TransitScan

Table des matières

1. Introduction

Dans un lointain futur[1] la téléportation est possible pour des robots mais la technologie n'étant pas totalement au point, il est nécessaire, d'une part, de fractionner les voyages en plusieurs étapes via des « hub » (Fig 1a) et, d'autre part, de scanner systématiquement les robots lors du transit dans chaque hub. On se propose ici d'analyser deux variantes d'un algorithme tirant parti du fait que le transit doit s'effectuer obligatoirement entre les deux terminaux du hub (Fig 1b). Chaque terminal d'arrivée est organisé en **nbF** files d'attentes par lesquelles arrivent les robots (sans limitation de taille de file d'attente). Le terminal de départ contient un couloir de sortie par file d'attente. On représente une *demande de transit* d'un robot par les indices[2] de la file d'attente d'arrivée et du couloir de sortie (ces indices peuvent être identiques). On se propose de traiter un ensemble prédéfini de demandes de transit en supposant que tous les robots sont déjà présents dans les files d'attente (Fig 1b). Sachant que chaque demande doit passer via l'unique scanner qui ne peut contenir qu'un seul robot à la fois, votre algorithme va devoir décider de la suite des files d'attentes et couloirs de sortie devant lesquels le scanner devra successivement se placer. La figure 1c illustre un cycle de l'algorithme par défaut ; l'algorithme s'exécute tant que les **nbF** files d'attente ne sont pas vides.

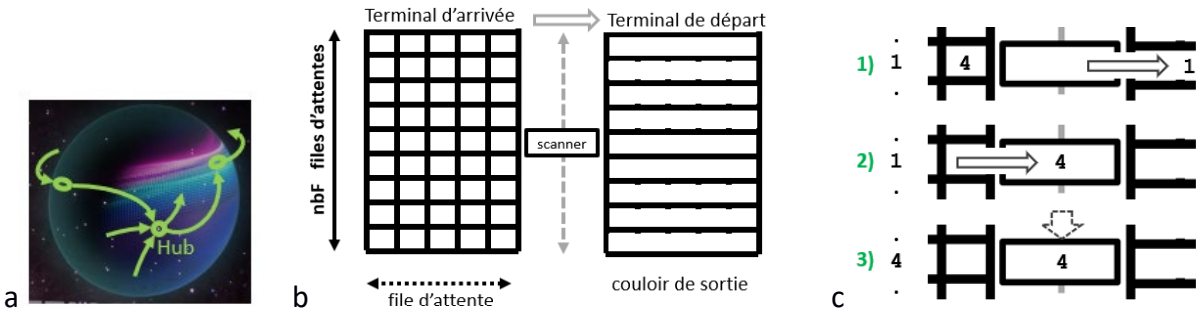

Figure 1 : (a) Hub de transit avec (b) à gauche un terminal d'arrivée organisé en **nbF** files d'attente (sans limitation de taille), au milieu le scanner par lequel doit passer chaque robot et à droite les nbC couloirs de sortie du terminal de départ. (c) **un cycle de mise à jour est une suite de 3 opérations successives** : 1) sortie du robot dans le couloir courant (ici 1), 2) entrée du robot en tête de la file d'attente courante vis à vis de ce couloir de sortie (algorithme par défaut) ; et 3) déplacement jusqu'au couloir de sortie d'indice exprimé par cette demande de transit (ici 4) pour y effectuer un nouveau cycle de mise à jour. Ces 3 opérations sont optionnelles ; le comportement détaillé est décrit en section 3.

Votre programme devra d'abord vérifier la validité des données du problème, puis afficher les informations suivantes : L'état initial des files d'attente, puis, si demandé avec une entrée dédiée, chaque cycle du scanner, et enfin des statistiques plus globales permettant de comparer les deux variantes de l'algorithme. Tout cela est détaillé en section 3.

[1] Ce contexte n'a aucune base scientifique ; il ne sert qu'à donner un cadre aux algorithmes demandés

[2] On utilise la convention du C++, c'est à dire entre **0** et **nbF-1**

2. Méthode de travail générale

2.1 Mise en oeuvre des grands principes

Le projet représente une quantité de travail bien supérieure aux exercices proposés dans les séries. La mise en oeuvre des principes ***d'abstraction*** et de ***ré-utilisation*** est un élément central dans la stratégie de résolution.

En effet, certaines tâches correspondent à un sous-problème (*abstraction*) dont la solution sous forme d'une fonction peut être utilisée pour résoudre une tâche plus complexe. C'est tout à fait normal qu'une telle fonction ne soit appelée qu'une seule fois car le but est la structuration de la solution d'une manière claire et lisible (un critère fréquent est que la taille maximum d'une fonction ne doit pas dépasser une page écran). Dans certains contextes la conception d'une fonction peut être ajustée à l'aide de paramètres pour pouvoir être *ré-utilisée* à plusieurs endroits dans le code.

