Compléments d'informatique

Projet 2
Testing COVID à l'ULiège

1er décembre 2020

Dans ce projet, on se propose d'implémenter un simulateur simplifié de l'évolution de l'épidémie au sein de l'université, avec pour objectif d'utiliser ce simulateur pour évaluer l'impact du testing hebdomadaire massif tel qu'il avait été organisé à l'ULiège en début de semestre. L'objectif pédagogique principal est de vous faire manipuler des types de données (liste liées, dictionnaires, graphes) et d'apprendre à les combiner pour obtenir une solution (efficace) à un problème.

Ce projet est à réaliser par **groupe de deux étudiants maximum**. La date limite de remise du projet sur la plateforme de soumission est indiquée dans Ecampus.

1 Principe général

L'objectif du projet est de simuler l'évolution de l'épidémie de COVID au sein d'une population d'étudiants, de manière à pouvoir évaluer l'intérêt du testing (voir les expérimentations à faire avec votre code dans la section 5). Pour cette simulation (très simplifiée), on prendra en compte les cours que suivent les étudiants chaque semaine pour déterminer la probabilité que deux étudiants s'infectent l'un l'autre et on utilisera cette probabilité pour propager le virus entre les étudiants. Le programme qu'on vous demande d'écrire prendra comme entrée un fichier donnant pour chaque étudiant la liste de ses cours sur le semestre courant et devra fournir en sortie un fichier contenant l'évolution du nombre d'étudiants dans différentes catégories prédéfinies (voir plus bas), moyennées sur plusieurs simulations. Pour réaliser cela, vous devrez passer par les trois étapes suivantes :

— A partir du fichier avec la liste des cours par étudiants, vous construirez une structure de données offrant un accès efficace à l'information sur le programme de cours complet (fichiers `program.h/.c`).

— A partir de la structure ainsi générée, vous construirez un graphe entre étudiants, dont les arêtes seront pondérées par les probabilités de contamination entre eux (fichiers `program2graph.h/.c`).
— Le simulateur proprement dit prendra en entrée le graphe construit à l'étape précédente pour générer les résultats de simulation (fichiers `simulation.h/.c`).

Le principe de chaque étape et les fonctions à implémenter sont détaillés dans la section 3 fichier par fichier. Le modèle simplifié de propagation du virus qui devra être implémenté dans le fichier `simulation.c` est décrit dans la section suivante.

2 Modèle de propagation du virus

Le modèle de propagation du virus qu'on propose d'utiliser est un modèle dit compartimental [1] en temps discret. A un certain instant t, chaque individu (un étudiant dans notre cas) est dans l'une de ces 5 catégories :

— Susceptibles (S) : des individus sains n'ayant jamais été infectés par le virus et donc susceptibles de l'attraper.
— Exposés (E) : des individus ayant contracté le virus mais pas encore contagieux et n'ayant pas de symptômes.
— Infectieux non isolés (I) : des individus infectés et contagieux, mais non isolés (par exemple, parce qu'ils sont asymptomatiques).
— Infectieux en quarantaine (Q) : des individus infectés et contagieux mais isolés (par exemple, parce qu'ils présentent des symptômes ou parce qu'ils ont été testés positifs).
— Immunisés (R) : des invidus ayant été infectés précédemment mais guéris et maintenant immunisés contre toute réinfection.

Pour déterminer l'état de chaque individu de la population au temps $t + 1$ à partir de l'état au temps t, on appliquera les règles suivantes (séquentiellement, dans cet ordre précis) :

1. Les individus E deviennent I ou Q : I avec une probabilité α et Q avec une probabilité $1 - \alpha$.
2. Les individus I et Q, en ce compris les individus E passés I ou Q à l'étape précédente, passent dans la catégorie R avec une probabilité μ.
3. Les individus I passent dans la catégorie Q avec une probabilité γ.
4. Chaque individu dans la catégorie I infecte ses voisins S avec une certaine probabilité expliquée ci-dessous. Un voisin S infecté de cette manière devient E.

