Optimisation des plus courts chemins construits par l'algorithme de Dijkstra pour un simulateur multi-agents spatialisé

Pape Adama Mboup[1,2,*], Pascal Handschumacher[3], Karim Konaté[1] et Jean Le Fur[4]
[1]Département de mathématiques et d'informatique, UCAD/18522 Dakar Sénégal,
[2]IRD, UMR 022 BIOPASS, Campus de Bel-Air, BP 1386, Dakar, CP 18524, Sénégal
[3]IRD, UMR 912 SESSTIM INSERM - IRD - U2, Faculté de géographie UdS, 3, rue de l'Argonne F-67000 Strasbourg, France
[4]IRD, UMR 022 CBGP, Campus international de Baillarguet, CS 30016, 34988 Montferrier-sur-Lez cedex, France
pamboup@hotmail.fr, p.handschumacher@unistra.fr, kkonate911@yahoo.fr, jean.lefur@ird.fr

***Abstract*—La modélisation relative au déplacement d'agents sur une topologie de type graphe par les systèmes multi-agents s'avère une approche fructueuse. Beaucoup de modèles liés aux déplacements d'agents utilisent l'algorithme de Dijkstra pour construire exactement les plus courts chemins. Cependant un problème majeur de ces modèles est la fréquence à laquelle les nombreux agents, et durant toute la simulation, utilisent Dijkstra pour construire leurs plus courts chemins entre les positions où ils se trouvent et les positions où ils veulent se rendre. Cette utilisation massive de l'algorithme nécessite un grand temps de calcul. Dans cet article, nous proposons une optimisation spatiale de la représentation informatique d'un graphe (matrice d'adjacence, liste d'adjacence) fondée sur un stockage optimisé de tout plus court chemin une fois construit. L'algorithme évite aux agents de se reconstruire des chemins déjà construits et détruits, et donc diminue considérablement le temps de calcul dû à la construction de plus court chemins.**

***Mots-clés*—Optimisation, optimisation spatiale, matrice d'adjacence, Dijkstra, plus courts chemins, tableau des précédents, matrice des précédents.**

I. Introduction

Le développement de la société sénégalaise depuis un siècle a conduit à des changements spectaculaires et multiples concernant tant l'usage des terres que la répartition des populations humaines, la dynamique des villes ou l'accélération des flux de population et de biens [2]. Ces changements s'accompagnent le plus souvent de conséquences pour la santé publique avec un développement concomitant des vecteurs de risques épidémiologiques. Parmi ceux-ci, le rat noir (Rattus rattus) représente un enjeu fort en termes de santé publique car il est hôte de nombreuses maladies transmissibles à l'homme telles que la leptospirose, le typhus murin, les borrélioses ou divers hantavirus [4]. Plus généralement, sa proximité à l'homme en fait un élément majeur de la diffusion et de la propagation de risques sanitaires.

Dans le cas de la colonisation du rat noir au Sénégal, l'un des obstacles majeurs à surmonter tient au fait que les problèmes à résoudre résultent de processus multiples qui opèrent et doivent être saisis à des échelles de temps, d'espace et d'analyse différents [1] [8].

Dans l'effort pour décrypter ces mécanismes complexes, nous avons construit un modèle permettant la simulation de l'histoire des transports humains au Sénégal (en utilisant les graphes) sur un siècle et de leur influence sur la colonisation du rat noir [11]. La construction des chemins pour les transporteurs circulant sur divers types de voies de transports (fleuves, rail, routes) sur l'ensemble du territoire sénégalais et sur un siècle impliqué une quantité énorme de chemins à construire dans le temps et dans l'espace. Et donc nécessite un grand temps de calcul qu'il faut impérativement diminuer.

Dans cet article, nous présentons comment nous avons passé de l'optimisation spatiale de la représentation informatique d'un graphe (matrice d'adjacence, liste d'adjacence) pour avoir plus d'espace pour un stockage encore optimisé de tout plus court chemin une fois construit. Afin d'évite aux agents transporteur de se reconstruire des chemins déjà construits au paravent, et donc de diminuer considérablement le temps de calcul dû à la construction de plus court chemin. Ainsi nous n'avons plus besoin de faire recourt à d'autre algorithmes comme A* [9], plus rapide que Dijkstra [3], mais ne garantissant pas exactement le plus court chemin.

