Projet Algorithme

Cabouat Charlotte
Santoro Fabrizio

Mai 2018

1 Ensemble du Projet

Le but de ce projet est de pouvoir connaître le plus court chemin selon le niveau du skieur pour aller d'un point à un autre de la station de ski de Serre Chevalier.

Pour cela nous avons utilisé l'algorithme de Dijkstra qui permet de déterminer le plus court chemin dans un graphe orienté.

2 Structure de Données

Il y a principalement deux structures de données importantes: le graphe et le tableau de l'algorithme.

Graphe

Le fichier data.dat contient toutes les informations nécessaires à la création du graphe, les sommets, ses successeurs, le coût de l'arc et enfin le type. Le graphe est donc un tableau contenant des sommets avec leurs arcs. Chaque sommets connaît sa position dans le tableau (utile pour s'y retrouver plus tard dans l'algorithme), il a aussi une liste chaîné d'arcs, qui eux mêmes contiennent la position du sommet d'arrivée et de départ dans le tableau, encore une fois pour la raison expliquée plus haut.

Tableau de Dijkstra

Ce que nous définissons comme tableau de dijkstra est le tableau utilisé lors de l'application de l'algorithme sur le graphe, il contient le coût des sommets ainsi que leurs père.

Les autres structures sont utiles à la configuration de l'algorithme, par exemple la structure de l'utilisateur sert à savoir son niveau, point de départ et d'arrivée. Une autre structure utile à mentionner est la pile utilisée pour écrire correctement le plus court chemin, si il existe.

3 Implémentation Algorithme de Dijkstra

Comme expliqué dans la discussion des structures de données utilisées dans le projet, l'implémentation se base surtout sur le tableau du graphe et le tableau de dijkstra. Le tableau de dijkstra est donc mis à jour à chaque sommet visité, le sommet avec le coût minimal est alors choisi comme prochain sommet à considérer.
Nous pouvons observer que pour voir si il existe un chemin entre les points choisi par l'utilisateur, il suffit que s'il ne reste que des sommets à temps infini, y compris le sommet d'arrivée, qu'y n'ont pas été considérés, alors le parcours n'existe pas. Le tableau du graphe est référencé par le tableau de dijkstra, plus précisément, les arcs de ses sommets, car ils sont utiles pour le bon affichage du parcours (le nom des pistes par exemple).

4 Gestion du Plan des Pistes

Nous avons créé une base de données pour pouvoir mettre en forme le graphe sous le principe suivant: node;
arc-nodeArrival-cost-diff
Les sommets peuvent être des points de départ de pistes ou de remonté mécanique, mais aussi des croisements de pistes.

La difficulté étant le niveau de la piste: Verte, Bleu,Rouge,Noir,Remonté Mécanique. Le coût est le temps pour faire le trajet d'un sommet à un autre. De base le coût est celui pour un skieur ayant un niveau expert.
Pour connaître le temps du niveau débutant on utilise un multiplicateur selon le niveau de la piste: x1.1 Verte,x1.5 Bleu,x2 Rouge,x3 Noir. Pour les remontés mécanique le temps reste le même.

Ensuite nous utilisons une fonction pour lire le fichier et ainsi implémenter les données dans l'algorithme.

