

[19] indexPath et lengthOfPath sont tous les deux initialisés à 0 à leur création, alors au début de la simulation indexPath = lengthOfPath.



Organigramme 2 : Organigramme du déplacement des rats via les véhicules

A chaque pas de temps :

Chaque transporteur consulte le nombre de rats dans son véhicule. S'il contient au moins un rat, il change de forme (dans l'affichage). Sinon il prend sa forme par défaut.

Une classe de type inspecteur est en charge de parcourir toutes les villes, compter le nombre de rats dans chacune et l'inscrire dans une base de données de sortie. Ces données seront exploitées par les statisticiens du projet.

5. Résultats

Pour faire des tests de validation, nous avons travaillé avec la carte raster théorique suivante :

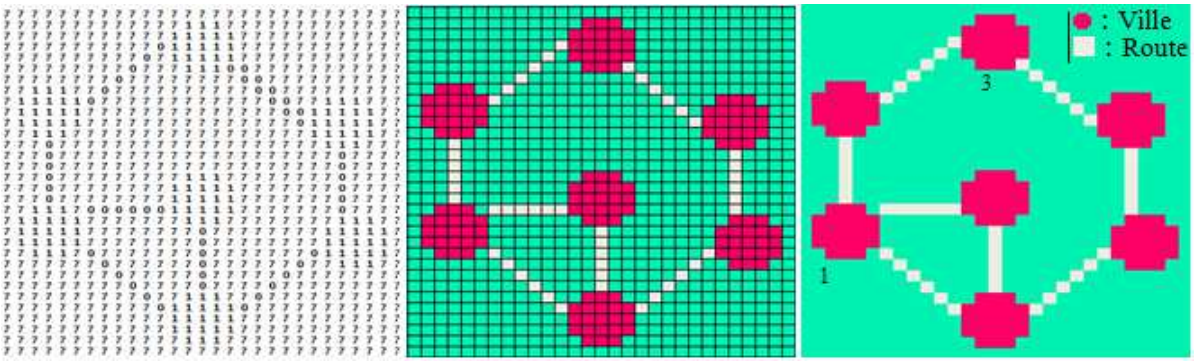


Figure 27 : aspects de la carte théorique utilisée pour la validation

Voici une représentation des sommets du graphe avec leur numéro respectif :

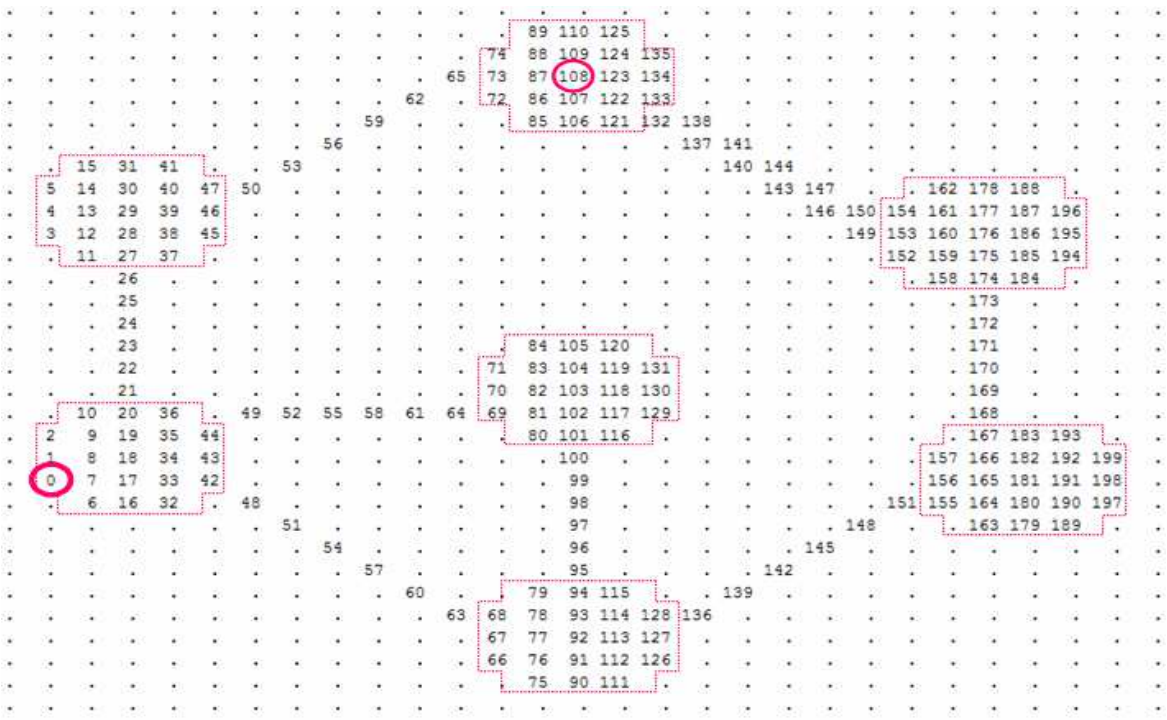


Figure 28 : Graphe des routes

Ce graphe a 200 sommets (numérotés de 0 à199, figure 28) et 1031 arêtes. Sa matrice d'adjacence est donc une matrice carrée d'ordre 200. Elle a 40 000 éléments qui sont tous false sauf 1031 (correspondant au nombre d'arêtes du graphe) (figure 29). Les parties les plus foncées de cette matrice représentent les True et les parties les moins foncées représentent les False.