2.2 Faire une analyse papier-crayon et pseudocode AVANT de vous lancer dans le codage

Le problème est décrit d'une manière indépendante d'un langage de programmation en section 3 ; c'est ce qu'on appelle les *spécifications*. Une telle description permet de réfléchir, *sans programmer*, à la décomposition du problème en un ensemble de sous-problèmes qui seront réalisés par des fonctions ; c'est ce qu'on appelle la phase *d'analyse*. Cette phase est typiquement faite avec un papier-crayon en pseudocode comme support pour organiser vos idées et faciliter le dialogue avec les assistant.e.s.

Le résultat de cette phase d'analyse est un ensemble de fonctions dont le *but* de chaque fonction est clairement identifié : sur quelles données travaille-t-elle ? Quel(s) résultat(s) fournit-elle ?

Ensuite seulement on peut passer à une méthode de codage rigoureuse qui va vous garantir une progression régulière dans la mise au point du projet. Elle est décrite ci-dessous.

2.3 Vérification précoce par les tests (*scaffolding*)

La clef du succès du codage est de ***vérifier*** que chaque fonction réalise bien son but avec un *solide éventail de tests pour lesquels on connait les résultats attendus*. Grâce à cette méthode un sous-problème est résolu une fois pour toute dès le niveau le plus élémentaire.

L'approche de vérification par des tests implique d'*écrire du code supplémentaire dédié à ces tests* AVANT de commencer à traiter les niveaux supérieurs du projet. Vous serez ainsi amené à écrire un certain nombre de petits programmes dédiés à ces tests. Cette méthode de travail est appelée *scaffolding* (échafaudage) et cherche à rendre votre code robuste à la grande variété des cas possibles.

En effet nous disposons d'outils tels que l'éditeur et le compilateur pour détecter certaines erreurs mais ils ne permettent pas de toutes les trouver. Nous savons déjà qu'il est recommandé de *recompiler très fréquemment* afin réduire au minimum le temps de recherche des ***erreurs syntaxiques*** (fautes d'orthographe/grammaire du langage C++). Le raisonnement est qu'une erreur se trouve dans la portion de code écrite depuis la dernière compilation avec succès, ce qui permet de réduire à très peu de lignes de codes l'espace de recherche de cette erreur. La méthode de l'échafaudage (scaffolding) est destinée à trouver les ***erreurs sémantiques*** que le compilateur ne trouve pas car le programme respecte la syntaxe du

langage ; les erreurs sémantiques vont produire un comportement incorrect du programme. Le point essentiel dans cette méthode est qu'il faut tester *chaque fonction* ***individuellement*** pour *vérifier* qu'elle produit le résultat attendu AVANT de l'utiliser ailleurs. Là aussi cette approche vous rend plus efficace dans le développement de code car l'erreur sémantique se trouve normalement dans la dernière fonction en cours de test car les autres fonctions qu'elle appelle doivent être déjà validées.

2.4 Bottom-up ou top-down ?

La méthode présentée dans la section précédente suggère de vérifier d'abord les fonctions de bas-niveaux avant celles qui les utilisent. C'est l'approche ***bottom-up*****.** C'est en général ce que nous recommandons pour un projet.

A l'inverse, il existe aussi une approche ***top-down*** pour la mise au point des fonctions ; il peut être légitime de vouloir tester une fonction **f()** qui appelle une fonction **g()** avant que **g()** soit écrite en détail. Cette approche *top-down* est possible si le résultat de **g()** est facile à définir, par exemple si elle renvoie `true` ou `false`, car la seule chose utilisée par **f()** est ce résultat.

On peut ainsi vérifier **f()** à l'aide d'une *forme minimale de la fonction* **g()** que l'on appelle un ***stub***. Cette forme minimale respecte le prototype prévu pour **g()** en terme de paramètres et de type de la valeur de retour ; par contre sa définition peut se restreindre à un corps vide si aucune valeur n'est retournée ou très peu de chose, par exemple une instruction return de *true* ou *false* si **g()** est supposée renvoyer un booléen.

2.5 Redirection des entrées-sorties pour automatiser les tests d'un programme

On utilisera **exclusivement** l'entrée standard (*clavier*) pour fournir les données et la sortie standard (*terminal*) pour afficher les résultats. Il ne faut pas écrire de code de lecture-écriture de fichier dans ce projet.

A la place, le mécanisme de *redirection* des entrées-sorties est vu en TP (semaine6) et permet de travailler avec des fichiers de test *sans avoir à changer une seule ligne de code*. En effet les données d'un test peuvent être éditées avec l'éditeur de programme et mémorisées dans un fichier de test. Ensuite il suffit de préciser le nom du fichier de test sur la ligne de commande qui lance l'exécution du programme et celui-ci obtient les données du fichier comme si elles venaient du clavier.

Exemple : si le nom de l'exécutable de votre projet est **proj** et que le nom d'un fichier de test est **t01.txt**, alors vous pouvez lancer votre exécutable dans le terminal de la façon suivante :

```
./proj < t01.txt
```

où le contenu de **t01.txt** est envoyé sur l'entrée standard pour cette exécution de **proj**.

