INSTITUT NATIONAL DES SCIENCES APPLIQUÉES ROUEN NORMANDIE

Alexis Imbert
Brice Grindel

Recherche opérationel - TP

Balade en ville

RO - ITI - 2022-2023

Table des matières

1 Etape 1 : Modéliser, définir le problème formel, associer une classe de complexité **3**

2 Etape 2 : l'algoritme **4**

3 Etape 3 : L'implémentation **8**

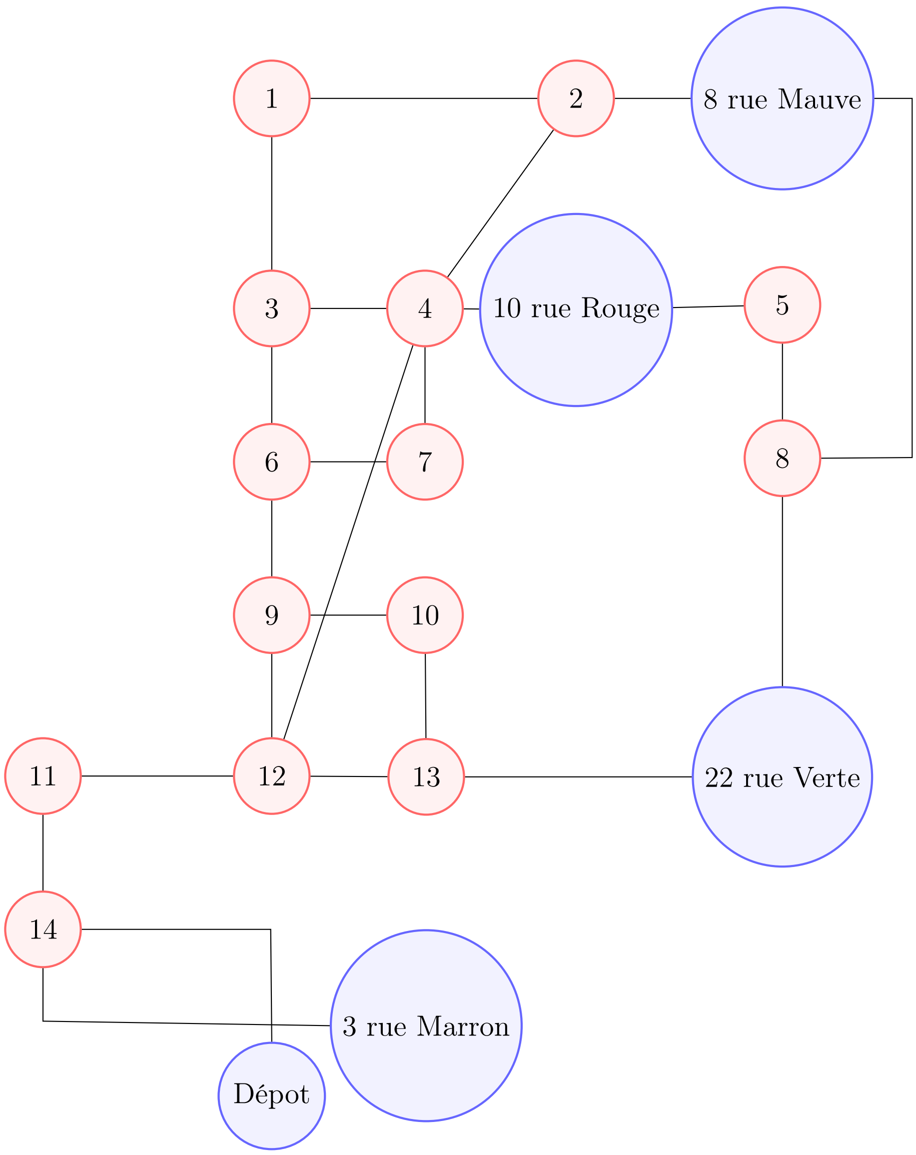
4 Etape 4 : L'adaptation **8**

1 Etape 1 : Modéliser, définir le problème formel, associer une classe de complexité

- — On propose de représenter par le graphe
  - Les noeuds représenteront les intersections, les addresses et les arrêts de métros.
  - Les arrêtes représenteront les portions de route ou entre 2 stations de métros.

