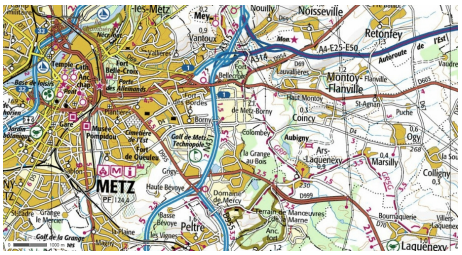
TL ASD Metz
Calcul d'itinéraires pour le transport routier



Objectifs

Imaginez que vous voulez développer un logiciel de cartographie destiné au marché des transporteurs routiers. Par rapport aux logiciels de cartographie « grand public » (Mappy, Google Map, etc), vous voulez offrir des fonctions adaptées spécifiquement à l'activité de vos clients, comme la possibilité pour le transporteur d'optimiser les tournées de ses camions selon un critère de coût (coût, délai de livraison, etc). Pour cela, il vous faudra résoudre successivement quatre problèmes :

1. Partie 1 : une carte routière représente essentiellement le réseau routier connectant les villes par des routes. La formalisation informatique d'un tel réseau est un *graphe*, c'est-à-dire un ensemble de sommets reliés par des arêtes. Il vous faudra donc implémenter une structure de données capable de représenter ce graphe en mémoire. Il vous faudra aussi un outil pour tracer ce graphe à l'écran.
2. Partie 2 : pour pouvoir tester l'efficacité des algorithmes de votre logiciel, il vous faudra disposer de cartes routières réalistes. On s'intéressera par ce biais aux *graphes de voisinage* et à leurs algorithmes de construction.
3. Partie 3 : une fonctionnalité de base de votre logiciel est de calculer l'itinéraire le plus court pour aller d'un point A à un point B. Pour ce faire vous implémenterez l'*algorithme de Dijkstra* vu en cours.
4. Partie 4 : enfin on s'intéressera au problème d'optimisation d'une tournée de camions. Ce problème s'apparente au problème du *voyageur de commerce* réputé NP-complet. On cherchera à le résoudre de

façon exacte à l'aide d'une approche de type backtracking avant d'en chercher des solutions approchées plus rapides à calculer.

Déroulement

Ce sujet fait l'objet des quatre séances de TL programmées à l'emploi du temps. Il se décompose en autant de parties consécutives. Chaque partie comporte une série de questions dites *prioritaires* qu'il faut traiter pour pouvoir passer à la partie suivante. Afin de tenir compte de vos rythmes de progression respectifs, certaines parties comportent également une série de questions *d'approfondissement*. Il vous est demandé de traiter d'abord les questions prioritaires associées à la séance en cours (ou aux séances précédentes si vous avez du retard). Une fois que vous les aurez traitées, vous aborderez selon vos intérêts les questions d'approfondissement de la séance en cours ou des séances précédentes. Vous ne devez pas prendre d'avance en abordant les questions des séances suivantes à moins d'avoir traité toutes les questions prioritaires et d'approfondissement des séances passées.

Compte-rendu

À l'issue de la dernière séance, vous disposerez d'un délai de cinq jours ouvrés pour remettre votre compte-rendu sur `http://moodle.supelec.fr`. **Attention à respecter les délais, le dépot Moodle n'acceptant pas les retards**. Ce compte-rendu sera un fichier archive (format `.zip` ou `.tar.gz` contenant :

- Le code source **commenté** (l'ensemble des fichiers `.py` que vous avez écrits)
- Un rapport au format pdf rédigé à l'aide de Latex ou MS Word. Ce document de quelques pages ne doit pas contenir de code source mais apporter des compléments d'information : il doit traiter des questions théoriques que vous avez pu aborder, commenter utilement les questions d'implémentation (considération sur la complexité, etc), mettre en évidence les résultats que vous obtenez (capture d'écran, tracé de courbe, etc) et faire une synthèse globale de votre projet.

Conseils

- Prenez le temps de lire les rubriques *indications de mise en œuvre* qui suivent la plupart des questions avant de vouloir y répondre.
- Les questions purement théoriques sont repérées par le symbole $\star$. Elles sont destinées à la compréhension générale du sujet mais il n'est pas nécessaire de les traiter en séance pour avancer dans votre réalisation. Il est conseillé de survoler ces questions en séance pour vous concentrez sur les questions pratiques qui nécessitent l'écriture de

code. Vous pourrez traiter ces questions théoriques lors de la rédaction du compte-rendu.

— Pour prendre de bonnes habitudes dès le début, vous vous efforcerez à écrire un code clair, commenté et dans la mesure du possible élégant. Voici quelques conseils :
  — Simplifiez dans votre code les expressions alambiquées. Prenez le temps d'écrire un code élégant qui sera plus court, moins sujet aux erreurs et bien souvent plus rapide. Par exemple si la condition d'itération d'une boucle parait compliquée, il est sûrement possible d'en simplifier l'expression. Pensez à utiliser les instructions `return` (qui sort de la fonction en plus de renvoyer la valeur de retour) et `break` (qui sort de la boucle) plutôt que d'introduire des variables booléennes pour gérer les conditions de sortie de boucle. Si à la sortie d'une boucle vous devez faire un test pour savoir de quelle manière vous êtes sorti de la boucle, déplacez les instructions conditionnées au test au niveau du point de sortie de la boucle, par exemple au niveau d'une instruction `break`. Rappelez vous aussi que les conditions de fin de boucle acceptent toute expression qui s'évalue en un booléen. Il ne sert à rien de comparer des booléens avec des constantes : `a == True` et `a == False` sont à remplacer par `a` et `not a`.
  — Pour surmonter les difficultés techniques, utilisez la documentation à votre disposition, à savoir :
    — Les annexes de ce sujet qui développent des points précis nécessaires au traitement de certaines questions. L'utilité d'une annexe est signalée au moment opportun.
    — Les supports de formation à Python de l'école que sont liesse et cours-python.
    — L'aide-mémoire Python memento permet de se remémorer rapidement la syntaxe des différents aspects du langage.
    — La documentation officielle à l'adresse python.org.
  — Donnez aux classes, fonctions et attributs des noms explicites. Pas de variable *truc*. Évitez les variables i ou j sauf quand cela ne prête pas à confusion (comme les indices de boucles).
  — Imposez-vous des conventions syntaxiques standard : commencent par une minuscule les noms d'attributs, méthodes, variables. Ex : « maMethode() », « monAttribut ». Commencent par une majuscule les noms de classes. Ex : « MaClasse »
  — Ajoutez des commentaires et documentez votre code quand c'est utile. Une ligne de commentaire commence par `#`. La documentation d'une classe ou d'une fonction est une chaîne de caractères située juste après la déclaration.

TL 1 : implémentation d'une structure de données de type graphe

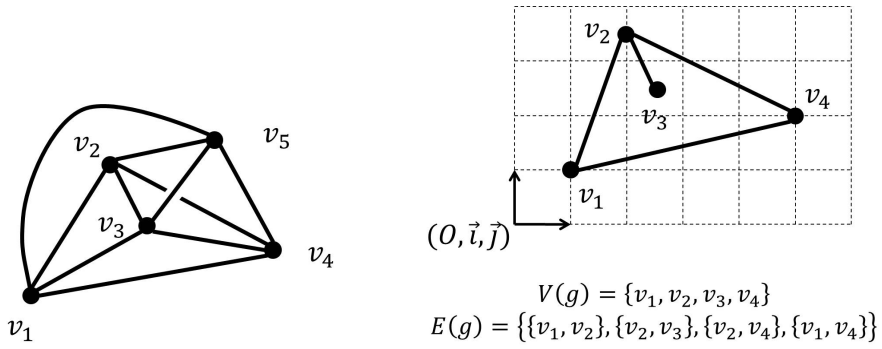
Questions prioritaires

Pour simplifier le problème, on supposera que les routes présentent des conditions de circulation identiques dans les deux sens : il n'existe pas de route à sens unique, il n'y a pas de dénivelé important ni de limitations de vitesse asymétriques, etc. La représentation abstraite d'une telle carte routière s'appelle un *graphe non orienté*. Un *graphe non orienté* est défini comme un couple $g = (V, E)$ d'un ensemble V de *sommets* (V pour vertex en anglais) et un ensemble E d'*arêtes* (E pour edge en anglais) constituées chacune de la paire de sommets qu'elles relient. Un graphe se dessine aisément en représentant les sommets par des points et les arêtes par des traits comme illustré sur la figure 1(a). Un graphe pour lequel il existe une représentation graphique dans le plan tel que ses arêtes ne se coupent pas (i.e ailleurs qu'en ses sommets) est un *graphe planaire*. Une représentation possible (i.e. un plongement dans le plan) d'un graphe planaire est appelée une *carte planaire* dont un exemple est le graphe de la figure 1(b). Selon le contexte, les sommets et les arêtes peuvent être associés à des informations supplémentaires. On parle alors de *graphe étiqueté* : chaque attribut d'une arête ou d'un sommet est une *étiquette* ou propriété.

Une carte routière est donc formellement une carte planaire dans laquelle les sommets et les arêtes représentent respectivement les carrefours et les routes. Chaque sommet est étiqueté par sa position (x, y) dans le plan (i.e sa longitude et latitude) et chaque arête est étiquetée par la longueur et la moyenne des limitations de vitesse du tronçon de route que l'arête représente. Cette liste n'est pas restrictive et vous pouvez ajouter d'autres propriétés aux sommets (altitude, nom du lieu-dit, etc) et aux arêtes (dénivelé cumulé, frais de péage, etc). Enfin chaque graphe portera un nom afin d'identifier plus facilement différentes cartes.

Une bonne pratique en génie logiciel est de commencer par concevoir l'interface de programmation de vos classes d'objets pour en vérifier la cohérence et la complétude, et ce avant même de les implémenter. Ainsi pour construire une carte et la parcourir, il faut introduire trois classes `Graphe`, `Sommet` et `Arete` et disposer au minimum des opérations suivantes :

- Le constructeur de la classe `Graphe` permet de créer un graphe vide.
- Le constructeur de `Sommet` permet de créer un sommet affecté des propriétés que vous aurez retenues pour les sommets.
- Le constructeur de `Arete` permet de créer une arête affectée des propriétés que vous aurez retenues pour les arêtes.
- La méthode `ajouteSommet` de la classe `Graphe` permet d'ajouter un sommet à un graphe.
- La méthode `connecte` de `Graphe` permet de connecter deux sommets



(a) Un graphe non planaire (b) Une carte planaire

FIGURE 1 – Exemples de graphe et de carte planaire.

du graphe par une nouvelle arête.
- La méthode `renomme` de `Graphe` permet de donner un nom au graphe (i.e. une chaîne de caractères de type `str`).
- La méthode `n` de `Graphe` qui renvoie le nombre de sommets du graphe.
- La méthode `m` de `Graphe` qui renvoie le nombre d'arêtes du graphe.
- La méthode `voisin` de la classe `Arete` prend en argument un sommet s appartenant à l'arête courante `self` et renvoie la référence sur le sommet voisin s' relié à s par `self`.

Cette liste n'est pas exhaustive. Libre à vous de la compléter lorsque de nouveaux besoins se font sentir.