Cette méthode de test est recommandée surtout pour relancer fréquemment et efficacement un ensemble de tests et vérifier que votre programme est toujours correct.

C'est aussi en redirigeant des fichiers de test sur l'entrée standard que nous noterons votre programme (précisions en section 4.4.2). Il faut donc strictement respecter cette consigne.

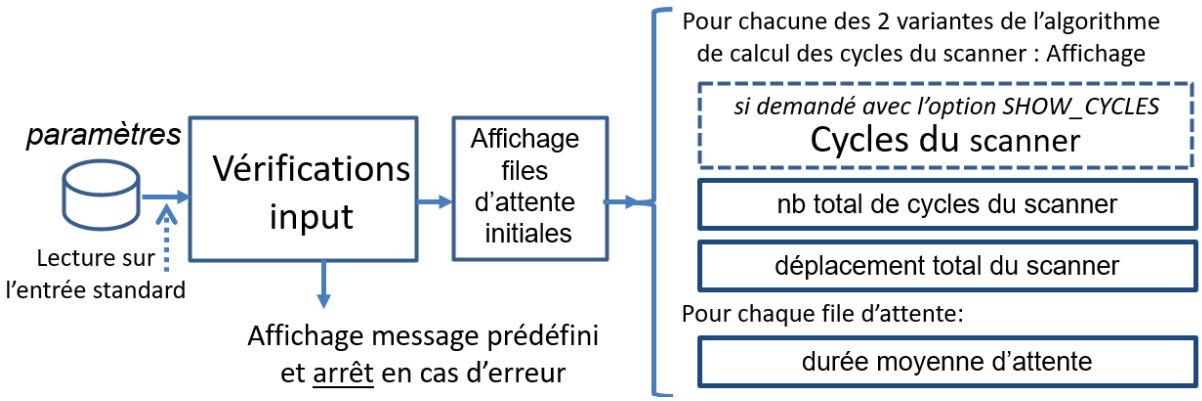

Figure 2: Tâches principales à effectuer par le programme : tout d'abord la vérification des paramètres avec arrêt en cas de détection d'erreur. Ensuite, on affiche l'état initial de chaque file d'attente. Ensuite, deux versions d'algorithmes sont exécutées à partir des mêmes données initiales. L'activation de l'option SHOW_CYCLE facilite la mise au point en affichant chaque cycle de mouvement du scanner. Les autres affichages sont les résultats globaux. Remarque : la lecture s'effectue sur l'entrée standard et les affichages sont effectués sur la sortie standard.

3. Spécifications détaillées

Cette section approfondit les éléments fournis en section 1 en cherchant à identifier des structures de données et des fonctions pouvant être ré-utilisées.

3.1 Buts des tâches

Le programme est une suite de trois tâches (Fig 2): d'abord la lecture et la vérification des données (ex : Fig 3 gauche), ensuite l'affichage de l'état initial des files d'attente (ex : 2 premières lignes de Fig 3 milieu) suivi par l'affichage des cycles (optionnel) et des statistiques finales permettant de comparer les deux variantes de l'algorithme.

3.1.1 Tâche 1 : lecture et vérification des paramètres

Plusieurs types d'erreurs doivent être détectés, dès la lecture de chaque paramètre. Dès qu'une erreur est détectée, le programme doit afficher un message d'erreur prédéfini puis s'arrêter. Nous désignons les messages d'erreur par une expression en majuscules dans la suite de cette section ; la manière de les afficher est fournie en section 4.4. En particulier, vous devez utiliser le fichier fourni **useful_stuff.txt** pour le recopier dans votre programme.

```
SHOW_CYCLES
2
0 1
0 0
-1 -1
```

```
Etat initial
0→1␣0↵
1↵
NEQLI↵
0→1→0␣1↵
1→0→1␣0↵
0→0→0␣1↵
0→0→1␣0↵
FANEQLI↵
0→1→0␣1↵
1→0→1␣0↵
0→0→0␣1↵
0→0→1␣0↵
Nombre de cycles↵
4→4↵
Deplacement total↵
2→2↵
Attente moyenne ↵
0→2.00→2.00↵
1↵
```

→ tabulation
␣ espace
↵ passage à la ligne

Figure 3: contenu du fichier de test t01.txt (gauche) ; affichage pour le fichier t01.txt lorsqu'il est redirigé sur l'entrée standard (milieu) ; symboles utilisés pour les tabulations, espace et passage à la ligne (ci-dessus). Les deux variantes de l'algorithme donnent le même affichage pour ce fichier de test.