II. Matériels et méthodes

Le simulateur est développé à partir de la plate-forme pluridisciplinaire de modélisation et de simulation de populations de rongeurs par les systèmes multi-agents SimMasto [5]. Développée en java sous Repast Simphony [7] et fondée sur une architecture trois tiers (data-business-presentation) [10], SimMasto est conçue robuste et flexible, ce qui a permis d'y greffer facilement un module permettant la gestion de déplacement d'agents sur une topologie de type graphe [11]. Dans cette topologie nous avons utilisons des graphes non-pondérés et non-orientés dont la seule information qui nous intéresse sur la représentation informatique d'un graphe est l'adjacence entre deux nœuds. Ainsi nous avons aisément modifié les structures de données classiques de représentation informatique d'un graphe.

Un graphe G est un couple (V, E) où $V = \{v_1, v_2, \ldots, v_n\}$ est un ensemble fini d'objets et $E = \{e_1, e_2, \ldots, e_m\}$ est sous-ensemble de VxV. Les éléments de V sont appelés les sommets ou nœuds du graphe et les éléments de E sont appelés les arêtes du graphe. Une arête relit deux nœuds adjacents. Pour faciliter la compréhension du lecteur, prenons le graphe G suivant (fig. 1) comme exemple pour le reste de l'article.

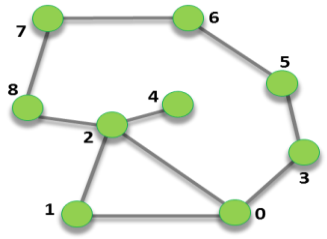


Fig. 1. *G* : exemple de graphe pour illustration (représentation graphique).
V = {0, 1, 2, 3, 4, 5, 6, 7, 8}, *E* = {(0,1), (0,3), (0,2), (1,2), (2,4), …, (8,7)}

La représentation informatique d'un graphe est généralement faite par une matrice ou une liste d'adjacence.

A. *Matrice et liste d'adjacence classique et leur inconvénient*

Une matrice d'adjacence (tableau I) du graphe G (V, E), non-pondéré et non-orienté, est la matrice $M(G) \in M_n(\mathbb{R})$ dont les coefficients $m_{i,j}$ sont définis par :

$$m_{i,j} = \begin{cases} 1 \text{ si } (v_i, v_j) \in E \text{ ou } (v_j, v_i) \in E \\ 0 \text{ si } (v_i, v_j) \notin E \text{ et } (v_j, v_i) \notin E \end{cases}$$

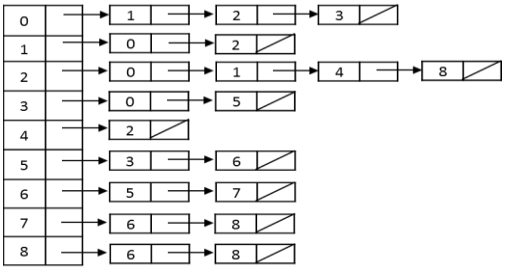
TABLEAU I
MATRICE D'ADJACENCE CLASSIQUE DE G

	0	1	2	3	4	5	6	7	8
0	0	1	1	1	0	0	0	0	0
1	1	0	1	0	0	0	0	0	0
2	1	1	0	0	1	0	0	0	1
3	1	0	0	0	0	1	0	0	0
4	0	0	1	0	0	0	0	0	0
5	0	0	0	1	0	0	1	0	0
6	0	0	0	0	0	1	0	1	0
7	0	0	0	0	0	0	1	0	1
8	0	0	1	0	0	0	0	1	0

Même si l'accès à un élément est très rapide avec les matrices classiques (complexité temporelle constante O(1)), l'inconvénient avec cette représentation est la complexité spatiale. L'espace mémoire utilisé est de l'ordre de N^2 pour un graphe d'ordre N, même lorsqu'il représente beaucoup moins de N^2 arêtes. Cette complexité spatiale peut conduire à l'utilisation de la représentation par les listes d'adjacences.

Une liste d'adjacence est un tableau de pointeurs ou la liste d'adjacence relative au sommet i est la liste dans un ordre arbitraire des sommets adjacents à i. Chaque élément de la liste est constitué du numéro du sommet et d'un pointeur sur le suivant. Le tableau II est une liste d'adjacence du graphe *G*.

TABLEAU II
LISTE D'ADJACENCE CLASSIQUE DE G



Même si avec les listes d'adjacences seules les arêtes entre les nœuds sont représentées, l'inconvénient avec cette représentation est que chaque arêtes est représentée deux fois (complexité spatiale) et que l'accès à un élément est généralement plus lent à cause des listes chainées (complexité temporelle).

