Graphes #2

Pierre Karpman

Lycée Champollion MP2I

https://membres-ljk.imag.fr/Pierre.Karpman/CPGE/2025/

Table des matières

1. Plus-court-chemins non pondérés à source unique (SSSP)

2. Plus-court-chemins avec pondération positive

3. Plus-court-chemins vers but unique

Recherche de plus-court-chemins non pondérés

Objectif

Soit :

- $G = (S, A)$ un graphe (éventuellement orienté)
- u un sommet de G

Pour tout v accessible depuis u, on veut trouver *un* plus-court-chemin non pondéré depuis u (un chemin $u = s_0 \to s_1 \to \cdots \to s_n = v$ minimisant n parmi tous les chemins $u \rightsquigarrow v$)

Variante : APSP

On veut les plus-court-chemins entre toutes paires de sommet

- S'obtient facilement en itérant un SSSP, mais pas la seule façon

SSSP non pondérés par BFS

Théorème

Un parcours en largeur d'origine u visite les sommets accessibles depuis u par distance non pondérée croissante, et les chemins reconstruits post-parcours sont des plus-court-chemins

Esquisse de preuve

On suppose qu'on n'enfile pas un sommet déjà présent dans la file

- Pourquoi cela ne change-t'il rien à l'algorithme ?
- Comment peut-on faire (efficacement) ?

Alors :

- Visite : par les propriétés de parcours
- Par distance non pondérée croissante : par l'invariant : « les distance à u des sommets dans la file sont de la forme $[d, \dots, d]$ ou $[d, \dots, d, (d+1), \dots, (d+1)]$ »
  - Vrai pour l'état (post-)initial de la file $[1, \dots, 1]$
  - Quand on défile un sommet à distance d, ses éventuels voisins enfilés sont à distance $d+1$ (ils n'ont pas été enfilés par des sommets à distance $< d$)
  - Les sommets à distance $d+1$ ne peuvent être enfilés que par des sommets à distance d

Table des matières

1. Plus-court-chemins non pondérés à source unique (SSSP)

2. Plus-court-chemins avec pondération positive

3. Plus-court-chemins vers but unique

Recherche de plus-court-chemins pondérés

Objectif

Soit :

- $G = (S, A, P)$ un graphe (éventuellement orienté) à arcs/arêtes à pondérations positives (par ex. dans $\mathbb{N}$ ou $\mathbb{R}^+$), représentées par $P : A \to \mathbb{N}$ (ou …)
- u un sommet de G

On veut trouver un plus-court-chemin pondéré depuis u vers tout sommet v accessible depuis u (un chemin $u = s_0 \to s_1 \to \cdots \to s_n = v$ minimisant $\sum_{i=0}^{n-1} P(s_i, s_{i+1})$ parmi tous les chemins $u \rightsquigarrow v$)

Remarque

La pondération positive écarte tout risque de cycle de poids négatif, mais certains algorithmes gèrent correctement les pondérations négatives en l'absence de tels cycles (et évt. détectent de tels cycles)

File de priorité

File de priorité

Structure hiérarchique de type *plus grande priorité en premier*

- on associe une *priorité* aux éléments (quand *pushed*)
  - par exemple un entier p
  - on *push* un couple (p, v)
- l'élément qui est *pop* est celui possédant la plus grande priorité parmi ceux encore présents
  - pour priorités entières, généralement celui de p minimum (file de priorité *min*) ou maximum (file de priorité *max*)
  - lors du *pop* on récupère le couple (p, v) entier (y compris avec la priorité associée)

Réalisation inefficace

Par une `list` de couples *(valeur, priorité)*, et recherche linéaire pour *pop*

Réalisations efficaces 🙈

Par exemple avec des *tas*

File de priorité efficace en Python, en boîte noire 

Le module *queue* implémente efficacement (?) une file de priorité (par défaut « min »)

```
from queue import PriorityQueue
```

- `def make(): return PriorityQueue()`
- `def isEmpty(pq): return pq.empty()`
- `def push(p, v, pq): pq.put((p, v))`
- `def pop(pq): return pq.get_nowait()`

