Projet Algorithmie et Complexité MACS 2

Martin Duguey[1]

3 avril 2022

Résumé

L'objet de ce projet est de simuler la propagation d'une épidémie de Covid-19 dans une population donnée. Cependant, il n'est en aucun cas question de résoudre numériquement un problème régit par des équations différentielles. Ces objets et leur résolution dépassent le cadre du cours. Tout l'enjeu de cette modélisation est donc d'utiliser à bon escient les objets présentés et expliqués dans le cours d'Algorithmie et Complexité de M. Roberto WOLFLER.

1 Introduction

Toute la complexité du projet réside dans le fait que l'on souhaite simuler une propagation d'épidémie, en l'occurence celle du Covid-19, à l'échelle d'une grande population, c'est à dire de l'ordre du million d'individus à minima. Il faut donc prendre en compte les données de cette épidémie pour approcher au mieux avec notre modèle la propagation dans le monde réel tout en faisant des choix de modélisation qui vont nous permettre d'avoir des résultats assez rapidement.

Pour vous présenter ce projet, nous allons voir étape par étape les choix de modélisation qui ont été fait ainsi que les outils mis en oeuvre pour les réaliser. Nous allons aborder la découverte du projet de la manière suivante :

- ◇ Création d'une population et liaison entre les individus
- ◇ Mise en place de la propagation
- ◇ Résultats obtenus et analyse

Pour mettre en place ce projet, nous avons fait le choix d'utiliser le langage C++ qui permet notamment d'effectuer de la programmation orientée objet.

2 Création d'une population et liaison entre les individus

2.1 Outils de modélisation

De manière à stocker de l'information concernant les individus il a fallu créer des objets de stockage.

— La liste chaînée [2] :

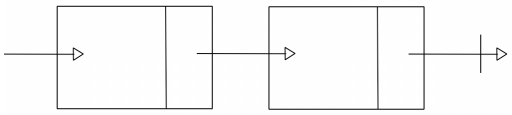

FIGURE 1: Représentation d'une liste chainée

Avec cette représentation et grâce à la programmation générique, on peut stocker tout type de données dans une liste chainée. On a choisi d'implémenter cette liste comme le ferait une pile par exemple. C'est à dire que chaque nouvel élément est ajouté en début de liste. On a choisi d'implémenter les fonctionnalités suivantes pour la liste chaînée : l'ajout d'un élément, la recherche d'un élément, la modification d'un élément, l'obtention du nombre de noeuds total de la liste et l'obtention de la donnée stockée en première position.

1. Sup Galilée, Filière Ingénieurs M.A.C.S, Institut Gailée, Université Sorbonne Paris Nord, F-93430, Villetaneuse, France

2. Le détail de cet objet est disponible via le fichier index.html que l'on retrouve dans le dossier documentation du projet

Etude de la complexité

Ajout d'un élément : On ajoute un élément en début de liste, donc l'opération n'implique qu'une gestion des pointeurs premier de la liste et suivant du noeud, c'est instantanné. La complexité est en $\mathcal{O}(1)$.

Recherche d'un élément : On recherche un élément dans une liste de taille n quelconque en commençant par le début. La pire configuration serait d'aller chercher le dernier élément, ce qui serait très couteux pour n grand. La complexité, dans le pire des cas, est donc en $\mathcal{O}(n)$.

Modification d'un élément : Là encore, il faut faire appel à la recherche d'un élément puisqu'on modifie un élément déjà existant dans la liste. La complexité, dans le pire des cas, est donc aussi en $\mathcal{O}(n)$.

Les autres opérations constituent simplement un renvoie d'attributs caractéristiques des classes liste et noeud, donc la complexité est en $\mathcal{O}(1)$.

— Le vecteur :

Etude de la complexité

Ajout d'un élément : L'ajout se fait, soit lors de l'initialisation, soit après l'avoir initialiser et dans ce cas on peut ajouter un élément à notre vecteur, au début ou à la fin. La compléxité en temps est donc, tout comme la pile, en $\mathcal{O}(1)$.

Recherche d'un élément : La particularité du vecteur est que l'on peut accéder à un élément juste en indiquant sa position dans le vecteur. On a alors une complexité en $\mathcal{O}(1)$.