B. *Optimisation spatiale de la représentation informatique d'un graphe*

Notre objectifs était d'avoir une structure de données permettant une représentation informatique d'un graphe avec une rapidité d'accès aux éléments égale à celle de la matrice d'adjacence, sans représenter les non-adjacences comme dans la liste d'adjacence et en éliminant les doublons d'informations se trouvant dans la matrice et la liste d'adjacence. Pour cela nous avons utilisé un tableau associatif dont les clés sont des numéros de nœuds, et que chaque clé est associée à un tableau de taille variable et dont le test pour savoir s'il contient un élément est de complexité O(1). Ainsi nous avons associé deux structures de données que nous appelons respectivement HashMap et HashSet (comme en Java) : (*i*) HashMap est un tableau associatif qui référence ses objets (valeur) par clé. Elle permet l'ajout (méthode put()) et la récupération (méthode get()) d'une valeur par une clé avec une complexité temporelle constante O(1). (*ii*) HashSet est tableau qui ne permet pas l'accès à un élément par index mais permet d'ajouter (méthode add()) et de vérifier la présence d'un élément en son sein (méthode contains()) avec une complexité temporelle constante O(1). En associant ces deux structures de données, nous avons écrit l'algorithme 1 ci-dessous permettant une construction optimisée de la représentation informatique d'un graphe que nous appelons matrice d'adjacence optimisée (matAdjOpt).

Principe de algorithme 1 : Soient nœud1 et nœud2 deux nœuds adjacents, de numéro respectif n1 et n2, avec n1 inférieur à n2. Alors dans la matAdjOpt n1 est une clé associée à un tableau contenant n2, et pas la réciproque du fait que la même information n'est stockée qu'une seule fois. De ce fait quand nous voulons tester l'adjacence entre deux nœuds, il faudra chercher dans matAdjOpt l'adjacence entre celui qui a le plus petit numéro et celui qui a le plus grand numéro.

```
Algorithme 1 : ConstructionDEmatAdjOpt
Variable :
    nodeList : un tableau simple contenant par ordre tous les nœuds du
    graphe. La position d'un nœud dans ce tableau correspond son numéro
    matAdjOpt: HashMap <Entier, HashSet<Nœud>>
    alreadyTreatedNode : HashSet <Nœud>
Début
    Pour numéro allant de 0 à la taille de nodeList faire
        oneNode = nodeList[numéro]
        matAdjOpt.put(numéro, new HashSet<Noeud>())
        alreadyTreatedNode.add(oneNode);
            Pour chaque noeud oneNeighboringNode dans le voisinage de
            oneNode faire
                // pour ne pas stocker deux fois la même information
                si(alreadyTreatedNode.contains(oneNeighboringNode))
                    continue
                Fin si
                matAdjOpt.get(numéro).add(oneNeighboringNode);
            Fin Pour
        // si le tableau associé à une clé est vide alors c'est supprimé
        si(edgesMatrix.get(numéro).size() == 0)
            edgesMatrix.remove(numéro)
        Fin si
    Fin Pour
Fin
```

Le tableau II représente la matrice d'adjacence optimisée du graphe G construit à partir de l'algorithme 1.

TABLEAU III
MATRICE D'ADJACENCE OPTIMISEE DE G

0	1	2	3	5	6	7
{1, 2, 3}	{2}	{4,8}	{5}	{6}	{7}	{8}

La construction de la matrice d'adjacence se fait une seule fois. Sauf si le graphe est mis à jour. Mais son utilisation (test d'adjacence entre deux nœuds) peut être très fréquente. Dans cette représentation optimisée, toutes les propriétés de la matrice d'adjacence classique ne sont pas conservées. Mais ce qui nous intéresse ici c'est de pouvoir tester l'adjacence entre deux nœuds. Ce que l'algorithme 2 fait avec une complexité temporelle constante O(1).

```
Algorithme 2: AdjacenceEntreDeuxNœuds(noeud1 : Nœud, noeud2 : Nœud)
Variables :
    numéroMin, numéroMax : entiers
Début
    numéroMin = min (noeud1.numéro, noeud2.numéro)
    numéroMax= max (noeud1.numéro, noeud2.numéro)
    renvoyer edgesMatrix.get(numéroMin).contains(numéroMax)
Fin
```

Tout en conservant le même ordre de grandeur de la rapidité du test d'adjacence entre deux nœuds qu'avec une matrice d'adjacence classique (seule la permutation les différencie), cette manière de faire nous permet de gagner, plus de 50% en espace (71,88 % avec le graphe *G*) par rapport à la matrice d'adjacence classique. Ce pourcentage dépend de la densité connexion entre les nœuds.

