Rapport du projet Courchevel

Adam KEDDIS

02 mai 2023

1 Description des données

Les données sont stockées dans des fichiers en *.txt*. Le nom du fichier est explicite et porte le nom des **structures des arcs**. La structure des données suit ce pattern :

Prédécesseur	**Successeur**	**Poids**	**Structure**
Folyères	Tania	3	VVV
Praz-juguet	Folyères	2	VVV
et cætera	*et cætera*	*et cætera*	*et cætera*

TABLE 1 – Tableau représentant le stockage des arcs

Les poids stockés dans les fichiers sont ceux d'un **professionnel**.
Ainsi il ne nous manquera plus qu'à multiplier le poids par les coefficients de difficultés attribuées au skieur. En ajoutant à **1** la difficulté nécessaire.

Piste	**Débutant**	**Intermédiaire**	**Professionnel**
Verte et bleue	1,75	1,45	1
Rouge	2,5	1,9	1
Noire	4	3	1

TABLE 2 – Coefficients de difficultés

La classe **Skieur** contient toutes les caractéristiques du skieur considéré :
— son prénom.
— sa vitesse.
— son lieu *qui change au cours du chemin.*
— sa destination.
— sa progression sur le chemin.
— sa position sur l'écran.
— et ses différentes matrices (*expliqué plus tard*).

La structure **position** contient toutes les positions relatives des différents lieux sur l'écran.

Lieu	**X**	**Y**
Merlet	70	200
Signal	40	400
et cætera	*et cætera*	*et cætera*

TABLE 3 – Stockage des positions

2 Initialisation des arcs

La structure choisie est relativement simple, elle permettra avec **une fonction intégrée à python** (*.split()*) de générer les arcs assez rapidement en considérant la ligne comme un tableau *cf Table 1.*

3 Les différentes matrices

Tout d'abord, l'initialisation des différentes matrices.
Une qui servira pour les **plus courts chemins**, l'autre qui servira à stocker les **successeurs** et enfin la dernière qui servira à stocker **les types de structures** qui nous permettrons de nous déplacer entre chaque arc.

L'initialisation se fait comme suit :

M_p la matrice des poids
M_st la matrice des structures
M_sc la matrice des successeurs

Début
$M_p \leftarrow Matrice$
$M_st \leftarrow Matrice$
$M_sc \leftarrow Matrice$
for $i = 1$ to n **do**
  for $j = 1$ to n **do**
    $M_p[i, j] \leftarrow +\infty$
    $M_st[i, j] \leftarrow None$
    $M_sc[i, j] \leftarrow None$
  end for
end for
Fin

Avec une fonction qui convertit les lieux en indices, qui ne sera pas mise en avant ici, ainsi le remplissage des matrices est relativement facile elle suit cette algorithme : *(cf annexe n°1)*

$ligne \leftarrow [predecesseur, successeur, poids, code]$
for each $ligne$ **do**
  if $M_p[ligne[0], ligne[1]] > ligne[2]$ **then**
    $M_p \leftarrow ligne[2]$
    $M_st \leftarrow ligne[3]$
  end if
  $M_sc[ligne[0], ligne[1]] \leftarrow ligne[1]$
end for

Ainsi en ayant le **prédécesseur** ainsi que le **successeur**, en transformant le *string* en *indice*, il est facile d'avoir le poids de l'arc, le type de structure de l'arc et le successeur (*qui servira plus tard*)

Code	Structure
VVV	piste verte
BBB	piste bleue
RRR	piste rouge
NNN	piste noire
TPH	téléphérique
TCB	télécabine
TSG	télésiège
TSI	téléski

TABLE 4 – Code associé à la structure

L'utilisation de l'algorithme de **Floyd-Warshall** pour trouver les **pcc** :

Algorithme de Floyd-Warhsall *(cf annexe n°2)*

for $k = 1$ to n **do**
  $M_sc[k, k] \leftarrow k$
  for $i = 1$ to n **do**
    for $j = 1$ to n **do**
      if $M_p[i, j] > M_p[i, k] + M_p[k, j]$ **then**
        $M_p[i, j] \leftarrow M_p[i, k] + M_p[k, j]$
        $M_sc \leftarrow M_sc[i, k]$
      end if
    end for
  end for
end for

Ainsi les **valeurs des pcc** pour chaque couple
de sommet possibles sont calculés
On a :

— $M_p[i,j]$ = **valeur du pcc de i à j**
— $M_sc[i,j]$ = **la structure de l'arc qui part de i vers j**
— $M_st[i,j]$ = **le successeur de i** sur le pcc allant de **i à j**

4 Fonctionnement Global

Voici l'ordre d'exécution :

— L'utilisateur donne son lieu de départ
— L'utilisateur donne sa destination
— Calcul du **pcc allant de lieu à destination**
— Calcul des **successeurs** *mit à la suite nous donne le chemin de i à j*
— Calcul des structures de chaque arc que l'utilisateur va prendre

Ainsi le défilement **successeur par successeur** donne le chemin jusqu'à que **successeur = destination**, alors l'utilisateur sera arrivé. *(cf annexe n°3)*

Pour le mouvement de l'utilisateur, il faut considérer **un espace vectoriel à 2 dimensions** où le vecteur de mouvement sera égal à :

$$\vec{mouvement} = (x_s - x_u, y_s - y_u) \tag{1}$$

Avec (x_u,y_u) = *position utilisateur*
et (x_s,y_s) = *position successeur*

Puis la *division de la différence par un nombre de frame* permet de se déplacer d'un Δx et d'un Δy par frame **donnant un aspect de mouvement réaliste.**

Les seules touches considérées dans l'interface sont :
— **ESPACE** permet d'avancer.
— **ECHAP** permet de reculer.

