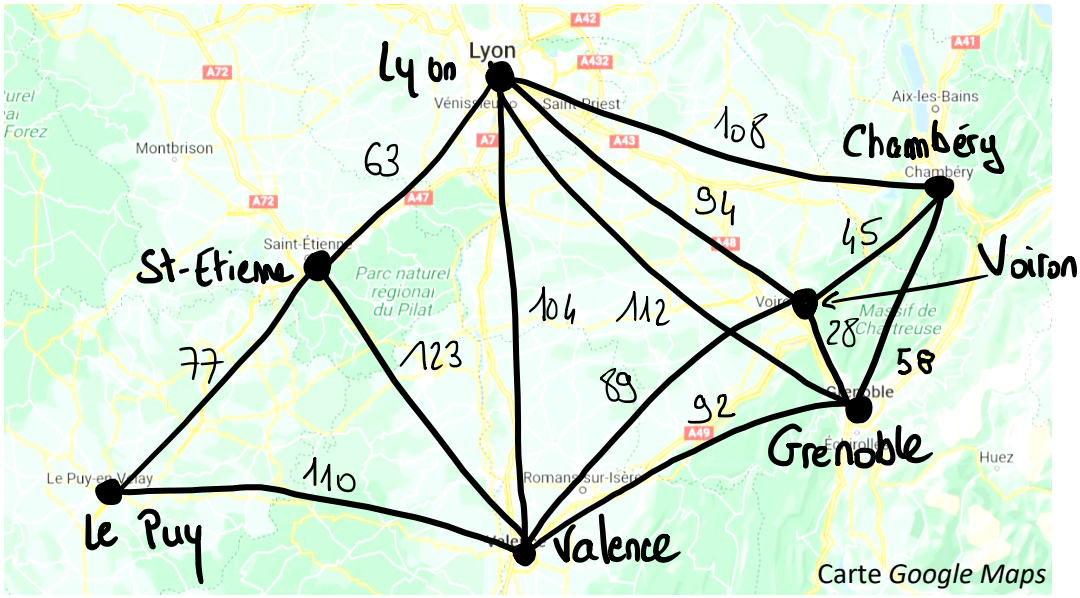
TD 23 – Partie 4 – Algorithme *A** – recherche du plus court chemin dans un graphe pondéré

Nous menons ici l'étude de l'intérêt de l'algorithme *A** par rapport à l'algorithme de *Dijkstra* vu à la dernière séance. Nous repartons donc du contexte de recherche du plus court trajet (en km) entre deux villes de la carte ci-dessous :



Objectif du TD :

Un corrigé (un peu modifié) du sujet précédent est donné dans le fichier *Dijkstra_Astar_eleve.py*. Vous allez partir de cette base, modifier légèrement l'algorithme de *Dijkstra*, puis vous en inspirer fortement pour construire l'algorithme *A**. Cela vous permettra de comparer l'efficacité relative des deux algorithmes.

A / Modification (légère) de *Dijkstra*

Vous remarquerez, à l'ouverture du fichier *Dijkstra_Astar_eleve.py*, que j'ai un peu modifié les arguments des fonctions, en ajoutant deux arguments (dont l'un des deux est optionnel) :

- `Dijkstra(Graphe, N_0, N_a = None, H = None)`
- `plus_court_chem(Graphe, N_0, N_a, H = None)`

Comprenons ensemble pourquoi ces ajouts. Je rappelle que l'objectif du TP est de comparer « l'efficacité » de *A** par rapport à *Dijkstra*.

- `N_a` : cet argument désigne le nœud d'arrivée. Par défaut, l'algorithme de *Dijkstra* s'exécute jusqu'à avoir trouvé les chemins optimaux menant depuis le nœud initial `N_0` jusqu'à **tout nœud** du graphe. Or, l'algorithme *A** est prévu pour trouver le chemin optimal menant d'un nœud initial `N_0` vers un nœud d'arrivée `N_a`. Pour pouvoir les comparer, nous allons donc modifier *Dijkstra* pour qu'il s'arrête dès qu'on a trouvé le chemin le plus court menant à un nœud d'arrivée pré-défini `N_a`, placé en argument de la fonction.
- `H` : cet argument désignera l'heuristique (estimation basse du coût restant à parcourir entre tout sommet $S_i \in S$ et le nœud d'arrivée `N_a`. Étant donné que *Dijkstra* n'utilise pas d'heuristique, cet argument n'est pas utilisé pour l'instant, d'où l'initialisation en tant qu'argument optionnel valant `None`.

