Chapitre 6

Localisation, cartographie et mobilité

Objectifs du chapitre

Thème : Localisation, cartographie et mobilité.
Comprendre le principe de la géolocalisation par satellites, utiliser des cartes numériques, décoder une trame NMEA, représenter un calcul d'itinéraire comme un problème de plus court chemin dans un graphe et maîtriser le partage de sa position.
Organisation : programme officiel, cours, méthodes, ateliers Python, exercices corrigés, QCM et fiche de synthèse.

Programme officiel (BO)

Thème « Localisation, cartographie et mobilité » – Contenus et capacités attendues

— **GPS, Galileo** : Décrire le principe de fonctionnement de la géolocalisation.

— **Cartes numériques** : Identifier les différentes couches d'information de GeoPortail pour extraire différents types de données. Contribuer à OpenStreetMap de façon collaborative.

— **Protocole NMEA 0183** : Décoder une trame NMEA pour trouver des coordonnées géographiques.

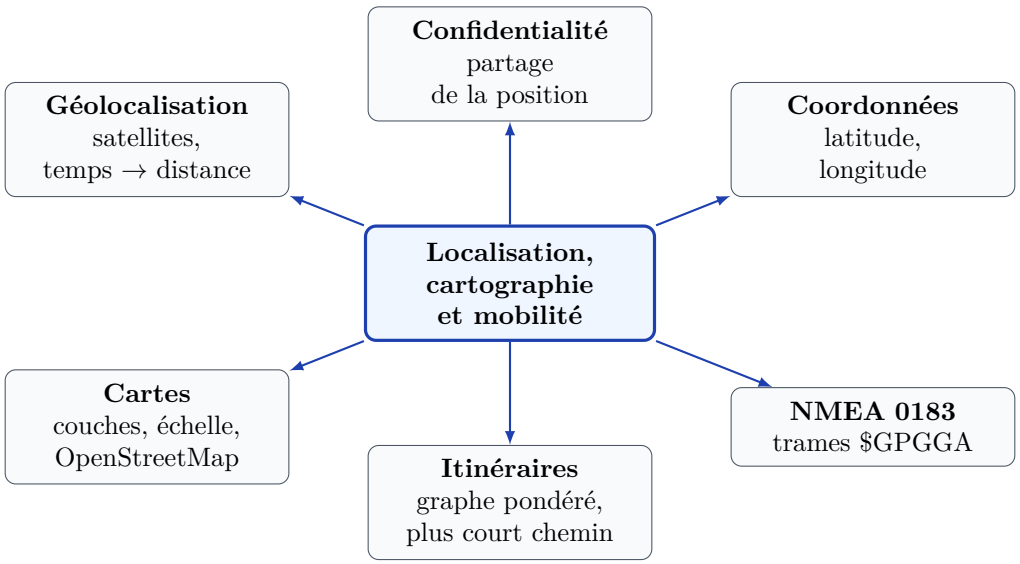
— **Calculs d'itinéraires** : Utiliser un logiciel pour calculer un itinéraire. Représenter un calcul d'itinéraire comme un problème sur un graphe.

— **Confidentialité** : Régler les paramètres de confidentialité d'un téléphone pour partager ou non sa position.

Source : BO spécial n°1 du 22 janvier 2019, programme de Sciences numériques et technologie de Seconde générale et technologique.

6.1 Carte mentale du chapitre

Carte mentale



6.2 La géolocalisation par satellites

Les systèmes de positionnement par satellites permettent à un récepteur (téléphone, voiture, montre) de connaître sa position partout sur Terre, avec une précision de quelques mètres. Le **GPS** américain (opérationnel en 1995) est le plus connu ; l'Europe a développé **Galileo** (services ouverts depuis 2016), la Russie GLONASS et la Chine BeiDou. Un téléphone récent utilise plusieurs de ces systèmes à la fois.

Définition – Coordonnées géographiques

- La **latitude** mesure, en degrés, l'angle au nord (N) ou au sud (S) de l'équateur : de 0° à 90°.
- La **longitude** mesure l'angle à l'est (E) ou à l'ouest (W) du méridien de Greenwich : de 0° à 180°.
- Exemple : la tour Eiffel est à environ 48,8584° N et 2,2945° E. On écrit aussi 48° 51,504′ N (degrés et minutes : $1° = 60'$).

