Optimisation de l'utilisation de l'algorithme de Dijkstra pour un simulateur multi-agents spatialisé

Pape Adama Mboup[1,2], Mamadou Lamine Mboup[3], Karim Konaté[1]
[1]Département de mathématiques et d'informatique, UCAD
[2]IRD, UMR 022 BIOPASS, Campus de Bel-Air, BP 1386
[3]Département Génie Informatique, ESP, UCAD
Dakar, Sénégal
pamboup@hotmail.fr

Pascal Handschumacher[4] et Jean Le Fur[5]
[4]IRD, UMR 912 SESSTIM INSERM - IRD - U2, Faculté de géographie UdS, 3, rue de l'Argonne F-67000
[5]IRD, UMR 022 CBGP, Campus international de Baillarguet, CS 30016, 34988
[4]Strasbourg, [5]Montferrier-sur-Lez, France

***Abstract*—Les techniques de modélisation basées sur le déplacement d'agents dans une topologie de type graphe s'avèrent une approche fructueuse. Beaucoup de modèles liés aux déplacements d'agents utilisent l'algorithme de Dijkstra pour construire, à coup sûr, les plus courts chemins. Cependant un problème majeur de ces modèles est la fréquence à laquelle les nombreux agents, 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 un algorithme permettant une optimisation spatiale de la représentation informatique d'un graphe (matrice d'adjacence, liste d'adjacence), suivi d'un stockage optimisé de tout plus court chemin une fois construit. Cette optimisation évite aux agents d'avoir à reconstruire des chemins déjà construits et supprimés. Ce qui réduit 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; système multi-agents.

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 du rat noir, un des principaux moteurs présumés de la colonisation du territoire est constitué par les transports humains : routiers, ferroviaires et fluviaux.

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, rails, routes) sur l'ensemble du territoire sénégalais pendant un siècle implique une grande quantité de chemins à construire dans le temps et dans l'espace. Ce qui nécessite donc un grand temps de calcul qu'il faut impérativement diminuer.

Dans cet article, nous présentons comment nous sommes passés 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'éviter aux agents transporteurs de se reconstruire des chemins déjà construits auparavant. Ce qui permet de diminuer considérablement le temps de calcul dû à la construction de plus court chemin. Avec cette optimisation, nous pouvons utiliser l'algorithme de Dijkstra pour avoir exactement les plus courts chemins. Ainsi nous n'avons plus besoin de recourir à d'autre algorithmes tel que A* [9], plus rapide que Dijkstra [3], mais ne garantissant pas d'avoir le plus court chemin.

II. MATÉRIELS ET MÉTHODES

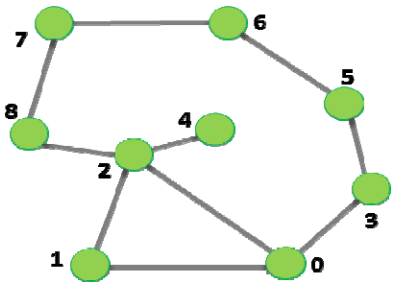
Le simulateur est développé à partir de la plate-forme pluridisciplinaire de modélisation et de simulation de populations de rongeurs par les systèmes multi-agents SimMasto [1]. Développée en java sous Repast Simphony [7] et fondée sur une architecture trois tiers (data-business-presentation) [10], SimMasto est conçu robuste et flexible. Elle contient une topologie de type grille : l'espace (y compris les villes et les routes) est discrétisé en cellules élémentaires. Ce qui a permis d'y greffer facilement un module permettant la gestion de déplacement d'agents sur une nouvelle topologie de type graphe [11]. Nous considérons un graphe comme étant un ensemble de villes reliées par un type de voie de transport. Les cellules des villes et des voies de transport constituent les nœuds du graphe. La distance entre deux nœuds successifs (arête) est égale à l'unité (1), ce qui a conduit à la possibilité d'utiliser des graphes non-pondéré tout en prenant en compte les distances [6].

Dans cette topologie nous avons utilisé des graphes non-orientés, non-pondérés et dont, sur la représentation informatique d'un graphe, seule une information nous intéresse : la liste des nœuds adjacents (le voisinage) à un nœud donné. Ce qui est très utile pour la phase de mise à jour des labels des voisins non marqués dans l'algorithme de Dijkstra [8].

