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 en MAJUSCULES : exemple de solution et barème détaillé

Prénom :

Numéro SCIPER :

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-clef. 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 problème selon les règles du projet est illustré avec la Figure 2 : un quartier Production ne permet pas d'atteindre trois autres quartiers.

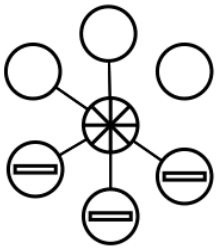

Fig 1 : un nœud n'est pas connecté au reste de la ville

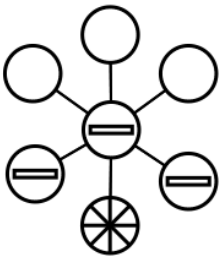

Fig 2 : un nœud Production empêche d'atteindre d'autres nœuds de la ville

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.

Exécuter Dijkstra à partir de n'importe quel nœud ; si ensuite il existe au moins un nœud dont le temps d'accès est infinite_time alors la ville en convient pas pour la simulation.

Alternatives : reconstruire l'ensemble des nœuds à partir de leurs voisins, le nb total est-il le même ? détecter s'il existe au moins un nœud sans voisin (valide dès qu'il y a plus d'un nœud).

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.

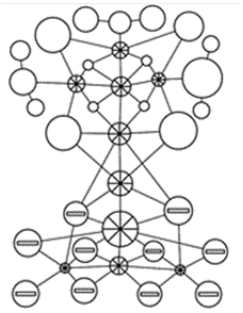

Fig 3 : exemple de ville pour laquelle le plus court chemin Logement -> Production produit un nombre trop grand de pendulaires pour les deux nœuds Production les plus proches des Logements.

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 ? (1pt)

Le total de la capacité des nœuds de production ≥ 50% du total de la capacité des nœuds de logement

2.2) Quel choix faites-vous pour représenter ces deux données de départ de de destination nécessaires pour chaque pendulaire ? (1pt) Nombreux choix possibles / doit être cohérent avec l'algo 2.5.

Par exemple, le nœud de départ = indice du nœud dans un tableau des pointeurs de nœuds de logement,
et le nœud de destination = indice du nœud dans un tableau des pointeurs de nœuds de production.

2.3) Quel choix faites-vous pour représenter l'ensemble des pendulaires ? (1pt)

Dans un unique tableau **tabPend** pour tous les pendulaires, par exemple un vector (construit par l'algorithme 2.5)

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). (1pt) Une solution possible :

Paramètre d'entrée : **tabLog** est le tableau des pointeurs de nœuds de logement de taille **nbLog**
Paramètre d'entrée : **tabProd** est le tableau des pointeurs de noeuds de production de taille **nbProd**
Paramètre de sortie : **tabPend** est le tableau des pendulaires qui est contruit en 2.5) dont le nombre est **nbPend**

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 ré-utilisation). (1pt)

Une boucle parcourt l'ensemble des nœuds logement et pour chaque pendulaire (moitié de capacité) on génère au hasard un indice de nœud de production qui a encore de la place et on ajoute ce pendulaire à tabPend.

Fonctionnalités devant être traitées dans le pseudocode (7pts) de l'exercice 2 :

1. L'ensemble des 50% de la capacité de tous les nœuds Logements doit obtenir un nœud de départ et un nœud de destination (2pts)
2. Chaque pendulaire doit obtenir une destination aléatoirement ; on ne peut pas affecter la même destination obtenue aléatoirement à plusieurs pendulaires à la fois. (1pt)
3. La capacité individuelle de chaque noeud de Production ne doit pas être dépassée ; cette condition doit être vérifiée avant d'affecter un nœud de production comme destination (2pts)
4. Une structure de donnée doit être construite pour créer/mémoriser les pendulaires (1pt)

Le dernier point concerne le style du pseudocode (numérotation de ligne décalée, indentation incorrecte, alignement incorrect d'instruction sur plusieurs lignes), les erreurs de syntaxe, et toute erreur algorithmique non pénalisée dans les 4 fonctionnalités précédentes.

