1 Graphes temporels

1) Dans le fichier `list.h`, on définit la structure `temps` puisqu'à la Q3, cela va être utile pour définir les arcs.

```
typedef struct {
  int heure;
  int minute;
} temps;
```

2) Dans le fichier `main.c`, on définit la fonction `Difference`.

```
int Difference(temps t1, temps t2){
  int m1 = t1.heure * 60 + t1.minute;
  int m2 = t2.heure * 60 + t2.minute;
  return abs(m2 - m1);
}
```

💬 Test de la fonction `Difference`

```
$ ./transports
Difference entre 10h15 et 11h45 : 90 minutes
```

3) On représente les graphes par une liste d'adjacence utilisant l'implémentation des listes chaînées déjà proposées. Dans le fichier `list.h`, on définit une nouvelle structure `arc` :

```
typedef struct {
  int u;
  int v;
  temps td;
  temps ta;
} arc;
```

Il faut ensuite adapter les listes :

```
struct list{
  arc value;
  struct list *next;
};
typedef struct list *list;
```

Dans cette implémentation imposée, le type `list` est un pointeur vers le type `struct list`.

Il faut également modifier les signatures et les définitions des fonctions `value`, `insertBegin` et `insertAfter` pour qu'elles gèrent des structures de type `arc` et non des `int`.

Finalement, il faut déclarer la structure de graphe. On déclare la librairie `graph` en créant un fichier `graph.h` et `graph.c`. On la déclare dans le fichier d'en-tête `graph.h`.

```
typedef struct {
  list *adj;   // tableau de listes
  int n;       // nombre de sommets
} graph;
```

On déclare et implémente également les fonctions utilitaires suivantes :

```
graph create_graph(int n);
void add_edge(graph g, int u, int v, temps td, temps ta);
void print_time(temps t);
void print_edge(arc e);
void print_graph(graph g);
```

On suppose que les sommets sont numérotés de 0 à $n-1$.

Implémentation d'un graphe temporel

Le choix de la correction va naturellement sur la création d'une librairie `graph`. Il faut discuter avec les candidat·e·s de leur proposition. Je pense que la question 6 aurait dû être posée à ce moment là pour que l'implémentation du graphe puisse être validée. On peut le demander aux candidat·e·s.

4) On traite successivement chaque transition imposée par la liste des sommets. Pour aller de u_i à u_{i+1}, on parcourt les arcs sortants de u_i et on choisit, parmi ceux allant vers u_{i+1} et partant après le temps courant, celui dont le départ est le plus tôt. On met à jour le temps courant avec son temps d'arrivée et on continue. Si une transition est impossible, le chemin n'existe pas. Sinon, la durée minimale est la différence entre le temps final et le temps initial.

```
bool before_or_equal(temps t1, temps t2){
  if (t1.heure < t2.heure) return true;
  if (t1.heure > t2.heure) return false;
  return t1.minute <= t2.minute;
}
int min_time_path(graph g, int vertices[], int p, temps t){
  if (p <= 1) return 0;
  temps t0 = t;
  temps cur = t;
  for (int i = 0; i < p - 1; i++){
    int start = vertices[i];
    int end = vertices[i + 1];
    list L = g.adj[start];
    bool found = false;
    arc best;
    while (!isEmpty(L)){
      arc e = value(L);
      if (e.v == end && before_or_equal(cur, e.td)){
        if (!found || before_or_equal(e.td, best.td)){
          best = e;
```

```
          found = true;
        }
      }
      L = next(L);
    }
    if (!found) return -1;
    cur = best.ta;
    // print_edge(best);
  }
  return Difference(t0, cur);
}
```

💬 Durée minimale

Par exemple :