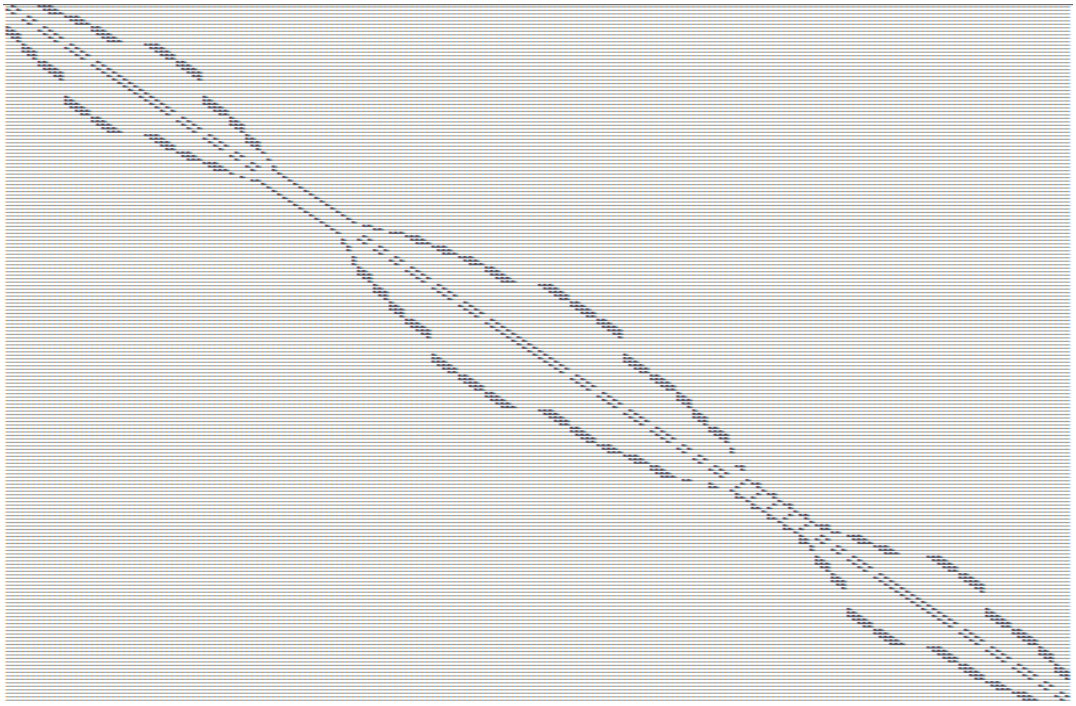


Figure 29 : Matrice d'adjacence du graphe (voir exemple simplifié Figure 11)

On a positionné un véhicule à la cellule numéro 0 de la ville 1. Le véhicule se donne aléatoirement comme destination la cellule numéro 108 de la ville 3 (Figure 28). Il appelle ensuite buildPath() . Cette dernière lui renvoie le chemin suivant :

Dans l'espace continu :
La position de départ : (1200.0, 4600.0),
La destination finale : (3800.0, 8200.0).
Le chemin de longueur 21 est:
0, 1, 2, 10, 21, 22, 23, 24, 25, 26, 27, 38, 46, 50, 53, 56, 59, 62, 65, 72, 86, 108.

Si nous vérifions, nous voyons que un plus court chemin entre le sommet 0 et le sommet 18 est bien celui représenté ci-dessus.

Une cellule a une longueur de 200 mètres. La vitesse de ce véhicule est de 50 mètres par pas de temps. Il faut alors 4 pas de temps pour que ce véhicule puisse parcourir une cellule entière. Ainsi il appelle sa méthode selectNextNode() après chaque 4 pas de temps.

Après 84 (21 x 4) pas de temps, le véhicule arrive à sa position finale (3800,0, 8200,0) en passant par les routes et par le plus court chemin (Figure 30). Il se parque alors et les rats dans un rayon de 30 mètres (dans cette version) autour de lui ont la possibilité d'y monter avant qu'il ne se choisisse une nouvelle destination et ne reparte.

Un autre véhicule avec la même position de départ et la même destination finale mais avec une vitesse de 600 mètres par pas de temps, prendra le même chemin mais n'arrivera à destination qu'après 7 (21 / 3) pas de temps.

