TIPE - La ville

Valentin FOULON

2021-2022

Table des matières

1 Introduction

1.1 Mots-clés

Français	**Anglais**
Problème de plus court chemin	Shortest path problem
Pavage triangulaire / hexagonal	Triangular / hexagonal tiling
Apprentissage automatique	Machine learning
Migration pendulaire (déplacement maison - travail)	Commuting
Optimisation	Optimization

1.2 Problématique

Comment créer une infrastructure de transports publics dans une ville afin de minimiser l'attente tout en respectant certaines conditions ?

1.3 Résumé

L'objet de ce TIPE est de partir d'une ville dans laquelle il n'y a aucune ligne de transport public, et à partir des points d'affluence de cette ville, les créer de la façon idéale, c'est-à-dire en permettant de transporter le plus de passagers possible, le plus rapidement possible, tout en respectant des conditions de distance d'accès au transport le plus proche, de fiabilité/sécurité, etc. Ici, l'exemple utilisé sera celui du métro, qui ne possède pas de contrainte de direction car il faut créer entièrement sa trajectoire. De plus, il est idéal de prévoir un

1.4 Motivation

Venant de la banlieue parisienne éloignée, j'ai remarqué que l'accès à Paris est difficile par les transports en commun (RER). En effet, les deux lignes principales pour rejoindre la capitale sont souvent problématiques, il y a toujours un incident, que ce soit des travaux sur les lignes en plein milieu de l'été, des bagages abandonnés, etc. J'ai donc voulu chercher des solutions à ce problème.

1.5 Lien avec la ville

Les villes tendent à s'agrandir avec le temps, ce qui nécessite des moyens de transport autre que la marche. Mais de plus en plus de villes réduisent la circulation des voitures, en réduisant leur vitesse (par exemple à Paris), en supprimant des voies, ou encore en créant des espaces entièrement piétons (c'est le cas de Dijon). Il faut donc pour cela proposer des moyens de déplacement alternatifs. Malheureusement, toutes les villes ne sont pas égales pour permettre à ses habitants de se déplacer. En particulier, les banlieues sont souvent laissées de côté et les temps de déplacement sont très élevés. Ce TIPE s'inscrit donc dans le thème de la ville.

2 Sous-problèmes

2.1 Distance entre les stations

La distance entre deux stations joue énormément dans le temps de déplacement. Il faut d'un côté rapprocher les stations afin que les passagers n'aient pas à marcher trop longtemps ni à faire de grandes distances pour se déplacer. Mais il faut également garder une distance suffisante entre deux stations. En effet, un métro doit faire des arrêts pour laisser monter et descendre et monter des passagers. Multiplier les arrêts implique de s'arrêter plus souvent, mais aussi avoir une vitesse moyenne inférieure, et enfin d'avoir moins de rames sur une même ligne.

Prenons deux lignes (1) et (2). La ligne (1) possède deux stations aux extrémités et la ligne (2) possède une station au milieu en plus. Sur chacune des deux lignes, cinq trains de 1000 passagers sont placés. En supposant que les stations ont une affluence infinie, que la vitesse des trains sur une ligne est constante et que les trains de la ligne (1) déposent 1000 passagers à chaque station et ceux de la ligne (2) déposent 500 passagers à chaque station, on obtient le résultat suivant :

pour que chaque ligne transporte le même nombre de passagers en un temps donné, il faut donc que les trains de la ligne (1) mettent le même temps que ceux de la ligne (2) pour aller d'une extrémité à l'autre . Cependant, dans des conditions réelles, les trains doivent accélérer et ralentir autour des stations. Si on rallonge la durée de trajet sur la ligne (2) de 10% pour prendre en compte les ralentissements, alors la ligne (1) transporte 12,5% de voyageurs de plus que la ligne (2).