- $(0, 1, 2, 3), 10h00$ a une durée minumum de 34 minutes.
- $(4, 3, 1, 2), 10h00$ n'a pas de chemin.
- $(1, 4, 3), 10h00$ a une durée minumum de 56 minutes.

```
$ ./transports
 (0 -> 1, 10:08, 10:13) (1 -> 2, 10:15, 10:27) (2 -> 3,
    10:31, 10:34)
Minimal duration (0,1,2,3): 34 minutes
 (4 -> 3, 10:00, 10:03) (3 -> 1, 10:33, 10:38)
Minimal duration (4,3,1,2): -1 minutes
 (0 -> 4, 10:48, 10:51) (4 -> 3, 10:53, 10:56)
Minimal duration (0,4,3): 56 minutes
```

5) Le graphe donné dans la figure 1 n'est pas fortement connexe puisqu'il n'y a pas de chemin temporel de D à C et de E à C. Si on ajoute l'arc proposé alors le graphe devient temporel.

6)

```
// Q6 Graphe - Convention : A=0, B=1, C=2, D=3, E=4
graph g = create_graph(5);
add_edge(g, 0, 1, (temps){10, 8},  (temps){10, 13});
add_edge(g, 0, 4, (temps){10, 48},  (temps){10, 51});
add_edge(g, 1, 2, (temps){10, 15}, (temps){10, 27});
add_edge(g, 2, 1, (temps){10, 42}, (temps){10, 49});
add_edge(g, 2, 3, (temps){10, 31}, (temps){10, 34});
add_edge(g, 3, 0, (temps){10, 42}, (temps){10, 46});
add_edge(g, 3, 1, (temps){10, 33}, (temps){10, 38});
add_edge(g, 4, 3, (temps){10, 0},  (temps){10, 3});
add_edge(g, 4, 3, (temps){10, 53}, (temps){10, 56});
/* Arc ajouté à la question 6 */
add_edge(g, 3, 0, (temps){10, 4}, (temps){10, 6});
```

7) Avec la périodicité 24 h, il faut adapter l'idée de la question 4 : si le prochain départ utile est déjà passé dans la journée, on le reprend le lendemain, donc on ajoute 24×60 minutes.

```
int min_time_path24(graph g, int vertices[], int p, temps t){
  if (p <= 1) return 0;
  int day = 24 * 60;
  int t0 = 60 * t.heure + t.minute;
  int cur = t0;
  for (int i = 0; i < p - 1; i++){
    int start = vertices[i];
    int end = vertices[i + 1];
    list L = g.adj[start];
    bool found = false;
    int best_arrival = 0;
    // arc best_edge;
    while (!isEmpty(L)){
      arc e = value(L);
      if (e.v == end){
        int departure = 60 * e.td.heure + e.td.minute;
        int arrival = 60 * e.ta.heure + e.ta.minute;
        // shift departure to the first feasible occurrence
        int departure_abs = departure;
        if (departure_abs < cur){
          departure_abs +=
        ((cur - departure_abs + day - 1) / day) * day;
        }
        // arrival follows the same duration
        int arrival_abs = departure_abs + (arrival - departure);
        if (!found || arrival_abs < best_arrival){
          best_arrival = arrival_abs;
      // best_edge = e;
          found = true;
        }
      }
      L = next(L);
    }
    if (!found) return -1;
    cur = best_arrival;
    // print_edge(best_edge);
  }
  return cur - t0;
}
```

💬 Durée minimale - périodicité de 24h

Par exemple :

- $(0, 1, 2, 3), 10h00$ a une durée minumum de 34 minutes.
- $(4, 3, 1, 2), 10h00$ a une durée de 1467 minutes donc le chemin termine le lendemain.
- $(1, 4, 3), 10h00$ a une durée minumum de 56 minutes.

```
$ ./transports
 (0 -> 1, 10:08, 10:13) (1 -> 2, 10:15, 10:27) (2 -> 3,
    10:31, 10:34)
Minimal duration (0,1,2,3): 34 minutes
 (4 -> 3, 10:00, 10:03) (3 -> 1, 10:33, 10:38) (1 -> 2,
    10:15, 10:27)
Minimal duration (4,3,1,2): 1467 minutes
 (0 -> 4, 10:48, 10:51) (4 -> 3, 10:53, 10:56)
Minimal duration (0,4,3): 56 minutes
```

8) Dans un graphe temporel 24h-périodique, chaque arc se répète indéfiniment tous les jours. Ainsi, dès qu'un arc existe entre deux sommets, il est toujours possible de l'emprunter, quitte à attendre le lendemain si nécessaire. Par conséquent, les contraintes temporelles ne limitent plus l'accessibilité entre sommets : seul compte le fait qu'un arc existe, indépendamment de son horaire.