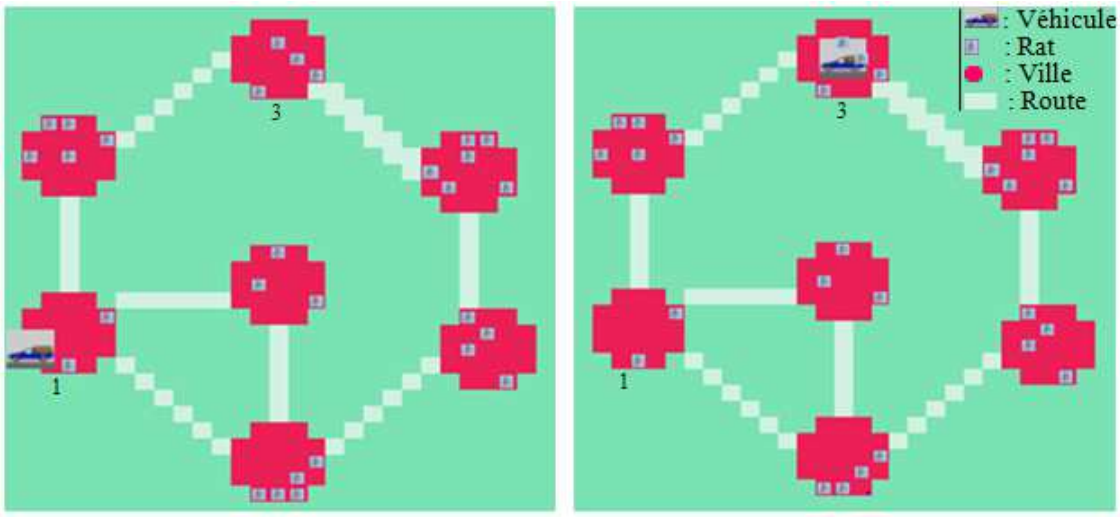


Figure 30 : Véhicule d'une ville à une autre

Nous avons ensuite recommencé avec une carte n'ayant que deux villes et deux routes de longueurs différentes. Nous avons ajouté 50 véhicules dans les deux villes. Et nous avons laissé la simulation tourner pendant plus d'un millier de pas de temps. Tous les véhicules ont pris le plus court chemin (Figure 31).

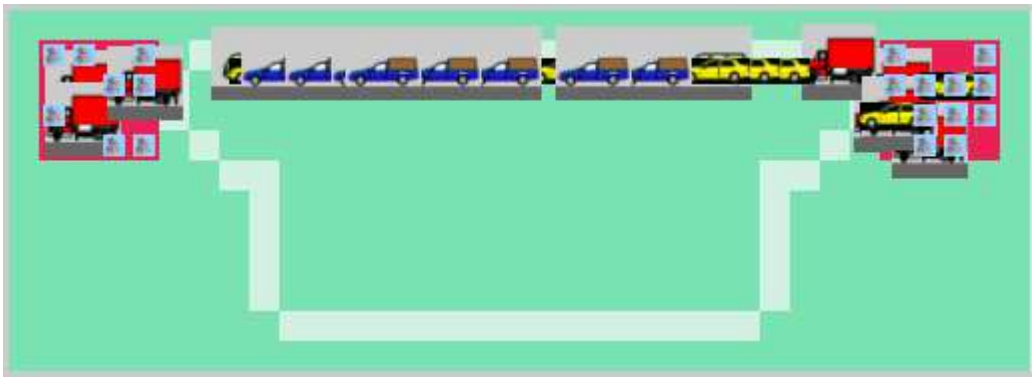


Figure 31 : Carte théorique à 2 villes avec 2 routes de longueur différente

Nous avons enfin vérifié l'algorithme avec les vraies cartes du Sénégal : tous les véhicules restaient sur les routes et empruntaient le plus court chemin pour aller d'une ville à une autre (Figure 32).

Les véhicules transportaient aussi des rats d'une ville à une autre et les bases de données de sortie récupéraient à chaque pas de temps le nombre de rongeurs dans chaque ville.

Les cartes se chargent successivement à chaque fois que la date de chargement programmée arrive.