// voici une solution possible ; on ne se préoccupe pas des performances dans la notation

```
1  // tabPend est initialement vide
2
3  // init état des nœuds de Production
4  Pour k de 1 à nbProd
5    tabProd[k]->count ← 0 // compteur pendulaires
6    tabProd[k]->full  ← false // booleen
7
8  // construit un seul tableau de pendulaires
9  Pour i de 1 à nbLog
10   Pour j de 1 à tabLog[i]->capacité/2
11
12     p.depart ← i   // p désigne un pendulaire
13
14     Faire
15       k ← hasard(1,nbProd)
16     Tant que tabProd[k]->full = true
17
18     p.dest ← k
19     tabProd[k]->count++
20     Si (tabProd[k]->count = tabProd[k]->capacité)
21       tabProd[k]->full ← true
22
23     tabPend.push_back(p)
24     nbPend++
25
26 Sortir tabPend
27
```

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 noeuds 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. (3pts)

Tous les pendulaires ayant le même nœud de départ peuvent tirer parti du même appel de l'algorithme de Dijkstra pour trouver le plus court chemin entre le nœud de départ et les autres nœuds de la ville.
Après l'appel de Dijkstra, pour tous ces pendulaires on construit le chemin depuis le nœud destination jusqu'au nœud de départ en suivant le champ parent. Il suffit ensuite d'inverser ce chemin pour obtenir l'itinéraire départ->destination.
L'algo cherche ensuite à valoriser au maximum l'appel de l'algo de Dijkstra en initialisant les autres pendulaires avec les mêmes départ et destination, ou seulement avec le même départ.

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 ré-utilisation). 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. (barème page suivante)

Dijkstra travaille sur l'ensemble complet des pointeurs de nœuds **tabNode** de taille **nbNode**

tabLink est un tableau de pointeurs de Nœud qui commence au nœud destination et qui remonte jusqu'au noeud départ. La fonction **build_parent_list** le construit.

La fonction **build_forward_list** construit **tabConnect** en inversant **tabLink**

En plus des éléments présenté dans l'ex2, **depart** et **dest**, chaque pendulaire mémorise un booleen **done** initialisé à **false**, et son itinéraire **tabConnect**. Ces 2 éléments sont initialisés dans la fonction **build_tabConnect** qui sort **destNode** car c'est utile pour chercher d'autres pendulaires de même destination.

3.3) Estimer l'ordre de complexité de votre algorithme. (2pts)

Il y a **nbLog** appel à l'algo de Dijkstra suivi par la construction du maximum de chemins possibles avec cet appel c'est-à-dire **nbProd** chemin ; puis chaque chemin est inversé :
O(nbLog * (O(Dijkstra) + construction au max de nbProd chemins et leur inversion))

La construction d'un chemin est linéaire en **nbNode** ; au maximum on a **nbProd** chemins. L'inversion du chemin est du même ordre que sa construction, donc on a 2*nbProd*nbNode ce qui est du même ordre que le coût indiqué en $O(nbNode^2)$ indiqué pour l'algo de Dijsktra utilisé en cours.

Donc il reste **O(nbLog * O(Dijkstra)) ~ $O(nbLog*nbNode^2)$**

Fonctionnalités devant être traitées dans le pseudocode (6pts) de l'exercice 3 :

1. L'ensemble de tous les pendulaires (3pts) dispose d'un moyen pour atteindre leur destination
    - Stratégie 1: mémoriser un chemin par pendulaire qui consiste en un ensemble de **pointeurs sur les nœuds** consécutifs entre le départ et la destination
        - Variante possible sur le partage de chemin entre plusieurs pendulaires
    - Stratégie 2 : mémoriser un chemin par pendulaire qui consiste en un ensemble des **liens** consécutifs entre le départ et la destination
        - Variante possible sur le partage de chemin entre plusieurs pendulaires
    - Stratégie 3 : mémoriser une **table de routage** au niveau de chaque noeud

2. C'est le plus court chemin départ-> destination de chaque pendulaire qui est construit (1pt)
    - Usage de Dijkstra avec le départ du pendulaire comme origine et travaillant sur l'ensemble des nœuds de la ville ; des variantes suffisamment bien détaillées peuvent être acceptées (ex : paramètres).
    - Ensuite retrouver l'information de destination pendulaire dans la ville

3. (2 pts) Le chemin est mémorisé dans le sens départ->destination [exception acceptée : OK si l'exercice 4 sait gérer un chemin mémorisé dans le sens destination->départ)]
    - L'algo de Dijkstra donne le moyen de construire facilement le chemin **destination -> départ** en ajoutant l'information du champ parent dans un vector ; il faut inverser **ce chemin** sauf si l'exo 4 sait gérer un chemin dans le sens destination->depart.
    - Si le chemin est mémorisé avec **push_front**(..) il n'a pas besoin d'être inversé **MAIS** la réponse aux questions doit documenter le fait d'utiliser un **deque** car cette opération n'existe pas pour un vector.
        - On accepte insert() sur un vector seulement si l'appel contient les 2 paramètres comme pour un vector C++.

