Compte-rendu du TP C++ 2
Spécification et conception

Marc Gagné Selma Nemmaoui

Novembre 2015

Préambule

L'objectif de ce programme est de gérer l'ajout et l'étude de grandes quantités de données provenant de capteurs répartis dans une ville comme Lyon. Grâce à l'entrée standard, l'utilisateur peut envoyer une série de commandes au programme, qui affiche les réponses éventuelles sur la sortie standard. Afin de pouvoir gérer efficacement au plus 20 000 000 d'événements et 1 500 capteurs, nous avons essayé d'être particulièrement attentifs à la performance de notre programme dès le début de notre conception, aussi bien en termes de vitesse qu'en termes de mémoire utilisée (note : malgré nos tentatives d'optimisation, le plus grand facteur limitant en termes de performance demeure la lecture des commandes, qui ralentit considérablement l'exécution du programme).

Ce document de conception et de spécification a pour but de présenter notre programme et d'expliquer les choix que nous avons réalisés.

1 Structure du programme

Pour modéliser le problème, nous l'avons divisé en trois unités représentatives :

— les événements, représentant l'état de la circulation dans une partie de la ville à un instant donné,

— les capteurs, générateurs d'événements, chacun repéré par un identifiant numérique,

— et la ville, un ensemble de capteurs représentant le plus haut niveau de notre programme.

Note : Les jours de la semaine sont numérotés de 1 à 7 (ou de 0 à 6 à l'intérieur du programme). Comme la correspondance entre ces nombres et le nom du jour n'était pas spécifiée, nous avons supposé que 1 correspond à lundi et 7 à dimanche dans ce document ; en réalité, cela n'a pas d'importance puisque le programme n'utilise pas les noms de journée, mais il est plus simple de parler de "lundi" et "mardi" que de "jour 1" et "jour 2", donc nous avons adopté cette convention dans notre documentation.

1.1 Les événements

Un événement est caractérisé par l'état du trafic au moment de sa génération (V, J, R ou N, représentés dans notre programme par une valeur entre 0 et 3) et la date et l'heure de sa génération (représentées dans notre programme par un *timestamp* en minutes, les secondes n'étant pas relevées). Comme le cahier des charges indique que les dates des événements seront comprises entre mai 2015 et septembre 2015, ceci nous permet de démarrer notre timestamp le 1er mars 2015 à 0h00 ($t = 0$) et de le terminer le 30 septembre 2015 à 23h59 ($t = (31 + 30 + 31 + 31 + 30) \times 24 \times 60 - 1 = 220\,319$), ce qui donne un total de 220 320 valeurs possibles du timestamp ; notre représentation d'un élément doit donc être capable de stocker ces deux valeurs sans pertes.

1.2 Les capteurs

Un capteur est caractérisé par son identifiant (un réel à valeur quelconque) et par les événements qui lui sont liés (jusqu'à 20 000 000 potentiellement, si tous les événements ont été générés par le même

capteur). Les événements eux-mêmes ne sont pas stockés dans des capteurs ; le capteur n'a qu'une liste de *liens* vers chaque événement, qui eux sont stockés ailleurs (voir section suivante pour plus de détails). Un lien vers un événement doit donc contenir tous les détails nécessaires pour pouvoir y pointer sans ambiguïtés.

L'avantage de stocker les données de cette manière est qu'elle nous offre plusieurs moyens d'accéder aux événements, selon le critère de recherche que l'on utilise. Si l'on doit accéder aux événements qui appartiennent au capteur 42, par exemple, il suffit de récupérer les liens du capteur 42 pour avoir la liste de tous les événements qui ont été générés par le capteur 42.

Note 1 : Si nous avions besoin de plus de critères de sélection que le jour de la semaine et le capteur, il suffirait de créer des nouvelles listes de liens générées selon le critère désiré.