Le paramètre α représente la probabilité qu'une personne infectée ne se mettent pas en quarantaine (parce qu'elle est asymptomatique ou qu'elle refuse sciemment de s'isoler), γ

1. `https://fr.wikipedia.org/wiki/Modèles_compartimentaux_en_épidémiologie`

est la probabilité qu'une personne I se mette en quarantaine suite à un résultat positif à un test, μ est la probabilité de guérison d'un individu infecté (I ou Q).

Probabilité d'infection entre étudiants. L'étape 4 de l'algorithme de propagation requière de déterminer pour chaque paire d'étudiants la probabilité que l'un infecte l'autre. Cette probabilité sera calculée selon l'idée que plus ces étudiants ont des cours en commun et plus ces cours concernent une population faible d'étudiants, plus ils ont des chances d'être en contact et donc de se passer le virus. On supposera que ces probabilités sont symétriques et on les calculera comme suit pour une paire d'étudiants s_1 et s_2 :

$$P^I(s_1, s_2) = 1 - \prod_{i=1}^{N^c} \left[P_i^C(s_1, s_2)(1-\beta) + (1 - P_i^C(s_1, s_2)) \right], \tag{1}$$

où N^C est le nombre total de cours, $P_i^C(s_1, s_2)$ est la probabilité que les étudiants entrent en contact lors du cours i et β est la probabilité qu'ils s'infectent mutuellement s'ils sont en contact. Si un des deux étudiants s_1 ou s_2 ne suit pas le cours i, alors $P_i^C(s_1, s_2) = 0$. Si s_1 et s_2 suivent tous les deux le cours i, on calculera $P_i^C(s_1, s_2)$ comme suit :

$$P_i^C(s_1, s_2) = \max\{1.0, \frac{4}{N_i^S - 1}\}, \tag{2}$$

où N_i^S est le nombre d'étudiants qui suivent le cours i.

La définition (2) est motivée par l'hypothèse (simpliste) qu'un étudiant interagit avec en moyenne 4 étudiants lors d'un cours et que ces 4 étudiants sont choisis au hasard parmi tous les étudiants exceptés lui-même. Le produit dans le membre de droite de la définition (1) est la probabilité que les étudiants ne s'infectent l'un l'autre dans aucun des cours. Dans ce produit n'interviennent que les cours que les deux étudiants suivent ensemble, puisque pour les autres cours la probabilité de contact vaut 0.

Limitations du modèle. Le modèle simple proposé ici fait des hypothèses assez fortes qui ont peu de chance d'être vérifiées en pratique, principalement pour faciliter l'implémentation. En plus de celles mentionnées ci-dessus, on peut mentionner le fait qu'on suppose qu'aucune infection entre étudiants ne peut avoir lieu en dehors des cours ou ne peut provenir d'une personne extérieure (si ce n'est au temps $t = 0$ pour initier l'épidémie). Les probabilités (2) sont supposées fixes et ne tiennent donc pas compte des étudiants Q qui n'assistent pas au cours pour cause d'isolement.

La probabilité de contact tient compte également de la totalité des cours suivis par les étudiants sur une semaine. On supposera donc qu'un pas de temps dans la simulation correspond à une semaine. Cela coïncide également avec le fait que le testing est réalisé à une cadence hebdomadaire. Le paramètre γ sera donc fonction de la proportion d'individus participant au testing par semaine et le modèle suppose que ces individus sont choisis aléatoirement parmi tous les individus.

3 Implémentation

Les différents fichiers à implémenter sont décrits ci-dessous. La section 3.4 décrit des types de données dont l'implémentation vous est fournie et que vous pouvez utiliser pour votre implémentation.