Le dernier point concerne le style du pseudocode (numérotation de ligne décalée, indentation incorrecte, alignement incorrect d'instruction sur plusieurs lignes), les erreurs de syntaxe, et toute erreur algorithmique non pénalisée dans les fonctionnalités précédentes.

```
1  // function build_parent_list(depNode, destNode)
2  // entrée depNode  pointeur Node de depart
3  // entrée destNode pointeur Node de destination
4  // sortie tabLink pointeurs Node destNode vers depNode
5
6  tabLink.push_back(destNode)       //initialement vide
7  goingBack ← destNode->parent
8  Tant que goingBack ≠ depNode
9    tabLink.push_back(goingBack)
10   goingBack ← goingBack->parent
11 tabLink.push_back(depNode)
12 sortir tabLink
13
14 // function build_forward_list(tabLink)
15 // entrée tabLink des pointeurs destNode vers depNode
16 // sortie tabConnect des pointeurs Node depNode vers destNode
17
18 Tant que tabLink.size()≠ 0
19   tabConnect.push_back(tabLink.back())
20   tabLink.pop_back()
21 sortir tabConnect
22
23 // function build_tabConnect(depNode, p)
24 // entrée : depNode ; entrée modifiée : pendulaire p
25 // sortie : destNode
26
27 destNode     ←tabProd[p.dest]
28 tabLink      ←build_parent_list(depNode, destNode)
29 p.tabConnect ←build_forward_link(tabLink)
30 p.done       ←true
31 Sortir destNode
32
33 // algo principal
34 // construit tabConnect pour chaque élément de tabPend
35 // entrée modifiée : tabPend
36
37 i←1
38 Tant que i ≤ nbPend
39   Si tabPend[i].done = false
40     depNode  ←tabLog[tabPend[i].depart]
41     Dijkstra(depNode, tabNode)
42     destNode ←build_tabConnect(depNode, tabPend[i])
43
44     j ←i+1  // recherche même config depNode/destNode
45     Tant que j ≤ nbPend
46       Si tabPend[j].done = false
47         Si depNode = tabLog[tabPend[j].depart]
48           Si destNode = tabProd[tabPend[j].dest]
49             tabPend[j].tabConnect ←tabPend[i].tabConnect
50             tabPend[j].done ←true
51           Sinon
52             build_tabConnect(depNode, tabPend[j])
53       j← j+1
54
55   i← i+1
56
57
```

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).(1pt)

Pour chaque Pendulaire l'exercice3 a construit un tableau des pointeurs de Node tabConnect de depNode à destNode. On pose que chaque Node de la Ville possède un tableau **tabVoisin** de Voisins ; un Voisin contient :

- un pointeur Node **next** vers un Node voisin
- un pointeur vers une structure Direction **dir** contenant :
    - un tableau **tab** de booleens à deux indices, de P lignes et de S colonnes. Chaque élément du tableau est initialisé à VIDE en début de simulation; l'autre état est PLEIN
    - un compteur de segments PLEIN **count** initialisé à 0 en début de simulation

Soit deux éléments consécutifs k et k+1 de tabConnect ; tabConnect[k] pointe sur un Node contenant tabVoisin dont un élément a pour champ **next** la valeur tabConnect[k+1] ; on utilise alors le champ **dir** de cet élément pour décider du déplacement du pendulaire.

La Ville tient à jour un tableau **tabDir** de l'ensemble des pointeurs de Directions pour l'affichage final.

4.2) Egalement 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.(1pt)

On caractérise l'état du pendulaire avec un champ **state** initialisé à WAIT et pouvant prendre les 2 autres valeurs STARTED, c'est-à-dire en chemin, et ARRIVED, c'est-à-dire arrivé à destination.

De plus, **crtConnect** est l'indice de l'élément courant de tabConnect initialisé à 1, **crtDir** est le pointeur vers la Direction décrivant l'état de la direction courante initialisé à nullptr, **crtPas** et **crtSeg** sont respectectivement les indices de ligne et de colonne du pendulaire dans la direction initialisés à 1.

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 ré-utilisation). (barème page suivante)

4.4) Estimer l'ordre de complexité de 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.(1pt)

La variable principale de la boucle est le nombre de pendulaires **nbPend**. Pour l'ensemble des *nb-passages* pendulaires qui se trouvent sur le dernier segment d'une direction il faut déterminer la *prochaine* direction si le pendulaire n'est pas encore arrivé à sa destination (coût linéaire en **nbNode**) et scanner les nb-passages' de cette prochaine direction pour trouver un premier segment libre.

Seul le sous-ensemble de pendulaires donné par la *somme des nb-passages* peut être amené à rechercher sa prochaine direction et examiner la disponibilité de *Max(nb-passages)* passages de leur prochaine direction potentielle. Dans le pire des cas, plutôt rarissime, où tous les pendulaires sont dans ce contexte, cela donne un coût calcul de l'ordre de **O(nbPend*(nbNode + Max(nb-passage))**

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. (1pt)

Il suffit d'initialiser un booleen **trafficJam** à **true** en ligne 11 de l'algo, puis de le faire passer à **false** dès qu'on fait bouger un pendulaire. Si après la boucle de mise à jour des pendulaires le booleen est encore à true c'est qu'effectivement AUCUN pendulaire n'a pu bouger pendant une mise jour et que cela ne peut que continuer ainsi car les pendulaires sont toujours mis à jour dans le même ordre.