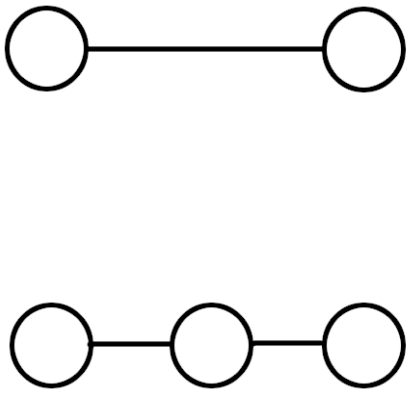
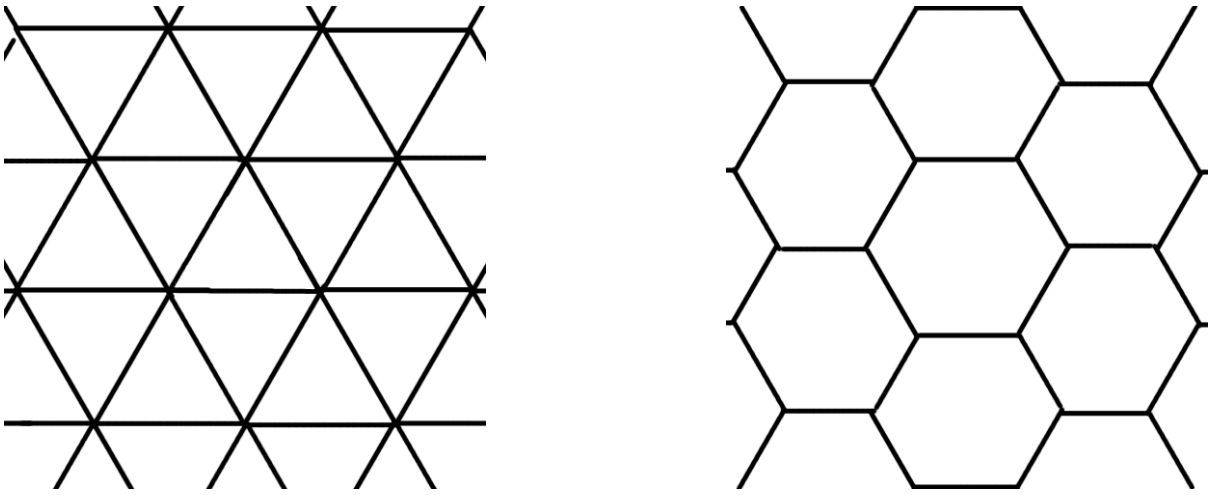


FIGURE 1 – Ligne (1) en bas et (2) en haut

Pour limiter que les passagers marchent trop, l'idéal est donc d'avoir plusieurs stations de lignes différentes autour d'une même zone. Pour s'assurer de ne jamais dépasser cette distance, il est possible de créer les stations sur un maillage triangulaire ou hexagonal, ce qui permet d'être sûr que la distance maximale d pour rejoindre une station quelconque à partir de toute position est majorée. Dans les deux cas : soit un triangle équilatéral ou un hexagone de côté a. On a :

$$d \leq \frac{1}{2}\sqrt{a^2 - \left(\frac{a}{2}\right)^2}$$



(a) Pavage triangulaire (b) Pavage hexagonal

FIGURE 2 – Pavages possibles

Hypothèse : laisser une distance maximale de 500m pour aller à n'importe quelle station
A vérifier

2.2 Espacement entre les trains

L'espacement entre les rames est aussi un facteur à prendre en compte. Il faut réussir à s'adapter à l'affluence des lignes, c'est-à-dire savoir combien de voyageurs sont attendus à un certain moment. Les heures de départ et de retour du travail nécessitent par exemple d'avoir plus de rames en circulation, de même pour les vacances, etc. Mais le nombre de rames sur

une ligne est limité. En effet, il faut pouvoir arrêter les autres rames à temps pour éviter des accidents, et éviter de faire attendre un train juste avant une station pour attendre que le précédent parte. De plus, avoir plus de rames sur une ligne augmente le risque d'incidents. Or pour éviter d'avoir une attente importante après l'incident, en général toute la ligne est mise en pause pour conserver un écart égal entre tous les trains.

A exploîter

2.3 Information aux voyageurs

Un autre élément très important est l'information aux voyageurs. Aujourd'hui, tout le monde possède un téléphone portable ainsi qu'un forfait de téléphone avec un accès à Internet. Grâce aux téléphones, il est possible d'améliorer encore les trajets des voyageurs. Par exemple, en centralisant tout sur une application mobile utilisant du machine learning, il serait possible de réduire considérablement les temps d'attentes :

— En permettant aux usagers de prévoir leur titre de transport, alors on peut réduire les files d'attente aux guichets, faire payer les usagers exactement ce qu'ils doivent payer en sachant leur point de départ et leur destination

— En leur proposant un itinéraire le plus adapté en fonction des prochains départs, du temps de trajet total (avec un algorithme de plus court chemin, possiblement de Dijkstra), du temps de correspondance (si il y en a une), et de l'affluence à l'heure du trajet

— Les deux points précédents réunis ont un autre avantage : limiter l'attente pendant les heures de pointe. En effet, si l'application sait quel sera la destination de l'usager, elle peut aussi prendre en compte l'affluence, et suggérer un itinéraire alternatif qui sera peut-être légèrement plus long, mais permettra aux usagers qui prennent le même chemin tous les jours puissent avoir un trajet de même durée que d'habitude

Algorithmes de plus court chemin à envisager :

— Dijkstra (O(|E| + |V|log|V|)) : $d_{ij} = min_k(d_{ij}, d_{ik} + w(k, j))$

— Bellman-Ford (O(|V||E|)) : pour u et v voisins $d[v] = min(d[v], d[u] + w(u, v))$

