ICC MT & EL **Date limite de téléversement : dimanche 31 mai 2020 23h59**

Travail théorique *individuel*

Compléter ce document avec un logiciel de traitement de texte puis produire un PDF pour le téléversement

NOM : KOCKISCH

Prénom : Matthias

Numéro SCIPER : 303000

Notations du pseudocode : étant donnée le lien avec le projet qui comprend l'usage de pointeurs nous autorisons d'écrire du pseudocode qui empreinte des concepts ou structures de données du C++ (attribut de structure de donnée, pointeur, vector) ou des expressions du C++ (affectation, incrémentation, opérateur d'accès à un attribut…). Veillez néanmoins à exprimer les algorithmes avec concision avec ces mots-clefs. Conservez toujours la même convention pour les indices (au choix : de 1 à nb_éléments, ou de 0 à nb_éléments -1).

Exercice 1 : Vérification de conditions supplémentaires sur la ville (4 pts)

Nous supposons que le fichier décrivant la ville *contient au moins un quartier* et ne contient pas d'erreur détectable selon le rendu1 du projet. Cela ne suffit cependant pas pour lancer une simulation intéressante. En effet un fichier correspondant à la figure 1 est correct selon les règles du projet mais présente un problème pour la simulation : un quartier n'est pas connecté au reste de la ville. L'autre cas qui pose un problème selon les règles du projet est illustré avec la Figure 2 : un quartier Production ne permet pas d'atteindre trois autres quartiers.

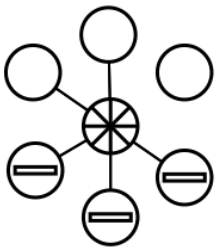

Fig. 1 : un nœud n'est pas connecté au reste de la ville

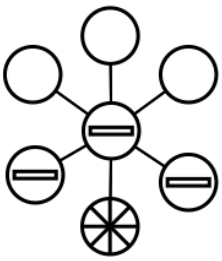

Fig. 2 : un nœud Production empêche d'atteindre d'autres nœuds de la ville

1.1) A partir de ce que vous avez vu en cours, proposez une méthode pour détecter de tels cas. La question générale à répondre est : ***existe-t-il au moins un quartier qui n'est pas atteignable par un autre quartier ?***

Précision : on ne se préoccupe pas de savoir quel(s) quartier(s) pose(nt) problème, on veut seulement une réponse par **oui** ou par **non** qui permettra de commencer une simulation ou pas.

À l'aide d'une double boucle for, on applique l'algorithme de Dijkstra sur chaque nœud en recherche de chaque nœud situé après le nœud de départ dans l'ordre de sauvegarde (si on a 4 nœuds, le 1er nœud cherche le 2^{e}, 3^{e}, puis 4^{e}, ensuite le 2^{e} cherche le 3^{e} puis 4^{e} et finalement le 3^{e} cherche le 4^{e}). Ainsi, toutes les connexions sont

testées. Si un des appels de la fonction Dijkstra ne peut pas arriver au nœud recherché, la fonction principale renvoit le signal « Oui ».

Si on veut uniquement vérifier que chaque nœud Logement peut accéder à tous les nœuds Production, il n'est que nécessaire de rechercher tous les nœuds Production à partir de chaque nœud Logement avec l'algorithme Dijkstra.

Exercice 2 : Définition d'une *destination aléatoire* pour chaque pendulaire (12pts)

Nous supposons à partir de maintenant que toutes les conditions exprimées dans l'exercice 1 sont remplies. Il reste un problème concernant le choix du quartier Production de destination pour chacun des pendulaires.
Si on choisit le quartier Production le plus proche du nœud Logement du pendulaire alors le risque est grand que la capacité des nœuds Productions ne soit pas suffisante pour le nombre des pendulaires les plus proches. Par exemple pour la ville de la figure 3, tous les pendulaires devraient se rendre vers les deux quartiers Production les plus proches des quartiers de Logement et visiblement ces deux quartiers ont une capacité insuffisante.

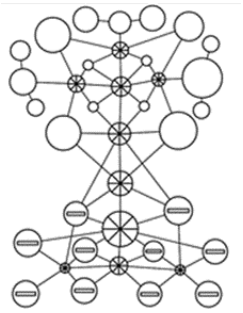

Fig. 3 : exemple de ville pour laquelle le plus court chemin Logement -> Production produit un nombre trop grand de pendulaires pour les deux nœuds Production les plus proches des Logements.