Mise en route : Démarrez votre machine sous Ubuntu. Une fois connecté sur votre compte, créez un répertoire, par exemple `TL-ASD` (reprendre le sujet du TD1 si nécessaire). Extrayez le contenu du fichier archive `code-tl-asd-metz.tar.gz` téléchargeable sur Moodle dans ce répertoire. Ouvrez un terminal depuis ce répertoire (en cliquant le bouton droit de la souris dans la fenêtre du répertoire) depuis lequel vous exécuterez le script Python3 `test.py` (qui contiendra les différentes fonctions de test qui vous seront demandées) à l'aide de la commande `python3 test.py`. Vous pouvez ouvrir un second terminal dans lequel vous testerez d'éventuelles commandes Python3, à l'aide de la commande `ipython3`. Enfin vous éditerez les différents fichiers à l'aide des éditeurs emacs ou gedit à l'aide des commandes `emacs <nom du fichier>.py &` ou `gedit <nom du fichier>.py &`.

Question 1.1. Créez ensuite un fichier `graphe.py` (par exemple à l'aide de la commande `emacs graphe.py &`) dans lequel vous définirez les trois classes `Graphe`, `Sommet` et `Arete` (attention aux minuscules/majuscules de façon à bien distinguer le fichier de la classe). Dans chaque classe déclarez les différentes méthodes identifiées sans chercher à les définir (i.e sans coder

leur contenu).

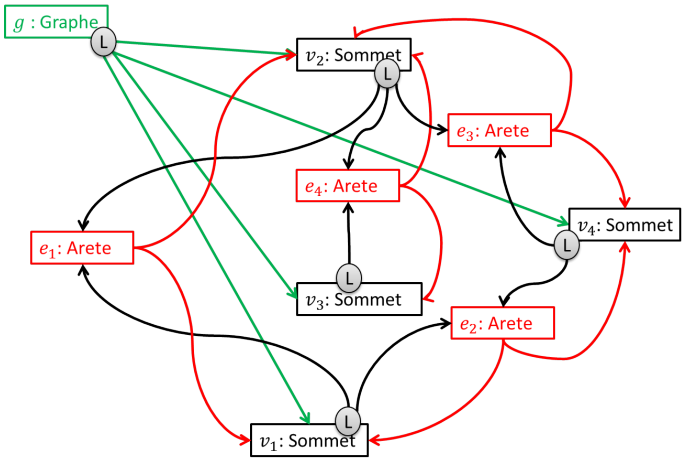
Pour implémenter une représentation d'un graphe, il existe essentiellement deux structures de données : celle fondée sur la *matrice d'adjacence* et celle fondée sur la notion de *listes d'incidence/d'adjacence*.

— Celle fondée sur la *matrice d'adjacence* consiste à représenter la structure du graphe par une matrice (i.e un tableau à deux dimensions) $n \times n$ où n est le nombre de sommets. Chaque sommet est indexé par un entier de 0 à $n-1$ de sorte que la case (i, j) de la matrice contient une référence sur l'arête, si elle existe, entre les sommets i et j.

— Celle fondée sur les *listes d'incidence* consiste à inclure dans l'objet de type `Sommet` décrivant chaque sommet s, une liste dont les éléments sont des références sur les arêtes incidentes à s (de type `Arete`). Un objet de type `Arete` contient deux références `s1` et `s2` sur les sommets que l'arête relie. Enfin l'objet de type `Graphe` contient la liste des références sur les sommets du graphe (de type `Sommet`). Cette représentation est illustrée sur la figure 2.

$g = (v_1, v_2, v_3, v_4)$
$e_1 = (v_1, v_2)$
$e_2 = (v_1, v_4)$
$e_3 = (v_2, v_4)$
$e_4 = (v_2, v_3)$
$L_{v_1} = (e_1, e_2)$
$L_{v_2} = (e_1, e_3, e_4)$
$L_{v_3} = (e_4)$
$L_{v_4} = (e_2, e_3)$

(a) Représentation formelle



(b) Représentation informatique

FIGURE 2 – Représentation par *listes d'incidence* du graphe de la figure 1(b). Sur la représentation informatique, un rectangle a : A représente un objet de la classe A identifié sous le nom a. Une flèche d'un objet a : A vers un objet b : B traduit la présence d'un attribut de la classe A qui contient l'adresse d'un objet b de type B. L'objet a a donc par ce biais accès aux attributs et aux méthodes de l'objet b. Lorsque les flèches sont issues d'un cercle Ⓛ, l'attribut est un tableau dynamique (c'est-à-dire une liste Python `[]`) afin de contenir non pas une mais un nombre arbitraire de références.

Une analyse comparative des complexités de l'une et l'autre solution montre que dans notre cas, il est plus intéressant d'utiliser les listes d'incidence.

Question 1.2. Implémentez les méthodes des classes `Graphe`, `Sommet` et `Arete` en utilisant le principe des listes d'incidence selon les indications suivantes.

Indications de mise en œuvre :
On utilisera des tableaux dynamiques (i.e des listes Python) pour implémenter les listes d'incidence et la liste de sommets d'un graphe. On pourra (si besoin) ajouter comme attribut de la classe `Graphe` la liste des arêtes en plus de la liste de sommets. On testera que la création d'un graphe ne produit pas d'erreur en exécutant le fichier `test.py` à l'aide de la commande `python3 test.py`. La fonction de test unitaire `testQuestion1_2()` qui construit le graphe de la figure 1 sera alors appelée, comme en témoigne le code source de `test.py` donné ci-dessous.

```
# coding: latin-1

import graphe

def creerGrapheFigure1():
    '''Crée le graphe de la figure 1'''
    g = graphe.Graphe("Graphe de la figure 1")
    s1 = g.ajouteSommet(1.0, 1.0)
    s2 = g.ajouteSommet(2.0, 3.5)
    s3 = g.ajouteSommet(2.5, 2.5)
    s4 = g.ajouteSommet(5.0, 2.0)
    g.connecte(s1, s2, 4.0, 90.)
    g.connecte(s1, s4, 5.2, 124.)
    g.connecte(s2, s3, 2.0, 54.)
    g.connecte(s2, s4, 5.0, 90.)
    return g

def testQuestion1_2():
    '''Teste que la création d'un graphe ne plante pas '''
    print "Question 1.2 :"
    creerGrapheFigure1()
    print "Ok. Pas de plantage"

def testQuestion1_3():
    ''' Teste l'affichage d'un graphe dans la console '''
    print "Question 1.3 :"
    g = creerGrapheFigure1()
    print(g)
```

```
if __name__ == "__main__":
    testQuestion1_2()
    testQuestion1_3()
```

Dans la suite, il est important d'être méthodique et d'adopter une méthode de développement incrémentale (i.e ne pas tester à la fin mais au fur et à mesure des développements). Pour ce faire vous distinguerez clairement le code constituant votre logiciel que vous placerez dans le fichier ***graphe.py****, du code regroupant les tests unitaires destinés à vérifier que votre logiciel fonctionne comme attendu, que vous placerez dans le fichier* ***test.py****. Il est impératif qu'à chaque fois qu'une question donne lieu à une nouvelle implémentation dans* ***graphe.py****, vous ajoutiez un test correspondant dans* ***test.py****, en y ajoutant une nouvelle méthode* ***testQuestionXXX()****.* Le contenu des deux premières méthodes de test vous est donné mais c'est à vous d'imaginer les tests pertinents pour les questions ultérieures. Le test réalisé par `testQuestion1_2()` consistera donc simplement à créer le graphe de la figure 1(b) sans planter l'exécution. Toutefois il ne permettra pas de vérifier à ce stade que le graphe créé est celui escompté. La question suivante permettra de faire cette vérification :

Question 1.3. Ajoutez aux classes `Graphe`, `Sommet` et `Arete` une méthode `__str__(self)` qui ne prend pas d'arguments (hormis l'objet courant `self`) et qui renvoie une chaîne de caractères décrivant respectivement un graphe, un sommet et une arete.

La chaîne décrivant un graphe devra ressembler au texte ci-dessous dans le cas du graphe de la figure 1(b) :

```
V(Graphe de la figure 1) = {
  v1 (x = 1.0 km y = 1.0 km)
  v2 (x = 2.0 km y = 3.5 km)
  v3 (x = 2.5 km y = 2.5 km)
  v4 (x = 5.0 km y = 2.0 km)
}
E(Graphe de la figure 1) = {
  {v1, v2} (long. = 4.0 km vlim. = 90.0 km/h)
  {v1, v4} (long. = 5.2 km vlim. = 124.0 km/h)
  {v2, v3} (long. = 2.0 km vlim. = 54.0 km/h)
  {v2, v4} (long. = 5.0 km vlim. = 90.0 km/h)
}
```

Indications de mise en œuvre :
Pour parcourir le contenu d'une liste (et plus généralement de toute structure de données fournie par Python), il est fortement recommander d'utiliser la syntaxe suivante (plus rapide car s'appuyant sur la notion d'itérateur et non sur les accès à partir d'un indice) :

```
L = [4, 2, 9]
# Ecrivez :
for x in L:
        print(x)
# Plutôt que :
for i in range(len(L)):
        print(L[i])
```

Pour numéroter les sommets et les nommer sous la forme `v1`, `v2`, etc, on pourra ajouter un attribut `indice` de type entier à la classe `Sommet`. La façon la plus simple de numéroter les sommets consiste à renuméroter les sommets à partir de 1 à chaque nouvel appel de `__str__`. Une solution plus économe en temps de calcul consiste à affecter un nouveau numéro à un sommet lors de sa création. Il faut alors passer au constructeur `__init__` de la classe `Sommet` son numéro égal à un plus le nombre de sommets déjà créé, qu'on peut obtenir comme la longueur du tableau contenant les sommets du graphe (utiliser la fonction `len(tableau)`). Par ailleurs le code suivant contient des constructions syntaxiques sur des chaînes de caractères qui peuvent vous être utiles pour arriver au résultat attendu :

```
a = 1.; b = 3.
s = "On veut calculer " + str(a) + " / " + str(b) + "\n"
        # str() convertit un objet en chaîne de caractère (à l'aide d'un
            appel à la méthode __str__)
        # Le + sert à concaténer des chaines.
        # \n est un caractère spécial qui provoque un retour à la ligne

s += '\tOn a : '
        # Les apostrophes '...' et les guillements "..." sont équivalents.
        # \t est un caractère spécial de tabulation (alignement par
            insertion d'espaces)
        # L'expression s += c équivaut à s = s + c;

s += '{} / {} = {:5.2f}\n'.format(a, b, a / b)
        # La méthode format permet d'injecter des valeurs dans une
            chaîne selon un certain format.
        # Chaque paire d'accollades {...} est remplacée dans l'ordre par
            un des arguments a, b et (a / b).
        # {:5.2f} permet de formater l'écriture du résultat a/b au
            minimum sur 5 caractères avec 2 chiffres après la virgule
```

Enfin pour tester votre méthode, on pourra utiliser le fait que la méthode `__str__` d'un objet o est appelé dès que o est affiché dans la console (à l'aide de `print(o)`) ou dès qu'il est converti explicitement en chaîne de caractères

(i.e. `print("Objet o = " + str(o))`). C'est ce que fait la fonction de test unitaire `testQuestion1_3()`.

Maintenant que l'on dispose d'une base fonctionnelle pour la classe `Graphe`, on veut produire une représentation graphique des graphes similaire à celle de la figure 1(a), où les sommets sont représentés par des petits disques et les arêtes par des traits reliant les sommets extrémités. On vous donne pour cela un fichier `graphique.py` qui permet de dessiner une carte constituée de formes géométriques élémentaires.

Question 1.4. Après avoir pris connaissance de l'interface de programmation du module `graphique` dans l'annexe 5.1, faites en sorte que l'appel à :

```
graphique.affiche(creerGrapheFigure1(), (3.,2.), 100.)
```

fonctionne et donne une représentation proche de la figure 1(a).