On peut alors considérer le graphe orienté sous-jacent, obtenu en oubliant les temps et en ne conservant que les arcs (u, v). Le graphe temporel est fortement connexe si et seulement si ce graphe orienté classique est fortement connexe, c'est-à-dire si, pour tout couple de sommets (u, v), il existe un chemin de u vers v.

💬 Graphe fortement connexe

Discussions sur l'interprétation d'un graphe temporel.

2 Parsing de fichier

9)

```c
temps parse_time(char *s){
  temps t;
  int sec;
  if (sscanf(s, "%d:%d:%d", &t.heure, &t.minute, &sec) != 3){
    fprintf(stderr, "Invalid time format\n");
    exit(EXIT_FAILURE);
  }
  return t;
}
```

💬 Parsing

Si besoin, on peut parler de la fonction `sscanf` dont le fonctionnement ressemble à celui de `fscanf`. Par exemple, on peut parser `16:34:18`.

```
$ ./transports
16:34
```

10) On écrit trois fonctions. La fonction `count_stops` compte le nombre de sommets qu'il doit y avoir dans le graphe à partir du fichier `stops.txt`.

```c
int count_stops(char *filename){
  FILE *f = fopen(filename, "r");
  if (f == NULL){
    fprintf(stderr, "Error opening file %s\n", filename);
    exit(EXIT_FAILURE);
  }
  int count = 0;
  char buffer[256];
  /* skip header line */
  if (fgets(buffer, sizeof(buffer), f) == NULL){
    fclose(f);
    return 0;
  }
  /* count remaining lines */
  while (fgets(buffer, sizeof(buffer), f) != NULL){
    count++;
  }
  fclose(f);
  return count;
}
```

La fonction `read_stop_time` lit une ligne du fichier `stop_times.txt` et récupère les informations.

```c
bool read_stop_time_line(FILE *f, char *trip_id, temps *t, int *
    stop_id){
  char time_str[16];
  int unused1, unused2;
  int n = fscanf(f, "%s %s %d %d %d",
                 trip_id, time_str, stop_id, &unused1, &unused2);
  if (n != 5){
    return false;
  }
  *t = parse_time(time_str);
  return true;
}
```

La fonction `read_graph` a pour objectif de construire le graphe temporel à partir des fichiers `stops.txt` et `stop_times.txt`. Dans un premier temps, on lit le fichier `stops.txt` afin de déterminer le nombre de sommets du graphe, ce qui permet

d'initialiser celui-ci. Ensuite, on parcourt le fichier `stop_times.txt` ligne par ligne, en extrayant uniquement les informations utiles : l'identifiant du trajet (`trip_id`), le temps (`stop_time`) et l'identifiant de l'arrêt (`stop_id`), les autres colonnes étant ignorées.

L'idée essentielle repose sur le fait que les lignes correspondant à un même `trip_id` décrivent les arrêts successifs d'un même véhicule. Ainsi, deux lignes consécutives de même `trip_id` définissent un déplacement entre deux arrêts consécutifs. On mémorise donc les informations de la ligne précédente, et pour chaque nouvelle ligne lue, si le `trip_id` est identique, on ajoute au graphe un arc allant de l'arrêt précédent vers l'arrêt courant, avec pour temps de départ le temps précédent et pour temps d'arrivée le temps courant. On met ensuite à jour les informations mémorisées et on poursuit la lecture jusqu'à la fin du fichier.

```
graph read_graph(char *stops_file, char *stop_times_file){
  int n = count_stops(stops_file);
  graph g = create_graph(n);
  FILE *f = fopen(stop_times_file, "r");
  if (f == NULL){
    fprintf(stderr, "Error opening file %s\n", stop_times_file);
    exit(EXIT_FAILURE);
  }
  char buffer[256];
  // skip header line
  if (fgets(buffer, sizeof(buffer), f) == NULL){
    fclose(f);
    return g;
  }
  char prev_trip[64];
  char trip[64];
  int prev_stop;
  int stop;
  temps prev_time;
  temps time;
  // read first useful line
  if (!read_stop_time_line(f, prev_trip, &prev_time, &prev_stop)){
    fclose(f);
    return g;
  }
  // read remaining lines
  while (read_stop_time_line(f, trip, &time, &stop)){
    if (strcmp(prev_trip, trip) == 0){
      add_edge(g, prev_stop, stop, prev_time, time);
    }
    strcpy(prev_trip, trip);
    prev_stop = stop;
    prev_time = time;
  }
  fclose(f);
  return g;
}
```