Avant d'entamer le TD, veuillez copier le fichier *Dijkstra_Astar_eleve.py* sur votre session, et travailler sur cette copie.

Q1 – Modifier dans le script la fonction `Dijkstra(Graphe, N_0, N_a = None, H = None)` pour que l'algorithme s'arrête dès qu'on connait le chemin optimal entre `N_0` et `N_a`.

Remarque : il s'agit d'une seule ligne à modifier ! Avec le test proposé, vous verrez le nombre d'itérations diminuer à 22.

B / Fonctions auxiliaires utiles pour *A**

Nous rappelons que l'algorithme *A** fait appel à trois fonctions :

- g : qui à un sommet S_i du graphe associe le coût estimé $g(S_i) \geq 0$ du meilleur chemin alors trouvé, menant du nœud initial `N_0` à ce sommet S_i.
- h : heuristique, que l'on suppose **cohérente** dans ce sujet, qui à tout sommet S_i du graphe associe le coût estimé $h(S_i) \geq 0$ du meilleur chemin restant à parcourir entre ce sommet S_i et l'objectif `N_a`.
- $f = g + h$ la fonction d'évaluation, qui estime donc, pour un sommet S_i du graphe, le coût total du chemin menant du départ `N_0` à l'objectif `N_a` en passant par ce sommet S_i.

Il est toutefois important de noter une subtilité :

- D, pour Dijkstra, était une application, c'est-à-dire que tout nœud S_i de *S* a une image $D(S_i) \in \mathbb{R}_+$. En effet, nous l'avions initialisé à l'infini pour tout nœud, et 0 pour `N_0`. Pour *A**, h est aussi une application : on associe à tout sommet S_i de *S* a une image $g(S_i) \in \mathbb{R}_+$, qui vaut 0 pour `N_0` .

- En revanche, pour *A** la fonction g n'est pas une application ! À l'initialisation, seul le sommet initial `N_0` a une image (g (`N_0`) qui vaut 0). Puis, au fur et à mesure que l'algorithme va découvrir le graphe, de plus en plus de sommets S_i du graphe se verront associée une image $g(S_i)$ (sachant que parfois, lors de la relaxation, l'image d'un sommet $g(S_i)$ pourra être mise à jour si l'on a trouvé un meilleur chemin `N_0`$\rightarrow S_i$).

  A fortiori, f sera défini sur le même ensemble que *g*, qui est donc un sous-ensemble de S.

Tout comme nous l'avons fait avec l'application *D* pour *Dijkstra*, nous allons implémenter ces fonctions au moyen de dictionnaires, qui seront notés *G*, *H* et *F* par la suite. Ces dictionnaires auront pour clé les nœuds S_i du graphe sur lesquels ces fonctions sont définies, et pour valeurs associées, respectivement $g(S_i)$, $h(S_i)$ et $f(S_i)$.

Suite à la remarque précédente, vous comprenez que pour tout nœud du graphe, `H[Noeud]` existe, mais en revanche il existe des nœuds pour lesquels `G[Noeud]` ou `F[Noeud]` n'existent pas, et renverraient une erreur :

```
KeyError: Noeud
```

En d'autres termes, pour un graphe donné, noté `Graphe`, les listes `G.keys()` et `F.keys()` sont égales, et incluses dans la liste `Graphe.keys()`, qui est égale à `H.keys()`. Pour la mise à jour de *G* et *F*, nous allons faire appel à une fonction auxiliaire.

Q2 – Écrire une fonction `somme(dico1, dico2)` qui prenne en argument deux dictionnaires tels que les clés de `dico1` sont incluses dans les clés de `dico2`. Cette fonction renverra un dictionnaire ayant les mêmes clés que `dico1` et qui à chacune de ces clés associe la somme des valeurs, pour `dico1` et `dico2`, associées à cette clé.

Exemple / test : posons

```
D1 = {'A': 1, 'B': 2}
D2 = {'A': 2, 'B': 4}
D3 = {'A': 3}
```

→ Alors :

```
print(somme(D1,D2))   # doit afficher {'A': 3, 'B': 6}
                      # idem si on print(somme(D2,D1))
print(somme(D3,D1))   # doit afficher {'A': 4}
print(somme(D1,D3))   # génère une erreur KeyError: 'B'
                      # car la clé 'B' de D1 n'existe pas
                      # dans D3
```

D'autre part, on rappelle qu'on avait codé pour *Dijkstra* la fonction suivante :

`relachement(Graphe, N_d, N_v, Dist, P)`, qui prend en argument un graphe, un nœud de départ (attention, ce n'est pas le nœud initial `N_0` !), un nœud `N_v` voisin de `N_d`, et deux fonctions `Dist` (correspondant à `D` pour *Dijkstra*, `G` pour *A**) et `P` (dictionnaire contenant les parents).

