INFO0947: Récursivité & Types Abstraits de Données

Groupe 33: Pavlov Aleksandr, Gendebien Alexandre

Table des matières

1 Introduction

Dans le cadre du cours INFO-0947, nous avons dû résoudre un problème donné et créer un algorithme en C capable de :

Créer une course de vélo fictive, composée d'escales représentées par des villes (et leurs coordonnées). Déterminer le meilleur temps (le plus petit) qu'un cycliste a mis pour parcourir la distance qui sépare deux villes, ainsi que le meilleur temps pour finir la course. Calculer le nombre total d'étapes qui composent la course et vérifier si la course forme un circuit.

Ce problème sera entièrement documenté en $\LaTeX$.

2 Spécifications Abstraites

2.1 TAD Escale

2.1.1 Syntaxe

Type : Escale

Utilise : Integer
String (nom)
Float (coordonnées)
Boolean

Opérations : escale_create : String $\times$ Float $\times$ Float $\rightarrow$ Escale
escale_get_name : Escale $\rightarrow$ String
escale_get_x : Escale $\rightarrow$ Float
escale_get_y : Escale $\rightarrow$ Float
escale_get_best_time : Escale $\rightarrow$ Float
escale_set_best_time : Escale $\times$ Float
escale_distance : Escale $\times$ Escale $\rightarrow$ Float
escale_equal : Escale $\times$ Escale $\rightarrow$ Boolean

2.1.2 Sémantique

Préconditions : $\forall n \in$ String, $\forall x, y \in$ Float, $\forall e \in$ Escale
$\forall x, -180 \leq x \leq 180, \forall y, -90 \leq y \leq 90$, escale_create(n, x, y)
escale_get_name(e)
escale_get_x(e)
escale_get_y(e)
escale_get_best_time(e)
$\forall t \geq 0$ Float, escale_set_best_time(e, t)
$\forall e1, e2 \in$ Escale, escale_distance(e1, e2)

Axiomes : $\forall e \in$ Escale, $\forall n \in$ String, $\forall x, y, t \in$ Float
escale_get_name(escale_create(n, x, y)) = n
escale_get_x(escale_create(n, x, y)) = x
escale_get_y(escale_create(n, x, y)) = y
escale_get_best_time(escale_create(n, x, y)) = 0
escale_distance(escale_create(n1, x1, y1), escale_create(n2, x2, y2)) = Haversine formula

escale_equal(escale_create(n1, x, y), escale_create(n2, x, y)) = true
escale_equal(escale_create(n1, x1, y1), escale_create(n2, x2, y2)) = false

		Opérations Internes	
		escale_create(·)	escale_set_best_time(·)
Observateurs	escale_get_name(·)	✓	∅
	escale_get_x(·)	✓	∅
	escale_get_y(·)	✓	∅
	escale_get_best_time(·)	✓	∅
	escale_distance(·)	✓	∅
	escale_equal(·)	✓	∅

2.2 TAD Course

2.2.1 Syntaxe

Type : Course

Utilise : Escale
Integer (index, comptage)
Float (temps total, meilleur temps)
Boolean

Opérations : course_create : Escale $\times$ Escale $\to$ Course
course_is_circuit : Course $\to$ Boolean
course_get_escales_count : Course $\to$ Integer
course_get_stages_count : Course $\to$ Integer
course_total_time : Course $\to$ Float
course_best_time_at : Course $\times$ Integer $\to$ Float
course_append : Course $\times$ Escale $\to$ Course
course_pop : Course $\to$ Course

2.2.2 Sémantique

Préconditions : $\forall e, e1, e2 \in$ Escale, $\forall n \in$ String, $\forall x, y, t \in$ Float $\forall i \in$ Integer
$\forall e1, e2 \in$ Escale, escale_get_best_time(e1) $= 0 \wedge$ escale_equal(e1, e2) = false, course_create(e1, e2)
course_is_circuit(c)
course_get_escales_count(c)
course_get_stages_count(c)
course_total_time(c)
$\forall i \in$ Integer, $0 \leq i <$course_get_escales_count(c), course_best_time_at(c, i)
course_append(c, e)
$\forall c \in$ Course, course_get_escales_count(c) > 0, course_pop(c)