C'est pourquoi nous posons que :
- chaque **pendulaire** mémorise son quartier de Logement de **départ** et son quartier Production de **destination**
- un algorithme doit choisir la destination de chaque pendulaire ***de manière aléatoire*** (précision ci-dessous) en veillant à ne pas dépasser la capacité des nœuds de production.

Rappel : un nœud peut mémoriser des données supplémentaires

Pour la mise en œuvre du hasard dans le choix de la destination, on suppose qu'on dispose d'une fonction **hasard(min, max)** qui donne à chaque appel un nombre entier dans l'intervalle [min, max] avec une probabilité uniforme. On pose que cet appel a un coût constant.

2.1) Quelle est la condition à remplir sur la capacité des nœuds de Production pour garantir d'avoir une solution ?

Vu que le nombre de pendulaires est égal à la moitié de la capacité totale des nœuds Logement et que tous les pendulaires doivent pouvoir se trouver dans des nœuds Production en même temps, la condition de réussite est que la capacité totale des nœuds Production soit égale ou supérieure à la moitié de la capacité totale des nœuds Logement de la ville.

2.2) Quel choix faites-vous pour représenter ces deux données de départ de de destination nécessaires pour chaque pendulaire ?

Pour pouvoir modéliser chaque pendulaire, je créé une struct « Pendulaire » contenant deux indices « pdep » et « pdst », qui correspondent respectivement aux indices du nœud de départ et du nœud de destination du pendulaire dans le conteneur des nœuds Production.

2.3) Quel choix faites-vous pour représenter l'ensemble des pendulaires ?

Vu que le nombre de pendulaires est constant durant la simulation, je créé un array « pendulaires » d'objets Pendulaire qui contient la totalité des pendulaires. « pendulaires » est déclaré avec une taille égale à la moitié de la somme des capacités des nœuds Logement. Il faudra bien sûr veiller à avoir affecté des valeurs concrètes à tous les pdep et pdst de l'array avant de les utiliser, sous danger de segmentation fault.

Au départ de la simulation, des pointeurs vers les pendulaires d'un nœud se trouvent dans un vector « npen » qui est un attribut des nœuds Logement et Production.

2.4) Préciser aussi ci-dessous le nom des ensembles de nœuds dont vous avez besoin pour cet exercice et les noms des variables indiquant le nombre d'éléments de chaque ensemble. Par exemple, y a-t-il un seul ensemble qui contient tous les nœuds ? Ou alors, vos nœuds sont-ils répartis selon leur type en plusieurs ensembles ? A vous de décider. Par la suite, *on suppose que ces ensembles sont accessibles en entrée des algorithmes sans se soucier de faire des appels de fonctions* (c'est du pseudocode, pas du code).

« nœuds » contient des pointeurs vers tous les nœuds de la ville. « logements » contient tous les nœuds du type Logement et « productions » contient les nœuds Production. Les sommes des capacités de nœuds Logement et Production sont respectivement nbp_logement et nbp_production. Les nombres de nœuds Logement et Production sont respectivement nb_logement et nb_production.

2.5) Décrire en une à deux phrases ci-dessous comment vous envisagez de résoudre cette tâche en indiquant si vous mémorisez des données supplémentaires dans certains types de nœuds. Puis préciser cette idée en fournissant le pseudocode d'un tel algorithme en complétant selon vos besoins la page suivante (le pseudocode ne doit pas être réparti entre deux pages). Si nécessaire l'algorithme peut être décomposé en fonctions (Principes d'abstraction et de réutilisation).

Les pendulaires sont tous initialisés avec l'array « pendulaires », les pdep sont affectés en parcourant l'ensemble des nœuds Logement et modifiant un nombre de pendulaires égal à la moitié de ce nœud. Ensuite, les pdst sont affectés à l'aide de la fonction hasard qui choisit un nœud Production parmi ceux qui ne sont pas pleins.

```
1  Si nbp_logement/2 > nbp_production
2    Afficher un message d'erreur (pas de capacité)
3    arrêter la simulation
4  vector d'entiers indices_p
5  Pour i de 0 à la taille de productions
6    ajouter i à la fin de indices_p
7  entier total ← 0
8  array d'entiers 0 capacites_p[0..nb_production-1]
9  Pour i de 0 à la taille de logements
10   Pour j de total à logements[i].nbp + total
11     total ← total + logements[i].nbp
12     pendulaires[j].pdep ← i
13     entier h ← hasard(0,indices_p.taille)
14     entier p ← indices_p[h]
15     pendulaires[j].pdst ← p
16     pointeur ptr_p ← adresse de pendulaires[j]
17     Ajouter ptr_p à la fin de logements[i].npen
18     capacite_p[p] ← capacite_p[p]+1
19     Si capacite_p[p] = productions[p]
20       indices_p[h] ← dernier élément de indices_p
21       supprimer le dernier élément de indices_p
```