3 Plus courts chemins

11) Les exemples suivent.

- *Chemin le plus tôt.* Ce critère est pertinent lorsqu'on souhaite arriver le plus tôt possible à destination, par exemple pour rejoindre des amis dès que possible. On suppose alors que l'on peut partir immédiatement, et l'objectif est de minimiser l'heure d'arrivée.
- *Chemin le plus tard.* Ce critère convient lorsqu'on doit être présent à une heure donnée, mais que l'on souhaite partir le plus tard possible. Par exemple, pour un rendez-vous médical, on préfère éviter d'arriver trop en avance et d'attendre inutilement.
- *Chemin le plus rapide.* Ici, l'heure exacte de départ importe peu, mais on cherche à minimiser la durée totale du trajet. Ce critère est naturel lorsqu'on peut choisir librement son moment de départ dans la journée, tout en souhaitant passer le moins de temps possible dans les transports.
- *Chemin le plus court.* Dans ce cas, on minimise uniquement la somme des durées des arcs parcourus, sans tenir compte du temps d'attente entre deux correspondances. Ce critère peut être pertinent si l'attente n'est pas considérée comme pénalisante, par exemple lorsqu'on peut profiter du temps de correspondance pour faire autre chose.

3.1 Le plus tôt/tard

12) La fonction principale est `earliest_arrival`.

- `int count_edges(graph g);` : cette fonction parcourt les listes d'adjacence du graphe afin de compter le nombre total d'arcs. Elle permet de connaître la taille nécessaire pour allouer un tableau contenant tous les arcs du graphe.

```
int count_edges(graph g){
  int count = 0;
  for (int u = 0; u < g.n; u++){
    list L = g.adj[u];
    while (!isEmpty(L)){
      count++;
      L = next(L);
    }
  }
  return count;
}
```

- `void fill_edges(graph g, arc edges[]);` : cette fonction parcourt de nouveau les listes d'adjacence et copie chaque arc dans un tableau. On obtient ainsi une représentation linéaire de tous les arcs du graphe, plus adaptée à un parcours global.

```
void fill_edges(graph g, arc edges[]){
  int k = 0;
  for (int u = 0; u < g.n; u++){
    list L = g.adj[u];
    while (!isEmpty(L)){
      edges[k] = value(L);
      k++;
      L = next(L);
    }
  }
}
```

- `int compare_edges(const void *a, const void *b);` : cette fonction utilise une fonction de tri pour ordonner les arcs selon leur temps de départ. Elle permet de trier le tableau d'arcs par ordre croissant des temps de départ.

```
int compare_edges(const void *a, const void *b){
  arc e1 = *(const arc *)a;
  arc e2 = *(const arc *)b;
  int t1 = 60 * e1.td.heure + e1.td.minute;
  int t2 = 60 * e2.td.heure + e2.td.minute;
  if (t1 < t2) return -1;
  if (t1 > t2) return 1;
  return 0;
}
```