Nous allons devoir modifier légèrement cette fonction pour prévoir le cas où `N_v` n'avait jamais été aperçu auparavant, auquel cas il n'a pas encore d'images `Dist[N_v]` ni `P[N_v]`. Ce cas ne servira pas pour *Dijkstra*, mais sera utile pour *A**.

Q3 – Modifier la fonction `relachement(Graphe, N_d, N_v, Dist, P)` pour prévoir le cas cité ci-dessus.

Petit encart « le saviez-vous ? » : *une fonction peut prendre en argument une fonction !*

```
def f1(x) :  # Soit une première fonction
    return 2*x

def f2(x) :  # ... et une seconde fonction
    return 3*x

def f_appel(fonction, x) : # c'est ce qu'on nomme une fonction d'appel, elle
                           # prend en argument une fonction définie par ailleurs
    return fonction(x)+1
```

Alors, si on appelle :

`f_appel(f1, 2)` → c'est `f1(2)+1` qui est calculé, donc on renvoie `5`

`f_appel(f2, 2)` → c'est `f2(2)+1` qui est calculé, donc on renvoie `7`

L'exemple ci-dessus n'a pas beaucoup d'intérêt, si ce n'est d'être illustratif. Ce genre de fonctions d'appel présente un intérêt lorsqu'on veut pouvoir comparer deux algorithmes sensés parvenir au même résultat, c'est notre cas ici.

Q4 – Remplacer le code de la fonction `plus_court_chem(...)` par le code suivant :

```
def plus_court_chem(Graphe, algo, N_0, N_a, H = None): # algo choisi pour la recherche
                                                       # du chemin optimal
    Parents, Nb_iterations = algo(Graphe,N_0,N_a, H) # c'est la seule ligne qui change!
    chemin = [N_a] # on part de l'arrivée
    pere = Parents[N_a]
    while pere != None : # on remonte jusqu'à N_d, qui n'a pas de "père" avec Dijkstra
        chemin.insert(0,pere)
        pere = Parents[pere] # on "remonte" d'une génération
    return chemin, Nb_iterations
```

Q5 – Modifier maintenant l'appel qui doit normalement être actuellement en dernière ligne de votre code :

```
print(plus_court_chem(km_traj,"Le Puy","Chambéry"))
```

pour qu'elle affiche, comme avant, le meilleur chemin et le nombre d'itérations pour le trajet demandé, en utilisant l'algorithme de *Dijkstra*.

C / Algorithme *A**, heuristique et test

D.1 / Algorithme *A**

Q6 – Copier-coller le code de la fonction `Dijkstra(...)`, et modifier quelques lignes pour obtenir la fonction `Astar(Graphe, N_0, N_a, H)`, qui prend en argument un graphe, deux nœuds du graphes (un initial, l'autre objectif), et une heuristique h implémentée comme dictionnaire. Cette fonction renverra, comme *Dijkstra*, le dictionnaire des parents, et le nombre d'itérations réalisées par l'algorithme avant qu'il ne termine.

Aide : • L'initialisation (avant le *while*) de `G` et `F` est à faire. Comme déjà vu en partie **B**, là où `Dist` était une application (associant une valeur à tout nœud du graphe), `G` ne l'est pas ! Le seul nœud connu est `N_a`. Voir cours.

• Les deux structures itératives sont extrêmement proches. Vous n'aurez qu'à ajouter une ligne pour la mise à jour de `F` (qui utilisera la fonction `somme(...)` codée en **Q**), et modifier très légèrement 2 lignes.