Lecture des données : les données sont fournies sur l'entrée standard (stdin = clavier) sur des lignes distinctes selon l'ordre indiqué sur l'exemple de la Figure 3 (gauche). Les données d'une même ligne peuvent être précédées, suivies et séparées par un ou plusieurs espaces ou une ou plusieurs tabulations. Les lignes des données peuvent être séparées par zéro ou plusieurs lignes vides. Les paragraphes suivants décrivent les données fournies au programme :

Option d'affichage : le premier paramètre est une chaîne de caractères indiquant si on veut afficher chaque cycle individuel du scanner. Le choix est entre **`SHOW_CYCLES`** et **`SHOW_NO_CYCLE`**. Tout autre chaîne produit la détection d'erreur: BAD_DISPLAY_TYPE.

Nombre de files d'attente des terminaux d'arrivée et de départ **nbF** : doit être strictement positif. (détection d'erreur: BAD_QUEUE_NB).

Demandes de transit : chaque demande est représentée par un couple d'indices compris entre **0** et **nbF-1**. Le premier désigne la file d'attente d'arrivée et le second le couloir de sortie désiré. Plusieurs demandes peuvent être fournies simultanément (même ligne de fichier de test) ou séparément. La présence d'une valeur négative dans un couple d'indices signale la fin des demandes de transit (il est possible qu'il n'y ait aucune demande). Tout autre valeur d'indice produit l'erreur: BAD_QUEUE_INDEX.

Si une erreur est détectée, l'affichage du message d'erreur fait quitter le programme avec le code C++ fourni (section 4.4).

L'affichage doit être organisé en plusieurs sections commençant par un titre strictement prédéfini (Fig1 milieu) ; ces textes sont également fournis dans **usefull_stuff.txt**.

3.1.2 Tâche 2 : Affichage de l'état initial des files d'attente

Si aucune erreur n'est détectée le programme continue avec la tâche2 qui consiste à afficher l'état initial des files d'attente selon le format suivant : une ligne par file d'attente qui commence par l'indice de la file d'attente. Si la file est vide on passe directement à la ligne (file d'indice 1 dans Fig 3 milieu). Sinon on affiche une tabulation suivie par la suite des indices de départ demandés dans leur ordre d'apparition (ex : Fig 3 milieu, l'indice 1 est en premier pour la file 0). On affiche un espace pour séparer les indices successifs (Fig 3 milieu).

3.1.3 Tâche 3 : Algorithmes de mise à jour et statistiques finales

Résumé. un cycle de mise à jour du scanner doit effectuer ces opérations (Fig1c):

1) S'il contient un robot, celui-ci sort dans le couloir de sortie.
2) [alg. par défaut] : entrée du robot en tête de la file d'attente courante sauf si elle est vide.
   [variante] : idem et sauf si la file d'attente a déjà été traité dans le ***tour*** courant
3) [alg. par défaut] : déplacement vers le couloir indiqué par la nouvelle demande de transit ou, si la file d'attente est vide, vers la file non-vide de plus petit indice.
   [variante] : déplacement vers le couloir indiqué par la nouvelle demande de transit ou, si la file d'attente est vide, ou si la file a déjà été traitée au ***tour*** courant, vers la file non-vide et non traitée au ***tour*** courant de plus petit indice.

3.1.3.1 L'algorithme par défaut <=> *Non-Empty Queue with Lowest Index* (NEQLI) : cherche à déplacer le scanner en ayant le plus faible coût calcul pour déterminer le prochain couloir. Pour cela on initialise un booleen **vide** à **vrai** pour chaque file vide et à faux sinon.

On suppose aussi que le scanner se trouve initialement devant la file d'attente d'indice 0 ; il est vide initialement donc rien ne sort à la première mise à jour. Ensuite on extrait la demande de transit qui se trouve en tête de la file d'attente 0. Le scanner va se déplacer vers le couloir de sortie dont l'indice est fourni par cette demande de transit et cela termine le premier cycle de mise à jour. Si par contre la file d'attente 0 est vide, l'algorithme doit le déplacer devant la file d'attente non-vide de plus faible indice et cela termine aussi le premier cycle de mise à jour (ex : comparer le cas t01.txt de la Fig3 avec t02.txt fourni en supplément).

Dans le cas général, si un robot est libéré dans un couloir de sortie vis-à-vis d'une file d'attente vide, alors on choisit toujours de déplacer le scanner devant la file non-vide de plus petit indice (cet indice doit être mise à jour à chaque cycle de l'algorithme) ; c'est pourquoi cet algorithme est appelé NEQLI (*Non-Empty Queue with Lowest Index*). Enfin, le scanner s'immobilise devant la dernière file d'attente traitée.

Ex t03.txt : la file d'attente 0 contient 3 robots qui veulent aller vers le couloir de sortie 1, tandis que la file d'attente 1 contient 3 robots qui veulent aller vers le couloir de sortie 0. Il suffit de 7 cycles de mise à jour dans l'ordre 0 1 0 1 0 1 0 pour traiter toutes ces demandes.