- `int first_edge_after(arc edges[], int m, temps td);` : cette fonction détermine l'indice du premier arc dont le temps de départ est supérieur ou égal à un temps donné. Elle peut être implémentée efficacement par une recherche dichotomique dans le tableau trié.

```
int first_edge_after(arc edges[], int m, temps td){
  int target = 60 * td.heure + td.minute;
  int left = 0;
  int right = m;
  while (left < right){
    int mid = (left + right) / 2;
    int departure = 60 * edges[mid].td.heure + edges[mid].td.
    minute;
    if (departure < target){
      left = mid + 1;
    }
    else{
      right = mid;
    }
  }
  return left;
}
```

- `temps earliest_arrival(graph g, int s, int d, temps td);` : cette fonction implémente l'algorithme principal. Elle calcule le plus tôt possible l'heure d'arrivée au sommet d en partant du sommet s à l'instant td. Elle parcourt les arcs triés et met à jour progressivement les meilleurs temps d'arrivée, selon un principe de relaxation similaire à l'algorithme de Dijkstra.

```
int earliest_arrival(graph g, int s, int d, temps td){
  int m = count_edges(g);
  arc *edges = malloc(m * sizeof(arc));
  if (edges == NULL){
    fprintf(stderr, "Memory allocation error\n");
    exit(EXIT_FAILURE);
  }
  fill_edges(g, edges);
  qsort(edges, m, sizeof(arc), compare_edges);
  int *T = malloc(g.n * sizeof(int));
  if (T == NULL){
    fprintf(stderr, "Memory allocation error\n");
    free(edges);
    exit(EXIT_FAILURE);
  }
  int infinity = 24 * 60 + 1;
  for (int u = 0; u < g.n; u++){
    T[u] = infinity;
  }
  int t0 = 60 * td.heure + td.minute;
  T[s] = t0;
  int i0 = first_edge_after(edges, m, td);
  for (int i = i0; i < m; i++){
    arc e = edges[i];
    int departure = 60 * e.td.heure + e.td.minute;
    int arrival = 60 * e.ta.heure + e.ta.minute;
    if (departure >= T[e.u] && arrival < T[e.v]){
      T[e.v] = arrival;
    }
  }
  int result = T[d];
  free(T); free(edges);
  return result;
}
```

L'algorithme proposé présente de fortes similitudes avec l'algorithme de Dijkstra. En effet, on maintient pour chaque sommet un tableau des meilleurs temps d'arrivée connus, et on applique un principe de relaxation des arcs : un arc (u, v) est utilisé pour améliorer le temps d'arrivée en v si l'on peut le prendre après être arrivé en u, et si cela permet d'arriver plus tôt en v. La différence fondamentale réside dans l'absence de file de priorité. Dans Dijkstra, celle-ci permet de traiter les sommets dans l'ordre croissant des distances. Ici, cet ordre est imposé naturellement par le tri préalable des arcs selon leur temps de départ, ce qui rend inutile l'utilisation d'une structure de priorité.

💬 Algorithme 1

On peut d'abord discuter de l'utilité des fonctions auxiliaires pour l'implémentation de l'algorithme 1. Si besoin, on peut évoquer l'existence de la fonction `qsort`. La fonction `earliest_arrival` implémente l'algorithme 1. La fonction `earliest_arrival_path` (fournie dans le fichier `main.c`) fait exactement la même chose que cette fonction mais permet d'afficher les arcs empruntés.

```
1 $ ./transports
2 Departure time from 108: 10:00
3 Path:
4  (108 -> 109, 10:15, 10:17) (109 -> 110, 10:17, 10:18) (110
      -> 111, 10:18, 10:20) (111 -> 112, 10:20, 10:21) (112 ->
      96, 10:21, 10:22) (96 -> 97, 10:30, 10:32) (97 -> 221,
      10:32, 10:33) (221 -> 222, 10:33, 10:34) (222 -> 223,
      10:34, 10:35) (223 -> 224, 10:35, 10:36) (224 -> 230,
      10:42, 10:43) (230 -> 231, 10:43, 10:44)
5  Arrival time at 231: 10:44
```

13) Les arcs dont le temps de départ est strictement supérieur au meilleur temps d'arrivée déjà trouvé pour la destination ne peuvent pas améliorer la solution, car ils conduisent nécessairement à des arrivées plus tardives. On peut donc arrêter le parcours à ce moment-là. C'est exactement comme en Dijkstra : une fois qu'on a trouvé une bonne solution, on évite d'explorer des chemins forcément moins bons.