Note 2 : Nous aurions également pû utiliser un pointeur à la place d'un lien, mais un pointeur contient moins d'informations que le lien, puisque le lien stocke le jour de la semaine de l'événement. De plus, sur un système 64-bit, le pointeur pourrait devenir deux fois plus grand en taille que le lien, dont la taille est indépendante de l'architecture de la machine.

1.3 La ville

La ville est le plus haut niveau de gestion de notre programme. Elle est composée d'une série de capteurs responsables pour la génération des événements ainsi que des événements eux-mêmes. Par défaut, il n'y a aucun capteur ; lorsqu'un événement est ajouté avec un ID correspondant à un capteur qui n'a pas encore été créé, un nouveau capteur avec cet ID est ajouté à la liste des capteurs. Les capteurs sont donc stockés dans l'ordre où ils sont ajoutés. Quant aux événements, ils sont triés selon le jour de la semaine où ils ont été générés, puis stockés dans la liste correspondant à cette journée ; le lien vers cet événement est également créé et ajouté au capteur associé.

Comme l'ID d'un capteur peut dépasser 1 499, on ne peut pas faire correspondre l'ID à la position du capteur dans un tableau. La ville doit également permettre d'accéder rapidement à un capteur selon son ID sans avoir à parcourir toute la liste des capteurs, ce qui nous amène à attacher une table de hachage à la ville, table qui crée une association entre l'ID et la position du capteur.

2 Classes

Pour implémenter le modèle du problème décrit dans la section précédente, nous avons réalisé six classes, dont trois qui représentent les entités présentées auparavant (à savoir un événement, un capteur et une ville). Même si nous avons essayé de réduire l'espace mémoire occupé par le programme lorsque possible, nous avons choisi de nous concentrer sur la performance du programme avant tout (en essayant de conserver un espace mémoire maximal raisonnable, soit autour de 200 Mo) ; les structures de données que nous avons utilisé reflètent ce choix, puisque nous utilisons beaucoup de tableaux pour autoriser des recherches et des ajouts rapides.

2.1 `City`

La classe `City` est responsable de l'ensemble du programme : elle stocke les événements et les capteurs, gère leur ajout et fournit les méthodes pour calculer les statistiques sur ces données.

Les méthodes associées aux commandes sont : `addEvent` pour `ADD`, `displaySensorStateStats` pour `STATS_C`, `displayDayTrafficJamStats` pour `JAM_DH`, `displayDayStateStats` pour `STATS_D7` et `displayOptimalDepartureTime` pour `OPT`. Chacune affiche ses résultats sur la sortie standard ; ces méthodes sont présentées dans la suite du document, donc nous ne traiterons que des mécanismes pour stocker les données dans cette section. À noter que, contrairement aux commandes tapées par l'utilisateur, les jours de la semaine passés en paramètre doivent avoir leurs valeurs comprises entre 0 et 6, et non pas entre 1 et 7.

2.1.1 `events`

Afin de pouvoir manipuler des quantités d'événements très grandes (jusqu'à 20 000 000 événements à la fois), `City` utilise sept `SegmentedTable` (voir ci-dessous) stockées dans la propriété `events`, chaque table correspondant à un jour de la semaine (exemple : tous les événements ayant eu lieu un lundi se trouvent dans la table `events[0]`). Chaque table segmentée est initialisée avec une taille de segment de $\left\lfloor \frac{20\ 000\ 000}{7} \right\rfloor + 1$ et un nombre maximal de segments de 7 : nous supposons qu'en moyenne, sur de très grandes quantités de données, les événements seront répartis à peu près équitablement sur les sept jours de la semaine ; dans une situation parfaitement équilibrée, un segment par journée serait alloué.

Cette table est directement utilisée par toutes les commandes qui ont besoin de trouver un événement selon le jour de la semaine (à savoir les commandes `JAM_DH <D7>` et `STATS_D7 <D7>`), et permet de parcourir en temps $\mathcal{O}(n)$ tous les événements qui ont eu lieu sur ce jour de la semaine.