  Si le métro est proche de d'un intersection on peut faire l'approximation que c'est le même noeud.

  Représentation sagitale :

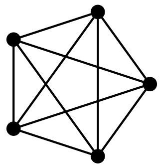


Pour simplifier le graphe : on peut extraire un graphe de distance géodésique entre les adresses tel que :

— Les noeuds représenteront les addresses

— Les arrêtes représenteront les chemins reliant ces addresses.

Représentation sagitale :



— Le graphe est non orienté. Pour le passage d'addresse en paramètre on peut passer les arrêtes sur lesquels sont ces arrêtes.

— On a deux problèmes formel sous jacents.

  — La recherche de plus cours chemin

  — La recherche d'un cycle hamiltonien

  Ce problème peut se ramener au problème du voyageur de commerce.

— Le problème du voyageur de commerce fait parti des problèmes NP-complet. C'est à dire que la résolution de ce type de problème est exponentiel. Toutefois nous ne sommes pas obliger d'abandonner tout de suite car ici le graphe est assez petit : seulement 5 addresses à parcourir.

  Le problème de plus court chemin peut etre résolu par l'algorithme de Dijkstra qui a une résolution polynomiale. Dans notre cas comme la valuation de toutes les arrêtes est égale à 1, cela revient à un parcours en largeur.

2 Etape 2 : l'algoritme

Etat de l'art `https://www.datavis.fr/playing/salesman-problem` Pour l'algorithme de dijktra dans le pseudo code on se base sur celui de M Delestre cours Algorithmique et programmation en C dans le cours sur "Les Graphes" slide 26-27 de la version 2.3.2

Une première solution est le brut force : énumération de tout les chemins possible dans le graphe simplifié et choix du plus cours.

Algorithme On admet posséder les fonctions et procédure suivante :

```
ajouterSommetDansGraphe(E/S: Graphe graphe, E:Sommet sommet) :
    Ajoute un sommet "sommet" au graphe "graphe"

ajouterDansListe(E/S:Liste liste, E:Element element):
    Ajoute l'element "element" dans la liste "liste"
```

```
retirerDansListe(E/S:Liste liste, E: Element element):
    Retire l'element "element" de la liste "liste"

ajouterArcEtiqueté( E/S : Graphe graphe,
                    E:    Sommet source,
                          Sommet destination,
                          Etiquette etiquette)
    Ajoute l'arc d'origine "source" et de destination "destination"
    au graphe "graphe" avec l'etiquette "etiquette"
```

On obtient le pseudo code suivant :

```
programme(Graphe ville, liste<Sommet> POIs)-> Liste<Sommet> :
    Role :
        A partir du graphe "ville" et des points d'intérêt "POIs"
        calcul le trajet optimal respectant les contraintes du sujet

    Déclaration :
        Graphe distances
        Liste<Sommet> destinations
        Liste<Sommet> parcouru
        Dictionnaire<Sommet:distance> distance;
    Début :
        Pour chaque addresse dans POIs :
            ajouterSommetDansGraphe(distances, addresse)
        Fin pour

        Pour chaque addresse dans POIs :
            destination = POIs
            ajouterDansListe(parcouru,addresse)
            Pour chaque sommet dans parcouru :
                retirerDansListe(destination, sommet)
            distance = DijkstraVersPOIs(ville, source, destinations)
            pour chaque sommet dans distance :
                ajouterArcEtiqueté(distances, addresse, sommet, distance[sommet])
        Fin pour

        #Creation de tous les cycles :
        /****************************** A COMPLETER **************************/
    Fin
```