Modification d'un élément : Puisque la modification implique la recherche de l'élément à modifier et que c'est cette étape qui est la plus couteuse, on a alors une complexité en $\mathcal{O}(1)$.

Comparaison de la vitesse de l'algorithme en utilisant la liste ou le vecteur

On a tracé le temps de création d'une population de taille $500, 1000, 1500, 2000, 3000, 5000, 10000, 25000, 50000, 100000$ avec des outils différents.

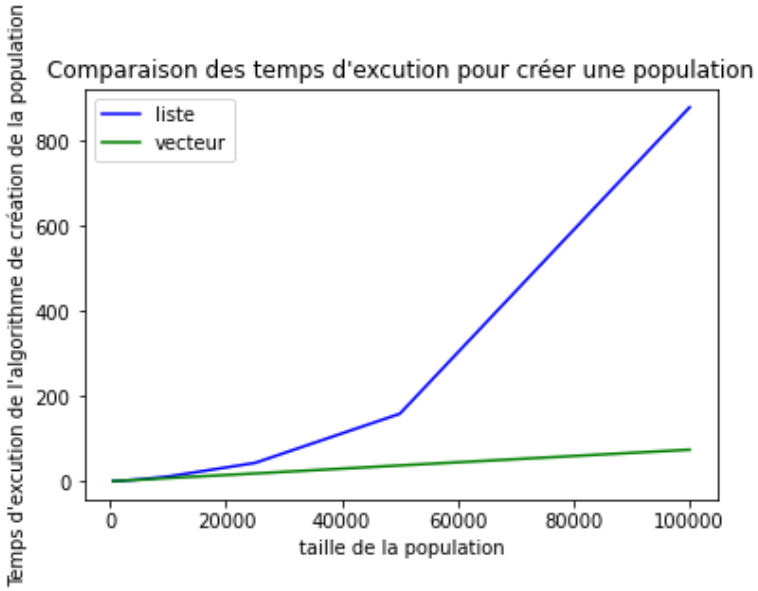

FIGURE 2: Complexité de l'algorithme de création de la population

On remarque visuellement les avantages de complexité qu'offre le vecteur. Et donc en stockant les données de chaque personne de la population dans un vecteur, on a une vitesse d'exécution de l'aglorithme qui est plus optimale que si on avait fait le choix de stocker ces données dans une liste.

2.2 Les classes population et personne

La seconde étape de la modélisation a été de créer deux classes [3] : une classe personne et une classe population.

La classe personne

La classe personne, qui définit une personne, possède les attributs suivant :

— Un identifiant (unsigned int)

— Un âge (int)

— Un sexe (string : "feminin" ou "masculin")

— Des coefficients de relation (double)

— Une liste de contacts (stockage d'entier unsigned int correspondant aux identifiants des personnes contact)

L'identifiant est tout simplement un entier $n \in [\![1, N]\!]$ avec N la taille de la population. L'âge est généré à partir de la fonction *gen_age*(). Le sexe est attribué par un tirage uniforme sur $[0, 1]$, donc on a une chance sur deux d'être l'un ou l'autre. Les coefficients de relation sont attribués aléatoirement par un tirage uniforme sur $[0, 1]$. Et enfin la liste de contacts est un attribut de type liste (sous-entendu liste chaînée, soit l'outil qu'on a choisi d'implémenter), qui contient des entiers (unsigned int) correspondant aux identifiants des personnes avec qui chaque personne est en contact. Initialement, cette liste est vide.

La classe population

La classe population, qui définit donc un ensemble de personnes, possède les attributs suivant. Pour lier ces personnes entre-elles, nous ferons intervenir dans cette partie la notion de graphe :

— Une taille (unsigned int)

— Un vecteur population (stockage d'objet de type personne)

— Un vecteur etats (stockage d'entiers int, 0 personne non-infectée, 1 personne contaminée, -1 personne déjà contaminée et immunisée, -2 personne décédée)

— Une liste d'infectés (stockage d'entier unsigned int correspondant aux identifiants des personnes infectés)

La taille de la population est un entier qui est définit en amont de la construction avec le constructeur *population*(). Le vecteur population contient toutes les personnes de la population. Le vecteur etats contient tous les états des personnes de la population. La liste d'infectés contient les identifiants des personnes initalement infectées (on considère qu'il y a 2 infectés initialement).