— Floyd-Warshall (O(|V|3)) : $\delta(i, j) = min_k(\delta(i, j), \delta(i, k) + \delta(k, j))$

A exploîter

2.4 Agrandissement

Lorsqu'une ville vient à s'agrandir dans une direction, il faut relier cette nouvelle partie au reste de la ville. Pour cela, deux solutions majeures sont envisageables :

— Agrandir une ligne existante

— Créer une nouvelle ligne

A exploîter

2.5 Autres idées à explorer

— L'impact des virages d'angle inférieur à 90° sur la vitesse d'un train

— Le potentiel gain de temps des trains autonomes

— L'impact de la longueur des rames

— Possibilité de "synchroniser" l'arrivée des rames dans une station pour les correspondances (soit en les faisant arriver en même temps soit en les alternant)

3 Sources

A compléter

4 Code

4.1 metro.c (incomplet)

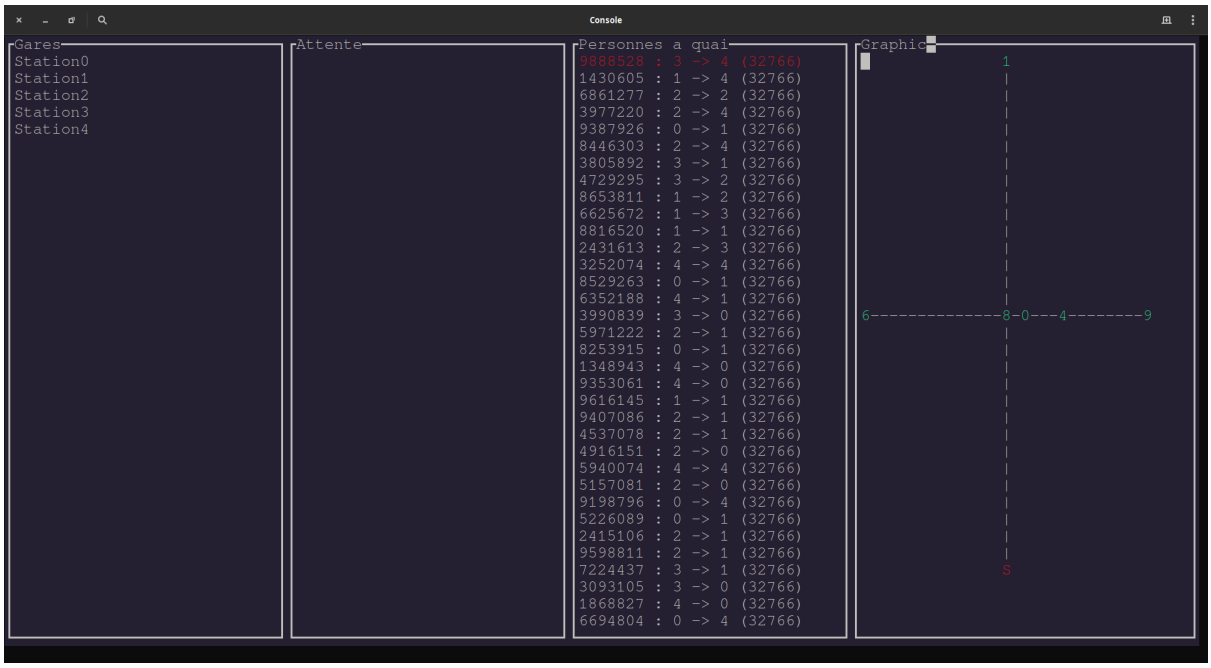


FIGURE 3 – L'interface graphique du programme (pas encore finie)

```c
#define _GNU_SOURCE
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <time.h>
#include <ncurses.h>
#include <string.h>
#include <sys/ioctl.h>
#include <locale.h>
#include <pthread.h>
#include "vector.h"

#define SEATS 1000

#define ROME 0
#define LA_FOURCHE 1
#define BLANCHE 2
#define PLACE_DE_CLICHY 3
#define LIEGE 4

#define DUREE_TRAJET 30
#define ARRET_STATION 15

#define DEFAULT 0
#define RED 1
#define GREEN 2
#define BLUE 3
#define YELLOW 4
#define CYAN 5
#define WHITE 6

int grid[31][31] = {-1};
```

```
//int ligne1[3] = {0, 1, 4};
//int ligne2[3] = {2, 1, 3};
bool verbose = false;
int cpos[2] = {1, 1};

int station_count = 5;

struct train {
  bool backwards;
  int ligne;
  int destination;
  int passagers;
  int station_actuelle;
  bool deplacement;
  int prochain_depart;
  int color;
};

typedef struct train train;

struct personne {
  int station_actuelle;
  int destination;
  int attente;
  int nom;
};
typedef struct personne personne;

struct gare {
  int personnes_a_quai;
  char* nom;
  int affluence;
  int pos[2];
};
typedef struct gare gare;
Vector(gare) stations;


struct ligne {
  int nbr;
  int* stations;
  int* interval_st;
  int len;
};
typedef struct ligne ligne;

ligne ligne1;
ligne ligne2;
ligne* lignes;

struct winsize w;

personne personnes[1000];
train trains[10];
gare gares[5];
int rame[1000] = {0};
int lcount;
int p_offset[3] = {0, 0, 0};
int w_offset = 2;
int ccount;

WINDOW* w_gares;
WINDOW* attente;
```

```
96  WINDOW* w_personnes;
97  WINDOW* graphic;
98
99  pthread_t ti;
100 bool editmode = false;
101
102
103 void print_grid();
104 int indexOf(int n, int* array, int len);
105 void print_personnes();
106 void print_gare();
107 void print_stations();
108 void print_status();
109 void* updatetime();
110 void getinput();
111 void define_stations();
112 void creer_station_sur_lignes(ligne* lines, int len, int index);
113 int get_train(int x, int y);
114
115 void print_personnes() {
116   wclear(w_personnes);
117   wresize(w_personnes, lcount, 30);
118   box(w_personnes, 0, 0);
119   mvwprintw(w_personnes, 0, 1, "Personnes a quai");
120   wrefresh(w_personnes);
121   for (int i = 0; i < lcount - 2; i++) {
122     if (w_offset == 2 && i == 0) {
123       wattron(w_personnes, COLOR_PAIR(RED));
124     }
125     mvwprintw(w_personnes, i + 1, 1, "%d : %d -> %d (%d)", personnes[i +
      p_offset[2]].nom, personnes[i + p_offset[2]].station_actuelle, personnes[i +
       p_offset[2]].destination, calculate_directions(personnes[i + p_offset[2]]))
      ;
126     if (w_offset == 2 && i == 0) {
127       wattroff(w_personnes, COLOR_PAIR(RED));
128     }
129   }
130   wrefresh(w_personnes);
131 }
132
133 void print_gare() {
134   // for (int i = 0; i < lcount - 2; i++) {
135   //   if (w_offset == 1 && i == p_offset[1]) {
136   //     wattron(attente, COLOR_PAIR(RED));
137   //   }
138   //   mvwprintw(w_personnes, i + 1, 1, "%d -> %d", personnes[i + p_offset[2]].
      nom, personnes[i + p_offset[2]].attente);
139   //   if (w_offset == 1 && i == p_offset[1]) {
140   //     wattroff(attente, COLOR_PAIR(RED));
141   //   }
142   // }
143   // wrefresh(w_personnes);
144 }
145
146 void print_stations() {
147   for (int i = 0; i < len(stations); i++) {
148     if (w_offset == 0 && i == p_offset[0]) {
149       wattron(w_gares, COLOR_PAIR(RED));
150     }
151     mvwprintw(w_gares, i + 1, 1, "%s", stations.tab[i].nom);
152     if (w_offset == 0 && i == p_offset[0]) {
153       wattroff(w_gares, COLOR_PAIR(RED));
154     }
```

```
155   }
156   wrefresh(w_gares);
157 }
158
159 void print_status() {
160   system("clear");
161   for (int i = 0; i < 10; i++) {
162     printf("\033[32mTrain #%d\033[0m\n", i);
163     printf("Destination : %s\n", stations.tab[trains[i].destination].nom);