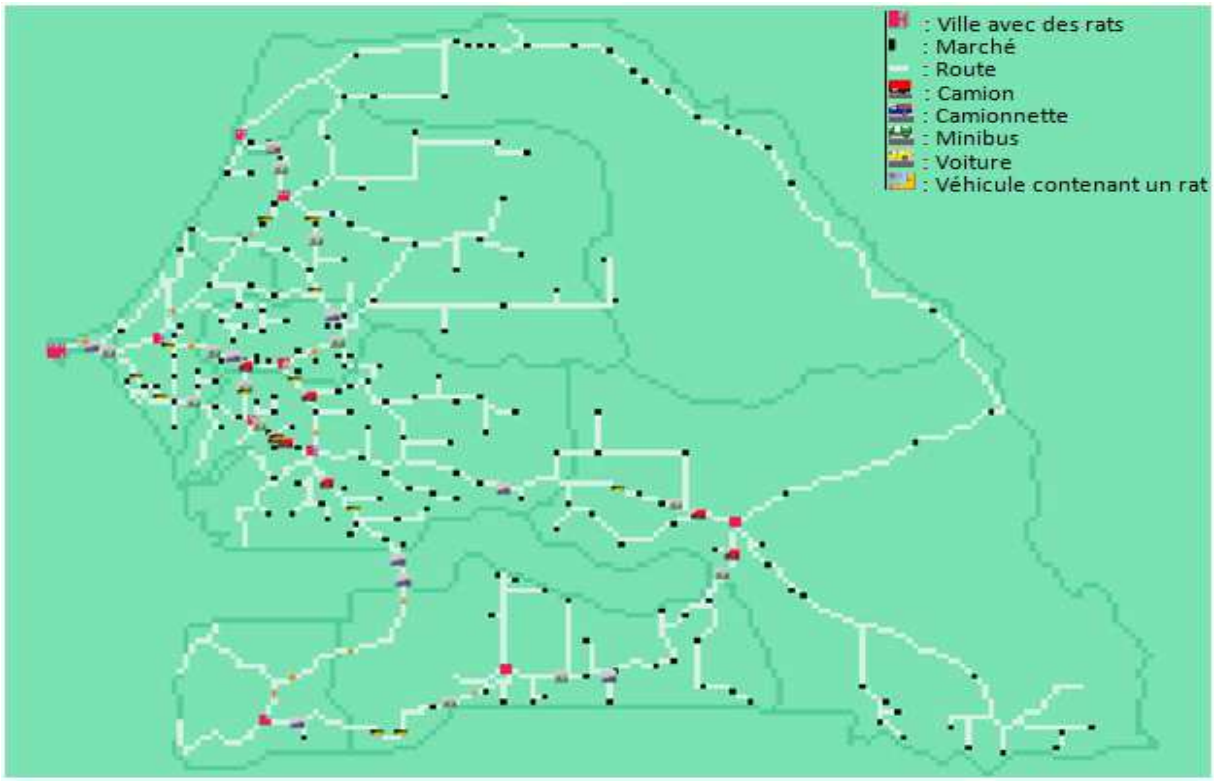


Figure 32 : Image de la simulation

Discussion

L'approche développée et les algorithmes utilisés ont permis d'atteindre les objectifs liés au cahier des charges, c'est-à-dire principalement simuler le trafic routier complet sur l'ensemble du Sénégal avec un ensemble souple d'objets transporteurs humains, leurs véhicules ainsi que des rats qui peuvent monter, descendre et se faire transporter d'une ville à l'autre par des véhicules.

Plusieurs approches alternatives auraient pu être développées pour atteindre cet objectif. On aurait pu ainsi s'intéresser aux projections de type réseau implémentées en natif dans Repast Simphony pour construire le graphe des routes. On a préféré utiliser la projection de type grille qui était déjà opérationnelle dans SimMasto. Cette approche nous a permis de calculer les plus courts chemins en calculant les distances entre les villes sans utiliser les graphes étiquetés. Il aurait aussi été possible d'utiliser les listes chaînées qui prennent moins d'espace mémoire. Notre choix s'est porté plutôt sur l'utilisation d'une matrice d'adjacence qui occupe plus d'espace mais permet de gagner beaucoup plus de temps qu'avec les listes chaînées. Enfin, nous avons utilisé l'algorithme dichotomie pour optimiser la recherche d'éléments sur les tableaux mais il aurait pu être aussi possible de modifier la classe SoilCell en y ajoutant un champ. Au prix d'une perte d'espace mémoire cela aurait permis d'optimiser le temps d'exécution en évitant toute recherche d'élément dans les tableaux. Il a cependant été jugé préférable de limiter au maximum la modification des classes de base du projet (utilisées pour d'autres applications). La procédure dichotomie a permis de rendre cet inconvénient très limité.

Conclusion

En conclusion, nous avons réussi à construire un simulateur multi-agents qui permet la représentation dans le temps des rats qui montent dans et descendent de divers types de véhicules circulant d'une ville à une autre sur les routes du Sénégal en prenant les plus courts chemins tout en prenant en compte l'évolution sur un siècle des villes, des marchés, des routes et des flux de véhicules.

Il apparaît dès à présent que ce simulateur peut être amélioré sur différents aspects. Il serait ainsi intéressant d'affiner les bases de données d'entrée du simulateur et leur ergonomie de façon à rendre accessible l'utilisation du simulateur par les biologistes et géographes. Il serait bon aussi d'améliorer le comptage des rats se trouvant dans les véhicules à chaque pas de temps car seuls les rats se trouvant dans les villes sont comptés actuellement. Sur un autre plan, il serait intéressant de prendre en compte des rats qui se multiplient dans le temps et qui après avoir perçu l'environnement peuvent faire le choix entre monter dans un véhicule, partir vers les champs, les entrepôts de nourriture, etc. Un travail utile concernerait aussi la lecture plus souple des cartes (ajout de nouvelles villes au lieu de relire une carte complète par exemple). Enfin, chaque fois qu'un transporteur demande le plus court chemin entre sa ville et une destination, il est nécessaire de construire le tableau de tous les plus courts chemins. Le temps de calcul pourrait être significativement diminué en stockant ce tableau une fois pour toutes dans un champ spécifique de chaque ville (sommet).

