Dijkstra et A*

TP 15

Graphes - v1.0

Lycée La Martinière Monplaisir, 41 Rue Antoine Lumière, 69372 Lyon

Pour les 2 premiers exercices, nous possédons un graphe de durée de trajet en minute entre différentes villes de France. Une représentation de ce graphe est donnée sur la Figure 1 et vous avez un fichier `Gduree.csv` correspondant à ce graphe. Nous avons également un fichier `coordonnes.csv` correspondant à une projection équirectangulaire des différentes villes, projection centrée sur le point de coordonnées $(46{,}5°, 3°)$ (point situé à mi-chemin entre Montluçon et Moulins).

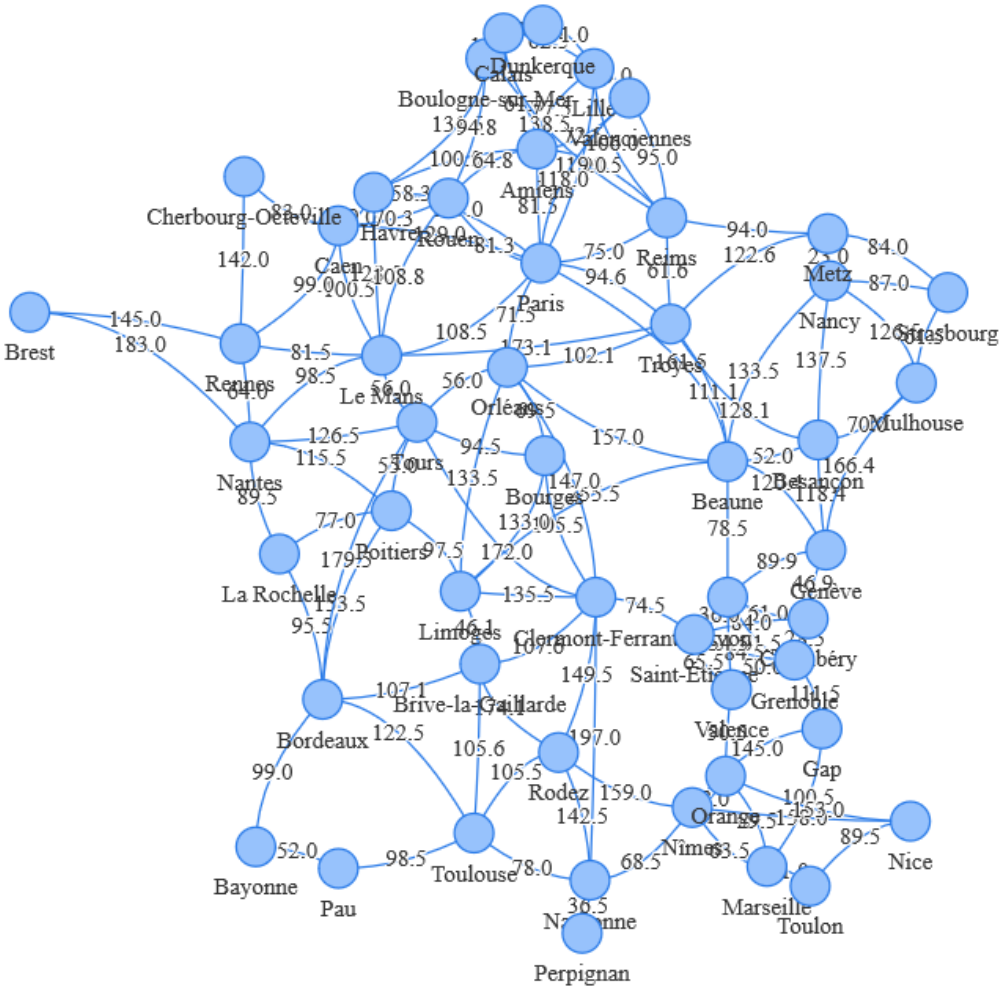

Figure 1 – Représentation du graphe pondéré

1 Optimisation de la liste de priorité avec le module `heapdict`

Le script `DijkstraComparaison_etudiants.py` permet de récupérer, à partir de `Gduree.csv`, la liste des 50 villes appelée `villes` et une représentation sous forme de dictionnaire d'adjacence pondéré pour le graphe : `G`.

Il possède également 2 fonctions pour rechercher la distance entre deux villes :

- un algorithme de Dijkstra avec une file de priorité codée avec un dictionnaire simple (`dict`) : `DureeDijkstraD` ;
- un algorithme de Dijkstra avec la file de priorité codée avec un dictionnaire par tas (`heapdict`) : `DureeDijkstraHd`.