164     printf("Ligne : %d\n", trains[i].ligne);
165     printf("Passagers : %d\n", trains[i].passagers);
166     printf("Station actuelle : %s\n", stations.tab[trains[i].station_actuelle].
      nom);
167     printf("Deplacement : %s\n", trains[i].deplacement ? "TRUE" : "FALSE");
168     printf("Prochain depart : %d s\n", trains[i].prochain_depart);
169     float percentage_done = 1.0 - (float) ((trains[i].prochain_depart >
      ARRET_STATION ? trains[i].prochain_depart - ARRET_STATION : 0)) /
      ARRET_STATION;
170     printf("Pourcentage trajet %lf\n", percentage_done);
171     printf("Backwards : %s\n", trains[i].backwards ? "TRUE" : "FALSE");
172   }
173 }
174
175 void* updatetime() {
176   while (true) {
177     ioctl(STDOUT_FILENO, TIOCGWINSZ, &w);
178     lcount = w.ws_row;
179     ccount = w.ws_col;
180     wresize(w_gares, lcount, 30);
181     wresize(attente, lcount, 30);
182     wresize(graphic, lcount, ccount - 90);
183     box(w_gares, 0, 0);
184     box(attente, 0, 0);
185     box(graphic, 0, 0);
186     mvwprintw(w_gares, 0, 1, "Gares");
187     mvwprintw(attente, 0, 1, "Attente");
188     mvwprintw(graphic, 0, 1, "Graphic");
189     refresh();
190     wrefresh(w_gares);
191     wrefresh(attente);
192     wrefresh(graphic);
193     usleep(50000);
194     for (int i = 0; i < 1000; i++) {
195       personnes[i].attente++;
196     }
197     for (int i = 0; i < 10; i++) {
198       if (trains[i].prochain_depart != 0) {
199         trains[i].prochain_depart--;
200         if (trains[i].prochain_depart <= 15) {
201           trains[i].deplacement = false;
202         } else {
203           trains[i].deplacement = true;
204         }
205       } else {
206         trains[i].prochain_depart = DUREE_TRAJET;
207         if (trains[i].backwards) {
208           int nxt = indexOf(trains[i].station_actuelle, lignes[trains[i].ligne
    ].stations, 3);
209           printf("%d\n", nxt);
210           trains[i].station_actuelle = lignes[trains[i].ligne].stations[nxt -
    1];
211           if (trains[i].station_actuelle == trains[i].destination && !trains[i
    ].deplacement) {
```

```
212             trains[i].destination = lignes[trains[i].ligne].stations[2 -
    indexOf(trains[i].destination, lignes[trains[i].ligne].stations, 3)];;
213             trains[i].backwards = !(trains[i].backwards);
214           }
215         } else {
216           int nxt = indexOf(trains[i].station_actuelle, lignes[trains[i].ligne
    ].stations, 3);
217           printf("%d\n", nxt);
218           trains[i].station_actuelle = lignes[trains[i].ligne].stations[nxt +
    1];
219           if (trains[i].station_actuelle == trains[i].destination && !trains[i
    ].deplacement) {
220             trains[i].destination = lignes[trains[i].ligne].stations[2 -
    indexOf(trains[i].destination, lignes[trains[i].ligne].stations, 3)];;
221             trains[i].backwards = !(trains[i].backwards);
222           }
223         }
224       }
225     }
226     for (int i = 0; i < len(stations); i++) {
227       int h = random() % 100;
228       if (h <= stations.tab[i].affluence) {
229         stations.tab[i].personnes_a_quai++;
230       }
231     }
232     print_personnes();
233     print_grid();
234     if (verbose) {
235       print_status();
236     }
237   }
238 }
239
240 void getinput() {
241   cbreak();
242   noecho();
243   int c;
244   while (true) {
245     c = getch();
246     switch(c) {
247       case 'A':
248       case KEY_UP:
249         if (p_offset[w_offset] != 0) p_offset[w_offset]--;
250         break;
251       case 'B':
252       case KEY_DOWN:
253         switch (w_offset) {
254           case 2:
255             if (p_offset[w_offset] != 1000 - lcount + 2) p_offset[w_offset]++;
256             break;
257           case 0:
258             if (p_offset[w_offset] != 4) p_offset[w_offset]++;
259             break;
260         }
261         break;
262       case 'D':
263       case KEY_LEFT:
264         if (w_offset != 0) w_offset--;
265         break;
266       case 'C':
267       case KEY_RIGHT:
268         if (w_offset != 3) w_offset++;
269         break;
```

```
270       case 'w':
271         if (cpos[0] > 1) {
272           cpos[0]--;
273           print_grid();
274         }
275         break;
276       case 'a':
277         if (cpos[1] > 1) {
278           cpos[1]--;
279           print_grid();
280         }
281         break;
282       case 's':
283         if (cpos[0] < 31) {
284           cpos[0]++;
285           print_grid();
286         }
287         break;
288       case 'd':
289         if (cpos[1] < 31) {
290           cpos[1]++;