3.1 Création du programme de cours

Les fonctions des fichiers `program.c/h` doivent permettre de charger le programme de cours à partir du fichier d'entrée et de le stocker dans une structure. Le fichier est constitué sur chaque ligne du matricule de l'étudiant (`S` suivi d'un nombre sur 6 chiffres) et la liste des cours auxquels il prend part. Comme à l'ULiège, l'identifiant du cours est un code en 4 lettres indiquant la matière et un code en 4 chiffres indiquant le numéro du cours. Le programme de cours sera représenté par une instance d'un type (opaque) `Program` que vous devrez implémenter vous-mêmes. Le choix de votre structure est libre mais vous devez la choisir de manière à ce que les opérations ci-dessous soient toutes aussi efficaces que possible.

Fonctions de l'interface. Les fonctions à implémenter dans `program.c` sont les suivantes :

`Program *prLoadProgramFromFile(char *filename)` : charge les informations dans le fichier `filename` et renvoie un pointeur vers une structure programme contenant ces informations.

`void prFree(Program *p)` : libére la mémoire prise par le structure.

`List *prGetCourseList(Program *p)` : renvoie la liste des identifiants de tous les cours (sans doublons).

`List *prGetStudentList(Program *p)` : renvoie la liste des matricules de tous les étudiants (sans doublons).

`List *prGetStudentCourseList(Program *p, char *student)` : renvoie un pointeur vers une structure de type `List` (voir 3.4) contenant la liste des cours (leur identifiant) pour l'étudiant `student`, NULL si l'étudiant n'existe pas.

`List *prGetCourseStudentList(Program *p, char *course)` : renvoie un pointeur vers une structure de type `List` contenant la liste des étudiants (leur matricule) pour le cours `course`, NULL si le cours n'existe pas.

`int prGetCourseNbStudents(Program *p, char *course)` : renvoie le nombre d'étudiants qui assistent au cours `course`.

`int prGetStudentNbCourses(Program *p, char *student)` : renvoie le nombre de cours au programme de l'étudiant `student`.

`List *prPrint(Program *p)` : affiche des informations sur le programme. Vous pouvez décider de ce que vous voulez afficher comme information. Cette fonction est fournie pour vous permettre de débugger votre code.

3.2 Création du graphe

La fonction qui va créer le graphe à partir du programme doit être implémentée dans le module `program2graph.c/.h`. Pour générer ce graphe, vous devez utiliser le type de données fourni dans le module `graph.c/.h` qui est mis à votre disposition (voir section 3.4).

Fonctions de l'interface. Les deux fonctions à implémenter dans `program.c` sont les suivantes :

`double computeStudentsInfectionProb(Program *p, char *st1, char *st2, float beta)` : calcule la probabilité que les étudiants s'infectent l'un l'autre en utilisant la formule (1) avec la valeur de β fournie en argument.

`Graph *programToGraph(Program *program, float beta)` : construit un graphe entre étudiants sur base des probabilités calculées par la fonction précédente. Ce graphe doit contenir une arête entre deux étudiants si et seulement si leur probabilité d'infection mutuelle est strictement supérieure à 0 (c'est-à-dire lorsqu'ils partagent un cours). Le poids de l'arête dans la structure est donnée par la probabilité d'infection mutuelle.

3.3 Simulateur

Le module `simulation.c/.h` s'occupe de réaliser une simulation de l'évolution de l'épidémie sur base d'un graphe. Ce module implémente un type de données (opaque) `Simul` qui enregistre l'état des individus de la population, ainsi que le graphe de leurs interactions. Dans les fonctions de l'interface les nœuds du graphe qui sert de base à la simulation sont identifiés par des entiers de 0 à $N-1$ si le graphe contient N nœuds (comme dans le module `graph.c/.h`). Les états sont codés dans les fonctions par les constantes `S`, `E`, `I`, `Q` et `R` définies par une énumération dans le fichier `simulation.h`.

Fonctions de l'interface. Les fonctions à implémenter dans `simulation.c` sont les suivantes :

`Simul *SimCreateSimulation(Graph *g)` : renvoie un pointeur vers une structure de type `Simul`. Tous les individus doivent être dans l'état S.

`void simFree(Simul *sim)` : libère la mémoire prise par `sim`.