La difficulté lors de la conception de cette classe a été de définir quelle personne de la population est en contact avec une autre personne de la population. Initialement, l'envie était de laisser cette tâche à une fonction qui tirerait des personnes au hasard pour les faire communiquer. Mais avant cela, il faut faire un point sur la notion de graphe. Nous avons fait le choix de voir notre population comme un graphe connexe non-orienté. Ainsi chaque personne est en relation avec d'autres personnes et donc pour n'importe quelle personne, donc noeud du graphe, on peut tourver un chemin passant par d'autres personnes pour arriver à une personne précise. De plus le caractère non-orienté du graphe implique qu'une personne A, qui fait partie des contacts d'une personne B, a aussi dans ses contacts la personne B. Ainsi en représentant notre graphe dans une matrice d'adjacence $\mathcal{M}$, on sait déjà que $\mathcal{M}$ est symétrique.

Il faut pouvoir gérer cette notion de symétrie entre chaque personne. C'est pour ça, que le tirage aléatoire pose problème. En effet, en tirant aléatoirement on a aucun contrôle sur les éventuels doublont (une personne ne peut pas occuper deux places dans la liste de contacts d'une autre personne), et ceci pose un énorme problème de complexité en temps. Pour palier, à ce problème, on aurait pu faire des tirages entres des personnes bien précises mais dans ce cas la notion de symétrie est beaucoup plus difficile à mettre en place. Ces problèmes nous pousse à imaginer une autre solution. L'idée retenue est donc de visualiser chaque individu dans un plan, en 2 dimensions. De ce fait, nous numérotons les individus de 1 à N, N étant la taille de la population. Ce numéro constitue en fait l'identifiant d'une personne. Ainsi on a la représentation suivante, pour une poputlation d'un million d'individus.

3. Le détail de ces classes est disponible via le fichier index.html que l'on retrouve dans le dossier documentation du projet

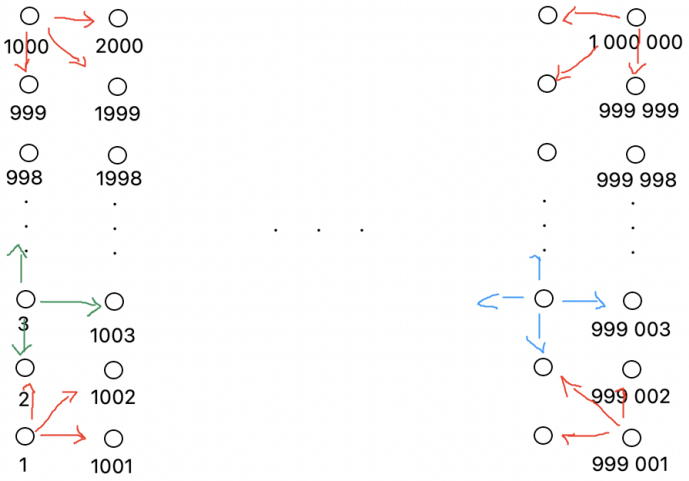

FIGURE 3: Représentation de la population dans le plan

Cette représentation de la population permet de connaître les personnes en contact avec chaque individu avant même de construire notre population. Ainsi, les individus qui sont à l'interieur ont 4 voisins (bleu sur la figure 3), les individus qui sont sur les bords ont 3 voisins (vert sur la figure 3) et les individus aux quatre coins de la représentation ont 3 voisins mais définis de manière plus arbitraire : on a pris les 3 individus les plus proches (rouge sur la figure 3).

Une fois notre population mise en place, nous pouvons passer à la propagation.

3 Mise en place de la propagation

Dans cette partie, nous avons considéré que la propagation était journalière. Plus exactement, nous avons fait le choix de mettre à jour nos données à chaque tour d'une fonction *propagation* et on déclare (arbitrairement) que l'appel de cette fonction correspond à la propagation sur un jour.

3.1 La fonction propagation_init()

Pour permettre une transimission plus pratique des données d'un jour à l'autre nous mettons en place une structure nommée *Resultat_propagation* [1]. On transmet entre autres d'un jour à l'autre, les identifiants des personnes infectées, la durée de leur infection, les identifiants des personnes décédées, la population et les états des personnes composant la population entre autres.