Exemple

```
In [11]: pq = make()
In [12]: push(1, "coucou", pq) ; push(3, "salut", pq) ;
         push(2, "hullo", pq)
In [13]: pop(pq)
Out[13]: (1, 'coucou')
In [14]: pop(pq)
Out[14]: (2, 'hullo')
In [15]: pop(pq)
Out[15]: (3, 'salut')
```

Algorithme « de Dijkstra »

Algorithme « de Dijkstra » : WFS avec une file de priorité *min*

Résout le SSSP pondéré pour des graphes représentés par listes d'adjacence, à pondération positive (et si implémenté d'une certaine façon, à pondération générale en l'absence de cycles de pondération négative 🙈)

Implémentation en Python (exemple)

```
def dfp(G, u):
  D = {} # (id : préd., distance)
  pq = make() # file de priorité min
  push(0, (u, None), pq) # distance à u, (sommet, prédécesseur)
  while not isEmpty(pq):
    d, (v, pred) = pop(pq) # le plus proche dans la f.p.
    if v not in D: # pas encore visité
      D[v] = pred, d # on a visité v depuis pred,
                     # distance (u,v) = d
      for w, wd in G[v]: # pour tous les voisins de v
        if w not in D:
          push(d + wd, (w, v), pq) # on *peut* visiter w depuis v
                                   # à distance d + wd ;
                                   # c'est un majorant de la distance
        # else : déjà vu (avec un chemin plus court)
  return D
```

Exemple

Correction : esquisse de preuve

Lemme

Les sommets sont visités par distance à u croissante, avec une distance égale à celle d'un plus-court-chemin vers u

Esquisse de preuve

- Vrai pour le premier sommet inséré (*viz.* u, distance 0 à lui-même, minimal)
- Quand on visite un sommet v avec (implicitement) un certain chemin (donné en remontant les prédécesseurs) et une certaine distance, tout autre chemin vers v ou vers un autre sommet pas encore visité doit passer par un arc encore dans la f.p., donc moins prioritaire, donc induisant un chemin plus long (évt. égal)

Et puisque DFP est un parcours depuis u :

Corollaire

Les chemins reconstruits post-parcours sont des plus-court-chemins pondérés depuis u vers tous les sommets accessibles

Analyse de coût

- Coût d'un parcours depuis u
- Mais qui utilise une file de priorité…
  - $O(A_u \times A_u)$ pour une f.p. très naïve
  - $O(A_u \times S_u)$ pour une f.p. naïve, avec baisse de priorité (on ne garde qu'une fois chaque sommet)
  - $O(A_u \times \log S)$ en utilisant une f.p. correcte (par ex. un tas)

Table des matières

1. Plus-court-chemins non pondérés à source unique (SSSP)

2. Plus-court-chemins avec pondération positive

3. Plus-court-chemins vers but unique

Nouvel objectif

Objectif

Soit :

- $G = (S, A, P)$ un graphe (éventuellement orienté) à arcs/arêtes à pondérations positives (par ex. dans $\mathbb{N}$ ou $\mathbb{R}^+$), représentées par $P : A \to \mathbb{N}$ (ou …)
- u un sommet « départ » de G
- v un sommet « destination » de G

On veut trouver un plus-court-chemin pondéré de u vers v, s'il existe un chemin

Remarque

Ce problème se résout en utilisant l'algorithme de Dijkstra (ou même un parcours en largeur en l'absence de pondération), mais c'est possiblement *overkill* : les PCC vers les sommets autres que *v ne nous intéressent pas*

Problématique sur un dessin

Métaphore par les diagrammes de rayonnement (???)

