Recherche de chemin — Algorithme de Dijkstra avec tas de Fibonacci et applications

Corentin Cadiou Lucas Verney

15 décembre 2014

Le but de ce projet était de réaliser une implémentation de l'algorithme de Dijkstra de recherche de chemin en utilisant des tas de Fibonacci.

Cette implémentation a été réalisée dans le langage C et est détaillée dans les Sections 1 et 2. Un "*wrapper*" python3 a été créé pour pouvoir interagir facilement avec l'algorithme, comme détaille dans la Section 3. Enfin, un script Python qui lit une carte OpenStreetMap, le convertit en un format d'entrée correct pour notre implémentation et effectue des recherches dedans a été écrit.

1 Algorithme de Dijkstra

L'algorithme étudié est l'algorithme de Dijkstra. Cet algorithme a une complexité algorithmique de $O(|E| + |V| \log |C|)$ dans le cas d'une implémentation avec une queue de priorité, comme par exemple avec un tas de Fibonacci (ou une queue de Brodal).

Le principe de l'algorithme est de parcourir progressivement tout le graphe. On ajoute initialement tous les nœuds associés à une distance infinie à une file de priorité. La source est attribuée à une distance nulle. On initialise de plus une liste contenant les nœuds déjà visités.

On itère ensuite sur la file de priorité jusqu'à ce qu'elle soit vide. À chaque itération, le nœud u de la file le plus proche de la source est extrait de la file et marqué comme visité. Pour tous ses voisins v non déjà visités, on calcule la distance du chemin allant de la source v en passant par u. S'il est plus faible que la distance de v à la source sans passer par v, on met à jour cette distance et on note que le prédécesseur de v est u.

Une preuve (succincte et partielle) que l'algorithme est correcte est donnée ci-dessous. On peut le prouver en supposant que l'itération marque les nœuds visités avec la distance minimale et que tous les nœuds non visités sont plus loin que la plus faible des distances dans la file de priorité. À l'itération suivante, un nœud voisin du nœud dans la file le plus proche de la source est sélectionné, sa distance est mise à jour. S'il devient alors l'élément le plus proche de la source, il ne peut être atteint en moins de distance depuis les nœuds visités dans la file puisqu'il est plus proche que tous les autres. De même, il n'est pas atteignable

par un nœud non encore visité, par hypothèse. S'il n'est pas le plus proche, soit il existe un chemin plus court l'atteignant et il sera alors trouvé, soit le chemin proposé initialement était le bon et on le saura quand tous les nœuds dans la file seront à une distance supérieure que celle calculée. On a donc conservé l'hypothèse par itération. Comme celle-ci est vraie pour la première itération (la distance la plus courte de la source à ses voisin est la distance entre ceux-ci), elle est vraie pour toutes les itérations. On ax donc trouvé la plus courte distance entre la source et chacun des nœuds du graphe.

On peut implémenter l'algorithme avec une queue supportant les trois opérations suivantes :

1. ajout d'une clé (la distance du nœud du bout à la source), avec décoration (les deux nœuds formant l'arête)

   `insert(from, to, distance)`

2. récupération du minimum et suppression de la file

   `extract_min()`

```
def Dijkstra(Graph, source):              # Source is the startpoint of the algorithm
    dist[source] = 0                      # Initialisation
    for vertex v in Graph:
        if v != source:
            dist[v]     = Infinity        # Unknown distance from source to v
            previous[v] = None            # Predecessor of v

    source.mark = scanned

    # Initialise the queue with the neighbours of the source
    for u in source.neighbours:
        d = dist[source] + length(source, u)
        Q.insert(u, v, d)

    while Q is not empty:                 # The main loop
        u, v, d = Q.extract_min()         # Remove and return best vertex
        if v.mark is not scanned:
            dist[v] = d
            prev[v] = u
            v.mark = scanned

            for w in v.neighbours:
                d = dist[v] + length(v, w)
                Q.insert(v, w, d)
    return previous
```

2 Tas de Fibonacci

Les tas de Fibonacci supportent les mêmes opérations que les tas binomiaux, vus en cours, avec l'avantage que toutes les opérations qui ne nécessitent pas la suppression d'un élément s'exécutent en temps $O(1)$.

Un tas de Fibonacci est un ensemble d'arbres ordonnées, non triés. Chaque nœud x contient un pointeur `x.parent` vers son père, `x.child` vers n'importe lequel de ses fils. Il possède en outre deux pointeurs `x.left` et `x.right` vers ses frères gauche et droite respectivement. Tous les enfants d'un nœud forment ainsi une liste circulaire doublement chaînée. Dans le cas particulier d'un fils unique, on a donc `x.left = x.right = x`.