Indications de mise en œuvre :
Dans le cas où votre classe `Graphe` ne contient pas la liste des arêtes, l'affichage de ces dernières nécessite de parcourir tous les sommets du graphe, puis pour chaque sommet v_1, considérer chaque sommet v_2 voisin de v_1 avant de tracer un trait entre v_1 et v_2. Comme une arête e est incidente à deux sommets, chaque arête risque alors d'être affichée deux fois. Afin de ne pas ralentir inutilement le tracé, on lancera l'affichage de l'arête e quand on traite le sommet courant v_1 uniquement si v_1 se confond avec $e.s_1$ (puisque v_1 vaudra tantôt l'attribut $e.s_1$ tantôt $e.s_2$). On pourra donner au titre de la fenêtre le nom du graphe affiché.

Pour faciliter le débogage de votre programme et pour mieux mettre en valeur vos futurs résultats, on aimerait pouvoir différencier certains sommets ou arêtes par des couleurs différentes. En particulier on aimerait distinguer les routes nationales (en rouge) des départementales (en jaune).

Question 1.5. Introduisez un nouvel attribut `couleur` dans les classes `Sommet` et `Arete` pour particulariser la couleur de chaque sommet et arête. Modifiez la procédure de tracé du graphe pour tenir compte de ces nouveaux attributs.

Indications de mise en œuvre :
On rappelle que la méthode `changeCouleur` de `Afficheur` attend en argument une couleur sous la forme d'un triplet `(R,G,B)` dont les composantes sont comprises entre 0 et 1 et désignent respectivement la composante rouge, verte et bleue. `(1,1,1)` désigne ainsi le blanc, `(0,0,0)` le noir, `(1,1,0)` le jaune, etc. On pourra tester sur un graphe de quelques sommets et arêtes.

TL 2 : génération d'une carte routière réaliste

On cherche dans cette partie à produire des graphes plans représentatifs de cartes routières dans lesquels les sommets localiseront les villes, croisements d'axes routiers, etc et les arêtes représenteront des tronçons de routes.

Questions prioritaires

La génération naive de graphes aléatoires ne permet pas d'obtenir des cartes routières réalistes car ces graphes ne sont généralement pas *planaires* (i.e. les arêtes se croisent sans s'intersecter en un sommet comme si les routes se croisaient au niveau de ponts sans passage possible d'une route à l'autre) ni *connexes* (i.e. le graphe est constitué de plusieurs blocs non connectés) et même si elles étaient par le plus grand hasard planaires et connexes, elles ne sont pas réalistes : deux villes proches peuvent ne pas être reliées quand au contraire deux villes éloignées peuvent l'être.

En réalité la construction d'un réseau routier suit une logique répondant à deux objectifs antagonistes : d'une part la qualité de service du réseau mesurée par le temps moyen pour aller d'une ville à une autre et d'autre part le coût de construction et d'entretien du réseau. La construction d'une route reliant directement deux villes A et B dépend donc de l'optimisation d'un problème global complexe et dynamique (évolution des populations, de l'économie, contraintes géographiques, etc). On simplifie donc le problème en se donnant pour toute paire de sommets A et B une condition géométrique simple pour décider de l'existence d'une route entre A et B. Cette condition doit être choisie pour exprimer grosso modo le compromis précédent rapidité/coût.

Par exemple, si on note $\mathcal{V} = \{P_i\}_{1 \leq i \leq n}$ le nuage des n points du plan qui représentent les villes et si $d(A, B)$ est la distance à vol d'oiseau séparant les points A et B, on peut considérer que la construction de la route supposée rectiligne reliant deux villes A à B n'est justifiée que s'il n'existe pas de point M le long de cette route qui soit plus proche d'une troisième ville C :

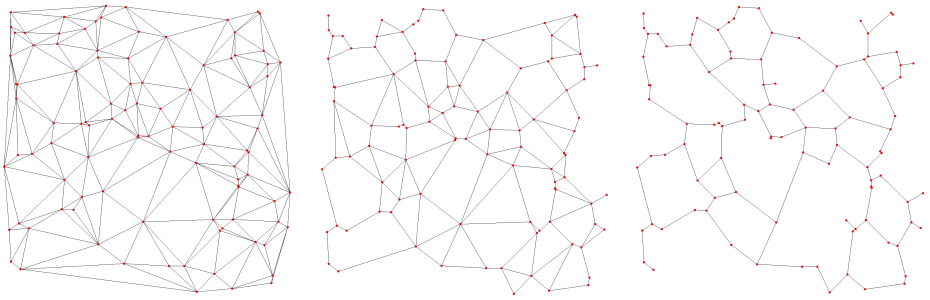
$$\forall C \in \mathcal{V}, \forall M \in [A, B], d(M, C) \geq \min(d(M, A), d(M, B))$$

Le cas échéant on considère qu'un détour par C pour aller de A en B présente un meilleur compromis rapidité/coût.

Question 2.1.⋆ À l'aide d'un dessin, montrez que cette condition revient à ce que pour toute paire A et B de sommets, $\{A, B\}$ forme une arête si et seulement si le disque ouvert de diamètre $[A, B]$ est vide de tout sommet. Un graphe plan vérifiant une telle condition est appelé *graphe de Gabriel*.

Le graphe de Gabriel est un exemple de *graphes de voisinage* pour lesquels les arêtes se définissent en fonction du positionnement relatif de leurs

sommets. Trois familles de graphes de voisinage sont illustrées sur la figure 3 : la *triangulation de Delaunay*, le *graphe de Gabriel* et le *graphe de voisinage relatif*. La construction des arêtes d'un graphe de voisinage suit toujours le



(a) Triangulation de Delaunay $\mathrm{TD}(\mathcal{V})$ (b) Graphe de Gabriel $\mathrm{GG}(\mathcal{V})$ (c) Graphe de voisinage relatif $\mathrm{GVR}(\mathcal{V})$

FIGURE 3 – Les trois graphes de voisinage construits à partir d'un même nuage de points $\mathcal{V}$.

même principe : une arête relie toute paire $\{P_i, P_j\}$ de sommets de $\mathcal{V}$ si et seulement si $\{P_i, P_j\}$ satisfait un certain critère de voisinage. Ce critère est choisi de manière à assurer le caractère planaire et connexe du graphe.

La triangulation de Delaunay [1] Dans ce graphe qu'on note $\mathrm{TD}(\mathcal{V})$, deux sommets P_i et P_j sont voisins (et donc reliés par une arête) si il existe un cercle passant par P_i et P_j tel que le disque fermé soit vide de tout point de $\mathcal{V}$ autre que P_i et P_j (voir figure 4(a)). Les questions d'approfondissement auront pour objet de montrer que ce graphe est planaire, connexe et forme une triangulation : son plongement dans le plan définit des faces qui sont toutes des triangles.

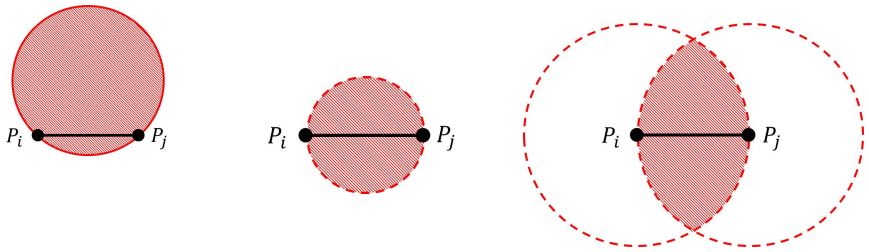
Le graphe de Gabriel $\mathrm{GG}(\mathcal{V})$ Deux sommets P_i et P_j sont voisins s'il n'existe pas de point de $\mathcal{V}$ plus proche du milieu de $\{P_i, P_j\}$ que P_i et P_j. Géométriquement, cela revient à dire que deux points sont reliés par une arête si et seulement si le disque de diamètre $[P_i, P_j]$ représenté sur la figure 4(b) est vide de tout autre point. De façon évidente, la triangulation de Delaunay contient le graphe de Gabriel : on dit que $\mathrm{GG}(\mathcal{V})$ est un *sous-graphe* de $\mathrm{TD}(\mathcal{V})$ et on note $\mathrm{GG}(\mathcal{V}) \subseteq \mathrm{TD}(\mathcal{V})$.

1. Les triangulations de Delaunay sont très importantes en informatique pour représenter la forme d'objets géométriques complexes, par exemple dans le domaine de la conception assistée par ordinateur (CAO) ou des jeux vidéo. Le *diagramme de Voronoï* qui se construit comme le graphe dual de la triangulation de Delaunay est aussi très important : les sommets du diagramme sont définis comme les centres des cercles circonscrits des triangles de Delaunay et ses arêtes sont les segments de médiatrices des arêtes de la triangulation qui joignent les centres des cercles circonscrits. Toute face F du diagramme de Voronoï est alors centrée sur un sommet S de Delaunay telle que F représente l'ensemble des points du plan pour lesquels le sommet de la triangulation le plus proche est S.

Le graphe de voisinage relatif $\mathrm{GVR}(\mathcal{V})$ Dans ce graphe deux sommets P_i et P_j sont voisins s'il n'existe pas d'autre point P_k à la fois plus proche de P_i que ne l'est P_j et plus proche de P_j que ne l'est P_i :

$$\forall k, \max\left(d(P_k, P_i), d(P_k, P_j)\right) \geq d(P_i, P_j)$$

Géométriquement, cela revient à dire que deux points sont reliés par une arête si et seulement si l'intersection des deux disques ouverts centrés sur P_i et P_j et de rayon $d(P_i, P_j)$ est vide de tout autre point (voir figure 4(c)). Le graphe de voisinage relatif est donc un sous-graphe du graphe de Gabriel : $\mathrm{GVR}(\mathcal{V}) \subseteq \mathrm{GG}(\mathcal{V}) \subseteq \mathrm{TD}(\mathcal{V})$ comme en atteste la figure 3.



(a) Triangulation de Delaunay (b) Graphe de Gabriel (c) Graphe de voisinage relatif

FIGURE 4 – Zones d'exclusion pour les trois familles de graphes de voisinage.

Des trois graphes de voisinage, le graphe de Gabriel $\mathrm{GG}(\mathcal{V})$ est le modèle le plus réaliste pour représenter la structure des routes que l'on observe sur les cartes routières. On va chercher à le construire à l'aide d'un algorithme naïf puis d'un second beaucoup plus efficace.

Question 2.2. À l'aide de l'annexe 5.2 qui indique comment effectuer des tirages aléatoires, écrivez une fonction `pointsAleatoires(n, L)` qui retourne un graphe sans arêtes, constitué uniquement de n sommets positionnés aléatoirement dans le rectangle défini par des abscisses et des ordonnées comprises dans l'intervalle $[-L/2, L/2]$. Testez en utilisant l'affichage graphique.

Question 2.3. Pour faciliter l'ajout de route à la carte, ajoutez une méthode `ajouteRoute(self, v1, v2, vmax)` à la classe `Graphe` qui relie les sommets `v1` et `v2` par une route (i.e une arête) dont la longueur sera supposée être la distance à vol d'oiseau et dont la vitesse maximale est `vmax`. Pour ce faire on pourra ajouter une méthode `distance(self, v)` dans la classe `Sommet` qui calcule la distance euclidienne entre le sommet courant `self` et le sommet `v`. Ajoutez ensuite deux méthodes `ajouteNationale(self, v1, v2)` et `ajouteDepartementale(self, v1, v2)` qui construisent entre deux

villes une route nationale (route rouge, 90 km/h de vitesse moyenne limite) et une départementale (route jaune, 60 km/h de vitesse moyenne limite).