2.1.2 `sensors`, `numSensors` et `sensorIndices`

La table `events` permet d'accéder rapidement à tous les événements d'une journée, mais n'est pas efficace lorsque l'on veut trier selon le capteur qui a généré l'événement. Les pointeurs vers les capteurs sont donc stockés dans la table `sensors`, allouée sur la pile avec une taille de 1 500 (nombre maximal de capteurs possible) ; la propriété `numSensors` contient le nombre de capteurs créés jusqu'ici (et est incrémentée à chaque ajout d'un capteur).

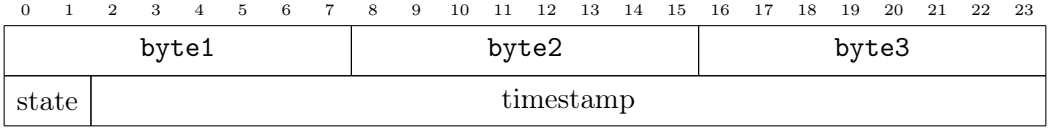
Le problème de ce mécanisme est qu'il n'est pas efficace lorsque l'on cherche un capteur selon son ID, et non pas selon sa position dans la table (ce qui est souvent le cas) ; c'est pour cela que la classe possède également la table de hachage `sensorIndices`, qui crée une correspondance entre l'ID d'un capteur (la clé) et son index dans `sensors` (la valeur). Comme la fonction de hachage n'est qu'une opération modulo sur l'ID, nous initialisons la table avec un modulo de $1\ 500 \times 10 = 15\ 000$ (nous supposons que les ID seront rarement supérieurs à 15 000, qui semble donc être une valeur raisonnable de modulo) et avec une taille de contenant de 1 500 (puisque, dans le pire des cas, il y a une collision sur chaque ID, donc la table de hachage devient un simple tableau).

S'il y a effectivement une collision sur chaque ID, alors notre table de hachage fonctionne en temps $\mathcal{O}(n)$ et ne sert à rien ; ceci semble improbable cependant, et dans nos tests elle fonctionne habituellement en temps $\mathcal{O}(1)$, puisqu'il y a peu de collisions. Pour réduire encore la chance des collisions, nous pourrions encore augmenter le modulo, ce qui ferait prendre encore plus de place au programme.

2.2 `Event`

La classe `Event` permet de stocker l'état de trafic associé à un événement ainsi que son timestamp. Comme il est possible d'avoir 20 000 000 d'événements, il est crucial que cette classe occupe le moins d'espace possible (sans qu'il n'y ait pertes d'informations).

Pour cela, nous sommes descendus au niveau des bits sur lesquels nous allons coder ces deux valeurs. L'état a 4 valeurs possible (V, J, R et N, qui correspondent respectivement à 0, 1, 2 et 3 dans notre programme), donc il peut être codé sur 2 bits. Le timestamp, comme nous avons vu auparavant, peut prendre 220 320 valeurs différentes ; il faut donc au moins 18 bits pour coder cette valeur (car $2^{18} = 262\ 144$). Ainsi, il nous faut 20 bits pour tout coder, ce qui est trop pour 2 bytes (16 bits) mais peut rentrer sur 3 bytes (24 bits). La classe a donc trois propriétés (`byte1`, `byte2` et `byte3`) sur lesquels nous répartissons les bits :

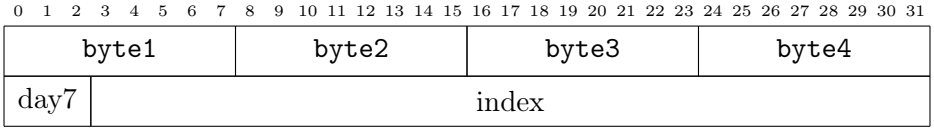


Lorsqu'un `Event` est initialisé, les valeurs passées en paramètre sont converties pour pouvoir être stockées sur ces trois bytes ; lorsque les méthodes `state()` ou `date()`, les valeurs sont extraites des bytes et renvoyées à l'utilisateur. La méthode `date()` en particulier renvoie toujours le timestamp de l'événement ; si elle est en plus appelée avec les variables `month`, `day`, `hour` et `minute` passées par

référence, elle attribuera à ces variables les valeurs correspondantes au timestamp (qui sera également renvoyé), ce qui permet de facilement récupérer la date sous deux formats différents.