On prendra comme notation $V = 50$ le nombre de sommets et $E = 122$ le nombre d'arête (notation anglosaxone provenant des termes *vertex/vertices* et *edge/edges*).

Question 1 Que renvoie les algorithmes `DureeDijkstraD` et `DureeDijkstraHd` ? Vérifiez que le trajet Reims-Grenoble dure toujours 312,7 minutes, quels que soient l'algorithme et le sens.

Pour comparer les 2 algorithmes, nous allons leur faire faire les calculs de dictionnaire de distance pour toutes les villes.

Nous utilisons la fonction `perf_counter_ns()` pour connaitre le temps de calcul : il suffit de placer le calcul dont on souhaite connaitre les performances entre les affectations de `tic` et `toc`. Ces deux affectations sont déjà présente dans le script.

Question 2 Quelle est la durée de calcul nécessaire pour calculer tous les trajets possibles en utilisant `DureeDijkstraD` ? Vérifier le résultat en répétant les calculs 100 fois. Sachant que la complexité théorique est de $O(V^2 + E)$, trouver le coefficient de proportionnalité k si on suppose que $t_{calcul} = k\left(V^2 + E\right)$.

Les différences de comportement et de performance entre les 2 structures sont présentées dans le tableau suivant (n correspond à la taille de la structure, `d` est le dictionnaire simple, `hd` est le dictionnaire optimisé, `o`, l'objet et `p` la priorité) :

Opération	Commande	Complexité	Commande	Complexité
Création vide	`d={}` ou `d=dict()`	$O(1)$	`hd=heapdict()`	$O(1)$
Ajout élément	`d[o]=p0`	$O(1)$	`hd[o]=p0`	$O\left(\log(n)\right)$
Modification élément	`d[o]=p1`	$O(1)$	`hd[o]=p1`	$O\left(\log(n)\right)$
Regarder minimum	`peekMin(d)` à coder	$O(n)$	`hd.peekitem()`	$O\left(1\right)$
Retirer minimum	`popMin(d)` à coder	$O(n)$	`hd.popitem()`	$O\left(\log(n)\right)$
Longueur	`len(d)`	$O(1)$	`len(hd)`	$O(1)$
Test vide	`if not d`	$O(1)$	`if not hd`	$O(1)$
Test appartenance	`if o in d`	$O(1)$	`if o in d`	$O(1)$
Obtention priorité	`d[o] -> p`	$O(1)$	`d[o] -> p`	$O(1)$

Question 3 Quelle est la complexité attendue pour `DureeDijkstraHd` ?

Question 4 En supposant que $t_{calcul} = k\left(V + E\right)\log_2(V)$ avec la même valeur de k que précédemment, quelle performance peut-on attendre de la modification faite entre les 2 algorithmes ? ;

Question 5 Vérifiez ce résultat en remplaçant `DureeDijkstraD` par `DureeDijkstraHd` dans le test de la performance temporelle.

Question 6 Le gain est-il aussi important qu'espéré ? Quelles sont les pistes qui permettent d'expliquer cette relative médiocrité ? En déduire des cas où le gain serait plus important.

2 De Dijkstra vers A*

Le script `DijkstraAetoile_etudiants.py` permet de récupérer, à partir de `Gduree.csv`, la liste des 50 villes appelée `villes` et une représentation sous forme de dictionnaire d'adjacence pondéré pour le graphe : `G`. De plus, il permet de construire un dictionnaire de coordonnées `coordonnees` à partir de `coordonnes.csv`, tel que `coordonnees[ville] -> (xVille, yVille)`.

Question 7 Mettre en route le script.

Vous devriez voir apparaitre 3 figures : l'une qui correspond aux villes visitées lors de la recherche d'un chemin entre Bordeaux et Reims, l'autre qui correspond aux villes visitées lors de la recherche d'une chemin entre Reims et Bordeaux. La surface des disques est proportionnelle à la priorité dans la file d'attente quand la ville a été enlevée.

Question 8 Constater que les visites ne font pas preuve de « bon sens ».

Pour résoudre le problème, l'algorithme A* reprend l'idée de l'algorithme de Dijkstra, mais ajoute une fonction heuristique permettant d'aller « dans la bonne direction ». Cette fonction s'ajoute à la distance réelle (distance provenant du graphe) et elle doit avoir la priorité de grandir quand on s'éloigne de l'arrivée.

On va utiliser assez naturellement une fonction proportionnelle à la distance à vol d'oiseau.