Références

- Baduel Q. (2009) Réalisation d'une chaîne de traitement conduisant à l'intégration de supports spatiaux et de données géoréférencées dans un simulateur multi-agents. *Rapport de stage IUT Montpellier, dépt. Informatique, avril 2009, 45p.*
- Comte A. (2012) Caractérisation des barrières à l'hybridation de deux espèces jumelles de rongeurs africains du genre Mastomys. Etude par simulation multi-agents à partir de deux expériences in situ. *Rapp. Master 2, Recherche - Ecologie-Biodiversité, spécialité Biodiversité Evolution, parcours Génétique et Biodiversité Univ. Montpellier 2, 44p.*
- Ferber J., (1995). Les systèmes multi-agents : vers une intelligence collective. *InterEditions, Paris.*
- Handschumacher P., Herve J.P., Hébrard G., (1992). Des aménagements hydro-agricoles dans la vallée du fleuve Sénégal ou le risque des maladies hydriques en milieu sahélien. *Sciences et changements planétaires/ Sécheresse, 3 (4), 219-226.*
- Holland J.J. (1996). Evolving virus plagues. *Proc. Nat. Acad. Sci. USA, Vol. 93, pp. 545-546.*
- Le Bot C. (2006). Théorie des graphes. *Cours INPG, 2006, 116p., http://blog.christophelebot.fr/wp-content/uploads/2007/03/theorie_graphes.pdf*
- Longueville, J.E. (2011). Modélisation spatialisée de la variabilité génétique de populations de rongeurs sauvages dans un paysage explicite. *Rapport de stage Master 2, parcours EEB, Univ. Lyon I, 46p.*
- Michael Mc et al., (2003). Climate change and human health, Risks and responses. *World Health Organization, 333 p.*
- Montcouquiol G., (2007). Théorie des Graphes. *Cours Univ. Orsay, 178p., http://www.math.u-psud.fr/~montcouq/Enseignements/Apprentis/cours.pdf*
- Ndiaye M.L. (2012). Modélisation de systèmes complexes : Les systèmes multi-agents. *Cours Ecole Supérieure Polytechnique Génie Electrique.*
- Nguyen, Q.T., Bouju, A. et P. Estrailler (2012) Multi-agent architecture with space-time components for simulation of urban transportation systems. *Procedia, Social and Behavioral Sciences, 54, 365-374.*
- Realini A. (2011) Conception et mise en œuvre d'une interface de visualisation d'une plate-forme de simulation individus-centrée - *Rapport DUT, IUT informatique Montpellier, 49p.*
- Ruskin, H.J., Wang, R., 2002. Modelling traffic flow at an urban unsignalized intersection. *In: Proceedings of International Conference on Computational Science, pp. 381–390.*
- Taylor P., Arntzen L., Hayter M., Iles M., Frean J. et Belmain S. (2008). Understanding and managing sanitary risks due to rodent zoonoses in an african city: beyond the Boston model. *Integrative Zoology 3: 38-50.*
- Zamith, M., Leal, R. Kischinhevsky, M. , Clua, E., Brandao, D., Montenegro, A. etLima, E. (2012) A Probabilistic Cellular Automata Model for Highway Traffic Simulation. *Procedia Computer Science 1 (2012) 337–345*

Annexes

A. Algorithme de Dijkstra avec exemple

```
Fonction dijkstra (entree : entier,
                   sortie : entier,
                   chemin[] : pointeur sur un tableau d'entier) renvoie un entier
Début
    matriceAdjacence : matrice booléenne      // matrice d'adjacence du graphe déjà construit
    nombreSommets : entier                    // nombre de sommets du graphe déjà renseigné
    i, m, distanceMin : entiers
  //tableau contenant les sommets marqué (sommet par lequel on passe)
    ensembleMarque[nombreSommets ] : tableau booléen de taille nombreSommets
 //tableau contenant les distances entre chaque sommet et le sommet d'entrée
    distance[nombreSommets] : tableau d'entiers de taille nombreSommets
 //tableau contenant le sommet précédent de chaque sommet
    precedent[nombreSommets] : tableau d'entiers de taille nombreSommets

 //Initialisation
    distance[entree] = 0
    precedent[entree] = entree
    ensembleMarque[entree] = True
    Pour i allant de 0 à nombreSommets par pas de 1
        si matriceAdjacence[entree][i] == True     //s'il y a une arête entre les sommets entree et i
            distance[i] = 1
            precedent[i] = entree
            ensembleMarque[i] = False
        sinon si i != entree                       //s'il n'y a pas une arête entre les sommets entree et i
            distance[i] = nombreSommets +10
            precedent[i] = i
            ensembleMarque[i] = False
        Fin si
    Fin pour
 //Fin initialiation (seul le sommet d'entrée est marqué à l'initialisation)

//Recherche de tous les plus courts chemins entre le sommet d'entrée et tous les autres
    Tant que True
        m = -1          // pour contenir le sommet non marqué suivant.
        distanceMin = nombreSommets + 11     // pour maximiser la distance minimale

      /*Phase de sélection: on cherche parmi les sommets non marqués celui ayant de plus court
                          chemin avec le sommet d'entree et on le sélectionne dans m */
        Pour i allant de 0 à nombreSommets par pas de 1
            Si ensembleMarque[i] == False et distance[i] < distanceMin
                distanceMin = distance[i]
                m = i          // m sera le sommet suivant avec le plus court chemin
            Fin Si
        Fin Pour
     //Test d'arrêt de la boucle tant que : s'il n'y a plus de point m ou s'il n'y a pas de chemin entre entree et m.
        Si m == -1 ou distance[m] == nombreSommets + 10)
            arrêt de la boucle tant que
        Fin Si
```