Indications de mise en œuvre :
La fonction `Math.sqrt(x)` du module `Math` calcule la racine carrée d'un nombre.

Question 2.4. En utilisant la fonction `pointsAleatoires(n, L)` codez les algorithmes naïfs de construction d'un graphe de Gabriel et d'un graphe de voisinage relatif dans des fonctions `gabriel(g)` et `gvr(g)` qui travailleront sur le graphe g reçu en argument, initialement supposé sans arêtes (i.e. un nuage de points produit par `pointsAleatoires(n, L)`). Testez en visualisant les cartes résultantes et en observant l'inclusion de l'une dans l'autre.

Indications de mise en œuvre :
Le test nécessite de dupliquer un nuage de points g. On pourra utiliser à cet effet la fonction `copy.deepcopy(g)` du module `copy`. Cette fonction renvoie une copie conforme de `g` disposant de ses propres sommets et arêtes (i.e. `deepcopy` duplique non seulement `g` mais aussi tous les objets accessibles depuis `g`, d'où le terme de copie « profonde »).

Question 2.5. Codez cette fois-ci une fonction `reseau(g)` qui construit à partir du nuage de points `g` une carte routière dans laquelle les nationales correspondent aux arêtes de GVR(g) et où les départementales sont les arêtes de GG(g).

Question 2.6. Mesurez le temps de calcul nécessaire à la construction d'un réseau routier pour différentes valeurs de n ; estimez empiriquement la complexité moyenne en fonction de n et comparez avec la théorie.

Indications de mise en œuvre :
L'annexe 5.3 indique comment obtenir le temps de l'horloge et comment afficher le résultat sous forme de graphique.

Questions d'approfondissement

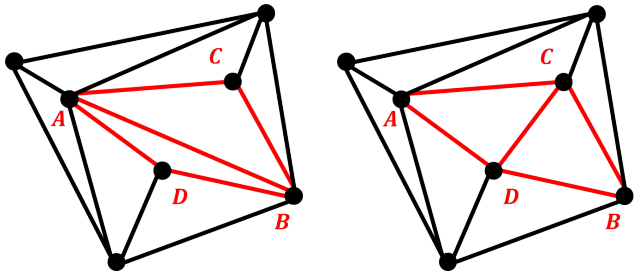
On s'intéresse maintenant à un algorithme de construction plus efficace du réseau routier. Pour cela on remarque que pour un nuage $\mathcal{V}$ de points donné, le graphe de voisinage relatif GVR($\mathcal{V}$) est un sous-graphe du graphe de Gabriel GG($\mathcal{V}$) qui lui-même est un sous-graphe de la triangulation de Delaunay TD($\mathcal{V}$). La construction du réseau routier peut donc se

faire en construisant d'abord la triangulation de Delaunay, puis en éliminant les arêtes de la triangulation qui ne vérifient pas le critère de voisinage du graphe de Gabriel. L'idée est alors de tirer parti des algorithmes de référence [2] qui sont capables de calculer la triangulation de Delaunay d'un nuage de n points avec une complexité moyenne en $O(n \cdot \log(n))$.

Pour faciliter le travail en séance, on admettra dans un premier temps le principe de l'algorithme de filtrage qu'on testera. C'est seulement dans un second temps qu'on s'intéressera à sa justification théorique.

2.1.1 Implémentation

Un algorithme efficace de calcul de la triangulation de Delaunay, dit de « *flipping* », est fourni dans le fichier `triangulation.py`. Celui-ci construit dans un premier temps une triangulation quelconque à partir d'un ensemble de points. Les arêtes de la triangulation sont représentées par une structure de données particulière appelée *quad-edge* qui détient des références non seulement sur ses deux sommets mais aussi sur les deux faces triangulaires adjacentes à l'arête. Les faces triangulaires sont représentées par une autre structure qui détient des références sur les trois sommets et les trois quad-aretes qui lui sont adjacentes. Dans un deuxième temps, l'algo-



(a) Avant basculement (b) Après basculement

FIGURE 5 – Le principe de flipping d'une triangulation.

rithme considère tour à tour chaque quad-edge de sommets $\{A, B\}$ comme la diagonale d'un quadrilatère (A, C, B, D). Si la somme des angles du quadrilatère en C et D dépasse π, le quad-edge est basculé (*flipped* en anglais) et devient la seconde diagonale $\{C, D\}$. L'algorithme s'arrête lorsqu'aucun quad-edge ne peut plus être basculé. On montre dans la partie théorique que cet algorithme termine toujours et produit en sortie la triangulation de Delaunay.

Question 2.7. Ajoutez dans `graphe.py` une fonction `delaunay(g)` qui construit la triangulation de Delaunay d'un nuage de points fourni par le

2. Ces algorithmes sont trop complexes pour être implémentés dans le cadre du TL.

graphe `g`. Utilisez pour ce faire le module `triangulation`. Il faudra éventuellement que vous adaptiez votre code (en renommant et/ou en ajoutant des méthodes) afin d'assurer la compatibilité avec la classe `Triangulation`. Évaluez ensuite la rapidité de l'algorithme pour différentes valeurs de n et comparez avec les résultats obtenus pour la construction naive du graphe de Gabriel.

Indications de mise en œuvre :
Pour créer une triangulation de Delaunay à partir d'un nuage de points `g`, il faut ajouter la fonction suivante :

```
import triangulation

def delaunay(g):
        # Crée une triangulation de Delaunay à partir d'un nuage de
            point
    t = triangulation.Triangulation(g)

    # Définit une fonction destinée à être appelée
    # pour toute paire (s1,s2) de sommets
    # qui forme une arête dans la triangulation de Delaunay.
    # v3 et v4 sont les troisièmes sommets des deux triangles
    # ayant (s1,s2) comme côté.
    def selectionneAretes(graphe, v1, v2, v3, v4):
        # Fait de chaque arête de la triangulation
        # une nationale
        graphe.ajouteNationale(v1, v2)

    # Construit le graphe de retour, égal à la triangulation de Delaunay
    g = t.construitGrapheDeSortie(selectionneAretes)

    g.renomme("Delaunay(" + str(g.n()) + ")");
    return g;
```

On va maintenant ajouter à la classe `Triangulation` une méthode pour extraire rapidement le graphe de Gabriel à partir de la triangulation de Delaunay. Afin d'éliminer rapidement les quad-aretes qui n'appartiennent pas au graphe de Gabriel, on utilisera le résultat établi à la question 15.

Question 2.8. Dupliquez la fonction `delaunay(g)` en une fonction `reseauRapide(g)` et adaptez cette dernière pour construire le réseau routier défini à la question 5. En particulier adaptez la fonction imbriquée `selectionneAretes`.

Testez en vérifiant que pour un même nuage de points, l'algorithme naif et l'algorithme rapide produisent le même réseau routier.

Indications de mise en œuvre :
Pour chaque quad-edge de sommets A et B, on vérifiera que les sommets C et D de ses faces adjacentes vérifient les conditions du graphe de Gabriel et du graphe de voisinage relatif.

Question 2.9. Concluez en comparant les temps de calcul mis par les deux algorithmes de construction du réseau routier.

2.1.2 Analyse théorique

On va démontrer le bien fondé de l'algorithme de filtrage proposé précédemment. Pour ce faire on rappelle le *théorème de l'angle inscrit* illustré par la figure 6 : étant donné deux points A et B distincts et un angle $\theta \in [0, \pi]$, la ligne de niveau $\mathcal{L}(\theta, A, B)$ des points C tels que l'angle $\widehat{ACB}$ en valeur absolu est égal à θ forme la réunion de deux arcs de cercles d'extrémités A et B et symétriques par rapport à la droite (AB). Étant donné un des deux arcs de cercle inscrit dans un des deux demi-plans de frontière (AB), la distance algébrique de son centre O à la droite (AB) vaut $d(\theta) = \frac{AB}{2\tan(\theta)}$. C'est une fonction décroissante de 0 à π. Le cas médian $\mathcal{L}(\frac{\pi}{2}, A, B)$ se confond avec le cercle de diamètre $[A, B]$ au delà duquel le centre O passe dans l'autre demi-plan. Par conséquent l'ensemble des points C tels que l'angle $\widehat{ACB} \geq \theta$ est la surface intérieure circonscrite par $\mathcal{L}(\theta, A, B)$. Enfin on notera qu'un cercle

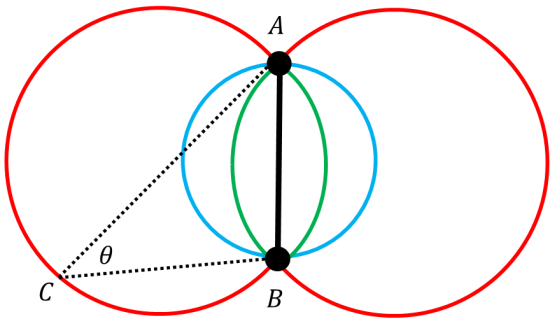


FIGURE 6 – Théorème de l'angle inscrit. Les lignes de niveau $\mathcal{L}(\theta, A, B)$ en rouge, bleu et vert correspondent respectivement à un angle θ inférieur, égal et supérieur à $\frac{\pi}{2}$.

passant par A et B décrit l'union de l'ensemble des points d'un certain angle θ dans un des deux demi-plans délimités par la droite (AB) et l'ensemble des points d'angle $\pi - \theta$ dans l'autre demi-plan. On montre tout d'abord un lemme utile par la suite :

Question 2.10.⋆ Pour toute paire $\{A, B\}$ de points de $\mathcal{V}$, soient $\mathcal{P}$ et $\mathcal{P}'$ les deux demi-plans fermés de frontière la droite (AB) privés des points A et B. Montrez grâce au théorème de l'angle inscrit que $\{A, B\}$ est une arête de la triangulation de Delaunay TD($\mathcal{V}$) si et seulement si :

$$\forall C \in \mathcal{P} \cap \mathcal{V}, \forall D \in \mathcal{P}' \cap \mathcal{V}, \widehat{ACB} + \widehat{ADB} \leq \pi$$

Question 2.11.⋆ Appliquez ce lemme pour montrer que le graphe TD($\mathcal{V}$) est planaire : les arêtes ne peuvent pas s'intersecter autrement qu'en un sommet.

Question 2.12.⋆ Un graphe planaire définit nécessairement des faces qui sont des polygones. Montrez toujours grâce au lemme que le graphe TD($\mathcal{V}$) est une triangulation : toutes ses faces sont des triangles.

Indice : pour toute face polygonale de $n \geq 4$ sommets, on considérera 4 sommets parmi les n possibles formant un quadrilatère $ABCD$. En appliquant le lemme on en déduira une condition absurde.

Question 2.13.⋆ Maintenant qu'on sait que TD($\mathcal{V}$) est une triangulation, montrez toujours grâce au lemme qu'un triplet $\{A, B, C\} \subseteq \mathcal{V}$ forme une face de la triangulation TD($\mathcal{V}$) si et seulement si le disque du cercle circonscrit du triangle ABC est vide de tout point de $\mathcal{V}$ autre que A, B et C.

Ce résultat aboutit à une seconde définition de la triangulation de Delaunay : une triangulation est de Delaunay si tous les cercles circonscrits à ses faces sont vides de tout sommet.