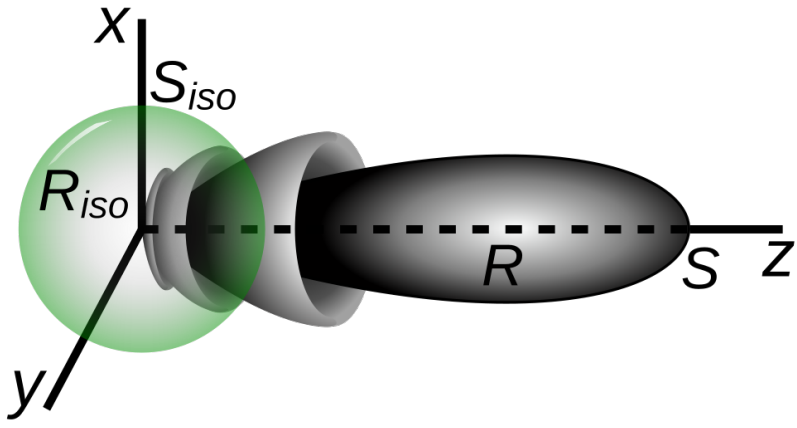

Figure:
https://commons.wikimedia.org/wiki/File:Antenna_directive_gain_diagram.svg

Algorithme A*

Idée

- (Si aucune idée de la bonne direction : utiliser « Dijkstra »)
- Sinon, on veut influencer les prochains nœuds visités en fonction de ce qu'on « pense » être la bonne direction
- C'est à dire les traiter avec une plus grande priorité
  - Traiter les sommets par ordre croissant de la longueur du plus court chemin (u, v) passant par eux

Heuristique

Une *heuristique* (**dans le contexte de A***) est une estimation $h(w, v)$ de la distance $\delta(w, v)$ entre w et v

Algorithme A*

- On fixe une heuristique h
- On applique « Dijkstra », en ajoutant un terme $h(\cdot, v)$ à chaque priorité
  - Si l'on devait insérer (d_w, w) dans la file (avec d_w un majorant de $\delta(u, w)$), on insère en fait $(d_w + h(w, v), w)$ (avec $d_w + h(w, v)$ une estimation de la longueur d'un chemin $u \to v$ passant par w)

Implémentation en Python (exemple **pour une heuristique monotone**)

```
def astar(G, u, v, h):
  D = {} # (id : préd., distance)
  pq = make() # file de priorité min
  push(h(u,v), (0, u, None), pq) # distance estimée à v,
                                 # (distance à u, sommet, prédécesseur)
  while not isEmpty(pq):
    _, (d, w, pred) = pop(pq)
    if w not in D:
      D[w] = pred, d
      if w == v:
        return D
      for nw, nwd in G[w]:
        if nw not in D:
          nd = d + nwd
          push(nd + h(nw, v), (nd, nw, w), pq)
  return D
```

Heuristique : quels critères ?

Heuristique admissible

Une heuristique est *admissible* si elle ne surestime jamais la distance : $h(w, v) \leq \delta(w, v)$ pour tout w. Elle est de plus *monotone* si pour tout arc (w, u), $h(w, v) \leq P(w, u) + h(u, v)$

Théorème (informel, admis)

A* avec une heuristique admissible renvoie un plus-court-chemin, s'il en existe un. De plus si l'heuristique est monotone, son coût (hors calcul d'heuristique) est majoré par celui de l'algorithme de Dijkstra

Remarques

- h identiquement nulle est toujours admissible dans un graphe à pondération positive ; A* avec cette heuristique correspond à « Dijkstra » (qui est bien correct)
- on peut relaxer la condition de minoration à un facteur près ; dans ce cas on obtient un chemin le plus court à un facteur près (on obtient un *algorithme d'approximation*)

Heuristique : quels choix ?

Choix de l'heuristique

- Guidé par le problème modélisé par le graphe
- Pas forcément « compliqué », par ex. la distance à vol d'oiseau pour un graphe de déplacement sur Terre

Conclusion A*

Comparaison avec « Dijkstra »

Avec une heuristique admissible et monotone :

- Coût : au plus pareil (possiblement *nettement moins*)
- Correct : oui
- Pas d'inconvénient à être utilisé (sauf si le calcul de l'heuristique coûte *cher*, ou si l'algorithme ne résout pas le problème voulu…)

Avec une heuristique admissible non monotone :

- Correct : oui
- Coût : possiblement (beaucoup (exponentiellement)) plus que Dijkstra

Avec une heuristique non admissible (forcément non monotone) :

- Correct : pas forcément

En animations

- https://en.wikipedia.org/wiki/User:Subh83/CommonsContrib
- https://www.redblobgames.com/pathfinding/a-star/introduction.html