Question 9 Écrire une fonction `VolOiseau(ville1, ville2) -> float` qui renvoie la distance à vol d'oiseau entre 2 villes. Vous utiliserez utilement `coordonnees`.

Pour l'algorithme A* `DureeAetoile(G, depart, arrivee, heuristique)`, la priorité sera donc $p(ville) = duree(ville, depart) + k \times volOiseau(ville, arrivee)$.

Question 10 Quelle doit être la dimension de la constante de proportionnalité k ? Pour être performant, il faudrait que $k \times volOiseau(ville, arrivee)$ soit environ égale à la durée d'un trajet entre *ville* et *arrivee* : en déduire une valeur de k pertinente.

Question 11 Mettre en place la valeur de k trouvée (dans la fonction `heuristique`) et constater l'amélioration de l'algorithme.

Question 12 Que se passe-t-il si vous mettez une valeur beaucoup trop grande ou beaucoup trop petite ? Vous pourrez faire une multiplication/division par 100 du coefficient pour vous rendre compte des changements.

Question 13 Conclure sur les avantages et inconvénients de l'algorithme A* par rapport à l'algorithme de Dijkstra, en particulier sur les points suivants :

- optimalité ;
- complexité de mise en œuvre ;
- nécessité d'une expertise (au sens d'une compréhension du problème « physique » sous-jacent) et/ou d'un choix arbitraire à optimiser ;
- efficacité (en terme de nombre sommets visités).

3 Algorithme de pathfinding

Pour la suite, nous allons nous intéresser à un algorithme de recherche de chemin (*pathfinding*) basé sur l'algorithme A*. Ce type d'algorithme est bien sûr utilisé dans des jeux vidéos : dans les jeux de stratégie par exemple pour déplacer ses unités d'un endroit à un autre de la carte, mais aussi les jeux en 3D où tous les personnages non-joueur doivent se déplacer d'un endroit à un autre.

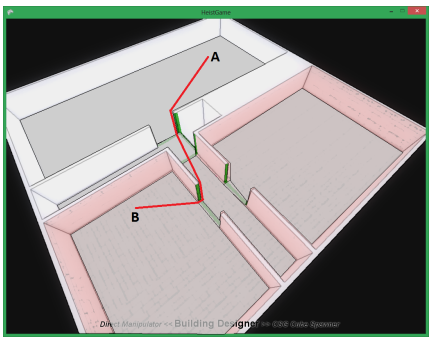

FIGURE 1 – Exemple de recherche de chemin dans un environnement 3D

Nous allons modéliser cela d'une façon très simple sur un damier de dimension 100×100 : chaque case accessible sera un sommet, et chaque déplacement d'une case à une voisine correspondra à une arête, et la durée de trajet sera le poids de cette arête. Les cases seront codées par un couple `(i, j)` (un tuple).

Le script `pathfinding_etudiants.py` permet de récupérer l'algorithme A* qui sera utilisé ici : l'algorithme retourne 2 listes et un flottant :

- le chemin d'un sommet `depart` à un sommet `arrivee` ;
- l'ensemble des sommets visités ;
- la durée du trajet (unité arbitraire).

3.1 Gestion d'un obstacle circulaire

L'obstacle aura la forme d'un disque : toutes les cases dont les coordonnées seront dans ce disque seront inaccessibles.

Question 1 Écrire une fonction `estDansDisque(pos, centre, rayon)` qui renvoie un booléen correspondant à la présence ou pas du point `pos` dans le disque défini par son centre et son rayon. Vous pourrez vous aider de la fonction `distance` qui donne la distance euclidienne entre 2 positions.

Une boucle imbriquée pour tester le fonctionnement de la fonction sur une figure est fournie dans le script sous forme commentée (raccourcis : `ctrl+T` pour décommenter, `ctrl+R` pour commenter).

Question 2 Vérifier que les points du disque apparaissent bien en noir tandis que les autres points apparaissent en bleu.

Pour la suite, on considérera que l'obstacle est un disque de centre `(50, 70)` et de rayon `20`.

Nous allons commencer simplement avec un terrain où la vitesse de déplacement est constante sur tout le terrain avec une valeur de 1 unité d'espace par unité de temps.

Question 3 Écrire une fonction `duree(pos1, pos2, vitesse)` qui renvoie la durée de trajet entre 2 positions pour une certaine vitesse.

Question 4 Écrire une fonction `listeVoisins(pos)` qui renvoie la liste des cases voisines pour une position donnée, sans considérer les obstacles, mais en considérant les bords : cela doit renvoyer une liste de 8 tuples dans le cas général, 5 tuples sur les bords et 3 tuples dans les coins.