291           print_grid();
292         }
293         break;
294       case 'n':
295         if (w_offset == 3) {
296           gare n = {
297             .personnes_a_quai = 0,
298             .affluence = 5,
299             .pos = {cpos[0], cpos[1]},
300             .nom = "Nouvelle gare",
301           };
302           realloc(stations.tab, (len(stations) + 1) * sizeof(gare));
303           stations.tab[len(stations)] = n;
304           stations.len++;
305           //push(stations, (gare) n);
306         }
307       case ' ':
308         editmode = !editmode;
309         break;
310       case '/':
311         nocbreak();
312         // Do sth
313         break;
314       case 'q':
315         pthread_cancel(ti);
316         system("reset");
317         exit(0);
318         break;
319       default:
320         break;
321     }
322     cbreak();
323     print_personnes();
324     print_stations();
325   }
326 }
327
328 void define_stations() {
329   for (int i = 0; i < 5; i++) {
330     asprintf(&gares[i].nom, "%s", stations.tab[i].nom);
331   }
332 }
```

```
333
334 void creer_station_sur_lignes(ligne* lines, int len, int index) {
335   for (int i = 0; i < len; i++) {
336     lines[i].len++;
337     lines[i].stations = realloc(lines[i].stations, lines[i].len * sizeof(int));
338     lines[i].stations[index] = station_count;
339   }
340   station_count++;
341 }
342
343 int get_train(int x, int y) {
344   int train_number = -1;
345   for (int k=0; k < 10; k++) {
346     int train_x = 0;
347     int train_y = 0;
348     float percentage_done = 1.0 - (float) ((trains[k].prochain_depart >
      ARRET_STATION ? trains[k].prochain_depart - ARRET_STATION : 0)) /
      ARRET_STATION;
349     switch (trains[k].ligne) {
350       case 0:
351         train_x = 15;
352         train_y = indexOf(trains[k].station_actuelle, lignes[trains[k].ligne].
      stations, 3);
353         if (trains[k].backwards) {
354           if (x == train_x && y == train_y * 15 - (int) (percentage_done * 15))
       {
355             train_number = k;
356           }
357         } else {
358           if (x == train_x && y == train_y * 15 + (int) (percentage_done * 15))
       {
359             train_number = k;
360           }
361         }
362         break;
363       case 1:
364         train_y = 15;
365         train_x = indexOf(trains[k].station_actuelle, lignes[trains[k].ligne].
      stations, 3);
366         if (trains[k].backwards) {
367           if (x == train_x * 15 - (int) (percentage_done * 15) && y == train_y)
       {
368             train_number = k;
369           }
370         } else {
371           if (x == train_x * 15 + (int) (percentage_done * 15) && y == train_y)
       {
372             train_number = k;
373           }
374         }
375         break;
376       default:
377         break;
378     }
379   }
380   return train_number;
381 }
382
383 int matching(int** tab1, int len1, int** tab2, int len2) {
384   for (int i = 0; i < len1; i++) {
385     for (int j = 0; j < 3; j++) {
386       for (int k = 0; k < len2; k++) {
387         for (int l = 0; l < 3; l++) {
```

```
          if (tab1[i][j] == tab2[k][l]) return 1;
        }
      }
    }
  }
  return -1;
}

int calculate_directions_aux(int depart, int arrivee) {
  int** ligne_s_depart;
  int found = 0;
  for (int i = 0; i < 2; i++) {
    if (indexOf(depart, lignes[i].stations, 3) != -1) {
      found++;
      ligne_s_depart = malloc(found * sizeof(int*));
      ligne_s_depart[found - 1] = lignes[i].stations;
    }
  }
  int** ligne_s_arrivee;
  int found2 = 0;
  for (int i = 0; i < 2; i++) {
    if (indexOf(arrivee, lignes[i].stations, 3) != -1) {
      found++;
      ligne_s_arrivee = malloc(found2 * sizeof(int*));
      ligne_s_arrivee[found2 - 1] = lignes[i].stations;
    }
  }
  int lesgo = matching(ligne_s_depart, found, ligne_s_arrivee, found2);
  if (lesgo != -1) {
    return lesgo;
  } else {
    for (int i = 0; i < found2; i++) {
      for (int k = 0; k < lignes[i].len; k++) {
        return calculate_directions_aux(depart, lignes[i].stations[k]);
      }
    }
  }
}

int calculate_directions(personne voyageur) {
  int depart = voyageur.station_actuelle;
  int arrivee = voyageur.destination;
  return calculate_directions_aux(depart, arrivee);
}

int indexOf(int n, int* array, int len) {
  for (int i = 0; i < len; i++) {
    if (array[i] == n) return i;
  }
  return -1;
}

void print_grid() {
  for (int i = 0; i < 31; i++) {
    for (int j = 0; j < 31; j++) {
      int train = get_train(i, j);
      //int train = -1;
      if (train == -1) {
        if (grid[i][j] == -1) {
          if (i == 15) {