3.1.3.2 Variante FANEQLI <=> « Fair Access » NEQLI :

L'algorithme par défaut peut produire de grosses différences entre les durées moyennes d'attente des files. Pour diminuer ce défaut, cette variante veut aussi garantir que les files d'attente non-vides sont équitablement accédées. Pour cela, on ajoute le concept de « tour » pendant lequel toutes les files non-vides doivent avoir été traitée une seule fois. Donc, en plus de continuer à tenir à jour le booléen **vide** de chaque file, chaque *nouveau* tour va initialiser *un booléen supplémentaire* **done** avec la valeur courante de **vide** et va le mettre à jour pendant le tour courant.

Comme dans l'algorithme par défaut, un tour commence par la file non-vide de plus petit indice. Ensuite, si le booléen **done** est **faux** pour la file vis-à-vis de la destination du robot de la file courante alors cette file de destination sera traitée au tour suivant. Si par contre son booléen **done** est **vrai**, on va seulement s'y déplacer pour déposer le robot mais sans rien extraire de cette file déjà traitée au tour courant. Il faudra alors déplacer le scanner (vide) devant la file non-vide de plus petit indice qui n'a pas encore été traitée dans le tour courant (c'est-à-dire avec **done** à **faux**). Si nécessaire il faut mettre à jour les booléens **done** et **vide**.

Ex t04.txt : la file d'attente 0 contient 3 robots qui veulent aller vers le couloir de sortie 0, tandis que la file d'attente 1 contient un robot qui veut aller vers le couloir de sortie 1.
Il suffit de 6 cycles de mise à jour dans l'ordre 0 0 0 0 1 1 pour traiter toutes ces demandes avec NEQLI mais la durée moyenne d'attente de 5 est plus longue pour la file d'indice 1 comparée à 2 (= (1+2+3)/3) pour la file d'indice 0. Avec FANEQLI, on équilibre les durées d'attente ; il faut alors 7 cycles dans l'ordre 0 0 1 1 0 0 0 produisant une durée d'attente de 4 (= (1+5+6)/3) pour la file 0 et de 3 pour la file 1.

Ex t05.txt : ce scénario plus complet montre l'intérêt de tirer parti de la file d'attente vis-à-vis de la destination du robot pour effectuer le tour courant plus efficacement que de prendre les files dans l'ordre croissant des indices de **done**. L'algorithme NEQLI traite les files dans l'ordre 0 0 0 0 0 1 4 3 2 2 4 3 3 produisant les durées moyennes 2.5, 6, 9.5, 10 et 9. La variante FANEQLI introduit le concept de tour, marqué ici par une virgule : 0 0 1 4 3 2 , 2 4 3 3 0, 0, 0, 0 produisant les durées moyennes 9.25, 3, 6.5, 7 et 6.

3.1.3.3 Format d'affichage des cycles (Fig 3 milieu) : Si l'affichage des cycles est demandé avec l'option SHOW_CYCLES, on affiche d'abord les cycles de NEQLI puis les cycles FANEQLI. L'option SHOW_NO_CYCLE ne produit aucun affichage (cf exemple fichier t06.txt et out06.txt).

Pour faciliter la mise au point, on demande d'afficher un seul cycle de mise à jour de l'algorithme par ligne. Chaque cycle affiche les éléments suivants : indice de la file d'attente courante, une tabulation, indice de la prochaine file d'attente, une tabulation, booléen de sortie, un espace, booléen d'entrée. Les deux booléens prennent les valeurs 0 ou 1, respectivement pour faux et vrai. Le booléen de sortie est vrai si un robot est sorti dans le couloir de sortie de même indice que la file d'attente courante. Le booléen d'entrée est vrai si un robot est entré dans le scanner depuis la file d'attente courante (Fig 1c).

3.1.3.4 Statistiques globales sur des lignes distinctes (Fig 3 milieu):

- Nombre total de cycles : d'abord pour NEQLI, tabulation, puis pour FANEQLI.
- Déplacement total : d'abord pour NEQLI, tabulation, puis pour FANEQLI. Le **déplacement total** est la somme des différences des indices des files d'attentes courantes.
- Durée moyenne d'attente pour chaque file d'attente : une ligne par file d'attente. Si la file est vide on passe directement à la ligne après l'indice de la file. Sinon la **durée moyenne** est obtenue en additionnant d'abord tous les indices de cycles (entre 1 et le nombre total de cycles) pour lesquels un robot a été extrait de la file d'attente, c'est-à-dire quand le booléen d'entrée est vrai (Ex : t04.txt). On divise ce total par la taille initiale de la file:

indice de la file, tabulation, durée moyenne NEQLI, tabulation, durée moyenne FANEQLI

4. Implémentation en langage C++

4.1 Contraintes spécifiques à ce projet

Indépendamment de ce que doit faire le programme (les spécifications décrites plus haut) nous imposons les contraintes suivantes de mise en oeuvre :

- Ecriture en C++ : votre code sera compilé avec l'option **`-std=c++17`**
- Tout votre code doit être écrit dans un seul fichier source.
- Il faut utiliser le type **queue** pour une file d'attente et **vector** pour mémoriser les ensembles de file d'attente (inclure les deux fichiers header correspondant).
- Le calcul des durées moyennes doit être fait en virgule flottante **double** précision. Il faut inclure le header **iomanip** et écrire cette instruction avant d'afficher les durées :

```
std::cout << setprecision(2) << fixed ;
```

4.2 Clarté et structuration du code avec des fonctions

La clarté de votre code sera prise en compte. Un examen manuel de votre code source sera effectué par une personne chargée d'évaluer le respect des conventions de programmation utilisées dans ce cours. Par souci d'efficacité seuls les codes indiqués dans nos conventions vous seront communiqués avec les numéros des lignes concernées dans votre fichier source.