C. Stockage de la matrice des précédent

L'algorithme de Dijkstra utilise une représentation informatique d'un graphe (matrice d'adjacence optimisée) pour construire pour un sommet de départ donné, un tableau contenant le sommet précédant chaque sommet du graphe en partant du sommet de départ et en prenant le plus court chemin. Nous appelons ce tableau *tabDesPrécedents* ; *matDesPrécedents* est le tableau contenant des *tabDesPrécedents*. Le tableau IV représente le *tabDesPrécedents* pour le nœud de départ 0 avec le graphe *G*. Ce tableau nous indique par exemple qu'en quittant le nœud 0 et en prenant le plus court chemin, le nœud 7 est précédé par le nœud 8 (voir fig. 1).

TABLEAU IV
TABDESPRECEDENTS POUR LE NŒUD DE DEPART 0 EN PRENANT LES PLUS COURTS CHEMINS

nœuds	0	1	2	3	4	5	6	7	8
nœud précédent	0	0	0	0	2	3	5	8	2

A partir de ce *tabDesPrécedents* il est possible de construire le plus court chemin entre 0 et tous les autres nœuds du graphe et réciproquement puisque le graphe est non-orienté. Par exemple le plus court chemin entre 0 et 7 c'est le plus court chemin entre 7 et 0 parcourut dans l'autre sens.

L'algorithme 3 construit à partir de *matDesPrécedents* le plus court chemin entre deux nœuds avec une complexité O(L) avec L la longueur du chemin.

```
Algorithme 3 : CheminFromMatDesPrécedents (entree: entier, sortie: entier)
Variables :
    chemin : liste d'entiers, exactement comme ArrayList en Java [12].
Début
    chemin.add(entree)
    chemin.add(sortie)
    tabDesPrécedents = matDesPrécedents.get(entree)
    Tant que (entrée != sortie)
        sortie = tabDesPrécedents.get(sortie)
        chemin.add(1, sortie)
    Fin tant que
Fin
```

TABLEAU V
MATDESPRECEDENTS DU GRAPHE G

nœuds		0	1	2	3	4	5	6	7	8
nœud précédent	0	0	0	0	0	2	3	5	8	2
	1	1	1	1	0	2	3	5	8	2
	2	2	2	2	0	2	3	7	8	2
	3	3	0	0	3	2	3	5	6	2
	4	2	2	4	0	4	3	7	8	2
	5	3	0	0	5	2	5	5	6	7
	6	3	0	8	5	2	6	6	6	7
	7	2	2	8	5	2	6	7	7	7
	8	2	2	8	0	2	6	7	8	8

Quand un utilisateur (agent) veut se construire un plus court chemin entre un nœud de départ *noeud1* et un nœud d'arrivé *noeud2*, Dijkstra lui construit le *tabDesPrécedents* pour *noeud1* pour ensuite lui rendre le chemin cherché. Au lieu de jeter ce *tabDesPrécedents* temporairement construit, l'idée est de le sauvegarder dans *matDesPrécedents* (tableau V). De ce fait quand un autre utilisateur voudra se construire un chemin dont *noeud1* est le nœud de départ ou d'arrivé, qu'il ne réinvente pas la roue en reconstruisant le même *tabDesPrécedents* ou en construisant un autre. Il ne fera que construire son chemin à partir d'un *tabDesPrécedents* déjà dans *matDesPrécedents*. Pour cela il y a deux manières de faire : construire toute la *matDesPrécedents* justes après la construction du graphe ou au fur et à mesure que les utilisateurs appellent Dijkstra.

1) Construction et stockage de la matDesPrécedents au fil du temps

Une des manières de faire est de ne pas stocker à priori les *tabDesPrécedents*, mais d'attendre qu'un utilisateur en construit un pour le sauvegardons pour tous. Ainsi un *tabDesPrécedents* qui est construit est construit tous et un *tabDesPrécedents* qui n'est jamais utilisé n'est jamais construit. Ce qui fait que l'espace mémoire gagné avec l'optimisation de la matrice d'adjacence qui peut stocker plus de la moitié de l'espace que peut prendre *matDesPrécedents* complète construit, pourrai contenir les seuls *tabDesPrécedents* utilisés (voir résultats). Il n'est pas possible de faire une optimisation spatiale de la *matDesPrécedents* du fait qu'il n'ait pas à priori toute l'information de sa partie triangulaire supérieure ou inférieure (voir le point 2 suivant).