Axiomes : $\forall c \in$ Course, $\forall e, e1, e2, e3 \in$ Escale, $\forall n \in$ String, $\forall x, y, t \in$ Float $\forall i \in$ Integer
course_create(e1, e2) = course_pop(course_append(course_create(e1, e2), e3))
course_is_circuit(course_create(e1, e2)) = false

$$\begin{aligned}
&\text{course_is_circuit}(\text{course_append}(\text{course_create}(e1, e2), e1)) = \text{true} \\
&\text{course_is_circuit}(\text{course_append}(\text{course_create}(e1, e2), e3)) = \text{false} \\
&\text{course_is_circuit}(\text{course_pop}(\text{course_create}(e1, e2))) = \text{false} \\
&\text{course_get_escales_count}(\text{course_create}(e1, e2)) = 2 \\
&\text{course_get_escales_count}(\text{course_append}(c, e)) = \text{course_get_escales_count}(c) + 1 \\
&\text{course_get_escales_count}(\text{course_pop}(c)) = \text{course_get_escales_count}(c) \text{ - } 1 \\
&\text{course_get_stages_count}(\text{course_create}(e1, e2)) = 1 \\
&\text{course_get_stages_count}(\text{course_append}(c, e)) = \text{course_get_stages_count}(c) + 1 \\
&\text{course_get_stages_count}(\text{course_pop}(c)) = \text{course_get_stages_count}(c) \text{ - } 1 \\
&\text{course_get_stages_count}(\text{course_pop}(\text{course_pop}(\text{course_create}(e1, e2)))) = 0 \\
&\text{course_total_time}(\text{course_create}(e1, e2)) = \text{escale_get_best_time}(e1) + \text{escale_get_best_time}(e2) \\
&\text{course_total_time}(\text{course_append}(c, e)) = \text{course_total_time}(c) + \text{escale_get_best_time}(e) \\
&\text{course_total_time}(\text{course_pop}(\text{course_append}(c, e))) = \text{course_total_time}(c) \\
&\text{course_best_time_at}(\text{course_create}(e1, e2), 1) = \text{escale_get_best_time}(e2) \\
&\text{course_best_time_at}(\text{course_append}(c, e), \text{course_get_escales_count}(c)) = \text{escale_get_best_time}(e) \\
&\text{course_best_time_at}(\text{course_pop}(\text{course_create}(e1, e2)), 0) = \text{escale_get_best_time}(e1)
\end{aligned}$$

		Opérations Internes		
		course_create(·)	course_append(·)	course_pop(·)
Observateurs	course_is_circuit(·)	✓	✓	✓
	course_get_escales_count(·)	✓	✓	✓
	course_get_stages_count(·)	✓	✓	✓
	course_total_time(·)	✓	✓	✓
	course_best_time_at(·)	✓	✓	✓

3 Structures de Données

Pour implémenter les différents TAD, nous avons choisi deux types de structures de données : le tableau dynamique et la liste chaînée.

3.1 Escale

```
typedef struct Escale {
    char *name;
    double x;
    double y;
    double time;
} Escale;
```

Listing 1 – Structure de Escale

3.2 Course (Tableau)

```
typedef struct Course {
    size_t escales_size;
    size_t escales_count;
```

```c
        Escale **escales;
    } Course;
```

Listing 2 – Structure de Course (tableau)

3.2.1 Avantages

- Accès rapide aux éléments par leur indice ($O(1)$).
- Moins de surcharge mémoire due aux pointeurs supplémentaires.
- Facile à parcourir séquentiellement.

3.2.2 Inconvénients

- Redimensionnement coûteux si la taille initiale est insuffisante ($O(n)$).
- Ajout et suppression au milieu nécessitent un déplacement des éléments ($O(n)$).

3.3 Course (Liste Chaînée)

```c
    typedef struct Course {
        Escale *escale;
        Course *next;
    } Course;
```

Listing 3 – Structure de Course (liste chainée)

3.3.1 Avantages

- Insertion et suppression en temps constant ($O(1)$) sans déplacement des éléments.
- Taille flexible sans besoin de redimensionnement.

3.3.2 Inconvénients

- Accès séquentiel aux éléments ($O(n)$) au lieu d'un accès direct.
- Surcharge mémoire due aux pointeurs supplémentaires.

Ces choix de structures de données permettent de répondre aux différentes exigences du problème. Le tableau est idéal pour un accès rapide et indexé, tandis que la liste chaînée convient mieux aux modifications fréquentes et dynamiques de la course.