Propriété – Principe de la géolocalisation

1. Une trentaine de satellites tournent autour de la Terre à environ **20 000 km** d'altitude ; chacun possède une horloge atomique très précise.
2. Chaque satellite émet en permanence un signal contenant **sa position** et l'**heure d'émission**.
3. Le récepteur mesure la durée de trajet du signal et en déduit sa distance au satellite : $\mathbf{d = c \times \Delta t}$, avec $c \approx 300\,000$ km/s (vitesse de la lumière).
4. Connaissant ses distances à plusieurs satellites, il calcule sa position par **trilatération** : il faut au moins **4 satellites** (3 pour la position, un 4^{e} pour corriger l'horloge du récepteur, moins précise).

Attention

Le récepteur ne fait que **recevoir** : il n'émet rien vers les satellites, qui ne savent pas où il est. Ce sont les applications du téléphone qui peuvent ensuite transmettre la position à des serveurs.

Exemple – Pourquoi les horloges doivent être si précises

Un signal parcourt 300 000 km en une seconde, donc 300 m en une microseconde (10^{-6} s). Une erreur d'une microseconde sur l'horloge produit une erreur de position d'environ **300 m**.

6.3 Les cartes numériques

Définition – Couches d'information

Une carte numérique est formée de **couches** superposées que l'on peut afficher ou masquer : fond de plan, photographies aériennes, relief, routes, limites administratives, cadastre, transports, zones inondables… Le site public **Géoportail** (IGN) propose des centaines de couches. Selon l'**échelle** (le niveau de zoom), le logiciel n'affiche pas les mêmes détails : les petites routes disparaissent quand on regarde tout le pays.

Définition – OpenStreetMap

OpenStreetMap (OSM) est une carte du monde **collaborative et libre**, construite par des millions de contributeurs bénévoles, sur le modèle de Wikipédia. Chacun peut y ajouter un banc, un chemin, un commerce. Ses données, sous licence libre, sont réutilisées par de nombreuses applications.

6.4 Le protocole NMEA 0183

Un récepteur GPS transmet ses informations sous forme de lignes de texte appelées **trames**, selon la norme **NMEA 0183**. La trame `$GPGGA` donne la position :

```
$GPGGA,143512.00,4851.5040,N,00217.6700,E,1,08,0.9,35.0,M,46.9,M,,*5C
```

Champ	Valeur	Signification
1	`$GPGGA`	type de trame (GP = GPS, GGA = position)
2	`143512.00`	heure UTC : 14 h 35 min 12 s
3 et 4	`4851.5040,N`	latitude : 48° 51,5040′ Nord
5 et 6	`00217.6700,E`	longitude : 002° 17,6700′ Est
7	`1`	qualité : 1 = position GPS valide (0 = pas de position)
8	`08`	nombre de satellites utilisés
9	`0.9`	précision horizontale (plus c'est petit, mieux c'est)
10 et 11	`35.0,M`	altitude : 35,0 m
fin	`*5C`	somme de contrôle pour détecter les erreurs de transmission

Méthode – Convertir en degrés décimaux

La latitude `4851.5040` se lit **48** degrés et **51,5040** minutes. Comme 1° = 60′ : 48 + 51,5040 ÷ 60 ≈ **48,8584°**. Pour la longitude, les degrés sont sur 3 chiffres : `00217.6700` donne 2 + 17,67 ÷ 60 ≈ **2,2945°**. Une latitude S ou une longitude W devient négative.

```
1 def en_degres(valeur, nb_chiffres_degres):
2     degres = int(valeur[:nb_chiffres_degres])
3     minutes = float(valeur[nb_chiffres_degres:])
```

```
    return degres + minutes / 60

print(en_degres("4851.5040", 2))    # latitude : 2 chiffres pour les degres
print(en_degres("00217.6700", 3))   # longitude : 3 chiffres pour les degres
```