4.3 Variables locales ou globales ?

4.3.1 Où déclarer les variables et les tableaux ?

La règle de base est qu'une variable (ou un tableau) n'est déclarée que ***localement***, là où elle est utilisée, le plus bas possible dans la hiérarchie des appels de fonctions. **Si et seulement si** une variable **x** (ou un tableau) déclarée dans une fonction **h()** est ***nécessaire*** pour une autre

fonction **f(),** alors elle est transmise en paramètre à cette fonction[3] au moment de l'appel **f(x)** avec transmission par valeur ou par référence selon vos besoins**.**

Conséquence: si deux fonctions **f()** et **g()** indépendantes l'une de l'autre doivent travailler sur la même variable **x** (ou un tableau) alors une fonction de niveau supérieur **h()** doit être écrite qui va appeler **f(x)** et **g(x)** avec transmission par valeur ou par référence selon vos besoins.

4.3.2 Qu'en est-il des ***constantes*** ?

Voici les règles que nous nous donnons, en conformité avec nos conventions de programmation :

- Si une constante n'est utilisée qu'à l'intérieur d'*une seule fonction*, alors la déclaration d'une variable avec **constexpr** peut être faite dans cette fonction ou globalement.
- Si la constante est utilisée dans *plus d'une fonction*, alors la déclaration d'une variable avec **constexpr** *doit* être faite de manière globale, en début de fichier comme décrit dans les conventions de programmation.
- L'alternative de la déclaration de symboles avec #define est autorisée pour des constantes utilisées dans plusieurs fonctions. Elles sont aussi déclarées en début de fichier comme décrit dans les conventions de programmation.

4.4 Fichiers de test, messages d'erreur :

Nous vous fournissons des fichiers de tests dans le folder du projet sur moodle (section 4.4.2).

4.4.1 Message d'erreur

Le fichier **useful_stuff.txt** contient les définitions C++ des constantes, dont celles des messages d'erreur et une fonction **print_error** qu'il faut recopier dans votre fichier source (cf conventions). La fonction **print_error** se charge de quitter le programme en appelant **exit(0)**.

Exemple d'utilisation : si votre code détecte l'erreur BAD_DISPLAY_TYPE il suffit de faire un appel en passant seulement la constante du message: **print_error(BAD_DISPLAY_TYPE).**

4.4.2 Vérification anticipée de l'exécution de votre projet

Pour chaque fichier de test public, noté **`txx.txt`**, nous fournissons le fichier texte de la sortie attendue, noté **`outxx.txt`**. Vous pouvez ainsi valider les tests publics en lui comparant votre propre sortie, notée par exemple **`affxx.txt.`** Tout d'abord, voici la commande qui mémorise votre affichage pour le fichier de test **`txx.txt`** :

```
./proj < txx.txt > affxx.txt
```

Ensuite, toujours dans le terminal, lancez la commande **diff** qui affiche les différences entre les deux fichiers qui suivent la commande

```
diff -s  outxx.txt affxx.txt
```

Le test est validé si elle ne trouve aucune différence, en affichant le message suivant :

```
Files outxx.txt and affxx.txt are identical
```

Nous automatiserons la vérification de l'exécution de votre projet de la même manière, en comparant toutes les sorties attendues avec celle obtenues avec votre projet (après

[3] Nous n'acceptons pas l'approche qui consiste à transmettre systématiquement un grand nombre de paramètres à la fonction **f()** et qui restent ensuite inutilisés dans cette fonction car cela la rend moins lisible. Conclusion : *chaque paramètre doit avoir sa justification*.

compilation par nos soins sur la VM). Nous utiliserons aussi des fichiers de tests privés, non-fournis, pour vous stimuler à créer d'autres scénarios de tests représentatifs des différents scénarios d'exécution du programme. Il suffit d'ouvrir votre éditeur de code source préféré pour écrire les données et de sauvegarder avec l'extension **.txt** .

5 Conseils et proposition d'ACTIONS pour mettre en œuvre votre projet

Les sous-sections « ACTIONS » sont seulement des suggestions pour organiser votre travail. Il est évident que chacune des ACTIONs proposées doit être testée avec les fichiers fournis et ceux que vous aurez créés vous-même, avant de mettre au point l'ACTION suivante. Pour être efficace, utilisez la redirection des entrées-sorties (série TP_s4.2). On suppose que le type des entrées est correct (string, type numérique).