On peut interrompre la boucle dès que l'on rencontre un arc dont le temps de départ est strictement supérieur au meilleur temps d'arrivée déjà trouvé pour la destination. En effet, les arcs étant parcourus dans l'ordre croissant des temps de départ, tous les arcs suivants partiront encore plus tard et ne pourront donc pas améliorer la solution courante. Dans la boucle principale, il faut ajouter le code suivant dans la boucle `for`, à la ligne 27.

```
1 if (departure > T[d]) break;
```

💬 Algorithme 1

Discussion sur le sens de cette condition supplémentaire.

14) On passe d'une propagation vers l'avant dans le temps, du départ vers l'arrivée, à une propagation vers l'arrière, de l'arrivée vers le départ. On adapte ainsi l'algorithme en raisonnant à l'envers : au lieu de propager les temps d'arrivée vers l'avant, on propage les temps de départ vers l'arrière à partir de la destination. On utilise les mêmes arcs, mais en inversant les conditions de mise à jour et en ajustant la manière dont les valeurs sont propagées.

```
int latest_departure(graph g, int s, int d, temps ta){
  int m = count_edges(g);
  arc *edges = malloc(m * sizeof(arc));
  if (edges == NULL){
    fprintf(stderr, "Memory allocation error\n");
    exit(EXIT_FAILURE);
  }
  fill_edges(g, edges);
  qsort(edges, m, sizeof(arc), compare_edges);
  int *T = malloc(g.n * sizeof(int));
  if (T == NULL){
    fprintf(stderr, "Memory allocation error\n");
    free(edges);
    exit(EXIT_FAILURE);
  }
  // initialize
  for (int i = 0; i < g.n; i++){
    T[i] = -1; // -infinity
  }
  int t_dest = 60 * ta.heure + ta.minute;
  T[d] = t_dest;
  // backward traversal
  for (int i = m - 1; i >= 0; i--){
    arc e = edges[i];
    int departure = 60 * e.td.heure + e.td.minute;
    int arrival = 60 * e.ta.heure + e.ta.minute;
    if (arrival <= T[e.v] && departure > T[e.u]){
      T[e.u] = departure;
    }
  }
  int result = T[s];
  free(T);
  free(edges);
  return result;
}
```

💬 Algorithme - départ le plus tard

On peut discuter sur les modifications à apporter à l'algorithme 1. La fonction `latest_departure` implémente ce qui est demandé. La fonction `latest_departure_path` (fournie dans le fichier `main.c`) fait exactement la même chose que cette fonction mais permet d'afficher les arcs empruntés.

```
$ ./transports
Arrival time at 231 from 108: 11:00
Path:
 (108 -> 109, 10:15, 10:17) (109 -> 110, 10:17, 10:18) (110
    -> 111, 10:18, 10:20) (111 -> 112, 10:20, 10:21) (112 ->
    96, 10:21, 10:22) (96 -> 97, 10:30, 10:32) (97 -> 221,
    10:32, 10:33) (221 -> 222, 10:33, 10:34) (222 -> 223,
    10:34, 10:35) (223 -> 224, 10:35, 10:36) (224 -> 230,
    10:42, 10:43) (230 -> 231, 10:43, 10:44)
Latest departure time: 10:15
```

3.2 Le plus rapide

15) On déclare la structure `intervalle` dans le fichier `graph.h`.

```
typedef struct {
  temps td;
  temps ta;
} intervalle;
```

16)

```
bool meilleur(intervalle c1, intervalle c2){
  return before_or_equal(c2.td, c1.td)
         && before_or_equal(c1.ta, c2.ta);
}
```