A. *Graphe*

Un graphe non-orienté G est un couple (V, E) où $V = \{v_1, v_2, ..., v_N\}$ est un ensemble fini d'objets et $E = \{e_1, e_2, ..., e_M\}$ est sous-ensemble de V x V. Les éléments de V (au nombre de N) sont appelés les sommets ou nœuds du graphe et les éléments de E (au nombre de M) sont appelés les arêtes du graphe. Une arête $\{v_i, v_j\}$ relie deux nœuds adjacents v_i et v_j ($v_i \in V$ et $v_j \in V$). On dit que l'arête $\{v_i, v_j\}$ est incidente aux nœuds v_i et v_j. Le degré D d'un nœud v_i du graphe G est le nombre d'arêtes incidentes à ce nœud. 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).



$V = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$, $E = \{\{0,1\}, \{0,2\}, \{0,3\}, \{1,2\}, \{2,4\}, \{2,8\}, \{3,5\}, \{5,6\}, \{6,7\}, \{7,8\}\}$.

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

1) *Matrice d'adjacence classique et ses inconvénients:* 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 :

$$m_{i,j} = \begin{cases} 1 \; si \; \{v_i, v_j\} \in E \\ 0 \; si \; \{v_i, v_j\} \notin E \end{cases}$$

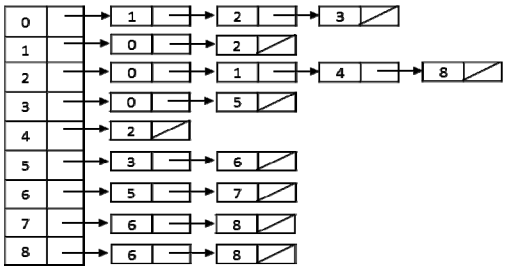
TABLEAU I MATRICE D'ADJACENCE CLASSIQUE DE G

	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

Il est vrai que l'accès à un élément est très rapide avec les matrices classiques (test d'adjacence entre deux nœuds en une complexité temporelle constante O(1)). Cependant un inconvénient avec cette représentation est la complexité temporelle de récupération de la liste des nœuds adjacents à un nœud donné (complexité linéaire O(N)). Il faut parcourir toute une ligne ou toute une colonne pour déterminer cette liste pour un nœud donné. Un autre inconvénient de cette représentation est la complexité spatiale. L'espace mémoire utilisé est de l'ordre de N^2 pour un graphe d'ordre N, même lorsque le graphe représente beaucoup moins que N^2 arêtes. Réduire cette complexité spatiale peut conduire à l'utilisation des listes d'adjacences.

2) Liste d'adjacence classique et ses inconvénients: Une liste d'adjacence classique est un tableau de pointeurs où 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 (listes chaînées). Le tableau II est une liste d'adjacence du graphe G. Par rapport à la matrice d'adjacence, la liste d'adjacence classique procure déjà une optimisation spatiale en ne représentant pas les non-adjacences. Par rapport à notre étude un inconvénient mineur des listes chainées est l'espace qu'occupent ses pointeurs et les procédures à mettre en place pour les manipuler.

TABLEAU II LISTE D'ADJACENCE CLASSIQUE DE G



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

Notre objectif était de trouver une nouvelle structure de données plus optimisée de représentation informatique d'un graphe. Il s'agit d'une structure qui garantirait l'optimisation spatiale comme avec une liste d'adjacence classique (c'est-à-dire ne représentant pas les non-adjacences), sans l'utilisation des listes chaînées.

Pour cela nous avons utilisé un tableau associatif (clé-valeur). Les clés sont les numéros des nœuds du graphe et les valeurs associées aux clés sont des tableaux de taille variable (collection en Java). Le tableau associatif est caractérisé par la complexité constante O(1) de récupération de la valeur associée à une clé (get(clé)). Les tableaux simples (valeurs) sont caractérisés par la simplicité de parcourir les éléments qu'ils contiennent et d'y accéder par indice. Ainsi nous avons associé deux structures de données que nous appelons respectivement HashMap et ArrayList (comme en Java) : (*i*) HashMap est un tableau associatif qui référence ses objets (valeurs) par clé. Elle permet l'ajout (fonction put()) et la récupération (fonction get()) d'une valeur par une clé avec une complexité temporelle constante O(1). (*ii*) ArrayList est un tableau qui permet d'ajouter (fonction add(élément)) un élément en son sein 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. Alors dans la *matAdjOpt*, n1 est une clé associée à un tableau contenant n2 et réciproquement.

```
Algorithme 1 : ConstructionDeLaMatriceDadjacenceOptimisée ()
Variable :
    nodeList : un tableau simple contenant par ordre tous les nœuds du
    graphe. La position d'un nœud dans ce tableau correspond à son numéro
    matAdjOpt: HashMap <Entier, ArrayList Nœud>>
Début
    Pour numéro allant de 0 à la taille de nodeList faire
        oneNode = nodeList[numéro]
        matAdjOpt.put(numéro, new ArrayList <Noeud>())
            Pour chaque numéro de noeud oneNeighboringNode dans le voi-
            sinage de oneNode faire
            matAdjOpt.get(numéro).add(oneNeighboringNode);
        Fin Pour
    Fin Pour
Fin algorithme
```