5.1 Tâche1 : ACTION1 : test de la lecture des paramètres

La section 3.1.1 et la Fig 3 gauche décrivent l'ordre de lecture des paramètres. Concevez une première version du programme qui se limite à la lecture et la détection d'erreur sur les paramètres fournis en entrée.

Dans ce premier exercice d'abstraction, rappelez-vous que la fonction principale **main()** joue seulement le rôle de « table des matières » et est constituée essentiellement d'appels de fonctions. Vous devez donc déléguer cette tâche de lecture à une ou plusieurs fonctions. Pour que cela fonctionne correctement **main()** doit déclarer les structures de données qui vont recevoir les données lues. Ces structures de données doivent être passées aux fonctions à qui on délègue la tâche de lecture ; on doit effectuer un passage de paramètre qui permet la modification des structures de données par la ou les fonctions de lecture/vérification.

Quel choix de structuration des données ?

- La valeur de **nbF** n'étant pas connue au moment de l'écriture du code, **vector** s'impose comme choix pour mémoriser *l'ensemble* des files d'attentes.
- Comme indiqué en section 4.1 une file d'attente individuelle est idéalement représentée par le type **queue** qui modélise un tel comportement (First-In, First-Out).
    - donc l'ensemble des files d'attente est un **vector** de **nbF** éléments **queue**.
- Par contre, pour les statistiques globales, on sait a priori qu'on ne compare que deux variantes d'algorithmes, on doit donc travailler avec **array**.
    - Réfléchissez à ce qui convient le mieux pour l'ensemble des durées moyennes par file d'attente.

5.2 Tâche2 : ACTION2 : Affichage des files d'attente

Dans **main()**, à la suite de la lecture des données, appelez une fonction d'affichage à qui vous passez la structure de donnée de l'ensemble initial des files d'attente (il ne doit pas être modifié par cet appel).

5.3 Tâche3 : ACTION3 : trouver et mettre à jour le plus petit indice de file non-vide

Avant de vous lancer dans le code, ébauchez le pseudocode de cet algorithme qui prend en entrée l'ensemble des files d'attente. Considérez l'initialisation d'un **vector** de **nbF** booléen, appelé **vide** dont un élément **vide[i]** est **vrai** si la file d'attente d'indice **i** est vide. Ensuite, connaissant la valeur courante de l'indice **iNonVide** de la première file non-vide, déterminez l'algorithme de mise à jour du **vector vide** et de l'indice **iNonVide** au moindre coût après chaque traitement d'une demande de transit. Testez plusieurs scénario dont le cas où toutes les files d'attentes sont vides.

5.4 Tâche3 : ACTION4 : Algorithme NEQLI / affichage des cycles

L'action précédente résout l'initialisation de deux informations clef pour l'algorithme NEQLI. Il reste à traiter l'ensemble des demandes de transit sachant que la position initiale du scanner **iScan** est la file d'attente d'indice **0** et que le scanner est initialement vide. Sur ce dernier point, il peut arriver plus d'une fois que le scanner soit positionné devant une file d'attente vide et qu'il doive faire son prochain déplacement à vide. Prévoyez un booléen **empty** initialisé à **vrai** qui servira à l'affichage et qu'il faudra mettre à jour si nécessaire.

Un compteur **nbCycle** de cycles de mise à jour doit être prévu ; il commence à 1. Si le scanner se trouve initialement devant une file vide (quand **iNonVide** > 0) alors ce premier cycle n'a aucune sortie ni entrée dans le scanner et **iScan** va prendre la valeur de **iNonVide** pour le cycle suivant.

Ensuite l'algorithme NEQLI doit fonctionner tant qu'il reste une file d'attente non-vide. Chaque cycle de mise à jour peut faire changer les variables **iScan**, **iNonVide**, **empty**, **vide** et **nbCycle**. Dans cette première phase de mise au point concentrez-vous seulement sur l'affichage de chaque cycle.

5.5 Tâche3 : ACTION5 : Algorithme NEQLI / statistiques globales

Reprenez votre algorithme pour l'évaluation des statistiques globales. Examinez d'abord le scenario d'une seule file non-vide ; comment calculez-vous la somme des durées d'accès aux éléments de la file ? Ensuite, comment est-ce adapté avec deux files non-vides et comment le déplacement total du scanner est-il mis à jour. Cela se généralise-t-il à d'autres configurations initiales des files d'attente ?

5.6 Tâche3 : ACTION6 : Algorithme FANEQLI

Reprenez la réflexion dans le cadre de l'ACTION4. Il faut ajouter le **vector done** de **nbF** booléen responsable de la gestion du concept de « tour ». Initialisez-le avec l'état actuel du vector **vide** au début de la boucle conditionnelle principale de NEQLI. L'indice de parcours **iDone** est initialisé avec **iNonVide** afin qu'une nouvelle boucle conditionnelle interne commence à la première file non-vide qui n'a pas encore été traitée dans le tour courant. Chaque passage dans cette boucle conditionnelle interne doit mettre à jour **iDone** et **done** en plus des variables de NEQLI. Le fichier de test t05.txt est discuté en page 6.