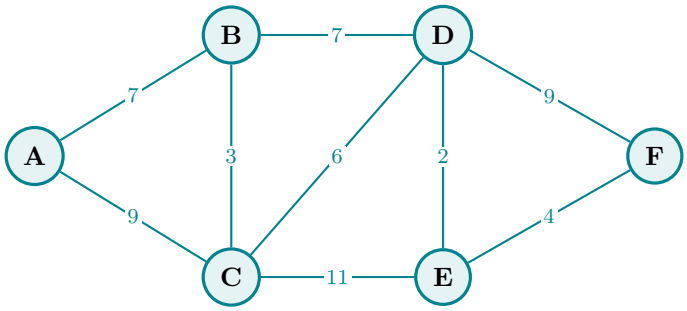
Animation interactive « Décoder une trame NMEA » : à manipuler sur https://utspe.com/snt/chapitre-6.

6.5 Calcul d'itinéraires

Définition – Graphe pondéré

Pour calculer un itinéraire, un logiciel représente le réseau routier par un **graphe** : les sommets sont les carrefours et les villes, les arêtes sont les tronçons de route. Chaque arête porte un **poids** : une distance, une durée, un coût (péage, carburant). Le graphe est dit **pondéré**. La **longueur** d'un chemin est la somme des poids de ses arêtes. Calculer le meilleur itinéraire, c'est chercher un **plus court chemin** entre deux sommets.

Voici un réseau de six carrefours ; les poids sont des durées en minutes.



Propriété – L'algorithme de Dijkstra (1959)

1. Au départ, la distance du sommet de départ vaut 0 et celle des autres est inconnue (∞).
2. On choisit, parmi les sommets non encore fixés, celui dont la distance provisoire est la plus petite, et on le **fixe**.
3. Pour chacun de ses voisins, on calcule la distance en passant par lui ; si elle est plus petite que la distance provisoire, on la remplace.
4. On recommence jusqu'à avoir fixé le sommet d'arrivée.

Ici, de A à F : on fixe A (0), B (7), C (9), D (14), E (16) puis F (20). Le plus court chemin est **A** $\to$ **B** $\to$ **D** $\to$ **E** $\to$ **F**, de longueur **20 minutes**.

Animation interactive « Le plus court chemin pas à pas » : à manipuler sur https://utspe.com/snt/chapitre-6.

Les logiciels de navigation (OpenStreetMap avec ses calculateurs d'itinéraires, applications GPS) utilisent des variantes très rapides de cet algorithme sur des graphes de plusieurs millions de sommets, en tenant compte du trafic en temps réel.

6.6 Confidentialité de la position

Propriété – Ce que révèle la position

L'historique de nos positions révèle notre domicile, notre lycée, nos habitudes, nos fréquentations, voire notre santé ou nos opinions (lieux de culte, manifestations). C'est une donnée personnelle très sensible.

Méthode – Régler le partage de sa position

— désactiver la localisation quand elle est inutile, ou n'autoriser une application que **pendant son utilisation** ;
— vérifier, application par application, qui a accès à la position (Réglages, Confidentialité, Service de localisation sur iPhone ; Paramètres, Localisation sur Android) ;
— désactiver la position **précise** quand une position approximative suffit (météo) ;
— désactiver l'historique des positions et la géolocalisation des photos ;
— ne partager sa position en temps réel qu'avec des proches, et pour une durée limitée.

6.7 Ateliers Python

Les ateliers se font sur ordinateur, dans un éditeur Python ou directement sur https://utspe.com/snt/chapitre-6. Pour chacun : recopier le programme de départ, suivre les étapes dans l'ordre et exécuter le programme après chaque modification.

Atelier 1 – Du temps de trajet à la distance calculs, f-strings, fonctions

Objectif : Calculer la distance à un satellite à partir de la durée de trajet de son signal, et mesurer l'effet d'une erreur d'horloge.

1. Exécuter le programme : il calcule la durée de trajet d'un signal émis par un satellite situé à 20 200 km.
2. Écrire une fonction `distance(duree)` qui renvoie la distance en km parcourue par le signal en `duree` secondes.
3. Le récepteur mesure 0,0712 s pour un autre satellite. À quelle distance est-il ?
4. Calculer l'erreur de distance provoquée par une erreur d'horloge d'une microseconde (0,000001 s), en mètres.