Question 5 Créer le graphe `G` sous forme d'un dictionnaire d'adjacence pondéré.

Question 6 Tracer le trajet entre les points `(15, 20)` et `(80, 90)`.

Question 7 Tester différents poids pour la fonction heuristique : constatez les différences de trajets et de performances (durée du trajet et nombre de sommets visités).

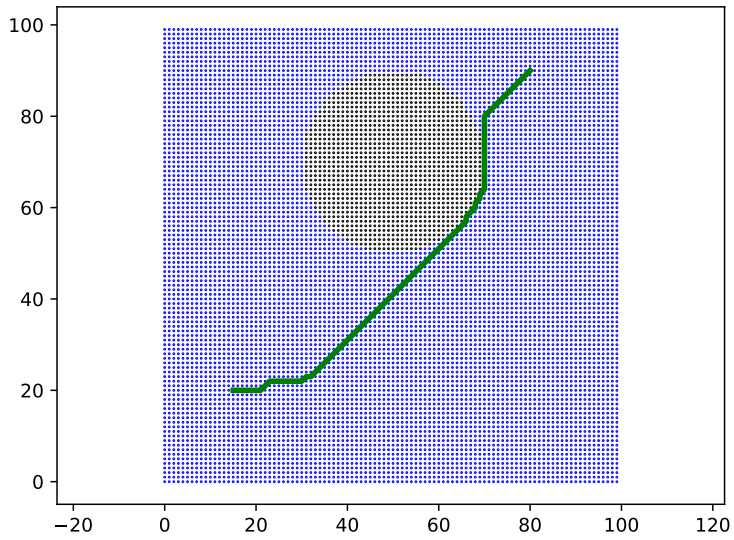

FIGURE 2 – Trajet trouvé entre `(15, 20)` et `(80, 90)`

3.2 Zone peu praticable

On souhaite ajouter une zone peu praticable dans notre damier (pour modéliser une forêt ou un marécage par exemple) : la vitesse sera environ 3 fois plus petite dans cette zone que dans une zone dégagée.

Cette zone sera défini par : tous les points dont l'abscisse est comprise entre 50 et 70.

Question 8 Écrire une fonction `zoneFaibleVitesse(pos)` qui renvoie un booléen correspond à la présence ou pas de `pos` dans la zone.

Question 9 Modifier le graphique présenté précédemment pour faire apparaitre cette zone en rouge.

La vitesse sera donc de `0.34` si les 2 sommets sont dans la zone, mais de `0.67` si un seul sommet est dans la zone.

Question 10 Écrire une fonction `vitesse(pos1, pos2)` qui renvoie `1`, `0.67` ou `0.34` suivant les cas.

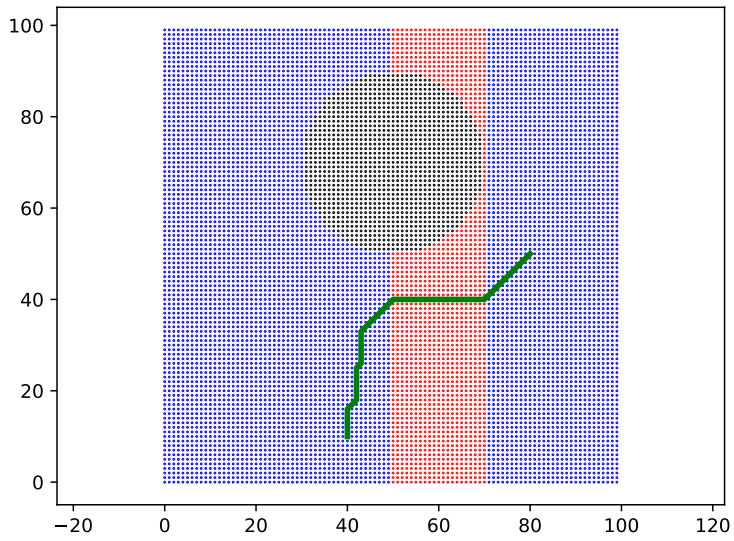

FIGURE 3 – Trajet trouvé entre `(40, 10)` et `(80, 50)`

Question 11 Modifier la création du graphe `G` pour prendre en compte cette information.

Question 12 Tester différents poids pour la fonction heuristique : constatez les différences de trajets et de performances (durée du trajet et nombre de sommets visités) pour un trajet entre `(40, 10)` et `(80, 50)`.