3.4 Structure de données

4 Specifications

4.1 Constructeur

course_create_list : Escale $\times$ Escale $\rightarrow$ Course

pre : $e_1 \neq e_2 \wedge e_1 \neq \text{NULL} \wedge e_2 \neq \text{NULL} \wedge \text{escale\_time}(e_1) = 0$

post :

taille course à la création = 2

première escale de la course = e_1

deuxième escale de la course = e_2

course_create_array : Escale $\times$ Escale $\rightarrow$ Course
pre : $e_1 \neq e_2 \wedge e_1 \neq$ NULL $\wedge\, e_2 \neq$ NULL $\wedge$ escale_time$(e_1) = 0$
post :
  course $\neq$ NULL
  taille de la course $= 2$
  première escale de la course $= e_1$
  deuxième escale de la course $= e_2$

escale_create : String $\times\, \mathbb{R} \times \mathbb{R} \rightarrow$ Escale
pre : name $\neq$ NULL
post :
  escale_name$(e) =$ name
  escale_x$(e) = x$
  escale_y$(e) = y$
  escale_best_time$(e) = 0$

4.2 Transformateur

escale_set_best_time : Escale $\times\, \mathbb{R} \rightarrow$ Void
pre : $e \neq$ NULL
post :
  escale_best_time$(e) =$ best_time

course_append : Course $\times$ Escale $\rightarrow$ Course
pre : $C \neq$ NULL $\wedge\, e \neq$ NULL
post :
  course_size$(C') =$ course_size$(C) + 1$
  course_last$(C') = e$

4.3 Observateur

course_is_circuit : Course $\rightarrow \mathbb{B}$
pre : $C \neq$ NULL
post :
si première étape = dernière étapereturn :true
sinon return :false

course_best_time : Course $\times\, \mathbb{N} \rightarrow \mathbb{R}$
pre : $C \neq$ NULL $\wedge\, 0 \leq$ index $<$ course_size(C)
post :
  result = escale_time(course_at$(C$, index$)$)

escale_get_x : Escale $\to \mathbb{R}$
pre : $e \neq$ NULL
post :
  result = escale_x(e)

escale_get_y : Escale $\to \mathbb{R}$
pre : $e \neq$ NULL
post :
  result = escale_y(e)

escale_distance : Escale $\times$ Escale $\to \mathbb{R}$
pre : $e_1 \neq$ NULL $\wedge$ $e_2 \neq$ NULL
post :

$$d = 2R \arcsin\left(\sqrt{\sin^2\left(\frac{\Delta\phi}{2}\right) + \cos(\phi_1)\cos(\phi_2)\sin^2\left(\frac{\Delta\lambda}{2}\right)}\right)$$

5 Invariants

5.1 Invariant du temps total

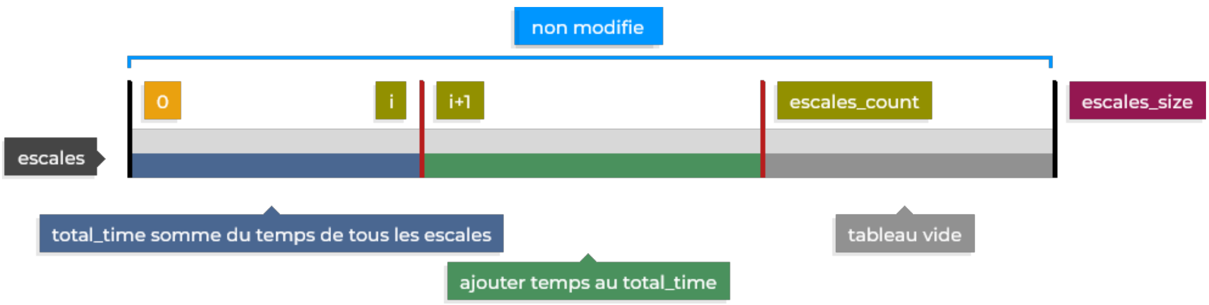

FIGURE 1 – Invariant graphique

Invariant formel :
$escale = escale_0$
$\wedge$
$0 < i < escale\_count$
$\wedge$
$total\_time = \sum_{i=0}^{escale\_count-1} \texttt{escale\_get\_best\_time}(escales[i])$

6 Implémentations Récursives

Définition récursive :
course_get_escales_count(course) =

$$\begin{cases} return\ 0; & \text{if } course == NULL \\ return\ course\_get\_escales\_count(course->next) + 1; & \text{otherwise} \end{cases}$$

$$\text{course\_get\_stages\_count(course)} = \begin{cases} return\ 0; & \text{if } course == NULL \\ return\ 0; & \text{if } course- > next == NULL \\ return\ course\_get\_stages\_count(course- > next) + 1; & \text{otherwise} \end{cases}$$

$$\text{course\_total\_time(course)} = \begin{cases} return\ 0; & \text{if } course == NULL \\ return\ (\\ course\_total\_time(course- > next) + \\ escale\_get\_best\_time(course- > escale) \\ ); & \text{otherwise} \end{cases}$$

$$\text{course\_best\_time\_at(course, index)} = \begin{cases} return\ escale\_get\_best\_time(course- > escale); & \text{if } index == 0 \\ return\ course\_best\_time\_at(course- > next, index - 1); & \text{otherwise} \end{cases}$$