`void simChangeNodeState(Simul *sim, int i, int newstate)` : change l'état du nœud i pour le nouvel état `newstate`.

`int simGetNodeState(Simul *sim, int i)` : renvoie l'état du nœud i.

`int simGetNbNodes(Simul *sim)` : renvoie le nombre de nœuds dans la simulation.

`int simGetNbInStates(Simul *sim, int state)` : renvoie le nombre de nœuds dans l'état `state`.

`int simPrintStat(Simul *sim)` : affiche des informations sur la simulation. Vous pouvez décider de ce que vous voulez afficher comme information. Cette fonction est fournie pour vous permettre de débugger votre code.

`int simInitSim(Simul *sim);` : initialise la simulation en remettant à l'état S tous les nœuds et en mettant à E un nœud au hasard. L'indice de ce nœud est renvoyé par la fonction.

`void simOneStepSimulation(Simul *sim, float alpha, float gamma, float mu)` : effectue une étape de simulation telle que décrite dans la section 2, en utilisant les valeurs de α, γ et μ données en argument. Cette fonction doit mettre à jour les états des nœuds en fonction des règles de propagation du virus expliquées dans la section 2.

`void simAverageSimulations(Simul *sim, float alpha, float mu, int nbSteps, int nbSimul, float results[nbSteps][NBSTATES])` : effectue `nbSimul` simulations de `nbSteps` pas en partant chaque fois d'un nouvel état aléatoire généré par `simInitSim`. Le tableau `results` devra contenir pour chaque pas de temps la moyenne sur les `nbSimul` simulations du nombre de nœuds dans chaque catégorie. Une simulation sera obtenue en appliquant `nbSteps`-1 fois la fonction `simOneStepSimulation` à partir d'une état généré par `simInitSim`.

3.4 Codes fournis

Type de données. Pour implémenter les différentes fonctions, nous vous fournissons les modules suivants :

- `list.c/.h` : qui implémente une liste liée permettant de stocker des pointeurs sur `void`. La structure utilisée pour `List` est non opaque. Vous pouvez donc utiliser les champs de cette structure dans vos différentes fonctions. On vous impose de l'utiliser pour les valeurs de retour de certaines fonctions de `program.c`.
- `dict.c/dict.h` : qui implémente un dictionnaire avec des clés sous la forme de chaîne de caractères et des données associées sous la forme de pointeurs sur `void`. L'implémentation utilise une table de hachage qui suit l'implémentation présentée dans le cours.
- `graph.c/.h` : qui implémente un type de données graphe. Le graphe est non dirigé et les arêtes pondérées. Le module offre aussi la possibilité d'annoter les nœuds avec des valeurs entières. Dans cette structure les nœuds sont codés par des entiers compris entre 0 et $N-1$ si N est le nombre de nœuds.

Les fonctions de l'interface sont (brièvement) commentées dans les fichiers d'entête. Le code de ces modules étant fournis, nous ne donnerons pas ici plus de détails sur leur utilisation. Quelques explications seront néanmoins fournies pendant le cours.

Données et fonction de test. Un fichier `test_simul.c` permettant de réaliser une simulation sur base des paramètres en ligne de commande et de créer un fichier de résultats vous est fourni pour tester votre code. Des fichiers contenant des listes d'étudiants de différentes tailles vous sont également fournis pour pouvoir réaliser des tests. Ces fichiers sont nommés `program_XXX_YYY.txt` où `XXX` est le nombre d'étudiants et `YYY` est le nombre total de cours. Dans ces fichiers, le nombre de cours par étudiant varie entre 3 et 10.

Par exemple, l'instruction suivante dans un terminal (après un `make`) :

```
./test_simul program_1000_200.txt 0.8 0.0 0.2 0.3 25 500 > res.csv
```

permet de lancer une expérience avec $\alpha = 0.8$, $\gamma = 0,0$, $\beta = 0,2$, $\mu = 0,3$, 25 pas de temps et une moyenne sur 500 simulations. Le résultat est écrit dans un fichier nommé `res.csv` qui peut être ouvert dans la plupart des tableurs (excel, openoffice, etc.).