2) Stockage de matDesPrécedents dès le début

Il est possible, juste après la construction du graphe, de construire complètement la *matDesPrécedents* avec une complexité temporelle égale à N fois la complexité de l'algorithme de Dijkstra soit $O(N^3)$ avec N le nombre de nœuds du graphe. Ainsi Dijkstra ne sera utilisé pour le reste de la simulation

(sauf si on met à jour le graphe). Dans ce cas il est possible de faire une optimisation spatiale de *matDesPrécedents* en ne stockant que la partie triangulaire inférieure ou supérieure (tableau V). En effet toute l'information pour construire un plus court chemin entre deux nœuds est contenue dans cette partie supérieure ou inférieure totalement construit.

a. Construction et stockage de la matDesPrécedents optimisée

Supposons que nous choisissons de stocker la partie triangulaire supérieure de la matrice. Alors cette partie ne contient donc que les nœuds précédents pour les chemins quittant un nœud de numéro plus petit vers un nœud de numéro plus grand. Nous pouvons utiliser la même structure de donnée que pour la matrice d'adjacence optimisée pour stocker numéros de nœuds associés à des tableaux de taille variable.

Après que Dijkstra ait construit le *tabDesPrécedents* pour un nœud de départ *nœudDépart*, il faut le tronquer et ne garder que la partie dont les indices sont supérieurs au numéro de *nœudDépart* avec une fonction appelée subList prenant deux paramètres que sont l'indice de début et l'indice de fin de la portion du tableau à garder (voir algorithme 4).

```
Algorithme 4 : ConstructionDEmatDesPrécedentsOptimisée
Variables :
    nœudDépart : entier
Début
    Pour nœudDépart allant de 0 à taille de nodeList faire
        tabDesPrécedents = Dijkstra (nœudDépart)
        matDesPrécedents.put ( nœudDépart,
            tabDesPrécedents.subList (nœudDépart + 1, taille de tabDes-
            Précedents))
    Fin Pour
Fin
```

Pour le graphe *G*, l'algorithme 4 donne la matrice des précédents optimisée suivante (tableau VI).

TABLEAU VI
MATDESPRECEDENTS OPTIMISEE DU GRAPHE G

nœuds		1	2	3	4	5	6	7	8
nœud précédent	0	0	0	0	2	3	5	8	2
	1	1	0	2	3	5	8	2	
	2	0	2	3	7	8	2		
	3	2	3	5	6	2			
	4	3	7	8	2				
	5	5	6	7					
	6	6	7						
	7	7							

Avec cette matrice, pour un nœud de départ de numéro nDépart, le nœud précédant un nœud d'arrivé de numéro nArrivé (nDépart inférieur à nArrivé), est directement retrouvable par l'instruction matDesPrécedents.get(nDépart). get(nArrivé - nDépart). Exemple si nDépart = 3 et nArrivé = 8 alors matDesPrécedents.get(3).get(8 - 3) == matDesPrécedents.get(3).get(5) == 2.

L'espace occupé par la *matDesPrécedents* optimisée est de N(N-1)/2 avec N le nombre de nœuds du graphe. C'est de l'ordre de la moitié de la matrice d'adjacence classique.

b. Construction d'un plus court chemin à partir de la matDesPrécedents optimisée

Pour mieux expliquer le principe, essayons de retrouver par exemple le plus court chemin (*chemin*) entre les nœuds 3 et 8. Au début, *chemin* ne contient que 3 et 8 (*chemin* = {3, 8}) et il faut insérer les nœuds intermédiaires dans *chemin*. Puisque 3 est inférieur à 8, il est possible de récupérer le nœud précédant le 8 par *matDesPrécedents*.get(3).get(8 - 3), ce qui donne 2. *Chemin* devient {3, 2, 8}. Il faut à présent chercher le chemin entre 3 et 2 ou bien entre 2 et 3 (2 < 3) ; Le précédent de 3 en quittant 2 vaut *matDesPrécedents*.get(2).get(3 - 2) c'est-à-dire 0. *Chemin* devient alors {3, 0, 2, 8}. Le précédent de 2 en quittant 0 est 0 lui-même donc le chemin est complet. C'est avec ce principe que nous avons établi l'algorithme 5 avec une complexité temporelle du même ordre que l'algorithme 3 c'est-à-dire O(L) avec L la longueur du chemin.

```
Algorithme 5 : CheminFromMatDesPrécedents (entree: entier, sortie: entier)
Variables :
    chemin : liste d'entiers, exactement comme ArrayList en Java [12].
    i : entier, indice d'insertion dans la liste chemin
    ie : entier, indice du nœud d'entrée.
    is : entier, indice du nœud de sortie.
    direction : entier, permet l'insertion à droite ou à gauche de sortie,
        (-1 : insertion à droite; +1 insertion à gauche).
Début
    chemin.add(entree) ; chemin.add(sortie) ;
    ie = 0 ; is = 1 ; i = is ; direction = +1 ; //insertion à gauche
    Tant que (True)
        sortie = matDesPrécedents.get(entree).get(sortie - entree)
        si entree == sortie alors arréter ; Fin si
        chemin.add(i, sortie) ;
        is = i ;
        si entree > sortie alors //permuter et changer de sens d'insertion
            entreeTmp = entree ; entree =  sortie ; sortie = entreeTmp ;
            iTmp = ie ; ie = is ; is = iTmp ;
            direction = direction * (-1) ;
        Fin si
        Si (direction == -1) alors //insertion à droite
            i = is + 1 ; ie = i + 1;
        Fin si
    Fin Tant que
Fin
```

