IN 103 - TD 5

Solution

On se propose de réaliser un mini-embryon de navigateur dans un plan du métro de Paris capable de nous donner un chemin entre deux stations. Il vous a été mentionné en cours que la structure de données permettant de représenter un réseau est le graphe.

1 Navigation entre les stations

Q1 Ce graphe est-il orienté ou non-orienté ?

Solution

Dans la réalité, la ligne 10 n'est pas totalement à double sens. Une petite section comporte une « boucle ». Mais pour simplifier l'exercice, nous ignorerons cette ligne et aurons donc un graphe non-orienté.

Nous choisissons donc de le considérer non-orienté. Les données du plan simplifié (pas de ligne 10 et pas de RER) vous sont fournies sous forme textuelle dans le fichier `plan.txt`. La structure de ce fichier est très simple. Une ligne de métro débute par son nom puis vient la liste de toutes les stations qui se suivent jusqu'à un caractère « . » qui marque la fin de la ligne de métro décrite. Chaque élément est séparé par un retour à la ligne (pas de métro !).

Pour référence, il vous est également donné le fichier `plan.jpg` qui est un extrait du plan de métro de la RATP, ce qui vous permettra de voir où se trouvent effectivement les stations du plan simplifié et les connexions entre les lignes (pour tester votre programme).

Q2 Proposez une structure de nœud du graphe, pour représenter une station. Pensez à l'utilisation que nous voulons faire de ce graphe.

Solution

Une station a un nom, donc il faudra un champ à cet effet. Ensuite, une station est connectée à **ses** voisins dans le graphe. Donc il nous faudra une liste chaînée de voisins, donc de stations. Pour terminer, vu que ce graphe est (heureusement) cyclique, il nous faudra un marqueur pour savoir si l'on est déjà passé par une station au long d'un parcours.

2 Création du plan

La structure de données que nous allons utiliser vous est fournie dans le fichier `station.h`. Pour le moment, **ne vous occupez pas** du dernier champ de la structure.

Q3 Quel va être le prototype de la fonction `load_map` chargée de créer le graphe en mémoire à partir d'un fichier de description ?

Solution

Cette fonction devra prendre en argument le nom du fichier. Pour ce qui est de la valeur retournée, on pourrait lui faire retourner un nœud quelconque du graphe car il n'y a pas de ≪racine ≫ dans ce graphe. En fait, ce ne nous serait pas d'une grande utilité car lorsque l'on voudra trouver une station pour calculer un trajet, on ne va pas le faire en parcourant le graphe (qui a des cycles). Donc la seule chose intéressante que l'on puisse retourner c'est un booléen disant si la construction du graphe a été un succès.

bool load_map (**char** ∗name);

Par contre, bien évidemment il faudra que cette fonction mémorise toutes les stations crées afin d'être capable de rechercher une station particulière pour :

— créer le graphe en partageant les stations,

— effectuer une recherche de chemin (points de départ et d'arrivée).

Une structure de liste chaînée sera tout à fait adaptée puisque l'on ne connait pas à l'avance le nombre de stations.

Pour alléger le travail, il vous est donné, dans le fichier `load_map_skel.(c|h)`, le squelette de la fonction `load_map` que vous allez écrire et 2 fonctions :

— **struct** station_t∗ get_or_create_station (**char** ∗name)
qui prend en argument un nom de station et recherche (dans la liste chaînée globale `all_stations`) cette station ou la crée si elle n'existe pas et retourne un pointeur dessus. Ceci permet d'être certain que les stations existent de manière unique.

— **struct** stations_list_t ∗ get_all_stations (**void**)
qui retourne la tête de la liste chaînée des stations créées (en fait, retourne la variable globale `all_stations`). Cette fonction permet juste de ne pas laisser visible la variable globale depuis les autres fichiers.

Le but est maintenant d'écrire la fonction qui va charger le plan en mémoire et construire la graphe.

Q4 Quelle va être la forme de l'algorithme de création du graphe ?

Solution

```
Ouvrir le fichier en lecture et vérifier le succès
Tant que l'on arrive à lire une chaîne dans le fichier
  Ignorer la chaîne lue car c'est le nom de la ligne
  Tant que l'on arrive à lire une chaîne dans le fichier et
           que l'on n'a pas lu une fin de ligne de métro (i.e. un ".")
    Rechercher ou créer la station
    Si l'on n'est pas la première station de la ligne
      Se rajouter comme voisin de la station précédente
      Rajouter la station précédente comme voisin de nous-mêmes
Fermer le fichier
```

Pour lire les noms de station, on n'utilisera **pas** `fscanf` car ces noms peuvent être composés (i.e. avec des espaces). À la place on utilisera la fonction :

char∗ fgets (**char** ∗str, **int** size , FILE ∗in) ;

qui permet de lire une **ligne de texte** dans un fichier. Une ligne de texte est une suite de caractères terminée par un retour chariot (`'\n'`). Cette fonction prend en arguments la zone mémoire où mettre les caractères lus, le nombre maximal de caractères à lire et le descripteur du fichier. Cette fonction retourne `NULL` en fin de fichier (donc principe différent de `fscanf`/`feof`).