Exercice 3 : Détermination des plus courts chemins départ -> destination (12 pts)

A ce stade, chaque pendulaire connaît son nœud de départ et son nœud de destination.

Nous imposons que la simulation soit effectuée
selon le plus court chemin départ->destination pour chaque pendulaire.

On demande d'utiliser l'algorithme de Dijkstra présenté en cours ; cette forme générale de l'algorithme trouve le plus court chemin vers tous les nœuds du graphe à partir d'un nœud de départ donné.

3.1) Décrire en trois-quatre phrases ci-dessous comment vous envisagez de résoudre cette tâche en indiquant comment est mémorisé le chemin de chaque pendulaire.

Je lance l'algorithme de Dijkstra entre les nœuds de départ et d'arrivée de chaque pendulaire, et je sauvegarde chaque étape du parcours dans un vector « chemin » qui est un attribut de Pendulaire.

3.2) Ecrire le pseudocode sur la page suivante (le pseudocode ne doit pas être réparti entre 2 pages). Si nécessaire l'algorithme peut être décomposé en fonctions (Principes d'abstraction et de réutilisation). L'algorithme de Dijkstra peut être appelé comme un simple appel de fonction en précisant ses paramètres : nœud de départ, ensemble de nœuds qui est modifié par l'appel.

3.3) Estimer l'ordre de complexité de votre algorithme.

La taille de pendulaires est égale à la moitié de la capacité totale de des nœuds Logement ; la complexité de la boucle « pour » est donc nbp_logement. La boucle « tant que » peut, dans le pire des cas, traverser l'ensemble des nœuds de la ville ; sa complexité est donc le nombre de nœuds. La complexité de mon algorithme est donc nbp_logement x O(AlgD) x nb_noeuds.

```
1  Pour i de 0 à la taille de pendulaires
2    dijkstra(pendulaires[i].pdep, noeuds)
3    entier ind ← pendulaires[i].pdst
4    Tant que (ind != pendulaires[i].pdep)
5      ajouter ind à la fin de pendulaires[i].chemin
6      ind ← nœuds[ind]->parent
```

Exercice 4 : Proposer le pseudocode de la simulation (12 pts)

A ce stade chaque pendulaire connaît sa suite des directions à suivre, c'est-à-dire la suite des nœuds consécutifs entre le nœud de départ et le nœud d'arrivée.

4.1) Nous devons maintenant préciser le choix effectué pour la représentation fine d'une **direction** selon les indications des *sections 3 à 5 du document décrivant le contexte de la simulation*. Quelle structure de donnée choisissez-vous pour représenter une **direction** ? A quel(s) endroit(s) cette information devrait-elle être mémorisée ? Rappel : on peut ajouter des données aux éléments qui représentent le graphe. Préciser quand ces structures de données doivent être initialisées (nommer les états possibles des segments).

Je modélise une direction par un vector 2D (vectors de taille le nombre de passages parallèles dans des vectors de taille le temps de parcours) contenant des pointeurs vers des pendulaires. Ce vector est sauvegardé dans un struct « Direction », contenant aussi les indices du nœud de départ « ddep » et du nœud d'arrivée « ddst », ainsi que le nombre de voies « dnbv » et la longueur « dlon » de la direction. Finalement, ces structs sont rangées dans un array « directions » de taille 2nb_liens (« nb_liens » étant le nombre de liens dans la ville).
Pour initialiser ces directions, j'assume que la totalité des connexions de la ville sont répertoriés dans un array « liens » de structs « Liens » contenant l'uid du nœud de départ « ldep » et d'arrivée « ldst » (comme elles sont représentées dans un fichier .txt fourni au programme Archipelago), ainsi que leur capacité « lcap », longueur « llon » et vitesse de parcours « lvit ».