Fonctionnalités devant être traitées dans le pseudocode de l'exercice 4 (7pts):

1. L'ensemble de tous les pendulaires est traité (0,5pt)

2. L'algo traite 3 états d'un pendulaire : AVANT le départ, ENTRE départ et destination, et SORTI à destination (1,5pt)

3. Chaque pendulaire dispose d'un moyen pour accéder à la direction suivante vers la destination (revoir la stratégie adoptée pour mémoriser/construire le chemin) (2,5pts)

4. vérification de passage libre effectuée avant d'entrer dans la première direction (0,5pt)

5. vérification de passage libre effectuée avant d'entrer dans la direction suivante (0,5pt)

6. un pendulaire progresse d'un seul segment si le suivant est libre (0,5pt)

7. gestion correcte de l'état PLEIN/VIDE de chaque segment (1pt)

Le dernier point concerne le style du pseudocode (numérotation de ligne décalée, indentation incorrecte, alignement incorrect d'instruction sur plusieurs lignes), les erreurs de syntaxe, et toute erreur algorithmique non pénalisée dans les fonctionnalités précédentes.

```
1   // function get_dir(i,tabConnect)
2   // entrée : i connexion à tester, tabConnect d'un pendulaire
3   // sortie : pointeur sur la direction correspondante
4
5   n ← tabConnect[i]
6   Pour chaque voisin de n->tabVoisin
7      Si voisin.next = tabConnect[i+1]
8         Sortir voisin.dir
9   Sortir nullptr                        // n'arrive jamais
10
11
12  // function free_first_seg(i,tabConnect)
13  // entrée : i connexion à tester, tabConnect d'un pendulaire
14  // sortie : 0 si échec, sinon valeur indice de passage libre
15
16  dir   ← get_dir(i,tabConnect)
17  ligne ← 1
18  Tant que ligne ≤ dir->tab.size()  //nb de passages de dir->tab
19     Si dir->tab[ligne][1]=PLEIN
20        ligne++
21     Sinon
22        dir->count++
23        dir->tab[ligne][1]=PLEIN
24        Sortir ligne
25  Sortir 0
26
27
28  // function remove_from_dir(dir,iPass,jSeg)
29  // entrée : pointeur Direction dir, indice passage iPass
30  // entrée : jSeg indice du dernier segment
31
32  dir->tab[iPass][jSeg]= VIDE
33  dir->count--
34
35
36  // function move_fw_in_dir(dir,iPass,jSeg)
37  // entrée : pointeur Direction dir, indice passage iPass
38  // entrée modifiée : jSeg si espace disponible
39  //    precondition : jSeg is not the last segment of dir->tab
40
41  Si tab[iPass][jSeg+1]= VIDE
42     dir->tab[iPass][jSeg]  = VIDE
43     dir->tab[iPass][jSeg+1]= PLEIN
44     jSeg++
45
```

```
1  // algo principal : affiche le nb de pendulaires par Direction
2  //                  à chaque mise à jour de la simulation
3
4  Pour chaque p dans tabPend
5    p.state ← WAIT
6    p.crtCon← 1
7    p.crtPas← 1
8    p.crtSeg← 1
9
10 Pour i de 1 à max
11
12   Pour chaque p dans tabPend
13     Si p.state ≠ ARRIVED
14
15       Si p.state = WAIT
16         iPas ←first_free_seg(1,p.tabConnect)
17         Si iPas ≠ 0
18           p.state  ← STARTED
19           p.crtPas ← iPas
20
21       Sinon // STARTED
22         dir ←get_dir(p.crtCon,p.tabConnect)
23
24         Si p.crtSeg = dir->tab[p.crtPas].size() //on last Seg
25
26            Si p.crtCon = p.tabConnect.size()-1  //last connec.
27              p.state ← ARRIVED
28              remove_from_dir(dir,p.crtPas,p.crtSeg)
29
30            Sinon
31              iPas← first_free_seg(p.crtCon+1,p.tabConnect)
32              Si iPas ≠ 0
33                Remove_from_dir(dir,p.crtPas,p.crtSeg)
34                p.crtCon++
35                p.crtPas ← iPass
36                p.crtSeg ← 1
37
38         Sinon
39           move_fw_in_dir(dir,p.crtPas,p.crtSeg)
40
41   // affichage count des Directions de la Ville (optionnel)
42
43   Afficher i
44   Pour chaque dir dans tabDir
45     Afficher dir.count
46   Afficher un passage à la ligne
47
```