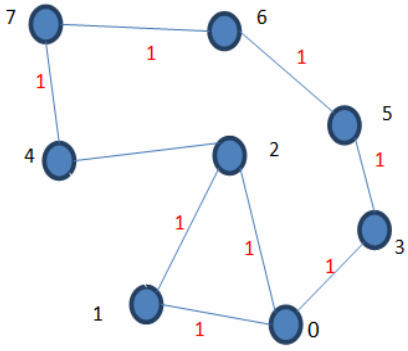
```
        //Marquage du sommet numéro m
            ensembleMarque[m] = True

        /* Phase de mise à jour des tableaux : Les distances des sommets non marqués et adjacents au
                                                sommet m qui vont être mis à jour.*/
            Pour i allant de 0 à nombreSommets par pas de 1
            Si matriceAdjacence[m][i] == True et ensembleMarque[i] == False et
               distance[m] + 1 < distance[i]  /*s'il ya une arête entre m et le sommet i est non marqué et
                     ( (distance (entree, m)) + (poid d'arête (m, i)) ) < (distance qui est déjà dans distance[i]) */
                  distance[i] = distance[m] + 1 // distance(entree, i) = distance[m] +1
                  precedent [i] = m   // le sommet i est maintenant précédé par le sommet m
            Fin Si
        Fin Pour
    // Fin recherche de tous les plus courts chemins

    // Creation du chemin
        renvoyer faireChemin(chemin, precedent, sortie)
Fin

Fonction faireChemin( chemin[] : pointeur sur un tableau d'entier,
                      precedent [] : pointeur sur un tableau d'entier,
                      i : entier ) renvoie un entier
Début
        Si precedent[i] == i
            chemin[0] = i
            renvoyer  0
        Fin Si
        l = 1 + faireChemin(chemin, precedent, precedent[i])
        chimin[l] = i
Fin
```

Déroulons l'algorithme avec l'exemple suivant (figure 33) : le sommet d'entrée est 0 et le sommet de sortie est 7.



	0	1	2	3	4	5	6	7
0	T	T	T	T	F	F	F	F
1	T	T	T	F	F	F	F	F
2	T	T	T	F	T	F	F	F
3	T	F	F	T	F	T	F	F
4	F	F	T	F	T	F	F	T
5	F	F	F	T	F	T	T	F
6	F	F	F	F	F	T	T	T
7	F	F	F	F	T	F	T	T

Figure 33 : Graphe et sa matrice d'adjacence.

Si on a un **graphe** et sa **matrice d'adjacence** on peut utiliser Dijkstra pour construire le plus court chemin entre deux sommets du graphe.

entree = 0

sortie = 7

chemin :

-1	-1	-1	-1	-1	-1	-1	-1

dijkstra(0, 7, chemin) //appel de la fonction dijkstra

Début du déroulement de dijkstra

Les trois tableaux sont créés mais ne contiennent rien avant l'initialisation (tableau).

Numéros	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Distance								
precedent								
ensembleMarque								

Résultat après l'initialisation.

Numéros	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Distance	0	1	1	1	18	18	18	18
Precedent	0	0	0	0	4	5	6	7
ensembleMarque	T	F	F	F	F	F	F	F

Commentaire et lecture des tableaux :

Soit **i** le numéro d'un sommet du graphe.
distance[**i**] = **a** signifie que **a** est plus petite distance entre le sommet d'entrée **0** et le sommet **i**
precedent[**i**] = **b** signifie que le sommet **i** est précédé par le sommet **b** dans le plus court chemin entre le sommet d'entrée **0** et le sommet **i**
ensembleMarque[**i**] = T signifie que le sommet i est marqué.
ensembleMarque[**i**] = F signifie que le sommet i n'est pas encore marqué.

Après l'étape d'initialisation de l'algorithme seul le sommet d'entrée **0** est marqué (T), tous les autres ne le sont pas (F). On a donc fini de travailler avec le sommet **0**. On connait alors tous les sommets avec lesquels il a une arête (**0**, **1**, **2** et **3**) et la distance entre lui et ces sommets :

Le sommet 0 précède les sommets **0**, **1**, **2** et **3** :
precedent[**0**]= 0, precedent[**0**]= 0, precedent[**0**]= 0 et precedent[**0**] = 0.
Les distances entre le sommet **0** et les sommets **0**, **1**, **2** et **3** sont :
distance[**0**] = 0, distance[**1**] = 1, distance[**2**] = 1 et distance[**3**] = 1.