Cette fonction prend comme argument un vecteur de personnes, un vecteur d'états et une liste contenant les identifiants des personnes infectées tous fournis par la création d'une population. Dans cette fonction, on initialise toutes les variables de comptage d'une variable R, qui est en fait une structure *Resultat_propagation*. Ensuite nous transmettons la population et les états à la variable R puis nous initialisons un vecteur $infectes$ qui va contenir les identifiants des personnes infectées. La nécessité de basculer d'une liste au vecteur vient du fait qu'à priori, et c'est le but de notre démarche, le nombre d'infectés va grandir au fur et à mesure de la simulation et donc la recherche d'un individu infecté se doit d'être optimale.

Enfin, la fonction retourne la variable R.

3.2 La fonction propagation()

L'idée de cette fonction est de récupérer la variable R du temps d'avant et de mettre à jour son contenu. Le premier appel de la fonction *propagation*() prend donc en argument la variable retournée par la fonction *propagation_init*().

1. Le détail de cette structure est à retrouver dans le fichier types.hpp

En plus d'une variable R la fonction de propagation prend en argument d'autres variables :

— Un taux de contagion (double compris entre 0 et 1)

— Un taux de mortalité (double compris entre 0 et 1)

— Un taux de gravité (double compris entre 0 et 1)

— Un age seuil (int)

Le taux de contagion permet de définir à partir de quel moment une personne infectée peut en infecter une autre. Par exemple, si une personne A est infectée et qu'elle compte parmis ses voisins une personne B non-infectée, alors B n'est infectée que si le coefficient de la relation qu'à A avec B est supérieur au taux de contagion. Une fois qu'une personne est infectée, elle dispose de 5 jours avant d'être considérée comme un cas grave. À chaque fois qu'une personne est infectée, on effectue un tirage uniforme entre 0 et 1 pour qu'elle ait une chance d'être totalement remis de la maladie. Si ce tirage est superieur au taux de gravité alors la personne est guéri et ne peut plus être infectée sinon elle reste infectée.

Si une personne est toujours infectée après 5 jours, elle peut potentiellement en mourir. Deux cas de figure se présente. Supposons que la personne concernée ait un âge inferieur à l'âge seuil. Dans ce cas, on effectue un tirage uniforme entre 0 et 1. Si ce tirage est inferieur au taux de mortalité alors la personne décède. Sinon elle passe un jour de plus en tant que cas grave. Supposons que la personne concernée ait un âge supérieur ou égal à l'âge seuil. On effectue alors un tirage uniforme entre 0 et 1. Si ce tirage est inferieur à deux fois le taux de mortalité alors la personne décède. Sinon elle passe un jour de plus en tant que cas grave. On note alors qu'une personne qui est considérée comme cas grave reste un cas grave, donc a des séquelles, si elle ne décède pas.

La nécessité d'instaurer un âge seuil vient du fait que l'on a constaté que la maladie du Covid-19 est plus présente, sous des formes graves donc mortelles, chez une population d'un âge plus avancé.

4 Résultats et analyse

4.1 Résultats

Nous avons exécuter le script *test.cpp*, avec les données numériques suivantes :

— taux de contagion = 0.51

— taux de mortalité = 0.1237

— taux de gravité = 0.3

— age seuil = 64

Voici les résultats que nous avons obtenus sur 1 an :

Pour une population d'un million de personnes, nous avons pour 2 infectés au départ, un total de 249 contaminations dont 133 graves. Il y a eu 26 décès, dont 14 hommes et 12 femmes. Pour cette taille de population, les contaminations représentent 0.0249% et les décés représentent 0.000026% de la population.

Pour une population de 100000 personnes, nous avons pour 2 infectés au départ, un total de 9448 contaminations dont 4186 graves. Il y a eu 700 décès, dont 370 hommes et 330 femmes. Pour cette taille de population, les contaminations représentent 9.448% de la population et les décés représentent 0.7% de la population.

En exécutant l'algorithme, pour des populations de $500, 1000, 1500, 2000, 3000, 5000, 10000, 25000, 100000, 1000000$ d'individus, on obtient la complexité en temps de notre algorithme.