2.3 `EventLink`

La classe `EventLink` fournit un lien vers un `Event` stocké dans la table `events` de `City`. Pour identifier uniquement un événement, il suffit de stocker le jour de la semaine où il a eu lieu et son index dans la table segmentée correspondant à ce jour. Tout comme pour la classe `Event`, nous désirons minimiser la taille de cette classe puisqu'elle peut être instanciée 20 000 000 de fois, donc elle travaille également sur les bits. Le jour de la semaine peut prendre 7 valeurs différentes, donc peut être codé sur 3 bits, tandis que l'index peut atteindre 20 000 000, donc il faut au moins 25 bits pour le coder (car $2^{25} = 33\ 554\ 432$). `EventLink` nécessite donc 28 bits, codés sur 4 bytes ainsi :



`EventLink` est initialisé comme `Event` : les valeurs passées en paramètre sont codées sur les 4 bytes, et lorsque le programme a besoin de les récupérer, il fait appel aux fonctions `day7()` et `index()`.

2.4 `HashTable`

La classe `HashTable` fournit une table de hachage pour des `short`s. Elle est censée être utilisée par la classe `City` pour accéder rapidement à un capteur par son ID. Pour ajouter une association clé/valeur, le programme appelle la méthode `add(key, value)`. La méthode `hash(value)` applique alors la fonction de hachage (qui est ici une simple opération modulo utilisant la propriété `modulo`) à la clé, renvoyant l'index d'un seau, et la valeur est ensuite ajoutée à ce seau. Pour récupérer cette valeur, il faut d'abord appeler la méthode `entries(key)` pour récupérer le nombre de valeurs dans le seau, puis itérer sur ces valeurs avec la méthode `get(key, index)` jusqu'à ce que la bonne valeur est retrouvée (un mécanisme externe est nécessaire pour savoir quelle valeur est la bonne).

La taille de chaque seau est stocké dans `bucketSize`, les seaux dans `buckets` et le nombre d'éléments dans chaque seau dans `bucketsNumElements`. Le nombre de seaux est égal à `modulo`, afin que chaque valeur possible ait un seau correspondant.

2.5 `SegmentedTable`

La classe `SegmentedTable` est un tableau qui augmente sa taille par morceaux de taille constante (spécifiée à l'initialisation) au besoin, ce qui permet à cette classe de stocker de très grandes quantités de données ; le nombre maximal de morceaux, appelés segments, est également précisé lors de l'initialisation. Les types qui peuvent être stockés sont `Event` et `EventLink`.

Pour ajouter une valeur à la fin du tableau, le programme utilise la méthode `append(element)` qui renvoie l'index de la valeur insérée ; s'il n'y a plus assez de place sur le segment actuel, cette méthode alloue de la mémoire pour un nouveau segment et poursuit l'ajout. Pour récupérer la valeur par la suite, il suffit d'appeler la méthode `get(index)` avec l'index obtenu lors de l'ajout. Finalement, la méthode `length()` renvoie le nombre total de valeurs qui ont été ajoutées.

L'avantage du tableau segmenté est qu'il permet d'avoir un tableau qui s'ajuste automatiquement lorsque l'on ajoute des valeurs, tout en fonctionnant comme un tableau normal pour l'accès aux valeurs. Les désavantages sont qu'il faut connaître à l'avance la taille maximale du tableau (ce qui est le cas dans notre programme) et qu'il est possible d'avoir des gaspillages de mémoire s'il y a beaucoup de segments presque vides (d'où l'intérêt de choisir des segments de bonne taille).

