Droulers Nicolas

Duval Valentin

Lamour Antoine

Compte-rendu du projet d'Algorithmique et programmation 4

L'intégralité des fichiers se trouve dans le lien suivant : https://github.com/m4ddabs/TPALGO.

Pour l'utiliser correctement, voici quelques informations. Les fichiers du programme principal (méthode aboutie) possèdent le suffixe "_fs". Les fichiers du programme secondaire (méthode simple) possèdent le suffixe "_no_fast_sorting". Ces fichiers utilisent la bibliothèque rapport_deformations{.c, .h}. Pour chacune des méthodes, vous trouverez un programme « main » permettant de les tester.

Nous avons commencé par créer un algorithme répondant à la demande de la façon la plus simple possible. Le paquet simulé n'étant pas trié, on parcourt pour chaque position la totalité du paquet. On mesure par la suite la valeur absolue de la différence entre cette valeur et chacune des autres positions du paquet. A chaque fois qu'elle sera inférieure ou égale à 100, on incrémente de 1 le nombre de voisins pour la position étudiée. Si le nombre de voisins après avoir parcouru tout le paquet est supérieur ou égal à 100, alors on pourra déclarer une alerte.

Cette solution est simple et fonctionne, mais elle n'est cependant pas du tout optimisée. En effet, la compilation est tout d'abord très longue (85 secondes) du fait que l'algorithme parcourt tout le paquet pour chaque position. De plus, ce programme ne gère pas les doublons (une même position peut subir plusieurs déformations donc elle pourrait avoir plusieurs alertes, or chaque position ne peut avoir qu'une seule alerte). Nous avons donc créé un programme afin de vérifier si une position se situe déjà dans le tableau d'alertes ou non :

Algorithme vérification : (position, tab_alerte, nb_deform_locales) →() **selon**

i←1

Tant que i<= ncomp(tab_alerte) **répéter :**

(pos, nbdeform) ← tab_alerte (i)

Si position = pos **alors** (« position déjà présente, le programme s'arrête ») **.**

.

pos ← position

nbdeform ← nb_deform_locales

tab_alerte (i) ← (pos, nbdeform)

.

Cette fonction sert à vérifier si une position existe déjà dans le tableau d'alertes. Si elle existe, elle n'ajoutera rien au tableau, et si elle n'existe pas, elle ajoutera la nouvelle alerte au tableau. Cette

fonction prend en entrée la position de l'alerte à vérifier, un tableau d'alertes et le nombre de déformations locales dans cette position à vérifier.

L'algorithme suivant se charge d'identifier les alertes.

La fonction ChercheAlerte prend comme entrée un paquet. Elle retourne un tableau qui contient toutes les alertes. Si il n'y a pas d'alertes, elle ne retourne 0.

Algorithme chercheAlerte (paquet) → (alertes) **selon** :

alertes ← 0

i ← 1

k ← 1

nmbdeformlocales ← 0

premierealerte ← 0

premierepos ← 0

Tant que premierealerte = 0 **et** i != ncomp(paquet) **répéter**:

Tant que k<=ncomp(paquet) **répéter**:

Si |paquet(k)-paquet(i)| <= 100 **alors**

nmbdeformlocales ← nmbdeformlocales + 1

.

k ← k+1

.

Si nmbdeformlocales >= 100 **alors**

alertes ← tab 1 ((position,nmb_deformation_locales))

alertes(1) ← (paquet(i),nmbdeformlocales)

nmbdeformlocales ← 0

premierespos ← i + 1

premierealerte ← 1

Sinon nmbdeformlocales ← 0, i ← i+1 **.**

.

Si alertes != 0 **alors**

i←premierepos

k←1

Tant que i <=ncomp(paquet) **répéter** :

Tant que k<=ncomp(paquet) **répéter** :

Si |paquet(k)-paquet(i)| <= 100 **alors**

nmbdeformlocales ← nmbdeformlocales + 1

.

.

Si nmbdeformlocales >= 100 **alors**

vérification(paquet(i), alertes, nmbdeformlocales)

nmbdeformlocales ← 0

Sinon nmbdeformlocales ← 0 **.**

.

.

.

L'algorithme de vérification a une complexité linéaire de par la présence de l'unique boucle tant que. L'algorithme pour trouver les alertes a une complexité quadratique n^2.

Ce programme détecte 31196 alertes, après simulation de 100'000 déformations sur 1'000'000 positions (correspondant à la longueur du câble).