17) Dans cette question, afin d'implémenter l'algorithme 2, on a besoin d'une liste chaînée manipulant des intervalles. Dans ce but, on crée un fichier d'entête `ilist.h` contenant la structure de donnée associée et les fonctions utiles. Le contenu de ce fichier d'en-tête est donnée ci-dessous.

```
#ifndef ILIST_H
#define ILIST_H

#include "graph.h"

typedef struct ilist_s {
  intervalle value;
  struct ilist_s *next;
} ilist_s;
typedef ilist_s* ilist;

ilist create_iList(void);
int isEmpty_iList(ilist L);
```

```
14 intervalle value_iList(ilist L);
15 ilist insertBegin_iList(ilist L, intervalle c);
16 ilist deleteBegin_iList(ilist L);
17
18 #endif
```

L'algorithme 2 peut donc être implémenté en utilisant cette nouvelle librairie.

- La fonction `add_interval` permet de maintenir, pour chaque sommet, un ensemble d'intervalles pertinents : lorsqu'un nouvel intervalle est considéré, il est ignoré s'il est inclus dans un intervalle déjà présent, et sinon il est ajouté en supprimant les intervalles qui lui sont inclus.

```
1  ilist add_interval(ilist L, intervalle c){
2    ilist cur = L;
3    ilist prev = NULL;
4    while (cur != NULL){
5      intervalle x = value_iList(cur);
6      // if an interval already in the list is better than c,
7      // do not add c
8      if (meilleur(x, c)){
9        return L;
10     }
11     // if c is better than x, remove x
12     if (meilleur(c, x)){
13       if (prev == NULL){
14         L = deleteBegin_iList(L);
15         cur = L;
16       }
17       else{
18         prev->next = deleteBegin_iList(cur);
19         cur = prev->next;
20       }
21     }
22     else{
23       prev = cur;
24       cur = cur->next;
25     }
26   }
27   return insertBegin_iList(L, c);
28 }
```

- La fonction `best_start_before` permet de sélectionner dans l'ensemble des intervalles associés à un sommet celui qui est compatible avec un arc donné (c'est-à-dire dont le temps d'arrivée est inférieur ou égal au temps de départ de l'arc) et qui permet de poursuivre le trajet dans les meilleures conditions. Elle correspond à l'étape de sélection d'un intervalle admissible dans l'algorithme.

```
1 int best_start_before(ilist L, temps limit){
2   bool found = false;
3   int best_start = -1;
4   int best_arrival = -1;
5   int limit_min = 60 * limit.heure + limit.minute;
```

```
  while (!isEmpty_iList(L)){
    intervalle c = value_iList(L);
    int arrival = 60 * c.ta.heure + c.ta.minute;
    if (arrival <= limit_min){
      if (!found || arrival > best_arrival){
        best_arrival = arrival;
        best_start = 60 * c.td.heure + c.td.minute;
        found = true;
      }
    }
    L = L->next;
  }
  return best_start;
}
```

- La fonction `fastest_path` implémente l'algorithme 2.

```
int fastest_path(graph g, int s, int d, intervalle c){
  int m = count_edges(g);
  arc *edges = malloc(m * sizeof(arc));
  if (edges == NULL){
    fprintf(stderr, "Memory allocation error\n");
    exit(EXIT_FAILURE);
  }
  fill_edges(g, edges);
  qsort(edges, m, sizeof(arc), compare_edges);
  int *T = malloc(g.n * sizeof(int));
  ilist *L = malloc(g.n * sizeof(ilist));
  if (T == NULL || L == NULL){
    fprintf(stderr, "Memory allocation error\n");
    free(edges);
    exit(EXIT_FAILURE);
  }
  int infinity = 24 * 60 + 1;
  //int td = 60 * c.td.heure + c.td.minute;
  int ta = 60 * c.ta.heure + c.ta.minute;
  for (int u = 0; u < g.n; u++){
    T[u] = infinity;
    L[u] = create_iList();
  }
  T[s] = 0;
  int i0 = first_edge_after(edges, m, c.td);
  for (int i = i0; i < m; i++){
    arc e = edges[i];
    int arrival = 60 * e.ta.heure + e.ta.minute;
    if (arrival > ta){
      continue;
    }
    if (e.u == s){
      intervalle start_interval = {e.td, e.td};
      L[s] = add_interval(L[s], start_interval);
    }
    int start = best_start_before(L[e.u], e.td);
```

```
37      if (start != -1){
38        intervalle new_interval = {
39          (temps){start / 60, start % 60},
40          e.ta
41        };
42        L[e.v] = add_interval(L[e.v], new_interval);
43        if (arrival - start < T[e.v]){
44          T[e.v] = arrival - start;
45        }
46      }
47    }
48    int result = (T[d] == infinity) ? -1 : T[d];
49    free(T);
50    free(L);
51    free(edges);
52    return result;
53  }
```