Programme de départ :

```
c = 300000      # vitesse de la lumiere en km/s

altitude = 20200
duree = altitude / c
print(f"Duree du trajet : {duree:.4f} s")

# a completer
```

Coup de pouce : `def distance(duree): return c * duree`. Pour l'erreur : `distance(0.000001)` donne des kilomètres ; multiplier par 1 000 pour obtenir des mètres.

Corrigé rédigé

```
c = 300000      # vitesse de la lumiere en km/s

altitude = 20200
```

```python
duree = altitude / c
print(f"Duree du trajet : {duree:.4f} s")

def distance(duree):
    return c * duree

print(f"Satellite a {distance(0.0712):.0f} km")
erreur = distance(0.000001) * 1000
print(f"Erreur d'une microseconde : {erreur:.0f} m")
```

Atelier 2 – Décoder une trame NMEA

split(), tranches, float()

Objectif : Écrire un programme qui extrait d'une trame $GPGGA l'heure, la latitude, la longitude et l'altitude, en degrés décimaux.

1. Exécuter : `trame.split(",")` donne la liste des champs. Repérer l'indice de chaque information.
2. Afficher l'heure sous la forme `14 h 35 min 12 s` à partir de `champs[1]`.
3. Utiliser `en_degres` pour calculer la latitude et la longitude ; les rendre négatives si l'hémisphère est S ou W.
4. Tester avec la trame `$GPGGA,170834.00,4322.0710,S,07027.4560,W,1,07,1.2,612.0,M,30.1,M,,*6D`.

Programme de départ :

```python
def en_degres(valeur, nb_chiffres_degres):
    degres = int(valeur[:nb_chiffres_degres])
    minutes = float(valeur[nb_chiffres_degres:])
    return degres + minutes / 60

trame = "$GPGGA,143512.00,4851.5040,N,00217.6700,E,1,08,0.9,35.0,M,46.9,M,,*5C"
champs = trame.split(",")
for i in range(len(champs)):
    print(i, champs[i])

# a completer
```

Coup de pouce : L'heure : `h = champs[1][0:2]`, `m = champs[1][2:4]`, `s = champs[1][4:6]`. Latitude : `en_degres(champs[2], 2)`, et `if champs[3] == "S": lat = -lat`.

Corrigé rédigé

```python
def en_degres(valeur, nb_chiffres_degres):
    degres = int(valeur[:nb_chiffres_degres])
    minutes = float(valeur[nb_chiffres_degres:])
    return degres + minutes / 60

def decoder(trame):
    champs = trame.split(",")
    heure = champs[1]
    print(f"Heure : {heure[0:2]} h {heure[2:4]} min {heure[4:6]} s (UTC)")
    lat = en_degres(champs[2], 2)
    if champs[3] == "S":
        lat = -lat
    lon = en_degres(champs[4], 3)
    if champs[5] == "W":
        lon = -lon
    print(f"Latitude : {lat:.4f} - Longitude : {lon:.4f}")
```

```
    print(f"Altitude : {champs[9]} m - {int(champs[7])} satellites")

decoder("$GPGGA,143512.00,4851.5040,N,00217.6700,E,1,08,0.9,35.0,M,46.9,M,,*5C")
decoder("$GPGGA,170834.00,4322.0710,S,07027.4560,W,1,07,1.2,612.0,M,30.1,M,,*6D")
```

Atelier 3 – Distance entre deux villes

module math, fonctions

Objectif : Calculer la distance « à vol d'oiseau » entre deux points connus par leurs coordonnées, avec la formule utilisée par les applications de cartographie.

1. Exécuter : la fonction `distance_km` utilise la formule de haversine (le rayon de la Terre vaut 6 371 km). Elle calcule la distance Paris–Lyon.
2. Calculer la distance Paris–Marseille (Marseille : 43,2965° N ; 5,3698° E).
3. Une voiture met 7 h 30 pour aller de Paris à Marseille par la route (environ 775 km). Comparer avec la distance à vol d'oiseau : de combien de pour cent la route est-elle plus longue ?