5 Code

Dijkstra.c

```
1
2  #include <stdlib.h>
3  #include <stdio.h>
4
5  #include "dijk.h"
6
7  void findShortestPath(user_t * user, graph_t * graph)
8  {
9    int i, index;
10   /*
11     Alloue un tableau de tous les sommets, pour avoir les distances
```

```
12      et les p res ,
13         la fin , remonte depuis la destination jusqu'   la source
14      la source a comme p re NULL
15    */
16    /*
17      Si le noeud analis  est la destination , on s'arr te , NON!
18      l'algo doit faire tous les noeuds!
19    */
20
21    dijNode_t * tabNode = malloc(sizeof(dijNode_t) * graph->sizeTab);
22    //used for knowing which nodes have been visited
23    int * checked = calloc(sizeof(int), graph->sizeTab);
24    char ok = 0;
25
26    for(i = 0; i < graph->sizeTab; i++)
27    {
28      tabNode[i].cost = -1;
29      tabNode[i].father = NULL;
30      tabNode[i].lineUsed = NULL;
31    }
32
33    //visite le premier d'abord
34    tabNode[user->startNode].cost = 0;
35    updateNeighboors(&graph->tabNode[user->startNode], tabNode, user)
        ;
36    checked[user->startNode] = 1;
37
38    //boucle, qui simule vraiment dijkstra
39    while(!ok && getVisited(checked, graph->sizeTab) != graph->
        sizeTab)
40    {
41      index = getMin(tabNode, graph->sizeTab, checked);
42      if(index != -1)
43      {
44        updateNeighboors(&graph->tabNode[index], tabNode, user);
45        checked[index] = 1;
46      }
47      else
48        ok = 1;
49    }
50
51    //parcours  trouv
52    if(checked[user->arrivalNode])
53    {
54      //affiche le parcours
55      printPath(tabNode,graph->sizeTab, user, graph);
56    }
57    else
58      printf("parcours impossible! \n");
59
60    free(tabNode);
61    free(checked);
62  }
63
64  void printPath(dijNode_t * tabNode, int size, user_t * user,
        graph_t * graph)
65  {
```

```
66    dijNode_t * index = &tabNode[user->arrivalNode];
67    printStruct_t * pile = malloc(sizeof(printStruct_t));
68    printStruct_t * pileNext = NULL;
69    float totalTime = 0;
70
71    pile->node = index;
72    pile->next = NULL;
73
74    printf("Vous partez de %s\n", graph->tabNode[user->startNode].
        name);
75
76    index = &tabNode[index->father->index];
77
78    while(index->father != NULL)
79    {
80      pileNext = malloc(sizeof(printStruct_t));
81
82      pileNext->node = index;
83      pileNext->next = pile;
84      pile = pileNext;
85
86      index = &tabNode[index->father->index];
87    }
88
89    pileNext = pile;
90
91    while(pileNext != NULL)
92    {
93      totalTime += pileNext->node->cost;
94      if(pileNext->node->lineUsed->diff == MECHANIC)
95        printf("Tu passera par un truc mechanique");
96      else
97        printf("Tu passera par une piste");
98
99      printf(" %s en %f mins\n"
100     , pileNext->node->lineUsed->name, pileNext->node->cost);
101
102     pileNext = pileNext->next;
103   }
104
105   printf("Vous arrivez %s\n", graph->tabNode[user->arrivalNode].
        name);
106
107   printf("Temps total estime: %f\n", totalTime);
108
109   freePath(pile);
110
111 }
112
113 void freePath(printStruct_t * pile)
114 {
115   printStruct_t * index = pile;
116   printStruct_t * next = pile->next;
117
118   while (next != NULL)
119   {
120     free(index);
```

```
121     index = next;
122     next = index->next;
123   }
124 
125   free(index);
126 }
127 
128 
129 int getVisited(int * tab, int size)
130 {
131   int i, visited = 0;
132 
133   for(i = 0; i < size; i++)
134   {
135     if(tab[i])
136       visited++;
137   }
138 
139   return visited;
140 }
141 
142 void updateNeighboors(node_t * node, dijNode_t * tabNode, user_t *
      user)
143 {
144   float cost;
145   arc_t * arc = node->arcs;
146 
147   while(arc != NULL)
148   {
149     cost = getRealTime(user->level, arc);
150 
151     //printf("index: %d, costnodearr: %f , cost: %f, costSommet: %f
      \n",
152     //node->index, tabNode[arc->indexArrival].cost, cost, tabNode[
      node->index].cost);
153 
154     if(tabNode[arc->indexArrival].cost == -1
155      || tabNode[node->index].cost + cost
156      < tabNode[arc->indexArrival].cost)
157     {
158       tabNode[arc->indexArrival].father = node;
159       tabNode[arc->indexArrival].lineUsed = arc;
160       tabNode[arc->indexArrival].cost = cost;
161     }
162     else
163       arc = arc->next;
164   }
165 }
166 
167 int getMin(dijNode_t * tab, int size, int * checked)
168 {
169   int i = 0, min = CONST_INFINITY, indexMin = -1;
170 
171   for(i = 0; i < size; i++)
172   {
173     if(!checked[i] && tab[i].cost > 0
174       && tab[i].cost < min)
```

```
175     {
176       min = tab[i].cost;
177       indexMin = i;
178     }
179   }
180 
181   return indexMin;
182 }
183 
184 float getRealTime(ELevel_t level, arc_t * arc)
185 {
186   float cost;
187   cost=arc->cost;
188 
189   if(arc->diff==GREEN)
190     cost=cost*(level == NEWBIE ? 1.1 : 1);
191   else if(arc->diff==BLUE)
192     cost=cost*(level == NEWBIE ? 1.5 : 1);
193   else if(arc->diff==RED)
194     cost=cost*(level == NEWBIE ? 2 : 1);
195   else if(arc->diff==BLACK)
196     cost=cost*(level == NEWBIE ? 3 : 1);
197   else
198     return arc->cost;
199 
200   return cost;
201 }
```