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

Dans cette représentation optimisée, toutes les propriétés de la matrice d'adjacence classique ne sont pas conservées. Cela ne nous pose pas de problème car ce qui nous intéresse ici c'est de pouvoir récupérer la liste des nœuds voisins à un nœud donné. Ce que l'algorithme 2 fait avec une complexité temporelle constante O(1).

```
Algorithme 2: ListDesVoisinDunNœuds(noeud1 : Nœud)
Début
    renvoyer edgesMatrix.get(noeud1.numéro)
Fin algorithme
```

Comparé à une matrice d'adjacence classique, cette manière de procéder nous permet de gagner plus de 50% en espace (64,20 % avec le petit graphe *G*). Ce pourcentage dépend de la densité des connexions entre les nœuds.

C. Stockage de la matrice des précédents

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 *tabDesPrécédents.* La matrice *matDesPrécédents* est composée des *tabDesPrécédents*. Le tableau IV représente le *tabDesPrécédents* 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 sera précédé par le nœud 8 (voir fig. 1).

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

nœuds	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
nœuds précédents	0	0	0	0	2	3	5	8	2

A partir de ce *tabDesPrécédents* (tableau IV) 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 est le plus court chemin entre 7 et 0 parcourut dans l'autre sens.

L'Algorithme 3 construit à partir de *matDesPrécédents* le plus court chemin entre deux nœuds avec une complexité temporelle O(L), L étant la longueur du chemin.

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

Quand un agent transporteur veut se construire un plus court chemin entre un nœud de départ *noeud1* et un nœud d'arrivée *noeud2*, il utilise l'algorithme de Dijkstra pour construire d'abord le *tabDesPrécédents* pour *noeud1* pour ensuite en construire le chemin cherché. Au lieu de jeter ce *tabDesPrécédents* temporairement construit, l'idée est de le sauvegarder dans *matDesPrécédents* (tableau V). De ce fait quand un autre utilisateur voudra se construire un chemin dont *noeud1* est le nœud de départ ou d'arrivée, qu'il ne réinvente pas la roue en appelant à nouveau Dijkstra pour reconstruire le même *tabDesPrécédents* ou un autre. Il ne fera que construire son chemin à partir d'un *tabDesPrécédents* déjà dans *matDesPrécédents*. Pour mettre en place ce procédé il y a deux manières de faire : soit construire toute la *matDesPrécédents* juste après la construction du graphe, soit la construire au fur et à mesure que les utilisateurs appellent Dijkstra.

TABLEAU V MATDESPRÉCÉDENTS 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

1) Construction et stockage de la matDesPrécédents au fur et à mesure que les utilisateurs appellent Dijkstra: Une des manières de faire est de ne pas construire et stocker a priori les *tabDesPrécédents* mais d'attendre qu'un utilisateur en construise un pour le sauvegarder en vue d'autres usages ultérieurs. Ainsi un *tabDesPrécédents* qui est construit l'est pour tous.

Il n'est pas possible de faire une optimisation spatiale de la *matDesPrécédents* du fait qu'elle ne contienne pas a priori toute l'information de sa partie triangulaire supérieure ou inférieure (voir le point 2 suivant).

2) Stockage de matDesPrécédents dès le début: Il est possible, juste après la construction du graphe, de construire complètement la *matDesPrécédents*. La complexité temporelle est, alors, égale à N fois la complexité de l'algorithme de Dijkstra, N étant le nombre de nœuds du graphe. Ainsi Dijkstra ne sera plus 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 *matDesPrécédents* en ne stockant que sa partie triangulaire inférieure ou supérieure. En effet toute l'information pour construire un plus court chemin entre deux nœuds est contenue dans cette partie supérieure ou inférieure totalement construite.

a) Construction et stockage de la matDesPrécédents optimisée: Supposons que nous choisissions 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 les numéros de nœuds associés à des tableaux de taille variable.