D. Stockage des plus courts chemins

Puisque le graphe est non orienté, le plus court chemin entre A et B est le même que celle entre B et A, seul le sens de parcourt les différencie. Il est donc possible de stocker les plus courts chemins une fois construit en faisant une optimisation spatiale. C'est-à-dire ne stocker que les plus courts chemins entre les nœuds de numéro inférieur et les nœuds de numéro supérieur. Par exemple, avec le graphe *G* nous pouvons stocker les chemins entre 0 et tous les autres nœuds ; entre 1 et tous les autres nœuds sauf 0 etc. (tableau VII). Ce qui donne au maximum, un nombre de chemins de l'ordre de N(N-1)/2 avec N le nombre de nœud du graphe. Ce stockage est fait dans une matrice appelé matDesPlusCourtsChemins.

TABLEAU VII
MATDESPLUSCOURTSCHEMINS ENTRE LES NŒUDS DE DEPART 0, 1, 2 ET 3 ET TOUS LES NŒUDS D'ARRIVEE DU GRAPHE G

0							1							2						3				
1	2	3	4	5	6	7	8	2	3	4	5	6	7	8	3	4	5	6	7	8	4	5	6	7	8

01	02	03	024	035	0356	0287	028	12	103	124	1035	10356	1287	128	203	24	2035	2876	287	28	3024	35	356	3567	3028

Comme avec la matDesPrécedents, il est possible de construire toute la matDesPlusCourtsChemins justes après la construction du graphe ou au fur et à mesure que les utilisateurs construisent les chemins. Algo ?

E. Construction ou récupération d'un plus court chemin

Lorsqu'un utilisateur veut se construire un plus courts chemin entre un nœud de départ nDépart et un nœud d'arrivé nArrivé, nous avons adapté le l'algorithme 5 suivant :

```
Algorithme 6 : FaireChemin (entree: entier, sortie: entier)
Variables :
    chemin : liste d'entiers, exactement comme ArrayList en Java [12].
Début
    Si matDesPlusCourtsChemins.get(entree).get(sortie) existe alors
        renvoie matDesPlusCourtsChemins.get(entree).get(sortie)
    Sinon si matDesPrécedents.get(entree) existe alors
        chemin = CheminFromMatDesPrécedents(entree, sortie)
        Si matDesPlusCourtsChemins.get(entree) n'existe pas alors
            matDesPlusCourtsChemins.put(entree,
                HashMap<entier, ArrayList< entier >)
        Fin si
        matDesPlusCourtsChemins.get(entree).put(sortie, chemin)
        renvoie chemin
    Sinon
        tabDesPrécedents = Dijkstra(entree)
        matDesPrécedents.put(entree, tabDesPrécedents)
        FaireChemin (entree, sortie)
    Fin si
Fin
```

A part le stockage de la matrice d'adjacence, nous avons ajouté deux autre niveaux de stockage permettant de gagner en temps processeur que sont la matrice des précédents et la matrice des plus courts chemins.

III. Resultats

Nous avons utilisé cette optimisation dans deux simulateurs contenant des graphes de type voies de transporteur (tableau VIII). Les 5 premiers graphes (numérotés de 1 à 5) sont pour le premier simulateur et le graphe 6 beaucoup plus grand est pour le deuxième simulateur. Nous notons que pour ces 6 cas, plus de 93% de l'espace qu'occupait la matrice d'adjacence classique était inutile. Seul moins de 7 % de l'information est utile et est stocké par la matrice d'adjacence optimisée avec quasiment la même rapidité de teste d'adjacence entre deux nœuds (algorithme 2).