Dijkstra.h

```
1 #ifndef DIJ_H
2 #define DIJ_H
3 
4 #include "graph.h"
5 
6 typedef struct dijNode
7 {
8   float cost;// -1 == infinity
9   node_t * father;
10  arc_t * lineUsed; // for printing the path
11 }dijNode_t;
12 
13 typedef struct printStruct
14 {
15   dijNode_t * node;
16   struct printStruct * next;
17 }printStruct_t;
18 
19 void findShortestPath(user_t * user, graph_t * graph);
20 void updateNeighboors(node_t * node, dijNode_t * tabNode, user_t *
       user);
21 void freePath(printStruct_t * pile);
22 void printPath(dijNode_t * tabNode, int size, user_t * user,
       graph_t * graph);
23 
24 int getMin(dijNode_t * tab, int size, int * checked);
25 int getVisited(int * tab, int size);
```

```
26 float getRealTime(ELevel_t diff, arc_t * arc);
27
28 #endif
```

Graph.c

```
1
2 graph_t * initGraph(const char * filePath)
3 {
4   FILE *finput = fopen(filePath, "r");
5   int nbSommet = 0, i;
6   graph_t * graph = NULL;
7   arc_t * arcInit;
8   bool end = 0;
9   char read; //used to find end line
10  char diff;
11
12  if(finput != NULL)
13  {
14    if(fscanf(finput, "%d\n", &nbSommet))
15    {
16      graph = malloc(sizeof(graph_t));
17      graph->tabNode = calloc(sizeof(node_t), nbSommet);
18      graph->sizeTab = nbSommet;
19
20      for(i = 0; i < nbSommet; i++)
21      {
22        graph->tabNode[i].index = i;
23        graph->tabNode[i].arcs = NULL;
24      }
25
26      /*
27        TODO: faire une fonction add arc pour les node_t, bien set
    les trucs    NULL
28      */
29
30      for(i = 0; i < nbSommet; i++)
31      {
32        end = 0;
33        graph->tabNode[i].name = malloc(sizeof(char) *
    CONST_NAME_LENGTH);
34        readName(finput, graph->tabNode[i].name);
35        fseek(finput, -1, SEEK_CUR);//resets the cursor correctly
36
37        while(!end)
38        {
39          //see if end of line
40          fscanf(finput, "%c", &read);
41          if(read != '\n')
42          {
43            arcInit = malloc(sizeof(arc_t));
44            arcInit->name = malloc(sizeof(char) * CONST_NAME_LENGTH
    );
45            arcInit->next = NULL;
46
47            arcInit->indexStart = i;
48
```

```
            readName(finput, arcInit->name);
            fscanf(finput, "%d-%f-%c",
             &arcInit->indexArrival, &arcInit->cost, &diff);

            arcInit->diff = getDiffFromChar(diff);

            addArc(&graph->tabNode[i], arcInit);
          }
          else //end of line
            end = 1;
        }

      }
    }
  }

  fclose(finput);
  return graph;
}
```

Graph.h

```

#ifndef GRAPH_H
#define GRAPH_H

#include "../Actor/user.h"
#include "consts.h"

typedef struct arc
{
  EDiff_t diff; //stores the difficulty of the line
  float cost; //cost between this points
  char * name;
  int indexStart;
  int indexArrival; // the index of the destination

  struct arc * next;
} arc_t;

typedef struct node
{
  int index;
  char * name;
  arc_t * arcs;
} node_t;

typedef struct graph
{
  node_t * tabNode;
  int sizeTab;
}graph_t;

/* #nbNode
#node;#namearc-#nodeArrival-#cost-#diff;
*/

```

```
36 //loads the graph struct from the specified file
37 graph_t * initGraph(const char *filePath);
38
39 void freeGraph(graph_t * graph);
40 void freeNode(node_t * node);
41
42 void addArc(node_t * node, arc_t * arc);
43
44 #endif
```