6. Rendu :

Noms des fichiers : tous les fichiers ont le même nom qui doit être votre numéro de SCIPER ; ils diffèrent seulement par leur extension :

.cc pour le fichier source, **.pdf** pour le rapport, **.zip** pour le fichier archive.

Par exemple pour une personne X de numéro SCIPER **123456**, le nom de fichier source est **123456.cc** son rapport est **123456.pdf** et ces deux fichiers sont dans le fichier archive **123456.zip** .

Téléversement du fichier archive : le fichier archive contient seulement l'unique **fichier source** et le **rapport**. Il DOIT obligatoirement être de type **.zip** ; la VM met à disposition cet outil de compression. Il faudra le téléverser (upload), au plus tard le **22/12 à 17h,** à l'aide du lien sur **moodle** (Topic 14) ; le site de téléversement sera ouvert plusieurs semaines avant la date limite sur moodle. Vous êtes responsables de *vérifier* que l'upload s'est bien passé en le téléchargeant (download) dans votre compte et en examinant que tout est bien présent. Vous pouvez toujours faire un nouvel upload jusqu'à la date limite.

Votre code source doit respecter les conventions de programmation du cours dont : au maximum 87 caractères par ligne (commentaire compris) et 40 lignes par fonction.

6.1 Rapport

Le Rapport est d'au maximum **2 pages** écrites avec un traitement de texte. Il ne contient PAS de page de titre, ni de table des matières. Le Rapport contient :

a) une déclaration concernant l'usage d'outils :
a1) edstem : indiquer en max 2 lignes comment vous vous êtes servi de cet outil par rapport à la discussion avec d'autres personnes et avec des assistants. Avez-vous consulté / posé des questions / contribué à des commentaires/des réponses ?
a2) IA générative : cet usage n'est pas interdit mais doit être documenté sur max 2 lignes. Ecrire soit « *Aucun usage d'IA générative* » ou préciser l'usage comme suit: 1) nom du/des outil(s)/éditeur(s). Ex : *ChatGPT4.0/OpenAI*, *Copilot/Microsoft*. 2) liste des tâches demandées : par exemple, *explication de concept, explication de message d'erreur, déverminage, production de code, reformulation de texte*…

b) Résultat de la phase **d'analyse** (max une demi-page, police de taille 11) :
Décrire la structure générale du programme en faisant ressortir, avec concision (Fig 5), la mise en oeuvre des principes *d'abstraction* et de *ré-utilisation* dans votre projet.

c) Fournir votre **Pseudocode** de NEQLI : Le pseudocode doit tenir en entier sur une seule page (soit sur la p 1 ou la p 2 mais pas entre les deux). Utilisez des fonctions. Pour ce projet, on autorise de s'aligner sur l'usage du C++ pour l'usage des indices de liste pour être en phase avec la donnée.

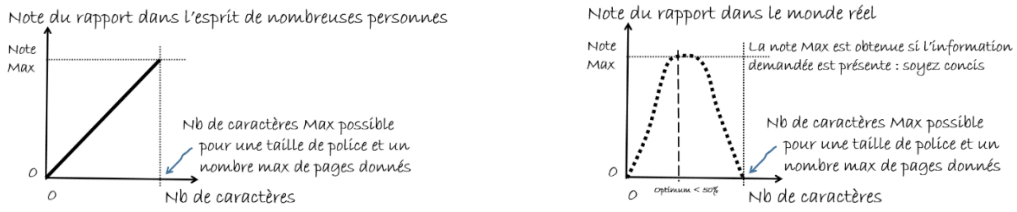

Figure 5 : un mauvais rapport dilue l'information utile jusqu'à atteindre le nombre maximum de caractères possibles sur la page (à cause de la croyance populaire de la figure de gauche) : dans la réalité ce type de rapport très compact sera très pénalisé parce qu'il est peu lisible (figure de droite) ; un bon rapport est celui qui fournit les informations demandées avec concision avec une mise en page aérée et lisible

6.2 Barème indicatif (12pts):

Ce barème est indicatif et provisoire. Il sera éventuellement modifié par la suite.

(2pt) rapport : soyez concis (Fig 5)
(4pt) Lisibilité, structuration du code et conventions de programmation du cours
(3pt) votre programme fonctionne correctement avec les fichiers publics
(3pt) votre programme fonctionne correctement avec les fichiers non-publics

Dernier rappel : le projet d'automne est INDIVIDUEL ; vous pouvez seulement utiliser des outils de type copilot de VSCode. Il est interdit de sous-traiter tout ou partie du travail à un tiers. De plus le détecteur de plagiat sera utilisé selon les recommandations du SAC. Le plagiat inclut la copie de code disponible sur internet.