$$\text{course\_append(course, escale)} = \begin{cases} course = malloc(sizeof(Course)); \\ course- > escale = escale; \\ course- > next = NULL; \\ return\ course; & \text{if } course == NULL \\ \\ course- > next = course\_append(course- > next, escale); \\ return\ course; & \text{otherwise} \end{cases}$$

$$\text{course\_pop(course)} = \begin{cases} free(course- > escale); \\ free(course); \\ return\ NULL; & \text{if } course- > next == NULL \\ \\ course- > next = course_pop(course- > next); \\ return\ course; & \text{otherwise} \end{cases}$$

$$\text{course\_free(course)} = \begin{cases} return; & \text{if } course- > next == NULL \\ \\ course\_free(course- > next); \\ free(course- > escale); \\ free(course); & \text{otherwise} \end{cases}$$

$$\text{course\_last(course)} = \begin{cases} return\ course; & \text{if } course- > next == NULL \\ \\ return\ course\_last(course- > next); & \text{otherwise} \end{cases}$$

7 Complexité

7.1 Fonctions sur Escale

$$
\begin{aligned}
\texttt{escale\_create} &: \mathcal{O}(1) \\
\texttt{escale\_get\_name} &: \mathcal{O}(1) \\
\texttt{escale\_get\_x} &: \mathcal{O}(1) \\
\texttt{escale\_get\_y} &: \mathcal{O}(1) \\
\texttt{escale\_get\_best\_time} &: \mathcal{O}(1) \\
\texttt{escale\_set\_best\_time} &: \mathcal{O}(1) \\
\texttt{escale\_distance} &: \mathcal{O}(1) \\
\texttt{escale\_equal} &: \mathcal{O}(1)
\end{aligned}
$$

7.2 Opérations sur Course (tableau dynamique)

Soit n le nombre d'escales, S la capacité du tableau.

$$
\begin{aligned}
\texttt{course\_create} &: \mathcal{O}(1) \\
\texttt{course\_is\_circuit} &: \mathcal{O}(1) \\
\texttt{course\_get\_escales\_count} &: \mathcal{O}(1) \\
\texttt{course\_get\_stages\_count} &: \mathcal{O}(1) \\
\texttt{course\_total\_time} &: \mathcal{O}(n) \\
\texttt{course\_best\_time\_at}(i) &: \mathcal{O}(1) \\
\texttt{course\_append} &: \begin{cases} \mathcal{O}(1) \\ \mathcal{O}(n) & \text{(réallocation)} \end{cases} \\
\texttt{course\_pop} &: \mathcal{O}(1)
\end{aligned}
$$

7.3 Opérations sur `Course` (liste chaînée)

Soit n le nombre d'escales.

$$
\begin{aligned}
\texttt{course\_create} &: \mathcal{O}(1) \\
\texttt{course\_is\_circuit} &: \mathcal{O}(n) \\
\texttt{course\_get\_escales\_count} &: \mathcal{O}(n) \\
\texttt{course\_get\_stages\_count} &: \mathcal{O}(n) \\
\texttt{course\_total\_time} &: \mathcal{O}(n) \\
\texttt{course\_best\_time\_at}(i) &: \mathcal{O}(i) \\
\texttt{course\_append} &: \mathcal{O}(n) \\
\texttt{course\_pop} &: \mathcal{O}(n)
\end{aligned}
$$

8 Tests Unitaires

Les tests unitaires sont situés dans le dossier `test/` et utilisent le framework `seatest`. Pour chaque implémentation (`course_liste` et `course_tableau`), deux fonctions principales sont testées :

— `test_course_append` : vérifie que l'ajout d'escales modifie correctement le nombre d'escales dans la course.

— `test_course_total_time` : vérifie que la somme des temps des escales est correcte après modification des temps et suppression d'éléments.

Les assertions utilisées sont `assert_ulong_equal` pour les entiers et `assert_double_equal` pour les valeurs réelles.

Justification des choix : Les tests ciblent les opérations principales et les cas limites.

Limite : Seules les fonctions d'ajout et de calcul du temps total sont testées. Les autres fonctions publiques ne sont pas couvertes par les tests actuels.

9 Conclusion

pour conclure ce rapport, nous pouvons dire que nous avons réussi à répondre au problème dans ça globalité. En créant un programme capable de créer une course fictive et d'en utiliser tous les éléments qui la composent affin de trouver par quelles ville la course passe, le meilleur temp qu'à mis un cycliste pour parcourir la distance entres deux villes, ou encore si la course forme un circuit.