Grâce à cette structure de listes circulaires doublement chaînées, un nœud peut être supprimé en temps $O(1)$.

Enfin, chaque nœud a un dernier champ, en plus des décorations éventuelles liées à notre utilisation, `x.degre` qui stocke le nombre d'enfants qu'il a.

Le tas en lui-même comporte deux éléments particuliers : un pointeur sur l'élément minimal, `fh.min`, et un pointeur sur n'importe laquelle des racines, `fh.root`. Le pointeur sur l'élément minimal nous permet de trouver le minimum dans le tas en temps $O(1)$. Les racines, comme tous les autres nœuds, sont reliées entre elles dans une liste doublement chaînée.

Les opérations d'insertion et d'extraction du minimum sont implémentées comme décrites en pseudo-code dans Charles E Leiserson, Ronald L Rivest, Clifford Stein, and Thomas H Cormen. *Introduction to algorithms*. The MIT press, 2001.

Une opération d'insertion se fait en temps $O(1)$, Une opération d'extraction du minimum se fait en temps $O(D(n))$ où $D(n)$ est un majorant du degré maximal d'un nœud. D'après la référence précédente, celui-ci est borné par $\log_\varphi(n)$ où n est le nombre total de nœuds dans le tas de Fibonacci, donc l'extraction de l'élément minimal est en $O(\log n)$.

3 Exposition des fonctionnalités en Python

Pour rendre l'interaction avec l'algorithme plus aisée ainsi que pour profiter des fonctionnalités plus haut niveau de Python, non pertinentes d'un point de vue algorithmique (comme la lecture d'un fichier XML), un "*wrapper*" python a été écrit en utilisant `Cython`. Ce "*wrapper*" expose une classe python appelée `Graph` représentant un graphe et qui donne accès au nombre de nœuds (`nb_vertices`), propose une méthode pour ajouter des arêtes (`addEdge`) ainsi que pour afficher le graphe ou les voisins d'un nœud (`get`). La méthode appelle en réalité des fonctions C et s'occupe uniquement de l'interfaçage entre les deux langages.

La classe propose aussi une méthode `dijkstra` qui lance l'algorithme sur le graphe construit. On a donc exposé entièrement tous les outils nécessaires à l'exploitation de l'algorithme. On trouvera le code dans l'Annexe A.

4 Application : recherche de chemin vers les transports publics les plus proches

Une fois le "*wrapper*" écrit, il a été possible d'utiliser les fonctionnalités haut niveau de Python comme la lecture du format XML (pour les cartes OpenStreetMap) et du JSON (pour les requêtes de l'API OpenStreetMap pour trouver les adresses).

Pour cela, le script Python commence par lire le fichier OSM. Il récupère toutes les balises `node` au format :

```
<node id="703217417" lat="48.8445203" lon="2.3439455" ... />
```

et les renumérote entre 0 et $n_{\text{nodes}} - 1$. Ces `node` seront considérés comme les nœuds du graphe. Elles correspondent à des intersections de rues ou des points particuliers (arrêts de bus, monuments, limites d'un bâtiment,).

La deuxième étape consiste à ajouter au graphe les arêtes, c'est-à-dire les routes. Celles-ci sont représentées comme suit :

```
<way id="4217181" visible="true" ...>
 <nd ref="25033471"/> <!-- Intersection rue d'Ulm, rue Érasme -->
 <nd ref="25033484"/>
 <nd ref="25033485"/>
 <tag k="highway" v="residential"/>
 <tag k="name" v="Rue Érasme"/>
</way>
```

Le chemin est formé d'une ligne brisée composée par la succession des nodes. Les routes n'ayant pas de `tag` avec la valeur `k="highway"` ne sont pas considérés (ce sont des bâtiments, des monuments, ...). Nous avons ajouté au graphe chacun des couples successifs de `nodes`, avec comme distance [1] :

$$D = 2R_{\text{earth}} \operatorname{atan}\left(\frac{\sqrt{a}}{\sqrt{1-a}}\right)$$
$$a = \left(\frac{\sin(\Delta_{\text{lat}})}{2}\right)^2 + \cos(\text{lat}_1)\cos(\text{lat}_2)\left(\frac{\sin(\Delta_{\text{lon}})}{2}\right)^2$$