Programme de départ :

```
import math

def distance_km(lat1, lon1, lat2, lon2):
    p1 = math.radians(lat1)
    p2 = math.radians(lat2)
    dp = math.radians(lat2 - lat1)
    dl = math.radians(lon2 - lon1)
    h = math.sin(dp / 2) ** 2 + math.cos(p1) * math.cos(p2) * math.sin(dl / 2) ** 2
    return 2 * 6371 * math.asin(math.sqrt(h))

paris = (48.8566, 2.3522)
lyon = (45.7640, 4.8357)
print(f"Paris - Lyon : {distance_km(paris[0], paris[1], lyon[0], lyon[1]):.0f} km")
```

Coup de pouce : Définir `marseille = (43.2965, 5.3698)`, calculer `d`, puis l'écart en pourcentage : `(775 - d) / d * 100`.

Corrigé rédigé

```
import math

def distance_km(lat1, lon1, lat2, lon2):
    p1 = math.radians(lat1)
    p2 = math.radians(lat2)
    dp = math.radians(lat2 - lat1)
    dl = math.radians(lon2 - lon1)
    h = math.sin(dp / 2) ** 2 + math.cos(p1) * math.cos(p2) * math.sin(dl / 2) ** 2
    return 2 * 6371 * math.asin(math.sqrt(h))

paris = (48.8566, 2.3522)
lyon = (45.7640, 4.8357)
marseille = (43.2965, 5.3698)
print(f"Paris - Lyon : {distance_km(paris[0], paris[1], lyon[0], lyon[1]):.0f} km")
d = distance_km(paris[0], paris[1], marseille[0], marseille[1])
print(f"Paris - Marseille : {d:.0f} km")
print(f"La route est plus longue de {(775 - d) / d * 100:.0f} %")
```

Atelier 4 – Longueur d'un itinéraire

dictionnaires imbriqués, boucles

Objectif : Calculer la durée d'un itinéraire dans le graphe du cours, puis comparer plusieurs itinéraires pour trouver le meilleur.

1. `routes["A"]["B"]` donne la durée du tronçon A–B. Exécuter le programme.
2. Écrire une fonction `longueur(chemin)` qui additionne les durées des tronçons successifs d'une liste de sommets comme `["A", "B", "D"]`.
3. Calculer la longueur de quatre itinéraires de A à F et afficher le plus court.
4. Que se passe-t-il si l'itinéraire contient un tronçon qui n'existe pas (par exemple A–D) ? Faire renvoyer `None` à la fonction dans ce cas.

Programme de départ :

```
routes = {
    "A": {"B": 7, "C": 9},
    "B": {"A": 7, "C": 3, "D": 7},
    "C": {"A": 9, "B": 3, "D": 6, "E": 11},
    "D": {"B": 7, "C": 6, "E": 2, "F": 9},
    "E": {"C": 11, "D": 2, "F": 4},
    "F": {"D": 9, "E": 4},
}

print("A-B :", routes["A"]["B"], "min")
print("Voisins de C :", routes["C"])

def longueur(chemin):
    total = 0
    # a completer
    return total

print(longueur(["A", "B", "D"]))
```

Coup de pouce : Parcourir les indices `for i in range(len(chemin) - 1):` et ajouter `routes[chemin[i]][chemin[i + 1]]`. Pour le tronçon inexistant : `if chemin[i + 1] not in routes[chemin[i]]: return None`.

Corrigé rédigé

```
routes = {
    "A": {"B": 7, "C": 9},
    "B": {"A": 7, "C": 3, "D": 7},
    "C": {"A": 9, "B": 3, "D": 6, "E": 11},
    "D": {"B": 7, "C": 6, "E": 2, "F": 9},
    "E": {"C": 11, "D": 2, "F": 4},
    "F": {"D": 9, "E": 4},
}

def longueur(chemin):
    total = 0
    for i in range(len(chemin) - 1):
        if chemin[i + 1] not in routes[chemin[i]]:
            return None
        total = total + routes[chemin[i]][chemin[i + 1]]
    return total

itineraires = [["A", "B", "D", "F"], ["A", "C", "E", "F"], ["A", "B", "D", "E", "F"], ["A
    ", "C", "D", "E", "F"]]
meilleur = itineraires[0]
```

```
for chemin in itineraires:
    print(" -> ".join(chemin), ":", longueur(chemin), "min")
    if longueur(chemin) < longueur(meilleur):
        meilleur = chemin
print("Le plus court :", " -> ".join(meilleur))
print(longueur(["A", "D"]))
```

Atelier 5 – Programmer l'algorithme de Dijkstra

while, dictionnaires, algorithme

Objectif : Programmer l'algorithme de Dijkstra pour obtenir la durée minimale depuis A vers tous les carrefours.