Tableau VIII
Resultat de l'Optimisation Spatiale de la Matrice d'Adjacence

	Graphe 1	Graphe 2	Graphe 3	Graphe 4	Graphe 5	Graphe 6
Nombre nœud	112	67	40	151	685	4677
E.M.A.C.	12544	4489	1600	22801	469225	21874329
E.M.A.O.	274	181	99	343	1960	4676
E. libéré	12270	4308	1501	22458	467265	21869653
E. libéré (%)	97,82	95,97	93,81	98,5	99,58	99,98

E.M.A.C. : Espace occupé par matrice d'adjacence classique ; E.M.A.O. : Espace occupé par la matrice d'adjacence optimisée ; E. libéré : espace libéré quand on utilise M.A.O. au lieu de M.A.C. ; E. libéré (%) : pourcentage d'espace libéré par rapport à la M.A.C ; l'espace c'est le nombre d'élément de type entier stocké.

L'espace que nous avons libéré avec l'optimisation spatiale est pour le stockage de matDesPrécedents et des plus courts chemins. Pour mieux étudier cela nous pouvons nous limiter au deuxième simulateur lancé avec 400 agents transporteurs utilisant le graphe 6. Parmi les deux manières de stocké la matDesPrécedents que nous avons proposé, nous avons adapté le 1er. C'est-à-dire le stockage au fil du temps. Ainsi, plus la simulation se passe plus matDesPrécedents est remplis, les plus courts chemins sont stockés (voir tableau IX) et donc moins Dijkstra est sollicité.

Tableau IX
Espace Occupe par matDesPrecedents et matDesPlusCourtsChemins avec le Graphe 6

Espace occupé \ Ticks	10 000 ticks	20 000 ticks	30 000 ticks
matDesPrécedents	252612	266 646	266 646
matDesPlusCourtsChemins	25156	28229	30254
tatal	277768	294875	296900

Un tick est un pas de temps de simulation. L'unité spatiale est l'espace mémoire que peut occuper un entier.

A 30 000 ticks, l'espace total occupé par la matAdjOpt, la matDesPrécedents et les plus courts chemins stockés vaut 301576. Ce qui est toujours inférieur à l'espace qu'aurait occupé la matrice d'adjacence classique pour le graphe 6 c'est-à-dire 21874329 (tableau VIII). Le nombre de chemin n'a pas explosé du fait qu'un chemin qui n'est jamais utilisé n'est jamais construit.

Pour estimer la rapidité gagnée, nous avons lancé le simulateur 2 pendant 10 000 ticks (pas de temps). Nous avons mesuré pour chaque tick et pour chaque construction d'un plus court chemin par un agent transporteur, la durée d'exécution dans chaque cas : cas où c'est Dijkstra qui est utilisé, cas où c'est matDesPrécedents qui est utilisé et le cas de la récupération directe d'un chemin déjà construit. Ces mesures sont réalisées avec la même machine et dans les mêmes conditions. Les résultats sont représentés dans le tableau X ci-dessous.

Tableau X
Durees Moyennes de Construction ou de Recuperation de plus Court Chemin apres 30 000 ticks

	dijkstra	matDesPrécedents	chemin prêt
Nombre de mesures	57	20665	1609
Durée moyenne (en ms)	768,5555556	0,010988669	0,000180206
Rap (durée i / durée i+1)	69940,73138	60,97837255	

Pendant 30 000 ticks et avec 400 agents transporteur, Dijkstra n'est appelé que 57 au lieu de 22274 fois (20665 + 1609) ; En d'autre terme 22274 plus court chemins sont construits en n'utilisant que 57 fois Dijkstra.

69940,73138 = 768,5555556 / 0,010988669 ;
60,97837255 = 0,010988669 / 0,000180206

Le tableau X montre que construire un plus court chemin à partir de matDesPrécedents (O(L)) est en moyenne plus de 69 000 fois plus rapide qu'en utilisant Dijkstra ($O(N^2)$). Et que récupérer un chemin à partir de matDesPlusCourtsChemins est en moyenne plus de 60 fois plus rapide qu'à partir de matDesPrécedents. Il faut aussi noter que les premiers appels de Dijkstra peuvent être assez lourds du fait que matDesPrécedents et matDesPlusCourtsChemins sont initialement vides.

Le tableau X montre aussi qu'avec 400 agents transporteurs et pendant 30 000 ticks, Dijkstra n'est utilisé que 57 fois pour construire matDesPrécedents à la demande des transporteurs. Un nœud de départ associé à un tabDesPrécédents (de taille N) peut être utilisé N fois pour construire N plus courts chemins différents (avec N le nombre de nœuds du graphe). Tandis qu'un chemin stocké est simplement pour un nœud de départ et un nœud d'arrivé. Ce qui explique que, jusqu'à 30 000 ticks, matDesPrécedents est plus sollicitée que matDesPlus-CourtsChemins.