On montre maintenant comment l'algorithme de flipping converge toujours en un nombre fini d'étapes vers la triangulation de Delaunay. Soit une triangulation T($\mathcal{V}$) quelconque de sommets $\mathcal{V}$. Soit $\theta(T)$ la liste des angles des triangles de T triée par ordre croissant. L'ensemble des triangulations possibles sur $\mathcal{V}$ induit un ensemble de listes possibles $\theta(T)$ noté Θ. Cet ensemble de listes peut être ordonné selon l'ordre lexicographique noté $\leq_\Theta$. Soit une triangulation T contenant deux faces adjacentes ABC et ABD telles que $\widehat{ACB} + \widehat{ADB} > \pi$ comme illustrée sur la figure 5. Soit la triangulation T' obtenue en basculant AB sur CD.

Question 2.14.⋆ Montrez en utilisant le théorème du cercle inscrit que $\widehat{BAC} < \widehat{BDC}$, $\widehat{BAD} < \widehat{BCD}$, $\widehat{ABC} < \widehat{ADC}$ et $\widehat{ABD} < \widehat{ACD}$. En déduire

que $\theta(T) <_\Theta \theta(T')$ et donc que l'algorithme de flipping calcule en un nombre fini d'étapes la triangulation $T_{max}(\mathcal{V})$ qui rend maximum $\theta(T)$ selon $\leq_\Theta$. En conclure que $T_{max}(\mathcal{V})$ est forcément la triangulation de Delaunay TD($\mathcal{V}$).

On cherche enfin à établir la complexité de l'algorithme de construction du graphe de Gabriel par filtrage des arêtes de la triangulation de Delaunay. Soit maintenant une arête $\{A, B\}$ d'une triangulation de Delaunay. Cette arête est incidente à deux faces $\{A, C, B\}$ et $\{A, D, B\}$.

Question 2.15.⋆ En utilisant le résultat de la question 13, montrez qu'il faut et qu'il suffit que les sommets C et D n'appartiennent pas au disque ouvert de diamètre (A, B) pour que l'arête $\{A, B\}$ satisfasse le critère de voisinage du graphe de Gabriel.

Question 2.16.⋆ Á l'aide de la question précédente, montrez que l'algorithme de filtrage des arêtes de TD($\mathcal{V}$) a une complexité en $O(n)$ dans le pire cas.

Indice : on remarquera que si f, e et b sont respectivement le nombre de faces internes (i.e. de triangles), d'arêtes et de bords (i.e. arêtes sur le bord extérieur) de la triangulation de Delaunay on a $2 \cdot e = 3 \cdot f + b$ et $n - e + f = 1$ (équation de Descartes-Euler).

Question 2.17.⋆ Les meilleurs algorithmes calculant la triangulation de Delaunay d'un nuage de n points ont une complexité dans le pire des cas de $\Theta(n \cdot \log(n))$. Quelle conclusion peut-on en tirer sur la complexité globale de l'algorithme de construction du graphe de Gabriel ?

TL 3 : algorithme de Dijkstra pour la recherche du plus court chemin

Cette partie porte sur le calcul du trajet optimal à suivre pour réaliser une livraison d'un point A à un point B. Ce problème est résolu efficacement par l'algorithme de Dijkstra.

Questions prioritaires

Étant donné un graphe dans lequel chaque arête présente un coût positif ou nul (distance, temps de parcours, coût de l'essence, etc), l'algorithme de Dijkstra calcule un chemin de coût minimal (i.e. un chemin des plus courts si le coût est la distance) pour aller d'un sommet de départ fixé noté d à tout autre sommet du graphe. L'algorithme se fonde sur un raisonnement par récurrence : rappelons qu'un chemin est une suite finie de sommets voisins. Soit donc $(s_0 = d, s_1, \ldots, s_i = s)$ un des chemins optimaux pour aller de d à un sommet arbitraire $s \neq d$. Un raisonnement par l'absurde nous indique que $(s_0 = d, s_1, \ldots, s_{i-1})$ est nécessairement un des chemins optimaux pour aller de d à s_{i-1}. Les chemins optimaux forment donc un arbre de racine le sommet d dans lequel chaque sommet $s \neq d$ a pour sommet parent l'avant dernier sommet du chemin optimal allant de d à s. L'algorithme de Dijkstra construit itérativement l'arbre T en partant de la racine d et en lui ajoutant à chaque itération le sommet s le plus proche de d qui n'est pas déjà dans T. Lorsque s est rajouté à T, on vérifie pour tout voisin $s' \notin T$ de s qu'il n'est pas moins coûteux de passer par s pour aller de d à s'. Le pseudo-code 1 fournit les détails de l'algorithme.

Question 3.1. Codez dans une méthode `dijkstra(self, depart)` de la classe `Graphe` l'algorithme de Dijkstra. L'algorithme prendra en argument le sommet de départ. L'algorithme produira le résultat par effet de bord en stockant les cumuls et les arêtes parentes dans des attributs de la classe `Sommet`. Testez l'algorithme en cherchant à minimiser le nombre de tronçons à parcourir pour aller de `depart` à tout autre sommet, c'est-à-dire en prenant un coût constant égal à 1 pour chaque arête. Pour représenter le résultat graphiquement on ajoutera une méthode `traceArbreDesChemins(self)` à la classe `Graphe` qui colorie d'une couleur spécifique les arêtes appartenant aux chemins optimaux. On vérifiera que les chemins optimaux forment bien un arbre, c'est-à-dire un graphe sans cycle. On pourra également identifier le sommet de départ à l'aide d'une couleur particulière.

Indications de mise en œuvre :
Pour implémenter l'algorithme, on introduira un attribut `cout` dans la classe `Arete` et des attributs `cumul` et `chemin` dans `Sommet`. L'appel à la méthode

Entrées : graphe g, sommet de départ d, fonction de coût $cout : E(g) \to \mathbb{R}^+$ des arêtes

Sorties : attributs $cumul[\cdot]$ et $chemin[\cdot]$ qui associent à tout sommet v respectivement le cumul des coûts du meilleur chemin reliant d à v et l'arête $\{v, v'\}$ de v reliant v à son sommet parent v'

pour *chaque sommet* s *de* g **faire**
  $cumul[s] \leftarrow +\infty$;
  $chemin[s] \leftarrow null$
fin
$cumul[d] \leftarrow 0$;
Créer une liste L de références sur des sommets ;
Mettre d dans L ;
tant que L *n'est pas vide* **faire**
  Extraire le sommet s de L de plus petit $cumul[s]$;
  pour *chaque arête* e *incidente à* s **faire**
    Soit s' le sommet incident à e et différent de s ;
    si $cumul[s] + cout[e] < cumul[s']$ **alors**
      si $cumul[s'] = +\infty$ **alors**
        ajouter s' dans L
      fin
      $chemin[s'] \leftarrow e$;
      $cumul[s'] \leftarrow cumul[s] + cout[e]$
    fin
  fin
fin

Algorithme 1 : L'algorithme de Dijkstra

`dijkstra` suppose que les attributs `cout` des arêtes aient été préalablement initialisés selon le critère à optimiser. La méthode `dijkstra` ne renvoie pas de valeur de retour. Le résultat de la méthode (i.e. l'arbre des chemins optimaux) sera produit par effet de bord, en modifiant les valeurs des attributs `cumul` et `chemin` de chaque sommet du graphe. Pour représenter une valeur infinie $+\infty$, on utilisera la valeur `sys.float_info.max` du module système `sys` qui correspond à la plus grande valeur possible que peut valoir un nombre de type `float`. On utilisera une liste Python pour représenter L. Pour chercher dans L le sommet de plus petit cumul, il faudra parcourir L pour mettre à jour le plus petit cumul c_{min} déjà trouvé ainsi que l'indice i_{min} associé à c_{min}. La méthode `pop(i)` de `list` permet de supprimer l'élément d'indice `i` et renvoie l'élément supprimé.

Question 3.2. Proposez deux méthodes `fixeCarburantCommeCout(self)` et `fixeTempsCommeCout(self)` qui initialisent respectivement le coût de chaque arête comme la distance et le temps minimum (en respectant les limitations de vitesse) pour la parcourir. Testez en appliquant l'algorithme de Dijkstra à l'une comme à l'autre fonction de coût. Comparez les résultats visuellement pour l'un et l'autre critère (en utilisant l'option `blocage=False` de la fonction `graphique.affiche`).

Question 3.3. Mesurez les temps de calcul de l'algorithme de Dijkstra en fonction du nombre n de sommets. Comparez les résultats obtenus avec la complexité théorique de l'algorithme (le nombre d'arêtes m étant clairement en $\Omega(n) \cap O(n^2)$).

Question 3.4. Afin de préparer la partie suivante, codez une méthode `cheminOptimal(self, arrivee)` de la classe `Graphe` qui retourne la liste des arêtes du chemin optimal qui va du sommet de départ au sommet `arrivee` (ce qui suppose que l'algorithme de Dijkstra ait été préalablement exécuté). Testez en implémentant une seconde méthode `colorieChemin(self, chemin, c)` qui rend visible un chemin (i.e. une liste d'arêtes) à l'aide d'une couleur `c` particulière. On pourra également distinguer le sommet de départ du sommet d'arrivée par deux couleurs différentes.

Indications de mise en œuvre :
Comme il faut construire le chemin en partant de `arrivee` jusqu'au sommet de départ, il serait logique d'insérer chaque nouvelle arête au début de la liste Python (qui est en réalité un tableau dynamique) afin que l'ordre des arêtes dans le vecteur corresponde bien à un parcours allant du départ à l'arrivée. Mais cela entraînerait une complexité en $\Theta(l^2)$ où l est la longueur

du chemin. Une meilleure solution consiste à ajouter chaque arête à la fin de la liste (méthode `append`) puis à inverser l'ordre des éléments à l'aide de la méthode `list.reverse`.

Questions d'approfondissement

On veut rendre encore plus rapide le calcul du plus court chemin entre deux sommets.

3.1.3 Recours à un tas binaire

L'utilisation qui est faite de la liste L est celle d'une *file de priorité* dont les opérations sont l'insertion d'un nouvel élément (en l'occurrence un sommet) et l'extraction de l'élément le plus prioritaire (en l'occurrence le sommet de plus petit cumul). Il existe une structure de donnée appelée *tas binaire* (i.e *heap* en anglais) vue en cours qui confère à ces deux opérations une complexité dans le pire des cas en $O(\log(n))$, n étant le nombre d'éléments dans le tas. Le fichier `tas.py` contient une classe `Tas` qui implémente un tas binaire et qui peut s'adapter à tout type d'élément. La fin du fichier `tas.py` contient un exemple d'utilisation dans lequel des tâches sont ordonnées par ordre chronologique qui est repris et développé dans l'annexe 5.4

Question 3.5. Recopiez l'algorithme de Dijkstra dans une nouvelle méthode `dijkstraAvecTas` et remplacez la liste `L` par un tas. Testez en vérifiant que pour un même graphe, les algorithmes de Dijkstra avec et sans tas donnent bien le même résultat.

Indications de mise en œuvre :
On veillera à passer en argument du constructueur du tas une fonction qui renvoie à partir d'un sommet un niveau de priorité adéquat exprimé à partir du cumul. On pensera à repositionner un sommet dans le tas lorsque son cumul est modifié lors d'une relaxation. Comme la fonction de repositionnement exige en argument la clé qui est associée à l'élément à repositionner, chaque sommet devra stocker dans un attribut la clé renvoyée lors de l'ajout du sommet au tas.