Deuxièmement, nous avons cherché à créer un nouveau programme, afin qu'il soit le plus rapide possible. Ainsi, nous nous sommes tout d'abord dit qu'il allait falloir trier le paquet, afin de réduire la zone de recherche des voisins et dans l'optique de gagner du temps à la compilation. La méthode utilisée ici est le « tri rapide ». Il fonctionne à partir d'un pivot afin de ranger toutes les valeurs dans l'ordre croissant. On utilise une fonction récupérée d'internet (sur le lien suivant : https://waytolearnx.com/2019/08/tri-rapide-en-c.html) :

Algorithme TriRapide : (tab, first, last) → () **selon**

pivot ← 0, i ← 0, j ← 0

Si (first<last) **alors :**

pivot ← first

i ← first

j ← last

Tant que i<j **répéter**

Tant que tab(i) <= tab(pivot) **et** i < last **répéter** i ← i+1 **.**

Tant que tab(j) > tab(pivot) **répéter** j ← j-1 **.**

Si (i<j) **alors** (tab(i), tab(j)) ← (tab(j), tab(i))**.**

(tab(pivot), tab(j)) ← (tab(j), tab(pivot))

TriRapid(tab, first, j-1)

TriRapid(tab, j+1, last)

.

.

Après utilisation de cette fonction, on obtient donc un paquet de positions triées dans l'ordre croissant. Nous nous sommes ensuite intéressés à la façon par laquelle nous allions résoudre le problème efficacement. Nous avons abouti sur l'idée suivante.

Tout d'abord, nous distinguerons indice dans le tableau paquet et valeur de la position à cet indice. Pour chaque indice, on prend la valeur de la position ± 100. Dans cet intervalle, on compte le nombre de positions, ce nombre sera le nombre de voisins de la position étudiée. Illustrons cela avec un exemple :

Indice dans le tableau	20	21	22	23	24	25
Valeur de la position	72	121	187	193	218	296

Ici, si on étudie l'indice 22 (position 187) et que l'on applique la méthode détaillée ci-dessus, le nombre de voisins sera de 3. A gauche, l'indice 21 uniquement sera comptabilisé et à droite, les indices 23-24 seront comptabilisés. Les indices 20 et 25 ne seront pas pris en compte car 72 et 296 n'appartiennent pas à l'intervalle [187-100 ; 187+100].

La fonction de recherche des alertes est la suivante. Elle prend en paramètre un paquet de positions ainsi qu'un tableau d'alertes. Pour chaque position, elle établit l'intervalle ± 100 comme expliqué ci-dessus tout en faisant attention grâce aux indices de ne pas sortir du tableau paquet. Si elle trouve une alerte et que le tableau d'alertes est vide, elle l'ajoute automatiquement. Sinon, elle utilise la fonction verification pour ne pas avoir de doublons. La fonction repérer_alertes renvoie en sortie le tableau d'alertes complet.

Algorithme repérer_alertes : (paquet, tab_alerte) →tab_alertes **selon**

i←1, j←1, voisins←0, NOMBRE_DEFORMATIONS ← 100000, SEUIL_ALERTE ← 100

Tant que i<=NOMBRE_DEFORMATIONS **répéter**

j←i

Tant que (paquet(i)-100)<=paquet(j) **et** j>0 **répéter**

voisins←voisins+1, j←j-1 **.**

j←i+1

Tant que paquet(j)<=(paquet(i)+100) **et** j<NOMBRE_DEFORMATIONS **répéter**

voisins←voisins+1, j←j+1 **.**

Si voisins>=SEUIL_ALERTE **alors**

Si tab_alerte = NULL **alors**

(pos, nbdeform) ← tab_alerte(1)

pos ← paquet(i), nbdeform ← voisins

Sinon verification(paquet(i), alerte, taille_tab_alerte, voisins) **.**

.

.

.

Cet algorithme a une complexité quadratique n^2 : on observe deux boucles tant que imbriquées.

Grâce à cette nouvelle méthode, le programme ne va plus parcourir tout le tableau à chaque fois donc on aura un gain de temps notable. Notons de plus qu'on utilisera le programme vérification pour éliminer les doublons. Après test, cette méthode permet au programme de se terminer en 3 secondes dans le cas le plus favorable et 5 secondes dans le cas le plus défavorable. Cette méthode est donc plus efficace que la première, mais pas suffisante pour remplir le cahier des charges. On note enfin que le nombre d'alertes trouvé est le même que pour la première méthode. Les deux méthodes sont donc peut-être fausses, elles restent cependant cohérentes.