1. Lire le programme : `dist` contient les distances provisoires, `fixes` la liste des sommets fixés.
2. Exécuter : pour l'instant, le programme fixe les sommets dans l'ordre alphabétique. Améliorer la condition pour choisir le sommet non fixé de plus petite distance provisoire.
3. Compléter la mise à jour des voisins : si `dist[u] + duree < dist[v]`, on remplace `dist[v]`.
4. Comparer les résultats avec l'animation, puis calculer les distances depuis F.

Programme de départ :

```
routes = {
    "A": {"B": 7, "C": 9},
    "B": {"A": 7, "C": 3, "D": 7},
    "C": {"A": 9, "B": 3, "D": 6, "E": 11},
    "D": {"B": 7, "C": 6, "E": 2, "F": 9},
    "E": {"C": 11, "D": 2, "F": 4},
    "F": {"D": 9, "E": 4},
}

infini = 1000000
dist = {}
for s in routes:
    dist[s] = infini
dist["A"] = 0
fixes = []

while len(fixes) < len(routes):
    u = None
    for s in routes:
        if s not in fixes and u is None:
            u = s   # a ameliorer : choisir le sommet non fixe le plus proche
    fixes.append(u)
    for v in routes[u]:
        duree = routes[u][v]
        # a completer : mettre a jour dist[v]
    print("On fixe", u, "a la distance", dist[u])
```

Coup de pouce : Pour choisir `u` : `if s not in fixes and (u is None or dist[s] < dist[u]): u = s`. Pour la mise à jour : `if dist[u] + duree < dist[v]: dist[v] = dist[u] + duree`.

Corrigé rédigé

```
routes = {
    "A": {"B": 7, "C": 9},
    "B": {"A": 7, "C": 3, "D": 7},
    "C": {"A": 9, "B": 3, "D": 6, "E": 11},
    "D": {"B": 7, "C": 6, "E": 2, "F": 9},
```

```
    "E": {"C": 11, "D": 2, "F": 4},
    "F": {"D": 9, "E": 4},
}

def dijkstra(depart):
    infini = 1000000
    dist = {}
    for s in routes:
        dist[s] = infini
    dist[depart] = 0
    fixes = []
    while len(fixes) < len(routes):
        u = None
        for s in routes:
            if s not in fixes and (u is None or dist[s] < dist[u]):
                u = s
        fixes.append(u)
        for v in routes[u]:
            duree = routes[u][v]
            if dist[u] + duree < dist[v]:
                dist[v] = dist[u] + duree
    return dist

print("Depuis A :", dijkstra("A"))
print("Depuis F :", dijkstra("F"))
```

6.8 Exercices corrigés

Exercice

Exercice 1 – Principe du GPS. 1. Pourquoi faut-il au moins 4 satellites pour se localiser ?
2. Le signal d'un satellite met 0,068 s pour arriver au récepteur. À quelle distance est le satellite ?
3. Vrai ou faux : « le satellite GPS sait où se trouve mon téléphone ».

Corrigé rédigé

1. Trois distances suffisent à déterminer la position dans l'espace ; la quatrième sert à corriger l'erreur de l'horloge du récepteur, beaucoup moins précise que les horloges atomiques des satellites.
2. d = 300 000 × 0,068 = **20 400 km**.
3. Faux : le récepteur ne fait que recevoir des signaux, il n'émet rien vers les satellites.

Exercice

Exercice 2 – Décoder une trame. Décoder la trame `$GPGGA,092751.00,4545.6420,N,00450.1980,E,1,10,0.8,172.4,M,49.2,M,,*63` : heure, latitude et longitude en degrés décimaux (au dix-millième), altitude, nombre de satellites. Dans quelle grande ville française se trouve le récepteur ?