Question 3.6. Comparez les temps de calcul des algorithmes de Dijkstra avec et sans tas en fonction du nombre n de sommets. Quand la version avec tas devient-elle intéressante ? Expliquez. En comparant les résultats expérimentaux avec les complexités théoriques, que peut-on déduire sur la complexité du nombre m d'arêtes en fonction de n pour les graphes de voisinage ?

3.1.4 Prise en compte du sommet d'arrivée

Dans bien des applications, on ne cherche pas à déterminer **tous** les chemins optimaux d'un sommet de départ d vers **tous** les autres sommets mais **un seul** chemin optimal d'un sommet de départ d vers **un seul** sommet d'arrivée a. L'algorithme de Dijkstra proposé précédemment fait alors de l'excès de zèle.

Question 3.7. Adaptez l'algorithme de Dijkstra (avec tas) pour prendre en compte le sommet d'arrivée au sein d'une méthode `dijkstraPartiel(self, depart, arrivee)` de la classe `Graphe`. La méthode retournera le nombre de sommets traités afin de pouvoir apprécier les économies de calcul réalisées. On pourra également mettre en évidence les sommets visités par une couleur spécifique. Testez la méthode en observant le nombre et la répartition spatiale des sommets visités.

3.1.5 L'algorithme A*

L'algorithme A*[3] est une amélioration de l'algorithme de Dijsktra qui réduit encore davantage le nombre de sommets visités pour établir un chemin optimal entre deux sommets d et a. L'algorithme A* repose sur l'existence d'une fonction $\hat{c}$ minorant le coût réel optimal entre deux sommets. Ainsi si on suppose que $c(x, y)$ désigne le coût réel d'un chemin optimal pour aller du sommet x au sommet y, $\hat{c}$ doit vérifier que pour tout sommet s :

$$\hat{c}(s, a) \leq c(s, a)$$

et que par conséquent :

$$c(d, s) + \hat{c}(s, a) \leq c(d, s) + c(s, a)$$

L'algorithme A* est identique à l'algorithme de Dijkstra (avec tas) mais extrait cette fois-ci de la liste L le sommet s qui présente non pas le cumul minimal $c(d, s)$ mais l'estimation minimale $\hat{c}(d, s, a) = c(d, s) + \hat{c}(s, a)$ calculée de manière optimiste pour aller de d en a en passant par s. L'algorithme s'arrête dès que le sommet d'arrivée a est extrait de L.

Question 3.8. Implémentez l'algorithme A* dans la classe `Graphe` à l'aide d'une méthode `astar(self, depart, arrivee)`. Cette méthode retournera le nombre de sommets visités et pourra marquer les sommets visités par une couleur spécifique. Testez la méthode en utilisant l'une ou l'autre

3. Cet algorithme, qu'on prononce « A star » ou « A étoile », est en pratique celui qui est utilisé dans les logiciels de planification de trajets (routiers, ferroviaires, aériens, etc) mais aussi dans certains jeux vidéo, pour guider les agents gérés par l'ordinateur vers un endroit donné.

fonction de coût. Observez la répartition spatiale des sommets visités. Vérifiez expérimentalement que l'algorithme donne la même solution que l'algorithme de Dijkstra. Comparez leur efficacité en terme de sommets visités.

Indications de mise en œuvre :
On réutilisera les attributs `cumul` et `chemin` introduits pour coder l'algorithme de Dijkstra. Afin de faciliter l'implémentation de A* pour différentes fonctions de coût, ajoutez un attribut `minCumulRestant` à la classe `Sommet` destiné à contenir $\hat{c}(s, a)$ où s désigne le sommet courant `self` et a le sommet d'arrivée. On ajoutera ensuite deux méthodes à la classe `Graphe`

— `fixeCarburantCommeCoutPourAstar(self, arrivee)`
— `fixeTempsCommeCoutPourAstar(self, arrivee)`

qui initialiseront selon la fonction de coût sélectionnée non seulement l'attribut `cout` de chaque arête mais aussi l'attribut `minCumulRestant` de chaque sommet. Pour déterminer les valeurs de `minCumulRestant`, on cherchera dans un premier temps une borne inférieure de la distance totale qui reste à parcourir pour aller d'un sommet au sommet d'arrivée. Puis on en déduira une seconde borne qui minore le temps de parcours.

Question 3.9. Comparez les temps de calcul des trois algorithmes Dijkstra sans tas, Dijkstra avec tas et prise en compte de l'arrivée, et A*. Concluez.

Question 3.10.⋆ Montrez que sous réserve que $\hat{c}$ minore la fonction c, A* trouve toujours un chemin optimal pour aller de d à a.

Indice : cela revient à dire que lorsque A* sort le sommet `arrivee` de la file de priorité, tout autre chemin pour aller de d à a a un coût supérieur ou égal à $cumul(a) + \hat{c}(a, a) = cumul(a)$.

TL 4 : Problème du voyageur de commerce

Au lieu de considérer d'emblée le problème complexe de la planification des déplacements des camions d'un transporteur routier (de manière à honorer un carnet de commande au moindre coût), on s'intéresse au problème similaire de la distribution de colis postaux. Ce dernier est en effet plus simple à formaliser : il consiste à l'aide d'un seul camion à remettre des colis à un ensemble de clients dispersés géographiquement en optimisant le temps total de livraison. Il n'y a donc jamais de retrait de marchandise chez le client et on pourra supposer que le camion est toujours assez grand pour contenir tous les colis à livrer. Ce problème connu sous le nom du *problème du voyageur de commerce* est un problème NP-complet. Il est donc du point de vue de la complexité tout aussi difficile à résoudre que le problème d'optimisation des tournées de camions.

Questions prioritaires

4.1.6 Résolution exacte

Le problème du voyageur de commerce considère un marchand qui doit visiter ses clients dans n villes distinctes en parcourant le moins de kilomètres possible, avant de rentrer chez lui. La solution au problème équivaut donc à trouver sur la carte (dont le nombre de villes sera sensiblement supérieur à n) le circuit le plus court parmi tous les circuits de la carte passant par la ville de résidence du marchand et les n villes de ses clients.

En supposant que chaque ville à visiter soit désignée par un indice de 1 à n et que la ville de départ porte le numéro 0 [4], le problème est défini formellement par la donnée d'une matrice positive C de taille $(n+1)\times(n+1)$ telle que chaque coefficient $c_{i,j}$ vaut le coût (temps, distance, etc) du trajet optimal pour aller de la ville i à la ville j. Une « solution candidate » au problème est donc définie par une permutation $p : \{1 \dots n\} \to \{1 \dots n\}$ telle que la suite des villes visitées est $(0, p(1), p(2), \dots, p(n), 0)$. La solution au problème est la permutation p parmi les $n!$ possibles qui minimise $C(p) = c_{0,p(1)} + \sum_{i=1}^{n-1} c_{p(i),p(i+1)} + c_{p(n),0}$.

Question 4.1. Codez une méthode `matriceCout(self, tournee)` dans la classe `Graphe` qui étant donné une liste `tournee` de $n+1$ sommets contenant en première position le sommet de départ puis les clients à livrer, renvoie la matrice de coût M de taille $(n+1) \times (n+1)$ définie plus haut. On testera qualitativement en vérifiant le contenu de la matrice calculée pour un nombre réduit de sommets et en prennant un coût fixe pour chaque arête. Afin d'identifier les sommets sélectionnés sur la carte, on affichera à côté de

4. NB : cet indice ne correspond pas à l'indice de la ville dans la liste des sommets du graphe, la carte ayant bien plus de sommets que les n villes sélectionnées.

chaque sommet le numéro de ligne qui lui correspond dans la matrice de coût.

Indications de mise en œuvre :
On appelera l'algorithme de Dijkstra pour remplir la matrice ligne par ligne. On pourra éventuellement utiliser le fait que la matrice de coût est symétrique pour n'appeler que n fois l'algorithme de Dijkstra. On n'utilisera pas des listes de listes mais des tableaux numériques à deux dimensions du module `numpy`, beaucoup plus rapides en temps d'accès, dont l'utilisation est résumée ci-dessous

```
import numpy
...
# Crée une matrice m x n remplie de zéros.
C = numpy.zeros((m,n))
# Accès à un élément d'une matrice
C[0,0] = 1.
```

Pour l'affichage des numéros des sommets, on ajoutera à la classe `Sommet` un attribut `label` initialisé à `None` destiné à accueillir une chaîne de caractères éventuelle décrivant le sommet (ici le numéro de ligne/colonne du sommet dans la matrice de coût). La méthode `trace` devra être modifiée pour afficher les labels de sommet qui sont différents de `None`. On utilisera à cet effet la méthode `traceTexte(self, pos, texte)` de la classe `Afficheur`.

Le problème du voyageur de commerce est un problème d'optimisation NP-complet dont le résultat attendu est double : il s'agit de retourner à la fois le chemin optimal à suivre et le coût minimal associé. La résolution exacte d'un problème NP-complet nécessite une approche de type « retour sur trace » (ou backtracking) de complexité exponentielle.

Question 4.2. Formalisez ce problème de backtracking comme vu en cours. On précisera ce qu'est un état, une représentation informatique possible de cet état, les extensions possibles d'un état, et la condition d'élagage qui réalise un retour arrière. Une fois ces notions clarifiées, codez l'algorithme de backtracking dans une méthode `voyageurDeCommerceNaif(self, tournee)` qui prend en argument la liste `tournee` des sommets à visiter (le premier sommet étant le sommet de départ) et qui retourne le couple `(minCout, meilleurItineraire)` où `meilleurItineraire` est la permutation de `tournee` qui correspond à la tournée de plus faible coût `minCout`. Testez à l'aide d'une méthode `traceItineraire(self, itineraire)` qui numérote les sommets de `itineraire` selon l'ordre de parcours.

Indications de mise en œuvre :

On implémentera le backtracking de manière récursive à l'aide d'une fonction `backtrack` définie dans la méthode `voyageurDeCommerceNaif`. Afin d'éviter de passer de nombreux arguments à la fonction `backtrack` (ce qui peut ralentir l'exécution du code lorsque les appels de fonction sont très fréquents comme c'est le cas ici) on définira les variables de travail de la fonction `backtrack` (ex : le trajet courant, le meilleur trajet trouvé, la matrice de coût, etc) comme des variables globales définies dans `voyageurDeCommerceNaif`. On utilisera pour cela l'instruction `nonlocal` comme sur l'exemple suivant :

```
classe Graphe:
        def voyageurDeCommerceNaif(self, tournee):
                meilleurItineraire = []
                minCout = sys.float_info.max
                ...
                def backtrack():
                        # nonlocal déclare les variables comme étant
                            celles définies dans la fonction englobante
                        nonlocal minCout, meilleurItineraire, ...

                        # Il s'agit de la variable minCout déclarée dans
                            voyageurDeCommerceNaif
                        minCout = ...
                ...
                # On lance le backtracking
                backtrack()
                return (minCout, meilleurItineraire)
```

On réalisera un retour en arrière dès que le coût du trajet partiel courant `itineraire` en cours de construction est supérieur ou égal au meilleur coût `minCout` trouvé jusqu'alors. Chaque fois qu'un meilleur itinéraire est trouvé, on mettra à jour `minCout` et `meilleurItineraire`. Pour copier la liste `itineraire` dans la liste `meilleurItineraire`, on pourra utiliser la ligne suivante :

```
meilleurItineraire = [ x for x in itineraire]
```

Afin de déterminer rapidement les extensions possibles applicables à `itineraire`, on pourra définir un tableau `visites` de booléens (également déclaré comme variable globale) et s'assurer qu'à tout moment `visites[i]` est vrai si et seulement si le sommet d'indice i est dans `itineraire`. Quand on obtiendra dans `itineraire` une permutation complète des sommets de `tournee`, on n'oubliera pas d'ajouter le coût final pour retourner à la ville de résidence (sommet 0) avant de tester si cet itinéraire constitue le meilleur itinéraire trouvé jusqu'à présent. Dans `traceItineraire`, on pourra en plus de numé-

roter les sommets de l'itinéraire, représenter l'itinéraire complet en coloriant les chemins les plus courts reliant chaque paire de sommets consécutifs de l'itinéraire. Il faudra pour cela rappeler l'algorithme de Dijkstra.