4.2) Également sur la base des indications des sections 3 à 5 du document décrivant le contexte de la simulation, quelles autres informations faut-il mémoriser pour chaque pendulaire pour pouvoir mettre à jour sa position au cours de la simulation.

Pour mon code, il ne faut ajouter d'attributs à Pendulaire.

4.3) Ecrire le pseudocode de la boucle de mise à jour asynchrone de la simulation sur la page suivante (le pseudocode ne doit pas être réparti entre 2 pages). Si nécessaire l'algorithme peut être décomposé en fonctions (Principes d'abstraction et de réutilisation).

4.4) Estimer l'ordre de complexité d'UN seul passage dans la boucle de la simulation (mise à jour de tous les pendulaires pour une seule seconde) en fonction de la ou des variables que vous jugerez pertinentes.

L'intérieur de la boucle traverse toutes les voies de toutes les directions de la ville, et peut ensuite parcourir toutes les directions. La complexité d'un passage est donc nb_voies x nb_directions.

4.5) Décrire en quelques phrases le complément qu'il faudrait ajouter à cet algorithme pour *détecter* qu'un embouteillage (*traffic jam*) ne permet pas à tous les pendulaires d'atteindre leur destination, même en effectuant cette simulation sur un nombre illimité de secondes.

Si la simulation peut durer un nombre illimité de secondes, c'est que plus aucun changement n'est fait pendant la simulation alors qu'au moins un pendulaire n'est pas arrivé à sa destination finale. Pour identifier une telle situation, il suffit d'utiliser un booléen initialisé à « vrai » à chaque passage de la boucle de la simulation et qui est modifié à « faux » si un élément sauvegardé est modifié. à la fin de la boucle, si ce booléen est toujours égal à « vrai », aucune modification n'a été effectuée et il y a donc un embouteillage permanent.

```
1  array de Direction directions[0..2(nb_liens-1)]
2  Pour i de 0 à nb_liens-1
3    directions[2i].ddep ← find_uid(liens[i].ldep)
4    directions[2i].ddst ← find_uid(liens[i].ldst)
5    directions[2i+1].ddep ← find_uid(liens[i].ldst)
6    directions[2i+1].ddst ← find_uid(liens[i].ldep)
7    S ← floor(liens[i].llon/liens[i].lvit)
8    P ← (liens[i].lcap)/(2*S)
9    Pour j de 0 à 1
10     directions[2i+j].dnbv ← P
11     directions[2i+j].dlon ← S
12 Pour t de 0 à t_max
13   Pour i de 0 à la taille de directions
14     dlon ← directions[i].dlon
15     Pour j de 0 à directions[i].dnbv
16       Si (taille de directions[i][j] = dlon)
17         pdst ← (*directions[i][j][dlon]).pdst
18         Si (directions[i].ddst = pdst)
19           ajouter directions[i][j][dlon] à la fin de
20                                              (*nœuds[pdst]).npen
21           directions[i][j][dlon] ← nullptr
22         Sinon
23           ind1 ← dern. élément de directions[i][j][dlon].chemin
24           ind2 ← av-d. élément de directions[i][j][dlon].chemin
25           d ← 0
26           Tant que (directions[d].ddep != ind1 et
27                     directions[d].ddst != ind2)
28             d ← d+1
29           Pour e de 0 à directions[d].dnbv
30             Si (directions[d][e][0] = nullptr)
31               directions[d][e][0] ← directions[i][j][dlon]
32               directions[i][j][dlon] ← nullptr
33               supr. dern. élém de directions[i][j][dlon].chemin
34               sortir de la boucl
35       Sinon
36         ptr_p ← dernier élément de directions[i][j]
37         ajouter ptr_p à la fin de directions[i][j]
38         avant-dernier élément de directions[i][j]←nullptr
39       Pour k de taille de dlon-1 à 0
40         Si (directions[i][j][k+1] = nullptr)
41           directions[i][j][k+1] ← directions[i][j][k]
42           directions[i][j][k] ← nullptr
43       Si (directions[i][j][0] = nullptr et
44           taille de nœuds[directions[i].ddep].npen != 0)
45         directions[i][j][0] ← dern. élément de
46                                (*nœuds[directions[i].ddep]).npen
47         suppr. dern. élém de (*noeuds[directions[i].ddep]).npen
48
49 find_uid(uid) {
50   Pour i de 0 à taille de nœuds
51     Si ((*nœuds[i]).uid = uid)
52       Renvoyer i
53 }
```