            mvwprintw(graphic, i + 1, j + 1, "%c", '-');
          } else if (j == 15) {
            mvwprintw(graphic, i + 1, j + 1, "%c", '|');
```

```
451           } else {
452             mvwprintw(graphic, i + 1, j + 1, "%c", ' ');
453           }
454         } else {
455           wattron(graphic, COLOR_PAIR(RED));
456           for (int k = 0; k < len(stations); k++) {
457             if (stations.tab[k].pos[0] == i && stations.tab[k].pos[1] == j) {
458               mvwprintw(graphic, i + 1, j + 1, "%c", stations.tab[k].nom[0]);
459             }
460           }
461           //mvwprintw(graphic, i + 1, j + 1, "%c", stations.tab[grid[i][j]].nom
    [0]);
462           wattroff(graphic, COLOR_PAIR(RED));
463         }
464       } else {
465         wattron(graphic, COLOR_PAIR(GREEN));
466         mvwprintw(graphic, i + 1, j + 1, "%d", train);
467         wattroff(graphic, COLOR_PAIR(GREEN));
468       }
469     }
470   }
471   wattron(graphic, COLOR_PAIR(WHITE));
472   mvwprintw(graphic, cpos[0], cpos[1], " ");
473   wattroff(graphic, COLOR_PAIR(WHITE));
474   refresh();
475   wrefresh(graphic);
476 }
477
478 int main(int argc, char* argv[]) {
479   init_vector(stations, gare, 5);
480   for (int i = 0; i < 5; i++) {
481     asprintf(&stations.tab[i].nom, "%s%d", "Station", i);
482   }
483   stations.tab[0].pos[0] = 15;
484   stations.tab[0].pos[1] = 15;
485   stations.tab[1].pos[0] = 15;
486   stations.tab[1].pos[1] = 0;
487   stations.tab[2].pos[0] = 0;
488   stations.tab[2].pos[1] = 15;
489   stations.tab[3].pos[0] = 30;
490   stations.tab[3].pos[1] = 15;
491   stations.tab[4].pos[0] = 15;
492   stations.tab[4].pos[1] = 30;
493   lignes = malloc(2 * sizeof(ligne));
494   ligne1.len = 3;
495   ligne1.nbr = 1;
496   ligne1.stations = malloc(3 * sizeof(int));
497   ligne1.stations[0] = 0;
498   ligne1.stations[1] = 1;
499   ligne1.stations[2] = 4;
500   ligne1.interval_st = malloc(2 * sizeof(int));
501   ligne1.interval_st[0] = ligne1.interval_st[1] = 15;
502   ligne2.len = 3;
503   ligne2.nbr = 2;
504   ligne2.stations = malloc(3 * sizeof(int));
505   ligne2.stations[0] = 2;
506   ligne2.stations[1] = 1;
507   ligne2.stations[2] = 3;
508   ligne2.interval_st = malloc(3 * sizeof(int));
509   ligne2.interval_st[0] = ligne2.interval_st[1] = 15;
510   lignes[0] = ligne1;
511   lignes[1] = ligne2;
512   srand(time(NULL));
```

```
513   setlocale(LC_ALL, "");
514   if (argc > 1 && strcmp(argv[1], "-v") == 0) {
515     verbose = true;
516   } else {
517     initscr();
518   }
519   start_color();
520   init_pair(RED, COLOR_RED, COLOR_BLACK);
521   init_pair(GREEN, COLOR_GREEN, COLOR_BLACK);
522   init_pair(WHITE, COLOR_BLACK, COLOR_WHITE);
523   ioctl(STDOUT_FILENO, TIOCGWINSZ, &w);
524   lcount = w.ws_row;
525   ccount = w.ws_col;
526   w_gares = newwin(lcount, 30, 0, 0);
527   attente = newwin(lcount, 30, 0, 30);
528   w_personnes = newwin(lcount, 30, 0, 60);
529   graphic = newwin(lcount, ccount - 90, 0, 90);
530   box(w_gares, 0, 0);
531   box(attente, 0, 0);
532   box(w_personnes, 0, 0);
533   box(graphic, 0, 0);
534   mvwprintw(w_gares, 0, 1, "Gares");
535   mvwprintw(attente, 0, 1, "Attente");
536   mvwprintw(w_personnes, 0, 1, "Personnes a quai");
537   mvwprintw(graphic, 0, 1, "Graphic");
538   refresh();
539   wrefresh(w_gares);
540   wrefresh(attente);
541   wrefresh(w_personnes);
542   wrefresh(graphic);
543   for (int i = 0; i < 31; i++) {
544     for (int j = 0; j < 31; j++) {
545       grid[i][j] = -1;
546     }
547   }
548   grid[0][15] = 0;
549   grid[15][15] = 1;
550   grid[15][0] = 2;
551   grid[15][30] = 3;
552   grid[30][15] = 4;
553   for (int i = 0; i < 1000; i++) {
554     personnes[i].nom = rand() % (10000000 - 1000000) + 1000000;
555     personnes[i].station_actuelle = rand() % 5;
556     personnes[i].destination = rand() % 5;
557     personnes[i].attente = rand() % 90;
558   }
559   for (int i = 0; i < 10; i++) {
560     trains[i].ligne = rand() % 2;
561     int st_dest = (int) ((rand() % 2) * 2);
562     trains[i].destination = lignes[trains[i].ligne].stations[st_dest];
563     trains[i].backwards = (st_dest == 0);
564     trains[i].passagers = 0;
565     trains[i].station_actuelle = lignes[trains[i].ligne].stations[rand() % 3];
566     trains[i].deplacement = true;
567     trains[i].prochain_depart = rand() % DUREE_TRAJET;
568   }
569   for (int i = 0; i < 5; i++) {
570     gares[i].personnes_a_quai = 0;
571     gares[i].affluence = rand() % 100;
572   }
573   define_stations();
574   print_stations();
575   print_personnes();
```

```
  print_grid();
  pthread_create(&ti, NULL, updatetime, NULL);
  getinput();
  pthread_join(ti, NULL);
  return 0;
}
```