On explicite les autres fonctions et procédures non admise :

```
Fonction DijkstraVersPOIs(Graphe ville,Sommet source, Liste<Sommet> destinations)
```

```
        ->Dictionnaire<Sommet:distance>
    Précondition :
        sommetPresent(ville, source)
        pour tout sommet dans detinations :
            sommetPresent(ville, destination)
    Role :
        Calcul les distances géodésique entre la source et
        toute les destinations "destinations"

    Déclaration :
        Arbre<Sommet>                        arbreRecouvrant
        Dictionnaire<Sommet, ReelPositif>    cout

    Début :
        arbreRecouvrant,cout ← Dijkstra(ville,source)
        retourne cout[sommet dans destinations]
    Fin
```

On suppose posséder les fonctions et procédures suivantes pour Dijktra

```
fonction arbreInitial (s : Sommet) -> Arbre<Sommet>
    qui crée un arbre possédant uniquement le noeud s

fonction arcsEntreArbreEtGraphe (a : Arbre<Sommet>, g :
                                 Graphe<Sommet>,ReelPositif>)
                                 -> Liste<Liste<Sommet>>
    qui retourne la liste des arcs (liste de deux sommets)
    dont le premier sommet appartient à a et le second sommet appartient
    à g et n'appartient pas à a

fonction arcMinimal (arcs : Liste<Liste<Sommet>>, cout : Dictionnaire<Sommet,
                    ReelPositif>)
                    -> Sommet, Sommet, ReelPositif
    précondition
        non estVide(arcs) et quelque soit i dans 1..longueur (arcs)
            estPresent(cout,obtenirElement(obtenirElement(arcs,i),1))
    qui retourne, parmi les arcs, l'arc (sommet source, sommet destination)
    dont le sommet destination est le plus proche (au sens du dictionnaire
    de cout) des sommets de a ainsi que le coût supplémentaire pour l'atteindre

procédure ajouterCommeFils (E/S :Arbre<Sommet> a,
                            E: Sommet sommetPere, sommetFils)
    Précondition :
```

```
    estPresent(a, sommetPere)
qui ajoute un nouveau noeud, à l'arbre a, contenant sommetFils
qui sera fils du noeud contenant sommetPere
```

Code de Dijkstra

```
Fonction Dijkstra(Graphe graphe, Sommet source)-> Arbre<Sommet>,
                                Dictionnaire<Sommet,ReelPositif> :
Précondition :
    sommetPresent(graphe,source)
Role :
    calcule les distance géodésique entre le sommet source
    et tout les autres sommets du graphe
Declaration :
    Arbre<Sommet>                       arbreRecouvrant
    Dictionnaire<Sommet, ReelPositif>   cout
    Liste<Liste<Sommet>>                l
    ReelPositif                         c
    Sommet                              sommetDeA, sommetAAjouter
Début
    arbreRecouvrant ← arbreInitial(s)
    cout ← dictionnaire()
    ajouter(cout,s,0)
    l ← arcsEntreArbreEtGraphe(g,arbreRecouvrant)

    tant que non estVide(l) faire
        sommetDeA,sommetAAjouter,c ← arcMinimal(l,cout)
        ajouter(cout,
        sommetAAjouter,
        obtenirValeur(cout,sommetDeA)+c)
        ajouterCommeFils(arbreRecouvrant,sommetDeA,sommetAAjouter)
        l ← arcsEntreArbreEtGraphe(g,arbreRecouvrant)
    fintantque

    retourner arbreRecouvrant, cout
Fin
```

3 Etape 3 : L'implémentation

4 Etape 4 : L'adaptation

— Dans notre modélisation on peut isoler les arrêtes du métro à part et vérifier si le chemin emprunté lors de la création du graphe simplifié passe par l'une des arrêtes du métro. Dans ce cas on peut affecter une variable booléenne.

— Si le métros tombe en panne il n'y aucun changement dans l'algorithme : on a juste à rentré un nouveau fichier .txt sans les arrêtes liée au métro

— Pour afficher les 2 résultats : on fait tourner l'algorithme sur les 2 fichiers d'entrée : avec et sans le métro.