5 Vérification de la complexité de l'algorithme

Nous avons vérifié que l'implémentation de l'algorithme de Dijkstra utilisant une file de priorité naïve s'effectue en temps $O(|U| + |V|^2)$ tandis que l'implémentation avec les tas de Fibonacci s'effectue en temps $O(|E|+|V|\log|V|)$. Pour cela, nous avons généré des graphes d'Erdös-Rényii de taille entre 10 et 10000 arêtes avec $p \sim 0.1$. Les courbes du temps de travail de l'algorithme ont ensuite été tracées. Les résultats montrent qu'effectivement, la non-implémentation des

1. Se référer à l'article Wikipédia correspondant pour plus d'informations sur cette distance : `https://en.wikipedia.org/wiki/Haversine_formula`

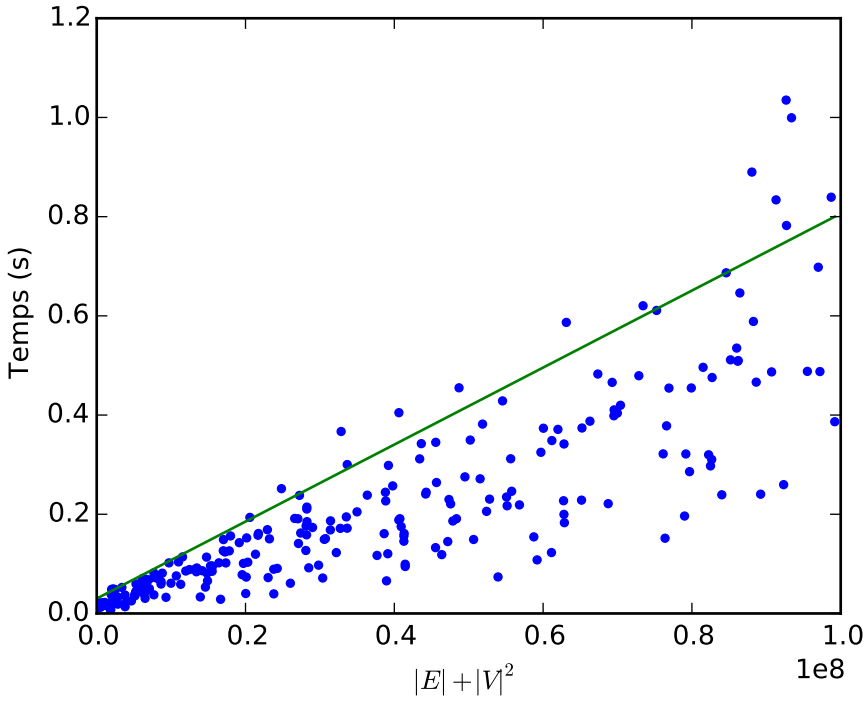

FIGURE 1 – Courbe du temps en fonction de $|E| + |V|^2$ en utilisant un tas simple. On constate que tous les points si situent en dessous de la droite, ce qui correspond bien à la définition d'un temps en $O(|E| + |V|^2)$.

tas de Fibonacci mène à une complexité en $O(|U| + |V|^2)$ comme montré sur la Figure **??**. Pour les tas de Fibonacci, on obtient bien un temps amorti en $O(|E| + |V| \log |V|)$ comme on peut l'observer sur la Figure **??**.

A Code du *wrapper*

Le *wrapper* utilise deux classes, l'une en `Cython` (lignes 7 à 72) qui interagit directement avec le code C et l'autre en Python pur (lignes 75 à 109), qui interagit avec la classe `Cython` (celle-ci ne permettant pas d'utiliser toutes les fonctionnalités de Python). "`from pyjkstra cimport ...`" est une routine `Cython` qui importe le fichier "pyjkstra.pxd" contenant les *headers* dans lesquels trouver les fonctions C appelées.