4.2 vector.h (nécessaire pour le code précédent)

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define str(ELT) #ELT
#define fmt(ELT) _Generic((ELT), int : "%d", float : "%f", char : "%c", signed char : "%c", unsigned char : "%c", short : "%hi", unsigned short : "%hu", unsigned int : "%u", long int : "%ld", long unsigned int : "%lu", long long : "%lld", unsigned long long : "%llu", double : "%lf", long double : "%Lf")
#define type(ELT) _Generic((ELT), int : (int) 0, float : (float) 0.0, char : (char) 'a', signed char : (signed char) 'a', short : (short) 0, unsigned short : (unsigned short) 0, long : (long) 0, unsigned long : (unsigned long) 0, long long : (long long) 0, unsigned long long : (unsigned long long) 0, double : (double) 0.0, long double : (long double) 0.0)
#define Vector(TYPE) struct {TYPE* tab; int len;}
#define push(VEC, ELT) \
  ({ \
    VEC.len++; \
    realloc(VEC.tab, sizeof(VEC) + sizeof(ELT)); \
    VEC.tab[VEC.len - 1] = ELT; \
  })
#define insert(VEC, ELT, INDEX) \
  ({ \
    VEC.len++; \
    realloc(VEC.tab, VEC.len * sizeof(ELT)); \
    for (int i = VEC.len - 1; i > INDEX; i--) { \
      VEC.tab[i] = VEC.tab[i - 1]; \
    }; \
    VEC.tab[INDEX] = ELT; \
  })
#define print_vector(VEC) \
  ({ \
    printf("Vector : %s\n", str(VEC)); \
    printf("Size : %d\n", VEC.len); \
    printf("Elements : "); \
    for (int i = 0; i < VEC.len; i++) { \
      printf(fmt(nth(VEC, i)), nth(VEC, i)); \
      printf(" "); \
    } \
    printf("\n"); \
  })
#define nth(VEC, N) VEC.tab[N]
#define delete_index(VEC, INDEX) \
  ({ \
    for (int i = INDEX; i < VEC.len; i++) { \
      VEC.tab[i] = VEC.tab[i + 1]; \
    } \
    if (VEC.len > 0) { \
      VEC.len--; \
      realloc(VEC.tab, VEC.len * sizeof(VEC.tab[0])); \
    } else { \
      realloc(VEC.tab, 0); \
    } \
  })
#define indexOfVector(ELT, VEC) \
  ({ \
```