Après que Dijkstra ait construit le *tabDesPrécédents* 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*. Ceci peut être fait avec une fonction que nous appelons subList (comme en Java). Cette fonction prend deux paramètres : l'indice de début et l'indice de fin de la portion du tableau à garder comme présenté dans l'algorithme 4.

```
Algorithme 4 : ConstructionDeLaMatriceDesPrécédentsOptimisée ()
Variables :
    nœudDépart : entier
Début
    Pour nœudDépart allant de 0 à taille de nodeList faire
        tabDesPrécédents = Dijkstra (nœudDépart)
        matDesPrécédents.put ( nœudDépart,
            tabDesPrécédents.subList (nœudDépart + 1,
                                      taille de tabDesPrécédents))
    Fin Pour
Fin algorithme
```

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

TABLEAU VI MATDESPRÉCÉDENTS 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 *matDesPrécédents.get(nDépart).get(nArrivé - nDépart)*. Exemple : si *nDépart = 3* et *nArrivé = 8* alors
matDesPrécédents.get(3).get(8-3) =
matDesPrécédents.get(3).get(5) = 2.

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

b) *Construction d'un plus court chemin à partir de la matDesPrécédents 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}). Il reste d'insérer, dans *chemin* et aux bons endroits, les nœuds intermédiaires. Puisque 3 est inférieur à 8, il est possible de récupérer le nœud précédant le nœud 8 par l'instruction *matDesPrécédents*.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 (puisque 2 < 3) ; Le précédent de 3 en quittant 2 vaut *matDesPrécédents*.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 : CheminFromMatriceDesPrécédents (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 = matDesPrécédents.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 algorithme
```

D. Stockage des plus courts chemins

Puisque le graphe est non orienté, le plus court chemin entre A et B est le même que celui entre B et A, seul le sens de parcours les différencie. Il est donc possible de stocker les plus courts chemins une fois construits en faisant une optimisation spatiale. C'est-à-dire ne stocker que les plus courts chemins

allant des nœuds de numéros inférieurs vers les nœuds de numéros supérieurs. 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 ainsi de suite (comme dans 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*.

Comme avec la *matDesPrécédents*, il est possible de construire toute la *matDesPlusCourtsChemins* justes après la construction du graphe ou au fur et à mesure que les agents transporteurs construisent les chemins.

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

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

E. *Construction ou récupération d'un plus court chemin*

Lorsqu'un agent transporteur veut se construire un plus court chemin entre un nœud de départ *entree* et un nœud d'arrivée *sortie*, il utilise la fonction *FaireChemin* de l'Algorithme 6. Cette fonction prend en paramètre le nœud le plus petit suivi du plus grand (en comparant *entree* et *sortie*, l'agent saura dans quelle sens il parcourra sont chemin). La fonction *FaireChemin* cherche en premier lieu si le chemin est déjà construit pour le renvoyer directement ; Sinon elle cherche si le *tabDesPrécédents* pour le nœud *entree* (premier paramètre) est déjà construit pour ensuite en élaborer le chemin, le sauvegarder dans *matDesPlusCourtsChemins* et le renvoyer ; Sinon elle appelle la fonction Dijkstra pour le nœud *entree*, renseigne *matDesPrécédents* et fait un appelle récursif de la fonction *FaireChemin*.