💬 Algorithme 2

On peut d'abord discuter de l'utilité des fonctions auxiliaires pour l'implémentation de l'algorithme 2. On peut discuter également de la librairie `ilist.h`. La fonction `fastest_path` implémente l'algorithme 2.

```
$ ./transports
Fastest path from 108 to 231 on [09:00,10:00]: 25 minutes
```

4 Temps de marche

18) Afin de prendre en compte des déplacements non contraints, tels que la marche, on enrichit le graphe de transport en ajoutant des arcs entre les sommets géographiquement proches. Pour cela, on utilise les coordonnées des stations fournies dans le fichier `stops.txt` et la fonction `distance`, qui permet de calculer la distance entre deux points géographiques.

Pour chaque paire de sommets u et v, on calcule leur distance et, si celle-ci est inférieure à 100 mètres (soit 0,1 km), on ajoute un arc entre u et v. Cet arc représente un déplacement à pied. Contrairement aux arcs issus des transports contraints, ces arcs de marche sont non contraints : dès qu'un sommet est atteint, il est possible de partir immédiatement vers un sommet voisin.

Dans l'algorithme de la question 12, cela se traduit par une propagation supplémentaire des temps d'arrivée : lorsqu'un sommet u est atteint à un instant $T[u]$, on peut également mettre à jour les sommets v voisins accessibles à pied, en ajoutant un temps de parcours correspondant à la distance. On obtient ainsi un graphe enrichi, permettant de modéliser les correspondances entre stations proches, comme dans le cas de grandes stations multi-modales.

On choisit d'ajouter un booléen `walk` dans la structure `arc` dans le fichier `list.h`. Par convention on prend `td = ta = (temps){0,0}` et on fixe la durée de marche à 2 minutes. Il faut adapter les fonctions du fichier `graph.h`, `graph.c` ainsi que la fonction `earliest_arrival` et `read_graph`. Il faut également déclarer la fonction `distance` dans `main.c`. Voir les fichiers associés pour les implémentations, plus précisément la fonction `earliest_arrival_walk`.

💬 Temps de marche

On peut d'abord discuter de comment on peut réaliser l'implémentation. La fonction

```
$ ./transports
Arrival without walking : 644
Arrival with walking :
Chemin trouvé de 108 vers 231 :
108
  -- transport --> 109  [10:15 -> 10:17]
  -- transport --> 110  [10:17 -> 10:18]
  -- transport --> 111  [10:18 -> 10:20]
  -- transport --> 112  [10:20 -> 10:21]
  -- transport --> 96  [10:21 -> 10:22]
  -- transport --> 97  [10:30 -> 10:32]
  -- transport --> 221  [10:32 -> 10:33]
  -- transport --> 222  [10:33 -> 10:34]
  -- walk (2 min) --> 231  [10:34 -> 10:36]
Arrivée : 10:36
636
```

19) La question fait référence au graphe enrichi construit à la question 18, dans lequel des arcs de marche ont été ajoutés entre les stations proches.

Les plus courts chemins de type « arrivée le plus tôt » sont en général plus courts, ou égaux, dans le graphe enrichi construit à la question 18. En effet, ce graphe contient tous les arcs du graphe initial, auxquels s'ajoutent des arcs de marche entre stations proches. L'ensemble des chemins possibles est donc plus riche, ce qui ne peut qu'améliorer, ou au pire conserver : si les arcs de marche n'apportent pas de meilleur chemin, le résultat reste inchangé.