IV. Discussion

L'optimisation temporelle que nous venons de présenter ne prend pas effet tout au début de la simulation ou de l'utilisation d'un graphe, ou tout juste après la mise à jour d'un graphe. Il faut donc s'attendre à une certaine lourdeur (dépendant de la taille du graphe et du nombre d'agents) pour les premiers appels de la méthode FaireChemin (de l'algorithme 6. Cette serait la même pendant toute la simulation cette optimisation n'était pas faite.

L'algorithme de recherche de chemin dans un graphe A* [9] qui est plus rapide que Dijkstra [3] et qui donne souvent de bon chemins pouvait être utilisé, mais nous avons préféré utiliser Dijkstra qui est suffisamment rapide et qui donne exactement le plus court chemin dans toutes les situations. Et que construire un plus court chemin à partir de matPrécédents est plus rapide et plus sure qu'avec A*. Il n'est pas souhaitable d'utiliser cette optimisation avec A*. En effet l'algorithme A* n'explore pas tous les nœuds et donc ne fournit pas de tab-DesPrécédents. Et il peut arriver qu'il fournisse un très mauvais chemin [13] pour être stocké et être réutilisé par d'autres agents.

Il faut aussi noter que si l'optimisation temporelle est suffisante avec le stockage de matDesPrécedents, alors on peut se passer du stockage des plus courts chemins.

V. Conclusion

Nous avons montré comment nous pouvons passer de l'optimisation spatiale de la représentation informatique d'un graphe pour stocker de manière optimale les tableaux des précédents et les plus courts chemins construits par l'algorithme de Dijkstra. Ainsi nous sommes arrivés à une optimisation temporelle de l'utilisation de ces plus courts chemins dans un simulateur Multi-agents.

Si la chronologie des créations et de gestion des graphes peut être maitrisée avant le commencement de la simulation, il serait possible d'extérioriser une bonne partie de la gestion des graphes et de construction des chemins. Et de le mettre sous forme chronographe [11]. Ainsi une phase pré-simulation pourrait permettre de construire le graphe et de stocker dans fichier tous les tableaux des précédents et ou les plus courts chemins entre certains nœuds. Les informations dans ce fichier seront récupérées au bon moment et stockées au bon endroit dans la simulation. Ce qui permettrait d'avoir les plus courts chemins pendant toute une simulation sans l'utilisation de Dijkstra ou d'un autre algorithme de recherche de chemin dans un graphe.

References

[1] P. Auger, J. Baudry, et F. Fournier, (1992). Hiérarchies et échelles en écologie. Naturalia Publications. Paris.

[2] CEDEAO et CSAO, *Atlas régional des transports et des télécommunications dans la Cedeao*, Abuja et Issy-les-Moulineaux, 2005.

[3] Dijkstra E. W., (1959). A note on two problems in connexion with graphs, *Numerische mathematik*, vol. 1, no 1, p. 269–271.

[4] L. Granjon et J.-M. Duplantier, "Les rongeurs de l'Afrique sahélo-soudanienne," in *Collection Faune et Flore Tropicale*, IRD Éditions / MNHN. Marseille, 2009.

[5] J. Le Fur, sept. 2013. A formal framework for linking multidisciplinary multiscale knowledge. A case study on rodent population dynamics and management. Presented at European Conference on Complex Systems, Barcelona, p. 5, sept-2013.

[6] P. A. Mboup, "Construction d'un environnement de simulation multi-agents pour l'étude de la diffusion du rat noir au Sénégal au cours du siècle écoulé," Mémoire de Master, Univ. Cheikh Anta Diop de Dakar, FST, Département Maths et Informatique, Dakar, 2012.

[7] M. J. North, N. T. Collier, J. Ozik, E. R. Tatara, C. M. Macal, M. Bragen, et P. Sydelko, (2013). Complex adaptive systems modeling with Repast Simphony, *Complex Adaptive Systems Modeling*, vol. 1, no 1, pp. 1–26.

[8] D. Pumain, (2006). Hierarchy in Natural and Social Sciences. *Springer*, Methodos Series 3.

[9] Hart, E. Peter, J. N. Nils, and R. Bertram (1968). A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics, IEEE Transactions on 4.2*: pp. 100-107.

[10] A. P. Tafti, S. Janosepah, N. Modiri, A. M. Noudeh & H. Alizadeh (2011). Development of a Framework for Applying ASYCUDA System with N-Tier Application Architecture. In *Software Engineering and Computer Systems* (pp. 533-541). Springer Berlin Heidelberg.

[11] Article 1 CNRIA

[12]http://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html#add(int, %20E)

[13] http://fr.wikipedia.org/wiki/Algorithme_A*