4.2 Analyse

On peut donc conclure (voir figure 4) que notre algorithme de propagation a une complexité linéaire. On aurait pû améliorer sa complexité en utilisant notamment pour le stockage des individus infectés un arbre AVL et autres structure de données mais la difficulté d'implémentation constitue un frein au projet.

On remarque d'après nos résultats que les proportions de contaminations pour une population d'un million d'individus et une population de 100000 personnes diffèrent grandement. À titre d'exemple, d'après wikipédia, sur la période mars 2020 à mars 2021, il y a eu en France, 4644423 contaminations, ce qui représente $6,9\%$

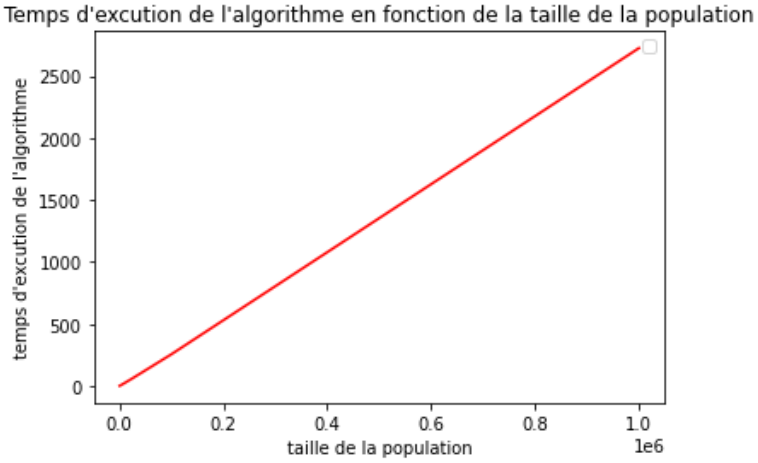

FIGURE 4: Complexité en temps de l'algorithme de propagation

de la population du pays. Et sur la même période, on compte en France, 95640 décès soit 0.14% de la population.

Les chiffres obtenus avec notre algorithme pour une population de 100000 personnes sont du même ordre de grandeur qu'à l'échelle de la population française sur la période mars 2020 à mars 2021. Toutefois, nous avons pris, par exemple, un taux de mortalité bien supérieur à ce qu'il est réellement à l'échelle de la population française. On explique donc ces résultats par le caractère très sommaire de notre algorithme face à la difficulté que représente la modélisation d'une épidémie. En effet, nous avons considéré que chaque personne est en contact avec au plus 4 personnes ce qui n'est pas le cas par exemple dans la vie quotidienne. Nous avons ensuite pris en compte l'âge mais nous n'avons pas pris en compte les facteurs de comorbidités et les maladies respiratoires qui sont des facteurs d'aggravation de la maladie du Covid-19 en cas d'infection. Tous ces facteurs mis bout à bout font que notre algorithme

5 Conclusion

Pour conclure, nous pouvons constater que tous nos choix de modélisation, dont les principaux sont détaillés en fin de partie 4.2, tendent, et ce n'est pas une surprise, à nous éloigner des résultats de la réalité. Cependant ce projet a été l'occasion de travailler sur l'implémentation de structures de données et d'avoir pleinement conscience de l'importance de la complexité d'un algorithme.

Annexe

Quelques points sur les fichiers C++ envoyés...

— *RND.hpp* et *RND.cpp* : déclaration et définition d'une fonction de tirage aléatoire.

— *declaration_fcts.hpp* et *fcts.cpp* : déclaration et définition des fonctions $gen\_age(), propagation\_init(), propagation()$

— *liste.hpp* et *noeud.hpp* permettent de créer l'objet *liste* qui est une liste chaînée.

— *personne.hpp* et *personne.cpp* permettent de créer l'objet *personne*.

— *population.hpp* et *population.cpp* permettent de créer l'objet *population*.

— *types.hpp* définit la structure *Resultat_propagation*.

— *test.cpp* est le script d'exécution de l'algorithme de propagation.

Attention, pour changer la taille de la population, il faut directement modifier N dans le fichier *population.cpp*. Cela ne marche que pour des populations dont la taille est multiple de 100. Lorsque $N \leqslant 1000$, il faut prendre $nb\_l\_grille = 100$ et lorsque $N > 1000$, il faut prendre $nb\_l\_grille = 1000$.