Les distances entre le sommet **0** et les sommets **4**, **5**, **6** et **7** sont temporairement dites infinies (ici on a donné arbitrairement 18 : nombre de sommets + 10) et chacun d'eux est temporairement précédé par lui même. En effet ces distances ne sont pas encore connues et les sommets qui précédent ces sommets ne sont pas encore visités (marqués).

Résultats après chaque itération de la partie recherche de tous les plus courts chemins entre le sommet d'entrée et tous les autres

Les zones surlignées en gris sont celles qui ont changé dans l'itération en cours.
Distance(i, j) signifie la distance entre deux sommets adjacents de numéro i et j.

1ère entrée dans la boucle tant que :

Numéros	0	1	2	3	4	5	6	7
Distance	0	1	1	1	18	18	18	18
Precedent	0	0	0	0	4	5	6	7

ensembleMarque	T	T	F	F	F	F	F	F

Commentaire :

Le sommet 1 est marqué, il est suivi par le sommet 2 non marqué mais la distance[2] est plus petit que distance[1] + distance(1, 2). Donc pas de mise à jour sur les tableaux distance et precedent.

2ère entrée dans la boucle tant que

Numéros	0	1	2	3	4	5	6	7
Distance	0	1	1	1	2	18	18	18
Precedent	0	0	0	0	2	5	6	7
ensembleMarque	T	T	T	F	F	F	F	F

Commentaire :

Le sommet 2 est marqué. Parmi les sommets reliés à 2, seul 4 est non marqué. Et puisque la distance[4] était 18, ce qui est supérieur à la distance[2] + distance(2, 4), alors on met à jour la distance[4] par la distance[2] + distance(2, 4), et 4 est maintenant précédé par 2.

3ère entrée dans la boucle tant que

Numéros	0	1	2	3	4	5	6	7
Distance	0	1	1	1	2	2	18	18
Precedent	0	0	0	0	2	3	6	7
ensembleMarque	T	T	T	T	F	F	F	F

4ère entrée dans la boucle tant que

Numéros	0	1	2	3	4	5	6	7
Distance	0	1	1	1	2	2	18	3
Precedent	0	0	0	0	2	3	6	4
ensembleMarque	T	T	T	T	T	F	F	F

5ère entrée dans la boucle tant que

Numéros	0	1	2	3	4	5	6	7
Distance	0	1	1	1	2	2	3	3
Precedent	0	0	0	0	2	3	5	4
ensembleMarque	T	T	T	T	T	T	F	F

6ère entrée dans la boucle tant que

Numéros	0	1	2	3	4	5	6	7
Distance	0	1	1	1	2	2	3	3
Precedent	0	0	0	0	2	3	5	4
ensembleMarque	T	T	T	T	T	T	T	F

7ère entrée dans la boucle tant que

Numéros	0	1	2	3	4	5	6	7
Distance	0	1	1	1	2	2	3	3
precedent	0	0	0	0	2	3	5	4
ensembleMarque	T	T	T	T	T	T	T	T

8ère entrée dans la boucle tant que

Il n'y a plus de sommets non marqué. Donc une condition de sortie de la boucle tant que est trouvée. Alors fin de la boucle tant que.

Fin recherche de tous les plus courts chemins

On a ainsi un tableau permettant d'avoir avec leur distance, tous les plus courts chemins entre le sommet 0 et tous les autres du graphe. Nous remarquons que le tableau chemin et l'entier sortie reçus en paramètre ne sont pas encore utilisé. En effet Dijkstra permet de construire des tableaux permettant d'avoir avec leur distance, tous les plus courts chemins entre un sommet d'entrée et tous les autres du graphe :

Un plus court chemin entre 0 et 7 est : (0, 2, 4, 7). En effet dans le tableau **precedent**, 7 est précédé par 4 qui est précédé par 2 qui est précédé par 0 (tableau). La distance est 3 (il y a trois arêtes entre les sommets 0 et 7).

Numéros	0	1	2	3	4	5	6	7
precedent	0	0	0	0	2	3	5	4
Distance	0	1	1	1	2	2	3	3
ensembleMarque	T	T	T	T	T	T	T	T

Le plus court chemin entre 0 et 6 est : (0, 3, 5, 6). En effet dans le tableau **precedent**, 6 est précédé par 5 qui est précédé par 3 qui est précédé par 0. Et la distance est 3.

Il suffit d'avoir le tableau **precedent** et le sommet de sortie pour récupérer le plus court chemin entre le sommet 0 et ce sommet. Sans oublier un tableau pour contenir le chemin.