4 Conseils d'implémentation

`program.c` : La fonction principale de ce module est la fonction `prLoadProgramFromFile`. Pour charger les données du fichier, vous pouvez vous inspirer du code du transparent 353 de la partie 6 du cours théorique. On vous demande de rendre l'accès aux différentes informations de la structure le plus efficace possible. Pour information, notre implémentation de ce module fait approximativement 150 lignes.

`program2graph.c` : Vous disposez de toutes les fonctions d'accès nécessaires dans `program.h` pour implémenter la fonction `computeStudentsInfectionProb`. Nous ne mettons pas de contraintes en terme de complexité sur cette fonction. La fonction `programToGraph` peut s'implémenter assez facilement en se basant sur la fonction `computeStudentsInfectionProb` et les fonctions mises à votre disposition dans `graph.h`.

Notre implémentation de `program2graph.c` fait approximativement 60 lignes.

`simulation.c` : Pour implémenter ce module, vous aurez besoin de pouvoir annoter les nœuds du graphe en fonction de leur état. Vous pouvez utiliser pour cela les fonctions `graphSetVertexLabel` et `graphGetVertexLabel` du module `graph.h`, mais d'autres mécanismes sont possibles. Pour simuler un événement qui doit se produire avec une probabilité p, il suffit de générer [2] un nombre aléatoire r selon une loi uniforme entre 0 et 1 et de provoquer l'événement si $r < p$.

Notre implémentation de ce fichier fait approximativement 130 lignes.

5 Analyse de complexité

On vous demande dans le rapport de fournir les temps de calcul des deux fonctions suivantes :

— `prLoadProgramFromFile`

— `program2graph`

lorsqu'elles sont utilisées sur les fichiers `program_1000_200.txt`, `program_2000_200.txt`, `program_4000_200.txt`, `program_8000_200.txt`. Donnez ensuite les complexités théoriques de ces deux fonctions en fonction de N le nombre d'étudiants, M le nombre de cours par étudiants (que vous pouvez supposer constant) et L le nombre total de cours. Si vous avez utilisé une table de hachage, donnez la complexité dans le pire cas et dans le cas où le comportement de la table de hachage est optimal (toutes les opérations sont en $O(1)$). Précisez du quel cas vous êtes le plus près avec vos résultats empiriques.

6 Expérimentations

Maintenant que votre code est terminé, vous pouvez l'utiliser pour aider le recteur à décider de l'intérêt de se lancer dans le testing hebdomadaire des étudiants. On fixera les valeurs suivantes par défaut pour les paramètres de propagation du virus : $\alpha = 0,8$, $\beta = 0,2$, et $\mu = 0,3$. $\alpha = 0,8$ signifie que 20% des étudiants infectieux s'isolent, $\beta = 0,2$ qu'un étudiant a une chance sur 5 d'infecter un étudiant lors d'un contact et $\mu = 0,3$ signifie qu'un étudiant a 30% de chance d'être guéri au début de chaque période.

En fixant ces valeurs de paramètres par défaut, on testera les trois conditions suivantes pour ce qui est du testing :

2. En C, vous pouvez le faire avec l'instruction `(float)rand()/(float)RAND_MAX`

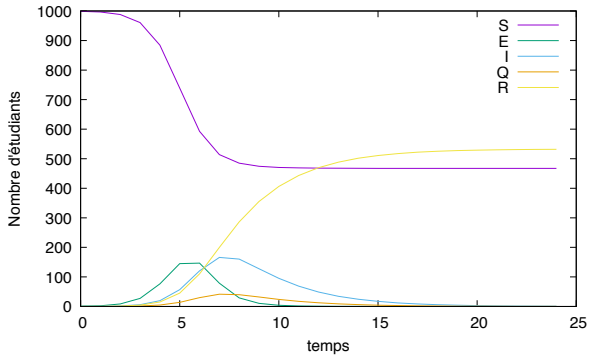

FIGURE 1 – Evolution du nombre moyen d'étudiants dans chaque catégorie en fonction du temps sans testing

— Condition 1 (pas de testing) : $\gamma = 0,0$
— Condition 2 (testing depuis le début) : $\gamma = 0,3$
— Condition 3 (testing retardé) : $\gamma = 0,0$ pour $t < 5$ et $\gamma = 0.3$ pour $t \geq 5$.