Question 4.3. Evaluez le temps de calcul en fonction du nombre de sommets n à visiter pour une carte donnée. Observez le caractère exponentiel de la complexité et concluez.

Questions d'approfondissement

4.1.7 Résolution approchée

L'approche de type backtracking présente l'énorme inconvénient d'être de complexité exponentielle avec le nombre n de villes à visiter. Cette approche est en pratique inutilisable dès que n est de l'ordre de quelques dizaines. On préfère alors résoudre de façon approchée le problème du voyageur de commerce dans la mesure où l'*algorithme d'approximation* est 1) d'une complexité polynomiale beaucoup plus rapide et 2) où la qualité du résultat approché peut être garantie par un *facteur d'approximation*. Dans le cas du problème du voyageur de commerce, il s'agit de trouver un facteur $\rho \geq 1$ vérifiant que pour tout graphe g, le coût $C(p(g))$ de la tournée $p(g)$ trouvée ne s'écarte pas trop du coût optimal $C^*(g)$ selon l'encadrement :

$$\forall g, C^*(g) \leq C(p(g)) \leq \rho\, C^*(g) \tag{1}$$

De nombreux algorithmes d'approximation ont été proposés pour répondre à différentes hypothèses (graphe planaire ou non, etc). Une des hypothèses de travail les plus courantes suppose que le coût associé à chaque arête du graphe respecte l'inégalité triangulaire. En d'autre terme le coût d'une arête peut être interprétée comme une distance. Sous cette hypothèse l'*algorithme de Christofides*, reposant sur les notions d'arbre recouvrant de poids minimal (vue en cours) et de couplage de poids minimal (non vue en cours) présente un facteur ρ d'approximation de $\frac{3}{2}$. On se propose de développer une version simplifiée moins efficace (dont le facteur ρ est seulement de 2) mais qui ne nécessite pas de recourir à la notion de couplage.

Question 4.4.⋆ Soient n villes d'une carte. On considère le graphe g complet ayant les n villes pour sommets et tel que le coût associé à chaque arête $\{v_1, v_2\}$ soit la distance du trajet le plus court de la carte pour se rendre de v_1 à v_2. Vérifiez que l'hypothèse d'inégalité triangulaire est bien vérifiée pour g. Soit alors T^* un arbre recouvrant de g dont le coût noté $C(T^*)$ est supposé minimal. Soit p^* une tournée optimale dont le coût $C(p^*) = C^*$ est minimal. Montrez que $C(T^*) \leq C^*$. Montrez ensuite qu'il est facile de

concevoir une tournée p à partir de T^* telle que $C(p) = 2\,C(T^*)$.

Question 4.5.⋆ A partir du résultat de la question précédente, concevez théoriquement un algorithme de facteur ρ d'approximation égal à 2. On pourra optimiser la tournée en évitant de se rendre plus d'une fois sur un sommet déjà visité. Comme le graphe g est complet et satisfait l'inégalité triangulaire, on pourra toujours prendre un raccourci en se rendant directement sur le prochain sommet de la tournée non encore visité.

On va maintenant implémenter et évaluer cet algorithme d'approximation.

Question 4.6. Implémentez l'algorithme de Prim vu en cours dans une méthode `Prim(self)` de la classe `Graphe` qui marquera les sommets et les arêtes du graphe appartenant d'un arbre recouvrant de coût minimal (voir indications) et retournera son coût.

Indications de mise en œuvre :
Le poids de chaque arête sera stocké dans l'attribut `cout` de la classe `Arete`. Les arêtes et les sommets déjà ajoutés à l'arbre recouvrant de poids minimal seront repérés à l'aide d'un attribut booléen `selection` dans les classes `Arete` et `Sommet`. On pourra intégrer une structure de tas binaire fournie par le fichier `tas.py` et dont l'utilisation est expliquée dans l'annexe 5.4. On ajoutera également une méthode `colorieSelection(self, couleur)` dans la classe `Graphe` pour colorier les sommets et les arêtes sélectionnées d'une certaine couleur. On vérifiera le bon fonctionnement de l'algorithme en traçant le graphe sélectionné par l'algorithme sur une carte routière.

Question 4.7. Implémentez une méthode `grapheDeCout(self, tournee)` dans la classe `Graphe` qui à partir d'une liste `tournee` de n sommets du graphe `self` crée un nouveau graphe complet (i.e toute paire de sommet est relié par une arête) à n sommets dans lequel 1) les n sommets du nouveau graphe sont en bijection avec les sommets de la tournée du graphe initial, 2) chaque arête entre deux sommets s_1 et s_2 a un coût (attribut `cout`) dont la valeur est égale au coût du trajet optimal reliant les deux sommets de la tournée en bijection avec s_1 et s_2.

Indications de mise en œuvre :
La bijection d'un sommet pourra se faire en ajoutant un attribut `image` à la classe `Sommet`. Cet attribut pointera sur le sommet en bijection avec le

sommet courant. Pour tester, on tracera le nouveau graphe en faisant de telle sorte que la position d'un sommet soit égale à la position de son sommet image par la bijection.

Question 4.8. Implémentez l'algorithme d'approximation dans une méthode `tourneeApproximative(self, tournee)` dans la classe `Graphe`.

Indications de mise en œuvre :
La méthode commencera par créer le graphe de coût, construira ensuite l'arbre recouvrant de poid minimal dans ce graphe, puis identifiera l'ordre de visite des villes en suivant les arêtes de l'arbre. On construira la liste des sommets à visiter de telle façon qu'un sommet apparaisse une et une seule fois dans la liste. On pourra éventuellement utiliser une pile pour mémoriser les indices des arêtes empruntées depuis chaque sommet afin de ne pas parcourir systématiquement les listes d'incidence depuis le début. On tracera la tournée ainsi obtenue. On pourra la comparer avec la tournée optimale calculée par backtracking.

Question 4.9. Mesurez et tracez le temps d'exécution pour l'algorithme en fonction du nombre n de villes à visiter. Comparez avec les temps obtenus pour une résolution exacte. Vérifiez la complexité théorique de l'algorithme. Estimez et tracez le facteur d'approximation $\frac{C(p)}{C^*}$ observé pour chaque valeur de n. Concluez.

4.1.8 Retour sur la résolution exacte

On cherche maintenant à accélérer la résolution exacte à l'aide du principe de « séparation évaluation » (branch and bound en anglais). Dans le cas où on cherche comme ici un coût minimum, ce principe suppose l'existence d'une fonction c_m qui associe à chaque état e (ici un trajet partiel) un minorant des coûts des états e' accessibles à partir de l'état e :

$$\forall e, \forall e', e' \geq_T e \Rightarrow c(e') \geq c_m(e)$$

En conséquence si au cours de la procédure de backtracking le plus petit coût de la meilleure solution déjà trouvé est c_{min} et que l'état courant e vérifie $c_m(e) \geq c_{min}$, les solutions construites à partir de e ne pourront jamais faire mieux que le minimum déjà trouvé. Un retour arrière depuis e peut donc éviter une recherche inutile. L'algorithme de backtracking de la question 2 correspond à un cas particulier de « séparation évaluation ». En effet lorsque le coût c est une fonction croissante dans l'espace d'état (i.e.$\forall e_1, \forall e_2, e_1 \leq_T e_2 \Rightarrow c(e_1) \leq c(e_2)$), comme c'est le cas ici, on peut toujours choisir $c_m = c$.

Dans le cas du voyageur de commerce, on peut proposer une fonction minorante c_m plus efficace, définie comme suit : soient pour toute ville d'indice i, les deux plus petits coûts $c_1(i)$ et $c_2(i)$ dans la liste des coûts observés pour relier le sommet i aux autres sommets $\{0, 1, ..., i-1, i+1, n\}$. Notons aussi l le nombre de sommets déjà ajoutés au trajet p de sorte que $p(l)$ soit le dernier sommet ajouté dans p. On peut alors définir la fonction c_m comme la somme du coût courant $C(p(e))$, de $\frac{c_1(p(l))}{2}$, de la somme des demi-sommes $\frac{c_1(i)+c_2(i)}{2}$ pour l'ensemble des villes i de 1 à n qui ne sont pas déjà dans p et de $\frac{c_1(0)}{2}$:

$$c_m(e) = C(p(e)) + \frac{c_1(p(l))}{2} + \sum_{i \notin p(e)} \frac{c_1(i) + c_2(i)}{2} + \frac{c_1(0)}{2}$$

Question 4.10. Copiez l'algorithme de backtracking dans une nouvelle méthode et améliorez le en y intégrant le principe de « séparation évaluation ».

Indications de mise en œuvre :
On ajoutera deux listes `c1` et `c2` qui seront initialisés respectivement par $\frac{c_1(i)}{2}$ et $\frac{c_2(i)}{2}$. On initialisera la variable `cout` au coût minoré c_m correspondant à un trajet vide puis on actualisera `cout` lorsqu'on se rend d'un sommet i à un sommet j en ajoutant le cout $C(i, j)$ et en retirant les termes minorants correspondant de i et j qui n'ont pluslieu d'être dans c_m.

Question 4.11.⋆ Démontrez le bien fondé de l'algorithme (i.e. que c_m minore bien le coût c).

Indice : étant donné un cycle complet passant par les $n+1$ villes de permutation p, on exprimera $C(p)$ comme

$$C(p) = \frac{1}{2}\left((c_{p(n),0} + c_{0,p(1)}) + (c_{0,p(1)} + c_{p(1),p(2)}) + (c_{p(1),p(2)} + c_{p(2),p(3)}) + \dots + (c_{p(n-1),p(n)} + c_{p(n),0})\right)$$

Annexes

5.1 Programmation de l'interface graphique

Le fichier `graphique.py` qui vous est fourni contient une classe `Afficheur` prête à l'emploi pour créer une fenêtre et y dessiner une carte constituée de formes géométriques élémentaires (disque, ligne, texte, etc). La fenêtre permet en outre de translater la carte à l'aide de la souris et de zoomer sur une zone à l'aide de la molette. Le script `polygone.py` donné à titre d'exemple illustre l'utilisation qui peut être faite de `graphique.py` pour tracer un polygone. Vous pouvez le tester en lançant `python3 polygone.py`.