Attention Le retour chariot de fin de ligne est stocké dans la chaîne. Il faudra donc penser à le supprimer.

Q5 Écrivez, dans le fichier `load_map_skel.c` (que vous pouvez renommer), la fonction `load_map`.

Solution

```
   ------------- load_map.c --------------
bool load_map (char *name)
{
  bool end ;
  FILE *in ;
  char *read ;
  char buffer[256] ;
  struct station_t *prev_station ;

  in = fopen (name, "rb") ;
  if (in == NULL) return (false) ;

  /* NOTE: since I *really* want to assign [read] then test its value, I use
     a = in the test. To prevent the compiler from generating a warning, I
     explicitely surrounded the expression by ().
     Hence wo *do* need this double-layer of parentheses. */
  while ((read = fgets (buffer, 255, in))) {
    /* Première chaîne: supposée nom de ligne. */
    // printf ("Found line: %s\n", buffer) ;
    prev_station = NULL ;

    /* Lecture des stations jusqu'à un ".". */
    end = false ;
    while ((read = fgets (buffer, 255, in)) && (! end)) {
      unsigned int len = strlen (buffer) ;
      /* On retire l'éventuel retour à la ligne happé par fgets (). */
      if (buffer[len - 1] == '\n') (buffer[len - 1] = '\0') ;
      /* On vérifie si on a atteint une fin de ligne de métro. */
      if (strcmp (buffer, ".") == 0) {
        // printf ("End of line.\n") ;
        end = true ;
      }
      else {
        /* Nom de station trouvé. */
        struct station_t *new_station = get_or_create_station (buffer) ;
        /* Si on n'est pas la première station de la ligne... */
        if (prev_station != NULL) {
          /* On se rajoute comme voisin de la station précédente. */
          struct stations_list_t *lst =
            malloc (sizeof (struct stations_list_t)) ;
          lst->station = new_station ;
          /* On ajoute le voisin en tête de la liste de voisins. */
          lst->next = prev_station->neighbours ;
          prev_station->neighbours = lst ;
          /* On rajoute la station précédente comme voisin de nous-même. */
          lst = malloc (sizeof (struct stations_list_t)) ;
          lst->station = prev_station ;
          lst->next = new_station->neighbours ;
          new_station->neighbours = lst ;
        }

        prev_station = new_station ;
      }
    }
  }

  fclose (in) ;
  return (true) ;
}
```

Dans le fichier `utils.(c|h)` vous sont données 2 fonctions :

- `print_stations` : permet d'afficher toutes les stations de la liste chaînée passée en argument.
- `find_station` permet de rechercher dans une liste chaînée une station par son nom (retourne `NULL` en cas d'échec).

3 Recherche de chemin

Q6 Votre programme devra trouver le chemin **le plus court en nombre de stations** et afficher les stations formant ce chemin. De quelle manière faut-il parcourir le graphe ?

Solution

Il faut effectuer un parcours en largeur.

Pour alléger le travail, il vous est donné dans le fichier `queue.(c|h)`, un ensemble de fonctions permettant de créer, manipuler, libérer des **files** de **stations** (**struct** station_t).

```
-------------- queue.h --------------
```

```
#ifndef __QUEUE_H__
#define __QUEUE_H__
#include <stdbool.h>
#include "station.h"

struct queue_t {
  unsigned int max_nb ;       /* Nombre max d'éléments. */
  unsigned int cur_nb ;       /* Nombre actuel d'éléments. */
  unsigned int first ;        /* Indice du premier élément. */
  struct station_t **data ;
};

struct queue_t* queue_alloc (unsigned int max_size) ;
void queue_free (struct queue_t *q) ;
struct station_t* take (struct queue_t *q) ;
void enqueue (struct queue_t *q, struct station_t *pi) ;
bool is_empty (struct queue_t *q) ;
#endif
```

Q7 Écrivez la fonction `shortest_path` qui prend en argument 2 pointeurs sur station (1 station de départ et 1 station d'arrivée) et retourne un booléen disant s'il existe un chemin entre ces 2 stations, **en effectuant la recherche du plus court chemin**.