À la fin : les **pcc** allant de **lieu à destination** avec la succession des **sommets** et le **temps** nécessaire pour chaque arc sont affichés. Il y a donc **3 skieurs avec 3 niveaux différents** afin de montrer les **pcc** possibles en fonction du niveau du skieur.

5 Améliorations

L'algorithme de Floyd-Warshall possède une complexité en $O(n^3)$, et calcul tous les **pcc** pour **chaque couple de sommets**.
Hors ici il n'y a d'intérêt que pour un. Il aurait donc mieux valu utiliser **l'algorithme de Dijkstra** avec une complexité de $O(n^2)$

Pour profiter pleinement du coût de l'algorithme, il aurait pu y avoir plusieurs demande à l'utilisateur pour une destination *plutôt que de calculer pour un seul pcc.*

Une fonction qui calcule le temps d'exécution est jointe, qui met en exergue une croissance *polynomiale. (cf annexe n°4)*
Ici la matrice est en 67x67, il y figure un temps d'exécution minime à l'échelle humaine.
Mais pour un graphe plus grand il faudrait revoir la stratégie ...

6 Annexe

```
with open(vert, 'r') as filin:
    ligne = filin.readline().split()

    while ligne != []:
        if len(ligne) != 1:

            if matrice_distance[have_indice.name_to_indice(ligne[0]), have_indice.name_to_indice(ligne[1])] > int(ligne[2]) * coeff_vert:

                matrice_distance[have_indice.name_to_indice(ligne[0]), have_indice.name_to_indice(ligne[1])] = int(ligne[2]) * coeff_vert

                matrice_structure[have_indice.name_to_indice(ligne[0]), have_indice.name_to_indice(ligne[1])] = code.code_to_string(ligne[3])

            matrice_successeurs[have_indice.name_to_indice(ligne[0]), have_indice.name_to_indice(ligne[1])] = ligne[1]


        ligne = filin.readline().split()
```

FIGURE 1 – Annexe n°1 : Remplissage de la matrice

```
def floyd_warshall(matrice, successeurs):
    for k in range(len(matrice)):
        successeurs[k, k] = have_indice.indice_to_name(k)
        matrice[k, k] = 0
        for i in range(len(matrice)):
            for j in range(len(matrice)):
                if matrice[i, j] > matrice[i, k] + matrice[k, j]:
                    matrice[i, j] = matrice[i, k] + matrice[k, j]
                    successeurs[i, j] = successeurs[i, k]
```

FIGURE 2 – Annexe n°2 : Fonction de Floyd-Warshall

```
valeur_pcc = matrice_distance[have_indice.name_to_indice(lieu), have_indice.name_to_indice(destination)]

cheminement = []

chemin = ""
actuel = lieu
successeurs = matrice_successeurs[have_indice.name_to_indice(actuel), have_indice.name_to_indice(destination)]
cpt = 1

while actuel != destination:
    chemin += f"{cpt}) {successeurs} pour " \
              f"{matrice_distance[have_indice.name_to_indice(actuel), have_indice.name_to_indice(successeurs)]} " \
              f"minutes avec : " \
              f"{matrice_structure[have_indice.name_to_indice(actuel), have_indice.name_to_indice(successeurs)]}/"
    actuel = successeurs
    successeurs = matrice_successeurs[have_indice.name_to_indice(actuel), have_indice.name_to_indice(destination)]
    cpt += 1
    cheminement.append(actuel)

chemin += f'Le trajet prendra à {prenom} en tout : {valeur_pcc} minutes'

return chemin, cheminement
```

FIGURE 3 – Annexe n°3 : Chemin du skieur

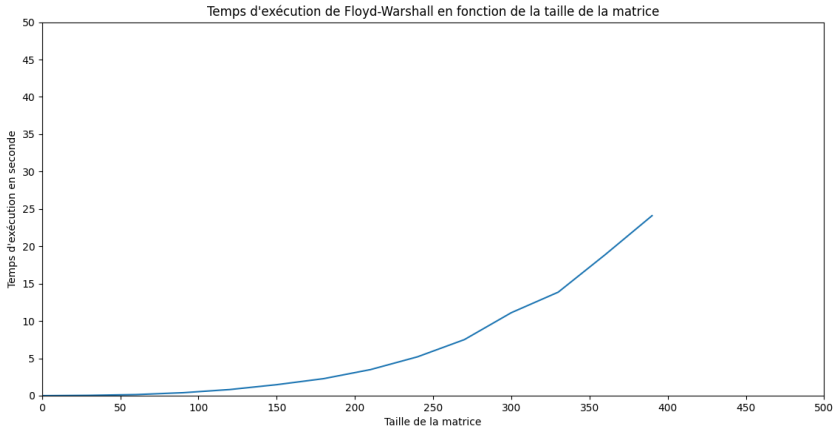

FIGURE 4 – Annexe n°4 : Durée d'exécution en seconde de Floyd-Warshall en fonction de la taille d'une matrice