Corrigé rédigé

Heure : 09 h 27 min 51 s (UTC). Latitude : 45 + 45,642 ÷ 60 ≈ **45,7607° N**. Longitude : 4 + 50,198 ÷ 60 ≈ **4,8366° E**. Altitude 172,4 m ; 10 satellites. Le récepteur est à **Lyon**.

```python
def en_degres(valeur, nb_chiffres_degres):
    degres = int(valeur[:nb_chiffres_degres])
    minutes = float(valeur[nb_chiffres_degres:])
    return degres + minutes / 60

champs = "$GPGGA,092751.00,4545.6420,N,00450.1980,E,1,10,0.8,172.4,M,49.2,M,,*63".split(",")
print(round(en_degres(champs[2], 2), 4), round(en_degres(champs[4], 3), 4))
```

Exercice

Exercice 3 – Degrés et minutes. 1. Convertir 43° 17,864′ en degrés décimaux.
2. Inversement, convertir 2,3522° en degrés et minutes.
3. Sur Terre, 1′ de latitude correspond à environ 1,852 km (un mille marin). Quelle distance correspond à 0,0001′ ?

Corrigé rédigé

1. $43 + 17{,}864 \div 60 \approx$ **43,2977°**.
2. $0{,}3522 \times 60 = 21{,}132$: **2° 21,132′**.
3. $0{,}0001 \times 1\,852$ m $\approx$ **0,19 m** : les trames NMEA donnent la position à une vingtaine de centimètres près (la précision réelle est de quelques mètres).

Exercice

Exercice 4 – Plus court chemin à la main. Dans le graphe du cours, appliquer l'algorithme de Dijkstra à la main en partant de F, et donner le plus court chemin de F à A et sa longueur.

Corrigé rédigé

On fixe F (0), E (4), D (6), C (12, par D), B (13, par D), puis A (20, par B). Plus court chemin : **F** $\to$ **E** $\to$ **D** $\to$ **B** $\to$ **A**, de longueur 20. C'est le chemin du cours parcouru à l'envers.

Exercice

Exercice 5 – Un tronçon fermé. Des travaux ferment le tronçon D–E. Dans le programme de l'atelier 5, supprimer ce tronçon (des deux côtés) et calculer la nouvelle durée minimale de A à F. Quel est le nouvel itinéraire ?

Corrigé rédigé

Sans D–E, il reste A $\to$ B $\to$ D $\to$ F ($7 + 7 + 9 =$ **23 min**) ; A $\to$ C $\to$ E $\to$ F donne 24 min. Le nouvel itinéraire est A $\to$ B $\to$ D $\to$ F.

```python
routes = {
    "A": {"B": 7, "C": 9},
    "B": {"A": 7, "C": 3, "D": 7},
    "C": {"A": 9, "B": 3, "D": 6, "E": 11},
    "D": {"B": 7, "C": 6, "E": 2, "F": 9},
    "E": {"C": 11, "D": 2, "F": 4},
    "F": {"D": 9, "E": 4},
}

del routes["D"]["E"]
```

```python
del routes["E"]["D"]

def dijkstra(depart):
    dist = {s: 1000000 for s in routes}
    dist[depart] = 0
    fixes = []
    while len(fixes) < len(routes):
        u = None
        for s in routes:
            if s not in fixes and (u is None or dist[s] < dist[u]):
                u = s
        fixes.append(u)
        for v in routes[u]:
            if dist[u] + routes[u][v] < dist[v]:
                dist[v] = dist[u] + routes[u][v]
    return dist

print(dijkstra("A"))
```

Exercice

Exercice 6 – Partager sa position. Pour chaque application, proposer un réglage de localisation adapté (jamais, pendant l'utilisation, toujours ; précise ou approximative) : météo ; navigation GPS ; jeu de cartes ; appareil photo ; application de partage de trajet avec ses parents.

Corrigé rédigé

Météo : pendant l'utilisation, approximative. Navigation : pendant l'utilisation, précise. Jeu de cartes : jamais. Appareil photo : jamais (sauf si l'on veut garder le lieu des photos, en sachant qu'il sera dans les métadonnées). Partage de trajet : pendant le trajet seulement, précise, avec des proches de confiance.

6.9 QCM – Teste tes connaissances

Pour chaque question, une seule réponse est correcte.