Solution

Bien évidemment, le graphe formé par le métro étant connexe, il y a toujours un chemin entre deux stations. Donc si votre programme répond « pas trouvé », vous êtes certain qu'il y a un problème quelque part.

Il faut noter que lorsque l'on a parcouru le graphe, les sommets visités ont eu leur marqueur `seen` mis à `true`. Donc ci l'on souhaite effectuer un nouveau parcours, il faudra au préalable remettre leur marqueur à `false`. Cela peut se faire facilement en parcourant la liste de tous les sommets du graphe (obtenue à l'aide de la fonction `get_all_stations` qui vous est donnée dans `load_map.(c|h)`).

```
-------------- search.c --------------
```

```c
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "station.h"
#include "load_map.h"
#include "queue.h"
#include "search.h"


bool shortest_path (struct station_t *src, struct station_t *dest)
{
  struct queue_t *queue = queue_alloc (512) ;
  bool found = false ;

  if (queue == NULL) {
    printf ("Erreur: impossible d'allouer la file.\n") ;
    return (false) ;
  }

  /* Racine = station de départ. La marquer. */
  src->seen = true ;
  /* La mettre dans la file. */
  enqueue (queue, src) ;
  /* Tant que la file n'est pas vide... */
  while ((! is_empty (queue)) && (! found)) {
    struct station_t *st = take (queue) ;
    printf ("Station courante: %s\n", st->name) ;
    /* Vérifier si on est arrivé à la destination. */
    if (st == dest) found = true ;
    else {
      /* Sinon on continue le parcours. */
      struct stations_list_t *neigh = st->neighbours ;
      while (neigh != NULL) {
        /* Vérifier si le voisin n'est pas déjà marqué. */
        if (! neigh->station->seen) {
          /* Pas marqué, donc le marquer et le mettre dans la file. */
          neigh->station->seen = true ;
          /* On se souvient que lorsque l'on traitera ce voisin, on y sera
             arrivé par l'intermédiaire du noeud actuel. */
          neigh->station->prev_in_path = st ;
          enqueue (queue, neigh->station) ;
        }
        /* Voisin suivant. */
        neigh = neigh->next ;
      }
    }
  }

  queue_free (queue) ;
  return (found) ;
}



/** Imprime le chemin trouve, en ordre inverse (donc, destination vers source).
    Si aucun chemin n'a été trouvé, le résultat sera imprédictible. */
void print_shortest_path_reversed (struct station_t *last)
{
  struct station_t *st = last ;
  while (st != NULL) {
    printf ("%s", st->name) ;
    st = st->prev_in_path ;
    /* Pour n'afficher une flèche que s'il y a bien d'autres stations après
       la courante. */
    if (st != NULL) printf (" <- ") ;
  }

  printf ("\n") ;
}
```

Q5 Que manque-t'il pour être capable d'afficher l'ensemble des stations du chemin **le plus court** trouvé ?

Solution

Il manque la capacité à retrouver par quel chemin on est passé. Donc il faudrait, en partant de la station destination effectivement trouvée être capable de ≪remonter ≫ le chemin en arrière. Donc pour chaque station appartenant au chemin, il faudrait mémoriser quelle était la station précédente dans le chemin.

La solution consiste à mémoriser lors du parcours, au moment d'insérer un voisin dans la file, quel est le noeud qui a causé cette insertion. En quelque sorte, qu'il est le ≪parent ≫ du voisin inséré dans la file.

Soit on maintient cette information dans une structure de données externe (tableau ou autre) soit on peut rajouter un champ dans la structure de noeuds `station_t` afin de mémoriser le pointeur vers le ≪parent ≫ qui nous a inséré dans la file. C'est le choix ici, d'où le champ `prev_in_path` présent dans la structure. C'est en fait une liste chaînée construite au sein du graphe.

Q6 Modifiez votre algorithme de parcours pour être capable d'afficher l'ensemble des stations du chemin **le plus court** trouvé. Pour simplifier, on pourra s'autoriser à afficher les stations composant le chemin en ordre inverse (donc, destination vers source).

Solution

Dans la fonction `shortest_path` on a rajouté le marquage par le champ `prev_in_path` de la structure `station_t` au moment d'enfiler un sommet et on a implémenté la fonction `print_shortest_path_reversed`.

Q7 Si vous avez le temps, écrivez votre `main`, sinon utilisez le fichier `main.c` qui vous est donné (il faut que vous ayez respecté les noms). Testez votre programme.