Dans la première condition, il n'y a pas de testing ($\gamma = 0,0$). Dans la deuxième condition, on suppose que 50% des étudiants participent au testing et que la sensibilité du test est 60% (60% des étudiants I testés sont positifs lorsqu'ils font le test), ce qui nous donne $\gamma = 0,5 \cdot 0,6 = 0,3$. Dans la troisième condition, les conditions de testing sont les mêmes mais on supposera que le testing ne peut démarrer que la cinquième semaine pour des raisons logistiques (comme ça avait été le cas à l'université).

Dans ces trois conditions, utilisez votre code sur le fichier `program_1000_200.txt` pour calculer les nombres d'étudiants dans les 5 catégories sur 25 semaines et en faisant des moyennes sur 500 simulations pour stabiliser les résultats. Tracez sur trois graphes séparés l'évolution des grandeurs obtenues et collectez ces 3 graphes dans un fichier unique `graphes.pdf`. La courbe que vous devriez obtenir dans la première condition devrait ressembler à celle de la figure 1, qui indique qu'en moyenne plus de la moitié des étudiants auront contracté le virus au moment de la fin de l'épidémie. En comparaison, les deux autres graphes qu'on vous demande de générer vous montreront combien d'infections pourront être évitées grâce au testing.

7 Soumission

Le projet doit être soumis via la plateforme de soumission sous la forme d'une archive (nommée impérativement `projet2.zip`, `projet2.tar.gz` ou `projet2.tgz`) contenant les fichiers suivants :

— `program.c`
— `program2graph.c`

— `simulation.c`
— `rapport.txt`
— `graphes.pdf`

Les fichiers `program.h`, `program2graph.h` et `simulation.h`, ainsi que les types fournis, qui ne doivent pas être modifiés, ne doivent pas être fournis dans l'archive. S'ils le sont malgré tout, ils seront remplacés par nos fichiers lors de la compilation de votre code, ce qui risque de poser problème si vous les avez modifiés. Si besoin, vous pouvez soumettre des modules supplémentaires sous la forme de fichiers `.c/.h` dont vous auriez besoin pour votre implémentation. Ces fichiers doivent cependant être inclus dans les fichiers `.c` des autres modules, pas dans les fichiers `.h`.

Le fichier `rapport.txt` contiendra les résultats demandés dans la section 5 et devra respecter scrupuleusement le format du fichier fourni (il sera analysé automatiquement). Le fichier `graphes.pdf` contiendra les graphes demandés dans la section 6. Notez que si le code que vous remettez ne permet vraisemblablement pas de faire l'analyse et de générer les graphes, ces derniers ne seront pas corrigés.

Dans `rapport.txt`, en cas de travail en groupe, il vous est demandé d'indiquer la **contribution de chacun au projet en pourcentage**. Ces valeurs doivent être acceptées par les deux étudiants du groupe lors de la soumission. Comme annoncé au début de l'année, la cote d'un étudiant pourrait être rabaissée en cas de participation trop faible aux projets.

Vos fichiers seront compilés avec les flags de compilation habituels (`-std=c99 -pedantic -Wall -Wextra -Wmissing-prototypes`), qui ne devront déclencher aucun avertissement lors de la compilation, sous peine d'affecter votre cote.

Toutes les soumissions seront soumises à un programme de détection de plagiat. En cas de plagiat avéré, les étudiant concernés se verront affecter une cote nulle au projet (au minimum).

Bon travail !