Pour dessiner un graphe dans un afficheur il suffit de procéder de façon similaire en appelant la fonction `affiche` de cette façon :

```
import graphique  # Importe le fichier graphique.py

g = Graphe(...)

# (x,y) correspond aux coordonnées du point de la carte qui doit se
    retrouver initialement au centre de la fenêtre.
# ech est le facteur d'échelle initial exprimé en nombre de pixels / unité
    de distance de la carte (km)
x = 0.; y = 0.
ech = 10.  # 10 pixels/km

# Affiche le graphe g dans une fenêtre
graphique.affiche(g, (x, y), ech)
```

Il est possible d'afficher plusieurs fenêtres l'une à la suite de l'autre grâce à l'option `blocage` :

```
import graphique

g1 = Graphe(...)
g2 = Graphe(...)

# Affiche g1 dans une première fenêtre
graphique.affiche(g1, (0., 0.), 10., blocage = False)
# Cet appel n'est pas bloquant.

# et g2 dans une seconde,
graphique.affiche(g2, (0., 0.), 10.)
# Cet appel bloque l'exécution jusqu'à fermeture des deux fenêtres
```

Encore faut-il que l'afficheur sache comment dessiner un graphe. Pour ce faire l'afficheur appelle la méthode `def trace(self, afficheur)` supposée être définie dans votre classe `Graphe`. Cette fonction est appelée par l'afficheur dès que le contenu de la fenêtre nécessite d'être redessiné. L'afficheur est passé en argument de la méthode `trace` et met à disposition différentes fonctions de tracé décrites ci-dessous :

```
class Graphe:
    ...
    def trace(self, afficheur):
        # Méthode appelée par l'afficheur pour dessiner le graphe
        # Exemples de méthodes mises à disposition par l'
            afficheur :

        afficheur.tracePoint((x,y)) # Trace un petit disque
            centré sur les coordonnées (x,y)

        afficheur.traceLigne((x1,y1), (x2,y2)) # Trace une
            ligne entre les points de coordonnées (x1,y1) et (x2,y2
            )

        afficheur.traceTexte((x,y), texte) # Ecrit un texte à
            partir des coordonnées (x,y)

        afficheur.changeCouleur((r,g,b)) # Change la couleur
            du tracé par la couleur (r,g,b)
        # r, g et b sont les composantes rouge, verte et bleue,
            chacune comprise entre 0. et 1.
        # Ex: (1.,0.,0.) est rouge, (0.,1.,0.) vert, (.8,.8,0.) jaune,
        # (.5,.5,.5) gris, (1.,1.,1.) blanc, (0.,0.,0.) noir, etc

        afficheur.renomme(titre)   # Renomme le nom de la
            fenêtre de l'afficheur par la chaîne titre.
```

C'est à vous de définir le contenu de cette méthode `trace`. La méthode `trace` de la classe `Polygone` du fichier `polygone.py` fournit un exemple d'utilisation de ces fonctions de tracé.

5.2 Utiliser un générateur aléatoire en Python

Un générateur aléatoire est une fonction f paramétrée par un nombre entier s appelé graîne (ou seed en anglais) qui construit une suite $(u_i)_{i\geq 1}$ par $u_{i+1} = f_s(u_i)$ et $u_0 = 0$. Les nombres u_i pour $i \geq 1$ sont dits pseudo-aléatoires : aléatoires car la fonction f est choisie pour que les nombres u_i ne présentent pas de corrélation évidente entre eux et pseudo car la génération de la suite (u_i) ne dépend que de la valeur de la graine s.

En Python, la fonction `random.random()` du module `random` renvoie un nombre tiré aléatoirement selon la distribution uniforme de l'intervalle $[0, 1[$ selon ce principe. À partir de là, on peut facilement obtenir n'importe quelle distribution continue ou discrète. En particulier une distribution uniforme continue sur $[a, b[$ s'obtient par `a + (b-a) * random.random()` et une distribution discrète sur les entiers compris entre a et b inclus correspond à `int(random.random() * (b - a + 1) + a)`. Si on veut contrôler le choix de la graine, par exemple pour reproduire de façon déterministe une même carte aléatoire, il faut initialiser le générateur aléatoire de Python avec une même graine à l'aide de la fonction `random.seed(graine)`. On peut également restorer l'état du générateur aléatoire dans un état antérieur qu'on avait pris soin de sauvegarder comme sur l'exemple suivant :

```
import random
...
# Sauvegarde l'état du générateur aléatoire
etatGenerateur = random.getstate()

# Affiche 10 nombres aléatoires compris entre 0 et 1
for i in range(10):
        print(random.random())

# Restore l'état du générateur aléatoire
random.setstate(etatGenerateur)

# Affiche les mêmes 10 nombres aléatoires
for i in range(10):
        print(random.random())
```

5.3 Mesurer le temps de calcul d'un algorithme

Le sujet du TL nécessite à plusieurs reprises de mesurer le temps d'exécution $t(n)$ d'un algorithme en fonction d'un paramètre d'entrée noté `n`. L'objectif est d'évaluer empiriquement la complexité temporelle moyenne et de pouvoir ainsi la comparer à la théorie. Python met à disposition les modules `time` pour mesurer le temps courant du système et `timeit` pour mesurer le temps d'exécution d'une fonction. `timeit` présente toutefois certains inconvénients et on préfère utiliser `time` dont la seule fonction à connaître est `time.time()` qui renvoie le temps courant en secondes. Le temps d'exécution d'une fonction test peut donc se mesurer comme la différence des temps juste avant et juste après l'appel à la fonction.

Il faut toutefois faire attention au temps qu'on mesure réellement. Les sorties dans la console à l'aide de `print(...)` prennent beaucoup de temps, de même que l'affichage graphique dans une fenêtre. Pour mesurer précisément l'efficacité d'un algorithme, il est donc nécessaire de supprimer toutes les opérations d'affichage (ou d'utiliser un mode « silencieux » spécial à l'aide d'une variable booléenne). Par ailleurs seul le temps d'exécution de l'algorithme à évaluer doit être mesuré : la génération des entrées de l'algorithme à tester doit se faire en dehors de l'intervalle de temps mesuré. Il est en effet fréquent que le temps nécessaire à la génération des entrées soit plus long que le test lui-même (et ce sera le cas au cours du TL). Enfin il faut réaliser des tests suffisamment longs (au moins de l'ordre du centième de secondes) pour s'affranchir des aléas statistiques et de la résolution limitée de l'horloge. Si un test présente une forte variabilité statistique il peut être nécessaire de le répéter plusieurs fois pour mesurer la moyenne des temps d'exécution.

Il est conseillé d'écrire une fonction `chronometre` qui réalise les tests de façon générique pour éviter de copier-coller le code. Une fonction de test type est donné ci-dessous :

```
def chronometre(fonctionTest, fonctionPreparation, parametres):
    '''Mesure le temps d'exécution fonctionTest pour différentes valeurs d'
        un paramètres'''

    temps = []
    # Pour chaque valeur de paramètre
    for p in parametres:
        # Génère les entrées du test pour la valeur p
        entrees = fonctionPreparation(p)

        # Lance le test pour ces entrées
        print("t({}) = ".format(p), end="", flush=True)
```

```
        debut = time.time()
        fonctionTest(entrees)
        fin = time.time()

        # Mesure le temps d'exécution
        t = (fin - debut)
        print("{:.2f} s".format(t))
        temps.append(t)
    return temps
```

Pour chaque valeur de paramètres, les entrées nécessaires au test sont générées grâce à une fonction `fonctionPreparation`. Le code suivante illustre l'utilisation de `chronometre` en comparant les temps d'exécution de la méthode de tri `sort` et de la fonction de tri `sorted`.

```
import random
...
# Teste la fonction sort
def testeSort(donnees):
    donnees.sort()

# Teste la fonction sorted
def testeSorted(donnees):
    sorted(donnees)

# Prépare un tri en remplissant un tableau de n valeurs aléatoires
def prepareTri(n):
    L = []
    for _ in range(n):
        L.append(random.random())
    return L

# Liste des tailles de tableau à tester
N = [100, 1000, 10000, 100000, 1000000, 10000000]

# Lance les tests
Tsort = chronometre(testeSort, prepareTri, N)
Tsorted = chronometre(testeSorted, prepareTri, N)
```

Pour représenter graphiquement les résultats d'un test, on utilisera le module `matplotlib`. On utilisera une échelle log log pour mesurer facilement la pente asymptotique et ainsi identifier la complexité moyenne d'un algorithme. Le code suivant affiche un graphique permettant de comparer les résultats du test précédent. Les deux courbes se distinguent par la couleur (`'ro-'` affiche

la courbe en rouge alors que `'bo-'` l'affiche en bleu) et par la légende (grâce à l'option `label`).

```
import matplotlib.pyplot as mp
...
# Trace une première courbe log log avec comme liste des abscisses N et
    liste des ordonnées Tsort
mp.loglog(N, Tsort, 'ro-', label = 'sort')
# Trace une seconde courbe
mp.loglog(N, Tsorted, 'bo-', label = 'sorted')
# Donne un nom à l'axe des abscisses
mp.xlabel('n')
# Donne un nom à l'axe des ordonnées
mp.ylabel('temps (s)')
# Donne un titre au graphique
mp.title(u"Temps pour trier un tableau")
# Affiche une légende (en utilisant les options label des courbes)
mp.legend()
# Lance l'affichage
mp.show()
```

5.4 Utiliser un tas binaire

Supposons que l'on veuille maintenir à jour un échéancier (i.e. une liste de tâches à effectuer avant une certaine date, triées par ordre chronologique). Plus exactement on veut stocker les tâches dans une structure de données telle qu'à tout instant on puisse y inscrire de nouvelles tâches et qu'on puisse déterminer la tâche la plus imminente à réaliser en priorité. Eventuellement on peut vouloir également modifier la date d'une tâche déjà inscrite dansl'échéancier.

La structure de tas binaire permet d'assurer une complexité en $O(\log(n))$ pour ces trois opérations. La classe `Tas` du fichier `tas.py` propose une implémentation d'un tas binaire. L'interface de programmation de cette classe est la suivante :

`__init__(self, f)` Constructeur d'un nouveau tas prennant en argument une fonction `f` qui prend en argument un élément à insérer dans le tas et renvoie son niveau de priorité. L'élément en tête du tas sera l'élément de plus grande priorité.

`ajoute(self, valeur)` Ajoute au tas un nouvel élément égal à `valeur`. Renvoie la clé associée au nouvel élément inséré, permettant d'y faire référence lors d'un repositionnement ultérieur de l'élément dans le tas.

`pop(self)` Retire du tas l'élément en tête (i.e. de plus forte priorité) et le renvoie.

`empty(self)` Renvoie vrai si le tas est vide.

`actualise(self, cle)` Actualise la position dans le tas de l'élément associé à la clé `cle` si sa valeur a changé.

Le repositionnement d'un élément dans le tas se fait à partir d'une « clé » associée à l'élément et retournée par la fonction d'insertion. Le code suivant illustre la façon d'utiliser cette classe :

```
import tas

class Tache:
        '''Tache devant être effectuée avant une date limite'''
        def __init__(self, jour, nom):
                self.jour = jour
                self.nom = nom
                # Cet attribut servira à accueillir la clé du tas
                self.cle = None

        def __str__(self):
                return self.nom + " ({})".format(self.jour)
```

```
# Le niveau de priorité d'un événement x est −x.jour :
# on trie donc les événements par ordre chronologique
def calculePriorite(tache):
        return −tache.jour

# On crée un tas dont les éléments sont des événements
# triés par ordre chronologique
T = tas.Tas(calculePriorite)

# Ajoute des événements au tas
evts = [ Tache(5, "T1"), Tache(10, "T2"), Tache(5, "T3"), Tache(3, "T4")]
for evt in evts:
        evt.cle = T.ajoute(evt)

# Supposons que la tache 2 devienne soudain assez urgente
evts[1].jour = 4
T.actualise(evts[1].cle)

# Retirons dans l'ordre chronologie les éléments du tas
while(not T.empty()):
        print(T.pop())
```