```
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 matDesPrécédents.get(entree) existe alors
        chemin = CheminFromMatDesPrécédents(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
        tabDesPrécédents = Dijkstra(entree)
        matDesPrécédents.put(entree, tabDesPrécédents)
        FaireChemin (entree, sortie)
    Fin si
Fin algorithme
```

En plus du stockage de la matrice d'adjacence, nous avons ajouté deux autres niveaux de stockage permettant de gagner en temps processeur que sont la matrice des précédents et la matrice des plus courts chemins.

III. RÉSULTATS

Nous avons utilisé cette optimisation dans deux simulateurs contenant des graphes de type voies de transport (tableau VIII). Les 5 premiers graphes (numérotés de 1 à 5) sont pour le premier simulateur. Le graphe 6 beaucoup plus grand est pour le deuxième simulateur. Les résultats du tableau VIII montrent que, pour ces 6 cas de graphe, de 87,63 à 99,96 % de l'espace qu'occupait la matrice d'adjacence classique était inutile. Seul moins de 13 % de l'information est utile et est stockée dans la matrice d'adjacence optimisée ; Avec la complexité de récupération du voisinage d'un nœud (algorithme 3) qui passe de O(N) avec la matrice d'adjacence classique à O(1) avec la matrice d'adjacence optimisée.

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.	548	362	198	686	3920	9352
E. libéré	11996	4127	1402	22115	465305	21864977
E. libéré (%)	95,63	91,94	87,63	96,99	99,16	99,96

E.M.A.C. : Espace occupé par la 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 de l'espace libéré par rapport à la M.A.C ; l'unité utilisée est le nombre d'élément de type entier stocké.

L'espace que nous avons libéré avec l'optimisation spatiale est pour le stockage de la matrice des précédents et de la matrice 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 stocker la *matDesPrécédents* que nous avons proposé, nous avons adapté la 1ère. C'est-à-dire le stockage des tableaux des précédents et des chemins au fur et à mesure que les agents en construisent. Ainsi, autant la simulation progresse, autant la matrice des précédents et la matrice des plus courts chemins se remplissent (voir tableau IX). Les appels de la fonction Dijkstra s'en trouvent considérablement réduits.

TABLEAU IX ESPACE OCCUPE PAR MATDESPRÉCÉDENTS ET MATDESPLUSCOURTSCHEMINS POUR LE GRAPHE 6

	10 000 ticks	**20 000 ticks**	**30 000 ticks**
matDesPrécédents	252612	266 646	266 646
matDesPlusCourtsChemins	25156	28229	30254
total	277768	294875	296900

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

A 30 000 ticks (pas de temps de simulation, 1 tick = 1 jour), l'espace total occupé par la *matAdjOpt*, la *matDesPrécédents* et la *matDesPlusCourtsChemins* vaut 301 576. Ce qui est toujours inférieur à l'espace qu'aurait occupée la matrice d'adjacence classique pour le graphe 6 c'est-à-dire 21 874 329 (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 30 000 ticks. 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 *matDesPrécédents* qui est utilisé ;
- 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 NOMBRE D'UTILISATION ET DUREES MOYENNES DANS CHAQUE CAS APRES 30 000 TICKS

	Dijkstra	matDesPrécédents	chemins prêts
Nombre d'utilisations	54	20399	1277
Durée moyenne (en ms)	2,40	0,01	0,00018

Le tableau X montre qu'avec 400 agents transporteurs et pendant 30 000 ticks, Dijkstra n'est utilisé que 54 fois au lieu de 21 676 fois (20 399 + 1 277) pour construire *matDesPrécédents* à la demande des transporteurs. En d'autres termes 21 676 plus courts chemins sont construits ou utilisés en n'appelant que 54 fois Dijkstra. Ce tableau montre aussi que construire un plus court chemin à partir de *matDesPrécédents* (O(L)) est en moyenne 240 (2,40 / 0,01) fois plus rapide qu'en utilisant la fonction Dijkstra. Récupérer un chemin à partir de *matDesPlusCourtsChemins* est en moyenne 55,56 (0,01 / 0,00018) fois plus rapide qu'à partir de *matDesPrécédents*.

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'arrivée. Ce qui explique que, jusqu'à 30 000 ticks, *matDesPrécédents* est plus sollicitée que *matDesPlusCourtsChemins*.

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, du nombre d'agents et de la qualité de l'implémentation de l'algorithme Dijkstra) pour les premiers appels de la fonction *FaireChemin* de l'algorithme 6. Cette lourdeur serait maintenue durant toute la simulation si cette optimisation n'était pas réalisée.

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. Construire un plus court chemin à partir de *matDesPrécédents* est plus rapide et plus sûr qu'à partir de l'algorithme 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 *tabDesPrécédents*. Il peut aussi arriver que cet algorithme fournisse un mauvais chemin [5] qui serait stocké et réutilisé par d'autres agents pendant toute une simulation.

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

V. CONCLUSION

Dans cette étude, 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. Ce qui permet d'éviter aux agents transporteurs de se reconstruire des chemins déjà construits auparavant et donc diminue considérablement le temps de calcul dû à la construction de plus court chemin. Ainsi nous sommes arrivés à une optimisation spatiale et temporelle de l'utilisation de l'algorithme de Dijkstra dans un simulateur Multi-agents spatialisé.

On peut à présent envisager la simulation de modèle permettant d'étudier la diffusion des rats par le transport humain par des réseaux routiers, ferrés et fluviaux assez grands, appréhendés sous forme de graphe.

RÉFÉRENCES

[1] 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.

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

[3] E. W. Dijkstra, (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] http://fr.wikipedia.org/wiki/Algorithme_A*.

[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] http://blog.christophelebot.fr/wp-content/uploads/2007/03/theorie_graphes.pdf.

[9] E. Peter Hart, J. N. Nils, and B. Raphael (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] P. A. Mboup, K. Konaté, P. Handschumacher et J. Le Fur (2015). "Des Connaissances au Modèle Multi-agents par l'approche orientée événement : construction d'un simulateur de la colonisation du rat noir au sénégal par les transports humains sur un siècle," Colloque National sur la Recherche en Informatique et ses Applications, Thies, Sénégal. Unpublished.

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