2.6 `Sensor`

La classe `Sensor` représente un capteur en stockant son ID dans la propriété `id` (qui peut être récupérée avec la méthode `getId()`) ainsi qu'un tableau segmenté d'`EventLink`s qui permet de trouver tous les événements générés par ce capteur.

Pour lier un événement à ce capteur, il faut d'abord créer et stocker l'événement, puis créer un lien et le passer à la méthode `addEventLink(el)`. Le nombre de liens (et donc d'événements) peut ensuite être récupéré avec la méthode `getNumEvents()`, et un lien individuel peut être récupéré avec la méthode `getEventLink(i)`. Il est également possible de récupérer directement l'événement, s'il n'est pas important de connaître la journée de la semaine, en utilisant la méthode `getEvent(i, events)` (à noter qu'il faut passer la table `events` de `City` pour que la méthode trouve l'événement).

3 Commandes

Les cinq commandes rentrées par l'utilisateur ont des méthodes correspondantes dans la classe `City`. Comme le traitement de ces commandes est l'aspect le plus lent du programme, nous avons choisi de désactiver la synchronisation entre la bibliothèque C++ `<ios>` et la bibliothèque C `<cstdio>`, ce qui permet, dans certains cas, de doubler la performance de la saisie d'informations.

Note : à priori, tous les paramètres pour les méthodes correspondant aux commandes sont identiques à ceux de la commande, sauf pour le jour de la semaine qui doit être compris entre 0 et 6, alors qu'il est saisi avec une valeur comprise entre 1 et 7.

Avertissement : si une commande est soumise avec des paramètres incorrects (par exemple un jour du mois avec la valeur 32), le programme termine immédiatement avec le code de sortie 1.

3.1 ADD <id> <yyyy> <mm> <dd> <h> <m> <d7> <trafic>

La commande `ADD` est associée à la méthode `addEvent`, qui ajoute un nouvel événement d'un capteur à la liste des événements enregistrés. Elle prend en paramètre des valeurs positives correspondant à l'identifiant du capteur, l'état du trafic, le mois (compris entre 5 et 9), le jour (compris entre 1 et le nombre de jours du mois), l'heure (inférieure à 24), la minute (inférieure à 59) et le jour de la semaine (compris entre 1 et 7).

Cette méthode vérifie d'abord si l'identifiant du capteur correspondant à l'événement à ajouter existe déjà, auquel cas elle ajoutera cet événement au capteur correspondant ; sinon, elle en crée un nouveau d'abord. Une fois l'identifiant du capteur correspondant trouvé ou créé, la méthode crée l'événement et un lien vers cet événement. L'événement en soi est stocké dans le tableau des jours de la semaine `events`, tandis que le lien est associé à un capteur dans le tableau des capteurs `sensors`.

3.2 STATS_C <idCaptor>

La commande `STATS_C` est associée à la méthode `displaySensorStateStats`, qui affiche les statistiques de trafic pour un capteur particulier, dont l'identifiant (un nombre naturel) est passé en paramètre. Cette méthode localise le capteur, itère sur chaque événement qui y est lié, compte le nombre d'états différents qui lui sont associés, puis affiche la fréquence d'apparition de chaque état par rapport au nombre total d'événements.

Si le capteur n'a pas été trouvé, rien n'est affiché en sortie.

3.3 JAM_DH <D7>

La commande `JAM_DH` est associée à la méthode `displayDayTrafficJamStats`, qui affiche les statistiques sur les embouteillages par heure, pour un jour de la semaine passé en paramètre (valeur positive comprise entre 0 et 6). Cette méthode localise le jour de la semaine, compte le nombre de fois où l'état était R ou N (ce qui correspond à un embouteillage) pour cette journée, pour chaque heure, puis affiche la fréquence d'embouteillage par heure relatif au nombre total d'événements pour cette heure, pour chaque heure de la journée (de 0h à 23h).