```
    int x = -1; \
    for (int i = 0; i < VEC.len; i++) { \
      if (VEC.tab[i] == ELT) { \
        x = i; \
        break; \
      } \
    } \
    x; \
  })
#define delete_element(VEC, ELT) \
  ({delete_index(VEC, index_of(VEC, ELT));})
#define set(VEC, INDEX, ELT) \
  ({ \
    if (VEC.len >= INDEX) { \
      VEC.tab[INDEX] = ELT; \
    } \
  })
#define init_vector(VEC, TYPE, LEN) \
  ({ \
    VEC.tab = malloc(LEN * sizeof(TYPE)); \
    VEC.len = LEN; \
  })
#define merge(VEC1, TYPE1, VEC2, TYPE2) \
  ({ \
    if (TYPE1 == TYPE2) { \
      Vector(TYPE1) newV; \
      newV.len = VEC1.len + VEC2.len; \
      int j = 0; \
      for (int i = 0; i < VEC1.len; i++) { \
        newV.tab[j] = VEC1.tab[i]; \
        j++; \
      } \
      for (int i = 0; i < VEC2.len; i++) { \
        newV.tab[j] = VEC2.tab[i]; \
        j++; \
      } \
      return newV; \
    } \
  })
#define iter(VEC, F) \
  ({ \
    for (int i = 0; i < VEC.len; i++) { \
      F(VEC.tab[i]); \
    } \
  })
#define exists(VEC, E) \
  ({ \
    bool found = false; \
    for (int i = 0; i < VEC.len; i++) { \
      if (VEC.tab[i == E]) found = true; \
    } \
    found; \
  })
#define len(VEC) \
  ({VEC.len;})
```