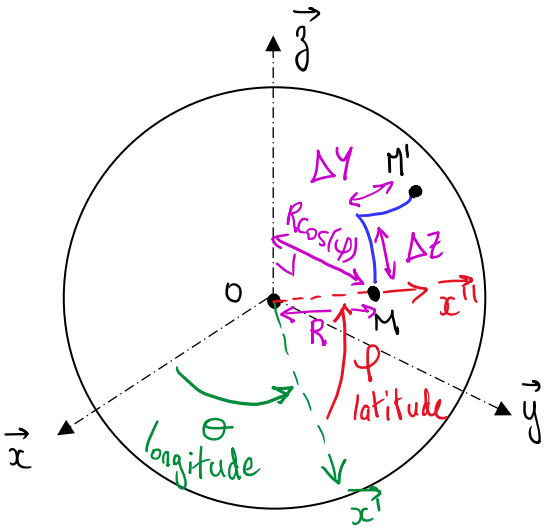
→ On aimerait bien pouvoir tester notre algorithme… mais pour cela il nous faut une heuristique h, qui doit être définie avant l'appel de l'algorithme (d'où le fait que `H` soit un argument de la fonction `Astar`).

D.2 / Heuristique

On se propose de fixer comme nœud d'arrivée (objectif) une ville donnée `N_a`, et de construire comme heuristique la distance géodésique approximative séparant toute ville du graphe de la ville `N_a`.

On se propose de fixer comme nœud d'arrivée (objectif) une ville donnée `N_a`, et de construire comme heuristique la distance géodésique approximative séparant toute ville du graphe de la ville `N_a`.

On se propose de fixer comme nœud d'arrivée (objectif) une ville donnée `N_a`, et de construire comme heuristique la distance géodésique approximative séparant toute ville du graphe de la ville `N_a`.



On assimile la Terre à une sphère de rayon R = 6370 km

Pour chaque ville S_i, on dispose de ses coordonnées GPS :
(φ_i, θ_i) avec φ_i latitude et θ_i longitude, données **en degrés** !

Pour deux villes S_i , S_j , on appelle distance géodésique ΔL la longueur curviligne de l'arc $\overset{\frown}{S_iS_j}$ en suivant la courbure de la Terre (dans le langage courant « distance à vol d'oiseau »).

Si les deux villes sont proches, c'est-à-dire si $\Delta L \ll R$, ce qui est notre cas ici puisqu'on reste au sein de la même région, on peut poser :

$$\Delta Z = R\Delta\varphi = R(\varphi_j - \varphi_i) \quad \text{(distance Nord-Sud)}$$
$$\varphi_M = \frac{\varphi_i+\varphi_j}{2} \text{ (latitude médiane)}$$
$$\Delta Y = R\cos(\varphi_M)\,\Delta\theta = R\cos(\varphi_M)\,(\theta_j - \theta_i) \quad \text{(distance Est-Ouest)}$$
$$\Delta L \approx \sqrt{\Delta Y^2 + \Delta Z^2} \quad \text{(distance géodésique approchée)}$$

Admettons que les coordonnées GPS, sous la forme (latitude, longitude), sont stockées dans un dictionnaire `Coord_GPS`. Ainsi, `Coord_GPS[Ville]` renvoie un tuple de taille 2 qui correspond aux coordonnées du centre-ville.

Q6 – Écrire une fonction `generer_normes(Graphe, N_a, Coord_GPS)` qui prenne en argument un graphe (dont les sommets sont des villes), une ville de destination `N_a` et un dictionnaire `Coord_GPS` contenant pour chaque ville du graphe les coordonnées de cette ville. La fonction renverra un dictionnaire, ayant les mêmes clés que `Graphe`, et associant à chaque clé la distance géodésique entre la ville concernée et `N_a`.

Pour les villes de la région Rhône-Alpes étudiées, un dictionnaire `GPS` est donné comme variable globale en fin de code (dans l'espace d'appel des fonctions). On souhaite se rendre à Chambéry.

Q7 – Générer l'heuristique `H_GPS`, variable globale, correspondant aux distances géodésiques de chacune des villes de la région à Chambéry, en km. Les distances seront arrondies à ± 10 m, c'est-à-dire ± 0.01 km.

Aide : `round(x, n)` renvoie l'arrondi d'un *float* `x` avec `n` (type *int*, $n \geq 0$) chiffres après la virgule.

Test : afficher `H_GPS` doit vous donner ceci :
{'Lyon': 87.21, 'Voiron': 34.19, 'Chambéry': 0, 'St-Etienne': 120.28, 'Valence': 106.92, 'Le Puy': 169.3, 'Grenoble': 44.5}

Q8 – Finalement, à l'aide de la fonction `plus_court_chem(...)` appelée pour cette heuristique, afficher le meilleur trajet et le nombre d'itérations calculées par l'algorithme *A**. Comparer par rapport l'efficacité de l'algorithme *Dijkstra*.