3.4 STATS_D7 <D7>

La commande `STATS_D7` est associée à la méthode `displayDayStateStats`, qui affiche les statistiques de trafic pour les jours de la semaine, pour un jour de la semaine passé en paramètre (valeur positive comprise entre 0 et 6). itère sur chaque événement de cette journée, compte le nombre d'états

différents qui lui sont associés, puis affiche la fréquence d'apparition de chaque état par rapport au nombre total d'événements.

3.5 `OPT <D7> <H_start> <H_end> <seg_count> <seg_1> ... <seg_n>`

La commande `OPT` est associée à la méthode `displayOptimalDepartureTime`, qui calcule le meilleur moment pour commencer à parcourir un ensemble de segments (chacun correspondant à un capteur). Elle accepte en paramètre un jour de la semaine (valeur positive comprise entre 0 et 6) sur lequel les calculs seront effectués (après calcul de la moyenne de toutes les instances de ce jour de la semaine), une plage horaire `hStart - hEnd` pour autoriser des départs seulement dans cette plage horaire, le nombre de segments (un nombre naturel strictement positif) et finalement un tableau de segments, identifiés par l'ID du capteur qui leur correspond (un nombre naturel). Si un capteur n'a pas été trouvé, rien n'est affiché en sortie.

La méthode commence par créer une journée de la semaine moyenne. Elle parcourt tous les capteurs des segments, puis parcourt chaque événement lié à ces capteurs pour garder seulement les événements qui ont eu lieu ce jour de la semaine et à partir de l'heure de départ de la plage horaire. Le jour de la semaine moyen est stocké sous la forme d'un tableau multidimensionnel qui attribue à chaque segment le nombre total de minutes entre l'heure de départ `hStart` et 0 :00 du jour suivant (afin de pouvoir couvrir tous les événements qui auraient eu lieu entre-temps) ; à chaque minute, un tableau de 4 entiers est attribué pour représenter le nombre d'états différents qui ont eu lieu ce jour de la semaine à cette heure et minute exactes.

Exemple : supposons que l'on exécute cette méthode pour un mercredi, avec pour premier segment le capteur 42. Pour créer le jour moyen, on parcourt d'abord tous les événements du capteur 42. On trouve qu'il y a eu quatre mercredis avec des événements pour ce capteur. Les états à 13h51 sur ces mercredis étaient V, R, J, et J. Donc, dans le tableau du capteur 42, à l'index correspondant à 13h51, on stocke la valeur `{1, 2, 1, 0}` (V, J, R, N).

Une fois que la journée moyenne a été créée, le calcul du meilleur moment de départ commence. Pour chaque minute de la plage horaire considérée, un nouveau voyage est essayé en itérant sur chaque segment. Un compteur compte le nombre de minutes écoulées depuis le début du voyage : à chaque nouveau segment, ce compteur est incrémenté selon l'état moyen du trafic à ce moment-là du voyage à ce capteur-là ; V ajoute 1 minute au temps de voyage, J ajoute 2 minutes, R ajoute 4 minutes et N ajoute 10 minutes. Le voyage est considéré non valable si :

— le voyage a pris trop de temps et a dépassé la journée,
— le voyage a pris trop de temps et est déjà moins optimal qu'un autre voyage calculé auparavant,
— ou il n'y a pas d'informations sur l'état de la circulation au moment où on arrive sur un segment.

Si un voyage est déclaré non valable, on l'interrompt et on passe à la minute d'après pour recommencer à calculer un nouveau voyage. Si un voyage est valable, cela veut dire qu'il est plus optimal que tous les autres voyages calculés auparavant, donc le moment de départ et le temps de voyage associés sont stockés dans des variables. Si le temps d'un voyage valable est égal au nombre de segments, le calcul est immédiatement interrompu car cela signifie que l'état de la circulation était V pour chaque segment, ce qui est la meilleure situation possible.

Quand les calculs sont terminés, si au moins un voyage valable a été calculé, le numéro de la journée, l'heure de départ, la minute de départ et la durée du voyage sont affichés sur la sortie standard.