Q1. Galileo est…

a) le système de positionnement par satellites européen
b) un logiciel de cartographie
c) un format de trame
d) un algorithme de plus court chemin

Q2. Un récepteur GPS calcule sa distance à un satellite grâce…

a) à la puissance du signal
b) à la durée de trajet du signal
c) à un message envoyé au satellite
d) à une antenne relais 4G

Q3. Combien de satellites faut-il au minimum pour une position fiable ?

a) 1
b) 2
c) 3
d) 4

Q4. La latitude d'un point indique…

a) sa position au nord ou au sud de l'équateur
b) sa position à l'est ou à l'ouest de Greenwich
c) son altitude
d) son fuseau horaire

Q5. Dans une trame $GPGGA, 4851.5040,N signifie…

a) 48,515040° N
b) 48° 51,5040′ N
c) 4 851 km au nord
d) 48° 51′ 50,40″ N

Q6. 48° 30′ en degrés décimaux, c'est…

a) 48,30°
b) 48,5°
c) 48,03°
d) 4,83°

Q7. Que produit ce programme ?

```
1 v = "00217.6700"
2 print(int(v[:3]))
```

a) 002
b) 2
c) 217
d) 00217

Q8. OpenStreetMap est…

a) une carte collaborative et libre
b) le site de l'IGN
c) un satellite
d) une norme de trames GPS

Q9. Dans un graphe pondéré pour un itinéraire, le poids d'une arête peut représenter…

a) le nombre de carrefours
b) la durée du trajet sur ce tronçon
c) la latitude
d) le nombre de satellites

Q10. Dans le graphe du cours, la longueur du chemin A → C → E → F est…

a) 20
b) 22
c) 24
d) 26

Q11. L'algorithme de Dijkstra fixe à chaque étape…

a) le sommet le plus éloigné
b) le sommet non fixé de plus petite distance provisoire
c) un sommet au hasard
d) le sommet d'arrivée

Q12. Pour une application météo, le réglage de localisation le plus adapté est…

a) toujours, précise
b) pendant l'utilisation, approximative
c) toujours, approximative
d) pendant l'utilisation, précise

Corrigé rédigé

Réponses du QCM.
Q1. a) GPS (États-Unis), Galileo (Europe), GLONASS (Russie), BeiDou (Chine).
Q2. b) $d = c \times \Delta t$, avec c la vitesse de la lumière.
Q3. d) Trois pour la position, un quatrième pour corriger l'horloge du récepteur.
Q4. a) La longitude se mesure par rapport au méridien de Greenwich.
Q5. b) Les deux premiers chiffres sont les degrés, la suite les minutes.
Q6. b) $30' = 30 \div 60 = 0{,}5°$.
Q7. b) v[:3] vaut "002" et int("002") vaut 2.
Q8. a) Elle est construite par des millions de contributeurs bénévoles.
Q9. b) Le poids peut être une distance, une durée ou un coût.
Q10. c) $9 + 11 + 4 = 24$; le plus court chemin A → B → D → E → F mesure 20.
Q11. b) C'est ce choix qui garantit que la distance fixée est bien minimale.
Q12. b) La météo n'a besoin que de la ville, et seulement quand on consulte l'application.

6.10 Fiche de synthèse

Fiche de synthèse

— **Géolocalisation :** Satellites (GPS, Galileo) à environ 20 000 km. Distance = c × durée du trajet du signal ; au moins 4 satellites. Le récepteur ne fait que recevoir.

— **Coordonnées :** Latitude (N/S de l'équateur) et longitude (E/W de Greenwich). $1° = 60'$; degrés décimaux = degrés + minutes ÷ 60.

— **Cartes :** Couches d'information (Géoportail), échelle ; OpenStreetMap, carte collaborative et libre.

— **NMEA 0183 :** Trame $GPGGA : heure, latitude (ddmm.mmmm), N/S, longitude (dddmm.mmmm), E/W, qualité, satellites, altitude.

— **Itinéraires :** Graphe pondéré ; longueur d'un chemin = somme des poids. Dijkstra fixe à chaque étape le sommet le plus proche non fixé.

— **Confidentialité :** La position est une donnée sensible : autoriser au cas par cas, « pendant l'utilisation », position approximative, pas d'historique.