Apelle de la fonction faireChemin dans Dijkstra

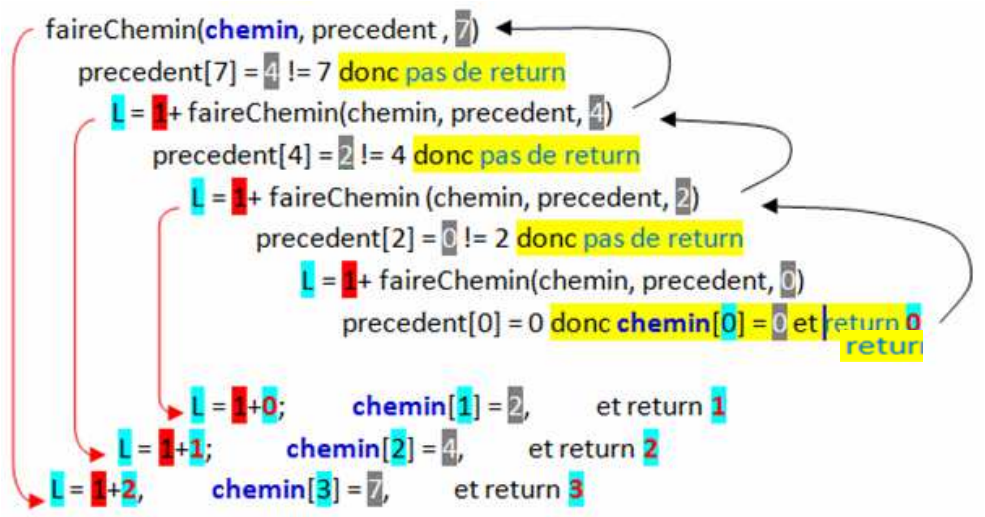
renvoyer **faireChemin(**chemin, precedent, sortie**)**

Fin du déroulement de dijkstra

Début du déroulement de faireChemin()

Numéros	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
precedent	0	0	0	0	2	3	5	4

L'algorithme est récursive, c'est-à-dire que l'on rappelle la fonction dans elle même jusqu'à ce que le précédent d'un sommet soit le sommet d'entrée 0.



Cette fonction construit directement le tableau **chemin** en mémoire et renvoie sa longueur.

Fin du déroulement de faireChemin()

Résultat :
chemin :

0	2	4	7	-1	-1	-1	-1

Le plus court chemin pour aller du sommet 0 au sommet 7 est (0, 2, 4, 7). Et la longueur est 3.

B. Liens utiles

- Site de Repast Simphony : http://www.repast.sourceforge.net/
  - Forum utilisateurs : http://sourceforge.net/mailarchive/forum.php?forum_name=repast-interest
  - Tutoriaux pour Repast Simphony : http://repast.sourceforge.net/docs/tutorial/SIM/
- SimMasto : http://www.mpl.ird.fr/ci/masto/index.htm
- Jean Le Fur : http://lefur.jean.free.fr/
  - Travaux des étudiants : http://lefur.jean.free.fr/1jean/liste_de_publications.htm#Chercheurs associés au projet Mo
- Théorie des graphes
  - http://www.extpdf.com/theori-de-grafhe-pdf.html
  - http://blog.christophelebot.fr/wp-content/uploads/2007/03/theorie_graphes.pdf
  - http://www.math.u-psud.fr/~montcouq/Enseignements/Apprentis/cours.pdf

C. Glossaire

- **Cluster** : Le cluster est un système informatique composé d'unités de calcul (microprocesseurs, cœurs, unités centrales) autonomes qui sont reliées entre elles à l'aide d'un réseau de communication.
- **Dimensions** : Une image peut être considérée comme une matrice de pixel. les dimensions sont le nombre de lignes et de colonnes de cette matrice.
- **Eclipse** : est un environnement de développement intégré libre, extensible, universel et polyvalent.
- **Généricité** : Comme les collections, un Treeset permet de contenir des données de type non homogènes. La généricité permet de lui donner un type générique pour lui indiquer qu'il ne peut contenir que des objets d'un certain type.
- **Getters** : accesseur public à la valeur d'un champ privé ou protégé.
- **GUI (Graphical User Interface)** : dispositif de dialogue homme-machine dans lequel les objets à manipuler sont dessinés sous forme de pictogrammes à l'écran, que l'usager peut utiliser en imitant la manipulation physique de ces objets avec un dispositif de pointage, le plus souvent une souris
- **Pixel** : est l'unité de base permettant de mesurer la définition d'une image numérique matricielle.
- **Raster** : une image graphique raster, ou bitmap, est une matrice de points représentant une structure de données sous forme de grille de pixel souvent rectangulaire, ou des points de couleur affichables par l'intermédiaire d'un moniteur, papier ou autre support d'affichage. Les images raster sont stockées dans des fichiers image avec différents formats.
- **SIG (Système d'Information Géographique)** : système d'information permettant de créer, d'organiser et de présenter des données spatialement référencées (géoréférencées).
- **TreeSet** : est un tableau de taille variable qui peut contenir des données non homogène et qui trie automatiquement les données qu'il contient.
  Type générique : voir généricité.