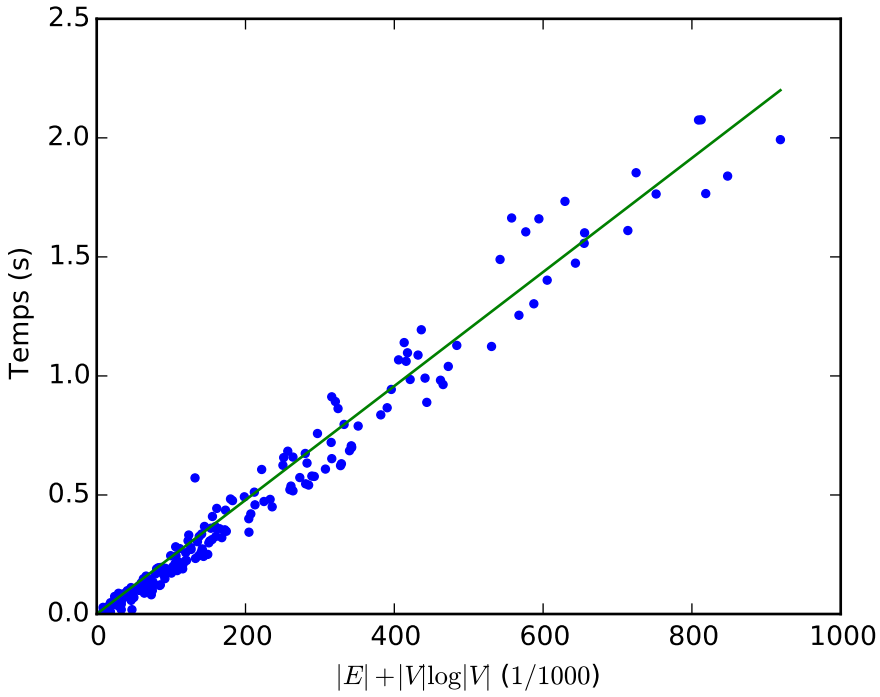

FIGURE 2 – Courbe du temps en fonction de $|E| + |V| \log |V|$ en utilisant les tas de Fibonacci. On constate que tous les points si situent en dessous de la droite, ce qui correspond bien à la définition d'un temps en $O(|E| + |V| \log |V|)$.

```
from libc.stdlib cimport malloc, free
from cpython cimport array
from pyjkstra cimport dijkstra as c_dijkstra
from pyjkstra cimport graph_t, createGraph, freeGraph
from pyjkstra cimport printGraph, printNode, addEdge, INT_MAX

cdef class c_Graph:
    '''Cython class that exposes a graph'''
    cdef graph_t * thisptr

    def __cinit__(self, int n):
        ''' Initialises a C pointer to the graph structure.'''
        self.thisptr = createGraph(n)
        if self.thisptr is NULL:
            raise MemoryError

    def __dealloc__(self):
        ''' Free the malloced memory. '''
        if self.thisptr is not NULL:
            freeGraph(self.thisptr)

    def __str__(self):
        ''' Print a representation of self.'''
        # Bad hack, it prints but returns an empty string ...
        printGraph(self.thisptr)
        return ""

    @property
    def nb_vertices(self):
        return self.thisptr.nb_vertices

    def get(self, int n):
        printNode(self.thisptr, n)

    def addEdge(self, int src, int dest, double weight):
        addEdge(self.thisptr, src, dest, weight)

    def dijkstra (self, int s):
        ''' Converts the Python objects to and from C variables and
        call the algorithm. '''

        cdef int l = self.thisptr.nb_vertices

        # Malloc arrays for return values of the dijkstra algorithm
        cdef int* prev_arg = <int*>malloc(sizeof(int)*l)
        cdef double* dist_arg = <double*>malloc(sizeof(double)*l)
```

```
47
48          # Call the algorithm
49          c_dijkstra(self.thisptr, s, prev_arg, dist_arg)
50
51          prev_out = []
52          dist_out = []
53          # Convert back from C-types to python object
54          for i in range(l):
55              if (prev_arg[i] == INT_MAX):
56                  val = None
57              else:
58                  val = prev_arg[i]
59
60              prev_out.append(val)
61
62
63              if (dist_arg[i] == INT_MAX):
64                  val = None
65              else:
66                  val = dist_arg[i]
67
68              dist_out.append(val)
69
70          # Free C arrays
71          free(dist_arg)
72          free(prev_arg)
73          return (prev_out, dist_out)
74
75  # This is the pure Python class that implements all "methods" of our graph
76  class Graph:
77      ''' A graph represented as an adjacency list.'''
78      c_graph = None
79      def __init__(self, int n):
80          ''' n is the number of vertices.'''
81          self.c_graph = c_Graph(n)
82
83      def __del__(self):
84          del self.c_graph
85
86      def __str__(self):
87          return self.c_graph.__str__()
88
89      @property
90      def nb_vertices(self):
91          return self.c_graph.nb_vertices
92
```

```
93      def addEdge(self, int src, int dest, double weight):
94          ''' Adds an edge to the graph from `src` to `dest` with weight `weight`.'''
95          self.c_graph.addEdge(src, dest, weight)
96
97      def get(self, int n):
98          return self.c_graph.get(n)
99
100     def dijkstra (self, int s):
101         ''' Implement the dijkstra path-finding algorithm.
102         Parameters:
103         G     dijkstra.Graph     a python representation of a graph
104         s     int                the starting point of the algorithm
105
106         Returns:
107         (prev, dist) with prev the antecedent in the path and dist the distance of
108         each node from the start
109         '''
110         return self.c_graph.dijkstra(s)
```